Programming Assignment 2
CS6700

Dueling-DQN and Monte-Carlo REINFORCE

April 7, 2024

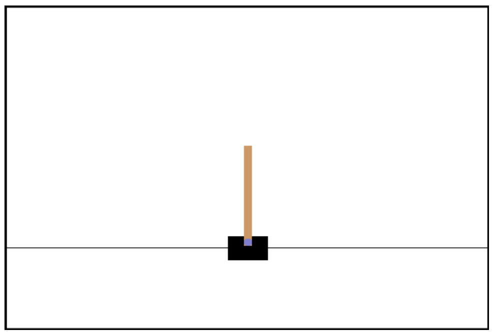
Balakumar R
ME20B043

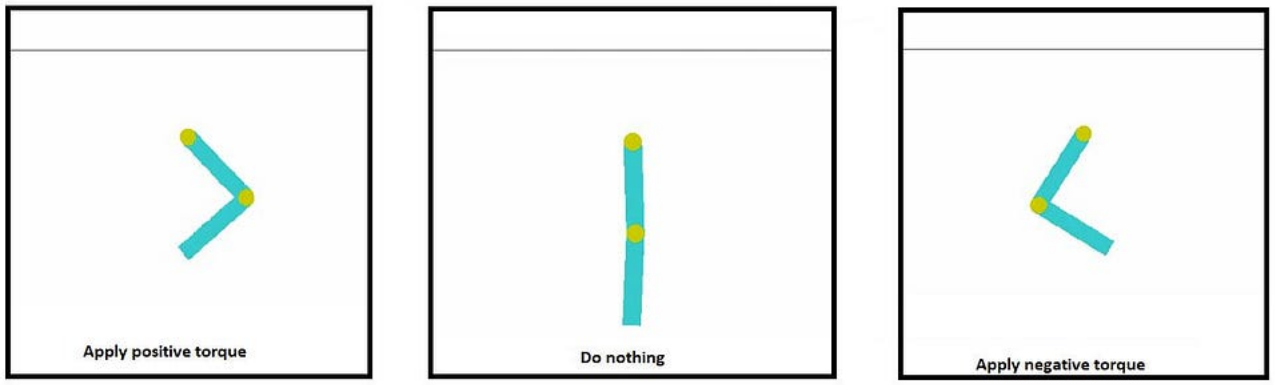
Girish Madhavan V
ME20B072

Project Code Repo

Contents

1 Environment





Gymmnasium Environments are:

- **Acrobot-v1:** The system consists of two links connected linearly to form a chain, with one end of the chain fixed. The joint between the two links is actuated. The goal is to apply torques on the actuated joint to swing the free end of the linear chain above a given height while starting from the initial state of hanging downwards.
- **CartPole-v1:** A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum is placed upright on the cart and the goal is to balance the pole by applying forces in the left and right direction on the cart.

1.1 Algorithms

The two algorithms which are used are Dueling-DQN and Monte-Carlo REINFORCE:

1.1.1 Dueling-DQN

Dueling-DQN aims to enhance the stability and efficiency of the learning process by explicitly separating the estimation of state values and action advantages, enabling the agent to better understand the underlying structure of the environment. This separation allows the agent to focus its attention on states that are most relevant for learning, thereby improving sample efficiency and accelerating convergence.

Type-1 update equation for the dueling network is:

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

Type-2 update equation for the dueling network is:

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

1.1.2 Monte-Carlo REINFORCE

Monte Carlo REINFORCE learns policies by directly optimizing the expected return through gradient ascent. This direct policy optimization enables Monte Carlo REINFORCE to handle both discrete and continuous action spaces seamlessly, making it applicable to a wide range of problems, including robotics, autonomous driving, and game playing.

w/o Baseline update equation of its policy parameter θ is given by

$$\theta = \theta + \alpha G_t \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{3}$$

w/ Baseline update equation of its policy parameter θ is given by

$$\theta = \theta + \alpha(G_t - V(S_t;\Phi)) \frac{\nabla_\theta \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \tag{4}$$

The baseline $V(\cdot;\Phi)$ is updated by TD(0) method.

