# CS6700: Reinforcement Learning

# Programming Assignment - II
# Report


HARSHAVARDHAN SRINIVAS PENTAKOTA [ME20B083]

PUTREVU SAI SATWIK [ME20B138]


April 9, 2024

**Table of Contents**

# 1 Algorithm - Dueling DQN

We have two types in Dueling DQN and 2 environments Cartpole-v1 and Acrobot.

Variations of Dueling DQN used:
Type-1:

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

```python
import torch
import torch.nn as nn
import torch.nn.functional as F


'''
Bunch of Hyper parameters (Which you might have to tune later)
'''
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 30       # how often to update the network (When Q target is present)


class DuelingQNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        super(DuelingQNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1_value = nn.Linear(state_size, fc1_units)
        self.fc2_value = nn.Linear(fc1_units, fc2_units)
        self.fc3_value = nn.Linear(fc2_units, 1)  # Output layer for value stream

        self.fc1_advantage = nn.Linear(state_size, fc1_units)
        self.fc2_advantage = nn.Linear(fc1_units, fc2_units)
        self.fc3_advantage = nn.Linear(fc2_units, action_size)  # Output layer for advantage stream

    def forward(self, state):
        x_value = F.relu(self.fc1_value(state))
        x_value = F.relu(self.fc2_value(x_value))
        value = self.fc3_value(x_value)

        x_advantage = F.relu(self.fc1_advantage(state))
        x_advantage = F.relu(self.fc2_advantage(x_advantage))
        advantage = self.fc3_advantage(x_advantage)

        # Combine value and advantage streams to get Q-values using Dueling DQN formula
        q_values = value + (advantage - advantage.mean(dim=1, keepdim=True))

        return q_values
```

Type-2:

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

```python
import torch
import torch.nn as nn
import torch.nn.functional as F



BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
LR = 5e-4               # learning rate
UPDATE_EVERY = 30       # how often to update the network (When Q target is present)


class DuelingQNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        super(DuelingQNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1_value = nn.Linear(state_size, fc1_units)
        self.fc2_value = nn.Linear(fc1_units, fc2_units)
        self.fc3_value = nn.Linear(fc2_units, 1)  # Output layer for value stream

        self.fc1_advantage = nn.Linear(state_size, fc1_units)
        self.fc2_advantage = nn.Linear(fc1_units, fc2_units)
        self.fc3_advantage = nn.Linear(fc2_units, action_size)  # Output layer for advantage stream

    def forward(self, state):
      x_value = F.relu(self.fc1_value(state))
      x_value = F.relu(self.fc2_value(x_value))
      value = self.fc3_value(x_value)
      x_advantage = F.relu(self.fc1_advantage(state))
      x_advantage = F.relu(self.fc2_advantage(x_advantage))
      advantage = self.fc3_advantage(x_advantage)

      # Combine value and advantage streams to get Q-values using Dueling DQN Type-2 formula
      q_values = value + (advantage - advantage.max(dim=1, keepdim=True)[0])

      return q_values
```

## 1.1 Environment - Cartpole-v1

```python
env = gym.make('CartPole-v1')
env.seed(0)

state_shape = env.observation_space.shape[0]
no_of_actions = env.action_space.n

print(state_shape)
print(no_of_actions)
print(env.action_space.sample())
print("----")
state = env.reset()
print(state)
print("----")
action = env.action_space.sample()
print(action)
print("----")
next_state, reward, done, info = env.step(action)
print(next_state)
print(reward)
print(done)
print(info)
print("----")
```

```
4
2
1
----
[ 0.01369617 -0.02302133 -0.04590265 -0.04834723]
----
1
----
[ 0.01323574  0.17272775 -0.04686959 -0.3551522 ]
1.0
False
{}
----
```

**Parameters Used:**

- Five Random seeds are chosen to consider the stochasticity of the problem, the mean and variance of the reward scores obtained are used for evaluating the algorithm.
- The average score of 200 is taken as the threshold for the environment to consider it as solved.
- Learning Rate = 5e-4, Batch size = 64, Gamma = 0.99, No of neurons in the hidden layers(128 ,128) are the parameters that gave the best performance in hyperparameter tuning. Epsilon and Tau for epsilon greedy and softmax policies are made to decay.

