DL Lab 5 – Value-Based Reinforcement Learning

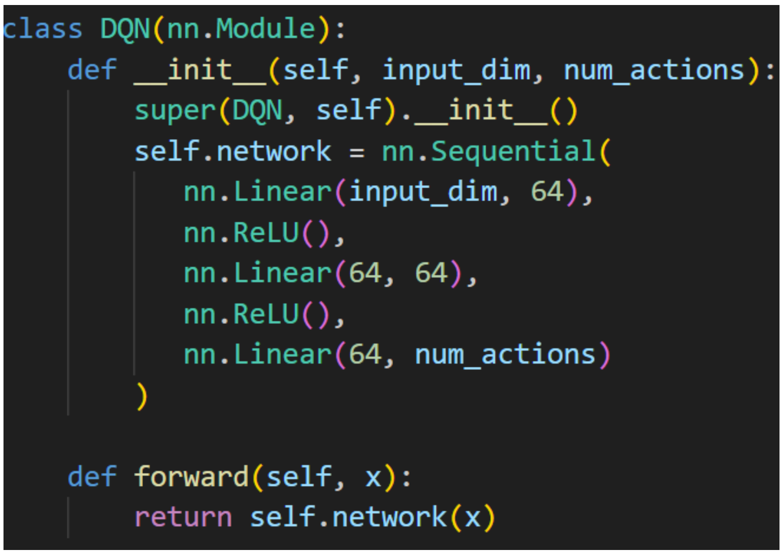
112AB8005

曾大衛

1 Introduction

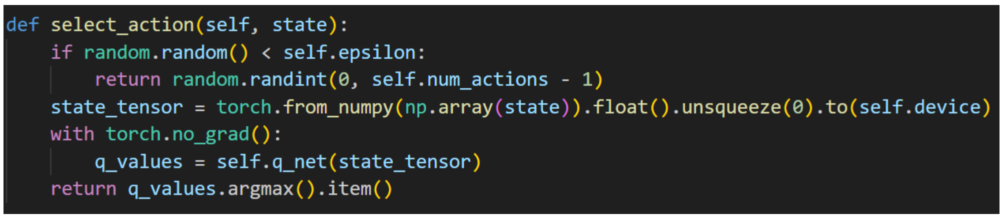
In this lab, I use DQN and DDQN to implement Cartpole and Pong environments respectively. For Cartpole environment, I set up a fully connected neural network architecture and epsilon-greedy policy since it is vector data in Cartpole. On the other hand, for Pong environment, I build a convolutional neural network because of image data in Pong. Moreover, Both Cartpole and Pong environments are with experience replay buffer to sample and update priorities during training procedure, which will calculate loss by actions, state and more from each experience batch. I also make a default value of 3 for multiple returns in DDQN program so that the estimated loss will be reduced, and the training process will be faster. Last, I test the trained models from DQN with two different environments to make videos and compare DQN and DDQN in Pong environment.

2 Task 1



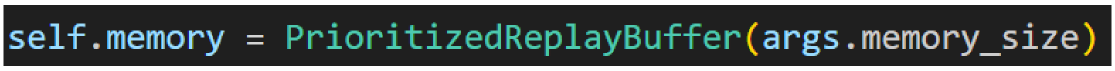
```python
class DQN(nn.Module):
    def __init__(self, input_dim, num_actions):
        super(DQN, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_dim, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, num_actions)
        )

    def forward(self, x):
        return self.network(x)
```

fully connected network

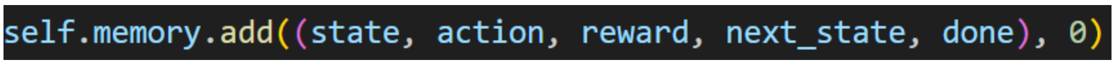


```python
def select_action(self, state):
    if random.random() < self.epsilon:
        return random.randint(0, self.num_actions - 1)
    state_tensor = torch.from_numpy(np.array(state)).float().unsqueeze(0).to(self.device)
    with torch.no_grad():
        q_values = self.q_net(state_tensor)
    return q_values.argmax().item()
```

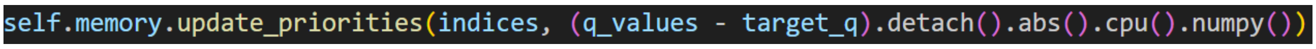
epsilon-greedy policy



```python
self.memory = PrioritizedReplayBuffer(args.memory_size)
```

