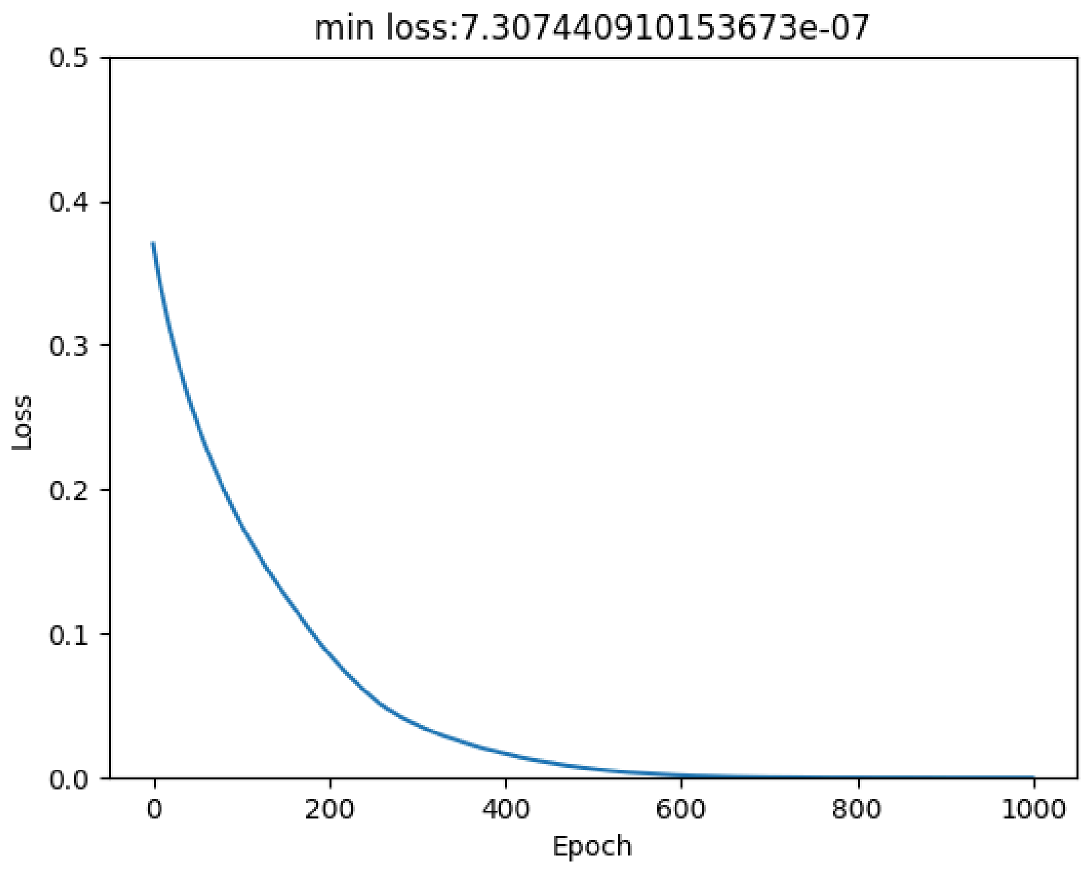
I Getting started with Training Neural Network in PyTorch (20%)

(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.