**Seeds Used:**

```python
all_scores_type1 = []
seeds = [10, 18, 36, 42, 85]
for i in seeds:
  begin_time = datetime.datetime.now()
  env.reset()
  env.seed(i)
  agent = DuelingTutorialAgent(state_size
  all_scores = ddqn()
  all_scores_type1.append(all_scores)
```

## Policy:

```python
def act(self, state, eps=1.0):


    state = torch.from_numpy(state).float().unsqueeze(0).to(device)
    self.qnetwork_local.eval()
    with torch.no_grad():
        action_values = self.qnetwork_local(state)
    self.qnetwork_local.train()

    ''' Epsilon-greedy action selection (Already Present) '''

    probabilities = softmax(action_values.cpu().data.numpy().flatten()/eps)
    return random.choices(population=range(len(probabilities)), weights=probabilities, k=1)[0]
```

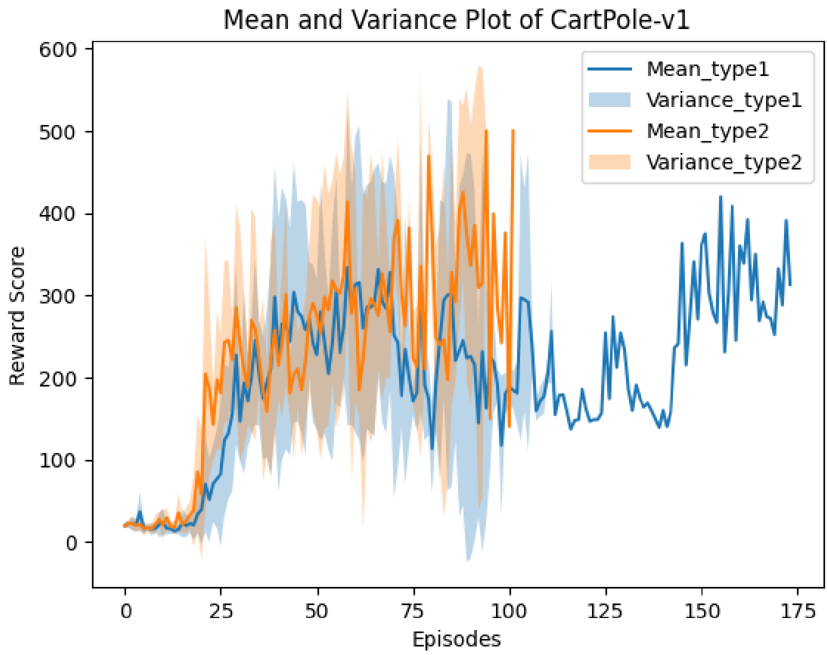
## Learning:

```python
def learn(self, experiences, gamma):
    """ +E EXPERIENCE REPLAY PRESENT """
    states, actions, rewards, next_states, dones = experiences

    ''' Get max predicted Q values (for next states) from target model'''
    Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

    ''' Compute Q targets for current states '''
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

    ''' Get expected Q values from local model '''
    Q_expected = self.qnetwork_local(states).gather(1, actions)

    ''' Compute loss '''
    loss = F.mse_loss(Q_expected, Q_targets)

    ''' Minimize the loss '''
    self.optimizer.zero_grad()
    loss.backward()

    ''' Gradiant Clipping '''
    """ +T TRUNCATION PRESENT """
    for param in self.qnetwork_local.parameters():
        param.grad.data.clamp_(-1, 1)

    self.optimizer.step()
```

## Algorithm:

```python
def ddqn(n_episodes=1000, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    all_scores = []
    scores_window = deque(maxlen=100)
    ''' last 100 scores for checking if the avg is more than 195 '''

    eps = eps_start
    ''' initialize epsilon '''

    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            next_state, reward, done, _ = env.step(action)
            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break

        scores_window.append(score)
        all_scores.append(score)

        eps = max(eps_end, eps_decay*eps)
        ''' decrease epsilon '''

        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)), end="")

        if i_episode % 100 == 0:
           print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
        if np.mean(scores_window)>=200.0:
           print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
           break
    return all_scores
```

