

CS6700

Reinforcement Learning

Programming Assignment 2

Archish S Vinayak Gupta
me20b032@smail.iitm.ac.in ee20b152@smail.iitm.ac.in



Indian Institute of Technology, Madras

Saturday 6th April, 2024

Contents

1	Introduction	2
2	Environment Description	2
2.1	CartPole	2
2.2	Acrobot	2
3	Deep Q-Learning	3
3.1	Dueling DQN	3
3.2	Experiments	4
3.2.1	on CartPole	7
3.2.2	on Acrobot	8
4	Policy Gradient Methods	9
4.1	Monte-Carlo REINFORCE	9
4.2	Experiments	10
4.2.1	on CartPole	13
4.2.2	on Acrobot	14
5	Conclusion	15

Scripts

The code for this assignment (and other assignments) and output files are logged in this [GitHub¹](#) repository.

1 Introduction

Deep Reinforcement Learning (Deep RL) utilizes Deep Learning to enhance traditional Reinforcement Learning algorithms. The principal idea is to make use of a neural network to parameterize the Q function (and possibly the policy π).

In this assignment, we study two methods of Deep RL: Dueling DQN [3.1](#) and Monte-Carlo REINFORCE [4.1](#).

2 Environment Description

We base our study on the [CartPole-v1](#) and the [Acrobot-v1](#) environments of the Open AI's [gymnasium](#) library.

2.1 CartPole

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 force in the left and right directions.

Action Space

The action can take values $\{0, 1\}$ (0 for left and 1 for right), indicating the direction of the fixed force that the cart is pushed with.

State Space

The observation space consists of the values corresponding to the Cart Position $(-4.8, 4.8)$, the Cart Velocity $(-\infty, \infty)$, the Pole Angle $(-24^\circ, 24^\circ)$ and the Pole Angular Velocity $(-\infty, \infty)$.

Rewards

The goal is to keep the pole upright for as long as possible. A reward of +1 is allotted for every step taken.

Termination

The episode terminates if

- Cart Position is not in the range $[-2.4, 2.4]$ (or)
- Pole Angle is not in the range $[-12^\circ, 12^\circ]$ (or)
- Episode length is greater than 200.

2.2 Acrobot

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.

Action Space

The action can take values $\{-1, 0, 1\}$ indicating torque applied on the actuated joint.

¹<https://github.com/Xerefic/CS6700>

State Space

The observation space consists of the values corresponding to the two rotational joint angles, Sine and Cosine of angles θ_1 and θ_2 , and the angular velocity of θ_1 and θ_2 in $(-4\pi, 4\pi)$, where θ_1 is the angle of the first joint with the y -axis and θ_2 is the relative angle to the first link.

Rewards

The goal is to have the free end reach a designated target. A reward of -1 is allotted for all the steps that do not reach the goal.

Termination

The episode terminates if

- Reaching the target height $-\cos(\theta_1) - \cos(\theta_2 + \theta_1) > 1$ (or)
- Episode length is greater than 500.

3 Deep Q-Learning

This strategy focuses on learning the parameterized Q function by comparing the Q-value predicted with a Q-target and uses gradient descent to update the parameters. The Deep Q-Learning algorithm has two phases

- Perform actions and store the observed experience tuples in a replay buffer,
- Use a randomly selected mini-batch of tuples and update the parameters using gradient descent.

Algorithm 3.1: Deep Q-Learning

Input: discount factor γ ; update steps c ; replay buffer size N

Initialize: replay memory \mathcal{D} with capacity N ; state-action value function Q with random weights Θ ; target function \hat{Q} with weights $\hat{\Theta} = \Theta$

```

1 for episode = 1 to  $M$  do
2   Initialize sequence  $s_1$ 
3   for  $t = 1$  to  $T$  do
4     With probability  $\epsilon$  select a random action  $a_t$  otherwise select  $a_t = \arg \max_{a \in \mathcal{A}(s_t)} Q(s_t, a; \Theta)$ 
5     Take action  $a_t$ , observe reward  $r_t$  and next state  $s_{t+1}$  and store transition  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{D}$ 
6     Sample random mini-batch  $\mathcal{D}_b \subseteq \mathcal{D}$  and compute  $\mathbf{y}$  as
        
$$y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a' \in \mathcal{A}(s_{j+1})} \hat{Q}(s_{j+1}, a'; \hat{\Theta}) & \text{otherwise} \end{cases} \quad \forall j = \{|\mathcal{D}_b|\}$$

7     Perform gradient descent step on  $(\mathbf{y} - \mathbf{Q}(\cdot; \Theta))^2$  for  $Q_j(\cdot; \Theta) = Q(s_j, a_j; \Theta) \forall j = \{|\mathcal{D}_b|\}$ 
8     if  $t \bmod c = 0$  then
9       reset  $\hat{\Theta} \leftarrow \Theta$ 

