Multi-Agent Reinforcement Learning with Gibbs Distributions

by Myles Alexander Foy
Stephenson College

Supervisor Professor Ian Jermyn

A dissertation submitted for the degree of
BSc Mathematics and Statistics

1 May 2026

Plagiarism declaration

This piece of work is a result of my own work and I have complied with the Department's guidance on multiple submission and on the use of AI tools. Material from the work of others not involved in the project has been acknowledged, quotations and paraphrases suitably indicated, and all uses of AI tools have been declared.

I, Myles Alexander Foy confirm that I have read, understood, and have adhered to the plagiarism declaration above.

Declaration of use of AI tools in the writing process

Location	Explanation of AI use	AI used
Page 2	Generated the TikZ code for Figure 2.1. The design and idea behind the figure are entirely my own. I had not previously used TikZ and was unsure where to begin, which is why I used Gemini to create a simple example. All other TikZ figures in this report were written by me with the help of the TikZ documentation.	Gemini 3

List of Figures

1.1 A rendering of 30 agents flocking amongst 50 obstacles (most of which are off-screen). The full video rendering can be found here. 2

2.1 A graphical model for Example 2.1. Edges represent being at war. 3

2.2 Pairwise factors for Example 2.1. 4

2.3 Factors over the 3-clique in Example 2.1. 4

2.4 Pixels in an image can be modelled as a graph. We infer that X is likely to be red, like its neighbours. 6

3.1 A Markov decision process. Our agent (the robot) interacts with its environment (the globe) by taking an action, a. This moves the robot from its current state s, to a new state, s', and the robot receives a reward, r. In an MDP, the agent always knows what state it is in. Image from Oliehoek and Amato (2016). . . . 10

3.2 An illustration of Example 3.1. 12

3.3 Treasure hunting continued. The values of each state under the policy described in Example 3.2 are shown in the grid on the right. 16

3.4 Synchronous data collection. n environments are run in parallel, and pass their rewards and next states to the agent. The agent then sequentially decides on rewards for each environment, and the environments receive these actions and compute the next set of rewards and states. Image is from Albrecht et al. (2024). 26

4.1 A Partially Observable Markov decision process. The situation is the same as in Figure 3.1, except now the agent receives a noisy and/or incomplete observation, o, of the new state, s', rather than observing the state itself. Image from Oliehoek and Amato (2016). 30

4.2 A Decentralised Partially Observable Markov decision process. The situation is the same as in Figure 4.1, except now we have multiple agents interacting cooperatively in the same environment. Each agent has the same reward function. Image from Oliehoek and Amato (2016). 32

4.3 Box pushing. (a) Robots R1 and R2 move in a grid to push the small box B and large box BB (which spans two cells), onto the goal, G. The small box can be pushed by a single robot; the large box must be pushed by both robots at the same time. (b) When a robot moves (or robots move) into a box, the box moves in the same direction. 33

4.4 Box pushing continued. Due to each robot only partially observing the state, the robots are unable to see each other and may collide while trying to get to the box, B. 35

5.1 Actions and obstacle detection. (a) illustrates actions in $\mathcal{A}_{i,d}$. (b) illustrates how $\boldsymbol{o}_{io}$ is calculated using LIDAR measurements. Figure is from Zhang et al. (2025). 41

5.2 Position alignment energy. $d_r = 0.6, c_{p1} = 0.03, c_{p2} = 1$. When the robots maintain their desired separation, the energy is at a minimum. 43

5.3 A demonstration of the velocity alignment energy. Two robots move such that their relative velocity opposes their relative position, i.e., they are moving closer together, and are thus penalised by the velocity alignment energy. ($\boldsymbol{v}_{ji}$ not to scale). 44

5.4 A demonstration of how more edges can degenerate collision avoidance. (a) shows a configuration with 14 edges, whereas (b) shows the same configuration with 18 edges. The red robot is pulled towards the centre of its neighbourhood, but this degenerates its ability to avoid the obstacle in the centre of the flock. Image from Zhang et al. (2025). 46

6.1 Graphs of the mean total return over all training episodes for all training configurations. The graph on the right is for 30 agents and 50 obstacles. 52

6.2 A rendering of 30 agents flocking amongst 50 obstacles (most of which are off-screen). Actions are represented by long blue lines protruding from the agents. The black lines with dots on the end show the rays of the LIDARs. The target is in light green on the right, and moves continuously to the right. At the bottom, an agent is moving towards an obstacle, but as shown in the full video rendering (which can be found here), the agent will move around it once it gets close enough for its LIDARs to detect the obstacle. 52

List of Tables

6.1 The flocking order under the simple method is evaluated for various numbers of robots and obstacles, and compared with the PPO-AA model from Zhang et al. (2025). 53

6.2 The success rate under the simple method is evaluated for various numbers of robots and obstacles, and compared with the PPO-AA model from Zhang et al. (2025). 53

Contents

1 Introduction **1**

2 Undirected Graphical Models **3**
- 2.1 Gibbs Distributions . . . 3
- 2.2 Markov Random Fields . . . 6
- 2.3 The Hammersley-Clifford Theorem . . . 7
- 2.4 Summary . . . 9

3 Reinforcement Learning **10**
- 3.1 Markov Decision Processes . . . 10
- 3.2 Solving MDPs . . . 17
- 3.3 Summary . . . 27

4 Generalisations of the Markov Decision Process **29**
- 4.1 Partially Observable Markov Decision Processes . . . 29
- 4.2 Decentralised Partially Observable Markov Decision Processes . . . 31
- 4.3 Summary . . . 36

5 Robot Flocking Control **37**
- 5.1 Simple flocking . . . 37
- 5.2 Gibbs distribution based flocking . . . 40
- 5.3 Summary . . . 49

6 Simulation and results **50**
- 6.1 Simulation setup . . . 50
- 6.2 Evaluation . . . 51

7 Summary and Conclusion **54**

Chapter 1

Introduction

Graphical structures can be found in abundance throughout various academic disciplines, as well as throughout daily life. Examples include graphs of syntax in linguistics (Krivochen (2023)), networks of habitats created for conservation of wildlife (Bunn et al. (2000)), graphs on everyday images, where each node is a pixel, and graphs of particles used in chemistry and statistical physics to make inferences about whole systems. Out of this final application arose the Ising model, which models the energy of interacting atomic nuclei (Murphy (2012, Subsection 19.4.1)), where the relationships between each pair of nuclei are represented by undirected edges. This model was the first Markov random field, although it is often formulated in terms of the closely related Gibbs distribution (also called a Boltzmann distribution (Koller and Friedman (2009, Subsection 4.4.1))). The first chapter of this report provides a detailed look at both of these structures.

One application of graphs is in modelling the behaviour of multi-robot systems (Zhang et al. (2025)). In Chapter 5, we discuss two methods for modelling and autonomously controlling a swarm of robot agents, in order to get them to move in a flock, similar to birds, or schools of fish. This behaviour allows the swarm to move quickly and cohesively from one place to another, and has applications in search and rescue and environmental surveillance (Zhang et al. (2025)).

We tackle this problem using multi-agent reinforcement learning techniques, which are developed in Chapter 3 and Chapter 4. Reinforcement learning (RL) is a branch of machine learning where an agent learns to interact with an environment in order to maximise some reward function (Sutton and Barto (2018)). RL is commonly applied as a solution method for Markov decision processes (MDPs), which are mathematical formalisations of sequential decision making processes. Early work on MDPs focused on situations where the number of possible states the environment could take on were finite. The earliest solution method, dynamic programming, introduced in Bellman (1957), iterated over each possible state and incrementally improved the agent's policy, which is a function that decides what action to take in a given situation. Reinforcement learning, on the other hand, aims to learn an optimal policy through interacting with the environment. Bolstered by the deep learning boom (LeCun et al. (2015)) in the 2010s, reinforcement learning has been applied to solve various complex tasks in recent years. The AlphaGo program (Silver et al. (2016)) developed by Google DeepMind, was trained using reinforcement learning techniques and defeated multiple top Go players, demonstrating the ability of reinforcement learning to contribute to the development of artificial intelligence agents with super-human abilities. More recently, reinforcement learning has been applied to robotic bimanual manipulation tasks (Xu et al. (2026)), where it helps to improve the speed and reliability of precise motions, like putting together a zip tie or aligning a screw.

We provide an in-depth discussion of MDPs, including how to solve them using gradient-based optimisation techniques in Chapter 3, and extend the MDP formulation to agents with limited information, and multiple agents, in Chapter 4. In Chapter 6, we implement simulated experiments with one of the models we discuss in Chapter 5, and compare it to the model in

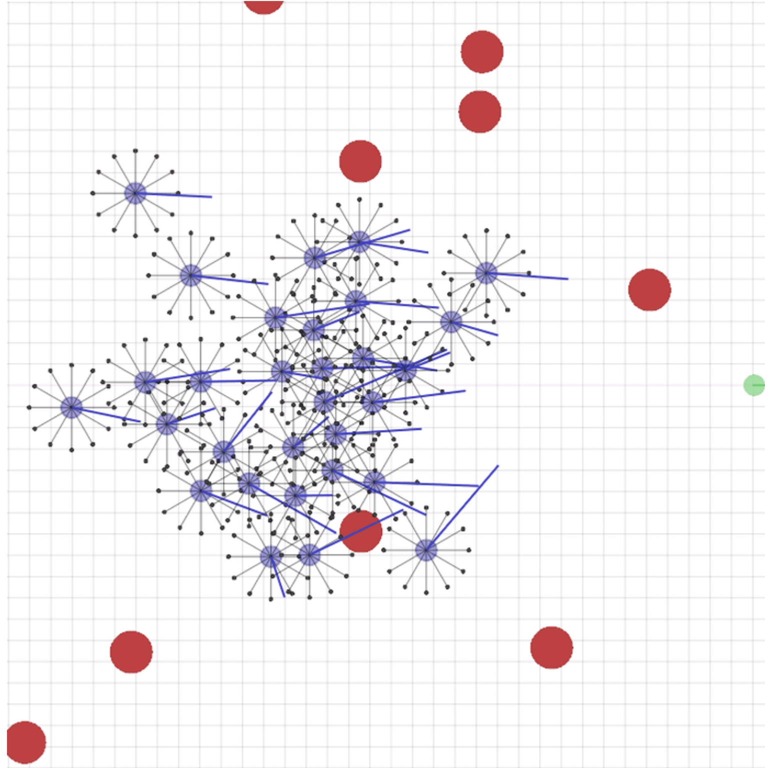

Figure 1.1: A rendering of 30 agents flocking amongst 50 obstacles (most of which are off-screen). The full video rendering can be found here.

Zhang et al. (2025), which models the robot swarm as a graph, and makes use of the properties of the Gibbs distribution that we develop in Chapter 2 to form a reward function that effectively captures our intentions for the swarm. A rendering of this simulation is provided in Figure 1.1.

Throughout this report we have tried to approach the topics from the perspective of building models about the world in order to achieve some goal. This practical mindset has not interfered, we hope, with the rigour with which we tackle the mathematical material, yet we feel that it does improve the pedagogical aspect of the presentation. By making clear the intuitions and intentions that underlie the introduction of the theory, we believe that a deeper understanding of the methods presented in this report can be acquired. With that, we begin our introduction to undirected graphical models.

Chapter 2

Undirected Graphical Models

Various structures of interest can be captured using with a graphical representation: from images, where each pixel is a vertex with edges connecting it to each neighbour, to social networks, where vertices are people and edges show relationships. In this chapter, we tackle the problem of constructing probabilistic models over graphs, which can be used to perform inference and learn about the behaviour or properties of vertices in these graphical representations. We introduce and discuss key attributes of Gibbs distributions and Markov Random Fields, tools which were originally developed for modelling phenomena in statistical physics, but have since seen extensive use in other fields, primarily computer vision, where they are used for tasks like image classification, semantic segmentation, stereo matching and more.

2.1 Gibbs Distributions

For many structures and groups of people, objects, etc., thinking about the relationships between each pair or other small subset of variables is often the intuitive way to understand the system. We can capture the influence a group of variables have on one another using fairly general functions called *factors*, also sometimes called *potential functions* (see Bishop (2006)). The only constraint on these functions is that they must be positive. In this section we showcase factors in a similar way to Koller and Friedman (2009).

Definition 2.1 (Factor)**.** *Let $\boldsymbol{X}$ be a random vector with support $\mathcal{X}$. A factor is a function $\phi : \mathcal{X} \to \mathbb{R}_{>0}$.*

The value of a factor can be thought of as quantifying how agreeable this pair of input values is to the modeller. For example, if we want to model two people having a debate, we could assign a high value to represent that they are agreeing with each other, and a low value to represent that they are not. We can also think of factors like unnormalised probabilities. If we imagine that some input values are going to be randomly generated, then we would want values that lead to a high factor output, that is, we would want the probability of such values occurring to be high. Similarly, we would want unfavourable values (ones which result in a small factor output) to have a low probability of occurring.

Example 2.1 (Mutually Assured Destruction)**.** *Each of A, B, and C are at war with one another, and tensions have reached the point where each country thinks nuclear war is likely. Each country can either fire their nukes at their two enemies, or, fearing mutually assured destruction, not fire them. If A fires their nukes, we say $A = 1$, if they do not, $A = 0$; this holds for B and C too.*

If any of the other countries fire their nukes, A and B are highly likely to fire back. C fears the complete destruction of humanity

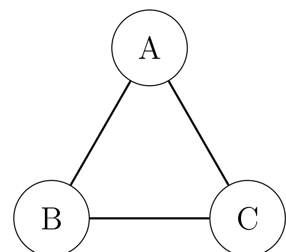

Figure 2.1: A graphical model for Example 2.1. Edges represent being at war.

A	B	$\phi(A,B)$
0	0	10
1	0	1
0	1	1
1	1	100

A	C	$\phi(A,C)$
0	0	10
1	0	1
0	1	1
1	1	200

B	C	$\phi(B,C)$
0	0	10
1	0	50
0	1	1
1	1	20

Figure 2.2: Pairwise factors for Example 2.1.

and is less likely to retaliate if B fires their nukes. C has a long and hateful history with A, however, so if A fires their nukes, C will almost certainly fire back.

The structure of this example is captured graphically in Figure 2.1, and the interactions are captured with the factors in Figure 2.2, which we present in tabular form, as in Koller and Friedman (2009).

We may be interested in performing inference on the global behaviour of this system, for example, calculating the probability of all three countries firing their nukes. To this end, we can combine the influence of the pairwise factors by multiplying them all together, then normalising the result to get a valid probability distribution

$$\mathbb{P}(A=a,B=b,C=c)=\frac{1}{Z}\phi(a,b)\phi(a,c)\phi(b,c)\ ,\quad \textit{where } Z=\sum_{a,b,c\in\{0,1\}}\phi(a,b)\phi(a,c)\phi(b,c).$$

The probability of all countries firing their nukes is

$$\mathbb{P}(A=1,B=1,C=1)=\frac{1}{406740}\times 100\times 200\times 20\approx 0.9834\ldots$$

which really puts the "assured" in mutually assured destruction!

The probability distribution we have demonstrated here is called a *Gibbs distribution*. Specifically, this is a Gibbs distribution over pairwise *cliques*.

Definition 2.2 (Clique)**.** *A clique is a set of vertices that are all neighbours of each other. We refer to a clique containing n vertices as an n-clique.*

For example, all pairs of vertices with an edge between them form a 2-clique, since both vertices in the set are neighbours of one another. We can have bigger cliques too though - in fact, A, B, and C in the nuclear warfare example are all in a 3-clique. We cannot, however, always combine 2-cliques into 3-cliques in general. We can define Gibbs distributions over cliques of any size, but often it is easier to conceptualise relationships between pairs of variables, even if there are bigger cliques available.

A	B	C	$\phi(A,B,C)$
0	0	0	1000
1	0	0	50
0	1	0	500
0	0	1	10
1	1	0	5000
1	0	1	20
0	1	1	200
1	1	1	400000

Figure 2.3: Factors over the 3-clique in Example 2.1.

Example 2.2 (Mutually Assured Destruction with a 3-clique)**.** *In this situation, as we have seen, it is intuitive to specify pairwise relationships, and then calculate the joint distribution*

by multiplying the factors together. We could have specified dynamics over the 3-clique directly, however, as in Figure 2.3.

These entries are $\phi(A,B,C) = \phi(A,B)\phi(A,C)\phi(B,C)$, as in the first part of this example. If the values were changed, this would be a different distribution, and hence model a slightly different situation.

It is important to emphasise that defining a 3-clique like this is quite unnatural from a modelling perspective, as considering three-way interactions is more difficult than reasoning about pairwise ones. It is also easier to discern how the different countries act with one another when using pairwise terms. For that reason, pairwise and unary terms (e.g., a factor defined over only one variable, say, $\phi(a)$) are frequently all that is used.

We are now ready to formally define a Gibbs distribution.

Definition 2.3 (Gibbs Distribution)**.** *Given a set of random variables $X = \{X_i : i \in \mathcal{V}\}$, on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and factors, $\phi_c(\boldsymbol{x}_c)$, over cliques $c \in \mathcal{C} \subseteq \mathcal{V}$, we say X has a Gibbs distribution, with probability mass/density function*

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \phi_c(\boldsymbol{x}_c)$$

where Z is called the partition function, and normalises the distribution so it integrates to 1:

$$Z = \sum_{\boldsymbol{x} \in \mathcal{X}} \prod_{c \in C} \phi_c(\boldsymbol{x}_c) \mathrm{d}\boldsymbol{x}$$

(for continuous X, we replace the sum with an integral).

We often formulate Gibbs distributions in terms of *energy functions*. These are just transformed factors, ψ, such that $\phi(\boldsymbol{x}) = \exp(-\psi(\boldsymbol{x}))$. This gives us

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{c \in \mathcal{C}} \phi_c(\boldsymbol{x}_c) = \frac{1}{Z} \exp\left(-\sum_{c \in \mathcal{C}} \psi_c(\boldsymbol{x}_c)\right) = \frac{1}{Z} \exp(-E(\boldsymbol{x}))$$

where $E(\boldsymbol{x})$ is called the *total energy* of the distribution.

Capturing interactions is often conceptually simpler when using energy functions rather than factors, a big part of this being because energy functions can be defined over all of $\mathbb{R}$ rather than just the positive reals.

Another benefit is that for continuous distributions, factors need to be integrable so we can normalise the distribution. This means the functions need to decay in some sense over the real line (so the integral doesn't go off to infinity). By specifying interactions as $\exp(-\psi(\boldsymbol{x}))$, we get this decay for free, and can then specify $\psi(\boldsymbol{x})$ using familiar functions like polynomials. If we specified factors instead, we couldn't set this to a polynomial, as then it wouldn't be integrable. In this sense, thinking of valid factors is harder than thinking of valid energy functions.

The downside of the generality of factors and energy functions is that the partition function is often intractable. Many methods have been devised to circumvent this issue and perform Bayesian inference using Gibbs distributions. Methods include Markov Chain Monte Carlo (Murphy (2012, chapter 24), Gelman (2013, chapter 11)), Variational Inference, including the Mean Field Method (Murphy (2012, chapter 21), Koller and Friedman (2009, chapter 11)), message-passing (Bishop (2006, chapter 8)), and more, but we do not cover these topics here.

2.2 Markov Random Fields

In the previous section, we covered the case where the variables of interest in our model can be represented on an undirected graph, and it is intuitive to specify pairwise interactions between them. We now turn our attention to a similarly common case where the behaviour of our variables in the graph depend only on the variables corresponding to neighbouring vertices.

This occurs, for example, in images, where we can model pixels as vertices with edges connecting them to the 4 adjacent pixels.

A common assumption is that neighbouring pixels are likely to have the same colours, or both be in the back/foreground of the image, since neighbouring pixels are likely to belong to the same object in the image. Pixels beyond this neighbourhood can thus be thought of as having their influence blocked by the neighbours.

Figure 2.4: Pixels in an image can be modelled as a graph. We infer that X is likely to be red, like its neighbours.

To illustrate this, consider Figure 2.4. Our pixel of interest is X. Given we know the neighbouring pixels are red, whether we know about the green pixel two edges away or not, our probability of X being red remains the same. [1]

With this motivation in mind, we provide the following definitions from Pollard (2004):

Definition 2.4 (Neighbourhood)**.** *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the neighbourhood of a vertex is the set of all vertices that have an edge connected to the vertex, i.e.*

$$\mathcal{N}_i = \{j \in \mathcal{V} : (i, j) \in \mathcal{E}\}.$$

Definition 2.5 (Markov Random Field)**.** *Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a finite collection of random variables $X = \{X_i : i \in \mathcal{V}\}$ with support $\mathcal{X}$, the probability mass/density function P on X is a **Markov Random Field** if for all $i \in \mathcal{V}$ and $\boldsymbol{x} \in \mathcal{X}$*

$$P(x_i|\{x_j : j \neq i\}) = P(x_i|\{x_j : j \in \mathcal{N}_i\}).$$

The equality above is called the *local Markov property*, as it is similar to the Markov property for Markov chains. MRFs also exhibit two more independence properties, which are helpful for developing algorithms on MRFs. We follow Koller and Friedman (2009), albeit with some changes in notation.

Definition 2.6 (Pairwise Markov Property)**.** *Consider a set of random variables $X = \{X_i : i \in \mathcal{V}\}$ on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The distribution on X has the Pairwise Markov Property if for all $i, j \in \mathcal{V}$ such that $(i, j) \notin \mathcal{E}$ we have*

$$X_i \perp X_j | X \backslash \{X_i, X_j\}.$$

Intuitively, random variables on any two unconnected vertices are independent if we know the values of all other random variables on the graph.

The next property requires us to introduce the concept of *separation* on a graph. First, recall that a *path* on a graph is a sequence of vertices that are connected by edges.

[1] Of course, this all changes if we are building a model for checkerboard patterns. It is important to remember that our models need only be useful, not generalise to every possible situation.

