Value-Based Reinforcement Learning

Lab #5 report

B103040039

王宗義

Deep learning

Spring 2025

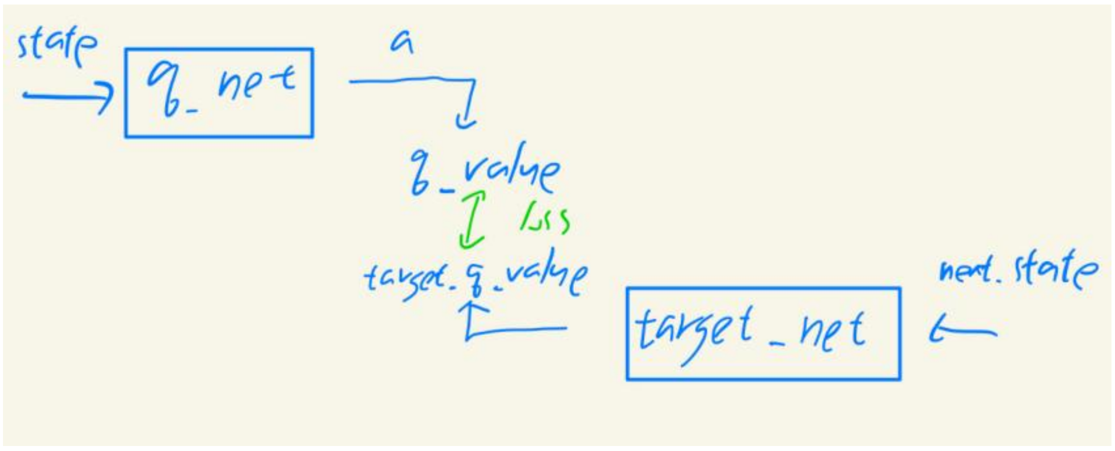
1. Introduction

The purpose of this report is to deeply investigate what kind of learning policies and sets of hyperparameters are the most effective in training Reinforcement Learning (RL) model, focusing on two games of OpenAI gymnasium: "CartPole-v1" and "ALE/Pong-v5". Overall, trying to adjust the parameters that increase training time and ensure the training data is effective is the best strategy to train a model successfully under certain constraints, such as the number of episodes and environment steps. In the subsequent paragraphs, the implementation of certain training strategies will be explained, like DDQN (Double DQN), PER (Prioritized Experience Replay), and multi-step return, introducing the visualization tool: "weight & bias". Furthermore, the result and comparison of each strategy will be provided. Some additional strategies will also be introduced at the end of the essay.

2. Implementation

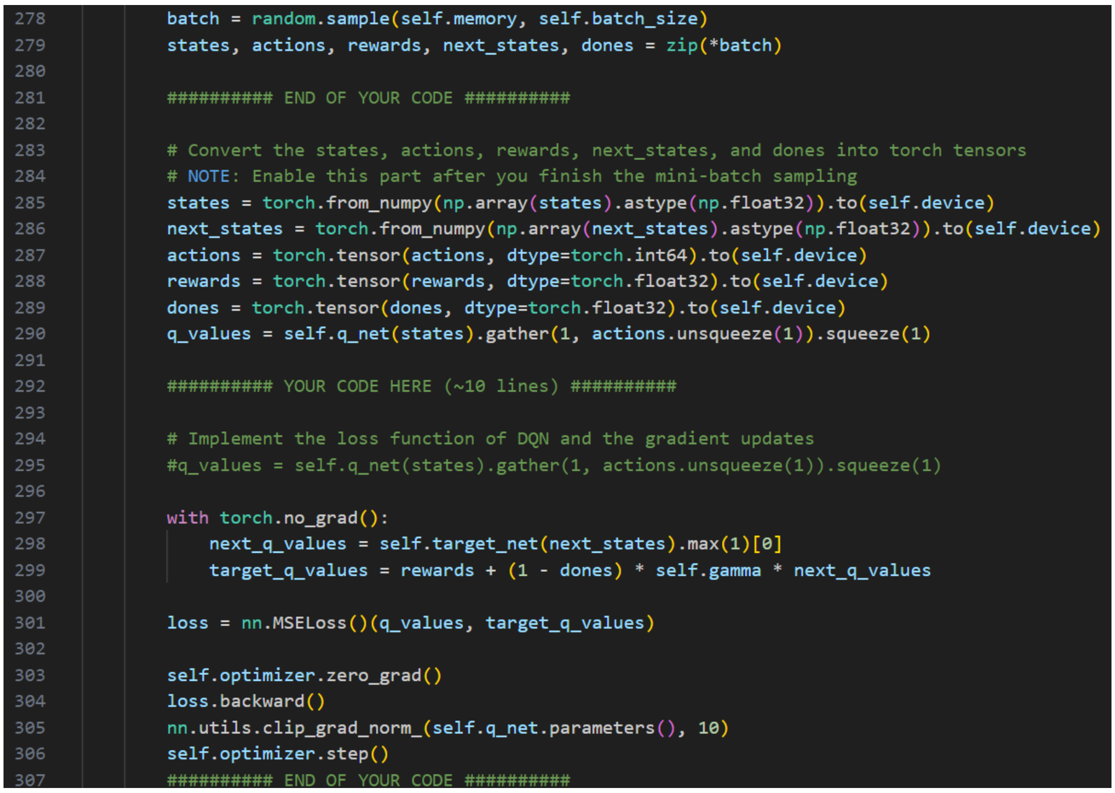
2.1 How to obtain Bellan Error for DQN

Starting with the method to get Bellan Error and loss used for training RL model, the program will first sample a set of training data (line 278 ~ line 279), acquiring the data: "states", "actions", "next_states", and "dones" (1=game over). After putting all of the required variable on cuda device, a "q_value" and "target_q_value" will be calculated by "q_net" and "target_net" respectively. Thereafter, the disparity of the two numbers will be converted into loss by MSE loss function (line 301). Graph 1 illustrates the workflow of calculating the loss.



Graph 1. Workflow of calculating the loss

After calculating the loss, backpropagation and optimizer are utilized to improve our model. Note that the "clip_grad_norm_" function in line 305 is applied to ensure the loss wouldn't exceed a certain limit (10 in this program). Graph 2 depicts the crucial part in training function.

