Solving Control Problems using Reinforcement Learning

ARI3212 Individual Module Assignment

Nicholas Vella
nicholas.vella.21@um.edu.mt
University of Malta

1 Introduction

1.1 Reinforcement Learning

Reinforcement learning is a field at the intersection of machine learning and optimal control. It focuses primarily on how an intelligent agent should act in a dynamic environment and maximize the total reward. But unlike other forms of machine learning, such as supervised learning (with labelled input-output pairs), or unsupervised learning, RL works based on different principles. Instead of detailed directives, the agent relies on experiments to learn the environment and obtain rewards or punishment. Venturing into the unknown versus exploiting existing knowledge. At the core of RL lies a constant balance between exploration and exploitation. Thus the agent has to choose when it can afford to take risks and when it cannot. The environment is usually represented as an MDP (Markov decision process), with states, actions, transition probabilities and the objective of determining a policy that maximises cumulative expected reward. RL has been applied effectively in a wide variety of fields, such as game theory, control theory, operations research, information theory and multi-agent systems. This is what underlies AI-based decision-making in inventory management, robotics and game playing. All told, RL is illustrative of how we living creatures learn through our engagement with the larger world. This is a strong method for training computers about choices, so it naturally has become an important topic in artificial intelligence. In the same way that we learn through life, agents can adjust to circumstances using techniques like Q-learning and SARSA.

1.2 How does RL differ from other ML approaches?

Reinforcement Learning (RL) is a distinct approach from within the Machine Learning (ML) paradigm. Unlike ML methods, RL learns by interacting with its environment and always attempts to maximize the reward signal. RL, unlike Supervised Learning, does not depend on labelled data. Instead, an agent learns from the results of its actions. Moreover, it does not require any external supervision, instead, an agent learns from the consequences of its actions.

In Supervised Learning, there is a clear target output for each of the training set's inputs. In RL the agent only receives a reward signal as feedback, although this may not necessarily be related to its actions. This presents one of the key challenges in RL, which is delayed rewards.

In Unsupervised Learning, the objective is to find structure in the input data (unlabeled) without human assistance. Ideal for clustering, data exploration and image recognition. On the other hand, RL focuses on finding a policy that maximizes the total reward, by balancing exploration (searching for new information) and exploitation (utilizing existing information).

In contrast to other ML methods, RL utilizes the aspect of time, dealing with problems that are sequential in nature, where current decisions affect future outcomes. Other ML techniques, usually assume data points to be independent and identically distributed. Moreover, RL differs from other ML methods with its ability to continually improve its performance and adapt to changing circumstances. Based on its mistakes and successes, the agent gradually refines its policy. This unique trait makes RL adaptable to dynamic environments, where adaptability is crucial.

1.3 Value Based, Policy Based and Actor Critic models.

In Reinforcement Learning (RL), there are three main methods for solving problems, which are Value-Based, Policy-Based, and Actor-Critic models. Each of these approaches has its unique characteristics and use cases.

Value-Based Model: These methods, which include Q-learning and Value Iteration, are concerned with calculating the value of each state or action. The desired result is a policy that would give the maximum expected cumulative reward for each state, given any action. But these methods work not by explicitly maintaining a policy, but rather by trying to infer the policy from the value function.

Policy-Based Methods: Unlike Value-Based methods, Policy-Based methods such as Policy Gradient or REINFORCE can do without the presence of the value function, and directly learn the policy function. These techniques change policy parameters in the direction to make the expected return higher. In particular, they have merits over other alternatives, such as high-dimensional or continuous action spaces, or when the policy is naturally expressed as a probability distribution.

Actor-Critic Methods: Methods based on action and critic, such as Advantage Actor-Critic (A2C) or Deep Deterministic Policy Gradient (DDPG), integrate the

advantages of Value-Based and Policy-Based methods. They possess not only a policy (the actor) but also a value function (the critic). The critic evaluates the policy, and the actor updates the policy in the direction suggested by the critic. The purpose of this approach is to minimize the discrepancy within the policy-based methods while utilizing their advantage of working in high-dimensional action spaces.

In summary, while Value-Based methods focus on finding the best value function, the Policy-Based methods directly optimize the policy. Actor-critic methods, on the other hand, leverage both a policy and a value function to optimize the learning process.

2 Background

2.1 Value-Based Methods

Value-based methods are implemented across many industries. From Deep Q-Networks mastering games like AlphaGo to enhancing robotics in navigation and task execution, these methods revolutionize decision-making. In finance, they leverage algorithms for adaptive trading [2], while autonomous vehicles [4], healthcare, and energy management benefit from tailored decision support.

In value-based reinforcement learning, the agent learns to estimate the value function, which offers a measure of how excellent each state or action is. The value function tells the agent how good it is to be in a specific state at a specific moment and under a specific policy. It is an estimate of the expected reward the agent will earn for adhering to a specific policy.

