DA6400 - Reinforcement Learning

Programming Assignment 2

Patel Shrey Rakeshkumar (ME21B138)
Param Jivrajani (PH21B006)

Table of Contents

1 Introduction **3**

2 Environments **3**

2.1 Environment 1 : CartPole-v1 3

2.1.1 Description of the Environment 3

2.2 Environment 2 : Acrobot-v1 4

2.2.1 Description of the Environment 4

3 Implementation and Code Snippets **4**

3.1 Dueling-DQN 5

3.2 Monte-Carlo REINFORCE 8

4 Values Used **10**

5 Results and Plots **11**

5.1 Dueling-DQN 11

5.2 Monte-Carlo REINFORCE 12

5.3 CartPole-v1 12

5.3.1 Dueling-DQN 12

5.3.2 Monte-Carlo REINFORCE 14

5.4 Acrobot-v1 17

5.4.1 Dueling-DQN 17

5.4.2 Monte-Carlo REINFORCE 19

6 Link to Github **21**

7 References **21**

8 Acknowledgements **21**

1 Introduction

This report consists of using OpenAI's Gym library to simulate multiple environments using Dueling-DQN and Monte-Carlo REINFORCE algorithms.

The environments simulated in this assignment are :

- **CartPole-v1**
- **Acrobot-v1**

2 Environments

2.1 Environment 1 : CartPole-v1

CartPole-v1 is one of the most widely used environments in gymnasium. The environment consists of the pole attached to a cart moving along a frictionless track. The goal is to balance the pole on the cart for as long as possible, applying left and right forces.

2.1.1 Description of the Environment

- **Action Space** : Consists of an ndarray of shape (1,) which can take the values {0,1}. The actions are encoded as follows :
  - 0 : Push cart to the left
  - 1 : Push cart to the right
- **Observation Space** : Consists of an ndarray of shape (4,) with the following values :

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -0.418 rad ($-24°$)	~ 0.418 rad ($24°$)
3	Pole Angular Velocity	-Inf	Inf

Table 1: Observation Space

  The cart x-position (index 0) can be take values between (-4.8, 4.8), but the episode terminates if the cart leaves the (-2.4, 2.4) range.

  The pole angle can be observed between (-0.418, 0.418) radians (or 24°), but the episode terminates if the pole angle is not in the range (-0.2095, 0.2095) (or $\pm 12°$)

- **Rewards** : A default reward of +1 is given for each step since the goal is to balance the pole for as long as possible.
- **Episode End** : The episode either by truncation after completion of the maximum steps, which is 500 for the CartPole-v1 environment, or by termination if any of the states crosses the threshold value given above.
- **Starting State** : All observations are assigned a uniformly random value in [-0.05, 0.05].

2.2 Environment 2 : Acrobot-v1

Acrobot-v1 is a system consisting of two links connected 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.

2.2.1 Description of the Environment

- **Action Space** : Consists of an ndarray of shape (1,) which can take the values {0,1}. The actions are encoded as follows :
  - 0 : Apply -1 Nm torque to the actuated joint
  - 1 : Apply 0 torque to the actuated joint
  - 2 : Apply +1 Nm torque to the actuated joint
- **Observation Space** : Consists of an ndarray of shape (6,) with the following values :

Num	Observation	Min	Max
0	Cosine of θ_1	-1	1
1	Sine of θ_1	-1	1
2	Cosine of θ_2	-1	1
3	Sine of θ_2	-1	1
4	Angular velocity of θ_1	~ -12.567 (-4π)	~ 12.567 (4π)
5	Angular velocity of θ_2	~ -28.274 (-9π)	~ 28.274 (9π)

Table 2: Observation ranges for θ_1 and θ_2

  θ_1 is the angle of the first joint and θ_2 is the angle of the second link relative to the first. An angle of $\theta_2 = 0$ represents both links having the same angle.

  The angular velocities of θ_1 and θ_2 are bounded at 4π, and 9π rad/s respectively.

- **Rewards** : A default reward of -1 is given for each step that does not reach the goal state, since the goal is to have the free end reach a designated target height in as few steps as possible. Achieving the target height results in a termination reward of 0. The reward threshold is set at -100, which is also used to calculate the regret.
- **Episode End** : The episode either by truncation after completion of the maximum steps, which is 500 for the v1 environment or by termination which is given by the condition $\cos(\theta_1 + \theta_2) - cos(\theta_1) > 1$.
- **Starting State** : Each parameter given in the observation is initialized uniformly between -0.1 and 0.1.

3 Implementation and Code Snippets

The implementation was done using the inbuilt ***gymnasium*** library in Python.

- The state space in the above environments are continuous with a Discrete Action space.

- For both the algorithms, neural networks are used to represent the value functions , the policy or the advantage function based on the algorithm.
- The ***Pytorch*** library was used to define the neural networks for both the cases.
- For both, Dueling DQN and Monte-Carlo REINFORCE , two types of updates are used, which have been explained below.