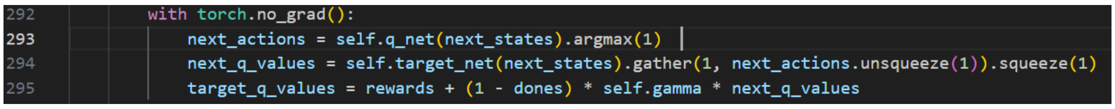


```
278         batch = random.sample(self.memory, self.batch_size)
279         states, actions, rewards, next_states, dones = zip(*batch)
280 
281         ########## END OF YOUR CODE ##########
282 
283         # Convert the states, actions, rewards, next_states, and dones into torch tensors
284         # NOTE: Enable this part after you finish the mini-batch sampling
285         states = torch.from_numpy(np.array(states).astype(np.float32)).to(self.device)
286         next_states = torch.from_numpy(np.array(next_states).astype(np.float32)).to(self.device)
287         actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
288         rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
289         dones = torch.tensor(dones, dtype=torch.float32).to(self.device)
290         q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
291 
292         ########## YOUR CODE HERE (~10 lines) ##########
293 
294         # Implement the loss function of DQN and the gradient updates
295         #q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
296 
297         with torch.no_grad():
298             next_q_values = self.target_net(next_states).max(1)[0]
299             target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
300 
301         loss = nn.MSELoss()(q_values, target_q_values)
302 
303         self.optimizer.zero_grad()
304         loss.backward()
305         nn.utils.clip_grad_norm_(self.q_net.parameters(), 10)
306         self.optimizer.step()
307         ########## END OF YOUR CODE ##########
```

Graph 2. Training function

2.2 How to modify DQN to Double DQN

Normally, the "q_value" is determined only by "target_net" (line 297~ line 299 in Graph 2). However, the "q_value" in this case is always higher than the reality, which means that the model can overestimate or be too positive to the current situation. To deal with the issue, DDQN was introduced. The theory of DDQN is that the action is decided by "q_net", and the current "q_value" is calculated by "target_q_net". In this way, the "q_value" will not be overestimated and the performance of the model can be better. According to Graph 3, the method of calculating "q_value" is changed. This time we first get the action decided by "q_net", and then calculate "target_q_values" (line 295).

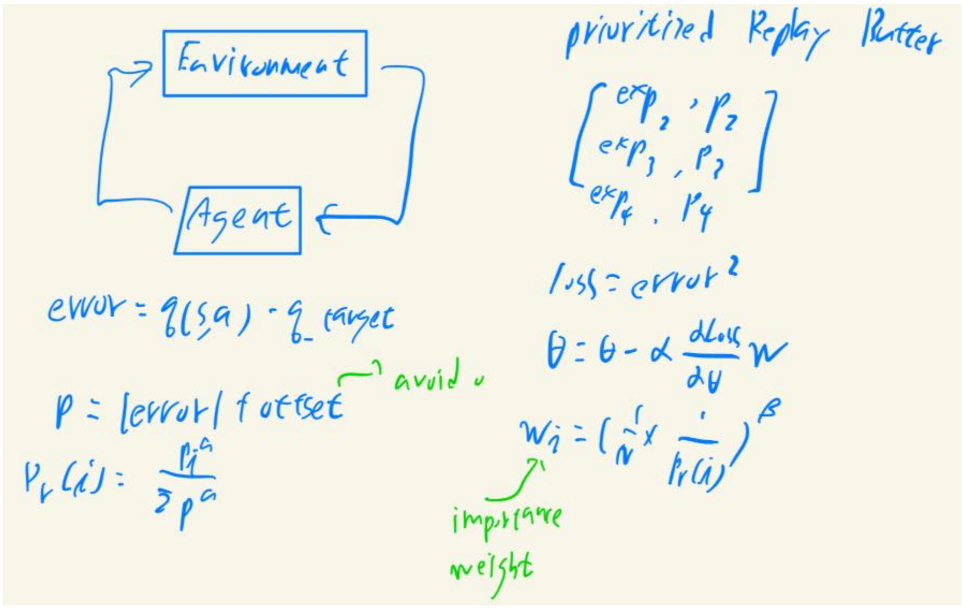


```
292         with torch.no_grad():
293             next_actions = self.q_net(next_states).argmax(1)
294             next_q_values = self.target_net(next_states).gather(1, next_actions.unsqueeze(1)).squeeze(1)
295             target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
```

Graph 3. Implementation of DDQN

2.3 How to implement memory buffer for PER

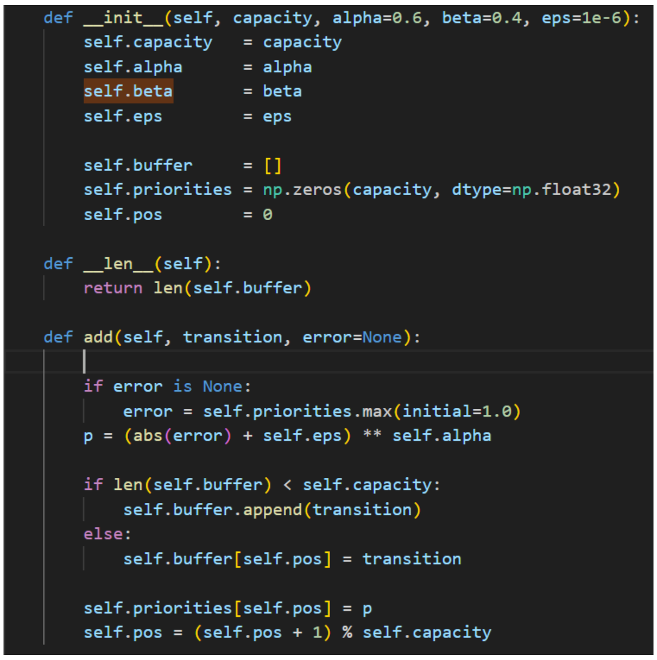
In order to implement PER in DQN, the structure of memory have to be changed. In addition, three new functions on manipulating this new memory are also required. In this case, a class called “PrioritizedReplayBuffer” is introduced, and the blueprint of PER is provided as Graph 4 below.



Graph 4. blueprint of PER

2.3.1 __init__ and add functions

Starting with initialize the class, the priorities are initialized to 0 to all of the data in the buffer. In terms of the “add” function, the program will first check whether the error (TD error) is zero or not. If it is, we treat the transition as “very important” so it gets sampled soon, or use the current maximum priority (or 1 if buffer is empty). Thereafter, the p value will be calculated in the following steps. Note that the program will also check whether the buffer is full. If the buffer is full, the new p value will overwrite the data, otherwise it will be appended. Graph 5 shows the detailed implementation.

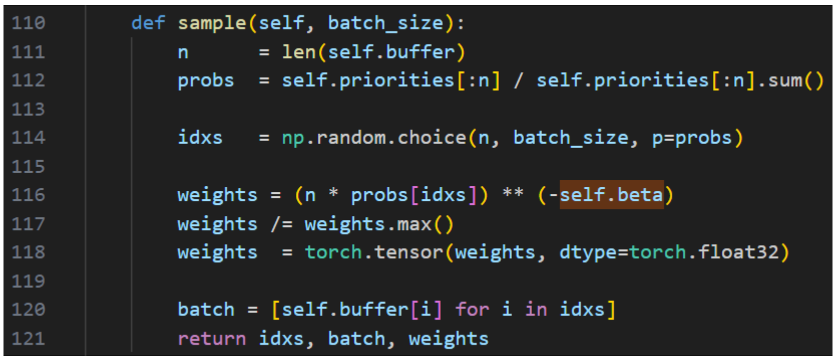


```python
def __init__(self, capacity, alpha=0.6, beta=0.4, eps=1e-6):
    self.capacity   = capacity
    self.alpha      = alpha
    self.beta       = beta
    self.eps        = eps

    self.buffer     = []
    self.priorities = np.zeros(capacity, dtype=np.float32)
    self.pos        = 0

def __len__(self):
    return len(self.buffer)

def add(self, transition, error=None):

    if error is None:
        error = self.priorities.max(initial=1.0)
    p = (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] = p
    self.pos = (self.pos + 1) % self.capacity
```

Graph 5. __init__ and add functions

2.3.2 Sample function

When sampling, the first n priorities will be converted into probability distribution P(i). Afterward, indices with batch_size will be sampled according to that distribution. Then importance-sampling weights will be computed. In the end, the function will assemble the actual transitions and return everything so that the agent can learn and later update the same indices. Graph 6 shows the detailed implementation.