Typically, the value function is represented as a function from the state or state-action space to the intended return. The total discounted reward earned by the agent from the current time-step to the end of the episode is the return. The discounting factor is a number between 0 and 1 that indicates how important future benefits are. If the discounting factor is near zero, the agent will prioritize immediate rewards. If it is near to one, the agent will regard future rewards as equally valuable as immediate rewards.

Given that the agent starts in a specific state and follows a specific policy, the value function calculates the expected return. Policy is a function that connects states to actions. The action chosen by the agent maximizes the value function for the present state. The process is repeated until the episode concludes.

Value-based reinforcement learning has the advantage of working effectively in domains with vast state spaces. Because there are so many alternative states and actions, learning a policy in such domains is difficult. Value-based approaches can manage this by explicitly learning to predict the value function. After learning the value function, the agent may quickly choose the appropriate action to do in each given condition by selecting the action with the highest value.

The Bellman Equation

The Bellman equation is the most significant in value-based reinforcement learning [1]. The equation connects the value function for a current state or action to the value function for states or actions in subsequent time steps. It enables the agent to adjust its value function estimate based on experience. There are two versions of the Bellman equation: one for the state-value function V and one for the action-value function Q. They are as follows:

$$V_t(s) = E_\pi[G_t|S_t = s]$$

is the Bellman equation for the state-value function V.

$V_t(s)$ is the value function for state s at time step t under policy. G_t denotes the return from time step t, and S_t denotes the state at time step t. Given that the agent follows the policy, the expectation extends to all possible future rewards beginning with the present time step.

The action-value function's Bellman equation

$$Q_t(s, a) = E_\pi[G_t|S_t = s, A_t = a]$$

$Q_t(s, a)$ denotes the value function at time-step t for acting a in state s under policy π. A_t is the action performed at time step t. Given that the agent follows the policy π, the expectation extends to all possible future rewards beginning with the present time step.

Q-Learning

A model-free, off-policy algorithm called Q-learning bootstraps the optimal action-value function from the current estimate of the Q-value for the upcoming state or action. The predicted discounted return from a specific action a from a state s and then pursuing the optimal policy is known as the Q-value. The optimal policy selects the action that leads to the highest Q-value for the next state or action.

This algorithm uses the Bellman equation, which is provided below, to update the Q-value:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Here, r is the reward for taking action a in state s, γ is the discount factor, $\max_{a'} Q(s', a')$ is the maximum Q-value for the following state s', and α is the learning rate that defines how much the new estimate replaces the old estimate.

Q-learning learns the optimal Q-value function regardless of the policy being followed and therefore is an off-policy algorithm. The algorithm chooses actions according to the ε-greedy policy, which chooses a random action with probability ε and the optimal action with probability 1-ε.

The fact that Q-learning can take a while to converge is one of its disadvantages. In rare circumstances, it may also be unstable. These problems are addressed by several Q-learning extensions.

2.2 Deep Q-Networks algorithm

Since its release, Deep Q-Networks has demonstrated the ability to achieve better-than-human performance

in numerous Atari games, making it one of the most well-known Reinforcement Learning algorithms to date. It was first described by Mnih et al. in the paper [3] in 2013. DQN is an advancement of the Q-learning algorithm. It aims to find an optimal policy for an agent to take actions in an environment to maximize cumulative rewards.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image 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 D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 1. DQN Psuedo code [5]

In the DQN algorithm, there are 4 main components which are the Q-value function, Loss function, Optimization algorithms and Experience replay.

1. Q-Value function: The Q-value function Q(s, a), represents the expected cumulative reward when taking action a in state s with a certain policy. Neural networks are implemented in DQN to approximate the Q-value for each possible action given the current state. The agent is capable of choosing the action with the highest expected value with the help of the network's predicted Q-values.
2. Loss function: The difference between the predicted Q-values and the target Q-values is calculated by the loss function in the training stage. In the DQN, the loss is calculated using the mean squared error (MSE) between the predicted and the target Q-values. The target Q-value is computed as the sum of the immediate reward and the discounted maximum Q-values for the next state. The MSE metric is particularly well-suited for the DQN framework as it penalizes larger deviations more significantly, ensuring a robust adjustment of the neural network's weights during training.
3. Optimization algorithms: Optimization algorithms are used to update the neural network's weights during training, in the DQN algorithm. Stochastic gradient descent (SGD), Adam and RMSProp are some of the most common optimization algorithms used in DQN. With the use of these optimization algorithms, the DQN manages to minimize its loss and improve Q-value predictions.
4. Experience Replay: Experience Replay consists of storing the agent's experiences (state, action, reward, next state) in a replay buffer. During training, the DQN samples from this buffer to break the temporal correlation between consecutive experiences. Experience Replay has an important role in the DQN algorithm as it stabilizes training and improves learning efficiency by reusing past experiences.