NOTE :

- **Acrobot-v1**

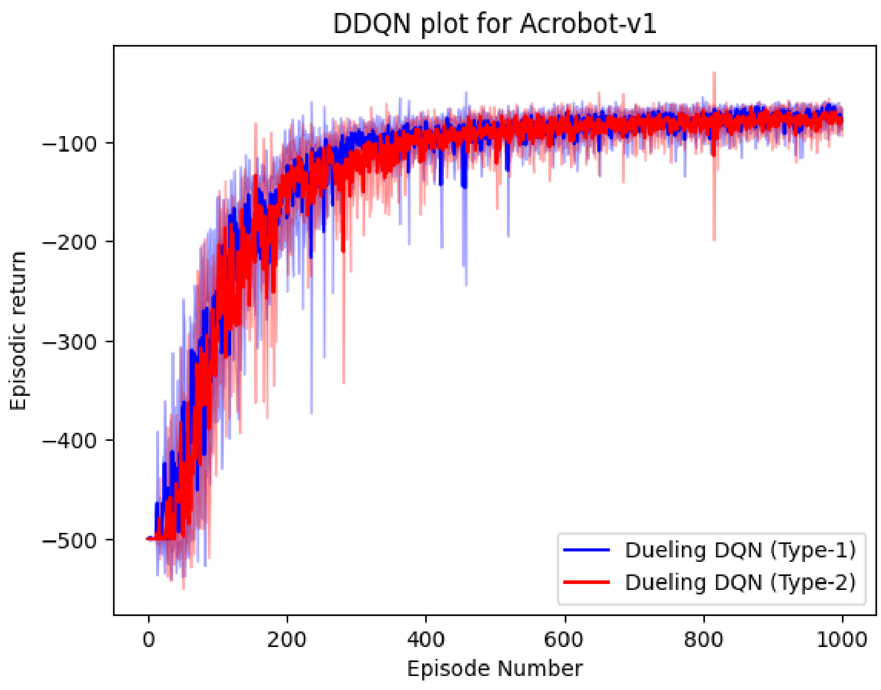
  The goal is to have the free end reach a designated target height in as few steps as possible, and as such all steps that do not reach the goal incur a reward of -1. Achieving the target height results in termination with a reward of 0. The reward threshold is -100. Truncation if Episode length is greater than 500.

- **CartPole-v1**

  Since the goal is to keep the pole upright for as long as possible, a reward of +1 for every step taken, including the termination step, is allotted. The threshold for rewards is 500 for v1. Truncation if Episode length is greater than 500.

2 Results

2.1 `DDQN - Acrobot-v1`



best model, mean reward $\implies$ Type-1 : -131.9398, Type-2 : -138.2376

Hyperparameter : $\alpha = 0.0005, \gamma = 0.99,$ `BUFFER_SIZE`$:10^5,$ `BATCH_SIZE` $= 128$

The hyperparameters are tuned to minimize regret and in this case it maximizes the return($\approx$ -100 is threshold) and is stable in that position for many episodes. The effect of hyperparameters are:

- `BUFFER_SIZE`**:** A larger buffer size allows the agent to store more experiences, potentially improving sample efficiency and aiding in the decorrelation of samples.
- `BATCH_SIZE`**:** The batch size determines the number of experiences sampled from the replay buffer for each training iteration and a larger batch size can lead to more stable updates.
- α: Learning rate, which determines the step size taken during gradient descent optimization.
- `UPDATE_EVERY`**:** Defines how often the target network (Q target) is updated with the parameters of the local network. Updating the target network less frequently can reduce the computational burden but may also lead to slower convergence.

Inference & Results:

Based on the above graph, it is evident that the Dueling DQN (Type-1) outperforms Dueling DQN (Type-2) in terms of episodic return when trained on the Acrobot-v1 environment. This is because the Type-1 DQN achieves a higher average episodic return over the course of training.

