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 ([1] and [2]) 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 [2]). 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 1, and the trajectory can be represented in terms of s_t, a_t, R_t as: $s_0 = 0, a_0 = 1, R_0 = -0.2, s_1 = 1, a_1 = 2, R_1 = 0, s_2 = 2, a_2 = 4, R_2 = 0, s_3 = 2, a_3 = 3, R_3 = (-0.1) \cdot (-10) = 1, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

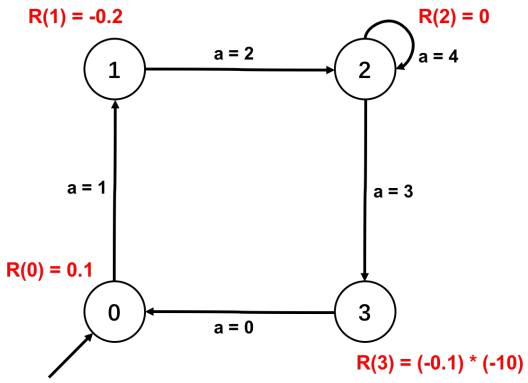


Figure 1: Example of a trajectory in the Test Environment

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.

   We can know from the table that if we move from state 2 and take action 1, then we can have $R = 2$. Therefore, if we can do $(2, 1)$ state-action pair most of time, we can get the rewards as large as possible.

   We have to start from 0, so in the beginning, the trajectory must be $0 \to 2$, in order to execute times of $(2, 1)$ state-action pair as many as possible.

   We are allowed to do 5 actions in one episode starts in 0, so we can at most do twice for $(2, 1)$ state-action pair, if we start from 0.
   Then the trajectory will become $0 \to 2 \to 1 \to 2 \to 1$

   For the last action, we have to move from 1 to 0, in order to get the maximum action rewards of state 1, $R = 0.1$.

   If we execute less than two times of $(2, 1)$ state-action pair, we would get much smaller rewards. Therefore, we have to execute $(2, 1)$ state-action pair twice. In addition, we can also know that the max rewards we can have for 4 time steps is 4, therefore, for 5 time steps, we must have $R = 4.1$, even this is not an optimal policy.

   Therefore, the maximum sum of rewards that can be achieved in a single trajectory will be 4.1 and the trajectory will be $0 \to 2 \to 1 \to 2 \to 1 \to 0$

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

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

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

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

In other words, we aim to minimize

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

Target Network DeepMind's paper [1] [2] 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

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

and the corresponding optimization objective becomes

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

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 [1] [2] 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 $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$?

   Since we have finite number of actions, it is more efficient to compute a vector from the state instead of computing different actions respectively.

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.

   We want to compute gradient of loss function, $l(x, \mathbf{w})$, and check if we can make sample updates of the following form become eq.2:

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

   Equatoin 2,

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

   Therefore, take gradient, $\nabla_{\mathbf{w}}$, of loss function of following equation,

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

   Thus, the loss function is

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

   The gradient $\nabla_{\mathbf{w}}$ of the loss function becomes

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

The update rule becomes,

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

which is not the same as eq.(2).

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.

We want to compute gradient of loss function, $l(x, \mathbf{w})$, of corresponding optimization objective, target network, and check if we can make sample updates of the following form become eq. 4:

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

Equation 4,

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

The corresponding optimization objective of target network,

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

The loss function $l(x, \mathbf{w})$,

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

$\nabla_{\mathbf{w}} l(x, \mathbf{w}) =$

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

The update rule becomes

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

which is the same as eq. (4) by scaling α a constant factor, 2.

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.

The target network helps reduce variance in the loss estimation created by Q-learning and avoids oscillation of the target policy. In terms of choosing C, if we picked a very large C, then the target network's parameters are updated too less times. The parameters of target networks will be obsolete.

If we picked a very small C, then the target network will be close to Q-network without target network, like approximation setting of update rule, eq. (2) and eq. (3), the parameters be updated almost every time. Since it's not an instance of stochastic gradient descent, it might not be able to converge to a local optimum. Therefore, a good choice of C is important when applying target network, it's more data efficient, but it will increase computational cost.

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

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

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?

