RL HW 3

John Vitz

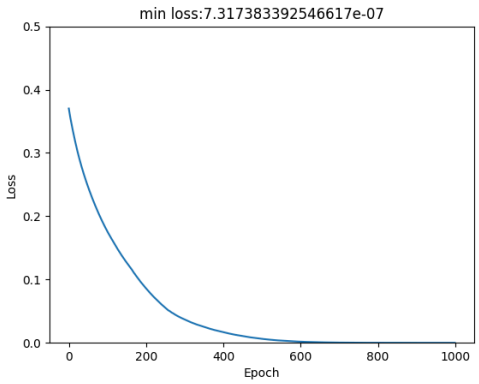
I Getting started with Training Neural Network in PyTorch (20%)

(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.

python main.py -method test-PyTorch -hidden_dim 10 -num_hidden 0 -lr 0.001



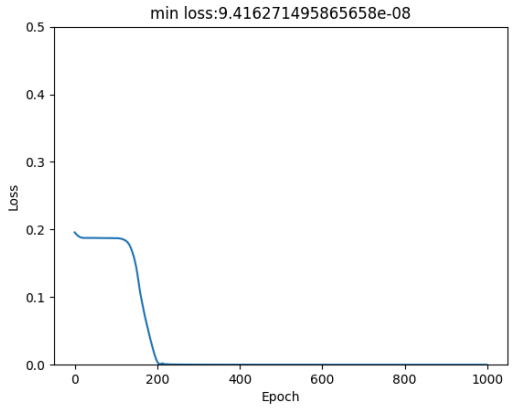
(2) Question: At approximately which epoch(200, 400, or 600) does the MSE loss converge? (Hint: A model is said to have converged when its loss function reaches a stable minimum or when the performance metrics stabilize.)

The model converges at 600 epochs as the loss converges to 0.0 on the graph.

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

(1) Please paste the plot of the loss.

python main.py -method test-PyTorch -hidden_dim 10 -num_hidden 9 -lr 0.001



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

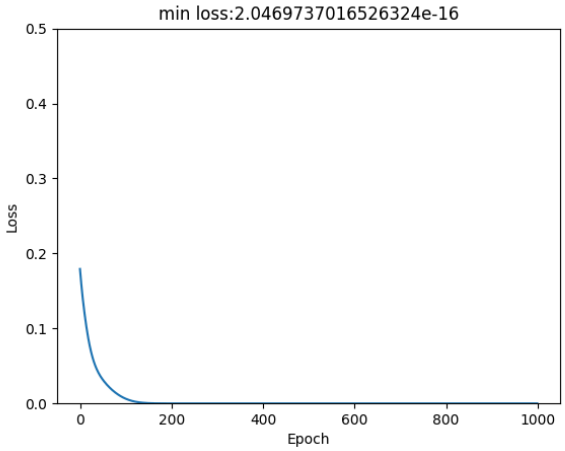
The model converges at approximately 200 epochs to 0.0 loss. Increasing the number of hidden layers does help the network converge faster, as the prior network took ~600 epochs to converge. Increasing the number of hidden layers allows for more behavior to be captured by the model.

(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?

The network needs more time to find a suitable position on the loss landscape where all of the randomly initialized hidden layers can begin to work in concert towards actual optimization.

(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.

python main.py -method test-PyTorch -hidden_dim 100 -num_hidden 0 -lr 0.001

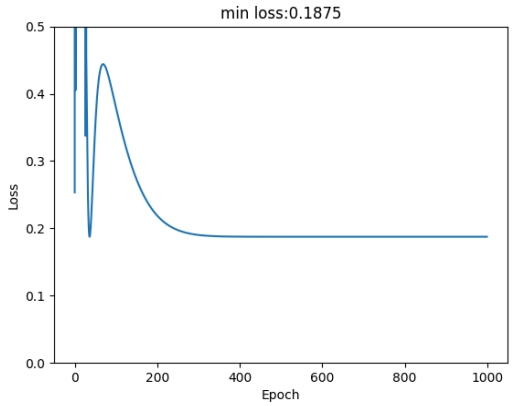


(1) Question: At approximately which epoch does the MSE loss converge? Does increasing the dimensions of the hidden layer help the network converge faster?

