CS6700: Reinforcement Learning
PA2 NS24Z111 EE21S048
April 06, 2024
Assignment 2

Team Members:
- Ragu B **(NS24Z111)**
- Matcha Naga Gayathri **(EE21S048)**

Dueling-DQN:

The state value function V(s) represents the value of being in a particular state regardless of the action taken, while the advantage function A(s, a) captures the additional value gained by taking a specific action 'a' in that state s. The Q-value Q(s, a) is then reconstructed from these two functions as

$$Q(s, a)=V(s)+A(s, a)-\text{mean}(A(s, \cdot))$$

where mean$(A(s, \cdot))$ is the mean advantage value across all actions in the state.

Implementation Details:

Given is the snippet of the DDQN implementation:

```
        # Compute Targets
        # targets = r + gamma * target q vals * (1 - dones)
        target_q_values = target_net(new_obses_t)
        max_target_q_values = target_q_values.max(dim=1, keepdim=True)[0]

        targets = rews_t + GAMMA * (1 - dones_t) * max_target_q_values

        # Compute Loss
        q_values = online_net(obses_t)
        action_q_values = torch.gather(input=q_values, dim=1, index=actions_t)

        # loss = nn.functional.smooth_l1_loss(action_q_values, targets)
        loss = nn.functional.mse_loss(action_q_values, targets)
```

- We ran for 250000 episodes, the plots illustrate the total reward values for every 1000 episodes.

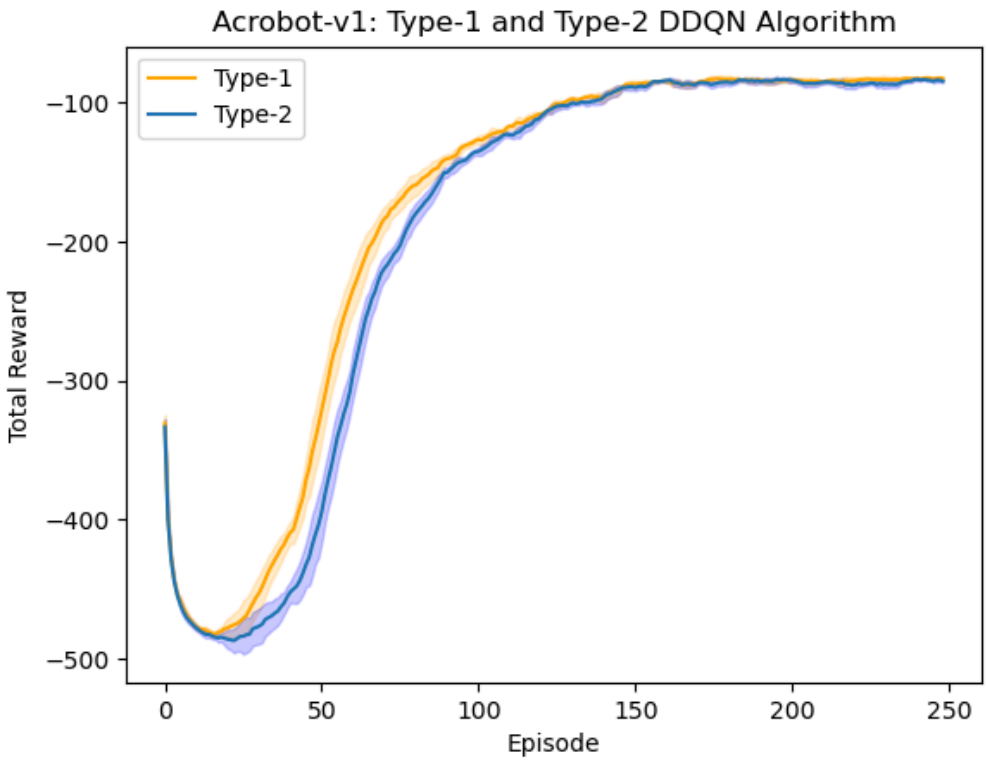
Type-1: Q(s, a)=V(s)+A(s, a)−mean(A(s, ·))