Its performance in the Atari 2600 games is evidence of DQN's adaptability. Each time the neural network processes a frame in a consecutive string of game frames, it generates a Q-value for each action. During game play, the agent selects actions based on the highest predicted Q-value, optimizing cumulative rewards. Such continuous learning from frames enables DQN to capture and process temporal precedence dependencies, reiteratively refining its predictions, and adaptively adjusting its policy. The algorithm's glorious success just shows how it can come out on top in such a variety of uncontrollable and difficult gaming environments.

Limitations are addressed through DQN improvements, such as Double DQN, Dueling DQN and Rainbow DQN. Q-value estimation in double DQN corrects the tendency to overestimate its value. A value-advantage separation is introduced by Dueling DQN, increasing the efficiency of learning value functions and advantage functions. With Rainbow DQN, the author integrates several improvements, including prioritized experience replay and distributional reinforcement learning, making for a complete stabilization, efficiency improvement, and all-around performance enhancement to DQN, which is a dominant force in addressing any number of challenges.

2.3 Policy Based Methods

Unlike value functions that are used to establish the optimal policy, policy-based methods in reinforcement learning require learning a policy directly. These techniques come in handy when there is a wide range of feasible policies or when intractability makes it challenging to define a value function. The main differences between value-based methods and policy-based methods are that policy-based methods do not learn a Q-Value Function and simply learn a policy directly, and policy-based methods can learn stochastic policies unlike value-based methods, which means they may take different actions given the same observation.

One policy-based technique known as policy gradient methods involves learning a policy by calculating the gradient of the expected reward with respect to the policy parameters and then utilizing that information to update the policy in a way that raises the expected reward [6]. Policy-gradient approaches seek to learn a parameterized policy using gradient-based

parameter optimization. In contrast to the value-based approach, a Q-Value Function is not taught. Because of this, it can be applied to situations in which the state-action space is vast or infinite. Policy-gradient methods, like other policy-based methods, can handle continuous actions and learn stochastic policies, something that value-based methods, like Q-Learning, cannot accomplish. Nevertheless, policy-gradient approaches have a problem with high gradient update variance estimates, which can complicate learning.

$$\theta_{t+1} = \theta_t + \alpha \cdot \hat{A}(s,a) \cdot \Delta_\theta \cdot \log \pi_\theta(s|a) \qquad (1)$$

Figure 2. Update equation in Policy-Gradient Methods

The equation shown in 2 is commonly used to update Policy-Gradient methods. In this equation, θ_t stands for the policy parameters at time step t, θ_{t+1} for the policy parameters at time step $t+1$, α is the learning rate, $\hat{A}(s,a)$ for the advantage function at state s and action a, and $\Delta\theta \log \pi(\theta|s,a)$ for the gradient of the log of the policy with regard to the policy parameters at state s and action a. The advantage function $\hat{A}(a|s)$ is a measure of how much better action a is at state s compared to the average of all actions. Usually, it is defined as the difference between the value function estimate at state s and action a and the expected return. The expected return from state s and action a is predicted by the value function estimate.

$$\theta = \theta + \alpha \cdot \gamma^t \cdot G \cdot \Delta \ln \pi_\theta(A_t|S_t) \qquad (2)$$

Figure 3. Update equation for the parameters θ of policy π in REINFORCE.

2 displays the primary update equation for Policy-Gradient Methods. 3 shows the update equation utilized in REINFORCE.

2.4 Actor-Critic Methods

One important set of algorithms within the RL framework is the actor-critic methods that try to combine the strengths of value-based and policy-based approaches. This section gives an insight into Actor-Critic methods and the distinction between stochastic and deterministic policies.

2.4.1 Definition and Components: Actor-Critic methods incorporate two distinct components: the actor and the critic. The actor is the one responsible for changing policies by making decisions on the agent's actions in that environment, while at the same time, the critic evaluates these selected actions by approximating state-action values to give a feedback to an actor. There is a strong interplay between these two elements which makes it more reliable and efficient than other single policy or value-based methods.

2.4.2 Stochastic vs. Deterministic Policies: There is a basic difference between stochastic and deterministic policies with respect to action selection. A policy that is stochastic, indicated by $\pi(a|s)$, gives a state's actions in terms of probability distribution. On the other hand, the deterministic policy, $\mu(s)$ maps states to actions with no randomization. This essential difference greatly affects how the RL agent explores and exploits.

2.4.3 Exploration and Exploitation Trade-off: Reinforcement learning (RL) is a trade-off between exploration, which is about finding the optimal actions, and exploitation, which is about using known optimal actions. Stochastic policies are inherently exploratory as they include randomness in the selection of actions. By so doing, it allows the agent to explore a variety of actions thus enhancing policy robustness. Deterministic policies lack innate randomness and may face challenges in exploring the action space effectively hence leading to sub-optimal policies.