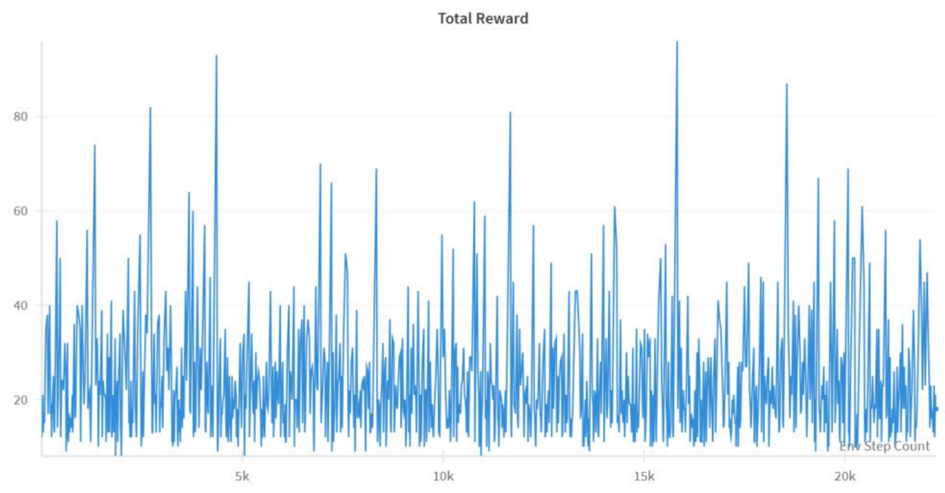


```python
self.memory.add((state, action, reward, next_state, done), 0)
```



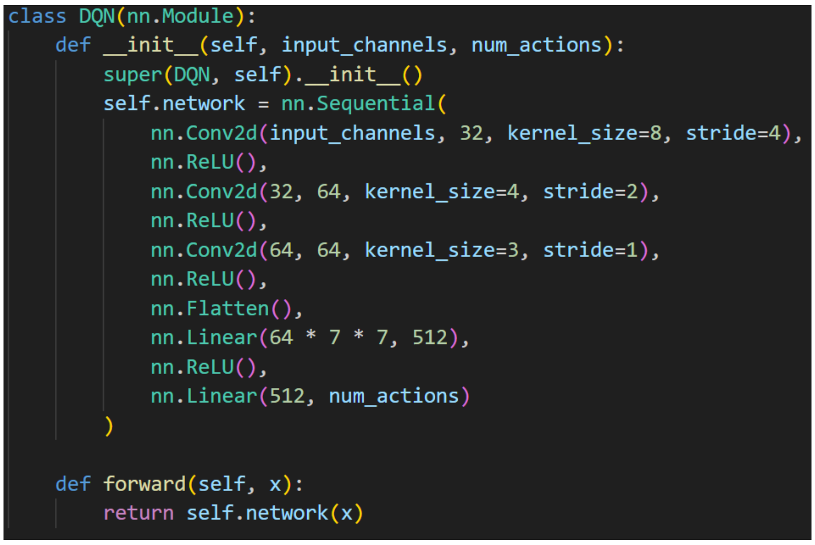
```python
self.memory.update_priorities(indices, (q_values - target_q).detach().abs().cpu().numpy())
```

experience replay buffer



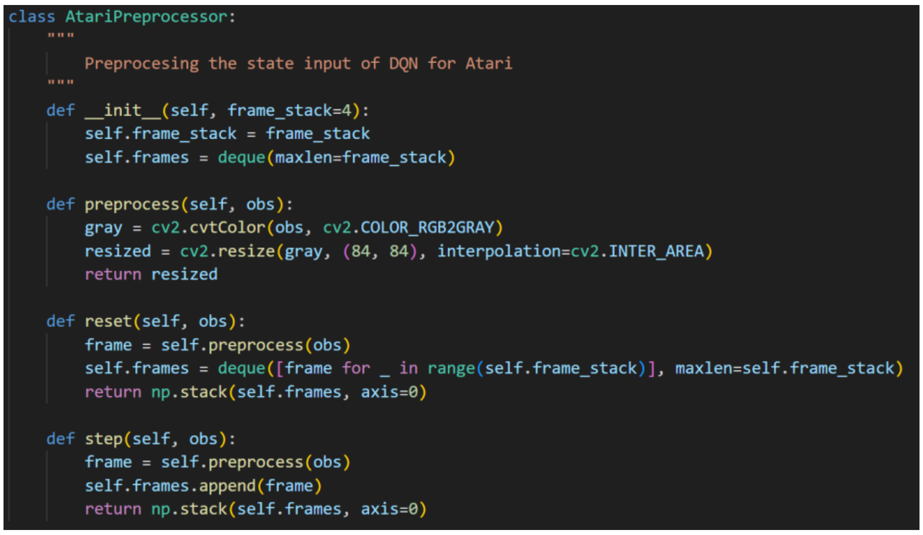
total episodic rewards vs environment steps

3 Task 2

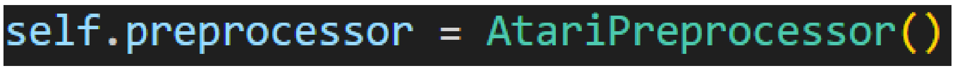


```python
class DQN(nn.Module):
    def __init__(self, input_channels, num_actions):
        super(DQN, self).__init__()
        self.network = nn.Sequential(
            nn.Conv2d(input_channels, 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(64 * 7 * 7, 512),
            nn.ReLU(),
            nn.Linear(512, num_actions)
        )

    def forward(self, x):
        return self.network(x)
```

