Deep Learning Homework 5 Report

Cheng-Liang Chi

April 30, 2025

Contents

1 Introduction

In this report, we will discuss the implementation of Deep Q-Learning (DQN) and its variants, including Double DQN [1], Prioritized Experience Replay [2], and Multi-Step Learning [3]. We will also analyze the performance of these algorithms on both the `CartPole-v1` and `ALE/Pong-v5` environments. The goal is to understand the impact of these techniques on the learning process and the final performance of the agent.

2 Implementation Details

2.1 Refactoring the Sample Code

First, since the sample code is not modular, we refactored the code into multiple classes. The main classes are:

- **DQN**: This class implements the neural network architecture. It is a subclass of the `torch.nn.Module` class.
- **DQNAgent**: This class represents the agent that interacts with the environment. It contains methods for selecting actions, storing experiences, and updating the Q-values. The agent is responsible for the exploration and exploitation of the environment. It also contains the experience replay buffer, which stores the agent's experiences for training. However, the backend neural network is modularized into a separate class. That is we can reuse the same agent class for different neural networks dealing with different environments we focused on.
- **Trainer**: This class is responsible for training the agent. It handles the training loop, including collecting experiences, updating the model, and logging results. By the way, the trainer class is also responsible for loading the environment and the agent. It will handle agent's actions and interact it with the environment.
- **Tester**: This class is responsible for testing the agent. It handles the testing loop, it is similar to the trainer class, but it does not update the model. The tester class is also responsible for loading the environment and the agent. It will handle agent's actions and interact it with the environment.

With this refactoring, I can easily add new features and techniques to the agent, trainer, and tester classes without modifying the core logic of the DQN algorithm. This modularity also allows us to reuse the same code for different tasks and environments, making it easier to experiment with different architectures and techniques.

2.1.1 DQN Class

The DQN class is a subclass of the `torch.nn.Module` class. It implements the neural network architecture for the DQN algorithm.

For the `CartPole-v1` environment, I used a simple feedforward neural network with two hidden layers. The input layer has 4 neurons (one for each state variable), and the output layer has 2 neurons (one for each action). Both hidden layers have 128 neurons, and the activation function is ReLU. The architecture is designed to be simple and efficient, as the `CartPole-v1` environment has a relatively low-dimensional state space. The output layer uses a linear activation function, which is suitable for the DQN algorithm since it outputs Q-values for each action.

The implementation of the `CartPoleDQN` class is as follows:

```python
class CartPoleDQN(nn.Module):
    def __init__(self, input_state, num_actions):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(input_state, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, num_actions),
        )

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

For the `ALE/Pong-v5` environment, I used a convolutional neural network (CNN) with three convolutional layers and two fully connected layers. The architecture is designed to effectively capture the spatial features of the input frames, allowing the agent to learn optimal policies for playing the game.

The implementation of the `PongDQN` class is as follows:

```python
class PongDQN(nn.Module):
    def __init__(self, input_channels, num_actions):
        super().__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: torch.Tensor):
        x = x.squeeze(-1)
        return self.network(x / 255.0)
```

2.1.2 DQNAgent Class

The DQNAgent class is responsible for interacting with the environment and managing the agent's experiences. It contains methods for selecting actions, storing experiences, and updating the Q-values.

The agent uses an ϵ-greedy policy for action selection, where it explores the environment with a probability of ϵ and exploits the learned Q-values with a probability of 1 - ϵ. The agent also maintains a replay buffer, which stores the agent's experiences for training. The replay buffer is implemented using the the `PrioritizedReplayBuffer` class, which will be discussed in Section 2.4.

The implementation of the `DQNAgent` class is as follows:

```python
class DQNAgent:
    def __init__(self, env, args):
        self.device = args.device
        logger.info(f"Using device: {self.device}")

        match env.spec.id:
            case "CartPole-v1":
                self.input_state = 4
                self.num_actions = 2
                self.DQN = CartPoleDQN
            case "ALE/Pong-v5":
                self.input_state = 4
                self.num_actions = 6
                self.DQN = PongDQN
            case _:
                raise ValueError(f"Unsupported environment: {env}")

        self.q_net = self.DQN(self.input_state, self.num_actions).to(self.device)
        self.q_net.apply(init_weights)

        self.save_dir = args.save_dir
        os.makedirs(self.save_dir, exist_ok=True)

    def train(self, args):
        self.batch_size = args.batch_size
        self.update_period = args.update_period
        self.gamma = args.discount_factor
        self.memory = PrioritizedReplayBuffer(
            args.memory_size, args.per_alpha, args.per_beta
        )

