DS 669
Reinforcement Learning
Programming Assignment #3

In this assignment, you will work with the `CartPole-v1` environment from the Gymnasium library and practice building and training neural networks using PyTorch. You will implement the Deep Q-Learning method and conduct experiments to explore the effects of different parameter settings, helping you better understand how these parameters influence the performance of the model.

Setup: You need a GPU, a CUDA environment, and PyTorch to complete this assignment. Please install PyTorch according to the official instructions. If your computer does not have a GPU or does not support CUDA, you can run your code on a CPU. However, we highly recommend you configuring the environment to use a GPU, as using a CPU may be very slow for the following assignments.

Submission:

- You need to complete the assignment by addressing all parts in each section. If a task is marked as **Question**, please provide a clear and detailed answer in your own words. If a task requires screenshots or plots, ensure they fully cover all the answers.
- Make sure to number each of your responses to easily match them with the corresponding exercises.
- When finished, submit your report in **PDF** form, `codebase_main.py`, `codebase_CartPole_test.py`, `codebase_torch_practice.py`, and `codebase_DQN.py` file on Canvas. **Please DO NOT** zip all your files together.

I Getting started with Training Neural Network in PyTorch (20%)

In this part, you will learn how to train a Neural Network in PyTorch and understand the effect of parameters. Please read through `codebase_torch_practice.py`, and `get_args.py`.

- In this section, we will only use the most essential components of PyTorch. If you have previous experience in PyTorch development, you can skip the learning phase and start coding with the provided instructions.
- Please read the tutorial on PyTorch.
- For completing this assignment, you do not need to thoroughly learn PyTorch. However, please familiarize yourself with the following components in PyTorch: (1) Tensor (2) Optimizer (3) Linear (4) ReLu (4) Loss. The `torchvision` library is not required for this assignment.
- We highly recommend you search a very simple PyTorch neural network example code, run it on your computer to verify the installation, and get familiar with the mentioned components.

(a) In this sub-question, please read and understand the code in the `Net` and `Practice` class in `torch_practice.py`. The `Net` class specifies the components and architecture of a neural network using PyTorch. The `Practice` class uses the `Net` to instantiate a network. It also defines the optimizer and loss function. That's where we train the network with input data.

- Keep all settings as default, `hidden_dim` as 10, `num_hidden` as 0, and learning rate `lr` as 0.001, and set the method in `get_args.py` as `test-PyTorch`. Run in `main.py` and a plot of the loss in 1000 epochs will be generated.

(1) Please paste the generated plot of the loss.

(2) **Question**: At approximately which epoch(200, 400, or 600) does the MSE loss converge? (**Hint:** A model is said to have converged when its loss function reaches a stable minimum or when the performance metrics stabilize.)

(b) Change the number of hidden layers from 0 to 9, and rerun the code.

(1) Please paste the plot of the loss.

(2) **Question**: At approximately which epoch does the MSE loss converge? Does increasing the number of hidden layers help the network converge faster?

(3) **Question**: Do you observe a period at the beginning of training where the loss does not decrease significantly? If so, can you briefly explain why increasing the number of layers might lead to this observation?

(c) Reset the number of hidden layers from 9 to 0 and change the hidden layer dimension `hidden_dim` from 10 to 100. Rerun the code and provide the plot of loss.

(1) **Question**: At approximately which epoch does the MSE loss converge? Does increasing the dimensions of the hidden layer help the network converge faster?

(d) Restore the hidden layer dimension to 10 and set the learning rate `lr` to 0.1. Rerun the code and paste the plot of the loss.

(1) **Question**: At approximately which step does the model converge to the lowest MSE? Does increasing the learning rate cause the network to converge faster? Can you briefly explain why?

Remember to set the learning rate as 0.01 for the rest part of this assignment.

II Environment: CartPole (10%)

In this section, you will get familiar with `CartPole` environment.
Recall two functions `env.reset` and `env.step` that you will need in your code about reset the environment and take actions. Here is a brief instruction of these two functions.

