CS6700: Reinforcement Learning

Programming Assignment - II Report

Siddhagavali Shital [ME20B166]

Srinivas Chowdary [ME20B174]

GitHub Repository Link

April 7, 2024

Table of Contents

1 Algorithem 1 - Dueling-DQN

Dueling DQN is an extension of the DQN algorithm, designed to improve learning efficiencyby decomposing the Q-value function into two separate streams: one estimating the state value and the other estimating the advantage of each action.

1.1 Code snippets and Outline :

1.1.1 Dueling DQN Network Architecture:

The ‘DuelingDQN‘ class defines a neural network for reinforcement learning with a dueling architecture, which separately estimates state values and advantages for actions. The network is part of the agent’s decision-making apparatus in the main code, guiding action selection to maximize cumulative rewards.

```python
class DuelingDQN(nn.Module):
    def __init__(self, lr, n_actions, name, input_dims, chkpt_dir):
        super(DuelingDQN, self).__init__()
        self.checkpoint_dir = chkpt_dir
        self.checkpoint_file = os.path.join(self.checkpoint_dir, name)

        self.fc1 = nn.Linear(*input_dims, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)

        self.V = nn.Linear(128, 1)
        self.A = nn.Linear(128, n_actions)

        self.optimizer = optim.Adam(self.parameters(), lr=lr)
        self.loss = nn.MSELoss()
        self.device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
        self.to(self.device)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))

        V = self.V(x)
        A = self.A(x)

        return V, A
```

Listing 1: Dueling DQN Network

1.1.2 Replay Buffer:

The ‘ReplayBuffer‘ class manages a cyclic buffer of a fixed size to store and retrieve experiences, enabling the reinforcement learning agent to learn from past transitions in a randomized manner, which is critical for stabilizing training by breaking correlation between consecutive experiences.

```python
class ReplayBuffer():
    def __init__(self, max_size, input_shape, n_actions, seed):
        self.mem_size = max_size
        self.---  # Defining more params

    def store_transition(self, state, action, reward, state_, done):
        index = self.mem_cntr % self.mem_size
        self.state_memory[index] = state
        self.new_state_memory[index] = state_
        self.action_memory[index] = action
        self.reward_memory[index] = reward
        self.terminal_memory[index] = done
        self.mem_cntr += 1

    def sample_buffer(self, batch_size):
        max_mem = min(self.mem_cntr, self.mem_size)
        batch = np.random.choice(max_mem, batch_size, replace=False)

        states = self.state_memory[batch]
        actions = self.action_memory[batch]
        rewards = self.reward_memory[batch]
        states_ = self.new_state_memory[batch]
        terminal = self.terminal_memory[batch]

        return states, actions, rewards, states_, terminal
```

Listing 2: Replay Buffer

1.1.3 DQN Agent:

The DQNAgent class encapsulates the functionality of a Dueling Deep Q-Network (DDQN) agent for reinforcement learning tasks. The class employs two neural network models, q_eval for evaluating the current policy and q_next for estimating future values, facilitating the separation of value and advantage streams as per the Dueling DQN architecture. Critical methods include:

1. choose_action: Determines the next action based on the current state, using an epsilon-greedy policy.

```
        def choose_action(self, observation):
            observations = np.array([observation])
            state = torch.tensor(observations).float().to(self.q_eval.device)
            if np.random.random() > self.epsilon:
                _, advantage = self.q_eval.forward(state)
                action = torch.argmax(advantage).item()
            else:
                action = np.random.choice(self.action_space)

            return action
```

Listing 3: Choosing Action using Epsilon Greedy

2. store_transition: Saves experience tuples to the replay buffer.

```
        def store_transition(self, state, action, reward, state_, done):
            self.memory.store_transition(state, action, reward, state_, done)
```

Listing 4: store transition

3. replace_target_network: Synchronizes the weights of the q_next network with q_eval periodically to stabilize learning.

```
        def replace_target_network(self):
            if self.learn_step_counter % self.replace_target_cnt == 0:
              self.q_next.load_state_dict(self.q_eval.state_dict())
```

Listing 5: Replace Traget Network

4. learn: Conducts a learning step, sampling a batch of experiences, calculating the loss, and updating the q_eval network's weights for both **Type-1** and **Type-2** equation according to given type.

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \frac{1}{|A|} \sum_{a' \in |A|} A(s,a';\theta) \right) \quad \text{(Type-1)}$$

Where $Q(s,a;\theta)$ represents the dueling Q-function with parameters θ.