        self.vanilla = args.vanilla
        self.target_net = self.DQN(self.input_state, self.num_actions).to(self.device)
        self.target_net.load_state_dict(self.q_net.state_dict())
        self.target_net.eval()
        self.optimizer = optim.Adam(self.q_net.parameters(), lr=args.lr)

        self.learn_count = 0

    def select_action(self, state, epsilon):
```

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

    def learn(self) -> float:
        if len(self.memory) < self.batch_size:
            return float("-inf")

        batch, indices, weights = self.memory.sample(self.batch_size)
        indices = torch.tensor(indices, dtype=torch.int64).to(self.device)
        weights = torch.tensor(weights, dtype=torch.float32).to(self.device)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.from_numpy(np.array(states).astype(np.float32)).to(self.device)
        next_states = torch.from_numpy(np.array(next_states).astype(np.float32)).to(
            self.device
        )
        actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
        rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
        dones = torch.tensor(dones, dtype=torch.float32).to(self.device)

        with torch.no_grad():
            if self.vanilla:
                target_q_values = (
                    rewards
                    + (1 - dones) * self.gamma * self.target_net(next_states).max(1)[0]
                )
            else:
                next_q_values = self.target_net(next_states).max(1)[0]
                target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
        q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

        td_errors = target_q_values - q_values
        loss = (td_errors**2 * weights).mean()
        # loss = nn.MSELoss()(q_values, target_q_values)
```

```python
        self.optimizer.zero_grad()
        loss.backward()
        # torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), 1.0)
        self.optimizer.step()

        self.memory.update_priorities(indices, td_errors.detach().cpu().numpy())

        # self.memory.beta = min(1.0, self.memory.beta + 0.000001)

        self.learn_count += 1
        if self.learn_count % self.update_period == 0:
            self.target_net.load_state_dict(self.q_net.state_dict())
            logger.debug(
                f"Target network updated at learn_count={self.learn_count/1000:.2f}k"
            )
            logger.debug(
                f"Memory size: {len(self.memory)}, beta: {self.memory.beta:.2f}"
            )
        return loss.item()
```