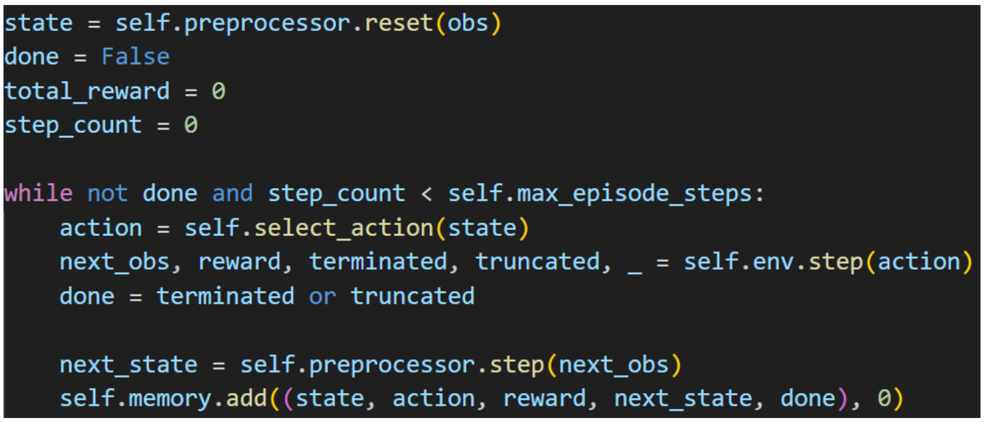
convolutional neural network



```python
class AtariPreprocessor:
    """
        Preprocesing the state input of DQN for Atari
    """
    def __init__(self, frame_stack=4):
        self.frame_stack = frame_stack
        self.frames = deque(maxlen=frame_stack)

    def preprocess(self, obs):
        gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
        resized = cv2.resize(gray, (84, 84), interpolation=cv2.INTER_AREA)
        return resized

    def reset(self, obs):
        frame = self.preprocess(obs)
        self.frames = deque([frame for _ in range(self.frame_stack)], maxlen=self.frame_stack)
        return np.stack(self.frames, axis=0)

    def step(self, obs):
        frame = self.preprocess(obs)
        self.frames.append(frame)
        return np.stack(self.frames, axis=0)
```

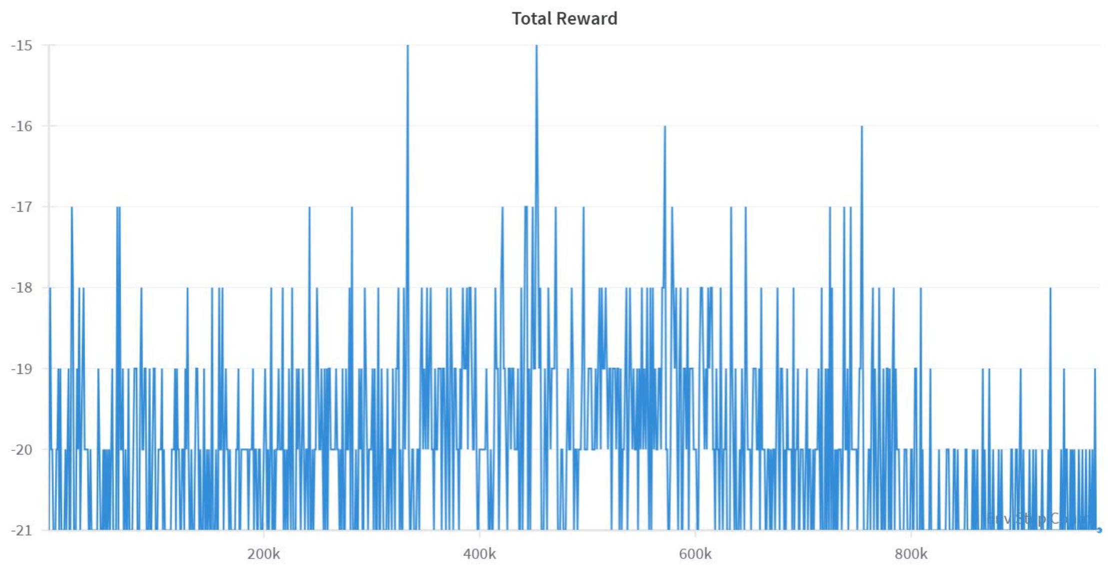


```python
self.preprocessor = AtariPreprocessor()
```



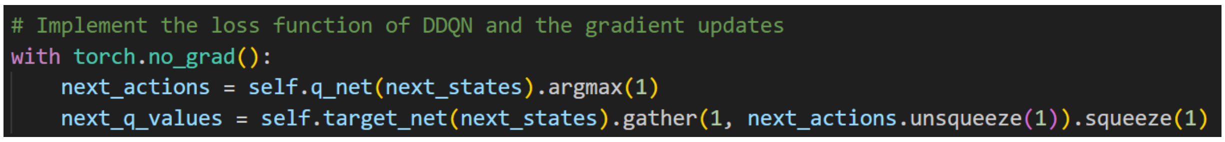
```python
state = self.preprocessor.reset(obs)
done = False
total_reward = 0
step_count = 0

while not done and step_count < self.max_episode_steps:
    action = self.select_action(state)
    next_obs, reward, terminated, truncated, _ = self.env.step(action)
    done = terminated or truncated

    next_state = self.preprocessor.step(next_obs)
    self.memory.add((state, action, reward, next_state, done), 0)
```

use Atari to preprocess input frames



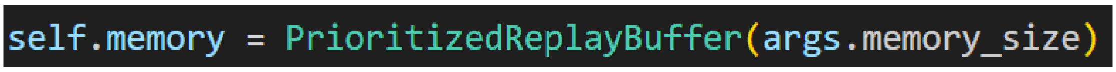
total episodic rewards vs environment steps