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a' \in |A|} A(s,a';\theta) \right) \quad \text{(Type-2)}$$

```python
        def learn(self):
            if self.memory.mem_cntr < self.batch_size:
                ------ # Some code

            V_s, A_s = self.q_eval.forward(states)
            V_s_, A_s_ = self.q_next.forward(states_)

            V_s_eval, A_s_eval = self.q_eval.forward(states_)

            if self.update_type == 'Type-1':
              q_pred = torch.add(V_s, (A_s - A_s.mean(dim=1, keepdim=True)))[
                  indices, actions]
              q_next = torch.add(V_s_, (A_s_ - A_s_.mean(dim=1, keepdim=True)))
              q_eval = torch.add(V_s_eval, (A_s_eval - A_s_eval.mean(dim=1, keepdim
                  =True)))
            elif self.update_type == 'Type-2':
              q_pred = torch.add(V_s, (A_s - torch.max(A_s, dim=1, keepdim=True)
                  [0]))[indices, actions]
              q_next = torch.add(V_s_, (A_s_ - torch.max(A_s_, dim=1, keepdim=True)
                  [0]))
              q_eval = torch.add(V_s_eval, (A_s_eval - torch.max(A_s_eval, dim=1,
                  keepdim=True)[0]))

            max_actions = torch.argmax(q_eval, dim=1)
            q_next[dones] = 0.0
            q_target = rewards + self.gamma*q_next[indices, max_actions]

            loss = self.q_eval.loss(q_target, q_pred).to(self.q_eval.device)
            loss.backward()
            self.q_eval.optimizer.step()
            self.learn_step_counter += 1
```

Listing 6: Learning Function

1.1.4 Training Dueling DQN Agent:

The `train_dueling_dqn` function trains a Dueling DQN agent across multiple episodes, managing environment interactions, experience replay, and policy learning, ultimately aiming to maximize cumulative rewards.

```python
def train_dueling_dqn(agent, env_name, episodes=1000, max_t=500, batch_size=64, gamma
    =0.99,
                      epsilon_decay=0.995, epsilon_min=0.01, target_update=10):
    env = gym.make(env_name)
    all_scores = []

    for seed in range(5):
        scores = []

        for i_episode in range(episodes):
            state = env.reset()
            env.seed(seed)
            done = False
            score = 0

            scores_window = deque(maxlen=100)
            for t in range(max_t):
                action = agent.choose_action(state)
                next_state, reward, done, _ = env.step(action)
                agent.store_transition(state, action, reward, next_state, done)
                agent.learn()
                state = next_state
                score += reward
                if done:
                    break

            scores.append(score)
            scores_window.append(score)

            agent.epsilon = max(epsilon_min, agent.epsilon * epsilon_decay)

            # ---

        all_scores.append(scores)

    return all_scores
```

Listing 7: Training Dueling DQN Agent

1.2 Hyperparameter tuning:

- Modified the architecture of the Deep Q-Network (DQN) by increasing the number of hidden layers from 1 to 3.
- This resulted in a notable improvement in the performance of the Acrobot Type 1 environment, with the agent reaching the maximum threshold within 200 episodes, compared to the previous 300 episodes.
- The modification also demonstrated a positive impact on Acrobot Type 2, indicating enhanced learning capabilities.
- Maintaining a learning rate of 0.0005 showed improvement in Type 2 for both environments to reach the threshold reward, while using 0.001 for Type 1 in both Acrobot and Cartpole environments.
- Experimentation with different batch sizes revealed that using a batch size of 64 produced the most suitable results. Increasing the batch size to 128 led to a significant increase in variance.

1.3 Acrobot-v1 Environment :

Here is the reward plot for the Acrobot-V1 environment 1:

1.3.1 Inference:

- **Type-1 Dueling DQN** (in blue) starts with a similar learning curve to Type-2 but demonstrates a smoother and higher ascent in terms of average score over episodes, indicating a more stable learning process. It approaches the desired threshold score, suggesting better policy learning.
- **Type-2 Dueling DQN** (in red) initially improves rapidly but exhibits greater variance in performance and does not converge to the -100 threshold. This might be due to the Type-2 update rule's potentially higher sensitivity to variations in the Q-value estimates.

