CS6700: Reinforcement Learning

Programming Assignment 2

Shreya .S. Ramanujam, EE21B126
Srikar Babu Gadipudi, EE21B138

April 6, 2024

Contents

1 Dueling DQN **1**
- 1.1 CartPole-v1 . . . 6
- 1.2 Acrobot-v1 . . . 8
- 1.3 Inferences . . . 10
  - 1.3.1 CartPole-v1 . . . 10
  - 1.3.2 Acrobot-v1 . . . 11

2 Monte-Carlo REINFORCE **13**
- 2.1 CartPole-v1 . . . 17
- 2.2 Acrobot-v1 . . . 19
- 2.3 Inferences . . . 21
  - 2.3.1 CartPole-v1 . . . 21
  - 2.3.2 Acrobot-v1 . . . 22

3 GitHub Repository **22**

List of Figures

1 Dueling DQN Architecture . . . 1
2 Episodic Reward v/s Episodes for both Types on the CartPole environment using Dueling DQN algorithm averaged over 5 seeds . . . 8
3 Episodic Reward v/s Episodes for both Types on the Acrobot environment using Dueling DQN algorithm averaged over 5 seeds . . . 10
4 Episodic Reward v/s Episodes for both Types on the Acrobot environment using Dueling DQN algorithm averaged over 10 seeds . . . 12
5 Episodic Reward v/s Episodes for both variants on the CartPole environment using REINFORCE algorithm averaged over 5 seeds . . . 19
6 Episodic Reward v/s Episodes for both variants on the Acrobot environment using REINFORCE algorithm averaged over 5 seeds . . . 21

List of Tables

1 Configurations for Type 1 on CartPole environment . . . 7
2 Configurations for Type 2 on CartPole environment . . . 7
3 Configurations for Type 1 on Acrobot environment . . . 9
4 Configurations for Type 2 on Acrobot environment . . . 9
5 Configurations for without baseline variant on CartPole environment . . . 18
6 Configurations for with baseline variant on CartPole environment. The Critic Hidden Layer size is fixed at 16 neurons. . . . 18
7 Configurations for without baseline variant on Acrobot environment . . . 20
8 Configurations for with baseline variant on Acrobot environment. The Critic Hidden Layer size is fixed at 16 neurons. . . . 20

1 Dueling DQN

Regular DQN agents can struggle to separate the overall value of a state from the specific benefit of each action. Dueling DQN tackles this by introducing a new architecture.
Dueling DQN splits the Q-value prediction into two parts:

- State-value (V(s))
- Advantage (A(s, a))

The architecture includes a shared feature extractor that analyzes the state of the environment. Two streams then emerge out of this feature extractor, one for state-value function and one for action advantage. Finally, these outputs are merged using an aggregation layer to create the final Q-values.

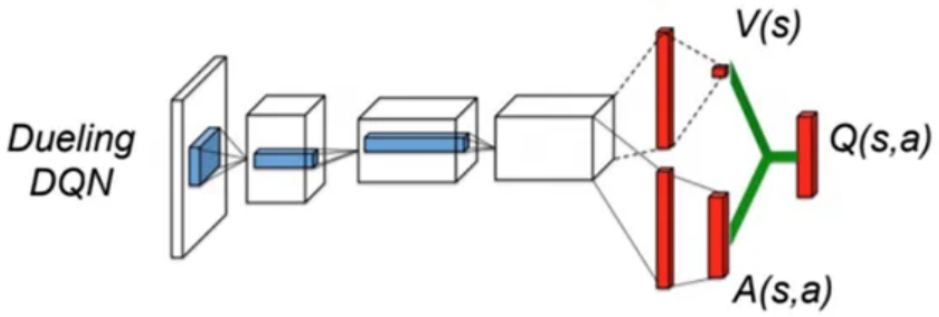

Figure 1: Dueling DQN Architecture

We consider two types of update equations, which are:

- **Type 1**:

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

- **Type 2**:

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

where θ corresponds to the shared network parameters and α, β corresponds to the parameters corresponding to state-value function estimation and action advantage function respectively. These equations essentially form two types of aggregation layers.

The parameters (θ, α and β) are all updated using the TD(0) update equation.

For all the experiments performed below, we considered the following architecture

- One hidden layer for the shared network with 64 neurons and ReLU activation function.
- One hidden layer for the state value function with 256 neurons and ReLU activation function.
- And finally, one hidden layer for action advantage function with 256 neurons and ReLU activation function.

For the target network updation we considered both copying weights and Polyak averaging. We observed the best performance when copying the weights of the local network to the target network at every 20 steps. Also, we observed the best performance when we used a batch size of 64.

For all the experiments performed, we consider a discount factor (γ) of 0.99, as instructed.

```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
# device = "cpu"
print(device)

class Memory:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = deque(maxlen=max_size)
        # self.seed = seed
        # random.seed(self.seed)

    def push(self, state, action, reward, next_state, done):
        experience = (state, action, np.array([reward]), next_state, done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        reward_batch = []
        next_state_batch = []
        done_batch = []

        # random.seed(self.seed)
        batch = random.sample(self.buffer, batch_size)

        for experience in batch:
            state, action, reward, next_state, done = experience
            state_batch.append(state)
            action_batch.append([action])
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append([done])

        return np.array(state_batch), np.array(action_batch),
        ↪  np.array(reward_batch), np.array(next_state_batch),
        ↪  np.array(done_batch)

    def __len__(self):
        return len(self.buffer)
```