MSE converges at around 120 epochs. Increasing the dimensions of the hidden layer helps the network to 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.

python main.py -method test-PyTorch -hidden_dim 100 -num_hidden 10 -lr 0.1



2.

(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 of this assignment.

The model converges to the lowest MSE at around 250 epochs. Increasing the learning rate does not cause the network to converge faster in this case. The learning rate set at 0.1 sets the network updates at a rate that is too high for the network to find an optimal point in the loss landscape before convergence.

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

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.

```
state:[ 0.01608499  0.02160615  0.00491348 -0.00964504]
state:[-0.0476647   0.04163745 -0.03514528 -0.00219863]
state:[-0.0039758   0.00794013 -0.04580192  0.03898295]
```

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

It is a random initial state, if it was fixed, each state would have the same values. I also ran the command multiple times, and the states vary across initialization, so there is no random seed being utilized / no deterministic function behind the generation of the initial state.

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

Reference Image: https://www.gymlibrary.dev/environments/classic_control/cart_pole/

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad (-24°)	~ 0.418 rad (24°)
3	Pole Angular Velocity	-Inf	Inf

The first dimension in the vector is Cart Position. The second dimension in the vector is Cart Velocity. The third dimension in the vector is Pole Angle. The fourth and final dimension in the vector is Pole 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?

The action that we should take to maintain the balance of the pole depends on both the Pole Angle and Pole Angular Velocity.

Pole Angle:

- If the pole angle is negative, this means that it needs to be pushed to the left to catch the pole to prevent an episode termination.

- If the pole angle is positive, this means that it needs to be pushed to the right to catch the pole to prevent an episode termination.

Pole Angle Velocity:

- A pole that is moving slowly in a direction only needs a small correction, while a pole moving fast in a direction may need a larger and/or quicker correction.

So, the best action that can be taken in this case depends on both the current angle and current angular velocity of the pole at a given time step.

(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.

```
state:[ 0.00224893 -0.01911942 -0.0039281   0.03329661], action:0, reward:1.0, done:False, truncated:False, next_state:[ 0.00186654 -0.21418482 -0.00326217  0.32473758]
state:[ 0.00186654 -0.21418482 -0.00326217  0.32473758], action:1, reward:1.0, done:False, truncated:False, next_state:[-0.00241716 -0.01901658  0.00323259  0.0310277 ]
```

(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?

No, these two actions do not restore the environment to its initial state. After the second action, the state is not the same as the initial state; the cart position, velocity, and pole angle are all different. So, even though the actions are symmetrical in the sense that we pushed the cart to the left and then pushed the cart to the right, they do not result in symmetrical end states. This is due to the fact that the environment is a physics simulation; the system has momentum and inertia, and reversing the previous action that the car takes does not translate to canceling out the effects of momentum and inertia on the pole.

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.

The CartPole environment is a continuous dynamic system that is directly influenced by the physical environment factors Inertia and Momentum. When the cart moves in a direction, it gains speed and the pole begins to tilt and swing, which builds up momentum in the cart and the pole.

When you move the cart back to the left, it doesn't perfectly undo what has already happened because of inertia and its built up angular momentum. The cart and the pole want to keep moving in the same direction from the step prior; the pole's angular momentum will keep it swinging in the same direction unless it is corrected by the cart push, and even then it is more likely that the cart push will only temporarily slow down / end up reversing the direction of the swing eventually anyway. So in conclusion, the system ends up in a different state even if you reverse the prior action because of inertia and momentum.

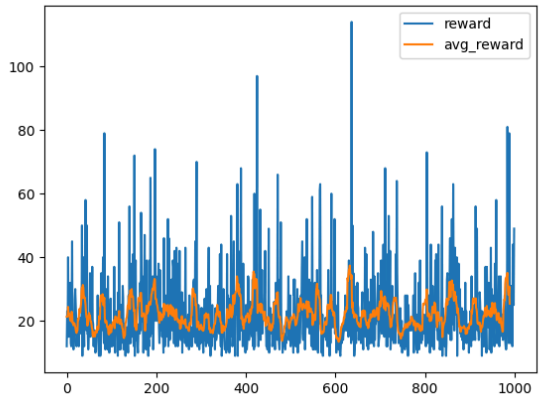
(c) Finish the code in the test_moves function. This function collects and generates the plots of the total rewards for 1000 episodes using three different policies: always to the left, always to the right, and a random policy. Please complete the code in the test_moves function.

