یادگیری تقویتی عمیق
با پایتون

ترجمه:
مهدی غضنفری
فائزه عسکری

زمستان ۱۴۰۴

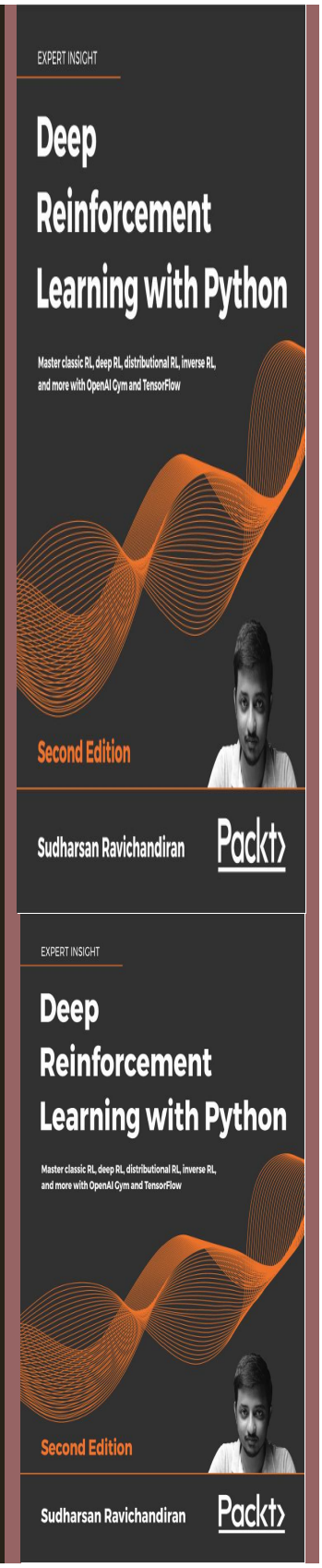
Deep Reinforcement Learning with Python

Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow

Sudharsan Ravichandiran

Second Edition

2020



یادگیری تقویتی عمیق با پایتون

(فصل اول ۱۰)

ترجمه:

مهدی غضنفری

فائزه عسکری

۱۴۰۴

ترجمه چاپ دوم کتاب:

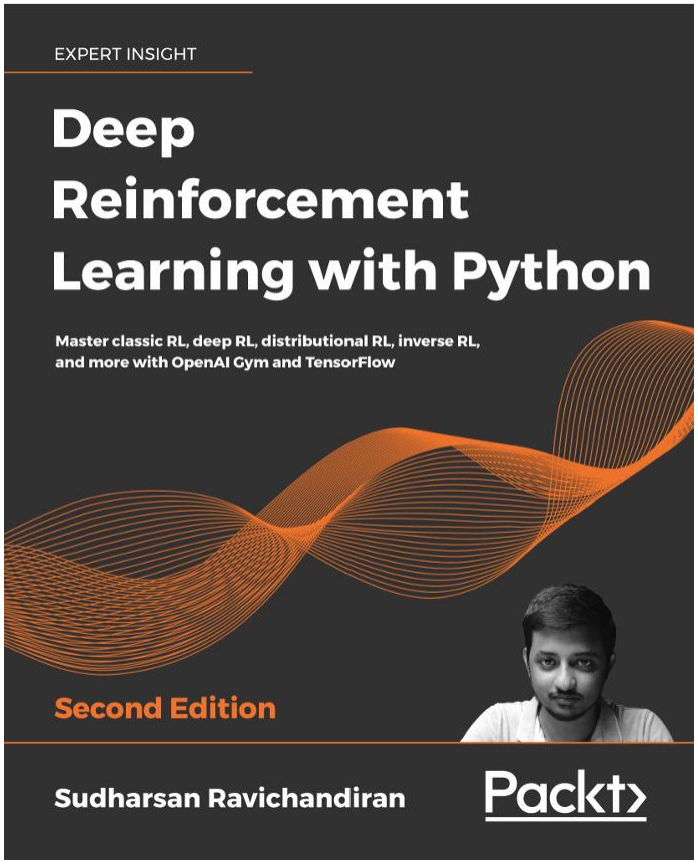
Deep Reinforcement Learning with Python

اثر:

Sudharsan Ravichandiran

دربرگیرنده مفاهیمی همچون:

Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow



مقدمه مترجمین

ن وَ الْقَلَمِ وَ ما یَسْطُرُونَ

در سال‌های اخیر، پیشرفت‌های چشمگیر در حوزه هوش مصنوعی، به‌ویژه در شاخه یادگیری ماشین، افق‌های تازه‌ای را پیش روی پژوهشگران و صنعت گشوده است. در این میان، یادگیری تقویتی (**RL** – Reinforcement Learning) به‌عنوان یکی از پویاترین و تأثیرگذارترین رویکردها، نقش محوری در توسعه عامل‌های هوشمند ایفا کرده است؛ عامل‌هایی که قادرند از طریق تعامل مستمر با محیط، راهبردهای بهینه را بیاموزند و در مسائل پیچیده تصمیم‌گیری عملکردی فراتر از روش‌های سنتی ارائه دهند. موفقیت‌های چشمگیر سامانه‌هایی نظیر DeepMind در بازی‌های پیچیده و کاربردهای صنعتی، نشان‌دهنده بلوغ تدریجی این حوزه و اهمیت روزافزون آن در پژوهش‌های معاصر است.

کتاب «یادگیری تقویتی عملی با پایتون» با هدف ارائه راهنمایی جامع، ساخت‌یافته و در عین حال کاربردی برای یادگیری و پیاده‌سازی الگوریتم‌های مدرن **RL** نگاشته شده است. این اثر تلاش می‌کند شکاف میان مبانی نظری و پیاده‌سازی عملی را پر کند؛ به‌گونه‌ای که خواننده نه تنها با مفاهیم ریاضی و چارچوب‌های نظری آشنا شود، بلکه بتواند آن‌ها را در قالب کدهای اجرایی و پروژه‌های واقعی به کار گیرد. بهره‌گیری از چارچوب قدرتمند TensorFlow 2 و محیط‌های استاندارد آزمایش مانند OpenAI Gym، امکان تجربه عملی و تکرارپذیر الگوریتم‌ها را برای مخاطب فراهم می‌سازد.

در بخش‌های ابتدایی کتاب، مبانی نظری یادگیری تقویتی با دقت و انسجام معرفی می‌شوند. مفاهیمی همچون فرآیند تصمیم‌گیری مارکوف (**MDP**)، معادله بلمن، سیاست و تابع ارزش، چارچوب ریاضی لازم برای درک ساختار مسائل **RL** را فراهم می‌کنند. همچنین برنامه‌ریزی پویا و روش‌های مبتنی بر تکرار مقدار و تکرار سیاست، به‌عنوان سنگ‌بنای توسعه الگوریتم‌های پیشرفته‌تر، به‌صورت تحلیلی و همراه با مثال‌های عددی تشریح شده‌اند. این بخش‌ها پایه‌ای محکم برای ورود به مباحث یادگیری تقویتی عمیق ایجاد می‌کنند.

در ادامه، کتاب طیف کاملی از روش‌های یادگیری تقویتی را در سه خانواده اصلی بررسی می‌کند: روش‌های مبتنی بر ارزش، روش‌های مبتنی بر سیاست و معماری‌های بازیگر–منتقد. الگوریتم‌های شاخص این حوزه، از جمله **DQN**، **TRPO**، **PPO**، **ACKTR**، **DDPG**، **TD3** و **SAC**، نه‌تنها از منظر شهودی معرفی شده‌اند، بلکه ریاضیات زیربنایی آن‌ها—شامل تقریب تابع، گرادیان سیاست، تخمین مزیت و پایدارسازی آموزش—به‌صورت نظام‌مند تحلیل شده است. هر الگوریتم با پیاده‌سازی گام‌به‌گام در پایتون و با تکیه بر TensorFlow 2 ارائه می‌شود تا خواننده بتواند مفاهیم انتزاعی را در قالب کد اجرایی مشاهده و آزمایش کند.

از ویژگی‌های متمایز این کتاب، توجه ویژه به روندهای نوین پژوهشی در **RL** است. فصل‌هایی مستقل به یادگیری تقویتی توزیعی، یادگیری تقلیدی، یادگیری تقویتی معکوس و یادگیری تقویتی فراشناختی اختصاص یافته‌اند؛ حوزه‌هایی که در سال‌های اخیر به‌عنوان مسیرهای امیدبخش برای افزایش کارایی، تعمیم‌پذیری و نمونه‌کارایی عامل‌های یادگیرنده مطرح شده‌اند. در این مباحث، خواننده با چالش‌های مقیاس‌پذیری، اکتشاف مؤثر، انتقال دانش و یادگیری در محیط‌های پیچیده و پیوسته آشنا می‌شود.

علاوه بر توسعه پیاده‌سازی از صفر، این کتاب نحوه استفاده عملی از کتابخانه—Stable Baselines نسخه بهبودیافته‌ای از ابزارهای پایه ارائه‌شده توسط—OpenAI را آموزش می‌دهد. این رویکرد به مخاطب امکان می‌دهد تا بدون درگیر شدن با جزئیات سطح پایین، الگوریتم‌های پیشرفته را در محیط‌های متنوع آزمایش کرده و بر تحلیل نتایج و تنظیم ابرپارامترها تمرکز کند. بدین ترتیب، کتاب هم برای پژوهشگرانی که به دنبال درک عمیق ساختار الگوریتم‌ها هستند و هم برای توسعه‌دهندگانی که به پیاده‌سازی سریع و کاربردی نیاز دارند، ارزشمند خواهد بود.

در فصل‌های پایانی، کتاب به رویکردهای نوظهور مانند یادگیری فراشناختی و عامل‌های تقویت‌شده با تصور و خیال می‌پردازد؛ رویکردهایی که می‌کوشند با بهره‌گیری از مدل‌سازی درونی محیط و توانایی پیش‌بینی، فرآیند تصمیم‌گیری را کارآمدتر و هوشمندانه‌تر سازند. این مباحث نشان می‌دهند که آینده یادگیری تقویتی تنها در افزایش قدرت محاسباتی خلاصه نمی‌شود، بلکه در ترکیب مدل‌محوری، تعمیم‌پذیری و سازگاری سریع با محیط‌های جدید معنا می‌یابد.

این کتاب برای دانشجویان تحصیلات تکمیلی، پژوهشگران هوش مصنوعی، مهندسان یادگیری ماشین و تمامی علاقه‌مندانی که قصد دارند از سطح مفاهیم پایه فراتر رفته و به درک عملی و پیشرفته **RL** دست یابند، طراحی شده است. انتظار می‌رود خواننده آشنایی مقدماتی با پایتون، جبر خطی و مفاهیم پایه یادگیری ماشین داشته باشد.

«یادگیری تقویتی عملی با پایتون» تلاشی است برای ارائه تصویری یکپارچه از نظریه و عمل در یکی از پیشروترین شاخه‌های هوش مصنوعی؛ مسیری که از معادله بلمن آغاز می‌شود و به طراحی عامل‌های هوشمندی ختم می‌گردد که قادرند در محیط‌های پیچیده، پویا و نامطمئن تصمیم‌هایی بهینه اتخاذ کنند.

مهدی غضنفری– فائزه عسکری

۱۴۰۴

پیش‌گفتار مولف

با افزایش قابل توجه در کیفیت و کمیت الگوریتم‌ها در سال‌های اخیر، کتاب "یادگیری تقویتی عملی با پایتون" یک راهنمای غنی از مثال‌ها برای یادگیری الگوریتم‌های پیشرفته یادگیری تقویتی (**RL**) و یادگیری تقویتی عمیق با استفاده از TensorFlow 2و ابزار OpenAI Gym است. علاوه بر بررسی مبانی **RL** و مفاهیم پایه‌ای مانند معادله بلمن، فرآیندهای تصمیم‌گیری مارکوف و برنامه‌ریزی پویا، این کتاب به طور عمیق به طیف کامل روش‌های **RL** مبتنی بر ارزش، مبتنی بر سیاست و بازیگر–منتقد می‌پردازد. در این کتاب الگوریتم‌های پیشرفته‌ای مانند **DQN**، **TRPO**، **PPO** و **ACKTR**، **DDPG**، **TD3** و **SAC** به طور کامل بررسی شده‌اند، ریاضیات پنهان را توضیح می‌دهند و پیاده‌سازی‌ها را از طریق مثال‌های ساده کد نمایش می‌دهند. چندین فصل جدید به تکنیک‌های جدید **RL** اختصاص یافته‌اند، از جمله **RL** توزیعی، یادگیری تقلیدی، یادگیری تقویتی معکوس و یادگیری تقویتی فراشناختی. شما خواهید آموخت که چگونه ازStable Baselines ، بهبود یافته کتابخانه پایه OpenAI، برای پیاده‌سازی بدون زحمت الگوریتم‌های **RL** محبوب استفاده کنید. این کتاب با مرور رویکردهای امیدوارکننده‌ای مانند یادگیری فراشناختی و عوامل تقویت‌شده با تصور به پایان می‌رسد.

این کتاب برای چه کسانی است؟

اگر شما یک توسعه‌دهنده یادگیری ماشینی هستید که تجربه کمی یا هیچ تجربه‌ای با شبکه‌های عصبی ندارید، به هوش مصنوعی علاقه‌مند هستید و می‌خواهید یادگیری تقویتی را از صفر بیاموزید، این کتاب برای شما مناسب است. آشنایی پایه‌ای با جبر خطی، حساب دیفرانسیل و انتگرال و پایتون لازم است. تجربه‌ای جزئی با TensorFlow یک مزیت محسوب می‌شود.

موضوعاتی که این کتاب پوشش می‌دهد

فصل اول، با عنوان **اصول یادگیری تقویتی**، به شما کمک می‌کند تا پایه‌ای قوی در مفاهیم **RL** بسازید. ما درباره عناصر کلیدی **RL** ، فرآیند تصمیم‌گیری مارکوف و چند مفهوم بنیادی مهم مانند فضای عمل‌ها، سیاست‌ها، پردینه‌ها، تابع ارزش و تابع **Q** خواهیم آموخت. در پایان فصل، با برخی کاربردهای جالب **RL** آشنا می‌شویم و همچنین به بررسی اصطلاحات و کلیدواژه‌های پرکاربرد در **RL** می‌پردازیم.

فصل دوم، با عنوان **راهنمای ابزار Gym**، یک راهنمای کامل برای ابزار Gym شرکت OpenAI ارائه می‌دهد. ما با پیاده‌سازی محیط‌های جالب ارائه شده توسط Gym، آن‌ها را به طور کامل درک خواهیم کرد. سفر عملی یادگیری تقویتی خود را از این فصل آغاز می‌کنیم و با استفاده از Gym چند مفهوم بنیادی **RL** را پیاده‌سازی می‌کنیم.

فصل سوم، با عنوان **معادله بلمن و برنامه‌نویسی پویا**، به ما کمک می‌کند تا معادله بلمن را به همراه ریاضیات گسترده به طور کامل درک کنیم. سپس دو الگوریتم کلاسیک **RL** به نام روش‌های تکرار ارزش و تکرار سیاست را یاد می‌گیریم که می‌توانیم از آن‌ها برای یافتن سیاست بهینه استفاده کنیم. همچنین می‌بینیم چگونه روش‌های تکرار ارزش و تکرار سیاست را برای حل مسئله دریاچه یخزده یا Frozen Lake پیاده‌سازی کنیم.

فصل چهارم، با عنوان **روش‌های مونت‌کارلو**، روش بدون مدل مونت‌کارلو را توضیح می‌دهد. ما یاد می‌گیریم که وظایف پیش‌بینی و کنترل چیستند و سپس به روش‌های پیش‌بینی و کنترل مونت‌کارلو به طور جزئی می‌پردازیم. سپس روش مونت‌کارلو را برای حل بازی بلک‌جک با استفاده از ابزار Gym پیاده‌سازی می‌کنیم.

فصل پنجم، با عنوان **درک یادگیری تفاضل زمانی**، به یکی از محبوب‌ترین و پرکاربردترین روش‌های بدون مدل به نام یادگیری تفاضل زمانی **(TD)** می‌پردازد. ابتدا یاد می‌گیریم که روش پیش‌بینی **TD** چگونه کار می‌کند و سپس روش کنترل **TD** درون‌سیاستی (عیان–سیاست) به نام **SARSA** و روش کنترل **TD** برون‌سیاستی (نهان–سیاست) به نام **Q learning** را به طور کامل بررسی می‌کنیم. همچنین روش‌های کنترل **TD** را برای حل مسئله دریاچه یخزده با استفاده از Gym پیاده‌سازی می‌کنیم.

فصل ششم، با عنوان **مطالعه موردی – مسئله MAB**، یکی از مسائل کلاسیک در **RL** به نام مسئله چندبازویی **(MAB)** را توضیح می‌دهد. فصل را با درک مسئله **MAB** آغاز می‌کنیم و سپس درباره چند استراتژی اکتشافی مانند epsilon-greedy، اکتشاف softmax، حد اطمینان بالا و روش نمونه‌برداری Thompson برای حل مسئله **MAB** به طور کامل یاد می‌گیریم.

فصل هفتم، با عنوان **مبانی یادگیری عمیق**، به ما کمک می‌کند تا پایه‌ای قوی در یادگیری عمیق بسازیم. فصل را با درک عملکرد شبکه‌های عصبی مصنوعی آغاز می‌کنیم و سپس چند الگوریتم جالب یادگیری عمیق مانند شبکه‌های عصبی بازگشتی، شبکه‌های **LSTM**، شبکه‌های عصبی کانولوشنی و شبکه‌های مولد تقابلی را یاد می‌گیریم.

فصل هشتم، با عنوان **مقدمه‌ای بر TensorFlow**، به یکی از محبوب ترین کتابخانه های یادگیری عمیق به نام TensorFlow می پردازد. ما با پیاده سازی یک شبکه عصبی برای شناسایی ارقام دست نویس، نحوه استفاده از TensorFlowرا درک خواهیم کرد. در مرحله بعد، یاد می گیریم چندین عملیات ریاضی را با استفاده از TensorFlow انجام دهیم. بعدا در باره TensorFlow 2.0 خواهیم آموخت و خواهیم دید چگونه با نسخه های قبلی TensorFlow تفاوت دارد.

فصل ۹، با عنوان **شبکه Q عمیق و گونه‌های آن**، به ما امکان می دهد سفر یادگیری تقویتی عمیق خود را آغاز کنیم. ما درباره یکی از محبوب ترین الگوریتمهای یادگیری تقویتی عمیق به نام شبکه **Q** عمیق (**DQN**) آشنا خواهیم شد. ما گام به گام نحوه کار **DQN** را همراه با محاسبات گسترده درک خواهیم کرد. ما همچنین یک **DQN** برای اجرای بازی های آتاری پیاده سازی خواهیم کرد. در ادامه، چند نوع جالب از **DQN** را بررسی خواهیم کرد که به آن ها **Dueling**، **Double DQN**، **DQN**، **DQN** با بازپخش تجربه اولویت دار و **DRQN** می پردازیم.

فصل ۱۰، با عنوان **روش گرادیان سیاست**، روش های گرادیان سیاست را پوشش می دهد. ما نحوه عملکرد روش گرادیان سیاست را همراه با استخراج دقیق درک خواهیم کرد. در ادامه، چندین روش کاهش واریانس مانند گرادیان سیاست با پاداش به مصرف و گرادیان سیاست با خط پایه را خواهیم آموخت. همچنین یاد خواهیم گرفت که چگونه یک عامل را برای وظیفه تعادل چرخ دستی با استفاده از گرادیان سیاست آموزش دهیم.

فصل ۱۱، با عنوان **روشهای بازیگر–منتقد A2C** و **A3C**، به چندین روش جالب منتقد بازیگر مانند بازیگر–منتقد مزیت و منتقد بازیگر مزیت ناهمزمان می پردازد. ما به طور مفصل یاد می گیریم که این روش های بازیگر–منتقد چگونه کار می‌کنند و سپس آنها را برای یک وظیفه کوهنوردی با استفاده از OpenAI Gym پیاده سازی خواهیم کرد.

فصل ۱۲، با عنوان **یادگیری DDPG**، **TD3** و **SAC**، الگوریتم های پیشرفته یادگیری تقویتی عمیق مانند گرادیان سیاستی عمیق، **DDPG** تأخیری دوگانه و بازیگر نرم را همراه با استخراج گام به گام پوشش می دهد. همچنین یاد خواهیم گرفت چگونه الگوریتم **DDPG** را برای انجام وظیفه تاب خوردن معکوس پاندول با استفاده از Gym پیاده سازی کنیم.

فصل ۱۳، با عنوان **روشهای TRPO**، **PPO** و **ACKTR**، به چندین روش گرادیان سیاستی محبوب مانند **TRPO** و **PPO** می‌پردازد. ما به صورت گام به گام ریاضیات پشت **TRPO** و **PPO** را بررسی خواهیم کرد و درک خواهیم کرد که چگونه **TRPO** و **PPO** به نماینده کمک می کنند تا سیاست بهینه را پیدا کند. سپس یاد می گیریم چگونه **PPO** را برای انجام وظیفه نوسان معکوس پاندول پیاده سازی کنیم. در پایان، به طور مفصل درباره روش بازیگر–منتقد به نام بازیگر–منتقد با استفاده از منطقه اعتماد کرونکر–فاکتور شده خواهیم آموخت.

فصل ۱۴، با عنوان **یادگیری تقویتی توزیعی**، الگوریتم های یادگیری تقویتی توزیعی را پوشش می دهد. فصل را با درک اینکه یادگیری تقویتی توزیعی چیست آغاز خواهیم کرد. سپس چندین الگوریتم جالب یادگیری تقویتی توزیعی مانند **DQN**-رسته‌ای، **DQN**رگرسیون چارکی و **DDPG** توزیع شده را بررسی خواهیم کرد.

فصل ۱۵، با عنوان **یادگیری تقلیدی و یادگیری تقویتی معکوس**، الگوریتم‌های تقلید و یادگیری تقویتی معکوس را توضیح می‌دهد. ابتدا، نحوه کار یادگیری تقلیدی نظارت‌شده، **DAgger** و یادگیری عمیق **Q** از روی نمونه‌ها را به تفصیل خواهیم فهمید. سپس، درباره یادگیری تقویتی معکوس با بیشینه انتروپی یاد خواهیم گرفت. در پایان فصل، با یادگیری تقلیدی تخاصمی مولد آشنا خواهیم شد.

فصل ۱۶، با عنوان **یادگیری تقویتی عمیق با Stable Baselines**، به ما کمک می‌کند تا نحوه پیاده‌سازی الگوریتم‌های تقویتی عمیق با استفاده از کتابخانه‌ای به نام Stable Baselines را بفهمیم. ما یاد می‌گیریم Stable Baselines چیست و چگونه می‌توان از آن به‌طور کامل با پیاده‌سازی چندین الگوریتم جذاب یادگیری تقویتی عمیق مانند **DQN**، **A2C**، **DDPG**، **TRPO** و **PPO** استفاده کرد.

فصل ۱۷، با عنوان **مرزهای یادگیری تقویتی**، چندین مسیر جالب در **RL** را پوشش می‌دهد، مانند فرایادگیری تقویتی، یادگیری تقویتی سلسله‌مراتبی و عامل‌های افزونه‌دار با تخیل.

برای بهره‌مندی کامل از این کتاب

برای بهره‌مندی کامل از این کتاب به نرم‌افزارهای زیر نیاز دارید

- Anaconda
- Python
- Any web browser

فایل‌های نمونه کد را دانلود کنید

شما می‌توانید فایل‌های نمونه کد این کتاب را از حساب کاربری خود در http://www.packtpub.com دانلود کنید. اگر این کتاب را از جای دیگری خریداری کرده‌اید، می‌توانید به http://www.packtpub.com/support مراجعه کرده و ثبت‌نام کنید تا فایل‌ها مستقیماً از طریق ایمیل برای شما ارسال شوند.

برای دانلود فایل‌های کد، مراحل زیر را دنبال کنید:

۱. وارد حساب کاربری خود شوید یا در http://www.packtpub.com ثبت‌نام کنید.
۲. . بر روی زبانه **SUPPORT** کلیک کنید.
۳. بر روی **Code Downloads & Errata** کلیک کنید.

٤. نام کتاب را در کادر **Search** وارد کرده و دستورالعمل‌های صفحه را دنبال کنید.

پس از دانلود فایل، لطفاً اطمینان حاصل کنید که پوشه را با استفاده از آخرین نسخه نرم‌افزارهای زیر از حالت فشرده خارج می‌کنید

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- And, 7-Zip / PeaZip for Linux

بسته کد مربوط به کتاب همچنین در گیت‌هاب در آدرس

https://github.com/PacktPublishing/Deep-Reinforcement-Learning-with-Python

میزبانی شده است. ما همچنین بسته‌های کد دیگری از مجموعه غنی کتاب‌ها و ویدئوهای خود داریم که در https://github.com/PacktPublishing/ در دسترس هستند. آن‌ها را بررسی کنید!

دانلود تصاویر رنگی

ما همچنین یک فایل PDF ارائه می‌دهیم که شامل تصاویر رنگی از اسکرین‌شات‌ها/نمودارهای استفاده شده در این کتاب است. می‌توانید آن را از اینجا دانلود کنید:

https://static.packt-cdn.com/ downloads/9781839210686_ColorImages.pdf

قراردادهای استفاده‌شده در این کتاب

چندین قرارداد متنی در طول کتاب استفاده شده است:

CodeInText: نشان‌دهنده کلمات کد در متن، نام جداول پایگاه داده، نام پوشه‌ها، نام فایل‌ها، پسوند فایل‌ها، مسیرها، URLهای مثال، ورودی کاربران و حساب‌های توییتر است. برای مثال:" epsilon_greedy محاسبه سیاست بهینه را انجام می‌دهد."

یک بلوک کد به صورت زیر تنظیم می‌شود:

```
def epsilon_greedy(epsilon):
    if np.random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(Q)
```

وقتی می‌خواهیم توجه شما را به بخش خاصی از یک بلوک کد جلب کنیم، خطوط یا موارد مرتبط برجسته می‌شوند:

```
def epsilon_greedy(epsilon):
    if np.random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(Q)
```

هر ورودی یا خروجی خط فرمان به صورت زیر نوشته می‌شود:

```
source activate universe
```

بولد: نشان‌دهنده یک اصطلاح جدید، یک کلمه مهم، یا کلماتی است که روی صفحه می‌بینید، برای مثال در منوها یا جعبه‌های گفتگو، همچنین در متن به این شکل ظاهر می‌شوند. برای مثال: «**فرآیند پاداش مارکوف (MRP)** یک گسترش از زنجیره مارکوف با تابع پاداش است.»

رسیدن به برخی نکات مهم هم بصورت زیر هشدار داده شده است:

Warnings or important notes appear like this.

Tips and tricks appear like this.

امیدوارم مطالعه کتاب حاضر برای علاقمندان سودبخش باشد.

فصل اول

مقدمات یادگیری تقویتی

یادگیری تقویتی (RL)[۱] یکی از حوزه‌های **یادگیری ماشین (ML)**[۲] است. برخلاف سایر گونه‌های یادگیری ماشین، مانند **یادگیری تحت نظارت** و **یادگیری بدون نظارت**، یادگیری تقویتی ضمن تعامل با محیط خود و به روش سعی و خطا کار می‌کند.

یادگیری تقویتی یکی از فعالترین حوزه‌های تحقیقاتی در زمینه **هوش مصنوعی** است و برخی معتقدند که یادگیری تقویتی، ما را یک گام مهم به سمت دستیابی به هوش مصنوعی عمومی نزدیک کرده است. یادگیری تقویتی در چند سال گذشته با طیف گسترده‌ای از کاربردها من‌جمله توسعه سیستمهای توصیه‌گر تا هدایت خودروهای خودران، به سرعت تکامل یافته است. دلیل اصلی این تکامل، ظهور **یادگیری تقویتی عمیق** است که ترکیبی از یادگیری عمیق و یادگیری تقویتی است. با ظهور الگوریتم‌ها و کتابخانه‌های جدید یادگیری تقویتی، این حوزه به وضوح یکی از امیدوارکننده‌ترین حوزه‌های یادگیری ماشین است.

در این فصل، با بررسی چندین مفهوم مهم و اساسی مرتبط با یادگیری تقویتی، تلاش می‌شود یک پایه قوی علمی در حوزه یادگیری تقویتی برای خواننده ایجاد شود.

در این فصل به موضوعات زیر می‌پردازیم:

- **عناصر کلیدی یادگیری تقویتی**
- **ایده اصلی یادگیری تقویتی**
- **الگوریتم یادگیری تقویتی**
- **تفاوت یادگیری تقویتی با سایر گونه‌های یادگیری ماشین**
- **فرآیندهای تصمیم گیری مارکوف**
- **مفاهیم اساسی یادگیری تقویتی**
- **کاربردهای یادگیری تقویتی**
- **واژه نامه یادگیری تقویتی**

---

[۱] Reinforcement Learning (RL)
[۲] Machine Learning (ML)

ما این فصل را با تعریف *عناصر کلیدی یادگیری تقویتی* آغاز خواهیم کرد. این موضوع کمک خواهد کرد تا *ایده اصلی یادگیری تقویتی* را تشریح نماییم.

عناصر کلیدی یادگیری تقویتی

بیایید با درک برخی از عناصر کلیدی یادگیری تقویتی شروع کنیم.

عامل[۱]

عامل، یک برنامه نرم‌افزاری است که تصمیم‌گیری هوشمندانه را یاد می‌گیرد. می‌توان گفت که **عامل** در ادبیات یادگیری تقویتی، عملا یک **یادگیرنده** است. به عنوان مثال، بازیکن در بازی شطرنج می‌تواند یک عامل باشد. از آنجایی که این بازیکن یاد می‌گیرد بهترین حرکات (تصمیم‌گیری) را برای برنده شدن در بازی انجام دهد، بنابر این یک عامل محسوب می‌شود. به طور مشابه، ماریو در یک بازی ویدئویی[۲] را می‌توان یک عامل در نظر گرفت زیرا ماریو، بازی را کاوش کرده و یاد می‌گیرد که بهترین حرکات را در بازی انجام دهد.

یادداشت مترجم

معرفی بازی ماریو: سوپر ماریو یکی از بازی‌های پلتفرم فانتزی یا سکوی بازی است که توسط نین‌تندو ساخته شده و شخصیت اصلی بازی، ماریو است. علاوه بر این، سری برادران سوپر ماریو که به‌طور مخفف ماریو نامیده می‌شود، بزرگترین بازی از مجموعه بازیهای سوپر ماریو است. حداقل یک بازی فوق‌العاده با شخصیت ماریو، برای هر کنسول ویدئویی نین‌تندو منتشر شده‌است. نین‌تندو کنسول بازی ۸ بیتی است که در سال ۱۹۸۳ میلادی توسط شرکت ژاپنی نین‌تندو با نامهای فمیکام[۳] و رایانه خانگی[۴] در ژاپن و سایر نقاط آسیا، و با نام سیستم سرگرمی نین‌تندو [۵] (**NES**) در آمریکا و اروپا عرضه شد. این محصول، بزرگترین دلیل موفقیت مجدد بازی‌های رایانه‌ای پس از رکود نسبی این صنعت در سال ۱۹۸۳ محسوب می‌شود. دستگاه **NES** که در ایران به اشتباه، بیشتر با نام

---

[۱] Agent
[۲] Super Mario Bros
[۳] Famicom
[۴] Family Computer
[۵] Nintendo Entertainment System

«میکرو» یا «میکرو جنیوس» شناخته می‌شد، در واقع نام یک محصول تایوانی مبتنی بر استانداردهای **NES** بود. این محصول با اینکه از لحاظ فنی کمی از سگا مستر سیستم ضعیف تر بود اما سلطهٔ کامل این محصول (ان ای اس) در بازار جهانی مانع موفقیت مستر سیستم شد. از این کنسول به عنوان بهترین کنسول بازی تمام ادوار یاد می‌شود.

بازی‌های سوپر ماریو، ماجراهای ماریو را دنبال می‌کنند که معمولاً در بازیهای سوپر ماریو یا قارچ خور (همان نامی که در ایران به آن معروف است) بازیکن، کنترل شخصیت ماریو را دارد. او اغلب توسط برادرش لوئیجی و گاهی توسط سایر اعضای خانواده، همراهی می‌شود. در این پلتفرم از بازی‌های ویدئویی، بازیکن می‌تواند حرکت کند و بر روی دشمنان بپرد و تا انتهای هر مرحله برود. بازی‌ها ماریو معمولاً داستان ساده‌ای دارد و از این قرار است، پرنسس پیچ که معشوقهٔ ماریو است و ماریو خاطرش را می‌خواهد توسط دشمن ماریو یعنی کاراکتر باوزر دزدیده شده و ماریو به دنبال نجات پرنسس است. اولین عنوان از این سری Super Mario Bros بود که برای سیستم سرگرمی نین‌تندو (**NES**) در سال ۱۹۸۵ عرضه شد و تقریباً مفاهیم و تعاریفی که برای بازی ماریو در آن بازی تعریف شد تا به امروز دست نخورده باقی مانده‌است. این شامل بسیاری از قدرت‌ها و مواردی است که به ماریو توانایی‌های ویژه مانند بزرگ شدن یا پرتاب توپ‌های با اندازه کوچک و بزرگ را داده است.

سری سوپر ماریو بخشی از مجموعه بزرگ ماریو است. این شامل دیگر ژانرهای بازی ویدیویی و همچنین رسانه‌هایی مانند فیلم، تلویزیون، رسانه‌های چاپی و کالایی می‌باشد. بیش از ۳۱۰ میلیون نسخه بازی در سری سوپر ماریو در سرتاسر جهان از سپتامبر ۲۰۱۵ فروخته شده‌است و آن را به عنوان بهترین سری بازی‌های ویدئویی در تاریخ سر زبان‌ها انداخته‌است.

پایان یادداشت

محیط[1]

محیط، همانا جهان یا دنیای عامل است. عامل در محیط خود، اقداماتش را انجام می‌دهد، یا اصطلاحا در **محیط** قرار دارد. به عنوان مثال، در بازی شطرنج، به صفحه شطرنج، محیط می‌گوییم زیرا شطرنج باز (عامل) یاد می‌گیرد که بازی شطرنج را در صفحه شطرنج (محیط) انجام دهد. به همین ترتیب در Super Mario Bros دنیای ماریو، محیط نامیده می‌شود.

[1] Environment

حالت و اقدام

حالت[۱] یک وضعیت، موقعیت یا لحظه‌ای در محیط است که عامل می‌تواند در آن قرار گیرد. در بالا گفتیم که عامل در محیط می‌ماند و طبیعتا موقعیتهای زیادی در محیط وجود دارد که عامل می‌تواند در آنها بماند و یا آنها را بازدید کند؛ به هر یک از این موقعیتها، حالت می گویند. به عنوان مثال، در مثال بازی شطرنج، هر وضعیت در صفحه شطرنج یک حالت نامیده می شود. نماد حالت معمولاً با s نشان داده می‌شود.

عامل با محیط تعامل دارد و با انجام یک **عمل** یا **اقدام**[۲]، از حالتی به حالت دیگر حرکت می‌کند. در محیط بازی شطرنج، اقدام، آن حرکتی است که بازیکن (عامل) انجام می‌دهد. اقدام معمولا با نماد a نشان داده می‌شود.

پاداش

ما در بالا گفتیم که عامل با انجام یک عمل یا اقدام با محیط خود تعامل دارد و از حالتی به حالت دیگر حرکت می‌کند. بدنبال هر اقدامی، عامل ما **پاداشی** دریافت می کند. **پاداش**[۳] چیزی نیست جز یک مقدار عددی، مثلاً مثبت یک (+۱) برای یک اقدام خوب و منفی یک (−۱) برای یک اقدام بد. اما چگونه در مورد خوب یا بد بودن یک اقدام، تصمیم بگیریم؟

در مثال بازی شطرنج ، اگر عامل حرکتی را انجام دهد که در آن یکی از مهره‌های شطرنجِ حریف را برداشته و بیرون بیندازد، اقدام خوبی محسوب می‌شود و عامل، پاداش مثبتی دریافت می‌کند. به همین ترتیب، اگر عامل، حرکتی انجام دهد که منجر به از دست دادن مهره شطرنج خود و بیرون انداختن آن توسط حریف شود، اقدام بدی انجام شده و یا آن عمل، بد محسوب می‌شود و عامل نتیجتا پاداش منفی دریافت می‌کند. پاداش را با نماد r نشان می‌دهند.

---

۱ State
۲ Action
۳ Reward

ایده اصلی یادگیری تقویتی

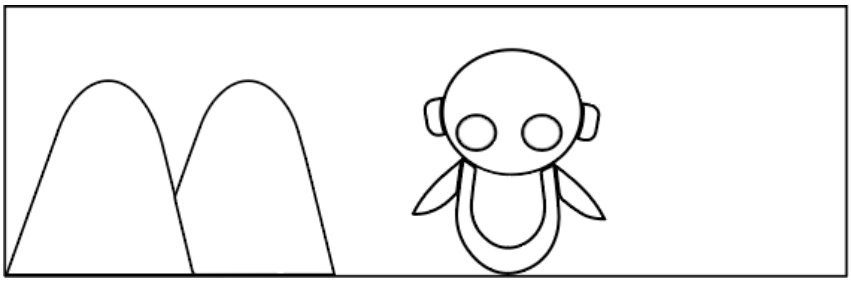
بیایید با یک قیاس شروع کنیم. فرض کنید داریم به یک سگ (که در زبان یادگیری تقویتی همان: عامل است) یاد می‌دهیم که توپی را بگیرد. به جای اینکه صریحاً به آن سگ یاد بدهیم که او باید توپ را بگیرد؛ ما فقط یک توپ پرتاب را می‌کنیم و هر بار که سگ توپ را می‌گیرد، به او یک کوکی (**پاداش**) می‌دهیم. اگر سگ نتواند توپ را بگیرد، ما به او کوکی نمی‌دهیم. بنابراین، سگ متوجه می‌شود که چه **اقدامی** باعث دریافت یک کوکی شده و آن اقدام را تکرار می‌کند. یعنی، سگ متوجه خواهد شد که گرفتن توپ، باعث دریافت کوکی مورد نظر شده و لذا سعی می‌کند گرفتن توپ را تکرار کند. بنابراین، و به این ترتیب، سگ یاد می‌گیرد که یک توپ را بگیرد در حالی که عملا هدفش به حداکثر رساندن کوکی‌هایی است که می‌تواند دریافت کند.

به طور مشابه، در ادبیات یادگیری تقویتی، ما به عامل، **آموزش** نمی‌دهیم که چه کاری انجام دهد یا چگونه آن را انجام دهد. در عوض، ما برای هر **عملی** که عامل انجام دهد، به او **پاداش** می دهیم. ما به عامل وقتی **عمل** خوبی انجام می‌دهد **پاداش** مثبت می‌دهیم و زمانی که **اقدام** بدی را انجام دهد، **پاداش** منفی می‌دهیم (انگار جریمه‌اش می‌کنیم). در آغاز، عامل با انجام یک اقدام یا عمل تصادفی (دلبخواه) شروع می‌کند و اگر آن **اقدام** خوب باشد، به عامل یک **پاداش** مثبت می‌دهیم، به این منظور که عامل بفهمد عمل خوبی انجام داده و آن عمل را تکرار کند. اگر **عملی** که عامل انجام می دهد بد باشد، به عامل **پاداش** منفی می‌دهیم تا عامل بفهمد عمل بدی انجام داده و آن عمل را تکرار نکند.

بنابراین، از فرایند یادگیری تقویتی می‌توان به عنوان یک فرآیند یادگیری آزمون و خطا هم یاد کرد که در آن عامل تلاش می‌کند اقدامات مختلف را انجام داده و یاد بگیرد که آن اقدامی خوب است که پاداش مثبت برایش دارد.

در مثال فوق برای آموزش سگ، آن سگ به زبان یادگیری تقویتی، نمایانگر عامل است و دادن یک کلوچه به سگ در هنگام گرفتن توپ، نوعی پاداش مثبت است و ندادن کلوچه به او، عملا یک پاداش منفی است. بنابراین، سگ (عامل) اقدامات مختلفی را بررسی می‌کند، یعنی گرفتن توپ و نگرفتن توپ، و می‌فهمد که گرفتن توپ، یک اقدام خوب است زیرا برای سگ یک پاداش مثبت (گرفتن یک کلوچه) به همراه دارد.

بیایید ایده یادگیری تقویتی را با یک مثال ساده، بیشتر بررسی کنیم. بیایید فرض کنیم که ما می‌خواهیم به یک ربات (یعنی عامل) آموزش دهیم که بدون برخورد به کوه راه برود (همانطور که در شکل ۱.۱ نشان داده شده است).



شکل ۱.۱: راه رفتن ربات

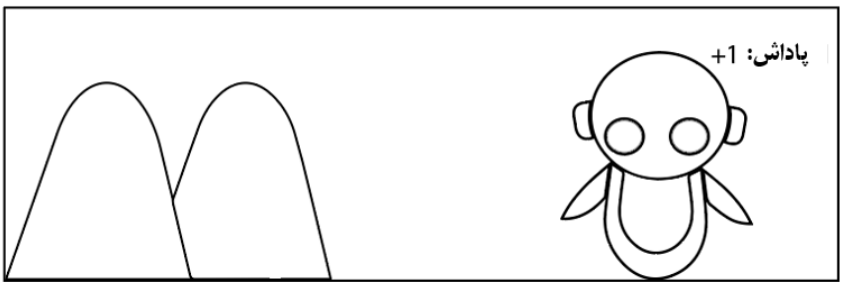
ما به ربات به صراحت یاد نمی‌دهیم که به سمت کوه نرود. در عوض، اگر ربات به کوه برخورد کرده و گیر کند، به آن یک پاداش منفی می‌دهیم، مثلاً منفی یک. بنابراین، ربات متوجه خواهد شد که برخورد به کوه، یک اقدام اشتباه است و آن عمل را تکرار نخواهد کرد[1]. شکل ۱.۲:

شکل ۱.۲: ربات به کوه برخورد می‌کند.

[1] **یادداشت مترجم:** به یاد بیاریم یکی از سرگرمیها و بازیهای دوران کودکی برای یافتن اشیاء پنهان شده و یا یک تغییر محسوس در اتاق. در این بازی، فردی را بعنوان بازیکن از اتاق بیرون می‌کردند و در غیاب او چیزی را در نقطه‌ای پنهان و یا جابجا می‌کردند. وقتی فرد به اتاق برمی‌گشت باید آن شیء پنهان شده را پیدا می‌کرد یا تغییر انجام شده را حدس می‌زد. دیگر افراد حاضر در محیط حق نداشتند بطور مستقیم یابنده (بازیکن اصلی) را راهنمایی کنند بلکه راهنمایی او با صدایی بود که برای او می‌نواختند. اگر او به شیء قایم شده و یا تغییر انجام شده نزدیک می‌شد این صدا بلند بود (+) و اگر از آن دور میشد این صدا یواش و آرام (–) بود. اگر او شیء مفقوده را پیدا می‌کرد و یا تغییر را اعلام می‌کرد، برنده بازی بود.

به همین ترتیب، هنگامی که ربات بدون برخورد به کوه در جهت درست راه می‌رود، به ربات یک جایزه مثبت می‌دهیم، مثلاً مثبت یک. بنابراین، ربات متوجه می‌شود که برخورد نکردن به کوه یک اقدام خوب است (شکل ۱.۳) و آن عمل را تکرار می‌کند.

بنابراین، در تنظیم‌گری یادگیری تقویتی، عامل ما اقدامات مختلف را بررسی می‌کند و بهترین عمل را بر اساس پاداش دریافتی می‌آموزد.



شکل ۱.۳: ربات از کوه دوری می‌کند.

اکنون ما ایده اولیه از نحوه عملکرد یادگیری تقویتی را بیان کردیم، در بخشهای آینده ضمن تشریح جزئیات بیشتری از این موضوع، مفاهیم مهم مربوط به یادگیری تقویتی را بحث خواهیم کرد.

الگوریتم یادگیری تقویتی

مراحل عملیاتی در یک الگوریتم معمولی یادگیری تقویتی به شرح زیر است:

۱. ابتدا عامل با انجام یک **اقدام**، با محیط خود تعامل می‌کند.

۲. با انجام این **اقدام**، عامل از حالتی به حالت دیگر حرکت می‌کند.

۳. سپس عامل بر اساس اقدامی که انجام داده است، **پاداش** دریافت می‌کند.

٤. بر اساس **پاداش**، عامل متوجه خوب یا بد بودن عمل می‌شود.

۵. اگر عمل خوب بود، یعنی اگر عامل **پاداش** مثبتی دریافت کرد، در آن صورت عامل، انجام آن **عمل** را ترجیح می‌دهد، در غیر این صورت، عامل برای جستجوی **پاداش** مثبت، **اقدامات** دیگری را انجام می‌دهد.

یادگیری تقویتی اساساً یک فرآیند یادگیری آزمون و خطا است. حالا بیایید مثال بازی شطرنج را مرور کنیم. عامل (که در اینجا عملا یک برنامه نرم‌افزاری است) نقش شطرنج باز را به عهده دارد. بنابراین، عامل با انجام یک اقدام (یعنی یک حرکت) در تعامل با محیط (یعنی صفحه شطرنج) است. اگر عامل بخاطر عمل خود، پاداش مثبتی دریافت کند، انجام آن عمل را ترجیح می‌دهد. در غیر این صورت، عمل متفاوتی را پیدا خواهد کرد که پاداش مثبتی به همراه داشته باشد.

در نهایت، هدف عامل به حداکثر رساندن **پاداشی** است که می‌گیرد. اگر عامل پاداش خوبی دریافت کند، به این معنی است که عمل خوبی انجام داده است. اگر عامل یک **عمل** خوب انجام دهد، به این معنی است که می‌تواند بازی را برنده شود. بنابراین، عامل یاد می‌گیرد که با به حداکثر رساندن **پاداش**، بازی را برنده شود.

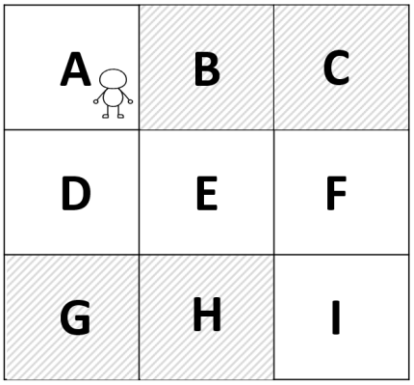
عامل و یادگیری تقویتی در دنیای مشبک[۱]

بیایید درک خود از یادگیری تقویتی را با نگاه به مثال ساده دیگری دنبال کنیم. محیط دنیای مشبک[۲] یا جدول–گونه شکل ۱.۴ را در نظر بگیرید.

موقعیتهای **A** تا **I** در محیط را حالتهای محیط می‌نامند. هدف عامل این است که با شروع از حالت **A** بدون بازدید از حالتهای سایه‌دار یا هاشور–خورده (**B**، **C**، **G** و **H**) به حالت **I** برسد. بنابراین، برای دستیابی به هدف، هر زمان که عامل ما از یک حالت هاشور–خورده بازدید کرد، یک پاداش منفی (مثلا منفی یک) دریافت کرده و وقتی وارد یک

[۱] Grid World
[۲] Grid World Environment

خانه سپید یا حالت بدون هاشور شد، یک پاداش مثبت (مثلا مثبت یک) را نصیب خود می‌کند. اقدامات در این محیط، حرکت به جهت‌های چهارگانه: *بالا، پایین، راست و چپ* هستند. عامل می‌تواند هر یک از این اقدامات چهارگانه را انجام دهد تا از حالت **I** از حالت **A** برسد.



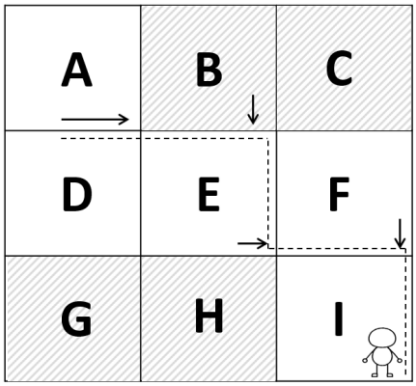
شکل ۱.۴: محیط دنیای مشبک یا جدول–گونه

اولین باری که عامل با محیط تعامل می‌کند (یعنی تکرار اول)، خیلی بعید است که عامل اقدام صحیحی را انجام دهد و بنابراین پاداش منفی دریافت می‌کند. یعنی در اولین تکرار، عامل در هر کدام از حالتها، یک اقدام تصادفی (دلبخواه) را انجام می‌دهد و این ممکن است عامل را به دریافت پاداش منفی سوق دهد. اما طی یک سری تکرار دیگر، عامل یاد می‌گیرد که در هر حالت از طریق پاداشی که به دست می‌آورد، عمل صحیح را انجام دهد و به او در رسیدن به هدف کمک کند. اجازه دهید این را با جزئیات بررسی کنیم.

تکرار یک

همانطور که گفتیم، در اولین تکرار، عامل در هر حالت خود، یک عمل تصادفی (دلبخواه) انجام می‌دهد. به عنوان مثال به شکل زیر نگاه کنید. در تکرار اول، عامل از حالت **A** به سمت *راست* حرکت می‌کند و به حالت جدید **B** می‌رسد. اما از آنجایی که **B** حالت هاشوردار است، عامل پاداش منفی دریافت می‌کند و بنابراین عامل متوجه می‌شود که حرکت به سمت *راست* در حالت **A** عمل خوبی نیست. لذا هنگامی که دفعه بعد از وضعیت **A** بازدید می کند، به جای حرکت به سمت *راست*، اقدام دیگری را امتحان می کند.

همانطور که شکل ۱.۵ نشان می‌دهد، عامل از حالت **B**، به سمت *پایین* حرکت می‌کند و به حالت جدید **E** می‌رسد. از آنجایی که **E** یک حالت بدون هاشور است، عامل یک پاداش مثبت دریافت خواهد کرد، بنابراین عامل درک خواهد کرد که پایین آمدن از حالت **B** اقدام خوبی است.



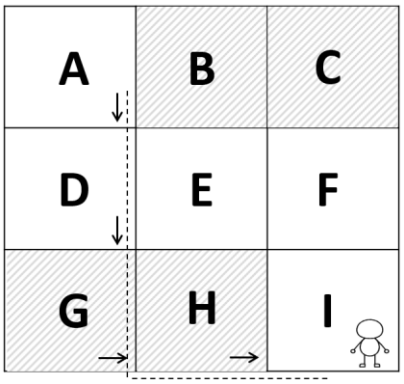
شکل ۱.۵: اقدامات انجام شده توسط عامل در تکرار اول

عامل از حالت **E**، به سمت *راست* حرکت می‌کند و به حالت **F** می‌رسد. از آنجایی که **F** یک حالت بدون هاشور است، عامل یک پاداش مثبت دریافت می‌کند و عملا می‌فهمد که وقتی در حالت **E** قرار دارد، حرکت به سمت *راست*، اقدام خوبی است. از حالت **F**، عامل به سمت *پایین* حرکت می‌کند و به حالت هدف **I** می‌رسد و پاداش مثبت دریافت می‌کند، بنابراین عامل متوجه می‌شود که *پایین* آمدن از حالت **F** (اقدام به سمت پایین در این حالت) اقدام خوبی است.

تکرار دو

در تکرار دوم، عامل مجددا از حالت **A** بازی را شروع می‌کند. این بار به جای حرکت به سمت *راست*، یک عمل متفاوت را امتحان می‌کند، زیرا عامل در تکرار قبلی یاد گرفت که حرکت به سمت *راست* در حالت **A** عمل خوبی نیست.

بنابراین، همانطور که شکل ۱.۶ نشان می دهد، در این تکرار عامل از حالت **A** به سمت *پایین* حرکت می‌کند و به حالت **D** می رسد. از آنجایی که **D** یک حالت بدون هاشور است، عامل یک پاداش مثبت دریافت می‌کند و اکنون عامل متوجه می‌شود که حرکت به سمت *پایین*، عمل خوبی در حالت **A** است.



شکل ۱.۶: اقدامات انجام شده توسط عامل در تکرار ۲

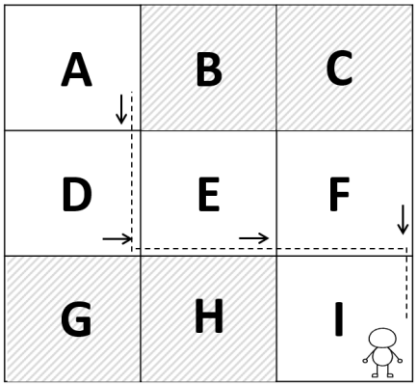
همانطور که در شکل قبل نشان داده شده است، از حالت **D**، عامل به سمت *پایین* حرکت کرده و به حالت **G** می‌رسد. اما از آنجایی که **G** یک حالت سایه‌دار (هاشور–خورده) است، عامل یک پاداش منفی دریافت می‌کند و بنابراین عامل متوجه می‌شود که حرکت به سمت *پایین* در حالت **D**، عمل خوبی نیست و هنگامی که دفعه بعد از حالت **D** بازدید کرد، به جای حرکت به سمت *پایین*، اقدام دیگری را امتحان می‌کند.

عامل از حالت **G**، به سمت *راست* حرکت کرده و به حالت **H** می‌رسد. از آنجایی که **H** یک حالت هاشوردار است، لذا عامل پاداش منفی دریافت کرده و می‌فهمد که حرکت به سمت *راست* در حالت **G**، عمل خوبی نیست.

حالا عامل از **H** به سمت *راست* حرکت می‌کند و به حالت هدف **I** رسیده و پاداش مثبت دریافت می‌کند، بنابراین عامل می‌فهمد که حرکت به سمت *راست* از حالت **H** عمل خوبی است.

تکرار سه

در تکرار سوم، عامل مجددا و برای بار سوم بازی را از حالت **A** شروع کرده و سمت به *پایین* حرکت می‌کند، زیرا در تکرار دوم، عامل ما یاد گرفت که حرکت به *پایین* در حالت **A** یک عمل خوب است. بنابراین، عامل از حالت **A** به *پایین* حرکت می‌کند و به حالت بعدی، **D** می‌رسد. همانطور که شکل ۱.۷ نشان می‌دهد:

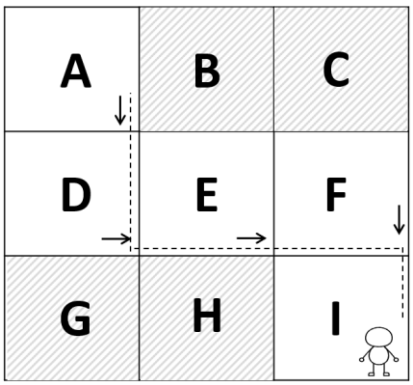


شکل ۱.۷: اقدامات انجام شده توسط عامل در تکرار ۳

اکنون، از حالت **D**، عامل به جای حرکت به سمت *پایین*، یک عمل متفاوت را امتحان می‌کند، زیرا در تکرار دوم عامل ما متوجه شد که حرکت به سمت *پایین* در حالت **D** عمل خوبی نیست. بنابراین، در این تکرار، عامل از حالت **D** به *راست* حرکت می کند و به حالت **E** می رسد. از حالت **E**، عامل به سمت *راست* حرکت کرده که او قبلاً در اولین تکرار این را یاد گرفته است حرکت به سمت *راست* از حالت **E** برایش خوش یمن بود و لذا اقدام خوبی محسوب می‌شود. درنتیجه این اقدام، عامل به حالت **F** می‌رسد.

اکنون، از حالت **F**، عامل با توجه به چیزی که در اولین تکرار یاد گرفته بود (یعنی حرکت به سمت *پایین* در حالت **F** عمل خوبی است)، به سمت *پایین* حرکت می‌کند و به حالت هدف **I** می‌رسد.

شکل ۱.۸ نتیجه تکرار سوم را نشان می‌دهد:



شکل ۱.۸: عامل بدون بازدید از حالتهای سایه‌دار به حالت هدف می‌رسد.

همانطور که می‌بینیم، عامل ما با موفقیت یاد گرفته است که بدون بازدید از حالتهای هاشوردار بر اساس پاداشها از حالت **A** به حالت هدف **I** برسد.

به این ترتیب، عامل اقدامات مختلفی را در هر حالت آزمایش کرده و بر اساس پاداشی که به دست می‌آورد، خوب یا بد بودن یک عمل را درک می‌کند. هدف عامل به حداکثر رساندن پاداش است. بنابراین، عامل همیشه سعی می‌کند اقدامات خوبی انجام دهد که پاداش مثبت برایش به همراه دارد. وقتی عامل در هر حالت، اقدامات خوبی انجام دهد، در نهایت عامل را به سمت رسیدن به هدف هدایت می‌کند.

توجه داشته باشید که هر **تکرار**[۱] در اصطلاح یادگیری تقویتی یک **پردینه** یا **اپیزود**[۲] نامیده می شوند. بعداً در همین فصل، درباره اپیزودهای یادگیری تقویتی، بیشتر خواهیم آموخت.

تفاوت یادگیری تقویتی با سایر پارادایمهای یادگیری ماشین

ما می‌توانیم یادگیری ماشین را به سه دسته طبقه‌بندی کنیم:

- **یادگیری تحت نظارت**
- **یادگیری بدون نظارت**
- **یادگیری تقویتی**

---

[۱] Iteration
[۲] Episode

با توجه به نوع کاربری واژه **اپیزود** در هنر و مهندسی می‌توان معادلهای متفاوتی برای آن در زبان فارسی پیشنهاد داد:
بر اساس کلمات مرکب: ۱. با بن‌مایه **"نما"** مثلا نماوا، نماوار، ۲. با بن‌مایه **"بخش"** مثلا بخشانه، بخشواره، بخش‌مند، بخشینه، ۳. با بن‌مایه **"ساخت"** مثلا سازه‌وار، سازه‌وند ۴. بر پایه **"راه و سیر"** مثلا ره‌سو (رهسو)، ره‌سیر، رهواره ۵. کاربرد **هنری**: پرده (پرده اول، پرده دوم) گاه (گاه اول، گاه دوم)، ترکیبات پرده: پردین، پردان، پردینه، پرداک، پرده‌وار، پرده‌سان، پردگان.
ما در این متن، از واژه **پرده** (و ترکیبات آن مثل **پردینه**) استفاده می‌کنیم تا بیشتر بر قطعه‌واری و بخشگاهی هر اپیزود تاکید کنیم و نه رهسویی و سوگاهی آن. همچنین وظیفه تصویرسازی واژه اپیزود که در واژگان مبتنی بر نما آمده است را می‌توان در کلمه "پرده" با معنای آن فضای هنری، حس کرد.

در **یادگیری نظارت شده**، ماشین از داده‌های آموزشی یاد می‌گیرد. داده‌های آموزشی از یک جفت ورودی و خروجی علامتدار یا برچسبدار[۱] تشکیل شده است. بنابراین، ما مدل (یا عامل) را با استفاده از داده‌های آموزشی به گونه‌ای آموزش می‌دهیم که مدل بتواند یادگیری خود را به داده‌های نادیدهٔ جدید تعمیم دهد. این نوع یادگیری، یادگیری تحت نظارت نامیده می‌شود زیرا داده‌های آموزشی به عنوان یک ناظر عمل می‌کند، در واقع زوجهای داده ما عملا جفتهای ورودی و خروجی با برچسب هستند که مدل را در مسیر یادگیری خود هدایت می‌کنند.

حال بیایید تفاوت بین یادگیری تحت نظارت و تقویتی را با یک مثال درک کنیم. مثال سگی را که قبلاً در ابتدای فصل مورد بحث قرار دادیم در نظر بگیرید. در آموزش نظارت شده، برای آموزش گرفتن توپ توسط سگ، ما با مشخص کردن گردش به چپ، حرکت به راست، برداشتن هفت قدم به جلو، گرفتن توپ و امثال آن، فرایند آموزش سگ را در قالب داده‌های آموزشی به صراحت انجام می دهیم. اما در یادگیری تقویتی، فقط یک توپ را پرتاب می‌کنیم و هر بار که سگ توپ را می‌گیرد به آن کوکی (بعنوان پاداش) می‌دهیم. بنابراین، سگ یاد می‌گیرد که توپ را بگیرد و این در حالی است که او عملا و فقط سعی می‌کند کوکی‌ها (پاداش) خودش را به حداکثر برساند.

بیایید یک مثال دیگر را در نظر بگیریم. فرض کنید می‌خواهیم با استفاده از یادگیری تحت نظارت، مدل را برای بازی شطرنج آموزش دهیم. در این مورد، ما مجموعه‌ای از داده‌های آموزشی خواهیم داشت که اولا شامل تمام حرکاتی است که یک بازیکن می‌تواند در هر حالت انجام دهد، و ثانیا همراه با برچسبهایی است که نشان می‌دهد آیا حرکت خوبی است یا خیر. سپس، ما مدل را آموزش می‌دهیم تا از این داده‌های آموزشی یاد بگیرد، در حالی که در مورد یادگیری تقویتی، ما هیچ نوع داده آموزشی به عامل داده نمی‌دهیم. در عوض، ما فقط به ازای هر **اقدام** یا عملی که عامل انجام می‌دهد به او **جایزه** می‌دهیم. سپس عامل در تعامل با محیط یاد می‌گیرد؛ یعنی بر اساس پاداشی که می‌گیرد، اقدامات خود را انتخاب می‌کند.

---

[۱] Labeled

مشابه یادگیری تحت نظارت[۱]، در **یادگیری بدون نظارت**، مدل (عامل) را بر اساس داده‌های آموزشی موجود، آموزش می‌دهیم. اما در مورد یادگیری بدون نظارت، داده‌های آموزشی حاوی هیچ برچسبی نیستند. یعنی فقط از ورودی تشکیل شده و نه خروجی. هدف از یادگیری بدون نظارت، تعیین الگوهای پنهان در ورودیهاست. این تصور غلط رایج است که یادگیری تقویتی نوعی یادگیری بدون نظارت است، اما اینطور نیست! در یادگیری بدون نظارت، مدل ما ساختار پنهان درون داده‌ها را می‌آموزد، در حالی که در یادگیری تقویتی، مدل با به حداکثر رساندن پاداش یاد می‌گیرد.

به عنوان مثال، یک سیستم توصیه فیلم را در نظر بگیرید. فرض کنید می خواهیم فیلم جدیدی را به کاربر توصیه کنیم. با یادگیری بدون نظارت، مدل (یا عامل) فیلم‌هایی شبیه به فیلم‌هایی را که کاربر (یا کاربران با نمایه‌ای مشابه کاربر) قبلاً دیده‌اند پیدا کرده و فیلم‌های جدید را به کاربر توصیه می‌کند. اما با یادگیری تقویتی، عامل ما دائماً از کاربر بازخورد دریافت می‌کند. این بازخورد نشان‌دهنده پاداش‌ها است (پاداش می‌تواند رتبه‌بندی‌هایی باشد که کاربر به فیلمهای تماشا کرده می‌دهد، یا زمان صرف شده برای تماشای فیلم، و یا زمان صرف شده برای تماشای تریلرها و دیگر شاخصهای مرتبط). بر اساس این پاداشها، یک عامل یادگیری تقویتی، ترجیحات کاربر را درک کرده و سپس فیلمهای جدید را بر اساس آن پیشنهاد می‌دهد.

از آنجایی که عامل یادگیری تقویتی با کمک پاداشها در حال یادگیری است، لذا می‌تواند بفهمد که آیا اولویت یا ذائقه کاربر تغییر کرده یا خیر و طبیعتا فیلمهای جدید را با توجه به اولویت تغییر یافته کاربر به صورت پویا پیشنهاد می‌دهد.

بنابراین، می‌توان گفت که در هر دو یادگیری نظارت شده و یادگیری بدون نظارت، مدل (یا عامل) بر اساس مجموعه داده‌های آموزشی (نشاندار یا بی‌نشان) یاد می‌گیرد، در حالی که در یادگیری تقویتی عامل ضمن تعامل مستقیم با محیط یاد می‌گیرد. بنابراین، یادگیری تقویتی اساساً یک تعامل بین عامل و محیط آن است.

---

[۱] **یادداشت مترجم:** از دوران دبیرستان به خاطر داریم که نوع خاصی از رابطه را یک **تابع** می‌نامیدیم که پس از اعمال بر مقادیر **دامنه**، مقادیر جدیدی تحت عنوان **بُرد** را به ما می‌داد. در تشابه با آن ادبیات:

در **یادگیری بدون نظارت** ما دامنه داریم اما تابع و بُرد را نداریم و تنها با بررسی مقادیر دامنه، بدنبال کشف الگوهای پنهان میان داده‌های خود هستیم که در اینجا به دامنه، **دیتاست** می‌گویند.

در **یادگیری با نظارت** ما هر دوی دامنه و بُرد را داریم اما تابع را نداریم و بدنبال پیدا کردن آن هستیم و لذا یک تابع را چنان آموزش می‌دهیم که با کمترین خطا، همان مقادیر برد را پیش‌بینی کند که در اینجا به بُرد، تگ یا **برچسب** می‌گویند.

قبل از بیان مفاهیم اساسی یادگیری تقویتی، ما یک فرآیند محبوب را برای کمک به تصمیم‌گیری در یک محیط یادگیری تقویتی معرفی می‌کنیم.

فرآیندهای تصمیم‌گیری مارکوف

فرآیند تصمیم‌گیری مارکوف[۱] **(MDF)** یک چارچوب ریاضی مناسب برای حل مسئله یادگیری تقویتی فراهم می‌کند. تقریباً تمام مسائل یادگیری تقویتی را می‌توان به عنوان یک فرایند تصمیم‌گیری مارکوف مدل کرد. **MDP**ها به طور گسترده‌ای برای حل مسائل مختلف بهینه‌سازی استفاده می‌شوند. در این قسمت متوجه خواهیم شد که **MDP** چیست و چگونه در یادگیری تقویتی استفاده می‌شود.

برای درک یک **MDP**، ابتدا باید در مورد خاصیت مارکوف و زنجیره مارکوف مباحثی را یاد بگیریم.

خاصیت مارکوف و زنجیره مارکوف

خاصیت مارکوف بیان می‌کند که آینده فقط به حال بستگی دارد و نه به گذشته. زنجیره مارکوف، که همچنین به عنوان فرآیند مارکوف شناخته می شود، متشکل از دنباله‌ای از حالات است که به شدت از خاصیت مارکوف تبعیت می‌کنند. یعنی زنجیره مارکوف، یک مدل احتمالی است که برای پیش‌بینی حالت بعدی صرفاً به وضعیت فعلی بستگی دارد نه حالت‌های قبلی، یعنی آینده به طور مشروط مستقل از گذشته است.

به عنوان مثال، اگر بخواهیم آب و هوا را پیش‌بینی کنیم و بعلاوه بدانیم که حالت فعلی ابری است، می‌توانیم پیش‌بینی کنیم که حالت بعدی ممکن است بارانی باشد. ما به این نتیجه رسیدیم که حالت بعدی فقط با در نظر گرفتن حالت فعلی (ابری) احتمالاً بارانی است و نه حالتهای قبلی که ممکن است آفتابی، بادی و غیره باشد.

---

[۱] Markov Decision Process (MDP)

با این حال، خاصیت مارکوف برای همه فرآیندها برقرار نیست. به عنوان مثال، در پرتاب تاس (حالت بعدی یعنی عددی که خواهیم دید) هیچ وابستگی به عدد قبلی که روی تاس نشان داده شده است (یعنی حالت فعلی) ندارد.

حرکت از حالتی به حالت دیگر را گذار و **احتمال** آن را **احتمال گذار** می گویند. احتمال گذار را با $P(s'|s)$ نشان می‌دهیم. نشان دهنده **احتمال** حرکت از حالت s به حالت بعدی s' است. با فرض اینکه ما سه حالت (ابری، بارانی و بادی) در زنجیره مارکوف داریم می‌توانیم **احتمال** گذار یا انتقال از یک حالت به حالت دیگر را با استفاده از جدولی به نام جدول مارکوف، همانطور که در جدول ۱.۱ آمده است، نشان دهیم.

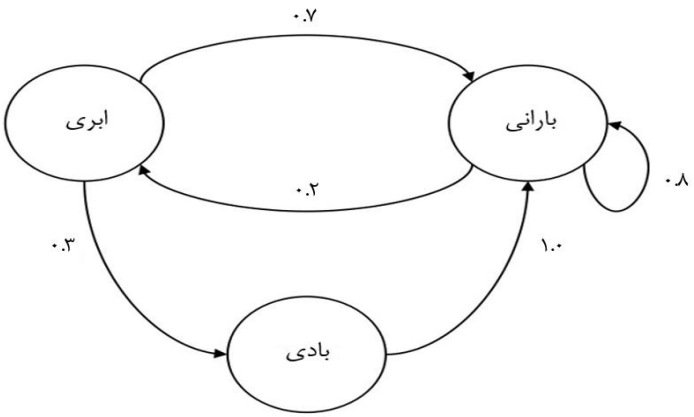
از *جدول ۱.۱* می توان نتیجه‌گیری کرد که:

- از حالت ابری با **احتمال** ۷۰٪ به حالت بارانی و با **احتمال** ۳۰٪ به حالت بادی (وزش باد) منتقل می‌شویم.
- از حالت بارانی با **احتمال** ۸۰٪ به حالت بارانی یکسان و با **احتمال** ۲۰٪ به حالت ابری منتقل می‌شویم.
- از حالت بادی (وزش باد)، با احتمال ۱۰۰٪ به حالت بارانی منتقل می‌شویم.

حالت فعلی	حالت بعدی	احتمال انتقال
ابری	بارانی	۰.۷
ابری	بادی	۰.۳
بارانی	بارانی	۰.۸
بارانی	بادی	۰.۲
بادی	بارانی	۱.۰

جدول ۱.۱: نمونه‌ای از جدول مارکوف

همانطور که در شکل ۱.۹ نشان داده شده است، می‌توانیم این اطلاعات انتقال زنجیره مارکوف را در قالب یک نمودار حالت نمایش دهیم:



شکل ۱.۹: نمودار حالت یک زنجیره مارکوف

همانطور که در شکل ۱.۱۰ نشان داده شده است، می‌توان **احتمالات انتقال** را در ماتریسی به نام **ماتریس انتقال**[۱] فرموله کرد:

	ابری	بارانی	بادی
ابری	۰.۰	۰.۷	۰.۳
بارانی	۰.۲	۰.۸	۰.۰
بادی	۰.۰	۱.۰	۰.۰

شکل ۱.۱۰: **یک ماتریس انتقال**

بنابراین، برای نتیجه‌گیری، می‌توان گفت که زنجیره مارکوف یا فرآیند مارکوف شامل مجموعه‌ای از حالتها به همراه **احتمالات** انتقال یا گذار آنهاست.

[۱] Transition Matrix

فرایند پاداش مارکوف

فرآیند پاداش مارکوف[۱] **(MRP)** نوعی گسترش یا توسعهٔ زنجیره مارکوف با اضافه کردن یک **تابع پاداش**[۲] به آن است. قبلا گفتیم که زنجیره مارکوف از حالتها و یک احتمال گذار تشکیل شده است. اما فرایند **پاداش** مارکوف از **حالتها، احتمال گذار و همچنین یک تابع پاداش** تشکیل شده است.

یک **تابع پاداش**، مقدار پاداشی را که در هر حالت به دست می‌آوریم به ما می‌گوید. به عنوان نمونه، بر اساس مثال آب و هوا در بالا، تابع پاداش، پاداش ما را در حالت ابری، پاداش ما در حالت وزش باد و کلا پاداش ما را در همه سایر حالات، به ما می‌گوید. تابع پاداش معمولاً با $R(s)$ مشخص می‌شود.

بنابراین، **MRP** شامل: حالتهای s، یک احتمال گذار $P(s'|s)$ و یک **تابع پاداش** $R(s)$ می‌باشد.

فرایند تصمیم گیری مارکوف

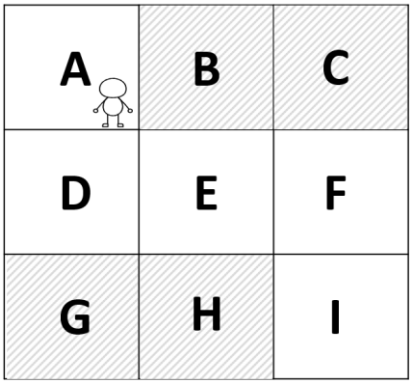
فرآیند تصمیم گیری مارکوف[۳] **(MDP)** ترکیب یا توسعه **MRP** با یک افزونه دیگر به نام: **اقدامات** است. قبلا گفتیم که **MRP** از حالتها، احتمال گذار و یک تابع پاداش تشکیل شده است.

اما **MDP** از حالتها، احتمال گذار، **تابع پاداش** و همچنین **اقدامات** تشکیل شده است. همچنین قبلا گفتیم که خاصیت مارکوف[۴] بیان می‌کند که حالت بعدی فقط به حالت فعلی وابسته است و مبتنی بر حالت قبلی نیست. اما آیا خاصیت مارکوف قابل استفاده در یادگیری تقویتی است؟ بله! در محیط یادگیری تقویتی، عامل فقط بر اساس وضعیت فعلی تصمیم می‌گیرد و نه بر اساس حالتهای گذشته. بنابراین، ما می‌توانیم یک محیط یادگیری تقویتی را به عنوان **MDP** مدل کنیم.

---

۱ The Markov Reward Process (MRP)
۲ Reward Function
۳ Markov Decision Process (MDP)
۴ Markov Property

مفهوم فوق با یک مثال جدولی، بهتر درک می‌شود. یک محیط یا یک دنیای مسئله را در نظر بگیرید. حال می‌توانیم این محیط را با استفاده از **MDP** مدل کنیم. به عنوان مثال، همان محیط دنیای مشبک را که قبلاً معرفی کردیم در نظر بگیرید. شکل ۱.۱۱ محیط جهان مشبک را نشان می‌دهد که در آن، هدف عامل دستیابی به حالت **I** با شروع از حالت **A** بدون مراجعه به حالتهای سایه‌دار است:



شکل ۱.۱۱: محیط جهانی مشبک یا جدول-گونه

یک عامل فقط بر اساس حالت فعلی که عامل در آن قرار دارد ، تصمیم می‌گیرد (اقدام می‌کند)؛ یعنی این تصمیم براساس حالت گذشته نیست. بنابراین، ما می توانیم محیط خود را به عنوان **MDP** مدل کنیم. ما آموختیم که **MDP** از چهارگانه: حالتها ، اقدامات، احتمالات گذار و یک تابع پاداش تشکیل شده است. حال، بیایید یاد بگیریم که چگونه این موضوع با محیط یادگیری تقویتی ما ارتباط دارد:

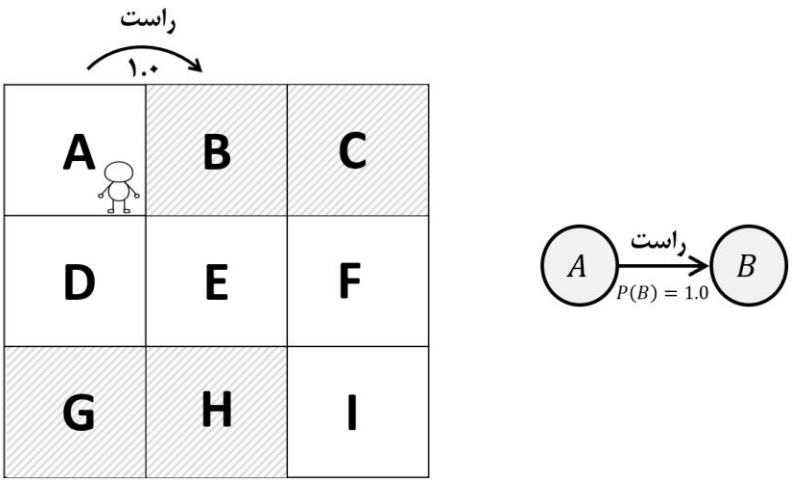
حالتها – مجموعه ای از وضعیتهای موجود در محیط. بنابراین، در محیط جهان مشبک، ما حالتهای **A** تا **I** را داریم.

اقدامات – مجموعه‌ای از **اعمال** یا **کنشهایی** است که عامل ما می‌تواند در هر حالت انجام دهد. یک عامل یک عمل را انجام می‌دهد و از یک حالت به حالت دیگر حرکت می‌کند. بنابراین، در محیط جهان مشبک یا جدولی، مجموعه **اقدامات** ممکن شامل جهات چارگانهٔ *بالا* ، *پایین*، چپ و *راست* است.

احتمال گذار – احتمال گذار یا انتقال توسط $P(s'|s,a)$ مشخص می‌شود. این احتمال به معنای شانس حرکت از حالت s به حالت بعدی s' است اگر تصمیم a را اتخاذ کنیم. اگر **MRP** را مجددا بررسی کنید، خواهید دید که احتمال

انتقال یا گذار به شکل $P(s'|s)$ بود، یعنی فقط **احتمال** رفتن از حالت s به حالت s' را نشان می‌داد، یعنی این احتمال، شامل اقدامات مرتبط نمی‌شد. اما در **MDP**، ما اقدامات را هم در نظر می‌گیریم، و بنابراین **احتمال** انتقال توسط $P(s'|s,a)$ مشخص میشود.

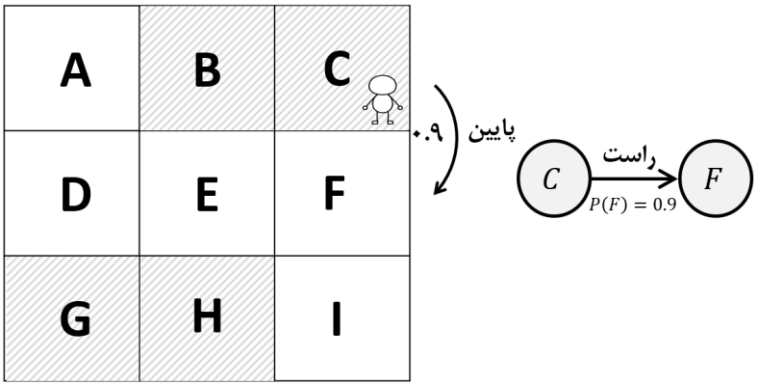
به عنوان مثال ، در محیط مشبک یا جدولی ما، احتمال انتقالِ حرکت از حالت **A** به حالت **B** در حالی که اقدام ما، حرکت به *راست* است برابر ۱۰۰٪ است. این عبارت می‌تواند به صورت $P(B|A,\text{right}) = ۱.۰$ بیان شود. ما همچنین می‌توانیم این را در نمودار حالت مشاهده کنیم، همانطور که در شکل ۱.۱۲ نشان داده شده است:



شکل ۱.۱۲: احتمال انتقال از سمت راست **A** به **B**

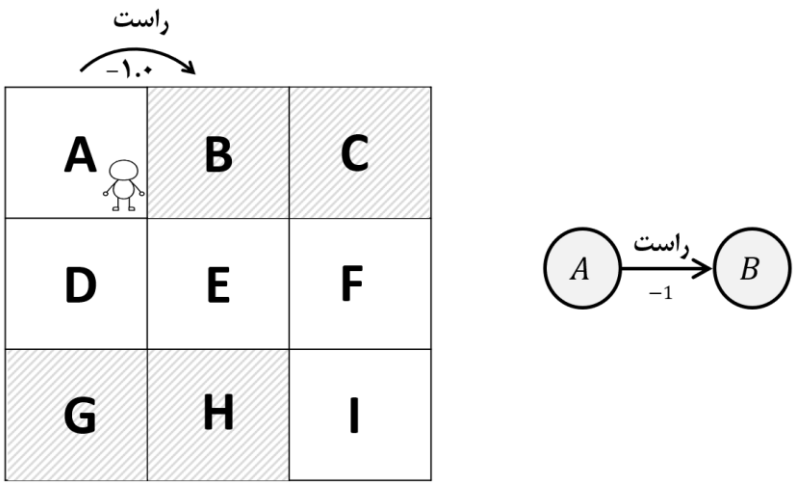
فرض کنید عامل ما در حالت **C** قرار دارد و احتمال انتقال از حالت **C** به حالت **F** در حالی که اقدام آن، حرکت به *پایین* است ۹۰٪ ست، چنین وضعیتی را می توان بصورت $P(F|C,\text{down})$ بیان کرد. ما همچنین می‌توانیم این را در نمودار حالت مشاهده کنیم، همانطور که در شکل ۱.۱۳ نشان داده شده است.

تابع پاداش – تابع پاداش به صورت $R(s,a,s')$ مشخص می‌شود. این تابع، نشان دهنده **پاداش** ما در هنگام گذار یا انتقال از حالت s به حالت s' در حالی است که **اقدام** a را انجام می‌دهیم.



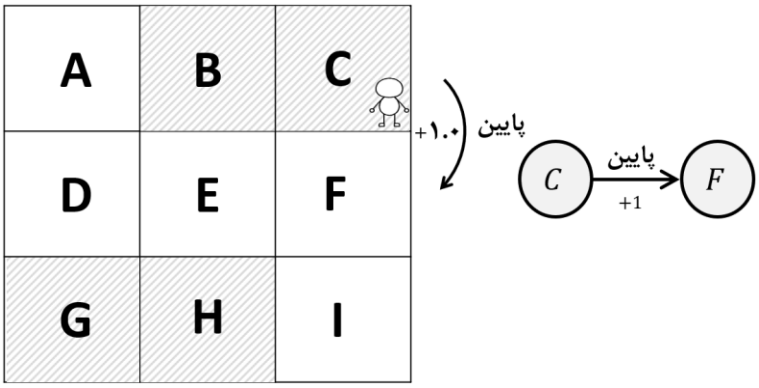
شکل ۱.۱۳: احتمال انتقال با حرکت پایین آمدن از **C** به **F**

اگر پاداشی را که ما هنگام انتقال از حالت **A** به حالت **B** با انجام اقدام حرکت به سمت *راست*، بدست می‌آوریم، منفی یک در نظر بگیریم، پس می‌توان آن را به عنوان $R(A, \text{right}, B) = -۱$ بیان کرد. ما همچنین می‌توانیم این را در نمودار حالت مشاهده کنیم، همانطور که در شکل ۱.۱۴ نشان داده شده است:



شکل ۱.۱۴: **پاداش** حرکت از سمت راست **A** به **B**

فرض کنید عامل ما در حالت **C** است و **پاداش** انتقال از حالت **C** به حالت **F** در حالی که **اقدام** ما حرکت به سمت *پایین* باشد، برابر مثبت یک است. درنتیجه می‌توان آن را به عنوان $R(C, \text{down}, F) = +۱$ بیان کرد. ما همچنین می‌توانیم این موضوع را در نمودار حالت مشاهده کنیم، همانطور که در شکل ۱.۱۵ نشان داده شده است.



شکل ۱.۱۵: **پاداش** حرکت از **C** به **F**

بنابراین، یک محیط یادگیری تقویتی می‌تواند به عنوان مدل **MDP** با چهارگانه: حالتها، **اقدامات**، احتمال انتقال و **تابع پاداش** نشان داده شود. اما صبر کنید! فایدهٔ نمایش محیط یادگیری تقویتی با استفاده از مدل **MDP** چیست؟ جواب این است: هنگامی که محیط خود را به عنوان **MDP** مدل می‌کنیم، می‌توانیم مسئله یادگیری تقویتی را به راحتی حل کنیم. به عنوان مثال، هنگامی که ما محیط جهانی مشبک خود را با استفاده از **MDP** الگوبرداری می‌کنیم، می‌توانیم به راحتی بیابیم که چگونه بدون مراجعه به حالتهای هاشوردار، از حالت **A** به حالت هدف **I** برسیم. ما در فصلهای آینده اطلاعات بیشتری در مورد این موضوع ارائه خواهیم داد. در قسمت بعد، ما مفاهیم اساسی‌تر یادگیری تقویتی را ارائه خواهیم کرد.

مفاهیم اساسی یادگیری تقویتی

در این بخش، ما با چندین مفهوم مهم و اساسی یادگیری تقویتی آشنا خواهیم شد.

الزامات ریاضی

قبل از پیش‌روی بیشتر، بیایید مفهوم امید ریاضی از دوران دبیرستان را مرور کنیم، زیرا ما در طول کتاب با امید ریاضی (میانگین) برخورد خواهیم کرد.

امید ریاضی[۱]

فرض کنیم ما یک متغیر X داریم و این متغیر شامل مقادیر ۱ ، ۲ ، ۳ ، ۴ ، ۵ ، ۶ خواهد شد. برای محاسبه مقدار متوسط X می‌توانیم جمع تمام مقادیر X را بر تعداد مقادیر X تقسیم کنیم. بنابراین، میانگین X برابر $\frac{۱+۲+۳+۴+۵+۶}{۶} = ۳.۵$ است.

حال بیایید فرض کنیم X یک متغیر تصادفی است. متغیر تصادفی (دلبخواه)، مقادیر خود را بر اساس یک آزمایش تصادفی[۲]، مانند پرتاب تاس، پرتاب سکه و غیره می‌گیرد. متغیر تصادفی این مقادیر را با احتمال خاصی اخذ می‌کند. حال تصور کنیم که ما یک تاس تراز و متعادل[۳] را پرتاب می کنیم، در این صورت مقادیر ممکن (X) برابر ۱ ، ۲ ، ۳ ، ۴ ، ۵ و ۶ است و احتمال بروز هر یک از این نتایج، همانطور که در جدول ۱.۲ نشان داده شده، معادل $\frac{۱}{۶}$ است.

X	۱	۲	۳	۴	۵	۶
P(x)	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$

جدول ۱.۲: احتمال دیدن هر عدد بعد از پرتاب تاس

خب حالا چگونه می‌توان مقدار متوسط متغیر تصادفی X را محاسبه کرد؟ از آنجا که هر مقدار احتمال وقوع خاص خود را دارد، ما نمی‌توانیم از روش معمول، میانگین را محاسبه کنیم. در عوض، ما میانگین وزنی را محاسبه می‌کنیم، یعنی مجموع مقادیر X که در احتمالات مربوطه آنها ضرب می شود، و به آن امید گفته می‌شود. امیدِ یک متغیر تصادفی X را می‌توان به این ترتیب تعریف کرد:

---

[۱] Expectation
[۲] Random Experiment
[۳] Fair Dice

$$E(X) = \sum_{i=1}^{N} x_i p(x_i)$$

بنابراین ، امیدِ متغیر تصادفی X برابراست با:

$$E(X) = ۱\left(\frac{۱}{۶}\right) + ۲\left(\frac{۱}{۶}\right) + ۳\left(\frac{۱}{۶}\right) + ۴\left(\frac{۱}{۶}\right) + ۵\left(\frac{۱}{۶}\right) + ۶\left(\frac{۱}{۶}\right) = ۳.۵$$

بعضا بجای واژهٔ امید، از واژه مقدار مورد انتظار[۱] یا مقدار انتظاری نیز استفاده می‌شود. بنابراین، مقدار مورد انتظار متغیر تصادفی X برابر ۳.۵ است. لذا، وقتی می گوییم مقدار انتظاری یا مقدار مورد انتظار یک متغیر تصادفی، اساساً منظورمان، میانگین وزنی است.

حال مقدار انتظاری یک تابع متغیر تصادفی را محاسبه می‌کنیم. فرض کنید $f(x) = x^۲$، پس می‌توان نوشت:

X	۱	۲	۳	٤	٥	٦
f(x)	۱	۴	۹	۱۶	۲۵	۳۶
P(x)	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$	$\frac{۱}{۶}$

جدول ۱.۳: احتمال پرتاب تاس

مقدار انتظاری یک تابع متغیر تصادفی را می‌توان به صورت زیر محاسبه کرد:

$$\mathbb{E}_{x\sim p(x)}[f(X)] = \sum_{i=1}^{N} f(x_i)p(x_i)$$

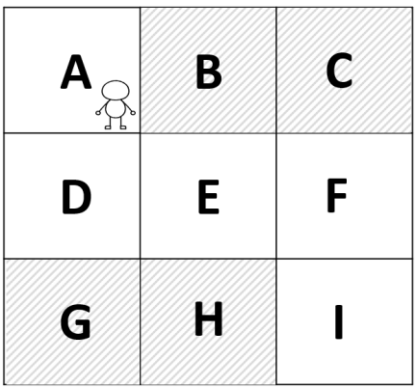
[۱] Expected Value

بنابراین ، مقدار مورد انتظار $f(X)$ برابر خواهد شد با:

$$E(f(X)) = ۱\left(\frac{۱}{۶}\right) + ۴\left(\frac{۱}{۶}\right) + ۹\left(\frac{۱}{۶}\right) + ۱۶\left(\frac{۱}{۶}\right) + ۲۵\left(\frac{۱}{۶}\right) + ۳۶\left(\frac{۱}{۶}\right) = ۱۵.۱$$

فضای کُنش[۱]

محیط جهانی مشبک یا جدولی نشان داده شده در شکل ۱.۱۶ را در نظر بگیرید:



شکل ۱.۱۶: محیط جهانی مشبک یا جدول-گونه

در محیط جهانی مشبک قبلی، هدف عامل این بود که بدون بازدید از حالتهای سایه‌دار، با شروع از حالت **A** به حالت **I** برسد. در هر یک از حالتها ، عامل می‌توانست برای دستیابی به هدف، هر یک از چهار **اقدام**: *بالا*، *پایین*، چپ و *راست* را انجام دهد. مجموعه‌ای از تمام **اقدامات** ممکن در محیط، را فضای عمل، فضای اقدام یا **فضای کُنش** می‌نامند. ما معمولا از ترکیب فضای کنش استفاده می‌کنیم.

[۱] Action Space

بنابراین، برای این محیط جهانی مشبک، فضای کُنش [*بالا* ، *پایین* ، چپ ، *راست*] خواهد بود. ما می توانیم فضاهای کنش را به دو نوع طبقه‌بندی کنیم:

- **فضای کنش گسسته**
- **فضای کنش پیوسته**

فضای کنش گسسته: هنگامی که فضای عمل ما شامل **اقدامات** گسسته است، به آن یک فضای عمل گسسته گفته می‌شود. به عنوان مثال، در محیط جهانی مشبک، فضای عمل ما شامل چهار اقدام گسسته است که *بالا* ، *پایین*، چپ، *راست* است و بنابراین به آن یک فضای عمل گسسته گفته می‌شود.

فضای کنش پیوسته: هنگامی که فضای اقدام ما شامل **اقداماتی** است که پیوسته هستند، آنرا فضای عمل پیوسته نامند. به عنوان مثال، بیایید فرض کنیم که ما یک عامل را برای رانندگی ماشین آموزش می‌دهیم، در این حالت فضای عمل ما متشکل از چندین مورد از اقداماتی خواهد بود که مقادیر پیوسته دارند، مانند: مقدار سرعتی که علائم جاده به ما علامت می‌دهد، مقادیر درجاتی که برای چرخاندن چرخ و غیره لازم داریم. در مواردی که فضای عمل ما شامل اقداماتی است که پیوسته است، به آن یک فضای اقدام پیوسته گفته میشود.

سیاست[۱]

سیاست یا **خطمشی** عملا رفتار عامل را در یک محیط تعریف می‌کند. **این سیاست (یا خطمشی) به عامل می‌گوید که در هر حالت چه اقدامی را انجام دهد.** به عنوان مثال، در محیط جهانی مشبک، ما حالتهای **A** تا **I** و چهار اقدام ممکن را داریم. این سیاست ممکن است به عامل بگوید که در حالت **A** به سمت *پایین* حرکت کند، در حالت **D** به سمت *راست* حرکت کند و امثال آن.

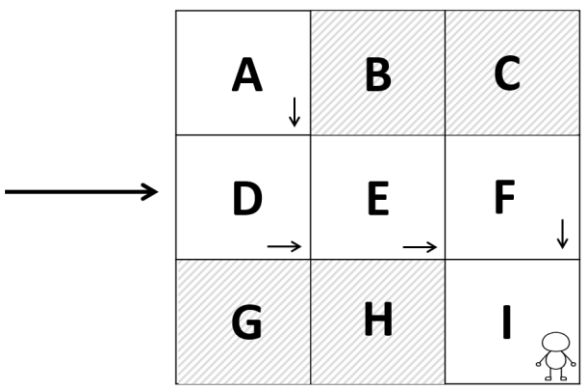
[۱] Policy

در اولین تعامل با محیط، با یک **سیاست** یا **خط‌مشی** تصادفی (دلبخواه) آغاز می‌کنیم، یعنی **سیاست** تصادفی به عامل می‌گوید که در هر حالت یک اقدام تصادفی انجام دهد. بنابراین، در تکرار اولیه، عامل در هر حالت یک عمل یا اقدام تصادفی (بختکی) انجام می‌دهد و سعی می‌کند بر اساس **پاداش** به دست آمده، یاد بگیرد این عمل خوب است یا بد. در طول یک مجموعه از تکرارها، یک عامل یاد می‌گیرد که در هر حالت اقدامات خوبی را انجام دهد، که پاداش مثبتی به همراه داشته باشد. بنابراین، در طی یک سری از تکرارها، عامل سیاست خوبی را یاد می‌گیرد که نتیجتا پاداش مثبت بدست می‌دهد.

این **سیاست** خوب، در اینجا، **سیاست (یا خط‌مشی) بهینه**[۱] نامیده می‌شود. **سیاست** بهینه آن **سیاستی** است که به عامل پاداش خوبی می‌دهد و به عامل کمک می‌کند تا به هدفش برسد. به عنوان مثال، در محیط جهانی مشبک ما، **سیاست** بهینه به عامل می‌گوید که در هر حالت اقدامی را انجام دهد به گونه‌ای که عامل بتواند بدون مراجعه به حالتهای سایه‌دار، از حالت شروع **A** به حالت هدف **I** برسد.

خط‌مشی بهینه در شکل ۱.۱۷ نشان داده شده است. همانطور که می‌توان مشاهده کرد، عامل بر اساس **سیاست** بهینه اقدام را در هر حالت انتخاب کرده و بدون مراجعه به حالتهای سایه‌دار، از حالت شروع **A** به حالت پایانی **I** می‌رسد.

حالت	اقدام
A	پایین
D	راست
E	راست
F	پایین



شکل ۱.۱۷: **سیاست** بهینه در محیط دنیای مشبک

[۱] Optimal Policy

بنابراین، **خط‌مشی** بهینه به عامل می گوید که اقدام صحیح را در هر حالت انجام دهد که در نتیجه آن عامل می تواند پاداش خوبی دریافت کند.

سیاست‌ها (یا **خط‌مشی‌ها**) را می‌توان به صورت زیر طبقه‌بندی کرد:

- **سیاست قطعی (معین)**[۱]
- **سیاست اتفاقی (پیشامدی)**[۲]

سیاست قطعی

به **خط‌مشی** یا **سیاستی** که تاکنون از آن بهره بردیم و در هر گام آنرا بکار بستیم، **سیاست قطعی**[۳] گویند. یک **سیاست قطعی** به عامل می‌گوید که یک اقدام خاصی را در یک حالت معین انجام دهد. بنابراین، **سیاست قطعی**، حالت را به یک اقدام خاص، نگاشت کرده و اغلب با حرف μ نشان داده می‌شود. با فرض قرار داشتن در حالت s در یک زمان، یک **سیاست قطعی** به عامل اعلام می‌کند که اقدام خاص a را انجام دهد. این موضوع می‌تواند به صورت زیر بیان شود:

$$a_t = \mu(S_t)$$

---

[۱] Deterministic
[۲] Stochastic

یادداشت مترجم: ما برای واژه Random از معادلهایی همچون تصادفی، دلبخواه، بختکی و امثال آن استفاده می‌کنیم که معمولا دارای توزیع احتمال مشخصی نیست. اما برای واژه Stochastic از واژه‌های **اتفاقی** و **پیشامدی** بهره می‌گیریم و نشاندهنده یک فرایند غیرقطعی است که دارای تابع **توزیع احتمال** هست. برای واژه Chance نیز از معادلهایی مانند شانس، اقبال و بخت استفاده می‌کنیم که میان دو واژه قبلی قرار می‌گیرد. همچنین برای کلمه Probability از واژه احتمالی استفاده می‌کنیم.

لذا در این کتاب، روش **سیاست تصادفی** (بختکی) با روش **سیاست اتفاقی (پیشامدی)** متفاوت است. دومی یک فرایند مبتنی بر **تابع توزیع احتمال** شناخته شده است ولی اولی نیست.

[۳] Deterministic Policy

به عنوان نمونه، مثال دنیای مشبک یا جدول-گونه را در نظر بگیرید. با توجه به حالت **A**، **سیاست قطعی** μ، به عامل بیان می‌کند که اقدامی یا حرکتی به سمت *پایین* را انجام دهد. این موضوع را می‌توان به صورت زیر بیان نمود:

$$\mu(A) = \text{Down}$$

بنابراین، طبق **سیاست قطعی** ، هر زمان که عامل مورد نظر، حالت **A** را می‌بیند، عمل حرکت به سمت پایین را انجام می‌دهد.

سیاست اتفاقی

بر خلاف یک **سیاست قطعی**، یک **سیاست احتمالی**، حالتی را مستقیماً به یک اقدام خاص نگاشت نمی‌کند، بلکه به جای این کار، حالت مورد نظر را به یک **توزیع احتمال** در فضای اقدام ما، نگاشت می‌کند.

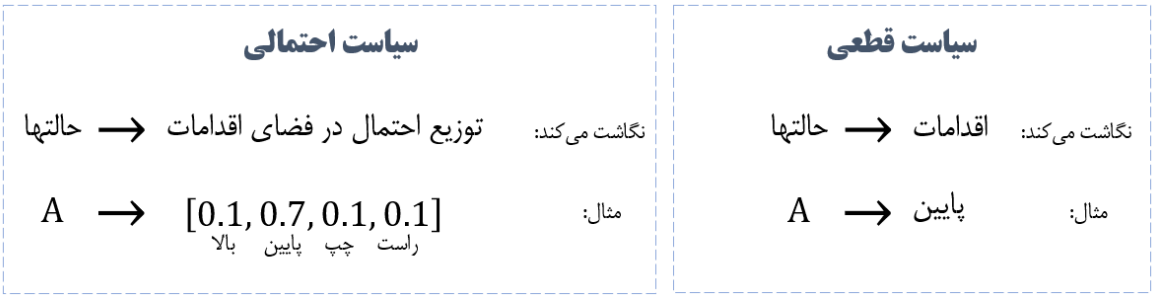
بر اساس آنچه گفته شد، **سیاست قطعی** به عامل می‌گوید که یک اقدام خاص را در یک حالت معین انجام دهد. بنابراین هر زمان که عامل از آن حالت بازدید کند، همیشه، همان عمل خاص را انجام می‌دهد. اما تحت یک **سیاست اتفاقی**، با درنظر گرفتن یک حالت مشخص، **سیاست اتفاقی**، عملا یک **توزیع احتمال** در یک فضای اقدام برمیگرداند. بنابراین عامل، در هربار که از حلت بازدید می‌کند، به جای انجام یک اقدام مشخص، بر اساس توزیع احتمال بازگشت شده توسط **سیاست اتفاقی**، اقدامات مختلفی را انجام می دهد.

بیایید این وضعیت را با یک مثال توضیح دهیم؛ ما می‌دانیم که فضای عمل محیط جهانی مشبک[1] ما، شامل چهار عمل است که [*بالا*، *پایین*، *چپ*، *راست*] هستند. **سیاست اتفاقی**، با توجه به یک حالت **A**، توزیع احتمال را در فضای عمل به صورت [۰.۱, ۰.۱, ۰.۷, ۰.۱] برمی گردلند. اکنون، هر زمان که عامل، حلت **A** را بازدید می‌کند، به جای اینکه هر بار همان عمل خاص را انتخاب کند، عامل ما، ۱۰٪ از اوقات تصمیم حرکت به سمت *بالا*، ۷۰٪ از زمانها،

[1] Grid World Environment's Action Space

تصمیم اقدام به طرف *پایین*، ۱۰٪ مواقع تصمیم عمل به جهت به چپ و ۱۰٪ دفعات تصمیم حرکت به سوی *راست* را اتخاذ می‌کند.

تفاوت بین **سیاست قطعی** و **سیاست اتفاقی** در شکل ۱.۱۸ نشان داده شده است. همانطور که می‌بینیم، **سیاست قطعی**، حالت مورد نظر را به یک اقدام خاص تصویر می‌کند، در حالی که **سیاست اتفاقی**، این حالت را به **توزیع احتمال** در یک فضای کنش نگاشت می‌کند.



شکل ۱.۱۸: تفاوت بین سیاستهای قطعی و اتفاقی (احتمالی)

بنابراین، **سیاست اتفاقی**، حالت را به **توزیع احتمال** در فضای کنش نگاشت نموده و اغلب توسط π نمایش داده می‌شود. لذا برای یک حالت s و اقدام a را در زمان t می‌توانیم **سیاست اتفاقی** را به صورت زیر بیان کنیم:

$$a_t \sim \pi(s_t)$$

بعضا عبارت فوق، بصورت ذیل نیز نمایش داده می شود:

$$\pi(a_t|s_t)$$

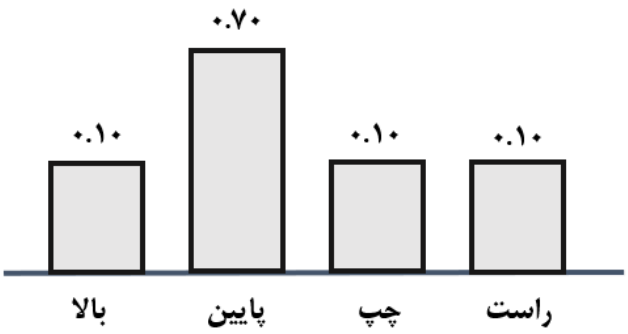
می‌توان **سیاست اتفاقی** را به دو نوع متفاوت دسته‌بندی کرد:

- **سیاست رسته‌ای (یا گسسته)**[۱]
- **سیاست گاوسی (یا پیوسته)**[۲]

[۱] Categorical Policy
[۲] Gaussian Policy

سیاست رسته‌ای

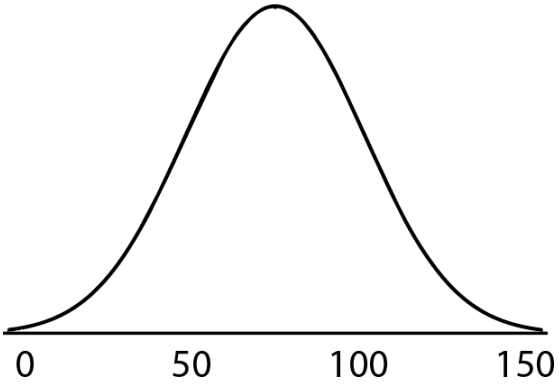
سیاست اتفاقی، را آنگاه یک سیاست رسته‌ای یا طبقه‌ای نامند، که فضای کنش (یا فضای اقدام)، گسسته باشد. یعنی زمانی که فضای کنش گسسته است، سیاست اتفاقی از یک توزیع **احتمال رسته‌ای** که بر روی **فضای کنش** برای انتخاب **اقدامات** تعریف شده، استفاده می کند. به عنوان مثال، در محیط جهان مشبک، ما اقدامات را بر اساس توزیع احتمال رسته‌ای (توزیع گسسته) انتخاب می‌کنیم زیرا فضای کنش محیط، گسسته است. همانطور که شکل ۱.۱۹ نشان می‌دهد، با توجه به حالت **A**، ما اقدامی را بر اساس توزیع احتمال رسته‌ای در فضای کنش، انتخاب می کنیم:



شکل ۱.۱۹: احتمال حرکت بعدی از حالت **A** برای فضای عمل گسسته

سیاست گاوسی

سیاست اتفاقی، زمانی گاوسی نامیده می‌شود که فضای کنش ما پیوسته باشد. یعنی در زمانی که فضای اقدام پیوسته است، سیاست اتفاقی در فضای عمل، از توزیع احتمال گاوسی برای انتخاب **اقدامات** (**کنش‌ها**) استفاده می‌کند. بیایید با یک مثال ساده این موضوع را نشان دهیم. فرض کنید ما در حال آموزش یک عامل برای رانندگی با خودرو هستیم و در واقع، یک اقدام پیوسته در فضای کنش خود داریم. اگر اقدام مورد نظر، سرعت ماشین باشد و مقدار سرعت ماشین بین ۰ تا ۱۵۰ کیلومتر در ساعت باشد، سیاست اتفاقی از توزیع گاوسی در فضای کنش برای انتخاب یک عمل استفاده می‌کند. این موضوع در شکل ۱.۲۰ نشان داده شده است:



شکل ۱.۲۰: توزیع گاوسی

پردینه (اپیزود)[۱]

هر عاملی با انجام اعمال خود، عملا در تعامل با محیط اســت یعنی از حالت اولیه شــروع کرده و به حالت نهایی می‌رســد. این برهمکنش[۲] (یا تعامل) عامل–محیط که از حالت اولیه شــروع می‌شــود تا حالت نهایی وجود دارد، یک **پرده یا اپیزود** نامیده می‌شــود. به عنوان مثال، در یک بازی ویدیویی مســابقه اتومبیل رانی، عامل، بازی را از حالت اولیه (نقطه شروع مسابقه) آغاز می‌کند و به حالت نهایی (نقطه پایان مسابقه) می‌رسد. به کلیه این اتفاقات، یک پرده گفته می‌شود. از آنجا که هر **پرده** یا **پردینه** در درس یادگیری تقویتی، نشاندهنده یک رهسو (یا بخشواره) است، لذا هر پردینه را اغلب یک **مسیر**[۳] هم می‌نامند: یعنی مسیری که عامل طی می‌کند و با τ نشان داده می‌شود.

یک عامل می تواند یک بازی را برای هر تعداد پردینه انجام دهد و هر پردینه مستقل از پرده‌های دیگر است. اجرای بازی برای چند پردینه، چه فایده ای دارد؟ برای یادگیری **سیاست** بهینه، یعنی **سیاستی** که به عامل می‌گوید در هر

[۱] Episode

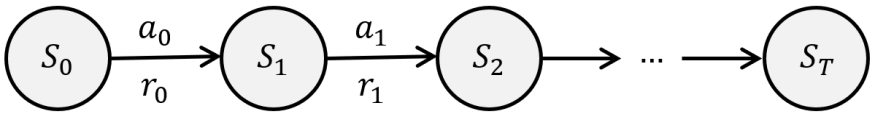
[۲] Interaction
[۳] Trajectory

حالت چه عملی را انجام دهد که آن عمل صحیح است. بنابراین، یک عامل، بازی را برای پرده‌ها (یا اپیزودهای) زیادی انجام می‌دهد.

به عنوان مثال، فرض کنید در حال انجام یک بازی اتومبیل رانی هستیم. بار اول، ممکن است بازی را نبریم، بنابراین چندین بار بازی را انجام می‌دهیم تا بیشتر در مورد بازی بفهمیم و چند استراتژی خوب برای برنده شدن در بازی کشف کنیم. به همین ترتیب ممکن است در رهسو یا پرده اول، عامل بازی را نبرد و در چندین پرده یا پردینه، بازی را اجرا می‌کند تا بیشتر در مورد محیط بازی و استراتژیهای خوب برای بردن بازی یاد بگیرد.

فرض کنید بازی را از یک حالت اولیه در یک گام زمانی $t = ۰$ شروع می کنیم و در گام زمانی T به حالت نهایی می‌رسیم. اطلاعات پرده یا رهسو با شروع از حالت اولیه تا حالت نهایی شامل تعامل عامل- محیط (مانند حالت، عمل، و پاداش) است، یعنی $(S_۰, A_۰, R_۰, S_۱, A_۱, R_۱, \dots, S_T)$.

شکل ۱.۲۱ نمونه ای از یک پرده/مسیر را نشان می دهد:



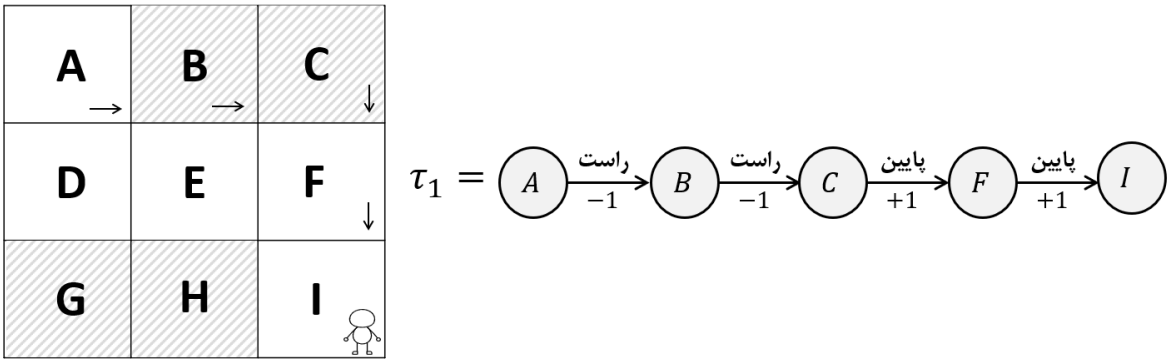
شکل ۱.۲۱: نمونه‌ای از یک پردینه (اپیزود)

بیایید درک خود از پردینه (بخشواره) و **سیاست** بهینه را با مرورمجدد محیط جهان مشبک تقویت کنیم. ما یاد گرفتیم که در محیط جهان مشبک، هدف عامل ما، رسیدن به حالت نهایی **I** با شروع از حالت اولیه **A** بدون دیدار از حالتهای هاشوردار است. یک عامل هنگام بازدید از حالتهای بدون هاشور یک جایزه ۱+ و در صورت بازدید از حالتهای سایه‌دار، پاداش ۱- دریافت می‌کند.

وقتی می‌گوییم یک پردینه (بخشواره) تولید کنید، به معنای رفتن از **حالت اولیهٔ** یک محیط به **حالتِ نهایی** در آن محیط است. عامل مورد نظر، اولین پرده را با استفاده از یک **سیاست تصادفی** (دلبخواه) تولید نموده که در واقع و عملا محیط را کاوش می‌کند. در ادامه و طی چندین پرده، **خطمشی** بهینه را یاد می‌گیرد.

پردینه اول (اپیزود یک)

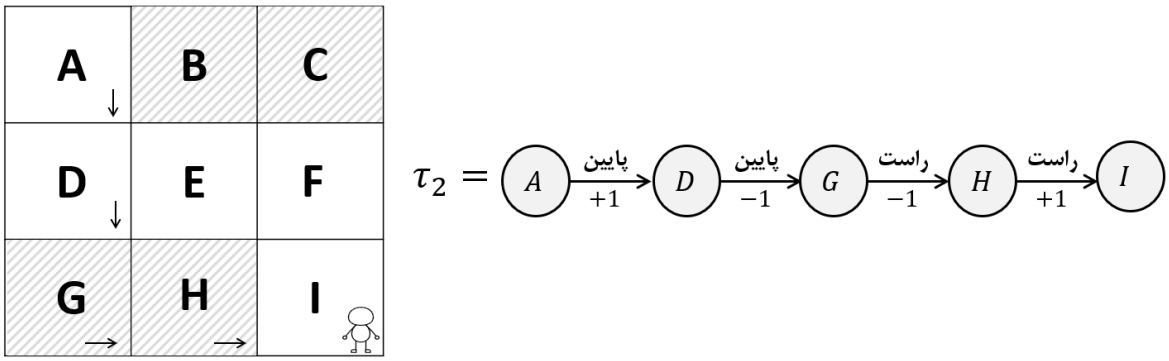
همانطور که شکل ۱.۲۲ نشان می‌دهد، در پرده اول، عامل از یک **خط‌مشی** تصادفی (بختکی) استفاده کرده و یک اقدام تصادفی را در هر حالت، از حالت اولیه تا حالت نهایی، انتخاب کرده و نهایتا **پاداش** را مشاهده می‌کند:



شکل ۱.۲۲: پردینه (اپزود) اول

پردینه دوم (اپیزود دو)

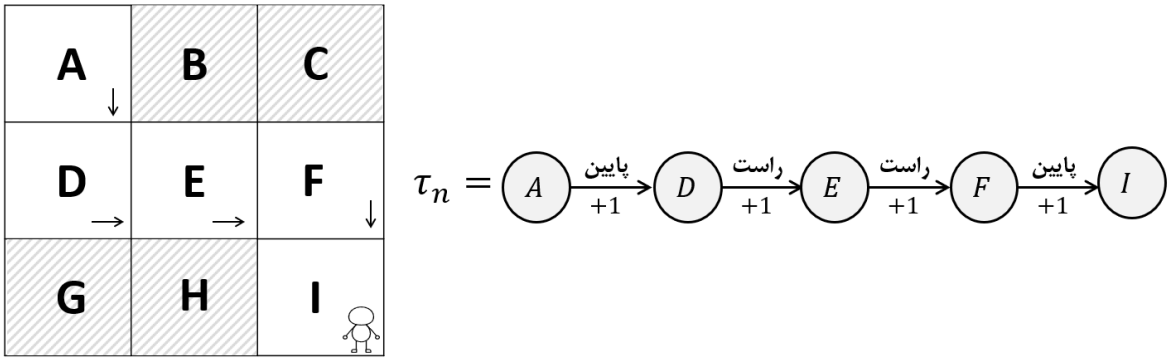
عامل ما در پرده دوم، **سیاست** متفاوتی را امتحان می‌کند تا از پاداش‌های منفی که در پرده قبلی دریافت کرده، اجتناب نماید. به عنوان مثال، همانطور که در قسمت قبل مشاهده کردیم، عامل در حالت **A**، اقدام حرکت به سمت *راست* را انتخاب کرد و یک **پاداش** منفی گرفت، لذا در این پردینه (اپیزود)، به جای انتخاب اقدام حرکت به سمت *راست* در حلت **A**، او تلاش می‌کند که یک اقدام متفاوت را انجام دهد و مثلا به طرف *پایین* حرکت کند، این موضوع در تصویر ۱.۲۳ نمایش داده شده است:



تصویر ۱.۲۳: پردینه (بخشواره) دوم

پردینه آخر (اپیزود n):

بنابراین، در طی یک سری از پرده‌ها، عامل ما، **خطمشی** بهینه را می‌آموزد، یعنی آن **سیاستی** که عامل را از حالت **A** بدون بازدید از حالت‌های سایه‌دار به حالت نهایی **I** می‌برد، همانطور که شکل ۱.۲۴ نشان می‌دهد:



تصویر ۱.۲۴: پردینه (بخشواره) n ام

وظایف پردینه‌ای و غیرپردینه‌ای

انجام وظیفه[۱] در حوزه یادگیری تقویتی را می‌توان به صورت زیر دسته‌بندی کرد:

- **وظایف پردینه‌ای (اپیزودیک)[۲]**
- **وظایف غیر پردینه‌ای (پیوسته)[۳]**

وظیفه پردینه‌ای: همانطور که از نام آن پیداست، وظیفه پردینه‌ای (اپیزودیک)، کاری است که یک حالت نهایی یا پایانی دارد (از ابتدا تا انتهای کار، در یک پرده، تمام می‌شود). یعنی وظایف پردینه‌ای وظایفی هستند که از پرده‌های مختلف تشکیل شده و بنابراین حالت پایانی دارند. به عنوان مثال در یک بازی اتومبیلرانی از نقطه شروع (وضعیت اولیه) شروع کرده و به مقصد (حالت پایانی) می‌رسیم.

وظیفه پیوسته: برخلاف وظایف پردینه‌ای، وظایف پیوسته شامل هیچ پردینه‌ای (یا اپیزودی) نمی‌شوند و بنابراین حالت پایانی ندارند. به عنوان مثال، یک ربات دستیار شخصی، حالت پایانی ندارد.

افق

افق، یک مدت یا گام زمانی است که تا آن زمان، عامل با محیط تعامل دارد. افق را می‌توان به دو دسته تقسیم کرد:

- **افق محدود**
- **افق نامحدود**

افق محدود: اگر برهمکنش عامل و محیط در یک گام زمانی خاص، متوقف شود، آن افق را افق محدود نامند. به عنوان مثال، در وظایف پردینه‌ای، یک عامل با شروع از حلت اولیه و در گام زمانی[۴] $t = ۰$ تعامل با محیط را

---

۱ Task
۲ Episodic Tasks
۳ Continuous Tasks
۴ Time Step

شروع کرده و در گام زمانی T به حالت نهایی می‌رسد. از آنجایی که تعامل عامل – محیط در گام زمانی T متوقف می‌شود، درواقع یک افق محدود را در نظر گرفته‌ایم.

افق نامحدود: اگر تعامل عامل و محیط هرگز متوقف نشود، آن را افق نامحدود می نامند. به عنوان مثال، ما آموختیم که یک وظیفه پیوسته، هیچ حالت پایانی ندارد. این بدان معنی است که تعامل عامل– محیط، هرگز در یک وظیفه مداوم متوقف نمی‌شود و بنابراین یک افق بی‌نهایت در نظر گرفته می‌شود.

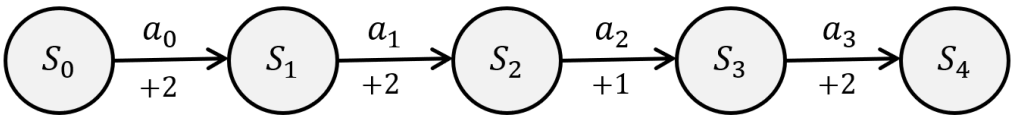
بازگشت و نرخ تنزیل (ضریب تخفیف)

بازده یا **بازگشت** را می توان به عنوان مجموع **پاداشهای** بدست آمده توسط عامل در یک پردینه (اپیزود) تعریف کرد. **بازگشت**، اغلب با R یا G نشان داده می شود. فرض کنید عامل از حالت اولیه و در گام زمانی $t = ۰$ شروع می‌کند و در گام زمانی T به حالت نهایی می‌رسد، در این صورت، **بازده** به دست آمده توسط عامل به صورت زیر نوشته می‌شود:

$$R(\tau) = r_o + r_۱ + r_۲ + \cdots + r_T$$

$$R(\tau) = \sum_{t=0}^{T} r_t$$

بیایید، این را با یک مثال درک کنیم. مسیر (اپیزود) τ را در نظر بگیرید:



شکل ۱.۲۵: پرده/مسیر τ

پیامد یا **بازده** مسیر، مجموع پاداشهاست، یعنی،

$$R(\tau) = ۲ + ۲ + ۱ + ۲ = ۷$$

بنابراین، می‌توان گفت که هدف عامل، حداکثر کردن **مقدار بازده** (بازگشت یا پیامد)[۱] است، یعنی حداکثر کردن مجموع **پاداش‌های** (پاداش‌های تجمعی) به‌دست‌آمده در یک پردینه (اپیزود). چگونه می توانیم بازده را به حداکثر برسانیم؟ اگر اقدام درست را در هر حالت انجام دهیم، می توانیم بازده را به حداکثر برسانیم. خوب، چگونه می توانیم اقدام صحیح را در هر حالت انجام دهیم؟ با استفاده از **خط‌مشی** بهینه می‌توانیم در هر حالت، عمل صحیح را انجام دهیم. بنابراین، ما می توانیم بازده را با استفاده از **سیاست** بهینه به حداکثر برسانیم. بنابراین، **سیاست** بهینه، **سیاستی** است که با انجام اقدام صحیح در هر حالت، حداکثر **بازده** (**مجموع پاداش**) را به عامل ما می‌رساند.

خوب، چگونه می توانیم **بازده** را برای وظایف پیوسته تعریف کنیم؟ یاد گرفتیم که در وظایف پیوسته هیچ حالت پایانی وجود ندارد، بنابراین می‌توانیم **بازده** را به عنوان مجموع پاداشها تا بی‌نهایت، تعریف کنیم:

$$R(\tau) = r_o + r_۱ + r_۲ + \cdots + r_\infty$$

اما چگونه می‌توانیم بازگشت را که تا بی نهایت جمع می‌شود، به حداکثر برسانیم؟ لذا یک اصطلاح جدید به نام **نرخ تنزیل** یا **ضریب تخفیف**[۲] γ تعریف کرده و بازگشت را به صورت زیر بازنویسی می‌کنیم:

$$R(\tau) = \gamma^o r_o + \gamma^۱ r_۱ + \gamma^۲ r_۲ + \cdots + \gamma^n r_\infty$$

$$R(\tau) = \sum_{t=0}^{\infty} \gamma^t r_t$$

---

[۱] Return
[۲] Discount Factor

خیلی هم خب، اما این ضـریب تخفیف γ چگونه به ما کمک می‌کند؟ این ضـریب، با تصـمیم‌گیری در مورد نحوه اهمیت دادن به پاداشـهای آینده و پاداشـهای جدیدتر (فوری)، به ما در جلوگیری از رسـیدن بازگشـت به بی نهایت، کمک می‌کند. مقدار ضـریب تخفیف (تنزیل) از ۰ تا ۱ متغیر اسـت. وقتی ضـریب تخفیف را روی یک مقدار کوچک (نزدیک به ۰) تنظیم می‌کنیم، به این معنی است که، به پاداشهای جدیدتر (فوری) نسبت به پاداشهای آینده، اهمیت بیشتری می‌دهیم. وقتی ضریب تخفیف (تنزیل) را روی مقدار بالا (نزدیک به ۱)، تنظیم کنیم، به این معنی است که به پاداشهای آینده اهمیت بیشتری می‌دهیم. بیایید این را با یک مثال و با مقادیر مختلف ضریب تخفیف، درک کنیم.

ضریب تخفیف کوچک[1]

بیایید ضـریب تخفیف (نرخ تنزیل) را برابر یک مقدار کوچک قرار دهیم، فرضـاً ۰.۲، یعنی مثلا قرار دهیم: $\gamma = ۰.۲$، سپس می‌توانیم بنویسیم:

$$\begin{aligned} R &= (\gamma)^{۰} r_۰ + (\gamma)^{۱} r_۱ + (\gamma)^{۲} r_۲ + \cdots \\ &= (۰.۲)^{۰} r_۰ + (۰.۲)^{۱} r_۱ + (۰.۲)^{۲} r_۲ + \cdots \\ &= (۱) r_۰ + (۰.۲) r_۱ + (۰.۰۴) r_۲ + \cdots \end{aligned}$$

از این معادله می‌توان مشاهده کرد که پاداش، در هر گام زمانی با یک ضریب تخفیف، وزن دهی می‌شود. با افزایش گامهای زمانی، ضـریب تخفیف (وزن) کاهش یافته و در نتیجه اهمیت پاداشـها در گامهای زمانی آینده نیز کاهش می‌یابد. یعنی از این معادله می‌توان مشاهده کرد که:

- در مرحله زمانی صفر، پاداش $r_۰$ با ضریب تنزیل یک وزن‌دهی می‌شود.
- در مرحله زمانی یک، پاداش $r_۱$ با ضریب تنزیل به شدت کاهش یافته ۰.۲ وزن‌دهی می‌شود.
- در مرحله زمانی دو، پاداش $r_۲$ با یک ضریب تنزیل به شدت کاهش یافته ۰.۰۴ وزن‌دهی می‌شود.

[1] Small Discount Factor

همانطور که مشـــاهده می‌کنیم، ضـــریب تخفیف (نرخ تنزیل) برای گامهای زمانی بعدی به شـــدت کاهش یافته و اهمیت بیشتری به پاداش فوری r_0، نسبت به پاداشهای به دست آمده در مراحل زمانی آینده، داده می‌شود. بنابراین، وقتی ضـــریب تنزیل را روی یک مقدار کوچک قرار می‌دهیم، نســـبت به پاداشـــهای آینده، به پاداش فوری اهمیت بیشتری می‌دهیم.

ضریب تخفیف بزرگ[۱]

حال، ضـــریب تخفیف (نرخ تنزیل) را روی یک مقدار بالا تنظیم می‌کنیم، فرضـــاً اگر ۰.۹ را درنظر بگیریم و قرار بدهیم، $\gamma = ۰.۹$، می‌توانیم بنویسیم:

$$\begin{aligned} R &= (\gamma)^0 r_0 + (\gamma)^1 r_1 + (\gamma)^2 r_2 + \cdots \\ &= (۰.۹)^0 r_0 + (۰.۹)^1 r_1 + (۰.۹)^2 r_2 + \cdots \\ &= (۱) r_0 + (۰.۹) r_1 + (۰.۸۱) r_2 + \cdots \end{aligned}$$

از این معادله می توان نتیجه گرفت که با افزایش گام زمانی، ضریب تنزیل (وزن) کاهش می‌یابد. اما در این وضعیت، به شدت گذشته کاهش نمی یابد زیرا در اینجا ما با $\gamma = ۰.۹$ شروع کردیم. پس در این صورت می توان گفت که به پاداشهای آینده اهمیت بیشتری می‌دهیم. یعنی از معادله می توان مشاهده کرد که:

- در مرحله زمانی صفر، پاداش r_0 با ضریب تخفیف ۱ وزن می‌شود.
- در مرحله زمانی یک، پاداش r_1 با ضریب تخفیف اندکی کاهش یافته ۰.۹ وزن‌دهی می‌شود.
- در مرحله زمانی دو، پاداش r_2 با ضریب تخفیف اندکی کاهش یافته ۰.۸۱ وزن‌دهی می‌شود.

[۱] Large Discount Factor

همانطور که مشـــاهده می‌کنیم، ضـــریب تخفیف، برای گامهای زمانی بعدی کاهش میابد، اما بر خلاف مورد قبلی، ضـــریب تخفیف به شـــدت کاهش نمی‌یابد. بنابراین، وقتی ضـــریب تخفیف را روی مقدار بالایی تنظیم می‌کنیم، به پاداشهای آینده نسبت به پاداش فوری، اهمیت بیشتری می‌دهیم.

وقتی فاکتور تخفیف را برابر صفر قرار دهیم، چه اتفاقی می‌افتد؟

وقتی ضـریب تخفیف (نرخ تنزیل) را برابر صـفر قرار می‌دهیم، یعنی $\gamma = ۰$، به این معنی اسـت که ما فقط پاداش فوری r_o را در نظر می‌گیریم و پاداش به دســـت آمده از گامهای زمانی آینده را درنظر نمی‌گیریم. بنابراین، وقتی ضـــریب تخفیف را روی صـــفر، تنظیم می‌کنیم، عامل هرگز یاد نمی‌گیرد زیرا او فقط پاداش فوری r_o را در نظر می‌گیرد. این وضوع در اینجا نشان داده شده است:

$$\begin{aligned} R &= (\gamma)^o r_o + (\gamma)^۱ r_۱ + (\gamma)^۲ r_۲ + \cdots \\ &= (o)^o r_o + (۰)^۱ r_۱ + (o)^۲ r_۲ + \cdots \\ &= (۱) r_o + (o) r_۱ + (۰) r_۲ + \cdots \\ &= r_o \end{aligned}$$

همانطور که مشاهده می‌کنیم، وقتی $\gamma = ۰$ را تنظیم می‌کنیم، بازگشت یا بازده ما فقط پاداش فوری $r.$ خواهد بود.

وقتی فاکتور تخفیف را برابر یک قرار می دهیم چه اتفاقی می افتد؟

وقتی ضـــریب تخفیف (نرخ تنزیل) را برابر یک قرار می‌دهیم، یعنی: $\gamma = ۱$ ، به این معنی اســـت که ما همه پاداشـــهای آینده را در نظر می‌گیریم. بنابراین، هنگامی که ضـــریب تخفیف را برابر یک قرار می‌دهیم ، عامل برای همیشـه یاد می‌گیرد و به دنبال تمام پاداشـهای آینده است که ممکن است به بی‌نهایت منجر شـود. این موضـوع در اینجا نشان داده شده است:

$$\begin{aligned} R &= (\gamma)^o r_o + (\gamma)^۱ r_۱ + (\gamma)^۲ r_۲ + \cdots \\ &= (۱)^o r_o + (۱)^۱ r_۱ + (۱)^۲ r_۲ + \cdots \\ &= r_o + r_۱ + r_۲ + \cdots \end{aligned}$$

همانطور که مشاهده می کنیم، وقتی قرار می دهیم $\gamma = ۱$، بازده یا بازگشت ما مجموع پاداشها تا بی‌نهایت خواهد بود. بنابراین، ما آموختیم که وقتی ضـریب تخفیف را روی صـفر تنظیم کنیم، عامل هرگز یاد نمی‌گیرد و تنها با در نظر گرفتن پاداش فوری عمل می‌کند، و وقتی ضـریب تخفیف را روی یک تنظیم کنیم، عامل برای همیشـه یاد می‌گیرد و به دنبال پاداشهای آینده است که منجر به میل به بی‌نهایت می‌شود. بنابراین، مقدار بهینه ضریب تخفیف بین ۰.۲ و ۰.۸ قرار دارد.

چرا هم پاداشهای فوری و هم پاداشهای آینده اهمیت دارند؟

ما بسـته به وظایف، به پاداشـهای فوری و آینده اهمیت می‌دهیم. در برخی از وظایف، پاداشـهای آینده مطلوب تر از پاداشـهای فوری هسـتند و بالعکس. در یک بازی شـطرنج، هدف ما شـکسـت دادن شـاه حریف اسـت. اگر به پاداش فوری اهمیت بیشـتری بدهیم، مثلا پاداشـی که با اقداماتی مانند شـکسـت دادن هر مهره شـطرنج حریف به دسـت می‌آید، آنگاه عامل به جای یادگیری هدف واقعی، اجرای این هدف فرعی را یاد می‌گیرد. بنابراین، در این مورد، ما به پاداشـهای آینده اهمیت بیشـتری نسـبت به پاداش فوری می‌دهیم، در حالی که در برخی موارد، نیاز اسـت پاداشـهای فوری را بر پاداشهای آینده ترجیح بدهیم. مثلا ترجیح می دهید امروز به شما شکلات داده بشود یا ۱۳ روز دیگر؟

در دو بخش بعدی، دو تابع اساسی یادگیری تقویتی را تحلیل خواهیم کرد.

تابع ارزش

تابع ارزش که **تابع ارزش حالت**[۱] نیز نامیده می شـود، ارزش یک حالت خاص را نشـان می دهد. ارزش یک حالت، **بازده** یا **بازگشتی** است که یک عامل با شروع از آن حالت و با پیگیری **سیاست** π دریافت می‌کند.

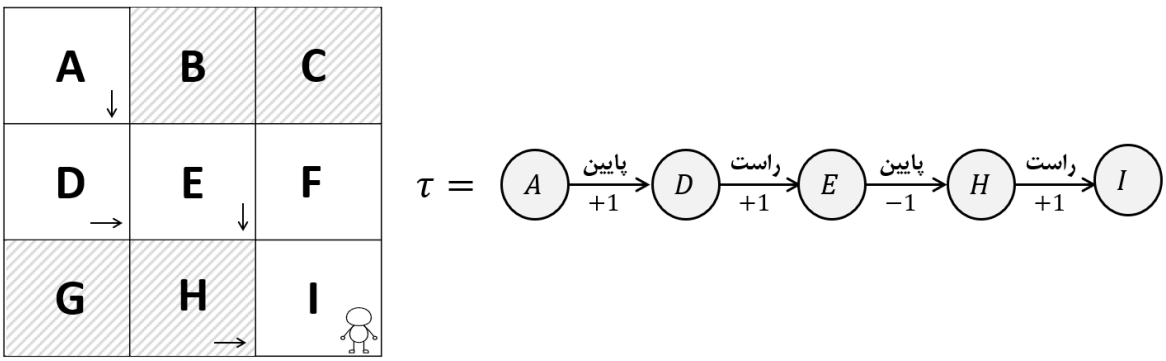
[۱] Value Function

ارزش یک حالت یا **تابع ارزش** معمولاً با $V(s)$ نشان داده می‌شود و می توان آن را به صورت زیر بیان کرد:

$$V^{\pi}(s) = [R(\tau)|s_o = s]$$

که در آن $s_o = s$ است و نشان می دهد که حالت شروع s است. به ارزش یک حالت، بعضا **حالت–ارزش** گفته می‌شود.

بیایید تابع ارزش را با طرح یک مثال درک کنیم. فرض کنید که طبق شکل ۱.۲۶، مسیر τ را با پیروی از سیاست π در محیط جهان شبکه خود، ایجاد می کنیم:



شکل ۱.۲۶: مثال **تابع ارزش**

حال، چگونه ارزش تمام حالت‌های موجود در مسیرمان را محاسبه کنیم؟ ما یاد گرفتیم که **ارزش** یک حالت، بازده بازگشتی است: یعنی مجموع پاداش‌های که یک عامل، که با شروع از آن حالت، طبق **سیاست** π، به دست می‌آورد. مسیر قبلی با استفاده از **خط‌مشی** π تولید می‌شود، بنابراین می توانیم بگوییم که ارزش یک حالت، بازده یا بازگشت مجموع پاداشهای **مسیری** است که از آن حالت شروع می‌شود:

- **ارزش** حالت **A** بازگشت یا **بازدهٔ مسیری** است که از حالت **A** شروع می‌شود. لذا:

$$V(\mathbf{A}) = ۱ + ۱ + (-۱) + ۱ = ۲$$

- **ارزش** حالت **D** **بازدهٔ مسیری** است که از حالت **D** شروع می‌شود. لذا:

$$V(\mathbf{D}) = ۱ - ۱ + ۱ = ۱$$

• **ارزش** حالت **E** **بازدهٔ مسیری** است که از حالت **E** شروع می‌شود. لذا:

$$V(\mathbf{E}) = -۱ + ۱ = ۰$$

• **ارزش** حالت **H** **بازدهٔ مسیری** است که از حالت **H** شروع می‌شود. لذا:

$$V(\mathbf{H}) = ۱$$

در مورد **ارزش** حالت نهایی **I** چطور؟ ما آموختیم که **ارزش یک حالت، بازگشت یا بازده (مجموع پاداشها) است که از آن حالت شروع می شود.** ما می‌دانیم که وقتی از حالتی به حالت دیگر منتقل می‌شویم، پاداشی به دست می‌آوریم. از آنجایی که **I** حالت نهایی است، ما هیچ انتقالی از حالت نهایی انجام نمی‌دهیم، بنابراین هیچ پاداشی و در نتیجه هیچ ارزشی برای حالت نهایی **I** وجود ندارد.

مخلص کلام آنکه، ارزش یک حالت، بازگشت پاداشتهای مسیری است که از آن حالت خاص شروع می‌شود.

اما صبرکنید! یک تغییر کوچک در اینجا وجود دارد: به جای اینکه **بازده** را مستقیماً به عنوان ارزش یک حالت در نظر بگیریم، ما گاهی از **بازده مورد انتظار**[۱] استفاده خواهیم کرد. بنابراین، **تابع ارزش** یا **ارزش حالت** s، را می‌توان به عنوان بازده مورد انتظاری که عامل با شروع از حالت s و دنبال کردن **خطمشی** π به دست می‌آورد، تعریف کرد. این وضعیت را می‌توان به صورت زیر بیان کرد:

$$V^{\tau}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_{\cdot} = s]$$

$$V^{\tau}(s) = \mathop{\mathbb{E}}_{\tau \sim \pi}[R(\tau)|s_o = s]$$

[۱] Expected Return

حال، سوال اینجاست که چرا باید بازده مورد انتظار را در نظر بگیریم؟ چرا نمی‌توانیم ارزش یک حالت را مستقیماً به عنوان بازده (بازگشت) محاسبه کنیم؟ جواب آنست که بازده، متغیر تصادفی است و ارزشهای مختلفی را با احتمال مختلفی می‌گیرد.[۱]

بیایید با یک مثال ساده این موضوع را بررسی کنیم. فرض کنید ما یک **سیاست اتفاقی** π داریم. آموختیم که بر خلاف **خطمشی قطعی**، که حالت را مستقیماً به اقدام نگاشت می‌کند، **خطمشی اتفاقی**، حالت را به توزیع احتمال در فضای کنش مرتبط یا نگاشت می‌کند. بنابراین، **سیاست اتفاقی**، اقدامات را بر اساس توزیع احتمال انتخاب می‌کند.

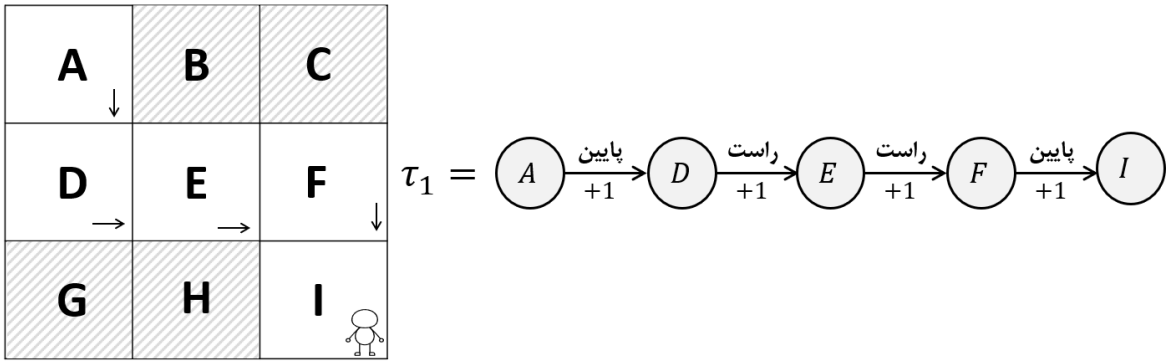
فرض کنید ما در حالت **A** هستیم و **خطمشی اتفاقی**، توزیع احتمال را در فضای عمل به صورت $[۰.۰, ۰.۸۰, ۰.۰۰, ۰.۲۰]$ برمی‌گرداند. این بدان معناست که با **سیاست اتفاقی**، در حالت **A**، اقدام حرکت به *پایین* را ۸۰ درصد مواقع انجام می‌دهیم، یعنی، $\pi(\text{down}|A) = ۰.۸$، و اقدام حرکت به سمت *راست* ۲۰ درصد مواقع انجام می‌شود یعنی، $\pi(\text{right}|A) = ۰.۲۰$.

بنابراین، در حالت **A**، تحت **سیاست اتفاقی** π، در ۸۰٪ مواقع، عامل ما حرکت به سمت *پایین* و در ۲۰٪ مواقع عامل ما حرکت به سمت *راست* را انتخاب می‌کند. به بیان دیگر، **سیاست احتمالی**، اقدام حرکت به سمت راست را در حالتهای **D** و **E** و عمل حرکت به طرف *پایین* را در حالت های **B** و **F** را در ۱۰۰ درصد مواقع، انتخاب می‌کند.

ابتدا، همانطور که در شکل ۱.۲۷ نشان داده شده است، یک اپیزود $\tau_۱$ را با استفاده از **سیاست اتفاقی** π تولید می‌کنیم.

---

[۱] **یادداشت مترجم:** همانطور که مشاهده می‌کنید ما تلاش داریم در متن کتاب، واژگان **سیاست** و نتایج بکارگیری هر **سیاست** یعنی **تابع ارزش** و **پاداشهای** هر پرده (اپیزود) را با دو فونت متفاوت نشان دهیم. حقیقت آنست که در فصول نخستین کتاب، فهم کارکرد و مدل ریاضی این دو (**سیاست** و **بازده**) ساده است اما در فصول پایانی و با مطرح شدن شبکه‌های عصبی، هر دوی اینها در قالب یک یا چند شبکه عمل می‌کنند که فهمیدن مطالب خوانده شده بدون درک نقش این دو، کمی مشکل می‌شود.

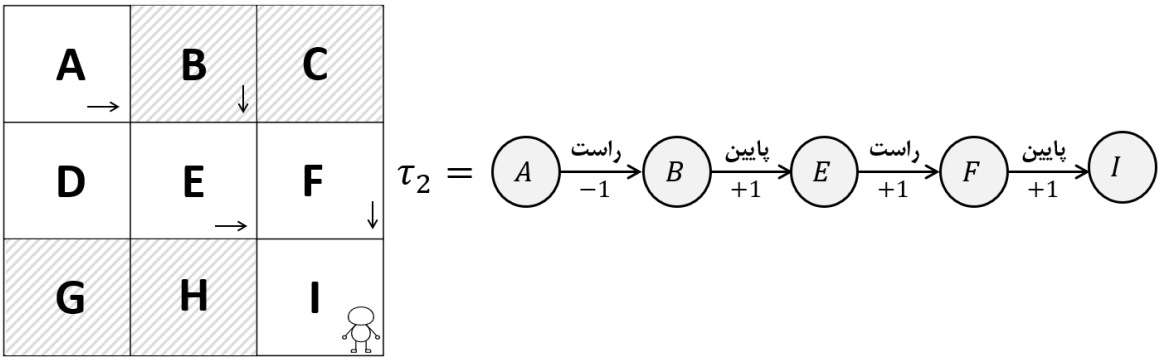


شکل ۱.۲۷: پردینه (اپیزود) τ_1

برای درک بهتر، اجازه دهید فقط روی **ارزش** حالت **A** تمرکز کنیم. ارزش حالت **A**، بازگشت مسیر (یا جمع پاداشهای مسیر) با شروع از حالت **A** است. لذا داریم:

$$V(A) = R(\tau_1) = 1 + 1 + 1 + 1 = 4$$

با توجه به شکل ۱.۲۸ فرض کنید ما یک پرده دیگر τ_2 را با استفاده از **سیاست احتمالی** π مشابه قبل، ایجاد کنیم.



شکل ۱.۲۸: پردینه (اپیزود) τ_2

ارزش حالت **A**، بازگشت (**مجموع پاداشهای**) مسیر از حالت **A** است، بنابراین،

$$.V(\mathbf{A}) = R(\tau_2) = -1 + 1 + 1 + 1 = 2$$

همانطور که مشاهده می‌کنید، گرچه ما از **سیاست** یکسانی استفاده کردیم ولی ارزشهای حالت **A** در مسیرهای τ_1 و τ_2 متفاوت شد. چرا چنین شد؟ این نتیجه بدین دلیل است که **سیاست** ما، یک **سیاست اتفاقی** است و اقدام حرکت به *پایین* در حالت **A**، در ۸۰٪ مواقع و اقدام حرکت به سمت *راست* در حالت **A**، در ۲۰ درصد مواقع، انجام می‌شود. بنابراین، مقدار **بازده** یا **بازگشت** در ۸۰ درصد مواقع برابر ۴ و در ۲۰ درصد مواقع برابر ۲ خواهد بود. بنابراین، به جای اینکه ارزش حالت را مستقیماً به عنوان بازده در نظر بگیریم، **بازده** مورد انتظار را درنظر می‌گیریم، زیرا **بازده**، با احتمال کمی مقادیر مختلفی را می‌گیرد. **بازده** مورد انتظار اساساً میانگین وزنی است، یعنی **مجموع بازگشت**، در احتمال آنها ضرب می‌شود. بنابراین، می‌توان نوشت.

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_o = s]$$

مقدار یک حالت **A** را می‌توان به صورت زیر بدست آورد:

$$\begin{aligned} V^{\pi}(\mathbf{A}) &= \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_o = A] \\ &= \sum_i R(\tau_i)\pi(a_i|A) \\ &= R(\tau_1)\pi(\text{down}|A) + R(\tau_2)\pi(right|A) \\ &= ۴(۰.۸) + ۲(۰.۲) \\ &= ۳.۶ \end{aligned}$$

بنابراین، **ارزش** یک حالت، **مقدار بازده** مورد انتظار مسیری است که از آن حالت شروع می شود. توجه داشته باشید که، **تابع ارزش** به **خط‌مشی** بستگی دارد، یعنی **ارزش** حالت، بر اساس **سیاست** انتخاب شده، متفاوت است. با توجه به **خط‌مشی‌های** مختلف، توابع ارزش زیادی می‌تواند وجود داشته باشد. **تابع ارزش** بهینه $V^*(s)$، حداکثر ارزش را در مقایسه با تمام توابع ارزش دیگر، به دست می‌دهد. می توان این تابع را به صورت زیر بیان کرد:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

به عنوان مثال، فرض کنید دو **سیاست** π_1 و π_2 را داریم. ارزش حالت s با استفاده از **خط‌مشی** π_1، برابر خواهد بود با $V^{\pi_1}(s) = ۱۳$ و ارزش حالت s با استفاده از **سیاست** π_2 برابر $V^{\pi_2}(s) = ۱۱$ می‌باشد. پس **ارزش** بهینه حالت s برابر $V^*(s) = ۱۳$ و برابر مقدار بیشینه آن خواهد بود. **سیاستی** که بیشینه **ارزش** حالت را می‌دهد، **سیاست** یا **خط‌مشی بهینه** π^* نامیده می‌شود. بنابراین، در این مورد، π_1 عملا **سیاست** بهینه[۱] است، زیرا حداکثر **ارزش** حالت را می‌دهد.

ما می‌توانیم **تابع ارزش** را در جدولی به نام **جدول ارزش** مشاهده کنیم. فرض کنید دو حالت s_o و s_1 داریم، سپس **تابع ارزش** را می توان به صورت زیر نمایش داد:

حالت	ارزش
s_o	۷
s_1	۱۱

جدول ۱.۴: **جدول ارزش**

با نگاه به جدول ارزش، می‌توان گفت که بهتر است در حالت s_1 باشیم تا حالت s_o، چرا که ارزش s_1 مقدار بیشتری دارد. بنابراین می‌توان گفت که حالت s_1 ، حالت بهینه است.

---

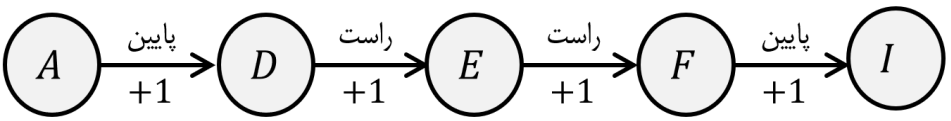
[۱] **یادداشت مترجم:** با ادبیات دنیای ریاضیات دبیرستان، در واقع، **سیاست**: همان **دامنه** و **ارزش**: همان **بُرد** است که **تابع ارزش**، مقادیر **دامنه** را با تغییراتی به **بُرد** نگاشت می‌کند؛ یعنی y=f(x) که در آن x مسیر (مجموعه اقدامات) تحت یک **سیاست** است و y **ارزش** آن مسیر است. البته مدنظرتان باشد که ما با مسئله بهینه‌سازی سروکار داریم که در آن، بدنبال x* (**مسیر بهینه**) و همچنین y* (**ارزش بهینه**) هستیم که به آنها **جواب بهینه** و **ارزش بیشینه** گفته می‌شود.

تابع Q

تابع Q، که تابع **ارزش** حالت–اقدام[۱] نیز نامیده می‌شود، **ارزش** یک جفت (حالت–اقدام) است. **ارزش** یک جفت حالت–اقدام، مقدار **بازدهی** است که عامل با شروع از حالت s و انجام عمل a با پیگیری **سیاست** π، به صورت زیر بدست می‌آورد. **ارزش** یک جفت (حالت–اقدام)، یا **تابع** Q معمولاً با $Q(s,a)$ نشان داده می‌شود و به عنوان **ارزش** Q، یا **ارزش** (حالت–اقدام)، شناخته می‌شود. این عبارت، به صورت زیر بیان می‌شود:

$$Q^{\pi}(s,a) = [R(\tau)|s_o = s,\ \ a_o = a]$$

توجه داشته باشید که تنها تفاوت بین **تابع ارزش** و **تابع** Q این است که در **تابع ارزش**، کل ارزش یک حالت را محاسبه می‌کنیم، در حالی که در **تابع** Q، صرفا ارزش یک جفت حالت–اقدام را محاسبه می‌کنیم. بیایید تابع Q را با یک مثال درک کنیم. مسیر شکل ۱.۲۹ که با استفاده از **خط‌مشی** π، تولید شده را در نظر بگیرید:



شکل ۱.۲۹: یک نمونه مسیر/ پردینه

ما آموختیم که **تابع** Q، **ارزش** یک جفت (حالت–اقدام) را محاسبه می‌کند. فرض کنید که می‌خواهیم **ارزش** Q را برای جفت حالت–اقدامِ **A** یعنی حرکت به سمت *پایین* محاسبه کنیم. به بیان دیگر می‌خواهیم **ارزش** Q در حرکت عامل به سمت *پایین* در حالت **A** را بدست آوریم. لذا **ارزش** Q، **بازگشت** یا **بازده** مسیر ما خواهد بود که از حالت **A** شروع کرده و اقدام حرکت به سمت *پایین* را انجام دهیم:

$$Q^{\pi}(A, \text{down}) = [R(\tau)|s_o = A,\ \ a_o = \text{down}]$$

---

[۱] State-Action Value Function

$$Q(A, \text{down}) = ۱ + ۱ + ۱ + ۱ = ۴$$

فرض کنید، ما باید **ارزش** Q را برای جفت حالت–اقدامِ **D** یعنی حرکت به سمت *راست* محاسبه کنیم. یعنی بدنبال **ارزش** Q در حرکت عامل به سمت *راست* در حالت **D** هستیم. **ارزش** Q، **بازگشت** مسیر ما خواهد بود که از حالت **D** شروع شده و اقدام حرکت به طرف *راست* را انجام می‌دهد:

$$Q^{\pi}(A, \text{right}) = [R(\tau)|s_o = D,\ \ a_o = \text{right}]$$

به طور مشابه، می‌توانیم ارزش Q را برای همه جفت‌های حالت–اقدام محاسبه کنیم. مشابه آنچه ما در مورد تابع ارزش یاد گرفتیم، به جای اینکه بازده را مستقیماً به عنوان ارزش Q یک جفت حالت–اقدام در نظر بگیریم، بعضا از بازده مورد انتظار استفاده می‌کنیم زیرا بازده، یک متغیر تصادفی است و ارزشهای متفاوتی را با مقادیر احتمالی مختلفی می‌گیرد. بنابراین، می‌توانیم تابع Q خود را به صورت زیر تعریف کنیم:

$$Q^{\pi}(s, a) = \mathop{\mathbb{E}}_{\tau \sim \pi} [R(\tau)|s_o = s,\ \ a_o = a\ ]$$

این بدان معنی است که ارزش Q، آن مقدار بازده مورد انتظاری است که عامل با شروع از حالت s و انجام اقدام a تحت خطمشی π، بدست می‌آورد.

مشابه تابع ارزش، تابع Q به خطمشی بستگی دارد، یعنی **ارزش** Q بر اساس **سیاست** انتخابی ما متفاوت است. با توجه به خطمشی‌های مختلف، توابع Q بسیار متفاوتی می‌تواند وجود داشته باشد. تابع Q بهینه، تابعی است که حداکثر مقدار Q را نسبت به سایر توابع Q برگردانده و می‌توان آن را به صورت زیر بیان کرد:

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$$

خطمشی بهینه π^*، آن **سیاستی** است که حداکثر ارزش Q را می‌دهد.

مانند **تابع ارزش**، می‌توان **تابع** Q را در یک جدول مشاهده کرد، که به آن جدول Q می‌گویند. فرض کنید دو حالت s_o و $s_۱$ و دو اقدام ۰ و ۱ داریم؛ در نتیجه آن تابع Q را می توان به صورت زیر نشان داد:

حالت	اقدام	ارزش
S_0	۰	۹
S_0	۱	۱۱
S_1	۰	۱۷
S_1	۱	۱۳

جدول ۱.۵: **جدول** Q

همانطور که مشاهده می‌کنیم، جدول Q، ارزش‌های Q تمام جفت‌های ممکن حالت–اقدام را نشان می‌دهد. ما آموختیم که خط‌مشی بهینه **سیاستی** است که حداکثر **بازده** (مجموع **پاداش**) را به عامل ما می‌دهد. ما می‌توانیم، فقط با انتخاب اقدامی که حداکثر مقدار Q را در هر حالت دارد، سیاست بهینه را از جدول Q استخراج کنیم. بنابراین، سیاست بهینه ما، اقدام ۱ را در حالت S_0 و اقدام ۰ را در حالت S_1 انتخاب می‌کند زیرا همانطور که در جدول ۱.۶ نشان می‌دهد، آن‌ها ارزش Q بالایی دارند.

جدول Q

حالت	اقدام	ارزش
S_0	۰	۹
S_0	۱	۱۱
S_1	۰	۱۷
S_1	۱	۱۳

سیاست بهینه

حالت	اقدام
S_0	۱
S_1	۰

جدول ۱.۶: سیاست بهینه استخراج شده از جدول Q

بنابراین، می‌توانیم **سیاست** بهینه را با محاسبه **تابع** Q استخراج کنیم.

یادگیری مبتنی بر مدل و بدون مدل

حال، بیایید به دو نوع مختلف یادگیری به نام‌های یادگیری با مدل و بی مدل نگاه کنیم.

یادگیری مبتنی بر مدل[۱]: در یادگیری با مدل، عامل ما، توصیف کاملی از محیط خواهد داشت. می‌دانیم که احتمال انتقال[۲]، بیانگر احتمال حرکت از حالت s به حالت بعدی s'، با انجام اقدام a است. **تابع پاداش** بیانگر آن است که، در حین حرکت از حالت s به حالت بعدی s'، با انجام اقدام a، به چه پاداشی می‌رسیم. زمانی‌که عامل، پویایی‌های محیط مدل را می‌داند، یعنی عامل، از **احتمالات انتقال** در محیط خود آگاه است، آنگاه این نوع یادگیری را، **یادگیری مبتنی بر مدل** نامند. بنابراین، در یادگیری با مدل، عامل از پویایی‌های مدل برای یافتن **خط‌مشی** یا **سیاست** بهینه استفاده می‌کند.

یادگیری بدون مدل[۳]: یادگیری بی مدل برای زمانی است که عامل از پویایی‌های محیط مدل خود اطلاعی نداشته باشد. یعنی در یادگیری بدون مدل، عامل ما، سعی می‌کند **سیاست** یا **خط‌مشی** بهینه را بدون پویایی مدل بیابد.

در گام بعد، انواع مختلف محیطی که یک عامل در آن کار می‌کند را بحث خواهیم کرد.

انواع مختلف محیط‌ها

در ابتدای فصل، آموختیم که محیط، همانا جهان عامل ماست و عامل در محیط، قرار دارد یا زندگی می‌کند. ما می‌توانیم محیط را به انواع مختلف دسته‌بندی کنیم.