- **Advantage Estimation:** Both Type-1 and Type-2 Dueling DQN utilize advantage estimation to improve learning efficiency. The goal is to isolate the advantage (goodness) of each action relative to the average or best action in a given state. This helps the agent focus on learning which actions are better than others, rather than just the overall value of a state.

  Type-1 Dueling DQN (Average Advantage):

  – Subtracts the average value of all actions $(A(s, .))$ from each individual action's Q-value $Q(s, a)$.

  – Focuses on learning the relative advantage of each action compared to the average.

Type-2 Dueling DQN (Max Advantage):

  - Takes the maximum advantage across all actions ($\max_{a' \in |A|}$ A(s, a')) and subtracts it from each individual action's Q-value.
  - Focuses on learning the relative advantage of each action compared to the best action.

- **Success Rate:** A similar success rate has been achieved which indicates that both algorithms achieved a similar level of competency.

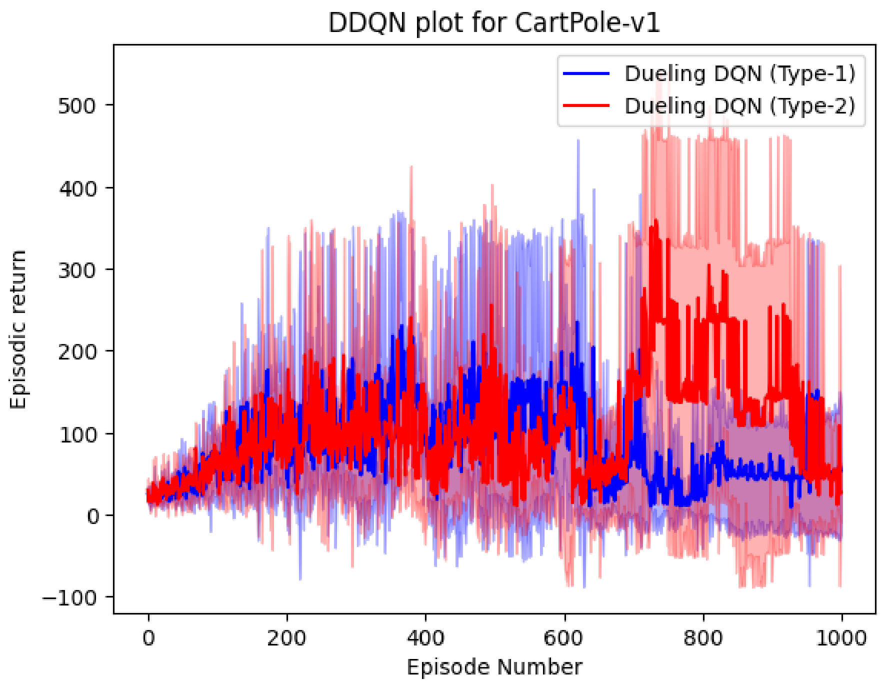
Conjectures:

- The specific advantage estimation method (average vs. max) could influence performance. Type-1's focus on the relative advantage to the average might lead to more stable learning and less overestimation of Q-values.
- Further investigation is needed to determine if the observed performance difference is statistically significant.

Result

In conclusion, a 1000-episode experiment suggests **Type-1** Dueling DQN outperforms Type-2 in the Acrobot-v1 environment. This is likely due to Type-1's stable learning process achieved through its average advantage estimation, potentially avoiding overestimation issues seen in Type-2. Further data analysis and exploration can strengthen these findings.

2.2 DDQN - CartPole-v1



best model, Mean reward $\implies$ Type-1 : 81.9572, Type-2 : 106.333

Hyperparameter : $\alpha = 0.005, \gamma = 0.99$, BUFFER_SIZE: 10^6, BATCH_SIZE $= 64$

The hyperparameters are tuned to minimize regret and in this case it maximizes the return(≈ 500 is threshold) but not stable in that position for many episodes and median line would pass through 150.

Inference & Results:

Based on the image, it appears that Type-2 Dueling DQN outperforms Type-1 Dueling DQN in terms of episodic return when applied to the CartPole-v1 environment. This is because the Dueling DQN (Type-2) line in the graph is consistently higher than the Dueling DQN (Type-1) line.

