# Deep Learning Lab5 Report

陳以瑄 (ID: 313551047)

# 1 Introduction

In this lab, I implemented the vanilla DQN and three enhancement techniques: (1) Double DQN, (2) Prioritized Experience Replay buffer, and (3) multi-step return, to solve the CartPole and Pong Atari tasks. Although the vanilla DQN is sufficient to solve the CartPole problem, it requires a significant amount of time to solve the Pong problem, taking about 4 million environment steps. By applying the enhancement techniques, the Pong problem can be solved in just 550k environment steps. I also conducted an ablation study on the CartPole environment, demonstrating that each of the three methods individually improves performance.

# 2 Implementation details

## 2.1 Task1 - Vanilla DQN on CartPole

### 2.1.1 Neural Network Architecture

For Task 1, aimed at solving the CartPole task, I implemented a simple neural network architecture. The network consists of two hidden layers, each with a dimension of 32.

```
class DQN(nn.Module):
    def __init__(self,input_dim, hidden_dim, num_actions):
        super(DQN, self).__init__()
        self.network = nn.Sequential(
           nn.Linear(input_dim, hidden_dim),
           nn.ReLU(),
           nn.Linear(hidden_dim, hidden_dim),
           nn.ReLU(),
           nn.Linear(hidden_dim, num_actions)
        )
    def forward(self, x):
        return self.network(x)
```

### 2.1.2 Replay Buffer

Since this is a vanilla DQN, the replay buffer uses uniform random sampling. I store transitions in a deque and sample batches randomly.

```
self.memory = deque(maxlen=args.memory_size)

batch = random.sample(self.memory, self.batch_size)
states, actions, rewards, next_states, dones = zip(*batch)
```

### 2.1.3 Bellman Error for DQN

The DQN loss is defined as $\mathcal{L}_{\text{DQN}}(\theta) = \sum_{(s,a,r,s')\in D} \left(r + \gamma \max_{a'\in A} Q(s', a'; \bar{\theta}) - Q(s, a; \theta)\right)^2$ where $\theta$ are the parameters of the current Q-network, and $\bar{\theta}$ are the parameters of the target network.

**Line 2**: Predict the current Q-value $Q(s, a; \theta)$ using the Q-network.

**Line 4**: Obtain the maximum next Q-value $\max_{a'\in A} Q(s', a'; \bar{\theta})$ using the target network.

**Line 5**: Compute the target Q-value as $r + \gamma \max_{a'} Q(s', a'; \bar{\theta})$.

**Lines 6–7**: Calculate the Bellman error (loss) as the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values.

```
# After getting states, next_states, actions, rewards, dones
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    q_nexts = self.target_net(next_states).max(1)[0]
    q_targets = rewards + self.gamma* q_nexts*(1-dones)
criterion = nn.MSELoss()
loss = criterion(q_values, q_targets)

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

## 2.2 Task2 - Vanilla DQN on Atari Pong

### 2.2.1 Neural Network Architecture

Since Task 2 involves solving a more complex environment, I design a deeper neural network consisting of two parts: a convolutional feature extractor with three layers, and a fully connected classifier that maps the extracted features to action values.

```
class DQN(nn.Module):
    def __init__(self, num_actions):
        super(DQN, self).__init__()
        self.cnn = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4),
                                 nn.ReLU(True),
                                 nn.Conv2d(32, 64, kernel_size=4, stride=2),
                                 nn.ReLU(True),
                                 nn.Conv2d(64, 64, kernel_size=3, stride=1),
                                 nn.ReLU(True)
                                 )
```

