باسمه تعالی

پروژه ۳ درس هوش مصنوعی – دکترهوش مصنوعی

سید امیرمهدی میرشریفی – ۹۸۳۱۱۰۵

۱- تکرار ارزش

در کلاس valueIterationAgent چندین تابع تعریف شده است که هر کدام توضیح داده میشود:

```python
def __init__(self, mdp, discount = 0.9, iterations = 100):
    """...
    self.mdp = mdp
    self.discount = discount
    self.iterations = iterations
    self.values = util.Counter() # A Counter is a dict with default 0

    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    same_values=0
    while self.iterations > 0 and same_values!= len(mdp.getStates()):
        same_values=0
        state_Values = self.values.copy()
        allStates = mdp.getStates()
        for state in allStates:
            allActions_State = mdp.getPossibleActions(state)
            QValues = []
            for action in allActions_State:
                finalStates = mdp.getTransitionStatesAndProbs(state, action)
                Average_value = 0
                for finalState in finalStates:
                    nextState,probability = finalState
                    Average_value += (probability * (mdp.getReward(state, action, nextState) + (discount * state_Values[nextState])))
                QValues.append(Average_value)
            if len(QValues) != 0:
                if max(QValues)== state_Values[state]: same_values+=1
                self.values[state] = max(QValues)
        self.iterations -= 1
```

در این تابع که تابع سازنده شی مورد نظر از کلاس هست ، در ابتدا فیلد های شی را تعریف میکنیم. سپس با تعداد دوری که در ورودی جهت تکرار ارزش گرفته ایم ، به همان تعداد ابتدا برای تمام حالت ها اکشن و احتمال آنها را میگیریم ، سپس ماکسیمم Qstate رو برایشان حساب میکنیم و ماکسیمم Q را به عنوان ارزش آن حالت قرار میدهیم. در ابتدای هر دور چک میکنیم اگر ارزش ها همگرا شده بودند دست از چک کردن بر میداریم. مکانیزم چک کردن همگرایی از طریق فیلد same_value شکل میگیرد که برای هر حالت اگر ارزش جدید با ارزش قبلی برابر باشد یکی به آن اضافه میکند . اگر در ابتدای دور بعد مقدار این متغیر برابر تعداد حالت ها باشد یعنی همگرا شده است . اگر نه که مجدد مقدار ان را صفر خواهیم کرد و یک دور جدید را آغاز میکنیم.

این بخش در سوال ۴ تغییری جزئی خواهد کرد.

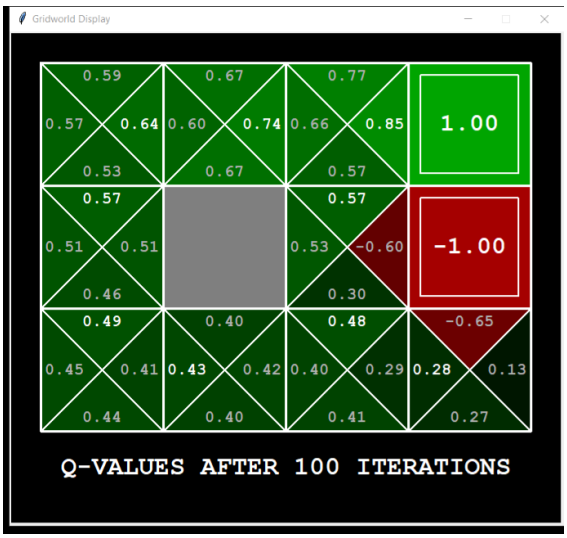
تابع بعدی تابعی است که با گرفتن حالت و اکشن مقدار Q متناسب با آن را حساب خواهد کرد. که این کار با فیلد هایی که در تابع آغازین مانند فیلد ارزش وضعیت ها ، پاداش ها و ... انجام میگیرد.

```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 ***"
    next_state = self.mdp.getTransitionStatesAndProbs(state, action)
    Qvalue=0
    for new_state in next_state:
        nextState, probability = new_state
        Qvalue += (probability * (self.mdp.getReward(state, action, nextState) + (self.discount * self.values[nextState])))

    return Qvalue
```