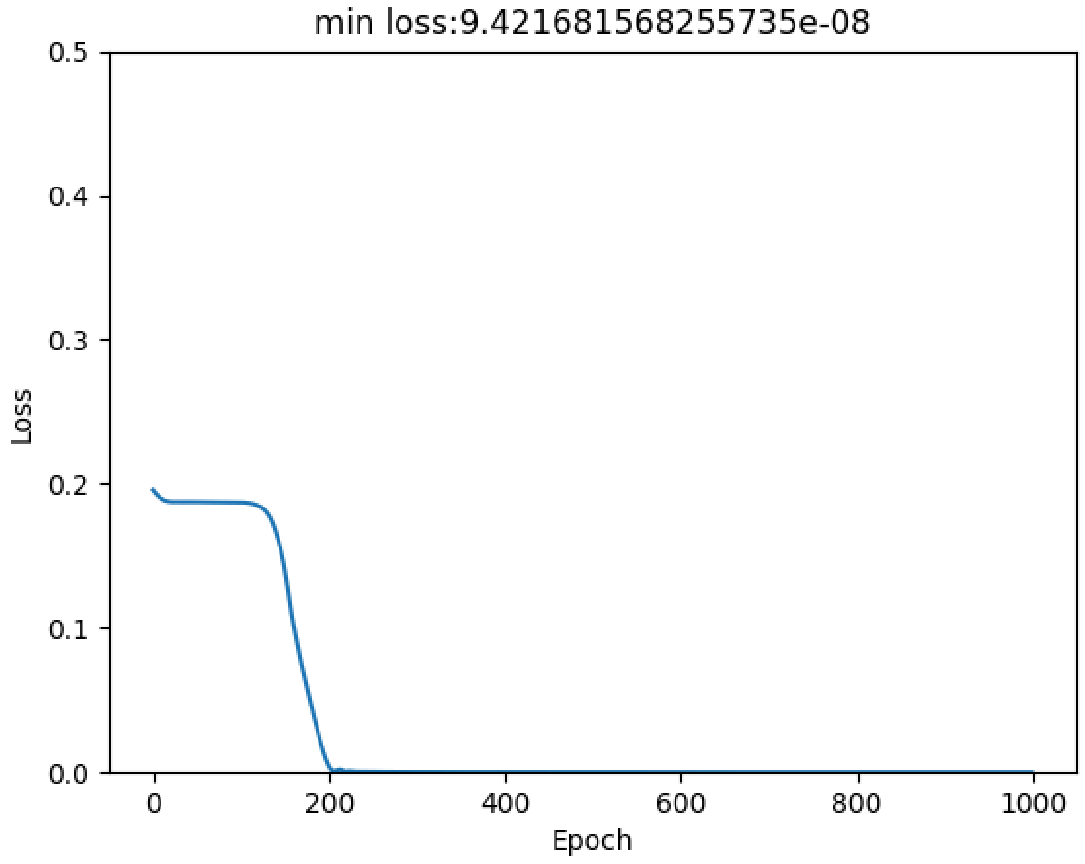


(2) **Question**: At approximately which epoch(200, 400, or 600) does the MSE loss converge? (**Hint**: A model is said to have converged when its loss function reaches a stable minimum or when the performance metrics stabilize.)

Around 400 epochs the loss stabilizes and shows slight decrease after making this point the convergence.

(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?

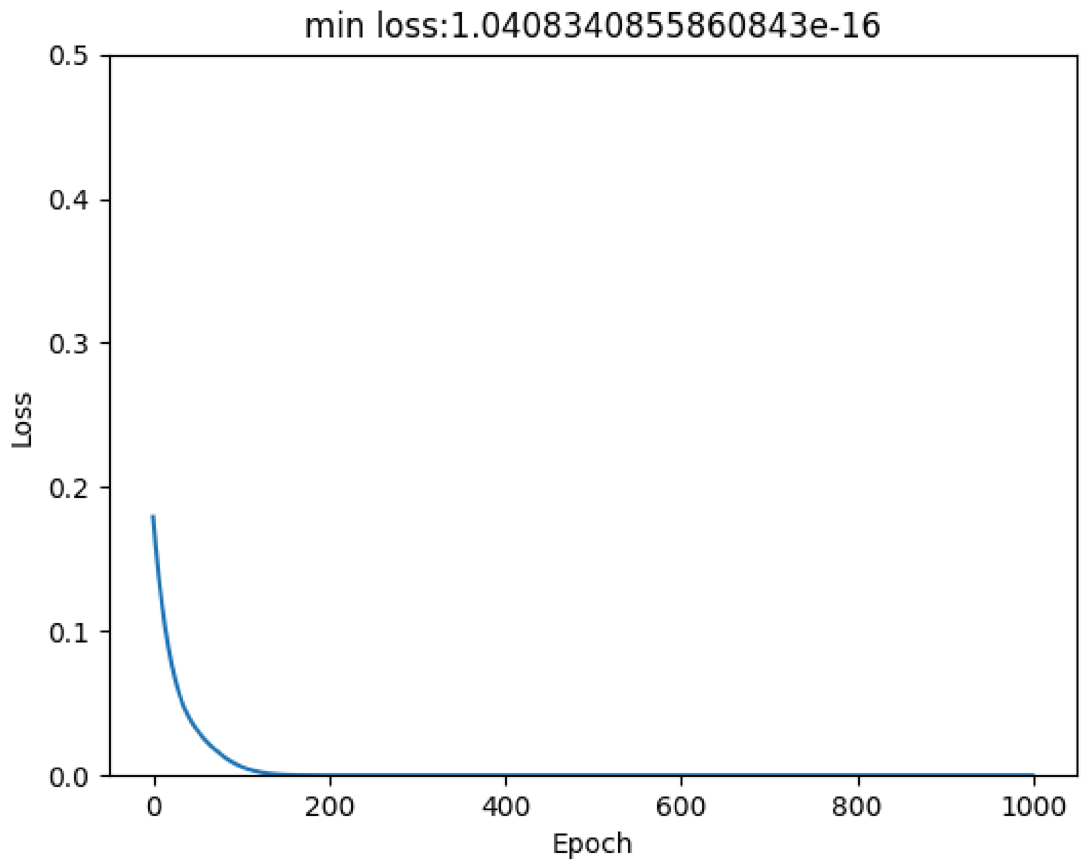
The model converges around 200 epochs, which is a lot earlier then hidden layers at 0. Increasing the number of hidden layers helps the network converge faster because of the added capacity.

(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?

Yes we do see a period where the loss does not decrease significantly. From epochs 50 to 150 there is a plateau because deeper networks can start with weights not being effective immediately, require more layers to set up gradients, and needs room to adjust because of the small learning rate.

(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.