Definition 2.7 (Active path)**.** *Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. Take vertices $v_1, \ldots, v_k \in \mathcal{V}$ such that $v_1 - \ldots - v_k$ is a path in $\mathcal{G}$, and take $I \subseteq \mathcal{V}$ to be a subset of vertices. We say the path is active given I if none of the vertices in the path are in I.*

Definition 2.8 (Separation)**.** *For subsets of vertices $I, J, K \subseteq \mathcal{V}$, we say K separates I from J on $\mathcal{G}$, or $sep_{\mathcal{G}}(I; J|K)$ if there is no active path between any $i \in I$ and any $j \in J$ given K.*

Intuitively, two sets of vertices are separated by another set of vertices if there is no way to move between the first set and the second without crossing the third. This idea is the basis of our final independence property.

Definition 2.9 (Global Markov Property)**.** *Consider a set of random variables $X = \{X_i : i \in \mathcal{V}\}$ on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The distribution on X has the Global Markov Property if for all $A, B, C \subseteq X$ with corresponding sets of vertices $I, J, K \subseteq \mathcal{V}$ such that $sep_{\mathcal{G}}(I; J|K)$, we have $A \perp B|C$.*

We find this definition to be, unfortunately, notation-heavy; the key idea is that separation in the graph implies conditional independence between the corresponding sets of random variables.

It can be shown that the global Markov property implies the local, and the local implies the pairwise. Furthermore, the pairwise property implies the global property under the assumption that the probability distribution on $\boldsymbol{X}$ is *positive*.

Definition 2.10 (Positive Distributions)**.** *For a set of random variables X with support $\mathcal{X}$, a probability distribution P on X is positive if $P(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \mathcal{X}$.*

The proofs of equivalence between these properties can be found in Koller and Friedman (2009).

2.3 The Hammersley-Clifford Theorem

In many situations, we seek a model which can be specified using pairwise energy functions, as in the case of a Gibbs distribution, and also has the independence properties of an MRF. Fortunately, we have the following two theorems:

Theorem 2.1. *All Gibbs distributions are Markov Random Fields.*

Theorem 2.2 (Hammersley-Clifford)**.** *For a positive distribution P, if P is a Markov random field, then P is a Gibbs distribution.*

The proof that all Gibbs distributions are Markov random fields essentially follows from the fact that we define a Gibbs distribution by specifying local interactions, i.e., those that only depend on other vertices of a clique, which is a subset of a vertex's neighbourhood. To prove the theorem, we need a couple of lemmas, the first of which we take from Koller and Friedman (2009), and the second of which is a completed exercise from the same book.

Lemma 2.1 (Decomposition property)**.** *For random variables X, Y, U, Z,*

$$X \perp Y, U|Z \implies X \perp Y|Z.$$

Proof. Assume $X \perp Y, U|Z$, then by the definition of conditional independence we have $P(x, y, u|z) = P(x|z)P(y, u|z)$. Now, using the Law of Total Probability, we see that

$$\begin{aligned} P(x, y|z) &= \sum_u P(x, y, u|z) \\ &= \sum_u P(x|z)P(y, u|z) \\ &= P(x|z) \sum_u P(y, u|z) \\ &= P(x|z)P(y|z). \end{aligned}$$

□

Lemma 2.2. *For subsets of random variables $A, B, C \subseteq X$ such that $A \cup B \cup C = X$, if the distribution of X, $P(\boldsymbol{x}) = f(\boldsymbol{a}, \boldsymbol{c})g(\boldsymbol{b}, \boldsymbol{c})$, for some functions f and g, then $A \perp B|C$.*

Proof. First, observe that

$$\begin{aligned} P(\boldsymbol{a}, \boldsymbol{b}|\boldsymbol{c}) &= \frac{P(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})}{P(\boldsymbol{c})} \\ &= \frac{P(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})}{\sum_{\boldsymbol{a},\boldsymbol{b}} P(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})} \\ &= \frac{f(\boldsymbol{a}, \boldsymbol{c})g(\boldsymbol{b}, \boldsymbol{c})}{\left[\sum_{\boldsymbol{a}} f(\boldsymbol{a}, \boldsymbol{c})\right]\left[\sum_{\boldsymbol{b}} g(\boldsymbol{b}, \boldsymbol{c})\right]}. \end{aligned}$$

We follow a similar procedure for $P(\boldsymbol{a}|\boldsymbol{c})$

$$\begin{aligned} P(\boldsymbol{a}|\boldsymbol{c}) &= \frac{P(\boldsymbol{a}, \boldsymbol{c})}{P(\boldsymbol{c})} \\ &= \frac{\sum_{\boldsymbol{b}} P(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})}{\sum_{\boldsymbol{a},\boldsymbol{b}} P(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c})} \\ &= \frac{\sum_{\boldsymbol{b}} f(\boldsymbol{a}, \boldsymbol{c})g(\boldsymbol{b}, \boldsymbol{c})}{\sum_{\boldsymbol{a},\boldsymbol{b}} f(\boldsymbol{a}, \boldsymbol{c})g(\boldsymbol{b}, \boldsymbol{c})} \\ &= \frac{f(\boldsymbol{a}, \boldsymbol{c}) \sum_{\boldsymbol{b}} g(\boldsymbol{b}, \boldsymbol{c})}{\left[\sum_{\boldsymbol{a}} f(\boldsymbol{a}, \boldsymbol{c})\right]\left[\sum_{\boldsymbol{b}} g(\boldsymbol{b}, \boldsymbol{c})\right]} \\ &= \frac{f(\boldsymbol{a}, \boldsymbol{c})}{\sum_{\boldsymbol{a}} f(\boldsymbol{a}, \boldsymbol{c})}. \end{aligned}$$

By following the same procedure for $P(\boldsymbol{b}|\boldsymbol{c})$, we see that

$$\begin{aligned} P(\boldsymbol{a}, \boldsymbol{b}|\boldsymbol{c}) &= \frac{f(\boldsymbol{a}, \boldsymbol{c})}{\sum_{\boldsymbol{a}} f(\boldsymbol{a}, \boldsymbol{c})} \cdot \frac{g(\boldsymbol{b}, \boldsymbol{c})}{\sum_{\boldsymbol{b}} g(\boldsymbol{b}, \boldsymbol{c})} \\ &= P(\boldsymbol{a}|\boldsymbol{c})P(\boldsymbol{b}|\boldsymbol{c}). \end{aligned}$$

□

We now present the proof of theorem 2.1 from Koller and Friedman (2009).

Proof of theorem 2.1. Let P be a Gibbs distribution on a set of random variables $X = \{X_i : i \in \mathcal{V}\}$ on a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Consider disjoint subsets $A, B, C \subseteq X$ with corresponding sets of vertices $I, J, K \subseteq \mathcal{V}$, such that $\text{sep}_{\mathcal{G}}(I; J|K)$. We aim to show that P possesses the global Markov property, that is, that $A \perp B|C$. Since the global property implies the pairwise, this is sufficient for showing that P is an MRF.

First, consider the case where $A \cup B \cup C = X$. Separation implies there are no direct edges between A and B, so any clique in $\mathcal{G}$ must be contained in either $A \cup C$ or $B \cup C$. We denote the set of cliques in $A \cup C$ as $\mathcal{C}_A$, and the set of cliques in $B \cup C$ as $\mathcal{C}_B$. Since P is a Gibbs distribution, we have

$$P(\boldsymbol{x}) = \frac{1}{Z} \prod_{c \in \mathcal{C}_A} \phi_c(\boldsymbol{x}_c) \cdot \prod_{c \in \mathcal{C}_B} \phi_c(\boldsymbol{x}_c).$$

As mentioned, none of the factors in the first product involve variables in B, and none of the factors in the second product involve variables in A. Thus, we can write

$$P(\boldsymbol{x}) = f(A, C) g(B, C),$$

where we have absorbed the normalising constant into $f(A, C)$. By Lemma 2.2, $A \perp B|C$.

We now consider the case where $A \cup B \cup C \subseteq X$. Let $U = X \backslash (A \cup B \cup C)$. U can be partitioned into disjoint sets U_1 and U_2 such that C separates $A' = A \cup U_1$ from $B' = B \cup U_2$. We can now apply the preceding argument to A', B' and C, so $A, U_1 \perp B, U_2|C$. Finally, we apply the decomposition property (Lemma 2.1) to yield $A \perp B|C$. □

The converse argument is trickier, and we do not present it here. The original proof can be found in Clifford (1990), and a simpler proof can be found in Pollard (2004), with more details in Koller and Friedman (2009).

2.4 Summary

We have introduced a class of models for capturing dependency structures that underlie various phenomena, allowing for statistical inference in a variety of situations. While we do not go into detail about how to perform inference, which is often done from a Bayesian perspective, we list the following techniques and sources for learning more about them: Markov chain Monte Carlo (Murphy (2012, chapter 24), Gelman (2013, chapter 11)), variational inference, including the mean field method (Murphy (2012, chapter 21), Koller and Friedman (2009, chapter 11)), message-passing (Bishop (2006, chapter 8)).

We proved that Gibbs distributions are Markov random fields and stated the Hammersley-Clifford theorem, which tells us that the converse holds for positive distributions. The former statement is often more useful, since in practice we often specify models by constructing energy functions (frequently an intuitive task) and forming a Gibbs distribution. It is then beneficial to know the Gibbs distribution is a Markov random field, as the independence properties have applications in the methods for inference listed above.

Later in this report, we will construct a Gibbs distribution to model communication between robots that are trying to navigate a difficult environment as a group without colliding with one another or objects. In the next chapter, we introduce reinforcement learning, a suite of methods for training an agent to make intelligent, autonomous decisions given observations of its environment. These methods, along with the Gibbs distribution, provide an efficient solution to the control problem.

Chapter 3

Reinforcement Learning

Reinforcement learning is a type of machine learning where an agent learns how to solve a decision problem from interactions with its environment. In this chapter, we discuss a formal mathematical setting for decision problems, the *Markov decision process*, and discuss how we can train an agent to find an optimal solution to such problems.

3.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a mathematical abstraction of sequential decision problems, from controlling the temperature in a manufacturing plant, to winning a game of chess (Russell and Norvig (2010, Chapter 17)). When we seek to solve a problem using reinforcement learning, it is customary to formalise it as an MDP (Sutton and Barto (2018, Chapter 3)). Reinforcement learning can be applied without the MDP framework (Sutton and Barto (2018, Chapter 17)), but we do not consider this here.

We first provide an informal overview, loosely following Sutton and Barto (2018). In the MDP framework, we have an *agent* that interacts with the *environment* and then receives a *reward*. This repeats for each time step of the decision process. For certain tasks, there is a last time-step, after which the process terminates; we call these tasks *episodic*, and we call all the time-steps from $t = 0$ (the start) to $t = T$ (the end) an *episode*. Tasks that continue forever are called *continuing*.

At each step of the decision process, the agent is in a certain *state* $S_t \in \mathcal{S}$. This could be the state of the environment around it, e.g., whether or not it is raining, the agent's current position, or even a state of belief, e.g., whether the agent thinks it can successfully jump over a wall in front of it. In a given state, the agent chooses an *action* $A_t \in \mathcal{A}$ to take. These actions move the agent to a new state S_{t+1} (which can be the same as S_t), and produce a reward $R_{t+1} \in \mathfrak{R} \subseteq \mathbb{R}$[1].

[1]We follow Sutton and Barto (2018) in using R_{t+1} to denote the reward received due to the action A_t, as it emphasises that this reward is received at the same time as the agent's movement to S_{t+1}

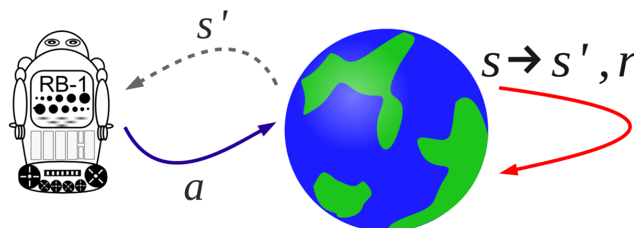

Figure 3.1: A Markov decision process. Our agent (the robot) interacts with its environment (the globe) by taking an action, a. This moves the robot from its current state s, to a new state, s', and the robot receives a reward, r. In an MDP, the agent always knows what state it is in. Image from Oliehoek and Amato (2016).

This is all illustrated in Figure 3.1. The state and action sets can be countable or uncountable. In the latter case, sums over these spaces should be replaced by integrals. Throughout the report, we simply write sums, but be aware that the theory applies to uncountable spaces too.

At $t = 0$, the agent is in an initial state S_0, which is sampled from the *initial state distribution* $\lambda(s)$. This can be a probability density function or a probability mass function depending on whether S_0 is a continuous or discrete random variable, i.e., whether $\mathcal{S}$ is uncountable or not.

Now we must discuss the dynamics of the MDP, that is, how the environment moves from one state to the next, and how the agent receives rewards. There are a few different yet common conventions in the literature, but we follow Albrecht et al. (2024, Section 3.3) in our definition. Once the agent has chosen an action $A_t = a$ to take in their current state $S_t = s$, they move to the next state $S_{t+1} = s'$ according to a *state transition probability function*, $\mathcal{T}(s'|s, a)$. This, like $\lambda(s)$, can be either a probability mass function or a probability density function. The reward received for making this transition, $R_{t+1} = r$, is given by the *reward function* $\mathcal{R}(s', a, s)$. We define this to be a deterministic function which outputs real numbers.

A common minor deviation from this definition is to define the reward function as $\mathcal{R}(a, s)$, that is, the next reward depends only on the current state and the current action. The three-argument reward function used in the definition above is slightly more general, but the two are actually equivalent, as shown in Albrecht et al. (2024, Section 2.8).

A more general way to define the dynamics of the MDP is to allow the reward received after a transition to be stochastic, as in Sutton and Barto (2018). In this case, it is customary to define the *dynamics function*, $P(s', r|s, a)$, in place of $\mathcal{T}$ and $\mathcal{R}$. $P(s', r|s, a)$ is the conditional probability mass/density over the next state and next reward, given the current state and current action. Note that since the reward R_{t+1} is stochastic even when we know s_t and s_{t+1} under this formulation, $P(s', r|s, a)$ is a probability mass function only if both $\mathcal{S}$ and $\mathfrak{R}$ are finite.

We do not use this formulation in this report because the case where $\mathcal{R}$ is deterministic is more common in the field of *multi-agent reinforcement learning*, which we discuss in Chapter 4. We will, however, briefly make use of this paradigm later during our proof of Theorem 3.1, therefore it is worth discussing the fact that deterministic rewards are a special case of stochastic rewards, where we have

$$P(s', r|s, a) = \begin{cases} \mathcal{T}(s'|s, a) & \text{if } r = \mathcal{R}(s', a, s) \\ 0 & \text{otherwise.} \end{cases}$$

Finally, we provide a brief insight into how MDPs acquired their name. The "Markov" in MDP refers to the fact that the state transition probability function exhibits the *Markov property*, that is, the next state depends only on the current state and action, not any previous states or actions. This means that for an MDP to accurately reflect the decision process it is modelling, the current state must contain all information about the past that is relevant for optimal decision-making. We assume that this is the case throughout the report.

We are now ready to formally define an MDP.

Definition 3.1 (Markov Decision Process)**.** *A Markov Decision Process (MDP) is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \lambda)$ *where* $\mathcal{S}$ *is the state space,* $\mathcal{A}$ *is the set of actions,* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ *is the reward function,* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *is the state transition probability function, and* $\lambda : \mathcal{S} \to [0, 1]$ *is the initial state distribution.*

Some definitions, such as the one used in Sutton and Barto (2018), allow the action space to

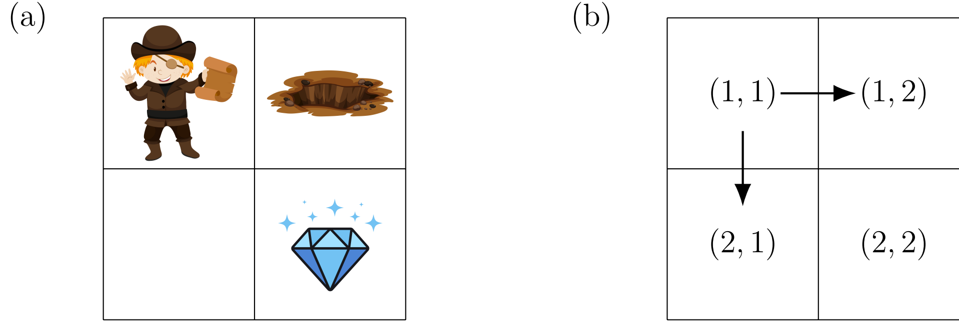

Figure 3.2: Treasure hunting. (a) The treasure hunter, pitfall, and diamond all live in a 2×2 grid world. The hunter seeks the diamond and wishes to avoid the pit. (b) The hunter can move to any of the two adjacent grid spaces, for example, when the hunter is at $(1, 1)$, he can move to $(2, 1)$ or $(1, 2)$[2].

vary based on the current state, that is, in place of $\mathcal{A}$ in the definition above $\{\mathcal{A}(s) : s \in \mathcal{S}\}$ is used. Practically, there is no difference between these two approaches, but the reward and state transition probability function have to be defined slightly differently for certain problems. This is illustrated in Example 3.1.

Example 3.1 (Treasure Hunting). *A treasure hunter lives in a 2 ×2 grid and is seeking a diamond while attempting to avoid a pitfall, as shown in Figure 3.2. The hunter begins at* $(1, 1)$*, the treasure is at* $(2, 2)$*, and the pit is at* $(1, 2)$*. This would typically be a simple task for the hunter, but unfortunately, he is drunk. The hunter can move into either of the two adjacent cells at each time step, but he has a* 0.2 *probability of accidentally stumbling into the wrong one, meaning the probability of him moving into the correct cell is only* 0.8*. The task ends when the hunter reaches the diamond, or after 10 time steps, at which point he gets dizzy from moving around so much and falls asleep.*

We can formulate this problem as an episodic MDP. Our state space is $\mathcal{S} = \{(i, j) : i, j \in \{1, 2\}\}$*, and our initial state distribution is*

$$\lambda(s) = \begin{cases} 1 & \text{if } s = (1, 1) \\ 0 & \text{otherwise.} \end{cases}$$

In this scenario, it is natural to define the action space as a function of the current state, that is, when the hunter is in state s*, the action space is* $\mathcal{A}(s)$*. When the hunter is in the top left of the grid, i.e.* $s = (1, 1)$*, he can move down or right, so* $\mathcal{A}((1, 1)) = \{down, right\}$*. Similarly, we have* $\mathcal{A}((1, 2)) = \{down, left\}$*,* $\mathcal{A}((2, 1)) = \{up, right\}$*, and* $\mathcal{A}((2, 2)) = \{up, left\}$*. The hunter's drunkenness is characterised by the transition probabilities.*

$$\mathcal{T}(s'|s, a) = \begin{cases} 0.8 & \text{if moving in direction } a \text{ would lead to state } s' \\ 0.2 & \text{if } s' \text{ is adjacent to } s \text{ but moving in direction } a \text{ would not lead to } s' \\ 0 & \text{otherwise.} \end{cases}$$

It is clear that $\mathcal{T}(s'|s, a)$ *provides a valid probability distribution, since* $\mathcal{T}(s'|s, a) \in (0, 1)$ *for all* s' *and* $\sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s, a) = 1$*.*

Alternatively, we can define a single action for all steps, but this requires us to redefine $\mathcal{T}$*. The action space is* $\mathcal{A} = \{up, down, right, right\}$*, and the transition probabilities are as before, except*

[2] Cartoon images from freepik.com.

now if the hunter takes an action that would move him off the grid, he instead remains where he is:

$$\mathcal{T}(s'|s,a) = \begin{cases} 1 & \textit{if } s' = s \textit{ and moving in direction } a \textit{ would take the hunter off the grid} \\ 0.8 & \textit{if moving in direction } a \textit{ would lead to state } s' \\ 0.2 & \textit{if } s' \textit{ is adjacent to } s \textit{ but moving in direction } a \textit{ would not lead to } s' \\ 0 & \textit{otherwise.} \end{cases}$$

Finally, our reward function is

$$\mathcal{R}(s', a, s) = \begin{cases} -5 & \textit{if } s' = (1,2) \\ ||s - (2,2)||_1 - ||s' - (2,2)||_1 & \textit{otherwise} \end{cases},$$

where $||\cdot||_1$ *denotes the* ℓ_1*-norm, also called the taxi-cab norm*[3]*. It can be thought of as measuring the distance between two points when you can only move up, down, left, or right, making it well suited to this example. This reward penalises the hunter for falling into the tar pit, or moving away from the diamond, and rewards him for moving towards it. Indeed, if the hunter moves from his starting position of* $(1,1)$ *to the empty square* $(2,1)$*, then we have*

$$\begin{aligned} \mathcal{R}((2,1), \textit{down}, (1,1)) &= ||(1,1) - (2,2)||_1 - ||(2,1) - (2,2)||_1 \\ &= ||(-1,-1)||_1 - ||(0,-1)||_1 \\ &= |-1| + |-1| - |0| + |-1| \\ &= 1 \end{aligned}$$

The reward does not depend on the current action, so $\mathcal{R}$ *is the same whether or not our action space depends on the current state.*

3.1.1 Policies and Total Reward

In this subsection we introduce tools for specifying which actions our agent should take, and how we can determine what optimal behaviour over an entire episode (or continuing task) looks like. We follow Sutton and Barto (2018, Chapter 3). We begin by defining a *policy*, which is a function that tells our agent how to act in all situations it may encounter.

Definition 3.2 (Policy)**.** *A policy* $\pi : \mathcal{A} \times \mathcal{S} \to [0,1]$ *tells us the probability of selecting action* $a \in \mathcal{A}$ *when in state* $s \in \mathcal{S}$*. We write* $\pi(a|s)$ *to indicate that this is a conditional probability distribution.*

A policy completely characterises the behaviour of our agent, since it provides an action for the agent to take in every state. The goal of RL is learning an *optimal policy*, but what does "optimal" mean here exactly?

Aiming to maximise the reward the agent receives at each time-step seems natural, but it can lead to missing the bigger picture. Consider a game of chess. Under normal circumstances, losing a queen is disastrous, and we may associate a large negative reward with such an action, but, if by sacrificing our queen we are able to checkmate our opponent, then we would certainly want to take this action. To get around this issue of short-sightedness, we consider the *return*, which is the sum of all rewards from the current time t until the end of the episode.