3.1 Dueling-DQN

- The Q-Network was represented as a neural network, using multiple layers , which was used as a hyper parameter.
- The ReLU activation was used for each layer except the last layer.
- Two types of updates were used for this algorithm :

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

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

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

```
class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, num_layers, fc_units_lis, Type):
        super(QNetwork, self).__init__()
        self.Type = Type
        self.fc_lis = nn.ModuleList()
        self.num_layers = num_layers

        self.fc_lis.append(nn.Linear(state_size, fc_units_lis[0]))
        for i in range(self.num_layers - 1):
            self.fc_lis.append(nn.Linear(fc_units_lis[i],fc_units_lis[i+1]))

        self.adv = nn.Linear(fc_units_lis[-1] , action_size)
        self.value = nn.Linear(fc_units_lis[-1] , 1)

    def forward(self , state):
        x = state
        for i in range(len(self.fc_lis)):
            x = F.relu(self.fc_lis[i](x))

        value = self.value(x)
        adv = self.adv(x)

        if self.Type == 'Type 1':
            adv_value = torch.mean(adv, dim=1, keepdim=True)

        if self.Type == 'Type 2':
            adv_value = torch.max(adv, dim=1, keepdim=True)[0]

        q_value = value + (adv-adv_value)

        return q_value
```

- Replay buffer was used to randomly sample experiences from the history and learn from those. This removes correlation between previous samples and ensures stable learning. Samples are chosen randomly from the memory, with each batch then used for learning.

```
def add(self, state, action, reward, next_state, done):
    idx = self.position
    self.state_memory[idx] = state
    self.action_memory[idx] = action
    self.reward_memory[idx] = reward
    self.next_state_memory[idx] = next_state
    self.done_memory[idx] = done

    self.position = (self.position + 1) % self.buffer_size
    self.size = min(self.size + 1, self.buffer_size)

def sample(self):
    indices = np.random.choice(self.size, self.batch_size, replace=False)

    # Return batch as numpy arrays (more efficient for GPU transfer)
    return (
        self.state_memory[indices],
        self.action_memory[indices],
        self.reward_memory[indices],
        self.next_state_memory[indices],
        self.done_memory[indices]
    )
```

- Next, we define the Agent, which learns the environment using the above networks
- The local Q network and the target Q network are initialized using the above QNetwork function. The action size, state size and the number of neurons for each layer of the network are passed as parameters.
- The memory array is initialized using the ReplayBuffer class
- The action is obtained using the ϵ-greedy and the Softmax policies. The action values are passed as a pytorch tensor.This is defined using the following code :

```
def get_action(self, action_values):
# Select an action based on the current policy (epsilon-greedy or softmax).
    if self.policy == 'epsilon_greedy':
        # With probability epsilon, choose random action (exploration)
        if np.random.random() < self.policy_param:
            return np.random.randint(0, self.action_size)
        else:
            # Otherwise choose best action (exploitation)
            return np.argmax(action_values.cpu().numpy())

    elif self.policy == 'softmax':
        # Softmax policy: choose actions probabilistically based on their Q-
                                                                values
        action_lis = action_values.cpu().numpy()[0]
        p_values = np.exp((action_lis-np.max(action_lis))/self.policy_param)
        sum_pi = sum(p_values)
        p_values /= sum_pi

    return np.random.choice(self.action_size , p=p_values)
```

- The ***step*** function is used to append the current state and reward values to the memory array. When the size of the memory array exceeds the batch size. the experiences are randomly sampled from it. Moreover, after every fixed number of steps (= update_every in the code) , the weights of the target network are updated to that of the current weights of the local network. This ensures that the target network moves slower compared to the local network.

```
def step(self,state,action,reward,next_state,done):
    self.memory.add(state=state,action=action,reward=reward,next_state=
                                                next_state,done=done)

    if self.memory.size >= self.batch_size:
        experiences = self.memory.sample()
        self.learn(experiences=experiences,gamma=self.discount_rate)

    self.steps=(self.steps + 1)%self.update_every

    if self.steps  == 0 :
        self.q_target.load_state_dict(self.q_local.state_dict())
```

- The ***act*** function is used to get the action values from the local network using the current network.

```
def act(self,state):
    state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
    self.q_local.eval()

    with torch.no_grad():
        action_values = self.q_local(state)
    self.q_local.train()
    action = self.get_action(action_values=action_values)
    return action
```

- The ***learn*** function is used to update the Q networks and update the network parameters using gradient descent. The loss is calculated between the expected Q network and the target Q network. The gradient are clipped between -1 and 1 to prevent from exploding gradients problem.

```
def learn(self,experiences,gamma):
    states , actions , rewards , next_states , dones = experiences
    states  = torch.from_numpy(states).float().to(self.device)
    actions = torch.from_numpy(actions).long().to(self.device)
    rewards = torch.from_numpy(rewards).float().to(self.device)
    next_states = torch.from_numpy(next_states).float().to(self.device)
    dones = torch.from_numpy(dones).float().to(self.device)

    actions_next = self.q_local(next_states).detach().max(1)[1].unsqueeze(1)
    q_targets_next = self.q_target(next_states).gather(1, actions_next)
    q_targets = rewards + gamma*q_targets_next*(1-dones)
    q_expected = self.q_local(states).gather(1, actions)
    loss = F.mse_loss(q_expected,q_targets)
    self.optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(self.q_local.parameters(), max_norm=1)
    self.optimizer.step()
```

- For training the agent, the average is taken over 3 experimemts with random seeds.

```
for episode in range(n_episodes):
    state , _ = env.reset()
    done = False
    score = 0
    while not done:
        action = agent.act(state)
        next_state , reward , terminated , truncated , _ = env.step(action)
        done = terminated or truncated
        agent.step(state , action , reward , next_state , done)
        state = next_state
        score += reward
    reward_avgs[i][episode] = score
    agent.decay()
```

3.2 Monte-Carlo REINFORCE

- The MC-REINFORCE algorithm utilizes the Monte-Carlo sampling to estimate the gradients for the policy optimization.
- Similar to the Dueling DQN, two types of updates are used here.

