CS 234 Winter 2019: Assignment #2

Due date: 2/06 (Wed) 11:59 PM (23:59) PST

These questions require thought, but do not require long answers. Please be as concise as possible.

We encourage students to discuss in groups for assignments. We ask that you abide by the university Honor Code and that of the Computer Science department. If you have discussed the problems with others, please include a statement saying who you discussed problems with. Failure to follow these instructions will be reported to the Office of Community Standards. We reserve the right to run a fraud-detection software on your code. Please refer to website, Academic Collaboration and Misconduct section for details about collaboration policy.
Please review any additional instructions posted on the assignment page. When you are ready to submit, please follow the instructions on the course website. **Make sure you test your code using the provided commands and do not edit outside of the marked areas.**

You'll need to download the starter code and fill the appropriate functions following the instructions from the handout and the code's documentation. Training DeepMind's network on `Pong` takes roughly **12 hours on GPU**, so **please start early**! (Only a completed run will recieve full credit) We will give you access to an Azure GPU cluster. You'll find the setup instructions on the course assignment page.

Introduction

In this assignment we will implement deep Q learning, following DeepMind's paper ([**mnih2015human**] and [**mnih-atari-2013**]) that learns to play Atari from raw pixels. The purpose is to understand the effectiveness of deep neural network as well as some of the techniques used in practice to stabilize training and achieve better performance. You'll also have to get comfortable with Tensorflow. We will train our networks on the `Pong-v0` environment from OpenAI gym, but the code can easily be applied to any other environment.

In Pong, one player wins if the ball passes by the other player. Winning a game gives a reward of 1, while losing gives a negative reward of -1. An episode is over when one of the two players reaches 21 wins. Thus, the final score is between -21 (lost episode) or +21 (won episode). Our agent plays against a decent hard-coded AI player. Average human performance is -3 (reported in [**mnih-atari-2013**]). If you go to the end of the homework successfully, you will train an AI agent with super-human performance, reaching at least +10 (hopefully more!).

1 Test Environment (5 pts)

Before running our code on Pong, it is crucial to test our code on a test environment. You should be able to run your models on CPU in no more than a few minutes on the following environment:

- 4 states: $0, 1, 2, 3$
- 5 actions: $0, 1, 2, 3, 4$. Action $0 \leq i \leq 3$ goes to state i, while action 4 makes the agent stay in the same state.
- Rewards: Going to state i from states 0, 1, and 3 gives a reward $R(i)$, where $R(0) = 0.1, R(1) = -0.2, R(2) = 0, R(3) = -0.1$. If we start in state 2, then the rewards defind above are multiplied by -10. See Table 1 for the full transition and reward structure.

- One episode lasts 5 time steps (for a total of 5 actions) and always starts in state 0 (no rewards at the initial state).

State (s)	Action (a)	Next State (s')	Reward (R)
0	0	0	0.1
0	1	1	-0.2
0	2	2	0.0
0	3	3	-0.1
0	4	0	0.1
1	0	0	0.1
1	1	1	-0.2
1	2	2	0.0
1	3	3	-0.1
1	4	1	-0.2
2	0	0	-1.0
2	1	1	2.0
2	2	2	0.0
2	3	3	1.0
2	4	2	0.0
3	0	0	0.1
3	1	1	-0.2
3	2	2	0.0
3	3	3	-0.1
3	4	3	-0.1

Table 1: Transition table for the Test Environment

An example of a path (or an episode) in the test environment is shown in Figure 1, and the trajectory can be represented in terms of s_t, a_t, R_t as: $s_0 = 0, a_0 = 1, R_0 = -0.2, s_1 = 1, a_1 = 2, R_1 = 0, s_2 = 2, a_2 = 4, R_2 = 0, s_3 = 2, a_3 = 3, R_3 = (-0.1) * (-10) = 1, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

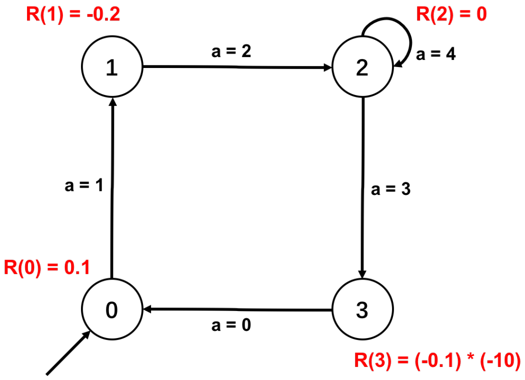


Figure 1: Example of a path in the Test Environment

1. (**written** 5pts) What is the maximum sum of rewards that can be achieved in a single episode in the test environment, assuming $\gamma = 1$?

