CS 234 Winter 2021: Assignment #2

Due date:
Part 1 (0-4): February 5, 2021 at 6 PM (18:00) PST
Part 2 (5-6): February 12, 2021 at 6 PM (18:00) 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 PyTorch. 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   Distributions induced by a policy (13 pts)

In this problem, we'll work with an infinite-horizon MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \gamma \rangle$ and consider stochastic policies of the form $\pi : \mathcal{S} \to \Delta(\mathcal{A})^1$. Additionally, we'll assume that $\mathcal{M}$ has a single, fixed starting state $s_0 \in \mathcal{S}$ for simplicity.

(a) (**written**, 3 pts) Consider a fixed stochastic policy and imagine running several rollouts of this policy within the environment. Naturally, depending on the stochasticity of the MDP $\mathcal{M}$ and the policy itself, some trajectories are more likely than others. Write down an expression for $\rho^\pi(\tau)$, the likelihood of sampling a trajectory $\tau = (s_0, a_0, s_1, a_1, \ldots)$ by running π in $\mathcal{M}$. To put this distribution in context, recall that $V^\pi(s_0) = \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^{\infty} \gamma^t \mathcal{R}(s_t, a_t) \,|\, s_0 \right]$.

Solution:

$$\rho^\pi(\tau) = \prod_{t=0}^{\infty} \pi(a_t|s_t) \mathcal{T}(s_{t+1}|s_t, a_t)$$

(b) (**written**, 5 pts) Just as ρ^π captures the distribution over trajectories induced by π, we can also examine the distribution over states induced by π. In particular, define the *discounted, stationary state distribution* of a policy π as

$$d^\pi(s) = (1 - \gamma) \sum_{t=0}^{\infty} \gamma^t p(s_t = s),$$

where $p(s_t = s)$ denotes the probability of being in state s at timestep t while following policy π; your answer to the previous part should help you reason about how you might compute this value. Consider an arbitrary function $f : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Prove the following identity:

$$\mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^{\infty} \gamma^t f(s_t, a_t) \right] = \frac{1}{(1 - \gamma)} \mathbb{E}_{s \sim d^\pi} \left[\mathbb{E}_{a \sim \pi(s)} \left[f(s, a) \right] \right].$$

Hint: You may find it helpful to first consider how things work out for $f(s, a) = 1, \forall (s, a) \in \mathcal{S} \times \mathcal{A}$.
Hint: What is $p(s_t = s)$?

Solution:

$$
\begin{aligned}
\mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^{\infty} \gamma^t f(s_t, a_t) \right] &= \sum_{t=0}^{\infty} \gamma^t \mathbb{E}_{\tau \sim \rho^\pi} \left[f(s_t, a_t) \right] \\
&= \mathbb{E}_{\tau \sim \rho^\pi} \left[f(s_0, a_0) \right] + \gamma \mathbb{E}_{\tau \sim \rho^\pi} \left[f(s_1, a_1) \right] + \gamma^2 \mathbb{E}_{\tau \sim \rho^\pi} \left[f(s_2, a_2) \right] + \ldots \\
&= \sum_{a_0} \pi(a_0|s_0) f(s_0, a_0) + \gamma \sum_{a_0} \pi(a_0|s_0) \sum_{s_1} \mathcal{T}(s_1|s_0, a_0) \sum_{a_1} \pi(a_1|s_1) f(s_1, a_1) + \ldots \\
&= \sum_{s} p(s_0 = s) \mathbb{E}_{a \sim \pi(s)}[f(s, a)] + \gamma \sum_{s} p(s_1 = s) \mathbb{E}_{a \sim \pi(s)}[f(s, a)] + \ldots \\
&= \sum_{s} \sum_{t=0}^{\infty} \gamma^t p(s_t = s) \mathbb{E}_{a \sim \pi(s)}[f(s, a)] \\
&= \frac{1}{(1 - \gamma)} \sum_{s} d^\pi(s) \mathbb{E}_{a \sim \pi(s)}[f(s, a)] = \frac{1}{(1 - \gamma)} \mathbb{E}_{s \sim d^\pi} \left[\mathbb{E}_{a \sim \pi(s)} \left[f(s, a) \right] \right]
\end{aligned}
$$

---

[1]For a finite set $\mathcal{X}$, $\Delta(\mathcal{X})$ refers to the set of categorical distributions with support on $\mathcal{X}$ or, equivalently, the $\Delta^{|\mathcal{X}|-1}$ probability simplex.

