به نام خدا

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده مهندسی کامپیوتر

مبانی و کاربردهای هوش مصنوعی ترم پاییز ۱۴۰۰

پروژه سوم

مهلت تحویل 1 بهمن ۱۴۰۰

نام و نام خانوادگی : محمد باقر شریفی

شماره دانشجویی : ۹۸۳۱۱۲۷

بخش اول (تکرار ارزش) :

در این بخش با K مرحله تخمین مقادیر بهینه ، Vk را محاسبه می کنیم . برای این منظور باید تابع زیر K مرحله برای تمامی state ها به جز terminal ها اجرا شود .

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

در این بخش می بایست سه متد زیر را پیاده سازی می کردیم که در ادامه به شرح هر کدام می پردازیم .

runValueIteration : در این متد باید بخش مربوط به ماکسیمم گیری QValue و محاسبه نهایی عبارت بالا را برای تمام state ها در هر iteration انجام می دادیم . همچنین باید Value مربوط به terminal ها را برابر پاداش مربوط به خروج از آن ها می گذاشتیم .

همینطور که در تصویر زیر مشاهده می کنید تمامی موارد انجام شده است .

```python
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    for i in range(self.iterations):
        iteration_values = util.Counter()
        for state in self.mdp.getStates():
            if not self.mdp.isTerminal(state):
                actions = self.mdp.getPossibleActions(state)
                max_value = max([self.computeQValueFromValues(state, action) for action in actions])
                iteration_values[state] = max_value
            else:
                self.values[state] = self.mdp.getReward(state, 'exit', '')
        self.values = iteration_values
```

computeQValueFromValues : در این متد باید QValue را برای هر state و action مشخص محاسبه می کردیم و این کار هم با کمک عبارت مربوط به محاسبه QValue انجام گرفته است . پیاده سازی این متد به صورت زیر صورت گرفته است .

```python
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    transition_states_and_probs = self.mdp.getTransitionStatesAndProbs(state, action)
    value = 0
    for next, T in transition_states_and_probs:
        stateTransitionReward = self.mdp.getReward(state, action, next)
        value += T * (stateTransitionReward + self.discount*self.values[next])

    return value
```

computeActionFromValues : در این متد باید با توجه به QValue های هر state بهینه ترین action را انتخاب می کردیم . این کار با ماکسیمم گیری بین QValue ها انجام گرفته است . پیاده سازی این متد به صورت زیر صورت گرفته است .

```python
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    stateAction = util.Counter()
    for action in self.mdp.getPossibleActions(state):
        stateAction[action] = self.computeQValueFromValues(state,action)
    return stateAction.argMax()
```

بخش دوم (تجزیه و تحلیل عبور از پل) :

در این بخش باید با تغییر یکی از دو پارامتر نویز و تخفیف کاری کنیم که عامل تکرار از پل عبور کند .

برای این منظور نویز را برابر صفر قرار می دهیم تا هنگام حرکت به هیچ حالت ناخواسته ای نرسیم و بتوانیم از پل عبور کنیم . تصویر کد مربوط به این بخش را می توانید مشاهده کنید .

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0
    return answerDiscount, answerNoise
```

بخش سوم (سیاست ها) :

این سوال پنج قسمت مختلف دارد که در هر قسمت باید با مشخص کردن تخفیف ، نویز و پاداش زندگی خواسته مسئله را پاسخ دهیم .

قسمت اول : در این قسمت باید به ترمینال نزدیک برسد و مسیرش را از کنار صخره ها انتخاب کند . برای این منظور باید پاداش زندگی را به قدری کم در نظر بگیریم که به انتخاب ترمینال دور تر یا مسیر دور تر تمایلی نداشته باشد و همچنین باید نویز به گونه ای انتخاب شود که انقدر کم نباشد که عامل ریسک کند و تا ترمینال دور تر از کنار صخره ها حرکت کند . در نهایت هر کدام را به صورت زیر انتخاب کردم که توسط autograder قابل قبول بود .

```
def question3a():
    answerDiscount = 0.9
    answerNoise = 0.3
    answerLivingReward = -1.5
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