When you call the `env.step`:

- The input of `env.step`:
  - action (ActType): an action the agent provides to update the environment state.
- The returns of `env.step`:
  - observation (ObsType): the next observation after taking actions.
  - reward (SupportsFloat): The reward as a result of taking the action.
  - done (bool): Whether the agent reaches the terminal state (as defined under the MDP of the task) which can be positive or negative. If true, the user needs to call the reset.
  - truncated (bool): Whether the truncation condition outside the scope of the MDP is satisfied.
  - info (dict): Contains auxiliary diagnostic information (helpful for debugging, learning, and logging). (Not used in this assignment.)
  - done (bool): **This was removed in OpenAI Gym v26.**

When you call the `env.reset`:

- The input of `env.reset`:
  - None
- The returns of `env.reset`:
  - observation (ObsType): the initial state.
  - info (dictionary): This dictionary contains auxiliary information complementing "observation". (Not used in this assignment.)

Please read through `codebase_CartPole_test.py`, `util.py`, and `get_args.py`. Additionally, refer to the CartPole instructions to understand the representation of the state and action in the Cartpole environment.

(a) You do not need to write any code for this question. Please:

- Set the parameter `method` in `get_args.py` to `test-CartPole`.

- Run `main.py` to execute the `init_states` function. This function will reset the environment and print the initial state three times.

(1) Please paste the screenshot of the three initial states.

(2) **Question**: Does the environment have a fixed initial state or a random initial state?

(3) **Question**: What is the position, velocity, angle and angular velocity of the cart and the pole in the initial state.

(4) **Question**: Based on the meaning of state in the previous question, which action—moving the cart to the left or to the right—should we take to maintain the balance of the pole?

(b) For this question, you need to complete the `test_each_action` function in the file `codebase_CartPole_test.py` between the pound sign lines denoted with "# Your Code #" in each function. You only need to write one line of code to call the `step` function to test the action. Please follow the instructions for `env.step`. Then, run the program in `main.py` to test your code.

(1) Please paste a screenshot of your function output. The screenshot must include all the printed outputs.

(2) **Question**: The function resets the environment and takes the two actions, moving to the right for one step and to the left for one step. Do these two actions restore the environment to its initial state?
Optional Question: The implementation of Cartpole is very complex and fully considers the physical theory of mechanics. Please do a brief survey and explain why moving to the right once and then back to the left once does not restore the environment to its initial state.

(c) Finish the code in the `test_moves` function. This function collects and generates the plots of the total rewards for 1000 episodes using three different policies: always to the left, always to the right, and a random policy. Please complete the code in the `test_moves` function.
Please read the Gym documentation to understand the representation of each action.
Please carefully follow the notes under the **# Your Code #** to finish this part.

(1) Please paste the generated plots of the total rewards for the three different policies.

(2) **Question:** Based on your plots, which policy do you believe is the most effective: random, always left, or always right? Could you briefly explain your reasoning?

III Implementation of DQN (70%)

In this section, you need to read and understand `main.py`, `util.py`, `get_args.py` and `codebase_DQN.py` in the project. Complete your code in `codebase_DQN.py` between the pound sign lines denoted with "# Your Code #" in each function. Run the program in `main.py` to test your code and obtain your results.

Please read the comments for each variable carefully and understand the program structure, the definition of inputs and outputs, and the notes under "# Your Code #".

(a) Implementation of DQN in the `codebase_DQN.py`. There are two classes: `Net` and `DQN`. The `Net` defines the general neural network, while the `DQN` class defines the deep Q-learning network. The function `test_dqn` defines the training process of DQN. We will implement the functions in each class and test them in this question.

