توی این پروژه قصد داریم با استفاده از الگوریتم‌های یادگیری تقویتی یک عامل (Agent) رو در یک محیط grid-world هدایت کنیم تا به موقعیت هدف برسه.

راه‌اندازی محیط پروژه

اول از همه پروژه رو از این لینک دریافت کنید. برای اجرای محیط پروژه به کتابخونه Gymnasium نیازه که باید با pip نصب کنید. کل کدهای پروژه در یک نوت‌بوک قرار داره و شما باید بخش‌های TODO رو انجام بدید.

آشنایی با کتابخانه Gymnasium

کتابخونه Gym که قبلا توسط OpenAI توسعه داده میشد، یک کتابخونه بسیار پرکاربرده که در اون environment های مختلفی پیاده‌سازی شده و شما می‌تونید با API ای که داره با محیط‌هاش کار کنید و الگوریتم‌های یادگیری تقویتی رو تست کنید. توضیحات کلی نحوه کار کردن با Gym رو در این لینک بخونید.

آشنایی با محیط Frozen Lake

محیط Frozen Lake یک grid-world ساده است که در اون عامل باید روی یک دریاچه یخ‌زده حرکت کنه، داخل آب نیافته و به خونه هدف برسه.

حالت‌ها: در مجموع ۱۶ خونه وجود داره در محیط که، از سطر بالا تا پایین، به ترتیب خونه‌ها شماره‌های ۰ تا ۱۵ می‌گیرن. عامل همواره از خونه ۰ شروع به حرکت می‌کنه و باید به خونه ۱۵ برسه. همچنین، محل سوراخ‌ها ثابته. در هر لحظه از زمان، عامل به شماره خونه‌ای که در اون قرار داره دسترسی داره.

اعمال: در هر لحظه از زمان، عامل می‌تونه به سمت چپ، پایین، راست یا بالا حرکت کنه. این اعمال به ترتیب از ۰ تا ۳ شماره‌گذاری میشن.

پاداش: اگر عامل وارد خونه هدف بشه امتیاز ۱+ دریافت می‌کنه و در تمامی خونه‌های دیگه امتیاز ۰ می‌گیره.

پایان قسمت: اگر عامل داخل سوراخ بیافته، به هدف برسه یا بیش از ۱۰۰ واحد زمان بگذره و اتفاقی نیافتاده باشه، episode به پایان می‌رسه.

دینامیک محیط: این محیط دوتا حالت داره، اگر is_slippery = False باشه، محیط به صورت قطعی کار می‌کنه و اگر عامل تصمیم بگیره به یک سمت حرکت کنه، موفق میشه دقیقا به همون سمت بره. اما اگر is_slippery = True باشه، محیط به صورت stochastic کار می‌کنه و عامل با احتمال ۱/۳ ممکنه به یکی از سمت‌های کناری سُر بخوره. به عنوان مثال، اگر عمل حرکت به سمت پایین انتخاب بشه، به احتمال ۱/۳ حرکت به سمت پایین اتفاق می‌افته، به احتمال ۱/۳ به سمت چپ، و به احتمال ۱/۳ به سمت راست.

سایر توضیحات در مورد این محیط رو در این لینک بخونید. بعد بخش‌های Utils و Random Walk نوت‌بوک رو بخونید و اجرا کنید تا با این محیط و API کتابخونه Gym آشنا بشید.

الگوریتم ۱: Iterative Policy Evaluation

قبل اینکه بریم سراغ یادگیری سیاست بهینه توسط عامل، بیاید فرض کنیم که یک Policy داریم و می‌خوایم اون رو ارزیابی کنیم. ما به دنبال اینیم که ببینیم اگر عامل طبق اون سیاست پیش بره، Value حالت‌هایی که در اون‌ها قرار می‌گیره چقدره. به طور دقیق، ما به دنبال تابع زیر هستیم:

Definition

The *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

Definition

The *return* G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

ارزش حالت S با داشتن سیاست π به این معنیه که اگر از اون حالت شروع کنیم و اعمال رو مطابق سیاستی که داریم انتخاب کنیم، به طور میانگین چه مجموع پاداش تخفیف‌یافته‌ای رو در آینده دریافت می‌کنیم.