[3] For $\boldsymbol{x} \in \mathbb{R}^n$ we have $||\boldsymbol{x}||_1 = \sum_{i=1}^n |x_i|$.

Definition 3.3 (Return)**.** *In the episodic case, the return at time t is the sum of all future rewards*

$$G_t := R_{t+1} + \ldots + R_T.$$

By aiming to maximise the expected total return, our agent is capable of considering delayed rewards, rather than just immediate ones, we therefore define an optimal policy to be one which maximises $\mathbb{E}_\pi[G_0]$, where $\mathbb{E}_\pi[\cdot]$ denotes the expectation given we follow π over the entire trajectory. More formally, we take the expectation and assume that the initial state is sampled from the initial state distribution, i.e., $S_0 \sim \lambda$, actions are sampled from the agent's policy, i.e., $A_t \sim \pi(\cdot|s_t)$, and successor states are sampled from the state transition probability function, i.e., $S_{t+1} \sim \mathcal{T}(\cdot|a_t, s_t)$ (Albrecht et al. (2024)).

For continuous tasks, the return is an infinite sum, and for unconstrained rewards, this sum diverges. Two additions are required: firstly, we introduce the *discount rate*, $\gamma \in [0, 1]$, secondly, we insist that $\mathfrak{R}$ is bounded.

Definition 3.4 (Discounted Total Reward)**.** *In the continuing case with a discount rate $\gamma \in [0, 1]$, we define the discounted total return at time t,*

$$G_t := R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

Proposition 3.1. *The discounted total return, G_t, converges if $\gamma < 1$ and $\mathfrak{R}$ is bounded.*

Proof. Let $B = \max\{|\inf \mathfrak{R}|, |\sup \mathfrak{R}|\}$. Then by the geometric series,

$$|G_t| = \left|\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}\right| \leq B \sum_{k=0}^{\infty} \gamma^k = \frac{B}{1-\gamma} < \infty.$$

□

The discount rate determines how much the agent takes into account future rewards. For $\gamma = 0$, the agent considers only immediate rewards, and as γ approaches 1, future rewards are given more and more weight.

Discounting is a useful tool in both the episodic and continuing cases, so it makes sense to redefine the return in a generalised form which applies to both situations. To do so, we introduce the convention that for an episodic task that lasts T time steps, $R_{T+k} = 0$ for all $k \geq 0$.

Definition 3.5 (Generalised Return)**.** *For both the episodic and continuing cases we define*

$$G_t := \sum_{k=0}^{T} \gamma^k R_{t+k+1}$$

where we may have $T = \infty$ or $\gamma = 1$, but not both at the same time.

3.1.2 Value Functions

Our goal is now clear: we wish to find a policy, $\pi(a|s)$, that maximises the expected total return, $\mathbb{E}_\pi[G_0]$. The question of what such a policy looks like and how we would go about finding one remains unanswered. This is the topic of the next section, but before we discuss how to work towards an optimal policy, we first need to introduce two useful functions for evaluating the quality of our policy. We define them following Sutton and Barto (2018, Section 3.5). The first is called the *state-value function*, or just the *value function*.

Definition 3.6 (Value Function)**.** *The value function or state-value function for π is the expected total return given we are at a state $s \in \mathcal{S}$ and follow π. That is,*

$$v_\pi(s) := \mathbb{E}_\pi[G_t | S_t = s],$$

where t is any time step.

Note that for episodic tasks that have terminal states, i.e., if the agent reaches a certain state, the episode will end before the maximum number of time steps is reached, we have that $v_\pi(s) = 0$ if s is terminal. This makes intuitive sense, because the expected return after that point will always be 0, since agents will never leave that state.

As shown in Albrecht et al. (2024, Section 2.4), the value function for a policy can be found by solving the system of linear equations provided by *Bellman's equation*, named after Richard E. Bellman who used the equation in the development of *dynamic programming* (Bellman (1957)), a method for solving MDPs. We discuss this further in Section 3.2.

Proposition 3.2 (Bellman's equation)**.** *For a policy π, we have*

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) \left(\mathcal{R}(s',a,s) + \gamma v_\pi(s')\right). \tag{3.1}$$

Proof. We provide a more detailed version of the proof from Albrecht et al. (2024, Section 2.4). First, observe that we have the following recursive relationship for G_t

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \dots \\ &= R_{t+1} + \gamma(R_{(t+1)+1} + \gamma R_{(t+1)+2} + \dots) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{3.2}$$

Now we simply unwrap the expectation in the definition of v_π:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_a \pi(a|s) \sum_{s'} \mathcal{T}(s'|s,a) \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_{t+1} = s', S_t = s, A_t = a] \end{aligned}$$

Now we observe that G_{t+1} is independent of S_t and A_t, and that since s', a, and s are fixed, $\mathbb{E}_\pi[R_{t+1} | S_{t+1} = s', S_t = s, A_t = a] = \mathcal{R}(s', a, s)$.

$$\begin{aligned} &= \sum_a \pi(a|s) \sum_{s'} \mathcal{T}(s'|s,a) \left(\mathcal{R}(s',a,s) + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']\right) \\ &= \sum_a \pi(a|s) \sum_{s'} \mathcal{T}(s'|s,a) \left(\mathcal{R}(s',a,s) + \gamma v_\pi(s')\right). \end{aligned}$$

□

Bellman's equation does indeed provide a system of linear equations - observe that we have

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) \mathcal{R}(s',a,s) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) v_\pi(s'),$$

where $\pi(a|s), \mathcal{T}(s'|s,a)$, and $\mathcal{R}(s',a,s)$ are all constants, since their inputs are fixed. We can, therefore, find the value function via Gaussian elimination, or any other suitable method for solving systems of linear equations.

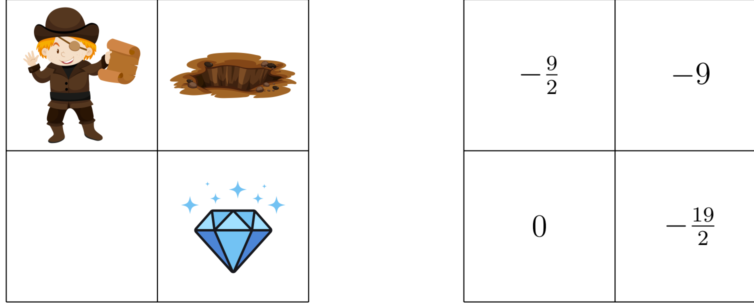

Figure 3.3: Treasure hunting continued. The values of each state under the policy described in Example 3.2 are shown in the grid on the right.

Example 3.2 (Treasure hunting continued)**.** *We return to the setting of Example 3.1, and investigate a possible policy and its values. In this example, we treat the case where* $\mathcal{A} = \{up, down, left, right\}$*, and* $\mathcal{T}$ *is defined with this in mind. We also set* $\gamma = 1$*, i.e., there is no discounting in the return, to make the computations easier.*

We consider the policy that selects all available actions with equal probability, that is $\pi(a|s) = \frac{1}{4}$ *for all* $s \in \mathcal{S}$ *and* $a \in \mathcal{A}$*. Clearly, this is not the optimal policy, so we expect the value of each state to be somewhat low, given our knowledge of* $\mathcal{R}$*.*

The Bellman equation (3.1) gives us

$$
\begin{aligned}
v_\pi((1,1)) = \frac{1}{4}\Big(&\mathcal{T}((1,1)|(1,1), up)[\mathcal{R}((1,1), up, (1,1)) + v_\pi((1,1))] + \\
&\mathcal{T}((1,2)|(1,1), up)[\mathcal{R}((1,2), up, (1,1)) + v_\pi((1,2))] + \\
&\ldots + \\
&\mathcal{T}((2,1)|(1,1), right)[\mathcal{R}((2,1), right, (1,1)) + v_\pi((2,1))] + \\
&\mathcal{T}((2,2)|(1,1), right)[\mathcal{R}((2,2), right, (1,1)) + v_\pi((2,2))]\Big).
\end{aligned}
$$

By following the formulae specified in Example 3.1, we get a lot of cancellations, and arrive at

$$
v_\pi((1,1)) = -1 + \frac{1}{2}v_\pi((1,1)) + \frac{1}{4}v_\pi((2,1)) + \frac{1}{4}v_\pi((1,2)).
$$

Denoting $v_\pi((x,y)) = v_{xy}$ *and repeating the above analysis for the other two non-terminal states, we get*

$$
\begin{aligned}
v_{11} &= -1 + \frac{1}{2}v_{11} + \frac{1}{4}v_{21} + \frac{1}{4}v_{12} \\
v_{12} &= -\frac{5}{2} + \frac{1}{2}v_{12} + \frac{1}{4}v_{11} + \frac{1}{4}v_{22} \\
v_{21} &= \frac{1}{2}v_{21} + \frac{1}{4}v_{11} + \frac{1}{4}v_{22}.
\end{aligned}
$$

Since $(2,2)$ *is the terminal state for this MDP, we must have* $v_\pi((2,2)) = v_{22} = 0$*. Substituting this into the above equations then performing some more substitutions leads to the following values, as shown in Figure 3.3.*

$$
v_\pi((1,1)) = -9,\ v_\pi((1,2)) = -\frac{19}{2},\ v_\pi((2,1)) = -\frac{9}{2},\ v_\pi((2,2)) = 0.
$$

As we would expect, the state with the pitfall has the lowest total value. This is because if we fall in the pit, then continue to choose actions with equal probability, we have a $\frac{1}{2}$ *chance of remaining in the pit, and losing another* 5 *reward.*

Another useful notion is the *action-value function* where we also condition on the action chosen at time t.

Definition 3.7 (Action-Value Function)**.** *The action-value function for* π *is the expected total return given we are at a state* $s \in \mathcal{S}$ *and take action* $a \in \mathcal{A}$*, then follow* π *thereafter. That is,*

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

A version of the Bellman equation exists for action-value functions too and can be found in Albrecht et al. (2024, Section 2.4).

3.2 Solving MDPs

Numerous approaches exist for solving MDPs. We distinguish between two key cases, that of *tabular methods* and *function approximation* methods. The earliest methods for solving MDPs were tabular methods, named as such because they apply to problems where the state and action spaces are small enough to allow for the value functions $v_\pi(s)$ and $q_\pi(s, a)$ to be represented as tables of values.

Dynamic programming, originally developed in Bellman (1957), is an important, tabular, non-RL method for solving MDPs. The method does not attempt to learn an optimal policy from experiencing interactions with the environment. Instead, it relies on knowing $\mathcal{T}(s'|s, a)$ and involves using a method called *policy iteration* (Sutton and Barto (2018, Section 4.3)) where an approximate value function $\widehat{v}_\pi(s)$ is updated to more closely match the true value function for our policy, $v_\pi(s)$, and then this approximation is used to update the policy, π, to be more likely to select the optimal action a, that is, the value that maximises $\mathcal{T}(s'|s, a)$, where $s' = \arg\max_{s \in \mathcal{S}} \widehat{v}_\pi(s)$. This procedure is then repeated, and it can be shown that the policy will eventually converge to the optimal policy for the MDP. This cyclic update procedure, called *generalised policy iteration* by Sutton and Barto (2018), underlies many RL algorithms, including other tabular methods like Q-learning and Monte Carlo methods (Sutton and Barto (2018)).

While tabular solution methods offer strong convergence guarantees, they are restricted to cases with small state and action spaces, and thus are often inapplicable to real-world problems, where we typically have incredibly large state spaces, like the number of possible positions in a chess game, or even uncountable state spaces, like the position of a robot. The problem is that it is computationally infeasible or even impossible to iterate over each state to apply tabular methods. Instead, we seek to approximate an optimal policy using a *parametrised policy* (Sutton and Barto (2018, Chapter 13)), where the values of our approximation $\pi_{\boldsymbol{\theta}}(a|s) = \pi(a|s, \boldsymbol{\theta})$ are controlled by a policy parameter $\boldsymbol{\theta}$ and the number of parameters is significantly smaller than the number of states. By changing $\boldsymbol{\theta}$, we affect not only the value of $\pi_{\boldsymbol{\theta}}(a|s)$, but also the value of $\pi_{\boldsymbol{\theta}}(a|s')$ for other states $s' \in \mathcal{S}$.

The immediate benefit of using a parametrised policy is that we can learn policies quicker and with less computation. The added bonus is that our policies are able to *generalise*: to take good actions in states that were not seen during training (Sutton and Barto (2018, Section 13.1)).

The downside is, unlike with tabular methods, we have no guarantee of an optimal solution, although later we will see that we can still guarantee *improvement* to our policy during training.

A final benefit mentioned in Sutton and Barto (2018) is that we can often take advantage of prior knowledge about a problem when structuring our approximate policy. For example, in situations where an agent acts based on visual input from camera images, it may be beneficial to create a policy based on either convolutional neural networks or vision transformers (Zhang et al. (2023)), which are specifically constructed to handle image data effectively.

Throughout the rest of this section, we only work with parametrised policies. To keep the notation light, we will henceforth refer to $\pi_{\boldsymbol{\theta}}$ as π, except for in certain cases where we feel it is more clear to be explicit. Unfortunately, the material in this section, and in later chapters, requires a basic knowledge of mini-batch stochastic gradient descent, and multi-layer perceptron (MLP) neural networks, although we do not have space to cover these topics in this report, coverage of these topics can be found in Sutton and Barto (2018, Sections 9.3 and 9.7), Albrecht et al. (2024, Sections 7.3 and 7.4), and Zhang et al. (2023, Chapters 5 and 12).

3.2.1 Approximate Policies

When approximating $\pi_{\boldsymbol{\theta}}(a|s)$, we require the output to be a probability distribution over the action set. For finite action sets, we use the *softmax* function.

Definition 3.8 (Softmax)**.** *For a vector $\boldsymbol{x} \in \mathbb{R}^n$,*

$$\textit{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}.$$

The elements of $\text{softmax}(\boldsymbol{x})$ are contained in $(0, 1)$ and sum to 1. Essentially, it turns a vector $\boldsymbol{x}$ into a probability distribution. This allows us to set $\pi_{\boldsymbol{\theta}}(a|s) = \text{softmax}(h(a, s, \boldsymbol{\theta}))$ for some *preference function* h which can take values anywhere on the real line. We then learn h using function approximation. Commonly, h is an MLP.

We now turn our attention to continuous action spaces. This case is similar, in that we train some function approximator h and then use its outputs to form a probability distribution, $\pi_{\boldsymbol{\theta}}(a|s)$. Since this distribution has to be continuous, however, we insist that h output values which can be transformed into parameters of some continuous probability distribution. It is common for continuous actions to be real valued vectors, i.e., $\mathcal{A} \subseteq \mathbb{R}^d$ for some $d \in \mathbb{N}$. This is the case, for example, when actions correspond to moving to some position in 3D space, or changing acceleration or velocity. In such a case, it is common to sample each component of the action vector from a normal distribution (Sutton and Barto (2018, Section 13.7)), that is, $a_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$, for $i = 1, \dots, d$.

As above, h in this case could be formed of $2d$ linear models, d models for the means and d for the variances. A more common and more general case (recalling that a linear model is simply a perceptron without an activation function), is to make h a neural network with $2d$ outputs. In both of these cases, the outputs will be real numbers, but we require a positive value for the variance of the normal distribution. This can be solved by applying an exponential function (Sutton and Barto (2018, Sectino 13.7)), a softplus function, $f(x) = \log(1 + e^x)$ (Bou et al. (2023)), or any other suitable function that maps from $\mathbb{R}$ to $\mathbb{R}_{>0}$. We insist that the first $\frac{d}{2}$ components of the output vector $h(a, s, \boldsymbol{\theta})$ correspond to the mean, and the last $\frac{d}{2}$ are to be used for the variance. Denoting the probability density function of a normal distribution as

$\mathcal{N}(x; \mu, \sigma^2)$, we have

$$\pi_{\boldsymbol{\theta}}(a|s) = \mathcal{N}(a_i; h_i(a, s, \boldsymbol{\theta}), f(h_{\frac{d}{2}+i}(a, s, \boldsymbol{\theta}))^2)^d.$$

3.2.2 Policy Gradient Methods

In what follows, we present policy gradient methods for solving episodic MDPs. These methods use gradient ascent techniques in order to maximise a *performance metric*, $J(\boldsymbol{\theta})$, which measures how good our parametrised policy, $\pi_{\boldsymbol{\theta}}(a|s)$, is. Formally, we will improve our policy parameter, $\boldsymbol{\theta}$, using the following update rule:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t),$$

where α is a learning rate to be tuned. This is identical to using gradient *descent* to minimise a loss function, but in RL, where we already think about maximising rewards, it makes sense to think of maximising performance rather than minimising losses.

In the episodic case, our performance metric is simply the expected total return

$$J(\boldsymbol{\theta}) = \mathbb{E}_\pi[G_0] = \mathbb{E}_{S_0 \sim \lambda}[v_\pi(S_0)].$$

A slightly different metric is used in Sutton and Barto (2018, Chapter 13), where it is assumed that episodes start in a fixed non-random initial state s_0, and thus

$$J(\boldsymbol{\theta}) = v_\pi(s_0) = \mathbb{E}_\pi[G_0 | S_0 = s_0].$$

In our case, we assume that the initial state is chosen according to the initial state distribution, $\lambda(s)$, so we do not condition on a fixed initial state s_0.

To perform gradient ascent, we need to be able to compute or estimate $\nabla J(\boldsymbol{\theta})$. This appears difficult at first, since the performance of a policy depends both on the distribution of states visited and the actions chosen in those states. This can be seen via the Bellman equation (Proposition 3.2) for the value function:

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_\pi[v_{\pi_{\boldsymbol{\theta}}}(S_0)] \\ &= \mathbb{E}_\pi\left[\sum_a \pi_{\boldsymbol{\theta}}(a|S_0) \sum_{s'} \mathcal{T}(s'|S_0, a)\left(r(s', a, s) + \gamma v_{\pi_{\boldsymbol{\theta}}}(s')\right)\right]. \end{aligned}$$

The dependence on $\mathcal{T}$, which is unknown, means we cannot compute $\nabla J(\boldsymbol{\theta})$ directly. Thankfully, the *policy gradient theorem* gives us a way to compute $\nabla J(\boldsymbol{\theta})$ without involving $\mathcal{T}$. To state it, we first need to develop the *on-policy state distribution.* We first follow Weng (2018) for the next definition.

Definition 3.9 (Probability of transitioning in k steps)**.** *We define $P_\pi(s \to s', k)$ to be the probability of transitioning from state s to s' after k steps while following policy π. Then we have $P_\pi(s \to s, 0) = 1$, and*

$$P_\pi(s \to s', 1) = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{T}(s'|s, a)$$

that is the probability of transitioning from s to s' in 1 step is the probability of choosing an action $a \in \mathcal{A}$ then transitioning to s' given you take that action, where a can be any action.

We can then define the general case recursively,

$$P_\pi(s \to x, k+1) = \sum_{s'} P_\pi(s \to s', k) P_\pi(s' \to x, 1)$$

that is, the probability of transitioning from s to x in $k+1$ steps is the probability of transitioning to some intermediary state s' in k steps, then transitioning to x in 1 step, where s' can be any state.

Next we have to define the expected time spent in state s. It is important to again distinguish between two cases. Recall that we defined our MDP to have an initial state distribution $\lambda(s)$, but, as discussed in Section 3.1, some sources (Sutton and Barto (2018), Sutton et al. (1999)) regard the initial state s_0 as fixed and deterministic. In the following definition we follow Silver et al. (2014) in taking an expectation with respect to the initial state S_0; in the fixed and deterministic case, there is no such expectation.

Definition 3.10 (Expected time spent in state s)**.** *For an episodic task with state space $\mathcal{S}$, we define $\eta(s)$ to be the average number of time steps spent at s in a single episode, that is,*

$$\eta(s) = \mathbb{E}_{S_0 \sim \lambda}\left[\sum_{t=0}^{\infty} \gamma^t P_\pi(S_0 \to s, t)\right]. \tag{3.3}$$

Here, we are using the interpretation that $(1-\gamma)$ is the probability that an episode ends on any given time step (Albrecht et al. (2024, Section 2.3)), so the summands are the probability that the episode does not end at time t and we are at state s at time t (equivalently, we have moved from the initial state, s_0, to s in t time steps).

Definition 3.11 (On-policy state distribution)**.** *For each state $s \in \mathcal{S}$, we define the on-policy state distribution to be the normalised expected time spent in state s, that is*

$$\mu(s) = \frac{\eta(s)}{\sum_{s' \in \mathcal{S}} \eta(s')}.$$

We can think of $\mu(s)$ as a scalar which tells us how frequently we are at a state s on average. We will shortly see how this allows for an intuitive understanding of the formula for $\nabla J(\boldsymbol{\theta})$ given by the policy gradient theorem.

Theorem 3.1 (Policy Gradient Theorem)**.** *For $J(\boldsymbol{\theta}) = \mathbb{E}_{S_0 \sim \lambda}[v_{\pi_{\boldsymbol{\theta}}}(s_0)]$,*

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_{\pi_{\boldsymbol{\theta}}}(s, a) \nabla \pi_{\boldsymbol{\theta}}(a|s)$$

where the gradients are with respect to $\boldsymbol{\theta}$.

The proof requires a couple of lemmas: both are from Sutton and Barto (2018); the second is a solved exercise.

