دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکده برق و کامپیوتر

گزارش تمرین شماره ۵
درس یادگیری تعاملی
پاییز ۱۴۰۰

نام و نام خانوادگی	سحر رجبی
شماره دانشجویی	۸۱۰۱۹۹۱۶۵

فهرست

چکیده

در این تمرین، به حل مسائل با حالت‌های پیوسته پرداختیم. در مواجهه با این مسائل، به دلیل نفـرین ابعاد و پیچیدگی‌های محاسباتی نمی‌توان از روش‌ها گذشته‌ی استفاده از حالت‌ها استفاده کرد و در اینجـا ما با استفاده از تخمین توابع و مدل‌های یادگیری عمیق سعی کردیم تا مـدل‌هایی آمـوزش دهیم کـه بـا گرفتن حالت به صورت پیوسته، قادر به تخمین ارزش اعمال در آن‌هـا باشـند. همچـنین در این روش‌هـا قابلیت تعمیم‌دهی به حالت‌های مجاور را نیز خواهیم داشت.

سوال ۱ - گسسته‌سازی با روش Tile Coding

هدف سوال

یکی از روش‌های استفاده از حالت‌های پیوسته، گسسته‌سازی آن‌هاست. درواقع در حالت ساده می‌توانیم هر بازه را با یک عدد نمایش دهیم و بعد از آن، مشابه قبل، می‌توانیم با حالت‌های گسسته کار کنیم. اما فرضی وجود دارد که احتمالاً ارزش حالت‌های مجاور، به یکدیگر نزدیک هستند. این فرض در تمامی حالات صادق نیست، اما در مسائل خاصی درست است. به این منظور در روش Tile Coding ما از چند tiling که با offset از یکدیگر قرار گرفته‌اند استفاده میکنیم که هر کدام از آن‌ها صفحات را به قسمت‌هایی تقسیم کرده‌اند. هر نقطه در فضای حالت ما می‌تواند در فقط و فقط یکی از این قسمت‌ها در هر tiling قرار بگیرد و ما از شناسه‌ی این tilingها به عنوان نمایش حالت استفاده خواهیم کرد در حالی که نقاط نزدیک به یکدیگر در فضای حالت، در برخی از این قسمت‌ها با یکدیگر اشتراک خواهند داشت در نتیجه ما تا حدی قابلیت تعمیم را هم در بین آن‌ها خواهیم داشت.

توضیح پیاده سازی

برای پیاده‌سازی این سؤال در ابتدا ما کلاس MountainCarTileCoder را طراحی کردیم که در آن تابع get_tiles با ورودی گرفتن مکان و سرعت عامل، با توجه به مقدار کمینه و بیشینه‌ی هر کدام از آن‌ها، مقدارها را اسکیل می‌کند و با استفاده از کتابخانه‌ی tiles (کدهای آن در بالای کدهای پیاده‌سازی‌شده قرار گرفته است) لیست مربوط به tileهایی که نقطه‌ی داده‌شده عضو آن‌هاست را برمی‌گرداند.

برای پیاده‌سازی محیط این مساله ما از محیط MountainCar_Env کتابخانه‌ی gym استفاده کردیم و سپس عاملی طراحی کردیم تا با تعامل با این محیط قادر به یادگیری راه حل این مساله باشد. برای این کار از الگوریتم SARSA استفاده کردیم. پیاده‌سازی این الگوریتم، مطابق شبه‌کد کتاب Sutton، که در شکل-۱ قابل مطالعه است انجام شده.

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$

مطابق این الگوریتم، ما در هر اپیزود استیت اولیه و عمل مربوطه را انتخاب می‌کنیم و بعد از اعمال آن در محیط، استیت بعدی، پاداش عمل و نهایی بودن یا نبودن استیت بعد را دریافت خواهیم کرد. در

حالتی که به یک استیت نهایی نرفته‌باشیم، با استفاده از سیاست فعلی، عمل استیت بعدی را انتخاب خواهیم کرد و با استفاده از ارزش استیت-عمل در حالت بعدی، مقدار return خود را برابر مقدار زیر تعریف می‌کنیم.