(c) (**written**, 5 pts) For any policy π, we define the following function

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s).$$

Prove the following statement holds for all policies π, π':

$$V^\pi(s_0) - V^{\pi'}(s_0) = \frac{1}{(1-\gamma)} \mathbb{E}_{s \sim d^\pi} \left[\mathbb{E}_{a \sim \pi(s)} \left[A^{\pi'}(s, a) \right] \right].$$

Solution:

$$V^\pi(s_0) - V^{\pi'}(s_0) = \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t \mathcal{R}(s_t, a_t) \right] - V^{\pi'}(s_0)$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t \left(\mathcal{R}(s_t, a_t) + V^{\pi'}(s_t) - V^{\pi'}(s_t) \right) \right] - V^{\pi'}(s_0)$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t \left(\mathcal{R}(s_t, a_t) + \gamma V^{\pi'}(s_{t+1}) - V^{\pi'}(s_t) \right) \right]$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\mathbb{E} \left[\sum_{t=0}^\infty \gamma^t \left(\mathcal{R}(s_t, a_t) + \gamma V^{\pi'}(s_{t+1}) - V^{\pi'}(s_t) \right) \Big| s_t, a_t \right] \right]$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t \left(\mathcal{R}(s_t, a_t) + \gamma \mathbb{E} \left[V^{\pi'}(s_{t+1}) \Big| s_t, a_t \right] - V^{\pi'}(s_t) \right) \right]$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t \left(Q^{\pi'}(s_t, a_t) - V^{\pi'}(s_t) \right) \right]$$

$$= \mathbb{E}_{\tau \sim \rho^\pi} \left[\sum_{t=0}^\infty \gamma^t A^{\pi'}(s_t, a_t) \right]$$

$$= \frac{1}{(1-\gamma)} \mathbb{E}_{s \sim d^\pi} \left[\mathbb{E}_{a \sim \pi(s)} \left[A^{\pi'}(s, a) \right] \right].$$

The function $A^\pi(s, a)$ is known as the advantage function which quantifies how much more advantageous it may (or may not) be to take action a in state s and follow policy π thereafter, rather than following policy π in state s.

1 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 $R(0) = 0.2, R(1) = -0.1, R(2) = 0.0, R(3) = -0.3$. 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	0.2
0	1	1	-0.1
0	2	2	0.0
0	3	3	-0.3
0	4	0	0.2
1	0	0	0.2
1	1	1	-0.1
1	2	2	0.0
1	3	3	-0.3
1	4	1	-0.1
2	0	0	-2.0
2	1	1	1.0
2	2	2	0.0
2	3	3	3.0
2	4	2	0.0
3	0	0	0.2
3	1	1	-0.1
3	2	2	0.0
3	3	3	-0.3
3	4	3	-0.3

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: $s_0 = 0, a_0 = 1, R_0 = -0.1, s_1 = 1, a_1 = 2, R_1 = 0.0, s_2 = 2, a_2 = 4, R_2 = 0.0, s_3 = 2, a_3 = 3, R_3 = 3.0, s_4 = 3, a_4 = 0, R_4 = 0.2, s_5 = 0$.
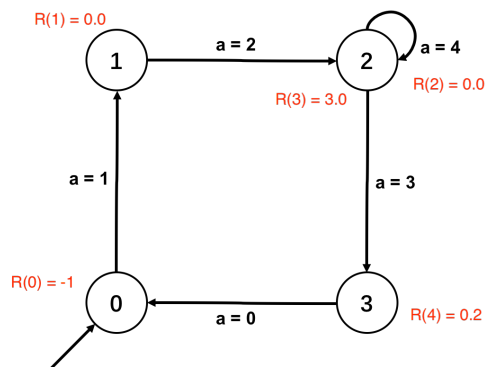

Figure 1: Example of a trajectory in the Test Environment

(a) (**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

<div style="color:red">

6.2

To prove this, let's prove an **upper bound** of 6.2 with 3 key observations

- first, the maximum reward we can achieve is 3 when we do $2 \to 3$.
- 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 6. Then, the best reward we can achieve that is not an optimal move (starting from state 1) is 0.2, which yields an upper bound of 6.2.