آخرین تابعی که در این بخش پیاده سازی شده است ، تابعی است که با توجه به وضعیتی که به تابع می دهیم حرکت و پالیسی مد نظر را برمیگرداند که این کار از طریق فراخوانی تابع بالا برای اکشن هایی که توان آن را دارد و حساب ماکسیمم آن و بهترین اکشن انجام میشود.

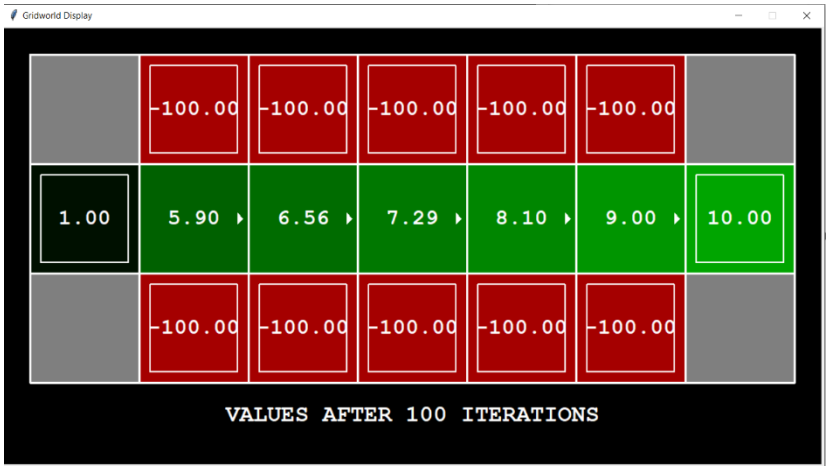
```python
def computeActionFromValues(self, state):
    """..."""
    "*** YOUR CODE HERE ***"
    if self.mdp.isTerminal(state):
        return None
    allActionsForState = self.mdp.getPossibleActions(state)
    optimalAction = ""
    maxSum = float("-inf")
    for action in allActionsForState:
        Qval = self.computeQValueFromValues(state, action)
        if (maxSum == 0.0 and action == "") or Qval >= maxSum:
            optimalAction = action
            maxSum = Qval

    return optimalAction
```

تصاویر نیز حاصل و نتیجه کد هستند:



با توجه به این تصویر:

همانطورکه مشاهده میکنید چون حرکت ما نویز دارد بنابراین هر چه در وسط پل هستیم احتمال افتادن ما در آتیش ها خیلی بیشتر میشود و هرچه به لبه ها نزدیک تر هستیم این احتمال کاهش میابد. بنابراین با تنها تغییر یک عامل و آن هم نویز به صفر میتوان به نتیجه زیر دست پیدا کرد:



همچنین کد مربوط به این بخش:

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

۳) سیاست ها

- خروجی نزدیک را ترجیح دهد و ریسک صخره را بپذیرد:

در این بخش ما می بایست هزینه زندگی را بالا در نظر بگیریم تا سراغ حالت پر ارزش نرود ، همچنین نویز را بسیار کم در نظر بگیریم تا ریسک صخره را بپذیرد.

```
def question3a():
    answerDiscount = 0.5
    answerNoise = 0.05
    answerLivingReward = -5.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

- خروجی نزدیک را ترجیح دهد و از صخره دوری کند:

در این مرحله همچنان باید هزینه زندگی بالا باشد اما نه به بالایی سری قبل تا باعث شود ریسک صخره را با وجود نویز بالا بپذیرد.