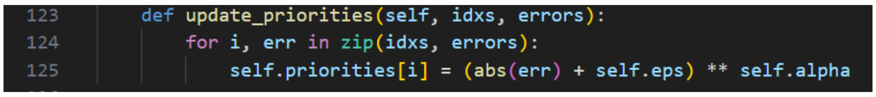


```python
    def sample(self, batch_size):
        n       = len(self.buffer)
        probs   = self.priorities[:n] / self.priorities[:n].sum()

        idxs    = np.random.choice(n, batch_size, p=probs)

        weights = (n * probs[idxs]) ** (-self.beta)
        weights /= weights.max()
        weights  = torch.tensor(weights, dtype=torch.float32)

        batch = [self.buffer[i] for i in idxs]
        return idxs, batch, weights
```

Graph 6. Sample function

2.3.3 Update priority function

After the agent computes a new TD-error for each sampled transition, the function will overwrite its priority so that the buffer stays up-to-date. Graph 7 shows the detailed implementation.

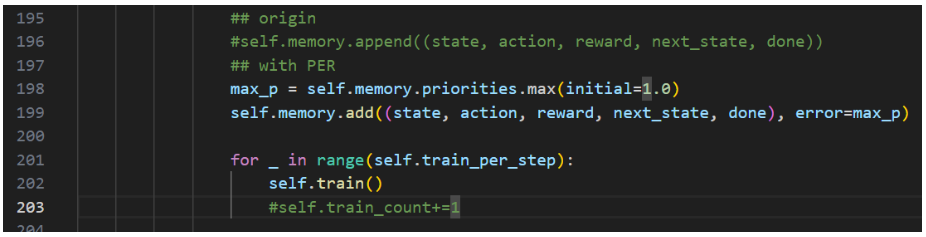


```python
    def update_priorities(self, idxs, errors):
        for i, err in zip(idxs, errors):
            self.priorities[i] = (abs(err) + self.eps) ** self.alpha
```

Graph 7. Update priority function

2.3.4 Implementation

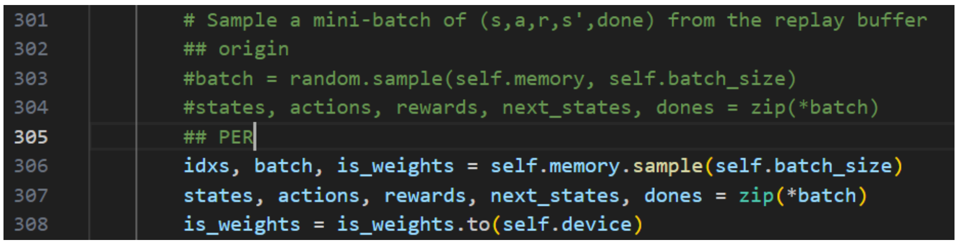
While the agent get the training data from the environment, the program will first acquire the largest priority in the current memory. Then take the priority as the error, adding to the memory (line 199).



```python
195                 ## origin
196                 #self.memory.append((state, action, reward, next_state, done))
197                 ## with PER
198                 max_p = self.memory.priorities.max(initial=1.0)
199                 self.memory.add((state, action, reward, next_state, done), error=max_p)
200 
201                 for _ in range(self.train_per_step):
202                     self.train()
203                     #self.train_count+=1
```

Graph 8. Implementation of add function

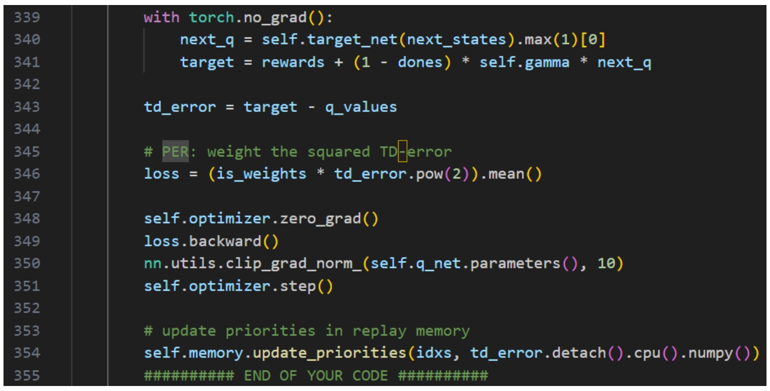
As the process of sampling, the only significant difference can be observed is that the new variable "is_weights" is introduced for later computation.



```python
        # Sample a mini-batch of (s,a,r,s',done) from the replay buffer
        ## origin
        #batch = random.sample(self.memory, self.batch_size)
        #states, actions, rewards, next_states, dones = zip(*batch)
        ## PER
        idxs, batch, is_weights = self.memory.sample(self.batch_size)
        states, actions, rewards, next_states, dones = zip(*batch)
        is_weights = is_weights.to(self.device)
```

Graph 9. Implementation of sample function

At the end of training process, update priorities function is utilized to update the priority of each data. At the same time, the variable "is_weights" is used to calculate the loss.



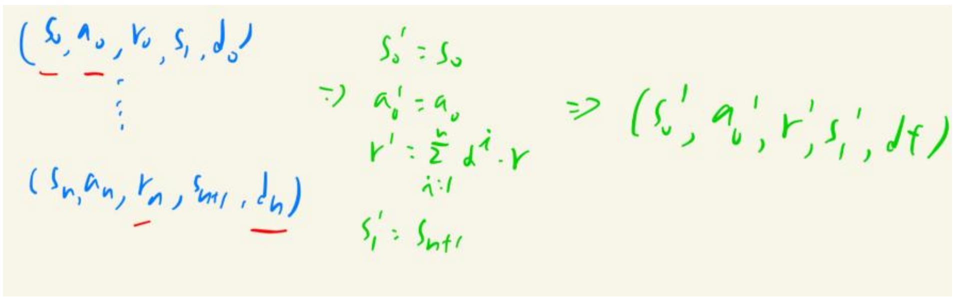
```python
        with torch.no_grad():
            next_q = self.target_net(next_states).max(1)[0]
            target = rewards + (1 - dones) * self.gamma * next_q

        td_error = target - q_values

        # PER: weight the squared TD-error
        loss = (is_weights * td_error.pow(2)).mean()

        self.optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self.q_net.parameters(), 10)
        self.optimizer.step()

        # update priorities in replay memory
        self.memory.update_priorities(idxs, td_error.detach().cpu().numpy())
        ########## END OF YOUR CODE ##########
```