Listing 1: Code snippet for the Replay Buffer

```python
class Dueling_DQN_T1(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim_common = 64, hidden_dim_adv = 256, hidden_dim_val = 256):
        super(Dueling_DQN_T1, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim_common = hidden_dim_common
        self.hidden_dim_adv = hidden_dim_adv
        self.hidden_dim_val = hidden_dim_val

        self.fc_common = nn.Linear(self.input_dim, self.hidden_dim_common)
        self.fc_val = nn.Linear(self.hidden_dim_common, self.hidden_dim_val)
        self.fc_adv = nn.Linear(self.hidden_dim_common, self.hidden_dim_adv)

        self.value_layer = nn.Linear(self.hidden_dim_val, 1)
        self.adv_layer = nn.Linear(self.hidden_dim_adv, self.output_dim)

    def forward(self, x):
        x = F.relu(self.fc_common(x))
        self.advantage = self.adv_layer(F.relu(self.fc_adv(x)))
        self.value = self.value_layer(F.relu(self.fc_val(x)))

        self.q = self.value + self.advantage - self.advantage.mean() # type 1

        return self.q

class Dueling_DQN_T2(nn.Module):
    def __init__(self, input_dim, output_dim, hidden_dim_common = 64, hidden_dim_adv = 256, hidden_dim_val = 256):
        super(Dueling_DQN_T2, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.hidden_dim_common = hidden_dim_common
        self.hidden_dim_adv = hidden_dim_adv
        self.hidden_dim_val = hidden_dim_val

        self.fc_common = nn.Linear(self.input_dim, self.hidden_dim_common)
        self.fc_val = nn.Linear(self.hidden_dim_common, self.hidden_dim_val)
        self.fc_adv = nn.Linear(self.hidden_dim_common, self.hidden_dim_adv)

        self.value_layer = nn.Linear(self.hidden_dim_val, 1)
        self.adv_layer = nn.Linear(self.hidden_dim_adv, self.output_dim)
```

```
43
44      def forward(self, x):
45          x = F.relu(self.fc_common(x))
46          self.advantage = self.adv_layer(F.relu(self.fc_adv(x)))
47          self.value = self.value_layer(F.relu(self.fc_val(x)))
48
49          self.q = self.value + self.advantage - torch.max(self.advantage) #
            ↪  type 2
50
51          return self.q
```

Listing 2: Code snippet for the Type 1 and Type 2 Neural Networks

```
1  # The same code with T2 instead of T1 corresponds to the agent of Type 2
2  class Dueling_DQN_Agent_T1:
3      def __init__(self, env, learning_rate=1e-4, gamma=0.99, tau=1e-2,
       ↪  max_memory_size=50000, seed = 0):
4
5          self.num_states = env.observation_space.shape[0]
6          self.num_actions = env.action_space.n
7          self.gamma = gamma
8          self.tau = tau
9          self.seed = seed
10         torch.manual_seed(self.seed)
11         torch.cuda.manual_seed(self.seed)
12         np.random.seed(self.seed)
13         random.seed(self.seed)
14
15         self.Q = Dueling_DQN_T1(self.num_states,
           ↪  self.num_actions).to(device)
16         self.Q_target = Dueling_DQN_T1(self.num_states,
           ↪  self.num_actions).to(device)
17         self.optimizer = torch.optim.Adam(self.Q.parameters(),
           ↪  lr=learning_rate)
18
19         self.memory = Memory(max_memory_size)
20
21         for target_param, param in zip(self.Q_target.parameters(),
           ↪  self.Q.parameters()):
22             target_param.data.copy_(param.data)
23             target_param.requires_grad = False
24
25         self.loss_criterion  = nn.MSELoss()
26
27     def get_action(self, state, epsilon, policy = choose_action_softmax):
28         q_values = self.Q(torch.Tensor(state).to(device))
29         return policy(q_values, self.num_actions, epsilon)
30
31     def update(self, batch_size, episode):
```

```
        states, actions, rewards, next_states, dones =
        ↪ self.memory.sample(batch_size)
        states = torch.FloatTensor(states).to(device)
        actions = torch.FloatTensor(actions).to(device)
        rewards = torch.FloatTensor(rewards).to(device)
        next_states = torch.FloatTensor(next_states).to(device)
        dones =
        ↪ torch.FloatTensor(np.array(dones).astype(np.uint8)).to(device)

        self.optimizer.zero_grad()
        with torch.no_grad():
            Q_targets_next =
            ↪ self.Q_target(next_states).detach().max(1)[0].unsqueeze(1)
            Q_targets = rewards + (self.gamma * Q_targets_next * (1 -
            ↪ dones))

        Q_expected = self.Q(states).gather(1, actions.long())

        loss = self.loss_criterion(Q_targets, Q_expected)
        loss.backward()
        self.optimizer.step()

        # copying weights from Q to Q_target
        if episode % 20 == 0:
            for target_param, param in zip(self.Q_target.parameters(),
            ↪ self.Q.parameters()):
                target_param.data.copy_(param.data)

        # polyak averaging
        # for target_param, param in zip(self.Q_target.parameters(),
        ↪ self.Q.parameters()):
        #     target_param.data.copy_(self.tau*param.data +
        ↪ (1-self.tau)*target_param.data)
```

Listing 3: Code snippet for the Agent

The agent in 3, corresponds to the agent of Type 1. The exact same agent is used for the agent of Type 2, where the network used is 'Dueling_DQN_T2', from 4, for both local and target networks.
The training loop used for training agents of both types is as follows

```
def train(env, agent, num_episodes, batch_size, env_threshold, epsilon =
↪ 1.0, epsilon_min = 0.01, epsilon_decay = 0.995, seed = 0, policy =
↪ choose_action_softmax):
    rewards = []
    scores_window = deque(maxlen=100)
    eps = epsilon

    for episode in range(num_episodes):
        state = env.reset(seed=seed)
```

```
        score = 0
        for i in range(500):
            action = agent.get_action(state, eps, policy = policy)
            next_state, reward, done, _ = env.step(action)
            agent.memory.push(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if len(agent.memory) > batch_size:
                agent.update(batch_size, episode)
            if done:
                break

        scores_window.append(score)
        rewards.append(score)

        eps = max(epsilon_min, epsilon_decay*eps)

        print('\rEpisode {}\tScore: {:.2f}'.format(episode, score), end="")

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

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

    return rewards
```

Listing 4: Code snippet for the training loop

A maximum of 500 timesteps is considered for both CartPole and Acrobot environments because this is the maximum number of timesteps to be considered for an episode as per documentation.