- Complete the code in the `__init__` function of the `Net` class. You need to define the network, which includes one input layer, `num_hidden` hidden layers, and one output layer. You only need to use `nn.Linear` to define the layers. No other layers (Flatten, BatchNorm, etc.) are needed. In the `forward` function, pass the input sequentially through the input layer, hidden layers, and output layer. Apply the activation function `F.relu(x)` between each layer. You **do not** need to add an activation function after the output layer. Return the output of the output layer. Please follow the note under each "# Your Code #" to complete this part.
  (**Hint:** You can refer to Part I for guidance on how to construct the network.)
- Complete the code in the `__init__` function in the `DQN` class. You need to define two networks, the Q-network and the target network, deploy these two networks onto the device. The device has already been initialized and saved in `self.device`, and define the optimizer and loss function. Please follow the note under each "# Your Code #" to complete this part.
- Complete the code in the `choose_action` function in `DQN` class. This function uses epsilon-greedy strategy to select an action. You need to finish the part that selects the action with the highest Q-value. First, use the Q-network to get the action values. Then, use the `torch.max()` method to get the action with the highest Q-value. To take the action, move the action value back to the CPU using `.cpu()`
- Complete the code in the `store_transition` function in the `DQN` class. This function stores the transition of the current step in the format of: [state, action, reward, next state]. You need to store the transition in the memory using a ring buffer. A ring buffer means that if the memory is full, you need to replace the oldest transition with the new transition. The code for replacing transition has been completed, and you only need to calculate the index of the oldest transition with the `memory_counter` and `memory_capacity`.
- Complete the code in the `update_target_net` function in `DQN` class. There are two ways to update the target network. In this part, you only need to finish the "hard" updating method. "Hard" updating means assigning the Q-network parameters to the target q-network every `target_replace_iter` steps.
- Complete the code in the `learn` function in the `DQN` class. The learning process can be split into three sub-parts: sampling data, computing loss, and updating networks.
  The sampling data part has been completed. We sample a batch-size memory using `np.random.choice` to get a list of random sampling indexes, select transitions using these indexes, and save the transitions. Please read the comments to familiarize yourself with the variable definitions.
  For the second part, you need to write 3 lines to compute the Q-value and target value.
  - Get the values of the sampled actions with the Q-network and save them to `q_eval`.
  - Calculate the action values of the next states through the target network.
  - Calculate the target values and save them in `q_target`. The loss function will compute the loss between `q_eval` and `q_target`.

  In the third part, clear the gradient using `zero_grad()`, perform backpropagation, and update the network.
  The last two parts have instruction notes under the "# Your Code #". Please read and follow them carefully.

- Complete the code in the `test_dqn` function. Most of this function has been finished. You only need to finish the learning part. The learning process starts when memory is full. In this task, we keep the original batch size (i.e., 60) and let the model learn only once.
- Set `method` in `get_args.py` to `DQN`. Set `hidden_dim` to 50, `num_hidden` to 0, and learning rate `lr` to 0.01. There is no need to worry about the `input_dim` and `output_dim` as they will be automatically set according to the dimension of the state and action. Set `target_update_method` to `hard` and `reward_method` to `return`, and `return` means we directly use the environment reward to train the networks.
- Run `main.py`. The code will generate the plot of episode rewards. The max value of the episode reward is 500.

(1) Please paste the plot of the generated episode rewards.

(2) **Question**: Do the episode rewards converge to the max episode reward?

(b) In this question, we aim to optimize the performance of DQN by replacing direct learning of the return reward with a ratio reward. The terminal condition of the CartPole environment is determined by the cart's the position and the the pole's angle exceeding certain thresholds. We use the ratio of the cart's position and the pole's angle to their respective thresholds to quantify the reward. Formally, we define the ratio reward r as:

$$r = \frac{r_x - |x|}{r_x} + \frac{r_\theta - |\theta|}{r_\theta}$$

where r_x is the cart position threshold, x is the cart position. r_θ is the pole angle threshold, θ is the pole angle.

- Please implement this method in the `ratio_reward` function. The detailed instructions can be found in the note under "# Your Code #".
- Set the `reward_method` as `ratio` in `get_args.py` file and rerun `main.py`.

(1) Please paste the plot of the generated episode rewards.

(2) **Question**: At approximately which episode does the model converge to the max episode reward?
(**Hint:** Convergence can be defined as consistently maintaining a value for a prolonged periods. There might be some episodes where the reward is not maximal after convergence. These can be viewed as fluctuation.)

