CS 4/5789 - Programming Assignment 3

March 15, 2024

This assignment is due on **March 29, 2024** at 11:59pm.

Section 0: Requirements

Please use Anaconda and create a new environment with the following packages:

- `conda install -c conda-forge gym==0.21.0`
- `conda install -c conda-forge pyglet==1.5.27`
- `conda install -c conda-forge box2d-py==2.3.8`
- `conda install pytorch==1.13.1`
- `conda install -c conda-forge tqdm==4.65.0`

If you have issues setting up your environment because of the specificity of your machine, we recommend setting up a Google Colab notebook. Feel free to clone the following Colab notebook. It is limited and only contains the commands necessary for setup, training, testing and visualization. To run your code, upload the project files directly. Note that the files will be removed when the runtime expires so make sure to save your work. The compute resources given by the free version of Google Colab are limited so training will likely take longer than on your local computer. This should be run without any hardware acceleration (the hardware acceleration should be None).

Section 1: Introduction

For this assignment, we will ask you to implement a Deep Q Network (DQN) to solve a cartpole environment (where you have to balance a pole on a cart) and a lunar lander environment (where you try to land a spaceship between two flags on the moon). We will be using CartPole-v0 and LunarLander-v2 for this assignment. In case you need a refresher on how Gym environments work in general, we invite you to check out this tutorial by Wen-Ding, a graduate TA of CS 4789 in Spring 2021.

Note: It is advised that students iterate or write code on the colab first. If a session is ended on google colab, progress is lost. You should write code locally (i.e. debug there) and then run the training/testing on google colab.

This assignment is worth **60 points** in total.

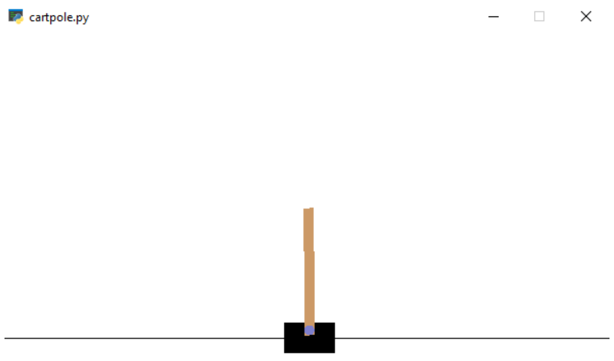

Figure 1: The CartPole environment.

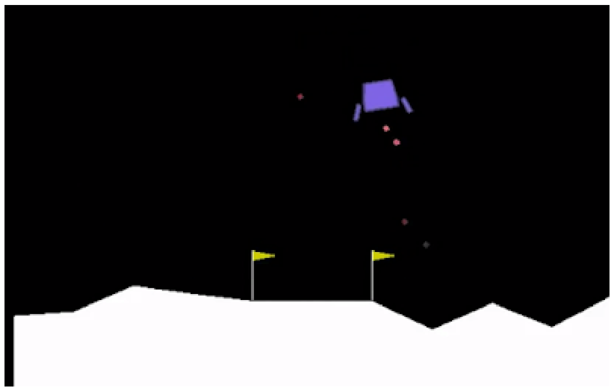

Figure 2: The LunarLander environment.

There are several files in this project.

- `network.py`: This contains code to implement the Q network agent. This Q network is our "policy" that we use to take actions in our environment.
- `utils.py`: This file is for general utilities needed for our experiments.
- `buffer.py`: This contains code implementing the replay buffer used throughout training. This replay buffer will store the transitions that we collect online.
- `train.py`: This contains the standard training loop. We train and save agents by running this file.
- `test.py`: Test file. Run this with `python test.py`. These file contains a test method where you can see how your trained agent does in the environment. You can change the arguments when you run the test file in order to test in different environments.

Section 2: Code Deliverables

There are 4 files that you will modify in this programming assignment:

- `network.py`: You will modify the `get_max_q`, `get_action`, and `get_targets` methods.
- `utils.py`: You will modify the `update_target` method.
- `buffer.py`: You will implement the replay buffer used in training.
- `train.py`: You will modify the training script to train your DQN.

The `TODOs` specify the method expectations.