Graph 10. Implementation of update priorities function

2.4 How to implement one-step return to multi-step return

2.4.1 add_n_step_transition function

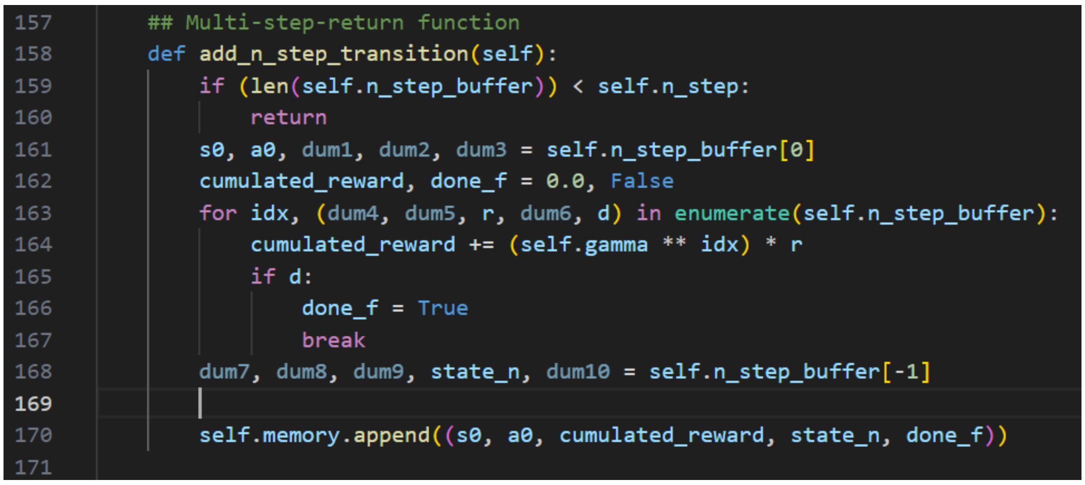
In terms of multi-step return, I introduced a new function called "add_n_step_transition". The blueprint of the function is provided as graph 11. Briefly speaking, the information of several training data will be compressed into one data.



$(s_0, a_0, r_0, s_1, d_0)$
$\vdots$
$(s_n, a_n, r_n, s_{n+1}, d_n)$
$\Rightarrow$
$s_0' = s_0$
$a_0' = a_0$
$r' = \sum_{i=1}^{n} d^i \cdot r$
$s_1' = s_{n+1}$
$\Rightarrow (s_0', a_0', r', s_1', df)$

Graph 11. Blueprint of add_n_step_transition function

Graph 12 illustrates the detail of the function. At first, the program will check whether the length of step_buffer is smaller than the n_step we set as an argument. Afterward, the state and action of the first data will be collected. Then, cumulated_reward will be calculated (line 164) and the variable to check whether it is finished, done_f, will be determined. In addition, the last next_state will also be collected. As all of the 5 elements are collected or computed, a new training data will be created and pushed into memory.

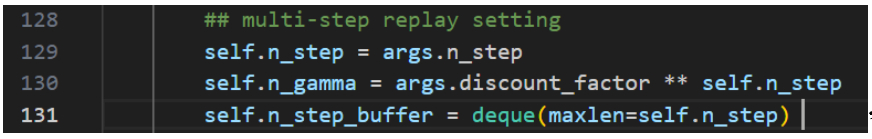


```
157    ## Multi-step-return function
158    def add_n_step_transition(self):
159        if (len(self.n_step_buffer)) < self.n_step:
160            return
161        s0, a0, dum1, dum2, dum3 = self.n_step_buffer[0]
162        cumulated_reward, done_f = 0.0, False
163        for idx, (dum4, dum5, r, dum6, d) in enumerate(self.n_step_buffer):
164            cumulated_reward += (self.gamma ** idx) * r
165            if d:
166                done_f = True
167                break
168        dum7, dum8, dum9, state_n, dum10 = self.n_step_buffer[-1]
169        
170        self.memory.append((s0, a0, cumulated_reward, state_n, done_f))
171
```

Graph 12. Detail information of add_n_step_transition function

2.4.2 implementation

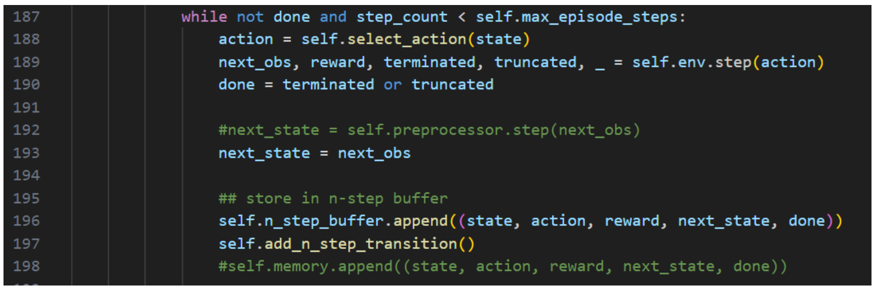
As for the implementation, the training data will first be collected into a container called “n_step_buffer” (graph 13 shows the initialization of it). While the container is full, add_n_step_transition function will transfer all of the data in “n_step_buffer” into the memory. Graph 14 shows the detailed implementation.



```python
        ## multi-step replay setting
        self.n_step = args.n_step
        self.n_gamma = args.discount_factor ** self.n_step
        self.n_step_buffer = deque(maxlen=self.n_step)
```

”

Graph 13. Initialization of the objects needed

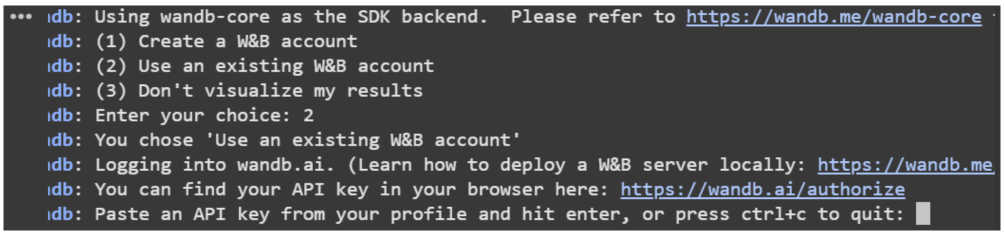


```python
            while not done and step_count < self.max_episode_steps:
                action = self.select_action(state)
                next_obs, reward, terminated, truncated, _ = self.env.step(action)
                done = terminated or truncated

                #next_state = self.preprocessor.step(next_obs)
                next_state = next_obs

                ## store in n-step buffer
                self.n_step_buffer.append((state, action, reward, next_state, done))
                self.add_n_step_transition()
                #self.memory.append((state, action, reward, next_state, done))
```