1.1 CartPole-v1

This environment is part of the Classic Control environments which contains general information about the environment. A pole is attached by an unactuated 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.

The state at every timestep is a ndarray with shape (4,) with the values corresponding to the following positions and velocities:

- Cart Position
- Cart Velocity

- Pole Angle
- Pole Angular Velocity

The action is a ndarray with shape (1,) which can take values 0, 1 indicating the direction of the fixed force the cart is pushed with.

- 0: Push cart to the left
- 1: Push cart to the right

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 following configurations of the previously mentioned model have been tried out for both Type 1 and Type 2 updates.

Policy	**Epsilon/Tau**	**Epsilon min**	**Epsilon decay**	**Learning Rate**	**Regret**
Softmax	1	-	-	1.00E-04	160185.8
Softmax	1	-	-	1.00E-03	166871.2
Epsilon Greedy	0.01	0.0001	0.99	1.00E-03	245299.2
Epsilon Greedy	0.1	0.0001	0.99	1.00E-03	229362.8

Table 1: Configurations for Type 1 on CartPole environment

Policy	**Epsilon/Tau**	**Epsilon min**	**Epsilon decay**	**Learning Rate**	**Regret**
Softmax	1	-	-	1.00E-04	167370.2
Softmax	1	-	-	1.00E-03	202587.2
Epsilon Greedy	0.01	0.0001	0.99	1.00E-03	223633.8
Epsilon Greedy	0.1	0.0001	0.99	1.00E-03	181109.8

Table 2: Configurations for Type 2 on CartPole environment

Here regret is defined as the sum of difference between the optimal reward (500) that can be obtained in an episode and the actual episodic rewards observed. These values are reported after averaging over 5 seeds to reduce fluctuations due to stochasticity in the environment and the algorithm. We considered a total of 500 episodes in all experiments.

Observing Tables 1 and 2, we compare the episodic reward plots obtained for softmax selection policy with $\tau = 1$ with no decay and learning rate of 10^{-4} for both Types averaged over 5 seeds. The error bar indicates one standard deviation difference.

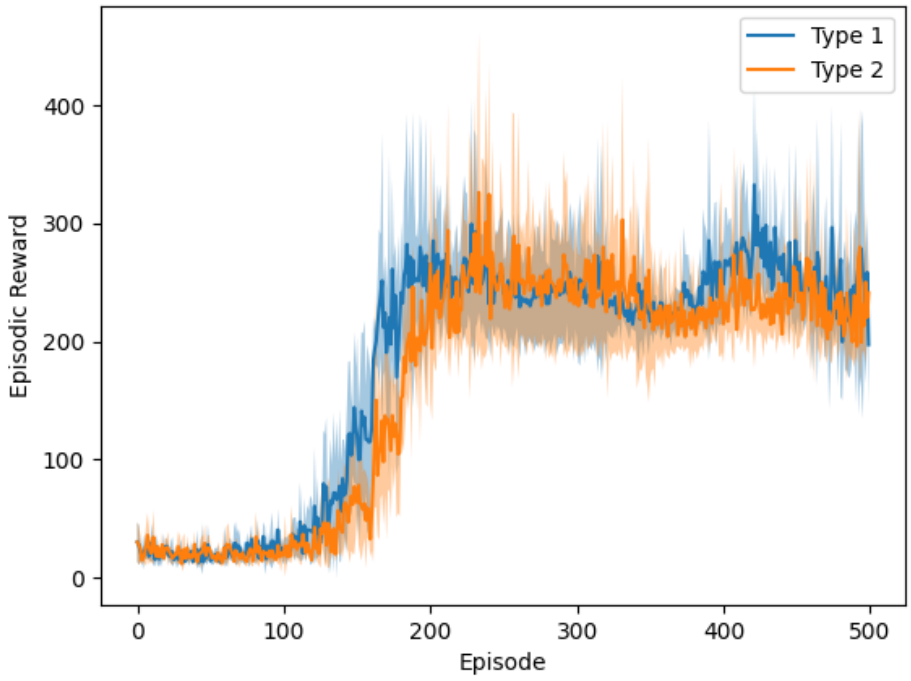

Figure 2: Episodic Reward v/s Episodes for both Types on the CartPole environment using Dueling DQN algorithm averaged over 5 seeds

1.2 Acrobot-v1

This environment is part of the Classic Control environments which contains general information about the environment. 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.

The observation is a ndarray with shape (6,) that provides information about the two rotational joint angles as well as their angular velocities:

- Cosine of θ_1
- Sine of θ_1
- Cosine of θ_2
- Sine of θ_2
- Angular velocity of θ_1
- Angular velocity of θ_2

where θ_1 is the angle of the first joint and θ_2 is relative to the angle of the first link.

The action is discrete, deterministic and represents the torque applied on the actuated joint between the two links.

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 following configurations of the previously mentioned model have been tried out for both Type 1 and Type 2 updates.