4 Task 3

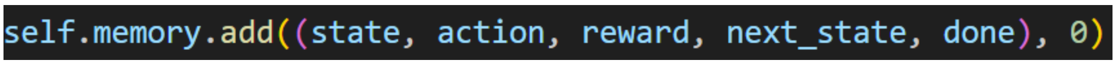


```
# Implement the loss function of DDQN and the gradient updates
with torch.no_grad():
    next_actions = self.q_net(next_states).argmax(1)
    next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
```

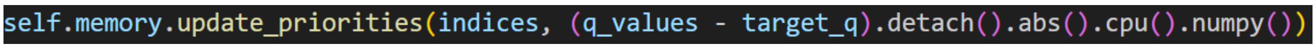
DDQN uses q net to select actions and target net to calculate next q values for these actions



```
self.memory = PrioritizedReplayBuffer(args.memory_size)
```

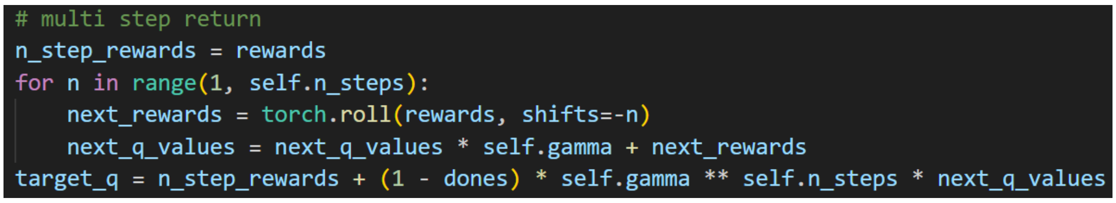


```
self.memory.add((state, action, reward, next_state, done), 0)
```



```
self.memory.update_priorities(indices, (q_values - target_q).detach().abs().cpu().numpy())
```

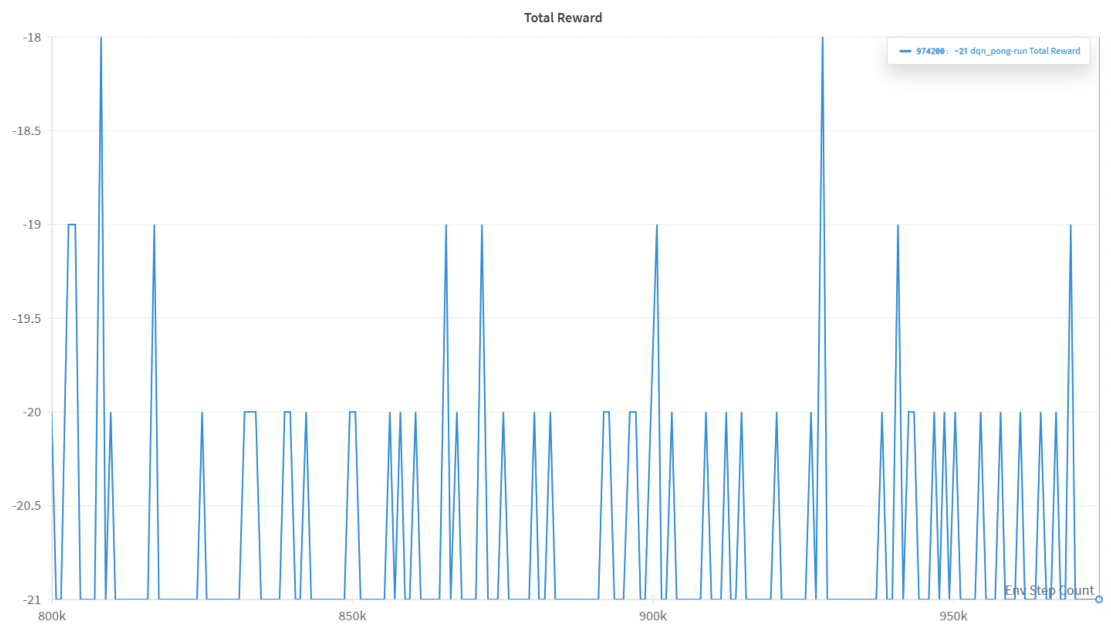
use prioritized experience replay to get priorities



