uc3m | Universidad Carlos III de Madrid

MASTER IN CONNECTED INDUSTRY 4.0

GRADUATE SCHOOL OF ENGİNEERİNG AND BASİC SCİENCES

MASTER THESIS

MANEUVERS TO AVOID NEARBY UAVS THROUGH REINFORCEMENT LEARNING

Author: Alper Soysal

Advisor: Prof. Luis Moreno Lorente Enrique

PUERTA DE TOLEDO, MADRID

JUNE - 2021

Contents

Index of figures

Index of tables

Acknowledgements

I would first like to thank my thesis advisor, Prof. Luis Moreno Lorente Enrique, to encourage me to complete the thesis project successfully. He offered great help in my work to learn how to discover Reinforcement Learning. Without his passionate input, the thesis project could not have been successfully conducted. Thanks for his suggestions and valuable feedback on my research as a great researcher, instructor, and role model.

My appreciation also extends to my renowned professors and academics, who enabled me to have an exciting learning experience. I would like to acknowledge my thesis committee members as contributors with all of their invaluable guidance. I am grateful to present my thanks to all graduate students who study in the same program.

I would especially like to thank my undergraduate advisors, Prof. Serdar İplikçi from Pamukkale University, who contributed my academic background to gain success in this research.

Also, I would like to give special thanks to Beste Şenocak,Öykü Hazal Aral, José Antonio Navarro, Sergio Gutiérrez Hernández, Pablo Aguado Oria. Their valuable support motivates me to continue working on these thesis.

I must express my gratitude to my fantastic family and close friends for their love, support, and motivation throughout my life. I undoubtedly could not have done this accomplishment without you. Thank you.

Abstract

Reinforcement Learning (RL) is a general-purpose framework for developing nonlinear controllers. It is based on a trial-and-error methodology. RL has traditionally been used in an application with low dimensions state space. However, the implementation of Deep Neural Networks as function approximators with RL has shown excellent results for high-dimensional systems control.

This thesis aims to implement a Deep Reinforcement Algorithm (DRL) on a 3D UAV collision avoidance and path planning problem. Maneuvers to avoid to nearby UAVS task formulated as an observable Markov Decision Process, and it is solved using a Deep Reinforcement Learning method. The major challenge on this application: 1) insufficient information of the environment. 2) complex motion kinematic of the UAV. Memory-based Deep Reinforcement Learning is used to tackle the first challenges. A policy is represented as a network with a memory unit that can recall previous observations in this framework. The second difficulty was solved using a simplified kinematic of the UAV. The action applied to the UAV using only one plane. In addition, the Fast-Marching Square planning approach is chosen, which provides a level of path quality and robustness that only a few methods can achieve.

In the simulation, the method is evaluated with the trained policy on a different scenario. Trained policy generates collision-free routes to the goal point and achieves the task sufficiently.

Chapter 1

Introduction

An unmanned aerial vehicle (UAV), commonly known as a drone, is an aircraft without a human pilot onboard and a type of unmanned vehicle [1]. Unmanned Aerial Vehicle has an enormous potential to aid human beings in the near future and received tremendous attention in various areas such as military services, commercial, Industry, and so on. There are applications in the army field, such as monitoring, target tracking; amongst these, one of the most significant issues is collisions avoidance, especially in an unsafe environment [2,3].

I would like to address the subject of Maneuvers to Avoid Nearby UAVs using the approach known as Reinforcement Learning (RL) as part of my research. I would like to use Deep Neural Networks (DNN) as a Function Approximator. DNN has shown remarkable results for the control of high-dimensional systems when used with RL [4].

The chapter will give brief information about UAV collision avoidance. The chapter will further discuss the issues with the implementation of DRL and the methodology built in the thesis to address this collision avoidance problem.

1.1 Collision avoidance for Unmanned Aerial Vehicle (UAV)

Collision avoidance for UAV is one of the most challenging control problems in recent years. To avoid a collision, it is essential to use an intelligent calculation method. The following task must be completed:

- The acquisition of potentially hazardous objects, both dynamically moving and stable
- Implementation of a collision avoidance algorithm

For the first task, information about trajectory is required. Three coordinates (x, y, z) are essential for defining the UAV's trajectory [5].

Deep Reinforcement Learning (DRL) popularity is increased during years. In 2016 , the combination of the DRL and Reinforcement Learning achieved remarkable results in two success stories. First, the DeepMind team created a single RL agent capable of playing numerous Atari 2600 video games on a human level[4].The set of actions is chosen based on the game's raw input images, while the score of the game represents the reward. AlphaGo [6] was the second success story. The team created a hybrid DRL agent capable of defeating the world champion in the Chinese board game Go. Thanks to these achievements, DRL is a suitable method to applied collision avoidance problems because it enables the development of a control policy based on raw input data.

A variety of approaches for UAV collision avoidance have been developed in previous literature. In [7], monocular RGB pictures are used as input, and the agent can learn from highly noisy depth predictions. Extensive real-world tests illustrate the possibility of transferring trained network visual information from virtual to real-world environments. The study [8] describes a multi-scenario, multi-stage training framework for optimizing a completely decentralized sensor-level collision avoidance policy. The approach is considered a first step toward minimizing the navigation performance gap between centralized and decentralized techniques. [9] the proposed vision-based method with a complex environment containing obstacles. The suggested framework may result in a policy that allows for a significantly greater number of steps. Decision-making network trained for decentralized computational guiding situations in [10]. LSTM networks and deep neural networks are used to provide an automated and safe environment. The method with segmentation network for continuous action space is proposed in [11]. The method does not require manual labeling, so labor and time are significantly saved.Many of these methods rely on models for environmental dynamics and UAVs. Furthermore, the accuracy of these models may also significantly affect the efficiency of the method.

1.2 Problem Definition

Reinforcement Learning has been successfully applied on application with a simulated environment. The requirements for this application are the high number of samples. In general, DRL algorithms need more than samples to reach a stable state. Obtaining so many samples in the real world without harming to UAV would be extremely tough. As

a result, learning a policy for a real system in a simulated environment is the most straightforward way and then learned policy could be deployed to the real system.

The proposition that UAVs can learn a complicated task via trial and error. The human mind is built to learn in this manner. Intelligent controllers should master tasks given sufficient practice. The learning processes cannot be done in the real world since UAVs are costly vehicles. Map information is provided as a simulated environment. The UAV's global path planner used earlier map information to assign numerous waypoints to the destination. Then, the intelligent controller will create a collision-safe action command to drive the UAV to the next waypoint without colliding.

One critical concern that must be addressed is that most of the classical RL approach requires data manipulation. It significantly caused labor and time loss. We overcome this issue by implementing the DRL. This method allows that the network can be fed with raw input data. We designed the network with multilayer and explored how these techniques can affect the problem. As a result, a unique methodology must be designed that allows DRL techniques to learn optimum policies for real-world systems, while also decreasing the sample size required for these systems.

1.3 Thesis objective

The objective of the thesis:

Application of the DRL network to collision avoidance problem. the goal of the thesis is to create an agent that learns maneuvers successfully to avoid the nearby UAVs.

To achieve this goal, existing methods are examined and compared their advantages and disadvantages. At first, the Markov Decision Process provides a standard framework for defining a decision-making scenario in the environment. The Markov Decision Process can be solved, but it requires the transition function and reward function. Besides, the Q value methods do not need to know how the environment works. Deep Q Learning (DQN) algorithms combine classic Q-learning techniques with a convolutional neural network to learn how to act directly from raw sensory input. Proximal Policy Optimization (PPO) and Asynchronous Actor-Critic Agent (A3C) learn based on the policy and provide monotonic policy improvements. We would like to compare the performance of these algorithms and implement the maneuver to avoid nearby UAV tasks.

Chapter 2

The Reinforcement Learning Problem

2.1 Introduction

Reinforcement Learning (RL) is a subset of machine learning in which learning is achieved by contact with an environment. It is objective-oriented learning where it is not taught to the learner what actions to take; instead, the learner learns from the result of its actions [12]. In particular, RL enables a computer to make sequential decisions to maximize the total reward for the task [13]. The RL includes an agent and an environment. The agent interacts with the environment by executing an action and receives a reward depending on the action is performed. Based on the reward, the agent realizes whether the action was good or bad. The difference between RL and supervised learning is that in RL, the agent does not know what the proper action is at a particular moment and must find that out based on trial and error. The elements of RL will be discussed in this chapter.

2.2 Markov Decision Process

Reinforcement learning uses the formal framework of Markov decision processes to describe the relationship between a learning agent and its environment considering the actions and rewards. The structure intends to represent essential artificial intelligence features in a simple way.

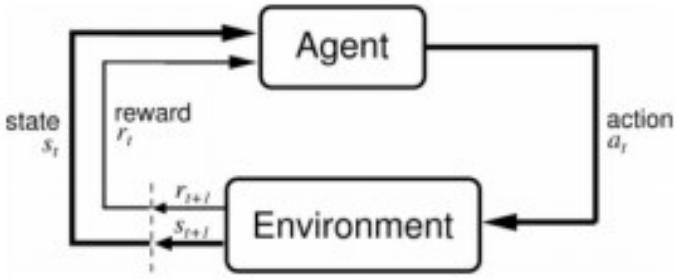


Figure 2. 1 Schematic diagram of the reinforcement learning process

Markov Decision Process (MDP) is defined as a four-tuple < S, A, R, T > where: S set of states, s ∈ S, s denotes possible states space; A set of actions, a ∈ A, a denotes possible actions space.

Scalar reward R, defines the reward for taking action in state s and ending up in state s'. Transition function, T, define the probability of taking action in state s and ending up in state s'.

$$p(s'|s,a) = P_r\,(S_{t+1} = s'|S_t = s, A_t = a) \tag{2.1}$$

The agent aims to increase its reward over time, granting slightly more choice to the short-term than the long-term reward. The discounted total reward over time for continuous task is given by:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.2}$$

where γ is a discount factor such that $0 < \gamma \leq 1$, the future reward is exponentially discounted [14]. For a discrete task, the discounted return G_t, which is given by:

$$G_t = r_t + r_{t+1} + r_{t+2} + \cdots + r_{N-1} \tag{2.3}$$

N represents the end of the episode; t donates an index of time. The reward function and the transition probabilities are the most critical components of MDP. In many cases, the environmental dynamics T and R are considered unknown.

2.3 Reinforcement Learning Framework

As mentioned before, reinforcement learning attempts to develop algorithms that can learn and respond to changes in the environment. The following diagram shows a flowchart that representation of a reinforcement learning scenario [15].

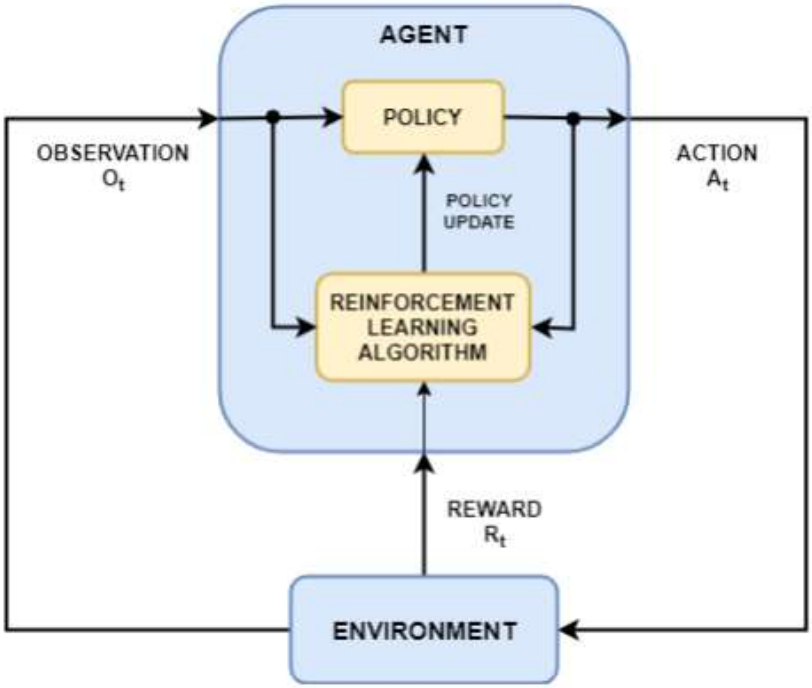


Figure 2. 2 General representation of a reinforcement learning scenario.

The steps in the standard RL algorithm are as follows:

1. The agent interacts with the environment by performing an action.
2. The agent performs an action and moves from one state to another.
3. The agent gets a reward, depending on the action it has carried out.
4. If the action were good, that is, if the agent obtained a positive reward, the agent would choose to perform that action; otherwise, the agent would attempt to perform another action, resulting in a positive reward.

2.3.1 Environment and Agent

Agents are a model/algorithm that makes intelligent decisions. By communicating with the environment, agents take action and receive rewards based on their actions. The environment briefly defines the world in which the agent acts. It is responsible for controlling agent behavior and providing feedback on how well the agent does a given task [12][15].

In specific, an agent can be composed of several functions and models to assist its decision-making. There are three major components that an agent can have: Policy, Value Function, and model.

2.3.2 Policy

A policy is an algorithm or a set of rules that defines the agent's behavior in the environment. The way the agent chooses which action to perform depends on the policy. More formally, a policy is a mapping from state to action, typically denoted as π. If the agent applies stochastic policy $\pi(a|s)$ at time t, $\pi(a|s)$ is a probability distribution over the actions A_t =a, for a state S_t = s. [14] The agents could follow deterministic policy π(s), at time t, π(s) maps a state S_t, over an action, A_t

$$A_t = \pi(\mathrm{St}) \tag{2.4}$$

(2.4) suggests that, given current states, an agent determines its actions. The deterministic policy could represent any interaction between environment and agent, proving a state as input and output an action [15].

2.3.3 Value Function

The Value Function is an approximation of how good it is to be in a specific state with a policy π for the agent. It is often donated by V(s). If the agent performs by following the policy π, V(s) is the expected discounted return of states. The Value Function can be defined as follows:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t \,|s_t = s] \tag{2.5}$$

In the value function, we can substitute the value of R_t from (2.2.2) as follows:

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \,|s_t = s\right] \tag{2.6}$$

Consideration of the value function for a state-action pair is more appropriate than a state. In this case, the Q function, also known as the action-value function, can be defined as expected rewards when taking action "a" in state s and, afterward, following policy π. The Q function can be defined as follows:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[\sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \,|s_t = s, a_t = a\right] \tag{2.7}$$

2.3.3.1 Bellman Equations

The Bellman equation is used to solve MDP, which means finding the value functions with optimal policies. The value function may determine different values following its policy. If $V^{\pi}(s) \geq V^{\pi'}(s)$ for s ∈ S, the value function $V^{\pi}(s)$ is known to be an optimal value function with an optimal policy π that yields the highest value compared to all other value functions. In general, the optimal value function $V^{*}(s)$ is defined as follows:

$$V^{*}(s) = max_{\pi} V^{\pi}(s) \tag{2.8}$$

Similarly, the optimal action-value function, $Q^{*}(s,a)$ is given by:

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.9}$$

In addition, the following equation is defined for the optimum policy:

$$V^{*}(s) = max_{a} Q^{*}(s,a) \tag{2.10}$$

Derivation of the bellman equation for the value function can be defined as follows:

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s')\right] \tag{2.11}$$

Similarly, the bellman equation for the Q function is:

$$Q^{\pi}(s,a) = \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} Q^{\pi}(s',a')\right] \tag{2.12}$$

As a result of substituting equation (2.2.12) and (2.2.10), The Bellman optimality equation is given as follows:

$$V^{*}(s) = max_{a} \sum_{s'} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \sum_{a'} Q^{\pi}(s',a')\right] \tag{2.13}$$

Solving the Bellman optimality equation is the proper way to find an optimal policy. Many methods have been suggested to solve this equation. The methods are categorized

as Model-Free and Model-Based Algorithms. Those methods will be discussed in the following sections [12][14] .