Please read the Gym documentation to understand the representation of each action.

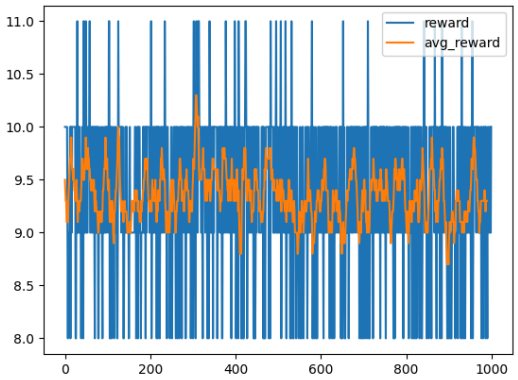
Please carefully follow the notes under the # Your Code # to finish this part.

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

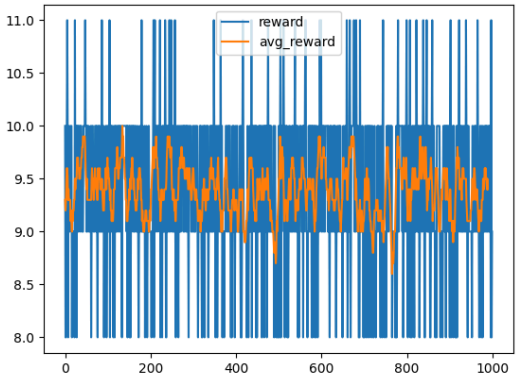
Random:



Right:



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?

Random is an effective policy compared to always right and always left. The reason for this is very simple: If the cart can only move in one direction, it can never attempt to correct a pole that is moving in the opposite direction.

This means that these always left or right policies fail:

- Firstly, on initialization when a pole is initialized with a velocity / momentum that propels the pole in the opposite direction of the policy.

- Secondly, on correction when a car is pushed and it causes the pole's angular velocity / momentum to change direction enough to propel the pole in the opposite direction of the policy.

- Lastly, the environment settings are initialized in a way that causes episodes to terminate if the Cart Position reaches the edge of the display; this means that the cart under an ‘always left’ or ‘always right’ policy will inevitably broach the bounds that it is allowed to move from the origin and cause an episode cancelation.

So, the ‘always’ policies do not work because they do not allow the cart the flexibility that it requires in order to maintain the conditions to receive a reward for long enough. In contrast, the random policy does allow for corrections, even though it doesn’t explicitly facilitate them.

Because the random policy samples from a 50/50 ‘coin flip’ esque probability distribution, it occasionally performs sequences of actions that happen to correct the pole's balance which is the reason why the random policy’s average reward plot shows a reward that is ~2 times higher than the ‘always’ policies.

Basically, the randomness gives it a nonzero chance to prevent all of the aforementioned problems with the always policies from resulting in an episode termination, which manifests in the form of the higher average reward for the policy shown in the plot, as well as the large amount of high end reward variability shown on the random policy reward plot, which is due to the random policy successively correcting the pole enough to prevent the termination state over the course of a few episodes.

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 defined 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 transitions 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.

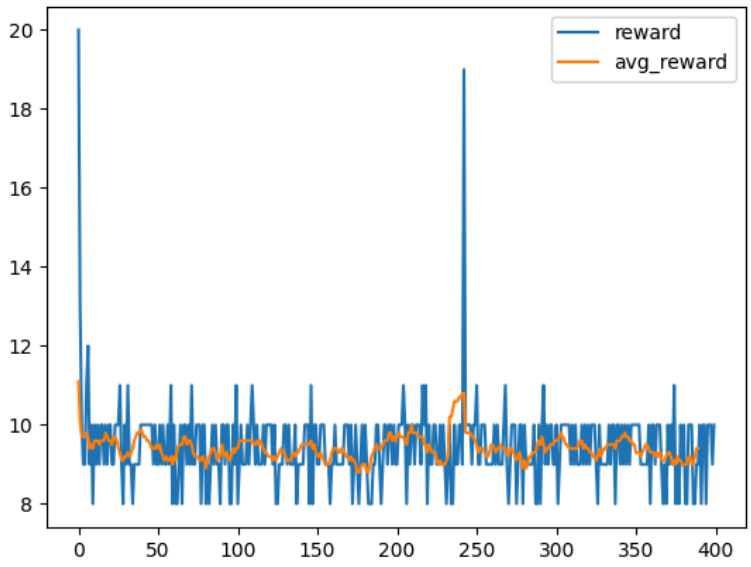
- Calculate the target values and save them in q_target. The loss function will compute the loss between q_eval and q_target.