In supervised learning, we want to break the correlation by separating the data into different datasets, such as training or testing sets. Since the strongly correlated samples somewhat violate the assumption of convergence of stochastic gradient descent. However, applying DQN with randomly distributed or non-stationary samples could be problematic and cause stability issues. Using unkonwn joint distribution $\mathcal{D}$ is a different scenarios compared with using replay buffer $\mathcal{D}$.
In DQN with replay buffer, we are allowed to learn more from previous experiences, however, this might cause stability issue, because of violating assumption of convergence of SGD. We still want to use replay buffer in order to force ourselves to do off-policy learning and having regularization of policy over time.

2 Linear Approximation (24 pts)

1. (**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

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

Substitute $\hat{q}(s,a;\mathbf{w}) = \mathbf{w}^\top\delta(s,a)$ with $\delta(s,a) = 1$ when $s' = s, a' = a$ and $\nabla_{\mathbf{w}}\mathbf{w}(s,a) = 1$,

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

Identify $\mathbf{w}(s,a)$ to $Q(s,a)$, we can get eq. (1),

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

2. (**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 $\mathbf{w}$.

$\nabla_{\mathbf{w}}\hat{q}(s,a;\mathbf{w}) = \nabla_{\mathbf{w}}\mathbf{w}^\top\delta(s,a) = \delta(s,a)$

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

3. (**coding**, 15 pts) We will now implement linear approximation in TensorFlow. This question will set up the pipeline for the remainder of the assignment. You'll need to implement the following functions in `q2_linear.py` (please 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 from Problem 0. Running this implementation should only take a minute or two.

4. (**written**, 3 pts) 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.

   Yes. The optimal reward is 4.1.

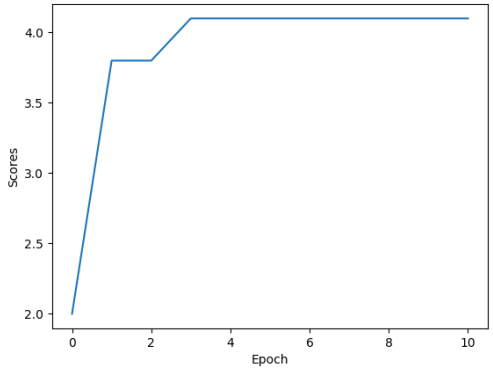


Figure 2: Linear Approximation Test Scores

3 Implementing DeepMind's DQN (13 pts)

1. (**coding**, 10 pts) Implement the deep Q-network as described in [1] 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.

2. (**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?

   The training time of DQN is longer than linear approximation, about 2 to 4 times longer, however, DQN requires less epochs than linear approximation.

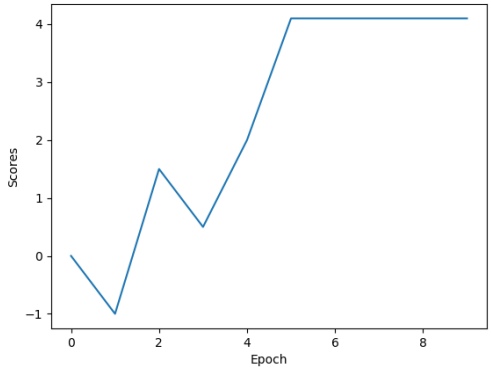


Figure 3: DQN Test Scores

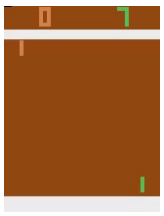
4 DQN on Atari (21 pts)

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 [1], 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 [1] 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