2.4.4 Performance Evaluation Metrics: Actor-Critic methods are evaluated by considering various evaluation metrics. The most important among them include return, advantage function and entropy. Return accounts for the total reward an agent accrues while the advantage function gauges the advantage of a specific action over the average. Entropy, closely tied to stochastic policies, quantifies the uncertainty in the policy's action selection. Consequently, these metrics are directly affected by choosing either stochastic or deterministic policies as that influences RL agent's learning dynamics.

2.4.5 Implementation Details: However, depending on whether stochastic or deterministic policies are used different implementation details of Actor-Critic methods would vary.Stochastic policies are typically characterized by gradient estimation using sampling-based methods, which makes the learning process a little bit uncertain. On the other hand, deterministic policies employ deterministic gradient ascent to update policies. Implementation intricacies like these can have profound effects on the efficiency and robustness of the learning algorithm.

In conclusion, Actor-Critic methods offer a powerful paradigm in reinforcement learning by combining strengths from both value and policy-based approaches. Stochastic and deterministic policies make all the difference in the exploration-exploitation dynamics and learning traits of the agent.

2.5 Two examples of Actor-Critic based algorithms

In this subsection, we will delve into the key components, training updates, and implementation details of

two prominent Actor-Critic algorithms: Proximal Policy Optimization with Generalized Advantage Estimation (PPO-GAE2) as a representative of the Stochastic Policy Approach, and Deep Deterministic Policy Gradients (DDPG) embodying the Deterministic Policy Approach.

2.5.1 Proximal Policy Optimization with Generalized Advantage Estimation (PPO-GAE2).

1. Overview :

PPO-GAE2 combines the strengths of Proximal Policy Optimization (PPO) and Generalized Advantage Estimation (GAE) to achieve a robust and efficient learning framework. PPO introduces a trust region approach to policy updates, ensuring that policy changes remain within a specified range. GAE, on the other hand, improves the estimation of advantages by reducing variance during training.

2. Model Training Update Equations :

Actor Policy Update:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} A_t^{\text{GAE2}}(s_t, a_t) \right]$$

The PPO surrogate objective encourages policy updates that maximize the expected cumulative advantage while staying within a trust region defined by a hyperparameter ϵ.

Critic Value Function Update:

$$L^{\text{Critic}}(\phi) = \frac{1}{2} \hat{\mathbb{E}}_t \left[\left(V_\phi(s_t) - \hat{V}_t^{\text{Target}} \right)^2 \right]$$

The critic is trained using the mean squared error loss between the predicted value and the target value, calculated using Generalized Advantage Estimation.

3. Code Implementation :

```
rho = torch.exp(new_logprobs - logprobs)
surrgt1 = rho * gae
surrgt2 = rho.clamp(1 - epsilon, 1 + epsilon) * gae
policy_loss = -torch.minimum(surrgt1, surrgt2).mean()
```

Figure 4. PPO-GAE2 Actor Policy Update

This code calculates advantages, surrogate policy losses, and updates the actor.

```
critic_loss = F.smooth_l1_loss(state_values, target).mean()
```

Figure 5. PPO-GAE2 Critic Value Function Update

This code calculates the critic loss using smooth L1 loss between predicted state values and target values with mean aggregation.

4. Pros and Cons :

Pros:

- Stable training with the clipped PPO objective.
- Efficient use of samples with GAE, reducing variance.

Cons:

- Sensitive to hyperparameter tuning, especially the clipping parameter ϵ.
- Requires careful balancing of entropy regularization for exploration.

5. Conclusion :

PPO-GAE2 stands out for its ability to balance stability and sample efficiency. The combination of PPO's clipped surrogate objective and GAE's variance reduction makes it particularly effective for handling environments with continuous action spaces.

2.5.2 Deep Deterministic Policy Gradients (DDPG).

1. Overview :

DDPG is designed for continuous action spaces, offering a deterministic policy that simplifies optimization. It employs an Actor-Critic architecture where the actor is responsible for policy approximation, and the critic evaluates the value function.

2. Model Training Update Equations :

Actor Policy Update:

$$\nabla_\theta J \approx \hat{\mathbb{E}} \left[\nabla_\theta Q(s, a|\theta_Q)\big|_{s=s_t, a=\mu(s_t|\theta_\mu)} \nabla_a \mu(s|\theta_\mu)\big|_{s_t} \right]$$

The actor's policy is updated based on the gradient of the critic's output with respect to the action, encouraging actions that maximize the expected cumulative reward.

Critic Value Function Update:

$$L(\theta_Q) = \hat{\mathbb{E}} \left[\left(Q(s, a|\theta_Q) - (r + \gamma Q(s', \mu(s'|\theta_{\mu_{\text{Target}}})|\theta_{Q_{\text{Target}}}) \right)^2 \right]$$

The critic is trained using the temporal difference error between the predicted and target Q-values.