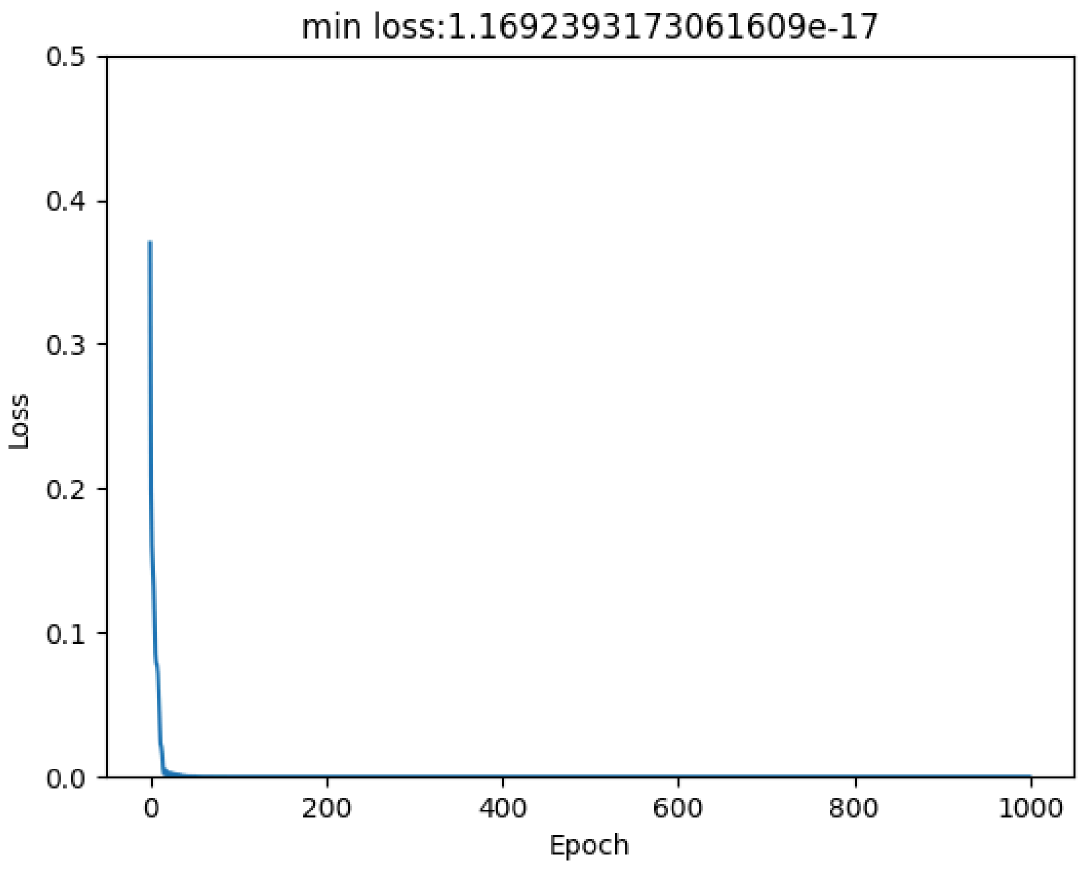
(1) **Question**: At approximately which epoch does the MSE loss converge? Does increasing the dimensions of the hidden layer help the network converge faster?



Around epochs 150-200. Increasing dimensions of the hidden layer help the network converge faster because it allows the model to fit more data and allows the model to approximate complex mappings better.

(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?



The lowest point to converge was at roughly 10-30 epochs. It is hard to tell because it has a big drop. The increasing learning rate causes the model to converge a lot faster. The learning rate converges faster because it allows the optimizer to make bigger parameter updates for each step which quickens the process of convergence. Since there are no hidden layers as well it makes it run faster and converge faster.

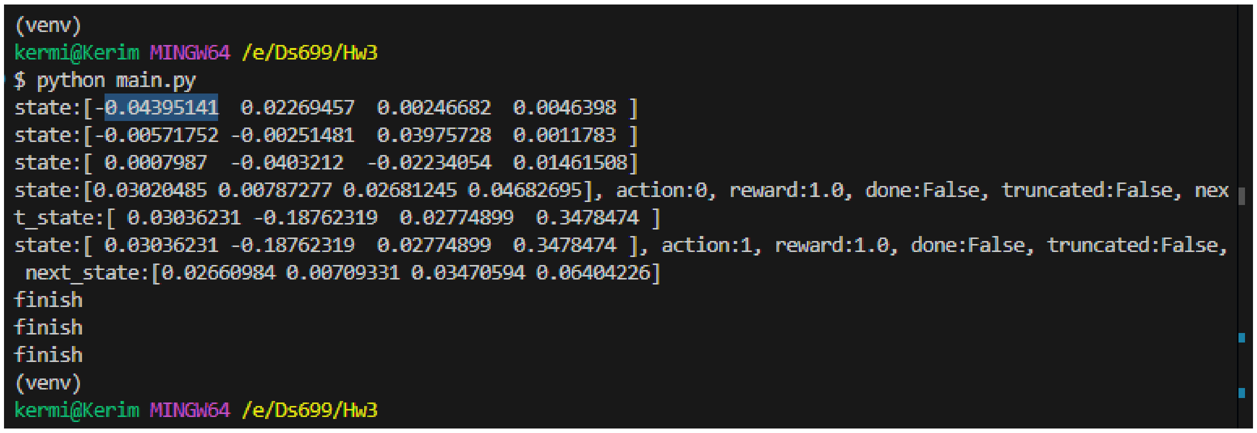
II Environment: CartPole (10%)