- **Separation of Advantages:** Both Type-1 and Type-2 Dueling DQN separate the value function (representing the overall goodness of a state) from the advantage function (representing how much better one action is compared to others in that state). This separation can be beneficial for learning, but how it plays out might depend on the environment.
  - **Type-1 (Average):** The advantage for each action is calculated relative to the average advantage across all possible actions in that state (s). This reduces the variance in Q-value estimates because the average advantage is forced to be zero, potentially offering more stable learning with less sensitivity to outliers in advantage values for specific actions.
  - **Type-2 (Max):** The advantage for each action is calculated relative to the maximum advantage across all possible actions in that state (s). This approach focuses on identifying the action with the highest advantage compared to the others, potentially leading to faster identification of the most beneficial action for a given state.
  - **Conclusion:** Simpler environments like **CartPole-v1**, where the optimal policy often involves a clear preference for one action, **Type-2**'s focus on the maximum advantage might be more beneficial. It can identify the superior action more efficiently.
- **Environment Specificity:** As inferred from above, CartPole-v1 is a relatively simple environment with a small action space. **Type-2**'s emphasis on the maximum advantage might help the agent learn this pattern faster.
- **Reduced Variance:** In CartPole-v1, where the optimal policy often involves similar advantages for all actions in most states, Type-1's focus on reducing variance might not be as significant.

Conjectures:

- Visualizing the learned advantage function for both types to see how they prioritize actions.
- Running additional trials with both types to confirm the consistency of the observed performance difference.

Result

This experiment suggests that **Type-2** Dueling DQN might be better suited for the CartPole-v1 environment in this specific case. However, further exploration and analysis are recommended to gain a deeper understanding of the observed performance difference.

Important snippets of Code [Acrobot and Cartpole DDQN]:

```
class QNetworkDueling(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=256, fc2_units=128, update_type='Type-1')
:
        super(QNetworkDueling, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, 128)

        self.value_stream = nn.Linear(128, 1)
        self.advantage_stream = nn.Linear(128, action_size)

        self.update_type = update_type

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

        value = self.value_stream(x)
        advantage = self.advantage_stream(x)

        if self.update_type == 'Type-1':
        q_values = value + advantage - advantage.mean(dim=1, keepdim=True)
        elif self.update_type == 'Type-2':
            q_values = value + advantage - advantage.max(dim=1, keepdim=True)[0]
        else:
            raise ValueError("Invalid update type. Supported types are 'Type-1' and 'Type-2'.")

        return q_values

```

Listing 1: QNetworkDueling class

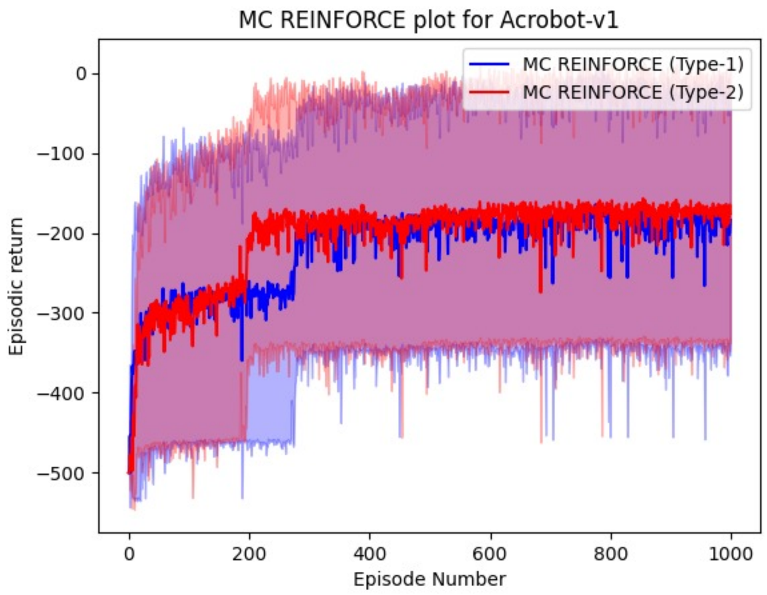
In above code,
Type-1 is defined as follows:

