Deep Reinforcement Learning
Assignment 2

2024 Spring

March 28, 2024

In this homework, you will the DQN [3] algorithm and several variations of it to solve the CartPole-v1 environment from Gymnasium.

1 About the Environment

The CartPole-v1 environment is a classic control problem where the agent controls a cart moving along a frictionless track. It has a continuous observation space containing information on cart position, cart velocity, pole angle, and pole angular velocity. The agent has a discrete action space of two actions: move left or move right.

The environment is **terminated** when the pole is more than 12 degrees from vertical, or the cart moves more than 2.4 units from the center, or **truncated** when the episode length is greater than 500. Please refer to the Documentation for more details.

2 About the Framework

In this homework, we are using the Hydra framework to manage the configuration of the experiments. Hydra is a framework for elegantly managing your hyperparameters and training results. It provides a simple interface for organizing and overriding your configuration. It also provides a powerful search space specification language that allows you to easily define and explore the hyperparameter space of your configuration, as well as to store the training results and output files in separate folders automatically for you to trace them later. Please refer to the Documentation for more details.

3 Requirements

Since it's hard to cover all the cases in automatic grading (due to the randomness of neural networks and sampling procedure), we will combine the automatic grading with manual grading, which requires you to not only pass the test cases but also provide a brief report on your results and findings. The total score of the test cases is 30 pts, while the report will be graded up to 70 pts. Please submit a zip file containing the following files on Web Learning:

- The auto-generated `runs` folder which contains the results for each of the required experiments and their corresponding config, log, and model files. You may remove the generated videos but leave the other files untouched. Please make sure the code you provide can reproduce the contents of the `runs` folder given the corresponding config file.

- A report.pdf file briefly summarizing your results. It should include the auto-generated results figure. You may also include any other findings you think are relevant.

You can find the reference training curves in the `gallery` folder. We don't require your results to be exactly the same as ours, but if they are drastically different from the reference curves, there may be something wrong with your implementation. Please make sure the best results of your agent converge to the maximum reward of 500 for each method.

Please refer to the `README.md` file for dependencies of this homework.

4 Deep Q-Network [30 pts]

In this section, you will implement a DQN agent and solve the CartPole-v1 environment. You'll need to read through the following files and implement the missing parts:

model.py This file contains the neural network models that we use to approximate the Q function. Read through the `Qnetwork` class to get a sense of how the network is structured. Note how we use the `instantiate` method of `hydra` to specify a class object from the config file. You don't need to implement anything in this file for this section.

buffer.py This file contains the replay buffer classes we use to store trajectories and sample mini-batches from them. In this section, read through the `add` and `sample` methods of the `ReplayBuffer` class, you don't need to implement anything for this section.

agent.py This file contains the core `DQNAgent` class that manages how to take actions and how to update the network. In this section, you'll need to implement the `get_action`, `get_Q_target`, and `get_Q` methods of the `DQNAgent` class. You can read through the `update` method to get a sense of how the agent does a one-step update. You don't need to implement anything under the `if self.use_double` condition for this section.

utils.py This file implements a handful of tool functions we use in the training process. Please implement the `get_epsilon` function used for the epsilon-greedy exploration. You can also read through the other functions, especially the `set_seed_everywhere` function, to get a sense of how we set the random seed for the experiment.

core.py This file contains the main training and evaluation loop. You don't need to implement anything in this file. Read through the `train` and `eval` functions to get a sense of how the training and evaluation process is structured.

main.py This is the main file that you'll run to start the training process. You don't need to implement anything in this file. Note we use the `hydra.main` decorator to specify the config file for the experiment.

After implementing the necessary parts of the DQN agent, you can run the following command to start the training process:

```
python main.py
```

The results and saved files will be stored in the `runs` folder, under the subfolder specified by the time of execution. You can find the training curves and a video of the trained agent in the subfolder. If you want to turn off this behavior and save everything in the current folder, you can change the `hydra.run.chdir` field in the `config.yaml` file to `false`.