(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.



```
(venv)
kermi@Kerim MINGW64 /e/Ds699/Hw3
$ python main.py
state:[-0.04395141  0.02269457  0.00246682  0.0046398 ]
state:[-0.00571752 -0.00251481  0.03975728  0.0011783 ]
state:[ 0.0007987  -0.0403212  -0.02234054  0.01461508]
state:[0.03020485 0.00787277 0.02681245 0.04682695], action:0, reward:1.0, done:False, truncated:False, nex
t_state:[ 0.03036231 -0.18762319  0.02774899  0.3478474 ]
state:[ 0.03036231 -0.18762319  0.02774899  0.3478474 ], action:1, reward:1.0, done:False, truncated:False,
 next_state:[0.02660984 0.00709331 0.03470594 0.06404226]
finish
finish
finish
(venv)
kermi@Kerim MINGW64 /e/Ds699/Hw3
```

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

It is random because each call to env.reset gives a different state which means CartPole environment starts in a random configuration.

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

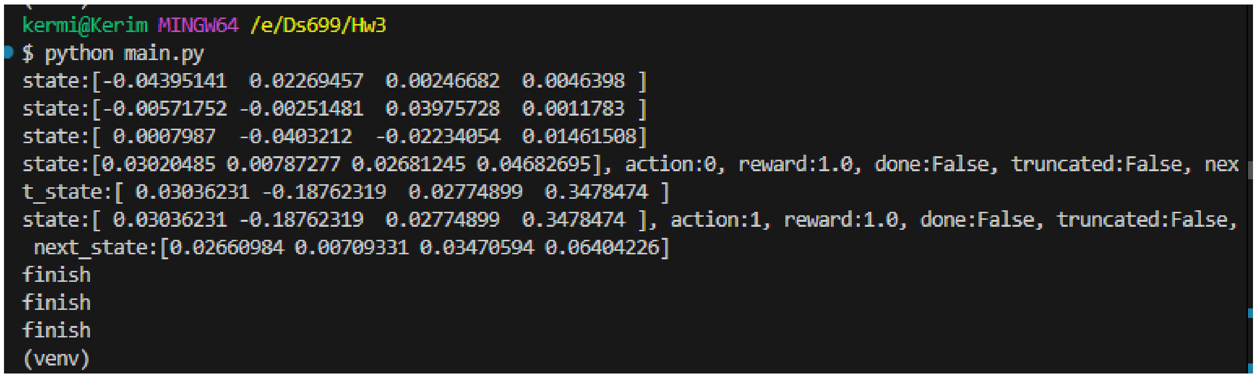
Initial State: [-0.04395141, 0.02269457, 0.00246682, 0.0046398]
Order: [position, velocity, angle, angular velocity]

(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?

This depends on the angular velocity. If angle is greater than 0 the pole is falling to the right so the action would be right but if angle is less than 0 then the pole is falling to the left so the action would be left.

(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.



```
kermi@Kerim MINGW64 /e/Ds699/Hw3
$ python main.py
state:[-0.04395141  0.02269457  0.00246682  0.0046398 ]
state:[-0.00571752 -0.00251481  0.03975728  0.0011783 ]
state:[ 0.0007987  -0.0403212  -0.02234054  0.01461508]
state:[0.03020485 0.00787277 0.02681245 0.04682695], action:0, reward:1.0, done:False, truncated:False, nex
t_state:[ 0.03036231 -0.18762319  0.02774899  0.3478474 ]
state:[ 0.03036231 -0.18762319  0.02774899  0.3478474 ], action:1, reward:1.0, done:False, truncated:False,
 next_state:[0.02660984 0.00709331 0.03470594 0.06404226]
finish
finish
finish
(venv)
```

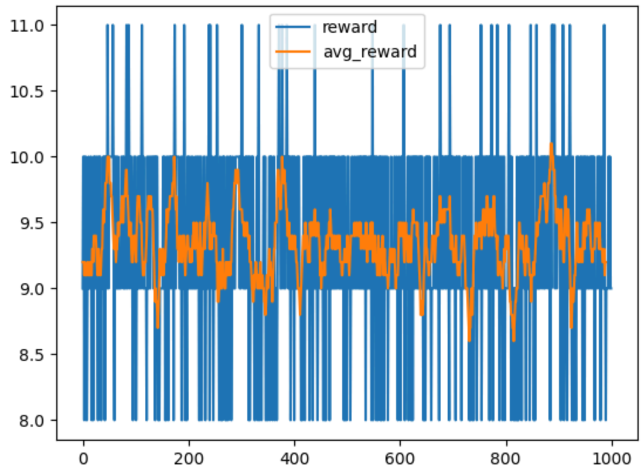
(2) **Question**: The function resets the environment and takes the two actions, moving to the right for one step and to the left for one step. Do these two actions restore the environment to its initial state?
Optional Question: The implementation of Cartpole is very complex and fully considers the 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.

