XCS234 Assignment 2

Due Sunday, September 14 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs234-scpd.slack.com/
2. Familiarize yourself with the collaboration and honor code policy before starting work.
3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. For SCPD classes, it is also important that students avoid opening pull requests containing their solution code on the shared assignment repositories. More information regarding the Stanford honor code can be found at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Writing Code and Running the Autograder

All your code should be entered into the `src/submission/` directory. When editing files in `src/submission/`, please only make changes between the lines containing `### START_CODE_HERE ###` and `### END_CODE_HERE ###`. Do not make changes to files outside the `src/submission/` directory.

The unit tests in `src/grader.py` (the autograder) will be used to verify a correct submission. Run the autograder locally using the following terminal command within the `src/` subdirectory:

```
$ python grader.py
```

There are two types of unit tests used by the autograder:

- `basic`: These tests are provided to make sure that your inputs and outputs are on the right track, and that the hidden evaluation tests will be able to execute.

- `hidden`: These unit tests are NOT visible locally. These hidden tests will be run when you submit your code to the Gradescope autograder via the online student portal, and will provide feedback on how many points you have earned. These tests will evaluate elements of the assignment, and run your code with more complex inputs and corner cases. Just because your code passed the basic local tests does not necessarily mean that they will pass all of the hidden tests.

For debugging purposes, you can run a single unit test locally. For example, you can run the test case `3a-0-basic` using the following terminal command within the `src/` subdirectory:

```
$ python grader.py 3a-0-basic
```

Before beginning this course, please walk through the Anaconda Setup for XCS Courses to familiarize yourself with the coding environment. Use the env defined in `src/environment.yml` to run your code. This is the same environment used by the online autograder.

Test Cases

The autograder is a thin wrapper over the python `unittest` framework. It can be run either locally (on your computer) or remotely (on SCPD servers). The following description demonstrates what test results will look like for both local and remote execution. For the sake of example, we will consider two generic tests: `1a-0-basic` and `1a-1-hidden`.

Local Execution - Basic Tests

When a basic test like `1a-0-basic` passes locally, the autograder will indicate success:

```
----- START 1a-0-basic:  Basic test case.
----- END 1a-0-basic [took 0:00:00.000062 (max allowed 1 seconds), 2/2 points]
```

When a basic test like `1a-0-basic` fails locally, the error is printed to the terminal, along with a stack trace indicating where the error occurred:

```
----- START 1a-0-basic:  Basic test case.
<class 'AssertionError'>
{'a': 2, 'b': 1} != None
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/Users/grinch/Local_Documents/SCPD/XCS221/A1/src/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/Users/grinch/Local_Documents/Software/anaconda3/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
----- END 1a-0-basic [took 0:00:00.003809 (max allowed 1 seconds), 0/2 points]
```


Remote Execution

Basic and hidden tests are treated the same by the remote autograder, however the output of hidden tests will only appear once you upload your code to GradeScope. Here are screenshots of failed basic and hidden tests. Notice that the same information (error and stack trace) is provided as the in local autograder, now for both basic and hidden tests.

1a-0-basic) Basic test case. (0.0/2.0)

```
<class 'AssertionError'>:  {'a': 2, 'b': 1} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 23, in test_0
    submission.extractWordFeatures("a b a"))
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

Just like in the local autograder, this error caused the test to fail.

Just like in the local autograder, start your debugging in line 23 of grader.py.

1a-1-hidden) Test multiple instances of the same word in a sentence. (0.0/3.0)

