Politecnico di Torino

Politecnico di Torino

Master's Degree in Mechatronic Engineering

A.a 2023/2024

April 2024

COMPLEX ENVIRONMENT EXPLORATION

Supervisors:

Prof. Marcello Chiaberge

PhD Andrea Eirale

Candidate:

Student Francesco Gervino

Abstract

Autonomous exploration of complex, unknown environments is a cutting-edge task not completely solved by the scientific community. When an agent needs to explore a maze without any a priori information about the environment, the lack of proper destinations and explicit task objectives make traditional navigation policies inappropriate. While the literature presents some sporadic deterministic systems able to face the tasks, learning approaches still need to be adequately investigated, which could prove more suitable and versatile for this purpose.
This thesis project's main goal is to develop a path planner able to optimise the exploration of complex unknown environments, such as mazes. The proposed solution exploits two cooperating modules: local and global planners. We model the scenario as a Markov Decision Process (MDP) and then train a Reinforcement Learning agent to solve the planning problem locally. This agent has access to image representations of a section of the global map, always centred in the robot reference frame, and decides the next navigation goal to complete the local exploration. The global planner is a deterministic system that recovers the navigation when a local solution is unavailable.
We compared our agent with a close-to-optimal, deterministic approach. The results obtained demonstrate the reinforcement learning agent's efficiency, reaching near-optimal levels in significantly less time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Objective of the thesis

Autonomous exploration of complex, unknown environments is a cutting-edge task not completely solved by the scientific community. When an agent needs to explore a maze without any a priori information about the environment, the lack of proper destinations and explicit task objectives make traditional navigation policies inappropriate. While the literature presents some sporadic deterministic systems able to face the tasks, learning approaches still need to be adequately investigated, which could prove more suitable and versatile for this purpose.
This thesis project's main goal is to develop a path planner able to optimise the exploration of complex unknown environments, such as mazes. The proposed solution exploits two cooperating modules: local and global planners.
The scenario is modelled as a Markov Decision Process (MDP) having the Local map as state and a pixel-wise action-space and rewards.
The agent is composed by a convolutional neural network trained by a Double DQN algorithm to solve the planning problem locally. This agent has access to image representations of a section of the global map, always centred in the robot reference frame, and decides the next navigation goal to complete the local exploration. The global planner uses a Greedy algorithm that recovers the navigation when a local solution is unavailable.
The trained agent is compared with a close-to-optimal deterministic approach (i.e. a completely greedy agent).
The results obtained demonstrate the reinforcement learning agent's efficiency, reaching near-optimal levels in significantly less time.

1.2 Organisation of the thesis

The thesis is composed of seven chapters, organised as follows:

- The first chapter is an introduction to the project, presenting the main reasons and goals, and concisely explaining the concepts covered in the following chapters.

- The second chapter is a theoretical explanation of the main search algorithms, starting with a brief description behind these methods. They are essential for finding a path in a map from a location to another.

- The third chapter begins with a brief clarification of some of the Machine Learning basis to finish explaining Deep Learning Neural Networks in details and why their applications in other Artificial Intelligence methods is so important.

- The fourth chapter is a theoretical introduction to Reinforcement Learning. It exposes the basic knowledge of the topic showing some of the basic algorithms. Then, it explains how Deep Reinforcement Learning works, focusing on Deep Q-Learning and Double DQN: the algorithm implemented in this project to train the agent.

- The fifth chapter presents the implementation of the code developed to create the entire system. It shows how the main Python classes work and how the training algorithm created with a Tensor-Flow library has been utilised.
  It also explains the ideas behind the development of the agent: from what the components of the MDP are to how the decision making is made.

- The sixth chapter summarises the major difficulties encountered in this thesis project, and shows the results of the main attempts.

- The seventh chapter concludes the paper, exposing how outcomes obtained by this work can be improved presenting a series of ideas that could enhance the training section or the project itself.

Chapter 2

Search Algorithms

Once the agent selects a place to go, a search algorithm is activated to find the optimal path that lead to that point.
In order to better understand how this algorithm works, a brief excursus on search algorithms is needed.

2.1 Search Problems

The algorithms that will be explained later are all based on **Classical Planning**.
It is a form of State Space Search Problem representation.

Formal definition:

- a finite and discrete **state space** S
- a known **initial state** $s_0 \in S$
- a non-empty set of goal states $S_G \subseteq S$ (or a **goal test**)
- set of **actions/operators** $A(s) \in A$ applicable in each state $s \in S$
- A **successor** function – a deterministic state transition function $f(a, s)$ such that $s' = f(a, s)$ stands for the state resulting from applying an applicable action a at state s
- A **cost function** $c(a, s)$ associating a positive cost to applying action a in s

Objective:

- Find a plan, a sequence of applicable actions, that leads from the initial state to a goal state.

- Optimality criteria - typically interested in shortest/cheapest plan (but other considerations are possible)

Typically, it is not possible to explicitly specify the **state space** S. Instead, **feature set** X is used to represent the state space such that a state is described via a set of random variables $X = x_1, ..., x_n$ such that each variable takes on values in some finite domain $Dom(x_i)$ which can be either Boolean or multi-valued.

2.2 Search space

The whole search problem can be represented differently:

- **State-space graph**
  - Every **state** is represented by a **node** and each of them occurs only once
  - The **actions** are expressed as **edges**
  - It might not be possible to hold the entire graph in memory all the time, hence it is expanded when necessary in the search

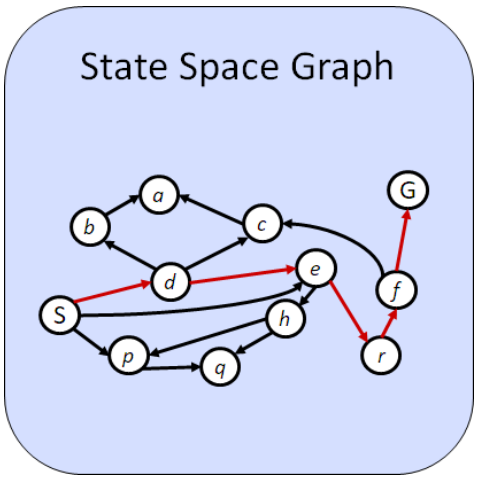


Figure 2.1: Example of State-Space Graph [8]

- **Search Tree**
  - Every **state** is represented by a **node**, which now correspond to **plans** that achieve those states
  - The **actions** are expressed as **edges**
  - The **start state** is the **root node**
  - Each **leaf node** represents a **path** from **start** to **leaf node**

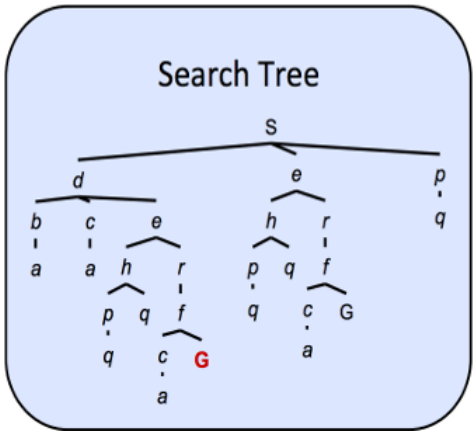


Figure 2.2: Example of Search Tree

For any algorithms, there are two important properties:
Resources:

- **Time Complexity**, which is concerned with the time the algorithm takes to finish
- **Memory Complexity**, that takes into account how much memory the algorithm needs

Quality of solution:

- **Optimality**, which evaluates the quality of the algorithm, in other words, if the algorithm finds the best or at least a good path
- **Completeness**, that checks if the algorithms can find a solution if one exists

2.3 Blind Search Algorithms

In **Tree Search** the number of nodes in the entire tree could be gigantic or even infinite, if a loop is present in a **State-Space graph**.
In order to avoid time-demanding searches, there are several variants for search algorithms for search trees, which could be expanded to graphs as well.
The first typology is called **Blind Search Algorithms**, since they do

not have any specific knowledge or information about the problem other than the initial state and the possible actions to take, and few variants are depicted in the following paragraphs.

2.3.1 Breadth-First Search (BFS)

The idea behind this algorithm is to expand the **shallowest node** first and to use a **queue** for the "nodes to be expanded".

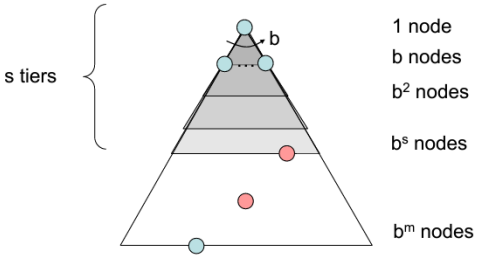


Figure 2.3: BFS scheme

- **Time Complexity** is $\mathbf{O(b^s)}$ with **s** being the layer number of the goal node
- **Memory Consumption** is the same, therefore it is equal to $\mathbf{O(b^s)}$
- It is **complete** since it will finish if the goal exists and lies at a finite layer $\mathbf{s} \neq \infty$
- **If** all **costs** are **equal** is **optimal** as it will find the **shortest** path
- The **Goal** node is selected when it is **added**, and not only when it is expanded

2.3.2 Uniform Cost Search (UCS)

In case the edges do not have equal cost, **BFS** could be enhanced into a **UCS** algorithm, which expands the nodes with the **lowest cost** first. It still uses a **queue**, but it is ordered by the cost.

- **Time Complexity** is $\mathbf{O(b^{C^*/\epsilon})}$ with $\mathbf{C^*}$ being the the cost of the goal node and ϵ being the minimum cost through the whole tree. $\mathbf{C^*/\epsilon}$ could be depicted as the "number of cost tiers until reaching the solution"
- **Memory Consumption** is the same, hence it is equal to $\mathbf{O(b^{C^*/\epsilon})}$
- The algorithm is **NOT complete**, because there could be infinite nodes in a wrong branch, where each node adds less and less to the overall cost

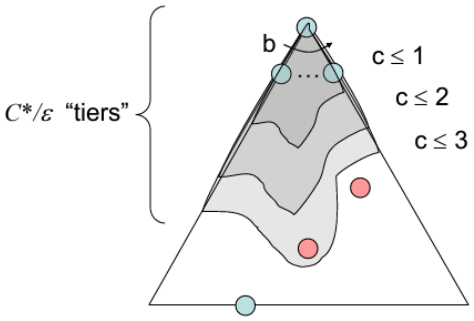


Figure 2.4: UCS scheme

- It is **optimal**
- The **Goal** node is only selected when it is **expanded**.

2.3.3 Depth-First Search (DFS)

The idea behind this algorithm is to expand the **deepest node** first and to use a **stack memory** for the "nodes to be expanded".

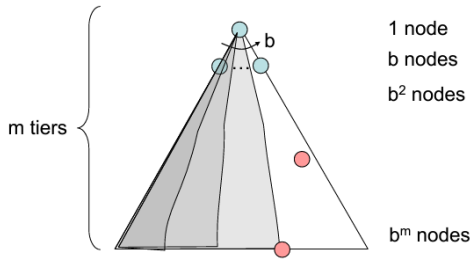


Figure 2.5: DFS scheme

- **Time Complexity** is $\mathbf{O(b^m)}$ with **m** being deepest layer
- **Memory Consumption** has a much better improvement, since it just stores the **b child nodes** of each layer. Therefore, it is equal to $\mathbf{O(bm)}$
- **If m** is **not infinite** the algorithm is **complete**, m could be infinite if there are loops in the graph
- It is **NOT optimal** since it might find a solution that is not the shortest one, because there could be a better solution in a really shallow area in another branch.
- The **Goal** node is selected when it is **added**, and not only when it is expanded

In order to reduce the **Time Complexity**, the DFS algorithm can be stopped after **L** layers, which causes the Time complexity to go down to $\mathbf{O(b^L)}$, this method is called **DFS-L**.

However, it is still **NOT complete** since the Goal state might lies on a deeper layer.

Iterative Deepening DFS (IDDFS)

This algorithm addresses all the issues of a simple DFS method incrementing the depth **L** of DFS-L every time the algorithm finishes its search. Doing that, the shallow nodes will be expanded multiple times and that will produces a higher **Time Complexity**, but a way lower **Memory Consumption**.

- **Time Complexity** is $\mathbf{O(b^s)}$ with **s** being the layer number of the goal node, since only the last search is relevant and during the last search $\mathbf{L = S}$
- **Memory Consumption** the same as for DFS with $\mathbf{O(bs)}$
- The algorithm is **complete** since DFS-L is complete when **s** lies on the **L** layer
- **If** all **costs** are **equal** is **optimal** as it will find the **shortest** path
- Iterative Deepening is not faster than BFS, but it needs less memory

2.4 Informed Search Algorithms

Informed Search utilises data from the nodes themselves to decide on the best next step.
A typical method to do this is by using a **heuristic**, which is often calculated by a heuristic function $\mathbf{h(s)}$ that **estimates** how close a state is to a goal, e.g the Euclidean distance.

2.4.1 Greedy best-first Search (GBS)

This search algorithm expands the node that has the lowest heuristic to the goal node and it utilises a queue ordered by the heuristics value.

- **Time Complexity** and **Memory Consumption** depend on the heuristic
- It is **NOT complete** when graph is **infinite**, even when start and goal node are not set in infinity, since it might explore an infinite branch with a lower heuristic

- The algorithm is **NOT optimal**, since the heuristic will most likely be different from the real cost

2.4.2 A-star (A*)

$\mathbf{A}^*$ algorithm expands the node that has the lowest $\mathbf{f(s)}$ value which can be evaluated as:

$$f(s) = h(s) + g(s) \tag{2.1}$$

where $\mathbf{h(s)}$ is the **heuristic function** and $\mathbf{g(s)}$ is the **true cost** to arrive on that state. Hence, $\mathbf{A}^*$ is a combination of **UCS** and **Greedy**.
This algorithm generates a **queue** ordered by the **f** value.

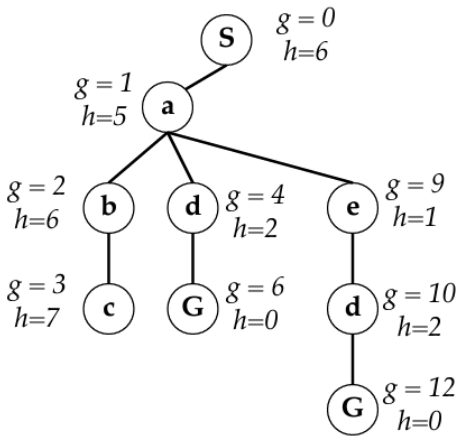


Figure 2.6: Example of a Search Tree solved with A*

In order to evaluate the **optimality** of an $\mathbf{A}^*$ algorithm two important **heuristic function** properties must be considered:

- **Admissibility**, a heuristic $\mathbf{h(s)}$ is **admissible** (optimistic) if:

  $$\forall x : 0 \leq h(x) \leq h^*(x) \tag{2.2}$$

  where $\mathbf{h}^*\mathbf{(x)}$ is the **true cost** to the nearest goal.
  The **Admissibility** ensures the **optimality** of an algorithm only if it is able to move nodes back from CLOSED to OPEN
- **Consistency**, a heuristic function is called **consistent** (also called **monotonic**) if:

  $$\forall s \in S, \forall s' \in SUCC(s) : h(s) - h(s') \leq COST(s, s') \tag{2.3}$$

  Therefore, a consistent heuristic is not only globally optimistic (like admissible heuristics) but also **locally optimistic**.

The **Consistency** guarantees the **optimality** of an algorithm, even if it does not shift nodes back from CLOSED to OPEN.

There are several enhanced versions of A* algorithm useful to change or calibrate some of its characteristics.

Weighted A*

It is used to tune the 'greediness' of the algorithm, giving more importance either to $\mathbf{g(s)}$ or to $\mathbf{h(s)}$. Indeed the $\mathbf{f(s)}$ is obtained by the following equation:

$$f(s) = (1 - w) \cdot g(s) + w \cdot h(s) \tag{2.4}$$

This brings three special cases:

- $\mathbf{w = 1}$, the algorithm becomes a **greedy** one
- $\mathbf{w = 0.5}$, the agent will perform a **uniform cost** search
- $\mathbf{w = 0.5}$, the algorithm will act as a normal A*

Therefore, increasing **w** speeds up the search to the detriment of a worse solution, but in a given bound (described by the w value). That is the reason why this method is defined as a **bounded sub-optimal** algorithm.

A*-epsilon

It is a greedy approach which prefers nodes that are already closer the goal.

In other words, instead of choosing the node with the best **f**-value, the algorithm chooses the one that has the best heuristic of all nodes with a good **f**-value.

This version follows these steps to select the next node:

- Obtain the node with the lowest **f**-value, called $\mathbf{f_{min}}$
- Collect every node that falls in a cost range, defined by:

$$f(s) \leq (1 + \epsilon) \cdot f_{min} \tag{2.5}$$

- Select the node with the lowest heuristic value $\mathbf{h_{focal}}$ among these nodes
- This new heuristic function is given in addition of the normal one

The effect is equal to the **weighted A*** as it improves the computation time by decreasing the quality of the solution, but only in a given bound. Therefore, increasing ϵ speeds up the search but it worsens the solution. The cost of the found solution can't be worse than $(\mathbf{1} + \epsilon) \cdot \mathbf{C_{optimal}}$

Iterative Deepening A* (IDA*)

This algorithm starts with high greediness and become more uniform if it does not find a solution.
It suspends the search when it reaches a node which is bigger than a maximum **f**-value called bound and it increments this value in every iteration.
The algorithm is explained by the following steps:

- Start with a bound that is equal to the heuristic of the start node. Hence, it only allows greedy searches, that decrease (or don't change) the **f**-value in each step
- Detect the highest **f**-value encountered during the search
- Set the aforementioned **f**-value as the bound
- Run until a solution is found

As the name suggests, this version is more similar to an Iterative Deepening than an A*, but it is dependent on **f**.
Maintaining an OPEN and CLOSED list is not necessary, therefore, it has a better **Memory Consumption** and the path is still **optimal**.