Lemma 3.1.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$$

Proof. From the properties of conditional expectation,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)\mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(s,a) \end{aligned}$$

□

Lemma 3.2.

$$q_\pi(s,a) = \sum_{s'\in\mathcal{S}} \mathcal{T}(s'|s,a)(\mathcal{R}(s',a,s) + \gamma v_\pi(s'))$$

Proof. This proof is somewhat involved, but simply boils down to using conditional expectation properties and marginalisation.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1}|S_t = s, A_t = a] + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s, A_t = a]. \quad \text{(Equation 3.2)} \end{aligned}$$

We now consider each component of the sum in turn,

$$\mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{s'\in\mathcal{S}} \mathcal{R}(s',a,s)\mathcal{T}(s'|s,a).$$

$$\begin{aligned} \mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] &= \sum_{s'} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_t = a, S_t = s]\mathcal{T}(s'|s,a) \\ &= \sum_{s'} \Bigg[\sum_{a'} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a', S_t = s, A_t = a] \\ &\quad \times \pi(a'|s')\Bigg]\mathcal{T}(s'|s,a). \end{aligned}$$

Where we use the fact that the policy depends only on the most recent state, since by the Markov property, it contains all the information we need to make an optimal decision. Below, we similarly observe that G_{t+1} is independent of A_t and S_t.

$$\begin{aligned} \mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] &= \sum_{s'}\sum_{a'} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a']\pi(a'|s')\mathcal{T}(s'|s,a) \\ &= \sum_{s'}\sum_{a'} q_\pi(s',a')\pi(a'|s')\mathcal{T}(s'|s,a). \end{aligned}$$

Bringing the equations back together, we get

$$\begin{aligned} q_\pi(s,a) &= \sum_{s'} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s',a,s) + \gamma\sum_{a'} \pi(a'|s')q_\pi(s',a')\right] \\ &= \sum_{s'} \mathcal{T}(s'|s,a)\left[\mathcal{R}(s',a,s) + \gamma v_\pi(s')\right], \end{aligned}$$

where we have used Lemma 3.1 on the last line. □

Now we can prove the Policy Gradient Theorem.

Proof. We extend the proof in Sutton and Barto (2018) by incorporating discounting, i.e., $\gamma \in [0, 1]$, considering the initial state distribution rather than assuming a fixed initial state, and including clarifications from Weng (2018).

For all $s \in \mathcal{S}$:

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a), \quad \text{(Lemma 3.1)} \\
&= \sum_a \left[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla q_\pi(s,a)\right] \\
&= \sum_a \left[\nabla \pi(a|s) q_\pi(s,a) + \pi(a|s) \nabla \sum_{s'} \mathcal{T}(s'|s,a)(\mathcal{R}(s',a,s) + \gamma v_\pi(s'))\right] \quad \text{(Lemma 3.2)} \\
&= \sum_a \left[\nabla \pi(a|s) q_\pi(s,a) + \gamma \pi(a|s) \sum_{s'} \mathcal{T}(s'|s,a) \nabla v_\pi(s')\right].
\end{aligned}$$

We denote $\varphi(s) = \sum_{a \in \mathcal{A}} \nabla \pi(a|s) q_\pi(s,a)$, to simplify the following manipulations. We can now begin to "unroll" $\nabla v_\pi(s)$.

$$\begin{aligned}
\nabla v_\pi(s) &= \varphi(s) + \gamma \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \mathcal{T}(s'|s,a) \nabla v_\pi(s') \\
&= \varphi(s) + \gamma \sum_{s'} \left(\sum_a \pi(a|s) \mathcal{T}(s'|s,a)\right) \nabla v_\pi(s') \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \nabla v_\pi(s') \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \left(\varphi(s') + \gamma \sum_a \pi(a|s') \sum_{s''} \mathcal{T}(s''|s',a) \nabla v_\pi(s'')\right) \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \left(\varphi(s') + \gamma \sum_{s''} P_\pi(s' \to s'', 1) \nabla v_\pi(s'')\right) \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \varphi(s') + \gamma^2 \sum_{s''} \sum_{s'} P_\pi(s \to s', 1) P_\pi(s' \to s'', 1) \nabla v_\pi(s'') \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \varphi(s') + \gamma^2 \sum_{s''} P_\pi(s \to s'', 2) \nabla v_\pi(s'') \\
&= \varphi(s) + \gamma \sum_{s'} P_\pi(s \to s', 1) \varphi(s') + \gamma^2 \sum_{s''} P_\pi(s \to s'', 2) \varphi(s'') \\
&\quad + \gamma^3 \sum_{s'''} P_\pi(s \to s''', 3) \nabla v_\pi(s''')
\end{aligned}$$

This recursive unrolling proceeds infinitely, leading to

$$\nabla v_\pi(s) = \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \gamma^k P_\pi(s \to x, k) \varphi(x).$$

Finally, we have

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &= \mathbb{E}_{S_0\sim\lambda}[\nabla v_\pi(S_0)] \\
&= \sum_{s_0\in\mathcal{S}} \lambda(s_0) \sum_{s\in\mathcal{S}} \sum_{k=0}^{\infty} \gamma^k P_\pi(s_0 \to s, k)\varphi(s) \\
&= \sum_{s} \varphi(s) \sum_{s_0} \lambda(s_0) \sum_{k=0}^{\infty} \gamma^k P_\pi(s_0 \to s, k) \\
&= \sum_{s} \varphi(s)\eta(s) \\
&= \sum_{s'} \eta(s') \sum_{s} \frac{\eta(s)}{\sum_{s'} \eta(s')} \varphi(s) \\
&\propto \sum_{s} \mu(s) \sum_{a} \nabla\pi(a|s) q_\pi(s,a).
\end{aligned}$$

□

We now have a form of $\nabla J(\boldsymbol{\theta})$ that does not depend on the transition probability, $\mathcal{T}(s'|s,a)$, nor the gradient of the unkown state-action value function, $q_\pi(s,a)$. Computing $\nabla J(\boldsymbol{\theta})$ now only requires us to know $\mu(s)$, which can be approximated from experience, and $\nabla\pi(a|s)$, where $\pi(a|s)$ is often a neural network, meaning we can calculate this derivative via backpropagation.

An alternative form of $\nabla J(\boldsymbol{\theta})$ is as an expectation with respect to both S_t and A_t. This form will be vital to the development of the Monte Carlo policy gradient method, REINFORCE, in the next subsection, as we can then approximate the expectation using samples collected from interacting with the environment.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_{s} \mu(s) \sum_{a} q_\pi(s,a)\nabla\pi(a|s) \\
&= \mathbb{E}_{s\sim\mu(s)}\left[\sum_{a} q_\pi(s,a)\nabla\pi(a|s)\right] \\
&= \mathbb{E}_{s\sim\mu(s)}\left[\sum_{a} \pi(a|s) q_\pi(s,a) \frac{\nabla\pi(a|s)}{\pi(a|s)}\right] \\
&= \mathbb{E}_{s\sim\mu(s),a\sim\pi(a|s)}\left[q_\pi(s,a)\nabla\log\pi(a|s)\right]
\end{aligned}$$

where in the final line we use the identity $\nabla \log x = \frac{\nabla x}{x}$.

As discussed in both Sutton and Barto (2018) and Albrecht et al. (2024), the final line of the above derivation provides an intuitive justification for why performing gradient ascent with $\nabla J(\boldsymbol{\theta})$ leads to policy improvement. $\nabla\pi(a|s)$ points towards the direction in parameter space that maximises $\pi(a|s)$, the probability of choosing action a in state s (or similar states: recall the use of a parametrised policy allows for generalisation). This term is weighted by $q_\pi(s,a)$, which measures the quality of the action. By adding $\nabla J(\boldsymbol{\theta})$ to $\boldsymbol{\theta}$, we are moving closer to the maximum for $\pi_{\boldsymbol{\theta}}(a|s)$ for high value actions, a, i.e., we increase the probability of selecting a when in state s in the future, if doing so leads to a high expected return.

3.2.3 REINFORCE

We now have all the tools we need to introduce the earliest policy gradient method, REINFORCE (Williams (1992)). The REINFORCE algorithm is a direct application of the expectation form

of Theorem 3.1 derived above. The algorithm runs the approximate policy over an entire episode and uses the collected data to form samples of $\nabla J(\boldsymbol{\theta})$, which are used to update $\boldsymbol{\theta}$ via stochastic gradient descent (SGD). By collecting many episodes of data, we can encounter more states and improve the policy's ability to perform well in all states, so we perform SGD for N iterations, where N is some large integer.

For an episode of data, $(s_0, a_0, s_1, r_1, a_1, \dots, s_{T-1}, a_{T-1}, r_T)$, and recalling that we must have $r_T = 0$, we use the following update rule

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha_{\boldsymbol{\theta}} \left(\sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \right) \nabla \log \pi(a_t | s_t, \boldsymbol{\theta}_t).$$

The proportionality constant of $\nabla J(\boldsymbol{\theta})$ is absorbed into the learning rate $\alpha_{\boldsymbol{\theta}}$. Using $g_t = \left(\sum_{k=t+1}^{T} \gamma^{k-t-1} r_k\right)$ as an estimate for $q_\pi(s, a)$ where r_k is collected from interactions with the environment and $a_k = a$ and $s_k = s$ is called Monte Carlo estimation in the RL literature Sutton and Barto (2018). It is shown in Singh and Sutton (1996) that such an estimate converges to $q_\pi(s, a)$ on average.

We give the full REINFORCE algorithm in a way similar to Sutton and Barto (2018) below.

Algorithm 1 REINFORCE

Require: Number of iterations: N, learning rate: α, differentiable policy $\pi(a|s, \boldsymbol{\theta})$
  Initialise parameter $\boldsymbol{\theta}$
  for N iterations **do**
    Collect an episode of data: $s_0, a_0, s_1, r_1, a_1, \dots, s_{T-1}, a_{T-1}, r_T$ by following π
    for $t = 0, \dots, T-1$ **do**
      $g \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha g \nabla \log \pi(a_t | s_t, \boldsymbol{\theta})$
    end for
  end for

A downside of the REINFORCE algorithm is that the Monte Carlo estimate of $q_\pi(s, a)$, g_t, has a high variance, due to being computed over an entire episode of data, where rewards can vary greatly. This leads to unstable policy training (Albrecht et al. (2024)). The first step to mitigating this issue is the introduction of a *baseline*, $b(s)$, to Theorem 3.1. We follow Sutton and Barto (2018) in showing that with the addition of a baseline, Theorem 3.1 still holds:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} (q_\pi(s, a) - b(s)) \nabla \pi(a|s) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \left(\sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla \pi(a|s) - b(s) \sum_{a \in \mathcal{A}} \nabla \pi(a|s) \right) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \left(\sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla \pi(a|s) - b(s) \nabla \sum_{a \in \mathcal{A}} \pi(a|s) \right) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \left(\sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla \pi(a|s) - b(s) \nabla 1 \right) \\
&= \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} q_\pi(s, a) \nabla \pi(a|s)
\end{aligned}$$

For the manipulation above to be valid, we require that the choice of baseline be any function that does not depend on a, the chosen action.

By subtracting a baseline from our estimate of $q_\pi(s,a)$, namely the sample return, g_t, we reduce its variance, thereby stabilising training (Albrecht et al. (2024)). In some states, all actions may have high values, making it difficult to determine the best one. This suggests that the baseline should vary with the current state. The standard choice is to use an approximate value function $V_\pi(s,\boldsymbol{w})$, where $\boldsymbol{w}$ are parameters. As with policies, V_π is often a neural network which we train to approximate the true value function $v_\pi(s)$.

With our goal of approximating $v_\pi(s)$ in mind, we construct the following loss function

$$\overline{\text{VE}}(\boldsymbol{w}) = \sum_{s\in\mathcal{S}} \mu(s)(v_\pi(s) - V_\pi(s,\boldsymbol{w})^2) = \mathbb{E}_{S\sim\mu}[(v_\pi(S) - V_\pi(S,\boldsymbol{w}))^2]$$

$\overline{\text{VE}}$ tells us the squared error between the true and approximate value functions averaged over all states and weighted by $\mu(s)$. When interacting with the environment, we assume that the states we encounter are distributed according to $\mu(s)$ (Sutton and Barto (2018, Section 9.3)). This idea motivates our method for minimising $\overline{\text{VE}}$: we minimise the unbiased estimator $\mathcal{L}(\boldsymbol{w}) = (v_\pi(s) - V_\pi(s,\boldsymbol{w}))^2$, where $s \sim \mu(s)$ are samples collected from interacting with the environment.

We used stochastic gradient ascent to optimise the policy, but since we seek to *minimise* $\mathcal{L}(\boldsymbol{w})$ (and thereby $\overline{\text{VE}}$), we use stochastic gradient *descent*. Our update rule is

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha_{\boldsymbol{w}} \nabla\mathcal{L}(\boldsymbol{w}).$$

Fortunately, computing $\nabla\mathcal{L}(\boldsymbol{w})$ is much simpler than computing $\nabla J(\boldsymbol{\theta})$.

$$\begin{aligned}\nabla\mathcal{L}(\boldsymbol{w}) &= \nabla(v_\pi(s) - V_\pi(s,\boldsymbol{w}))^2 \\ &= -2(v_\pi(s) - V_\pi(s,\boldsymbol{w}))\nabla V_\pi(s,\boldsymbol{w}).\end{aligned}$$

Similarly to $J(\boldsymbol{\theta})$, which depends on $q_\pi(s,a)$, our loss function depends on the true value function of our policy, $v_\pi(s)$, which is of course unknown. We use g_t, computed using data collected from interacting with the environment, as a Monte Carlo estimate of $v_\pi(s)$ (Sutton and Barto (2018, Section 5.1)), leading to the following update rule.

$$\boldsymbol{w}_{t+1} \leftarrow \boldsymbol{w}_t - \alpha_{\boldsymbol{w}} \left(\left(\sum_{k=t+1}^{T} \gamma^{k-t-1} r_k \right) - V_\pi(s_t,\boldsymbol{w}_t) \right) \nabla V_\pi(s_t,\boldsymbol{w}_t),$$

where as with $\boldsymbol{\theta}$, we absorb the constant of 2 into the learning rate $\alpha_{\boldsymbol{w}}$. We provide the REINFORCE with baseline algorithm from Sutton and Barto (2018) below. Note that we do not absorb the negative sign into $\alpha_{\boldsymbol{w}}$, since learning rates are strictly positive.

3.2.4 Synchronous data collection

When implementing policy gradient methods, it is common to make use of the parallel processing capabilities of modern GPUs by simulating multiple environments concurrently (Clemente et al. (2017)). We follow Albrecht et al. (2024, Subsection 8.2.8) in discussing one paradigm for making use of parallel environments, called *synchronous data collection*. When collecting data synchronously, the parallel environments run independently, and return a vector of rewards and states to the agent. The agent then decides on an action to take in each environment. This

Algorithm 2 REINFORCE with baseline

Require: Number of iterations: N, learning rates: $\alpha_{\boldsymbol{\theta}}, \alpha_{\boldsymbol{w}}$, differentiable policy $\pi(a|s, \boldsymbol{\theta})$, differentiable value function $V_\pi(s, \boldsymbol{w})$

Initialise parameters $\boldsymbol{\theta}, \boldsymbol{w}$

for N iterations **do**

  Collect an episode of data: $s_0, a_0, s_1, r_1, a_1, \ldots, s_{T-1}, a_{T-1}, r_T$ by following π

  for $t = 0, \ldots, T-1$ **do**

    $g \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$

    $A \leftarrow g - V_\pi(s_t, \boldsymbol{w})$

    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} A \nabla_{\boldsymbol{\theta}} \log \pi(a_t|s_t, \boldsymbol{\theta})$

    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} A \nabla_{\boldsymbol{w}} V_\pi(s_t, \boldsymbol{w}))$

  end for

end for

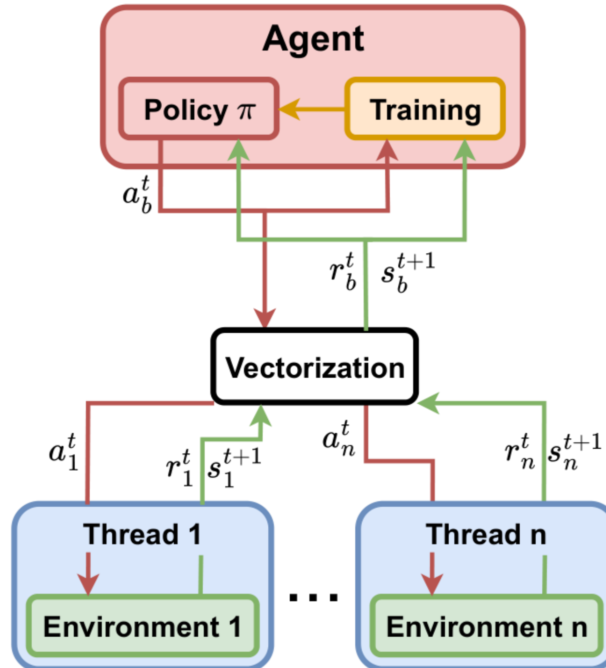

Figure 3.4: Synchronous data collection. n environments are run in parallel, and pass their rewards and next states to the agent. The agent then sequentially decides on rewards for each environment, and the environments receive these actions and compute the next set of rewards and states. Image is from Albrecht et al. (2024).

process then repeats until all episodes are finished. The process is called "synchronous" because the agent must wait for all environments to have returned a vector of rewards and states, and then chooses an action to take in each environment in order. The primary benefit of this is that it means we have more data during each update step of REINFORCE with baseline, meaning we can average the gradients used to update V_π and π_θ over the data from each environment. This makes the gradients more stable and makes training quicker. Synchronous data collection is illustrated in Figure 3.4.

An alternative to synchronous data collection is *asynchronous data collection*, where we parallelise not just the environments, but also the agent itself. This is discussed in Albrecht et al. (2024, Subsection 8.2.8), but we do not cover it here. Unlike synchronous data collection, asynchronous data collection is not as easy to implement. We provide the REINFORCE with baseline with synchronous data collection algorithm from Albrecht et al. (2024) below, to demonstrate that it is simple to implement synchronous data collection. Note that when we take the mean of the loss values, the constants of $\frac{1}{K}$ can be absorbed into the learning rates. We choose to write the constant explicitly, to emphasise that we are taking a mean.

Algorithm 3 REINFORCE with baseline and synchronous data collection

Require: Number of iterations: N, learning rates: $\alpha_{\boldsymbol{\theta}}, \alpha_{\boldsymbol{w}}$, differentiable policy $\pi(a|s, \boldsymbol{\theta})$, differentiable value function $V_\pi(s, \boldsymbol{w})$
  Initialise parameters $\boldsymbol{\theta}, \boldsymbol{w}$
  Initialise K environments
  for N iterations **do**
    for $t = 0, \ldots, T-1$ **do**
      Collect a batch of current states from each environment, $\begin{pmatrix} s_{t,1} & \ldots & s_{t,K} \end{pmatrix}^T$
      Apply actions $a_{t,k} \sim \pi_{\boldsymbol{\theta}}(\cdot|s_{t,k})$ in environment k for each $k \in \{1, \ldots, K\}$
      Receive rewards $\begin{pmatrix} r_{t,1} & \ldots & r_{t,K} \end{pmatrix}^T$
    end for
    for $t = 0, \ldots, T-1$ **do**
      for $k = 1, \ldots, K$ **do**
        $g_k \leftarrow \sum_{l=t+1}^{T} \gamma^{l-t-1} r_{l,k}$
        $A_k \leftarrow g_k - V_\pi(s_{t,k}, \boldsymbol{w})$
      end for
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \frac{1}{K} \sum_{k=1}^{K} A_k \nabla_{\boldsymbol{\theta}} \log \pi(a_{t,k}|s_{t,k}, \boldsymbol{\theta})$
      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \frac{1}{K} \sum_{k=1}^{K} A_k \nabla_{\boldsymbol{w}} V_\pi(s_{t,k}, \boldsymbol{w}))$
    end for
  end for

3.3 Summary

In this chapter, we provided a small yet informative glimpse into the vast field of reinforcement learning. We covered the basic MDP setting, which is a mathematical abstraction of sequential decision making processes. Policy gradient methods were introduced for finding optimal policies, including the earliest policy gradient methods REINFORCE (Algorithm 1) and its extension REINFORCE with baseline (Algorithm 2) (Williams (1992)).

In the next section, we will see how the MDP paradigm can be extended to situations where the agent is unable to observe the entire state, as is common in most real world applications.

We will also introduce multi-agent reinforcement learning, and extend the MDP paradigm to deal with multiple agents in a single environment.

Chapter 4

Generalisations of the Markov Decision Process

Various generalisations of the MDP framework exist for modelling more complex scenarios. One of the earlier extensions is the *partially observable Markov decision process* (POMDP) (Russell and Norvig (2010, Section 17.4)), which models scenarios where the agent cannot observe the full state and instead receives an incomplete and/or noisy *observation* of the environment. There are also various frameworks for multi-agent decision-making, the most general framework being *partially observable stochastic game* (POSG) (Albrecht et al. (2024, Section 3.4)), which encompasses, MDPs, POMDPs, certain models from the field of game theory (see Albrecht et al. (2024, Chapter 3)), and more. Of particular interest to us is the *decentralised partially observable Markov decision process* (Dec-POMDP) (Oliehoek and Amato (2016)), a POSG where agents share the same reward function, and hence seek to cooperate with one another to achieve their shared goal. In this chapter, we first take a detailed look at POMDPs, before investigating Dec-POMDPs and some of the challenges we face when solving them using *multi-agent reinforcement learning* (MARL).

4.1 Partially Observable Markov Decision Processes

This section follows Russell and Norvig (2010, Section 17.4) with some additions from Albrecht et al. (2024).

Under the MDP paradigm, we assume that our agent is capable of observing the state it is currently in. Unfortunately, this is almost never the case in the real world. Machines and robots can only observe the world through sensors which possess noise, rendering their readings not entirely reliable. Often, observations of a whole state are not attainable: vision is blocked by walls, economic agents have limited knowledge about their environment, etc. Fortunately, these situations and more can all be modelled using POMDPs.

In a POMDP setting, the agent can no longer observe the true state s_t, at each time step, but instead receives a noisy and/or incomplete *observation* of the state, $o_t \in \mathcal{O}$. We quantify the uncertainty in the observation via a *sensor model* $\mathcal{Z} : \mathcal{O} \times \mathcal{S} \to [0, 1]$. $\mathcal{Z}(o_t|s_t)$ is the conditional probability distribution on O_t given $S_t = s_t$. In some texts, e.g. Albrecht et al. (2024, Section 3.4), sensor models also condition on the previous action taken by our agent, but we do not consider this extension here. In most situations, the observation depends only on the state. For example, the images recorded by a camera on a robot depend only on the robot's position (state), not on the action it chose to arrive in that state. A POMDP is illustrated in Figure 4.1, and we give the definition from Russell and Norvig (2010, Section 17.4).