**Reward Plot comparing Type-1 and Type-2:**




## 1.2 Observations and Inferences:

- Softmax policy performed better for both the algorithms when compared to epsilon greedy policy.
- Both the Algorithms didn't converge for a threshold of 500 and very fluctuating so we tested for 200.
- We can infer that the variance of both the algorithms is quite large indicating instability in the process of learning shows that both the algorithms struggled to learn and stabilize for CartPole environment.
- From the reward curves we can infer that DDQN Type 2 shows consistency in achieving high scores and solving the environment within a relatively small range of episodes for 200 value of threshold.
- It not only learned faster but also for random seeds it took similar number of seeds to converge.
- Dueling DQN of type 1 showed more variability in performance for different number of seeds.
- From the experiment we can say that Dueling DQN algorithm could not converge or learn for the CartPole environment.

## 1.3 Environment - Acrobot-v1:

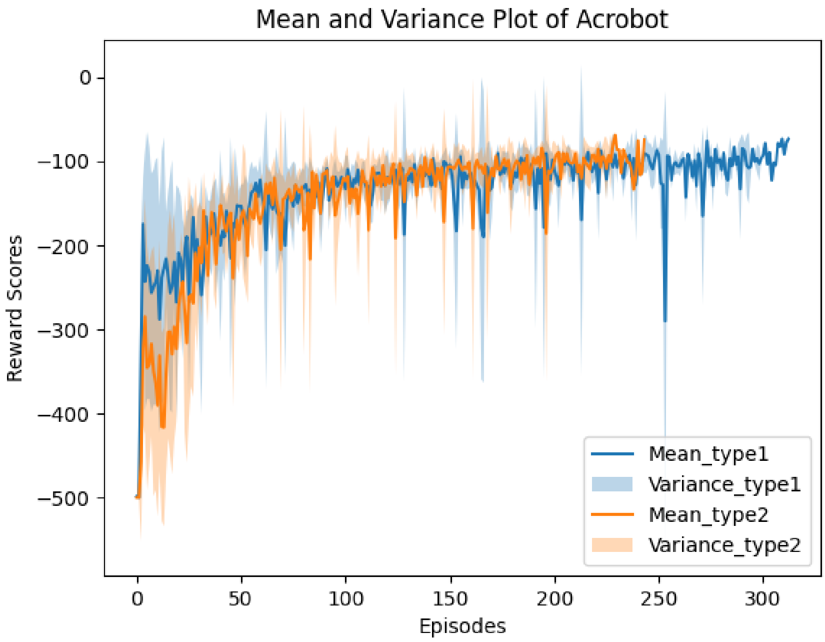
```python
import gym
env = gym.make('Acrobot-v1')
env.seed(0)
state_shape = env.observation_space.shape[0]
no_of_actions = env.action_space.n
print("State shape:", state_shape)
print("Number of actions:", no_of_actions)
print("Sampled action:", env.action_space.sample())
print("----")
state = env.reset()
print("Initial state:", state)
print("----")
# Sample a random action
action = env.action_space.sample()
print("Random action:", action)
print("----")
# Take the sampled action in the environment
next_state, reward, done, info = env.step(action)
print("New state:", next_state)
print("Reward:", reward)
print("Episode done?", done)
print("Additional info:", info)
print("----")
```

```
State shape: 6
Number of actions: 3
Sampled action: 1
----
Initial state: [ 0.99962485  0.02738891  0.9989402  -0.04602639 -0.09180529 -0.09669447]
----
Random action: 2
----
New state: [ 0.99996984 -0.0077642   0.9997182  -0.02373883 -0.25169677  0.31000718]
Reward: -1.0
Episode done? False
```

**Parameters used:**

- Five Random seeds are chosen to consider the stochasticity of the problem, the mean and variance of the reward scores obtained are used for evaluating the algorithm.
- The average score of -100 is taken as the threshold for the environment to consider it as solved.
- Learning Rate = 1e-4, Batch size = 64, Gamma = 0.99, No of neurons in the hidden layers(128 ,128) are the parameters that gave the best performance in hyperparameter tuning. Epsilon and Tau for epsilon greedy and softmax policies are made to decay.