Policy	**Epsilon/Tau**	**Epsilon min**	**Epsilon decay**	**Learning Rate**	**Regret**
Softmax	1	-	-	1.00E-03	239227.8
Epsilon Greedy	0.01	0.0001	0.99	1.00E-03	221793.4
Epsilon Greedy	0.1	0.0001	0.99	1.00E-03	196055

Table 3: Configurations for Type 1 on Acrobot environment

Policy	**Epsilon/Tau**	**Epsilon min**	**Epsilon decay**	**Learning Rate**	**Regret**
Softmax	1	-	-	1.00E-03	252094.8
Epsilon Greedy	0.01	0.0001	0.99	1.00E-03	260968.2
Epsilon Greedy	0.1	0.0001	0.99	1.00E-03	259550.6

Table 4: Configurations for Type 2 on Acrobot environment

Here regret is defined as the sum of difference between the optimal reward (0) that can be obtained in an episode and the actual episodic rewards observed. These values are reported after averaging over 5 seeds to reduce fluctuations due to stochasticity in the environment and the algorithm. We considered a total of 1000 episodes in all experiments.

Observing Tables 3 and 4, we compare the episodic reward plots obtained for softmax selection policy with $\tau = 1$ with no decay and learning rate of 10^{-3} for both Types averaged over 5 seeds. The error bar indicates one standard deviation difference.

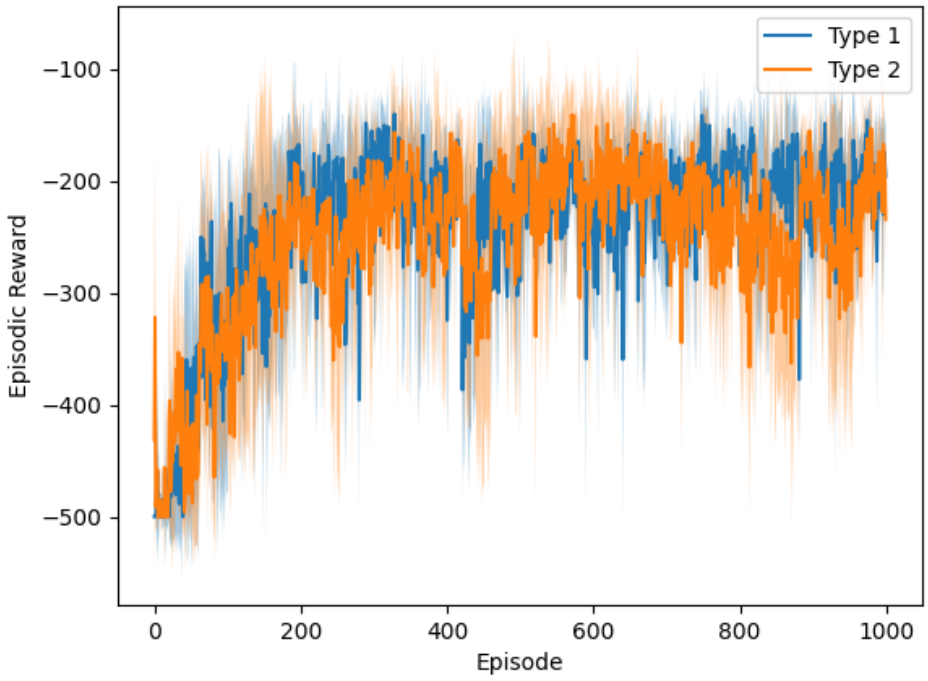

Figure 3: Episodic Reward v/s Episodes for both Types on the Acrobot environment using Dueling DQN algorithm averaged over 5 seeds

1.3 Inferences

1.3.1 CartPole-v1

We conclude the following observations from the experiments performed on the CartPole-v1 environment.

- From Tables 1 and 2, we observe in general softmax selection policy performs better than the epsilon greedy selection policy. But there are some edge cases where the epsilon greedy selection policy surpasses the performance of the softmax selection policy when appropriate epsilon, epsilon decay and minimum epsilon values are considered.
- From the plot 2 we observe that the variance in the learning curves of both types is nearly the same. We also observe that Type 1 formulation converges sooner than Type 2 formulation.
- Both types converge to an average episodic reward of approximately 250. We know that the CartPole environment is considered to be solved when the running average over the past 100 episodes is greater than 195. Therefore, we can conclude this environment to be solved with both types.
- From the plot 2 and the values of regret noted in Tables 1 and 2, we can infer that Type 1 formulation (subtracting average advantage) performs slightly better than Type 2 formulation (subtracting maximum advantage).
- We hypothesize this happens because subtracting the average advantage removes a common bias from all Q-values. By removing this bias, the remaining advantage

estimates become more focused on the relative value of each action compared to others in the same state. Whereas in the case of subtracting the maximum advantage, if one action consistently has a slightly higher advantage estimate than others, even if the difference isn't significant, subtracting the maximum will push all other Q-values down disproportionately. This can lead to the agent neglecting potentially good actions because their advantage estimates appear much lower.

- Due to the large number of parameters to be estimated every training loop takes approximately 20 minutes for both types.
- We can obtain smoother learning curves if we average over more number of seeds, this essentially averages over the stochasticity present in the algorithm.

1.3.2 Acrobot-v1

We conclude the following observations from the experiments performed on the Acrobot-v1 environment.

- From Tables 3 and 4, we observe in general softmax selection policy performs better than the epsilon greedy selection policy. But there are some edge cases where the epsilon greedy selection policy surpasses the performance of the softmax selection policy when appropriate epsilon decay and minimum epsilon values are considered.
- From the plot 3, we observe similar variances in the learning curve, although averaging over a higher number of seeds would give us a better understanding of the exact performance variance in both types.
- From the Tables 3 and 4 we observe that Type 1 performs slightly better than Type 2 as seen from the accumulated value of regret. This is as expected as Type 1 formulation removes bias, whereas Type 2 formulation might lead to overestimation or underestimation of the Q values. Refer to the explanation mentioned above (in the CartPole-v1 inferences subsection).
- Being a complex environment to solve, we considered a total number of 1000 episodes and observed that it takes around 600 episodes for the algorithm for both types to converge.
- Due to the large of parameters to be estimated every training loop takes approximately 30-40 minutes for both types.
- We can obtain smoother learning curves if we average over more number of seeds, this essentially averages over the stochasticity present in the algorithm. For example, when we average the results for 10 seeds instead of 5 we get Figure 4 which slightly reduces the stochasticity of the learning curve.

