DA6400: Reinforcement Learning
Programming Assignment 2 Report

KUSH RUSHABH SHAH | CH21B053 M S SAI TEJA | ME21B171

April 14, 2025

Abstract

This report describes the implementation and evaluation of two reinforcement learning algorithms on Gymnasium environments. In particular, we implemented two variants of Dueling-DQN (Type-1 and Type-2) and two variants of the Monte Carlo REINFORCE algorithm (with and without a TD(0) baseline) on Acrobot-v1 and CartPole-v1. The experimental results are averaged over 5 random seeds, and performance is compared in terms of episodic returns and variance.

1 Introduction

In this assignment, we are implementing two families of RL algorithms:

- **Dueling-DQN:** where the Q-value is decomposed into a state value function $V(s)$ and an advantage $A(s,a)$. Two update rules are implemented:
  - **Type-1:**
  $$Q(s,a;\theta) = V(s;\theta) + \Big(A(s,a;\theta) - \frac{1}{|A|}\sum_{a' \in A} A(s,a';\theta)\Big)$$
  - **Type-2:**
  $$Q(s,a;\theta) = V(s;\theta) + \Big(A(s,a;\theta) - \max_{a' \in A} A(s,a';\theta)\Big)$$
- **Monte-Carlo REINFORCE:** where the policy is updated using entire episode returns. Two variants are implemented:
  - Without baseline:
  $$\theta \leftarrow \theta + \alpha\, G_t \,\frac{\nabla \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}$$
  - With baseline, where the baseline $V(s;\Phi)$ is updated using TD(0):
  $$\theta \leftarrow \theta + \alpha\,(G_t - V(s_t;\Phi))\,\frac{\nabla \pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}$$

All experiments use a discount factor $\gamma = 0.99$ and are averaged over 5 random seeds.

2 Environments

We test the algorithms on:

Acrobot-v1: A two-link pendulum where the goal is to swing the free end above a given height.

CartPole-v1: A cart-pole system in which the objective is to balance an upright pole on a moving cart.

3 Algorithm Implementations

3.1 Dueling-DQN

The network architecture for Dueling-DQN uses a shared feature extraction layer and splits the output into:

1. A value stream $V(s)$.
2. An advantage stream $A(s, a)$.

The final Q-values are computed according to either Type-1 or Type-2 aggregation.

3.1.1 Dueling-DQN Code Snippet

```
class DuelingDQN(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_dim=128):
        super(DuelingDQN, self).__init__()
        % Shared feature extraction
        self.feature = nn.Sequential(
            nn.Linear(state_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU()
        )
        % Value stream: computes V(s)
        self.value_stream = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim//2),
            nn.ReLU(),
            nn.Linear(hidden_dim//2, 1)
        )
        % Advantage stream: computes A(s,a)
        self.advantage_stream = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim//2),
            nn.ReLU(),
            nn.Linear(hidden_dim//2, action_dim)
        )

    def forward(self, state, aggregation_type=1):
        features = self.feature(state)
        value = self.value_stream(features)
        advantages = self.advantage_stream(features)
        if aggregation_type == 1:
            % Type-1: subtract the mean advantage
            return value + (advantages - advantages.mean(dim=1, keepdim=True))
        else:
            % Type-2: subtract the maximum advantage
            return value + (advantages - advantages.max(dim=1, keepdim=True)
                [0])
```