Definition 4.1 (Partially Observable Markov Decision Process)**.** *A Partially Observable Markov Decision Process (POMDP) is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, \mathcal{Z}, \mathcal{R}, \lambda)$ *where* $\mathcal{S}$ *is the state space,* $\mathcal{A}$ *is the*

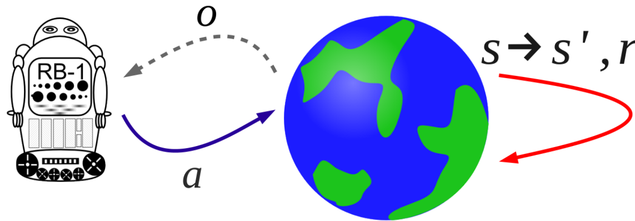

Figure 4.1: A Partially Observable Markov decision process. The situation is the same as in Figure 3.1, except now the agent receives a noisy and/or incomplete observation, o, of the new state, s', rather than observing the state itself. Image from Oliehoek and Amato (2016).

set of available actions, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is our state transition probability function, $\mathcal{O}$ is the set of possible observations our agent may receive, $\mathcal{Z} : \mathcal{O} \times \mathcal{S} \to [0, 1]$ is the sensor model, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, and $\lambda : \mathcal{S} \to [0, 1]$ is the initial state distribution.

As with the MDP framework, our goal is to find an optimal policy, π, that is, one which maximises the expected total return, $\mathbb{E}_\pi[G_0]$. A key difference under the POMDP framework is that the agent can no longer use policies of the form $\pi(a|s)$, nor are they able to evaluate value functions of the form $v_\pi(s)$, since the state s is unknown. While the Markov assumption ensures that all the information necessary for decision-making is contained within the current state, s_t, this is typically not the case for our observations. Instead, the agent must approximate the state using the *action-observation history*, $h_t = (o_0, a_0, \ldots, o_t, a_{t-1})$ (Albrecht et al. (2024, Section 3.4)), which contains all actions taken and observations received before and at time t.

When using function approximation methods as in Section 3.2, our approximate policy and value function are inherently robust to partial observability. As discussed in, Sutton and Barto (2018, Section 17.3), if an aspect of the state is unobservable, the policy and value function can be learnt such that the action distribution and values they output do not depend on that aspect of the state, and instead rely upon the action-observation history for providing similar information. A practical obstacle is that often our approximate policy and value functions are neural networks or linear models Sutton and Barto (2018, Section 9.4), which both expect fixed-length inputs.

This leaves us with two popular options, as discussed in Albrecht et al. (2024, Subsection 10.5.2). The first idea is to only use the k most recent action-observation pairs. This is suitable in situations where it helps to recall what has happened very recently, but observations and actions far in the past do not provide any benefits for decision-making. A commonly used special case of this is just using the latest observation as input to the approximate policy and value functions, as is done in Zhang et al. (2025). This is suitable for situations where only the agent's immediate observation is relevant; we cover such a case in Chapter 5. The second idea is to construct a policy that handles sequentially collected data, a common choice for this being a recurrent neural network (RNN) (Albrecht et al. (2024, Section 7.5), Zhang et al. (2023, Chapter 9)). Unfortunately, RNNs are outside the scope of this report.

A classical approach to this second idea is available when the sensor model $\mathcal{Z}$ is explicitly known by the agent. In this case, updating an approximate state variable with sequentially received noisy/incomplete measurements is well studied, and is referred to as the *filtering task* (Albrecht et al. (2024, Subsection 3.4.1), Thrun et al. (2005, Chapters 2-4)). Filtering involves maintaining a *belief state*, which is a probability distribution that measures how likely the agent thinks it is in any given state, given the action-observation history.

Definition 4.2 (Belief State)**.** *The agent's belief state at time $t > 0$ is the conditional probability*

distribution over $\mathcal{S}$ *given* $O_t = o_t, \ldots, O_1 = o_1$ *and* $A_{t-1} = a_{t-1}, \ldots, A_1 = a_1$*. That is*

$$b_t(s) = P(s|o_0, a_0, \ldots, o_t, a_{t-1})$$

where, as is often the case, P *is either a probability mass function or probability distribution function depending on whether* $\mathcal{S}$ *is countable or not. For* $t = 0$*, the belief state is the initial state distribution, that is*

$$b_0(s) = \lambda(s).$$

The belief state at time t, $b_t(s)$, can be computed recursively given the previous belief state, $b_{t-1}(s)$, and the most recent action and observation, a_{t-1} and o_t. This update is called the *Bayes filter*:

$$b_t(s) = \eta \mathcal{Z}(o_t|s) \sum_{s' \in \mathcal{S}} \mathcal{T}(s|a_{t-1}, s') b_{t-1}(s')$$

The derivation follows from applying Bayes' rule and taking advantage of conditional probabilities, and can be found in Thrun et al. (2005). In most cases, the Bayes Filter update step is intractable, due to the sum/integral over $\mathcal{S}$ (which can be large if not infinite), and the sum/integral required to calculate the normalisation constant, η, which faces the same issue of intractability as Z in Chapter 2. Fortunately, there are various algorithms which circumvent this issue. Kalman filters (Thrun et al. (2005, Chapter 3)) and their various extensions assume that the state and observations follow Gaussian distributions, which makes the update rule tractable. Particle filters and their various derivatives (Thrun et al. (2005, Section 4.3)) approximate the update step using Monte Carlo methods. We do not go into detail about these methods here.

As discussed in Albrecht et al. (2024, Subsection 3.4.1), computing and storing these exact belief states is computationally demanding for problems with large state spaces, which is why it is more common to pass action-observation histories to approximate policies and value functions directly, as discussed above. This is especially prevalent in multi-agent decision problems, which we cover in the next section, where in order to apply the Bayes filter update, agents would need to maintain probability distributions over the possible observations and actions of other agents, which would quickly become computationally infeasible as the number of agents increases.

4.1.1 Reward under partial observability

Briefly, we discuss how the reward function, $\mathcal{R}$ is affected by partial observability. It is commonly assumed that the agent does not know the reward function, so it is not an issue that the agent does not know the state and cannot, therefore, compute the reward. From a practical view, if the environment is simulated, then the true state is known by the simulator, and can be used to compute $\mathcal{R}(s', a, s)$. Alternatively, we can construct rewards based on the observation history rather than the state, like with our policy. This is necessary in cases where the environment is not simulated, for example, if we are trying to get a robot to do something in the real world. This is what is done in Zhang et al. (2025), and we see an example of history based rewards in Chapter 5.

4.2 Decentralised Partially Observable Markov Decision Processes

In the Dec-POMDP framework, we have multiple agents cooperating in the same environment. The agents collect their own observations, and use these to influence their own policies. All agents have the same reward function (Albrecht et al. (2024)). Communication between agents

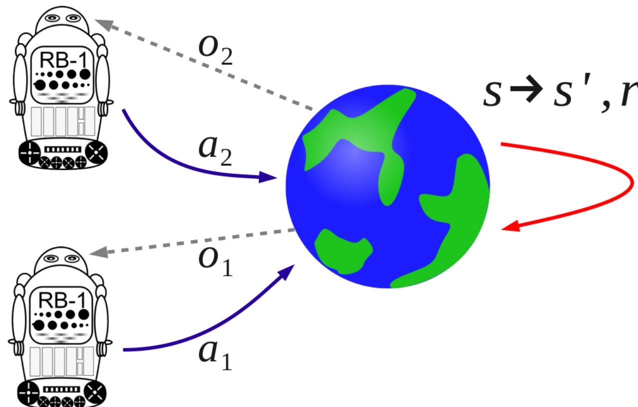

Figure 4.2: A Decentralised Partially Observable Markov decision process. The situation is the same as in Figure 4.1, except now we have multiple agents interacting cooperatively in the same environment. Each agent has the same reward function. Image from Oliehoek and Amato (2016).

is allowed. In the literature, communication typically refers to agents learning what information to communicate with one another, or learning an emergent protocol for communication (Zhu et al. (2024)), but in this report when we refer to communication we are referring only to agents sharing some or all of their observations with other agents, as in Zhang et al. (2025). Regardless of the types of communication allowed, Dec-POMDPs are very general. A Dec-POMDP is illustrated in Figure 4.2. We loosely follow the definition given in Oliehoek and Amato (2016):

Definition 4.3 (Decentralised Partially Observable Markov Decision Process)**.** *A Dec-POMDP is a tuple* $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, \mathcal{Z}, \mathcal{R}, \lambda)$*, where* $\mathcal{V} = \{1, \ldots, n\}$ *is the set of agents,* $\mathcal{S}$ *is the global state space,* $\mathcal{A} = \prod_{i \in \mathcal{V}} \{\mathcal{A}_i\}$ *is the joint set of all possible actions,* $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ *is the joint state transition probability function,* $\mathcal{O} = \prod_{i \in \mathcal{V}} \{\mathcal{O}_i\}$ *is the joint observation space,* $\mathcal{Z} : \mathcal{O} \times \mathcal{S} \to [0, 1]$ *is the joint sensor model,* $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ *is the global reward function, and* $\lambda : \mathcal{S} \to [0, 1]$ *is the joint initial state distribution.*

Our goal now is to find an optimal policy, π_i, for each agent, where as always, optimal means that the policy maximises $\mathbb{E}_\pi[G_0]$. We write, $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n)$, and call $\boldsymbol{\pi}$ the *joint policy.* As discussed in Section 4.1, the policy for agent i, π_i, is conditioned on agent i's action-observation history, $h_{i,t}$, and how much of the history is used as an input is problem dependent.

4.2.1 Centralisation and Decentralisation training and execution

We discuss some commonly made distinctions regarding the extent of the partial observability of agents in multi-agent reinforcement learning in general, following Albrecht et al. (2024). This has implications for how the Dec-POMDP (or more generally, POSG) is solved, including whether each agent maintains a unique policy and value function, or if the functions are shared between all agents.

Under the *centralised training* (CT) paradigm, agents have access to various centrally shared pieces of information. This could be as simple as agents sharing their action-observation histories, while still maintaining their own separate policies, or it could involve sharing a single central policy and value function which are computed using all action-observation histories. If there is no such central pool of information, we are said to be operating under the decentralised training (DT) paradigm. Typically, this means agents only observe their own action-observation history during training and use this to train individual policies and value functions, but it could be the case that a subsets of agents share information with one another, but not globally. An

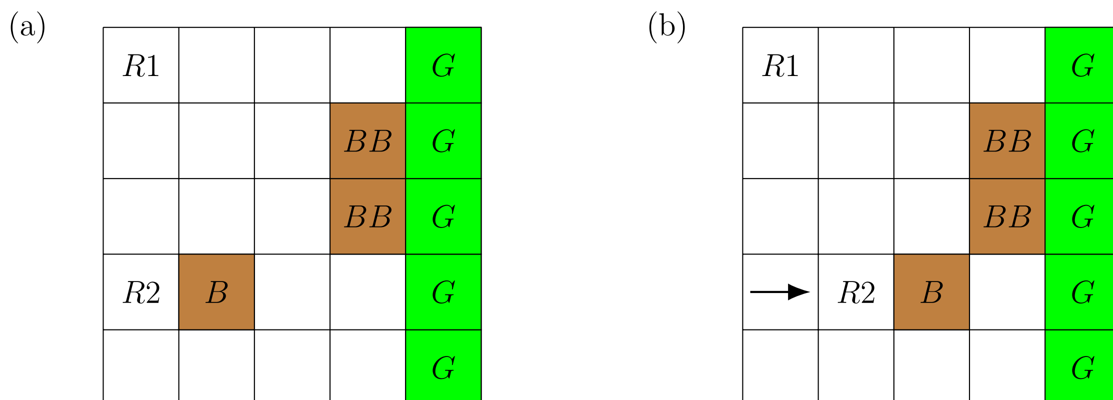

Figure 4.3: Box pushing. (a) Robots R1 and R2 move in a grid to push the small box B and large box BB (which spans two cells), onto the goal, G. The small box can be pushed by a single robot; the large box must be pushed by both robots at the same time. (b) When a robot moves (or robots move) into a box, the box moves in the same direction.

example of this form of DT, where *local communication* is allowed between agents, is given in Chapter 5. Similar distinctions apply during execution, leading to the centralised execution (CE) and decentralised execution (DE) paradigms.

Despite the name including the word "decentralised", Dec-POMDPs can be solved using the centralised training with centralised execution (CTCE) or centralised training with decentralised execution (CTDE) paradigms. This is, however, a departure from the formal definition of a Dec-POMDP, since agents' observations now depend on the previous actions of other agents (Albrecht et al. (2024, Subsection 9.1.1)), so it makes more sense to consider applying the CTCE and CTDE paradigms in the case of the more general POSG (Albrecht et al. (2024, Section 3.4)). Under the CTDE paradigm, agents have access to central information during training, which they can use, for example, to train a central value function that can evaluate policies from a global perspective, speeding up policy training (Albrecht et al. (2024, Subsection 9.1.3)). These policies, however, have to depend only on an agent's own action-observation history, so that the agent can take actions in a decentralised manner during execution. Whether we attempt to solve a Dec-POMDP/POSG using CTCE, CTDE, or decentralised training with decentralised execution (DTDE) depends on the decision process, computational considerations, and real world limitations.

As an example, consider a large swarm of robots in the real world. It may be infeasible to send all the data from each robot to a centralised computer, compute an action for each robot using a centralised policy and value function, then send the actions to the robots, within a short enough time frame to allow the robots to take actions in a reasonable amount of time. In this case, decentralised execution is necessary. If the robots are to be trained in a simulation, then it would be possible to use CTDE, since agents could easily be allowed access to all of the action-observation histories of all other agents. We cover a problem of this nature in Chapter 5, where we use the DTDE paradigm.

4.2.2 Challenges within MARL

Various hindrances to learning arise due to the presence of multiple agents in the environment. The following example illustrates the key issues

Example 4.1 (Box pushing)**.** *We present a modification of the cooperative box pushing problem presented in Seuken and Zilberstein (2007). Consider two robots in a grid trying to push a*

small box and a large box to a goal, as shown in Figure 4.3 (a). The robots can move one step up, down, left, or right. If a robot takes an action that would move it off the grid, it stays in place instead. If the robot moves into the small box, B, the box moves along with the robot, as shown in Figure 4.3 (b). For the robots to move the big box, represented in the figure by two cells both filled with BB and referred to in text as BB, the robots need to both push the box from the same side. The robots can only observe the four adjacent cells. The robots receive a large reward for moving the large box to the goal, a smaller reward for the smaller box, a negative reward for colliding, and a small negative reward in any other case. All entities are initialised randomly within the grid, with the exceptions that the boxes cannot start on the goal or in the left-most column. The scenario ends after 15 time steps.

We can model this as a Dec-POMDP. Denote $\boldsymbol{x}_1 = (x_1, y_1)$, $\boldsymbol{x}_2 = (x_2, y_2)$, $\boldsymbol{x}_B = (x_B, y_B)$, $\boldsymbol{x}_{BB} = (x_{BB}, y_{BB})$, *to be the coordinates of our entities, where* $\boldsymbol{x}_{BB}$ *denotes the coordinate of the top half of the big box, and all coordinates are in* $\{1, \ldots, 5\}^2$, *where* $(1,1)$ *is the top left of the grid and* $(5,5)$ *is the bottom right.*

- $\mathcal{V} = \{1, 2\}$
- $\mathcal{S} = \{(\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_B, \boldsymbol{x}_{BB}) : x_{entity}, y_{entity} \in \{1, \ldots, 5\}\}$
- $\mathcal{A}_i = \{up, down, left, right\}$
- $\mathcal{O}_i = \{(o_{i,up}, o_{i,down}, o_{i,left}, o_{i,right})\}$

$$\text{where } o_{i,d} = \begin{cases} B & \text{if } p(i,d) = \boldsymbol{x}_B \\ BB & \text{if } p(i,d) = \boldsymbol{x}_{BB} \\ wall & \text{if } p(i,d) \notin \{1, \ldots, 5\}^2 \\ nothing & \text{otherwise} \end{cases} \text{ and } p(i,d) = \begin{cases} (x_i, y_i + 1) & \text{if } d = up \\ (x_i, y_i - 1) & \text{if } d = down \\ (x_i - 1, y) & \text{if } d = left \\ (x_i + 1, y) & \text{if } d = right \end{cases}$$

- $\mathcal{R}(s, a, s') = \begin{cases} 100 & \text{if } (x_{BB}, y_{BB}) = (x_G, y_G) \quad \text{(big box on goal)} \\ 40 & \text{if } (x_B, y_B) = (x_G, y_G) \quad \text{(small box on goal)} \\ -5 & \text{if } (x_1, y_1) = (x_2, y_2) \quad \text{(robots collide)} \\ -1 & \text{otherwise} \end{cases}$

In this toy example, the state transition probability function, $\mathcal{T}$, *and sensor model,* $\mathcal{Z}$ *could be specified, but not defining them explicitly is illustrative of the typical case in which they are unknown. Similarly, we do not explicitly define the initial state distribution,* λ, *since we do not make use of it when using policy gradient methods, and hence can assume it is unknown. You will notice that the notation is heavy here due to the size of the state and observations, as well as the fact that there are many tuples due to the presence of multiple agents. Conceptually, the lines above simply capture the setup discussed at the start of this example.*

Let us consider a possible way in which this scenario could play out. If the first robot (R1) were to unhelpfully wander away from the box while the second robot (R2) pushes the box to the goal, the global reward received is 40. *If we use this reward to train R1's (currently very bad) policy, we would end up evaluating it favourably, due to the high reward.*

The example above introduces the *multi-agent credit assignment problem* (MACAP), not to be confused with the similar but different notion of credit assignment in the general RL literature (for example in Pignatelli et al. (2023)), which is the problem of determining how much a past action has influenced future rewards.

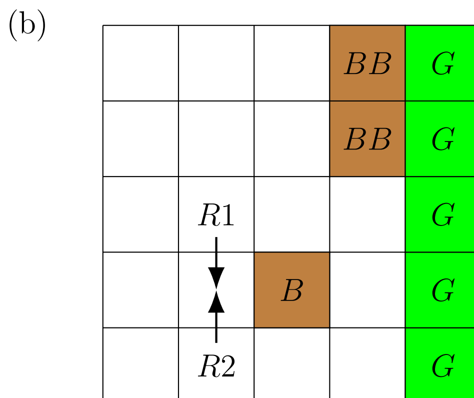

Figure 4.4: Box pushing continued. Due to each robot only partially observing the state, the robots are unable to see each other and may collide while trying to get to the box, B.

The MACAP involves determining how much each agent contributed to the global reward. Without this information, we struggle to accurately update each agent's policy, as in the example above. There are many methods in the literature for tackling this problem, as covered in Wong et al. (2023). Here, we only discuss the technique used in Zhang et al. (2025), which is to construct the global reward in such a way that it can be approximated by a sum of local rewards for each agent. This may require that the agent receives knowledge of other agents' states via communication. This technique is limited in its application, since it assumes that the problem can be segmented in such a way, which is not always the case.

Example 4.2 (Box pushing continued)**.** *To solve the MACAP, we introduce local rewards for each robot, based only on the robot's observations. We then sum these together to derive the global reward. By designing these local rewards properly, we can maintain the desired global behaviour which may have been encoded in the original global reward function, while also making sure the reward signals each robot receives are useful for training.*

$$\mathcal{R}_i(s, a, s') = \begin{cases} 100 & \textit{if } (x_{BB}, y_{BB}) = (x_G, y_G) \qquad \textit{(big box on goal)} \\ 0.8 & \textit{if robot } i \textit{ moves the small box towards the goal} \\ -5 & \textit{if } (x_1, y_1) = (x_2, y_2) \qquad \textit{(robots collide)} \\ -1 & \textit{otherwise} \end{cases}.$$

Formally, we can calculate whether the robot has moved the small box towards the goal by checking $||\boldsymbol{x}_{t,B} - \boldsymbol{x}_g||_1 < ||\boldsymbol{x}_{t-1,B} - \boldsymbol{x}_g||_1$, *and* $x_{t,i} = x_{t-1,B}$.

The robots no longer get a reward for putting the small box on the goal, but instead, receive a reward every time they move it towards the goal. They also receive a penalty of −1 for not doing anything of note, to further incentivise moving the boxes. The robots still both receive a big reward for the big box being put on the goal since they both have to move it together, so credit should be given to both robots when it reaches the goal.

One issue still remains, as illustrated in Figure 4.4: the robots may still collide with one another, since they do not know where the other robot is unless it is adjacent to them.

The MACAP is a symptom of a more general problem, that of *non-stationarity*. Most RL algorithms assume that the environment is stationary, meaning that it does not change unless interacted with by our agent. With multiple agents in the environment, however, this is no

longer the case. Our agent's observations of the environment alone may no longer be sufficient to determine an optimal action, since the actions of other agents may cause the environment to change in ways our robots cannot observe, such as the two robots in Example 4.1 trying to move into the same square and colliding. Various techniques are proposed in the literature to alleviate this issue, many of which are detailed in Papoudakis et al. (2019). We focus only on introducing communication between agents, which, as mentioned previously, also has benefits for the MACAP.

As discussed at the start of this section, we only consider simple communication where agents share some or all of their observations with some or all of the other agents. We can model this type of communication implicitly by expanding the observation space of the receiving agents. More elaborate communication, discussed in Zhu et al. (2024), may involve adding explicitly communication actions to the action space(s) of the communicator(s).

Example 4.3 (Box pushing continued again)**.** *We now allow the robots to implicitly communicate their observations to one another. Formally, this means we extend the observation space to include the observations of the other agent:*

$$\mathcal{O}_i = \{(o_{j,up}, o_{j,down}, o_{j,left}, o_{j,right}) : o_{j,direction} \in \{B, BB, nothing, robot, wall\}, j \in \{1, 2\}\}.$$

With this information, each robot may now learn to keep a square between themselves and the other robot. This would have to be relaxed when moving towards the big box, however, since the robots need to be adjacent to one another to push it successfully. This problem would be mitigated in a real-world scenario, where robot positions are vectors of real numbers, rather than integers, allowing for more nuanced movements.