Considering the path $0 \to 2 \to 3 \to 2 \to 3 \to 0$ proves that we can **achieve** this upper bound.

</div>

2 Tabular Q-Learning (8 pts)

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.

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., $\arg\max_{a \in \mathcal{A}} Q(s, a)$) is chosen.

(a) (**coding**, 3 pts) Implement the `get_action` and `update` functions in `q2_schedule.py`. Test your implementation by running `python q2_schedule.py`.

Overestimation bias 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.

(b) (**written**, 5 pts) 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. (Note that this expectation is over the randomness in Q resulting from the stochasticity of the exploration process.) Show that, even in this seemingly benign case, the estimator overestimates the real target in the following sense:

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

<div style="color:red">

Solution: Let s be an arbitrary state. Since we assume that Q is unbiased, it suffices to show that the expectation of the max is at least the max of the expectation, i.e.

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

</div>

Observe that for all actions a we have $\max_{a'} Q(s, a') \geq Q(s, a)$ with probability 1, which implies that

$$\forall a, \quad \mathbb{E}\left[\max_{a'} Q(s, a')\right] \geq \mathbb{E}[Q(s, a)]$$

(Abstractly, if $X \geq Y$ w.p. 1, then $\mathbb{E}[X] \geq \mathbb{E}[Y]$.) Since this inequality holds for any a and the left-hand side does not depend on a, it implies the inequality above it. (Abstractly, if $\forall x$ it holds that $c \geq f(x)$, then $c \geq \max_x f(x)$.)

3 Q-Learning with Function Approximation (13 points)

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 parametric function $Q_\theta(s, a)$ where $\theta \in \mathbb{R}^p$ are the parameters of the function (typically the weights and biases of a linear function or a neural network). In this *approximation setting*, the update rule becomes

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s', a') - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a) \tag{2}$$

where (s, a, r, s') is a transition from the MDP.

To improve the data efficiency and stability of the training process, DeepMind's paper [1] employed two strategies:

- A *replay buffer* to store transitions observed during training. When updating the Q function, transitions are drawn from this replay buffer. This improves data efficiency by allowing each transition to be used in multiple updates.

- A *target network* with parameters $\bar{\theta}$ to compute the target value of the next state, $\max_{a'} Q(s', a')$. The update becomes

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\bar{\theta}}(s', a') - Q_\theta(s, a) \right) \nabla_\theta Q_\theta(s, a) \tag{3}$$

Updates of the form (3) applied to transitions sampled from a replay buffer $\mathcal{D}$ can be interpreted as performing stochastic gradient descent on the following objective function:

$$L_{\mathrm{DQN}}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} Q_{\bar{\theta}}(s', a') - Q_\theta(s, a) \right)^2 \right] \tag{4}$$

Note that this objective is also a function of both the replay buffer $\mathcal{D}$ and the target network $Q_{\bar{\theta}}$. The target network parameters $\bar{\theta}$ are held fixed and not updated by SGD, but periodically – every C steps – we synchronize by copying $\bar{\theta} \leftarrow \theta$.

We will now examine some implementation details.

(a) (**written** 3 pts) DeepMind's deep Q network (DQN) takes as input the state s and outputs a vector of size $|\mathcal{A}|$, the number of actions. What is one benefit of computing the Q function as $Q_\theta(s, \cdot) \in \mathbb{R}^{|\mathcal{A}|}$, as opposed to $Q_\theta(s, a) \in \mathbb{R}$?