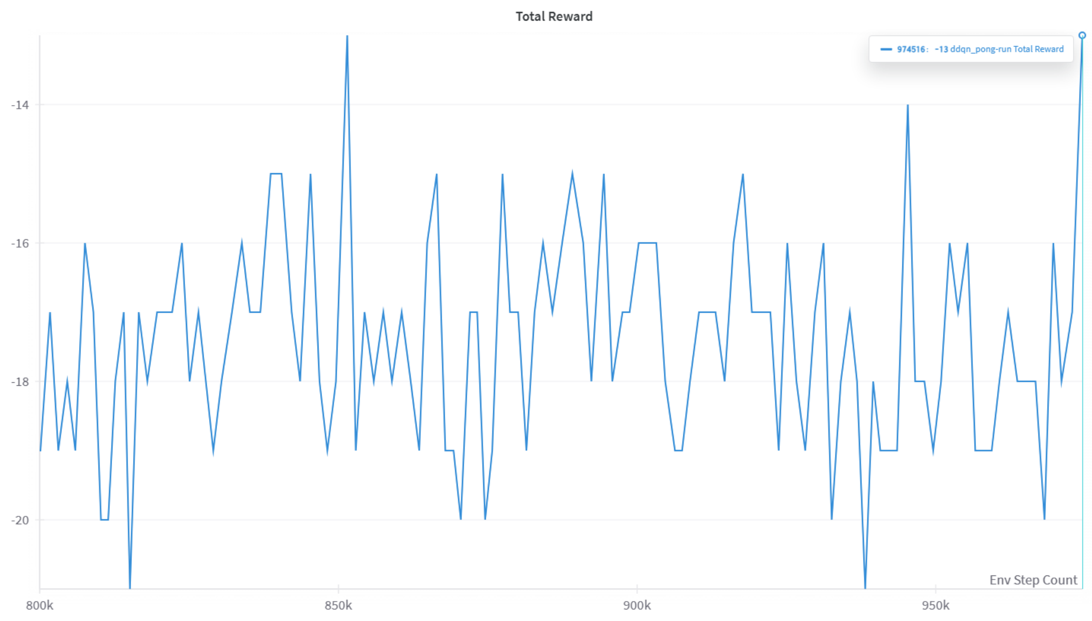
```
# multi step return
n_step_rewards = rewards
for n in range(1, self.n_steps):
    next_rewards = torch.roll(rewards, shifts=-n)
    next_q_values = next_q_values * self.gamma + next_rewards
target_q = n_step_rewards + (1 - dones) * self.gamma ** self.n_steps * next_q_values
```

use multiple returns to minimize loss

DQN vs DDQN in pong environment



DQN's total reward

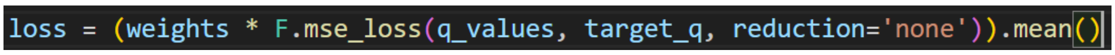


DDQN's total reward

As the photos shown above, I choose reward range from 800k to around 980k, and the learning rate is 1e-4. The highest total reward from DQN is -18, but that from DDQN is -13. Additionally, reward from DDQN is obviously higher than DQN, so it is more efficient for DDQN to learn things than DQN.

5 Discussion

5.1 Bellman error for DQN



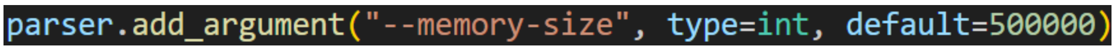
```
loss = (weights * F.mse_loss(q_values, target_q, reduction='none')).mean()
```

The Bellman error is the difference between the predicted q value and the target q value.

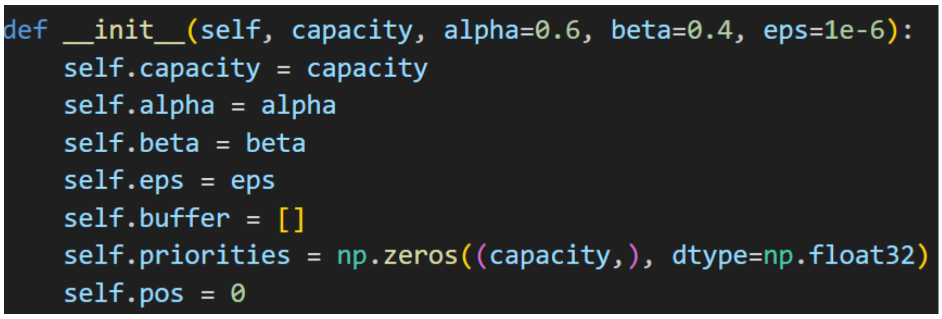
5.2 Modify DQN to Double DQN

As mentioned in section 4(Task 3), in DDQN, the current network selects the next action, and the target network is used to estimate the q value for that action, reducing overestimation bias.