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.

python main.py -method DQN -batch_size 60 -hidden_dim 50 -num_hidden 0 -lr 0.01 -target_update_method hard -reward_method return



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

No, the episode rewards do not converge to the max episode reward. The max reward is much higher than the reward pictured here; the maximum reward is 500 in the CartPole environment that we are using.

(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 position and 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:
$r = ((r_x - |x|) / r_x) + ((r\theta - |\theta)| / r\theta)$

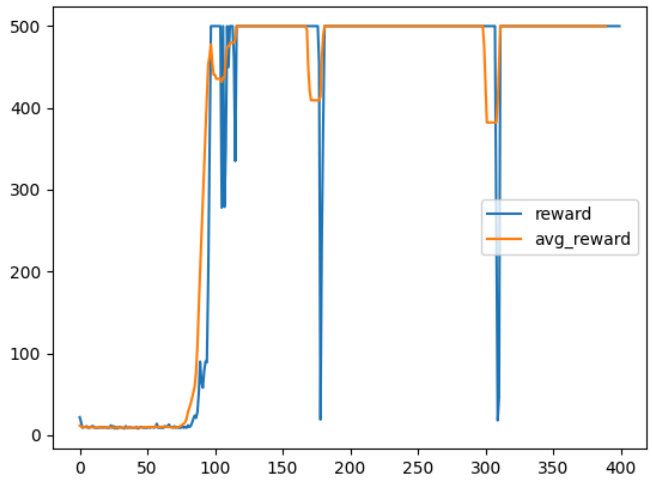
where rx is the cart position threshold, x is the cart position. rθ is the pole angle threshold, θ is the pole angle.

- 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 a ratio in get_args.py file and rerun main.py.

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

python main.py -method DQN -batch_size 60 -hidden_dim 50 -num_hidden 0 -lr 0.01 -target_update_method hard -reward_method ratio



(2) Question: At approximately which episode does the model converge to the max episode reward?

It converges to the max episode reward at episode ~120, where after it has some very large reward fluctuations but mostly maintains the maximum reward until the final episode.

(Hint: Convergence can be defined as consistently maintaining a value for 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?

Environment return does not lead to convergence because the model is not informed by the reward as to what kind of actions are optimal given the state input. This means that the model struggles to learn any sort of optimal behavior, as it simply doesn't have a way of "pointing" its efforts towards specific behaviors. Environment return reward is sparse; it either occurs or doesn't. This sparsity means that the model is unable to differentiate between good behavior and behavior that is suboptimal in the case of a positive reward; the model is also unable to differentiate between slightly bad and extremely bad behavior in the case of a negative reward. Basically, the sparse reward does not allow the agent to learn which moves keep the pole upright better than other moves, which results in a total failure to learn anything useful at all.

On the other hand, the ratio reward directly shapes the way that the model is incentivised. Through the ratio reward, both the behaviors that facilitate a higher reward are encouraged. The first term of the function that involves the position of the cart incentivises the cart to try to stay within the edge bounds of the environment, which helps to prevent premature terminations of each episode. The second term that involves the angle of the pole helps to incentivise a minimal pole angle, which penalizes cart actions that would exacerbate the angle while incentivising minimization of the angle. Ratio reward provides a dense reward; really good behavior is heavily incentivized, really bad behavior is heavily disincentivized, and the nuance in between the two extremes is captured. This is why the ratio reward leads to convergence where the environment return does not.

(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.