```
def question3b():
    answerDiscount = 0.5
    answerNoise = 0.3
    answerLivingReward = -2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

- خروجی دور را ترجیح دهد و ریسک صخره را بپذیرد:

در این مرحله نویز و هزینه پایین را پایین و discount را بالا در نظر میگیریم

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.05
    answerLivingReward = -1.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

- خروجی دور را بپذیرد ولی از صخره دوری کند

در این مرحله نویز را بالا و هزینه زندگی را پایین در نظر میگیریم تا مسیر دور را انتخاب کند و بخاطر نویز از صخره دوری کند

```
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.3
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

- از هردو صخره اجتناب کند

برای این مرحله تنها کاری که لازم است این است که هزینه زندگی را یک عدد مثبت و بیشتر از پر ارزش ترین پاداش در نظر بگیریم .

```
def question3e():
    answerDiscount = 0.9
    answerNoise = 0.15
    answerLivingReward = 15
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

گزارش: آیا استفاده از الگوریتم تکرار ارزش تحت هر شرایطی به همگرایی می انجامد؟

بله . این الگوریتم تضمین میکند که همیشه به حالت بهینه همگرا شود.

۴) تکرار ارزش ناهمزمان

در آخر بخش یک بیان شد که بخشی از کد تابع سازنده کلاس valueIterationAgent تغییر خواهد کرد.

این تغییر در این بخش با انتقال کد valueIteraion به تابع runValueIteraion است که با توجه به این که در کلاس AsynchronousValueIterationAgent هم نیاز به کد valueIteraion اما به مدلی دیگر است ، و همچنین از کلاس valueIteraionAgent ارث بری می کند بنابراین کد هر کدام را در تابع runValueIteraion پیاده سازی کردیم که برای کلاس Asynchronous یک بار بازنویسی یا override صورت خواهد گرفت . تابع runValueIteraion برای کلاس AsynchtonousValueIteraionAgent است. برای کلاس valueIterationAgent نیز همان محتوایی است که در سوال یک آمده است.همچنین تابع runValueIteration در تابع سازنده صدا زده میشوند و متناسب با نوع کلاس عملیات تکرار ارزش انجام میشود.

```
def runValueIteration(self):
    mdpStates = self.mdp.getStates()
    indexIterator = 0

    while self.iterations > 0:
        self.iterations -= 1
        if indexIterator == len(mdpStates): indexIterator = 0
        targetState = mdpStates[indexIterator]
        indexIterator += 1
        if self.mdp.isTerminal(targetState):
            continue
        bestAction = self.computeActionFromValues(targetState)
        QValue = self.computeQValueFromValues(targetState,
                                              bestAction)
        self.values[targetState] = QValue
```

در این تابع همانطور که خواسته شده اگر حالت ما ترمینال باشد آن را رد میکند . اگر تمام حالت ها چک بشوند به وسیله فیلد indexIterator مجددا از حالت اولیه شروع میشود.

سوال: روش‌های بروزرسانی‌ای که در بخش اول (بروزرسانی با استفاده از batch) و در این بخش (بروزرسانی به صورت تکی) پیاده کرده‌اید را با یکدیگر مقایسه کنید. (یک نکته مثبت و یک نکته منفی برای هرکدام)

زمانی که ما در هر دور تمام حالات را بررسی میکنیم نیاز به یک آرایه داریم که حالات را درون آن کپی میکنیم اما زمانی که تکی این کار را انجام بدهیم نیاز به این کار نیست. این نکته مثبت. اما نکته منفی آن است که عملا تعداد باری که ما چک کردن را برای تمام حالات انجام میدهیم برابر است با تعداد داده شده تقسیم بر تعداد حالات که در این صورت اگر همگرایی در تکرار بالا به دست بیاید احتمالا در این حالت همگرا نشود.