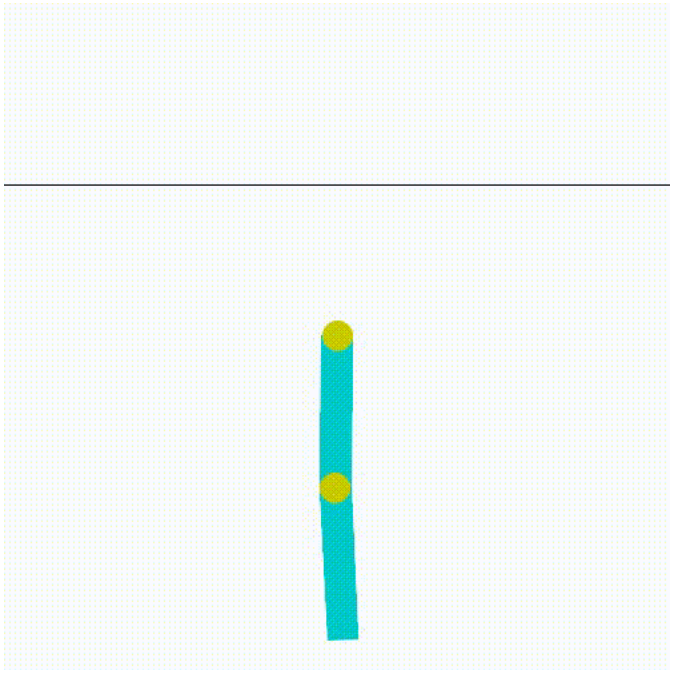
```
        value = self.value_stream(x)
        advantage = self.advantage_stream(x)

        '''Combining value and advantage streams'''
        q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))
        # q_values = value + (advantage - advantage.max(dim=1, keepdim=True)[0])

        return q_values
```

Type-2: Q(s, a)=V(s)+A(s, a)−max(A(s, ·))

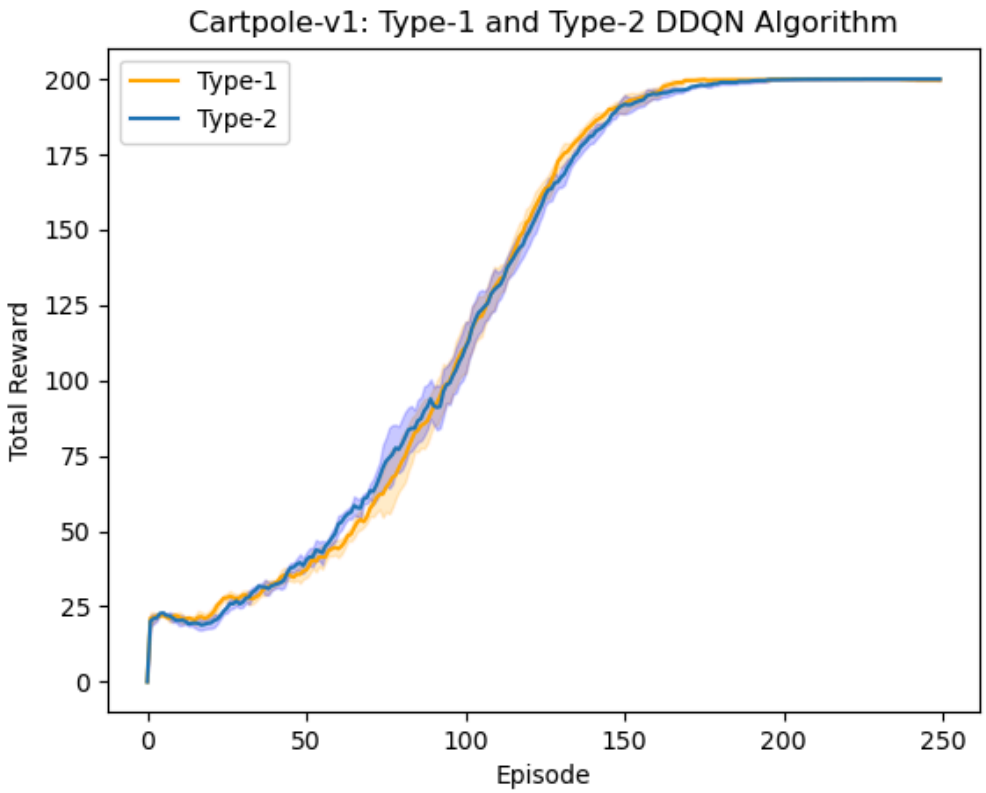
```
        value = self.value_stream(x)
        advantage = self.advantage_stream(x)

        '''Combining value and advantage streams'''
#         q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))
        q_values = value + (advantage - advantage.max(dim=1, keepdim=True)[0])

        return q_values
```