No, even though we moved left then right we did not go to the exact original state.

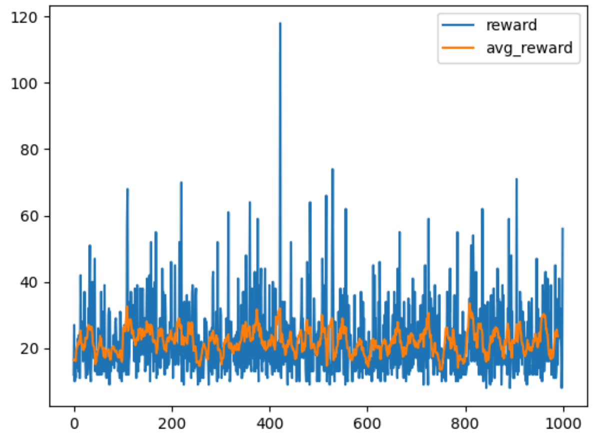
(c) Finish the code in the `test_moves` function. This function collects and generates the plots of the total rewards for 1000 episodes using three different policies: always to the left, always to the right, and a random policy. Please complete the code in the `test_moves` function.
Please read the 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.

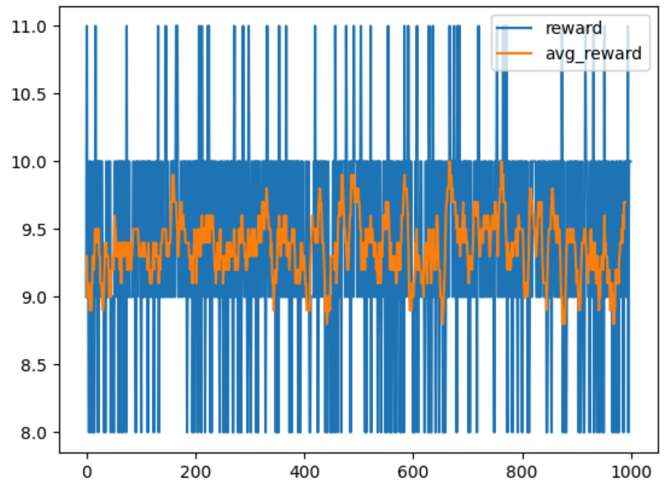
Right



Random



Left



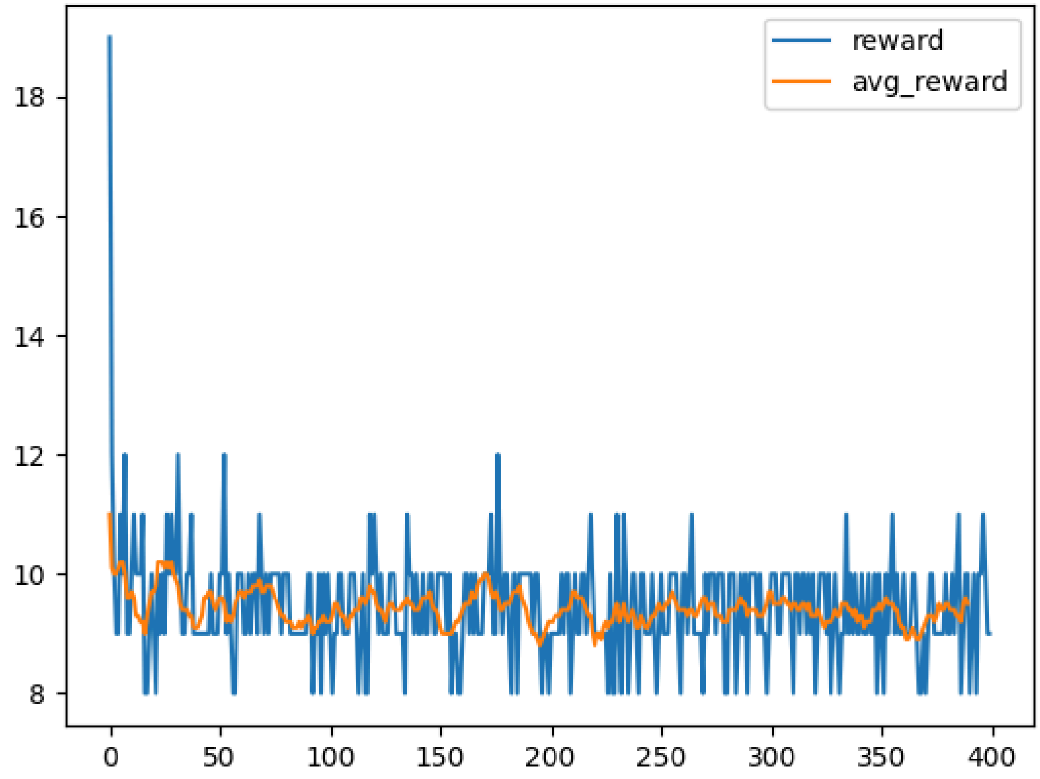
(2) **Question:** Based on your plots, which policy do you believe is the most effective: random, always left, or always right? Could you briefly explain your reasoning?