Solution: In order to compute the greedy action or make a Q-Learning update, we must compute all the Q values for the given state in order to determine the max. If we parameterized the DQN to take both a state and action as input, we would need to execute $|\mathcal{A}|$ forward-passes (one per legal action) in order to compute this (arg)max. 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.

(b) (**written** 5 pts) Describe the tradeoff at play in determining a good choice of C. In particular, why might it not work to take a very small value such as $C = 1$? Conversely, what is the consequence of taking C very large? What happens if $C = \infty$?

Solution: Observe that the objective (4) depends on the target network $Q_{\bar{\theta}}$, so whenever $\bar{\theta}$ is updated, the objective function being optimized changes. Updating $\bar{\theta}$ very frequently (i.e. small C) may cause the optimization procedure not to converge.

On the other hand, while infrequent updates (i.e. large C) improve the stability of training, they will likely increase the total amount of time required to arrive at good estimates of Q^*, since Q_θ will be updated towards incorrect target values. In the extreme case $C = \infty$, the target network is never updated, so Q_θ will converge but to very incorrect Q values.

(c) (**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

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

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_θ is a predictive model parameterized by θ.

This objective looks very similar to the DQN objective (4). How are these two scenarios different? (There are at least two significant differences.)

Solution: One key distinction is that in supervised learning, the data distribution is assumed to be fixed. In RL, the policy is changing as learning progresses, so the distribution of interest is changing during learning as well. In fact, the distribution is somewhat under our control, since it is influenced by the algorithm's exploration process.

Another important point is that the target (i.e. label) in supervised learning is known, whereas the target in Q learning is unknown and must be estimated. As pointed out above, this can destabilize the optimization if not accounted for.

4 Linear Approximation (23 pts)

(a) (**written**, 5 pts) 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 equations (1) and (2) from above are exactly the same when this form of linear approximation is used.

Solution: We have

$$\nabla_\theta Q_\theta(s, a) = \nabla_\theta \left(\theta^\top \delta(s, a)\right) = \delta(s, a)$$

so the update rule becomes

$$\theta \leftarrow \theta + \alpha \left(r + \gamma \max_{a' \in A} \theta^\top \delta(s', a') - \theta^\top \delta(s, a)\right) \delta(s, a)$$

$$= \theta + \alpha \left(r + \gamma \max_{a' \in A} \theta_{s', a'} - \theta_{s, a}\right) \delta(s, a)$$

Written out element-wise, this is

$$\theta_{\bar{s},\bar{a}} \leftarrow \begin{cases} \theta_{s,a} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \theta_{s',a'} - \theta_{s,a}\right) & \text{if } (\bar{s}, \bar{a}) = (s, a) \\ \theta_{\bar{s},\bar{a}} & \text{otherwise} \end{cases}$$

which is exactly identifiable with (1), since $Q_\theta(s, a) = \theta_{s,a}$.

(b) (**coding**, 15 pts) We will now implement linear approximation in PyTorch. This question will set up the pipeline for the remainder of the assignment. You'll need to implement the following functions in `q3_linear_torch.py` (please read through `q3_linear_torch.py`):

- `initialize_models`
- `get_q_values`
- `update_target`
- `calc_loss`
- `add_optimizer`

Test your code by running `python q3_linear_torch.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Problem 0. Running this implementation should only take a minute.

(c) (**written**, 3 pts) Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q3_linear` to your writeup.
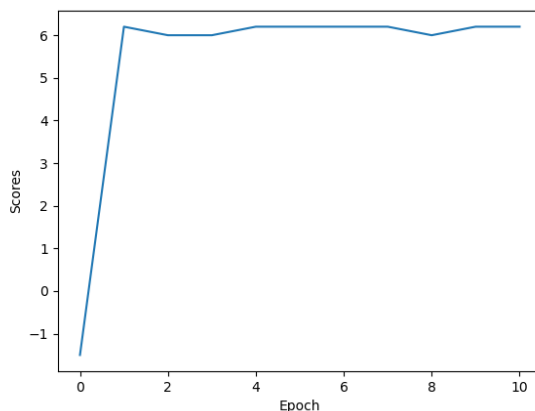
Solution: Yes, 6.2. See Figure 2.

