Lab 5: Value-Based Reinforcement Learning

Name: CHU, PO-JUI Student ID: 11028141

Date: April 30, 2025

1 Introduction

This experiment focuses on implementing and comparing the performance of classical Q-learning and its deep variant, Deep Q-Network (DQN), across two reinforcement learning tasks from the OpenAI Gym platform: CartPole-v1 (with vector-based observations) and Pong-v5 (with visual observations). The primary objective is to quantify the impact of vanilla DQN versus several enhanced variants—namely Double DQN, Prioritized Experience Replay (PER), Multi-Step Return, Dueling Network, and Cosine Learning Rate Scheduling—in terms of both sample efficiency and final performance.

In Task 1, vanilla DQN was successfully applied to solve CartPole-v1, achieving a stable average reward of 500. Task 2 extended the same vanilla DQN architecture to Pong-v5, where the agent eventually reached an average score of 19 after approximately 6.5 million environment steps. In Task 3, we integrated three major enhancements—Double DQN, PER, and Multi-Step Return—into the training pipeline. As a result, the enhanced agent was able to achieve a stable score of 19 within just 800k environment steps, marking an 8× improvement in sample efficiency compared to the vanilla setup.

The overall framework consists of three main components:

1. **Preprocessing:** We design an `IdentityPreprocessor` for CartPole-v1 to convert raw state vectors to `float32`, and an `AtariPreprocessor` for Pong-v5 that converts RGB frames to grayscale, resizes them to 84×84, and stacks four frames. For the enhanced tasks, we also employ `LazyFrames` to defer frame stacking until sampling, saving approximately 4× memory.

2. **Network Architectures:** We implement three policy networks: an MLP_DQN (4–256–256–2) for Task 1, a standard Atari CNN (Nature DQN) for Task 2, and a DuelingDQN (shared CNN with value and advantage streams) for Task 3, comparing their parameter counts and performance across tasks.

3. **Training Strategy:** All tasks use ϵ-greedy exploration and Huber loss for TD-error. We periodically synchronize the target network. Replay buffers vary by task: uniform sampling for Tasks 1 and 2, and PER combined with an N-step buffer ($n = 5$) for Task 3, allowing us to assess the impact of different sampling strategies on learning efficiency.

Finally, we compare vanilla DQN with enhanced DQN variants (Double DQN, PER, Multi-Step Return, Dueling Network) in terms of sample efficiency and final performance based on average episode reward and the number of steps to convergence.

2 Shared Components

2.1 Preprocessing

Each task applies a different preprocessing pipeline based on the nature of the observation:

Task 1 (CartPole-v1): Uses the `IdentityPreprocessor` to convert raw vector-based observations into `float32` format.

```
class IdentityPreprocessor:
    def reset(self, obs):
        return np.array(obs, dtype=np.float32)
    def step(self, obs):
        return np.array(obs, dtype=np.float32)
```

Listing 1: IdentityPreprocessor for CartPole-v1

Task 2 (Pong-v5): Applies the `AtariPreprocessor`, which converts RGB frames to grayscale, resizes them to 84×84, and stacks four consecutive frames to form the input.

```
class AtariPreprocessor:
    def __init__(self, stack_size=4):
        self.frames = deque(maxlen=stack_size)
    def reset(self, obs):
        gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
        resized = cv2.resize(gray, (84,84))
        self.frames.clear()
        for _ in range(stack_size):
            self.frames.append(resized)
        return np.stack(self.frames, axis=0)
    def step(self, obs):
        gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY)
        resized = cv2.resize(gray, (84,84))
        self.frames.append(resized)
        return np.stack(self.frames, axis=0)
```

Listing 2: AtariPreprocessor for Pong-v5

Task 3 (Enhanced Pong): Uses the same preprocessing pipeline as Task 2 but incorporates `LazyFrames`, a memory-efficient wrapper that defers the actual stacking operation until sampling, effectively reducing memory usage by approximately **4** times

```
class LazyFrames:
    """Share memory between stacked frames to save RAM (Task 3)."""
    def __init__(self, frames: List[np.ndarray]):
        self._frames = list(frames)
        self._out = None

    def __array__(self, dtype=None, copy=False):
        if self._out is None:
            self._out = np.stack(self._frames, axis=0)
            self._frames = None  # drop reference
        arr = self._out
```

```
        if dtype is not None:
            arr = arr.astype(dtype)
        return arr.copy() if copy else arr

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

    def __getitem__(self, idx):
        return self.__array__()[idx]
```

Listing 3: LazyFrames Wrapper (Used in Task 3)

2.2 Network Architectures

We design different network architectures depending on the task complexity and input type:

- **Task 1:** MLP_DQN with a structure of 4–256–256–2, totaling approximately 0.18 million parameters.
- **Task 2:** CNN_DQN following the standard Nature DQN convolutional architecture, with about 1.68 million parameters.
- **Task 3:** DuelingDQN architecture that shares a CNN backbone but splits into separate value and advantage streams, containing roughly 1.72 million parameters.