```
<class 'AssertionError'>:  {'a': 23, 'ab': 22, 'aa': 24, 'c': 16, 'b': 15} != None
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
    yield
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 605, in run
    testMethod()
  File "/autograder/source/graderUtil.py", line 54, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/graderUtil.py", line 83, in wrapper
    result = func(*args, **kwargs)
  File "/autograder/source/grader.py", line 31, in test_1
    self.compare_with_solution_or_wait(submission, 'extractWordFeatures', lambda f: f(sentence))
  File "/autograder/source/graderUtil.py", line 183, in compare_with_solution_or_wait
    self.assertEqual(ans1, ans2)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 829, in assertEqual
    assertion_func(first, second, msg=msg)
  File "/autograder/source/miniconda/envs/XCS221/lib/python3.6/unittest/case.py", line 822, in _baseAssertEqual
    raise self.failureException(msg)
```

This error caused the test to fail.

Start your debugging in line 31 of grader.py.

Finally, here is what it looks like when basic and hidden tests pass in the remote autograder.

1a-0-basic) Basic test case. (2.0/2.0)

1a-1-hidden) Test multiple instances of the same word in a sentence. (3.0/3.0)

0 Introduction

In this assignment we will be exploring deep Q-learning. In particular, we will explore the application of deep Q-learning in training an agent to outperform the average human in an Atari game known as Pong! The ultimate goal of this assignment is to demonstrate the effectiveness of combining the reinforcement learning concepts we have learnt thus far with the effectiveness of neural networks as function approximators. In addition, we will convey the importance of some of the techniques used in practice to stabilize training and achieve better performance.

Some of the latter questions in this assignment make reference to research papers by DeepMind. Below you may find links to two papers which are closely related to this assignment (Note: questions whose answer requires knowledge from a given paper will contain a link to the paper within the corresponding question):

1. Human Level Control Through Deep Reinforcement Learning
2. Playing Atari with Deep Reinforcement Learning

It is important to note that this assignment contains 3 unique environment configuration files. Below we provide an explanation of each file as well as an outline of when you should use each file:

1. `environment.yml`: this is the default conda environment file which can be used for locally testing code. This is the same environment that is used by our autograder to test your code.
2. `environment_cuda.yml`: this conda environment is identical to `environment.yml` except that it ensures that the version of Pytorch which is installed is compatible with CUDA. This environment will be used when training models on the Azure vm instance or your local host using a Nvidia GPU.

Advice

- It takes approximately **6 hours** to train the DQN used in this assignment on the Atari enviroment (Q4). Be sure to allocate enough time at the end of the assignment to account for this.
- In this assignment you will make use of an Azure VM (or alternatively your local machine if it has a powerful GPU). Please make sure to terminate any vm instance you are no longer using to avoid the loss of your allocated GPU time which is limited.
- Throughout the assignment we will be using Pytorch to train our neural networks. It is strongly recommended you become familiar with the basics of Pytorch before starting the coding exercises. Please consult our Pytorch tutorial for a full review of Pytorch essentials.
- When running `run.py` with the DQN model in Q4, please ensure that the submission folder has write permissions for your user so that model weights can be saved there. This can be accomplished on the Azure VM through running `sudo chmod -R a+rw submission` from the assignment `src` directory.
- If you encounter the following error when installing on a MacOS with M1 chip run the following command `conda install openblas`.

```
bin/../lib/liblapack.3.dylib' (no such file), '/usr/local/lib/liblapack.3.dylib' (no such
  file), '/usr/lib/liblapack.3.dylib' (no such file)
```

Coding Deliverables

For this assignment, please submit the following files to gradescope to receive points for coding questions:

- `src/submission/__init__.py`
- `src/submission/q1_schedule.py`
- `src/submission/q2_linear_torch.py`
- `src/submission/q3_dqn_torch.py`
- `src/submission/model.weights`

1 Tabular Q-Learning

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

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

where $\alpha > 0$ is the learning rate, $\gamma \in [0,1)$ the discount factor.

In addition, to formalizing our update rule for Q-learning in the tabular setting we must also consider strategies for exploration. In this question, we will be considering an ϵ-greedy exploration strategy. This strategy means that each time we look to choose an action A, we will do so as follows,

$$A \leftarrow \begin{cases} argmax_{a \in \mathcal{A}} Q(s,a) & \text{with probability } 1-\epsilon \\ a \in \mathcal{A} \text{ chosen uniformly at random} & \text{with probability } \epsilon \end{cases}$$

In the following questions, you will need to implement both the update rule and ϵ-greedy exploration strategy for Q-learning in the tabular setting.

(a) **[8 points (Coding)]**

Implement the `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running:

```
$ python q1_schedule.py
```

(b) **[5 points (Written)]**

We will now examine the issue of overestimation bias in Q-learning. The crux of the problem is that, since we take a max over actions, errors which cause Q to overestimate will tend to be amplified when computing the target value, while errors which cause Q to underestimate will tend to be suppressed.

Assume for simplicity that our Q function is an unbiased estimator of Q^*, meaning that $\mathbb{E}[Q(s,a)] = Q^*(s,a)$ for all states s and actions a. Show that, even in this seemingly benign case, the estimator overestimates the real target in the following sense:

$$\forall s, \quad \mathbb{E}\left[\max_a Q(s,a)\right] \geq \max_a Q^*(s,a)$$

Note: The expectation $\mathbb{E}[Q(s,a)]$ is over the randomness in Q resulting from the stochasticity of the exploration process.

Based on Jensen's inequality, we have:

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$$

for random variable X and a convex function f.

The maximum function is a convex function and we consider Q-values for all actions in a given state as a random variable. Applying Jensen's inequality, we get:

$$\mathbb{E}[\max_{a \in \mathcal{A}} Q(s,a)] \geq \max_{a \in \mathcal{A}} (\mathbb{E}[Q(s,a)])$$

We are given that $\mathbb{E}[Q(s,a)] = Q^*(s,a)$, so

$$\mathbb{E}[\max_{a \in \mathcal{A}} Q(s,a)] \geq \max_{a \in \mathcal{A}} Q^*(s,a)$$

2 Linear Approximation

In the following question, we will implement linear approximation in PyTorch. If you feel you need a refresher of Pytorch concepts and functionality please consult the Pytorch Tutorial we have made available. It is worth noting that the functions you implement in this section will be used in future questions of this assignment. Therefore, in order to avoid the loss of points, please ensure you that your implementation passes the basic test cases we have provided in `grader.py` for this question.

Outside of checking the tests within `grader.py` you can also test the performance of your implementation locally on the test environment **using your computer's CPU** by running:

```
$ python run.py --config_filename=q2_linear
```

(a) **[5 points (Written)]**

Suppose we represent the Q function as $Q_\theta(s,a) = \theta^\top \delta(s,a)$, where $\theta \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ and $\delta : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^{|\mathcal{S}||\mathcal{A}|}$ with

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

Compute $\nabla_\theta Q_\theta(s,a)$ and write the update rule for θ. Argue that equation for the tabular q-learning update rule we saw before:

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

and the following equation:

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

are exactly the same when this form of linear approximation is used.

The partial derivative of a dot product is simply the other vector in the dot product (i.e. $\partial(u \cdot v)/\partial u = v$)), so

$$\nabla_\theta Q_\theta(s,a) = \frac{\partial Q_\theta(s,a)}{\partial \theta_{s,a}} = \frac{\partial \theta_{s,a} \cdot \delta(s,a)}{\partial \theta_{s,a}} = \delta(s,a)$$

Substitute this and into the linear update equation:

$$\theta \leftarrow \theta + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)) \nabla_\theta Q_\theta(s,a)$$

We get:

$$\theta \leftarrow \theta + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)) \delta(s,a)$$

Shows that only the single θ component correspond to (s,a) will be updated and thus identical to:

$$\theta \cdot \delta(s,a) \leftarrow \theta \cdot \delta(s,a) + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a)) \delta(s,a)$$

And therefore

$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a))$$

(b) **[4 points (Coding)]** Implement the `initialize_models` function in `submission/q2_linear_torch.py` which will define our linear model for both the Q-network and the target network.

(c) **[4 points (Coding)]** Implement the `get_q_values` function in `submission/q2_linear_torch.py` to output Q-values for a particular state and network.

(d) **[4 points (Coding)]** Implement the `update_target` function in `submission/q2_linear_torch.py` in order to update the target network with the Q-network weights.

(e) **[4 points (Coding)]** Implement the `calc_loss` function in `submission/q2_linear_torch.py` which will define our loss function.

(f) **[4 points (Coding)]** Implement the `add_optimizer` function in `submission/q2_linear_torch.py` which will define our optimizer.

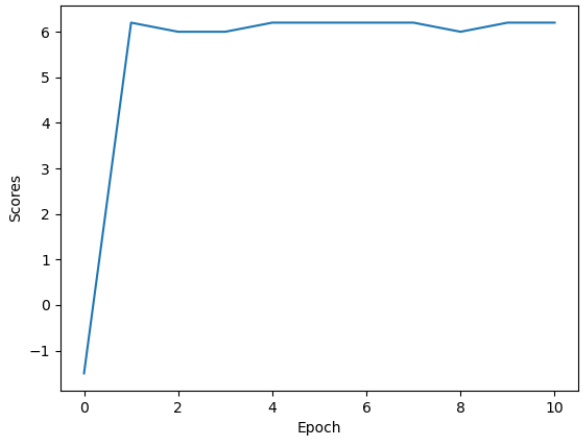
(g) **[4 points (Coding)]**

In this question, our grader will evaluate the performance of your linear implementation on the test environment based on the code you have already developed in previous questions in this section (No additional code needs to be written for this question).

If you would like, you can observe the performance metrics of your model through running the following command:

```
$ python run.py --config_filename=q2_linear
```

This should train your linear model on the test environment with the configuration defined in `config/q2_linear.yml`. You may view the evaluation scores from your training run under the following directory `results/q2_linear/`. We expect your implementation to achieve the optimal return on the test environment. Below we have provided a plot of scores which we expect the scores generated by your implementation to closely resemble:



Note: You will be need these results to provide responses to future questions which are made available online via Gradescope.

3 Implementing DeepMind's DQN

We are now ready to implement Deep Mind's DQN using Pytorch.

In the following questions, you will implement the deep Q-network as described in Human Level Control Through Deep Reinforcement Learning by implementing `initialize_models` and `get_q_values` in `q3_dqn_torch.py`. The rest of the code inherits from what you wrote for linear approximation.

You will be able to test your implementation locally on the test environment by running:

```
$ python run.py --config_filename=q3_dqn
```

(a) **[5 points (Coding)]** Implement the `initialize_models` function in `submission/q3_dqn_torch.py` which will define our network architecture for both the Q-network and the target network.

(b) **[6 points (Coding)]** Implement the `get_q_values` function in `submission/q3_dqn_torch.py` to output Q-values for a particular state and network.

(c) **[3 points (Written)]**

Compare the performance of your linear and DQN implementations on the test environment.

Note: The comparison may not be perfectly fair in this case as both experiments use different configs. This comparison should highlight the necessity of finding the right complexity of a model for a given problem.

The linear model is obviously learn quicker and give you more stable results while DQN can sometimes fail to reach the optimal policy with the epoch we choose. DQN might need even more training steps to reach the optimal policy.

In conclusion, for the simple test environment, the linear model is the better choice due to its efficiency and appropriate complexity. The DQN is unnecessarily powerful for this task.

4 DQN on Atari

Now that you have completed your implementation of DQN it is time to train an agent to play Pong using your code. Before we start this section we would also like to provide a kind reminder to **terminate any unused Azure instances in order to avoid running out of credits**.

A Brief History of Pong:

Pong is a table tennis themed arcade game which was officially released by Atari in 1972. At this point in time, Atari consisted of only two people Nolan Bushnell and Allan Alcorn. The creation of the arcade game was assigned to Allan Alcorn, unbeknownst to him it was assigned as a training exercise. Despite initially being set as a training exercise, Pong went on to become an officially released arcade game. The release of Pong was a major success and today it is often touted as being responsible for the rise in popularity of arcade video games.

Fortunately, we will have the opportunity to test the performance of our reinforcement learning agent on this historic game. What's more, we can expect our implementation of DQN to perform at a super human level. Let's get started!

Simulating Atari Games:

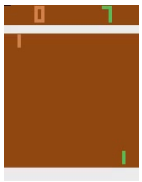
We will leverage the gym python package to simulate the arcade game Pong. In general, the gym package provides a suite of environments within which you may test reinforcement learning algorithms. For details of all the available environments and usage of the gym python package, please consult the official documentation for the library.