Graph 14. Implementation of multi-step return

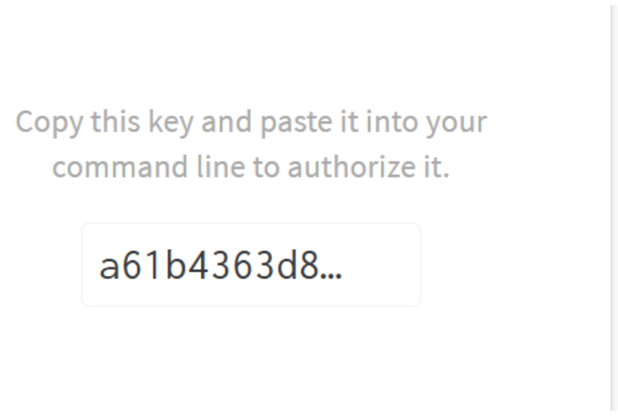
2.5 How to use weight & bias to track the performance

While running the code, the package called “Wandb” will ask you the following action, (1) for creating account, (2) for using existed account, and (3) for choosing not to visualize the performance. If you have already created your own account, input number 2 and “Wandb” will randomly generate a private key for you to observe the performance of models. As Graph 15 below. After entering the key from the website https://wandb.ai/authorize (Graph 16), you will finally receive a website to check the performance (Graph 17).



```
idb: Using wandb-core as the SDK backend.  Please refer to https://wandb.me/wandb-core
idb: (1) Create a W&B account
idb: (2) Use an existing W&B account
idb: (3) Don't visualize my results
idb: Enter your choice: 2
idb: You chose 'Use an existing W&B account'
idb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me
idb: You can find your API key in your browser here: https://wandb.ai/authorize
idb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
```

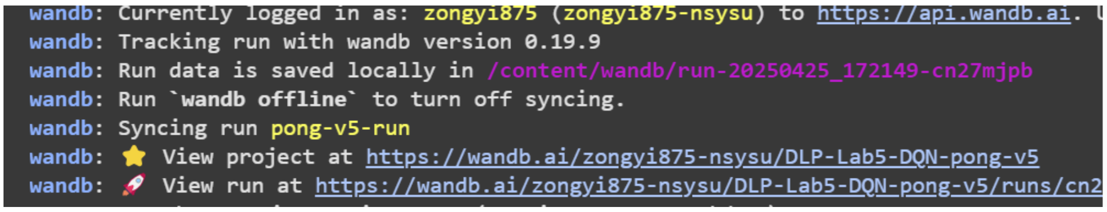
Graph 15. Messages from “Wandb”



Copy this key and paste it into your command line to authorize it.

a61b4363d8...

Graph 16. Key generated by "Wandb"



```
wandb: Currently logged in as: zongyi875 (zongyi875-nsysu) to https://api.wandb.ai.
wandb: Tracking run with wandb version 0.19.9
wandb: Run data is saved locally in /content/wandb/run-20250425_172149-cn27mjpb
wandb: Run `wandb offline` to turn off syncing.
wandb: Syncing run pong-v5-run
wandb: ⭐ View project at https://wandb.ai/zongyi875-nsysu/DLP-Lab5-DQN-pong-v5
wandb: 🚀 View run at https://wandb.ai/zongyi875-nsysu/DLP-Lab5-DQN-pong-v5/runs/cn2
```

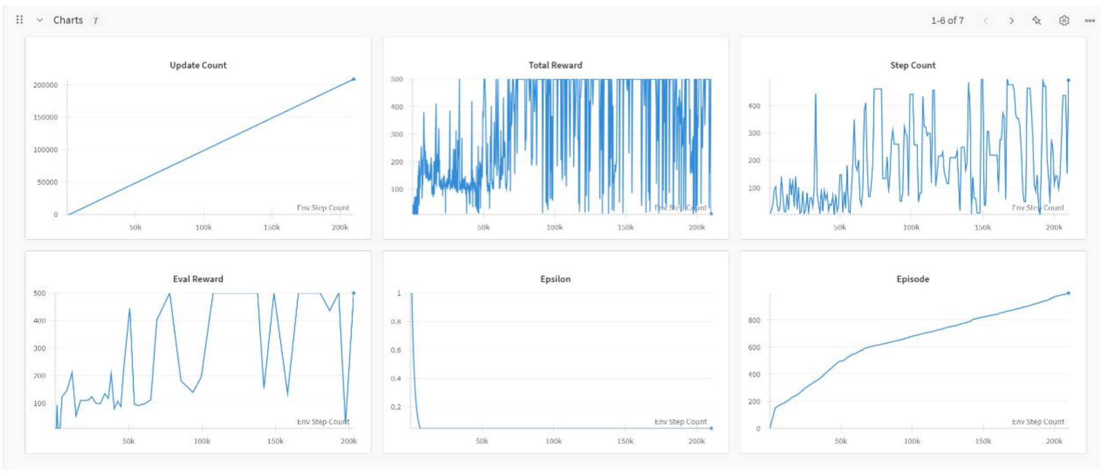
Graph 17. Website of getting access to the graph

3. Analysis & Discussions

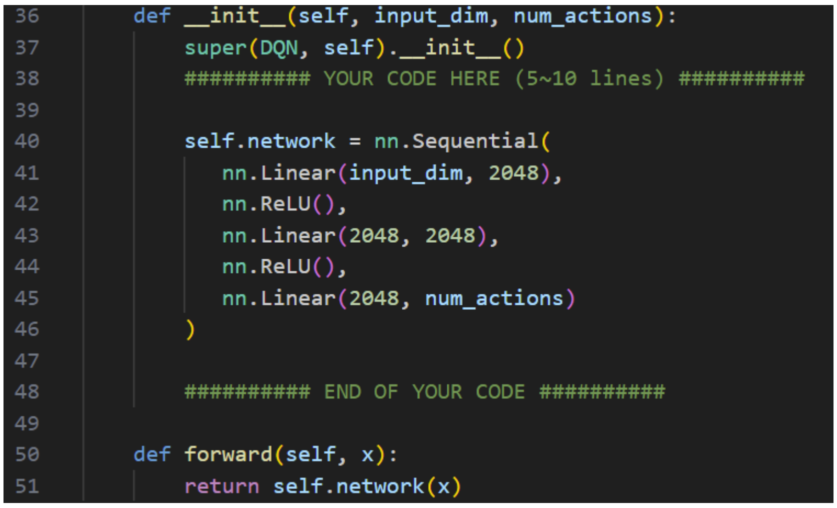
3.1 Training curve of task 1

Graph 18 depicts the training curve of task (cartpole). At the end of training, the model reached the highest score, 500, despite of some fluctuations. Another observation worth mentioning is that the first highest score occurred at nearly 500K (environment steps).

Note that I change the structure of model as shown graph 19.