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \qquad \text{(w/o Baseline)}$$

  In the presence of baseline, $V(\cdot; \boldsymbol{\Phi})$, the update equation is given by

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha(G_t - V(S_t; \boldsymbol{\Phi})) \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})} \qquad \text{(w/ Baseline)}$$

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

- The Q-Network class is defined to initialize the policy and value networks. This part is similar to the Dueling DQN algorithm except that the final layer of the Policy network also has a softmax activation used to choose an action.

```
def forward(self , state):
    x = state
    for i in range(len(self.fc_lis)-1):
        x = F.relu(self.fc_lis[i](x))
    x = self.fc_lis[-1](x)
    if self.softmax_use:
        x = self.softmax(x)
    return x
```

- The Agent_MC class is defined to learn the environment using the above networks. Here, we initialize 2 optimizers and 2 learning rates, for the value network and the policy network, respectively.
- The ***get_action*** function gives the action value using the probabilities given by the policy network.

```python
def get_action(self, state):
    state = torch.tensor(state, dtype=torch.float32).to(self.device)
    state = torch.unsqueeze(state, 0)
    prob = self.policy_network.forward(state)
    prob = torch.squeeze(prob, 0)
    action = prob.multinomial(num_samples=1)
    action = action.item()
    return action
```

- The ***learn*** function implements the update step for the networks using gradient descent for the policy parameters. It first calculates the returns for each step of the episode.
  In the case with the Baseline, the model cumulatively updates the value function parameters using TD(0) , and uses the value function as the baseline for the returns, while updating the policy parameters. In this case, the value function parameters are updated after episode.
  Without the baseline, it uses only the returns for updating the policy parameters.
  For both of the above update methods, the policy parameters are updated cumulatively for each episode.

```python
def learn(self, state_lis, action_lis, reward_lis, gamma):
    states = torch.tensor(np.array(state_lis), dtype=torch.float32).to(self.
                                                device)
    actions = torch.tensor(np.array(action_lis), dtype=torch.long).to(self.
                                                device)  # Changed to long
    rewards = torch.tensor(np.array(reward_lis), dtype=torch.float32).to(self.
                                                device)

    # Calculate returns using vectorized operations
    returns = []
    G = 0.0
    for i, reward in enumerate(reversed(reward_lis)):
        G = reward + gamma * G
        returns.insert(0, G)
    returns = torch.tensor(returns, dtype=torch.float32).to(self.device)
    if self.baseline:
        # Value network update with TD(0)
        current_values = self.value_network(states).squeeze()
        # Calculate next state values
        next_states = [state_lis[i+1] if i < len(state_lis)-1 else None for i in
                                                    range(len(state_lis))]
        # Create mask and next values
        non_terminal_mask = torch.tensor(
            [ns is not None for ns in next_states],
            dtype=torch.bool
        ).to(self.device)
        next_values = torch.zeros(len(next_states)).to(self.device)
        if non_terminal_mask.any():
            non_terminal_next_states = torch.tensor(
                np.array([ns for ns in next_states if ns is not None]),
                dtype=torch.float32
            ).to(self.device)
            with torch.no_grad():
                next_values[non_terminal_mask] = self.value_network(
                    non_terminal_next_states
                ).squeeze()
        # Calculate TD targets
        td_targets = rewards + gamma * next_values
```

```
        # Value network update
        value_loss = F.mse_loss(current_values, td_targets.detach())
        self.optimizer2.zero_grad()
        value_loss.backward()
        self.optimizer2.step()
        current_values = self.value_network(states).squeeze()
        advantages = (returns - current_values.detach())
    else:
        advantages = returns
    # Policy network update
    action_probs = self.policy_network(states)
    selected_probs = action_probs.gather(1, actions.unsqueeze(1)).squeeze()
    policy_loss = (-torch.log(selected_probs + 1e-8) * advantages).mean()
    self.optimizer1.zero_grad()
    policy_loss.backward()
    torch.nn.utils.clip_grad_norm_(self.policy_network.parameters(), max_norm=1.
                                                                        0)
    self.optimizer1.step()
```

- The plots were calculated using the mean of the episode score and the rolling mean.