**Reward Plot for comparing Type-1 and Type-2:**




## 1.4 Observations and Inferences:

- Similar to CartPole environment, in Acrobot environment softmax policy performed better than epsilon greedy policy for both the types of Dueling DQN.
- From the reward plot we can observe that Type 1 initially learned faster than Type 2 algoritmn for this environment but it took more episodes on averge to converge and learn.
- For this environment unlike CartPole both the algorithm's variance is not much deviated showing consistency and convergence.
- Dueling DQN of type 2 converged in fewer episodes on average compared to type 1
- Both the algorithm didnt show much variation in learning for different values of seeds showing stability in learning in the algorithms.

## 2 Algorithm - MC REINFORCE

We have two types in MC REINFORCE and 2 environments Cartpole-v1 and Acrobot.

**Variations of MC REINFORCE used:**
Type-1:

$$\theta = \theta + \alpha G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \tag{1}$$

Type-2:

$$\theta = \theta + \alpha(G_t - V(S_t|\Phi)) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \tag{2}$$

### Algorithm for Type 1:

**Algorithm 1** Monte Carlo REINFORCE with Baseline

**Input:** A differentiable policy parameterization $\pi(a|s, \theta)$
**Algorithm parameter:** Step size $\alpha > 0$
**Initialize** policy parameter $\theta \in \mathbb{R}^{d_0}$ (e.g., to 0)
  **while** true **do**
    Generate an episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
    **for** $t = 0, 1, ..., T-1$ **do**
      $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (return $G_t$)
      $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi(A_t|S_t, \theta)$
    **end for**
  **end while**

### Code snippets for loading Environment:

```python
''''
Function to load environment
'''
def LoadingEnv(seed = 0,name = 'CartPole-v1'):
  env = gym.make(name)
  env.seed(seed)
  state_shape = env.observation_space.shape[0]
  no_of_actions = env.action_space.n
  return env,state_shape, no_of_actions
```

### Code snippets for Policy Network

```python
#policy Network
class JNetwork(nn.Module):

    def __init__(self, state_size, action_size, seed, fc1_units=128,
     ↪ fc2_units=64):

        super(JNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
```

```
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        """Build a network that maps state -> action values."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)

```

**Code snippets for Agent - Type I:**

```
'''
CREATING REINFORCE agent
'''
class REINFORCEAgentType1():
    def __init__(self, state_size, action_size,
     ↪ seed,fc1_units,fc2_units,lr=0.01):
        # PARAMETERS OF AGENT
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # CREATE A NETWORK
        self.policy_network = JNetwork(state_size, action_size,
         ↪ seed,fc1_units,fc2_units).to(device)
        self.optimizer = optim.Adam(self.policy_network.parameters(),
         ↪ lr=lr)


        self.saved_log_probs = []
        self.rewards = []
        self.states  =[]

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        logits = self.policy_network(state)

        # WE USE SOFTMAX FOR REINFORCE
        probs = torch.softmax(logits, dim=1)

        # USING TORCH CATEGORICAL DISTRUBTION FOR SMAPLING
        # USING SOFTMAX WEIGHTS
        m = torch.distributions.Categorical(probs)
        action = m.sample()

        # STORING LOG OF OUTPUT POLICY
        self.saved_log_probs.append(m.log_prob(action))
```

```python

        # RETURNING ACTION
        return action.item()

    def learn(self, gamma):

        # FINDING G(t)
        G = []
        for t in range(len(self.rewards)):
          rewards_t  = self.rewards[t:]
          discounts_t = [gamma ** i for i in range(len(rewards_t) + 1)]
          G.append(sum([a * b for a, b in zip(discounts_t,rewards_t)]))

        policy_loss = []
        for t,log_prob in enumerate(self.saved_log_probs):
            policy_loss.append(-log_prob * G[t])
        policy_loss = torch.cat(policy_loss).sum()

        # doing gradient descent( we need gradient ascent
        #so we use -log_loss as loss)
        self.optimizer.zero_grad()
        policy_loss.backward()
        self.optimizer.step()


        # RESERTING CONTAINERS
        self.saved_log_probs = []
        self.rewards = []
        self.states = []
```

**Algorithm 2** Policy Gradient with State-Value Function Approximation

**Input:** A differentiable policy parameterization $\pi(a|s,\theta)$
**Input:** A differentiable state-value function parameterization $v(s,w)$
**Algorithm parameters:** Step sizes $\alpha_\theta > 0$, $\alpha_w > 0$
**Initialize** policy parameter $\theta \in \mathbb{R}^{d_\theta}$ and state-value weights $w \in \mathbb{R}^{d_w}$ (e.g., to 0)