3. Code Implementation :

```
actor_loss = -critic(states, actor(states)).mean()
```

Figure 6. DDPG Actor Policy Update

This code computes the actor loss as the negative mean of the critic's output for given states and corresponding actor's actions.

```
Qvals = critic(states, actions)
with torch.no_grad():
    actions_ = actor_target(next_states)
    Qvals_ = critic_target(next_states, actions_)
    Qvals_[dones] = 0.0
    target = rewards + GAMMA * Qvals_
critic_loss = F.smooth_l1_loss(target, Qvals)
```

Figure 7. DDPG Critic Value Function Update

This code calculates the critic loss using a smooth L1 loss based on the temporal difference (TD) error.

4. Pros and Cons :

Pros:

- Effective in handling continuous action spaces.
- Stability improvement through target networks.
- Straightforward optimization with a deterministic policy.

Cons:

- Prone to overestimation bias in Q-value estimates.
- Sensitive to hyperparameter tuning, especially the choice of the discount factor (γ).
- Exploration-exploitation balance can be challenging, and additional mechanisms like Ornstein-Uhlenbeck noise may be required.

5. Conclusion :
DDPG excels in environments with continuous action spaces, offering a straightforward optimization approach with a deterministic policy. The use of target networks enhances stability during training.

2.5.3 Conclusion.
In conclusion, PPO-GAE2 and DDPG represent two robust Actor-Critic algorithms, each tailored to specific challenges in reinforcement learning. PPO-GAE2 finds its strengths in balancing stability and sample efficiency, while DDPG excels in handling continuous action spaces. The selection between these algorithms depends on the characteristics of the environment and the objectives of the learning task.

3 Methodology

3.1 Experiment 1

3.1.1 Problem Definition.
The objective of Experiment 1 is to employ standard DQN, Double DQN, Noisy DQN, and the combination of Double and Noisy DQN to successfully achieve an average reward of 195 over the last 50 episodes in the "LunarLander-v2" environment. To safely land the spacecraft on the landing pad, the agent in the LunarLander-v2 environment needs to learn how to control the thrust of the lunar lander's engines. The spacecraft's position, velocity, and other pertinent variables are observed by the environment and provided to the agent. The agent's task is to control the spacecraft's descent by issuing thrust commands. If the spacecraft lands on the landing pad without incident, the agent gets rewarded, if it crashes or runs out of fuel, it incurs a penalty.
In the LunarLander-v2 environment, the agent's objective is to figure out a policy that maximizes the total reward it gets over time. To safely land the spacecraft on the landing pad, the agent must develop the ability to make wise decisions regarding how to control the spacecraft's descent. The action space in the LunarLander environment has a length of four, and the observation vector has a length of eight.

3.1.2 Configuration and Hyperparameters.
Standard DQN :

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	128
BUFFER_SIZE	10000
MIN_REPLAY_SIZE	5000
EPS_START	0.9
EPS_END	0.05
EPS_DECAY	0.997
TARGET_UPDATE_FREQ	5
LEARNING_RATE	0.001

Table 1. DQN Hyperparameters

Table 1 shows the parameters for the DQN algorithm. The DNN is composed of 8 inputs, 64 hidden neurons, and 4 outputs. The Activation Function is `tanh`. The optimiser is `Adam`. The Loss Function is `MSELoss`.

Double DQN :

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10000
MIN_REPLAY_SIZE	5000
EPS_START	1.0
EPS_END	0.05
EPS_DECAY	0.997
TARGET_UPDATE_FREQ	5
LEARNING_RATE	0.0005

Table 2. Double DQN Hyperparameters

Table 2 shows the parameters for the Double DQN algorithm. The DNN is composed of 8 inputs, 64 hidden neurons, and 4 outputs. The Activation Function is `tanh`. The optimiser is `Adam`. The Loss Function is `MSELoss`.

Noisy DQN :

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	32
BUFFER_SIZE	10000
MIN_REPLAY_SIZE	5000
TARGET_UPDATE_FREQ	5
LEARNING_RATE	0.0005

Table 3. Noisy DQN Hyperparameters

Table 3 shows the parameters for the Noisy DQN algorithm. The DNN is composed of 8 inputs, 128 hidden neurons (NoisyLinear), 64 hidden neurons (NoisyLinear), and 4 outputs (NoisyLinear). The Activation Function for the first layer is `ReLU`, for the second layer is `tanh`, and for the third layer is not explicitly specified (uses the default linear activation). The optimiser is `RMSprop`. The Loss Function is `MSELoss`.