Listing 1: Dueling-DQN Network Architecture and Forward Pass

```
class DuelingDQNAgent:
    """Agent implementing Dueling DQN with both Type-1 and Type-2 update rules.
        """
    def __init__(self, state_dim, action_dim, aggregation_type=1,
                 lr=0.001, gamma=0.99, epsilon_start=1.0,
                 epsilon_min=0.01, epsilon_decay=0.995,
                 buffer_size=10000, batch_size=64,
                 target_update=10, hidden_dim=128):
        self.state_dim = state_dim
        self.action_dim = action_dim
```

```python
        self.aggregation_type = aggregation_type
        self.gamma = gamma
        self.epsilon = epsilon_start
        self.epsilon_min = epsilon_min
        self.epsilon_decay = epsilon_decay
        self.batch_size = batch_size
        self.target_update = target_update
        self.update_count = 0

        # Initialize networks
        self.policy_net = DuelingDQN(state_dim, action_dim, hidden_dim)
        self.target_net = DuelingDQN(state_dim, action_dim, hidden_dim)
        self.target_net.load_state_dict(self.policy_net.state_dict())
        self.target_net.eval()

        # Initialize optimizer
        self.optimizer = optim.Adam(self.policy_net.parameters(), lr=lr)

        # Initialize replay buffer
        self.buffer = ReplayBuffer(buffer_size)

        # Device configuration
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.policy_net.to(self.device)
        self.target_net.to(self.device)

    def select_action(self, state, training=True):
        """Select action using epsilon-greedy policy."""
        if training and np.random.rand() < self.epsilon:
            # Exploration: random action
            return np.random.randint(self.action_dim)
        else:
            # Exploitation: greedy action
            state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
            with torch.no_grad():
                q_values = self.policy_net(state, self.aggregation_type)
            return q_values.argmax().item()

    def update(self):
        """Update the policy network using a batch of experiences."""
        if len(self.buffer) < self.batch_size:
            return 0

        # Sample batch from replay buffer
        states, actions, rewards, next_states, dones = self.buffer.sample(self.batch_size)

        # Convert to tensors
        states = torch.FloatTensor(states).to(self.device)
        actions = torch.LongTensor(actions).to(self.device)
        rewards = torch.FloatTensor(rewards).to(self.device)
        next_states = torch.FloatTensor(next_states).to(self.device)
        dones = torch.FloatTensor(dones).to(self.device)

        # Get current Q values
        q_values = self.policy_net(states, self.aggregation_type).gather(1, actions.unsqueeze(1))

        # Get next Q values from target network
        with torch.no_grad():
            next_q_values = self.target_net(next_states, self.aggregation_type).max(1)[0]
```

```python

        # Compute target Q values
        target_q_values = rewards + (1 - dones) * self.gamma * next_q_values

        # Compute loss
        loss = F.mse_loss(q_values.squeeze(), target_q_values)

        # Optimize
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update target network periodically
        self.update_count += 1
        if self.update_count % self.target_update == 0:
            self.target_net.load_state_dict(self.policy_net.state_dict())

        # Decay epsilon
        self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)

        return loss.item()

    def train(self, env, num_episodes, max_steps=500):
        """Train the agent over multiple episodes."""
        episode_returns = []

        for episode in range(num_episodes):
            state, _ = env.reset()
            episode_return = 0

            for step in range(max_steps):
                # Select and perform action
                action = self.select_action(state)
                next_state, reward, done, truncated, _ = env.step(action)

                # Store transition in replay buffer
                self.buffer.push(state, action, reward, next_state, done)

                # Update state and episode return
                state = next_state
                episode_return += reward

                # Update the network
                loss = self.update()

                if done or truncated:
                    break

            episode_returns.append(episode_return)
            print(f"Episode {episode+1}, Return: {episode_return}, Epsilon: {
                self.epsilon:.4f}")

        return episode_returns
```

Listing 2: Dueling-DQN Agent Code

3.2 Monte-Carlo REINFORCE

The REINFORCE agent collects complete episode returns, then updates the policy network using either:

- The full Monte Carlo return G_t (without baseline), or

- The advantage $G_t - V(s_t)$ (with baseline), where the value network is updated at each step using TD(0).

