CS 234 Winter 2020: Assignment #2

Due date: February 5, 2020 at 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 games from raw pixels. The purpose is to demonstrate the effectiveness of deep neural networks as well as some of the techniques used in practice to stabilize training and achieve better performance. In the process, you'll become familiar 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 scores if the ball passes by the other player. An episode is over when one of the players reaches 21 points. Thus, the total return of an episode is between -21 (lost every point) and $+21$ (won every point). Our agent plays against a decent hard-coded AI player. Average human performance is -3 (reported in [**mnih-atari-2013**]). In this assignment, you will train an AI agent with super-human performance, reaching at least $+10$ (hopefully more!).

0 Test Environment (6 pts)

Before running our code on Pong, it is crucial to test our code on a test environment. In this problem, you will reason about optimality in the provided test environment by hand; later, to sanity-check your code, you will verify that your implementation is able to achieve this optimality. 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 trajectory (or episode) in the test environment is shown in Figure 5, and the trajectory can be represented in terms of s_t, a_t, R_t as: $s_0 = 0, a_0 = 1, R_0 = -0.2, s_1 = 1, a_1 = 2, R_1 = 0, s_2 = 2, a_2 = 4, R_2 = 0, s_3 = 2, a_3 = 3, R_3 = (-0.1) \cdot (-10) = 1, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

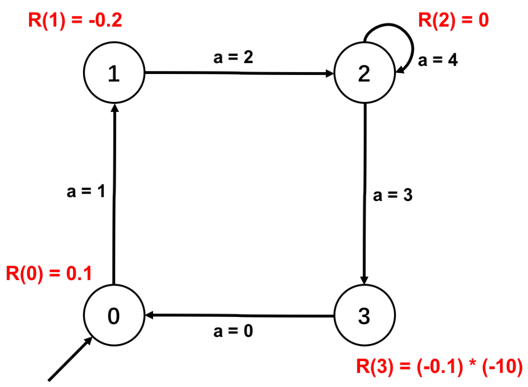


Figure 1: Example of a trajectory in the Test Environment

1. (**written** 6 pts) What is the maximum sum of rewards that can be achieved in a single trajectory in the test environment, assuming $\gamma = 1$? Show first that this value is attainable in a single trajectory, and then briefly argue why no other trajectory can achieve greater cumulative reward.

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 $\underbrace{2 \to 1}$. R(1) = -10 × (-0.2) = 2
- 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.

R(2)=0 R(2)=0 R(0) = 0.1

Considering the path $0 \to 2 \to 1 \to 2 \to 1 \to 0$ proves that we can **achieve** this upper bound.

↓ initial R(0) = 0 R(1)=2 R(1)= 2

1 Q-Learning (24 pts)

Tabular setting If the state and action spaces are sufficiently small, we can simply maintain a table containing the value of $Q(s, a)$ – an estimate of $Q^*(s, a)$ – for every (s, a) pair. In this *tabular setting*, given an experience sample (s, a, r, s'), the update rule is

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

where $\alpha > 0$ is the learning rate, $\gamma \in [0, 1)$ 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*, the update rule becomes

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

In other words, we aim to minimize

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

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

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

and the corresponding optimization objective becomes

$$L^-(\mathbf{w}) = \underset{s,a,r,s' \sim \mathcal{D}}{\mathbb{E}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}^-\right) - \hat{q}(s, a; \mathbf{w}) \right)^2 \right] \tag{5}$$

The target network's parameters are updated to match the Q-network's parameters every C training iterations, and are kept fixed between individual training updates.

Replay Memory As we play, we store our transitions (s, a, r, s') in a buffer $\mathcal{D}$. 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 For exploration, we use an ϵ-greedy strategy. This means that with probability ϵ, an action is chosen uniformly at random from $\mathcal{A}$, and with probability $1-\epsilon$, the greedy action (i.e., $\arg\max_{a \in \mathcal{A}} \hat{q}(s, a, \mathbf{w})$) is chosen. DeepMind's paper [**mnih2015human**] [**mnih-atari-2013**] linearly anneals ϵ from 1 to 0.1 during the first million steps. At test time, the agent chooses a random action with probability $\epsilon_{\text{soft}} = 0.05$.

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 $|\mathcal{A}|$. In the `Pong` environment, we have $|\mathcal{A}| = 6$ actions, so $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^6$.
- 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 ϵ-greedy strategy.