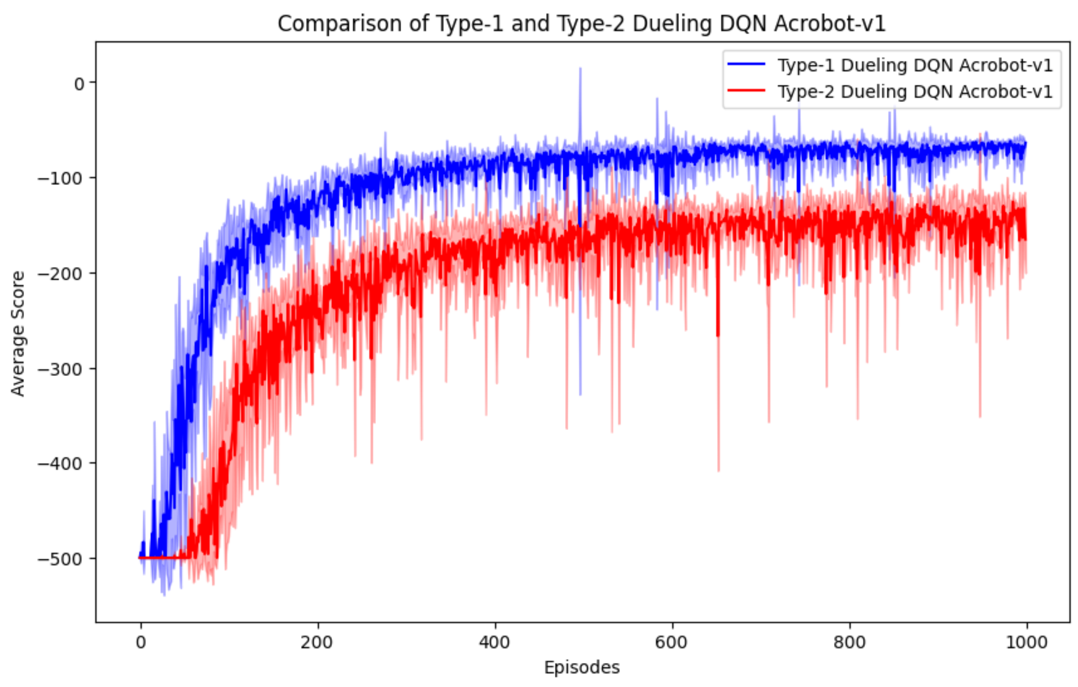

Figure 1: Dueling-DQN in Acrobot-V1

- The Type-1 update rule seems to provide more consistent performance and potentially better policy by the end of the training episodes, as indicated by the higher average score and approach towards the threshold.

1.4 CartPole-v1 Environment :

Here is the reward plot for the CartPole-V1 environment 2:

1.4.1 Inference:

- **Type-1 Dueling DQN** (in blue) shows a trend of improvement with considerable variance throughout the training episodes. This variant reaches the environment's threshold reward, indicating effective learning and adaptation to the goal of balancing the pole.
- **Type-2 Dueling DQN** (in red) displays significant variance and, similar to Type-1, reaches the threshold reward of the environment. However, the performance is more oscillatory, which might suggest that while the policy is effective, it might not be as robust as that learned by Type-1.
- Despite reaching the threshold reward, both Type-1 and Type-2 exhibit a level of instability, which could be improved with further tuning of the learning algorithm and exploration strategy.
- Throughout the 1000 episodes, both algorithms exhibit very high deviations across the 5 runs, and their means also fluctuate significantly. This suggests that these algorithms may not be optimal for use in the CartPole environment, as they have a lower standard deviation for the Acrobot environment.