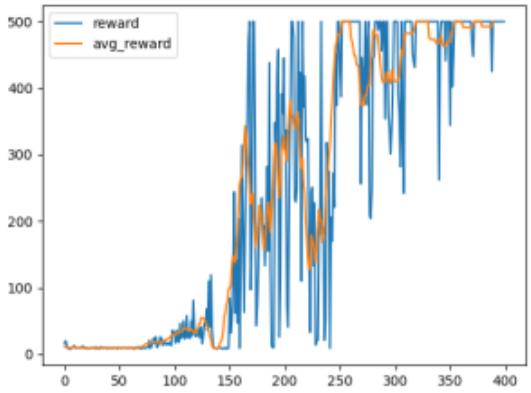


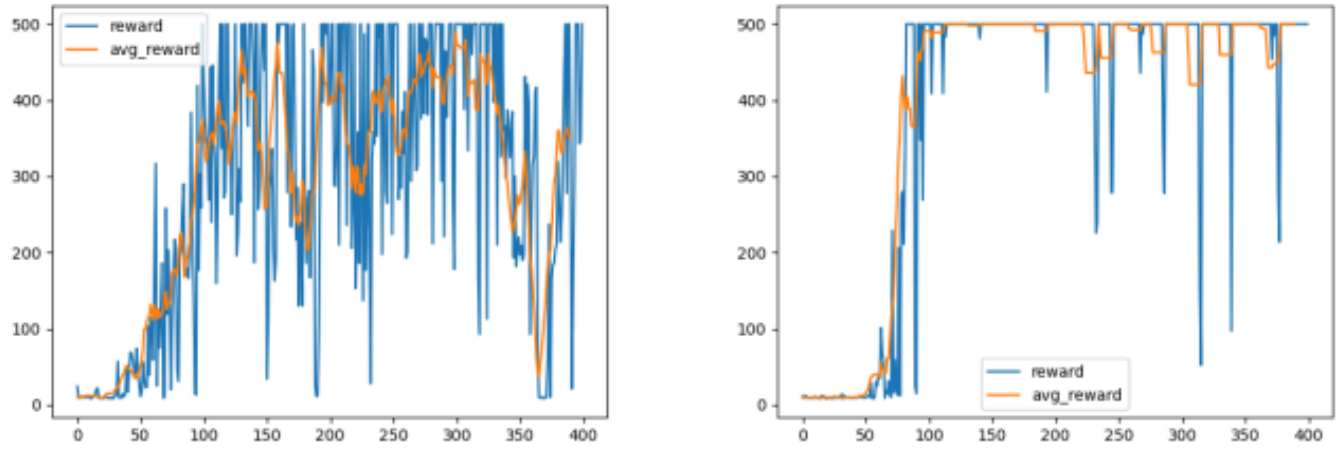
Figure 1: target_replace_iter = 2

(1) Question: Compared 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?

Steve's struggles to converge at `target_replace_iter` = 2 versus my example at 10. The reason why it struggles to converge is likely due to the fact that the choosing to update model behavior from an update frequency of 2 steps results in a series of reward updates that is much higher in terms of variance than my example at `target_replace_iter` = 10. My example updates with more bias, as the larger update frequency step number results in updates that are typically much less extreme than the smaller `target_replace_iter` = 2 that Steve used while training his network.

The effect of update frequency on DQN performance is obvious here: if you update too frequently, your updates will tend to have a high enough variance that it can prevent you from optimally (if at all) reaching convergence. In contrast with this, if you update too slowly, your updates can end up having a bias that is too high; if this occurs, the network can slow down in learning or stop altogether due to the inability to learn more nuanced patterns that may lead to further general optimizations. So, there needs to be a balance between bias and variance in the form of the `target_replace_iter` value being set to a value that allows for stable updates as well as nuanced 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 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?

Based on Steve's results, the DQN with a `memory_capacity` of 100 performs better than the DQN with a `memory_capacity` of 10. DQN_{100} basically converges, while DQN_{10} fails to converge. The DQN with a `memory_capacity` or experience replay buffer of 100 performs better because the model has an adequate number of diverse past states to sample from. The higher memory capacity facilitates the learning of long term patterns, and also allows for sampling of states that are less likely to be directly/highly correlated to each other, which means that gradient updates are more likely to be derived from a diverse range of states that are independently and identically distributed.