3.2.1 REINFORCE Agent Code Snippet

```
class REINFORCEAgent:
    def __init__(self, state_dim, action_dim, use_baseline=False,
                 lr_policy=1e-3, lr_value=1e-3, gamma=0.99, hidden_dim=128):
        self.use_baseline = use_baseline
        self.gamma = gamma
        self.policy_net = PolicyNetwork(state_dim, action_dim, hidden_dim)
        self.policy_optimizer = optim.Adam(self.policy_net.parameters(), lr=
            lr_policy)
        if use_baseline:
            self.value_net = ValueNetwork(state_dim, hidden_dim)
            self.value_optimizer = optim.Adam(self.value_net.parameters(), lr=
                lr_value)
        self.device = torch.device("cuda" if torch.cuda.is_available() else "
            cpu")
        self.policy_net.to(self.device)
        if use_baseline:
            self.value_net.to(self.device)

    % Update baseline using TD(0) at each step
    def _update_baseline_td0(self, state, reward, next_state, done):
        state_tensor = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        next_state_tensor = torch.FloatTensor(next_state).unsqueeze(0).to(self.
            device)
        with torch.no_grad():
            next_value = 0 if done else self.value_net(next_state_tensor)
        reward_tensor = torch.tensor([reward], device=self.device, dtype=torch.
            float32)
        td_target = reward_tensor + self.gamma * next_value
        current_value = self.value_net(state_tensor)
        value_loss = F.mse_loss(current_value, td_target)
        self.value_optimizer.zero_grad()
        value_loss.backward()
        self.value_optimizer.step()

    % Policy is updated after the episode using the complete trajectory
    def _update_policy(self, states, rewards, log_probs):
        returns = self.calculate_returns(rewards)
        states = torch.FloatTensor(np.array(states)).to(self.device)
        if self.use_baseline:
            with torch.no_grad():
                state_values = self.value_net(states).squeeze(-1)
            advantages = returns - state_values
        else:
            advantages = returns
        if len(advantages) > 1:
            advantages = (advantages - advantages.mean()) / (advantages.std() +
                1e-8)
        policy_loss = 0
        for log_prob, advantage in zip(log_probs, advantages):
            policy_loss -= log_prob * advantage
        self.policy_optimizer.zero_grad()
        policy_loss.backward()
        self.policy_optimizer.step()

    % Helper: compute Monte Carlo returns
    def calculate_returns(self, rewards):
        returns = []
```

```python
        R = 0
        for r in reversed(rewards):
            R = r + self.gamma * R
            returns.insert(0, R)
        return torch.tensor(returns, device=self.device, dtype=torch.float32)

    def select_action(self, state):
        state = torch.FloatTensor(state).to(self.device)
        dist = self.policy_net(state)
        action = dist.sample()
        return action.item(), dist.log_prob(action)

    def train(self, env, num_episodes, max_steps=500):
        episode_returns = []
        for episode in range(num_episodes):
            state, _ = env.reset()
            states, rewards, log_probs = [], [], []
            episode_return = 0
            for step in range(max_steps):
                action, log_prob = self.select_action(state)
                next_state, reward, done, truncated, _ = env.step(action)
                states.append(state)
                rewards.append(reward)
                log_probs.append(log_prob)
                if self.use_baseline:
                    self._update_baseline_td0(state, reward, next_state, done or truncated)
                state = next_state
                episode_return += reward
                if done or truncated:
                    break
            self._update_policy(states, rewards, log_probs)
            episode_returns.append(episode_return)
            print(f"Episode {episode+1}, Return: {episode_return}")
        return episode_returns
```

Listing 3: REINFORCE with TD(0) Baseline

4 Experimental Setup

Experiments are run with $\gamma = 0.99$ and averaged over 5 random seeds to account for stochasticity. Hyperparameters are tuned by minimizing the regret in each experiment. The episodic return as a function of episode number is plotted, showing both the mean and variance across the seeds.

5 Results

5.1 Dueling-DQN Results

Figure 1 shows the mean episodic return of Type-1 and Type-2 variants on CartPole-v1, while Figure 2 shows the corresponding results on Acrobot-v1.

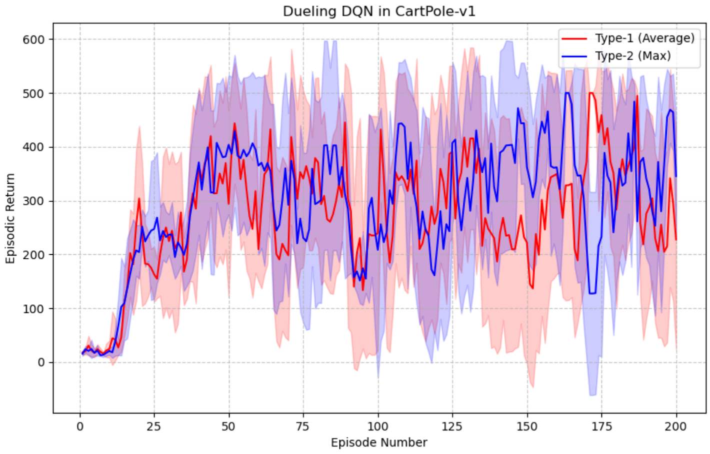

Figure 1: Comparison of Type-1 and Type-2 Dueling-DQN on CartPole-v1.