2.4 Model-Free algorithms

The bellman optimality equation is solved by model-free methods where the model of the environment is not existing. The transition probabilities and the reward functions are not known. So those values are approximated by doing the system rollout. Depending on the way of finding an optimal policy, Model-free methods are categorized as Value Function-Based methods and Policy-Based methods. Moreover, the model-free algorithm is classified as off -policy and on-policy. On-Policy approaches use the current action to determine the action-value-function through the same policy. In contrast, Off-Policy uses a different policy to change the same action-value function.

2.4.1 Value function-based algorithms

Monte Carlo Methods

The value function and policy can be identified by the Monte Carlo method since the environment's model is unknown. The algorithm can learn from the agent's experience by storing the episode reward for the corresponding action state pair. The Monte Carlo method cannot recalculate its value and policy based on the fashion of gaining experience during the learning (online learning) since the methods need to wait till the end of the episode [16]. The methods follow an iterative scheme consisting of two phases called estimation of the value Function and optimization of the value function. In the first phase, the return is calculated and stored for the episode by initializing random value to value function. Instead of the expected return, the mean return is taken as an estimation of the value function. Besides, policy iteration is implemented to find an optimal policy. The value function is calculated based on the randomly estimated policy, known as Policy Evaluation. If the estimated value function is not optimal, the algorithm finds new, improved policy, the step known as Policy Improvement. In the second phase, those two steps interact with each other, and the algorithm effectively runs as a loop. The policy is always updated concerning the value function, and the value function is continually updated under the policy until there is no change. When convergence reaches between the policy and the value function, it means that the optimum value function and the optimum policy is found [12].

Temporal-difference methods

The temporal difference (TD) method learns from experience as Monte Carlo methods. The main differences are that TD follows the online fashion, which means the completion of the episode end is not needed for return. TD learning follows an update rule, given by:

$$V(s) = V(s) + \alpha(r + \gamma V(s') - V(s)) \tag{2.14}$$

Where α donates learning rate, also called step size, r is a reward, s' is current state, and s is a previous state.

The method estimates the value function based on equation (2.2.14). However, the optimal value function is not found. For that, two algorithms commonly used to optimize the value function are called Q learning and SARSA.

2.4.1.1 Exploration vs Exploitation

The exploration vs exploitation is a general trade-off problem in Reinforcement Learning. Exploration means that the agent tries all possible actions to explore the unknown states. On the contrary, in exploitation, the agent selects proper action, which provides a high reward given current states. The balance in the Exploration vs Exploitation dilemma significantly improves the agent learning performance. Firstly, the agent expected to explore the action space intensely because the knowledge of the agent was not sufficient yet. When the knowledge is gathered, the agent is required to exploit for a better result. Some approaches for the exploration vs exploitation dilemma will be described following subsection.

2.4.1.1.1 Epsilon-Greedy Strategy Selection

The most traditionalist action selection policy is called Epsilon-Greedy Strategy. The agent will always choose the action with the highest reward. This action is selected with probability ε and determines a random function with probability 1 – ε. The epsilon greedy function is given by:

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \dfrac{\varepsilon}{|\mathcal{A}(s)|} & if\ a = argmaxQ(s,a)\ , a \epsilon \mathcal{A}(s) \\ \dfrac{\varepsilon}{|\mathcal{A}(s)|} & otherwise \end{cases} \tag{2.15}$$

The ε takes a value between 1 and 0 and decreases over time.

2.4.1.1.2 On-Policy versus Off-Policy Methods

As stated before, on-policy methods rely on the same policy to select the action while, off policy, the method uses different policies. Both methods relate to the Exploration vs Exploitation dilemma. Suppose the greedy exploration is considered under the on-policy methods. In that case, the agent will select the action immediately. On the other hand, off-policy methods estimate the optimal value, eliminating the impact of the bad decision on the agent's part.

2.4.1.2 Q learning

Q learning is an off-policy control algorithm that estimates the value function, recalculate the state-action pair by following the update rule given by:

$$Q(s,a) = Q(s,a) + a(r + \gamma maxQ(s'a') - Q(s,a)) \tag{2.16}$$

The implementation algorithm of Q learning is shown below:

Algorithm 2.1: Q learning

Initialize random $Q(s,a)$
Repeat
Initialize the first step s_1
For r i ← 1 to N **do**
Chose at from s_t by exploration strategy -greedy
Apply a_t
Move s_{t+1}, receive reward r_t
$Q(s_t,a_t) = Q(s_t,a_t) + a(r_{t+1} + \gamma max_a Q(s_{t+1},a_{t+1}) - Q(s_t,a_t))$
End for
Until end of the episode

2.4.1.3 SARSA

SARSA (State-Action-Reward-State-Action) is an on-policy control algorithm that estimates the value function, focuses on state action value by following the update rule given by:

$$Q(s,a) = Q(s,a) + a(r + \gamma Q(s'a') - Q(s,a)) \quad (2.17)$$

The algorithm described below:

Algorithm 2.2: SARSA

Initialize random $Q(s,a)$
Repeat
Initialize the first step s_1
Chose a_1 by exploration strategy -greedy)
For r i ← 1 to N **do**
Apply a_t
Move s_{t+1}, receive reward r_t
Select next action a_{t+1} by exploration strategy -greedy)
$Q(s,a) = Q(s,a) + a(r + \gamma Q(s'a') - Q(s,a))$
End for
Until end of the episode

Function Approximation

Earlier, the values of the value function for each state are stored in a tabular form. When the complexity of the problems state is enormous, the use of tabular format is not efficient. This tabular form requires a tremendous amount of memory.

The function approximators are proposed to deal with the problem. Different types of function approximators are used in literature. The neural network approach commonly used nowadays, before neural network trend, Linear function approximators [14] is often enough for less complex environments. The advantages of the neural network approach are that the representation capability for the value function is high when the environment is complex. The high capacity reduces the training time of reinforcement learning and memory usages. Although there are dual benefits of the neural network approach, the application of neural networks as a function approximator needs some additional changes to allow them to be implemented effectively.

2.4.2 Policy-Based Methods

2.4.2.1 Policy Gradient Method

Policy Gradient Methods is one of the RL reinforcement class, which optimizes the expected return J with the parameter vector θ_π. $J(\theta_\pi)$represents the quality of the policy which defined in each episode, and the objective function of the optimization problem is given by:

$$\theta^* = \arg\max_{\theta} J(\theta) \tag{2.18}$$

The gradient ascent method is used to update the policy parameter vector. The positive update direction is given by the gradient $\nabla_\theta J_\theta$, which aims to identify policy performance [17]. The update is given by:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t) \tag{2.19}$$

Policy Gradient Method has many advantages. Firstly, they learn the related probabilities by taking an action that is not possible in the value-based approach. Secondly, their convergence properties are better than the value function approach because the policy modifies explicitly [18]. The policy parameter is more manageable to represent specific issues than the value function, so prominent oscillation on the policy is avoided. lastly, the Policy Gradient Methods is appropriate for the continuous action spaces [14].

2.4.2.2 Actor-Critic Method

The Actor-Critic method is categorized as an intersection of value-based method and policy-based method. Therefore, it takes advantage of those two methods. The method consists of two networks. The Policy network is called Actor, which selects an action to take in a given state. The critic is the value network that evaluates the goodness of the action which is taken. The critic network is considered as feedback that assists the actor-network in finding an optimum policy [46].

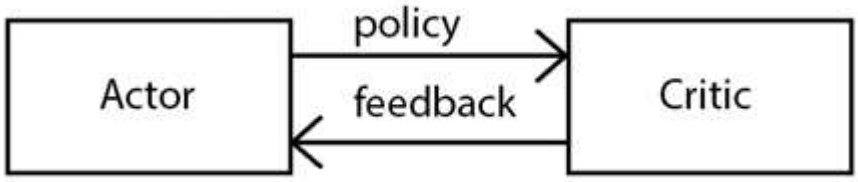


Figure 2. 3 : The Actor-Critic Structure

In the previous section, as stated that the reinforce algorithm compute the gradient as follow:

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - V_\phi(s_t) \tag{2.20}$$

The reward is calculated immediately by taking the current state reward and the discounted value of the next step:

$$R \approx r + \gamma V(s') \tag{2.21}$$

The policy gradient can be rewritten by replacing R_t with equation (2.2.21):

$$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_t|s_t)(r + \gamma V_\phi(s_t^{'}) - V_\phi(s_t) \tag{2.22}$$

The modification in the policy gradient allows that the reward is calculated in each step of the episode. After computing the policy gradient, the policy network (Actor) parameters θ is updated as:

$$\theta = \theta + \alpha \nabla_\theta J(\theta) \tag{2.23}$$

The parameters of the critic network ϕ are also updated in each step of the episode similarly θ. The loss of the critic network is given by:

$$J(\phi) = r + \gamma V_\phi(s_t') - V_\phi(s_t) \tag{2.24}$$

Here, $r + \gamma V_\phi(s_t')$donates the target value, while $V_\phi(s_t)$ represents the value, which is predicted by the network. The loss function optimizes using by gradient $\nabla_\phi J(\phi)$ and, the critic network parameters ϕ is updated by the rule as following [45]:

$$\phi = \phi - \alpha \nabla_\phi J(\phi) \tag{2.25}$$

The algorithm of the Actor-Critic Method is described below:

Algorithm 2.3: Actor-Critic Method

Initialize ϕ randomly.
Initialize θrandomly.
Repeat
For t ← 0 to T-1, **do**
Select action a_t using policy $\pi_\theta(s_t)$
Apply the action a_t
Move s_t', receive reward r
Evaluate the policy gradient:
$\nabla_\theta J(\theta) = \nabla_\theta \log \pi_\theta(a_t|s_t)(r + \gamma V_\phi(s_t') - V_\phi(s_t)$
Update the actor-network parameters θ:
$\theta = \theta + \alpha\nabla_\theta J(\theta)$
Calculate the Loss function of the Critic Network:
$J(\phi) = r + \gamma V_\phi(s_t') - V_\phi(s_t)$
Evaluate the gradient ascent$\nabla_\phi J(\phi)$
Update the Critic Network parameters ϕ:
$\phi = \phi - \alpha\nabla_\phi J(\phi)$
End for
Until end of the episode

2.5 Model-Based Algorithms

As mentioned earlier, Model-based methods do not require the environment model, which means the agents use the model of the environment $P(s'|s,a)$ to learn the value function. If the sequence of action $a_1, a_2, \ldots a_n$ apply in environment state s, the agent would approximate how the state changes and different state changes are realized. Accordingly, the agent selects the best action to take based on environment knowledge. The flexibility of the action selection significantly improves the sample efficiency of the method.

2.6 Summary

In this chapter, a variety of the essential principles related to reinforcement learning are introduced. Many standard RL algorithms have been explained. Briefly, if the sample efficiency is a crucial factor, Model-based methods are recommended. However, when the knowledge of the environment is not available, the model-free methods are appropriate. Traditional approaches are not good enough for the dimensionally high problem. In the collision avoidance problem, the Deep neural network (DNN) concept is proposed because of its complexity. The fundamentals of the DNN and the usage as a function approximator will be discussed in the next chapter.

Chapter 3

Deep Neural Network Concept

3.1 Introduction

Deep neural networks (DNN) are a widespread subfield of Machine learning (ML) that is combined with algorithms inspired by the biological structure and functionality of the human brain. Many applications are based on DNN, such as speech recognition [19], image classification [20], UAV navigation, and object detection [21,22], autonomous driving [23]. Recent research proves that Deep Learning (DL) could manage all the ML tasks and achieve an improved result. Therefore, DL has become a pervasive field to solve estimation problems rather than only being limited to areas of the image, voice recognition,etc.

Chapter 3 will give information about the basic concepts of the neural network, while the emphasis is on Convolutional Neural Networks. Then, the layer of the Deep Neural Network will be presented.

3.2 Fundamental of Deep Neural Networks

3.2.1 Neuron Model

The human brain can perform complex processing. The neurons are tightly connected and interact with each other in parallel. These neurons receive and send signals via the connection(synapses). Increased or weakened power of the synaptic links makes our learning easier. The illustration of the biological diagram can be shown below:

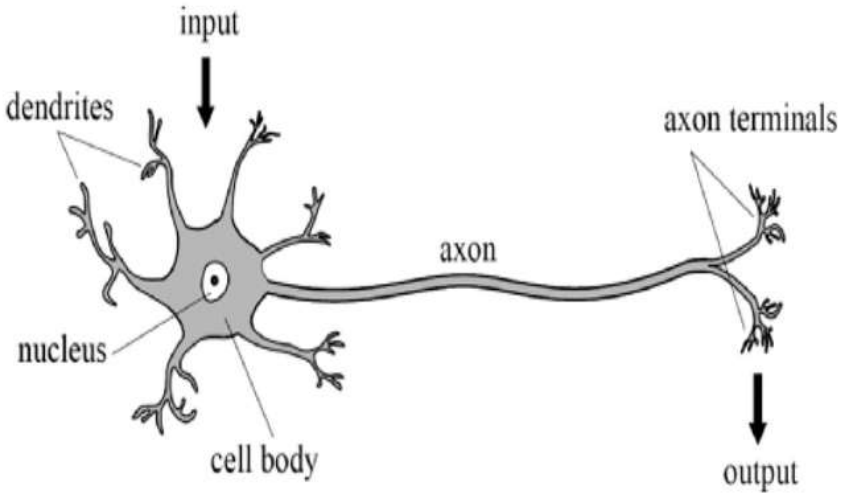


Figure 3. 1: Biological neuron diagram [24]

It can be seen in the diagram, each neuron comprises of dendrites as input and axons as an output. The weight of the information is received and sent via the connection of those components. The artificial neurons and neural networks are mainly biologically inspired by neural concepts in the human brain.

3.2.2 Artificial Neuron

The artificial neuron is the basic component of the DNN. Figure 3.1 represents the structure of the artificial neuron. In the representation, $\vec{X} = (x_{1,} x_{2,} \dots, x_{n,})$ donates the input of the neuron and $W = (w_{1,} w_{2,} \dots, w_{n,})$ weight of the neurons calculated based on $\vec{X}$. Artificial neurons attempt to approximate input and output. The output is calculated by the sum of the W and a bias term 'b'. the function that represents the input and output relationship is given by:

$$W^T \vec{X} + \vec{b} = \vec{y} \tag{3.1}$$

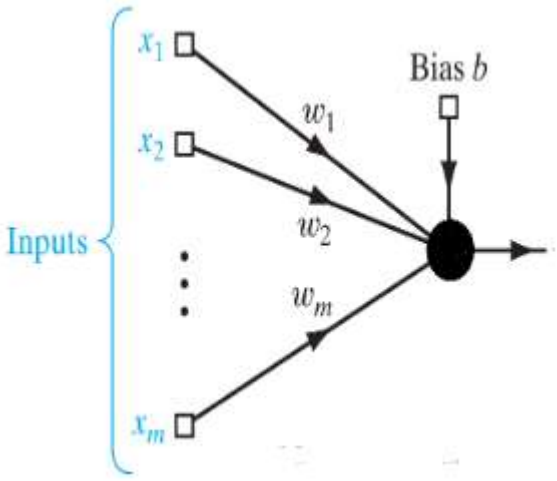


Figure 3. 2: Representation of the Artificial Neuron

3.3 Feedforward Neural Networks

Feedforward neural network is an architecture that mainly consists of an input layer, hidden layer, and output layer. Each layer may have several neuron connections with other neurons in the hieratical organization. Based on the received input, connected neurons send a signal to each other, and the network learns through the particular feedback mechanism

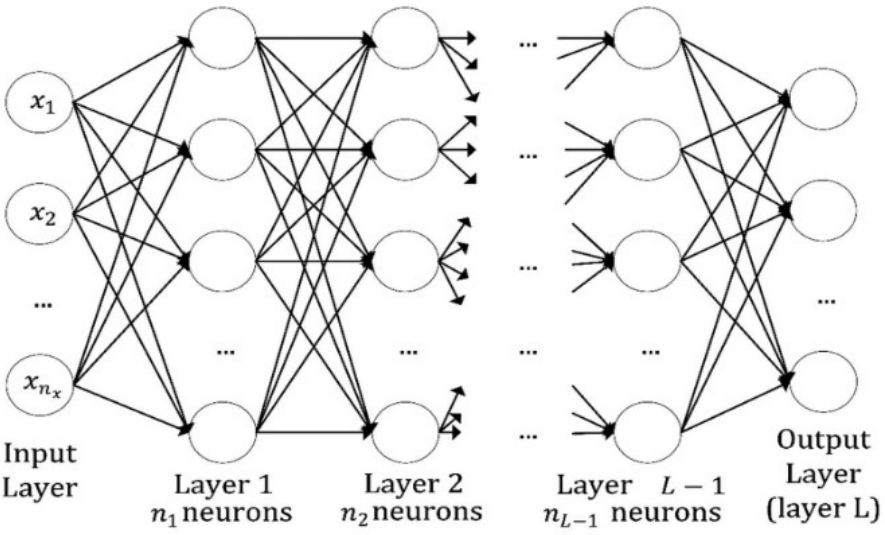


Figure 3. 3: Representation of the Feedforward Neural Network [25]

Basic Neural networks cannot deal with the nonlinear problem. Current models use an activation function that provides nonlinear output. The form of the model can be given by:

$$y = \phi\left(\sum_i w_i x_i + b\right) \tag{3.2}$$

In the equation, x denotes input of the model, y is output, w represents the weights, φ is an activation function, and b is bias.

