Solving Control Problems using Reinforcement Learning

Gianluca Aquilina
348904L
gianluca.aquilina.22@um.edu.mt

ABSTRACT
Reinforcement learning is a powerful framework that shows how an intelligent agent should take actions in an environment to maximise the reward signal. This project applies the Deep Q-Networks (DQN) and Deep Deterministic Policy Gradient (DDPG) to solve the *LunarLander-v3* and *LunarLanderContinuous-v3* environments, respectively. For DQN, we tuned 2 hyperparameters: learning rate and hidden layer's size, and chose the Double DQN as the DQN improvement. After injecting some noise in both DQN models, we noticed that, as noise was increasing, the Standard DQN was more robust, while the Double DQN struggled with varied performances and degradation. For DDPG, we optimised the Tau value and the batch size. After analysing the niose effects on the exploration side, DDPG shows its resilience under noise but has increased learning variance. Overall, these findings provide insights into RL algorithm robustness for real-world applications.

INTRODUCTION
Reinforcement learning is a type of machine learning algorithm whose main objective is to optimise the agent to pick actions to maximise the total future reward. The goal is to learn a policy that maps states to actions to maximise the expected return [1].

Reinforcement learning is a subset of machine learning, and the definition of reinforcement learning differs from other machine learning methods, such as supervised learning. For instance, in the study of [2], it explains the difference between supervised learning and reinforcement learning. One of the differences mentioned is that in supervised learning, the system is provided with input and output pairs and learns to map inputs to outputs. Whilst for RL, it doesn't have such pairs but, instead, it teaches the agent by interacting with the environment, receiving rewards and new states as feedback.

Reinforcement Learning has 3 types of models that it uses:

1. Value Based Models: This model focuses on estimating the value function, which predicts the future reward from a given state. It doesn't explicitly maintain a policy, as the policy is implicit in the value function. An example of a Value based algorithm is Q-Learning [1]
2. Policy Based Models: This algorithm directly learns the policy that maps the states to actions without using a value function. It is useful for environments with high-dimensional or continuous action spaces. An application of this algorithm is the REINFORCE-algorithm [1]
3. Actor Critic Models: This uses both the value-based and policy-based approaches as the actor updates the policy based on the feedback from the critic (value function), which evaluates the actions taken by the actor. An example of this algorithm is the Advantage Actor Critic Methods (A2C) [1]

BACKGROUND
In this chapter, I will explain more in detail the 3 methods mentioned in the Introduction section, and specifically the Deep Q-Networks (DQN) and Deep Deterministic Policy Gradient (DDPG).

Value Based Methods
As mentioned in the previous section, the value based methods in reinforcement learning use value functions to evaluate and improve policies. The following methods are all cases of value based methods:

1. Dynamic Programming: This requires full knowledge of the Markow Decision Process (MDP) and operates in 2 main steps:
   (a) Prediction: This is the policy evaluation stage where it computes the value function for a given policy using the Bellman Expectation Equation from [3] by updating the expected rewards and future values of next states [3]
   $$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) \left[r + \gamma V(s')\right]$$
   (b) Control: This is the policy improvement stage where it uses the previous value function to improve the policy by choosing actions that maximise expected rewards using the equation from [3]
   $$V(s) \leftarrow \max_a \sum_{s',r} p(s',r \mid s,a) \left[r + \gamma V(s')\right]$$
2. Monte Carlo Methods: This method doesn't require prior knowledge of either MDP transitions or rewards but instead learns directly from full episodes of experience. The Monte

Carlo method, based on sampled returns from episodes, estimates and predicts value functions using the formula below in [3]

$$V(s) = \frac{1}{N(s)} \sum_{i=1}^{N(s)} G_i$$

In the prediction phase, there are 2 types of MC methods: First-Visit MC, where it updates *V(s)* using only the first occurrence of s in an episode, whilst Every-Visit MC updates *V(s)* using all occurences of s in an episode [3].

In the control phase, to optimise the policy, there are 2 types of MC methods: On-Policy MC, or ε-greedy method, where it generates episodes following the current policy while making sure of exploration, and Off-Policy MC, or Importance Sampling method, where it learns from episodes generated by another policy [3].

3. Temporal Difference Learning: It combines Monte Carlo sampling with Dynamic Programming updates, which allows agents to learn online. In the prediction phase, the TD learning updates value functions cumulatively using the TD error to adjust the *V(s)* before the episode ends [3].

$$V(s) = V(s) + \alpha \left[R + \gamma V(s') - V(s)\right]$$

In the control phase, to optimize the TD-based policy, there are 2 types according to [3]

(a) SARSA, which is an On-Policy TD Control, updates using the current policy's action selection

$$Q(s,a) = Q(s,a) + \alpha \left[R + \gamma Q(s',a') - Q(s,a)\right]$$

(b) Q-Learning, which is an Off-Policy TD Control, uses the greedy action

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

4. Deep Q-Networks (DQN): DQN uses deep neural networks with reinforcement learning to, using high-dimensional inputs, allow the agents to learn the optimal policies. DQN approximates the action-value function using a neural network, which enables effective decision-making in complex environments [4]. This algorithm utilises some key techniques to achieve stable learning.

Firstly, the experience replay is used so that DQN stores agent experiences in a replay buffer to avoid continuous updates and samples experiences randomly during training instead of updating the model sequentially.

```
import torch
import numpy as np

states = torch.from_numpy(np.array(batch.
    states, dtype=np.float32))
actions = torch.from_numpy(np.array(batch.
    actions, dtype=np.int64))
rewards = torch.from_numpy(np.array(batch.
    rewards, dtype=np.float32))
next_states = torch.from_numpy(np.array(
    batch.next_states, dtype=np.float32))
```

Listing 1. Storage of agent experiences in the replay memory

The loss function used for training is found in the study of [4] below:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i) \right)^2 \right] \quad (1)$$

θ_i represents the current network parameters and θ_i^- represents the target network parameters, which are updated periodically [4]. The aim of this is to reduce correlations between predicted and target Q-values and to achieve stable learning by updating the Q-values periodically instead of every step

```
# Current Q-values
qvalues = dqn_policy(b_states).gather(1,
    b_actions)

# Target Q-values
with torch.no_grad():
    target_qvalues = dqn_target(
        b_next_states)
    max_target_qvalues = torch.max(
        target_qvalues, axis=1).values.
        unsqueeze(1)
    expected_qvalues = b_rewards + GAMMA *
        (1 - b_dones.type(torch.int64)) *
        max_target_qvalues

# MSE Loss
loss = loss_fn(qvalues, expected_qvalues)
```

Listing 2. The loss function for training in code

After the loss is computed, gradient updates adjust the netowrk's weights to minimize the error using the code below

```
# Initialize optimizer with learning rate
optimizer = torch.optim.Adam(dqn_policy.
    parameters(), lr=learning_rate)

# Gradient update steps:
optimizer.zero_grad()
loss.backward()

# Gradient clipping
for param in dqn_policy.parameters():
    param.grad.data.clamp_(-1, 1)

# Update network parameters
optimizer.step()
```

Listing 3. The gradient updates in code

DQN uses a target network, as mentioned above, and there are 2 types of target network updates. The soft target network update gradually blends target network weights with primary network weights, while hard target network updates is when the target network is entirely replaced with the primary network every step

```
# Hard update
if episode % TARGET_UPDATE_FREQ == 0:
    dqn_target.load_state_dict(dqn_policy.
        state_dict())
```

Listing 4. The hard target network update in code

Moreover, DQN makes use of the epsilon-greedy strategy, which starts with high exploration and gradually reduces the exploration over time as the agent learns, which ensures the agent doesn't get stuck in suboptimal policies

```
EPS_START = 0.9     # Starting value of
    epsilon
EPS_END = 0.05      # Minimum value of
    epsilon
EPS_DECAY = 0.997   # Decay rate for epsilon

def epsilon_greedy_policy(epsilon, obs,
    dqn_policy, env):
    rnd_sample = random.random()
    if rnd_sample <= epsilon:  #
        Exploration
        action = env.action_space.sample()
    else:  # Exploitation
        with torch.no_grad():
            action = int(torch.argmax(
                dqn_policy(torch.Tensor(obs
                ))))
    return action

# In training loop:
eps_threshold = EPS_START

if done:
    episode += 1
    # Decay epsilon
    eps_threshold = np.max((eps_threshold *
        EPS_DECAY, EPS_END))
```

Listing 5. The epsilon-greedy algorithm in code

(a) Double DQN (DDQN): This algorithm is an enhancement of the Standard DQN and the crucial improvement over the standard DQN is how target Q-values are computed, as, unlike Q-Learning and DQN, it introduces 2 separate networks for selecting actions and evaluating actions to ensure more accurate value estimates and improved policy learning. The target Q-value update in Double DQN is shown in the study of [5] below:

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \theta_t); \theta_t')$$

In this update, it first use the online network θ to select the best action a in the next state by having the highest Q-value, then the target network θ^- evaluates the Q-value of the selected action to prevent overestimating its own predictions and lastly, the Bellman equation is used to compute the final target Q-value

```
with torch.no_grad():
    _, next_actions = dqn_policy(
        b_next_states).max(dim=1,
        keepdim=True)    # Selecting
        action using policy network
    next_action_values = dqn_target(
        b_next_states).gather(1,
        next_actions)   # Evaluating
        using target network
    expected_qvalues = b_rewards +
        GAMMA * (1 - b_dones.type(torch
        .int64)) * next_action_values #
         Using Bellman Equation
```

Listing 6. The target Q-value update implementation in code

Policy Based Methods

Policy-based methods directly learn a policy that maps states to actions. The main differences between value-based methods and policy-based methods are that value-based methods estimate a value function and get a policy from it; they are typically deterministic and require explicit exploration strategies, while policy-based methods optimise the policy directly using gradient descent, can handle stochastic and continuous actions, and include inherent exploration [6].

The policy gradient update equation updates the policy parameters proportional to the gradient of the expected reward with respect to the policy parameters [6].

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right]$$

1. REINFORCE Algorithm: A Monte-Carlo Policy-Gradient Control, which estimates policy gradients using Monte Carlo methods

$$\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s,a) R$$

Actor-Critic Methods

Actor-Critic methods are a type of reinforcement learning algorithm that combines both policy-based and value-based methods. These methods use 2 function approximators: actor, a policy function parameterised by theta: $\pi_\theta(s,a)$, and critic, a value function parameterised by w: $\hat{q}_w(s,a)$. In short, the process starts with the actor with a random policy and the critic with a random value function, and at each timestep, the current state is passed as input through the actor to select an action. The selected action and the current state are passed through the Critic to compute the Q-value, and the action is performed in the environment, resulting in a new state and a reward. Finally, the actor updates its policy parameters using the q-value from the critic, and the critic updates its value parameters based on the new state and reward [7]

The differences between the stochastic and the deterministic policy approaches are that the stochastic policy outputs a probability distribution over actions, and this approach is useful in environments with high uncertainty and where exploration is crucial, while the deterministic approach outputs a specific action directly without any randomness, and this approach is useful in environments where the optimal action is more predictable and less variable [7].

Deep Deterministic Policy Gradient

DDPG is a model-free and off-policy algorithm designed for environments with continuous spaces. It combines the strengths of value-based and policy-based methods, following an actor-critic approach [8]. The following is an overview of its main structure and update mechanism:

1. Actor Network: It is responsible for learning the policy, which maps states to specific actions and outputs a deterministic action given a state [8]

```
class PolicyNet(nn.Module):
```

```
    def __init__(self, input_size,
        hidden_units, output_size, pmin,
        pmax):
        super(PolicyNet, self).__init__()
        self.pmin = pmin
        self.pmax = pmax
        self.model = nn.Sequential(
            nn.Linear(input_size,
                hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, int(
                hidden_units/2)),
            nn.ReLU(),
            nn.Linear(int(hidden_units/2),
                output_size),
            nn.Tanh()
        )
```

Listing 7. The actor network in code

2. Critic Network: It evaluates the action taken by the actor by estimating the Q-value [8]

```
class DQN(nn.Module):
    def __init__(self, input_size,
        hidden_units):
        super(DQN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size,
                hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, int(
                hidden_units/2)),
            nn.ReLU(),
            nn.Linear(int(hidden_units/2),
                1)
        )
```

Listing 8. The critic network in code

3. Target Networks: DDPG uses target networks for both the actor and critic and are copies of the original network that are slowly updated to stabilise training [8]

```
# Creation
actor_target = copy.deepcopy(actor)
critic_target = copy.deepcopy(critic)

# Update
for target_param, param in zip(actor_target
    .parameters(), actor.parameters()):
    target_param.data.copy_(param.data *
        tau + target_param.data * (1.0 -
        tau))

for target_param, param in zip(
    critic_target.parameters(), critic.
    parameters()):
    target_param.data.copy_(param.data *
        tau + target_param.data * (1.0 -
        tau))
```

Listing 9. The target networks in code

4. Replay Buffer: Similar to DQN, it stores the experiences in the replay memory, and the mini-batches of experiences are randomly sampled to improve learning stability [8]

```
class ReplayMemory():
    def __init__(self, env, capacity,
        batch_size):
        self.env = env
        self.memory = deque(maxlen=capacity
            )
        self.rewards = deque(maxlen=50)
        self.batch_size = batch_size

    def sample_batch(self):
        batch = random.sample(self.memory,
            self.batch_size)
        batch = Transition(*zip(*batch))
        states = torch.from_numpy(np.array(
            batch.states, dtype=np.float32)
            )
        actions = torch.from_numpy(np.array
            (batch.actions, dtype=np.
            float32))
        rewards = torch.from_numpy(np.array
            (batch.rewards, dtype=np.
            float32)).unsqueeze(1)
        dones = torch.from_numpy(np.array(
            batch.dones, dtype=np.bool8)).
            unsqueeze(1)
        next_states = torch.from_numpy(np.
            array(batch.next_states, dtype=
            np.float32))
        return states, actions, rewards,
            dones, next_states
```

Listing 10. The experience memory in code

The update mechanism is as follows:

1. Critic Update: The critic network is updated by minimising the MSBE (Mean-squared Bellman error). The target Q-value is computed using the target networks ($y = r + \gamma Q_{\text{target}}(s', \mu_{\text{target}}(s'))$) and the critic network parameters are updtaed by minimizing the loss: $L = \frac{1}{N}\sum(Q(s,a) - y)^2$ [8]

```
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)
```

Listing 11. The critic update in code

2. Actor Update: The actor network is updated using the policy gradient. The gradient is used to update the actor network parameters [8]

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

Listing 12. The actor update in code

3. Target Network Update: The target networks are updated using soft updates to slowly track the learned networks: $\theta_{\text{target}} \leftarrow \tau\theta + (1-\tau)\theta_{\text{target}}$ [8]

```
for target_param, param in zip(actor_target
    .parameters(), actor.parameters()):
        target_param.data.copy_(param.data
            * tau + target_param.data *
            (1.0 - tau))
```

```
for target_param, param in zip(
    critic_target.parameters(), critic.
    parameters()):
        target_param.data.copy_(param.data
            * tau + target_param.data *
            (1.0 - tau))
```

Listing 13. The soft target network updates in code

Comparison to DQN

Below are the main differences between DDPG and DQN:

1. DQN is designed for discrete actions, while DDPG handle continuous actions [9]
2. DQN uses a value-based approach by selecting actions based on the Q-values, but DDPG uses a policy-based approach with an actor network to directly output actions [9]
3. DDPG consists of 2 networks: an actor and critic network, whilst DQN consists of one network that outputs the Q-values for all actions [9]
4. DDPG has 2 target networks: a target actor and a target critic network, whilst DQN has one target Q-network [9]
5. DDPG uses Ornstein-Uhlenbeck process to add noise to the action for exploration; however, DQN uses the ϵ-greedy strategy, where a random action is chosen instead of the action with the highest value [9]

METHODOLOGY

Experiment 1: LunarLander-v3

The first experiment was to solve the LunarLander-v3 from the Gymnasium library. The objective was that the spacecraft must land safely in a designated landing pad. It has 8-dimensional state space and 4 discrete actions: do nothing, fire left engine, fire main engine, and fire right engine. The rewards for this environment are +100 for landing and -100 for crash, fuel usage penalties, and step rewards for approaching the landing pad.

The algorithms of standard DQN and double DQN improvement were used for this experiment. The network architecture chosen was a simple 2-layer neural network with *tanh* activation function used for better gradient flow. The input and output layers match the state space and discrete actions space, respectively.

```
class DQN(nn.Module):
    def __init__(self, ninputs, noutputs,
        hidden_size):
        super(DQN, self).__init__()
        self.a1 = nn.Linear(ninputs,
            hidden_size)
        self.a2 = nn.Linear(hidden_size,
            noutputs)

    def forward(self, X):
        o = self.a1(X)
        o = torch.tanh(o)
        o = self.a2(o)
        return o
```

Listing 14. The network architecture in code

In the hyperparameter tuning phase, I decided to use the hidden layer size (64, 128, 256) and learning rate (0.0005, 0.001, 0.01). After the tuning process, which involved in training models with different combinations of these hyperparameters and selecting the one combination that took the least number of episodes to reach the average score of 195 over the last 50 episodes using grid search, the combination was hidden size layers set to 128 neurons and setting learning rate as 0.001 as it took 598 episodes to be considered as solved.

```
# Grid search implementation
for lr in learning_rates:
    for hidden_size in hidden_sizes:
        results.append({
            'learning_rate': lr,
            'hidden_size': hidden_size,
            'episodes_to_solve':
                solved_episode,
            'avg_reward': final_avg_reward
        })
```

Listing 15. Tuning Process in code

The performance of the DQN model was evaluated using the criteria to solve the experiment (an average reward of 195 over the last 50 episodes) and the noise robustness tests to analyze the model's stability under varying levels of noise.

```
if len(replay_memory.rewards) == 50 and
    avg_res >= 195:
                print(f'Solved␣at␣episode:␣{
                    episode}␣Avg␣Results:␣{
                    avg_res:.2f}')
                break
```

Listing 16. The criteria to solve the experiment in code

```
NOISE_LEVELS = {
    'LOW': 0.05,
    'MEDIUM': 0.15,
    'HIGH': 0.25
}
```

Listing 17. Different Gaussian noise levels in code

Below are all the libraries used to analyse Experiment 1:

```
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
from typing import Sequence
from collections import namedtuple, deque
import itertools
import random
import warnings
warnings.filterwarnings("ignore")
```

Listing 18. Libraries used in code

Experiment 2: LunarLanderContinuous-v3

The LunarLanderContinuous-v3 environment is a continuous action problem where the agent must control a lunar lander and land it safely on a landing pad. The state space consists of 8 variables. Unlike the discrete version of experiment 1, the agent directly controls the thrust values for the main

engine and side engines, which both have a continuous thrust value between -1 and 1. The rewards are split exactly like LunarLander-v3 Discrete version in the first experiment. The objective of this experiment is to learn a continuous control policy that maximises the cumulative reward.

The algorithm chosen was the DDPG algorithm, which included the actor-critic method that learns both a policy function and a value function (explained in the Background Section). The network architecture was split into 2:

1. Actor Network: A deep, fully connected feedforward neural network that outputs continuous control values. The input layers had the same amount of neurons as the amount of state space variables, and the 2 hidden layers have 128 neurons and using the ReLU activation function. The output layer has 2 neurons using the Tanh activation to keep the output between -1 and 1 due to the thrust values.

```
class PolicyNet(nn.Module):
    def __init__(self, input_size,
        hidden_units, output_size, pmin,
        pmax):
        super(PolicyNet, self).__init__()
        self.pmin = pmin
        self.pmax = pmax
        self.model = nn.Sequential(
            nn.Linear(input_size,
                hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, int(
                hidden_units/2)),
            nn.ReLU(),
            nn.Linear(int(hidden_units/2),
                output_size),
            nn.Tanh()
        )

    def forward(self, x):
        x = self.model(x) * self.pmax
        torch.clip_(x, self.pmin, self.pmax
            )
        return x
```

Listing 19. Actor Network in code

2. Critic Network: This network evaluates the state-action pairs and estimates expected future rewards. The input layer had the same number as the combination of state and action variables (8 + 2) and has 2 hidden layers with 128 neurons and the ReLU activation function being used. The output layer contains 1 neuron, which is the Q-value estimation

```
class DQN(nn.Module):
    def __init__(self, input_size,
        hidden_units):
        super(DQN, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size,
                hidden_units),
            nn.ReLU(),
            nn.Linear(hidden_units, int(
                hidden_units/2)),
            nn.ReLU(),
            nn.Linear(int(hidden_units/2),
                1)
        )

    def forward(self, state, action):
        x = torch.cat([state, action], 1)
            # Concatenating state and
            action
        x = self.model(x)
        return x
```

Listing 20. Critic Network in code

For the hyperparameter tuning phase, the 2 crucial hyperparameters tuned were the soft update rate (0.001, 0.005, 0.01) and batch size (32, 64, 128) and the best combination was found identically, like in the discrete problem, where the soft update rate was set to 0.005 and the batch size was set to 128.

The performance of the DDPG model was evaluated using the criteria needed to solve the experiment and the noise robustness tests to analyse the model performance under different levels of Ornstein-Uhlenback noise.

Below are all the libraries used for this experiment:

```
import gymnasium as gym
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import optim
import matplotlib.pyplot as plt
from typing import Sequence
from collections import namedtuple, deque
import itertools
import random
import copy
from itertools import count
import warnings
warnings.filterwarnings("ignore")
```

Listing 21. Libraries used in code

RESULTS AND DISCUSSION

Experiment 1: LunarLander-v3

After adding the noise to both standard DQN and double DQN models, below is the impact of noise on performance:

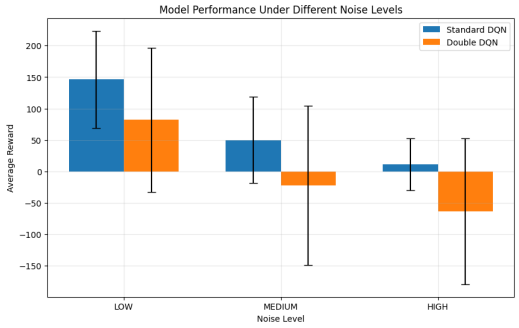

Figure 1. Noise Impaction Results

Some observations are that in low noise, the standard DQN performs better than the double DQN, as it achieves a higher mean reward and a more consistent learning curve, and for medium and low noise, both drastically drop their performance; however, the double DQN struggled more, as, for instance, in medium noise, the double DQN had negative rewards, while the standard DQN had slightly positive rewards.

```
Testing with LOW noise (σ=0.05)
Standard DQN - Mean reward: 146.10 ± 77.48
Double DQN - Mean reward: 81.80 ± 114.83

Testing with MEDIUM noise (σ=0.15)
Standard DQN - Mean reward: 49.70 ± 68.59
Double DQN - Mean reward: -22.45 ± 126.69

Testing with HIGH noise (σ=0.25)
Standard DQN - Mean reward: 11.06 ± 41.21
Double DQN - Mean reward: -63.32 ± 116.09
```

Figure 2. Average Rewards with different noise

From these observations, one can notice that Standard DQN is more resistant to noise, while Double DQN manages poorly when noise increases, which leads to more unstable learning. As well, Double DQN had a greater collapse than Standard DQN as noise increased.

Experiment 2: LunarLanderContinuous-v3

After adding noise, we tested the DDPG's robustness using learning curves and episodes required to solve the environment under different noise levels

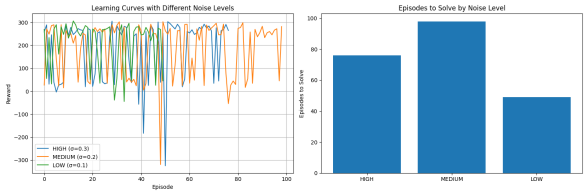

Figure 3. Noise Impaction Results

```
Testing HIGH noise level (sigma=0.3)...
Episode: 10  Return: 243.20  Average Return: 135.61
Episode: 20  Return: 21.08  Average Return: 174.28
Episode: 30  Return: 286.13  Average Return: 168.12
Episode: 40  Return: 245.06  Average Return: 173.92
Episode: 50  Return: -324.75  Average Return: 156.77
Episode: 60  Return: 253.69  Average Return: 178.22
Episode: 70  Return: 34.47  Average Return: 184.18
Solved in 76 episodes! Average Return: 198.38

Testing MEDIUM noise level (sigma=0.2)...
Episode: 10  Return: 240.52  Average Return: 190.41
Episode: 20  Return: 253.68  Average Return: 183.17
Episode: 30  Return: 286.98  Average Return: 201.79
Episode: 40  Return: 263.32  Average Return: 186.65
Episode: 50  Return: 263.21  Average Return: 183.04
Episode: 60  Return: 34.97  Average Return: 178.19
Episode: 70  Return: 285.02  Average Return: 185.28
Episode: 80  Return: 273.62  Average Return: 166.68
Episode: 90  Return: 282.73  Average Return: 178.42
Solved in 98 episodes! Average Return: 196.38

Testing LOW noise level (sigma=0.1)...
Episode: 10  Return: 231.25  Average Return: 214.95
Episode: 20  Return: 271.14  Average Return: 231.56
Episode: 30  Return: 55.91  Average Return: 212.31
Episode: 40  Return: 244.93  Average Return: 213.79
Solved in 49 episodes! Average Return: 212.81
```

Figure 4. Average Returns with different noise

From these readings, one can notice that low noise converged the fastest in terms of episodes, with stable learning curves. Learning starts to slow down in the medium and high noise setting, but surprisingly, the high noise setting has faster convergence than medium noise

The key takeaways from here are that DDPG is highly resistant to noise, as it maintains a very strong average return across all levels. However, noise does affect the learning speed, with the medium noise causing the most instability.

Challenges Encountered for both Experiments

One challenge I faced was the hyperparameter tuning, as it really took a long period of time to train all the combinations of the parameters, so very time-consuming.

CONCLUSION

In this study, we explored the value-based methods and the policy-based methods, which were explained and tested in the 2 experiments. As a quick overview of the outcomes, for discrete action spaces, Standard DQN is more reliable than Double DQN, especially in noisy settings. For continuous action spaces, DDPG proves to be very robust to the *LunarLanderContinuous-v3* environment, but it requires attentive noise adjustments to ensure that the learning remains stable. Other important steps were the hyperparameter tuning, which allows us to optimise and improve the values to get the least number of episodes to solve the experiments. Future work can be done to attempt to improve this project, such as using deling DQN, prioritising experience replay, or trying out TD3 for performance gain. This project has provided me valuable insights into choice of reinforcement Learning models, different methods to get the best average reward, and adding noise to test its robustness.

References

[1] Dr. Vince Vella. "Lecture Notes on Introduction". Distributed in ARI 3212, Department of Artificial Intelligence, University of Malta. Accessed: 2024-09-29. 2024.

[2] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. "Reinforcement learning: A survey". In: *Journal of artificial intelligence research* 4 (1996), pp. 237–285.

[3] Dr. Vince Vella. "Lecture Notes on Basic RL Methods". Distributed in ARI 3212, Department of Artificial Intelligence, University of Malta. Accessed: 2024-09-29. 2024.

[4] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[5] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[6] Richard S Sutton et al. "Policy gradient methods for reinforcement learning with function approximation". In: *Advances in neural information processing systems* 12 (1999).

[7] Dr. Vince Vella. "Lecture Notes on Actor Critic Methods". Distributed in ARI 3212, Department of Artificial Intelligence, University of Malta. Accessed: 2024-09-29. 2024.

[8] Ebrahim Hamid Sumiea et al. “Deep deterministic policy gradient algorithm: A systematic review”. In: *Heliyon* (2024).

[9] Dr. Vince Vella. “Lecture Notes on More SOTA models - PPO GAE, DDPG, TD3, SAC”. Distributed in ARI 3212, Department of Artificial Intelligence, University of Malta. Accessed: 2024-09-29. 2024.