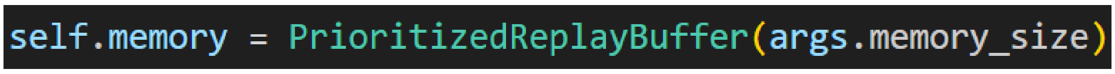
5.3 Implement the memory buffer for PER



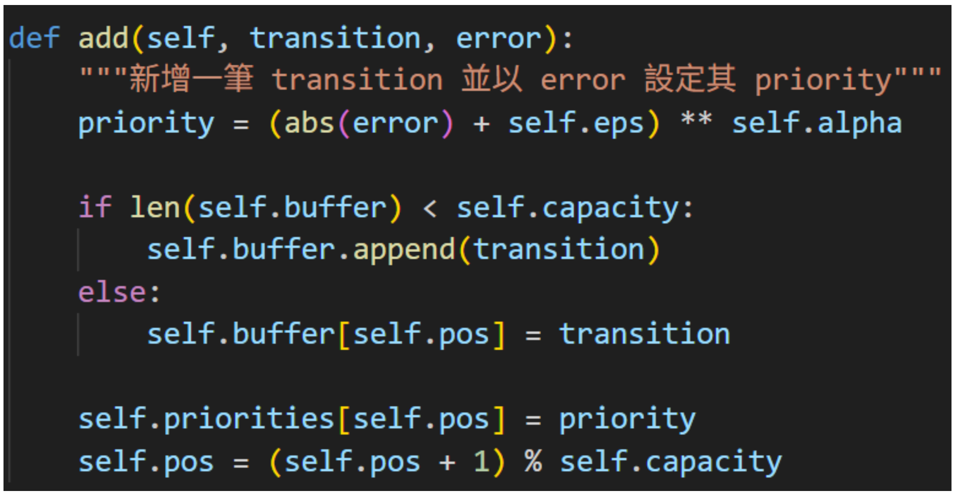
```python
parser.add_argument("--memory-size", type=int, default=500000)
```



```python
def __init__(self, capacity, alpha=0.6, beta=0.4, eps=1e-6):
    self.capacity = capacity
    self.alpha = alpha
    self.beta = beta
    self.eps = eps
    self.buffer = []
    self.priorities = np.zeros((capacity,), dtype=np.float32)
    self.pos = 0
```



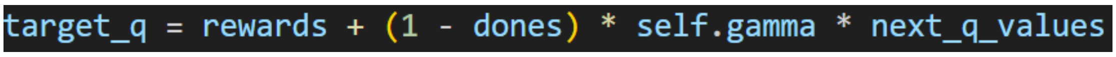
```python
self.memory = PrioritizedReplayBuffer(args.memory_size)
```



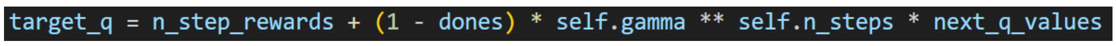
```python
def add(self, transition, error):
    """新增一筆 transition 並以 error 設定其 priority"""
    priority = (abs(error) + self.eps) ** self.alpha

    if len(self.buffer) < self.capacity:
        self.buffer.append(transition)
    else:
        self.buffer[self.pos] = transition

    self.priorities[self.pos] = priority
    self.pos = (self.pos + 1) % self.capacity
```

Firstly, I set up memory size for 500k for PER and instantiate it in DQNAgent. When using PER, it will calculate priority and check memory buffer and default memory size.

5.4 Modify the 1-step return to multi-step return



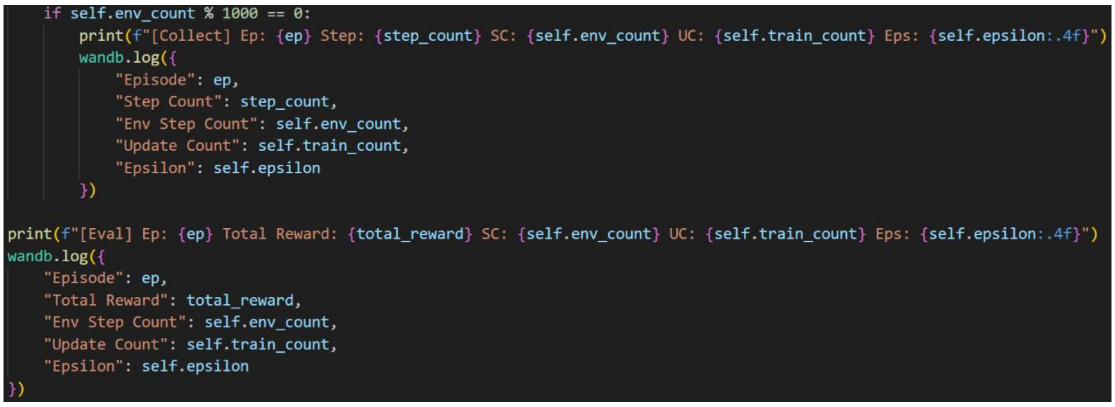
```python
target_q = rewards + (1 - dones) * self.gamma * next_q_values
```



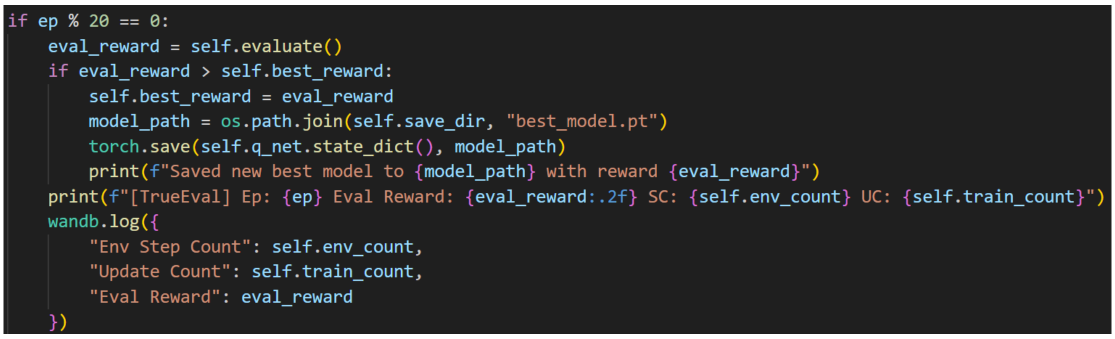
```python
target_q = n_step_rewards + (1 - dones) * self.gamma ** self.n_steps * next_q_values
```