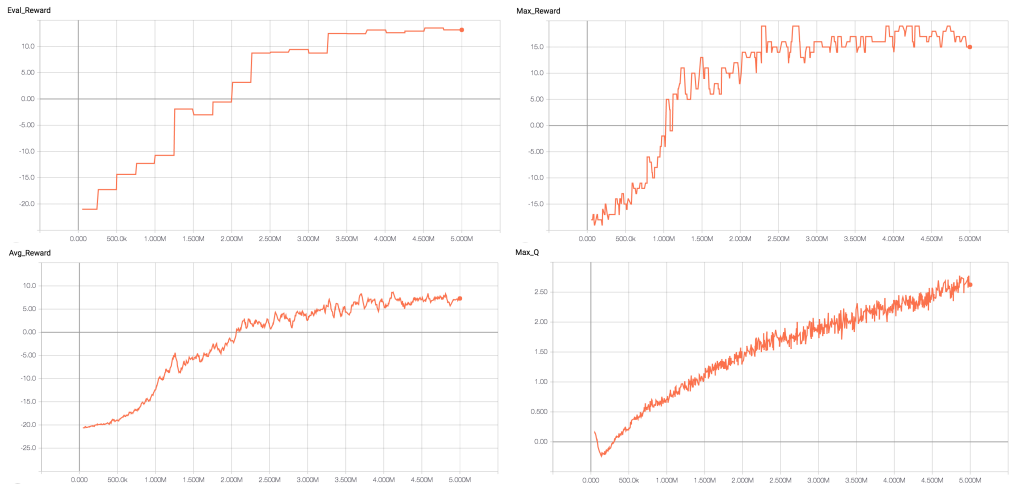
1. (**coding and written**, 5 pts). Now we're ready to train on the Atari `Pong-v0` environment. First, launch linear approximation on pong with `python q4_train_atari_linear.py` **on Azure's GPU**. This will train the model for 500,000 steps and should take approximately an hour. Briefly qualitatively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

   After running 500,000 linear approximation model on pong, the scores is still low and it doesn't really make progress, that means linear approximation is not good enough for Atari.

2. (**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:



   I have tried to run several different models for Atari. The difference between each model is either having activation function or having same padding or not.
   The first time I started with having activation function, relu, for each convolutional layer and without

same padding. Unfortunately, the result did not go well, we only got around -20 scores. Then I tried to turn off activation function for the CNN, the scores became around -2.5

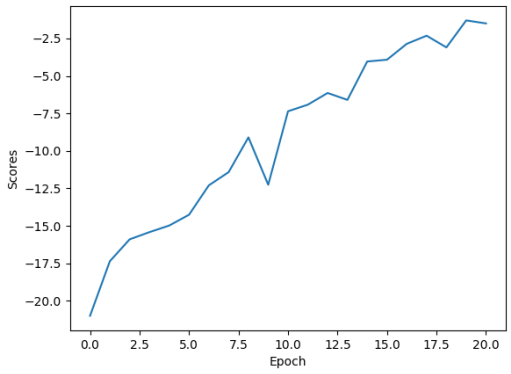


Figure 5: One layer relu and without same padding

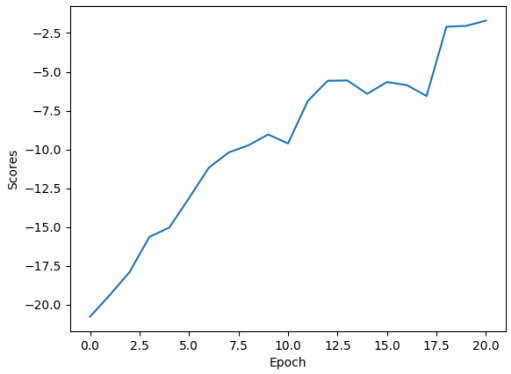


Figure 6: Without relu and without same padding

After that, I tried again the model without setting up any activation function and same padding and used layer.conv2d instead of tf.layer.conv2d. The result became good. We got about 11.9 scores after 20 epoches.

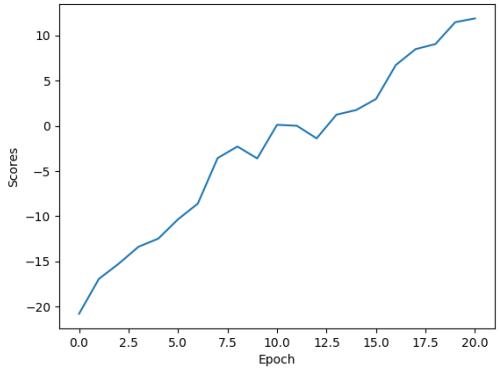


Figure 7: Without setting up anything

The final version of model I tried, went pretty well. The score is 13.68. It has last hidden layer with relu activation function and same padding for each layer. I am glad I finally made this :)