```
        self.classifier = nn.Sequential(nn.Linear(7*7*64, 512),
                                        nn.ReLU(True),
                                        nn.Linear(512, num_actions)
                                        )

    def forward(self, x):
        x = x.float() / 255.
        x = self.cnn(x)
        x = torch.flatten(x, start_dim=1)
        x = self.classifier(x)
        return x
```

## 2.3 Task3 - Enhance DQN on Atari Pong

### 2.3.1 Modify DQN to Double DQN

The Double DQN loss is defined as:
$\mathcal{L}_{\text{DDQN}}(\theta) = \sum_{(s,a,r,s')\in D} \left(r + \gamma Q(s', argmax_{a'\in A} Q(s,a;\theta); \bar{\theta}) - Q(s,a;\theta)\right)^2$, where $\theta$ are the parameters of the current Q-network, and $\bar{\theta}$ are the parameters of the target network.
**Line 2:** Predict the current Q-value $Q(s,a;\theta)$ using the behavior net.
**Line 4:** Use the behavior network to select the next action $\max_{a'} Q(s',a';\theta)$.
**Line 5:** Evaluate the next Q-value $Q(s', arg\max_{a'} Q(s,a;\theta); \bar{\theta})$ using the target network to avoid overestimation bias.
**Line 6—7:** Compute the Bellman error (loss) as the Mean Squared Error (MSE) between the predicted Q-values and the target Q-values.

```
# After getting states, next_states, actions, rewards, dones
q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)
with torch.no_grad():
    next_actions = torch.argmax(self.q_net(next_states), dim = 1)
    q_nexts = self.target_net(next_states.squeeze(-1)).gather(1,
   next_actions.unsqueeze(1)).squeeze(1)
    q_targets = rewards+ self.gamma * q_nexts*(1-dones)

criterion = nn.MSELoss()
loss = criterion(q_values, q_targets)

self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

### 2.3.2 Implement the memory buffer for PER

1. Add
Use a First-In-First-Out (FIFO) approach to add transitions.

Assign priorities $p$ as $p = |\delta| + \epsilon$, and save $p^\alpha$ for efficient future computation.

```
def add(self, transition, error=1):
    if len(self.buffer)< self.capacity:
        self.buffer.append(None)
    self.buffer[self.pos] = transition
    self.priorities[self.pos] = (abs(error) + self.epsilon)** self.alpha
    self.pos = (self.pos+1)% self.capacity
```

In DQNAgent run(), after getting the (s,a,r,s',a'), insert the transition into the PER buffer with the highest priority.

```
if len(self.memory.buffer):
    prior = self.memory.priorities[: len(self.memory.buffer)].max()
    self.memory.add((state, action, reward, next_state, done), error=prior)
else:
    self.memory.add((state, action, reward, next_state, done), error = 1)
```

2. Sample

Sample a batch of transitions with the probability $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$. Then, calculate the importance sampling weight $w_i = \left(\frac{1}{NP(i)}\right)^\beta$ and return the indices, samples, and weights.

```
def sample(self, batch_size):
    if len(self.buffer) == self.capacity:
        priorities = self.priorities
    else:
        priorities = self.priorities[: self.pos]
    probs = priorities/priorities.sum()
    indices = np.random.choice(len(probs), batch_size, p = probs)
    samples = [self.buffer[idx] for idx in indices]

    weights = (len(self.buffer)* probs[indices])**(-self.beta)
    weights/= weights.max()
    return indices, samples, weights
```

In DQNAgent train(), sample a batch of transition.

```
indices, batch, weights = self.memory.sample(self.batch_size)
states, actions, rewards, next_states, dones = zip(*batch)
```

3. Update

```
def update_priorities(self, indices, errors):
    for idx, err in zip(indices, errors):
        self.priorities[idx] = (abs(err) + 1e-6) ** self.alpha
    return
```

In DQNAgent train(), after getting the new Bellman error, update the priorities.

```
BM_errors = (q_values - q_targets.squeeze()).detach().cpu().numpy()
self.memory.update_priorities(indices, BM_errors)
```

### 2.3.3 Modify the 1-step return to multi-step return

For each step, save the tuple $(s, a, r, s', d)$ into the nstep_buffer. Once the buffer reaches n-step, calculate the discounted cumulative n-step reward and save (s, a, n-step R, n-step s', n-step d) into the replay buffer.

```
self.nstep_buffer.append( (state, action, reward, next_state, done))
if len(self.nstep_buffer) == self.n_steps:
    R = 0.0
    s_n = next_state
    done_n = done
    for k, (_, _, r, s_next, d) in enumerate(reversed(self.nstep_buffer)):
        R = r + (self.gamma * R * (1 - d))
        if d:
            s_n = s_next
            done_n = True
            break
    s_0, a_0, _, _, _ = self.nstep_buffer[0]
    self.memory.append((s_0, a_0, R, s_n, done_n))
```