Solution: The optimal reward of the Test environment is

$$4.1$$

To prove this, let's prove an **upper bound** of 4.1 with 3 key observations

- first, the maximum reward we can achieve is 2 when we do 2 → 1.
- second, after having performed this optimal transition, we have to wait at least one step to execute it again.

As we have 5 steps, we can execute 2 optimal moves. Executing less than 2 would yield a strictly smaller result. We need to go to 2 twice, which gives us 0 reward on 2 steps. Thus, we know that 4 steps gives us a max of 4. Then, the best reward we can achieve that is not an optimal move (starting from state 1) is 0.1, which yields an upper bound of 4.1.
Considering the path $0 \to 2 \to 1 \to 2 \to 1 \to 0$ proves that we can **achieve** this upper bound.

2 Q-learning (12 pts)

Tabular setting In the *tabular setting*, we maintain a table $Q(s, a)$ for each tuple state-action. Given an experience sample (s, a, r, s'), our update rule is

$$Q(s,a) = Q(s,a) + \alpha \left(r + \gamma \max_{a' \in A} Q\left(s', a'\right) - Q\left(s, a\right) \right), \tag{1}$$

where $\alpha \in \mathbb{R}$ is the learning rate, γ the discount factor.

Approximation setting Due to the scale of Atari environments, we cannot reasonably learn and store a Q value for each state-action tuple. We will instead represent our Q values as a function $\hat{q}(s, a, \mathbf{w})$ where $\mathbf{w}$ are parameters of the function (typically a neural network's weights and bias parameters). In this *approximation setting*, our update rule becomes

$$\mathbf{w} = \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in A} \hat{q}\left(s', a', \mathbf{w}\right) - \hat{q}\left(s, a, \mathbf{w}\right) \right) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}). \tag{2}$$

In other words, we are try to minimize

$$L(\mathbf{w}) = \mathbf{E}_{s,a,r,s'} \left[r + \gamma \max_{a' \in A} \hat{q}\left(s', a', \mathbf{w}\right) - \hat{q}(s, a, \mathbf{w}) \right]^2 \tag{3}$$

Target Network DeepMind's paper [**mnih2015human**] [**mnih-atari-2013**] maintains two sets of parameters, $\mathbf{w}$ (to compute $\hat{q}(s, a)$) and $\mathbf{w}^-$ (target network, to compute $\hat{q}(s', a')$) such that our update rule becomes

$$\mathbf{w} = \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in A} \hat{q}\left(s', a', \mathbf{w}^-\right) - \hat{q}\left(s, a, \mathbf{w}\right) \right) \nabla_{\mathbf{w}} \hat{q}(s, a, \mathbf{w}). \tag{4}$$

The target network's parameters are updated with the Q-network's parameters occasionally and are kept fixed between individual updates. Note that when computing the update, we don't compute gradients with respect to $\mathbf{w}^-$ (these are considered fixed weights).

Replay Memory As we play, we store our transitions (s, a, r, s') in a buffer. Old examples are deleted as we store new transitions. To update our parameters, we *sample* a minibatch from the buffer and perform a stochastic gradient descent update.

ϵ-Greedy Exploration Strategy During training, we use an ϵ-greedy strategy. DeepMind's paper [**mnih2015human**] [**mnih-atari-2013**] decreases ϵ from 1 to 0.1 during the first million steps. At test time, the agent choses a random action with probability $\epsilon_{soft} = 0.05$.

There are several things to be noted:

- In this assignment, we will update $\mathbf{w}$ every `learning_freq` steps by using a `minibatch` of experiences sampled from the replay buffer.
- DeepMind's deep Q network takes as input the state s and outputs a vector of size = number of actions. In the `Pong` environment, we have 6 actions, thus $\hat{q}(s, \mathbf{w}) \in \mathbb{R}^6$.
- The input of the deep Q network is the concatenation 4 consecutive steps, which results in an input after preprocessing of shape $(80 \times 80 \times 4)$.