G = reward + discount-factor*Q(next_state, next_action)

و از آن برای به‌روزرسانی وزن‌ها استفاده خواهیم کرد. تفاوت این الگوریتم با حالت گسسته آن است که در حالت گسسته، ما q-valueها را در جدول و یا ساختمان‌داده‌های مختلف نگهداری می‌کردیم. حال آنکه درحالت پیوسته، ما از توابع تخمین -در این حالت tile-coding- استفاده می‌کنیم و ارزش هر استیت را به صورت زیر محاسبه خواهیم کرد:

$Q(s, a) = w^T$tile-coding(s, a)

و با این تعریف، در هر مرحله، فرمول زیر برای به‌روزرسانی وزن‌های w که باید در طی روند یادگیری انتخاب شوند، در الگوریتم استفاده خواهد شد:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w}) \right] \nabla \hat{q}(S, A, \mathbf{w})$$

و از آنجایی که q-valueها نسبت به wخطی هستند، مقدار مشتق این تابع نسبت به q، به سادگی برابر با مقدار خروجی تابع tile-coding خواهد بود. شکل ۲ پیاده‌سازی ما از این الگوریتم را نمایش می‌دهد که مطابق با همین شبه‌کد انجام شده است.

```
f sarsa(self):

  steps = []
  for episode in range(self.episodes):

      self.env.reset()
      step_counter = 0

      s = self.env.state
      a = self.select_action(s)
      s_coded = self.state_coder.get_tiles(s[0], s[1], [a], self.env)
      print(f'episode: {episode}')
      while True:

          step_counter += 1
          next_s, reward, done, _ = self.env.step(a)
          if done:
              self.theta += self.alpha*(reward - self.q_values[tuple(s_coded.tolist())])*s_coded
              break
          next_a = self.select_action(next_s)
          next_s_coded = self.state_coder.get_tiles(next_s[0], next_s[1], [next_a], self.env)
          G = reward + self.discount*self.q_values[tuple(next_s_coded.tolist())]
          self.theta += self.alpha*(G - self.q_values[tuple(s_coded.tolist())])*s_coded

          s, s_coded, a = next_s, next_s_coded, next_a
          self.update_q(s_coded)
          if (episode+1)%5 == 0:
              self.epsilon *= 0.9
      print(step_counter)
      steps.append(step_counter)

  return steps
```

شکل ۲- پیاده‌سازی الگوریتم SARSA برای حالت پیوسته مطابق شبه‌کد شکل ۱

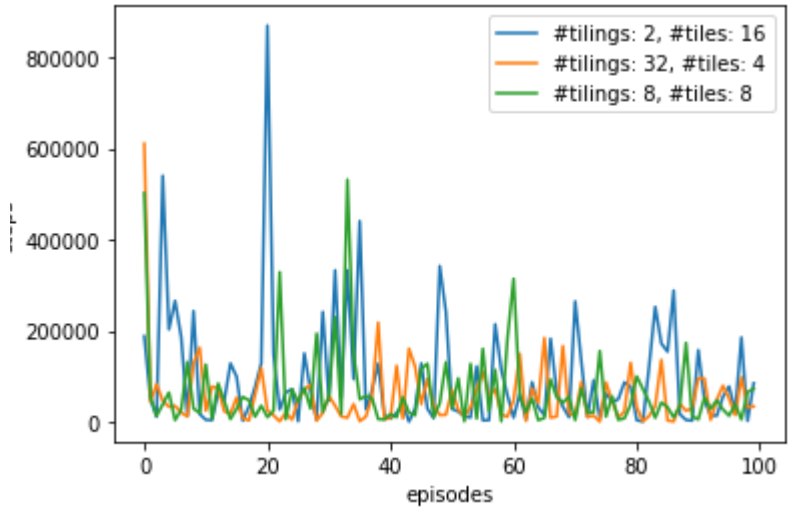
برای نگهداری مقادیر q-value، بعد از تبدیل حالت به نمایش tile-coding، از یک dictionary استفاده کردیم و تابع select_action هم به این صورت عمل می‌کند که برای هر حالتی که در ورودی به آن داده‌شود، با توجه به مقدار q-value مربوط به هر عمل در آن استیت، با اجرای الگوریتم epsilon-greedy به احتمال epsilon یک عمل را به صورت تصادفی انتخاب خواهد کرد و یا با احتمال ۱-epsilon عمل با بیشترین ارزش را انتخاب می‌کند.

ما این الگوریتم را به ازای نرخ یادگیری برابر با ۰.۵ تقسیم بر تعداد tilingها و همچنین مقدار اولیه‌ی epsilon برابر با ۰.۲ اجرا کردیم.

