CAS4160 Homework 3: Q-Learning

[JongwookJeon]
[2020142139]

1 Introduction

The goal of this assignment is to help you understand **Q-learning methods**, including Deep Q-Network (DQN) and Double DQN.

The first part of this assignment includes quizzes about DQN. Then, in the second part of the assignment, you will implement a working version of Q-learning and evaluate Q-learning for playing Atari games. Our code will work with both state-based environments, where the input is a low-dimensional list of numbers (like Cartpole), but we will also support learning directly from pixels (like BankHeist)!

This assignment will be faster to run on a GPU, though it is still possible to complete on a CPU as well. Therefore, we recommend using VESSL AI or Colab if you do not have a GPU available to you. Section **4.2** takes about 15 minutes with a GPU and 30 minutes without it, while Section **5.2** requires approximately 12 hours with a GPU and 24 hours without it in total.

2 Deep Q-Network Quiz

Let's review what we have learned in the lecture about Deep Q-Network (DQN). Answer the following True/False questions:

I. Q-Learning cannot leverage off-policy samples, resulting in poor sample efficiency.

- ☐ True
- ■ False

  Q-learning is one of the off-policy algorithms

II. Without an actor, evaluating Q-values for all possible actions is infeasible with continuous action space.

- ■ True
- ☐ False

III. One of the main challenges in DQN is the moving target, which happens when the agent estimates Q-values and target value using the same neural network. To avoid this, we can use the fixed target network within an inner loop.

- ■ True
- ☐ False

IV. We often use epsilon scheduling to encourage more exploration over time.

- ☐ True
- ■ False

  We often use epsilon scheduling to encourage less exploration over time.

3 Code Structure Overview

The training begins with the script `run_hw3.py`. This script contains the function `run_training_loop`, where the training, evaluation, and logging happens.

You will implement a DQN agent, `DQNAgent`, in `cas4160/agents/dqn_agent.py` and a DQN training loop in `cas4160/scripts/run_hw3.py`. In addition, you should start by reading the following files thoroughly:

- `cas4160/env_configs/dqn_basic_config.py`: builds networks and generates configuration for the basic DQN problems (`CartPole-v1`, `LunarLander-v2`).
- `cas4160/env_configs/dqn_atari_config.py`: builds networks and generates configuration for the Atari DQN problems (`Breakout`, `BankHeist-v5`).
- `cas4160/infrastructure/replay_buffer.py`: implementation of replay buffer. To efficiently store and sample observations with frame-stack in DQN, we will use `MemoryEfficientReplayBuffer`. Try to understand what each method does (particularly `insert`, which is called after a frame, and `on_reset`, which inserts the first observation from a trajectory) and how it differs from the regular replay buffer.
- `cas4160/infrastructure/atari_wrappers.py`: contains some wrappers specific to the Atari environments. These wrappers can be key to getting challenging Atari environments to work!

3.1 Important Implementation Tricks

The starter code include a few implementation tricks to stabilize training. You do not need to do anything to enable these, but you should look at the implementations and think about why they work.

- **Exploration scheduling for ϵ-greedy actor.** This starts ϵ at a high value, close to random sampling, and decays it to a small value during training (`exploration_schedule` in `cas4160/scripts/run_hw3.py`).
- **Learning rate scheduling.** Decay the learning rate from a high initial value to a lower value at the end of training (`DQNAgent.lr_scheduler`).
- **Gradient clipping.** If the gradient norm is larger than a threshold, scale the gradients down so that the norm is equal to the threshold (`DQNAgent.update_critic`).
- **Atari wrappers.** (in `cas4160/infrastructure/atari_wrappers.py`)
  - **Grayscale.** Convert RGB images $(84 \times 84 \times 3)$ to grayscale images (84×84).
  - **Frame-skip.** Keep the same constant action for 4 steps and ignore intermediate inputs.
  - **Frame-stack.** Stack the last 4 grayscale frames to use as the input $(84 \times 84 \times 4)$.

4 Deep Q-Learning

4.1 Implementation

Implement the basic DQN algorithm. You will implement an update for the Q-network and a target network, as well as functions for ϵ-greedy sampling and collecting trajectories:

- Implement a DQN critic update in `update_critic` function by filling in the unimplemented sections (marked with TODO(student)) in `cas4160/agents/dqn_agent.py`.