2.1.3 Trainer Class

The Trainer class is responsible for training the agent. It handles the training loop, including collecting experiences, updating the model, and logging results. The trainer class is also responsible for loading the environment and the agent. It will handle agent's actions and interact it with the environment. The implementation of the `Trainer` class is as follows:

```python
class Trainer:
    def __init__(self, args) -> None:
        self.save_dir = args.save_dir
        self.env = gym.make(args.env_name, render_mode="rgb_array")
        if args.env == "pong":
            self.env = AtariPreprocessing(
                self.env,
                frame_skip=1,
                grayscale_newaxis=True,
                screen_size=84,
                grayscale_obs=True,
                noop_max=30,
            )
            self.env = FrameStackObservation(self.env, 4)
        self.env.action_space.seed(args.seed)
        self.env.observation_space.seed(args.seed)
        self.env.reset(seed=args.seed)

        self.num_actions = self.env.action_space.n  # type: ignore
```

```python
        logger.info(f"Environment: {self.env.spec.id}")  # type: ignore
        logger.info(f"Action Space: {self.env.action_space}")
        logger.info(f"Observation Space: {self.env.observation_space}")

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        logger.info(f"Using device: {self.device}")

        self.agent = DQNAgent(self.env, args=args)
        self.agent.train(args)
        self.preprocessor = DummyPreprocessor()

        self.epsilon = args.epsilon_start
        self.epsilon_decay = args.epsilon_decay
        self.epsilon_min = args.epsilon_min

        self.episode = 0
        self.env_step = 0
        self.best_reward = 0 if self.env == "cartpole" else -21

        self.learn_per_step = args.learn_per_step

        self.eval_episodes = args.eval_episodes

    def run(self, episodes=1000):
        with Progress(
            SpinnerColumn(),
            *Progress.get_default_columns(),
            TimeElapsedColumn(),
            MofNCompleteColumn(),
            console=console,
        ) as progress:
            task = progress.add_task("[cyan]Training...", total=episodes)
            eval_task = progress.add_task(
                "[cyan]Episode 0: Evaluating...",
                total=self.eval_episodes,
            )
            for ep in range(episodes):
                progress.update(task, description=f"[cyan]Episode {ep}: Training...")
                self.episode = ep
                if self.env_step > 20e6:
                    logger.info(f"Reached 20M steps, stopping training.")
                    break
                if (ep + 1) % 20 == 0:
                    logger.info(
                        f"Episode {self.episode}: Environment Step: {self.env_step/1000:.2f}k, epsilon: {self.epsilon:.4f}"
                    )
```

```python

                self.train()

                if (ep + 1) % (episodes // 20) == 0:
                    model_path = os.path.join(self.save_dir, f"model_ep{ep}.pt")
                    torch.save(self.agent.q_net.state_dict(), model_path)
                    logger.info(f"Saved model checkpoint to {model_path}")

                if (ep + 1) % (episodes // 50) == 0:
                    eval_rewards = []
                    progress.reset(eval_task)
                    progress.update(
                        eval_task, description=f"[cyan]Episode {ep}: Evaluating..."
                    )
                    for _ in range(self.eval_episodes):
                        eval_rewards.append(self.evaluate())
                        progress.update(eval_task, advance=1)
                    eval_reward = sum(eval_rewards) / len(eval_rewards)
                    max_eval_reward = max(eval_rewards)
                    logger.info(
                        f"Episode {ep} - Eval Reward: {eval_reward:.2f} {eval_rewards}"
                    )

                    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.agent.q_net.state_dict(), model_path)
                        logger.info(
                            f"Saved new best model to {model_path} with reward {eval_reward}"
                        )
                    wandb.log(
                        {
                            "Episode": ep,
                            "Env Step Count": self.env_step,
                            "Eval Reward": eval_reward,
                            "Max Eval Reward": max_eval_reward,
                        }
                    )
                progress.update(task, advance=1)

    def train(self):
        avg_loss = 0

        obs, _ = self.env.reset(seed=random.randint(0, 10000))
```

```python
        state = self.preprocessor.reset(obs)

        done = False
        total_reward = 0

        while not done:
            action = self.agent.select_action(state, self.epsilon)
            next_obs, reward, terminated, truncated, _ = self.env.step(action)
            next_state = self.preprocessor.step(next_obs)
            done = terminated or truncated

            self.agent.memory.add((state, action, reward, next_state, done), 1)
            state = next_state
            total_reward += float(reward)
            self.env_step += 1

            for _ in range(self.learn_per_step):
                loss = self.agent.learn()
                if loss != float("-inf") and self.epsilon > self.epsilon_min:
                    self.epsilon *= self.epsilon_decay
                if loss != float("-inf"):
                    avg_loss += loss
        avg_loss /= self.learn_per_step
        wandb.log(
            {
                "Episode": self.episode,
                "Env Step Count": self.env_step,
                "Total Reward": total_reward,
                "Epsilon": self.epsilon,
                "Loss": avg_loss,
            }
        )

    def evaluate(self):
        obs, _ = self.env.reset(seed=random.randint(0, 10000))
        state = self.preprocessor.reset(obs)
        frames = [self.env.render()]

        done = False
        total_reward = 0

        while not done:
            action = self.agent.select_action(state, 0.0)
            next_obs, reward, terminated, truncated, _ = self.env.step(action)
            state = self.preprocessor.step(next_obs)
            done = terminated or truncated
```

```python
            total_reward += float(reward)
            frames.append(self.env.render())

        return total_reward
```

2.1.4 Tester Class

The Tester class is responsible for testing the agent. It handles the testing loop, it is similar to the trainer class, but it does not update the model. The tester class is also responsible for loading the environment and the agent. It will handle agent's actions and interact it with the environment. The implementation of the `Tester` class is as follows:

```python
class Tester:
    def __init__(self, args) -> None:
        self.save_dir = args.save_dir
        self.env = gym.make(args.env_name, render_mode="rgb_array")
        if args.env == "pong":
            self.env = AtariPreprocessing(
                self.env,
                frame_skip=1,
                grayscale_newaxis=True,
                screen_size=84,
                grayscale_obs=True,
                noop_max=30,
            )
            self.env = FrameStackObservation(self.env, 4)

        self.num_actions = self.env.action_space.n  # type: ignore

        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        logger.info(f"Using device: {self.device}")

        self.agent = DQNAgent(self.env, args=args)
        self.agent.q_net.load_state_dict(
            torch.load(args.model_path, map_location=self.device)
        )
        self.agent.q_net.to(self.device)
        self.agent.q_net.eval()

        self.preprocessor = DummyPreprocessor()

        self.episode = 0
        self.best_reward = 0 if self.env == "cartpole" else -21

        self.visualize = args.visualize
        self.save_dir = args.save_dir
        self.seed = args.seed
        gif_dir = os.path.join(self.save_dir, "gifs")
```

```python
        os.makedirs(gif_dir, exist_ok=True)

    def run(self, episodes):
        with Progress(
            SpinnerColumn(),
            *Progress.get_default_columns(),
            TimeElapsedColumn(),
            MofNCompleteColumn(),
            console=console,
        ) as progress:
            task = progress.add_task("[cyan]Testing...", total=episodes)
            total_reward = 0
            for ep in range(episodes):
                self.episode = ep
                eval_reward = self.evaluate(seed=self.seed + ep)
                logger.info(f"Episode {ep} - Test Reward: {eval_reward:.2f}")
                total_reward += eval_reward
                wandb.log(
                    {
                        "Episode": ep,
                        "Test Reward": eval_reward,
                    }
                )
                progress.update(task, advance=1)
            total_reward /= episodes
            logger.info(f"Average Test Reward: {total_reward}")

    def evaluate(self, seed):
        obs, _ = self.env.reset(seed=seed)
        state = self.preprocessor.reset(obs)
        frames = [self.env.render()]

        done = False
        total_reward = 0

        while not done:
            action = self.agent.select_action(state, 0.0)
            next_obs, reward, terminated, truncated, _ = self.env.step(action)
            done = terminated or truncated
            total_reward += float(reward)
            state = self.preprocessor.step(next_obs)
            frames.append(self.env.render())

        if self.visualize:
            gif_path = os.path.join(self.save_dir, "gifs", f"test_{self.episode}.gif")
            imageio.mimsave(gif_path, frames, fps=30)  # type: ignore
            logger.info(f"Saved test episode frames to {gif_path}")
```

```python
        return total_reward
```

