Deep Learning Lab5

110550085 房天越

1. Introduction

In this homework, we aim to solve the Cartpole challenge and Atari challenge in gymnasium provided by OpenAI. We apply Vanilla DQN to solve the two tasks. Also, for the Atari task, we also explore several enhancements to the DQN algorithm, including Double DQN, Prioritized Experience Replay, and multistep return.

For results, we found that it is easy to solve the cartpole challenge, and could get an average score of 500 with Vanilla DQN. However, for the Atari task, the Vanilla DQN takes 4M steps, and could still not get an average score of 19. For the enhancement task, our methods yield a better result, and have the model converged much more quickly than the Vanilla DQN.

2. Your Implementation

I. Task1-Cartpole

For this task, we use a simple fully connected neural network to approximate the Q function, and use an epsilon greedy policy for action selection. Also, we use experience replay with uniform sampling and a target network.

```
self.network = nn.Sequential(
    nn.Linear(4, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, num_actions)
)
```

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

For the training process, we use MSE loss as the loss function and applied the above techniques.

```
# Decay function for epsilin-greedy exploration
if self.epsilon > self.epsilon_min:
    self.epsilon *= self.epsilon_decay
self.train_count += 1
```

```
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
loss = nn.MSELoss()(q_values, target_q_values)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

II. Task2-Atari with Vanilla DQN

For this task, we use a convolutional neural network (CNN) instead of the simple fully connected neural network.

Also, we tried to use linear epsilon decay and two-stage epsilon greedy approach to solve it.

```
self.network = nn.Sequential(
    nn.Conv2d(4, 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)
)
```

```
# Decay function for epsilin-greedy exploration
step =  self.env_count
if step <= self.decay_steps1:
    frac = step / self.decay_steps1
    # 從 start 線性降到 mid
    self.epsilon = self.epsilon_start + frac * (self.epsilon_mid - self.epsilon_start)
elif step <= self.decay_steps1 + self.decay_steps2:
    frac = (step - self.decay_steps1) / self.decay_steps2
    # 從 mid 線性降到 end
    self.epsilon = self.epsilon_mid + frac * (self.epsilon_end - self.epsilon_mid)
else:
    # 保持最終值
    self.epsilon = self.epsilon_end
```

III. Task3-Atari with Enhancement of DQN

In this task, we use enhancement techniques for the DQN algorithm to accelerate the convergence and yield a better result.

The techniques include Double DQN, Prioritized Experience Replay and Multi-Step Return, let's discuss each of them:

A. Double DQN

We pick the best next action with the online network (q_net) but evaluate its value with a separate target network (target_net).

```
best_action = self.q_net(next_states).argmax(1, keepdim=True)
next_q_values = self.target_net(next_states).gather(1, best_action).squeeze()

target_q_values = rewards + (1 - dones) * (self.gamma ** self.n_step) * next_q_values
```

B. Prioritized Experience Replay

We weight each stored transition by its absolute TD-error (plus ε) raised to α. High-error samples get replayed more often. We then

correct the bias with importance-sampling weights raised to –β.

```
if error is not None:
    p = (abs(error) + 1e-5) ** self.alpha
else:
    p = self.priorities.max() if len(self.buffer)>0 else 1.0
```

```
n = len(self.buffer)
probs = self.priorities[:n]
P = probs / (probs.sum() + 1e-8)


indices = np.random.choice(n, batch_size, p=P)
transitions = [self.buffer[i] for i in indices]

weights = (n * P[indices]) ** (-self.beta)
weights /= weights.max()
```

C. Multi-Step Return

We accumulate the next n rewards, discounted by γ^i, then store a single "n-step transition", so learning sees a longer-horizon update and can propagate reward signals faster.