2.3 Replay Buffer

The experience replay strategy differs across tasks:

- **Task 1 & 2:** Use a uniform sampling strategy implemented via a deque buffer.
- **Task 3:** Employs Prioritized Experience Replay (PER) in combination with an N-step return buffer ($n = 5$), enhancing sample quality and learning speed.

2.4 Training & Tracking

All tasks follow a shared training protocol:

- Exploration follows an ϵ-greedy strategy.
- The Huber loss is used to compute the temporal difference error.
- The target network is periodically synchronized with the online network.
- Training metrics are logged in real-time using Weights & Biases:

```
wandb.log({
    "episode": ep,
    "reward": total,
    "epsilon": eps,
    "loss": loss.item(),
    "steps": env_step,
})
```

Listing 4: Logging with Weights & Biases

3 Task 1: Vanilla DQN on CartPole-v1

3.1 Hyperparameters

Parameter	Value
Learning rate	5×10^{-4}
ϵ-decay	0.9999
Target sync	every 1,000 steps
Replay start	1,000 steps
Batch size	64
Memory size	100,000
Max episodes	3,000 (early stop)

3.2 Network Architecture

The MLP_DQN used in Task 1 consists of three layers:

1. **Input layer to Hidden Layer 1:**
   - A linear projection using `nn.Linear(state_dim, 256)` maps the environment state vector to a 256-dimensional space.
   - A ReLU activation function introduces non-linearity: `nn.ReLU()`.
2. **Hidden Layer 1 to Hidden Layer 2:**
   - Another linear transformation `nn.Linear(256, 256)` keeps the feature dimension unchanged.
   - ReLU is again applied to strengthen representational capacity.
3. **Hidden Layer 2 to Output Layer:**
   - `nn.Linear(256, num_actions)` maps the features to action values, producing the Q-value estimate for each action.

```
class MLP_DQN(nn.Module):
    def __init__(self, state_dim, num_actions):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 256), nn.ReLU(),
            nn.Linear(256, 256), nn.ReLU(),
            nn.Linear(256, num_actions),
        )
```

Listing 5: MLP_DQN Architecture

3.3 Bellman Error and Implementation

The DQN loss function is based on the temporal difference (TD) error, defined as:

$$L(\theta) = \tfrac{1}{2}\big(r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \bar{\theta}) - Q(s_t, a_t; \theta)\big)^2, \quad \delta_t = r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \bar{\theta}) - Q(s_t, a_t; \theta)$$

This loss measures the discrepancy between the predicted Q-value and the target Q-value computed using the target network $\bar{\theta}$. The components are:

- $r_t + \gamma \max_{a'} Q(s_{t+1}, a'; \bar{\theta})$ is the target Q-value representing the best future return from the next state.
- $Q(s_t, a_t; \theta)$ is the current Q-value estimate from the online network.
- The TD error δ_t reflects how much the current estimate deviates from the target. A nonzero δ_t indicates learning is still in progress.
- The squared TD error is minimized using gradient descent to update the network parameters θ.

This loss formulation is the core mechanism in DQN for guiding the network toward accurate Q-value estimation.

```
q = q_net(states).gather(1, actions).squeeze(1)
with torch.no_grad():
    next_q = target_net(next_states).max(1)[0]

target = rewards + (1 - dones) * gamma * next_q
loss   = F.smooth_l1_loss(q, target)
```

Listing 6: Bellman Loss Calculation

The target network is synchronized every 10,000 gradient updates using:

```
target_net.load_state_dict(q_net.state_dict())
```

3.4 Preprocessor

The `IdentityPreprocessor` is responsible for converting the raw observations into float32 vectors suitable for neural network input.

```
processor = IdentityPreprocessor()
state0 = processor.reset(obs0)  # float32 vector
```

3.5 Training Run and Early Stopping

The training loop includes an early stopping mechanism to detect convergence:

→ If an episode achieves a reward ≥ 500, the agent is evaluated over 20 episodes. When the average reward in those episodes is also ≥ 500, training stops and the model is saved.

```
if self.task_num == 1 and total >= 500:
    avg = self.evaluate(20)
    if avg >= 500:
        torch.save(self.q_net.state_dict(), os.path.join(self.save_dir, "
            best_model_task1.pt"))
        print("  [Save] best_model_task1.pt")
        print("Solved CartPole, stop.")
        break
```

Listing 7: Early-Stopping Logic

4 Task 2: Vanilla DQN on Pong-v5

4.1 Hyperparameters

Parameter	Value
Learning rate	1×10^{-4}
Replay start	50,000 steps
ϵ (start/decay/end)	1 / 0.999995 / 0.05
γ	0.99
Batch size	32
Memory size	90,000
Target sync	every 10,000 steps

4.2 Preprocessing and CNN Architecture

Same as Task 1, the preprocessing and CNN architecture follow the standard DQN setup described earlier.

4.3 Training Procedure

During training, key metrics such as reward, ϵ, and environment step count are logged using Weights & Biases. When an episode achieves a reward of at least 17, the current model is evaluated over 20 episodes. If the average reward from this evaluation reaches or exceeds 19, the model is considered successful, and its parameters are saved to `results/model_task2_{ep}_{reward}.pt`.

Once the model achieves this level of performance three times, training is terminated. This strategy preserves three high-performing checkpoints, allowing for more flexible evaluation and selection of the best model in subsequent analysis.

```
if self.task_num == 2 and total >= 17:
    pth = os.path.join(self.save_dir, f"model_task2_ep{ep}_r{int(total)}.
        pt")
    torch.save(self.q_net.state_dict(), pth)
    print(f"  [Snapshot] saved {pth}")

    mean = self.evaluate(20)
    if mean >= 19:
        self.success_cnt += 1
        print("Solved Pong reward over 19 scores.")
    else:
        print(f"Average score only {mean:.1f} < 19, continue training ")

    if self.success_cnt >= 3:
        print("Solved Pong after 3 consecutive successes, stop.")
        break
```

Listing 8: Task 2 Evaluation and Stopping Logic

This setup establishes a strong performance baseline for the vanilla DQN model on Pong, and serves as a foundation for evaluating the enhancements introduced in Task 3.

5 Task 3: Enhanced DQN on Pong-v5

5.1 Hyperparameters

Parameter	Value
Learning rate	3.75×10^{-5}
γ	0.99
N-step	5
Memory size	1,000,000
Batch size	32
PER α/β_0	0.6 / 0.4→1.0
ϵ (start/decay/end)	1 / 0.9999925 / 0.05
Target sync	every 8,000 steps
Train steps	1,000,000
Snapshots	every 200,000 steps

5.2 Double DQN

Double DQN addresses the overestimation bias commonly found in vanilla DQN by decoupling action selection and action evaluation. Instead of using the same network to both select and evaluate actions, Double DQN splits the process to reduce upward bias in Q-value estimation.

Specifically, the next action a^* is selected using the online network:

$$a^* = \arg\max_a Q(s', a; \theta)$$

Then, the value of that action is evaluated using the target network:

$$Q(s', a^*; \bar{\theta})$$

The new TD target becomes:

$$y = r + \gamma\, Q(s', a^*; \bar{\theta})$$

Thus, the Double DQN loss is given by:

$$L_{\text{DDQN}}(\theta) = \tfrac{1}{2}\left(r + \gamma\, Q(s', \arg\max_a Q(s', a; \theta); \bar{\theta}) - Q(s, a; \theta)\right)^2$$

By decoupling action selection from action evaluation, Double DQN mitigates overestimation and improves training stability.

```
with torch.no_grad():
    next_actions = self.online_net(next_states).argmax(dim=1, keepdim=True
        )
    next_q = self.target_net(next_states).gather(1, next_actions).squeeze
        (1)
target = rewards + (1 - dones) * (self.gamma**n_step) * next_q
td_error = target - q_values
```

Listing 9: Double DQN Update

5.3 Prioritized Experience Replay (PER)

PER improves sample efficiency by sampling more frequently from transitions with high learning potential, typically those with larger TD errors. Each transition is assigned a priority:

$$p_i = |\delta_i| + \varepsilon$$

Sampling probability is controlled by exponent α:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

When $\alpha = 0$, sampling becomes uniform. Larger α emphasizes high-priority transitions.

To correct sampling bias, importance-sampling weights are applied:

$$w_i = (NP(i))^{-\beta}$$

β increases from 0.4 to 1 linearly during training and is normalized to ensure numerical stability.