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

Type-2 is defined as follows:

$$Q(s,a;\theta) = V(s;\theta) + \left(A(s,a;\theta) - \max_{a'\in|A|} A(s,a';\theta)\right) \tag{5}$$

2.3 Monte Carlo REINFORCE - Acrobot-v1



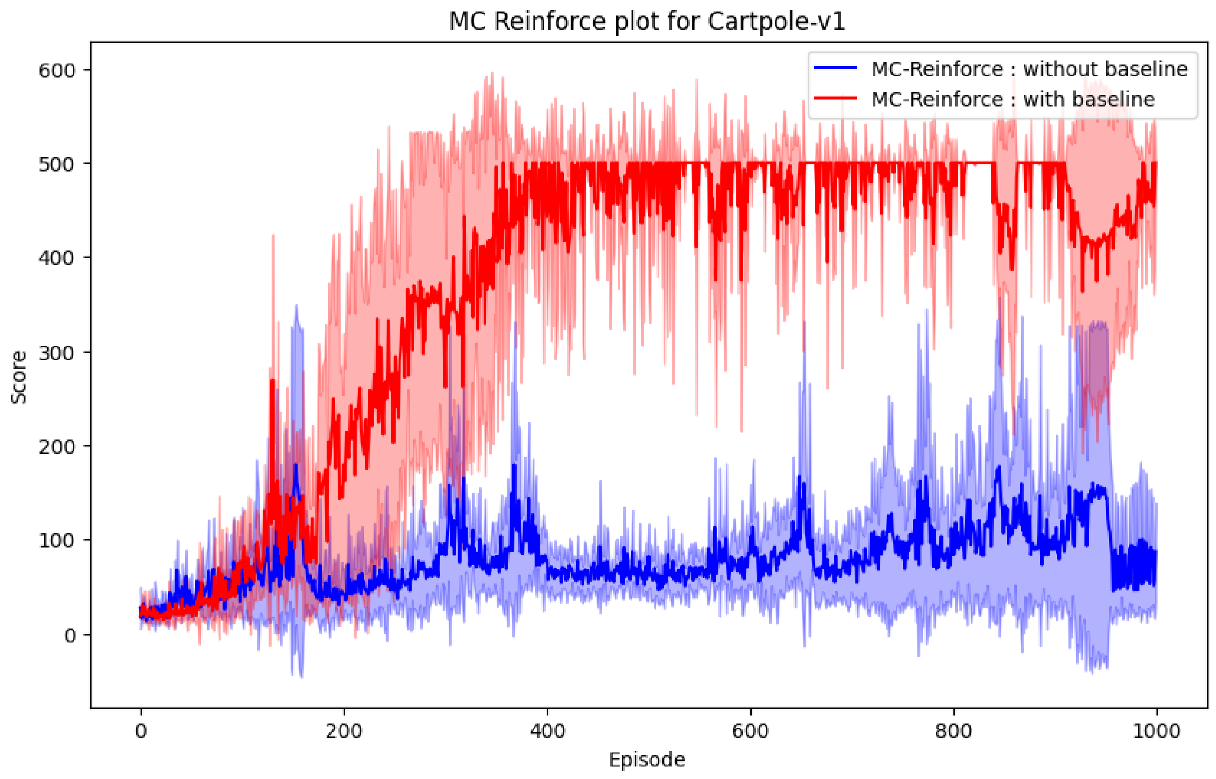
Inference & Results:

From the graph it can be seen that Monte Carlo Reinforce with baseline (Type 2) outperforms without baseline (Type 1) by a small margin and the variance between them almost remains the same, this could be due to the following reasons :

- **Highly Stochastic Environment :** The Acrobot environment might be inherently more stochastic than CartPole. This means that even with a perfect value function, the rewards received for a specific action in a state can vary significantly. This high variance can overshadow the baseline's benefit.
- **Baseline Correlation with Policy :** If the learned value function is highly correlated with the policy network's action selection, subtracting it might not effectively remove the bias. This could happen if both networks share similar architectures or training data, leading them to learn similar representations of the state space.
- **Environment Complexity :** Another Factor to be noted would be the environment dynamics being signifcantly more complex for Acrobot as compared to Cartpole, this could impact the performance of our algorithm