Combined Double and Noisy DQN :

Hyperparameter	Value
GAMMA	0.98
BATCH_SIZE	64
BUFFER_SIZE	50000
MIN_REPLAY_SIZE	10000
EPS_START	0.5
EPS_END	0.01
EPS_DECAY	0.995
TARGET_UPDATE_FREQ	5
LEARNING_RATE	0.0005

Table 4. Combined Double and Noisy DQN Hyperparameters

Table 4 shows the parameters for the Combined Double and Noisy DQN algorithm. The DNN is composed of 8 inputs, 128 hidden neurons (for Double DQN), 256 hidden neurons (Noisy Linear), 128 hidden neurons (Noisy Linear), and 4 outputs (Noisy Linear). The Activation Function for the Double DQN part is `tanh` for the first layer. For the Noisy DQN part, it's `ReLU` for the first layer, `tanh` for the second layer, and not explicitly specified (uses the default linear activation) for the third layer. The optimiser is `RMSprop`. The Loss Function is `MSELoss`.

Hyperparameters :
`GAMMA` represents the discount factor. `BATCH_SIZE` represents the amount of transitions sampled from the Replay Memory and passed to the DNN. `BUFFER_SIZE` represents the maximum amount of transitions stored by the Replay Memory at any given time. `MIN_REPLAY_SIZE` represents the minimum amount of transitions stored by the Replay Memory at any given time. `EPS_START` represents the starting value of ϵ in the ϵ-Greedy Policy. `EPS_DECAY` is multiplied with ϵ every episode. `EPS_END` is the minimum value ϵ can be. `TARGET_UPDATE_FREQ` is the minimum value that the ϵ can be. `LEARNING_RATE` is the learning rate used for the optimiser.

3.1.3 Measures to validate experiment.
The average reward over the last 50 episodes was calculated at every episode and stored in an array. This is so we can evaluate the model's performance as it reaches the average score of 195.

3.2 Experiment 2

3.2.1 Problem Definition.
The aim of Experiment 2 is to successfully achieve an average reward of 195 over the last 50 episodes in the 'LunarLanderContinuous-v2' environment using two actor-critic algorithms which are PPO-GAE2 and DDPG. In contrast to the discrete LunarLander-v2 environment (Experiment 1), LunarLanderContinuous only has an action space of 2, the left and right thrusters. However since the actions are continuous, they also have the ability to not use any one thruster. The observation vector is an 8-dimensional vector, similar to the discrete LunarLander-v2 environment. Similarly, the agent also receives a reward for successfully landing the spacecraft on the landing pad, and incurs a penalty for crashing or running out of fuel.

3.2.2 Configuration and Hyperparameters.
Stochastic Policy Approach (PPO-GAE2) :

Hyperparameter	Value
nenvs	16
hidden_sizes	128
actor_learning_rate	0.00002
critic_learning_rate	0.001
gamma	0.999
lmbda	0.95
epsilon	0.18
batchsize	64
epoch_repeat	15
memsteps	10000
entropy_coef	0.01

Table 5. Hyperparameters for Stochastic Policy Approach (PPO-GAE2) algorithm

1. **nenvs**: Number of environments. In the context of the Stochastic Policy Approach (PPO-GAE2) algorithm, it specifies the number of parallel environments running concurrently, set to 16.
2. **hidden sizes**: The number of units in the hidden layers of the neural networks. Here, it is set to 128.
3. **actor learning rate**: The learning rate used for updating the parameters of the actor network. It is set to 0.00002.
4. **critic learning rate**: The learning rate used for updating the parameters of the critic network. It is set to 0.001.
5. **gamma**: Represents the discount factor, which determines the importance of future rewards. Set to 0.999.
6. **lmbda**: A parameter in the Generalized Advantage Estimation (GAE) calculation, controlling the trade-off between bias and variance in estimating advantages. Set to 0.95.
7. **epsilon**: A parameter in the Proximal Policy Optimization (PPO) algorithm, representing the

clipping threshold for the ratio of new and old policy probabilities. Set to 0.18.

8. **batchsize**: The size of the mini-batches used for training. It is set to 64.
9. **epoch repeat**: The number of times to repeat the training epochs. In this case, it is set to 15.
10. **memsteps**: The number of steps before performing a policy update. It is set to 10000.
11. **entropy coef**: Coefficient controlling the entropy regularization term in the loss function. Set to 0.01.

These hyperparameters collectively define the configuration of the Stochastic Policy Approach (PPO-GAE2) algorithm and influence its learning behavior in the given environment.

Deterministic Policy Approach (DDPG) :

Hyperparameter	Value
GAMMA	0.99
BATCH_SIZE	64
BUFFER_SIZE	10000
MIN_REPLAY_SIZE	5000
TAU	0.01
actor_learning_rate	0.0003
critic_learning_rate	0.0003

Table 6. Hyperparameters for the DDPG (Deep Deterministic Policy Gradients) algorithm

1. **GAMMA**: Represents the discount factor.
2. **BATCH SIZE**: Represents the amount of transitions sampled from the Replay Memory and passed to the DNN.
3. **BUFFER SIZE**: Represents the maximum amount of transitions stored by the Replay Memory at any given time.
4. **MIN REPLAY SIZE**: Represents the minimum amount of transitions stored by the Replay Memory at any given time.
5. **TAU**: Used for taking a weighted average of the target network and local network policy parameters in order to update the target network parameters.
6. **ACTOR LEARNING RATE**: The learning rate specifically used for updating the actor network in the ADAMW optimizer, set to 0.0003.
7. **CRITIC LEARNING RATE**: The learning rate specifically used for updating the critic network in the ADAMW optimizer, set to 0.0003.