Section 3: Deep Q Network

In class, we have touched on Q-learning, an off-policy reinforcement learning algorithm that aims to find the optimal Q value function $Q^\star$ by minimizing the difference between the Q values $Q(s,a)$ and the one-step Bellman optimal Q values $r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a'} Q(s',a')\right]$. This is because the optimal Q function $Q^\star$ satisfies

$$Q^\star(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a'} Q^\star(s',a')\right],$$

for all (s,a), and so minimizing the difference will hopefully bring us closer to finding the optimal Q function. At a terminal state, by the Bellman equations in the finite horizon setting, the value function, and thus the Q function, at all states s and actions a are equal to 0. The terminal states in CartPole are if we keep the pole balancing for 200 timesteps and when the angle between the pole and the cart gets below some angular threshold. In LunarLander, the terminal states are when the lander flies off of the screen, crashes to the moon's surface or safely lands.

In particular, **tabular Q-learning** keeps a table/matrix of Q values $T_Q \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{A}|}$, and at every iteration of training, given a transition (s,a,r,s'), our Q update is as follows:

$$T_Q(s,a) \leftarrow T_Q(s,a) + \alpha \left(r + \gamma \max_{a'} T_Q(s',a') - T_Q(s,a)\right).$$

Here, α is a learning rate parameter that regulates how much we change our Q value at each iteration.

The main bottleneck in tabular Q-learning algorithms is that when the number of states $|\mathcal{S}|$ and the number of actions $|\mathcal{A}|$ are very large, our table ends up being gigantic, and in complex environments even infeasible to keep in memory. This is particularly the case in Atari games, where the number of states (and the state size) were so large that any tabular Q-learning method would take an insanely long time (longer than the average human life expectancy, assuming we have infinite memory) to converge and spit out an optimal Q function.

How do we fix this issue? Researchers at DeepMind noted that we can do away with the Q table entirely and replace it with a deep neural network. In particular, this neural network specifies a function $f : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{A}|}$ that takes in a state s and returns the Q values of all the actions $a \in |\mathcal{A}|$. This led to the introduction of the **deep Q network (DQN)**, which showed amazing results on the Atari game suite.

In this assignment, we will be working in simpler environments with discrete action spaces, so $\mathcal{A}$ is finite in size.

3.1 DQN update scheme

We can treat the Q-learning scheme as an optimization problem, where, once again, we aim to minimize the difference between the Q function at a given timestep and the one-step Bellman optimal values at the next step. In particular, if we specify our Q function by a set of parameters θ (given by our neural network), we can write our one step Bellman optimal targets as

$$y(s,a) = r(s,a) + \gamma \mathbb{E}_{s' \sim P(s,a)} \left[\max_{a'} Q_\theta(s',a')\right],$$

and write our optimization problem as

$$\theta^\star = \arg\min_\theta \mathbb{E}_{s,a} \left(Q_\theta(s,a) - y(s,a)\right)^2.$$

We can do this optimization via **gradient descent**, where we treat our optimization problem as a loss function with respect to θ and optimize. In particular, if we let

$$\mathcal{L}(\theta) = \mathbb{E}_{s,a} \left(Q_\theta(s,a) - y(s,a)\right)^2$$

be our loss function, we can write our gradient descent update at timestep t as

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta \mathcal{L}(\theta_t).$$

3.2 Ingredients for update scheme

There are a lot of terms that we need to compute in order to do our optimization. The major term that we need to compute are our one-step Bellman optimal **targets** $r(s, a) + \gamma \mathbb{E}_{s' \sim P(s,a)} [\max_{a'} Q_\theta(s', a')]$.

TODO: Modify the `get_max_q` function in `network.py` in order to return the max Q value given next states s'. This method should take in a PyTorch tensor of shape (B, D_s) as input, where B is the batch size and D_s is the state dimension, and return a PyTorch tensor of size B, where each entry in the output is the maximum Q value of each state in the batch.

Once we compute our max Q values, we can compute our targets.

TODO: Modify the `get_targets` function in `network.py`, which, given rewards r, next states s' and terminal signals d, computes the target for our Q function. See the `forward` method for additional info. Note that if s' is a terminal state (given by the done flag), the Q function should return 0 by definition. All inputs are PyTorch tensors, and the rewards are of shape B, the next states are of shape (B, D_s), and the terminal signals are of shape B, where B is the batch size. Your function should return a tensor of size B.

Computing the action at every timestep

So far, we have assumed that we have (s, a, r, s', d) tuples ready to do our update. However, given that reinforcement learning is fundamentally *interactive*, we have to actually get these transitions by rolling out our policy while training.