Both left and right lead to failure pretty quickly by destabilizing the pole but the random policy finds balance in actions. This makes the episodes longer on average and has some high rewards spikes. Random is the best in this scenario because it does not fail while left and right do.

III Implementation of DQN (70%)

(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.

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

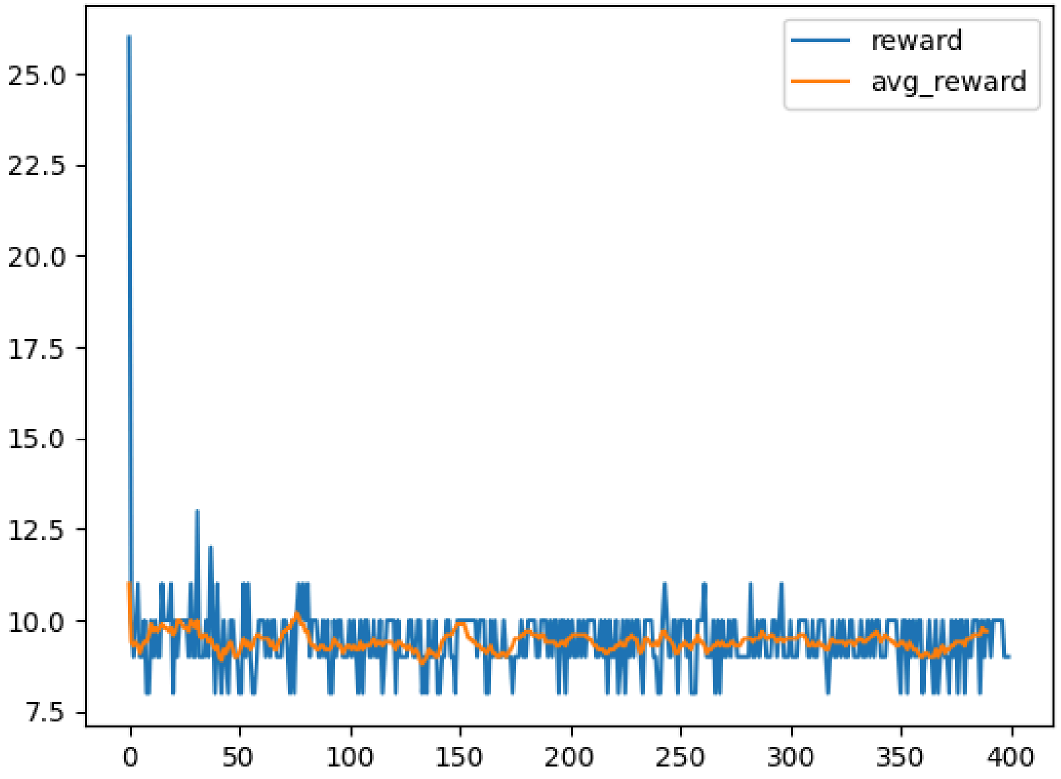


(2) **Question**: Do the episode rewards converge to the max episode reward?

No, the episodes do not converge to the maximum value of 500. The average is around 9-10 which means that DQN is not learning effectively.

(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:

(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.)

The maximum is 26 but it does not converge to the max episode reward of 500. The reward stays consistent after 50 and stays relatively low.

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

Even though the ratio did not converge as well it has better results because it is more consistent throughout the graph. Ratio is better because it shows how close the cart and pole are to failure, helps the model learn useful gradients early, and rewards actions that delay failure. Return does not give any signals until the episode ends and displays lower learning.

(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?

Yes, Steve’s episode converges to the max reward of 500 but not smoothly. The plot shows rapid growth after about 100 episodes but with big fluctuations after meaning there is some sort of instability in learning. This is probably because the target network frequently at every 2 steps changes the stabilizing role of the target network. The agent is learning but because Q-value changes too quickly it causes the fluctuations in training. The higher value of 10 allows the network to be more stable even though it did not converge. Overall Steve’s DQN converges better than my results; it has a lot more instability in learning.

(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.

(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?

The buffer size of 100 performs better than 10. This is because with 100 DQN is able to stabilize and converge smoothly because of the training data through multiple episodes improving the learning. A small buffer of 10 DQN trains on recent transitions which is too correlated causing the model to be overfitted resulting in unstable learning.

(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

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

Increasing dimension of hidden layer improves DQN convergence. Steve’s model has hidden dimension of 256 which converged the maximum reward around 100 epochs while my results had hidden dimension 50 and did not improve. A larger hidden layer helps the network to learn better in complex scenarios, model more Q-value functions, and improve generalization. Because my model has smaller hidden dimensions my version failed to capture important features of the environment causing bad results.