۵)تکرار ارزش اولویت بندی شده

در این بخش همانند مراحل گفته شده در دستورکار ، ابتدا یک صف جهت تعریف اولویت حالت ها تعریف میکنیم که یک صف اولویت از نوع مین هیپ است. همچنین یک کانتر برای تمام حالت هایی که ترمینال نباشند تشکیل میشود تا حالت های پسینشان در آن نگه داری بشود که در ابتدا برای هر حالت یک مجموعه تهی است.

```
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    self.queue = util.PriorityQueue()
    self.predecessors = util.Counter()

    for s in self.mdp.getStates():
        if not self.mdp.isTerminal(s):
            self.predecessors[s] = set()

    for s in self.mdp.getStates():
        if self.mdp.isTerminal(s):
            continue
```

سپس برای هر حالت ابتدا حالت های پسینشان و در ادامه اولویت آن ها را محاسبه و در صف اولویت پوش میکنیم.

```
    # compute predecessors for state s
    possibleActions = self.mdp.getPossibleActions(s)
    for action in possibleActions:
        nextTransitions = self.mdp.getTransitionStatesAndProbs(s, action)
        for nextState, prob in nextTransitions:
            if prob != 0 and not self.mdp.isTerminal(nextState):
                self.predecessors[nextState].add(s)

    # calculate priority and push into queue
    currentValue = self.values[s]
    bestAction = self.computeActionFromValues(s)
    highestQValue = self.computeQValueFromValues(s, bestAction)
    diff = abs(currentValue - highestQValue)
    self.queue.push(s, -diff)
```

و ادامه کد هم که همان مراحل پیاده سازی گزارشکار:

```
for iter in range(0, self.iterations):
    if self.queue.isEmpty():
        # terminate
        return

    s = self.queue.pop()

    # calculate Q-value for updating s
    bestAction = self.computeActionFromValues(s)
    self.values[s] = self.computeQValueFromValues(s, bestAction)

    for p in self.predecessors[s]:
        currentValue = self.values[p]
        bestAction = self.computeActionFromValues(p)
        highestQValue = self.computeQValueFromValues(p, bestAction)
        diff = abs(currentValue - highestQValue)
        if diff > self.theta:
            self.queue.update(p, -diff)
```

۶) یادگیری

بیشتر توابعی که در ادامه بیان میشوند پیش تر تعریف و توضیح داده شده بودند اما مجددا توضیح آن بیان خواهد شد:

تابع سازنده: در این بخش یک کانتر برای Qvalue ها تعریف میکنیم تا هنگام ساخت شی ساخته بشود.

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

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

تابع getQvalue : در این تابع به ازای حالت و اکشن کیو ولیو مخصوص آن را بر میگردانیم

```
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.QValues[(state, action)]
```

تابع computeValueFromQValues : در این تابع با حالت ورودی ، اکشن بهینه را دریافت و سپس به واسطه آن مقدار کیو ولیو آن را به عنوان ارزش حالت ورودی برمیگردانیم

```
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 ***"
    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return 0.0
    optimalAction = self.getPolicy(state)
    return self.getQValue(state, optimalAction)
```

تابع computeActionFromQValue : در این تابع اکشن بهینه برای حالت ورودی انتخاب میشود.

```
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 ***"
    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return None
    actionVals = {}
    optimalQValue = float('-inf')
    for action in actions:
        QValue = self.getQValue(state, action)
        actionVals[action] = QValue
        if QValue > optimalQValue:
            optimalQValue = QValue
    optimalActions = [a for a,v in actionVals.items() if v == optimalQValue]

    return random.choice(optimalActions)
```

تابع update : همانطور که از نامش پیداست و بر اساس فرمولی که برای یادگیری داشتیم مقدار را متناسب با اکتشافی بودن آن و اهمیت حالات جدید و قدیم به روزرسانی می کند.

```
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 ***"
    oldValue = self.getQValue(state,action)
    futureQValue = self.getValue(nextState)
    newQValue = oldValue + self.alpha * \
                (reward + (self.discount * futureQValue) - oldValue)
    self.QValues[(state, action)] = newQValue
```