In contrast, a `memory_capacity` or experience replay buffer of 10 performs poorly due to its restricted number of past states to sample from. There is no long term pattern preservation over the course of the states, and all of the states in the window are heavily related to each other, which means that updates are both heavily biased towards immediately preceding states and extremely high variance due to the small size of the memory window compounded with the fact that each of the states is dependent on one another.

(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?

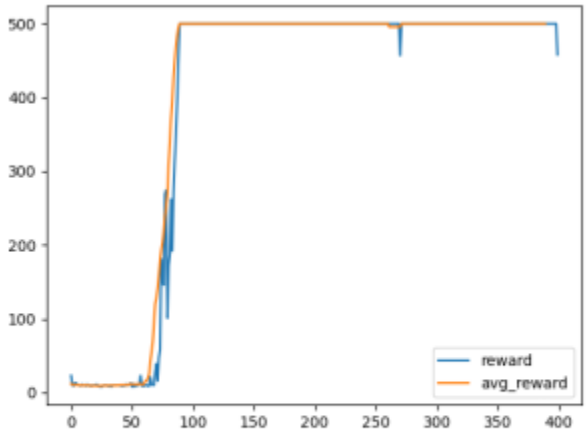


Figure 3: dim=256

Increasing the number of dimensions in the `hidden_layer` layer does improve convergence. This occurs because the model has more space with which it can capture incoming information, which allows it to approximate more complex behaviors over the course of each episode.

This increase in the `hidden_layer` dimension allows the model to capture more optimal behavioral patterns that prevent it from having the same magnitude / frequency of fluctuations over the course of reaching convergence compared to my model at `hidden_dim` = 50, as well as in the episodes after convergence.

(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.

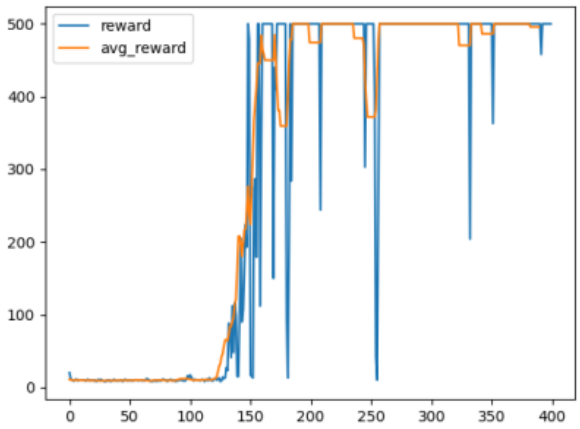


Figure 4: batch=10, train once

(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.)

No, it is not fair to set the batch size as 10 and still only learn once for each step compared with a batch size of 60. Reducing the batch size and holding other factors constant reduces the number of transitions that the network trains on over the course of 100 steps.

The total number of transitions that the model learns from is equal to the total number of steps times the transitions per batch times the number of times that the batch is utilized to inform the update for a given step.

If we train 100 steps:

@batch_size= 60: 1 learn / step * 100 steps * 60 transitions per batch = 6000 transitions passed
@batch_size= 10: 1 learn / step * 100 steps * 10 transitions per batch = 1000 transitions passed

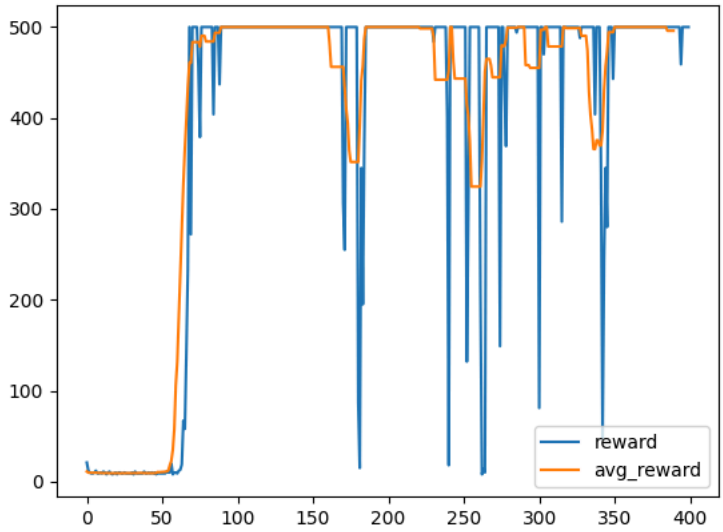
If we use a `batch_size` = 10 and still only update a single step, the model trains on 1/6 the amount of transition samples of the original model that trained on `batch_size` = 60. That means that the model processes significantly less transitions when it is learning, which means that the

10 transitions per batch model has a smaller overall chance to learn than the 60 transitions per batch model.