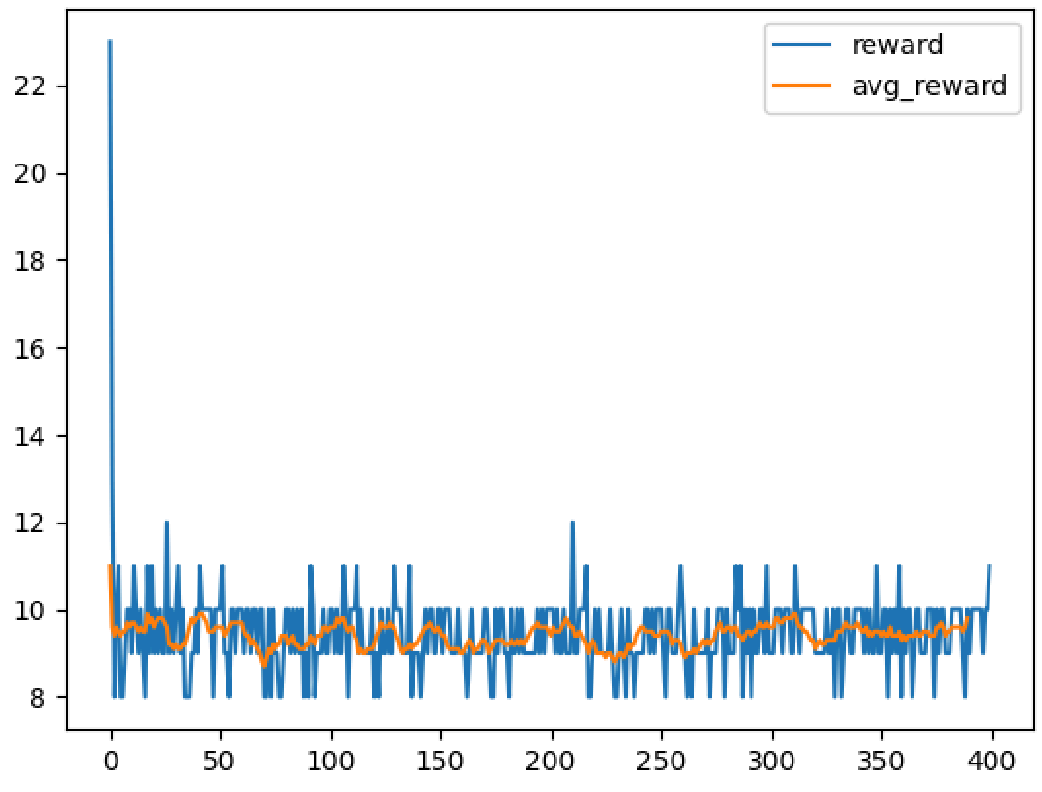
(f) Steve wanted to explore how batch size impacts the performance of DQN. He kept the same settings as in Part III.b but reduced the `batch_size` to 10. He obtained the results shown in Figure 4

(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.)

No it is not fair to compare a batch size of 10 to 60 when training once per step. With batch size of 60 each update will have 60 transitions over 100 steps meaning there would be 6000 total transitions. With a batch size of 10 it would only be 1000 total transitions. With a smaller batch size the experiment will have slower convergence, more variance, and signs of underfitting early in training making this an unfair comparison.

(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?

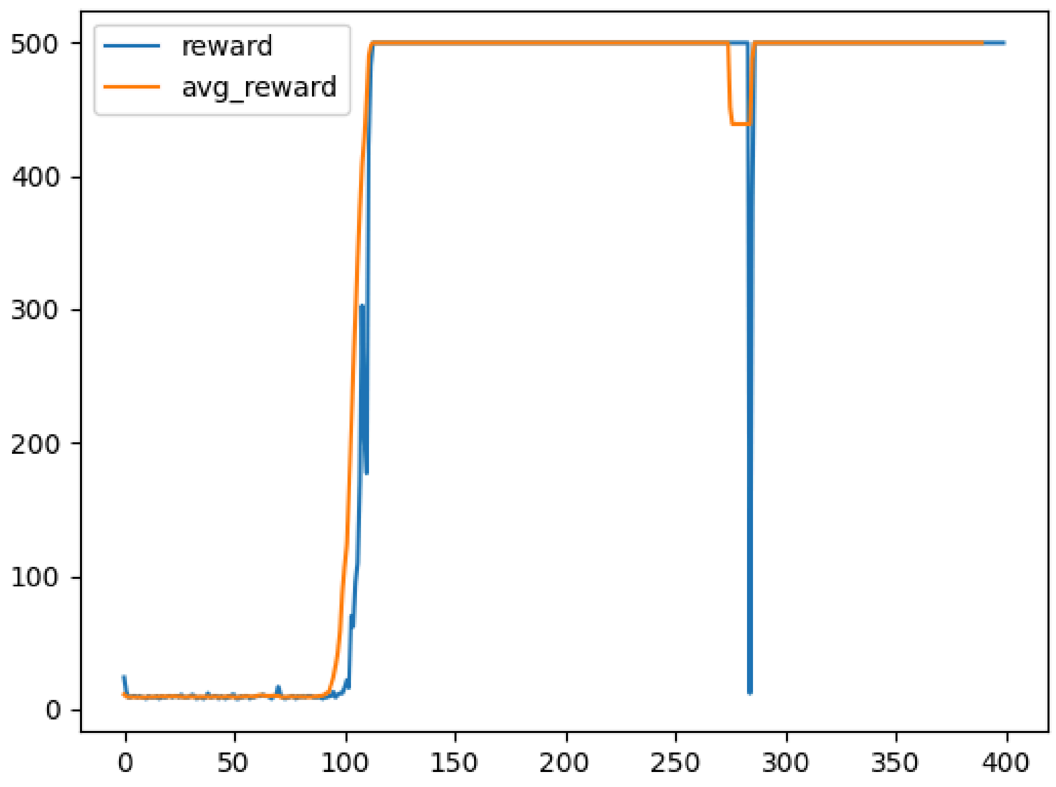
Decreasing the batch size did not improve performance of DQN. Even with a fair number of training updates the model does not have enough reward signal strength to learn effectively here. We can see that Steve's performance is better because he has a larger hidden dimension and most likely a better weight initialization setup.

