1. Introduction

This lab aims at practicing DQN and some enhancements of DQN such as double DQN, prioritized experience replay and multi-step return. There are two game environments used in this lab, cartpole and pong games. For cartpole, it's much easier to train than pong game, the model converges quickly without any enhancements. As for pong game which is one of the Atari games, the state is defined as image, so it takes longer time to train, besides, the enhancements are all applied to the pong game and the ablation study will also be mentioned later.

2. Your implementation

- How do you obtain the Bellman error for DQN?

The Bellman error in DQN is the difference between the target Q-value and the current Q-value estimate, it's calculated by mean square error, and the target Q-value is calculated by the code segment here.

```
########## YOUR CODE HERE (~10 lines) ##########
# Implement the loss function of DQN and the gradient updates
with torch.no_grad():
    next_q_values = self.target_net(next_states).max(1)[0]
    target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
```

The error represents how far your current Q-value estimates are from what they should be according to the Bellman equation.

- How do you modify DQN to Double DQN?

In DQN implementation, the Q-value is determined directly by the target Q network, which will result in overestimation problem. To overcome this issue, change the Q value estimation process can be an easy solution to solve the overestimation problem, instead of selecting actions and estimating the Q value both in target Q network, Double DQN uses current Q network to select the actions and target Q network estimates the Q value, the implementation is showed below.

```
with torch.no_grad():
    next_actions = self.q_net(next_states).argmax(dim=1, keepdim=True)
    next_q_values = self.target_net(next_states).gather(1, next_actions).squeeze(1)
    target_q_values = rewards + (1 - dones) * (self.gamma ** self.memory.n_steps) * next_q_values
```

- How do you implement the memory buffer for PER?

First, introduce two variable alpha and beta, alpha is used for controlling the prioritization strength, beta is used for controlling the importance sampling. When new transition is going to be added into the replay buffer, the PER will assign a priority value according to its Bellman error to the transition, the higher Bellman

error means higher priority since it has more information to learn. When sampling from the buffer, it first computes the probabilities of all transitions with softmax operation, then using these probabilities to compute the importance sampling. Since it's no longer sampling from the same distribution that your agent originally experienced, this changes the expected value of your updates and can lead to biased Q-value estimates, so we need importance sampling weights to correct these biases. As for the update_priorities, it computes the new priority values according to the absolute value of the Bellman error, the epsilon is for the purpose of avoiding 0. There are some screenshots for implementations of prioritized experience replay.

```
def add(self, transition, error):
    ########## YOUR CODE HERE (for Task 3) ##########
    if len(self.buffer) < self.capacity:
        self.buffer.append(transition)
    else:
        self.buffer[self.pos] = transition
    priority = self.max_priority if error is None else (np.abs(error) + 1e-5) ** self.alpha

    self.priorities[self.pos] = priority
    self.pos = (self.pos + 1) % self.capacity
    ########## END OF YOUR CODE (for Task 3) ##########
    return
```

```
def sample(self, batch_size):
    ########## YOUR CODE HERE (for Task 3) ##########
    N = len(self.buffer)

    if N == 0:
        return None, None, None
    priorities = self.priorities[:N]

    probs = priorities / np.sum(priorities)
    indices = np.random.choice(N, batch_size, replace=False, p=probs)
    samples = [self.buffer[idx] for idx in indices]
    weights = (N * probs[indices]) ** (-self.beta)
    weights = weights / np.max(weights)
    return samples, indices, weights
    ########## END OF YOUR CODE (for Task 3) ##########
```

```
def update_priorities(self, indices, errors):
    ########## YOUR CODE HERE (for Task 3) ##########
    for idx, error in zip(indices, errors):
        if idx < len(self.buffer):
            priority = (np.abs(error) + 1e-5) ** self.alpha
            self.priorities[idx] = priority
            self.max_priority = max(self.max_priority, priority)
    ########## END OF YOUR CODE (for Task 3) ##########
    return
```

- How do you modify the 1-step return to multi-step return?