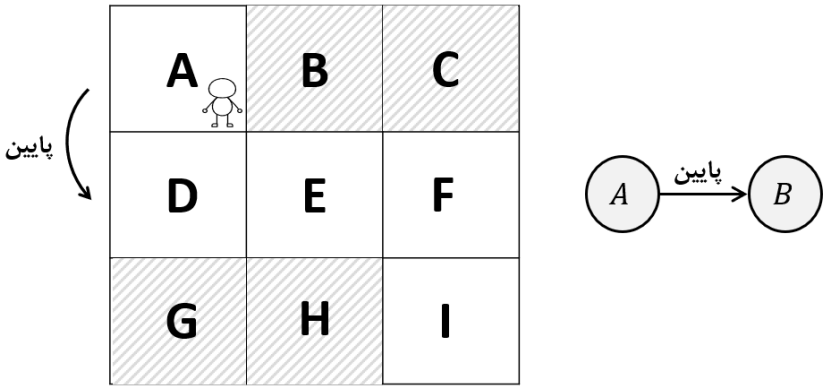
محیطهای قطعی و اتفاقی

- **محیطهای قطعی (معین)**
- **محیطهای اتفاقی (پیشامدی)**

---

[۱] Model-Based Learning
[۲] Transition Probability
[۳] Model-Free Learning

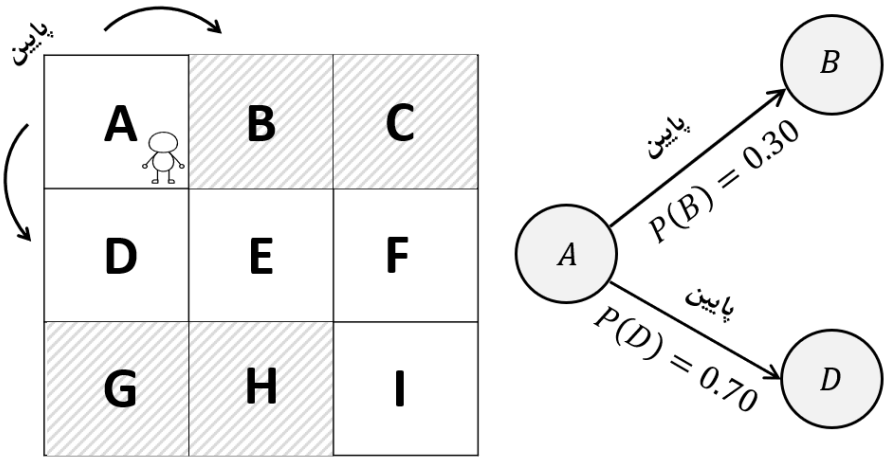
محیط قطعی[۱]: در یک **محیط قطعی**، مطمئن هستیم که وقتی یک عامل، اقدام a را در حالت s انجام می‌دهد، آنگاه، او همیشه به حالت s' می‌رسد. به عنوان مثال، اجازه دهید محیط جهان مشبک خود را در نظر بگیریم. فرض کنید عامل در حالت **A** است و هنگامی که از حالت **A** به پایین حرکت می‌کند، همیشه به حالت **D**، می‌رسد. بنابراین، محیط را یک **محیط قطعی** می‌نامند.



شکل ۱.۳۰: محیط قطعی

محیط اتفاقی[۲]: در یک **محیط اتفاقی**، نمی‌توان گفت که عامل با انجام اقدام a، در حالت s، همیشه به حالت s'، می‌رسد. زیرا نوعی تصادفی بودن[۳]، همواره مرتبط با محیط **اتفاقی** خواهد بود. حال بیایید فرض کنیم، محیط جهان شبکه ما یک **محیط اتفاقی** است. فرض کنیم، عامل ما در حالت **A** است؛ حالا اگر او از حالت **A** به *پایین* حرکت کند، آنگاه عامل همیشه به حالت **D** نمی‌رسد. به این معنی که او در ۷۰٪ مواقع، به حالت **D** و ۳۰٪ اوقات به حالت **B** می‌رسد. یعنی اگر عامل در حالت **A**، به سمت *پایین* حرکت کند، آنگاه، عامل با **احتمال** ۷۰٪ به حالت **D** و با **احتمال** ۳۰٪ به حالت **B** می‌رسد، همانطور که در شکل ۱.۳۱ نشان داده شده است:

[۱] Deterministic Environment
[۲] Stochastic Environment
[۳] Randomness



شکل ۱.۳۱: محیط اتفاقی

محیط‌های گسسته و پیوسته

محیط گسسته[۱]: محیط گسسته، محیطی است که فضای اقدام در آن محیط، گسسته است. به عنوان مثال، در محیط جهان شبکه، ما یک فضای کنش یا اقدام گسسته داریم که از اقدامات [بالا، پایین، چپ، راست] تشکیل شده است و بنابراین محیط جهان شبکه ما گسسته است.

محیط پیوسته[۲]: یک محیط پیوسته، محیطی است که فضای کنش یا اقدام در آن محیط، پیوسته باشد. به عنوان مثال، فرض کنید ما در حال آموزش یک عامل، برای رانندگی با یک خودرو هستیم، در اینصورت فضای اقدام ما پیوسته خواهد بود، با چندین اقدام پیوسته مانند تغییر سرعت خودرو، تعداد درجاتی که عامل برای چرخاندن چرخ نیاز دارد و غیره. در چنین حالتی، فضای اقدام محیط ما عملا پیوسته است.

---

۱ Discrete Environment
۲ Continuous Environment

محیط‌های پردینه‌ای و غیرپردینه‌ای

محیط پرده‌وار (اپیزودیک): در یک محیط پردینه‌ای، اقدام فعلی یک عامل، بر اقدامات آتی او، تأثیری نخواهد داشت و بنابراین به محیط پرده‌وار (اپیزودیک)، محیط غیر متوالی[۱] نیز می‌گویند.

محیط غیر-پرده‌وار (غیر اپیزودیک)[۲]: در یک محیط غیرپردینه‌ای، اقدام فعلی یک عامل، بر اقدامات آینده تأثیر می‌گذارد و بنابراین به محیط غیر پرده‌وار (غیر اپیزودیک)، محیط متوالی[۳] نیز می‌گویند. به عنوان مثال، یک صفحه شطرنج یک محیط متوالی است زیرا اقدام فعلی عامل، بر اقدامات آینده در یک مسابقه شطرنج تأثیر می‌گذارد.

محیط‌های تک و چند عاملی

- **محیط تک‌عاملی**[۴]: وقتی محیط ما فقط از یک عامل تشکیل شده باشد، آن را محیط تک عاملی می‌نامند.
- **محیط چندعاملی**[۵]: زمانی که محیط ما از چندین عامل تشکیل شده باشد به آن محیط چندعاملی می‌گویند.

ما بسیاری از مفاهیم یادگیری تقویتی را پوشش داده‌ایم. اکنون، با بررسی چند کاربرد هیجان‌انگیز یادگیری تقویتی، این فصل را به پایان می‌رسانیم.

کاربردهای یادگیری تقویتی

یادگیری تقویتی، در چند سال گذشته با طیف گسترده‌ای از کاربردها، از انجام بازی‌ها تا راندن ماشین‌های خودران[۶] به سرعت تکامل یافته است. یکی از دلایل اصلی این تکامل، یادگیری تقویتی عمیق[۷] (**DRL**) است که ترکیبی از

---

۱ Non-Sequential Environment
۲ Non-Episodic Environment
۳ Sequential Environment
۴ Single-Agent Environment
۵ Multi-Agent Environment
۶ Self-Driving Cars
۷ Deep Reinforcement Learning (DRL)

یادگیری تقویتی و یادگیری عمیق است. ما در مورد الگوریتم‌های پیشرفته یادگیری عمیق در فصل‌های آینده خواهیم آموخت، پس همچنان منتظر باشید! در این بخش، به برخی از کاربردهای واقعی یادگیری تقویتی، نگاهی خواهیم داشت:

• **ساخت و تولید**[۱]: در فضاهای تولیدی، این ربات‌های هوشمند هستند که با استفاده از یادگیری تقویتی آموزش می‌بینند تا اشیا را در موقعیت مناسب قرار دهند. استفاده از ربات‌های هوشمند، باعث کاهش هزینه‌های نیروی کار و افزایش بهره‌وری می‌شود.

• **قیمت‌گذاری پویا**[۲]: یکی از کاربردهای محبوب یادگیری تقویتی، قیمت‌گذاری پویا است. قیمت‌گذاری پویا به این معناست که قیمت محصولات را بر اساس تقاضا و عرضه تغییر می‌دهیم. ما می‌توانیم عامل یادگیری تقویتی را برای قیمت‌گذاری پویای محصولات با هدف به حداکثر رساندن درآمد، آموزش دهیم.

• **مدیریت موجودی**[۳]: یادگیری تقویتی، به طور گسترده در مدیریت موجودی (که یک فعالیت تجاری مهم است)، استفاده می‌شود. برخی از این فعالیت‌ها، شامل مدیریت زنجیره تامین، پیش‌بینی تقاضا و مدیریت چندین عملیات انبار (مانند قراردادن محصولات در انبارها برای مدیریت کارآمد فضا) است.

• **سامانه توصیه‌گر**[۴]: از یادگیری تقویتی، به طور گسترده در ساخت یک سامانه توصیه‌گر (بویژه در جایی که رفتار کاربر دائماً تغییر می‌کند) استفاده می‌شود. به عنوان مثال، در سامانه‌های توصیه‌گر موسیقی، رفتار یا ترجیحات موسیقی کاربر هر از گاهی تغییر می‌کند. بنابراین، در این موارد، استفاده از یک عامل یادگیری تقویتی، می‌تواند خیلی مفید باشد چر که عامل دائماً از طریق تعامل با محیط یاد می‌گیرد.

• **جستجوی معماری عصبی**[۵]: برای اینکه یک شبکه عصبی بتواند یک وظیفه معین را با دقت خوبی انجام دهد، لازم است معماری شبکه به درستی طراحی شود. با یادگیری تقویتی، می‌توانیم فرآیند جستجوی پیچیده معماری عصبی را

---

۱ Manufacturing
۲ Dynamic Pricing
۳ Inventory Management
۴ Recommendation System
۵ Neural Architecture Search

با آموزش عامل برای یافتن بهترین معماری عصبی برای یک کار معین و با هدف به حداکثر رساندن دقت آن، بطور خودکار انجام دهیم.

• **پردازش زبان طبیعی**[۱]: با افزایش محبوبیت الگوریتم‌های تقویتی عمیق[۲]، یادگیری تقویتی، به طور گسترده‌ای در چندین فعالیت پردازش زبان طبیعی، مانند خلاصه‌سازی متن، ربات‌های چت و غیره، استفاده شده است.

• **امور مالی**[۳]: مدیریت سبد مالی نیازمند بازتوزیع منابع مالی[۴] در میان تعداد خاصی از محصولات و یا پروژه‌های مالی است. هم اینک یادگیری تقویتی، به طور گسترده در مدیریت پرتفوی مالی استفاده می‌شود. یادگیری تقویتی، همچنین در پیش‌بینی و تجارت در بازارهای معاملات تجاری، استفاده می‌شود. شرکت جی پی مورگان از یادگیری تقویتی، برای ارائه نتایج بهتر در اجرای معاملات برای سفارشات بزرگ، بطور موفقیت‌آمیزی استفاده کرده است.

واژه‌نامه یادگیری تقویتی

ما چندین مفهوم مهم و اساسی یادگیری تقویتی را یاد گرفتیم. در این بخش، ما مجددا چندین اصطلاح مهم را که برای درک فصلهای آینده بسیار مفید هستند، مرور می‌کنیم.

عامل: عامل، یک برنامه نرم‌افزاری است که تصمیم‌گیری هوشمندانه را یاد می‌گیرد (مانند برنامه نرم‌افزاری که شطرنج را هوشمندانه بازی می‌کند.)

محیط : محیط، همان جهان یا دنیای عامل است. اگر مثال شطرنج را در نظر بگیریم، صفحه شطرنج محیطی است که عامل، در آن شطرنج بازی می‌کند.

---

۱ Natural Language Processing (NLP)
۲ Deep Reinforcement Algorithms
۳ Finance
۴ Fund

حالت: حالت، موقعیت یا لحظه‌ای در محیطی است که عامل می‌تواند در آن قرار گیرد. مثلاً، به تمام موقعیت‌های صفحه شطرنج، حالت می‌گویند.

اقدام: عامل با انجام یک **اقدام** یا **عمل**، با محیط تعامل می‌کند و از حالتی به حالت دیگر می‌رود، مثلاً حرکاتی که شطرنج بازان انجام می‌دهند، اقدام هستند.

پاداش: پاداش، یک مقدار عددی است که، عامل بر اساس اقدام خود دریافت می‌کند. یک **پاداش** را به عنوان یک امتیاز در نظر بگیرید. به عنوان مثال، یک نماینده ۱+ امتیاز (پاداش) برای یک اقدام خوب و ۱- امتیاز (پاداش) برای یک اقدام بد، دریافت می‌کند.

فضای کنش: به مجموعه تمامی **اقدامات** ممکن در یک محیط، **فضای اقدام** (کنش) می‌گویند. فضای عمل زمانی که فضای کنش یا اقدام ما از کنشهای گسسته تشکیل شده باشد، فضای کنش یا اقدام گسسته نامیده می‌شود و زمانی که فضای کنش ما از اقدامات پیوسته تشکیل شده باشد، به آن فضای اقدامات، فضای کنش پیوسته گویند.

سیاست: عامل، بر اساس **سیاست** (یا **خطمشی**)، تصمیم می‌گیرد. یک **سیاست** به عامل می‌گوید که در هر حالت چه اقدامی را انجام دهد. **سیاست را می‌توان مغز یک عامل دانست.** اگر **سیاستی**، دقیقاً یک حالت را به یک عمل خاص منتسب کند (تناظر یک به یک)، **سیاست قطعی** نامیده می‌شود. برخلاف یک **سیاست قطعی**، یک **خطمشی اتفاقی**، حالت را به یک **توزیع احتمال** در فضای عمل نگاشت می‌کند. **سیاست** بهینه، مجموعهٔ کنشهایی است که حداکثر پاداش را بدهد.

پردینه (اپیزود): تعامل عامل- محیط از حالت اولیه تا حالت پایانی، یک پردینه نامیده می‌شود. هر پرده در انجام وظایف یادگیری تقویتی نشاندهنده یک خط سیر (سوگاه یا رهسو) است و آنرا را اغلب یک مسیر می‌نامند.

وظایف منقطع و پیوسته: یک وظیفه یادگیری تقویتی را اگر حالت پایانی داشته باشد، وظیفه پرده‌وار (پردینه‌ای) و اگر حالت پایانی نداشته باشد، وظیفه پیوسته یا متوالی (غیر پردینه‌ای) می‌گویند.

افق: افق را می‌توان طول عمر عامل در نظر گرفت، یعنی گام زمانی که تا آن زمان، عامل با محیط تعامل دارد. اگر تعامل عامل و محیط در یک مرحله زمانی خاص متوقف شود، افق محدود[۱] نامیده می‌شود و زمانی که تعامل محیط–عامل برای همیشه ادامه یابد، افق نامحدود[۲] است.

بازده: بازگشت یا **بازده**، مجموع **پاداش‌های** دریافت شده توسط عامل در یک اپیزود است.

نرخ تنزیل (ضریب تخفیف): فاکتور تخفیف، کمک می‌کند تا کنترل کنیم که آیا می‌خواهیم به پاداش فوری[۳]، یا پاداش‌های آینده، اهمیت دهیم. ضریب تخفیف، از مقدار ۰ تا ۱ متغیر است. ضریب تخفیف نزدیک به ۰ نشان می‌دهد که ما به پاداش‌های فوری اهمیت بیشتری می‌دهیم، در حالی که ضریب تخفیف نزدیک به ۱، بیانگر آن است که ما به پاداش‌های آینده نسبت به پاداش‌های فوری اهمیت بیشتری می‌دهیم.

تابع ارزش: تابع **ارزش** یا مقدار حالت، مقدار **بازگشت** مورد انتظاری است که، یک عامل، با شروع از حالت S و پیگیری **سیاست** π، دریافت می‌کند.

تابع Q: تابع Q یا «ارزش یک زوج حالت–عمل»، دلالت بر **بازگشت** مورد انتظاری دارد که یک عامل با شروع از حالت S و انجام **عمل** a، با پیگیری **خط‌مشی** π، به دست می‌آورد.

یادگیری مبتنی بر مدل و بدون مدل: زمانی که عامل تلاش می‌کند، **خط‌مشی** بهینه را با پویایی‌های مدل یاد بگیرد، آن را یادگیری با مدل می‌نامند. و هنگامی که عامل تلاش می‌کند تا **سیاست** بهینه را بدون پویایی‌های مدل یاد بگیرد، آن را یادگیری بی مدل می‌نامند.

---

[۱] Finite Horizon
[۲] Infinite Horizon
[۳] Immediate Reward

محیط قطعی و اتفاقی: وقتی یک عامل، عمل a را در حالت s انجام می‌دهد و هر بار به حالت s' می‌رسد، آن محیط را، **محیط قطعی** می‌نامند. هنگامی که یک عامل، عمل a را در حالت s انجام می‌دهد و هر بار بر اساس **توزیع احتمال** به حالتهای مختلفی می‌رسد، آنگاه محیط را یک **محیط اتفاقی** می‌نامند.

خلاصه

ما این فصل را با درک ایده اصلی یادگیری تقویتی آغاز کردیم. ما آموختیم که یادگیری تقویتی، یک فرآیند یادگیری آزمون و خطا است و یادگیری در یادگیری تقویتی، بر اساس **پاداشی** است که به ازای یک **اقدام** بدست می‌آید. سپس تفاوت بین یادگیری تقویتی و سایر پارادایمهای یادگیری ماشین، مانند یادگیری تحت نظارت و بدون نظارت را بررسی کردیم. در ادامه، ما در مورد فرایند تصمیم‌گیری مارکوف و نحوه مدل‌سازی محیط یادگیری تقویتی، به عنوان یک فرایند یادگیری مارکوف، یاد گرفتیم. در مرحله بعد، ما چندین مفهوم اساسی مهم در یادگیری تقویتی را درک کردیم و در پایان فصل به برخی از کاربردهای واقعی یادگیری تقویتی، پرداختیم.

بنابراین، در این فصل، چندین مفهوم اساسی یادگیری تقویتی را آموختیم. در فصل بعد، ما سفر یادگیری تقویتی خود را با اجرای تمام مفاهیم اساسی که در این فصل آموخته‌ایم و با استفاده از ابزار محبوبی به نام Gym آغاز خواهیم کرد.

سوالات

بیایید دانش جدید خود را با پاسخ به این سؤالات ارزیابی کنیم:

۱. یادگیری تقویتی، چه تفاوتی با سایر پارادایمهای یادگیری ماشین دارد؟
۲. در چارچوب یادگیری تقویتی، به چه چیزی محیط گفته می‌شود؟
۳. تفاوت بین **سیاست قطعی** و **سیاست اتفاقی** چیست؟
۴. پردینه (اپیزود) چیست؟

۵. چرا به فاکتور تخفیف (تنزیل) نیاز داریم؟

۶. **تابع ارزش** چه تفاوتی با **تابع** Q دارد؟

۷. تفاوت بین **محیط قطعی** و **محیط اتفاقی** چیست؟

برای مطالعه بیشتر

Reinforcement Learning: A Survey by *L. P. Kaelbling, M. L. Littman, A. W. Moore,* available at https://arxiv.org/abs/cs/9605103

فصل دوم

راهنمای جعبه ابزار Gym

OpenAI، یک **سازمان تحقیقاتی هوش مصنوعی (AI)** است که هدف آن ایجاد **هوش مصنوعی عمومی**[۱]**(AGI)** است. OpenAI، یک جعبه ابزار[۲] معروف به نام Gym را برای آموزش یک عامل یادگیری تقویتی ارائه می‌دهد.

بیایید فرض کنیم که باید به عامل خود، رانندگی ماشین را آموزش بدهیم. ما به محیطی برای آموزش عامل نیاز داریم. آیا می‌توانیم عامل خود را در محیط واقعی برای رانندگی ماشین آموزش دهیم؟ نه! زیرا ما آموخته‌ایم که **یادگیری تقویتی** (**RL**) یک فرآیند یادگیری آزمون و خطاست. بنابراین در حالی که ما عامل خود را آموزش می‌دهیم، اشتباهات زیادی در طول یادگیری مرتکب می‌شود. به عنوان مثال، فرض کنید عامل به وسیله نقلیه دیگری برخورد کرده و یک پاداش منفی دریافت می‌کند. سپس یاد می‌گیرد که ضربه زدن به وسایل نقلیه دیگر عمل خوبی نیست و سعی می‌کند دیگر این عمل را انجام ندهد. اما ما نمی‌توانیم عامل یادگیری تقویتی را در محیط واقعی با ضربه زدن به وسایل نقلیه دیگر آموزش دهیم. به همین دلیل است که ما از شبیه‌سازها استفاده می‌کنیم و عامل یادگیری تقویتی را در محیط‌های شبیه‌سازی شده آموزش می‌دهیم.

ابزارهای زیادی وجود دارند که یک محیط شبیه‌سازی شده برای آموزش یک عامل یادگیری تقویتی را فراهم می‌کنند. یکی از این ابزارهای محبوب Gym است. جعبه ابزار Gym، محیط‌های مختلفی را برای آموزش یک عامل یادگیری تقویتی، از وظایف کنترل کلاسیک گرفته تا محیط‌های بازی Atari فراهم می‌کند. ما می‌توانیم عامل یادگیری تقویتی خود را برای یادگیری در این محیط‌های شبیه‌سازی شده با استفاده از الگوریتم‌های مختلف یادگیری تقویتی، آموزش دهیم. در این فصل ابتدا Gym را نصب می‌کنیم و سپس به بررسی محیط‌های مختلف Gym می‌پردازیم. همچنین مفاهیمی را که در فصل قبل آموختیم را با آزمایش در محیط Gym بیشتر تجربه می‌کنیم.

در سراسر کتاب از جعبه ابزار Gym برای ساخت و ارزیابی الگوریتم‌های یادگیری تقویتی استفاده خواهیم کرد، بنابراین در این فصل، با جعبه ابزار Gym آشنا خواهیم شد.

در این فصل با موضوعات زیر آشنا می‌شویم:

---

[۱] Artificial General Intelligence
[۲] Toolkit

- **راه اندازی سامانه**
- **نصب آناکوندا[۱] و جیم[۲]**
- **درک محیط جیم**
- **تولید اپیزود در محیط جیم**
- **کاوش بیشتر در محیط‌های جیم**
- **تعادل آونگ وارونه[۳] با سیاست تصادفی (بختکی)**
- **معرفی یک برنامه برای بازی تنیس**

راه اندازی سامانه

در این بخش، نحوه نصب چندین پیشنیاز (یا وابستگان نرم‌افزاری[۴]) که برای اجرای کدهای مورد استفاده در سراسر کتاب مورد نیاز است را یاد خواهیم گرفت. ابتدا نحوه نصب **آناکوندا** را یاد گرفته و سپس نحوه نصب جیم را بررسی می‌کنیم.

نصب آناکوندا

آناکوندا یک سامانه منبع باز پایتون[۵] است که به طور گسترده‌ای برای محاسبات علمی و پردازش حجم زیادی از داده‌ها استفاده می‌شود. این سامانه یک محیط مدیریت پکیج[۶] عالی را فراهم کرده و از سیستم عامل‌های ویندوز، مک و لینوکس پشتیبانی می‌کند. آناکوندا با نصب پایتون به همراه سایر پکیج‌های محبوب مورد استفاده برای محاسبات علمی مانند NumPy، SciPy و غیره ارائه می‌شود.

---

۱ Anaconda
۲ Gym
۳ Cart-Pole
۴ Dependencies
۵ Open-Source Distribution of Python
۶ Package Management Environment

برای دانلود آناکوندا به سایت https://www.anaconda.com/download مراجعه کنید، در آنجا گزینه‌ای برای دانلود آناکوندا و برای پلتفرم‌های مختلف مشاهده خواهید کرد. اگر از ویندوز یا مک استفاده می‌کنید، می‌توانید مستقیماً نصب‌کننده گرافیکی[۱] را مطابق با معماری دستگاه خود دانلود کرده و آناکوندا را با استفاده از نصب‌کننده گرافیکی نصب کنید.

اگر از لینوکس استفاده می‌کنید، مراحل زیر را دنبال کنید:

۱.ترمینال را باز کرده و دستور زیر را برای دانلود آناکوندا تایپ کنید:

```
wget https://repo.continuum.io/archive/Anaconda3-5.0.1-
Linux-x86_64.sh
```

۲. پس از دانلود آن، می‌توان آناکوندا را با دستور زیر نصب کرد:

```
bash Anaconda3-5.0.1-Linux-x86_64.sh
```

پس از نصب موفقیت آمیز آناکوندا، باید یک محیط مجازی ایجاد کنیم. چرا نیاز به محیط مجازی داریم؟ فرض کنید ما روی پروژه **A** کار می‌کنیم که از NumPy نسخه ۱.۱۴ استفاده می‌کند و پروژه **B** که از نسخه NumPy نسخه ۱.۱۳ استفاده می‌کند. بنابراین، برای کار روی پروژه **B**، یا باید NumPy را تنزل درجه[۲] دهیم و یا NumPy را دوباره نصب کنیم. در هر پروژه، از کتابخانه‌های مختلف با نسخه‌های مختلف استفاده می‌کنیم که برای پروژه‌های دیگر قابل استفاده نیستند. لذا به جای تنزل یا ارتقاء نسخه‌ها یا نصب مجدد کتابخانه‌ها برای هر پروژه جدید، از یک محیط مجازی استفاده می‌کنیم.

محیط مجازی فقط یک محیط ایزوله برای یک پروژه خاص است، به طوری که هر پروژه می‌تواند وابستگان نرم‌افزاری خاص خود را داشته و بر روی پروژه‌های دیگر تاثیری نداشته باشد. ما با استفاده از دستور زیر یک محیط مجازی ایجاد می‌کنیم و محیط خود را به نوعی یک دنیا یا جهان[۳] (`universe`) می‌نامیم:

---

[۱] Graphical Installer
[۲] Downgrade
[۳] Universe

```
conda create --name universe python=3.6 anaconda
```

توجه داشته باشید که ما از پایتون نسخه ۳.۶ استفاده می‌کنیم. پس از ایجاد محیط مجازی، می توانیم با استفاده از دستور زیر آن را فعال کنیم:

```
source activate universe
```

انجام شد! اکنون، نحوه نصب آناکوندا و ایجاد یک محیط مجازی را یاد گرفتیم، در قسمت بعدی نحوه نصب جیم را یاد می‌گیریم.

نصب جعبه ابزار جیم

در این قسمت با نحوه نصب جعبه ابزار جیم آشنا می‌شویم. قبل از شروع، ابتدا بیایید محیط مجازی خود، جهان (یا `universe`) خودمان را فعال کنیم:

```
source activate universe
```

اکنون وابستگان زیر را نصب کنید:

```
sudo apt-get update
sudo apt-get install golang libcupti-dev libjpeg-turbo8-dev make tmux
htop chromium-browser git cmake zlib1g-dev libjpeg-dev xvfb libav-tools
xorg-dev python-opengl libboost-all-dev libsdl2-dev swig
conda install pip six libgcc swig
conda install opencv
```

ما می‌توانیم جیم را مستقیماً با استفاده از `pip` نصب کنیم. توجه داشته باشید که در سراسر کتاب از جیم نسخه ۰.۱۵.۴ استفاده خواهیم کرد. با استفاده از دستور زیر می‌توانیم جیم را نصب کنیم:

```
pip install gym==0.15.4
```

همچنین می‌توانیم جیم را با شبیه‌سازی مخزن جیم[۱] به صورت زیر نصب کنیم:

```
cd ~
git clone https://github.com/openai/gym.git
cd gym
pip install -e '.[all]'
```

رفع خطاهای رایج

در صورتی که هنگام نصب جیم، با هر یک از خطاهای زیر مواجه شدید، دستورات زیر به شما کمک خواهد کرد:

- **عدم موفقیت در ساخت چرخ برای pachi-py یا شکست در ساخت چرخ برای pachi-py atari-py:**

```
sudo apt-get update
sudo apt-get install xvfb libav-tools xorg-dev libsdl2-dev swig
cmake
```

- **عدم موفقیت در ساخت چرخ برای mujoco-py:**

```
git clone https://github.com/openai/mujoco-py.git
cd mujoco-py

sudo apt-get update
sudo apt-get install libgl1-mesa-dev libgl1-mesa-glx libosmesa6-
dev python3-pip python3-numpy python3-scipy

pip3 install -r requirements.txt
sudo python3 setup.py install
```

- **خطا: دستور 'gcc' دچار شکست و خروج از وضعیت ۱ شد:**

[۱] Cloning the Gym Repository

```
sudo apt-get update

sudo apt-get install python-dev
sudo apt-get install libevent-dev
```

اکنون که جیم را با موفقیت نصب کردیم، در بخش بعدی، بیایید سفر یادگیری تقویتی خود را آغاز کنیم.

ایجاد اولین محیط جیم برای کار

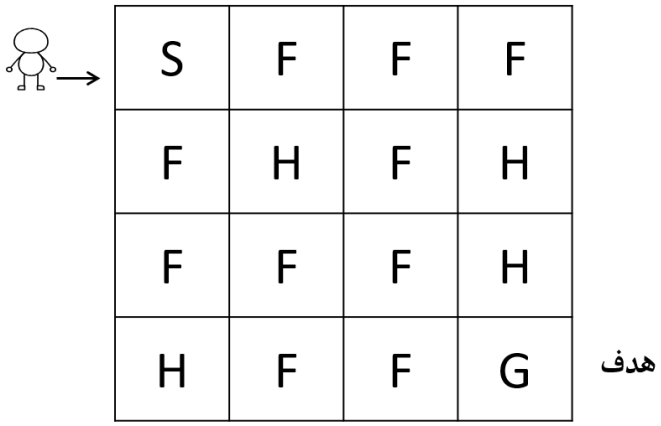
ما آموخته‌ایم که جیم، محیط‌های مختلفی را برای آموزش یک عامل یادگیری تقویتی فراهم می‌کند. برای درک واضح نحوه طراحی محیط جیم، با محیط اصلی جیم شروع می‌کنیم. پس از آن، دیگر محیط‌های پیچیده جیم را خواهیم فهمید.

یکی از ساده‌ترین محیط‌ها به نام محیط دریاچه یخ زده (یا منجمد)[۱] را معرفی می‌کنیم. شکل ۲.۱ محیط دریاچه یخ زده را نشان می‌دهد. همانطور که مشاهده می‌کنیم، در محیط دریاچه یخ زده، هدف عامل شروع از حالت اولیه **S** و رسیدن به حالت هدف **G** است.

در محیط قبلی موارد زیر اعمال می‌شود:

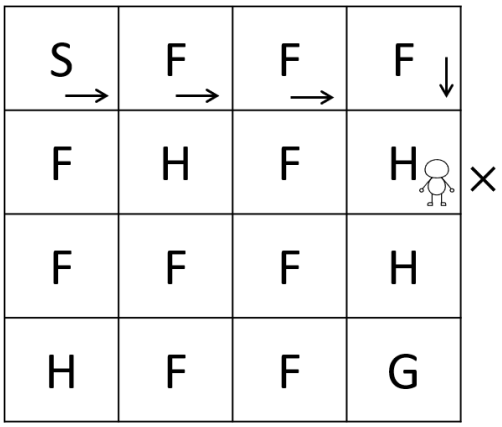
- حرف **S** حالت شروع را نشان می‌دهد. (Starting)
- حرف **F** حالت یخ‌زده را نشان می‌دهد. (Frozen)
- حرف **H** حالت حفره را نشان می‌دهد. (Hole)
- حرف **G** حالت هدف را نشان می‌دهد. (Goal)

[۱] Frozen Lake Environment



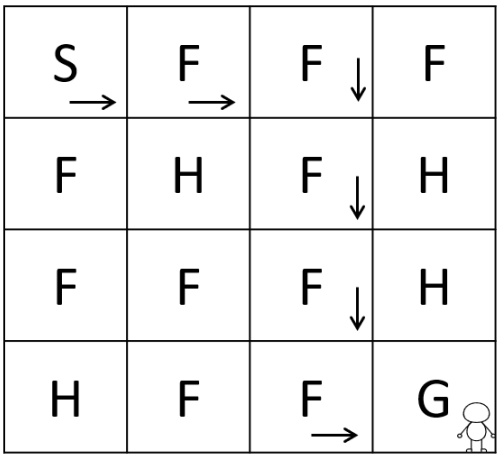
شکل ۲.۱ : محیط دریاچه یخ‌زده

بنابراین، عامل باید از حالت **S** شروع کند و به حالت هدف **G** برسد. اما مشکل آنست که اگر عامل از حالت **H** که حالت حفره است بازدید کند، همانطور که شکل ۲.۲ نشان می‌دهد، آنگاه عامل در حفره می‌افتد و می‌میرد.



شکل ۲.۲: عامل در یک حفره سقوط می‌کند.

بنابراین، باید مطمئن شویم که عامل از **S** شروع کرده و بدون قرار گرفتن در حالت حفره **H** همانطور که در شکل ۲.۳ نشان داده شده است، به **G** می‌رسد:



شکل ۲.۳: عامل به حالت هدف می‌رسد.

هر خانه جدول مشبک[۱] در محیط قبلی، نشانگر یک حالت است، بنابراین ما ۱۶ حالت (**S** تا **G**) داریم و ۴ اقدام ممکن داریم که عبارتند از *بالا*، *پایین*، *چپ* و *راست*. ما یاد گرفتیم که هدف ما رسیدن به حالت **G** از **S** بدون بازدید از **H** است. بنابراین، پاداش ۱+ را برای حالت هدف **G** و ۰ را برای همه حالت‌های دیگر تعیین می‌کنیم.

بنابراین، ما یاد گرفتیم که محیط دریاچه یخ زده چگونه کار می‌کند. حال برای آموزش عامل خود در محیط دریاچه یخ‌زده، ابتدا باید محیط را با کدگذاری از ابتدا در پایتون ایجاد کنیم. اما خوشبختانه ما مجبور نیستیم این کار را انجام دهیم! از آنجایی که جیم، محیط‌های مختلفی را فراهم می‌کند، می‌توانیم به طور مستقیم جعبه ابزار جیم را وارد کرده و یک محیط دریاچه یخ‌زده را ایجاد کنیم.

اکنون، ما یاد خواهیم گرفت که چگونه با استفاده از جیم، محیط دریاچه یخ زده خود را ایجاد کنیم. قبل از اجرای هر کدی، مطمئن شوید که جهان محیط مجازی ما را فعال کرده‌اید. ابتدا، بیایید کتابخانه جیم را وارد کنیم:

```
import gym
```

در مرحله بعد، می‌توانیم با استفاده از تابع `make` یک محیط جیم ایجاد کنیم. تابع `make` به شناسه محیط[۲] به عنوان

[۱] Grid Box
[۲] Environment ID

یک پارامتر نیاز دارد. در جیم، شناسه محیط FrozenLake-v0 است. بنابراین، ما می‌توانیم محیط دریاچه یخ‌زده خود را به صورت زیر ایجاد کنیم:

```
env = gym.make("FrozenLake-v0")
```

پس از ایجاد محیط، می‌توانیم با استفاده از تابع render، ببینیم که محیط ما چگونه به نظر می‌رسد:

```
env.render()
```

کد قبلی، محیط زیر را ارائه می‌دهد:

شکل ۲.۴: محیط دریاچه یخ زده جیم

همانطور که مشاهده کردیم، محیط دریاچه یخ زده از شانزده حالت (**S** تا **G**) تشکیل شده است. حالت **S** با رنگ قرمز نمایان شده تا نشان دهد این حالت فعلی ما است، یعنی عامل، در حالت **S** است. بنابراین هر زمان که یک محیط ایجاد می‌کنیم، عامل ما، همیشه از یک حالت اولیه شروع می‌کند که در این مورد، حالت **S** است.

همین بود! ایجاد محیط با استفاده از جیم، به همین سادگی است. در بخش بعدی، با بیان تمام مفاهیمی که در فصل قبل یاد گرفتیم، محیط جیم را بیشتر درک خواهیم کرد.

کاوش در محیط

در فصل قبل، آموختیم که محیط یادگیری تقویتی را می‌توان به عنوان فرآیند تصمیم گیری مارکوف (**MDP**) مدل کرد و **MDP** شامل موارد زیر است:

- **حالت‌ها:** مجموعه‌ای از ایستایش یا وضعیتهای موجود در محیط.
- **اقدامات:** مجموعه‌ای از **اعمال** یا **کنشهایی** است که عامل می‌تواند در هر حالت انجام دهد.
- **احتمال انتقال**[۱]: احتمال انتقال، با نماد $P(s'|s,a)$ نشان داده می‌شود. این نماد، به احتمال حرکت از حالت s به حالت s'، درحین انجام یک عمل a دلالت دارد.
- **تابع پاداش**[۲]: **تابع پاداش** با نماد $R(s,a,s')$ نشان داده می‌شود. این نماد به معنای **پاداشی** است که عامل هنگام انجام یک عمل a با حرکت از حالت s به حالت s' دریافت می‌کند.

اکنون، بیایید بفهمیم که چگونه می‌توان تمام اطلاعات فوق را از محیط دریاچه یخ زده‌ای که به تازگی با استفاده از جیم، ایجاد کردیم به دست‌آوریم.

حالتها

یک فضای حالت، شامل تمام حالتهای ما می‌شود. فقط با تایپ `env.observation_space` به صورت زیر، می‌توانیم تعداد حالت‌های موجود در محیط خود را بدست آوریم:

```
print(env.observation_space)
```

با این کد، مطالب زیر چاپ خواهد شد:

---

۱ Transition Probability
۲ Reward Function

```
Discrete(16)
```

به این معنی است که ما ۱۶ حالت گسسته در فضای حالت خود داریم که از حالت **S** شروع می شود و تا حالت **G** وجود دارد. توجه داشته باشید که در جیم، حالت‌ها به صورت یک عدد کدگذاری می‌شوند، بنابراین حالت **S** به صورت ۰، حالت **F** به صورت ۱ رمزگذاری[۱] می‌شود، و به همین ترتیب. همانطور که در شکل ۲.۵ نشان داده شده است:

0 S	1 F	2 F	3 F
4 F	5 H	6 F	7 H
8 F	9 F	10 F	11 H
12 H	13 F	14 F	15 G

شکل ۲.۵: شانزده حالت گسسته

اقدامات

ما آموختیم که **فضای کنش**، شامل تمام **اقدامات** ممکن در محیط است. ما می‌توانیم فضای کنش یا اقدام را با استفاده از `env.action_space` بدست آوریم:

```
print(env.action_space)
```

با این کد، مطلب زیر چاپ خواهد شد:

```
Discrete(4)
```

[۱] Encoded

این نشان می دهد که ما چهار عمل گسسته در فضای عمل خود داریم که عبارتند از چپ، *پایین*، *راست* و *بالا*. توجه داشته باشید که مانند حالت‌ها، در اینجا اقدامات نیز به صورت اعدادی رمزگذاری می‌شوند که در جدول ۲.۱ نشان داده شده است:

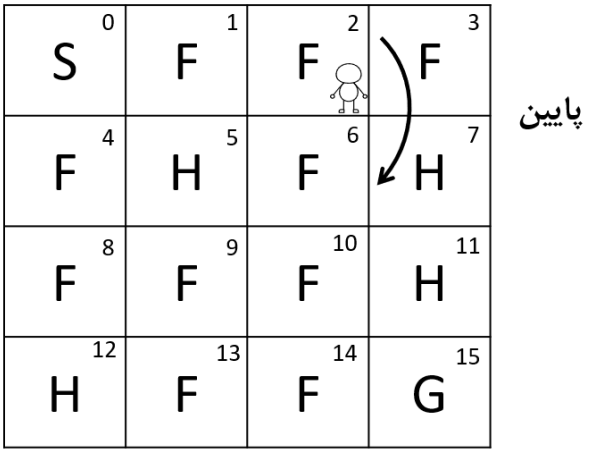
اقدام	شماره
چپ	۰
پایین	۱
راست	۲
بالا	۳

جدول ۲.۱: چهار اقدام گسسته

احتمال انتقال و تابع پاداش

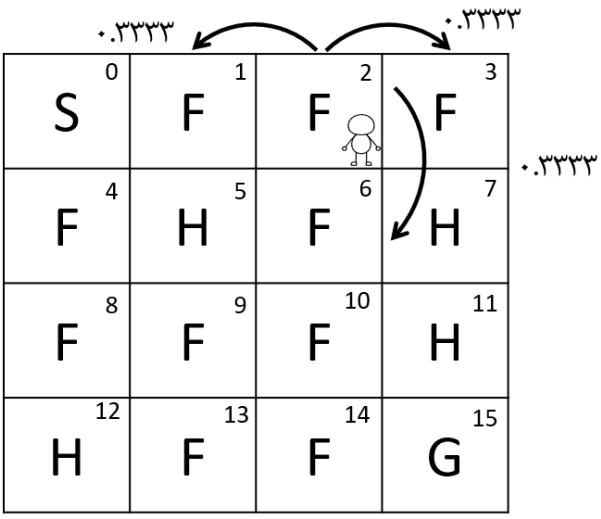
حال، بیایید نحوه به دست آوردن احتمال انتقال و **تابع پاداش** را بررسی کنیم. ما یاد گرفتیم که در **محیط اتفاقی**، نمی‌توان گفت که با انجام یک **اقدام** a، عامل همیشه به حالت بعدی خواهد رسید، دقیقاً به این دلیل که مقداری تصادفی بودن مرتبط با **محیط اتفاقی** وجود دارد و با انجام یک اقدام a در حالت s، عامل با کمی **احتمال** به حالت بعدی s' می‌رسد.

فرض کنید در حالت ۲ یعنی (**F**) هستیم. حال اگر اقدام ۱ (حرکت به *پایین*) را در حالت ۲ انجام دهیم، می‌توانیم به حالت ۶ مطابق شکل ۲.۶ برسیم:



شکل ۲.۶: عامل ما، اقدام حرکت به سمت *پایین* را در حالت ۲ انجام می‌دهد.

محیط دریاچه منجمد ما، یک **محیط اتفاقی** است. وقتی محیط ما **اتفاقی** است، اینگونه نیست که همیشه با انجام اقدام ۱ (حرکت به سمت *پایین*) در حالت ۲، به حالت ۶ برسیم و با **احتمال** کمی به حالت‌های دیگر هم می‌رسیم. بنابراین وقتی یک اقدام ۱ (حرکت به طرف *پایین*) را در حالت ۲ انجام می‌دهیم، به حالت ۱ به **احتمال** ۰.۳۳۳۳۳، به حالت ۶ به **احتمال** ۰.۳۳۳۳۳ و به حالت ۳ به **احتمال** ۰.۳۳۳۳۳ می رسیم، که در شکل ۲.۷ نشان داده شده است:



شکل ۲.۷: احتمال انتقالهای عامل در حالت ۲

همانطور که می بینیم، در یک **محیط اتفاقی**، با احتمال کمی به حالت‌های بعدی می‌رسیم. حالا بیایید یاد بگیریم که چگونه با استفاده از محیط جیم، این احتمال انتقال را بدست آوریم.

فقط با تایپ `env.P[state][action]` می‌توانیم **احتمال انتقال** و **تابع پاداش** را بدست آوریم. بنابراین، برای به دست آوردن احتمال انتقال از حالت **S**، به حالت‌های دیگر با انجام اقدام *راست*، می‌توانیم `env.P[S][right]` را تایپ کنیم. اما ما نمی‌توانیم فقط حالت **S** و اقدام حرکت به *راست* (*right*) را مستقیماً تایپ کنیم، زیرا آن‌ها به صورت اعداد کدگذاری می‌شوند. ما آموختیم که حالت **S** به صورت ۰ و اقدام حرکت به *راست* به صورت ۲ رمزگذاری شده است، بنابراین، برای به دست آوردن احتمال انتقال حالت **S** با انجام اقدام *راست*، `env.P[0][2]` را تایپ می‌کنیم که در زیر نشان داده شده است:

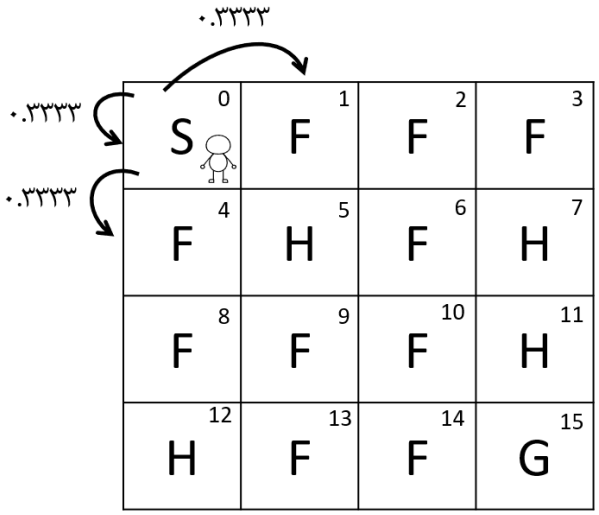
```
print(env.P[0][2])
```

کد بالا، برای ما خواهد نوشت:

```
[(0.33333, 4, 0.0, False),
 (0.33333, 1, 0.0, False),
 (0.33333, 0, 0.0, False)]
```

این به چه معناست؟ خروجی ما به صورت [(**احتمال انتقال**، حالت بعدی، **پاداش**، آیا در وضعیت پایان هستیم؟)] است. به این معنی که اگر یک **اقدام** ۲ (*راست*) را در حالت ۰ (**S**) انجام دهیم، آنگاه:

- به حالت ۴ (**F**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر دریافت می‌کنیم.
- به حالت ۱ (**F**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر دریافت می‌کنیم.
- به همان حالت ۰ (**S**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر دریافت می‌کنیم.



شکل ۲.۸: احتمال انتقال عامل در حالت صفر را نشان می‌دهد.

بنابراین، وقتی `env.P[state][action]` را تایپ می‌کنیم، نتیجه را به شکل [(**احتمال انتقال**، حالت بعدی، پاداش، حالت پایانی است؟)] دریافت می‌کنیم. آخرین پارامتر یک مقدار دوتایی[1] است و به ما می‌گوید که آیا حالت بعدی یک حالت پایانی است یا خیر. از آنجایی که ۴، ۱ و ۰ حالت‌های پایانی نیستند، خروجی false را خواهیم داشت.

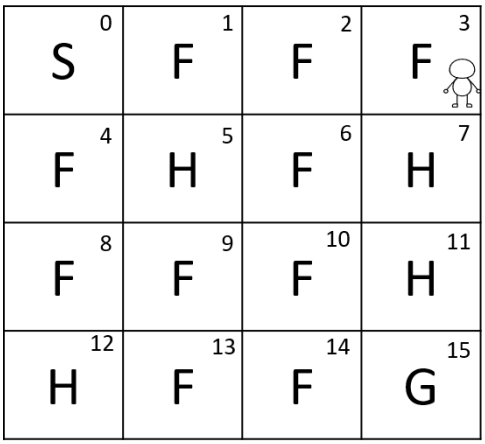
خروجی `env.P[0][2]`، برای وضوح بیشتر در جدول ۲.۲ نشان داده شده است:

احتمال انتقال (گذار)	حالت بعدی	پاداش	آیا در حالت پایانی هستیم؟
۰.۳۳۳۳	۴ (**F**)	۰.۰	خیر
۰.۳۳۳۳	۱ (**F**)	۰.۰	خیر
۰.۳۳۳۳	۰ (**S**)	۰.۰	خیر

جدول ۲.۲: خروجی دستور `env.P[0][2]`

بیایید با یک مثال دیگر این موضوع را وارسی کنیم. فرض کنید در حالت ۳ یا (**F**) هستیم همانطور که شکل ۲.۹ نشان می‌دهد:

[1] Boolean



شکل ۲.۹: عامل در حالت ۳

فرض کنید اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ یا (**F**) انجام می‌دهیم. سپس احتمال انتقال حالت ۳ یا (**F**) با انجام اقدام ۱ (حرکت به سمت *پایین*) را می‌توان به صورت زیر به دست آورد:

```
print(env.P[3][1])
```

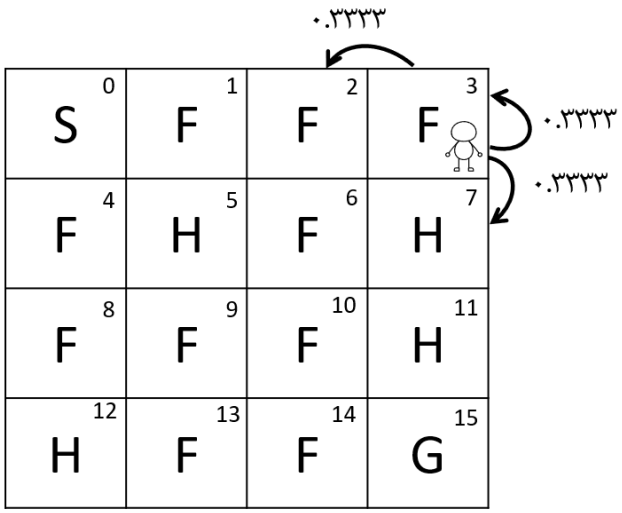
بر اساس این کد، مطالب زیر چاپ خواهد شد:

```
[(0.33333, 2, 0.0, False),
 (0.33333, 7, 0.0, True),
 (0.33333, 3, 0.0, False)]
```

همانطور که یاد گرفتیم، خروجی ما به شکل [(**احتمال انتقال**، حالت بعدی، **پاداش**، حالت پایانی است؟)] می‌باشد. این بدان معناست که اگر اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ (**F**) انجام دهیم، آنگاه:

- به حالت ۲ یا (**F**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر را دریافت می‌کنیم.
- به حالت ۷ یا (**H**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر را دریافت می‌کنیم.
- به همان حالت ۳ یا (**F**) با **احتمال** ۰.۳۳۳۳۳ می‌رسیم و **پاداش** صفر را دریافت می‌کنیم.

شکل ۲.۱۰ **احتمال انتقال** را نشان می‌دهد:



شکل ۲.۱۰: احتمال انتقالات عامل در حالت ۳

خروجی `env.P[3][1]`، برای وضوح بیشتر در جدول ۲.۳ نشان داده شده است:

آیا در حالت پایانی هستیم؟	پاداش	حالت بعدی	احتمال انتقال(گذار)
خیر	۰.۰	(**F**) ۲	۰.۳۳۳۳
بله	۰.۰	(**H**) ۷	۰.۳۳۳۳
خیر	۰.۰	(**F**) ۳	۰.۳۳۳۳

جدول ۲.۳: خروجی دستور `env.P[3][1]`

همانطور که مشاهده می‌کنیم، در ردیف دوم خروجی (`True` ، ۰.۰ ، ۷ ، ۰.۳۳۳۳) داریم و آخرین مقدار در اینجا به عنوان `True` مشخص شده است. به این معنی است که حالت ۷ یک حالت پایانی است. یعنی اگر اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ یا (**F**) انجام دهیم، با احتمال ۰.۳۳۳۳ به حالت ۷ یا (**H**) می‌رسیم و از آنجایی که ۷ یا (**H**) یک حفره است، عامل اگر به حالت ۷ برسد، می‌میرد (**H**). بنابراین، ۷ یا (**H**)، یک حالت پایانی است و بنابراین به عنوان `True` مشخص می‌شود.

لذا، ما یاد گرفتیم که چگونه فضای حالت، **فضای اقدام**، احتمال انتقال و **تابع پاداش** را با استفاده از محیط جیم، بدست آوریم. در بخش بعدی نحوه تولید یک اپیزود را یاد خواهیم گرفت.

ساخت پردینه (اپیزود) در محیط جیم

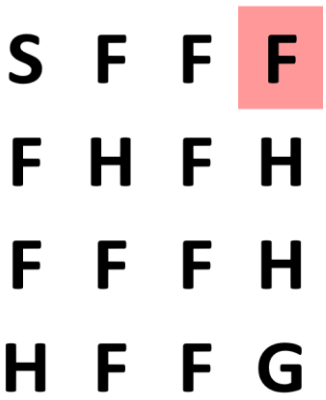
ما آموختیم که برهمکنش عامل–محیط که از حالت اولیه شروع شده و تا حالت پایانی ادامه دارد، یک **پردینه (یا اپیزود)** نامیده می‌شود. در این قسمت با نحوه تولید اپیزود در محیط جیم آشنا می‌شویم.

قبل از شروع، با تنظیم مجدد محیط خود؛ حالت را مقداردهی اولیه می‌کنیم. تنظیم مجدد، عامل ما را به حالت اولیه برمی‌گرداند. می‌توانیم با استفاده از تابع `reset()`، محیط خود را مطابق شکل زیر بازنشانی کنیم:

```
state = env.reset()
```

انتخاب اقدام

برای اینکه عامل، با محیط تعامل داشته باشد، باید **اقداماتی** را در محیط انجام دهد. پس ابتدا بیایید نحوه انجام یک **اقدام** در محیط جیم را بیاموزیم. فرض کنید در حالت ۳ یا (**F**) هستیم، همانطور که شکل ۲.۱۱ نشان می‌دهد:



شکل ۲.۱۱: عامل در حالت ۳ در محیط دریاچه یخزده قرار دارد.

فرض کنید باید اقدام ۱، (حرکت به سمت *پایین*) را انجام دهیم و به حالت جدید ۷ یا (**H**) برویم. چگونه می‌توانیم این کار را انجام دهیم؟ با استفاده از تابع `step`، می‌توانیم یک عمل را انجام دهیم. ما فقط باید اقدام خود را به عنوان پارامتر در تابع `step` وارد کنیم. بنابراین، می‌توانیم اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ یا (**F**) با استفاده از

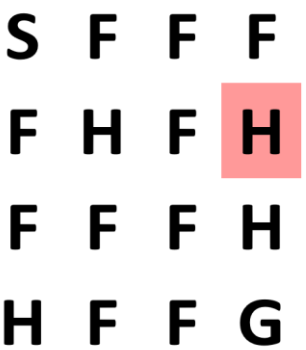
تابع `step` به صورت زیر انجام دهیم:

```
env.step(1)
```

حال بیایید محیط خود را با استفاده از تابع `render` مصورسازی کنیم:

```
env.render()
```

همانطور که در شکل ۲.۱۲ نشان داده شده است، عامل، اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ یا (**F**) انجام می‌دهد و به حالت بعدی ۷ یا (**H**) می‌رسد:



شکل ۲.۱۲: عامل در حالت ۷ در محیط دریاچه یخزده

توجه داشته باشید که هر زمان که با استفاده از دستور `env.step()` اقدامی انجام می‌دهیم، یک چندگانه[۱]، حاوی ۴ مقدار را خروجی می‌دهد. بنابراین، وقتی اقدام ۱ (حرکت به سمت *پایین*) را در حالت ۳ یا (**F**) با استفاده از `env.step(1)`، انجام می‌دهیم، که این دستور، خروجی را به صورت زیر می‌دهد:

```
(7, 0.0, True, {'prob': 0.33333})
```

همانطور که ممکن است حدس بزنید، به این معنی است که وقتی اقدام ۱ (حرکت به *پایین*) را در حالت ۳ یا (**F**) انجام

[۱] Tuple

می‌دهیم:

- به حالت بعدی ۷ یا (**H**) می‌رسیم.
- عامل، **پاداش** ۰.۰ را دریافت می‌کند.
- از آنجایی که حالت بعدی ۷ یا (**H**) یک حالت پایانی است، به عنوان `True` مشخص می‌شود.
- به حالت بعدی ۷ یا (**H**) با احتمال ۰.۳۳۳۳۳ می‌رسیم. بنابراین، ما فقط می‌توانیم این اطلاعات را به صورت زیر ذخیره کنیم:

```
(next_state, reward, done, info) = env.step(1)
```

که به قرار زیرند:

- `next_state`، حالت بعدی را نشان می‌دهد.
- `reward`، نشان‌دهنده پاداش به دست آمده است.
- `done`، نشاندهنده اتمام‌یافتگی (یا انجام‌شدگی)[۱] است. به این معنا که آیا پردینه (اپیزود) ما به پایان رسیده است یا خیر. یعنی اگر حالت بعدی، حالت پایانی باشد، اپیزود ما به پایان می‌رسد، بنابراین `done`، به عنوان `True` علامتگذاری می‌شود، در غیر این صورت به عنوان `False` برچسب‌گذاری می‌شود.
- `info`: به غیر از **احتمال انتقال**، در برخی موارد، اطلاعات دیگری را نیز به‌عنوان اطلاعات ذخیره می‌کنیم که برای اهداف اشکال‌زدایی[۲] استفاده می‌شود.

همچنین می‌توانیم از **فضای اقدام** خود نمونه‌برداری کنیم و یک عمل تصادفی (بختکی) برای کشف محیط خود انجام دهیم. ما می‌توانیم یک اقدام را با استفاده از تابع `sample`، نمونه‌برداری کنیم:

```
random_action = env.action_space.sample()
```

---

[۱] ممکن است در برنامه‌نویسی از "done" به‌عنوان نام متغیر استفاده کنید تا نشان دهد یک عملیات یا فرایند خاص به پایان رسیده است.

[۲] Debugging

پس از اینکه یک اقدام را از **فضای کنش** خود نمونه‌برداری کردیم، سپس اقدام نمونه‌برداری شده خود را با استفاده از تابع `step` انجام می‌دهیم:

```
next_state, reward, done, info = env.step(random_action)
```

اکنون که نحوه انتخاب اقدامات در محیط را یاد گرفتیم، بیایید ببینیم که چگونه یک پردینه (اپیزود) تولید کنیم.

تولید یک پردینه (اپیزود)

حالا بیایید یاد بگیریم که چگونه یک پردینه (بخشواره)، تولید کنیم. اپیزود بیانگر تعامل محیط–عامل است که از حالت اولیه شروع می‌شود و تا حالت پایانی ادامه دارد. عامل، با انجام اقدامی در هر حالت با محیط تعامل دارد. اگر عامل به حالت پایانی برسد، یک پردینه (اپیزود) پایان می‌یابد. بنابراین، در محیط دریاچه یخ زده، اگر عامل به حالت پایانی برسد، که یا حالت حفره (**H**) یا حالت هدف (**G**) است، پردینه (اپیزود) اپیزود پایان می‌یابد.

بیایید بیاموزیم که چگونه یک اپیزود را با خطمشی تصادفی (بختکی) تولید کنیم. ما آموختیم که خطمشی تصادفی یک اقدام تصادفی (دلبخواه) را در هر حالت انتخاب می‌کند. بنابراین، ما یک اپیزود را با انجام اقدامات تصادفی در هر حالت ایجاد خواهیم کرد. بنابراین برای هر مرحله زمانی در اپیزود، در هر حالت، یک اقدام تصادفی انجام می‌دهیم و اگر عامل به حالت پایانی برسد، اپیزود ما به پایان می‌رسد.

ابتدا بیایید تعداد گامهای زمانی را تنظیم کنیم:

```
num_timesteps = 20
```

برای هر گام زمانی:

```
for t in range(num_timesteps):
```

به طور تصادفی (دلبخواهی) یک **اقدام** را با نمونه‌برداری از **فضای اقدام** انتخاب کنید:

```
random_action = env.action_space.sample()
```

اقدام انتخاب شده را انجام دهید:

```
next_state, reward, done, info = env.step(random_action)
```

اگر حالت بعدی حالت پایانی است، آنگاه بایستید. این نشان می‌دهد که اپیزود ما به پایان می‌رسد:

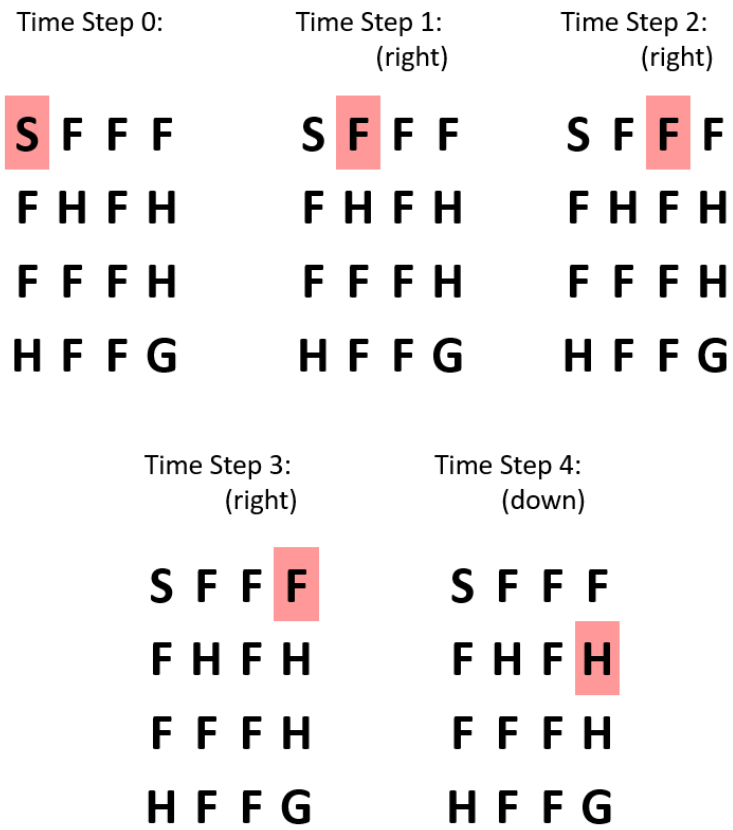
```
if done:
    break
```

قطعه کامل قبلی برای وضوح کار ارائه شده است. کد زیر نشان می‌دهد که در هر گام زمانی، یک اقدام را با نمونه‌برداری تصادفی (بختکی) از **فضای گنش** انتخاب می‌کنیم و اگر عامل به حالت پایانی برسد، اپیزود ما به پایان می‌رسد:

```
import gym
env = gym.make("FrozenLake-v0")

state = env.reset()

print('Time Step 0 :')
env.render()

num_timesteps = 20

for t in range(num_timesteps):
  random_action = env.action_space.sample()

  new_state, reward, done, info = env.step(random_action)
  print ('Time Step {} :'.format(t+1))


  env.render()

  if done:
    break
```

کد قبلی چیزی شبیه به شکل ۲.۱۳ را چاپ می‌کند. توجه داشته باشید که ممکن است هر بار که کد قبلی را اجرا می‌کنید، نتیجه متفاوتی دریافت کنید زیرا عامل در هر مرحله زمانی یک عمل تصادفی (دلبخواه) انجام می‌دهد.

همانطور که از خروجی زیر مشاهده می کنیم، در هر گام زمانی، عامل در هر حالت یک اقدام تصادفی (بختکی) انجام می‌دهد و پس از رسیدن عامل به حالت پایانی، اپیزود ما به پایان می‌رسد. همانطور که شکل ۲.۱۳ نشان می‌دهد، در گام زمانی ۴، عامل به حالت پایانی **H** می رسد و بنابراین اپیزود به پایان می‌رسد.



شکل ۲.۱۳: اقدامات انجام شده توسط عامل در هر گام زمانی

به‌جای تولید یک پردینه (اپیزود)، می‌توانیم با انجام برخی اقدامات تصادفی (دلبخواه) در هر حالت، مجموعه‌ای از اپیزودها را نیز تولید کنیم که کد آن در ادامه آمده است.

بنابراین، می‌توانیم با انتخاب یک اقدام تصادفی (بختکی) در هر حالت با نمونه‌برداری از فضای اقدام، یک پردینه (اپیزود) تولید کنیم. اما صبر کنید! این چه فایده‌ای دارد؟ چرا ما اصلا نیاز به تولید یک پردینه (بخشواره) داریم؟

در فصل قبل، یاد گرفتیم که یک عامل می‌تواند سیاست بهینه (یعنی اقدام صحیح در هر حالت) را با تولید چندین اپیزود پیدا کند. اما در مثال قبل، ما فقط در هر حالت، در تمام اپیزودها، اقدامات تصادفی انجام دادیم. چگونه عامل می‌تواند سیاست بهینه را پیدا کند؟ همینطور، در مورد محیط دریاچه یخ زده، چگونه عامل می‌تواند خطمشی بهینه‌ای را پیدا کند که به او می‌گوید بدون بازدید از حالت‌های حفره **H** از حالت **S** به حالت **G** برسد؟

```
import gym
env = gym.make("FrozenLake-v0")
```

```
num_episodes = 10
num_timesteps = 20

for i in range(num_episodes):

    state = env.reset()
    print('Time Step 0 :')
    env.render()

    for t in range(num_timesteps):
        random_action = env.action_space.sample()

        new_state, reward, done, info = env.step(random_action)
        print ('Time Step {} :'.format(t+1))


        env.render()
        if done:
            break
```

اینجاست که ما به یک الگوریتم یادگیری تقویتی نیاز داریم. یادگیری تقویتی همه چیز در مورد یافتن خطمشی بهینه است، یعنی سیاستی که به ما می‌گوید در هر حالت، چه اقدامی را انجام دهیم. در فصلهای آینده. ما یاد خواهیم گرفت که چگونه سیاست بهینه را با تولید مجموعه‌ای از اپیزودها، با استفاده از الگوریتم‌های مختلف یادگیری تقویتی پیدا کنیم. در این فصل، ما بر روی آشنایی با محیط جیم و عملکردهای مختلف جیم تمرکز خواهیم کرد، زیرا در طول دوره کتاب از محیط جیم استفاده خواهیم کرد.

تاکنون متوجه شده‌ایم که محیط جیم با استفاده از محیط اصلی دریاچه یخزده چگونه کار می‌کند، اما جیم دارای بسیاری از عملکردهای دیگر و همچنین چندین محیط جالب دیگر است. در بخش بعدی با سایر محیط‌های جیم همراه با بررسی عملکردهای جیم آشنا می‌شویم.

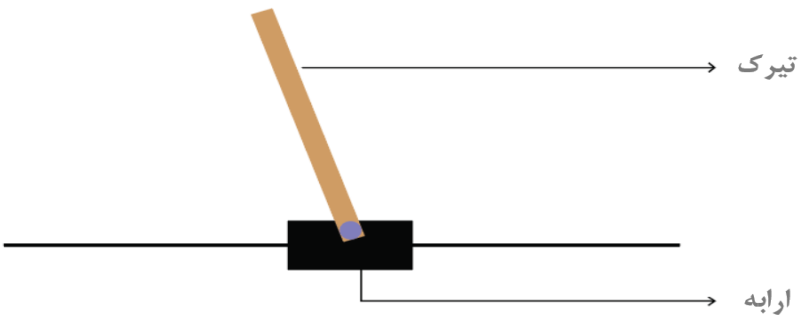
محیط‌های بیشتر جیم

در این بخش، چندین محیط جالب جیم را به همراه بررسی عملکردهای مختلف جیم، بررسی خواهیم کرد.

محیط‌های کنترل کلاسیک

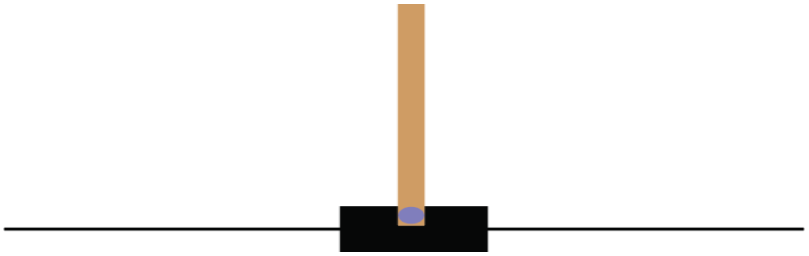
جیم، محیط‌هایی را برای چندین کار کنترلی کلاسیک، مانند: تعادل ارابه و تیرک (یا تعادل آونگ وارونه)[۱]، در وضعیت قائم (یا عمودی) قرار دادن پاندول معکوس[۲]، بالا رفتن خودرو از کوهستان[۳] و غیره فراهم می‌کند. بیایید یاد بگیریم، که چگونه یک محیط جیم برای یک کار **تعادل آونگ وارونه** ایجاد کنیم. محیط آونگ وارونه در زیر نشان داده شده است:

[۱] Cart-Pole Balancing
[۲] Inverted Pendulum
[۳] Mountain Car Climbing



شکل ۲.۱۴: مثال ارابه و تیرک (آونگ وارونه)

تعادل آونگ وارونه، یکی از مسائل کنترل کلاسیک است. همانطور که در شکل ۲.۱۴ نشان داده شده است، تیرک یا میله به گاری یا ارابه وصل شده است و هدف عامل ما متعادل کردن تیرک روی ارابه است، یعنی هدف عامل ما این است که، تیرک را مستقیماً روی گاری ایستاده نگه دارد. همانطور که در شکل ۲.۱۵، نشان داده شده است:



شکل ۲.۱۵: هدف این است که تیرک را عمود نگه دارید.

بنابراین، عامل سعی می‌کند ارابه را به چپ و راست هل دهد تا تیرک مستقیم روی ارابه بایستد. بنابراین، عامل ما دو عمل را انجام می‌دهد که عبارتند از هل دادن ارابه به طرف چپ و هل دادن ارابه به سمت *راست*، تا تیرک را مستقیماً روی ارابه به صورت ایستاده دربیاورد. همچنین می‌توانید یک ویدیوی بسیار جالب را در آدرس زیر مشاهده کنید: `https://youtu.be/qMlcsc43-lg` که نشان می‌دهد چگونه عامل یادگیری تقویتی، با حرکت ارابه به چپ و *راست*، تیرک را روی ارابه را متعادل می‌کند.

حالا بیایید نحوه ایجاد محیط Cart-Pole را با استفاده از جیم بیاموزیم. شناسه محیطی برای محیط آونگ وارونه در جیم، `CartPole-v0` است، بنابراین می‌توانیم از تابع `make` برای ایجاد محیط آونگ وارونه مانند زیر استفاده کنیم:

```
env = gym.make("CartPole-v0")
```

پس از ایجاد محیط، می توانیم محیط خود را با استفاده از تابع `render` مشاهده کنیم:

```
env.render()
```

همچنین می توانیم محیط رندر شده را با استفاده از تابع `close` ببندیم:

```
env.close()
```

فضای حالت

حالا بیایید به فضای حالت محیط آونگ وارونه، نگاه کنیم. صبر کنید! حالت‌های اینجا چیست؟ در محیط دریاچه یخ زده، ما شانزده حالت گسسته از **S** تا **G** داشتیم. اما چگونه می‌توانیم حالت‌ها را در اینجا توصیف کنیم؟ آیا می‌توانیم وضعیت را با موقعیت ارابه توصیف کنیم؟ بله! توجه داشته باشید که موقعیت ارابه، یک مقدار پیوسته است. بنابراین، در این حالت، فضای حالت ما مقادیر پیوسته خواهد بود، برخلاف محیط دریاچه یخ زده که فضای حالت ما مقادیر گسسته (**S** تا **G**) داشت.

اما تنها با موقعیت ارابه نمی‌توانیم حالت محیط را به طور کامل توصیف کنیم. بنابراین سرعت ارابه، زاویه تیرک و سرعت تیرک در انتهای آن[۱] را در نظر می‌گیریم. بنابراین می‌توانیم فضای حالت خود را با آرایه‌ای از مقادیر به صورت زیر توصیف کنیم:

```
array([cart position, cart velocity, pole angle, pole velocity at the
tip])
```

توجه داشته باشید که همه این مقادیر پیوسته هستند، یعنی:

۱. مقدار موقعیت ارابه از ۴.۸- تا ۴.۸ متغیر است.

[۱] Pole Velocity at the Tip

۲. مقدار سرعت ارابه از `-inf` تا `inf` یعنی (∞- تا ∞) متغیر است.
۳. مقدار زاویه تیرک از ۰.۴۱۸- رادیان تا ۰.۴۱۸ رادیان است.
٤. مقدار سرعت تیرک در انتهای آن از `-inf` تا `inf` متغیر است.

بنابراین، فضای حالت ما حاوی آرایه‌ای از مقادیر پیوسته است. بیایید یاد بگیریم که چگونه می‌توانیم این را از جیم بدست آوریم. برای بدست آوردن فضای حالت و مشاهده آن، کافی است دستور آنرا بصورت زیر تایپ کنیم: `env.observation_space`

```
print(env.observation_space)
```

با ورود این کد، مقدار زیر چاپ خواهد شد:

```
Box(4,)
```

Box، نشان می‌دهد که فضای حالت ما از مقادیر پیوسته تشکیل شده است نه مقادیر گسسته. یعنی در محیط دریاچه یخ زده فضای حالت را به صورت `Discrete(16)` به دست آوردیم که نشان می‌دهد ۱۶ حالت گسسته (**S** تا **G**) داریم. اما اکنون فضای حالت خود را با `Box(4,)` نشان می‌دهیم، که نشان می‌دهد فضای حالت ما پیوسته است و از آرایه‌ای از ۴ مقدار تشکیل شده است.

برای مثال، اجازه دهید محیط خود را بازنشانی کنیم و ببینیم فضای حالت اولیه ما چگونه خواهد بود. با استفاده از تابع `reset` می‌توانیم محیط را بازنشانی کنیم:

```
print(env.reset())
```

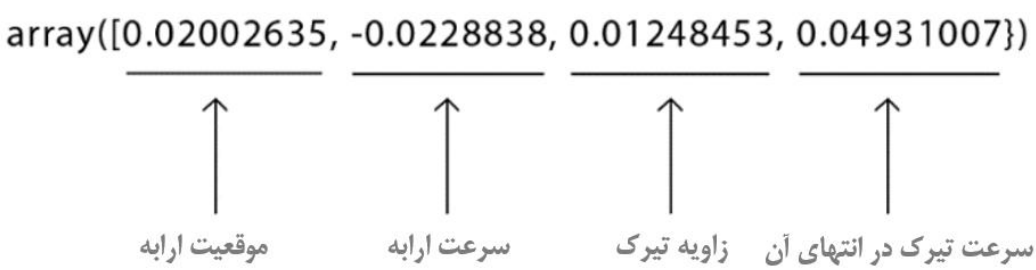
باکد قبلی، مقادیر زیر چاپ خواهد شد:

```
array([ 0.02002635, -0.0228838 ,  0.01248453,  0.04931007])
```

توجه داشته باشید که در اینجا فضای حالت به طور تصادفی (بختکی) مقداردهی اولیه می‌شود و بنابراین هر بار که کد

قبلی را اجرا می‌کنیم، مقادیر متفاوتی دریافت خواهیم کرد.

نتیجه کد قبلی نشان می‌دهد که فضای حالت اولیه ما شامل یک آرایه از ۴ مقدار است که به ترتیب موقعیت ارابه، سرعت ارابه، زاویه میله و سرعت تیرک در انتهای آن را نشان می‌دهد. یعنی:



شکل ۲.۱۶: فضای حالت اولیه

خب، چگونه می‌توانیم مقادیر حداکثر و حداقل فضای حالت خود را بدست آوریم؟ ما می‌توانیم حداکثر مقادیر فضای حالت خود را با استفاده از دستور `env.observation_space.high` بدست آوریم به همین ترتیب، حداقل مقادیر فضای حالت ما با استفاده از دستور `env.observation_space.low` قابل دستیابی است.

به عنوان مثال، بیایید به حداکثر مقدار فضای حالت خود نگاه کنیم:

```
print(env.observation_space.high)
```

کد قبلی چاپ خواهد شد:

```
[4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38]
```

دلالت بر این دارد که:

۱. حداکثر مقدار موقعیت ارابه برابر ۴.۸ است.

۲. ما آموختیم که حداکثر مقدار سرعت گاری `+inf` است و می‌دانیم که بی‌نهایت در واقع یک عدد نیست، بنابراین با استفاده از بزرگترین مقدار واقعی مثبت، یعنی $3.4028235e^{38}$ نمایش داده می‌شود.

۳. حداکثر مقدار زاویه تیرک ۰.۴۱۸ رادیان است.

٤. حداکثر مقدار سرعت تیرک، در انتهای میله `+inf` است، بنابراین با استفاده از بزرگترین مقدار واقعی مثبت $3.402823e^{38}$ نشان داده می‌شود.

به طور مشابه، ما می‌توانیم حداقل مقدار فضای حالت خود را به صورت زیر بدست آوریم:

```
print(env.observation_space.low)
```

کد قبلی، مقادیر زیر را چاپ خواهد کرد:

```
[-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38]
```

این، بیان می‌کند که:

۱. حداقل مقدار موقعیت ارابه ۴.۸- است.

۲. ما آموختیم که حداقل مقدار سرعت ارابه `-inf` است و می‌دانیم که بی‌نهایت یک عدد نیست، بنابراین با استفاده از بزرگترین مقدار واقعی منفی، $-3.402823e^{38}$ نشان داده می‌شود.

۳. حداقل مقدار زاویه تیرک ۰.۴۱۸- رادیان است.

٤. حداقل مقدار سرعت تیرک در نوک `-inf` است، بنابراین با استفاده از بزرگترین مقدار واقعی منفی $-3.402823e^{38}$ نشان داده می‌شود.

فضای کنش

حالا بیایید به فضای کنش نگاه کنیم. قبلاً یاد گرفتیم که در محیط Cart-Pole، دو اقدام انجام می‌دهیم که عبارتند از هل‌دادن ارابه یا گاری به چپ و هل‌دادن آن به راست و بنابراین فضای کنش، گسسته است زیرا فقط دو اقدام گسسته داریم.

برای به دست آوردن فضای کنش، فقط می‌توانیم `env.action_space` را همانطور که در زیر اشاره شده است،

تایپ کنیم:

```
print(env.action_space)
```

با این کد، مقادیر زیر چاپ خواهد شد:

```
Discrete(2)
```

همانطور که مشاهده می‌کنیم، `Discrete(2)`، به این معنی است که فضای کنش ما گسسته است و ما دو عمل در فضای اقدام خود داریم. توجه داشته باشید که اقدامات به صورت اعدادی مانند جدول ۲.۴ کدگذاری می‌شوند:

شماره	اقدام
۰	هل دادن ارابه به سمت چپ
۱	هل دادن ارابه به سمت راست

جدول ۲.۴: دو اقدام ممکن برای حرکت آونگ وارونه

تعادل آونگ وارونه با سیاست تصادفی

بیایید یک عامل با **خط‌مشی** تصادفی (دلخواه) ایجاد کنیم، یعنی عاملی را ایجاد کنیم که یک **عمل** تصادفی (بختکی) را در محیط انتخاب کرده و سعی می‌کند تیرک را متعادل کند. هر بار که تیرک مستقیم روی ارابه می‌ایستد، عامل، یک جایزه ۱+ دریافت می‌کند. ما بیش از ۱۰۰ پردینه (اپیزود) تولید خواهیم کرد و شاهد **بازگشت** یا **بازده** (یعنی مجموع **پاداش‌ها**) در هر پردینه (اپیزود) خواهیم بود. بیایید این را گام به گام یاد بگیریم.

ابتدا، بیایید محیط Cart-Pole خود را ایجاد کنیم:

```
import gym
env = gym.make('CartPole-v0')
```

تعداد پردینه‌ها (اپیزودها) و تعداد مراحل زمانی در هر پردینه (اپیزود) را تنظیم کنید:

```
num_episodes = 100
num_timesteps = 50
```

برای هر پردینه (اپیزود):

```
for i in range(num_episodes):
```

بازده یا `return` را روی ۰ تنظیم کنید:

```
    Return = 0
```

با تنظیم مجدد محیط، حالت را راه‌اندازی کنید:

```
    state = env.reset()
```

حلقه‌ای برای هر گام از پردینه (اپیزود) ایجاد کنید:

```
    for t in range(num_timesteps):
```

محیط را رندر کنید:

```
        env.render()
```

به طور تصادفی (دلبخواهی) یک **اقدام** را با نمونه‌برداری از محیط انتخاب کنید:

```
        random_action = env.action_space.sample()
```

اقدام انتخاب شده تصادفی (یا بختکی) را انجام دهید:

```
        next_state, reward, done, info = env.step(random_action)
```

بازده (بازگشت) را به روزرسانی کنید:

```
        Return = Return + reward
```

اگر حالت بعدی حالت پایانی است، اپیزود را تمام کنید:

```
        if done:
            break
```

برای هر ۱۰ پردینه (بخشواره)، بازده (مجموع **پاداش‌ها**) را چاپ کنید:

```
    if i%10==0:
        print('Episode: {}, Return: {}'.format(i, Return))
```

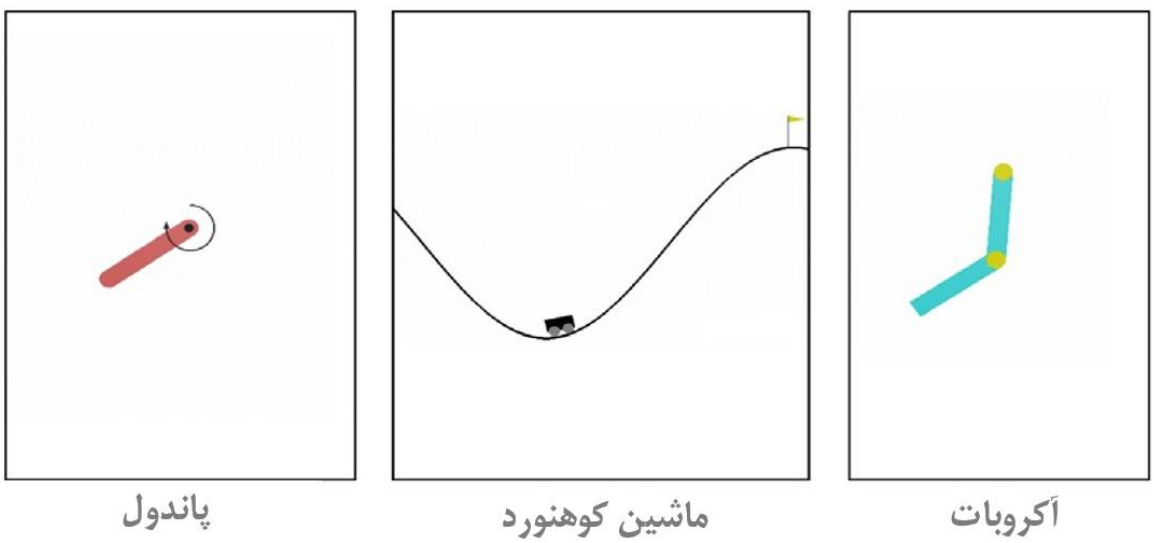
محیط را ببندید:

```
env.close()
```

کد قبلی مجموع **پاداش‌های** به دست آمده در هر ۱۰ پردینه (اپیزود) را بعنوان خروجی می‌دهد:

```
Episode: 0, Return: 14.0
Episode: 10, Return: 31.0
Episode: 20, Return: 16.0
Episode: 30, Return: 9.0
Episode: 40, Return: 18.0
Episode: 50, Return: 13.0
Episode: 60, Return: 25.0
Episode: 70, Return: 21.0
Episode: 80, Return: 17.0
Episode: 90, Return: 14.0
```

بنابراین، در اینجا ما با یکی از مسائل کنترلی جالب و کلاسیک به نام تعادل ارابه و تیرک (آونگ وارونه) و نحوه ایجاد محیط متعادل‌کننده ارابه با استفاده از جیم آشنا شدیم. جیم، چندین محیط کنترل کلاسیک دیگر را همانطور که در شکل ۲.۱۷ نشان داده شده است فراهم می‌کند:



شکل ۲.۱۷: محیطهای کنترل کلاسیک

شما همچنین می توانید با ایجاد هر یک از محیطهای بالا با استفاده از جیم، آزمایشهای جالبی را انجام دهید. شما می‌توانیم تمام محیطهای کنترل کلاسیک ارائه شده توسط جیم را در آدرس زیر بررسی کنیأ:

```
https://gym.openai.com/envs/#classic_control
```

محیطهای بازی آتاری

آیا شما از طرفداران بازیهای آتاری هستید؟ اگر بله، پس این بخش برای شما جالب خواهد بود. آتاری ۲۶۰۰ یک کنسول بازی ویدیویی از یک شرکت بازی‌ساز به نام آتاری است. کنسول بازی آتاری، چندین بازی محبوب را ارائه می‌دهد که شامل Pong، Space Invaders، Ms. Pac-Man، Break Out، Centipede و بسیاری دیگر می‌شود. آموزش عامل یادگیری تقویتی ما برای بازیهای آتاری، یک کار جالب و همچنین چالش‌برانگیز است. اغلب، بیشتر الگوریتمهای یادگیری تقویتی، در محیطهای بازی آتاری، برای ارزیابی دقت الگوریتم آزمایش می‌شوند.

در این قسمت با نحوه ساخت محیط بازی آتاری با استفاده از جیم آشنا می‌شویم. جیم حدود ۵۹ محیط بازی آتاری از جمله Pong، Space Invaders، Air Raid، Asteroids، Centipede، Ms. Pac-Man و غیره را فراهم می‌کند. برخی از محیطهای بازی آتاری ارائه شده توسط جیم، در شکل ۲.۱۸ نشان داده شده است تا شما را هیجان‌زده نگه دارد:

شکل ۲.۱۸: محیطهای بازی آتاری

هر محیط بازی آتاری در جیم، دارای ۱۲ نوع یا گونه مختلف است. اجازه دهید این موضوع را با محیط بازی Pong درک کنیم. محیط بازی Pong دارای ۱۲ نوع یا گونه مختلف خواهد بود که در قسمت‌های بعدی توضیح داده شده است.

محیط عمومی

- **Pong-v0 و Pong-v4**: می‌توانیم یک محیط Pong با شناسه محیط[۱] به صورت Pong-v0 یا Pong-v4 ایجاد کنیم. خیلی هم خب، اما حالت محیط ما چگونه است؟ از آنجایی که ما با محیط بازی سروکار داریم، می توانیم فقط تصویر صفحه بازی خود را به عنوان **حالت** در نظر بگیریم. اما نمی‌توانیم مستقیماً با تصویر خام برخورد کنیم، بنابراین مقادیر پیکسل صفحه بازی خود را به عنوان حالت در نظر می‌گیریم. در بخش آینده بیشتر در این مورد خواهیم آموخت.

- **Pong-ram-v0 و Pong-ram-v4**: به ترتیب مشابه Pong-v0 و Pong-v4 است. با این حال، در اینجا، حالت محیط، RAM دستگاه آتاری است که به جای مقادیر پیکسل صفحه نمایش، تنها ۱۲۸ بایت است.

محیط قطعی

- **PongDeterministic-v0 و PongDeterministic-v4**: در این گونه خاص، همانطور که از نامش پیداست، هر بار که محیط را مقداردهی اولیه کنیم، موقعیت اولیه بازی یکسان خواهد بود و حالت محیط، مقادیر پیکسل صفحه بازی است.

- **Pong-ramDeterministic-v0 و Pong-ramDeterministic-v4**: این دو گونه بازی، به ترتیب مشابه PongDeterministic-v0 و PongDeterministic-v4 هستند، اما در اینجا حالت، RAM دستگاه آتاری است.

[۱] Environment id

بدون پرش فریم[۱]

- **PongNoFrameskip-v0** و **PongNoFrameskip-v4**: در این گونه بازی، هیچ فریم بازی رد نمی‌شود. تمام صفحه‌های بازی برای نماینده قابل مشاهده هستند و حالت، مقدار پیکسل صفحه بازی است.
- **Pong-ramNoFrameskip-v0** و **Pong-ramNoFrameskip-v4**: این گونه بازی، شبیه به PongNoFrameskip-v0 و PongNoFrameskip-v4 است، اما در اینجا حالت همانا RAM دستگاه آتاری است.

بنابراین در محیط آتاری، حالت محیط ما گاهی صفحه نمایش بازی و گاهی رم دستگاه آتاری خواهد بود. توجه داشته باشید که همانند بازی Pong، سایر بازی‌های آتاری در محیط جیم، دارای شناسه یکسانی هستند. برای مثال، فرض کنید می‌خواهیم یک محیط Space Invaders قطعی ایجاد کنیم، می‌توانیم آن را با شناسه `SpaceInvadersDeterministic-v0` ایجاد کنیم. فرض کنید می‌خواهیم یک محیط Space Invaders بدون پرش فریم ایجاد کنیم، در اینجا می‌توانیم آن را با شناسه `SpaceInvadersNoFrameskip-v0` ایجاد کنیم. شما می‌توانید تمام محیط‌های بازی آتاری را که توسط Gym ارائه می‌شود در ادرس زیر بررسی کنید:

https://gym.openai.com/envs/#atari

فضای حالت و عمل

حال بیایید فضای حالت و فضای کنش محیط‌های بازی آتاری را با جزئیات بررسی کنیم.

فضای حالت

[۱] No Frame Skipping

در این قسمت، فضای حالت بازیهای آتاری در محیط جیم را بررسی می‌کنیم. بیایید این موضوع را با بازی Pong یاد بگیریم. متوجه شدیم که در محیط آتاری، حالت محیط یا مقادیر پیکسل صفحه بازی یا رم دستگاه آتاری خواهد بود. ابتدا بیایید فضای حالت را درک کنیم که در آن وضعیت محیط، مقادیر پیکسل صفحه نمایش بازی است.

بیایید یک محیط Pong با تابع `make` ایجاد کنیم:

```
env = gym.make("Pong-v0")
```

در اینجا، صفحه بازی، حالت محیط ماست. بنابراین، ما فقط تصویر صفحه بازی را به عنوان حالت درنظر می‌گیریم. با این حال، ما نمی‌توانیم مستقیماً با تصاویر خام برخورد کنیم، بنابراین مقادیر پیکسل تصویر (صفحه نمایش بازی) را به عنوان حالت خود در نظر می‌گیریم. ابعاد پیکسل تصویر ۳ خواهد بود که شامل ارتفاع تصویر، عرض تصویر و تعداد کانال است.

بنابراین، حالت محیط ما آرایه‌ای خواهد بود که حاوی مقادیر پیکسل صفحه بازی است:

```
[Image height, image width, number of the channel]
```

توجه داشته باشید که مقادیر پیکسل از ۰ تا ۲۵۵ متغیر است. برای بدست آوردن فضای حالت، فقط می‌توانیم `env.observation_space` را همانطور که در زیر نشان داده‌ایم، تایپ کنیم:

```
print(env.observation_space)
```

با این کد، مقادیر زیر چاپ خواهد شد:

```
Box(210, 160, 3)
```

این کد، نشان می‌دهد که فضای حالت ما یک آرایه سه بعدی با شکل [۲۱۰,۱۶۰,۳] است. همانطور که آموختیم، ۲۱۰ نشان‌دهنده ارتفاع تصویر، ۱۶۰ نشان دهنده عرض تصویر، و ۳ نشان دهنده تعداد کانال‌ها است.

به عنوان مثال، ما می‌توانیم محیط خود را تنظیم مجدد کنیم و ببینیم که فضای حالت اولیه چگونه به نظر می‌رسد. با

استفاده از تابع reset می‌توانیم محیط را بازنشانی کنیم:

```
print(env.reset())
```

کد قبلی، آرایه‌ای را چاپ می‌کند که مقدار پیکسل صفحه نمایش اولیه بازی را نشان می‌دهد.

حال، بیایید یک محیط Pong ایجاد کنیم که حالت محیط ما به جای مقدار پیکسل صفحه نمایش بازی، رم دستگاه آتاری باشد:

```
env = gym.make("Pong-ram-v0")
```

بیایید به فضای حالت نگاه کنیم:

```
print(env.observation_space)
```

کد قبلی، مقادیر زیر را چاپ خواهد کرد:

```
Box(128,)
```

این بدان معناست که فضای حالت ما یک آرایه ۱ بعدی حاوی ۱۲۸ مقدار است. می‌توانیم محیط خود را تنظیم مجدد کنیم و ببینیم که فضای حالت اولیه چگونه به نظر می‌رسد:

```
print(env.reset())
```

توجه داشته باشید که این مورد، برای همه بازیهای آتاری در محیط جیم صدق می‌کند، به عنوان مثال، اگر یک محیط مهاجم فضایی[۱] با مقدار پیکسل صفحه بازی به عنوان حالت محیط خود ایجاد کنیم، فضای حالت ما یک آرایه سه بعدی به شکل Box(210, 160, 3) خواهد بود. با این حال، اگر محیط Space Invaders را با رم ماشین آتاری به عنوان حالت محیط خود ایجاد کنیم، فضای حالت ما یک آرایه با شکل Box(128,) خواهد بود.

[۱] Space Invaders

فضای کنش

حال بیایید **فضای اقدام** (یا **فضای کنش**) را بررسی کنیم. به طور کلی، محیط بازی آتاری دارای ۱۸ اقدام در فضای کنش است و اقدامات، مطابق جدول ۲.۵، از ۰ تا ۱۷ کدگذاری شده‌اند:

Name	Action
0	Noop
1	Fire
2	Up
3	Right
4	Left
5	Down
6	Up Right
7	Up Left
8	Down Right
9	Down Left
10	Up Fire
11	Right Fire
12	Left Fire
13	Down Fire
14	Up Right Fire
15	Up Left Fire
16	Down Right Fire
17	Down Left Fire

جدول ۲.۵: اقدامات محیط بازی آتاری

توجه داشته باشید که تمام ۱۸ اقدام قبلی برای همه محیط‌های بازی آتاری قابل اجرا نیستند و **فضای اقدام**، از یک بازی به بازی دیگر متفاوت است. به عنوان مثال، برخی از بازیها تنها از شش اقدام قبلی به عنوان **فضای کنش** خود استفاده می‌کنند و برخی از بازیها تنها از ۹ اقدام قبلی به عنوان فضای اقدام خود استفاده می‌کنند، در حالی که برخی دیگر از تمام ۱۸ اقدام قبلی استفاده می‌کنند. بیایید این را با یک مثال با استفاده از بازی Pong درک کنیم:

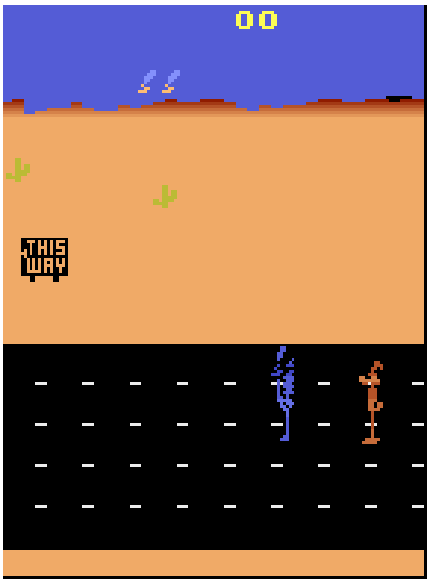
```
env = gym.make("Pong-v0")
print(env.action_space)
```

کد قبلی، مقدار زیر چاپ خواهد شد:

```
Discrete(6)
```

این کد نشان می‌دهد که در **فضای کنش** Pong، ما ۶ اقدام داریم که اقدامات از ۰ تا ۵ کدگذاری می‌شوند. بنابراین اقدامات ممکن در بازی Pong عبارتند از هیچ[۱] (بدون هیچگونه عملی یا بی‌خیال)، *آتش*، *بالا*، *راست*، چپ و *پایین*.

حال بیایید نگاهی به **فضای اقدام** بازی کوکوی (فاخته) دونده[۲] داشته باشیم. در صورتی که تا به حال با این بازی برخورد نکرده‌اید، در اینجا صفحه بازی به صورت زیر ارائه شده است:



شکل ۲.۱۹: محیط کوکوی دونده (فاخته یا صلصل)

بیایید، **فضای کنش** بازی کوکوی دونده را ببینیم:

---

[۱] Noop (No Action)
[۲] Road Runner

```
env = gym.make("RoadRunner-v0")
print(env.action_space)
```

با کد قبلی، مقدار زیر چاپ خواهد شد:

```
Discrete(18)
```

این کد، به ما نشان می‌دهد که **فضای کنش**، در بازی Road Runner شامل تمام ۱۸ عمل است.

معرفی یک برنامه برای بازی تنیس

در این بخش، بیایید نحوه ایجاد یک عامل برای بازی تنیس را بررسی کنیم. بیایید یک عامل با یک سیاست تصادفی (بختکی) ایجاد کنیم، به این معنی که عامل، به طور تصادفی (دلبخواه) یک عمل را از فضای عمل انتخاب کرده و عمل انتخابی تصادفی را انجام می‌دهد.

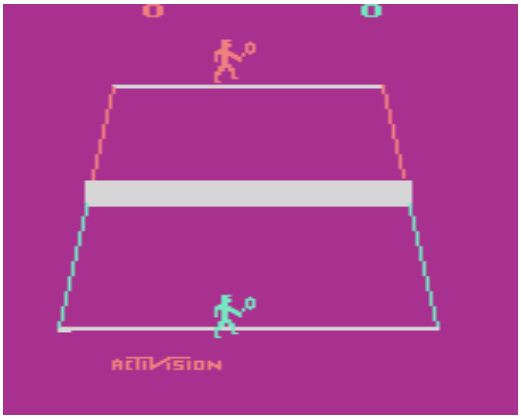
ابتدا بیایید محیط تنیس خود را ایجاد کنیم:

```
import gym
env = gym.make('Tennis-v0')
```

حال بیایید محیط تنیس را ببینیم:

```
env.render()
```

کد قبلی وضعیت زیر را نمایش می دهد:



شکل ۲.۲۰: محیط بازی تنیس

تعداد اپیزودها و تعداد گامهای زمانی را در اپیزود تنظیم کنید:

```
num_episodes = 100
num_timesteps = 50
```

برای هر پردینه (اپیزود):

```
for i in range(num_episodes):
```

بازده یا بازگشت را روی ۰ تنظیم کنید:

```
    Return = 0
```

با تنظیم مجدد محیط، حالت را راه اندازی کنید:

```
    state = env.reset()
```

برای هر گام از پردینه (بخشواره):

```
    for t in range(num_timesteps):
```

محیط را رندر کنید:

```
        env.render()
```

به طور تصادفی (دلبخواهی) یک اقدام را با نمونه‌برداری از محیط انتخاب کنید:

```
        random_action = env.action_space.sample()
```

عمل انتخاب شده به صورت تصادفی را انجام دهید:

```
        next_state, reward, done, info = env.step(random_action)
```

بازده را به‌روز رسانی کنید:

```
        Return = Return + reward
```

اگر حالت بعدی، حالت پایانی است، پردینه (اپیزود) را تمام کنید:

```
        if done:
            break
```

برای هر ۱۰ پردینه (اپیزود)، بازگشت (مجموع **پاداش‌ها**) را چاپ کنید:

```
    if i%10==0:
        print('Episode: {}, Return: {}'.format(i, Return))
```

محیط را ببندید:

```
env.close()
```

کد قبلی بازده (مجموع **پاداش‌ها**) به دست آمده از هر ۱۰ پردینه (اپیزود) را خروجی می‌دهد:

```
Episode: 0, Return: -1.0
Episode: 10, Return: -1.0
Episode: 20, Return: 0.0
Episode: 30, Return: -1.0
Episode: 40, Return: -1.0
Episode: 50, Return: -1.0
Episode: 60, Return: 0.0
Episode: 70, Return: 0.0
Episode: 80, Return: -1.0
Episode: 90, Return: 0.0
```

ضبط بازی

ما به تازگی یاد گرفتیم که چگونه یک عامل ایجاد کنیم که به طور تصادفی (بختکی) یک عمل را از فضای اقدام (کنش) انتخاب کرده و بازی تنیس را انجام دهد. آیا می‌توانیم بازی عامل را هم ضبط کنیم و به صورت ویدیویی ذخیره کنیم؟ بله! جیم یک کلاس پوشش[۱] ارائه می‌کند که می‌توانیم از آن برای ذخیره بازی عامل به‌عنوان ویدیو استفاده کنیم.

برای ضبط بازی، سیستم ما باید از FFmpeg پشتیبانی کند. FFmpeg چارچوبی است که برای پردازش فایل‌های رسانه‌ای استفاده می‌شود. بنابراین قبل از حرکت به جلو، مطمئن شوید که سیستم شما از FFmpeg، پشتیبانی می‌کند.

همانطور که کد زیر نشان می‌دهد، می‌توانیم بازی خود را با استفاده از پوشش مانیتور[۲] ضبط کنیم. سه پارامتر را نیاز داریم: محیط، دایرکتوری که می‌خواهیم ضبط‌های خود را در آن ذخیره کنیم و گزینه `force`. اگر این گزینه را بصورت `force=False` تنظیم کنیم، به این معنی است که هر بار که می‌خواهیم ضبط‌های جدید را ذخیره کنیم، باید یک دایرکتوری جدید ایجاد کنیم، و وقتی قرار دهیم `force=True`، ضبط‌های قدیمی در دایرکتوری، پاک

[۱] Wrapper Class
[۲] Monitor Wrapper

می‌شوند و با ضبط‌های جدید جایگزین می‌شوند:

```
env = gym.wrappers.Monitor(env, 'recording', force=True)
```

الان فقط کافی است پس از ایجاد محیط خود، کد قبلی را اضافه کنیم. بیایید یک مثال ساده بزنیم و ببینیم ضبط بازی چگونه انجام می‌شود. اجازه دهید عامل شما، بازی تنیس را به صورت تصادفی (بختکی) برای یک پردینه (بخشواره)، انجام دهد و روند بازی[۱] عامل را در قالب یک ویدیو ضبط کنید:

```
import gym
env = gym.make('Tennis-v0')

#Record the game
env = gym.wrappers.Monitor(env, 'recording', force=True)

env.reset()

for _ in range(5000):

    env.render()
    action = env.action_space.sample()
    next_state, reward, done, info = env.step(action)


    if done:
        break
env.close()
```

پس از پایان اپیزود، دایرکتوری جدیدی به نام ضبط را مشاهده خواهیم کرد و می‌توانیم فایل ویدیویی را با فرمت MP4 در این فهرست پیدا کنیم که روند بازی عامل ما را مطابق شکل ۲.۲۱ نشان می‌دهد:

[۱] Gameplay

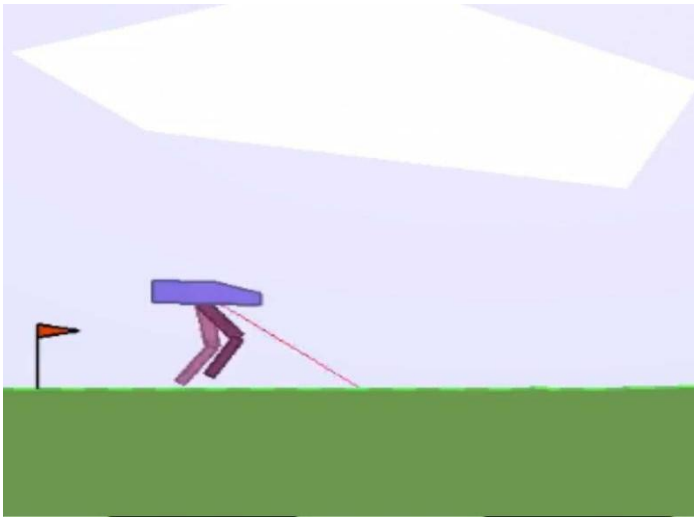
شکل ۲.۲۱: نمایش روند بازی تنیس

محیط‌های دیگر

جدا از کنترل کلاسیک و محیط‌های بازی آتاری که در مورد آن صحبت کردیم، جیم، چندین دسته مختلف از محیط را نیز ارائه می‌دهد. بیایید در مورد آنها بیشتر بدانیم.

Box2D

Box2D یک شبیه‌ساز دوبعدی است که عمدتاً برای آموزش عامل ما برای انجام کارهای کنترلی مداوم مانند راه رفتن استفاده می‌شود. به عنوان مثال، جیم، یک محیط Box2D به نام `BipedalWalker-v2` ارائه می‌دهد که می‌توانیم از آن برای آموزش راه رفتن عامل خود استفاده کنیم. محیط `BipedalWalker-v2` در شکل ۲.۲۲ نشان داده شده است:



شکل ۲.۲۲: محیط واکر دوپا

ما می‌توانیم چندین محیط دیگر Box2D ارائه شده توسط جیم را در اینجا بررسی کنیم:

https://gym.openai.com/envs/#box2d.

MuJoCo

MuJoCo سر واژگان **Multi-Joint dynamics with Contact** است و یکی از محبوب‌ترین شبیه‌سازهایی است که برای آموزش عامل ما برای انجام کارهای کنترلی مداوم استفاده می‌شود. به عنوان مثال، MuJoCo یک محیط جالب به نام `HumanoidStandup-v2` را ارائه می‌دهد که می‌توانیم از آن برای آموزش عامل خود برای ایستادن، استفاده کنیم. محیط `HumanoidStandup-v2` در شکل ۲.۲۳ نشان داده شده است:

شکل ۲.۲۳: محیط ایستاده انسان نما[۱]

ما می‌توانیم چندین محیط دیگر Mujoco ارائه شده توسط جیم را در اینجا بررسی کنیم:

`https://gym.openai.com/envs/#mujoco`.

رباتیک

جیم، محیط‌های مختلفی را برای انجام وظایف مبتنی بر هدف[۲] برای ربات‌های دستی واکشی و سایه‌ای[۳] فراهم می‌کند. به عنوان مثال، جیم، محیطی به نام `HandManipulateBlock-v0` را فراهم می‌کند، که می‌توان از آن برای آموزش عامل خود برای تغییر جهت جعبه[۴] با استفاده از یک دست رباتیک استفاده کرد. محیط `HandManipulateBlock-v0` در شکل ۲.۲۴ نشان داده شده است:

[۱] The Humanoid Stand Up Environment
[۲] Goal-Based Tasks
[۳] Fetch And Shadow Hand Robots
[۴] Orient a Box

شکل ۲.۲۴: محیط مربوط به عملیات دستی یا دست‌کاری بلوک[۱]

ما می‌توانیم چندین محیط رباتیک ارائه شده توسط جیم را در اینجا بررسی کنیم:

https://gym.openai.com/envs/#robotics.

متن اسباب بازی[۲]

متن اسباب بازی، ساده‌ترین محیط مبتنی بر متن است. قبلاً در ابتدای این فصل با یکی از این محیط‌ها آشنا شدیم که محیط دریاچه یخ‌زده است. ما می‌توانیم سایر محیط‌های متن اسباب بازی جالب ارائه شده توسط جیم را در آدرس زیر بررسی کنیم:

https://gym.openai.com/envs/#toy_text

الگوریتم‌ها

[۱] The Hand Manipulate Block Environment
[۲] Toy Text

آیا به جای استفاده از عامل یادگیری تقویتی خود برای بازی کردن، می‌توانیم از این عامل برای حل برخی از مسائل جالب استفاده کنیم؟ بله! محیط الگوریتمی، چندین مسئله جالب مانند کپی‌کردن یک دنباله معین، انجام جمع و غیره را فراهم می‌کند. ما می‌توانیم از عامل یادگیری تقویتی، برای حل این مسائل با یادگیری نحوه انجام محاسبات استفاده کنیم. به عنوان مثال، جیم، محیطی به نام `ReversedAddition-v0` را فراهم می‌کند که می‌توانیم از آن برای آموزش عامل خود برای جمع کردن اعداد چند رقمی استفاده کنیم. ما می‌توانیم محیط‌های الگوریتمی ارائه شده توسط جیم را در اینجا بررسی کنیم:

`https://gym. openai.com/envs/#algorithmic.`

مجمل و کوته‌واره در باره محیط[۱]

ما با چندین نوع محیط جیم آشنا شدیم. آیا بهتر نیست اطلاعاتی درباره همه محیط‌ها به صورت یکجا داشته باشیم؟ بله! ویکی جیم، شرحی از همه محیط‌ها با شناسه محیط، فضای حالت، فضای اقدام و محدوده پاداش در یک جدول ارائه می‌کند:

`https://github.com/openai/gym/ wiki/Table-of-environments`

همچنین می‌توانیم تمام محیط‌های موجود در جیم را با استفاده از روش `registry.all()` بررسی کنیم.

```
from gym import envs
print(envs.registry.all())
```

کد قبلی، تمام محیط‌های موجود در جیم را چاپ می‌کند.

بنابراین در این فصل با جعبه ابزار جیم و همچنین چندین محیط جالب ارائه شده توسط جیم آشنا شدیم. در فصل‌های آینده، ما یاد خواهیم گرفت که چگونه عامل یادگیری تقویتی خود را در محیط جیم آموزش دهیم تا خط‌مشی بهینه را پیدا کند.

---

[۱] Environment Synopsis

خلاصه

ما این فصل را با درک نحوه راه‌اندازی ماشین خود با نصب آناکوندا و جعبه ابزار جیم آغاز کردیم. ما یاد گرفتیم که چگونه با استفاده از تابع `gym.make()` یک محیط جیم ایجاد کنیم. بعداً نحوه به دست آوردن فضای حالت محیط با استفاده از `env.observation_space` و **فضای اقدام** (**فضای کنش**) محیط با استفاده از `env.action_space` را بررسی کردیم. سپس یاد گرفتیم که چگونه با استفاده از `env.P`، **احتمال انتقال** و **تابع پاداش** محیط را بدست آوریم. در ادامه آن، نحوه ساخت اپیزود با استفاده از محیط جیم را نیز یاد گرفتیم. ما متوجه شدیم که چگونه در هر گام از اپیزود، با استفاده از تابع `env.step()` یک **اقدام** را انتخاب کنیم.

ما روش‌های کنترل کلاسیک را در محیط جیم درک کردیم. در مورد فضای حالت پیوسته محیط‌های کنترل کلاسیک و نحوه ذخیره آنها در یک آرایه، نیز مطالبی آموختیم. ما همچنین یاد گرفتیم که چگونه یک تیرک را با استفاده از یک عامل تصادفی (بختکی) متعادل کنیم. بعداً با محیط‌های جالب بازی آتاری و نحوه نام‌گذاری محیط‌های بازی آتاری در جیم آشنا شدیم و سپس فضای حالت و **فضای اقدام** آنها را بررسی کردیم. همچنین یاد گرفتیم که چگونه بازی عامل را با استفاده از کلاس پوشش، ضبط کنیم و در پایان فصل، محیط‌های دیگری را که توسط جیم ارائه شده بود، کشف کردیم.

در فصل بعد، نحوه یافتن **سیاست بهینه** با استفاده از دو الگوریتم جالب به نام‌های **تکرار ارزش** و **تکرار سیاست** را یاد خواهیم گرفت.

سوالات

بیایید دانش جدید خود را با پاسخ به سؤالات زیر ارزیابی کنیم:

۱. استفاده از جعبه ابزار جیم چیست؟

۲. چگونه در جیم، محیطی را ایجاد کنیم؟

۳. چگونه **فضای اقدام** (**فضای کنش**) محیط جیم را بدست آوریم؟

٤. چگونه محیط جیم را تجسم کنیم؟

٥. چند محیط کنترل کلاسیک ارائه شده توسط جیم را نام ببرید.

٦. چگونه با استفاده از محیط جیم اپیزود تولید کنیم؟

۷. فضای حالت محیط‌های جیم آتاری چگونه است؟

۸. چگونه روند بازی (گیم‌پلی) عامل را ضبط کنیم؟

برای مطالعه بیشتر

برای اطلاعات بیشتر منابع زیر را بررسی کنید:

• برای کسب اطلاعات بیشتر در مورد جیم، به آدرس زیر مراجعه کنید:

http://gym.openai.com/docs/

- همچنین می‌توانیم مخزن جیم[۱] را بررسی کنیم تا بفهمیم محیط‌های جیم چگونه کدگذاری می‌شوند:

https://github.com/openai/gym.

[۱] Gym Repository

فصل سوم

معادله بلمن

و

برنامه‌ریزی پویا

در فصل قبل، آموختیم که در یادگیری تقویتی هدف ما یافتن **سیاست بهینه** است. **خط‌مشی** یا **سیاست بهینه**، سیاستی است که در هر حالت، اقدام صحیح را انتخاب می‌کند تا عامل بتواند حداکثر بازده را به دست آورد و به هدف خود دست یابد. در این فصل، با دو الگوریتم **یادگیری تقویتی کلاسیک** جالب به نام‌های روش‌های **تکرار ارزش**[۱] و **تکرار سیاست**[۲] آشنا می‌شویم که می‌توانیم از آن‌ها برای یافتن خط‌مشی بهینه استفاده کنیم.

قبل از اینکه مستقیماً به روش‌های ***تکرار ارزش*** و ***تکرار سیاست*** بپردازیم، ابتدا با معادله بلمن آشنا می‌شویم. معادله بلمن در یادگیری تقویتی همه جا حاضر است و برای یافتن مقدار بهینه و ***توابع*** Q استفاده می‌شود. ما خواهیم دانست که معادله بلمن چیست و چگونه مقدار بهینه و ***توابع*** Q را پیدا می‌کند.

پس از درک معادله بلمن، با دو روش برنامه‌ریزی پویای جالب به نام ***تکرار ارزش*** و ***تکرار سیاست*** آشنا می‌شویم که از معادله بلمن برای یافتن **خط‌مشی** بهینه استفاده می‌کنند. در پایان فصل، نحوه حل مسئله دریاچه منجمد یا یخ‌زده[۳] را با یافتن **خط‌مشی** بهینه با استفاده از روش‌های ***تکرار ارزش*** و ***تکرار سیاست*** یاد خواهیم گرفت.

در این فصل، با موضوعات زیر آشنا می‌شویم:

- **معادله بلمن**
- **معادله بهینگی بلمن**
- **رابطه بین تابع ارزش و تابع Q**
- **برنامه‌ریزی پویا – روشهای تکرار ارزش و تکرار سیاست**
- **حل مسئله دریاچه یخ‌زده با استفاده از تکرار ارزش و تکرار سیاست**

---

[۱] Value Iteration Method
[۲] Policy Iteration Method
[۳] Frozen Lake

معادله بلمن

معادله بلمن، که به نام ریچارد بلمن نامگذاری شده است، **به ما کمک می‌کند تا فرآیند تصمیم‌گیری مارکوف (MDP) را حل کنیم.** وقتی می‌گوییم **MDP** را حل کنید، منظورمان یافتن **خط‌مشی** بهینه است.

همانطور که در مقدمه فصل بیان شد، معادله بلمن حضور همه جانبه در یادگیری تقویتی دارد و از آن به طور گسترده‌ای برای یافتن مقدار بهینه **تابع ارزش** و **تابع** Q به صورت بازگشت‌وار[۱] استفاده می‌شود. محاسبه مقدار بهینه **تابع ارزش** و **تابع** Q، بسیار مهم است زیرا زمانی که مقدار بهینه **تابع ارزش**، یا **تابع** Q بهینه را داشته باشیم، می‌توانیم از آنها برای استخراج **سیاست** بهینه استفاده کنیم.

در این بخش، می‌آموزیم که معادله بلمن دقیقا چیست و چگونه می‌توان از آن برای یافتن مقدار بهینه و توابع Q استفاده کرد.

معادله بلمن تابع ارزش

معادله بلمن، بیان می‌کند که ارزش یک حالت را می‌توان از طریق مجموع پاداش فوری بعلاوه ارزش تنزیل شده حالات بعدی، به دست آورد. فرض کنید یک **اقدام** a را در حالت s انجام می‌دهیم و به حالت بعدی s' می‌رویم و یک **پاداش** r بدست می‌آوریم، سپس معادله بلمنِ **تابع ارزش** را می‌توان به صورت زیر بیان کرد:

$$V(s) = R(s, a, s') + \gamma V(s')$$

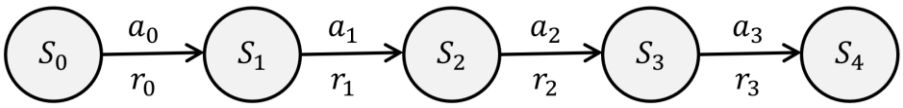
نمادهای مورد استفاده در معادله فوق به این صورت معرفی می‌شوند:

- $R(s, a, s')$ **پاداش فوری** است که با انجام **اقدام** a در حالت s و رسیدن به حالت بعدی s' بدست می‌آید.
- γ بیانگر ضریب تنزیل (فاکتور تخفیف) است.

[۱] Recursively

- $V(s')$ بیانگر **ارزش حالت بعدی** است.

بیایید، با یک مثال، معادله بلمن را درک کنیم. ابتدا ما یک مسیر (خط سیر) τ، با **سیاست** π ایجاد می‌کنیم:



تصویر ۳.۱: خط سیر اقدامات و حالات

فرض کنید، باید **ارزش** حالت $s_۲$ را محاسبه کنیم. با توجه به معادله بلمن، **ارزش** حالت $s_۲$ به صورت زیر محاسبه می‌شود:

$$V(s_۲) = R(s_۲, a_۲, s_۳) + \gamma V(s_۳)$$

در معادله قبل، $R(s_۲, a_۲, s_۳)$ نشان‌دهنده **پاداش فوری** است که هنگام انجام یک اقدام $a_۲$ در حالت $s_۲$ و حرکت به حالت $s_۳$ به دست می‌آوریم. بر اساس این مسیر، می‌توانیم بگوییم که پاداش فوری $R(s_۲, a_۲, s_۳)$، برابر $r_۲$ است. و عبارت $\gamma V(s_۳)$ **ارزش تنزیل شده حالت بعدی** است.

بنابراین، با توجه به معادله بلمن، مقدار حالت $s_۲$ به صورت زیر نوشته می‌شود:

$$V(s_۲) = r_۲ + \gamma V(s_۳)$$

بنابراین، **معادله بلمن تابع ارزش** را می‌توان به صورت زیر بیان کرد:

$$V^{\pi}(s) = R(s, a, s') + \gamma V^{\pi}(s')$$

که در آن، بالانویس π به این معنی است که ما از **خط‌مشی** یا **سیاست** π استفاده می‌کنیم. عبارت سمت راست، یعنی:

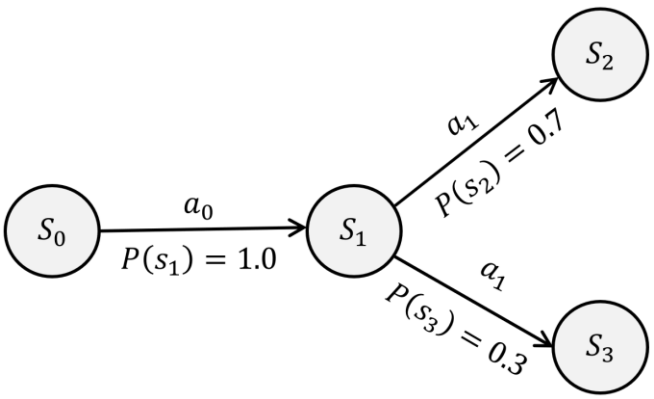
$$R(s, a, s') + \gamma V^{\pi}(s')$$

اغلب **پیشینه یا پشتیبان بلمن**[۱] نامیده می‌شود.

معادله بلمن قبلی فقط زمانی کار می‌کند که یک **محیط قطعی** داشته باشیم. فرض کنید در یک **محیط اتفاقی** باشیم، در آن صورت، وقتی یک اقدام a را در حالت s انجام می‌دهیم، تضمین نمی‌شود که حالت بعدی ما همیشه s' باشد، ممکن است برخی از حالت‌های دیگر نیز باشد. به عنوان مثال، به مسیر در شکل ۳.۲ نگاه کنید.

همانطور که مشاهده می‌کنیم، زمانی که اقدام a_1 را در حالت s_1 انجام می‌دهیم، با **احتمال** ۰.۷، ما به حالت s_2، می‌رسیم و با **احتمال** ۰.۳، به حالت s_3 می‌رویم.

بنابراین، وقتی اقدام a_1 را در حالت s_1 انجام می‌دهیم، ۷۰ درصد احتمال دارد که حالت بعدی s_2 باشد و ۳۰ درصد احتمال دارد که حالت بعدی s_3 باشد. فهمیدیم که معادله بلمن مجموع **پاداش** فوری و **ارزش** تنزیل شده حالت بعدی است. اما زمانی که حالت بعدی ما به دلیل ماهیت **اتفاقی** بودن **محیط** تضمین نمی‌شود، چگونه می‌توانیم معادله بلمن خود را تعریف کنیم؟



شکل ۳.۲: **احتمال گذار** انجام اقدام a_1 در حالت s_1

در این مورد، می‌توانیم معادله بلمن خود را با امید (میانگین وزنی)، یعنی مجموع پشتیبان بلمن را در **احتمال گذار** متناظر حالت بعدی ضرب کنیم:

[۱] Bellman Backup

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

در معادله قبل موارد زیر اعمال می شود:

- $P(s'|s,a)$ بیانگر **احتمال گذار** رسیدن به s' با انجام یک اقدام a در حالت s است.
- $[R(s,a,s') + \gamma V^{\pi}(s')]$ بیانگر **پشتیبان بلمن** هست.
- بیایید با در نظرگرفتن همان مسیری که استفاده کردیم، این معادله را بهتر درک کنیم. همانطور که متوجه شدیم، وقتی یک اقدام a_1 را در حالت s_1 انجام می‌دهیم، با **احتمال** ۰.۷ به حالت s_2 و با **احتمال** ۰.۳ به حالت s_3 می‌رویم. بنابراین، می‌توانیم بنویسیم:

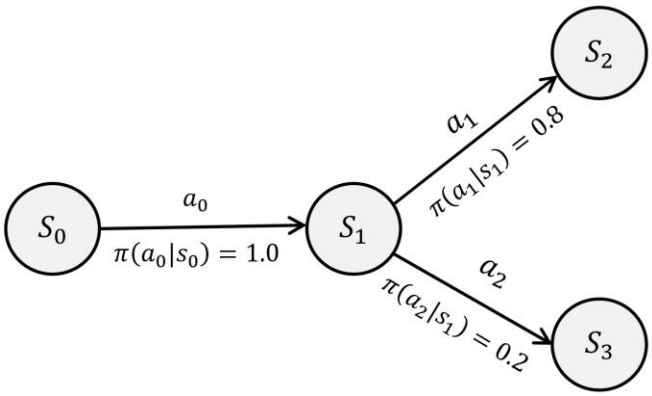
$$V(s_1) = P(s_2|s_1,a_1)[R(s_1,a_1,s_2) + V(s_2)] + P(s_3|s_1,a_1)[R(s_1,a_1,s_3) + V(s_3)]$$

$$V(s_1) = ۰.۷۰[R(s_1,a_1,s_2) + V(s_2)] + ۰.۳۰[R(s_1,a_1,s_3) + V(s_3)]$$

بنابراین، معادله بلمن، **تابع ارزش** شامل ماهیت **اتفاقی** (**پیشامدی**) بودن **محیط** را با استفاده از امید ریاضی (میانگین وزنی) به صورت زیر بیان می‌شود:

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

خب، تاکنون ما با **سیاست** قطعی کار می‌کردیم اما اگر خط‌مشی ما یک **سیاست** **اتفاقی** باشد چه؟ ما آموختیم که با یک خط‌مشی **اتفاقی**، اقدامات را بر اساس **توزیع احتمال** انتخاب می‌کنیم. یعنی به جای انجام همان اقدام در یک حالت، یک اقدام را بر اساس توزیع احتمال در فضای عمل انتخاب می‌کنیم. اجازه دهید این موضوع را با ترسیم یک مسیر متفاوت، که در شکل ۳.۳ نشان داده شده است، درک کنیم. همانطور که می‌بینیم در حالت s_1، با **احتمال** ۰.۸ عمل a_1 را انتخاب می‌کنیم و به حالت s_2 می‌رسیم و با **احتمال** ۰.۲ عمل a_2 را انتخاب می‌کنیم و به حالت s_3 می‌رسیم:



شکل ۳.۳: مسیر با استفاده از خطمشی **اتفاقی**

بنابراین، می‌کنیم، حالت بعدی ما همیشه یکسان نخواهد بود؛ حالات مختلف با احتمال کمی احتمال وقوع دارند. حال، چگونه می‌توانیم معادله بلمن را شامل سیاست **اتفاقی** (**پیشامدی**) تعریف کنیم؟

• ما آموختیم که برای گنجاندن ماهیت **محیط اتفاقی** در معادله بلمن، امید ریاضی (میانگین وزنی) را در نظر می‌گیریم، یعنی یک مجموعی از پشتیبان بلمن در احتمال گذار متناظر حالت بعدی ضرب می‌شود.

• به طور مشابه، برای گنجاندن ماهیت **سیاست اتفاقی** در معادله بلمن، می‌توانیم از انتظار (میانگین وزنی) استفاده کنیم، یعنی مجموع پشتیبان بلمن در احتمال اقدام مربوطه ضرب شود.

بنابراین، معادله نهایی بلمن **تابع ارزش** را می‌توان به صورت زیر نوشت:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s\prime} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')] \qquad (۱)$$

معادله قبلی، همینطور به عنوان **معادله انتظاری بلمن**[۱] تابع ارزش نیز شناخته می‌شود. می‌توانیم معادله فوق را به صورت امیدریاضی بیان کنیم. بیایید تعریف امیدریاضی را به خاطر بیاوریم:

[۱] Bellman Expectation Equation

$$\mathbb{E}_{x\sim p(x)}[f(X)] = \sum_{x} p(x)f(x)$$

در معادله (۱) بالا داریم: $f(x) = R(s,a,s') + \gamma V^{\pi}(s')$ توجه کنید که $P(x) = P(s'|s,a)$ احتمالات مربوط به **محیط** **اتفاقی** و $\pi(a|s)$، احتمالات مربوط به **خط‌مشی** **اتفاقی** را نشان می‌دهند.

بنابراین، می توانیم معادله بلمنِ تابع مقدار را به صورت زیر بنویسیم:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a\sim\pi \\ s'\sim P}}[R(s,a,s'] + \gamma V^{\pi}(s')] \qquad (۲)$$

معادله بلمن تابع Q

حالا بیایید نحوه محاسبه معادله بلمن **تابع ارزش حالت–اقدام** یعنی تابع Q را بیاموزیم. معادله بلمن تابع Q، به جز یک تفاوت کوچک، شباهت زیادی به معادله بلمن تابع ارزش دارد. مشابه معادله بلمن تابع مقدار، معادله بلمن تابع Q بیان می‌کند که ارزش Q یک جفت حالت–اقدام را می‌توان به عنوان مجموع **پاداش فوری**[۱] و **ارزش Q تنزیل شده** زوج حالت–اقدام بعدی به دست آورد.

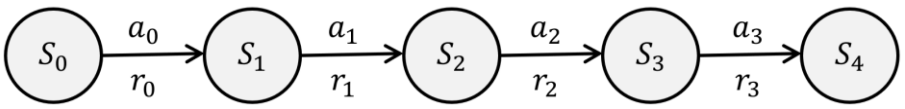
$$Q(s,a) = R(s,a,s') + \gamma Q(s',a')$$

پارامترها در معادله بالا بشرح زیرند:

- $R(s,a,s')$ بیانگر **پاداش فوری** بدست آمده در زمانی است که یک **اقدام** a در حالت s انجام می‌شود و به حالت بعدی s' می‌رسیم.
- γ فاکتور تخفیف است.
- $Q(s',a')$ **ارزش** Q زوج حالت–اقدام بعدی است.

[۱] Immediate Reward

بیایید با یک مثال این موضوع را وارسی کنیم. فرض کنید که با استفاده از **سیاستهایی** که در شکل ۳.۴ نشان داده شده است، یک مسیر تولید می‌کنیم:



شکل ۳.۴: خط سیر تولید شده

فرض کنید، قصد داریم **ارزش** Q یک جفت حالت–اقدام $(S_۲, a_۲)$، را محاسبه کنیم. پس، با توجه به معادله بلمن، می‌توان نوشت:

$$Q(S_۲, a_۲) = R(S_۲, a_۲, S_۳) + \gamma Q(S_۳, a_۳)$$

در معادله بالا، $R(S_۲, a_۲, a_۳)$ نشان‌دهنده پاداش فوری است که در حین انجام یک اقدام $a_۲$ در حالت $S_۲$ و حرکت به حالت $S_۳$ به دست می‌آوریم. از مسیر قبل، می‌توانیم بگوییم که پاداش فوری $R(S_۲, a_۲, S_۳)$، برابر $r_۲$ است. و عبارت $\gamma Q(S_۳, a_۳)$ ، ارزش Q تنزیل شده زوج حالت–اقدام بعدی را نشان می‌دهد. بدین ترتیب:

$$Q(S_۲, a_۲) = r_۲ + \gamma Q(S_۳, a_۳)$$

بنابراین، معادله بلمن برای تابع Q را می‌توان به صورت زیر بیان کرد:

$$Q^{\pi}(s, a) = R(s, a, s') + \gamma Q^{\pi}(s', a')$$

که در آن، بالانویس فرمول، نشان می‌دهد که ما از خطمشی π استفاده می‌کنیم.

در اینجا نیز عبارت سمت راست، یعنی $R(s, a, s') + \gamma Q^{\pi}(s', a')$ **پشتیبان بلمن** است.

مشابه آنچه در معادله بلمن تابع ارزش یاد گرفتیم، **معادله بلمن فوق فقط زمانی کار می‌کند که یک محیط قطعی داشته باشیم** زیرا در **محیط اتفاقی**، حالت بعدی ما همیشه یکسان نخواهد بود و بر اساس **توزیع احتمال** خواهد بود.

فرض کنید یک **محیط اتفاقی** (**پیشامدی**) داریم، در نتیجه وقتی یک اقدام a را در حالت s انجام می‌دهیم، تضمینی نیست که حالت بعدی ما همیشه s' باشد؛ حالت بعدی، می‌تواند با احتمال کمی، برخی از حالات دیگر باشد.

بنابراین، همانطور که در بخش قبل انجام دادیم، می‌توانیم از امید ریاضی (میانگین وزنی)، یعنی مجموع پشتیبان بلمن ضرب در احتمال گذار متناظر آن‌ها در حالت بعدی، استفاده کنیم و معادله بلمن تابع Q را بازنویسی کنیم:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma Q^{\pi}(s',a')]$$

به طور مشابه، وقتی از یک **سیاست اتفاقی** و نه **سیاست قطعی** استفاده می‌کنیم، حالت بعدی ما همیشه یکسان نخواهد بود. حالات مختلف با مقداری احتمال، امکان‌پذیرخواهد بود. بنابراین، برای گنجاندن ماهیت **اتفاقی** (**پیشامدی**) بودن **خط‌مشی**، می‌توانیم معادله بلمن خود را با امیدریاضی (میانگین وزنی) بازنویسی کنیم، یعنی مجموع پشتیبان بلمن در احتمال اقدام مربوطه ضرب می‌شود، درست مانند آنچه در معادله بلمن تابع ارزش انجام دادیم. بنابراین، معادله بلمن تابع Q به صورت زیر است:

$$Q^{\pi}(s,a) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma Q^{\pi}(s',a')]$$

اما صبر کنید! یک تغییر کوچک در معادله بالا وجود دارد. چرا باید عبارت $\sum_a \pi(a|s)$ را در مورد **تابع** Q اضافه کنیم؟ زیرا در **تابع ارزش** $V(s)$، فقط یک حالت s به ما داده می‌شود و یک اقدام a را بر اساس **خط‌مشی** π انتخاب می‌کنیم. بنابراین، ما عبارت $\sum_a \pi(a|s)$ را اضافه کردیم تا ماهیت **اتفاقی** (**شانسی**) بودن **خط‌مشی** را در برگیرد. اما در مورد تابع Q، $Q(s,a)$، هم حالت s و هم اقدام a، به ما داده می‌شود، بنابراین نیازی به اضافه کردن عبارت $\sum_a \pi(a|s)$ در معادله ما نیست، زیرا ما هیچ اقدام aای را بر اساس سیاست π، انتخاب نمی‌کنیم.

با این حال، اگر به معادله بالا نگاه کنید، باید هنگام محاسبه **ارزش** Q، زوج حالت–اقدام بعدی $Q(s',a')$، نیاز داریم که اقدام a' را بر اساس **سیاست** π، انتخاب کنیم، در حالتی که a'، داده نشده است. در نتیجه، ما تنها می‌توانیم

عبارت $\sum_{a'} \pi(a'|s')$ را قبل از ارزش Q زوج حالت–اقدام قرار دهیم. درنتیجه، معادله بلمن نهایی **تابع** Q می‌تواند به این صورت نوشته شود:

$$Q^{\pi}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s')Q^{\pi}(s',a')] \qquad (۳)$$

معادله (۳) به عنوان **معادله انتظاری بلمن** تابع Q نیز شناخته می‌شود. همچنین می‌توانیم معادله (۳) را به شکل امید ریاضی بیان کنیم:

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi} Q^{\pi}(s',a')] \qquad (۴)$$

اکنون که ما فهمیدیم معادله انتظار بلمن چیست، در بخش بعدی، با معادله بهینگی بلمن آشنا خواهیم شد و چگونگی مفید بودن آن برای یافتن مقدار بهینه بلمن و توابع Q را بررسی خواهیم کرد.

معادله بهینگی بلمن

معادله بهینگی بلمن، **ارزش** بهینه بلمن و **توابع** Q را به ما ارائه می‌کند. ابتدا، اجازه دهید تابع **ارزش** بهینه بلمن را بررسی کنیم. ما آموختیم که معادله بلمن **تابع ارزش** به صورت زیر بیان می‌شود:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi \\ s' \sim P}}[R(s,a,s') + \gamma V^{\pi}(s')]$$

در فصل اول، یاد گرفتیم که **تابع ارزش** به **خطمشی** بستگی دارد، یعنی ارزش حالت، بر اساس سیاستی که انتخاب می‌کنیم متفاوت است. با توجه به **سیاستهای** مختلف، **توابع ارزش** متفاوتی، می‌تواند وجود داشته باشد. تابع ارزش بهینه $V^{*}(s)$، وضعیتی است که بیشینه ارزش را در مقایسه با در مقایسه با همه تابع ارزش‌های دیگر دارد. به طور مشابه،

توابع ارزش بلمن متفاوت بسیاری با توجه به **سیاست‌های** مختلف می‌تواند وجود داشته باشد. تابع ارزش بهینه بلمن، تابع با میزان ارزش بهینه است.

خوب، چگونه ما می‌توانیم تابع ارزش بهینه بلمن که ارزش بیشینه دارد را محاسبه کنیم؟

ما می‌توانیم **تابع ارزش** بهینه بلمن را با انتخاب اقدامی که حداکثر ارزش را می‌دهد محاسبه کنیم. اما ما نمی‌دانیم که کدام اقدام حداکثر ارزش را می‌دهد، بنابراین، ارزش حالت را با استفاده از تمام اقدامات ممکن محاسبه می‌کنیم و سپس حداکثر ارزش را به عنوان ارزش حالت انتخاب می کنیم.

یعنی، به جای استفاده از **سیاست** π برای انتخاب اقدام، **ارزش** حالت را با استفاده از تمام اقدامات ممکن محاسبه می‌کنیم و سپس حداکثر ارزش را به عنوان ارزش حالت انتخاب می کنیم. از آنجایی که ما از هیچ سیاستی استفاده نمی‌کنیم، می‌توانیم انتظارات مربوط به **خط‌مشی** π را حذف کنیم، لذا نماد حداکثر را بر روی اقدام اضافه کرده **و تابع ارزش** بهینه بلمن خود را به صورت زیر بیان کنیم:

$$V^*(s) = \max_a \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma V^*(s')] \qquad (۵)$$

این دقیقاً مانند معادله بلمن است، با این تفاوت که در اینجا ما به جای امید ریاضی (میانگین موزون) سیاست، بیشینه‌گیری را روی تمام اقدامات ممکن انجام می‌دهیم، زیرا ما فقط به حداکثر ارزش علاقه‌مندیم. بیایید با یک مثال این را بفهمیم. فرض کنید، ما در یک حالت s هستیم و دو اقدام ممکن در این حالت داریم. اگر اقدامات، ۰ و ۱ باشند. آنگاه $V^*(s)$ ، به صورت زیر محاسبه می‌شود:

$$V^*(s) = \max \begin{pmatrix} \mathbb{E}_{s' \sim P}[R(s,۰,s') + \gamma V^*(s')] \\ \mathbb{E}_{s' \sim P}[R(s,۱,s') + \gamma V^*(s')] \end{pmatrix}$$

همانطور که از معادله بالا مشاهده می‌کنیم، ارزش حالت را با استفاده از تمام اقدامات ممکن (۰ و ۱) محاسبه می‌کنیم و سپس حداکثر ارزش را به عنوان ارزش حالت انتخاب می‌کنیم.

اکنون به تابع Q بهینه بلمن، نگاه می‌کنیم. ما آموختیم که معادله بلمن تابع Q به صورت زیر بیان می‌شود:

$$Q^{\pi}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \mathbb{E}_{a' \sim \pi} Q^{\pi}(s',a')]$$

درست همانطور که با **تابع ارزش** بهینه بلمن یاد گرفتیم، به جای استفاده از **خط‌مشی**، برای انتخاب اقدام a'، در حالت بعدی s'، همه اقدامات ممکن را در حالت s' انتخاب کرده و حداکثر **ارزش** Q را محاسبه می‌کنیم. این موضوع را می‌توان به صورت زیر بیان کرد:

$$Q^{*}(s,a) = \mathbb{E}_{s' \sim P}\left[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a')\right] \qquad (۶)$$

بیایید با یک مثال این را بررسی کنیم. فرض کنید که ما در یک حالت s با یک اقدام a هستیم. ما اقدام a در حالت s را انجام می‌دهیم و به حالت بعدی s'، می‌رسیم. ما نیاز داریم که ارزش Q را برای حالت بعدی s'، محاسبه کنیم. اقدامات زیادی در حالت s' می‌تواند وجود داشته باشد. فرض کنید دو اقدام ۰ و ۱ در حالت s' داریم. سپس می‌توانیم تابع بهینه بلمن Q را به صورت زیر بنویسیم:

$$Q^{*}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \max \begin{pmatrix} Q^{*}(s',۰) \\ Q^{*}(s',۱) \end{pmatrix}]$$

بنابراین، برای خلاصه‌سازی، معادلات بهینگی بلمن تابع ارزش و تابع Q عبارتند از:

$$V^{*}(s) = \max_{a} \mathbb{E}_{s' \sim P}\left[R(s,a,s') + \gamma V^{*}(s')\right]$$
$$Q^{*}(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \max_{a'} Q^{*}(s',a')]$$

همچنین می‌توانیم انتظارات را گسترش دهیم و معادلات بهینگی بلمن قبلی را به صورت زیر بازنویسی کنیم:

$$V^{*}(s) = \max_{a} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{*}(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

ارتباط بین تابع ارزش و تابع Q

بیایید کمی در مسیر بحث خود تغییر بدهیم و تابع ارزش و تابع Q را که در فصل یک (مبانی یادگیری تقویتی) به آن پرداختیم، خلاصه کنیم. ما آموختیم که ارزش یک حالت (**تابع ارزش**) نشان‌دهنده بازگشت (بازده) مورد انتظار است که از آن حالت و با پیگیری یک **خط‌مشی** π، شروع می‌شود:

$$V^{\pi}(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_{.} = s]$$

به طور مشابه، ارزش Q یک زوج حالت–اقدام، (تابع Q) نشان‌دهنده بازده موردانتظار شروع شده از آن زوج حالت–اقدام دنبال‌کننده سیاست π است:

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\tau\sim\pi}[R(\tau)|s_{.} = s, a_{.} = a\,]$$

ما آموختیم که تابع ارزش بهینه، حداکثر ارزش– حالت را می‌دهد:

$$V^*(s) = \max_{\pi} V^{\pi}(s)$$

و تابع Q بهینه، حداکثر ارزش اقدام–حالت (ارزش Q) را می‌دهد:

$$Q^{\pi}(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

آیا می‌توانیم رابطه‌ای بین تابع ارزش بهینه و تابع Q بهینه بدست آوریم؟ می‌دانیم که **تابع ارزش بهینه** بیشترین بازگشت مورد انتظار را وقتی دارد که از حالت s شروع می‌کنیم، و **تابع Q بهینه**، حداکثر بازده مورد انتظار را وقتی دارد که از حالت s شروع به انجام برخی اعمال همچون a اقدام می‌کنیم. بنابراین، می‌توان گفت که تابع ارزش بهینه عملا

حداکثر ارزش Q بهینه در تمام اقدامات ممکن است و می‌توان آن را به صورت زیر بیان کرد (یعنی V را از Q استخراج کنیم):

$$V^*(s) = \max_a Q^*(s,a)$$

خوب، حالا بیایید به معادلات بلمن خود برگردیم. قبل از ادامه، اجازه دهید فقط معادلات بلمن را مرور کنیم:

- **معادله انتظاری بلمن برای تابع ارزش و تابع Q:**

$$V^\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^\pi(s')]$$

$$Q^\pi(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q^\pi(s',a')]$$

- **معادله بهینگی بلمن برای تابع ارزش و تابع Q:**

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)\,[R(s,a,s') + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

ما آموختیم که تابع بهینه بلمن Q، به صورت زیر بیان می شود:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

اگر یک تابع ارزش بهینه $V^*(s)$ داشته باشیم، آنگاه می‌توانیم از آن برای استخراج تابع بهینه بلمن Q قبلی استفاده کنیم، (یعنی می توانیم Q را از V استخراج کنیم):

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

معادله بالا، یکی از مفیدترین دستاوردها یا یافته‌ها در یادگیری تقویتی است و خواهیم دید که چگونه به ما در بخش بعدی برای یافتن خط‌مشی بهینه، کمک خواهد کرد.

بنابراین، به طور خلاصه، یاد گرفتیم که می‌توانیم V را از Q استخراج کنیم:

$$V^*(s) = \max_a Q^*(s,a) \qquad (۷)$$

و Q را از V استخراج کنید:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \qquad (۸)$$

با جایگزینی معادله (۸) در رابطه (۷)، می‌توانیم بنویسیم:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

همانطور که مشاهده می‌کنیم، ما الان تابع ارزش بهینه بلمن را به دست آوردیم. اینک که معادله بلمن و رابطه بین تابع ارزش و تابع Q را فهمیدیم، **می‌توانیم در مورد نحوه استفاده از این معادلات برای یافتن سیاست بهینه**، به بخش بعدی برویم.

برنامه‌ریزی پویا

برنامه‌ریزی پویا[۱] (**DP**) تکنیکی برای حل مسائل پیچیده است. در برنامه‌ریزی پویا، به جای اینکه یک مسئله پیچیده را به طورکلی حل کنیم، مسئله را به مسائل فرعی ساده تقسیم می‌کنیم، سپس برای هر یک از مسائل فرعی، راه‌حل را محاسبه و ذخیره می‌کنیم. اگر مسئله فرعی مشابهی رخ دهد، دوباره آنرا محاسبه نمی‌کنیم. در عوض، ما از راه‌حل محاسبه شدهٔ قبلی استفاده می‌کنیم. بنابراین، برنامه‌ریزی پویا به کاهش شدید زمان محاسبات کمک می‌کند. این تکنیک، کاربردهای زیادی را در زمینه‌های مختلف از جمله علوم کامپیوتر، ریاضیات، بیوانفورماتیک و غیره دارد.

در اینجا با دو روش مهم که از برنامه‌ریزی‌پویا برای یافتن **خط‌مشی** بهینه استفاده می‌کنند، آشنا می‌شویم. این دو روش عبارتند از:

- **تکرار ارزش**
- **تکرار سیاست**

توجه داشته باشید که برنامه‌ریزی پویا، یک روش **مبتنی بر مدل** است. به این معنی که تنها زمانی برای یافتن سیاست بهینه به ما کمک می‌کند که **پویایی مدل** (**احتمال گذار**[۲]) محیط شناخته شده باشد. اگر **پویایی مدل** را نداشته باشیم، نمی‌توانیم روش‌های برنامه‌ریزی پویا را اعمال کنیم.

	**بخشهای بعدی با محاسبات دستی توضیح داده می‌شوند، برای فهم بیشتر، با یک مداد و کاغذ ادامه دهید.**

[۱] Dynamic Programming
[۲] Transition Probability

تکرار ارزش

در روش **تکرار ارزش**، سعی می‌کنیم سیاست بهینه را پیدا کنیم. ما آموختیم، آن **سیاستی** بهینه است که به عامل می‌گوید در هر حالت، اقدام صحیح را انجام دهد. برای یافتن سیاست بهینه، ابتدا تابع ارزش بهینه را محاسبه می‌کنیم و زمانی که به **تابع ارزش بهینه** رسیدیم، می‌توان از آن برای استخراج **سیاست بهینه** استفاده کرد. حال بیایید بررسی کنیم، چگونه می‌توان تابع ارزش بهینه را محاسبه کرد؟ ما می‌توانیم از معادله بهینه بلمن تابع ارزش استفاده کنیم. ما آموختیم که با توجه به معادله بهینگی بلمن، تابع ارزش بهینه را می‌توان به صورت زیر محاسبه کرد:

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \qquad (۹)$$

در بخش رابطه بین تابع ارزش و تابع Q، یاد گرفتیم که با داشتن تابع ارزش، می‌توانیم تابع Q را استخراج کنیم:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')] \qquad (۱۰)$$

با جایگزینی (۱۰) در (۹)، می توانیم بنویسیم:

$$V^*(s) = \max_a Q^*(s,a)$$

بنابراین، ما می‌توانیم تابع ارزش بهینه را فقط با درنظرگرفتن حداکثر مقدار تابع Q بهینه، محاسبه کنیم. بنابراین، برای محاسبه ارزش یک حالت، ارزش Q را برای همه زوج‌های حالت–اقدام محاسبه می‌کنیم. سپس حداکثر ارزش Q را به عنوان ارزش حالت انتخاب می‌کنیم.

بیایید با یک مثال این موضوع را درک کنیم. فرض کنید ما دو حالت s_0 و s_1 داریم و در این حالت‌ها دو اقدام ممکن داریم که این اقدامات را ۰ و ۱ درنظر می‌گیریم. ابتدا ارزش Q را برای همه زوج‌های ممکن حالت–اقدام محاسبه می‌کنیم. جدول ۳.۱ ارزش‌های Q را برای همه زوج‌های ممکن حالت–اقدام نشان می‌دهد:

حالت	اقدام	ارزش
s_0	۰	۲.۷
s_0	۱	۳
s_1	۰	۴
s_1	۱	۲

جدول ۳.۱: ارزش‌های Q تمام زوج‌های ممکن حالت–اقدام

سپس در هر حالت، حداکثر ارزش Q را به عنوان ارزش بهینه یک حالت، انتخاب می‌کنیم. بنابراین، ارزش حالت s_0 برابر ۳ و ارزش حالت s_1 برابر ۴ است. ارزش بهینه حالت (تابع ارزش) در جدول ۳.۲ نشان داده شده است:

حالت	ارزش
s_0	۳
s_1	۴

جدول ۳.۲: ارزش‌های حالت بهینه

هنگامی که **تابع ارزش بهینه** را بدست آوردیم، می‌توانیم از آن برای استخراج **خط‌مشی بهینه** استفاده کنیم.

اکنون که درک اولیه‌ای از نحوه یافتن تابع ارزش بهینه با استفاده از روش تکرار ارزش، پیدا کردیم، در بخش بعدی به جزئیات خواهیم پرداخت و نحوه دقیق عملکرد روش تکرار ارزش و یافتن خط‌مشی بهینه آن از تابع ارزش بهینه را خواهیم آموخت.

الگوریتم تکرار ارزش

الگوریتم ***تکرار ارزش*** به صورت زیر است:

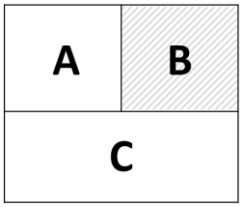
۱. **تابع ارزش بهینه** را با بیشینه‌گیری روی تابع Q محاسبه کنید، یعنی $V^*(s) = \max_a Q^*(s,a)$

۲. **سیاست بهینه** را از **تابع ارزش بهینه** محاسبه‌شده استخراج کنید.

بیایید وارد جزئیات شویم و نحوه عملکرد دو مرحله فوق را دقیقاً یاد بگیریم. برای درک بهتر، اجازه دهید روش **تکرار ارزش** را به صورت دستی انجام دهیم. محیط جهان مشبک کوچک[۱] در شکل ۳.۵ را در نظر بگیرید. فرض کنید در حالت **A** هستیم و هدفمان این است که بدون بازدید از حالت سایه‌دار **B** به حالت **C** برسیم و درنظر بگیریم که دو اقدام داریم:

- عدد ۰ برای چپ یا *راست* و عدد ۱ *برای بالا* یا *پایین*

آیا می‌توانید حالا بگویید که در اینجا سیاست بهینه چیست؟ خطمشی بهینه در اینجا همان است که به ما می‌گوید اقدام ۱ را در حالت **A** انجام دهیم تا بتوانیم بدون بازدید از **B** به **C** برسیم. حال، خواهیم دید که چگونه با استفاده از تکرار ارزش، این خطمشی بهینه را پیدا کنیم.



شکل ۳.۵: محیط دنیای مشبک

جدول ۳.۳ پویایی مدل حالت **A** را نشان می‌دهد:

حالت (s)	اقدام (a)	حالت بعدی (s')	احتمال گذار $P(s'\|s,a) or\ P^a_{ss'}$	تابع پاداش $R(s,a,s') or\ R^a_{ss'}$
A	۰	A	۰.۱	۰
A	۰	B	۰.۸	-۱
A	۰	C	۰.۱	۱

[۱] Small Grid World Environment

A	۱	A	۰.۱	۰
A	۱	B	۰.۰	−۱
A	۱	C	۰.۹	۰

جدول ۳.۳: پویایی‌های مدل حالت **A**

مرحله ۱ – محاسبه تابع ارزش بهینه

ما می‌توانیم **تابع ارزش بهینه** را با بیشینه‌گیری بر روی **تابع** Q محاسبه کنیم.

$$V^*(s) = \max_a Q^*(s,a)$$

یعنی **ارزش** Q، را برای همه زوج‌های حالت–اقدام محاسبه می‌کنیم و سپس حداکثر **ارزش** Q را به عنوان ارزش یک حالت انتخاب می‌کنیم.

ارزش Q، برای حالت s و اقدام a را می‌توان به صورت زیر محاسبه کرد:

$$Q(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$$

برای سادگی علامت‌گذاری، می‌توانیم $P(s'|s,a)$ را با $P^a_{ss'}$ و $R(s,a,s')$ را با $R^a_{ss'}$ نشان دهیم و معادله قبلی را به این صورت بازنویسی کنیم:

$$Q(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')] \qquad (۱۱)$$

بنابراین، با استفاده از معادله قبلی، می‌توانیم تابع Q را محاسبه کنیم. اگر به معادله نگاه کنید، برای محاسبه تابع Q، به احتمال گذار $P_{ss'}^{a}$، تابع پاداش $R_{ss'}^{a}$، و ارزش حالت بعدی $V(s')$ نیاز داریم. پویایی‌های مدل، احتمال انتقال $P_{ss'}^{a}$ و تابع پاداش $R_{ss'}^{a}$ را در اختیار ما قرار می‌دهد. اما در مورد ارزش حالت بعدی $V(s')$ چطور؟ ما ارزش هیچ حالتی را هنوز نمی‌دانیم. بنابراین، ما تابع ارزش (ارزش‌های حالت) را همانطور که در جدول ۳.۴ نشان داده شده، با مقادیر تصادفی (دلبخواه) یا صفر، مقداردهی اولیه می‌کنیم[۱] و تابع Q را محاسبه می‌کنیم.

حالت	ارزش
A	۰
B	۰
C	۰

جدول ۳.۴: جدول ارزش اولیه

تکرار یک:

بیایید ارزش Q حالت **A** را محاسبه کنیم. ما در حالت **A** دو اقدام داریم که ۰ و ۱ هستند. بنابراین، ابتدا ***ارزش*** Q را برای حالت **A** و اقدام ۰ محاسبه می‌کنیم (توجه داشته باشید که از ضریب تخفیف $\gamma = ۱$ در این بخش استفاده می‌کنیم.):

$$\begin{aligned} Q(A, ۰) &= P_{AA}^{۰}[R_{AA}^{۰} + \gamma V(A)] + P_{AB}^{۰}[R_{AB}^{۰} + \gamma V(B)] + P_{AC}^{۰}[R_{AC}^{۰} + \gamma V(C)] \\ &= ۰.۱(۰ + ۰) + ۰.۸(-۱ + ۰) + ۰.۱(۱ + ۰) \\ &= -۰.۷ \end{aligned}$$

حال، بیایید ارزش Q را برای حالت **A** و اقدام ۱ محاسبه کنیم:

$$Q(A, ۱) = P_{AA}^{۱}[R_{AA}^{۱} + \gamma V(A)] + P_{AB}^{۱}[R_{AB}^{۱} + \gamma V(B)] + P_{AC}^{۱}[R_{AC}^{۱} + \gamma V(C)]$$

[۱] **توجه:** اگر مقداردهی اولیه نکنیم آنگاه نمی‌توانیم حل مسئله را آغاز کرده و ادامه دهیم. به این تکنیک Bootstrapping می‌گویند که ما معادلهایی همچون خود–راه‌اندازی، خود–برپایی، خود–آیی، خود–بنایی، خود–ابتنایی یا خود–آوری برای ان پیشنهاد می‌دهیم در فصل پنجم، در خصوص این مفهوم بیشتر خواهیم گفت.

$$= ۰.۱(۰ + ۰) + ۰.۰(-۱ + ۰) + ۰.۹(۱ + ۰)$$
$$= ۰.۹$$

پس از محاسبه **ارزش‌های** Q، برای هر دو اقدام در حالت **A**، می‌توانیم جدول Q را همانطور که در جدول ۳.۵ نشان داده شده است، به روز کنیم:

حالت	اقدام	ارزش
A	۰	-۰.۷
A	۱	۰.۹
B	۰	
B	۱	
C	۰	
C	۱	

جدول ۳.۵: جدول Q

ما آموختیم که **ارزش بهینه** یک حالت فقط بیشینه **تابع** Q است. این مقدار برابر است با $V^*(s) = \max_a Q^*(s,a)$ با مشاهده جدول ۳.۵، می‌توانیم بگوییم که ارزش حالت **A**، یعنی $V(A)$، برابر است با $Q(A,۱)$، به این خاطر که $Q(A,۱)$ ارزش بالاتری از $Q(A,۰)$ دارد. لذا $V(A) = ۰.۹$. ما می‌توانیم، همانطورکه در جدول ۳.۶ نشان داده شده است، ارزش حالت **A** را در جدول ارزش خود به روزرسانی می‌کنیم.

حالت	ارزش
A	۰.۹
B	۰
C	۰

جدول ۳.۶: جدول **ارزش** به روز شده

به همین صورت، برای محاسبه ارزش حالت **B**، یعنی $V(B)$، ما ابتدا ارزش Q را برای $Q(B,0)$ و $Q(B,1)$ محاسبه کرده و سپس بیشترین ارزش Q را به عنوان ارزش حالت **B**، انتخاب می‌کنیم. و به همین ترتیب، برای محاسبه ارزش سایر حالت‌ها، ارزش Q را برای همه زوج‌های حالت–اقدام، محاسبه می‌کنیم و حداکثر ارزش Q را به عنوان ارزش یک حالت انتخاب می‌کنیم. پس از محاسبه ارزش همه حالت‌ها، جدول ارزش به روز شده ما ممکن است شبیه جدول ۳.۷ باشد. این نتیجه تکرار اول است:

حالت	ارزش
A	۰.۹
B	−۰.۲
C	۰.۵

جدول ۳.۷: جدول **ارزش** از تکرار ۱

با این حال، تابع ارزش (جدول ارزش) نشان داده شده در جدول ۳.۷ که در نتیجه اولین تکرار به دست آمده است، تابعی بهینه نیست. اما چرا؟ ما آموختیم که تابع ارزش بهینه، بیشینه تابع Q بهینه است. یعنی:

$$V^*(s) = \max_a Q^*(s,a)$$

بنابراین، برای یافتن تابع ارزش بهینه، به تابع Q بهینه نیاز داریم. اما تابع Q، ممکن است، در تکرار اول بهینه نباشد، به این خاطر که ما تابع Q را بر اساس مقادیر حالت اولیه تصادفی (دلبخواه)، محاسبه کردیم.