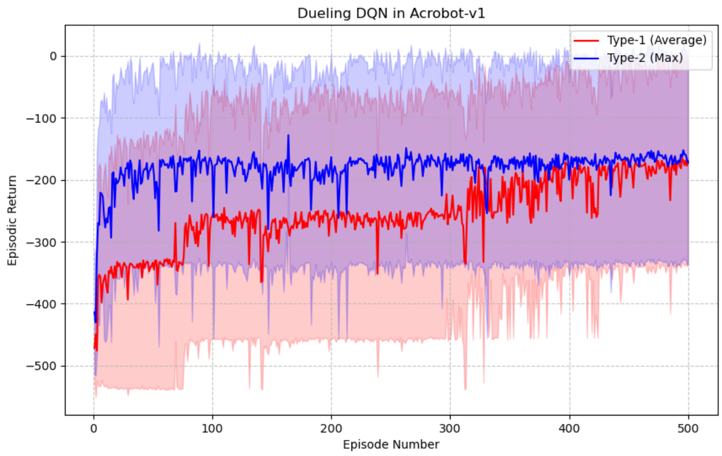

Figure 2: Comparison of Type-1 and Type-2 Dueling-DQN on Acrobot-v1.

5.2 Monte-Carlo REINFORCE Results

Figures 3 and 4 display the performance of REINFORCE with and without a baseline (using TD(0)) on CartPole-v1 and Acrobot-v1, respectively.

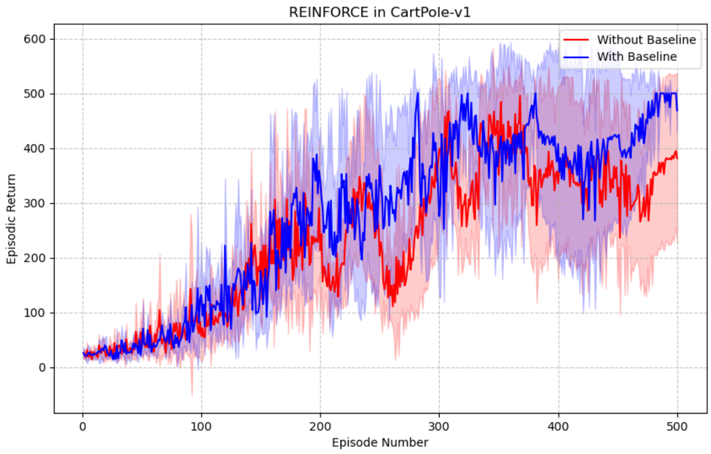

Figure 3: Monte-Carlo REINFORCE on CartPole-v1: With and Without Baseline.

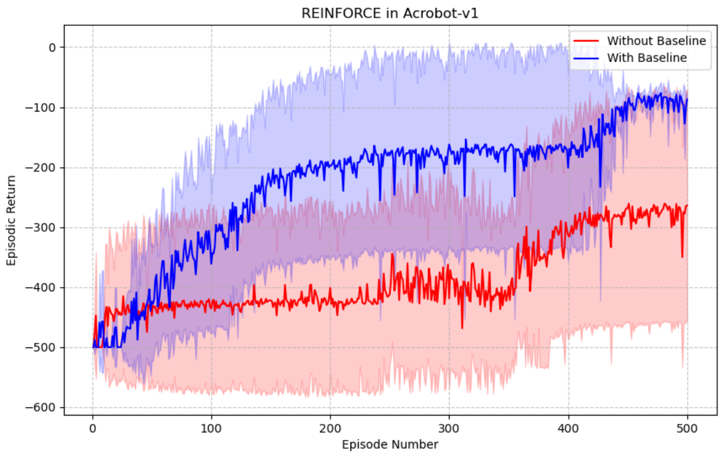

Figure 4: Monte-Carlo REINFORCE on Acrobot-v1: With and Without Baseline.

6 Hyperparameter Tuning

We tested various hyperparameter settings. The top setting for each algorithm were:

6.1 DQN Duelling - Cartpole-v1: Average

The hyperparameters used for DQN Duelling on Cartpole-v1 (Average) are:

- **Batch Size**
- **Epsilon decay**
- **Hidden Dimension**
- **Learning rate**
- **Target Update**

Figure 5 shows the parallel coordinates plot for the average performance.