نتایج

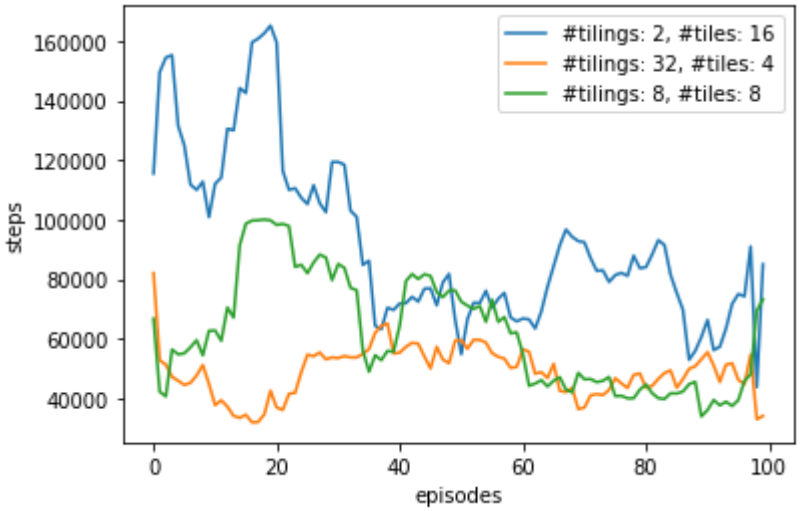
از آنجایی که مقدار اولیه‌ی ارزش حالت-عمل‌ها را برابر با صفر در نظر گرفتیم و در هر حرکت، عامل پاداش -۱ را دریافت خواهد کرد (البته به جز استیت هدف)، تمامی حالات جز نقطه‌ی هدف، در ابتدا ارزشی منفی دارند. در این حالت، در الگوریتم epsilon-greedy، حالت-عمل‌های دیده نشده ارزش صفر، و سایر حالت-عمل‌ها ارزش منفی خواهند داشت. در نتیجه با احتمال بیشتری، عامل به سمت استیت‌های ندیده خواهد رفت. اما همانطور که در Figure10.2 در کتاب Sutton هم مشخص است، تعداد قدم‌های لازم برای حل این مساله در اپیزودهای ابتدایی بسیار زیاد است. (نمودار به صورت log-scale رسم شده است!) و عامل به تدریج با propagate کردن ارزش استیت نهایی می‌تواند به سیاست بهینه برسد. البته در استیت‌های اولیه که ایجنت دیدی ندارد و باید با سعی و خطا به استیت نهایی برسد؛ چالش جدی خواهیم داشت! شکل ۳ نمودار تعداد مراحل در هر اپیزود را برای سه کانفیگ گفته‌شده نشان می‌دهد.

پ.ن: متأسفانه به علت خرابی سیستم، مجبور به استفاده از گریس و همچنین سیستمی قدیمی بودم و زمان اینکه در اپیزودهای بیشتری و تعداد تکرار بیشتر الگوریتم را اجرا کنم؛ نداشتم.



شکل ۳- نمودار تعداد مراحل هر اپیزود به ازای سه کافیگ متفاوت برای tile-coding

همانطور که مشخص است، در این تعداد اپیزود، حالت ۳۲ tiling که هر کدام هر یک از ابعاد استیت را به ۴ بازه تقسیم کرده است؛ بهترین نتیجه و کمترین تعداد استپ برای رسیدن به حالت نهایی را دارد. این نتیجه احتمالاً به علت generalization بالای این coding به علت استفاده از تعداد زیادی tiling است. همچنین می‌دانیم که راه حل کلی این مساله این است که سرعت و شتاب با یکدیگر هم‌جهت باشند تا عامل بتواند با رفت و آمدهای متوالی، سرعت اولیه‌ی لازم برای رسیدن به بالای تپه را پیدا کند. در نتیجه در این مساله اتفاقاً generalization اهمیت بسیار بسیار بالایی دارد. چرا که در نیمی از مسیر باید شتاب -۱ داشته‌باشیم و در نیم دیگر آن شتاب +۱ و این در نقاط مجاور هر نیمه هم تفاوتی ندارد. به این ترتیب نتیجه‌ی به‌دست‌آمده منطقی هم هست و با توجه به اینکه نتیجه‌ی ۸ tiling که هر کدام در هر بعد ۸ tile ایجاد می‌کنند از نتیجه‌ی ۲ tiling و ۱۶ tile به مراتب بهتر است، با یک شاهد دیگر هم این ادعا اثبات می‌شود. شکل ۴ که با استفاده از یک moving average با اندازه‌ی ۱۰ رسم شده، این ادعا را به‌طور واضح‌تری نشان می‌دهد.