4 Values Used

- The parameters used in the code are :

Parameter	**Value**
Discount Factor	0.99
Algorithm	Dueling-DQN / MC-REINFORCE
Environment	CartPole-v1 / Acrobot-v1
Type	Type 1 / Type 2

Table 3: Experiment Parameters and Their Values

- The hyperparameters used are :

Hyperparameters
Learning Rate (Dueling-DQN)
Learning Rate 1 (MC-REINFORCE)
Learning Rate 2 (MC-REINFORCE)
Policy (Dueling-DQN)
Parameter Value (Dueling-DQN)
Final Parameter Value (Dueling-DQN)
Decay Parameter (Dueling-DQN)
Neural Network Layers (For both)
Batch Size (Dueling-DQN)
Buffer Size (Dueling-DQN)
Update Steps (Dueling-DQN)

Table 4: List of Hyperparameters Used

5 Results and Plots

- Comparative plot for the average reward and the rolling mean of the reward for the top 3 best hyperparameter values for each algorithm
- Comparative plot between the best hyperparameter value from Type 1 and Type 2 updates for both the algorithms (Dueling DQN and Monte-Carlo REINFORCE) for both the environments.
- The hyperparameters were optimized to obtain the minimum regret.
- The hyperparameters were obtained by performing a sweep on wandb within a range of values for each hyperparameter and using the **Bayes** method. The example code for the sweep is given below :

```
# Initialize sweep with data from .yaml file
with open("sweep.yaml", "r") as file:
    sweep_configuration = yaml.safe_load(file)
wandb.require("core")
# Getting arguments for entity and project name
parser = argparse.ArgumentParser(
    description="Training script that return model weights")
parser.add_argument(
    '-we',
    '--wandb_entity',
    type=str,
    default=None,
    help='Wandb Entity used to track experiments in the Weights & Biases
                                                dashboard')
parser.add_argument(
    '-wp',
    '--wandb_project',
    type=str,
    default=None,
    help='Project name used to track experiments in Weights & Biases dashboard')
parser.add_argument(
    '-c',
    '--count',
    type = int,
    default=100,
    help = 'Maximum number of sweep per agent')
args = parser.parse_args()
sweep_id = wandb.sweep(
    sweep_configuration,
    entity=args.wandb_entity,
    project=args.wandb_project)
# Initilizing agent
wandb.agent(sweep_id, count=args.count)
```

- The *yaml* file defines the parameter range for each hyperparameter values
- The 3 hyperparameters with the least regret were chosen for comparison, averaged over 5 random seeds.

5.1 Dueling-DQN

The two update rules for Dueling DQN are compared in this section. Type 1 uses the mean of the Action network and Type 2 uses the max of the Action network to update the Q-network.

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

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

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

5.2 Monte-Carlo REINFORCE

The two update rules for Monte-Carlo REINFORCE are compared in this section. The update Without Baseline considers only the Returns for update while the update With Baseline uses the Value function as the reference for the Returns while computing the update in the parameters. These updates are compared using the 3 best hyperparameter values for each environment

5.3 CartPole-v1

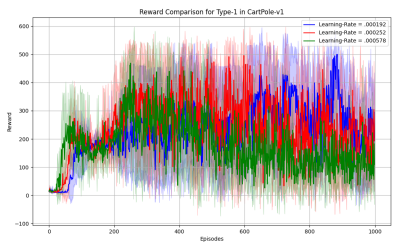
The CartPole-v1 runs for 500 steps per episode. The hyperparameters were tuned and the 3 best hyperparameter values with minimum regret are shown here.

5.3.1 Dueling-DQN

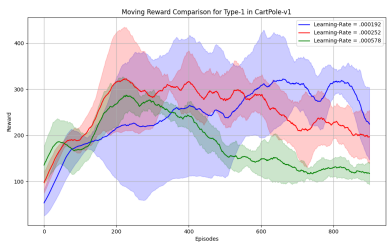
Type 1

Parameter	**Value**	**Parameter**	**Value**	**Parameter**	**Value**
Learning Rate	0.0003	Learning Rate	0.0002	Learning Rate	0.0006
Policy	'ϵ_greedy'	Policy	'ϵ_greedy'	Policy	'ϵ_greedy'
Parameter Value	0.405	Parameter Value	0.273	Parameter Value	0.401
Final Parameter Value	0.173	Final Parameter Value	0.138	Final Parameter Value	0.159
Decay Parameter	0.990	Decay Parameter	0.990	Decay Parameter	0.990
Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]
Batch Size	128	Batch Size	128	Batch Size	128
Buffer Size	1000000	Buffer Size	1000000	Buffer Size	1000000
Update Steps	200	Update Steps	200	Update Steps	200

Table 5: Hyperparameter values: Type 1



(a) Total Average Reward



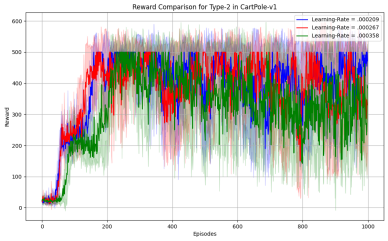
(b) Rolling mean reward

Figure 1: CartPole - Type 1

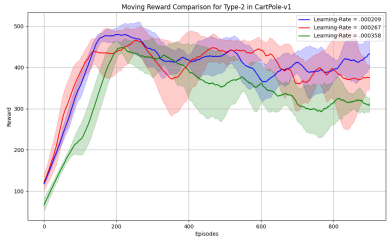
Type 2

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate	0.0002	Learning Rate	0.0003	Learning Rate	0.0004
Policy	'softmax'	Policy	'softmax'	Policy	'softmax'
Parameter Value	9.788	Parameter Value	9.113	Parameter Value	11.699
Final Parameter Value	0.411	Final Parameter Value	0.379	Final Parameter Value	0.752
Decay Parameter	0.954	Decay Parameter	0.959	Decay Parameter	0.963
Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]
Batch Size	128	Batch Size	128	Batch Size	128
Buffer Size	1000000	Buffer Size	1000000	Buffer Size	1000000
Update Steps	100	Update Steps	100	Update Steps	200