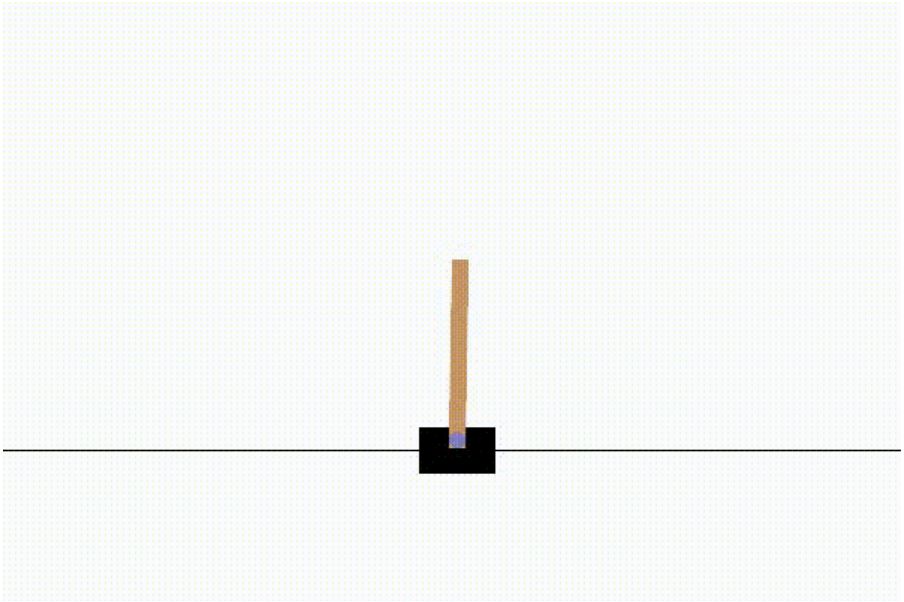
Acrobot-v1:





CartPole-v1:


Cartpole-v1: Type-1 and Type-2 DDQN Algorithm
Type-1
Type-2
Total Reward
Episode
0
25
50
75
100
125
150
175
200
0
50
100
150
200
250



INFERENCE:

Both Type-1 and Type-2 converge to the same total reward over time. Type-1 module replaces the max operator with an average in the equation. This loses the original semantics of V and A because of using average which off-targets by a constant.

But the Type-1 has more stable optimization. In Type-1 the advantages only need to change as fast as the mean, instead of having to compensate any change to the optimal action's advantage in Type-2.

Throughout the training process, an epsilon-greedy action selection strategy is employed. This strategy gradually decreases the epsilon value from 1.0 to 0.02 as training progresses. This approach increased exploration at the start of the training and gradually shifted towards exploitation as training nears completion.

Monte-Carlo REINFORCE:

Given below is a snippet of both policy and state-value networks.

[4]:
```python
#Using a neural network to learn our policy parameters
class PolicyNetwork(nn.Module):

    #Takes in observations and outputs actions
    def __init__(self, observation_space, action_space):
        super(PolicyNetwork, self).__init__()
        self.input_layer = nn.Linear(observation_space, 128)
        self.output_layer = nn.Linear(128, action_space)

    #forward pass
    def forward(self, x):
        #input states
        x = self.input_layer(x)

        #relu activation
        x = F.relu(x)

        #actions
        actions = self.output_layer(x)

        #get softmax for a probability distribution
        action_probs = F.softmax(actions, dim=1)

        return action_probs
```

[5]:
```python
#Using a neural network to learn state value
class StateValueNetwork(nn.Module):

    #Takes in state
    def __init__(self, observation_space):
        super(StateValueNetwork, self).__init__()

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

    def forward(self, x):
        #input layer
        x = self.input_layer(x)

        #activiation relu
        x = F.relu(x)

        #get state value
        state_value = self.output_layer(x)

        return state_value
```

Without Baseline:

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

```
        #iterate rewards from Gt to G0
        for r in reversed(rewards):

            #Base case: G(T) = r(T)
            #Recursive: G(t) = r(t) + G(t+1)^DISCOUNT
            total_r = r + total_r * GAMMA

            #append to discounted rewards
            discounted_rewards.append(total_r)
```

```
        #adjusting policy parameters with gradient ascent
        loss = []
        for r, lp in zip(rewards, log_probs):
            #we add a negative sign since network will perform gradient descen
            loss.append(-r * lp)

        #Backpropagation
        policy_optimizer.zero_grad()
        sum(loss).backward()
        policy_optimizer.step()
```

With Baseline: **delta = G-V**

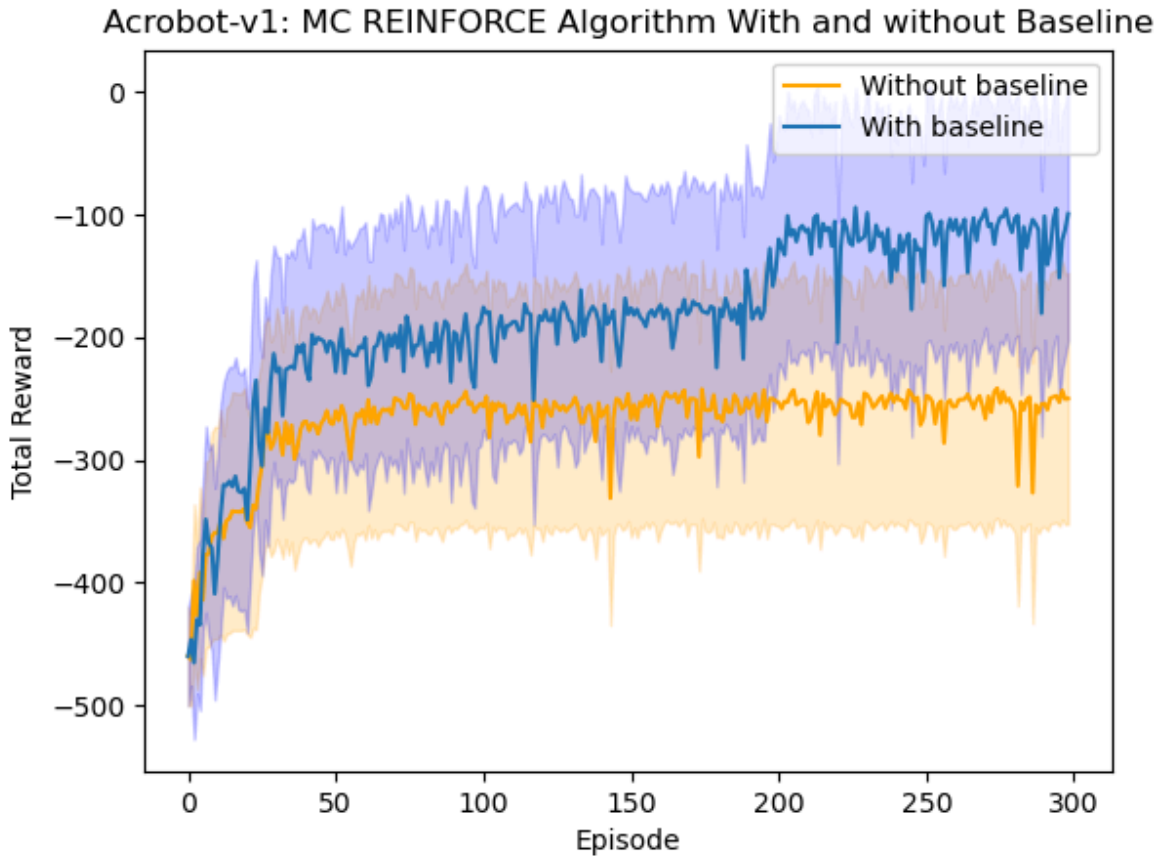
REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