2.4.3 Summary Table

Algorithm		**Complete**	**Optimal**	**Time**	**Space**
Blind	BFS	Y	Y*	$O(b^m)$	$O(b^m)$
	DFS	N**	N	$O(b^m)$	$O(bm)$
	IDDFS	Y	Y	$O(b^m)$	$O(bm)$
	UCS	Y***	Y***	$O(b^m)$	$O(b^m)$
Informed	GBS	N**	N	$O(b^m)$	$O(b^m)$
	A*	Y*** ∘	Y*** ∘	$O(b^m)$	$O(b^m)$
	IDA*	Y*** ∘	Y*** ∘	$O(b^m)$	$O(mb)$

Table 2.1: Summary Table of Search Algorithms

Legend:
*: When every edge has unitary cost
**: Swaps to Y when the graph has no cycles, hence the tree is finite
***: When costs are positive and cannot be infinitely precise
∘: When heuristic function is admissible

Chapter 3

Machine Learning

The concept of classification anticipates the rise of Machine Learning, tracing its roots back through centuries of statistical methodologies used for categorisation and pattern recognition. However, recent technological advancements, particularly in Deep Learning, have substantially enhanced its effectiveness and applicability. To comprehensively understand what a classification problem entails and why Deep Learning has emerged as a powerful solution, it is essential to delve into the foundational principles of machine learning.
Learning means that an agent improves its performance on future tasks after making observations.
It is dependent on different factors:

- What component of an agent should be improved.
  For example:
  - Decision: Get actions from the current state.
  - Observation: Get relevant properties of the world from perception sequences.
  - Prediction: Get result from a potential action.
  - Evaluation: Get desirability information from the current world.

- What prior knowledge has the agent.

- Which representation is used for the data.

- Through which feedback does the agent learn?
  - **Unsupervised learning:** Learn patterns without feedback.
  - **Reinforcement learning:** Learn from rewards or punishments.
  - **Supervised learning:** Learn from training data.

3.1 Supervised Learning

Supervised learning algorithms elaborate a dataset containing features associated with each example, or object, which has a label or target that the algorithm needs to predict.
There are several types of Supervised learning algorithms, in this chapter classification and linear regression will be analysed.

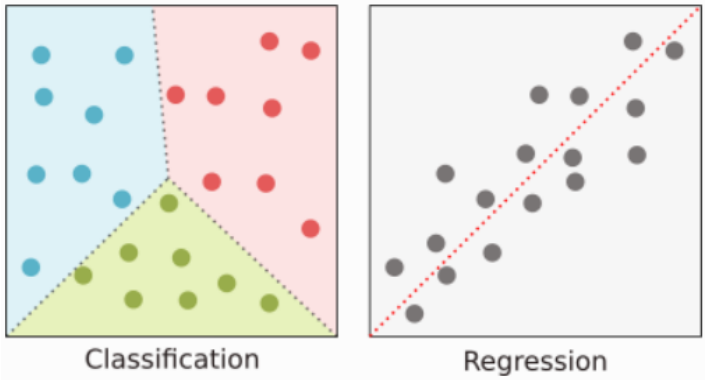


Figure 3.1: Example of classification and linear regression [7]

3.1.1 Classification

A typical sub-problem for supervised learning is classification. A classifier is a function mapping between objects $o \in O$ and classes $c(o)$.
If there are only two classes, it is called binary classification.
The objects and the class can also be denoted as x and y respectively.
A classification is performed using an expert or learning from a set of examples:

- Given a **Training set**: $\{(x_1, y_1), (x_2, y_2), ...(x_N, y_N)\}$
- An unknown **Target classifier**: $f : x \rightarrow y$ that gives a mapping between input x and output y.
- The goal is to find a function h, called **hypothesis**, that approximates f.
- When h is obtained, the output related to other objects outside the training set can be predicted. That is called **generalisation**.

Most of the times additional data are present and they are given by a set of functions called **features** or **attributes** that characterise every single

object.
It is now important to identify the methods to determine if a hypothesis is good.

First of all, a hypothesis is **consistent** with a training set if:

- Given a target classifier $f_c := O \rightarrow \{0, 1\}$
- Given some hypothesis $h : O \rightarrow \{0, 1\}$
- $O' \subseteq O$ iff $\forall o \in O' \, [h(o) = f_c(o)]$

However, a consistent hypothesis is not always worth it. For example, in a noisy training set some labels could be wrong and the hypothesis wants to learn them anyway. This problem will be addressed later in the chapter.
Typically a **test set** is set aside to test the quality of the hypothesis. Theoretical models typically assume that the training set and test set are sampled from the same distribution.

A "good" hypothesis can be measured by two values:

Let **N** denote the size of the test set.

Error: $\frac{False\_Positive + False\_Negative}{N}$
Accuracy: $\frac{True\_Positive + True\_Negative}{N}$

The first one evaluates the percentage of errors of an hypothesis whilst the latter one considers the number of correct predictions made by the hypothesis.

3.1.2 Linear Regression

Another simple example of supervised learning is Linear regression. S the name implies, it solves a regression problem, where the goal is to predict a continuous scalar value $y \in \mathbb{R}$ for a given input vector $\mathbf{x} \in \mathbb{R}^n$. In other words, the output variable is a real or continuous value rather than a discrete category or class.
Therefore, the predicted model of y can be defined as:

$$\hat{y} = f(\mathbf{x}) = \theta \cdot \mathbf{x}$$

where θ is a vector of parameters, sometimes called **weight** vector and indicated by $\mathbf{w}$ and $\hat{y}$ is the prediction of the model. Each parameter θ_i influences the corresponding feature x_i.
Hence, the linear regression tries to estimate the probability distribution of y in function of $\mathbf{x}$ and θ,$p(y|\mathbf{x}, \theta)$, such that:

$$p(y|\mathbf{x}, \theta) = f(\mathbf{x}) = \theta \cdot \mathbf{x}$$

Now, the performance of the model can be measured computing the **mean squared error** on the test set:

$$MSE = \frac{1}{m} \sum_i (\hat{\mathbf{y}} - \mathbf{y})_i^2$$

where $\hat{\mathbf{y}}$ are the prediction on the test set, $\mathbf{y}$ are the true outputs and m is the number of examples.
Therefore, since the goal is to design a model that finds the parameters θ of the true model, it is necessary to minimise the MSE of the training set.
This cost function can be reduced with the**Normal Equation**:

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}\mathbf{y}$$

where $\hat{\theta}$ are the predicted parameters and $\mathbf{X}$ is the training set itself.

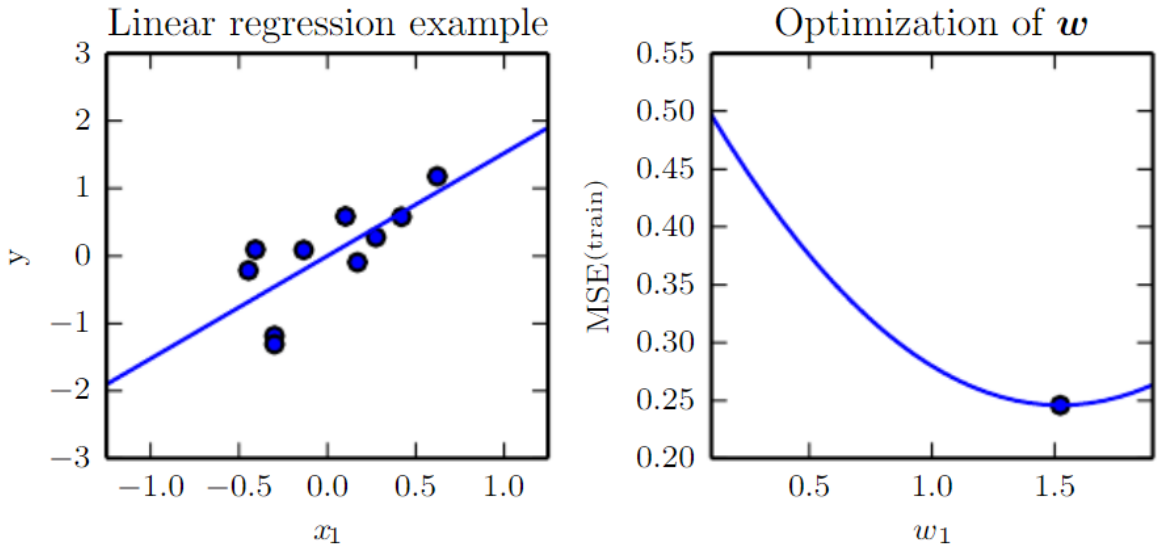


Figure 3.2: Example of linear regression and optimisation of the weight [5]

It is worth noting that, an additional parameter called **intercept term** b

is added to the predicted model:

$$\hat{y} = \theta\mathbf{x} + b$$

This term is a **bias** parameter which helps the linear function to not necessary pass through the origin.

3.2 Overfitting and Underfitting

The primary challenge in machine learning is ensuring that an algorithm performs effectively on new, unseen data, not just on the data it was trained on. This capability to perform well on unfamiliar inputs is known as **generalisation**.
Unlike simple optimisation problems, which minimise errors on the training set, machine learning goes further by aiming to minimise the generalisation error, also known as the test error. This error represents the anticipated error on new inputs, as it considers various possible inputs drawn from the expected distribution encountered in real-world scenarios.
The factors that determine how well a ML algorithm will perform are the following two abilities:

- Minimise the training error
- Minimise the gap between training and test error

If the algorithm fails to ensure the first ability, **underfitting** occurs; whilst **overfitting** happens when the second ability is not satisfied. Roughly speaking underfitting occurs when the algorithm fails to learn how to manipulate the training data, while overfitting is present when the algorithm fails to generalise a task; hence, it is not able to produce the same results on a test set.
The likelihood of a model overfitting or underfitting can be managed by adjusting its **capacity**. In simple terms, a model's capacity refers to its capability to accommodate a diverse range of functions. Models with low capacity might find it challenging to fit the training data adequately, while those with high capacity can suffer from overfitting, where they memorise specific properties of the training data that may not generalise well to unseen test data.

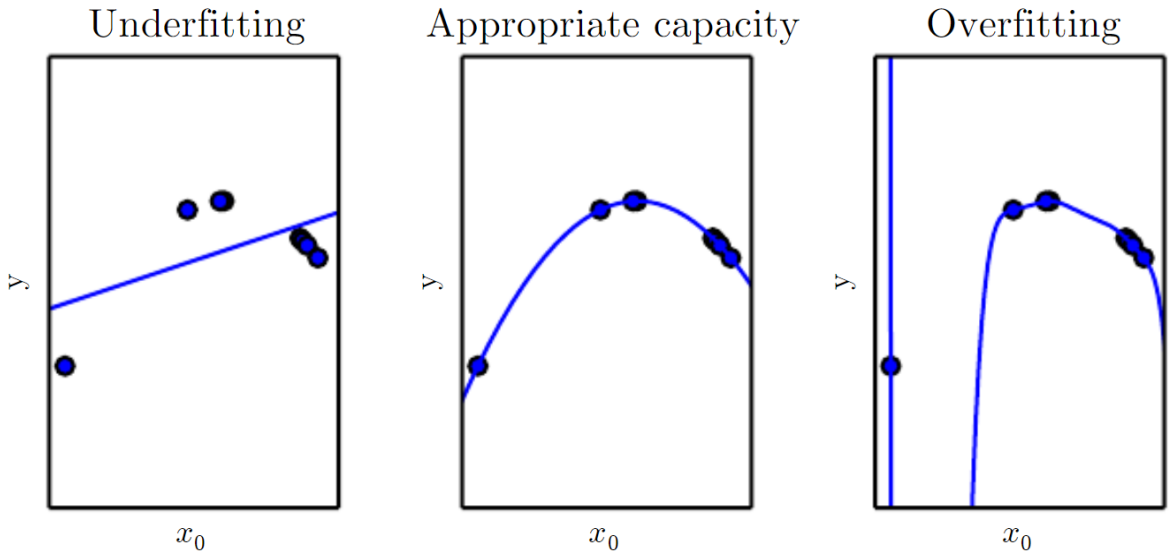


Figure 3.3: Example of underfitting, overfitting and appropriate capacity

3.3 Deep Learning

Deep Learning is a sub-field of machine learning that focuses on training algorithms to learn representations of data through multiple layers of abstraction. At the core of deep learning are **neural networks**, computational models inspired by the structure and function of the human brain. Neural networks consist of interconnected nodes organised in layers. Each node, or neuron, processes information and passes it on to the next layer. Deep neural networks contain multiple hidden layers between the input and output layers, enabling them to learn complex patterns and representations from raw data.
Through a process called backpropagation, neural networks adjust their internal parameters during training to minimise the difference between predicted and actual outputs. This enables them to generalise well to new, unseen data, making them powerful tools for tasks such as image and speech recognition, natural language processing, and many others.

3.3.1 Artificial Neuron

The basic computational unit of the brain is the neuron, with approximately 86 billion neurons comprising the human nervous system. These neurons are interconnected via synapses. Neurons receive input signals through dendrites and produce output signals along their axons. The

strength of these connections, or synapses, influences the interaction between neurons.
In mathematical models of neurons, input signals travelling along axons interact multiplicatively with dendrites based on synaptic strengths. These strengths, represented as **weights** w, are adjustable and determine the influence of one neuron on another. The summation of these signals at the cell body determines if the neuron fires, sending a spike along its axon. Timing of spikes is typically disregarded, and only the frequency of firing, conveyed by the firing rate or **activation function** f, is considered.
The activation function helps in squashing the signal strength to a manageable range.

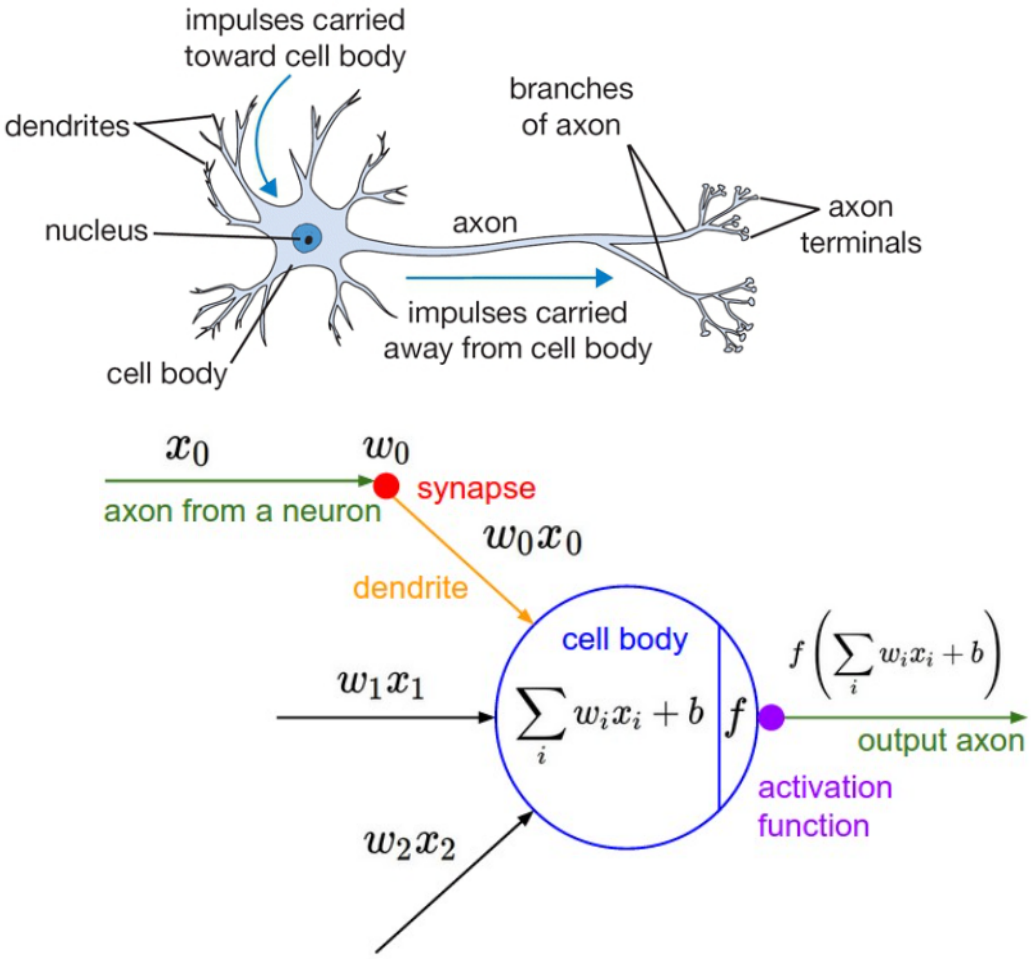


Figure 3.4: Comparison between biological and artificial neuron [1]

3.3.2 Architecture of Artificial Neural Networks (NN)

Grouping up artificial neurons and connecting them to each other creates the so called Neural Network.
The neurons that compose a network are sorted by layers:

- **Input Layer:** it receives the input signals x_i that the network will process and manipulate
- **Hidden Layers:** they are composed by neurons that receive weighted signals. Each neuron manipulates these signals with an activation function a_j producing an output signal
- **Output Layer:** it works similarly to the Hidden Layers but this is the actual output of the neural network.

Each connection between two neurons is characterised by a weight w_{ij} which decreases or boosts the numerical signal carried by that connection. In order to improve the flexibility of a NN, sometimes a bias b_j is added to the input signal of a neuron enabling the network to model more complex relationships in the data.