Overall the plots suggest using **Monte Carlo Reinforce with baseline** derived from value functions to obtain better convergence and computational efficiency

2.4 Monte Carlo REINFORCE - CartPole-v1



Inference & Results:

From the graph, it can be seen that the MC Reinforce algorithm with a baseline generally outperforms the MC Reinforce algorithm without a baseline. This suggests that using a baseline can improve the performance of the MC Reinforce algorithm for the Cartpole environment. Some possible reasons why using a baseline might improve performance:

- A potential explanation for this difference between **with baseline** and **without baseline** could be the baseline's role in reducing the variance of the estimated rewards. By subtracting the baseline value from the actual reward, Monte Carlo Reinforce with Baseline focuses on learning the difference between the experienced reward and the expected average reward. This can lead to more stable learning and faster identification of actions that consistently yield higher-than-average rewards.
- A baseline can help to focus the agent's learning on the parts of the state space that are most relevant to the task. This can help the agent to learn more quickly.
- By reducing variance and focusing on advantages, Monte Carlo Reinforce with Baseline learns the optimal policy for CartPole-v1 more efficiently compared to the version without a baseline.

Overall, the results of this experiment suggest that using a baseline can be an effective way to improve the performance of the MC Reinforce algorithm for the Cartpole environment. Also the decrease in sum rewards for Monte Carlo Reinforce with baseline case after 800 episodes could be due to the presence of stochastic effects or seeding the environment.

Conjecture :

- We can observe significant difference in performance in the Cartpole environment as compared to Acrobot, as the case with baseline converges within **400** episodes as compared to the case without baseline being unable to converge even after **1000** episodes
- Moreover our observations regarding the reduced variance after introducing baseline strongly reflect the theory in the case of CartPole in contrast to Acrobot

Important snippets of Code [Acrobot and Cartpole MC REINFORCE]:

```python
def train(self, episodes):
    scores = []

    for episode in range(episodes):
        state = self.env.reset()
        done = False
        rewards = []
        log_probs = []
        states = []  # For TD(0) update
        baseline_values = []

        while not done:
            action, log_prob = self.select_action(state)
            next_state, reward, done, _ = self.env.step(action)
            rewards.append(reward)
            log_probs.append(log_prob)
            states.append(state)  # For TD(0) update
            state = next_state

        if self.use_baseline:
            # Update baseline values using TD(0)
            self.update_baseline(states, rewards)
            # Get baseline values for the current episode
            with torch.no_grad():
                states_tensor = torch.tensor(states).float()
                baseline_values = self.value_network(states_tensor).squeeze().tolist()

        discounted_returns = self.calculate_discounted_returns(rewards)
        self.update_policy(discounted_returns, log_probs, baseline_values)
        scores.append(np.array(rewards).sum())
        if episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(episode, np.mean(scores)))

    return scores
```

Listing 2: Monte Carlo Reinforce Training Function

```python
class Policy(nn.Module):
    def __init__(self, observation_space, action_space):
        super(Policy, self).__init__()
        self.input_layer = nn.Linear(observation_space, 128)
        self.output_layer = nn.Linear(128, action_space)

    def forward(self, x):
        x = self.input_layer(x)
        x = F.relu(x)
        actions = self.output_layer(x)
        action_probs = F.softmax(actions, dim=1)

        return action_probs
```

Listing 3: Policy Learning Network

```python
class Value(nn.Module):
    def __init__(self, observation_space):
        super(Value, self).__init__()

        self.input_layer = nn.Linear(observation_space, 128)
        self.output_layer = nn.Linear(128, 1)

    def forward(self, x):
        x = self.input_layer(x)
        x = F.relu(x)
        state_value = self.output_layer(x)
        return state_value
```

Listing 4: Value Learning Network