Since it is a N-step target, when we compute the q_target, we should discount by $\gamma^N$

```
q_targets = rewards+ (self.gamma ** self.n_steps) * q_nexts*(1-dones)
```

## 2.4 Explain how you use Weight & Bias to track the model performance

I just followed the code given in the Sample Code and added the log in the DQNAgent's train() to log the training loss.

```
wandb.log({
    "Train Loss": loss.item(),
    "Q Mean": q_values.mean().item(),
    "Q Std": q_values.std().item()
})
```

# 3 Analysis & Discussion

## 3.1 Training Curve

### 3.1.1 Task 1

Hyperparameter: lr = 0.00019; batch size = 128; replay-start-size = 500
Result: Converge in 80k env steps.


### 3.1.2 Task 2

hyperparameter: lr = 0.0001; batch size = 32; replay-start-size = 50000
Result: Reach 500 eval reward in 4M env step


### 3.1.3 Task 3

hyperparameter: n-step = 4; lr = 0.00025; batch size = 32; replay-start-size = 30000
Result: Reach 500 eval reward around 550k env step

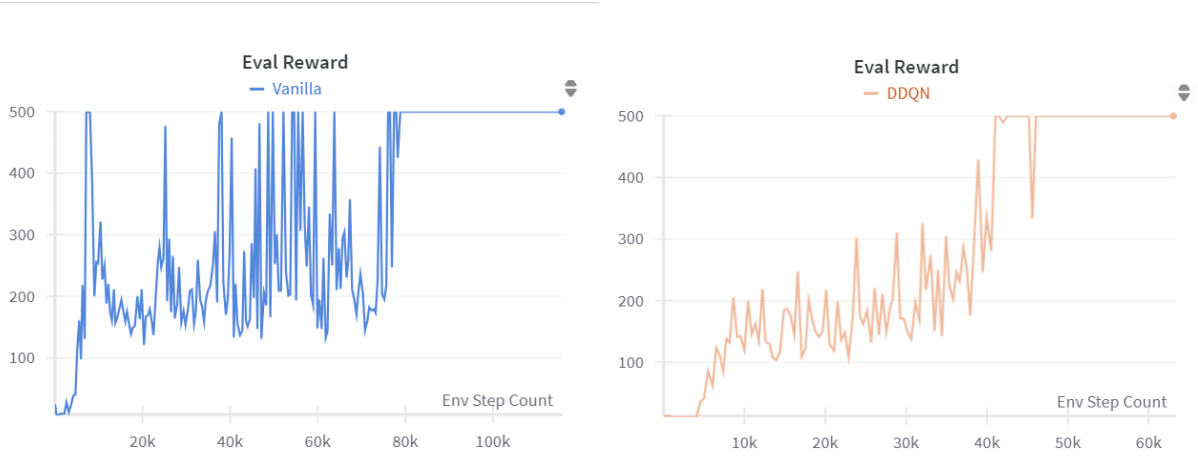
## 3.2 Comparison

### 3.2.1 Compare the Result over Task 2 and 3

From the plot, we can easily observe that applying the three enhancement techniques significantly improves overall performance. The original version required 4 million environment steps to reach the goal, whereas the enhanced version only took 550k environment steps.