Data Preprocessing:

The Atari environment from Gymnasium (former 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, we will apply the following preprocessing to the observations:

1. 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.
2. Convert the encoded frame to grey scale; crop and rescale it to $(80 \times 80 \times 1)$. (See Figure 1)

The above preprocessing is applied to the 4 most recent observations and these encoded frames are stacked together to produce the input (of shape $(80 \times 80 \times 1)$) 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 DeepMind's paper for more details.



(a) Original input $(210 \times 160 \times 3)$

(b) Input after preprocessing $(80 \times 80 \times 1)$

Figure 1: `Pong` environment

Setting up your Virtual Machine

Follow the instructions in the Azure Guide in order to create your vm instance and deploy the assignment code to your vm. Though you will need the GPU to train your model, we strongly advise that you first develop the code locally and ensure that it runs, before attempting to train it on your vm (passing the grader tests and debugging on the test environment should be sufficient). GPU time is expensive and limited. It takes approximately **8 hours** to train our DQN model. We don't want you to accidentally use all your GPU time for the assignment, debugging

your model rather than training and evaluating it. Finally, **make sure that your VM is turned off whenever you are not using it.**

In order to run the model code on your vm, please run the following command to create the proper virtual environment:

```
$ conda update -n base conda
$ conda env create --file environment_cuda.yml
```

Note: we are using the gpu version of the conda environment file.

For local development and testing, see the Introduction for instructions on which environment you should use.

If you wish to monitor your model training in real-time, you may optionally activate tensorboard and port forward the tensorboard outputs on your vm to an available port on your local machine (we strongly recommend that you do). This will enable you to view the tensorboard dashboard and your model training metrics on your local machine in real-time. To get this setup you should first start training your implementation of DQN in order to generate training metrics for tensorboard to plot (or optionally have already trained your implementation and simply wish to visualize the results).

By default tensorboard runs on port 6006. Therefore, we will look to port forward 6006 on your vm to an available port on your local machine using ssh. For this you will need to identify an available port on your local machine. To test if a process is already running on a particular port simply enter the following command in your bash session `lsof -i : <port_number>`. If this returns a process, you will need to either kill this process or find a new port (don't kill any processes without understanding its purposes and the reprecusions of stopping the process). Once you have established an available port, let's say for example 12345, then run the following command (which will forward vm's port 6006 to your desktop) *from your local machine*:

```
# Run this from your local machine where you run the browser
$ ssh -L 12345:localhost:6006 -p xxxxx student@ml-lab-xxxxxxx.eastus.cloudapp.azure.com
```

Where, in the above command, you will need to fill in the sections marked with "x". These values should align with those you regularly use to ssh onto your vm (see Azure Guide for further details). You should now be prompted to enter the password you have already setup for your virtual machine. Enter your password and run the following command to start tensorboard:

```
# Run this from vm
$ tensorboard --logdir=<results_folder> --host 0.0.0.0
```

Finally, open up a browser on your local machine and visit *http://localhost:12345* to view the tensorboard dashboard. Note that use *http* not *https* for the above command.

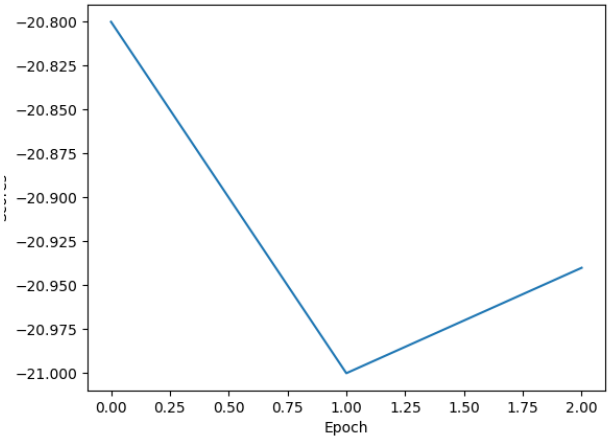
(a) **[3 points (Written)]**

Now we're ready to train on the Atari `Pong` environment. First, launch linear approximation on pong by running the following command:

```
$ python run.py --config_filename=q4_linear
```

This will train the model for $500,000$ steps and should take approximately half an hour. Once training is complete observe the contents of `results/q4_linear` and in particular the plot `scores.png` (the Azure Guide contains details of moving files from the vm to your local machine).

Using this plot, describe the performance of your linear model on the `Pong-v5` environment.



The performance of the linear model on the Pong-v5 environment is extremely poor and shows no evidence of learning.

(b) **[5 points (Coding)]**

In this question, we'll train the agent with DeepMind's architecture on the Atari `Pong` environment. Run the following command to start the training process:

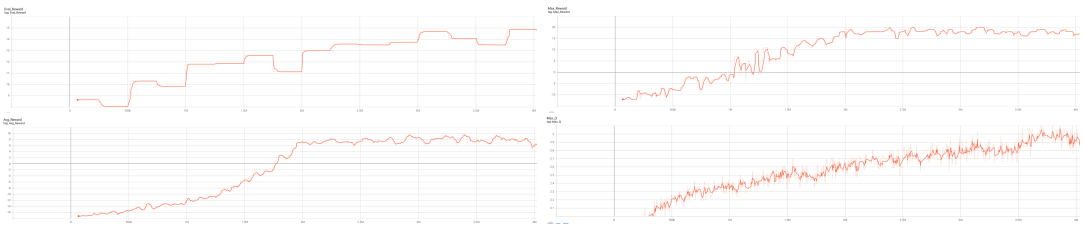
```
$ python run.py --config_filename=q4_dqn
```

To speed up training, we have trained the model for 5 million steps (these pretrained weights will be automatically loaded once you run the above command). You are responsible for training it to completion, which should take **6 hours**. You should get a score of around 12-15 after 4 million total time steps. As stated previously, the DeepMind paper claims average human performance is −3. Once your model has fully trained download the following file to your local machine `src/submission/model.weights` and include these weights with your code submission to Gradescope.

Note: the weights file needs to be in the submission folder for the autograder to read them when you run the grader on your local machine.

As the training time is roughly 6 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, any training processes which are running within this session will terminate. In order to avoid this, you can start a session with Tmux which will persist even if you lose your connection to the vm. See the Azure guide for further details).
- The evaluation score printed on terminal should start at 6 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.
- Please find our Tensorboard graphs from one training session below.



(c) **[3 points (Written)]**

Given the results generated from training both DQN and a linear approximator on the Atari environment `Pong-v5` compare the performance of the DeepMind DQN architecture with the linear Q value approximator. In particular, what explains the observed performance gap between the two network architectures?

The DQN is obviously improving its performance over time while the linear model doesn't seem to improve.

The DQN has a deep CNN layer that are specifically designed for image data. It gives the DQN the understanding of the game state directly from pixels, which is necessary for high-level strategic play.

On the other hand, the linear model treat each pixels as an independent feature and attempts to find a simple linear relationship between pixel values and strategy, which is insufficient.

(d) **[3 points (Written)]**

Will the performance of DQN always improve monotonically over time? Explain why or why not.