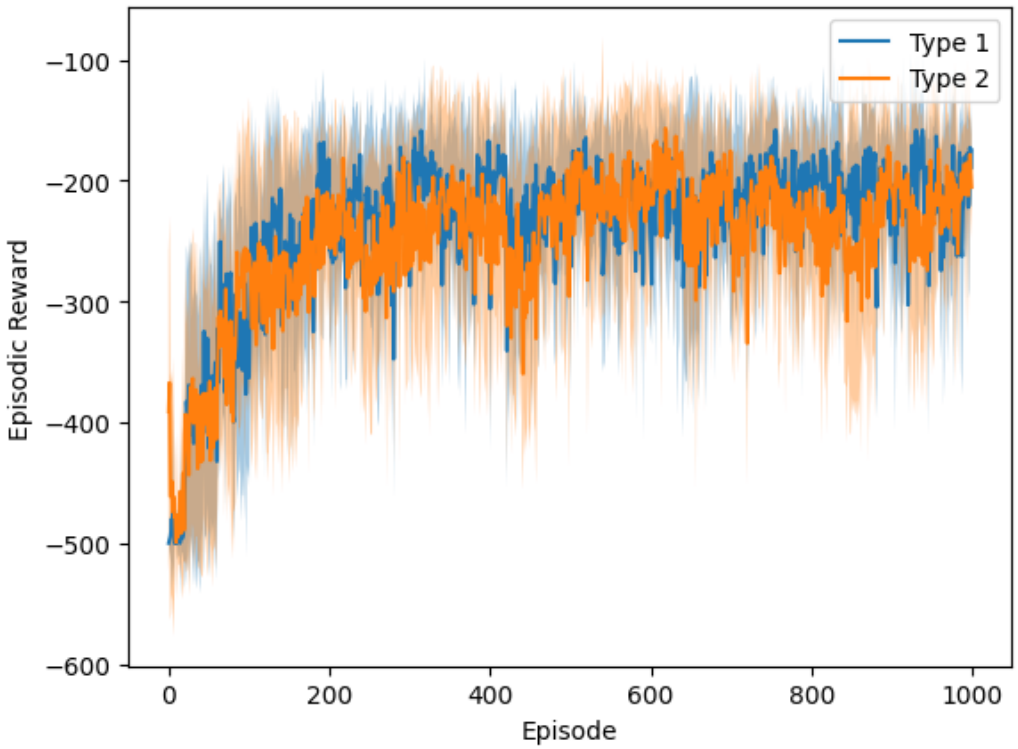

Figure 4: Episodic Reward v/s Episodes for both Types on the Acrobot environment using Dueling DQN algorithm averaged over 10 seeds

2 Monte-Carlo REINFORCE

The Monte-Carlo REINFORCE algorithm is a fundamental technique in reinforcement learning (RL). It belongs to a class of algorithms called policy gradient methods. REINFORCE aims to improve this policy by rewarding actions that lead to higher returns (sum of future rewards) over time. The policy is parametrized using various functions, typically neural networks.

We consider the following update equations for the parameters of the policy:

- **Without Baseline**:

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

- **With Baseline**:

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

Here, $V(S_t; \phi)$ is the state-value function. We use a neural network (Critic network) to obtain this state value function. The critic neural network parameters are updated using the regular TD update equation.

We consider a one-hidden-layer neural network to represent the policy $\pi(A_t|S_t, \theta)$ (the number of neurons used in the hidden layer is varied). We also considered a neural network with one hidden layer consisting of 16 neurons for the critic network (for the with baseline case). The activation function used in all the hidden layers is ReLU activation.

For all the experiments performed, we consider a discount factor (γ) of 0.99 and average the rewards over 5 seeds, as instructed.

The following code is used for parametrizing the policy in both cases using a neural network with the architecture mentioned above.

```
class ReinforcePolicyT1(nn.Module):
  def __init__(self, input_dim, num_actions, hidden_dim = 32):
    super(ReinforcePolicyT1, self).__init__()
    self.input_dim = input_dim
    self.num_actions = num_actions
    self.hidden_dim = hidden_dim
    self.fc1 = nn.Linear(self.input_dim, self.hidden_dim)
    self.fc2 = nn.Linear(self.hidden_dim, self.num_actions)

  def forward(self, x):
    x1 = F.relu(self.fc1(x))
    x2 = F.softmax(self.fc2(x1), dim=0)
    return x2

  def get_action(self, probs):
    action = np.random.choice([i for i in range(self.num_actions)],
    ↪  p=probs.detach().numpy())
```

```
    log_prob = torch.log(probs[action])
    return action, log_prob
```