Insertion logic:

```
def push(self, transition: Transition):
    max_prio = self.priorities.max() if self.size > 0 else 1.0
    self.buffer[self.pos] = transition
    self.priorities[self.pos] = max_prio
    self.pos = (self.pos + 1) % self.capacity
    self.size = min(self.size + 1, self.capacity)
```

Sampling logic:

```
def sample(self, batch_size: int):
    prios = self.priorities[: self.size] ** self.alpha
    probs = prios / prios.sum()
    indices = np.random.choice(self.size, batch_size, p=probs)
    samples = [self.buffer[idx] for idx in indices]
    beta = min(1.0, self.beta_start + (self.frame / self.beta_frames) *
       (1.0 - self.beta_start))
    self.frame += 1
    weights = (self.size * probs[indices]) ** (-beta)
    weights /= weights.max()
    return samples, indices, torch.tensor(weights, dtype=torch.float32)
```

Updating priorities:

```
def update_priorities(self, indices, priorities):
    for idx, prio in zip(indices, priorities):
        self.priorities[idx] = prio
```

Compared to uniform replay, PER focuses updates on transitions with high learning value, accelerating convergence and improving final performance.

5.4 Multi-Step Return (n = 5)

Multi-step return accumulates actual rewards over n steps and incorporates them into a single update:

$$R_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n \max_{a'} Q(s_{t+n}, a')$$

→ This enables faster propagation of delayed rewards and improves learning in long-horizon tasks like Pong.

```
class NStepBuffer:
    def __init__(self, n, gamma):
        self.n, self.gamma = n, gamma
        self.buf = deque(maxlen=n)
    def push(self, tr):
        self.buf.append(tr)
        if len(self.buf) < self.n: return None
        R, next_s, done = 0, self.buf[-1].next_state, self.buf[-1].done
        for idx, t in enumerate(self.buf):
            R += (self.gamma**idx) * t.reward
            if t.done: break
        s0, a0 = self.buf[0].state, self.buf[0].action
        return Transition(s0, a0, R, next_s, done)
```

Listing 10: Multi-step Return Buffer

5.5 Dueling Network

The Dueling DQN architecture separates the Q-value estimation into two parallel streams: one for the state-value function $V(s)$ and another for the advantage function $A(s, a)$. These are then combined to form the final Q-value estimate:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

```
class DuelingDQN(nn.Module):
    def __init__(self, in_ch, num_actions):
        super().__init__()
        self.feature = nn.Sequential(
            nn.Conv2d(in_ch, 32, 8, 4), nn.ReLU(),
            nn.Conv2d(32, 64, 4, 2), nn.ReLU(),
            nn.Conv2d(64, 64, 3, 1), nn.ReLU(), nn.Flatten()
        )
        self.fc_val = nn.Sequential(
            nn.Linear(64*7*7, 512), nn.ReLU(), nn.Linear(512, 1)
        )
        self.fc_adv = nn.Sequential(
            nn.Linear(64*7*7, 512), nn.ReLU(), nn.Linear(512, num_actions)
        )

    def forward(self, x):
        x = x / 255.0
        f = self.feature(x)
        v = self.fc_val(f)
        a = self.fc_adv(f)
        return v + (a - a.mean(dim=1, keepdim=True))
```

Listing 11: Dueling Network Architecture

5.6 Cosine Learning Rate Schedule

A cosine annealing scheduler is used to gradually decay the learning rate from a higher value in the early stages to a minimal value at the end of training. This helps balance fast exploration and late-stage stability.

```
self.optimizer = optim.Adam(self.online.parameters(), lr=args.lr)
self.lr_scheduler = optim.lr_scheduler.CosineAnnealingLR(
    self.optimizer, T_max=args.steps, eta_min=1e-6
)
```

Listing 12: Cosine Annealing Learning Rate

The learning rate starts high to encourage exploration and decays smoothly to 1×10^{-6} toward the end to enhance stability.

5.7 Training Summary

Every 200k steps, a regular snapshot of the model is saved. If any episode reaches a reward ≥ 16, a 20-episode evaluation is performed. If the average reward surpasses the previously best performance at that step interval, the current model is saved as a best snapshot.
At most five `best_{200,400,...}k.pt` models are retained by the end of training.

```
if total_reward >= 16:
    avg = self.evaluate(20)
    interval = step // 200_000
    if avg >= best_eval.get(interval, -float('inf')):
        best_eval[interval] = avg
        fname = os.path.join(self.args.snapshot_dir, f"best_{interval*200}
            k.pt")
        torch.save(self.online.state_dict(), fname)
        print(f"[Eval] avg {avg:.2f} > best, saved {fname}")

if step in self.args.snapshots:
    fname = os.path.join(self.args.snapshot_dir, f"model_{step//1000}k.pt"
        )
    torch.save(self.online.state_dict(), fname)
    print(f"[Snapshot] saved {fname}")
```