Hopefully, this example has demonstrated the many challenges involved in multi-agent reinforcement learning.

4.3 Summary

We covered some common extensions to the MDP paradigm for modelling more complicated decision problems, particularly those where our agents can only partially observe the state. We discussed common techniques for dealing with partial observability, and covered filtering for POMDPs. Classical methods for solving POMDPs are covered in Russell and Norvig (2010, Section 17.4), and involve forming an MDP over belief states and solving it with a generalised version of dynamic programming.

We introduced multi-agent reinforcement learning and discussed using Dec-POMDPs to model the related decision problems. Having multiple agents in the environment leads to the issue of non-stationarity, where the agent's observations may no longer contain enough information for optimal decision making. We also covered the multi-agent credit assignment problem, where agents with a joint reward might be rewarded for the efforts of other agents, leading poor policies to be valued highly. In the next chapter, we use the techniques presented here to control a multi-robot swarm, demonstrating that the theory has real-world applicability.

Chapter 5

Robot Flocking Control

In this chapter we demonstrate a real-world application of Gibbs distributions and reinforcement learning to solve a modelling and control problem: the implementation of flocking behaviour in multi-robot systems. Flocking is a behaviour of groups of birds and other animals where the group moves together cohesively in a certain direction. Similar behaviour, called schooling, is exhibited by fish. Flocking provides various benefits to these animals, the key one being that it allows them all to quickly move to a single location without colliding with one another. This property makes it a desirable behaviour to implement in robotic swarms that are to be used for tasks like search and rescue or environmental surveillance.

An early seminal work on modelling flocking behaviour was made by Reynolds (1987), where three key requirements for accurate flocking behaviour were identified:

- Collision avoidance - flock mates must not collide with one another or the environment.
- Velocity matching - each flock member should have the same speed and heading direction.
- Flock centering - flock mates should attempt to stay close to one another.

Our goal is to control each robot to flock towards a target while maintaining the above behaviours. We model the control problem as a Dec-POMDP, and seek an optimal policy for each robot. We choose a decentralised model, where robots act as independent decision makers, rather than being controlled by a central authority, which often becomes computationally infeasible for large swarms (Park et al. (2022)).

In this chapter, we introduce two RL based methods for robot flocking control. The first is a simple model that uses potential based rewards, where we reward each robot for moving closer to the goal, moving closer to a set distance away from its neighbours, and for avoiding collisions. The second is a more complex model developed in Zhang et al. (2025), which uses Gibbs distribution based rewards. By modelling the interactions between members of the flock as a Gibbs distribution, we can specify energy functions that reward following the three flocking behaviours and penalise deviations from them. The joint distribution will then reward following all of these behaviours, and penalise deviations from them, making it an ideal reward function for our Dec-POMDP.

The methods we develop can be applied to a broad range of robots, with the only requirement being that they are capable of locomotion, thus this discussion applies to all forms of unmanned aerial vehicles (UAVs, often called drones), self-driving cars, bipedal humanoid robots, unmanned underwater vehicles (UUVs, often called underwater drones), and more.

5.1 Simple flocking

The simple flocking solution presented here is a modified version of the VMAS flocking scenario (Bettini et al. (2022)). We add a *tracking reward* which motivates the robots to move towards

the target, expand the collision reward to also penalise collisions with obstacles, edit the LIDAR sensor the robots possess to allow them to see other agents, rather than only obstacles, and tune how much each reward contributes to the total reward. The tracking reward is used in the similar *navigation* scenario (Bettini et al. (2022)), and is described in depth below.

5.1.1 The Dec-POMDP

With the computational benefits of decentralised decision-making in mind, we seek to model the flocking control problem as a Dec-POMDP. We consider each element of the tuple $(\mathcal{V}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{O}, \mathcal{Z}, \mathcal{R}, \lambda)$ in turn. $\mathcal{V} = \{1, \ldots, n\}$ is the set of all robots. $\mathcal{S}$ is the state space, which encompasses the states of all the robots. We do not define it explicitly, since we make no use of it. The initial state distribution, λ, is treated in the same way. It is also unlikely that agents would be aware of λ in most real world scenarios. The sensor model, $\mathcal{Z}$, and state transition probability function $\mathcal{T}$ are unknown, as is typical for real world problems.

Defining $\mathcal{A}$ and $\mathcal{O}$ first requires a discussion about the modelling of the robots' dynamics. We model the robot dynamics using standard Newtonian mechanics, that is, each robot, i, has position $\boldsymbol{p}_i$, velocity $\boldsymbol{v}_i = \frac{d\boldsymbol{p}_i}{dt}$, and acceleration $\boldsymbol{a}_i = \frac{d\boldsymbol{v}_i}{dt}$ vectors. All three vectors are in $\mathbb{R}^2$ for simplicity. The flock is given a target position, $\boldsymbol{p}_c$, to flock towards. This setup influences our action space, and we decide that robots are controlled by changing their accelerations, as in Zhang et al. (2025).

For the simple model, we use continuous actions. We allow the robots to change their accelerations up to a maximum magnitude. We have

$$\mathcal{A} = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{R}^2 : x, y \in [-3, 3] \right\}.$$

The use of continuous actions is chosen for ease of programming. We could have easily used discrete actions that allow the robots to move in a finite number of directions at a finite acceleration magnitude, as we will see shortly in the method by Zhang et al. (2025).

Each robot observes its own position and velocity, as well as the relative position between itself and the target. Finally, the agent observes its distance from neighbouring robots and obstacles in the environment. These distances are provided by LIDAR sensors on the robot. These LIDARs have 12 directions, evenly spread around the robot. Formally, we have that each agent, i, receives a tuple of observations $\boldsymbol{o}_i = (\boldsymbol{p}_i, \boldsymbol{v}_i, \boldsymbol{p}_i - \boldsymbol{p}_c, \boldsymbol{o}_{io})$, where $\boldsymbol{o}_{io} \in [0, 0.2]^{12}$ is a vector containing the distances to nearby objects in the 12 LIDAR directions. The LIDARs have a max range of 0.2.

The reward function can be split into three parts. We write the rewards as functions of the agents' action-observation histories, as discussed in Subsection 4.1.1, although each part of the reward uses a different subset of the histories.

The first reward is the *collision reward*,

$$\mathcal{R}_i^c(\boldsymbol{o}_{io,t}) = -\delta_c \sum_{l=1}^{12} \mathbb{1}(\{o_{il,t} = 0\}),$$ [1]

[1] In the actual simulation code, discussed in Chapter 6, we test $o_{il} \in (-0.005, 0.005)$ to avoid any floating-point related issues.

where $\delta_c > 0$ is a constant to be tuned as a hyperparameter, and $\mathbb{1}(A)$ is the *indicator function*,

$$\mathbb{1}(A) = \begin{cases} 1 & \text{if } s \text{ is a true statement} \\ 0 & \text{otherwise} \end{cases}$$

The collision reward penalises robots for colliding with each other and with objects.

The second reward is the position tracking reward,

$$\mathcal{R}_i^p(\boldsymbol{o}_{i,t}, \boldsymbol{o}_{i,t-1}) = \delta_p(||\boldsymbol{p}_{i,t-1} - \boldsymbol{p}_{c,t-1}||_2 - ||\boldsymbol{p}_{i,t} - \boldsymbol{p}_{c,t}||_2),$$

where $\delta_p > 0$ is another hyperparameter. The reward motivates the agents to move towards the target by providing them with a reward proportional to how much closer they have gotten to the target since the last time step. Rewards of this variety, where we take the difference between some metric at the previous time step and the current time step, are called *potential rewards* and are used frequently in RL (Müller and Kudenko (2025)).

The final part of the reward is the *distance reward*,

$$\mathcal{R}_i^d(\{\boldsymbol{o}_{i,t}, \boldsymbol{o}_{i,t-1}\}_{i\in\mathcal{V}}) = \frac{\delta_d}{n-1}\left(\sum_{j\neq i}(||\boldsymbol{p}_{i,t-1} - \boldsymbol{p}_{j,t-1}||_2 - d_r)^2 - \sum_{j\neq i}(||\boldsymbol{p}_{i,t} - \boldsymbol{p}_{j,t}||_2 - d_r)^2\right)$$

where $\delta_d > 0$ is another scaling hyperparameter, and d_r is the desired distance to maintain between robots. Recall that $\sum_{j\neq i}$ means the sum over all robots except i. The distance reward is best described from the ground up. First, we compute the distance between a pair of robots, then we compute the squared error between the actual distance between the pair and the desired distance we want the robots to maintain between each other. We then average this error over all the robots. This provides us with a measurement of the average failure to adhere to the flock centering criterion, while maintaining enough distance to avoid collisions, at time step $t-1$. We repeat this calculation for time step t, then take the difference between the two. This works in the same way as the position tracking reward: we want to minimise the mean squared error we computed, so if it is smaller at time step t than it was at time step $t-1$, we give the robot a positive reward, and if it it is larger at time step t than it was at time step $t-1$, then we give the robot a negative reward.

The total reward is simply the sum of these three components,

$$\mathcal{R}_i(\{\boldsymbol{o}_{i,t}, \boldsymbol{o}_{i,t-1}\}_{i\in\mathcal{V}}) = \mathcal{R}^d(\{\boldsymbol{o}_{i,t}, \boldsymbol{o}_{i,t-1}\}_{i\in\mathcal{V}}) + \mathcal{R}^p(\boldsymbol{o}_{i,t}, \boldsymbol{o}_{i,t-1}) + \mathcal{R}^c(\boldsymbol{o}_{io,t}).$$

5.1.2 The policy

The approximate policy and value function are both MLPs. Recall that MLPs take vectors as inputs, meaning we need to transform the information in the observation tuple o_i into a vector. To do so, we concatenate all the observation components together into a single vector in $\mathbb{R}^{18}$.

Each network has 2 hidden layers comprised of 256 perceptrons each. The final layer of π_i has four layers, one for each parameter of the normal distribution for each component of the action, which are then passed to a normal parameter extractor function (Section 3.2.1). The action components are then sampled from the implied distributions. The final layer of V_{π_i} is a single perceptron, representing the estimated value of the current state.

In the interests of improving this method's performance, we use two algorithms that we unfortunately do not have time to cover in the report. The first is Adam (Kingma and Ba

(2015)), which can be thought of as an improved version of batched stochastic gradient descent. The second is Proximal Policy Optimisation (PPO) (Schulman et al. (2017)), which can be thought of as an improved version of REINFORCE with baseline. PPO is extended to use batched environments in exactly the same way as we extended REINFORCE with baseline in Subsection 3.2.4.

We operate under the DTDE paradigm discussed in Subsection 4.2.1. In this case, each agent learns its own policy and own value function using just its current observation as an input. PPO is applied to each agent in exactly the same way as if this was a single agent problem. Since there is absolutely no communication between any of the agents, this method is referred to as *independent learning* (Albrecht et al. (2024, Section 9.3)).

We implement and experiment with this simple method in Chapter 6.

5.2 Gibbs distribution based flocking

We now turn our attention to the method developed by Zhang et al. (2025), in which a Gibbs distribution is formed over the observations of our robots. We construct energy functions which penalise deviations from the three desired flocking behaviours and reward adherence to them. The second major change is in the policy. We no longer use a simple MLP as our policy, but instead use the *attention structure* developed by Zhang et al. (2025). This makes use of *attention layers*, which were originally introduced for performing natural language translation (Vaswani et al. (2017)), but have since proven incredibly effective in a variety of machine learning tasks including computer vision (Dosovitskiy et al. (2020)), robotics (Zitkovich et al. (2023)), and of course natural language processing, where they form the basis for large language models (Radford et al. (2019)). In this application, the attention layers allow our robots to pay attention to the behaviour of their neighbours, preempting their movements and allowing for improved collision avoidance.

5.2.1 The Dec-POMDP

The robot dynamics are unchanged. Like with the simple model, all vectors are in $\mathbb{R}^2$, although Zhang et al. (2025) claim the algorithm can be extended to $\mathbb{R}^3$ with almost no modifications. Robots are controlled by changing their accelerations as in the simple method, except now we use a finite action set instead.

Our action space is finite and formed of two types of actions, $\mathcal{A}_i = \mathcal{A}_{i,d} \cup \mathcal{A}_{i,c}$. The first is the *discrete action set*, $\mathcal{A}_{i,d} = \{\boldsymbol{a}_i = M\boldsymbol{e} : M \in \{M_1, \ldots, M_m\}, \boldsymbol{e} \in \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_l\}\}$. Each action in this set changes robot i's acceleration, $\boldsymbol{a}_i$, to have magnitude M in direction $\boldsymbol{e}$, where the choice of magnitudes and directions are finite, and directions are spread equally around the robot. To be precise, each $\boldsymbol{e}_i$ points at an angle of $\frac{2\pi i}{l}$ from the positive x-axis, as illustrated in Figure 5.1. The second set is the *continuous policy set*, $\mathcal{A}_{i,c} = \{-\boldsymbol{v}_i, \boldsymbol{v}_c - \boldsymbol{v}_i\}$. These actions, according to Zhang et al. (2025), are introduced to allow the robot to decelerate smoothly and improve robots' ability to match the desired velocity, for the swarm, which in this method is defined explicitly as $\boldsymbol{v}_c$

In order to define the observation space $\mathcal{O}$, we must first define the neighbourhood of robot i: $\mathcal{N}_i = \{j \in \mathcal{V} : ||i - j||_2 < d_{in}\}$, where d_{in}, the *interaction range*, is a hyperparameter that determines the radius of robot neighbourhoods.

Each robot receives an observation $\boldsymbol{o}_i = (\boldsymbol{o}_{ii}, (\boldsymbol{o}_{ij})_{j\in\mathcal{N}_i}, (A_j)_{j\in\mathcal{N}_i}, \boldsymbol{o}_{io})$. $\boldsymbol{o}_{ii} = (\boldsymbol{p}_i, \boldsymbol{v}_i, \boldsymbol{p}_c, \boldsymbol{v}_c)$ is

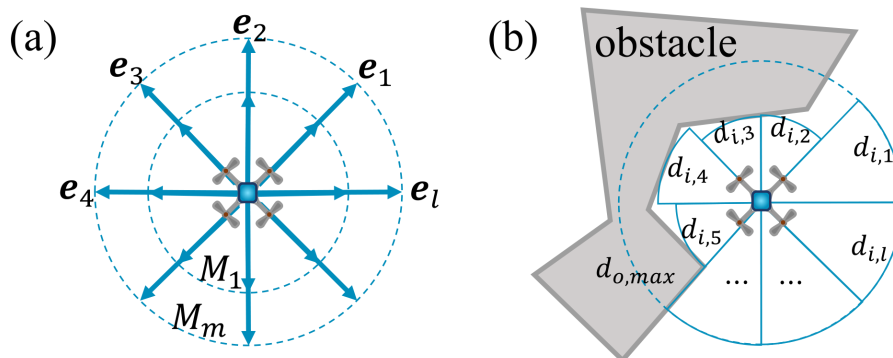

Figure 5.1: Actions and obstacle detection. (a) illustrates actions in $\mathcal{A}_{i,d}$. (b) illustrates how $\boldsymbol{o}_{io}$ is calculated using LIDAR measurements. Figure is from Zhang et al. (2025).

called the *ego state*, since it stores information about the robot's self. $\boldsymbol{o}_{ij} = (\boldsymbol{p}_j - \boldsymbol{p}_i, \boldsymbol{v}_j - \boldsymbol{v}_i)$ is called the *relative state*, and contains the relative position and velocity to the robot's neighbour j. Similarly to the simple method, $\boldsymbol{o}_{io} = \begin{pmatrix} d_{i,1} & \dots & d_{i,l} \end{pmatrix}^T$ is the vector of distances to objects in the environment (i.e., anything other than other robots in the flock) in the l directions $\{\boldsymbol{e}_1, \dots, \boldsymbol{e}_l\}$, as with the actions. The objects are sensed by LIDARs on the robots.

As discussed in Section 4.2, in the most general case, the sensor model is conditioned on the current state as well as the previous joint action. In this application, the previous joint action includes the communication to neighbours action, which provides our robot with A_j, the previous action distribution of neighbour j. To understand what this means, we must very briefly take a look at the policies for each robot. Unlike in Section 4.1, we do not use a Bayes filter to define a belief state, instead, we take the naïve approach of simply approximating the state at time t, s_t, with our observation at time t, $o_{i,t}$, thus, the policy for robot i is $\pi_{\boldsymbol{\theta}_i}(a_i|o_{i,t})$. This is a probability distribution over the action set $\mathcal{A}_i$. A_j, then, simply denotes $\pi_{\boldsymbol{\theta}_j}(a_j|o_{j,t-1})$, which tells us the probability of robot j having taken each of its possible actions at the previous time step. This distribution is communicated by each robot to their neighbour at the end of each time step. The reason we include this in our observation is to allow our robots to preempt their neighbours' movements. This is discussed in more detail in Section 5.2.5.

The reward function, $\mathcal{R}$, is constructed based on a Gibbs distribution over the robots, where each robot is modelled as a node in a graph. As such, we must first specify this Gibbs distribution and its related energy functions before we define $\mathcal{R}$ in Section 5.2.4.

5.2.2 The Gibbs Distribution

In the previous section, we defined the neighbourhood of a robot to be all other robots within a certain radius, the interaction distance, d_{in}. We now use this to construct a graphical representation of our swarm. Unlike in Chapter 2, we have defined our neighbourhood, and now use it to construct the edges in our graph, rather than defining the neighbourhood based on the edges in our graph. The benefit of this is that it makes the modelling of the graph trivial. We define our graph to be $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{E} = \{(i,j) : i \in \mathcal{V}, j \in \mathcal{N}_i\}$. Note that the edges here are *ordered pairs*. This is non-standard, but it plays a key role in proving Proposition 5.2, and follows the definition given in Zhang et al. (2025). Fortunately, this does not affect the form of the probability density function for our Gibbs distribution, which remains the same as in Chapter 2.

The random variables in this formulation are O_i, the observations. To see why this is reasonable,

consider that we must define energy functions over these random variables, in order to model the desired behaviour of our robots. Since all the robots know is their observations, and, in a real world setting, robots must be able to compute their own rewards, which we construct using these energy functions, we must have that the random variables assigned to each robot are their observations.

5.2.3 Energy Functions

To construct a Gibbs distribution over $\mathcal{G}$, we need to specify our energy functions. This problem lends itself naturally to the specification of pairwise and unary energy functions, since our goal for the robots is simply for them to not collide with one another (which is a pair-based notion), not to crash into obstacles, to move cohesively, and to move towards the goal (these are all unary notions). The total energy is therefore

$$E(\boldsymbol{o}) = \sum_{(i,j)\in\mathcal{E}} \psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j) + \sum_{i\in\mathcal{V}} \psi_u(\boldsymbol{o}_i),$$

where ψ_u is the total unary energy, and ψ_p is the total pairwise energy. These themselves are sums of smaller energy functions, which each serve an individual role. In the remainder of this section, we discuss each constituent energy function in turn, then combine them into an unnormalised joint distribution which we use as our reward.

Pairwise Energy

The pairwise energy is composed of two smaller energies,

$$\psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j) = \psi_d(\boldsymbol{o}_i, \boldsymbol{o}_j) + \psi_v(\boldsymbol{o}_i, \boldsymbol{o}_j).$$

The first is the *position alignment energy*,

$$\psi_d(\boldsymbol{o}_i, \boldsymbol{o}_j) = c_{p1}[1 - \exp(-c_{p2}(d_{ij} - d_r))]^2 - c_{p1},$$

where $d_{ij} = ||\boldsymbol{p}_i - \boldsymbol{p}_j||_2$ is the distance between robots i and j, d_r is the desired distance between robots, and c_{p1} and c_{p2} are coefficients which determine how much this energy term contributes to the total energy.

The position alignment energy is based on the *Morse potential* (Morse (1929)) from chemistry, where it models energy wells. Here, it maintains separation between robots, helping to avoid collisions. Recall that lower energy means a higher joint probability. As shown in Figure 5.2 the energy is at a minimum when $d_{ij} = d_r$, which is suitable as this is the desirable state of the system. As robots get further away from one another, $d_{ij} > d_r$, the energy increases, since this is less desirable. As robots get closer together, $d_{ij} < d_r$, the energy increases sharply, heavily penalising the robots for being too close and risking a collision. These properties are given formally in the following proposition. The statements are given without proof in Zhang et al. (2025), and here we elaborate on them.

Proposition 5.1 (Properties of the Position Alignment Energy)**.** *For $\boldsymbol{o}_i$ and $\boldsymbol{o}_j$ such that $d_{ij} = x$, we denote $\psi_d(\boldsymbol{o}_i, \boldsymbol{o}_j)$ as $\psi_d|_{d_{ij}=x}$. We have the following properties of ψ_d:*

1. *$\psi_d(\boldsymbol{o}_i, \boldsymbol{o}_j)$ is at a minimum when $d_{ij} = d_r$*

2. *$\psi_d|_{d_{ij}=d_r} = -c_{p1}$*

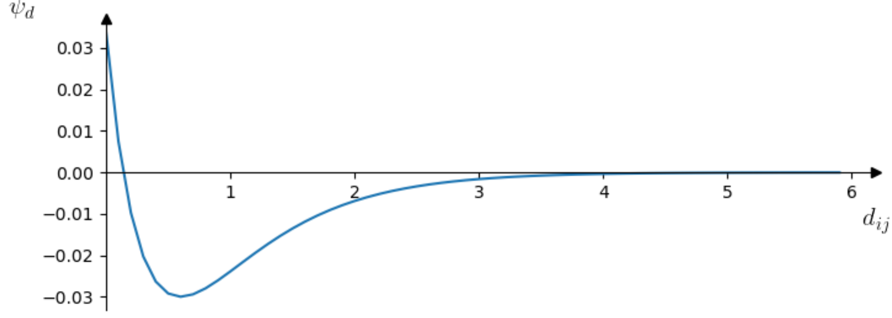

Figure 5.2: Position alignment energy. $d_r = 0.6, c_{p1} = 0.03, c_{p2} = 1$. When the robots maintain their desired separation, the energy is at a minimum.