همونطور که توی درس خوندید، یکی از راه‌هایی که می‌تونیم تابع ارزش رو بدست بیاریم، استفاده از Dynamic Programming هستش. به طور خاص، در اینجا ما می‌خوایم از Iterative Policy Evaluation استفاده کنیم. این الگوریتم از یک حدس اولیه نسبت به تابع ارزش شروع

می‌کنه و بعد با استفاده از Bellman Equation مقادیر ارزش رو به مرور بروزرسانی می‌کنه تا وقتی که الگوریتم همگرا بشه. سودوکد الگوریتم به شکل زیره:

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

برای پیاده‌سازی این الگوریتم، نیاز داریم که مدلی از محیط رو داشته باشیم. این مدل به ما میگه که چه حالاتی و اعمالی وجود داره، کدوم حالت‌ها terminal state هستن، اعمال باعث چه تغییری در حالت عامل میشه، reward ها به چه شکل هستن و غیره. ما این مدل رو در کلاس FrozenLake MDP پیاده‌سازی کردیم. کدهای این کلاس و توابعش رو بخونید و اجرا بگیرید تا باهاشون آشنا بشید. پارامتر is_slippery رو هم تغییر بدید تا تاثیرش رو ببینید.

حالا شما باید تابع policy_evaluation رو پیادسازی کنید. دقت کنید که توی این پروژه ما به دنبال یادگیری سیاست deterministic هستیم. همچنین، برای شرط خاتمه حلقه، می‌تونید طبق سودوکد بالا دلتا حساب کنید یا اینکه شبیه کد نوت‌بوک، num_iteration در نظر بگیرید.

سپس برای هر دو سیاستی که در نوت‌بوک نوشته شده و با is_slippery = False کدتون رو اجرا کنید. بعد state value های بدست اومده رو گزارش و تحلیل کنید.

الگوریتم ۲: Policy Iteration

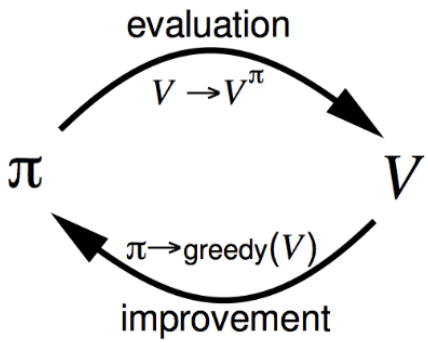
حالا که تونستیم سیاست‌ها رو ارزیابی کنیم، قدم بعدی اینه که سیاست بهینه رو بدست بیاریم. در اینجا می‌خوایم از الگوریتم Policy Iteration استفاده کنیم. به طور خلاصه، این الگوریتم از یک سیاست رندوم شروع می‌کنه و اون رو ارزیابی می‌کنه، بعدش با توجه به ارزش حالت‌ها، سیاست رو به صورت حریصانه بروزرسانی می‌کنه تا اعمال بهتری انتخاب بشن. این کار تکرار میشه تا سیاست ما بهینه بشه. سودوکد و شمای کلی این الگوریتم به شکل زیر هست:

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \left[r + \gamma V(s') \right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma V(s') \right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow false$
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2



شما باید توابع greedy_policy_improvement و policy_iteration رو پیاده‌سازی کنید. سپس بر روی محیط با is_slippery = False و بعد is_slippery = True تست کنید و سیاست بدست آمده را تحلیل کنید.

آیا برای is_slippery = True آیا اصلا سیاستی می‌تونه وجود داشته باشه که بتونه به صورت ۱۰۰ درصدی عامل رو به مقصد برسونه؟

الگوریتم ۳: Q-Learning

تا اینجا فرض کردیم که ما به مدل environment دسترسی داریم و عامل بدون اینکه با محیط تعامل داشته باشه، دینامیک محیط رو به طور کامل از قبل می‌دونه. اما این پیش فرض همیشه برقرار نیست. حالا می‌خوایم به سراغ الگوریتم‌های Model-free بریم. به طور خاص، ما می‌خوایم الگوریتم Q-Learning رو پیاده‌سازی کنیم. سودوکد این الگوریتم به شکل زیر هستش:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop 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

برای پیاده‌سازی این الگوریتم شما باید توابع موجود در کلاس QAgent و تابع train رو پیاده‌سازی کنید. دقت کنید که دیگه نباید از Frozenlake MDP استفاده کنید. همچنین، توجه کنید که الگوریتم Q-learning تعدادی هایپرپارامترها داره و ممکنه نیاز شه که اون‌ها را تغییر بدید تا الگوریتم همگرا بشه.

بر روی محیط با is_slippery = False و بعد is_slippery = True تست کنید و سیاست بدست آمده را تحلیل کنید.