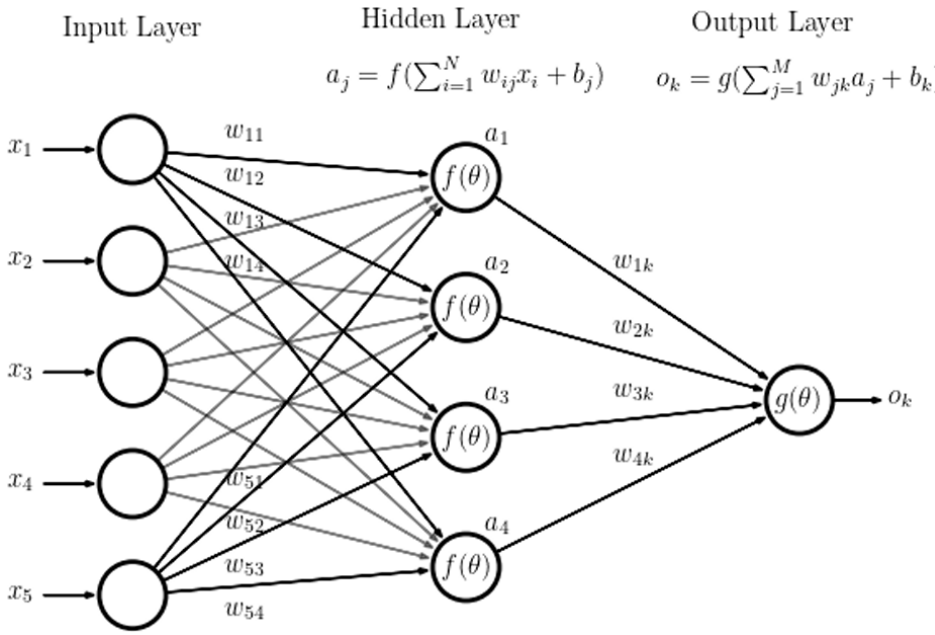


Figure 3.5: Example of neural network [21]

3.3.3 Activation Functions

Activation functions are crucial components in artificial neural networks, serving as mathematical operations that determine the output of a neuron. They introduce non-linearity, enabling neural networks to learn complex patterns and relationships within data.
Each function transforms the input data into a specific range of outputs, facilitating the network's ability to make predictions or classifications. Activation functions play a pivotal role in optimising the performance of neural networks by controlling the flow of information between layers, ultimately enhancing the network's learning capabilities.
Amongst them, the most utilised are the following:

Sigmoid function

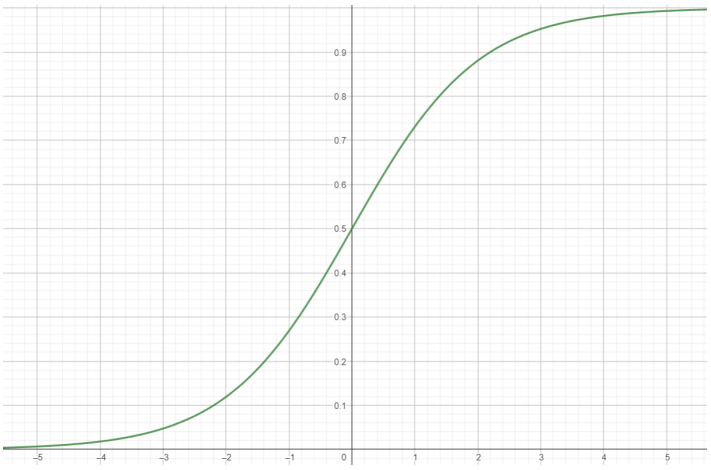


Figure 3.6: Sigmoid activation function

The sigmoid function, characterised by its distinctive S-shaped curve, is a mathematical tool widely used across various fields due to its capability of transforming any real number into a value between 0 and 1.
The sigmoid function is typically denoted by the Greek letter σ and defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Sigmoid units are prone to the "vanishing gradient" problem, which poses a significant challenge to learning in deep neural networks.
When input values become highly positive or negative, the sigmoid function saturates, causing it to output values very close to 0 or 1, with an

almost flat slope in these regions. Consequently, the gradient approaches zero, leading to minimal weight adjustments during backpropagation. This issue is especially pronounced for neurons in the earlier layers of deep networks. The negligible gradients impede learning, causing it to progress painfully slowly or even come to a halt. This phenomenon is commonly known as the vanishing gradient problem in neural networks.

tanh function

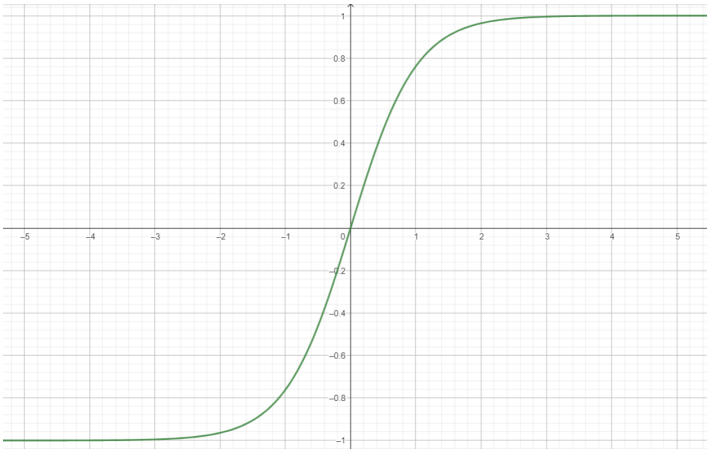


Figure 3.7: tanh activation function

The tanh function, unlike the sigmoid function, produces output values ranging from -1 to +1, making it more effective in handling negative values. Its zero-centred nature, with outputs symmetrically distributed around the origin, is often considered advantageous as it aids in faster convergence of learning algorithms.

$$tanh(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Due to its wider range and zero-centredness, the tanh function exhibits stronger gradients compared to the sigmoid function. These stronger gradients contribute to quicker learning and convergence during training, as they are more resistant to the issue of vanishing gradients, which often hinder learning in deep networks.
However, despite these benefits, the tanh function still encounters the vanishing gradient problem, particularly in deep networks with numerous layers.

ReLU function

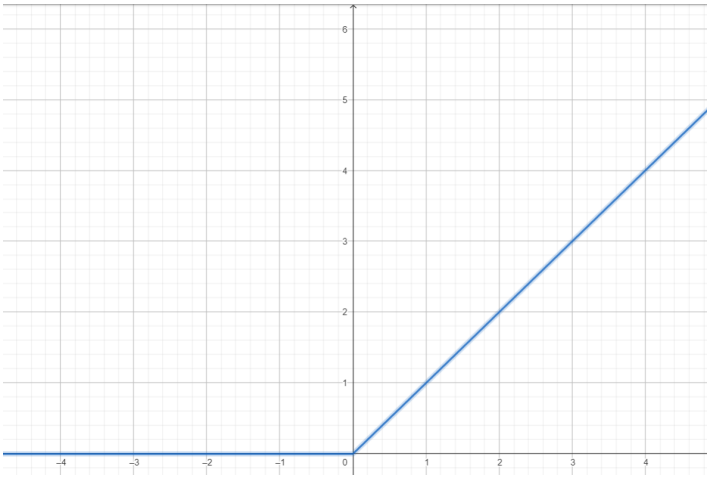


Figure 3.8: ReLU activation function

The Rectified Linear Unit (ReLU) activation function has the form:

$$f(z) = max(0, z)$$

The Rectified Linear Unit (ReLU) function is characterised by thresholding the input at zero, returning 0 for negative values and the input itself for positive values.
For inputs greater than 0, ReLU behaves as a linear function with a gradient of 1. This preserves the scale of positive inputs and ensures that the gradient remains unchanged during backpropagation. This property is crucial for addressing the vanishing gradient problem commonly encountered in deep neural networks.
Although ReLU exhibits linearity for half of its input space, it is considered a non-linear function due to its non-differentiable point at x=0, where it abruptly transitions from x. This non-linearity enables neural networks to learn intricate patterns and relationships in the data.
By outputting zero for all negative inputs, ReLU naturally induces sparse activations. Consequently, only a subset of neurons are activated at any given time, leading to more efficient computation.
Moreover, the computational simplicity of the ReLU function makes it cost-effective in terms of computation. Its straightforward thresholding operation at zero allows neural networks to scale to numerous layers without significantly increasing the computational burden, unlike more complex activation functions such as tanh or sigmoid.

Softmax function

The softmax activation function, also known as the normalised exponential function, is highly beneficial in multi-class classification scenarios. It operates on a vector, often called logits, which contains the raw predictions or scores for each class computed by preceding layers of a neural network. For an input vector x with elements $x1, x2, ..., x_N$, the softmax function is defined as:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

The output of softmax constitutes a probability distribution summing up to one. Each element of the output denotes the probability that the input belongs to a specific class.
The use of the exponential function guarantees non-negative output values, a critical requirement as probabilities cannot be negative.
Softmax accentuates disparities in the input vector. Even minor discrepancies in input values can result in significant variations in output probabilities, with the highest input value often prevailing in the resulting probability distribution.
The probabilities derived from softmax can be interpreted as confidence scores for each class, offering insights into the model's certainty regarding its predictions.
However, due to its ability to amplify differences, softmax may be sensitive to outliers or extreme values. For instance, if the input vector contains exceptionally large values, softmax may disproportionately suppress the probabilities of other classes, leading to an overly confident model.

3.3.4 Backpropagation

Backpropagation is a technique used in artificial neural networks to train them. It involves adjusting the weights of connections between neurons by propagating the error backwards from the output layer to the input layer. This adjustment helps the network learn to produce more accurate outputs for a given input.
Roughly speaking, the neural network learns from mistakes: the network identifies how much it has deviated from the expected outcome and adjusts itself accordingly to minimise that deviation in future predictions.
Before delving into a deeper explanation, a definition of the error δ_j^l must

be given:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

This error refers to the j^{th} neuron in layer l and z_j^l is the input of such neuron.
C is called **Cost function** and evaluates how the network is adapting its weight. Its general expression is the following:

$$C(w,b) = \frac{1}{2n}\sum_x ||y(x) - a^L(x)||^2$$

where w and b are the weight and bias of the network, n is the number of sample used in the learning process, x is the input signal, $y(x)$ is the desired output, L is the number of layers and $a^L(x)$ is the vector of activation output from the network when x is input.
Backpropagation is based on four equations:

- **Error in the output layer**

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$$

  The expression considers the influence of the jth output on the cost function and how fast the activation function σ is changing at z_j^L.
  The equation in a matrix-based form can be expressed using an element-wise product:

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

- **Error in terms of next layer error**

$$\delta^l = ((w^{l+1})^T\delta^{l+1}) \odot \sigma'(z^l)$$

  This step is fundamental and lies at the core of the backpropagation principle. It involves propagating the error from one layer to the preceding one while accounting for the activation functions in between.

- **Error in terms of any bias in the network**

$$\frac{\partial C}{\partial_j^l} = \delta^l$$

- **Error in terms of weights**

$$\frac{\partial C}{\partial w_{jk}^{l}} = a_{k}^{l-1}\delta_{j}^{l}$$

By the use of these equations, the objective of backpropagation is to minimise the Cost function.

3.3.5 Convolutional Neural Network (CNN)

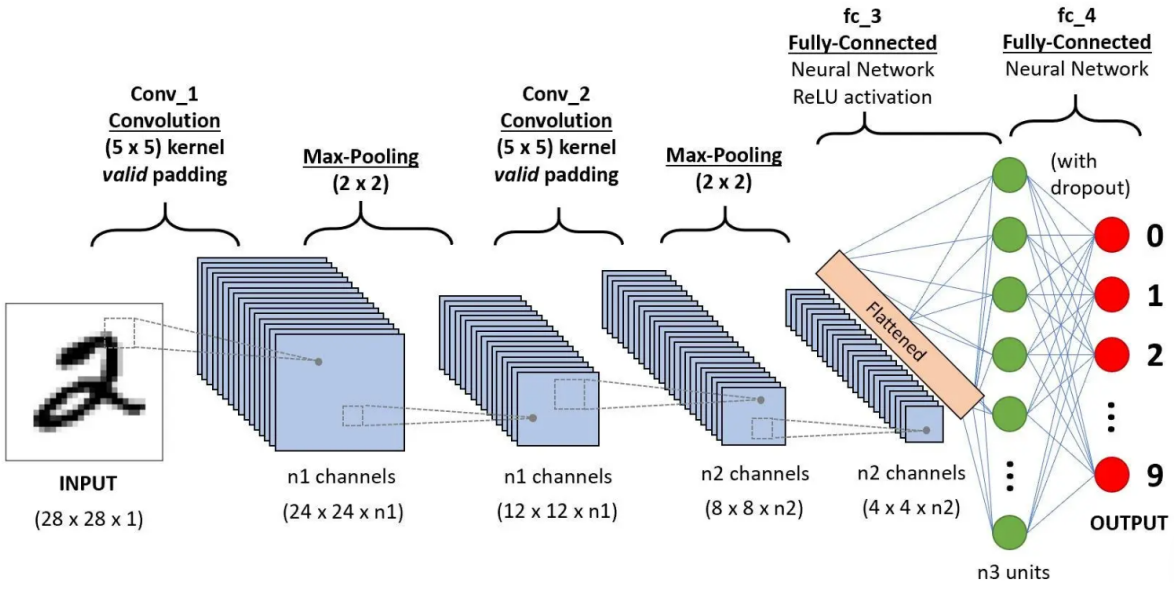


Figure 3.9: CNN architecture example [12]

Convolutional Neural Networks (CNNs) are a type of artificial neural network commonly used in image recognition and classification tasks. Inspired by the human visual system, CNNs are designed to automatically and adaptively learn spatial hierarchies of features from input images. They employ convolutional layers to apply filters or kernels over the input data, extracting features such as edges, textures, and patterns. Pooling layers are then used to reduce the spatial dimensions of the features while retaining important information. Finally, fully connected layers are employed for classification or regression tasks based on the extracted features. CNNs have become incredibly successful in various applications including image recognition, object detection, and even natural language processing.

CNNs respect the spatial structure of the input, keeping it a matrix, where each neuron contains the intensity of each pixel.

Filter

Regarding the implementation of a hidden layer, instead of connecting every input pixel with every hidden neuron, connections are established only within small, localised regions of the input image. Each connection is assigned a weight, and each new hidden neuron is given a bias. By sliding the region across the entire image, the hidden layer can be constructed. Each neuron in a hidden layer will share the same weights and bias, implying that all neurons will detect the same feature in the input image but at different positions. This characteristic makes CNNs well-suited to handling translation variances, as they can recognise features in the image even if they are translated to different positions. Hence, the mapping from the input to the hidden layer is referred to as a **feature map**, with the weights and bias being termed **shared weights** and **shared bias**, respectively. These shared weights and bias parameters define a matrix known as a **kernel**, **filter**, or feature detector.

Pooling

In a CNN a pooling layer is typically inserted immediately after a convolutional layer. The pooling layer serves to simplify the information contained in the preceding convolutional layer.
Max Pooling returns the maximum value from the section of the image covered by the kernel, whilst **Average Pooling** returns the average of all the values from the section of the image covered by the Kernel.

Flattening and Fully Connected Layers

At the end of a CNN the matrix of neurons is converted in an array. This process is called Flattening. The flattened neurons are often followed by a couple of Fully Connected Layers, which consists in a layer where each neuron is connected with all the neurons of the previous layer.

Chapter 4

Reinforcement Learning

Reinforcement Learning (RL) is a sub-field of Machine Learning focused on learning to map situations to actions to maximise a numerical reward signal. Unlike supervised learning, where actions are explicitly labelled, in RL, the learner must discover which actions yield the most reward through trial and error. In challenging scenarios, actions may not only influence immediate rewards but also impact subsequent situations and, consequently, all future rewards. This interplay between trial-and-error search and delayed rewards forms the core of reinforcement learning.

4.1 Elements of Reinforcement Learning

Reinforcement Learning is characterised by several components, some of them are necessary whilst others might be optional:

- **Agent**, it is the component that makes the decision of what **action** to take. The **agent** perceives the **environment** through a set of observations or sensors. These observations provide information about the current **state** of the **environment**. Receiving them, it is able to interact with the **environment** selecting an **action** from an action-space domain, which can be either discrete, continuous or a discrete-continuous hybrid.

- **Environment**, which is the system with which the **agent** interacts. The **agent** analyses the **environment** through observations and derives a set of data called **state**.

- **State**, it is what the **agent** receives as input data and it is a depiction

of the current configuration of the **environment**. Every **state** is defined in a state-space domain and it is usually made of every useful piece of information for the **agent** so that it is able to make the wisest decision.

- **Policy**, which defines the learning **agent**'s way of behaving at a given time: $\pi : S \mapsto A$. It can be either stochastic or deterministic and be represented by a simple function whereas in other cases it may involve extensive computation such as a search process. The aim of RL is to obtain the optimal **policy**, since it would mean that the **agent** is able to accumulate the highest possible **reward**.

- **Reward function**, it is a scalar value that the **agent** receives as feedback from the **environment** after taking a particular **action** in a specific **state**: $R(s, a) = r$.