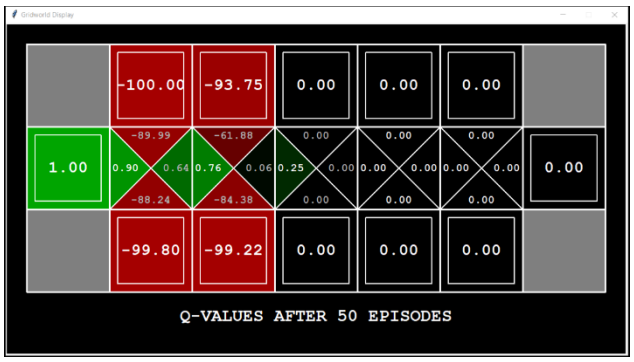
۷) اپسیلون حریصانه

تابع getAction: با گرفتن حالت ، اگر صفر باشد همان اکشن بهینه را استفاده خواهد کرد که این حالت ، حالت بهره برداری و اگر اپسیلون صفر نباشد وارد فاز اکتشافی میشود و اکشن رندوم دریافت خواهد کرد.

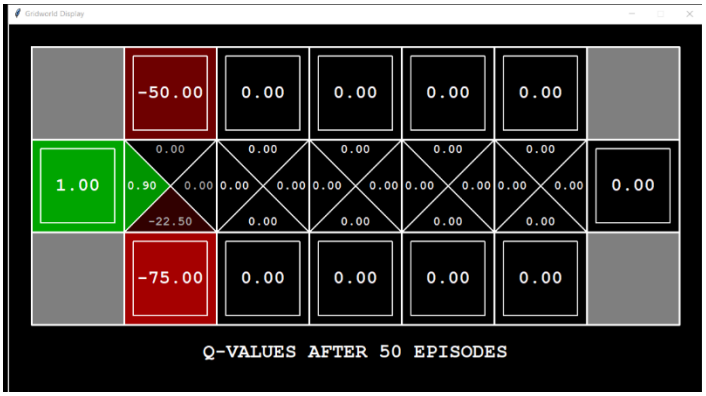
```
def getAction(self, state):
    """...
    # Pick Action
    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return None
    action = None
    if not util.flipCoin(self.epsilon):
        # exploit
        action = self.getPolicy(state)
    else:
        # explore
        action = random.choice(actions)
    return action
```

۸)بررسی دوباره عبور از پل

نتیجه یادگیری بدون نویز و ۵۰ آموزش بدین شکل میشود:



و زمانی که اپسیلون را صفر میکنیم:



همانطور که ملاحظه میشود کم و زیاد کردن اپسلون روند کشف حالات جدید را تغییر میدهد بدین معنی که اپسیلون صفر یعنی هیچ گونه اکتشافی نباشد و اپسیلون یک یعنی دائم در حال اکتشاف.

۹) پکمن

چون در تابع getAction و computeActionFromValue این که حالات دیده نشده هم در نظر گرفته بشوند برای اکشن اعمال شد در این بخش نیاز به کار خاصی نبود و نمره کامل دریافت میشود.

۱۰) یادگیری تقریبی

کد این بخش به آسانی و در دو تابع پیاده سازی میشود که به آن می پردازیم:

تابع getQvalue : در این تابع بر اساس وزن ها و فیچر ها کیو ولیو محاسبه میشود:

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    feats = self.featExtractor.getFeatures(state, action)
    w = self.getWeights()
    return w * feats
```

تابع update : در این بخش هم طبق فرمول مربوطه وزن ها به روز رسانی خواهند شد

```
def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    feats = self.featExtractor.getFeatures(state, action)
    diff = reward + self.discount * self.computeValueFromQValues(nextState) - self.getQValue(state, action)
    for feat in feats:
        self.weights[feat] += self.alpha * diff * feats[feat]
```

در آخر ممنونم از شما تدریسیار گرامی بابت وقتی که گذاشتید و پوزش بابت کم و کاستی ها احتمالی متاثر از کمبود وقت