3. For all $\varepsilon \in (0, d_r)$*,* $\psi_d|_{d_{ij}=d_r-\varepsilon} > \psi_d|_{d_{ij}=d_r+\varepsilon}$

Proof. The first two statements are trivial. We have $c_{p1} > 0$ and $[1-\exp(-c_{p2}(d_{ij}-d_r))]^2 \geq 0$, thus, if we can get $[1-\exp(-c_{p2}(d_{ij}-d_r))]^2 = 0$ for some value of d_{ij}, this must be the minimum. We see that $d_{ij} = d_r$ satisfies this. Indeed, we have

$$\begin{aligned}\psi_d|_{d_{ij}=d_r} &= c_{p1}[1-\exp(-c_{p2}(d_r-d_r))]^2 - c_{p1} \\ &= c_{p1}[1-\exp(0)]^2 - c_{p1} \\ &= c_{p1}[0]^2 - c_{p1} \\ &= -c_{p1}\end{aligned}$$

For the third statement, we have $\psi_p|_{d_{ij}=d_r-\varepsilon} = c_{p1}[1-\exp(c_{p2}\varepsilon)]^2 - c_{p1}$ and $\psi_p|_{d_{ij}=d_r+\varepsilon} = c_{p1}[1-\exp(-c_{p2}\varepsilon)]^2 - c_{p1}$. From this we see that, setting $x = c_{p2}\varepsilon$

$$\psi_p|_{d_{ij}=d_r-\varepsilon} > \psi_p|_{d_{ij}=d_r+\varepsilon} \iff |1-e^x| > |1-e^{-x}|.$$

First, observe that $x = c_{p2}\varepsilon > 0$, so $e^x > 1$ and $1-e^x < 0$. This means $|1-e^x| = e^x - 1$. Similarly, $e^{-x} < 1$, so $1-e^{-x} > 0$ and $|1-e^{-x}| = 1-e^{-x}$.

Now we show that $e^x - 1 > 1 - e^{-x}$ iff $x > 0$.

$$\begin{aligned}& 1-e^{-x} < e^x - 1 \\ \iff & 2 < e^x + e^{-x} \\ \iff & e^{2x} - 2e^x + 1 > 0 \\ \iff & (e^x-1)^2 > 0 \\ \iff & e^x > 1 \\ \iff & x > 0.\end{aligned}$$

Since $c_{p2}\varepsilon > 0$, we have shown that $\psi_p|_{d_{ij}=d_r-\varepsilon} > \psi_p|_{d_{ij}=d_r+\varepsilon}$. □

The second pairwise energy term in the *velocity alignment energy*,

$$\psi_v(\boldsymbol{o}_i, \boldsymbol{o}_j) = c_v \max\left(0, -\boldsymbol{v}_{ji}^T \frac{\boldsymbol{p}_{ji}}{||\boldsymbol{p}_{ji}||^2}\right),$$

where c_v is, again, a coefficient that determines how much this energy term contributes to the total energy. $\boldsymbol{p}_{ji} = \boldsymbol{p}_j - \boldsymbol{p}_i$ is called the *relative position* of j with respect to i. It is the position

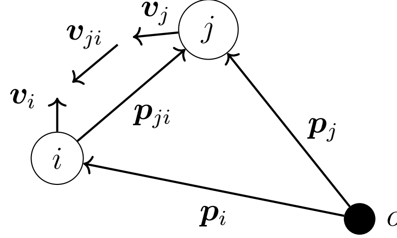

Figure 5.3: A demonstration of the velocity alignment energy. Two robots move such that their relative velocity opposes their relative position, i.e., they are moving closer together, and are thus penalised by the velocity alignment energy. ($\boldsymbol{v}_{ji}$ not to scale).

of j when we consider the origin of our coordinate space to be at $\boldsymbol{p}_i$. Similarly, the *relative velocity* of j with respect to i, $\boldsymbol{v}_{ji} = \boldsymbol{v}_j - \boldsymbol{v}_i$, can be thought of as the velocity of robot j from the perspective of robot i. If robot i were to perceive itself as stationary, then it would perceive robot j moving with velocity $\boldsymbol{v}_{ji}$. We make this more apparent with an example.

Example 5.1 (Velocity alignment energy)**.** *Two robots, robot A and robot B, are moving towards each other, as shown in Figure 5.3. We have*

$$\boldsymbol{p}_i = \begin{pmatrix}-4\\0.8\end{pmatrix}, \boldsymbol{p}_j = \begin{pmatrix}-2\\2.5\end{pmatrix}, \boldsymbol{v}_i = \begin{pmatrix}0\\0.8\end{pmatrix}, \boldsymbol{v}_j = \begin{pmatrix}-1\\0.1\end{pmatrix}, \boldsymbol{p}_{ji} = \begin{pmatrix}2\\1.7\end{pmatrix}, \boldsymbol{v}_{ji} = \begin{pmatrix}-1\\-0.9\end{pmatrix}.$$

Intuitively, we would expect the velocity alignment energy to penalise the robots in this situation, since they are on a collision course with one another: recall that the energy seeks to minimise velocity in the direction of relative position, but in this case, the relative velocity is pointing along the same direction.

We have $||\boldsymbol{p}_{ji}||_2^2 = 2^2 + 1.7^2 = 6.89$ *and* $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} = -3.53$*, so overall*

$$\psi_v(\boldsymbol{o}_i, \boldsymbol{o}_j) = c_v \max\left(0, \frac{3.53}{6.89}\right) = 0.512 c_v \quad \textit{(to 3 s.f).}$$

Recall that $c_v > 0$*, so this is indeed a penalty, since the minimum for* ψ_v *is 0.*

It is clear from the definition that $\psi_v(\boldsymbol{o}_i, \boldsymbol{o}_j)$ is at a minimum of 0 iff $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} \geq 0$.

Unary Energy

The next five energy functions depend only on the state of a singular robot, and are therefore called unary energy functions. While the pairwise energies take care of avoiding collisions with other robots and flock centering, the unary energies take care of velocity matching and avoiding collisions with obstacles.

The unary energy is comprised of five smaller energies,

$$\psi_u(\boldsymbol{o}_i, \boldsymbol{a}_i) = \psi_t + \psi_k + \psi_c + \psi_o + \psi_b$$

The first and arguably most important energy is the *position tracking energy*,

$$\psi_t(\boldsymbol{o}_i) = c_{t1}(||\boldsymbol{p}_i - \boldsymbol{p}_c||_2 - c_{t2}).$$

This energy has the same job as the position tracking energy from Section 5.1, that is, it motivates the robots to move towards the target. Unlike the potential reward used in Section 5.1, however,

this energy simply takes the distance between the robot and the target, then subtracts c_{t2} and scales the result by c_{t1}. This provides very fine control over how this energy will affect the total energy, and hence the total reward, i.e., we have a lot of flexibility in how much the robots prioritise the optimal flocking behaviours when they contradict the goal of moving towards the target. Clearly, $\psi_t(\boldsymbol{o}_i)$ is at a minimum when $\boldsymbol{p}_i = \boldsymbol{p}_c$, that is, the robots reach the target.

The next unary energy is the motion smoothness energy,

$$\psi_k(\boldsymbol{o}_i) = c_k||\boldsymbol{v}_i - \boldsymbol{v}_c||_2^2.$$

This energy is entirely dedicated to velocity matching. Quite simply, the robot is penalised for larger values of the Euclidean distance between the robot's velocity and the desired velocity for the flock, $\boldsymbol{v}_c$. As with the position tracking energy, the motion smoothness energy is at a minimum when $\boldsymbol{v}_i = \boldsymbol{v}_c$, i.e., the robot is moving at its desired velocity, and since all robots are trying to minimise this energy, the flock is incentivised to move with the same velocity.

The third unary term is the control optimisation energy, $\psi_c(\boldsymbol{a}_i) = c_c||\boldsymbol{a}_i||_2^2$[2]. This energy penalises the robots for taking choosing actions which lead to them having a high acceleration, as this can lead to unstable motion. Unlike the previous two energies, minimising this energy would not be ideal, as then the robots would cease to move after a short time. This tells us that the value for c_c should be quite small, and indeed, Zhang et al. (2025) set $c_c = 0.00001$, to prevent this energy from overly impeding the motion of the flock.

The next two terms are for avoiding obstacles in the environment, and they mimic the energies used for the pairwise energies. The first is the *obstacle avoidance energy*, which is once again based on the Morse potential (Morse (1929)), with a couple of changes.

$$\psi_o(\boldsymbol{o}_i) = c_{o1}[1 - \exp(-c_{o2}\min(0, d_{oi} - d_{or}))]^2,$$

where d_{or} is the reaction distance to obstacles. If a robot is closer than this distance to an obstacle, it is unlikely to be able to course correct in time if it is going to crash. Unlike the position alignment energy, we do not subtract c_{o1} from the energy. This means the minimum of this energy is 0, indeed, for $d_{oi} > d_{or}$, i.e., the desirable case, where the robot has time to react to any oncoming obstacles, we have

$$\begin{aligned}\psi_o(\boldsymbol{o}_i) &= c_{o1}[1 - \exp(-c_{o2} * 0)]^2 \\ &= c_{o1}[1 - 1]^2 \\ &= 0.\end{aligned}$$

For $d_{oi} > d_{or}$, however, we have

$$\begin{aligned}\psi_o(\boldsymbol{o}_i) &= c_{o1}[1 - \exp(-c_{o2}(d_{oi} - d_{or}))]^2 \\ &= c_{o1}[1 - \exp(\varepsilon)]^2 > 0,\end{aligned}$$

for $\varepsilon = -c_{o2}(d_{oi} - d_{or}o) > 0$. This means we penalise the robot for so close to obstacles that it cannot react to them.

The final energy is the *collision aversion energy*, which limits robot velocity towards obstacles,

$$\psi_b(\boldsymbol{o}_i) = c_b \max\left(0, -\frac{\boldsymbol{v}_{oi} \cdot \boldsymbol{p}_{oi}}{||\boldsymbol{p}_{oi}||^2}\right),$$

[2] Zhang et al. (2025) give this energy as $\psi_c(\boldsymbol{a}_i) = c_c\boldsymbol{a}_i^2$, which is non-standard notation, but we take it to mean either $||\boldsymbol{a}_i||_2^2$, $\boldsymbol{a}_i^T\boldsymbol{a}_i$, or $\boldsymbol{a}_i \cdot \boldsymbol{a}_i$, which all, fortunately, mean the same thing.

Figure 5.4: A demonstration of how more edges can degenerate collision avoidance. (a) shows a configuration with 14 edges, whereas (b) shows the same configuration with 18 edges. The red robot is pulled towards the centre of its neighbourhood, but this degenerates its ability to avoid the obstacle in the centre of the flock. Image from Zhang et al. (2025).

where $\boldsymbol{p}_{oi} = \boldsymbol{p}_{o,min} - \boldsymbol{p}_i$ is the relative position of the nearest obstacle, and $\boldsymbol{v}_{oi} = \boldsymbol{v}_{o,min} - \boldsymbol{v}_i$ is the relative position of the nearest obstacle. This is the definition given by Zhang et al. (2025), although, all obstacles are stationary, so it only makes sense that we have $\boldsymbol{v}_{o,min} \equiv \mathbf{0}$. This energy works in exactly the same way as the collision avoidance energy, $\psi_v(\boldsymbol{o}_i, \boldsymbol{o}_j)$, except with obstacles instead of a second robot.

Total Energy

Since robot neighbourhoods are defined based on the distance between robots, the number of edges in our graph changes over time. The effect of the position alignment energy, as we discussed previously, is to maintain a separation distance of d_r between all neighbours in a neighbourhood. Thus, each robot is forced into the centre of their own neighbourhood. This is exactly what we desire for fulfilling the flock centering criterion, but it leads to more collisions with obstacles in the environment if followed too rigidly, as in Figure 5.4. To combat this issue, Zhang et al. (2025) introduced the *normalised pairwise energy*,

$$E_p(\boldsymbol{o}) = \frac{|\mathcal{V}|}{|\mathcal{E}|} \sum_{(i,j)\in\mathcal{E}} \psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j). \tag{5.1}$$

By dividing the pairwise energy by the number of edges, we reduce the influence of the pairwise energy on the total energy for flocks with lots of edges, which means the flock will value unary terms like the obstacle avoidance energy and collision aversion energy, preventing the situation in Figure 5.4.

The inclusion of $|\mathcal{V}|$ in the normalised pairwise energy serves a different purpose. For a fixed number of obstacles, the more robots we have in the flock, the higher the chance that a collision will be with another robot rather than with an obstacle. Thus, as the number of robots in the flock increases, we want each robot to prioritise robot-to-robot collision avoidance more and more, i.e., we want to *increase* the weight our pairwise terms have on the total energy, which is why we *multiply* the pairwise energy by $|\mathcal{V}|$. Since the number of robots in the flock does not change, we could have simply absorbed this term into c_{p1} and c_v, but then we would have to retune these coefficients whenever we wanted to apply this algorithm to a flock of a different size.

The total energy is now redefined to be

$$E(\boldsymbol{o}) = E_p(\boldsymbol{o}) + E_u(\boldsymbol{o}), \tag{5.2}$$

where $E_u(\boldsymbol{o}) = \sum_{i\in\mathcal{V}} \psi_u(\boldsymbol{o}_i)$. The total energy captures all of our intentions for the flock behaviour, and by choosing a $\boldsymbol{o}$ that minimises it, we have determined an optimal configuration of our flock. In this sense, our energy function can evaluate the quality of the flock's current joint observation, and we can use it to evaluate how well the robots are performing at the flocking task. It is in this sense, a reward function. The issue is, reward functions are something we aim to maximise, but the total energy is something we aim to minimise, thus we instead consider the unnormalised joint distribution. Recall from Section 2.1 that,

$$P(\boldsymbol{o}) \propto \exp(-E(\boldsymbol{o})).$$

The constant of proportionality, $\frac{1}{Z}$, would be intractable for this application, but thankfully, reward functions are basically identical up to a multiplicative constant.

5.2.4 Decentralising reward

Our decision to use DTDE to solve the flocking problem means we need a way to segment the global reward into local rewards for each robot. We also seek to tackle the MACAP by constructing the local rewards such that robots are rewarded for their own performance alone.

To this end, Zhang et al. (2025) define the *approximate pairwise energy*,

$$\widehat{E}_p(\boldsymbol{o}) = \sum_{i\in\mathcal{V}} \frac{1}{|\mathcal{N}_i|} \sum_{j\in\mathcal{N}_i} \psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j).$$

This energy is a sum over each robot, so we can express the total approximate energy as

$$\begin{aligned} \widehat{E}(\boldsymbol{o}) &= E_u(\boldsymbol{o}) + \widehat{E}_p(\boldsymbol{o}) \\ &= \sum_{i\in\mathcal{V}} \psi_u(\boldsymbol{o}_i, \boldsymbol{a}_i) + \frac{1}{|\mathcal{N}_i|} \sum_{j\in\mathcal{N}_i} \psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j), \end{aligned}$$

which gives us an approximate reward of

$$\begin{aligned} \widehat{r} &= \exp(-\widehat{E}(\boldsymbol{o})) \\ &= \prod_{i\in\mathcal{V}} \exp\left(-\psi_u(\boldsymbol{o}_i, \boldsymbol{a}_i) - \frac{1}{|\mathcal{N}_i|} \sum_{j\in\mathcal{N}_i} \psi_p(\boldsymbol{o}_i, \boldsymbol{o}_j)\right) \\ &= \prod_{i\in\mathcal{V}} r_i \end{aligned}$$

where r_i are our local rewards for each robot.

It is left to demonstrate that maximising these local rewards leads to maximising the global reward. First, observe that maximising $\prod_{i\in\mathcal{V}} r_i$ is equivalent to maximising r_i for all $i \in \mathcal{V}$, and that maximising r_i requires minimising both $E_u(\boldsymbol{o})$ and $\widehat{E}_p(\boldsymbol{o})$, whereas maximising r requires minimising both $E_u(\boldsymbol{o})$ and $E_p(\boldsymbol{o})$. Our goal then is to show that $E_p(\boldsymbol{o})$ and $\widehat{E}_p(\boldsymbol{o})$ share the same minimum condition and value.

Proposition 5.2. *The total pairwise energy, $E_p(\boldsymbol{o})$, and the approximate pairwise energy, $\widehat{E}_p(\boldsymbol{o})$, share the same minimum condition and value, and this condition and value allows for the maximisation of total reward, G_t, at any time step t.*

Proof. As shown in Proposition 5.1, ψ_d is at a minimum of $-c_{p1}$ when $d_{ij} = ||\boldsymbol{p}_{ji}||_2 = d_r$. We also know that ψ_v is at a minimum of 0 when $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} \geq 0$. The minimum point of E_p and $\widehat{E}_p$ must therefore satisfy $||\boldsymbol{p}_{ji}||_2 = d_r$ and $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} \geq 0$ for all $(i,j) \in \mathcal{E}$.

We have $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} \geq 0$ iff $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} > 0$ or $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} = 0$. We now show that if $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} > 0$ for some $(i,j) \in \mathcal{E}$, then E_p and $\widehat{E}_p$ are at a minimum, but will move out of this minimum at a future time step of the decision process, and hence achieving this minimum will not suffice for maximising the total reward, G_t.

We define the angle between $\boldsymbol{v}_{ji}$ and $\boldsymbol{p}_{ji}$ to be $\theta \in (-\pi, \pi]$. Recall from Linear Algebra that $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} = ||\boldsymbol{v}_{ji}||_2 ||\boldsymbol{p}_{ji}||_2 \cos(\theta)$, hence $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} > 0$ implies $\cos(\theta) > 0$, thus $\theta \in \left(-\frac{\pi}{2}, \frac{\pi}{2}\right)$. This means the two robots will move towards one another at the next time step, reducing $||\boldsymbol{p}_{ji}||_2$ and thereby increasing ψ_d, which means E_p and $\widehat{E}_p$ will not be at a minimum.

We now have that the minimum of E_p and $\widehat{E}_p$ must satisfy $||\boldsymbol{p}_{ji}||_2 = d_r$ and $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} = 0$. If $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji} = 0$, then $\boldsymbol{v}_{ji}$ is perpendicular to $\boldsymbol{p}_{ji}$, and one robot will move with circular motion around the other. This means the distance between the two, namely $||\boldsymbol{p}_{ji}||_2$, will be kept constant, and this minimum is sustainable over all time steps t.

Now we show that the two energies share a minimum value. Given $||\boldsymbol{p}_{ji}||_2 = d_r$ and $\boldsymbol{v}_{ji} \cdot \boldsymbol{p}_{ji}$, we have

$$
\begin{aligned}
E_p(\boldsymbol{o}) &= \frac{|\mathcal{V}|}{|\mathcal{E}|} \sum_{(i,j)\in\mathcal{E}} -c_{p1} \\
&= \frac{|\mathcal{V}|}{|\mathcal{E}|} \sum_{i\in\mathcal{V}} \sum_{j\in\mathcal{N}_i} -c_{p1} \\
&= -c_{p1} \frac{|\mathcal{V}|}{|\mathcal{E}|} \sum_{i\in\mathcal{V}} -|\mathcal{N}_i| \\
&= -c_p \frac{|\mathcal{V}|}{|\mathcal{E}|} |\mathcal{E}| \\
&= -c_{p1} |\mathcal{V}|.
\end{aligned}
$$

Note that our non-standard definition of $\mathcal{E}$ is key here, if we did not use ordered pairs in the definition, this derivation would not work, as we would be double counting edges.

We also have

$$
\begin{aligned}
\widehat{E}_p &= \sum_{i\in\mathcal{V}} \frac{1}{|\mathcal{N}_i|} \sum_{j\in\mathcal{N}_i} -c_{p1} \\
&= -c_{p1} \sum_{i\in\mathcal{V}} \frac{|\mathcal{N}_i|}{|\mathcal{N}_i|} \\
&= -c_{p1} |\mathcal{V}|.
\end{aligned}
$$

This completes our proof. □

5.2.5 The Action-Attention Structure

The policy for this method is significantly more involved than the simple MLP used for the simple method. The *action attention structure*, as it is called in Zhang et al. (2025), is a neural network which makes use of *attention layers* (Vaswani et al. (2017)). Unfortunately, we do not have space in this report to cover attention, but we give a high-level overview instead, following Zhang et al. (2023).

The attention mechanism is inspired by biological attention, and allows neural networks to direct their attention to specific pieces of data when forming their outputs. This is achieved by mapping the observation tuple, $\boldsymbol{o}_i$ into a high dimensional embedding space. The dimensionality of the embedding space is not given in Zhang et al. (2025), but embedding spaces in robotics commonly have 64 or 128 dimensions (Yan et al. (2024)). In this space, each dimension has some sort of semantic meaning associated with it, which allows the policy to learn complex relationships between observations and ideal actions.

Specifically, we pass $\boldsymbol{o}_{ij}$ through two separate embedding layers, one which produces *query vectors*, $\boldsymbol{q}_j$. These can be thought of as "questions" based off of $\boldsymbol{o}_{ij}$. When we implement the attention mechanism, we will identify *key vectors*, which in this case, are given by the previous action distributions of neighbours, that is, $\boldsymbol{k}_j = A_j$, that match these queries. Stronger matches mean we give more weight to the corresponding *value vectors*, $\boldsymbol{v}_j$, which are also produced by embedding $\boldsymbol{o}_{ij}$ in a different embedding space. These embedding layers are linear layers (i.e., a layer of perceptrons) with ReLU activation functions. During training, the model learns what weights lead to the most suitable embeddings. In this sense, the model is learning what queries it should ask given knowledge of $\boldsymbol{o}_{ij}$, and knowledge of the space of keys. Intuitively, we can imagine the model learning to ask questions like "I know that robot j is close beside me - is it going to move in front of me?". The key, in this case the previous action distribution of neighbours, provides the answer. By using the previous action distribution, A_j as an estimate for what moves robot j is likely to take at the next time step, we can answer the question: e.g., "There is a high probability that the robot will move in front of you". Since this key matches the query, the corresponding value will be given a high weighting. We can imagine that the model would learn to take this as a signal that robot i need to avoid the oncoming frontal collision. The output of the attention layer is passed to an MLP which learns a mapping from the weighted sum of the values, $\boldsymbol{v}_j$ to actions in $\mathcal{A}$.