**loop**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot,\theta)$
  **for** $t = 0, 1, \ldots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (return $G_t$)
    $\delta \leftarrow R_{t+1} + \gamma v(S_{t+1}, w) - v(S_t, w)$
    $w \leftarrow w + \alpha_w \delta \nabla_w v(S_t, w)$
    $\theta \leftarrow \theta + \alpha_\theta (G - V(S_t|W) \nabla_\theta \ln \pi(A_t|S_t,\theta)$
  **end for**
**end loop**

• The baseline network is the same neural structure as the policy network. we used different learning rate to update the baseline network.

**Code snippets for Agent - Type II:**

```python
# Define the neural network architecture
class BNetwork(nn.Module):
    def __init__(self, input_size, output_size, seed, fc1_units=128, fc2_units=64):
        super(BNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(input_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, output_size)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)

# Define the REINFORCE agent with baseline
class REINFORCEAgentType2():
    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=64, lr1=0.01, lr2=0.01):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Policy network
        self.policy_network = JNetwork(state_size, action_size, seed, fc1_units, fc2_units).to(device)
        self.optimizer = optim.Adam(self.policy_network.parameters(), lr=lr1)

        # Baseline network (state-value function)
        self.baseline_network = BNetwork(state_size, 1, seed, fc1_units, fc2_units).to(device)
        self.baseline_optimizer = optim.Adam(self.baseline_network.parameters(), lr=lr2)

        self.saved_log_probs = []
        self.rewards = []
        self.states = []

    def act(self, state):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        logits = self.policy_network(state)
        probs = torch.softmax(logits, dim=1)
        m = torch.distributions.Categorical(probs)
        action = m.sample()
        self.saved_log_probs.append(m.log_prob(action))
        return action.item()

```

```
    def learn(self, gamma):
        G = []
        for t in range(len(self.rewards)):
            rewards_t = self.rewards[t:]
            discounts_t = [gamma ** i for i in range(len(rewards_t) + 1)]
            G.append(sum([a * b for a, b in zip(discounts_t, rewards_t)]))
        value_loss = []
        for t in range(len(G)-1):
          state = torch.from_numpy(self.states[t]).float().unsqueeze(0).to(device)
          next_state = torch.from_numpy(self.states[t+1]).float().unsqueeze(0).to(device)
          reward = torch.tensor(self.rewards[t], dtype=torch.float32)
          # Compute TD(0) error
          target = reward + gamma * self.baseline_network(next_state)
          prediction = self.baseline_network(state)
          td_error = target - prediction
          # Compute loss and update the value network
          loss = td_error ** 2
          value_loss.append(loss)


        value_loss = torch.cat(value_loss).sum()
        self.baseline_optimizer.zero_grad()
        value_loss.backward()
        self.baseline_optimizer.step()

        policy_loss = []
        for t, log_prob in enumerate(self.saved_log_probs):
            state = torch.from_numpy(self.states[t]).float().unsqueeze(0).to(device)
            advantage = G[t] - self.baseline_network(state)  # Use the same delta calculated for baseline update
            policy_loss.append(-log_prob * advantage)
        policy_loss = torch.cat(policy_loss).sum()

        self.optimizer.zero_grad()
        policy_loss.backward()
        self.optimizer.step()

        # Reset containers
        self.saved_log_probs = []
        self.rewards = []
        self.states = []

```

**Code snippets for Reinforcement Algorithm:**

```python
# Define the REINFORCE algorithm
def ReinforceAlgo(env, agent, n_episodes=10000, max_t=1000, gamma=0.99,reward_threshold = 195, flag = True):
    scores_window = deque(maxlen=100)
    rewards_per_episode = []
    for i_episode in range(1, n_episodes+1):
        state = env.reset()
        score = 0
        for t in range(max_t):
            agent.states.append(state)
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            # strong rewards for each timpestep of episode
            agent.rewards.append(reward)
            state = next_state
            # net score of episode
            score += reward
            if done:
                break

        scores_window.append(score)
        rewards_per_episode.append(score)

        agent.learn(gamma)

        if i_episode % 100 == 0:
          print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))

        if flag:
          if np.mean(scores_window) >= reward_threshold:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_window)))
            break

    return rewards_per_episode
```

**Algorithm Description:** The provided snippets outline two variations of the REINFORCE algorithm: one without baseline (Type I) and another with a baseline network (Type II). Both algorithms follow a similar structure but differ in how they update the policy.