(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

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

Based on Steve's result, increasing the learning rate to 0.1 did not lead to a better convergence because even though he has a higher learning rate can increase early learning it makes the training process unstable and harder to control. We can see in his graph that there are a lot of overshooting in Q-values, inconsistent convergence patterns and high rewards led by deep crashes. A smaller learning rate of 0.01 is more stable for optimizing and allows the network to enhance Q-values over time and more consistently.

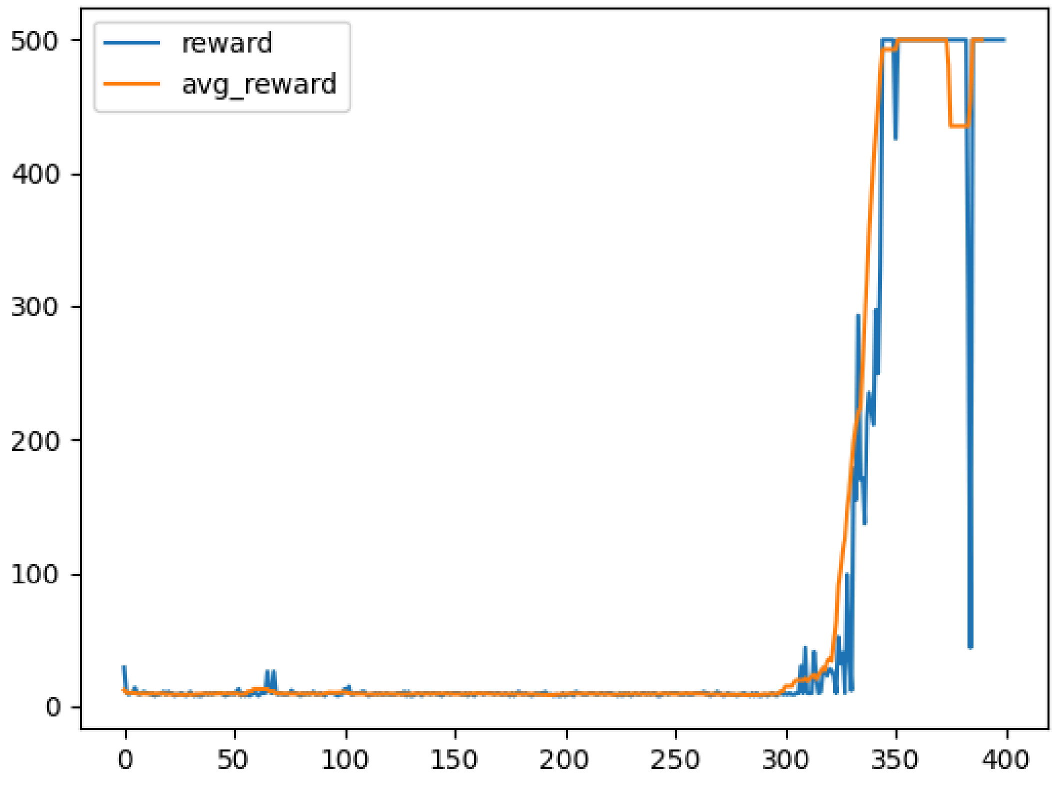
(1) Please paste the plot of the generated episode rewards.



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

The soft method converges earlier and is a lot smoother compared to the hard update method. This is because the hard update method has a target replace iteration at 10 which makes the network sync in intervals of 10. This causes unstable learning but with soft updates there are small syncs of Q-net into the target net which makes the results more stable for target values, smoother training, and earlier convergence (around 100).

(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?

Decreasing soft update tau to 0.001 makes the model converge around 320-340 epochs. This version however has more instability because a smaller tau moves the weights slower from Q-network weights into the target network. Even though this can help spikes to recent changes it causes a delay stabilization of the target network making the process slower to improvements. Overall we reach the top rewards quickly as well but the smaller soft update tau made the learning more unstable and caused a big dip.