2.2 Hyperparameters

The hyperparameters used in the training process are crucial for the performance of the DQN algorithm.

For Tasks 1 and 2, we are required to use vanilla DQN without any modifications. To meet this requirement, certain hyperparameters must be set accordingly. For example, setting $\alpha = 0$ in the `PrioritizedReplayBuffer` makes the buffer behave like a uniform replay buffer. Additionally, setting `update_period = 1` in the DQNAgent class ensures that the model is updated every time a batch is sampled from the replay buffer, aligning with the standard behavior of vanilla DQN. And last, setting `n_steps=1` in the DQNAgent class ensures that the agent uses one-step returns for training, which is standard in vanilla DQN.

2.3 Bellman Equation

The Bellman equation is a fundamental concept in reinforcement learning that describes the relationship between the value of a state and the values of its successor states. In the context of DQN, the Bellman equation is used to update the Q-values based on the agent's experiences. The Bellman equation for Q-learning is given by:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a') \tag{1}$$

where:

- $Q(s, a)$ is the Q-value for state s and action a.
- r is the reward received after taking action a in state s.
- γ is the discount factor, which determines the importance of future rewards.
- s' is the next state after taking action a in state s.
- $\max_{a'} Q(s', a')$ is the maximum Q-value for the next state s' over all possible actions a'.

The DQN algorithm uses a neural network to approximate the Q-values, and the Bellman equation is used to update the weights of the network during training. The Q-value update is performed using the following loss function:

$$L(\theta) = \mathbb{E}_{(s,a,r,s') \sim D}\left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{2}$$

where:

- $L(\theta)$ is the loss function.
- θ are the weights of the Q-network.
- θ^- are the weights of the target network, which are updated periodically.

- D is the replay buffer containing the agent's experiences.
- $\mathbb{E}$ is the expectation operator, which averages over the experiences in the replay buffer.

The loss function measures the difference between the predicted Q-value and the target Q-value. The weights of the Q-network are updated using gradient descent to minimize this loss. The target Q-value is computed using the Bellman equation, and the target network is used to stabilize the training process. The target network is a separate copy of the Q-network, updated periodically every C steps (a hyperparameter) by copying the weights of the Q-network. This periodic update reduces the variance of the Q-value updates, making the training process more stable and efficient.

The implementation of the Bellman equation in the DQN algorithm is done in the `learn` method of the `DQNAgent` class. In this method, the Q-values are updated based on the agent's experiences, ensuring that the learning process aligns with the principles of reinforcement learning. The implementation of the `learn` method is as follows:

```python
    def learn(self) -> float:
        if len(self.memory) < self.batch_size:
            return float("-inf")

        batch, indices, weights = self.memory.sample(self.batch_size)
        indices = torch.tensor(indices, dtype=torch.int64).to(self.device)
        weights = torch.tensor(weights, dtype=torch.float32).to(self.device)
        states, actions, rewards, next_states, dones = zip(*batch)

        states = torch.from_numpy(np.array(states).astype(np.float32)).to(self.device)
        next_states = torch.from_numpy(np.array(next_states).astype(np.float32)).to(
            self.device
        )
        actions = torch.tensor(actions, dtype=torch.int64).to(self.device)
        rewards = torch.tensor(rewards, dtype=torch.float32).to(self.device)
        dones = torch.tensor(dones, dtype=torch.float32).to(self.device)

        with torch.no_grad():
            if self.vanilla:
                target_q_values = (
                    rewards
                    + (1 - dones) * self.gamma * self.target_net(next_states).max(1)[0]
                )
            else:
                next_q_values = self.target_net(next_states).max(1)[0]
                target_q_values = rewards + (1 - dones) * self.gamma * next_q_values
        q_values = self.q_net(states).gather(1, actions.unsqueeze(1)).squeeze(1)