**Neural Network Architecture:** Both types of agents utilize neural networks for policy approximation and baseline estimation. The architecture consists of fully connected layers (with ReLU activations) followed by an output layer. The baseline network in Type II has a single output for state-value estimation.

## 2.1 Environment - "Acrobot-v1"

• Action space = 3 • State space = 6

**Hyperparameter tunning:**

• We conducted different experiments for Type - I of the algorithm with different hyperparameters with convergence criteria of -100.

| fc1_units | fc2_units | Learning rate | Average Score[ all episodes] | Training Time (s) |
|---|---|---|---|---|
| 128 | 64 | 1e-06 | -495.34 | 3595.88 |
| 128 | 64 | 0.1 | -499.97 | 3935.26 |
| 128 | 64 | 0.0001 | -179.00 | 1432.77 |
| 128 | 128 | 0.0001 | -215.31 | 1117.49 |
| 256 | 128 | 0.0001 | -153.91 | 509.76 |
| 256 | 256 | 0.0001 | -146.45 | 422.55 |

**Table 1:** Hyperparameter Tuning Results

**Table 2:** final Hyperparameters Used in the Type 1 Model

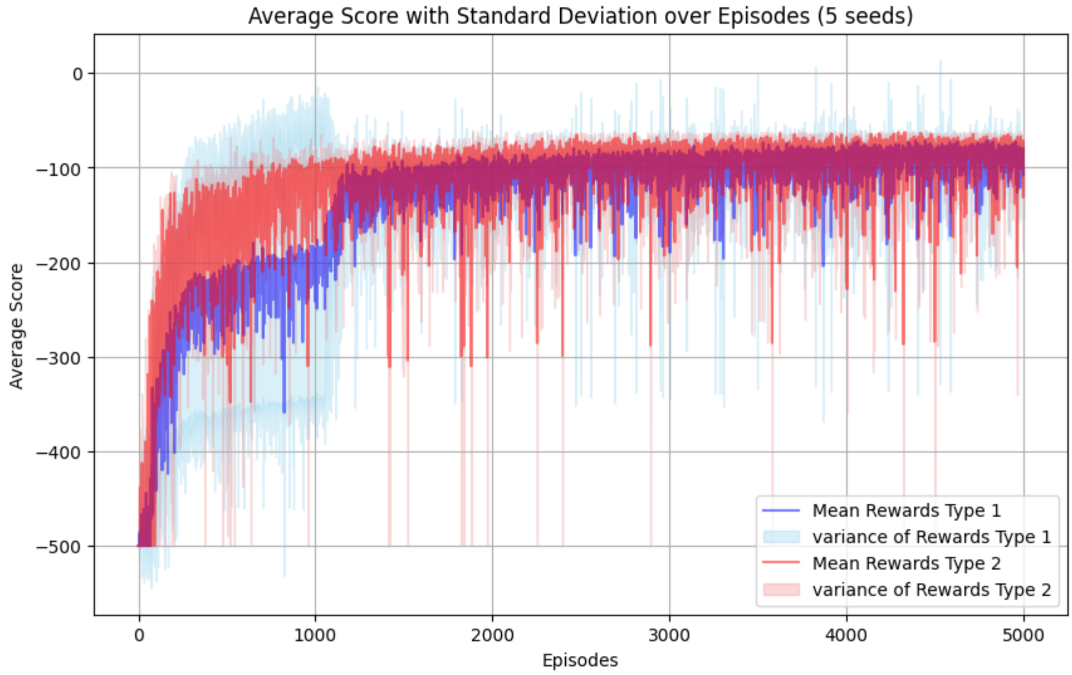
| Hyperparameter | Value |
|---|---|
| Learning Rate | 0.0001 |
| Mode of update | Batch mode update for each episode |
| Maximum timesteps | 500 |
| Policy Network: Hidden Layer 1 | 256 nodes |
| Policy Network: Hidden Layer 1 | 256 nodes |
| gamma | 0.99 |