```

Output: Parameterized Q function $Q(\cdot; \Theta)$

3.1 Dueling DQN

Dueling DQN is an extension of the Deep Q-Learning Algorithm 1, designed to improve learning efficiency by decomposing the Q-value function into two separate streams - one estimating the state value and the other estimating the advantage of each action. The update equation for the dueling network is modified as

$$Q(s, a; \Theta) \doteq V(s; \Theta) + \left(A(s, a; \Theta) - \frac{1}{|\mathcal{A}(s)|} \sum_{a' \in \mathcal{A}(s)} A(s, a'; \Theta) \right) \quad (\text{Type 1})$$

$$Q(s, a; \Theta) \doteq V(s; \Theta) + \left(A(s, a; \Theta) - \max_{a' \in \mathcal{A}(s)} A(s, a'; \Theta) \right) \quad (\text{Type 2})$$

where $Q(\cdot; \Theta)$ represents the parameterized dueling Q function.

3.2 Experiments

We simulate a total of 2×2 experiments on the CartPole 2.1 and Acrobot 2.2 environments using both the update types (Type 1, Type 2). We repeat each experiment 5 times, with each run consisting of 1000 episodes (until termination). To ensure that the experiments are consistent, we re-initialize the seed to be 184(32+152) before the start of every experiment.

We iterate through a combination of hyperparameters for each experiment to find the optimal values through grid search. We fix the value of the discount factor $\gamma = 0.99$ and use a linear scheduler for the exploration factor ϵ as $\epsilon_t = \max\left(\epsilon_{\min}, \epsilon_{\max} - (\epsilon_{\max} - \epsilon_{\min}) \times \frac{t}{\text{num_episodes}}\right)$ for $\epsilon_{\min} = 10^{-4}$, $\epsilon_{\max} = 0.1$.

Parameter	Values
replay_size	[5000]
update_steps	[4, 10, 12]
hidden_size	[32, 48, 64]

Table 3.1: Experiment Configurations

Optimality Measure

We use maximum episodic return (i.e., minimizing regret) optimality to filter out the “best” hyperparameter configuration for each experiment. Episodic return $G_t = \sum_{t=0}^T \gamma^t r_t$ encapsulates both the best performing algorithm and the time taken for learning the algorithm.

The “best” hyperparameter configuration identified for Type 1 case.

Implementation Details

We use a shared feature extractor for the value and the advantage functions. At the start of every run, the weights of the neural network are re-initialized according to Xavier Normal. The network is trained using AdAM optimizer with a learning rate of 10^{-4} .

Note: The code snippets are beautified for demonstration purposes. Actual code may vary.

```

1  class QNetwork(nn.Module):
2      def __init__(self, state_size, action_size, hidden_size):
3          super(QNetwork, self).__init__()
4
5          self.fc = nn.Sequential(
6              nn.Linear(state_size, hidden_size),
7              nn.ReLU(),
8              nn.Linear(hidden_size, 4*hidden_size),
9              nn.ReLU(),
10         )
11         self.value = nn.Linear(4*hidden_size, 1)
12
13         self.advantage = nn.Linear(4*hidden_size, action_size)
14
15     def forward(self, x, rule):
16         x = self.fc(x)
17         value = self.value(x)
18         advantage = self.advantage(x)
19
20         if rule == 'type1':
21             Q = value + advantage - advantage.mean(dim=1, keepdim=True)
22         elif rule == 'type2':
23             Q = value + (advantage - torch.max(advantage, dim=1, keepdim=True)[0])
24         return Q
25
26     def select_action(self, state):
27         with torch.no_grad():
28             Q = self.forward(state)
29             action = Q.argmax(dim=1)
30         return action

```

Listing 3.1: Q Network Definition

```

1  class Memory:
2      @property
3      def size(self):
4          return len(self.buffer)
5
6      def __init__(self, size):
7          self.buffer = deque(maxlen=size)
8
9      def add(self, experience):
10         state, action, reward, next_state, done = experience
11         self.buffer.append((state, action, reward, next_state, done))
12
13     def sample(self, batch_size, continuous=True):
14         batch_size = min(batch_size, self.size)
15
16         if continuous:
17             idx = np.random.randint(0, self.size - batch_size)
18             return [self.buffer[i] for i in range(idx, idx + batch_size)]
19         else:
20             idx = np.random.choice(range(self.size), batch_size, replace=False)
21             return [self.buffer[i] for i in idx]
22
23     def clear(self):
24         self.buffer.clear()