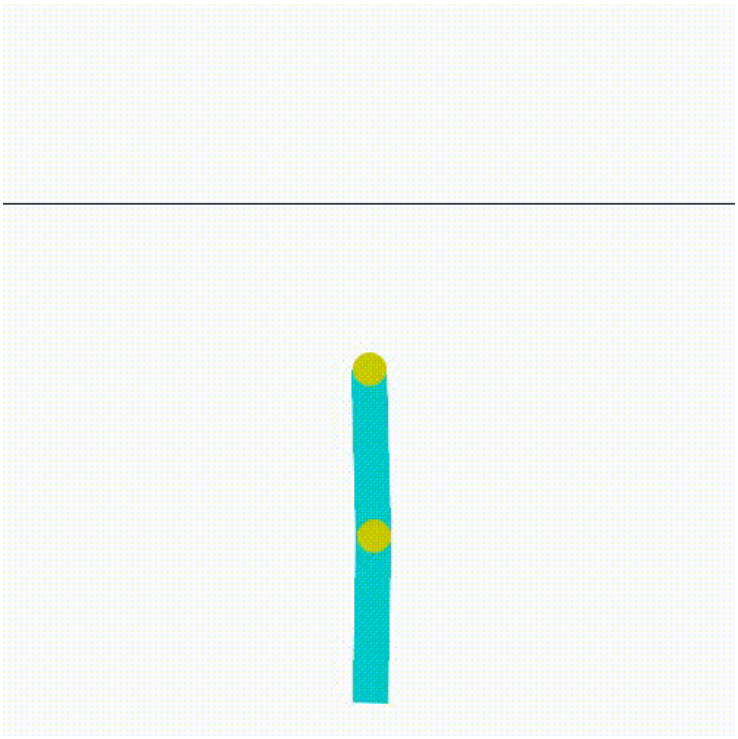
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
    $\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$

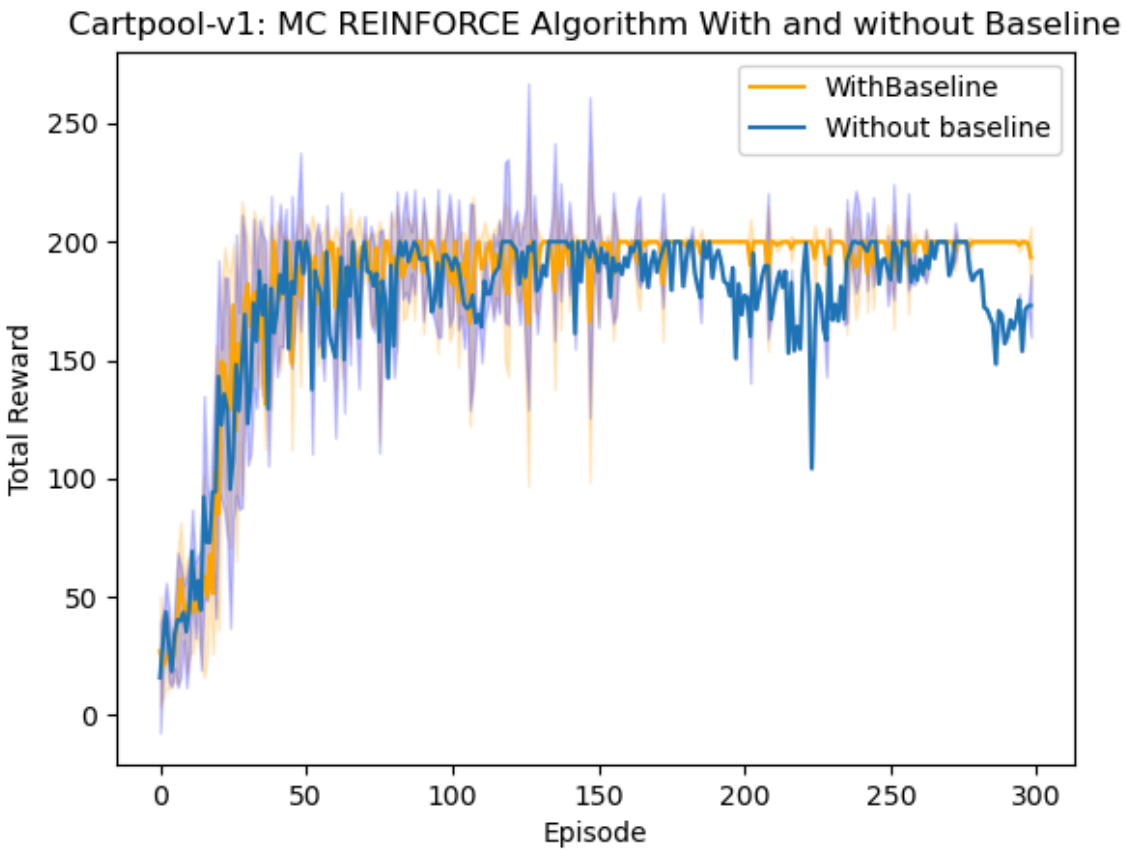
```
#store updates
policy_loss = []

#calculate loss to be backpropagated
for d, lp in zip(deltas, log_probs):
    #add negative sign since we are performing gradient ascent
    policy_loss.append(-d * lp)

#Backpropagation
optimizer.zero_grad()
sum(policy_loss).backward()
optimizer.step()
```