- **Value function**, whilst the **reward** indicates the quality of an **action** in the immediate case, the **value function** estimates what is good in the long run, since a low-reward action might bring the agent to a state closed to the goal one or create a situation where the agent enters in a loop of positive rewards.

  $$V^{\pi}(s) = \mathbb{E}_{a\sim\pi}[G_t | s_t = s, a_t = a]$$

  where $\mathbf{G_t}$ is the sum of the **rewards** in a given amount of **actions** starting from the state $\mathbf{s_t}$, which is also called **Return**:

  $$G_t = \sum_{i=0}^{N} r_{t+i+1}$$

  where **N** can be infinite. A **Discount factor** γ is often utilised to deteriorate the reward after every step inducing the agent to obtain the rewards faster. Therefore, $\mathbf{G_t}$ has now the following form:

  $$G_t = \sum_{i=0}^{N} \gamma^i r_{t+i+1} \qquad \gamma \in [0, 1)$$

- **Model** (optional), which mimics the behaviour of the **environment**, allowing inferences to be made about how it will behave. The model is used for **planning**, since an **agent** tries to predict how the environment will be in the long run deciding accordingly what the most

appropriate **action** is. Methods for solving RL problems that use models and planning are called **model-based**, as opposed to simpler **model-free** methods, which utilises a trial-and-error learners.

4.2 Markov Decision Process (MDP)

MDPs are a mathematically idealised form of the reinforcement learning problem. It is a method used to model a problem or a situation that fits perfectly with RL, but first, a definition of what **"Markov"** means is mandatory.
Andrey Andreyevich Markov, 1856-1922, was a Russian mathematician best known for his work on stochastic processes who devised the **Markov Chain**, which describes a sequence of possible events in which the probability of each of them depends only on the state attained in the previous step. Indeed, **"Markov"** generally means that given the present state, the future is independent of the past; therefore, for Markov decision processes, the **action outcomes** depend only on the **current state**.
To sum up, a Markov Decision Process is a discrete-time control process that models decision making in situations where outcomes are partly random and partly under the control of a decision maker.

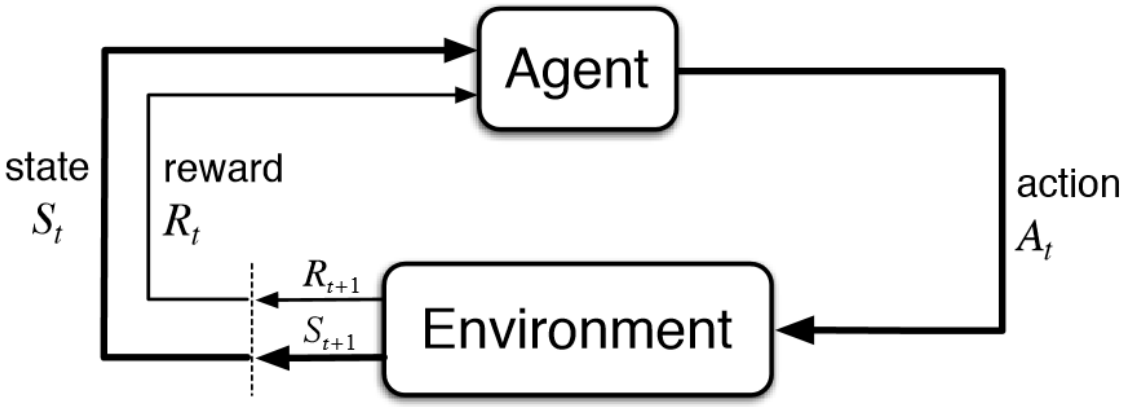


Figure 4.1: The agent–environment interaction in an MDP [16]

An MDP is defined by:

- A **set of states s** $\in$ **S**, which is typically defined over a **feature set** $\mathbf{X} = \mathbf{x_1}, ..., \mathbf{x_n}$, where each **state** has a unique combination of these values and each variable is defined by a finite domain $\mathbf{Dom}(\mathbf{x_i})$

- A **set of actions a** $\in$ **A**, as said before, the **actions** determine how the **agent** interacts with the **environment**
- A **transition function** $\mathbf{T}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, which is also called the **model** or the **dynamics**. It defines the probability that performing an **action a** from a **state s** leads to the **next state s'**, i.e. $\mathbf{P}(\mathbf{s}'|\mathbf{s}, \mathbf{a})$
- A **reward function** $\mathbf{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}') \in \mathbb{R}$, which can be sometimes defined as $\mathbf{R}(\mathbf{s}, \mathbf{a}), \mathbf{R}(\mathbf{s})$ or $\mathbf{R}(\mathbf{s}')$

Occasionally, it is necessary to define:

- A **start state** $\mathbf{s_0}$, where the **agent** starts the simulation
- A **set of terminal states**, wherein the **agent** terminates its run
- A **set of goal states**, which are the **states** the **agent** tries to reach during the simulation

Since MDP is discrete, the agent interacts with the environment during each specific **time step t**. At each time step t, the agent receives a **tuple** $(\mathbf{s_t}, \mathbf{a_t}, \mathbf{r_{t+1}}, \mathbf{s_{t+1}})$, which contains information about the state s_t the agent was at time t, the action a_t it took, the reward r_{t+1} it gained reaching the state s_{t+1} with that action.
These **tuples**, also called **SARS**, are the bricks that make up a **trajectory**, which is precisely a sequence of SARSs from the starting one until the one that ends the so called **episode**. E.g. $s_0, a_0, r_1, s_1, ..., r_T, s_T$
It is worth noting that at each state s, all the actions might not be available (i.e. $A(s) \subseteq A$ where $A(s)$ is the set of possible actions in state s)

4.2.1 Partially-Observable MDP (POMDP)

A Partially Observable Markov Decision Process (POMDP) is a mathematical model used in decision-making scenarios where an agent interacts with an environment, but the agent does not have complete information about the state of the environment. In a POMDP, the environment is modelled as a Markov Decision Process, but the agent receives only partial and potentially noisy observations about the true state.
As an MDP, a POMDP is defined by:

- A **set of states** $\mathbf{s} \in \mathbf{S}$
- A **set of actions** $\mathbf{a} \in \mathbf{A}$
- A **transition function** $\mathbf{T}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$
- A **reward function** $\mathbf{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}') \in \mathbb{R}$
- A **discount factor** γ

In addition, it is also characterised by:

- A **set of observations** $\mathbf{o} \in \mathbf{O}$, when an **observation** occurs it affects the **observation probability** of an event. E.g. there are two doors and there is some noise from the left one, the probability of encountering something in the left room is higher than the right one.
- An **Observation Probability** $\boldsymbol{\Omega}(\mathbf{s}, \mathbf{a}, \mathbf{o})$ or $\mathbb{O}(\mathbf{o}|\mathbf{s}, \mathbf{a})$, which determines the **probability** of reaching a **state** given an **action** and an **observation**
- An **initial belief** $\mathbf{b_0} : \mathbf{S} \to [\mathbf{0}, \mathbf{1}]$, which is the **probability** of being in different **initial states**

4.3 Bellman Equations

As previously said, the objective in RL and in an MDP is to find an **optimal policy** π^* (the apex * stands for **optimal**)and the **Bellman equation** is the perfect method to exploit this optimal policy and that is why it is present in every Reinforcement Learning literature.
The **Bellman equation** decomposes the **value function** into two parts, the **immediate reward** plus the **discounted future values**.
This equation simplifies the computation of the value function, since, instead of summing over multiple time steps, it obtains the optimal solution of a complex problem by dividing it into simpler, recursive sub-problems and finding their optimal solutions.
In the case of the **State-value function** $\mathbf{V}^{\pi}(\mathbf{s})$, which is the expected utility starting in s and acting according to π thereafter, the Bellman

equation is defined as:

$$V^{\pi}(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(S_t) \cdot (\pi(S_t), S_{t+1})]$$
$$= \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

where $\pi(s)$ is the taken action.
Before evaluating the **optimal State-Value function $\mathbf{V}^*(\mathbf{s})$**, it is mandatory to define the value (utility) of a **q-state(s,a) $\mathbf{Q}^{\pi}(\mathbf{s}, \mathbf{a})$**, which is the expected utility starting out having taken action a from state s and acting according to π thereafter:

$$Q^{\pi}(s, a) = \sum_{s'} P(s'|s, a) \cdot [R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

Now, the **optimal policy $\pi^*(\mathbf{s})$** can be derived by the following recursive method:

$$V^*(s) = \max_a Q^*(s, a)$$
$$Q^*(s, a) = \sum_{s'} P(s'|s, a) \cdot [R(s, \pi(s), s') + \gamma V^*(s')]$$
$$V^*(s) = \max_a \sum_{s'} P(s'|s, a) \cdot [R(s, \pi(s), s') + \gamma V^*(s')]$$

Therefore, **value functions** allow defining a partial ordering over policies such that $\pi \geq \pi'$ if $V^{\pi} \geq V^{\pi'}, \forall s \in S$. Using this definition, it can be stated that it **exists** an **optimal policy $\pi^*(\mathbf{s})$** for **any MDP**, which is better than or equal to any other policy $\pi^* \geq \pi, \forall \pi$ and all the optimal policies achieve the optimal state value function and the optimal action-value function.

4.4 Dynamic Programming

Dynamic Programming (DP) is a useful mathematical optimisation technique that fits perfectly in the field of Reinforcement Learning (RL) to solve problems where an agent makes sequential decisions over time. Indeed, this method allows to compute an optimal policy for an MDP since the future state depends only on the current state and action.
This approach can be used to tackle complex problems by breaking them

into more feasible sub-problems, which can be obtained with the Bellman equations. The, the sub-problems are solved and their solutions combined. Dynamic programming is based on the **Principle of Optimality**, which ensures that any optimal policy can be subdivided into two components

4.4.1 Value Iteration

The first DP algorithm described is called **Value iteration**, which aims to determine the optimal value function. It is an **iterative** process that systematically rectifies these value estimates until they converge to their optimal values.
This algorithm follows these steps:

- Initialise the time step to zero
- Calculate the optimal values for the state wherein the agent is utilising the Bellman optimality equation
- Increase the allowed time step
- Repeat the algorithm until the values converge to a fixed one or the variation is smaller than a determined range

An important feature of this method is **Policy Extraction**, which determines how many steps are needed to obtain an optimal policy. In order to do that, it is necessary to find the best action that maximises the q-value in a given state.
This can be obtained by two variants:

- Search among all the q-values for a given state. The action that belongs to the highest one is optimal:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}}\, Q^*(s, a)$$

- Calculate all the q-values and find the optimal value:

$$\pi^*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s, a) \cdot [R(s, \pi(s), s') + \gamma V^{\pi}(s')]$$

It can be clearly seen that it is possible to make one less step if the q-values are already known.
Another important aspect is **Value Convergence** since it might be impossible to reach a constant state value. Therefore, terminating the algorithm after gaining a good result is necessary to not loom into an infinite solution.

There are two main ideas to obtain that: the first one is to stop the iteration after a fixed amount of steps, which is a quite naive method since it is impossible to know 'a priori' how long it takes to reach a decent solution. On the contrary, the second idea can overcome this issue and it is composed by the following steps:

- Define a **discount** $\mathbf{0 < \gamma < 1}$
- Since the **Bellman equation** is a **contraction**, i.e. a function that brings two values 'closer' to each other, the function will get closer and closer after each iteration to a **fixed point**.
  E.g. $f(x) = x/2$ is a contraction because:

  $$\forall a, b \in \mathbb{R}: \quad |a - b| > |f(a) - f(b)|$$

  This function $f(x)$ brings any number closer to the fixed point 0
- Rewrite the **Bellman optimality equation** to a **vector equation**:

  $$V_{k+1} = B \cdot V_k$$

  where $\mathbf{V_k}$ contains the values for all states at time $\mathbf{k}$
- Since the distance between two vectors as the norm:

  $$||V - V'||$$

  The following formula can be derived:

  $$||B \cdot V_k - B \cdot V'_k|| \leq \gamma \cdot ||V_k - V'_k||$$

  Therefore, Bellman is a contraction if $\gamma < 1$

It is worth noting that for a given error ϵ, we need at least $\mathbf{N}$ iterations, which can be calculated using the biggest reward $\mathbf{R_{max}}$.

$$N = log(\frac{2R_{max}}{\epsilon(1-\gamma)})/log(\frac{1}{\gamma})$$

The **Time Complexity** for each iteration in value iteration is $\mathbf{O(N^2_{states} \cdot N_{actions})}$
This method carries out an interesting effect: the **policy** might **converge** much **faster** than the **values**.
The policy is optimal if the **policy loss** (i.e. the difference between what

the best policy is and the current one) is zero.
The **value loss** can be denoted as $||\mathbf{V}^{\pi} - \mathbf{V}^{*}||$, which means that every state counts into it.
It is interesting to note that if the value iterations is stopped at some value V_k, there will be an error denoted as: $||\mathbf{V_k} - \mathbf{V}^{*}|| = \epsilon$. Hence, the policy loss for the policy extracted is bounded by: $||\mathbf{V}_{\pi} - \mathbf{V}^{*}|| \leq \mathbf{2}\epsilon$.

Asynchronous Value Iteration

The original Value Iteration algorithm updates each state in every iteration. However, any sequences of Bellman updates will converge if every state is visited infinitely often: Hence, updating the states whose neighbours have recently changed can decrease the computational cost of the algorithm. In other words, the main idea is upgrading states whose value it is expected to change, i.e. if $|V_{i+1}(s) - V_i(s)|$ is high enough then the predecessors of s will be updated.

4.4.2 Policy Iteration

Policy iteration, or approximation in the policy space, is an algorithm that uses the special structure of infinite-horizon stationary dynamic programming problems to find all optimal policies.
It is a dynamic programming technique for calculating a policy directly, rather than calculating an optimal $V(s)$ and extracting a policy; but one that uses the concept of values.
This algorithm is composed by two main parts:

- **Policy evaluation**, which consists in choosing a random policy π. Then it calculates the values $\mathbf{V}^{\pi}(\mathbf{s})$ for all states for this policy, using the Bellman Equation.
  This process is similar to Value Iteration, but the maximum is not computed since the actions are fully determined by the policy.
  Policy evaluation starts initialising:

  $$\forall s \in S: \quad V_0^{\pi}(s) = 0$$

  Then it iterates until convergence:

  $$V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s, \pi(s)) \cdot [R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

The **Time Complexity** for each iteration of this algorithm is $\mathbf{O}(\mathbf{N}^{\mathbf{2}}_{\mathbf{states}})$.

- **Policy Improvement**, which searches for a better policy by using **policy extraction**:

$$\pi_{i+1}(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s'|s,a) \cdot [R(s,\pi(s),s') + \gamma V^{\pi_i}(s')]$$

This formula might seem to return the original policy π_i, but this is avoid by calculating the values without maximum operation, since all possible actions are explored.
The **Time Complexity** for policy improvement is $\mathbf{O}(\mathbf{N}^{\mathbf{2}}_{\mathbf{states}} \cdot \mathbf{N}_{\mathbf{actions}})$.

To sum up, Policy evaluation is also called **Passive RL** and measures how good is a given policy: the agent has a fixed policy and tries to learn the utilities of states by observing the world go by. This method often serves as a component of active learning algorithms.
On the contrary, Policy improvement, or **Active RL**, finds what is a good or optimal policy: the agent tries to find an optimal policy (or at least good policy) by acting in the world. Analogous to solving the underlying MDP, but without first being given the MDP model.

4.5 Model-based and Model-free approaches

These active and passive RL methods can be divided into two different type of algorithms:

4.5.1 Model-based RL

Model-based RL algorithms learn a model of the environment (i.e., the reward and transition functions) or an approximation of the model by interacting with it. Then they compute a policy using the model (e.g., using planning).
Therefore, this model predicts the consequences of actions, enabling the agent to plan ahead.
Initially, the agent learns or is provided with a model of the environment, typically in the form of transition probabilities (how likely the environment will transition from one state to another given an action). After that, with the learned model, the agent can simulate future states and

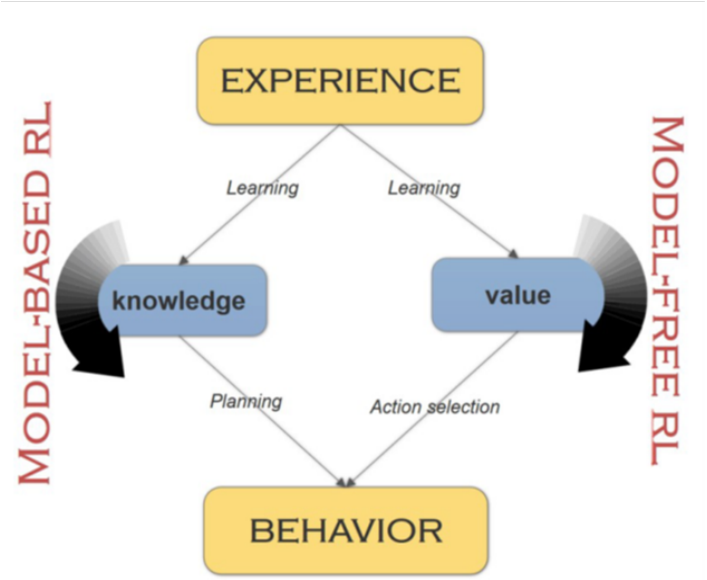


Figure 4.2: Scheme representing the difference between Model-based and Model-free RL [18]

rewards, allowing it to plan its actions by considering possible future outcomes. Techniques like dynamic programming or tree search algorithms can be used for planning.
This method has several advantages such as performing more efficient planning by leveraging the learned model to simulate future scenarios and, in some cases, model-based approaches can require fewer interactions with the environment to learn an optimal policy.
On the other hand, if the learned model does not accurately capture the dynamics of the environment, the plans made based on it can be sub-optimal and building and maintaining an accurate model can be computationally expensive, especially in complex environments.