Listing 13: Training Snapshot Logic

6 Experimental Results

6.1 Evaluation script

6.1.1 `test_model_task1.py` — CartPole-v1 Evaluation Script

This script loads a trained MLP_DQN model and runs it on the CartPole-v1 environment using a greedy ($\epsilon = 0$) policy. It prints the return for each episode and reports the overall mean and standard deviation.

```
import torch, gymnasium as gym, numpy as np, argparse
from dqn import MLP_DQN, IdentityPreprocessor
```

```
def main(path, episodes, render):
    device = "cuda" if torch.cuda.is_available() else "cpu"
    env = gym.make("CartPole-v1", render_mode="human" if render else None)
    model = MLP_DQN(4, 2).to(device)
    model.load_state_dict(torch.load(path, map_location=device))
    model.eval()
    pre = IdentityPreprocessor()

    rewards = []
    for ep in range(episodes):
        obs, _ = env.reset(); state = pre.reset(obs)
        done = False; total = 0
        while not done:
            with torch.no_grad():
                act = model(torch.from_numpy(state).float().unsqueeze(0).
                    to(device)).argmax().item()
            nxt, r, term, trunc, _ = env.step(act)
            done, total = term or trunc, total + r
            state = pre.step(nxt)
        rewards.append(total)
        print(f"ep{ep:03d}  {total}")
    print(f"Mean {np.mean(rewards):.1f}    {np.std(rewards):.1f}")

if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--model", required=True)
    p.add_argument("--eps", type=int, default=20)
    p.add_argument("--render", action="store_true")
    args = p.parse_args()
    main(args.model, args.eps, args.render)
```

Listing 14: Task 1 Evaluation Script

Key Point: This script reuses the existing `MLP_DQN` and `IdentityPreprocessor` classes, keeping the implementation clean and minimal.

6.1.2 `test_model_task2.py` — Pong-v5 Vanilla Evaluation Script

This script loads a standard CNN DQN model for Pong-v5 and evaluates it over multiple episodes. Each episode's visual frames are saved as video using `imageio`.

```
import torch, numpy as np, random, argparse, os, imageio
import gymnasium as gym, cv2
from collections import deque
import torch.nn as nn

class DQN(nn.Module):
    def __init__(self, input_channels, num_actions):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(input_channels, 32, 8, 4), nn.ReLU(),
            nn.Conv2d(32, 64, 4, 2), nn.ReLU(),
            nn.Conv2d(64, 64, 3, 1), nn.ReLU(), nn.Flatten(),
```

```
            nn.Linear(64*7*7, 512), nn.ReLU(),
            nn.Linear(512, num_actions)
        )
    def forward(self, x):
        return self.net(x / 255.0)

class AtariPreprocessor:
    def __init__(self, frame_stack=4):
        self.frames = deque(maxlen=frame_stack)
    def preprocess(self, obs):
        gray = cv2.cvtColor(obs, cv2.COLOR_RGB2GRAY) if obs.ndim == 3 else
            obs
        return cv2.resize(gray, (84,84), interpolation=cv2.INTER_AREA)
    def reset(self, obs):
        f = self.preprocess(obs)
        self.frames = deque([f]*self.frames.maxlen, maxlen=self.frames.
           maxlen)
        return np.stack(self.frames, 0)
    def step(self, obs):
        self.frames.append(self.preprocess(obs))
        return np.stack(self.frames, 0)

def evaluate(args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    random.seed(args.seed); np.random.seed(args.seed); torch.manual_seed(
       args.seed)
    env = gym.make("ALE/Pong-v5", render_mode="rgb_array")
    pre = AtariPreprocessor(); num_actions = env.action_space.n

    model = DQN(4, num_actions).to(device)
    model.load_state_dict(torch.load(args.model_path, map_location=device)
       )
    model.eval()

    os.makedirs(args.output_dir, exist_ok=True)
    rewards = []
    for ep in range(args.episodes):
        obs, _ = env.reset(seed=args.seed + ep)
        state = pre.reset(obs); done=False; total=0; frames=[]
        while not done:
            frames.append(env.render())
            with torch.no_grad():
                action = model(torch.from_numpy(state).float().unsqueeze
                   (0).to(device)).argmax().item()
            nxt, r, term, trunc, _ = env.step(action)
            done, total = term or trunc, total + r
            state = pre.step(nxt)
        out = os.path.join(args.output_dir, f"eval_ep{ep}.mp4")
        with imageio.get_writer(out, fps=30) as vid:
            for f in frames: vid.append_data(f)
        print(f"Ep {ep}: {total}     {out}")
        rewards.append(total)
    mean, std = np.mean(rewards), np.std(rewards)
    print(f"\nMean over {len(rewards)} eps     {mean:.1f}    {std:.1f}")
```

```
if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--model-path", required=True)
    p.add_argument("--output-dir", default="./eval_videos")
    p.add_argument("--episodes", type=int, default=10)
    p.add_argument("--seed", type=int, default=313551076)
    args = p.parse_args()
    evaluate(args)
```