Graph 18. Training curve of task 1

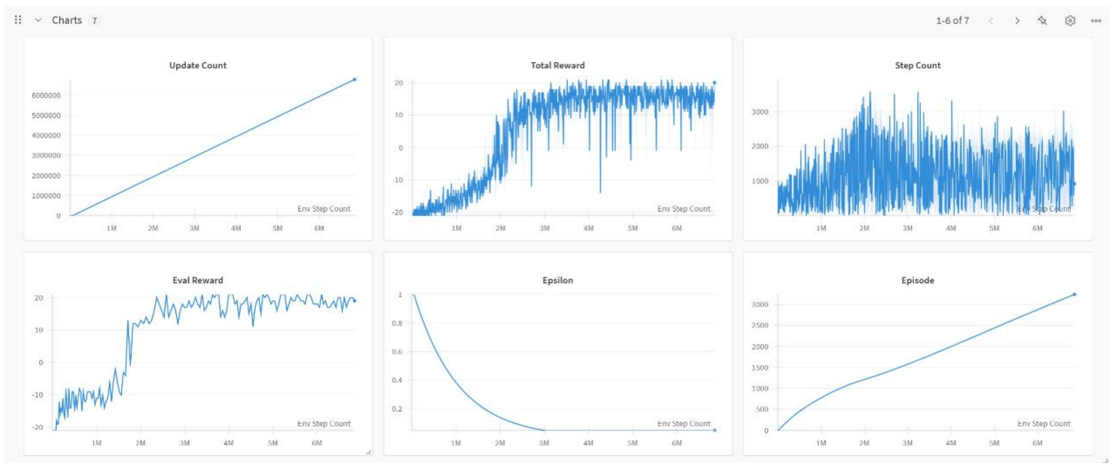


```python
    def __init__(self, input_dim, num_actions):
        super(DQN, self).__init__()
        ########## YOUR CODE HERE (5~10 lines) ##########

        self.network = nn.Sequential(
            nn.Linear(input_dim, 2048),
            nn.ReLU(),
            nn.Linear(2048, 2048),
            nn.ReLU(),
            nn.Linear(2048, num_actions)
        )

        ########## END OF YOUR CODE ##########

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

Graph 19. The structure of DQN model in task 1

3.2 Training curve of task 2

Graph 20 depicts the training curve of task 2 (ALE/pong). According to the data, the performance of the model was bad at the beginning of training. However, after training for roughly 2M environment steps, the performance finally became better.



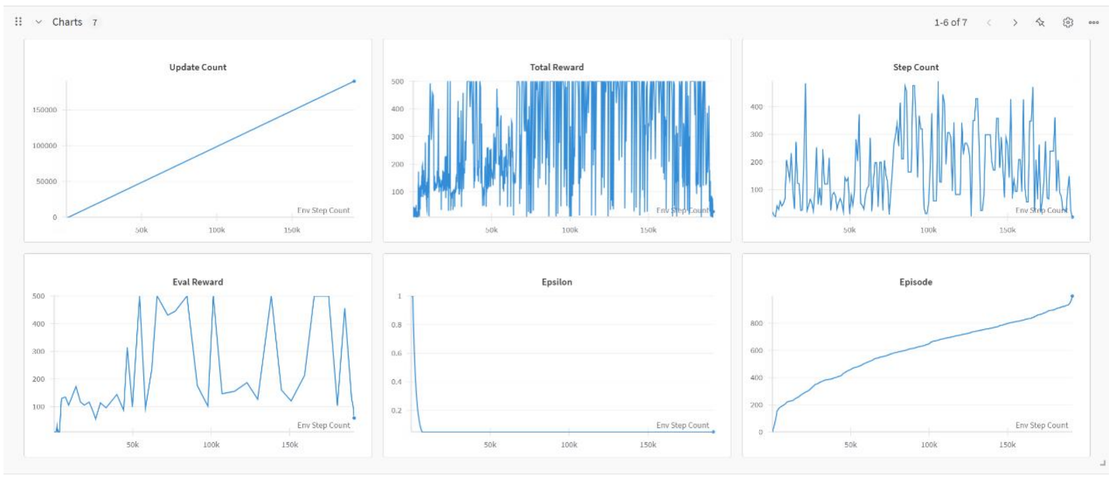
Graph 20. Training curve of task 2

3.3 Effect of each strategy on cartpole

In order to investigate the effect of each training strategy, a ablation study was conducted on cartpole game since the required training time is far less than that of ALE/pong.

3.3.1 DDQN on cartpole

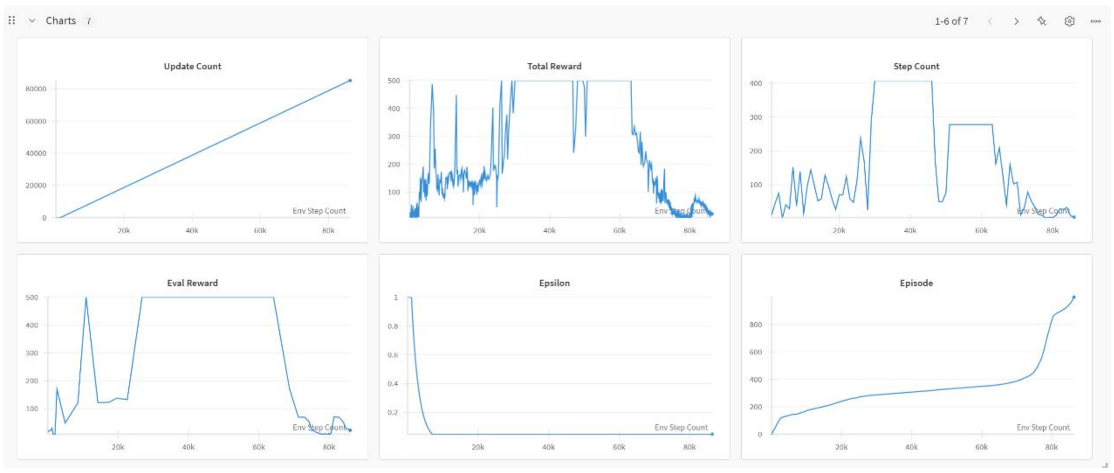
Graph 21 illustrates the training curve on cartpole with only DDQN. According to the data, the first highest score happened after approximately 50K environment steps. Although the number is close to the one of graph 18, the frequency of reaching highest score was higher.



Graph 21. Training curve on cartpole (DDQN only)

3.3.2 Multi-step return on cartpole

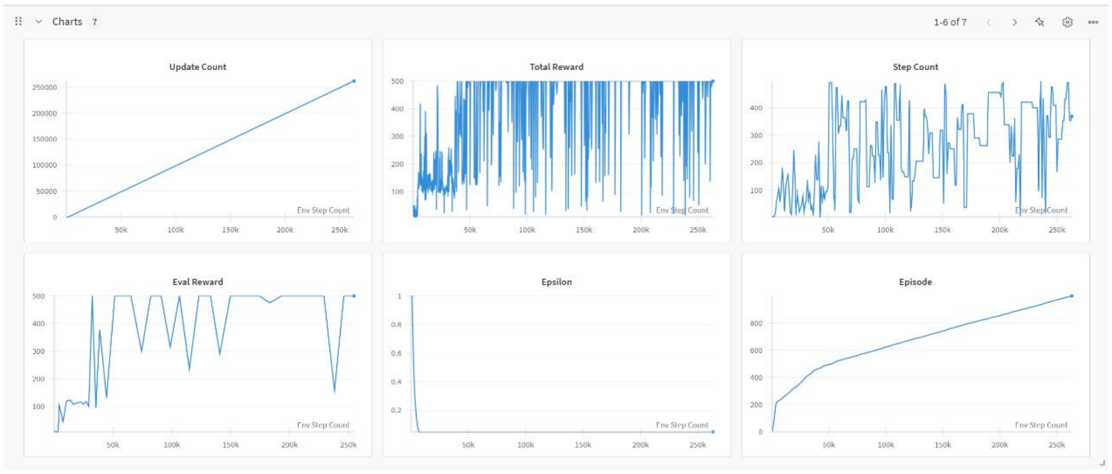
Graph 22 illustrates the training curve on cartpole with only DDQN. According to the picture, the first highest score was achieved at roughly 20K environment steps, which is faster than DDQN. Thereafter, the number remained stable at 500 scores for a while.



Graph 22. Training curve on cartpole (multi-step return only)

3.3.3 PER on cartpole

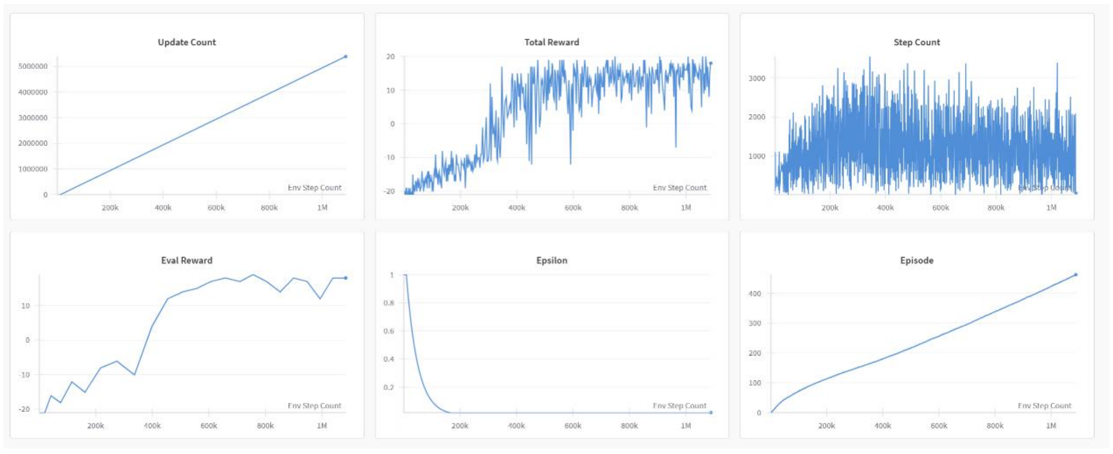
Graph 23 illustrates the training curve on cartpole with only DDQN. According to the data, the first highest score was reached at 50K environment steps, which is similar to DDQN. But the performance is much more stable than DDQN.