```
def store(self, transition):
    self.n_step_buffer.append(transition)
    if len(self.n_step_buffer) < self.n_step:
        return
    cumulative_reward = 0
    next_state = None
    done = False
    for idx, (s, a, r, ns, d) in enumerate(self.n_step_buffer):
        cumulative_reward += (self.gamma ** idx) * r
        next_state = ns
        done = done or d
    s0, a0, _, _, _ = self.n_step_buffer[0]
    self.memory.add((s0, a0, cumulative_reward, next_state, done))
```

```
target_q_values = rewards + (1 - dones) * (self.gamma ** self.n_step) * next_q_values
```

IV. Questions on the Spec (Some already discussed in Task3)

A. How do you obtain the Bellman Error for DQN?

During the training process, sample a batch from the replay buffer, use the net Q(s, a) to predict the target value

$y = r + (1\text{-done})\ \gamma \max_{a'} Q_{target}(s',a')$.

The Bellman error is y – Q(s, a), and the loss is its square.

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

We use wandb.log to track parameters like the current loss, average Q value, and epsilon. In the evaluation part, we also record the reward, eval loss, and steps.

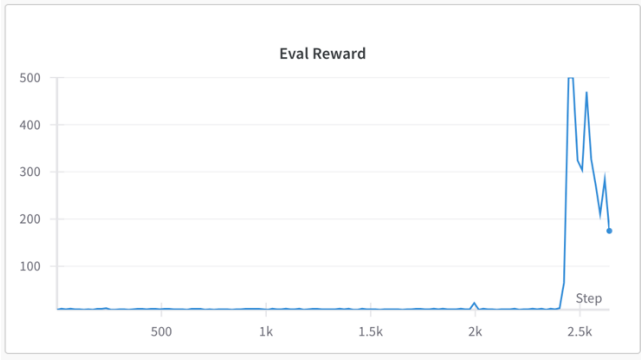
The tracked parameters are drawn as curves, we can also compare

different runs to adjust and debug.

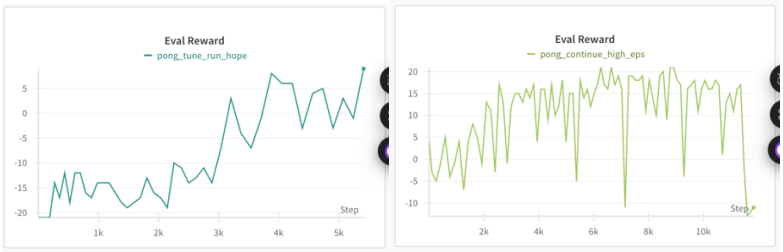
3. Analysis and Discussions

I. Plot the training curves

A. Task1



B. Task2



Two charts are shown here because I got broken pipe during the first training, and I continued training from the previous best record in the second chart.

C. Task3

The eval score reached 19 at step about 1.2M.

II. Analyze the sample efficiency with and without DQN enhancements

Despite the fact that we still need more than 1M steps for it to converge, we can still observe that the sample efficiency with DQN is much better than that with Vanilla DQN. The steps required in the fully enhanced one

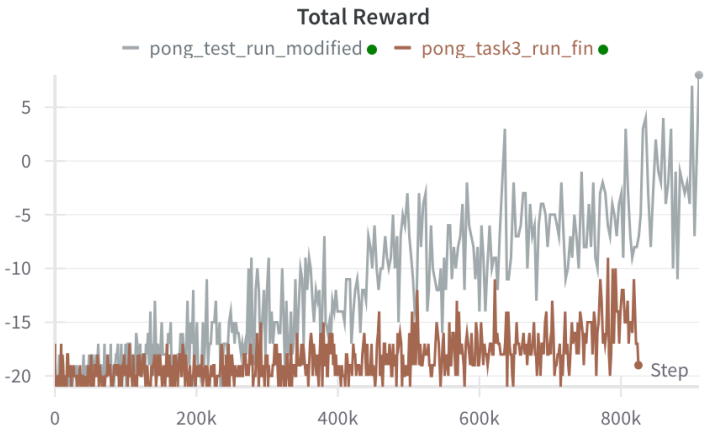
is less than half of the vanilla one.

Due to the limit of time and resource, I was only able to do the comparison of Fully Enhanced version and the Vanilla version. However, we can still observe the significant difference between them.

III. Additional analysis on other training strategies.

I tried to use different epsilon values and decay steps.

The best result happens by using 500000 steps for epsilon from 1.0 to 0.1, and 500000 steps for epsilon from 0.1 to 0.01, and keep using 0.01 for step > 1M. Here we can see a clear comparison.



The gray one is epsilon from 1.0 to 0.1 in 500000 steps, and the brown one is epsilon from 1.0 to 0.1 in 1M steps. While in the original paper, they used 1.0 to 0.1 in 1000000 steps, we can observe that the performance of it is significantly worse than my best approach.