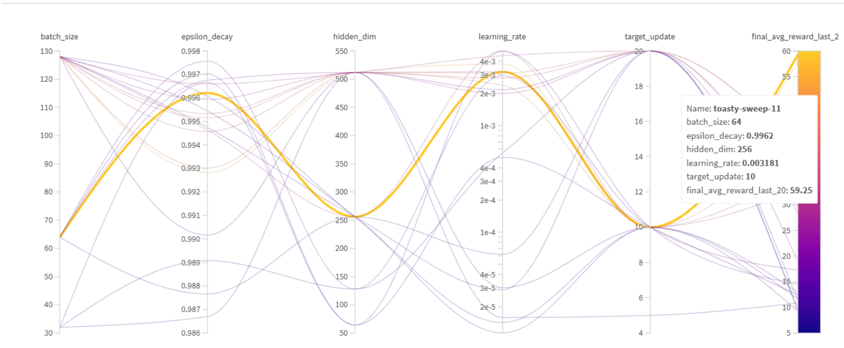

Figure 5: DQN Duelling - Cartpole-v1 (Average): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Batch Size:** 64
- **Epsilon decay:** 0.9962
- **Hidden Dimension:** 256
- **Learning rate:** 0.003181
- **Target Update:** 10

6.2 DQN Duelling - Cartpole-v1: Maximum

The hyperparameters used for DQN Duelling on Cartpole-v1 (Maximum) are:

- **Batch Size**
- **Epsilon decay**
- **Hidden Dimension**
- **Learning rate**
- **Target Update**

Figure 6 shows the parallel coordinates plot for the maximum performance.

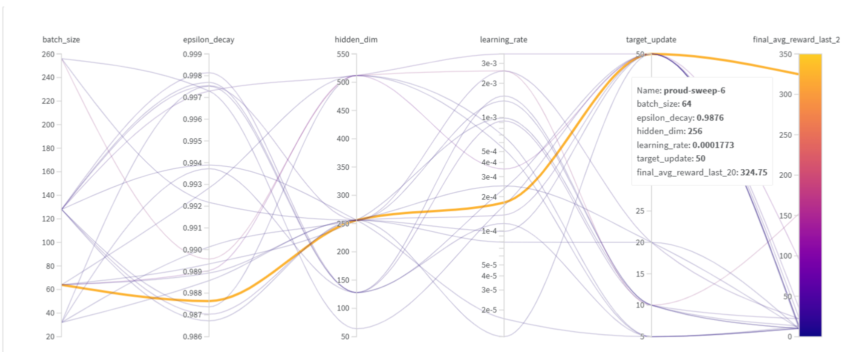

Figure 6: DQN Duelling - Cartpole-v1 (Maximum): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Batch Size:** 64
- **Epsilon decay:** 0.9876
- **Hidden Dimension:** 256
- **Learning rate:** 0.0001773
- **Target Update:** 50

6.3 DQN Duelling - Acrobot-v1: Average

The hyperparameters used for DQN Duelling on Acrobot-v1 (Average) are:

- **Batch Size**
- **Epsilon decay**
- **Hidden Dimension**
- **Learning rate**
- **Target Update**

Figure 7 shows the parallel coordinates plot for the average performance.

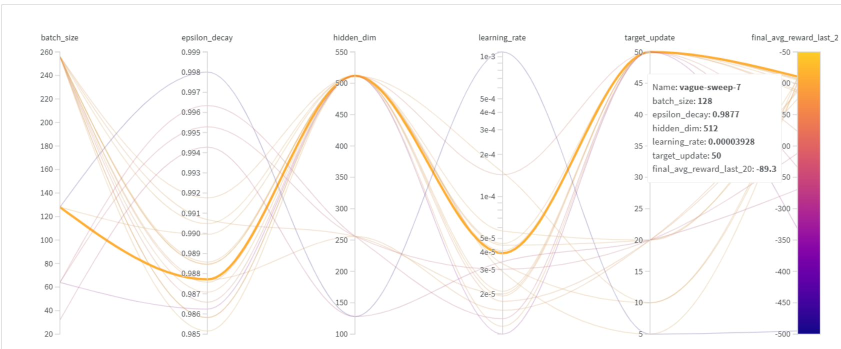

Figure 7: DQN Duelling - Acrobot-v1 (Average): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Batch Size:** 128
- **Epsilon decay:** 0.9877
- **Hidden Dimension:** 512
- **Learning rate:** 0.00003928
- **Target Update:** 50