**Table 3:** final Hyperparameters Used in the Type 1I Model

| Hyperparameter | Value |
|---|---|
| Policy Network: Learning Rate | 0.0001 |
| Baseline Network: Learning Rate | 0.0001 |
| Mode of update | Batch mode update for each episode |
| Maximum timesteps | 500 |
| Policy Network: Hidden Layer 1 | 256 nodes |
| Policy Network: Hidden Layer 1 | 256 nodes |
| Baseline Network: Hidden Layer 1 | 256 nodes |
| Baseline Network: Hidden Layer 1 | 256 nodes |
| gamma | 0.99 |

**Reward Plot of Two Types of MC REINFORCE Algorithm:**

• We conducted both types of algo for 5000 episodes each and computed the average reward across 5 different random seeds for each type.



## 2.2 Environment - "CartPole-v1"

• Action space = 2 • State space = 4

**Hyperparameter tunning:**

• We conducted different experiments for Type - I of the algorithm with different hyperparameters with convergence criteria of 195.

| fc1_units | fc2_units | Learning rate | Average Score[ all episodes] | Training Time (s) |
|---|---|---|---|---|
| 128 | 64 | 1e-06 | 21.95 | 108.49 |
| 128 | 64 | 0.1 | 9.36 | 48.67 |
| 128 | 64 | 0.0001 | 78.22 | 70.12 |
| 64 | 32 | 0.0001 | 73.46 | 119.17 |
| 256 | 128 | 0.0001 | 97.63 | 67.03 |
| 128 | 128 | 0.0001 | 84.54 | 60.19 |

**Table 4:** Hyperparameter Tuning Results

**Table 5:** final Hyperparameters Used in the Type 1 Model

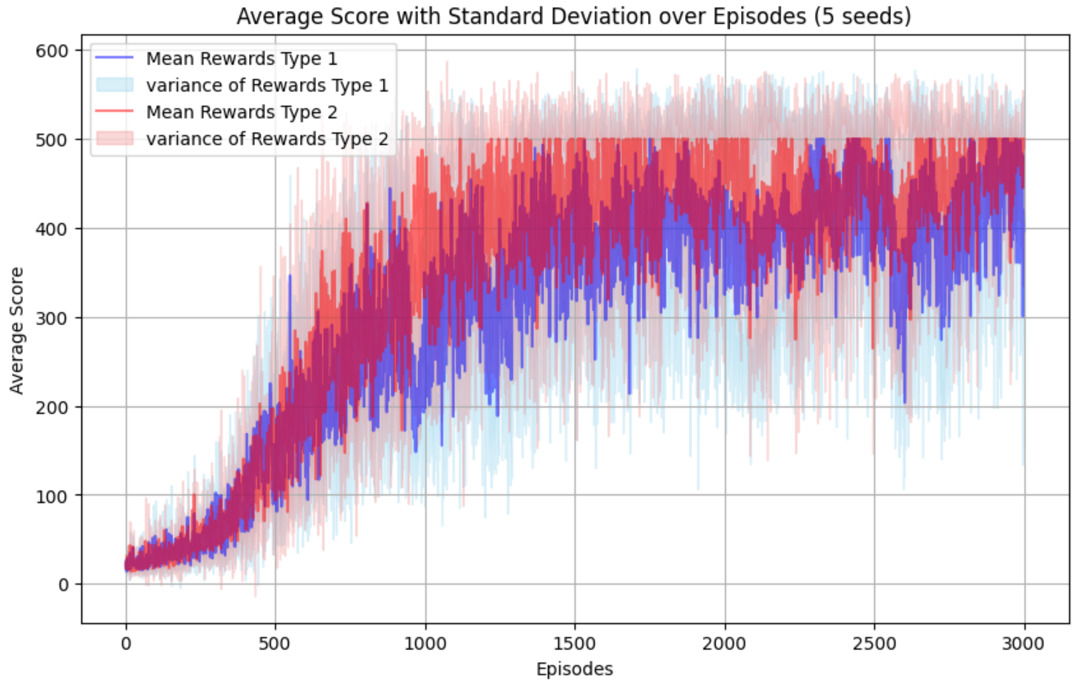
| **Hyperparameter** | **Value** |
|---|---|
| Learning Rate | 0.0001 |
| Mode of update | Batch mode update for each episode |
| Maximum timesteps | 500 |
| Policy Network: Hidden Layer 1 | 256 nodes |
| Policy Network: Hidden Layer 1 | 128 nodes |
| gamma | 0.99 |