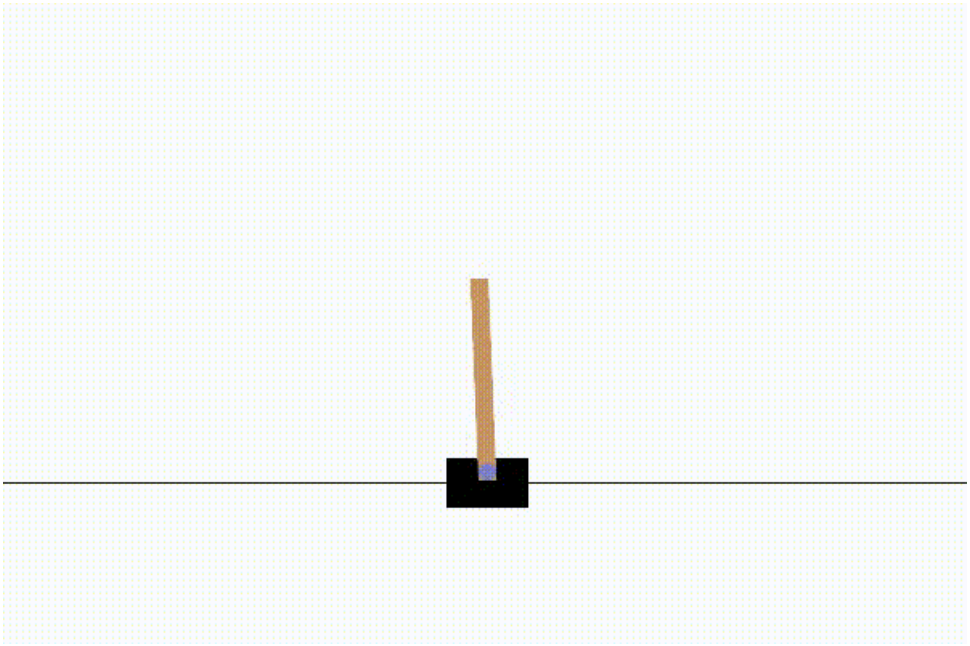
Acrobot-v1:





CartPole-v1:


Cartpool-v1: MC REINFORCE Algorithm With and without Baseline
WithBaseline
Without baseline
Total Reward
Episode
0
50
100
150
200
250
300



INFERENCE:

Adding a baseline to REINFORCE can make it learn much faster, as shown in plot 3 and plot 4.

Variance Reduction:
The baseline helps to reduce the variance of the policy gradient estimates, leading to more stable learning.

Bias:
Adding a baseline introduces bias into the gradient estimate, but it often leads to faster convergence and improved sample efficiency.

Policy Improvement:
REINFORCE with a baseline can result in faster policy improvement compared to REINFORCE without a baseline, especially in environments with high variance in rewards.

Exploration vs. Exploitation:
The choice of baseline affects the balance between exploration and exploitation. A good baseline can help the agent focus on exploring promising regions of the state space while exploiting valuable actions.

In summary, while both REINFORCE with and without a baseline use the Monte Carlo Policy Gradient method, the inclusion of a baseline can lead to more efficient and stable learning by reducing variance and improving sample efficiency.

Document link:
https://docs.google.com/document/d/1tJpKFojObaYbvxv1UoqMCQOn0aPTmfbkAf5TxuZ8i-Q/edit?usp=sharing
Gitlink: https://github.com/GayathriMatcha/CS6700_RL_PA2.git