It needs to modify how the transitions are added into the buffer and how the target

Q values are computed. Instead of storing the transition immediately, the transitions will be stored into another queue called recent transitions, and waiting for more transitions (typically the length of the queue equals n_steps). When collecting n_steps recent transitions or the episode ends, the multi-step transition is created, which is composed of the starting state, starting action, total reward of the recent transitions, final state and final done. The recent transitions are combined into a larger transition to help propagate reward signals faster through value function and can significantly speed up learning.
When calculating the target Q value, there is a modification needed in multi-step return as shown below.

```
with torch.no_grad():
    next_actions = self.q_net(next_states).argmax(dim=1, keepdim=True)
    next_q_values = self.target_net(next_states).gather(1, next_actions).squeeze(1)
    target_q_values = rewards + (1 - dones) * (self.gamma ** self.memory.n_steps) * next_q_values
```

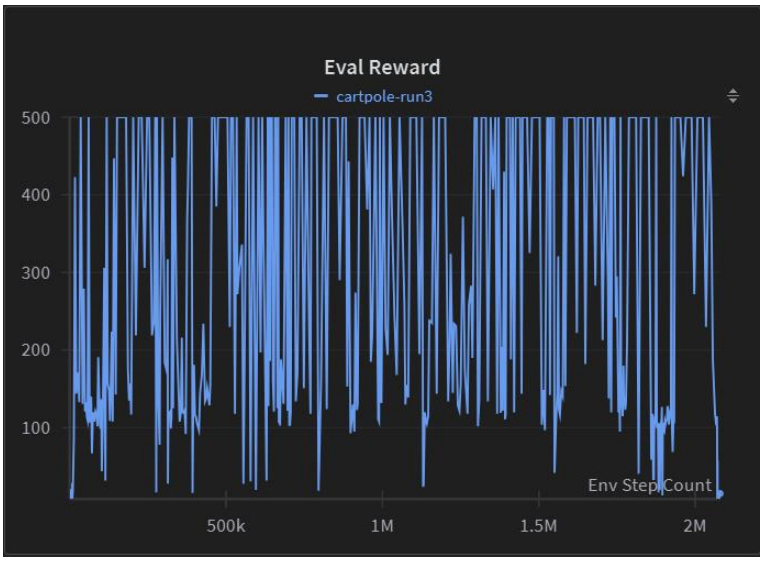
Since the reward now contains the sum of discounted rewards over n steps, it needs to discount the bootstrap value by γ^n instead of just γ.

- Explain how you use Weight & Bias to track the model performance.

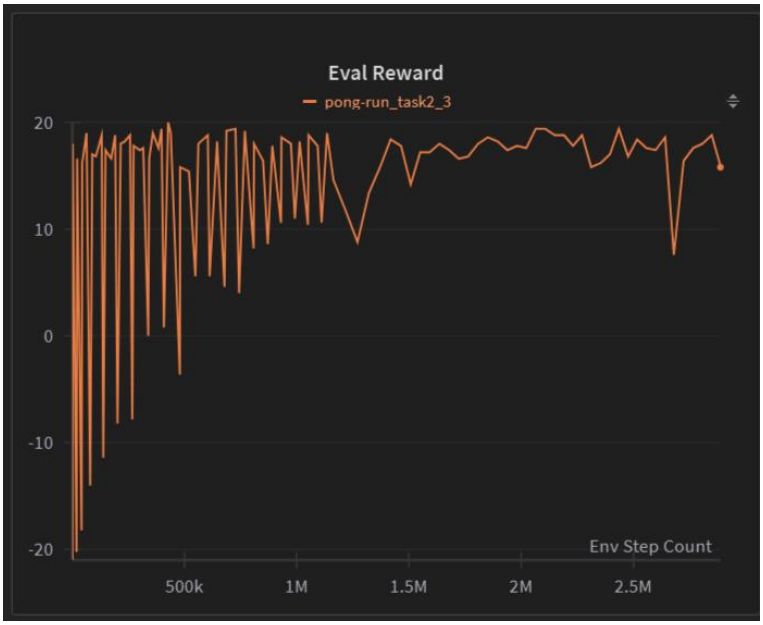
I use Weight & Bias to record several information such as episode, environment step count, train count, buffer size, total reward, evaluation reward, mean Q value and standard deviation of Q value. In addition, I realize Weight & biases can use retraining as well if encountering accident that kills your execution. In general, this tool is user-friendly tool in training model.