We will now examine these assumptions and implement the epsilon-greedy strategy.

1. (**written** 3pts) What is one benefit of using experience replay?

   Solution: Deep Q learning suffers from stability issues:

   (a) strongly correlated samples break the iid assumption necessary to assume convergence of SGD methods, that are based on estimation of gradients.

   (b) we forget rare experiences from which the agent could learn more

   (c) non-stationary distribution of samples, as samples are drawn from the experience of an evolving agent whose policy can quickly change, which can be problematic for deep learning methods.

   Experience replay answers

   (a) by sampling independent samples,

   (b) by keeping a large history

   (c) by keeping examples from all the agent's policies, acting as a regularization of the policy over time.

   Note that experience replay, by having samples from different policies, forces us to use an off-policy algorithm like Q-learning.

2. (**written** 3pts) What is one benefit of the target network?

   Solution: The **target network**

   (a) avoids the oscillation of the target policy (issue c.)

   (b) breaks correlation with the Q-network, that would create a lot of variance in the loss estimation (oscillation).

3. (**written** 3pts) What is one benefit of representing the Q function as $\hat{q}(s, \mathbf{w}) \in \mathbb{R}^K$

   Solution: As we have finite number of actions K, it is computationally more efficient to compute a vector $\in \mathbb{R}^K$ from the state than compute K scalars separately.

4. (**coding** 3pts) Implement the `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running `python q1_schedule.py`.

3 Linear Approximation (26 pts)

1. (**written** 3pts) Show that Equations (1) and (2) from section 2 above are exactly the same when $\hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T x(s, a)$, where $\mathbf{w} \in \mathbb{R}^{|S||A|}$ and $x : S \times A \rightarrow \mathbb{R}^{|S||A|}$ such that

$$x(s,a)_{s',a'} = \begin{cases} 1 & \text{if } s' = s, a' = a \\ 0 & \text{otherwise} \end{cases}$$

for all $(s,a) \in S \times A$, $x(s,a)$ is a vector of length $|S||A|$ where the element corresponding to $s' \in S, a' \in A$ is 1 when $s' = s, a' = a$ and is 0 otherwise.

Solution:

Let's denote $\mathbf{w}_{s,a}$ the component of $\mathbf{w}$ that corresponds to the entry equal to 1 in $x(s,a)$. Let's write (2) for $\mathbf{w}_{s,a}$:

$$\mathbf{w}_{s,a} = \mathbf{w}_{s,a} + \alpha(r + \gamma \max_{a' \in \mathcal{A}} \mathbf{w}_{s',a'} - \mathbf{w}_{s,a}) \nabla_{\mathbf{w}_{s,a}} \mathbf{w}_{s,a}$$

And we obtain (1) for $\mathbf{w}_{s,a}$ that we can identify to $Q(s,a)$.

2. (**written** 3pts) Derive the gradient with regard to the value function parameter $\mathbf{w} \in \mathbb{R}^n$ given $\hat{q}(s,a,\mathbf{w}) = \mathbf{w}^T x(s,a)$ for any function $x(s,a) \mapsto x \in \mathbb{R}^n$ and write the update rule for $\mathbf{w}$.

   Solution: We have

$$\begin{aligned} \nabla_{\mathbf{w}} \hat{q}(s,a,\mathbf{w}) &= \nabla_{\mathbf{w}} \mathbf{w}^T x(s,a) \\ &= x(s,a) \end{aligned}$$

   And the update rule becomes

$$\mathbf{w} = \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in A} \hat{q}\left(s', a', \mathbf{w}\right) - \hat{q}\left(s, a, \mathbf{w}\right) \right) x(s,a) \tag{5}$$

3. (**coding** 15pts) We will now implement linear approximation in Tensorflow. This question will setup the whole pipeline for the remiander of the assignment. You'll need to implement the following functions in `q2_linear.py` (pleasd read through`q2_linear.py`) :

   - `add_placeholders_op`
   - `get_q_values_op`
   - `add_update_target_op`
   - `add_loss_op`
   - `add_optimizer_op`

   Test your code by running `python q2_linear.py` **locally on CPU**. This will run linear approximation with Tensorflow on the test environment. Running this implementation should only take a minute or two.