```

```
        td_errors = target_q_values - q_values
        loss = (td_errors**2 * weights).mean()
        # loss = nn.MSELoss()(q_values, target_q_values)
        self.optimizer.zero_grad()
        loss.backward()
        # torch.nn.utils.clip_grad_norm_(self.q_net.parameters(), 1.0)
        self.optimizer.step()

        self.memory.update_priorities(indices, td_errors.detach().cpu().numpy())

        # self.memory.beta = min(1.0, self.memory.beta + 0.000001)

        self.learn_count += 1
        if self.learn_count % self.update_period == 0:
            self.target_net.load_state_dict(self.q_net.state_dict())
            logger.debug(
                f"Target network updated at learn_count={self.learn_count/1000:.2f}k"
            )
            logger.debug(
                f"Memory size: {len(self.memory)}, beta: {self.memory.beta:.2f}"
            )
        return loss.item()
```

2.4 Prioritized Experience Replay

2.5 Multi-Step Reward

The multi-step reward is a technique used in reinforcement learning to improve the efficiency of learning by considering multiple steps of experience at once. In the context of DQN, multi-step rewards are used to update the Q-values based on a sequence of actions and rewards, rather than just the immediate reward. This approach allows the agent to learn from longer-term dependencies and can lead to faster convergence and better performance. The multi-step reward is computed by summing the rewards over a sequence of n steps, discounted by the discount factor γ:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} \tag{3}$$

where:

- R_t is the multi-step reward at time step t.
- r_t is the immediate reward at time step t.
- γ is the discount factor.
- n is the number of steps to consider for the multi-step reward.
- r_{t+k} is the immediate reward at time step $t + k$.

The multi-step reward is used to update the Q-values in the same way as the standard Bellman equation, but it incorporates the rewards from multiple steps. This allows the agent to learn from longer-term dependencies and can lead to faster convergence and better performance. The multi-step reward is implemented in the `PrioritizedReplayBuffer` class.

```python
class PrioritizedReplayBuffer:
    def __init__(self, capacity, alpha=0.6, beta=0.4, n_steps=3, gamma=0.99):
        self.capacity = capacity
        self.alpha = alpha
        self.beta = beta
        self.n_steps = n_steps
        self.gamma = gamma

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

    def add(self, transition, error):
        self.n_step_buffer.append(transition)
        if len(self.n_step_buffer) < self.n_steps:
            return

        # Compute multistep reward and next state
        reward, next_state, done = self._get_multistep_transition()
        state, action = self.n_step_buffer[0][:2]
        multistep_transition = (state, action, reward, next_state, done)

        priority = (abs(error) + 1e-6) ** self.alpha

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

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

    def _get_multistep_transition(self):
        reward, next_state, done = 0, None, False
        for idx, (_, _, r, ns, d) in enumerate(self.n_step_buffer):
            reward += (self.gamma**idx) * r
            next_state, done = ns, d
            if done:
                break
        return reward, next_state, done

    def sample(self, batch_size):
        assert len(self.buffer) >= batch_size, "Not enough samples to sample from"
```

```python

        valid_size = len(self.buffer)
        probs = self.priorities[:valid_size]
        probs = probs / probs.sum()

        indices = np.random.choice(valid_size, batch_size, p=probs)
        samples = [self.buffer[i] for i in indices]

        total = valid_size
        weights = (total * probs[indices]) ** (-self.beta)
        weights /= weights.max()
        return samples, indices, weights.astype(np.float32)

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

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

2.6 Weight & Biases Using Techniques

Since we are required to complete 3 different tasks in two different environments, I used the Weight & Biases (WandB) library to track the training process and visualize the results.

2.6.1 Categorizing the Results

To categorize the results, I used the `wandb.init` method to create a new run for each task and add tags to the run. This allows me to easily filter and compare the results of different runs in the WandB dashboard for the same task.

2.6.2 Snapshotting the Code

I also used set argument `save_code` to `True` and setting the `code_dir` argument to the whole current directory to save all `.py` files in the WandB run. This allows me to easily reproduce the results and compare the code used for different runs.

2.6.3 Hyperparameter Tuning

I used the `wandb.config` method to define the hyperparameters for each run. This allows me to easily track and compare the hyperparameters used for different runs in the WandB dashboard.

2.6.4 Result Plot Generation

I used the `wandb.log` method to log the results of each run. This allows me to easily visualize the results in the WandB dashboard and compare the performance of different runs. This also allows me to easily export the plots and use them in the report.

2.6.5 WandB Initialization

The implementation of the WandB initialization in the `Trainer` class is as follows:

```
    wandb.init(
        project="DLP-Lab5-DQN",
        name=f"{args.exp}",
        tags=[args.env, "train"],
        save_code=True,
        settings=wandb.Settings(code_dir="."),
    )
    wandb.config.update(args)
```

And the implementation of the WandB initialization in the `Tester` class is as follows:

```
        project="DLP-Lab5-DQN",
        name=f"{args.exp}",
        tags=[args.env, "test"],
        save_code=True,
        settings=wandb.Settings(code_dir="."),
    )
    wandb.config.update(args)
```

3 Discussion

3.1 Task 1: `CartPole-v1` with vanilla DQN

3.1.1 Training Commands

For the task 1, which required to use the vanilla DQN algorithm, I used the following command to train the agent:

```
python3 trainer.py --env cartpole --exp report --vanilla --epsilon-decay
↪ 0.99
```

The command specifies the environment as `CartPole-v1`, the experiment name as `report`, and the epsilon decay rate as 0.99. The `--vanilla` flag indicates that the agent should use the vanilla DQN algorithm without any modifications. Unlike Double DQN (DDQN), vanilla DQN uses the same network for both action selection and value estimation, which can lead to overestimation of Q-values. DDQN mitigates this issue by using the target network for value estimation while using the main network for action selection. The `--epsilon-decay` flag specifies the decay rate for the epsilon value, which controls the exploration-exploitation trade-off during training. The other hyperparameters are set to their default values.

The hyperparameters values are as follows:

- **Batch size**: 32

- **Discount factor**: 0.99
- **Learning rate**: 0.0001
- **Memory size**: $100,000$
- **Epsilon start**: 1
- **Epsilon decay**: 0.99
- **Epsilon min**: 0.015
- **PER alpha**: 0
- **PER beta**: 1
- **N Steps**: 1
- **Update Period**: 1
- **Number of episodes**: 500
- **Seed**: 42
- **Evaluation episodes**: 10

The training process consists of 500 episodes, and the evaluation reward is calculated as the average reward over 10 evaluation episodes. The training process is performed using the `Trainer` class, which handles the training loop and updates the model.

3.1.2 Training Curves

The training curves for the task 1 are shown in Figure 1.

Figure 1: Training curves for the `CartPole-v1` environment.

3.1.3 Testing Commands

We can test the trained agent using the following command:

```
python3 tester.py --env cartpole --exp report --model
 ↪  ./results/cartpole/report/best_model.pt --episodes 30
```

This can achieve the score over than 480 in 30 consecutive episodes.

3.2 Task 2: ALE/Pong-v5 with vanilla DQN

3.2.1 Training Commands

For the task 2, which required to use the vanilla DQN algorithm, I used the following command to train the agent:

```
python3 trainer.py --env pong --exp report --vanilla --episodes 2500
 ↪  --eval-episodes 2
```

The command specifies the environment as `ALE/Pong-v5`, the experiment name as `report`, and the `--vanilla` flag indicates that the agent should use the vanilla DQN algorithm without any modifications. Unlike Double DQN (DDQN), vanilla DQN uses the same network for both action selection and value estimation, which can lead to overestimation of Q-values. DDQN mitigates this issue by using the target network for value estimation while using the main network for action selection. The `--episodes` flag specifies the maximum number of episodes for training, which is set to 2500 in this case. The `--eval-episodes` flag specifies the number of evaluation episodes, which is set to 2 in this case. The other hyperparameters are set to their default values.

The hyperparameters values are as follows:

- **Batch size**: 32
- **Discount factor**: 0.99
- **Learning rate**: 0.000025
- **Memory size**: $100,000$
- **Epsilon start**: 1
- **Epsilon decay**: 0.99999
- **Epsilon min**: 0.005
- **PER alpha**: 0
- **PER beta**: 1
- **N Steps**: 1
- **Update Period**: 1

- **Number of episodes**: 500
- **Seed**: 42
- **Evaluation episodes**: 2

3.2.2 Training Curves

The training curves for the task 2 are shown in Figure 2.

Figure 2: Training curves for the `ALE/Pong-v5` environment.

We can see that the agent can achieve the score of 19 in some of the evaluation episodes.

3.2.3 Testing Commands

We can test the trained agent using the following command:

```
python3 tester.py --env pong --exp report --model ./results/pong/report/best_model.pt --episodes 20
```

This can achieve the score of 19.45 in 20 consecutive episodes.

3.3 Task 3: `ALE/Pong-v5` with DQN Variants

3.3.1 Training Commands

For the task 3, which required to use the DQN variants.

However, I use the vanilla DQN algorithm for the `ALE/Pong-v5` environment and **it can achieve the score of 21 (evaluated in 30 consecutive episodes) in about $140k$ environment steps**.

The training commands are as follows:

```
python3 trainer.py --env pong --exp task3 --vanilla --update-period 1 --n-step 1
```

The command specifies the environment as `ALE/Pong-v5`, the experiment name as `task3`, and the `--vanilla` flag indicates that the agent should use the vanilla DQN algorithm without any modifications.

The other hyperparameters are set to their default values. The hyperparameters values are as follows:

- **Batch size**: 32
- **Discount factor**: 0.99
- **Learning rate**: 0.0001
- **Memory size**: $100,000$
- **Epsilon start**: 1
- **Epsilon decay**: 0.99999
- **Epsilon min**: 0.005
- **PER alpha**: 0
- **PER beta**: 1
- **N Steps**: 1
- **Update Period**: 1
- **Number of episodes**: 500
- **Seed**: 42
- **Evaluation episodes**: 10

3.3.2 Training Curves

The training curves for the task 3 are shown in Figure 3.

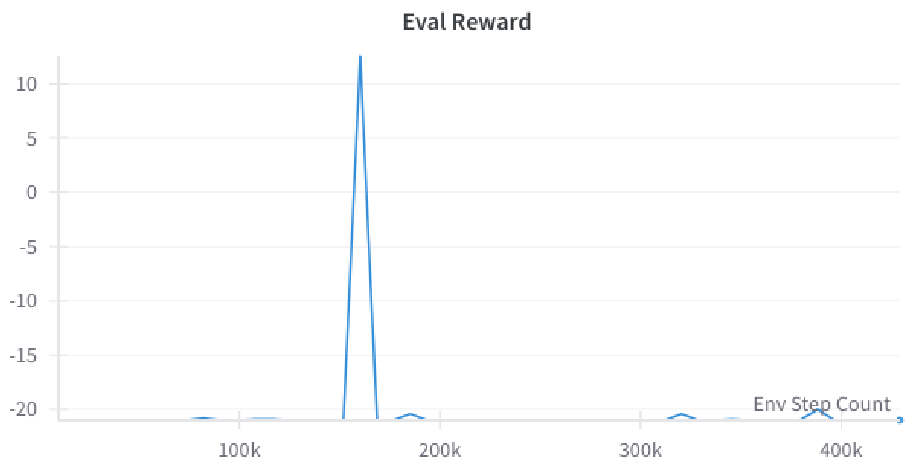

Figure 3: Training curves for the `ALE/Pong-v5` environment.

We can see that the evaluation result shows that the agent can not achieve well performance in the ALE/Pong-v5 environment. It always get the score of -21 except a special snapshot that the agent can get the score over than 10. By the way, this evaluation score during the training process is the average score of 10 episodes.

After inspecting the special snapshot, I found that the agent can get the score of 21 in some episodes by moving to a specific position and hitting the ball at the right time without moving the paddle again. However, in the other episodes, the agent can not hit any ball and the score is -21.

3.3.3 Testing Commands

Therefore, during the testing process, if we use this snapshot to test the agent in the ALE/Pong-v5 environment, the agent can get the score of 21 in 30 consecutive episodes by the following testing command:

```
python3 tester.py --env pong --exp task3 --model
 ↪  ./results/pong/task3/best_model.pt --seed 92439410 --episodes 30
```

The command specifies the environment as ALE/Pong-v5, the experiment name as task3, and the --model flag indicates that the agent should use the model saved in the best_model.pt file, you should change the path to your own path. The --seed flag specifies the first random seed for the environment, and the following seed for the environment will be $seed$+episode_number. That is to say, the first episode will use the seed 92439410, the second episode will use the seed 92439411, and so on. The --episodes flag specifies the number of evaluation episodes, which is set to 30 in this case.

3.4 Analyze each Technique

The following sections outline the key techniques employed in the DQN algorithm and their impact on the training process. Due to the extensive number of experiments conducted, including numerous subtle variations in hyperparameters, this report focuses on describing the techniques and analyzing their effects rather than presenting all training curves. For further details or to reproduce the results, please refer to the implementation in the trainer.py and dqn.py files.

3.4.1 Double DQN

Vanilla DQN uses a single target network to estimate the Q-value of the action it also selected, which tends to overestimate action values because both selection and evaluation share the same (noisy) numbers. Double DQN (DDQN) keeps the overall architecture and loss function the same but splits those two roles: the online network picks the action $a = \arg\max_a Q_{\text{online}}(s', a)$, while the target network supplies its value $Q_{\text{target}}(s', a)$ for the TD target. By decoupling selection from evaluation, DDQN dramatically reduces this positive bias, yielding more accurate value estimates, stabler learning curves, and usually better final performance—especially in environments with many actions or large reward variance—without adding extra networks or hyper-parameters beyond the standard target-network update already present in DQN.

3.5 Prioritized Experience Replay

Prioritized Experience Replay (PER) replaces the uniform-random sampling used in vanilla DQN's replay buffer with a probability $P_i \propto |\delta_i|^\alpha$, where δ_i is the transition's latest TD-error and $\alpha \in [0, 1]$ controls how aggressively "surprising" experiences are favored. Transitions that the network currently mis-predicts (large $|\delta|$) are therefore replayed more often, accelerating the correction of large errors and speeding convergence. Because this biased sampling breaks the i.i.d. assumption, PER attaches an importance-sampling weight $w_i = (\frac{1}{N}\frac{1}{P_i})^\beta$ (with annealed $\beta \in [0, 1]$) to each gradient to recover an unbiased estimate of the expected update. In practice, PER improves sample efficiency and final scores on many Atari and continuous-control tasks, costs only an extra log-time lookup via a SumTree or segment tree, and stacks well with other improvements like Double DQN or dueling networks—but it introduces two extra hyper-parameters (α, β) and can over-focus on a small subset of transitions if α is set too high.

3.6 Multi-Step Learning

Multi-step (n-step) learning replaces DQN's single-step TD target with an n-step return that rolls rewards forward for n steps before bootstrapping:

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n \max_{a'} Q_{\text{target}}(s_{t+n}, a').$$

This amplifies the learning signal by injecting several real rewards before any bootstrapping noise, letting the agent propagate credit faster through time and making early training less sensitive to inaccurate value estimates. Because the return mixes Monte-Carlo (more accurate but high-variance) and TD (more biased but low-variance) signals, n controls a bias-variance trade-off: $n = 1$ reduces to vanilla DQN, $n \to \infty$ becomes a full Monte-Carlo target, and typical values $n \in [3, 10]$ give the best of both worlds. Multi-step targets are cheap to compute in a replay buffer (store cumulative discounted reward and terminal flag) and combine smoothly with other upgrades such as Double DQN and PER, often yielding faster convergence and higher final scores—especially in sparse-reward or long-horizon tasks—without adding new networks or extra hyper-parameters beyond the choice of n.

3.7 Additional Training Tricks

3.7.1 Multiple Evaluation Episodes

I found that the evaluation process is stochastic and the evaluation score is not stable during the training process. This means that the evaluation score can be affected by the random seed used in the environment and will cause the evaluation score to be unexpectly high or low. This will cause the stored best model to be not the best model in the training process. Therefore, I used multiple evaluation episodes to calculate the average evaluation score during the training process. This can help to reduce the variance of the evaluation score and make the evaluation score more stable. This can be achieved by using the `-eval-episodes` flag in the training command. The `-eval-episodes` flag specifies the number of evaluation episodes, which is set to 10 by default.

3.7.2 Learning Rate Scheduler

I found that the learning rate is a very important hyper-parameter in the training process. The learning rate controls the step size of the gradient descent algorithm and can affect the convergence speed and stability of the training process. From my experiments, I found that if the learning rate is too high, the training process will stuck at a suboptimal solution and the evaluation score will not improve no matter how many episodes are trained. On the other hand, if the learning rate is too low, the training process will be very slow and the evaluation score, but the evaluation score will keeps improving. Therefore, I used a learning rate scheduler to reduce the learning rate during the training process, which can help to improve the convergence speed and stability of the training process.

References

[1] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, 2016, pp. 2094–2100. [Online]. Available: `https://arxiv.org/abs/1509.06461`.

[2] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *4th International Conference on Learning Representations (ICLR)*, arXiv:1511.05952, 2016. [Online]. Available: `https://arxiv.org/abs/1511.05952`.

[3] B. Daley, M. White, and M. C. Machado, "Averaging *n*-step returns reduces variance in reinforcement learning," in *Proceedings of the 41st International Conference on Machine Learning (ICML)*, arXiv:2402.03903, 2024. [Online]. Available: `https://arxiv.org/abs/2402.03903`.