- Implement `update` of `cas4160/agents/dqn_agent.py`, which calls `update_critic` and updates the target critic, if necessary.
- Implement ϵ-greedy sampling in `get_action` function in `cas4160/agents/dqn_agent.py`.
- Implement the TODOs in `cas4160/scripts/run_hw3.py`.
- Implement the TODOs in `sample_trajectory` function of `cas4160/infrastructure/utils.py`.

  Hint: A trajectory can end in two ways: the actual end of the trajectory (`terminated=True`, usually triggered by catastrophic failure, like crashing), or *truncation* (`truncated=True`), where the trajectory does not actually end but we stop simulation for some reason (e.g. exceeding the maximum episode length).

  In `gymnasium`, there are two corresponding boolean values among the return items of `env.step`. Here, `terminated` flag represents the actual end of the trajectory, whereas `truncated` flag represents the truncation of trajectory due to other reasons, including reaching the maximum episode length. In this latter case, you should still reset the environment, but the `done` flag for TD-updates (stored in the replay buffer) should be `False`.

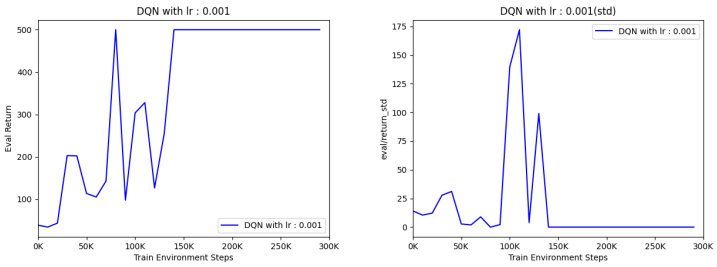
4.2 Experiment

DQN-CartPole. Test your DQN implementation on `CartPole-v1` with `experiments/dqn/cartpole.yaml`. It should reach reward of nearly 500 around 300K steps (around **15 minutes**).

```
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/cartpole.yaml --seed 1
```

- Plot the learning curve with environment steps on the x-axis and eval return (`eval_return`) on the y-axis. You can use `cas4160/scripts/parse_tensorboard.py` as in Homework 1 and 2.

  Answer:



(a) eval_return (b) std

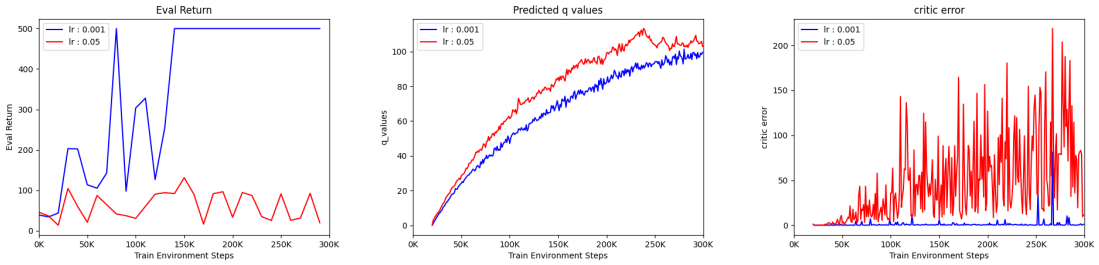
Figure 1: **DQN with lr = 0.001**

The overall y-value, eval_return of this plot is increasing as the x-value, steps increases. At the beginning, the eval_return jumps up and down a lot because the epsilon is still high. This means that the agent has a lot of exploration. This is because of the epsilon scheduler. The returns become much more stable and eventually converge close to the maximum of 500. Since Cartpole only has two discrete actions, it learns quickly with only small fluctuations due to the exploration. In a more complex or continuous action space, it needs more exploration and a slower epsilon decay, so convergence would take longer.

Plotting the standard deviation alongside the mean return helps to show how variable the performance is. At the beginning, it jumps up and down a lot, similar to the eval_return because of the explorations. And eventually the std gets 0 and is stabled.
Finally, choices like replay-buffer size, learning rate scheduler, and target update period all influence how smooth or noisy the learning curve turns out.

