CP8319: Reinforcement Learning (Winter 2022)
Default Project
Due: Tuesday April 19, 2022 (Midnight 11:59PM or 23:59)
Instructor: Nariman Farsad

Please Read Carefully:

- This is the default project for the students.
- **The default project must be done individually**. It is treated as an extended assignment.
- You need to follow the instructions for the submission of the final project and the accompanying report, which will be posted to D2L later. But generally, the final report for the default project will be in the form of a detailed answer to the questions asked in this document, as well as a detailed description on all the methods that were considered in question 5. The students should highlight the best performance they could achieve.
- The default project is accompanied by a set of starter code that helps you get started.
- For instructions on setting up the python environment for the project read the "README.md" file.

In this project we will implement deep Q-learning, following DeepMind's paper ([2] and [1]) 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 [1]). In this project, you will train an AI agent with super-human performance.

1. (10 points) **Test Environment** 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: $s_0 = 0, a_0 = 1, R_0 = -0.2, s_1 = 1, a_1 = 2, R_1 = 0, s_2 = 2, a_2 = 4, R_2 = 0, s_3 = 2, a_3 = 3, R_3 = (-0.1) \cdot (-10) = 1, s_4 = 3, a_4 = 0, R_4 = 0.1, s_5 = 0$.

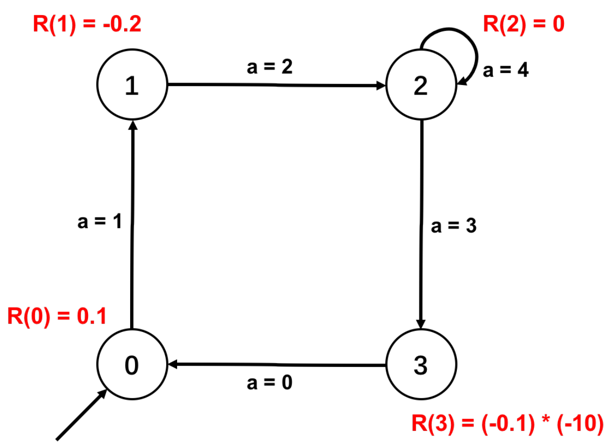


Figure 1: Example of a trajectory in the Test Environment

(a) (**written** 10 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.

2. (20 points) **Q-Learning and Value Function Approximation**

The following reviews some of the topics we learned in the lectures. Feel free to skip to the question part, but it might be good to read through this as a review.

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

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a) \right) \tag{0.1}$$

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

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s,a;\mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(s,a;\mathbf{w}). \tag{0.2}$$

In other words, we aim to minimize

$$L(\mathbf{w}) = \mathop{\mathbb{E}}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}(s', a'; \mathbf{w}) - \hat{q}(s,a;\mathbf{w}) \right)^2 \right] \tag{0.3}$$

Target Network DeepMind's paper [2] [1] maintains two sets of parameters, $\mathbf{w}$ (to compute $\hat{q}(s,a)$) and $\mathbf{w}^-$ (target network, to compute $\hat{q}(s',a')$) such that our update rule becomes

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}^-\right) - \hat{q}\left(s, a; \mathbf{w}\right) \right) \nabla_{\mathbf{w}} \hat{q}(s, a; \mathbf{w}). \tag{0.4}$$

and the corresponding optimization objective becomes

$$L^-(\mathbf{w}) = \mathop{\mathbb{E}}_{s,a,r,s' \sim \mathcal{D}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}} \hat{q}\left(s', a'; \mathbf{w}^-\right) - \hat{q}(s, a; \mathbf{w}) \right)^2 \right] \tag{0.5}$$

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 [2] [1] 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 project, 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 $(80 \times 80 \times 4)$.

Now for the question part, we will now examine these assumptions and implement the ϵ-greedy strategy.

(a) (**written** 5 pts) What is one benefit of representing the Q function as $\hat{q}(s; \mathbf{w}) \in \mathbb{R}^{|\mathcal{A}|}$?

(b) (**coding/written** 5 pts) Implement the `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running `python q1_schedule.py`. Report if the test pass in the written portion and if you conducted any extra tests.

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.

(c) (**written** 5 pts) Consider the first of these two extremes: standard Q-Learning without a target network, whose weight update is given by equation **(0.2)** 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 **(0.3)**? Argue mathematically why or why not.

(d) (**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 **(0.4)** 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 **(0.5)**? Argue mathematically why or why not.

3. (20 points) **Linear Approximation and DQN**

We will now implement and test a linear approximator and a DQN approximator.

(a) (**coding/written**, 10 pts) First, we implement linear approximation in PyTorch. This question will set up the pipeline for the remainder of the project. You'll need to implement the following functions in `q3_linear.py` (please read through `q3_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 q3_linear.py` **locally on CPU**. This will run linear approximation with PyTorch on the test environment from Question 1. Running this implementation should only take a minute or two. 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.

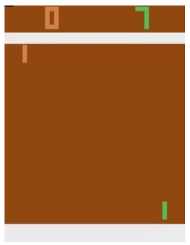
(b) (**coding/written**, 10 pts) Implement the deep Q-network as described in [2] by implementing `get_q_ values_op` in `q4_nature.py`. The rest of the code inherits from what you wrote for linear approximation. Test your implementation **locally on CPU** on the test environment by running `python q4_nature.py`. Running this implementation should only take a minute or two. 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?

4. (20 points) **(Testing on Atari)**

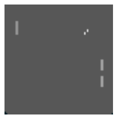
The Atari environment from OpenAI gym returns observations (or original frames) of size $(210 \times 160 \times 3)$, the last dimension corresponds to the RGB channels filled with values between 0 and 255 (`uint8`). Following DeepMind's paper [2], we will apply some preprocessing to the observations:

- 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 2)

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 [2] 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 2: `Pong-v0` environment

(a) (**coding/written**, 10 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`. This will train the model for 500,000 steps and should take approximately a few hours on CPU. Briefly qualitatively describe how your agent's performance changes over the course of training. Also plot the total reward the agent achieves as it trains and present it in your report. Do you think that training for a larger number of steps would likely yield further improvements in performance? Explain your answer.

(b) (**coding/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` (preferably on a GPU). We have trained the model for a few million steps and this is the model that is initially loaded. Feel free to continue the training and improve the model. Attach the plot `scores.png` from the directory `results/q6_train_atari_nature` to your writeup. Also check the `results/q6_train_atari_nature/monitor` folder for sample videos of the agent playing at various stages of training. The DeepMind paper claims average human performance is −3. The trained model performs around −11. See the files `SamplePerformanceBeforeTraining.mp4`, `SamplePerformanceAfterAFewMillionTraining.mp4` that show what the performance was initially and after training for a few million iterations. Please note that to continue the training to perform past human-level performance, you still need to train this for another half day to a day on a GPU. Report your plots and observations.

5. (30 points) **(Train Your Best Model)** Now that you have trained the DeepMind's architecture, feel free to implement any other method(s) and report the best set of results you can achieve on the game of pong. A big portion of the grade associated with part of the project will be graded based on the relative performance achieved by other students in the class. In your report, clearly describe the method(s) that you implemented and the performance you achieved on the best method. Also includes plots for the total reward obtained as the agent learns.

References

[1] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. 2013.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.