```

Listing 3.2: Replay Buffer

```
1  with torch.no_grad():
2      onlineQ_next = self.online_Qnetwork(next_state)
3      targetQ_next = self.target_Qnetwork(next_state)
4      online_action = onlineQ_next.argmax(dim=1, keepdim=True)
5      y = batch_reward + gamma * targetQ_next.gather(1, online_action) * (1 - done)
6
7  loss = F.mse_loss(self.online_Qnetwork(state).gather(1, action), y)
8  optimizer.zero_grad()
9  loss.backward()
10 optimizer.step()
```

Listing 3.3: Learning

3.2.1 on CartPole

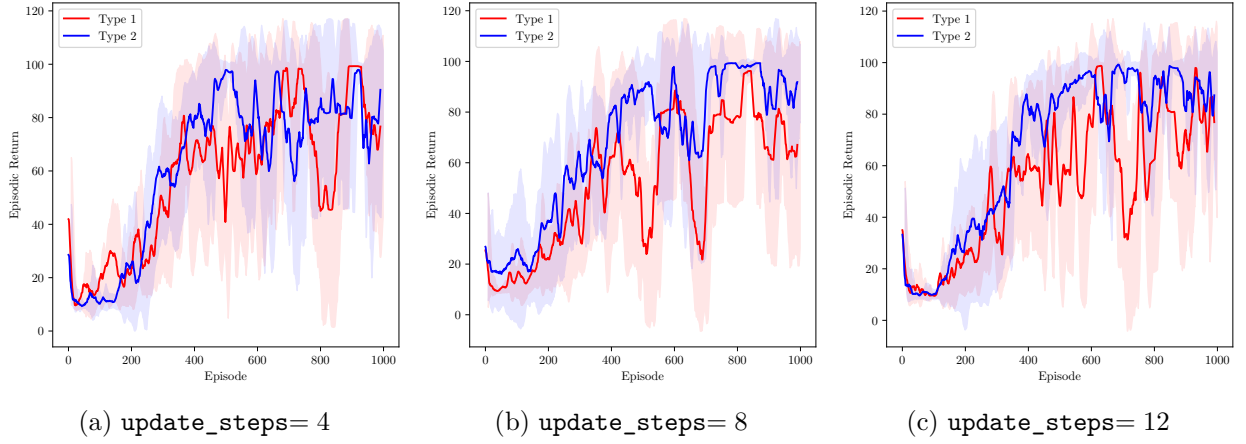


Figure 3.1: Experiments for CartPole Environment

Parameter	Values
replay_size	5000
update_steps	12
hidden_size	64

Table 3.2: “Best” Configuration for CartPole Environment

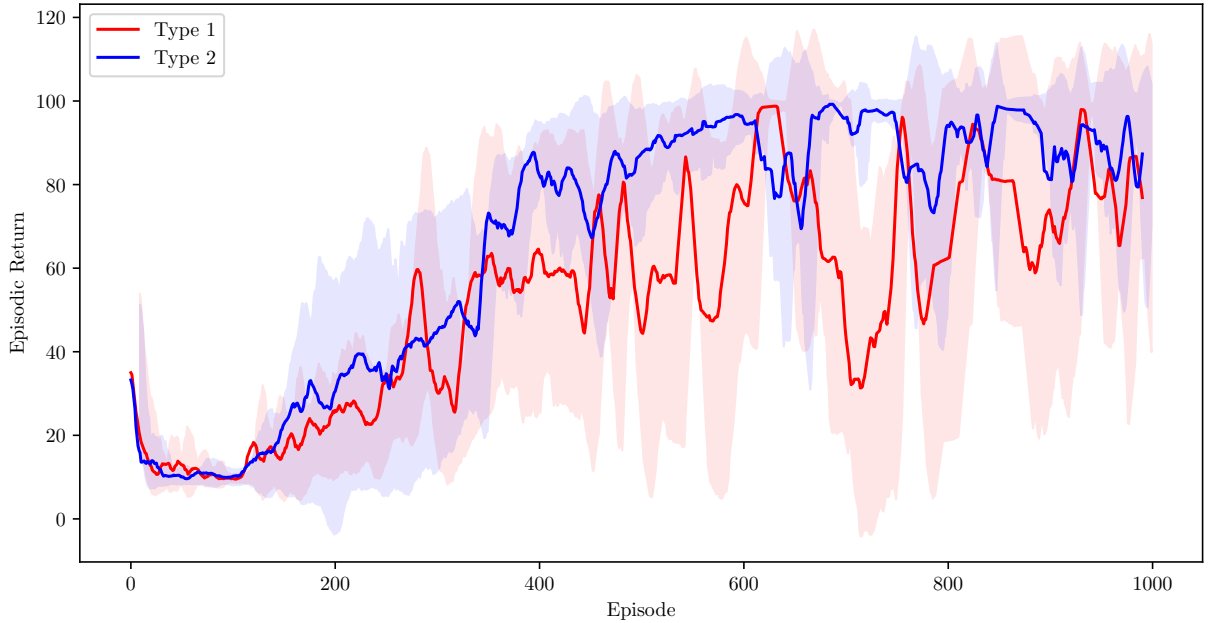


Figure 3.2: Dueling DQN on CartPole Environment

Observations:

- Both Type 1 and Type 2 Dueling DQN configurations demonstrate quick convergence, with reward values settling around episodes 400-500.
- They achieve a high final average reward of approximately 100-110, showcasing strong performance in the Cartpole environment.
- However, the reward curves display some fluctuations, particularly noticeable in later episodes with several large oscillations.

3.2.2 on Acrobot

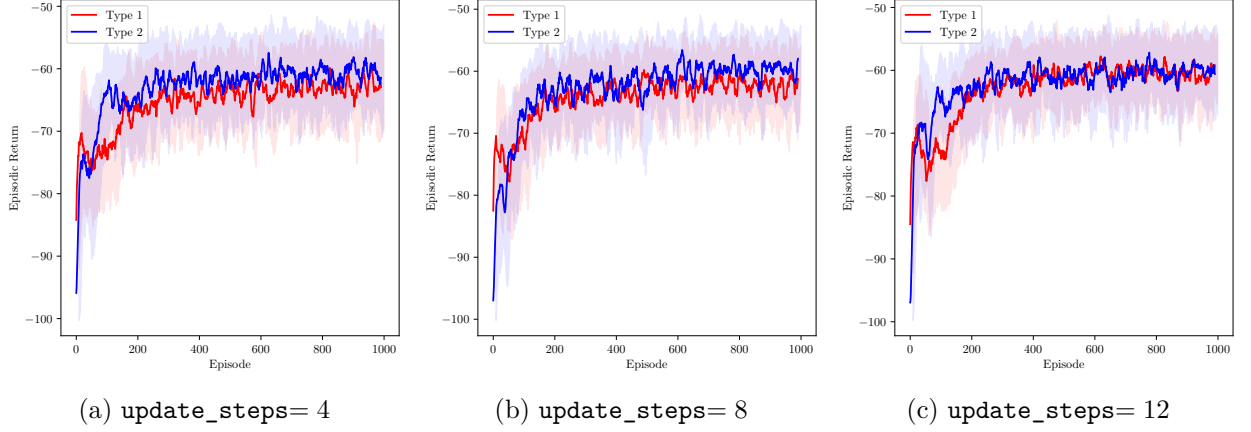


Figure 3.3: Experiments for Acrobot Environment

Parameter	Values
replay_size	5000
update_steps	12
hidden_size	48

Table 3.3: “Best” Configuration for Acrobot Environment

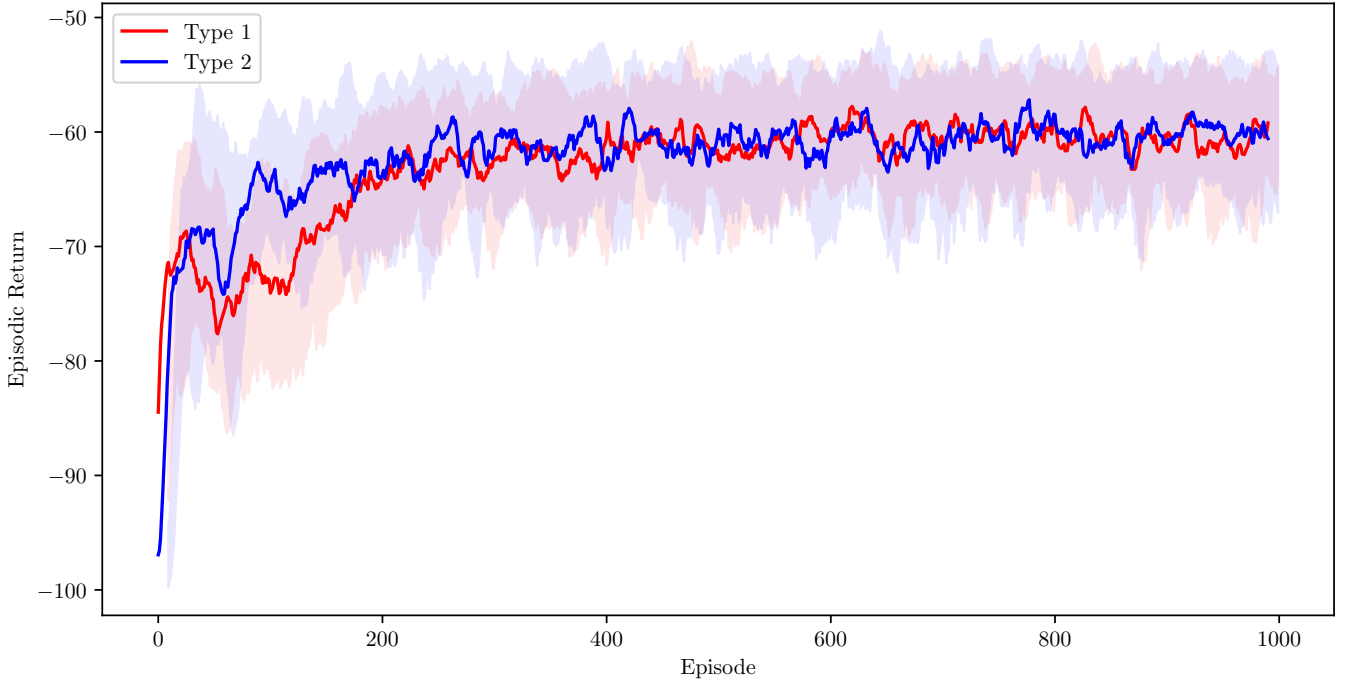


Figure 3.4: Dueling DQN on Acrobot Environment

Observations:

- In contrast to Cartpole, the Dueling DQN algorithms converge rapidly in the Acrobot environment, with reward values stabilizing around 200-300 episodes.
- The reward curves demonstrate reduced volatility, featuring minor oscillations throughout the learning phase. This suggests stable learning and policy updates by the algorithms.

4 Policy Gradient Methods

In Policy Gradient methods, the policy π is parameterized as it offers an advantage over state action value based methods for continuous action spaces. The policy is parameterized as an exponential softmax distribution

$$\pi(a|s; \Theta) \doteq \frac{e^{h(s,a;\Theta)}}{\sum_{a' \in \mathcal{A}(s)} e^{h(s,a';\Theta)}}$$

where $h(\cdot; \Theta)$ maybe a neural network. Parameterizing policies according to the softmax in action preferences enables the selection of actions with arbitrary probabilities. With continuous policy parameterization, the action probabilities change smoothly as a function of the learned parameter.

In this assignment, we focus on episodic Policy Gradient method, for which the performance measure is defined as the value of the start state of each episode

$$J(\Theta) = v_{\pi_\Theta}(s_0)$$

where v_{π_Θ} is the true value function for π_Θ . The policy gradient theorem for the episodic case establishes that

$$\nabla J(\Theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s; \Theta)$$

where $\mu(\cdot)$ is the on-policy distribution under π .

Algorithm 4.1: (Episodic) Policy Gradient

Input: learning rate α ; discount factor γ

Initialize: policy function π with random weights Θ

1 **for** episode = 1 to M **do**

2 Initialize trajectory \mathcal{T}

3 **for** $t = 1$ to T (or till termination) **do**

4 Take action a_t following $\pi(\cdot|s; \Theta)$, observe reward r_t and next state s_{t+1} and store transition (s_t, a_t, r_t) in \mathcal{T}

5 **for** $t = 1$ to T **do**

6 Compute $\nabla J_t(\Theta)$ for start state s_t from \mathcal{T}

7 $\Theta \leftarrow \Theta + \alpha \gamma^t \nabla J_t(\Theta)$

Output: Parameterized policy $\pi(\cdot; \Theta)$

4.1 Monte-Carlo REINFORCE

The MC-REINFORCE algorithm utilizes Monte Carlo sampling to estimate gradients for policy optimization.

$$\begin{aligned} \nabla J(\Theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s; \Theta) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(s_t, a) \nabla \pi(a|s_t; \Theta) \right] \\ &= \mathbb{E}_\pi \left[\sum_a \pi(a|s_t; \Theta) q_\pi(s_t, a) \frac{\nabla \pi(a|s_t; \Theta)}{\pi(a|s_t; \Theta)} \right] \\ &= \mathbb{E}_\pi \left[q_\pi(s_t, a_t) \frac{\nabla \pi(a_t|s_t; \Theta)}{\pi(a_t|s_t; \Theta)} \right] \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(a_t|s_t; \Theta)}{\pi(a_t|s_t; \Theta)} \right] \quad (\text{because } \mathbb{E}_\pi[G_t|s_t, a_t] = q_\pi(s_t, a_t)) \end{aligned}$$

The update equation of its policy parameter Θ is given by

$$\Theta \leftarrow \Theta + \alpha G_t \frac{\nabla \pi(a_t | s_t; \Theta)}{\pi(a_t | s_t; \Theta)} \quad (\text{w/o Baseline})$$

$$\Theta \leftarrow \Theta + \alpha (G_t - V(s_t; \Phi)) \frac{\nabla \pi(a_t | s_t; \Theta)}{\pi(a_t | s_t; \Theta)} \quad (\text{w/ Baseline})$$

where $V(\cdot; \Phi)$ is the state value function updated by TD(0) method. It can be shown that by introducing a baseline (as long as it does not vary with a)

$$\nabla J(\Theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a | s; \Theta)$$

the equation remains valid.

4.2 Experiments

We simulate a total of 2×2 experiments on the CartPole 2.1 and Acrobot 2.2 environments using both the update types (**w/o Baseline**, **w/ Baseline**). We repeat each experiment 5 times, with each run consisting of 5000 episodes (until termination). To ensure that the experiments are consistent, we re-initialize the seed to be 184 before the start of every experiment.

We iterate through a combination of hyperparameters for each experiment to find the optimal values through grid search. We fix the value of the discount factor $\gamma = 0.99$ throughout the experiments.

Parameter	Values
<code>hidden_size</code>	[64, 128, 256]

Table 4.1: Experiment Configurations

Optimality Measure

We use maximum episodic return (i.e., minimizing regret) optimality to filter out the “best” hyperparameter configuration for each experiment. Episodic return $G_t = \sum_{t=0}^T \gamma^t r_t$ encapsulates both the best performing algorithm and the time taken for learning the algorithm.

The “best” hyperparameter configuration identified for **w/o Baseline** case.

Implementation Details

We use independent feature extractors for the policy and the value functions. At the start of every run, the weights of the neural network are re-initialized according to Xavier Normal. The network is trained using AdAM optimizer with a learning rate of 10^{-4} .

Note: The code snippets are beautified for demonstration purposes. Actual code may vary.

```
1 class PolicyNetwork(nn.Module):
2     def __init__(self, state_size, action_size, hidden_size):
3         super(PolicyNetwork, self).__init__()
4
5         self.fc = nn.Sequential(
6             nn.Linear(state_size, hidden_size),
7             nn.ReLU(),
8             nn.Linear(hidden_size, 4*hidden_size),
9             nn.ReLU(),
10            nn.Linear(4*hidden_size, action_size)
11        )
12
13    def forward(self, x):
14        return self.fc(x)
15
16    def select_action(self, state):
17        with torch.no_grad():
18            action = self.forward(state)
19            m = Categorical(F.softmax(action, dim=-1))
20        return m.sample()
```

Listing 4.1: Policy Network Definition

```
1 class ValueNetwork(nn.Module):
2     def __init__(self, state_size, hidden_size):
3         super(ValueNetwork, self).__init__()
4
5         self.fc = nn.Sequential(
6             nn.Linear(state_size, hidden_size),
7             nn.ReLU(),
8             nn.Linear(hidden_size, 4*hidden_size),
9             nn.ReLU(),
10            nn.Linear(4*hidden_size, 1)
11        )
12
13    def forward(self, x):
14        return self.fc(x)
```

Listing 4.2: Value Network Definition

```

1  G = []
2  total = 0
3  with torch.no_grad():
4      for r in batch_reward[:, -1]:
5          total = r + gamma * total
6          G.insert(0, total)
7      G = (G - G.mean()) / G.std()
8
9  action_prob = self.policy.forward(batch_state)
10 batch_log_prob = F.log_softmax(action_prob, dim=-1).gather(1, batch_actions.unsqueeze(1))
11 if rule == 'wbaseline':
12     loss_value = F.mse_loss(self.value(state), G)
13     optimizer_value.zero_grad()
14     loss_value.backward()
15     optimizer_value.step()
16
17     deltas = G - value.detach()
18     loss_policy = (-log_prob * deltas).mean()
19 elif rule == 'wbaseline':
20     deltas = G
21     loss_policy = (-log_prob * deltas).mean()
22
23 optimizer_policy.zero_grad()
24 loss_policy.backward()
25 optimizer_policy.step()