1. (**written** 3 pts) What is one benefit of representing the Q function as $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$?

   Solution: In order to make a Q-Learning update, we must identify the action which maximizes $\hat{q}$ at the state s'. If we parameterized the DQN to take both a state and action as input, we would need to execute $O(|\mathcal{A}|)$ forward-passes (one per legal action) in order to compute this argmax. By instead parameterizing the DQN such that it takes only a state as input and outputs the state-action values for all legal actions in that state simultaneously, we need only execute a single forward-pass per Q-Learning update.

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

We will now investigate some of the theoretical considerations involved in the tuning of the hyperparameter C which determines the frequency with which the target network weights $\mathbf{w}^-$ are updated to match the Q-network weights $\mathbf{w}$. On one extreme, the target network could be updated *every* time the Q-network is updated; it's straightforward to check that this reduces to not using a target network at all. On the other extreme, the target network could remain fixed throughout the entirety of training.

Furthermore, recall that stochastic gradient descent minimizes an objective of the form $J(\mathbf{w}) = \mathbb{E}_{x\sim\mathcal{D}}[l(x, \mathbf{w})]$ by making sample updates of the following form:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla_{\mathbf{w}} l(x, \mathbf{w})$$

Stochastic gradient descent has many desirable theoretical properties; in particular, under mild assumptions, it is known to converge to a local optimum. In the following questions we will explore the conditions under which Q-Learning constitutes a stochastic gradient descent update.

3. (**written** 5 pts) Consider the first of these two extremes: standard Q-Learning without a target network, whose weight update is given by equation **(2)** above. Is this weight update an instance of stochastic gradient descent (up to a constant factor of 2) on the objective $L(\mathbf{w})$ given by equation **(3)**? Argue mathematically why or why not.

   Solution: No. For notational simplicity, write $L(\mathbf{w}) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[l(s, a, r, s', \mathbf{w})]$. Differentiating l with respect to $\mathbf{w}$, we see that:

   $$\nabla_{\mathbf{w}} l(s, a, r, s', \mathbf{w}) = 2(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w}))\nabla_{\mathbf{w}}(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w}))$$

   Because the term $\gamma \max_{a'} \hat{q}(s', a'; \mathbf{w})$ depends on $\mathbf{w}$, it follows that:

   $$\nabla_{\mathbf{w}}(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s, a; \mathbf{w})) \neq -\nabla_{\mathbf{w}}\hat{q}(s, a; \mathbf{w})$$

   and thus the weight update above is not of the form $\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla_{\mathbf{w}} l(s, a, r, s', \mathbf{w})$.

4. (**written** 5 pts) Now consider the second of these two extremes: using a target network that is never updated (i.e. held fixed throughout training). In this case, the weight update is given by equation **(4)** above, treating $\mathbf{w}^-$ as a constant. Is this weight update an instance of stochastic gradient descent (up to a constant factor of 2) on the objective $L^-(\mathbf{w})$ given by equation **(5)**? Argue mathematically why or why not.

   Solution: Yes. Again for notational simplicity, write $L^-(\mathbf{w}) = \mathbb{E}_{s,a,r,s'\sim\mathcal{D}}[l^-(s, a, r, s', \mathbf{w})]$. Differentiating l^- with respect to $\mathbf{w}$, we see that:

   $$\nabla_{\mathbf{w}} l^-(s, a, r, s', \mathbf{w}) = 2(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}^-) - \hat{q}(s, a; \mathbf{w}))\nabla_{\mathbf{w}}(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}^-) - \hat{q}(s, a; \mathbf{w}))$$

   $$= -2(r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}^-) - \hat{q}(s, a; \mathbf{w}))\nabla_{\mathbf{w}}\hat{q}(s, a; \mathbf{w})$$

   where crucially we used the fact that $r + \gamma \max_{a'} \hat{q}(s', a'; \mathbf{w}^-)$ doesn't depend on $\mathbf{w}$. Thus the weight update above is indeed of the form $\mathbf{w} \leftarrow \mathbf{w} - \alpha\nabla_{\mathbf{w}} l^-(s, a, r, s', \mathbf{w})$.