These hyperparameters play a crucial role in defining the behavior of the DDPG algorithm and how it learns from the environment. The description provided captures the essence of each hyperparameter and its role in the algorithm

3.2.3 Measures to validate experiment.

Like Experiment 1, the average reward over the last 50 episodes was calculated at every episode and stored in an array. This is so we can evaluate the model's performance as it reaches the average score of 195.

3.3 Libraries used

The core Python libraries used to perform both experiments were:

- **gymnasium (as gym)**: A toolkit for developing and comparing reinforcement learning algorithms. It provides a variety of pre-built environments for testing and developing RL agents.
- **numpy (as np)**: A library for numerical operations in Python. It is commonly used for handling arrays and mathematical operations.
- **torch**: The PyTorch library, which is widely used for deep learning. It provides tools for building and training neural networks.
- **matplotlib.pyplot (as plt)**: A plotting library for creating visualizations in Python.

4 Results and Discussion

4.1 Experiment 1

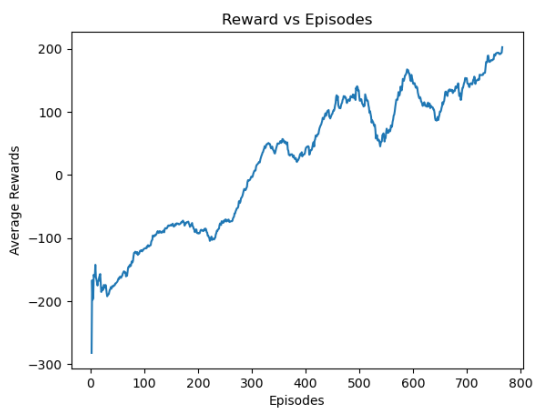

Figure 8. Standard DQN graph

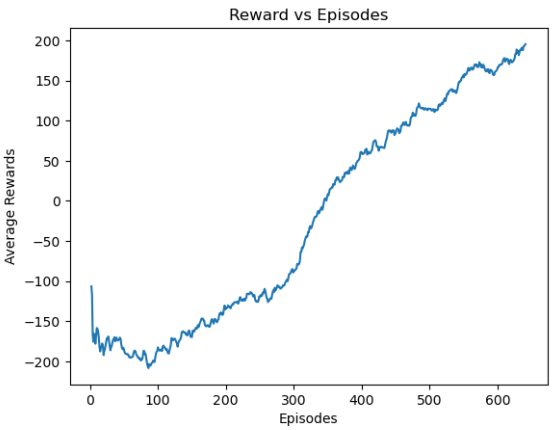

Figure 9. Double DQN graph

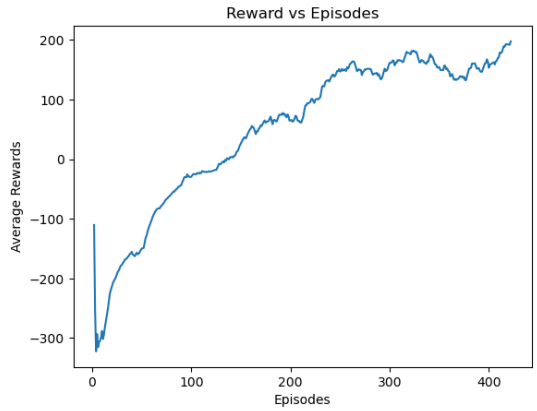

Figure 10. Noisy DQN graph

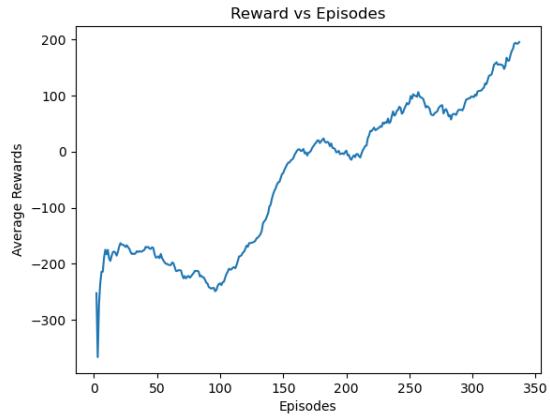

Figure 11. Combined Double and Noisy DQN graph

In the first experiment, it is noticed that Double DQN demonstrated faster convergence, reaching the average reward of 195 at episode 641 (Figure 9), in contrast to the standard DQN, which took until episode 766 (Figure 8). This improved convergence in Double DQN can be credited to its effective handling of overestimation bias. By separating the processes of action selection and evaluation, Double DQN mitigates overoptimistic value estimations, resulting in a more accurate representation of state-action values.

