CS60077 Reinforcement Learning 2021: Homework-2

Due date:
November 06, 2021 at 11:59 PM

This homework is predominantly a coding assignment and there are no theory questions. We have provided a starter code to help you. Note that you have to fill only in the appropriate spaces given in the starter code. Do not alter any other portion of the code. Read the instructions carefully **both in this document AND the starter codes given to you.**

Submission:

1. **The entire code including the starter code and the files you have edited.**
2. **A writeup answering the questions in the assignment and showing the plots.**

Include everything in a folder and submit a single zip file with the name `<Roll No.>_HW2.zip`.

Introduction

In this homework you 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. We are expecting that you will have some familiarity with PyTorch (the reason that we kept Deep Learning as a pre-requisite).
We are aware that not everyone of you are equipped with compute resources to train deep RL algorithms on complex environments and we want you to utilise your allotted GCP credits for your project. So, instead of using an Atari Environment (well, which would have been fun!), we are providing a simpler test environment which you can easily run locally on CPU.

Test Environment

We are providing you with this simple test environment for your code. 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: $s_0 = 0, a_0 = 1, R_0 = -0.1, s_1 = 1, a_1 = 2, R_1 = 0.0, s_2 = 2, a_2 = 4, R_2 = 0.0, s_3 = 2, a_3 = 3, R_3 = 3.0, s_4 = 3, a_4 = 0, R_4 = 0.2, s_5 = 0$.

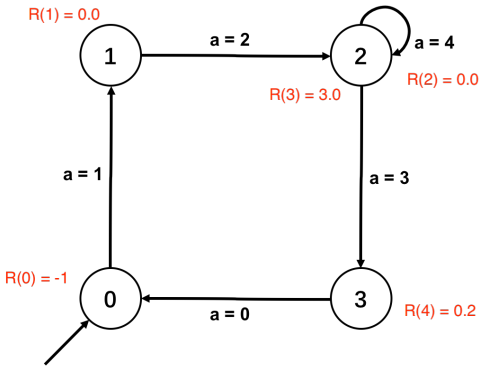

Figure 1: Example of a trajectory in the Test Environment

1 Tabular Q-Learning (5 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.
Implement the `get_action` and `update` functions in `q1_schedule.py`. Test your implementation by running `python q1_schedule.py`.

2 Q-Learning with Function Approximation (15 points)

Due to the scale of real-world 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:

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

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

We will now examine some implementation details.

2.1 Linear Approximation (10 pts)

We will now implement linear approximation in PyTorch. This question will set up the pipeline for next part of your Assignment. You'll need to implement the following functions in `q2_1_linear_torch.py` (please read through `q2_1_linear_torch.py`):

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

Test your code by running `python q2_1_linear_torch.py` locally on CPU. This will run linear approximation with PyTorch on the test environment from Problem 1. Running this implementation should only take a minute.
Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q2_1` linear to your writeup.

2.2 Implementing DeepMind's DQN (5 pts)

Implement the deep Q-network as described in [1] by implementing `initialize_models` and `get_q_values` in `q2_2_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 q2_2_nature_torch.py`. Running this implementation should only take a minute or two.
Attach the plot of scores, `scores.png`, from the directory `results/q2_2_nature` to your writeup. Compare this model with linear approximation. How do the final performances compare? How about the training time?

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.