5. (**written** 3 pts) An obvious downside to holding the target network fixed throughout training is that it depends on us knowing good weights for the target network *a priori*; but if this was the case, we wouldn't need to be training a Q-network at all! In light of this, together with the discussion above regarding the convergence of stochastic gradient descent and your answers to the previous two parts, describe the fundamental tradeoff at play in determining a good choice of C.

Solution: The more frequently we update the target network, the better it will reflect our most up-to-date knowledge about how to best act in the environment, and thus the more useful it will be as a source of target values. However, more-frequent target network updates pulls the training process closer to that of standard Q-Learning, which as we saw in part 3 is an imperfect approximation to the principled schema of gradient descent. Intuitively, more-frequent target network updates yields a stronger dependence of $\mathbf{w}^-$ on $\mathbf{w}$, which in turn more-strongly violates the assumption $\nabla_{\mathbf{w}}(r_t + \gamma \max_a \hat{q}(s_{t+1}, a; \mathbf{w}^-)) = 0$ which is required for gradient descent. On the other hand, less-frequent target network updates lead to less-informative training targets, but result in the Q-network weight update steps more-closely following the true gradient direction.

6. (**written**, 5 pts) In supervised learning, the goal is typically to minimize a predictive model's error on data sampled from some distribution. If we are solving a regression problem with a one-dimensional output, and we use mean-squared error to evaluate performance, the objective writes

$$L(\mathbf{w}) = \mathop{\mathbb{E}}_{(\mathbf{x},y)\sim\mathcal{D}}[(y - f(\mathbf{x};\mathbf{w}))^2]$$

Difference between RL and supervised learning.

where $\mathbf{x}$ is the input, y is the output to be predicted from $\mathbf{x}$, $\mathcal{D}$ is a dataset of samples from the (unknown) joint distribution of $\mathbf{x}$ and y, and $f(\cdot;\mathbf{w})$ is a predictive model parameterized by $\mathbf{w}$.

This objective looks very similar to the DQN objective stated above. How are these two scenarios different? Hint: how does this dataset $\mathcal{D}$ differ from the replay buffer $\mathcal{D}$ used above?

Solution: A key distinction is that in supervised learning, the data distribution is fixed. In RL, the policy is changing as learning progresses, so the distribution of interest is changing during learning as well. In addition, the target (i.e. label) in supervised learning is known, whereas the target in policy evaluation (Q^π) is unknown and must be estimated.

2 Linear Approximation (24 pts)

(a) (**written**, 3 pts) Show that Equations (1) and (2) from problem 1 above are exactly the same when $\hat{q}(s,a;\mathbf{w}) = \mathbf{w}^\top\delta(s,a)$, where $\mathbf{w} \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ and $\delta : \mathcal{S}\times\mathcal{A} \to \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ with

Eq (1) : $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a))$

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

$w^T\delta(s,a) = w^T\delta(s,a) + \alpha(r + \gamma \max_{a'} w^T\delta(s',a') - w^T\delta(s,a))$, $\delta(s,a) = 1$

Solution:

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

Eq (2) :
$$\mathbf{w}_{s,a} = \mathbf{w}_{s,a} + \alpha(r + \gamma \max_{a'\in\mathcal{A}} \mathbf{w}_{s',a'} - \mathbf{w}_{s,a})\boxed{\nabla_{\mathbf{w}_{s,a}}\mathbf{w}_{s,a}} = \delta(s,a) = 1$$

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

(b) (**written**, 3 pts) Assuming $\hat{q}(s,a;\mathbf{w})$ takes the form specified in the previous part, compute $\nabla_{\mathbf{w}}\hat{q}(s,a;\mathbf{w})$ and write the update rule for $\mathbf{w}$.

Solution: We have

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

And the update rule becomes

$$\mathbf{w} = \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in A} \hat{q}\left(s', a', \mathbf{w}\right) - \hat{q}\left(s, a, \mathbf{w}\right) \right) \delta(s, a) \tag{6}$$

(c) (**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.

(d) (**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.