Furthermore, the impressive performance of Noisy DQN, achieved by episode 422 (Figure 10), can be explained by its innovative approach to action selection. Introducing random variations to action values, Noisy DQN encourages exploration, allowing the algorithm to navigate a more extensive state-action space. This deliberate injection of unpredictability safeguards the algorithm from getting stuck in less effective strategies, making it more flexible and responsive to a range of dynamic environments. Ultimately, these elements contribute to its effectiveness in achieving a quicker convergence.

The combined model, bringing together Double DQN and Noisy DQN, demonstrates notable benefits when contrasted with each model on its own. Double DQN shines in tackling overestimation bias, while Noisy DQN enhances exploration through stochastic action selection. At episode 337 (Figure 11), the combined model harmoniously combines these strengths, striking a balance that not only cuts down overestimation bias but also fuels exploration. This blend culminates in more effective convergence and enhanced learning dynamics, outshining the performance of solely relying on Double DQN or Noisy DQN in isolation.

This experiment concludes that the best model for the "LunarLander-v2", was the combined model which converged that the episode 337.

4.2 Experiment 2

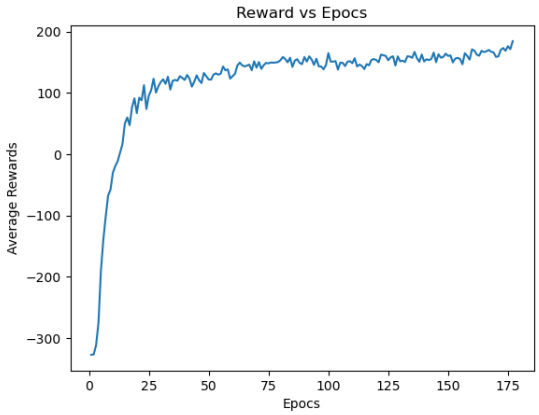

Figure 12. Stochastic Policy Approach (PPO-GAE2) graph

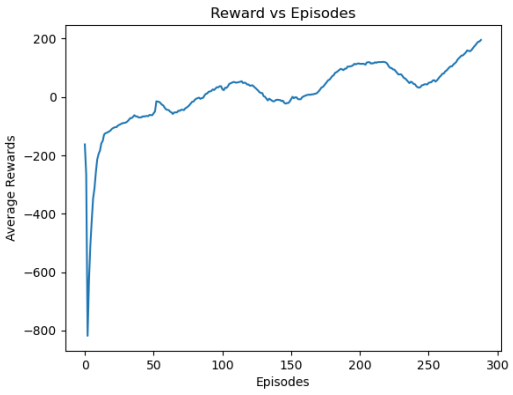

Figure 13. DDPG (Deep Deterministic Policy Gradients) graph

In Experiment Two, the Stochastic Policy Approach (PPO-GAE2) showcases remarkable performance, achieving an average reward of 195 at epoch 177 (Figure 12). This success can be attributed to its utilization of a stochastic policy, which promotes exploration and adaptability in the face of uncertainties. On the other hand,

the Deterministic Policy Approach (DDPG) reaches the same benchmark at epoch 288 (Figure 13). The deterministic nature of DDPG, while providing stability, might hinder exploration in complex environments. Comparatively, PPO-GAE2's stochasticity proves advantageous in challenging scenarios, enabling faster and more efficient learning dynamics.

References

[1] Hardik Dave. 2021. Understanding the Bellman Optimality Equation in Reinforcement Learning. *Analytics Vidhya* (2021). https://www.analyticsvidhya.com/blog/2021/02/understanding-the-bellman-optimality-equation-in-reinforcement-learning/

[2] Bao F. Kong Y. Ren Z. Dai Q. Deng, Y. 2017. Deep direct reinforcement learning for financial signal representation and trading. *IEEE transactions on neural networks and learning systems* 28, 3 (2017), 653–664. https://doi.org/10.1109/TNNLS.2016.2522401

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. *CoRR* abs/1312.5602 (2013). arXiv:1312.5602 http://arxiv.org/abs/1312.5602

[4] Cheng C. Saigol K. Lee K. Yan X. Theodorou E. Boots B. Pan, Y. 2017. Agile autonomous driving using end-to-end deep imitation learning. In *Robotics: Science and Systems*. https://arxiv.org/abs/1709.07174

[5] Unnat Singh. 2019. Deep Q-Network with Pytorch. https://towardsdatascience.com/deep-q-network-with-pytorch-146bfa939dfe.

[6] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 1999. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, Vol. 12.