Table 6: Hyperparameter values: Type 2

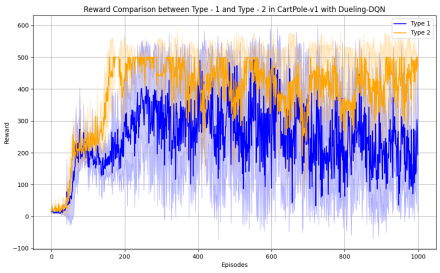


(a) Total Average Reward

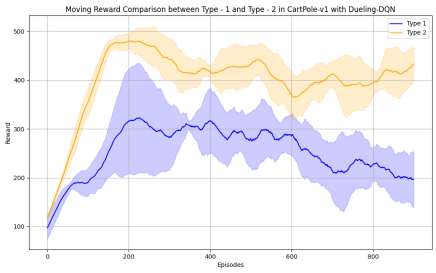


(b) Rolling mean reward

Figure 2: CartPole - Type 2

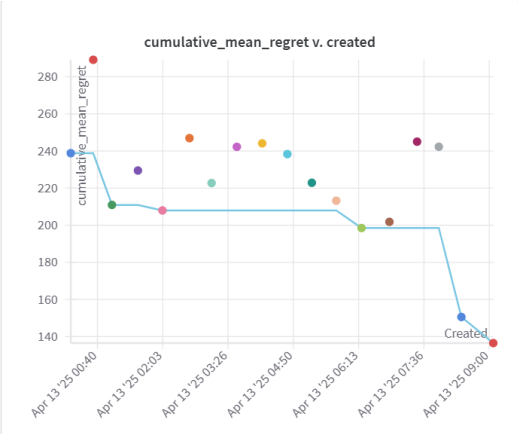


(a) Total Average Reward

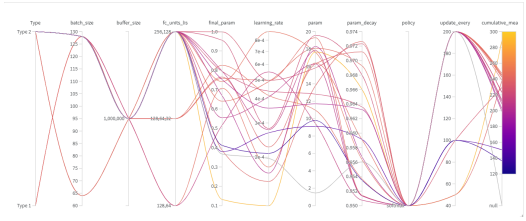


(b) Rolling Mean Reward

Figure 3: Type 1 vs Type 2 for Dueling-DQN CartPole



(a) Results



(b) Sweep

Figure 4: Results and Sweep Dueling-DQN CartPole

Inferences

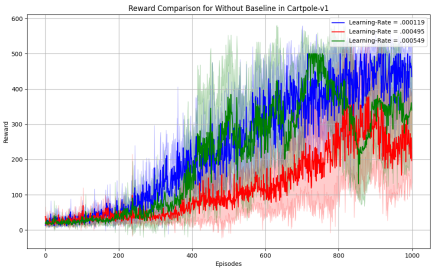
- Dueling-DQN for CartPole ran for 500 steps for each episode and was able to cross the threshold reward of 475 in about **100** episodes for Type 2 update.
- It took longer for DDQN to run to CartPole than other algorithms and environments because it was able to reach a reward of $\geq$ 400 and even 500 for a lot of episodes.
- Type 2 update seemed to give much better results compared to Type 1 for CartPole.It also converges much faster than Type 1 update.
- The possible reason for this is that for balancing the CartPole, using the maximum of the Action function ensures that we choose the best action and after a certain time doesn't need to explore.
- The variance for the results for Type 2 is also lesser than that for Type 1, which also owes to choosing the greedy action in this case.

5.3.2 Monte-Carlo REINFORCE

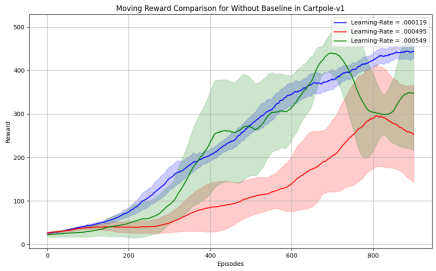
Without Baseline

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate 1	0.00012	Learning Rate 1	0.00049	Learning Rate 1	0.00055
Learning Rate 2	0.0131	Learning Rate 2	0.0001	Learning Rate 2	0.0001
Neural Network Layers	[256, 128]	Neural Network Layers	[128, 64]	Neural Network Layers	[256, 128]