شکل ۴- نمودار تعداد مراحل در هر اپیزود با استفاده از پنجره‌ی متحرک با اندازه‌ی ۱۰

روند اجرای کد پیاده‌سازی

به دلیل مشکلات با اجرای محیط‌های gym، فایل mountaincar_env.py در کنار فایل کدها قرار دارد و همچنین در صورتی که بخواهید محیط را رندر کنید هم فایل pyglet_rendering.py در کنار فایل کدها قرار گرفته و در صورتی که این دو فایل، در کنار Question1.ipynb باشند، این کد بدون مشکل اجرا خواهد شد.

سوال ۲- کنترل فرود هواپیما

هدف سوال

در این بخش با استفاده از Deep Q-learning باید سیاست بهینه برای فرود هواپیما را یادبگیریم که در آن وزن‌ها را با استفاده از یک شبکه‌ی عصبی آموزش می‌دهیم و از این وزن‌ها درواقع برای ربط دادن استیت‌ها به ارزش استیت-اکشن‌ها استفاده خواهیم کرد.

توضیح پیاده سازی

برای پیاده‌سازی این سوال، ما نیاز به یک شبکه‌ی عصبی داریم که آموزش ببیند برای اینکه استیت‌های ورودی را به ارزش اعمال در استیت ورودی مپ کند. برای این کار ما کلاس شکل ۵ را طراحی کردیم. در استیت ورودی خود، تصویر و یا سیگنالی شبیه به آن نداشتیم، پس می‌توانستیم به سادگی از یک شبکه‌ی MLP ساده با ۳ لایه‌ی Fully-connected استفاده کنیم.

```
ass DeepNetwork (nn.Module):

  def __init__(self, input_d, output_d):

      super(DeepNetwork, self).__init__()


      self.ln1 = nn.Linear(input_d, 15)
      self.ln2 = nn.Linear(15, 10)
      self.output = nn.Linear(10, 4)

  def forward(self, state):

      x = F.relu(self.ln1(state))
      x = self.dropout(x)
      x = F.relu(self.ln2(x))
      x = self.dropout(x)
      out = self.output(x)
```

Figure ۱: شکل ۵- کلاس Deep Q-Learning Network

برای پیاده‌سازی عامل، در ابتدا پیاده‌سازی یک buffer برای ذخیره‌ی تجربیات را توضیح خواهیم داد تا نهایتاً عمل‌کرد عامل را شرح بدهیم. کلاس ReplayBuffer در فایل experience_replay.py به این منظور ایجاد شده است تا بتوانیم تجربیات حاصل از تعامل عامل و محیط را در آن ذخیره کنیم و بعد با استفاده از نمونه‌گیری از این تجربیات، از آن‌ها برای آموزش مدل استفاده کنیم. به دو علت این کار لازم است؛ اول اینکه آموزش مدل به ازای هر نمونه به صرفه و مناسب نیست و علاوه بر بالابردن زمان آموزش، می‌تواند باعث واگرایی بشود. دوم اینکه ترتیب مشاهدات بدست‌آمده در اثر تعامل با محیط، می‌تواند منجر به ایجاد بایاس در مدل و یا overfit و در نتیجه کاهش generalization بشود. به این منظور ما مجموعه‌ی (state, action, next_state, reward, done)ـ را در این بافر ذخیره می‌کنیم و سپس با نمونه‌گیری از این بافر به آموزش مدل می‌پردازیم. کلاس پیاده‌سازی شده دارای ۲ متد است. متد add_trial در صورتی که بافر فضای آزاد داشته‌باشد مجموعه‌ی گفته‌شده را ذخیر می‌کند و متد sample به اندازه‌ی batch-size از بافر نمونه برمیدارد.

در نهایت برای پیاده‌سازی DQNAgent، به این صورت عمل می‌کنیم که در هر اپیزود، عامل استیت اولیه و اکشن را مشخص می‌کند، استیت بعدی، پاداش و نهایی یا غیرنهایی بودن استیت بعدی را دریافت

می‌کند و این تجربه را در بافر خود ذخیره می‌کند در صورتی که بافر به اندازه‌ی batch-size نمونه داشته باشد، عامل با استفاده از تابع batch-train یک سمپل از تجربیات بافر برمی‌دارد و مدل را بر روی آن آموزش می‌دهد. سپس به استیت بعدی در اپیزود می‌رود و این کار را تکرار می‌کند تا زمانی که به استیت نهایی برسد و اپیزود دیگری را آغاز کند. در مرحله‌ی آموزش، تابع خطا سعی در کمینه‌کردن مقدار اختلاف G در روش Q-Learning و تخمین ارزش-عمل استیت‌ها را دارد. در Q-Learning مقدار G مطابق زیر تعریف می‌شود:

$$G = reward + discount*Max_a Q(s_{t+1}, a)$$

درواقع تابع خطا مطابق رابطه‌ی زیر خواهد بود:

$$E = (G - Q(s_t, a_t))^2$$

شکل ۶ پیاده‌سازی تابع train برای اجرای روند آموزش و شکل ۷ تابع train_batch که برای آموزش مدل شبکه‌ی عصبی بر روی نمونه‌ی گرفته‌شده از بافر است را نشان می‌دهد