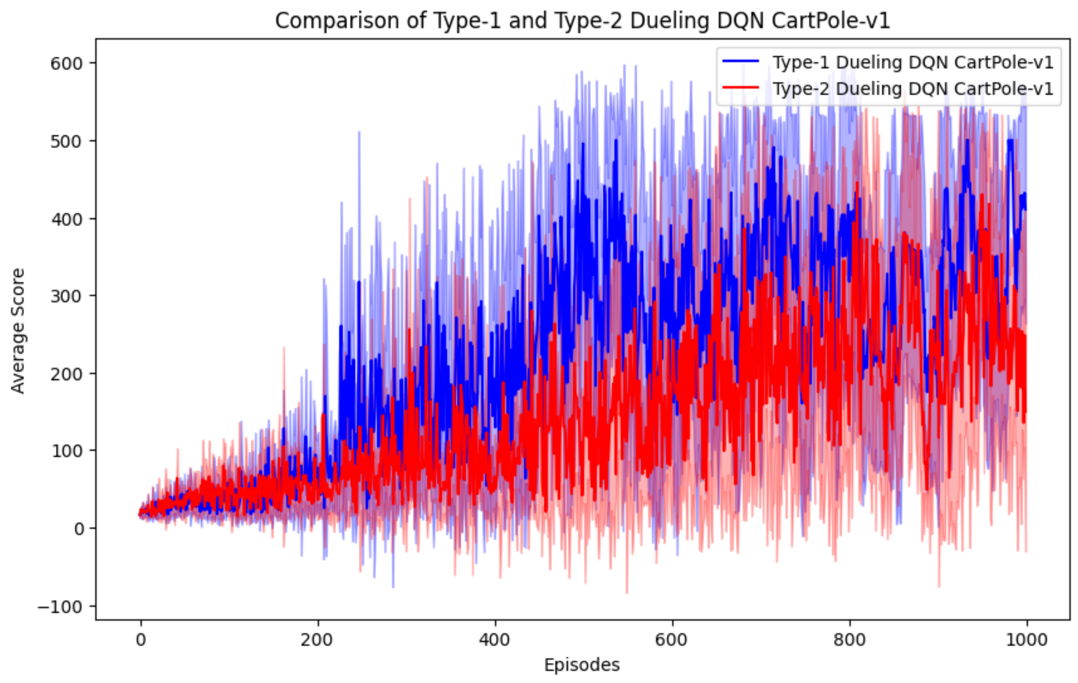

Figure 2: Dueling-DQN in CartPole-V1

2 Algorithem 2 - Monte Carlo REINFORCE

2.1 Hyperparameter tuning.

For both models, with and without baseline, the policy network has 2 hidden layers. To develop an effective algorithm, we executed the model multiple times, varying the number of nodes in both layers. We experimented with different values for L1 and L2, specifically [16, 32, 64, 128], resulting in 16 runs for each MC Reinforce configuration across all environments, totaling 64 runs.

The code used to run the hyperparameter tuning loops:

```
num_runs = 5
num_episodes = 1000

layer1=[16,32,64,128]
layer2=[16,32,64,128]
values=[]
for number1 in range(0,len(layer1)):
  l1=layer1[number1]
  for number2 in range(0,len(layer2)):
    l2=layer2[number2]
    average_score = []
    for run in range(num_runs):
        print(f"{l1,l2} run - {run + 1}")
        env = gym.make(env_name)
        env.reset()
        agent = Agent1(alpha=0.001, input_size=[env.observation_space.shape[0]], g=0.99,
             n=env.action_space.n, l1=l1, l2=l2)
        score_history = []
```

Here is the code for policy's network:

```
class net1(nn.Module):
    def __init__(self, alpha, input, l1, l2, n):
        super(net1, self).__init__()
        self.input = input[0]
        self.l1 = l1
        self.l2 = l2
        self.n = n
        self.fc1 = nn.Linear(self.input, self.l1)
        self.fc2 = nn.Linear(self.l1, self.l2)
        self.fc3 = nn.Linear(self.l2, self.n)
        self.optimizer = optim.Adam(self.parameters(), lr=alpha)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

After running these experiments, we selected the L1 and L2 values that yielded higher rewards quickly while maintaining a more stable trend. Here are the best values for each experiment:

Table 1: Experiment configurations

Experiment	L1	L2
CartPole	64	32
CartPole with baseline	32	64
Acrobot	64	32
Acrobot with baseline	32	128

Even with the same initialization, each network gives different results for each run. It might be better to run 10/20 runs and take the average of these for a more reliable outcome.

2.2 Code

Here is the code used for learning without Baseline.

```
    def learn(self):
        self.policy.optimizer.zero_grad()
        G=np.zeros_like(self.memory_r, dtype=np.float64)
        for t in range(len(self.memory_r)):
            sum_g=0
            discount=1
            for k in range(t, len(self.memory_r)):
                sum_g += self.memory_r[k] * discount
                discount *= self.g
            G[t]=sum_g
        mean=np.mean(G)
        std=np.std(G) if np.std(G) > 0 else 1
        G=(G - mean) / std
        G=T.tensor(G, dtype=T.float)
        loss=0
        for g_, logprob in zip(G, self.memory_a):
            loss += -g_* logprob
        loss.backward()
        self.policy.optimizer.step()
        self.memory_r=[]
        self.memory_a=[]
```

$\theta = \theta + \alpha \frac{G_t \nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$

For learning with Baseline we made a few modifications:

```
        def learn(self):
        self.policy.optimizer.zero_grad()
        G = np.zeros_like(self.memory_r, dtype=np.float64)
        for t in range(len(self.memory_r)):
            G_sum = 0
            discount = 1
            for k in range(t, len(self.memory_r)):
                G_sum += self.memory_r[k] * discount
                discount *= self.g
            G[t] = G_sum

        self.baseline = np.mean(G)
        G = G - self.baseline
        G = T.tensor(G, dtype=T.float)

        loss = 0
        for g_, logprob in zip(G, self.memory_a):
            loss += -g_ * logprob

        loss.backward()
        self.policy.optimizer.step()

        self.memory_a = []
        self.memory_r = []
```

$\theta = \theta + \alpha \frac{(G_t - V(S_t;\Phi))\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$

V is taken to be the mean of G.

2.3 CartPole

2.3.1 Inferences

In figure 3, Type 1 is MC reinforce without baseline.

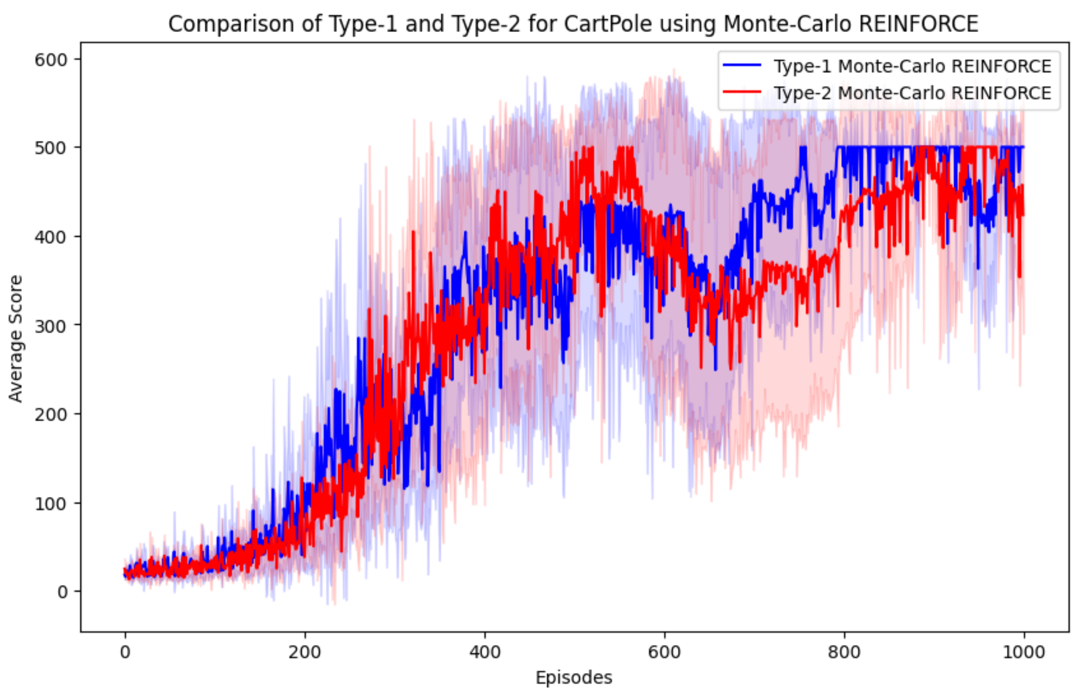

Figure 3: Cartpole-MC REINFORCE

Both algorithms converge with similar trends; In MC REINFORCE without a baseline, the update rule for the policy parameters relies solely on the observed returns without any adjustment for the expected return. The initial increase in performance can be attributed to the algorithm learning favorable actions through exploration and exploitation of the environment. The fluctuations around a score of 350 suggest that the algorithm struggles to consistently estimate the gradient accurately. This could be due to high variance in the returns, leading to unstable updates of the policy parameters. The eventual rise to a score of 500 indicates that the algorithm manages to find a policy that achieves the task's objective, albeit with high variance in the learning process.

MC REINFORCE with a baseline introduces a baseline term to reduce the variance of the policy gradient estimates. The steady increase in performance can be attributed to the baseline term stabilizing the updates of the policy parameters by reducing the variance of the estimated gradients. Reaching a score of 500 demonstrates that the algorithm effectively learns to maximize the expected return with the aid of the baseline, achieving similar performance to the algorithm without a baseline. The temporary decrease to 300 before rising again suggests that despite the baseline's stabilizing effect, the algorithm may still encounter challenges such as suboptimal exploration or local optima. However, it manages to recover and continue improving towards the optimal policy.

There is a lower standard deviation during the initial rise and higher variance across the 5 runs after reaching higher rewards, and towards the end, the deviation decreases. This could be because for each run in between, there is a higher fluctuation as there is more exploration, leading to different rewards.

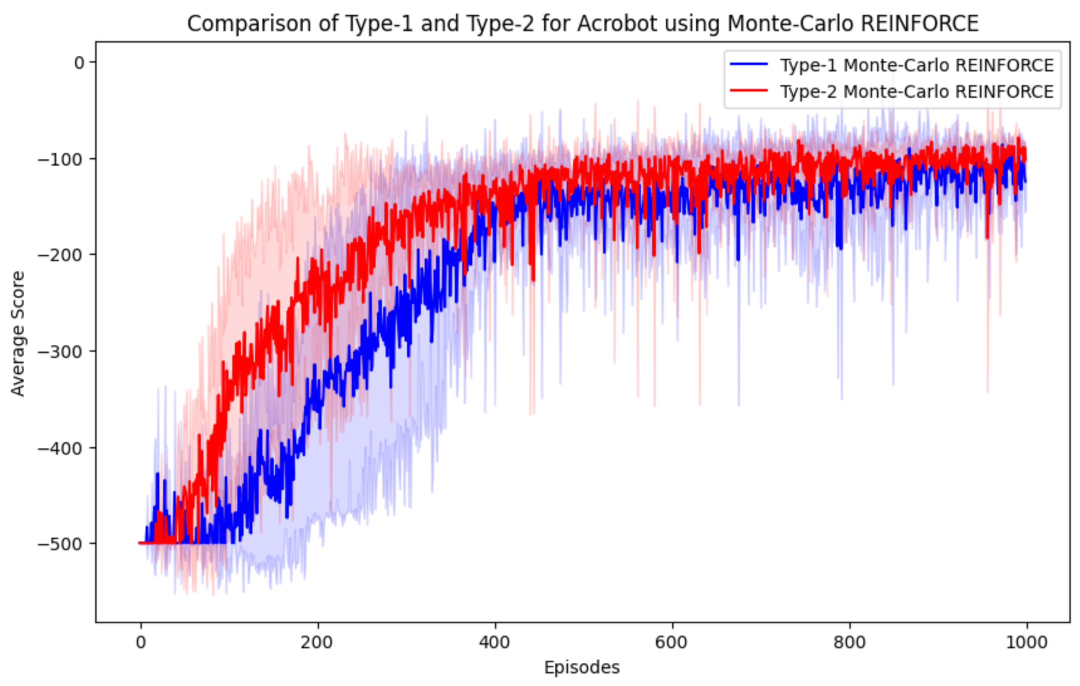

Figure 4: Acrobot-MC REINFORCE

2.4 Acrobot

2.4.1 Inferences

In figure 4 Type 1 is MC reinforce without baseline. We can see that the algorithm with baseline reaches convergence much quicker than the algorithm without baseline and also has a higher consistent reward after converging.

The algorithm with a baseline converges faster compared to the one without. This is because the baseline effectively reduces the variance in the gradient estimates, leading to more stable updates of the policy parameters. Mathematically, the baseline acts as a reference point that helps estimate the advantage function better, which in turn leads to more accurate gradient estimates. As a result, the policy parameters converge towards the optimal policy more efficiently.

After convergence, the algorithm with a baseline achieves a consistently higher reward compared to the algorithm without. This indicates that the learned policy is more effective and robust in achieving the task's objective. The presence of the baseline reduces the likelihood of large fluctuations in the policy's performance, resulting in a more reliable and stable policy.

It is intriguing to note that for both algorithms, there is a higher standard deviation during the initial rise and lower variance across the 5 runs after converging, showing that although initially each run explores differently, later on they converge to an optimal policy.

2.5 Conclusion

In both environments, both algorithms converge above the threshold value and maintain it towards the end of 1000 episodes. The algorithm with a baseline performs slightly better.

However, for a more comprehensive study, we need to observe the performance of each algorithm over a larger number of episodes, perhaps 2000 or 5000. Additionally, running these algorithms for 10 to 20 iterations and calculating their averages and variances would provide a more robust analysis.