No, the performance of DQN is improving in the long run but it fluctuate a lot locally. It could be the ϵ-greddy exploration where it randomly choose the actions sometimes that causes the fluctuation. Also, the deep neural network is a highly non-linear function approximation. A small parameter change can lead to significant change in the policy.

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the | README.md | for this assignment includes instructions to regenerate this handout with your typeset LaTeX solutions.

1.b

Based on Jensen's inequality, we have:

$$\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$$

for random variable X and a convex function f.

The maximum function is a convex function and we consider Q-values for all actions in a given state as a random variable. Applying Jensen's inequality, we get:

$$\mathbb{E}[\max_{a \in \mathcal{A}} Q(s,a)] \geq \max_{a \in \mathcal{A}} (\mathbb{E}[Q(s,a)])$$

We are given that $\mathbb{E}[Q(s,a)] = Q^*(s,a)$, so

$$\mathbb{E}[\max_{a \in \mathcal{A}} Q(s,a)] \geq \max_{a \in \mathcal{A}} Q^*(s,a)$$

2.a

The partial derivative of a dot product is simply the other vector in the dot product (i.e. $\partial(u \cdot v)/\partial u = v$), so

$$\nabla_\theta Q_\theta(s,a) = \frac{\partial Q_\theta(s,a)}{\partial \theta_{s,a}} = \frac{\partial \theta_{s,a} \cdot \delta(s,a)}{\partial \theta_{s,a}} = \delta(s,a)$$

Substitute this and into the linear update equation:

$$\theta \leftarrow \theta + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)$$

We get:

$$\theta \leftarrow \theta + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a))\delta(s,a)$$

Shows that only the single θ component correspond to (s,a) will be updated and thus identical to:

$$\theta \cdot \delta(s,a) \leftarrow \theta \cdot \delta(s,a) + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a))\delta(s,a)$$

And therefore

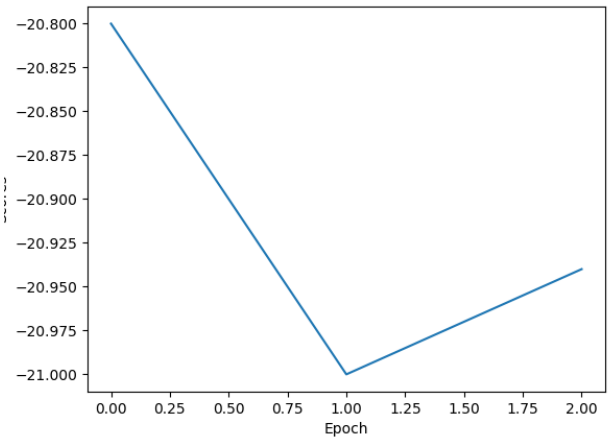
$$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha(r + \gamma \max_{a'} Q_\theta(s',a') - Q_\theta(s,a))$$

3.c

The linear model is obviously learn quicker and give you more stable results while DQN can sometimes fail to reach the optimal policy with the epoch we choose. DQN might need even more training steps to reach the optimal policy.

In conclusion, for the simple test environment, the linear model is the better choice due to its efficiency and appropriate complexity. The DQN is unnecessarily powerful for this task.

4.a



The performance of the linear model on the Pong-v5 environment is extremely poor and shows no evidence of learning.

4.c

The DQN is obviously improving its performance over time while the linear model doesn't seem to improve.

The DQN has a deep CNN layer that are specifically designed for image data. It gives the DQN the understanding of the game state directly from pixels, which is necessary for high-level strategic play.

On the other hand, the linear model treat each pixels as an independent feature and attempts to find a simple linear relationship between pixel values and strategy, which is insufficient.

4.d

No, the performance of DQN is improving in the long run but it fluctuate a lot locally. It could be the ϵ-greddy exploration where it randomly choose the actions sometimes that causes the fluctuation. Also, the deep neural network is a highly non-linear function approximation. A small parameter change can lead to significant change in the policy.