4.5.2 Model-free RL

Model-free RL does not require an explicit model of the environment. Instead, it directly learns the optimal policy or value function through trial-and-error interactions with the environment.
The agent estimates the value function (expected cumulative reward) directly from experience, without explicitly modelling the environment dynamics and, based on the estimated value function, the agent updates its policy to maximise expected future rewards.
This type of algorithms are simpler to implement and do not require ex-

plicit knowledge of the environment dynamics and can handle environments with complex dynamics or unknown dynamics more robustly. However, these methods often require more interactions with the environment to learn an optimal policy compared to model-based approaches and balancing exploration (trying new actions to discover optimal ones) and exploitation (leveraging known actions for immediate reward) is a challenging aspect of model-free RL that will be covered later in this section.

4.6 Passive Reinforcement Learning

Passive Reinforcement Learning algorithms review an existing policy. Hence, its aim is to determine the values for all states for a given policy π.
The idea is to execute a set of trials, or rollouts, using π. Some methods are presented in this paper.

4.6.1 Monte Carlo Estimation

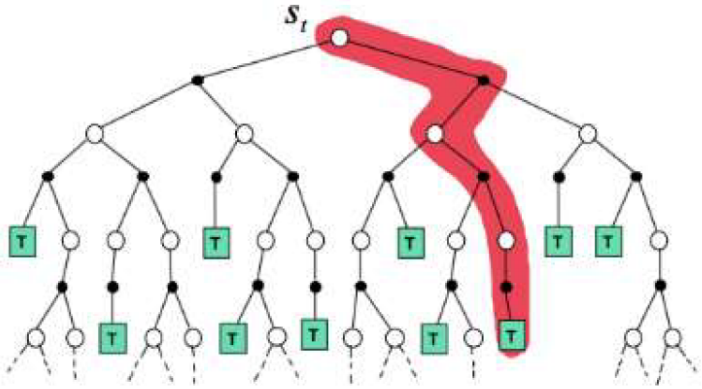


Figure 4.3: Example of how MC Estimation acts [15]

Monte Carlo Estimation, or **Direct utility estimation**, is a **Model-free** approach that runs trials evaluating the **reward-to-go** (i.e. the sum of discounted rewards from that state until a terminal state is reached) of a state and using this value to update the estimated utility of that state. After each trial the new estimate is updated as follows:

$$\hat{x}_{n+1} = \hat{x}_n + \frac{1}{n+1}(x_{n+1} - \hat{x}_n)$$

where n is the number of samples made, $\hat{x}_n$ is the old estimate and x_{n+1} is the new sample.

Each trial (simulation) provides a sample of this quantity for each state visited during that trial, if a state is visited multiple times in a simulation every visit handles the samples as independent.
The problem of this method is that it ignores that the utility of a state is determined by the reward and the expected utility of the successor states. Thus, with direct estimation utility values do not obey the Bellman equations for a fixed policy. Typically, this means the converge very slowly to correct utilities values. Therefore, it requires a lot of sequences.

4.6.2 Adaptive Dynamic Programming (ADP)

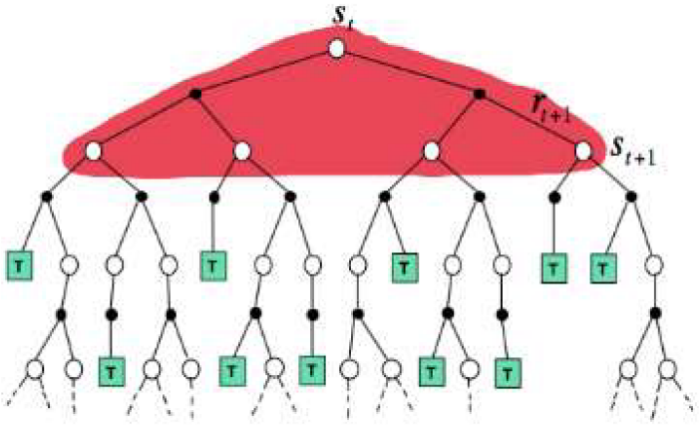


Figure 4.4: Example of how ADP acts

Adaptive Dynamic Programming is a **Model-based** approach which follows the given policy for a while. Then, it learns the transition and reward functions estimating models based on observations.
It firstly assumes that the reward function is deterministic, which means it collects the reward for each transition once. Finally, it uses the estimated model to compute utility (e.g., using value iteration).
In order to learn the model, it records how often state s' is reached when executing action a in state s. Then, it estimates transition model $P(s'|s,a)$ as the fraction of times it sees s' after taking action a in state s.
where *percept* indicates the current state s' and reward signal r, whilst the persistent data are:

- The fixed policy π.
- An MDP with model P, rewards R, actions A and discount γ.
- A table of utilities for states initially empty U.

Algorithm 1: Passive ADP

Data: percept
persistent data : $\pi, mdp, U, N_{s'|s,a}, s, a$;
if *s'* **is** *new* **then**
  $U[s'] \leftarrow 0$;
end
if s **is not** *null* **then**
  $N_{s'|s,a}[s, a] + +$;
  $R[s, a, s'] \leftarrow r$;
  `A[s].append(a)`;
  $P(\cdot|s, a) \leftarrow$ `Normalise`$(N_{s'|s,a}[s, a])$;
  $U \leftarrow$ `PolicyEvaluation`(π, U, mdp);
  $s, a \leftarrow s', \pi[s']$;
  return a
end

- A table of outcome count vectors indexed by state and action initially zero $N_{s'|s,a}$.
- The previous state and action initially null s, a.

The model that is learned can be used for different agents with different objectives (and reward functions).
However, the approach is limited only by its ability to learn the model and, in its basic form, it is intractable for large spaces.

4.6.3 Temporal-difference (TD) Learning

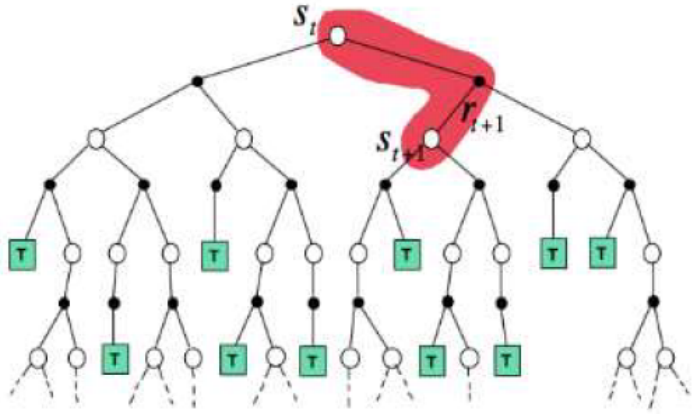


Figure 4.5: Example of how TD acts

In order to avoid the computational expense of full DP policy evaluation, another approach is to adjust the utilities of the observed states, so

that they agree with the Bellman equations.
Temporal Difference Learning is a **model-free** approach. Hence, it does not try to estimate the entire transition or reward function. Instead, it calculates the local updates of utility or value function on a per-action basis.
For each transition from s to s', it performs the following update called **Temporal Difference equation**:

$$V^{\pi}(s) = V^{\pi}(s) + \alpha[R(s, \pi(s), s') + \gamma V^{\pi}(s') - V^{\pi}(s)]$$

where α is the **learning rate**, that determines how much this equation is affected by the updates.
TD uses **bootstrapping**, i.e. estimating a value based on another estimation: the algorithm estimates the value of state based on the value of its consecutive states.
This method is one of the key ideas in modern RL.
Temporal-Difference (TD) Learning is a combination of Monte Carlo methods, since it can learn from experience without knowing the model, and Dynamic programming, as it updates an estimate based on other learned estimates.
It is worth noting that each error is proportional to the change over time of the prediction, that is, to the temporal differences in predictions.
$R(s, \pi(s), s') + \gamma V^{\pi}(s')$ is a noisy sample of the utility based on the next step: the TD term represents the error between the observed and estimated values, which is important to minimise. Therefore, the TD update is about to maintain a "mean" of (noisy) utility samples, but if the learning rate decreases appropriately with the number of samples (e.g. $\frac{1}{n}$), then the utility estimates will converge to true values.

4.7 Active Reinforcement Learning

In the case it is needed to find a good or optimal policy while acting in an uncertain environment, it is necessary to utilise an Active Reinforcement Learning algorithm.
A very simple and naive approach would be using a random policy and run trials based on that, then learn the transition function and the rewards using passive reinforcement learning and finally running normal value or

Algorithm 2: Passive TD

Data: percept
persistent data : π, s, U, N_s;
if *s'* **is** *new* **then**
  $U[s'] \leftarrow 0$;
end
if s **is not** *null* **then**
  $N_s[s] + +$;
  $U \leftarrow U[s] + \alpha(N_s[s]) \cdot (r + \gamma U[s'] - U[s])$;
  $s \leftarrow s'$;
  return $\pi[s']$
end

policy iteration to get the best policy.
However, this method may have to run random trials for a very long time until it gets good estimates for the transition function.
Before analysing several interesting Active RL methods, it is necessary to explain the exploration vs exploitation problem.

4.7.1 Exploration vs Exploitation

Exploitation and exploration represent two fundamental strategies that an agent can employ to maximise its cumulative reward while interacting with an environment.
These strategies are essential for balancing the trade-off between exploiting known information to gain immediate rewards and exploring unknown regions of the environment to discover potentially better actions. On the one hand, **Exploration** involves deliberately selecting actions that may not be optimal according to current knowledge or policy in order to gather additional information about the environment.
On the other hand, **Exploitation** implies selecting actions that are believed to be the best based on current knowledge or past experiences. The agent leverages its existing understanding of the environment to choose actions that are expected to yield high rewards.
In order to obtain an optimal behaviour, the algorithm needs to follow a scheme called **Greedy in the Limit with Infinite Exploration (GLIE)**, which can be obtained satisfying the following two properties:

- If a state is visited infinitely often, then each action in that state is

chosen infinitely often (Exploration). No action is "forgotten".

- In the limit (as $t \to \infty$), the learning policy becomes greedy with respect to the learned model (Exploitation). "Greedy" means that it will choose the action that maximises the q-value of the next step. This is the base idea for policy extraction too.

If an exploration scheme is GLIE, it will eventually obtain optimal behaviour. Two main approaches to get a good trade-off between exploration and exploitation are:

- **ϵ-greedy**: on time step t the agent selects a random action with probability $p(t)$ and a greedy action with probability $1 - p(t)$.
  The aim of this algorithm is to converge the probability of doing a random action to 0 as the simulation goes on.
  An example could be $p(t) = \frac{1}{n}$.
- **Boltzmann Exploration**: the agent selects an action with probability:

$$P(a|s) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{a' \in A} e^{\frac{Q(s,a')}{T}}}$$

  where T is the "temperature". Large T means that each action has about the same probability. Small T leads to more greedy behaviour. It is usual to start with large T and decrease it with time

4.7.2 ADP-based RL

It is a **model-based** approach that starts with a random model and it calculates the optimal policy π^* for this random model using value iteration or policy iteration.
During the trials it firstly ignores the policy π acting randomly (exploration). Then, it slowly shift towards using the policy (exploitation).
It updates the transition probabilities and rewards based on the algorithm also used in ADP. However, the difference with the previous ADP algorithm is that it does not calculate the values in the end.
The process is repeated until convergence.
This algorithm can be also performed by using **Optimistic Exploration** which consists in keeping track which state-action pairs (s, a) were visited how often in the second part of ADP-based RL. If the number is too small,

it just replaces the value for that state with the optimal value, which is:

$$V^{max} = \frac{R^{max}}{1-\gamma}$$

If the number is bigger than a threshold,it performs normal value iteration.

4.7.3 TD-based Active RL

Despite being inspired by the TD Learning in passive RL, this algorithm is **model-based**.
It only extracts the policy in the end.
During each step it initialises all values randomly and an arbitrary model. Then it performs policy extraction to get an action for the state wherein it currently is.
After executing an action based on an exploration scheme, it updates the model by using the algorithm from ADP and updates the values using the temporal difference.

4.7.4 Q-Learning

The third and final Active RL algorithm is Q-Learning, which is a **model-free** approach that extends TD-based active RL by removing the model and it is split in single action steps.
Instead of updating the model or values, we directly update the **q-values** utilising a formula similar to TD Learning:

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

It performs the updates after each action as it happens in TD.

4.8 Deep Reinforcement Learning

In the previous sections, the basics of Reinforcement Learning (RL) are covered, including techniques like Dynamic Programming (DP), Monte Carlo, and Temporal Difference (TD) methods. Now, as Deep Reinforcement Learning (DRL) is introduced, it is important to acknowledge the limitations of traditional tabular methods, especially in tasks with large

Algorithm 3: Q-learning

```
Data: α ∈ (0, 1], ϵ
Initialise Q(s, a), ∀s ∈ 𝕊, a ∈ 𝔸(s);
for each episode do
    Initialise S;
    for step in episode do
        Choose a from s using ϵ-greedy;
        Take action a, observe r, s′;
        Choose a from s′ using ϵ-greedy;
        Q^π(s, a) ← Q^π(s, a) + α[R(s, a, s′) + γ max_a′ Q^π(s′, a′) − Q^π(s, a)];
        s ← s′;
        if s is terminal then
            break
        end
    end
end
```

state spaces. Updating tables with data in a timely manner becomes prohibitively expensive, making it impractical to find an optimal policy and value function.

In such scenarios, encountering many new states is inevitable. The algorithm must learn to generalise its knowledge from past experiences to make effective decisions in unfamiliar situations. This challenge of generalisation often translates into a function approximation problem, with the value function being the primary target. This paper focuses on solutions utilising artificial neural networks for function approximation.

A key departure in this new framework is the representation of value functions. Instead of using tables, these functions are represented with parametric functional forms. This shift not only offers more efficient representation but also enables adaptation to complex, high-dimensional state spaces commonly encountered in real-world RL tasks.

Roughly speaking, in order to solve complex systems or train an agent to act on environments unseen during the training, it is necessary to make the agent able to generalise the situation wherein it is involved so that it will be versatile.

4.8.1 Experience Replay

Experience replay, a technique widely employed in reinforcement learning, was first examined by Lin in 1992. However, its recent resurgence can be

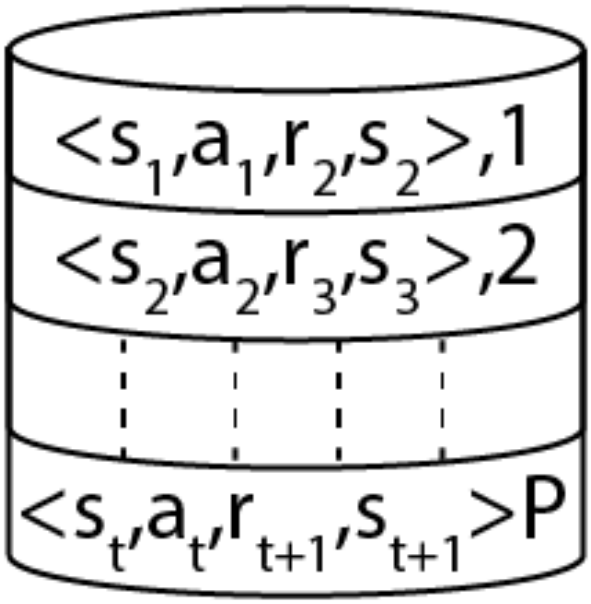


Figure 4.6: Experience replay depiction [10]

largely attributed to its implementation in the Deep Q-Network (DQN) algorithm by Mnih et al. [19] (2013), aimed at mastering ATARI games using Deep Reinforcement Learning (DRL).
This method involves storing transition tuples $(S_t, A_t, R_{t+1}, S_{t+1})$ in a memory buffer termed replay memory at each time step. These tuples encapsulate the agent's progression from one state, S_t, to the next, S_{t+1}, by executing a specific action, A_t, and receiving a reward, R_t. Once the replay memory accumulates sufficient transitions, mini-batches can be randomly sampled, enabling the agent's neural network to be trained using past experiences.
Experience replay offers several benefits. Firstly, it enhances the algorithm's data efficiency by enabling the agent to update its weights multiple times using experienced events. Moreover, it alleviates the learning process's instability caused by temporally correlated training samples, as reinforcement learning discourages consecutive sample use. Furthermore, experience replay smooths the learning process by averaging behaviour distribution across numerous previous states, thereby reducing the target function's reliance on current weights.
In summary, experience replay is well-suited for off-policy learning, providing a robust mechanism for training reinforcement learning agents effi-

ciently.

4.8.2 Target Network

In addition to experience replay, another approach to mitigate algorithm instability involves the utilisation of a secondary network termed the target network, as proposed by Mnih et al. This network serves to decouple the target function's dependence on the primary network's weights, aiming to enhance algorithm convergence, especially when employing TD-error.
The procedure entails periodically updating the parameters of the target network with those of the primary network after a set number of training steps. By doing so, the target network serves as a stable reference point, aiding in smoothing out fluctuations and promoting more consistent learning.
This strategy effectively breaks the direct link between the target function and the primary network's weights, contributing to improved stability and convergence in the training process.

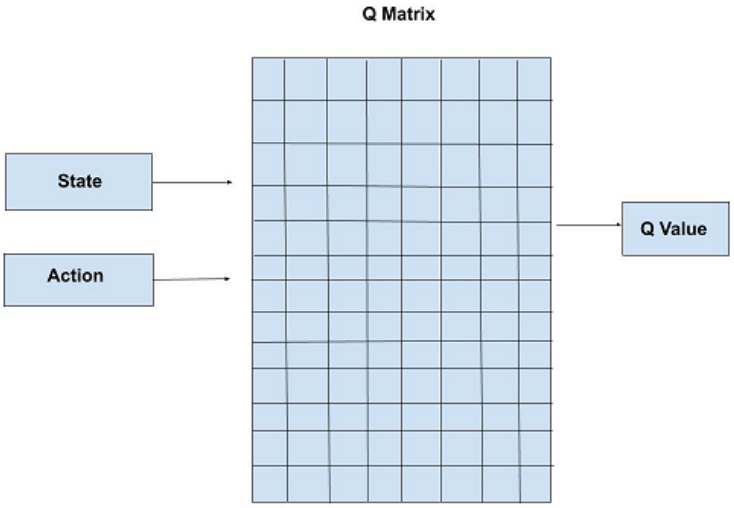
4.8.3 Deep Q-Learning (DQN)

Deep Q-Learning is a significant type of reinforcement learning algorithm that employs a deep neural network to estimate the Q-function, aiding in decision-making across various domains like gaming, robotics, and autonomous vehicles. This function helps determine the best action to take in a given state, based on expected cumulative rewards.
Unlike traditional Q-Learning, which uses tables for value representation, Deep Q-Learning utilises deep neural networks, called **Deep Q-network (DQN)**, to handle large state and action spaces, as well as complex inputs like images or sensor data.