(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 the same as part III.b, except set batch_size to 10, and then adjust the code in the function test_dqn.)

In order to be fair, we have to apply the learning from the batch_size of 10 six times, which will make the number of transitions each model learns from the same. That being said, doing so still sacrifices the diversity of transitions you get with the larger batch size.

python main.py -method DQN -batch_size 10 -hidden_dim 50 -num_hidden 0 -lr 0.01 -target_update_method hard -reward_method ratio



My change to `test_dqn` function:

```
############################
# Your Code #
# start to learn when memory is full. learning one time for each step for the original batch size
# In the question about the batch size, you need to think about when changing batch size from 60
# many times of learning can make the performance comparison fair.

if dqn.memory_counter >= dqn.memory_capacity:
    #dqn.learn(target_update_method=args.target_update_method)
    # for batch_size=10, do 6 updates/step so we see 60 samples per step

    # Part 3 (f) question (2)
    if args.batch_size == 10:
        for j in range(6):
            dqn.learn(target_update_method=args.target_update_method)
    else:
        dqn.learn(target_update_method=args.target_update_method)

############################
```

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

Decreasing the batch size affects the performance of the DQN by making the updates higher variance. Applying a loop to apply the same number of learning steps for the `batch_size` = 10 as it does by default to the `batch_size` == 60 step results in the utilization of the same number of state transitions being used per update, however it does not compensate for the fact that smaller batch sizes have a higher variance.

The fact that this specially handled `batch_size` = 10 case in the first 100 training steps is directly comparable to the `batch_size` = 60 case in the first ~100 steps. However, as you can observe from the higher amount of fluctuation after convergence, the DQN with `batch_size` = 10 is more volatile than the DQN with `batch_size` = 60. The lower `batch_size` at 10 versus 60 results in a less diverse pool of states being used to determine how to update the DQN; this manifests itself in the plot where there is obviously more fluctuation from the convergence point of the model.

(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

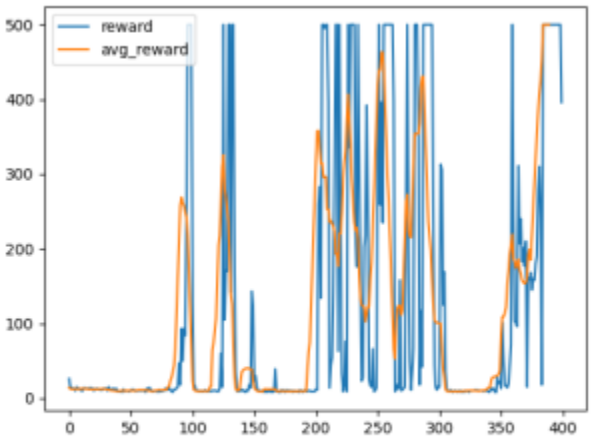


Figure 5: learning rate=0.1

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

Increasing the learning rate does not improve convergence, as the plot shows that the model goes through multiple spikes and troughs while attempting to learn with the `lr` = 0.1. The problem here is that the high learning rate results in network updates during training that are so large that the immediate updates fully dominate the update direction of the gradient during each training step. This high learning rate results in very unstable, large network updates that completely bias the network towards the immediate history window and destroy its ability to retain any learned information over the long term (evidenced by the multiple spikes and troughs). Even when the network does learn something useful by chance, the high learning rate means that it is more likely than not to be completely forgotten over the course of the next few updates, since the size of the gradient updates is so overwhelming that the model cannot retain useful information. In summary, the high learning rate is too volatile to facilitate optimal, stable learning.

(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.