One of the main challenges in reinforcement learning is that of **exploration vs. exploitation**, where at every timestep the agent chooses whether to "explore" the environment by taking a random action or "exploit" it by taking the action that is optimal with respect to its current Q function. The way that most people do this is via an **ϵ-greedy** policy, where with probability ϵ we take a random action (explore), otherwise we take the current optimal action (exploit). Naturally, during training, we decrease ϵ to lean more and more on our learned Q function.

TODO: Modify the `get_action` method, which, given a state s and exploration parameter ϵ, returns an action a that the policy prescribes. Return the action as a NumPy array or an integer.

3.3 Storing our transitions

During training, we want to store our experience in a replay buffer and sample experience every time we need to do a Q update. In particular, we want to store transitions of (s, a, r, s', d) pairs in a replay buffer.

TODO: Modify the methods in `buffer.py` to store and sample transitions. Keep in mind that we store and sample PyTorch tensors from the buffer. Check out what we imported at the top of the file in order to best do these parts.

3.4 Stability of training

Deep Q networks, while comparable to neural networks used in supervised learning, have a slight hitch that makes them more difficult to train: the targets they are predicting are based off of the current Q function estimates. This makes training harder (think about it as a dog trying to chase its own tail). In this vein, to get closer to supervised learning, we initialize a **target network** $Q_{\bar{\theta}}$ with parameters $\bar{\theta}$, which gets updated every few episodes of training as follows via a **soft update**:

$$\bar{\theta} \leftarrow (1 - \tau)\bar{\theta} + \tau\theta,$$

where τ is a temperature hyperparameter that controls how close $\bar{\theta}$ gets to our online Q network parameters θ.

TODO: Modify the `update_target` method in `utils.py` to update the target network. The method will take in the target network (of type `torch.nn.Module`), the current network (also of type `torch.nn.Module`), and the temperature parameter τ (a float).

3.5 Training

We are finally ready to move on to training our Q network.

TODO: Fill out the TODO sections in `train.py`. To learn about optimizers and training a neural network in PyTorch, see the tutorials at this link.

When you are ready to test, your numbers will likely have some variation. However, the average episode length and reward should both be 200.0 for the CartPole environment. For the LunarLander environment, the expected episode length of a working policy is around 400.0, and the reward incurred should be around 50.0. We encourage you to see if you can achieve even better rewards.

3.6 Report

Once finishing the training loop, run the following commands:

- `python train.py --target --hidden_sizes 128` for CartPole-v0
- `!python train.py --env LunarLander-v2 --target --hidden_sizes 64 64 --num_episodes 1500 --tau 1e-2 --batch_size 64 --gamma 0.99 --eps 0.995 --learning_freq 4 --target_update _freq 4 --max_size 100000` for LunarLander-v2

For LunarLander-v2, this could take a long time (on a CPU, it can take around 10 minutes). If needed, definitely port your code to Google Colab.

Report Deliverable 1. Report your mean rewards and mean episode lengths for both environments, using the given choices of hyperparameters. This can be done by running `python test.py` with the correct arguments in the main method. See the commented code in the main method to see how to best run your agents in the correct environments.

Report Deliverable 2. Please also report your methods and findings in the following areas (if you have limited compute power, definitely prioritize working with CartPole to answer these questions, but for full credit please provide answers with both environments):

- How important is training with the target network? Can you get similar performance by just removing the target network? If you get similar (or even better) performance, what do you think is the reason why? Note that in order to disable training with the target network, you can remove the `--target` argument from the training command.
- Does updating the target network more than once every episode (i.e. moving the call to `update_targets` inside the episode `for` loop) improve performance?
- Can you find a hyperparameter setting whose performance exceeds the setting we provided in the above training commands?

3.7 Submission

Please submit a zip file containing the following files (make sure that they are at the top level of your submission, i.e. not contained within any inner folders):

```
submission.zip
├── Answers.pdf
├── network.py
├── utils.py
├── buffer.py
├── train.py
├── test.py
├── trained_agent_cartpole-v0.pt
└── trained_agent_lunarlander-v2.pt
```

where `Answers.pdf` contains the Report Deliverables from section 3.6, and `trained_agent_cartpole-v0.pt` and `trained_agent_lunarlander-v2.pt` are the models obtained from training with the default hyperparameters.