(3) **Question**: Can you explain why using the environment return does not lead to convergence, whereas the ratio reward does?

(c) A computer science student, Steve, conducted an experiment to understand how the frequency of updating the target network affects the performance of DQN. He set the value of `target_replace_iter` to 2 and obtained the reward results shown in Figure 1.

(1) **Question**: Compare to the result of `target_replace_iter` as 10 (your result from part III.b.1), does the episode reward from Steve's result converge nicely? If it does not, can you explain why? How does the target network updating frequency affect the performance of DQN?

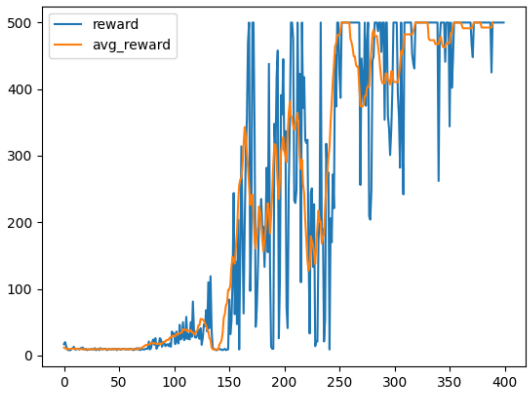
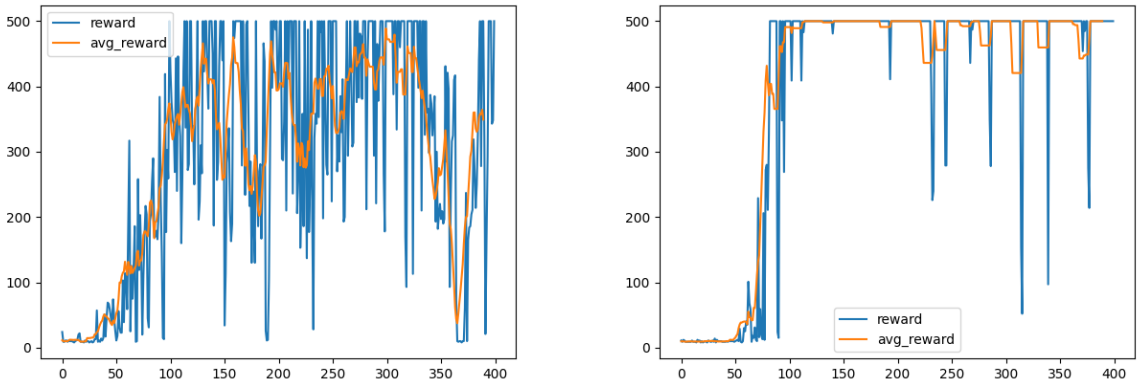


Figure 1: target_replace_iter = 2

(d) Continuing with Steve's experiment, he tried to understand how the buffer size affected the performance of DQN by setting `target_replace_iter` to 10, `batch_size` to 10, and Set the `memory_capacity` to 10 and 100. He obtained the rewards results shown in the Figure 2.



(a) 10

(b) 100

Figure 2: buffer size 10 and 100

(1) **Question**: Based on Steve's results, which `memory_capacity` makes DQN perform better: 10 or 100? Can you explain why the performance deteriorates when the buffer size is too small?

(e) Further continuing with Steve's experiment, he wanted to examine the effect of the hidden layer dimension on the performance of DQN. He kept all the settings the same as part III.b, except he set the `hidden_dim` to 256. His result is shown in Figure 3.

(1) **Question**: Please compare Steve's result with your result for part III.b.1. Does increasing

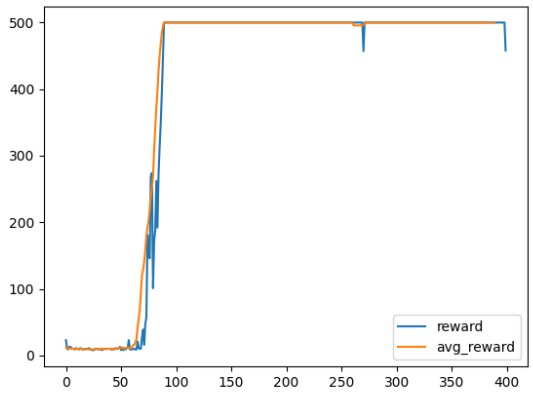


Figure 3: dim=256

the dimension of the hidden layer improve convergence? Can you explain why?