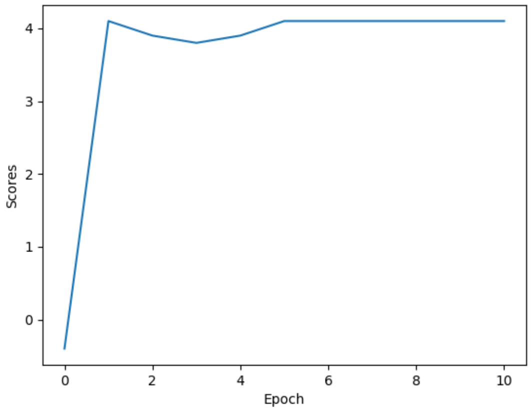


Figure 2: Reference plot for linear approximation with Tensorflow on the Test Env

3 Implementing DeepMind's DQN (13 pts)

(a) (**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.

(b) (**written** 3 pts) 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:

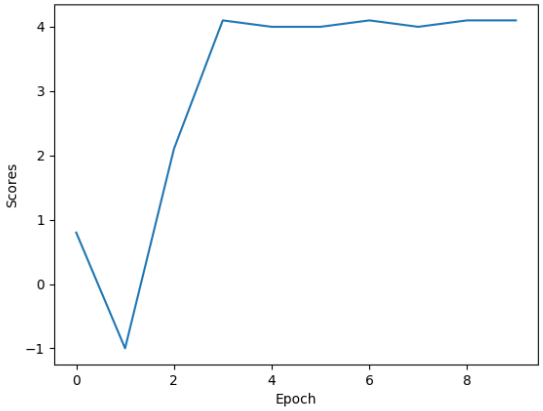


Figure 3: Reference plot for the nature network with Tensorflow on the Test Env

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.

4 DQN on Atari (21 pts)

Reminder: Please remember to kill your VM instances when you are done using them!!

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 × 160 × 3) with RGB colors (b) After preprocessing in grey scale of shape (80 × 80 × 1)

Figure 4: `Pong-v0` environment

(a) (**coding and written**, 5 pts). 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. Briefly qualitatively describe how your agent's performance changes over the course of training. Do you think that training for a larger number of steps would likely yield further improvements in performance? Explain your answer.

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.

(b) (**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.

Solution:

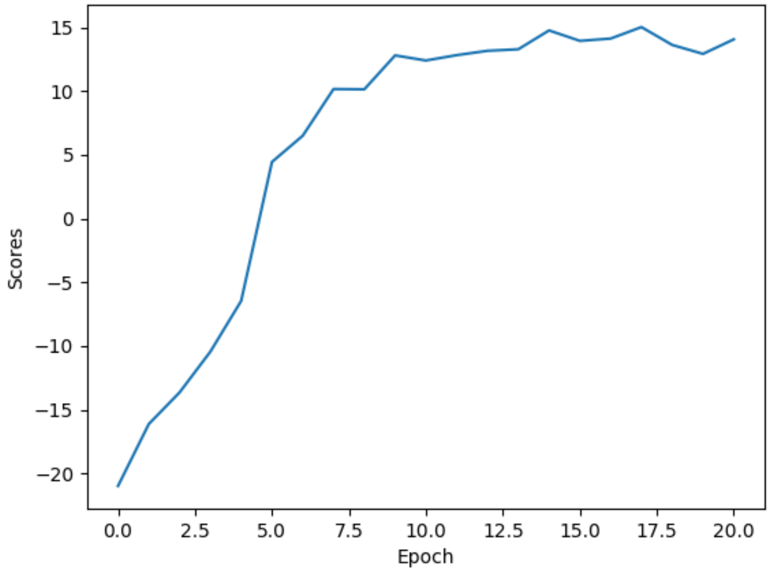
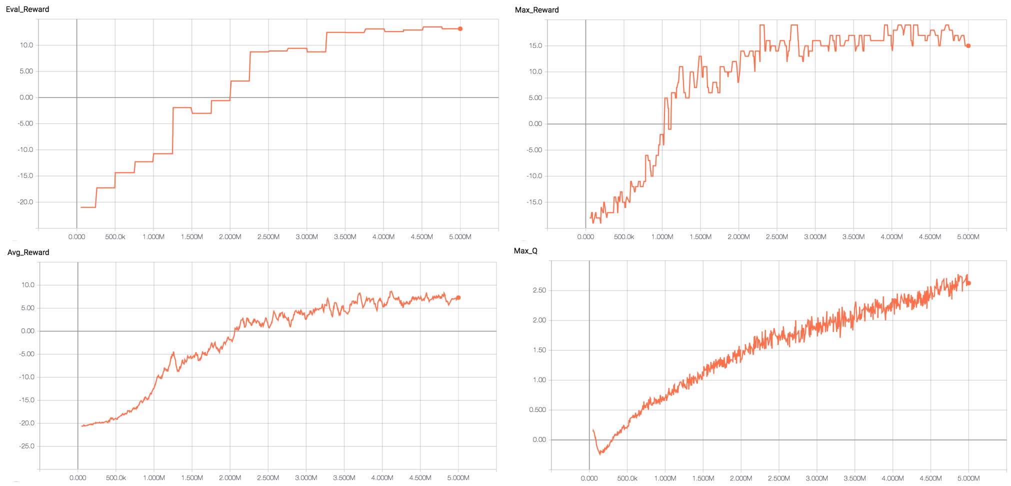