4. (**written** 5pts) Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q2_linear` to your writeup.

   Solution: Yes, 4.1. See Figure 2.

4 Implementing DeepMind's DQN (15 pts)

1. (**coding** 10pts) Implement the deep Q-network as described in [**mnih2015human**] by implementing `get_q_ values_op` in `q3_nature.py`. The rest of the code inherits from what you wrote for linear approximation. Test your implementation **locally on CPU** on the test environment by running `python q3_nature.py`. Running this implementation should only take a minute or two.

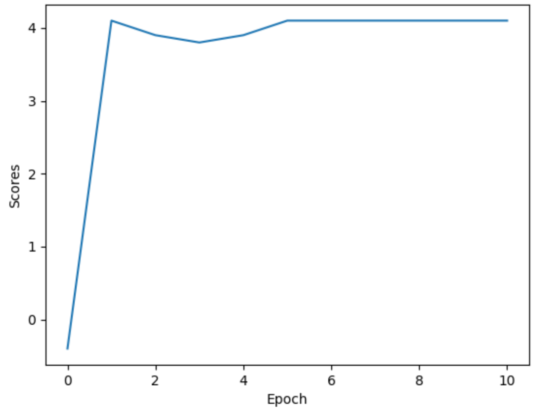


Figure 2: Reference plot for linear approximation with Tensorflow on the Test Env

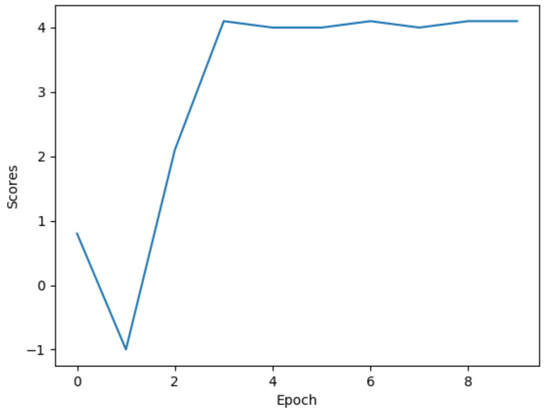


Figure 3: Reference plot for the nature network with Tensorflow on the Test Env

2. (**written** 5pts) Attach the plot of scores, `scores.png`, from the directory `results/q3_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?

   Solution:

   See Figure 3. We manage to get the same performance. We also observe that the training time for the nature paper is much slower (roughly 61s against 16s). Also, we do 10 times less updates for the nature paper, which sums up to a ratio of $10 \times 4 = 40$!

   This points out the necessity of finding the right complexity of model given a problem.

   Note: The comparison may not be fair in this case as both experiments use different configs and environments. Try changing q2's config and environment to match q3's and you should get a similar conclusion.

5 DQN on Atari (27 pts)

The Atari environment from OpenAI gym returns observations (or original frames) of size $(210 \times 160 \times 3)$, the last dimension corresponds to the RGB channels filled with values between 0 and 255 (`uint8`). Following DeepMind's paper [**mnih2015human**], we will apply some preprocessing to the observations:

- Single frame encoding: To encode a single frame, we take the maximum value for each pixel color value over the frame being encoded and the previous frame. In other words, we return a pixel-wise max-pooling of the last 2 observations.
- Dimensionality reduction: Convert the encoded frame to grey scale, and rescale it to $(80 \times 80 \times 1)$. (See Figure 4)

The above preprocessing is applied to the 4 most recent observations and these encoded frames are stacked together to produce the input (of shape $(80 \times 80 \times 4)$) to the Q-function. Also, for each time we decide on an action, we perform that action for 4 time steps. This reduces the frequency of decisions without impacting the performance too much and enables us to play 4 times as many games while training. You can refer to the *Methods Section* of [**mnih2015human**] for more details.