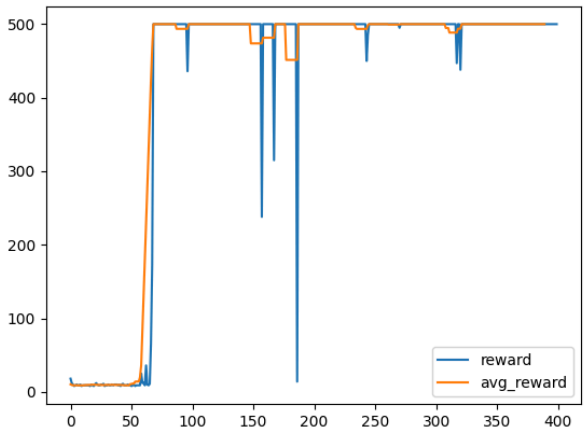
My change to `update_target_net`:

```
elif target_update_method == 'soft':
    ##########################
    # Your Code #
    # update the target network by using the soft update method. The formula is:
    # target_net = target_net * (1 - tau) + eval_net * tau
    # tau is the soft_update_tau
    # Tou need to use the state_dict() method to get the parameters of the network
    # The parameters of the network are stored in a dictionary. So you need to update the parameters for each key in the dictionary
    # You also need to use the load_state_dict() method to load the parameters to the target network

    # Soft update: target = target * (1 - tau) + q_net * tau
    q_params = self.q_net.state_dict()
    target_params = self.target_net.state_dict()

    for key in q_params:
        target_params[key] = ((1 - self.soft_update_tau) * target_params[key]) + (self.soft_update_tau * q_params[key])

    self.target_net.load_state_dict(target_params)
```

There is no detail about having to reset anything else, so I used settings from the prior question as baseline before editing my call.

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

python main.py -method DQN -batch_size 10 -hidden_dim 50 -num_hidden 0 -lr 0.01 -target_update_method soft -reward_method ratio -soft_update_tau 0.01

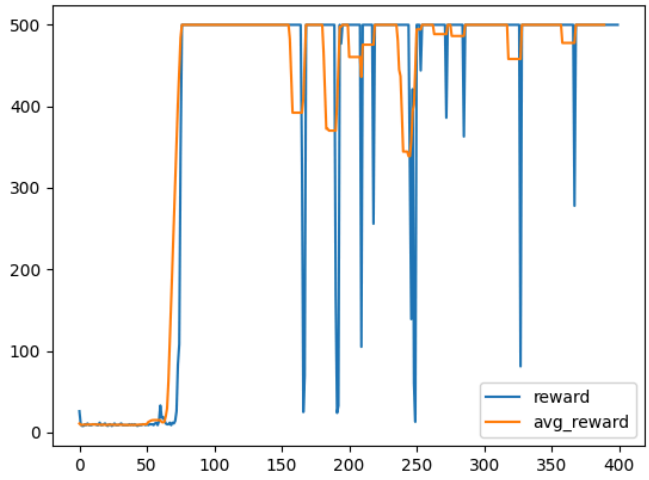


(2) Question: Compared to the hard updating method, does the soft method make the model converge earlier or later?

Compared to the hard updating method, which converged at ~120 training steps, the soft method makes the model converge earlier with respect to the number of episodes. This is because the `soft_update_tau` value of 0.01 makes the target network blend 1% of the online Q-network's weights into the target network. By making tiny updates every step and a larger update every ten episodes, the target network is kept stable while also remaining up to date, which allows for the network to learn faster and consequently causes the agent to converge to the optimal policy faster.

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

python main.py -method DQN -batch_size 10 -hidden_dim 50 -num_hidden 0 -lr 0.01 -target_update_method soft -reward_method ratio -soft_update_tau 0.001



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

Decreasing the `soft_update_tau` to 0.001 causes the DQN to converge later; the model converges at episode ~75 versus the prior `soft_update_tau` = 0.01 convergence at ~65. This occurs because a smaller `soft_update_tau` corresponds to a smaller percentage of the online Q-network being utilized to perform network updates. This basically is a trade of variance for more bias towards the target network's existing parameters. As a result, the TD targets remain stale and the learning signals update more slowly, delaying the agent's convergence.