CSCE 775-001, Spring 2023
Assignment 2

Username: [your email username]
Name: [your first and last name]

By turning in this assignment, I agree by the honor code of USC Columbia.

Submission. You need to submit the following files to Blackboard:

- A pdf file named as assignment2_⟨*username*⟩.pdf, where you replace ⟨*username*⟩ with your email username. This pdf file contains your answers to the written problems. Edit the assignment2.tex file (or scan your handwritten script) to fill in your answers, and submit the pdf file generated from the edited tex file.
- A zip file named as assignment2_⟨*username*⟩.zip, where you replace ⟨*username*⟩ with your email username. This zip file contains the three .py files you will edit for problem 2.

1 Implementing Q-learning [100pt]

This problem is adapted from Emma Brunskill's course.

In this problem, we will implement variants of Q-learning, including tabular Q-learning, Q-learning with linear function approximators, and Deep Q-Network (DQN). Your implementation will be tested on a test environment on your local computer with CPU and OpenAI gym's `Pong-v0` environment (an Atari game) on the recommended platform, Google Colab.

The starter code is in folder `starter_code_torch`.

Working locally. Please be sure you have Anaconda or Miniconda installed. Create conda environment on your local system: replace `<your-system>` with your system, either mac or windows:

```
cd starter_code_torch
conda env create -f cs234-torch-<your-system>.yml
conda activate cs234-torch
```

Working remotely on Google Colab. If you are not familiar with Google Colab, just create an account at Google Colab and follow the tutorials. Google Colab is very nice to run Python code without the hassle of installing libraries and to leverage a GPU or TPU.

1.1 Test environment [15pt]

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 defined 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 1, and the trajectory can be represented in terms of s_t, a_t, R_t as: $s_0 = 0, a_0 = 1, R_0 = -0.1, s_1 = 1, a_1 = 2, R_1 = 0.0, s_2 = 2, a_2 = 4, R_2 = 0.0, s_3 = 2, a_3 = 3, R_3 = 3.0, s_4 = 3, a_4 = 0, R_4 = 0.2, s_5 = 0$.

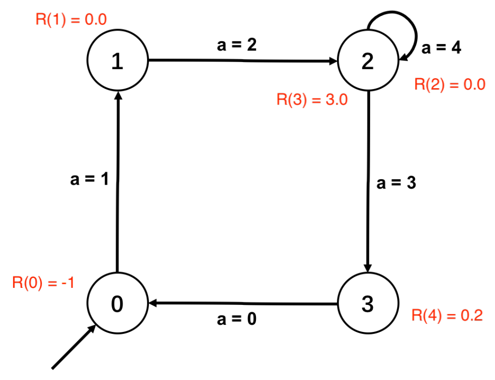

Figure 1: Example of a trajectory in the Test Environment

(**writing**) 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.

Your answer here.

1.2 Tabular Q-learning [20pt]

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.

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.

(**coding**) Implement the `get_action` and `update` functions in `q2_schedule.py`. Test your implementation by running `python q2_schedule.py`. Include your `q2_schedule.py` in the zip file for submission.

1.3 Q-Learning with Function Approximation [35pt]

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 DQN 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

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

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:

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

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 $\bar{\theta} \leftarrow \theta$.

(a) (**coding**) 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 1.1. Running this implementation should only take a minute. Include your `q3_linear_torch.py` in the zip file for submission.

(b) (**writing**) 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.

Your answer here.

(c) (**coding**) Implement the deep Q-network as described in 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. Include your `python q4_nature_torch.py` in the zip file for submission.

(d) (**writing**) 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?

Your answer here.

1.4 DQN on Atari [30pt]

The `Pong-v0` 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 the DQN paper, 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 3)

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.

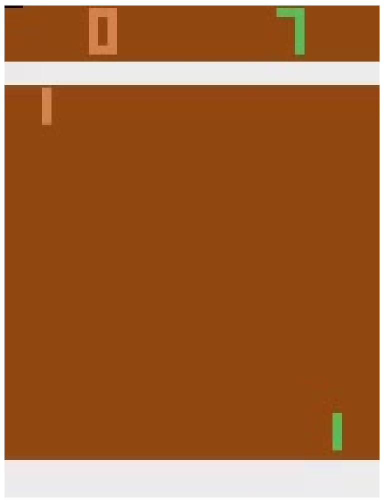

Figure 2: Original input $(210 \times 160 \times 3)$ with RGB colors

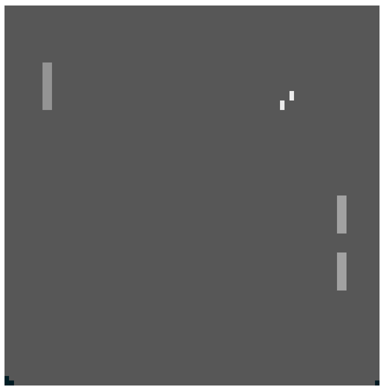

Figure 3: After preprocessing in grey scale of shape $(80 \times 80 \times 1)$

(a) (**coding and written**). 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` **(recommended on Google Colab'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.

Your answer here.