همانطور که در ادامه نمایش داده شده است، زمانی که ما محاسبه تابع Q را شروع کردیم، از مقادیر حالت اولیه تصادفی (بختکی) استفاده کردیم.

$$V^*(s) = \max_a Q^*(s,a)$$
$$\sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V(s')\right]$$
ارزشهای تصادفی

بنابراین، کاری که می‌توانیم انجام دهیم این است که در تکرار بعدی، در حالی که تابع Q را محاسبه می‌کنیم، می‌توانیم از ارزش‌های حالت به‌روز شده به دست آمده در نتیجه اولین تکرار استفاده کنیم.

یعنی در تکرار دوم، برای محاسبه تابع ارزش، ارزش Q همه زوج‌های حالت–اقدام را محاسبه کرده و حداکثر مقدار Q را به عنوان ارزش یک حالت انتخاب می‌کنیم. برای محاسبه مقدار Q، باید ارزش‌های حالت را بدانیم، در اولین تکرار، از ارزش‌های حالت اولیه تصادفی (بختکی) استفاده کردیم. اما در تکرار دوم، از مقادیر حالت به روز شده (جدول ارزش‌ها) به دست آمده از تکرار اول، استفاده می‌کنیم که در زیر نشان داده شده است[۱]:

$$V^*(s) = \max_a Q^*(s,a)$$
$$\sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V(s')\right]$$
از ارزشهای حالت در تکرار اول استفاده میکنیم

تکرار دوم:

بیایید ارزش Q حالت **A** را محاسبه کنیم. به یاد داشته باشید که هنگام محاسبه ارزش Q، از ارزش‌های حالت به روز شده از تکرار قبلی استفاده می‌کنیم.

[۱] وقتی همه خانه‌های جدول مشبک، داری مقدار ارزش اولیه و یا به‌روز شده باشند، نیازی به استفاده مجدد از تکنیک خود–آغازگری یا خود–ابتنائی نیست.

ابتدا ارزش Q حالت **A** و اقدام ۰ را محاسبه می‌کنیم:

$$\begin{aligned} Q(A,۰) &= P_{AA}^{۰}[R_{AA}^{۰} + \gamma V(A)] + P_{AB}^{۰}[R_{AB}^{۰} + \gamma V(B)] + P_{AC}^{۰}[R_{AC}^{۰} + \gamma V(C)] \\ &= ۰.۱(۰ + ۰.۹) + ۰.۸(-۱ - ۰.۲) + ۰.۱(۱ + ۰.۵) \\ &= -۰.۷۲ \end{aligned}$$

حال، بیایید ارزش Q را برای حالت **A** و اقدام ۱ محاسبه کنیم:

$$\begin{aligned} Q(A,۱) &= P_{AA}^{۱}[R_{AA}^{۱} + \gamma V(A)] + P_{AB}^{۱}[R_{AB}^{۱} + \gamma V(B)] + P_{AC}^{۱}[R_{AC}^{۱} + \gamma V(C)] \\ &= ۰.۱(۰ + ۰.۹) + ۰.۰(-۱ - ۰.۲) + ۰.۹(۱ + ۰.۵) \\ &= ۱.۴۴ \end{aligned}$$

همانطور که مشاهده می‌کنیم، از آنجایی که ارزش Q اقدام ۱ در حالت **A** بالاتر از اقدام ۰ است، ارزش حالت **A** به ۱.۴۴ تبدیل می‌شود. به طور مشابه، ما ارزش را برای همه حالت‌ها محاسبه می‌کنیم و جدول ارزش را به روز می‌کنیم.

جدول ۳.۸ جدول ارزش به روز شده را نشان می‌دهد:

حالت	ارزش
A	۱.۴۴
B	-۰.۵
C	۱.۰

جدول ۳.۸: جدول **ارزش** از تکرار ۲

تکرار ۳:

همان مراحل تکرار قبل را مجددا تکرار کرده و ارزش همه حالت‌ها را با انتخاب بیشینه ارزش Q، محاسبه می‌کنیم. به یاد بیاورید که هنگام محاسبه ارزش Q، از ارزش‌های حالت به روز شده (جدول ارزش‌ها) به دست آمده از تکرار قبلی استفاده می‌کنیم. بنابراین، ما از ارزش‌های حالت به روز شده از تکرار ۲ برای محاسبه ارزش Q استفاده می‌کنیم.

جدول ۳.۹، ارزش‌های حالت به روز شده به دست آمده در نتیجه تکرار سوم را نشان می‌دهد:

حالت	ارزش
A	۱.۹۴
B	−۰.۷
C	۱.۳

جدول ۳.۹: جدول **ارزش** از تکرار ۳

لذا، این مراحل را برای چندین بار تکرار کرده تا زمانی که تابع ارزش بهینه را پیدا کنیم. **اما چگونه می‌توانیم بفهمیم که آیا تابع ارزش بهینه را پیدا کرده‌ایم یا خیر؟** وقتی تابع ارزش (جدول ارزش) با تکرارها تغییر نمی‌کند یا زمانی که با کسری بسیار کوچک تغییر می‌کند، می‌توان گفت که به همگرایی رسیده‌ایم، یعنی یک تابع مقدار بهینه پیدا کرده‌ایم.

خوب، چگونه می‌توانیم بفهمیم که آیا جدول ارزش نسبت به تکرار قبلی تغییر می‌کند یا نه؟ ما می‌توانیم تفاوت بین جدول ارزش به دست آمده از تکرار قبلی و جدول ارزش به دست آمده از تکرار فعلی را محاسبه کنیم. اگر تفاوت بسیار کوچک باشد (مثلاً، تفاوت کمتر از یک عدد آستانه[۱] بسیار کوچک باشد) می‌توان گفت که به همگرایی رسیده‌ایم زیرا تغییر زیادی در تابع مقدار وجود ندارد. برای مثال، فرض کنید جدول ۳.۱۰، جدول ارزش به دست آمده، به عنوان نتیجه **تکرار چهار** را نشان می‌دهد:

حالت	ارزش
A	۱.۹۴
B	−۰.۷
C	۱.۳

جدول ۳.۱۰: جدول **ارزش** از تکرار چهار

همانطور که می‌بینیم، تفاوت بین جدول ارزشهای به دست آمده در نتیجه تکرار ۴ و تکرار ۳ بسیار ناچیز است. بنابراین، می‌توان گفت که به همگرایی رسیده‌ایم و جدول ارزشهای به‌دست‌آمده در نتیجه تکرار ۴ را به عنوان تابع ارزش بهینه

[۱] Threshold Number

خود در نظر می‌گیریم. لطفا توجه داشته باشید که مثال بالا فقط برای درک بهتر نسبت به این موضوع است؛ در عمل، ما نمی‌توانیم فقط در چهار تکرار به همگرایی برسیم و معمولا تکرارهای زیادی برای این اتفاق نیاز است.

اکنون که تابع ارزش بهینه را پیدا کردیم، در مرحله بعد از این **تابع ارزش بهینه** برای استخراج یک **خط‌مشی بهینه** استفاده می‌کنیم.

مرحله ۲ – استخراج سیاست بهینه را از تابع ارزش بهینه بدست آمده از مرحله ۱

در نتیجه مرحله ۱، تابع ارزش بهینه را به دست آوردیم:

حالت	ارزش
A	۱.۹۵
B	−۰.۷۲
C	۱.۳

جدول ۳.۱۱: جدول **ارزش** بهینه (**تابع ارزش**)

حال چگونه می‌توانیم **سیاست** بهینه را از **تابع ارزش بهینه** بدست آمده استخراج کنیم؟

ما معمولاً از **تابع** Q برای محاسبه **سیاست** استفاده می‌کنیم. می‌دانیم که **تابع** Q، **ارزش** Q را برای هر زوج حالت–اقدام می‌دهد. هنگامی که ارزش‌های Q را برای همه زوج‌های حالت–اقدام محاسبه کردیم، با انتخاب اقدامی که حداکثر ارزش Q را در هر حالت دارد، سیاست را استخراج می‌کنیم. به عنوان مثال، جدول Q را در جدول ۳.۱۲ در نظر بگیرید. ارزش‌های Q را برای همه زوج‌های حالت–اقدام نشان می‌دهد. اکنون می‌توانیم با انتخاب اقدام ۱ در حالت s_0 و اقدام ۰ در حالت s_1، سیاست را از تابع Q (جدول Q) استخراج کنیم زیرا حداکثر ارزش Q را دارند.

حالت	اقدام	ارزش Q
$\mathbf{s_0}$	۰	۱
$\mathbf{s_0}$	۱	۴
$\mathbf{s_1}$	۰	۷
$\mathbf{s_1}$	۱	۲

جدول ۳.۱۲: جدول Q

خوب، اکنون تابع Q را با استفاده از **تابع ارزش** بهینه بدست آمده از مرحله ۱ محاسبه می‌کنیم. هنگامی که تابع Q را داریم، با انتخاب اقدامی که حداکثر ارزش Q را در هر حالت دارد، خط‌مشی را استخراج می‌کنیم. از آنجایی که ما تابع Q را با استفاده از **تابع ارزش** بهینه محاسبه می‌کنیم، **خط‌مشی** استخراج شده از **تابع** Q خط‌مشی بهینه خواهد بود.

ما آموختیم که **تابع** Q را می‌توان به صورت زیر محاسبه کرد:

$$Q(s,a) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V(s')]$$

حال، هنگام محاسبه **ارزشهای** Q، از تابع ارزش بهینه‌ای که از مرحله یک به دست آوردیم استفاده می‌کنیم. پس از محاسبه **تابع** Q، می‌توانیم با انتخاب اقدامی که **حداکثر ارزش** $\boldsymbol{Q}$ را دارد، **سیاست بهینه** را استخراج کنیم:

$$\pi^* = \arg\max_a Q(s,a)$$

برای مثال، بیایید ارزش Q را برای تمام اقدامات در حالت **A** با استفاده از تابع ارزش بهینه، محاسبه کنیم. ارزش Q، برای اقدام ۰ در حالت **A** به این صورت محاسبه می‌شود:

$$\begin{aligned} Q(A,۰) &= P_{AA}[R_{AA} + \gamma V(A)] + P_{AB}[R_{AB} + \gamma V(B)] + P_{AC}[R_{AC} + \gamma V(C)] \\ &= ۰.۱(۰ + ۱.۹۵) + ۰.۸(-۱ - ۰.۷۲) + ۰.۱(۱ + ۱.۳) \\ &= -۰.۹۵۱ \end{aligned}$$

ارزش Q را برای حالت **A** و اقدام ۱، به این صورت محاسبه می‌کنیم:

$$\begin{aligned} Q(A,۱) &= P_{AA}^{۱}[R_{AA}^{۱} + \gamma V(A)] + P_{AB}^{۱}[R_{AB}^{۱} + \gamma V(B)] + P_{AC}^{۱}[R_{AC}^{۱} + \gamma V(C)] \\ &= ۰.۱(۰ + ۱.۹۵) + ۰.۰(-۱ - ۰.۷۲) + ۰.۹(۱ + ۱.۳) \\ &= ۲.۲۶ \end{aligned}$$

از آنجایی که $Q(A,۱)$، بالاتر از $Q(A,۰)$ است، **سیاست بهینه** ما اقدام ۱ را به عنوان اقدام بهینه در حالت **A** انتخاب می‌کند. جدول ۳.۱۳، جدول Q را پس از محاسبه **ارزش‌های** Q برای همه زوج‌های حالت–اقدام، با استفاده از **تابع ارزش بهینه** نشان می‌دهد:

حالت	اقدام	ارزش Q
A	۰	-۰.۹۵
A	۱	۲.۲۶
B	۰	-۰.۵
B	۱	۰.۵
C	۰	-۱.۱
C	۱	۱.۴

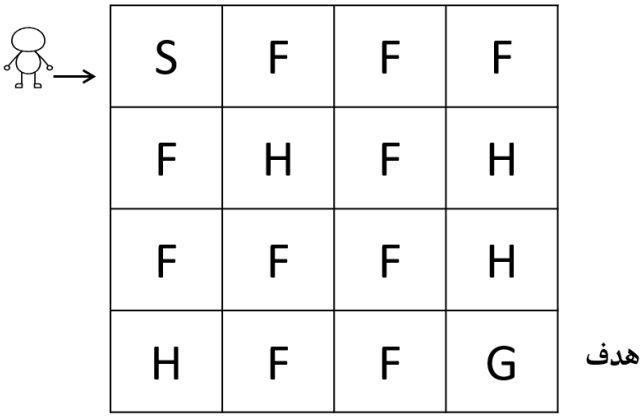
جدول ۳.۱۳: جدول Q

از این جدول Q، **اقدامی** را در هر حالتی انتخاب می‌کنیم که حداکثر **ارزش** را به عنوان **سیاست** بهینه دارد. بنابراین، سیاست بهینه ما **اقدام** ۱ را در حالت **A**، **اقدام** ۱ را در حالت **B** و **اقدام** ۱ را در حالت **C** انتخاب می‌کند. بنابراین، طبق سیاست بهینه ما، اگر اقدام ۱ را در حالت **A** انجام دهیم، می‌توانیم بدون بازدید از حالت **B** به حالت **C** برسیم.

در این بخش، نحوه محاسبه **سیاست بهینه** را با استفاده از **روش تکرار ارزش** یاد گرفتیم. در قسمت بعدی نحوه پیاده‌سازی روش تکرار ارزش برای محاسبه سیاست بهینه در محیط دریاچه یخ زده با استفاده از جعبه ابزار جیم را خواهیم آموخت.

حل مسئله دریاچه یخ زده با روش تکرار ارزش

در فصل قبل، با محیط دریاچه منجمد آشنا شدیم. محیط دریاچه منجمد در شکل ۳.۶ نشان داده شده است.

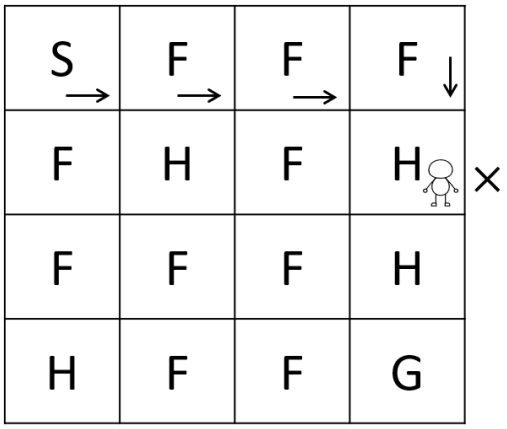


شکل ۳.۶: محیط دریاچه یخزده

اجازه دهید، کمی محیط دریاچه یخزده را مرور کنیم. در محیط دریاچه منجمد نشان داده شده در شکل ۳.۶، موارد زیر اعمال می‌شود:

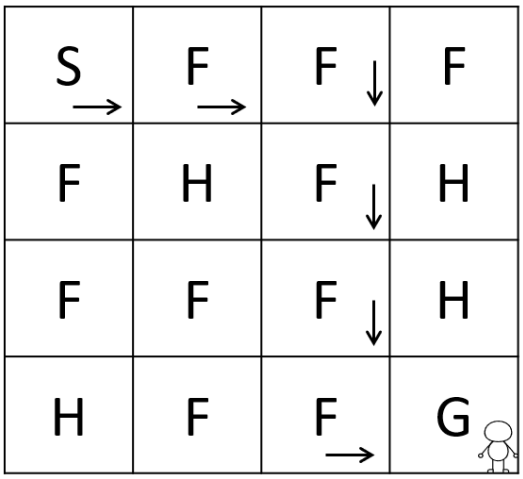
- **S** دلالت بر حالت شروع دارد.
- **H** بر حالات حفره دلالت دارد.
- **F** به حالت‌های منجمد اشاره دارد.
- **G** بر حالت هدف دلالت دارد.

بخاطر دارید که در محیط دریاچه یخزده، هدف ما رسیدن به حالت هدف **G** از حالت شروع **S** بدون بازدید از حالات حفره **H** است. یعنی در حین تلاش برای رسیدن به حالت هدف **G** از حالت شروع **S**، اگر عامل حالت‌های **H** را مشاهده می‌کند، سپس داخل سوراخ می‌افتد و می‌میرد همانطور که شکل ۳.۷ نشان می دهد.



شکل ۳.۷: عامل ما به حفره سقوط می‌کند.

بنابراین، می‌خواهیم عامل از حالت‌های حفره **H** برای رسیدن به حالت هدف **G** اجتناب کند، همانطور که در ۳.۸ نشان داده شده است.



شکل ۳.۸: عامل به «حالت هدف» می‌رسد.

چگونه می‌توانیم به این هدف برسیم؟ یعنی چگونه بدون بازدید از **H** بتوانیم از حالت **S** به حالت **G** برسیم؟ ما آموختیم که خط‌مشی بهینه به عامل می‌گوید که در هر حالت، اقدام صحیح را انجام دهد. بنابراین، اگر خط‌مشی بهینه را پیدا کنیم، می‌توانیم از حالت **S** بدون بازدید از حالت **H** به حالت **G** برسیم. بسیارخوب، چگونه می‌توانیم خط‌مشی بهینه را پیدا کنیم؟ ما می‌توانیم از روش تکرار ارزش که به تازگی یاد گرفتیم برای یافتن خط‌مشی بهینه استفاده کنیم.

به یاد داشته باشید که در جعبه ابزار **جیم**، همه حالت‌های ما (یعنی از حالت **S** تا حالت **G**) با اعدادی از ۰ تا ۱۶ کدگذاری شده و هر چهار اقدام ما یعنی: اقدام به سمتهای چپ، *پایین*، *بالا*، راست، با اعدادی از ۰ تا ۳ کدگذاری می‌شوند.

در این بخش، نحوه یافتن خطمشی بهینه با استفاده از روش تکرار ارزش را یاد خواهیم گرفت، که در نتیجه آن، عامل می‌تواند بدون بازدید از حالت **H** بتواند از حالت **S** به حالت **G** برسد. ابتدا کتابخانه های لازم را وارد می‌کنیم:

```
import gym
import numpy as np
```

حالا بیایید محیط دریاچه یخزده را با استفاده از جیم ایجاد کنیم:

```
env = gym.make('FrozenLake-v0')
```

بیایید با استفاده از تابع رندر به محیط دریاچه منجمد نگاه کنیم:

```
env.render()
```

با وارد کردن دستور بالا، تصویر زیر نمایش داده می‌شود:

شکل ۳.۹: محیط دریاچه یخزده جیم

همانطور که متوجه شدیم عامل ما در حالت **S** است و باید بدون بازدید از حالت‌های **H** به حالت **G** برسد. بنابراین، بیایید یاد بگیریم که چگونه سیاست بهینه را با استفاده از روش تکرار ارزش محاسبه کنیم.

در روش **تکرار ارزش**، دو مرحله را انجام می‌دهیم:

۱. **تابع ارزش بهینه** را با بیشینه‌گیری روی **تابع** Q محاسبه کنید، یعنی $V^*(s) = \max_a Q^*(s,a)$

۲. **سیاست بهینه** را از تابع ارزش بهینه محاسبه شده استخراج کنید.

ابتدا بیایید نحوه محاسبه تابع ارزش بهینه را بیاموزیم و سپس خواهیم دید که چگونه خط‌مشی بهینه را از تابع ارزش بهینه محاسبه شده، استخراج کنیم.

محاسبه تابع ارزش بهینه

برای محاسبه تابع ارزش بهینه به صورت تکراری، یعنی از طریق بیشینه‌گیری بر روی تابع Q، که در آن: $V^*(s) = \max_a Q^*(s,a)$ است، ما تابعی را تحت نام `value_iteration` تعریف می‌کنیم. برای درک بهتر، بیایید از نزدیک به هر خط تابع نگاه کنیم، و سپس ما به کل تابع در پایان نگاه خواهیم کرد، تا شفافیت بیشتری بدست دهد.

تابع `value_iteration` را به گونه‌ای تعریف کنید، محیط را به عنوان پارامتر درنظر بگیرد:

```
def value_iteration(env):
```

تعداد تکرارها را تنظیم کنید:

```
    num_iterations = 1000
```

عدد آستانه را برای بررسی همگرایی تابع ارزش تنظیم کنید:

```
    threshold = 1e-20
```

همچنین فاکتور تخفیف γ را روی ۱ قرار می‌دهیم:

```
    gamma = 1.0
```

اکنون، با صفر کردن ارزش همه حالت‌ها، جدول ارزش را مقداردهی اولیه می‌کنیم:

```
    value_table = np.zeros(env.observation_space.n)
```

برای هر تکرار:

```
    for i in range(num_iterations):
```

جدول ارزش را به روز کنید، یعنی یاد گرفتیم که در هر تکرار، از جدول ارزش به روز شده (ارزش‌های حالت) از تکرار قبلی استفاده می‌کنیم:

```
        updated_value_table = np.copy(value_table)
```

اکنون، تابع ارزش (ارزش حالت) را با بیشینه‌گیری از ارزش Q محاسبه می‌کنیم:

$$V^*(s) = \max_a Q^*(s,a)$$

که در آن:

$$Q(s,a) = \sum_{s'} P^a_{SS'}\left[R^a_{SS'} + \gamma V(s')\right]$$

بنابراین، برای هر حالت، ارزشهای Q تمام اقدامات موجود در حالت را محاسبه کرده و سپس ارزش حالت را به عنوان حالتی که حداکثر ارزش Q را دارد به روز می‌کنیم:

```
        for s in range(env.observation_space.n):
```

ارزش Q تمام اقدامات را محاسبه کنید،

$$Q(s,a) = \sum_{s'} P^a_{SS'}\left[R^a_{SS'} + \gamma V(s')\right]$$

```
            Q_values = [sum([prob*(r + gamma * updated_value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                  for a in range(env.action_space.n)]
```

ارزش حالت را به عنوان حداکثر ارزش Q به روزرسانی کنید، $V^*(s) = \max_a Q^*(s,a)$:

```
            value_table[s] = max(Q_values)
```

پس از محاسبه جدول ارزش، یعنی ارزش همه حالت‌ها، بررسی می‌کنیم که آیا تفاوت بین جدول ارزشهای به دست آمده در تکرار فعلی و تکرار قبلی، کمتر یا مساوی یک مقدار آستانه است یا خیر. اگر اختلاف کمتر از آستانه باشد، حلقه را می‌شکنیم و جدول ارزش را به عنوان تابع ارزش بهینه خود برمی‌گردانیم، همانطور که کد زیر نشان می‌دهد:

```
        if (np.sum(np.fabs(updated_value_table - value_table)) <=
threshold):
            break

    return value_table
```

قطعه کامل[۱] زیر از تابع `value_iteration` نشان داده شده تا وضوح بیشتری بدست دهد:

[۱] Complete Snippet

```
def value_iteration(env):

    num_iterations = 1000
    threshold = 1e-20
    gamma = 1.0
```

```
    value_table = np.zeros(env.observation_space.n)

    for i in range(num_iterations):

        updated_value_table = np.copy(value_table)

        for s in range(env.observation_space.n):

            Q_values = [sum([prob*(r + gamma * updated_value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                   for a in range(env.action_space.n)]

            value_table[s] = max(Q_values)


        if (np.sum(np.fabs(updated_value_table - value_table)) <=
threshold):
             break

    return value_table
```

اکنون که تابع ارزش بهینه را با بیشینه‌گیری ارزشهای Q محاسبه کردیم، بیایید ببینیم که چگونه خطمشی بهینه را از تابع ارزش بهینه استخراج کنیم.

استخراج سیاست بهینه از تابع ارزش بهینه

در مرحله قبل، **تابع ارزش بهینه** را محاسبه کردیم. حال، بیایید ببینیم که چگونه **سیاست بهینه** را از تابع ارزش بهینه محاسبه شده استخراج کنیم.

ابتدا، تابعی به نام `extract_policy` تعریف می‌کنیم که `value_table` را به عنوان پارامتر می‌گیرد:

```
def extract_policy(value_table):
```

ضریب تخفیف γ را روی ۱ قرار دهید:

```
    gamma = 1.0
```

ابتدا، **خط‌مشی** را با صفر مقداردهی می‌کنیم، یعنی اقدامات را برای همه حالت‌ها صفر می‌کنیم:

```
    policy = np.zeros(env.observation_space.n)
```

اکنون تابع Q را با استفاده از **تابع ارزش** بهینه بدست آمده از مرحله قبل، محاسبه می‌کنیم. ما آموختیم که تابع Q را می‌توان به صورت زیر محاسبه کرد:

$$Q(s,a) = \sum_{s'} P^a_{SS'}\left[R^a_{SS'} + \gamma V(s')\right]$$

پس از محاسبه تابع Q، می‌توانیم با انتخاب **اقدامی** که حداکثر **ارزش** Q را دارد، **سیاست** را استخراج کنیم. از آنجایی که ما تابع Q را با استفاده از تابع ارزش بهینه محاسبه می‌کنیم، سیاست استخراج شده از تابع Q، خط‌مشی بهینه خواهد بود.

$$\pi^* = \arg\max_a Q(s,a)$$

همانطور که کد زیر نشان می‌دهد، برای هر حالت، ارزش‌های Q را برای تمام اقدامات موجود در حالت، محاسبه می‌کنیم و سپس با انتخاب اقدامی که حداکثر ارزش Q را دارد، سیاست را استخراج می‌کنیم.

برای هر حالت:

```
    for s in range(env.observation_space.n):
```

ارزش Q تمام اقدامات موجود در حالت را محاسبه کنید، $Q(s,a) = \sum_{s'} P^a_{SS'}\left[R^a_{SS'} + \gamma V(s')\right]$:

```
        Q_values = [sum([prob*(r + gamma * value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                   for a in range(env.action_space.n)]
```

با انتخاب اقدامی که حداکثر ارزش Q را دارد، $\pi^* = \arg\max_a Q(s,a)$ سیاست را استخراج کنید:

```
        policy[s] = np.argmax(np.array(Q_values))


    return policy
```

قطعه کامل تابع `extract_policy` در اینجا نشان داده شده است تا شفافیت بیشتری به ما بدهد:

```
def extract_policy(value_table):

    gamma = 1.0

    policy = np.zeros(env.observation_space.n)

    for s in range(env.observation_space.n):

        Q_values = [sum([prob*(r + gamma * value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                   for a in range(env.action_space.n)]

        policy[s] = np.argmax(np.array(Q_values))

    return policy
```

کل داستان همین بود! اکنون، خواهیم دید که چگونه سیاست بهینه را در محیط دریاچه منجمد خود استخراج کنیم.

همایندِ مطالب (جمع‌بندی)[۱]

همانطور که دیدید، در محیط دریاچه یخ‌زده، هدف ما یافتن خط‌مشی بهینه‌ای است که اقدام صحیح را در هر حالت انتخاب می‌کند تا بتوانیم از حالت **A** بدون بازدید از حالت‌های حفره، به حالت **G** برسیم.

ابتدا، تابع ارزش بهینه را با استفاده از تابع `value_iteration` با درنظرگرفتن محیط دریاچه یخ‌زده به عنوان پارامتر، محاسبه می‌کنیم:

```
optimal_value_function = value_iteration(env)
```

سپس، خط‌مشی بهینه را با استفاده از تابع `extract_policy`، از تابع ارزش بهینه استخراج می‌کنیم:

```
optimal_policy = extract_policy(optimal_value_function)
```

ما می‌توانیم خط‌مشی بهینه به دست آمده را چاپ کنیم:

```
print(optimal_policy)
```

کد قبلی موارد زیر را چاپ می‌کند. همانطور که می‌توانیم مشاهده کنیم، سیاست بهینه ما به ما می‌گوید که در هر حالت، اقدام صحیح را انجام دهیم:

```
[0. 3. 3. 3. 0. 0. 0. 0. 3. 1. 0. 0. 0. 2. 1. 0.]
```

اکنون که ما یاد گرفتیم که ***تکرار ارزش*** چیست و چگونه روش ***تکرار ارزش*** را برای محاسبه ***خط‌مشی*** بهینه در محیط دریاچه یخ‌زده خود انجام دهیم، در بخش بعدی با روش جالب دیگری به نام ***تکرار سیاست*** آشنا خواهیم شد.

---

[۱] Putting It All Together

تکرار سیاست

در روش **تکرار ارزش**، ابتدا **تابع ارزش بهینه** را با بیشینه‌گیری بر **تابع** Q (مقادیر Q) بصورت تکراری محاسبه کردیم. هنگامی که تابع ارزش بهینه را پیدا کردیم، از آن برای استخراج **سیاست بهینه** استفاده کردیم. درحالی که در روش **تکرار خط‌مشی**، سعی می‌کنیم **تابع ارزش بهینه** را با استفاده از **سیاست** به صورت تکراری، محاسبه کنیم، تا زمانی که تابع ارزش بهینه را بیابیم، و می‌توانیم از آن برای استخراج **سیاست** بهینه استفاده کنیم.

ابتدا بیایید یاد بگیریم که چگونه **تابع ارزش** را با استفاده از یک **سیاست** محاسبه کنیم. فرض کنید ما یک سیاست π داریم، چگونه می‌توانیم **تابع ارزش** را با استفاده از **سیاست** π محاسبه کنیم؟ در اینجا، می‌توانیم از معادله بلمن استفاده کنیم. ما آموختیم که با توجه به معادله بلمن، می‌توانیم **تابع ارزش** را با استفاده از **سیاست** π، به صورت زیر محاسبه کنیم:

$$V^{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

بیایید فرض کنیم سیاست ما یک **سیاست قطعی** است، بنابراین می‌توانیم این عبارت $\sum_a \pi(a|s)$ را از معادله پیشین حذف کنیم، چون الان با **سیاست اتفاقی** کاری نداریم. به این ترتیب، معادله بلمن را به صورت زیر بازنویسی می‌کنیم:

$$V^{\pi}(s) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi}(s')]$$

برای سادگی نمادگذاری، می‌توانیم $P(s'|s,a)$ را با $P^{a}_{SS'}$ و $R(s,a,s')$ با $R^{a}_{SS'}$ نشان دهیم و معادله قبلی را به صورت زیر بنویسیم:

$$V^{\pi}(s) = \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')]$$

بنابراین، با استفاده از معادله بالا **می‌توانیم تابع ارزش** را با استفاده از یک **سیاست** محاسبه کنیم. هدف ما یافتن تابع ارزش بهینه است زیرا هنگامی که تابع ارزش بهینه را پیدا کردیم، می‌توانیم از آن برای استخراج خط‌مشی بهینه استفاده کنیم.

در اینجا هیچ سیاستی به عنوان یک ورودی به ما داده نخواهد شد. بنابراین، ما یک سیاست تصادفی (دلبخواهی) را مقداردهی اولیه می‌کنیم و تابع ارزش را با استفاده از این سیاست تصادفی (بختکی) محاسبه می‌کنیم. سپس بررسی می‌کنیم که آیا تابع ارزش محاسبه شده بهینه است یا خیر. البته قاعدتا بهینه نخواهد بود زیرا براساس خط‌مشی تصادفی (دلبخواهی) محاسبه می‌شود.

بنابراین، یک سیاست جدید از تابع ارزش محاسبه شده، استخراج می‌کنیم، سپس از سیاست جدید استخراج شده برای محاسبه تابع ارزش جدید استفاده می‌کنیم و نهایتا بررسی می‌کنیم که آیا تابع ارزش جدید بهینه است یا خیر. اگر بهینه بود، متوقف می‌شویم، در غیر این صورت این مراحل را برای یک سری تکرارها، تکرار می‌کنیم. برای درک بهتر، به مراحل زیر توجه کنید.

تکرار ۱: فرض کنید π_0 **سیاست** تصادفی (بختکی) ما باشد. ما از این **سیاست** تصادفی برای محاسبه **تابع ارزش** V^{π_0} استفاده می‌کنیم. قاعدتا تابع ارزش ما بهینه نخواهد بود زیرا بر اساس خط‌مشی تصادفی (بختکی) محاسبه می‌شود. بنابراین، از V^{π_0}، یک سیاست جدید π_1 استخراج می‌کنیم.

تکرار ۲: اکنون از **سیاست جدید** π_1 بدست آمده از تکرار قبلی، برای محاسبه **تابع ارزش جدید** V^{π_1}، استفاده می‌کنیم، سپس بررسی می‌کنیم که آیا V^{π_1} بهینه است یا خیر. اگر بهینه باشد، توقف می‌کنیم، در غیر این صورت از این تابع ارزش V^{π_1}، یک خط مشی جدید π_2 استخراج می‌کنیم.

تکرار ۳: اکنون از **سیاست جدید** $\pi_۲$، بدست آمده از تکرار قبلی برای محاسبه **تابع ارزش جدید** $V^{\pi_۲}$ استفاده می‌کنیم، سپس بررسی می‌کنیم که آیا $V^{\pi_۲}$ بهینه است یا خیر. اگر بهینه باشد، توقف می‌کنیم، در غیر این صورت از این تابع ارزش $V^{\pi_۲}$، یک خط‌مشی جدید $\pi_۳$ استخراج می‌کنیم.

این فرآیند را برای چندین بار تکرار می‌کنیم تا زمانی که تابع ارزش بهینه V^{π^*} را، همانطور که در زیر نشان داده شده است، پیدا کنیم:

$$\pi_۰ \rightarrow V^{\pi_۰} \rightarrow \pi_۱ \rightarrow V^{\pi_۱} \rightarrow \pi_۲ \rightarrow V^{\pi_۲}\pi_۳ \rightarrow V^{\pi_۳} \rightarrow \cdots \rightarrow \pi^* \rightarrow V^{\pi^*}$$

مرحله قبل، **ارزیابی** و **بهبود** سیاست نامیده می‌شود. ارزیابی سیاست به این معناست که در هر مرحله، خط‌مشی را با بررسی بهینه بودن تابع ارزش محاسبه شده با استفاده از آن خط‌مشی، ارزیابی می‌کنیم. بهبود سیاست به این معنی است که در هر مرحله، خط‌مشی بهبودیافته جدیدی را برای محاسبه تابع ارزش بهینه پیدا می‌کنیم.

هنگامی‌که **تابع ارزش بهینه** V^{π^*} را پیدا کردیم، به این معنی است که **سیاست بهینه** را نیز پیدا کرده‌ایم. یعنی اگر V^{π^*}، بهینه باشد، سیاستی که برای محاسبه V^{π^*} استفاده می‌شود، یک خط‌مشی بهینه خواهد بود.

برای درک بهتر نحوه عملکرد **تکرار خط‌مشی**، اجازه دهید مراحل زیر را با شبه کد[۱] بررسی کنیم. در اولین تکرار، یک خط‌مشی تصادفی (بختکی) را مقداردهی اولیه می‌کنیم و از آن برای محاسبه تابع ارزش استفاده می‌کنیم:

```
policy = random_policy
value_function = compute_value_function(policy)
```

از آنجایی که ما تابع ارزش را با استفاده از سیاست تصادفی محاسبه کردیم، قاعدتا تابع ارزش محاسبه‌شده بهینه نخواهد بود. بنابراین، ما باید یک خط‌مشی جدید پیدا کنیم که با آن بتوانیم تابع ارزش بهینه را محاسبه کنیم.

[۱] Pseudocode

بنابراین، یک خط‌مشی جدید از تابع ارزش محاسبه شده با استفاده از یک خط‌مشی یا سیاست تصادفی (دلبخواهی)، استخراج می‌کنیم:

```
new_policy = extract_policy(value_function)
```

اکنون از این **سیاست جدید** برای محاسبه **تابع ارزش جدید** استفاده خواهیم کرد:

```
policy = new_policy
value_function = compute_value_function(policy)
```

اگر تابع ارزش جدید بهینه باشد، توقف می‌کنیم، در غیر این صورت گام‌های قبل را برای تعدادی تکرار، ادامه می‌دهیم تا تابع ارزش بهینه را پیدا کنیم. شبه‌کد زیر به ما درک بهتری می‌دهد:

```
policy = random_policy
for i in range(num_iterations):
    value_function = compute_value_function(policy)
    new_policy = extract_policy(value_function)

    if value_function = optimal:
        break
    else:
        policy = new_policy
```

اما صبرکنید! چگونه می‌گوییم تابع ارزش ما بهینه است؟ اگر تابع ارزش، در طول تکرار تغییر نمی‌کند، می‌توان گفت که تابع ارزش ما بهینه است. خوب، چگونه می‌توانیم بررسی کنیم که تابع ارزش در طول تکرار تغییر می‌کند یا خیر؟

ما یاد گرفتیم که تابع ارزش را با استفاده از یک سیاست محاسبه می‌کنیم. اگر خط‌مشی در طول تکرار تغییر نکند، تابع ارزش ما نیز در طول تکرارها تغییر نمی‌کند. بنابراین، هنگامی که سیاست در طول تکرار تغییر نداشته باشد، می‌توان گفت که تابع ارزش بهینه را پیدا کرده‌ایم.

بنابراین، طی یک سری تکرار، وقتی خط‌مشی موجود و خط‌مشی جدید، یکسان می‌شوند، می‌توان گفت که تابع ارزش بهینه را به دست آورده‌ایم. شبه کد نهایی زیر برای فهم بهتر ارائه شده است:

```
policy = random_policy
for i in range(num_iterations):
    value_function = compute_value_function(policy)
    new_policy = extract_policy(value_function)
```

```
    if policy == new_policy:
        break
    else:
        policy = new_policy
```

بنابراین، زمانی که سیاست ، تغییر نمی‌کند، یعنی زمانی که سیاست موجود و سیاست جدید یکسان می‌شوند، می‌توان گفت که تابع ارزش بهینه را به‌دست آورده‌ایم و سیاستی که برای محاسبه تابع ارزش بهینه استفاده می‌شود، خط‌مشی بهینه خواهد بود.

به یاد داشته باشید که در روش **تکرار ارزش**، تابع ارزش بهینه را با استفاده از بیشینه‌گیری تابع Q (ارزش Q)، به صورت تکراری، محاسبه می‌کنیم و پس از یافتن تابع ارزش بهینه، خط‌مشی بهینه را از آن استخراج می‌کنیم. اما در روش **تکرار سیاست**، تابع ارزش بهینه را با استفاده از خط‌مشی، به صورت تکرارشونده، محاسبه می‌کنیم و زمانی که تابع ارزش بهینه را پیدا کردیم، آن سیاستی که برای محاسبه تابع ارزش بهینه استفاده می‌شود، خط‌مشی بهینه خواهد بود.

اکنون که درک اولیه‌ای از نحوه عملکرد روش تکرار خط‌مشی داریم، در بخش بعدی به جزئیات خواهیم پرداخت و نحوه محاسبه تکرار سیاست به صورت دستی را خواهیم آموخت.

الگوریتم ـ تکرار سیاست

مراحل الگوریتم **تکرار سیاست**، به صورت زیر ارائه شده است:

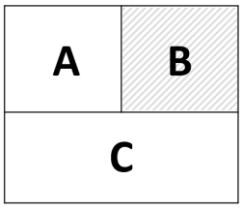
۱. با یک **خط‌مشی** تصادفی (بختکی)، کار را شروع کنید.

۲. **تابع ارزش** را با استفاده از **سیاست** داده شده، محاسبه کنید.

۳. یک **خط‌مشی جدید** را با استفاده از **تابع ارزش** به دست آمده از مرحله ۲، استخراج کنید.

۴. اگر **سیاست** استخراج شده، با **خط‌مشی** استفاده شده در مرحله ۲ یکسان است، توقف کنید، در غیر این صورت **خط‌مشی جدید** استخراج شده را به مرحله ۲ ببرید و مراحل ۲ تا ۴ را تکرار کنید.

حالا بیایید وارد جزئیات شویم و یاد بگیریم که مراحل قبلی، دقیقا چگونه کار می‌کنند. برای درک واضح، اجازه دهید تکرار سیاست را به صورت دستی انجام دهیم. بیایید همان محیط جهان شبکه‌ای را که در روش تکرار ارزش استفاده کردیم، در نظر بگیریم. فرض کنید در حالت **A** هستیم و هدف ما رسیدن به حالت **C** بدون بازدید از حالت سایه‌دار **B** است و می‌گوییم دو اقدام داریم: عدد ۰ برای اقدامات چپ یا *راست* و عدد ۱ برای حرکات *بالا* یا *پایین*:



شکل ۳.۱۰: محیط دنیای مشبک

می‌دانیم که در محیط فوق، سیاست بهینه همان است که به ما می‌گوید اقدام ۱ را در حالت **A** انجام دهیم تا بتوانیم بدون بازدید از حالت **A** به حالت **C** برسیم. حال خواهیم دید که چگونه با استفاده از تکرار سیاست، این خطمشی بهینه را پیدا کنیم.

جدول ۳.۱۴ پویایی‌های مدل حالت **A** را نشان می‌دهد:

حالت (s)	اقدام (a)	حالت بعدی (s')	احتمال گذار $P(s'\|s,a) or\ P^a_{ss'}$	تابع پاداش $R(s,a,s') or\ R^a_{ss'}$
A	۰	**A**	۰.۱	۰
A	۰	**B**	۰.۸	−۱
A	۰	**C**	۰.۱	۱

A	۱	**A**	۰.۱	۰
A	۱	**B**	۰.۰	−۱
A	۱	**C**	۰.۹	۰

جدول ۳.۱۴: پویایی‌های مدل حالت **A**

مرحله ۱ – شروع اولیه با یك سیاست تصادفی (بختکی)

ابتدا، یک سیاست تصادفی را برای شروع اولیه درنظر می‌گیریم. همانطور که زیر نشان می‌دهد، خطمشی تصادفی یا دلبخواه، به ما می‌گوید که اقدام ۱ را در حالت **A**، اقدام ۰ را در حالت **B** و اقدام ۱ را در حالت **C** انجام دهیم:

$$A \rightarrow ۱$$
$$B \rightarrow ۰$$
$$C \rightarrow ۱$$

مرحله ۲ ـ محاسبه تابع ارزش با استفاده از سیاست داده شده

این مرحله، با اختلاف کمی، دقیقاً مشابه نحوه محاسبه تابع ارزش در روش تکرار ارزش است. در تکرار ارزش، تابع ارزش را با بیشینه‌گیری بر تابع Q محاسبه کردیم. اما در اینجا، در تکرار خطمشی، تابع ارزش را با استفاده از سیاست، محاسبه می‌کنیم.

برای درک بهتر این مرحله، بیایید به سرعت به یاد بیاوریم که چگونه تابع ارزش را در تکرار ارزش، محاسبه می‌کنیم. در تکرار ارزش، تابع ارزش بهینه را به عنوان بیشینه تابع Q بهینه محاسبه می‌کنیم، همانطور که در ادامه نشان داده‌ایم:

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = P^a_{SS'}[R^a_{SS'} + \gamma V(s')]$$

در تکرار خط مشی، ما تابع ارزش را با استفاده از یک خط‌مشی، محاسبه می‌کنیم، **برخلاف تکرار ارزش، که در آن تابع ارزش را با استفاده از حداکثرگیری روی تابع Q، محاسبه می‌کنیم. تابع ارزش، با استفاده از یک سیاست π، می‌تواند به صورت زیر محاسبه شود:**

$$V^{\pi}(s) = \sum_{s'} P^{a}_{SS'}\left[R^{a}_{SS'} + \gamma V^{\pi}(s')\right]$$

اگر به معادله قبل نگاه کنید، برای محاسبه تابع ارزش، به احتمال گذار $P^{a}_{SS'}$، تابع پاداش $R^{a}_{SS'}$ و ارزش حالت بعدی $V^{\pi}(s')$ نیاز داریم. مقادیر احتمال گذار $P^{a}_{SS'}$، و تابع پاداش $R^{a}_{SS'}$، می‌توانند از پویایی‌های مدل بدست بیایند. اما ارزش حالت بعدی $V^{\pi}(s')$ چطور؟ ما ارزش هیچ حالتی را در حال حاضر نمی‌دانیم. بنابراین، تابع ارزش (ارزش‌های حالت) را با مقادیر تصادفی (بختکی) یا صفر (همانند شکل ۳.۱۵) مقداردهی اولیه می‌کنیم و تابع ارزش را محاسبه می‌کنیم:

حالت	ارزش
A	۰
B	۰
C	۰

جدول ۳.۱۵: جدول ارزش اولیه

تکرار اول:

بیایید ارزش حالت **A** را محاسبه کنیم (توجه داشته باشید که در اینجا، بر خلاف تکرار ارزش، که در آن ارزش Q را برای تمام اقدامات موجود در حالت محاسبه کرده و حداکثر مقدار را انتخاب می‌کنیم، فقط ارزش اقدام ارائه شده توسط سیاست را محاسبه می‌کنیم).

بنابراین، اقدام ارائه شده توسط سیاست، برای حالت **A** برابر اقدام ۱ است و می‌توانیم ارزش حالت **A** را همانطور که در ادامه نشان می‌دهیم، محاسبه کنیم (توجه داشته باشید که در سراسر این بخش از یک ضریب تخفیف $\gamma = ۱$ استفاده کرده‌ایم):

$$V(A) = P_{AA}^{۱}[R_{AA}^{۱} + \gamma V(A)] + P_{AB}^{۱}[R_{AB}^{۱} + \gamma V(B)] + P_{AC}^{۱}[R_{AC}^{۱} + \gamma V(C)]$$
$$= ۰.۱(۰ + ۰) + ۰.۰(-۱ + ۰) + ۰.۹(۱ + ۰)$$
$$= ۰.۹$$

به طور مشابه، ما ارزش را برای همه حالت‌ها با استفاده از اقدام ارائه شده توسط سیاست محاسبه می‌کنیم. جدول ۳.۱۶ ارزش‌های حالت به روزشده به دست آمده در نتیجه اولین تکرار را نشان می‌دهد.

حالت	ارزش
A	۰.۹
B	-۰.۲
C	۰.۱

جدول ۳.۱۶: جدول ارزش از تکرار ۱

با این حال، تابع ارزش (جدول ارزش) نشان داده شده در جدول ۳.۱۶ که در نتیجه اولین تکرار به دست آمده است، دقیق نخواهد بود. یعنی ارزشهای حالت (تابع ارزش)، طبق سیاست داده شده دقیق نخواهد بود.

توجه داشته باشید که برخلاف روش تکرار ارزش، در اینجا ما بررسی نمی‌کنیم که آیا تابع ارزش ما بهینه است یا خیر. ما فقط بررسی می‌کنیم که آیا تابع ارزش ما مطابق با خطمشی داده شده، با دقت محاسبه شده است یا خیر.

تابع ارزش، دقیق نخواهد بود، زیرا زمانی که ما محاسبه تابع ارزش را با استفاده از خطمشی داده شده شروع کردیم، از ارزشهای حالت اولیه به طور تصادفی (بختکی) استفاده کردیم:

$$V^{\pi}(s) = \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma V^{\pi}(s')\right]$$

ارزشهای تصادفی

بنابراین، در تکرار بعدی، هنگام محاسبه تابع ارزش، از ارزش‌های حالت به دست آمده از اولین تکرار به‌روز شده استفاده می‌کنیم:

$$V^{\pi}(s) = \sum_{S'} P^{a}_{SS'}\left[R^{a}_{SS'} + \gamma V^{\pi}(s')\right]$$

ارزشهای حالت از تکرار اول استفاده می شود.

تکرار دوم:

اکنون، در تکرار دوم، تابع ارزش را با استفاده از خط‌مشی π محاسبه می‌کنیم. به یاد داشته باشید که هنگام محاسبه ***تابع ارزش***، از مقادیر حالت به روز شده (جدول ارزش‌ها) بدست آمده از تکرار اول استفاده می‌کنیم.

به عنوان مثال، بیایید ارزش حالت **A** را محاسبه کنیم:

$$\begin{aligned} V(A) &= P^{\backprime}_{AA}[R^{\backprime}_{AA} + \gamma V(A)] + P^{\backprime}_{AB}[R^{\backprime}_{AB} + \gamma V(B)] + P^{\backprime}_{AC}[R^{\backprime}_{AC} + \gamma V(C)] \\ &= ۰.۱(۰ + ۰.۹) + ۰.۰(-۱ - ۰.۲) + ۰.۹(۱ + ۰.۱) \\ &= ۱.۰۸ \end{aligned}$$

به طور مشابه، ما ارزش را برای همه حالت‌ها با استفاده از اقدام ارائه شده توسط سیاست ، محاسبه می‌کنیم. جدول ۳.۱۷، ارزش‌های حالت به روز شده به دست آمده در نتیجه تکرار دوم را نشان می‌دهد:

حالت	ارزش
A	۱.۰۸
B	-۰.۵
C	۰.۵

جدول ۳.۱۷: جدول ارزش از تکرار ۲

تکرار سوم:

به همین ترتیب، در تکرار ۳، تابع ارزش را با استفاده از سیاست π محاسبه می‌کنیم و در حین محاسبه تابع ارزش، از ارزشهای حالت به روز شده (جدول ارزش) به دست آمده از تکرار ۲ استفاده می‌کنیم.

جدول ۳.۱۸، ارزش‌های حالت به روز شده به دست آمده از تکرار سوم را نشان می‌دهد:

حالت	ارزش
A	۱.۴۵
B	−۰.۹
C	۰.۶

جدول ۳.۱۸: جدول ارزش از تکرار ۳

ما این کار را برای بسیاری از تکرارها، تکرار می‌کنیم تا زمانی که جدول ارزش تغییر نکند یا با تکرارها تغییر بسیار کمی داشته باشد. به عنوان مثال، فرض کنید جدول ۳.۱۹ جدول ارزش‌های بدست آمده در نتیجه تکرار ۴ را نشان می‌دهد:

حالت	ارزش
A	۱.۴۶
B	−۰.۹
C	۰.۶۱

جدول ۳.۱۹: جدول ارزش از تکرار ۴

همانطور که می‌بینیم، تفاوت بین جداول ارزش بدست آمده از تکرار ۴ و تکرار ۳ بسیار کم است. بنابراین، می‌توان گفت که جدول ارزش، در طول تکرارها تغییر چندانی نمی‌کند و روی این تکرار متوقف می‌شویم و آن را به عنوان تابع ارزش نهایی خود در نظر می‌گیریم.

مرحله ۳ ـ استخراج سیاست جدید با استفاده از تابع ارزش به دست آمده از مرحله قبل

در نتیجه مرحله ۲، **تابع ارزش** را به دست آوردیم که با استفاده از **خط‌مشی** تصادفی (دلبخواه) داده شده محاسبه می‌شود. با این حال، این **تابع ارزش**، بهینه نخواهد بود زیرا با استفاده از **سیاست** تصادفی محاسبه می‌شود. بنابراین یک خط‌مشی جدید از تابع ارزش بدست آمده در مرحله قبل استخراج می‌کند. تابع ارزش (جدول ارزش) به دست آمده از مرحله قبل در جدول ۳.۲۰ نشان داده شده است.

حالت	ارزش
A	۱.۴۶
B	−۰.۹
C	۰.۶۱

جدول ۳.۲۰: جدول ارزش از مرحله قبل

خب، چگونه می‌توانیم یک خط‌مشی جدید را از تابع ارزش استخراج کنیم؟ (نکته: این مرحله دقیقاً مشابه نحوه استخراج یک خط‌مشی با داشتن **تابع ارزش**، در قدم ۲ از روش تکرار ارزش است.)

برای استخراج یک سیاست جدید، تابع Q را با استفاده از **تابع ارزش** (جدول ارزش) به دست آمده از مرحله قبل محاسبه می‌کنیم. هنگامی که **تابع** Q را محاسبه می‌کنیم، ما اقداماتی را در هر حالت انتخاب می‌کنیم که حداکثر ارزش را به عنوان یک **خط‌مشی جدید** دارند. می‌دانیم که تابع Q را می‌توان به صورت زیر محاسبه کرد:

$$Q(s,a) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V(s')]$$

حال در حین محاسبه ارزش‌های Q از تابع ارزشی که از مرحله قبل به دست آوردیم، استفاده می‌کنیم.

به عنوان مثال، بیایید ارزش Q را برای همه اقدامات در حالت **A**، با استفاده از تابع ارزش بدست آمده از مرحله قبل، محاسبه کنیم. ارزش Q برای اقدام ۰ در حالت **A**، به صورت زیر محاسبه می‌شود:

$$Q(A,۰) = P_{AA}^{۰}[R_{AA}^{۰} + \gamma V(A)] + P_{AB}^{۰}[R_{AB}^{۰} + \gamma V(B)] + P_{AC}^{۰}[R_{AC}^{۰} + \gamma V(C)]$$
$$= ۰.۱(۰ + ۱.۴۶) + ۰.۸(-۱ - ۰.۹) + ۰.۱(۱ + ۰.۶۱)$$
$$= -۱.۲۱$$

ارزش Q برای اقدام ۱ در حالت **A**، به صورت زیر محاسبه می‌شود:

$$Q(A,۱) = P_{AA}^{۱}[R_{AA}^{۱} + \gamma V(A)] + P_{AB}^{۱}[R_{AB}^{۱} + \gamma V(B)] + P_{AC}^{۱}[R_{AC}^{۱} + \gamma V(C)]$$
$$= ۰.۱(۰ + ۱.۴۶) + ۰.۰(-۱ - ۰.۹) + ۰.۹(۱ + ۰.۶۱)$$
$$= ۱.۵۹$$

جدول ۳.۲۱، جدول Q را پس از محاسبه ارزش‌های Q برای همه زوج‌های حالت–اقدام، نشان می‌دهد:

حالت	اقدام	ارزش
A	۰	-۱.۲۱
A	۱	۱.۵۹
B	۰	۰.۱
B	۱	۰.۰
C	۰	۰.۵
C	۱	۰.۰

جدول ۳.۲۱: جدول Q

از این جدول Q، در هر حالت، اقداماتی را که حداکثر ارزش را دارند به عنوان یک خط‌مشی جدید انتخاب می‌کنیم.

$$A \rightarrow ۱$$
$$B \rightarrow ۰$$
$$C \rightarrow ۰$$

مرحله ٤ – بررسی سیاست جدید

اکنون، بررسی می‌کنیم که آیا خط‌مشی جدید استخراج شده از مرحله ۳، با خط‌مشی مورد استفاده ما در مرحله ۲، یکسان است یا خیر. اگر یکسان است، متوقف می‌شویم، در غیر این صورت، سیاست جدید استخراج شده را به مرحله ۲ منتقل می‌کنیم و مراحل ۲ تا ۴ را تکرار می‌کنیم.

بنابراین، در این بخش، نحوه محاسبه سیاست بهینه را با استفاده از روش تکرار خط‌مشی، یاد گرفتیم. در بخش بعدی، نحوه اجرای روش تکرار سیاست برای محاسبه سیاست بهینه در محیط دریاچه یخ‌زده را با استفاده از جعبه ابزار جیم، خواهیم آموخت.

حل مسئله دریاچه یخ‌زده با تکرار سیاست

ما یاد گرفتیم که در محیط دریاچه منجمد، هدف ما رسیدن به حالت هدف **G** با شروع از حالت **S** و بدون بازدید از حالت‌های حفره **H** است. حال بیایید نحوه محاسبه خط‌مشی بهینه را با استفاده از ***روش تکرار خط‌مشی*** و در محیط دریاچه‌یخ زده محاسبه کنیم.

ابتدا بیایید کتابخانه های لازم را وارد کنیم:

```
import gym
import numpy as np
```

حال بیایید محیط دریاچه یخ زده را با استفاده از جیم ایجاد کنیم:

```
env = gym.make('FrozenLake-v0')
```

تا اینجا یاد گرفتیم که در ***روش تکرار سیاست***، ما تابع ارزش را با استفاده از خط‌مشی، به صورت تکراری محاسبه می‌کنیم. هنگامی که تابع ارزش بهینه را پیدا کردیم، سیاستی که برای محاسبه تابع ارزش بهینه استفاده می‌شود، سیاست بهینه خواهد بود.

بنابراین، ابتدا بیایید نحوه محاسبه ***تابع ارزش*** را با استفاده از ***خط‌مشی*** یاد بگیریم.

محاسبه تابع ارزش با استفاده از سیاست

این مرحله در **روش تکرار سیاست**، دقیقاً مشابه همان روشی است که با استفاده از آن، تابع ارزش را در روش تکرار ارزش محاسبه کردیم، اما تفاوت اندکی وجود دارد. در اینجا، تابع ارزش را با استفاده از سیاست، محاسبه می‌کنیم، اما در **روش تکرار ارزش**، تابع ارزش را با بیشینه‌گیری از ارزش‌های Q محاسبه می‌کنیم. حالا بیایید یاد بگیریم که چگونه تابعی را تعریف کنیم که تابع ارزش را با استفاده از سیاست داده شده، محاسبه می‌کند.

ابتدا تابعی به نام `compute_vlaue_function` تعریف می‌کنیم که خطمشی را به عنوان پارامتر در نظر می‌گیرد:

```
def compute_value_function(policy):
```

حالا بیایید تعداد تکرارها را تعریف کنیم:

```
    num_iterations = 1000
```

ارزش آستانه[۱] را تعریف کنید:

```
    threshold = 1e-20
```

مقدار ضریب تخفیف γ را برابر ۱ قرار می‌دهیم:

```
    gamma = 1.0
```

اکنون، با صفر کردن تمام ارزشهای حالت، جدول ارزشها را مقداردهی اولیه می‌کنیم:

```
    value_table = np.zeros(env.observation_space.n)
```

[۱] Threshold Value

برای هر تکرار:

```
    for i in range(num_iterations):
```

جدول ارزش را به‌روزرسانی می‌کنیم؛ یاد گرفتیم که در هر تکرار، از جدول ارزشهای به‌روز شده (ارزشهای حالت) از تکرار قبلی، استفاده می‌کنیم:

```
        updated_value_table = np.copy(value_table)
```

اکنون، **تابع ارزش** را با استفاده از **سیاست داده شده** محاسبه می‌کنیم. ما آموختیم که یک **تابع ارزش** را می‌توان با توجه به برخی **سیاست‌های π**، به صورت زیر، محاسبه کرد:

$$V^{\pi}(s) = \sum_{s'} P^{a}_{SS'}[R^{a}_{SS'} + \gamma V^{\pi}(s')]$$

بنابراین، برای هر حالت، اقدام را مطابق با سیاست انتخاب می‌کنیم و سپس ارزش حالت را با استفاده از اقدام انتخاب شده، به صورت بیان شده در ادامه، به روزرسانی می‌کنیم:

برای هر حالت:

```
        for s in range(env.observation_space.n):
```

اقدام در حالت را با توجه به سیاست، انتخاب کنید:

```
            a = policy[s]
```

ارزش حالت را با استفاده از اقدام انتخاب شده محاسبه کنید: $V^{\pi}(s) = \sum_{s'} P^{a}_{SS'}[R^{a}_{SS'} + \gamma V^{\pi}(s')]$

```
            value_table[s] = sum(
                [prob * (r + gamma * updated_value_table[s_])
                    for prob, s_, r, _ in env.P[s][a]])
```

پس از محاسبه جدول ارزش، یعنی ارزش همه حالت‌ها، بررسی می‌کنیم که آیا تفاوت بین جدول ارزشهای به دست آمده در تکرار فعلی و تکرار قبلی، کمتر یا مساوی یک ارزش آستانه است یا خیر. اگر کمتر باشد، حلقه را می‌شکنیم و جدول ارزش را به عنوان تابع ارزش دقیق سیاست داده شده، برمی‌گردانیم:

```
        if (np.sum(np.fabs(updated_value_table - value_table)) <=
threshold):
            break

    return value_table
```

اکنون که تابع ارزش خطمشی را محاسبه کردیم، بیایید ببینیم که چگونه خطمشی را از تابع‌ارزش، استخراج می‌کنیم.

استخراج سیاست از تابع ارزش

این مرحله در ***روش تکرار سیاست***، دقیقاً مشابه نحوه استخراج خطمشی از تابع ارزش، در ***روش تکرار ارزش*** است. بنابراین، مشابه آنچه در روش تکرار ارزش، یاد گرفتیم، تابعی به نام `extract_policy` برای استخراج یک سیاست با توجه به تابع ارزش، تعریف می‌کنیم:

```
def extract_policy(value_table):

    gamma = 1.0
    policy = np.zeros(env.observation_space.n)
    for s in range(env.observation_space.n):

        Q_values = [sum([prob*(r + gamma * value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])
                                   for a in range(env.action_space.n)]

        policy[s] = np.argmax(np.array(Q_values))

    return policy
```

همایندِ مطالب گفته شده

ابتدا اجازه دهید تابعی به نام `policy_iteration` تعریف کنیم که محیط را به عنوان پارامتر در نظر می‌گیرد:

```
def policy_iteration(env):
```

تعداد تکرارها را تنظیم کنید:

```
    num_iterations = 1000
```

ما آموختیم که در **روش تکرار سیاست**، با مقداردهی اولیه یک خط‌مشی تصادفی (بختکی)، شروع می‌کنیم. بنابراین، سیاست تصادفی را مقداردهی اولیه می‌کنیم، که اقدام ۰ را در همه حالت‌ها، انتخاب می‌کند:

```
    policy = np.zeros(env.observation_space.n)
```

برای هر تکرار:

```
    for i in range(num_iterations):
```

تابع ارزش را با استفاده از خط‌مشی، محاسبه کنید:

```
        value_function = compute_value_function(policy)
```

سیاست جدید را از **تابع ارزش محاسبه شده**، استخراج کنید:

```
        new_policy = extract_policy(value_function)
```

اگر `policy` و `new_policy`، یکسان هستند، حلقه را بشکنید:

```
        if (np.all(policy == new_policy)):
            break
```

در غیر این صورت، خطمشی فعلی را به `new_policy` به‌روزرسانی کنید:

```
        policy = new_policy

    return policy
```

اکنون، بیایید نحوه استفاده از ***روش تکرار سیاست*** و یافتن سیاست بهینه در محیط دریاچه یخ زده را بیاموزیم. بنابراین، ما فقط محیط دریاچه یخ زده را برای تابع `policy_iteration` به کار می‌گیریم و سیاست بهینه را دریافت می‌کنیم (همانطور که در اینجا نشان داده شده است):

```
optimal_policy = policy_iteration(env)
```

ما می‌توانیم سیاست بهینه را چاپ کنیم:

```
print(optimal_policy)
```

کد قبلی، موارد زیر را چاپ می‌کند:

```
array([0., 3., 3., 3., 0., 0., 0., 0., 3., 1., 0., 0., 0., 2., 1., 0.])
```

همانطور که می‌توانیم مشاهده کنیم، خطمشی بهینه، به ما می‌گوید که در هر حالت، اقدام صحیح را انجام دهیم. بنابراین، ما یاد گرفتیم که چگونه روش تکرار خطمشی را برای محاسبه خطمشی بهینه استفاده کنیم.

آیا برنامه‌ریزی پویا برای همه محیط‌ها قابل استفاده است؟

در برنامه‌ریزی پویا، یعنی در ***روش تکرار ارزش*** و در ***روش تکرار سیاست***، سعی می‌کنیم سیاست بهینه را پیدا کنیم.

تکرار ارزش: در ***روش تکرار ارزش***، تابع ارزش بهینه را با بیشینه‌گیری از تابع Q (ارزش‌های Q) به صورت مکرر، محاسبه می‌کنیم:

$$V^*(s) = \max_a Q^*(s,a)$$

که در آن، $Q(s,a) = \sum_{s'} P^a_{SS'}\left[R^a_{SS'} + \gamma V(s')\right]$. بعد از پیداکردن تابع ارزش بهینه، سیاست بهینه را از آن استخراج می‌کنیم.

تکرار سیاست: در ***روش تکرار سیاست***، تابع ارزش بهینه را با استفاده از خطمشی و به صورت مکرر، محاسبه می‌کنیم:

$$V^\pi(s) = \sum_{s'} P^a_{SS'}[R^a_{SS'} + \gamma V^\pi(s')]$$

ما با سیاست تصادفی (دلبخواه) شروع می‌کنیم و تابع ارزش را محاسبه می‌کنیم. بعد اینکه، تابع ارزش بهینه را پیدا کردیم، سیاستی که برای ایجاد تابع ارزش بهینه استفاده می‌شود، خطمشی بهینه خواهد بود.

اگر به دو معادله قبل نگاه کنید، برای یافتن خطمشی بهینه، تابع ارزش و تابع Q را محاسبه می‌کنیم. اما برای محاسبه ارزش و تابع Q، نیاز داریم که **احتمال گذار** $P^a_{SS'}$ محیط را بدانیم و زمانی که ما **احتمال گذار** محیط را نمی‌دانیم، نمی‌توانیم تابع ارزش و تابع Q را محاسبه کنیم تا سیاست بهینه را پیدا کنیم.

یعنی برنامه‌ریزی پویا، یک روش مبتنی بر مدل است و برای اعمال این روش، باید پویایی‌های مدل (احتمال گذار) محیط را بدانیم و وقتی پویایی‌های مدل را نمی‌دانیم، نمی‌توانیم روش برنامه‌ریزی پویا را اعمال کنیم. .

خیلی خب، در حالی که **پویایی‌های محیط** را نمی‌دانیم، چگونه می‌توانیم خط‌مشی بهینه را پیدا کنیم؟ در چنین حالتی، می‌توانیم از روش‌های بدون مدل استفاده کنیم. در فصل بعدی با یکی از روش‌های جالب بدون مدل به نام **روش مونت‌کارلو** آشنا می شویم و نحوه استفاده از آن برای یافتن سیاست بهینه بدون نیاز به دانستن **پویایی‌های مدل** را می‌آموزیم.

خلاصه

ما این فصل را با درک معادله بلمن از **تابع ارزش** و **تابع** Q، شروع کردیم. ما آموختیم که بر اساس معادله بلمن، **ارزش** یک حالت، مجموع **پاداش فوری** و **ارزش تنزیل شده** حالت بعدی است و ارزش یک زوج حالت–اقدام، مجموع پاداش فوری و تنزیل شده زوج حالت–اقدام بعدی است. سپس درمورد تابع ارزش بهینه بلمن و تابع Q، آموختیم که حداکثر ارزش را می‌دهد.

بعد از آن، ما در مورد رابطه بین **تابع ارزش** و **تابع** Q یاد گرفتیم. آموختیم که تابع ارزش را می‌توان از تابع Q، به عنوان $V^*(s) = \max_a Q^*(s,a)$ استخراج کرد و سپس یاد گرفتیم که تابع Q، می‌تواند از تابع ارزش به صورت $Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$، مستخرج شود.

سپس، با دو روش جالب به نامهای ***تکرار ارزش*** و ***تکرار سیاست*** آشنا شدیم، که از برنامه‌ریزی پویا برای یافتن خط‌مشی بهینه استفاده می‌کنند.

در ***روش تکرار ارزش***، ابتدا تابع ارزش بهینه را با بیشینه‌گیری بر روی تابع Q به صورت تکراری محاسبه می‌کنیم. هنگامی که تابع ارزش بهینه را پیدا کردیم، از آن برای استخراج سیاست بهینه، استفاده می‌کنیم. در ***روش تکرار***

سیاست، سعی می‌کنیم، تابع ارزش بهینه را با استفاده از خط‌مشی به صورت تکراری، محاسبه کنیم. هنگامی که تابع ارزش بهینه را پیدا کردیم، سیاستی که برای ایجاد تابع مقدار بهینه استفاده می‌شود، به عنوان سیاست بهینه استخراج می‌شود.

سوالات

بیایید سعی کنیم به سوالات زیر پاسخ دهیم تا دانش خود را در مورد آنچه در این فصل آموختیم، ارزیابی کنیم:

۱. معادله بلمن را تعریف کنید.
۲. تفاوت بین معادله انتظاری بلمن و معادله بهینگی بلمن[۱] چیست؟
۳. چگونه **تابع ارزش** را از **تابع** Q استخراج کنیم؟
٤. چگونه **تابع** Q را از **تابع ارزش** استخراج کنیم؟
٥. مراحل مربوط به روش **تکرار ارزش** چیست؟
٦. مراحل مربوط به روش **تکرار سیاست (خط‌مشی)** چیست؟
۷. روش **تکرار سیاست** چه تفاوتی با روش **تکرار ارزش** دارد؟

[۱] Bellman Optimality Equation

فصل چهارم

روش‌های مونت کارلو

در فصل قبل، یاد گرفتیم که چگونه خط‌مشی بهینه را با استفاده از دو روش جذاب **برنامه‌ریزی پویا** به نام **تکرار ارزش** و **تکرار سیاست**، محاسبه کنیم. **برنامه‌ریزی پویا یک روش مبتنی بر مدل است و برای محاسبه تابع ارزش و تابع Q به منظور پیداکردن سیاست بهینه، به پویایی مدل محیط نیاز دارد.**

اما بیایید فرض کنیم که **پویایی مدل محیط** را نداریم. در آن صورت، چگونه **تابع ارزش** و **تابع Q** را محاسبه کنیم؟ اینجا، همان جایی است که از **روش‌های بدون مدل** استفاده می‌کنیم. روش‌های بدون مدل، به **پویایی‌های مدل محیط** برای محاسبه تابع ارزش و تابع Q، به منظور یافتن خط‌مشی بهینه، نیاز ندارند. یکی از این روش‌های محبوب بدون مدل، **روش مونت کارلو (MC)** است.

ما این فصل را با شناخت **روش مونت‌کارلو** آغاز می‌کنیم، سپس به دو نوع وظیفه مهم در یادگیری تقویتی به نام وظایف **پیش‌بینی** و **کنترل**[۱] می‌پردازیم. بعداً یاد خواهیم گرفت که چگونه از روش مونت‌کارلو، در یادگیری تقویتی استفاده می‌شود و در مقایسه با **روش برنامه‌ریزی پویا** (که در فصل قبل با آن آشنا شدیم)، چقدر سودمند است. در ادامه متوجه خواهیم شد که روش **پیش‌بینی مونت‌کارلو** و انواع مختلف زیر–روش‌های **پیش‌بینی مونت‌کارلو** چیست. همچنین یاد خواهیم گرفت که چگونه یک عامل را برای بازی بلک جک با روش پیش‌بینی مونت کارلو آموزش دهیم.

سپس با روش **کنترل مونت کارلو** و انواع روش‌های فرعی **کنترل مونت کارلو** آشنا می شویم. در ادامه یاد می‌گیریم که چگونه با روش کنترل مونت کارلو یک عامل را برای بازی بلک جک آموزش دهیم.

به طور خلاصه، در این فصل با موضوعات زیر آشنا می شویم:

- **آشنایی با روش مونت کارلو**
- **وظایف پیش بینی و کنترل**
- **روش پیش بینی مونت کارلو**
- **بازی بلک جک با روش پیش بینی مونت کارلو**
- **روش کنترل مونت کارلو**
- **بازی بلک جک با روش کنترل مونت کارلو**

[۱] Prediction and Control Tasks

آشنایی با روش مونت کارلو

قبل از اینکه بفهمیم روش مونت کارلو چگونه در یادگیری تقویتی به کار می‌آید، ابتدا بیایید بدانیم که روش مونت کارلو چیست و چگونه کار می کند. **روش مونت کارلو یک تکنیک آماری است که برای یافتن راه‌حل تقریبی نمونه‌گیری** استفاده می‌کند.

به عنوان مثال، روش مونت کارلو با نمونه‌گیری، مقدار مورد انتظار یک متغیر تصادفی را تقریب می‌زند، و زمانی که اندازه نمونه بیشتر باشد، تقریب بهتر خواهد بود. فرض کنید یک متغیر تصادفی X داریم و می‌خواهیم مقدار مورد انتظار X (امید ریاضی X)، یعنی $E(X)$، را محاسبه کنیم. می‌توانیم این مقدار را با جمع مقادیر X ضرب در احتمالات مربوط به آنها به صورت زیر محاسبه کنیم:

$$E(X) = \sum_{i=1}^{N} x_i p(x_i)$$

اما آیا به جای محاسبه امیدهای ریاضی به این صورت، می‌توانیم آن را با روش مونت کارلو آنرا تقریب بزنیم؟ بله! می‌توانیم مقدار مورد انتظار X را فقط با نمونه‌برداری از مقادیر X برای N بار تخمین بزنیم و مقدار متوسط X را به عنوان مقدار مورد انتظار X به صورت زیر محاسبه کنیم:

$$\mathbb{E}_{x\sim p(x)}[X] \approx (\frac{1}{N}) \sum_{i} x_i$$

وقتی N بزرگتر باشد، تقریب ما بهتر خواهد بود. بنابراین با روش مونت کارلو می‌توان از طریق نمونه‌گیری، جواب را تقریب زد و زمانی که اندازه نمونه بزرگ باشد تقریب ما بهتر خواهد بود. در بخشهای آینده، جزئیات نحوه استفاده از روش مونت کارلو در یادگیری تقویتی را خواهیم آموخت.

وظایف پیش بینی و کنترل

در یادگیری تقویتی، ما دو وظیفه مهم را انجام می‌دهیم که عبارتند از: **وظیفه پیش‌بینی** و **وظیفه کنترل**

وظیفه پیش‌بینی (احصاء)[۱]

در وظیفه پیش‌بینی، یک خط‌مشی (سیاست) π به عنوان ورودی داده می‌شود و سعی می‌کنیم با استفاده از سیاست داده شده، تابع ارزش یا تابع Q، را پیش‌بینی کنیم. اما انجام این کار چه فایده‌ای دارد؟ هدف ما ارزیابی **سیاست** داده شده است. یعنی باید مشخص کنیم که **سیاست** داده شده خوب است یا بد. چگونه می‌توانیم این موضوع را بررسی کنیم؟ اگر عامل با استفاده از **سیاست** داده شده، **بازده** خوبی به دست آورد، می‌توانیم بگوییم که **سیاست** ما خوب است. بنابراین، برای ارزیابی **خط‌مشی** داده شده، باید بدانیم که اگر عامل از **سیاست** داده شده استفاده کند، چه **بازگشتی** (**بازدهی**) به دست می‌آورد. در اینجا برای بدست آوردن **بازده** عملا **تابع ارزش** یا **تابع** Q را با استفاده از **خط‌مشی** داده شده پیش‌بینی می‌کنیم.

به این معنا که ما آموختیم که **تابع ارزش** یا ارزش یک حالت، نشان‌دهنده بازده مورد انتظاری است که یک عامل با شروع از آن حالت و با انجام اقداماتی مطابق با **سیاست‌های** π به دست می‌آورد. بنابراین، با **پیش‌بینی تابع ارزش** با استفاده از **سیاست** π مدنظر، می‌توانیم بفهمیم که اگر عامل از **سیاست** π مذکور استفاده کند، چه بازدهی مورد انتظاری را در هر حالت به دست می‌آورد. اگر بازگشت خوب باشد، می توان گفت که آن سیاست خوب است.

به طور مشابه، ما آموختیم که تابع Q یا ارزش Q نشان‌دهنده بازده مورد انتظاری است که عامل با شروع از حالت s و انجام یک اقدام a پیرو سیاست π به دست می‌آورد. بنابراین، با **پیش‌بینی تابع** Q با استفاده از **خط‌مشی** π مذکور، می‌توانیم بفهمیم که اگر عامل از خط‌مشی داده‌شده استفاده کند، چه بازدهی مورد انتظاری را در هر زوج حالت–اقدام به دست می‌آورد. اگر بازگشت خوب باشد، می توان گفت که سیاست مربوطه خوب است. بنابراین، می‌توانیم خط‌مشی

[۱] Prediction Task

داده شده را با محاسبه تابع ارزش و تابع Q ارزیابی کنیم.

توجه داشته باشید که **در وظیفه پیش‌بینی، هیچ تغییری در سیاست ورودی داده شده ایجاد نمی‌کنیم**. ما خطمشی داده شده را ثابت نگه می‌داریم و تابع ارزش یا تابع Q را با استفاده از آن خطمشی، پیش‌بینی می‌کنیم و بازده مورد انتظار را بدست می‌آوریم. بر اساس بازده مورد انتظار، ما سیاست داده شده را ارزیابی می‌کنیم.

وظیفه کنترل (وارسی)[۱]

برخلاف وظیفه پیش‌بینی، **در وظیفه کنترل، هیچ سیاستی به عنوان ورودی به ما داده نمی‌شود. در وظیفه کنترل، هدف ما یافتن سیاست بهینه است**. بنابراین، ما با یک **سیاست تصادفی** (بختکی) شروع می‌کنیم و سعی می‌کنیم با تکرار این روش، **خطمشی بهینه** را پیدا کنیم. یعنی سعی می‌کنیم سیاست بهینه‌ای پیدا کنیم که حداکثر **بازدهی** را داشته باشد.

بنابراین، به طور خلاصه، در وظیفه **پیش‌بینی (احصاء)**، ما **خطمشی** ورودی داده شده را با پیش‌بینی **تابع ارزش** یا **تابع** Q ارزیابی می‌کنیم، تا بازده مورد انتظاری حاصل از بکارگیری یک **خطمشی** معین را بدست آوریم. حال آنکه در وظیفه **کنترل (وارسی)**، هدف ما یافتن **خطمشی** بهینه است و **هیچ سیاستی به عنوان ورودی به ما داده نخواهد شد**. بنابراین ما با مقداردهی اولیه یک خطمشی تصادفی شروع کرده و سعی می‌نمائیم خطمشی بهینه را به طور مکرر پیدا کنیم.

اکنون که متوجه شدیم وظایف **پیش‌بینی** و **کنترل** چیست، در بخش بعدی، نحوه استفاده از روش مونت کارلو برای انجام وظایف **پیش‌بینی** و **کنترل** را خواهیم آموخت.

---

[۱] Control Task

پیش‌بینی مونت کارلو

در این بخش، نحوه استفاده از روش مونت کارلو برای انجام **وظیفه پیش‌بینی** را یاد می‌گیریم. ما آموخته‌ایم که در وظیفه پیش‌بینی، یک **خط‌مشی** به ما داده می‌شود و ما **تابع ارزش** یا **تابع** Q را با استفاده از این سیاست، پیش‌بینی و ارزیابی می‌کنیم.

ابتدا یاد می‌گیریم که چگونه **تابع ارزش** را با استفاده از **سیاست** داده شده و با روش مونت‌کارلو پیش‌بینی کنیم. بعداً به پیش‌بینی **تابع** Q با استفاده از **خط‌مشی** خود نگاه خواهیم کرد. خب، بیایید این بخش را شروع کنیم.

چرا برای پیش‌بینی تابع ارزش یک سیاست معین، به روش مونت کارلو نیاز داریم؟ چرا نمی‌توانیم تابع ارزش را با استفاده از روش‌های برنامه‌ریزی پویا که در فصل قبل با آن آشنا شدیم پیش‌بینی کنیم؟ ما یاد گرفتیم که برای محاسبه تابع ارزش با استفاده از روش برنامه‌ریزی پویا، باید پویایی مدل (احتمال انتقال) را بدانیم و **زمانی که پویایی مدل را نمی‌دانیم، از روش‌های بدون مدل استفاده می‌کنیم.**

روش مونت کارلو یک **روش بدون مدل**[۱] است، به این معنی که به پویایی‌های مدل برای محاسبه تابع ارزش نیاز ندارد.

ابتدا اجازه دهید تعریف **تابع ارزش** را مرور کنیم. **تابع ارزش** یا **ارزش** حالت s را می‌توان به عنوان بازده مورد انتظاری که عامل با شروع از حالت S و پیروی از **خط‌مشی** π به دست می‌آورد، تعریف کرد. می‌توان آن را به صورت زیر بیان کرد:

$$V^{\pi}(s) = \underset{\tau \sim \pi}{\mathbb{E}}[R(\tau)|s_{.} = s]$$

خب، چگونه می‌توانیم ارزش حالت (**تابع ارزش**) را با استفاده از روش مونت کارلو تخمین بزنیم؟ در ابتدای فصل متوجه

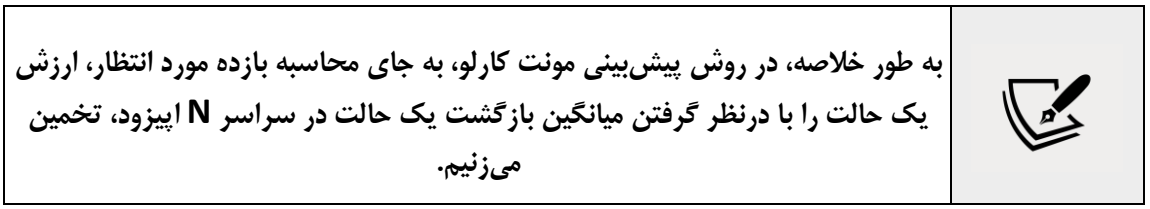
[۱] Model-Free Method

شدیم که روش مونت‌کارلو با نمونه‌گیری، ارزش مورد انتظار یک متغیر تصادفی را تقریب می‌زند و زمانی که اندازه نمونه بیشتر باشد، تقریب بهتر خواهد بود. آیا می‌توانیم از این مفهوم روش مونت کارلو برای پیش‌بینی ارزش یک حالت استفاده کنیم؟ بله!

به منظور تقریب **ارزش** حالت با استفاده از روش مونت‌کارلو، ما از اپیزودها (یا مسیرها) و براساس **خط‌مشی** داده‌شده برای مثال N بار، نمونه می‌گیریم و سپس ارزش یک حالت را از طریق میانگین بازگشت آن حالت، در سراسر اپیزودهای نمونه‌گیری شده، محاسبه می‌کنیم. فرمول این کار را می‌توان به صورت زیر بیان کرد:

$$V(s) \approx \frac{1}{N}\sum_{i=1}^{N} R_i(s)$$

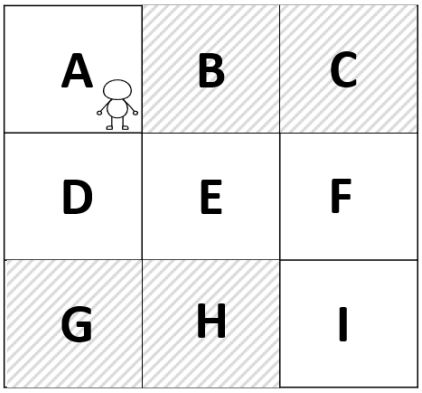
از معادله قبلی، می‌توان فهمید که ارزش یک حالت s را می‌توان با محاسبه میانگین بازگشت حالت s در N اپیزود، تقریب زد. تقریب ما زمانی بهتر خواهد بود که N بیشتر باشد.



به طور خلاصه، در روش پیش‌بینی مونت کارلو، به جای محاسبه بازده مورد انتظار، ارزش یک حالت را با درنظر گرفتن میانگین بازگشت یک حالت در سراسر N اپیزود، تخمین می‌زنیم.

خب، بیایید با یک مثال درک بهتری از نحوه تخمین **ارزش** یک حالت (**تابع ارزش**) با استفاده از روش مونت کارلو، داشته باشیم. برای این کار، همان محیط جهان مشبک[1] مورد علاقه خود از فصل یک را، در نظر بگیرید (شکل ۴.۱). هدف ما رسیدن به حالت **I** از حالت **A** بدون بازدید از حالت‌های سایه‌دار است، و عامل وقتی از حالت‌های بدون سایه بازدید می‌کند، پاداش ۱+ و هنگام بازدید از حالت‌های سایه‌دار، پاداش ۱- دریافت می‌کند:

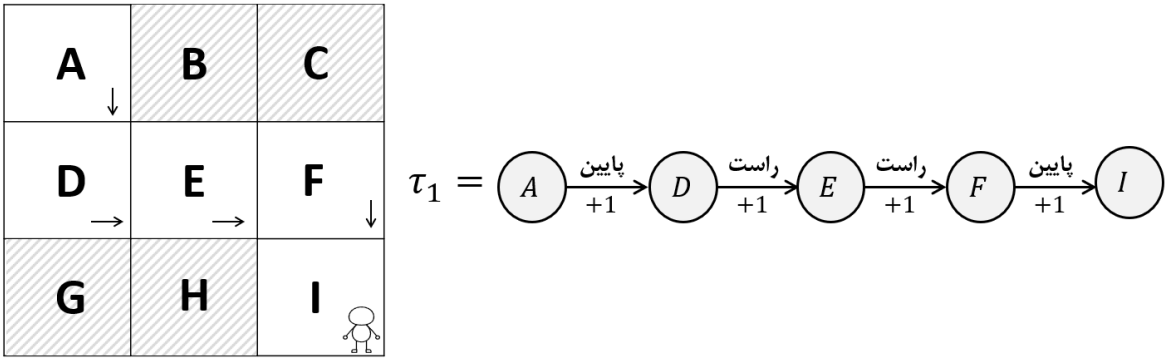
[1] Grid World Environment



شکل ۴.۱: محیط دنیای مشبک

فرض کنید که ما یک **سیاست اتفاقی** (**پشامدی**) π داریم. همچنین فرض کنید، در حالت **A**، **سیاست اتفاقی** π ما، ۸۰ درصد از مواقع، اقدام *پایین* و ۲۰ درصد از اوقات اقدام *راست* را انتخاب می‌کند، و بعلاوه، ۱۰۰درصد زمانها، اقدام *راست* را در حالت‌های **D** و **E** و اقدام *پایین* را در حالت‌های **B** و **F** انتخاب می‌کند.

ابتدا، یک پردینه (اپیزود) τ_1 را با استفاده از **خطمشی اتفاقی** داده شده π، که در شکل ۴.۲ نشان‌داده شده است، تولید می‌کنیم.



شکل ۴.۲: پردینه (اپیزود) τ_1

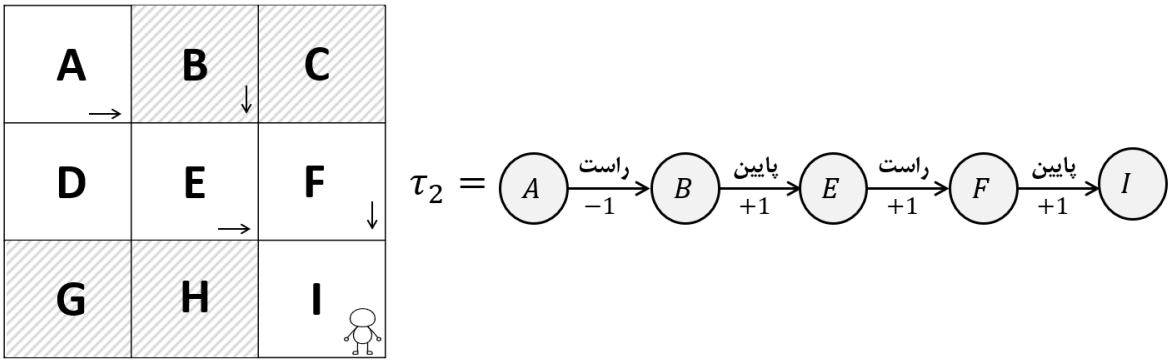
برای یک درک بهتر، اجازه دهید فقط بر روی حالت **A** تمرکز کنیم. اکنون بازگشت یا بازده حالت **A** را محاسبه می‌کنیم. بازگشت یا بازده یک حالت، مجموع پاداش‌های مسیری است که از آن حالت شروع می‌شود. بنابراین، **بازده**

حالت **A** به صورت زیر محاسبه می‌شود:

$$R_۱(A) = ۱ + ۱ + ۱ + ۱ = ۴$$

که در آن زیرنویس ۱ در $R_۱$ بازگشت از پردینه (اپیزود) ۱ یا بازده آنرا نشان می‌دهد.

فرض کنید ما یک پردینه (اپیزود) دیگر $\tau_۲$ را با استفاده از همان **خطمشی اتفاقی** π داده شده، همانند شکل ۴.۳، تولید می‌کنیم:

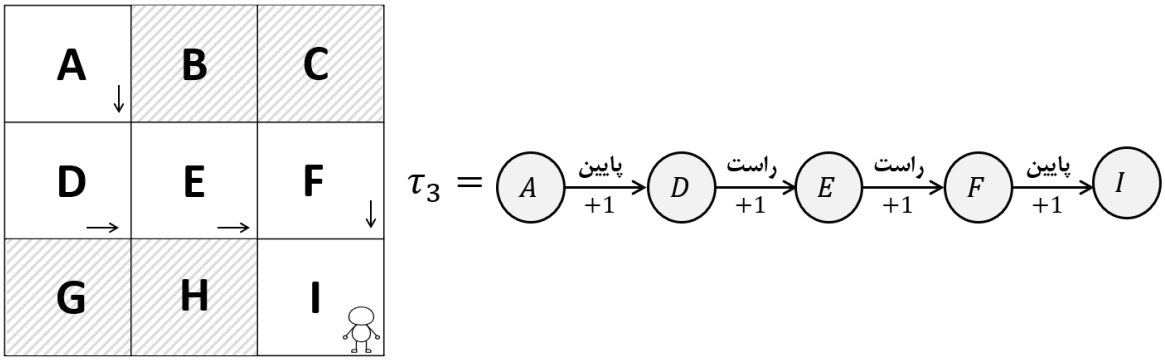


شکل ۴.۳: پردینه (اپیزود) $\tau_۲$

حال بیایید **بازده** حالت **A** را محاسبه کنیم. بازگشت حالت **A** یا بازده آن برابر است با:

$$R_۲(A) = -۱ + ۱ + ۱ + ۱ = ۲$$

فرض کنید ما یک پردینه (اپیزود) دیگر $\tau_۳$، را با استفاده از همان **خطمشی** احتمالی داده شده در شکل ۴.۴ تولید می‌کنیم:



شکل ۴.۴: پردینه (اپیزود) τ_3

حال بیایید **بازده** حالت **A** را محاسبه کنیم. بازگشت این حالت معادل $R_3(A) = 1 + 1 + 1 + 1 = 4$ است.

بنابراین، ما سه پردینه تولید کردیم و بازگشت حالت **A** را در هر سه پردینه محاسبه کردیم. حال چگونه می‌توانیم ارزش حالت **A** را محاسبه کنیم؟ ما آموختیم که در روش مونت‌کارلو، ارزش یک حالت را می‌توان با محاسبه میانگین بازگشت حالت در N اپیزود (یا مسیرهای Nگانه) تقریب زد:

$$V(s) \approx \frac{1}{N}\sum_{i=1}^{N} R_i(s)$$

ما باید ***ارزش*** حالت **A** را محاسبه کنیم، بنابراین می‌توانیم آن را با در نظر گرفتن میانگین بازدهی حالت **A** در سراسر N اپیزود، به صورت زیر محاسبه کنیم:

$$V(A) \approx \frac{1}{N}\sum_{i=1}^{N} R_i(A)$$

ما سه پردینه تولید کرده‌ایم، درنتیجه:

$$V(A) \approx \frac{1}{3}\sum_{i=1}^{3} R_i(A)$$

$$V(A) = \frac{۱}{۳}(R_۱(A) + R_۲(A) + R_۳(A))$$

$$V(A) = \frac{۴ + ۲ + ۴}{۳} = ۳.۳$$

بنابراین، ارزش حالت **A**، معادل ۳.۳ است. به طور مشابه، ما می‌توانیم ارزش همه حالت‌های دیگر را فقط با در نظر گرفتن میانگین بازده حالت در سه پردینه، محاسبه کنیم.

برای درک آسان‌تر، در مثال قبلی، ما فقط سه پردینه، تولید کردیم. برای یافتن تخمین بهتر و دقیق‌تری از ارزش حالت، باید اپیزودهای زیادی (نه فقط سه) تولید کنیم و میانگین بازده حالت را به عنوان ارزش حالت محاسبه کنیم.

بنابراین، در روش پیش‌بینی مونت کارلو، برای پیش‌بینی ارزش یک حالت (**تابع ارزش**) با استفاده از **خط‌مشی** π ورودی داده شده، تعداد N پردینه، با استفاده از **سیاست** داده شده تولید کرده و سپس ارزش یک حالت را به عنوان میانگین بازگشت حالت در سراسر این N پردینه، محاسبه می‌کنیم.

توجه داشته باشید که هنگام محاسبه بازگشت حالت، ما می‌توانیم ضریب تخفیف را هم لحاظ کنیم و بازده تنزیل شده را محاسبه کنیم، اما برای سادگی ضریب تخفیف لحاظ نشده است.

اکنون، که ما درک اولیه‌ای از نحوه کار «**روش پیش‌بینی مونت کارلو**» برای پیش‌بینی تابع ارزش یک **سیاست** داده شده را پیدا کردیم، بیایید با ورود به جزئیات آن، الگوریتم روش پیش‌بینی مونت کارلو را بهتر یاد بگیریم. در بخش در بخش بعدی این جزئیات را بررسی کنیم.

الگوریتم پیش‌بینی مونت‌کارلو

الگوریتم **پیش‌بینی مونت کارلو** به صورت زیر ارائه شده است:

۱. اجازه دهید $\text{total\_return}(s)$، نشاندهنده مجموع بازگشت (**بازده**) یک حالت در چندین پردینه و $N(s)$ بیانگر یک شمارنده باشد، یعنی تعداد دفعاتی که یک حالت در چندین پردینه (اپیزود) اپیزود، بازدید شده است. $\text{total\_return}(s)$ و $N(s)$ را برای همه حالت‌ها، برابر صفر قرارد دهید. **خط‌مشی** π، به عنوان ورودی داده می‌شود.

۲. برای M بار تکرار:

۱. با استفاده از **خط‌مشی** π، یک پردینه ایجاد کنید.

۲. تمام **پاداش‌های** به دست آمده در اپیزود را در لیستی به نام **پاداش** ذخیره کنید.

۳. برای هر گام t در اپیزود:

۱. بازگشت (بازده) حالت s_t را به صورت $\text{R}(\text{s}_\text{t}) = \text{sum}(\text{rewards}[\text{t:}])$ محاسبه کنید.

۲. **بازده** کل حالت s_t را به صورت زیر، به‌روز کنید.

$$\text{total\_returns}(\text{s}_\text{t}) = \text{total\_returns}(\text{s}_\text{t}) + \text{R}(\text{s}_\text{t})$$

۳. شمارنده را به صورت $N(s_t) = N(s_t) + ۱$ به روز کنید.

۳. **ارزش** یک حالت را فقط با میانگین‌گیری، محاسبه کنید، یعنی:

$$V(s) = \frac{\text{total\_return(s)}}{N(s)}$$

الگوریتم بالا نشان می‌دهد که **ارزش** حالت، فقط میانگین **بازده** حالت در اپیزودهای قبلی است.

برای درک بهتر نحوه عملکرد الگوریتم بالا، بیایید یک مثال ساده بزنیم و ارزش هر حالت را به صورت دستی محاسبه کنیم. فرض کنید باید ارزش سه حالت s_0، s_1 و s_2 را محاسبه کنیم. ما می‌دانیم که وقتی از حالتی به حالت دیگر منتقل می‌شویم، یک پاداشی به دست می‌آوریم. بنابراین، پاداش برای حالت نهایی، صفر خواهد بود زیرا ما هیچ انتقالی

از حالت نهایی انجام نمی‌دهیم. بنابراین، ارزش حالت نهایی $S_۲$، صفر خواهد بود. حال باید ارزش دو حالت $S_۰$ و $S_۱$ را پیدا کنیم.

	**بخشهای بعدی با محاسبات دستی توضیح داده میشوند، برای فهم بیشتر، با یک مداد و کاغذ ادامه دهید.**

مرحله ۱:

همان طور که جدول ۴.۱ نشان می‌دهد، $\text{total\_returns}(s)$ و $N(s)$ را برای همه حالت‌ها برابر صفر، مقداردهی کنید:

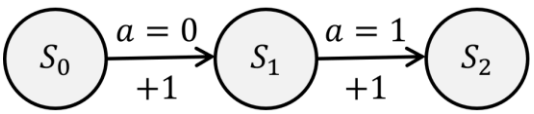
حالت	$\text{total\_returns}(s)$	$N(s)$
s_0	۰	۰
s_1	۰	۰

جدول ۴.۱: مقادیر (ارزشهای) اولیه

به ما یک **خطمشی اتفاقی** داده شده است. در حالت s_0، **خطمشی اتفاقی** ما، اقدام ۰ را برای ۵۰٪ مواقع و اقدام ۱ را برای ۵۰٪ مواقع انتخاب می‌کند. همچنین این خطمشی، اقدام ۱ را در حالت $S_۱$ برای ۱۰۰٪ مواقع انتخاب می‌کند.

مرحله ۲: تکرار ۱

همانطور که در شکل ۴.۵ نشان داده شده است، یک پردینه (اپیزود)، با استفاده از **خطمشی** π ورودی داده شده ایجاد کنید:



شکل ۴.۵: ایجاد یک پردینه با استفاده از **خطمشی** π داده شده

تمام پاداش‌های به دست آمده در پردینه را در لیستی به نام **پاداش** ذخیره کنید. بنابراین، $\text{rewards} = [۱, ۱]$

ابتدا **بازده** حالت s_0، (مجموع پاداش از s_0) را محاسبه می‌کنیم:

$$\begin{aligned} R(s_۰) &= \text{sum}(\text{rewards}[۰:]) \\ &= \text{sum}([۱,۱]) \\ &= ۲ \end{aligned}$$

بازده کل حالت s_0 را در جدول به صورت زیر به روز کنید:

$$\begin{aligned} \text{total\_returns}(s_۰) &= \text{total\_return}(s_۰) + R(s_۰) \\ &= ۰ + ۲ = ۲ \end{aligned}$$

تعداد دفعاتی که از حالت s_0 در جدول ما بازدید شده است، را به‌روزرسانی کنید:

$$\begin{aligned} N(s_۰) &= N(s_۰) + ۱ \\ &= ۰ + ۱ = ۱ \end{aligned}$$

حال، بیایید بازده حالت $s_۱$ (مجموع پاداش‌ها $s_۱$) را محاسبه کنیم:

$$\begin{aligned} R(s_۱) &= \text{sum}(\text{rewards}[۱:]) \\ &= \text{sum}([۱]) \\ &= ۱ \end{aligned}$$

بازده کل حالت $s_۱$ را در جدول به صورت زیر به روز کنید:

$$\begin{aligned} \text{total\_return}(s_۱) &= \text{total\_returns}(s_۱) + R(s_۱) \\ &= ۰ + ۱ = ۱ \end{aligned}$$

تعداد دفعاتی که از حالت $s_۱$ در جدول ما بازدید شده است، را به‌روزرسانی کنید:

$$\begin{aligned} N(s_۱) &= N(s_۱) + ۱ \\ &= ۰ + ۱ = ۱ \end{aligned}$$

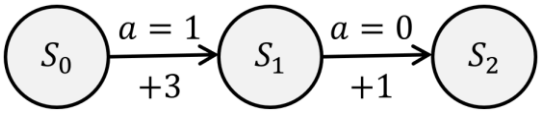
جدول به روز شده ما، پس از تکرار ۱، به شرح زیر است:

حالت	total_returns(s)	$N(s)$
$\mathbf{s_0}$	۲	۱
$\mathbf{s_1}$	۱	۱

جدول ۴.۲: جدول به روز شده پس از اولین تکرار

مرحله ۲: تکرار ۲

فرض کنید ما پردینه دیگری را با استفاده از همان **سیاست** π داده شده که شکل ۴.۶ نشان می‌دهد، تولید می‌کنیم:



شکل ۴.۶: ایجاد یک پردینه (اپیزود) با استفاده از **خط‌مشی** π داده شده

تمام پاداش‌های به دست آمده در اپیزود را در لیستی به نام پاداش ذخیره کنید. بنابراین، $\text{rewards} = [۳,۱]$

ابتدا بازده حالت $S_.$ (مجموع پاداش از $S_.$) را محاسبه می‌کنیم:

$$\begin{aligned} R(s_.) &= \text{sum}(\text{rewards}[۰:]) \\ &= \text{sum}([۳,۱]) \\ &= ۴ \end{aligned}$$

بازده کل حالت $S_.$ را در جدول به صورت زیر به روز کنید:

$$\begin{aligned} \text{total\_returns}(s_.) &= \text{total\_returns}(s_.) + R(s_.) \\ &= ۲ + ۴ = ۶ \end{aligned}$$

تعداد دفعاتی که از حالت $S_.$ در جدول ما بازدید شده است، به‌روزرسانی کنید:

$$N(s_۰) = N(s_۰) + ۱$$
$$= ۱ + ۱ = ۲$$

حال، بیایید بازده حالت $s_۱$ (مجموع پاداش از $s_۱$) را محاسبه کنیم:

$$R(s_۱) = \text{sum}(\text{rewards}[۱:])$$
$$= \text{sum}([۱])$$
$$= ۱$$

بازده حالت $s_۱$ را در جدول به صورت زیر به روز کنید:

$$\text{total\_returns}(s_۱) = \text{total\_returns}(s_۱) + R(s_۱)$$
$$= ۱ + ۱ = ۲$$

تعداد دفعات بازدید از حالت را به‌روزرسانی کنید:

$$N(s_۱) = N(s_۱) + ۱$$
$$= ۱ + ۱ = ۲$$

جدول به روز شده ما پس از تکرار دوم به شرح زیر است:

حالت	$\text{total\_returns}(s)$	$N(s)$
s_0	۶	۲
s_1	۲	۲

جدول ۴.۳: جدول به‌روز شده بعد از تکرار دوم

از آنجایی که ما به صورت دستی محاسبه می‌کنیم، برای سادگی، اجازه دهید در دو تکرار توقف کنیم. یعنی ما فقط دو پردینه تولید می‌کنیم.

مرحله ۳:

اکنون می توانیم ارزش حالت را به صورت زیر محاسبه کنیم:

$$V(s) = \frac{\text{total\_returns}(s)}{N(s)}$$

بدین ترتیب:

$$V(s_۰) = \frac{\text{total\_returns}(s_۰)}{N(s_۰)} = \frac{۶}{۲} = ۳$$

$$V(s_۱) = \frac{\text{total\_returns}(s_۱)}{N(s_۱)} = \frac{۲}{۲} = ۱$$

بنابراین، ما ارزش حالت را فقط با درنظر گرفتن میانگین بازده در چندین پردینه، محاسبه کردیم. توجه داشته باشید که در مثال قبل، برای محاسبات دستی خود، ما فقط دو پردینه تولید کردیم، اما برای تخمین بهتر ارزش حالت، چندین پردینه تولید می‌کنیم و سپس میانگین بازده آن پردینه‌ها (نه فقط ۲) را محاسبه می‌کنیم.

انواع پیش بینی مونت‌کارلو

ما به تازگی یاد گرفتیم که الگوریتم **پیش‌بینی مونت کارلو** چگونه کار می‌کند. می‌توانیم الگوریتم پیش‌بینی مونت کارلو را به دو نوع دسته‌بندی کنیم:

- **مونت کارلوی اولین بازدید**[۱]
- **مونت کارلوی هر بازدید**[۲]

[۱] First-Visit Monte Carlo
[۲] Every-Visit Monte Carlo

روش مونت کارلوی اولین بازدید

ما آموختیم که در روش پیش‌بینی مونت کارلو، ارزش حالت را فقط با در نظر گرفتن میانگین بازده حالت در چندین اپیزود تخمین می‌زنیم. می‌دانیم که در هر پردینه، یک حالت را می‌توان چندین بار بازدید کرد. در **روش اولین بازدید مونت کارلو**، اگر همان حالت دوباره در همان پردینه، بازدید شود، بازگشت یا بازده آن حالت را دوباره محاسبه نمی‌کنیم. به عنوان مثال، حالتی را در نظر بگیرید که یک عامل در حال بازی مار و پله[۱] است. اگر عامل بر روی مار فرود آید، پس احتمال زیادی وجود دارد که عامل به حالتی که قبلاً از آن بازدید کرده بود بازگردد. بنابراین، هنگامی که عامل دوباره همان حالت را مشاهده می‌کند، ما بازده آن حالت را برای بار دوم محاسبه نمی‌کنیم.

در ادامه، **الگوریتم مونت کارلوی اولین بازدید** را نشان می‌دهیم. همانطور که جمله پررنگ شده زیر، بیان می‌کند، ما بازده حالت s_t را فقط در صورتی محاسبه می‌کنیم که این بازدید برای اولین بار در اپیزود اتفاق افتد:

۱. اجازه دهیدtotal_return(s)، بیانگر مجموع **بازگشت** یا **بازده** یک حالت در چندین پردینه و $N(s)$ شمارنده باشد، یعنی تعداد دفعاتی که یک حالت در چندین پردینه بازدید شده است. مقادیر اولیه این دو پارامتر را برای همه حالت‌ها صفر در نظر بگیرید. حال **خط‌مشی** π به عنوان ورودی، داده می‌شود.

۲. برای M تعداد تکرار:

۱. با استفاده از **خط‌مشی** π، یک پردینه ایجاد کنید.

۲. **پاداش‌های** به دست آمده در پردینه را در لیستی به نام **پاداش‌ها** ذخیره کنید.

۳. برای هر مرحله t در پردینه:

اگر حالت s_t برای اولین بار در اپیزود اتفاق می‌افتد:

۱. **بازده** حالت s_t را به صورت $R(s_t) = \text{sum}(\text{rewards}[t:])$ محاسبه کنید.

۲. **بازده** کل حالت s_t را به صورت زیر، به‌روز کنید:

[۱] Snakes And Ladders

$$\text{total\_return}(s_t) = \text{total\_return}(s_t) + R(s_t)$$

۳. شمارنده را به صورت $N(s_t) = N(s_t) + ۱$ به روز کنید.

۳. **ارزش** یک حالت را فقط با گرفتن میانگین محاسبه کنید، یعنی:

$$V(s) = \frac{\text{total\_return}(s)}{N(s)}$$

روش مونت کارلوی هر بازدید

همانطور که ممکن است حدس زده باشید، ***مونت کارلو هر بازدید*** درست برعکس ***مونت کارلوی اولین بازدید*** است. در اینجا، هر بار که یک حالت در پردینه بازدید می‌شود، بازده را محاسبه می‌کنیم. الگوریتم مونت کارلوی هر بازدید، همان الگوریتمی است که قبلاً در ابتدای این بخش دیدیم و به شرح زیر است.

۱. اجازه دهید $\text{total\_return}(s)$، مجموع **بازگشت** یک حالت در چندین پردینه و $N(s)$ شمارنده باشد، یعنی تعداد دفعاتی که یک حالت در چندین پردینه بازدید شده است. مقدار این دو پارامتر را در همه حالت‌ها برابر صفر قرار دهید. **خط‌مشی** π به عنوان ورودی، داده می‌شود.

۲. برای M تعداد تکرار:

۱. با استفاده از **خط‌مشی** π، یک پردینه ایجاد کنید.

۲. تمام **پاداش‌های** به دست آمده در اپیزود را در لیستی به نام **پاداش‌ها** ذخیره کنید.

۳. برای هر مرحله t در پردینه:

۱. **بازده** حالت s_t را به صورت $R(s_t) = \text{sum}(\text{rewards}[t:])$ محاسبه کنید.

۲. **بازده** کل حالت s_t را به صورت زیر به روزرسانی کنید:

$$\text{total\_return}(s_t) = \text{total\_return}(s_t) + R(s_t)$$

۳. شمارنده را به صورت $N(s_t) = N(s_t) + ۱$ به روز کنید.

۳. **ارزش** یک حالت را فقط با گرفتن میانگین محاسبه کنید، یعنی:

$$V(s) = \frac{\text{total\_return}(s)}{N(s)}$$

اکنون که فهمیدیم روش پیش‌بینی مونت کارلو چگونه تابع ارزش خط‌مشی داده شده را پیش‌بینی می‌کند، در بخش بعدی نحوه پیاده‌سازی روش پیش‌بینی مونت کارلو را خواهیم آموخت.

پیاده‌سازی روش پیش‌بینی مونت کارلو[۱]

در این قسمت نحوه بازی بلک جک با ***روش پیش‌بینی مونت کارلو*** را یاد می گیریم. قبل از ورود به جزئیات، بیایید نحوه عملکرد بازی بلک جک و قواعد آن را درک کنیم.

نحوه بازی بلک جک

بازی بلک جک که با نام بازی ۲۱ نیز شناخته می شود، یکی از روشهای بازی با کارتهای عدد–دار[۲] است. بازی از یک بازیکن و یک پخش‌کننده (بانکدار) تشکیل شده است. قواعد بازی بشرح زیر است:

قاعده اول: طرفی که ارزش مجموع همه کارت‌هایش ۲۱ باشد، برنده است.

قاعده دوم: طرفی که ارزش کارتهایش از ۲۱ کمتر ولی بیشتر از ارزش کارتهای رقیب باشد، برنده است.

قاعده سوم: طرفی که ارزش سرجمع کارتهایش بیشتر از ۲۱ باشد، سوخته است.

قاعده چهارم: اگر ارزش سرجمع کارتهای دو طرف برابر باشد، بازی به تساوی کشیده شده است.

قاعده پنجم: ارزش کارت آس[۳] را دارنده آن مشخص می‌کند که عدد یک یا یازده است.

اگر یکی از این معیارهای برنده شدن برآورده شود، بازیکن ما، برنده بازی است. در غیر این صورت پخش‌کننده، برنده

---

[۱] برای وفاداری به متن اصلی و با انگیزه یادگیری بهتر، این قسمت از فصل نیز علیرغم نوع مثال، ترجمه شده است.

[۲] Dealer

[۳] Ace

بازی است. بیایید این را با جزئیات بیشتر درک کنیم.

برای کارتهای سرباز (**J**)، شاه (**K**) و ملکه (**Q**) مقدار ۱۰ در نظر گرفته می‌شود. مقدار آس (**A**) بسته به انتخاب بازیکن می تواند ۱ یا ۱۱ باشد. یعنی بازیکن می‌تواند در خلال بازی تصمیم بگیرد که آیا مقدار آس عدد یک باشد یا مقدار یازده.

ارزش بقیه کارت ها (یعنی کارتهای با شماره ۲ تا ۱۰) فقط ارزش اسمی آنهاست. به عنوان مثال، ارزش کارت ۲ همان ۲، ارزش کارت ۳ همان ۳ خواهد بود و الی آخر.

این بازی از یک بازیکن و یک پخش‌کن تشکیل شده است. البته می‌تواند بازیکنان زیادی در یک زمان وجود داشته باشند اما فقط یک پخش‌کننده وجود دارد. همه بازیکنان با پخش‌کننده رقابت می‌کنند و نه با بازیکنان دیگر. بیایید موردی را در نظر بگیریم که در آن فقط یک بازیکن و یک پخش‌کن وجود دارد. در اینجا سعی می‌کنیم بازی بلک جک را با انجام بازی در صورتهای مختلف آن درک کنیم. فرض کنید ما، **بازیکن** بوده و در حال رقابت با **پخش‌کن** هستیم.

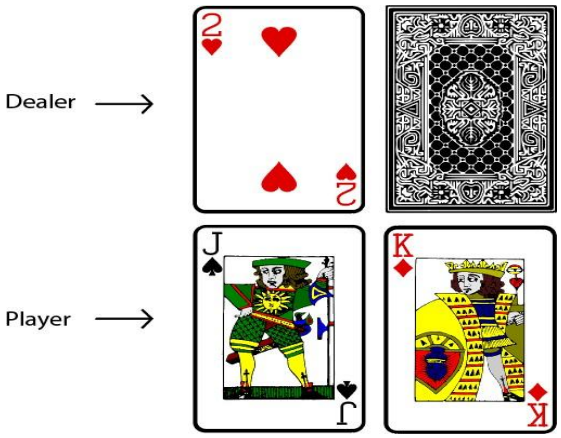
مورد ۱: زمانی که بازیکن برنده بازی است.

در ابتدا به یک بازیکن دو کارت داده می شود. هر دوی این کارتها رو به بالا هستند، یعنی هر دو کارت بازیکن برای پخش‌کن قابل مشاهده است. به همین ترتیب، به پخش‌کن نیز دو کارت داده می‌شود، اما یکی از کارتهای پخش‌کن رو به بالا و دیگری رو به پایین است. یعنی پخش‌کن فقط یکی از کارتهای خود را نشان می‌دهد.

همانطور که در شکل ۴.۷ می بینیم، بازیکن، دو کارت (هر دو رو به بالا) و پخش‌کن نیز دو کارت (فقط یک کارت رو به بالا) دارد.

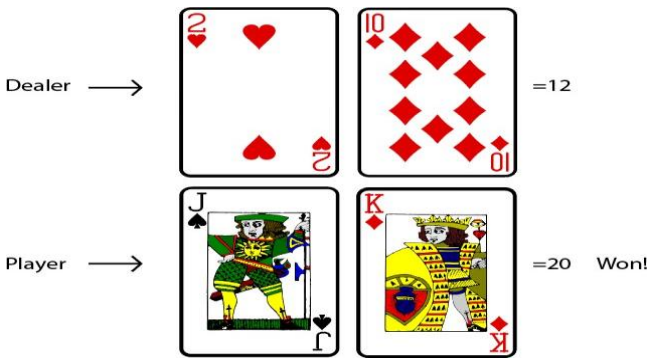
حالا بازیکن یکی از دو عمل ممکن یعنی **تداوم** و **توقف** را انجام می دهد. اگر ما (که بازیکن هستیم) اقدام تداوم را انجام دهیم، یک کارت دیگر دریافت می‌کنیم. ولی اگر ما اقدام توقف را اجرا کنیم، به این معنی است که دیگر به

کارت نیاز نداریم و به پخش‌کن می‌گوئیم که تمام کارت‌های خود را نشان دهد. هر کسی که مجموع ارزش کارتهایش برابر با ۲۱ و یا بیشتر از بازیکن دیگر باشد (حتی اگر ۲۱ نباشد)، برنده بازی است.



شکل ۴.۷: جمع اعداد دو کارت بازیکن ۲۰ است ولی پخش‌کن، یک کارت ۲ و یک کارت رو به پایین دارد.

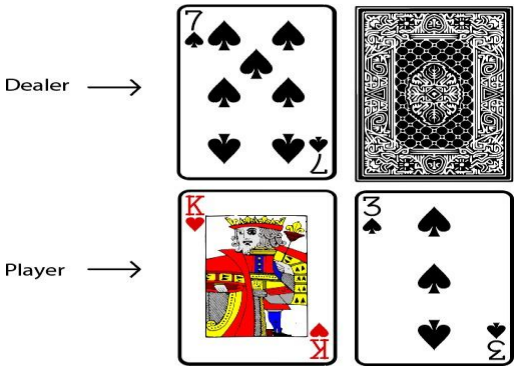
ما آموختیم که ارزش کارتهای **J**، **K** و **Q** برابر ۱۰ است. همانطور که در شکل ۴.۷ نشان داده شده است، بازیکن ما، کارتهای **J** و **K** را دارد که مجموع آنها ۲۰ است (۱۰+۱۰). بنابراین، ارزش کل کارتهای ما در حال حاضر یک عدد بزرگ است گرچه از ۲۱ تجاوز نمی‌کند. به این دلیل ما اقدام توقف را برمی‌گزینیم، و این اقدام به پخش‌کن می‌گوید که کارتهای خود را نشان دهد. همانطور که در شکل ۴.۸ مشاهده می کنیم، پخش‌کن اکنون تمام کارتهای خود را نشان داده است. ارزش کل کارتهای پخش‌کن ۱۲ و ارزش کل کارتهای ما (بازیکن) ۲۰ است که بزرگتر از ۱۲ است و هر چند به ۲۱ نرسیده، اما ما برنده بازی هستیم.



شکل ۴.۸: بازیکن برنده می شود!

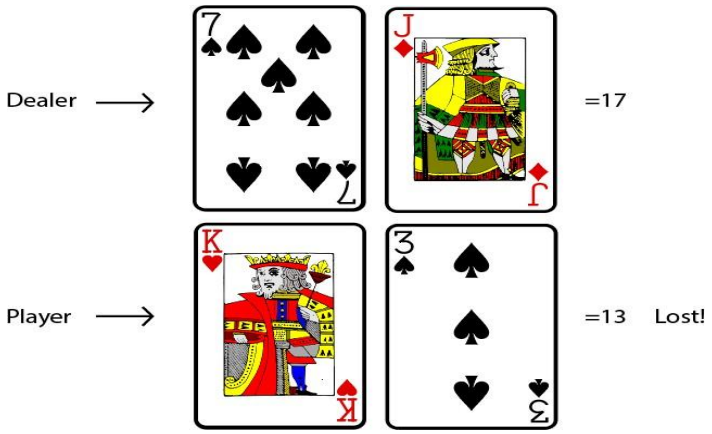
مورد دوم: زمانی که بازیکن بازی را می بازد.

شکل ۴.۹ نشان می‌دهد که ما دو کارت داریم و پخش‌کن نیز دو کارت دارد و تنها یکی از کارتهای پخش‌کن برای ما قابل مشاهده است. حال باید تصمیم بگیریم کدامیک از دو اقدام تداوم و یا توقف را انتخاب کنیم. شکل ۴.۹ نشان می دهد که ما دو کارت **K** و ۳ را داریم که مجموع آنها ۱۳ است (۱۰+۳). بیایید کمی خوشبین باشیم و امیدوار باشیم که ارزش کل کارتهای پخش‌کن از ما بیشتر نباشد. بنابراین ما متوقف شده و انتخاب این اقدام به پخش‌کن می‌گوید که کارتهای خود را نشان دهد.



شکل ۴.۹: بازیکن ۱۳ و فروشنده ۷ با یک کارت رو به پایین دارد.

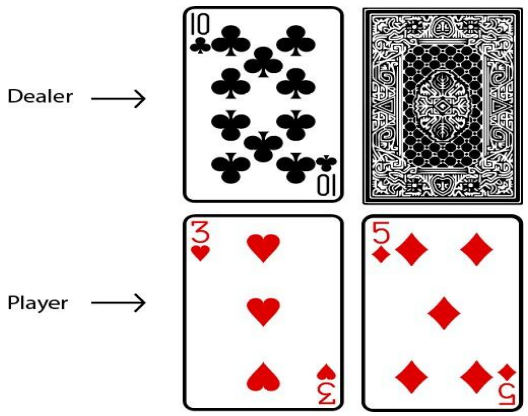
همانطور که در شکل ۴.۱۰ مشاهده می کنیم، مجموع کارت پخش‌کن ۱۷ است، اما جمع کارتهای ما تنها ۱۳ است، بنابراین بازی را باختیم. یعنی کارتهای پخش‌کن ارزش بیشتری از کارتهای ما دارد. گرچه از عدد ۲۱ تجاوز نکرده است، اما بنابه قاعده دوم بازی یعنی سرجمع بیشتر، پس او برنده و ما بازنده‌ایم.



شکل ۴.۱۰: پخش‌کن برنده می شود!

مورد ۳: زمانی که بازیکن می‌سوزد.

شکل ۴.۱۱ نشان می دهد که ما دو کارت داریم و پخش‌کن نیز دو کارت دارد اما تنها یکی از کارتهای پخش‌کن برای ما قابل مشاهده است:

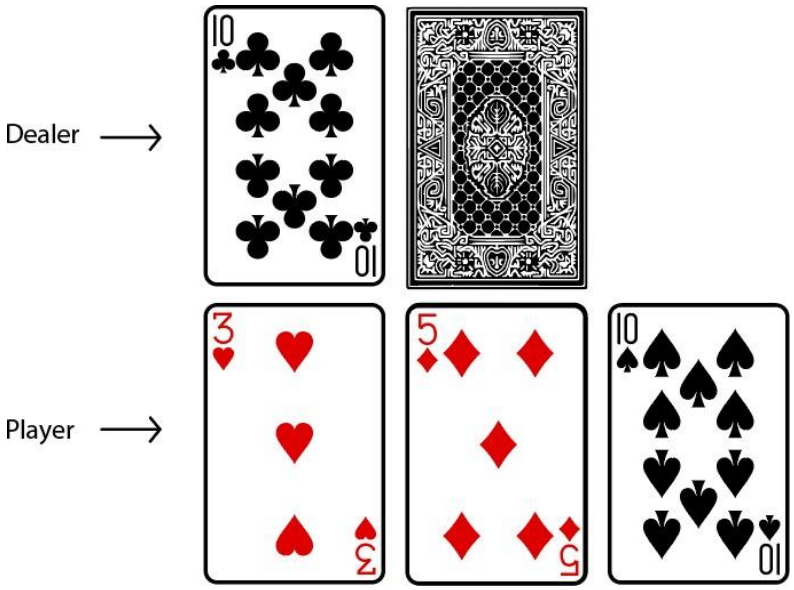


شکل ۴.۱۱: بازیکن دارای سرجمع ۸ و پخش‌کن دارای ۱۰ با یک کارت رو به پایین است.

حال باید تصمیم بگیریم که کدام اقدام را انتخاب کنیم: توقف یا تداوم. ما یاد گرفتیم که هدف بازی این است که مجموع ارزش کارت‌ها ۲۱ باشد، یا مقداری بزرگتر از سرجمع عدد کارتهای پخش‌کن، اگر به ۲۱ نرسیدیم. در حال حاضر، ارزش کل کارت‌های ما فقط ۳+۵ = ۸ است. بنابراین، ما اقدام تداوم را برمی‌گزینیم تا بتوانیم مقدار مجموع

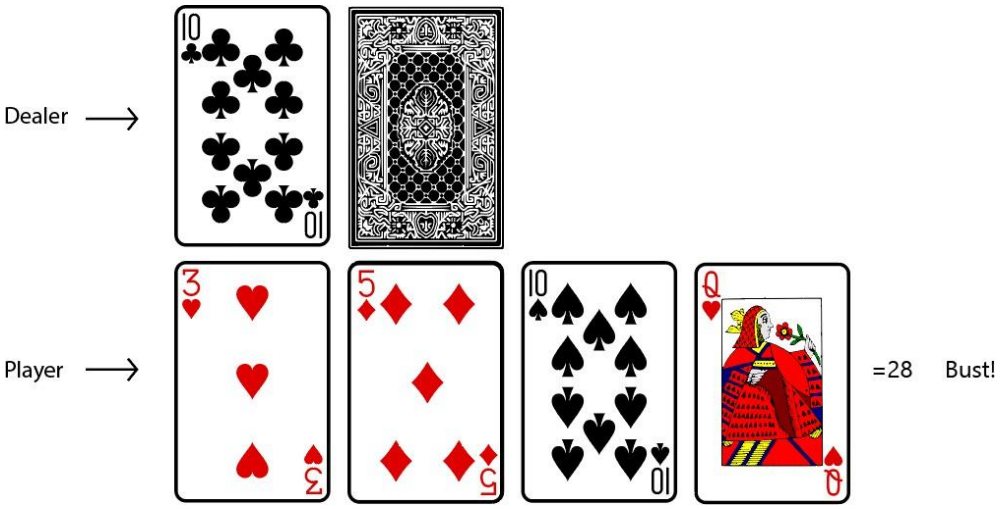
خود را بیشتر کنیم. پس از اعلام تداوم بازی، کارت جدیدی دریافت می نیم که در شکل ۴.۱۲ نشان داده شده است

همانطور که می بینیم، یک کارت جدید دریافت کردیم. در حال حاضر، ارزش کل کارتهای ما ۱۸=۳+۵+۱۰ است. مجدداً باید تصمیم بگیریم که آیا باید ادامه دهیم یا بایستیم. بیایید کمی حریص باشیم و ادامه دهیم تا بتوانیم مقدار جمع خود را کمی بزرگتر کنیم. همانطور که در شکل ۴.۱۳ نشان داده شده است، یک کارت دیگر دریافت کردیم، اما اکنون ارزش کل کارتهای ما می شود ۲۸= ۳+۵+۱۰+۱۰ که از ۲۱ فراتر می رود که در این صورت ما به **فنا** رفتیم (یا **ورشکست** شدیم) و اصطلاحا سوختیم[۱] و لذا نتیجتا بازی را باختیم.



شکل ۴.۱۲: بازیکن جمعا ۱۸ و پخش‌کن یک کارت ۱۰ با یک کارت رو به پایین دارد.

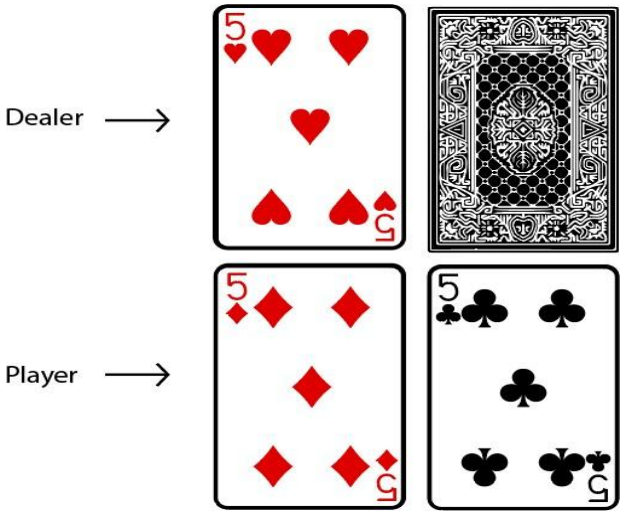
[۱] Bust



شکل ۴.۱۳: بازیکن به فنا رفت!

مورد ٤: بازیکن کارت آس مفید یا قابل استفاده دارد.

ما گفتیم که ارزش آس می تواند ۱ یا ۱۱ باشد و بازیکن می تواند در طول بازی، ارزش کارت آس را تعیین کند. بیایید یاد بگیریم که چگونه این کار می‌کند.

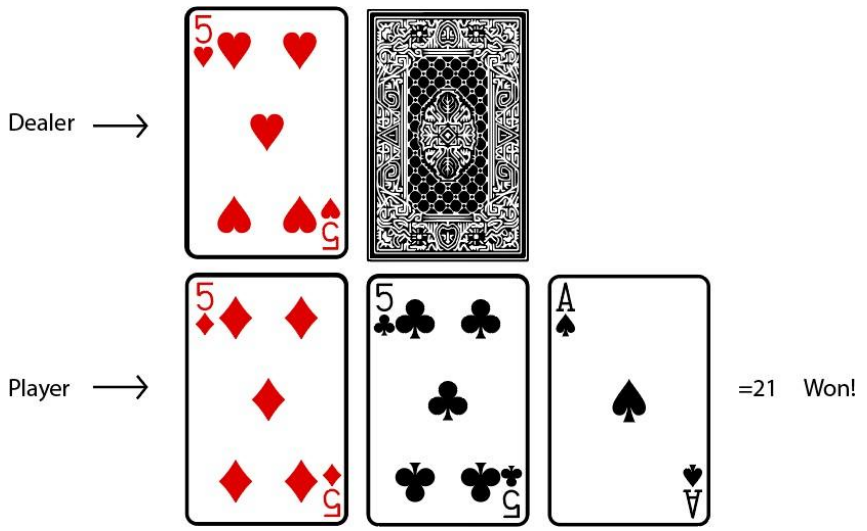


شکل ۴.۱۴: جمع اعداد بازیکن ۱۰ و پخشنده یک کارت با عدد ۵ و یک کارت رو به پایین دارد.

همانطور که شکل ۴.۱۴ نشان می دهد، دو کارت به ما داده شده است و پخش‌کن نیز دو کارت دارد و تنها یکی از کارتهای پخش‌کن رو به بالا است.

در این حالت، ارزش کل کارت های ما ۱۰= ۵+۵ است. بنابراین، ادامه می‌دهیم تا بتوانیم مقدار مجموع خود را بزرگتر کنیم. همانطور که شکل ۴.۱۵ نشان می دهد، پس از انتخاب اقدام تداوم، کارت جدیدی دریافت کردیم که یک آس است. حالا می‌توانیم مقدار آس را ۱ یا ۱۱ در نظر بگیریم. اگر مقدار آس را ۱ در نظر بگیریم، ارزش کل کارت‌های ما ۱۱=۱+۵+۵ خواهد بود. اما اگر مقدار آس را ۱۱ در نظر بگیریم، ارزش کل کارت‌های ما ۲۱= ۵+۵+۱۱ خواهد بود.

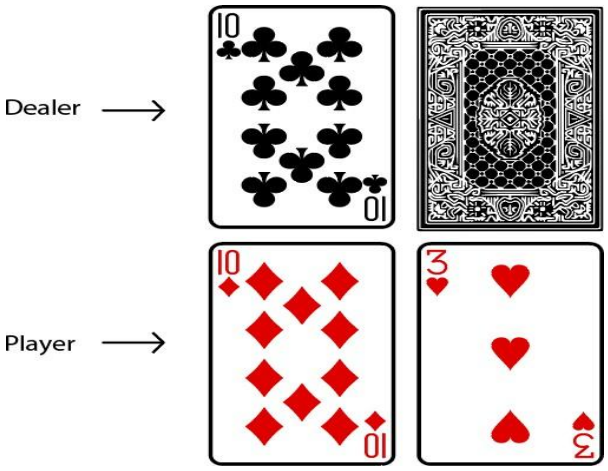
بدین ترتیب مقدار آس را ۱۱ قرار می دهیم و برنده بازی می‌شویم و در این حالت کارت آس را **کارت آس مفید** یا قابل استفاده می‌نامند زیرا به ما کمک کرد تا بازی را ببریم



شکل ۴.۱۵: بازیکن از آس به عنوان عدد ۱۱ استفاده می کند و برنده بازی است.

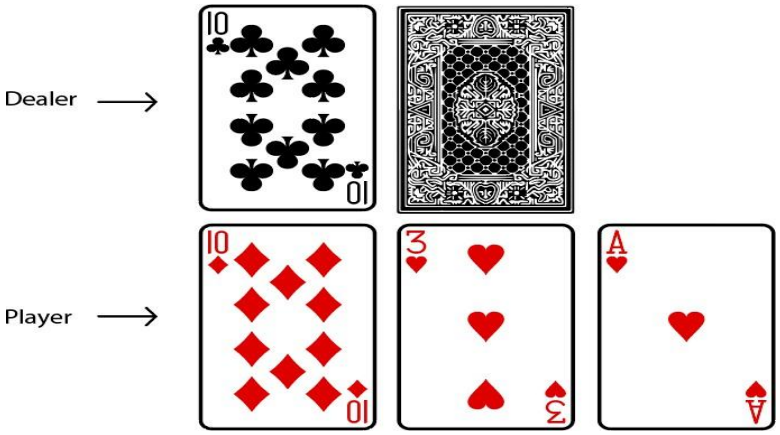
مورد ۵: بازیکن کارت آس نامفید یا غیرقابل استفاده دارد.

شکل ۴.۱۶ نشان می دهد که ما دو کارت داریم و پخش‌کن دو کارت با یک رو به بالا دارد:



شکل ۴.۱۶: بازیکن ۱۳ و فروشنده ۱۰ با یک کارت رو به پایین دارد.

همانطور که می بینیم، ارزش کل کارتهای ما ۱۳ (۱۰+۳) است. ما عمل تدواوم را انتخاب می‌کنیم تا بتوانیم مقدار مجموع خود را کمی بزرگتر کنیم:

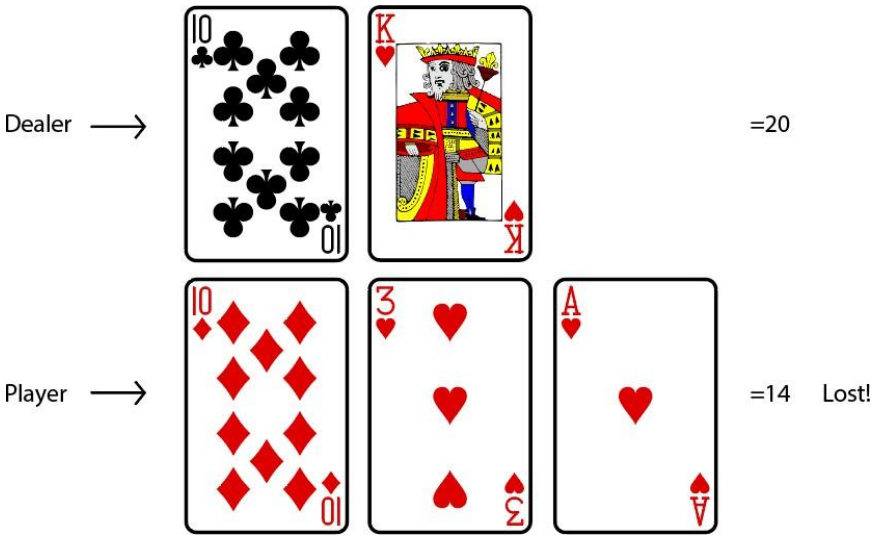


شکل ۴.۱۷: بازیکن باید از آس به عنوان ۱ استفاده کند والا می‌سوزد

همانطور که در شکل ۴.۱۷ نشان داده شده است، ما یک کارت جدید دریافت کردیم که یک Ace است. حالا می‌توانیم مقدار آس را ۱ یا ۱۱ انتخاب کنیم. اگر ۱۱ را انتخاب کنیم، مقدار مجموع ما ۲۳=۱۰+۳+۱۱ می‌شود؛ یعنی وقتی آس خود را روی ۱۱ قرار می‌دهیم، مقدار مجموع ما از ۲۱ بیشتر می‌شود و بازی را می‌بازیم. لذا، به جای انتخاب $Ace = 11$

۱۱ مقدار Ace را برابر ۱ قرار می دهیم. بنابراین مقدار جمع ما ۱۴= ۱۰+۳+۱ می‌شود.

مجدداً باید تصمیم بگیریم که آیا باید برویم یا بایستیم. بیایید بگوییم ما امیدواریم که ارزش مجموع پخش‌کن کمتر از ارزش ما باشد. همانطور که شکل ۴.۱۸ نشان می‌دهد، پس از انجام عمل ایستادن یا توقف، هر دو کارت پخش‌کن نشان داده می‌شود که مجموع کارتهای او ۲۰ است، اما جمع کارتهای ما فقط ۱۴ است و بنابراین بازی را می‌بازیم. در این حالت از کارت آس به عنوان **آس نامفید** یا غیرقابل استفاده یاد می‌کنند زیرا به ما کمکی برای برنده شدن در بازی نمی‌کند.



شکل ۴.۱۸: بازیکن ۱۴ دارد و پخش‌کن۲۰ و لذا و برنده می‌شود.

مورد ششم: زمانی که بازی مساوی شود

اگر مجموع ارزش کارتهای بازیکن و پخش‌کننده یکسان باشد، مثلاً ۲۰، آنگاه می‌گویند بازی به تساوی کشیده شد.

اکنون که نحوه بازی بلک جک را فهمیدیم، بیایید روش پیش بینی مونت کارلو را در بازی بلک جک پیاده‌سازی کنیم. اما قبل از شروع، ابتدا بیاموزیم که چگونه محیط بلک جک در Gym طراحی شده است.

محیط بلك جك در كتابخانه Gym

كتابخانه Gym را وارد كنید:

```
import gym
```

شناسه محیط بلک جک `Blackjack-v0` است. بنابراین، ما می‌توانیم بازی بلک جک را با استفاده از تابع `make` به شرح زیر، ایجاد کنیم:

```
env = gym.make('Blackjack-v0')
```

اکنون، بیایید به حالت در محیط بلک جک نگاه کنیم. ما فقط می توانیم محیط خود را بازتنظیم کرده و آنرا در حالت اولیه قرار دهیم:

```
print(env.reset())
```

توجه داشته باشید که هر بار که کد قبلی را اجرا می کنیم، ممکن است نتیجه متفاوتی بگیریم، زیرا حالت اولیه به طور تصادفی مقداردهی اولیه می شود. کد قبلی چیزی شبیه به این را چاپ می کند:

```
(15, 9, True)
```

همانطور که مشاهده می کنیم ، حالت ما به صورت یک چندتایی نشان داده می شود ، اما این به چه معناست؟ ما یاد گرفتیم که در بازی بلک جک، دو کارت به ما داده می‌شود و همچنین می‌توانیم یکی از کارتهای پخش‌کن را ببینیم. بنابراین، ۱۵ به این معنی است که ارزش مجموع کارتهای ماست و ۹ به منزله ارزش اسمی یکی از کارتهای پخش‌کن است، `True` به این معناست که ما یک آس قابل استفاده داریم، و اگر آس قابل استفاده نداشته باشیم، `False` را خواهیم داشت.

بنابراین ، در محیط بلک جک ، حالت به صورت یک چندتایی متشکل از سه مقدار نشان داده می شود:

۱. ارزش مجموع کارتهای ما
۲. ارزش اسمی یکی از کارتهای پخش‌کن
۳. ارزش بولین: `True` یعنی داشتن آس مفید و `False` یعنی نداشتن آس مفید

بیایید به فضای اقدامات محیط بلک جک خود نگاه کنیم:

```
print(env.action_space)
```

کد قبلی برای ما چاپ می‌کند:

```
Discrete(2)
```

همانطور که می توانیم مشاهده کنیم ، این بدان معناست که ما دو عمل یا اقدام در فضای عمل خود داریم که ۰ و ۱ هستند:

- اقدام **توقف** با ۰ نشان داده می‌شود
- اقدام **تداوم** با ۱ نشان داده می‌شود

خب، در مورد پاداش چطور؟ پاداش به شرح زیر اختصاص داده می شود:

- ✓ **۱.۰+** مقدار پاداش، در صورت برنده شدن در بازی
- ✓ **۱.۰−** مقدار پاداش، در صورت باخت بازی
- ✓ **۰** مقدار پاداش، در صورت تساوی بازی

اکنون که متوجه شدیم محیط بلک جک در جیم چگونه طراحی شده است، بیایید پیاده‌سازی روش پیش‌بینی **MC** را در بازی بلک جک شروع کنیم. ابتدا سراغ روش **MC** هر–بازدید می‌رویم و سپس یاد می‌گیریم که چگونه پیش‌بینی **MC** اولین–بازدید را پیاده‌سازی کنیم.

روش پیش بینی MC هر_بازدید برای بازی بلک جک

برای درک واضح این بخش، باید روش مونت کارلو را که قبلا یاد گرفتیم، مرور کنید. بیایید اکنون بفهمیم که چگونه پیش‌بینی **MC** هر بازدید را با بازی بلک جک، گام به گام پیاده‌سازی کنیم:

کتابخانه‌های لازم را وارد کنید:

```
import gym
import pandas as pd
from collections import defaultdict
```

یک محیط بلک جک ایجاد کنید:

```
env = gym.make('Blackjack-v0')
```

تعریف یک خطمشی

ما یاد گرفتیم که در روش پیش‌بینی، یک **خطمشی** ورودی به ما داده می شود و **تابع ارزش** سیاست ورودی (داده شده) را پیش‌بینی می‌کنیم. بنابراین، اکنون، ابتدا یک تابع خطمشی را تعریف می‌کنیم تا به عنوان یک سیاست ورودی عمل کند. یعنی ما **سیاست ورودی** را تعریف می‌کنیم تا **تابع ارزش** آن در مراحل آتی پیش‌بینی شود.

همانطور که در کد زیر نشان داده شده است، تابع سیاست ما حالت را به عنوان ورودی می‌گیرد. حال اگر `state[0]`، یعنی مجموع کارتهای ما، بزرگتر از ۱۹ باشد، عمل ۰ (ایستاده یا توقف) را برمی‌گرداند، در غیر این صورت اقدام ۱ را (یعنی ادامه یا تداوم) برمی گرداند:

```
def policy(state):
    return 0 if state[0] > 19 else 1
```

ما یک سیاست بهینه تعریف کردیم: انجام یک عمل ۰ (ایستادن یا توقف) زمانی که مقدار مجموع ما در حال حاضر

بیشتر از ۱۹ است، که کاملا منطقی است. یعنی وقتی مقدار مجموع بیشتر از ۱۹ باشد، مجبور نیستیم یک عمل ۱ (ادامه یا تداوم) انجام دهیم و یک کارت جدید دریافت کنیم، که ممکن است باعث باخت بازی یا سوختن ما شود.

به عنوان مثال، بیایید با تنظیم مجدد محیط، همانطور که به شرح زیر نشان داده شده است، یک حالت اولیه ایجاد کنیم:

```
state = env.reset()
print(state)
```

فرض کنید کد قبلی، موارد زیر را چاپ می کند:

```
(20, 5, False)
```

همانطور که می توانیم متوجه شویم `state[0]=20` یعنی مقدار مجموع کارتهای ما ۲۰ است، بنابراین در این صورت، خط مشی ما عمل • (ایستادن) را همانطور که در زیر نشان می دهد، برمی گرداند:

```
print(policy(state))
```

بر اساس کد قبلی، مقدار زیر چاپ می شود:

```
0
```

حال که سیاست را تعریف کردیم، در بخ های بعدی تابع ارزش (مقادیر حالت) این سیاست را پیش‌بینی خواهیم کرد.

تولید یك پردینه

در مرحله بعد، ما با استفاده از خطمشی داده شده یک پردینه ایجاد می کنیم، بنابراین تابعی به نام

generate_episode تعریف می‌کنیم که سیاست را به عنوان ورودی می‌گیرد و با استفاده از خطمشی داده شده، اپیز پردینه ود را تولید می‌کند. ابتدا، بیایید تعداد مراحل زمانی را تنظیم کنیم:

```
num_timesteps = 100
```

برای درک واضح، بیایید به تابع خط به خط نگاه کنیم:

```
def generate_episode(policy):
```

بیایید لیستی به نام episode را برای ذخیره پردینه (اپیزود) تعریف کنیم:

```
    episode = []
```

با تنظیم مجدد محیط، حالت را مقداردهی اولیه کنید:

```
    state = env.reset()
```

سپس برای هر مرحله زمانی:

```
    for t in range(num_timesteps):
```

اقدام را طبق خطمشی داده شده انتخاب کنید:

```
        action = policy(state)
```

اقدام را انجام دهید و اطلاعات وضعیت بعدی را ذخیره کنید:

```
        next_state, reward, done, info = env.step(action)
```

حالت، اقدام و پاداش را در فهرست پردینه (اپیزود) ما ذخیره کنید:

```
            episode.append((state, action, reward))
```

اگر حالت بعدی یک حالت نهایی است، حلقه را بشکنید، در غیر این صورت حالت بعدی را به عنوان وضعیت فعلی بروزرسانی کنید:

```
        if done:
            break

        state = next_state

    return episode
```

بیایید نگاهی بیندازیم که خروجی تابع `generate_episode` ما چگونه است. توجه داشته باشید که ما با استفاده از خطمشی که قبلا تعریف کردیم یک پردینه ایجاد می‌کنیم:

```
print(generate_episode(policy))
```

کد قبلی ما، چیزی شبیه به زیر را چاپ می کند:

```
[((10, 2, False), 1, 0), ((20, 2, False), 0, 1.0)]
```

همانطور که می‌بینیم خروجی ما به شکل **[(حالت، اقدام، پاداش)]** است. همانطور که قبلا نشان داده شد، ما در پردینه خود دو حالت داریم. ما اقدام ۱ (تداوم) را در حالت `(10, 2, False)` انجام دادیم و ۰ پاداش دریافت کردیم و عمل ۰ (توقف) را در حالت `(20, 2, False)` انجام دادیم و پاداش ۱.۰ دریافت کردیم.

اکنون که یاد گرفتیم چگونه با استفاده از سیاست داده شده یک اپیزود تولید کنیم، در مرحله بعد، نحوه محاسبه ارزش حالت (تابع ارزش) را با استفاده از روش **MC** هر-بازدید بررسی خواهیم کرد.

محاسبه تابع ارزش

ما یاد گرفتیم که به منظور پیش‌بینی **تابع ارزش**، چندین اپیزود را با استفاده از **سیاست** داده شده تولید کنیم و ارزش حالت را به عنوان بازده متوسط در چندین پردینه (اپیزود) محاسبه کنیم. بیایید حال ببینیم چگونه آن را پیاده‌سازی کنیم.

ابتدا، ما `total_return` و `N` را به عنوان یک فرهنگ لغت برای ذخیره **بازده کل** و تعداد دفعات بازدید از حالت در اپیزودها به ترتیب تعریف می‌کنیم:

```
total_return = defaultdict(float)
N = defaultdict(int)
```

تعداد تکرارها را تنظیم کنید، یعنی تعداد اپیزودهایی را که می خواهیم تولید کنیم:

```
num_iterations = 500000
```

سپس، برای هر تکرار:

```
for i in range(num_iterations):
```

پردینه را با استفاده از سیاست داده شده ایجاد کنید. یعنی با استفاده از تابع خطمشی که قبلا تعریف کردیم:

```
    episode = generate_episode(policy)
```

تمام حالت ها، اقدامات و پاداش به دست آمده از پردینه را ذخیره کنید:

```
    states, actions, rewards = zip(*episode)
```

سپس، برای هر مرحله از پردینه:

```
    for t, state in enumerate(states):
```

بازگشت یا **بازده** R هر حالت را به عنوان مجموع پاداشها محاسبه کنید، $R(s_t)=sum(rewards[t:])$:

```
        R = (sum(rewards[t:]))
```

مقدار `total_return` برای state را به صورت $total\_return(s_t) = total\_return(s_t) + R(s_t)$ به‌روز کنید:

```
        total_return[state] =  total_return[state] + R
```

تعداد دفعات بازدید از حالت در پردینه را به صورت $N(s_t) = N(s_t) + 1$ به روز کنید:

```
        N[state] =  N[state] + 1
```

پس از محاسبه `total_return` و N فقط می‌توانیم آنها را به یک فریم داده‌ای پاندا[1] تبدیل کنیم تا درک بهتری داشته باشیم. توجه داشته باشید که این فقط برای ارائه درک روشنی از الگوریتم است. ما لزوما مجبور نیستیم خروجی خود را به فریم داده‌ای پاندا تبدیل کنیم، ما همچنین می‌توانیم آنرا را به طور موثر فقط با استفاده از فرهنگ لغت پیاده‌سازی کنیم.

فرهنگ لغت `total_return` را به یک دیتافریم تبدیل کنید:

```
total_return = pd.DataFrame(total_return.items(),columns=['state',
'total_return'])
```

فرهنگ لغت شمارنده N را به یک دیتافریم تبدیل کنید:

[1] Pandas Data

```
N = pd.DataFrame(N.items(),columns=['state', 'N'])
```

دو دیتافریم را در حالتها ادغام کنید:

```
df = pd.merge(total_return, N, on="state")
```

به چند ردیف اول دیتافریم نگاه کنید:

```
df.head(10)
```

کد قبلی موارد زیر را نمایش می دهد. همانطور که می توانیم مشاهده کنیم، ما بازده کل و تعداد دفعات بازدید از حالت را داریم.

	state	total_return	N
۰	(7, 10, False)	-۵۲	۹۸
۱	(11, 10, False)	٦	۱۹۰
۲	(15, 10, False)	-۲۱۹	۳۹٤
۳	(12, 10, False)	-۱٦۰	۳۳۱
٤	(18, 10, False)	-۲٦۹	٤۰٦
٥	(14, 10, True)	-۲۱	٤۹
٦	(21, 10, True)	۱۷٦	۱۹۳
۷	(16, 4, False)	-٥٥	۸۳
۸	(10, 10, False)	-۲۳	۱٥۰
۹	(20, 10, False)	۲۷٥	٥۸٥

شکل ۴.۱۹: بازده کل و تعداد دفعاتی که از یک حالت بازدید شده است.

در مرحله بعد، می توانیم مقدار حالت را به عنوان بازده متوسط محاسبه کنیم:

$$V(s) = \frac{\text{total\_return}(s)}{N(s)}$$

سپس می‌توانیم بنویسیم:

```
df['value'] = df['total_return']/df['N']
```

بیایید به چند ردیف اول دیتافریم نگاه کنیم:

```
df.head(10)
```

کد قبلی چیزی شبیه به این را نمایش می‌دهد:

	state	total_return	N	value
۰	(7, 10, False)	-۵۲,۰	۹۸	-۰,۵۳۰۶۱۲
۱	(11, 10, False)	۶,۰	۱۹۰	۰,۰۳۱۵۷۹
۲	(15, 10, False)	-۲۱۹,۰	۳۹۴	-۰,۵۵۵۸۳۸
۳	(12, 10, False)	-۱۶۰,۰	۳۳۱	-۰,۴۸۳۳۸۴
۴	(18, 10, False)	-۲۶۹,۰	۴۰۶	-۰,۶۶۲۵۶۲
۵	(14, 10, True)	-۲۱,۰	۴۹	-۰,۴۲۸۵۷۱
۶	(21, 10, True)	۱۷۶,۰	۱۹۳	۰,۹۱۱۹۱۷
۷	(16, 4, False)	-۵۵,۰	۸۳	-۰,۶۶۲۶۵۱
۸	(10, 10, False)	-۲۳,۰	۱۵۰	-۰,۱۵۳۳۳۳
۹	(20, 10, False)	۲۷۵,۰	۵۸۵	۰,۴۷۰۰۸۵

شکل ۴.۲۰: ارزش به عنوان میانگین بازده هر حالت محاسبه می‌شود.

همانطور که می توانیم مشاهده کنیم، اکنون ارزش حالت را داریم که فقط میانگین بازگشت حالت در چندین پردینه

است. بنابراین، با استفاده از روش **MC** هر بار بازدید، تابع ارزش سیاست مورد نظر را با موفقیت پیش‌بینی کرده‌ایم.

بسیار خب، بیایید مقدار برخی از حالتها را بررسی کنیم و بفهمیم که تابع ارزش ما با توجه به سیاست داده شده چقدر دقیق تخمین زده می‌شود. به یاد دارید که وقتی شروع به کار کردیم، برای تولید اپیزودها، از خطمشی بهینه استفاده کردیم، که عمل ۰ (ایستادن) را زمانی که مقدار مجموع بزرگتر از ۱۹ است و عمل ۱ (ادامه) را زمانی که مقدار مجموع کمتر از ۱۹ است، انتخاب می‌کند.

بیایید مقدار حالت `(21,9,False)` را ارزیابی کنیم، همانطور که مشاهده می‌کنیم، مقدار مجموع کارتهای ما در حال حاضر ۲۱ است و بنابراین این حالت خوبی است و باید مقدار بالایی داشته باشد. بیایید ببینیم ارزش تخمینی ما از حالت چقدر است:

```
df[df['state']==(21,9,False)]['value'].values
```

کد قبلی چیزی شبیه به زیر را چاپ خواهد کرد:

```
array([1.0])
```

همانطور که مشاهده می کنیم، ارزش حالت بالا است.

حال، بیایید مقدار حالت `(5,8,False)` را بررسی کنیم. همانطور که متوجه می‌شویم، ارزش مجموع کارتهای ما فقط ۵ است و حتی تک کارت پخش‌کن دارای ارزش بالا، یعنی ۸ است. در این مورد، ارزش حالت باید کمتر باشد. بیایید ببینیم ارزش تخمینی ما از حالت چقدر است:

```
df[df['state']==(5,8,False)]['value'].values
```

کد قبلی چیزی شبیه به این را چاپ می کند:

```
array([-1.0])
```

همانطور که می بینیم، ارزش حالت کمتر است.

بنابراین، ما یاد گرفتیم که چگونه با استفاده از روش پیش بینی **MC** هر بار بازدید، تابع ارزشی سیاست مورد نظر را پیش‌بینی کنیم. در بخش بعدی، نحوه محاسبه مقدار حالت با استفاده از روش **MC** بازدید اول را بررسی خواهیم کرد.

روش پیش‌بینی MC اولین_بازدید برای بازی بلک جک

پیش‌بینی ***تابع ارزش*** با استفاده از روش **MC** ***بازدید اول*** دقیقا مشابه نحوه پیش‌بینی ***تابع ارزش*** با استفاده از روش **MC** ***هر بازدید*** است، با این تفاوت که در اینجا بازگشت یا بازده یک حالت را فقط برای اولین بار وقوع آن در پردینه محاسبه می‌کنیم. کد **MC** برای اولین بازدید همان چیزی است که در **MC** هر بازدید دیده‌ایم، به جز اینکه در اینجا، ما بازده را فقط برای اولین بار وقوع آن (همانطور که نشان داده شده است) محاسبه می‌کنیم:

```
for i in range(num_iterations):

    episode = generate_episode(env,policy)
    states, actions, rewards = zip(*episode)

    for t, state in enumerate(states):
        if state not in states[0:t]:
```

```
            R = (sum(rewards[t:]))
            total_return[state] = total_return[state] + R
            N[state] = N[state] + 1
```

شما می‌توانید کد کامل را از مخزن GitHub کتاب دریافت کنید و نتایجی مشابه آنچه در بخش **MC** هر بار بازدید دیدیم دریافت خواهید کرد.

بنابراین، ما یاد گرفتیم که چگونه تابع ارزشی سیاست داده شده را با استفاده از روشهای **MC** اولین بازدید و هر بار بازدید پیش بینی کنیم.

به‌روزرسانی میانگین افزایشی[۱]

در هر دو روش مونت‌کارلوی اولین–بازدید و مونت‌کارلوی هر–بازدید، ارزش یک حالت را به عنوان میانگین (میانگین حسابی) بازگشت حالت در چندین پردینه برآورد می‌کنیم که به شرح زیر نشان داده شده است:

$$V(s) = \frac{\text{total\_return}(s)}{N(s)}$$

به جای استفاده از **میانگین حسابی** برای تقریب ارزش حالت، می‌توانیم از **میانگین افزایشی** نیز استفاده کنیم که به صورت زیر بیان می‌شود:

$$N(s_t) = N(s_t) + 1$$

$$V(s_t) = V(s_t) + \frac{1}{N(s_t)}(R_t - V(s_t))$$

اما چرا به میانگین افزایشی نیاز داریم؟ محیط را غیر ایستا[۲] در نظر بگیرید. در این صورت، لازم نیست که بازگشت حالت تمام پردینه‌ها را محاسبه کنیم و میانگین بگیریم. از آنجایی که محیط غیرایستا است، می‌توانیم بازده اپیزودهای قبلی را نادیده بگیریم و فقط از بازده آخرین پردینه‌ها برای محاسبه میانگین استفاده کنیم. بنابراین، می‌توانیم ارزش حالت را با استفاده از میانگین افزایشی به صورت زیر محاسبه کنیم:

$$V(s_t) = V(s_t) + \alpha\big(R_t - V(s_t)\big)$$

که درآن $\alpha = \frac{1}{N(s_t)}$ و R_t برابر بازگشت حالت s_t است.

[۱] Incremental Mean Updates
[۲] Non-Stationary

پیش‌بینی مونت‌کارلو برای تابع Q

تا اینجا یاد گرفتیم که چگونه **تابع ارزش** را برای **خط‌مشی** داده شده و با استفاده از روش مونت کارلو، پیش‌بینی کنیم. در این قسمت نحوه پیش‌بینی **تابع** Q را برای **خط‌مشی** داده شده و با استفاده از روش مونت کارلو خواهیم دید.