Listing 15: Task 2 Evaluation Script

Note: The model's internal **network** attribute was renamed to **net**, and all inputs are normalized to $[0, 1]$. Videos and score statistics are saved for every run.

6.1.3 `test_model_task3.py` — Pong-v5 Enhanced Evaluation Script

This script evaluates a fully enhanced Dueling DQN model trained with Double DQN, PER, Multi-Step Return, and Cosine LR schedule. It saves rendered videos and prints episode statistics.

```
import torch, numpy as np, random, argparse, os, imageio
import gymnasium as gym, cv2
from collections import deque
from task3_gpt import DuelingDQN, AtariPreprocessor

def evaluate(args):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    random.seed(args.seed); np.random.seed(args.seed); torch.manual_seed(
        args.seed)
    env = gym.make("ALE/Pong-v5", render_mode="rgb_array")
    pre = AtariPreprocessor(stack_size=4); num_actions = env.action_space.
        n

    model = DuelingDQN(4, num_actions).to(device)
    model.load_state_dict(torch.load(args.model_path, map_location=device)
        )
    model.eval()

    os.makedirs(args.output_dir, exist_ok=True)
    rewards = []
    for ep in range(args.episodes):
        obs, _ = env.reset(seed=args.seed + ep)
        state = pre.reset(obs); done=False; total=0; frames=[]
        while not done:
            frames.append(env.render())
            with torch.no_grad():
                action = model(torch.from_numpy(np.array(state, dtype=np.
                    float32))
                                        .unsqueeze(0).to(device)).argmax().
                                            item()
            nxt, r, term, trunc, _ = env.step(action)
            done, total = term or trunc, total + r
            state = pre.step(nxt)
```

```
        out = os.path.join(args.output_dir, f"eval_ep{ep}.mp4")
        with imageio.get_writer(out, fps=30) as vid:
            for f in frames: vid.append_data(f)
        print(f"Ep {ep}: {total}     {out}")
        rewards.append(total)
    mean, std = np.mean(rewards), np.std(rewards)
    print(f"\nMean over {len(rewards)} eps     {mean:.1f}    {std:.1f}")

if __name__ == "__main__":
    p = argparse.ArgumentParser()
    p.add_argument("--model-path", required=True)
    p.add_argument("--output-dir", default="./eval_videos_task3")
    p.add_argument("--episodes", type=int, default=10)
    p.add_argument("--seed", type=int, default=2)
    args = p.parse_args()
    evaluate(args)
```

Listing 16: Task 3 Evaluation Script

Enhancements Tested:

- Double DQN
- Prioritized Experience Replay with Multi-Step Return
- Dueling Network Architecture
- Cosine Learning Rate Schedule

6.2 Task 1: CartPole-v1

Convergence to the maximum reward of 500 was achieved at approximately 30k environment steps, with early stopping triggered at Episode 310. The agent surpassed a score of 200 within the first 10k steps, experienced some fluctuation between 20k and 100k steps, and finally stabilized at the 500-point ceiling.

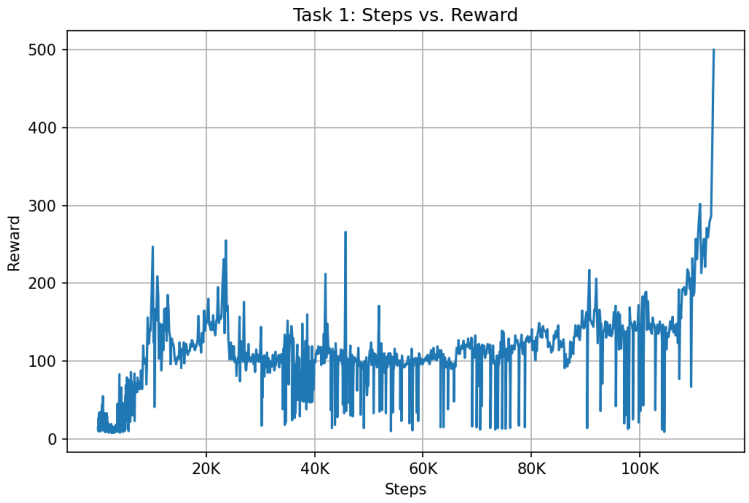

Figure 1: Learning curve for Task 1 (CartPole-v1).