Listing 5: Code snippet for the Policy Network

```
class ReinforceAgentT1:
  def __init__(self, env, gamma = 0.99, learning_rate = 1e-3, seed = 0,
  ↪  hidden_dim = 64):
    self.num_states = env.observation_space.shape[0]
    self.num_actions = env.action_space.n
    self.hidden_dim = hidden_dim
    self.policy = ReinforcePolicyT1(self.num_states, self.num_actions,
    ↪  self.hidden_dim)
    self.learning_rate = learning_rate
    self.gamma = gamma
    self.seed = seed
    self.optimizer = torch.optim.Adam(self.policy.parameters(),
    ↪  lr=self.learning_rate)
    torch.manual_seed(self.seed)
    torch.cuda.manual_seed(self.seed)
    np.random.seed(self.seed)
    random.seed(self.seed)

  def update(self, rewards_list, log_prob_list):
    score = 0
    rev_rewards_list = rewards_list[::-1]
    num_steps = len(rewards_list)
    discounted_rewards = []

    for i in range(num_steps):
      score = score*self.gamma + rev_rewards_list[i]
      discounted_rewards.append(score)

    discounted_rewards = torch.tensor(discounted_rewards[::-1])
    discounted_rewards = (discounted_rewards - discounted_rewards.mean()) /
    ↪  (discounted_rewards.std() + eps)

    loss_list = []
    for log_prob, Gt in zip(log_prob_list, discounted_rewards):
        loss_list.append(-log_prob * Gt)

    loss = torch.stack(loss_list).sum()
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

Listing 6: Code snippet for the Agent following without baseline update

```
class Critic(nn.Module):
  def __init__(self, input_dim, hidden_dim = 16): #16
```

```
    super(Critic, self).__init__()
    self.input_dim = input_dim
    self.output_dim = 1
    self.hidden_dim = hidden_dim
    self.fc1 = nn.Linear(self.input_dim, self.hidden_dim)
    self.fc2 = nn.Linear(self.hidden_dim, self.output_dim)

  def forward(self, x):
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x # returns value of that state
```

Listing 7: Code snippet for the Critic Network

```
class ReinforceAgentT2:
  def __init__(self, env, gamma = 0.99, learning_rate_actor = 1e-2,
  ↪  learning_rate_critic = 1e-2, seed = 0):
    self.num_states = env.observation_space.shape[0]
    self.num_actions = env.action_space.n
    self.policy = ReinforcePolicyT2(self.num_states, self.num_actions)
    self.critic = Critic(self.num_states)
    self.learning_rate_actor = learning_rate_actor
    self.learning_rate_critic = learning_rate_critic
    self.gamma = gamma
    self.seed = seed
    self.optimizer_actor = torch.optim.Adam(self.policy.parameters(),
    ↪  lr=self.learning_rate_actor)
    self.optimizer_critic = torch.optim.Adam(self.critic.parameters(),
    ↪  lr=self.learning_rate_critic)
    torch.manual_seed(self.seed)
    torch.cuda.manual_seed(self.seed)
    np.random.seed(self.seed)
    random.seed(self.seed)

  def update(self, rewards_list, log_prob_list, value_list,
  ↪  next_values_list):

    score = 0
    rev_rewards_list = rewards_list[::-1]
    num_steps = len(rewards_list)
    discounted_rewards = []
    delta_list = []

    for i in range(num_steps):
      score = score*self.gamma + rev_rewards_list[i]
      discounted_rewards.append(score)

    discounted_rewards = torch.tensor(discounted_rewards[::-1])
    discounted_rewards = (discounted_rewards - discounted_rewards.mean()) /
    ↪  (discounted_rewards.std() + eps)
```

```
32
33      actor_loss_list = []
34      for log_prob, Gt, value in zip(log_prob_list, discounted_rewards,
        ↪  value_list):
35          actor_loss_list.append(-log_prob * (Gt - value))
36
37      actor_loss = torch.stack(actor_loss_list).sum()
38      self.optimizer_actor.zero_grad()
39      actor_loss.backward(retain_graph = True)
40      self.optimizer_actor.step()
41
42      target = []
43      with torch.no_grad():
44        for i in range(num_steps):
45          target.append(rewards_list[i] + self.gamma*next_values_list[i])
46      target = torch.tensor(target, requires_grad = False)
47      value_list = torch.tensor(value_list, requires_grad = True)
48      critic_loss = F.mse_loss(value_list, target)
49      self.optimizer_critic.zero_grad()
50      critic_loss.backward(retain_graph = True)
51      self.optimizer_critic.step()
```

Listing 8: Code snippet for the Agent following with baseline update

The training loop used for training agents of both variants is same/similar, which is as follows

```
1  def train(env, agent, num_episodes = 1000, seed = 0, max_steps = 500):
2    rewards_per_ep = []
3    scores_window = deque(maxlen=20)
4
5    for episode in range(num_episodes):
6      state = env.reset(seed = seed)
7      log_prob_list = []
8      rewards_list = []
9      value_list =[]
10     next_state_values = []
11
12     for i in range(max_steps):
13       probs = agent.policy(torch.Tensor(state))
14       action, log_prob = agent.policy.get_action(probs)
15       value = agent.critic(torch.Tensor(state))
16       next_state, reward, done, _ = env.step(action)
17       next_value = agent.critic(torch.Tensor(next_state))
18       rewards_list.append(reward)
19       log_prob_list.append(log_prob)
20       value_list.append(value)
21       next_state_values.append(next_value)
22       state = next_state
23
```

```
      if done:
        break

    total_reward = sum(rewards_list)
    rewards_per_ep.append(total_reward)
    agent.update(rewards_list, log_prob_list, value_list,
    ↪  next_state_values)
    scores_window.append(total_reward)

    print('\rEpisode {}\tScore: {:.2f}'.format(episode, total_reward),
    ↪  end="")

    if episode % 20 == 0:
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(episode,
        ↪  np.mean(scores_window)))