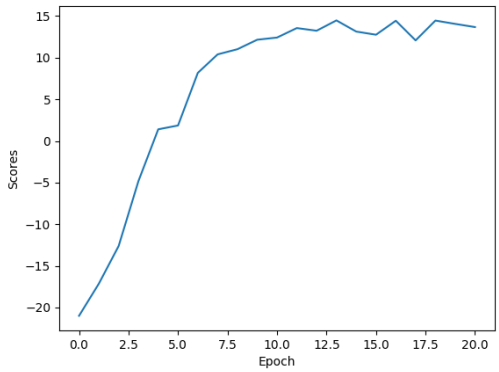


Figure 8: DQN for Atari - Final Result

3. (**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?

   The linear approximation model is not good enough for Atarti. We can also know that for analyzing image data, convolutional nueral networks work much better and have much higher efficiency to train the model with less parameters.

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

   DQN will improve monotonically over time, however, at some point it will slow down and almost seem like it's converged to a value. We can know that from the result of DQN Atari training, it slows down to increase score after millions of steps.

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 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.

1. (**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 $\hat{q}$.

   Prove

   $$\left| \mathbb{E}_\pi \left[\hat{q}_n(s,a) \right] - Q^\pi(s,a) \right| < \left| \mathbb{E}_\pi \left[\hat{q}_1(s,a) \right] - Q^\pi(s,a) \right|$$

   substitute
   $\hat{q}_n(s,a) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{q}(s_{t+n}, a_{t+n})$
   $\hat{q}_1(s,a) = r_t + \gamma \hat{q}(s_{t+1}, a_{t+1})$
   $Q^\pi(s,a) = \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \right]$

   The above equation becomes,

   $$\left| \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n \hat{q}(s_{t+n}, a_{t+n}) \right] - \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \right] \right| <$$

   $$\left| \mathbb{E}_\pi \left[r_t + \gamma \hat{q}(s_{t+1}, a_{t+1}) \right] - \mathbb{E}_\pi \left[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \right] \right|$$

   Apply linearity of expectation equation,

[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.

$$\left|\mathbb{E}_\pi\Big[\gamma^n \hat{q}(s_{t+n}, a_{t+n}) - \gamma^n r_{t+n} - \gamma^{n+1} r_{t+n+1} - \dots\Big]\right| <$$

$$\left|\mathbb{E}_\pi\Big[\gamma \hat{q}(s_{t+1}, a_{t+1}) - \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots\Big]\right|$$

Move γ to the front,

$$\Big|\gamma^n\Big|\left|\mathbb{E}_\pi\Big[\hat{q}(s_{t+n}, a_{t+n}) - r_{t+n} - \gamma^1 r_{t+n+1} - \dots\Big]\right| <$$

$$\Big|\gamma\Big|\left|\mathbb{E}_\pi\Big[\hat{q}(s_{t+1}, a_{t+1}) - r_{t+1} + \gamma^1 r_{t+2} + \dots\Big]\right|$$

Use $Q^\pi(s,a) = \mathbb{E}_\pi\Big[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \Big| s_t, a_t\Big]$,

$$\Big|\gamma^n\Big|\left|\mathbb{E}_\pi\Big[\hat{q}(s_{t+n}, a_{t+n}) - Q^\pi(s_{t+n}, a_{t+n})\Big]\right| <$$

$$\Big|\gamma\Big|\left|\mathbb{E}_\pi\Big[\hat{q}(s_{t+1}, a_{t+1}) - Q^\pi(s_{t+1}, a_{t+1})\Big]\right|$$

Since $\text{Bias}(\hat{q}(s,a)) = \text{Bias}(\hat{q}(s',a'))$ and $\gamma < 1$, we can have

$$\Big|\gamma^n\Big|Bias(\hat{q}(s_{t+n}, a_{t+n})) < \Big|\gamma\Big|Bias(\hat{q}(s_{t+1}, a_{t+1}))$$

to be true.

References

[1] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[2] Volodymyr Mnih et al. "Playing Atari With Deep Reinforcement Learning". In: *NIPS Deep Learning Workshop*. 2013.