پیش‌بینی **تابع** Q برای **خط‌مشی** داده‌شده با استفاده از روش مونت کارلو دقیقاً مشابه نحوه پیش‌بینی **تابع ارزش** در بخش قبل است، با این تفاوت که در اینجا از **بازده زوج** (حالت–اقدام) استفاده می‌کنیم، در حالی که در مورد تابع ارزش، ما از **بازده حالت**، استفاده کردیم. یعنی، درست همانطور که ارزش یک حالت (تابع ارزش) را با محاسبه میانگین بازده حالت در چندین اپیزود تقریب زدیم، می‌توانیم ارزش یک زوج حالت–اقدام (تابع Q) را نیز با محاسبه میانگین بازگشت زوج حالت–اقدام در چندین اپیزود، تخمین بزنیم.

بنابراین، ما چندین پردینه (اپیزود) را با استفاده از **سیاست** داده شده π تولید می‌کنیم، سپس، $\text{total\_return}(s,a)$، مجموع **بازگشت** زوج حالت–اقدام در چندین اپیزود را محاسبه می‌کنیم، و همچنین $N(s,a)$، که بیانگر تعداد دفعاتی است که زوج حالت–اقدام در چندین پردینه بازدید شده است را محاسبه می‌کنیم. سپس تابع Q یا ارزش Q را به عنوان میانگین بازگشت زوج حالت–اقدام محاسبه می‌کنیم که به صورت زیر نشان داده شده است:

$$Q(s,a) = \frac{\text{total\_return}(s,a)}{N(s,a)}$$

یک مثال کوچک را در نظر بگیرید. فرض کنید ما دو حالت s_0 و s_1 و دو اقدام ممکن ۰ و ۱ را داریم. اکنون $\text{total\_return}(s,a)$ و $N(s,a)$ را محاسبه می‌کنیم. فرض کنید جدول ما بعد از محاسبه، شبیه جدول ۴.۴ است. وقتی مقادیر را داشته باشیم، می‌توانیم ارزش Q را فقط با گرفتن میانگین محاسبه کنیم، یعنی:

$$Q(s,a) = \frac{\text{total\_return}(s,a)}{N(s,a)}$$

حالت	اقدام	total_returns(s, a)	$N(s, a)$
s_0	۰	۴	۲
s_0	۱	۲	۲
s_1	۰	۲	۲
s_1	۱	۲	۱

جدول ۴.۴: نتیجه دو اقدام در دو حالت

بنابراین، می‌توانیم ارزش Q را برای همه زوج‌های حالت–اقدام به صورت زیر محاسبه کنیم:

$$Q(s_{.}, ۰) = \frac{\text{total\_return}(s_{.}, ۰)}{N(s_{.}, ۰)} = \frac{۴}{۲} = ۲$$

$$Q(s_{.}, ۱) = \frac{\text{total\_return}(s_{.}, ۱)}{N(s_{.}, ۱)} = \frac{۲}{۲} = ۱$$

$$Q(s_{۱}, ۰) = \frac{\text{total\_return}(s_{۱}, ۰)}{N(s_{۱}, ۰)} = \frac{۲}{۲} = ۱$$

$$Q(s_{۱}, ۱) = \frac{\text{total\_return}(s_{۱}, ۱)}{N(s_{۱}, ۱)} = \frac{۲}{۱} = ۲$$

الگوریتم پیش‌بینی **تابع** Q با استفاده از بازده زوج حالت–اقدام در روش مونت کارلو به شرح زیر است. همانطور که می‌بینیم، دقیقاً مشابه نحوه پیش‌بینی **تابع ارزش** با استفاده از بازگشت حالت در روش مونت کارلو است، با این تفاوت که در اینجا تابع Q را با استفاده از بازگشت یک زوج حالت–اقدام پیش‌بینی می‌کنیم:

۱. فرض کنید $\text{total\_return}(s, a)$ مجموع **بازده** یک زوج حالت–اقدام در چندین پردینه و $N(s, a)$ تعداد دفعاتی باشد که یک زوج حالت–اقدام در چندین پردینه بازدید شده است. مقدار $\text{total\_return}(s, a)$ و $N(s, a)$ را برای همه زوج‌های state-action برابر با صفر مقداردهی کنید. **خط‌مشی** π به عنوان ورودی، داده می‌شود.

۲. برای M تعداد تکرار:

۱. یک پردینه با استفاده از **خط‌مشی** π ایجاد کنید.

۲. تمام **پاداش‌های** به دست آمده در پردینه را در لیستی به نام **پاداش‌ها** ذخیره کنید.

۳. برای هر مرحله t در پردینه:

۱. بازگشت را برای زوج حالت–اقدام محاسبه کنید،

$$R(s_t, a_t) = \text{sum}(\text{rewards}[t:])$$

۲. بازگشت (بازده) کل زوج حالت–اقدام را به روز کنید،

$$\text{total\_return}(s_t, a_t) = \text{total\_return}(s_t, a_t) + R(s_t, a_t)$$

۳. شمارنده را به صورت $N(s_t, a_t) = N(s_t, a_t) + 1$ به‌روز کنید.

۳. **تابع** Q (**ارزش** Q) را فقط با گرفتن میانگین محاسبه کنید، یعنی:

$$Q(s, a) = \frac{\text{total\_return}(s, a)}{N(s, a)}$$

به یاد بیاورید که در روش **پیش‌بینی مونت‌کارلوی تابع ارزش**، دو نوع مونت‌کارلو را یاد گرفتیم: ***مونت‌کارلوی اولین بازدید*** و ***مونت‌کارلوی هر بازدید***. در مونت‌کارلوی اولین بازدید، بازگشت حالت را فقط برای اولین باری که از حالت در پردینه بازدید می‌شود، محاسبه می‌کنیم و در مونت‌کارلوی هربازدید، بازگشت حالت را هر بار که حالت در پردینه بازدید می‌شود، محاسبه می‌کنیم.

به طور مشابه، در روش **پیش‌بینی مونت‌کارلوی تابع** Q، دو نوع مونت‌کارلو داریم: ***مونت‌کارلوی اولین بازدید*** و ***مونت‌کارلوی هر بازدید***. در مونت‌کارلوی اولین بازدید، بازگشت زوج حالت–اقدام را فقط برای اولین باری که زوج حالت–اقدام در پردینه بازدید می‌شوند و در مونت‌کارلوی هر بازدید، بازگشت زوج حالت–اقدام را هر بار که این زوج حالت–اقدام، در پردینه بازدید می‌شوند، محاسبه می‌کنیم.

همانطور که در قسمت قبل گفته شد، به جای استفاده از **میانگین حسابی**[۱]، می توانیم از **میانگین تدریجی یا افزایشی**[۲] نیز استفاده کنیم. می‌دانیم که ارزش یک حالت را می‌توان با استفاده از میانگین افزایشی به صورت زیر محاسبه کرد:

$$V(s_t) = V(s_t) + \alpha\big(R_t - V(s_t)\big)$$

به طور مشابه، ما همچنین می‌توانیم ارزش Q را با استفاده از میانگین افزایشی به صورت زیر محاسبه کنیم:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big(R_t - Q(s_t, a_t)\big)$$

حال که نحوه انجام **وظیفه پیش‌بینی را با روش مونت‌کارلو** آموختیم، در قسمت بعدی، نحوه انجام **وظیفه کنترلی را با روش مونت کارلو** خواهیم آموخت.

کنترل (وارسی) مونت کارلو

در **وظیفه کنترل** یا **وظیفه وارسی**، هدف ما یافتن **خط‌مشی** بهینه است. برخلاف **وظیفه پیش‌بینی**، در اینجا، هیچ سیاستی به عنوان ورودی به ما داده نخواهد شد. بنابراین، ما با مقداردهی اولیه یک **خط‌مشی** تصادفی (بختکی) شروع می‌کنیم، و سپس سعی می‌کنیم خط‌مشی بهینه را به صورت تکرارشونده پیدا کنیم. یعنی سعی می‌کنیم، **سیاست** بهینه‌ای پیدا کنیم که حداکثر **بازدهی** را داشته باشد. در این قسمت نحوه انجام وظیفه کنترل برای یافتن خط‌مشی بهینه با استفاده از روش مونت کارلو را یاد می‌گیریم.

خوب، ما یاد گرفتیم که در وظیفه **کنترل**، هدف ما یافتن **خط‌مشی** بهینه است. اول، چگونه می‌توانیم یک **سیاست** را محاسبه کنیم؟ ما آموختیم که این **خط‌مشی** را می‌توان از **تابع** Q استخراج کرد. یعنی اگر **تابع** Q را داشته باشیم، می‌توانیم با انتخاب یک اقدام، در هر حالت که حداکثر **ارزش** Q را داشته باشد، سیاست استخراج کنیم.

---

[۱] Arithmetic Mean
[۲] Incremental Mean

$$\pi = \arg\max_{a} Q(s, a)$$

بنابراین، برای محاسبه یک **سیاست**، باید **تابع** Q را محاسبه کنیم. اما چگونه می‌توانیم **تابع** Q را محاسبه کنیم؟ می‌توانیم **تابع** Q را مشابه آنچه در **روش پیش‌بینی مونت کارلو** آموختیم محاسبه کنیم. یعنی در روش پیش‌بینی مونت‌کارلو، یاد گرفتیم که وقتی یک خطمشی به آن داده می‌شود، می‌توانیم چندین پردینه را با استفاده از آن خطمشی تولید کنیم و **تابع Q** (ارزش Q) را به عنوان میانگین بازگشت زوج حالت–اقدام، در چندین پردینه، محاسبه کنیم.

ما می توانیم مشابه همین گام را در اینجا برای محاسبه **تابع** Q انجام دهیم. اما در روش **کنترل**، هیچ **سیاستی** به عنوان ورودی به ما داده نمی‌شود. بنابراین، ما یک **خطمشی** تصادفی (دلبخواه) را برای مقدار اولیه درنظر می‌گیریم و سپس **تابع** Q را با استفاده از **خطمشی** تصادفی (بختکی) محاسبه می‌کنیم. یعنی درست همانطور که در روش پیش‌بینی یاد گرفتیم، چندین اپیزود را با استفاده از خطمشی تصادفی خود تولید می‌کنیم. سپس **تابع** Q (**ارزش** Q) را به عنوان میانگین بازگشت یک زوج حالت–اقدام در چندین پردینه، محاسبه می‌کنیم، همانطور که در ادامه نشان داده‌ایم:

$$Q(s, a) = \frac{\text{total\_return}(s, a)}{N(s, a)}$$

فرض کنید پس از محاسبه **تابع** Q به عنوان میانگین بازگشت زوج حالت–اقدام، **تابع** Q ما مانند جدول ۴.۵ باشد.

حالت	اقدام	ارزش
S_0	۰	۲
S_0	۱	۱
S_1	۰	۱
S_1	۱	۲

جدول ۴.۵: جدول Q

از تابع Q قبلی، می‌توانیم با انتخاب یک اقدام که در هرحالتی، حداکثر مقدار Q را دارد، یک خط‌مشی جدید استخراج کنیم. یعنی:

$$\pi = \arg\max_{a} Q(s, a)$$

سیاست جدید، اقدام ۰ را در حالت s_0 و اقدام ۱ را در حالت s_1 انتخاب می‌کند زیرا دارای حداکثر ارزش Q است.

با این حال، این خط‌مشی جدید یک خط‌مشی بهینه نخواهد بود. زیرا این خط‌مشی جدید از تابع Q استخراج می‌شود که با استفاده از خط‌مشی تصادفی (بختکی) محاسبه می‌شود. یعنی، ما یک خط مشی تصادفی (دلبخواه) را مقداردهی اولیه کردیم و چندین اپیزود را با استفاده از خط‌مشی تصادفی تولید کردیم، سپس تابع Q را با گرفتن میانگین بازگشتی زوج حالت–اقدام در چندین پردینه ، محاسبه کردیم. بنابراین، ما از خط مشی تصادفی برای محاسبه تابع Q استفاده می‌کنیم و لذا سیاست جدید استخراج شده از تابع Q، یک خط‌مشی بهینه نخواهد بود.

اما اکنون که یک خط‌مشی جدید از تابع Q استخراج کرده‌ایم، می‌توانیم از این خط‌مشی جدید برای تولید اپیزودها در تکرار بعدی و محاسبه Q جدید استفاده کنیم. سپس از تابع Q جدید، یک خط‌مشی جدید استخراج می‌کنیم. این مراحل را به طور مکرر تا زمانی که خط‌مشی بهینه را پیدا کنیم، تکرار می‌کنیم. این موضوع، در مراحل زیر به وضوح توضیح داده شده است:

تکرار ۱– **خط‌مشی** تصادفی (دلبخواه) اولیه را π_0 درنظر بگیرید. ما از این **خط‌مشی** تصادفی (بختکی) برای تولید یک پردینه (اپیزود) استفاده می‌کنیم و سپس **تابع** Q مربوطه، Q^{π_0} را با گرفتن میانگین بازگشتی زوج حالت–اقدام، محاسبه می‌کنیم. سپس از این **تابع** Q^{π_0}، یک سیاست جدید π_1 استخراج می‌کنیم. این ***خط‌مشی جدید*** π_1 سیاست بهینه نخواهد بود زیرا از **تابع** Qای استخراج می‌شود که با استفاده از سیاست تصادفی محاسبه می‌شود.

تکرار ۲– بنابراین، از ***سیاست جدید*** π_1 مشتق شده از تکرار قبلی برای تولید یک پردینه و محاسبه تابع Q جدید Q^{π_1} به عنوان میانگین بازگشت یک زوج حالت–اقدام، استفاده می‌کنیم. سپس از این تابع Q، یک سیاست جدید π_2

استخراج می‌کنیم. اگر **خط‌مشی** $\pi_۲$ بهینه باشد، متوقف می‌شویم، در غیر این صورت به تکرار ۳ می‌رویم.

تکرار ۳- اکنون، از **سیاست جدید** $\pi_۲$ بدست آمده از تکرار قبلی، برای تولید یک پردینه و محاسبه تابع Q جدید $Q^{\pi_۲}$ استفاده می‌کنیم. سپس از این تابع $Q^{\pi_۲}$، یک **سیاست جدید** $\pi_۳$ استخراج می‌کنیم. اگر $\pi_۳$ بهینه باشد، متوقف می‌شویم، در غیر این صورت به تکرار بعدی می‌رویم.

ما این فرآیند را برای چندین بار تکرار می‌کنیم تا زمانی که **سیاست بهینه** π^* را پیدا کنیم. همانطور که در شکل ۴.۲۱ نشان داده شده است:

$$\pi_۰ \rightarrow Q^{\pi_۰} \rightarrow \pi_۱ \rightarrow Q^{\pi_۱} \rightarrow \pi_۲ \rightarrow Q^{\pi_۲} \rightarrow \pi_۳ \rightarrow Q^{\pi_۳} \rightarrow \cdots \rightarrow \pi^* \rightarrow Q^{\pi^*}$$

شکل ۴.۲۱: مسیر یافتن سیاست بهینه

این مرحله، **ارزیابی و بهبود خط‌مشی** نامیده می‌شود و شبیه **روش تکرار خط‌مشی** است که در فصل ۳ (معادله بلمن و برنامه‌ریزی پویا) به آن پرداختیم. **ارزیابی سیاست** [۱] به این معناست که در هر مرحله، ما سیاست را ارزیابی می‌کنیم. **بهبود سیاست** [۲] به این معناست که در هر مرحله با بیشینه‌گیری از ارزش Q، خط‌مشی را بهبود می‌بخشیم. توجه داشته باشید که در اینجا، ما خط‌مشی را به شیوه‌ای حریصانه انتخاب می‌کنیم به این معنی که ما فقط با پیداکردن بیشینه **ارزش** Q، **خط‌مشی** π را انتخاب می‌کنیم و بنابراین می‌توانیم خط‌مشی خود را یک خط‌مشی حریصانه بنامیم.

اکنون که درک اولیه‌ای از نحوه عملکرد **روش کنترل مونت‌کارلو** داریم، در بخش بعدی، الگوریتم **روش کنترل مونت‌کارلو** را بررسی کرده و با جزئیات بیشتری در مورد آن آشنا خواهیم شد.

---

[۱] Policy Evaluation
[۲] Policy Improvement

الگوریتم کنترل مونت‌کارلو

مراحل زیر **الگوریتم کنترل مونت کارلو** را نشان می‌دهد. همانطور که مشاهده می‌کنیم، برخلاف **روش پیش‌بینی مونت کارلو**، در اینجا هیچ **سیاستی** داده نمی‌شود. بنابراین، ما با مقداردهی اولیه خط‌مشی تصادفی (دلبخواه) شروع کرده و از این سیاست تصادفی برای تولید یک پردینه در اولین تکرار استفاده می‌کنیم. سپس، تابع Q (ارزش Q) را به عنوان میانگین بازگشت زوج حالت–اقدام محاسبه می‌کنیم.

هنگامی که تابع Q را داریم، با انتخاب یک اقدام در هر حالت که دارای حداکثر **ارزش** Q است، یک **سیاست جدید** استخراج می‌کنیم. در تکرار بعدی، از سیاست جدید استخراج شده برای تولید یک پردینه و محاسبه تابع Q جدید (ارزش Q) به عنوان میانگین بازگشت زوج حالت–اقدام استفاده می‌کنیم. ما این مراحل را برای یافتن سیاست بهینه، چندین بار تکرار می‌کنیم.

یک مسئله دیگر اینکه، باید توجه داشته باشیم که همانطور که در روش پیش‌بینی مونت کارلوی اولین بازدید یاد گرفتیم، در اینجا، بازگشت زوج حالت–اقدام را فقط برای اولین باری که یک زوج حالت–اقدام در پردینه بازدید می‌شود، محاسبه می‌کنیم.

برای درک بهتر، می‌توان الگوریتم **کنترل مونت‌کارلو** را با **پیش‌بینی مونت کارلوی تابع** Q مقایسه نمود. یک تفاوتی که می‌توان مشاهده کرد این است که، در اینجا، تابع Q را در هر تکرار محاسبه می‌کنیم. اما اگر متوجه شده باشید، در پیش بینی مونت‌کارلوی تابع Q، ما تابع Q را بعد از تمام تکرارها محاسبه می‌کنیم. دلیل محاسبه تابع Q در هر تکرار در اینجا این است که ما به تابع Q برای استخراج سیاست جدید نیاز داریم تا بتوانیم از سیاست جدید استخراج شده در تکرار بعدی، برای تولید یک پردینه (اپیزود)، استفاده کنیم:

۱. فرض کنید $\text{total\_return}(s,a)$ مجموع **بازگشت** یک زوج حالت–اقدام در چندین اپیزود و $N(s,a)$ تعداد دفعاتی باشد که یک زوج حالت–اقدام در چندین اپیزود، بازدید شده است. مقدار $\text{total\_return}(s,a)$ و

$N(s,a)$ را برای همه زوج‌های حالت–اقدام، برابر صفر مقداردهی کنید و یک **خط‌مشی** تصادفی (بختکی) π را برای مقدار اولیه درنظر بگیرید.

۲. برای M تعداد تکرار:

۱. یک پردینه با استفاده از **خط‌مشی** π ایجاد کنید.

۲. تمام **پاداش‌های** به دست آمده در پردینه را در لیستی به نام **پاداش** ذخیره کنید.

۳. برای هر مرحله t در اپیزود:

اگر (s_t, a_t) برای اولین بار در پردینه اتفاق می‌افتد:

۱. بازگشت یک زوج حالت–اقدام، $R(s_t,a_t) = \text{sum}(\text{rewards}[t:])$ را محاسبه کنید

۲. **بازده** کل زوج حالت–اقدام را به‌صورت زیر، به‌روزرسانی کنید،

$$total\_return(s_t,a_t) = total\_return(s_t,a_t) + R(s_t,a_t)$$

۳. شمارنده را به صورت $N(s_t,a_t) = N(s_t,a_t) + ۱$ به روز کنید.

۴. **ارزش** Q را فقط با گرفتن میانگین محاسبه کنید، یعنی،

$$Q(s_t,a_t) = \frac{total\_return(s_t,a_t)}{N(s_t,a_t)}$$

٤. **خط‌مشی** به روز شده جدید π را با استفاده از **تابع** Q محاسبه کنید:

$$\pi = \arg\max_a Q(s,a)$$

از الگوریتم قبلی، می‌توانیم مشاهده کنیم که با استفاده از سیاست π یک پردینه تولید می‌کنیم. سپس برای هر مرحله در پردینه ، بازگشت زوج حالت–اقدام را محاسبه کرده و تابع Q مربوط به $Q(s_t,a_t)$ را به عنوان بازده متوسط محاسبه می‌کنیم، سپس از این تابع Q، یک سیاست جدید π را استخراج می‌کنیم. این مرحله را به طور مکرر تکرار می‌کنیم تا سیاست بهینه π را پیدا کنیم. بنابراین، ما یاد گرفتیم که چگونه وظیفه کنترل را با استفاده از روش مونت کارلو انجام دهیم.

روش‌های کنترل را می‌توان به دو نوع طبقه‌بندی کرد:

- کنترل با سیاست عیان (**سیاست-همگاه**)[۱]
- کنترل با سیاست نهان (**سیاست-ناهمگاه**)[۲]

کنترل براساس سیاستِ همگاه (سیاستِ عیان)ـ در روش **کنترل** براساس "**همگاه بودن سیاست**"، عامل با استفاده از فقط یک گونه **سیاست** رفتار کرده و مرتبا سعی می‌کند **همان سیاست** را بهبود بخشد. یعنی در روش با سیاست همگاه (یا **عیان-سیاست**)، اپیزودهایی را با استفاده از یک سیاست تولید کرده و لذا همان خطمشی را به طور مکرر بهبود می‌دهیم تا خطمشی بهینه را پیدا کنیم. به عنوان مثال، روش کنترل مونت‌کارلو که در بالا یاد گرفتیم، می‌تواند کنترل مونت‌کارلو بر اساس سیاست همگاه نامیده شود زیرا ما در حال تولید پردینه‌ها با استفاده از یک خطمشی π هستیم، و همچنین سعی می‌کنیم **همان** سیاست π را در هر تکرار بهبود دهیم تا سیاست بهینه را محاسبه کنیم.

کنترل سیاست ناهمگاه (سیاستـ نهان) – در روش **کنترل** با سیاست ناهمگاه (یا **نهان-سیاست**)، عامل با استفاده از یک **سیاست واسط** همچون b رفتار کرده و سعی می‌کند **سیاستِ اصلی** یعنی π را بهبود بخشد. یعنی در روش خطمشی ناهمگاه، با استفاده از یک **سیاست واسط**، پردینه‌هایی را تولید کرده و سعی می‌کنیم سیاستهای مختلف را به صورت مکرر بهبود بخشیم تا خطمشی بهینه را پیدا کنیم.

نحوه عملکرد دقیق دو روش کنترل قبلی را در بخش‌های بعدی خواهیم آموخت.

---

[۱] On-Policy Control
[۲] Off-Policy Control

برای این دو واژه می‌توان از معادلهای زیر بهره برد:
برای واژه On معادلهایی همچون همگاه، عیان، باز، روشن، فعال، درگیر، نقش‌آفرین، در متن، همزمان، همپا
برای واژه Off معادلهایی همچون ناهمگاه، نهان، بسته، خاموش، منفعل، در سایه، در حاشیه، ناهمزمان، ناهمپا
باید دقت نمود که درروش دوم، عملا دو تا سیاست داریم: اصلی و واسط. یک سیاست اصلی، که پنهان و Off است و یک سیاست واسط (رفتار) که محاسبات بر مبنای آن انجام می‌شود و نتیجه نهایی آن بعنوان سیاست اصلی (هدف) انتخاب می‌گردد.

کنترل مونت کارلو ک سیاست همگاه (عیان–سیاست)[۱]

دو نوع روش کنترل مونت کارلو براساس سیاست همگاه یا عیان وجود دارد:

- شروعهای کاوشی مونت‌کارلو[۲]
- مونت کارلو با سیاست حریصانه اپسیلون[۳]

شروع کاوشی مونت کارلو

ما قبلاً نحوه عملکرد ***روش کنترل مونت‌کارلو*** را یاد گرفته‌ایم. یکی از مواردی که ممکن است بخواهیم در نظر بگیریم **کاوش** یا **اکتشاف**[۴] است. در هر حالت، چندین اقدام می‌تواند وجود داشته باشد: برخی از اقدامات بهینه خواهند بود، در حالی که برخی دیگر خیر. برای درک اینکه آیا یک اقدام بهینه است یا خیر، عامل ما باید با انجام آن اقدام، کاوش کند. اگر عامل ما، یک اقدام خاص را در یک حالت، جستجو یا کاوش نکند، هرگز متوجه نخواهد شد که آیا آن اقدام خوب است یا خیر. بنابراین، چگونه می‌توانیم این مسئله را حل کنیم؟ یعنی چگونه می‌توانیم از اکتشاف یا جستجوی کافی، اطمینان حاصل کنیم؟ اینجا جایی است که «**شروع کاوشی مونت کارلو**» به ما کمک می‌کند.

در روش ***شروع کاوشی مونت‌کارلو***، به همه زوج‌های (حالت–اقدام) یک مقدار **احتمال غیر صفر** تخصیص می‌دهیم تا شانس انتخاب شدن بعنوان زوج (حالت–اقدام) اولیه را داشته باشند. بنابراین قبل از ایجاد هر اپیزودی، ما ابتدا زوج (حالت–اقدام) اولیه را به طور تصادفی انتخاب می‌کنیم و در ادامه پردینه را از این زوج (حالت–اقدام) اولیه با پیروی از خط‌مشی π، تولید می‌کنیم. سپس، در هر تکرار، خط‌مشی ما با روش حریصانه، به‌روزرسانی می‌شود (انتخاب حداکثر ارزش Q؛ برای جزئیات بیشتر به بخش بعدی با عنوان «مونت کارلو با خط‌مشی حریصانه اپسیلون» مراجعه کنید).

مراحل زیر، الگوریتم **شروعهای کاوشی کنترل مونت‌کارلو** را نشان می‌دهد. این، اساساً همان چیزی است که قبلاً

---

[۱] On-Policy Monte Carlo Control
[۲] Monte Carlo Exploring Starts
[۳] Monte Carlo with The Epsilon-Greedy Policy
[۴] Exploration

برای بخش الگوریتم کنترل مونت‌کارلو یاد گرفتیم، با این تفاوت که در اینجا، یک زوج (حالت–اقدام) اولیه را انتخاب می‌کنیم و پردینه‌هایی را که از این زوج حالت–اقدام اولیه شروع می‌شود، تولید می‌کنیم (همانطور که در جمله پررنگ شدهٔ زیر نوشته شده است).

۱. فرض کنید $total\_return(s,a)$ مجموع **بازگشت** یک زوج حالت–اقدام در چندین پردینه و $N(s,a)$، تعداد دفعاتی باشد که یک زوج حالت–اقدام در چندین پردینه ، بازدید شده است. مقدار $total\_return(s,a)$ و $N(s,a)$ را برای همه زوج‌های حالت–اقدام، برابر صفر، مقداردهی کنید و یک **خط‌مشی** تصادفی (بختکی) π را برای مقدار اولیه درنظر بگیرید.

۲. برای M تعداد تکرار:

۱. یک حالت اولیه s_0 و اقدام اولیه a_0 را به صورت تصادفی (دلبخواه) در حالی انتخاب می‌کنیم که همه زوج‌های حالت–اقدام، احتمالی بیشتر از صفر دارند.

۲. یک پردینه از حالت اولیه انتخاب شده s، و **اقدام** a. با استفاده از **سیاست** π تولید کنید.

۳. تمام **پاداش‌های** به دست آمده در پردینه (اپیزود) را در فهرستی به نام **پاداش** ذخیره کنید.

۴. برای هر مرحله t در اپیزود:

اگر (s_t, a_t) برای اولین بار در پردینه اتفاق می‌افتد:

۱. **بازگشت** یک زوج حالت–اقدام، $R(s_t,a_t) = \text{sum}(\text{rewards}[t:])$ را محاسبه کنید.

۲. **بازده کل** زوج حالت–اقدام را به‌صورت زیر، به‌روزرسانی کنید،

$$total\_return(s_t,a_t) = total\_return(s_t,a_t) + R(s_t,a_t)$$

۳. شمارنده را به صورت $N(s_t,a_t) = N(s_t,a_t) + ۱$ به روز کنید.

۴. **ارزش** Q را فقط با گرفتن میانگین محاسبه کنید، یعنی،

$$Q(s_t,a_t) = \frac{\text{total\_return}(s_t,a_t)}{N(s_t,a_t)}$$

۵ **سیاست** به‌روز شده جدید π را با استفاده از **تابع** Q محاسبه کنید:

$$\pi = \arg\max_a Q(s,a)$$

یکی از اشکالات عمده **روش شروعهای کاوشی**، این است که برای هر محیطی قابل اجرا نیست. یعنی ما نمی‌توانیم به‌طور تصادفی (بختکی) هر زوج حالت–اقدام را به عنوان یک زوج حالت–اقدام اولیه انتخاب کنیم، زیرا در برخی محیطها فقط یک زوج حالت–اقدام، وجود دارد که می‌تواند به عنوان زوج حالت–اقدام اولیه، عمل کند. بنابراین ما نمی‌توانیم به طور تصادفی هر زوج حالت–اقدامی را به عنوان زوج حالت–اقدام اولیه، انتخاب کنیم.

به عنوان مثال، فرض کنید ما در حال آموزش یک عامل، برای انجام یک بازی اتومبیل‌رانی هستیم؛ نمی‌توانیم پردینه را در یک موقعیت شانسی به عنوان حالت اولیه و یک اقدام احتمالی، به عنوان اقدام اولیه شروع کنیم، زیرا یک حالت شروع ثابت و یک اقدام مشخص به عنوان حالت و اقدام اولیه داریم.

بنابراین، برای غلبه بر این مشکل در شروع کاوشی، در بخش بعدی، با روش کنترل مونت کارلو با نوع جدیدی از خطمشی به نام «**سیاست اپسیلون حریصانه**» آشنا خواهیم شد.

مونت کارلو با سیاست اپسیلون حریصانه

قبل از ادامه، ابتدا اجازه دهید بفهمیم که سیاست اپسیلون حریصانه چیست که در مبحث یادگیری تقویتی، در همه‌جا حضور دارد.

ابتدا بیایید یاد بگیریم که **سیاست حریصانه**[1] چیست. **سیاست حریصانه**، سیاستی است که بهترین اقدام موجود را در لحظه انتخاب می کند. به عنوان مثال، فرض کنید در حالت **A** هستیم و چهار اقدام ممکن در حالت داریم. درنظر می‌گیریم اقدامات *بالا*، *پایین*، چپ و *راست* باشد. اما بیایید فرض کنیم عامل ما فقط دو اقدام *بالا* و *راست*، را در حالت **A** بررسی کرده است. ارزش Q اقدامات بالا و *راست* در حالت **A** در جدول ۴.۶ نشان داده شده است.

ما آموختیم که سیاست حریصانه، بهترین اقدام موجود در حال حاضر را انتخاب می‌کند. بنابراین سیاست حریصانه،

[1] Greedy Policy

جدول Q را بررسی می‌کند و اقدامی را انتخاب می‌کند که حداکثر ارزش Q را در حالت **A** دارد. همانطور که می‌بینیم، اقدام *بالا*، حداکثر **ارزش** Q را دارد. بنابراین **سیاست حریصانه** ما، اقدام *بالا* را در حالت **A** انتخاب می‌کند.

ارزش	اقدام	حالت
۳	*بالا*	**A**
۱	*راست*	**A**

جدول ۴.۶: عامل فقط دو اقدام را در حالت **A** بررسی کرده است.

اما یک مشکل با **سیاست حریصانه** این است که هرگز سایر اقدامات ممکن را بررسی نمی‌کند و به جای آن، همیشه بهترین اقدام موجود در لحظه را انتخاب می‌کند. در مثال قبل، سیاست حریصانه همیشه اقدام *بالا* انتخاب می‌کند. اما ممکن است اقدامات دیگری در حالت **A** وجود داشته باشد که بهتر از اقدام *بالا* باشد که عامل هنوز آن را کاوش نکرده است. یعنی ما هنوز دو اقدام دیگر یعنی *پایین* و چپ در حالت **A** داریم که عامل هنوز کاوش نکرده است و ممکن است بهتر از اقدام حرکت به *بالا* باشند.

بنابراین، اکنون سؤال این است که آیا عامل باید تمام اقدامات دیگر در این حالت را بررسی کند و بهترین اقدام را به عنوان اقدامی که دارای حداکثر ارزش Q است، انتخاب کند یا از بهترین اقدام از اقدامات قبلاً کاوش شده را استفاده کند؟ این مخمصه را **دوراهه اکتشاف- انتفاع** یا **دوگانه کاوش-یابش**[۱] می‌نامند.

فرض کنید، مسیرهای زیادی از محل کار ما تا خانه وجود دارد و ما تاکنون تنها دو مسیر را بررسی کرده‌ایم. بنابراین،

---

[۱] Exploration-Exploitation Dilemma

معادلهای فارسی برای واژه Exploration شامل: شناسایی، اکتشاف، کاوش، جستجو، تحقیق، پژوهش، بررسی، کندوکاو، گشت و گذار، گشتن

و معادلهای فارسی برای واژه Exploitation شامل: بهره‌مندی، بهره‌برداری، انتفاع، استخراج، استثمار، استحصال، استفاده، بهره‌جویی، بهره‌یابی، برداشتن، سود، فایده، منفعت، یابش، بازگشت است.

معادلهای فارسی برای واژه Dilemma شامل: دوراهی، مضل، مخمصه و امثال آنست

معادل پیشنهادی برای ترکیب فوق: **دوگانه اکتشاف-انتفاع**، یا **مخمصه کاوش-یابش**، یا **دوراهی گشت-بازگشت**، **یا پی بردن-بهره بردن** و امثال آن است.

برای رسیدن به خانه، می‌توانیم از بین دو مسیری که کاوش کرده‌ایم، مسیری را که ما را سریع‌تر به خانه می‌برد، انتخاب کنیم. با این حال، هنوز بسیاری از مسیرهای دیگر وجود دارد که ما هنوز کاوش نکرده‌ایم که ممکن است حتی بهتر از مسیر بهینه فعلی ما باشد. سوال این است که آیا باید مسیرهای جدید را کاوش کنیم (اکتشاف) یا اینکه همیشه باید از مسیر بهینه فعلی خود استفاده کنیم (انتفاع)؟

برای پرهیز از این معضل، سیاست جدیدی به نام **سیاست اپسیلون حریصانه**[۱] را معرفی می‌کنیم. در اینجا، همه اقدامات با احتمال غیر صفر (یعنی به مقدار اپسیلون[۲]) امتحان می‌شوند. با احتمالی به اندازه اپسیلون، اقدامات مختلف را به صورت تصادفی بررسی می‌کنیم و با یک احتمال یک منهای اپسیلون ($\epsilon - ۱$)، اقدامی را انتخاب می‌کنیم که حداکثر ارزش Q را داشته باشد. یعنی به احتمال ϵ ، یک اقدام تصادفی (**اکتشاف**) و به احتمال $\epsilon - ۱$ ، بهترین اقدام (**انتفاع**) را انتخاب می‌کنیم.

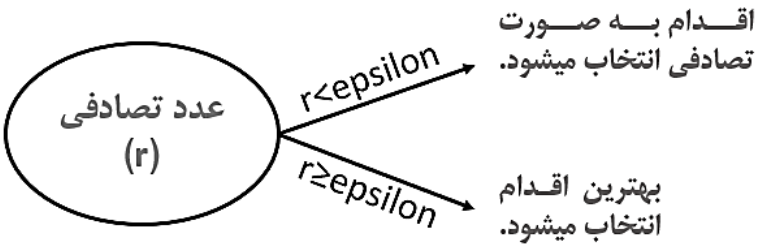
در **سیاست اپسیلون حریصانه**، اگر مقدار اپسیلون را روی صفر قرار دهیم، تبدیل به یک **خط‌مشی حریصانه** (فقط انتفاع) می‌شود و وقتی مقدار اپسیلون را روی ۱ قرار می‌دهیم، در آن صورت همیشه فقط **کاوش** را انجام می‌دهیم. بنابراین، مقدار اپسیلون باید به طور بهینه بین ۰ و ۱ انتخاب شود.

فرض کنید اپسیلون را برابر ۰.۵ تنظیم می‌کنیم؛ سپس یک عدد تصادفی از توزیع یکنواخت تولید می‌کنیم و اگر عدد تصادفی کمتر از اپسیلون (۰.۵) باشد، یک اقدام تصادفی (**کاوش**) را انتخاب می‌کنیم، اما اگر عدد تصادفی بزرگتر یا مساوی با اپسیلون باشد، بهترین اقدام را انتخاب می‌کنیم، یعنی اقدامی که حداکثر ارزش Q را دارد (**یابش**).

بنابراین، به این ترتیب، اقداماتی را که قبلاً ندیده‌ایم را با احتمال اپسیلون کاوش می‌کنیم و بهترین اقدامات را از میان اقدامات کاوش شده با احتمال $\epsilon - ۱$ انتخاب می‌کنیم. همانطور که شکل ۴.۲۲ نشان می‌دهد، اگر عدد تصادفی که از توزیع یکنواخت تولید می‌کنیم، کمتر از اپسیلون باشد، یک اقدام تصادفی را انتخاب می‌کنیم. اگر عدد تصادفی بزرگتر یا مساوی با اپسیلون باشد، بهترین اقدام را انتخاب می‌کنیم:

---

[۱] Epsilon-Greedy Policy
[۲] Epsilon



شکل ۴.۲۲: سیاست اپسیلون حریصانه

تصویر زیر، کد پایتون را برای **سیاست اپسیلون حریصانه** نشان می‌دهد:

```
def epsilon_greedy_policy(state, epsilon):
    if random.uniform(0,1) < epsilon:
         return env.action_space.sample()
    else:
         return max(list(range(env.action_space.n)), key = lambda x:
q[(state,x)])
```

اکنون که متوجه شدیم سیاست اپسیلون حریصانه چیست و چگونه برای حل معضل اکتشاف- انتفاع از آن استفاده می‌شود، در بخش بعدی به نحوه استفاده از خطمشی حریصانه اپسیلون در روش کنترل مونت کارلو خواهیم پرداخت.

الگوریتم کنترل مونت‌کارلو با خطمشی اپسیلون حریصانه

الگوریتم کنترل مونت کارلو با خطمشی حریصانه اپسیلون، اساساً مشابه **الگوریتم کنترل مونت کارلو** است که قبلاً یاد گرفتیم با این تفاوت که در اینجا اقداماتی را بر اساس خطمشی حریصانه اپسیلون انتخاب می‌کنیم تا از معضل اکتشاف- انتفاع جلوگیری کنیم. مراحل زیر، الگوریتم مونت کارلو با خطمشی حریصانه اپسیلون را نشان می‌دهد:

۱. فرض کنید $\text{total\_return}(s,a)$ مجموع **بازگشت** یک زوج حالت-اقدام در چندین اپیزود و $N(s,a)$

تعداد دفعاتی باشد که یک زوج حالت–اقدام در چندین پردینه، بازدید شده است. مقدار $\text{total\_return}(s,a)$ و $N(s,a)$ را برای همه زوج‌های حالت–اقدام، برابر صفر مقداردهی کنید و یک **خط‌مشی** تصادفی π را برای مقدار اولیه درنظر بگیرید.

۲. برای M تعداد تکرار:

۱. یک پردینه با استفاده از **خط‌مشی** π ایجاد کنید.

۲. تمام **پاداش‌های** به دست آمده در پردینه را در لیستی به نام **پاداش** ذخیره کنید.

۳. برای هر مرحله t در اپیزود:

اگر (s_t, a_t) برای اولین بار در پردینه اتفاق می‌افتد:

۱. **بازگشت** (**بازده**) یک زوج حالت–اقدام، $R(s_t,a_t) = \text{sum(rewards[t:])}$ را محاسبه کنید.

۲. **بازده کل** زوج حالت–اقدام را به‌صورت زیر، به‌روزرسانی کنید،

$$\text{total\_return}(s_t,a_t) = \text{total\_return}(s_t,a_t) + \text{R}(s_t,a_t)$$

۳. شمارنده را به صورت $N(s_t,a_t) = N(s_t,a_t) + ۱$ به روز کنید.

۴. **ارزش** Q را فقط با گرفتن میانگین محاسبه کنید، یعنی،

$$Q(s_t,a_t) = \frac{\text{total\_return}(s_t,a_t)}{N(s_t,a_t)}$$

۵. **سیاست به‌روز شده جدید** π را با استفاده از تابع Q محاسبه کنید. قرار دهید:

$$a^* = \arg\max_a Q(s,a).$$

سیاست π، بهترین اقدام a^* را با **احتمال** $\epsilon - ۱$ و اقدام تصادفی را با **احتمال** ϵ انتخاب می‌کند.

همانطور که مشاهده می‌کنیم، در هر تکرار، پردینه را با استفاده از سیاست π، تولید کرده و همچنین سعی می‌کنیم همان سیاست π را در هر تکرار بهبود دهیم تا سیاست بهینه محاسبه شود.

پیاده سازی کنترل MC با سیاست همگاه (عیان_سیاست)

اکنون، بیایید یاد بگیریم که چگونه روش **کنترل MC** را با **سیاست اپسیلون حریصانه** برای انجام بازی بلک جک پیاده سازی کنیم. یعنی خواهیم دید که چگونه می توانیم از روش کنترل **MC** برای یافتن سیاست بهینه در بازی بلک جک استفاده کنیم.

ابتدا، بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import pandas as pd
import random
from collections import defaultdict
```

یک محیط بلک جک ایجاد کنید:

```
env = gym.make('Blackjack-v0')
```

فرهنگ لغت را برای ذخیره مقادیر Q مقداردهی اولیه کنید:

```
Q = defaultdict(float)
```

فرهنگ لغت را برای ذخیره بازده کل جفت state-action مقداردهی اولیه کنید:

```
total_return = defaultdict(float)
```

فرهنگ لغت را برای ذخیره تعداد دفعات بازدید از یک جفت حالت–اقدام مقداردهی اولیه کنید:

```
N = defaultdict(int)
```

تعریف سیاست اپسیلون_حریصانه

ما یاد گرفتیم که اقدامات خود را بر اساس سیاست حریصانه اپسیلون انتخاب کنیم، بنابراین تابعی به نام epsilon_greedy_policy تعریف می کنیم که حالت و مقدار Q را به عنوان ورودی می‌گیرد و عملی را که باید در حالت داده شده انجام شود برمی‌گرداند:

```
def epsilon_greedy_policy(state,Q):
```

مقدار اپسیلون را روی ۰.۵ تنظیم کنید:

```
    epsilon = 0.5
```

یک مقدار تصادفی را از توزیع یکنواخت نمونه برداری کنید. اگر مقدار نمونه برداری شده کمتر از اپسیلون باشد، یک عمل تصادفی را انتخاب می کنیم، در غیر این صورت بهترین اقدامی را انتخاب کنید که حداکثر مقدار Q را داشته باشد:

```
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return max(list(range(env.action_space.n)), key = lambda x:
Q[(state,x)])
```

تولید یک پردینه (اپیزود)

اکنون، بیایید با استفاده از سیاست حریصانه اپسیلون یک پردینه تولید کنیم. ما تابعی به نام generate_episode تعریف می کنیم که مقدار Q را به عنوان ورودی می گیرد و پردینه را برمی‌گرداند.

ابتدا، بیایید تعداد مراحل زمانی را تنظیم کنیم:

```
num_timesteps = 100
```

حال، بیایید تابع را تعریف کنیم:

```
def generate_episode(Q):
```

فهرستی را برای پردینه مقداردهی اولیه کنید:

```
    episode = []
```

حالت را با استفاده از تابع بازتنظیم راه‌اندازی کنید:

```
    state = env.reset()
```

سپس برای هر مرحله زمانی:

```
    for t in range(num_timesteps):
```

عمل را مطابق با سیاست اپسیلون–حریصانه انتخاب کنید:

```
        action = epsilon_greedy_policy(state,Q)
```

عمل انتخاب شده را انجام دهید و اطلاعات وضعیت بعدی را ذخیره کنید:

```
        next_state, reward, done, info = env.step(action)
```

حالت، اقدام و پاداش را در فهرست قسمت ذخیره کنید:

```
        episode.append((state, action, reward))
```

اگر حالت بعدی حالت نهایی است، حلقه را بشکنید، در غیر این صورت حالت بعدی را به حالت فعلی به‌روزرسانی کنید:

```
        if done:
            break

        state = next_state

    return episode
```

محاسبه سیاست بهینه

اکنون، بیایید یاد بگیریم که چگونه سیاست بهینه را محاسبه کنیم. ابتدا ، بیایید تعداد تکرارها ، یعنی تعداد اپیزودهایی را که می خواهیم تولید کنیم، تنظیم کنیم:

```
num_iterations = 500000
```

برای هر تکرار:

```
for i in range(num_iterations):
```

ما یاد گرفتیم که در روش **کنترل سیاست-هنگام**، هیچ سیاستی به عنوان ورودی به ما داده نخواهد شد. بنابراین، ما یک سیاست تصادفی (بختکی) را در اولین تکرار مقداردهی اولیه کرده و با محاسبه مقدار Q، سیاست را به صورت تکراری بهبود می بخشیم. از آنجا که ما سیاست را از تابع Q استخراج می کنیم ما مجبور نیستیم به صراحت سیاست را تعریف کنیم. با بهبود مقدار Q، سیاست نیز به طور ضمنی بهبود می یابد. یعنی در اولین تکرار، پردینه را با استخراج خطمشی اپسیلون–حریصانه از تابع Q مقداردهی اولیه تولید می‌کنیم. در طی یک سری تکرارها، تابع Q بهینه را پیدا خواهیم کرد و از این رو سیاست بهینه را نیز پیدا می کنیم.

بنابراین، در اینجا تابع Q مقداردهی اولیه خود را برای تولید یک پردینه (اپیزود) ارسال می‌کنیم:

```
    episode = generate_episode(Q)
```

همه جفتهای state-action را در قسمت دریافت کنید:

```
    all_state_action_pairs = [(s, a) for (s,a,r) in episode]
```

تمام جوایز به دست آمده در پردینه را در فهرست پاداشها ذخیره کنید:

```
    rewards = [r for (s,a,r) in episode]
```

برای هر مرحله از پردینه:

```
    for t, (state, action,_) in enumerate(episode):
```

اگر جفت حالت–عمل برای اولین بار در پردینه اتفاق می افتد:

```
        if not (state, action) in all_state_action_pairs[0:t]:
```

مقدار بازده یا R هر جفت حالت–عمل را به عنوان مجموع پاداشها محاسبه کنید ($R(s_t, a_t)$ = sum(rewards[t:])

```
            R = sum(rewards[t:])
```

بازده کل جفتهای حالت–اقدام را به صورت زیر به‌روزرسانی کنید:

$$\text{total\_return}(s_t, a_t) = \text{total\_return}(s_t, a_t) + R(s_t, a_t)$$

```
            total_return[(state,action)] = total_return[(state,action)]
+ R
```

تعداد دفعاتی که جفت حالت–اقدام به صورت $N(s_t, a_t) = N(s_t, a_t) + 1$ بازدید می شود، به‌روز کنید:

```
            N[(state, action)] += 1
```

مقدار Q را فقط با در نظر گرفتن میانگین محاسبه کنید، یعنی،

$$Q(s_t, a_t) = \frac{\text{total\_return}(s_t, a_t)}{N(s_t, a_t)}$$

```
            Q[(state,action)] = total_return[(state, action)] /
N[(state, action)]
```

بنابراین در هر تکرار، مقدار Q بهبود یافته و سیاست نیز بهبود می‌یابد. پس از تمام تکرارها، می‌توانیم نگاهی به مقدار Q هر جفت حالت–اقدام در فریم داده pandas برای وضوح بیشتر بیندازیم. ابتدا، بیایید فرهنگ لغت مقدار Q را به یک دیتافریم پانداس تبدیل کنیم:

```
df = pd.DataFrame(Q.items(),columns=['state_action pair','value'])
```

بیایید به چند ردیف اول دیتافریم نگاه کنیم:

```
df.head(11)
```

	state_action pair	value
۰	((12, 10, False), 0)	-۰,٥٦٢٣٧٢
۱	((12, 10, False), 1)	-۰,٤٩٦١٦٤
۲	((14, 3, True), 0)	-۰,٢٢٢٢٢٢
۳	((14, 3, True), 1)	۰,۰٨٦٩٥٧
٤	((19, 2, False), 0)	۰,٣٥٤٥٤٥
٥	((19, 2, False), 1)	-۰,٦٩٥١٢٢
٦	((16, 10, False), 0)	-۰,٥٣۰٦٦٢
۷	((14, 5, False), 0)	-۰,١١٢٧٨٢
۸	((17, 9, True), 0)	-۰,٣٣٣٣٣٣
۹	((20, 9, False), 1)	-۰,٨٦١٧٨٩
۱۰	((19, 10, False), 0)	-۰,۰١٤٤٧٩

شکل ۴.۲۳: مقادیر Q جفتهای حالت–عمل

همانطور که مشاهده می‌کنید، ما مقادیر Q را برای همه جفتهای حالت–عمل داریم. اکنون می‌توانیم با انتخاب اقدامی که حداکثر مقدار Q را در هر حالت دارد، سیاست را استخراج کنیم. به عنوان مثال، فرض کنید ما در حالت `(21,8,True)` هستیم. حال، آیا باید اقدام ۰ (توقف) یا اقدام ۱ (تداوم) را انجام دهیم؟ انجام عمل ۰ (توقف) در اینجا منطقی‌تر است، زیرا مقدار مجموع کارتهای ما در حال حاضر ۲۱ است و اگر عمل ۱ (تداوم) را انجام دهیم بازی ما منجر به شکست می‌شود.

توجه داشته باشید که به دلیل شرایط تصادفی، ممکن است نتایج متفاوتی نسبت به آنچه در اینجا نشان داده شده دریافت کنید. بیایید به مقادیر Q تمام اقدامات در این حالت نگاه کنیم، `(21,8,True)`:

```
df[124:126]
```

کد قبلی موارد زیر را چاپ می‌کند:

	state_action pair	value
۱۲٤	((21, 8, True), 0)	۰,۹۳٦۷۸۲
۱۲۵	((21, 8, True), 1)	

شکل ۴.۲۴: مقادیر Q حالت (۲۱, ۸, True)

همانطور که مشاهده می کنیم، ما حداکثر مقدار Q را برای عمل ۰ (توقف) در مقایسه با عمل ۱ (تداوم) داریم. بنابراین، ما عمل ۰ را در حالت `(21,8,True)` انجام می دهیم. به طور مشابه، با همین روش، می توانیم با انتخاب اقدامات در هر حالتی که حداکثر مقدار Q را دارند، خطمشی را استخراج کنیم.

در بخش بعدی، با یک روش **کنترل سیاست ناهمگاه** آشنا می شویم که از دو سیاست مختلف استفاده می‌کند.

کنترل مونت کارلوی سیاست ناهمگاه (یا نهان_سیاست)[۱]

مونت کارلو با سیاست ناهمگاه (نهان-سیاست) یکی دیگر از روش‌های جالب **کنترل مونت کارلو** است. در روش با سیاست–ناهمگاه از دو **سیاست** به نام **سیاست رفتار** (**سیاست واسط**)[۲] و **سیاست هدف** (**سیاست اصلی**)[۳] استفاده می‌کنیم. همانطور که از نام آن‌ها پیداست، ما با استفاده از سیاست واسط، رفتار می‌کنیم (تولید اپیزودها) و سعی می‌کنیم خطمشی دیگری به نام سیاست هدف یا سیاست اصلی را بهبود بخشیم.

در روش **خطمشی همگاه (عیان-سیاست)**، تنها یک پردینه را با استفاده از سیاست π تولید می‌کنیم و همان سیاست π را به طور مکرر بهبود می‌دهیم تا سیاست بهینه را پیدا کنیم. اما در روش **سیاست ناهمگاه (نهان-سیاست)**، ما یک پردینه را با استفاده از **سیاست واسط** به نام سیاست رفتار b ایجاد می‌کنیم و سعی می‌کنیم به طور مکرر، یک خطمشی دیگری به نام **سیاست اصلی** یا هدف π را بهبود بخشیم.

یعنی در روش با **خطمشی-همگاه** (عیان–سیاست) یاد گرفتیم که عامل با استفاده از خطمشی π یک پردینه تولید می‌کند. سپس برای هر مرحله از پردینه ، بازگشت زوج حالت–اقدام را محاسبه می‌کنیم و **تابع** Q مربوط به $Q(s_t, a_t)$، را به عنوان بازده متوسط محاسبه می‌کنیم، سپس از این تابع Q، یک سیاست جدید π استخراج می‌کنیم. ما این مرحله را به طور مکرر تکرار می‌کنیم تا خطمشی بهینه را پیدا کنیم.

اما در روش با **خطمشی ناهمگاه** (یا نهان–سیاست)، عامل با استفاده از یک سیاست واسط بنام **سیاست رفتار** b، یک پردینه تولید می‌کند. سپس برای هر مرحله از پردینه، بازده زوج حالت–اقدام را محاسبه می‌کنیم و تابع Q مربوط به $Q(s_t, a_t)$، را به عنوان بازده متوسط محاسبه می‌کنیم، سپس از این تابع Q، یک خطمشی جدید به نام **سیاست هدف** π استخراج می‌کنیم. ما این مرحله را به طور مکرر تکرار می‌کنیم تا سیاست هدف بهینه π را پیدا کنیم.

---

[۱] Off-Policy Monte Carlo Control
[۲] Behavior Policy
[۳] Target Policy

خط‌مشی رفتار (سیاست واسط) معمولاً بر روی خط‌مشی اپسیون حریصانه تنظیم می‌شود و بنابراین عامل با خط‌مشی اپسیلون حریصانه، محیط را بررسی کرده و یک پردینه تولید می‌کند. برخلاف خط‌مشی رفتار، در خط‌مشی هدف (سیاست اصلی) همواره سیاست ما حریصانه است و بنابراین خط‌مشی هدف همیشه بهترین اقدام را در هر حالت انتخاب می‌کند[۱].

اکنون بیایید بفهمیم که **روش مونت کارلوی سیاست-ناهمگاه** دقیقا چگونه کار می‌کند.

ابتدا **تابع** Q را با مقادیر تصادفی مقداردهی اولیه می‌کنیم. سپس با استفاده از **خط‌مشی رفتار (سیاست واسط)**، که خط‌مشی حریصانه اپسیلون است، یک اپیزود تولید می‌کنیم. یعنی از **تابع** Q بهترین اقدام (اقدامی که حداکثر ارزش Q را دارد) را با **احتمال** $1-\epsilon$ و اقدام تصادفی را با **احتمال** ϵ انتخاب می‌کنیم.

سپس برای هر مرحله از پردینه، بازگشت زوج حالت–اقدام را محاسبه می‌کنیم و **تابع** Q مربوط به $Q(s_t, a_t)$ را به عنوان بازده متوسط محاسبه می‌کنیم. به جای استفاده از میانگین حسابی برای محاسبه **تابع** Q، می‌توانیم از میانگین افزایش تدریجی استفاده کنیم. می‌توانیم **تابع** Q را با استفاده از میانگین افزایشی به صورت زیر محاسبه کنیم:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\big(R_t - Q(s_t, a_t)\big)$$

پس از محاسبه **تابع** Q، **سیاست هدف** π را با انتخاب یک اقدام که در هر حالت، بیشینه **ارزش** Q را دارد، استخراج می‌کنیم که به صورت زیر نشان داده شده است:

$$\pi(s_t) = \arg\max_a Q(s_t, a)$$

الگوریتم به صورت زیر ارائه شده است:

سیاست اول (سیاست رفتار) بنوعی گویی کاوه و اهلِ کاوستان است چون بدنبال کاویدن (و تفحص) هم هست، اما سیاست دوم (سیاست هدف) انگار کامران و اهلِ کامرانیه است چون تنها نگرش کامرانی (تمتع) دارد!! 😂😂

۱. **تابع** Q مربوط به $Q(s_t, a_t)$ را با مقادیر تصادفی (بختکی) راه‌اندازی کنید، خط‌مشی رفتار (**سیاست واسط**) b را برای حریصانه–اپسیلون بودن، تنظیم کنید، همچنین **سیاست هدف** π را برای حریصانه بودن، تنظیم کنید.

۲. برای M تعداد تکرار پردینه:

۱. یک پردینه (اپیزود) را با استفاده از **سیاست رفتار** b تولید کنید.

۲. **بازگشت** (**بازده**) R را برابر صفر قرار دهید.

۳. برای هر مرحله‌ی t در پردینه ، $t = T-۱, T-۲, \dots, ۰$:

۱. **بازگشت** $R = R + r_{t+۱}$ را محاسبه کنید.

۲. **ارزش** Q را به صورت $Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_t - Q(s_t, a_t))$ محاسبه کنید.

۳. **سیاست اصلی** (یا سیاست هدف) $\pi(s_t) = \arg\max_a Q(s_t, a)$ را محاسبه کنید.

۳. **خط‌مشی هدف** π را بدست آورید.

همانطور که از الگوریتم قبلی می‌توان مشاهده کرد، ابتدا ارزش‌های Q همه جفت‌های حالت–اقدام را برابر مقادیر تصادفی قرار می‌دهیم و سپس با استفاده از خط‌مشی رفتار، یک پردینه تولید می‌کنیم. سپس در هر مرحله از پردینه، تابع Q به‌روز شده (ارزش‌های Q) را با استفاده از میانگین افزایشی محاسبه کرده و سپس سیاست هدف را از تابع Q به‌روز شده استخراج می‌کنیم. همانطور که می‌توانیم متوجه شویم، در هر تکرار، تابع Q به طور مداوم در حال بهبود است و از آنجایی که ما خط‌مشی هدف را از تابع Q استخراج می‌کنیم، خط‌مشی هدف ما نیز در هر تکرار بهبود می‌یابد.

همچنین توجه داشته باشید که از آنجایی که این یک روش ناهمگاه (یا **نهان–سیاست**) است، پردینه با استفاده از خط‌مشی رفتار (یعنی واسط) تولید می‌شود و سعی می‌کنیم خط‌مشی هدف را بهبود بخشیم.

اما صبر کنید! اینجا یک مسئله کوچک وجود دارد. از آنجایی که ما در حال یافتن خط‌مشی هدف π از تابع Q هستیم که بر اساس اپیزودهای تولید شده توسط یک سیاست متفاوت به نام سیاست رفتار محاسبه شده است، بنابر این سیاست هدف ما نادرست خواهد بود. زیرا توزیع سیاست رفتار (واسط) و توزیع سیاست هدف (اصلی) متفاوت خواهند بود. بنابراین، برای اصلاح این موضوع، تکنیک جدیدی به نام **نمونه‌برداری اهمیت**[۱] را معرفی می‌کنیم. این روش، تکنیکی برای تخمین ارزش‌های یک توزیع در زمانی است که نمونه هایی از توزیع دیگری در اختیار داریم.

فرض کنید می‌خواهیم امیدریاضی تابع $f(x)$ را که در آن مقدار x از توزیع $p(x)$، نمونه‌برداری شده است، محاسبه کنیم. یعنی: ($x \sim p(x)$) را محاسبه کنیم، با این حساب می‌توانیم بنویسیم:

$$\mathbb{E}_{x\sim p(x)}[f(x)] = \int_x p(x)f(x)dx$$

با روش نمونه‌برداری اهمیت، انتظارات را با استفاده از توزیع متفاوت $q(x)$ تخمین می‌زنیم؛ یعنی به جای نمونه‌برداری x از $p(x)$ از توزیع متفاوت $q(x)$ به شکل زیر استفاده می‌کنیم:

$$\mathbb{E}[f(x)] \approx \int_x f(x)\frac{p(x)}{q(x)}q(x)dx$$
$$\mathbb{E}[f(x)] \approx \frac{1}{N}\sum_i f(x_i)\frac{p(x_i)}{q(x_i)}$$

نسبت $p(x)/q(x)$ را نسبت نمونه‌برداری اهمیت یا تصحیح اهمیت[۲] می‌گویند.

بسیار خب، نمونه‌برداری اهمیت چگونه به ما کمک می‌کند؟ ما آموختیم که با نمونه‌برداری اهمیت، می‌توانیم ارزش یک توزیع را با نمونه‌برداری از توزیع دیگر با استفاده از نسبت نمونه‌برداری اهمیت تخمین بزنیم. در کنترل خط‌مشی ناهمگاه (یا نهان–سیاست) می‌توانیم با استفاده از نسبت نمونه‌برداری اهمیت، خط‌مشی هدف را با نمونه‌ها (اپیزودها)

[۱] Importance Sampling
[۲] Importance Sampling Ratio or Importance Correction

از خطمشی رفتار تخمین بزنیم.

نمونه‌برداری اهمیت دو نوع دارد:

- **نمونه‌برداری اهمیت معمولی**[۱]
- **نمونه‌برداری اهمیت وزنی**[۲]

در نمونه‌برداری اهمیت معمولی، نسبت نمونه‌برداری اهمیت، نسبت **خطمشی هدف** به **خطمشی رفتار** خواهد بود $\frac{\pi(a|s)}{b(a|s)}$ ، ولی در نمونه‌برداری اهمیت وزنی، نسبت نمونه‌برداری اهمیت، نسبت وزنی **خطمشی هدف** به **خطمشی رفتار** خواهد بود $W\frac{\pi(a|s)}{b(a|s)}$.

حال بیایید بفهمیم که چگونه از نمونه‌برداری اهمیت وزنی در روش مونت‌کارلوی سیاست ناهمگاه استفاده می‌کنیم. فرض کنید W وزن باشد و $C(s_t, a_t)$ بیانگر مجموع تجمعی اوزان، در تمام اپیزودها باشد. ما یاد گرفتیم که تابع Q (ارزش‌های Q) را با استفاده از میانگین افزایشی به صورت زیر محاسبه می‌کنیم:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(R_t - Q(s_t, a_t))$$

اکنون، ما محاسبه تابع Q خود را با نمونه‌برداری اهمیت وزنی به شکل زیر تغییر می‌دهیم:

$$Q(s_t, a_t) = Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}(R_t - Q(s_t, a_t))$$

الگوریتم روش مونت کارلوی **سیاست-ناهمگاه** در ادامه نشان داده شده است. ابتدا یک پردینه را با استفاده از خطمشی رفتار (سیاست واسط) تولید می‌کنیم و سپس بازگشت R را برابر صفر و وزن W را برابر یک، مقداردهی می‌کنیم. سپس در هر مرحله از پردینه، بازده را محاسبه می‌کنیم و وزن تجمعی را به صورت $C(s_t, a_t) = C(s_t, a_t) +$

[۱] Ordinary Importance Sampling
[۲] Weighted Importance Sampling

W به‌روز می‌کنیم. پس از به روزرسانی تجمعی وزن‌ها، ارزش Q را به صورت زیر به‌روز می‌کنیم

$$Q(s_t, a_t) = Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}(R_t - Q(s_t, a_t)).$$

از ارزش Q، خطمشی هدف (سیاست–اصلی) را به صورت $\pi(s_t) = \arg\max_a Q(s_t, a)$ استخراج می‌کنیم. زمانی که اقدام s_t ارائه شده توسط خطمشی رفتار و خطمشی هدف یکسان نیست، حلقه را شکسته و پردینه بعدی را تولید می‌کنیم. در غیر این صورت وزن را به این صورت به‌روز می‌کنیم:

$$W = W\frac{1}{b(a_t|s_t)}$$

الگوریتم کامل روش **مونت کارلوی سیاست-ناهمگاه** (نهان–سیاست) در مراحل زیر توضیح داده شده است:

۱. **تابع** Q مربوط به $Q(s, a)$ را با مقادیر تصادفی مقداردهی کنید، **خطمشی رفتار** b را به صورت اپسیلون–حریصانه و **خطمشی هدف** π را به صورت روش حریصانه تنظیم کنید و وزن‌های تجمعی را به صورت $C(s, a) = 0$ مقداردهی کنید.

۲. برای M تعداد اپیزود:

۱. یک پردینه با استفاده از **خطمشی رفتار** b ایجاد کنید.

۲. **بازگشت** R را برابر صفر و وزن W را برابر یک، مقداردهی کنید.

۳. برای هر مرحله t در اپیزود، $t = T-1, T-2, \dots, 0$:

۱. **بازده** را به صورت $R = R + r_{t+1}$ محاسبه کنید.

۲. وزن‌های تجمعی $C(s_t, a_t) = C(s_t, a_t) + W$ را به روزرسانی کنید.

۳. **ارزش** Q را به این صورت به‌روزرسانی کنید:

$$Q(s_t, a_t) = Q(s_t, a_t) + \frac{W}{C(s_t, a_t)}\big(R_t - Q(s_t, a_t)\big)$$

۴. **خط‌مشی هدف** را محاسبه کنید: $\pi(s_t) = \arg\max_a Q(s_t, a)$

۵. اگر $a_t \neq \pi(s_t)$ آنگاه حلقه را بشکنید.

۶. وزن‌ها را به صورت $W = W \frac{۱}{b(a_t|s_t)}$ به‌روزرسانی می‌کنیم.

۳. سیاست هدف π را بدست آورید.

یادداشت مترجم

نسبت نمونه گیری اهمیت (نسبت **IS**)، که همچنین به عنوان تصحیح اهمیت شناخته می‌شود، یک مفهوم کلیدی در یادگیری تقویتی و روشهای مونت کارلو، به ویژه در یادگیری ناهمگاه است. برای تنظیم احتمالات مسیرهای نمونه‌برداری شده از یک سیاست رفتاری (π_b) برای مطابقت با مسیرهای مشاهده شده تحت یک سیاست هدف (π_t) استفاده می شود.

تعریف

نسبت نمونه گیری اهمیت (نسبت **IS**) برای یک مسیر معین (یا گام زمانی) به صورت زیر تعریف می شود:

$$\rho_t = \frac{\pi_t(a|s_t)}{\pi_b(a|s_t)}$$

که در آن:

$\pi_t(a_t|s_t)$ = احتمال اقدام a_t در حالت s_t تحت **سیاست هدف** است.

$\pi_b(a_t|s_t)$ = احتمال اقدام a_t در حالت s_t تحت **سیاست رفتار** است.

فواید

– اصلاح عدم تطابق بین سیاست رفتار (که برای اکتشاف استفاده می‌شود) و سیاست هدف (در حال یادگیری) را تصحیح می کند.

– امکان وزن دهی مجدد بازده‌ها با سیاست رفتار یعنی π_b برای تخمین مقادیر مورد انتظار تحت سیاست هدف π_t

کاربرد

۱. روشهای سیاست ناهمگاه مونت کارلو:

نسبت **IS** تجمعی برای یک مسیر به شرح زیر است:

$$\rho_{:T} = \prod_{t=0}^{T} \frac{\pi_t(a|s_t)}{\pi_b(a|s_t)}$$

که برای وزندهی بازده هنگام محاسبه توابع مقدار استفاده می‌شود.

۲. یادگیری تفاوت زمانمند با سیاست ناهمگاه (**TD**):

– در الگوریتم هایی مانند Q-learning، نمونه گیری اهمیت، هنگام استفاده از اقدامات اکتشافی، به‌روزرسانیها را تنظیم می‌کند.

۳. بهینه سازی سیاست مجاورتی[۱] (**PPO**):

– از نسبتهای اهمیت برش‌خورده[۲] برای تثبیت به‌روزرسانیهای سیاست استفاده می‌کند.

خواص

[۱] Proximal Policy Optimization (PPO)
[۲] Clipped Importance Ratios

– برآورد بی‌طرفانه یا بی‌سویش: اطمینان حاصل می‌کند که انتظارات تحت سیاست هدف یعنی π_t حفظ می‌شوند.

– واریانس بالا: اگر π_b و π_t به طور قابل توجهی متفاوت باشند، این نسبت می‌تواند بی‌اندازه بزرگ شود و منجر به یادگیری ناپایدار شود.

– روشهای کوتاه‌شده[۱]: تکنیکهایی مانند پیش–تصمیم **IS** یا **IS** وزندار به کاهش واریانس کمک می‌کنند.

مثالی در یادگیری با سیاست ناهمگاه

گیریم:

– π_t یک سیاست قطعی باشد (که در آن، همیشه عمل a انجام می‌شود)

– π_t یک سیاست اکتشافی باشد (که در آن عمل a را با احتمال ۰.۵ می گیرد).

– مسیر مشاهده شده بصورت روبرو است: $(s_t, a, r_{t+1}, s_{t+1})$

نسبت **IS** برای این مرحله به شرح زیر است:

$$\rho_t = \frac{\pi_t(a|s_t)}{\pi_b(a|s_t)} = \frac{۱.۰}{۰.۵} = ۲.۰$$

این بدان معناست که بازده حاصل از این مرحله، باید **دو برابر** شود تا احتمال پایین انتخاب عمل a تحت سیاست π_b جبران گردد.

نتیجه‌گیری این بخش

نسبت نمونه گیری اهمیت در **RL** با سیاست ناهمگاه، بسیار مهم است و امکان یادگیری از داده‌های اکتشافی را فراهم

[۱] Truncation Methods

کرده و در عین حال اختلافات سیاستها را اصلاح می‌کند. با این حال ، باید مراقب بود که واریانس آن، اغلب از طریق تکنیکهای برش یا نرمال‌سازی، مدیریت شود.

پایان یادداشت

آیا روش مونت‌کارلو برای همه کارها (وظایف) قابل استفاده است؟

ما آموختیم که مونت‌کارلو یک روش بدون مدل است، و بنابراین برای یافتن سیاست بهینه، به پویایی مدل محیط برای محاسبه تابع ارزش و تابع Q نیاز ندارد. روش مونت‌کارلو، تابع ارزش و تابع Q را به ترتیب با گرفتن میانگین بازده حالت و میانگین بازگشت زوج حالت–اقدام محاسبه می‌کند.

اما یک مسئله در مورد روش مونت‌کارلو این است که فقط برای کارهای پردینه‌ای (اپیزودیک) قابل استفاده است. ما یاد گرفتیم که در روش مونت کارلو، با گرفتن میانگین بازده حالت، ارزش حالت را محاسبه می‌کنیم و بازده مجموع پاداش‌های اپیزود است. اما وقتی اپیزودی وجود ندارد، یعنی اگر وظیفه ما یک کار پیوسته (وظیفه غیرپردینه‌ای) باشد، نمی‌توانیم روش مونت‌کارلو را اعمال کنیم.

خب، چگونه ارزش حالتی را که در آن یک وظیفه پیوسته داریم و همچنین در جایی که پویایی‌های مدل محیط را نمی‌دانیم، محاسبه کنیم؟ اینجا، جایی است که ما از روش جالب بدون مدل دیگری به نام **یادگیری تفاضل زمانی** یا **تفاوت زمانمند**[۱] استفاده می‌کنیم. در فصل بعدی، دقیقاً یاد خواهیم گرفت که یادگیری تفاوت زمانی چگونه کار می‌کند.

[۱] Temporal Difference (TD) Learning

خلاصه

ما فصل را با درک اینکه روش مونت‌کارلو چیست، آغاز کردیم. ما آموختیم که در روش مونت‌کارلو، امیدریاضی یک متغیر تصادفی را با نمونه‌گیری تقریب می‌زنیم و زمانی که اندازه نمونه بیشتر باشد، تقریب بهتر خواهد بود. سپس با وظایف **پیش‌بینی (احصاء)** و **کنترل (وارسی)** آشنا شدیم. در وظیفه **پیش‌بینی**، خط‌مشی داده‌شده را با **پیش‌بینی تابع ارزش** یا **تابع** Q ارزیابی می‌کنیم، که به ما کمک می‌کند تا بازده مورد انتظاری یک عامل در صورت استفاده از خط‌مشی معینی را درک کنیم. در وظیفه **کنترل**، هدف ما یافتن **خط‌مشی** بهینه است و هیچ **سیاستی** به عنوان ورودی به ما داده نخواهد شد، بنابراین با مقداردهی اولیه یک خط‌مشی تصادفی (بختکی) شروع می‌کنیم و سعی می‌کنیم خط‌مشی بهینه را به صورت تکراری پیدا کنیم.

با حرکت رو به جلو، یاد گرفتیم که چگونه از روش مونت کارلو برای انجام وظیفه پیش‌بینی استفاده کنیم. ما آموختیم که ارزش یک حالت و ارزش یک زوج حالت–اقدام را می‌توان به ترتیب با گرفتن میانگین بازده حالت و میانگین بازگشت زوج حالت–اقدام در چندین اپیزود، محاسبه کرد.

همچنین با **روش مونت‌کارلوی اولین بازدید** و **روش مونت‌کارلوی هر بازدید** آشنا شدیم. در مونت‌کارلوی اولین بازدید، بازده را فقط برای اولین باری که از حالت در اپیزود بازدید می‌شود، محاسبه می‌کنیم، و در مونت‌کارلوی هر بازدید، بازده را هر بار که حالت در اپیزود، بازدید می‌شود، محاسبه می‌کنیم.

در ادامه، نحوه انجام یک **وظیفه کنترلی** با استفاده از **روش مونت کارلو** را بررسی کردیم. ما در مورد دو نوع مختلف از روش‌های کنترلی: **کنترل با سیاست همگاه (عیان)** و **کنترل با سیاست ناهمگاه (نهان)** آموختیم.

در روش با سیاست همگاه، ما با استفاده از یک خط‌مشی، پردینه‌ها را تولید می‌کنیم و همچنین همان خط‌مشی را به طور مکرر برای یافتن خط‌مشی بهینه بهبود می‌دهیم. ما برای اولین بار در مورد روش «**شروع کاوشی مونت‌کارلو**» یاد گرفتیم که در آن همه زوج‌های اقدام–حالت را روی یک احتمال غیرصفر تنظیم کردیم تا از کاوش کامل، اطمینان

حاصل کنیم. در ادامه، در مورد کنترل مونت کارلو با یک **سیاست اپسیلون حریصانه** آشنا شدیم که در آن با احتمال اپسیلون، یک اقدام تصادفی (اکتشاف) و با احتمال (اپسیلون –۱)، بهترین اقدام (انتفاع) را انتخاب می‌کنیم.

در پایان فصل، روش **کنترل مونت کارلوی سیاست-ناهمگاه** را مورد بحث قرار دادیم که در آن از دو سیاست متفاوت به نام **خط‌مشی رفتار** برای تولید اپیزود و **خط‌مشی هدف** برای یافتن سیاست بهینه استفاده می‌کنیم.

سوالات

بیایید دانش خود را در مورد روش‌های مونت کارلو را با پاسخ به سوالات زیر، ارزیابی کنیم:

۱. روش مونت‌کارلو چیست؟
۲. چرا روش مونت‌کارلو بر برنامه‌ریزی پویا ترجیح داده می‌شود؟
۳. وظایف **پیش‌بینی (احصاء)** چه تفاوتی با وظایف **کنترلی (وارسی)** دارد؟
٤. روش پیش‌بینی مونت‌کارلو، چگونه تابع ارزش را پیش‌بینی می‌کند؟
٥. تفاوت بین **مونت‌کارلوی اولین بازدید** و **مونت‌کارلوی هر بازدید** چیست؟
٦. چرا از به‌روزرسانی‌های میانگین افزایشی استفاده می‌کنیم؟
۷. تفاوت کنترل با **خط‌مشی همگاه (عیان-سیاست)** و کنترل با **خط‌مشی ناهمگاه (نهان-سیاست)** چیست؟
۸. **سیاست اپسیلون-حریصانه** چیست؟

پیوست پردینه فصل چهار

بازی با سرجمع ۲۱

بلک‌جک یا بیست‌ویک یک بازی کارتی است. بلک‌جک، یکی از محبوب‌ترین بازی‌های کارتی است که هدف آن رسیدن به مجموع امتیازی نزدیک‌تر به ۲۱ نسبت به دیلر، بدون تجاوز از این عدد، است. در این بازی، کارت‌های عددار به ارزش عددی خود و کارت‌های حرفی (سرباز، بی بی، شاه) هر کدام به ارزش ۱۰ و آس نیز می‌تواند به ارزش ۱ یا ۱۱ محسوب شود.

بازی بلک‌جک با دریافت دو کارت اولیه آغاز می‌شود و بازیکن می‌تواند با دریافت کارت‌های بیشتر، مجموع امتیاز خود را افزایش دهد. اگر مجموع امتیاز بازیکن از ۲۱ بیشتر شود، او باخته است. پس از اینکه همه بازیکنان تصمیم خود را گرفتند، دیلر کارت‌های خود را رو می‌کند و سعی می‌کند به مجموعی نزدیک به ۲۱ برسد. اگر دیلر از ۲۱ بگذرد، تمام بازیکنانی که هنوز در بازی هستند برنده می‌شوند.

بلک‌جک یک بازی ترکیبی از شانس و مهارت است. اگرچه نتیجه بازی تا حد زیادی به کارت‌هایی که به بازیکن داده می‌شود بستگی دارد، اما تصمیم‌گیری‌های استراتژیک بازیکن نیز نقش مهمی در موفقیت او ایفا می‌کند. بسیاری از بازیکنان از سیستم‌های کارت شماری برای افزایش شانس برد خود استفاده می‌کنند، اما این روش‌ها در بسیاری از بازیها ممنوع است.

تاریخچه

پیش‌درآمد بلک جک، بازی بیست و یک بوده که خود خاستگاهی ناشناخته دارد. اولین مرجع مکتوب در این زمینه به کتابی از نویسنده شهیر اسپانیایی میگوئل د سروانتس[۱] بازمی‌گردد که بیشتر به خاطر نوشتن رمان دن کیشوت معروف

[۱] Miguel De Cervantes

است. سروانتس خود یک از علاقه‌مندان قمار بود و شخصیت‌های اصلی داستان رینکونت و کورتادیلو در «رمان‌های سرمشق» چند بازیکن متقلب بودند که در شهر سویل کار می‌کردند.

آنها در تقلب در بازی ونتیونا (معادل اسپانیایی بیست و یک) مهارت داشتند و رمان بیان می‌کند که هدف از این بازی، رسیدن به امتیاز ۲۱ بدون فراتر رفتن از آن است و ارزش آس معادل ۱ یا ۱۱ است. بازی با دست ورق‌های اسپانیای باراخا[1] انجام می‌شود که فاقد کارت‌های هشت و نه و ده است. این داستان کوتاه بین سال‌های ۱۶۰۱ و ۱۶۰۲ نوشته شده، بدین ترتیب می‌توان نتیجه گرفت که ونتیونا از آغاز قرن هفدهم یا قبل از آن در کاستیا بازی می‌شده است. ارجاعات بعدی به این بازی در فرانسه و اسپانیا دیده می‌شود.

هنگامی که بیست و یک در ایالات متحده معرفی شد، خانه‌های شرط‌بندی برای جلب علاقه بازیکنان اقدام به پرداخت بونوس‌هایی می‌کردند. یکی از این بونوس‌ها، پرداخت ده به یک بود که در صورت وجود دستی شامل آس پیک و سرباز سیاه (سرباز پیک یا خاج) فعال می‌شد.

این دست «بلک جک» نامیده می‌شد و به مرور زمان این نام روی بازی ماندگار شد، حتی با وجودی‌که بونوس ده به یک به زودی برداشته شد. در بازی مدرن، بلک جک به هر دستی شامل یک آس به اضافه یک کارت ده یا عکس، صرف نظر از خال یا رنگ اشاره دارد.

نحوه بازی

هدف این بازی، دستیابی به امتیاز ۲۱ یا نزدیک‌ترین امتیاز به آن است، بدون اینکه از این عدد تجاوز شود. ارزش کارت‌ها به این صورت است: کارت‌های ۲ تا ۱۰ مطابق عدد خود امتیاز دارند، کارت‌های عکس‌دار (شاه، بی‌بی، سرباز) هر کدام ۱۰ امتیاز دارند و آس می‌تواند ۱ یا ۱۱ امتیاز باشد، بسته به اینکه کدام مقدار برای بازیکن بهتر است. بازی بین بازیکنان و دیلر (پخش‌کننده) انجام می‌شود و هدف اصلی، شکست دادن دیلر است، نه رقابت مستقیم با سایر بازیکنان.

در ابتدای بازی، هر بازیکن دو کارت دریافت می‌کند که معمولاً به رو هستند، در حالی که دیلر یک کارت به رو و یک کارت به پشت می‌گیرد. بازیکنان پس از دیدن کارت‌های خود، می‌توانند تصمیم بگیرند که کارت بیشتری بگیرند (Hit) یا دست خود را ثابت نگه دارد (Stand). هدف این است که دست بهینه‌ای تشکیل دهند که یا دقیقاً ۲۱

[1] Baraja

امتیاز داشته باشد یا به آن نزدیک‌تر باشد، اما اگر مجموع امتیاز کارت‌های بازیکن از ۲۱ فراتر رود، بازیکن «Bust» شده و به‌طور خودکار بازنده می‌شود.

اگر بازیکنی در همان دو کارت ابتدایی به امتیاز ۲۱ برسد (معمولاً با یک آس و یک کارت عکس‌دار)، دست او به عنوان «بلک‌جک» شناخته می‌شود و معمولاً یک برد فوری محسوب می‌شود، مگر اینکه دیلر نیز بلک‌جک داشته باشد که در این صورت بازی مساوی است. در موارد دیگر، بازی ادامه می‌یابد و دیلر نیز باید پس از پایان نوبت بازیکنان، دست خود را بازی کند. قوانین معمولاً دیلر را ملزم می‌کنند که تا زمانی که مجموع کارت‌های او کمتر از ۱۷ است، کارت بکشد و در ۱۷ یا بیشتر بایستد.

بلک‌جک همچنین شامل گزینه‌های استراتژیک دیگری است. برای مثال، بازیکن ممکن است بتواند شرط خود را دو برابر کند (Double Down) و تنها یک کارت اضافی بگیرد، یا اگر دو کارت یکسان داشته باشد، آن‌ها را به دو دست جداگانه تقسیم (Split) کند. علاوه بر این، در برخی نسخه‌ها، بازیکن می‌تواند در صورت داشتن دست نامطلوب، شرط خود را تسلیم کند (Surrender) و بخشی از شرط خود را بازپس گیرد. این تصمیمات باید با دقت و بر اساس کارت‌های خود بازیکن و کارت‌های قابل مشاهده دیلر گرفته شود.

در پایان هر دست، برنده تعیین می‌شود. اگر امتیاز بازیکن بیشتر از دیلر باشد، یا اگر دیلر Bust شود، بازیکن برنده است و معمولاً به نسبت ۱:۱ جایزه می‌گیرد. اگر بازیکن بلک‌جک داشته باشد، معمولاً به نسبت ۳:۲ پرداخت می‌شود. اما اگر دیلر برنده شود یا بازی مساوی شود، بازیکن شرط خود را از دست می‌دهد یا پس می‌گیرد. در نهایت، ترکیبی از شانس و استراتژی است که بلک‌جک را به یکی از جذاب‌ترین بازی‌های کارتی تبدیل کرده است.

استراتژی اساسی بازی

استراتژی اساسی بلک‌جک مجموعه‌ای از قوانین و تصمیم‌گیری‌های بهینه است که بر اساس احتمال‌ها و ریاضیات طراحی شده‌اند تا بازیکن شانس بیشتری برای برد در مقابل دیلر داشته باشد. این استراتژی به شما می‌گوید که در هر شرایطی (با توجه به کارت‌های شما و کارت دیلر) چه کاری بهتر است انجام دهید، مانند گرفتن کارت (Hit)، ثابت ماندن (Stand)، تقسیم کردن دست (Split)، یا دو برابر کردن شرط (Double Down). هدف این است که با پیروی از این استراتژی، مزیت خود را به حداکثر برسانید.

اولین اصل استراتژی پایه این است که تصمیم‌گیری شما باید بر اساس ارزش دست شما و کارت به‌روی دیلر باشد. برای مثال، اگر مجموع کارت‌های شما ۱۲ باشد و کارت دیلر بین ۴ تا ۶ باشد، بهترین تصمیم ایستادن (Stand) است، زیرا احتمال Bust شدن دیلر در این حالت زیاد است. اما اگر کارت دیلر ۷ یا بالاتر باشد، شما باید کارت دیگری بگیرید (Hit)، زیرا احتمال دارد دیلر دست قوی‌تری داشته باشد.

یکی از نکات کلیدی در استراتژی پایه این است که همیشه وقتی دو کارت اول شما آس و ۸ هستند، دست خود را تقسیم کنید. اما هرگز دو ۱۰ یا دو ۵ را تقسیم نکنید. دلیل این امر این است که دو ۱۰ برابر ۲۰ امتیاز است که دست بسیار قوی‌ای محسوب می‌شود، و دو ۵ برای دو برابر کردن شرط مناسب است. همچنین اگر دیلر کارت ضعیفی مانند ۶ داشته باشد، تقسیم کردن کارت‌ها می‌تواند شانس برد شما را افزایش دهد.

در استراتژی پایه، دو برابر کردن شرط نیز اهمیت زیادی دارد. این کار زمانی توصیه می‌شود که شانس زیادی برای بهبود دست خود داشته باشید. برای مثال، اگر مجموع دست شما ۱۱ باشد و کارت دیلر ۶ باشد، بهترین گزینه دو برابر کردن شرط است، زیرا احتمال زیادی وجود دارد که کارت بعدی شما ۱۰ باشد و به ۲۱ برسید.

در نهایت، یکی دیگر از قوانین مهم این است که اگر دیلر یک کارت آس داشته باشد و به شما گزینه بیمه[۱]() داده شود، هرگز بیمه نگیرید. بیمه شرطی است که در بلندمدت به نفع ما است. به‌طور کلی، استراتژی پایه یک ابزار بسیار مفید است، اما نیاز به تمرین و حافظه دارد تا بتوانید آن را به‌طور مؤثر در بازی به کار ببرید.

استراتژی اساسی بازی در جدول زیر نشان داده شده است.

Player hand	Dealer's face-up card									
	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	A

[۱] Insurance

Hard totals (excluding pairs)										
۱۸–۲۱	S	S	S	S	S	S	S	S	S	S
۱۷	S	S	S	S	S	S	S	S	S	Us
۱٦	S	S	S	S	S	H	H	Uh	Uh	Uh
۱۵	S	S	S	S	S	H	H	H	Uh	Uh
۱۳–۱٤	S	S	S	S	S	H	H	H	H	H
۱۲	H	H	S	S	S	H	H	H	H	H
۱۱	Dh	Dh	Dh	Dh	Dh	Dh	Dh	Dh	Dh	Dh
۱۰	Dh	Dh	Dh	Dh	Dh	Dh	Dh	Dh	H	H
۹	H	Dh	Dh	Dh	Dh	H	H	H	H	H
۵–۸	H	H	H	H	H	H	H	H	H	H

Soft totals										
	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	A
A,9	S	S	S	S	S	S	S	S	S	S
A,8	S	S	S	S	Ds	S	S	S	S	S
A,7	Ds	Ds	Ds	Ds	Ds	S	S	H	H	H
A,6	H	Dh	Dh	Dh	Dh	H	H	H	H	H
A,4–A,5	H	H	Dh	Dh	Dh	H	H	H	H	H
A,2–A,3	H	H	H	Dh	Dh	H	H	H	H	H
A,A	H	H	H	H	Dh	H	H	H	H	H
Pairs										
	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	A
A, A	SP	SP	SP	SP	SP	SP	SP	SP	SP	SP

۱۰,۱۰	S	S	S	S	S	S	S	S	S	S
۹,۹	SP	SP	SP	SP	SP	S	SP	SP	S	S
۸,۸	SP	SP	SP	SP	SP	SP	SP	SP	SP	Usp
۷,۷	SP	SP	SP	SP	SP	SP	H	H	H	H
٦,٦	SP	SP	SP	SP	SP	H	H	H	H	H
٥,٥	Dh	Dh	Dh	Dh	Dh	Dh	Dh	Dh	H	H
٤,٤	H	H	H	SP	SP	H	H	H	H	H
۲,۲–۳,۳	SP	SP	SP	SP	SP	SP	H	H	H	H

S = Stand
H = Hit
Dh = Double (if not allowed, then hit)
Ds = Double (if not allowed, then stand)
SP = Split
Uh = Surrender (if not allowed, then hit)
Us = Surrender (if not allowed, then stand)
Usp = Surrender (if not allowed, then split)

فصل پنجم

یادگیری تفاوت زمانمند

یادگیری تفاوت زمانمند یا تفاضل زمانی[۱] (TD) یکی از محبوب‌ترین و پرکاربردترین روش‌های **بدون مدل** است. دلیل این امر این است که یادگیری تفاوت زمانمند، مزایای هر دو روش **برنامه‌ریزی پویا** و روش **مونت‌کارلو** را که در فصل‌های قبل به آن پرداختیم، ترکیب می‌کند.

ما این فصل را با درک اینکه چگونه یادگیری تفاوت زمانمند (یا مابه‌التفاوت زمانمند) دقیقاً در مقایسه با روش‌های برنامه‌ریزی پویا و مونت‌کارلو مفید است، آغاز خواهیم کرد. بعداً یاد خواهیم گرفت که چگونه کار **پیش‌بینی (احصاء)** را با استفاده از یادگیری تفاوت‌زمانمند انجام دهیم. در ادامه، نحوه انجام وظایف ***کنترل (وارسی)*** تفاوت زمانمند را با یک **روش کنترل تفاوت زمانمند با سیاست همگاه**[۲] به نام **سارسا**[۳] و یک **روش کنترل تفاوت زمانمند با سیاست ناهمگاه** به نام **یادگیری Q**، خواهیم آموخت.

ما همچنین خواهیم آموخت که چگونه با استفاده از ***سارسا*** و روش ***یادگیری*** Q، خط‌مشی بهینه را در محیط دریاچه یخ‌زده پیدا کنیم. در پایان فصل، روش‌های برنامه‌ریزی پویا، مونت‌کارلو و تفاوت زمانمند را با هم مقایسه می‌کنیم.

بنابراین در این فصل با موضوعات زیر آشنا می شویم:

- **یادگیری تفاوت زمانمند (یا تفاضل زمانی)[۴]**
- **روش پیش‌بینی تفاوت زمانمند[۵]**
- **روش کنترل تفاوت زمانمند[۶]**
- **کنترل تفاوت زمانمند با سیاست همگاه – سارسا[۷]**
- **کنترل تفاوت زمانمند با سیاست ناهمگاه – یادگیری Q[۸]**
- **پیاده‌سازی سارسا و یادگیری Q برای یافتن خط‌مشی بهینه**
- **تفاوت بین یادگیری Q و سارسا**
- **مقایسه روش‌های برنامه‌ریزی پویا، مونت کارلو و تفاوت زمانمند**

---

[۱] Temporal Difference
برای کلمات این ترکیب می‌توان معادلهای متفاوتی بکار گرفت. مثلا برای واژه اول معادلهای همچون: تفاوت، تفاضل، مابه‌التفاوت، مازاد، تمایز، فرق و امثال آنها و برای واژه دوم معادلهای همچون: زمانمند، وابسته به زمان، زمانی، موقتی، زمانی ، گذرا و امثال آنها را استفاده کرد.

[۲] On-Policy TD Control Method
[۳] SARSA (State, Action, Reward, State, Action)
[۴] TD Learning
[۵] TD Prediction Method
[۶] TD Control Method
[۷] On-Policy TD Control – SARSA
[۸] Off-Policy TD Control – Q Learning

یادگیری تفاوت زمانمند (تفاضل زمانی)

الگوریتم یادگیری تفاوت زمانمند یا **تفاضل زمانی** (یا تفاوتها در گذار زمان) توسط ریچارد ساتُن[۱] در سال ۱۹۸۸ معرفی شد. در مقدمه همین فصل متوجه شدیم که دلیل محبوبیت روش تفاوت زمانمند آنست که **این روش، ترکیبی از مزایای روش برنامه‌ریزی پویا و مونت کارلو است.** اما آن مزایا چیست؟

ابتدا اجازه دهید مزایا و معایب برنامه‌ریزی پویا و روش مونت کارلو را به سرعت مرور کنیم.

الف) برنامه‌ریزی پویا (**DP**)— مزیت روش برنامه‌ریزی پویا این است که از **معادله بلمن** برای محاسبه ارزش یک حالت استفاده می‌کند. یعنی آموخته‌ایم که طبق معادله بلمن، **ارزش یک حالت را می‌توان به عنوان مجموع پاداش فوری بعلاوه ارزش تنزیل شده حالات بعدی به دست آورد.** به این مفهوم، **خود–برپایی** (یا **خود–ابتنائی**)[۲] می‌گویند. یعنی برای محاسبه ارزش یک حالت، لازم نیست تا پایان اپیزود منتظر بمانیم، در عوض، با استفاده از معادله بلمن، می‌توانیم ارزش یک حالت را فقط بر اساس ارزش حالت بعدی تخمین بزنیم، به این کار، تکنیک خودآوری (خود–ابتنائی یا بوت‌استرپینگ) می‌گویند.

به یاد دارید که چگونه ***تابع ارزش*** را در روش‌های برنامه‌ریزی پویا (***تکرار ارزش*** و ***تکرار سیاست***) تخمین زدیم؟ ما ***تابع ارزش*** (ارزش یک حالت) را به صورت زیر تخمین زدیم:

$$V(s) = \sum_{s'} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma V(s')\right]$$

همانطور که ممکن است به خاطر داشته باشید، در **DP** ما یاد گرفتیم که برای یافتن ارزش یک حالت، لازم نیست تا پایان اپیزود منتظر بمانیم. در عوض، ما از تکنیک خودآوری یا خود–ابتنائی استفاده می‌کنیم، که در آن، با تخمین ارزش حالت بعدی یعنی $V(s')$، ارزش حالت فعلی، یعنی $V(s)$ را تخمین می‌زنیم.

با این حال، نقطه ضعف برنامه‌ریزی پویا (**DP**) این است که روش **DP** تنها زمانی قابل اعمال است که **پویایی‌های مدل محیط** را بدانیم. یعنی برنامه‌ریزی پویا، یک روش مبتنی بر مدل است و برای استفاده از آن، باید **احتمال گذار** را بدانیم. وقتی پویایی‌های مدل محیط ندانیم، نمی‌توانیم روش برنامه‌ریزی پویا را اعمال کنیم.

---

[۱] Richard S. Sutton

[۲] Bootstrapping

بعنوان معادل فارسی این واژه می‌توان به خودراه‌اندازی، خودآوری، خودابتنائی، ابتناگری، بازآوری، خودراهبری، خودآغازی، خودبرپایی، برپاگری، خودبپایی، خودسازی، خودآیی، خودبنایی، و امثال آن اشاره کرد.

یادداشت مترجم: تاریخچه کلمه Bootstrap چیست؟

عبارت Bootstrap به تنهایی به معنی "خود راه انداز" بوده و کلمه Bootstraping به معنی راه اندازی یک فرآیند بصورت مستقل و بدون استفاده از منابع خارجی می باشد. این عبارت در علم کامپیوتر کمی کوتاهتر شده و با عنوان Booting بکار می‌رود، که نشان دهنده فرآیند راه اندازی سیستم و واردسازی اطلاعات اولیه نرم افزار در حافظه کامپیوتر می‌باشد. این تکنیک هم اکنون در مباحث مربوط به طراحی وب و کاربردهای آن اهمیت یافته است.

کلمه Boot در زبان انگلیسی به معنی پوتین یا چکمه می باشد، برخی از پوتین‌ها **زبانهٔ کوچکی** در انتهای خود دارند که هنگام پوشیدن پوتین برای قرارگرفتن بهتر پا در آن، استفاده می‌شود (چیزی شبیه به پاشنه کش). این زبانه با نام Bootstrap شناخته می‌شود.

در برنامه‌ریزی پویا، این اصطلاح به تکنیکی آماری اشاره دارد که برای تخمین پارامترهای یک جمعیت یا مدل از طریق نمونه‌برداری با جایگزینی از داده‌ها استفاده می‌شود. این روش معمولاً برای ارزیابی عملکرد مدل‌های یادگیری ماشین یا تخمین آماره‌ها در مسائل پیچیده به کار می‌رود.

کاربرد خود–راه‌اندازی در برنامه‌ریزی پویا

خود–راه‌اندازی یا خود–ابتنایی در برنامه‌ریزی پویا می‌تواند به عنوان یک ابزار مکمل برای تخمین پارامترها یا ارزیابی مدل‌ها استفاده شود. این روش شامل مراحل زیر است:

نمونه‌برداری با جایگزینی: داده‌های موجود چندین بار با جایگزینی نمونه‌برداری می‌شوند تا مجموعه‌ای از نمونه‌های جدید ایجاد شود.

محاسبه آماره‌ها: بر اساس نمونه‌های جدید، آماره‌هایی مانند میانگین، انحراف معیار و غیره محاسبه می‌شوند.

تکرار فرآیند: این فرآیند چندین بار تکرار می‌شود تا تخمین‌های قابل اعتمادی از پارامترها یا عملکرد مدل به دست آید.

استفاده از نتایج: نتایج حاصل از Bootstrap می‌توانند به عنوان ورودی برای تصمیم‌گیری در برنامه‌ریزی پویا استفاده می‌شوند.

مزایا

امکان ارائه بازه‌های اطمینان برای تخمین‌ها.

کاهش وابستگی به فرضیات توزیعی داده‌ها.

مناسب برای داده‌های کوچک یا پیچیده.

پایان یادداشت

ب) روش مونت کارلو (MC)– مزیت روش مونت کارلو این است که یک **روش بدون مدل** است، به این معنی که برای تخمین **تابع ارزش** و **تابع** Q، نیازی به شناخت پویایی‌های مدل محیط نیست.

با این حال، عیب روش مونت‌کارلو این است که در **MC** برای تخمین **ارزش** حالت یا **ارزش** Q **باید تا پایان اپیزود صبر کنیم** و اگر اپیزودی طولانی باشد، زمان زیادی برای ما خواهد داشت. همچنین، نمی‌توانیم روش‌های مونت کارلو را برای وظایف پیوسته (کارهای غیر اپیزودیک) اعمال کنیم.

ج) یادگیری تفاوت زمانمند (TD)– حال بیایید به **روش تفاضل زمانی** برگردیم. الگوریتم یادگیری تفاوت زمانمند (**TD**) از مزایای روشهای برنامه‌ریزی پویا (**DP**) و مونت کارلو (**MC**) بهره می‌برد. بنابراین، درست مانند برنامه‌ریزی پویا، از **خود-ابتنائی یا خود-برپایی بهره می‌برد** تا برای محاسبه **ارزش حالت** یا **ارزش** Q تا پایان یک اپیزود، منتظر نمانیم؛ و درست مانند روش مونت کارلو، **یک روش بدون مدل است** و درنتیجه برای محاسبه **ارزش** حالت یا **ارزش** Q به پویایی‌های مدل محیط نیازی ندارد. اکنون که ایده اصلی در پشت الگوریتم یادگیری تفاوت زمانمند (**TD**) را شناختیم، بیایید وارد جزئیات شویم و دقیقاً نحوه عملکرد آن را یاد بگیریم.

مشابه آنچه در فصل چهار، «روش‌های مونت کارلو»، آموختیم، می‌توانیم از الگوریتم یادگیری تفاوت زمانمند برای وظایف **پیش‌بینی** و **کنترل** استفاده کنیم و بنابراین می‌توانیم یادگیری تفاوت زمانمند را به دسته‌های زیر تقسیم کنیم:

- **انجام پیش‌بینی در روش تفاوت زمانمند**
- **انجام کنترل در روش تفاوت زمانمند**

معنی روش‌های **پیش‌بینی** و **کنترل** را در فصل قبل آموختیم. بیایید قبل از اینکه پیشتر برویم، آنها را کمی مرور کنیم.

در روش **پیش‌بینی (احصاء)**، یک **خط‌مشی** به عنوان ورودی داده می‌شود و سعی می‌کنیم با استفاده از **سیاست** داده شده، **تابع ارزش** یا **تابع** Q را پیش‌بینی کنیم. وقتی **تابع ارزش** را با استفاده از **خط‌مشی** مورد نظر پیش‌بینی می‌کنیم، بدین معناست که بدنبال آنیم که با توجه حالتی که عامل در آن است، اگر از **سیاست** داده شده استفاده شود، تا چه حد خوب است. یعنی

می‌توان گفت که یک عامل در هر حالتی که طبق **سیاست** داده شده عمل کند، چه **بازدهی** مورد انتظاری را می‌تواند ایجاد کند.

در **روش کنترل (وارسی)**، هیچ سیاستی به عنوان ورودی به ما داده نمی‌شود و هدف در روش **کنترل**، یافتن **سیاست** بهینه است. بنابراین، یک **سیاست** تصادفی را مقداردهی اولیه کرده و سپس سعی می‌نمائیم **خط‌مشی** بهینه را به صورت تکراری پیدا کنیم. یعنی سعی می‌کنیم **سیاست** بهینه‌ای پیدا کنیم که حداکثر **بازدهی** را به ما بدهد.

ابتدا بیایید ببینیم که چگونه از یادگیری تفاوت زمانمند (تفاضل زمانی) برای انجام کار **پیش‌بینی** استفاده کنیم و سپس نحوه استفاده از یادگیری تفاوت زمانمند را برای کار **کنترل**، یاد خواهیم گرفت.

پیش‌بینی تفاوت زمانمند

برای **پیش‌بینی** در روش تفاوت زمانمند، یک **خط‌مشی** به عنوان ورودی داده می‌شود و سعی می‌کنیم با استفاده از این **خط‌مشی**، **تابع ارزش** را تخمین بزنیم. یادگیری تفاوت زمانمند از طرفی همانند برنامه‌ریزی پویا، از تکنیک ابتناگری یا برپاگری استفاده می‌کند، بنابراین لازم نیست تا پایان پردینه (اپیزود) منتظر بمانید؛ و از طرف دیگر همانند روش مونت‌کارلو، برای محاسبه **تابع ارزش** یا **تابع Q** به پویایی مدل محیط نیازی ندارد. حال، بیایید ببینیم که قاعده بروزرسانی یادگیری تفاوت زمانمند چگونه طراحی شده است که بتواند مزایای فوق را در برگیرد.

بخاطر دارید که **در روش مونت کارلو (MC)، ارزش یک حالت را با محاسبه بازده آن تخمین می‌زدیم:**

$$V(s) \approx R(s)$$

با این حال، از آنجا یک **مقدار بازده تنها (منفرد)**[1] نمی‌تواند ارزش یک حالت را به **طور درستی** تقریب بزند، بنابراین در روش **MC**، به تعداد N پردینه تولید می‌شد و ما **ارزش** یک حالت را به عنوان میانگین **بازده** یک حالت در همه این N پردینه، محاسبه می‌کردیم:

[1] Single Return Value

$$V(s) \approx \frac{1}{N}\sum_{i=1}^{N} R_i\,(s)$$

اما **در روش MC باید تا پایان اپیزود صبر می‌کردیم تا ارزش یک حالت را محاسبه کنیم** و هنگامی که اپیزود طولانی بود، زمان زیادی برای محاسبه لازم بود. یکی دیگر از مشکلات روش مونت‌کارلو (**MC**) این بود که ما نمی‌توانستیم آن را برای کارها (یا وظایف) غیراپیزودیک (کارهای پیوسته) اعمال کنیم.

بنابراین در روش جدید، یعنی در یادگیری تفاوت زمانمند (TD)، از خودآوری یا خود-ابتنائی استفاده می‌کنیم و ارزش یک حالت را به صورت زیر تخمین می‌زنیم:

$$V(s) \approx r + \gamma V(s')$$

معادله بالا به ما می‌گوید که می‌توانیم ***ارزش*** حالت را فقط با محاسبه پاداش فوری r و ارزش تنزیلی حالت بعدی، یعنی $\gamma V(s')$ تخمین بزنیم. همانطور که در معادله بالا مشاهده می‌کنید، مشابه آنچه در روش‌های برنامه‌ریزی پویا **DP** (***تکرار ارزش*** و ***تکرار سیاست***) یاد گرفتیم، **ما در اینجا، خودبرپایی یا خود-ابتنائی را انجام می‌دهیم ولیکن مانند روش DP نیازی به دانستن پویایی‌های مدل نداریم.**

بنابراین، با استفاده از یادگیری تفاوت زمانمند (**TD**)، ارزش یک حالت به صورت تقریبی برابر است با:

$$V(s) \approx r + \gamma V(s')$$

با این حال، تنها مقدار $r + \gamma V(s')$، نمی‌تواند ارزش یک حالت را بدرستی تقریب بزند. بنابراین، می‌توانیم از یک مقدار میانگین استفاده کرده و به جای گرفتن میانگین حسابی، از **میانگین افزایش تدریجی** استفاده کنیم.

در روش مونت‌کارلو (**MC**)، نحوه استفاده از میانگین افزایش تدریجی برای تخمین ***ارزش*** حالت را یاد گرفتیم که به صورت زیر ارائه می‌شود:

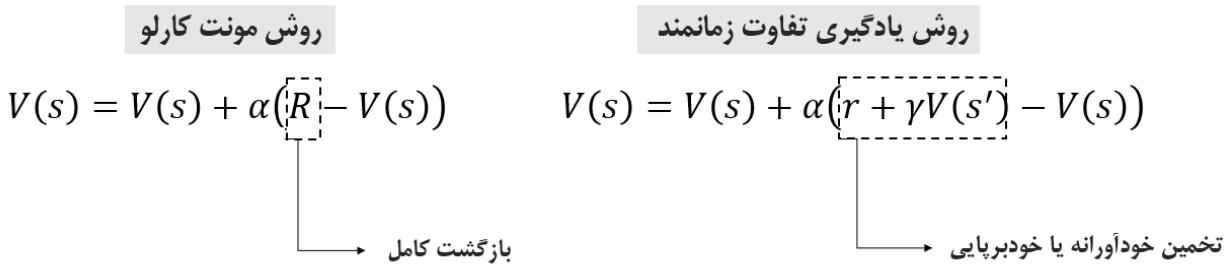
$$V(s) = V(s) + \alpha\big(R - V(s)\big)$$

به همین ترتیب، در یادگیری تفاوت زمانمند (**TD**)، می‌توان از میانگین افزایشی استفاده کرد و ***ارزش*** حالت را تخمین زد، همانطور که در اینجا نشان داده شده است:

$$V(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$$

این معادله را **قاعده به‌روزرسانی یادگیری TD (تفاوت زمانمند)** می‌نامند. همانطور که مشاهده می‌کنیم، تنها تفاوت بین یادگیری تفاوت زمانمند (**TD**) و روش مونت‌کارلو (**MC**) این است که برای محاسبه **ارزش** حالت، در روش مونت–کارلو از **بازگشت کامل** $\boldsymbol{R}$ استفاده می‌کنیم که با استفاده از اپیزود کامل محاسبه می‌شود، در حالی که در روش یادگیری تفاوت زمانمند، ما از **تخمین خودآورانه یا خودبرپایی** $\boldsymbol{r + \gamma V(s')}$ استفاده می‌کنیم، بنابراین، ما نیازی نداریم که تا پایان پردینه برای محاسبه **ارزش** حالت منتظر بمانیم. با این حساب، ما می‌توانیم یادگیری تفاوت زمانمند (**TD**) را برای کارهای غیراپیزودیک (ناپردینه‌ای) نیز اعمال کنیم.

موارد زیر تفاوت بین روش مونت‌کارلو (**MC**) و یادگیری تفاوت زمانمند (**TD**) را نشان می‌دهد:



شکل ۵.۱: مقایسه بین یادگیری تفاوت زمانمند و مونت‌کارلو

بنابراین، قاعده به‌روزرسانی یادگیری تفاوت زمانمند (**TD**) ما این است:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

ما آموختیم که $r + \gamma V(s')$ تخمینی از ارزش حالت $V(s)$ است. بنابراین، ما می‌توانیم، $r + \gamma V(s')$ را **هدف TD (تفاوت زمانمند)**، بنامیم. لذا، تفریق $V(s)$ از $r + \gamma V(s')$، به این معنی است که مقدار پیش‌بینی‌شده را از مقدار هدف، کم می‌کنیم، و این معمولاً **خطای TD (تفاوت زمانمند)** نامیده می‌شود. خب، در مورد α چه؟ این پارامتر اساساً نرخ یادگیری است که **اندازه گام** نیز نامیده می‌شود. یعنی:

$$V(s) = V(s) + \alpha(r + \gamma V(s') - V(s))$$

نرخ یادگیری (اندازه گام) — خطای TD (تفاوت زمانمند)

قاعده به‌روزرسانی یادگیری تفاوت زمانمند اساساً بیان می‌کند که:

[ارزش یک حالت – (ارزش حالت بعدی) *ضریب تخفیف + پاداش] *نرخ یادگیر + ارزش یک حالت = ارزش یک حالت

هم اینک، ما قاعده به‌روزرسانی یادگیری **TD** و نیز نحوه استفاده از یادگیری **TD** برای تخمین ارزش یک حالت را یادگرفتیم. در بخش بعدی، به الگوریتم **پیش‌بینی TD** نگاهی خواهیم انداخت و یک درک شفاف‌تری از روش یادگیری **TD** کسب خواهیم کرد.

الگوریتم پیش‌بینی تفاوت‌زمانی

ما یاد گرفتیم که در کار **پیش‌بینی**، بر اساس یک **خط‌مشی**، **تابع ارزش** را با استفاده از همان **خط‌مشی**، تخمین می‌زنیم. بنابراین، می‌توانیم بگوییم که یک عامل در هر حالتی که طبق **سیاست** داده شده عمل کند، چه **بازده** مورد انتظاری را می‌تواند به دست آورد.

ما آموختیم که قاعده به‌روزرسانی یادگیری تفاوت زمانمند به صورت زیر ارائه می‌شود:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

بنابراین، با استفاده از این معادله، می‌توان **تابع ارزش سیاست** داده شده را تخمین زد.

قبل از اینکه مستقیماً به الگوریتم نگاه کنیم، بهتر است برای درک بهتر، ابتدا محاسبات را به صورت دستی انجام دهیم تا می‌بینیم که چگونه با استفاده از قاعده به‌روزرسانی یادگیری تفاوت زمانمند دقیقاً ارزش یک حالت تخمین زده می‌شود.

	**بخشهای بعدی با محاسبات دستی توضیح داده میشوند، برای فهم بیشتر، با یک مداد و کاغذ ادامه دهید.**

بیایید ***پیش‌بینی*** تفاوت‌زمانمند (**TD**) را با محیط دریاچه یخزده بررسی کنیم. ما آموخته‌ایم که در محیط دریاچه یخزده، هدف عامل، رسیدن به حالت هدف **G** از حالت شروع **S** بدون بازدید از حالت‌های حفره **H** است. اگر عامل از حالت **G** بازدید کند، پاداش ۱ را تعیین می‌کنیم و اگر عامل، از هر حالت دیگری بازدید کند، ما پاداش برابر صفر را به آن تخصیص می‌دهیم. شکل ۵.۲ محیط دریاچه یخزده را نشان می‌دهد.

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

شکل ۵.۲: محیط دریاچه یخزده

ما چهار اقدام در فضای اقدام خود داریم که *بالا*، *پایین*، چپ و *راست* هستند و ۱۶ حالت از S تا G داریم. برای درک راحت‌تر، به جای اینکه حالات و اقدامات را با اعداد رمزگذاری کنیم، آن‌ها را همانطور که هستند نگه می‌داریم. یعنی بیایید هر اقدام را با واژه‌های *بالا*، *پایین*، چپ و *راست* نشان دهیم و هر حالت را با موقعیت آن‌ها در شبکه نشان دهیم. یعنی حالت اول **S** با (۱،۱) و حالت دوم **F** با (۱،۲) و به همین ترتیب تا آخرین حالت **G** که با (۴،۴) نشان داده می‌شود.

اکنون بیایید نحوه انجام ***پیش‌بینی*** تفاوت زمانمند (**TD**) را در محیط دریاچه یخ زده بیاموزیم. می‌دانیم که در روش ***پیش‌بینی*** تفاوت زمانمند، یک خط‌مشی به ما داده می‌شود و ***تابع ارزش (ارزش حالت)*** را با استفاده از همان ***خط‌مشی*** داده شده، پیش‌بینی می‌کنیم. فرض کنید سیاست زیر به ما داده شده است. این، اساساً به ما می‌گوید که در هر حالت چه ***اقدامی*** را انجام دهیم:

اقدام	حالت
راست	(۱،۱)
راست	(۱،۲)
چپ	(۱،۳)
⋮	⋮
پایین	(۴،۴)

جدول ۵.۱: سیاست داده شده

اکنون، نحوه تخمین تابع ارزش خط‌مشی قبلی را با استفاده از روش یادگیری تفاوت زمانمند خواهیم دید. قبل از ادامه، ابتدا ارزش تمام حالت‌ها را با ارزش‌های تصادفی (بختکی)، مقداردهی اولیه می‌کنیم، همانطور که در اینجا نشان داده شده است:

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

حالت	ارزش
(۱،۱)	۰.۹
(۱،۲)	۰.۶
(۱،۳)	۰.۸
⋮	⋮
(۴،۴)	۰.۷

شکل ۵.۳: حالت‌ها را با ارزش‌های تصادفی، مقداردهی اولیه کنید.

فرض کنید ما در حالت (۱،۱) هستیم و طبق خط‌مشی داده شده، اقدام حرکت به راست را انجام می‌دهیم و به حالت بعدی (۱،۲) می‌رویم و پاداش r برابر صفر را دریافت می‌کنیم. بیایید در این بخش، نرخ یادگیری α را برابر ۰.۱ و ضریب تخفیف γ را برابر ۱ قرار دهیم. حالا چطور می‌توانیم ارزش حالت را به‌روزرسانی کنیم؟

معادله به‌روزرسانی تفاوت زمانمند را به یاد بیاورید:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

در معادله قبل، ارزش حالت $V(s)$ را با $V(۱,۱)$ و حالت بعدی $V(s')$ را با $V(۱,۲)$ جایگزین کنید، می‌توانیم بنویسیم:

$$V(۱,۱) = V(۱,۱) + \alpha\big(r + \gamma V(۱,۲) - V(۱,۱)\big)$$

با جایگزینی پاداش $r = ۰$ ، نرخ یادگیری $\alpha = ۰.۱$ و نرخ تخفیف $\gamma = ۱$، می‌توانیم بنویسیم:

$$V(۱,۱) = V(۱,۱) + ۰.۱\big(۰ + ۱ * V(۱,۲) - V(۱,۱)\big)$$

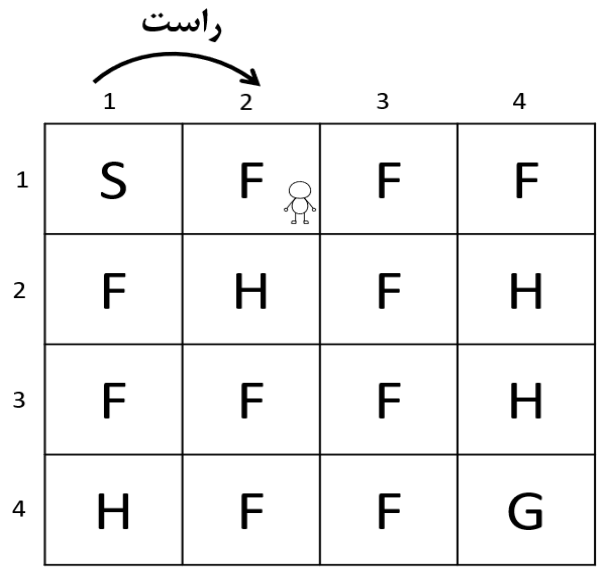
می‌توانیم ارزش‌های حالت را از جدول مقادیر نشان داده شده در قبل بدست آوریم. یعنی از جدول ارزش قبل مشاهده می‌کنیم که ارزش حالت (۱،۱) برابر ۰.۹ و ارزش حالت بعدی (۱،۲) برابر ۰.۶ است. با جایگزینی این ارزش‌ها در معادله قبل، می‌توانیم بنویسیم:

$$V(۱,۱) = ۰.۹ + ۰.۱(۰ + ۱ * ۰.۶ - ۰.۹)$$

بنابراین، ارزش حالت (۱،۱) برابر می‌شود با:

$$V(۱,۱) = ۰.۸۷$$

بنابراین، همانطور که شکل ۵.۴ نشان می‌دهد، ارزش حالت (۱،۱) را به صورت ۰.۸۷ در جدول ارزش به‌روز می‌کنیم:



ارزش	حالت
۰.۸۷	**(۱،۱)**
۰.۶	(۱،۲)
۰.۸	(۱،۳)
⋮	⋮
۰.۷	(۴،۴)

شکل ۵.۴: ارزش حالت (۱،۱) به‌روز می‌شود.

اکنون در حالت (۱،۲) هستیم. با توجه به خط‌مشی داده شده در حالت (۱،۲)، اقدام مناسب را انتخاب می‌کنیم و به حالت بعدی (۱،۳)، می‌رویم و یک پاداش r برابر ۰ دریافت می‌کنیم. می‌توانیم ارزش حالت را به صورت زیر محاسبه کنیم:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

با جایگزین کردن ارزش حالت $V(s)$ با $V(۱,۲)$ و حالت بعدی $V(s')$ با $V(۱,۳)$ می‌توانیم بنویسیم:

$$V(۱,۲) = V(۱,۲) + \alpha\big(r + \gamma V(۱,۳) - V(۱,۲)\big)$$

با جایگزینی پاداش $r = ۰$ ، نرخ یادگیری $\alpha = ۰.۱$ و نرخ تخفیف $\gamma = ۱$، می‌توانیم بنویسیم:

$$V(۱,۲) = V(۱,۲) + ۰.۱\big(۰ + ۱ \times V(۱,۳) - V(۱,۲)\big)$$

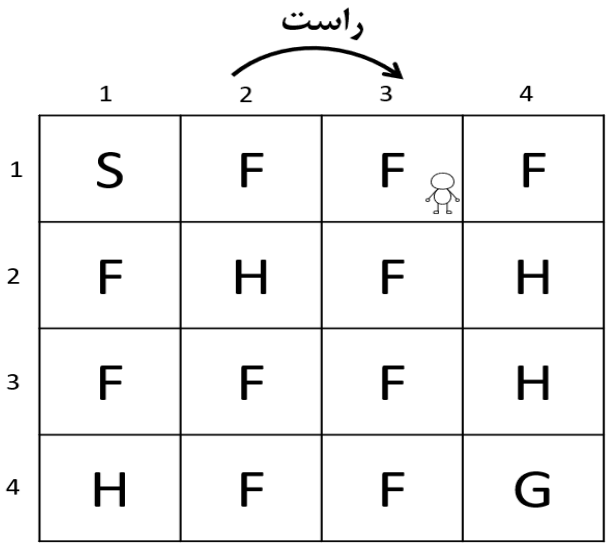
از جدول مقادیر قبل، مشاهده می‌کنیم که ارزش حالت (۱،۲)، برابر ۰.۶ و ارزش حالت بعدی (۱،۳)، برابر ۰.۸ است، بنابراین می‌توانیم بنویسیم:

$$V(۱,۲) = ۰.۶ + ۰.۱(۰ + ۱ \times ۰.۸ - ۰.۶)$$

بنابراین، ارزش حالت (۱،۲) برابر می‌شود با:

$$V(۱,۲) = ۰.۶۲$$

بنابراین، همانطور که شکل ۵.۵ نشان می‌دهد، در جدول مقدار ارزش حالت (۱،۲) را به ۰.۶۲ به‌روز می‌کنیم:



ارزش	حالت
۰.۸۷	(۱،۱)
۰.۶۲	**(۱،۲)**
۰.۸	(۱،۳)
⋮	⋮
۰.۷	(۴،۴)

شکل ۵.۵: ارزش حالت (۱،۲) به روز می‌شود.

اکنون در حالت (۱،۳) هستیم. ما اقدام چپ را مطابق خط‌مشی خود انتخاب می‌کنیم و به حالت بعدی (۱،۲) می‌رویم و یک پاداش r برابر صفر دریافت می‌کنیم. می‌توانیم ارزش حالت را به صورت زیر محاسبه کنیم:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

با جایگزین کردن ارزش حالت $V(s)$، با $V(۱،۳)$ و حالت بعدی $V(s')$ با $V(۱،۲)$ ، خواهیم داشت:

$$V(۱،۳) = V(۱،۳) + \alpha\big(r + \gamma V(۱،۲) - V(۱،۳)\big)$$

با جایگزینی پاداش $r = ۰$ ، نرخ یادگیری $\alpha = ۰.۱$ و نرخ تخفیف $\gamma = ۱$، می‌توانیم بنویسیم:

$$V(۱،۳) = V(۱،۳) + ۰.۱\big(۰ + ۱ * V(۱،۲) - V(۱،۳)\big)$$

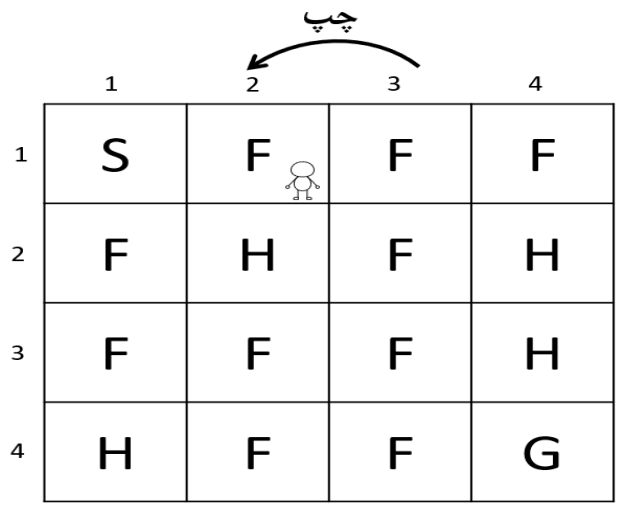
توجه داشته باشید که در هر مرحله از مقادیر به‌روز شده استفاده می‌کنیم، یعنی ارزش حالت (۱،۲) در مرحله قبل با ۰.۶۲ به‌روز می‌شود، همانطور که در جدول مقادیر قبل نشان داده شده است. بنابراین، $V(۱،۲)$ را با ۰.۶۲ و $V(۱،۳)$ را با ۰.۸ جایگزین می‌کنیم:

$$V(۱،۳) = ۰.۸ + ۰.۱(۰ + ۱ * ۰.۶۲ - ۰.۸)$$

بنابراین، ارزش حالت (۱،۳) برابر می‌شود با:

$$V(۱،۳) = ۰.۷۸۲$$

بنابراین، همانطور که شکل ۵.۶ نشان می‌دهد، در جدول مقدار ارزش حالت (۱،۳) را به ۰.۷۸۲ به‌روز می‌کنیم:



ارزش	حالت
۰.۸۷	(۱،۱)
۰.۶۲	(۱،۲)
۰.۷۸۲	**(۱،۳)**
⋮	⋮
۰.۷	(۴،۴)

شکل ۵.۶: ارزش حالت (۱،۳) به‌روز می‌شود.

بنابراین، به این ترتیب، ارزش هر حالت را با استفاده از خط‌مشی داده شده محاسبه می‌کنیم. با این حال، محاسبه ارزش حالت فقط برای یک اپیزود، دقیق نخواهد بود. بنابراین، ما این مراحل را برای چندین اپیزود تکرار می‌کنیم و برآوردهای دقیق ارزش حالت (تابع ارزش) را محاسبه می‌کنیم.

الگوریتم **پیش‌بینی** تفاوت زمانمند به صورت زیر ارائه می‌شود:

۱. یک **تابع ارزش** $V(s)$ را با مقادیر تصادفی (بختکی) مقداردهی اولیه کنید. ضمنا یک **سیاست** π داده شده است.
۲. برای هر اپیزود:
    ۱. پارامتر S را مقداردهی اولیه کنید.
    ۲. برای هر مرحله از پردینه (اپیزود):
        ۱. طبق **خط‌مشی** داده شده π یک **اقدام** a را در حالت S انجام دهید، **پاداش** r را دریافت کنید و به حالت بعدی S' بروید.
        ۲. **ارزش** حالت را به روز کنید: $$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$
        ۳. **مقدار** S را به‌روزرسانی کنید، یعنی $S = S'$ (این مرحله به این معنی است که حالت بعدی S' را به وضعیت فعلی S تغییر می‌دهیم.
        ۴. اگر S حالت پایانی نیست، مراحل ۱ تا ۴ را تکرار کنید

اکنون که یاد گرفتیم روش **پیش‌بینی** تفاوت زمانمند (**TD**)، چگونه **تابع ارزش** سیاست داده شده را پیش‌بینی می‌کند، در بخش بعدی، بیایید نحوه اجرای روش **پیش‌بینی** تفاوت زمانمند را برای پیش‌بینی **ارزش** حالت‌ها در محیط دریاچه یخزده، بیاموزیم.

پیش‌بینی ارزش حالت‌ها در محیط دریاچه یخزده

آموختیم که در روش **پیش‌بینی**، خط‌مشی به عنوان یک ورودی داده می‌شود و **تابع ارزش** را با استفاده از **خط‌مشی** داده شده، پیش‌بینی می‌کنیم. بنابراین، بیایید یک **سیاست** تصادفی (بختکی) را مقداردهی اولیه کنیم و **تابع ارزش** (ارزش حالت‌ها) محیط دریاچه منجمد را با استفاده از همین **خط‌مشی** تصادفی پیش‌بینی کنیم.

ابتدا بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import pandas as pd
```

اکنون محیط دریاچه یخ زده را با استفاده از جیم ایجاد می‌کنیم:

```
env = gym.make('FrozenLake-v0')
```

خط‌مشی تصادفی را تعریف کنید، که **اقدام** تصادفی را با نمونه‌برداری از **فضای اقدام** برمی‌گرداند:

```
def random_policy():
    return env.action_space.sample()
```

بیایید دیکشنری را برای ذخیره **ارزش** حالت‌ها تعریف کنیم و **ارزش** همه حالت‌ها را مقداردهی اولیه برابر صفر کنیم:

```
V = {}
for s in range(env.observation_space.n):
    V[s]=0.0
```

فاکتور تخفیف γ و نرخ یادگیری α را تعریف کنید:

```
alpha = 0.85
gamma = 0.90
```

تعداد اپیزودها و تعداد گامهای زمانی[۱] هر اپیزود را تنظیم کنید:

```
num_episodes = 50000
num_timesteps = 1000
```

محاسبه ارزش حالت‌ها

حالا بیایید **تابع ارزش** (ارزش حالت‌ها) را با استفاده از **خط‌مشی** تصادفی داده شده، محاسبه کنیم.

برای هر اپیزود:

---

[۱] time steps

```
for i in range(num_episodes):
```

با تنظیم مجدد محیط، حالت را راه‌اندازی کنید:

```
    s = env.reset()
```

برای هر مرحله از اپیزود:

```
    for t in range(num_timesteps):
```

براساس **خط‌مشی** تصادفی، اقدامی را انتخاب کنید:

```
        a = random_policy()
```

اقدام انتخاب شده را انجام دهید و اطلاعات حالت بعدی را ذخیره کنید:

```
        s_, r, done, _ = env.step(a)
```

ارزش حالت را به صورت $V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$ محاسبه کنید:

```
        V[s] += alpha * (r + gamma * V[s_]-V[s])
```

حالت بعدی را به حالت فعلی، به‌روز کنید $s = s'$:

```
        s = s_
```

اگر حالت فعلی، حالت پایانی است، آنگاه متوقف شوید:

```
        if done:
            break
```

پس از تمام تکرارها، ما **ارزش‌های** همه حالت‌ها را با توجه به **خط‌مشی** تصادفی داده شده، خواهیم داشت.

ارزیابی ارزش‌های حالت‌ها

حالا بیایید **تابع ارزش** خود (ارزش حالت‌ها) را ارزیابی کنیم. برای وضوح بیشتر، ابتدا دیکشنری **ارزش** خود را به یک دیتافریم پانداس تبدیل می‌کنیم:

```
df = pd.DataFrame(list(V.items()), columns=['state', 'value'])
```

قبل از بررسی ارزش‌های حالت‌ها، بیایید به یاد بیاوریم که در جیم، تمام حالت‌های موجود در محیط دریاچه یخزده به صورت اعداد رمزگذاری می‌شوند. از آنجایی که ما شانزده حالت داریم، تمام حالت‌ها به اعداد ۰ تا ۱۵ (همانند شکل ۵.۷) کدگذاری می‌شوند.

0 S	1 F	2 F	3 F
4 F	5 H	6 F	7 H
8 F	9 F	10 F	11 H
12 H	13 F	14 F	15 G

شکل ۵.۷: حالت‌ها، به صورت اعداد رمزگذاری شده‌اند.

حالا بیایید **ارزش** حالت‌ها را بررسی کنیم:

```
df
```

کد بالا، مقادیر زیر را چاپ خواهد کرد.

همانطور که مشاهده می‌کنیم، اکنون **ارزش‌های** همه حالت‌ها را داریم. ارزش حالت ۱۴ زیاد است زیرا می‌توانیم از حالت ۱۴ به راحتی به حالت هدف ۱۵ برسیم و همچنین همانگونه که می‌بینیم ارزش‌های تمام حالت‌های پایانی (حالت‌های حفره و حالت هدف) صفر است.

توجه داشته باشید که از آنجایی که ما با یک **خط‌مشی** تصادفی راه‌اندازی یا شروع کرده‌ایم، ممکن است هر بار که کد قبلی را اجرا می‌کنید، نتایج متفاوتی دریافت کنید.

حالت	ارزش
۰	۰.۱۲۴۱۸۰۷
۱	۰.۰۰۲۴۹۱۱
۲	۰.۰۰۰۱۸۹۷
۳	۰.۰۰۰۰۰۰۰
۴	۰.۰۲۴۲۷۰۸
۵	۰.۰۰۰۰۰۰۰
۶	۰.۰۰۰۸۲۰۸
۷	۰.۰۰۰۰۰۰۰
۸	۰.۱۶۰۵۳۷۹
۹	۰.۰۲۳۰۶۷۷
۱۰	۰.۰۰۳۵۵۸۱
۱۱	۰.۰۰۰۰۰۰۰
۱۲	۰.۰۰۰۰۰۰۰
۱۳	۰.۴۰۶۳۴۳۶
۱۴	۰.۸۷۷۰۳۰۲
۱۵	۰.۰۰۰۰۰۰۰

شکل ۵.۸: جدول ارزش

اکنون فهمیدیم چگونه می‌توان از یادگیری تفاوت زمانمند (تفاضل زمانی یا موقتی) برای وظایف **پیش‌بینی** استفاده کرد. در بخش بعدی نحوه استفاده از یادگیری تفاوت زمانمند را برای کارهای **کنترلی** خواهیم آموخت.

کنترل تفاوت زمانمند

در روش **کنترل**، هدف ما یافتن **خط‌مشی** بهینه است، بنابراین با یک خط‌مشی تصادفی (بختکی) اولیه شروع می‌کنیم و سپس سعی می‌کنیم **خط‌مشی** بهینه را به صورت تکراری پیدا کنیم. در فصل قبل آموختیم که روش **کنترل** را می‌توان به دو دسته طبقه‌بندی کرد:

- **کنترل با خط‌مشی همگاه (عیان-سیاست)**
- **کنترل با خط‌مشی ناهمگاه (نهان-سیاست)**

ما در فصل قبل، یاد گرفتیم که **کنترل با خطمشی همگاه** و **کنترل با خطمشی ناهمگاه** چیست. بیایید قبل از ادامه، آنها را کمی مرور کنیم. در **کنترل با خطمشی همگاه**، عامل با استفاده از یک **خطمشی**، رفتار کرده و سعی می‌کند همان **سیاست** را بهبود بخشد. یعنی در روش **سیاست همگاه**، با استفاده از یک **سیاست**، اپیزودهایی تولید می‌کنیم و دقیقا همان **خطمشی** را به طور مکرر بهبود می‌دهیم تا **سیاست** بهینه را پیدا کنیم.

در روش **کنترل با سیاست-ناهمگاه**، عامل با استفاده از یک **سیاست واسط** رفتار می‌کند و سعی می‌کند **خطمشی** دیگری (و نه همان **خطمشی یا سیاست**) را بهبود بخشد. یعنی در روش سیاست ناهمگاه، ما اپیزودها را با استفاده از یک خطمشی تولید کرده و سعی می‌کنیم یک خطمشی دیگری (**سیاست هدف**) را به طور مکرر برای یافتن خطمشی بهینه، بهبود دهیم.

اکنون، نحوه انجام وظایف کنترلی با استفاده از یادگیری تفاوت زمانمند را یاد خواهیم گرفت. ابتدا نحوه اجرای **کنترل** تفاوت زمانمند با سیاست **همگاه** را یاد گرفته و سپس با **کنترل** تفاوت زمانمند سیاست **ناهمگاه** آشنا خواهیم شد.

کنترل تفاوت زمانمند با سیاست_همگاه: سارسا

در این بخش، ما نگاهی به الگوریتم محبوب **کنترل** تفاوت زمانمند (**TD**) با خطمشی همگاه به نام **سارسا** (**SARSA**) که ترکیب سرواژگان **حالت**- **اقدام**- **پاداش**- **حالت**-**اقدام** است، خواهیم داشت. ما می‌دانیم که در **کنترل** تفاوت زمانمند، هدف ما یافتن **خطمشی** یا **سیاست** بهینه است. خب، چگونه می‌توانیم یک سیاست استخراج کنیم؟ می‌توانیم **خطمشی** را از **تابع** Q استخراج کنیم. یعنی زمانی که **تابع** Q را داریم، می‌توانیم با انتخاب اقدامی که در هر حالتی حداکثر **ارزش** Q را دارد، سیاست را استخراج کنیم.

خوب، چگونه می‌توانیم **تابع** Q را در یادگیری تفاوت زمانمند محاسبه کنیم؟ ابتدا، بیایید به یاد بیاوریم که چگونه **تابع ارزش** را محاسبه می‌کنیم. در یادگیری تفاوت زمانمند، **تابع ارزش** به صورت زیر محاسبه می‌شود:

$$V(s) = V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$$

ما می‌توانیم این قاعده به‌روزرسانی را از نظر **تابع** Q، به صورت زیر بازنویسی کنیم:

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

اکنون، **تابع** Q را با استفاده از قاعده به‌روزرسانی یادگیری تفاوت زمانمند قبلی محاسبه کرده و سپس یک **خط‌مشی** از آنها استخراج می‌کنیم. همچنین می‌توانیم قاعده به‌روزرسانی قبلی را **قاعده به‌روزرسانی سارسا** بنامیم.

اما صبر کنید! در روش **پیش‌بینی**، یک خط‌مشی به عنوان ورودی به ما داده شد، بنابراین ما با استفاده از آن خط‌مشی در محیط اقدام کرده و **تابع ارزش** را محاسبه کردیم. اما در اینجا، ما هیچ **خط‌مشی** یا **سیاستی** به عنوان ورودی نداریم. پس چگونه می‌توانیم در محیط عمل کنیم؟

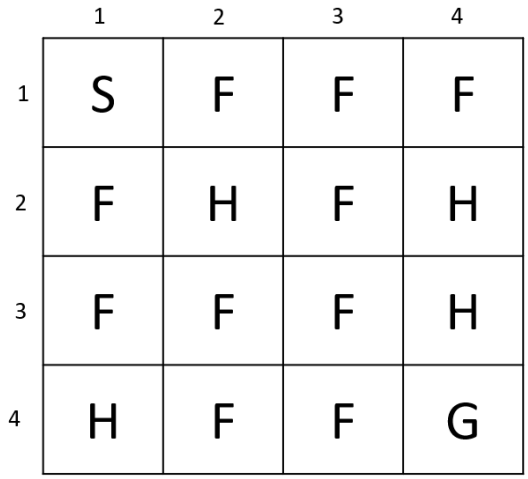
بنابراین، ابتدا **تابع** Q را با مقادیر تصادفی یا با صفر، مقداردهی اولیه می‌کنیم. سپس از این تابع Q که به طور تصادفی مقداردهی اولیه شده است، یک **خط‌مشی** استخراج می‌کنیم و در محیط اقدام می‌کنیم. **خط‌مشی** اولیه ما قطعاً بهینه نخواهد بود زیرا از **تابع** Q و با مقداردهی اولیه تصادفی، استخراج می‌شود، اما در هر اپیزود، **تابع** Q (**ارزش‌های** Q) را به روز می‌کنیم. بنابراین، در هر اپیزود، می‌توانیم از **تابع** Q به‌روز شده برای استخراج یک **خط‌مشی** جدید استفاده کنیم. بنابراین، ما **سیاست** بهینه را پس از یک سری پردینه (اپیزود)، به دست خواهیم آورد.

نکته مهمی که باید به آن توجه کنیم این است که در روش **سارسا** به جای اینکه کاری کنیم **خط‌مشی** ما حریصانه عمل کند، از **خط‌مشی اپسیلون حریصانه** استفاده می‌کنیم. یعنی در یک سیاست حریصانه، ما همیشه اقدامی را انتخاب می‌کنیم که حداکثر **ارزش** Q را داشته باشد. اما، با سیاست اپسیلون حریصانه، یک اقدام تصادفی با احتمال اپسیلون را انتخاب می‌کنیم و بهترین اقدام (اقدامی با حداکثر **ارزش** Q) را با احتمال $1 - \epsilon$ انتخاب می‌کنیم.

برای درک بهتر الگوریتم و قبل از اینکه مستقیماً به آن نگاه کنیم، بیایید ابتدا به صورت دستی محاسبات را انجام داده تا ببینیم که دقیقاً چگونه **تابع** Q (**ارزش** Q) با استفاده از قاعده به‌روزرسانی **سارسا** تخمین زده می‌شود و چگونه می‌توانیم **خط‌مشی** بهینه را پیدا کنیم.

بگذارید همین محیط دریاچه یخ زده را در نظر بگیریم. قبل از ادامه، جدول Q (یا **تابع** Q) را با مقادیر تصادفی (بختکی)، مقداردهی اولیه می‌کنیم. شکل ۵.۹، محیط دریاچه یخ زده را به همراه جدول Q حاوی مقادیر تصادفی نشان می‌دهد:

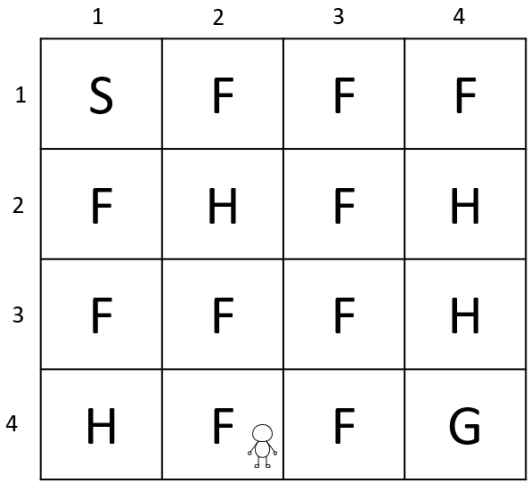
ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۳	بالا	(۴،۲)
۰.۵	پایین	(۴،۲)
۰.۱	چپ	(۴،۲)
۰.۸	راست	(۴،۲)
⋮	⋮	⋮
۰.۵	راست	(۴،۴)



شکل ۵.۹: محیط دریاچه منجمد و جدول Q با مقادیر تصادفی

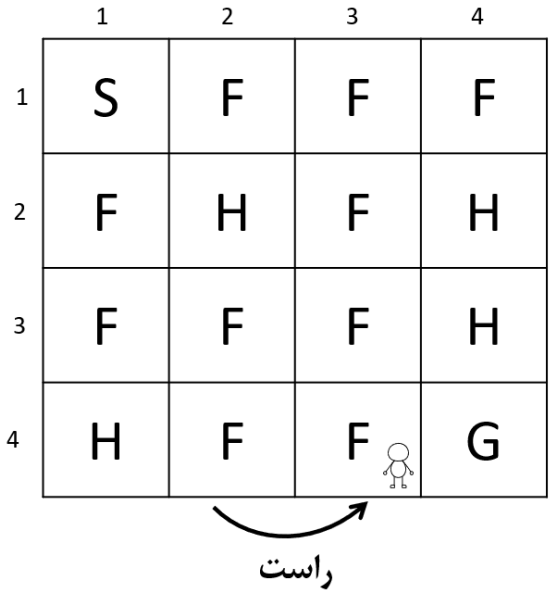
فرض کنید در حالت (۴،۲) هستیم. اکنون باید یک اقدام را در این حالت انتخاب کنیم. چگونه می‌توانیم یک اقدام را انتخاب کنیم؟ ما یاد گرفتیم که در روش **سارسا**، یک اقدام را بر اساس **سیاست اپسیلون حریصانه**، انتخاب می‌کنیم. با احتمال اپسیلون، یک اقدام تصادفی و با احتمال $1 - \epsilon$ بهترین اقدام (اقدامی که حداکثر **ارزش** Q را دارد) را انتخاب می‌کنیم. فرض کنید ما از یک احتمال $1 - \epsilon$ استفاده کرده و بهترین اقدام را انتخاب می‌کنیم. بنابراین، در حالت (۴،۲)، به سمت *راست* حرکت می‌کنیم زیرا بالاترین **ارزش** Q را در مقایسه با سایر اقدامات دارد، همانطور که در اینجا نشان داده شده است:

ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۳	بالا	(۴،۲)
۰.۵	پایین	(۴،۲)
۰.۱	چپ	(۴،۲)
۰.۸	**راست**	**(٤،۲)**
⋮	⋮	⋮
۰.۵	راست	(۴،۴)



شکل ۵.۱۰: عامل ما در حالت (۴،۲) است.

بسیار خوب، بنابراین، اقدام حرکت به راست را در حالت (۴،۲) انجام می‌دهیم و همانطور که شکل ۵.۱۱ نشان می‌دهد، به حالت بعدی (۴،۳) می‌رویم.



حالت	اقدام	ارزش
(۱،۱)	بالا	۰.۵
⋮	⋮	⋮
(۴،۲)	بالا	۰.۳
(۴،۲)	پایین	۰.۵
(۴،۲)	چپ	۰.۱
(٤،۲)	**راست**	**۰.۸**
⋮	⋮	⋮
(۴،۴)	راست	۰.۵

شکل ۵.۱۱: اقدام با حداکثر ارزش Q در حالت (۴،۲) را انجام می‌دهیم.

بنابراین، ما در حالت (۴،۲) به حالت بعدی (۴،۳) حرکت کردیم و یک پاداش r برابر صفر دریافت کردیم. بیایید نرخ یادگیری α را ۰.۱ نگه داریم، و ضریب تخفیف γ را برابر ۱ قرار دهیم. حال، چگونه می‌توانیم ارزش Q را به‌روزرسانی کنیم؟

اجازه دهید قاعده به‌روزرسانی سارسا را به یاد بیاوریم:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a))$$

با جایگزین کردن زوج حالت–اقدام $Q(s,a)$ با $Q((۴،۲), \text{راست})$ و حالت بعدی s' با (۴،۳) در معادله قبل، می‌توانیم بنویسیم:

$$Q((۴،۲), \text{راست}) = Q((۴،۲), \text{راست}) + \alpha(r + \gamma Q((۴،۳), a') - Q((۴،۲), \text{راست}))$$

با جایگزینی پاداش $r = ۰$، نرخ یادگیری $\alpha = ۰.۱$ ، و ضریب تخفیف $\gamma = ۱$، می‌توانیم بنویسیم:

$$Q\left((۴،۲), \text{راست}\right) = Q\left((۴،۲), \text{راست}\right) + ۰.۱\left(۰ + ۱ \times Q((۴،۲), a') - Q\left((۴،۲), \text{راست}\right)\right)$$

از جدول Q قبلی، می‌توان مشاهده کرد که ارزش (راست, $Q((۴,۲)$ برابر **۰.۸** است. بنابراین، با جایگزینی (راست, $Q((۴,۲)$ با **۰.۸**، می‌توانیم معادله قبلی را به صورت زیر بازنویسی کنیم:

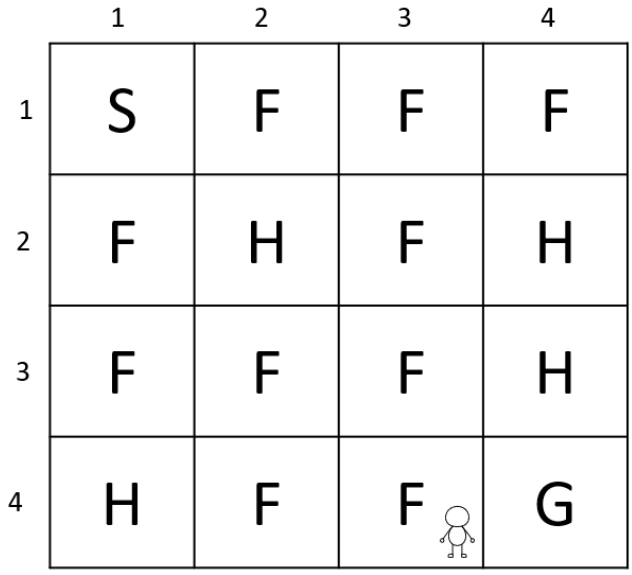
$$Q((۴,۲), \text{راست}) = ۰.۸ + ۰.۱(۰ + ۱ \times Q((۴,۳), a') - ۰.۸)$$

خوب، در مورد عبارت $Q((۴,۳), a')$ چطور؟ همانطور که در معادله قبلی می‌بینید، ما عبارت $Q((۴,۳), a')$ را داریم که نشان‌دهنده ارزش Q زوج حالت–اقدام بعدی است.

از آنجایی که به حالت بعدی (۴،۳) رفته‌ایم، باید یک اقدام را در این حالت برگزینیم تا ارزش Q مربوط به زوج حالت–اقدام بعدی را بدست آوریم. بنابراین، ما از همان سیاست اپسیلون حریصانه برای انتخاب اقدام استفاده می‌کنیم. یعنی یا یک اقدام تصادفی با احتمال ϵ و یا بهترین اقدام با حداکثر ارزش Q را با احتمال $۱ - \epsilon$، انتخاب می‌کنیم.

فرض کنید از احتمال اپسیلون استفاده کرده و اقدام تصادفی را برگزینیم. همانطور که شکل ۵.۱۲ نشان می‌دهد، در حالت (۴،۳)، اقدام حرکت به راست را به طور تصادفی انتخاب می‌کنیم. همانطور که می‌بینید، اگرچه اقدام حرکت به راست، حداکثر ارزش Q را ندارد، ما آن را به طور تصادفی با احتمال ϵ انتخاب کردیم:

ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۱	چپ	(۴،۲)
۰.۸	راست	(۴،۲)
۰.۱	بالا	(۴،۳)
۰.۳	پایین	(۴،۳)
۱.۰	چپ	(۴،۳)
۰.۹	**راست**	**(٤،۳)**
⋮	⋮	⋮
۰.۵	راست	(۴،۴)



شکل ۵.۱۲: یک اقدام تصادفی (بختکی) را در حالت (۴،۳) انجام می‌دهیم.

بنابراین، اکنون قاعده به‌روزرسانی ما به این صورت می‌شود:

$$Q((۴,۲), \text{راست}) = ۰.۸ + ۰.۱\left(۰ + ۱ \times Q\left((۴,۳), \text{راست}\right) - ۰.۸\right)$$

از جدول Q قبلی، می‌بینیم که ارزش $\boldsymbol{Q((4,3), \text{راست})}$، برابر **۰.۹** است. بنابراین، با جایگزینی ارزش $\boldsymbol{Q((4,3), \text{راست})}$ با **۰.۹**، می‌توانیم معادله فوق را به صورت زیر بازنویسی کنیم:

$$Q((۴,۲), \text{راست}) = ۰.۸ + ۰.۱(۰ + ۱ \times ۰.۹ - ۰.۸)$$

بنابراین، **ارزش** Q برابر می‌شود با:

$$Q((۴,۲), \text{راست}) = ۰.۸۱$$

بنابراین، به این ترتیب، **تابع** Q را با به‌روزرسانی ارزش Q زوج حالت–اقدام در هر مرحله از اپیزود، به‌روز می‌کنیم. پس از تکمیل یک اپیزود، یک سیاست جدید از **تابع** Q به روز شده استخراج کرده و از این **خط‌مشی** جدید برای اقدام در محیط بهره می‌بریم. (به یاد داشته باشید که **در سارسا، سیاست ما همیشه یک سیاست ϵ حریصانه است**). این مراحل را برای چندین اپیزود تکرار می‌کنیم تا خط‌مشی بهینه را پیدا کنیم. الگوریتم **سارسا** ارائه شده در ادامه، به درک بهتر این موضوع کمک می‌کند.

الگوریتم سارسا به صورت زیر ارائه شده است:

۱. یک **تابع** Q مربوط به $Q(s,a)$ را با مقادیر تصادفی (دلبخواه) مقداردهی اولیه کنید.
۲. برای هر اپیزود:
  ۱. s را مقداردهی اولیه کنید.
  ۲. یک **سیاست** را از $Q(s,a)$ استخراج کنید و یک **اقدام** a را برای انجام در حالت s انتخاب کنید.
  ۳. برای هر مرحله از پردینه:
    ۱. اقدام a را انجام دهید و به حالت بعدی s' بروید و **پاداش** r را مشاهده کنید.
    ۲. در حالت s'، اقدام a' را با استفاده از **خط‌مشی اپسیلون حریصانه** انتخاب کنید.
    ۳. ارزش Q را به‌روز کنید: $Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$

۴. $s = s'$ و $a = a'$ را به‌روزرسانی کنید

یعنی زوج (حالت s' – اقدام a') بعدی را به زوج (حالت s – اقدام a) فعلی، به‌روز کنید.

۵. اگر s حالت پایانی نیست، مراحل ۱ تا ۵ را تکرار کنید

اکنون، نحوه عملکرد الگوریتم **سارسا** را یاد گرفتیم، در بخش بعدی، الگوریتم **سارسا** را برای یافتن خط‌مشی بهینه، پیاده‌سازی می‌کنیم.

محاسبه خط‌مشی بهینه با استفاده از سارسا

حال بیایید **سارسا** را برای یافتن **سیاست** بهینه در محیط دریاچه یخ‌زده، پیاده‌سازی کنیم.

ابتدا بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import random
```

اکنون با استفاده از جیم، محیط دریاچه یخ زده را ایجاد می‌کنیم:

```
env = gym.make('FrozenLake-v0')
```

بیایید دیکشنری را برای ذخیره **ارزش** Q مربوط به زوج حالت–اقدام تعریف کنیم و **ارزش** اولیه Q را برای همه زوج‌های حالت–اقدام برابر صفر مقداردهی اولیه کنیم.