Table 7: Hyperparameter values: Without Baseline



(a) Total Average Reward



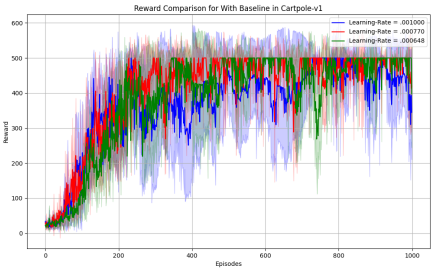
(b) Rolling mean reward

Figure 5: CartPole - Without Baseline

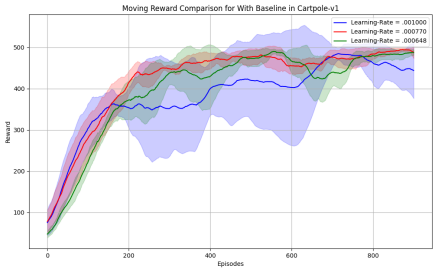
With Baseline

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate 1	0.0011	Learning Rate 1	0.0008	Learning Rate 1	0.0006
Learning Rate 2	0.0488	Learning Rate 2	0.0189	Learning Rate 2	0.0134
Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]	Neural Network Layers	[256, 128]

Table 8: Hyperparameter values: With Baseline

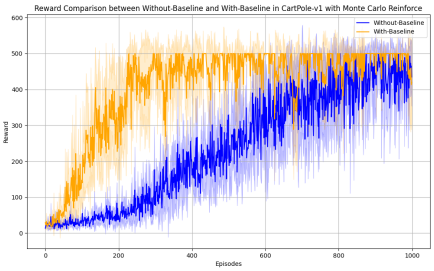


(a) Total Average Reward

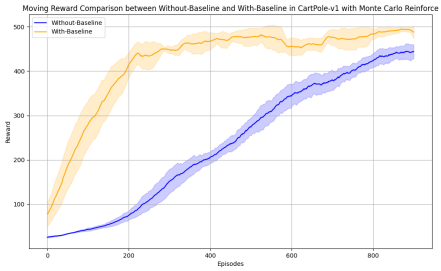


(b) Rolling mean reward

Figure 6: CartPole - With Baseline

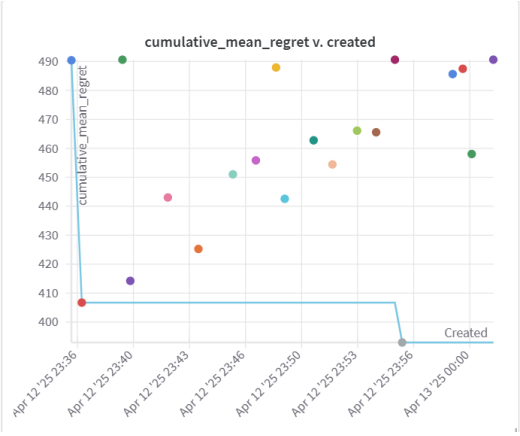


(a) Total Average Reward

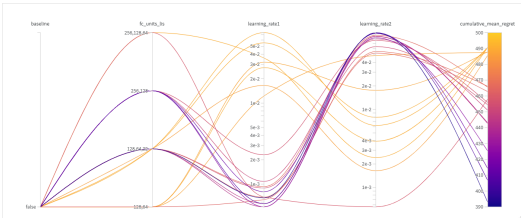


(b) Rolling Mean Reward

Figure 7: With Baseline vs Without Baseline MC-REINFORCE CartPole



(a) Results



(b) Sweep

Figure 8: Results and Sweep MC-REINFORCE CartPole

Inferences

- MC-REINFORCE for CartPole was faster in convergence compared to Dueling-DQN.
- For CartPole, the update with baseline gave significantly better results than without baseline.
- It also converged to the threshold reward of 475 in about 200 episodes.
- The variance for the results for with baseline is also slightly less than that for with, which is also due reduced variance in gradient updates.

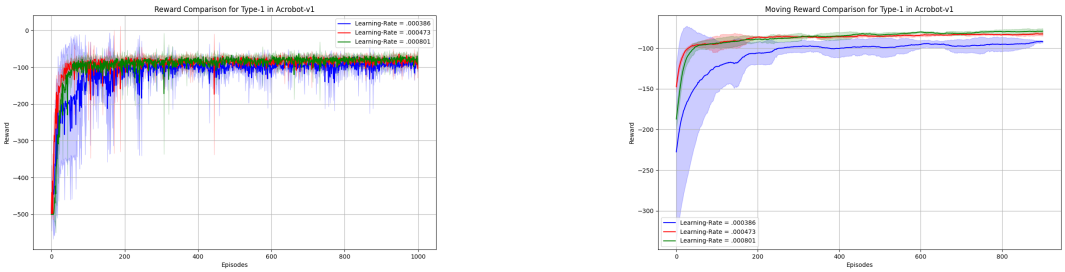
5.4 Acrobot-v1

5.4.1 Dueling-DQN

Type 1

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate	0.0005	Learning Rate	0.0004	Learning Rate	0.0008
Policy	'ϵ_greedy'	Policy	'ϵ_greedy'	Policy	'softmax'
Parameter Value	0.535	Parameter Value	0.054	Parameter Value	3.990
Final Parameter Value	0.032	Final Parameter Value	0.140	Final Parameter Value	0.141
Decay Parameter	0.978	Decay Parameter	0.960	Decay Parameter	0.950
Neural Network Layers	[128, 64]	Neural Network Layers	[128, 64, 32]	Neural Network Layers	[128, 64, 32]
Batch Size	256	Batch Size	256	Batch Size	128
Buffer Size	500000	Buffer Size	1000000	Buffer Size	1000000
Update Steps	100	Update Steps	100	Update Steps	100