The first image is 1-step return to get target q value, and the second image is multiple returns. As you can see, the most obvious difference between is that it will use n steps to calculate target q in multiple returns though 1-step return will not do it.

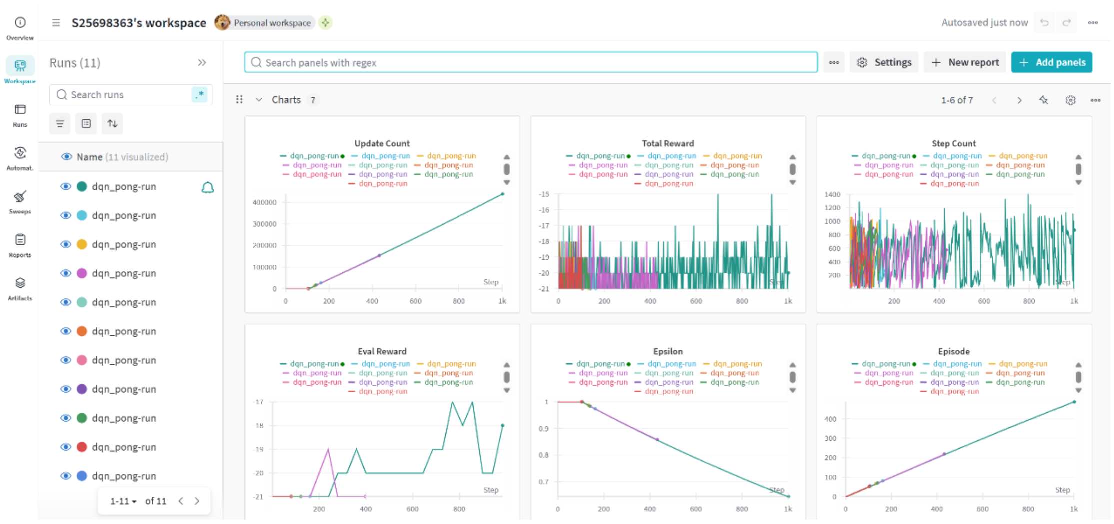
5.5 Use Weight & Bias to track the model performance



```python
    if self.env_count % 1000 == 0:
        print(f"[Collect] Ep: {ep} Step: {step_count} SC: {self.env_count} UC: {self.train_count} Eps: {self.epsilon:.4f}")
        wandb.log({
            "Episode": ep,
            "Step Count": step_count,
            "Env Step Count": self.env_count,
            "Update Count": self.train_count,
            "Epsilon": self.epsilon
        })

print(f"[Eval] Ep: {ep} Total Reward: {total_reward} SC: {self.env_count} UC: {self.train_count} Eps: {self.epsilon:.4f}")
wandb.log({
    "Episode": ep,
    "Total Reward": total_reward,
    "Env Step Count": self.env_count,
    "Update Count": self.train_count,
    "Epsilon": self.epsilon
})
```



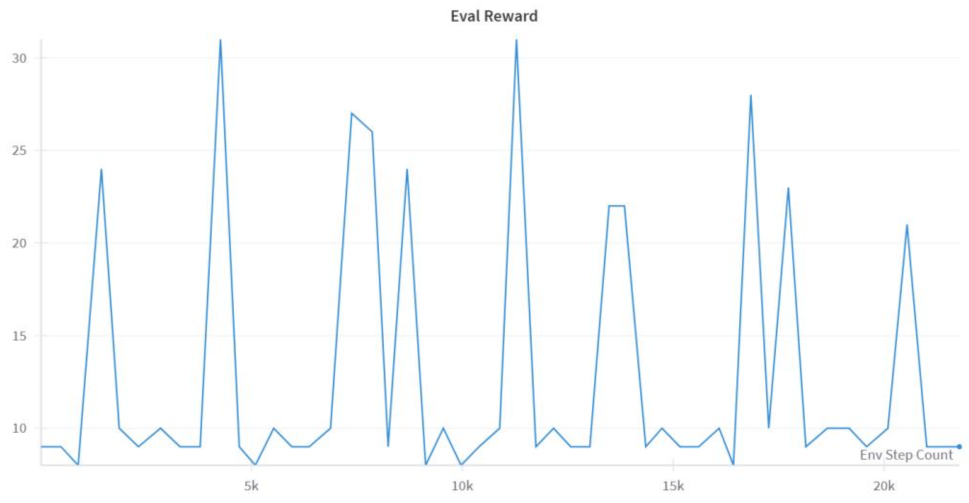
```python
if ep % 20 == 0:
    eval_reward = self.evaluate()
    if eval_reward > self.best_reward:
        self.best_reward = eval_reward
        model_path = os.path.join(self.save_dir, "best_model.pt")
        torch.save(self.q_net.state_dict(), model_path)
        print(f"Saved new best model to {model_path} with reward {eval_reward}")
    print(f"[TrueEval] Ep: {ep} Eval Reward: {eval_reward:.2f} SC: {self.env_count} UC: {self.train_count}")
    wandb.log({
        "Env Step Count": self.env_count,
        "Update Count": self.train_count,
        "Eval Reward": eval_reward
    })
```