Graph 23. Training curve on cartpole (PER only)

3.4 Training curve of task 3

Graph 24 illustrates the training curve of task 3. According to the line chart, the DQN model enhanced significantly at roughly 300K, reaching first 19 scores at 150K. Thereafter, the performance remained steady until 1M environment steps, despite of some fluctuations.

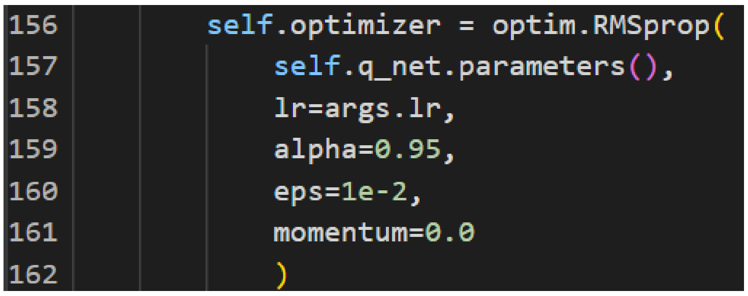


Graph 24. Training curve of task3 (ALE/pong)

4. Additional Analysis

4.1 RMSprop

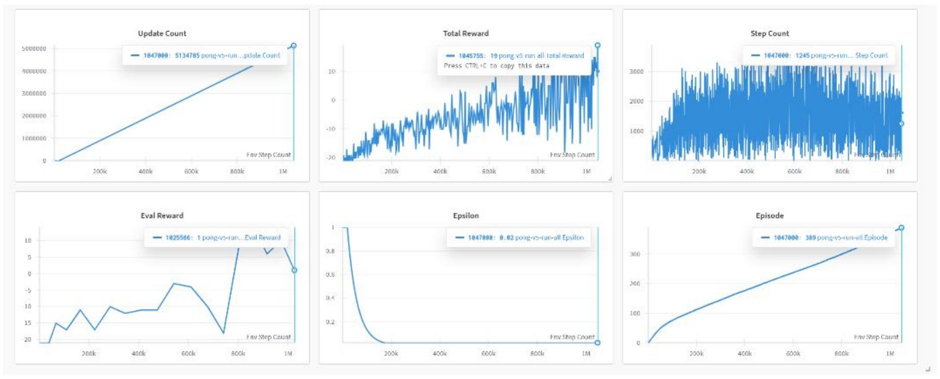
From the thesis of DeepMind, Playing Atari with Deep Reinforcement Learning (https://arxiv.org/abs/1312.5602). They used "RMSprop" ass their optimizer instead of "Adam" to deal with complicated loss. Graph 25 illustrates the detailed implementation. "alpha" represent the Smoothing constant of the optimizer, and it can Keeps a moving average of squared gradients. The parameter "eps" works like an offset to avoid divided by zero (default 0.01). The parameter "momentum" stands for Traditional momentum term applied after normalization (default 0.0)



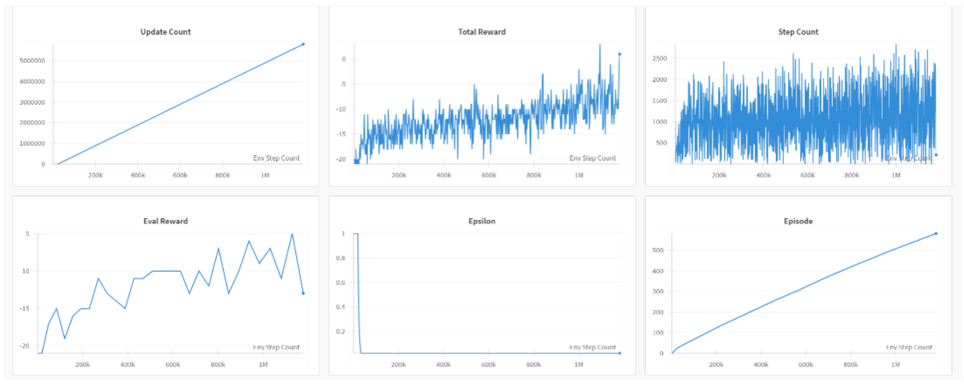
```
        self.optimizer = optim.RMSprop(
            self.q_net.parameters(),
            lr=args.lr,
            alpha=0.95,
            eps=1e-2,
            momentum=0.0
            )
```

Graph 25. Implementation of RMSprop

Graph 26 depicts the performance after adopting "RMSprop" as the optimizer. It reaches 19 scores at roughly 1M environment steps. In terms of "Adam" (graph 27), its number only outweigh 0 at 1M.



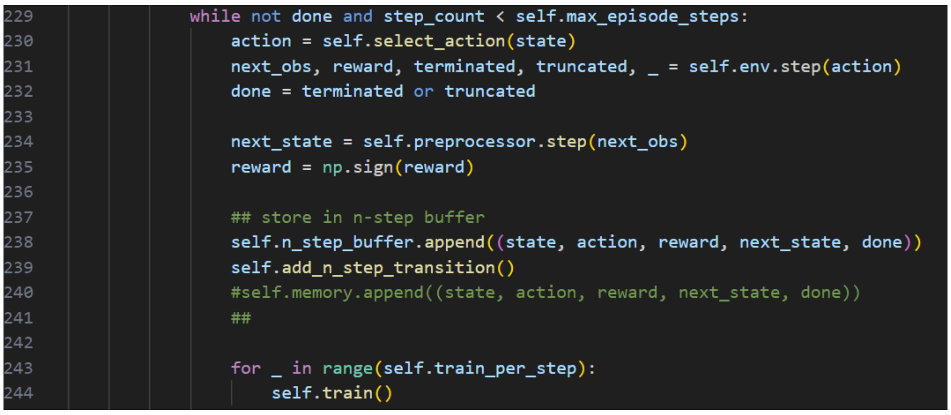
Graph 26. Training curve of RMSprop



Graph 27. Training curve of Adam

4.2 reward clipping

In the training process, the loss might become unstable since the scores in ALE/pong will fluctuated from -21 to 21, which make the training task very difficult. To combat the issue, a technique called reward clipping was introduced (line 235 in graph 28). Once calculate the sign value of reward in training, the reward will be restricted in the range of -1 and +1, and consequently makes training easier.

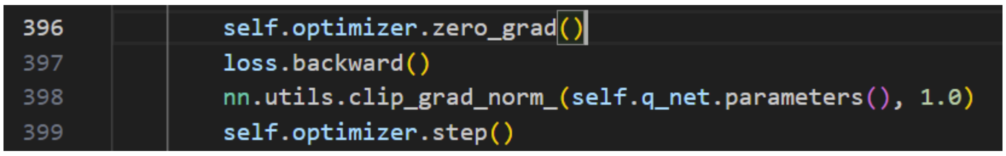