(f) Steve wanted to explore how batch size impacts the performance of DQN. He kept the same settings as in Part III.b but reduced the `batch_size` to 10. He obtained the results shown in Figure 4.

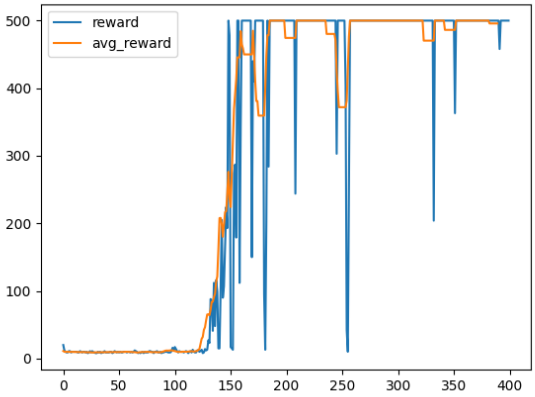


Figure 4: batch=10, train once

(1) **Question**: Is it fair to set batch size as 10 and still only learn once for each step compared with a batch size of 60? How many transitions have been passed into the network to train the model if we train 100 steps and the batch size is 60? How about if the batch size is 10? (**Hint:** In this question, we do not care about duplicates. For example, if one transition has been used twice to train the model, we consider as two transitions.)

(2) Can you help Steve to enhance the performance of DQN by adjusting the number of learning iterations. Please paste the plot of the generated episode rewards.

(**Hint:** Keep all settings same as part III.b, except set `batch_size` to 10, and then adjust the code in the function `test_dqn`.)

(3) **Question**: How does decreasing the batch size affect the performance of DQN? Can you explain why?

(g) Steve also conducted an experiment to see how the learning rate affected the performance of DQN. He changed `lr` to 0.1, and obtained the result shown in the Figure 5

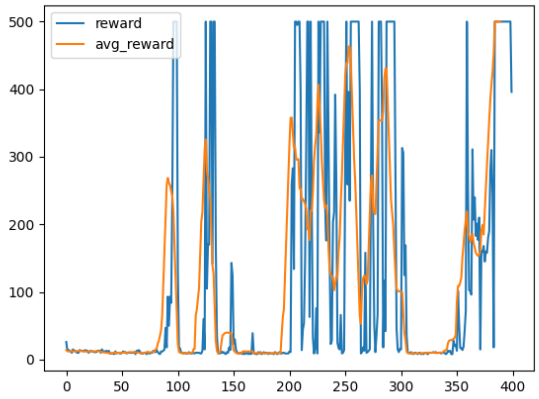


Figure 5: learning rate=0.1

(1) **Question**: Based on Steve's results, can you determine if increasing the learning rate improves convergence? Can you explain why?

(h) In this question, we try to further optimize the DQN with a soft target network updating method. The soft update method is that for each step, we conduct:

$$target\_net = target\_net * (1 - tau) + q\_net * tau$$

- Please complete the code in `update_target_net` with `target_update_method` set to `soft`. Please follow the note to finish this part.
- Reset the learning rate `lr` to 0.01. Set the `target_update_method` in the `get_args.py` to `soft`. Set the `soft_update_tau` to 0.01.

(1) Please paste the plot of the generated episode rewards.

(2) **Question**: Compared to the hard updating method, does the soft method make the model converge earlier or later?

(3) Set the `soft_update_tau` to 0.001 and rerun the code. Paste the plot of the generated episode rewards.

(4) **Question**: Does decreasing the `soft_update_tau` cause the DQN to converge earlier or later? Can you explain why?