3.3.1 Activation functions

The activation function is a node that determines the nonlinear output or set of output. It provides a satisfactory solution when the problem is nonlinear. Many activation functions proposed in the literature such as sigmoid, hyperbolic tangent, SoftMax, rectified linear unit. In the multi-layer model, the Activation function is often chosen as a hyperbolic tangent [26]. The activation functions that used in the neural network can be shown below:

Sigmoid	$f(x) = \frac{1}{1+e^x}$ the output varies in (0,1)
Hyperbolic Tangent (*tanh*)	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ the output varies in (-1,1)
SoftMax	$f(x) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}}$ K donates the number of elements.
Rectified Linear Unit (ReLu)	$f(x) = \max(0, x) = \begin{cases} 0 & if\ x < 0 \\ x & if\ x \geq 0 \end{cases}$

Table 3. 1: Activation Functions

Among the activation functions are shown above, ReLu is most widely used because all negative values of x are turned into 0, which allows the model to train and converge faster.

3.3.2 Loss Function

The loss function is a vital aspect of the neural network as well as their learning process. Model error is calculated from the predicted output, Model error also represents the systems performance [24].

There are two types of errors called Local Error, Global Error. Local error is the difference between one specific real sample output and predicted output. However, the global error is a measure calculated by the sum of all local errors and shows how the model works on the training data.

Many methods proposed in the literature like Mean Absolute Error (MAE), Mean squared error (MSE), Root Mean squared error (RMSE). Commonly used methods are shown below:

Mean Absolute Error (MAE)	$MAE = \frac{\sum_{i=1}^{N} \left\| \hat{y}_i - y_i \right\|}{N}$
Mean Squared Error (MSE)	$MSE = \frac{1}{N} \sum_{i} \left\| \hat{y}_i - y_i \right\|_2^2$
Root Mean Squared Error (RMSE)	$RMSE = \sqrt{\frac{\sum_{i=1}^{N} \left\| \hat{y}_i - y_i \right\|_2^2}{N}}$

Table 3. 2: Loss Functions

Each method has its use case, advantages, and drawbacks, so choosing the loss function depends on the learning task.

Training the Neural network

The neural network learning process aims to find a weights W. Variety of learning algorithms have been proposed for the training. The most used training method is backpropagation. The main steps of the backpropagation are started with initializing the weight randomly. Choosing the weight that will make the activation function zero is an important issue that needs to be avoided. In the second step, Feedforward, the input nodes send the received signal to the neuron in the hidden layer. Each output neuron calculates the response against the given input. The output for the pattern computes the error between the predicted value and real value. When all errors are calculated, it goes back layer by layer, starts from the output layer, and affects all the neurons in the hidden layer. lastly, the weight vector W is updated by the gradient descent method [27,28].

The gradient descent method updates the weight in the direction of the local-global minimum by taking a step toward the negative direction of the loss gradient. It can be represented in the following form:

$$x_{k+1} = x_k - c_k \nabla f(x) \tag{3.3}$$

The update rule for the weight is given by:

$$\theta_l = \theta_l - \alpha \frac{\partial J}{\partial \theta_l} \tag{3.4}$$

θ_l donates model parameter, which includes W and b. Here, the aim is to minimize the overall network loss and to change its weights accordingly.

Regularization

One of the typical problems in the neural network is overfitting. The regularization is a technique used for preventing the overfitting problem. Three most popular and well-known methods are L_1, L_2 and dropout.

L_1 and L_2 regularization techniques are based on adding a penalty to the objective function, which decreases the capacity of the model. Adding a penalty to the objective function also pushes and makes closer the weight to the direction of the origin. As a result, the objective function is given by:

$$J(w) = E(w) + \alpha\Omega(w) \tag{3.5}$$

The added penalty for L_1 and L_2 regularizations are shown below [29]:

L_1 regularization: $\Omega = \|w\|_l = \sum_i w_i$

L_2 regularization: $\Omega = \frac{1}{2}\|w\|^2 o$

The idea of the dropout is that some neurons from the hidden or input layer are deleted with predefined probability $P[l]$ in each iteration. Deleting the neurons means setting their output zero during the forward pass process, which does not affect all the neural networks during the training. Considering a hidden layer with 10 neurons, if the $P[l] = 0.5$, it means that half of the neuron in this hidden layer would be removed in the training iteration. Commonly, $P[l]$ is set to 0.2 in the input layer. 0.5 has the best regularizing impact on the hidden layer during the training. A dropout regularization is a powerful approach because of its cost-effectiveness [24,29,25].

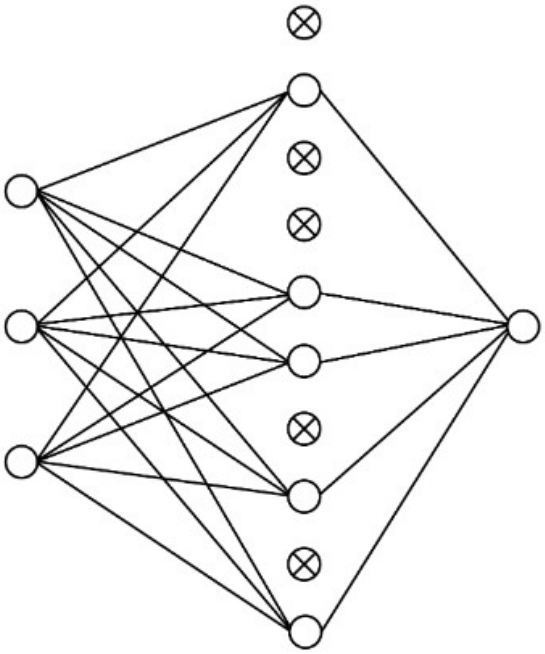


Figure 3. 4: The neural network after applied the dropout regularization.

3.4 Convolutional Neural Network

The convolutional neural network is a multi-layer architecture that has the capability to process large image data sets and extract the features without needing any handcrafted set of features. Similar to other multi-layer networks, CNN comprised a set of layers with weight W and bias b and trained with backpropagation algorithms. However, CNN uses filters instead of single weights vector to minimize the number of weights and extract features. The architecture considers the input as an image inspired by the human visual cortex and how the human brain recognizes the object in the world. This concept makes the forward function easy to implement [30]. As stated before, CNN is made of multi-layer, which can be shown in figure 3.5. The layer of CNN will be described in the following subsections.

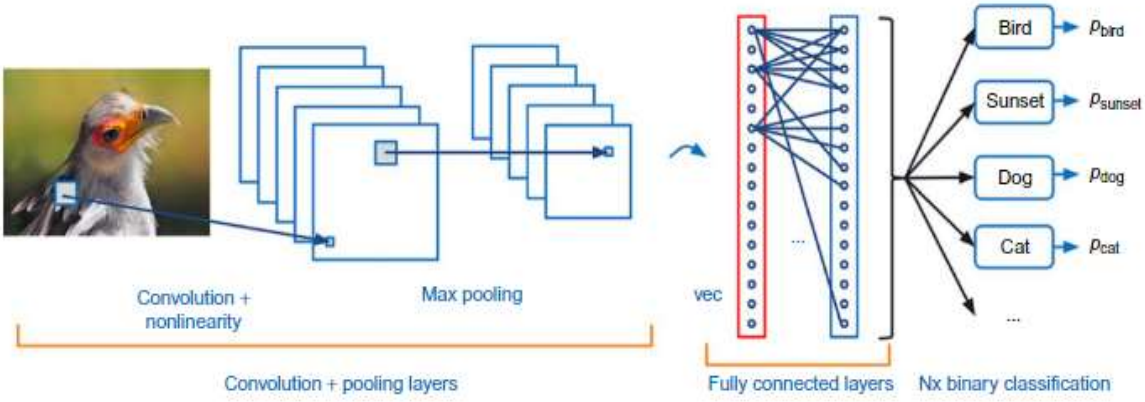


Figure 3. 5: Architecture of the Convolutional Neural Network [31]

3.4.1 Convolutional Layer

The convolutional layer connects the layer by a linear operation known as convolutions. Mathematically, the convolution is given by:

$$(x * w)(a) = \int x(t)w(a-t)da \tag{3.6}$$

In equation 3.6, x denotes the input, w represents the kernel, and the output called features map in the concept of CNN. In this case, t should consider discrete since the kernel and input are continuous functions so that the discrete convolution equation can be defined as below:

$$(x * w)(a) = \sum_{a} x(t)w(t-a) \tag{3.7}$$

In the CNN, the input is commonly multidimensional, and the kernel usually has a multidimensional parameter. It means that finite summation can be implemented in equation 3.7, and the discrete convolution can be rewrite shown below:

$$(l * K)(i,j) = \sum_{m}\sum_{n} l(m,n)K(i-m, j-n) \tag{3.8}$$

In the convolution layer, the size for each kernel is the same but, the values of the kernel are different, after performing convolution between those kernels and input, The features map is obtained by applying the activation function called Rectified Linear

Units (ReLU) to the result of the convolution. ReLU increases the nonlinearity and allows for fast convergence.

3.4.2 Max and Average Pooling Layers

The pooling layer is commonly located between 2 convolution layers. It takes the output from one conventional layer to reduce the number of the parameters and dimensionality of the feature map. It has no learnable parameters. There are two types of pooling functions called Max and average pooling functions. As their name indicates, max-pooling considers the maximum value in each input feature. In contrast, the average pooling considers the average value. The max-pooling function is mostly used because of its simplicity and high performance. Consider the pooling with the 4x4 input and 2x2 kernel, figure 3.6 shows the result of the pooling by using both Max and average:

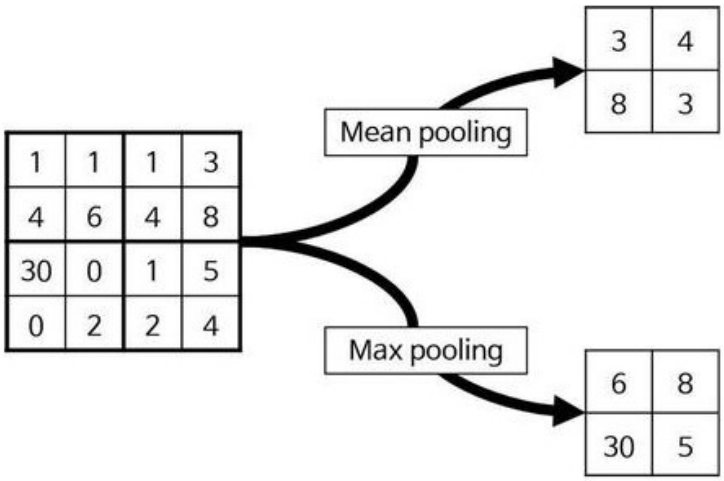


Figure 3. 6: Pooling using two different methods [32]

3.4.3 Fully Connected Layer

The fully connected layer is located at the end of the CNN architecture. The neurons are connected from the last pooling layer to the Fully connected layer. The output values of the features map are received here, and final parameters are calculated by minimizing the loss function following the method called backpropagation, which is explained before.

3.5 Summary

In this chapter, the fundamentals of the deep neural network and its learning process are discussed, and the deep neural network architecture CNN is introduced. CNN proved that it could extract high-level features from the high dimensional data without handcrafted features. The neural network can be used as a function approximator with RL because of its fast-training time and ability to scale dimension.

Chapter 4

Deep Reinforcement Learning Agents

4.1 Introduction

Deep Reinforcement Learning (DRL) has been implemented successfully and achieved a remarkable result in reinforcement learning research. It is the combination of deep neural network and reinforcement learning, which means Deep Neural Network (DNN) is used as a function approximator or policy. RL framework has proved that the algorithm can learn the task such as playing Atari games [33], planning for autonomous vehicle [16]. Besides, DRL is a suitable approach that it can easily handle the task for UAV and Autonomous Vehicle.

In this chapter, a variety of the DRL agents include Policy Gradient Agents, and Deep Q-Network Agents will be explained in detail. The chapter also provides an overview of DRL algorithms.

4.2 Deep Q-Network Agents

As introduced in chapter 2, the Q learning approach has a finite number of states and a limited number of actions. The algorithm tries to find an optimal Q function through all the possible state-action pairs. This approach is not suitable considering the environment, which has a huge number of states and possible actions. Calculation of the Q function through all the possible state-action pairs is a time-consuming process. The new approach called Deep Q Network (DQN) [4] is introduced to overcome this issue. The parameters θ is used to estimate the Q function, and θ refers to the parameter of DNN. The network is trained with the loss function L (θ), which is given by:

$$L_i(\theta_i) = E[(y_i - Q(x, a; \theta_i)^2)] \tag{4.1}$$

Where $y_i = r + \gamma max_{a'} Q(s', a'; \theta)$ is a target value [34, 12]. The gradient descent method is applied to minimize the loss function.

4.2.1 Experience Replay

As seen in equation (4.1), Q values are estimated from one state s to the next state s'. In this case, DNN cannot distinguish between those steps, and it causes a common problem in a neural network known as overfitting.

To make a more stable training, a replay buffer is created by collecting all transitions as $< s, a, r, s' >$. The algorithm trains the model by choosing the sample randomly from this buffer to avoid the correlation between the steps. This method knows as experience replay. Experience refers to the transition information.

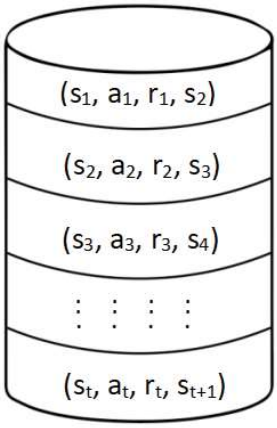


Figure 4. 1: Replay Buffer [35]

Considering that the replay buffer has a limited size, it can store a fixed amount of information. Therefore, the replay buffer is implemented as a queue instead of a list. When the replay buffer is full, the old transition is deleted, and the recently provided transition is added into the replay buffer. As a result, the experience replay method allows the network to learn better [35, 36, 12]. The complete algorithm of DQN with experience replay is shown below:

Algorithm 4.1: Deep Q Learning with Experience Replay

Initialize the Replay Memory D
Initialize the action value function Q with random weight θ
Repeat
Initialize the first step s_1
For r i ← 1 to N **do**
Chose at using Q by exploration strategy -greedy
Apply a_t
Move s_{t+1}, receive reward r_t
Add transition (s_t, a_t, r_t, s_{t+1}) into Replay Buffer D
Sample transition randomly from D

$$y_i = \begin{cases} r_j & if\ episode\ terminates\ at\ s_{t+1} \\ y_i = r + \gamma max_{a'} Q(s', a'; \theta) & otherwise \end{cases}$$

Perform gradient descent on $y_i - Q(x, a; \theta_i)^2$
End for
Until end of the episode

4.2.2 Double DQN

The basic DQN tends to overestimate the Q value, which affects the training performance. The max operator in the Bellman equation causes the overestimation. To handle this issue, Double DQN [37] improvement has been introduced. For choosing and applying the action, the max operator uses the same value. When $Q(s', a'; \theta)$ contains any error because of the noisy environment, $max_{a'} Q(s', a'; \theta)$ is positively affected, which means the Q value is overestimated. To solution for that, a modification for the bellman equation is given by:

$$Q(s_t, a_t) = r_t + \gamma \max_a Q'(s_{t+1}, \operatorname{argmax}_a Q(s_{t+1}, a)) \quad (4.2)$$

Equation (4.2) showed that the Q function is separated into two; each of them calculated independently. The Q value is estimated using the main network weights, while the other Q' value parametrised by the target network weights [12,35,36].

4.2.3 Duelling DQN

There might be a unique state which does not contribute to the learning process. For instance, the state that returns a bad reward after applying all possible actions. Computing the Q value is not efficient in the case of those specific states. To handle this issue, another approach was proposed as dueling DQN [38]. The Q value also can be calculated using the advantage function instead of computing it directly in each state-action pair. The advantage function $A(s,a)$ represent the differences between the Q function and Value function. The different version of the Q function can be defined by rewriting $A(s,a)$:

$$Q(s,a) = V(s) + A(s,a) \tag{4.3}$$

In Dueling DQN, figure 4.2 shows that the last layer of the network is divided into two for computing the Advantage function and value function separately. Those two layers combined, adding another layer called the aggregate layer, which calculates equation (4.3).

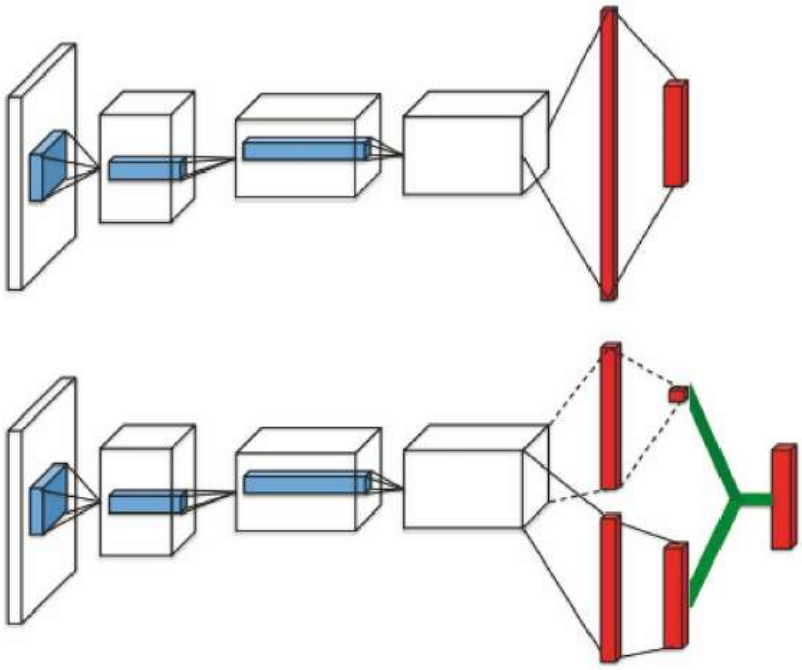


Figure 4. 2: Basic DQN Structure(top)
Dueling DQN architecture (bottom) [4]

4.3 Asynchronous Advantage Actor-Critic Algorithm

DQN algorithm achieved remarkable success in the RL problem. However, they use more memory and need a previous policy to find an optimum policy. To reduce memory usage and increase the training time, the Asynchronous Advantage Actor-Critic algorithm (A3C) [39] proposed. [39] proves that the A3C algorithm cuts the training time without losing the performance. The structure of the A3C algorithm is that more than one agent is used for learning in parallel. The worker agents (workers network) use their exploration policy and copy of the environment to determine experience. Obtained experiences are aggregated and sent to the global agent. The architecture of A3C is shown below:

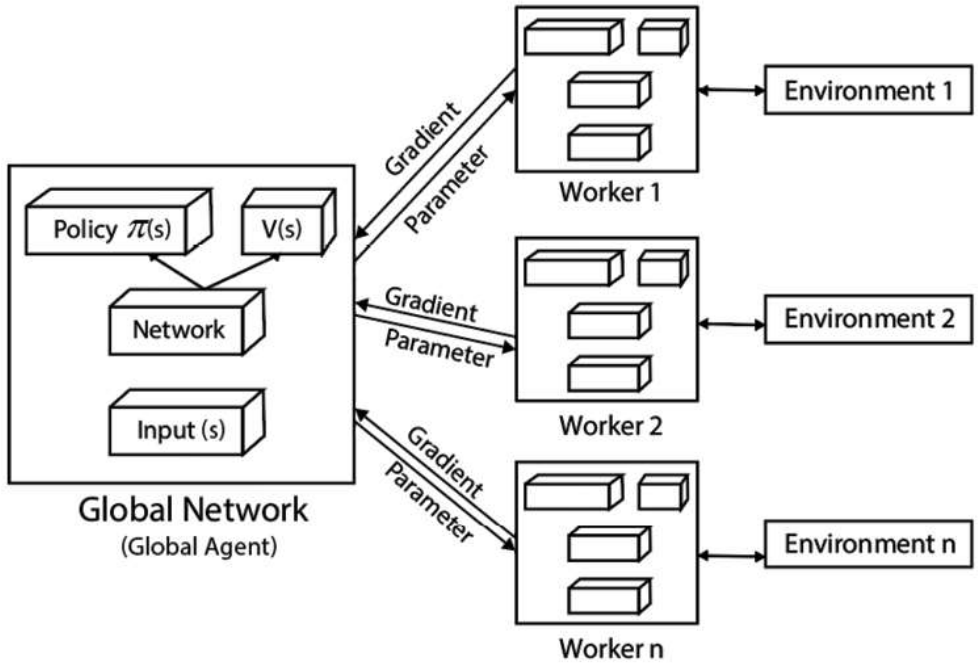


Figure 4. 3: The Architecture of A3C [35]

As seen in figure 4.3, AC3 consists of multiple Worker Agents, which follows the Actor-Critic architecture and collects experience by interacting with their copy of the environment. The global agent receives the experience from the worker agents and updates its parameters. Then, the updated parameters of global agents are sent back to worker agents systematically.

The agent updates the actor-network by computing the policy gradient using equation 2.2.22. Adding the entropy of policy to the equation 2.2.22 achieved better exploration [39]. The modified actor loss function is given by:

$$J(\theta) = \log \pi_\theta(a_t|s_t)\left(r + \gamma V_\phi(s'_t) - V_\phi(s_t)\right) + \beta H(\pi(s)) \tag{4.4}$$

Where H is the entropy of the policy, β regulates the strength of the entropy.

One of the A3C's benefits that implementing the experience replay is not essential to avoid correlation between the episodes. As the algorithm uses numerous agents who interact with their environment and pass the experience to the global agent, the possibility of having a correlation between the experiences is low. The experience replay requires much memory, keeping all the experiences. A3C does not need that memory. It would decrease memory usage and training time.

4.4 Trust region policy optimisation

Trust Region Policy Optimization [40] is one of the policy gradient algorithms trying to find an optimum policy by updating the parameter. In the traditional policy gradient algorithm, the learning rate α is kept small, so the policy improves slowly. Suppose the learning rate α is high. In this case, the difference between the policy used in the previous iteration and the policy used in the current iteration would be increased, causing a model collapse problem. To solve this problem, TRPO improves the policy in each step and applies a constrain using the measure known as Kullback–Leibler (KL). The measure identifies the differences between the given distribution. The following objective function is optimized in each iteration:

$$\begin{aligned} maximize\ \theta \quad & E_s\, \pi_{\theta_{old}}, a\pi_{\theta_{old}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A_{\theta_{old}}(s,a) \right] \\ subject\ to \quad & E_{s,}\, \pi_{\theta_{old}} \left[DKL(\pi_{\theta_{old}}(-|s) || \pi_{\theta_{old}}(-|s)) \right] \leq \delta \end{aligned} \tag{4.5}$$

δ donates the step size, which controls the KL divergence between the old policy and new policy. $\mathcal{P}_{\tilde{\pi}}(s)$ is discounted visitation frequency of the new policy.

TRPO assures monotonous policy improvement. It guarantees that the policy is improved in every iteration. The algorithm also has an outstanding performance in continuous environments [35].

4.5 Proximal Policy Optimization Agents (PPO)

The Proximal Policy Optimization algorithm [41] is a popular policy gradient method that proposed improvement upon the TRPO algorithm. In TRPO, the algorithm ensures policy improvement by imposing the KL constraint. δ controls the KL divergence between the old and new policies. This divergence is needed to calculate in each

episode, and it requires more computation power. PPO does not impose any constraint; instead, modify TRPO objective function to ensure policy improvement. Two types of PPO algorithms are proposed in the literature.

Proximal Policy Optimization with Penalty

In this approach, the Penalty term is used in the TRPO objective function instead of KL constraint. Also, modifying the penalty coefficient during the training time guarantees policy improvements. Before the modification, the TRPO objective function simplified by writing $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$:

$$L(\theta) = \mathbb{E}_t[r_t(\theta)A_t] \tag{4.6}$$

$A_{\theta_{old}}(s, a)$ can be written as A_t since the objective function is computed using the old policy $\pi_{\theta_{old}}$. The new objective function can be rewritten by changing the Kl constrain to penalty term:

$$L(\theta) = \max_\theta \mathbb{E}_t\left[r_t(\theta)A_t - \beta KL(\pi_\theta(a_t|s_t)||\pi_{\theta_{old}}(a_t|s_t))\right] \tag{4.7}$$

β donates the penalty coefficient and controls the penalty size. The meaning of large β is the huge difference between the old and the new policy. When the β is small, the tolerance between the old and new policy is high [42].

Proximal policy Optimization with Clipped objective

PPO algorithm with penalty is remarkably reduced to the need of the computation power. However, finding a suitable penalty coefficient is still an issue. There is no fixed penalty coefficient because each environment has different properties. Besides, Maximizing $L(\theta)$ leads to an enormous policy update. To manage the policy update, the objective function is rewritten by adding a new function known as clipped:

$$L(\theta) = \mathbb{E}_t[min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)] \tag{4.8}$$

The first term of the equation $min(r_t(\theta)A_t$ represents the objective function. The second term is known as clipped objective function and given by:

$$clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t \tag{4.9}$$

As seen in equation (4.9), the $r_t(\theta)$ is clipped between the interval $1-\varepsilon$ and $1+\varepsilon$. The $r_t(\theta)$ managed by considering the advantage function A_t. There are two possible cases:

Case 1: $A_t < 0$

A positive advantage points out that the corresponding action should be selected among all possible actions. $r_t(\theta)$ for this action is increased to have a higher probability of being selected. However, a huge increment on the value of $r_t(\theta)$ leads to significant policy differences. $r_t(\theta)$ is clipped at 1+ε to manage the increment.

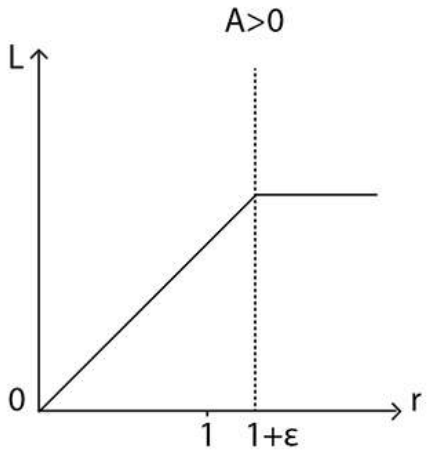


Figure 4. 4: The Value of $r_t(\theta)$, A >0 [41]

Case 2: $A_t < 0$

Negative advantage means that the corresponding action should not be selected among all possible actions. $r_t(\theta)$ for this action is decreased to have a less probability of being selected. However, a Huge decreasement in the value of $r_t(\theta)$ leads to significant policy differences. $r_t(\theta)$ is clipped at 1-ε to manage the decreasement [35].

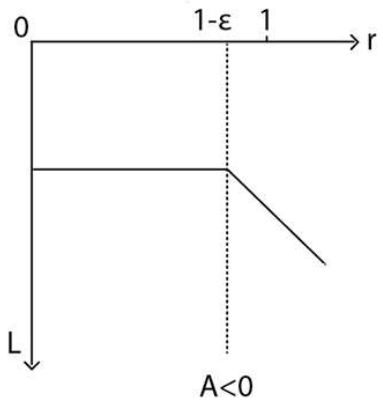


Figure 4. 5 : The Value of $r_t(\theta)$, A < 0 [41]

The PPO with clipped Objective algorithm is shown below:

Algorithm 4.2: PPO with Clipped Objective

Initialize ϕ randomly
Initialize θ randomly
Repeat
Collect *N* number of trajectories using policy π_θ
Compute reward R_t
Evaluate the gradient of the objective function $\nabla_\theta L(\theta)$
Update the policy network parameter θ
$\theta = \theta + \alpha\nabla_\theta J(\theta)$
Calculate the Loss function of the value network:
$J(\phi) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T-1}(R_t - V_\phi(s_t))^2$
Evaluate the gradient ascent $\nabla_\phi J(\phi)$
Update the value network parameter ϕ
$\phi = \phi - \alpha\nabla_\phi J(\phi)$
Until the end of the episode

4.6 Summary

The chapter introduced popular and advance DRL algorithms. Each algorithm has its advantages and drawbacks. DQN can learn from off-policy data, while A3C and PPO require on policy data to learn. All DRL algorithms except DQN can be implemented to the environment, which has a continuous action.A3C and PPO are more powerful in terms of policy update because they guaranteed monotonic policy improvements. Contemporary, their sample efficiency is less.

Chapter 5

Fast Marching Methods

5.1 Introduction

Path planning has been a popular topic in the literature. Many approaches are proposed to fulfil all the criteria that the path planning algorithm requires, such as robustness, reliability, safety, so on. The path planning method can be classified into 5 categories. Firstly, geometric methods are robust algorithms that link the robot's initial and final configuration with minimum rout points. Secondly, graph-based methods use the graph that evaluates the corresponding between the defect illustration. The robot agent sets a novel path while weakening those defects. Graph based method are also categorized as grid-based methods, combinational methods, sampling-based methods. In the grid-based method, space discretizes into a grid cell, and all cells are connected. Moving from one cell to another is represented via cost function, and the shortest path is evaluated with minimum cost. Combinatorial methods are a robust path planning by linking the autonomous planning and arranging of goals and action. The sampling-based method uses the collision avoidance algorithm to search in space for an optimum solution [43,44,45,46].

In this thesis, one of the powerful grid-based path planning algorithms known as the Fast-Marching Square planning method (FM2) [47] is selected. The path estimated by FM2 is considered as an observation of the RL environment. This chapter will be discussed in the state of the art of the FM algorithm and its mathematical formulations. Later, the chapter will explain the improved method called Fast Marching Square planning method (FM2) and Robot Formation Path Planning based on the FM2 method.

5.2 Introduction to Fast Marching Methods

The Fast-Marching method [47] is a level set of the method initially introduced by Osher and Sethian. It is a numerical algorithm that is used for predicting the motion of the front wave. The FM method can be implemented easily in various complex environments since it does not require any significant modification. The FM method has

been used in various field such as medical image processing [48], computer vision [49], chemistry [50], compute of trajectories [51].

5.3 Fast Marching Method

The Fast-Marching Method is a scalable and straightforward method that computes and evaluates the interface's motion by propagating the wave. The interface can be in 2D or 3D. The purpose is the measure the time T, which the wave needs to reach every point. The wave can be propagated from multiple points. However, each point has a unique wave. The Eikonal equation can represent the motion of the front:

$$1 = V(x)|T(x)| \tag{5.1}$$

Where x is the position, V(x) donates the speed in the given position, T(x) is the arrival time function. In the FMM, the wave speed depends on various conditions such as time, position, and interface geometry. However, it cannot be negative [51].

To solve equation (5.1), a discrete solution is proposed in [47]. The Eikonal equation can be e discretized using an approximator to the gradient:

$$\left[\max\left(D_{ij}^{-x}T,0\right)^2 + min(D_{ij}^{+x}T,0)^2 + \max\left(D_{ij}^{-y}T,0\right)^2 + min(D_{ij}^{+y}T,0)^2\right] = 1/F^2 \tag{5.2}$$

Where

$$T(x,y,0) = 0,$$

$$D_{ij}^{-x}T = \frac{T_{ij} - T_{i-1,j}}{x_i - x_{i-1}}, D_{ij}^{+x}T = \frac{T_{i+1,j} - T_{ij}}{x_{i+1} - x_i}, \tag{5.3}$$

$$D_{ij}^{-y}T = \frac{T_{ij} - T_{i,j-1}}{y_i - y_{j-1}}, D_{ij}^{+y}T = \frac{T_{i,j+1} - T_{ij}}{y_{j+1} - y_j} \tag{5.4}$$

A simpler version of the equation is given by:

$$\max\left(\frac{T_{i,j} - \min\left(T_{i-1,j}, T_{i+1,j}\right)}{\Delta x}, 0\right) + \max\left(\frac{T_{i,j} - \min\left(T_{i,j-1}, T_{i,j+1}\right)}{\Delta y}, 0\right) = \frac{1}{V_{i,j}^2} \tag{5.5}$$

Where i donates row, j donates column of the 2d grid map. Δx and Δy are the grid spacing for the directions of x and y, respectively. For simplicity, the equation is modified:

$$max\left(\frac{T - T_1}{\Delta x}, 0\right)^2 + max\left(\frac{T - T_2}{\Delta y}, 0\right)^2 = \frac{1}{V_{i,j}^2} \tag{5.6}$$

$$\begin{aligned} T &= T_{i,j} \\ T_1 &= \min\left(T_{i-1,j}, T_{i+1,j}\right) \\ T_2 &= \min\left(T_{i,j-1}, T_{i,j+1}\right) \end{aligned} \tag{5.7}$$

When the T < T1, the equation (5.6) reduces and is given by:

$$\left(\frac{T - T_1}{\Delta x}\right) = \frac{1}{V_{i,j}} \tag{5.8}$$

If T <T2, the equation (5.6) reduces and is given by:

$$\left(\frac{T - T_2}{\Delta y}\right) = \frac{1}{V_{i,j}} \tag{5.9}$$

5.3.1 Implementation of the Fast-Marching Method

Equation (5.5) can be solved iteratively over a grid map. The grid map cells must be identified as one of the following categories.

Unknown: Cells whose T value has not yet been determined (the wavefront has not reached the cell).

Narrow Band: Cells that should be on the front wave in the next iteration. They are given a T value, which may vary in future iterations of the algorithm.

Frozen: Cells that the wave has already passed over hence, their T value is fixed.

The algorithm is divided into three phases: initialization, main loop, and finalization.

Initialization:

The algorithm begins by setting T to 0 in the cell or cells from whence the wave originates. These cells are identified as frozen. Following that, it identifies all of their Manhattan neighbors as a narrow band and computes T for each of them.

Main loop:

The algorithm will solve the Eikonal equation (5.5) for the Manhattan neighbors (that are not yet frozen) of the narrow band cell with the lowest T value in each iteration. This cell is then identified as frozen. The narrow band keeps an ordered list of its cells. Cells are sorted according to their T value, from lowest to highest.

Finalization:

The process is completed when all of the cells are frozen. The process can be seen in figure 5.1 and figure 5.2.

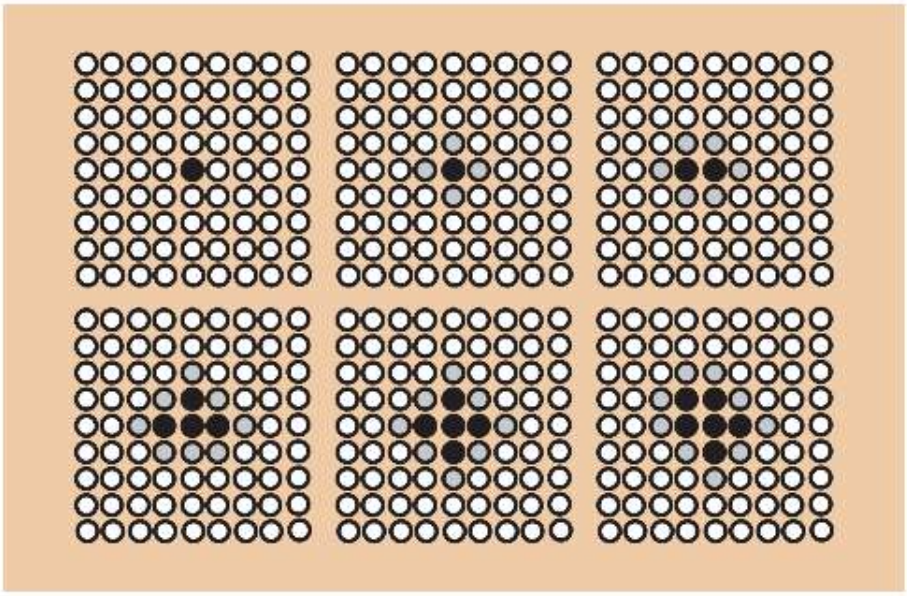


Figure 5. 1: Iterations of the FMM with one wave sources [52]

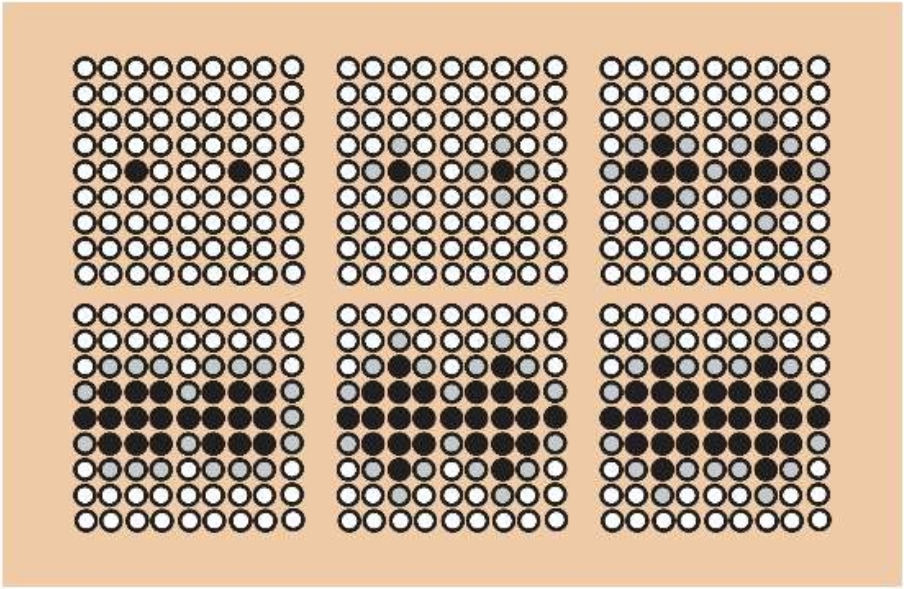


Figure 5. 2**:** Iterations of the FMM with two wave sources [52]

In Figure 5.1, iterative wave expansion with one source point is shown. Figure 5.2 illustrates iterative wave expansion with two source points. T value of the black points will not change, and they labeled as frozen. Gray points are the narrow band, whereas white ones are unknown. Figure 5.1 clearly shows how the waves expand concentrically to the source while the waves develop independently and expand together. The cells with lower T are expanded first in the iterative process [52].

5.3.2 Path planning with the Fast-Marching Method

Path planning problem can be solved using the Fast-Marching Method and plenty of path planning algorithm detailed in the introduction of chapter 5.

The configuration space χ correlates to the FMM's domain χ so they both named the same. χ_{obs} is a subset of X that depicts the points in space where the wave cannot propagate. The remaining cells are named as χ_{free}.

It assumes that $\chi_{goal} \subset \chi_{free}$ represents at least one cell for the well-behaved path planning. Similarly, start $\chi_s \subset \chi_{free}$ is represented by at least one cell. To find the path, the wave is propagated from χ_s to χ_{goal}, calculating a time-of-arrival map $\mathcal{T}$. Using gradient descent from χ_{goal} over $\mathcal{T}$, Gradient descent will calculate the path to the unique minimum of $\mathcal{T}$, resulting in $\sigma(0)$ = Xs. The computed path returns inverted because waypoints travel from χ_{goal} to χ_s.

The cost function, in terms of optimality, is defined as c = $\mathcal{T}$. That is, the arrival time of each cell indicates its cost from the initial point. FMM assures that the gradient descent path is optimum, which means every cell has the lowest possible value $\mathcal{T}_i$assigned, and there is no better option to reach that cell. The Sobel operator is used to determine the highest gradient direction over the grid map:

$$grad_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \times \mathcal{T} \quad grad_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \times \mathcal{T} \tag{5.10}$$

The path is computed iteratively. $grad_x$ and $grad_y$ are calculated in every pi until pi reaches the minimum. As a result, FMM extracted the path include a large number of points, which is vital for implementing path following on an actual robot application.

$$\begin{aligned} mod_i &= \sqrt{grad_{ix}^2 + grad_{iy}^2} \\ alpha_i &= \arctan\left(\frac{grad_{iy}}{grad_{ix}}\right) \\ p(i+1)x &= p_{ix} + step.\cos(alpha_i) \\ p(i+1)y &= p_{iy} + step.\sin(alpha_i) \end{aligned} \tag{5.11}$$

5.4 Fast Marching Square planning method (FM2)

The fast-Marching method generates trajectories. However, the method does not guarantee smoothness and sufficient safety distance to the obstacles. Lack of smoothness may be challenging for the mobile robot. The FM2 method [53] proposed to provide a more safe path planning by applying twice the FMM.

Let us assume the 3D environment as input Wo, is a binary map. The 3D environment can be seen in figure 5.3. The buildings are represented as obstacles, and the rest is free space.

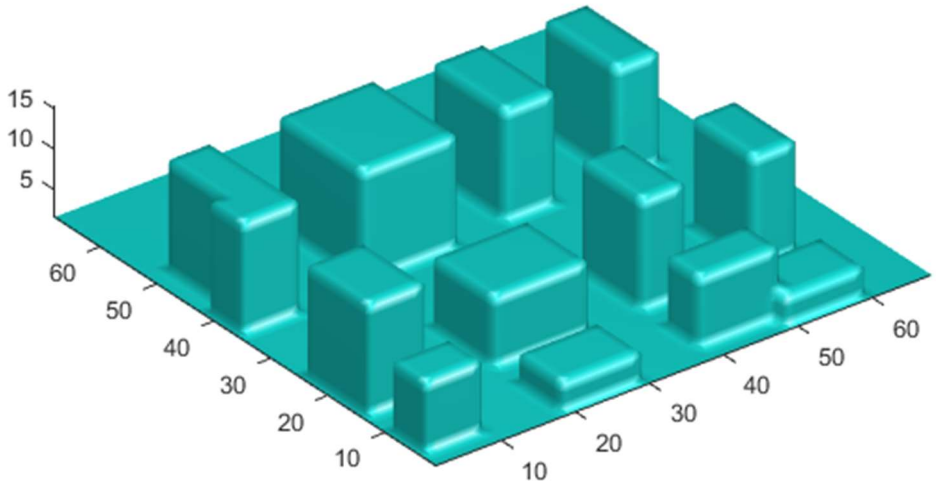


Figure 5. 3: 3D Environment

The FM method applied to binary map Wo. Instead of having one wave in the target point, all obstacles are considered as a wave source. It allows expanding several waves at the same time. The potential field of the original map W is obtained due to applying the wave expansion. W is known as a velocity map, representing the maximum allowed speed for the robot at each point. The speed is less when the robot is close to obstacles. Contrary, the speed is greater when the robot is far away from the obstacles.

The path is calculated by implementing the FM algorithm again to the target point, considering this point as a wave source. The wave is expanding from the target point until it reaches the starting point. The resulting maps donate the time of arrival map T. The gradient descent method is applied over T to provide a fast and smooth path.

The FM2 method can be implemented as both a local and global planning method since it does not require a high computational cost. In both cases, the method can provide a safe and smooth path. The FM2 is sufficient to provide a path and is used as a path planner for the UAV in the RL environment. The next section will discuss the implementation of the FM2 to the robot path planning.

5.4.1 Robot Path Planning Based on the FM2 method

The robot path planning aims to obtain the robot's path and position, considering the environment's characteristics and the final goal. As stated before, the FM method is applied twice. The first potential map W is created as a result of implementation. Then, the FM method applied again over W, resulting in second potential map D. The method is explained below step by step:

1) The environment map Wo is created where the value 0 represents obstacles, and the value 1 means free spaces.

2) The FMM is applied to Wo, and the first potential map W is obtained.

3) The second potential map D is created by applying the FMM to W.

4) Applying the gradient descent on the second potential map, D obtains the robot path.

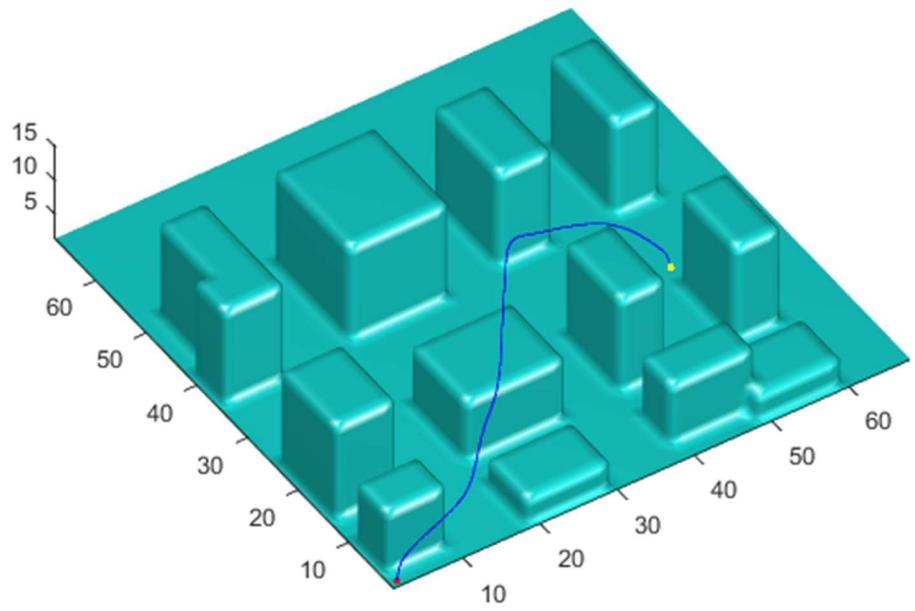


Figure 5. 4: Observed path using the FM2 method.

In figure 5.4, The red point represents the initial point, and the yellow point represents the endpoint. The path is smooth enough. The points are provided in the path are not close to the obstacles. Therefore, the observed path guarantees a safe way to reach the endpoint.

5.5 Summary

In this chapter, the FM method's mathematical formulation is explained, and the algorithm used for robot path planning is presented. It can be concluded that the FMM provides a path that may close to the obstacles. It may cause unsafe movement for the robot. As an improvement of FMM, the FM2 methods are discussed. It obtains two potential maps by expanding two-wave. First wave expansion calculates the velocity map, which identifies the highest permissible speed at each point. The second expansion is used to forming the target point and initial point. The robot path is obtained using gradient descent.

Chapter 6

Methods for Maneuvers to avoid collision

6.1 Introduction

In chapter 5, many deep reinforcement learning algorithms introduced, and their cons and pros are explained. DDQN algorithm shows good performance on high dimensional tasks. The method is model-free, requires no knowledge of the environment, and can learn from off-policy data.

This chapter examines the DDQN algorithm implementation in a simulated environment for manoeuvres to avoid nearby UAV. The findings are compared DRL algorithms explained in chapter 4, and the performance is discussed.

6.2 Proposed Environments

The environment is comprised of several aspects that influence the Reinforcement Learning Agent. Selecting an appropriate method for the agent to interact with the certain environment will be efficient. These environments can be 2D or 3D grids.
Important features of environments are [54]:

- Deterministic
- Observable
- Discrete or continuous
- Single or multiagent.

Proposed environment 3D city illustration which includes number building as an obstacles and open wide area. The size of the 3D map is 70 x 70 x20. The environment is representative since it contains high dimensional observation. 3D city illustration is shown figure 6.1:

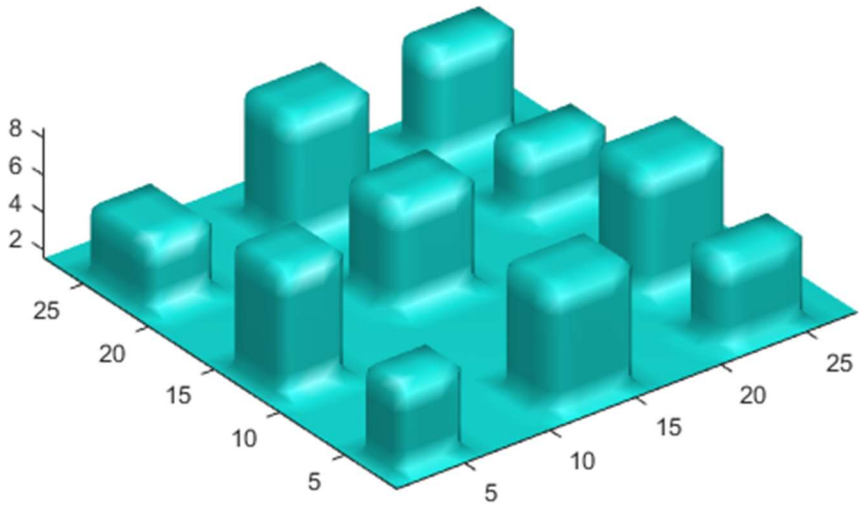


Figure 6. 1: 3D city illustration

There are two UAV in the environment called UAV master and UAV slave. The master UAV has a fixed path while the slave UAV requires to adjust its path in order to avoid collision with the master UAV path. The authors split the tasks for benchmarking continuous control [55] into four categories: basic tasks, locomotion tasks, partially observable tasks, and hierarchical tasks. The proposed environment consists of Locomotion Tasks and Hierarchical Tasks. The goal for the Locomotion Tasks is to avoid collision with the master UAV path as much as possible. Furthermore, a significant amount of exploration is required to learn to reach without getting stuck at local optima. Many real-world applications have a hierarchical structure, where higher-level decisions reuse lower-level capabilities [56]. The slave UAV can reuse the Locomotion Tasks when exploring the environment. The agent needs to learn how to reach the goal point via its action, while it needs to avoid collision with master path and environment obstacles.

6.2.1 State Space

To do the task effectively, the RL-agent must obtain all essential information regarding the current state of the environment. The raw data that has been determined as relevant is provided in the following:

Pose of the slave UAV: The vector that consist of the waypoint of the slave UAV. Each waypoint position can be written in Euclidean space with the (x, y, z)-coordinate. The state-space of the agent can be written as follow:

$$s = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{6.1}$$

$\dot{x}$ represents the relative positon along the x axis. Similarly, $\dot{y}$ and $\dot{z}$ represents the relative position along the y and z axis. At the start of every episode, the slave UAV is initialized in the random position as a starting point. Described relevant information may be supplied into a Neural Network in a variety of ways. In this thesis, we feed the raw data (position of the slave UAV) directly.

6.2.2 Action Space

Discrete action space has been proposed in this thesis. Only one plane is used for avoidance in order to reduce complexity. A vertical plane for up-down movements and a horizontal plane for the left, right movements. Horizontal movement is efficient in a wide range of areas in terms of energy consumption, while vertical movement is the proper way to avoid collision between buildings. The collision incident appears mostly between the buildings, so vertical movement up /down is used as an action. The action space allows ten discrete actions, which is the angular velocity (θ) in degree.

$$\theta = [-45, -25, 0, 25, 45] \tag{6.2}$$

Negative angular speeds stand for the movement direction. If the agent chooses positive angular velocity, the UAV moves up along the positive z-axis. If the agent chooses negative angular velocity, the UAV moves down along the negative z-axis. The linear velocity (ϑ) is set to constant because of the problem complexity.The simplified kinematic model of the UAV is given.

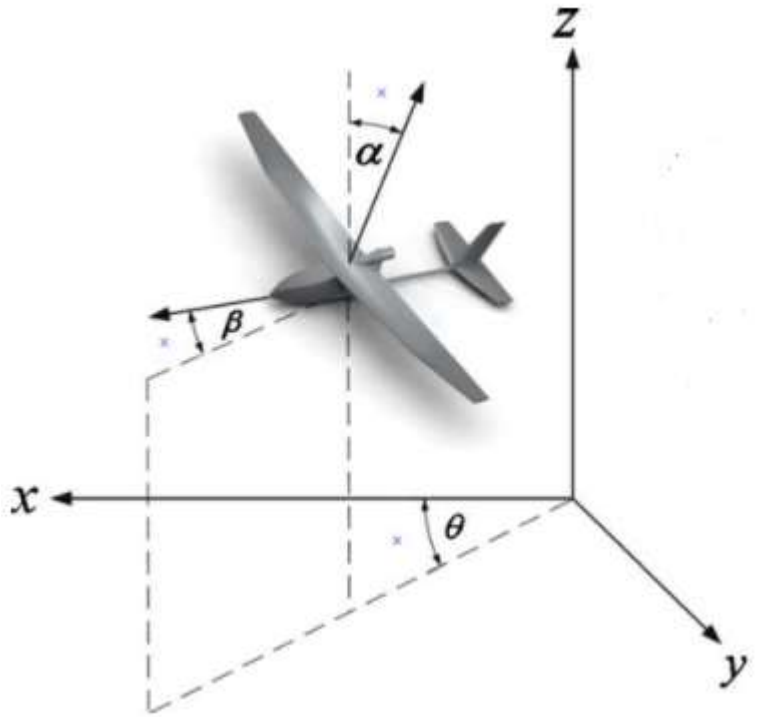


Figure 6. 2 : The simplified kinematic model of the UAV

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \\ \dot{\beta} \\ \dot{\alpha} \end{bmatrix} = v^1 \begin{bmatrix} \cos\theta\cos\beta \\ \sin\theta\cos\beta \\ \beta \\ 0 \\ 0 \\ 0 \end{bmatrix} + v^2 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + v^3 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + v^4 \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{6.3}$$

$$\dot{z} = \vartheta \times \sin\theta \tag{6.4}$$

The agent can independently choose the action θ for the action. The movement is implemented to the path using equation (6.4) The action of the agent is described following:

Up θ=[45,25]
Down θ=[−45,25]
Do not move θ=[0]

6.2.3 Reward Function

The design of reward functions is a vital part of any RL process. The idea is to create an incentive for the agent to learn specific behavioural patterns. In this thesis, reaching the goal point is objectively desirable, but this goal must be achieved without the case of collision. Moving far away from the goal point would be ineffective while the agent avoids the collision with the master path. As a result, a reward function that incorporates these critical features of UAV motion control is developed. the reward function used in this thesis consists of the following three components:

- Reward represents bonus when the UAV reaches the goal point: r_{goal}
- Reward represents the penalty when the UAV collide with obstacles or out of environment space $r_{collison}$
- Reward represents bonus that given each step of the agent r_s

In many applications, giving a mixed reward signal with a continuous and discrete reward signal is advantageous. The continuous reward signal can promote convergence by giving a smooth reward near goal states, while the discrete reward signal can be deployed to force the system away from bad states. The equation for the total reward function is given as below:

$$r_t = r_{goal} + r_{collison} + r_s \tag{6.5}$$

The agent receives a reward after each step and the reward function is defined as:

$$r_s = 1 - \left(\frac{d_g}{d_{gmax}}\right)^{0.4} \tag{6.6}$$

where d_g is the distance from the UAV to its goal position, d_{gmax} is set to be the diagonal distance of the map. In our scenario, if the agent collides or reaches out of map limit, the episode ends with a penalty of $r_{collison} = -100$. If the agent reached the goal point, the episode ends with a positive reward of $r_{goal} = 150$.
The total reward for each episode is calculated using equation below:

$$r = \begin{cases} r_{goal} & if\ the\ agent\ reaches\ the\ goal \\ r_{collison} & if\ there\ is\ a\ collision\ or\ moving\ out\ of\ the\ map \\ r_s & otherwise \end{cases} \quad (6.7)$$

If a critical scenario involves collision and moving out of the map, the current episode will end.

6.3 Learning Parameters

Environment State space is used as an input of the DDQN algorithm. The policy's output is the angular velocity that can be applied to kinematic equation of the UAV. Result of those equations generates the position for the UAV. Neural Network Architecture is shown in figure 6.3:

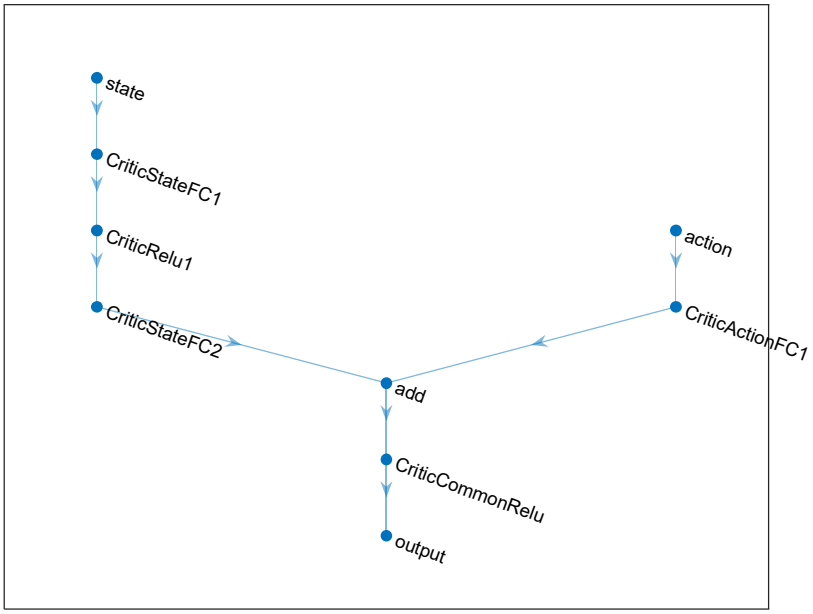


Figure 6. 3: Neural network architecture of the algorithm

The state path initialized with one fully connected layer 84 and 32 hidden neurons, respectively. The ReluLayer is implemented between those two layers. The action path consists of one fully connected layer with 32 hidden neurons. Action and State paths are connected with one common ReluLAyer. The output layer is a fully connected layer that has one output for each valid action.

6.4 Training Parameters

Epsilon-greedy exploration is used to select the appropriate action in the training. This method allows the agent to choose a random action with probability ε. Otherwise, the robot will choose the action from the neural network with the highest value. A greater Epsilon value indicates that the agent explores the action space at a higher rate. In the exploration stage, ε is set to 1, and the UAV always chooses a random action to obtain different training data for updating the neural network. However, the UAV also need to exploit the actions so, it cannot always choose a random action. ε is uptaded using following formula:

$$\varepsilon = \varepsilon \times (1 - \beta) \tag{6.8}$$

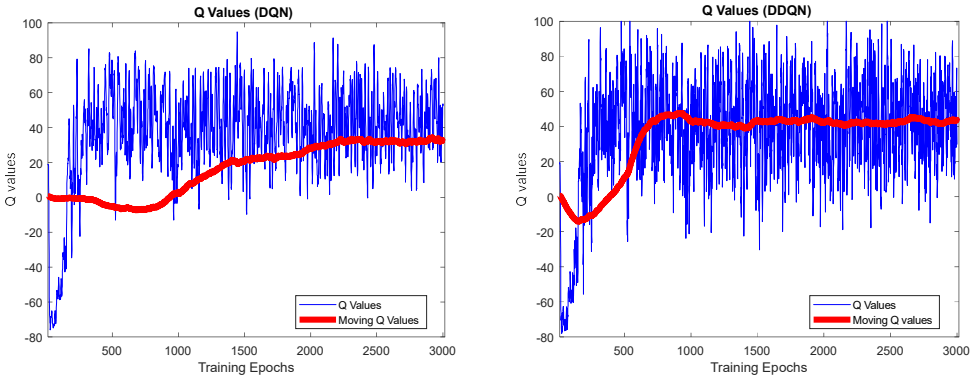
Where β stands for epsion decay. If ε is lower than ε_{min}, ε, ε is updated using equation (6.8) at the end of each episode. A lower epsilon decay indicates that the robot chooses less random actions while the agent rely on the prior experiences it has already obtained. When the ε reaches the ε_{min} , it means that the agent will choose the random action with probability of % $\%(100 \times \varepsilon_{min})$ [57].
The rest of training parameters can be seen in the table 4.1:

Hyperparameter	Value
Training Episodes	3000
Target Smooth Factor	0.001
'Experience Buffer Length	1000000
Discount Factor	0.95
Mini Batch Size	32
Target Update Frequency	1

Table 4. 1 DQN hyperparameters used for the experiments

6.5 Result and Discussion

Experimental performance evaluation of DQN and Double DQN has been performed in paper [58]. Double DQN method provide remarkably performance on the overestimation problem. Figures 6.4 and 6.5 demonstrate an analysis of the average Q-values and cumulative reward.



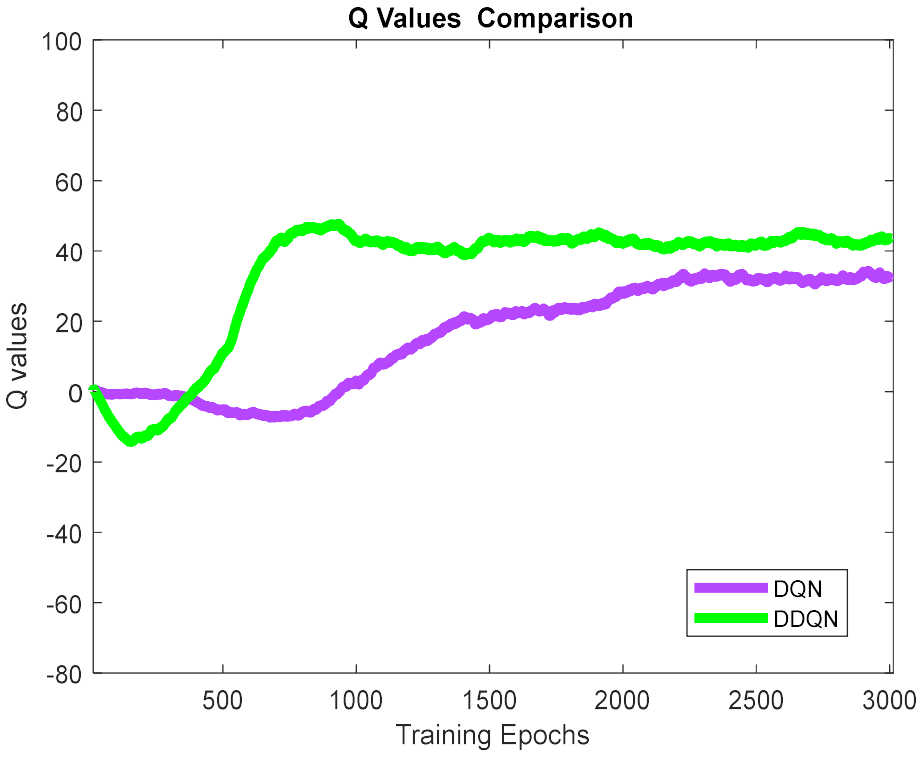


Figure 6. 4: Q values calculated by DQN and DDQN.

Q values are compared in the figure 6.4. The learning process of DDQN is more stable than that in DQN. İn addition, The Q values of DQN are higher at the beginning of the training and later drops because of the overestimation. The average reward for both methods is analyzed to determine whether DDQN is the acceptable proposed approach. The average Reward comparison is shown in figure 6.4. The result illustrates that DDQN learn faster than DQN and also reached higher reward in the end. Besides, DQN provide more stable learning experience. As a result, in the evaluation of the training, we implement collision avoidance method designed based on advanced DDQN model with fixed Q target and experience replay.

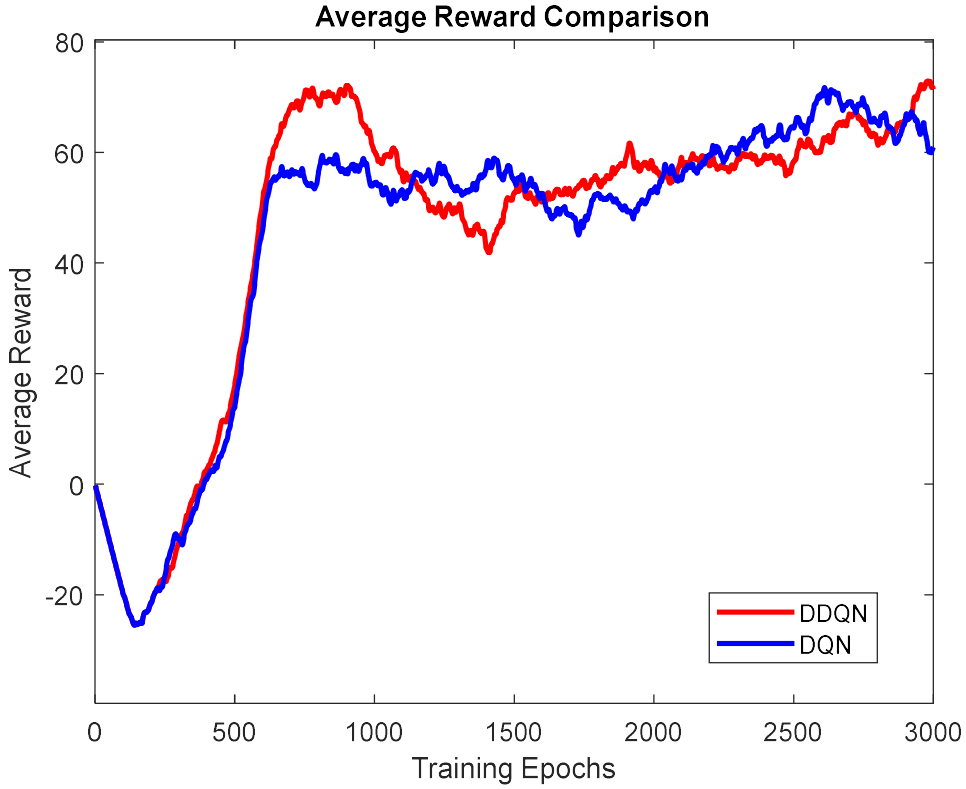


Figure 6. 5: Average Reward comparison between DQN and DDQN

In this thesis, three training has been done with different parameters of β 0.0050, 0.0055, 0.0057, respectively. ε_{min} is set to 0.05. If $\beta = 0.999$, ε reaches the

minimum at $\log_{0.999}^{0.05} = 2994$ episodes. For this reason, the agent is trained for 3000 episodes in every experiments.

Figure 6.6 shows the training results for comparing the agent performance with different decay rates.

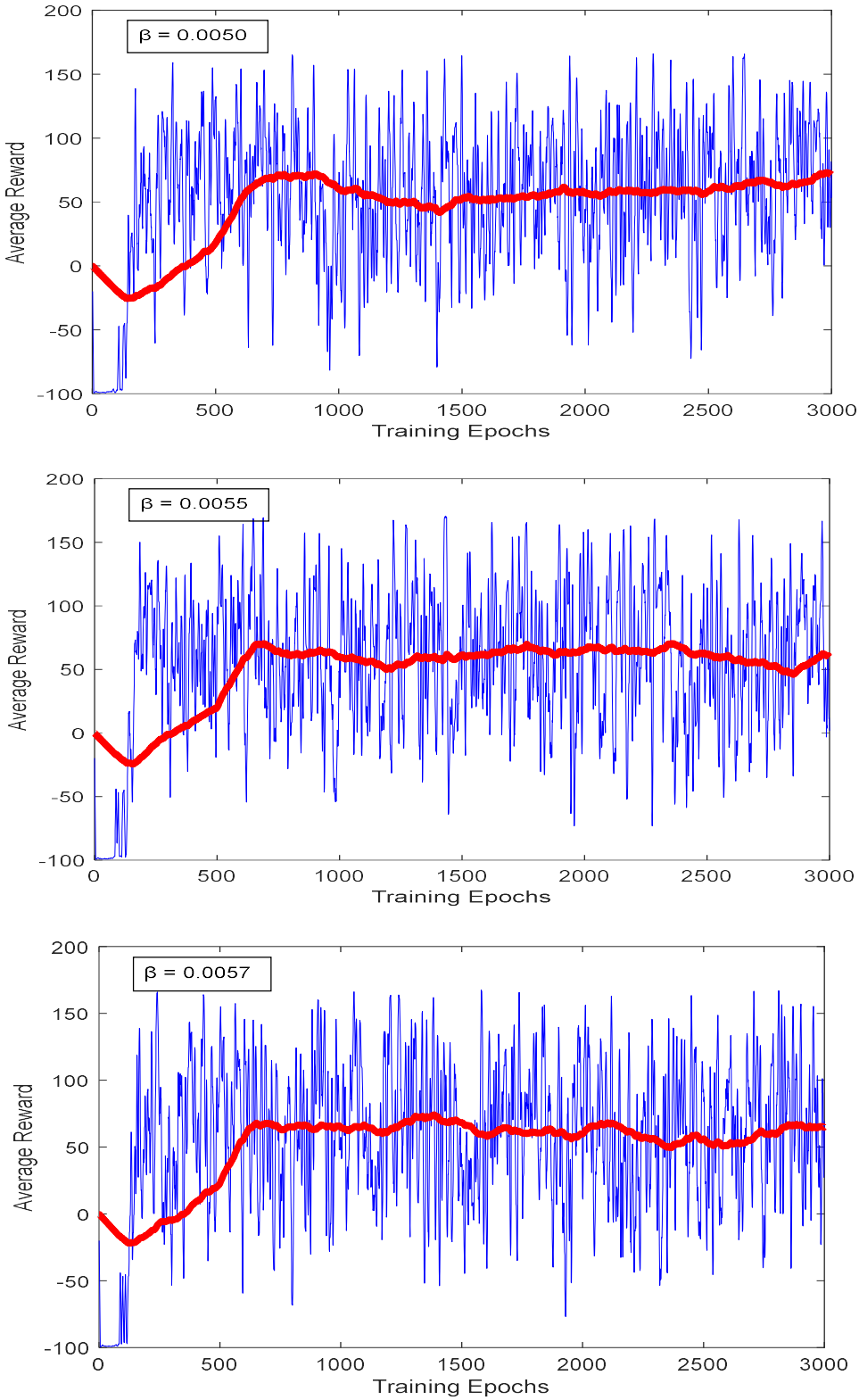


Figure 6. 6: Comparisons of training performance with varied decay rates

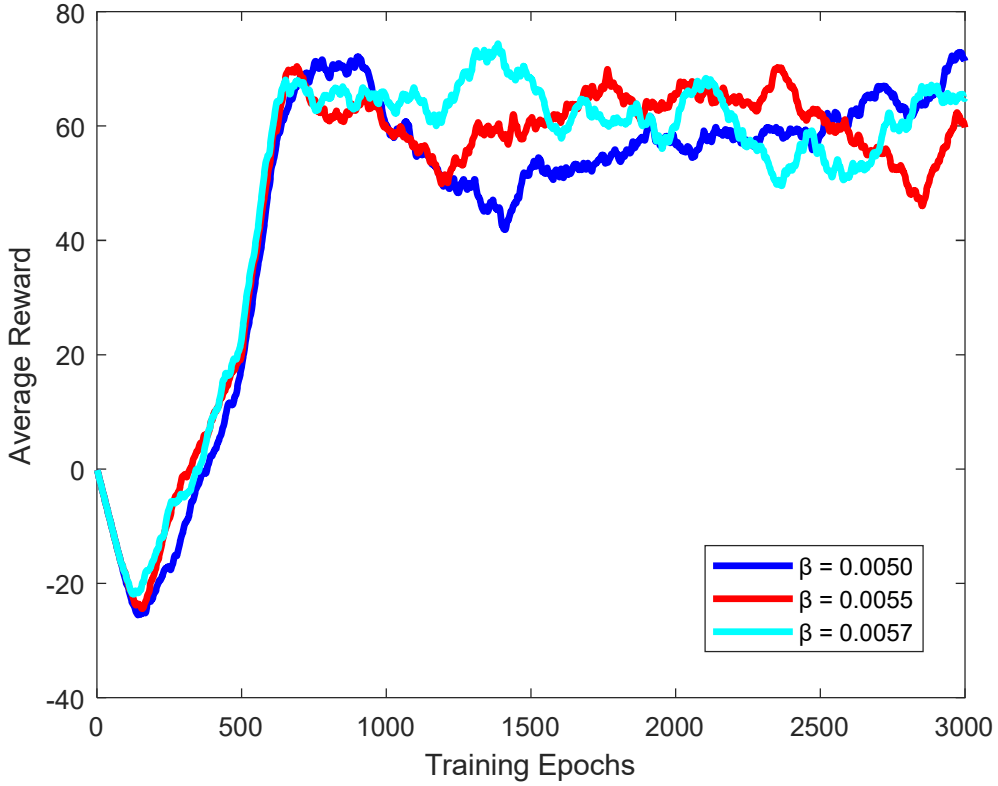


Figure 6. 7: The agent average reward comparison with different decay rate

The agent obtains a non-stable reward with the low value of β.the reason is that the agent chooses a high probability action with the highest Q value. Initially, a larger value of β led the robot to explore more action. İt means the agent tries multiple behaviors when faced with the same situation to decide which one was better.

Figure 6.7 shows that the learning process with the highest values of β is less stable than the others. It can be concluded that the lowest value of the β dropped in the middle of the processes. the agent discovers the environment poorly, but it reaches the maximum reward at the end of the episode.

The initial point is gathered randomly in order to test the agent's behavior. The test points are all different from the points used for training the agent. Test initial points fed to the agent neural network, and the validation result is shown in figure 6.8:

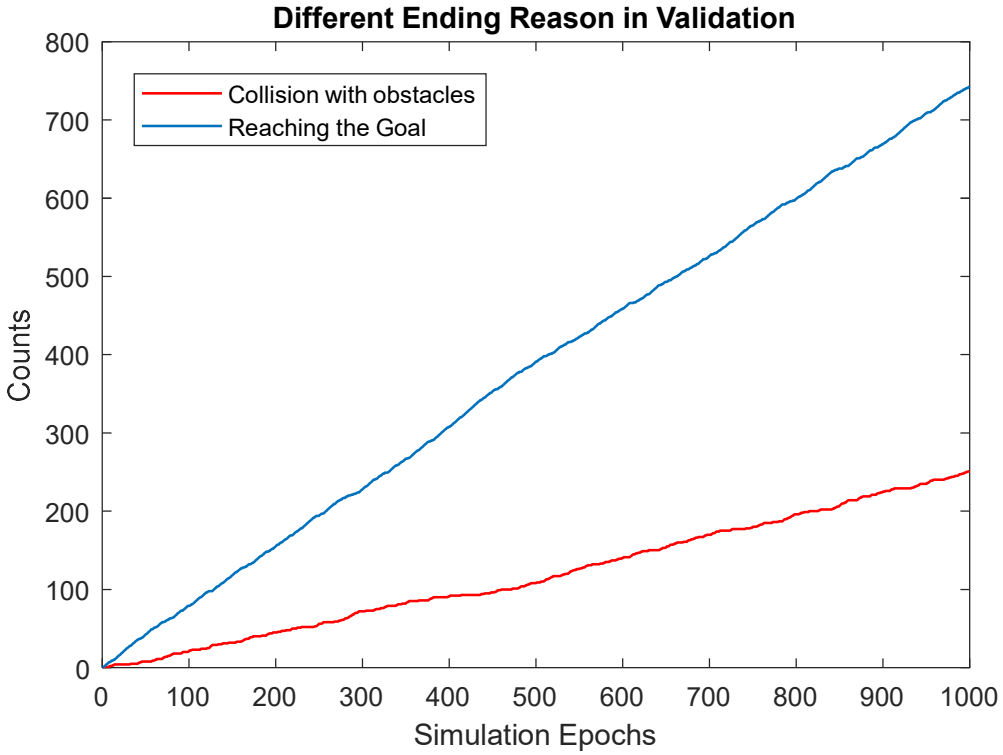


Figure 6. 8:The number of the different ending in the simulation

Figure 6.8 displays how the total number of alternative ending reasons changes throughout simulation processes. During the early stage of the simulation, Most of the episodes ending with reaching the goal point. In the middle of the simulation, the increase in the collision with obstacles is lower than that in reaching the goal. At the end of the simulation process, the trained agent reaches the goal point in 749 episodes and collide with the obstacles in 251 episodes. These results mean that the agent learns to avoid the master UAV path as an obstacle and reach its goal point.

6.5.1 Experiments

The goal of training allows the UAV to learn optimal strategy in reaching the goal point and avoid the collision with a nearby UAV path. İn this model, the slave UAV is only learner/agent while the master UAV is part of the environment. the specified goal was achieved gradually by increasing the problem complexity. The following identified problem complexity:

1)Deciding the which direction to go
2)Keep the safety distance to building and master UAV path as an obstacle.
3)Reach the goal point safely.

The agent can solve all the problems explained above. the qualitative testing, three different scenarios are set up to examine different behavior of the agent. Firstly, the slave UAV faces the collision with the master bath in the beginning. Secondly, the test involves collision in the middle of the path. Lastly, the Slave UAV collides with the master UAV to close the goal point. The sample path of the completed collision avoidance task with different initial points is showed in figure 6.9 and figure 6.10.

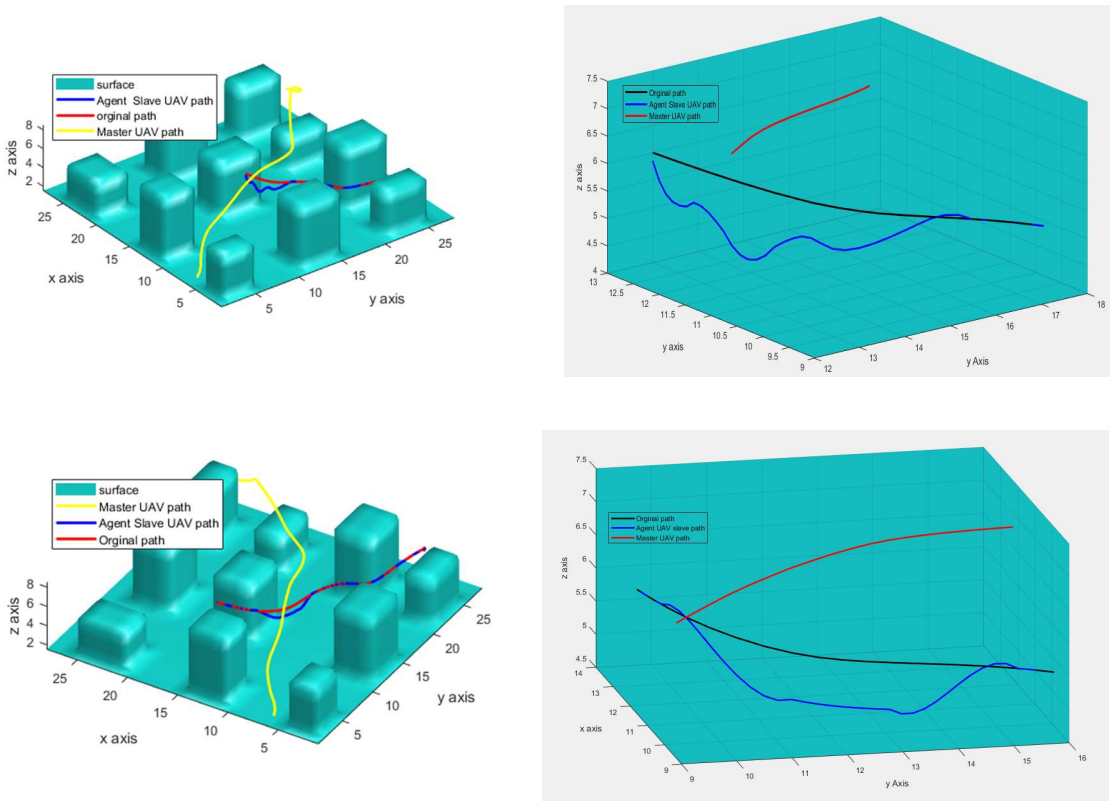


Figure 6. 9: Different scenarios in 3D city draw and their paths in close scale

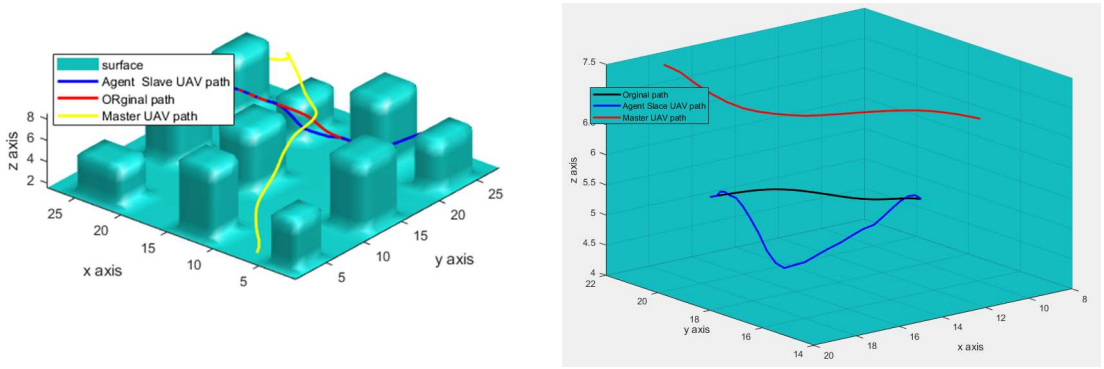


Figure 6. 10: Scenario including Collision which closes the goal point in 3D city and its close scale.

Experiments show that the slave UAV is able to learn which direction to go before achieving the main task. The risky situation is detected because the distance between the slave UAV path and Master UAV path in specific points is lower than the threshold. In our case, the threshold distance is set to 2 cell sizes. When the risky situation is detected, the agent does action, which resulted in moving away from the risky point. The agent also does not move so far away from the original path because the main goal is to reach the goal point. After the risky situation was handled, the result shows that the slave UAV is getting closer to the original path to achieve the task.

6.6 Summary

The chapter explained the training and experimental result of implementing the DDQN for the UAV simulation environment. DDQN method is solved the collision avoidance problem for the specialized discrete action.no hand-coded controller is needed since the DDQN method learns the policy for the whole state space.

Regardless of performance, the primary disadvantage of implementing the DDQN method is the quantity of the samples. Such a large number of samples would be hard to acquire in the real world.

Chapter 7

Conclusions and Future Work

In this chapter, the main conclusions of the whole document are extracted, and future work is explained.

7.1 Conclusions

Deep Reinforcement Learning is the future of Reinforcement Learning. It has demonstrated enormous performance on decision-making control to avoid collision on 3D environments from starch. However, the application of DRL to real-world UAVs is currently restricted. In this thesis, we proposed a collision avoidance with path planner strategy based on an observed date learned by RL in this paper. This controller can avoid obstacles in changing environment. Furthermore, it does not need hand-engineered components because it generates action commands straight from raw input data. A simulated environment is created virtually that UAV can learn how to avoid the nearby UAVs.

The training of the agent is difficult. We used the techniques to assure successful and time-efficient training. The neural network approach to the RL learning g method is used to achieve sufficient results. In the ablation research, these approaches have been shown to be effective for collision avoidance and significantly expedite convergence.

In chapter 5, The Fast-Marching Method is extremely adaptable, not only for path planning but also for more complicated applications that need a path planning step. We have explained how to use the Fast-Marching Square technique in Path planning for UAVs. Thanks to method compatibleness, Fast Marching Square can be implemented regardless of the problem's number of dimensions because the Fast-Marching Method is specified for any number of dimensions.

The primary goal of using DDQN for UAV was to determine if a policy could be developed to control UAV over the whole action space. In Chapter 6, it was demonstrated that DDQN could learn a policy for the proposed task of UAV. We deployed the optimal policy and tested it in a different scenario. We successfully proved in the experiment that our policy can create a collision-free route to the goal point.

7.2 Future Work

The one potential future work might be to use a different RL agent, which is performed better on the proposed task. The type of action has an impact on the policy performance. We can design continuous action that the agent explores more and result in a high number of maneuverers.

The model used to learn the policy is DRL that does not give any confidence limits for its predictions. It would be beneficial to expand the model with Bayesian DNN, which may offer specific uncertainty bounds on prediction. This uncertainty can be reached by adding the extra cost to the reward function while the policy's ability to explore such areas of the state space is limited.

The thesis used simulation to test the efficiency of the proposed approach with three different scenarios. Therefore, the next step would be to try a different scenario since three scenarios are typically insufficient to evaluate some aspect of the method, such as the number of updates necessary for a particular issue. More experimentation allows us to find out in which situation this technique works best. This may also provide a general estimation of the number of updates necessary for a particular problem.

References

[1] A. Sharma *et al.*, "Communication and networking technologies for UAVs: A survey," *J. Netw. Comput. Appl.*, vol. 168, no. 102739, p. 102739, 2020.

[2] V. González, C. A. Monje, L. Moreno, and C. Balaguer, "UAVs mission planning with flight level constraint using Fast Marching Square Method," *Rob. Auton. Syst.*, vol. 94, pp. 162–171, 2017.

[3] Q. Cheng, X. Wang, J. Yang, and L. Shen, "Automated enemy avoidance of unmanned aerial vehicles based on reinforcement learning," *Appl. Sci. (Basel)*, vol. 9, no. 4, p. 669, 2019.

[4] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] V. D. Berdonosov, A. A. Zivotova, Z. Htet Naing, and D. O. Zhuravlev, "Speed approach for UAV collision avoidance," *J. Phys. Conf. Ser.*, vol. 1015, p. 052002, 2018.

[6] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[7] L. Xie, S. Wang, A. Markham, and N. Trigoni, "Towards monocular vision based obstacle avoidance through deep reinforcement learning," *arXiv [cs.RO]*, 2017.

[8] P. Long, T. Fanl, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[9] J. Roghair, K. Ko, A. E. N. Asli, and A. Jannesari, "A vision based Deep reinforcement learning algorithm for UAV obstacle avoidance," *arXiv [cs.AI]*, 2021.

[10] Y. Zhao, J. Guo, C. Bai, and H. Zheng, "Reinforcement learning-based collision avoidance guidance algorithm for fixed-wing UAVs," *Complexity*, vol. 2021, pp. 1–12, 2021.

[11] S.-Y. Shin, Y.-W. Kang, and Y.-G. Kim, "Obstacle avoidance drone by deep reinforcement learning and its racing with human pilot," *Appl. Sci. (Basel)*, vol. 9, no. 24, p. 5571, 2019.

[12] S. Ravichandiran, *Hands-On Reinforcement Learning with Python: Master reinforcement and deep reinforcement learning using OpenAI Gym and TensorFlow*. Birmingham, England: Packt Publishing, 2018.

[13] A. Palmas *et al.*, *The The Reinforcement Learning Workshop: Learn how to apply cutting-edge reinforcement learning algorithms to a wide range of control problems*. Birmingham, England: Packt Publishing, 2020.

[14] R. S. Sutton and A. G. Barto, Eds., "Reinforcement Learning: An Introduction Second edition," 2017.

[15] S. Saito, Y. Wenzhuo, and R. Shanmugamani, *Python Reinforcement Learning Projects: Eight hands-on projects exploring reinforcement learning algorithms using TensorFlow*. Birmingham, England: Packt Publishing, 2018.

[16] C. You, J. Lu, D. Filev, and P. Tsiotras, "Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning," *Rob. Auton. Syst.*, vol. 114, pp. 1–18, 2019.

[17] M. P. Deisenroth, G. Neumann, and J. Peters, *A survey on policy search for robotics*. Hanover, MD: now, 2013.

[18] J. Kober and J. Peters, "Policy search for motor primitives in robotics," *Mach. Learn.*, vol. 84, no. 1–2, pp. 171–203, 2011.

[19] L. Deng, G. Hinton, and B. Kingsbury, "New types of deep neural network learning for speech recognition and related applications: an overview," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013.

[20] D. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012.

[21] M. Lodeiro-Santiago, P. Caballero-Gil, R. Aguasca-Colomo, and C. Caballero-Gil, "Secure UAV-based system to detect small boats using neural networks," *Complexity*, vol. 2019, pp. 1–11, 2019.

[22] R. P. Padhy, S. Verma, S. Ahmad, S. K. Choudhury, and P. K. Sa, "Deep neural network for autonomous UAV navigation in indoor corridor environments," *Procedia Comput. Sci.*, vol. 133, pp. 643–650, 2018.

[23] M. Bojarski *et al.*, "End to end learning for self-driving cars," *arXiv [cs.CV]*, 2016.

[24] J. Dawani, *Hands-On Mathematics for Deep Learning: Build a solid mathematical foundation for training efficient deep neural networks*. Birmingham, England: Packt Publishing, 2020.

[25] U. Michelucci, *Applied deep learning: A case-based approach to understanding deep neural networks*, 1st ed. Berlin, Germany: APress, 2018.

[26] Y. Tan, Y. Shi, and B. Niu, Eds., *Advances in swarm intelligence: 10Th international conference, ICSI 2019, Chiang Mai, Thailand, July 26-30, 2019, proceedings, part II*. Cham, Switzerland: Springer Nature, 2019.

[27] L. V. Fausett, *Fundamentals of neural networks: Architectures, algorithms and applications: United States edition*. Upper Saddle River, NJ: Pearson, 1993.

[28] C. Grosan and A. Abraham, *Intelligent Systems: A Modern Approach*, 2011th ed. Berlin, Germany: Springer, 2013.

[29] D. Sarkar, R. Bali, and T. Ghosh, *Hands-On Transfer Learning with Python: Implement advanced deep learning and neural network models using TensorFlow and Keras*. Birmingham, England: Packt Publishing, 2018.

[30] K. Boonyuen, P. Kaewprapha, U. Weesakul, and P. Srivihok, "Convolutional neural network inception-v3: A machine learning approach for leveling short-range rainfall forecast model from satellite image," in *Lecture Notes in Computer Science*, Cham: Springer International Publishing, 2019, pp. 105–115.

[31] N. Dey, A. S. Ashour, S. J. Fong, and S. Borra, Eds., *U-Healthcare Monitoring Systems: Volume 1: Volume 1: Design and Applications*. San Diego, CA: Academic Press, 2018.

[32] P. Kim, *MATLAB deep learning: With machine learning, neural networks and artificial intelligence*, 1st ed. New York, NY: APRESS, 2017.

[33] V. Mnih *et al.*, "Playing Atari with deep reinforcement learning," *arXiv [cs.LG]*, 2013.

[34] F. R. Yu and Y. He, *Deep reinforcement learning for wireless networks*, 1st ed. Cham, Switzerland: Springer Nature, 2019.

[35] S. Ravichandiran, *Deep Reinforcement Learning with Python: Master classic RL, deep RL, distributional RL, inverse RL, and more with OpenAI Gym and TensorFlow, 2nd Edition*, 2nd ed. Birmingham, England: Packt Publishing, 2020.

[36] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more, 2nd Edition*, 2nd ed. Birmingham, England: Packt Publishing, 2020.

[37] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with Double Q-learning," *arXiv [cs.LG]*, 2015.

[38] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," *arXiv [cs.LG]*, 2015.

[39] V. Mnih *et al.*, “Asynchronous methods for deep reinforcement learning,” *arXiv [cs.LG]*, 2016.

[40] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, “Trust Region Policy Optimization,” *arXiv [cs.LG]*, 2015.

[41] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” *arXiv [cs.LG]*, 2017.

[42] L. Graesser and W. L. Keng, *Foundations of deep reinforcement learning: Theory and practice in python*. Boston, MA: Addison Wesley, 2020.

[43] G. Giardini and T. Kalmár-Nagy, “Genetic Algorithm for combinatorial path planning: The subtour problem,” *Math. Probl. Eng.*, vol. 2011, pp. 1–31, 2011.

[44] N. A. Melchior and R. Simmons, “Graph-based trajectory planning through programming by demonstration,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2012.

[45] Z. Aljarboua, “Geometric Path Planning for General Robot Manipulators,” 2009.

[46] S. M. LaValle, *Planning Algorithms*. Cambridge, England: Cambridge University Press, 2009.

[47] A. Sethian, “Level Set Methods and Fast Marching Methods,” *Cambridge, U.K: Cambridge Univ. Press*, 1999.

[48] X. Song, M. Cheng, B. Wang, S. Huang, X. Huang, and J. Yang, “Adaptive fast marching method for automatic liver segmentation from CT images: Adaptive fast marching method for automatic liver segmentation,” *Med. Phys.*, vol. 40, no. 9, p. 091917, 2013.

[49] J. Monsegny *et al.*, “Fast marching method in seismic ray tracing on parallel GPU devices,” in *Communications in Computer and Information Science*, Cham: Springer International Publishing, 2019, pp. 101–111.

[50] Y. Liu and P. W. Ayers, “Finding minimum energy reaction paths on ab initio potential energy surfaces using the fast marching method,” *J. Math. Chem.*, vol. 49, no. 7, pp. 1291–1301, 2011.

[51] S. Garrido, L. Moreno, M. Abderrahim, and F. Martin, “Path planning for mobile robot navigation using Voronoi diagram and fast marching,” in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2006.

[52] A. Valero-Gomez, J. V. Gomez, S. Garrido, and L. Moreno, “The path to efficiency: Fast marching method for safer, more efficient mobile robot trajectories,” *IEEE Robot. Autom. Mag.*, vol. 20, no. 4, pp. 111–120, 2013.

[53] S. Garrido, L. Moreno, M. Abderrahim, and D. Blanco, “Fm 2: A real-time sensor-based feedback controller for mobile robots,” *Int. J. Robot. Autom.*, vol. 24, no. 1, 2009.

[54] N. A. and B. M, *Reinforcement Learning with Keras, TensorFlow, and ChainerRL*. In: Reinforcement Learning. Apress, 2018.

[55] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, “Benchmarking deep reinforcement learning for continuous control,” *arXiv [cs.LG]*, 2016.

[56] S. R. Ronald Parr, “Reinforcement Learning with Hierarchies of Machines,” *Computer Science Division, UC Berkeley*, 1998.

[57] S. Feng and B. Sebastian, “Ben-Tzvi collision Avoidance Method Based on Deep Reinforcement Learning,” *Robotics*.

[58] S. Feng, H. Ren, X. Wang, and P. Ben-Tzvi, “Mobile robot obstacle avoidance based on Deep Reinforcement Learning,” in *Volume 5A: 43rd Mechanisms and Robotics Conference*, 2019.