**Table 6:** final Hyperparameters Used in the Type 1I Model

| **Hyperparameter** | **Value** |
|---|---|
| Policy Network: Learning Rate | 0.0001 |
| Baseline Network: Learning Rate | 0.0001 |
| Mode of update | Batch mode update for each episode |
| Maximum timesteps | 500 |
| Policy Network: Hidden Layer 1 | 256 nodes |
| Policy Network: Hidden Layer 1 | 128 nodes |
| Baseline Network: Hidden Layer 1 | 256 nodes |
| Baseline Network: Hidden Layer 1 | 128 nodes |
| gamma | 0.99 |

**Reward Plot of Two Types of MC REINFORCE Algorithm:**

• We conducted both types of algo for 3000 episodes each and computed the average reward across 5 different random seeds for each type.



## 2.3 Description of details of experiments :

1. **Hyperparameter Tuning:** Hyperparameters such as learning rates and network architecture (number of units in hidden layers) have been experimented with to find the optimal combination for training stability and convergence. The tuning results are provided for both environments (Acrobot-v1 and CartPole-v1).

2. **Performance Evaluation:** The performance of both algorithms is evaluated based on the rewards achieved over multiple episodes. The reward plots depict the learning curves of both algorithms across different episodes and random seeds.

3. **Convergence:** Convergence criteria are defined based on reaching a certain threshold of the score window(len = 100), indicating successful learning. For example, in Acrobot-v1, the convergence criteria for Type I was -100, while for CartPole-v1, it was 195 in experiments of hyperparameter tunning. For reward plots, we just ran each for fixed no. of episodes.

4. **Baseline Network:** Type II incorporates a baseline network to reduce the variance of the policy gradient estimates. It estimates the state-value function and updates the baseline alongside the policy network.

5. **Mode of Update:** Both types of agents use batch mode update for each episode, where the policy and baseline networks are updated after collecting experiences from a single episode.

6. **Gamma (Discount Factor):** A discount factor (gamma) of 0.99 is used in both algorithms to discount future rewards.

## 2.4 Inferences and conjectures from all your experiments and results:

1. **Effect of Hyperparameters on Performance:**Overall, in both environments. Learning rates of 1e-6 and 1e-1 did not yield satisfactory results, but a **learning rate of 1e-4 performed well**. Therefore, we fixed the learning rate and experimented with varying numbers of nodes to find the optimal neural network structure.

2. **Impact of Network Architecture:** It's observed that **deeper networks** with more units performed better in both environments.

3. **Baseline Network's Role**: The addition of a baseline network (Type II) helps reduce the variance of policy gradient estimates, potentially leading to more stable training and faster convergence. This suggests that incorporating a state-value function approximation alongside the policy network improves learning efficiency. **From reward Plots especially in cartpole env, The variance of type 2 is much less compared to type 1.**

4. **Difference in Environments:** The performance of the algorithms varies across different environments (Acrobot-v1 and CartPole-v1), which have distinct state and action spaces. The acrobat took less time to converge compared to the cartpole.

5. **Policy and Value Network Updates**: Type II algorithm updates both the policy and baseline networks in a coordinated manner, leveraging the TD(0) error for value network updates. This approach potentially facilitates more effective learning by considering the difference between predicted and target values. **we can see in reward plots that type II reached the threshold faster than Type I**

**Overall for MC REINFORCE:** In both environments, the Type II reinforcement learning algorithm outperformed Type I. **Type II exhibited faster convergence and lower variance**, leading to more stable training dynamics. By incorporating a baseline network and leveraging TD(0) error for coordinated updates, Type II achieved superior performance compared to Type I. Notably, Type II demonstrated more significant improvements in the Acrobot environment compared to CartPole, where the differences were less pronounced. Overall, Type II is preferred for its efficiency and effectiveness in learning tasks

.

**GITHUB REPO LINK:**
GITHUB USERNAMES:
ME20B083 : harshavardhan379
ME20B138 : SATWIK-ME20B138

GITHUB REPO LINK : LINK.