Figure 5: Evolution of the evaluation score on `Pong-v0` for the first 5 million time steps. One epoch corresponds to 250k time steps.

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:



(c) (**written**, 3 pts) In a few sentences, compare the performance of the DeepMind DQN architecture with the linear Q value approximator. 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 analyze the image without having too many parameters.

(d) (**written**, 3 pts) Will the performance of DQN over time always improve monotonically? Why or why not?

Solution: No; monotonic performance improvement is by no means guaranteed. One straightforward way to see this is that due to the random nature of DQN's exploration strategy (i.e. ϵ-greedy), there is no way for the agent to ever avoid poor behavior for certain.

5 n-step Estimators (12 pts)

We can further understand the effects of using a bootstrapping target by adopting a statistical perspective. As seen in class, the Monte Carlo (MC) target is an unbiased estimator[1] of the true state-action

[1] Recall that the bias of an estimator is equal to the difference between the expected value of the estimator and the quantity which it is estimating. An estimator is unbiased if its bias is zero.

value, but it suffers from high variance. On the other hand, temporal difference (TD) targets are biased due to their dependence on the current value estimate, but they have relatively lower variance.

There exists a spectrum of target quantities which bridge MC and TD. Consider a trajectory $s_1, a_1, r_1, s_2, a_2, r_2, \ldots$ obtained by behaving according to some policy π. Given a current estimate $\hat{q}$ of Q^π, let the **n-step SARSA target** (in analogy to the TD target) be defined as:

$$r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{q}(s_{t+n}, a_{t+n})$$

(Recall that the 1-step SARSA target is given by $r_t + \gamma \hat{q}(s_{t+1}, a_{t+1})$).

Given that the n-step SARSA target depends on fewer sample rewards than the MC estimator, it is reasonable to expect it to have lower variance. However, the improved bias of this target over the standard (i.e. 1-step) SARSA target may be less obvious.

(a) (**written**, 12 pts) Prove that for a given policy π in an infinite-horizon MDP, the n-step SARSA target is a less-biased (in absolute value) estimator of the true state-action value function $Q^\pi(s_t, a_t)$ than is the 1-step SARSA target. Assume that $n \geq 2$ and $\gamma < 1$. Further, assume that the current value estimate $\hat{q}$ is uniformly biased across the state-action space (that is, $\text{Bias}(\hat{q}(s,a)) = \text{Bias}(\hat{q}(s',a'))$ for all states $s, s' \in \mathcal{S}$ and all actions $a, a' \in \mathcal{A}$). You need not assume anything about the specific functional form of $\hat{q}$.

Solution: Let b denote the uniform bias value across the state-action space. We will subscript all expectations with the random variables that they are averaging over; all such averaging is done with respect to both the policy π and the stochasticity in the MDP dynamics. Furthermore, as $\hat{q}$ is estimated from data, it is also a random quantity; all expectations also average over this estimation of $\hat{q}$. Our solution will not assume a deterministic reward function, although its fine if yours did. First, let us consider the bias of standard SARSA. We begin by appealing to the definition of bias and the definition of the state-action value function:

$$Bias(r_t + \gamma \hat{q}(s_{t+1}, a_{t+1})) = \mathbb{E}_{r_t, s_{t+1}, a_{t+1}}[r_t + \gamma \hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - Q^\pi(s_t, a_t)$$

$$= \mathbb{E}_{r_t, s_{t+1}, a_{t+1}}[r_t + \gamma \hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - \mathbb{E}_{r_i \forall i \geq t}[\sum_{i=t}^{\infty} \gamma^{i-t} r_i \mid s_t, a_t]$$