- Run DQN on `CartPole-v1`, but create `experiments/dqn/cartpole_lr_5e-2.yaml` by modifying the config file `experiments/dqn/cartpole.yaml` (change the `learning rate` to 0.05, where the default learning rate is 0.001), and run with this config file. What happens to (a) the predicted Q-values, (b) the critic error, and (c) the eval returns? Please provide three plots to compare the results of two different learning rates. Can you relate this to any topics from class? Provide your reasoning/explanation.

Answer:



(a) eval_return (b) predicted q_values (c) critic error

Figure 2: **lr=0.001 vs lr=0.05**

The eval_return between the lr = 0.05 and lr = 0.001 results significantly different. While the lr=0.001's eval_return climb smoothly from 0 to 500 by around 150k steps, the lr=0.05's eval_return barely rise above 100 and has big fluctuations. This is because of the large size of learning rate. When the learning rate is too large, it overshoots the optimum and updates diverge. In DQN, it also makes the moving target issue even worse.
We can see the reason of the result (a) from (b) and (c). At (b), lr = 0.05 predicted q values increases similar with lr =0.001. It seems having good training but it is actually overestimates more aggressively, pushing q values up into the 80-100 range much faster than lr = 0.001. The reason for this can be clearly seen at (c). (c) shows the critic error for lr =0.05 explodes and never settles, whereas for lr = 0.001, critic loss stays tiny after a while.
So although the red network predicts big returns, its predictions are widely off, and the agent never learns a policy that achieves them in the environment. The online network jumps ahead of the target network and make the TD targets meaningless.

5 Double Q-Learning

5.1 Implementation

Let's try to stabilize learning. The double-Q trick avoids overestimation bias in the critic update by using two different networks to *select* the next action $\mathbf{a}'$ and to *estimate* its value:

$$\mathbf{a}' = \arg\max_{\mathbf{a}'} Q_\phi(\mathbf{s}', \mathbf{a}')$$

$$Q_{\text{target}} = r + \gamma(1 - d_t) Q_{\phi'}(\mathbf{s}', \mathbf{a}').$$

In our case, we'll keep using the target network $Q_{\phi'}$ to estimate the action's value, but we'll select the action using Q_ϕ (the online Q-network).

Implement this functionality in `cas4160/agents/dqn_agent.py`.

5.2 Experiments

For this problem, each experiment will take about 2 hours with a GPU (12 hours in total), or 4 hours without (24 hours in total), so start early!

- Run DQN with three different seeds on `BankHeist-v5`:

```
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist.yaml --seed 1
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist.yaml --seed 2
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist.yaml --seed 3
```

  Your returns should improve until around 150.

- Run Double DQN with three seeds on `BankHeist-v5`:

```
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist_ddqn.yaml --
    seed 1
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist_ddqn.yaml --
    seed 2
python cas4160/scripts/run_hw3.py -cfg experiments/dqn/bankheist_ddqn.yaml --
    seed 3
```

  You should expect a return of **300** by the end of training. (***Disclaimer***: *for some seeds, it might not reach the return of* ***300****. That is fine as far as you observe it can outperform the vanilla DQN on average*).

  Plot the **returns** from these **three** seeds of Double DQN in red, and the "vanilla" DQN results in blue, on the same set of axes. Compare DQN and Double DQN, and describe in your own words what might cause this difference.

  Answer:

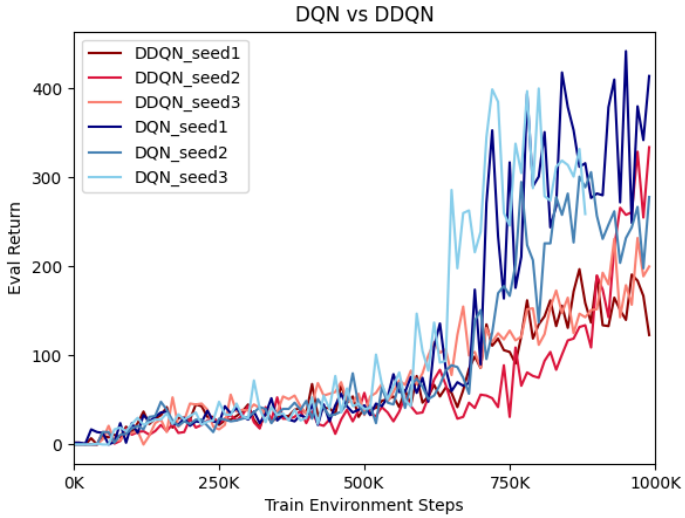


Figure 3: **DQN(blue)vsDDQN(red)**

The difference in result(eval_return) is quite obvious. The DQN(Blue curves) has higher results compared to the DDQN(red curves). The DDQN curves all rise steadily and by 1M steps cluster around a return of $\tilde{3}00$, whereas the DQN curves all over the place some spikes up toward 400 or more, but then big drops and widely different end-points.
This is because of the DQN's overestimating problem. In DQN max operator in TD target tends to latch onto noisy overestimates, so the Q-network updates aggressively, but the resulting policy is unstable and often falls back.
DDQN splits action selection from value estimation, which tames the overoptimistic Q-values and produces more consistent TD targets.
As the result, DDQN reduces max-operator bias and the curves do not oscillate and converge around 300 stably.

6 Experimenting with Hyperparameters

Let's analyze the sensitivity of Q-learning to hyperparameters on the `CartPole-v1` environment. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the default value, and plot eval returns with all four values on the same graph. Explain why you chose this hyperparameter in the caption. Create four config files in `experiments/dqn/hyperparameters` (refer to `cas4160/env_configs/basic_dqn_config.py` to see which hyperparameters you are able to change). You can use any of the base YAML files as a reference.

Hyperparameter options could include:

- Learning rate
- Network architecture
- Exploration schedule (or, if you'd like, you can implement an alternative to ϵ-greedy)

Answer:

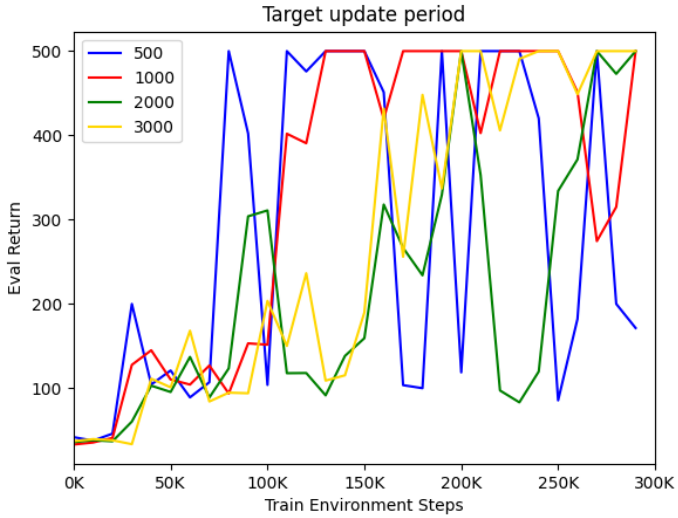


Figure 4: **Target update period : [500, 1000, 2000, 3000]**

I chose hyperparameter as the target update period. I compared the result of eval_return changing how often it copies the Q network into the target network every 500, 1000, 2000, 3000 steps.
In the period as 500 steps (blue curve), we can see a quick jump up toward 500 returns by around 100K steps, but it also collapses back down several times. This can means too much chasing the moving target makes the training noisy. the period as 1000 steps (red curve), we can see it still learns fast and reaches 500, but has fewer catastrophic drops, so it is overall more stable.
In the period as 2000 steps (green curve), the agent takes longer to climb out of the 100–200 return range, and it only finally reaches the top after 150–200k steps.
In the period as 3000 steps (gold curve), the agent takes much longer time to reach the maximum return. To have a good performance, we need a balance. If the update of the target network is too frequently, it can perform instability by letting the moving target shift with every mini-batch. If the update is too rare, TD target become stable and learning grinds to a crawl. In CartPole environment, 1000-step update period gave the best trade-off between speed and stability.

7 Discussion

Please provide us a rough estimate, in hours, for each problem, how much time you spent. This will help us calibrate the difficulty for future homework.

- **Quizzes**: **00 hours**
- **Deep Q-Learning**: **03 hours**
- **Double Q-Learning**: **10 hours**
- **Experimenting with Hyperparameters**: **02 hours**

Feel free to share your feedback here, if any: **We would really appreciate your feedback to improve the reinforcement learning class.**

8 Submission

Please submit the code, tensorboard logs, and the **"report"** in a single zip file, `hw3_[YourStudentID].zip`. Do not include videos as the file size should be less than 50MB. The structure of the submission file should be: