CS6700
REINFORCEMENT LEARNING (SPRING 2024)
PROGRAMMING ASSIGNMENT 2

Dueling DQN and Monte−Carlo REINFORCE

Instructor: Prof. Balaraman Ravindran

Students:
ME20B087 Janmenjaya Panda
ME20B122 Nishant Sahoo

Release Date: 23/03/2024
Submission Deadline: 07/04/2024

GitHub Repo Submission: Link

Department of Computer Science & Engineering
Indian Institute of Technology Madras

Contents

1 Environment Description

This exercise aims to study the two variants each of: **Dueling-DQN**[1] and **Monte-Carlo REINFORCE**[2].

In this programming task, we shall utilize the following Gymnasium environments for training and evaluating several policies. The following introduction of Gymnasium has been sourced from the official documentation of Gymnasium available at Gymnasium

Gymnasium is a maintained fork of OpenAI's Gym library. The Gymnasium interface is simple, pythonic, and capable of representing general RL problems, and has a compatibility wrapper for old Gymnasium environments. Figure 1 shows the Lunar Lander v2 environment in the Gymnasium rendered in the human mode.

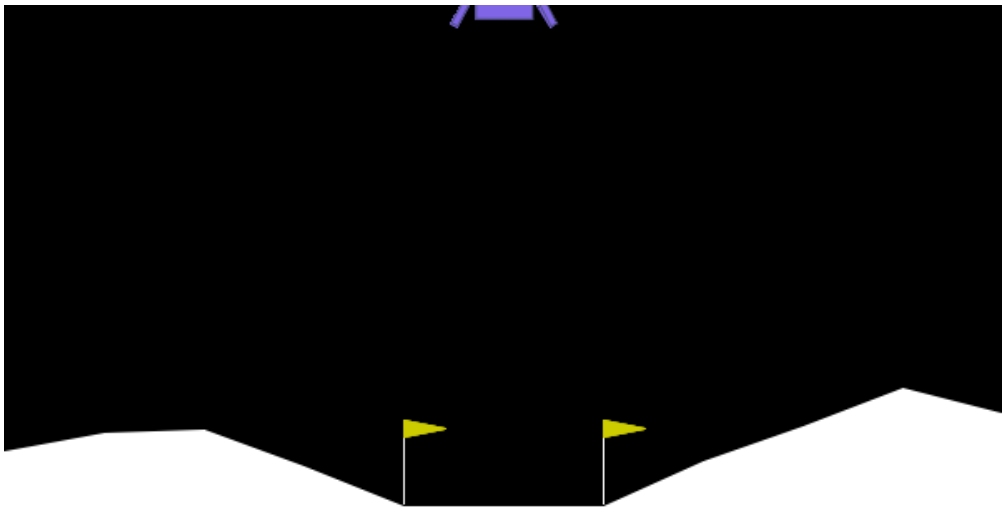


Figure 1: The Lunar Lander v2 environment in the Gymnasium: an API standard for RL with a diverse collection of reference environments

```
import gymnasium as gym

env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
   action = env.action_space.sample()  # this is where you would insert your policy
   observation, reward, terminated, truncated, info = env.step(action)

   if terminated or truncated:
      observation, info = env.reset()

env.close()
```

We shall be dealing with two environments in the Gymnasium discussed followingly.

[1] **D**eep - **Q**- **N**etwork [2] **RE**ward **I**ncrement = **N**onnegative **F**actor × **O**ffset **R**einforcement × **C**haracteristic **E**ligibility.

1.1 Acrobot v1

The following description of Acrobot v1 has been sourced from the official documentation of Acrobot v1 available at Acrobot v1.

Figure 2: The Acrobot v1 environment in the Gymnasium

Acrobot v1 is part of the Classic Control environments which contains general information about the environment.

Variable	Value
Action Space	`Discrete(3)`
Observation Space	`Box([ -1. -1. -1. -1. -12.566371 -28.274334], [ 1. 1. 1. 1. 12.566371 28.274334], (6,), float32)`
import	`gymnasium.make(“Acrobot-v1”)`

Table 1: Acrobot v1 specifications

1.1.1 Description

The Acrobot environment is based on Sutton’s work in “Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding"[5] and Sutton and Barto’s book[6]. 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.

As seen in the Gif: two blue links are connected by two green joints. The joint in between the two links is actuated. The goal is to swing the free end of the outer-link to reach the target height (black horizontal line above system) by applying torque on the actuator.

1.1.2 Action Space

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

Num	Action	Unit
0	apply -1 torque to the actuated joint	torque ($N \cdot m$)
1	apply 0 torque to the actuated joint	torque ($N \cdot m$)
2	apply 1 torque to the actuated joint	torque ($N \cdot m$)

Table 2: The action space of Acrobot v1

1.1.3 Observation Space

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

Num	Observation	Min	Max
0	Cosine of `theta1`	-1	1
1	Sine of `theta1`	-1	1
2	Cosine of `theta2`	-1	1
3	Sine of `theta2`	-1	1
4	Angular velocity of `theta1`	$\sim$ -12.567 (-4π)	$\sim$ 12.567 (4π)
5	Angular velocity of `theta2`	$\sim$ -28.274 (-9π)	$\sim$ 28.274 (9π)

Table 3: The observation space of Acrobot v1

where

- `theta1` is the angle of the first joint, where an angle of 0 indicates the first link is pointing directly downwards.
- `theta2` is relative to the angle of the first link. An angle of 0 corresponds to having the same angle between the two links.

The angular velocities of `theta1` and `theta2` are bounded at $\pm 4\pi$, and $\pm 9\pi$ rad/s respectively. A state of `[1, 0, 1, 0, ..., ...]` indicates that both links are pointing downwards.

1.1.4 Rewards

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 reward threshold is -100.

1.1.5 Starting State

Each parameter in the underlying state (`theta1`, `theta2`, and the two angular velocities) is initialized uniformly between -0.1 and 0.1. This means both links are pointing downwards with some initial stochasticity.

1.1.6 Episode End

The episode ends if one of the following occurs:

1. Termination: The free end reaches the target height, which is constructed as: `-cos(theta1) - cos(theta2 + theta1) > 1.0`, or otherwise
2. Truncation: Episode length is greater than 500 (200 for v0).

1.1.7 Arguments

No additional arguments are currently supported during construction.

```
import gymnasium as gym
env = gym.make('Acrobot-v1')
```

On reset, the options parameter allows the user to change the bounds used to determine the new random state.

By default, the dynamics of the acrobot follow those described in Sutton and Barto's book Reinforcement Learning: An Introduction. However, a `book_or_nips` parameter can be modified to change the pendulum dynamics to those described in the original NeurIPS paper [5].

```
# To change the dynamics as described above
env.unwrapped.book_or_nips = 'nips'
```

The dynamics equations were missing some terms in the NIPS paper which are present in the book. R. Sutton confirmed in personal correspondence that the experimental results shown in the paper and the book were generated with the equations shown in the book. However, there is the option to run the domain with the paper equations by setting `book_or_nips = 'nips'`.

1.1.8 Version History

- v1: Maximum number of steps increased from 200 to 500. The observation space for v0 provided direct readings of `theta1` and `theta2` in radians, having a range of `[-pi, pi]`. The v1 observation space as described here provides the sine and cosine of each angle instead.
- v0: Initial versions release (1.0.0) (removed from gymnasium for v1)

1.2 CartPole v1

The description provided for CartPole v1 is derived from the documentation accessible on the official CartPole v1 webpage at CartPole v1.

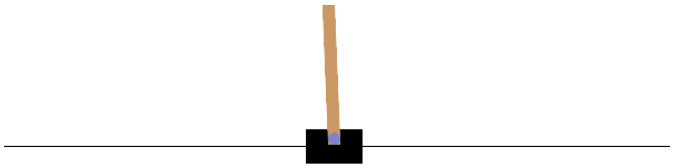


Figure 3: The CartPole v1 environment in the Gymnasium

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

Variable	Value
Action Space	`Discrete(2)`
Observation Space	`Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38], [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,), float32)`
import	`gymnasium.make("CartPole-v1")`

Table 4: CartPole v1 specifications

1.2.1 Description

This environment corresponds to the version of the cart-pole problem described by Barto, Sutton, and Anderson in "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem"[2]. A pole is attached by an un-actuated 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.

1.2.2 Action Space

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

Note: The velocity that is reduced or increased by the applied force is not fixed and it depends on the angle the pole is pointing. The center of gravity of the pole varies the amount of energy needed to move the cart underneath it

1.2.3 Observation Space

The observation is a `ndarray` with shape `(4,)` with the values corresponding to the following positions and velocities:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-inf	inf
2	Pole Angle	$\sim$ -0.418 rad (-24°)	$\sim$ 0.418 rad (24∘)
3	Pole Angular Velocity	-inf	inf

Table 5: The observation space of Acrobot v1

Note: While the ranges above denote the possible values for observation space of each element, it is not reflective of the allowed values of the state space in an unterminated episode. Particularly:

- 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 (-.418, .418) radians (or $\pm$24°), but the episode terminates if the pole angle is not in the range `(-.2095, .2095)` (or $\pm$12°)

1.2.4 Rewards

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 threshold for rewards is 500 for v1 and 200 for v0.

1.2.5 Starting State

All observations are assigned a uniformly random value in `(-0.05, 0.05)`.

1.2.6 Episode End

The episode ends if any one of the following occurs:

1. Termination: Pole Angle is greater than $\pm$12°.
2. Termination: Cart Position is greater than $\pm$2.4 (center of the cart reaches the edge of the display)
3. Truncation: Episode length is greater than 500 (200 for v0)

1.2.7 Arguments

```
import gymnasium as gym
gym.make('CartPole-v1')
```

On reset, the options parameter allows the user to change the bounds used to determine the new random state.

2 Algorithms and Implementations

This section discusses the theory and the implementational aspects of the concerned variants of both algorithms, that are — Dueling DQN and Monte Carlo REINFORCE.

2.1 Dueling DQN

We consider a sequential decision-making setup, in which an agent interacts with an environment $\mathcal{E}$ over discrete time steps, see Sutton and Barto [6] for an introduction. In the Acrobot domain, for example, the agent perceives the observation s_t consisting of the angles and angular velocities wrt the two rotation joint axis, that may be represented as the consistution of M parametric frames: $s_t = (x_{t-M+1}, \dots, x_t) \in \mathcal{S}$ at time step t. The agent then chooses an action from a discrete set $a_t \in \mathcal{A} := 1, ..., |\mathcal{A}|$, see the Acrobot section and observes a reward signal r_t produced by the game emulator.

The agent seeks maximize the expected discounted return, where we define the discounted return as

$$G_t := \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$$

In this formulation, $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards. For an agent behaving according to a stochastic policy π, the values of the state-action pair (s, a) and the state s are defined as follows:

$$\begin{aligned} Q^\pi(s, a) &:= \mathbb{E}\left[G_t | s_t = s, a_t = a, \pi\right] \\ V^\pi(s) &:= \mathbb{E}_{a\sim\pi(s)}\left[Q^\pi(s, a)\right] \end{aligned} \tag{1}$$

The preceding state-action value function (Q function for short) can be computed recursively with dynamic programming:

$$Q^\pi(s, a) = \mathbb{E}\left[r + \gamma \mathbb{E}_{a'\sim\pi(s')}\left[Q^\pi(s', a')\right] | s, a, \pi\right]$$

We let the optimal $Q^*(s, a) := \max_\pi Q^\pi(s, a)$. Under the deterministic policy $a = \arg\max_{a'\in\mathcal{A}} Q^*(s, a')$, it follows that $V^*(s) = \max_a Q^*(s, a)$. From this, it also follows that the optimal Q function satisfies the Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{a'}\left[r + \gamma \max_{a'} Q^*(s', a') | s, a\right] \tag{2}$$

As per [9], we also define another important quantity, the advantage function, relating the value and Q functions as follows:

$$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{3}$$

Note that $\mathbb{E}_{a\sim\pi(s)}\left[A^\pi(s, a)\right] = 0$. Intuitively, the value function V measures the how good it is to be in a particular state s. The Q function, however, measures the the value of choosing a particular action when in this state. The advantage function subtracts the value of the state from the Q function to obtain a relative measure of the importance of each action.

2.1.1 Deep $Q-$Network

The Deep $Q-$Network (DQN) algorithm combines the power of deep learning with traditional $Q-$learning. It was introduced by DeepMind [4] in 2013 and has since become a foundational algorithm in the field of deep reinforcement learning. The value functions as described in the preceding section are high dimensional objects. To approximate them, we can use a deep Q-network: $Q(s, a; \theta)$ with parameters θ. To estimate this network, we optimize the following sequence of loss functions at iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[(y_i - Q(s, a; \theta))^2\right] \tag{4}$$

with

$$y_i = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{5}$$

where θ^- represents the parameters of a fixed and separate target network. We could attempt to use standard $Q-$learning to learn the parameters of the network $Q(s, a; \theta)$ online. However, this estimator performs poorly in practice. A key innovation in [1] was to freeze the parameters of the target network $Q(s^t, a^t; 0^-$ for a fixed number of iterations while updating the online network $Q(s, a; \theta_i)$ by gradient descent. (This greatly improves the stability of the algorithm.) The specific gradient update is

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_(s, a, r, s')\left[(y_i - Q(s, a; \theta_i))\, \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{6}$$

This approach is model free in the sense that the states and rewards are produced by the environment. It is also offpolicy because these states and rewards are obtained with a behavior policy (epsilon greedy in DQN) different from the online policy that is being learned.

Another key ingredient behind the success of DQN is experience replay [3][4]. During learning, the agent accumulates a dataset $\mathcal{D}_t = \{e_1, e_2, \ldots, e_t\}$ of experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ from many episodes. When training the $Q-$Network, instead only using the current experience as prescribed by standard TD$-$Learning, the network is trained by sampling mini-batches of experiences from D uniformly at random. The sequence of losses thus takes the form:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(\mathcal{D})}\left[(y_i - Q(s, a; \theta_i))^2\right] \tag{7}$$

Experience replay increases data efficiency through the re-use of experience samples in multiple updates and, importantly, it reduces variance as uniform sampling from the replay buffer reduces the correlation among the samples used in the update.

At its core, a DQN is designed to approximate the optimal action-value function $Q^*(s, a)$ which represents the expected cumulative future rewards of taking action a in state s, and then following the optimal policy thereafter. The key innovation of DQN is the use of deep neural networks to represent this action-value function, allowing it to handle high-dimensional state spaces commonly encountered in real-world applications. Figure 4 captures the architecture of a standard DQN.

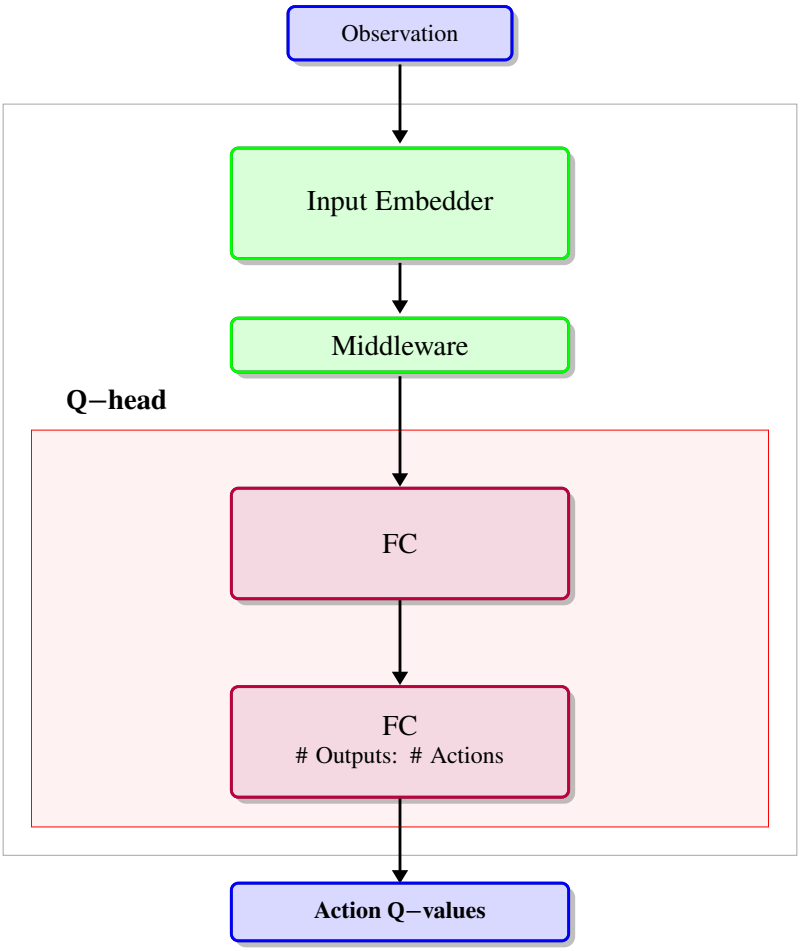


Figure 4: Architecture of Deep $Q-$Network

The training process of a DQN involves iteratively improving its approximation of the action-value function by minimizing the temporal difference error, which measures the discrepancy between the predicted Q-values and the observed rewards. This is achieved through a form of gradient descent optimization. Thus, in Deep $Q-$Learning, we create a loss function that compares our $Q-$Value prediction and the $Q-$Target and uses gradient descent to update the weights of our Deep $Q-$Network to approximate our $Q-$Values better.

$$Q-\text{Target: } y_j = r_j + \gamma \max_{a'} \hat{Q}\left(\Phi_{j+1}, a'; \theta^-\right)$$

$$Q-\text{Loss: } y_j - Q\left(\Phi_j, a_j; \theta\right)$$

DQN utilizes experience replay and target networks to stabilize and improve learning efficiency. Experience replay involves storing past experiences (state, action, reward, next state) in a replay buffer and randomly sampling batches of experiences for training. This helps to break correlations between consecutive experiences and provides more efficient use of experience data. Target networks, on the other hand, are used to stabilize the training process by fixing the parameters of a separate target network for a certain number of iterations before updating them with the parameters of the main network. The Deep $Q-$Learning training algorithm has two phases:

1. **Sampling:** We perform actions and store the observed experience tuples in a replay memory.

2. **Training:** Select a small batch of tuples randomly and learn from this batch using a gradient descent update step.

Algorithm 1 summarizes the usage of replay and target network in the DQN.

Algorithm 1 : Deep $Q-$Learning with Experience Replay [4]

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with random weights $\theta^{-} = \theta$
for episode $= 1, 2, \ldots, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\Phi_1 = \Phi(s_1)$
    for $t = 1, 2, \ldots, T$ **do**
        With probability ϵ select a random action a_t, otherwise select $a_t = \max_a Q^*(\Phi(s_t), a; \theta)$ ▷ Sampling
        Execute action a_t in emulator and observe reward r_t and O/P x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\Phi_{t+1} = \Phi(s_{t+1})$
        Store transition $(\Phi_t, a_t, r_t, \Phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $\left(\Phi_j, a_j, r_j, \Phi_{j+1}\right)$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}\left(\Phi_{j+1}, a'; \theta^{-}\right), & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $\left(y_j - Q\left(\Phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameter θ ▷ Traning
        In every C steps, reset $\hat{Q} = Q$
    end for
end for

As the reader may observe from Algorithm 1, during training, DQN aims to maximize the expected cumulative reward by selecting actions that maximize the estimated $Q-$value. Once trained, the DQN can be used to act in a real environment by selecting actions based on the learned $Q-$Values.

Overall, DQN represents a powerful approach to learning optimal decision-making policies in complex environments and has been successfully applied to a wide range of tasks, including playing Atari games, robotic control, and autonomous driving. There are several variants and extensions of the original Deep $Q-$Network (DQN) algorithm, each tailored to address specific challenges or improve performance in certain scenarios:

1. **Double DQN (DDQN):** It addresses the overestimation bias inherent in traditional Q-learning methods by decoupling action selection from action evaluation.

2. **Prioritized Experience Replay (PER):** It prioritizes experiences based on their importance in learning, focusing more on experiences that lead to larger temporal difference errors.

3. **Dueling DQN:** It introduces a network architecture that separately estimates the value of being in a state and the advantage of taking each action, allowing for more efficient learning.

4. **Rainbow:** This variant of DQN, that is a combination of various extensions including DDQN, PER, Dueling DQN, and others, aims to achieve state-of-the-art performance by leveraging the strengths of each individual component.

5. **Distributional DQN:** It models the distribution of returns rather than their expected values, enabling more robust and flexible representations of uncertainty and variability in the environment

These variants and extensions highlight the ongoing research efforts to enhance the capabilities and effectiveness of DQN-based algorithms across a wide range of reinforcement learning tasks. Now, let's restrict our attention to the Dueling Deep $Q-$Learning.

2.1.2 Double Deep $Q-$Network

The previous section described the main components of DQN as presented in [4]. The paper of dueling DQN [9] uses the improved Double DQN (DDQN) learning algorithm of van Hasselt et al. [8], in 2015. In $Q-$learning and DQN, the max operator uses the same values to both select and evaluate an action. In 2010, van Hasselt [7] showed that this can therefore lead to overoptimistic value estimates. To mitigate this problem, DDQN uses the following target:

$$y_i = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta_i); \theta^-) \tag{8}$$

DDQN is the same as for DQN, as discussed above, but with the target y_i replaced by Expression 8. Algorithm 2 demonstrates the principle of DDQN.

Algorithm 2 : Double Deep $Q-$Network Algorithm [9]

Initialize the empty replay buffer $\mathcal{D}$; initial network parameter θ and the copy of θ; that is — θ^-
Let N_r be the replay buffer maximum size, N_b be the training batch size and N^- be the target network replacement frequency.
for episode $e \in \{1, 2, 3, \dots, M\}$ **do**
    Initialize the frame sequence $x \leftarrow ()$
    for $t \in \{1, 2, \dots, T\}$ **do**
        Set state $s \leftarrow x$, sample action $a \sim \pi_{\mathcal{B}}$
        Sample next frame x^t from environment $\mathcal{E}$ given (s, a) and receive reward r, and append x^t to x
        if $|x| > N_f$ **then**
            Delete oldest frame $x_{t_{min}}$ from x
        end if
        Set $s' \leftarrow x$, and add transition tuple (s, a, r, s') to $\mathcal{D}$, replacing the oldest tuple if $|\mathcal{D}| \geqslant N_r$
        Sample a minibatch of N_b tuples $(s, a, r, s') \sim \mathcal{U}(\mathcal{D})$
        Construct target values, one for each of the N_b tuples
        Define $a^{max}(s'; \theta) = \arg\max_{a'} Q(s', a'; \theta)$
        $y_j \leftarrow \begin{cases} r & \text{if } s' \text{ is terminal} \\ r + \gamma Q(s', a^{max}(s'; \theta); \theta^-), & \text{otherwise} \end{cases}$
        Do a gradient descent step with loss $\left\| y_j - Q(s, a; \theta) \right\|^2$
        Replace target parameters $\theta^- \leftarrow \theta$ in every N^- steps
    end for
end for

2.1.3 Towards decoupling the State Values and Adavantages

Introduced in 2016, by Google Deepmind [9], Dueling DQN is an extension of the DQN algorithm, designed to improve learning efficiency by decomposing the $Q-$Value function into two separate streams:

1. one estimating the state value, and
2. the other estimating the advantage of each action.

The two streams are combined via a special aggregating layer to produce an estimate of the state-action value function Q. This dueling network should be understood as a single Q network with two streams that replaces the popular single-stream Q network in existing algorithms such as Deep Q-Networks (DQN; [4]). The dueling network automatically produces separate estimates of the state value function and advantage function, without any extra supervision.

2.1.4 Variants and architectures of Dueling DQN

The fundamental idea driving the architecture of Dueling DQN, depicted in Figure 6, is the recognition that in numerous states, it is unnecessary to evaluate the value of each potential action. For instance, in the context of the Enduro game, the critical decision of whether to move left or right only becomes relevant when a collision is imminent. While certain states demand precise action selection, many others bear no significant consequence based on the chosen action. However, for algorithms reliant on bootstrapping, accurately estimating state values holds considerable significance across all states.

The dueling DQN architecture employs two separate sequences (or streams) of fully connected layers instead of a single sequence. These streams are configured to independently estimate the value and advantage functions. Subsequently, the outputs of these streams are integrated to generate a unified Q function. Similar to the approach outlined in [4], the network produces a set of Q values, each corresponding to a distinct action.

Given that the dueling network outputs a Q function, it can be effectively trained using various existing algorithms, such as DDQN and SARSA. Furthermore, it stands to benefit from any enhancements made to these algorithms, including advancements in replay memory, exploration policies, intrinsic motivation, and so forth. Notably, the module responsible for amalgamating the outputs from the two streams of fully connected layers to derive a Q estimate necessitates careful and deliberate design considerations.

From the expressions for advantage $Q^\pi(s,a) = V^\pi(s) + A^\pi(s,a)$ and state-value $V^\pi(s) = \mathbb{E}_{a\sim\pi(s)}[Q^\pi(s,a)]$, it follows that $\mathbb{E}_{a\sim\pi(s)}[A^\pi(s,a)] = 0$. Moreover, for a deterministic policy, $a^* = \arg\max_{a'\in\mathcal{A}} Q(s,a')$, it follows that $Q(s,a^*) = V(s)$ and hence $A(s,a^*) = 0$.

Let us consider the dueling network shown in Figure 6, where we make one stream of fully-connected layers output a scalar $V(s;\theta,\beta)$, and the other stream output an $|\mathcal{A}|-$dimensional vector $A(s,a;\theta,\alpha)$. Here, θ denotes the parameters of the convolutional layers, while α and β are the parameters of the two streams of fully-connected layers. Using the definition of advantage, we might be tempted to construct the aggregating module as follows:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + A(s,a;\theta,\alpha), \tag{9}$$

Note that this expression applies to all (s,a) instances; that is, to express equation 9 in matrix form we need to replicate the scalar, $V(s;\theta,\beta)$, precisely $|\mathcal{A}|$ times.

Nevertheless, it is essential to bear in mind that $Q(s,a;\theta,\alpha,\beta)$ merely represents a parameterized approximation of the actual $Q-$function. Additionally, it would be erroneous to deduce that $V(s;\theta,\beta)$

serves as a reliable estimator for the state-value function, or to assume equivalently that $A(s, a; \theta, \alpha)$ furnishes an accurate estimation of the advantage function.

Equation 9 is unidentifiable in the sense that given Q we cannot recover V and A uniquely. To see this, add a constant to $V(s; \theta, \beta)$ and subtract the same constant from $A(s, a; \theta, \alpha)$. This constant cancels out resulting in the same Q value. This lack of identifiability is mirrored by poor practical performance when this equation is used directly.

To address this issue of identifiability, we can force the advantage function estimator to have zero advantage at the chosen action. That is, we let the last module of the network implement the forward mapping:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha). \qquad \text{(Type II)}$$

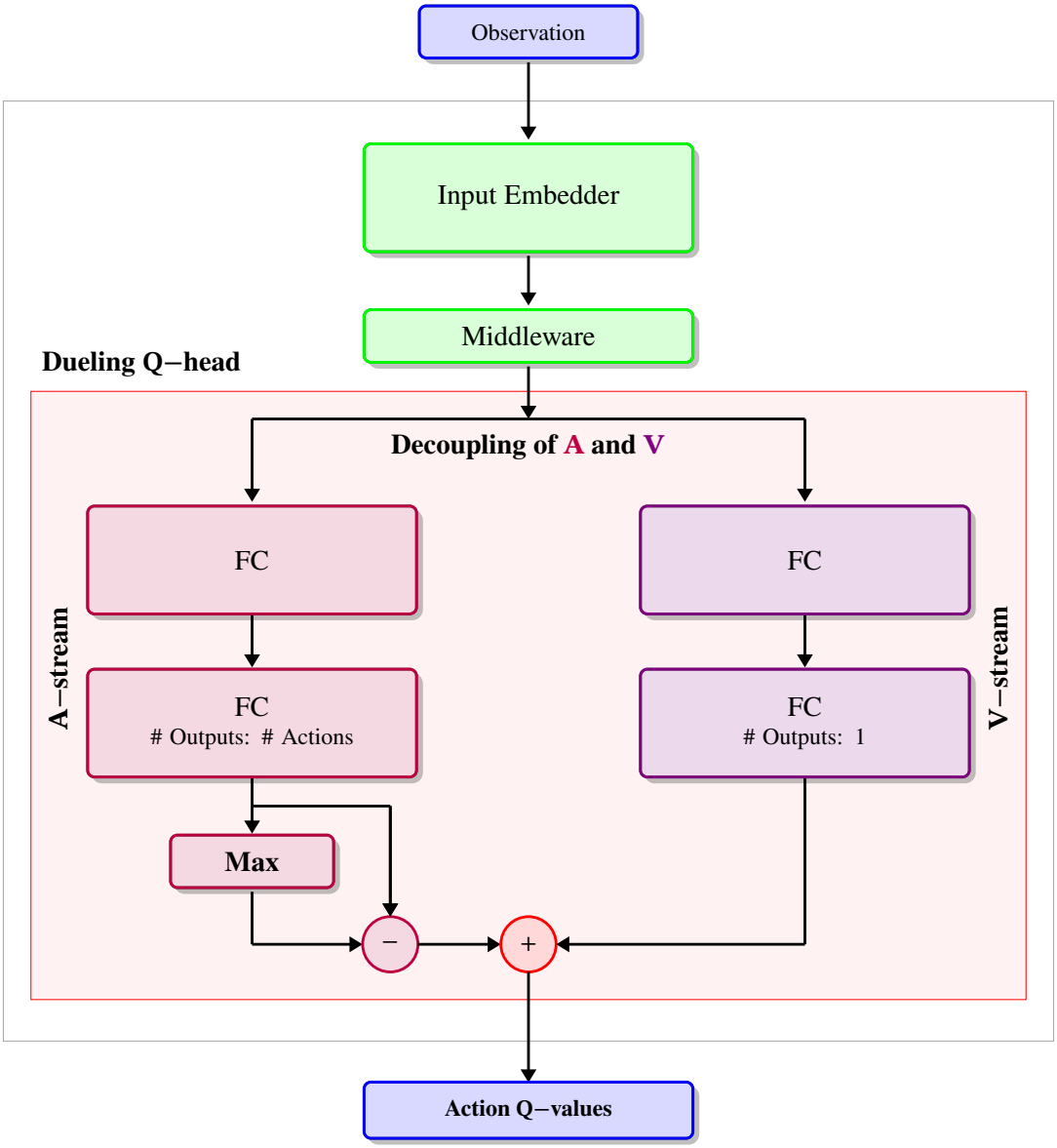


Figure 5: Architecture of Dueling DQN: Type II

Now, for $a^* = \arg\max_{a' \in \mathcal{A}} Q(s, a'; \theta, \alpha, \beta) = \arg\max_{a' \in \mathcal{A}} A(s, a'; \theta, \alpha)$, we obtain $Q(s, a^*; \theta, \alpha, \beta) = V(s; \theta, \beta)$. Hence, the stream $V(s; \theta, \beta)$ provides an estimate of the value function, while the other stream produces an estimate of the advantage function.

An alternative module replaces the max operator with an average:

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

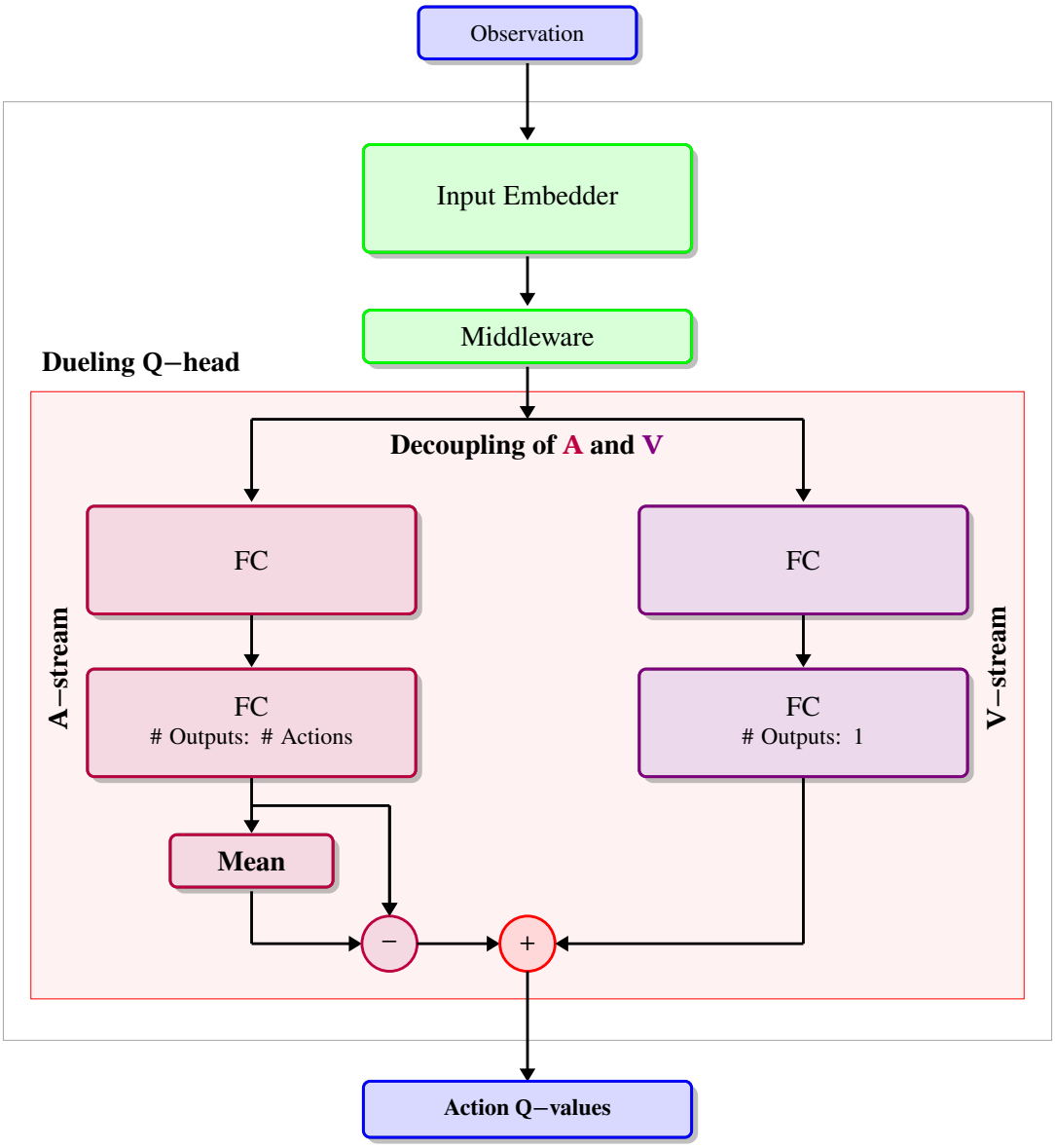


Figure 6: Architecture of Dueling DQN: Type I

On one hand, this adjustment deviates from the original interpretations of V and A due to their constant offset, yet on the other hand, it enhances optimization stability: by utilizing Equation Type I, the fluctuations in advantages only need to match the mean, instead of having to counterbalance changes

in the optimal action's advantage as indicated in Equation Type II.

It's worth mentioning that while subtracting the mean in Equation Type I aids in identifiability, it doesn't alter the relative ranking of the A (and thus Q) values, thereby preserving any greedy or ϵ-greedy policy based on Q values from Equation 9. During decision-making, evaluating the advantage stream alone is sufficient.

Importantly, Equation Type II is integrated and executed within the network structure, rather than being treated as a separate algorithmic step. Training the dueling architectures, akin to standard Q networks, solely involves back-propagation. The estimations $V(s;\theta,\beta)$ and $A(s,a;\theta,\alpha)$ are computed automatically without necessitating additional supervision or algorithmic adjustments.

Given that the dueling architecture maintains the same input-output interface as standard Q networks, all learning algorithms associated with Q networks (e.g., DDQN and SARSA) can be readily applied to train the dueling architecture.

2.1.5 Implementation of classes and methods related to Dueling DQN

The implementation concerned classes and methods in Python-3.8.10 goes as follows:

```
# Experience Replay Buffer
class ReplayBuffer:
    def __init__(self, buffer_size):
        self.buffer = deque(maxlen=buffer_size)

    def add(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        samples = random.sample(self.buffer, batch_size)
        states, actions, rewards, next_states, dones = zip(*samples)
        return np.array(states), np.array(actions), np.array(rewards),
         ↪  np.array(next_states), np.array(dones)

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

```
# Dueling DQN
class DuelingDQN(nn.Module):
    def __init__(self, state_dim, action_dim, update_type=1, args=args):
        super(DuelingDQN, self).__init__()
        self.fc1 = nn.Linear(state_dim, args.h1_dim)
        self.fc2 = nn.Linear(args.h1_dim, args.h2_dim)

        self.value = nn.Linear(args.h2_dim, 1)
        self.advantage = nn.Linear(args.h2_dim, action_dim)
        self.update_type = update_type
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity='relu', mode='fan_in') #
                 ↪  He initialization for ReLU
```

```
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        value = self.value(x)
        advantage = self.advantage(x)

        if self.update_type == 1:
            q = value + (advantage - advantage.mean(dim=-1, keepdim=True))
        elif self.update_type == 2:
            q = value + (advantage - advantage.max(dim=-1, keepdim=True)[0])
        else:
            raise NotImplementedError("Update type not implemented")

        return q
```

```
# Dueling DQN Agent
class DuelingDQNAgent:
    def __init__(self, state_dim, action_dim, update_type=1, args=args):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.update_type = update_type
        self.args = args
        self.q_net = DuelingDQN(state_dim, action_dim, update_type, self.args).to(device)
        self.target_net = copy.deepcopy(self.q_net)
        self.target_net.eval()
        self.optimizer = optim.Adam(self.q_net.parameters(), lr=self.args.lr)
        self.loss_fn = nn.MSELoss()
        self.replay_buffer = ReplayBuffer(self.args.buffer_size)
        self.steps = 0

    def act(self, state, epsilon):
        if np.random.rand() < epsilon:
            return np.random.choice(self.action_dim)
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        q_values = self.q_net(state)
        return q_values.argmax().item()

    def update(self, state, action, reward, next_state, done):
        self.replay_buffer.add(state, action, reward, next_state, done) # Add to replay
        ↪  buffer
        if len(self.replay_buffer) < self.args.batch_size: # Wait until buffer is filled
            return 0

        states, actions, rewards, next_states, dones =
        ↪  self.replay_buffer.sample(self.args.batch_size)
        states = torch.FloatTensor(states).to(device)
        actions = torch.LongTensor(actions).to(device)
        rewards = torch.FloatTensor(rewards).to(device)
        next_states = torch.FloatTensor(next_states).to(device)
        dones = torch.FloatTensor(dones).to(device)
```

```
        q_values = self.q_net(states)
        next_q_values = self.target_net(next_states).detach()

        q_value = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)
        next_q_value = next_q_values.max(dim=-1)[0]
        target = rewards + self.args.gamma * next_q_value * (1 - dones)

        loss = self.loss_fn(q_value, target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step() # Update policy network

        self.steps += 1
        if self.steps % self.args.target_update_freq == 0:
            self.target_net.load_state_dict(self.q_net.state_dict()) # Update target
             ↪ network

        return loss.item()
```

2.2 Monte-Carlo REINFORCE

The Monte Carlo Policy Gradient, commonly known as REINFORCE, is a policy gradient method used in reinforcement learning. It involves an agent making sequential decisions within an environment. Unlike value-based methods (such as Q-learning), policy gradient methods work by directly updating the policy parameters to maximize the expected cumulative reward.

It considers methods for learning the policy parameter based on the gradient of some scalar performance measure $J(\theta)$ with respect to the policy parameter. The objective function $J(\theta)$ for the policy gradient is defined as:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T} r_{t+1}\right] \tag{10}$$

Note that $J(\theta)$ represents the cumulative reward; τ denotes a trajectory sampled from policy π_θ and as usual r_s refers to the reward at time s. Here the policy gradient assumes a stochastic (non-determinstic) policy and the policy function π_θ is parametrized by a neural network or other function approximations. These methods seek to maximize performance, so their updates approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J(\theta_t)} \tag{11}$$

2.2.1 Policy Gradient Methods

In policy gradient methods, the policy can be parameterized in any way, as long as $\pi(a|s,\theta)$ is differentiable wrt its parameters, that is, as long as $\nabla\pi(a|s,\theta)$ (the column vector of partial derivatives of $\pi(a|s,\theta)$ with respect to the components of θ) exists and is finite for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$, and $\theta \in \mathbb{R}^{d'}$. In practice, to ensure exploration we generally require that the policy never becomes deterministic (i.e., that $\pi(a|s,\theta) \in (0,1)$, for all s, a, θ).

The policy gradient theorem for the episodic case establishes that:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \tag{12}$$

Figure 7 captures the architecture for the implementation of the policy gradient method:

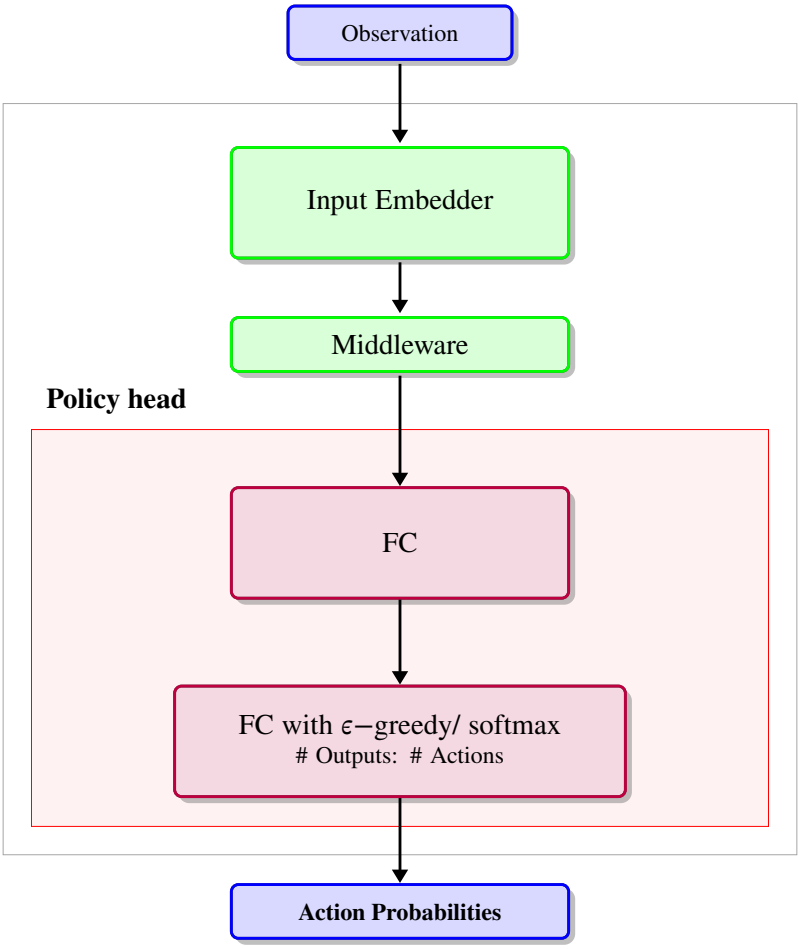


Figure 7: Architecture of Policy Gradient

2.2.2 REINFORCE: Monte Carlo Policy Gradient

Observe at the expression given in 12. Note that the RHS is a sum over states weighted by how often the states occur under the target policy π; if π is followed, then states will be encountered in these proportions. Thus:

$$\begin{aligned} \nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta) \\ &= \mathbb{E}_\pi \left[\sum_a q_\pi(s_t,a) \nabla \pi(a|s_t,\theta) \right] \end{aligned} \tag{13}$$

Consequently, we may formulate the stochastic gradient ascent algorithm as follows:

$$\theta_{t+1} = \theta_t + \alpha \sum_a \hat{q}(s_t,a,w) \nabla \pi(a|s_t,\theta) \tag{14}$$

where $\hat{q}$ is some learned approximation to q_π. This algorithm is known as an *all-actions method* as its update involves all of the actions. Having introduced this, we may restrict our attention to the classical REINFORCE algorithm (that was introduced by Williams [10] in 1992) whose update at time t involves just a_t, the one action actually taken at time t.

Note that

$$\begin{aligned}\nabla J(\theta) &\propto \mathbb{E}_\pi\left[\sum_a \pi(a|s_t,\theta) q_\pi(s_t,a)\frac{\nabla\pi(a|s_t,\theta)}{\pi(a|s_t,\theta)}\right] \\ &= \mathbb{E}_\pi\left[q_\pi(s_t,a_t)\frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\right] \text{ (Replacing } a \text{ by sample } a_t \sim \pi) \\ &= \mathbb{E}_\pi\left[G_t\frac{\nabla\pi(a_t|s_t,\theta)}{\pi(a_t|s_t,\theta)}\right] \text{ (because } \mathbb{E}_\pi\left[G_t|s_t,a_t\right] = q_\pi(s_t,a_t))\end{aligned}$$

Using this sample to instantiate the generic stochastic gradient ascent algorithm yields the REINFORCE update as follows:

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla\pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)} \qquad \text{(w/o baseline)}$$

This update is intuitively appealing. It involves adjusting parameters based on a combination of the return G_t and a gradient vector. This vector represents the direction in parameter space that maximizes the likelihood of repeating the action a_t in future instances of state s_t. The update incrementally adjusts the parameter vector in this direction, with the magnitude proportional to the return and inversely proportional to the probability of the action taken. This approach ensures that parameters favor actions with higher returns while preventing overly frequent actions from gaining undue advantage.

It's important to note that REINFORCE utilizes the complete return up to time t, encompassing all future rewards until the episode's conclusion. Therefore, REINFORCE is akin to Monte Carlo algorithms and is applicable strictly in episodic scenarios, where updates are made retrospectively after each episode concludes.

The final pseudocode update appears somewhat different from the REINFORCE update rule. One notable distinction is the use of the compact expression $\nabla \ln \pi(a_t|s_t,\theta_t)$ for the fractional vector $\frac{\nabla\pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}$. However, these two expressions for the vector are equivalent, as indicated by the identity $\nabla \ln x = \frac{\nabla x}{x}$.

This vector is commonly referred to as the eligibility vector in the literature and is the sole instance where the policy parameterization is involved in the algorithm.

Algorithm 3 : REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_* without baseline [6]

Input: a differentiable policy parameterization $\pi(a|s,\theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter: $\theta \in \mathbb{R}^{d'}$ (e.g. to $\mathbb{0}$)
for each episode **do**
    Generate an episode $s_o, a_o, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ following $\pi(\cdot|\cdot,\theta)$
    for each step of episode $t = 0, 1, 2, \ldots, T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
        $\theta \leftarrow \theta + \alpha\gamma^k G \nabla \ln \pi(a_t|s_t,\theta)$
    end for
end for

2.2.3 REINFORCE with Baseline

The policy gradient theorem 12 can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$ as follows:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a (q_\pi(s,a) - b(s)) \nabla \pi(a|s,\theta) \tag{15}$$

The baseline can be any function, even a random variable, as long as it does not vary with a; the equation remains valid because the subtracted quantity is zero.

The policy gradient theorem with baseline 15 can be used to derive an update rule. The update rule that we end up with is a new version of REINFORCE that includes a general baseline $V(\cdot,\Phi)$:

$$\theta_{t+1} = \theta_t + \alpha\Big(G_t - V(S_t,\Phi)\Big)\frac{\nabla \pi\left(A_t|S_t,\theta_t\right)}{\pi\left(A_t|S_t,\theta_t\right)} \qquad \text{(w/ Baseline)}$$

Since the baseline could potentially be uniformly zero, this update represents a clear extension of the REINFORCE method. Generally, while the baseline maintains the expected value of the update, its impact on variance can be significant. For instance, as demonstrated in Section 2.8, a comparable baseline can markedly diminish variance, thus accelerating the learning process in gradient bandit algorithms. In those algorithms, the baseline merely constitutes a single value, typically the average of observed rewards. However, in the context of Markov Decision Processes (MDPs), the baseline should adapt according to the state. In certain states where all actions possess high values, a higher baseline is necessary to distinguish between actions of varying value, whereas in other states where all actions yield low values, a lower baseline is more suitable.

One natural choice for the baseline is an estimate of the state value, $\hat{v}(S_t,w)$, where $w \in \mathbb{R}^m$ is a weight vector learned. Because REINFORCE is a Monte Carlo method for learning the policy parameter, θ, it seems natural to also use a Monte Carlo method to learn the state-value weights, w. A complete pseudocode algorithm for REINFORCE with a baseline using such a learned state-value function as the baseline is given in the box below.

Algorithm 4 : REINFORCE with Baseline (episodic), for estimating $\pi_\theta \approx \pi_*$ [6]

Input: A differentiable policy parametrization $\pi(a|s,\theta)$
Input: A differentiable state-value function parametrization $\hat{v}(s,w)$
Algorithm Parameters: Step size: $\alpha^\theta > 0$, $\alpha^w > 0$
Initialize policy parameter: $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$ (e.g., to $\mathbb{0}$)
for each episode **do**
    Generate an episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot,\theta)$
    for each step of episode $t = 0, 1, 2, \ldots, T-1$ **do**
        $G \leftarrow \sum_{k=t+1}^{T} \gamma r^{k-t-1}$
        $\delta \leftarrow G - \hat{v}(s_t, w)$
        $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(s_t, w)$
        $\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla \ln \pi(a_t|s_t,\theta)$
    end for
end for

2.2.4 Implementation of concerned classes and methods in Monte-Carlo REINFORCE

The implementation classes and methods related to Monte-Carlo REINFORCE in Python-3.8.10 goes as follows:

```
# Policy Network
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, args=args):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, args.h1_dim)
        self.fc2 = nn.Linear(args.h1_dim, args.h2_dim)
        self.fc3 = nn.Linear(args.h2_dim, action_dim)
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity='relu', mode='fan_in')
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.softmax(x, dim=-1)
```

```
# Value Network
class ValueNetwork(nn.Module):
    def __init__(self, state_dim, args=args):
        super(ValueNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, args.h1_dim)
        self.fc2 = nn.Linear(args.h1_dim, args.h2_dim)
```

```
        self.fc3 = nn.Linear(args.h2_dim, 1)
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Linear):
                nn.init.kaiming_normal_(m.weight, nonlinearity='relu', mode='fan_in') #
                 ↪  He initialization for ReLU
                nn.init.constant_(m.bias, 0)

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

```
# Monte Carlo REINFORCE Agent
class MonteCarloREINFORCEAgent:
    def __init__(self, state_dim, action_dim, baseline=False, args=args):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.args = args
        self.policy_net = PolicyNetwork(state_dim, action_dim, self.args).to(device)
        self.value_net = ValueNetwork(state_dim, self.args).to(device)
        self.optimizer_theta = optim.Adam(self.policy_net.parameters(),
         ↪  lr=self.args.lr_theta)
        self.optimizer_w = optim.Adam(self.value_net.parameters(), lr=self.args.lr_w)
        self.baseline = baseline
        self.loss_fn = nn.MSELoss()
        self.steps = 0

    def act(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(device)
        probs = self.policy_net(state)
        action = torch.distributions.Categorical(probs).sample()
        return action.item()

    def update(self, trajectory):
        states, actions, rewards = zip(*trajectory) # Unzip trajectory
        states = torch.FloatTensor(states).to(device)
        actions = torch.LongTensor(actions).to(device)
        rewards = torch.FloatTensor(rewards).to(device)

        returns = self._compute_returns(rewards)

        if self.baseline: # Subtract baseline from returns
            values = self.value_net(states).squeeze()
            loss_w = self.loss_fn(values, returns)
            returns = returns - values.detach()
            self.optimizer_w.zero_grad()
            loss_w.backward()
            self.optimizer_w.step() # Update value network
```

```
        log_probs = torch.log(self.policy_net(states))
        log_probs_actions = log_probs.gather(1, actions.unsqueeze(1)).squeeze()
        loss_theta = -torch.mean(log_probs_actions * returns)

        self.optimizer_theta.zero_grad()
        loss_theta.backward()
        self.optimizer_theta.step() # Update policy network

        self.steps += 1
        return loss_theta.item()

    def _compute_returns(self, rewards):
        returns = []
        G = 0
        for r in reversed(rewards):
            G = r + self.args.gamma * G
            returns.insert(0, G)
        return torch.FloatTensor(returns).to(device)
```

2.3 The Implementation of Training

The implementation of the training for both Dueling DQN and Monte-Carlo REINFORCE in Python-3.8.10 goes as follows:

```
# Training for Dueling DQN and REINFORCE
class Trainer:
    def __init__(self, env_name, agent_type, update_type=None, baseline=False, args=args,
     ↪  save_results=True):
        self.env_name = env_name
        self.agent_type = agent_type
        self.update_type = update_type
        self.baseline = baseline
        self.env = gym.make(env_name)
        self.state_dim = self.env.observation_space.shape[0]
        self.action_dim = self.env.action_space.n
        self.args = args
        self.save_results = save_results

    def get_agent(self):
        if self.agent_type == "dueling_dqn":
            return DuelingDQNAgent(self.state_dim, self.action_dim, self.update_type,
             ↪  self.args)
        elif self.agent_type == "reinforce":
            return MonteCarloREINFORCEAgent(self.state_dim, self.action_dim,
             ↪  self.baseline, self.args)
        else:
            raise NotImplementedError("Agent not implemented")

    def train(self):
        rewards = np.zeros((self.args.runs, self.args.max_episodes))
        losses = np.zeros((self.args.runs, self.args.max_episodes))
        returns = np.zeros((self.args.runs, self.args.max_episodes))
        for run in tqdm(range(self.args.runs)):
```

```
            seed_all(seed=run)
            self.agent = self.get_agent()
            rewards[run], losses[run], returns[run] = self.train_single_run()
            clear_output(wait=True)
        self.rewards = rewards
        self.losses = losses
        self.returns = returns
        if self.save_results:
            self.save() # Save the results
        return rewards, losses, returns

    def train_single_run(self):
        if self.agent_type == "dueling_dqn":
            return self.train_DuelingDQN()
        elif self.agent_type == "reinforce":
            return self.train_MC_REINFORCE()
        else:
            raise NotImplementedError("Agent not implemented")

    def train_DuelingDQN(self):
        rewards = []
        losses = []
        episodic_returns = []
        frames = []  # List to store frames
        for episode in (range(self.args.max_episodes)):
            state = self.env.reset()
            episode_reward = 0
            episodic_return = 0
            for step in range(self.args.max_steps):
                epsilon = max(self.args.min_epsilon, self.args.epsilon0 *
                 ↪  self.args.epsilon_decay**self.agent.steps)
                action = self.agent.act(state, epsilon)
                next_state, reward, done, _ = self.env.step(action)
                loss = self.agent.update(state, action, reward, next_state, done)
                state = next_state
                episode_reward += reward
                episodic_return = self.args.gamma * episodic_return + reward
                if episode == self.args.max_episodes - 1:
                    frames.append(self.env.render(mode='rgb_array'))  # Record frames
                if done:
                    break
            rewards.append(episode_reward)
            losses.append(loss)
            episodic_returns.append(episodic_return)
            if episode % self.args.print_freq == 0:
                print("Episode: {}, Reward: {}".format(episode, episode_reward))
        self.env.close()
        self.save_simulation(frames,
         ↪  f"simulations/{self.env_name}_{self.agent_type}.mp4")
        return rewards, losses, episodic_returns

    def train_MC_REINFORCE(self):
        rewards = []
        losses = []
        episodic_returns = []
```

```
        frames = []  # List to store frames
        for episode in (range(self.args.max_episodes)):
            state = self.env.reset()
            episode_reward = 0
            episodic_return = 0
            trajectory = []
            for step in range(self.args.max_steps):
                action = self.agent.act(state)
                next_state, reward, done, _ = self.env.step(action)
                trajectory.append((state, action, reward))
                state = next_state
                episode_reward += reward
                episodic_return = self.args.gamma * episodic_return + reward
                if episode == self.args.max_episodes - 1:
                    frames.append(self.env.render(mode='rgb_array'))  # Record frames
                if done:
                    break
            rewards.append(episode_reward)
            loss = self.agent.update(trajectory)
            losses.append(loss)
            episodic_returns.append(episodic_return)
            if episode % self.args.print_freq == 0:
                print("Episode: {}, Reward: {}".format(episode, episode_reward))
        self.env.close()
        self.save_simulation(frames,
         ↪  f"simulations/{self.env_name}_{self.agent_type}.mp4")
        return rewards, losses, episodic_returns
```

3 HYPERPARAMETERS AND OTHER (CHOSEN) VALUES

This section represents the various hyperparameters used for training. The other parameters used for training are mentioned in the respective plots of each experiment.

3.1 Dueling DQN : Acrobot-v1

Batch Size	Buffer Size	Target Update Freq	Average Rewards Mean
1024	10000	20	-83.4460
128	10000	20	-83.5726
1024	10000	50	-85.3008
128	10000	50	-85.9546
32	10000	20	-91.4034
32	10000	50	-91.5146
128	1000	20	-93.7470
128	1000	50	-93.9198
32	1000	50	-94.2002
32	1000	20	-94.8462
1024	1000	20	-499.0578
1024	1000	50	-499.1188

3.2 Dueling DQN : CartPole-v1

Batch Size	Buffer Size	Target Update Freq	Average Rewards Mean
128	10000	20	335.6390
128	1000	20	323.8020
32	10000	20	322.5494
128	10000	50	322.4030
32	1000	20	305.1562
32	10000	50	302.2162
32	1000	50	295.2856
128	1000	50	292.4920
1024	10000	50	285.0252
1024	10000	20	283.0640
1024	1000	20	22.2286
1024	1000	50	21.9122

3.3 Monte-Carlo REINFORCE : Acrobot-v1

h1_dim	h2_dim	Average Rewards Mean
128	64	-165.4448
256	64	-261.7226
128	128	-264.9214
128	512	-369.1785
256	128	-417.1384

3.4 Monte-Carlo REINFORCE : CartPole-v1

h1_dim	h2_dim	Average Rewards Mean
128	512	278.7774
512	512	267.0242
128	128	266.6860
128	64	258.8150
256	128	209.5964
256	64	209.4506

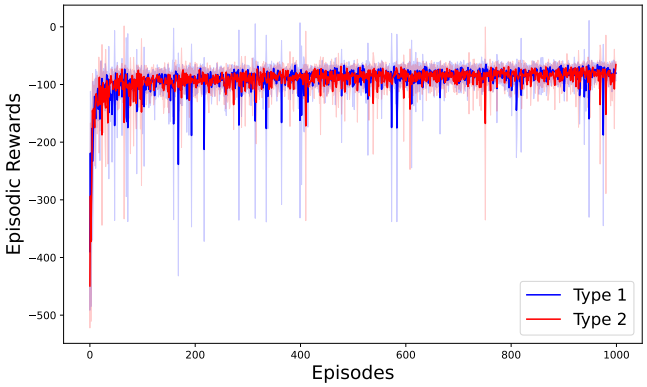
4 PLOTS

4.1 Dueling DQN

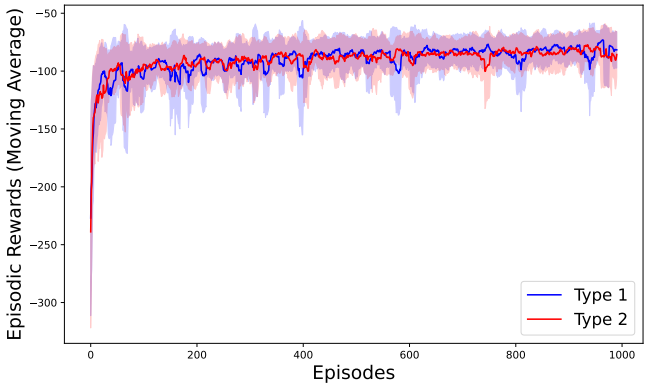
4.1.1 Acrobot v1

Parameter	Value
# Neurons (HL1)	128
# Neurons (HL2)	128
Discount Factor (γ)	0.99
Learning Rate (α)	0.0001
Optimizer	Adam
Loss Function	MSE
Weight Initialization Rule	Kaiming-He
Batch size	1024
Buffer size	10000
Target Update Frequency	20

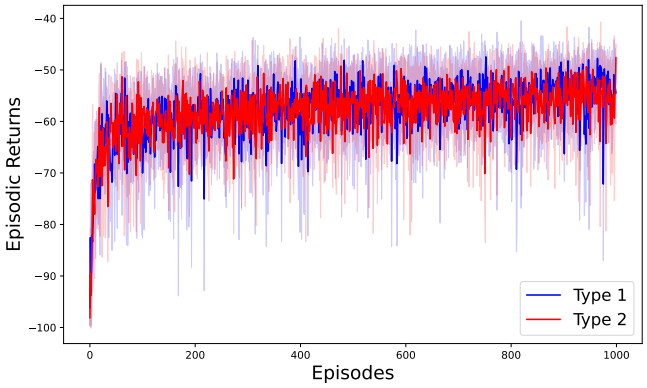
Parameter	Value
Max # Episodes	1000
Max # steps	500
# Runs	5
Initial Epsilon (ϵ_o)	1.0
Min Epsilon (ϵ_{min})	0.0001
Epsilon Decay (ϵ_{decay})	0.98^t
Window size for moving average Plot	20
Mean of Episodic Reward (Type I)	-83.4460
Simulation after training	Link



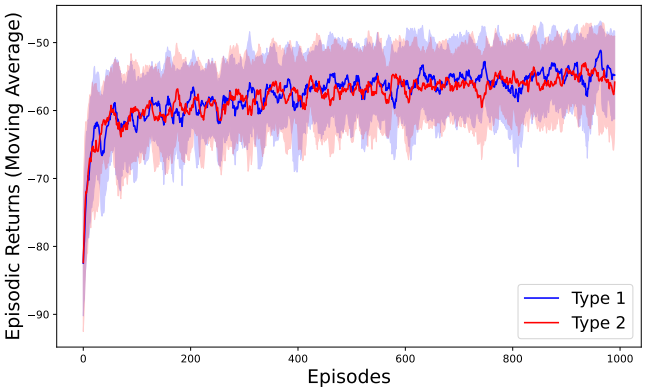
(a) Episodic Reward vs Episode



(b) Episodic Reward (moving avg) vs Episode



(c) Episodic Return vs Episode



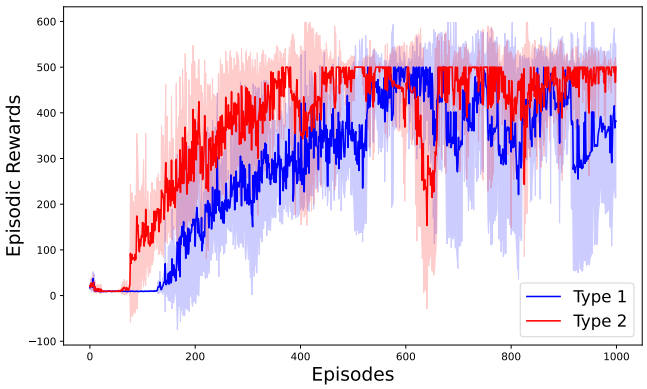
(d) Episodic Return (moving avg) vs Episode

Figure 8: Dueling DQN: Acrobot v1

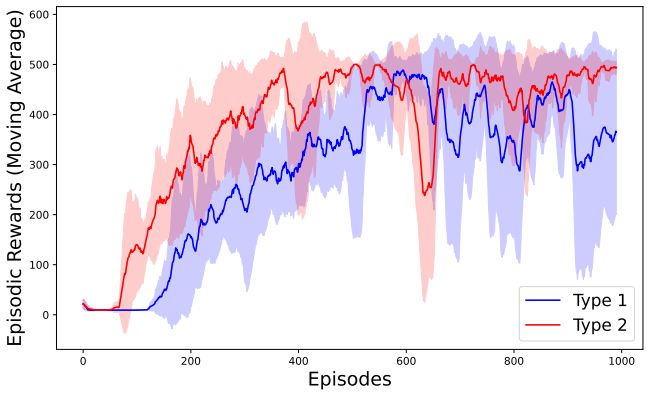
4.1.2 CartPole v1

Parameter	Value
# Neurons (HL1)	128
# Neurons (HL2)	128
Discount Factor (γ)	0.99
Learning Rate (α)	0.0001
Optimizer	Adam
Loss Function	MES
Weight Initialization Rule	Kaiming-He
Batch size	128
Buffer size	10000
Target Update Frequency	20

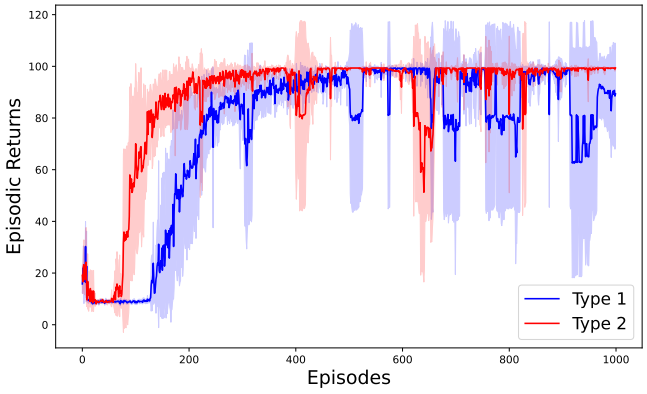
Parameter	Value
Max # Episodes	1000
Max # steps	500
# Runs	5
Initial Epsilon (ϵ_o)	1.0
Min Epsilon (ϵ_{min})	0.0001
Epsilon Decay (ϵ_{decay})	0.98^t
Window size for moving average Plot	20
Mean of Episodic Reward (Type I)	335.6390
Simulation after training	Link



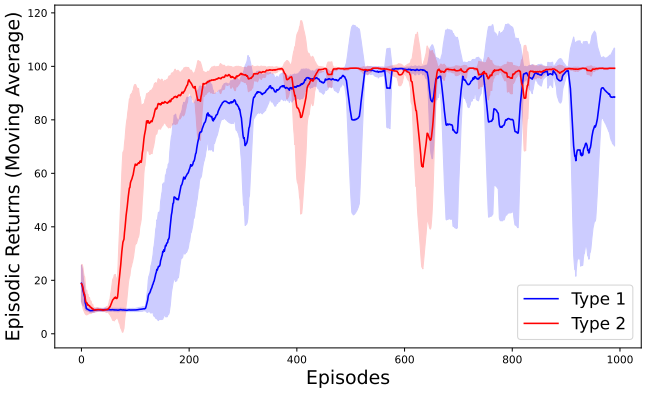
(a) Episodic Reward vs Episode



(b) Episodic Reward (moving avg) vs Episode



(c) Episodic Return vs Episode



(d) Episodic Return (moving avg) vs Episode

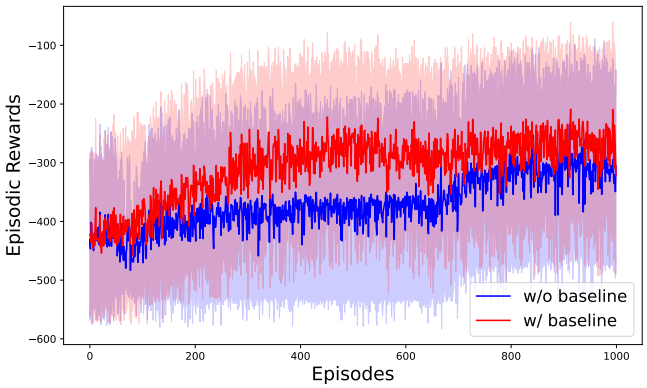
Figure 9: Dueling DQN: CartPole v1

4.2 Monte−Carlo REINFORCE

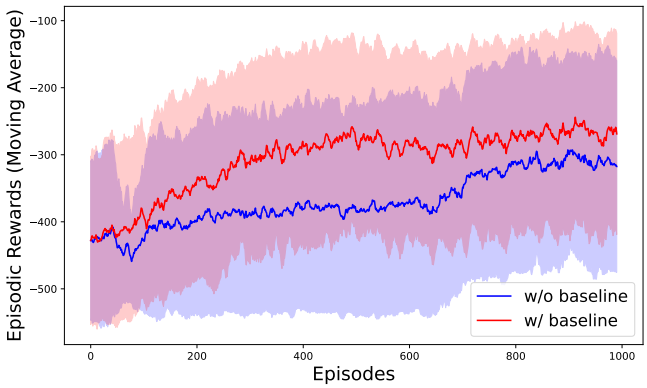
4.2.1 Acrobot v1

Parameters	Value
# Neurons (HL1)	128
# Neurons (HL2)	64
Discount Factor (γ)	0.99
Learning Rate [Policy] (η_θ)	0.0001
Learning Rate [Value] (η_w)	0.0001
Optimizer	Adam
Weight Initialization Rule	Kaiming-He

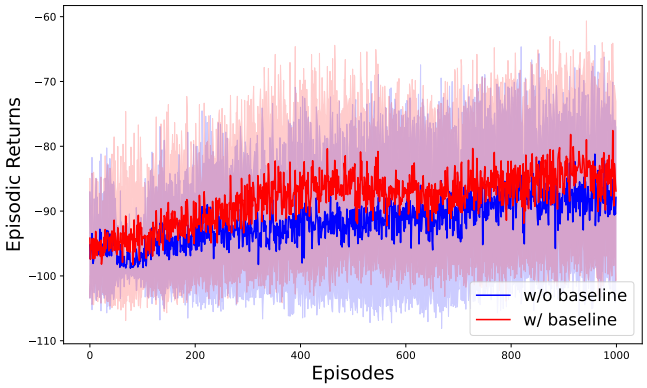
Parameters	Value
Loss Function	MSE
Max # Episodes	1000
Max # steps	500
# Runs	5
Window size for moving average Plot	20
Mean of Episodic Reward (Type I)	-165.4448
Simulation after training	Link



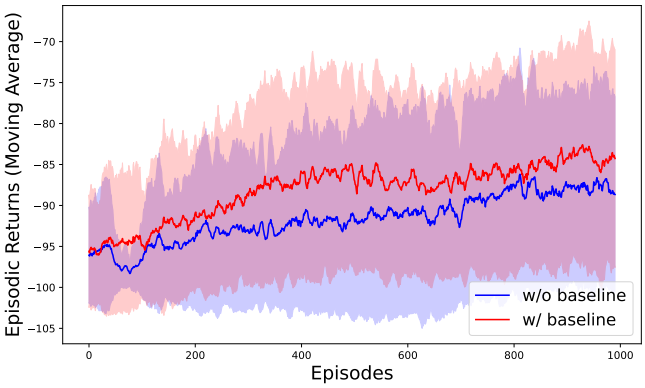
(a) Episodic Reward vs Episode



(b) Episodic Reward (moving avg) vs Episode



(c) Episodic Return vs Episode



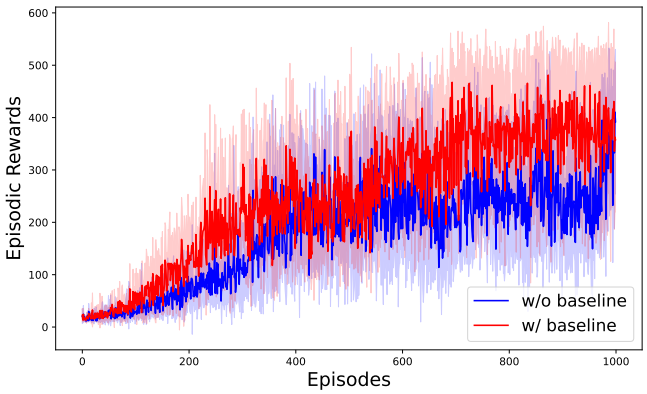
(d) Episodic Return (moving avg) vs Episode

Figure 10: Monte-Carlo REINFORCE: Acrobot v1

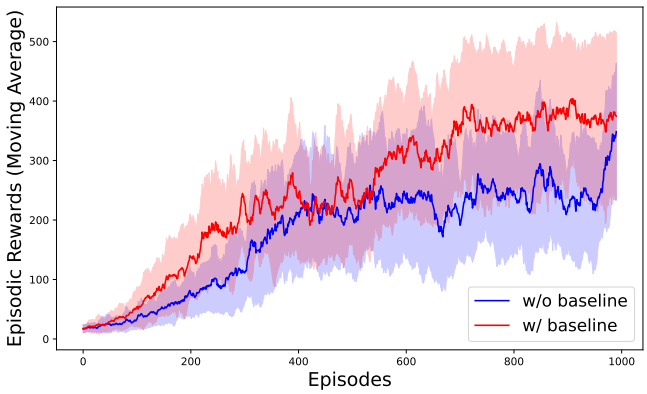
4.2.2 CartPole v1

Parameters	Value
# Neurons (HL1)	128
# Neurons (HL2)	512
Discount Factor (γ)	0.99
Learning Rate [Policy] (η_θ)	0.0001
Learning Rate [Value] (η_w)	0.0001
Optimizer	Adam
Weight Initialization Rule	Kaiming-He

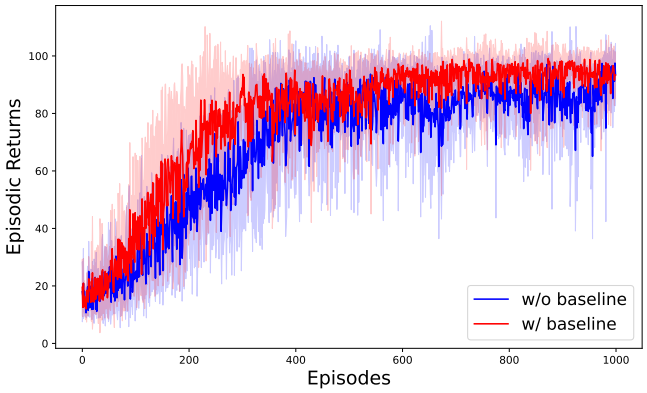
Parameters	Value
Loss Function	MSE
Max # Episodes	1000
Max # steps	500
# Runs	5
Window size for moving average Plot	20
Mean of Episodic Reward (Type I)	278.7774
Simulation after training	Link



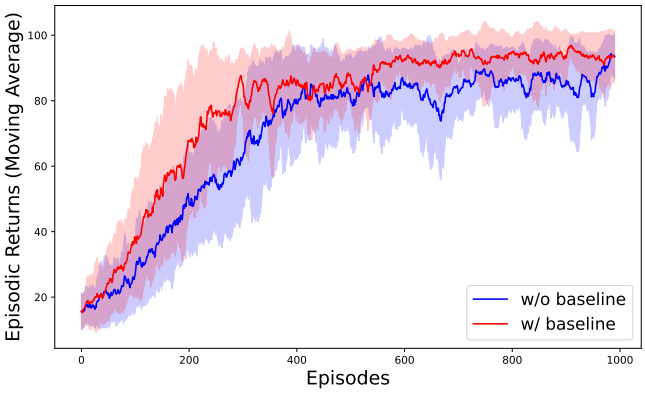
(a) Episodic Reward vs Episode



(b) Episodic Reward (moving avg) vs Episode



(c) Episodic Return vs Episode



(d) Episodic Return (moving avg) vs Episode

Figure 11: Monte-Carlo REINFORCE: CartPole v1

5 Observation and Inference

This section sheds light on the observations and conclusions drawn from the information presented in the preceding sections.

5.1 Duelling DQN: Type I (avg) × Type II (max)

1. Clearly, for Acrobot v1, as far as the mean over 5 runs or the moving average mean (for episodic reward or episodic return) is concerned, both Type I and Type II methods concerning Duelling DQN performed analogously, as that can be seen in the plots. But, note that Type I suffered more variance suggesting that it has more stability of optimization as compared to Type II.

2. In CartPole v1, Type I converged to a better episodic reward/ return as compared to Type II. Analogous to the previous observation, here we observed a higher variance in Type I as compared to Type II.

5.2 Monte-Carlo REINFORCE: w/o Baseline × w/ Baseline

The following inferences were made: Note that Type II performed better than Type I. Here is why:

1. **Variance Reduction:** Introducing a baseline reduces the variance of the estimated returns. In Monte Carlo RL, the estimated return for a state-action pair is the sum of rewards from that state-action pair until the end of the episode. By subtracting a baseline, which is typically the state-value function estimate for that state, the variance of these returns can be reduced. Lower variance typically leads to more stable and faster learning.

2. **Bias Control:** Adding a baseline introduces a bias into the estimation, but this bias is usually beneficial. It's biased in the sense that the estimator is no longer unbiased for the true return, but this bias can actually be desirable because it often leads to lower mean squared error in the estimates.

3. **Faster Learning:** With reduced variance and controlled bias, the learning process tends to converge faster. This is because the estimates are less noisy and thus provide a clearer signal to the learning algorithm.

4. **Improved Exploration-Exploitation Tradeoff:** The reduced variance can help the agent better discern which actions are more promising, allowing for a more efficient exploration-exploitation tradeoff. This means the agent can focus its exploration efforts on actions that are more likely to lead to higher returns.

5. **Stability:** Baselines can also stabilize the learning process by preventing large fluctuations in the estimated returns. This stability can make the learning algorithm more robust and less sensitive to small changes in the environment or the learning parameters.

Overall, by using a baseline in Monte Carlo Reinforcement Learning, the algorithm can achieve better performance by reducing variance, controlling bias, accelerating learning, improving exploration-exploitation tradeoff, and enhancing stability. However, it's worth noting that the choice of baseline and its implementation can impact the performance, and it may require experimentation to find the most suitable baseline for a given problem.

6 REFERENCES

[1] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. In *International Conference on Learning Representations (ICLR)*, 2015.

[2] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(3):571–582, 1990.

[3] Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1993.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[5] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems 8 (NIPS 1995)*, pages 1038–1044. MIT Press, 1996.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition, 2018.

[7] Hado van Hasselt. Double q-learning. In *Advances in Neural Information Processing Systems (NIPS)*, volume 23, pages 2613–2621, 2010.

[8] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[10] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, May 1992.