Table 9: Hyperparameter values: Type 1



(a) Total Average Reward

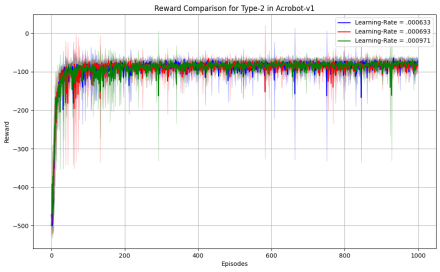
(b) Rolling mean reward

Figure 9: Acrobot - Type 1

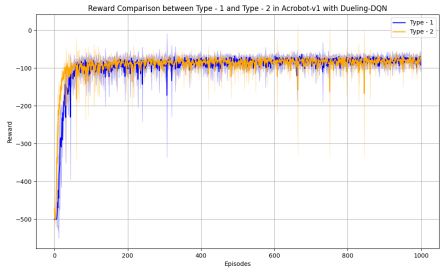
Type 2

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate	0.0006	Learning Rate	0.0007	Learning Rate	0.0010
Policy	'softmax'	Policy	'softmax'	Policy	'softmax'
Parameter Value	0.473	Parameter Value	0.495	Parameter Value	0.329
Final Parameter Value	0.126	Final Parameter Value	0.124	Final Parameter Value	0.123
Decay Parameter	0.963	Decay Parameter	0.962	Decay Parameter	0.964
Neural Network Layers	[128, 64, 32]	Neural Network Layers	[256, 128]	Neural Network Layers	[128, 64, 32]
Batch Size	128	Batch Size	128	Batch Size	128
Buffer Size	1000000	Buffer Size	1000000	Buffer Size	1000000
Update Steps	100	Update Steps	100	Update Steps	100

Table 10: Hyperparameter values: Type 2

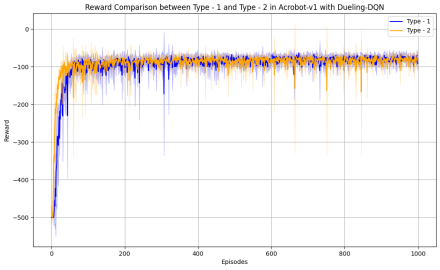


(a) Total Average Reward

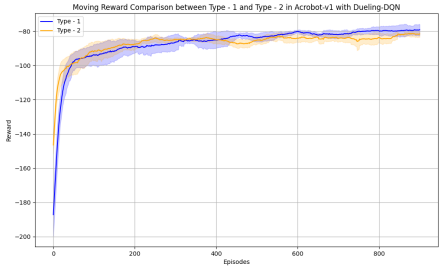


(b) Rolling mean reward

Figure 10: Acrobot - Type 2

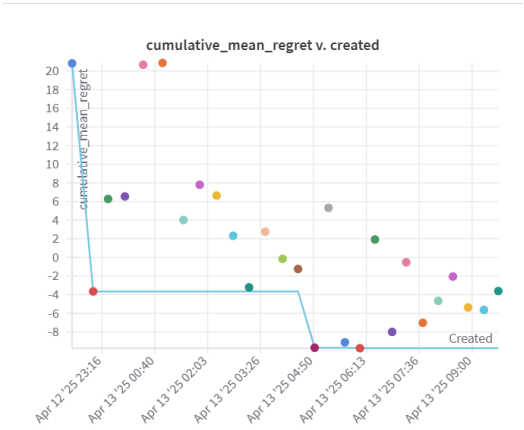


(a) Total Average Reward

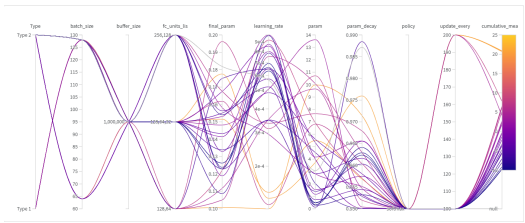


(b) Rolling Mean Reward

Figure 11: Type 1 vs Type 2 for Dueling-DQN Acrobot



(a) Results



(b) Sweep

Figure 12: Results and Sweep Dueling-DQN Acrobot

Inferences

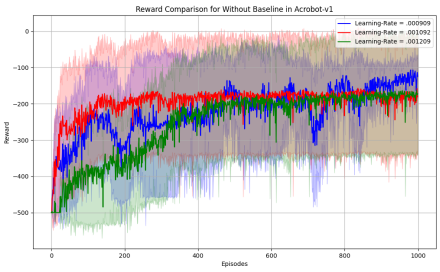
- Dueling-DQN for Acrobot was faster in convergence compared to the CartPole environment.
- For Acrobot, both the update rules seemed to give really good results and converged to the threshold reward of -100 fairly quickly in around 50-75 episodes.
- The variance for the results for Type 2 is slightly less than that for Type 1, which is also due to choosing the greedy action in this case.