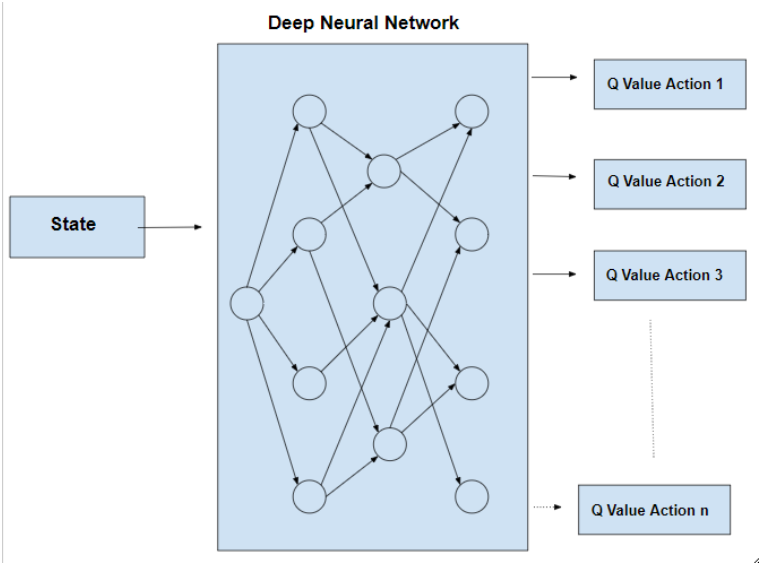
The DQN is usually trained with stochastic gradient descent (SGD), which aims to minimise the loss function $L_i(\theta_i)$ at each time step:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(s,a)}[y_i - Q(s,a,\theta_i))^2]$$

where y_i is the target on iteration i and $\rho(s,a)$ is the probability distribution over states and actions also called **behaviour distribution**.



((a)) Q-Learning [2]



((b)) Deep Q-Learning

However, implementing Deep Q-Learning comes with challenges. The Q-function's non-linear nature and numerous local minima can hinder neural network convergence. To address this, techniques such as experience replay and target networks have been developed to improve stability and convergence.

Deep Q-Learning finds application in diverse fields, including training agents for games like Atari and Go, and controlling robots for tasks such as grasping and navigation. Its adaptability underscores its importance in artificial intelligence and reinforcement learning.

To sum up, DQN is a **model-free** method which approximates the Q-values of each action with a Neural Network. As for the Q-Learning al-

gorithm, it utilises an **off-policy** approach since it does not always select action with the policy to be improved but it can greedly performs exploitation by choosing the action with the highest Q-value. That is caused by the ϵ-greedy approach.

Algorithm 4: Deep Q-Learning

Data: replay memory D of capacity N, number of episodes M, ϵ
Initialise approximating function Q;
Initialise D;
for *each episode* **do**
  Initialise S;
  for t ***in*** *episode* **do**
    Choose a_t from s_t using ϵ-greedy;
    Take action a_t, observe r_t, s_{t+1};
    `D.append(`$[s_t, a_t, r_t, s_{t+1}]$`);`
    $[s_k, a_k, r_k, s_{k+1}] \leftarrow D[random]$;
    if s ***is*** *terminal* **then**
      $y_i \leftarrow r_k$;
    end
    else
      $y_i \leftarrow r_k + \gamma \max_{a'} Q(s_{k+1}, a'; \theta)$;
    end
    Perform a gradient descent step on loss $y_i - Q(s, a, \theta_i))^2$;
  end
end
return Q

Double DQN

Taking the maximum overestimated values implies estimating the maximum value itself, leading to systematic overestimation and introducing maximisation bias in Q-learning. As Q-learning involves bootstrapping, where estimates are derived from existing estimates, this overestimation poses significant challenges.
To address maximisation bias in Q-learning, using two separate Q-value estimators is suggested. Each estimator updates the other, ensuring unbiased Q-value estimates of actions chosen using the alternate estimator.
This technique called Double Q-Learning had different versions, the one that will be shown in this paper is made by Hasselt et al. 2015, where there are two independents model of Q. One used for action selection **Q'**, called

target network, and the other for action evaluation **Q**, called **primary network**.
Hence, the equation to evaluate the Q-value is now defined as:

$$Q^*(s_t, a_t) \approx r_t + \gamma Q(s_{t+1}, \underset{a'}{\operatorname{argmax}}\, Q'(s_t, a_t))$$

The target network slowly copies the parameters of Q and it is usually updated through Polyak averaging:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

where θ' is the target network parameter, θ is the primary network parameter, and τ is called **rate of averaging**.

Chapter 5

Software Implementation

In this chapter it is explained how the whole code[1] has been implemented. In this thesis, an agent is trained to explore complex unknown environments, such as mazes, represented by grid maps. It was simulated in Python, creating the maze and agent classes from scratch. The robot scanned the surrounding area with a simulated 2D LiDAR.
The planner is divided into two parts: a Local Planner, which makes decisions observing a section of the global map wider than the agent's field of view and a Greedy Global Planner.
A Double DQN algorithm [3], which is an enhanced version of the DQN, was utilised to train the Local Planner; the Markov Decision Process model of the agent was the following: Local map as a **State**, a cluster of map pixels as an **Action** and the **Reward Function** takes into account how many new map squares are seen throughout an action, how many steps are made and if a wrong move (an inaccessible location) is selected.
The **Terminal State** of the Local Planner is a dead end. It is detected when there is nothing left to explore on the map, which means that none of the 'steppable' squares has anything unknown nearby. When the agent comes to a dead end, the Greedy Global Planner takes the next action.
The agent was compared with a completely greedy one (Greedy Local Planner and Greedy Global Planner) and the results showed that the trained agent is much more efficient.

[1] `https://github.com/PIC4SeRThesis/FrancescoGervino`

5.1 Maze Generation

The environment wherein the agent acts is a **maze**. The maze is a **grid map** made of a $\mathbf{N} \times \mathbf{N}$ **matrix**. Hence, every value of the matrix corresponds to a square (i.e. a location) of the map and the set of indexes correlated with that value are the set of coordinates that belong to that square. E.g. the value of $maze[3][4]$ defines what the square at coordinates $(3, 4)$ is.
Now, it is necessary to define what these values can be. Each element of the matrix is an **integer** and can have the following values:

- **maze[x][y] = 1**, it represents a **wall**, the agent cannot be in that square and see through it
- **maze[x][y] = 2**, it is regarded as an **explorable square**, i.e. an empty space wherein the agent can step and it does not block the agent's view
- **maze[x][y] = 3**, it represents an **explored square**, which is an **explorable square** the agent has already seen. This information is essential to detects when the agent has explored a specified percentage of the grid map

Since the agent's task is to explore a complex environment, this one must be **fully explorable**.
In order to do that, at the beginning the maze is just a matrix whose elements are all equal to one, i.e. a maze entirely made of walls. Then the algorithm 'digs' into the map **empty spaces** that can be explored. It starts from the centre of the map and continues to create explorable squares next to the one generated before. Using this method, the 'full-explorability' is ensured.
Two different algorithms to generate mazes have been implemented:

- **Maze()**, which generates labyrinths randomly picking an empty space and creating a small number of explorable squares in a random direction starting from that picked space. It results in a map characterised by thicker paths, which can occasionally be considered as rooms

- **Maze2()**, which creates labyrinths randomly creating a small number of explorable squares in a random direction starting from the last space created. It results in a map characterised by thinner paths, that makes the map remind more of a typical maze

The Class **Maze()** was utilised firstly, but then it was decided to use **Maze2()** algorithm as it depicts a maze better.
Both classes are composed by three functions:

- **__init__(SIZE, ITERS)**, as soon as the object of class **Maze** is initialised the environment is created, the global variable **grid_map** represents the **maze**. It starts as a **np.ones** matrix and it is then crafted in a **for loop** where, for a fixed number of iterations, a random direction is selected by picking a random key of the dictionary **moves**, which depict the possible directions to choose.
  At the end, an **updated** version of the **map** is created. This variable will be manipulated by the **agent** and that is useful to detect what percentage of the map is explored.
  The **init** receives **two inputs** which determine the **size** of the map and the number of **iterations** used in the for loop to create the maze. This second variable has been added since it cannot be the same for every size and finding a formula to determine it it is not trivial

- **print_map()**, as the name suggests, this function prints the map by using the python library **matplotlib.pyplot**, the colour palette is derived by **viridis**. Hence, the lightness value increases monotonically through the colourmaps, since it is used for sequential plots

- **print_updated_map()**, which prints the updated version of the map, i.e. the map with the knowledge of where the agent has been throughout the simulation. It is useful for understanding how the training is going

Algorithm 5: Maze

```
Data: size, iters
maze ← ones(size, size);
empty_spaces = [];
maze[size//2, size//2] ← 2;
empty_spaces.append([size//2, size//2]);
for iters do
    space ← pick_a_space(empy_spaces);
    move ← pick_random_move;
    for 10 do
        x, y ← space + move;
        maze[x][y] ← 2;
        space ← [x, y];
        empty_spaces.append(space);
    end
end
```

Algorithm 6: Maze2

```
Data: size, iters
maze ← ones(size, size);
empty_spaces = [];
maze[size//2, size//2] ← 2;
empty_spaces.append([size//2, size//2]);
for iters do
    move ← pick_random_move;
    for 5 do
        x, y ← space + move;
        maze[x][y] ← 2;
        space ← [x, y];
        empty_spaces.append(space);
    end
end
```

5.2 Agent

5.2.1 Reinforcement Learning Model

RL problems are typically modelled as Markov Decision Processes as it has been done for this system.
In this case, the MDP elements are the following:

- **State**: at each time instant, the agent has access to a portion of the explored global map, namely a local map centred on the agent. Each cell of both the global and local maps is associated with a value indicating accessible, unknown, and occupied spaces, in addition to the agent's location.
  It is indeed a **grid map** of size $\mathbf{N \times N}$ **matrix** with values from 0 to 1, since it will be the input of the neural network and normalised values are easier manageable.

- **Action**: the policy has to choose the next navigation goal from a cluster of pixels on the local map. These pixels are grouped in larger spaces in order to reduce the action space and make training and deployment of the network more efficient.

- **Reward function**: this function incentives exploration based on the number of new spaces the agent has explored during the previous action and the number of steps it took; each step involves a penalty. It also penalises the agent when an unfeasible action is taken (e.g., selecting a location containing only unknown pixels or walls).
  The reward function is defined as:

$$R(new\_squares, steps) = \begin{cases} penalty & \textbf{If } action \textbf{ is not } feasible \\ new\_squares - steps & \textbf{otherwise} \end{cases}$$

  where:

  - **penalty** is the negative reward given when the agent selects a wrong action
  - **new_squares** are the formerly unseen places detected by the agent during the move
  - **steps** are the squares the robot has been during the action

  - **step_penalty** is the constant value that determines the negative rate for every step made during the move

- **start state**: initially, the agent is spawned at a randomly accessible location in the maze. The whole environment is unknown to the agent except the area reached by the simulated 2D LiDAR. The policy has to choose a space out of the accessible, known locations on this local grid map that constitutes the next navigation goal. When a feasible action is selected, an A-star algorithm finds the optimal path to reach the selected goal. In such a way, the agents explore the environment while creating the global, explored map.

- **terminal state**: the agent ends its run if it encounters a dead end; it occurs when there is nothing left to explore on the local map, which means that none of the accessible spaces has any unknown space nearby. In this case, the global planner is invoked. The global planner is a deterministic system that recovers the navigation by selecting the nearest accessible square with at least one unknown pixel nearby on the global explored map. From there, the local planner can start exploring again. Another terminal state, which terminates both the local and the global planner, arises when a fixed percentage of the maze has been explored.

5.2.2 Python Class

In order to create this Reinforcement Learning model, a Python class called "**Agent**" has been developed.
The class is featured by a considerable number of functions and inputs. Hence, a description of the parameters that characterise the agent is important to understand the development of the code.
As it can be clearly seen from the image, the **local map** is made up of the following parameters:

- **View**, which is the LiDAR's field of view. It regulates the size of the real local map.

- **Expand**, that selects the number of pixels to add to the local map. This influences the agent's knowledge since it determines how big the

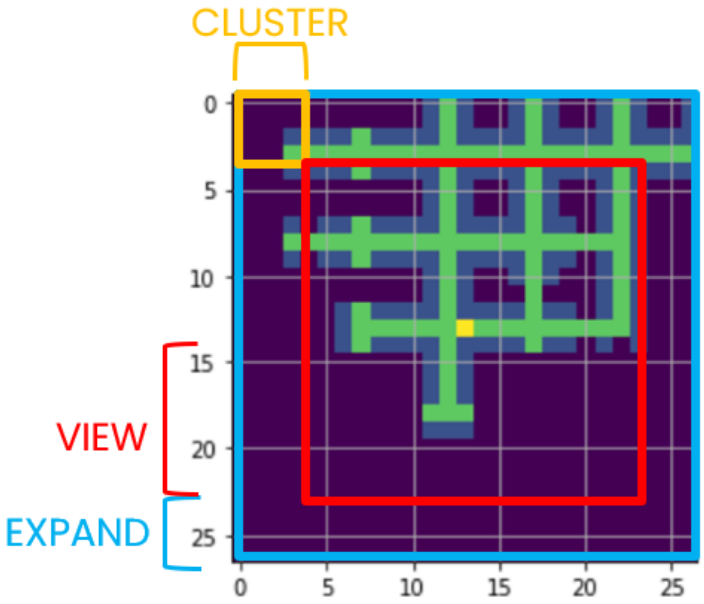


Figure 5.1: Agent's local map showcasing essential components

part of the global map to examine will be

- **Cluster**, which is the size of the square that groups up the pixels. It is of paramount importance since it stabilises the training of the neural network because it reduces the number of outputs.

It is worth noting that the map size created by the parameters View and Expand has to be **divisible by** the number chosen for the **Cluster** variable:

$$Map = 2V + 2E + 1 \qquad \forall V, E \in \mathbb{N} : \ C | Map$$

where **Map** is one size of the map and **V**, **E** and **C** are respectively the values of View, Expand and Cluster.
In addition to these parameters, there are other inputs:

- **maze**, the agent receives the environment where it is located so that it can interact with it.
- **initial position**, that is the location where the agent spawns, it is randomly selected among all the possible empty spaces of the map.
- **maze_squares**, which is the number of empty spaces present in the map. This parameter is useful to make the agent understand when it has explored a percentage of the maze.
- **percentage**, which determines the percentage of the map the agent has to explore.

- **alpha**, that is the coefficient aforementioned when the reward function was explained. It is needed to stabilise the training.
- **penalty**, that determines the negative reward given to the agent when it selects an unavailable action.
- **step_penalty**, that is the negative reward given for every step made by the agent during an action.

Now that the input parameters are described, the functions of the agent's class can be explained.
During the **initialisation** all the dynamic parameters are set to default, the robot position is set to the centre of the map and all agents map are created as a matrix of zeros (i.e. unknown spaces).
It is also called a function that 'cleans' the map of the maze's class from the updates done by a previous agent, this will be useful for the **evaluation** part where a Greedy agent and a trained one will explore the same map.
A **terminal function** that detects when the agent is in a **terminal state** checks if the robot is in a **Dead end** or it has explored all the map or a percentage of it.

Move functions

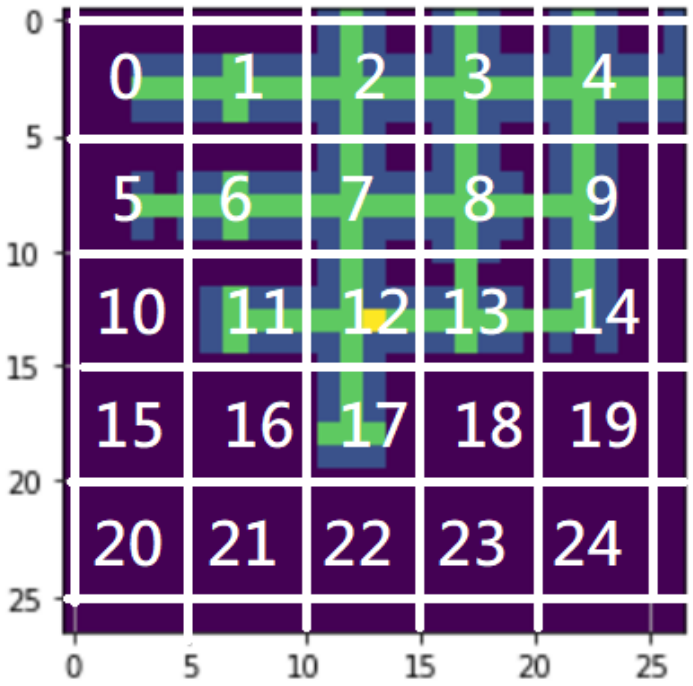


Figure 5.2: Example on how the actions are numbered

The main functions of agent's class are **move(action)**, **greedy_move(action)**

and **local_greedy_move(action)**.
The first one basically receives an action and, since this value refers to a set of pixels and these sets are ordered in rows as shown in the figure below it converts this value into the set of coordinates it corresponds to.
This is done converting firstly the action value into the corresponding coordinate set that defines that particular pixel cluster. Then the algorithm searches the most promising pixel in the cluster to go to.
This operation is explained by the following pseudo-code.