```
229            while not done and step_count < self.max_episode_steps:
230                action = self.select_action(state)
231                next_obs, reward, terminated, truncated, _ = self.env.step(action)
232                done = terminated or truncated
233
234                next_state = self.preprocessor.step(next_obs)
235                reward = np.sign(reward)
236
237                ## store in n-step buffer
238                self.n_step_buffer.append((state, action, reward, next_state, done))
239                self.add_n_step_transition()
240                #self.memory.append((state, action, reward, next_state, done))
241                ##
242
243                for _ in range(self.train_per_step):
244                    self.train()
```

Graph 28. Implement of reward clipping

4.3 gradient clipping

Another technique is called gradient clipping. Graph 29 (line 398) shows the detailed implementation. By doing so, the gradient of the model will be stabilized, ultimately makes training easier.

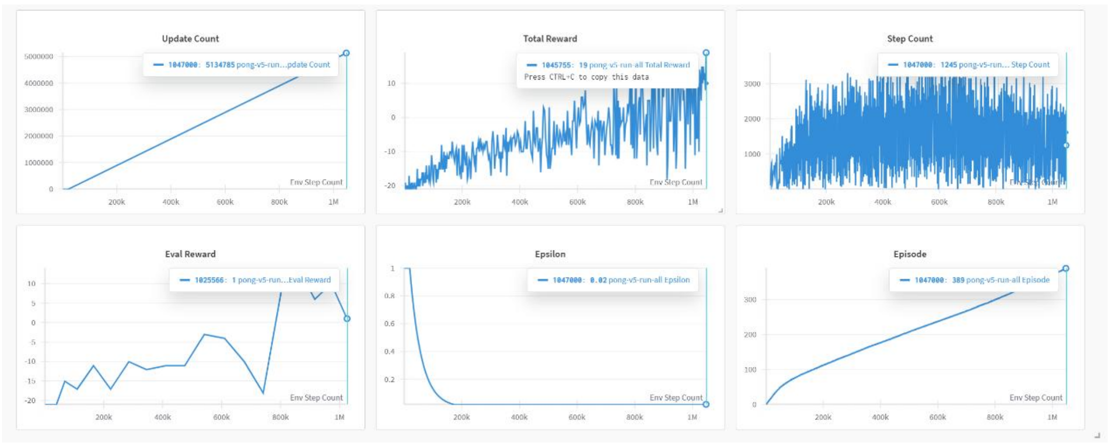


```
396        self.optimizer.zero_grad()
397        loss.backward()
398        nn.utils.clip_grad_norm_(self.q_net.parameters(), 1.0)
399        self.optimizer.step()
```

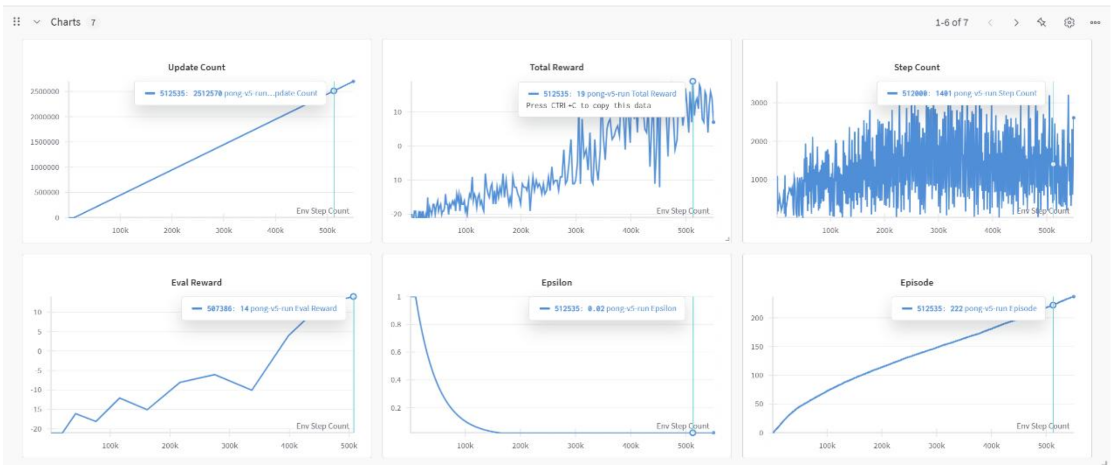
Graph 29. Implement of gradient clipping

4.4 result after adopting reward clipping and gradient clipping

Graph 30 and 31 illustrate the training curve before and after adopting reward clipping and gradient clipping. Overall, the policy with reward clipping and gradient clipping are more efficient than that without them. According to the data, although the performance of graph 30 is a little better than graph 31 in the range of 0 ~ 200k environment steps, the performance of the latter improves drastically in the consequent steps. Furthermore, the first 19 scores appear at approximately 1M and 500K in graph 30 and graph 31 respectively.



Graph 30. Training curve before adopting reward clipping and gradient clipping



Graph 31. Training curve after adopting reward clipping and gradient clipping