قسمت دوم : در این قسمت باید به ترمینال نزدیک برسد اما مسیرش را از کنار صخره ها انتخاب نکند . برای این منظور باید پاداش زندگی بیشتر باشد تا با مسیر طولانی مشکلی نداشته باشد و از طرفی باید تخفیف کاهش یابد تا به تمایلی به ترمینال با ارزش بیشتر نداشته باشد . نویز هم همچنان باید قابل توجه باشد تا تمایلی به حرکت از کنار صخره ها نداشته باشد . در نهایت هر کدام را به صورت زیر انتخاب کردم که توسط autograder قابل قبول بود .

```
def question3b():
    answerDiscount = 0.5
    answerNoise = 0.3
    answerLivingReward = -0.5
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

قسمت سوم : در این قسمت باید به ترمینال دور تر برسد و مسیرش را از کنار صخره ها انتخاب کند . برای این منظور باید نویز را کم کنیم تا ریسک حرکت از کنار صخره ها کم شود و تخفیف باید افزایش یابد تا پاداش ترمینال دور تر اهمیت پیدا کند و پاداش زندگی باید کمتر شود که مسیر دور تر را انتخاب نکند . در نهایت هر کدام را به صورت زیر انتخاب کردم که توسط autograder قابل قبول بود .

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -0.8
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

قسمت چهارم : در این قسمت باید به ترمینال دور تر برسد اما مسیرش را از کنار صخره ها انتخاب نکند . برای این منظور باید نویز افزایش یابد تا ریسک حرکت از کنار صخره را نپذیرد همچنین پاداش زندگی باید افزایش یابد تا مشکلی با حرکت در مسیر طولانی نداشته باشد . تخفیف هم همچنان باید زیاد باشد تا ارزش ترمینال دور تر اهمیت پیدا کند . در نهایت هر کدام را به صورت زیر انتخاب کردم که توسط autograder قابل قبول بود .

```
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.3
    answerLivingReward = -0.1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

قسمت پنجم : در این قسمت باید از دو خروجی و صخره اجتناب کند و هرگز متوقف نشود . برای این منظور تخفیف را صفر می کنیم تا ارزش ترمینال ها اهمیتی پیدا نکند و همچنین پاداش زندگی را بسیار افزایش می دهیم . در نهایت هر کدام را به صورت زیر انتخاب کردم که توسط autograder قابل قبول بود .

```
def question3e():
    answerDiscount = 0
    answerNoise = 0.5
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

بخش چهارم (تکرار ارزش ناهمزمان) :

در این بخش ما در هر تکرار ، فقط یک حالت را بروزرسانی می کنیم و نحوه عملکرد تکرار ارزش چرخه ای است . در این بخش می بایست تنها متد runValueIteration پیاده سازی می کردیم که پیاده سازی آن در تصویر زیر آمده است . این بخش هم مشابه بخش اول است با این تفاوت که در هر iteration یک state را

به روز می کنیم و به سراغ بعدی می رویم و پس از اتمام دوباره از ابتدا شروع می کنیم .

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    states = self.mdp.getStates()
    for i in range(self.iterations):
        if not self.mdp.isTerminal(states[i % len(states)]):
            actions = self.mdp.getPossibleActions(states[i % len(states)])
            max_value = max([self.computeQValueFromValues(states[i % len(states)], action) for action in actions])
            self.values[states[i % len(states)]] = max_value
        else:
            self.values[states[i % len(states)]] = self.mdp.getReward(states[i % len(states)], 'exit', '')
```

بخش پنجم (تکرار ارزش اولویت بندی شده) :

در این بخش الگورتیم مشخصی را باید پیاده سازی می کردیم که هدف آن این است که به روزرسانی های مقادیر حالت را به سمتی متمرکز کند که احتمالا سیاست ها را تغییر دهد .

در این بخش به پیاده سازی هر کدام از گام های الگوریتم داخل متد runValueIteration پرداختیم .

در اولین گام به تعریف ساختمان داده های مورد نیاز و در ادامه به مشخص کردن همه پسین های هر حالت پرداختیم . تصویر کد مربوط به آن را در زیر مشاهده می کنید . برای این کار هر کدام از state ها را در نظر می گیریم و به دنبال سایر state هایی می گردیم که امکان انتقال به state اول را دارند و همه را در یک مجموعه جمع می کنیم .

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    priority_queue = util.PriorityQueue()
    movements = ['north', 'south', 'east', 'west']
    predecessors = {}
    states = self.mdp.getStates()

    for state_1 in states:
        self.values[state_1] = 0
        pred_set = set()
        if not self.mdp.isTerminal(state_1):
            for state_2 in states:
                if not self.mdp.isTerminal(state_2):
                    for move in movements:
                        if move in self.mdp.getPossibleActions(state_2):
                            for next, T in self.mdp.getTransitionStatesAndProbs(state_2, move):
                                if (next == state_1) and (T > 0):
                                    pred_set.add(state_2)
        predecessors[state_1] = pred_set
```

سپس در گام بعد برای هر حالت غیر پایانی ، قدر مطلق تفاضل بین مقدار فعلی آن حالت و بیشترین

QValue مربوط به آن حالت را حساب می کنیم و آن را diff می نامیم و سپس با الویت diff– به صف الویت اضافه می کنیم تا حالت هایی با بیشترین خطا الویت بیشتری داشته باشند . تصویر کد مربوط به آن را در زیر مشاهده می کنید .