```
Q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s,a)] = 0.0
```

حالا بیایید سیاست **اپسیلون حریصانه** را تعریف کنیم. ما یک عدد تصادفی از توزیع یکنواخت داریم و اگر عدد تصادفی کوچکتر از اپسیلون باشد، **اقدام** تصادفی را انتخاب می‌کنیم، در غیر این صورت بهترین **اقدامی** را انتخاب می‌کنیم که حداکثر **ارزش** Q را داشته باشد:

```
def epsilon_greedy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return max(list(range(env.action_space.n)), key = lambda x:
Q[(state,x)])
```

ضریب تخفیف γ، نرخ یادگیری α و مقدار اپسیلون را مقداردهی اولیه کنید:

```
alpha = 0.85
gamma = 0.90
epsilon = 0.8
```

تعداد پردینه‌ها و تعداد مراحل زمانی در پردینه را تنظیم کنید:

```
num_episodes = 50000
num_timesteps = 1000
```

محاسبه خطمشی

برای هر پردینه (اپیزود):

```
for i in range(num_episodes):
```

با تنظیم مجدد محیط، حالت را راه‌اندازی کنید:

```
    s = env.reset()
```

اقدام را با استفاده از **خطمشی اپسیلون حریصانه**، انتخاب کنید:

```
    a = epsilon_greedy(s,epsilon)
```

برای هر مرحله در پردینه:

```
    for t in range(num_timesteps):
```

اقدام انتخاب شده را انجام دهید و اطلاعات حالت بعدی را ذخیره کنید:

```
        s_, r, done, _ = env.step(a)
```

اقدام بعدی a' در حالت بعدی s' را با استفاده از سیاست **اپسیلون حریصانه**، انتخاب کنید:

```
        a_ = epsilon_greedy(s_,epsilon)
```

ارزش Q زوج حالت–اقدام بعدی را محاسبه کنید:

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

```
        Q[(s,a)] += alpha * (r + gamma * Q[(s_,a_)]-Q[(s,a)])
```

$s = s'$ و $a = a'$ را به‌روزرسانی کنید (زوج حالت s' – اقدام a' بعدی را به حالت s – اقدام a فعلی به روز کنید)

```
        s = s_
        a = a_
```

اگر حالت فعلی، حالت پایانی است، آنگاه متوقف شوید:

```
        if done:
            break
```

توجه داشته باشید که در هر تکرار ما **تابع** Q را به‌روزرسانی می‌کنیم. بعد از همه تکرارها، **تابع** بهینه Q را خواهیم داشت. وقتی‌که ما **تابع** بهینه Q را داریم، آنگاه می‌توانیم **سیاست** بهینه را با انتخاب اقدامی که بیشترین **ارزش** Q را در هر حالت دارد، استخراج کنیم.

روش کنترل تفاوت زمانمند با سیاست ناهمگاه (اصطلاحا یادگیری Q)

در این قسمت با الگوریتم **کنترل** تفاوت زمانمند (**TD**) با خطمشی ناهمگاه (**زمان-سیاست**) بنام **یادگیری** Q آشنا می‌شویم. **یادگیری** Q یکی از الگوریتم‌های بسیار محبوب در یادگیری تقویتی است و خواهیم دید که این الگوریتم در فصل‌های دیگر نیز مطرح می‌شود. **یادگیری** Q یک الگوریتم با **خطمشی ناهمگاه** است، به این معنی که ما از **دو خطمشی** مختلف استفاده می‌کنیم، یکی برای **رفتار در محیط** (انتخاب یک اقدام در محیط) و دیگری برای یافتن **خطمشی بهینه**.

ما یاد گرفتیم که در روش **سارسا**، اقدام a را در حالت s، با استفاده از سیاست **اپسیلون حریصانه** انتخاب می‌کنیم، به حالت بعدی s' می‌رویم، و ارزش Q را با استفاده از قاعده به‌روزرسانی نشان داده شده در اینجا به روز می‌کنیم:

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

در معادله قبل، برای محاسبه ارزش Q زوج حالت–اقدام بعدی، $Q(s',a')$ ، نیاز داریم که یک اقدام را انتخاب کنیم. بنابراین، ما اقدام را با استفاده از همان خطمشی **اپسیلون حریصانه** انتخاب می‌کنیم و ارزش Q زوج حالت–اقدام بعدی را به‌روزرسانی می‌کنیم.

اما برخلاف روش **سارسا**، در روش **یادگیری** Q، از **دو خطمشی** متفاوت استفاده می‌کنیم (واژه ناهمگاه اشاره به همین دوگانگی دارد). یکی **سیاست اپسیلون حریصانه** و دیگری **سیاست مطلقا حریصانه** است. برای انتخاب یک «اقدام در محیط»، از یک **خطمشی اپسیلون حریصانه** استفاده می‌کنیم، اما در حین «به‌روزرسانی ارزش Q زوج حالت–اقدام بعدی» از یک **خطمشی کاملا حریصانه** استفاده می‌کنیم.

یعنی اقدام a را در حالت s را با استفاده از خطمشی اپسیلون حریصانه، انتخاب می‌کنیم و به حالت بعدی s' می‌رویم و ارزش Q را با استفاده از قاعده به‌روزرسانی نشان داده شده در ادامه، به‌روز می‌کنیم:

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

در معادله قبل، برای محاسبه ارزش Q زوج حالت–اقدام بعدی، $Q(s',a')$، باید یک اقدام را انتخاب کنیم. در اینجا، اقدام را با استفاده از سیاست کاملا حریصانه انتخاب می‌کنیم و ارزش Q زوج حالت–اقدام بعدی را به روز می‌کنیم. می‌دانیم که سیاست کاملا حریصانه یا مطلقا حریصانه همیشه اقدامی را انتخاب می‌کند که حداکثر ارزش را داشته باشد. بنابراین، می‌توانیم معادله را به صورت زیر تغییر دهیم:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

همانطور که از معادله قبل مشاهده می‌کنیم، **عملگر** max نشان می‌دهد که در حالت s'، اقدام a'ی را انتخاب می‌کنیم که حداکثر ارزش Q را دارد.

بنابراین، به طور خلاصه، در روش **یادگیری** Q، یک اقدام در «محیط» را با استفاده از **سیاست اپسیلون حریصانه**، انتخاب می‌کنیم، اما در حین «محاسبه» ارزش Q زوج حالت–اقدام بعدی، از **سیاست کاملا حریصانه** استفاده می‌کنیم. بنابراین، قاعده به‌روزرسانی **یادگیری** Q به صورت زیر ارائه می‌شود:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

بیایید با محاسبه دستی ارزش Q با استفاده از قاعده به‌روزرسانی **یادگیری** Q، این را بهتر درک کنیم. بیایید از همان مثال دریاچه یخ زده استفاده کنیم. جدول Q را با مقادیر تصادفی، مقداردهی اولیه می‌کنیم. شکل ۵.۱۳ محیط دریاچه یخ زده را به همراه جدول Q حاوی ارزش تصادفی نشان می‌دهد.

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

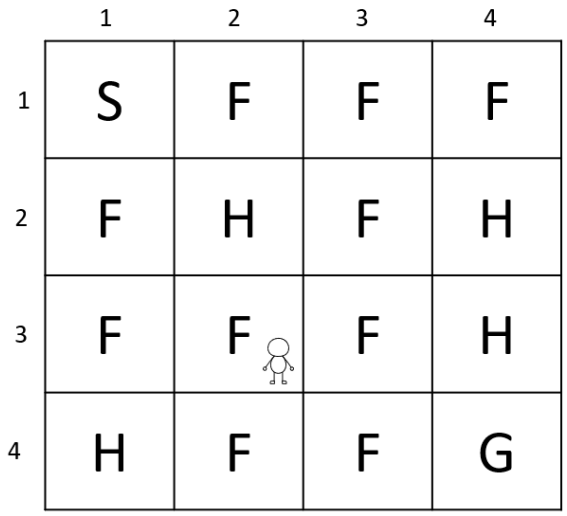
ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۱	بالا	(۳،۲)
۰.۸	پایین	(۳،۲)
۰.۵	چپ	(۳،۲)
۰.۶	راست	(۳،۲)
⋮	⋮	⋮
۰.۵	راست	(۴،۴)

شکل ۵.۱۳: محیط دریاچه یخ زده با جدول Q که به صورت تصادفی مقداردهی اولیه شده است.

فرض کنید در حالت (۳،۲) هستیم. حال باید در این حالت، اقدامی را انتخاب کنیم. چگونه می‌توانیم یک اقدام را برگزینیم؟ ما یک اقدام را با استفاده از **خط‌مشی اپسیلون حریصانه** انتخاب می‌کنیم. بنابراین، با احتمال اپسیلون، یک اقدام تصادفی و با احتمال $\epsilon - ۱$ بهترین اقدامی را برگزینیم که حداکثر **ارزش** Q را داشته باشد.

فرض کنید از احتمال $\epsilon - ۱$ استفاده کرده و بهترین اقدام را انتخاب می‌کنیم. بنابراین، در حالت (۳،۲)، اقدام حرکت به پایین را برمی‌گزینیم زیرا همانطور که شکل ۵.۱۴ نشان می‌دهد، بالاترین **ارزش** Q را در مقایسه با سایر اقدامات در آن حالت دارد:

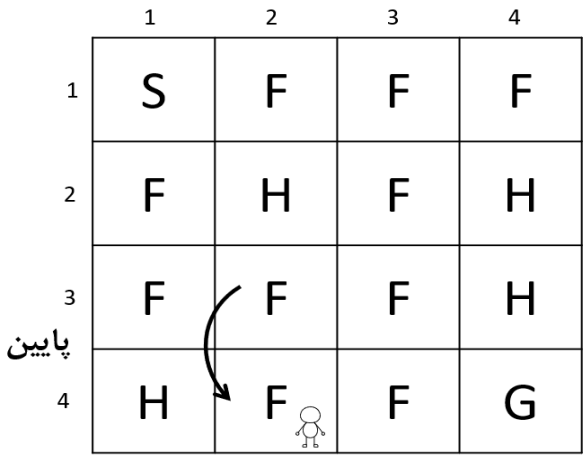
ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۱	بالا	(۳،۲)
۰.۸	**پایین**	**(۳،۲)**
۰.۵	چپ	(۳،۲)
۰.۶	راست	(۳،۲)
⋮	⋮	⋮
۰.۵	راست	(۴،۴)



شکل ۵.۱۴: اقدام با حداکثر ارزش Q در حالت (۳،۲) را انجام می دهیم.

بسیار خب، همانطور که شکل ۵.۱۵ نشان می‌دهد، ما اقدام حرکت به پایین را در حالت (۳،۲) انجام می‌دهیم و به حالت بعدی (۴،۲) می‌رویم.

ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۱	بالا	(۳،۲)
۰.۸	**پایین**	**(۳،۲)**
۰.۵	چپ	(۳،۲)
۰.۶	راست	(۳،۲)
⋮	⋮	⋮
۰.۵	راست	(۴،۴)



شکل ۵.۱۵: به پایین حرکت می‌کنیم و به حالت (۴،۲) می‌رویم.

بنابراین، ما در حالت (۳،۲) به حالت بعدی (۴،۲) حرکت کرده و یک پاداش r برابر صفر دریافت می‌کنیم. بیایید نرخ یادگیری α را ۰.۱، و ضریب تخفیف γ را برابر یک قرار دهیم. حالا چگونه می‌توانیم به‌روزرسانی ارزش Q را انجام دهیم؟

بیایید قاعده به‌روزرسانی یادگیری Q خود را به یاد بیاوریم:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

زوج حالت–اقدام $Q(s,a)$ را با $Q((s,a),$ پایین$)$ و حالت بعدی s' را با (۴،۲) در معادله قبلی جایگزین کرده و می‌توانیم بنویسیم:

$$Q((۳،۲),\text{پایین}) = Q((۳،۲),\text{پایین}) + \alpha\left(r + \gamma \max_{a'} Q((۴،۲),a') - Q((۳،۲),\text{پایین})\right)$$

با جایگزینی پاداش $r = ۰$، نرخ یادگیری $\alpha = ۰.۱$، و ضریب تخفیف $\gamma = ۱$، می‌توانیم بنویسیم:

$$Q((۳،۲),\text{پایین}) = Q((۳،۲),\text{پایین}) + ۰.۱ * \left(۰ + ۱ \times \max_{a'} Q((۴،۲),a') - Q((۳،۲),\text{پایین})\right)$$

از جدول Q قبلی، می‌توان مشاهده کرد که ارزش Q مربوط به $Q((۳،۲),$ پایین$)$ برابر ۰.۸ می‌شود. بنابراین با جایگزینی $Q((۳،۲),$ پایین$)$ با ۰.۸، می‌توانیم معادله قبلی را به صورت زیر بازنویسی کنیم:

$$Q((۳,۲), \text{پایین}) = ۰.۸ + ۰.۱\left(۰ + ۱ \times \max_{a'} Q((۴,۲), a') - ۰.۸\right)$$

همانطور که مشاهده کردید، در معادله قبلی عبارت $\max_{a'} Q((۴,۲), a')$ را داریم که نشان‌دهنده ارزش Q مربوط به زوج حالت–اقدام بعدی حالت جدید (۴،۲) است که به آن منتقل شده‌ایم. برای محاسبه ارزش Q حالت بعدی، ابتدا باید یک اقدام را انتخاب کنیم. در اینجا، یک اقدام را با استفاده از **خطمشی حریصانه** برمی‌گزینیم، یعنی اقدامی که حداکثر ارزش Q را دارد.

همانطور که شکل ۵.۱۶ نشان می‌دهد، اقدام حرکت به سمت راست، بیشینه ارزش Q را در حالت (۴،۲) دارد. بنابراین، اقدام حرکت به راست را گزینش کرده و ارزش Q زوج حالت–اقدام بعدی را به روز می‌کنیم:

	1	2	3	4
1	S	F	F	F
2	F	H	F	H
3	F	F	F	H
4	H	F	F	G

ارزش	اقدام	حالت
۰.۵	بالا	(۱،۱)
⋮	⋮	⋮
۰.۳	بالا	(۴،۲)
۰.۵	پایین	(۴،۲)
۰.۱	چپ	(۴،۲)
۰.۸	راست	(۴،۲)
⋮	⋮	⋮
۰.۵	راست	(۴،۴)

شکل ۵.۱۶: اقدام با بیشینه ارزش Q در حالت (۴،۲) را انجام می‌دهیم.

بنابراین، اکنون قاعده به‌روزرسانی ما به این صورت می‌شود:

$$Q((۳,۲), \text{پایین}) = ۰.۸ + ۰.۱\left(۰ + ۱ \times Q\left((۴,۲), \text{راست}\right) - ۰.۸\right)$$

از جدول Q قبلی، می‌توان مشاهده کرد که ارزش Q مربوط به $\left((۴,۲), \text{راست}\right)$ برابر ۰.۸ است. بنابراین، با جایگزینی ارزش $Q\left((۴,۲), \text{راست}\right)$ با ۰.۸، می‌توانیم معادله فوق را به صورت زیر بازنویسی کنیم:

$$Q\left((۳,۲), \text{پایین}\right) = ۰.۸ + ۰.۱(۰ + ۱ \times ۰.۸ - ۰.۸)$$

بنابراین، ارزش Q برابر می‌شود با:

$$Q((۳,۲), \text{پایین}) = ۰.۸$$

به طور مشابه، ارزش Q را برای همه زوج‌های حالت–اقدام، به‌روزرسانی می‌کنیم. یعنی یک اقدام در محیط را با استفاده از **خط‌مشی اپسیلون حریصانه**، انتخاب کرده و در حین به‌روزرسانی ارزش Q مربوط به زوج حالت–اقدام بعدی، از **خط‌مشی کاملا حریصانه** استفاده می‌کنیم. بنابراین، ما **ارزش** Q را برای هر زوج حالت–اقدام، به روز می‌رسانیم.

به این ترتیب، **تابع** Q با به‌روزرسانی **ارزش** Q زوج حالت–اقدام در هر مرحله از اپیزود، به‌هنگام می‌شود. ما یک **خط‌مشی** جدید از **تابع** Q به‌روز شده در هر مرحله از اپیزود، استخراج کرده و از این **خط‌مشی** جدید استفاده می‌کنیم. (به یاد داشته باشید که ما یک اقدام در محیط را با استفاده از **خط‌مشی اپسیلون حریصانه** انتخاب می‌کنیم، اما هنگام به‌روزرسانی **ارزش** Q مربوط به زوج حالت–اقدام بعدی، از **خط‌مشی مطلقا حریصانه** استفاده می‌کنیم). پس از چندین اپیزود، تابع Q بهینه را خواهیم داشت. الگوریتم یادگیری Q ارائه شده در ادامه، به ما در درک بهتر این موضوع کمک می‌کند.

الگوریتم یادگیری Q به صورت زیر ارائه شده است:

۱. یک **تابع** $Q(s,a)$ را با مقادیر تصادفی (دلبخواه) مقداردهی اولیه کنید.
۲. برای هر اپیزود:
  ۱. s را مقداردهی اولیه کنید.
  ۲. برای هر مرحله از پردینه:
    ۱. یک **خط‌مشی** را از $Q(s,a)$ استخراج کنید و یک **اقدام** a را برای انجام در حالت s انتخاب کنید.
    ۲. **اقدام** a را انجام دهید، به حالت بعدی s' بروید و **پاداش** r را مشاهده کنید.
    ۳. **ارزش** Q را به این صورت به‌روز کنید:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

    ۴. $s = s'$ را به روز کنید (حالت بعدی s' را به حالت فعلی s به‌روز کنید)
  ۵. اگر s حالت پایانی نیست، مراحل ۱ تا ۵ را تکرار کنید

اکنون که نحوه عملکرد الگوریتم **یادگیری** Q را آموختیم، در بخش بعدی، **یادگیری** Q را برای یافتن **خط‌مشی بهینه** پیاده‌سازی می‌کنیم.

محاسبه خط‌مشی بهینه با استفاده از یادگیری Q

اکنون بیایید **یادگیری** Q را برای یافتن **خط‌مشی** بهینه در محیط دریاچه یخ‌زده پیاده‌سازی کنیم.

ابتدا بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import numpy as np
import random
```

اکنون محیط دریاچه یخ‌زده را با استفاده از جیم، ایجاد می‌کنیم:

```
env = gym.make('FrozenLake-v0')
```

بیایید دیکشنری را برای ذخیره **ارزش‌های** Q زوج‌های حالت–اقدام تعریف کنیم و **ارزش‌های** Q همه زوج‌های حالت–اقدام را برابر صفر مقداردهی اولیه کنیم:

```
Q = {}
for s in range(env.observation_space.n):
    for a in range(env.action_space.n):
        Q[(s,a)] = 0.0
```

حالا بیایید سیاست **اپسیلون حریصانه** را تعریف کنیم. یک عدد تصادفی از توزیع یکنواخت تولید می‌کنیم. اگر عدد تصادفی کمتر از اپسیلون باشد، اقدام تصادفی را انتخاب می‌کنیم، در غیر این صورت بهترین اقدام را که حداکثر ارزش Q را داشته باشد، انتخاب می‌کنیم:

```
def epsilon_greedy(state, epsilon):
    if random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return max(list(range(env.action_space.n)), key = lambda x:
Q[(state,x)])
```

ضریب تخفیف γ، نرخ یادگیری α و ارزش اپسیلون را مقداردهی اولیه کنید:

```
alpha = 0.85
gamma = 0.90
epsilon = 0.8
```

تعداد پردینه‌ها و تعداد مراحل زمانی را در پردینه تنظیم کنید:

```
num_episodes = 50000
num_timesteps = 1000
```

خطمشی را محاسبه کنید.

برای هر پردینه:

```
for i in range(num_episodes):
```

با تنظیم مجدد محیط، حالت را مقداردهی اولیه کنید:

```
    s = env.reset()
```

برای هر مرحله از پردینه:

```
    for t in range(num_timesteps):
```

اقدام را با استفاده از **خطمشی اپسیلون حریصانه** انتخاب کنید:

```
        a = epsilon_greedy(s,epsilon)
```

اقدام انتخاب شده را انجام دهید و اطلاعات حالت بعدی را ذخیره کنید:

```
        s_, r, done, _ = env.step(a)
```

حال، بیایید ارزش Q زوج حالت–اقدام را به این صورت محاسبه کنیم:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

ابتدا اقدام a'ای را که حداکثر ارزش Q را در حالت بعدی s' دارد، انتخاب کنید:

```
        a_ = np.argmax([Q[(s_, a)] for a in range(env.action_space.n)])
```

اکنون می‌توانیم ارزش Q زوج حالت–اقدام را به صورت زیر محاسبه کنیم:

```
        Q[(s,a)] += alpha * (r + gamma * Q[(s_,a_)]-Q[(s,a)])
```

$s = s'$ را به‌روزرسانی کنید. (حالت بعدی s' را به حالت فعلی به‌روز کنید):

```
        s = s_
```

اگر حالت فعلی، حالت نهایی است، متوقف شوید:

```
        if done:
            break
```

پس از تمام تکرارها، تابع Q بهینه را خواهیم داشت. سپس می‌توانیم خط‌مشی بهینه را با انتخاب اقدامی که بیشینه ارزش Q را در هر حالت دارد، استخراج کنیم.

تفاوت بین یادگیری Q و سارسا

درک تفاوت بین **یادگیری** Q و **سارسا** بسیار مهم است. بنابراین، بیایید در مورد تفاوت **یادگیری** Q و **سارسا**، خلاصه‌ای کوتاه ارائه کنیم.

سارسا یک **الگوریتم با خط‌مشی همگاه (هیان-سیاست) است**، به این معنی که ما از یک **خط‌مشی اپسیلون حریصانه** برای انتخاب یک اقدام در محیط و همچنین برای محاسبه **ارزش** Q زوج حالت–اقدام بعدی استفاده می‌کنیم. قاعده به‌روزرسانی **سارسا** به شرح زیر است:

$$Q(s,a) = Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big)$$

یادگیری Q یک **الگوریتم با خط‌مشی ناهمگاه (نهان-سیاست)** است، به این معنی که ما از یک **خط‌مشی اپسیلون حریصانه** برای انتخاب یک اقدام در محیط استفاده می‌کنیم، اما برای محاسبه **ارزش** Q زوج حالت–اقدام بعدی از یک **خط‌مشی مطلقا حریصانه** استفاده می‌کنیم. قاعده به‌روزرسانی **یادگیری** Q به شرح زیر است:

$$Q(s,a) = Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right)$$

مقایسه روش‌های برنامه‌ریزی پویا، مونت کارلو و تفاوت زمانمند

تاکنون چندین الگوریتم یادگیری تقویتی جالب و مهم مانند **برنامه‌ریزی پویا** (**DP**) شامل دو زیر روش (***تکرار ارزش*** و ***تکرار خط‌مشی***)، روش‌های **مونت‌کارلو** (**MC**) و روش‌های یادگیری **تفاوت زمانمند** (**TD**) یا تفاضل زمانی را برای یافتن خط‌مشی بهینه یاد گرفته‌ایم. این‌ها را الگوریتمهای کلیدی در یادگیری تقویتی کلاسیک می‌نامند و درک تفاوت بین این سه الگوریتم بسیار مهم است. بنابراین، در این بخش، تفاوت‌های بین روش‌های یادگیری برنامه‌ریزی پویا، مونت‌کارلو و تفاوت زمانمند را مرور می‌کنیم.

برنامه‌ریزی پویا (DP)، یعنی روش‌های ***تکرار ارزش*** و ***تکرار خط‌مشی***، که یک روش مبتنی بر مدل محیط است. به این معنی که ما خط‌مشی بهینه را با استفاده از پویایی‌های مدل محیط محاسبه می‌کنیم. وقتی پویایی‌های مدل محیط را نداریم، نمی‌توانیم روش برنامه‌ریزی پویا را اعمال کنیم.

همچنین با **روش مونت‌کارلو (MC)** آشنا شدیم. مونت‌کارلو یک روش بدون مدل است، به این معنی که ما خط‌مشی بهینه را بدون استفاده از پویایی‌های مدل محیط محاسبه می‌کنیم. اما یک مشکلی که در روش مونت‌کارلو با آن روبرو هستیم این است که فقط برای کارهای اپیزودیک و نه برای کارهای مداوم و مستمر قابل استفاده است.

ما با روش جالب بدون مدل دیگری به نام **یادگیری تفاوت زمانمند یا تفاضل زمانی (TD)** آشنا شدیم. یادگیری تفاوت زمانمند، هم از روش برنامه‌ریزی پویا با ویژگی خود–راه‌اندازی (بوت‌استرپ کردن) و هم از روش مونت‌کارلو با ویژگی بدون مدل بودن (آزاد از مدل)، بهره می‌برد.

به خاطر یادگیری همه الگوریتم‌های مهم یادگیری تقویتی به شما تبریک می‌گوییم. در فصل بعد، ما به یک مطالعه موردی به نام مسئله ماشین بخت‌آزمایی چند دسته[۱] خواهیم پرداخت.

خلاصه

ما این فصل را با درک یادگیری تفاوت زمانمند شروع کردیم (ایده اصلی در رویکرد یادگیری تفاوت‌زمانمند، **یادگیری براساس تفاوت بین پیش‌بینی‌های پی‌درپی زمانی** است) و در ادامه چگونگی استفاده این روش از دو روش دیگر یعنی برنامه‌ریزی پویا و مونت‌کارلو را گفتیم. ما یاد گرفتیم که، درست مانند برنامه‌ریزی پویا، یادگیری تفاوت زمانمند از تکنیک خود–راه‌اندازی (ابتناگری با برپاگری) بهره می‌برد، و درست مانند روش مونت‌کارلو، یادگیری تفاوت زمانمند یک روش بدون مدل است.

در ادامه یاد گرفتیم که چگونه **وظیفه پیش‌بینی** را با استفاده از یادگیری تفاوت زمانمند انجام دهیم و سپس الگوریتم **روش پیش‌بینی** تفاوت زمانمند را بررسی کردیم.

در ادامه، یاد گرفتیم که چگونه از ***یادگیری تفاوت زمانمند برای یک کار کنترلی*** استفاده کنیم. ابتدا با روش کنترل تفاوت زمانمند با داشتن **خط‌مشی همگاه** (***عیان-سیاست***) به نام ***سارسا*** آشنا شدیم و سپس روش کنترل تفاوت زمانمند با داشتن **خط‌مشی ناهمگاه** (***نهان-سیاست***) به نام ***یادگیری*** Q را یادگرفتیم. ما همچنین یاد گرفتیم که چگونه با استفاده از روش‌های ***یادگیری*** Q و ***سارسا***، خط‌مشی بهینه را در محیط دریاچه یخ‌زده پیدا کنیم.

ما بعلاوه تفاوت بین روش‌های ***یادگیری*** Q و ***سارسا*** را یاد گرفتیم. ما فهمیدیم که ***سارسا*** یک الگوریتم **خط‌مشی همگاه** است، به این معنی که از یک خط‌مشی **حریصانه اپسیلون** یکتا و یگانه برای انتخاب یک اقدام در محیط و همچنین برای محاسبه **ارزش** Q زوج حالت–اقدام بعدی استفاده می‌کنیم، در حالی که ***یادگیری*** Q یک الگویتم **خط‌مشی ناهمگاه** است، به این معنی که از یک **خط‌مشی اپسیلون حریصانه** برای انتخاب یک اقدام در محیط استفاده می‌کنیم، اما برای محاسبه **ارزش**

[۱] Multi-Armed Bandit

Q زوج حالت–اقدام بعدی از یک **سیاست مطلقا حریصانه** استفاده می‌کنیم. در پایان فصل، روش‌های برنامه‌ریزی پویا (**DP**)، مونت کارلو (**MC**) و تفاوت زمانمند (**TD**) را با هم مقایسه کردیم.

در فصل بعدی به مسئله جالبی به نام مسئله **ماشین بخت‌آزمایی چند دسته** خواهیم پرداخت.

سوالات

بیایید دانش جدید خود را با پاسخ به سؤالات زیر ارزیابی کنیم:

۱. یادگیری تفاوت زمانمند (تفاضل زمانی) چه تفاوتی با روش مونت کارلو دارد؟
۲. مزیت استفاده از روش یادگیری تفاوت زمانمند چیست؟
۳. خطای تفاوت زمانمند چیست؟
٤. قاعده به‌روزرسانی یادگیری تفاوت زمانمند چیست؟
٥. روش **پیش‌بینی** تفاوت زمانمند چگونه کار می‌کند؟
٦. **سارسا** چیست؟
۷. **یادگیری** Q چه تفاوتی با **سارسا** دارد؟

برای مطالعه بیشتر

برای اطلاعات بیشتر به لینک زیر مراجعه کنید:

- **Learning to Predict by the Methods of Temporal Differences** by *Richard S. Sutton*, available at https://link.springer.com/content/pdf/10.1007/BF00115009.pdf

پیوست فصل پنجم

اندکی بیشتر در باره مفهوم تفاوت زمانمند

یادگیری تفاوت زمانی یک روش پیش‌بینی است. این روش به صورت عمده برای حل مسائل **یادگیری تقویتی** مورد استفاده بود است. روش تفاوت زمانی ترکیبی از ایده‌های **مونت کارلو و برنامه‌ریزی پویا** است. این روش مشابه **روش مونت کارلو** است چرا که یادگیری در آن با استفاده از نمونه برداری از محیط با توجه به یک یا چند **سیاست خاص** انجام می‌شود. روش تفاوت زمانی به این دلیل به تکنیک‌های **برنامه‌ریزی پویا** شباهت دارد که این روش تخمین کنونی را بر اساس تخمین‌های یادگیری شده (فرایندی که به خودراه اندازه معروف است) به دست می‌آورد. الگوریتم یادگیری تفاوت زمانی به مدل یادگیری تفاوت زمانی در حیوانات نیز مرتبط است.

به عنوان یک روش پیش‌بینی، یادگیری تفاوت زمانی این واقعیت را در نظر می‌گیرد که پیش‌بینی‌های آینده نیز معمولاً از جهاتی دارای همبستگی هستند. در روش‌های یادگیری مبتنی بر پیش‌بینی نظارتی، عامل تنها از مقادیر دقیقاً مشاهده شده یاد می‌گیرد: یک پیش‌بینی انجام می‌شود، و زمانی که مشاهده ممکن باشد، پیش‌بینی به تطابق بهتری با مشاهده خواهد رسید. ایده اساسی یادگیری تفاوت زمانی این است که پیش‌بینی‌ها با پیش‌بینی‌هایی دقیق‌تر دیگری از آینده تنظیم کنیم. همان گونه که از مثال زیر بر می‌آید این رویه نوعی از فرایند خود راه اندازه است:

> فرض کنید که می خواهید وضعیت هوای روز شنبه را پیش‌بینی کنید و مدلی دارید که هوای روز شنبه را با استفاده از وضعیت هوای داده شده برای تمام روزهای هفته، پیش‌بینی می‌کند. در شرایط عادی، باید تا شنبه صبر کنید تا بتوانید تمامی مدل‌های خود را تنظیم نمایید. با این وجود، زمانی که مثلاً جمعه است، می‌توانید ایده بسیار خوبی از این داشته باشید که هوای روز شنبه احتمالاً به چه صورتی خواهد بود و به همین صورت می‌توانید مثلاً مدل روز شنبه خود را قبل از این که شنبه برسد، تغییر دهید.

به بیان ریاضی، هم در رویکرد استاندارد و هم در رویکرد تفاوت زمانی، تلاش ما بر این است که **تابع هزینه** را که مرتبط با خطاهای ما در پیش‌بینی یک یا چند **متغیر تصادفی** $E[z]$ است، بهینه‌سازی نماییم. حال آن که در رویکرد استاندارد به گونه‌ای فرض می نماییم که $E[z]=z$ (که z همان متغیر مشاهده شده‌است) و در رویکرد **TD** از یک مدل استفاده می نماییم. برای حالت خاص در یادگیری تقویتی، که کاربرد عمده روش‌های تفاوت زمانی است، z همان بازگشت کل و $E[z]$ با استفاده از **معادله بلمن** بازگشت داده شده‌است.

ایده اصلی در رویکرد یادگیری تفاوت‌زمانی، یادگیری براساس تفاوت بین پیش‌بینی‌های پی‌درپی زمانی است و برای بروزرسانی نیازی به صبرکردن تا پایان مسیر نیست. به عبارت دیگر، هدف از یادگیری این است که پیش‌بینی کنونی یادگیرنده برای الگوی فعلی ورودی، بیشتر با پیش‌بینی بعدی در مرحله بعدی مطابقت داشته باشد.

الگوریتم تفاوت زمانی در علوم عصبی

الگوریتم تفاوت زمانی در زمینه **علوم عصبی** نیز مورد توجه خاصی بوده‌است. پژوهشگران دریافته‌اند که نرخ ارسال الکتریکی **نورون‌های** پخش‌کننده دوپامین در ناحیه تگمنتوم شکمی و جسم سیاه را می‌توان به تابع خطای این الگوریتم نسبت داد. تابع خطا، میزان تفاوت میان **پاسخ**[۱] تخمین زده شده در هر **حالت**[۲] داده شده یا زمان خاصی و پاسخ دقیقی که به دست آمده را نشان می‌دهد. هر چه قدر این تابع بزرگ‌تر باشد، تفاوت میان پاسخ به دست آمده و مورد نظر بیشتر بوده‌است. زمانی که این تابع با محرکی که پاسخ آینده را به صورت دقیق منعکس می‌کند، خطا می‌تواند برای نسبت دادن آن محرک به پاسخ آینده استفاده شود.

به نظر می‌رسد که سلول‌های **دوپامین** نیز به صورت مشابهی عمل می‌کنند. در یکی از آزمایش‌های انجام شده، اندازه‌گیریهایی از سلول‌های دوپامین در یک میمون در حال آموزش انجام شد تا بتوان یک **محرک** را با **پاسخ** (جایزه) مربوط به آن که آب میوه بود، مرتبط کنند. در ابتدا نرخ ارسال الکتریکی سلول‌های دوپامین زمانی که میمون با آب میوه مواجه می‌شد، افزایش یافت که نشان می‌دهد که تفاوتی در پاسخ‌های مورد نظر و واقعی وجود دارد. در طول زمان، این ارسال به سمت اولین محرک مطمئن برای پاسخ بازگشت. به محض این که میمون به صورت کامل آموزش دید، هیچ افزایشی در در نرخ ارسال در هنگام مواجه با یک پاسخ مورد انتظار نبود. در ادامه، نرخ ارسال الکتریکی برای سلول‌های دوپامین، زمانی که پاسخ مورد نظر دریافت نشد، به زیر سطح فعال شدن کاهش یافت. این یافته‌ها تا حد زیادی با **تابع خطا** در یادگیری تفاوت زمانی که در زمینه **یادگیری تقویتی** مطرح است، مرتبط شده‌است.

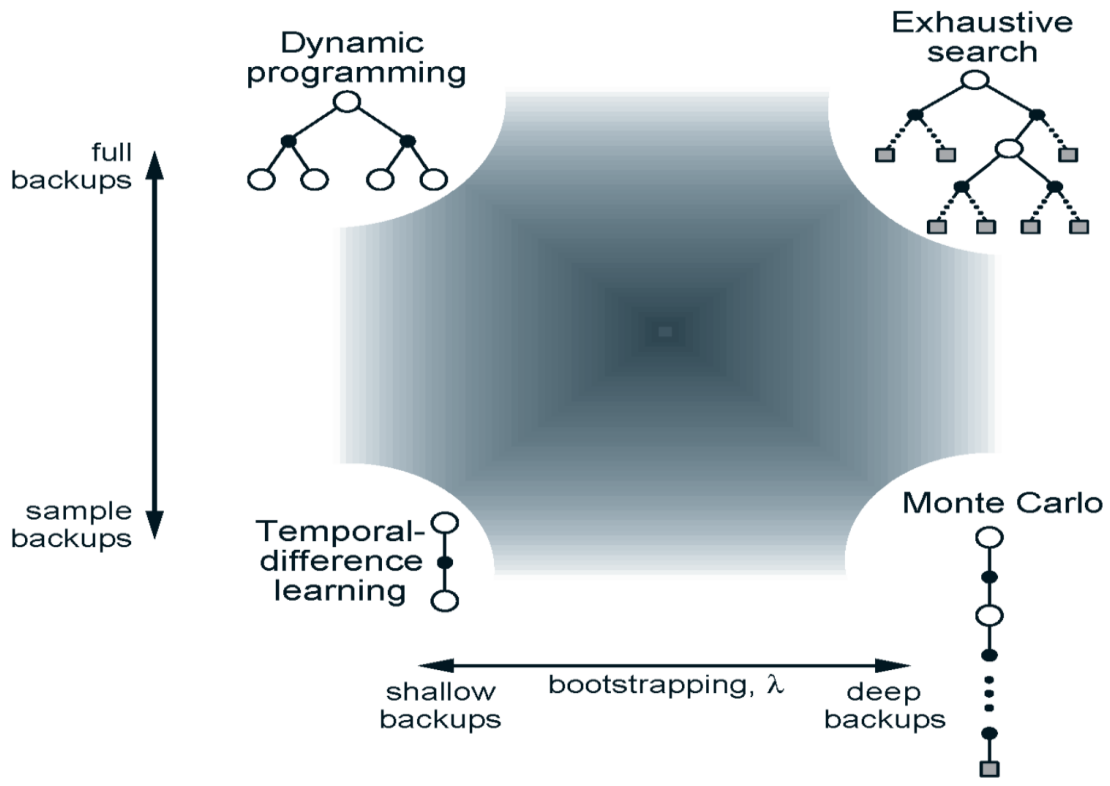
---

[۱] Reward
[۲] State

رابطه میان این مدل و کارکردهای بالقوه نورولوژیکی زمینه پژوهشی را به وجود آورده که هدف در استفاده از **TD** برای توضیح بسیاری از جنبه‌های پژوهش‌های رفتاری را دارد. این رابطه همچنین برای مطالعه شرایطی مانند اسکیزوفرنی و تبعات دستکاری‌های دارویی سطح دوپامین در یادگیری، مورد استفاده قرار گرفته‌است.

مقایسه کلی روش‌های یادگیری تقویتی

تمامی **روش‌های یادگیری تقویتی** دارای ایده‌ای یکسان هستند. اول، هدف تمام آن‌ها تخمین **تابع ارزش** است. دوم، تمامی آن‌ها با نگهداری مقادیر در عبور از حالت‌های مختلف ممکن یا واقعی عمل می‌کنند. سوم، تمامی آن‌ها یک راهبرد برای پیمایش کلی **سیاست** (**GPI**) دارند، به این معنی که آن‌ها یک **تابع ارزش تقریبی** و یک **سیاست تقریبی** را نگه می‌دارند و پیوسته هر کدام از آن‌ها را بر مبنای دیگری بهبود می‌بخشند.



تصویری کلی از روش‌های یادگیری تقویتی

دو بعد مهم در روش‌های یادگیری تقویتی در شکل بالا مشاهده می‌شود. این ابعاد با نوع نگهداری که برای بهبود تابع ارزش استفاده شده‌است، مرتبط هستند. بُعد عمودی نشان دهنده این است که نگهداری آیا در نمونه‌هاست (بر اساس گذر از نمونه‌ها)

یا نگهداری کامل است (بر اساس توزیع گذرهای ممکن). نگهداری‌های کامل نیازمند یک مدل هستند حال آن که نگهداری‌های نمونه‌ای می‌توانند بدون مدل نیز عمل کنند. بُعد افقی به عمق این نگهداری‌ها، یعنی درجه خود–راه اندازی یا خود–برپایی یا خود–ابتنائی[۱]، مرتبط است. سه گوشه از چهار گوشه اشاره شده از روش‌های مهم در تخمین ارزش‌ها هستند: **برنامه‌ریزی پویا، یادگیری تفاوت زمانی و الگوریتم مونت کارلو.**

تفاوت میان خود–راه‌اندازی و نمونه‌گیری

خود–راه‌اندازی یا خود–برپایی (بوت استرپینگ): وقتی چیزی را بر اساس یک تخمین دیگر، تخمین می زنید. به عنوان مثال، در روش Q-learning، این موضوع زمانی اتفاق می‌افتد که برآورد پاداش فعلی خود (r_t) را با افزودن عبارت تصحیح، یعنی

$$\max a' Q(s', a')$$

که حداکثر ارزش اقدام در میان تمام اقدامات حالت بعدی است، تغییر می دهید، اساسا شما ارزش فعلی اقدام خود یعنی Q را با استفاده از برآوردی از آینده Q تخمین می زنید.

نمونه برداری: نمونه‌ها را به عنوان تحقق (مقادیر مختلف) یک تابع تصور کنید. بسیاری از اوقات برآورد یا ارائه یک بیان تحلیلی از فرآیند اساسی که مشاهدات شما را ایجاد کرده است واقعا دشوار است. با این حال، مقادیر نمونه‌برداری می‌تواند به شما کمک کند تا بسیاری از ویژگیهای مکانیسم مولد زیربنایی را تعیین کنید و حتی ویژگیهای آن را فرض کنید. نمونه‌گیری می تواند به اشکال مختلفی در **RL** باشد. به عنوان مثال، یادگیری Q یک تخمین نقطه‌ای مبتنی بر نمونه از تابع ارزش اقدام بهینه (معادله بلمن) است. در دنیایی که عامل شما چیزی نمی‌داند، نمی‌توانید از برنامه‌ریزی پویا برای تعیین پاداش مورد انتظار از هر حالت استفاده کنید. بنابراین، شما باید «تجربه» را از دنیای خود نمونه بگیرید و پاداش مورد انتظار را از هر حالتی تخمین بزنید. در روشهای گرادیان سیاست، برای تعیین گرادیان پاداش مورد انتظار خود، باید مسیرها را روی حالتها و اقدامات از یک توزیع احتمال نمونه‌برداری کنید، زیرا نمی توانید آن را به صورت تحلیلی تعیین کنید.

[۱] Bootstrapping

فصل ششم

مطالعه موردی

مسئله بخت‌آزمایی چند_اهرمی

(MAB)

تا کنون در فصلهای قبلی، مفاهیم اساسی یادگیری تقویتی و همچنین چندین الگوریتم جالب یادگیری تقویتی را آموخته‌ایم. ما یک **روش مبتنی بر مدل** به نام **برنامه‌ریزی پویا** و یک **روش بدون مدل** به نام روش **مونت کارلو** یاد گرفتیم و سپس با روش **تفاوت زمانمند** آشنا شدیم که مزایای برنامه‌ریزی پویا و روش مونت کارلو را ترکیب می‌کند.

در این فصل، با یکی از مسائل کلاسیک در یادگیری تقویتی به نام مسئله **بخت‌آزمایی چند بازویی**[۱] (**MAB**) آشنا خواهیم شد. ما فصل را با درک مسئله **MAB** شروع کرده و سپس با چندین استراتژی اکتشافی (کاوشی) به نامهای اپسیلون–حریصانه، کاوش بیشینه–نرم[۲]، حد بالای اطمینان[۳] و نمونه‌گیری تامپسون[۴] برای حل مسئله **MAB** آشنا می‌شویم. پس از آن، یاد خواهیم گرفت که چرا دانستن **MAB** برای حل سایر مسائل در دنیای واقعی مفید است.

در ادامه خواهیم فهمید که چگونه با استفاده از تکنیک **MAB**، بهترین بنر تبلیغاتی را بیابیم تا اغلب کاربران روی آن کلیک کنند. در پایان فصل، با بخت‌آزمایی بافتار (زمینه‌ای)[۵] و نحوه استفاده از آن در موارد مختلف آشنا خواهیم شد.

در این فصل با موارد زیر آشنا می‌شویم:

- **مسئله ماشین بخت‌آزمایی چند–اهرمی (MAB)**
- **روش اپسیلون–حریصانه**
- **روش اکتشاف بیشینه–نرم (Softmax)**
- **الگوریتم حد بالای اطمینان**
- **الگوریتم نمونه گیری تامپسون**
- **کاربردهای MAB**
- **یافتن بهترین بنر تبلیغاتی با استفاده از MAB**
- **بخت‌آزمایی بافتاری (زمینه‌ای)**

---

۱ Multi-Armed Bandit (MAB)
بعنوان معادل فارسی این ترکیب می‌توان از ماشین بخت‌آزمایی چند اهرمی، چند دسته‌ای، چند دستی، چند بازویی، چند گزینه‌ای و امثال آن استفاده کرد.
۲ Softmax
۳ Upper Confidence Bound
۴ Thompson Sampling
۵ Contextual Bandits

مسئله ماشین بخت‌آزمایی چند–اهرمی (MAB)

مسئله **MAB** یکی از مسائل کلاسیک در یادگیری تقویتی است. **MAB**، یک دستگاه بخت‌آزمایی یا یک ماشین چاکدار (شکافدار)[۱] است که بازو (یا اهرم یا دسته) آنرا می‌کشیم و بر اساس برخی از توزیع احتمالات، یک پرداخت (یا پاداش) دریافت می‌کنیم. دستگاهی که تنها یک شکاف داشته باشد را ماشین بخت‌آزمایی تک–بازویی می‌نامند و زمانی که دستگاه چندین شکاف داشته باشد، به آن **MAB** یا دستگاه بخت‌آزمایی k–بازوئی می‌گویند که در آن k نشان دهنده تعداد دستگاههای شکافدار است.

شکل ۶.۱ یک دستگاه بخت‌آزمایی ۳ بازویی را نشان می‌دهد.

شکل ۶.۱: دستگاه‌های شکافدار بخت‌آزمایی۳ دسته‌ای

ماشینهای بخت‌آزمایی یکی از محبوب‌ترین بازیها برای امتحان شانس هستند که در آن بازو را می‌کشیم و پاداش

---

[۱] Slot Machine

ماشین اِسلات که آنرا به انگلیسی: (Slot Machine) به انگلیسی آمریکایی: فروت ماشین (Fruit Machine) به انگلیسی بریتانیایی: ماشین پوکر (Poker Machine) می‌نامند، یک ماشین بخت‌آزمایی است که یک بازی شانسی را برای مشتریان خود ایجاد می‌کند. ماشین‌های اسلات همچنین به‌عنوان "راهزنان یک دست" به دلیل داشتن یک اهرم مکانیکی بزرگ در کنار دستگاه و نیز توانایی بازی برای خالی کردن جیب و کیف پول بازیکنان مانند دزدان، مورد تحقیر قرار می‌گیرند.

چیدمان استاندارد یک ماشین اسلات دارای صفحه‌ای است که سه یا چند قرقره را نشان می‌دهد که هنگام فعال شدن بازی می‌چرخند. برخی از ماشینهای اسلات مدرن هنوز هم دارای یک اهرم به عنوان یک ویژگی طراحی آشنانمایی برای شروع بازی هستند. با این حال، مکانیک ماشینهای اولیه توسط مولدهای تولید اعداد تصادفی جایگزین شده‌اند و اکثر آنها اکنون با استفاده از دکمه‌ها و صفحه‌های لمسی کار می‌کنند.

می‌گیریم. اگر پاداش ۱- بگیریم بازی را باخته و اگر پاداش ۱+ بگیریم بازی را می‌بریم. ممکن است چندین دستگاه شکافدار وجود داشته باشد که از هر دستگاه شکافدار به عنوان یک بازو یاد می‌شود. به عنوان مثال، ماشین شکافدار ۱ به عنوان بازوی ۱، ماشین شکافدار ۲ به عنوان بازوی ۲ و غیره شناخته می‌شود. بنابراین، هر زمان که می‌گوییم بازوی n، در واقع به این معنی است که ما به دستگاه شکافدار nام اشاره می‌کنیم.

هر بازو این دستگاه، توزیع احتمال خاص خود را دارد که احتمال برد و باخت بازی آنرا نشان می‌دهد. به عنوان مثال، فرض کنید دو بازو داریم. اجازه دهید احتمال برنده شدن در صورت کشیدن بازوی ۱ (یعنی ماشین شکافدار ۱) معادل ۰.۷ و احتمال برنده شدن در صورت کشیدن بازوی ۲ (یعنی ماشین شکافدار ۲) برابر ۰.۵ باشد.

حال، اگر بازوی ۱ را بکشیم، ۷۰ درصد زمانها، بازی را می‌بریم و پاداش ۱+ را دریافت می‌کنیم، و اگر بازوی ۲ را بکشیم، ۵۰ درصد مواقع بازی را می‌بریم و پاداش ۱+ را دریافت می‌کنیم.

بنابراین، می‌توان گفت که کشیدن بازوی ۱ مطلوب است زیرا باعث می‌شود ۷۰ درصد مواقع بازی را ببریم. با این حال، ما توزیع احتمال بازو (دستگاه شکافدار) را نمی‌دانیم و آنرا به ما نخواهد داد. ما باید بفهمیم کدام بازو به ما کمک می‌کند تا بیشتر اوقات بازی را ببریم و پاداش خوبی به ما می‌دهد.

بسیار خب، چگونه می‌توانیم این بازو را پیدا کنیم؟

فرض کنید یک بار **بازوی** ۱ را کشیدیم و **پاداش** ۱+ دریافت کردیم و **بازوی** ۲ را هم یک بار کشیدیم و **پاداش** ۰ را دریافت کردیم. از آنجایی که **بازوی** ۱ **پاداش** ۱+ می‌دهد، نمی‌توانیم پس از تنها یک بار کشیدن آن، نتیجه‌گیری کنیم که **بازوی** ۱ بهترین بازو است. ما باید هر دو بازو را بارها بکشیم و میانگین **پاداش** ناشی از هر یک از بازوها را محاسبه کنیم و بعد از آن می‌توانیم بازویی را انتخاب کنیم که حداکثر میانگین **پاداش** را به عنوان بهترین **بازو** می‌دهد.

بیایید **بازو** را با a نشان دهیم و میانگین **پاداش** ناشی از کشیدن **بازو** a را به صورت زیر تعریف کنیم:

$$Q(a) = \frac{\text{Sum of rewards obtained from the arm}}{\text{Number of times the arm was pulled}}$$

که در آن $Q(a)$ نشان دهنده میانگین پاداش بازوی a است.

بازوی بهینه یا a^* آن بازوی است که **حداکثر میانگین پاداش** را به ما می‌دهد ، یعنی:

$$a^* = \arg\max_a Q(a)$$

بسیار خب، ما آموخته ایم که بازویی که حداکثر میانگین پاداش را می‌دهد بازوی بهینه است. اما چگونه می‌توانیم آن را پیدا کنیم؟

ما بازی را برای چندین دور انجام داده و در هر دور فقط می‌توانیم یک بازو را بکشیم. فرض کنید در دور اول، بازوی ۱ را می‌کشیم و پاداش آنرا مشاهده می‌کنیم و در دور دوم نیز بازوی ۲ را کشیده و باز هم پاداش دیگری را مشاهده می‌کنیم. به همین ترتیب، در هر دور، بازوی ۱ یا بازوی ۲ را کشیده و پاداش را مشاهده می‌کنیم. پس از اتمام چندین دور بازی، میانگین پاداش هر یک از بازوها را محاسبه کرده و سپس بازویی که حداکثر میانگین پاداش را دارد به عنوان بهترین بازو انتخاب می‌کنیم.

اما این روش خوبی برای یافتن بهترین بازو نیست. فرض کنید ما ۲۰ بازو داریم. اگر در هر دور به کشیدن بازوی متفاوتی ادامه دهیم، در بیشتر دورها، بازی را می‌بازیم و پاداش ۰ دریافت می‌کنیم. در کنار یافتن بهترین بازو، هدف ما باید به حداقل رساندن هزینه شناسایی بهترین بازو (یعنی کمتر کشیدن بازوهای بی پاداش) باشد. به این هزینه، معمولا **پشیمانی یا تاسف**[۱] (یا اتلاف) گفته می‌شود.

بنابراین، ما باید بهترین بازو را پیدا کرده اما در عین حال پشیمانی را به حداقل برسانیم. یعنی ما باید بهترین بازو را پیدا کنیم، اما نمی‌خواهیم در نهایت بازوهایی را انتخاب کنیم که باعث می‌شود در بیشتر دورها، بازی را از دست بدهیم.

حال سوال این است: آیا باید در هر دور، بازوی متفاوتی را کشف کنیم، یا باید فقط بازویی را انتخاب کنیم که در

[۱] Regret

دورهای قبلی پاداش خوبی برای ما به ارمغان آورده است؟ این منجر به وضعیتی به نام معضل یا دوگانهٔ اکتشاف–انتفاع یا (کاوش–یابش) یا (آینده–حال) یا (نون تره–نون کره) یا (گشتن–برداشتن) می‌شود که در فصل ۴، در مبحث **روشهای مونت کارلو** با آن آشنا شدیم. بنابراین، برای حل این مشکل، از روش **اپسیلون–حریصانه** استفاده می‌کنیم؛ یعنی بازویی که در دورهای قبلی پاداش خوبی برای ما به ارمغان آورده است، را با احتمال ε -۱ انتخاب کرده و بازوی دیگری را بطور تصادفی با احتمال ε انتخاب می‌کنیم. پس از اتمام چندین دور، بهترین بازو را به عنوان بازویی که حداکثر میانگین پاداش را دارد برمی‌گزینیم.

مشابه روش اپسیلون–حریصانه، چندین استراتژی اکتشاف مختلف وجود دارد که به ما کمک می‌کند تا بر معضل یا **دوگانه اکتشاف–انتفاع** غلبه کنیم. در بخش آینده، با جزئیات در مورد چندین استراتژی اکتشاف مختلف و اینکه چگونه به ما در یافتن بازوی بهینه کمک می‌کنند، بیشتر خواهیم آموخت، اما ابتدا بیایید به ایجاد ماشین بخت‌آزمایی در پایتون نگاه کنیم.

ایجاد ماشین بخت‌آزمایی در پایتون (جیم؟!)

قبل از ادامه، بیایید یاد بگیریم که چگونه با جعبه ابزار Gym یک محیط بخت‌آزمایی ایجاد کنیم. جیم با یک محیط بخت‌آزمایی از پیش بسته‌بندی ارائه نمی‌شود. بنابراین، ما باید یک محیط بخت‌آزمایی ایجاد کنیم و آن را با جیم ادغام کنیم. به جای ایجاد محیط بخت‌آزمایی از ابتدا، از نسخه منبع باز محیط بخت‌آزمایی ارائه شده توسط جسی کوپر استفاده خواهیم کرد.

ابتدا، بیایید مخزن بخت‌آزمایی Gym را شبیه‌سازی کنیم:

```
git clone https://github.com/JKCooper2/gym-bandits
```

در مرحله بعد، می‌توانیم آن را با استفاده از `pip` نصب کنیم:

```
cd gym-bandits
pip install -e .
```

پس از نصب، gym_bandits و همچنین کتابخانه gym را وارد می‌کنیم:

```
import gym_bandits
import gym
```

gym_bandits چندین نسخه از محیط بخت‌آزمایی را ارائه می‌دهد. ما می‌توانیم نسخه‌های مختلف بخت‌آزمایی را در https://github.com/JKCooper2/gym-bandits بررسی کنیم.

یک ماشین بخت‌آزمایی ساده با ۲ بازو ایجاد کرده که شناسه محیط آن بصورت زیر است:

BanditTwoArmedHighLowFixed-v0

```
env = gym.make("BanditTwoArmedHighLowFixed-v0")
```

از آنجا که ما یک ماشین بخت‌آزمایی ۲ بازوه ایجاد کردیم، فضای عمل ما ۲ خواهد بود (زیرا ۲ بازو یا ۲ دسته وجود دارد)، همانطور که در اینجا نشان داده شده است:

```
print(env.action_space.n)
```

با کد قبلی، مطلب زیر چاپ می‌شود:

```
2
```

همچنین می‌توانیم توزیع احتمال بازو را با موارد زیر بررسی کنیم:

```
print(env.p_dist)
```

با کد بالا، مطلب زیر چاپ می‌شود:

```
[0.8, 0.2]
```

این نشان می‌دهد که با بازوی اول در ۸۰ درصد مواقع بازی را می‌بریم و با بازوی دوم در ۲۰ درصد مواقع بازی را

می‌بریم. هدف ما این است که بفهمیم کشیدن کدام اهرم باعث می‌شود بیشتر اوقات بازی را ببریم.

اکنون که یاد گرفتیم چگونه محیطهای ماشین بخت‌آزمایی را در Gym ایجاد کنیم، در بخش بعدی به بررسی استراتژیهای مختلف اکتشافی برای حل مسئله **MAB** پرداخته و آنها را با Gym پیاده‌سازی می‌کنیم.

استراتژیهای اکتشاف

در ابتدای فصل، ما با مخمصه اکتشاف–انتفاع در مسئله **MAB** آشنا شدیم. برای غلبه بر این دوگانه، از استراتژیهای مختلف اکتشاف (یا کاوش و جستجو) استفاده می‌کنیم و بهترین بازو را پیدا می‌کنیم. استراتژیهای مختلف اکتشاف در اینجا فهرست شده است:

۱. **اپسیلون حریصانه**
۲. **اکتشاف بیشینه–نرم (Softmax)**
۳. **حد بالای اطمینان**
٤. **نمونه گیری تامسون**

اکنون، برای پیدا کردن بهترین بازو، همه این استراتژیهای اکتشاف را با جزئیات بررسی کرده و آنها را پیاده‌سازی می‌کنیم

استراتژی اپسیلون حریصانه

ما در فصلهای قبلی با الگوریتم اپسیلون–حریصانه آشنا شدیم. با اپسیلون– حریصانه، بهترین بازو را (با احتمال یک منهای اپسیلون) انتخاب کرده و یک بازوی تصادفی دیگر را (با احتمال اپسیلون) انتخاب می‌کنیم. بیایید یک مثال ساده بزنیم و یاد بگیریم که چگونه بهترین بازو را با روش اپسیلون– حریصانه با جزئیات بیشتری پیدا کنیم.

فرض کنید ما دو بازو داریم: بازوی ۱ و بازوی ۲. فرض کنید با بازوی یک، ۸۰ درصد مواقع بازی را می‌بریم و بازوی

دو ۲۰ درصد مواقع منجر به برد می‌شود. بنابراین، می‌توان گفت که بازوی ۱ بهترین بازو است زیرا باعث می‌شود ۸۰ درصد مواقع بازی را ببریم. اکنون، بیایید یاد بگیریم که چگونه این را با روش اپسیلون- حریصانه پیدا کنیم.

ما ابتدا پارامترهای `count` (تعداد دفعات کشیدن بازو)، `sum_rewards` (مجموع پاداشهای بدست آمده از کشیدن بازو) و Q (میانگین پاداش بدست آمده از کشیدن بازو) را مقداردهی اولیه می‌کنیم، همانطور که جدول ۶.۱ نشان می‌دهد:

بازو	تعداد	جمع پاداشهای بدست آمده	Q
بازوی یک	۰	۰	۰
بازوی دو	۰	۰	۰

جدول ۶.۱: متغیرها را با مقدار صفر راه اندازی کنید.

دور اول:

فرضا در دور اول بازی، یک **بازوی** تصادفی با احتمال اپسیلون انتخاب می‌کنیم. گیریم بازوی یک را به طور تصادفی کشیده و **پاداشی** را مشاهده می‌کنیم. فرض کنید پاداشی که با کشیدن بازوی یک به دست می‌آید ۱ باشد. بنابراین، ما جدول خود را با تنظیم `count` بازوی یک به عدد ۱ و `sum_rewards` بازوی یک بر روی عدد ۱ به‌روز می‌کنیم، و بنابراین میانگین **پاداش** Q **بازوی** یک، پس از دور اول ۱ است، همانطور که جدول ۶.۲ نشان می‌دهد:

بازو	تعداد	جمع پاداشهای بدست آمده	Q
بازوی یک	۱	۱	۱
بازوی دو	۰	۰	۰

جدول ۶.۲: نتایج پس از دور اول

دور دوم:

فرضا در دور دوم، بهترین اهرم را با احتمال (۱ منهای اپسیلون) انتخاب می‌کنیم. بهترین اهرم همان **بازویی** است که حداکثر میانگین **پاداش** را داشته است. بنابراین، جدول خود را بررسی کرده تا ببینیم کدام بازو حداکثر میانگین پاداش را دارد. از آنجایی که بازوی یک حداکثر میانگین پاداش را دارد، بازوی یک را کشیده و پاداش را مشاهده می‌کنیم. فرض کنید **پاداش** به دست آمده از کشیدن **بازوی** یک همان ۱ باشد.

بنابراین، ما جدول خود را با تنظیم `count` بازوی یک به عدد ۲ و `sum_rewards` بازوی یک را روی عدد ۲ به‌روز می‌کنیم، و لذا میانگین پاداش Q بازوی یک، همانطور که جدول ۶.۳ نشان می‌دهد، پس از دور دوم ۱ است:

بازو	تعداد دفعات کشیدن	جمع پاداشهای بدست آمده	Q
بازوی یک	۲	۲	۱
بازوی دو	۰	۰	۰

جدول ۶.۳: نتایج پس از دور دوم

دور سوم:

گیریم در دور سوم، یک بازوی تصادفی با احتمال اپسیلون انتخاب می‌کنیم. فرض کنید به طور تصادفی **بازوی** دو را کشیده و **پاداش** را مشاهده می‌کنیم. همچنین فرض کنید پاداشی که با کشیدن بازوی دو به دست می‌آید ۰ باشد. بنابراین، ما جدول خود را با تنظیم مقدار `count` بازوی دو، بر روی عدد ۱ و مقدار `sum_rewards` بازوی دو، بر روی عدد ۰ به‌روز می‌کنیم. با این حساب، میانگین پاداش Q بازوی دو پس از دور سوم معادل عدد ۰ است که جدول ۶.۴ آنرا نشان می‌دهد.

دور چهارم:

در دور چهارم، بهترین **بازو** را با احتمال ۱ منهای اپسیلون انتخاب می‌کنیم. بنابراین، **بازوی** یک را می‌کشیم زیرا

حداکثر میانگین **پاداش** را دارد. بگذارید پاداشی که با کشیدن بازوی ۱ به دست می‌آید این بار ۰ باشد.

بازو	تعداد دفعات کشیدن	جمع پاداشهای بدست آمده	Q
بازوی یک	۲	۲	۱
بازوی دو	۱	۰	۰

جدول ۶.۴: نتایج پس از دور سوم

اکنون، جدول خود را با تنظیم مقدار `count` بازوی یک، بر روی عدد ۳ و مقدار `sum_rewards` بازوی دو، بر روی ۲ به‌روز می‌کنیم. تا اینجا، میانگین پاداش Q بازوی یک پس از دور چهارم معادل ۰.۶۶ خواهد بود که در جدول ۶.۵ آمده است:

بازو	تعداد دفعات کشیدن	جمع پاداشهای بدست آمده	Q
بازوی یک	۳	۲	۰.۶۶
بازوی دو	۱	۰	۰

جدول ۶.۵: نتایج کار پس از دور چهارم

ما این روند را برای چندین دور تکرار می‌کنیم. یعنی برای چندین دور بازی، بهترین بازو را با احتمال ۱ منهای اپسیلون انتخاب می‌کنیم و بعلاوه یک بازوی تصادفی را با احتمال اپسیلون می‌کشیم.

جدول ۶.۶ جدول به روز شده را پس از ۱۰۰ دور بازی نشان می‌دهد:

بازو	تعداد دفعات کشیدن	جمع پاداشهای بدست آمده	Q
بازوی یک	۷۵	۶۱	۰.۸۱
بازوی دو	۲۵	۲	۰.۰۸

جدول ۶.۶: نتایج کار پس از دور صدم

از جدول ۶.۶ می‌توان نتیجه گرفت که بازوی ۱ بهترین بازو است زیرا حداکثر میانگین پاداش را دارد.

پیاده سازی اپسیلون_حریصانه

اکنون، بیایید یاد بگیریم که روش اپسیلون–حریصانه را برای یافتن بهترین بازو پیاده‌سازی کنیم. اول بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import gym_bandits
import numpy as np
```

برای درک بهتر، بیایید ماشین بخت‌آزمایی را تنها با دو بازو ایجاد کنیم:

```
env = gym.make("BanditTwoArmedHighLowFixed-v0")
```

بیایید توزیع احتمال بازو را بررسی کنیم:

```
print(env.p_dist)
```

بر اساس کد قبلی، چاپ می‌شود:

```
[0.8, 0.2]
```

می‌توانیم مشاهده کنیم که با بازوی ۱ به احتمال ۸۰ درصد بازی را می‌بریم و با بازوی ۲ به احتمال ۲۰ درصد بازی را می‌بریم. در اینجا، بهترین اهرم، بازوی ۱ است، زیرا با بازوی ۱ ما بازی را با احتمال ۸۰ درصد برنده می‌شویم. حالا بیایید ببینیم چگونه با استفاده از روش اپسیلون–حریصانه این بهترین بازو را پیدا کنیم.

ابتدا بیایید متغیرها را مقداردهی اولیه کنیم.

شمارش یا `count` را برای ذخیره تعداد دفعاتی که بازو کشیده می‌شود مقداردهی اولیه کنید:

```
count = np.zeros(2)
```

پارامتر `sum_rewards` را برای ذخیره مجموع جوایز هر بازو مقداردهی اولیه کنید:

```
sum_rewards = np.zeros(2)
```

Q را برای ذخیره میانگین پاداش هر بازو مقداردهی اولیه کنید:

```
Q = np.zeros(2)
```

تعداد دورها (تکرارها) را تنظیم کنید:

```
num_rounds = 100
```

حال، بیایید تابع `epsilon_greedy` را تعریف کنیم.

ابتدا یک عدد تصادفی از یک توزیع یکنواخت تولید می‌کنیم. اگر عدد تصادفی کمتر از اپسیلون باشد، بازوی تصادفی را می‌کشیم. در غیر این صورت، ما بهترین بازوی که دارای حداکثر میانگین پاداش است را می‌کشیم، همانطور که در اینجا نشان داده شده است:

```
def epsilon_greedy(epsilon):

    if np.random.uniform(0,1) < epsilon:
        return env.action_space.sample()
    else:
        return np.argmax(Q)
```

حالا بیایید بازی را انجام دهیم و سعی کنیم بهترین بازو را با استفاده از روش اپسیلون حریصانه پیدا کنیم.

برای هر دور:

```
for i in range(num_rounds):
```

بازو را بر اساس روش اپسیلون حریصانه انتخاب کنید:

```
    arm = epsilon_greedy(epsilon=0.5)
```

بازو را بکشید و پاداش و اطلاعات وضعیت بعدی را ذخیره کنید:

```
    next_state, reward, done, info = env.step(arm)
```

مقدار count را برای این بازو، یک واحد افزایش دهید:

```
    count[arm] += 1
```

مجموع جوایز بازو را به‌روز کنید:

```
    sum_rewards[arm]+=reward
```

میانگین پاداش بازو را به روز کنید:

```
    Q[arm] = sum_rewards[arm]/count[arm]
```

پس از تمام دورها، میانگین پاداش به دست آمده از هر یک از بازوها را بررسی می‌کنیم:

```
print(Q)
```

کد قبلی چیزی شبیه به این را چاپ می‌کند:

```
[0.83783784 0.34615385]
```

اکنون می‌توانیم بازوی بهینه را به عنوان بازوی که حداکثر میانگین پاداش را دارد انتخاب کنیم:

$$a^* = \arg\max_a Q(a)$$

از آنجایی که بازوی ۱ میانگین پاداش بالاتری نسبت به بازوی ۲ دارد، بازوی بهینه ما بازوی ۱ خواهد بود:

```
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

با کد قبلی، مطاب زیر چاپ می‌شود:

```
The optimal arm is arm 1
```

بنابراین، بازوی بهینه را با استفاده از روش اپسیلون- حریصانه پیدا کرده‌ایم.

استراتژی اکتشاف بیشینه_نرم

اکتشاف بیشینه–نرم (Softmax) که به عنوان روش جستجو یا اکتشاف بولتزمن[۱] نیز شناخته می‌شود، یکی دیگر از استراتژیهای اکتشاف مفید برای برای یافتن بازوی بهینه است.

در سیاست اپسیلون–حریصانه، یاد گرفتیم که بهترین بازو را با احتمال ۱ منهای اپسیلون و یک بازوی تصادفی با احتمال اپسیلون انتخاب می‌کنیم. همانطور که ممکن است متوجه شده باشید، در سیاست حریصانه اپسیلون، همه بازوهای غیر بهترین به طور مساوی مورد بررسی قرار می‌گیرند. یعنی همه بازوهای غیربهترین احتمال انتخاب یکنواختی دارند. به عنوان مثال، فرض کنید ما ۴ بازو داریم و بازو ۱ بهترین بازو است. سپس بازوهای غیر بهترین یعنی [بازوی ۲، بازوی ۳، بازوی ۴] را به طور یکنواخت بررسی می‌کنیم.

فرضا بازوی ۳ هرگز بازوی خوبی نیست و همیشه پاداش ۰ می‌دهد. در این مورد، به جای کاوش مجدد بازوی ۳، می‌توانیم زمان بیشتری را صرف کاوش در بازوی ۲ و بازوی ۴ کنیم. اما مشکل روش اپسیلون حریصانه این است که ما همه بازوهای غیر بهترین را به طور مساوی بررسی می‌کنیم. بنابراین، تمام بازوهای غیر بهترین یعنی [بازوی ۲، بازوی ۳، بازوی ۴] به طور مساوی مورد بررسی قرار خواهند گرفت.

برای جلوگیری از این امر، لازم است که بتوانیم بازوی ۲ و بازوی ۴ را بر بازوی ۳ اولویت دهیم. در این صورت می‌توانیم بازوی ۲ و بازوی ۴ را بیشتر از بازوی ۳ کاوش کنیم.

بسیار خب، اما چگونه به بازوها اولویت بدهیم؟ می‌توان با اختصاص یک احتمال به همه بازوها بر اساس میانگین پاداش Q به آنها بنوعی اولویت بدهیم. بازویی که حداکثر میانگین پاداش را داشته باشد، احتمال بالایی خواهد داشت و همه بازوهای غیر بهترین احتمال متناسب با میانگین پاداش خود دارند.

به عنوان مثال، همانطور که جدول ۶.۷ نشان می‌دهد، بازوی ۱ بهترین بازو است زیرا میانگین پاداش بالایی دارد. بنابراین، احتمال بالایی را به بازوی ۱ اختصاص می‌دهیم. بازوهای ۲، ۳ و ۴ غیربهترین بازوها هستند و ما باید آنها را

[۱] Boltzmann Exploration

بررسی کنیم. همانطور که می‌بینیم، بازوی ۳ دارای میانگین پاداش صفر است. بنابراین، به جای انتخاب همه بازوهای غیربهترین به طور یکنواخت، اولویت بیشتری به بازوهای ۲ و ۴ نسبت به بازوی ۳ می‌دهیم. بنابراین، احتمال بازوی ۲ و ۴ در مقایسه با بازوی ۳ زیاد خواهد بود:

بازو	Q
بازوی یک	۰.۸۴
بازوی دو	۰.۲۴
بازوی سه	۰.۰
بازوی چهار	۰.۱۳

جدول ۷-۶: میانگین پاداش برای یک ماشین بخت‌آزمایی با ۴ بازو

بنابراین، در روش اکتشاف بیشینه-نرم، بازوها را بر اساس مقدار احتمال انتخاب می‌کنیم. **در واقع احتمال کشیدن هر بازو با میانگین پاداش آن نسبت مستقیم دارد:**

$$p_t(a) \propto Q_t(a)$$

اما صبر کنید، احتمالات باید به ۱ برسد، درست است؟ میانگین پاداش (مقدار Q) به ۱ نمی‌رسد. بنابراین، همانطور که در اینجا نشان داده شده است، آنها را با تابع Softmax به احتمالات تبدیل می‌کنیم:

$$P_t(a) \propto \frac{\exp\big(Q_t(a)\big)}{\sum_{i=1}^{n} \exp\big(Q_t(i)\big)} \qquad (۱)$$

بنابراین، اکنون هر بازو بر اساس یک مقدار احتمال انتخاب خواهد شد. با این حال، در دورهای اولیه، میانگین پاداش صحیح هر بازو را نمی‌دانیم، بنابراین انتخاب بازو بر اساس احتمال میانگین پاداش در دورهای اولیه نادرست خواهد

بود. برای جلوگیری از این امر، پارامتر جدیدی به نام T را معرفی می‌کنیم. T را پارامتر دما[۱] می‌نامند.

همانطور که در اینجا نشان داده شده است می‌توان معادله قبلی را با دما T بازنویسی کرد:

$$P_t(a) \propto \frac{\exp(Q_t(a)/T)}{\sum_{i=1}^{n} \exp(Q_t(i)/T)} \qquad (۲)$$

خب، چگونه این T به ما کمک خواهد کرد؟ وقتی T بزرگ است، همه بازوها احتمال انتخاب شدن مساوی دارند و وقتی T کوچک باشد، بازویی که حداکثر میانگین پاداش را دارد، احتمال بالایی خواهد داشت. بنابراین، ما T را در دورهای اولیه روی یک عدد بالایی تنظیم می‌کنیم و پس از یک سری دور، مقدار T را کاهش می‌دهیم. این بدان معناست که در دور اولیه ما همه بازوها را به طور مساوی بررسی کرده و پس از یک سری دور، بازوی بهتر را انتخاب می‌کنیم که احتمال بالایی دارد.

بیایید این موضوع را با یک مثال ساده، درک کنیم. فرض کنید ما چهار بازو داریم، بازوی ۱ تا بازوی ۴. فرض کنید بازوی ۱ را بکشیم و پاداش ۱ دریافت کنیم. سپس میانگین پاداش بازوی یک معادل ۱ و میانگین پاداش سایر بازوها ۰ خواهد بود، همانطور که جدول ۶.۸ نشان می دهد:

بازو	Q
بازوی یک	۱
بازوی دو	۰
بازوی سه	۰
بازوی چهار	۰

جدول ۶.۸: میانگین پاداش هر بازو

حال، اگر میانگین پاداش را با استفاده از تابع بیشینه–نرم که در معادله (۱) داده شده را به احتمالات تبدیل کنیم، آنگاه احتمالات ما به صورت زیر است:

[۱] Temperature Parameter

بازو	Q	احتمال
بازوی یک	۱	۰.۴۷۵
بازوی دو	۰	۰.۱۷۴
بازوی سه	۰	۰.۱۷۴
بازوی چهار	۰	۰.۱۷۴

جدول ۶.۹: احتمال انتخاب هر بازو

همانطور که مشاهده می‌کنیم، احتمال ۴۷ درصد برای بازوی ۱ و احتمال ۱۷ درصد برای سایر بازوها داریم. اما ما نمی‌توانیم فقط با یک بار کشیدن بازوی ۱ احتمال بالایی را برای بازوی ۱ اختصاص دهیم. بنابراین، T را روی یک عدد بالا تنظیم می‌کنیم، مثلا $T = ۳۰$ و احتمالات را بر اساس معادله (۲) محاسبه می‌کنیم. اکنون احتمالات ما تبدیل می‌شود:

بازو	Q	احتمال
بازوی یک	۱	۰.۲۵۳
بازوی دو	۰	۰.۲۴۸
بازوی سه	۰	۰.۲۴۸
بازوی چهار	۰	۰.۲۴۸

جدول ۶.۱۰: احتمال انتخاب هر بازو با $T = ۳۰$

همانطور که می‌بینیم، اکنون همه بازوها احتمال انتخاب شدن یکسانی دارند. اکنون بازوها را بر اساس این احتمال بررسی می‌کنیم و در طی یک سری دور، مقدار T کاهش می‌یابد و احتمال بالایی برای بهترین بازو خواهیم داشت. بیایید فرض کنیم پس از حدود ۳۰ دور، میانگین پاداش همه بازوها به شرح زیر است:

بازو	Q
بازوی یک	۰.۸۴
بازوی دو	۰.۲۴
بازوی سه	۰.۰
بازوی چهار	۰.۱۳

جدول ۶.۱۱: میانگین پاداش برای هر بازو پس از ۳۰+ دور

ما یاد گرفتیم که مقدار T در طی چندین دور، کاهش می‌یابد. فرض کنید مقدار T به گونه‌ای کاهش یافته که اکنون ۰.۳ است، یعنی $T = ۳۰$. لذا احتمالات به مقادیر زیر تبدیل می‌شوند:

بازو	Q	احتمال
بازوی یک	۰.۸۴	۰.۷۷۵
بازوی دو	۰.۲۴	۰.۱۰۴
بازوی سه	۰.۰	۰.۰۴۷
بازوی چهار	۰.۱۳	۰.۰۷۲

جدول ۶.۱۲: احتمالات برای هر بازو با T اکنون روی ۰.۳ تنظیم شده است

همانطور که می‌بینیم ، بازوی ۱ در مقایسه با سایر بازوها، احتمال بالایی دارد. بنابراین، بازوی ۱ را به عنوان بهترین بازو انتخاب می‌کنیم و بازوی غیر بهترین یعنی [بازوی ۲، بازوی ۳، بازوی ۴] را بر اساس احتمالات آنها در دورهای بعدی بررسی می‌کنیم.

با این حساب، در دور اولیه، ما نمی‌دانیم کدام بازو بهترین بازو است. بنابراین به جای اختصاص احتمال بالا به بازو بر اساس میانگین پاداش، احتمال مساوی را به همه بازوها در دور اولیه با مقدار بالای T اختصاص داده و در یک سری دور، مقدار T را کاهش می‌دهیم و احتمال بالایی را به بازویی که میانگین پاداش بالایی دارد اختصاص می‌دهیم.

پیاده‌سازی روش کاوش بیشینه_نرم

اکنون، بیایید یاد بگیریم که چگونه روش اکتشاف یا کاوش بیشینه–نرم را برای یافتن بهترین بازو پیاده سازی کنیم. اول بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import gym_bandits
import numpy as np
```

بیایید همان ماشین بخت‌آزمایی با دو بازو را در نظر بگیریم که در بخش اپسیلون حریصانه دیدیم:

```
env = gym.make("BanditTwoArmedHighLowFixed-v0")
```

حالا، بیایید متغیرها را مقداردهی اولیه کنیم.

مقداردهی اولیه `count` برای ذخیره تعداد دفعاتی که بازو کشیده می‌شود:

```
count = np.zeros(2)
```

مقداردهی اولیه `sum_rewards` برای ذخیره مجموع جوایز هر بازو:

```
sum_rewards = np.zeros(2)
```

مقداردهی اولیه Q برای ذخیره میانگین پاداش هر بازو:

```
Q = np.zeros(2)
```

تعداد دورها (تکرارها) را تنظیم کنید:

```
num_rounds = 100
```

اکنون، تابع بیشینه–نرم را با دمای T تعریف می‌کنیم:

$$P_t(a) = \frac{\exp(Q_t(a)/T)}{\sum_{i=1}^{n} \exp(Q_t(i)/T)}$$

```
def softmax(T):
```

احتمال هر بازو را بر اساس معادله قبلی محاسبه کنید:

```
    denom = sum([np.exp(i/T) for i in Q])
    probs = [np.exp(i/T)/denom for i in Q]
```

بازو را بر اساس توزیع احتمال محاسبه شده بازوها انتخاب کنید:

```
    arm = np.random.choice(env.action_space.n, p=probs)

    return arm
```

حالا بیایید بازی را انجام دهیم و سعی کنیم با استفاده از روش کاوش بیشینه–نرم بهترین بازو را پیدا کنیم

بیایید با تنظیم دما T روی یک عدد بالا شروع کنیم، مثلا ۵۰:

```
T = 50
```

برای هر دور:

```
for i in range(num_rounds):
```

بازو را بر اساس روش اکتشاف بیشینه–نرم انتخاب کنید

```
    arm = softmax(T)
```

بازو را بکشید و پاداش و اطلاعات وضعیت بعدی را ذخیره کنید:

```
    next_state, reward, done, info = env.step(arm)
```

`count` بازو را ۱ واحد افزایش دهید:

```
    count[arm] += 1
```

مجموع جوایز بازو را به روز کنید:

```
    sum_rewards[arm]+=reward
```

میانگین پاداش بازو را به روز کنید:

```
    Q[arm] = sum_rewards[arm]/count[arm]
```

دما را کاهش دهید:

```
    T = T*0.99
```

بعد از تمام دورها ، مقدار Q را بررسی می‌کنیم ، یعنی میانگین پاداش همه بازوها:

```
print(Q)
```

کد قبلی چیزی شبیه به این را چاپ می‌کند:

```
[0.77700348 0.1971831 ]
```

همانطور که می‌بینیم ، بازوی ۱ میانگین پاداش بالاتری نسبت به بازوی ۲ دارد ، بنابراین بازوی ۱ را به عنوان بازوی بهینه انتخاب می‌کنیم:

```
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

کد قبلی، مقدار زیر را چاپ می‌کند:

```
The optimal arm is arm 1
```

بنابراین، بازوی بهینه را با استفاده از روش اکتشاف بیشینه–نرم پیدا کرده ایم.

استراتژی حد بالای اطمینان

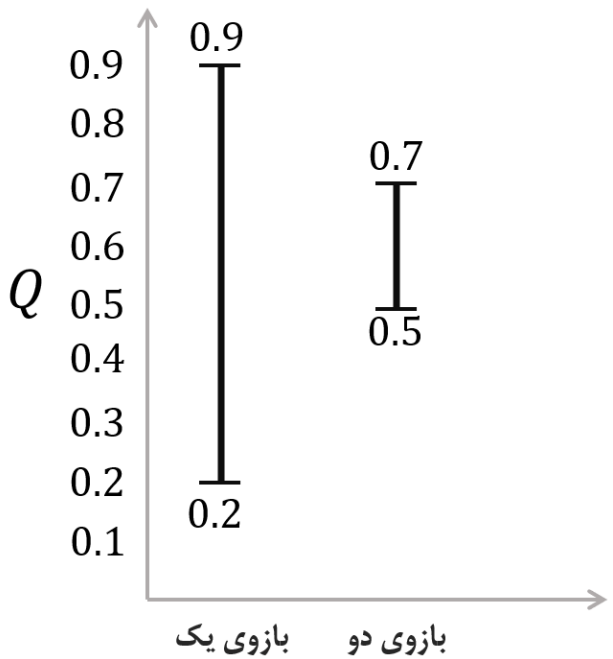
در این بخش، الگوریتم جالب دیگری به نام **حد بالای اطمینان** (**UCB**) را برای مدیریت معضل دوگانه اکتشاف–انتفاع (یا کاوش–یابش) معرفی خواهیم کرد. الگوریتم **UCB** بر اساس اصل: **خوش‌بینی در مواجهه با عدم قطعیت** است. بیایید یک مثال ساده بزنیم و بفهمیم الگوریتم **UCB** دقیقا چگونه کار می‌کند.

گیریم ما دو بازو داریم: بازوی ۱ و بازوی ۲. فرض کنید ما بازی را به مدت ۲۰ دور با کشیدن بازوی ۱ و بازوی ۲ به صورت تصادفی انجام داده و متوجه شدیم که میانگین پاداش بازوی یک معادل ۰.۶ و میانگین پاداش بازوی دو معادل ۰.۵ است. اما چگونه می‌توانیم مطمئن باشیم که این **پاداش متوسط** واقعا دقیق است؟ یعنی چگونه می‌توانیم مطمئن باشیم که این میانگین پاداش، نشان دهنده میانگین واقعی (میانگین جمعیت) است؟ اینجاست که از فاصله اطمینان استفاده می‌کنیم.

فاصله اطمینان نشان دهنده فاصله‌ای است که مقدار واقعی در آن قرار دارد. بنابراین، در محیط ما، فاصله اطمینان نشان دهنده فاصله‌ای است که میانگین پاداش واقعی بازو در آن قرار دارد.

به عنوان مثال، همانگونه که از روی شکل ۶.۲، می‌توانیم ببینیم فاصله اطمینان بازوی یک بین ۰.۲تا ۰.۹ است که نشان می‌دهد میانگین پاداش بازوی یک در محدوده ۰.۲ تا ۰.۹ قرار دارد. به مقدار ۰.۲ **حد پایین اطمینان** و به مقدار ۰.۹ **حد بالای اطمینان** می‌گویند. به طور مشابه، می‌توان مشاهده کرد که فاصله اطمینان بازوی دوم بین ۰.۵ تا ۰.۷ است که نشان می‌دهد میانگین پاداش بازوی ۲ در محدوده ۰.۵ تا ۰.۷ قرار دارد. جایی که ۰.۵ حد پایین اطمینان و ۰.۷ حد بالای اطمینان است.

خب، از شکل ۶.۲، می‌توانیم فواصل اطمینان بازوی ۱ و بازوی ۲ را ببینیم. حالا، چگونه می‌توانیم تصمیم بگیریم؟ یعنی چگونه می‌توانیم تصمیم بگیریم که بازوی ۱ را بکشیم یا بازوی ۲؟ اگر به دقت نگاه کنیم، می‌بینیم که فاصله اطمینان بازوی ۱ زیاد و فاصله اطمینان بازوی ۲ کوچک است.



شکل ۶.۲: فواصل اطمینان برای بازوهای ۱ و ۲

هنگامی‌که فاصله اطمینان زیاد است، در مورد مقدار میانگین مطمئن نیستیم. از آنجایی که فاصله اطمینان بازوی ۱

زیاد است (۰.۲ تا ۰.۹)، مطمئن نیستیم که با کشیدن بازوی ۱ چه پاداشی به دست می‌آوریم زیرا میانگین پاداش از ۰.۲ تا ۰.۹ متغیر است. بنابراین، عدم قطعیت زیادی در بازوی ۱ وجود دارد و ما مطمئن نیستیم که بازوی ۱ پاداش بالا می‌دهد یا پاداش.

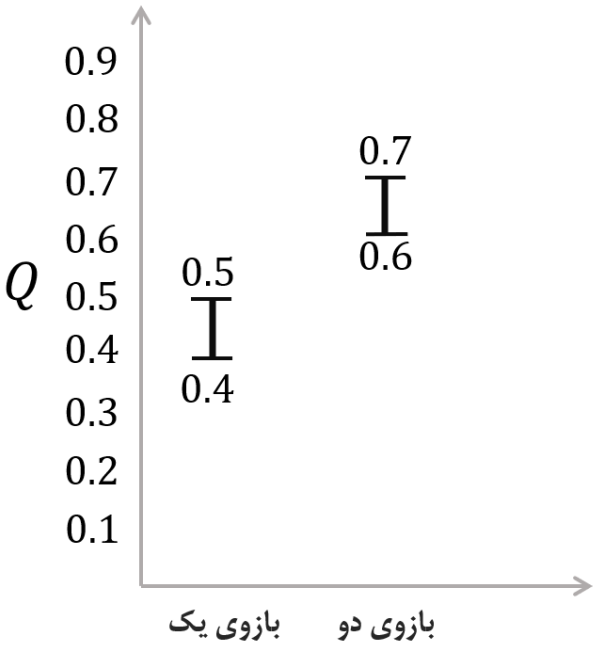
وقتی فاصله اطمینان کوچک باشد، در مورد مقدار میانگین مطمئن هستیم. از آنجایی که فاصله اطمینان بازوی ۲ کم است (۰.۵ تا ۰.۷)، می‌توانیم مطمئن باشیم که با کشیدن بازوی ۲ پاداش خوبی دریافت خواهیم کرد زیرا میانگین پاداش ما در محدوده ۰.۵ تا ۰.۷ است.

اما دلیل کوچک بودن فاصله اطمینان بازوی ۲ و بزرگ بودن فاصله اطمینان بازوی ۱ چیست؟ در ابتدای بخش متوجه شدیم که بازی را به مدت ۲۰ دور با کشیدن بازوی ۱ و بازوی ۲ به صورت تصادفی انجام دادیم و میانگین پاداش بازوی ۱ و بازوی ۲ را محاسبه کردیم. فرض کنید بازوی دوم ۱۵ بار کشیده شده و بازوی اول فقط ۵ بار کشیده شده است. از آنجایی که بازوی ۲ بارها کشیده شده است، فاصله اطمینان بازوی ۲ کوچک است و میانگین پاداش معینی را نشان می‌دهد. از آنجایی که بازوی ۱ کمتر کشیده شده است، فاصله اطمینان بازو زیاد است و نشاندهنده میانگین پاداش ناقطعی است. بنابراین، نشان می‌دهد که بازوی ۲ بسیار بیشتر از بازوی ۱ مورد کاوش قرار گرفته است.

بسیار خب، برگردیم به سؤالمان، آیا باید بازوی ۱ را بکشیم یا بازوی ۲؟ در **UCB**، ما همیشه بازویی را انتخاب می‌کنیم که حد بالای اطمینان بزرگتری دارد، بنابراین در مثال خود، بازوی ۱ را انتخاب می‌کنیم زیرا دارای حد بالای اطمینان آن ۰.۹ است. اما چرا ما باید بازویی را انتخاب کنیم که بزرگترین حد بالای اطمینان را داشته باشد؟ انتخاب بازوی با بیشترین حد بالا به ما کمک می‌کند تا بازویی را انتخاب کنیم که حداکثر پاداش را می‌دهد.

اما یک عیب یا مشکل پنهانی وجود دارد. وقتی فاصله اطمینان زیاد باشد، در مورد میانگین پاداش مطمئن نخواهیم بود. فرضا، در مثال ما، بازوی ۱ را انتخاب می‌کنیم زیرا دارای حد بالای اطمینان ۰.۹ است. ولی از آنجا که، محدوده یا فاصله اطمینان بازوی ۱ بزرگ است، میانگین پاداش ما می‌تواند از ۰.۲ تا ۰.۹ باشد، بنابراین حتی می‌توانیم پاداش کمی دریافت کنیم. اما اشکالی ندارد، ما هنوز بازوی ۱ را انتخاب می‌کنیم زیرا به هر حال موجب اکتشاف بیشتر می‌شود. هنگامی‌که بازو به خوبی کاوش شود، نتیجتا فاصله اطمینان کوچکتر می‌شود.

همانطور که ما بازی را برای چندین دور با انتخاب بازویی که دارای **UCB** بالا است، انجام می‌دهیم، فاصله اطمینان هر دو بازو کوچکتر یا کوتاهتر شده و مقدار میانگین دقیق‌تری را نشان می‌دهد. به عنوان مثال، همانطور که در شکل ۶.۳ می‌بینیم، پس از انجام چندین دور بازی، فاصله اطمینان هر دو بازو کوچک شده و مقدار میانگین دقیق‌تری را نشان می‌دهد:



شکل ۶.۳: فواصل اطمینان برای بازوهای ۱ و ۲ پس از چندین دور

در شکل ۶.۳ می‌بینیم که فاصله اطمینان هر دو بازو کوچک است و میانگین دقیق تری‌داریم. از آنجایی که در **UCB** بازویی را انتخاب می‌کنیم که بالاترین **UCB** را دارد، بازوی ۲ را به عنوان بهترین بازو انتخاب می‌کنیم.

بنابراین، در **UCB**، ما همیشه بازویی را انتخاب می‌کنیم که بیشترین حد بالای اطمینان را داشته باشد. در دورهای اولیه، ممکن است بهترین بازو را انتخاب نکنیم زیرا فاصله اطمینان بازوها در دور اولیه زیاد خواهد بود. اما در یک سری دورها، فاصله اطمینان کمتر می‌شود و ما بهترین بازو را انتخاب می‌کنیم.

گیریم $N(a)$ تعداد دفعاتی باشد که بازو a کشیده شده است و t تعداد کل دور، سپس حد بالای اطمینان بازوی A

را می‌توان به صورت زیر محاسبه کرد.

$$\mathrm{UCB}(a) = Q(a) + \sqrt{\frac{۲\log(t)}{N(a)}} \qquad (۳)$$

ما بازویی که بزرگترین حد بالای اطمینان را دارد به عنوان بهترین بازو انتخاب می‌کنیم:

$$a^* = \arg\max_a \mathrm{UCB}(a)$$

الگوریتم **UCB** به شرح زیر است:

۱. بازویی را انتخاب کنید که حد بالای اطمینان آن بزرگتر است
۲. بازو را بکشید و پاداش دریافت کنید
۳. میانگین پاداش و فاصله اطمینان بازو را به‌روز کنید
۴. مراحل ۱ تا ۳ را برای چندین دور تکرار کنید

پیاده سازی UCB

اکنون، بیایید یاد بگیریم که چگونه الگوریتم **UCB** را برای یافتن بهترین بازو پیاده سازی کنیم.

ابتدا، بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import gym_bandits
import numpy as np
```

بیایید همان ماشین بخت‌آزمایی دو بازویی را که در بخش قبلی دیدیم، ایجاد کنیم:

```
env = gym.make("BanditTwoArmedHighLowFixed-v0")
```

حالا، بیایید متغیرها را مقداردهی اولیه کنیم.

مقداردهی اولیه `count` برای ذخیره تعداد دفعاتی که بازو کشیده می‌شود:

```
count = np.zeros(2)
```

مقداردهی اولیه `sum_rewards` برای ذخیره مجموع پاداشی که هر بازو می‌گیرد:

```
sum_rewards = np.zeros(2)
```

مقداردهی اولیه Q برای ذخیره کردن میانگین پاداشی که هر بازو می‌گیرد:

```
Q = np.zeros(2)
```

تعداد دورها (تکرارها) را تنظیم کنید:

```
num_rounds = 100
```

اکنون، **تابع UCB** را تعریف می‌کنیم، که بهترین بازو را به عنوان بازویی با بالاترین **UCB** برمی‌گرداند:

```
def UCB(i):
```

آرایه `numpy` را برای ذخیره **UCB** همه بازوها راه اندازی کنید:

```
    ucb = np.zeros(2)
```

قبل از محاسبه **UCB**، حداقل یک بار همه بازوها را بررسی می‌کنیم، بنابراین برای ۲ دور اول، ما مستقیما بازوی مربوط به شماره دور را انتخاب می‌کنیم:

```
    if i < 2:
        return i
```

اگر دور بزرگتر از ۲ باشد ، **UCB** همه بازوها را همانطور که در معادله (۳) مشخص شده است محاسبه می‌کنیم و بازویی را که بالاترین **UCB** را دارد برمی‌گردانیم:

```
    else:
        for arm in range(2):
            ucb[arm] = Q[arm] + np.sqrt((2*np.log(sum(count))) /
count[arm])

        return (np.argmax(ucb))
```

حالا بیایید بازی را انجام دهیم و سعی کنیم با استفاده از روش **UCB** بهترین بازو را پیدا کنیم.

برای هر دور:

```
for i in range(num_rounds):
```

بازو را بر اساس روش **UCB** انتخاب کنید:

```
    arm = UCB(i)
```

بازو را بکشید و پاداش و اطلاعات وضعیت بعدی را ذخیره کنید:

```
    next_state, reward, done, info = env.step(arm)
```

مقدار `count` بازو را ۱ واحد افزایش دهید:

```
    count[arm] += 1
```

مجموع جوایز بازو را به روز کنید:

```
    sum_rewards[arm]+=reward
```

میانگین پاداش بازو را به روز کنید:

```
    Q[arm] = sum_rewards[arm]/count[arm]
```

پس از تمام دورها، می‌توانیم بازوی بهینه را به عنوان بازوی که حداکثر میانگین پاداش را دارد انتخاب کنیم:

```
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

با کد بالا مطلب پایین چاپ می‌شود:

```
The optimal arm is arm 1
```

بنابراین، بازوی بهینه را با استفاده از روش **UCB** پیدا کردیم.

استراتژی نمونه گیری تامپسون

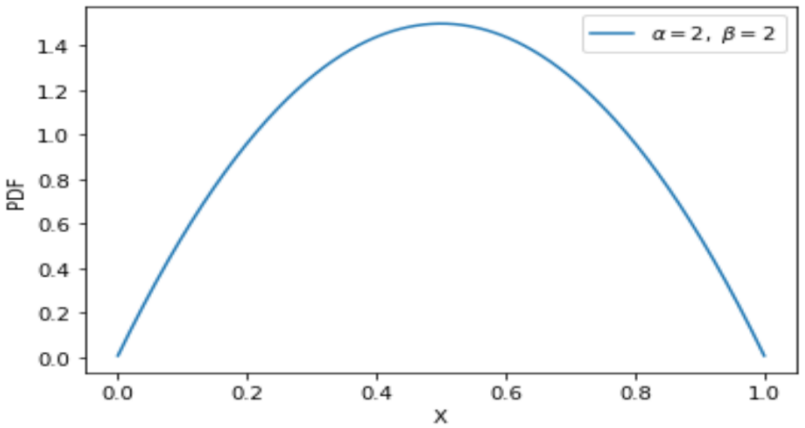
نمونه‌گیری تامپسون (TS) یکی دیگر از استراتژیهای اکتشافی جالب برای غلبه بر معضل یا دوگانه اکتشاف–انتفاع (یا گشتن–برداشتن) است و مبتنی بر توزیع بتا است. بنابراین، قبل از اینکه به نمونه برداری تامپسون بپردازیم، بیایید ابتدا توزیع بتا را درک کنیم. توزیع بتا یک تابع توزیع احتمال است و به صورت زیر بیان می‌شود:

$$f(x) = \frac{۱}{B(\alpha, \beta)} x^{\alpha - ۱}(۱ - x)^{\beta - ۱}$$

که در فرمول بالا داریم:

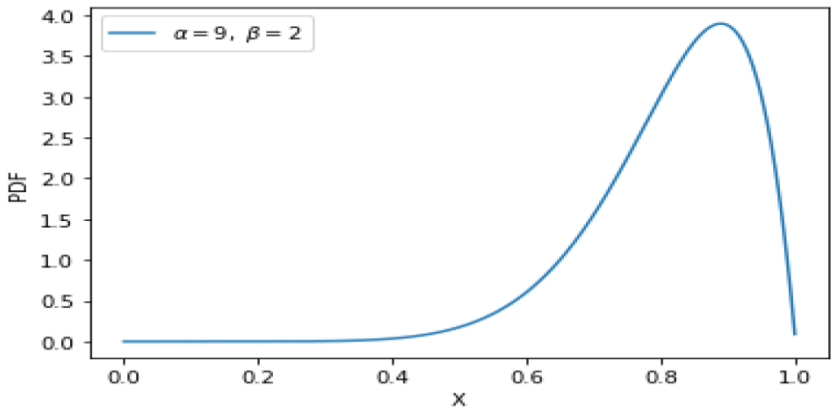
$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}$$

و $\Gamma(.)$ تابع گاما است. شکل توزیع، توسط دو پارامتر α و β کنترل می‌شود. وقتی مقادیر α و β یکسان باشند، یک توزیع متقارن خواهیم داشت. به عنوان مثال، همانطور که شکل ۶.۴ نشان می‌دهد، از آنجا که مقدار α و β با هم برابر یعنی ۲ است، ما یک توزیع متقارن داریم.



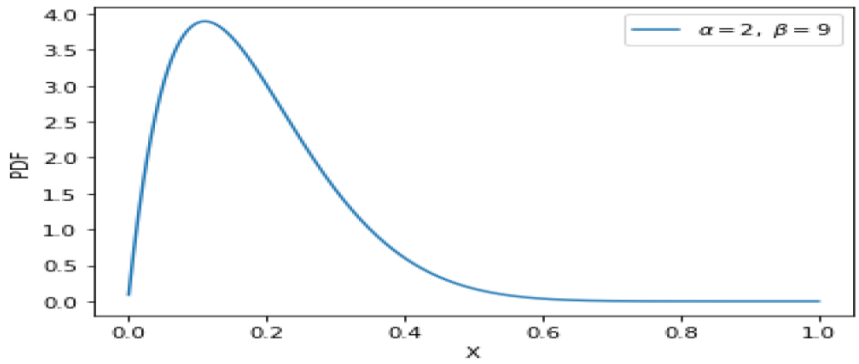
شکل ۶.۴: توزیع بتای متقارن

وقتی مقدار α از مقدار β بیشتر باشد، آنگاه ما احتمالات نزدیکتر به ۱ خواهیم داشت. به عنوان مثال، همانطور که شکل ۶.۵ نشان می‌دهد، از آنجا که مقدار $\alpha = ۹$ و $\beta = ۹$ است، احتمال مورد نظر ما به ۱ نزدیکتر است.



شکل ۶.۵: توزیع بتا که در آن $\alpha > \beta$

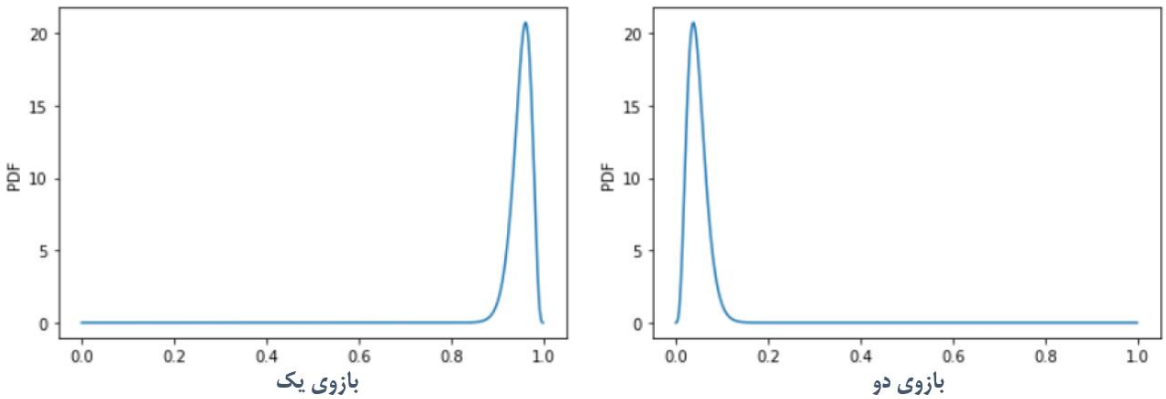
وقتی مقدار β بیشتر از α است باشد، آنگاه احتمالات بالا به مقدار ۰ نزدیکتر و از مقدار ۱ دورترند. به عنوان مثال، همانطور که در نمودار زیر نشان داده شده است، از آنجایی که مقدار $\alpha = ۹$ و $\beta = ۹$ است، احتمال بالایی نزدیک به ۰ داریم.



شکل ۶.۶: توزیع گاما که در آن $\alpha < \beta$

اکنون که یک ایده اولیه از توزیع بتا داریم، بیایید نحوه عملکرد نمونه‌گیری تامپسون و نحوه استفاده از توزیع بتا را بررسی کنیم. درک توزیع واقعی هر بازو بسیار مهم است زیرا هنگامی‌که توزیع واقعی بازو را بدانیم، به راحتی می‌توانیم بفهمیم که آیا بازو پاداش خوبی به ما می‌دهد یا خیر. یعنی می‌توانیم بفهمیم که آیا کشیدن بازو کمکی به موفقیت و برنده شده می‌کند یا خیر. به عنوان مثال، فرض کنید ما دو بازو داریم : بازوی ۱ و بازوی ۲.

شکل ۶.۷ توزیع واقعی دو بازو را نشان می‌دهد.



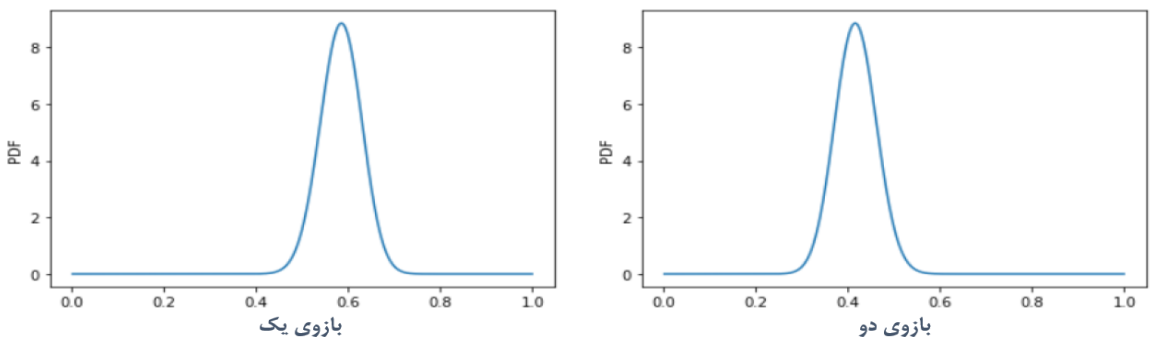
شکل ۶.۷: توزیع واقعی برای بازوهای ۱ و ۲

از شکل ۶.۷ می‌بینیم که کشیدن بازوی ۱ بهتر از کشیدن بازوی ۲ است زیرا بازوی ۱ احتمالات زیادی نزدیک به ۱ دارد، اما بازوی ۲ احتمال بالایی نزدیک به ۰ دارد. بنابراین، اگر بازوی ۱ را بکشیم، پاداش ۱ می‌گیریم و بازی را

می‌بریم، اما اگر بازوی ۲ را بکشیم، یک پاداش ۰ گرفته و بازی را از دست دادن می‌دهیم. بنابراین، هنگامی‌که توزیع واقعی اهرمها یا دسته‌ها را بدانیم سپس می‌توانیم بفهمیم کدام بازو بهترین بازو است.

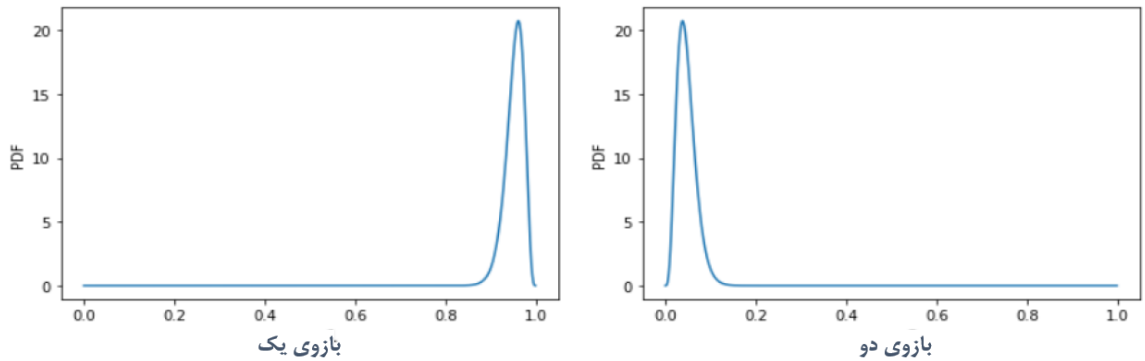
اما چگونه می‌توانیم توزیع واقعی بازوی ۱ و بازوی ۲ را بیاموزیم؟ اینجاست که از روش نمونه گیری تامپسون استفاده می‌کنیم. نمونه‌گیری تامپسون یک روش احتمالاتی است و بر اساس **توزیع پیشین** است.

ابتدا n نمونه از بازوی ۱ و بازوی ۲ گرفته و توزیع آنها را محاسبه می‌کنیم. با این حال، در تکرارهای اولیه، توزیعهای محاسبه شده بازوی ۱ و بازوی ۲ با توزیع واقعی یکسان نخواهند بود و بنابراین ما آن را **توزیع پیشین** می‌نامیم. همانطور که شکل ۶.۸ نشان می‌دهد، ما توزیع پیشین بازوی ۱ و بازوی ۲ را داریم که با توزیع واقعی آنها متفاوت است:



شکل ۶.۸: توزیع پیشین برای بازوهای ۱ و ۲

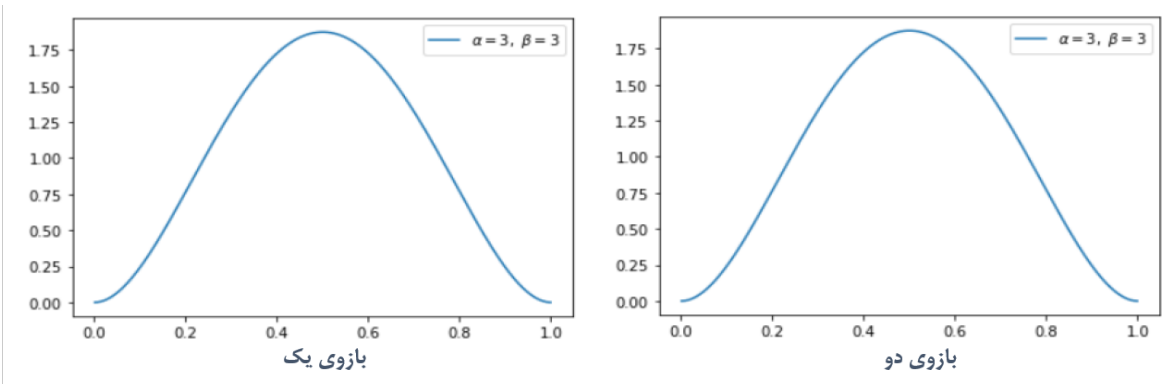
اما در طول یک سری تکرار، توزیع واقعی بازوی ۱ و بازوی ۲ را یاد می‌گیریم و همانطور که شکل ۶.۹ نشان می‌دهد، توزیعهای قبلی بازوها مانند توزیع واقعی پس از یک سری تکرارها به نظر می‌رسند:



شکل ۶.۹: توزیعهای پیشین به توزیعهای واقعی نزدیکتر می‌شوند.

هنگامی‌که توزیع واقعی همه بازوها را یاد گرفتیم، می‌توانیم به راحتی بهترین بازو را انتخاب کنیم. بسیار خب، اما دقیقا چگونه توزیع واقعی را یاد می‌گیریم؟ بیایید این را با جزئیات بیشتری بررسی کنیم.

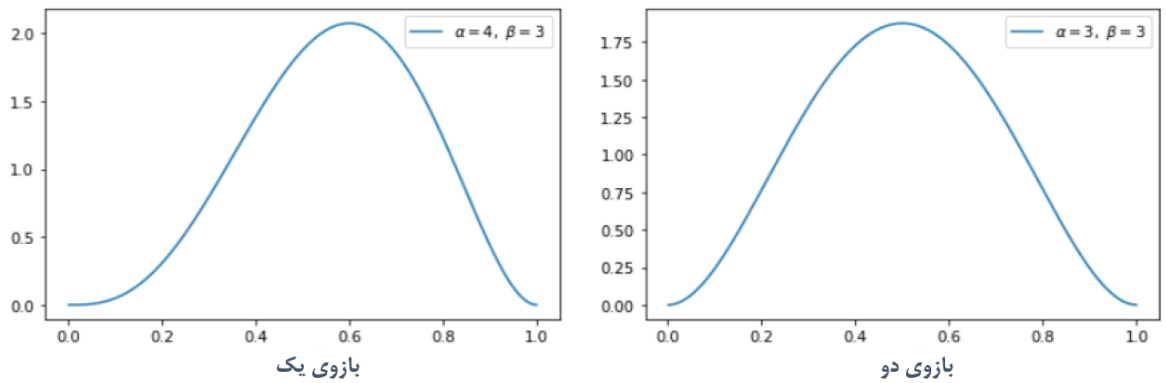
در اینجا، ما از توزیع بتا به عنوان توزیع پیشین استفاده می‌کنیم. فرضا ما دو اهرم یا بازو داریم، بنابراین ما دو توزیع بتا (توزیعهای پیشین) خواهیم داشت و هر دو پارامتر α و β را با اندازه مساوی، مثلا ۳، مقداردهی اولیه می‌کنیم، همانطور که شکل ۶.۱۰ نشان می‌دهد.



شکل ۶.۱۰: توزیعهای پیشین برای بازوهای ۱ و ۲ یکسان به نظر می‌رسند

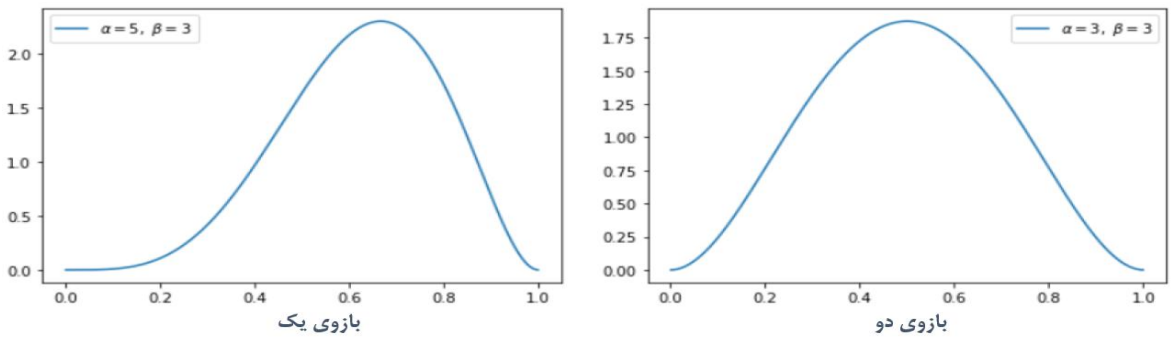
همانطور که می‌بینیم، از آنجایی که ما آلفا و بتا را به صورت برابر مقداردهی اولیه کردیم، توزیعهای بتا بازوی ۱ و بازوی ۲ یکسان به نظر می‌رسند.

در دور اول، ما فقط به طور تصادفی مقداری را از این دو توزیع نمونه‌برداری می‌کنیم و بازویی را انتخاب می‌کنیم که حداکثر مقدار نمونه‌برداری شده را داشته باشد. فرض کنید مقدار نمونه‌برداری شده بازوی ۱ بالاتر باشد، بنابراین در این مورد، بازوی ۱ را می‌کشیم. گیریم با کشیدن بازوی ۱، بازی را بردیم، پس حالا توزیع بازوی ۱ را با افزایش مقدار آلفای توزیع به میزان ۱ واحد به‌روز می‌کنیم؛ یعنی مقدار آلفا را به صورت $\alpha = \alpha + 1$ به‌روز می‌کنیم. همانطور که شکل ۶.۱۱ نشان می‌دهد، مقدار آلفای توزیع بازوی ۱ واحد افزایش می‌یابد و لذا، احتمالات بالای توزیع بتای بازوی ۱ در مقایسه با بازوی ۲، کمی نزدیکتر به ۱ شده است.



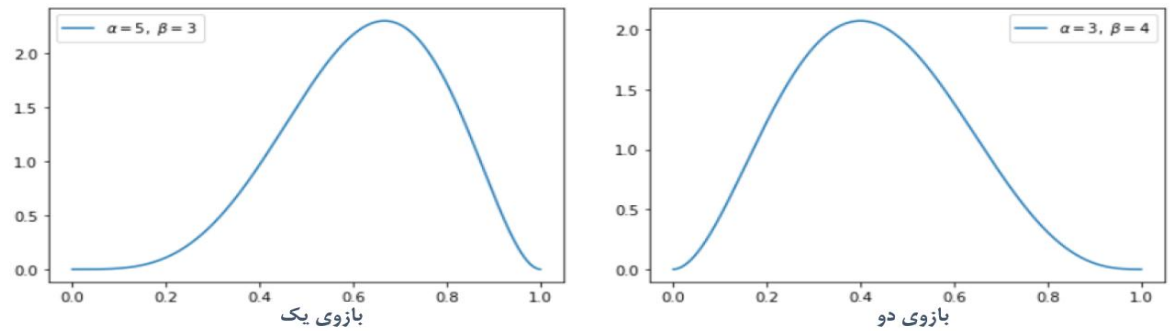
شکل ۶.۱۱: توزیع پیشین برای بازوهای ۱ و ۲ پس از دور ۱

در دور بعدی، دوباره یک مقدار را به صورت تصادفی از این دو توزیع نمونه‌برداری می‌کنیم و بازویی را انتخاب می‌کنیم که حداکثر مقدار نمونه‌برداری شده را داشته باشد. فرض کنید در این دور نیز حداکثر مقدار نمونه‌برداری شده را از بازوی ۱ دریافت کردیم. پس بازو ۱ را دوباره می‌کشیم. فرض کنید با کشیدن بازوی ۱ بازی را می‌بریم، سپس توزیع بازوی ۱ را با به روز رسانی مقدار آلفا به $\alpha = \alpha + 1$ به روز می‌کنیم. همانطور که شکل ۶.۱۲ نشان می‌دهد، مقدار آلفای توزیع بازوی ۱ افزایش یافته است و احتمالات بالا در توزیع بتای بازوی ۱ احتمال کمی نزدیکتر به ۱ شده است.



شکل ۶.۱۲: توزیع پیشین برای بازوهای ۱ و ۲ پس از دور ۲

به طور مشابه، در دور بعدی، دوباره به طور تصادفی مقداری را از این توزیعها نمونه‌برداری می‌کنیم و بازویی با حداکثر مقدار را می‌کشیم. فرض کنید این بار حداکثر مقدار را بازوی ۲ دارد، بنابراین بازوی ۲ را می‌کشیم و بازی را انجام می‌دهیم. فرض کنید با کشیدن بازوی ۲ بازی را از دست می‌دهیم. پس توزیع بازوی ۲ را با به‌روزرسانی مقدار بتا به صورت زیر به‌روز می‌کنیم $\beta = \beta + ۱$. همانطور که شکل ۶.۱۳ نشان می‌دهد، مقدار بتای بازوی توزیع ۲ افزایش می‌یابد و توزیع بتا بازوی ۲ احتمال کمی نزدیک به ۰ دارد:

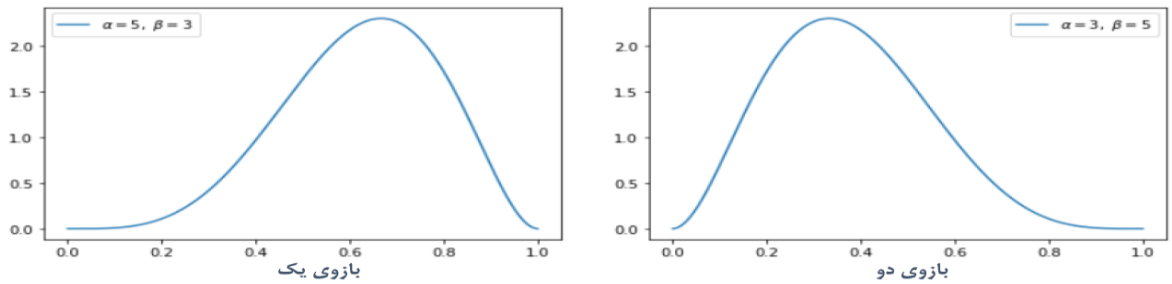


شکل ۶.۱۳: توزیع پیشین برای بازوهای ۱ و ۲ پس از دور ۳

دوباره در دور بعدی، ما به طور تصادفی، مقداری را از توزیع بتا بازوی ۱ و بازوی ۲ نمونه برداری می‌کنیم. فرض کنید مقدار نمونه از بازوی ۲ بیشتر است، بنابراین بازوی ۲ را می‌کشیم. و گیریم با کشیدن بازوی ۲ دوباره بازی را می‌بازیم. پس توزیع بازوی ۲ را با به‌روزرسانی مقدار بتا به صورت $\beta = \beta + ۱$ به روز می‌کنیم. همانطور که شکل ۶.۱۴ نشان می‌دهد، مقدار بتای توزیع بازوی ۲ یک واحد ۱ افزایش می‌یابد و همچنین توزیع بتا بازوی ۲ احتمال کمی

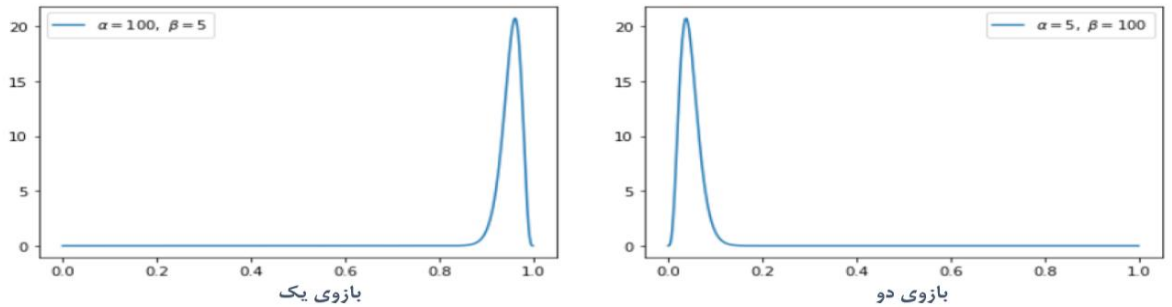
زیادتر نزدیک به ۰ دارد.

خب، آیا متوجه شدید که ما اینجا چه می‌کنیم؟ اگر با کشیدن هر بازو، بازی را ببریم، ما اساسا مقدار آلفای توزیع آن بازو را بیشتر می‌کنیم، در غیر این صورت مقدار بتای آنرا افزایش می‌دهیم. اگر این کار را به طور مکرر برای چندین دور انجام دهیم، می‌توانیم توزیع واقعی بازو را بیاموزیم.



شکل ۶.۱۴: توزیع پیشین برای بازوهای ۱ و ۲ پس از دور ۴

گیریم پس از چندین دور، توزیع ما مانند شکل ۶.۱۵ خواهد بود. همانطور که می‌بینیم، توزیع هر دو بازو شبیه توزیع واقعی است.



شکل ۶.۱۵: توزیع پیشین برای بازوهای ۱ و ۲ پس از چندین دور

حال اگر از هر یک از این توزیعها مقداری را نمونه برداری کنیم، مقدار نمونه‌برداری شده همیشه برای بازوی ۱ بالاتر خواهد بود و همیشه بازوی ۱ را می‌کشیم و بازی را می‌بریم.

مراحل مربوط به روش نمونه گیری تامسون در اینجا آورده شده است:

۱. توزیع بتا را با آلفا و بتا با مقادیر مساوی برای همه k دسته یا بازو راه اندازی کنید.
۲. یک مقدار از توزیع بتا تمام k بازوها نمونه‌برداری کنید.
۳. بازویی که مقدار نمونه‌برداری شده آن بالا است را بکشید.
٤. اگر بازی را بردیم، مقدار آلفای توزیع را به روز کنید $\alpha = \alpha + ۱$
٥. اگر بازی را باختیم، مقدار بتا توزیع را به روز کنید $\beta = \beta + ۱$
٦. مراحل ۲ تا ٥ را برای بسیاری از دورها تکرار کنید

پیاده‌سازی نمونه‌گیری تامپسون

اکنون، بیایید یاد بگیریم که چگونه روش نمونه‌گیری تامپسون را برای یافتن بهترین بازو پیاده‌سازی کنیم.

ابتدا، بیایید کتابخانه‌های لازم را وارد کنیم:

```
import gym
import gym_bandits
import numpy as np
```

برای درک بهتر، بیایید همان ماشین بخت‌آزمایی دو بازو را که در بخش قبلی دیدیم، ایجاد کنیم:

```
env = gym.make("BanditTwoArmedHighLowFixed-v0")
```

حالا، بیایید متغیرها را مقداردهی اولیه کنیم.

`count` اولیه برای ذخیره تعداد دفعاتی که بازو کشیده می‌شود:

```
count = np.zeros(2)
```

`sum_rewards` را برای ذخیره مجموع جوایز هر بازو مقداردهی اولیه کنید:

```
sum_rewards = np.zeros(2)
```

Q را برای ذخیره میانگین پاداش هر بازو مقداردهی اولیه کنید:

```
Q = np.zeros(2)
```

مقدار آلفا را به صورت ۱ برای هر دو بازو مقداردهی اولیه کنید:

```
alpha = np.ones(2)
```

مقدار بتا را به صورت ۱ برای هر دو بازو مقداردهی اولیه کنید:

```
beta = np.ones(2)
```

تعداد دورها (تکرارها) را تنظیم کنید:

```
num_rounds = 100
```

حال، بیایید تابع `thompson_sampling` را تعریف کنیم.

کار کد زیر این است که به طور تصادفی مقادیر را از توزیعهای بتای هر دو بازو نمونه‌برداری کرده و سپس بازوی دارای حداکثر مقدار نمونه‌برداری شده را به ما بدهد:

```
def thompson_sampling(alpha,beta):
    samples = [np.random.beta(alpha[i]+1,beta[i]+1) for i in range(2)]
    return np.argmax(samples)
```

حالا بیایید بازی را انجام دهیم و سعی کنیم با استفاده از روش نمونه‌گیری تامپسون بهترین بازو را پیدا کنیم.

برای هر دور:

```
for i in range(num_rounds):
```

بازو را بر اساس روش نمونه‌گیری تامپسون انتخاب کنید:

```
    arm = thompson_sampling(alpha,beta)
```

بازو را بکشید و پاداش و اطلاعات وضعیت بعدی را ذخیره کنید:

```
    next_state, reward, done, info = env.step(arm)
```

مقدار count بازو را ۱ واحد افزایش دهید:

```
    count[arm] += 1
```

مجموع جوایز بازو را به روز کنید:

```
    sum_rewards[arm]+=reward
```

میانگین پاداش بازو را به روز کنید:

```
    Q[arm] = sum_rewards[arm]/count[arm]
```

اگر در بازی برنده شویم ، یعنی اگر پاداش برابر با ۱ باشد ، مقدار آلفا را به $\alpha = \alpha + ۱$ به‌روز می‌کنیم، در غیر این صورت مقدار بتا را به $\beta = \beta + ۱$ به‌روز می‌کنیم:

```
    if reward==1:
        alpha[arm] = alpha[arm] + 1
    else:
        beta[arm] = beta[arm] + 1
```

پس از تمام دورها، می‌توانیم بازوی بهینه را به عنوان بازوی که بالاترین میانگین پاداش را دارد انتخاب کنیم:

```
print('The optimal arm is arm {}'.format(np.argmax(Q)+1))
```

کد قبلی چاپ می‌کند:

```
The optimal arm is arm 1
```

بنابراین، بازوی بهینه را با استفاده از روش نمونه گیری تامپسون یافتیم.

کاربردهای MAB

تا کنون، ما مسئله **MAB** و چگونگی حل آن با استفاده از استراتژیهای مختلف اکتشاف یا کاوش را یاد گرفتیم. اما هدف ما فقط استفاده از این الگوریتمها برای بازی ماشینهای اسلات نیست. ما می‌توانیم استراتژیهای مختلف اکتشاف را در چندین مورد استفاده مختلف اعمال کنیم.

به عنوان مثال، ماشینهای بخت‌آزمایی می‌توانند به عنوان جایگزینی برای آزمایش **AB**[۱] استفاده شوند. آزمون **AB** یکی از رایجترین روشهای کلاسیک آزمایش است. فرض کنید ما دو نسخه از پیاده‌سازی وب‌سایت خود داریم. و می‌خواهیم بدانیم کدام نسخه از صفحه فرود[۲] بیشتر مورد پسند کاربران است. در این مورد، ما تست **AB** را انجام می‌دهیم تا بفهمیم کدام نسخه از صفحه فرود بیشتر مورد علاقه کاربران است. بنابراین، ما نسخه ۱ صفحه فرود خود را به مجموعه خاصی از کاربران نشان داده و نسخه ۲ صفحه فرود را به مجموعه دیگری از کاربران نمایش می‌دهیم. سپس چندین معیار مانند نرخ کلیک، میانگین زمان صرف شده در وب‌سایت و غیره را اندازه‌گیری می‌کنیم تا بفهمیم کدام نسخه از صفحه فرود بیشتر مورد پسند کاربران است. هنگامی‌که متوجه شدیم کدام نسخه از صفحه فرود بیشتر مورد علاقه کاربران است، شروع به نمایش آن نسخه به همه کاربران خواهیم کرد.

لذا، در آزمایش **AB**، زمان جداگانه‌ای را برای هر یک از فعالیتهای اکتشاف و انتفاع برنامه‌ریزی می‌کنیم. یعنی آزمایش **AB** دارای دو دوره اختصاصی مختلف برای فعالیتهای اکتشاف و انتفاع است. اما مشکل آزمایش **AB** این است که پشیمانی زیادی را به همراه خواهد داشت. ما می‌توانیم با استفاده از استراتژیهای اکتشافی مختلفی که برای حل مسئله **MAB** یادگرفتیم، پشیمانی را به حداقل برسانیم. بنابراین، به جای انجام کاوش و یابش به طور کامل و جداگانه، می‌توانیم اکتشاف و انتفاع را بطور همزمان و به همان سبک تطبیقی که در بخش های قبلی آموختیم، انجام دهیم.

---

[۱] **آزمون آ/ب** به انگلیسی A/B testing نام فرایندی است که برای تشخیص اینکه از میان دو ویژگی «آ» و «ب» کدام‌یک مناسب‌تر است به کار می‌رود. در آزمون‌های آ/ب، دو پیاده‌سازی متفاوت به صورت آزمایشی به دو گروه از کاربران ارائه می‌شود. مقایسه نتایج به‌دست‌آمده از گروه‌ها می‌تواند به انتخاب پیاده‌سازی مناسب‌تر کمک کند. به عبارت یگر تست A/B اصطلاحی در بازاریابی و کسب‌وکار هوشمند است که برای مقایسه یک آزمایش در ۲ حالت مختلف A و B به کار می‌رود. این تست که با نام‌های bucket tests و split-run testing نیز شناخته می‌شود یک روش بسیار عالی برای پیدا کردن بهترین استراتژی بازاریابی برای کسب‌وکار شما است.

[۲] Landing Page

مدل ماشینهای بخت‌آزمایی به طور گسترده‌ای برای بهینه‌سازی وب‌سایت، به حداکثر رساندن نرخ تبدیل، تبلیغات آنلاین، کمپین و غیره استفاده می‌شوند.

یافتن بهترین بنر تبلیغاتی با استفاده از مدل بخت‌آزمایی

در این بخش بیایید ببینیم چگونه بهترین بنر تبلیغاتی را با استفاده از مدل بخت‌آزمایی پیدا کنیم. فرض کنید ما در حال پیاده‌سازی یک وب‌سایت هستیم و پنج بنر مختلف برای یک تبلیغ در وب‌سایت خود داریم و می‌خواهیم بفهمیم کدام بنر تبلیغاتی بیشتر مورد پسند کاربران است.

ما می‌توانیم این مشکل را به عنوان یک مسئله **MAB** قالب‌بندی کنیم. پنج بنر تبلیغاتی نشان دهنده پنج بازوی ماشین بخت‌آزمایی است و اگر کاربر روی تبلیغات کلیک کند، پاداشی معادل ۱+ و در صورت عدم کلیک کاربر بر روی آگهی، پاداشی برابر ۰ اختصاص می‌دهیم. بنابراین، برای اینکه بفهمیم کدام بنر تبلیغاتی بیشتر توسط کاربران کلیک می‌شود، یعنی کدام بنر تبلیغاتی می‌تواند حداکثر پاداش را به ما بدهد، می‌توانیم از استراتژیهای مختلف اکتشاف استفاده کنیم. در این بخش، بیایید فقط از یک روش حریصانه اپسیلون برای یافتن بهترین بنر تبلیغاتی استفاده کنیم.

ابتدا، بیایید کتابخانه های لازم را وارد کنیم:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
plt.style.use('ggplot')
```

ایجاد یک مجموعه داده

اکنون، بیایید یک مجموعه داده ایجاد کنیم. ما یک مجموعه داده با پنج ستون تولید می‌کنیم که پنج بنر تبلیغاتی را نشان می‌دهد و ۱۰۰,۰۰۰ ردیف تولید می‌کنیم که در آن مقادیر موجود در ردیف ها ۰ یا ۱ خواهد بود و نشان می‌دهد

آیا بنر تبلیغاتی توسط کاربر کلیک شده است (۱) یا خیر یعنی کلیک نشده (۰):

```
df = pd.DataFrame()
for i in range(5):
    df['Banner_type_'+str(i)] = np.random.randint(0,2,100000)
```

بیایید به چند ردیف اول مجموعه داده خود نگاه کنیم:

```
df.head()
```

کد قبلی موارد زیر را چاپ می‌کند. همانطور که می‌بینیم، ما پنج بنر تبلیغاتی (۰ تا ۴) و ردیفهایی داریم که از مقادیر ۰ یا ۱ تشکیل شده اند که نشان می‌دهد آیا روی بنر کلیک شده است (۱) یا اینکه روی آن کلیک نشده است (۰).

	Banner_type_0	Banner_type_1	Banner_type_2	Banner_type_3	Banner_type_4
۰	۰	۰	۱	۱	۱
۱	۱	۱	۰	۱	۱
۲	۲	۰	۱	۱	۱
۳	۳	۱	۱	۱	۱
٤	٤	۰	۰	۱	۰

شکل ۶.۱۴: کلیک در هر بنر

متغیرها را مقداردهی اولیه کنید

حال، بیایید برخی از متغیرهای مهم را مقداردهی اولیه کنیم. تعداد تکرارها را تنظیم کنید:

```
num_iterations = 100000
```

تعداد بنرها را تعریف کنید:

```
num_banner = 5
```

پارامتر `count` برای ذخیره تعداد **دفعات کلیک** بر روی **بنر** را مقدار دهی اولیه کنید:

```
count = np.zeros(num_banner)
```

پارامتر sum_rewards را برای ذخیره مجموع **جوایز (پاداشها)** به دست آمده از هر **بنر**، مقداردهی اولیه کنید:

```
sum_rewards = np.zeros(num_banner)
```

پارامتر Q را برای ذخیره میانگین **پاداش** هر **بنر**، مقداردهی اولیه کنید:

```
Q = np.zeros(num_banner)
```

یک لیست برای ذخیره بنرهای انتخاب شده تعریف کنید:

```
banner_selected = []
```

تعریف روش اپسیلون حریصانه

حال، بیایید روش اپسیلون–حریصانه را تعریف کنیم. ما یک مقدار تصادفی از یک توزیع یکنواخت تولید می‌کنیم. اگر مقدار تصادفی کمتر از اپسیلون باشد، بنر تصادفی را انتخاب می‌کنیم. در غیر این صورت، بهترین بنر را انتخاب می‌کنیم که حداکثر میانگین پاداش را داشته باشد:

```
def epsilon_greedy_policy(epsilon):

    if np.random.uniform(0,1) < epsilon:
        return np.random.choice(num_banner)
    else:
        return np.argmax(Q)
```

تست بخت‌آزمایی را اجرا کنید

اکنون، ما سیاست اپسیلون حریصانه را اجرا می‌کنیم تا بفهمیم کدام بنر تبلیغاتی بهتر از بقیه است.

برای هر تکرار:

```
for i in range(num_iterations):
```

بنر را با استفاده از سیاست اپسیلون حریصانه انتخاب کنید:

```
    banner = epsilon_greedy_policy(0.5)
```

پاداش بنر را دریافت کنید:

```
    reward = df.values[i, banner]
```

شمارنده را افزایش دهید:

```
    count[banner] += 1
```

مجموع جوایز (پاداشها) را ذخیره کنید:

```
    sum_rewards[banner]+=reward
```

میانگین پاداش را محاسبه کنید:

```
    Q[banner] = sum_rewards[banner]/count[banner]
```

بنر را در لیست انتخاب شده بنر ذخیره کنید:

```
    banner_selected.append(banner)
```

پس از تمام دورها، می‌توانیم بهترین بنر را به عنوان بنری که دارای حداکثر میانگین پاداش است انتخاب کنیم:

```
print( 'The best banner is banner {}'.format(np.argmax(Q)))
```

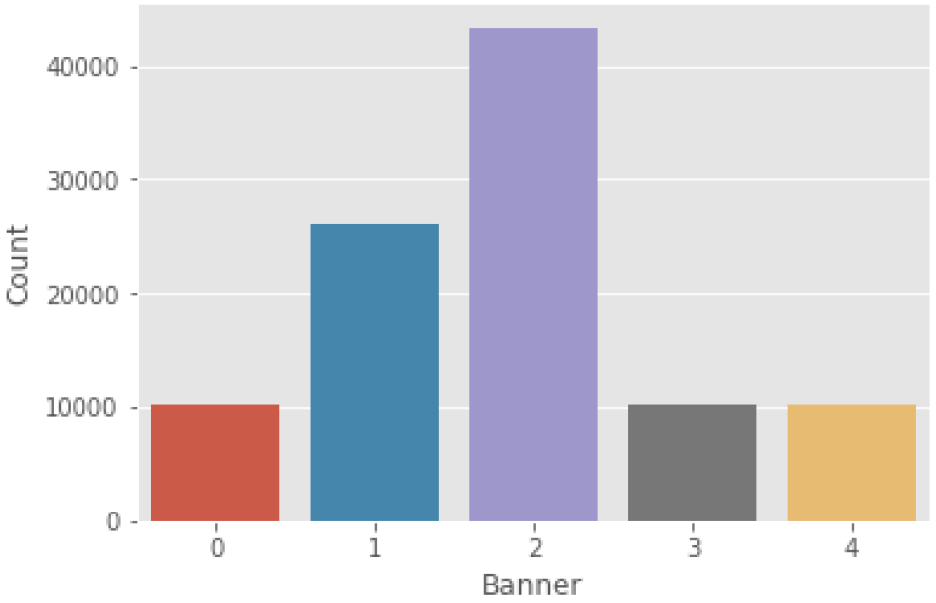
با کد قبلی، چاپ می‌شود:

```
The best banner is banner 2
```

همچنین می‌توانیم ترسیم کنیم و ببینیم کدام بنر بیشتر انتخاب می‌شود:

```
ax = sns.countplot(banner_selected)
ax.set(xlabel='Banner', ylabel='Count')
plt.show()
```

کد قبلی موارد زیر را ترسیم می‌کند. همانطور که می‌بینیم، بنر ۲ اغلب اوقات انتخاب شده است.



شکل ۶.۱۵: بنر ۲ بهترین بنر تبلیغاتی است.

بنابراین، ما یاد گرفته ایم که چگونه بهترین بنر تبلیغاتی را با قالب‌بندی مشکل خود بعنوان یک مسئله **MAB** پیدا کنیم.

روش بخت‌آزمایی بافتارک (زمینه‌ای)

ما به تازگی یاد گرفتیم که چگونه از روش بخت‌آزمایی برای یافتن بهترین بنر تبلیغاتی مورد علاقه کاربران استفاده کنیم. اما ترجیح بنر از کاربری به کاربر دیگر متفاوت است. یعنی کاربر **A** بنر ۱ را دوست دارد، اما کاربر **B** ممکن است بنر ۳ را دوست داشته باشد و غیره. هر کاربر ترجیحات خاص خود را دارد. بنابراین، ما باید بنرهای تبلیغاتی را با توجه به هر کاربر شخصی سازی کنیم. چگونه می‌توانیم این کار را انجام دهیم؟ اینجاست که ما از بخت‌آزمایی بافتار

(زمینه‌ای) استفاده می‌کنیم.

در مسئله **MAB**، ما فقط **عمل** را انجام می‌دهیم و **پاداش** دریافت می‌کنیم. اما با بخت‌آزمایی زمینه‌ای، ما بر اساس حالت محیط، اقداماتی انجام می‌دهیم و آن حالت دربرگیرنده زمینه است. به عنوان مثال، در مثال بنر تبلیغاتی، حالت، رفتار کاربر را مشخص می‌کند و ما با توجه به حالت (رفتار کاربر) اقدام می‌کنیم (بنر را نشان می‌دهیم) که منجر به حداکثر پاداش (کلیکهای تبلیغاتی) می‌شود.

بخت‌آزمایی بافتار به طور گسترده ای برای شخصی‌سازی محتوا با توجه به رفتار کاربر استفاده می‌شوند. آنها همچنین برای حل مشکلات شروع سرد که سیستمهای توصیه با آن مواجه هستند استفاده می‌شوند. نتفلیکس از بخت‌آزمایی بافتار برای شخصی‌سازی آثار هنری در برنامه‌های تلویزیونی با توجه به رفتار کاربر استفاده می‌کند.

خلاصه

ما فصل را با درک اینکه مسئله **MAB** چیست و چگونه می‌توان آنرا با استفاده از چندین استراتژی اکتشاف یا کاوش حل کرد، شروع کردیم. ما ابتدا در مورد روش اپسیلون- حریصانه یاد گرفتیم، جایی که یک بازوی دلبخواه (تصادفی) را با احتمال اپسیلون و بهترین بازو را با احتمال یک منهای اپسیلون انتخاب می‌کنیم. در مرحله بعد، با روش اکتشاف بیشینه-نرم آشنا شدیم، جایی که هر **بازو** را بر اساس توزیع احتمال آن انتخاب کرده و احتمال انتخاب هر بازو متناسب با میانگین **پاداش** آن است.

پس از آن، با الگوریتم **UCB** آشنا شدیم، جایی که **بازویی** را انتخاب می‌کنیم که دارای بیشترین **مقدار** در محدوده اطمینان را دارد. سپس، روش نمونه گیری تامسپون را بررسی کردیم که در آن توزیع بازوها بر اساس توزیع بتا را یاد گرفتیم.

در ادامه، یاد گرفتیم که چگونه می‌توان از **MAB** به عنوان جایگزینی برای تست **AB** استفاده کرده و چگونه می‌توانیم با چارچوب‌بندی مشکل به عنوان یک مسئله **MAB**، بهترین بنر تبلیغاتی را پیدا کنیم. در پایان فصل، ما همچنین

مروری بر روش بخت‌آزمایی بافتاری داشتیم.

در فصل بعدی، با چندین **الگوریتم یادگیری عمیق** جالب آشنا خواهیم شد که برای یادگیری تقویتی عمیق ضروری هستند.

سوالات

بیایید دانشی را که در این فصل به دست آورده ایم با پاسخ به سوالات زیر ارزیابی کنیم:

۱. مسئله **MAB** چیست؟
۲. سیاست اپسیلون حریصانه چگونه یک بازو را انتخاب می‌کند؟
۳. اهمیت پارامتر T در اکتشاف بیشینه–نرم (softmax) چیست؟
٤. چگونه حد اطمینان بالا (**UCB**) را محاسبه کنیم؟
٥. وقتی مقدار آلفا بالاتر از مقدار بتا در توزیع بتاست چه اتفاقی می‌افتد؟
٦. مراحل نمونه گیری تامپسون چیست؟
۷. مسئله بخت‌آزمایی بافتاری یا زمینه‌ای چیست؟

برای مطالعه بیشتر

برای اطلاعات بیشتر، این منابع جالب را بررسی کنید:

- **Introduction to Multi-Armed Bandits** by *Aleksandrs Slivkins*, https://arxiv. org/pdf/1904.07272.pdf

- **A Survey on Practical Applications of Multi-Armed and Contextual Bandits** by *Djallel Bouneffouf, Irina Rish,* https://arxiv.org/pdf/1904.10040.pdf

- **Collaborative Filtering Bandits** by *Shuai Li, Alexandros Karatzoglou, Claudio Gentile*, https://arxiv.org/pdf/1502.03473.pdf

پیوست فصل ۶

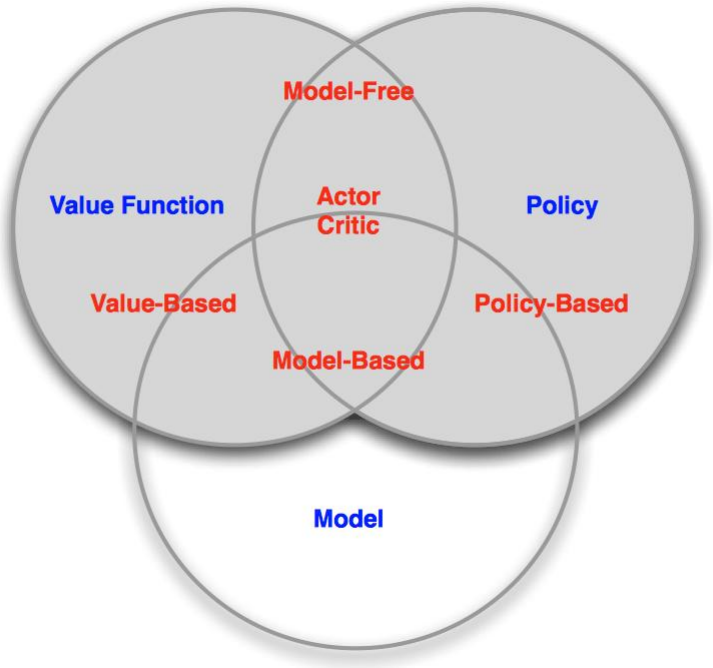
خلاصه تصویری

Categorizing RL agents[۱]

RL agents could be categorized into several categories:

- *Value Based*
  - No Policy (Implicit)
  - Value Function
- Policy Based
  - Policy
  - No Value Function
- Actor Critic
  - Policy
  - Value Function
- Model Free
  - Policy and/or Value Function
  - No Model
- Model Based
  - Policy and/or Value Function
  - Model

[۱] https://www.٥٢coding.com.cn/٢٠١٧/١٢/١٦/RL%٢٠-%٢٠Model-Free%٢٠Prediction/



Problems within Reinforcement Learning

This section only proposes questions without providing the solutions.

Learning and Planning

Two fundamental problems in sequential decision making:

- Reinforcement Learning
  - The environment is initially unknown
  - The agent interacts with the environment
  - The agent improves its policy
- Planning:
  - A model of the environment is known
  - The agent performs computations with its model (without any external interaction)
  - The agent improves its policy a.k.a. deliberation, reasoning, introspection, pondering, thought, search

Introduction

Part one:

- Model-free prediction
- **Estimate** the value function of an *unknown* MDP

Part two:

- Model-free control
- **Optimize** the value function of an unknown MDP

Why we care about model-free control? So, let's see some example problems that can be modelled as MDPs:

- Helicopter, Robocup Soccer, Quake
- Portfolio management, Game of Go...

For most of these problems, either:

- MDP model is **unknown**, but experience can be sampled
- MDP model is known, but is **too big to use**, except by samples

Model-free control: Model-free control can solve these problems.

There are two branches of model-free control:

- **On-policy:** On-policy learning
  - "Learn on the job"
  - Learn about policy π from experience sampled from π
- **Off-policy:** Off-policy learning
  - "Look over someone's shoulder"
  - Learn about policy π from experience sampled from μ

Table of Contents

Off-Policy Learning

Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$ while following behavior policy $\mu(a|s)$

$$\{S_1, A_1, R_2, ..., S_T\} \sim \mu$$

So, why is this important? There are several reasons:

- Learn from observing human or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, ..., \pi_{t-1}$
- Learn about **optimal** policy while following exploratory policy
- Learn about **multiple** policies while following

Batch MC and TD

We know that MC and TD converge: $V(s) \to v_\pi(s)$ as experience $\to \infty$. But what about batch solution for finite experience? If we **repeatly** train some *finite* sample episodes with MC and TD respectively, do the two algorithms give **same** result?

AB Example

To get more intuition, let's see the *AB* example.

There are two states in a MDP, A, B with no discounting. And we have 8 episodes of experience:

- A, 0, B, 0
- B, 1
- B, 1
- B, 1
- B, 1
- B, 1
- B, 1
- B, 0

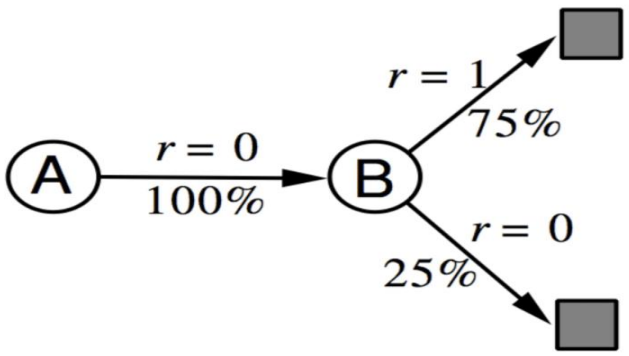
For example, the first episode means we in state A and get 0 reward, then transit to state B getting 0 reward, and then terminate.

So, What is $V(A), V(B)$?

First, let's consider $V(B)$. B state shows 8 times and 6 of them get reward 1, 2 of them get reward 0. So $V(B) = \frac{6}{8} = 0.75$ according to TD and MC.

However, if we consider $V(A)$, MC method will give $V(A) = 0$, since A just shows in one episode and the reward of that episode is 0. TD method will give $V(A) = 0 + V(B) = 0.75$.

The MDP of these experiences can be illustrated as



Certainty Equivalence

As we show above,

- **MC** converges to solution with **minimum mean-squared error**
  - Best fit to the **observed returns**

$$\sum_{k=1}^{K}\sum_{t=1}^{T_k}(G_t^k - V(s_t^k))^2$$

  - In the AB example, $V(A) = 0$
- **TD(0)** converges to solution of **max likelihood Markov model**
  - Solution to the **MDP** $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ **that best fits the data**

$$\hat{\mathcal{P}}_{s,s'}^a = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\mathbf{1}(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

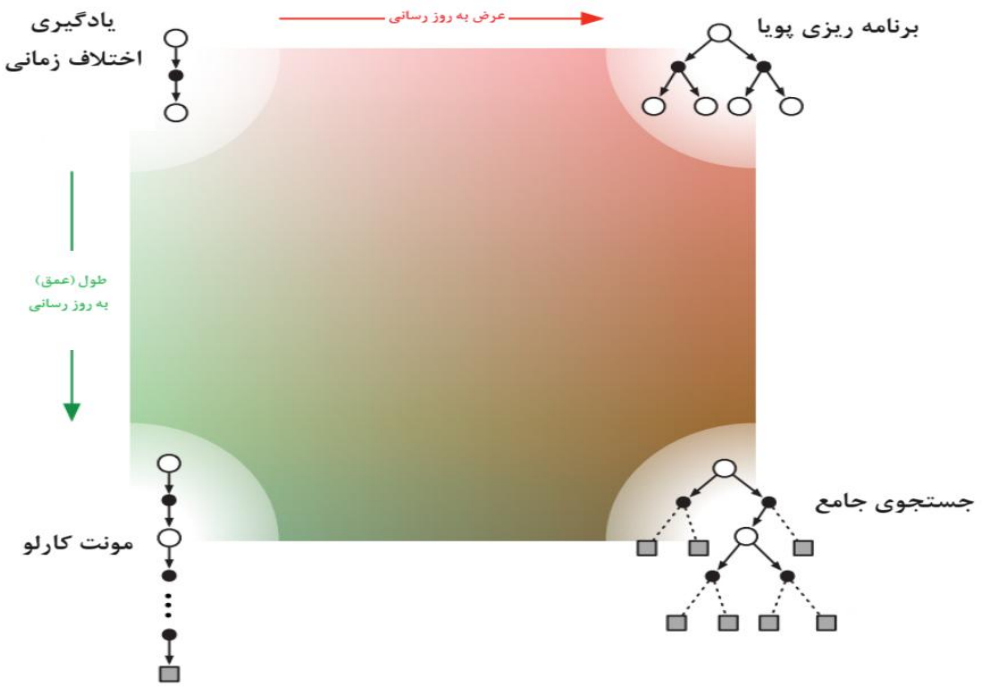
$$\hat{\mathcal{R}}_s^a = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\mathbf{1}(s_t^k, a_t^k = s, a)r_t^k$$

(First, count the transitions. Then compute rewards.)

  - In the AB example, $V(A) = 0.75$

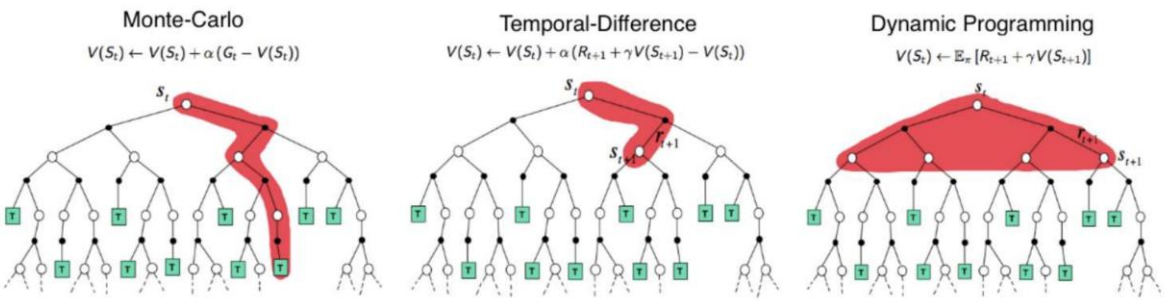
Advantages and Disadvantages of MC vs. TD (۲)

- TD exploits **Markov property**
  - Usually more efficient in Markov environments
- MC does **not** exploit Markov property
  - Usually more effective in non-Markov environments

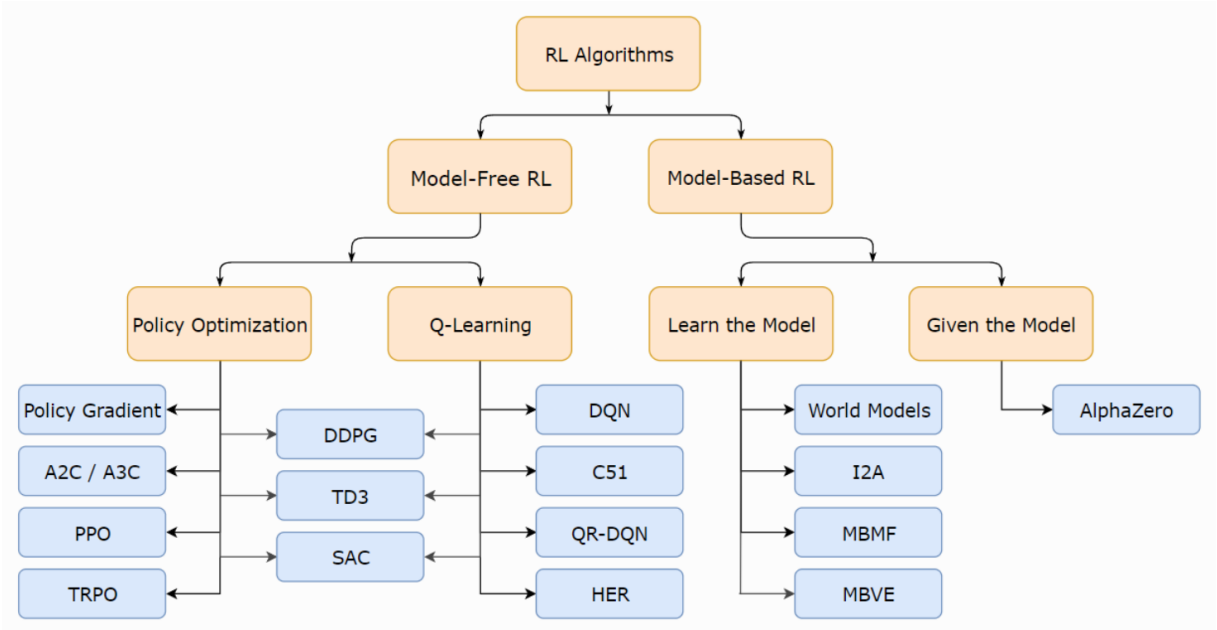


برشی از فضای روشهای یادگیری تقویتی، که دو بعد از مهمترین ابعاد بررسی شده را برجسته می‌کند: **عمق و عرض به روزرسانی‌ها.**

روشهای محاسبه ارزش



شکل زیر یک طبقه‌بندی کلی، هرچند نه همه جانبه، از طرحهای الگوریتم یادگیری تقویتی عمیق را ارائه می‌دهد.

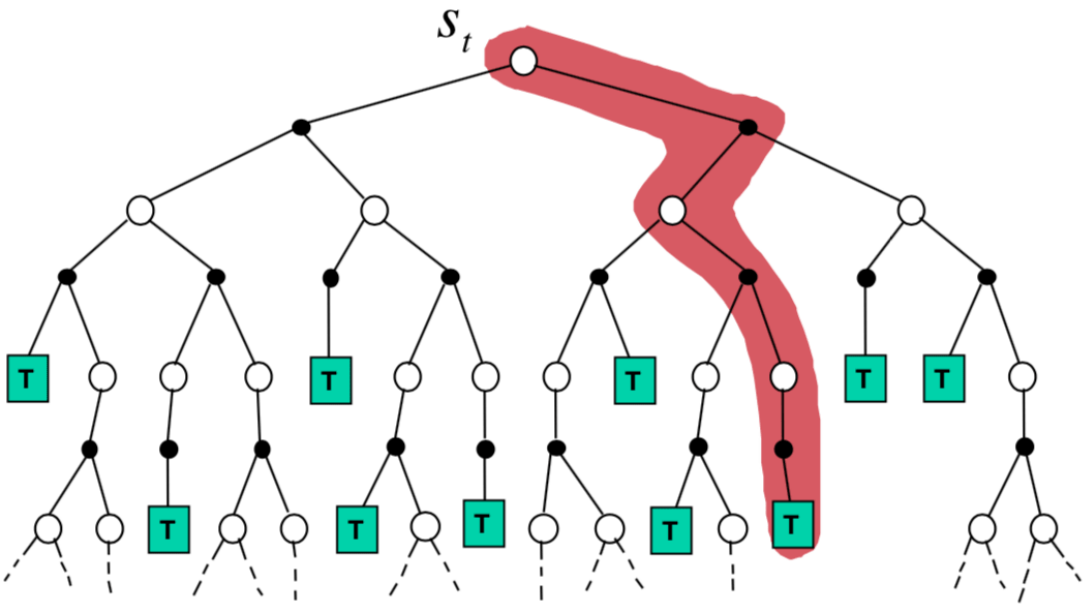


توجه داشته باشید که الگوریتم‌های Monte-Carlo و Temporal-Difference هر دو نمونه بارزی از رویکردهای بدون مدل هستند.

Unified View[۱]

Monte-Carlo Backup

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$

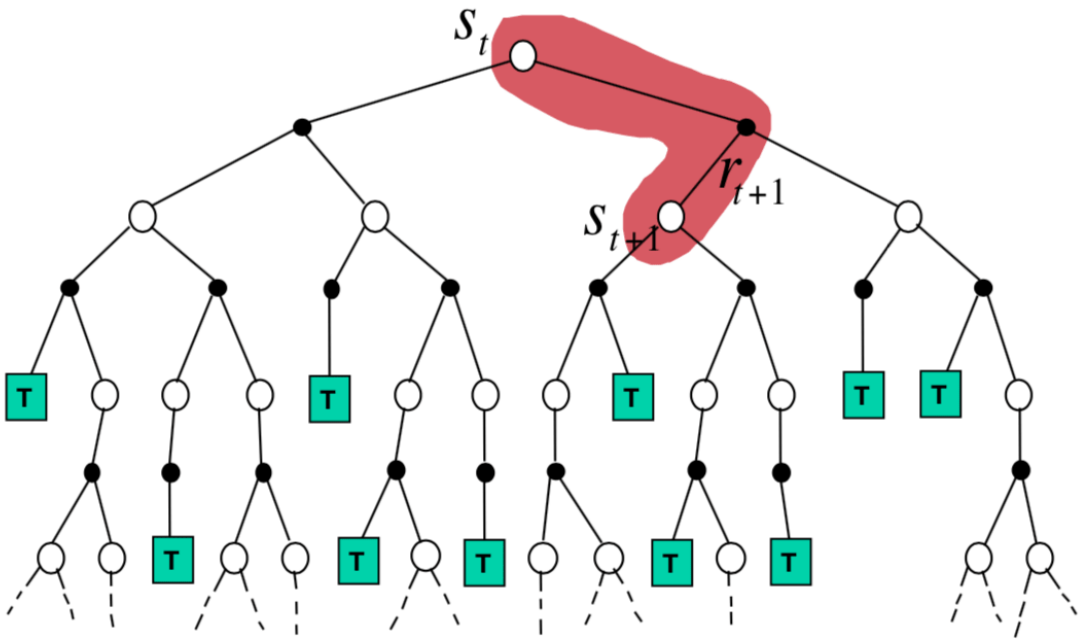


We start from S_t to look-ahead and build a look-ahead tree. What Monte-Carlo do is to sample a episode until it terminates and use the episode to update the value of state S_t.

[۱] https://www.۵۲coding.com.cn/۲۰۱۷/۱۲/۱۶/RL%۲۰-%۲۰Model-Free%۲۰Prediction/

Temporal-Difference Backup

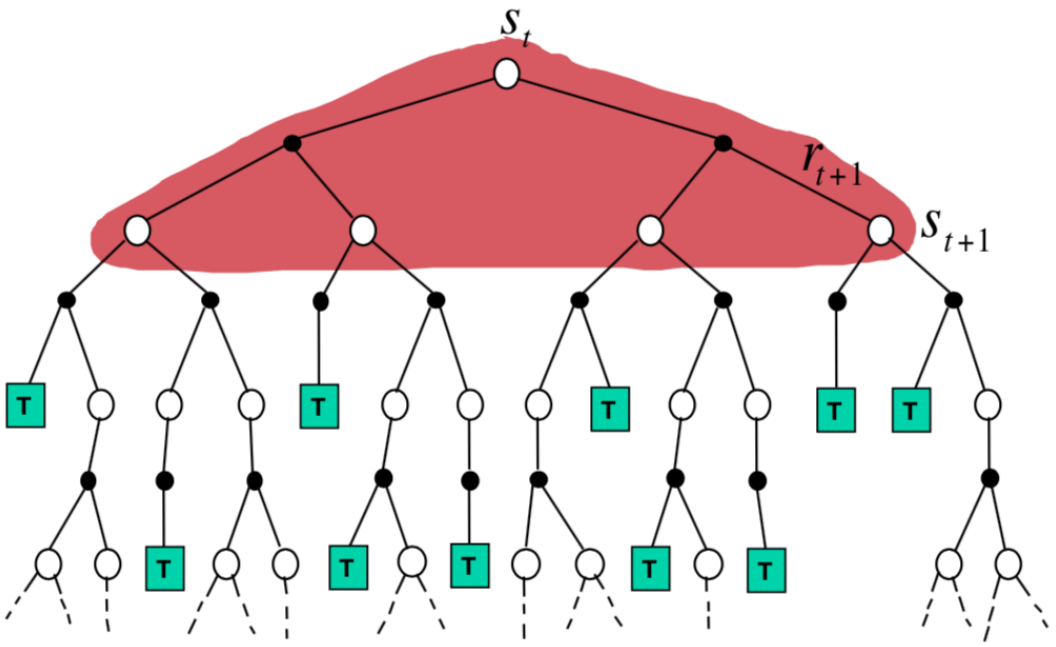
$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$



On the contrary, TD backup just sample one-step ahead and use the value of S_{t+1} to update S_t.

Dynamic Programming Backup

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1}) \right]$$



Bootstrapping and Sampling

- **Bootstrapping**: update involves an estimate
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps
- **Sampling**: update samples an expectation
  - MC samples
  - DP does not sample
  - TD sample

Offline updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\Vert$	$\Vert$	$\Vert$
Forward view	TD(0)	Forward TD(λ)	MC
Online updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\Vert$	$\nVert$	$\nVert$
Forward view	TD(0)	Forward TD(λ)	MC
	$\Vert$	$\Vert$	$\Vert$
Exact Online	TD(0)	Exact Online TD(λ)	Exact Online TD(1)

$=$ here indicates equivalence in total update at end of episode.

Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

TD does not follow the gradient of *any* objective function. This is why TD can diverge when off-policy or using non-linear function approximation. **Gradient TD** follows true gradient of projected Bellman error.

Convergence of Control Algorithms

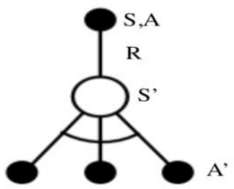
Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

Convergence of Linear Least Squares Prediction Algorithm

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

The **Q-learning** target then simplifies:



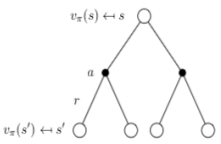
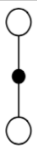
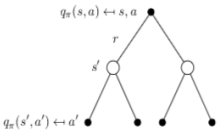
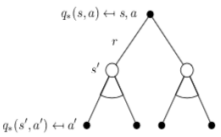
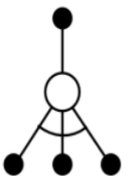
$$Q(S,A) \leftarrow Q(S,A) + \alpha \left(R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right)$$

Of course, the Q-learning control still converges to the optimal action-value

function, $Q(s,a) \rightarrow q_*(s,a)$.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Relationship Between DP and TD

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 Q-Value Iteration	 Q-Learning

Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}\left[R + \gamma V(S') \mid s\right]$	TD Learning $V(S) \overset{\alpha}{\leftarrow} R + \gamma V(S')$
Q-Policy Iteration $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma Q(S', A') \mid s, a\right]$	Sarsa $Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma Q(S', A')$
Q-Value Iteration $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') \mid s, a\right]$	Q-Learning $Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$

where $x \overset{\alpha}{\leftarrow} y \equiv x \leftarrow x + \alpha(y - x)$

فصل هفتم

مبانی یادگیری عمیق

در فصول قبلی، چندین الگوریتم یادگیری تقویتی را معرفی کردیم و توضیح دادیم که آنها چگونه کار کرده و چگونه سیاست بهینه را پیدا می‌کنند. در فصول آینده، با **یادگیری تقویتی عمیق**[۱] (**DRL**) آشنا خواهیم شد که ترکیبی از **یادگیری عمیق** و **یادگیری تقویتی** است. برای درک DRL، باید پایه‌ای قوی در یادگیری عمیق داشته باشیم. بنابراین، در این فصل، چندین الگوریتم مهم یادگیری عمیق را یاد خواهیم گرفت.

یادگیری عمیق زیرمجموعه‌ای از حوزه **یادگیری ماشین** و یکی از مباحث مهم **شبکه‌های عصبی** است. یادگیری عمیق یک دهه است که وجود دارد، اما دلیل محبوبیت آن در حال حاضر به دلیل پیشرفتهای محاسباتی و در دسترس بودن حجم عظیمی‌از داده‌ها است. با این حجم عظیم داده‌ها، الگوریتمهای یادگیری عمیق می‌توانند از الگوریتم‌های یادگیری ماشین کلاسیک بهتر عمل کنند.

ما این فصل را با شناخت نورون‌های بیولوژیکی و مصنوعی شروع می‌کنیم و سپس با **شبکه‌های عصبی مصنوعی**[۲] (**ANN**) و نحوه پیاده سازی آنها آشنا می‌شویم. در ادامه، با چندین الگوریتم یادگیری عمیق جالب مانند **شبکه عصبی بازگشتی**[۳] (**RNN**)، **حافظه کوتاه-مدت دیرپا**[۴] (**LSTM**)، **شبکه عصبی همتابی**[۵] (**CNN**) و **شبکه مولد تخاصمی**[۶] (**GAN**) آشنا خواهیم شد.

- **عصب‌های طبیعی و مصنوعی**
- **شبکه‌های عصبی مصنوعی**
- **شبکه‌های عصبی بازگشتی (RNN ها)**
- **حافظه کوتاه-مدت دیرپا (LSTM)**
- **شبکه‌های عصبی همتابی یا پیچشی (CNN ها)**
- **شبکه‌های مولد تخاصمی یا تقابلی (GAN ها)**

[۱] Deep Reinforcement Learning (DRL)
[۲] Artificial Neural Networks (ANNs)
[۳] Recurrent Neural Network (RNN)
[۴] Long Short-Term Memory (LSTM)
[۵] Convolutional Neural Network (CNN)
[۶] Generative Adversarial Network (GAN)

بیایید فصل را با درک نحوه عملکرد نورون‌های بیولوژیکی و مصنوعی شروع کنیم.

عصب‌های طبیعی و مصنوعی

قبل از ادامه، ابتدا بررسی خواهیم کرد که نورون چیست و چگونه نورون‌ها در مغز ما در واقع کار می‌کند، و سپس در مورد نورون‌های مصنوعی یاد خواهیم گرفت.

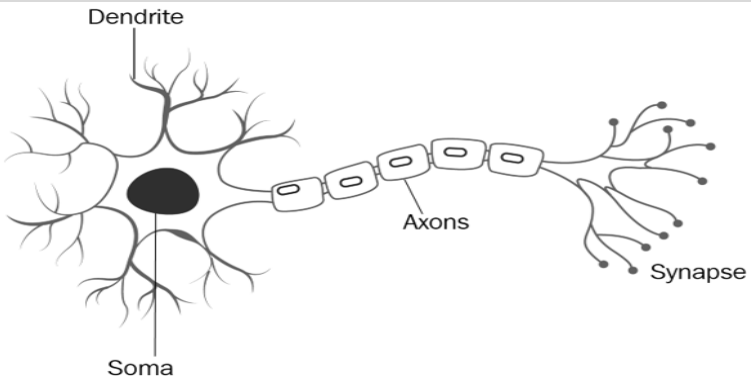
نورون را می‌توان به عنوان واحد محاسباتی پایه در مغز انسان تعریف کرد. نورون‌ها واحدهای اساسی مغز و سیستم عصبی ما هستند. مغز ما تقریبا دارای ۱۰۰ میلیارد نورون است. نورونها از طریق ساختاری به نام **سیناپس**[۱] به یکدیگر متصل می‌شوند که مسئول دریافت داده ورودی از محیط خارجی از طریق اندام‌های حسی، ارسال دستورالعمل‌های حرکتی به عضلات ما و انجام سایر فعالیت‌ها هستند.

یک نورون همچنین می‌تواند ورودی‌های نورون‌های دیگر را از طریق یک ساختار شاخه مانند به نام **دندریت**[۲] دریافت کند. این ورودی‌ها می‌توانند تقویت یا تضعیف شوند. یعنی با توجه به اهمیت آنها، وزین (وزندار) شده و سپس در بدنه سلولی به نام **سوما**[۳] جمع می‌شوند. این ورودی‌ها در بدنه سلول، تجمیع شده و پردازش می‌شوند و سپس از طریق **آکسونها**[۴] حرکت کرده و از آنجا به سایر نورون‌ها ارسال می‌شوند. شکل ۷.۱ یک نورون بیولوژیکی پایه را نشان می‌دهد.

حال، ببینیم نورون‌های مصنوعی چگونه کار می‌کنند. بیایید فرض کنیم سه ورودی $x_۱$، $x_۲$ و $x_۳$ برای پیش‌بینی خروجی y داریم. این ورودی‌ها در اوزان $w_۱$، $w_۲$ و $w_۳$ ضرب شده و به صورت زیر با هم جمع می‌شوند:

$$x_۱.w_۱ + x_۲.w_۲ + x_۳.w_۳$$

[۱] Synapse
[۲] Dendrite
[۳] Soma
[۴] Axons



شکل ۷.۱: نورون بیولوژیک (عصب طبیعی)

اما چرا این ورودی‌ها را در وزنها ضرب می‌کنیم؟ زیرا همه ورودی‌ها در محاسبه خروجی y به یک اندازه مهم نیستند. فرض کنید $x_۲$ در محاسبه خروجی در مقایسه با دو ورودی دیگر اهمیت بیشتری دارد. لذا، مقدار بالاتری را به $w_۲$ نسبت به دو وزن دیگر اختصاص می‌دهیم. بنابراین، با ضرب وزن‌ها در ورودی‌ها، $x_۲$ مقدار بالاتری نسبت به دو ورودی دیگر خواهد داشت. به عبارت ساده، از اوزان برای تقویت ورودی‌ها استفاده می‌شود. پس از ضرب ورودی‌ها در اوزان، آنها را با هم جمع کرده و مقداری به نام سویش یا سویانه[۱] با علامت b به حاصل‌جمع اضافه می‌کنیم:

$$z = (x_۱.w_۱ + x_۲.w_۲ + x_۳.w_۳) + b$$

اگر به معادلهٔ قبلی از نزدیک نگاه کنید، ممکن است آشنا به نظر برسد. آیا z شبیه معادلهٔ رگرسیون خطی نیست؟ آیا این فقط معادلهٔ یک خط مستقیم نیست؟ می‌دانیم که معادلهٔ یک خط مستقیم به صورت زیر داده می‌شود:

$$z = mx + b$$

در اینجا، m وزن‌ها (یا ضرایب هستند)، x ورودی و b سویش (عرض از مبدا) است.

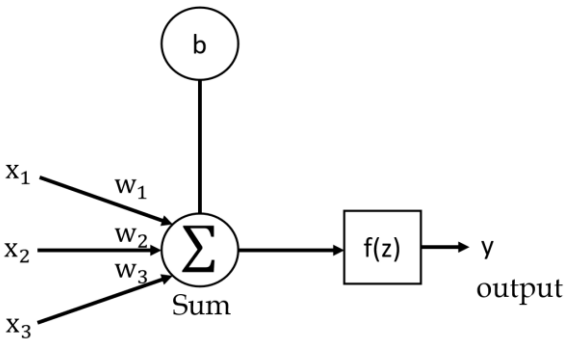
[۱] Bias

به نظر می‌رسد باید معادل فارسی این واژه را در حالت فعل متعددی، فعل لازم، اسم، صفت و دیگر وضعیتها وضع کرد. لذا بعنوان معادل فعل متعددی آن: سویاندن (به سویی روانه کردن) و برای فعل لازم آن: سوییدن (به سویی روانه شدن) را پیشنهاد می‌دهیم. با این حساب: سویانش و سویاننده حالت اسم و صفت فاعلی (از روی فعل اول)، و سویش و سوئیده حالت اسم و صفت فاعلی و مفعولی (برای فعل دوم) در زبان پارسی خواهد شد. لذا، سویش و سویانه اسم آن هستند. بهرحال فارغ از حالت فاعلی و یا مفعولی، بعنوان معادل آن در اینجا، **سویش** و **سویانه** پیشنهاد می‌شود. هرچند واژه‌های سوگیری، متمایل و حتی اُریب همچنان بجای آن استعمال می‌شود.

خب، بله. پس، تفاوت بین نورون‌ها و رگرسیون خطی چیست؟ در نورون‌ها، با اعمال تابع $f(.)$ عملا غیرخطی بودن را به نتیجه یعنی z، اعمال می‌کنیم. این تابع را تابع **فعالسازی** یا **تابع انتقال** می‌نامند. بنابراین، خروجی ما تبدیل می‌شود:

$$y = f(z)$$

شکل ۷.۲ یک نورون مصنوعی را نشان می‌دهد.



شکل ۷.۲: نورون مصنوعی

بنابراین، هر نورون، ورودی ما یعنی x را می‌گیرد، آن را در وزن مورد نظر یعنی w ضرب می‌کند و سویش یعنی b را به آن اضافه می‌کند، که نهایتا z را تشکیل می‌دهد، و سپس تابع فعالسازی را روی z اعمال کرده و خروجی y را بعنوان خروجی تحویل می‌دهد.

شبکه‌های عصبی مصنوعی[۱] (ANN) و لایه‌های آن

در حالی که نورون‌ها واقعا جالب هستند، ما نمی‌توانیم فقط از یک عصب (نورون) برای انجام کارهای پیچیده استفاده کنیم. به همین دلیل است که مغز ما میلیاردها نورون دارد که در لایه‌های مختلف روی هم چیده شده و یک شبکه را تشکیل می‌دهند. به همین ترتیب ، نورون‌های مصنوعی به صورت لایه لایه‌ای چیده شده اند. هر لایه به گونه‌ای

[۱] Artificial Neurons Network (ANN)

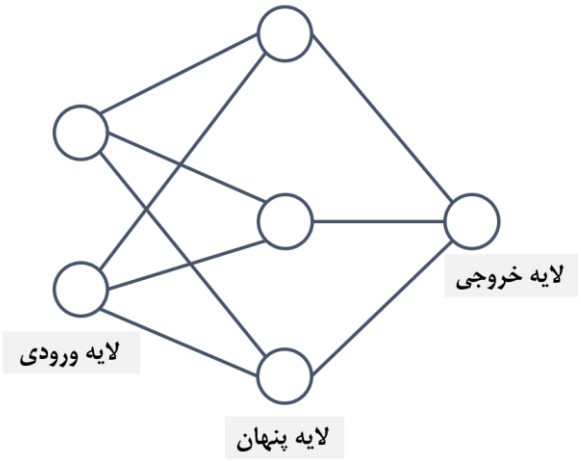
به هم متصل می‌شود که اطلاعات بتوامد از یک لایه به لایه دیگر انتقال یابد.

یک **ANN** معمولی از لایه‌های زیر تشکیل شده است:

- لایه ورودی
- لایه پنهان
- لایه خروجی

هر لایه دارای مجموعه‌ای از نورون‌ها است و نورون‌های یک لایه با تمام نورون‌های لایه‌های دیگر تعامل دارند. با این حال، خود نورون‌های یک لایه با یکدیگر تعامل نخواهند داشت. این وضعیت بدان دلیل است که ارتباط یا یال بین نورون‌های لایه‌های مجاور وجود دارد؛ ولیکن، نورون‌های یک لایه هیچ ارتباطی با هم ندارند. ما از اصطلاح **گره‌ها** یا **واحدها** برای نشان دادن نورون‌ها در شبکه عصبی مصنوعی استفاده می‌کنیم.

شکل ۷.۳ یک **ANN** معمولی را نشان می‌دهد.



شکل ۷.۳: شبکه عصبی مصنوعی

لایه ورودی

لایه ورودی جایی است که ورودیها را به شبکه تغذیه می‌کنیم. تعداد نورون‌ها در لایه ورودی، تعداد ورودی‌هایی است

که به شبکه می‌دهیم. هر ورودی در پیش‌بینی مقدار خروجی تأثیر خواهد داشت. با این حال، هیچ محاسباتی در لایه ورودی انجام نمی‌شود. از این لایه فقط برای انتقال اطلاعات از دنیای خارج به شبکه استفاده می‌شود.

لایه پنهان

هر لایه بین لایه ورودی و لایه خروجی را **لایه پنهان** نامند. این لایه، داده‌های دریافتی از لایه ورودی را پردازش می‌کند. لایه پنهان وظیفه استخراج روابط پیچیده بین ورودی و خروجی را بر عهده دارد. یعنی لایه پنهان، الگو را در مجموعه داده مشخص می‌کند. این لایه عمدتا مسئول یادگیری، نمایش داده‌ها و استخراج ویژگی‌ها است.

هر تعداد لایه پنهان می‌تواند در یک شبکه وجود داشته باشد. ولیکن، ما باید تعداد آنرا با توجه به مسئله خود، انتخاب کنیم. برای حل یک مسئله بسیار ساده، می‌توانیم فقط از یک لایه پنهان استفاده کنیم، اما برای انجام وظایف پیچیده مانند تشخیص تصویر، از لایه‌های پنهان زیادی استفاده می‌کنیم، جایی که هر لایه وظیفه استخراج ویژگی‌های مهم را بر عهده دارد. یک شبکه عصبی را زمانی **شبکه عصبی عمیق** گویند که تعداد زیادی لایه‌های پنهان داشته باشد.

لایه خروجی

پس از پردازش ورودیها، لایه پنهان نتیجه کار خود را به لایه خروجی ارسال می‌کند. همانطور که از نام آن پیداست، لایه خروجی، مقادیر نهایی را منتشر می‌کند. تعداد نورون‌ها در لایه خروجی بر اساس نوع مسئله‌ای است که قرار است شبکه ما حل کند.

اگر مسئله ما، یک طبقه‌بندی باینری باشد، تعداد نورون‌های لایه خروجی، تنها یک لایه است که به ما می‌گوید که هر ورودی، متعلق به کدام کلاس است. اگر مسئله ما، یک طبقه‌بندی چند کلاسی باشد، مثلا پنج کلاس، و اگر بخواهیم احتمال هر کلاس را به عنوان خروجی بدست آوریم، تعداد نورون‌ها در لایه خروجی، پنج تاست که هر کدام احتمال عضویت ورودی را در این کلاسها مشخص می‌کنند. اگر مسئله ما، رگرسیون باشد، باز هم فقط یک نورون در لایه خروجی نیاز داریم.

بررسی توابع فعالسازی

تابع فعالسازی[۱] که همچنین به عنوان **تابع انتقال** نیز شناخته می‌شود، نقش حیاتی در شبکه‌های عصبی ایفا می‌کند. از این توابع برای اعمال وضعیت غیرخطی بودن در شبکه‌های عصبی استفاده می‌شود. همانطور که قبلا یاد گرفتیم، تابع فعالسازی یعنی $f(.)$ را بر روی ورودی تجمیعی یعنی z اعمال می‌کنیم، به بیان دیگر، ورودی که در اوزان ضرب شده و یک سویش هم به آنها اضافه شده است؛ یعنی z در $f(z)$ بصورت زیر بدست می‌آید:

$$z = (input * weights) + bias$$

اگر تابع فعالسازی را اعمال نکنیم، رفتار یک نورون به سادگی شبیه رگرسیون خطی است. هدف از تابع فعالسازی معرفی یک **تبدیلگر غیرخطی** برای یادگیری الگوهای بنیادین پیچیده در داده‌ها است. حال بیایید به برخی از توابع فعالسازی جالب و رایج نگاه کنیم.

تابع سیگموئید

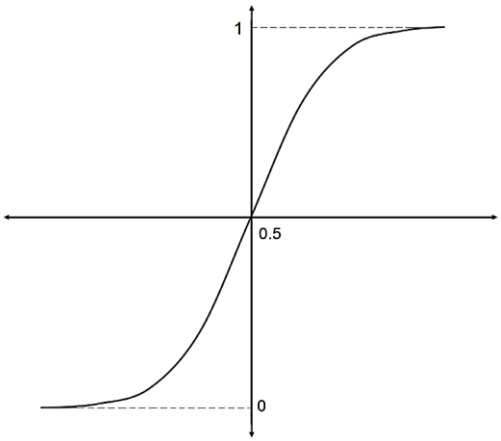
تابع سیگموئید[۲] یکی از متداولترین توابع فعالسازی است. این تابع، مقادیر دریافتی را با عددی بین ۰ و ۱ مقیاس می‌کند. تابع سیگموئید را می‌توان به صورت زیر تعریف کرد:

$$f(x) = \frac{۱}{۱ + e^{-x}}$$

این تابع یک منحنی S شکل است که در شکل ۷.۴ نشان داده شده است.

---

[۱] Activation Function
[۲] Sigmoid Function



شکل ۷.۴: تابع سیگموئید

این تابع، مشتق‌پذیر[۱] است، به این معنی که می‌توانیم شیب منحنی را در هر دو نقطه پیدا کنیم. بعلاوه این تابع، [redacted] است، بدین معنا که یا کاملا غیرافزایشی یا کاملا غیرکاهشی است. تابع سیگموئید به عنوان تابع [redacted] نیز شناخته می‌شود. همانطور که می‌دانیم مقدار احتمال، عددی بین ۰ تا ۱ است و از آنجایی که تابع سیگموئید مقدار دریافتی را می‌فشرد تا در فاصله ۰ تا ۱ به ما خروجی بدهد، لذا از این تابع برای پیش‌بینی احتمال خروجی استفاده می‌شود.

تابع تانژانت هایپربولیك (هذلولی)

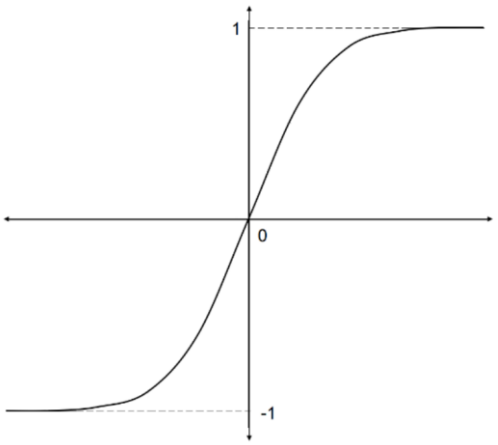
یک **تابع تانژانت هایپربولیک**[۴] یا **تابع مماس هذلولی** (**tanh**) مقدار خروجی را عدد بین ۱- تا ۱+ را بدست می‌دهد و به صورت زیر بیان می‌شود:

$$f(x) = \frac{۱ - e^{-۲x}}{۱ + e^{-۲x}}$$

[۱] Differentiable
[۲] Monotonic
[۳] Logistic Function
ریشه و معنای واژه لوگستیک (که معمولا لجستیک نوشته و خوانده می‌شود) یا با ریشه و مفهوم واژه لجستیک در مدیریت زنجیره ارزش و نیز ترابری نظامی، یکسان نیست.

[۴] Hyperbolic Tangent (tanh)

این تابع نیز شبیه منحنی S شکل است. برخلاف تابع سیگموئید، که مرکزیت آن عدد ۰.۵ است، این تابع، همانطور نمودار زیر نشان می‌دهد، با مرکزیت صفر است.



شکل ۷.۵: تابع tanh

تابع واحد خطی اصلاح شده

تابع واحد خطی اصلاح شده[۱] (ReLU) یکی دیگر از توابع فعالسازی رایج بوده و مقداری از صفر تا بی نهایت را بعنوان خروجی می‌دهد. این تابع اساسا یک تابع **تکه تکه** یا **قطعه‌وار**[۲] است و می‌توان آن را به صورت زیر بیان کرد.

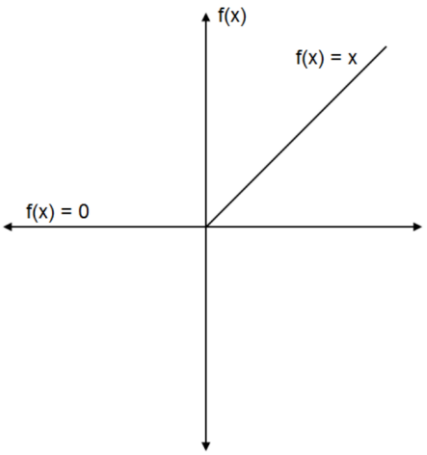
$$f(x) = \begin{cases} & for\ \ x < \\ x & for\ \ x \geq \end{cases}$$

یعنی $f(x)$ زمانی که مقدار x کمتر از صفر باشد صفر را برمی‌گرداند و $f(x)$ زمانی که مقدار x بزرگتر یا مساوی صفر باشد، خود x را برمی‌گرداند. همچنین می‌توان آن را به صورت زیر بیان کرد:

$$f(x) = max(\ , x)$$

[۱] Rectified Linear Unit (ReLU)
[۲] Piecewise Function

شکل ۷.۶ تابع ReLU را نشان می‌دهد:



شکل ۷.۶: تابع واحد خطی اصلاح شده

همانطور که در نمودار بالا می‌بینیم، وقتی هر ورودی منفی را به تابع ReLU تغذیه می‌کنیم، این تابع، ورودی منفی را به صفر تبدیل می‌کند.

تابع بیشینه_نرم

تابع بیشینه–نرم اساسا تعمیم تابع **سیگموئید** است. از این تابع معمولا روی لایه نهایی شبکه و هنگام انجام وظایف طبقه‌بندی چندکلاسه اعمال می‌شود. این تابع، احتمالات هر کلاس را به ازاء هر خروجی می‌دهد و بنابراین، مجموع مقادیر بیشینه–نرم همیشه برابر با ۱ خواهد بود. این تابع را می‌توان به صورت زیر نشان داد.

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

همانطور که در شکل ۷.۷ نشان داده شده است، تابع بیشینه–نرم ورودی‌های خود را به احتمالات تبدیل می‌کند.

[1] softmax

$$\begin{bmatrix} ۰.۵ \\ ۱.۳ \\ ۱.۱ \end{bmatrix} \rightarrow [Softmax] \rightarrow \begin{bmatrix} ۰.۱۹۸ \\ ۰.۴۴۰ \\ ۰.۳۶۰ \end{bmatrix}$$

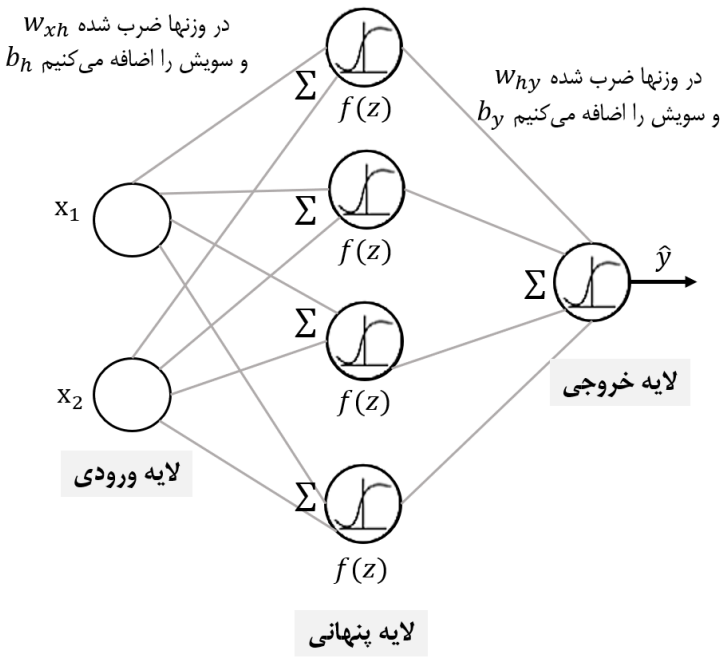
شکل ۷.۷: عملکرد بیشینه–نرم

اکنون که برخی از توابع مختلف فعالسازی را معرفی کردیم، در بخش بعدی، با «انتشار رو به جلو» در شبکه عصبی مصنوعی آشنا خواهیم شد

انتشار رو به جلو[۱] در شبکه‌های عصبی مصنوعی

در این بخش، خواهیم دید که چگونه یک **ANN** از طریق نورون‌های چیده شده در لایه‌ها، یاد می‌گیرد. تعداد لایه‌های یک شبکه برابر با تعداد لایه‌های پنهان به علاوه تعداد لایه‌های خروجی است. ما هنگام محاسبه تعداد لایه‌ها در یک شبکه، لایه ورودی را در نظر نمی‌گیریم. همانطور که در نمودار زیر نشان داده شده است، یک شبکه عصبی دو لایه: با یک لایه ورودی x، یک لایه پنهان h و یک لایه خروجی y را در نظر بگیرید.

[۱] Forward propagation



شکل ۷.۸: انتشار رو به جلو در شبکه عصبی مصنوعی

بیایید در نظر بگیریم که ما دو ورودی داریم، x_1 و x_2، و باید خروجی یعنی $\hat{y}$ را پیش‌بینی کنیم. از آنجایی که ما دو ورودی داریم، تعداد نورون‌ها در لایه ورودی دو تاست. تعداد نورون‌ها در لایه پنهان را روی چهار و تعداد نورون‌ها در لایه خروجی را روی یک تنظیم می‌کنیم. اکنون، ورودی‌ها در وزنها ضرب شده و سپس سویش (سویانه) را اضافه می‌کنیم و مقدار حاصل را به لایه پنهانی که تابع فعالسازی در آن اعمال می‌شود، ارسال می‌کنیم.

قبل از آن، باید ماتریس اوزان را مقداردهی اولیه کنیم. در دنیای واقعی، ما نمی‌دانیم کدام ورودی مهمتر از دیگری است تا بتوانیم آنها را وزن بیشتری داده و خروجی را محاسبه کنیم. بنابراین، ما به طور تصادفی (بختکی) اوزان و سویش را مقداردهی اولیه می‌کنیم. وزن و سویش بین ورودی به لایه پنهان را به ترتیب با W_{xh} و b_h نشان می‌دهیم. در مورد ابعاد ماتریس وزن چطور؟ ابعاد ماتریس وزن باید تعداد نورون‌های لایه فعلی (ضربدر ×) تعداد نورون‌ها در لایه بعدی باشد. چرا اینطور است؟

زیرا این یک قاعده پایه ضرب ماتریسهاست. برای ضرب هر دو ماتریس، AB، تعداد ستون‌های ماتریس A باید برابر با تعداد سطرهای ماتریس B باشد. بنابراین، بُعد ماتریس وزن W_{xh} باید تعداد نورون‌ها در لایه ورودی × تعداد

نورون‌ها در لایه پنهان، یعنی ۴ × ۲ باشد.

$$z_1 = XW_{xh} + b_h$$

معادله قبلی نشان می‌دهد که z_1 = (وزن × ورودی) + سویش است. اکنون، این مقدار به لایه پنهان منتقل می‌شود. در لایه پنهان، یک تابع فعالسازی را روی z_1 اعمال می‌کنیم. بیایید از تابع فعالسازی سیگموئید σ استفاده کنیم. پس، می‌توانیم بنویسیم:

$$a_1 = \sigma(z_1)$$

پس از اعمال تابع فعالسازی، دوباره نتیجه یعنی a_1 را در یک ماتریس وزن جدید ضرب می‌کنیم و یک مقدار سویش جدید به آن اضافه می‌کنیم که بین لایه پنهان و لایه خروجی جریان دارد. ما می‌توانیم این ماتریس وزن و سویش را به ترتیب با علائم W_{hy} و b_y نشان دهیم. بُعد ماتریس وزن یعنی W_{hy}، تعداد نورون‌ها در لایه پنهان × تعداد نورون‌ها در لایه خروجی خواهد بود. از آنجایی که ما چهار نورون در لایه پنهان و یک نورون در لایه خروجی داریم، ابعاد ماتریس یعنی W_{hy} بصورت۱×۴ خواهد بود. بنابراین، ما a_1 را در ماتریس وزن یعنی W_{hy}، ضرب کرده و سویش یعنی b_y را به آن اضافه می‌کنیم، و نهایتا نتیجه یعنی z_2 را به لایه بعدی که لایه خروجی است، منتقل می‌کنیم:

$$z_2 = a_1 W_{hy} + b_y$$

حال، در لایه خروجی، تابع سیگموئید را روی z_2 اعمال می‌کنیم، که منجر به یک مقدار خروجی می‌شود:

$$\hat{y} = \sigma(z_2)$$

کل این فرآیند از لایه ورودی تا لایه خروجی به عنوان **انتشار رو به جلو** شناخته می‌شود. بنابراین، به منظور پیش‌بینی مقدار خروجی، ورودی‌ها از لایه ورودی به لایه خروجی منتشر شده یا به حرکت درمی‌آیند. در طول این انتشار، آنها در وزن مربوطه در هر لایه ضرب شده و یک تابع فعالسازی بر روی آنها اعمال می‌شود. مراحل کامل انتشار رو به جلو به شرح زیر است:

$$z_1 = XW_{xh} + b_h$$

$$a_1 = \sigma(z_1)$$

$$z_2 = a_1 W_{hy} + b_y$$

$$\hat{y} = \sigma(z_2)$$

مراحل انتشار رو به جلو در مثال قبل را می‌توان در پایتون به شرح زیر پیاده‌سازی کرد:

```
def forward_prop(X):
    z1 = np.dot(X,Wxh) + bh
    a1 = sigmoid(z1)
    z2 = np.dot(a1,Why) + by
    y_hat = sigmoid(z2)

    return y_hat
```

انتشار رو به جلو جالب است، اینطور نیست؟ اما چگونه بفهمیم خروجی تولید شده توسط شبکه عصبی صحیح است یا خیر؟ ما تابع جدیدی به نام **تابع هزینه** (J) را تعریف می‌کنیم که به عنوان **تابع ضرر**[1] (L) نیز شناخته می‌شود، که به ما می‌گوید شبکه عصبی ما چقدر خوب عمل می‌کند. توابع هزینه‌های مختلفی وجود دارد. ما از میانگین مربعات خطا به عنوان تابع هزینه استفاده خواهیم کرد که می‌تواند به عنوان میانگین اختلاف مربع بین خروجی واقعی و خروجی پیش‌بینی شده تعریف شود:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

در اینجا، n تعداد نمونه‌های آموزشی، yخروجی واقعی و $\hat{y}$ خروجی پیش‌بینی شده است.

خب، بنابراین ما یاد گرفتیم که از یک تابع هزینه برای ارزیابی شبکه عصبی استفاده می‌شود. یعنی به ما می‌گوید که

[1] Loss Function

شبکه عصبی ما چقدر در پیش‌بینی خروجی خوب عمل می‌کند. اما سوال این است که شبکه ما واقعا کجا یاد می‌گیرد؟ در انتشار رو به جلو، شبکه فقط در تلاش است تا خروجی را پیش‌بینی کند. اما چگونه یاد می‌گیرد که خروجی صحیح را پیش‌بینی کند؟ در بخش بعدی به بررسی این موضوع می‌پردازیم.

یک ANN چگونه یاد می‌گیرد ؟

اگر هزینه یا ضرر بسیار زیاد باشد، به این معنی است که شبکهٔ ما خروجی صحیح را پیش‌بینی نمی‌کند. بنابراین، هدف ما کمینه کردن تابع هزینه است تا پیش‌بینی‌های شبکه عصبی ما بهتر باشد. چگونه می‌توانیم تابع هزینه را به حداقل برسانیم؟ یعنی چگونه می‌توانیم ضرر/ هزینه را به حداقل برسانیم؟ ما یاد گرفتیم که شبکه عصبی با استفاده از انتشار رو به جلو پیش‌بینی می‌کند. بنابراین، اگر بتوانیم برخی از مقادیر را در انتشار رو به جلو تغییر دهیم، می‌توانیم خروجی صحیح را پیش‌بینی کرده و زیان را به حداقل برسانیم. ولی چه مقادیری را می‌توانیم در انتشار رو به جلو تغییر دهیم؟ بدیهی است که ما نمی‌توانیم ورودی و خروجی را تغییر دهیم. اکنون فقط مقادیر اوزان و سویش باقی مانده که امکان تغییر دارند. به یاد داشته باشید که ما ماتریسهای وزن را به طور تصادفی مقداردهی اولیه کردیم. از آنجایی که وزن‌ها تصادفی هستند، آنها دقیق و درست نخواهند بود. اکنون، این ماتریسهای وزنی (W_{hy} و W_{xh}) را به‌روز می‌کنیم به گونه‌ای که شبکه عصبی ما خروجی صحیحی بدست دهد. خب این ماتریس‌های وزن را چگونه به‌روز کنیم؟ در اینجا از یک تکنیک جدید به نام **گرادیان کاهشی** یا **نشیب نزولی**[۱] استفاده می‌کنیم.

با گرادیان کاهشی یا نشیب نزولی، شبکه عصبی قادر است مقادیر بهینه ماتریسهای اوزان (که ابتدائا مقادیر اولیه تصادفی داشتند) را یاد بگیرد. با مقادیر بهینه وزن‌ها، شبکه ما می‌تواند خروجی صحیح را پیش‌بینی کرده و ضرر یا هزینه را به حداقل برساند.

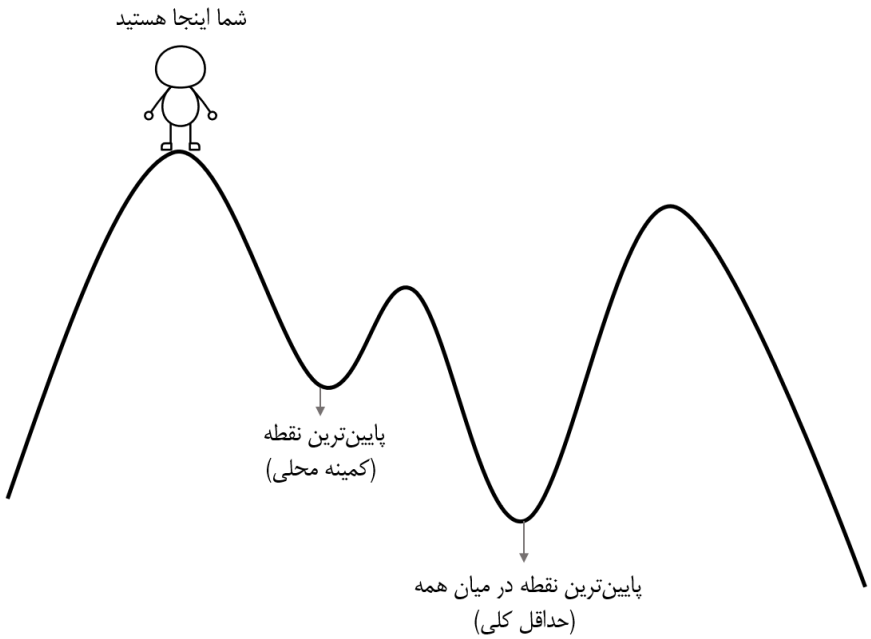
اکنون، نحوه یادگیری مقادیر بهینه وزن‌ها با استفاده از نشیب نزولی را بررسی خواهیم کرد. گرادیان کاهشی یکی از متداول ترین الگوریتمهای بهینه‌سازی است. از این روش برای به حداقل رساندن تابع هزینه استفاده می‌شود، که به

[۱] Gradient Descent

ما امکان می‌دهد خطا را به حداقل رسانده و کمترین مقدار خطای ممکن را بدست آوریم. اما نشیب نزولی چگونه اوزان بهینه را پیدا می‌کند؟ بیایید با یک قیاس شروع کنیم.

همانطور که در نمودار زیر نشان داده شده است، تصور کنید که در بالای یک تپه هستیم و می‌خواهیم به پایین‌ترین نقطه تپه برسیم. ممکن است مناطق زیادی وجود داشته باشند که پایین‌ترین نقاط تپه به نظر می‌رسند، اما ما باید به نقطه‌ای برسیم که واقعا پایین‌ترین نقطه است.

یعنی نباید در نقطه‌ای گیر کنیم که معتقد باشیم پایین‌ترین نقطه است در حالی که نقطه دیگری بعنوان پایین‌ترین نقطه در میان همه (حداقل کلی)[۱] وجود دارد:

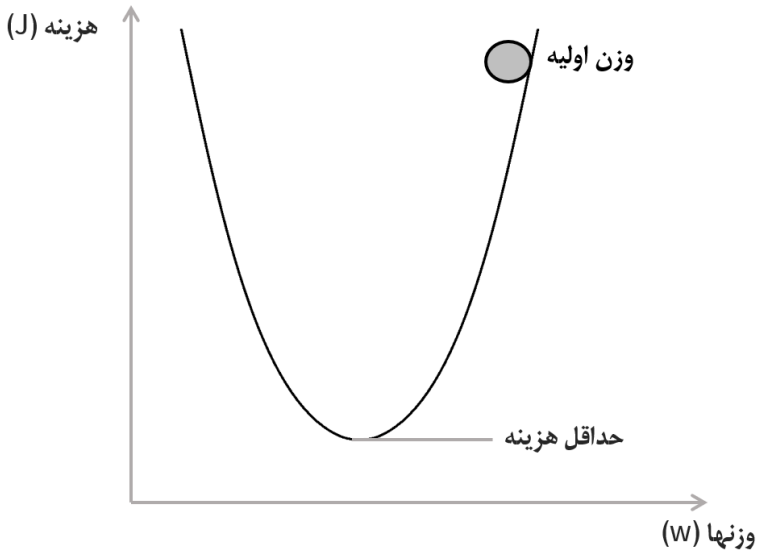


شکل ۷.۹: قیاس نشیب نزولی

به طور مشابه، ما می‌توانیم تابع هزینه خود را به صورت زیر نشان دهیم. این نمودار، هزینه در برابر وزن، نمایش می‌دهد. هدف ما به حداقل رساندن تابع هزینه است. یعنی باید به پایین‌ترین نقطه برسیم که در آن هزینه حداقل

[۱] Global Lowest Point

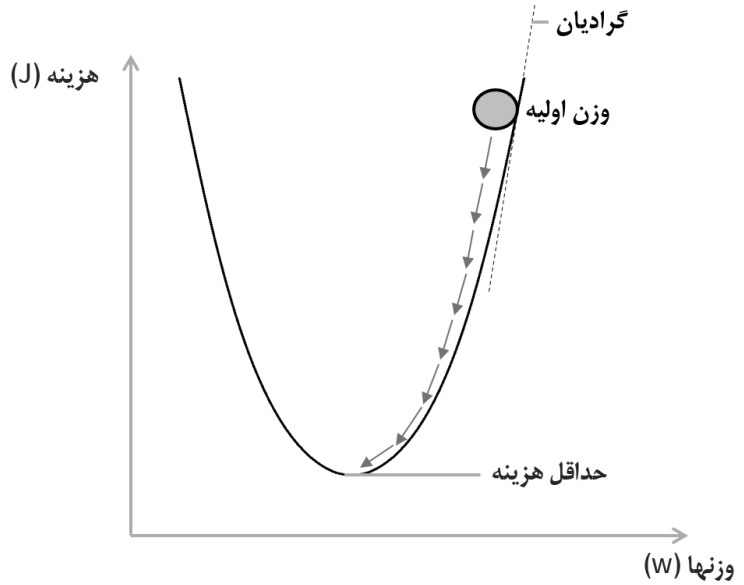
باشد. نقطه سیاه رنگ در نمودار زیر، اوزان اولیه تصادفی را نشان می‌دهد. اگر این نقطه را به سمت پایین حرکت دهیم، می‌توانیم به نقطه‌ای برسیم که هزینه حداقل باشد.



شکل ۷.۱۰: گرادیان نزولی (نشیب کاهشی)

اما چگونه می‌توانیم این نقطه (یا وزن اولیه) را به سمت پایین حرکت دهیم؟ چگونه می‌توانیم پایین بیاییم و به پایین‌ترین نقطه برسیم؟ گرادیان‌ها برای حرکت از یک نقطه به نقطه دیگر استفاده می‌شوند. بنابراین، می‌توانیم این نقطه (وزن اولیه) را با محاسبه گرادیان تابع هزینه بر حسب آن نقطه (وزن اولیه)، حرکت دهیم که بزبان ریاضی محاسبه مقدار $\frac{\partial J}{\partial W}$ است.

همانطور که در نمودار زیر نشان داده شده است، گرادیان‌ها (یا نشیبها)، مشتقاتی هستند که در واقع شیب یک خط مماس را نشان می‌دهند. بنابراین، با محاسبه گرادیان، ما نزول می‌کنیم (به سمت پایین حرکت می‌کنیم) و به پایین‌ترین نقطه‌ای می‌رسیم که هزینه حداقل است. کاهش گرادیان، یک الگوریتم بهینه‌سازی مرتبه اول است، به این معنی که ما فقط اولین مشتق را هنگام انجام به‌روزرسانی‌ها در نظر می‌گیریم:



شکل ۷.۱۱: گرادیان کاهشی (نشیب نزولی)

بنابراین، با گرادیان کاهشی (نشیب نزولی)، اوزان خود را به موقعیتی منتقل می‌کنیم که هزینه آن حداقل باشد. اما با این حال، چگونه وزن‌ها را به‌روز کنیم؟

در نتیجهٔ انتشار رو به جلو، ما در لایه خروجی هستیم. اکنون شبکه را از لایه خروجی به لایه ورودی **بازنشر** (یا **پس‌انتشار**[۱]) می‌کنیم و گرادیان تابع هزینه را با توجه به تمام وزن‌های بین لایه خروجی و ورودی محاسبه می‌کنیم تا بتوانیم خطا را به حداقل برسانیم. پس از محاسبه گرادیان‌ها، وزن‌های قدیمی خود را با استفاده از **قاعده به‌روز رسانی اوزان** که در زیر آمده است به‌روز می‌کنیم:

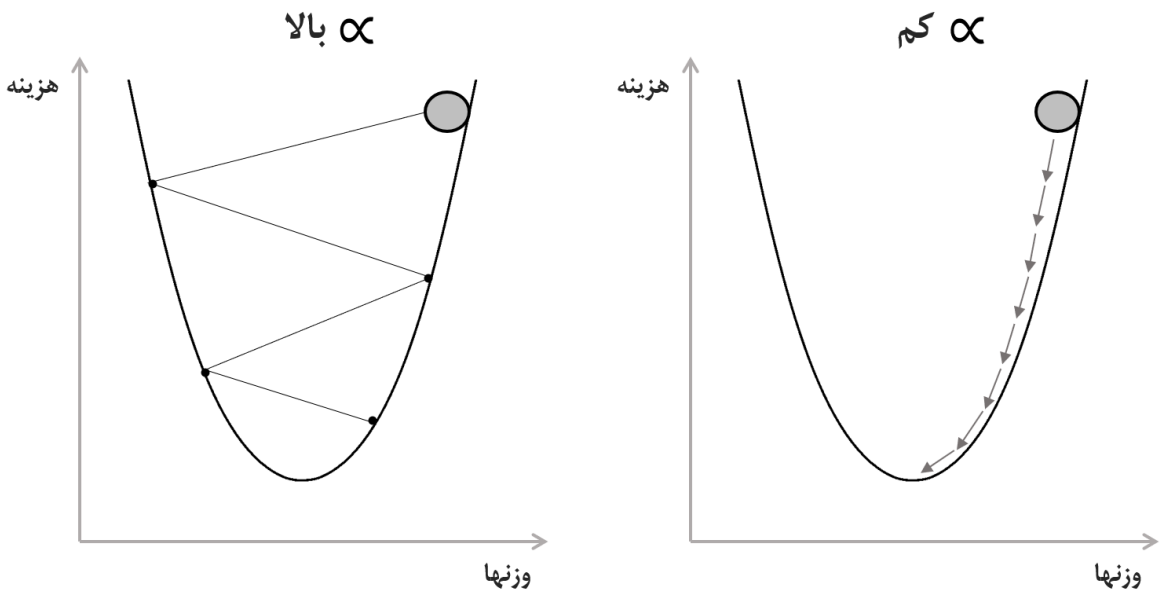
$$W = W - \alpha \frac{\partial J}{\partial W}$$

این بدان معناست که *وزن جدید = وزن قبلی منهای* (α × *نشیب*) است.

[۱] Backpropagate

α چیست؟ به آن **میزان** یا **نرخ یادگیری**[۱] می‌گویند. همانطور که در نمودار زیر نشان داده شده است، اگر سرعت یادگیری کم باشد، یک قدم کوچک به سمت پایین برمی‌داریم و نزول گرادیان ما می‌تواند کُند باشد.

گرچه نرخ یادگیری بالا، باعث می‌شود گام بزرگی برداشته و نزول گرادیان ما سریع شود، اما ممکن است نتوانیم به کمینه کلی (سراسری) برسیم و در کمینه محلی، گیر کنیم. بنابراین، میزان یادگیری باید به طور بهینه انتخاب شود:



شکل ۷.۱۲: تأثیر میزان یادگیری

کل این فرآیندِ باز انتشار شبکه از لایه خروجی به لایه ورودی و به‌روز رسانی وزن شبکه با استفاده از گرادیان کاهشی برای به حداقل رساندن زیانها را **پس–انتشار خطا** می‌نامند. اکنون که فهم اولیه‌ای از پس–انتشار داریم، درک خود را گام به گام در مورد این موضوع با جزئیات بیاموزید. ما قصد داریم به ریاضیات جالبی نگاه کنیم، بنابراین کلاه حساب دیفرانسیل و انتگرال خود را بپوشید و مراحل را دنبال کنید.

بنابراین، ما دو وزن داریم، یکی W_{xh}، که وزن از لایه ورودی به لایه پنهان است، و دیگری W_{hy}، که وزن لایه میان لایه پنهان و لایه خروجی است. ما باید مقادیر بهینه را برای این دو وزن آنگونه پیدا کنیم که کمترین خطا را

[۱] Learning Rate

به ما بدهد. بنابراین، ما باید مشتق تابع هزینه J را با توجه به این وزن‌ها محاسبه کنیم. از آنجایی که ما در حال پس‌انتشار هستیم، یعنی از لایه خروجی به لایه ورودی می‌رویم، اولین وزن ما W_{hy} خواهد بود. بنابراین، اکنون باید مشتق J را بر حسب W_{hy} محاسبه کنیم. خب حالا مشتق را چگونه محاسبه کنیم؟ ابتدا، بیایید تابع هزینه خود را به یاد بیاوریم:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

ما نمی‌توانیم مشتق را مستقیما از معادله بالا محاسبه کنیم زیرا عبارت W_{hy} در آن وجود ندارد. بنابراین، به جای محاسبه مستقیم مشتق، مشتق جزئی را محاسبه می‌کنیم. بیایید معادله انتشار رو به جلو خود را به یاد بیاوریم:

$$\hat{y} = \sigma(z_2)$$

$$z_2 = a_1 W_{hy} + b_y$$

ابتدا مشتق جزئی از تابع هرینه را برحسب $\hat{y}$ محاسبه کرده و سپس مشتق جزئی از $\hat{y}$ را بر حسب z_2 محاسبه می‌کنیم. از روی z_2، می‌توانیم مستقیما مشتق مورد نظر خود یعنی برحسب W_{hy} را محاسبه کنیم. این اساسا یک **قاعده زنجیره‌ای مشتق‌گیری** است. بنابراین، مشتق J بر حسب W_{hy} به شرح زیر می‌شود:

$$\frac{\partial J}{\partial W_{hy}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{dz_2}{dW_{hy}} \qquad (1)$$

اکنون، ما هر یک از عبارات معادله قبلی را محاسبه می‌کنیم:

$$\frac{\partial J}{\partial \hat{y}} = (y - \hat{y})$$

$$\frac{\partial \hat{y}}{\partial z_2} = \sigma'(z_2)$$

در اینجا، σ' مشتق تابع فعالسازی سیگموئید ما است. ما می‌دانیم که تابع سیگموئید بشکل زیر است:

$$\sigma(z) = \frac{}{+ e^{-z'}}$$

بنابراین مشتق تابع سیگموئید بصورت می‌شود.

$$\sigma'(z) = \frac{e^{z}}{(+ e^{-z})}$$

مشتق جزئی بعدی بصورت زیر خواهد شد:

$$\frac{dz_۲}{dW_{hy}} = a_۱$$

بنابراین، با جایگزینی تمام عبارات قبلی در معادله (۱) می‌توانیم بنویسیم:

$$\frac{\partial J}{\partial W_{hy}} = (y - \hat{y}).\sigma'(z_۲).a_۱ \qquad (۲)$$

اکنون باید مشتق J را با توجه به وزن بعدی خود، W_{xh} محاسبه کنیم.

به طور مشابه، ما نمی‌توانیم مشتق W_{xh} را مستقیما از J محاسبه کنیم زیرا جمله W_{xh} در تابع J وجود ندارد. بنابراین، ما باید از قاعده زنجیره‌ای استفاده کنیم. بیایید مراحل انتشار رو به جلو را دوباره به یاد بیاوریم:

$$\hat{y} = \sigma(z_۲)$$

$$z_۲ = a_۱W_{hy} + b_y$$

$$a_۱ = \sigma(z_۱)$$

$$z_۱ = XW_{xh} + b_h$$

حال، طبق قانون زنجیره ، مشتق J بر حسب W_{xh} به صورت زیر خواهد شد:

$$\frac{\partial J}{\partial W_{xh}} = \frac{\partial J}{\partial \hat{y}} . \frac{\partial \hat{y}}{\partial z} . \frac{\partial z}{\partial a} . \frac{\partial a}{\partial z} . \frac{\partial z}{\partial W_{xh}} \qquad (۳)$$

ما قبلا دیدیم که چگونه دو عبارت اول را در معادله قبلی محاسبه کنیم.

اکنون خواهیم دید که چگونه بقیه عبارات را محاسبه کنیم:

$$\frac{\partial z_۲}{\partial a_۱} = W_{hy}$$

$$\frac{\partial a_۱}{\partial z_۱} = \sigma'(z_۱)$$

$$\frac{\partial z_۱}{dW_{xh}} = X$$

بنابراین ، با جایگزینی تمام عبارات قبلی در معادله (۳) ، می‌توانیم بنویسیم:

$$\frac{\partial J}{\partial W_{xh}} = (y - \hat{y}) . \sigma'(z_۲) . \partial W_{hy} . \sigma'(z_۱) . x \qquad (۴)$$

بعد از اینکه گرادیان‌ها را برای هر دو وزن W_{hy} و W_{xh} محاسبه کردیم، وزن‌های اولیه خود را طبق قاعده به‌روزرسانی اوزان، به‌روز می‌کنیم:

$$W_{hy} = W_{hy} - \alpha \frac{\partial J}{\partial W_{hy}} \qquad (۵)$$

$$W_{xy} = W_{xy} - \alpha \frac{\partial J}{\partial W_{xh}} \qquad (۶)$$

خودشه! به این ترتیب وزن یک شبکه را به‌روز می‌کنیم و زیان را به حداقل می‌رسانیم.

اکنون، بیایید ببینیم که چگونه الگوریتم پس–انتشار را در پایتون پیاده‌سازی کنیم.

در هر دو معادله (۲) و (۴)، ما عبارت $(\hat{y} - y).\sigma'(z)$ را داریم. بنابراین به جای اینکه بارها و بارها آنها را محاسبه کنیم، فقط آنها را `delta2` می‌نامیم:

```
delta2 = np.multiply(-(y-yHat),sigmoidPrime(z2))
```

اکنون، گرادیان را بر حسب W_{hy} محاسبه می‌کنیم. به معادله (۲) مراجعه کنید:

```
dJ_dWhy = np.dot(a1.T,delta2)
```

ما گرادیان را بر حسب W_{xh} محاسبه می‌کنیم. به معادله (۴) مراجعه کنید:

```
delta1 = np.dot(delta2,Why.T)*sigmoidPrime(z1)

dJ_dWxh = np.dot(X.T,delta1)
```

ما وزن‌ها را با توجه به معادله قاعده به‌روزرسانی اوزان (۵) و (۶) به شرح زیر، به‌روز خواهیم کرد

```
Wxh = Wxh - alpha * dJ_dWhy
Why = Why - alpha * dJ_dWxh
```

کد کامل برای پس–انتشار به شرح زیر است:

```
def backword_prop(y_hat, z1, a1, z2):
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))
    dJ_dWhy = np.dot(a1.T, delta2)

    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)
    dJ_dWxh = np.dot(X.T, delta1)

    Wxh = Wxh - alpha * dJ_dWhy
    Why = Why - alpha * dJ_dWxh

    return Wxh,Why
```

تموم شد! جدای از این، انواع مختلفی از روش‌های کاهش گرادیان مانند **گرادیان کاهشی تصادفی**، **گرادیان کاهشی کوچک-دسته** RMSprop،Adam و غیره وجود دارد.

قبل از ادامه، بیایید با برخی از اصطلاحات پرکاربرد در شبکه‌های عصبی آشنا شویم:

- **عبور رو به جلو یا گذر پیشرو**[۱]: عبور رو به جلو به معنای انتشار رو به جلو از لایه ورودی به لایه خروجی است.
- **عبور رو به عقب یا گذر پسرو**[۲]: عبور به عقب به معنای پس‌انتشار از لایه خروجی به لایه ورودی است.
- **دوره**[۳]: اپوک یا دوره، تعداد دفعاتی که شبکه عصبی کل داده‌های آموزشی ما را می‌بینند، مشخص می‌کند. بنابراین می‌توان گفت که یک دوره برای همه نمونه‌های آموزشی برابر با یک عبور رو به جلو و یک عبور رو به عقب است.
- **اندازه دسته**[۴]: اندازه دسته، تعداد نمونه‌های آموزشی استفاده شده در [illegible] را مشخص می‌کند.
- **تعداد تکرارها**[۵]: تعداد تکرارها به معنای تعداد عبورها یا گذرهاست؛ که در آن یک عبور = یک عبور رو به

[۱] Forward Pass
[۲] Backward Pass
[۳] Epoch
[۴] Batch Size
[۵] Number of Iterations

جلو + یک عبور رو به عقب است.

گیریم که ما ۱۲،۰۰۰ نمونه آموزشی داریم و اندازه دسته ما ۶،۰۰۰ است. سپس دو تکرار طول می‌کشد تا یک دوره را کامل کنیم. یعنی در اولین تکرار، ۶،۰۰۰ نمونه اول را عبور می‌کنیم و یک عبور رو به جلو و یک عبور به عقب انجام می‌دهیم. در تکرار دوم، ۶،۰۰۰ نمونه بعدی را عبور می‌دهیم و یک پاس رو به جلو و یک پاس به عقب انجام می‌دهیم. پس از دو تکرار، شبکه عصبی ما کل ۱۲،۰۰۰ نمونه آموزشی را می‌بیند که آن را به یک دوره تبدیل می‌کند.

همایندِ مطالب

با کنار هم قرار دادن تمام مفاهیمی‌که تاکنون آموخته ایم، خواهیم دید که چگونه یک شبکه عصبی را از ابتدا بسازیم. ما خواهیم فهمید که چگونه شبکه عصبی یاد می‌گیرد تا عملگر گیت XOR را انجام دهد. گیت XOR فقط زمانی ۱ را برمی‌گرداند که دقیقا فقط یکی از ورودی‌های آن ۱ باشد، در غیر این صورت همانطور که در جدول ۷.۱ نشان داده شده است، مقدار۰ را برمی‌گرداند.

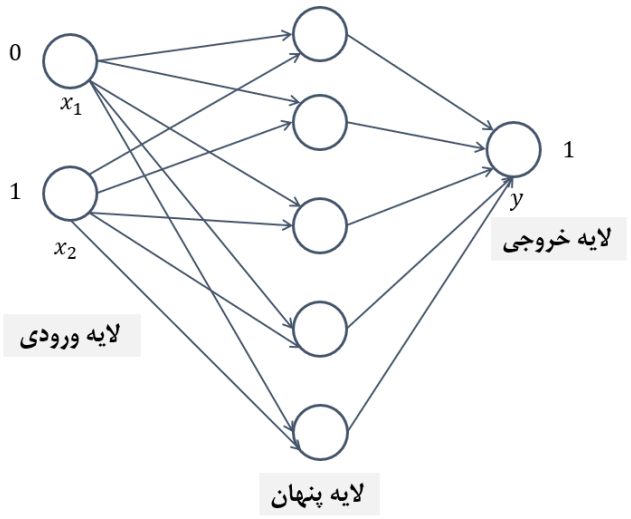
Input(x)		Output(y)
x_1	x_2	y
۰	۰	۰
۰	۱	۱
۱	۰	۱
۱	۱	۰

جدول ۷.۱: عملگر XOR

ساخت یك شبکه عصبی از ابتدا

برای انجام عملیات عملگر گیت XOR، همانطور که در نمودار زیر نشان داده شده است، یک شبکه عصبی دو لایه ساده می‌سازیم. همانطور که می‌بینید، ما یک لایه ورودی با دو گره، یک لایه پنهان با پنج گره و یک لایه خروجی

متشکل از یک گره داریم:



شکل ۷.۱۳: شبکه عصبی مصنوعی

ما گام به گام خواهیم فهمید که چگونه یک شبکه عصبی منطق XOR را یاد می‌گیرد.

۱. ابتدا کتابخانه‌ها را وارد کنید:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

۲. داده‌ها را همانطور که در جدول XOR قبلی نشان داده شده است آماده کنید:

```
X = np.array([ [0, 1], [1, 0], [1, 1],[0, 0] ])
y = np.array([ [1], [1], [0], [0]])
```

۳. تعداد گره‌ها را در هر لایه تعیین کنید.

```
num_input = 2
num_hidden = 5
num_output = 1
```

٤. وزن‌ها و سویشها را به صورت تصادفی مقداردهی اولیه کنید. ابتدا، وزن‌های لایه پنهان را مقداردهی اولیه می‌کنیم:

```
Wxh = np.random.randn(num_input,num_hidden)
bh = np.zeros((1,num_hidden))
```

٥. حالا اوزان بین لایه پنهان به لایه خروجی را مقداردهی اولیه می‌کنیم:

```
Why = np.random.randn (num_hidden,num_output)
by = np.zeros((1,num_output))
```

٦. تابع فعالسازی سیگموئید را تعریف کنید:

```
def sigmoid(z):
    return 1 / (1+np.exp(-z))
```

٧. مشتق تابع سیگموئید را تعریف کنید:

```
def sigmoid_derivative(z):
     return np.exp(-z)/((1+np.exp(-z))**2)
```

٨. انتشار رو به جلو را تعریف کنید:

```
def forward_prop(x,Wxh,Why):
    z1 = np.dot(x,Wxh) + bh
    a1 = sigmoid(z1)
    z2 = np.dot(a1,Why) + by
    y_hat = sigmoid(z2)

    return z1,a1,z2,y_hat
```

۹. انتشار پسرو را تعریف کنید:

```
def backword_prop(y_hat, z1, a1, z2):
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))
    dJ_dWhy = np.dot(a1.T, delta2)
    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)
    dJ_dWxh = np.dot(x.T, delta1)

    return dJ_dWxh, dJ_dWhy
```

۱۰. تابع هزینه را تعریف کنید:

```
def cost_function(y, y_hat):
    J = 0.5*sum((y-y_hat)**2)

    return J
```

۱۱. نرخ یادگیری و تعداد تکرارهای آموزشی را تنظیم کنید:

```
alpha = 0.01
num_iterations = 5000
```

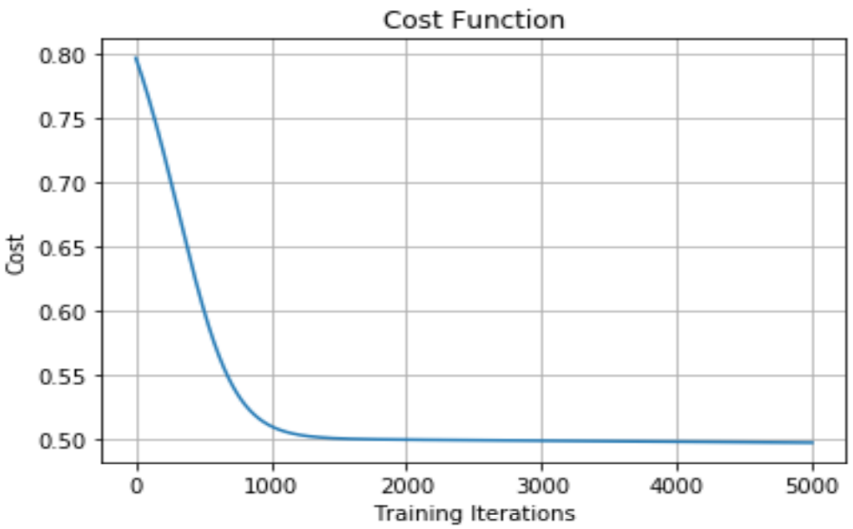
۱۲. حال اجازه دهید آموزش شبکه را با کد زیر شروع کنیم:

```
cost =[]

for i in range(num_iterations):
    z1,a1,z2,y_hat = forward_prop(X,Wxh,Why)
    dJ_dWxh, dJ_dWhy = backword_prop(y_hat, z1, a1, z2)

    #update weights
    Wxh = Wxh -alpha * dJ_dWxh
    Why = Why -alpha * dJ_dWhy

    #compute cost
    c = cost_function(y, y_hat)

    cost.append(c)
```

۱۳. تابع هزینه را ترسیم کنید:

```
plt.grid()
plt.plot(range(num_iterations),cost)

plt.title('Cost Function')
plt.xlabel('Training Iterations')
plt.ylabel('Cost')
```

همانطور که در نمودار زیر مشاهده می‌کنید، هزینه در تکرارهای آموزشی کاهش می‌یابد:



شکل ۷.۱۴: تابع هزینه

بنابراین، ما درک کلی از شبکه‌های عصبی مصنوعی و نحوه یادگیری آنها داریم.

شبکه‌های عصبی بازگشتی (RNN)

خورشید از سمت _____ *طلوع می‌کند.* The sun rises in the ____.

اگر از ما خواسته شود که عبارت خالی در جمله قبل را پیش‌بینی کنیم، احتمالا می‌گوییم شرق. چرا ما پیش‌بینی می‌کنیم که کلمه شرق در اینجا کلمه مناسبی خواهد بود؟ زیرا ما کل جمله را خواندیم، متن را درک کردیم و پیش‌بینی کردیم که کلمه شرق کلمه مناسبی برای تکمیل جمله خواهد بود.

اگر از یک شبکه عصبی پیشخور[۱] (شبکه ای که در بخش قبل یاد گرفتیم) برای پیش‌بینی جای خالی استفاده کنیم، کلمه مناسب را پیش‌بینی نمی‌کند. این به این دلیل است که در شبکه‌های پیشخور، هر ورودی مستقل از ورودی دیگر است و آنها پیش‌بینی‌ها را فقط بر اساس ورودی فعلی ایجاد می‌کنند، یعنی و ورودی قبلی را به خاطر نمی‌آورند.

بنابراین، ورودی شبکه، فقط کلمه قبل از جای خالی، یعنی واژه the است. با استفاده از این واژه یعنی تنها ورودی

[۱] Feedforward Neural Network

ممکن، شبکه ما نمی‌تواند کلمه صحیح بعدی را پیش‌بینی کند، زیرا زمینه یا بافتار[۱] جمله را نمی‌داند، به این معنی که مجموعه کلمات قبلی را نمی‌داند تا متن جمله را درک کند و کلمه بعدی مناسب را پیش‌بینی کند.

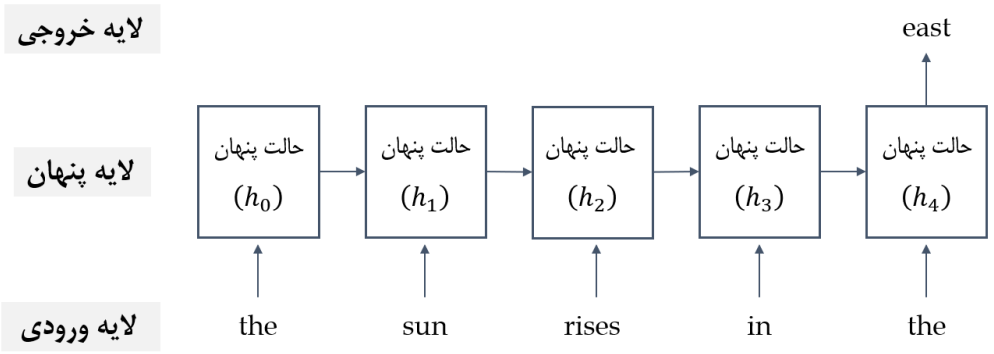
این مثال نشاندهنده جایی است که ما از **شبکه‌های عصبی بازگشتی[۲] (RNNs)** استفاده می‌کنیم. آنها خروجی را نه تنها بر اساس ورودی فعلی، بلکه بر اساس حالت پنهان قبلی نیز پیش‌بینی می‌کنند. چرا آنها باید خروجی را بر اساس ورودی فعلی و حالت پنهان قبلی پیش‌بینی کنند؟ چرا آنها نمی‌توانند فقط از ورودی فعلی و ورودی قبلی استفاده کنند؟

این بدان دلیل است که ورودی قبلی فقط اطلاعات مربوط به کلمه قبلی را ذخیره می‌کند، در حالی که حالت پنهان قبلی، اطلاعات متنی مربوط به تمام کلمات جمله‌ای را که شبکه تاکنون دیده، ضبط می‌کند. اساسا حالت پنهان قبلی مانند حافظه عمل می‌کند و زمینه یا بافتار جمله را گرفته و نگهداری می‌کند. با داشتن این زمینه (بافتار) و نیز ورودی فعلی، می‌توانیم کلمه مربوطه را پیش‌بینی کنیم.

به عنوان مثال، بیایید همان جمله را در نظر بگیریم، .The sun rises in the ____ همانطور که در شکل زیر نشان داده شده است، ابتدا کلمه the را به عنوان ورودی ارسال می‌کنیم، و سپس کلمه بعدی، sun، را به عنوان ورودی ارسال می‌کنیم؛ اما همراه با این عبور، حالت پنهان قبلی یعنی h. را نیز عبور می‌دهیم. بنابراین، هر بار که کلمه ورودی جدید را ارسال می‌کنیم، حالت پنهان قبلی را نیز به عنوان یک ورودی مکمل ارسال می‌کنیم.

در مرحله آخر، ما کلمه the و همچنین حالت پنهان قبلی h_3 را منتقل می‌کنیم که اطلاعات متنی در مورد توالی کلماتی را که شبکه تاکنون دیده، ضبط می‌کند. بنابراین، h_3 به عنوان حافظه عمل کرده و اطلاعات مربوط به تمام کلمات قبلی را که شبکه دیده است ذخیره می‌کند. با حضور h_3 و کلمه ورودی فعلی (the)، می‌توانیم کلمه بعدی مربوطه بعدی را پیش‌بینی کنیم:

[۱] Context
[۲] Recurrent Neural Networks (RNNs)



شکل ۷.۱۵: شبکه عصبی بازگشتی

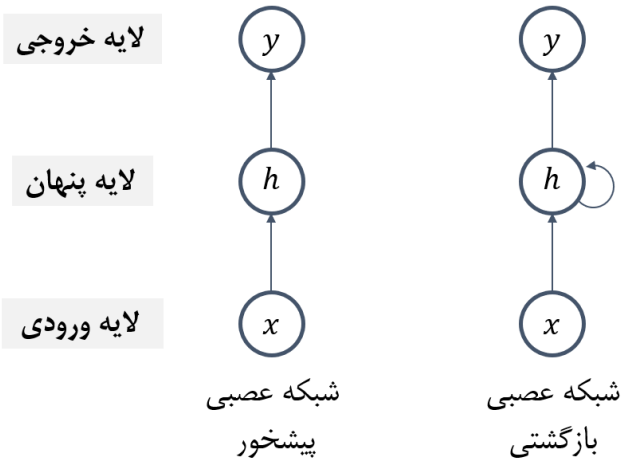
به طور خلاصه، یک **RNN** از حالت پنهان قبلی به عنوان حافظه استفاده می‌کند تا اطلاعات متنی یا زمینه‌ای (ورودی‌ها) را که شبکه تاکنون دیده ضبط و ذخیره کند.

RNNها به طور گسترده برای موارد استفاده که شامل داده‌های متوالی مانند سری‌های زمانی، متن، صدا، گفتار، ویدئو، آب و هوا و موارد دیگر هستند، استفاده می‌شوند. آنها در کارهای مختلف **پردازش زبان طبیعی**[۱] (**NLP**) مانند **ترجمه زبان**، **تجزیه و تحلیل احساسات**، **تولید متن** و غیره بسیار مورد استفاده قرار گرفته اند.

تفاوت بین شبکه‌های پیشخور و RNNها

مقایسه‌ای بین یک **RNN** و یک شبکه پیشخور در شکل ۷.۱۶ آمده است.

[۱] Natural Language Processing (NLP)



شکل ۷.۱۶: تفاوت بین شبکه پیشخور و شبکه عصبی بازگشتی

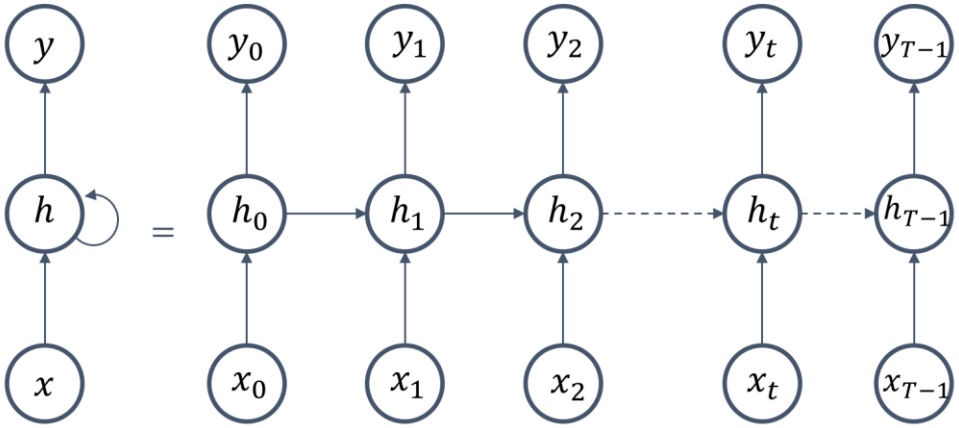
همانطور که در نمودار قبلی مشاهده می‌کنید، **RNN** حاوی یک اتصال حلقه‌ای در لایه پنهان است، که بدین معنا است که ما از حالت پنهان قبلی (بعنوان مکمل) همراه با ورودی جدید برای پیش‌بینی خروجی استفاده می‌کنیم.

هنوز گیج هستید؟ بیایید به نسخه باز شدهٔ[۱] زیر از **RNN** نگاه کنیم. اما صبر کنید. نسخه باز شدهٔ **RNN** چیست؟

این بدان معناست که ما درون شبکه را برای یک توالی کامل، از حالت فشرده، باز می‌کنیم. بیایید فرض کنیم که ما یک جمله ورودی با T کلمه داریم؛ پس ما ۰ تا $T\text{-}1$ لایه خواهیم داشت، یک لایه برای هر کلمه، همانطور که در شکل ۷.۱۷ نشان داده شده است.

همانطور که در شکل ۷.۱۷ مشاهده می‌کنید، در مرحله زمانی $t=1$، خروجی $y_۱$ بر اساس ورودی فعلی $x_۱$ و حالت پنهان قبلی $h_.$ پیش‌بینی می‌شود. به طور مشابه، در مرحله زمانی $t=2$، مقدار $y_۲$ با استفاده از ورودی فعلی $x_۲$ و حالت پنهان قبلی $h_۱$ پیش‌بینی می‌شود. این شیوه‌ای که **RNN** کار می‌کند. در هر لایه، ورودی فعلی و حالت پنهان قبلی برای پیش‌بینی خروجی، لازم است.

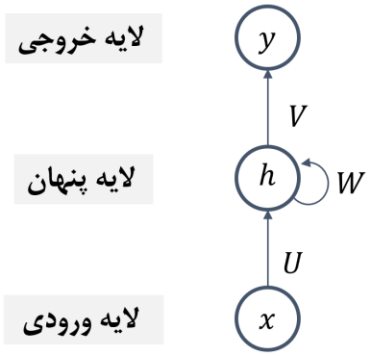
[۱] Unrolled Version



شکل ۷.۱۷: **RNN** بازشده (برای دیدن اجزاء شبکه)

انتشار رو به جلو در RNNها

بیایید ببینیم که چگونه یک **RNN** از انتشار رو به جلو برای پیش‌بینی خروجی استفاده می‌کند. اما قبل از اینکه مستقیما وارد شویم، بیایید با نمادها آشنا شویم:



شکل ۷.۱۸: انتشار رو به جلو در **RNN**

شکل قبلی موارد زیر را نشان می‌دهد:

۱. U نشان دهنده ماتریس وزن لایه ورودی به لایه پنهان است
۲. W نشان دهنده ماتریس وزن لایه پنهان به لایه پنهان است

۳. V نشان دهنده ماتریس وزن لایه پنهان به لایه خروجی است

حالت پنهان h در یک مرحله زمانی t را می‌توان به صورت زیر محاسبه کرد:

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

یعنی حالت پنهان در یک مرحله زمانی t به‌صورت زیر بدست می‌آید:

خروجی = *tanh([input to hidden layer weight x input] +*
[hidden to hidden layer weight x previous hidden state])

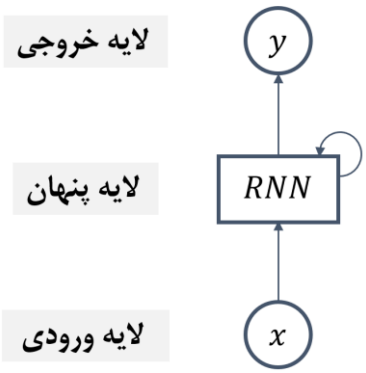
و خروجی در یک مرحله زمانی t را می‌توان به شرح زیر محاسبه کرد:

$$\hat{y}_t = \text{softmax}(Vh_t)$$

یعنی خروجی در یک مرحله زمانی t با محاسبه زیر حاصل می‌شود:

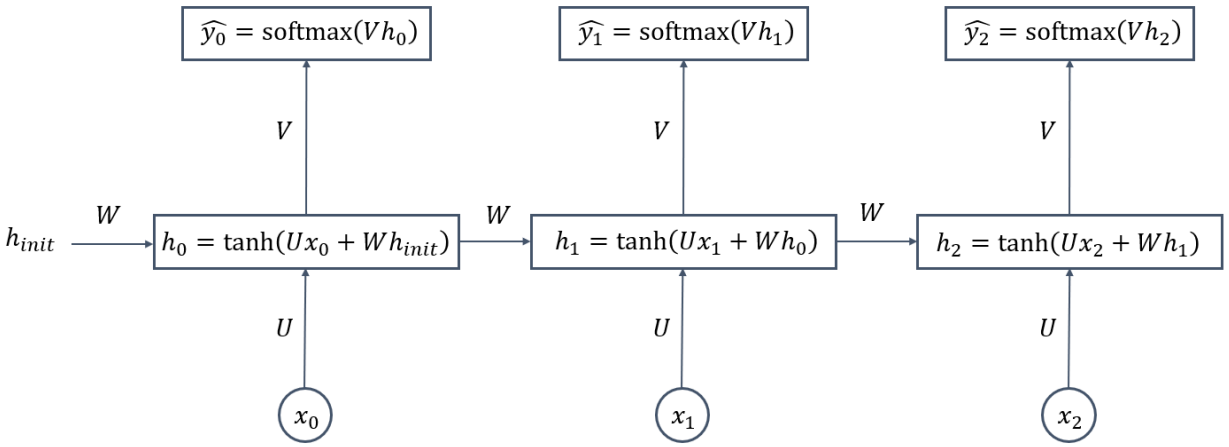
خروجی = *softmax (hidden to output layer weight x hidden state at a time t).*

ما همچنین می‌توانیم **RNN**ها را همانند شکل زیر نشان دهیم. همانطور که می‌بینید، لایه پنهان، بصورت یک بلوک **RNN** نشان داده می‌شود، که به این معنی است که شبکه ما یک **RNN** است و از حالت‌های پنهان قبلی در پیش‌بینی خروجی استفاده می‌شود:



شکل ۷.۱۹: انتشار رو به جلو در یک **RNN**

شکل ۷.۲۰ نشان می‌دهد که انتشار رو به جلو در یک نسخه بازشده از **RNN** چگونه کار می‌کند:



شکل ۷.۲۰: نسخه بازشده – انتشار رو به جلو در **RNN**

ما حالت پنهان اولیه یعنی h_{init} را با مقادیر تصادفی، مقداردهی اولیه می‌کنیم. همانطور که در شکل قبلی دیدید، خروجی یعنی $\hat{y}_.$، بر اساس ورودی فعلی یعنی $x_.$ و حالت پنهان قبلی، که یک حالت پنهان اولیه است، یعنی h_{init}، با استفاده از فرمول زیر محاسبه می‌شود.

$$h_. = \tanh(Ux_. + Wh_{init})$$

$$\hat{y}_. = \text{softmax}(Vh_.)$$

به طور مشابه، به نحوه محاسبه خروجی $\hat{y}_۱$ نگاه کنید. مجددا مدل ما برای محاسبه خود، ورودی فعلی، یعنی $x_۱$ و حالت پنهان قبلی، یعنی $h_.$ را می‌گیرد:

$$h_۱ = \tanh(Ux_۱ + Wh_.)$$

$$\hat{y}_۱ = \text{softmax}(Vh_۱)$$

بنابراین، در انتشار رو به جلو به منظور پیش‌بینی خروجی، **RNN** از ورودی فعلی وحالت پنهان قبلی استفاده می‌کند.

پس‌انتشار در طول زمان

ما به تازگی یاد گرفتیم که انتشار رو به جلو در **RNN** چگونه کار می‌کند و چگونه خروجی را پیش‌بینی می‌کند. اکنون، ما ضرر و ریان، یعنی L را در هر مرحله زمانی t، محاسبه می‌کنیم تا تعیین کنیم که **RNN** چقدر خروجی را خوب پیش‌بینی کرده است. ما از **تابع زیان آنتروپی متقاطع**[۱] به عنوان تابع مطلوب برای مدل کردن زیان شبکه استفاده می‌کنیم.

زیان L در یک مرحله زمانی t را می‌توان به شرح زیر داد:

$$L_t = -y_t \log(\hat{y}_t)$$

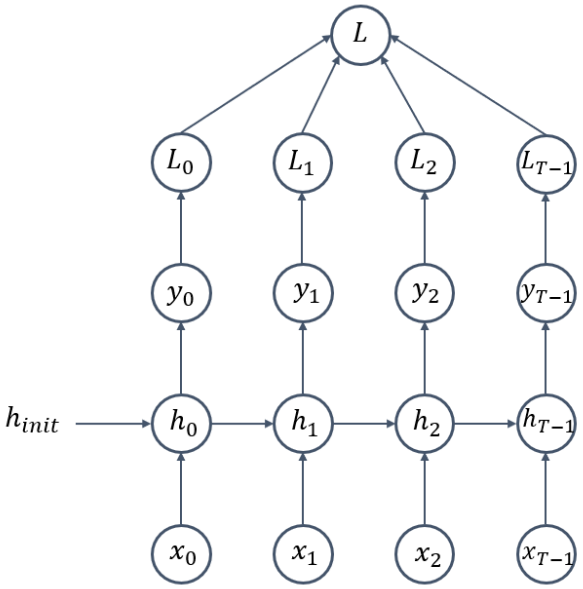
در اینجا، y_t خروجی واقعی و $\hat{y}$ خروجی پیش‌بینی شده در یک مرحله زمانی t است.

ضرر نهایی، مجموع ضرر در تمام مراحل زمانی است. فرض کنید $T\text{-}1$ لایه داشته باشیم، پس، ضرر نهایی را می‌توان به شرح زیر محاسبه کرد:

$$L = \sum_{j=0}^{T-1} L_j$$

شکل ۷.۲۱ نشان می‌دهد که ضرر نهایی با مجموع ضرر در همه گامهای زمانی به دست می‌آید.

[۱] Cross-Entropy Loss Function



شکل ۷.۲۱: پس–انتشار در یک **RNN**

ما ضرر را محاسبه کردیم، اکنون هدف ما به حداقل رساندن ضرر است. چگونه می‌توانیم ضرر را به حداقل برسانیم؟ ما می‌توانیم با یافتن وزن‌های بهینه **RNN** زیان را به حداقل برسانیم. همانطور که یاد گرفتیم، ما سه وزن در **RNN** داریم. U: اوزان لایه ورودی به پنهان؛ W: اوزان لایه پنهان به پنهان؛ و V: اوزان لایه پنهان به خروجی.

ما باید مقادیر بهینه را برای هر سه وزن پیدا کنیم تا ضرر را به حداقل برسانیم. ما می‌توانیم از الگوریتم کاهش گرادیان مورد علاقه خود برای یافتن وزن‌های بهینه استفاده کنیم. ما با محاسبه نشیب یا گرادیان تابع ضرر با توجه به تمام وزن‌ها شروع می‌کنیم. سپس، وزن‌ها را طبق قاعده به‌روزرسانی اوزان به شرح زیر به‌روز می‌کنیم:

$$V = V - \alpha \frac{\partial L}{\partial V}$$

$$W = W - \alpha \frac{\partial L}{\partial W}$$

$$U = U - \alpha \frac{\partial L}{\partial U}$$

با این حال، ما با **RNN** مشکل داریم. محاسبه گرادیان شامل محاسبه گرادیان با توجه به تابع فعالسازی است. وقتی گرادیان را با توجه به تابع سیگموئید یا tanh محاسبه می‌کنیم، گرادیان بسیار کوچک می‌شود. هنگامی‌که شبکه را در طی چندین مرحله زمانی بیشتر پس–انتشار می‌کنیم و گرادیان‌ها را ضرب می‌کنیم، گرادیان‌ها کوچک و کوچکتر می‌شوند. به این مسئله **ناپدید شدن** یا **امحاء گرادیان** می‌گویند.

از آنجایی که گرادیان با گذشت زمان ناپدید می‌شود، ما نمی‌توانیم اطلاعاتی در مورد وابستگی‌های طولانی مدت بیاموزیم، یعنی **RNN**ها نمی‌توانند اطلاعات را برای مدت طولانی در حافظه حفظ کنند. مسئله امحاء گرادیان نه تنها در **RNN**ها بلکه در سایر شبکه‌های عمیق نیز رخ می‌دهد: شبکه‌هایی که در آن لایه‌های پنهان زیادی داریم و زمانی که از توابع سیگموئید/tanh استفاده می‌کنیم.

یکی از راه حل‌ها برای جلوگیری از مشکل ناپدید شدن گرادیان، استفاده از ReLU به عنوان یک تابع فعالسازی است. با این حال، ما یک نوع از **RNN** به نام **حافظه کوتاه-مدت دیرپا** (**LSTM**) داریم که می‌تواند مشکل امحاء گرادیان را به طور موثر حل کند. در بخش آینده خواهیم دید که این شبکه چگونه کار می‌کند.

LSTM: راه گریز از مشکل امحاءِ نشیب

در حین بحثِ پس–انتشار برای یک **RNN**، ما در مورد مشکلی به نام **گرادیان‌های ناپدید شده** گفتیم. به دلیل مشکل امحاء گرادیان، ما نمی‌توانیم شبکه را به درستی آموزش دهیم و این باعث می‌شود که **RNN** نتوانند دنباله‌های طولانی را در حافظه، حفظ نکند. برای درک منظور ما از این مشکل، بیایید یک جمله کوچک را در نظر بگیریم:

آسمان ... است. The sky is __.

یک **RNN** می‌تواند به راحتی بر اساس اطلاعاتی که دیده است، جای خالی را به صورت **آبی** پیش‌بینی کند، اما نمی‌تواند وابستگی‌های طولانی مدت را پوشش دهد. خب این یعنی چه؟ بیایید جمله زیر را در نظر بگیریم تا مشکل

[1] Vanishing Gradient Problem

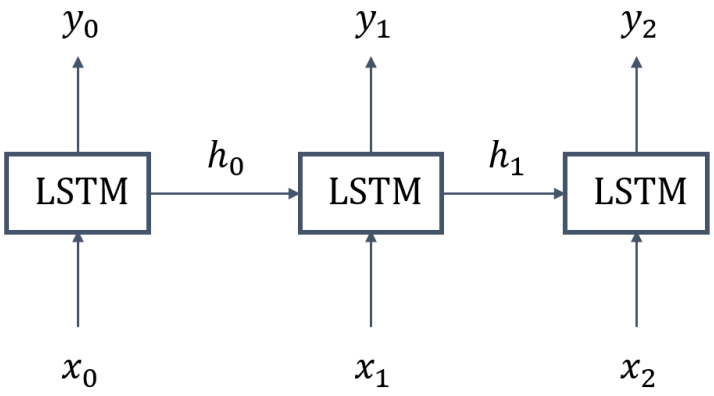
را بهتر درک کنیم:

آرچی ۱۳ سال در چین زندگی کرد. او عاشق گوش دادن به موسیقی خوب است. او از طرفداران طنز است. او به زبان ..... مسلط است.

حال، اگر از ما خواسته شود که کلمه گمشده در جمله قبل را پیش‌بینی کنیم، آن را **چینی** پیش‌بینی می‌کنیم، اما چگونه آن را پیش‌بینی می‌کنیم؟ ما به سادگی جملات قبلی را به یاد آوردیم و فهمیدیم که آرچی ۱۳ سال در چین زندگی کرده است. این ما را به این نتیجه رساند که آرچی ممکن است به زبان چینی مسلط باشد. از سوی دیگر، یک **RNN** نمی‌تواند تمام این اطلاعات را در حافظه خود نگه دارد تا بگوید آرچی به زبان چینی مسلط است.

به دلیل مشکل امحاء گرادیان، نمی‌تواند اطلاعات را برای مدت طولانی در حافظه خود به خاطر بسپارد. یعنی وقتی دنباله ورودی، طولانی باشد، حافظه **RNN** (حالت پنهان) نمی‌تواند تمام اطلاعات را در خود جای دهد. برای کاهش این مشکل، از سلول **حافظه کوتاه-مدت دیرپا** (**LSTM**) استفاده می‌کنیم.

حافظه کوتاه-مدت دیرپا (**LSTM**) گونه ای از **RNN** است که مشکل ناپدیدی نشیب (امحاءگرادیان) را حل می‌کند و اطلاعات را تا زمانی که مورد نیاز باشد در حافظه نگه می‌دارد. اساسا، سلول‌های **RNN** با سلول‌های **LSTM** در واحدهای پنهان جایگزین می‌شوند، همانطور که در شکل ۷.۲۲ نشان داده شده است:



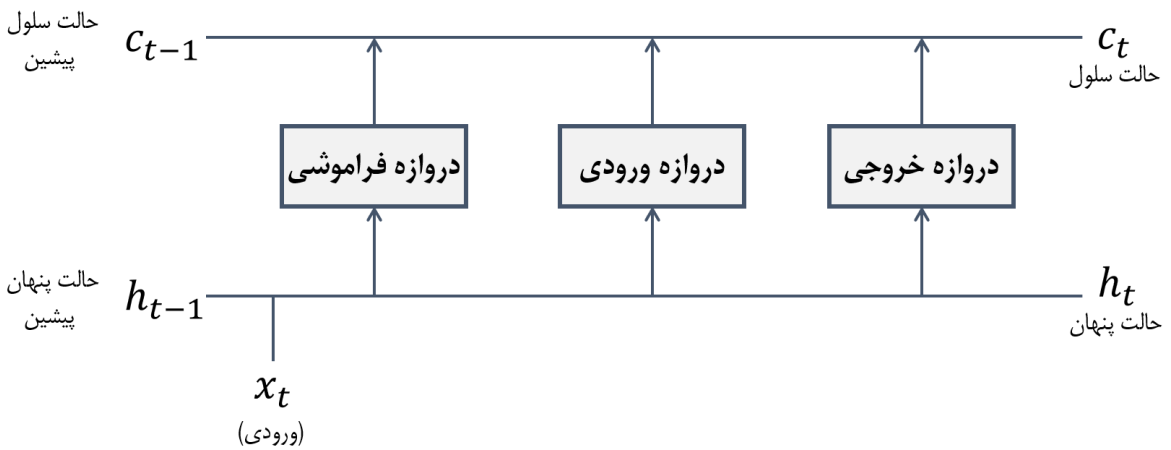
شکل ۷.۲۲: شبکه حافظه کوتاه مدت -دیرپا (**LSTM**)

در بخش بعدی نحوه عملکرد سلول‌های **LSTM** را خواهیم فهمید.

آشنایی با سلول حافظه کوتاه‌ـمدت دیرپا (LSTM)

چه چیزی سلول‌های **LSTM** را بسیار خاص می‌کند؟ چگونه سلول‌های **LSTM** به وابستگی طولانی مدت دست می‌یابند؟ چگونه می‌داند چه اطلاعاتی را نگه دارد و چه اطلاعاتی را از حافظه دور بیندازد؟

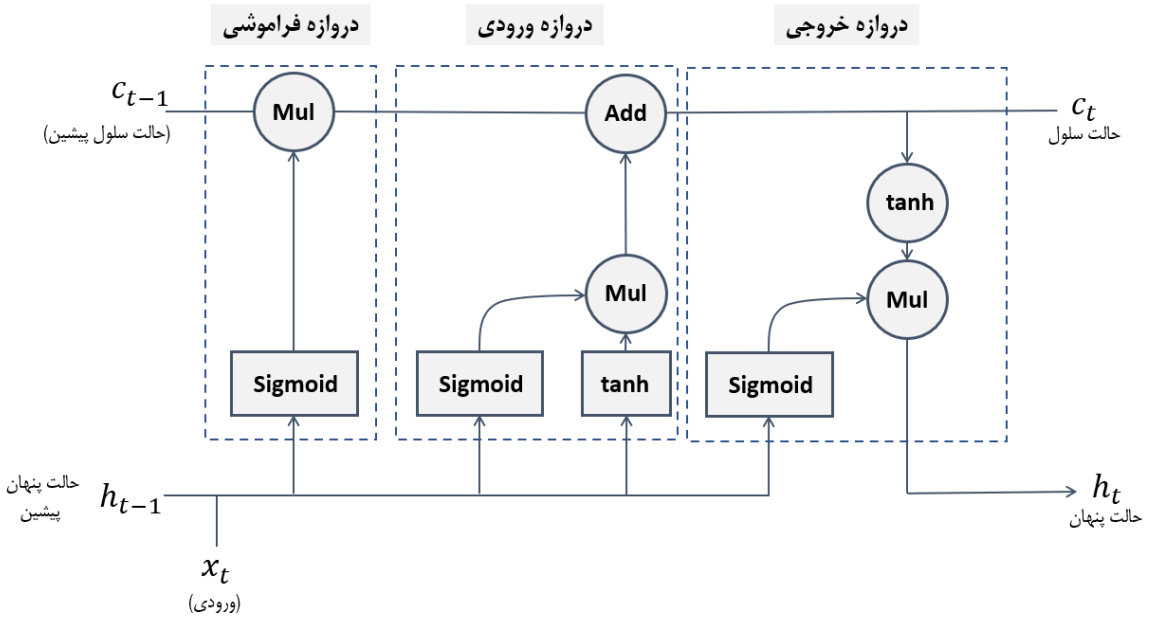
همه اینها توسط سازه‌های خاصی به نام **دروازه‌ها**[1] به دست می‌آید. همانطور که در نمودار زیر نشان داده شده است، یک سلول **LSTM** معمولی از سه گیت ویژه به نام‌های دروازه ورودی، دروازه خروجی و دروازه فراموشی تشکیل شده است:



شکل ۷.۲۳: دروازه‌های حافظه کوتاه–مدت دیرپا (**LSTM**)

این سه دروازه، وظیفه تصمیم‌گیری در خصوص برخورد با اطلاعات را دارند: چه اطلاعاتی را به حافظه اضافه کنند، چه اطلاعاتی را بعنوان خروجی بیرون دهند و چه اطلاعاتی را فراموش کنند. با استفاده از این دروازه‌ها، یک سلول **LSTM** به طور موثر اطلاعات را فقط تا زمانی که لازم باشد در حافظه نگه می‌دارد. شکل ۷.۲۴ یک سلول **LSTM** معمولی را نشان می‌دهد:

[1] Gates



شکل ۷.۲۴: سلول **LSTM**

اگر به سلول **LSTM** نگاه کنید، خط افقی بالایی را **حالت سلول**[۱] می‌نامند. اینجا، جایی است که اطلاعات جریان می‌یابد. اطلاعات درون حالت سلول به طور مداوم توسط گیت‌های **LSTM** به‌روز می‌شود. اکنون عملکرد این دروازه‌ها را خواهیم دید:

دروازه فراموشی: دروازه فراموشی مسئول تصمیم‌گیری در مورد اطلاعاتی است که نباید در «حالت سلول» باشد. به گزاره زیر نگاه کنید:

هری خواننده خوبی است. او در نیویورک زندگی می‌کند. زین نیز خواننده خوبی است.

به محض اینکه شروع به صحبت در مورد زین می‌کنیم، شبکه متوجه می‌شود که موضوع از هری به زین تغییر کرده است و اطلاعات مربوط به هری دیگر مورد نیاز نیست. اکنون، دروازه فراموشی اطلاعات مربوط به هری را از حالت سلول حذف/فراموش می‌کند.

[۱] Cell State

دروازه ورودی: گیت ورودی مسئول تصمیم‌گیری در مورد اطلاعاتی است که باید در حافظه ذخیره شود. بیایید همین مثال را در نظر بگیریم:

هری خواننده خوبی است. او در نیویورک زندگی می‌کند. زین نیز خواننده خوبی است.

بنابراین، پس از اینکه دروازه فراموشی اطلاعات را از «حالت سلول» حذف کرد، گیت ورودی تصمیم می‌گیرد که چه اطلاعاتی باید در حافظه باشد. در اینجا، از آنجایی که اطلاعات مربوط به هری توسط گیت فراموشی از حالت سلول حذف می‌شود، گیت ورودی تصمیم می‌گیرد وضعیت سلول را با اطلاعات مربوط به زین به‌روز کند.

دروازه خروجی: گیت خروجی مسئول تصمیم‌گیری در مورد اطلاعاتی است که باید از «سلول حالت» در یک زمان خاص t نشان داده شود. اکنون، جمله زیر را در نظر بگیرید:

اولین آلبوم زین، موفقیت بزرگی داشت. به _____ تبریک می‌گویم.

در اینجا، تبریک صفتی است که برای توصیف یک اسم استفاده می‌شود. لایه خروجی، Zayn (اسم) را پیش‌بینی می‌کند تا جای خالی را پر کند.

بنابراین، با استفاده از **LSTM**، می‌توانیم بر مشکل گرادیان ناپدید شده که در **RNN** با آن مواجه هستیم، غلبه کنیم. در بخش بعدی، الگوریتم جالب دیگری به نام **شبکه عصبی همتابی یا پیچشی** (**CNN**) را یاد خواهیم گرفت.

شبکه عصبی همتابی (CNN) چیست؟

شبکه عصبی همتابی[۱] (**CNN**) که به عنوان **ConvNet** نیز شناخته می‌شود، یکی از پرکاربردترین الگوریتم‌های یادگیری عمیق برای وظایف بینایی کامپیوتر است. فرض کنید ما در حال انجام یک کارِ تشخیصِ تصویر هستیم. تصویر زیر را در نظر بگیرید.

[۱] Convolutional Neural Network (CNN)

برای این واژه، معادلهای دیگری همچون درهم پیچیدن، درهم تافتن، درهم تابیدن، درهم تنیدن و همگشت نیز مطرح است.

ما می‌خواهیم **CNN** ما تشخیص دهد که در تصویر زیر، یک اسب است:

تصویر ۷.۲۵: تصویر حاوی یک اسب

چگونه می‌توانیم این کار را انجام دهیم؟ وقتی تصویر را به یک کامپیوتر تغذیه می‌کنیم، اساسا آن را به ماتریسی از مقادیر پیکسل تبدیل می‌کند. مقادیر پیکسل از ۰ تا ۲۵۵ متغیر است و ابعاد این ماتریس [*عرض تصویر* × *ارتفاع تصویر* × *تعداد کانال*] خواهد بود. یک تصویر به رنگ خاکستری دارای یک کانال، ولی تصاویر رنگی دارای سه کانال **قرمز، سبز و آبی** (**RGB**)[۱] هستند.

فرض کنید ما یک تصویر ورودی رنگی با عرض ۱۱ و ارتفاع ۱۱ داریم، یعنی ۱۱×۱۱ پس ابعاد ماتریس ما [۳×۱۱×۱۱] خواهد بود. همانطور که در نماد [۳×۱۱×۱۱] می‌بینید، مقادیر ۱۱×۱۱ نشان دهنده عرض و ارتفاع تصویر و عدد ۳ نشان دهنده شماره کانال است، زیرا ما یک تصویر رنگی داریم. بنابراین، ما یک ماتریس سه بُعدی خواهیم داشت.

اما تجسم یک ماتریس سه بُعدی دشوار است، بنابراین، به خاطر درک بهتر، بیایید یک تصویر خاکستری را به عنوان ورودی خود در نظر بگیریم. از آنجایی که تصویر با وضعیت خاکستری فقط یک کانال دارد، یک ماتریس دو بعدی دریافت خواهیم کرد.

[۱] Red, Green, and Blue (RGB)

همانطور که در نمودار زیر نشان داده شده است، تصویر خاکستری ورودی به ماتریسی از مقادیر پیکسل از ۰ تا ۲۵۵ تبدیل می‌شود که مقادیر پیکسل نشان دهنده شدت پیکسل‌ها در آن نقطه است:

شکل ۷.۲۶: تصویر ورودی به ماتریس مقادیر پیکسل تبدیل می‌شود.

	مقادیر داده شده در ماتریس ورودی این مثال، فقط مقادیری دلخواه برای درک بهتر مثال هستند.

بسیار خب، اکنون ما یک ماتریس ورودی از مقادیر پیکسل داریم. بعد چه اتفاقی می‌افتد؟ **CNN** چگونه متوجه می‌شود که این تصویر حاوی یک اسب است؟ **CNN** از سه لایه مهم زیر تشکیل شده‌اند:

۱. لایه همتابی یا پیچش
۲. لایه ادغام
۳. لایه کاملا متصل

با کمک این سه لایه، **CNN** تشخیص می‌دهد که تصویر حاوی یک اسب است. اکنون هر یک از این لایه‌ها را با جزئیات بررسی خواهیم کرد.

لایه‌های پیچشی

لایه پیچشی یا همتابی اولین و لایه اصلی **CNN** است. این لایه یکی از بلوک‌های سازنده **CNN** است و برای استخراج ویژگی‌های مهم از تصویر استفاده می‌شود.

ما تصویری از یک اسب داریم. به نظر شما چه ویژگی‌هایی وجود دارد که به ما کمک می‌کند تا بفهمیم این تصویر اسب است؟ می‌توان گفت ساختار بدن، صورت، پاها، دم و غیره. اما **CNN** چگونه این ویژگی‌ها را درک می‌کند؟ اینجاست که ما از یک عملیات همتابی استفاده می‌کنیم که تمام ویژگی‌های مهم را از تصویر (که مشخصه اسب است) استخراج می‌کند. بنابراین، عملیات همتابی به ما کمک می‌کند تا بفهمیم تصویر در مورد چیست.

بسیار خوب، این عملیات همتابی یا کانولوشن دقیقا چیست؟ چگونه انجام می‌شود؟ چگونه ویژگی‌های مهم را استخراج می‌کند؟ بیایید به این موضوع با جزئیات نگاه کنیم.

همانطور که می‌دانیم، هر تصویر ورودی با ماتریسی از مقادیر پیکسل نشان داده می‌شود. به غیر از ماتریس ورودی، ماتریس دیگری به نام **ماتریس فیلتر** نیز داریم.

ماتریس فیلتر به عنوان **هسته**[۱] یا به سادگی یک **فیلتر** نیز شناخته می‌شود، در شکل ۷.۲۷ آمده است.

0	1
1	0

فیلتر

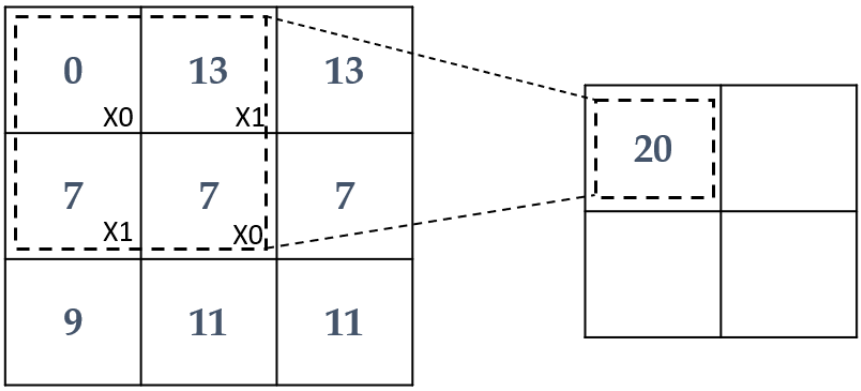
0	13	13
7	7	7
9	11	11

ماتریس ورودی

شکل ۷.۲۷: ماتریس ورودی و فیلتر

[۱] Kernel

ماتریس فیلتر را می‌گیریم، و آنرا یک پیکسل بر روی ماتریس ورودی می‌کشانیم (یا می‌لغزانیم)، سپس ضرب عنصر در عنصر[1] را انجام داده، و نتایج را جمع می‌کنیم که منجر به تولید یک عدد واحد می‌شود. این بسیار گیج کننده است، اینطور نیست؟ بیایید با کمک نمودار زیر این را بهتر درک کنیم:



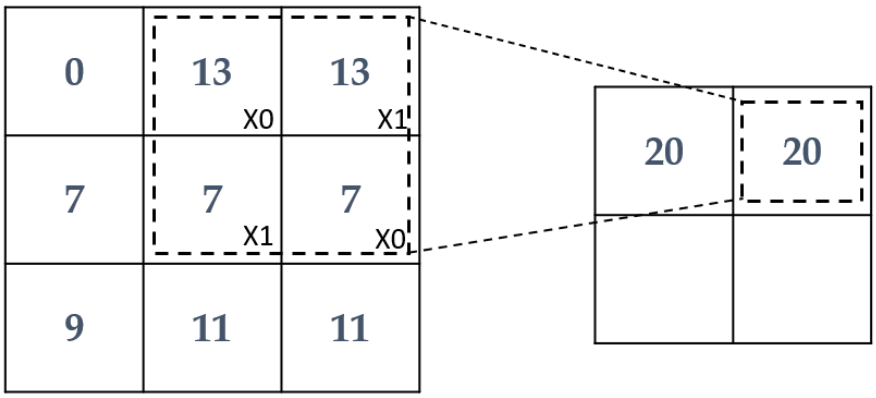
شکل ۷.۲۸: عملیات کانولوشن یا در هم‌تافتن (همتابی)

همانطور که در نمودار قبلی مشاهده کردید، ماتریس فیلتر را گرفتیم و آن را بر روی ماتریس ورودی قرار دادیم، ضرب آرایه در آرایه را انجام دادیم، نتایج آنها را جمع کرده و یک عدد واحد را تولید کردیم. نتیجه این فرایند، به شرح زیر نشان داده شده است:

$$(۰ * ۰) + (۱۳ * ۱) + (۷ * ۱) + (۷ * ۰) = ۲۰$$

اکنون، فیلتر را یک پیکسل دیگر روی ماتریس ورودی می‌لغزانیم؛ یعنی با یک پیکسل جلو رفتن، همان کار را دوباره انجام می‌دهیم. این وضعیت در شکل ۷.۲۹ نشان داده شده است:

[1] Element-Wise Multiplication

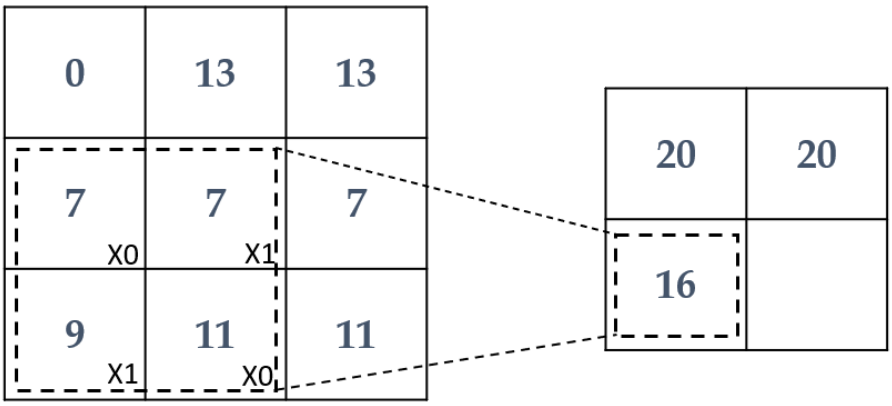


شکل ۷.۲۹: عملیات کانولوشن یا درهم‌تافتن (همتابی)

نتیجه این محاسبه بشرح زیر است:

$$(۱۳ * ۰) + (۱۳ * ۱) + (۷ * ۱) + (۷ * ۰) = ۲۰$$

دوباره، ماتریس فیلتر را یک پیکسل جلو می‌بریم (می‌لغزانیم) و همان عملیات را انجام می‌دهیم، همانگونه که در شکل ۷.۳۰ نشان داده شده است:

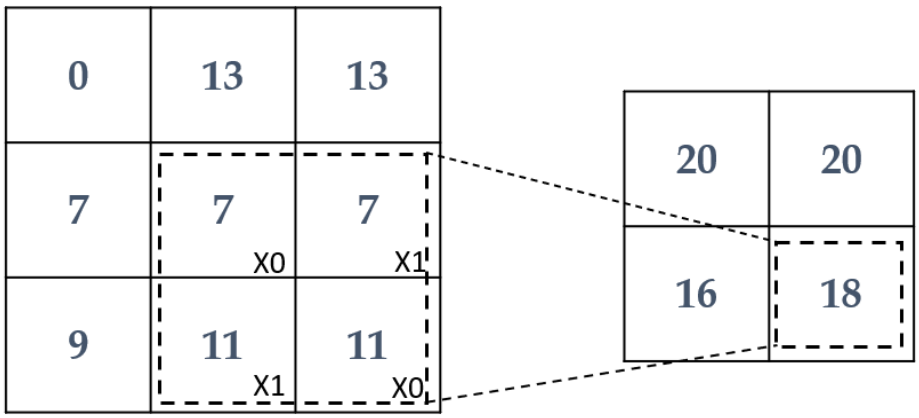


شکل ۷.۳۰: عملیات کانولوشن (همتابی)

نتیجه این محاسبه بشرح زیر است:

$$(۷ * ۰) + (۷ * ۱) + (۹ * ۱) + (۱۱ * ۰) = ۱۶$$

اکنون، دوباره، ماتریس فیلتر را یک پیکسل روی ماتریس ورودی جلو می‌بریم و همان عملیات را انجام می‌دهیم همانطور که در شکل ۷.۳۱ نشان داده شده است:



شکل ۷.۳۱: عملیات کانولوشن (همتابی)

که نتیجه می‌شود:

$$(۷ * ۰) + (۷ * ۱) + (۱۱ * ۱) + (۱۱ * ۰) = ۱۸$$

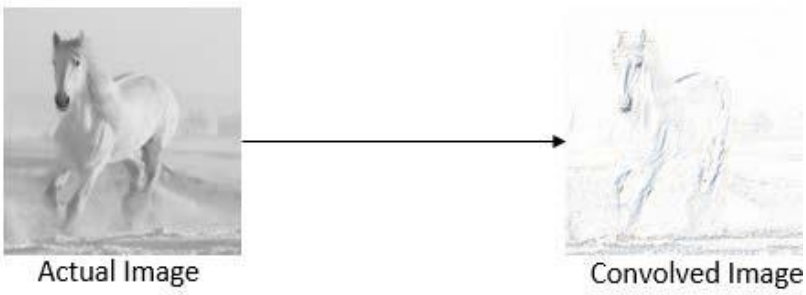
خب. ما داریم چه کار می‌کنیم؟ ما اساسا یک ماتریس فیلتر را بر روی کل ماتریس ورودی می‌کشیم و هر بار یک پیکسل آنرا به جلو می‌لغزانیم، سپس ضرب عنصر در عنصر را انجام داده و نتایج آنها را جمع می‌کند، که ماتریس جدیدی به نام **نقشه ویژگی**[۱] یا **نقشه فعالسازی**[۲] ایجاد می‌کند. به این کار **عملیات همتابی یا پیچش**[۳] می‌گویند.

همانطور که یاد گرفتیم، عملیات کانولوشن برای استخراج ویژگی‌ها استفاده می‌شود و ماتریس جدید، یعنی نقشه‌های ویژگی، ویژگی‌های استخراج شده را نشان می‌دهد. اگر نقشه‌های ویژگی را ترسیم کنیم، می‌توانیم ویژگی‌های استخراج شده توسط عملیات همتابی را ببینیم.

شکل ۷.۳۲ تصویر واقعی (تصویر ورودی) و تصویر همتابی (نقشه ویژگی) را نشان می‌دهد. می‌توانیم ببینیم که فیلتر

[۱] Feature Map
[۲] Activation Map
[۳] Convolution Operation

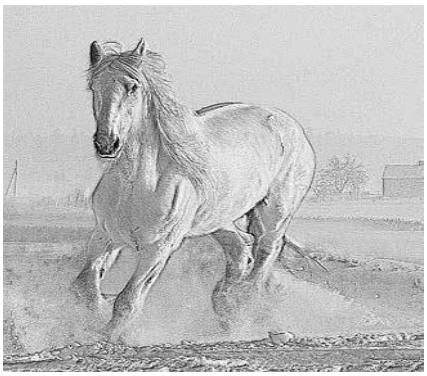
ما لبه‌های تصویر واقعی را به عنوان یک ویژگی شناسایی کرده است:



از فیلترهای مختلفی برای استخراج ویژگی‌های مختلف از تصویر استفاده می‌شود. به عنوان مثال، اگر از یک فیلتر شفاف‌ساز[۱] زیر استفاده کنیم

$$\begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

تصویر ما را شفاف می‌کند که در شکل زیر نشان داده شده است:

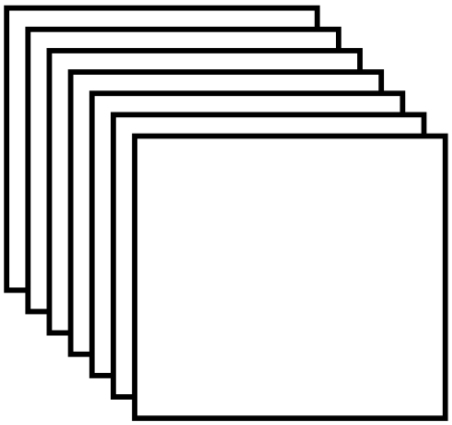


شکل ۷.۳۳: تصویر شفاف شده

بنابراین، ما یاد گرفتیم که با فیلترها می‌توانیم ویژگی‌های مهمی را از تصویر با استفاده از عملیات کانولوشن استخراج

[۱] Sharpen Filter

کنیم. بنابراین، به جای استفاده از یک فیلتر، می‌توانیم از چندین فیلتر برای استخراج ویژگی‌های مختلف از تصویر و تولید چندین نقشه ویژگی استفاده کنیم. بنابراین، عمق نقشه ویژگی، تعداد فیلترها خواهد بود. اگر از هفت فیلتر برای استخراج ویژگی‌های مختلف از تصویر استفاده کنیم، عمق نقشه ویژگی ما هفت خواهد بود:



نقشه ویژگی با عمق هفت

تصویر ۷.۳۴: نقشه‌های ویژگی

بسیار خوب، ما یاد گرفتیم که فیلترهای مختلف ویژگی‌های مختلفی را از تصویر استخراج می‌کنند. اما سوال اینجاست که چگونه می‌توانیم مقادیر صحیح را برای ماتریس فیلتر تنظیم کنیم تا بتوانیم ویژگی‌های مهم را از تصویر استخراج کنیم؟ نگران نباشید! ما فقط ماتریس فیلتر را به طور تصادفی مقداردهی اولیه می‌کنیم و مقادیر بهینه ماتریس فیلتر، که با آن می‌توانیم ویژگی‌های مهم را از تصاویر استخراج کنیم، از طریق پس–انتشار آموخته می‌شود. با این حال، ما فقط باید اندازه فیلتر و تعداد فیلترهایی را که می‌خواهیم استفاده کنیم مشخص کنیم.

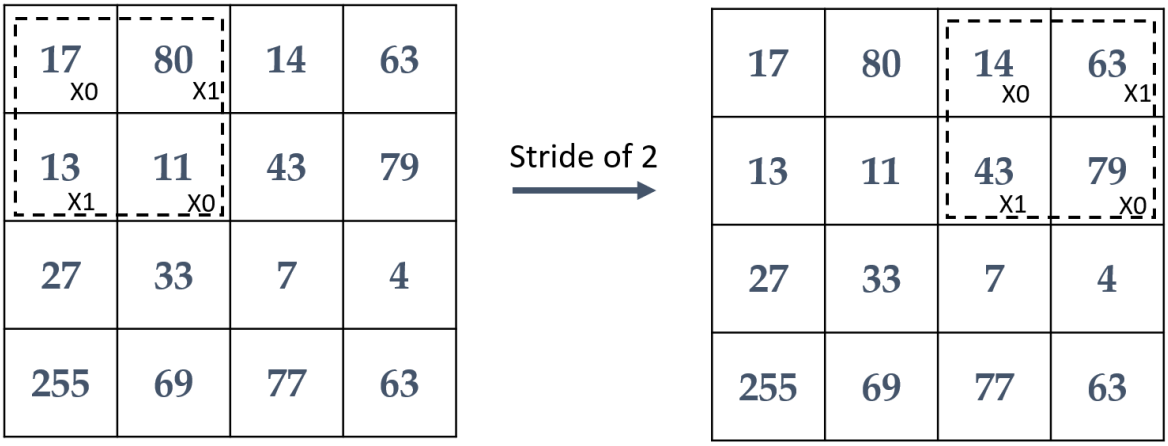
گام پیشروی[1]

ما به تازگی یاد گرفتیم که عملیات همتابی چگونه کار می‌کند. ما با ماتریس فیلتر یک پیکسل روی ماتریس ورودی می‌لغزیم و عملیات همتابی را انجام می‌دهیم. اما لازم نیست فقط به تعداد یک پیکسل روی ماتریس ورودی بلغزیم، ما می‌توانیم با هر تعداد پیکسل روی ماتریس ورودی بلغزانیم.

[1] Stride

تعداد پیکسل‌هایی که توسط ماتریس فیلتر روی ماتریس ورودی می‌لغزیم، **گام پیشروی** نامیده می‌شود.

اگر گام پیشروی را روی ۲ تنظیم کنیم، آنگاه با «ماتریس فیلتر» روی «ماتریس ورودی» به اندازه دو پیکسل می‌لغزیم. شکل ۷.۳۵ یک عملیات همتابی با گام ۲ را نشان می‌دهد:



شکل ۷.۳۵: عملیات گام پیشروی

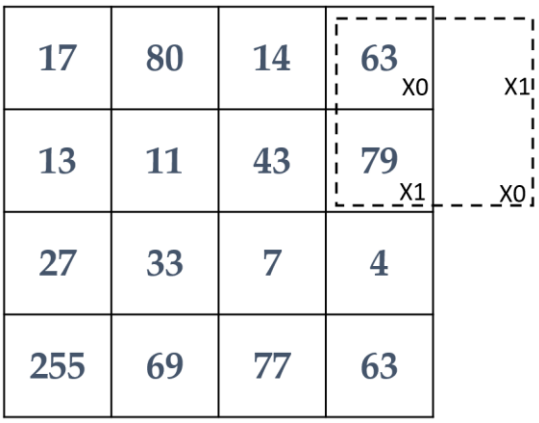
اما چگونه عدد گام را انتخاب کنیم؟ ما به تازگی یاد گرفتیم که یک گام، تعداد پیکسل‌هایی است که ماتریس فیلتر خود را حرکت می‌دهیم. بنابراین، هنگامی‌که گام، روی یک عدد کوچک تنظیم می‌شود، می‌توانیم نمایش دقیق‌تری از تصویر را نسبت به زمانی که گام، روی یک عدد بزرگ تنظیم شده، رمزگذاری کنیم. با این حال، یک گام با مقدار بالا، زمان کمتری نسبت به گام پیشروی کوچک می‌برد.

بالشتک[1]

با عملیات همتابی، ما ماتریس فیلتر را بر روی ماتریس ورودی می‌لغزانیم. اما در برخی موارد، فیلتر کاملا با ماتریس ورودی مطابقت ندارد. منظور ما از این موضوع چیست؟ به عنوان مثال، فرض کنید ما در حال انجام یک عمل همتابی با گام پیشروی ۲ هستیم. وضعیتی وجود دارد که وقتی ماتریس فیلتر خود را دو پیکسل حرکت می‌دهیم، به مرز می‌رسد و ماتریس فیلتر با ماتریس ورودی مطابقت ندارد. یعنی قسمتی از ماتریس فیلتر ما خارج از ماتریس ورودی

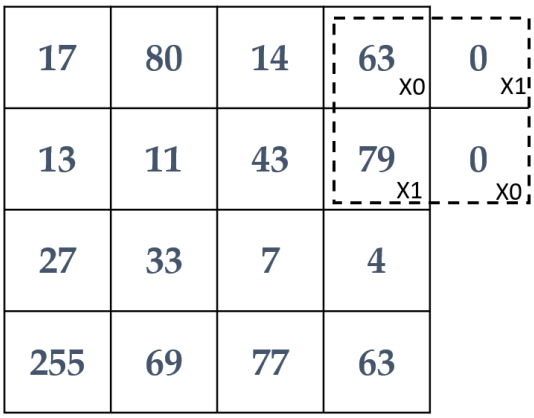
[1] Padding

قرار می‌گیرد، همانطور که در نمودار زیر نشان داده شده است:



شکل ۷.۳۶: عملیات بالشتک‌گذاری

در این مورد، ما ترفند بالشتک‌گذاری را انجام می‌دهیم. همانطور که در شکل ۷.۳۷ نشان داده شده است، می‌توانیم به سادگی ماتریس ورودی را با صفر بپوشانیم تا فیلتر بتواند با ماتریس ورودی مطابقت داشته باشد. بالشتک‌گذاری با مقدار صفر در ماتریس ورودی، **بالشتک همانی** یا **بالشتک صفر**[۱] نامیده می‌شود:

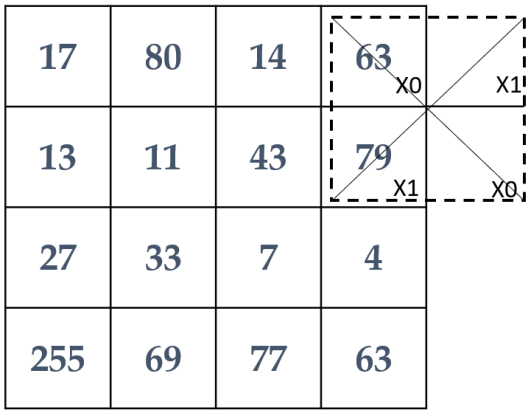


شکل ۷.۳۷: بالشتک همانی

به جای بالشتک‌گذاری با مقادیر صفر، می‌توانیم به سادگی، بی‌خیال ناحیه‌ای از ماتریس ورودی شویم که فیلتر در آن

[۱] Same Padding or Zero Padding

قرار نمی‌گیرد. به این ترفند، **بالشتک معتبر**[۱] می‌گویند:



شکل ۷.۳۸: بالشتک معتبر

فشرده‌سازی یا ادغام لایه‌ها[۲]

خیلی هم خب. اکنون، ما عملیات همتابی را تمام کرده‌ایم. در نتیجه عملیات همتابی، ما چند نقشه ویژگی داریم. اما نقشه‌های ویژگی از نظر ابعاد بسیار بزرگ هستند. به منظور کاهش ابعاد نقشه‌های ویژگی، عملیات فشرده‌سازی را انجام می‌دهیم. این امر ابعاد نقشه‌های ویژگی را کاهش می‌دهد و فقط جزئیات لازم را نگه می‌دارد تا میزان محاسبات کاهش یابد.

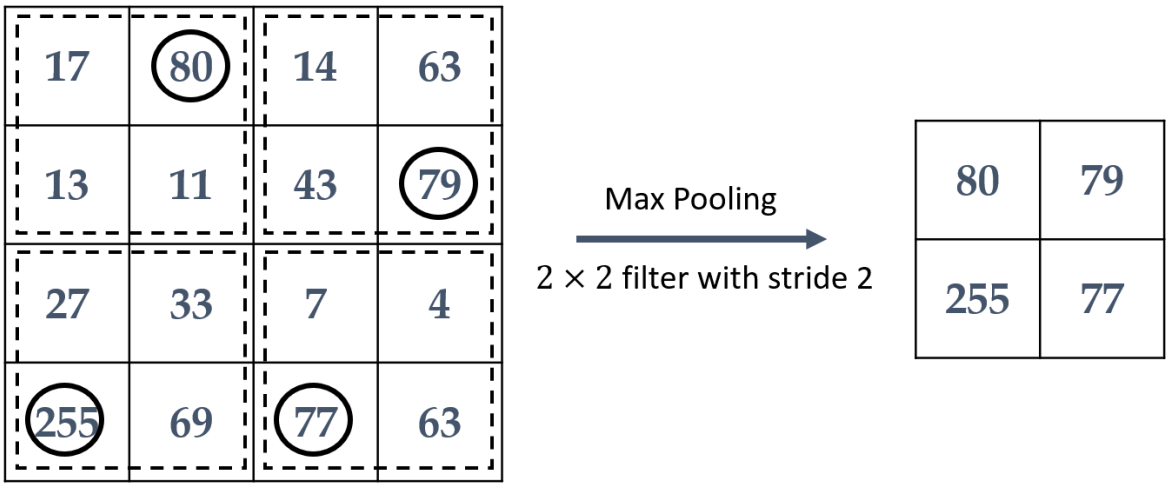
به عنوان مثال، برای تشخیص اسب از تصویر، باید فقط ویژگی‌های اسب را استخراج کرده و نگه داریم. ما به سادگی می‌توانیم ویژگی‌های ناخواسته مانند پس زمینه تصویر و موارد دیگر را کنار بگذاریم. به عملیات فشرده‌سازی، عملیات **نمونه-کاهی** یا **زیر-نمونه**[۳] نیز گفته می‌شود. این تکنیک، ترجمه یا برداشت **CNN** را **پایا**[۴] می‌کند. بنابراین، لایه فشرده‌سازی یا ادغام تنها با حفظ ویژگی‌های مهم، ابعاد فضایی را کاهش می‌دهد.

---

۱ Valid Padding
۲ Pooling Layers
۳ Downsampling or Subsampling
۴ Invariant

انواع مختلفی از عملیات ادغام وجود دارد، که از آن جمله‌اند: **بیشینه ادغام**[۱]، **متوسط ادغام**[۲] **و جمع ادغام**[۳].

در **حداکثر (بیشینه) ادغام**، ما فیلتر را بر روی ماتریس ورودی می‌لغزیم و به سادگی حداکثر مقدار را از پنجرهٔ فیلتر می‌گیریم. این موضع در شکل ۷.۳۹ نشان داده شده است.



شکل ۷.۳۹: بیشینه ادغام

در **متوسط ادغام**، مقدار متوسط ماتریس ورودی را در پنجرهٔ فیلتر می‌گیریم؛ و در **جمع ادغام**، تمام مقادیر ماتریس ورودی را در پنجرهٔ فیلتر، تجمیع می‌کنیم.

لایه‌های کاملا متصل

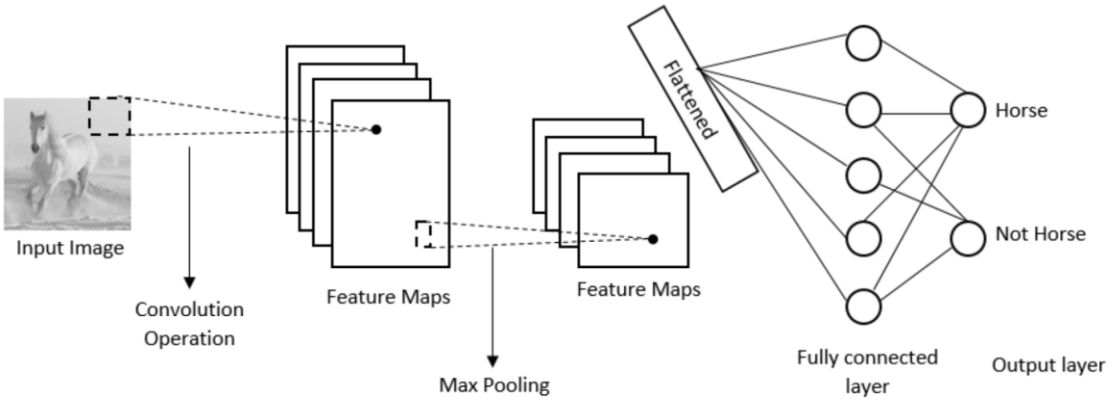
تاکنون، ما یاد گرفته‌ایم که لایه‌های همتابی و ادغام چگونه کار می‌کنند. یک **CNN** می‌تواند چندین لایهٔ همتابی و نیز چندین لایهٔ ادغام داشته باشد. با این حال، کار این لایه‌ها فقط آنست که ویژگی‌ها را از تصویر ورودی استخراج کرده و نقشه ویژگی را تولید کنند. یعنی آنها فقط استخراج کننده‌های ویژگی هستند.

---

۱ Max Pooling
۲ Average Pooling
۳ Sum Pooling

با توجه به هر تصویری، لایه‌های همتابی، ویژگی‌ها را از تصویر استخراج کرده و یک نقشه ویژگی تولید می‌کنند. اکنون، ما باید این ویژگی‌های استخراج شده را طبقه‌بندی کنیم. بنابراین، ما به الگوریتمی نیاز داریم که بتواند این ویژگی‌های استخراج شده را طبقه‌بندی کرده و به ما بگوید که آیا ویژگی‌های استخراج شده ویژگی‌های یک اسب هستند یا خیر. برای انجام این دسته‌بندی، از یک شبکه عصبی پیشخور استفاده می‌کنیم. ما نقشه ویژگی را **صاف**[۱] کرده و آن را به یک بردار تبدیل نموده و سپس آن را به عنوان ورودی، به شبکه پیشخور، تغذیه می‌کنیم.

شبکه پیشخور، این نقشه ویژگی مسطح را به عنوان ورودی گرفته، بر آن یک تابع فعالسازی مانند سیگموئید را اعمال کرده و خروجی را برمی‌گرداند و بیان می‌کند که آیا تصویر حاوی اسب است یا خیر. این کار را یک لایه‌ای انجام می‌دهد که **لایه کاملا متصل**[۲] نامیده می‌شود و در نمودار زیر نشان داده شده است:



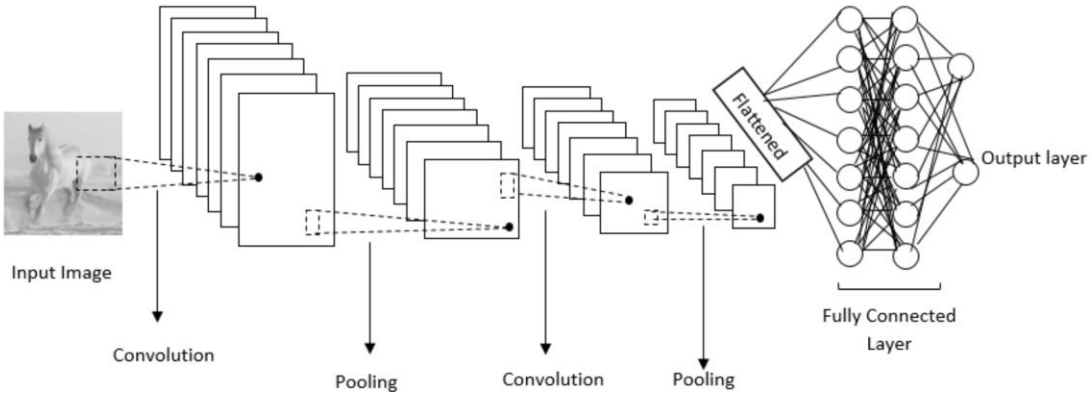
شکل ۷.۴۰: لایه کاملا متصل

بیایید ببینیم که چگونه همه اینها با همدیگر منطبق و سازگار می‌شوند.

معماری CNNها

[۱] Flat
[۲] Fully Connected Layer

شکل ۷.۴۱ معماری یک **CNN** را نشان می‌دهد:



شکل ۷.۴۱: معماری CNN

همانطور که متوجه خواهید شد، ابتدا تصویر ورودی را به لایه همتابی تغذیه می‌کنیم، جایی که عملیات همتابی را برای استخراج ویژگی‌های مهم از تصویر و ایجاد نقشه‌های ویژگی اعمال می‌کنیم. سپس نقشه‌های ویژگی را به لایه ادغام منتقل می‌کنیم، جایی که ابعاد نقشه‌های ویژگی کاهش می‌یابد.

همانطور که در نمودار قبلی نشان داده شده است، ما می‌توانیم چندین لایه همتابی و ادغام داشته باشیم، و همچنین باید توجه داشته باشیم که لایه ادغام لزوما نباید بعد از هر لایه همتابی وجود داشته باشد. می‌تواند لایه‌های همتابی زیادی وجود داشته باشد و به دنبال آن یک لایه ادغام وجود داشته باشد.

بنابراین، پس از لایه‌های همتابی و ادغامی، نقشه‌های ویژگی حاصل را صاف کرده و آن را به یک «لایه کاملا متصل» تغذیه می‌کنیم، که اساسا یک «شبکه عصبی پیشخور» است و تصویر ورودی داده شده را بر اساس نقشه‌های ویژگی طبقه‌بندی می‌کند.

اکنون که نحوه عملکرد **CNN**ها را یاد گرفتیم، در بخش بعدی، با الگوریتم جالب دیگری به نام «شبکه مولد متخاصم» یا «مولد–تخاصمی» آشنا خواهیم شد.

شبکه‌های مولد تخاصمی یا تقابلی (GANs) [۱]

شبکه‌های مولد تخاصمی یا تقابلی به طور گسترده برای تولید نقاط داده جدید استفاده می‌شوند. آنها را می‌توان برای هر نوع مجموعه داده‌ای اعمال کرد، اما به طور رایج برای تولید تصاویر استفاده می‌شود. برخی از کاربردهای GAN شامل تولید چهره‌های واقعی انسان، تبدیل تصاویر در مقیاس خاکستری به تصاویر رنگی، ترجمه توضیحات متنی به تصاویر واقعی و بسیاری موارد دیگر است.

GANها در سال‌های اخیر آنقدر تکامل یافته اند که می‌توانند تصویری بسیار واقع گرایانه ایجاد کنند. شکل زیر تکامل **GAN**ها را در تولید تصاویر در طول پنج سال نشان می‌دهد:

شکل ۷.۴۲: تکامل **GAN** در طول سال‌ها

در حال حاضر در مورد **GAN** هیجان زده هستید؟ اکنون خواهیم دید که دقیقا چگونه کار می‌کنند. قبل از ادامه، بیایید یک قیاس ساده را در نظر بگیریم. فرض کنید شما پلیس هستید و وظیفه شما یافتن پول تقلبی است و نقش جاعل، ایجاد پول جعلی و فریب پلیس است.

جاعل دائما سعی می‌کند پول جعلی را به گونه‌ای ایجاد کند که آنقدر واقع بینانه باشد که نتوان آن را از پول واقعی

---

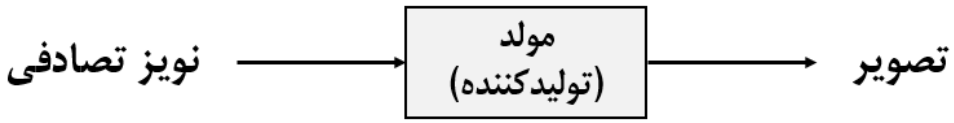
[۱] Generative Adversarial Networks (GAN)
این شبکه‌ها برای اولین بار در مقاله‌ای با عنوان Generative Adversarial Networks در سال ۲۰۱۴ توسط افراد زیر معرفی شد:
Ian J Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio

تمیز داد. اما پلیس باید تشخیص دهد که آیا پول واقعی است یا جعلی. بنابراین، جاعل و پلیس اساسا یک بازی دو نفره انجام می‌دهند که در آن یکی سعی می‌کند دیگری را شکست دهد. **GAN**ها چیزی شبیه به این کار می‌کنند. آنها از دو جزء مهم تشکیل شده‌اند:

۱. **مولد یا تولیدکننده**[۱]
۲. **ممیز یا تمیزدهنده**[۲]

شما می‌توانید مولد را مشابه جاعل درک کنید، در حالی که ممیز، مشابه پلیس است. یعنی نقش مولد، ایجاد پول جعلی است و نقش ممیز، تشخیص جعلی یا واقعی بودن پول است.

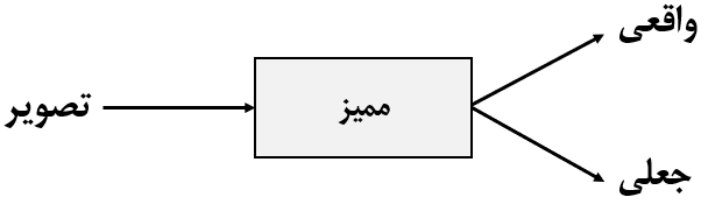
بدون پرداختن به جزئیات، ابتدا درک اولیه ای از **GAN**ها به دست خواهیم آورد. فرض کنید می‌خواهیم **GAN** ما ارقام دستنویس تولید کند. چگونه می‌توانیم این کار را انجام دهیم؟ ابتدا، مجموعه داده‌ای حاوی مجموعه ای از ارقام دستنویس را می‌گیریم. مثلا، مجموعه داده **MNIST**. مولد، توزیع تصاویر در مجموعه دادهٔ ما را یاد می‌گیرد. بنابراین، مولد ما، توزیعِ ارقامِ دستنویس[۳] در مجموعه آموزشی را یاد می‌گیرد. وقتی مولد، توزیع تصاویر در مجموعه داده ما را یاد گرفت، و ما یک نویز تصادفی را به مولد تغذیه کنیم؛ مولد، نویز تصادفی را به یک رقم دستنویس جدید، شبیه به ارقام موجود در مجموعه آموزشی ما و بر اساس توزیع آموخته شده، تبدیل می‌کند



شکل ۷.۴۳: مولد (تولید کننده)

هدفِ ممیز انجام یک کار طبقه‌بندی است. یعنی با گرفتن یک تصویر، آن را به عنوان واقعی یا جعلی طبقه‌بندی می‌کند. یعنی اینکه آیا این تصویر از مجموعه آموزشی است و یا اینکه تصویر توسط مولد ما تولید شده است:

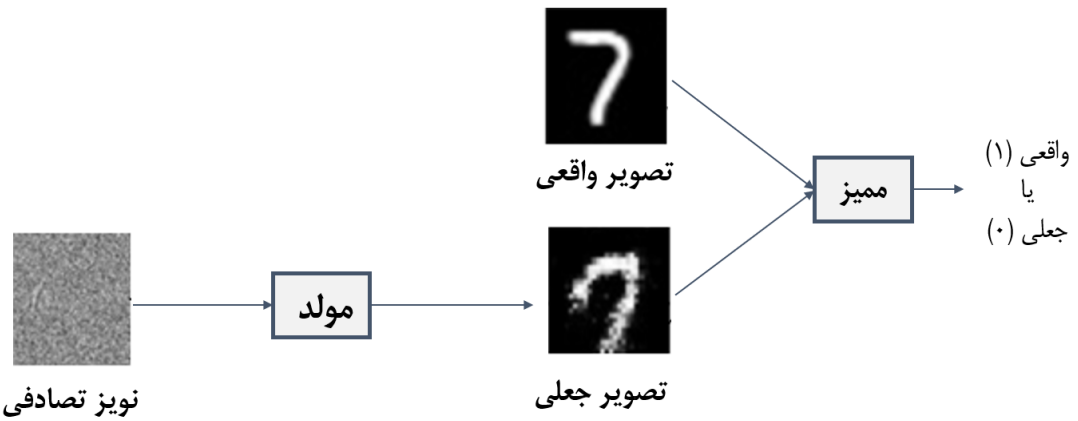
[۱] Generator
[۲] Discriminator
[۳] Handwritten Digits



شکل ۷.۴۴: ممیز (متمایز کننده)

مؤلفهٔ مولدِ **GAN** اساسا یک مدلِ تولیدگر[۱] است و مؤلفهٔ ممیز اساسا یک مدلِ تفکیک‌گر[۲] است. بنابراین، مولد، توزیع کلاس را یاد می‌گیرد و ممیز، مرز تصمیم‌گیری یک کلاس را فرا می‌گیرد

همانطور که شکل ۷.۴۵ نشان می‌دهد، ما نویز تصادفی را به مولد تغذیه کرده و سپس مولد، این نویز تصادفی را به تصویری جدید، شبیه به آنچه در مجموعه آموزشی خود داریم، تبدیل می‌کند، اما دقیقا مشابه تصاویر موجود در مجموعه آموزشی نیست. تصویر تولید شده توسط ژنراتو (مولد) تصویر جعلی نامیده می‌شود و تصاویر موجود در مجموعه آموزشی ما تصاویر واقعی نامیده می‌شوند. ما تصاویر واقعی و جعلی را به ممیز می‌دهیم، که احتمال واقعی بودن آنها را به ما می‌گوید. اگر تصویر جعلی باشد، ممیز عدد ۰ و اگر واقعی باشد عدد ۱ را برمی‌گرداند.



شکل ۷.۴۵: شبکه‌های متخاصم مولد

اکنون که درک اولیه‌ای از مولدها و ممیزها داریم، هر یک از اجزا را با جزئیات مطالعه خواهیم کرد.

---

[۱] Generative
[۲] Discriminative

جزئیات مدل مولد (یا تولیدکننده)[۱]

جزء مولد یا ژنراتورِ یک **GAN**، یک مدلِ تولیدگر است. وقتی می‌گوییم مدل تولیدگر، باید بدانیم که دو نوع مدل تولیدگر وجود دارد: یکی **مدل چگالی ضمنی**[۲] و دیگری **مدل چگالی صریح**[۳]. مدل چگالی ضمنی از هیچ گونه تابع چگالی صریح برای یادگیری توزیع احتمال استفاده نمی‌کند، در حالی که مدل چگالی صریح، همانطور که از نامش پیداست، از یک تابع چگالی صریح استفاده می‌کند. **GAN**ها در دسته اول قرار می‌گیرند. یعنی آنها یک مدل چگالی ضمنی هستند. بیایید جزئیات را بررسی کرده و بفهمیم که چگونه **GAN**ها یک مدل چگالی ضمنی هستند.

فرض کنید ما یک مولد داریم که با علامت G نشانش می‌دهیم. این مولد، اساسا یک شبکه عصبی است که توسط θ_g پارامتری شده است. نقش شبکهٔ مولد، تولید تصاویر جدید است. چگونه این کار را انجام می‌دهند؟ ورودی مولد چه باید باشد؟

ما یک نویز تصادفی یعنی z را از یک توزیع نرمال یا یکنواخت یعنی P_z نمونه‌برداری می‌کنیم. سپس این نویز تصادفی z، را به عنوان ورودی به مولد تغذیه کرده و بدنبال آن، مولد این نویز را به یک تصویر تبدیل می‌کند:

$$G(z;\theta_g)$$

تعجب آور است، اینطور نیست؟ مولد ما، چگونه نویز تصادفی را به یک تصویر واقعی تبدیل می‌کند؟

فرض کنید ما یک مجموعه داده داریم که شامل مجموعه‌ای از چهره‌های انسان است و می‌خواهیم مولد ما یک چهره انسانی جدید ایجاد کند. ابتدا، مولد با یادگیری توزیع احتمال تصاویر در مجموعه آموزشی ما، تمام ویژگی‌های چهره را یاد می‌گیرد. هنگامی‌که مولد، توزیع احتمال صحیح را یاد می‌گیرد، می‌تواند چهره‌های کاملا جدیدی را ایجاد کند.

اما مولد چگونه توزیع مجموعه آموزشی را یاد می‌گیرد؟ یعنی مولد چگونه توزیع تصاویر چهرهٔ انسان را در مجموعهٔ

---

۱ Breaking Down the Generator
۲ Implicit Density Model
۳ Explicit Density Model

آموزشی یاد می‌گیرد؟

مولد چیزی نیست جز یک شبکه عصبی. بنابراین، آنچه اتفاق می‌افتد این است که شبکه عصبی، توزیع تصاویر را در مجموعه آموزشی ما به طور ضمنی یاد می‌گیرد. بیایید این توزیع را توزیع مولد، P_g بنامیم. در اولین تکرار، مولد یک تصویر واقعا مغشوش تولید می‌کند. اما در طی یک سری تکرار، توزیع احتمال دقیق مجموعه آموزشی ما را یاد می‌گیرد، یعنی می‌آموزد که یک تصویر صحیح با تنظیم پارامتر θ_g تولید کند.

توجه به این نکته مهم است که ما از توزیع یکنواخت P_z برای یادگیری توزیع مجموعه آموزشی خود استفاده نمی‌کنیم. بلکه از آن فقط برای نمونه‌برداری از نویز تصادفی استفاده می‌شود. ما این نویز تصادفی را به عنوان ورودی مولد به آن تغذیه می‌کنیم. شبکه مولد به طور ضمنی، توزیع مجموعه آموزشی ما را یاد می‌گیرد. این توزیع را توزیع مولد، P_g می‌نامند و به همین دلیل است که ما شبکه مولد خود را یک مدل چگالی ضمنی می‌نامیم.

اکنون که مولد را درک کردیم، بیایید به سراغ ممیز (تمیزدهنده) برویم.

جزئیات مدل ممیز (تمیزدهنده)[1]

همانطور که از نام آن پیداست، ممیز، یک مدلِ تفکیک‌گر است. فرض کنید ما یک تمیزدهنده بنام D داریم. این مدل همچنین یک شبکه عصبی است و توسط θ_d پارامتری شده است.

هدف تمیزدهنده، تشخیص بین دو طبقه است. یعنی با توجه به یک تصویر x، باید تشخیص دهد که آیا تصویر از یک توزیع واقعی است یا یک توزیع جعلی (توزیع مولد). یعنی ممیز باید تشخیص دهد که آیا تصویر ورودی داده شده از مجموعه آموزشی است یا تصویر جعلی تولید شده توسط مولد:

[1] Breaking Down the Discriminator

$$D(x;\theta_d)$$

بیایید توزیع مجموعه آموزشی خود را توزیع داده واقعی بنامیم، که با P_r نشان داده می‌شود. می‌دانیم که توزیع مولد با P_g نشان داده می‌شود.

بنابراین، تمایزدهنده D اساسا سعی می‌کند تشخیص دهد که آیا تصویر x از P_r است یا P_g

با این حال، آنها چگونه یاد می‌گیرند؟

تاکنون، ما فقط نقش مولد و ممیز را مطالعه کردیم، اما آنها دقیقا چگونه یاد می‌گیرند؟ مولد چگونه یاد می‌گیرد که تصاویر واقع‌گرایانه جدید تولید کند و چگونه ممیز یاد می‌گیرد که بین تصاویر به درستی تمایز قائل شود؟

ما می‌دانیم که هدف مولد، تولید یک تصویر به گونه‌ای است که ممیز را فریب دهد تا ممیز باور کند که تصویر تولید شده از یک توزیع واقعی است.

در اولین تکرار، مولد یک تصویر مغشوش تولید می‌کند. وقتی این تصویر را به ممیز تغذیه می‌کنیم، به راحتی می‌تواند تشخیص دهد که تصویر از یک توزیع مولد است. مولد این را به عنوان یک ضرر یا شکست می‌داند و سعی می‌کند خود را بهبود بخشد، زیرا هدف آن فریب دادن ممیز است. یعنی اگر مولد بداند که ممیز به راحتی تصویر تولید شده را به عنوان یک تصویر جعلی تشخیص می‌دهد، پس به این معنی است که تصویری شبیه به آنچه در مجموعه آموزشی وجود دارد تولید نمی‌کند. این بدان معناست که مولد هنوز توزیع احتمال مجموعه آموزشی را یاد نگرفته است.

بنابراین، مولد پارامترهای خود را به گونه‌ای تنظیم می‌کند که توزیع احتمال صحیح مجموعه آموزشی را یاد بگیرد. همانطور که می‌دانیم مولد، یک شبکه عصبی است، ما به سادگی پارامترهای شبکه را از طریق پس‌انتشار به روز می‌کنیم. هنگامی‌که توزیع احتمال تصاویر واقعی را یاد گرفت، می‌تواند تصاویری مشابه تصاویر موجود در مجموعه آموزشی تولید کند

بسیار خوب، در مورد ممیز چطور؟ ممیز چگونه یاد می‌گیرد؟ همانطور که می‌دانیم، نقش ممیز یا تفکیک‌گر این است

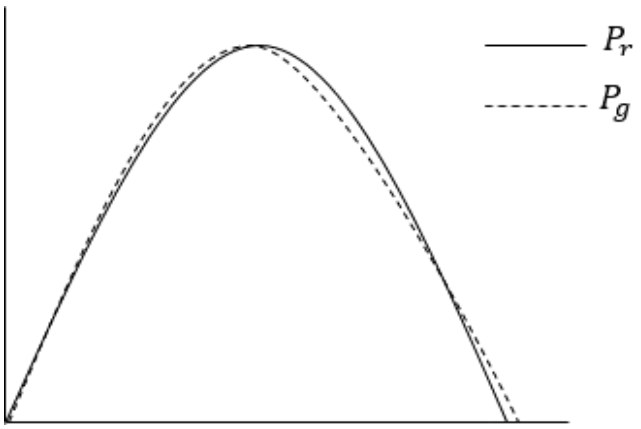
که بین تصاویر واقعی و جعلی تمایز قائل شود.

اگر ممیز، تصویر تولید شده را به اشتباه طبقه‌بندی کند؛ یعنی اگر متمایزکننده تصویر جعلی را به عنوان یک تصویر واقعی طبقه‌بندی کند، به این معنی است که ممیز یاد نگرفته است که بین تصویر واقعی و جعلی تفاوت قائل شود. بنابراین، پارامتر شبکه ممیز را از طریق پس–انتشار به‌روز می‌کنیم تا متمایز کننده یاد بگیرد که بین تصاویر واقعی و جعلی تمیز قائل شده و آنها را طبقه بندی کند.

بنابراین، اساسا، مولد سعی می‌کند با یادگیری توزیع داده‌های واقعی، P_r، ممیز را فریب دهد و ممیز در تلاش است تا بفهمد که آیا تصویر از یک توزیع واقعی است یا جعلی. حال سوال این است که چه زمانی، آموزش شبکه را با توجه به این واقعیت که مولد و ممیز با یکدیگر رقابت می‌کنند، متوقف کنیم؟

اساسا هدف **GAN** تولید تصاویری مشابه تصاویری است که در مجموعه آموزشی وجود دارد. فرض کنید ما می‌خواهیم یک چهره انسانی ایجاد کنیم. ما توزیع تصاویر را در مجموعه آموزشی یاد می‌گیریم و تعدادی چهره‌های جدید ایجاد می‌کنیم. بنابراین، برای یک مولد، ما باید ممیز بهینه را پیدا کنیم. منظور ما از این مفهوم چیست؟

ما می‌دانیم که یک توزیع مولد با P_g و توزیع داده واقعی با P_r نشان داده می‌شود. اگر مولد، توزیع داده واقعی را به طور کامل یاد بگیرد، P_g برابر با P_r است، همانطور که شکل ۷.۴۶ نشان می‌دهد:



شکل ۷.۴۶: مولد و توزیع داده‌های واقعی

اگر $P_g = P_r$ باشد، آنگاه ممیز نمی‌تواند در مورد اینکه تصویر ورودی از یک توزیع واقعی یا جعلی است، تمایز قائل شود، بنابراین فقط مقدار ۰.۵ را به عنوان یک احتمال برمی‌گرداند در واقع، زمانی که دو توزیع، عین هم باشند، ممیز دچار اشتباه می‌شود.

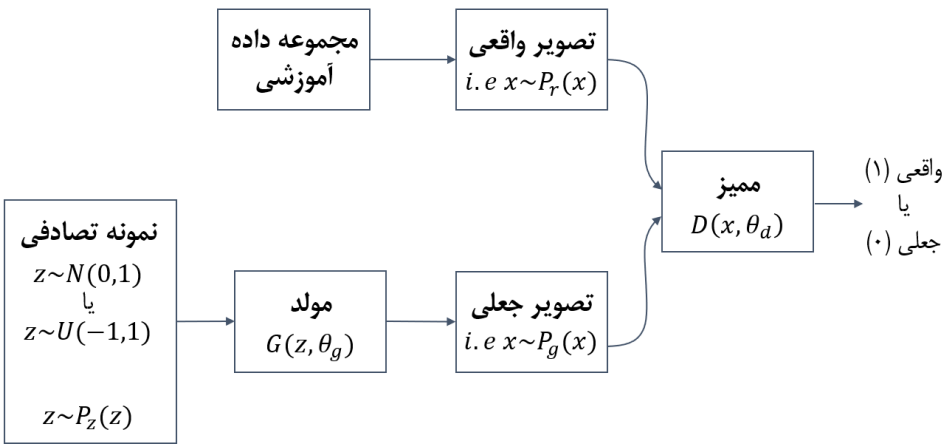
بنابراین، برای هر مولد، یک ممیز بهینه را می‌توان به صورت زیر ارائه داد:

$$D(x) = \frac{P_r(x)}{P_r(x) + P_g(x)} = \frac{۱}{۲}$$

بنابراین، وقتی ممیز فقط احتمال ۰.۵ را برای همه تصاویر مولد برمی‌گرداند، می‌توان گفت که مولد، توزیعِ تصاویر را در مجموعه آموزشی ما یاد گرفته و ممیز را با موفقیت فریب داده است.

معماری یک GAN

شکل ۷.۴۷ معماری یک **GAN** را نشان می‌دهد:



شکل ۷.۴۷: معماری **GAN**

همانطور که در نمودار قبلی نشان داده شده است، مولد یا ژنراتور G نویز تصادفی، z، را به عنوان ورودی با نمونه‌برداری از یک توزیع یکنواخت یا نرمال می‌گیرد و با یادگیری ضمنی توزیع مجموعه آموزشی، یک تصویر جعلی ایجاد می‌کند.

ما یک تصویر x، از توزیع داده واقعی $x \sim P_r(x)$، و توزیع داده‌های جعلی $x \sim P_g(x)$ نمونه‌برداری کرده و آنرا به ممیز، D تغذیه می‌کنیم. ما تصاویر واقعی و جعلی را به ممیز داده و ممیز یک کار طبقه‌بندی باینری را انجام می‌دهد. یعنی وقتی تصویر جعلی است، مقدار ۰ و زمانی که تصویر واقعی است، مقدار ۱ را برمی‌گرداند.

ابهام‌زدایی از تابع زیان

اکنون تابع ضرر **GAN** را بررسی خواهیم کرد. قبل از ادامه، اجازه دهید نمادگذاری را خلاصه کنیم:

۱. اغتشاش (نویز) که به عنوان ورودی به مولد تغذیه می‌شود با z نشان داده می‌شود.
۲. توزیع یکنواخت یا نرمال که نویز z از آن نمونه‌برداری می‌شود با P_z نشان داده می‌شود.
۳. یک تصویر ورودی با x نشان داده می‌شود.
۴. توزیع داده‌های واقعی یا توزیع مجموعه آموزشی ما توسط P_r نشان داده شده است.
۵. توزیع داده‌های جعلی یا توزیع مولد توسط P_g نشان داده می‌شود.

وقتی $x \sim P_r(x)$ را می‌نویسیم، به این معنی است که تصویر x از **توزیع واقعی** P_r نمونه‌برداری شده است. به طور مشابه، نماد $x \sim P_g(x)$ نشان می‌دهد که تصویر x از **توزیع مولد** یا ژنراتور یعنی P_g نمونه‌برداری شده است. بعلاوه $z \sim P_z(z)$ به این معنی است که ورودی مولد، z، از **توزیع یکنواخت**، P_z نمونه‌برداری می‌کند

ما آموخته‌ایم که مولد و ممیز هر دو شبکه عصبی هستند و هر دو پارامترهای خود را از طریق پس–انتشار به‌روز می‌کنند. اکنون باید پارامتر مولد بهینه، θ_g و پارامتر ممیز، θ_d را پیدا کنیم.

تابع زیان ممیز

اکنون به تابعِ زیانِ ممیز نگاه خواهیم کرد. ما می‌دانیم که هدف ممیز، طبقه‌بندی واقعی یا جعلی بودن تصویر است. بیایید ممیز را با D نشان دهیم. تابع ضرر ممیز به شرح زیر است:

$$\max_{d} L(D,G) = \mathbb{E}_{x\sim P_r(x)}[\log D(x;\theta_d)] + \mathbb{E}_{z\sim P_z(z)}\left[\log\left(۱ - D\big(G(z;\theta_g);\theta_d\big)\right)\right]$$

با این حال، این به چه معناست؟ بیایید هر یک از عبارات را یکی به یکی درک کنیم

عبارت اول

بیایید به عبارت اول نگاه کنیم:

$$\mathbb{E}_{x\sim P_r(x)} \log\big(D(x)\big)$$

در اینجا، $x\sim P_r(x)$ به این معنی است که ما در حال نمونه برداری ورودی x از توزیع داده واقعی یعنی P_r هستیم. بنابراین x یک تصویر واقعی است.

$D(x)$ به این معنی است که ما تصویر ورودی x را به ممیز D تغذیه می‌کنیم و ممیز، احتمال اینکه تصویر ورودی x یک تصویر واقعی باشد را برمی‌گرداند. از آنجایی که x از توزیع داده واقعی یعنی P_r نمونه‌برداری شود، می‌دانیم که x یک تصویر واقعی است. بنابراین، ما باید احتمال $D(x)$ را به حداکثر برسانیم:

$$max\, D(x)$$

اما به جای به حداکثر رساندن احتمالات خام، ما لگاریتم احتمالات ورود به سیستم را به حداکثر می‌رسانیم، بنابراین، می‌توانیم موارد زیر را بنویسیم:

$$max \log D(x)$$

بنابراین، معادله نهایی ما به شرح زیر است:

$$max\, \mathbb{E}_{x\sim P_r(x)} \log[D(x)]$$

عبارت بعد از max، نشاندهنده **انتظارات لگاریتم-راستنمایی**[۱] تصاویر ورودی نمونه برداری شده، از توزیع داده‌های واقعی است.

عبارت دوم

حال، بیایید به عبارت دوم نگاه کنیم:

$$\mathbb{E}_{z\sim P_z(z)}\left[\log\left(۱ - \mathrm{D}(\mathrm{G}(\mathrm{z}))\right)\right]$$

در اینجا، $z\sim P_z(z)$ نشان می‌دهد که ما در حال نمونه‌برداری نویز تصادفی z از توزیع یکنواخت P_z هستیم. $\mathrm{G}(\mathrm{z})$ به این معنی است که مولد G، نویز تصادفی z را به عنوان ورودی می‌گیرد و یک تصویر جعلی را بر اساس توزیع ضمنی آموخته شده P_g برمی‌گرداند.

در فرمول بالا $D(G(z))$ به این معناست که ما تصویر جعلی تولید شده توسط مولد را به ممیز D تغذیه می‌کنیم و احتمال واقعی بودن تصویر ورودی جعلی را برمی‌گرداند.

اگر $D(G(z))$ را از ۱ کم کنیم، احتمال جعلی بودن تصویر ورودی جعلی را برمی‌گرداند:

$$۱ - D\big(G(z)\big)$$

از آنجایی که می‌دانیم z یک تصویر واقعی نیست، ممیز این احتمال را به حداکثر می‌رساند. یعنی ممیز، احتمال طبقه‌بندی z به عنوان یک تصویر جعلی را به حداکثر می‌رساند ، بنابراین می‌نویسیم:

$$max\ ۱ - D\big(G(z)\big)$$

به جای به حداکثر رساندن احتمالات خام، ما لگاریتم احتمال را به حداکثر می‌رسانیم:

[۱] log-likelihood

$$max \log\Big(۱ - D\big(G(z)\big)\Big)$$

لذا عبارت زیر به معنای انتظاراتِ لگاریتم راستنمایی جعلی بودن تصاویر ورودی تولید شده توسط مولد است.

$$\mathbb{E}_{z\sim P_z(z)}\Big[\log\Big(۱ - \mathrm{D}\big(\mathrm{G}(\mathrm{z})\big)\Big)\Big]$$

عبارت نهایی

بنابراین، با ترکیب این دو عبارت، تابع ضرر ممیز به شرح زیر داده می‌شود:

$$\max_d L(D,G) = \mathbb{E}_{x\sim P_r(x)}[\log D(x;\theta_d)] + \mathbb{E}_{z\sim P_z(z)}\Big[\log\Big(۱ - D\big(G\big(z;\theta_g\big);\theta_d\big)\Big)\Big]$$

در اینجا، θ_g و θ_d به ترتیب پارامترهای شبکه مولد و ممیز هستند. بنابراین، هدفِ ممیز، یافتن θ_d مناسب است تا بتواند تصویر را به درستی طبقه‌بندی کند.

تابع زیان مولد

تابع زیان مولد به شرح زیر است:

$$\min_g L(D,G) = \mathbb{E}_{z\sim P_z(z)}\Big[\log\Big(۱ - D\big(G\big(z;\theta_g\big);\theta_d\big)\Big)\Big]$$

ما می‌دانیم که هدف مولد، فریبِ ممیز برای طبقه‌بندی تصویر جعلی به عنوان یک تصویر واقعی است.

در بخش **تابع زیان ممیز**، دیدیم که عبارت:

$$\mathbb{E}_{z\sim P_z(z)}\big[\log\big(۱ - D\big(G(z)\big)\ \big)\big]$$

به احتمال طبقه‌بندی تصویر ورودی جعلی به عنوان یک تصویر جعلی دلالت دارد و ممیز احتمالات طبقه‌بندی صحیح تصویر جعلی را به عنوان جعلی به حداکثر می‌رساند.

اما مولد می‌خواهد این احتمال را به حداقل برساند. از آنجایی که مولد می‌خواهد ممیز را فریب دهد، این احتمال (که یک تصویر ورودی جعلی توسط ممیز به عنوان جعلی طبقه‌بندی شود) را به حداقل می‌رساند. بنابراین، تابع زیان مولد را می‌توان به صورت زیر بیان کرد:

$$\min_g L(D,G) = \mathbb{E}_{z\sim P_z(z)}\left[\log\left(1 - D\left(G\left(z;\theta_g\right);\theta_d\right)\right)\right]$$

تابع زیان کلی

ما به تازگی تابع زیان مولد و ممیز را یاد گرفتیم. با ترکیب این دو تابع ضرر، تابع ضرر نهایی خود را به صورت زیر می‌نویسیم:

$$\min_G \max_D L(D,G) = \mathbb{E}_{x\sim P_r(x)}[\log D(x)] + \mathbb{E}_{z\sim P_z(z)}\left[\log\left(1 - D\left(G(z)\right)\right)\right]$$

بنابراین، تابع هدف ما اساسا یک تابع هدف min-max است، یعنی **بیشینه‌سازی برای ممیز** و **کمینه‌سازی برای مولد**، و ما می‌خواهیم پارامتر بهینه مولد، θ_g و پارامتر بهینه ممیز، θ_d را از طریق پس–انتشار شبکه‌های مربوطه پیدا کنید.

بنابراین، ما افزایش گرادیان (صعود نشیب) را انجام می‌دهیم که به معنای حداکثرسازی تابع زیان ممیز است:

$$\nabla\theta_d \frac{1}{m}\sum_{i=1}^{m}\left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

و ما کاهش گرادیان (نزول نشیب) را برای به حداقل رساندن تابع زیان مولد انجام می‌دهیم:

$$\nabla\theta_g \frac{1}{m}\sum_{i=1}^{m}\log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

خلاصه

ما فصل را با درک عصب‌های طبیعی و مصنوعی شروع کردیم. سپس با شبکه‌های عصبی مصنوعی (**ANN**ها) و لایه‌های آنها آشنا شدیم. ما انواع مختلفی از توابع فعالسازی و نحوه استفاده از آنها برای معرفی غیرخطی بودن در شبکه را یاد گرفتیم.

بعدا با انتشار **رو به جلو (پیشرو)** و انتشار **رو به عقب (پسرو)** در شبکه عصبی آشنا شدیم. در مرحله بعد، یاد گرفتیم که چگونه یک **ANN** را پیاده‌سازی کنیم. در مرحله بعد، با نوعی **شبکه عصبی بازگشتی** (**RNN**) به نام **حافظه کوتاه-مدت دیرپا** (**LSTM**) آشنا شدیم. در ادامه، ما در مورد شبکه‌های عصبی همتابی (**CNN**ها)، نحوه استفاده آنها از انواع مختلف لایه‌ها و معماری **CNN**ها مطالبی را یاد گرفتیم.

در پایان فصل، با الگوریتم جالبی به نام **شبکه‌های مولد تخاصمی یا تقابلی** (**GAN**) آشنا شدیم. ما مولفه مولد و ممیز **GAN** را درک کردیم و همچنین معماری **GAN** را با جزئیات بررسی نمودیم. در ادامه، تابع زیان **GAN** را به طور مفصل بررسی کردیم. در فصل بعدی، با یکی از پرکاربردترین چارچوب‌های یادگیری عمیق یعنیTensorFlow آشنا خواهیم شد.

سوالات

بیایید درک خود را از الگوریتم‌های یادگیری عمیق با پاسخ دادن به سوالات زیر ارزیابی کنیم:

۱. تابع فعالسازی چیست؟

۲. تابع بیشینه-نرم (softmax) را تعریف کنید.

۳. دوره (epoch) چیست؟

٤. برخی از کاربردهای شبکه‌های عصبی بازگشتی (**RNN**) چیست؟

٥. مشکل امحاء گرادیان را توضیح دهید.

۶. انواع مختلف عملیات ادغام چیست؟

۷. مولفه‌های مولد و ممیز **GAN**ها را توضیح دهید.

بیشتر بخوانید

برای کسب اطلاعات بیشتر در مورد الگوریتم‌های یادگیری عمیق، می‌توانید کتاب زیر از همین نویسنده را بررسی کنید:

Hands-on Deep Learning Algorithms with Python, also published by Packt, at https://www.packtpub.com/in/big-data-and-business-intelligence/ hands-deep-learning-algorithms-python.

فصل هشتم

اصول مقدماتی تنسور–فلو

تنسور–فلو[1] است یکی از محبوبترین کتابخانه‌های یادگیری عمیق است. در فصول بعدی ما از تنسور–فلو برای ساخت مدلهای یادگیری تقویتی عمیق استفاده خواهیم کرد. بنابراین، در این فصل، با تنسور–فلو و عملکردهای آن آشنا می‌شویم.

ما در مورد کارکرد گرافهای محاسباتی[2] و نیز چگونگی استفاده از تنسور–فلو خواهیم آموخت. ما همچنین **تنسور–بُرد**[3] را بررسی خواهیم کرد، که یک ابزار تجسم‌سازی توسط تنسور–فلو است که برای تجسم مدلها از آن استفاده می‌شود. در ادامه، نحوه ساخت یک شبکه عصبی با استفاده از تنسور–فلو برای طبقه‌بندی اعداد یا ارقام دستنویس را بحث خواهیم کرد.

در ادامه، با تنسور–فلو نسخه ۲.۰ که آخرین نسخه تنسور–فلو است، آشنا خواهیم شد. ما خواهیم فهمید که تنسور–فلو ۲.۰ چه تفاوتی با نسخه‌های قبلی خود دارد و چگونه از Keras به عنوان واسط برنامه‌نویسی کاربری[4] (**API**) سطح بالای خود استفاده می‌کند.

در این فصل با موارد زیر آشنا می‌شویم:

- **تنسور–فلو**
- **گرافها و بخشهای محاسباتی**
- **متغیرها، ثابت‌ها و متغیرها**
- **تنسور–بُرد**
- **طبقه‌بندی ارقام دست نویس در تنسور–فلو**
- **عملیات ریاضی در تنسور–فلو**
- **تنسور–فلو نسخه ۲.۰ و Keras**

---

[1] TensorFlow

تنسور، نامی است برای ماتریسهایی که هر آرایه آن به نوبه خود، یک ماتریس است.

[2] Computational Graphs
[3] TensorBoard
[4] Application Programming Interface (API)

تنسور_فلو چیست؟

تنسور–فلو یک کتابخانه نرم‌افزاری منبع باز از گوگل است که به طور گسترده برای محاسبات عددی استفاده می‌شود. این کتابخانه یکی از پرکاربردترین کتابخانه‌ها برای ساخت مدل‌های یادگیری عمیق است. این کتابخانه بسیار مقیاس‌پذیر است و بر روی چندین پلتفرم مانند ویندوز، لینوکس، macOS و اندروید اجرا می‌شود. در ابتدا توسط محققان و مهندسان تیم Google Brain توسعه داده شد. و همانطور که گفته شد از Keras به عنوان واسط برنامه‌نویسی کاربری[۱] (**API**) سطح بالای خود استفاده می‌کند.

تنسور–فلو از اجرا بر روی همه چیز، از جمله **CPU**ها، **GPU**ها، **TPU**ها، که واحدهای پردازش تنسور هستند، و سکوها (پلتفرم‌های) موبایل و تعبیه‌شده[۲] پشتیبانی می‌کند. تنسور–فلو به دلیل معماری انعطاف پذیر و سهولت استقرار، به یک کتابخانه محبوب در بین بسیاری از محققان و دانشمندان برای ساخت مدل‌های یادگیری عمیق تبدیل شده است.

در تنسور–فلو، هر عمل محاسباتی با یک گرافِ جریانِ داده نشان داده می‌شود که به عنوان **گراف محاسباتی** نیز شناخته می‌شود، جایی که **گره‌ها**، عملیاتی مانند جمع یا ضرب و **یالها**، تنسورها را نشان می‌دهند. البته می‌توان گرافهای جریان داده را نیز به اشتراک گذاشت و بر روی بسیاری از سیستم عامل‌های مختلف اجرا کرد. تنسور–فلو یک ابزار تجسم به نام **تنسور–بُرد** برای تجسم گرافهای جریان داده ارائه می‌دهد.

تنسور–فلو ۲.۰ آخرین نسخه تنسور–فلو است. در فصل‌های آینده، از تنسور–فلو ۲.۰ برای ساخت مدل‌های یادگیری

---

[۱] واسط برنامه نویسی کاربردی، یا رابط یا میانجی برنامه کاربردی، رابط برنامه‌نویسی کاربردی بوده که مانند پل ارتباطی بین دو نرم‌افزار عمل می‌کند و به برنامه‌نویسان این امکان را داده تا با استفاده از توابع و دستورات مشخص از قابلیت‌های یک سیستم یا برنامه استفاده کنند. برای مثال، با یکی از انواع API به داده‌های یک وب‌سرویس یا خدمات یک پلتفرم ابری وصل شوند. در واقع، به‌جای اینکه خودتان صفر تا صد یک سرویس را کدنویسی کنید، با استفاده از API می‌توانید به سرویس یا سروری وصل شده و اطلاعات آن را دریافت کنید. API شاه‌کلیدی است که با آن می‌توان قفل درهای آهنی برنامه‌نویسی را باز کرده و راه مخفی ارتباط با سرویس‌های مختلف را کشف کرد. تا سال ۲۰۱۸، بیش از ۱۱۰ هزار API معرفی شد. تخمینی که تا سال ۲۰۲۳ وجود دارد، این رقم را به ۵۰۰ هزار API رسانده است. این آمار، رشد چشمگیر محبوبیت API را میان شرکت‌ها و سازمان‌ها نشان می‌دهد. تکنولوژی که توانسته باعث بهبود هرچه سریع‌تر توسعه نرم‌افزار و ارتباط بین سرویس شود. API مانند یک مترجم بین دو نفر با زبان‌های مختلف عمل می‌کند. فرض کنید یکی از افراد به زبان انگلیسی (فرستنده درخواست) و دیگری به زبان فارسی (گیرنده درخواست) صحبت می‌کند. API به‌عنوان مترجم میان این دو، تنها راه ارتباطی آن‌ها برای برقراری ارتباط بین آن‌ها است.

[۲] Embedded Platforms

تقویتی عمیق استفاده خواهیم کرد. با این حال، درک نحوه عملکرد تنسور–فلو ۱.X مهم است. بنابراین، ابتدا استفاده از تنسور–فلو ۱.X را یاد خواهیم گرفت و سپس تنسور–فلو ۲.۰ را بررسی خواهیم کرد.

شما می‌توانید تنسور–فلو را به راحتی از طریق `pip` فقط با تایپ دستور زیر در ترمینال خود نصب کنید:

```
pip install tensorflow==1.13.1
```

ما می‌توانیم نصب موفقیت آمیز تنسور–فلو را با اجرای دستور ساده زیر بررسی کنیم: **سلام تنسور–فلو!**

```
Hello TensorFlow!
```

```
import tensorflow as tf

hello = tf.constant("Hello TensorFlow!")
sess = tf.Session()
print(sess.run(hello))
```

برنامه بالا باید `Hello TensorFlow!` را چاپ کند. اگر خطایی دریافت کردید، احتمالا تنسور–فلو را به درستی نصب نکرده‌اید.

درک گرافها و کار_دوره‌های محاسباتی

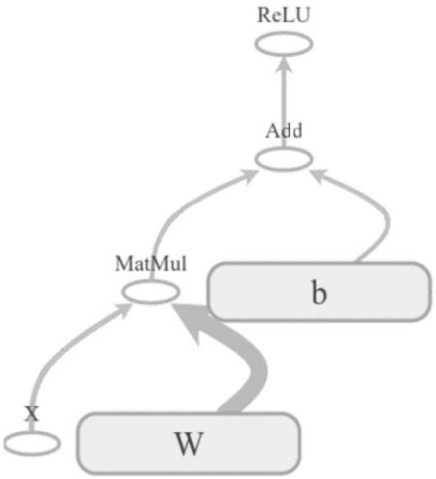
همانطور که آموختیم، هر محاسباتی در تنسور–فلو با یک گراف محاسباتی نشان داده می‌شود. آنها از چندین گره و یال تشکیل شده اند که **گره‌ها** عملیات ریاضی مانند جمع و ضرب هستند و **یالها**، تنسور هستند. نمودارها یا گرافهای محاسباتی در بهینه‌سازی منابع و ارتقا عملکرد محاسبات توزیع شده[۱] بسیار کارآمد هستند.

یک گراف محاسباتی شامل چندین عملیات تنسور–فلو است که بصورت گره‌ها در آن گراف چیده شده‌اند

زمانی که روی ساخت یک شبکه عصبی واقعا پیچیده کار می‌کنیم، یک گراف محاسباتی به ما کمک می‌کند تا معماری

[۱] Promote Distributed Computing

شبکه را درک کنیم. به عنوان مثال، بیایید یک لایه ساده را در نظر بگیریم، $h = \text{Relu}(WX + b)$ گراف محاسباتی آن به صورت زیر نشان داده می‌شود:



شکل ۸.۱: گراف (یا نمودار) محاسباتی $h = \text{Relu}(WX + b)$

دو نوع وابستگی در گراف محاسباتی وجود دارد که وابستگی مستقیم و غیرمستقیم نامیده می‌شوند. فرض کنید گره b را داریم که ورودی آن به خروجی گره a بستگی دارد؛ این نوع وابستگی را **وابستگی مستقیم** نامند، همانطور که در کد زیر نشان داده شده است:

```
a = tf.multiply(8,5)
b = tf.multiply(a,1)
```

هنگامی‌که گره b برای ورودی خود به گره a وابسته نیست، همانطور که در کد زیر نشان داده شده است، **وابستگی غیرمستقیم**[۱] نامیده می‌شود.