3. Analysis and discussions

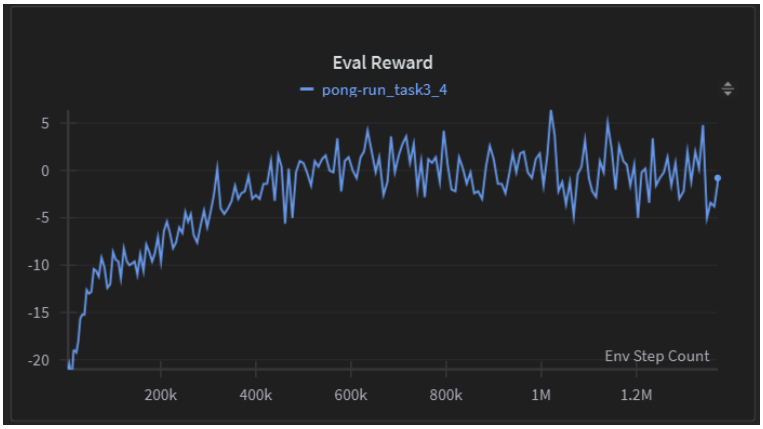
- Plot the training curves (evaluation score versus environment steps) for Task1, Task2 and Task3 respectively.
  Task1:



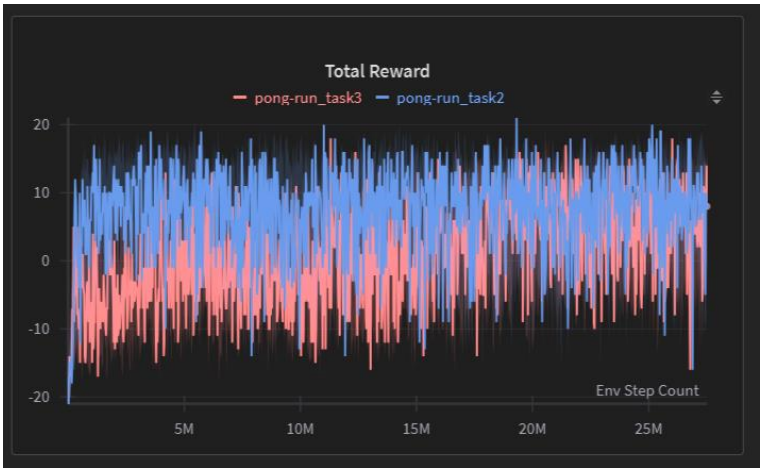
Task2:



Task3:



- Analyze the sample efficiency with and without the DQN enhancements. If possible, perform an ablation study on each technique separately.



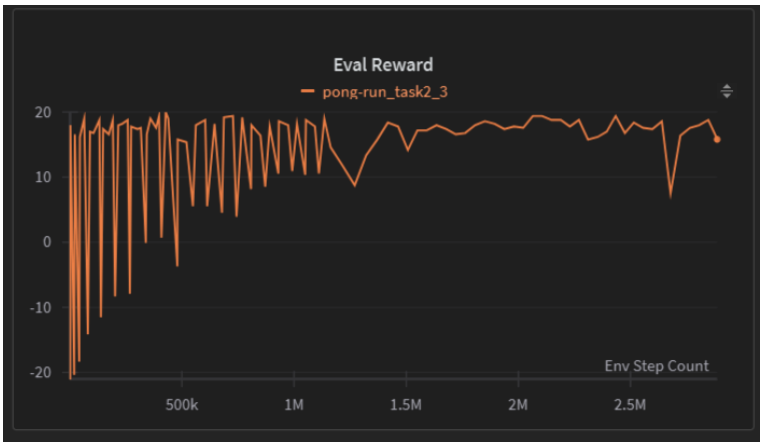
In this analysis, the DQN works better than DDQN, I think it's a kind of probabilistic issue, DQN isn't always better than DDQN and vice versa.

In general, DDQN works better than DQN, it’s just a coincidence. However, I still try to figure out the reasons. First, pong is relatively simple, so the overestimation bias in DQN might not be as harmful as in more complex environments. Second, pong's sparse, binary reward structure might benefit from the slight "optimism" that DQN's overestimation provides.
As for the ablation study, it will be mentioned at the end.

4. Additional analysis on other training strategy

- In task2:

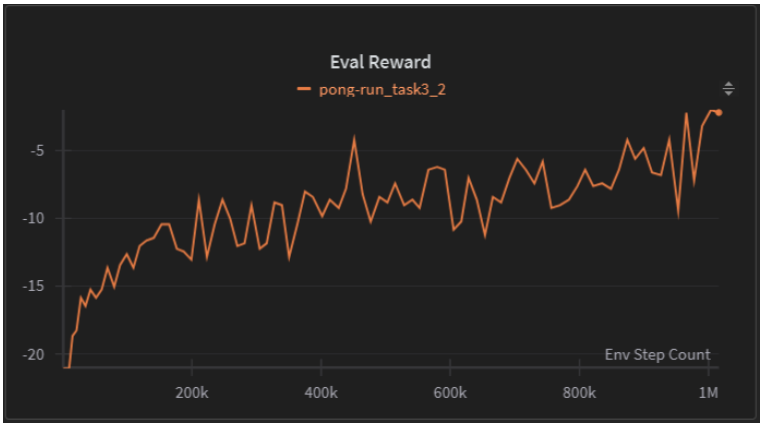
The initial running seems a little unstable, it hardly keeps reward 19 on average in evaluation. To solve the unstable issue, I try a lower epsilon minimum, increase the memory size and batch size. There is the result after modifications.