(a) Original input $(210 \times 160 \times 3)$ with RGB colors (b) After preprocessing in grey scale of shape $(80 \times 80 \times 1)$

Figure 4: `Pong-v0` environment

1. (**written** 2pts) Why do we use the last 4 time steps as input to the network for playing Atari games?
   Solution:

   In Pong, if we only have one frame, we cannot determine the velocity nor the direction of the ball, which are essential to play correctly. Thus, to take a good decision, we need different time steps so that we can evaluate the trajectory of the ball.

   More generally, we combine multiple steps so that we transform our world dynamics into a (almost) *markovian* world. The choice of 4 is arbitrary and could be adapted in other problems.

2. (**written** 5pts) What's the number of parameters of the DQN model (for `Pong`) if the input to the Q-network is a tensor of shape $(80, 80, 4)$ and we use "SAME" padding? How many parameters are required for the linear Q-network, assuming the input is still of shape $(80, 80, 4)$? How do the number of parameters compare between the two models?

   Solution:

   The number of parameters by layer is

   - $8 \times 8 \times 4 \times 32 + 32$ for the first convolution
   - $4 \times 4 \times 32 \times 64 + 64$ for the second convolution
   - $3 \times 3 \times 64 \times 64 + 64$ for the third convolution
   - $6,400 \times 512 + 512$ for the first fully connected layer (with SAME padding)
   - $512 \times 6 + 6$ for the last layer (for Pong)

The total is

$$3,358,374$$

(notice that the convolution only accounts for $77,984$ parameters!)

For linear approximation, the number of parameters is

$$25,600 * 6 + 6 = 153,606$$

Which is roughly 20 times less!

3. (**coding and written** 5pts). Now, we're ready to train on the Atari `Pong-v0` environment. First, launch linear approximation on pong with `python q4_train_atari_linear.py` **on Azure's GPU**. This will train the model for 500,000 steps and should take approximately an hour. What do you notice about the performance?

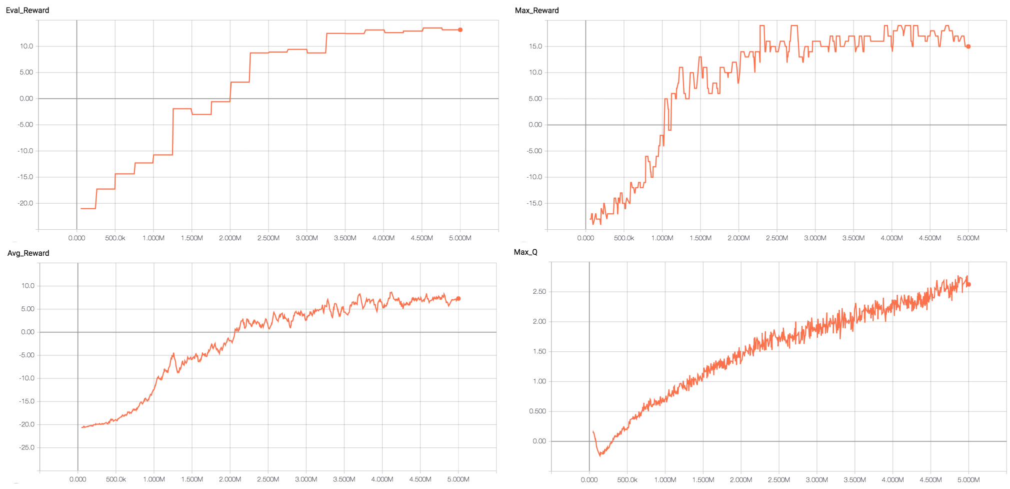
   Solution:

   After a few epochs, we notice that the score doesn't really progress, around -19.6. Linear approximation is not powerful enough for Pong.

4. (**coding and written** 10 pts). In this question, we'll train the agent with DeepMind's architecture on the Atari `Pong-v0` environment. Run `python q5_train_atari_nature.py` **on Azure's GPU**. This will train the model for 5 million steps and should take around **12 hours**. Attach the plot `scores.png` from the directory `results/q5_train_atari_nature` to your writeup. You should get a score of around 13-15 after 5 million time steps. As stated previously, the Deepmind paper claims average human performance is -3.