5 Double DQN [10 pts]

To improve the stability of the DQN algorithm, we can use the **Double DQN** algorithm [1]. The main idea of Double DQN is to use the original Q-network to select actions while using the target network to estimate the next Q-values.

$$Y_i = R_i + \gamma Q\left(s_i', \arg\max_a Q\left(s_i', a; w\right); w^-\right)$$

In this section, please read through the following files and implement the missing parts:

agent.py Implement the `get_Q_target` function under the `if self.use_double` condition.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py agent.use_double=true
```

This will override the default config of the `agent.use_double` field to `True`.

6 Dueling DQN [10 pts]

In this section, you will implement the Dueling DQN algorithm [5], which is another extension of the DQN algorithm that can improve its performance by separating the Q-value estimation into two streams: one for estimating the state value function and the other for estimating the advantage function. Please read through the following files and implement the missing parts:

model.py We have defined the architecture of the Dueling network. Please implement the `forward` function of the `DuelingQnetwork` class.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py agent.use_dueling=true
```

7 Prioritized Experience Replay [10 pts]

This section covers the Prioritized Experience Replay (PER) algorithm [4], another extension of the DQN algorithm that can improve its performance by using a prioritized replay buffer based on the TD error of the transition:

$$\delta_i = |Q\left(s_i, a_i; w\right) - Y_i|$$
$$p_i = (\delta_i + \epsilon)^\alpha$$

where ϵ is a small constant to avoid zero priority, α is a hyperparameter that controls the degree of prioritization. The probability of sampling a transition is proportional to its priority:

$$P(i) = \frac{p_i}{\sum_j p_j}$$

Prioritized replay introduces bias because it doesn't sample experiences uniformly at random due to the sampling proportion corresponding to TD error. We can correct this bias by using importance sampling weights:

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}$$

where N is the size of the replay buffer and β is a hyperparameter that controls the degree of importance sampling. w_i fully compensates for the non-uniform probabilities if $\beta = 1$. These weights can be folded into the Q-learning update by using $w_i\delta_i$ instead of δ_i.

For stability reasons, we always normalize weights by $\max(w_i)$.

$$w_i = \frac{w_i}{\max(w_i)}$$

In practice, we don't update the priorities of the transitions in the entire replay buffer. Instead, we store a new transition to the transition buffer with the maximum priority and update the priorities of the sampled transitions in the replay buffer after each update.

Please read through the following files and implement the missing parts:

buffer.py Read the `__init__` function for the `PrioritizedReplayBuffer` class. Implement the `sample` function of the class, and read the `update_priorities` function to get a sense of how the priorities are updated after each sample.

core.py Read the relevant parts of the `train` function (under the `isinstance(buffer, PrioritizedReplayBuffer)` condition) to get a sense of how the PER algorithm is implemented in the train loop.

After implementing the necessary parts, you can run the following command to start the training process:

```
python main.py buffer.use_per=true
```

8 N-Step Return [10 pts]

In this section, you will implement the N-step return algorithm, which is another extension of the DQN algorithm that can improve its performance by using an n-step estimation for the training target to reduce the bias of that estimation. There's no direct reference for this method, but you can refer to [2] for a detailed discussion. The N-step return is defined as:

$$Y_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k R_{t+k+1} + \gamma^n \max_a Q\left(s_{t+n}, a; w^-\right)$$

In practice, we can use an n-step buffer to store the last n transitions. The state and next state of the current transition are the first and last states in the buffer, respectively. The reward of the current transition will be the discounted sum of the rewards in the buffer.

Please read through the following files and implement the missing parts (**10 pts**):

buffer.py Read the `__init__` and `add` functions for the `NStepReplayBuffer` class. Implement the `n_step_handler` function of the class.

agent.py Read the `__init__` function of the `DQNAgent` class again and notice how we handle the discount factor for n-step DQN.

The N-Step return can be used along with the prioritized replay buffer. It's OPTIONAL for you to read through the following files and implement the missing parts:

buffer.py Find a convinient way to implement the `PrioritizedNStepReplayBuffer` class, choose either to inherit the `PrioritizedReplayBuffer` or the `NStepReplayBuffer` class. Implement the `add` function of the class. Some of the class methods are the same as the two classes, so you can reuse them.

After implementing the necessary parts, you can run the following command to start the training with an n-step replay buffer(replace the n with the actual number):

```
python main.py buffer.nstep=n
```

You can run the following command to train with a prioritized n-step replay buffer:

```
python main.py buffer.use_per=true buffer.nstep=n
```

References

[1] H. V. Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *ArXiv*, abs/1509.06461, 2015.

[2] J. Fernando Hernandez-Garcia and Richard S. Sutton. Understanding multi-step deep reinforcement learning: A systematic study of the dqn target. *ArXiv*, abs/1901.07510, 2019.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

[4] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[5] Ziyun Wang, Tom Schaul, Matteo Hessel, H. V. Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *ArXiv*, abs/1511.06581, 2015.