### 3.2.2 Compare the Result between DQN and DDQN on CartPole

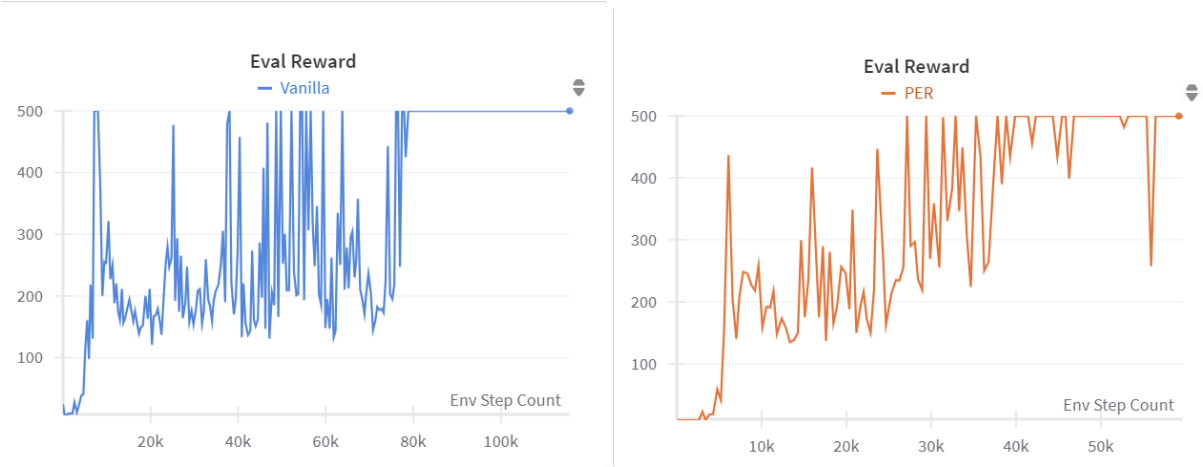
We can observe that the DDQN version exhibits more stable reward values, as it avoids overestimating the Q-values. As a result, the DDQN version converges faster (50k $<$ 80k environment steps).




### 3.2.3 Compare the Result of PER on CartPole

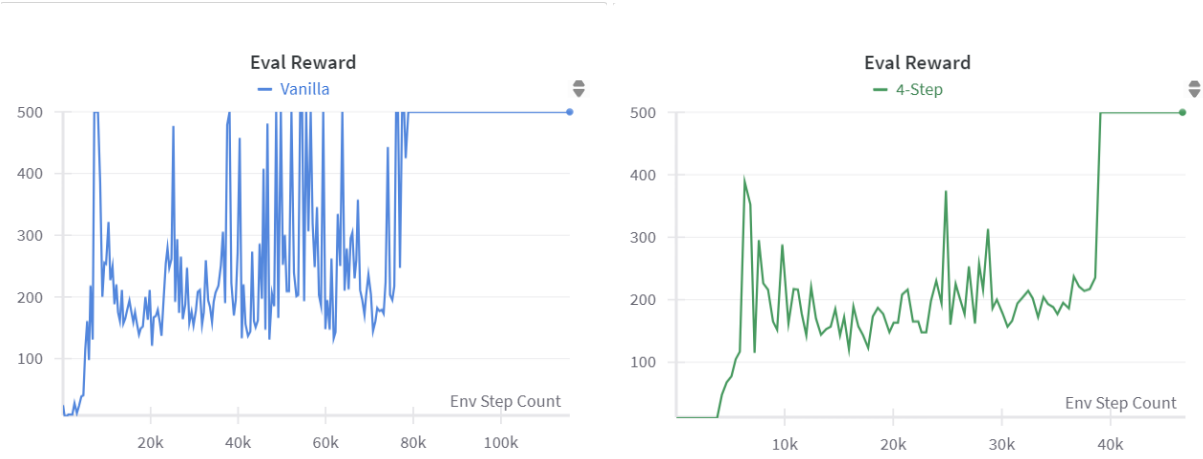
We can observe that, since the PER buffer prioritizes transitions with larger Bellman errors—indicating they require more learning—the version with PER converges faster

(40k < 80k environment steps).




### 3.2.4 Compare the Result of NStep Return on CartPole

We can observe that the version using 4-step returns exhibits more stable reward values, resulting in a steadier increase and faster convergence (50k < 80k environment steps).




# 4 Other training strategy

## 4.1 RMSProp Optimizer

Since the environment is noisy, I applied the RMSProp optimizer instead of Adam to achieve more stable convergence.

## 4.2 Reward Clipping

Since the reward in Pong ranges from -21 to 21, which introduces significant variability, I clip the reward to between ±1 to indicate to the agent whether it wins or loses, making the learning process more stable.

## 4.3 Optuna Hyperparameter Tunning

Since the learning rate is critical, a small learning rate can lead to slower convergence, while a large learning rate may cause overshooting. Therefore, I use Optuna to search for the optimal learning rate within the range of [1e-5,1e-3].