6.4 DQN Duelling - Acrobot-v1: Maximum

The hyperparameters used for DQN Duelling on Acrobot-v1 (Maximum) are:

- **Batch Size**
- **Epsilon decay**

- **Hidden Dimension**
- **Learning rate**
- **Target Update**

Figure 8 shows the parallel coordinates plot for the maximum performance.

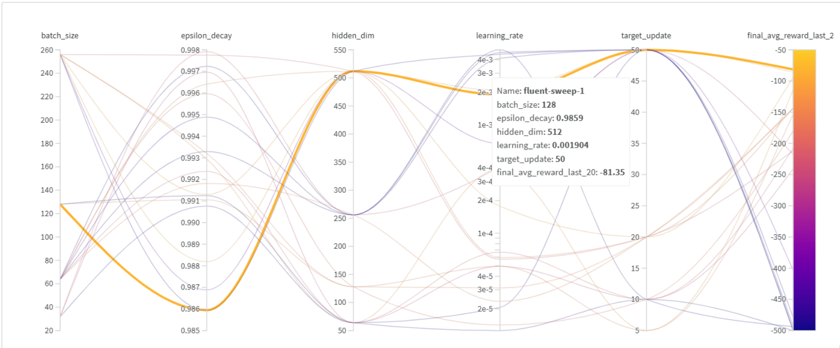

Figure 8: DQN Duelling - Acrobot-v1 (Maximum): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Batch Size:** 128
- **Epsilon decay:** 0.9859
- **Hidden Dimension:** 512
- **Learning rate:** 0.001904
- **Target Update:** 50

6.5 REINFORCE - Cartpole-v1 (No Baseline)

The hyperparameters used for REINFORCE on Cartpole-v1 without a baseline are:

- **Hidden Dimension**
- **Policy Network Learning Rate**

Figure 9 shows the parallel coordinates plot for the no-baseline configuration.

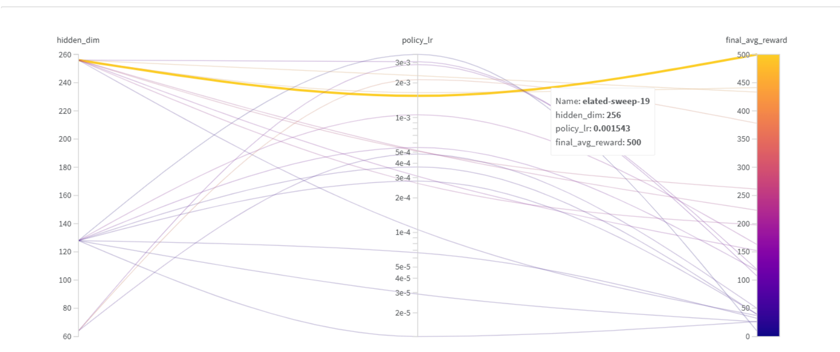

Figure 9: REINFORCE - Cartpole-v1 (No Baseline): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Hidden Dimension:** 256
- **Policy Learning Rate:** 0.001543

6.6 REINFORCE - Cartpole-v1 (Baseline)

The hyperparameters used for REINFORCE on Cartpole-v1 with a baseline are:

- **Hidden Dimension**
- **Policy Learning Rate**
- **Value Network Learning Rate**

Figure 10 shows the parallel coordinates plot for the baseline configuration.