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

5 Implementing DeepMind's DQN (13 pts)

(a) (**coding** 10pts) Implement the deep Q-network as described in [1] by implementing `initialize_models` and `get_q_values` in `q4_nature_torch.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 q4_nature_torch.py`. Running this implementation should only take a minute or two.
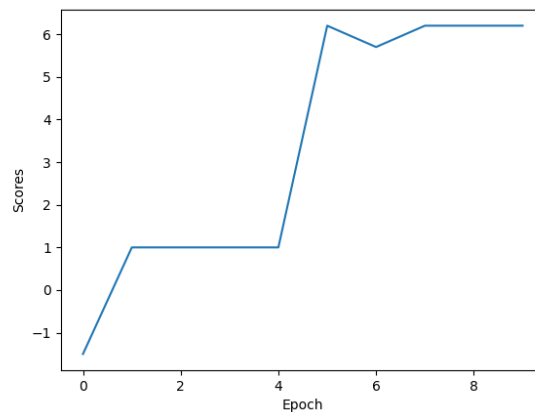
Figure 3: Reference plot for the nature network with PyTorch on the Test Env

(b) (**written** 3 pts) Attach the plot of scores, `scores.png`, from the directory `results/q4_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?

Solution:

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

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

Note: The comparison may not be fair in this case as both experiments use different configs and environments. Try changing q4's config and environment to match q4's and you should get a similar conclusion.

6   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.
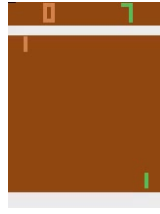
(a) Original input ($210 \times 160 \times 3$) with RGB colors

(b) After preprocessing in grey scale of shape ($80 \times 80 \times 1$)

Figure 4: `Pong-v0` environment

(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 q5_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 q6_train_atari_nature.py` **on Azure's GPU**. This will train the model for 4 million steps. To speed up training, we have trained the model for 2 million steps. You are responsible for training it to completion, which should take **12 hours**. Attach the plot `scores.png` from the directory `results/q6_train_atari_nature` to your writeup. You should get a score of around 11-13 after 4 million total time steps. As stated previously, the DeepMind paper claims average human performance is -3.
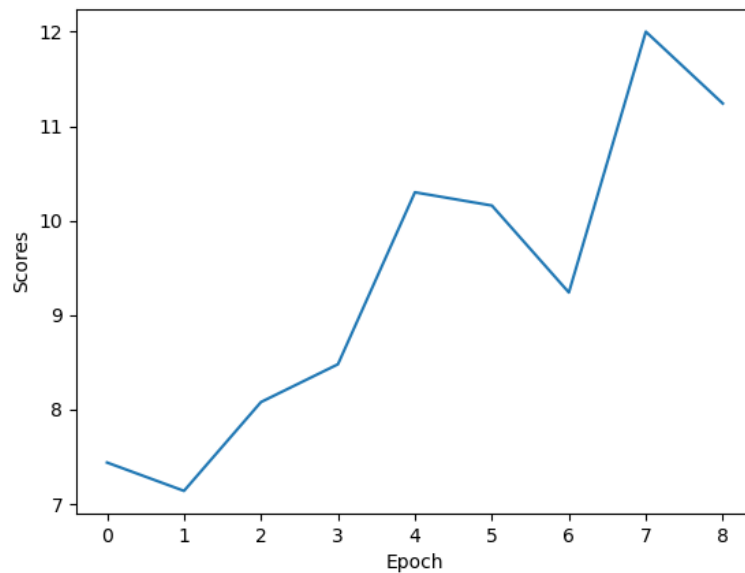
Solution:

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 11 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 compared to the convolutional network. Vision problems are inherently hierarchical and nonlinear, and the convolution operation is well-suited to capturing spatial patterns.

(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. There could be many reasons:

- The target Q-values are likely incorrect because we bootstrap from estimates.
- Even if the targets are correct, the randomness of SGD may temporarily lead to a worse Q function.
- The replay buffer contains only a finite amount of data and does not completely capture the dynamics of the MDP.

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.