6.3 Task 2: Pong-v5 (Vanilla DQN)

The agent struggled to escape the $[-20, -15]$ range even after 1 M steps. A stable +19 average was first achieved around 6.5 M steps. This slow progress is attributed to single-step returns, uniform sampling, and using one network for both action selection and evaluation.

Figure 2: Learning curve for Task 2 (Pong-v5, Vanilla DQN).

6.4 Bonus: Ablation Study on Dueling Network

To quantify the isolated contribution of the Dueling Network architecture, we conducted an ablation study comparing the following two setups:

- **Vanilla DQN (baseline)**
- **Vanilla DQN + Dueling Architecture**

6.4.1 Baseline Performance Without Dueling

This setup uses a plain Vanilla DQN model evaluated on Pong-v5. The learning curve blue line shows steps returns over time. We observe that the agent remains stuck in the -15 to -5 reward range for the first 200 episodes (roughly 400k environment steps), and only begins to reach around 0 after that point, with high variance in returns. Due to the very slow improvement and unstable learning, the training was halted prematurely.

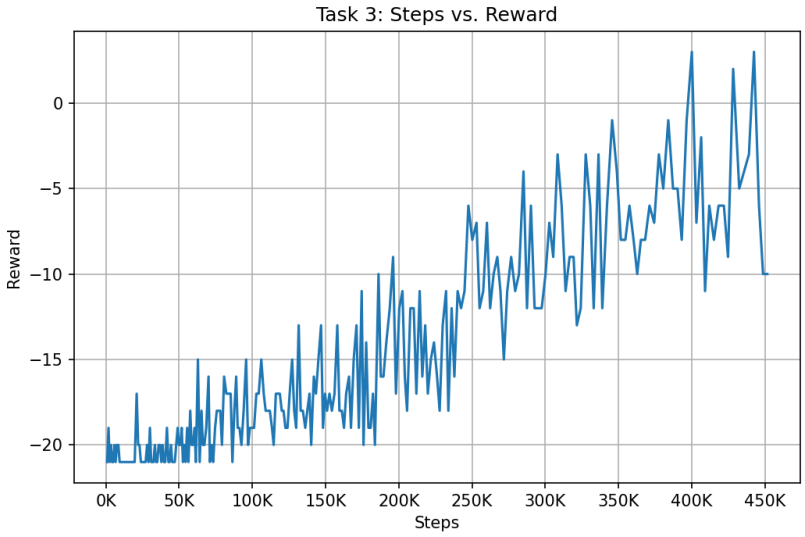

Figure 3: Learning curve of Vanilla DQN baseline on Pong-v5.

6.4.2 Dueling Architecture Implementation

To incorporate the Dueling Network, we split the fully connected layers at the end of the CNN backbone into two streams:

- A value stream $V(s)$ estimating the value of a given state
- An advantage stream $A(s, a)$ estimating the relative benefit of each action

The two outputs are combined using the following formula:

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s,a') \right)$$

All other hyperparameters—learning rate, ϵ-greedy schedule, and replay buffer size—were kept identical to the baseline for fair comparison.

Experimental Results

In the resulting learning curves, the yellow line represents the baseline Vanilla DQN, while the orange line shows the DQN with Dueling architecture.

Key observations:

- The dueling setup significantly reduces variance in learning and improves stability.
- It reaches and sustains an average score of 19 by roughly 2 million environment steps (around episode 1000).
- In contrast, the baseline DQN struggles to exceed zero reward even after 400k steps and often regresses.