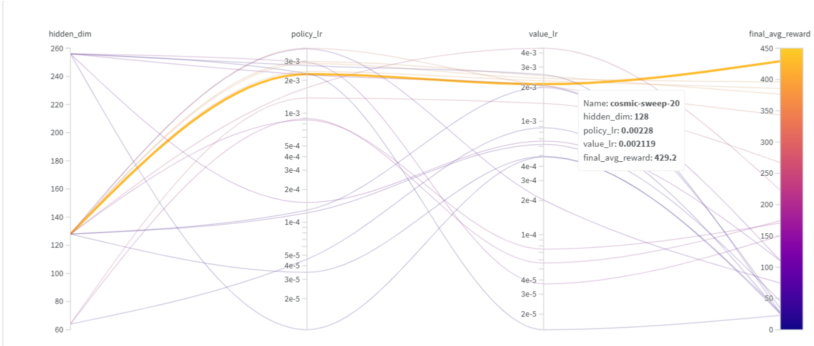

Figure 10: REINFORCE - Cartpole-v1 (Baseline): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Hidden Dimension:** 128
- **Policy Learning Rate:** 0.00228
- **Value Network Learning Rate:** 0.002119

6.7 REINFORCE - Acrobot-v1 (No Baseline)

The hyperparameters used for REINFORCE on Acrobot-v1 without a baseline are:

- **Hidden Dimension**
- **Policy Network Learning Rate**

Figure 11 shows the parallel coordinates plot for the no-baseline configuration.

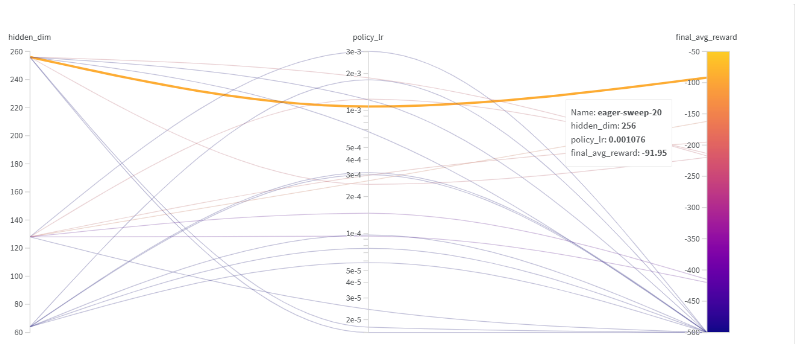

Figure 11: REINFORCE - Acrobot-v1 (No Baseline): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Hidden Dimension:** 256
- **Policy Learning Rate:** 0.001076

6.8 REINFORCE - Acrobot-v1 (Baseline)

The hyperparameters used for REINFORCE on Acrobot-v1 with a baseline are:

- **Hidden Dimension**
- **Policy Learning Rate**
- **Value Network Learning Rate**

Figure 12 shows the parallel coordinates plot for the baseline configuration.

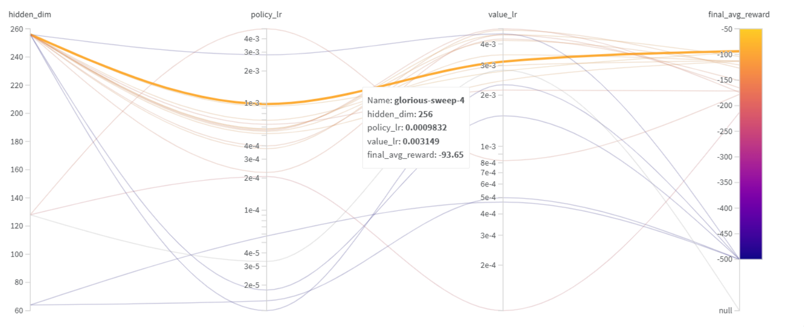

Figure 12: REINFORCE - Acrobot-v1 (Baseline): Parallel coordinates plot.

The best hyperparameter configuration for this experiment was:

- **Hidden Dimension:** 256
- **Policy Learning Rate:** 0.0009832
- **Value Network Learning Rate:** 0.003149

7 Conclusions and Discussions

The experimental results indicate that:

- For Dueling-DQN, the aggregation method (mean vs. max) significantly affects the learning dynamics.
- Max aggregation tends to converge much earlier than average aggregation. But, average aggregation converges as well in 500 episodes for acrobot-v1.
- In Monte-Carlo REINFORCE, incorporating a baseline updated via TD(0) reduces the gradient variance and leads to more stable learning. It can be observed from the graphs how using baseline is much better especially for acrobot-v1.
- Performance trends are consistent across both CartPole-v1 and Acrobot-v1.

8 Code Repository

The complete source code is available on GitHub at: `https://github.com/DA6400-RL-JanMay2025/programming-assignment-02-ch21b053_me21b171`.