We then pull out the $\mathbb{E}_{r_t}[r_t \mid s_t, a_t]$ from both terms, which will cancel:

$$= \mathbb{E}_{r_t}[r_t \mid s_t, a_t] - \mathbb{E}_{r_t}[r_t \mid s_t, a_t] + \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}[\hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - \mathbb{E}_{r_i \forall i \geq t+1}[\sum_{i=t+1}^{\infty} \gamma^{i-t} r_i \mid s_t, a_t]$$

Next, pull out a factor of γ from the summation:

$$= \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}[\hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - \gamma \mathbb{E}_{r_i \forall i \geq t+1}[\sum_{i=t+1}^{\infty} \gamma^{i-t-1} r_i \mid s_t, a_t]$$

E(X)

We now make the observation that the second term looks quite similar to $Q^\pi(s_{t+1}, a_{t+1})$, except that it is not conditioned on s_{t+1}, a_{t+1}. However, the law of total expectation allows us to write:

E(X) = E(E(X|Y))

$$= \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}[\hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}\left(\mathbb{E}_{r_i \forall i \geq t+1}[\sum_{i=t+1}^{\infty} \gamma^{i-t-1} r_i \mid s_{t+1}, a_{t+1}] \mid s_t, a_t\right)$$

E(E(X|Y))

From which we recognize the definition of $Q^\pi(s_{t+1}, a_{t+1})$:

$$= \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}[\hat{q}(s_{t+1}, a_{t+1}) \mid s_t, a_t] - \gamma \mathbb{E}_{s_{t+1}, a_{t+1}}[Q^\pi(s_{t+1}, a_{t+1}) \mid s_t, a_t]$$

We now apply the linearity of expectation to get:

$$= \gamma \mathbb{E}_{s_{t+1},a_{t+1}}\Big(\hat{q}(s_{t+1},a_{t+1}) - Q^{\pi}(s_{t+1},a_{t+1}) \mid s_t, a_t\Big)$$

$$= \gamma \mathbb{E}_{s_{t+1},a_{t+1}}\big[Bias(\hat{q}(s_{t+1},a_{t+1})) \mid s_t, a_t\big]$$

And finally, because we assume that this bias is identical across all possible states s_{t+1} and actions a_{t+1}, we have:

$$= \gamma b$$

Now for the bias of the n-step estimator. We'll be much looser with the notation here, as it closely resembles the previous derivation. You will not be penalized if the argument that you gave looks more similar to the following than the preceding.

$$Bias(r_t + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{q}(s_{t+n}, a_{t+n}))$$

$$= \mathbb{E}[r_t + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{q}(s_{t+n}, a_{t+n}) \mid s_t, a_t] - \mathbb{E}[\boxed{\sum_{i=t}^{\infty}} \gamma^{i-t} r_i \mid s_t, a_t]$$

$$= \underbrace{\mathbb{E}[r_t + \ldots + \gamma^{n-1} r_{t+n-1} \mid s_t, a_t] - \mathbb{E}[r_t + \ldots + \gamma^{n-1} r_{t+n-1} \mid s_t, a_t]}_{= 0} + \gamma^n \mathbb{E}[\hat{q}(s_{t+n}, a_{t+n}) \mid s_t, a_t] - \mathbb{E}[\boxed{\sum_{i=t+n}^{\infty}} \gamma^{i-t} r_i \mid s_t, a_t]$$

$$= \gamma^n \mathbb{E}\Big(\hat{q}(s_{t+n}, a_{t+n}) - \sum_{i=t+n}^{\infty} \gamma^{i-t-n} r_i \mid s_t, a_t\Big)$$

$$= \gamma^n \mathbb{E}\Big(\hat{q}(s_{t+n}, a_{t+n}) - Q^{\pi}(s_{t+n}, a_{t+n}) \mid s_t, a_t\Big)$$

$$= \gamma^n \mathbb{E}\big[Bias(\hat{q}(s_{t+n}, a_{t+n})) \mid s_t, a_t\big]$$

$$= \gamma^n b$$

Because $\gamma < 1$, we have $\gamma^n < \gamma$. As such, we conclude that the n-step estimator has less bias (in an absolute sense) than the 1-step estimator, as desired.