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

  return rewards_per_ep
```

Listing 9: Code snippet for the training loop

A maximum of 500 timesteps is considered for both CartPole and Acrobot environments because this is the maximum number of timesteps to be considered for an episode as per documentation.

2.1 CartPole-v1

This environment is part of the Classic Control environments which contains general information about the environment. A pole is attached by an unactuated 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.

The state at every timestep is a ndarray with shape (4,) with the values corresponding to the following positions and velocities:

- Cart Position
- Cart Velocity
- Pole Angle
- Pole Angular Velocity

The action is a ndarray with shape (1,) which can take values 0, 1 indicating the direction of the fixed force the cart is pushed with.

- 0: Push cart to the left

- 1: Push cart to the right

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.

Some configurations of the previously mentioned model that have been tried out for both variants are given below.

Policy Hidden Layer neurons	**Learning Rate**	**Regret**
64	1.00E-03	224202.6
32	5.00E-03	96623.0

Table 5: Configurations for without baseline variant on CartPole environment

Policy HL neurons	**Policy Learning Rate**	**Critic Learning Rate**	**Regret**
24	1.00E-03	1.00E-02	299171.4
32	1.00E-03	1.00E-02	340071.6
24	1.00E-02	1.00E-02	122451.2
32	1.00E-02	1.00E-02	114714.8
32	1.00E-02	5.00E-03	65617.4

Table 6: Configurations for with baseline variant on CartPole environment. The Critic Hidden Layer size is fixed at 16 neurons.

Here regret is defined as the sum of difference between the optimal reward (500) that can be obtained in an episode and the actual episodic rewards observed. These values are reported after averaging over 5 seeds to reduce fluctuations due to stochasticity in the environment and the algorithm. In all experiments, we consider the number of episodes up until convergence is observed. In both tables, only the last row has been run for 500 episodes, and the remaining configurations were run for 1000 episodes.

Observing Tables 5 and 6 and the learning curves, we pick the least regret configurations for both variants to compare. We compare the episodic rewards v/s number of episodes for the case with 32 neurons in the policy network with 5×10^{-3} as the learning rate for the first variant (without baseline) and the case with 32 neurons in the policy network with 10^{-2} as the actor's learning rate and 5×10^{-3} as the critic's learning rate. The resulting curve is averaged over 5 seeds and the error bar in the plot indicates one standard deviation difference.

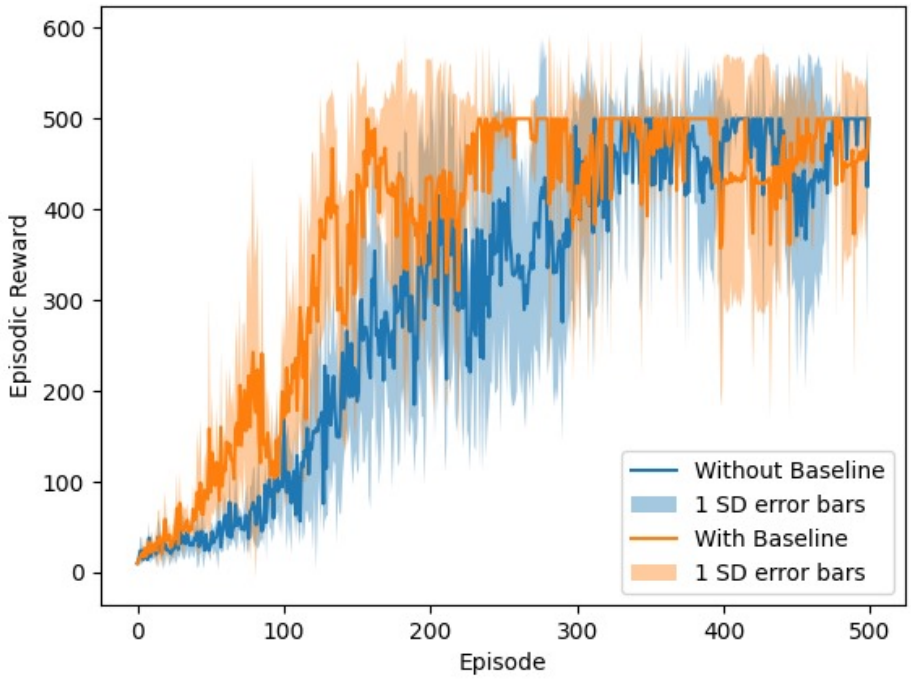

Figure 5: Episodic Reward v/s Episodes for both variants on the CartPole environment using REINFORCE algorithm averaged over 5 seeds

2.2 Acrobot-v1

This environment is part of the Classic Control environments which contains general information about the environment. 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.

The observation is a ndarray with shape (6,) that provides information about the two rotational joint angles as well as their angular velocities:

- Cosine of θ_1
- Sine of θ_1
- Cosine of θ_2
- Sine of θ_2
- Angular velocity of θ_1
- Angular velocity of θ_2

where θ_1 is the angle of the first joint and θ_2 is relative to the angle of the first link.

The action is discrete, deterministic and represents the torque applied on the actuated joint between the two links.

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.

Some configurations of the above-mentioned model that were tried out for both Type 1 and Type 2 updates are given below.

Hidden Layer neurons	**Learning Rate**	**Regret**
24	1.00E-03	201706.8
32	5.00E-03	92454.6

Table 7: Configurations for without baseline variant on Acrobot environment

Policy HL neurons	**Policy Learning Rate**	**Critic Learning Rate**	**Regret**
24	1.00E-03	1.00E-02	250664.6
32	1.00E-03	1.00E-02	297337.6
24	1.00E-02	1.00E-02	124653.2
32	1.00E-02	1.00E-02	67496.4

Table 8: Configurations for with baseline variant on Acrobot environment. The Critic Hidden Layer size is fixed at 16 neurons.

Here, regret is defined as the sum of difference between the optimal reward (0) that can be obtained in an episode and the actual episodic rewards observed. These values are reported after averaging over 5 seeds to reduce fluctuations due to stochasticity in the environment and the algorithm. In all experiments, we consider the number of episodes up until convergence is observed. In both tables, only the last row has been run for 500 episodes, and the remaining configurations were run for 1000 episodes.