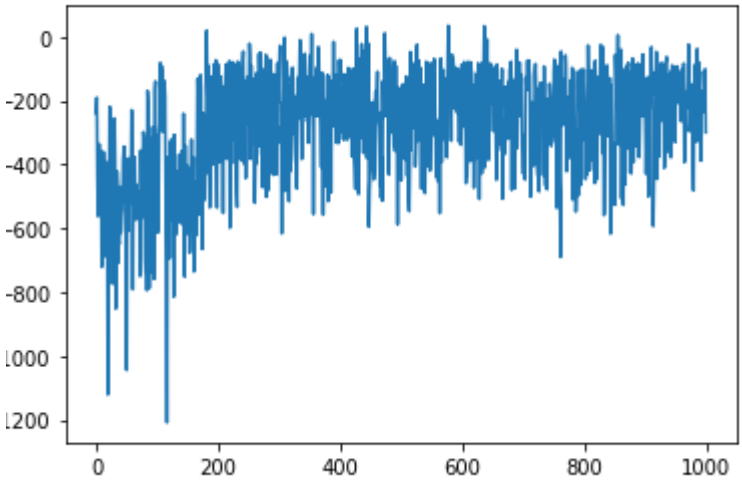
```
f train(self):

    rewards = []
    for episode in range(self.episodes):

        state = self.env.reset()
        if episode%100 == 0:
            print(f'episode: {episode}')

        episode_rewards = 0
        while True:
            a = self.select_action(state)
            next_state, reward, done, _ = self.env.step(a)
            episode_rewards += reward

            self.memory.add_trial(state, a, reward, next_state, done)

            if len(self.memory.buffer) > self.batch_size:
                self.batch_train()

            state = next_state
              self.env.render()
            if done:
                break

        rewards.append(episode_rewards)
        self.epsilon *= 0.99

    return rewards
```

```
f batch_train(self):

  batch = self.memory.sample(self.batch_size)
  states = torch.tensor([b['state'] for b in batch])
  actions = torch.tensor([b['action'] for b in batch])
  rewards = torch.tensor([b['reward'] for b in batch])
  next_states = torch.tensor([b['next_state'] for b in batch])
  dones = torch.tensor([1 if b['done'] else 0 for b in batch])

  self.optim.zero_grad()

  x = self.dqn(states)
  x = torch.tensor([x[a] for x, a in zip(x, actions)])
  y = rewards + torch.mul((self.discount*self.dqn(next_states).max(1).values.unsqueeze(1)), 1 - dones)

  loss = self.loss_func(x, y)
  loss.backward()
  self.optim.step()
```

شکل ۷- پیاده‌سازی تابع batch_train

نتایج

نتیجه‌ی آموزش این مدل در اثر تعامل با محیط و نمونه‌برداری از مشاهدات، منجر به مشاهده‌ی پاداش‌های زیر شده است.



شکل ۸- نمودار پاداش دریافتی در اپیزودهای مختلف در روند یادگیری

همانطور که مشاهده می‌شود، پاداش دریافتی توسط Agent به‌تدریج در حال افزایش است.

در مورد اثر سایز بافر بر روی نتیجه، هرچه بافر ما سایز کوچک‌تری داشته‌باشد به این معناست که ما دیتای با تنوع کمتری برای آموزش مدل داریم. در نتیجه، به تدریج مدل ما بر روی آن overfit خواهد کرد. هرچه سایز بافر ما بزرگ‌تر باشد، تنوع بیشتری از مشاهدات را پوشش خواهیم داد و همچنین برای یادگیری با توجه به این مشاهدات، باید زمان بیشتری هم برای آموزش مدل صرف کنیم.

روند اجرای کد پیاده‌سازی

فایل lunar_lander.py به علت مشکلات ایمپورت از کتابخانه‌ی gym و همچنین pyglet_rendering.py برای رندر کردن ترایال‌ها، باید در کنار فایل جوپیتر باشند. همچنین فیلمی از روند یادگیری این ایجنت در پوشه‌ی مربوطه پیوست شده است.