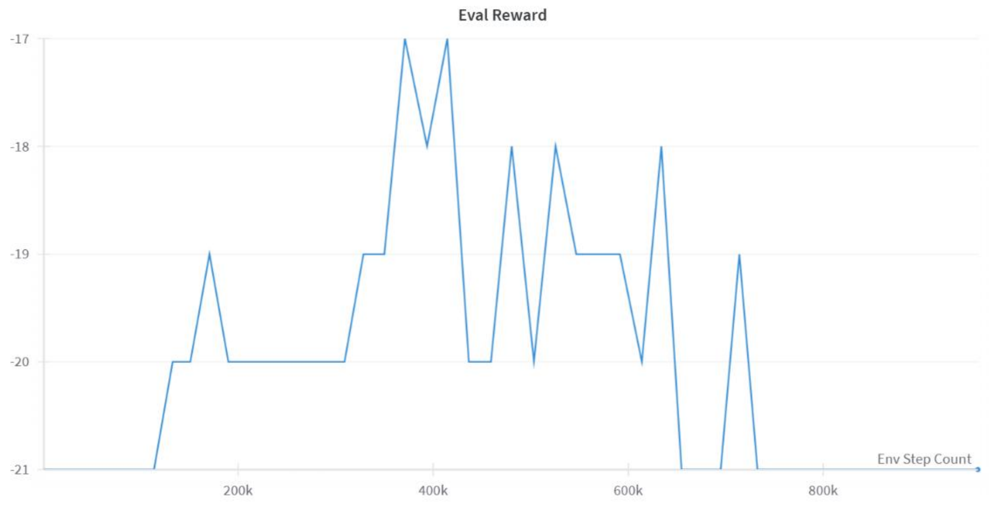
First, get an account from WandB. Second, write the log you would like to record. Third, go to WandB to see the plots for your logs.

5.6 Training curves plots for three tasks

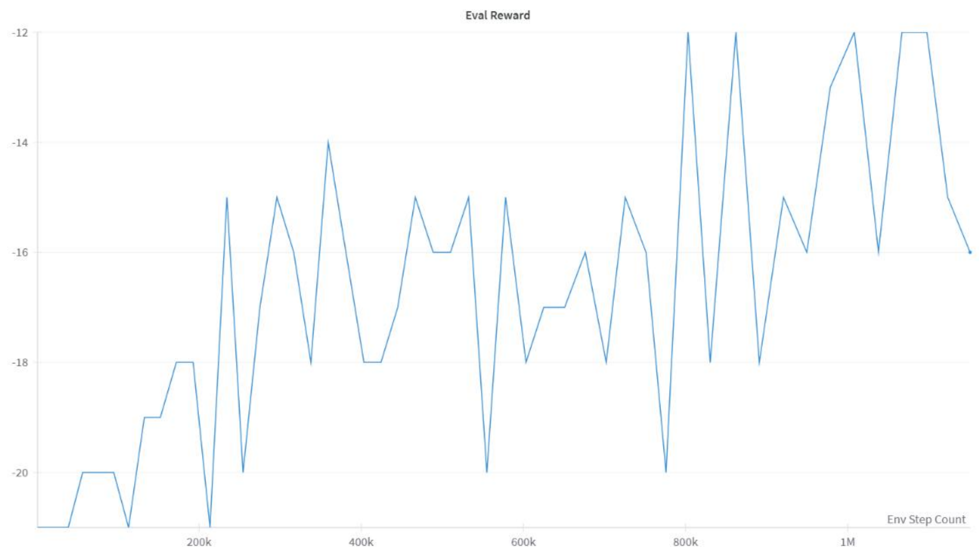
Task 1:



Task 2:



Task 3:



5.7 Analysis for sample efficiency with and without enhanced DQN

The most different points between DQN and enhanced DQN(DDQN) with sample efficiency are 1-step return vs multiple returns and the way they update q values.

For DQN, it uses a replay buffer that stores experiences from previous time steps. During training, the agent samples a random batch of experiences (state, action, reward, next state) from the buffer to compute the loss and update the q values. However, q values from DQN can be overestimated due to the max operator in the Q-learning update, leading to instability and slower convergence. In other words, it will cause losses to get much larger when training and evaluating. For DDQN, it reduces the overestimation bias by using two networks—one for selecting actions and one for evaluating the chosen actions. In addition, DDQN uses n-step returns instead of just 1-step, allowing the agent to learn from longer sequences of rewards, which can speed up the learning process by reducing the variance in reward estimates.