Algorithm 7: Action search

```
Data: action, cols, dim
% Creating cluster coordinates;
x ← action//(cols/dim);
y ← action − x · (cols/dim);
% Converting them into map's coordinates of the first
 pixel of the cluster;
X ← x · dim;
Y ← y · dim;
for i in range(X,X+dim) do
    for j in range(Y,Y+dim) do
        path ← A_star(i, j);
        if is_promising([i,j] then
            break;
        end
    end
end
return [i, j], path
```

where **cols** and **dim** are respectively the number of columns of the local map and the dimension of the cluster.
The **A_star** function searches the optimal path to arrive in a pixel, whilst **is_promising** detects if that pixel has an unknown square in its surroundings, if it does, the for loop will stop as it has found a promising place to go. It is worth noting that the search algorithm converts the coordinates **[i,j]** from local to global.
After this algorithm, the move function checks if there is an available path to reach one of the cluster's pixels (i.e. **len**$(path) \neq 0$).
If the **action** is **wrong**, the function returns a **penalty**. Otherwise, it

counts every new square detected during the action by using the function **scan_area** and every step made during the movement and returns the evaluated **reward**.

Algorithm 8: Move

```
Data: action, cols, dim
[i, j], path ← Action_search(action, cols, dim);
if len(path) is 0 then
    return penalty
end
mx ← 0;
my ← 0;
reward ← 0;
penalty ← len(path) · step_penalty;
while len(path) ≠ 0 do
    step ← path.pop(0);
    step ← step + [mx, my];
    δmx, δmxy, δrew ← scan_area(step);
    mx ← mx + δmx;
    my ← my + δmy;
    reward ← reward + δrew;
end
return reward · α
```

As it can be clearly seen, the move function extrapolates every single step that composes the path and evaluates the reward summing the contribution of each step.
The values **mx** and **my** are necessary to consider how the global map has changed after each step, so that the coordinates of the following steps can be updated. As said before, there are two other move function:

- **greedy_move**: it is the function used by the **Greedy Global Planner**, which selects the closest explorable square from the **Global Map** with an unknown pixel nearby by using the A_star function. The **heuristic** used to define the closest place is the **Euclidean distance**.

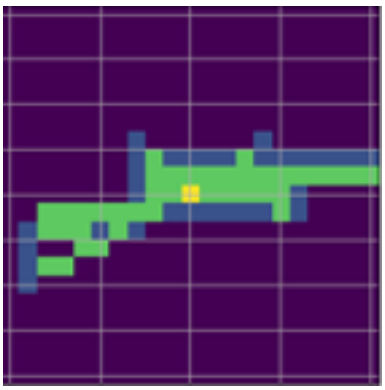
- **local_greedy_move**: this function is necessary for the **evaluation** part, since it used by the **Greedy Local Planner**, which selects the

closest explorable square from the **Local Map** with an unknown pixel nearby by using the A_star function.
The **heuristic** used to define the closest place is the **Euclidean distance**.

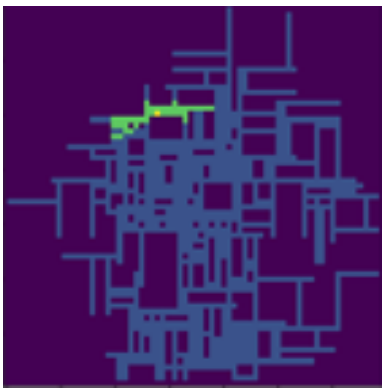
Path Planning

When an action is selected, the agent finds the optimal path to arrive in that place with an **A_star** algorithm.
It is **optimal** because the heuristic utilised is the **Euclidean distance**, which provides both **admissibility** and **consistency** since it is the minimum distance between two points, satisfying both equations (2.2) and (2.3).

5.2.3 Mapping



((a)) Local Map



((b)) Global Map

Another function of paramount importance is **scan_area**, which is essential to update the global and local maps and count all the squares scanned by the agent.
Agent's maps have a wider range of values than those of the maze class. Thus, in addition to the values described previously, there are two new type of pixels:

- **maze**[**x**][**y**] = **0**, it represents an **unknown** square, it can be either an explorable place or a wall but the agent does not know. This type of pixel is the key to evaluate the quality of actions for the training agent and the quality of explorable squares for the greedy agent. Since an

action that allows the robot to see a considerable number of unknown pixels is decent and an explorable square with unknown pixels nearby is a promising place to go.

- **maze**[**x**][**y**] = **4**, it is regarded as the **agent's** square. It is more of a visual thing since the robot is always in the middle of the local map, but it is useful to know where the agent is in the global map

It is worth noting that neither in the local nor global map pixels with the value of 2 are present, considering that if an explorable square is seen by the agent it automatically becomes an explored square by definition.
It is necessary to explain that the robot utilises a simulated 2D LiDAR to examine the area. An algorithm generates 360 'beams' (one for each degree); each beam has a particular angle that will determine what square the agent is going to examine. The beam continues to scan the map until it detects a wall (i.e. a pixel of value 1). In this case, the agent will receive the information about the wall but the scanning performed by that beam will not go further.
Since the agent does not have any knowledge about the size of the maze,

Figure 5.3: LiDAR 360° 2D Laser Scanner [11]

the global map changes dimensions as the robot explores the environment. In order to do that, the algorithm will just change the size of the map's matrix if the agent goes either to the eastern or southern borders of the map, whilst it will additionally change the coordinates of what it has already scanned if the robot moves to the western or northern border considering that the newly gathered knowledge is added in the first indexes of the ma-

trix.
After the agent has finished scanning the area, it cuts a part of the global map that surrounds its position and it uses it to update the local map.

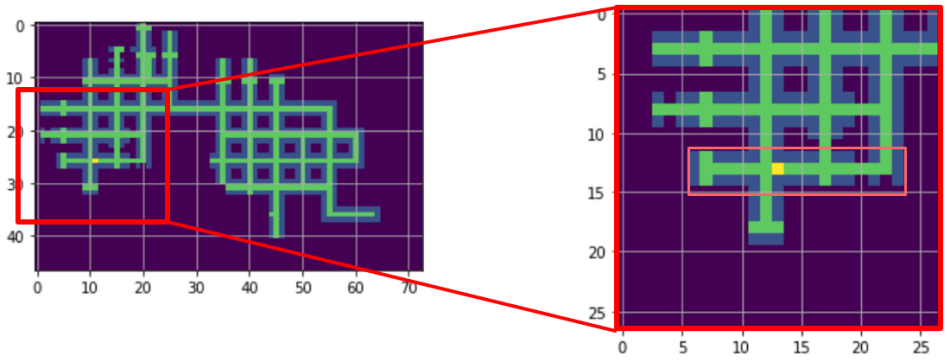


Figure 5.4: Global to local map

5.2.4 Decision Making

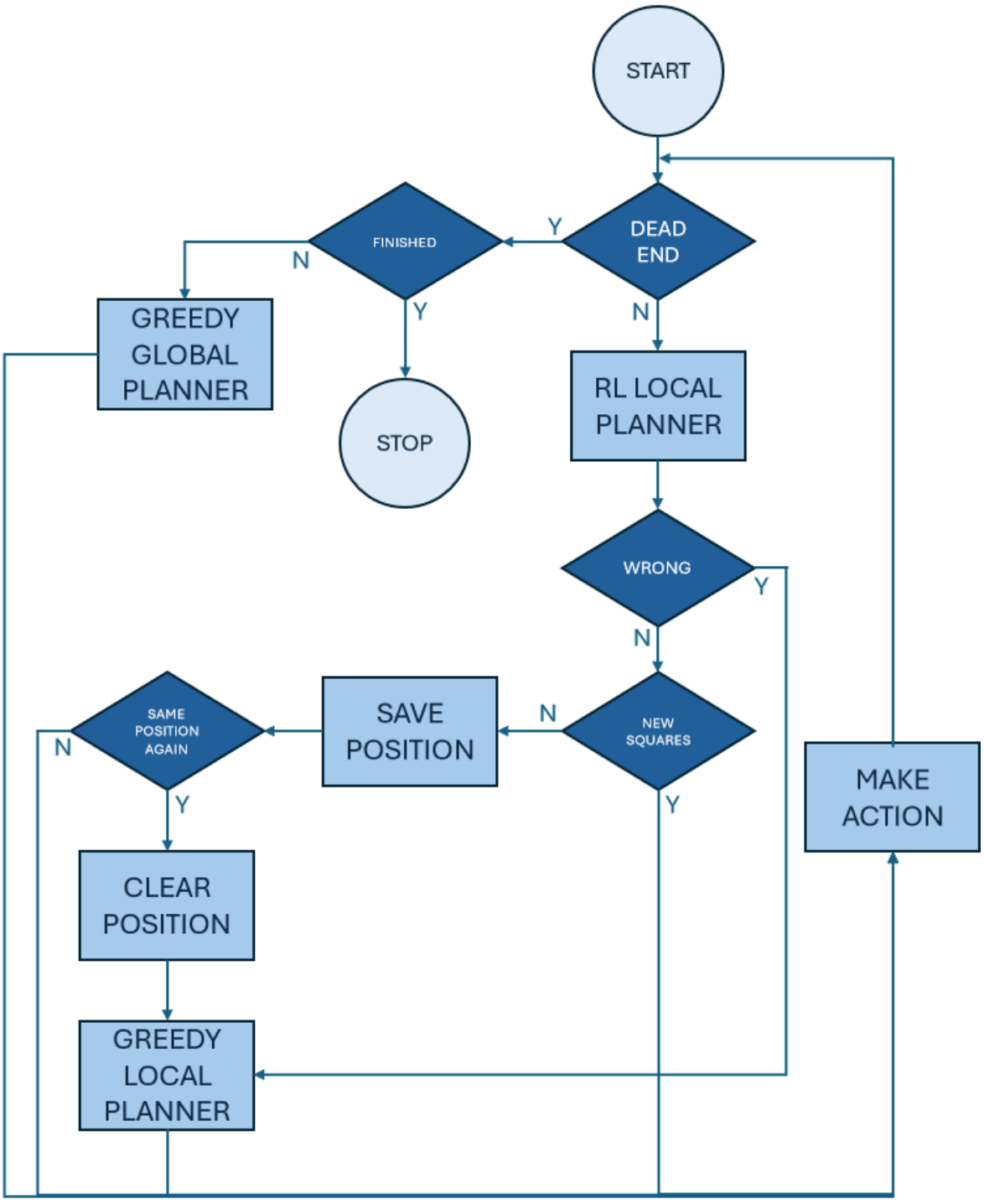


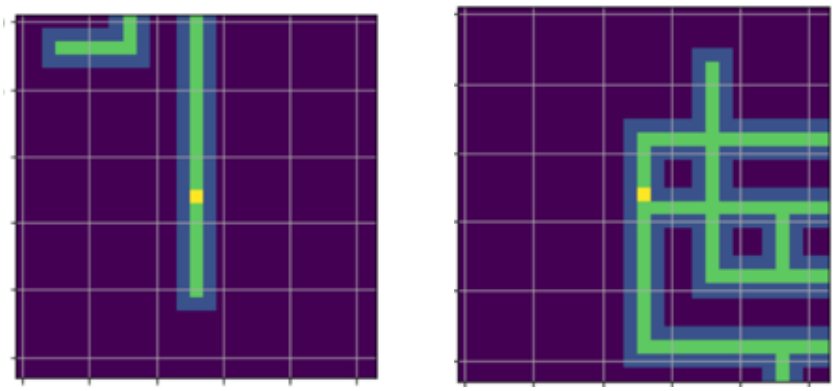
Figure 5.5: Decision Making's Flow Chart

Agent's decision making is well depicted by the flowchart above.

The entire process can be described by the following steps:

- The Trained Local Planner selects the next action.
- The algorithm checks if the action is available. If it is not, the next action is now chosen by a Greedy Local Planner and executed.
- If the action is feasible, the agent controls if new squares have been scanned. If they have not, it saves the position and checks if this saved position was already encountered. If this happens, it means the robot is moving in a loop as it continues to encounter the same maps. Hence, since it selects the action with the highest Q-value, it will always choose the same actions. In short words: same maps means same actions.
  In order to avoid that, if a loop is detected, a Greedy Local Planner picks the next action and the loop positions are deleted because that particular loop has been avoided.
- Now, in the case the action is available and a loop is not detected, the robot makes the action.
- After the robot has concluded an action, the algorithm checks if the robot has explored enough space to call the terminal state of the Global Planner. If it has yet to explore, it examines if it detects a Dead End. If it does, the agent calls the Greedy Global Planner that will decide the next move. Otherwise, the process starts again choosing an action with the Trained Local Planner.

A **Dead End** is detected when the agent cannot see any promising place in its Local Map. In technical words, every explorable square in the local map (i.e. the pixels with value 3) has not an unknown square in its surroundings.



((a)) ((b))

Figure 5.6: Example of Dead ends

Algorithm 9: Dead End detection

```
Data: Map
for i in len(Map) do
    for j in len(Map[0]) do
        if (Map[i][j] is 3 and find_zeros(i,j,Map)) then
            return False
        end
    end
end
return True
```

The function **find_zeros** is very useful and it is indeed utilised for various functions (from the dead end detection to the greedy algorithm) and it simply checks if a square has an unknown pixel nearby.

5.3 Training and Evaluation

The agent is trained using a Double DQN algorithm as it is perfectly suitable with discrete state and action spaces. Moreover, a model-free method is necessary for this kind of problems since it is impossible to predict what the next state will be given the complexity of the state. Hence, it is only possible to learn what is the optimal action to perform in a specific state.
Considering that in this MDP model a state is a matrix that represents a map, Convolutional Neural Networks are used to manipulate the states

Algorithm 10: find_zeros

```
Data: x, y, Map
for i in range(x-1,x+2) do
    for j in range(y-1,y+2) do
        if (i ≥ 0 and i < len(Map) and j ≥ 0 and j < len(Map[0])) then
            if Map[i][j] is 0 then
                return True
            end
        end
    end
end
return False
```

and generalise them. Specifically, two layers of CNN with 64 filters each of dimension 3×3 and stride respectively 1 and 2 have been developed. The CNN is then flattened and linked to a final fully-connected layer that represents the output.

5.3.1 Parameters and Hyperparameters

The purpose of the training is to find the optimal combination of parameters in order to obtain the best achievable performance from the agent. The training is interspersed with phases in which the performance of the network is evaluated for several steps. The evaluation is useful to understand how the agent is learning and perhaps it helps finding after how many steps the performance of the training saturates.
Before the actual training starts, there is a warm-up of a considerable number of steps to familiarise the agent with the environment.
The variables that affect the training and has to be tuned are the following:

Parameters

- **ALPHA:** the coefficient that multiply the reward of each action. It is useful to stabilise the training.
- **CLUSTER:** the size of the square that encapsulates the pixels. As said before, it is necessary to reduce the action-space. Hence, it decreases the number of outputs of the neural network.
- **EXPAND:** it determines how many squares of the global map the agent can see in addition to what it can already see from its sensor. The higher is this value the more difficult the training will be.

- **PENALTY:** the negative reward for choosing an unfeasible action.
- **PERCENTAGE:** the percentage of the maze the agent has to explore to finish its task.
- **SIZE:** the size of the map.
- **VIEW:** it determines how far the agent can see.

Hyperparameters

- **batch_size:** the number of experiences (or transitions) sampled from the replay memory to update the neural network in a single iteration. It influences the stability and efficiency of the learning process.
- **collect_steps_per_iteration:** the number of steps collected in one iteration.
- **conv_layer_params:** a tuple containing the parameters to create the CNN structure.
- **decay_steps:** the number of steps the training needed to gradually change epsilon from its initial value to its final.
- **epsilon:** a vector containing the initial and final values of epsilon (i.e. the coefficient for performing the $\epsilon - \textbf{greedy}$ algorithm)
- **eval_interval:** it does not affect the training since it defines after how many steps an evaluation takes place during the training.
- **gamma:** the discount factor of the MDP.
- **initial_collect_steps:** the number of steps performed by the agent during the warm-up.
- **learning_rate:** the learning rate of the training to update the neural network.
- **log_interval:** the number of steps between one calculation of the loss function and another.
- **number_of_iterations:** the number of iterations performed during a training.
- **num_eval_episodes:** the number of simulations to evaluate the performance of the agent during an evaluation.
- **replay_buffer_max_length:** the dimension of replay buffer. It is determined by the number of steps performed during the training.
- **tau:** the coefficient derived from the Double DQN algorithm that determines how fast the target network is updated.

5.3.2 TensorFlow's library

The library used to develop the training code is TF-Agents [17]: a reliable, scalable and easy to use TensorFlow library for Contextual Bandits and Reinforcement Learning.
The training code has been inspired by the following tutorials from the aforementioned GitHub repository:

- **dqn tutorial**, which explains how to use the library dedicated to the DQN algorithm. It is worth noting that the DQN is actually a Double DQN, since it is a better version independently from the task.
- **environments tutorial**, which is important to understand how to create a environment wherein the policy selects an action. Especially in this case, where the variables need to be of a specific type, compatible with the TF-Agents libraries.
- **reinforce tutorial**. This is the skeleton of the code implemented in the training part of this thesis' project. The type of RL algorithm has been changed (a DQN method has been used instead).
- **checkpoint policy saver tutorial**, which was essential to save the trained policy in a zip file.