```
for state in states:
    if not self.mdp.isTerminal(state):
        value = self.values[state]
        max_value = max([self.getQValue(state, action) for action in self.mdp.getPossibleActions(state)])
        diff = abs(value - max_value)
        priority_queue.push(state, -diff)
```

در گام بعد به انجام به روز رسانی تا زمانی که صف الویت خالی شود و یا iteration ها تمام شود ادامه می دهیم . در هر iteration یک حالت را از صف الویت خارج می کنیم در صورتی که ترمینال نبود ارزش آن را به روز می کنیم و برای تمام پسین های آن مقدار diff را دوباره به دست می آوریم و درصورتی که از teta بیشتر بود مقدار آن را در صف الویت به روز رسانی می کنیم . تصویر کد مربوط به آن را در زیر مشاهده می کنید .

```
for i in range(self.iterations):
    if priority_queue.isEmpty():
        return
    state = priority_queue.pop()
    self.values[state] = max([self.getQValue(state, action) for action in self.mdp.getPossibleActions(state)])
    for pred in predecessors[state]:
        max_ = max([self.getQValue(pred, action) for action in self.mdp.getPossibleActions(pred)])
        diff = abs(self.values[pred] - max_)
        if diff > self.theta:
            priority_queue.update(pred, -diff)
```

هر سه تکه کد بالا در متد runValueIteration قرار دارند .

بخش ششم (یادگیری Q) :

در این بخش باید constructor و متد های getQValue ، computeValueFromQValues ، computeActionFromQValues و update پیاده سازی می کردیم .

در constructor با استفاده از کلاس util.Counter ارزش ها را تعریف می کنیم .

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    self.values = util.Counter()
```

در متد getQValue با دریافت state و action ارزش مورد نظر آن را بر می گردانیم .

```
def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    return self.values[(state,action)]
```

در متد computeValueFromQValues از بین action های موجود بیشترین QValue را باز می گردانیم .

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    legal_actions = self.getLegalActions(state)
    if len(legal_actions) != 0:
      return max([self.getQValue(state,action) for action in legal_actions])
    return 0
```

در متد computeActionFromQValues چک می کنیم در بین action های موجود کدام ها بیشترین QValue را دارند و از بین آن ها یکی را به صورت رندوم باز می گردانیم .

```
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"
    legal_actions = self.getLegalActions(state)
    stateMaxQValue = self.computeValueFromQValues(state)
    if len(legal_actions) != 0:
      max_actions = []
      for action in legal_actions :
        if self.getQValue(state, action) == stateMaxQValue :
          max_actions.append(action)
      return random.choice(max_actions)

    return None
```

در متد update با داشتن state ، action ، nextState و reward ، QValue فعلی را و ارزش state بعدی را محاسبه می کنیم . سپس با قرار دادن این داده ها به همراه alpha در عبارت مربوطه می توانیم QValue جدید را حساب کنیم . پیاده سازی این متد به صورت زیر صورت گرفته است .

```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
     # temporal update of q-values
    q_value = self.getQValue(state,action)
    next_q_alue = self.computeValueFromQValues(nextState)
    value = q_value + self.alpha*(reward + self.discount*(next_q_alue)- q_value)
    self.values[(state,action)] = value
```

بخش هفتم (epsilon حریصانه) :

در این بخش باید متد getAction را پیاده سازی می کردیم . این متد به این صورت پیاده سازی شده است که با دریافت یک state ، action های موجود را مشخص می کند و در صورتی که action ای موجود باشد ، با نسبت (epsilon) به (یک منهای epsilon) action رندوم انجام می دهد و در سایر مواقع action بهینه را از computeActionFromQValues دریافت می کند و انجام می دهد .

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    if len(legalActions) != 0:
      if util.flipCoin(self.epsilon) == True:
        action = random.choice(legalActions)
      else:
        action = self.computeActionFromQValues(state)
    return action
```

با انجام این بخش نمره بخش ۹ هم به دریافت شد و این بخش نیازی به پیاده سازی ندارد .

بخش هشتم (بررسی دوباره عبور از پل) :

در این بخش باید episilon و LearningRate را مشخص کنیم به طوری که بیش از ۹۹ درصد مواقع سیاست بهینه بعد از ۵۰ بار تکرار یاد گرفته شود ، اما این کار ممکن نیست و ۵۰ اپیزود برای این یاد گیری بسیار کم است و به اپیزود های بیشتری نیاز است .

```
def question8():
    answerEpsilon = None
    answerLearningRate = None
    #return answerEpsilon, answerLearningRate
    return 'NOT POSSIBLE'
```

بخش نهم :

این بخش به پیاده سازی ای نیاز نداشت و با انجام بخش ۷ این بخش نیز انجام شد .

در نهایت نمره کامل ۹ بخش اول دریافت شد و می توانید خروجی autograder را در زیر مشاهده بفرمایید .

```
Finished at 21:28:52

Provisional grades
==================
Question q1: 4/4
Question q2: 1/1
Question q3: 5/5
Question q4: 1/1
Question q5: 3/3
Question q6: 4/4
Question q7: 2/2
Question q8: 1/1
Question q9: 1/1
Question q10: 0/3
------------------
Total: 22/25
```