Observing Tables 7 and 8 and the learning curves, we pick the least regret configurations for both variants to compare. We compare the episodic rewards v/s number of episodes for the case with 32 neurons in the policy network and 5×10^{-3} as the learning rate for the first variant (without baseline) against the case with 32 neurons in the policy network with 10^{-2} as the actor's and critic's learning rates. The resulting curve is averaged over 5 seeds and the error bar in the plot indicates one standard deviation difference.

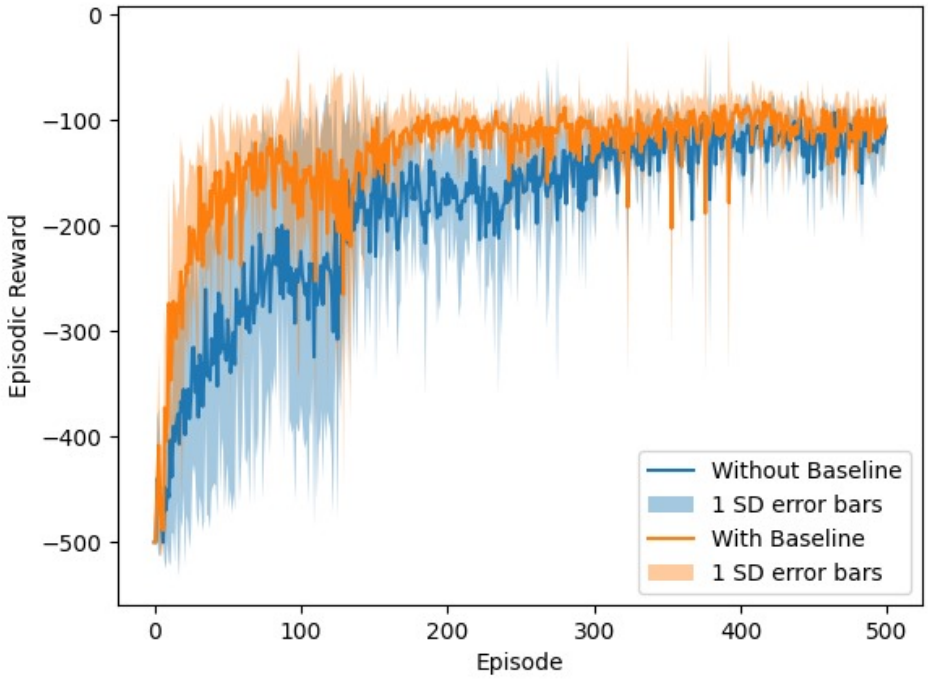

Figure 6: Episodic Reward v/s Episodes for both variants on the Acrobot environment using REINFORCE algorithm averaged over 5 seeds

2.3 Inferences

2.3.1 CartPole-v1

We conclude the following observations from the experiments performed on the CartPole-v1 environment.

- From Tables 5 and 6, we observe that the configurations with 32 neurons for both without baseline and with baseline variants converge very fast, hence we considered 500 episodes for training whereas in all the other cases we trained the agent for 1000 episodes.
- We also observe, from Table 6, that our regret optimal configuration has critic learning rate lower than the policy learning rate (although this is counter intuitive from theory).
- From the plot 5, we observe that the variance in the learning curves of the with baseline variant of REINFORCE is much lower than that of the without baseline variant. This is expected since adding a baseline reduces variance in the policy gradient algorithms. Baselines help stabilize the learning process by preventing the algorithm from overestimating or underestimating the performance of different actions.
- We further observe that the baseline variant converges faster and has lower regret than the without baseline variant, as observed from Figure 5 and Tables 5 and 6.
- Both types converge to an average episodic reward of almost 500. We know that the CartPole-v1 environment is considered to be solved when the running average over the past 100 episodes is greater than 195, with the maximum reward per episode being 500. Therefore, we can conclude this environment to be solved with both variants.

- Every training loop takes approximately 1.2 minutes for without baseline variant and 2.6 minutes for the with baseline variant. We hypothesize that this is due to the higher number of parameters to be estimated in the baseline variant, since we have an extra critic network to update.
- We can obtain smoother learning curves if we average over more number of seeds, this essentially averages over the stochasticity present in the algorithm.

2.3.2 Acrobot-v1

We conclude the following observations from the experiments performed on the Acrobot-v1 environment.

- From Tables 7 and 8, we observe that the configurations with 32 neurons for both without baseline and with baseline variants converge very fast, hence we considered 500 episodes for training whereas in all the other cases we trained the agent for 1000 episodes.
- We also observe, from Table 8, that our regret optimal configuration has a critic learning rate equal to the policy learning rate, although theoretically, a higher critic learning rate should be better.
- From the plot 6, we observe that the variance in the learning curves of the with baseline variant of REINFORCE is much lower than that of the without baseline variant. This is expected since adding a baseline reduces variance in the policy gradient algorithms. Baselines help stabilize the learning process by preventing the algorithm from overestimating or underestimating the performance of different actions.
- We further observe that the baseline variant converges much faster and has lower regret than the without baseline variant, as observed from Figure 6 and Tables 7 and 8. We can see that the baseline variant has converged to -100 reward around 200 episodes itself, whereas the without baseline variant reaches -100 reward only around 400 episodes.
- Both types converge to an average episodic reward of around -100 in 500 episodes. From documentation, we know that the Acrobot-v1 environment is considered to be solved when the average reward is -100. Therefore, we can conclude this environment to be solved with both variants in 500 episodes.
- Every training loop takes approximately 1 minute for without baseline variant and 1.2 minutes for the with baseline variant. We hypothesize that the baseline variant taking longer is due to the higher number of parameters to be estimated in the baseline variant since we have an extra critic network to update.
- We can obtain smoother learning curves if we average over more number of seeds, this essentially averages over the stochasticity present in the algorithm.

3 GitHub Repository

Please find the repository with all the codes, plots and report in this GitHub repository - Link.