5.3.3 Evaluation

After the training, the neural network is evaluated comparing its performance with whose obtained by a completely greedy agent.
Firstly, the network is uploaded and unzipped from a zip file and then the algorithm creates a maze which is explored by the trained agent and the greedy one. This operation is computed for several iterations.
After that, the time spent for the simulation and the reward obtained are compared evaluating the ratio between the trained agent and the greedy one (i.e. $\frac{RL\ agent}{Greedy\ agent}$).
The time is evaluated using the **time** library from Python.
Therefore, performance is evaluated in terms of results and time consumption to compute the action.
The agent has to explore the 80% of the map to conclude its task.

Algorithm 11: evaluation

```
Data: maze_params, agent_params, dirname
policy ← unzip(dirname);
for _ in range(10) do
    maze ← Maze2(maze_params);
    init_pos ← maze.empty_spaces[random];
    DRL_agent ← Agent([agent_params, init_pos]);
    start ← time.time();
    while not DRL_agent.explored() do
        % Local Planner;
        while not DRL_agent.dead_end() do
            DRL_agent.move(policy);
        end
        if not DRL_agent.explored() then
            % Global Planner;
            DRL_agent.greedy_move();
        end
    end
    end ← time.time();
    DRL_time.append(end-start);
    DRL_reward.append(DRL_agent.total_reward);
    Greedy_agent ← Agent([agent_params, init_pos]);
    start ← time.time();
    while not Greedy_agent.explored() do
        % Local Planner;
        while not Greedy_agent.dead_end() do
            Greedy_agent.local_greedy_move();
        end
        if not Greedy_agent.explored() then
            % Global Planner;
            Greedy_agent.greedy_move();
        end
    end
    end ← time.time();
    Greedy_time.append(end-start);
    Greedy_reward.append(Greedy_agent.total_reward);
end
reward_ratio ← mean(DRL_reward)/mean(Greedy_reward);
time_ratio ← mean(DRL_time)/mean(Greedy_time);
```

Chapter 6

Results

In this chapter, the main results obtained and the failed attempts during the development of this project are presented.
These results are mainly derived from the tuning of the parameters showed in the previous chapter and the utilised method. Therefore, a clear explanation of why some variables has been set to a fixed value is given, in addition to some suppositions about the reasons why some parameters affected in that particular way the training.

6.1 Global Planner (failed)

The first attempt was the most ambitious: training the agent on its global map. In this case the robot would be really able to make decisions on the entire map it built throughout the exploration.
However, the input was too big and complex to manipulate and the neural network was not capable of generalising the map and learning a decent policy.
A couple of failed attempts are shown to clarify the failure.
It is noteworthy that the training is valued on the average reward obtained during the training's evaluation steps, and it can be clearly seen that the reward is always below 0.

6.2 Local Planner

As mentioned in Chapter 5, the final version of the agent is trained to make decisions on its Local Map.
The results obtained from the first trainings were promising since the av-

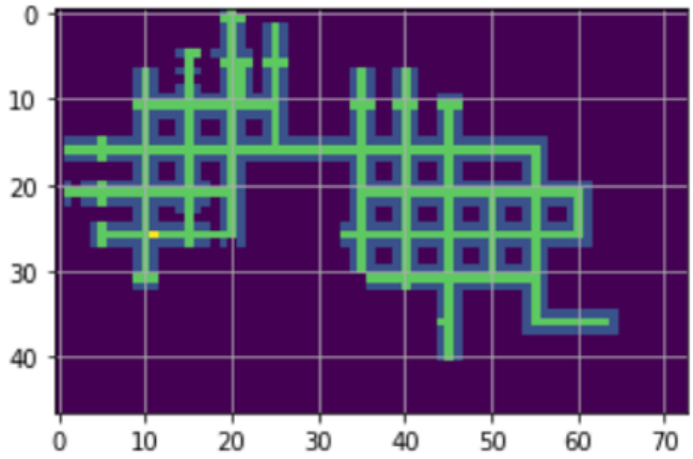
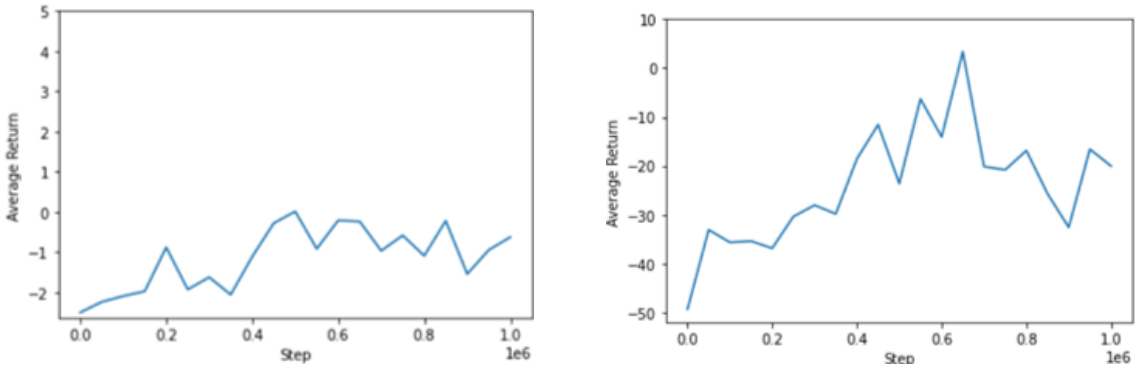


Figure 6.1: Agent's global map



((a)) ((b))

Figure 6.2: Two failed trainings

erage return increased to an acceptable value where it saturated after approximately 500.000 iterations.

6.3 Fixed Parameters

Some of the parameters and hyperparameters shown in the previous chapter have a fixed value for different reasons that will be explained later in this section. These value are depicted by the following table.

Penalty	**Step Penalty**	τ	γ	**Steps per Iteration**	**Iterations**
10	1	1	1	5	$5 \cdot 10^5$

Table 6.1: Fixed Parameters and Hyperparameters

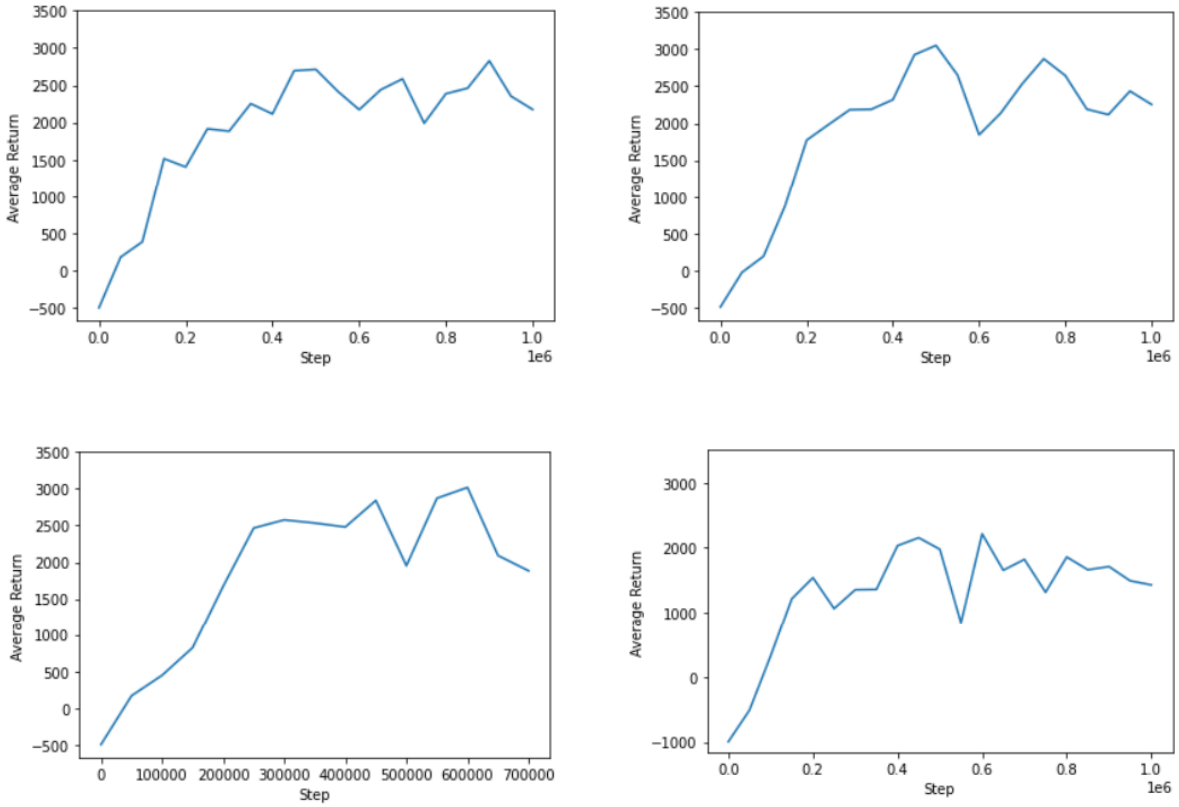


Figure 6.3: Successful trainings with Local Planner

This variables has these specific values for the following reasons:

- **Penalty**. This value is able to both maintain the stability of the training and make the agent understand to choose available actions.
- **Step Penalty**. A couple of attempts were made changing this value from 1, but during every training the agent was stuck in selecting the place it is. Probably because the penalty different from 1 is too big and the robot thinks that the best thing to do is staying still.
  The trainings show that the agent learns to do only the first actions.
- τ. It is set to 1 since the target network needs to learn quickly, otherwise the network will take too much time to just learn to avoid unfeasible actions.
- γ. It cannot be less than 1 or the agent will start staying still as the step penalty gets closed to 0, since it diminishes after every step.

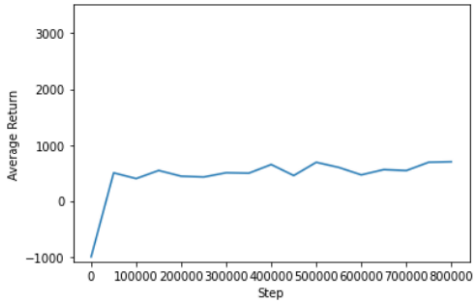


Figure 6.4: Training with Step Penalty = 2

- **Steps per Iteration**. After several attempts, 5 steps per iteration seems enough to make the agent understand to choose the available actions and learn a good policy.

- **Iterations**. Since after the first attempts the return obtained saturates after 500.000 iterations. In order to save time and memory the number of iterations has been set to this value.

The ϵ value for the ϵ-greedy algorithm starts with 1 (the policy explores the environment with completely random actions), then it decades to 0 after 250.000 iterations. Therefore, the policy solely exploits its learned knowledge from half training.

6.4 Promising Trainings

The agent has been trained with different setups.
Here are reported the most successful training attempts for each map dimension. For all of them, the agent had to explore the entire map to achieve its task. It is worth noting that the last two columns refer to the

α	SIZE	VIEW	EXPAND	MAP	CLUSTER	REWARD	TIME
10^{-4}	200	11	11	45×45	5	0.88	0.67
10^{-4}	250	12	15	55×55	5	0.82	1.96
$7 \cdot 10^{-5}$	300	12	20	65×65	5	0.84	2.48
$3 \cdot 10^{-5}$	350	12	25	75×75	7	0.80	2.94

Table 6.2: Promising Trainings

evaluation part results, both values are the solution of the ratio between the reinforcement learning agent's results and the one obtained by a greedy agent.

As it can be clearly seen, even though these agents trained successfully, only the first one obtained good results, as it takes less significant time to obtain a decent reward.

6.5 Final Agent

Taking into consideration the results obtained in the previous section, various agents were trained with different hyperparameters maintaining the same local map's proportion and agent's setup of the first training from the previous table.

Table 6.3: Hyperparameters values used during the training.

Learning rate	**Initial ϵ**	**Final ϵ**	**ϵ decay steps**	τ	γ	**Steps per Iteration**
10^{-4}	1	0	$2.5 \cdot 10^5$	1	1	5

After numerous attempts, a trained agent was able to gain 91% of greedy's reward in less than half the time (44% of the time spent by the deterministic agent).
It is worth noting that this agent has been trained giving the task of exploring 80% of the maze.
The agent has then been trained on different maze size to verify its efficiency.

Table 6.4: Performances comparison between a traditional greedy algorithm and our RL agent. Column *Maze size* report the lateral dimension of the square containing the maze, while the last two columns report the ratios between the RL agent's and greedy agent's results for both the final reward and the computation time. Every time is expressed in seconds.

Maze size	**Greedy reward**	**Greedy time**	**RL reward**	**RL time**	**Reward ratio**	**Time ratio**
200	9359.0	86.94	8516.7	38.25	0.91	0.44
250	12014.1	134.42	10212.0	68.55	0.85	0.51
300	16872.6	244.41	13835.5	153.97	0.82	0.63
350	20272.5	291.14	16420.7	205.98	0.81	0.71
400	27694.7	451.2	21324.9	591.07	0.77	1.31

The tests are conducted on mazes featuring different sizes; in column *Maze size* is reported the lateral dimension of the square containing the maze, in

pixels. The last two columns report the ratios between the RL agent's and greedy agent's results for both the final reward and the computation time. As can be seen, the RL agent can reach reward values very close to the near-optimal greedy algorithm in much less computational time. However, performance worsens with increasing maze size; this is also because both agents share the same greedy global planner, which needs more time when the map is larger. Hence, as the maze becomes bigger, the computation time is influenced more by the global planner's decision than the local planner's.

Chapter 7

Conclusions

Results obtained can be considered acceptable as they satisfy the request of the thesis. However, there are plenty of changes and improvements to be implemented in this work.
First of all, to obtain more realistic results, the environment could be simulated on Gazebo and the code could be implemented on a robot, such as a Turtlebot3.
The most important improvement would be applied in the learning process. Training an agent with a constant set of parameters and hyperparameters has a lot of limitations, therefore I believe utilising Curriculum Learning algorithms would address the problem in a much better way since it fits perfectly with this kind of environment.
The algorithms presented in Narvekar et al. (2016) [14] could be utilised to create a dynamic training that would definitely improve agent's performance.
The implementation of a Markov Decision Process for the curriculum design agent (CMDP) [9] to structure the task sequence automatically is another interesting enhancement for this project. These systems could be trained with a recursive Monte-Carlo method or a Continuous Space Representation [13].
In this project, the Double DQN network was made on CNNs and a flatten layer which is then fully connected to the output layer of the actions. However, utilising a global map as input would help the agent to be more versatile and efficient since it still uses a greedy global planner. Hence, there are two possible solutions: resize the input map to a fixed size, but it would cause uncertainty problems, or utilise a Fully Convolutional Network (FCN) [6] since it is appropriate for pixel-wise agents and it has already been used in Unknown Environment Exploration by Li et al. (2020) [4],

which created an FCQN: a Fully Connected Q Network enhanced with a Dueling Network algorithm [20].

Bibliography

[1] Cs231n convolutional neural networks for visual recognition. biological motivation and connections. https://cs231n.github.io/neural-networks-1/.

[2] GeeksforGeeks. Deep q-learning. https://www.geeksforgeeks.org/deep-q-learning.

[3] Arthur Guez Hadovan Hasselt and David Silver. Deep reinforcement learning with double q-learning. *Google DeepMind*, 2015.

[4] Qichao Zhang Haoran Li and Dongbin Zhao. Deep reinforcement learning-based automatic exploration for navigation in unknown environment. *IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS, VOL. 31, NO. 6, JUNE 2020.*

[5] Yoshua Bengio Ian Goodfellow and Aaron Courville. Deep learning. http://www.deeplearningbook.org, 2016.

[6] Evan Shelhamer Jonathan Long and Trevor Darrell. Fully convolutional networks for semantic segmentation. *arXiv:1411.4038 [cs.CV]*, 2014.

[7] Dan Klein and Pieter Abbeel. https://maplearn.readthedocs.io/en/latest/maplearn.ml.html.

[8] Dan Klein and Pieter Abbeel. Cs188 intro to ai at uc berkeley. https://ai.berkeley.edu.

[9] Sanmit Narvekar and Peter Stone. Autonomous task sequencing for customize dcurriculum design in reinforcement learning. *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[10] Sudharsan Ravichandiran. Hands-on reinforcement learning with python. https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788836524.

[11] Robopeak. Lidar 360 2d laser scanner 6m range 5.5hz.

[12] Sumit Saha. A guide to convolutional neural networks — the eli5 way. https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/.

[13] Jivko Sinapov Sanmit Narvekar and Peter Stone. Learning curriculum policies for reinforcement learning. *International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Montreal, Canada*, 2019.

[14] Matteo Leonetti Sanmit Narvekar, Jivko Sinapov and Peter Stone. Source task creation for curriculum learning. *International Conference on Autonomous Agents and Multiagent Systems (AAMAS), Singapore*, 2016.

[15] David Silver. Lecture 4 of ucl course on reinforcement learning.

[16] Richard S. Sutton and Andrew G. Barto. Reinforcement learning. *TUGBoat*, 14(3):342–351, 1993.

[17] TF-Agents. https://github.com/tensorflow/agents.

[18] Toussiant. Model-free and model-based reinforcement learning. 2010.

[19] David Silver Alex Graves Ioannis Antonoglou Daan Wierstra Volodymyr Mnih, Koray Kavukcuoglu and Martin Riedmiller. Playing atari with deep reinforcement learning. https://arxiv.org/abs/1312.5602, 2013.

[20] M. Hessel H. Van Hasselt M. Lanctot Z. Wang, T. Schaul and N. De Freitas. Dueling network architectures for deep reinforcement learning. *Proc. IEEE Int. Conf. Mach. Learn. (ICML), Jun. 2016.*

[21] J.T. Vanderplas Ž. Ivezić, A.J. Connolly and A. Gray. Statistics, data mining and machine learning in astronomy. *Princeton, NJ: Princeton University Press,*, 2014.