   As the training time is roughly 12 hours, you may want to check after a few epochs that your network is making progress. The following are some training tips:

   - If you terminate your terminal session, the training will stop. In order to avoid this, you should use `screen` to run your training in the background.
   - The evaluation score printed on terminal should start at -21 and increase.
   - The max of the q values should also be increasing
   - The standard deviation of q shouldn't be too small. Otherwise it means that all states have similar q values
   - You may want to use Tensorboard to track the history of the printed metrics. You can monitor your training with Tensorboard by typing the command `tensorboard --logdir=results` and then connecting to `ip-of-you-machine:6006`. Below are our Tensorboard graphs from one training session:



5. (**written** 5pts) Compare the performance of the DeepMind architecure with the linear Q-network approximation. How can you explain the gap in performance?

Solution:

We reach around 15 with the nature network, compared to -19 with the linear approximation. This gap in performance is explained by the insufficient complexity of the linear approximation model, and the exceptional efficiency of the convolutional networks to analyse the image without having too much parameters.

6 Real world RL with neural networks (10 pts)

Given a stream of batches of n environment interactions (s_i, a_i, r_i, s'_i) we want to learn the optimal value function using a neural network. The underlying MDP has a finite sized action space.

1. (**written** 4pts) Your friend first suggests the following approach

   (a) Initialize parameters ϕ of neural network V_ϕ

   (b) For each batch of k tuples (s_i, a_i, r_i, s'_i) do Stochastic Gradient Descent with loss function $\sum_{i=0}^{k} |y_i - V_\phi(s_i)|^2$ where $y_i = \max_{a_i}[r_i + \gamma V_\phi(s'_i)]$

   What is the problem with this approach? (Hint: Think about the type of data we have)

   Solution: We cannot take a max over actions as done in the above algorithm because we are using a batch of (s, a, r, s') tuples and it is possible for a particular state our dataset does not have datapoints corresponding to all actions.

   Note: This algorithm is really a natural extension of Value Iteration to function approximation land. However, unlike Value Iteration in which we assume a known transition model, it is now impossible to perform this calculation. In order to take max over actions, we either need to know the transition model or have a simulator where we can reset back to the original state and try all different actions.

   Also, note that in comparison with Value Iteration, we have used $V_\phi(s'_i)$ as a sample of $\mathbb{E}_{s'}[V_\phi(s')]$. This is also done in TD learning and is not a problem. In general, we can draw samples to approximate an expectation. However, we cannot draw samples to approximate a max. Hence, the key takeaway is that we can sample to approximate expectation but not max.

2. (**written** 3pts) Your friend now suggests the following

(a) Initialize parameters ϕ of neural network for state-action value function $Q_\phi(s, a)$

(b) For each batch of k tuples (s_i, a_i, r_i, s'_i) do Stochastic Gradient Descent with loss function $\sum_{i=0}^{k} |y_i - Q_\phi(s_i, a_i)|^2$ where $y_i = r_i + \gamma V(s'_i)$

Now as we just have the network $Q_\phi(s, a)$ how would you determine $V(s)$ needed for the above training procedure?

Solution: We can simply take the maximum over all possible actions of $Q(s, a)$.

Note: Notice that the only difference between this question and the previous one is that we learn Q instead of V. Because we learn Q, the max over actions can be taken over Q which we know how to do by just forward propagating our network using different actions. This is also why we have Q-learning and not V-learning when we do not have access to the transition model.

3. (**written** 3pts) Is the above method of learning the Q network guaranteed to give us an approximation of the optimal state action value function?

Solution: No it is not guaranteed. We do not know anything about the size of n and the total number of environment interactions. Also the quality of the network we train depends a lot on the data being representative of the entire state, action space. Also function approximation due to the nature of SGD and the ever moving targets associated with the above procedure may never converge to an optimal function.

Note: To avoid ambiguity, this question should be phrased as: "Is the above method of learning the Q network guaranteed to converge to the optimal state action value function?". Note that even if we assume an oracle that can find the global minimum of any loss function given any dataset and an infinite amount of data generated from a policy, we are still not guaranteed to converge. Even if we further assume that the true Q* lies in the space of functions that we can learn, we are still not guaranteed to converge. In particular, if the data is generated by a deterministic policy, we will never get all (s,a) pairs for all s and all a. Without all (s,a) pairs, we will never be able to learn Q*. However, due to the ambiguity of the question, we accept a variety of answers.