```

Listing 4.3: Learning

4.2.1 on CartPole

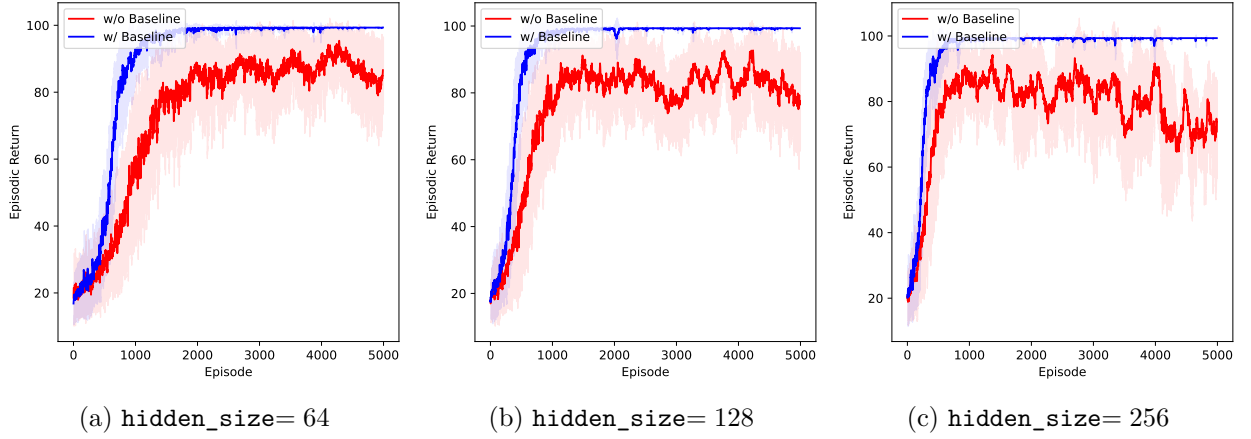


Figure 4.1: Experiments for CartPole Environment

Parameter	Values
hidden_size	256

Table 4.2: “Best” Configuration for CartPole Environment

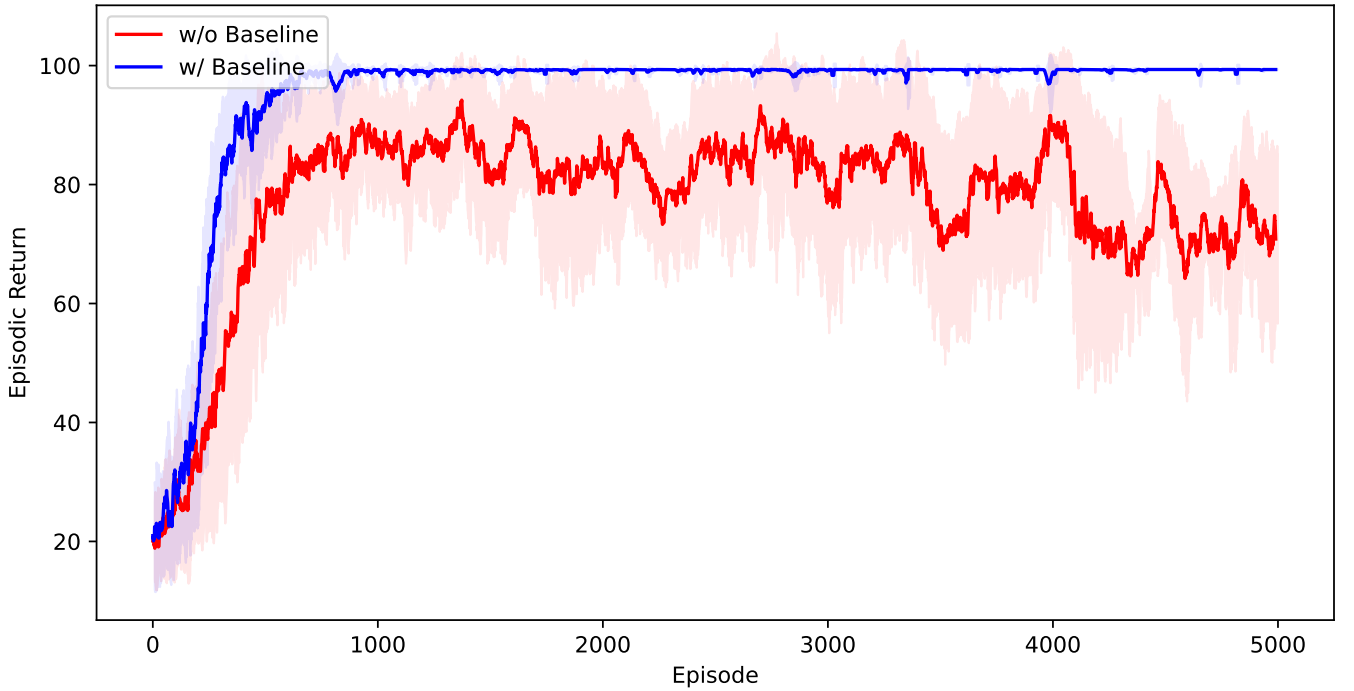


Figure 4.2: Monte-Carlo REINFORCE on CartPole Environment

Observations:

- MC-Reinforce shows a different learning pattern compared to the Dueling DQN approach, learns much faster in comparison.
- Regardless of varying hidden sizes, a consistent trend of low variance is observed.

4.2.2 on Acrobot

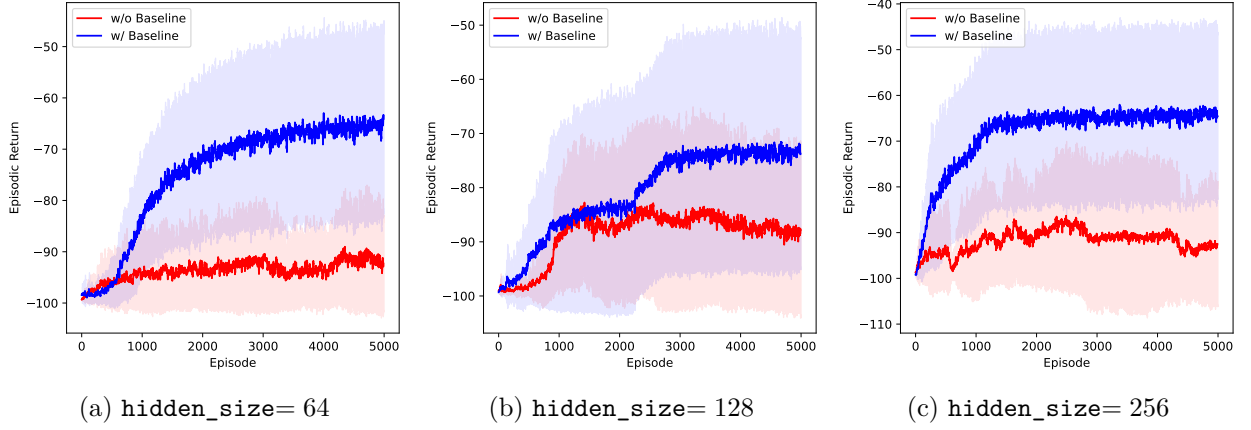


Figure 4.3: Experiments for Acrobot Environment

Parameter	Values
hidden_size	128

Table 4.3: “Best” Configuration for Acrobot Environment



Figure 4.4: Monte-Carlo REINFORCE on Acrobot Environment

Observations:

- Agents with baseline appear to perform better as opposed to without baseline.
- Across varying hidden sizes, we note a consistent trend of high variance in the results.

5 Conclusion

Based on extensive experimentation on the three environments, we arrive at the following conclusion:

- During the experiments with Dueling DQN, the maximum limit of 1000 episodes was consistently used. Notably, none of these experiments surpassed this limit, indicating that DQN agents can handle these challenging classical environments within a reasonable time-frame without requiring extensive tuning.
- When examining Reinforce-MC, we see that the agent converges to a global optimum rather quickly. Adjusting hyperparameters such as hidden layer sizes does not significantly improve convergence for the Monte Carlo agents.
- The neural network architecture should be sufficiently representative but not too sophisticated. Excessively large networks can impede performance, with the level of complexity needed varying based on the environment's complexity.
- The performance of both Dueling DQN and MC-Reinforce exhibits a high level of stochasticity. A specific set of hyperparameters that works well with one random seed may not yield similar results with a different seed. Although averaging can mitigate the impact of stochastic effects, it is crucial to acknowledge that results can vary significantly due to stochasticity.