5.4.2 Monte-Carlo REINFORCE

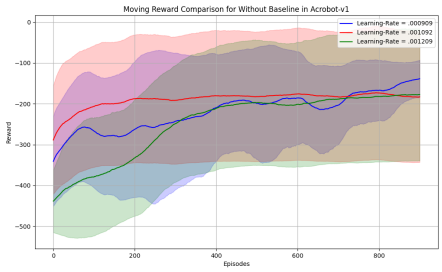
Without Baseline

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate 1	0.00109	Learning Rate 1	0.00121	Learning Rate 1	0.00091
Learning Rate 2	0.0168	Learning Rate 2	0.0231	Learning Rate 2	0.0620
Neural Network Layers	[256, 128]	Neural Network Layers	[128, 64]	Neural Network Layers	[256, 128]

Table 11: Hyperparameter values: Without Baseline



(a) Total Average Reward



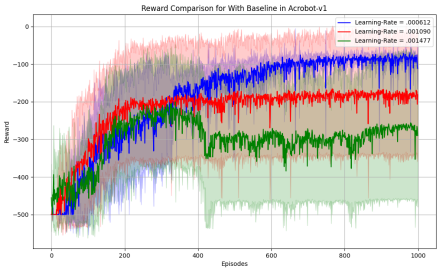
(b) Rolling Mean Reward

Figure 13: Acrobot - Without Baseline

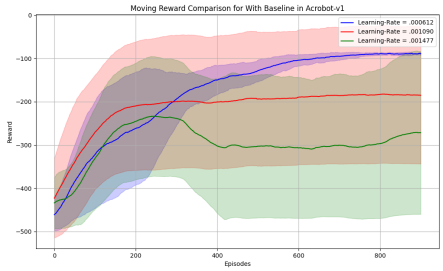
With Baseline

Parameter	Value	Parameter	Value	Parameter	Value
Learning Rate 1	0.00061	Learning Rate 1	0.00109	Learning Rate 1	0.00148
Learning Rate 2	0.0274	Learning Rate 2	0.0321	Learning Rate 2	0.0261
Neural Network Layers	[128, 64]	Neural Network Layers	[128, 64]	Neural Network Layers	[128, 64]

Table 12: Hyperparameter values: With Baseline

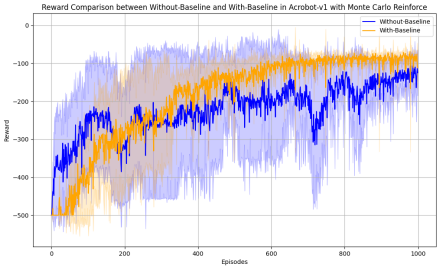


(a) Total Average Reward

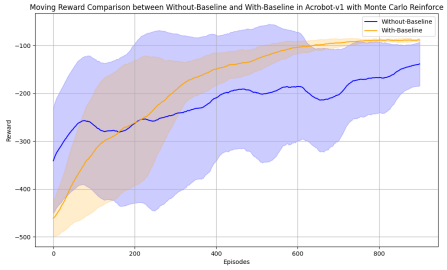


(b) Rolling mean reward

Figure 14: Acrobot - With Baseline

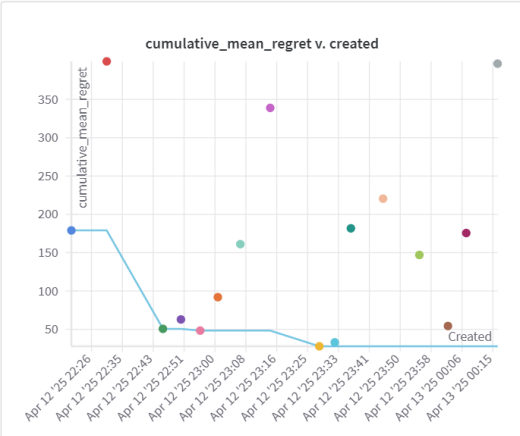


(a) Total Average Reward

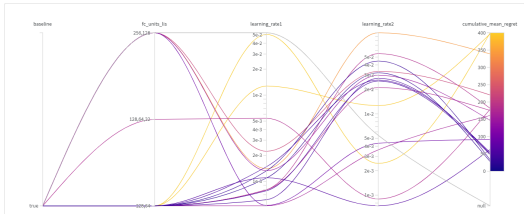


(b) Rolling Mean Reward

Figure 15: With Baseline vs Without Baseline MC-REINFORCE Acrobot



(a) Results



(b) Sweep

Figure 16: Results and Sweep MC-REINFORCE Acrobot

Inferences

- MC-REINFORCE for Acrobot was faster in convergence compared to Dueling-DQN.
- For Acrobot, the update with baseline gave significantly better results than without baseline.
- It also converged to the threshold reward of -100 in about 400 episodes, and also crosses it after that.
- The variance for the results for with baseline is also slightly less than that for with, which is also due to the reduced variance in gradient updates.
- It can also be noted that the variance of the reward averages decreases for both the updates as the episodes increase because the algorithms starts to learn the environment.

6 Link to Github

The code along with the README file, requirements.txt file and all the other essential files have been uploaded in this Github repo. The link to the Github classroom repo is attached here.

7 References

- Gymnasium Documentation
- Wandb Documentation

8 Acknowledgements

We would like to thank Prof.Balaram Ravindran and the Teaching Assistants for clearing our queries during the assignment.