```
a = tf.multiply(8,5)
b = tf.multiply(4,3)
```

بنابراین، اگر بتوانیم این وابستگی‌ها را درک کنیم، می‌توانیم محاسبات مستقل را در منابع موجود، توزیع کرده و زمان محاسبات را کاهش دهیم. هر زمان که تنسور–فلو را وارد می‌کنیم، یک گراف پیش‌فرض به طور خودکار ایجاد می‌شود

[۱] indirect dependency

و تمام گره‌هایی که ایجاد می‌کنیم با گراف پیش‌فرض مرتبط هستند. ما همچنین می‌توانیم به جای استفاده از گراف پیش‌فرض، گرافهای خودمان را ایجاد کنیم و این کار به هنگام ساخت چندین مدل که به یکدیگر وابسته نیستند، بسیار مفید است.

گراف تنسور–فلو را می‌توان با دستور `tf.Graph()` ایجاد کرد.

```
graph = tf.Graph()

with graph.as_default():
     z = tf.add(x, y, name='Add')
```

اگر بخواهیم گراف پیش‌فرض را پاک کنیم (یعنی اگر بخواهیم متغیرها و عملیات تعریف شده قبلی را در گراف پاک کنیم)، می‌توانیم این کار را با استفاده از `tf.reset_default_graph()` انجام دهیم.

کار–دوره (یا جلسه)[1]

همانطور که در بخش قبلی ذکر شد، یک گراف محاسباتی با وجود عملیات بر روی گره‌ها و تنسورها بر روی یالهای آن ایجاد می‌شود. حال برای اجرایی کردن گراف از کار–دوره (یا جلسه) تنسور–فلو استفاده می‌کنیم.

همانطور که در کد زیر نشان داده شده است، یک کار–دوره (جلسه) تنسور–فلو را می‌توان با استفاده از دستور `tf.Session()` ایجاد کرد:

```
sess = tf.Session()
```

پس از ایجاد جلسه، می‌توانیم گراف خود را با استفاده از روش `sess.run()` اجرا کنیم.

هر نوع محاسبات در تنسور–فلو با یک گراف محاسباتی نشان داده می‌شود، بنابراین ما باید یک گراف محاسباتی برای همه چیز اجرا کنیم. یعنی برای محاسبه هر چیزی در تنسور–فلو، باید یک کار–دورهٔ تنسور–فلو ایجاد کنیم.

[1] Sessions

بیایید کد زیر را برای ضرب دو عدد اجرا کنیم:

```
a = tf.multiply(3,3)
print(a)
```

به جای چاپ عدد ۹، یک شی یا موجودیت تنسور–فلو[۱] را چاپ می‌کند.

```
Tensor("Mul:0", shape=(), dtype=int32).
```

همانطور که قبلا بحث کردیم، هر زمان که تنسور–فلو را وارد می‌کنیم، یک گراف محاسباتی پیش‌فرض به طور خودکار ایجاد شده و همه گره‌ها به گراف متصل می‌شوند. از این رو، وقتی a را چاپ می‌کنیم، فقط شی تنسور–فلو را برمی‌گرداند زیرا مقدار a هنوز محاسبه نشده است، و گراف محاسباتی هنوز اجرا نشده است.

برای اجرای گراف، باید جلسه تنسور–فلو را به شرح زیر مقداردهی اولیه و اجرا کنیم.:

```
a = tf.multiply(3,3)
with tf.Session as sess:
    print(sess.run(a))
```

حال، کد قبلی، عدد ۹ را چاپ می‌کند.

اکنون که با جلسات (کار–دوره‌ها) آشنا شدیم، در بخش بعدی با متغیرها، ثابت‌ها و جایبان (نگهدارنده مکان)[۲] آشنا خواهیم شد.

متغیرها، ثابت‌ها و جای‌بانها (قابها)

متغیرها، ثابت‌ها و جایبان‌ها عناصر اساسی تنسور–فلو هستند. با این حال، همیشه بین این سه سردرگمی‌وجود دارد. بیایید هر عنصر را یکی یکی بررسی کنیم و تفاوت بین آنها را بیاموزیم.

---

[۱] TensorFlow object
[۲] placeholder

متغیرها

متغیرها ظروفی هستند که برای ذخیره مقادیر استفاده می‌شوند. از متغیرها به عنوان ورودی به چندین عملیات دیگر در یک گراف محاسباتی استفاده می‌شوند. همانطور که در کد زیر نشان می‌دهد، یک متغیر را می‌توان با استفاده از تابع `tf.Variable()` ایجاد کرد:

```
x = tf.Variable(13)
```

بیایید با استفاده از `tf.Variable()` متغیری به نام `W` بصورت زیر ایجاد کنیم:

```
W = tf.Variable(tf.random_normal([500, 111], stddev=0.35),
name="weights")
```

همانطور که در کد قبلی مشاهده می‌کنید، ما با استفاده از مقادیر تصادفی از یک توزیع نرمال با انحراف استاندارد ۰.۳۵، یک متغیر `W`، ایجاد کردیم.

هدف از وجود پارامتر `name` در عبارت `tf.Variable()` چیست؟

از این پارامتر برای تنظیم نام متغیر در گراف محاسباتی استفاده می‌شود. بنابراین، در کد قبلی، پایتون متغیر را به صورت `W` ذخیره می‌کند اما در گراف تنسور–فلو، به عنوان وزن `weights` ذخیره می‌شود.

پس از تعریف یک متغیر، باید تمام متغیرهای گراف محاسباتی را مقداردهی اولیه کنیم. این کار را می‌توان با استفاده از `tf.global_variables_initializer()` انجام داد.

هنگامی‌که یک جلسه (کار–دوره) ایجاد می‌کنیم، ما عملیات مقداردهی اولیه را اجرا می‌کنیم که به معنای مقداردهی اولیه تمام متغیرهای تعریف شده است و تنها پس از آن است که می‌توانیم عملیات دیگر را اجرا کنیم، همانطور که در کد زیر نشان داده شده است:

```
x = tf.Variable(1212)
init = tf.global_variables_initializer()

with tf.Session() as sess:
  sess.run(init)
  print(sess.run(x))
```

ثوابت

ثابت‌ها، برخلاف متغیرها، نمی‌توانند مقادیر خود را تغییر دهند. یعنی ثابت‌ها، تغییرناپذیر هستند. هنگامی‌که مقادیر به آنها اختصاص داده می‌شود، نمی‌توان آنها را در طول برنامه تغییر داد. همانطور که کد زیر نشان می‌دهد، می‌توانیم با استفاده از دستور `tf.constant()` ثوابت را ایجاد کنیم:

```
x = tf.constant(13)
```

جای‌بانها و دیکشنری‌های تغذیه

در جایی که فقط نوع و بُعد را تعریف می‌کنیم، ما می‌توانیم جایبان‌ها را به عنوان متغیرها در نظر بگیریم، اما مقداری را به آنها اختصاص نمی‌دهیم. مقادیر جایبان‌ها در زمان اجرا، تغذیه می‌شوند. ما داده‌ها را با استفاده از جایبان‌ها به گرافهای محاسباتی تغذیه می‌کنیم. جایبان‌ها بدون هیچ مقداری تعریف می‌شوند.

یک جایبان را می‌توان با استفاده از دستور `tf.placeholder()` تعریف کرد. جایبان‌ها یک آرگومان اختیاری به نام `shape` دارند که ابعاد داده‌ها را نشان می‌دهد. اگر `shape` روی `None` تنظیم شده باشد، می‌توانیم داده‌ها را با هر اندازه‌ای در زمان اجرا تغذیه کنیم. یک جایبان را می‌توان به صورت زیر تعریف کرد:

```
x = tf.placeholder("float", shape=None)
```

به بیان ساده، ما از `tf.variable` برای ذخیره داده‌های خود و از `tf.placeholder` برای اخذ داده‌های خارجی استفاده می‌کنیم.

بیایید یک مثال ساده را برای درک بهتر جایبان‌ها در نظر بگیریم:

```
x = tf.placeholder("float", None)
y = x+3

with tf.Session() as sess:
    result = sess.run(y)
    print(result)
```

اگر کد قبلی را اجرا کنیم، خطا برمی‌گرداند زیرا ما در حال تلاش برای محاسبه y هستیم، جایی که y=x+3 و x یک نگهدارنده مکان (جایبان) است که مقدار آن اختصاص داده نشده است. همانطور که یاد گرفتیم، مقادیر برای جایبان‌ها در زمان اجرا اختصاص داده می‌شود. ما مقادیر نگهدارنده مکان (جایبان) را با استفاده از پارامتر feed_dict اختصاص می‌دهیم. پارامتر feed_dict اساسا یک فرهنگ لغت است که در آن کلید، نشاندهنده نام جایبان و مقدار آن، نشان دهنده مقدار جایبان است.

همانطور که در کد زیر مشاهده می‌کنید ، ما جایبان را بصورت feed_dict = {x:5} را تنظیم کردیم، که به این معنی است که مقدار x یعنی جایبان، معادل ۵ است:

```
with tf.Session() as sess:
    result = sess.run(y, feed_dict={x: 5})
    print(result)
```

کد قبلی ۸.۰ را برمی‌گرداند.

همین بود! در بخش بعدی باتنسور–بُرد آشنا خواهیم شد.

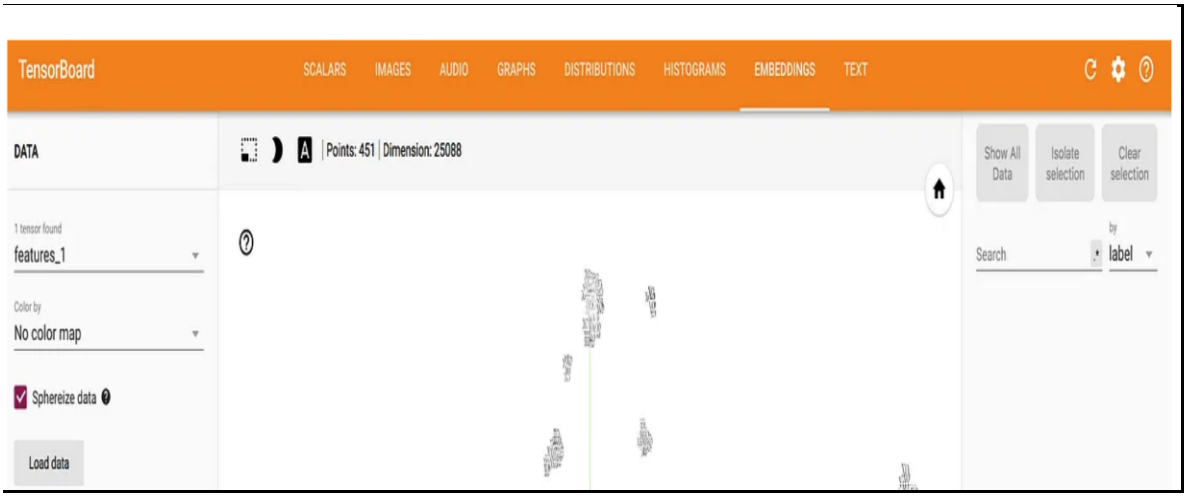
معرفی تنسور–بُرد

تنسور–بُرد، ابزار تجسم تنسور–فلو است که می‌توان از آن برای تجسم گراف محاسباتی استفاده کرد. همچنین می‌توان

از آن برای ترسیم سنجه‌ها یا معیارهای کمّی مختلف و نتایج چندین محاسبات میانی استفاده کرد. رفع اشکالات در شبکه، به هنگامی که داریم بر روی شبکه عصبی عمیق واقعی کار می‌کنیم، کاری گیج کننده می‌شود. بنابراین، اگر بتوانیم گراف محاسباتی را درتنسور–بُرد تجسم کنیم، می‌توانیم به راحتی چنین مدل‌های پیچیده‌ای را درک کرده، آنها را اشکال زدایی و بهینه کنیم. تنسور–بُرد همچنین از اشتراک گذاری پشتیبانی می‌کند.

همانطور که شکل ۸.۲ نشان می‌دهد، پنل تنسور–بُرد از چندین تب (برگه یا زبانه) تشکیل شده است:

TensorBoard	اسکالرها	تصاویر	صدا	گرافها	توزیع‌ها	هیستوگرام‌ها	تعبیه‌شده‌ها



شکل ۸.۲: تنسور–بُرد

زبانه‌ها یا (برگه‌ها)کاملا خود توضیحی هستند. برگهٔ SCALARS اطلاعات مفیدی در مورد متغیرهای اسکالر که در برنامه خود استفاده می‌کنیم نشان می‌دهد. به عنوان مثال، نشان می‌دهد که چگونه مقدار یک متغیر اسکالر به نام `loss` ضرر در چندین تکرار تغییر می‌کند.

زبانه یا برگه (تب) GRAPHS، گراف محاسباتی را نشان می‌دهد. زبانه‌های DISTRIBUTIONS و HISTOGRAMS توزیع یک متغیر را نشان می‌دهند. به عنوان مثال، توزیع وزن و هیستوگرام مدل ما را می‌توان

در زیر این زبانه‌ها مشاهده کرد. زبان EMBEDDINGS برای تجسم بردارهای با ابعاد بالا، مانند جاسازی کلمات (تعبیه) استفاده می‌شود.

بیایید یک گراف محاسباتی پایه ساخته و آن را در تنسور–بُرد تجسم کنیم. فرض کنید ما چهار مقدار ثابت داریم که به شرح زیر نشان داده شده است:

```
x = tf.constant(1,name='x')
y = tf.constant(1,name='y')
a = tf.constant(3,name='a')
b = tf.constant(3,name='b')
```

بیایید x و y و نیز a و b را در هم ضرب کرده و آنها را به عنوان prod1 و prod2 ذخیره کنیم، همانطور که در کد زیر نشان داده شده است:

```
prod1 = tf.multiply(x,y,name='prod1')
prod2 = tf.multiply(a,b,name='prod2')
```

حالا prod1 و prod2 را در با هم جمع کرده و مجموع را در sum ذخیره کنید:

```
sum = tf.add(prod1,prod2,name='sum')
```

اکنون، می‌توانیم همه این عملیات را در تنسور–بُرد تجسم کنیم. برای تجسم در تنسور–بُرد، ابتدا باید فایل‌های رویداد خود را ذخیره کنیم. این کار را می‌توان با استفاده از tf.summary.FileWriter() انجام داد. این کار، دو پارامتر مهم logdir و graph را می‌طلبد.

همانطور که از نام آن پیداست، logdir دایرکتوری را مشخص می‌کند که در آن می‌خواهیم گراف را ذخیره کنیم. همچنین graph مشخص می‌کند که کدام گراف را میخواهیم ذخیره کنیم:

```
with tf.Session() as sess:
    writer = tf.summary.FileWriter(logdir='./graphs',graph=sess.graph)
    print(sess.run(sum))
```

در کد قبلی، ./graphs دایرکتوری است که ما فایل رویداد خود را در آن ذخیره می‌کنیم و sess.graph گراف

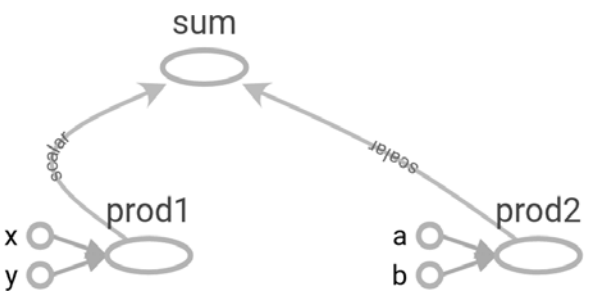
فعلی را در جلسه تنسور–فلوی ما مشخص می‌کند. بنابراین، ما گراف فعلی کار–دورهٔ (جلسه) تنسور–فلو را در دایرکتوری گرافها ذخیره می‌کنیم.

برای راه اندازی تنسور–بُرد، به ترمینال خود بروید، دایرکتوری کاری را پیدا کرده و دستور زیر را تایپ کنید:

```
tensorboard --logdir=graphs --port=8000
```

پارامتر `logdir` دایرکتوری را نشان می‌دهد که فایل رویداد در آن ذخیره می‌شود و `port` شماره پورت است. هنگامی‌که دستور قبلی را اجرا کردید، مرورگر خود را باز کنید و `http://localhost:8000/` را تایپ کنید.

در پنل تنسور–بُرد، در زیر تب **GRAPHS**، می‌توانید گراف محاسباتی را مشاهده کنید:



شکل ۸.۳: گراف محاسباتی

همانطور که متوجه شدید، تمام عملیاتی که ما تعریف کرده‌ایم به وضوح در گراف نشان داده شده است.

ایجاد یك محدوده یا حوزهٔ نام[1]

محدوده‌سازی[2] برای کاهش پیچیدگی استفاده می‌شود و به ما کمک می‌کند تا با گروه‌بندی گره‌های مرتبط با هم،

[1] Name Scope
[2] Scoping

اسکوپ، محدوده‌ای است که متغیرها در آن در دسترس هستند. متغیرها می‌توانند محلی، سراسری و حتی غیرمحلی باشند. به عبارت دیگر، **اسکوپ** را می‌توانیم به‌عنوان مرزهایی در نظر بگیریم که مشخص می‌کنند کدام بخش از کد به کدام متغیرها دسترسی دارند.
پایتون از قوانین خاصی به نام LEGB برای تعیین محدوده دسترسی متغیرها استفاده می‌کند. نام این قانون از ابتدای نام چهار مورد زیر ایجاد شده است:

مدل خود را بهتر درک کنیم. داشتن محدوده یا حوزه نام به ما کمک می‌کند تا عملیات مشابه را در یک گراف گروه‌بندی کنیم. این کار زمانی مفید است که ما در حال ساخت یک معماری پیچیده هستیم. محدوده را می‌توان با استفاده از tf.`tf.name_scope()` ایجاد کرد. در مثال قبلی، ما دو عملیات ضرب و جمع را انجام دادیم. ما به سادگی می‌توانیم آنها را در دو محدودهٔ نام مختلف به عنوان `Product` و `sum` گروه‌بندی کنیم.

در بخش قبلی دیدیم که چگونه `prod1` و `prod2` ضرب را انجام داده و نتیجه را محاسبه می‌کنند. ما یک محدودهٔ نام به اسم `Product` تعریف کرده و عملیات `prod1` و `prod2` را گروه‌بندی می‌کنیم، همانطور که در کد زیر نشان داده شده است:

```
with tf.name_scope("Product"):
    with tf.name_scope("prod1"):
        prod1 = tf.multiply(x,y,name='prod1')

    with tf.name_scope("prod2"):
        prod2 = tf.multiply(a,b,name='prod2')
```

اکنون، محدودهٔ نام را برای `sum` تعریف کنید:

```
with tf.name_scope("sum"):
    sum = tf.add(prod1,prod2,name='sum')
```

فایل را در دایرکتوری `graphs` ذخیره کنید:

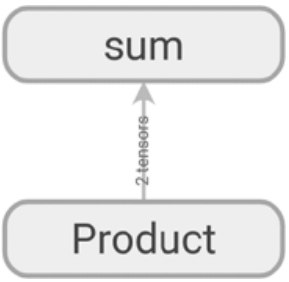
```
with tf.Session() as sess:
    writer = tf.summary.FileWriter('./graphs', sess.graph)
    print(sess.run(sum))
```

گراف را در تنسور–بُرد تجسم کنید:

- **Local** (محلی): متغیرهایی که درون یک تابع پایتون تعریف می‌شوند.
- **Enclosing** (محصور) متغیرهایی که درون یک تابع تو در تو تعریف شده‌اند.
- **Global** (سراسری): متغیرهایی که به‌طور سراسری و بیرون از توابع تعریف می‌شوند.
- **Built-in** (داخلی): متغیرهای داخلی پایتون که به‌طور پیش‌فرض وجود دارند.

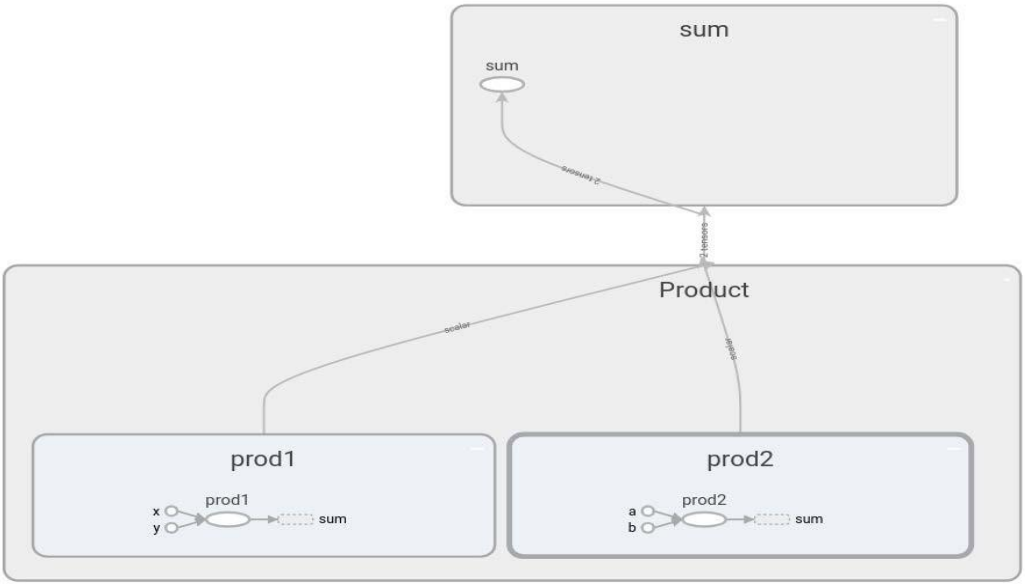
```
tensorboard --logdir=graphs --port=8000
```

همانطور که ممکن است متوجه شوید، اکنون ما فقط دو گره داریم، **sum** و **Product**



شکل ۸.۴: یک گراف محاسباتی

هنگامی‌که روی گره‌ها دوبار کلیک می‌کنیم، می‌توانیم ببینیم که محاسبات چگونه اتفاق می‌افتد. همانطور که می‌بینید، **گره‌های prod1** و **prod2** در محدوده **ضرب** گروه‌بندی می‌شوند و نتایج آنها به **گره جمع** ارسال شده و در آنجا جمع می‌شوند. در زیر می‌توانید ببینید که **گره‌های prod1** و **prod2** چگونه مقدار خود را محاسبه می‌کنند:



شکل ۸.۵: یک گراف محاسباتی با جزئیات

گراف قبلی فقط یک مثال ساده است. هنگامی‌که ما روی یک پروژه پیچیده با عملیات زیاد کار می‌کنیم، محدودهٔ نام

به ما کمک می‌کند تا عملیات مشابه را با هم گروه‌بندی کنیم و ما را قادر می‌سازد تا گراف محاسباتی را بهتر درک کنیم.

اکنون که با تنسور–فلو آشنا شدیم، در بخش بعدی، بیایید ببینیم چگونه با استفاده از تنسور–فلو طبقه بندی ارقام دستنویس بسازیم.

طبقه بندی اعداد دست‌نویس با استفاده از تنسور_فلو

با کنار هم قرار دادن تمام مفاهیمی‌که تاکنون آموخته ایم، خواهیم دید که چگونه می‌توانیم از تنسور–فلو برای ساخت یک شبکه عصبی برای تشخیص اعداد دست‌نویس استفاده کنیم. اگر اخیرا در حوزه یادگیری عمیق کار کرده اید حتما با مجموعه داده **MNIST** مواجه شده اید. به آن [illegible] یادگیری عمیق گفته شده است. این مجموعه داده، شامل ۵۵,۰۰۰ نقطه داده از ارقام دستنویس (۰ تا ۹) است.

در این بخش خواهیم دید که چگونه می‌توانیم از شبکه عصبی خود برای تشخیص این ارقام دستنویس استفاده کنیم. برای این کار از تنسور–فلو و تنسور–بُرد استفاده خواهیم کرد.

ورود (فراخوان) کتابخانه‌های مورد نیاز

به عنوان اولین قدم، بیایید همه کتابخانه‌های مورد نیاز را وارد کنیم:

[1] *Hello World*

```
import warnings
warnings.filterwarnings('ignore')

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
tf.logging.set_verbosity(tf.logging.ERROR)

import matplotlib.pyplot as plt
%matplotlib inline
```

بارگذاری مجموعه داده

مجموعه داده را با استفاده از کد زیر بارگیری کنید:

```
mnist = input_data.read_data_sets("data/mnist", one_hot=True)
```

در کد قبلی، data/mnist به مکانی اشاره دارد که مجموعه داده **MNIST** را در آن ذخیره می‌کنیم و one_hot=True نشان می‌دهد که ما در حال رمزگذاری برچسب‌ها بصورت تک-نشان[1] یا هستیم (۰ تا ۹).

با اجرای کد زیر خواهیم دید که چه چیزی در داده‌های خود داریم:

```
print("No of images in training set {}".format(mnist.train.images.
shape))
print("No of labels in training set {}".format(mnist.train.labels.
shape))
```

```
print("No of images in test set {}".format(mnist.test.images.shape))
print("No of labels in test set {}".format(mnist.test.labels.shape))

No of images in training set (55000, 784)
No of labels in training set (55000, 10)
No of images in test set (10000, 784)
No of labels in test set (10000, 10)
```

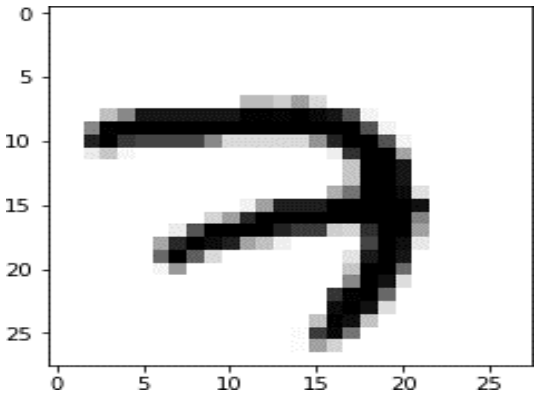
ما ۵۵،۰۰۰ تصویر در مجموعه آموزشی داریم، هر تصویر اندازه ۷۸۴ داشته و ۱۰ برچسب دارد که در واقع اعداد ۰ تا

[1] One-Hot

۹ هستند. به طور مشابه، ما ۱۰،۰۰۰ تصویر در مجموعه آزمایشی داریم. اکنون، ما یک تصویر ورودی را ترسیم می‌کنیم تا ببینیم چگونه به نظر می‌رسد:

```
img1 = mnist.train.images[0].reshape(28,28)
plt.imshow(img1, cmap='Greys')
```

بنابراین، تصویر ورودی ما به صورت زیر است:



شکل ۸.۶: تصویر رقم ۷ از مجموعه آموزشی

تعریف تعداد نورون‌ها در هر لایه

ما یک شبکه عصبی چهار لایه، با سه لایه پنهان و یک لایه خروجی خواهیم ساخت. از آنجایی که اندازه تصویر ورودی ۷۸۴ است، بنابر این ما `num_input` را روی ۷۸۴ تنظیم می‌کنیم و از آنجایی که ۱۰ رقم دستنویس (۰ تا ۹) داریم، ۱۰ نورون را در لایه خروجی تنظیم می‌کنیم. ما تعداد نورون‌ها را در هر لایه به شرح زیر تعریف می‌کنیم:

```
#number of neurons in input layer
num_input = 784

#num of neurons in hidden layer 1
num_hidden1 = 512

#num of neurons in hidden layer 2
num_hidden2 = 256

#num of neurons in hidden layer 3
num_hidden_3 = 128

#num of neurons in output layer
num_output = 10
```

تعریف متغیرها

همانطور که یاد گرفتیم، ابتدا باید جای‌بانهای `input` و `output` را تعریف کنیم. مقادیر جای‌بانها در زمان اجرا از طریق `feed_dict` وارد می‌شوند:

```
with tf.name_scope('input'):
    X = tf.placeholder("float", [None, num_input])

with tf.name_scope('output'):
    Y = tf.placeholder("float", [None, num_output])
```

از آنجایی که ما یک شبکه چهار لایه داریم، چهار وزن و چهار سویانه (یا سویش)[۱] داریم. ما وزن‌های خود را با ترسیم مقادیر از **توزیع نرمال قصیر (کوتاه شده)**[۲] با انحراف معیار ۰.۱ مقداردهی اولیه می‌کنیم. به یاد داشته باشید، ابعاد ماتریس وزن باید *تعداد نورون‌های لایه قبلی* × *تعداد نورون‌های لایه فعلی* باشد. به عنوان مثال، ابعاد ماتریس وزن w3 باید *تعداد نورون‌ها در لایه پنهان ۲* × *تعداد نورون‌ها در لایه پنهان ۳* باشد.

[۱] Biase
[۲] Truncated Normal Distribution

ما اغلب تمام وزن‌ها را در یک فرهنگ لغت به شرح زیر تعریف می‌کنیم:

```
with tf.name_scope('weights'):

 weights = {
 'w1': tf.Variable(tf.truncated_normal([num_input, num_hidden1],
stddev=0.1),name='weight_1'),
 'w2': tf.Variable(tf.truncated_normal([num_hidden1, num_hidden2],
```

```
stddev=0.1),name='weight_2'),
 'w3': tf.Variable(tf.truncated_normal([num_hidden2, num_hidden_3],
stddev=0.1),name='weight_3'),
 'out': tf.Variable(tf.truncated_normal([num_hidden_3, num_output],
stddev=0.1),name='weight_4'),
 }
```

شکل سویش (سویانه) باید تعداد نورون‌ها در لایه فعلی باشد. به عنوان مثال، بُعد سویش b2، تعداد نورون‌ها در لایه پنهان ۲ است. ما مقدار سویش را به عنوان یک ثابت تعیین می‌کنیم. فرضاً مقدار ۰.۱ در تمام لایه‌ها:

```
with tf.name_scope('biases'):

    biases = {
        'b1': tf.Variable(tf.constant(0.1, shape=[num_
hidden1]),name='bias_1'),
        'b2': tf.Variable(tf.constant(0.1, shape=[num_
hidden2]),name='bias_2'),
        'b3': tf.Variable(tf.constant(0.1, shape=[num_
hidden_3]),name='bias_3'),
        'out': tf.Variable(tf.constant(0.1, shape=[num_
output]),name='bias_4')
    }
```

انتشار رو به جلو

اکنون عملیات انتشار رو به جلو (پیشرو) را تعریف می‌کنیم. ما از تابع فعالسازی ReLU در همه لایه‌ها استفاده خواهیم

کرد. در لایه‌های آخر، تابع فعالسازی سیگموئید را اعمال می‌کنیم، همانطور که در کد زیر نشان داده شده است:

```
with tf.name_scope('Model'):

    with tf.name_scope('layer1'):
        layer_1 = tf.nn.relu(tf.add(tf.matmul(X, weights['w1']),
biases['b1']) )

    with tf.name_scope('layer2'):
        layer_2 = tf.nn.relu(tf.add(tf.matmul(layer_1, weights['w2']),
biases['b2']))

    with tf.name_scope('layer3'):
        layer_3 = tf.nn.relu(tf.add(tf.matmul(layer_2, weights['w3']),
biases['b3']))
```

```
    with tf.name_scope('output_layer'):
        y_hat = tf.nn.sigmoid(tf.matmul(layer_3, weights['out']) +
biases['out'])
```

محاسبه زیان و پس انتشار

در مرحله بعد، تابع ضرر خود را تعریف خواهیم کرد. ما از آنتروپی متقابل softmax به عنوان تابع ضرر خود استفاده خواهیم کرد. تنسور–فلو، تابع `tf.nn.softmax_cross_entropy_with_logits()` را برای محاسبه ضرر آنتروپی متقابل بیشنه–نرم فراهم می‌کند. دو پارامتر را به عنوان ورودی، `logits` و `labels` می‌گیرد:

۱. پارامتر `logits` عملا مقادیر `logits` پیش‌بینی شدهٔ شبکه ما را مشخص می‌کند؛ به عنوان مثال، `y_hat`

۲. پارامتر `labels` در عمل برچسب‌های واقعی را مشخص می‌کند؛ به عنوان مثال، برچسب‌های واقعی Y

میانگین تابع ضرر را با استفاده از `tf.reduce_mean()` در نظر می‌گیریم:

```
with tf.name_scope('Loss'):
        loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_
logits(logits=y_hat,labels=Y))
```

اکنون، ما باید با استفاده از پس–انتشار، تابع ضرر را به حداقل برسانیم. نگران نباشید! ما مجبور نیستیم مشتقات همه وزن‌ها را به صورت دستی محاسبه کنیم. در عوض، می‌توانیم از بهینه‌ساز تنسور–فلو استفاده کنیم:

```
learning_rate = 1e-4
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

دقت محاسباتی

ما دقت مدل خود را به شرح زیر محاسبه می‌کنیم:

- پارامتر `y_hat` نشاندهنده احتمال پیش‌بینی شده برای هر کلاس از مدل ما است. از آنجایی که ما ۱۰ کلاس داریم، ۱۰ احتمال خواهیم داشت. اگر احتمال در موقعیت ۷ زیاد باشد، به این معنی است که شبکه ما تصویر ورودی را به صورت رقم ۷ با احتمال بالا پیش‌بینی می‌کند. تابع `tf.argmax()` شاخص بزرگترین مقدار را برمی‌گرداند. بنابراین، `tf.argmax(y_hat,1)` شاخص عددی را می‌دهد که احتمال آن زیاد است. لذا، اگر احتمال در شاخص ۷ زیاد باشد، ۷ را برمی‌گرداند.

- پارامتر Y نشان دهنده برچسب‌های واقعی است و آنها مقادیر رمزگذاری شده تک–نشان (one-hot) هستند. یعنی پارامتر Y در همه جا از صفر تشکیل شده است به جز در موقعیت تصویر واقعی، جایی که مقدار آن ۱ است. به عنوان مثال، اگر تصویر ورودی ۷ باشد، Y در همه شاخص‌ها به جز شاخص ۷ که مقدار آن ۱ است، مقدار ۰ دارد. بنابراین، `tf.argmax(Y,1)` عدد ۷ را برمی‌گرداند زیرا در آنجا بیشترین احتمال یعنی ۱ داریم.

بنابراین، `tf.argmax(y_hat,1)` رقم پیش بینی شده و `tf.argmax(Y,1)` رقم واقعی را به ما می

تابع `tf.equal(x, y)` متغیرهای x و y را به عنوان ورودی می‌گیرد و ارزش درستی (x == y) را به صورت عنصر به عنصر و یا آرایه–وار برمی‌گرداند. بنابراین عبارت:

```
correct_pred = tf.equal(predicted_digit,actual_ digit)
```

وقتی دربردارنده True است که در آن ارقام واقعی و پیش‌بینی شده یکسان باشند، و وقتی شامل False است که در آن ارقام واقعی و پیش‌بینی شده یکسان نیستند. ما مقادیر **بولین** در correct_pred را با استفاده از **عملیات تبدیل**[۱] تنسور–فلو، tf.cast(correct_pred, tf.float32) به مقادیر شناور یا **اعشاری** تبدیل می‌کنیم. پس از تبدیل آنها به مقادیر اعشاری، میانگین را با استفاده از tf.reduce_mean() می‌گیریم.

بنابراین، tf.reduce_mean(tf.cast(correct_pred, tf.float32)) میانگین پیش‌بینی‌های صحیح را به ما می‌دهد:

```
with tf.name_scope('Accuracy'):

    predicted_digit = tf.argmax(y_hat, 1)
    actual_digit = tf.argmax(Y, 1)

    correct_pred = tf.equal(predicted_digit,actual_digit)
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

خلاصه‌سازی

ما همچنین می‌توانیم تجسم کنیم که چگونه ضرر و دقت مدل ما در طی چندین تکرار در تنسور–بُرد تغییر می‌کند. بنابراین، ما از tf.summary() برای دریافت خلاصه متغیر استفاده می‌کنیم. از آنجایی که **ضرر** و **دقت** متغیرهای اسکالر هستند، ما از عبارت tf.summary scalar() مانند زیر، استفاده می‌کنیم.

```
tf.summary.scalar("Accuracy", accuracy)
tf.summary.scalar("Loss", loss)
```

در مرحله بعد، تمام خلاصه‌های گراف خود را با استفاده از tf.summary.merge_all() ادغام می‌کنیم. ما این کار را به این دلیل انجام می‌دهیم که به هنگام داشتن خلاصه‌های زیاد، عملا اجرا و ذخیره آنها ناکارآمد می‌شود،

[۱] cast operation

بنابراین به جای چندین بار اجرای آنها، یک بار در جلسه (کار–دوره) خود، آنها را اجرا می‌کنیم:

```
merge_summary = tf.summary.merge_all()
```

آموزش مدل

اکنون زمان آن رسیده است که مدل خود را آموزش دهیم. همانطور که یاد گرفتیم، ابتدا باید همه متغیرها را مقداردهی اولیه کنیم:

```
init = tf.global_variables_initializer()
```

اندازه دسته، تعداد تکرارها و نرخ یادگیری را به شرح زیر تعریف کنید:

```
learning_rate = 1e-4
num_iterations = 1000
batch_size = 128
```

جلسه تنسور–فلو را شروع کنید:

```
with tf.Session() as sess:
```

همه متغیرها را مقداردهی اولیه کنید:

```
    sess.run(init)
```

فایل‌های رویداد را ذخیره کنید:

```
    summary_writer = tf.summary.FileWriter('./graphs', graph=tf.get_
default_graph()
```

مدل را برای تعدادی از تکرارها آموزش دهید:

```
    for i in range(num_iterations):
```

دسته‌ای از داده‌ها را با توجه به اندازهٔ دسته دریافت کنید:

```
        batch_x, batch_y = mnist.train.next_batch(batch_size)
```

شبکه را آموزش دهید:

```
        sess.run(optimizer, feed_dict={ X: batch_x, Y: batch_y})
```

ضرر و دقت را برای هر یکصدمین تکرار، چاپ کنید:

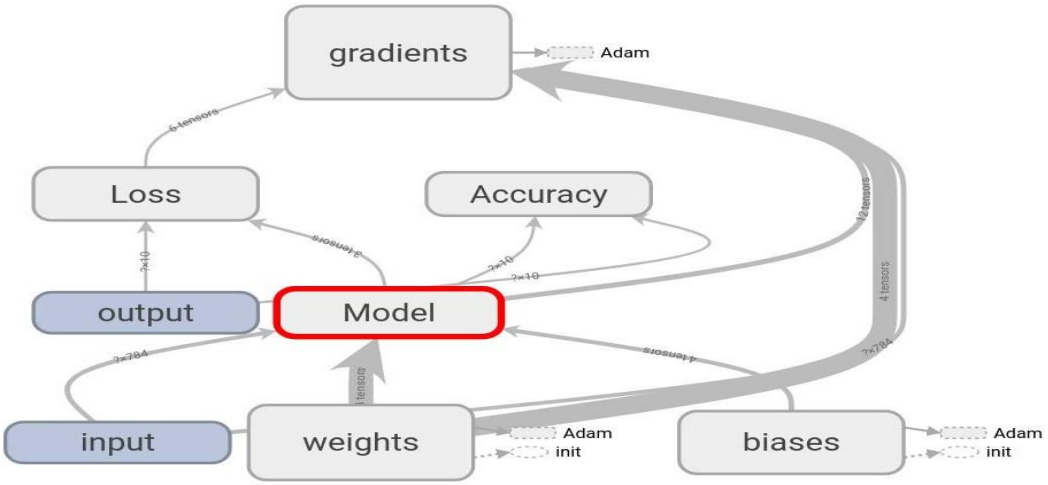
```
        if i % 100 == 0:

            batch_loss, batch_accuracy,summary = sess.run(
                [loss, accuracy, merge_summary],
                feed_dict={X: batch_x, Y: batch_y}
                )

            #store all the summaries
            summary_writer.add_summary(summary, i)

            print('Iteration: {}, Loss: {}, Accuracy: {}'.
format(i,batch_loss,batch_accuracy))
```

همانطور که ممکن است از خروجی زیر متوجه شوید، زیان کاهش می یابد و دقت در تکرارهای مختلف آموزشی افزایش می یابد:

```
Iteration: 0, Loss: 2.30789709091, Accuracy: 0.1171875
Iteration: 100, Loss: 1.76062202454, Accuracy: 0.859375
Iteration: 200, Loss: 1.60075569153, Accuracy: 0.9375
Iteration: 300, Loss: 1.60388696194, Accuracy: 0.890625
Iteration: 400, Loss: 1.59523034096, Accuracy: 0.921875
Iteration: 500, Loss: 1.58489584923, Accuracy: 0.859375
Iteration: 600, Loss: 1.51407408714, Accuracy: 0.953125
Iteration: 700, Loss: 1.53311181068, Accuracy: 0.9296875
Iteration: 800, Loss: 1.57677125931, Accuracy: 0.875
Iteration: 900, Loss: 1.52060437202, Accuracy: 0.9453125
```

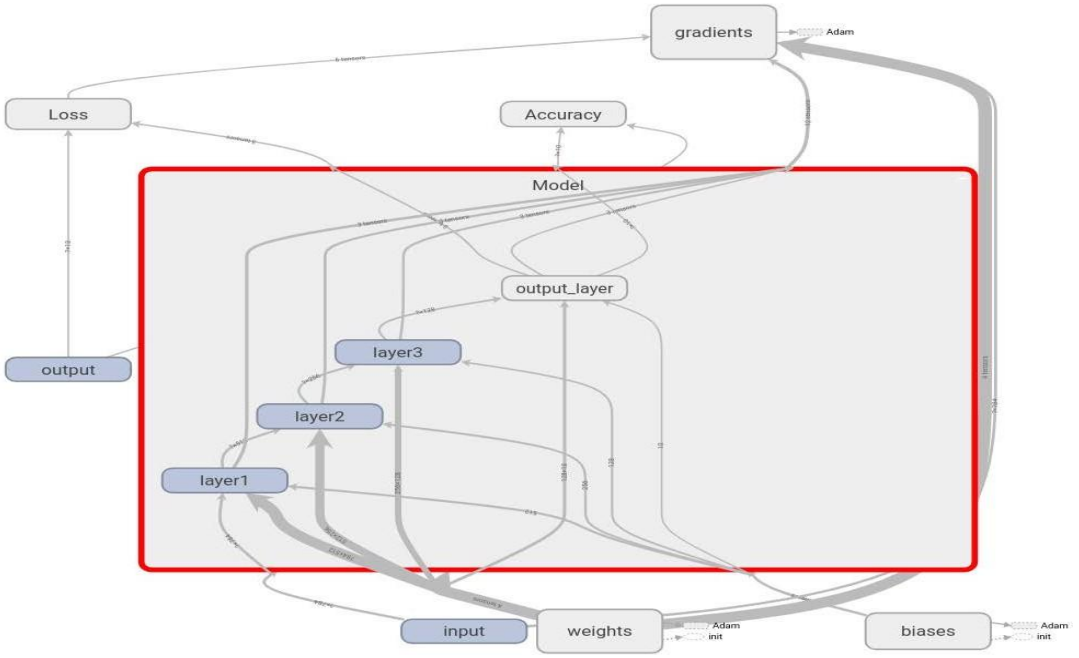
تجسم گرافها در تنسور_بُرد

همانند شکل ۸.۷، پس از آموزش، می‌توانیم گراف محاسباتی خود را در تنسور–بُرد تجسم کنیم، همانطور که می‌بینید، **مدل** ما **ورودی**، **اوزان** و **سویشها** را به عنوان درونداد می‌گیرد و برونداد را برمی‌گرداند. ما **زیان** و **دقت** را بر اساس خروجی مدل محاسبه می‌کنیم. ما با محاسبه **گرادیان‌ها** و به‌روزرسانی **اوزان**، مقدار **زیان** را به حداقل می‌رسانیم:



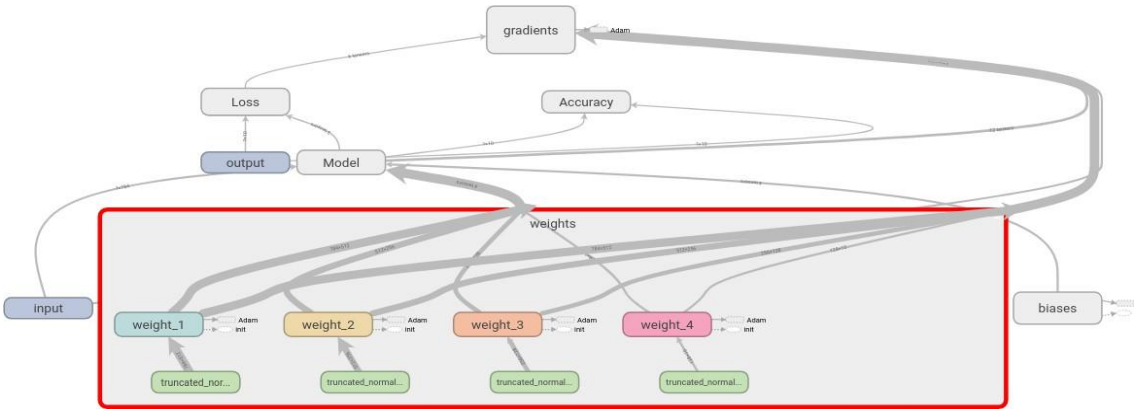
شکل ۸.۷: گراف محاسباتی

اگر دوبار کلیک کنیم و **مدل** را گسترش دهیم، می‌بینیم که سه لایه مخفی و یک لایه خروجی داریم:



تصویر ۸.۸: گسترش گره مدل (گرهی که مدل را نشان می‌دهد)

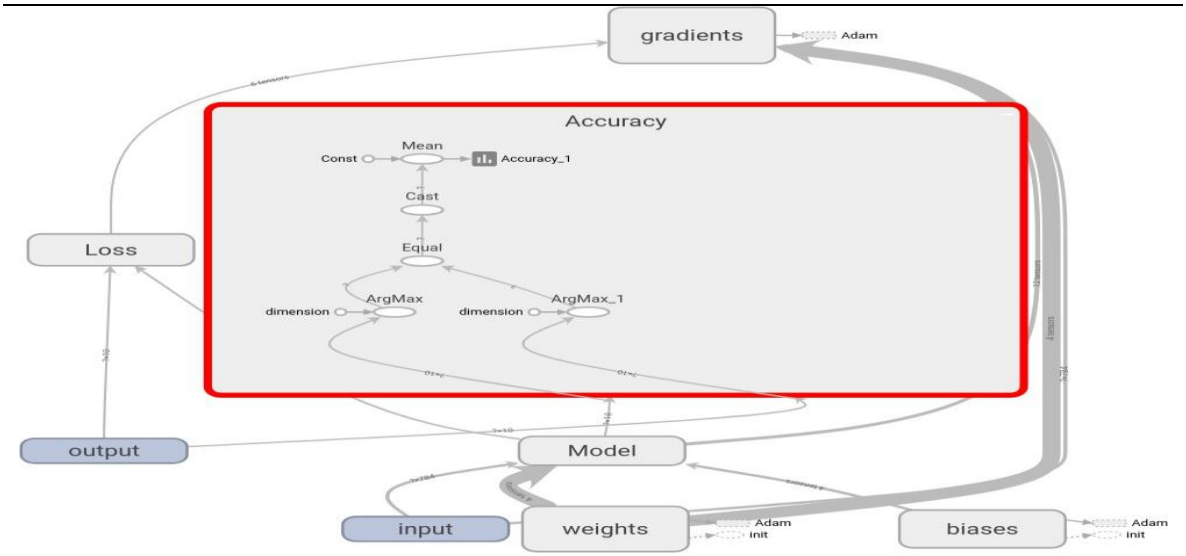
به طور مشابه، می‌توانیم دوبار کلیک کرده و هر گره را ببینیم. به عنوان مثال، اگر **اوزان** را باز کنیم، می‌توانیم ببینیم که چگونه چهار وزن با استفاده از توزیع نرمال کوتاه شده[1]، مقداردهی اولیه شده و چگونه با استفاده از بهینه‌ساز Adam به روز می‌شوند:



شکل ۸.۹: گسترش گره وزن‌ها

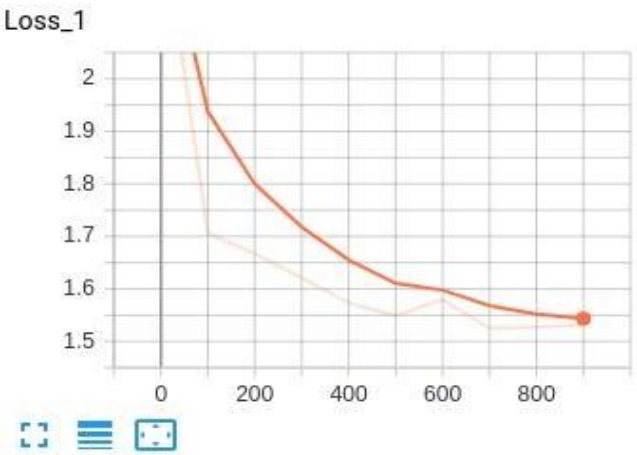
همانطور که یاد گرفتیم، گراف محاسباتی به ما کمک می‌کند تا بفهمیم در هر گره چه اتفاقی می‌افتد. با دوبار کلیک بر روی گره **دقت** می‌توانیم ببینیم که چگونه دقت، محاسبه می‌شود.

[1] Truncated Normal Distribution



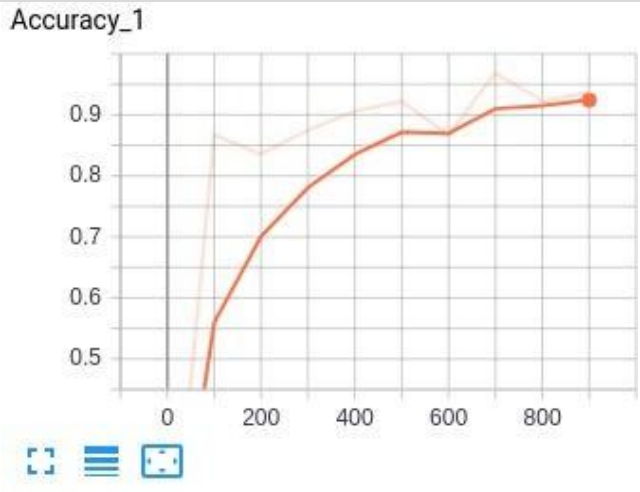
تصویر ۸.۱۰: گسترش گره دقت

به یاد داشته باشید که ما همچنین خلاصه‌ای از متغیرهای ضرر و دقت خود را ذخیره کرده ایم. ما می‌توانیم آنها را در زیر زبانه **SCALARS** در تنسور–بُرد پیدا کنیم. شکل ۸.۱۱ نشان می‌دهد که چگونه ضرر در تکرارها کاهش می‌یابد:



شکل ۸.۱۱: گراف تابع ضرر

شکل ۸.۱۲ نشان می‌دهد که چگونه دقت در تکرارها افزایش می‌یابد:



شکل ۸.۱۲ گراف دقت

همین و بس! در بخش بعدی با یکی دیگر از ویژگی‌های جالب تنسور–فلو به نام اجرای مشتاق آشنا می‌شویم.

معرفی اجرای مشتاق یا تند و تیز[۱]

اجرای تند و تیز در تنسور–فلو بیشتر پایتونی[۲] (به بیان عامه: اژدرسان یا افعی‌وار) است و امکان نمونه سازی سریع را فراهم می‌کند. برخلاف حالت گراف، که در آن هر بار که می‌خواهیم هر عملیاتی را انجام دهیم، باید یک گراف بسازیم، اجرای مشتاقانه از **پارادایم برنامه نویسی امری**[۳] پیروی می‌کند، جایی که هر عملیاتی را می‌توان بلافاصله انجام داد، بدون نیاز به ایجاد گراف، درست مانند پایتون. از این رو، با اجرای تند و تیز، می‌توانیم با جلسات و جای‌بانها خداحافظی کنیم. همچنین برخلاف مدل گراف، در اینجا، فرآیند اشکال‌زدایی با توجه به نمایش فوری خطای زمان–اجرا[۴]، آسانتر است. به عنوان مثال، در حالت گراف، برای محاسبه هر چیزی، جلسه را اجرا می‌کنیم. همانطور که در کد زیر نشان

---

۱ Eager Execution

۲ Pythonic

واژه **Python** به معنای افعی، غیبگو، اژدها و امثال آن نیز هست. و معانی دیگر آن: اسطوره یونان (مار عظیمی که توسط آپولو کشته می شود)، مار پیتون، اژدر مار، نیز می‌باشد لذا برای **Pythonic** معانی همچون: پیشگویانه، پیامبرانه، غیبگویانه، وحی آمیز، مانند اژدر مار، مانند مار بزرگ، برغمان سان، پایتون مانند، افعی وار هم ذکر شده است.

۳ Imperative Programming

۴ Immediate Runtime Error

داده شده است، برای ارزیابی مقدار z، باید جلسه تنسور–فلو را اجرا کنیم:

```
x = tf.constant(11)
y = tf.constant(11)
z = x*y

with tf.Session() as sess:
    print(sess.run(z))
```

با اجرای مشتاقانه، ما نیازی به ایجاد یک جلسه (کار–دوره) نداریم؛ ما به سادگی می‌توانیم z را محاسبه کنیم، درست مانند آنچه در پایتون انجام می‌دهیم. برای فعال کردن اجرای تند و تیز، کافی است تابع زیر را فراخوانی کنید:

`tf.enable_eager_execution()` function

```
x = tf.constant(11)
y = tf.constant(11)
z = x*y

print(z)
```

که موارد زیر را برمی‌گرداند:

```
<tf.Tensor: id=789, shape=(), dtype=int32, numpy=121>
```

برای به دست آوردن مقدار خروجی ، می‌توانیم موارد زیر را چاپ کنیم:

```
z.numpy()

121
```

اگرچه اجرای مشتاقانه، پارادایم برنامه نویسی امری را امکان پذیر می‌کند، اما در این کتاب، بیشتر مثال‌ها را در حالت غیرمشتاق بررسی می‌کنیم تا الگوریتم‌ها را از ابتدا بهتر درک کنیم. در بخش بعدی نحوه انجام عملیات ریاضی با استفاده از تنسور–فلو را خواهیم دید.

عملیات ریاضی در تنسور–فلو

اکنون، برخی از عملیات‌ها در تنسور–فلو را با استفاده از اجرا در مُد مشتاق (تند و تیز) بررسی خواهیم کرد

```
x = tf.constant([1., 2., 3.])
y = tf.constant([3., 2., 1.])
```

بیایید با چند عملیات محاسباتی اساسی شروع کنیم. از `tf.add` برای اضافه کردن دو عدد استفاده کنید:

```
sum = tf.add(x,y)
sum.numpy()

array([4., 4., 4.], dtype=float32)
```

تابع `tf.subtract` برای یافتن تفاوت بین دو عدد استفاده می‌شود:

```
difference = tf.subtract(x,y)
difference.numpy()

array([-2.,  0.,  2.], dtype=float32)
```

تابع `tf.multiply` برای ضرب دو عدد استفاده می‌شود:

```
product = tf.multiply(x,y)
product.numpy()

array([3., 4., 3.], dtype=float32)
```

دو عدد را با استفاده از `tf.divide` تقسیم کنید:

```
division = tf.divide(x,y)
division.numpy()

array([0.33333334, 1.        , 3.        ], dtype=float32)
```

ضرب نقطه‌ای را می‌توان به صورت زیر محاسبه کرد:

```
dot_product = tf.reduce_sum(tf.multiply(x, y))
dot_product.numpy()

10.0
```

در مرحله بعد ، بیایید شاخص حداقل و حداکثر عناصر را پیدا کنیم:

```
x = tf.constant([10, 0, 13, 9])
```

شاخص حداقل مقدار با استفاده از `tf.argmin()` محاسبه می‌شود:

```
tf.argmin(x).numpy()

1
```

شاخص حداکثر مقدار با استفاده از `tf.argmax()` محاسبه می‌شود:

```
tf.argmax(x).numpy()

2
```

کد زیر را اجرا کنید تا اختلاف مربع بین x و y را پیدا کنید:

```
x = tf.Variable([1,3,5,7,11])
y = tf.Variable([1])

tf.math.squared_difference(x,y).numpy()

[  0,   4,  16,  36, 100]
```

بیایید مبدل[1] را امتحان کنیم؛ یعنی تبدیل از یک نوع داده به نوع دیگر. حالا دستور می‌دهیم نوع x را چاپ کند:

```
print(x.dtype)

tf.int32
```

همانطور که در کد زیر نشان داده شده، می‌توانیم نوع x را که `tf.int32` است به `tf.float32` تبدیل کنیم:

```
x = tf.cast(x, dtype=tf.float32)
```

[1] Typecasting

حالا نوع x را بررسی کنید. tf.float32 به شرح زیر خواهد بود:

```
print(x.dtype)

tf.float32
```

دو ماتریس را به هم ملحق[۱] کنید:

```
x = [[3,6,9], [7,7,7]]
y = [[4,5,6], [5,5,5]]
```

ماتریس‌ها را به صورت ردیفی به هم الحاق کنید:

```
tf.concat([x, y], 0).numpy()

array([[3, 6, 9],
       [7, 7, 7],
       [4, 5, 6],
       [5, 5, 5]], dtype=int32)
```

از کد زیر برای الحاق ماتریس‌ها به صورت ستونی استفاده کنید:

```
tf.concat([x, y], 1).numpy()

array([[3, 6, 9, 4, 5, 6],
       [7, 7, 7, 5, 5, 5]], dtype=int32)
```

ماتریس x را با استفاده از تابع stack، پشته[۲] کنید:

---

۱ Concatenate

۲ Stack

پشته، یک ساختمان داده‌ای است که عمل حذف و اضافه از بالای ان انجام می‌شود. یعنی در یک پشته، موقعی که می‌خواهیم یک عنصر را حذف کنیم، اخرین عنصری که وارد پشته شده را حذف می‌کنیم. اصطلاحا می‌گویند پشته از اصل LIFO (Last In First Out) تبعیت می‌کند و بدین معناست که آخرین موردی که وارد پشته می شود، اولین موردی است که خارج می شود. این مشخصه پشته‌ها را از سایر ساختارهای داده خطی، مانند صف، متمایز می کند. یک مثال روزمره، ورود و خروج در آسانسور است: آخرین کسی که وارد آسانسور می شود اولین کسی است که از آن خارج می شود!

```
tf.stack(x, axis=1).numpy()

array([[3, 7],
       [6, 7],
       [9, 7]], dtype=int32)
```

حالا بیایید ببینیم چگونه عملیات reduce_mean را انجام دهیم:

```
x = tf.Variable([[1.0, 5.0], [2.0, 3.0]])

x.numpy()

array([[1., 5.],
       [2., 3.]]
```

مقدار میانگین x را محاسبه کنید؛ یعنی ۴/(۳.۰ + ۲.۰ + ۵.۰ + ۱.۰):

```
tf.reduce_mean(input_tensor=x).numpy()

2.75
```

میانگین را در سراسر ردیف محاسبه کنید؛ یعنی ۲/(۵.۰ + ۱.۰) و ۲/(۳.۰ + ۲.۰):

```
tf.reduce_mean(input_tensor=x, axis=0).numpy()

array([1.5, 4. ], dtype=float32)
```

میانگین را در سراسر ستون محاسبه کنید؛ یعنی ۲.۰/(۵.۰ + ۱.۰) و ۲.۰/(۳.۰ + ۲.۰):

```
tf.reduce_mean(input_tensor=x, axis=1, keepdims=True).numpy()

array([[3. ],
       [2.5]], dtype=float32)
```

مقادیر تصادفی را از توزیع‌های احتمال بکشید:

```
tf.random.normal(shape=(3,2), mean=10.0, stddev=2.0).numpy()

tf.random.uniform(shape = (3,2), minval=0, maxval=None, dtype=tf.
float32,).numpy()
```

احتمالات بیشینه–نرم softmax را محاسبه کنید:

```
x = tf.constant([7., 2., 5.])

tf.nn.softmax(x).numpy()

array([0.8756006 , 0.00589975, 0.11849965], dtype=float32)
```

اکنون، نحوه محاسبه گرادیانها را بررسی خواهیم کرد.

تابع مربع را تعریف کنید:

```
def square(x):
  return tf.multiply(x, x)
```

گرادیان‌ها را می‌توان با تابع `square` و با استفاده از کد `tf.GradientTape` محاسبه کرد.

```
with tf.GradientTape(persistent=True) as tape:
     print(square(6.).numpy())

36.0
```

تنسور_فلو ۲.۰ و Keras

تنسور–فلو ۲.۰ دارای ویژگی‌های بسیار جالبی است. مثلا حالت اجرای مشتاق را به طور پیش‌فرض تنظیم می‌کند. این کتابخانه، یک گردش کار ساده را ارائه می‌دهد و از Keras به عنوان **API** اصلی برای ساخت مدل‌های یادگیری عمیق استفاده می‌کند. همچنین با نسخه‌های تنسور–فلو ۱.x سازگار است.

برای نصب تنسور–فلو ۲.۰، ترمینال خود را باز کرده و دستور زیر را تایپ کنید:

```
pip install tensorflow==2.0.0-alpha0
```

از آنجایی که تنسور–فلو ۲.۰ از Keras به عنوان یک **API** سطح–بالا استفاده می‌کند، ما نحوه Keras را در بخش بعدی بررسی خواهیم کرد.

بونجور کراس

Keras یکی دیگر از کتابخانه‌های یادگیری عمیق است که پرکاربرد دارد. این کتابخانه توسط François Chollet در گوگل توسعه داده شده است. کراس، به دلیل نمونه‌سازی سریع[۱] خود خوشنام است، چون ساخت مدل را واقعا ساده می‌کند. این یک کتابخانهٔ سطح بالا است، به این معنی که هیچ عملیات سطح پایینی مانند همتابی (کانولوشن) را به تنهایی انجام نمی‌دهد. برای انجام این کار از یک موتور پشتیبان مانند تنسور–فلو استفاده می‌کند. **API** Keras در `tf.keras` موجود است و تنسور–فلو ۲.۰ از آن به عنوان **API** اصلی استفاده می‌کند.

ساخت مدل در Keras شامل چهار مرحله مهم است:

۱. تعریف مدل
۲. کامپایل مدل
۳. برازش مدل
۴. ارزیابی مدل

تعریف مدل

اولین قدم تعریف مدل است. Keras از دو **API** مختلف را برای تعریف مدل استفاده کرد:

۱. **API** متوالی

[۱] fast prototyping

۲. API

تعریف یك مدل متوالی

در یک مدل متوالی، هر لایه را روی لایه‌های قبلی قرار داده و پشته می‌کنیم:

```
from keras.models import Sequential
from keras.layers import Dense
```

ابتدا، بیایید مدل خود را به عنوان یک مدل `Sequential()` تعریف کنیم، به شرح زیر است:

```
model = Sequential()
```

اکنون، لایه اول را همانطور که در کد زیر نشان داده شده است، تعریف کنید:

```
model.add(Dense(13, input_dim=7, activation='relu'))
```

در کد قبلی، `Dense` یک لایه کاملا متصل را نشان می‌دهد، `input_dim` نشانگر بُعد ورودی است و `activation`، تابع فعالسازی است. ما می‌توانیم هر تعداد لایه را که می‌خواهیم، یکی بالای دیگر، روی هم دهیم.

لایه بعدی را با تابع فعالسازی `relu` به شرح زیر تعریف کنید:

```
model.add(Dense(7, activation='relu'))
```

لایه خروجی را با تابع فعالسازی `sigmoid` تعریف کنید:

```
model.add(Dense(1, activation='sigmoid'))
```

بلوک کدِ[۱] نهایی مدلِ متوالی، به صورت زیر نشان داده شده است. همانطور که می‌بینید، کد Keras بسیار ساده‌تر از کد تنسور–فلو است:

```
model = Sequential()
model.add(Dense(13, input_dim=7, activation='relu'))
model.add(Dense(7, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

تعریف یک مدل تابعی

یک مدل تابعی[۲]، انعطاف‌پذیری بیشتری نسبت به یک مدل متوالی ارائه می‌دهد. به عنوان مثال، در یک مدل تابعی، ما به راحتی می‌توانیم هر لایه را به لایه دیگری متصل کنیم، در حالی که در یک مدل متوالی، هر لایه در یک پشته، یکی بالای دیگری قرار دارد. یک مدل تابعی، هنگام ایجاد مدل‌های پیچیده، مانند گرافهای غیرچرخه‌ای جهتدار، مدل‌هایی با مقادیر ورودی متعدد، مقادیر خروجی چندگانه و لایه‌های مشترک مفید است. اکنون خواهیم دید که چگونه یک مدل تابعی را در Keras تعریف کنیم.

اولین قدم تعریف ابعاد ورودی است:

```
input = Input(shape=(2,))
```

```
relu
```

```
layer1 = Dense(10, activation='relu')
```

ما لایه ۱ را تعریف کردیم، اما ورودی لایه ۱ از کجا می‌آید؟ ما باید همانطور که نشان داده شده است، ورودی لایه ۱ را در یک نماد پرانتز در انتها مشخص کنیم:

---

۱ Code Block
۲ Functional Model

```
layer1 = Dense(10, activation='relu')(input)
```

ما لایه بعدی یعنی لایه ۲، را با ۱۳ نورون و تابع فعالساز `relu` تعریف می‌کنیم. ورودی لایه ۲ از لایه ۱ می‌آید، بنابراین همانطور که در کد زیر نشان داده شده است، در پرانتز انتهایی اضافه می‌شود:

```
layer2 = Dense(10, activation='relu')(layer1)
```

اکنون می‌توانیم لایه خروجی را با تابع فعالساز سیگموئید تعریف کنیم. ورودی به لایه خروجی از لایه ۲ می‌آید، بنابراین در انتها در پرانتز اضافه می‌شود:

```
output = Dense(1, activation='sigmoid')(layer2)
```

پس از تعریف تمام لایه‌ها، مدل را با استفاده از یک کلاس `Model` تعریف می‌کنیم، جایی که باید `inputs` و `outputs` را به شرح زیر مشخص کنیم:

```
model = Model(inputs=input, outputs=output)
```

کد کامل مدل تابعی در اینجا نشان داده شده است:

```
input = Input(shape=(2,))
layer1 = Dense(10, activation='relu')(input)
layer2 = Dense(10, activation='relu')(layer1)
output = Dense(1, activation='sigmoid')(layer2)
model = Model(inputs=input, outputs=output)
```

همگردانی (کامپایل کردن) مدل

اکنون که مدل را تعریف کردیم، مرحله بعدی کامپایل (همگردانی) آن است. در این مرحله، ما نحوه یادگیری مدل را تنظیم می‌کنیم. هنگام کامپایل کردن مدل، سه پارامتر تعریف می‌کنیم:

۱. پارامتر `optimizer`: این پارامتر، الگوریتم بهینه‌سازی ما را تعریف می‌کند. به عنوان مثال: نزول گرادیان.

۲. پارامتر `loss`: این همان تابع هدفی است که ما در حال تلاش برای کمینه کردن آن هستیم. به عنوان مثال: تابع میانگین مربع خطا یا تابع زیان آنتروپی متقاطع.

۳. پارامتر `metrics`: این پارامتر شاخص یا معیاری است که از طریق آن می‌خواهیم عملکرد مدل را ارزیابی کنیم. به عنوان مثال: `accuracy`. همچنین می‌توانیم بیش از یک معیار را مشخص کنیم.

کد زیر را برای کامپایل (همگردانی) مدل اجرا کنید:

```
model.compile(loss='binary_crossentropy', optimizer='sgd',
metrics=['accuracy'])
```

آموزش مدل

ما مدل را تعریف و همچنین کامپایل کردیم. اکنون، ما مدل را آموزش خواهیم داد. آموزش مدل را می‌توان با استفاده از تابع برازش `fit` انجام داد. ما ویژگی‌های خود را مشخص می‌کنیم: `x`؛ برچسب‌ها، `y`؛ تعداد `epochs` که می‌خواهیم مدل را برای آنها آموزش دهیم؛ و `batch_size`، را به شرح زیر تعیین می‌کنیم:

```
model.fit(x=data, y=labels, epochs=100, batch_size=10)
```

ارزیابی مدل

پس از آموزش مدل، آنرا بر روی مجموعه آزمون ارزیابی خواهیم کرد:

```
model.evaluate(x=data_test,y=labels_test)
```

ما همچنین می‌توانیم مدل را با همان مجموعه داده آموزش، ارزیابی کنیم و این به ما کمک می‌کند تا دقت آموزش را درک نماییم:

```
model.evaluate(x=data,y=labels)
```

و تمام! حال بیایید ببینیم چگونه تنسور–فلو ۲.۰ وظیفه طبقه‌بندی اعداد (ارقام) دستنویس در **MNIST** را انجام می‌دهد.

طبقه بندی اعداد MNIST با استفاده از تنسور_فلو ۲.۰

اکنون، خواهیم دید که چگونه می‌توانیم طبقه‌بندی اعداد دستنویس **MNIST** را با استفاده از تنسور–فلو ۲.۰ انجام دهیم. در مقایسه با تنسور–فلو ۱.x فقط به چند خط کد نیاز دارد. همانطور که آموختیم، تنسور–فلو ۲.۰ از Keras به عنوان **API** سطح بالا خود استفاده می‌کند. فقط باید `tf.keras` را به کد Keras اضافه کنیم.

بیایید با بارگذاری مجموعه داده شروع کنیم:

```
mnist = tf.keras.datasets.mnist
```

یک مجموعه آموزشی و یک مجموعه آزمایشی با کد زیر ایجاد کنید:

```
(x_train,y_train), (x_test, y_test) = mnist.load_data()
```

مجموعه‌های آموزشی و آزمایشی را با تقسیم مقادیر **X** بر حداکثر مقدار **X** نرمال کنید؛ یعنی ۲۵۵.۰:

```
x_train, x_test = tf.cast(x_train/255.0, tf.float32), tf.cast(x_
test/255.0, tf.float32)
y_train, y_test = tf.cast(y_train,tf.int64),tf.cast(y_test,tf.int64)
```

مدل متوالی را به صورت زیر تعریف کنید:

```
model = tf.keras.models.Sequential()
```

حالا، بیایید لایه‌هایی را به مدل اضافه کنیم. ما از یک شبکه سه لایه، با تابع ReLU در لایه پنهان و softmax در لایه پایانی استفاده می‌کنیم:

```
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(256, activation="relu"))
model.add(tf.keras.layers.Dense(128, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="softmax"))
```

مدل را با اجرای خط کد زیر کامپایل کنید:

```
model.compile(optimizer='sgd', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

مدل را آموزش دهید:

```
model.fit(x_train, y_train, batch_size=32, epochs=10)
```

مدل را ارزیابی کنید:

```
model.evaluate(x_test, y_test)
```

خودشه. همین و بس! نوشتن کد با **API** Keras به همین سادگی است.

خلاصه

ما این فصل را با درک تنسور–فلو و نحوه استفاده از گرافهای محاسباتی شروع کردیم. ما یاد گرفتیم که محاسبات در تنسور–فلو با یک گراف محاسباتی نشان داده می‌شود که از چندین گره و یال تشکیل شده است، جایی که گره‌ها عملیات ریاضی هستند، مانند جمع و ضرب، و یالها همانا، تنسور هستند.

در مرحله بعد، متوجه شدیم که متغیرها ظروفی هستند که برای ذخیره مقادیر استفاده می‌شوند و به عنوان ورودی چندین عملیات دیگر در یک گراف محاسباتی مورد استفاده قرار می‌گیرند. ما همچنین یاد گرفتیم که جای‌بانها مانند متغیرها هستند، جایی که ما فقط نوع و ابعاد را تعریف می‌کنیم اما مقادیر را اختصاص نمی‌دهیم و مقادیر برای جای‌بانها در زمان اجرا، تغذیه می‌شوند. با حرکت رو به جلو، ما در مورد تنسور–بُرد که تجسم تنسور–فلو است آشنا شدیم. ما آموختیم که تنسور–بُرد می‌توان برای تجسم یک گراف محاسباتی استفاده کرد. ما همچنین اجرای مشتاقانه (سریع و چابک) را بررسی کردیم که بیشتر پایتون–مانند[۱] است و امکان نمونه‌سازی سریع را فراهم می‌کند.

[۱] Pythonic

ما متوجه شدیم که برخلاف حالت گراف، که در آن برای انجام هر عملیاتی باید هر بار یک گراف بسازیم، اجرای مشتاقانه از پارادایم برنامه نویسی امری (دستوری) پیروی می‌کند، جایی که هر عملیاتی را می‌توان بلافاصله انجام داد، بدون نیاز به ایجاد گراف، درست مانند پایتون.

در فصل بعدی، یکی از الگوریتم‌های محبوب یادگیری تقویتی عمیق (**DRL**) به نام **شبکه Q عمیق**[۱] یا (**DQN**) را در فصل بعدی یاد می‌گیریم، تا سفر **یادگیری تقویتی عمیق** (**DRL**) خود را با درک آن، آغاز می‌کنیم.

سوالات

بیایید دانش خود را در مورد تنسور–فلو با پاسخ دادن به سوالات زیر آزمایش کنیم:

۱. جلسه (کار–دوره) تنسور–فلو چیست؟

۲. یک مکان نگهدارنده (جای‌بان) تعریف کنید.

۳. تنسور–برد چیست؟

۴. چرا حالت اجرای مشتاق (تند و تیر) مفید است؟

۵. تمام مراحل ساخت مدل با استفاده از Keras چیست؟

۶. مدل تابعی Keras چه تفاوتی با مدل متوالی آن دارد؟

[۱] Deep Q Ntework (DQN)

بیشتر بخوانید

می‌توانید با بررسی مستندات رسمی تنسور–فلو، کمی بیشتر در مورد آن بدانید:

https://www.tensorflow.org/tutorials.

پیوست ۱– فصل ۸

تنسور چیست؟

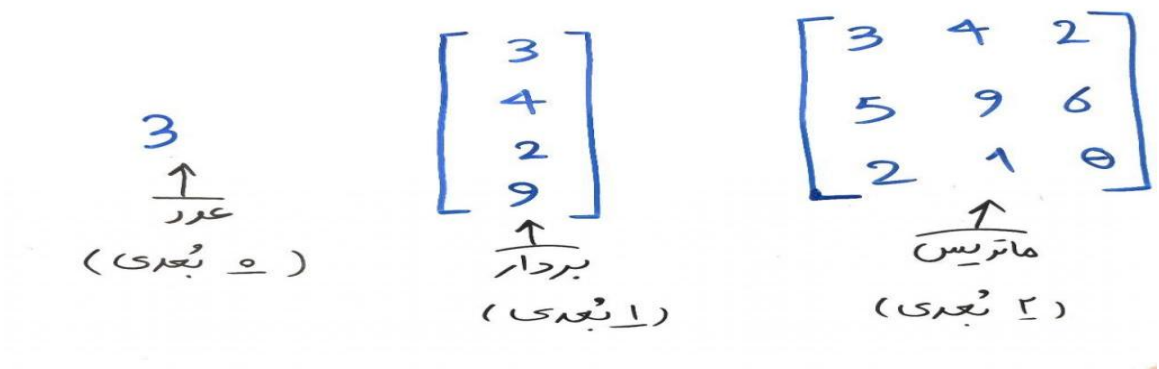
مسعود کاویانی

عدد (Scalar)، بردار (Vectors)، ماتریس (Matrix) و تنسور (Tensor) چیستند؟

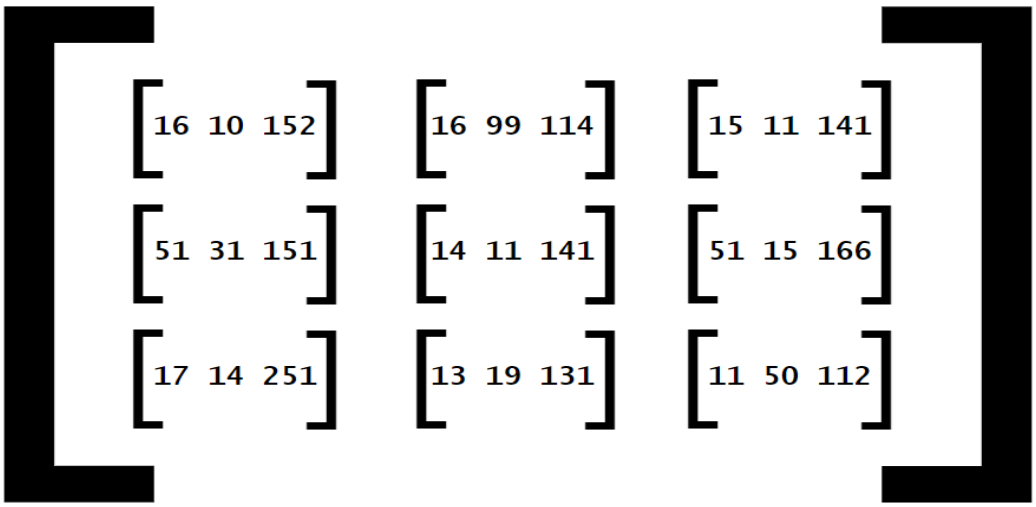
این درس از مجموعه دوره ریاضیات پایه و جبر خطی (Linear Algebra) برای یادگیری ماشین است.[۱]

پایه عملیاتِ یادگیری‌ماشین و یادگیری‌عمیق، **اعداد** هستند. این اعداد با قرار گرفتن در کنارِ هم ، **بردارها** را می‌سازند و بردارها در کنار یکدیگر قرار می‌گیرند و **ماتریس‌ها** را تشکیل می‌دهند. مطمئن هستم که تقریباً همه کسانی که این نوشته را می‌خوانند به نوعی با این سه عنصر (**عدد، بردار و ماتریس**) آشنایی دارند. عناصری که در داده‌کاوی و یادگیری‌ماشین و مخصوصاً یادگیری عمیق بسیار کاربردی هستند. در این درس می‌خواهیم این سه عنصر (به همراه یک عنصر دیگر به نامِ **تنسور**) را با هم مرور کنیم. تصویر زیر را در مورد عدد، بردار و ماتریس ببینید:

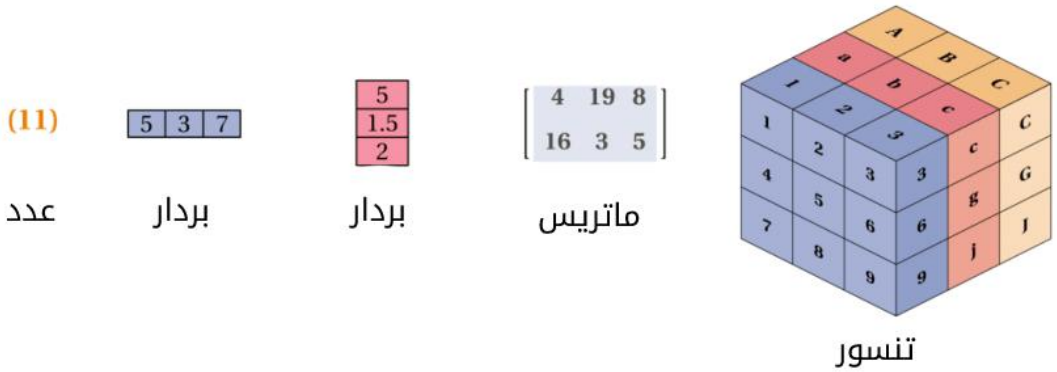
[۱] https://chistio.ir/%D۸٪A۷٪D۸٪B۹٪D۸٪AF%D۸٪A۷٪D۸٪AF-%D۸٪A۸٪D۸٪B۱٪D۸٪AF%D۸٪A۷٪D۸٪B۱-vectors-%D۹٪۸۵٪D۸٪A۷٪D۸٪AA%D۸٪B۱٪DB%۸C%D۸٪B۳-matrix-%D۸٪AA%D۹٪۸٦٪D۸٪B۳٪D۹٪۸۸٪D۸٪B۱-tensor-%DA%۸٦٪DB%۸C%D۸٪B۳٪D۸٪AA/



قطعاً با این عناصر آشنایی دارید اما شاید تسنور (Tensor) واژه‌ی جدیدی باشد. تنسور در واقع یک ماتریس است که هر کدام از خانه‌های آن به جای این‌که یک عدد داشته باشند، می‌تواند چندین عدد را در خود جای دهد. شکل زیر را نگاه کنید:

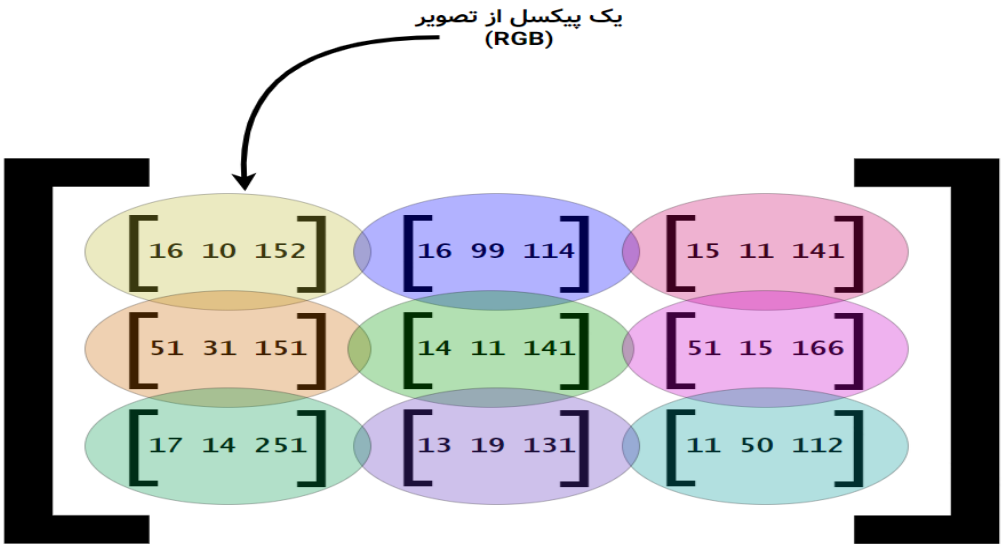


تصویر بالا یک نمونه تنسور است که سه بُعد دارد. به بیانی دیگر تصویرِ بالا یک ماتریس است که هر کدام از خانه‌های آن، خود یک بردار هستند. در واقع تنسور یک بُعد بیشتر از ماتریس دارد. شکل زیر، خلاصه چیزی است که تا حالا در مورد عدد، بردار، ماتریس و تنسور گفته‌ایم:



پس به طور کلی: عدد ۰ بُعد دارد، بردار ۱ بُعدی است، ماتریس ۲ بُعدی و در نهایت تنسور ۳ بُعد دارد.

اگر و درس طبقه‌بندی را خوانده باشید، مشاهده می‌کنید که در واقع یک ماتریس از مشتریان ساخته‌ایم که هر مشتری یک بردار است (و این یکی از کاربردهای اساسی بردار و ماتریس در عملیات یادگیری ماشین و داده‌کاوی است). اما در مورد کاربردِ تنسور می‌توان مثالِ یک عکس مانند (**JPEG**) را آورد. فرض کنید می‌خواهیم یک تصویر را به کامپیوتر به وسیله تنسور وارد کنیم. در واقع بایستی کاری کنیم که این تصویر برای کامپیوتر و به تبعِ آن الگوریتم‌های داده‌کاوی قابلِ فهم باشد. عرض و ارتفاع تصویر که مانند یک ماتریس است و هر خانه ماتریس هم یک پیکسل (pixel) از آن تصویر است. همان‌طور که می‌دانید در سیستم **RGB** هر پیکسل از مجموعه رنگ قرمز، سبز و آبی (**RGB**) تشکیل شده است. پس هر کدام از خانه‌های این ماتریس را می‌توان ترکیبی از این سه رنگ دانست. حال دوباره شکلِ زیر را نگاه کنید:



ما یک تصویر را به یک تنسور تبدیل کرده‌ایم. هر کدام از خانه‌های این ماتریس، نمایان‌گرِ یک پیکسل است، که خود از سه رنگ (یک بردارِ سه عضوی) تشکیل شده است.

پایه عملیات یادگیری ماشین و یادگیری عمیق را می‌توان این عناصر دانست. در واقع جبرخطی با استفاده از این عناصر، فرمول‌ها و فرآیندهای خود را می‌سازد.

پیوست ۲ – فصل ۸

مفاهیم اصلی تنسور

سید سراج حمیدی (+)

سید سراج حمیدی دانش‌آموخته مهندسی برق است و به ریاضیات و زبان و ادبیات فارسی علاقه دارد. او آموزش‌های مهندسی برق، ریاضیات و ادبیات مجله فرادرس را می‌نویسد.

نتسورها (یا تانسورها) در بسیاری از حوزه‌های علم فیزیک از جمله الاستیسیته، مکانیک سیالات و نسبیت عام، چارچوب ریاضی فشرده و مختصری را برای فرمول‌بندی و حل مسائل گوناگون فراهم می‌کنند و به همین دلیل، از اهمیت خاصی برخوردار هستند. از زمینه‌های دیگری که مفهوم تانسور در آن‌ها کاربرد دارد، می‌توان به یادگیری عمیق در علم داده اشاره کرد. در این آموزش، مفاهیم مربوط به تانسور را بیان می‌کنیم.

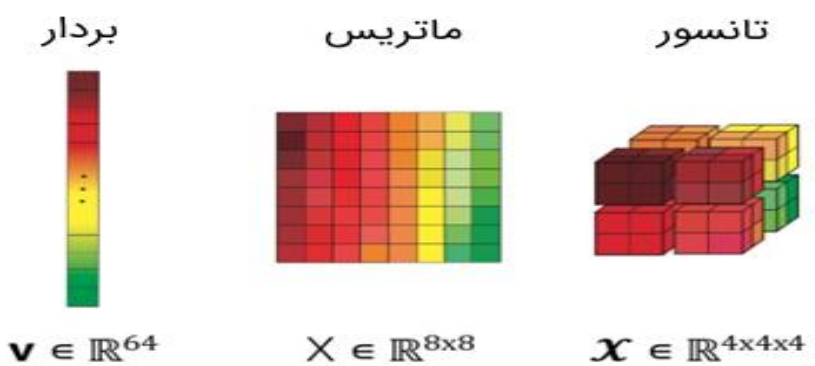
فهرست مطالب پیوست ۲

- تعریف تنسور
- جمع و تفریق تنسورها
- ادغام و ضرب مستقیم
- تقارن و پادتقارن
- مثال

تعریف تنسور

تنسور (Tensor) ، نقطه‌ای از فضا است که توسط یک یا چند شاخص که بیانگر مرتبه آن است، توصیف می‌شود. به‌طور کلی، تنسوری با مرتبه n در فضای m بعدی، n شاخص و mn مؤلفه دارد و از قواعد تبدیل معینی تبعیت می‌کند. مثلاً، تانسوری با مرتبه یک در فضای سه‌بعدی، یک شاخص و ۳ مؤلفه دارد. در واقع، تانسورها تعمیمی از اسکالرها (که بدون شاخص هستند)، بردارها (که یک شاخص دارند) و ماتریس‌ها (که دو شاخص دارند) با تعداد دلخواهی از شاخص‌ها هستند.

برای مثال، همان‌گونه که در شکل زیر می‌بینید، بردار، ماتریس و تنسور ۶۴ مؤلفه دارند که این مؤلفه‌ها در تنسور به‌صورت سه‌بعدی هستند. از این رو می‌توان گفت تنسورها آرایه‌های چندبعدی دارند.



نمادگذاری یک تانسور شبیه ماتریس است (یعنی $A = \{a_{ij}\}$)، البته تانسور می‌تواند تعداد دلخواهی از شاخص‌ها را به صورت $a_{ijk\dots}$، $a^{ijk\dots}$، $a_i^{jk\dots}$ و ... داشته باشد. به‌طور کلی، تانسوری مثل t با مرتبه $r+s$ می‌تواند یک تانسور از نوع آمیخته (r,s) باشد (یعنی $t^{\alpha_1\dots\alpha_r}{}_{\beta_1\dots\beta_s}$ که r (تعداد شاخص‌های بالا) را شاخص‌های «پادوردا» (Contravariant) و s (تعداد شاخص‌های پایین) را شاخص‌های «هموردا» (Covariant) می‌نامند. اصطلاحاً گفته می‌شود تانسور نسبت به شاخص‌های بالا پادوردا و نسبت به شاخص‌های پایین هموردا است. توجه داشته باشید که محل قرار گرفتن شاخص‌های پادوردا و هموردا نیز حائز اهمیت است. برای مثال، $\alpha_{\mu\nu}{}^{\lambda}$ و $\alpha_{\mu}{}^{\nu\lambda}$ با هم متفاوت هستند.

تانسورهای مرتبه صفر، یک و دو به ترتیب، اسکالر، بردار و ماتریس نامیده می‌شوند. از این رو، در نمادگذاری تانسوری، برداری مثل $\mathbf{v}$ را به‌صورت v_i $(i = 1, \dots, m)$ و ماتریس را که تانسوری از نوع $(1,1)$ است، به‌شکل $a_i{}^j$ نمایش می‌دهند.

نمادگذاری تانسوری این امکان را فراهم می‌کند که اتحادهای برداری و کلی‌تر را به‌طور فشرده و مختصر بنویسیم. برای مثال، ضرب داخلی $\mathbf{u} \cdot \mathbf{v}$ را می‌توان به‌صورت تانسوری زیر نوشت:

$$\mathbf{u} \cdot \mathbf{v} = u_i v^i$$

در این‌جا طبق قرارداد جمع انیشتین که بیان می‌کند هرگاه شاخصی در یک طرف معادله دو بار (یک بار به صورت شاخص بالا و یک بار به صورت شاخص پایین) ظاهر شود، روی آن شاخص جمع زده

می‌شود، روی شاخص i جمع می‌زنیم. مثلاً، در فضای سه‌بعدی، ضرب داخلی بالا را می‌توان این‌گونه نوشت:

$$\mathbf{u} \cdot \mathbf{v} = u_i v^i = \sum_{i=1}^{3} u_i v^i = u_1 v^1 + u_2 v^2 + u_3 v^3.$$

به‌طور مشابه، می‌توانیم ضرب خارجی را به‌صورت خلاصه زیر بنویسیم:

$$(u \times v)_i = \epsilon_{ijk} u^j v^k$$

که ϵ_{ijk} **تانسور جایگشت** نام دارد و به نماد «لوی-چیویتا» (Levi-Civita) معروف است. زمانی که تعداد جایگشت‌های سه شاخص i، j و k زوج و فرد باشد، مقدار این تانسور به ترتیب برابر با 1 و -1 خواهد بود و در صورتی که حداقل دو تا از شاخص‌های i، j و k برابر باشند، مقدار آن صفر خواهد شد. برای مثال، اگر در فضای سه‌بعدی مؤلفه اول ضرب خارجی $(u \times v)_i$ را به‌دست آوریم، خواهیم داشت:

$$(u \times v)_1 = \epsilon_{1jk} u^j v^k$$

طبق تعریف، اگر دو شاخص برابر باشند، مقدار تانسور جایگشت صفر می‌شود، بنابراین در فضای سه‌بعدی فقط دو حالت داریم:

$$(u \times v)_1 = \epsilon_{123} u^2 v^3 + \epsilon_{132} u^3 v^2$$

در جمله اول، جایگشتی نداریم اما در جمله‌ی دوم، ترتیب قرار گرفتن شاخص‌ها متفاوت بوده و بین 2 و 3، یک جایگشت صورت گرفته است. از این رو، مقدار تانسور -1 خواهد بود:

$$(u \times v)_1 = u^2 v^3 - u^3 v^2$$

با دستکاری شاخص‌های تانسور (بالا و پایین آوردن شاخص‌ها) می‌توان عباراتی را که به‌شکل تانسور نوشته شده‌اند ساده کرد. این کار را می‌توان توسط تانسوری به‌نام **تانسور متریک** (g_{ij}، g^{ij}، $g_i{}^j$ و...) انجام داد. به‌ازای جابه‌جایی هر شاخص از یک تانسور متریک استفاده می‌کنیم. برای مثال:

$$g^{ij}A_j = A^i, \quad g_{ij}A^j = A_i$$

$$g^{ik}g^{jl}A_{ij} = A^{kl}, \quad g_{ik}g_{jl}A^{ij} = A_{kl}$$

عبارت g_{ij} یک تانسور مرتبه دو است و به فضا و ابعادی بستگی دارد که محاسبات را در آن انجام می‌دهیم. این تانسور معمولاً به‌صورت یک ماتریس قطری است و در این حالت، g^{ij} که وارون g_{ij} است نیز قطری خواهد بود.

جمع و تفریق تانسورها

اگر دو تانسور A و B هم‌مرتبه بوده و شاخص‌های هموردا و پادوردای یکسانی داشته باشند، می‌توان آن‌ها را با هم جمع یا از هم کم کرد که حاصل آن نیز تانسوری با همان مرتبه و با همان شاخص‌ها خواهد بود:

$$A^{ij} + B^{ij} = C^{ij},$$

$$A_{ij} + B_{ij} = C_{ij},$$

$$A^i{}_j + B^i{}_j = C^i{}_j.$$

لازم به ذکر است که هر دو تانسور A و B باید در یک فضا و با تعداد ابعاد یکسان تعریف شده باشند.

ادغام و ضرب مستقیم

تعمیم ضرب داخلی تانسورها، «ادغام» (Contraction) تانسور گفته می‌شود و شامل برابر قرار دادن دو شاخص متفاوت (یکی پادوردا و دیگری هموردا) و جمع بستن روی آن شاخص با استفاده از

قرارداد جمع انیشتین است. به بیان دیگر، این کار، تانسور نوع (r, s) را به یک تانسور نوع $(r-1, s-1)$ تبدیل می‌کند. مثلاً با ادغام دو شاخص μ و λ در تانسور $t^{\mu\nu}_{\lambda}$ خواهیم داشت:

$$t^{\mu\nu}{}_{\mu} = t^{\nu}.$$

همان‌گونه که می‌بینیم، با ادغام، دو واحد از مرتبه تانسور کم می‌شود.

اگر دو تانسور در هم ضرب شوند، حاصل، تانسوری خواهد شد که مرتبه آن مساوی با مجموع مرتبه‌های دو تانسور اولیه است:

$$A_{ij}B^{kl} = C^{kl}_{ij}$$

در صورتی که یکی از شاخص‌های B^{kl} با یکی از شاخص‌های A_{ij} برابر باشد، می‌توان از ادغام شاخص‌ها استفاده کرد:

$$A_{ik}B^{kl} = C^{l}_{i}.$$

چنانچه تمام شاخص‌های B^{kl} و A_{Ij} با هم برابر باشند، حاصل‌ضرب آن‌ها یک تانسور مرتبه صفر یا به‌عبارتی، یک اسکالر خواهد بود.

تقارن و پادتقارن

ترتیب قرار گرفتن تانسورها اهمیت دارد. به‌عنوان نمونه، تانسورهای $t^{\mu\nu}$ و $t^{\nu\mu}$ با هم متفاوت هستند، اما در بعضی موارد این دو تانسور با هم برابرند، یعنی:

$$t^{\mu\nu} = t^{\nu\mu}$$

در این حالت می‌گوییم تانسور متقارن است. ولی اگر داشته باشیم:

$$t^{\mu\nu} = -t^{\nu\mu}$$

تانسور پادمتقارن خواهد بود.

مثال

در این‌جا برای درک بهتر مفهوم تانسور و نحوه کاربرد آن، تانسوری به نام تانسور الکترومغناطیسی را معرفی می‌کنیم که در نسبیت عام کاربرد فراوانی دارد. این تانسور یک تانسور مرتبه دو و پادمتقارن است:

$$F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu \qquad F_{\mu\nu} = -F_{\nu\mu}$$

که در آن، $\partial_\mu = \frac{\partial}{\partial x^\mu} = (\frac{\partial}{C\partial t}, \frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$ مشتق جزئی در فضا-زمان مینکوفسکی (ct, x, y, z) است و A چهاربردار پتانسیل نام دارد که شکل پادوردا و هموردای آن به‌ترتیب، $A^\mu = (\frac{\phi}{c}, \overrightarrow{A})$ و $A_\mu = (\frac{\phi}{c}, -\overrightarrow{A})$ هستند. همچنین، $\overrightarrow{A}$ پتانسیل برداری، ϕ پتانسیل اسکالر و C سرعت نور است.

واضح است که این تانسور در فضا-زمان چهاربعدی، $4^2 = 16$ مؤلفه دارد و به‌شکل یک ماتریس 4×4 است که عناصر قطر اصلی آن یعنی $(F_{tt}, F_{xx}, F_{yy}, F_{zz})$ صفر هستند. برای نمونه، چند مؤلفه این تانسور را در فضای «مینکوفسکی» (Minkowski) به‌دست می‌آوریم:

$$F_{tx} = \partial_t A_x - \partial_x A_t$$

$$= \frac{\partial(-A_x)}{C\partial t} - \frac{\partial}{\partial t} - \frac{\partial}{\partial x}(\frac{\phi}{C}) = \frac{1}{C}(-\frac{\partial A_x}{\partial t} - \frac{\partial \phi}{\partial x})$$

از آن‌جایی که $-\frac{\partial \vec{A}}{\partial t} - \vec{\nabla}\phi = \vec{E}$ است، داریم:

$$F_{tx} = -F_{xt} = \frac{E_x}{C}$$

$$F_{xy} = \partial_x A_y - \partial_y A_x = \frac{\partial}{\partial x}(-A_y) - \frac{\partial}{\partial y}(-A_x) =$$

$$-\left(\frac{\partial A_y}{\partial x} - \frac{\partial A_x}{\partial y}\right) = -(\vec{\nabla} \times \vec{A})_z = -B_z$$

$$F_{xy} = -F_{yx} = -B_z$$

سایر مؤلفه‌ها نیز به همین صورت محاسبه می‌شوند:

$$F_{\mu\nu} = \begin{pmatrix} 0 & E_x/C & E_y/C & E_z/C \\ -E_x/C & 0 & -B_z & B_z \\ -E_y/C & B_z & 0 & -B_x \\ -E_z/C & -B_y & B_x & 0 \end{pmatrix}$$

برای به‌دست آوردن تانسور پادوردای الکترومغناطیسی، کافی است تانسور پادوردای متریک را در $F_{\mu\nu}$ ضرب کنیم:

$$F^{\mu\nu} = g^{\mu\alpha} g^{\nu\beta} F_{\alpha\beta}$$

در فضا-زمان چهاربعدی مینکوفسکی، به‌دلیل اینکه چهار بعد داریم، تانسور متریک را به‌شکل یک ماتریس 4×4 در نظر می‌گیریم که جز عناصر قطری، سایر عناصر آن صفر هستند:

$$g_{\mu\nu} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

عبارت g^{ij} وارون g_{ij} است و چون g_{ij} یک ماتریس قطری است، g^{ij} نیز قطری خواهد بود. با این تفاوت که مؤلفه‌های قطری آن، وارون مؤلفه‌های قطری g_{ij} است. بنابراین در این فضا-زمان خواهیم داشت:

$$g_{ij} = g^{ij}$$

$$g_{tt} = g^{tt} = 1, \quad g_{xx} = g_{yy} = g_{zz} = g^{xx} = g^{yy} = g^{zz} = -1$$

اکنون با استفاده از این تانسور، مؤلفه‌های $F^{\mu\nu}$ را به‌دست می‌آوریم:

$$F^{tx} = g^{t\alpha} g^{x\beta} F_{\alpha\beta}$$

از آن‌جایی که فقط عناصر قطری تانسور متریک غیرصفر هستند، تنها یک حالت غیرصفر داریم:

$$F^{tx} = g^{tt} g^{xx} F_{tx} = -\frac{E_x}{C} \Rightarrow F^{tx} = -F^{xt} = -\frac{E_x}{C}$$

$$F^{xy} = g^{x\alpha} g^{y\beta} F_{\alpha\beta} = g^{xx} g^{yy} F_{xy} = -B_z \Rightarrow F^{xy} = -F^{yx} = -B_z$$

سایر مؤلفه‌ها را می‌توان به‌طور مشابه به‌دست آورد:

$$F_{\mu\nu} = \begin{pmatrix} 0 & -E_x/C & -E_y/C & -E_z/C \\ E_x/C & 0 & -B_z & B_y \\ E_y/C & B_z & 0 & -B_x \\ E_z/C & -B_y & B_x & 0 \end{pmatrix}$$

اگر شکل هموردا و پادوردای تانسور الکترومغناطیسی در هم ضرب شوند، یک کمیت عددی به‌دست می‌آید:

$$F_{\mu\nu} F^{\mu\nu} = 2\left(B^2 - \frac{E^2}{C^2}\right)$$

فصل نهم

شبکه Q عمیق و انواع آن

در این فصل، بیایید با یکی از محبوبترین الگوریتمهای **یادگیری تقویتی عمیق**[۱] (**DRL**) به نام **شبکه Q عمیق**[۲] یا (**DQN**) شروع کنیم. درک **DQN** بسیار مهم است زیرا بسیاری از الگوریتمهای پیشرفته یادگیری تقویتی عمیق (**DRL**) مبتنی بر **DQN** هستند. الگوریتم **DQN** برای اولین بار توسط محققان DeepMind گوگل در سال ۲۰۱۳ در مقاله‌ای[۳] پیشنهاد شد. آنها معماری **DQN** را توصیف کرده و توضیح دادند که چرا در انجام بازیهای آتاری با دقتی در سطح انسان بسیار موثر است. ما این فصل را با یادگیری اینکه شبکه Q عمیق دقیقا چیست و چگونه در یادگیری تقویتی استفاده می‌شود، شروع می‌کنیم. در مرحله بعد، به الگوریتم **DQN** می‌پردازیم. سپس یاد خواهیم گرفت که چگونه **DQN** را برای اجرای بازیهای آتاری پیاده‌سازی کنیم.

پس از آشنایی با **DQN**، چندین نوع **DQN** مانند **DQN** دوگانه[۴]، **DQN** با اولویت تجربه[۵]، **DQN** رقابتی (هماوردی)[۶] و شبکه Q بازگشتی عمیق[۷] را با جزئیات پوشش خواهیم داد.

۱. **DQN چیست؟**
۲. **الگوریتم DQN**
۳. **بازیهای آتاری با DQN**
٤. **DQN دوتایی (دوگانه)**
٥. **DQN با اولویت تجربه**
٦. **DQN رقابتی (هماوردی)**
۷. **شبکه Q بازگشتی عمیق (DRQN)**

---

۱ Deep Reinforcement Learning (DRL)
۲ Deep Q Network
۳ “Playing Atari with Deep Reinforcement Learning”
٤ Double DQN
٥ DQN With Prioritized Experience Replay
٦ Dueling DQN
۷ Deep Recurrent Q Network (DRQN)

DQN چیست؟

هدف از یادگیری تقویتی، یافتن **سیاست** بهینه است، یعنی **سیاستی** که حداکثر **بازده** (یا همان مجموع **پاداشهای** یک اپیزود) را به ما می‌دهد. برای محاسبه **سیاست** یا **خطمشی**، ابتدا **تابع** Q را محاسبه می‌کنیم. هنگامی که **تابع** Q را داریم، با انتخاب آن **اقدامی** که در هر حالت، حداکثر **مقدار** Q را دارد، **خطمشی** را استخراج می‌کنیم. به عنوان مثال، بیایید فرض کنیم که ما دو حالت **A** و **B** داریم و **فضای کنش** ما از دو اقدام تشکیل شده است؛ بگذارید اقدامات ما شامل *بالا* و *پایین* باشند. بنابراین، برای اینکه بفهمیم کدام **عمل** را در حالت **A** و **B** انجام دهیم، ابتدا **مقدار** Q همه جفتهای حالت–عمل را محاسبه می‌کنیم، همانطور که جدول ۹.۱ نشان می‌دهد:

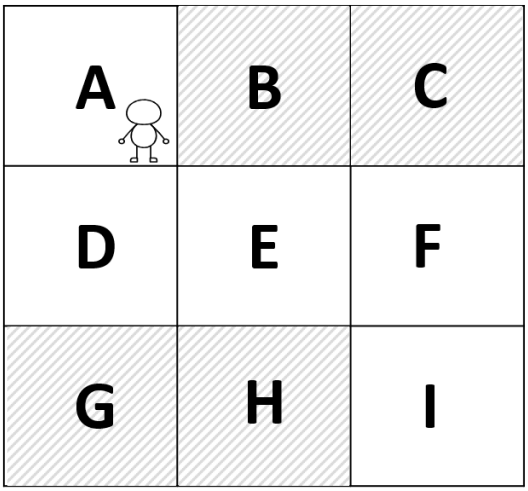
حالت	اقدام	ارزش
A	بالا	۱۷
A	پایین	۱۰
B	بالا	۱۱
B	پایین	۲۰

جدول ۹.۱: ارزش Q جفتهای حالت–عمل

هنگامی‌که ارزش یا مقدار Q را برای همه جفتهای state-acion به دست آوردیم، آن عملی را در یک حالت خاص انتخاب می‌کنیم که حداکثر مقدار Q را دارد. بنابراین، ما اقدام *بالا* را در حالت **A** و اقدام *پایین* را در حالت **B** انتخاب می‌کنیم زیرا آنها حداکثر مقدار Q را دارند. ما تابع Q را در هر تکرار بهبود می‌بخشیم و هنگامی‌که تابع Q بهینه را به دست آوردیم، می‌توانیم سیاست بهینه را از آن استخراج کنیم.

اکنون، بیایید محیط جهانی جدول–گونه خود را دوباره بررسی کنیم، همانطور که در شکل ۹.۱ نشان داده شده است.

ما یاد گرفتیم که در محیط دنیای جدول–گونه، هدف عامل ما رسیدن به حالت **I** از حالت **A**، بدون بازدید از حالتهای سایه‌دار است، و در هر حالت، عامل باید یکی از چهار عمل را انجام دهد: *بالا*، *پایین*، چپ، *راست*.



شکل ۹.۱: محیط جهانی مشبک

برای محاسبه (احصای) **سیاست**، ابتدا **ارزش** Q همه جفتهای حالت–اقدام را محاسبه می‌کنیم. در اینجا، تعداد حالتها، ۹ حالت (**A** تا **I**) است و ما ۴ اقدام در **فضای کنش** خود داریم، بنابراین جدول Q ما از ۳۶ = ۴ × ۹ ردیف تشکیل شده است که حاوی **ارزشهای** Q همه جفتهای حالت–عمل ممکن است. هنگامی‌که مقادیر Q را به دست آوردیم، با انتخاب عمل در هر یک از حالتها که حداکثر مقدار Q را دارد، **خط‌مشی** را استخراج می‌کنیم. اما آیا محاسبه **مقدار** Q به طور جامع برای همه جفتهای حالت–عمل رویکرد خوبی است؟ بیایید این را با جزئیات بیشتری بررسی کنیم.

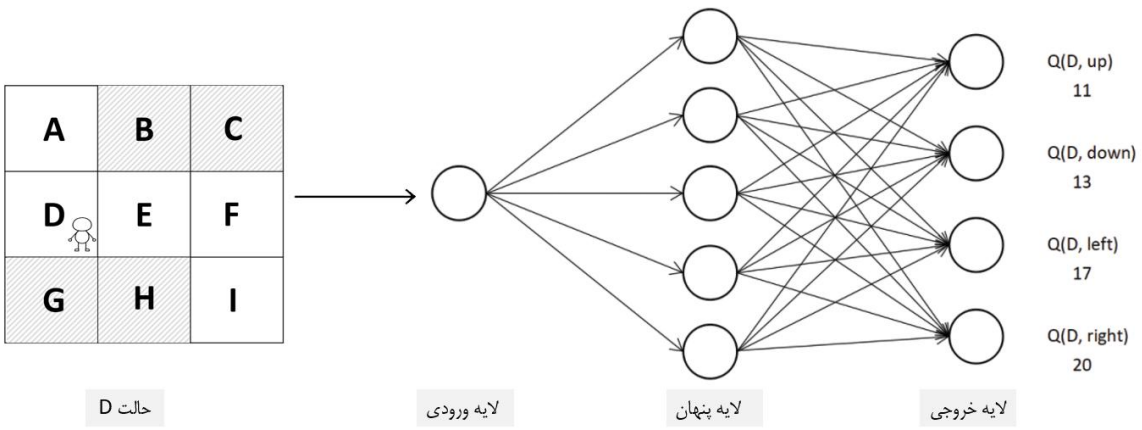
بیایید فرض کنیم محیطی داریم که در آن ۱۰۰۰ حالت و ۵۰ عمل ممکن در هر حالت داریم. در این صورت، جدول Q ما از ۱۰۰۰×۵۰ = ۵۰۰۰۰ ردیف تشکیل شده است که حاوی مقادیر Q همه جفتهای حالت–عمل ممکن است. در مواردی از این دست، جایی که محیط ما از تعداد زیادی حالت–عمل تشکیل شده است ، محاسبه مقادیر Q همه جفت های حالت–عمل ممکن به صورت جامع، بسیار زمانبر باشد.

به جای محاسبه مقادیر Q با این روش، آیا می‌توانیم آنها را با استفاده از یک **تقریب‌ساز تابع**[۱]، مانند یک **شبکه عصبی** تخمین بزنیم؟ بله! ما می‌توانیم **تابع** Q خود را با یک پارامتر همچون θ، پارامتری کرده و مقدار Q را بدین صورت محاسبه می‌کنیم که **پارامتر** θ مورد نظر همانا پارامتر یک شبکه عصبی باشد. بنابراین، ما فقط وضعیت محیط

[۱] Function Approximator

را به یک شبکه عصبی تغذیه کرده و آن شبکه، **مقدار** Qی تمام اقدامات ممکن در آن حالت را برمی‌گرداند. هنگامی‌که **مقادیر** Q را به دست آوردیم، می‌توانیم بهترین **عمل** را به عنوان **اقدامی**که حداکثر **مقدار** Q را دارد انتخاب کنیم.

به عنوان مثال، بیایید محیط جهانی مشبک خود را در نظر بگیریم. همانطور که شکل ۹.۲ نشان می‌دهد، ما فقط حالت **D** را به عنوان ورودی به شبکه تغذیه می‌کنیم و مقدار Q تمام اقدامات در حالت **D** را که *بالا*، *پایین*، چپ و *راست* هستند، به عنوان خروجی برمی‌گرداند. سپس، **اقدامی** را انتخاب می‌کنیم که حداکثر مقدار Q را داشته باشد. از آنجایی که اقدام $right$ دارای حداکثر مقدار Q است، ما اقدام حرکت به *راست* را در حالت **D** انتخاب می‌کنیم:



شکل ۹.۲: شبکه Q عمیق

از آنجایی که ما از یک شبکه عصبی برای تقریب **مقدار** Q استفاده می‌کنیم، این شبکه عصبی را شبکه Q می‌نامند و اگر از یک شبکه عصبی عمیق برای تقریب **ارزش** Q استفاده کنیم، آنگاه آن شبکه عصبی عمیق را **شبکه عمیق** Q (**DQN**) می‌نامند.

ما می‌توانیم تابع Q خود را با $Q_\theta(s,a)$ نشان دهیم، جایی که **پارامتر** θ در زیرنویس این نماد، نشان می‌دهد **تابع** Q ما توسط θ پارامتری شده، که همان θی پارامتر شبکه عصبی ما است.

ما پارامتر شبکه، یعنی θ را با مقادیر تصادفی (دلبخواهی)، مقداردهی اولیه می‌کنیم و از روی آن، **تابع** Q (مقادیر Q)

را تقریب می‌زنیم، اما از آنجایی که θ را با مقادیر تصادفی، مقداردهی اولیه کرده‌ایم، تابع تقریبی Q، بهینه نخواهد بود. بنابراین، ما شبکه را چندین تکرار برای یافتن پارامتر بهینه θ آموزش می‌دهیم. هنگامی‌که θ بهینه را پیدا کنیم، **تابع** Q بهینه را هم خواهیم داشت. سپس می‌توان **سیاست** بهینه را از **تابع** Q بهینه استخراج کرد.

بسیار خب، اما چگونه می‌توانیم شبکه خود را آموزش دهیم؟ در مورد داده‌های آموزشی و تابع زیان چطور؟ آیا این یک کار طبقه‌بندی است یا رگرسیون؟ اکنون که درک اولیه‌ای از نحوه عملکرد **DQN** پیدا کردیم، در بخش بعدی به جزئیات آن پرداخته و به تمامی این سوالات، جواب می‌دهیم.

آشنایی بیشتر با DQN

در این بخش خواهیم فهمید که **DQN** دقیقا چگونه کار می‌کند. ما یاد گرفتیم که از **DQN** برای تقریب مقدار Qی تمامی اقدامات در یک حالت خاص است. البته مقدار Q فقط یک عدد پیوسته است، بنابراین ما اساسا از **DQN** خود برای انجام یک کار رگرسیون استفاده می‌کنیم.

خب، در مورد داده‌های آموزشی چطور؟ برای این کار، ما از انبارکی به نام **انباره بازپخش**[۱] برای جمع آوری تجربیات عامل استفاده کرده و بر اساس این تجربه، شبکه خود را آموزش می‌دهیم. بیایید انباره بازپخش را با جزئیات بررسی کنیم.

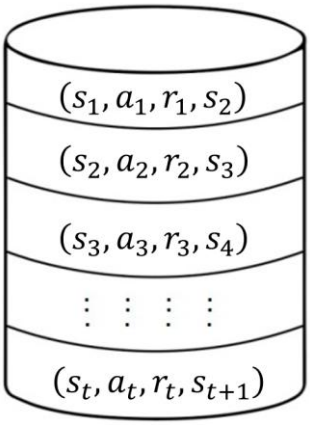
انباره بازپخش

می‌دانیم که عامل ما با انجام اقدامی همچون a از حالت s به حالت بعدی s' منتقل شده و سپس پاداش r را دریافت می‌کند. ما می‌توانیم این **اطلاعات انتقال**[۲] (s, a, r, s') را در انبارکی به نام **انباره بازپخش** یا **بازپخش تجربه**، ذخیره کنیم. انباره بازپخش معمولا با $\mathcal{D}$ نشان داده می‌شود. این اطلاعات انتقالی، اساسا تجربهٔ عامل است. ما تجربهٔ

[۱] Replay Buffer
[۲] Transition Information

عامل را از چندین اپیزود در انباره بازپخش، ذخیره می‌کنیم. ایده کلیدی استفاده از انباره بازپخش برای ذخیرهٔ تجربهٔ عامل این است که می‌توانیم **DQN** خود را با همین تجربه (انتقال) که از انباره ما نمونه‌برداری شده، آموزش دهیم.

انباره بازپخش در اینجا نشان داده شده است:



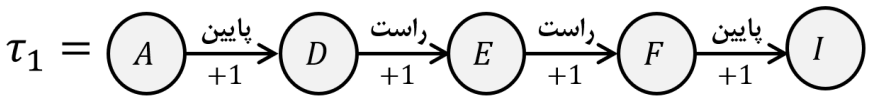
شکل ۹.۳: انباره بازپخش (یا بافر پخش مجدد)

مراحل زیر به ما کمک می‌کند تا بفهمیم چگونه اطلاعات انتقال را در انباره بازپخش $\mathcal{D}$ ذخیره می‌کنیم:

۱. انباره بازپخش $\mathcal{D}$ را با مقادیر اولیه، راه اندازی کنید.
۲. برای هر پردینه (اپیزود)، گام ۳ را انجام دهید.
۳. برای هر گام از اپیزود:
    ۱. یک انتقال ایجاد کنید، یعنی، عمل a را در حالت s انجام دهید، که نتیجتا به حالت بعدی s' خواهید رفت و پاداش r را دریافت کنید.
    ۲. اطلاعات انتقال (s, a, r, s') را در انباره بازپخش $\mathcal{D}$ ذخیره کنید.

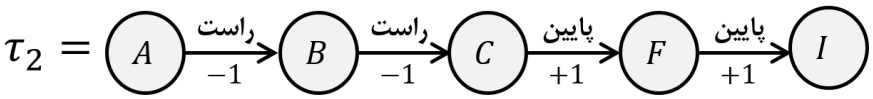
همانطور که در مراحل قبلی توضیح داده شد، ما اطلاعاتِ انتقالِ عامل را در بسیاری از پردینه‌ها (اپیزودها) جمع‌آوری کرده و آن را در انباره بازپخش، ذخیره می‌کنیم. برای درک واضح این موضوع، بیایید محیط جهانی مشبک مورد علاقه خود را در نظر بگیریم. بیایید فرض کنیم دو اپیزود/مسیر زیر را داریم:

پردینه (اپیزود) یک:



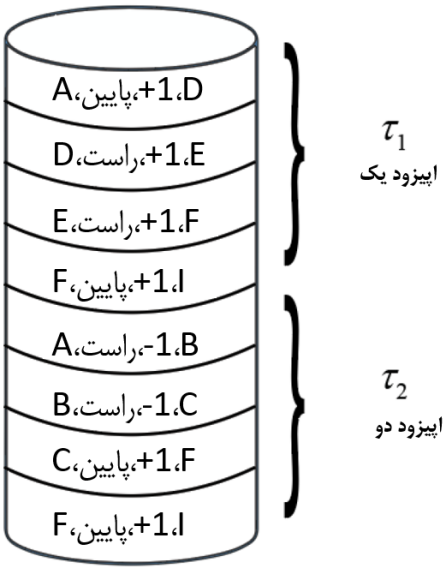
شکل ۹.۴: مسیر ۱

پردینه (اپیزود) دو:



شکل ۹.۵: مسیر ۲

اکنون، این اطلاعات در انباره بازپخش، ذخیره می‌شود، همانطور که شکل ۹.۶ نشان می‌دهد:



تصویر ۹.۶: انباره بازپخش

همانطور که شکل ۹.۶ نشان می‌دهد، ما **اطلاعات انتقال** را با چیدمان یا انبارش متوالی (یعنی یکی پس از دیگری)، ذخیره می‌کنیم. حال ما شبکه خود را با نمونه‌برداری یک دستهٔ کوچک از انتقال‌ها از انباره بازپخش، آموزش می‌دهیم. صبر کنید! در اینجا یک مسئله کوچک وجود دارد. از آنجایی که ما تجربه عامل (یعنی وضعیت انتقالها) را یکی پس

از دیگری به صورت متوالی جمع‌آوری می‌کنیم، تجربه عامل بسیار همبسته خواهد شد. به عنوان مثال، همانطور که در شکل قبلی نشان داده شده است، انتقال‌ها با ردیفهای بالا و پایین مرتبط خواهند بود. اگر شبکه خود را با این تجربه همبسته، آموزش دهیم، شبکه عصبی ما به راحتی **بیش-برازش**[۱] می‌شود. بنابراین، برای مبارزه با این موضوع، یک دسته کوچک از انتقال‌ها را بطور **تصادفی** (دلبخواه) از انباره بازپخش نمونه‌برداری می‌کنیم و شبکه را آموزش می‌دهیم.

توجه داشته باشید که انباره بازپخش، اندازه محدودی دارد، یعنی یک انباره بازپخش، فقط تعداد ثابتی از تجربیات عامل را ذخیره می‌کند. بنابراین، وقتی انباره پر شد، تجربه قدیمی را با تجربه جدید جایگزین می‌کنیم. یک انباره بازپخش معمولا بر اساس روش اولین ورودی-اولین خروجی[۲] (**FIFO**) عمل می‌کند بنابراین، اگر انباره برای پذیرش تجربه جدید ظرفیت نداشته باشد، تجربه قدیمی را حذف کرده و تجربه جدید را به انباره اضافه می‌کنیم.

ما آموختیم که شبکه خود را با نمونه‌برداری تصادفی یک دسته کوچک تجربه از انباره، آموزش می‌دهیم. اما آموزش دقیقا چگونه انجام می‌شود؟ چگونه شبکه ما یاد می‌گیرد که با استفاده از این دسته کوچک از نمونه‌ها، تابع Q بهینه را تقریب بزند؟ این دقیقا همان چیزی است که در بخش بعدی به آن می‌پردازیم.

تابع زیان

ما یاد گرفتیم که در **DQN**، هدف ما **پیش‌بینی مقدار** Q است که یک مقدار پیوسته است. بنابراین، در **DQN** ما اساسا یک کار رگرسیون را انجام می‌دهیم. ما به طور کلی از **میانگین مربعات خطا (MSE)** به عنوان تابع ضرر برای کار رگرسیون استفاده می‌کنیم. **MSE** را می‌توان به عنوان میانگین مجذور اختلاف بین مقدار هدف و مقدار پیش‌بینی تعریف کرد، همانطور که در اینجا نشان داده شده است:

$$\text{MSE} = \frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2$$

[۱] Overfit
[۲] First in First Out

جایی که y مقدار **هدف** است، $\hat{y}$ مقدار **پیش‌بینی** شده و K تعداد نمونه‌های آموزشی است.

اکنون، بیایید یاد بگیریم که چگونه از **MSE** در **DQN** استفاده کنیم و شبکه را آموزش دهیم. ما می‌توانیم شبکه خود را با به حداقل رساندن **MSE** بین **مقدار** Q **هدف** و **مقدار** Q **پیش‌بینی** شده آموزش دهیم. خب، چگونه می‌توانیم مقدار هدف Q را بدست آوریم؟ مقدار Q هدف ما باید مقدار Q بهینه باشد تا بتوانیم با به حداقل رساندن خطای بین مقدار Q بهینه و مقدار Q پیش‌بینی، شبکه خود را آموزش دهیم. اما چگونه می‌توانیم مقدار بهینه Q را محاسبه کنیم؟ اینجاست که **معادله بلمن** به ما کمک می‌کند. در فصل ۳، **معادله بلمن** و **برنامه‌ریزی پویا**، متوجه شدیم که مقدار بهینه Q را می‌توان با استفاده از معادله بهینه بلمن به دست آورد:

$$Q^*(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

که در آن $R(s,a,s')$ نشان دهنده **پاداش** فوری r است که هنگام انجام یک عمل a در حالت s و حرکت به حالت بعدی یعنی s' به دست می‌آوریم، بنابراین می‌توانیم $R(s,a,s')$ را با r بصورت زیر نشان دهیم:

$$Q^*(s,a) = \underset{s' \sim P}{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

در معادله بالا، می‌توانیم نماد انتظار را حذف کنیم. ما مقدار انتظاری را با نمونه‌برداری از تعداد K انتقال از انباره بازپخش و در نظر گرفتن مقدار متوسط آنها، تخمین می‌زنیم؛ در این باره، بعدا بیشتر سخن خواهیم گفت.

بنابراین، طبق معادله بهینه‌سازی بلمن، **مقدار** Q بهینه فقط مجموع **پاداش** و حداکثر **مقدار** Q تخفیف‌یافتهٔ جفت حالت–عمل بعدی است ، یعنی:

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a') \qquad (۱)$$

بنابراین، ما می‌توانیم ضرر خود را به عنوان تفاوت بین **مقدار هدف** (مقدار Q بهینه) و **مقدار پیش‌بینی شده** (مقدار Q پیش‌بینی شده توسط **DQN** تعریف کنیم و تابع ضرر L را به صورت زیر بیان کنیم:

$$L(\theta) = Q^*(s,a) - Q_\theta(s,a)$$

با جایگزینی معادله (۱) در معادله قبلی، می‌توانیم بنویسیم:

$$L(\theta) = r + \gamma \max_{a'} Q(s',a') - Q_\theta(s,a)$$

ما **مقدار Q پیش‌بینی شده** را با استفاده از شبکه‌ای که با θ پارامتری‌شده، محاسبه می‌کنیم. چگونه می‌توانیم **مقدار هدف** را محاسبه کنیم؟ در اینجا یاد گرفتیم که مقدار هدف: مجموع پاداش و حداکثر مقدار Q تخفیف‌دار مربوط به جفت state-action بعدی است. چگونه مقدار Q جفت حالت–عمل بعدی را محاسبه کنیم؟

$$L(\theta) = r + \gamma \max_{a'} \boxed{Q(s',a')} - Q_\theta(s,a)$$

یعنی چگونه مقدار $\boxed{Q(s',a')}$ را محاسبه کنیم؟

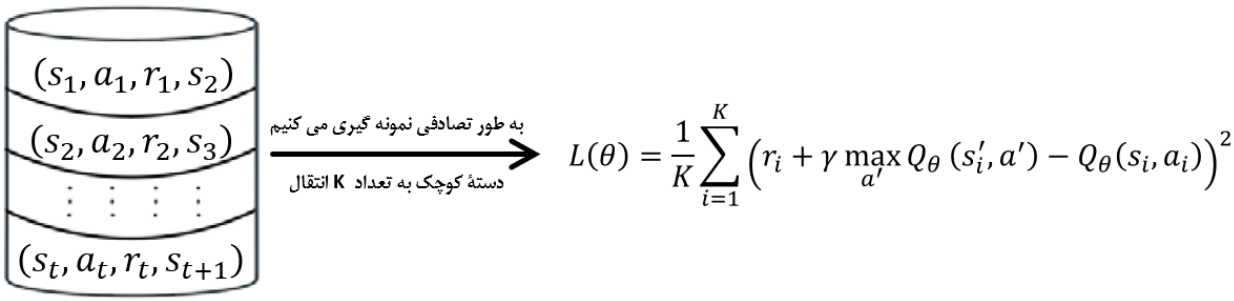
مشابه مقدار Q پیش‌بینی شده، می‌توانیم مقدار Q جفت حالت–عمل بعدی را در هدف با استفاده از همان **DQN** پارامتری شده با θ محاسبه کنیم. بنابراین، می‌توانیم تابع ضرر خود را به صورت زیر بازنویسی کنیم:

$$L(\theta) = r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)$$

همانطور که نشان داده شده است، هم **مقدار هدف** (مقدار Q بهینه) و هم **مقدار Q پیش‌بینی شده** توسط θ، پارامتری شده‌اند.

به جای محاسبه ضرر، در قالب تفاوت بین مقدار Q هدف و مقدار Q پیش‌بینی، می‌توانیم از **MSE** به عنوان تابع ضرر خود استفاده می‌کنیم. ما یاد گرفتیم که تجربه عامل را در انبارکی به نام انباره بازپخش ذخیره کرده‌ایم. بنابراین، ما به طور تصادفی یک دستهٔ کوچک[۱] به *تعداد* K انتقال (s,a,r,s') از انباره بازپخش، نمونه‌برداری کرده و آموزش شبکه را با هدف به حداقل رساندن **MSE**، (همانند شکل زیر) انجام می‌دهیم:

[۱] Minibatch



شکل ۹.۷: تابع زیان **DQN**

بنابراین ، تابع ضرر ما را می‌توان به صورت زیر بیان کرد:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_\theta(s_i', a') - Q_\theta(s_i, a_i)\right)^2$$

برای سادگی نمادگذاری، می‌توانیم مقدار هدف را با y نشان داده و معادله قبلی را به صورت زیر بازنویسی کنیم:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(y_i - Q_\theta(s_i, a_i)\right)^2$$

جایی که داریم:

$$y_i = r_i + \gamma \max_{a'} Q_\theta(s_i', a')$$

ما آموخته ایم که **مقدار هدف** فقط مجموع **پاداش** و حداکثر **مقدار** Q مخفف (تنزیلی یا تخفیف داده شده) مربوط به جفت state-action بعدی است. اما اگر حالت بعدی یعنی s' یک حالت پایانی باشد چه؟ اگر حالت بعدی s' پایانه (ترمینال) باشد، ما نمی‌توانیم مقدار Q را محاسبه کنیم زیرا هیچ اقدامی‌در حالت پایانه انجام نمی‌دهیم، بنابراین در این صورت، مقدار هدف فقط همان پاداش خواهد بود، همانطور که در اینجا نشان داده شده است:

$$y_i = \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_\theta(s', a') & \text{if } s' \text{ is not terminal} \end{cases}$$

بنابراین، تابع ضرر ما به صورت زیر می‌شود:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(y_i - Q_\theta(s_i, a_i)\right)^2$$

ما شبکه خود را با به حداقل رساندن تابع ضرر آموزش می‌دهیم. ما می‌توانیم با یافتن مقدار بهینه پارامتر θ، تابع ضرر را به حداقل برسانیم. بنابراین، ما از گرادیان نزولی برای یافتن پارامتر بهینه θ استفاده می‌کنیم. ما گرادیان تابع ضرر خود، یعنی $\nabla_\theta L(\theta)$ را محاسبه کرده و پارامتر شبکه θ را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta - \alpha\nabla_\theta L(\theta)$$

شبکه هدف[1]

در بخش آخر، یاد گرفتیم که شبکه را با به حداقل رساندن تابع ضرر، آموزش می‌دهیم. تابع ضرر، همانا **MSE** بین مقدار **هدف** و مقدار **پیش‌بینی** است، همانطور که در اینجا نشان داده شده است:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_\theta\left(s_i', a'\right) - Q_\theta(s_i, a_i)\right)^2$$

با این حال، یک مشکل کوچک در تابع ضرر ما وجود دارد. ما آموخته‌ایم که مقدار هدف فقط مجموع پاداش و حداکثر مقدار Q مخفف (تنزیل یافته) جفت state-action بعدی است. ما، هم مقدار Qی جفت حالت–عمل بعدی در هدف و هم مقادیر Qهای پیش‌بینی شده را با استفاده از همان شبکه پارامتری شده با θ بدست می‌آوریم، همانطور که در اینجا نشان داده شده است:

[1] Target Network

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} \underbrace{Q_\theta(s', a')}_{Compute\ using\ \theta} - \underbrace{Q_\theta(s_i, a_i)}_{Compute\ using\ \theta}\right)^2$$

از آنجایی که هم **مقدار هدف** و هم **مقدار پیش‌بینی شده** به یک پارامتر، یعنی θ بستگی دارند، این امر باعث بی‌ثباتی در **MSE** شده و یادگیری شبکه، بخوبی انجام نمی‌شود. این مشکل، همچنین باعث واگرایی زیادی در خلال آموزش می‌شود.

بیایید این مشکل را با یک مثال ساده، تشریح کنیم. ما در اینجا، به دلخواه، اعدادی را در نظر می‌گیریم تا درک آن آسانتر شود. قاعدتاً ما باید سعی می‌کنیم تفاوت بین مقدار هدف و مقدار پیش‌بینی را به حداقل برسانیم. بنابراین، در هر تکرار، گرادیان ضرر را محاسبه کرده و پارامتر شبکه خود یعنی θ را به‌روز می‌کنیم تا بتوانیم مقدار پیش‌بینی خود را با مقدار هدف یکسان کنیم.

فرض کنید در تکرار ۱، مقدار هدف ۱۳ و مقدار پیش‌بینی شده ۱۱ باشد. بنابراین، پارامتر θ خود را به‌روز می‌کنیم تا مقدار پیش‌بینی را با مقدار هدف که ۱۳ است، مطابقت دهیم. اما در تکرار بعدی، مقدار هدف به ۱۵ تغییر می‌کند و مقدار پیش‌بینی به ۱۳ تبدیل می‌شود، زیرا پارامتر شبکه یعنی θ را به‌روز کردیم. بنابراین، دوباره پارامتر θ را به‌روز می‌کنیم تا مقدار پیش‌بینی را با مقدار هدف مطابقت دهیم، که اکنون ۱۵ است. اما در تکرار بعدی، مقدار هدف به ۱۷ تغییر می‌کند و مقدار پیش‌بینی ۱۵ می‌شود زیرا پارامتر شبکه خود θ را به‌روز کردیم.

همانطور که جدول ۹.۲ نشان می‌دهد، در هر تکرار، مقدار پیش‌بینی شده سعی می‌کند همان مقدار هدف باشد، که مدام در حال تغییر است:

ارزش پیش بینی شده	ارزش هدف
۱۱	۱۳
۱۳	۱۵
۱۵	۱۷

جدول ۹.۲: مقدار هدف و مقدار پیش‌بینی شده

این وضعیت بدان دلیل است که مقادیر هدف و پیش‌بینی هر دو به یک پارامتر یعنی θ، بستگی دارند. اگر θ را به‌روز کنیم، هم مقادیر هدف و هم مقادیر پیش‌بینی تغییر می‌کنند. بنابراین، مقدار پیش‌بینی همچنان سعی می‌کند با مقدار هدف یکسان باشد، اما مقدار هدف به دلیل به‌روز رسانی پارامتر شبکه یعنی θ تغییر می‌کند.

چگونه می‌توانیم از این امر جلوگیری کنیم؟ آیا می‌توانیم مقدار هدف را برای مدتی مسدود (فریز) کنیم و فقط مقدار پیش‌بینی شده را محاسبه کنیم تا مقدار پیش‌بینی شده ما با مقدار هدف مطابقت منطقی داشته باشد؟ بله! برای انجام این کار، شبکه عصبی دیگری به نام **شبکه هدف** را برای محاسبه مقدار Qی جفت حالت–عمل بعدی در هدف معرفی می‌کنیم. پارامتر **شبکه هدف** با θ' نشان داده می‌شود. بنابراین، شبکه Qی **عمیق اصلی** ما که برای پیش‌بینی مقادیر Q استفاده می‌شود، پارامتر بهینه θ را با استفاده از گرادیان نزولی یاد می‌گیرد. **شبکه هدف** برای مدتی منجمد شده و سپس پارامتر **شبکه هدف** یعنی θ' فقط با کپی کردن پارامتر شبکهٔ Qی **عمیق اصلی**، یعنیθ، به‌روز می‌شود. انجماد **شبکه هدف** برای مدتی و سپس به‌روز رسانی پارامتر θ' آن از روی پارامتر **شبکه اصلی** θ، باعث تثبیت آموزش می‌شود.

لذا، اکنون تابع loss ما را می‌توان به صورت زیر بازنویسی کرد:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a') - Q_\theta(s_i, a_i)\right)^2$$

بنابراین، مقدار Q جفت حالت–عمل بعدی بعنوان **مقدار هدف** توسط **شبکه هدف** با پارامتر θ' محاسبه می‌شود و مقدار Q **پیش‌بینی شده** توسط **شبکه اصلی** ما از روی پارامتر θ بدست می‌آید:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(r_i + \gamma \max_{a'} \underbrace{Q_{\theta'}(s', a')}_{\text{Compute using } \theta'} - \underbrace{Q_\theta(s_i, a_i)}_{\text{Compute using } \theta}\right)^2$$

برای سادگی نماد، می‌توانیم مقدار **هدف** خود را با y نشان دهیم و معادله قبلی را به صورت زیر بازنویسی کنیم:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}\left(y_i - Q_\theta(s_i, a_i)\right)^2$$

ما هم اینک چندین مفهوم مربوط به **DQN** را یاد گرفته‌ایم، از جمله بازپخش تجربه، تابع ضرر و شبکه هدف. در بخش بعدی، همه این مفاهیم را کنار هم قرار می‌دهیم و می‌بینیم که **DQN** چگونه کار می‌کند.

همایندِ مطالب DQN

ابتدا پارامتر **شبکه اصلی** θ را با مقادیر تصادفی، مقداردهی اولیه می‌کنیم. ما یاد گرفتیم که **پارامتر شبکه هدف** فقط یک کپی از **شبکه اصلی** است. بنابراین، ما پارامتر **شبکه هدف** θ' را فقط با کپی کردن پارامتر **شبکه اصلی** θ مقداردهی اولیه می‌کنیم. ما همچنین انباره بازپخش $\mathcal{D}$ را مقداردهی اولیه می‌کنیم.

اکنون، برای هر گام از اپیزود، وضعیت محیط را به شبکه خود تغذیه می‌کنیم و شبکه در پاسخ، همه مقادیر Qی اقدامات ممکن در آن حالت را بعنوان خروجی می‌دهد. سپس، اقدامی را انتخاب می‌کنیم که حداکثر مقدار Q را دارد:

$$a = \arg\max_a Q_\theta(s, a)$$

البته اگر فقط اقدامی را انتخاب کنیم که بالاترین مقدار Q را داشته باشد، هیچ اقدام جدیدی را بررسی نخواهیم کرد. بنابراین، برای جلوگیری از این امر، برخی اقدامات را با استفاده از سیاست اپسیلون–حریصانه انتخاب می‌کنیم. بر اساس سیاست اپسیلون–حریصانه، یک اقدام تصادفی را با احتمال اپسیلون، و بهترین اقدام با حداکثر بازده را با احتمال یک منهای اپسیلون انتخاب می‌کنیم.

از آنجا که ما پارامتر شبکه خود یعنی θ را با مقادیر تصادفی مقداردهی اولیه کردیم، **اقدامی** که با گرفتن حداکثر Q انتخاب می‌کنیم، عمل بهینه نخواهد بود. اما اشکالی ندارد، ما به سادگی عمل انتخاب شده را انجام داده، به حالت بعدی می‌رویم و **پاداش** را دریافت می‌کنیم. اگر عمل خوب باشد، پاداش مثبت دریافت می‌کنیم و اگر بد باشد، پاداش

منفی خواهد بود. ما تمام این اطلاعات انتقال (s, a, r, s') را در انباره بازپخش $\mathcal{D}$ ذخیره می‌کنیم.

در مرحله بعد، به طور تصادفی یک دسته کوچک، به اندازهٔ K انتقال، را از انباره بازپخش نمونه‌برداری کرده و ضرر را محاسبه می‌کنیم. ما آموخته‌ایم که تابع ضرر ما به صورت زیر محاسبه می‌شود

$$L(\theta) = \frac{}{K}\sum_{i=}^{K}\left(y_i - Q_\theta(s_i, a_i)\right)$$

که در آن رابطه، مقدار y بصورت زیر است:

$$y_i = r_i + \gamma \max_{a'} Q_{\theta'}\,(s_i', a')$$

در تکرارهای اولیه، مقدار ضرر بسیار زیاد خواهد بود زیرا پارامتر شبکه ما یعنی θ فقط مقادیر تصادفی است. برای به حداقل رساندن زیان، ما نشیب زیان را محاسبه کرده و **پارامتر شبکه اصلی** یعنی θ را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta - \alpha \nabla_\theta L(\theta)$$

ما **پارامتر شبکه هدف** یعنی θ' را در هر مرحله زمانی به‌روز نمی‌کنیم. زیرا ما **پارامتر شبکه هدف** یعنی θ' را برای چندین مرحله زمانی مسدود (فریز) کرده و سپس مقادیر **پارامتر شبکه اصلی** یعنی θ را عنوان مقادیر **پارامتر شبکه هدف** یعنی θ' کپی می‌کنیم.

ما مراحل قبلی را برای چندین اپیزود تکرار کرده تا مقدار Q بهینه را تقریب بزنیم. هنگامی‌که مقدار Q بهینه را به دست آوردیم، **سیاست بهینه** را از آنها استخراج می‌کنیم. برای درک دقیق‌تر، الگوریتم **DQN** در بخش بعدی آورده شده است.

الگوریتم DQN

الگوریتم **DQN** در مراحل زیر ارائه شده است:

۱. پارامتر **شبکه اصلی** θ را با مقادیر تصادفی مقداردهی اولیه کنید.

۲. پارامتر **شبکه هدف** θ' را با کپی کردن پارامتر **اصلی شبکه** θ مقداردهی اولیه کنید.

۳. انباره بازپخش $\mathcal{D}$ را راه اندازی کنید.

٤. برای N پردینه (اپیزود)، گام ۵ را انجام دهید.

۵. برای هر گام در پردینه (اپیزود)، یعنی برای $t = 0,\ldots,T\text{-}1$

۱. حالت s را مشاهده کنید و با استفاده از سیاست اپسیلون حریصانه، یک **عمل** را انتخاب کنید، یعنی با احتمال ε، اقدام تصادفی a را انتخاب کرده و با احتمال $\varepsilon - ۱$، عمل دیگری را بصورت زیر انتخاب کنید:

$$a = \arg\max_{a} Q_\theta(s, a)$$

۲. **اقدام** انتخاب شده را انجام داده و به حالت بعدی s' رفته و **پاداش** r را دریافت کنید.

۳. اطلاعات انتقال را در انباره بازپخش $\mathcal{D}$ ذخیره کنید.

٤. بطور تصادفی یک دسته کوچک شامل K انتقال از انباره بازپخش $\mathcal{D}$ نمونه‌برداری کنید.

۵. **مقدار هدف**، یعنی تابع زیر را محاسبه کنید:

$$y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a')$$

٦. تابع زیان را محاسبه کنید:

۷.

$$L(\theta) = \frac{۱}{K}\sum_{i=۱}^{K}\left(y_i - Q_\theta(s_i, a_i)\right)^۲$$

۸. نشیبهای زیان را محاسبه کرده و پارامتر شبکه اصلی θ را با استفاده از نزول گرادیان به‌روز کنید:

$$\theta = \theta - \alpha\nabla_\theta L(\theta)$$

۹. پارامتر **شبکه هدف** یعنی θ' را برای چندین گام زمانی فریز کرده و سپس آن را فقط با کپی کردن

پارامتر **شبکه اصلی** θ به‌روز کنید.

اکنون که فهمیدیم **DQN** چگونه کار می‌کند، در بخش بعدی، نحوه پیاده‌سازی آن را یاد خواهیم گرفت.

انجام بازی آتاری با استفاده از DQN

آتاری ۲۶۰۰ یک کنسول بازی ویدیویی محبوب از یک شرکت بازی‌سازی به نام آتاری است. کنسول بازی آتاری چندین بازی محبوب مانند Pong، Space Invaders، Ms.Pac-Man، Breakout، Centipede و بسیاری دیگر را ارائه می‌دهد. در این بخش یاد خواهیم گرفت که چگونه یک **DQN** برای انجام بازیهای آتاری بسازیم. ابتدا، بیایید معماری **DQN** را برای انجام بازیهای آتاری بررسی کنیم.

معماری DQN

در محیط آتاری، تصویر صفحه نمایش بازی، وضعیت محیط است. بنابراین، ما فقط تصویر صفحه بازی را به عنوان ورودی به **DQN** تغذیه می‌کنیم و مقادیر Q تمام اقدامات موجود در حالت را برمی‌گرداند. از آنجایی که ما با تصاویر سر و کار داریم، به جای استفاده از یک **شبکه عصبی عمیق وانیلی**[۱] برای تقریب مقدار Q، می‌توانیم از یک **شبکه عصبی همتابی** (**CNN**) استفاده کنیم زیرا برای مدیریت تصاویر بسیار موثر است.

بنابراین، اکنون **DQN** ما یک **CNN** است. ما تصویر صفحه نمایش بازی (یعنی حالت بازی) را بعنوان ورودی به **CNN** می‌دهیم و شبکه مقادیر Qی تمام اقدامات ممکن در این حالت خاص را بعنوان خروجی می‌دهد.

---

[۱] Vanilla

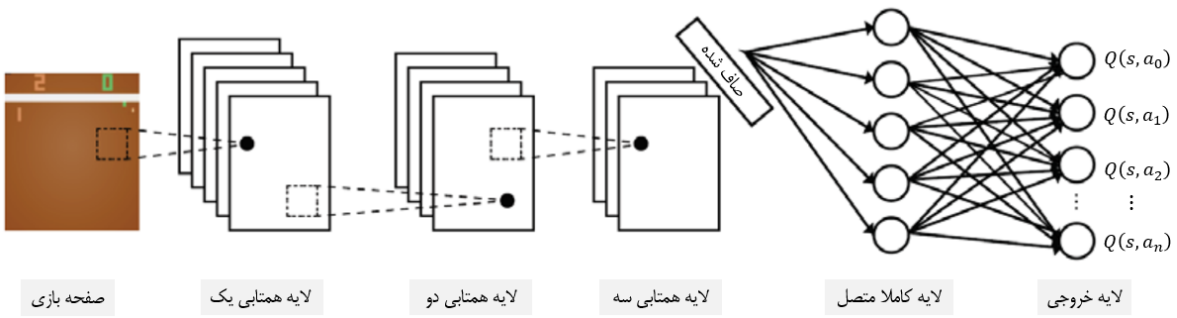
در زمینه شبکه‌های عصبی، "Vanilla" به معنای ساده یا معمولی است. این اصطلاح معمولاً برای اشاره به یک شبکه عصبی ساده و ابتدایی استفاده می‌شود که بدون هیچ پیچیدگی یا افزودن ویژگی‌های خاص، تنها از لایه‌های کاملاً متصل (Fully Connected) تشکیل شده است. در این نوع شبکه‌ها، هر نرون در یک لایه به تمام نرون‌های لایه بعد متصل است.

Vanilla GAN نیز به همین ترتیب، نوعی از شبکه‌های مولد تخاصمیا تقابلی (GAN) است که در آن، مولد و ممیز به صورت شبکه‌های عصبی کاملاً متصل(MLP) ساده پیاده‌سازی می‌شوند. این نوع ازGANها از یک فرمول بهینه‌سازی حداقل–حداکثر استفاده می‌کنند و تلفات آنتروپی متقابل سیگموئید را برای بهینه‌سازی ممیز به کار می‌برند.

Vanilla RNN نیز نوعی از شبکه‌های عصبی بازگشتی است که از یک ساختار ساده استفاده می‌کند. در این نوع RNN، ورودی فعلی و حالت پنهان قبلی به عنوان ورودی گرفته می‌شود و خروجی فعلی و حالت پنهان بعدی تولید می‌شود.

همانطور که شکل ۹.۸ نشان می‌دهد، با توجه به تصویر صفحه بازی، لایه‌های همتابی، ویژگیها را از تصویر استخراج کرده و یک نقشه ویژگی تولید می‌کنند. در مرحله بعد، نقشه ویژگی را صاف کرده و نقشه ویژگی مسطح را به عنوان ورودی به شبکه پیشخور (فیدفوروارد) تغذیه می‌کنیم. این شبکه پیشخور، نقشه ویژگی مسطح مورد نظر را به عنوان ورودی می‌گیرد و مقادیر Q تمام اقدامات موجود در این حالت را برمی‌گرداند.

توجه داشته باشید که ما عملیات ادغام را انجام نمی‌دهیم. عملیات ادغام زمانی مفید است که وظایفی مانند تشخیص شیء، طبقه بندی تصویر و غیره را انجام دهیم که در آن موقعیت شیء را در تصویر در نظر نگرفته و فقط می‌خواهیم بدانیم که آیا شیء مورد نظر در تصویر وجود دارد یا خیر. به عنوان مثال، اگر بخواهیم تشخیص دهیم که آیا سگی در یک تصویر وجود دارد یا خیر؟ در این صورت، فقط به دنبال این می‌گردیم که آیا سگی در تصویر وجود دارد یا خیر و موقعیت سگ را در تصویر بررسی نمی‌کنیم. بنابراین، در این مورد، از یک عملیات ادغام یا تجمیع[۱] برای شناسایی اینکه آیا سگ در تصویر وجود دارد یا خیر، صرف نظر از موقعیت سگ استفاده می‌شود.



DQN

اما در تنظیمات ما، عملیات ادغام (یا تجمیع) نباید انجام شود زیرا برای درک وضعیت فعلی بازی، موقعیت بسیار مهم است. به عنوان مثال، در پُنگ[۲]، ما فقط نمی‌خواهیم طبقه‌بندی کنیم که آیا توپی روی صفحه بازی وجود دارد یا خیر. ما می‌خواهیم موقعیت توپ را بدانیم تا بتوانیم عمل بهتری انجام دهیم. بنابراین، ما عملیات ادغام را در معماری **DQN** خود لحاظ نمی‌کنیم.

اکنون که معماری **DQN** را برای انجام بازیهای آتاری درک کردیم، در بخش بعدی پیاده‌سازی آن را شروع خواهیم

[۱] Pooling Operation
[۲] Pong

کرد.

آشنایی عملی با DQN

بیایید **DQN** را برای انجام بازی Ms Pacman پیاده سازی کنیم. ابتدا، بیایید کتابخانه های لازم را وارد کنیم:

```
import random
import gym
import numpy as np
from collections import deque
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten, Conv2D,
MaxPooling2D
from tensorflow.keras.optimizers import Adam
```

اکنون، بیایید محیط بازی Ms Pacman را با استفاده از Gym ایجاد کنیم:

```
env = gym.make("MsPacman-v0")
```

اندازه حالت را تنظیم کنید:

```
state_size = (88, 80, 1)
```

تعداد اقدامات را دریافت کنید:

```
action_size = env.action_space.n
```

صفحه بازی را پیش‌پردازش کنید

ما یاد گرفتیم که حالت بازی (تصویری از صفحه بازی) را به عنوان ورودی به **DQN** که یک **CNN** است تغذیه می‌کنیم و مقادیر Q تمام اقدامات آن حالت را خروجی می‌دهد. با این حال، تغذیه مستقیم یک تصویر صفحه نمایش خام، کارآمد نیست، زیرا اندازه صفحه نمایش خام بازی ۳×۱۶۰×۲۱۰ خواهد بود. این از نظر محاسباتی گران خواهد بود.

برای جلوگیری از این امر، صفحه بازی را از قبل پردازش می‌کنیم و سپس صفحه بازی از پیش پردازش شده را به **DQN** تغذیه می‌کنیم. ابتدا تصویر صفحه نمایش بازی را برش داده و اندازه آن را تغییر می‌دهیم، تصویر را به مقیاس خاکستری تبدیل کرده، آن را عادی می‌کنیم و سپس تصویر را به ۸۸×۸۰×۱ تغییر شکل می‌دهیم. در مرحله بعد، این تصویر صفحه نمایش بازی از پیش‌پردازش شده را به عنوان ورودی به **CNN** تغذیه می‌کنیم که مقادیر Q را برمی‌گرداند.

حال، بیایید تابعی به نام `preprocess_state` تعریف کنیم که حالت بازی (تصویر صفحه بازی) را به عنوان ورودی می‌گیرد و حالت بازی از پیش‌پردازش شده را برمی‌گرداند:

```
color = np.array([210, 164, 74]).mean()

def preprocess_state(state):
```

تصویر را برش و تغییر اندازه دهید:

```
    image = state[1:176:2, ::2]
```

تصویر را به مقیاس خاکستری تبدیل کنید:

```
    image = image.mean(axis=2)
```

شفافیت تصویر را بهبود دهید:

```
    image[image==color] = 0
```

تصویر را نرمال کنید:

```
    image = (image - 128) / 128 - 1
```

تصویر را تغییر شکل[۱] داده و خروجی بگیرید:

[۱] Reshape

```
    image = np.expand_dims(image.reshape(88, 80, 1), axis=0)

    return image
```

تعریف کلاس DQN

بیایید کلاسی به نام **DQN** را تعریف کنیم که در آن الگوریتم **DQN** را پیاده‌سازی خواهیم کرد. برای درک واضح‌تر، بیایید کد را خط به خط، بررسی کنیم. همچنین می‌توانید از مخزن GitHub کتاب به کد کامل دسترسی داشته باشید:

```
class DQN:
```

تعریف روش init

ابتدا، بیایید روش `init` را تعریف کنیم:

```
    def __init__(self, state_size, action_size):
```

اندازه حالت را تعریف کنید:

```
        self.state_size = state_size
```

اندازه اقدام را تعریف کنید:

```
        self.action_size = action_size
```

انباره بازپخش را تعریف کنید:

```
        self.replay_buffer = deque(maxlen=5000)
```

ضریب تخفیف (فاکتور تخفیف) را تعریف کنید:

```
        self.gamma = 0.9
```

مقدار اپسیلون را تعریف کنید:

```
        self.epsilon = 0.8
```

نرخ به‌روز رسانی را تعریف کنید (نرخی که می‌خواهیم شبکه هدف را بر اساس آن، به‌روز کنیم):

```
        self.update_rate = 1000
```

شبکه اصلی را تعریف کنید:

```
        self.main_network = self.build_network()
```

شبکه هدف را تعریف کنید:

```
        self.target_network = self.build_network()
```

وزن **شبکه اصلی** را در **شبکه هدف** کپی کنید:

```
        self.target_network.set_weights(self.main_network.get_
weights())
```

ساخت DQN

حالا، بیایید **DQN** را بسازیم. ما یاد گرفته‌ایم که برای انجام بازیهای آتاری، از **CNN** به عنوان **DQN** استفاده کنیم که تصویر صفحه بازی را به عنوان ورودی می‌گیرد و مقادیر Q را برمی‌گرداند. ما **DQN** را با سه لایه همتابی (کانولوشن) تعریف می‌کنیم. لایه‌های همتابی، ویژگیها را از تصویر استخراج کرده و نقشه‌های ویژگی را خروجی می‌دهند و سپس نقشه ویژگی به دست آمده توسط لایه‌های همتابی را صاف می‌کنیم و نقشه‌های ویژگی مسطح شده را به شبکه پیشخور (لایه کاملا متصل) تغذیه می‌کنیم تا مقادیر Q را برگرداند:

```
    def build_network(self):
```

اولین لایه همتابی (کانولوشن) را تعریف کنید:

```
        model = Sequential()
        model.add(Conv2D(32, (8, 8), strides=4, padding='same', input_
shape=self.state_size))
        model.add(Activation('relu'))
```

لایه همتابی دوم را تعریف کنید:

```
        model.add(Conv2D(64, (4, 4), strides=2, padding='same'))
        model.add(Activation('relu'))
```

سومین لایه همتابی را تعریف کنید:

```
        model.add(Conv2D(64, (3, 3), strides=1, padding='same'))
        model.add(Activation('relu'))
```

نقشه‌های ویژگی به دست آمده در نتیجه لایه همتابی سوم را صاف کنید:

```
        model.add(Flatten())
```

نقشه‌های مسطح را به لایه کاملا متصل، تغذیه کنید:

```
        model.add(Dense(512, activation='relu'))
        model.add(Dense(self.action_size, activation='linear'))
```

مدل را با تابع ضرر به صورت **MSE** کامپایل کنید:

```
        model.compile(loss='mse', optimizer=Adam())
```

نتایج مدل را برگردانید:

```
        return model
```

ذخیره انتقال[1]

[1] Storing the transition

ما آموخته‌ایم که می‌توانیم **DQN** را با نمونه‌برداری تصادفی یک دسته کوچک از انتقالهای درون انباره بازپخش، آموزش دهیم. بنابراین، تابعی به نام store_transition را تعریف می‌کنیم که اطلاعات انتقال را در انباره بازپخش، ذخیره می‌کند:

```
    def store_transistion(self, state, action,
                          reward, next_state, done):
        self.replay_buffer.append((state, action,
                                   reward, next_state, done))
```

تعریف سیاست اپسیلون حریصانه

ما یاد گرفتیم که در **DQN**، برای مراقبت از مبادله اکتشاف–انتفاع، با استفاده از سیاست اپسیلون–حریصانه اقدام را انتخاب می‌کنیم. بنابراین، اکنون تابعی به نام epsilon_greedy را برای انتخاب یک عمل با استفاده از سیاست اپسیلون–حریصانه تعریف می‌کنیم:

```
    def epsilon_greedy(self, state):
        if random.uniform(0,1) < self.epsilon:
            return np.random.randint(self.action_size)
        Q_values = self.main_network.predict(state)
        return np.argmax(Q_values[0])
```

آموزش را تعریف کنید

حال بیایید تابعی به نام `train` را برای شبکه آموزشی تعریف کنیم:

```
    def train(self, batch_size):
```

دسته کوچکی از انتقالها را از درون انباره بازپخش، نمونه‌گیری کنید:

```
        minibatch = random.sample(self.replay_buffer, batch_size)
```

مقدار هدف را با استفاده از شبکه هدف محاسبه کنید:

```
        for state, action, reward, next_state, done in minibatch:
            if not done:
                target_Q = (reward + self.gamma * np.amax(
                    self.target_network.predict(next_state)))
            else:
                target_Q = reward
```

مقدار پیش‌بینی شده را با استفاده از شبکه اصلی محاسبه کنید و مقدار پیش‌بینی شده را در `Q_values` ذخیره کنید:

```
            Q_values = self.main_network.predict(state)
```

مقدار هدف را به‌روز کنید:

```
            Q_values[0][action] = target_Q
```

شبکه اصلی را آموزش دهید:

```
            self.main_network.fit(state, Q_values, epochs=1,
                                  verbose=0)
```

به‌روز رسانی شبکه هدف

حال ، بیایید تابعی به نام `update_target_network` را برای به‌روز رسانی وزن **شبکه هدف** با کپی کردن از **شبکه اصلی** تعریف کنیم:

```
    def update_target_network(self):
        self.target_network.set_weights(self.main_network.get_
weights())
```

آموزش DQN

حالا، بیایید شبکه را آموزش دهیم. ابتدا تعداد اپیزودهایی که می‌خواهیم شبکه را برای آنها آموزش دهیم تنظیم کنیم:

```
num_episodes = 500
```

تعداد گام زمانی را تعریف کنید:

```
num_timesteps = 20000
```

اندازه دسته را تعریف کنید:

```
batch_size = 8
```

تعداد «صفحات بازی گذشته» مورد نظر خود را وارد کنید:

```
num_screens = 4
```

کلاس **DQN** را نمونه‌سازی[۱] کنید:

```
dqn = DQN(state_size, action_size)
```

وضعیت اتمام یافتگی را روی `False` تنظیم کنید:

```
done = False
```

`time_step` را مقداردهی اولیه کنید:

```
time_step = 0
```

برای هر اپیزود داریم:

```
for i in range(num_episodes):
```

بازگشت را روی صفر تنظیم کنید:

```
    Return = 0
```

صفحه بازی را پیش‌پردازش کنید:

```
    state = preprocess_state(env.reset())
```

[۱] Instantiate

برای هر گام از اپیزود داریم:

```
    for t in range(num_timesteps):
```

محیط را رندر کنید:

```
        env.render()
```

گام زمانی را به‌روز کنید:

```
        time_step += 1
```

شبکه هدف را به‌روز کنید:

```
        if time_step % dqn.update_rate == 0:
            dqn.update_target_network()
```

اقدام را انتخاب کنید:

```
        action = dqn.epsilon_greedy(state)
```

عمل یا **اقدام** انتخاب شده را انجام دهید:

```
        next_state, reward, done, _ = env.step(action)
```

حالت بعدی را پیش‌پردازش کنید:

```
        next_state = preprocess_state(next_state)
```

اطلاعات انتقال را ذخیره کنید:

```
        dqn.store_transistion(state, action, reward, next_state, done)
```

حالت فعلی را به حالت بعدی به‌روز کنید:

```
        state = next_state
```

مقدار بازگشتی را به‌روز کنید:

```
        Return += reward
```

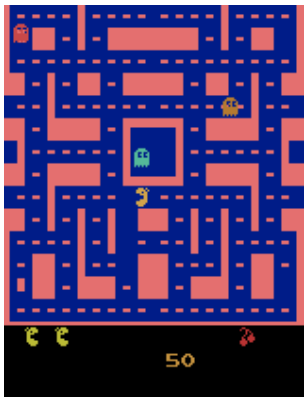
اگر اپیزود تمام شد، بازده را چاپ کنید:

```
        if done:
            print('Episode: ',i, ',' 'Return', Return)
            break
```

اگر تعداد انتقالها در انباره بازپخش بیشتر از اندازه دسته است، شبکه را آموزش دهید:

```
        if len(dqn.replay_buffer) > batch_size:
            dqn.train(batch_size)
```

با رندر کردن محیط، همچنین می‌توانیم مشاهده کنیم که چگونه عامل یاد می‌گیرد که بازی را در یک سری اپیزود انجام دهد:



شکل ۹.۹: عامل **DQN** در حال یادگیری بازی

اکنون که یاد گرفتیم **DQN**ها چگونه کار می‌کنند و چگونه یک **DQN** برای انجام بازیهای آتاری بسازیم، در بخش بعدی، نوع جالبی از **DQN** به نام **DQN** دوتایی یا دوگان را یاد خواهیم گرفت.

DQN دوتایی

ما آموختیم که در **DQN**، **مقدار هدف** به صورت زیر محاسبه می‌شود:

$$y = r + \gamma \max_{a'} Q_{\theta'}(s', a')$$

یکی از مشکلات **DQN** این است که تمایل دارد مقدار Qی جفت state-action بعدی در هدف را بیش‌برآورد کند:

$$y = r + \gamma \underbrace{\max_{a'} Q_{\theta'}(s', a')}_{Over-estimates\ Q\ Value}$$

این بیش‌برآورد به دلیل وجود عملگر حداکثر (max) است. بیایید ببینیم چگونه این بیش‌برآورد با یک مثال اتفاق می‌افتد. فرض کنید ما در یک حالت s' هستیم و سه **عمل** a_1، a_2 و a_3 داریم. فرض کنید a_3 **عمل** بهینه در حالت s' است. وقتی شما مقادیر Qی تمام **اقدامات** موجود در حالت را تخمین s' بزنید، مقدار Q تخمینی، مقداری اغتشاش خواهد داشت و با مقدار واقعی آن متفاوت است. به دلیل این نویز، **عمل** a_2 مقدار Q بالاتری نسبت به **عمل** بهینه a_3 دریافت می‌کند.

می‌دانیم که **مقدار هدف** به صورت زیر محاسبه می‌شود:

$$y = r + \gamma \max_{a'=\{a_1, a_2, a_3\}} Q_{\theta'}(s', a')$$

حال، اگر بهترین عمل را به عنوان اقدامی با حداکثر مقدار، انتخاب کنیم، در نهایت عمل a_2 را به جای عمل بهینه a_3 انتخاب می‌کنیم، همانطور که در اینجا نشان داده شده است:

$$y = r + \gamma Q_{\theta'}(s', a_2)$$

بنابراین، چگونه می‌توانیم از شر این بیش‌برآورد خلاص شویم؟ ما می‌توانیم فقط با تغییر شیوه محاسبه **مقدار هدف**، به صورت زیر، از شر این بیش‌برآورد بگریزیم:

$$y = r + \gamma Q_{\theta'}\left(s', \arg\max_{a'} Q_{\theta'}(s', a')\right)$$

همانطور که مشاهده می‌کنید، اکنون ما **دو تابع** Q در محاسبه مقدار هدف خود داریم. یکی: **تابع** Qی پارامتری شده

توسط پارامتر **شبکه اصلی** θ برای انتخاب **عمل**، و دیگری: **تابع** Qی پارامتری شده توسط پارامتر **شبکه هدف** θ' برای **محاسبه مقدار** Q.

بیایید با شکستن معادلهٔ قبلی به دو مرحله، آنرا بهتر درک کنیم:

۱. **انتخاب اقدام**: ابتدا مقادیر Q تمام جفتهای state-action بعدی را با استفاده از **شبکه اصلی** که توسط θ پارامتری شده است، محاسبه کرده و سپس اقدام a' را برمی‌گزینیم چون حداکثر مقدار Q را دارد:

$$y = r + \gamma Q_{\theta'}(s', \underbrace{\arg\max_{a'} Q_{\theta}(s', a')}_{\substack{Select\ the\ action\ a', which\ has \\ the\ maximum\ Q\ Value\ computed \\ by\ main\ network\ \theta}})$$

۲. **محاسبه مقدار** Q: هنگامی‌که عمل a' را برگزیدیم، آنگاه مقدار Q را با استفاده از **شبکه هدف** پارامتری θ' برای آن عمل انتخاب شده (یعنی a') محاسبه می‌کنیم

$$y = r + \gamma Q_{\theta'}(s', \underbrace{(s', a')}_{\substack{Compute\ the\ Q\ Value\ for\ the \\ selected\ action\ á\ using\ target \\ network\ \theta'}})$$

بیایید با یک مثال، موضوع را بهتر درک کنیم. فرض کنید اسم حالت s' مثلا E باشد، پس می‌توانیم بنویسیم

$$y = r + \gamma Q_{\theta'}\left(E, arg\max_{a'} Q_{\theta}(E, a')\right)$$

ابتدا مقادیر Q همه اقدامات در حالت E را با استفاده از **شبکه اصلی** پارامتری شده با θ، محاسبه می‌کنیم و سپس **اقدامی** را برمی‌گزینیم که حداکثر **مقدار** Q را دارد. بیایید فرض کنیم اقدامی‌که حداکثر مقدار Q را دارد right است:

$$y = r + \gamma Q_{\theta'}(E, \underbrace{\arg\max_{a'} Q_{\theta}(E, a')}_{right})$$

اکنون، می‌توانیم مقدار Q را با استفاده از **شبکه هدف** پارامتری شده با θ'، محاسبه کنیم. البته برای این کار، **اقدامی** را برای محاسبه برمی‌گزینیم که توسط **شبکه اصلی**، انتخاب شده است و در اینجا right است. بنابراین، می‌توانیم بنویسیم:

$$y = r + \gamma Q_{\theta'}(E, \text{right})$$

آیا هنوز مشخص نیست داریم چکار می‌کنیم؟ تفاوت بین نحوه محاسبه مقدار هدف در **DQN** و **DQN** دوگانه در اینجا نشان داده شده است:

DQN دوگانه	**DQN**
$y = r + \gamma \underbrace{Q_{\theta'}(s', \overbrace{arg\max_{a} Q_{\theta}(s', a')}^{Action\ selection\ using\ main\ network\ \theta})}_{Q\ Value\ computation\ using\ target\ network\ \theta'}$	$y = r + \gamma \underbrace{\max_{a'} Q_{\theta'}(s', a')}_{Compute\ using\ target\ network\ \theta'}$

شکل ۹.۱۰: تفاوت بین **DQN** و **DQN** دوگانه

بنابراین، ما یاد گرفتیم که در یک **DQN** دوگانه، **مقدار هدف** را با استفاده از **دو تابع** Q محاسبه می‌کنیم. یک **تابع** Q پارامتری شده توسط پارامتر **شبکه اصلی** θ برای انتخاب اقدامی که حداکثر مقدار Q را دارد. و دیگری **تابع** Q که توسط پارامتر **شبکه هدف** θ' پارامتری شده است، و مقدار Q را با استفاده از **عمل** انتخاب شده توسط **شبکه اصلی** محاسبه می‌کند:

$$y = r + \gamma Q_{\theta'}(s', \arg\max_{a'} Q_{\theta}(s', a'))$$

بجز روش محاسبه مقدار هدف، در سایر قدمها، **DQN** دوگانه دقیقا مانند **DQN** عمل می‌کند. برای وضوح بیشتر، الگوریتم **DQN** دوگان در بخش بعدی آورده شده است.

الگوریتم DQN دوگانه

الگوریتم **DQN** دوگانه در اینجا نشان داده شده است. همانطور که می‌بینیم، به جز محاسبات مقدار هدف (**گام پررنگ شده**)، بقیه مراحل دقیقا مشابه **DQN** است:

۱. پارامتر **شبکه اصلی** θ را با مقادیر تصادفی مقداردهی اولیه کنید.

۲. پارامتر **شبکه هدف** θ' را با کپی کردن پارامتر **شبکه اصلی** θ مقداردهی اولیه کنید.

۳. انباره بازپخش $\mathcal{D}$ را راه اندازی کنید

۴. برای N پردینه (اپیزود)، گام ۵ را انجام دهید

۵. برای هر گام در پردینه (اپیزود)، یعنی برای $t = 0,\ldots,T\text{-}1$:

۱. حالت s را مشاهده کنید و با استفاده از سیاست اپسیلون حریصانه، یک **عمل** را انتخاب کنید، یعنی با احتمال ε، اقدام تصادفی a را انتخاب کرده و با احتمال $\varepsilon - ۱$، عمل دیگری را بصورت زیر انتخاب کنید:

$$a = \arg\max_{a} Q_{\theta}(s, a)$$

۲. **اقدام** انتخاب شده را انجام داده و به حالت بعدی s' رفته و **پاداش** r را دریافت کنید.

۳. اطلاعات انتقال را در انباره بازپخش $\mathcal{D}$ ذخیره کنید.

۴. بطور تصادفی یک دسته کوچک شامل K انتقال از انباره بازپخش $\mathcal{D}$ نمونه‌برداری کنید.

۵. مقدار هدف را محاسبه کنید، یعنی

$$y_i = r_i + \gamma Q_{\theta'}(s_i', \arg\max_{a'} Q_{\theta'}(s_i', a'))$$

۶. تابع زیان را محاسبه کنید:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2$$

۷. نشیبهای زیان را محاسبه کرده و پارامتر **شبکه اصلی** θ را با استفاده از نزول گرادیان به‌روز کنید:

$$\theta = \theta - \alpha\nabla_\theta L(\theta)$$

۸. پارامتر **شبکه هدف** یعنی θ' را برای چندین گام زمانی، فریز کرده و سپس آن را فقط با کپی کردن پارامتر **شبکه اصلی** θ به روز کنید.

اکنون که یاد گرفتیم **DQN** دوگانه چگونه کار می‌کند، در بخش بعدی با نوع جالبی از **DQN** به نام **DQN** با بازپخش تجربه اولویتدار آشنا خواهیم شد.

DQN با بازپخش تجربه اولویتدار

ما یاد گرفتیم که در **DQN**، به طور تصادفی یک دسته کوچک شامل K انتقال را از درون انباره بازپخش، نمونه‌برداری کنیم و شبکه را آموزش دهیم. به جای انجام این کار، آیا می‌توانیم انتقالات داخل انباره بازپخش را اولویت‌بندی کرده و انتقالهایی که اولویت بالایی برای یادگیری دارند را نمونه‌برداری کنیم؟

بله! اما اولا، چرا باید اولویت را برای انتقال تعیین کنیم و چگونه می‌توانیم تصمیم بگیریم که کدام انتقال باید اولویت بیشتری نسبت به بقیه داشته باشد؟ بیایید این موضوع را با جزئیات بیشتری بررسی کنیم.

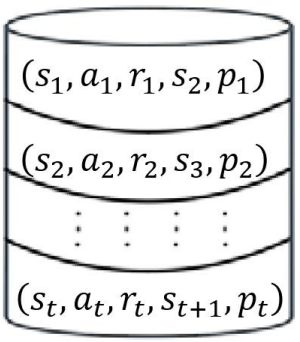
همانطور که در زیر نشان داده شده است، خطای **TD** یعنی δ، تفاوت بین مقدار هدف و مقدار پیش‌بینی شده است،

$$\delta = r + \gamma \max_{a'} Q_{\theta'}(s', a') - Q_\theta(s, a)$$

انتقالی که خطای **TD** بالایی دارد به این معنی است که انتقال صحیح نیست، و بنابراین ما باید در مورد آن انتقال بیشتر بدانیم تا خطا را به حداقل برسانیم. انتقالی که خطای **TD** پایینی دارد به این معنی است که انتقال خوبی است.

بطور مقایسه‌ای، ما همیشه می‌توانیم از اشتباهات خود نسبت به آنچه در حال حاضر در آن مهارت داریم بیشتر بیاموزیم، آیا این گزاره رو قبول دارید؟ به طور مشابه، ما می‌توانیم از انتقالهایی که خطای **TD** بالایی دارند نسبت به انتقالهایی که خطای **TD** پایین دارند، بیشتر بیاموزیم. بنابراین، می‌توانیم اولویت بالایی را به انتقالهایی بدهیم که خطای **TD** بیشتری دارند و قاعدتا، اولویت پایینی را به انتقالهایی بدهیم که خطای **TD** کمتری دارند.

ما می‌دانیم که اطلاعات انتقال شامل (s, a, r, s') است و همراه با این اطلاعات، اولویت p را نیز اضافه می‌کنیم و انتقال را به همراه اولویت آن، در انباره بازپخش خود به شکل (s, a, r, s', p) ذخیره می‌کنیم. شکل زیر، انباره بازپخش حاوی اطلاعات انتقال به همراه درجه اولویت را نشان می‌دهد:



شکل ۹.۱۱: انباره بازپخش اولویتدار

در بخش بعدی، یاد خواهیم گرفت که چگونه انتقالات خود را با استفاده از خطای **TD** و بر اساس دو نوع روش اولویت‌بندی متفاوت، اولویت‌دهی کنیم.

انواع روشهای اولویت‌بندی

ما می‌توانیم انتقالات خود را با استفاده از دو روش زیر اولویت‌بندی کنیم:

۱. اولویت‌بندی تناسبی[۱]
۲. اولویت‌بندی رتبه‌ای[۲]

[۱] Proportional Prioritization
[۲] Rank-Based Prioritization

اولویت‌بندی تناسبی یا متناسب

ما یاد گرفتیم که انتقال را می‌توان با استفاده از خطای **TD** اولویت‌بندی کرد، بنابراین اولویت p برای انتقال i، فقط معادل خطای **TD** آن خواهد بود:

$$p_i = |\delta_i|$$

توجه داشته باشید که ما مقدار مطلق خطای **TD** را به عنوان اولویت در نظر می‌گیریم تا اولویت را مثبت نگه داریم. خب، در مورد انتقالی که خطای **TD** آن صفر است چطور؟ فرض کنید ما یک انتقال i داریم و خطای **TD** آن صفر است، سپس اولویت انتقال i فقط صفر خواهد بود:

$$p_i = ۰$$

اما تخصیص عدد صفر بعنوان اولویت انتقال، مطلوب نیست و اگر اولویت انتقال را صفر تعیین کنیم، آن انتقال خاص به هیچ وجه در آموزش ما استفاده نخواهد شد. بنابراین، برای جلوگیری از این مشکل، مقدار کوچکی به نام epsilon را به خطای **TD** خود اضافه می‌کنیم. بنابراین، حتی اگر خطای **TD** صفر باشد، به دلیل این اپسیلون، همچنان اولویت کوچکی خواهد داشت. به طور دقیق‌تر، افزودن اپسیلون به خطای **TD** تضمین می‌کند که هیچ انتقالی با اولویت صفر وجود نخواهد داشت. بنابراین، اولویت را می‌توان به صورت زیر تغییر داد:

$$p_i = |\delta_i| + \varepsilon$$

به جای اینکه اولویت را به عنوان یک عدد خام داشته باشیم، می‌توانیم آن را به احتمال تبدیل کنیم تا اولویتهایی از ۰ تا ۱ داشته باشیم. همانطور که در اینجا نشان داده شده است می‌توانیم **اولویت** را به **احتمال** تبدیل کنیم:

$$P(i) = \frac{p_i}{\sum_k p_k}$$

معادله قبلی، مقدار احتمال p را برای انتقال i محاسبه می‌کند.

آیا می‌توانیم میزان اولویت‌بندی را نیز کنترل کنیم؟ یعنی به جای نمونه‌برداری از انتقال اولویتدار، آیا می‌توانیم یک

انتقال تصادفی نیز داشته باشیم؟ بله! برای این کار پارامتر جدیدی به نام α را معرفی می‌کنیم و معادله خود را به صورت زیر بازنویسی می‌کنیم. وقتی مقدار α زیاد است، مثلا ۱، فقط انتقالهایی را که اولویت بالایی دارند نمونه‌برداری می‌کنیم و وقتی مقدار α کم است، مثلا ۰، فقط یک انتقال تصادفی را نمونه‌برداری می‌کنیم:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

بنابراین، ما یاد گرفتیم که چگونه با استفاده از روش اولویت‌بندی متناسب، به یک انتقال اولویت اختصاص دهیم. در بخش بعدی، روش اولویت‌بندی دیگری به نام اولویت‌بندی مبتنی بر رتبه را یاد خواهیم گرفت.

اولویت‌بندی رتبه‌ای

اولویت‌بندی بر اساس رتبه (یا رتبه‌ای)، ساده‌ترین نوع اولویت‌بندی است. در اینجا، ما اولویت را بر اساس رتبه یک انتقال، اختصاص می‌دهیم. رتبه انتقال چقدر است؟ رتبه یک انتقال i را می‌توان به عنوان محل آن انتقال در انباره بازپخش تعریف کرد که در آن انتقالها از خطای **TD** بالا به سمت خطای **TD** پایین مرتب می‌شوند

بنابراین، ما می‌توانیم اولویت انتقال i را با استفاده از رتبه به صورت زیر تعریف کنیم:

$$p_i = \frac{۱}{Rank(i)}$$

همانطور که در بخش قبلی یاد گرفتیم، اولویت را به احتمال تبدیل می‌کنیم:

$$P(i) = \frac{p_i}{\sum_k p_k}$$

مشابه آنچه در بخش قبل آموختیم، می‌توانیم یک پارامتر α به معادله برای کنترل میزان اولویت‌بندی، که در این شرایط، معادله نهایی به صورت زیر خواهد شد:

$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

تصحیح سوگیری (سویانه)

ما یاد گرفتیم که چگونه انتقالها را با استفاده از دو روش اولویت‌بندی تناسبی و اولویت‌بندی رتبه‌ای، اولویت دهیم. اما مشکل این روشها این است که ما نسبت به نمونه‌هایی که اولویت بالایی دارند سوگیری زیادی خواهیم داشت. یعنی وقتی به نمونه‌هایی که خطای **TD** بالایی دارند اهمیت بیشتری می‌دهیم، اساسا به این معنی است که ما فقط از زیر مجموعه‌ای از نمونه‌ها یاد می‌گیریم که خطای **TD** بالایی دارند.

خب، مشکل این کار چیست؟ این کار منجر به مشکل **بیش‌برازش** می‌شود و عامل ما نسبت به آن انتقالهایی که خطای **TD** بالایی دارند بسیار **سوگیرانه** (با سویش بالا) خواهد بود. برای مبارزه با این موضوع، ما از وزنهای اهمیت w استفاده می‌کنیم. وزنهای مهم به ما کمک می‌کنند تا وزن انتقالهایی را که بارها اتفاق افتاده‌اند کاهش دهیم. وز اهمیت w انتقال i را می‌توان به صورت زیر بیان کرد:

$$w_i = (\frac{1}{N} . \frac{1}{P(i)})^\beta$$

در عبارت بالا، N طول انباره بازپخش ما را نشان داده و $P(i)$ احتمال انتقال i را نشان می‌دهد. خب، پس آن پارامتر β چیست؟ این پارامتر، وزن اهمیت را کنترل می‌کند. ما با مقادیر کوچک β شروع می‌کنیم، از ۰.۴ و آن را به سمت ۱ بازپخت می‌کنیم.

بنابراین، در این بخش، یاد گرفتیم که چگونه به انتقالها در **DQN** با بازپخش تجربه اولویتدار اهمیت بدهیم. در بخش بعدی، با یکی دیگر از انواع جالب **DQN** به نام **DQN** آشنا خواهیم شد.

DQN رقابتی (هماوردی)

قبل از ادامه، بیایید در مورد یکی از مهمترین عملکردها در یادگیری تقویتی به نام **تابع مزیت**[۱] بیاموزیم. تابع مزیت به عنوان تفاوت بین تابع Q و تابع ارزش تعریف و به صورت زیر بیان می‌شود:

$$A(s,a) = Q(s,a) - V(s)$$

بسیار خب، اما کاربرد تابع مزیت چیست؟ اصلا به چه معناست؟ اول، بیایید تابع Q و تابع ارزش را به یاد بیاورید:

- **تابع Q**: این تابع، بازده مورد انتظاری یک عامل است که از حالت s شروع کرده و با انجام **اقدام** a، سیاست π را عینا دنبال می‌کند.
- **تابع ارزش**: این تابع، بازده مورد انتظاری یک عامل است که از حالت s شروع کرده و **سیاست** π را عینا دنبال می‌کند.

قاعدتا، شما تفاوت بین تابع Q و تابع ارزش را می‌دانید: **تابع** Q، ارزشِ یک جفت حالت–عمل را به ما می‌دهد، در حالی که **تابع ارزش**، عملا ارزشِ یک حالت را صرف نظر از عمل می‌دهد. خب، حالا تفاوت بین تابع Q و تابع ارزش به ما می‌گوید که اقدام a، در مقایسه با میانگین اقداماتی که در حالت s قابل انجام هستند، تا چه حد خوب است.

بنابراین، **تابع مزیت** به ما می‌گوید که در حالت s، **اقدام** a، در مقایسه با متوسط **اقدامات** دیگر، چقدر خوب است. اکنون که فهمیدیم **تابع مزیت** چیست، بیایید ببینیم چرا و چگونه می‌توانیم از آن در **DQN** استفاده کنیم.

یادداشت مترجم:

تابع مزیت به زبان ساده به ما می‌گوید که **این اقدام خاص در این حالت چقدر از میانگین اقدامات ممکن بهتر (یا بدتر) است؟**

تعریف ریاضی

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

[۱] Advantage Function

- $Q^{\pi}(s,a)$: مقدار (**ارزش**) ناشی انجام **اقدام** a در حالت s و بعد دنبال‌کردن **سیاست** π
- $V^{\pi}(s)$: مقدار میانگین **ارزش** حالت s وقتی طبق **سیاست** π رفتار می‌کنیم (یعنی انتظار ارزش اقدامات مختلف طبق توزیع سیاست)

پس:

- اگر $A > ۰$ ← این اقدام **بهتر از میانگین** است ← باید احتمال انتخابش را **افزایش** دهیم.
- اگر $A < ۰$ ← این اقدام **بدتر از میانگین** است ← باید احتمال انتخابش را **کاهش** دهیم.
- اگر $A \approx ۰$ ← این اقدام، تقریباً در حد انتظار است ← احتمال انتخابش را تغییر زیادی نمی‌دهیم.

چرا تابع مزیت خیلی مفید است؟ (مزایای اصلی)

مثال الگوریتم‌ها	توضیح ساده	مزیت
PPO, A۲C, A۳C, SAC	نسبت به استفاده خام از G (بازگشت) واریانس خیلی کمتری دارد.	کاهش واریانس گرادیان
تقریباً همه Policy Gradient مدرن	لازم نیست بدانیم اقدام چقدر خوب است مطلقاً، فقط نسبت به بقیه چطور است.	تمرکز روی «نسبی بودن»
بدون تابع مزیت یادگیری خراب می‌شود	کمک می‌کند نوسانات یادگیری کمتر شود.	یادگیری پایدارتر
REINFORCE با baseline	در اصل تابع مزیت یک نوع خط-مبنی خیلی هوشمند است.	Baseline subtraction

مثال خیلی ساده

فرض کن در یک بازی:

- در حالت فعلی میانگین امتیاز یعنی ۱۰ ≈ V(s) است

- اگر حرکت A را انجام دهید ← انتظار امتیاز اقدام ≈ ۱۸ ← **مزیت** برابر ۸ + ← خیلی خوب است!
- اگر حرکت B را انجام دهید ← انتظار امتیاز این اقدام ≈ ۴ ← **مزیت** برابر ۶ - ← خیلی بد است!

$$Q(s, a_A) = ۱۸ \rightarrow Advantage_A = ۱۸ - ۱۰ = +۸ \rightarrow Good\ Avtion$$

$$Q(s, a_B) = ۴ \rightarrow Advantage_B = ۴ - ۱۰ = -۶ \rightarrow Bad\ Action$$

پس الگوریتم باید احتمال حرکت A را خیلی بیشتر کرده و احتمال اقدام B را کم کند.

خلاصه به زبان روزمره

تابع مزیت یعنی:

«این کار تو نسبت به کار معمولی که معمولاً می‌کنی چقدر بهتر یا بدتر بود؟»

اگر بهتر بود ← بیشتر انجامش بده، ولی اگر بدتر بود ← کمتر انجامش بده

این ایده تقریباً در تمام الگوریتم‌های مدرن یادگیری تقویتی مبتنی بر گرادیان سیاست مثل PPO ، SAC، A۲C، IMPALA و ... قلب اصلی یادگیری است.

پایان یادداشت

درک DQN رقابتی (هماوردی)

ما آموخته‌ایم که در یک **DQN**، حالت مسئله را به عنوان ورودی، تغذیه می‌کنیم و شبکهٔ ما، مقدار Q را برای همه اقدامات در آن حالت محاسبه می‌کند. به جای محاسبه **مقادیر** Q به این روش، آیا می‌توانیم مقادیر Q را با استفاده از **تابع مزیت** محاسبه کنیم؟ ما آموختیم که **تابع مزیت** به صورت زیر داده می‌شود:

$$A(s,a) = Q(s,a) - V(s)$$

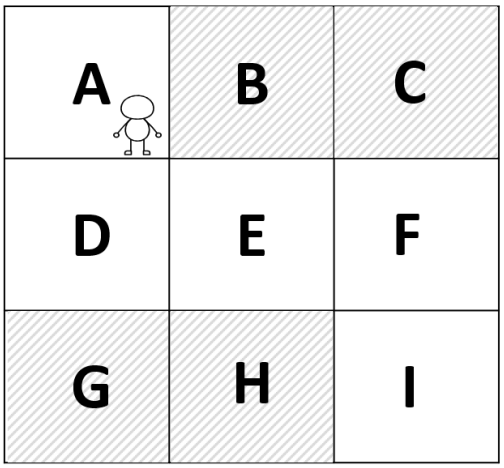
ما می‌توانیم معادله بالا را بصورت زیر بازنویسی کنیم:

$$Q(s,a) = V(s) + A(s,a)$$

همانطور که از معادله قبلی می‌بینیم، می‌توانیم **مقدار** Q را فقط با جمع کردن **تابع ارزش** و **تابع مزیت** با هم محاسبه کنیم. صبر کنید! چرا باید این کار را انجام دهیم؟ محاسبه مستقیم مقدار Q چه اشکالی دارد؟

بیایید فرض کنیم در حالتی همچون S هستیم و ۲۰ **عمل** ممکن برای انجام در این حالت داریم. محاسبه مقادیر Q همه این ۲۰ **عمل** در حالت S مفید نخواهد بود زیرا اکثر **اقدامات** هیچ تاثیری بر حالت نخواهند داشت و همچنین اکثر **اقدامات**، مقدار Q مشابهی خواهند داشت. منظور ما از این موضوع چیست؟ بیایید این موضوع را با محیط جهان مشبکی که در شکل ۹.۱۲ نشان داده شده است درک کنیم.

همانطور که می‌بینیم ، عامل در حالت **A** است. در این حالت، محاسبه **مقدار** Q **عمل** در حالت **A** چه فایده‌ای دارد؟ حرکت *به سمت بالا* در حالت **A** هیچ تاثیری نخواهد داشت و عامل را به جایی نمی‌رساند. به طور مشابه، به محیطی فکر کنید که در آن فضای عمل ما بزرگ است، مثلا ۱۰۰. در این حالت، بیشتر **اقدامات** در حالت داده شده هیچ تاثیری نخواهند داشت. همچنین، زمانی که **فضای عمل** بزرگ است، اکثر **اقدامات**، **مقدار** Q مشابهی خواهند داشت.



شکل ۹.۱۲: محیط جهانی مشبک

حال، بیایید در مورد **ارزش** یک حالت صحبت کنیم. توجه داشته باشید که همه حالتها برای یک عامل مهم نیستند. ممکن است حالتی وجود داشته باشد که همیشه پاداش بدی می‌دهد، مهم نیست که چه عملی انجام می‌دهیم. در این صورت، محاسبه مقدار Q همه اقدامات ممکن در این حالت (وقتی می‌دانیم که این حالت همیشه پاداش بدی به ما می‌دهد) کار مفیدی نیست.

بنابراین، برای حل این مشکل می‌توانیم **تابع** Q را به عنوان مجموع **تابع ارزش** و تابع مزیت محاسبه کنیم. یعنی از طریق **تابع ارزش**، می‌توانیم (بدون محاسبه مقادیر همه **اقدامات** در یک حالت)، بفهمیم که آیا این حالت ارزشمند است یا خیر. و از روی تابع مزیت، می‌توانیم بفهمیم که آیا یک عمل واقعا خوب است یا فقط همان ارزش سایر اقدامات را به ما می‌دهد.

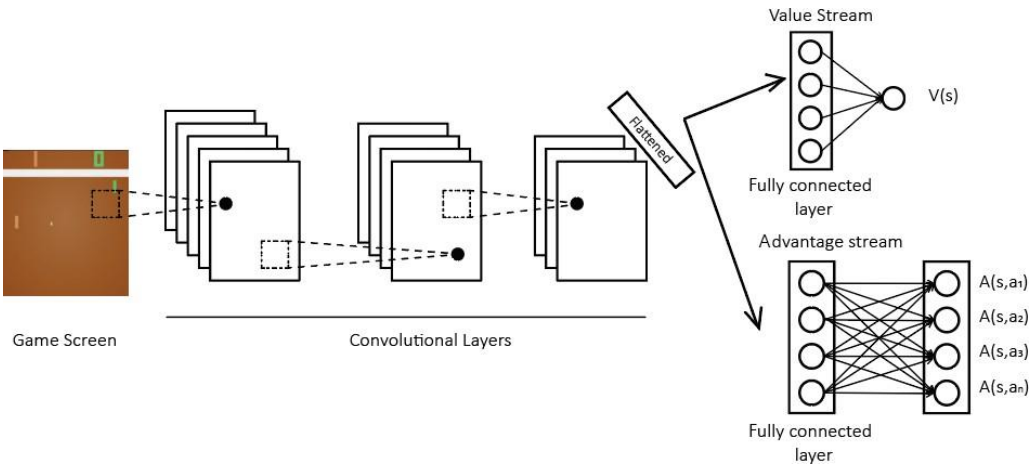
اکنون که یک ایده اولیه از **DQN** رقابتی (یا هماوردی) داریم، بیایید معماری **DQN** رقابتی را در بخش بعدی بررسی کنیم.

معماری یك DQN رقابتی

ما آموختیم که در یک **DQN** رقابتی، مقادیر Q را می‌توان به صورت زیر محاسبه کرد:

$$Q(s,a) = V(s) + A(s,a)$$

چگونه می‌توانیم شبکه عصبی خود را طوری طراحی کنیم که مقادیر Q را به این روش منتشر کند؟ ما می‌توانیم لایه نهایی شبکه خود را به دو **جریان**[۱] تقسیم کنیم. **جریان اول**، **تابع ارزش** و **جریان دوم**، تابع مزیت را محاسبه کند. با توجه به هر حالت به عنوان ورودی، **جریان ارزش**، عملا **ارزش** آن حالت را می‌دهد، در حالی که **جریان مزیت**، در واقع مزیت تمام اقدامات ممکن در آن حالت را می‌دهد. به عنوان مثال، همانطور که شکل ۹.۱۳ نشان می‌دهد، ما وضعیت بازی (صفحه بازی) را به عنوان ورودی به شبکه تغذیه می‌کنیم. **جریان ارزش**، در اینجا، ارزش آن حالت را محاسبه می‌کند در حالی که **جریان مزیت**، در واقع، مقادیر مزیت همه اقدامات در آن حالت را محاسبه می‌کند:



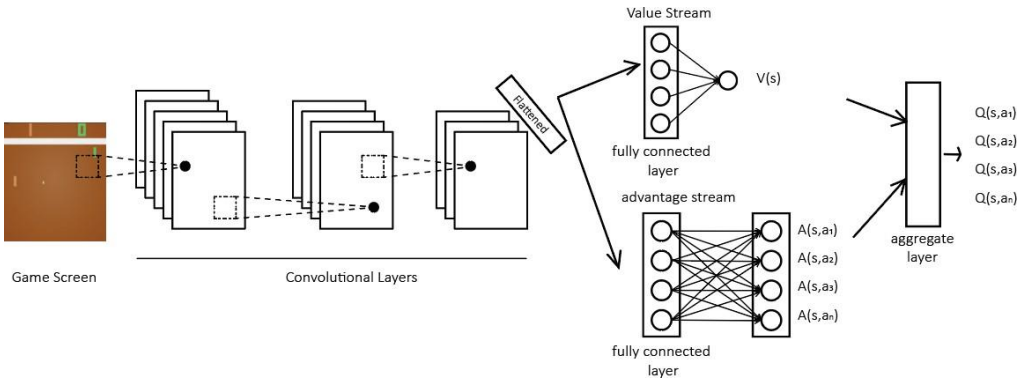
شکل ۹.۱۳: معماری یک **DQN** رقابتی

ما یاد گرفتیم که ارزش Q را از طریق جمع کردن **ارزش حالت** و **ارزش مزیت** با هم، محاسبه می‌کنیم، بنابراین **جریان ارزش** و جریان مزیت را با استفاده از لایه دیگری (به نام **لایه تجمیع**[۲]) ترکیب می‌کنیم و **مقدار** Q را همانگونه که شکل ۹.۱۴ نشان می‌دهد، محاسبه می‌کنیم

با این حساب، **جریان ارزش** عملا ارزش حالت را محاسبه می‌کند، **جریان مزیت**، در واقع ارزش مزیت را محاسبه

[۱] Stream
[۲] Aggregate Layer

می‌کند و **لایه تجمیع**، این جریانها را با هم جمع می‌کند تا مقدار Q را محاسبه کند:



شکل ۹.۱۴: معماری یک **DQN** رقابتی شامل یک لایه تجمیعی

اما در اینجا یک مشکل کوچک وجود دارد. فقط جمع کردن مقدار حالت و مقدار مزیت در لایه تجمیع و محاسبه مقدار Q ما را به سمت مشکل **قابلیت شناسایی**[۱] سوق می‌دهد.

بنابراین، برای مقابله با این مشکل، **تابع مزیت** را برای عمل انتخاب شده، صفر می‌کنیم. ما می‌توانیم با کم کردن مقدار مزیت متوسط، یعنی مزیت متوسط همه اقدامات در **فضای کنش**، همانطور که در اینجا نشان داده شده است، به این هدف دست یابیم:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{\mathcal{A}}\sum_{a'} A(s,a'))$$

جایی که $\mathcal{A}$ طول فضای کنش یا عمل[۲] را نشان می‌دهد.

بنابراین، می‌توانیم معادله نهایی خود را برای محاسبه مقدار Q با پارامترها به صورت زیر بنویسیم:

[۱] Identifiability
[۲] Length of the Action Space

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{\mathcal{A}}\sum_{a'} A(s,a';\theta,\alpha))$$

در معادله بالا، θ پارامتر شبکه همتابی است، β پارامتر **جریان ارزش** است و α پارامتر جریان مزیت است. پس از محاسبه مقدار Q، می‌توانیم عمل را انتخاب کنیم:

$$a = arg\max_{a} Q(s,a;\theta,\alpha,\beta)$$

بنابراین، تنها تفاوت بین **DQN** رقابتی و **DQN** این است که در یک **DQN** رقابتی، به جای محاسبه مستقیم مقادیر Q، آنها را با ترکیب مقدار حالت و مقدار مزیت محاسبه می‌کنیم.

در بخش بعدی، نوع دیگری از **DQN** به نام «شبکه Q بازگشتی عمیق» را بررسی خواهیم کرد.

شبکه Q بازگشتی عمیق

شبکه Q بازگشتی عمیق (**DRQN**) دقیقا مشابه **DQN** است اما با لایه‌های بازگشتی. اما لایه‌های بازگشتی در **DQN** چه فایده ای دارد؟ برای پاسخ به این سوال، ابتدا بیایید مشکلی به نام **فرآیند تصمیم‌گیری رویت‌پذیر جزئی مارکوف**[۱] (**POMDP**) را درک کنیم.

یک محیط موقعی **POMDP** نامیده می‌شود که مجموعه محدودی از اطلاعات در مورد محیط در دسترس باشد. تاکنون، در فصلهای قبلی، ما یک **MDP** کاملا قابل مشاهده را دیده‌ایم که در آن همه اقدامات و حالتهای ممکن را می‌دانیم؛ اگرچه ممکن است از احتمالات انتقال و پاداش آگاه نباشیم، اما دانش کاملی از محیط داشتیم. به عنوان مثال، در محیط دریاچه یخ زده، ما از تمام حالات و اقدامات محیط آگاهی کاملی داشتیم.

اما بیشتر محیطهای دنیای واقعی فقط تا حدی قابل مشاهده هستند. ما نمی‌توانیم همه حالات را ببینیم. به عنوان

[۱] Partially Observable Markov Decision Process (POMDP)

مثال، یک عامل را در نظر بگیرید که یاد می‌گیرد در یک محیط واقعی راه برود. در این حالت، عامل دانش کاملی از محیط (دنیای واقعی) نخواهد داشت. هیچ اطلاعاتی خارج از دید خود نخواهد داشت.

بنابراین، در **POMDP**، حالتها فقط اطلاعات جزئی ارائه می‌دهند، اما نگه داشتن اطلاعات مربوط به حالتهای گذشته در حافظه به عامل کمک می‌کند تا ماهیت محیط را بیشتر درک کرده و سیاست بهینه را پیدا کند. بنابراین، در **POMDP**، ما باید اطلاعات مربوط به حالتهای گذشته را حفظ کنیم تا بتوانیم اقدام بهینه را انجام دهیم

بنابراین، آیا می‌توانیم از «شبکه‌های عصبی بازگشتی» برای درک و حفظ اطلاعات مربوط به حالتهای گذشته استفاده کنیم تا در موقع نیاز، آن اطلاعات را در دسترس ما قرار دهد؟ بله، **شبکه عصبی بازگشتی با حافظه کوتاه-مدت دیرپا** (**LSTM RNN**) برای حفظ کردن، فراموش کردن و به‌روزرسانی اطلاعات در صورت لزوم، بسیار مفید است. بنابراین، می‌توانیم از لایه **LSTM** در **DQN** برای حفظ اطلاعات مربوط به حالتهای گذشته تا وقت مورد نیازش، استفاده کنیم. حفظ اطلاعات در مورد حالات گذشته، هنگامی که مشکل **POMDP** داریم، به ما کمک می‌کند.

اکنون که درک اولیه‌ای از علت نیاز به **DRQN** و چگونگی حل مشکل **POMDP** توسط آن پیدا کردیم، به معماری **DRQN** می‌پردازیم

معماری DRQN

شکل ۹.۱۵ معماری **DRQN** را نشان می‌دهد. همانطور که می‌بینیم معماری این شبکه، شبیه معماری **DQN** است با این تفاوت که دارای یک لایه **LSTM** است:

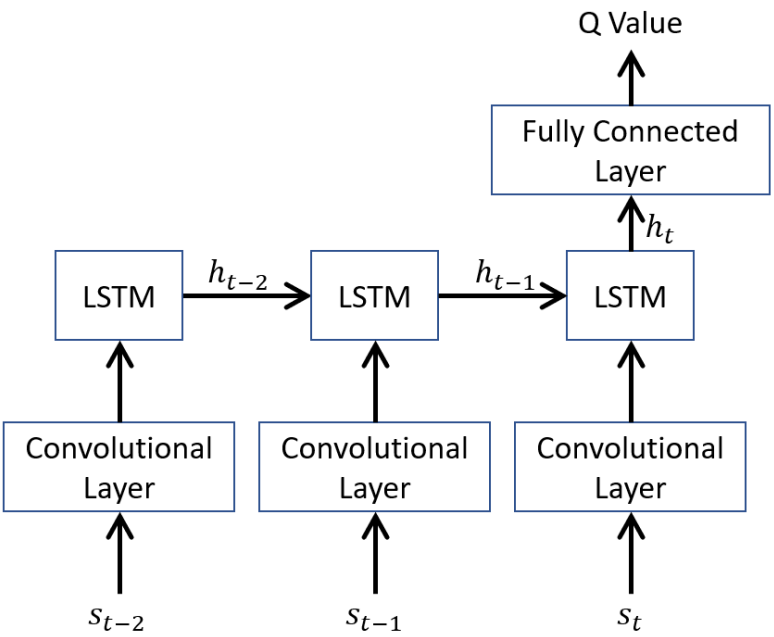
شکل ۹.۱۵: معماری **DRQN**

ما صفحه بازی را به عنوان ورودی، از لایه همتابی عبور می‌دهیم. لایه همتابی، تصویر را به هم می‌پیچد (همتافته می‌کند) و یک نقشه ویژگی تولید می‌کند. سپس نقشه ویژگی به دست آمده به لایه **LSTM** منتقل می‌شود. لایه

LSTM دارای حافظه‌ای برای نگهداری اطلاعات است. بنابراین، اطلاعات مربوط به حالتهای مهم قبلی بازی را حفظ کرده و حافظه خود را در طول مراحل زمانی در صورت لزوم به‌روز می‌کند. سپس، حالت پنهان را از لایه **LSTM** به لایه کاملا متصل تغذیه می‌کنیم تا مقدار Q را خروجی دهد.

شکل ۹.۱۶ به ما کمک می‌کند تا بفهمیم **DRQN** دقیقا چگونه کار می‌کند. گیریم که باید مقدار Q را برای حالت s_t و اقدام a_t محاسبه کنیم. برخلاف **DQN**، ما مقدار Q را به صورت $Q(s_t, a_t)$ به طور مستقیم محاسبه نمی‌کنیم. بلکه همانطور که می‌بینیم، همراه با حالت فعلی s_t، از حالت پنهان h_t نیز برای محاسبه مقدار Q استفاده می‌کنیم. دلیل استفاده از حالت پنهان این است که اطلاعات مربوط به حالتهای بازی گذشته را در حافظه نگه می‌دارد.

از آنجایی که ما از سلولهای **LSTM** استفاده می‌کنیم، حالت پنهان h_t شامل اطلاعاتی در مورد حالتهای بازی گذشته در حافظه، تا لحظه مورد نیاز، خواهد بود:



شکل ۹.۱۶: معماری **DRQN**

به جز این تغییر، **DRQN** درست مانند **DQN** کار می‌کند. صبر کنید! در مورد انباره بازپخش چطور؟ در **DQN**، ما یاد گرفتیم که اطلاعات انتقال را در انباره بازپخش، ذخیره کرده و شبکه خود را با نمونه‌برداری از یک دسته کوچک

تجربه، آموزش می‌دهیم. ما همچنین یاد گرفتیم که اطلاعات انتقال به صورت متوالی یکی پس از دیگری در انباره پخش قرار می‌گیرند، بنابراین برای جلوگیری از تجربه همبستگی، به طور تصادفی یک دسته کوچک از تجربه را از انباره پخش نمونه برداری کرده و شبکه را آموزش می‌دهیم.

اما در مورد **DRQN**، ما به اطلاعات متوالی نیاز داریم تا شبکهٔ ما بتواند اطلاعات حالتهای بازی گذشته را حفظ کند. درست است که ما به اطلاعات متوالی نیاز داریم، ولیکن در عین حال، نمی‌خواهیم به هنگام آموزش شبکه با تجربیات همبسته، دچار وضعیت بیش‌برازش شویم. چگونه می‌توانیم به این هدف برسیم؟

برای دستیابی به این هدف، در یک **DRQN** (به جای نمونه‌برداری تصادفی یک دسته کوچک از **انتقالها**) ما به طور تصادفی یک دسته کوچک از **اپیزودها** را نمونه‌برداری می‌کنیم. یعنی می‌دانیم که در هر اپیزود، ما اطلاعات انتقالی را داریم که متوالیا دنبال شده است. بنابراین ما یک دسته کوچک تصادفی از اپیزودها را می‌گیریم و در هر **اپیزود**، اطلاعات انتقالی را خواهیم داشت که به ترتیب انجام شده است. لذا، به این ترتیب، ما می‌توانیم در مدل خود، هم **تصادفی بودن** و هم **اطلاعات انتقالی** را داشته باشیم که پی در پی انجام می‌گیرد. به این فرایند، **به‌روزرسانی‌های متوالی خودراه‌انداز (ابتناگر)**[۱] می‌گویند

پس از نمونه برداری دسته کوچکی از اپیزودها به صورت تصادفی، می‌توانیم **DRQN** را درست مانند شبکه **DQN** با به حداقل رساندن زیان **MSE** آموزش دهیم. برای کسب اطلاعات بیشتر، می‌توانید به مقاله **DRQN** ارائه شده در بخش مطالعه بیشتر مراجعه کنید

خلاصه

ما این فصل را با یادگیری اینکه شبکه‌های Q عمیق چیست و چگونه از آنها برای تقریب **مقدار** Q استفاده می‌شود، شروع کردیم. ما یاد گرفتیم که در یک **DQN**، از انبارکی به نام انباره بازپخش برای ذخیره تجربه عامل استفاده می‌کنیم. سپس، به طور تصادفی یک دسته کوچک از تجربه را از انباره بازپخش، نمونه‌برداری کرده و با به حداقل رساندن **MSE**، شبکه را آموزش می‌دهیم. در ادامه، ما الگوریتم **DQN** را با جزئیات بیشتری بررسی کردیم و سپس

[۱] Bootstrapped Sequential Updates

یاد گرفتیم که چگونه **DQN** را برای اجرای بازیهای آتاری پیاده سازی کنیم.

پس از آن، متوجه شدیم که **DQN**، مقدار **هدف** را به دلیل داشتن عملگر بیشینه، بصورت بیش‌برآورد، تخمین می‌زند. بنابراین، ما از **DQN** دوگانه استفاده کردیم، که در آن دو **تابع** Q در محاسبه مقدار هدف خود داریم. یک تابع Q پارامتر شده توسط پارامتر **شبکه اصلی** θ برای انتخاب عمل و تابع Q دیگر پارامتری شده توسط پارامتر **شبکه هدف** θ' برای محاسبه مقدار Q استفاده می‌شود.

در ادامه، ما در مورد **DQN** با پخش مجدد تجربه اولویت‌بندی شده یاد گرفتیم، جایی که انتقالات بر اساس خطای **TD** اولویت‌بندی می‌شوند. ما دو نوع مختلف از روشهای اولویت‌بندی به نامهای اولویت‌بندی متناسب و اولویت‌بندی مبتنی بر رتبه را بررسی کردیم.

در مرحله بعد، با نوع جالب دیگری از **DQN** به نام **DQN** رقابتی (هماوردی) آشنا شدیم. در **DQN** رقابتی به جای محاسبه مستقیم مقادیر Q، آنها را با استفاده از دو جریان به نام جریان **ارزش** و جریان مزیت محاسبه می‌کنیم.

در پایان فصل، ما در مورد **شبکه** Q **بازگشتی عمیق (DRQN)** و نحوه حل مشکل **فرآیندهای تصمیم‌گیری رویت‌پذیر جزئی مارکوف** آشنا شدیم.

در فصل بعدی، با الگوریتم محبوب دیگری به نام **گرادیان سیاست** آشنا خواهیم شد.

سوالات

بیایید درک خود را از **DQN** و انواع آن با پاسخ دادن به سوالات زیر ارزیابی کنیم:

۱. چرا به **DQN** نیاز داریم؟

۲. انباره بازپخش (پخش مجدد) چیست؟

۳. چرا به شبکه هدف نیاز داریم؟

٤. **DQN** دوگانه چه تفاوتی با **DQN** دارد؟

٥. چرا باید انتقالها را اولویت‌بندی کنیم؟

٦. تابع مزیت چیست؟

۷. چرا به لایه‌های حافظه کوتاه–مدت دیرپا (**LSTM**) در شبکه Q بازگشتی عمیق (**DRQN**) نیاز داریم؟

بیشتر بخوانید

برای کسب اطلاعات بیشتر می‌توان به مقالات زیر مراجعه کرد:

- **Playing Atari with Deep Reinforcement Learning** by *Volodymyr Mnih, et al.*, https://arxiv.org/pdf/1312.5602.pdf
- **Deep Reinforcement Learning with Double Q-learning** by *Hado van Hasselt, Arthur Guez, David Silver*, https://arxiv.org/pdf/1509.06461.pdf
- **Prioritized Experience Replay** by *Tom Schaul, John Quan, Ioannis Antonoglou and David Silver*, https://arxiv.org/pdf/1511.05952.pdf
- **Dueling Network Architectures for Deep Reinforcement Learning** by *Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas*, https://arxiv.org/pdf/1511.06581.pdf
- **Deep Recurrent Q-Learning for Partially Observable MDPs** by *Matthew Hausknecht and Peter Stone*, https://arxiv.org/pdf/1507.06527.pdf

فصل دهم

روش گرادیان سیاست

در فصل‌های قبل، یاد گرفتیم که چگونه از الگوریتم‌های یادگیری تقویتی «**مبتنی بر ارزش**» برای محاسبه خطمشی بهینه استفاده کنیم. به این معنا که با روش‌های مبتنی بر ارزش (یا **ارزش-محور**[۱]، تابع Q بهینه را به صورت تکراری محاسبه کرده و از روی تابع Q بهینه، سیاست بهینه را استخراج می‌کنیم. در این فصل، با روش‌های «**مبتنی بر سیاست**» (یا **سیاست-محور**[۲]) آشنا می‌شویم، که سیاست بهینه را بدون نیاز به محاسبه تابع Q بهینه، محاسبه می‌کند.

ما این فصل را با بررسی معایب تشخیص **سیاست** از طریق روشِ محاسبه **تابع** Q شروع کرده و سپس یاد می‌گیریم که چگونه روش‌های مبتنی بر سیاست، خطمشی ما را مستقیما و بدون محاسبه **تابع** Q یاد می‌گیرند. در مرحله بعد، یکی از محبوبترین روش‌های مبتنی بر سیاست به نام «**گرادیان سیاست**» را بررسی خواهیم کرد. ابتدا مروری گسترده بر الگوریتم گرادیان خطمشی (یا نشیب سیاست) داشته و سپس با جزئیات بیشتر در مورد آن بحث خواهیم کرد.

در ادامه، ما همچنین یاد خواهیم گرفت که چگونه **نشیب (گرادیان) سیاست** را گام به گام استخراج کنیم و الگوریتم روش گرادیان سیاست را با جزئیات بیشتری بررسی می‌نماییم. در پایان فصل، با تکنیک‌های کاهش واریانس در روش نشیب سیاست آشنا خواهیم شد.

در این فصل با مباحث زیر آشنا می‌شویم:

- **چرا روش‌های سیاست-محور؟**
- **معرفی اجمالی روش نشیب سیاست**[۳]
- **استخراج نشیب سیاست**
- **الگوریتم نشیب سیاست**

[۱] Value-Based Algorithms
[۲] Value-Based Methods
[۳] Policy Gradient Intuition

- **نشیب سیاست با پاداش باقیمانده (پاداش زین پس)**[۱]
- **نشیب سیاست با خط مبنا (یا خط پایه)**[۲]
- **الگوریتم نشیب سیاست با خط-مبنا**[۳]

چرا روش‌های سیاست_محور؟

هدف از یادگیری تقویتی، یافتن سیاست بهینه است؛ یعنی آن سیاستی که حداکثر بازده را فراهم می‌کند. تاکنون چندین الگوریتم مختلف برای محاسبه سیاست بهینه آموخته‌ایم و همه این الگوریتم‌ها، روش‌های مبتنی بر ارزش بوده‌اند. صبر کنید! روش‌های «**مبتنی بر ارزش یا ارزش-محور**» یعنی چه؟ بیایید بطور خلاصه بگوییم که روش‌های مبتنی بر ارزش کدامند و مشکلات مرتبط با آنها چیست و سپس با روش‌های «**مبتنی بر سیاست یا سیاست-محور**» آشنا خواهیم شد. جمع بندی همیشه خوب است، اینطور نیست؟

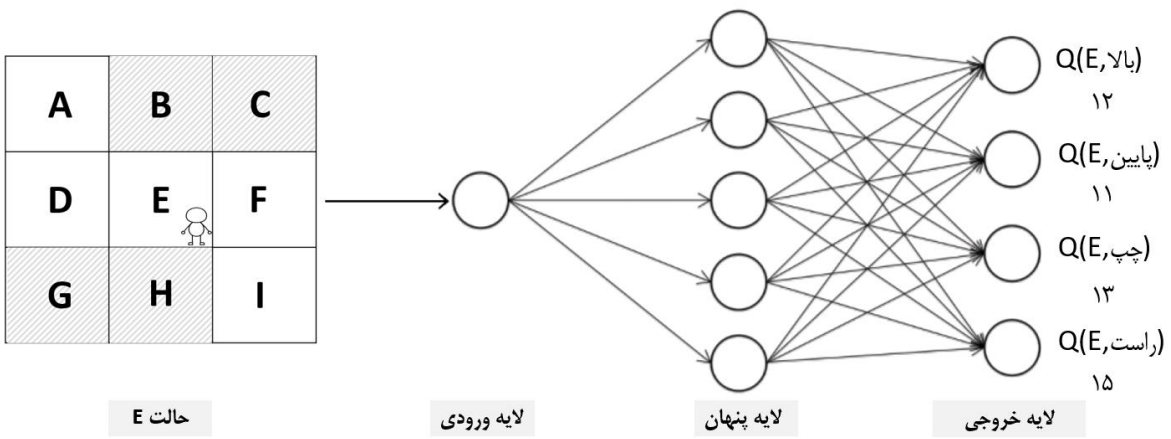
با روش‌های مبتنی بر **ارزش**، ما **سیاست** بهینه را از تابع Qی بهینه (مقادیر Q) استخراج می‌کنیم، به این معنی که ابتدا مقادیر Q همه جفت‌های حالت-عمل را برای یافتن سیاست محاسبه کرده، و سپس ما خط‌مشی را با انتخاب اقدام خوب در هر حالت، استخراج می‌کنیم (منظور از اقدام مناسب در هر حالت، آن اقدامی از میان همه اقدامات موجود در آن حالت است که دارای حداکثر مقدار Q است). به عنوان مثال، فرض کنید ما دو حالت s_0 و s_1 داریم و فضای کنش ما دو اقدام دارد؛ مثلا گیریم اقدامات ما ۰ و ۱ باشند. ابتدا مقدار Q تمام جفت‌های state-action را، همانند جدول زیر، محاسبه می‌کنیم. اکنون، ما سیاست را از تابع Q (مقادیر Q) با انتخاب عمل ۰ در حالت s_0 و عمل ۱ در حالت s_1 استخراج می‌کنیم، زیرا آنها حداکثر مقدار Q را دارند:

[۱] Reward-To-Go
[۲] Baseline
[۳] Policy Gradient with Baseline

حالت	اقدام	ارزش
$s_۰$	۰	۱۰
$s_۰$	۱	۵
$s_۱$	۰	۱۰
$s_۱$	۱	۱۱

جدول ۱۰.۱: جدول Q

بعدا متوجه شدیم که محاسبه تابع Q، زمانی که محیط ما دارای تعداد زیادی حالت و عمل است، کاری بس دشوار می‌باشد زیرا محاسبه مقادیر Q همه جفت‌های حالت–عمل ممکن، راحت نخواهد بود. بنابراین، ما به شبکه Q عمیق (یا **DQN**) متوسل شدیم. در **DQN**، ما از یک شبکه عصبی برای **تقریب تابع** Q (مقدار Q) استفاده کردیم. با توجه به یک حالت، شبکه مقادیر Q تمام اقدامات ممکن در آن حالت را برمی‌گرداند. به عنوان مثال، محیط دنیای مشبک را در نظر بگیرید. با توجه به یک حالت، **DQN** ما، **مقادیر** Q تمام اقدامات ممکن در آن حالت را برمی‌گرداند. سپس اقدامی را انتخاب می‌کنیم که **بالاترین مقدار** Q را دارد. همانطور که در شکل ۱۰.۱ می‌بینیم، با توجه به حالت **E**، روش **DQN** مقدار Q تمام اقدامات ممکن (*بالا، پایین، چپ، راست*) را برمی‌گرداند. سپس عمل *راست* را در حالت **E** انتخاب می‌کنیم زیرا حداکثر مقدار Q را دارد:



شکل ۱۰.۱: **DQN**

بنابراین، در **روش‌های مبتنی بر ارزش**، تابع Q را به صورت تکراری بهبود می‌بخشیم و هنگامی‌که تابع Q بهینه را به دست آوردیم، با انتخاب یک عمل در هر حالت که دارای حداکثر مقدار Q است، سیاست بهینه را استخراج می‌کنیم.

یکی از معایب **روش مبتنی بر ارزش** این است که فقط برای محیط‌های گسسته (محیط‌هایی با فضای کنش گسسته) مناسب است و نمی‌توانیم روش‌های مبتنی بر ارزش را در محیط‌های پیوسته (محیط‌هایی با فضای کنش پیوسته) به کار ببریم.

ما آموخته ایم که یک فضای کنش گسسته، دارای مجموعه‌ای گسسته از اقدامات است. به عنوان مثال، محیط دنیای مشبک دارای کنش‌های گسسته (*بالا، پایین، چپ و راست*) است و فضای کنش پیوسته، شامل اقداماتی است که مقادیر پیوسته هستند، به عنوان مثال، کنترل سرعت ماشین.

تا کنون، ما فقط با یک محیط گسسته سروکار داشته‌ایم که یک فضای کنش گسسته دارد، بنابراین به راحتی مقدار Q همه جفت‌های حالت–عمل ممکن را محاسبه کردیم. اما وقتی فضای کنش ما پیوسته است، چگونه می‌توانیم مقدار Q همه جفت‌های حالت–عمل ممکن را محاسبه کنیم؟ فرض کنید ما در حال آموزش یک عامل برای رانندگی با ماشین هستیم که یک عمل مداوم و مستمر در فضای کنش خود داریم. مثلا فرض کنید اقدام ما، سرعت خودرو باشد و مقدار سرعت خودرو از ۰ تا ۱۵۰ کیلومتر در ساعت باشد. در این مورد، چگونه می‌توانیم مقدار Q همه جفت‌های حالت–عمل ممکن را با عمل یک مقدار پیوسته محاسبه کنیم؟

در این حالت می‌توانیم اقدامات پیوسته را به سرعت (۰ تا ۱۰) به عنوان عمل ۱، سرعت (۱۰ تا ۲۰) به عنوان عمل ۲ و غیره تقسیم کنیم. پس از گسسته‌سازی، می‌توانیم مقدار Q تمام جفت‌های حالت–عمل ممکن را محاسبه کنیم. البته، گسسته‌سازی همیشه مطلوب نیست. ممکن است چندین ویژگی مهم را از دست بدهیم و یا در یک فضای کنش با مجموعه عظیمی از اقدامات قرار بگیریم.

اکثر مشکلات دنیای واقعی دارای فضای کنش مداوم هستند، مثلا یک ماشین خودران، یا رباتی که راه رفتن و موارد دیگر را یاد می‌گیرد. آنها علاوه بر داشتن فضای کنش مستمر، ابعاد بالایی نیز دارند. بنابراین، **DQN** و سایر روش‌های مبتنی بر ارزش نمی‌توانند به طور موثر با فضای کنش پیوسته مقابله کنند.

لذا، ما از روش‌های «**مبتنی بر سیاست**» استفاده می‌کنیم. با روش‌های مبتنی بر سیاست، برای یافتن سیاست بهینه نیازی به **محاسبه تابع** Q (مقادیر Q) نداریم. در عوض، می‌توانیم آنها را مستقیما محاسبه کنیم. یعنی برای استخراج سیاست، نیازی به **تابع** Q نداریم. «**روش‌های مبتنی بر سیاست**» مزایای متعددی نسبت به «**روش‌های مبتنی بر ارزش**» دارند و می‌توانند هم فضاهای کنش گسسته و هم فضاهای کنش پیوسته را مدیریت کنند.

ما یاد گرفتیم که **DQN** با استفاده از سیاست حریصانه اپسیلون با معضل اکتشاف–انتفاع برخورد می‌کند. با سیاست اپسیلون–حریصانه، یا بهترین عمل را با احتمال [۱ منهای اپسیلون] یا یک عمل تصادفی با احتمال [اپسیلون] انتخاب می‌کنیم. اکثر روش‌های «**مبتنی بر سیاست**» از یک «**سیاست احتمالی**[۱]» استفاده می‌کنند. ما می‌دانیم که با یک سیاست احتمالی، اقدامات را بر اساس توزیع احتمال آنها بر روی فضای کنش انتخاب می‌کنیم، که به عامل اجازه می‌دهد تا به جای انجام یک عمل مشابه و یکسان در هر بار، اقدامات مختلف را کشف کند. بنابراین، روش‌های مبتنی بر سیاست، با استفاده از یک سیاست احتمالی، به طور ضمنی، مراقب مخصمه مربوط به مبادله اکتشاف–انتفاع هستند. با این حال، چندین روش مبتنی بر سیاست وجود دارد که از یک **سیاست قطعی**[۲] نیز استفاده می‌کنند، که در فصول آینده با آنها بیشتر آشنا خواهیم شد.

بسیار خب، روش‌های **مبتنی بر سیاست** دقیقا چگونه کار می‌کنند؟ چگونه آنها یک سیاست بهینه را بدون محاسبه تابع Q پیدا می‌کنند؟ در بخش بعدی با این موضوع آشنا خواهیم شد. اکنون که درک اولیه‌ای از چیستی رویکرد **سیاست-مبنا** و همچنین معایب رویکرد **ارزش-مبنا** پیدا کردیم، اجازه دهید، در بخش بعدی با یکی از روشهای بنیادین مبتنی بر سیاست به نام «**گرادیان سیاست**» آشنا شویم

[۱] Stochastic Policy
این روش، در صورت وجود یک تابع توزیع احتمال مشخص، به سراغ مقادیر تصادفی (بختکی) نمی‌رود. در غیر این صورت، ممکن است بطور تصادفی (بختکی و دلبخواه) شروع کند تا تابع توزیع مربوطه را بیابد.

[۲] Deterministic Policy

معرفی اجمالی روش گرادیان سیاست

گرادیان سیاست یکی از محبوبترین الگوریتم‌ها در یادگیری تقویتی عمیق است. همانطور که آموختیم، گرادیان یا **نشیب سیاست**، یک روش مبتنی بر **سیاست** است که با استفاده از آن می‌توانیم سیاست یا خط‌مشی بهینه را بدون محاسبه **تابع** Q پیدا کنیم. این روش، سیاست بهینه را با «پارامتری کردن مستقیم **سیاست**» با استفاده از پارامتری همچون θ پیدا می‌کند.

روش **گرادیان سیاست** از یک **سیاست** احتمالی شروع می‌کند. ما آموخته‌ایم که با یک **سیاست** احتمالی، یک عمل را بر اساس توزیع احتمال در فضای کنش انتخاب می‌کنیم. فرض کنید ما یک **سیاست** احتمالی π داریم، که این سیاست احتمالی، مقدار احتمال برای انجام اقدام a را در حالت s به ما می‌دهد، و می‌توان آنرا با $\pi_\theta(a|s)$ نشان داد. در روش گرادیان سیاست، ما از یک سیاست پارامتری استفاده می‌کنیم، لذا می‌توانیم **خط‌مشی** خود را به صورت $\pi_\theta(a|s)$ نشان دهیم، جایی که θ بیان می‌کند که **سیاست** ما پارامتری شده است.

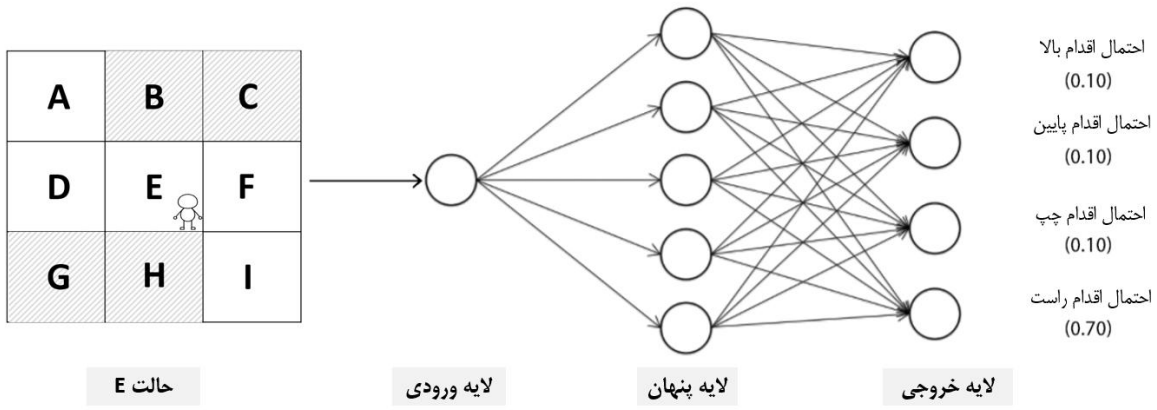
صبر کنید! منظور ما از بیان یک **سیاست** پارامتری چیست؟ دقیقا چه چیزی است؟ به خاطر دارید که با **DQN**، ما یاد گرفتیم که **تابع** Q خود را پارامتری کرده تا بر اساس آن بتوانیم **ارزش** Q را محاسبه کنیم. خب ما در اینجا می‌توانیم همین کار را انجام دهیم، با این تفاوت که به جای پارامتری کردن **تابع** Q، مستقیما **سیاست** را برای محاسبه سیاست بهینه، پارامتری می‌کنیم. یعنی می‌توانیم از هر «تقریب‌ساز تابع» برای یادگیری **سیاست بهینه** استفاده کنیم. فرض کنید نماد θ همان پارامتر تقریب‌ساز تابع ما باشد. ما عموما از یک **شبکه عصبی** به عنوان **تقریب‌ساز تابع خود** استفاده می‌کنیم. بنابراین، ما یک **خط‌مشی** π داریم که توسط θ پارامتری شده، در جائیکه θ عملا پارامتر شبکه عصبی است.

فرض کنید ما یک شبکه عصبی با یک پارامتر θ داریم. ابتدا «حالت محیط» را به عنوان ورودی به شبکه، تغذیه می‌کنیم و شبکه ما، احتمال تمام اقداماتی را که می‌توان در آن حالت انجام داد، بعنوان خروجی می‌دهد. یعنی یک

«**توزیع احتمال**» در فضای کنش ما، تولید می‌کند. ما گفتیم که گرادیان سیاست، از یک سیاست احتمالی استفاده می‌کند. بنابراین، سیاست احتمالی مورد نظر، اقدامی را بر اساس توزیع احتمال داده شده توسط شبکه عصبی، انتخاب می‌کند. به این ترتیب می‌توانیم بدون استفاده از تابع Q، مستقیما سیاست را بدست آوریم.

بیایید با یک مثال، بفهمیم که روش گرادیان سیاست، چگونه کار می‌کند. برای درک بهتر مثال، محیط جهان مشبک مورد علاقه خود را در نظر بگیریم. ما می‌دانیم که فضای کنش در محیط دنیای مشبک، چهار عمل ممکن دارد: *بالا*، *پایین*، چپ و *راست*.

با توجه به هر حالت به عنوان ورودی، شبکه عصبی ما، توزیع احتمال در فضای کنش، را بعنوان خروجی می‌دهد. یعنی همانطور که در شکل ۱۰.۲ نشان داده شده است، وقتی حالت **E** را به عنوان ورودی به شبکه تغذیه می‌کنیم، توزیع احتمال بر روی تمام اقدامات ممکن در فضای کنش را به ما برمی‌گرداند. اکنون، سیاست احتمالی ما یک عمل را بر اساس توزیع احتمال داده شده توسط شبکه عصبی انتخاب می‌کند. مثلا، اقدام: *بالا* را در۱۰ درصد مواقع، اقدام: *پایین* را در ۱۰ درصد اوقات، اقدام: چپ را در ۱۰ درصد زمانها، و اقدام: *راست* را در ۷۰ درصد دفعات، برمی‌گزینند.



سیاست

ما نباید **روش DQN** را با روش **گرادیان سیاست** اشتباه بگیریم! در روش **DQN**، ما حالت را به عنوان ورودی به شبکه تغذیه می‌کردیم و شبکه ما مقادیر Qی تمام «**اقدامات ممکن**» در آن حالت را برمی‌گرداند. سپس اقدامی

انتخاب می‌شد که حداکثر مقدار Q را می‌داشت. در روش گرادیان سیاست گرچه، مانند **DQN**، ما حالت را به عنوان ورودی به شبکه تغذیه می‌کنیم ولیکن، شبکه ما در اینجا، «**توزیع احتمال**» بر روی یک فضای کنش را برمی‌گرداند؛ سپس سیاست احتمالی ما از این توزیع احتمال خروجی، بهره گرفته و اقدامی را بر اساس آن، انتخاب می‌کند.

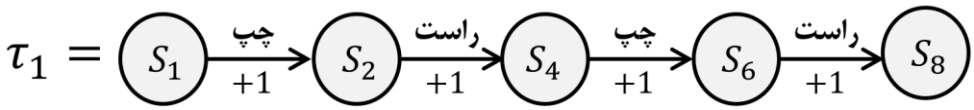
بسیار خب، در روش گرادیان سیاست، شبکه ما توزیع احتمال (یا احتمالات اقدامات) در فضای کنش را برمی‌گرداند، اما احتمالات چقدر دقیق هستند؟ شبکه چگونه یاد می‌گیرد؟

برخلاف یادگیری تحت نظارت، در اینجا ما هیچ داده برچسب گذاری شده‌ای برای آموزش شبکه خود نداریم. بنابراین، شبکهٔ ما اقدام مناسب برای انجام در یک حالت خاص را نمی‌داند. یعنی شبکه نمی‌داند کدام عمل حداکثر پاداش را می‌دهد. بنابراین، احتمالات عمل داده شده توسط شبکه عصبی ما، در تکرارهای نخستین، دقیق نخواهد بود و ممکن است پاداش بدی دریافت کنیم.

اما همین هم خوب است! ما به سادگی، عمل را بر اساس توزیع احتمال داده شده توسط شبکه انتخاب می‌کنیم، پاداش را ذخیره کرده و به حالت بعدی می‌رویم تا جاییکه اپیزود به انتها برسد. یعنی ما یک اپیزود را بازی می‌کنیم و حالت‌ها، اقدامات و پاداش‌های آنرا ذخیره می‌کنیم. خب همین‌ها عملا به داده‌های آموزشی مورد نیاز ما تبدیل می‌شوند. اگر در یک اپیزود برنده شویم، یعنی اگر اپیزود، بازده مثبت یا بازده بالایی (شامل مجموع پاداش‌های اپیزود) داشته باشد، احتمال تمام اقداماتی که در هر حالت تا پایان اپیزود انجام داده‌ایم را افزایش می‌دهیم. ولی اگر بازده منفی یا بازده کم باشد، آنگاه احتمال تمام اقداماتی که در هر حالت تا پایان اپیزود انجام داده‌ایم را کاهش می‌دهیم.

بیایید با یک مثال ساده، این روش را بهتر درک کنیم. فرض کنید حالت‌های S_1 تا S_8 را داریم و هدف ما رسیدن به حالت $S8$ است. فرض کنید فضای کنش ما فقط از دو اقدام تشکیل شده است: *چپ* و *راست*. بنابراین، هنگامی‌که هر حالتی را به شبکه تغذیه می‌کنیم، توزیع احتمال بر روی دو عمل موجود را برمی‌گرداند.

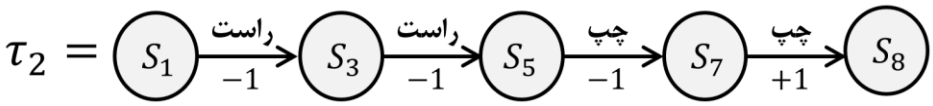
مسیر زیر را در نظر بگیرید (اپیزود) τ_1، که در آن اقدامی را در هر یک از حالات و بر اساس توزیع احتمال بازگردانده شده توسط شبکه (با استفاده از یک سیاست احتمالی) انتخاب می‌کنیم:



شکل ۱۰.۳: مسیر τ_1

بازگشت یا بازده این مسیر $R(\tau_1) = 1 + 1 + 1 + 1 = 4$ است. از آنجایی که بازده مثبتی دریافت کردیم، احتمالات تمام اقداماتی که در هر حالت، تا پایان پردینه (اپیزود) انجام دادیم را افزایش می‌دهیم. یعنی احتمال عمل چپ در S_1، احتمال اقدام *راست* در S_2 و غیره را تا پایان پردینه (اپیزود) افزایش می‌دهیم.

فرض کنید مسیر دیگری بنام τ_2 ایجاد شده، که در آن ما یک عمل را در هر حالت بر اساس توزیع احتمال بازگردانده شده توسط شبکه و با استفاده از یک سیاست احتمالی انتخاب می‌کنیم، همانطور که در شکل ۱۰.۴ نشان داده شده است:



شکل ۱۰.۴: مسیر ۲τ

بازگشت یا بازده این مسیر $R(\tau_2) = -1 - 1 - 1 + 1 = -2$ است. از آنجایی که بازده منفی دریافت کردیم، احتمالات تمام اقداماتی که در هر حالت انجام دادیم را تا پایان پردینه (اپیزود) کاهش می‌دهیم. یعنی احتمالات عمل *راست* را در S_1، احتمال اقدام *راست* در S_3 و غیره را تا پایان اپیزود کاهش خواهیم داد.

بسیار خب، اما دقیقا چگونه می‌توانیم این احتمالات را افزایش و کاهش دهیم؟ ما یاد گرفتیم که اگر بازگشت مسیر مثبت باشد، احتمال همه اقدامات در اپیزود را افزایش دهیم، در غیر این صورت آن را کاهش دهیم. دقیقا چگونه می‌توانیم این کار را انجام دهیم؟ اینجاست که «**روش پس‌انتشار**» به ما کمک می‌کند. ما می‌دانیم که شبکه عصبی را می‌توان با روش پس‌انتشار، آموزش داد.

بنابراین، در طول پس‌انتشار، شبکه ما گرادیان‌ها را محاسبه کرده و پارامترهای شبکه θ را به‌روز می‌کند. به‌روز رسانی گرادیان‌ها به گونه‌ای انجام می‌شود که آن اقداماتی که بازده بالایی را به همراه دارند، احتمالات بالایی بگیرند و

اقداماتی که بازده پایینی دارند، احتمالات پایینی دریافت کنند.

به طور خلاصه، در روش گرادیان سیاست، از یک شبکه عصبی برای یافتن خطمشی بهینه استفاده می‌کنیم. ما پارامتر شبکه یعنی θ را با مقادیر تصادفی (بختکی) مقداردهی اولیه می‌کنیم. در ادامه، حالت را به عنوان ورودی به شبکه تغذیه می‌کنیم و شبکه ما، احتمال اقدامات در آن حالت را برمی‌گرداند. در تکرار اولیه، از آنجایی که شبکه با هیچ داده‌ای، آموزش ندیده است، «مقدار احتمال اقدامات» را بطور تصادفی (بختکی) ارائه می‌دهد. در عین حال، ما یکی از همین اقدامات را بر اساس «توزیع احتمال اقدامات» داده شده توسط شبکه انتخاب کرده و نهایتا، داده‌های بدست آمده یعنی: **«نام هر حالت، اقدام انجام شده در آن حالت و پاداش آن تا پایان پردینه (اپیزود)»** را ذخیره می‌کنیم. اکنون، اینها به داده‌های آموزشی ما تبدیل می‌شوند. اگر در یک اپیزود پیروز شویم، یعنی اگر بازده بالایی داشته باشیم، احتمالات بالایی را به تمام اقدامات اپیزود اختصاص می‌دهیم، در غیر این صورت، احتمالات کمی را به تمام اقدامات اپیزود تخصیص می‌دهیم.

از آنجایی که ما از یک شبکه عصبی برای یافتن **خطمشی** بهینه استفاده می‌کنیم، می‌توانیم این شبکه عصبی را یک **شبکه سیاست** بنامیم. اکنون که درک اجمالی از روش گرادیان **سیاست** پیدا کردیم، در بخش بعدی خواهیم آموخت که شبکه عصبی دقیقا چگونه **سیاست** بهینه را بدست می‌آورد. یعنی یاد خواهیم گرفت که محاسبات گرادیان دقیقا چگونه اتفاق می‌افتد و چگونه شبکه را آموزش می‌دهیم.

درک گرادیان سیاست

در بخش آخر یاد گرفتیم که در روش **نشیب سیاست**، گرادیان‌ها را به گونه ای به‌روز می‌کنیم که اقداماتی که بازده بالایی به همراه دارند، احتمال بالایی داشته باشند و اقداماتی که بازده پایینی به همراه دارند، احتمال کمی‌خواهند داشت. در این بخش یاد خواهیم گرفت که دقیقا چگونه این کار را انجام می‌دهیم.

هدف از روش نشیب سیاست، یافتن پارامتر بهینه θ شبکه عصبی است تا شبکه، توزیع احتمال صحیح را در فضای کنش برگرداند. بنابراین، هدف شبکه ما اختصاص احتمالات بالا به اقداماتی است که بازده مورد انتظار مسیر را به حداکثر می‌رساند. لذا، می‌توانیم تابع هدف خود یعنی J را به صورت زیر بنویسیم:

$$J(\theta) = E_{\tau \sim \pi_\theta(\tau)}[R(\tau)]$$

در معادله قبلی، نمادهای زیر را داریم:

- τ، مسیر است.
- $\tau \sim \pi_\theta(\tau)$ نشان می‌دهد که ما در حال نمونه‌برداری از مسیر بر اساس خط‌مشی π ارائه شده توسط شبکه با پارامتر θ هستیم.
- $R(\tau)$، بازگشت یا بازده مسیر τ است.

بنابراین، بیشینه‌سازی تابع هدف ما، بازگشت یا بازدهٔ مسیر را بیشینه می‌کند. چگونه می‌توانیم تابع هدف قبلی را به حداکثر برسانیم؟ ما مرتبا در باره مسئله کمینه‌سازی صحبت کردیم، جایی که تابع ضرر (تابع هدف) را با محاسبه نشیب تابع ضرر و به‌روز رسانی پارامتر با استفاده از نزول گرادیان، به حداقل می‌رساندیم. اما در اینجا، هدف ما به حداکثر رساندن تابع هدف است، بنابراین ما نشیب‌های تابع هدف را محاسبه کرده و بجای نزول نشیب، در اینجا صعود گرادیان را انجام می‌دهیم.

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

جایی که $\nabla_\theta J(\theta)$ نمادِ نشیب تابع هدف ما است. بنابراین، ما می‌توانیم پارامتر بهینهٔ θ شبکه خود را با استفاده از روش گرادیان افزایشی پیدا کنیم. گرادیان $\nabla_\theta J(\theta)$ از رابطه زیر مشتق می‌شود:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)]$$

در بخش بعدی خواهیم آموخت که دقیقا چگونه این گرادیان را به دست آوریم. در این بخش، بیایید فقط بر روی بدست آوردن یک فهم کامل‌تر از گرادیان سیاست تمرکز کنیم.

ما یاد گرفتیم که پارامتر شبکهٔ خود را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

با جایگزینی مقدار گرادیان، معادلهٔ به‌روز رسانی پارامتر، به صورت زیر تبدیل می‌شود:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

در معادله قبلی، نمادهای مورد استفاده، بصورت زیر هستند:

- $\log \pi_\theta(a_t|s_t)$ نشان دهنده لگاریتم احتمال انجام عمل a در حالت s و در زمان t است.
- $R(\tau)$ نشان دهنده بازگشت یا بازده مسیر است.

ما یاد گرفتیم که گرادیان‌ها را به گونه‌ای به‌روز می‌کنیم که اقداماتی که بازده بالایی دارند احتمال بالایی بگیرند و اقداماتی که بازده پایینی به همراه دارند، احتمال پایینی داشته باشند. حالا بیایید ببینیم دقیقا چگونه این کار را انجام می‌دهیم.

مورد ۱:

فرض کنید با استفاده از خطمشی π_θ یک اپیزود (مسیر) ایجاد می‌کنیم ، جایی که θ پارامتر شبکه است. پس از تولید اپیزود، بازگشت اپیزود را محاسبه می‌کنیم. اگر بازگشت یا بازده اپیزود منفی باشد، مثلا ۱-، یعنی $R(\tau) = -۱$ باشد، احتمال تمام اقداماتی که در هر حالت انجام داده‌ایم را تا پایان اپیزود کاهش می‌دهیم.

ما یاد گرفتیم که معادله به‌روز رسانی پارامتر ما به صورت زیر ارائه می‌شود:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

در معادله قبلی، $\log \pi_\theta(a_t|s_t)$ را در بازده منفی $R(\tau) = -۱$ ضرب کرده‌ایم که نشان می‌دهد ما در حال کاهش لگاریتم احتمال عمل a_t در حالت s_t هستیم. بنابراین، ما یک به‌روز رسانی منفی انجام می‌دهیم. یعنی:

برای هر گام از اپیزود ، $t=0,\ldots,T-1$، پارامتر θ را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, (-۱)$$

این بدان معناست که ما احتمال تمام اقداماتی که در هر حالت تا پایان اپیزود انجام داده‌ایم را کاهش می‌دهیم.

مورد ۲:

فرض کنید با استفاده از خطمشی π_θ یک اپیزود (مسیر) ایجاد می‌کنیم، جایی که θ پارامتر شبکه است. پس از تولید اپیزود، بازگشت اپیزود را محاسبه می‌کنیم. اگر بازگشت اپیزود مثبت باشد، مثلا ۱+ ، یعنی $R(\tau) = +۱$ باشد، احتمال تمام اقداماتی را که در هر حالت انجام داده‌ایم تا پایان اپیزود افزایش می‌دهیم.

ما یاد گرفتیم که معادله به‌روز رسانی پارامتر ما به صورت زیر ارائه می‌شود:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

در معادله قبلی، ضرب $\log \pi_\theta(a_t|s_t)$ در بازده مثبت، $R(\tau) = +۱$ به این معنی است که ما در حال افزایش لگاریتم احتمال انتخاب عمل a_t در حالت s_t هستیم. بنابراین، ما یک به‌روز رسانی مثبت انجام می‌دهیم. یعنی:

برای هر مرحله از اپیزود، $t=0,\ldots,T-1$، پارامتر θ را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, (+۱)$$

بنابراین، اگر بازده مثبت داشته باشیم، احتمال همه اقدامات انجام شده در آن اپیزود را افزایش داده و در غیر این صورت احتمال آنها را کاهش می‌دهیم.

ما یاد گرفتیم که برای هر مرحله از اپیزود، $t=0,\ldots,T-1$، پارامتر θ را به‌روز کنیم:

$$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

ما به سادگی می‌توانیم معادله قبلی را به صورت زیر نشان دهیم:

$$\theta = \theta + \alpha \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

بنابراین، اگر اپیزود (مسیر) بازده بالایی داشته باشد، احتمال تمام اقدامات اپیزود را افزایش می‌دهیم، در غیر این صورت احتمال تمام اقدامات اپیزود را کاهش می‌دهیم. ما یاد گرفتیم که

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)]$$

در مورد این نماد انتظار چطور؟ ما هنوز آن نماد را در معادله به‌روز رسانی خود قرار نداده‌ایم. با توجه به روش مونت کارلو می‌دانیم که می‌توان با استفاده از میانگین، انتظارات را تقریب زد. بنابراین، با استفاده از روش تقریب مونت کارلو، عبارت انتظار را به یک نماد مجموع بر روی N مسیر، تغییر می‌دهیم. لذا، معادله به‌روز رسانی ما به صورت زیر تبدیل می‌شود:

$$\theta = \theta + \alpha \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)$$

این معادله نشان می‌دهد که به جای به‌روز رسانی پارامتر بر اساس یک مسیر واحد، ما مجموعه‌ای از N مسیر را بر اساس خط‌مشی π_θ جمع‌آوری کرده و پارامتر را بر اساس مقدار متوسط آنها، به‌روز می‌کنیم، یعنی:

$$\theta = \theta + \alpha \underbrace{\frac{1}{N} \sum_{i=1}^{N}}_{\text{Sum over N trajrctories}} \underbrace{\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)}_{\text{For each step in the trajectory}}$$

بنابراین، ابتدا N مسیر، که دنبال‌کنندهٔ π_θ را جمع‌آوری کنید؛ نماد این فرایند به شکل زیر است:

$$\{\tau^i\}_{i=1}^{N}$$

حالا گرادیان مورد نظر را به صورت زیر محاسبه کنید:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} [\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)]$$

و سپس پارامتر خود را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

اما ما نمی‌توانیم پارامتر بهینه θ را با به‌روز رسانی پارامتر فقط برای یک تکرار پیدا کنیم. بنابراین، گام قبل را برای مراحل بسیاری، تکرار کرده تا پارامتر بهینه را پیدا کنیم.

اکنون که درک اساسی از نحوه عملکرد روش **نشیب سیاست** پیدا کردیم، در بخش بعدی یاد خواهیم گرفت که چگونه گرادیان سیاست $\nabla_\theta J(\theta)$ را استخراج کنیم. پس از آن، گام به گام با الگوریتم گرادیان سیاست به طور مفصل آشنا خواهیم شد.

استخراج نشیب سیاست

در این بخش به جزئیات بیشتری می‌پردازیم و نحوه محاسبه گرادیان $\nabla_\theta J(\theta)$ را یاد می‌گیریم. ضمنا نشان می‌دهیم که چگونه این نشیب برابر معادلهٔ زیر است:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)]$$

بیایید عمیقا در جذابیت ریاضیات غرق شویم! و ببینیم که چگونه مشتق تابع هدف J را با توجه به پارامتر مدل θ در گامهای ساده محاسبه کنیم. از فرمولهای پیشرو نترسید، در واقع یک مشتق بسیار ساده است. قبل از ادامه، بیایید برخی از پیشنیازهای ریاضی را مرور کرده تا مطالب را بهتر درک کنیم.

تعریف انتظار یا امید ریاضی

فرض کنید X یک متغیر تصادفی گسسته باشد که **تابع جرم احتمال**[۱] یعنی (**pmf**) آن، به صورت $p(x)$ داده شده است. همچنین گیریم f تابعی از یک متغیر تصادفی گسسته X باشد. آنگاه انتظار یک تابع $f(X)$ را می‌توان به صورت زیر تعریف کرد:

[۱] Probability Mass Function (pmf)

$$\mathbb{E}_{x\sim p(x)}[f(X)] = \sum_{x} p(x)f(x) \quad (۱)$$

حال فرض کنید X یک متغیر تصادفی پیوسته باشد که **تابع چگالی احتمال**[۱] یعنی (**pdf**) آن، به صورت $p(x)$ داده می‌شود. باز هم گیریم f تابعی از یک متغیر تصادفی پیوسته X باشد. آنگاه انتظار یک تابع $f(X)$ را می‌توان به صورت زیر تعریف کرد:

$$E_{x\sim p(x)}[f(X)] = \int_{x} p(x)f(x)\,dx \quad (۲)$$

حتما می‌دانید که مشتق لگاریتم به صورت زیر ارائه می‌شود:

$$\nabla_\theta log x = \frac{\nabla_\theta x}{x} \quad (۳)$$

ما ضمنا می‌دانیم که هدف شبکه ما، بیشینه‌سازی بازده مورد انتظار مسیر است. بنابراین، می‌توانیم تابع هدف خود یعنی J را به صورت زیر بنویسیم:

$$J(\theta) = E_{\tau\sim\pi_\theta(\tau)}[R(\tau)] \quad (۴)$$

نمادهای معادله قبلی، به صورت زیرند:

- τ نشاندهنده مسیر است.
- $\tau\sim\pi_\theta(\tau)$ نشان می‌دهد که ما در حال نمونه‌برداری از مسیر بر اساس سیاست π هستیم که توسط شبکه با پارامتر θ داده می‌شود.

[۱] Probability Density Function (pdf)

- $R(\tau)$ بازگشت یا بازده مسیر است.

همانطور که می‌بینیم، تابع هدف ما، یعنی معادله (۴)، به شکل انتظاری است. از تعریف انتظارات ارائه شده در معادله (۲)، می‌توانیم مفهوم انتظار را توسعه داده و معادله (۴) را به صورت زیر بازنویسی کنیم:

$$J(\theta) = \int \pi_\theta(\tau) R(\tau) d\tau$$

اکنون، ما مشتق تابع هدف J خود را با توجه به θ محاسبه می‌کنیم:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) R(\tau) d\tau$$

با ضرب و تقسیم طرف راست معادله بر $\pi_\theta(\tau)$، داریم:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) \frac{\pi_\theta(\tau)}{\pi_\theta(\tau)} R(\tau) d\tau$$

با بازچیدمان عبارات در معادلهٔ قبلی، می‌توانیم بنویسیم:

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} R(\tau) d\tau$$

با استفاده از معادلهٔ (۳) و جایگزینی آن در معادلهٔ بالا خواهیم داشت:

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta log \pi_\theta R(\tau) d\tau$$

از تعریف ارائه شده برای انتظار در معادله (۲)، می‌توانیم معادله قبلی را در قالب نماد انتظار به صورت زیر بازنویسی کنیم:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau)\, R(\tau)] \qquad (۵)$$

معادله قبلی، گرادیان تابع هدف را به ما می‌دهد. اما ما هنوز معادله را حل نکرده‌ایم. همانطور که می‌بینید، در معادله قبلی، عبارت $\nabla_\theta \log \pi_\theta(\tau)$ را داریم. حالا خواهیم دید که چگونه می‌توانیم آن را محاسبه کنیم.

توزیع احتمال مسیر را می‌توان به صورت زیر ارائه داد:

$$\pi_\theta(\tau) = p(s_۰)\prod_{t=۰}^{T-۱}\pi_\theta(a_t|s_t)p(s_{t+۱}|s_t,a_t)$$

جایی که $p(s_۰)$ **توزیع پیشین حالت**[۱] است. با گرفتن لگاریتم از هر دو طرف، می‌توانیم بنویسیم:

$$\log \pi_\theta(\tau) = \log\left[p(s_۰)\prod_{t=۰}^{T-۱}[\pi_\theta(a_t|s_t)\, p(s_{t+۱}|s_t,a_t)]\right]$$

می‌دانیم که لگاریتم یک حاصلضرب برابر با مجموع لگاریتمهاست، یعنی، $\log(ab) = \log a + \log b$

با اعمال این قاعده لگاریتمی در معادله قبلی، می‌توانیم بنویسیم:

$$\log \pi_\theta(\tau) = \log p(s_۰) + \log\prod_{t=۰}^{T-۱}[\pi_\theta(a_t|s_t)\, p(s_{t+۱}|s_t,a_t)]$$

باز هم، ما همان قاعده را اعمال می‌کنیم، یعنی log of product = sum of logs و لذا نماد $\log \Pi$ به نماد $\sum \log s$ تغییر می‌یابد، که در زیر آمده است.

[۱] Initial State Distribution

$$\log \pi_\theta(\tau) = \log p(s\ ) + \sum_{t=}^{T-} \log \pi_\theta(a_t|s_t) + \log\ p(s_{t+}\ |s_t, a_t)$$

اکنون، مشتق را بر حسب θ محاسبه می‌کنیم:

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta[\log p(s_.) + \sum_{t=.}^{T-۱} \log \pi_\theta(a_t|s_t) + \log p(s_{t+۱}|s_t, a_t)]$$

توجه داشته باشید که ما مشتق را بر حسب θ محاسبه می‌کنیم و همانطور که در معادله قبلی می‌بینیم، جمله اول و جمله آخر در سمت راست[۱] (**RHS**) معادله، به θ بستگی ندارد و بنابراین هنگام محاسبه مشتق، صفر می‌شوند. بنابراین معادله ما بصورت زیر تبدیل می‌شود:

$$\nabla_\theta \log \pi_\theta(\tau) = \sum_{t=0}^{T-1} \nabla_\theta \log\ \pi_\theta(a_t|s_t)$$

اکنون که مقدار $\nabla_\theta \log \pi_\theta(\tau)$ را پیدا کردیم، آنرا در معادله (۵) جایگزین کرده و لذا می‌توانیم بنویسیم:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)]$$

خودشه. اما آیا می‌توانیم از این «نماد انتظار» خلاص شویم؟ بله! ما می‌توانیم از روش تقریب مونت کارلو استفاده کنیم و انتظار را به مجموع N مسیر تغییر دهیم. بنابراین، گرادیان نهایی ما به شکل زیر تبدیل می‌شود:

$$\nabla_\theta J(\theta) \approx \frac{۱}{N} \sum_{i=۱}^{N} \left[\sum_{t=.}^{T-۱} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right] \qquad (۶)$$

[۱] Right-Hand Side (RHS)

معادله (۶) نشان می‌دهد که به جای به‌روز رسانی یک پارامتر بر اساس یک مسیر واحد، ما اطلاعات N مسیر را جمع‌آوری کرده و پارامتر را بر اساس مقدار متوسط آن در N مسیر به‌روز می‌کنیم.

لذا پس از محاسبه گرادیان، می‌توانیم پارامتر خود را به صورت زیر به‌روز کنیم:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

بنابراین، در این بخش، ما یاد گرفتیم که چگونه گرادیان سیاست را استخراج کنیم. در بخش بعدی به جزئیات بیشتری می‌پردازیم و گام به گام با الگوریتم گرادیان سیاست آشنا می‌شویم.

الگوریتم ـ گرادیان سیاست

الگوریتم گرادیان سیاستی که تاکنون در مورد آن بحث کردیم اغلب REINFORCE یا گرادیان سیاست مونت کارلو[۱] نامیده می‌شود. گامهای الگوریتم روش REINFORCE در مراحل زیر آورده شده است:

۱. پارامتر شبکه یعنی θ را با مقادیر تصادفی (بختکی)، مقداردهی اولیه کنید.

۲. با پیروی از **خط‌مشی** π، به تعداد N مسیر ایجاد کنید:

$$\{\tau^i\}_{i=1}^{N}$$

۳. **بازگشت** یا **بازده** مسیر را محاسبه کنید: $R(\tau)$

۴. گرادیان‌ها را محاسبه کنید:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau) \right]$$

۵. پارامتر شبکه را به‌روز کنید: $\theta = \theta + \alpha \nabla_\theta J(\theta)$

۶. گامهای ۲ تا ۵ را برای چندین مرحله تکرار کنید.

[۱] Monte Carlo Policy Gradient

همانطور که از این الگوریتم می‌بینیم ، پارامتر θ در هر تکرار به‌روز می‌شود. از آنجایی که ما از سیاست پارامتری شده π_θ استفاده می‌کنیم، خطمشی ما در هر تکرار به‌روز می‌شود.

الگوریتم **نشیب سیاست** را که به تازگی یاد گرفتیم یک روش بر اساس «**سیاست همگاه**» است، زیرا ما فقط از یک سیاست واحد استفاده می‌کنیم. یعنی ما از یک خطمشی برای تولید مسیرها استفاده کرده و همچنین با به‌روز رسانی پارامتر شبکه θ در هر تکرار، همان سیاست را بهبود می‌بخشیم.

ما یاد گرفتیم که در روش **گرادیان سیاست** (روش REINFORCE)، از یک **شبکه سیاست** استفاده می‌کنیم تا توزیع احتمال را در فضای کنش برگرداند و سپس یک اقدام را بر اساس توزیع احتمال بازگردانده شده توسط شبکه، با استفاده از یک سیاست احتمالی انتخاب کنیم. اما این فقط در مورد یک فضای کنش گسسته صدق می‌کند، و ما از **سیاست رسته‌ای یا طبقه‌ای**[1] به عنوان سیاست احتمالی خود استفاده می‌کنیم.

خب اگر فضای کنش ما پیوسته باشد چه باید کرد؟ یعنی وقتی فضای کنش پیوسته است، چگونه می‌توانیم اقدامات را انتخاب کنیم؟ در اینجا، شبکه سیاست ما نمی‌تواند توزیع احتمال را در فضای کنش برگرداند زیرا فضای کنش پیوسته است. بنابراین، در این حالت، شبکه سیاست ما میانگین و واریانس اقدام را به عنوان خروجی برمی‌گرداند و سپس با استفاده از این میانگین و واریانس یک توزیع گاوسی تولید کرده و با نمونه‌برداری از این توزیع گاوسی با استفاده از سیاست گاوسی یک عمل را انتخاب می‌کنیم.

در فصل‌های آینده در این مورد بیشتر خواهیم آموخت. بنابراین، می‌توانیم روش گرادیان سیاست را در هر دو فضای کنش گسسته و کنش پیوسته اعمال کنیم. در مرحله بعد، دو روش مفید را برای کاهش واریانسِ به‌روز رسانی گرادیان سیاست بررسی خواهیم کرد.

[1] Categorical Policy

روش‌های کاهش واریانس

در بخش قبل، یکی از ساده‌ترین روش‌های گرادیان سیاست به نام روش REINFORCE را یاد گرفتیم. یکی از مسائل عمده‌ای که در روش گرادیان سیاست (که در بخش قبلی یاد گرفتیم) با آن مواجه هستیم این است که گرادیان یا نشیب $\nabla_\theta J(\theta)$ در هر بار به‌روز رسانی، واریانس بالایی خواهد داشت. واریانس بالا، اساسا به دلیل تفاوت عمده در بازده اپیزودهاست. همچنین ما یاد گرفتیم که «**نشیب سیاست**»، دارای روش «**سیاست همگاه**» است، به این معنی که ما آن سیاستی را بهبود می‌بخشیم که با همان سیاست در هر تکرار، اپیزودها را ایجاد می‌کنیم. از آنجایی که خط‌مشی در هر تکرار بهبود می‌یابد، بازگشت ما در هر اپیزود بسیار متفاوت است و واریانس بالایی در به‌روزرسانی‌های گرادیان ایجاد می‌کند. وقتی گرادیان‌ها، واریانس بالایی داشته باشند، آنگاه زمان زیادی برای رسیدن به همگرایی مورد نیاز خواهد بود.

بنابراین، اکنون با دو روش مهم برای **کاهش واریانس** آشنا می‌شویم:

۱. **گرادیان‌های سیاست با پاداش باقیمانده[۱] (علی و معلولی[۲])**

۲. **گرادیان‌های سیاست با خط-مبنا**

---

[۱] Reward-To-Go

برای این عبارت ترکیبی، بجز ترجمه واژه به واژه، یعنی پاداش-برای-رفتن، یا پاداش-رفتن، می‌توان ترکیبات دیگری با توجه به فرایند ریاضی آن پیشنهاد داد. مثلا:

- بازده مسیر ازحالا به بعد، از اینجا به بعد، از نیمه راه تا پایان، از این حالت تا آخر، از میانه راه تا انتها، بازده زین پس
- بازده موثر مسیر، بازده بین راهی، بازده مسیرِ بی سر، بازده مسیرِ بی رأس، بازده با شروعِ میانی، بازده با شروعِ بین راهی
- پاداشهای باقیمانده، پاداشهای آینده، پاداشهای آتی، پاداش به جلو، پاداشهای جلویی

[۲] Causality

اصلی که بنا بر آن، احتمال وقوع حالتهای آینده و مقادیر چشم داشتی (یا انتظاری) مشاهده پذیرِ سامانهٔ کوانتومی را می توان از وضعیت کنونی آن به دست آورد.

گرادیان سیاست با پاداش-باقیمانده

ما یاد گرفتیم که گرادیان سیاست به صورت زیر محاسبه می‌شود:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right]$$

حال، ما یک تغییر کوچک در معادله قبلی ایجاد می‌کنیم. ما می‌دانیم که بازگشت یا بازده یک مسیر، مجموع پاداشهای آن مسیر است ، یعنی:

$$R(\tau) = \sum_{t=0}^{T-1} r_t$$

به جای استفاده از بازده مسیر یعنی $R(\tau)$، از مفهومی به نام «پاداش زین پس[1]» یا «پاداش–باقیمانده» با نماد R_t استفاده می‌کنیم. پاداش–باقیمانده اساسا بازده مسیری است که از حالت s_t شروع می‌شود. یعنی به جای ضرب لگاریتم احتمالات در بازده مسیر کامل یا همان $R(\tau)$، در اینجا در هر مرحله از اپیزود، آنها را در «پاداش زین پس» یعنی R_t ضرب می‌کنیم. پاداش–باقیمانده به معنای بازده مسیری است که از حالت s_t شروع می‌شود. اما چرا باید این کار را انجام دهیم؟ بیایید این موضوع را با بررسی جزئیات یک مثال، بهتر درک کنیم.

---

[1] گرادیان سیاست با پاداش زین پس (Reward-to-go=RTG) تکنیکی است در یادگیری تقویتی که مجموع پاداشهای آینده را از یک گام زمانی خاص به بعد نشان می‌دهد (از حالا به بعد). این روش، گرادیان خطمشی استاندارد را با در نظر گرفتن پاداش تجمعی از مرحله زمانی فعلی تا پایان اپیزود (پاداش–به–جلو) هنگام تخمین تابع مزیت، تغییر می‌دهد. این رویکرد به کاهش واریانس در به‌روزرسانی‌های گرادیان خطمشی کمک می‌کند و منجر به یادگیری پایدارتر و اغلب سریعتر می‌شود. RTG اساسا تخمین دقیقتری از تأثیر هر اقدام، ارائه داده و عامل را به سمت اقداماتی هدایت می‌نماید که به پاداشهای بالاتر در آینده کمک می‌کند. از این تکنیک معمولا در الگوریتمها و محیطهای مختلف یادگیری تقویتی استفاده می‌شود.

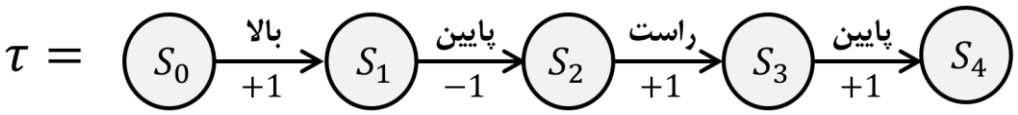
ما یاد گرفتیم که به تعداد N مسیر تولید می‌کنیم و گرادیان را به صورت زیر محاسبه می‌کنیم:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R(\tau)\right]$$

برای درک بهتر، بیایید با تنظیم $N = 1$، فقط یک مسیر را در نظر بگیریم تا بتوانیم بنویسیم:

$$\nabla_\theta J(\theta) = \sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

گیریم، ما مسیر زیر را با دنبال کردن خطمشی π_θ ایجاد کردیم:



شکل ۱۰.۵: یک مسیر کامل

بازده مسیر قبلی $R(\tau) = +1 - 1 + 1 + 1 = 2$ است. اکنون، می‌توانیم گرادیان را به صورت زیر محاسبه کنیم:

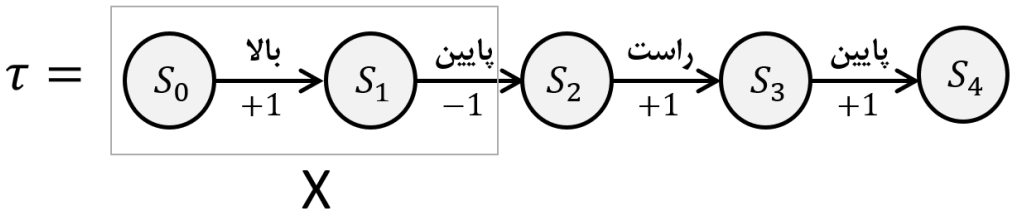
$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(up|s_0)\, R(\tau) + \nabla_\theta \log \pi_\theta(down|s_1)\, R(\tau) + \nabla_\theta \log \pi_\theta(right|s_2)\, R(\tau) + \nabla_\theta \log \pi_\theta(down|s_3)\, R(\tau)$$

همانطور که از معادله قبلی می‌توانیم مشاهده کنیم، در هر مرحله از اپیزود، لگاریتم احتمال اقدام را در بازده مسیر کامل $R(\tau)$ ضرب می‌کنیم که در مثال قبلی ۲ است.

بیایید فرض کنیم که می‌خواهیم بدانیم که *اقدام راست* در حالت s_2 چقدر خوب است. اگر بفهمیم که اقدام right در حالت s_2 عمل خوبی است، می‌توانیم احتمال حرکت به *راست* را در حالت s_2 افزایش دهیم، در غیر این صورت

آن را کاهش می‌دهیم. خوب، چگونه می‌توانیم بگوییم که آیا اقدام right در حالت $s_۲$ خوب است یا خیر؟ همانطور که در بخش قبلی یاد گرفتیم (هنگام بحث در مورد روش REINFORCE)، اگر بازگشت مسیر یعنی $R(\tau)$ زیاد است، سپس احتمال *عمل راست* را در حالت $s_۲$ افزایش می‌دهیم، در غیر این صورت آن را کاهش می‌دهیم.

اما اکنون مجبور نیستیم این کار را انجام دهیم. در عوض، ما می‌توانیم بازگشت (یعنی مجموع پاداش‌های مسیر) را فقط از حالت $s_۲$ به بعد، محاسبه کنیم، زیرا هیچ فایده‌ای ندارد که پاداش‌هایی از مسیر را در نظر بگیریم که قبل از انجام عمل right در حالت $s_۲$ وجود دارند. همانطور که شکل ۱۰.۶ نشان می‌دهد، گنجاندن تمام پاداش‌هایی که قبل از انجام *اقدام راست* در حالت $s_۲$ به دست می‌آوریم، به ما کمک نمی‌کند تا بفهمیم *اقدام راست* در حالت $s_۲$ چقدر خوب است:



شکل ۱۰.۶: همان مسیر شکل قبل

بنابراین، به جای بهره‌گیری از بازده کامل مسیر در تمام گامهای اپیزود، ما می‌توانیم از پاداش-باقیمانده یا پاداش-به-جلو یعنی R_t، استفاده می‌کنیم، که بازگشت مسیری است که از حالت s_t (و نه $s_۰$) شروع می‌شود.

بنابراین، اکنون می‌توانیم بنویسیم:

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(up|s_۰)\, R_۰ + \nabla_\theta \log \pi_\theta(down|s_۱)\, R_۱ + \nabla_\theta \log \pi_\theta(right|s_۲)\, R_۲ + \nabla_\theta \log \pi_\theta(down|s_۳)\, R_۳$$

جایی که $R_۰$ بازده مسیری را نشان می‌دهد که از حالت $s_۰$ شروع می‌شود، $R_۱$ نشان دهنده بازده مسیری است که از حالت $s_۱$ شروع شده و الی آخر. اگر $R_۰$ مقدار بالایی داشته باشد، احتمال اقدام در حالت $s_۰$ را افزایش می‌دهیم، در غیر این صورت آنرا کاهش می‌دهیم. اگر $R_۱$ مقدار بالایی داشته باشد، احتمال عمل را در حالت $s_۱$ افزایش می‌دهیم، در غیر این صورت آن را کاهش می‌دهیم.

بنابراین، اکنون، می‌توانیم معادلهٔ پاداش–باقیمانده (پاداش زین پس) را به صورت زیر تعریف کنیم:

$$R_t = \sum_{t'=0}^{T-1} r(s_{t'}, a_{t'})$$

معادله بالا بیان می‌کند که پاداش–باقیمانده (یا پاداش زین پس) R_t مجموع پاداش‌های مسیری است که از حالت s_t شروع می‌شود. بنابراین، اکنون می‌توانیم گرادیان خود را براساس روش reward-to-go به جای بازده کل مسیر به صورت زیر بازنویسی کنیم:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=0}^{T-1} r(s_{t'}, a_{t'}) \right]$$

و به سادگی می‌توانیم معادله قبلی را به صورت زیر بیان کنیم:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t \right]$$

جایی که پاداش–باقیمانده بصورت زیر است:

$$R_t = \sum_{t'=0}^{T-1} r(s_{t'}, a_{t'})$$

پس از محاسبه گرادیان، پارامتر را به صورت زیر به‌روز می‌کنیم:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

اکنون که فهمیدیم نشیب خطمشی به کمک پاداش–باقیمانده چیست، در بخش بعدی، الگوریتم را برای وضوح بیشتر بررسی خواهیم کرد.

الگوریتم: گرادیان سیاست پاداش‌ـباقیمانده

الگوریتم «**گرادیان سیاست با بازده زمن پس**» شبیه به روش REINFORCE است، با این تفاوت که اکنون به جای استفاده از بازگشت کامل مسیر، عملا پاداش- باقیمانده یا زین پس (بازگشت مسیر شروع شده از حالت S_t) را محاسبه می‌کنیم، همانطور که در اینجا نشان داده شده است:

۱. پارامتر شبکه یعنی θ را با مقادیر تصادفی (بختکی)، مقداردهی اولیه کنید.

۲. تعداد N مسیر با پیروی از سیاست π_θ ایجاد کنید . یعنی:

$$\{\tau^i\}_{i=1}^{N}$$

۳. مقدار پاداشهای باقیمانده (زین پس) یعنی R_t را محاسبه کنید.

٤. گرادیان را بصورت زیر محاسبه کنید:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t\right]$$

۵. پارامتر شبکه را بصورت زیر، به‌روز کنید:

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

٦. گامهای ۲ تا ۵ را برای چندین مراحل تکرار کنید.

از الگوریتم بالا، می‌توانیم مشاهده کنیم که به جای **بازگشت یا بازده مسیر**، از **پاداش-باقیمانده** استفاده می‌کنیم. برای به دست آوردن درک روشنی از نحوه عملکرد گرادیان سیاست پاداش-باقیمانده، بیایید آن را در بخش بعدی پیاده‌سازی کنیم.

تعادل آونگ وارونه با نشیب سیاست

اکنون، بیایید یاد بگیریم که چگونه الگوریتم **گرادیان سیاست** با پاداش–به–جلو (زین پس) را برای انجام وظیفه متعادل کردن آهنگ وارونه[۱] پیاده‌سازی کنیم.

برای درک روشنی از نحوه عملکرد روش **نشیب سیاست**، ما از TensorFlow در حالت غیرمشتاق با غیرفعال کردن رفتار TensorFlow 2 استفاده می‌کنیم.

ابتدا، بیایید کتابخانه‌های لازم را وارد کنیم:

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import numpy as np
import gym
```

محیط آونگ وارونه را با استفاده از gym ایجاد کنید:

```
env = gym.make('CartPole-v0')
```

شکل حالت (فضای کنش) را وارد کنید:

```
state_shape = env.observation_space.shape[0]
```

تعداد اقدامات را وارد کنید:

```
num_actions = env.action_space.n
```

محاسبه پاداش تخفیف‌دار و نرمال‌شده

به جای استفاده مستقیم از جوایز، می‌توانیم از پاداش‌های تخفیف‌دار و عادی استفاده کنیم.

[۱] Cart Pole Balancing

ضریب تخفیف یا تنزیل یعنی γ را تنظیم کنید:

```
gamma = 0.95
```

بیایید تابعی به نام `discount_and_normalize_rewards` برای محاسبه پاداش‌های تخفیف‌دار و نرمال‌شده تعریف کنیم:

```
def discount_and_normalize_rewards(episode_rewards):
```

آرایه‌ای را برای ذخیره جوایز تخفیف‌دار راه اندازی کنید:

```
    discounted_rewards = np.zeros_like(episode_rewards)
```

پاداش تخفیف‌دار را محاسبه کنید:

```
    reward_to_go = 0.0
    for i in reversed(range(len(episode_rewards))):
        reward_to_go = reward_to_go * gamma + episode_rewards[i]
        discounted_rewards[i] = reward_to_go
```

هنجارسازی و بازگرداندن پاداش را انجام دهید:

```
    discounted_rewards -= np.mean(discounted_rewards)
    discounted_rewards /= np.std(discounted_rewards)

    return discounted_rewards
```

ایجاد شبکه سیاست

ابتدا، بیایید مکان–نگهدارنده (جای‌بان) را برای حالت تعریف کنیم:

```
state_ph = tf.placeholder(tf.float32, [None, state_shape], name="state_
ph")
```

جای‌بان را برای اقدام تعریف کنید:

```
action_ph = tf.placeholder(tf.int32, [None, num_actions], name="action_
ph")
```

مکان نگهدارنده را برای پاداش تخفیف‌دار تعریف کنید:

```
discounted_rewards_ph = tf.placeholder(tf.float32, [None,],
name="discounted_rewards")
```

لایه ۱ را تعریف کنید:

```
layer1 = tf.layers.dense(state_ph, units=32, activation=tf.nn.relu)
```

لایه ۲ را تعریف کنید. توجه داشته باشید که تعداد واحدها در لایه ۲، برابر با تعداد اقدامات باشد:

```
layer2 = tf.layers.dense(layer1, units=num_actions)
```

توزیع احتمال بر روی فضای کنش را به عنوان خروجی شبکه بدست آورید. برای اینکار تابع softmax را بر روی نتیجه لایه ۲، اعمال کنید:

```
prob_dist = tf.nn.softmax(layer2)
```

ما یاد گرفتیم که گرادیان را به صورت زیر محاسبه می‌کنیم:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t\right]$$

پس از محاسبه گرادیان، پارامتر شبکه را با استفاده از صعود گرادیان به‌روز می‌کنیم:

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

شاید می‌دانید که یک قرارداد متعارف برای انجام کمینه‌سازی به جای بیشینه‌سازی وجود دارد. بنابراین، می‌توانیم تابع هدف بیشینه‌سازی قبلی را فقط با افزودن یک علامت منفی، به تابع هدف کمینه‌سازی تبدیل کنیم. این را می‌شود با استفاده از `tf.nn.softmax_cross_entropy_with_logits_v2` پیاده‌سازی کرد. بنابراین، می‌توان

سیاست لگاریتم منفی را به صورت زیر تعریف کرد:

```
neg_log_policy = tf.nn.softmax_cross_entropy_with_logits_v2(logits =
layer2, labels = action_ph)
```

حال، بیایید تابع ضرر را تعریف کنیم:

```
loss = tf.reduce_mean(neg_log_policy * discounted_rewards_ph)
```

عملیات آموزش را برای به حداقل رساندن زیان با استفاده از بهینه‌ساز Adam تعریف کنید:

```
train = tf.train.AdamOptimizer(0.01).minimize(loss)
```

آموزش شبکه

اکنون، بیایید شبکه را برای چندین تکرار، آموزش دهیم. برای سادگی، بیایید فقط یک اپیزود در هر تکرار ایجاد کنیم.

تعداد تکرارها را تنظیم کنید:

```
num_iterations = 1000
```

جلسه (کار–دوره) TensorFlow را شروع کنید:

```
with tf.Session() as sess:
```

تمام متغیرهای TensorFlow را مقداردهی اولیه کنید:

```
    sess.run(tf.global_variables_initializer())
```

حلقه تکرار بسازید:

```
    for i in range(num_iterations):
```

یک لیست خالی برای ذخیره حالت‌ها، اقدامات و پاداشهای به دست آمده در اپیزود، مقداردهی اولیه کنید:

```
        episode_states, episode_actions, episode_rewards = [],[],[]
```

اتمام‌یافتگی (پایان) را روی False تنظیم کنید:

```
        done = False
```

بازده (یا بازگشت) را مقداردهی اولیه کنید:

```
        Return = 0
```

با تنظیم مجدد محیط، حالت را مقداردهی اولیه کنید:

```
        state = env.reset()
```

در حالی که هنوز اپیزود تمام نشده است:

```
        while not done:
```

حالت را تغییر شکل (Reshape) دهید:

```
            state = state.reshape([1,4])
```

حالت را به شبکه سیاست، تغذیه می‌کنیم و شبکه بعنوان خروجی، توزیع احتمال بر روی فضای کنش را برمی‌گرداند، که به خط‌مشی احتمالی ما یعنی π تبدیل می‌شود:

```
            pi = sess.run(prob_dist, feed_dict={state_ph: state})
```

اکنون، با استفاده از این سیاست احتمالی، اقدامی را انتخاب می‌کنیم:

```
            a = np.random.choice(range(pi.shape[1]), p=pi.ravel())
```

عمل انتخاب شده را انجام دهید:

```
            next_state, reward, done, info = env.step(a)
```

محیط را رندر کنید:

```
            env.render()
```

بازگشت (بازده) را به‌روز کنید:

```
            Return += reward
```

به روش تک–نشان[۱]، اقدام خود را رمزگذاری کنید:

```
            action = np.zeros(num_actions)
            action[a] = 1
```

حالت، اقدام و پاداش را در لیست‌های مربوطه ذخیره کنید:

```
            episode_states.append(state)
            episode_actions.append(action)
            episode_rewards.append(reward)
```

حالت را به حالت بعدی به‌روز کنید:

```
            state=next_state
```

پاداش تخفیف‌دار و نرمال‌شده را محاسبه کنید:

```
        discounted_rewards= discount_and_normalize_rewards(episode_
rewards)
```

فرهنگ لغت تغذیه[۲] را تعریف کنید:

```
        feed_dict = {state_ph: np.vstack(np.array(episode_states)),
                     action_ph: np.vstack(np.array(episode_actions)),
                     discounted_rewards_ph: discounted_rewards
                    }
```

[۱] One-hot
[۲] Feed Dictionary

شبکه را آموزش دهید:

```
        loss_, _ = sess.run([loss, train], feed_dict=feed_dict)
```

بازگشت را برای هر ۱۰ تکرار چاپ کنید:

```
        if i%10==0:
            print("Iteration:{}, Return: {}".format(i,Return))
```

اکنون که یاد گرفتیم چگونه الگوریتم گرادیان سیاست را با روش پاداشهای-آتی یا پاداشهای-باقیمانده، پیاده سازی کنیم، در بخش بعدی، تکنیک جالب دیگری را برای کاهش واریانس به نام «**گرادیان سیاست با خط مبنا**» یاد خواهیم گرفت.

گرادیان سیاست با روش خط_مبنا

ما آموختیم که با استفاده از یک شبکه عصبی، سیاست بهینه را پیدا کرده و پارامتر شبکه خود را با استفاده از صعود گرادیان به‌روز کنیم:

$$\theta = \theta + \alpha \nabla_\theta J(\theta)$$

جایی که مقدار گرادیان عبارت است از:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t \right]$$

اکنون، برای کاهش واریانس، تابع جدیدی به نام تابع **خط-مبنا** یا تابع **خط-پایه**[1] را معرفی می‌کنیم. تفریق خط مبنای b از بازده (پاداش-باقیمانده) یعنی R_t موجب کاهش واریانس خواهد شد، بنابراین می‌توانیم گرادیان را به صورت زیر بازنویسی کنیم:

[1] baseline function

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t - b)\right]$$

صبر کنید. «تابع خط مبنا» دیگر چیست؟ و چگونه کم کردن آن از R_t، واریانس را کاهش می‌دهد؟ هدف از خط مبنا، کاهش واریانس در بازده است. بنابراین، اگر خط مبنای b مقداری باشد که بتواند بازده مورد انتظار از حالتی که عامل در آن قرار دارد را به ما بدهد، کم کردن b در هر مرحله، باعث کاهش واریانس در بازده می‌شود.

چندین گزینه برای توابع خط مبنا وجود دارد. ما می‌توانیم هر تابعی را به عنوان یک تابع پایه یا تابع مبنا انتخاب کنیم، اما تابع خط مبنا نباید به پارامتر شبکه ما بستگی داشته باشد. یک خط مبنای ساده می‌تواند میانگین بازده مسیرهای نمونه‌برداری شده باشد:

$$b = \frac{1}{N}\sum_{i=1}^{N} R(\tau)$$

بنابراین، کاهش بازده فعلی یعنی R_t و میانگین بازده به اندازه b به ما کمک می‌کند تا واریانس را کاهش دهیم. همانطور که می‌بینیم، تابع مبنا یا پایهٔ ما به پارامتر شبکه θ بستگی ندارد. بنابراین، ما می‌توانیم از هر تابعی به عنوان یک تابع پایه استفاده کنیم ولی نباید بر پارامتر شبکه ما یعنی θ تأثیر بگذارد.

یکی از محبوب‌ترین توابع خط پایه، **تابع ارزش**[1] است. ما یاد گرفتیم که تابع ارزش یا مقدار یک حالت، بازدهی مورد انتظاری است که یک عامل از آن حالت پس از اجرای سیاست π به دست می‌آورد. بنابراین، کاهش ارزش یک حالت (بازده مورد انتظار) و بازده فعلی R_t می‌تواند واریانس را کاهش دهد. لذا، می‌توان گرادیان را به صورت زیر بازنویسی کرد:

[1] value function

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t - V(s_t))\right]$$

به غیر از **تابع ارزش**، می‌توانیم از توابع پایه مختلف مانند **تابع** Q، تابع مزیت و موارد دیگر نیز استفاده کنیم. در فصل بعدی با آنها بیشتر آشنا خواهیم شد.

اما اکنون سوال این است که چگونه می‌توانیم تابع پایه را یاد بگیریم؟ فرض کنید ما از **تابع ارزش** به عنوان تابع مبنا استفاده می‌کنیم. چگونه می‌توانیم «تابع ارزش بهینه» را یاد بگیریم؟ درست همانطور که ما سیاست را تقریب می‌کنیم، می‌توانیم **تابع ارزش** را نیز با استفاده از شبکه عصبی دیگری که توسط ϕ پارامتری شده است، تقریب بزنیم.

یعنی از شبکه دیگری برای تقریب **تابع ارزش** (مقدار یک حالت) استفاده می‌کنیم و می‌توان این شبکه را **شبکه ارزش** نامید. خب، چگونه می‌توانیم این **شبکه ارزش** را آموزش دهیم؟

از آنجایی که ارزش حالت، یک مقدار پیوسته است، می‌توانیم با به حداقل رساندن **میانگین مربعات خطا** (**MSE**)، شبکه را آموزش دهیم. **MSE** را می‌توان در اینجا به عنوان میانگین مجذور اختلاف بین بازده واقعی R_t و بازده پیش‌بینی شده، یعنی $V_\phi(s_t)$ تعریف کرد. بنابراین، تابع هدف شبکه ارزش را می‌توان به صورت زیر تعریف کرد:

$$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}(R_t - V_\phi(s_t))^2$$

ما می‌توانیم خطا را با استفاده از روش نزول گرادیان به حداقل برسانیم و پارامتر شبکه را به صورت زیر به‌روز کنیم:

$$\phi = \phi - \alpha V_\phi J(\phi)$$

بنابراین، در روش گرادیان سیاست با رویکرد خط مبنا، عملا واریانس ما در به‌روزرسانی‌های گرادیان با استفاده از تابع خط مبنا به حداقل می‌رسد. یک تابع پایه یا تابع مبنا می‌تواند هر تابعی باشد ولیکن نباید به پارامتر شبکه یعنی θ

بستگی داشته باشد. ما از **تابع ارزش** به عنوان یک تابع پایه استفاده می‌کنیم، سپس برای تقریب **تابع ارزش** از یک شبکه عصبی متفاوت استفاده می‌کنیم که با ϕ پارامتری شده است و با به حداقل رساندن **MSE** **تابع ارزش بهینه** را پیدا می‌کنیم.

گرادیان سیاست

- **شبکه سیاستِ پارامتری شده توسط θ**: این شبکه، با اعمال گرادیان افزایشی، عملا خطمشی بهینه را بصورت زیر می‌یابد:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t - V_\phi(s_t))\right]$$

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

- **شبکه ارزشِ پارامتری شده توسط ϕ:** یکی از وظایف این شبکه (با بهره‌گیری از مفهوم خط مبنا)، تصحیح واریانس در به‌روز رسانی گرادیان است. این شبکه، همچنین مقدار بهینهٔ ارزش یک حالت را با انجام گرادیان نزولی پیدا می‌کند:

$$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}(R_t - V_\phi(s_t))^2$$

$$\phi = \phi - \alpha\nabla_\phi J(\phi)$$

توجه داشته باشید که تا اینجا ما یاد گرفتیم که چگونه روش گرادیان سیاست با کمک مفهوم تابع مبنا و با استفاده از دو شبکه یعنی **شبکه سیاست** و **شبکه ارزش**، کار می‌کند.

الگوریتم – REINFORCE با خط‌ـمبنا

الگوریتم روش **گرادیان سیاست با کمک تابع خط-مبنا** (یا همان REINFORCE با خط–مبنا) در ادامه آمده است:

۱. پارامتر **شبکه سیاست** یعنی θ و پارامتر **شبکه ارزش** یعنی ϕ را مقداردهی اولیه کنید.

۲. به تعداد N مسیر را با دنبال‌روی از خط‌مشی π ایجاد کنید: $\{\tau^i\}_{i=1}^{N}$

۳. بازده (**پاداش**–باقیمانده، زین پس) یعنی R_t را محاسبه کنید.

۴. گرادیان **سیاست** را بصورت زیر محاسبه کنید:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t|s_t)\,(R_t - V(s_t))\right]$$

۵. پارامتر **شبکه سیاست** یعنی θ را با استفاده از نشیب صعودی به صورت زیر به‌روز کنید:

$$\theta = \theta + \alpha\nabla_\theta J(\theta)$$

۶. مجذور مربعات خطاهای شبکه ارزش (**MSE**) را محاسبه کنید:

$$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}(R_t - V_\phi(s_t))^2$$

۷. گرادیانهای $\nabla_\phi J(\phi)$ را محاسبه کرده و پارامتر **شبکه ارزش** یعنی ϕ را با استفاده از گرادیان نزولی، بروزرسانی نمایید:

$$\phi = \phi - \alpha\nabla_\phi J(\phi)$$

۸. گامهای ۲ تا ۷ را برای چندین مرحله تکرار کنید.

خلاصه

ما در ابتدای فصل یادآوری کردیم که بر اساس روش‌های **مبتنی بر ارزش** (یا **ارزش-محور**)، ما سیاست بهینه را از **تابع** Q بهینه (مقادیر Q) استخراج می‌کنیم. سپس یاد گرفتیم که محاسبه **تابع** Q هنگامیکه که فضای کنش ما پیوسته است، کاری بس دشوار می‌باشد. برای اینکار ما می‌توانیم فضای کنش را گسسته‌سازی کنیم. با این حال، گسسته‌سازی همیشه مطلوب نیست و منجر به از دست دادن چندین ویژگی مهم و یک فضای کنش با مجموعه عظیمی از اقدامات می‌شود.

از این رو، به روش **مبتنی بر سیاست** (یا **سیاست-محور**) متوسل شدیم. در روش سیاست محور، ما **سیاست** بهینه را بدون تابع Q محاسبه می‌کنیم. ما یکی از محبوب ترین روش‌های مبتنی بر سیاست به نام **گرادیان سیاست** را یاد گرفتیم که در آن سیاست بهینه را مستقیما با پارامتری‌سازی سیاست با استفاده از برخی پارامترها همچون θ پیدا می‌کنیم.

همچنین یاد گرفتیم که در روش گرادیان سیاست، اقدامات را بر اساس توزیع احتمال عمل که خروجی شبکه ماست گزینش کنیم. اگر در یک پردینه (اپیزود) را برنده شویم، یعنی اگر بازده بالایی داشته باشیم، آنگاه احتمالات بالایی را به تمام اقدامات موجود در آن پردینه (اپیزود) اختصاص می‌دهیم، در غیر این صورت به تمام اقدامات موجود در آن پردینه (اپیزود) احتمالات پایینی تخصیص می‌دهیم. بعدا یاد گرفتیم که چگونه گام به گام گرادیان سیاست را استخراج کنیم و سپس الگوریتم روش گرادیان سیاست را با جزئیات بیشتری بررسی کردیم.

در بخش پایانی فصل، با روش‌های کاهش واریانس مانند پاداش-به-جلو (پاداش زین پس) و نیز روش **گرادیان سیاست با کمک تابع-مبنا** آشنا شدیم. در روش گرادیان خطمشی با کمک تابع-مبنا از دو شبکه به نام‌های **شبکه سیاست** و **شبکه ارزش** استفاده می‌شود. نقش **شبکه سیاست**، یافتن خطمشی بهینه است و نقش **شبکه ارزش**، تصحیح به‌روز رسانی گرادیان در شبکه خطمشی با برآورد تابع ارزش است.

در فصل بعدی، با مجموعه جالب دیگری از الگوریتم‌ها به نام روش‌های بازیگر-منتقد آشنا خواهیم شد.

سوالات

بیایید درک خود را از روش گرادیان سیاست با پاسخ دادن به سوالات زیر ارزیابی کنیم:

۱. روش **مبتنی بر ارزش** (یا **ارزش-محور**) چیست؟
۲. چرا به روش **مبتنی بر سیاست** (یا **سیاست-محور**) نیاز داریم؟
۳. روش **گرادیان سیاست** چگونه کار می‌کند؟
۴. چگونه گرادیان را در روش **گرادیان سیاست** محاسبه کنیم؟
۵. پاداش زین پس (یا پاداش-باقیمانده) چیست؟
۶. **گرادیان سیاست** به کمک تابع-پایه چیست؟
۷. تابع پایه (یا تابع مبنا) را تعریف کنید.

بیشتر بخوانید

برای کسب اطلاعات بیشتر در مورد گرادیان خطمشی، می‌توان به مقاله زیر مراجعه کرد:

- **Policy Gradient Methods for Reinforcement Learning with Function Approximation** by *Richard S. Sutton et al.*, https://papers.nips.cc/paper/1713-policy-gradient-methods-for-reinforcement-learning-with-function-approximation.pdf