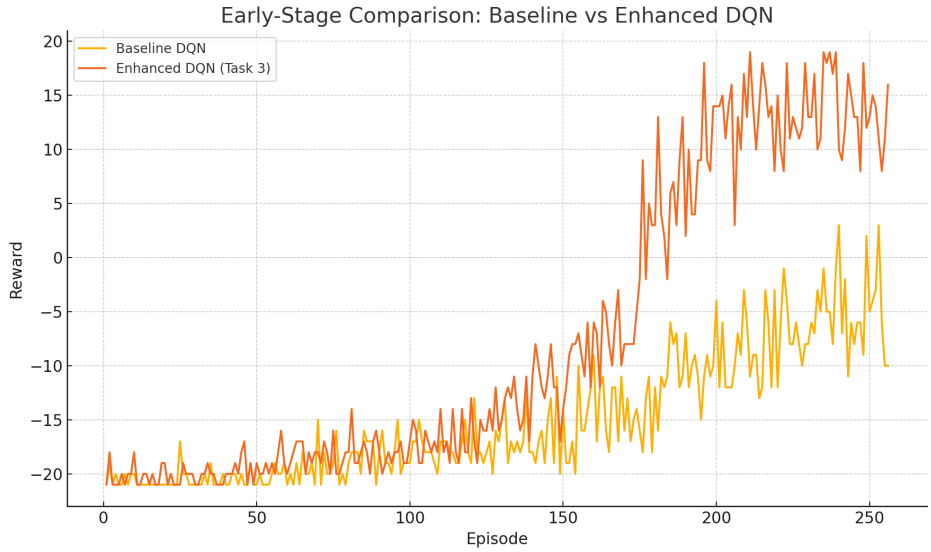

Figure 4: Learning curves for Vanilla DQN (orange) and Dueling DQN (blue) on Pong-v5.

Discussion

Stability: By separating state-value estimation from action-specific advantages, the Dueling architecture prevents noisy updates in states where all actions are similarly valued.

Sample Efficiency: Improved value estimation helps reduce exploration noise and narrows the policy search space early in training, accelerating convergence.

Conclusion: Although the Dueling Network introduces only a small architectural change, it greatly enhances learning stability and sample efficiency in high-variance environments such as Pong-v5. It is a highly effective addition worth considering in practical implementations.

6.5 Task 3: Pong-v5 (Enhanced DQN)

- 200k steps: average reward of +14.3
- 400k steps: average reward of +17.8
- 735k steps: first stable achievement of score ≥ 19
- 1M steps: maintained between +18 and +19

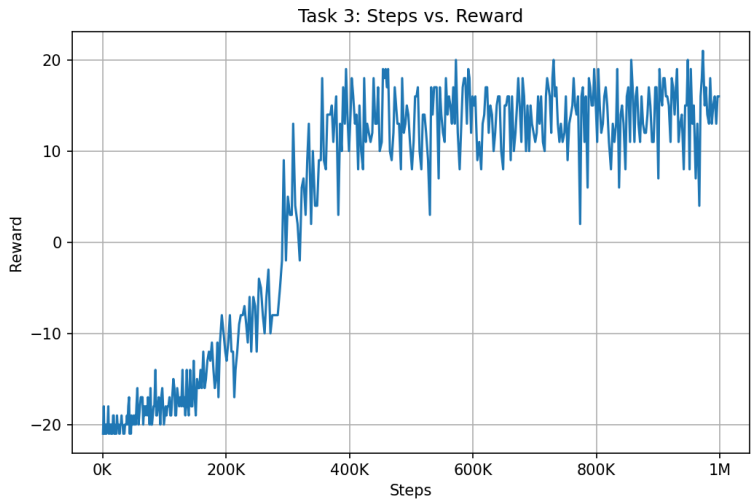

Figure 5: Learning curve for Task 3 (Pong-v5, Enhanced DQN).

6.6 Sample Efficiency Comparison

Method	**Steps to** ≥ 19	**Relative to Vanilla DQN**
Vanilla DQN (Task 2)	$\sim$6.5M	$1\times$
Enhanced DQN (Task 3)	$\sim$0.735M	$\downarrow\approx 89\%$

The enhanced DQN architecture, through the integration of multiple improvements, significantly reduced the number of steps required to solve Pong—from several million down to under one million. This demonstrates a notable boost in both sample efficiency and training stability.

7 Conclusion and Future Work

Conclusion: The use of enhancements such as Double DQN, Prioritized Experience Replay (PER), Multi-Step Return, and the Dueling Network Architecture significantly improves both the sample efficiency and overall performance of DQN. When combined, these techniques reduce the number of environment steps required to solve Pong by approximately 89%.

Future Work:

- Incorporate Rainbow extensions, including NoisyNet and Distributional RL, to further optimize performance.
- Dynamically adjust the replay buffer size and ϵ-decay schedule to better adapt to different tasks.
- Explore adaptive N-step returns and bidirectional priority functions for more refined replay mechanisms.

References

[1] Hessel, M., et al. (2018). *Rainbow: Combining improvements in deep reinforcement learning.* AAAI.

[2] Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). *Prioritized experience replay.* arXiv:1511.05952.