Although it shows more aggressive in the beginning, but it converges finally and keeps at high rewards successfully.

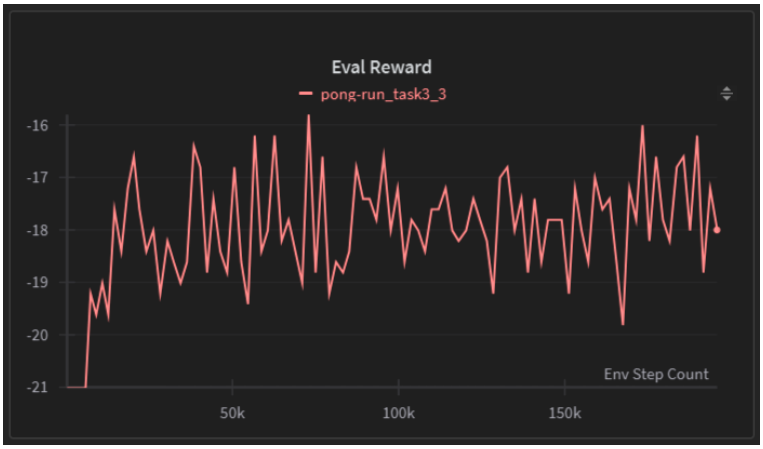
- In task3:
  In this task, the goal is to achieve higher reward as early as possible with three enhancements. Based on the experience in task2, it takes a long time to converge to reward 20, so this task seems to be a tough mission.
  In the first try, with using three enhancements, the result shows not good

enough.

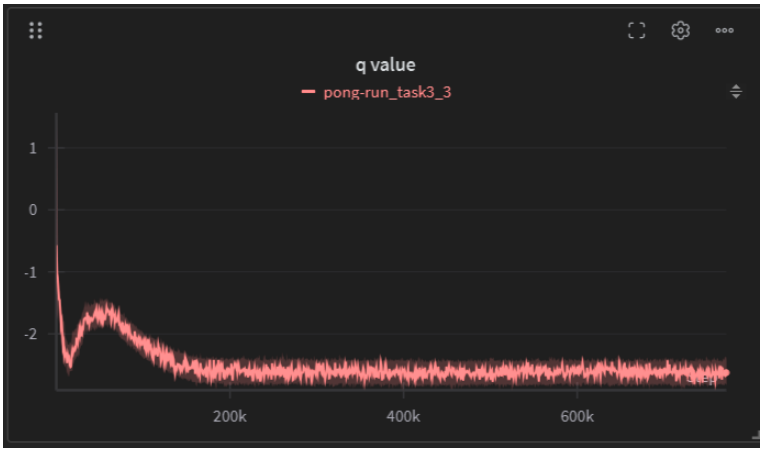


Though the reward rises above -5 within 1 million environment steps, but it's still far from the evaluation standard.

Next, let's try another form of DQN called Dueling DQN which separates the Q value function into value and advantage, the former indicates how good a state is, the other indicates how much better specific actions are relative to others. Besides, I also use the frame skipping to try whether it helps to achieve the goal or not. In frame skipping, the agent repeats the same action for several consecutive frames with the benefits including computational efficiency, smoother behavior and reducing the state space complexity.
Unfortunately, the evaluation reward is still bad.

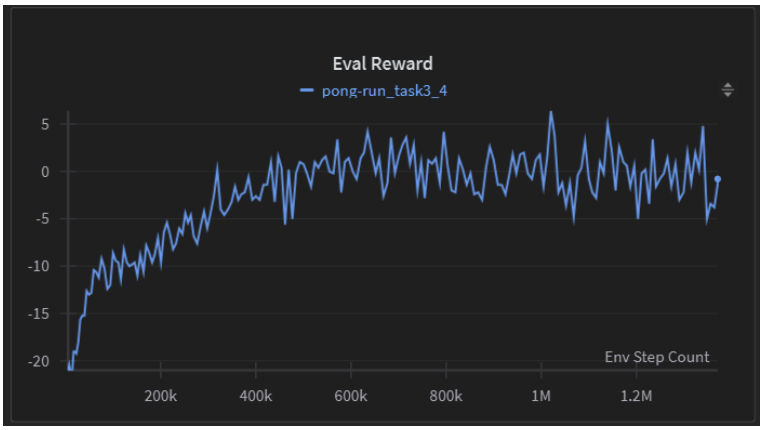


Since my goal here is try to achieve reward 19 before 200k environment step, I only train for few episodes.
In addition, the Q value doesn't perform well either.



Keep trying different methods, let’s introduce warmup stage in run function in order to collect more experiences.

The learning rate scheduler is used in this case as well.



Unfortunately, these methods all failed at the end, even using the reward shaping cannot achieve the goal.

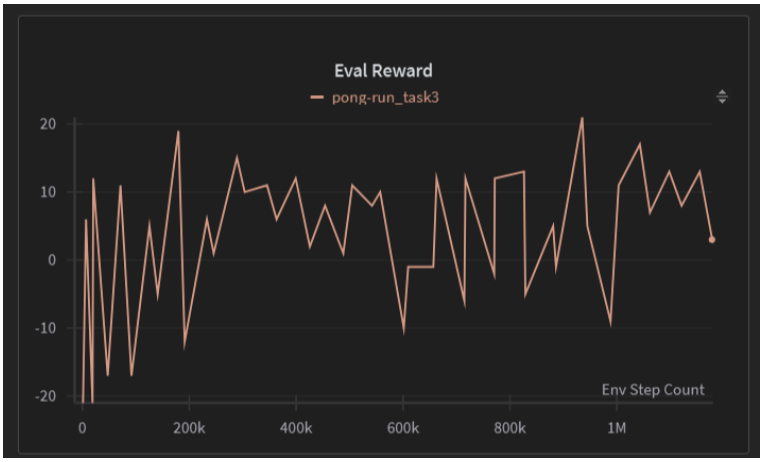
After trying several methods to achieve the goal, all of these are failed.

I think the only way to achieve this is use the retraining method.

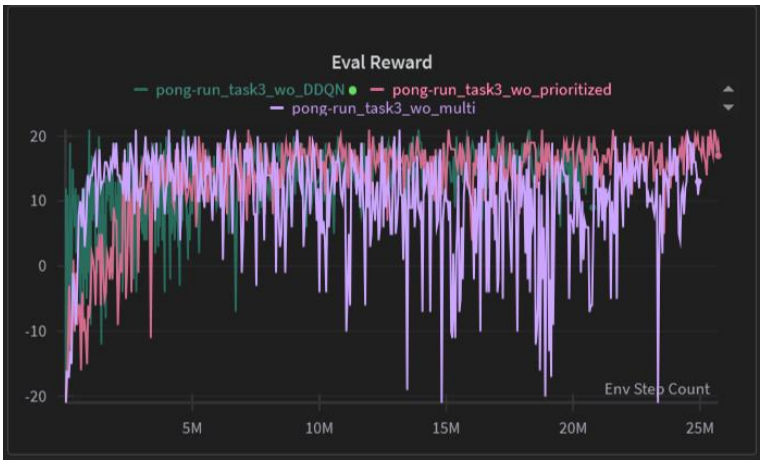
To be honest, I don’t mean to use the retraining method to get the score in task3.

In one of my training processes, the computer was shutdown by electrical accident after long training time, so I decide to use the retraining.

And here is the retraining result.



- Ablation study:
  Here is the result of the ablation study, I think there are a lot of interesting things to illustrate.



It shows the different results with removing each of the enhancing strategies in DQN. At the beginning, the green curve (without DDQN) rises quickly to achieve reward 20, I think the reason may have something to do with the overestimation issue in DQN.
Next, the pink curve (without PER) shows slowest rising in evaluation reward. Since the purpose of PER is to sample those transitions with higher TD error, which means they have more valuable information the model should learn or the transition contains information that contradicts the agent's current understanding of the environment. Take a look at the other two approaches that use the PER, both of them get higher reward at very beginning. As a result, it can substantially improve the agent's policy.
Finally, the purple curve (without multi-step return) shows extremely more unstable than the other two approaches, the main reason in this unstable issue may cause by relying on the immediate reward. Since the multi-step return in consecutive frames can smoothen the impact of single reward, especially when

the agent chooses a random action that causes a bad result, besides, it may have some chances to get better result when the agent takes the consecutive frames into considerations. To sum up, the multi-step return plays an extremely important role in the enhancement of DQN and seems to have fewer negative outliers.

In general, all methods eventually reach high evaluation scores.