The key takeaway here is that, similar to Example 4.3, increasing the ability of the robots to consider the actions of other robots should increase their performance. In the following chapter, we will see how this action attention network provides improvements for collision avoidance compared to the simple method.

5.3 Summary

In this chapter, we outlined two methodologies for formalising and solving the flocking problem (Reynolds (1987)). The first of these methods uses an MLP based policy and potential rewards, whereas the second uses a more sophisticated attention based policy, local communication between agents, and constructs a Gibbs distribution over the agents' observations, which is used to form individual rewards and thereby a global reward.

In the next chapter, we implement the first method in simulation, evaluate it, and compare it against the results from Zhang et al. (2025), in which the second method was introduced.

Chapter 6

Simulation and results

In this chapter we describe how we used VMAS (Bettini et al. (2022)) and torchRL (Bou et al. (2023)) which make use of vectorisation to allow for quicker model training using parallel processing and batched environments (Albrecht et al. (2024)). We implement the simple flocking method discussed in the previous chapter and evaluate the flocking performance against the criteria from the previous chapter using metrics from Vásárhelyi et al. (2018) and Zhang et al. (2025).

6.1 Simulation setup

In order to simulate the simple flocking Dec-POMDP, we need a way to simulate the environment and our agents, including computing the rewards and observations each agent receives at each time step. To this end, we use the Vectorized Multi-Agent Simulator (VMAS) environment. We provide some information about the simulator, following Bettini et al. (2022). VMAS is capable of simulating multiple agents in a 2D world, subject to the laws of standard Newtonian mechanics, which is how we modelled our agents in Chapter 5. VMAS is written to run batched environments on graphical processing units (GPUs), which can execute instructions in parallel on each element of a vector of data. As discussed in Subsection 3.2.4, the use of batched environments in this way can greatly speed up the training of policies and value functions.

VMAS provides multiple scenarios out of the box, which can be thought of as pre-programmed Dec-POMDPs. As discussed in Section 5.1, we have modified the included flocking scenario by editing the reward, as well as changing the behaviour of the flocking target and the size of the arena to match the specifications used in Zhang et al. (2025).

The policy, value function, and PPO method are all implemented using the PyTorch and torchRL libraries (Paszke et al. (2019), Bou et al. (2023)), which provide classes and functions for building neural network models, data structures for efficiently storing rewards, actions, observations, etc., and inbuilt optimisation functions, including Adam and PPO.

As in Zhang et al. (2025), robots and obstacles live in a 15m $\times$ 15m square arena, that is $[-7.5, 7.5]^2$. Obstacles are randomly generated within $[-3.5, 3.5] \times [-7.5, 7.5]$ and agents are randomly generated in $[-7.5, -4.5] \times [-7.5, 7.5]$. The flocking target is initialised at $(0, 0)^T$ and accelerates with acceleration $(0.2, 0)^T$ at every time step.

We train the policy and value functions on an Nvidia GTX 1050Ti GPU, we run 50 episodes, where each episode lasts $T = 500$ time steps, as in Zhang et al. (2025). The hyperparameters used for training are $\delta_c = -0.4, \delta_p = 1, \delta_d = 2, d_r = 0.1, \alpha_{\boldsymbol{w}} = \alpha_{\boldsymbol{\theta}} = 5 \times 10^{-4}, \gamma = 0.99$[1].

[1] Code can be found at https://github.com/temperancee/robot_flocking_dissertation.

6.2 Evaluation

We evaluate the performance of our simple method using metrics from Zhang et al. (2025) and Vásárhelyi et al. (2018) that are designed to measure the robots' adherence to the flocking criteria from Reynolds (1987), as defined in Chapter 5.

Firstly, we introduce the *success rate* metric from Zhang et al. (2025) to evaluate the collision avoidance and motion safety of the robots in the flock.

Definition 6.1 (Success Rate Metric)**.** *Denote the number of robots who do not collide with either other robots or obstacles in an episode as n_s. The success rate metric is*

$$\Phi_s = \frac{n_s}{|\mathcal{V}|}.$$

To evaluate the flock's adherence to the velocity matching criterion, we use the *flocking order* metric, which is defined in Vásárhelyi et al. (2018) and also used in Zhang et al. (2025).

Definition 6.2 (Flocking Order Metric)**.** *Denote the velocity of robot i at time step t as $\boldsymbol{v}_{i,t}$, and the neighbourhood of robot i at time step t as $\mathcal{N}_{i,t}$. The flocking order metric is*

$$\Phi_{order} = \frac{1}{T}\sum_{t=1}^{T}\frac{1}{|\mathcal{V}|}\sum_{i\in\mathcal{V}}\frac{1}{|\mathcal{N}_{i,t}|}\sum_{j\in\mathcal{N}_{i,t}}\frac{\boldsymbol{v}_{i,t}\cdot\boldsymbol{v}_{j,t}}{||\boldsymbol{v}_{i,t}||_2||\boldsymbol{v}_{j,t}||_2},$$

The flocking order measures the correlation of velocities between neighbours, averaged over each neighbour, each neighbourhood, and each time step (Soria et al. (2021)).

Recall again from Linear Algebra that for an angle θ between vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^2$, we have $\boldsymbol{a}\cdot\boldsymbol{b} = ||\boldsymbol{a}||_2||\boldsymbol{b}||_2\cos(\theta)$. Thus, when two robots i and j, are matching their velocities at time t, the angle between $\boldsymbol{v}_{i,t}$ and $\boldsymbol{v}_{j,t}$ will be small, maximising $\boldsymbol{v}_{i,t}\cdot\boldsymbol{v}_{j,t}/||\boldsymbol{v}_{i,t}||_2||\boldsymbol{v}_{j,t}||_2$. By averaging over a neighbourhood, we get a measure of the average velocity matching in that neighbourhood. For large flocks, high velocity matching in all neighbourhoods can be treated as a good solution for the flocking problem, even when lacking global velocity matching (Vásárhelyi et al. (2018)). Indeed, consider a large area where the flock has to split into two to navigate around an obstacle, then join back together afterwards. This motivates averaging our metric over all neighbourhoods. Finally, by averaging over time, we get a metric for evaluating the velocity matching over the entire episode.

In order to evaluate the effectiveness of the simple method for both small and large swarms in open and cluttered environments, we apply the method to flocks of 5 and 10 in environments of 10, 20, and 50 obstacles. We also apply the method to a flock of 30 robots in an environment with 50 obstacles. For each agent-obstacle number pair, we simulate 10 trajectories where the agents follow their trained policy network, and compute the average flocking order, Φ_{order}, and the average success rate, Φ_s over these trajectories. These metrics are presented and compared to the method presented in Zhang et al. (2025) (PPO-AA). The flocking order is shown in Table 6.1, and the success rate is shown in Table 6.2. Graphs of the mean total return observed over the training of each pair of policy and value function networks are given in Figure 6.1. Finally, we provide a rendering of 30 simulated agents flocking amongst 50 obstacles in Figure 6.2[2].

[2] A full video of this simulated trajectory can be found here.

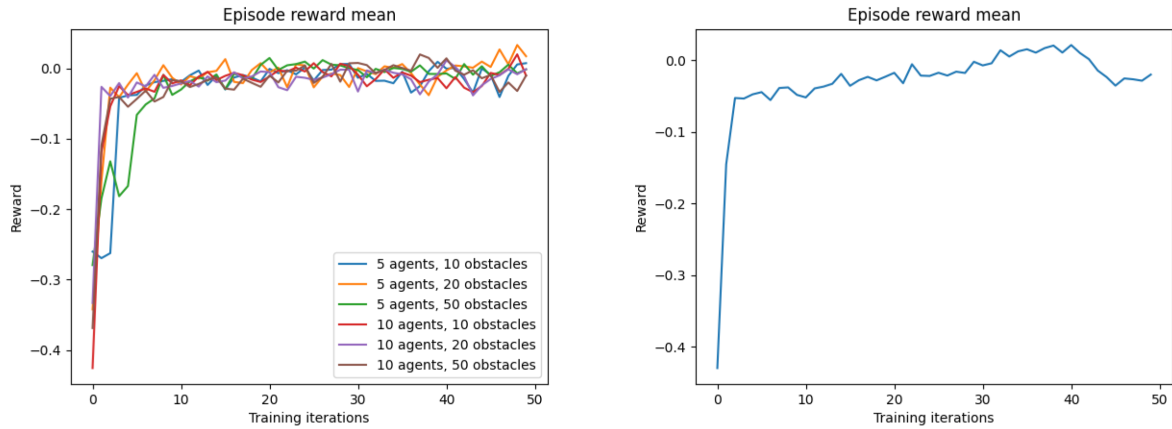

Figure 6.1: Graphs of the mean total return over all training episodes for all training configurations. The graph on the right is for 30 agents and 50 obstacles.

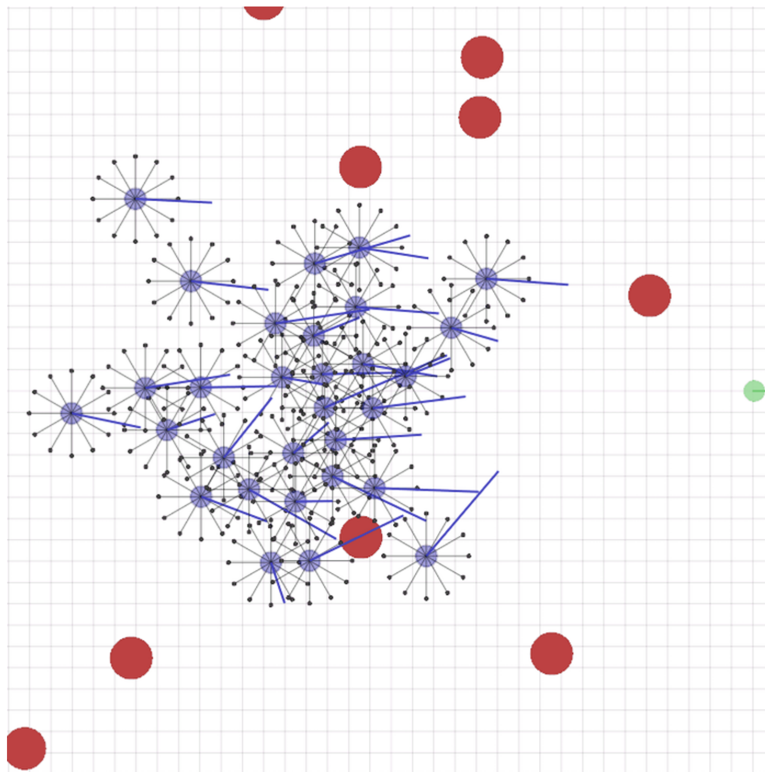

Figure 6.2: A rendering of 30 agents flocking amongst 50 obstacles (most of which are off-screen). Actions are represented by long blue lines protruding from the agents. The black lines with dots on the end show the rays of the LIDARs. The target is in light green on the right, and moves continuously to the right. At the bottom, an agent is moving towards an obstacle, but as shown in the full video rendering (which can be found here), the agent will move around it once it gets close enough for its LIDARs to detect the obstacle.

(agents, obstacles)	(5, 10)	(5, 20)	(5, 50)	(10, 10)
PPO-AA	-	-	-	0.77 ± 0.26
Simple method	0.79 ± 0.05	0.75 ± 0.08	0.76 ± 0.06	0.72 ± 0.06
(agents, obstacles)	(10, 20)	(10, 50)	(30, 50)	
PPO-AA	0.74 ± 0.26	0.54 ± 0.33	0.57 ± 0.26	
Simple method	0.82 ± 0.03	0.73 ± 0.03	0.672 ± 0.013	

Table 6.1: The flocking order under the simple method is evaluated for various numbers of robots and obstacles, and compared with the PPO-AA model from Zhang et al. (2025).

(agents, obstacles)	(5, 10)	(5, 20)	(5, 50)	(10, 10)
PPO-AA	-	-	-	1.00 ± 0.00
Simple method	0.80 ± 0.00	0.80 ± 0.00	0.80 ± 0.00	0.90 ± 0.00
(agents, obstacles)	(10, 20)	(10, 50)	(30, 50)	
PPO-AA	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	
Simple method	0.90 ± 0.00	0.90 ± 0.00	0.97 ± 0.00	

Table 6.2: The success rate under the simple method is evaluated for various numbers of robots and obstacles, and compared with the PPO-AA model from Zhang et al. (2025).

An investigation of the measurements in Table 6.1 and Table 6.2 reveals that while the more sophisticated PPO-AA achieves better collision avoidance, the simple method still manages to achieve respectable results, and often achieves higher flocking order when more obstacles are in the environment. This, however, is likely influenced by how the objects were generated. We speculate that the simulation used in Zhang et al. (2025) had more closely packed obstacles, meaning the robots had to spend more time avoiding obstacles, rather than maintaining the flocking heading. While this is worthy of further investigation, we unfortunately do not carry out any further tests here.

The success rate stays at 0.8 for the 5 robot scenarios and 0.9 for the 10 robot scenarios, indicating that one robot always seems to crash under the simple method, but other than that, the rest of the robots manage to always avoid collisions successfully, as indicated by the 0.00 standard deviations. The flocking order indicates that the robots are matching their velocity directions more often than not, which is good. It is especially remarkable that the simple method manages to succeed with 30 agents and 50 obstacles, despite the cluttered environment.

It is no surprise that PPO-AA performs better than the simple method with regards to the success rate. The action attention policy network allows for agents to predict the movements of their neighbours, which is likely the cause of PPO-AA's higher success rate. As discussed above, it is surprising that the simple flocking method performs so well with regards to the flocking order. We would expect PPO-AA's more sophisticated reward function to result in a better flocking order: the motion smoothness energy specifically should have coerced the flock into matching their velocities, thereby improving the flocking order.

While we would have liked to have presented a simulation of the method presented in Zhang et al. (2025), the incomplete listing of hyperparameters and incomplete description of the action attention network, for example, the sizes of linear layers and the number of attention heads were not included in the paper, made implementing the method a struggle. During my attempts, the robots failed to learn any meaningful behaviour, and simply wandered aimlessly.

Chapter 7

Summary and Conclusion

The methods we have presented for flocking control can be extended in various directions. The use of Gibbs distribution based rewards and a neural network with attention layers as a policy is very similar to the use of graph neural networks (GNNs) and graph attention networks (GATs), which are neural networks (with attention layers, in the later case), that take in graphs as inputs, allowing for a more direct treatment of graphical structures. The flocking problem has been tackled using GATs in Xiao et al. (2023), Yan et al. (2024), and Chen et al. (2024), where the methods achieve good performance: in Chen et al. (2024), the flocking order and success rate metrics are used, and the results are comparable to and in some cases better than the results in Zhang et al. (2025). The use of centralised training decentralised execution (CTDE) in place of decentralised training decentralised execution (DTDE) is another topic worth investigation. Using a centralised value network (i.e., one that has access to all agents' observation data) during training, while maintaining that policies can only use local information could lead to improved policies as discussed in Albrecht et al. (2024, Subsection 9.1.3), since agents would be critiqued with respect to the global configuration.

The aim of this report was to present a clear and well-motivated introduction to both undirected graphical models and reinforcement learning and show how they can be applied to solve various modelling and control problems, specifically in the domain of multi-robotic systems. To this end, we discussed Markov random fields and Gibbs distributions, and showed their equivalence for positive distributions by introducing Theorem 2.1 and the Hammersley-Clifford theorem. We then introduced reinforcement learning and Markov decision processes, giving various examples of MDPs to showcase their generality. For solving MDPs, we introduced policy gradient methods, covering REINFORCE (Algorithm 1) and REINFORCE with baseline (Algorithm 2) (Williams (1992)). Finally we generalised the MDP paradigm by introducing POMDPs and Dec-POMDPs, and discussed issues that arise when moving from the MDP paradigm to these more complex frameworks.

Our coverage of these vast topics was necessarily limited, and there is a plethora of techniques that have been left out of this report. Chief among them are the methods for performing inference on Markov random fields and Gibbs distributions. As discussed at the end of Chapter 2, there is a wide array of methods for performing inference on these models, including Markov chain Monte Carlo (Murphy (2012, chapter 24), Gelman (2013, chapter 11)), variational inference, including the mean field method (Murphy (2012, chapter 21), Koller and Friedman (2009, chapter 11)), and message-passing (Bishop (2006, chapter 8)).

Similarly, the space of solution methods for MDPs contains many more methods than just the policy gradient methods we have covered here. As briefly mentioned in Chapter 3, there are many tabular methods for solving MDPs with finite state and action spaces, including dynamic programming (Bellman (1957)), Monte Carlo methods (Sutton and Barto (2018, Chapter 5)), temporal difference methods (Sutton and Barto (2018, Chapter 6)), and methods from the planning literature (Oliehoek and Amato (2016), Sutton and Barto (2018, Chapter 9)). Some of

these methods, like Q-learning, can be extended into function approximation methods, like deep Q networks (DQN) (Albrecht et al. (2024, Section 8.1)). In the realm of policy gradient methods, further developments of the ideas introduced in REINFORCE with baseline (Algorithm 2) exist. Advantage actor critic (A2C) works similarly to REINFORCE with baseline, except returns are computed over n steps, rather than the whole episode, which can often lead to quicker policy training (Albrecht et al. (2024, Subsection 8.2.5)). Proximal Policy Optimisation (PPO) (Schulman et al. (2017)), which we used for training in Chapter 5 works similarly to A2C, but introduces weights to avoid large gradient updates, which helps to stabilise training.

We hope that the presentation of the material in this report was clear and easy to understand. Thank you for reading.

Bibliography

S. V. Albrecht, F. Christianos, and L. Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024. URL https://www.marl-book.com.

R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

M. Bettini, R. Kortvelesy, J. Blumenkamp, and A. Prorok. VMAS: A vectorized multi-agent simulator for collective robot learning. In *International Symposium on Distributed Autonomous Robotic Systems*, pages 42–56, 2022.

M. Bettini, A. Prorok, and V. Moens. Benchmarl: Benchmarking multi-agent reinforcement learning. *Journal of Machine Learning Research*, 2024.

C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

A. Blake, P. Kohli, and C. Rother. *Markov Random Fields for Vision and Image Processing*. MIT Press, 2011.

A. Bou, M. Bettini, S. Dittert, V. Kumar, S. Sodhani, X. Yang, G. De Fabritiis, and V. Moens. Torchrl: A data-driven decision-making library for pytorch. *arXiv*, 2023. URL https://arxiv.org/abs/2306.00577.

A. G. Bunn, D. L. Urban, and T. H. Keitt. Landscape connectivity: a conservation application of graph theory. *Journal of environmental management*, 2000.

S. Chen, Y. Sun, P. Li, L. Zhou, and C.-T. Lu. Learning decentralized flocking controllers with spatio-temporal graph neural network. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2596–2602, 2024.

A. V. Clemente, H. N. Castejón, and A. Chandra. Efficient parallel methods for deep reinforcement learning. *arXiv*, 2017. URL https://arxiv.org/abs/1705.04862.

P. Clifford. Markov random fields in statistics. *Disorder in physical systems: A volume in honour of John M. Hammersley*, 1990.

C. S. De Witt, T. Gupta, D. Makoviichuk, V. Makoviychuk, P. H. Torr, M. Sun, and S. Whiteson. Is independent learning all you need in the starcraft multi-agent challenge? *arXiv*, 2020. URL https://arxiv.org/abs/2011.09533.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv*, 2020. URL https://arxiv.org/abs/2010.11929.

A. Gelman. *Bayesian Data Analysis*. Chapman and Hall/CRC, 2013.

D. Huh and P. Mohapatra. Decentralized multi-agent filtering. *arXiv*, 2023. URL https://arxiv.org/abs/2301.08864.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015. URL http://arxiv.org/abs/1412.6980.

D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

D. G. Krivochen. *Syntax on the edge: A graph-theoretic analysis of sentence structure*. Brill, 2023.

Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 2015.

D. J. MacKay. *Information theory, inference and learning algorithms*. Cambridge University Press, 2003.

T. Mikolov, W. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

T. Minka. Divergence measures and message passing. *Technical report, Microsoft Research*, 2005.

P. M. Morse. Diatomic molecules according to the wave mechanics. II. Vibrational levels. *Physical review*, 1929.

H. Müller and D. Kudenko. Improving the effectiveness of potential-based reward shaping in reinforcement learning. *arXiv*, 2025. URL https://arxiv.org/abs/2502.01307.

K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

J. R. Norris. *Markov Chains*. Cambridge University Press, 1997.

F. Oliehoek and C. Amato. *A Concise Introduction to Decentralised POMDPs*. Springer, 2016.

G. Papoudakis, F. Christianos, A. Rahman, and S. V. Albrecht. Dealing with non-stationarity in multi-agent deep reinforcement learning. *arXiv*, 2019. URL https://arxiv.org/abs/1906.04737.

J. Park, D. Kim, G. C. Kim, D. Oh, and H. J. Kim. Online distributed trajectory planning for quadrotor swarm with feasibility guarantee using linear safe corridor. *IEEE Robotics and Automation Letters*, 2022.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 2019.

E. Pignatelli, J. Ferret, M. Geist, T. Mesnard, H. van Hasselt, O. Pietquin, and L. Toni. A survey of temporal credit assignment in deep reinforcement learning. *arXiv*, 2023. URL https://arxiv.org/abs/2312.01072.

D. Pollard. Hammersley-Clifford theorem for Markov random fields. Yale University, Statistics 251 Lecture Notes, 2004. URL http://www.stat.yale.edu/~pollard/Courses/251.spring04/Handouts/Hammersley-Clifford.pdf.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 25–34, 1987.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 4th edition, 2010.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv*, 2017. URL https://arxiv.org/abs/1707.06347.

S. Seuken and S. Zilberstein. Improved memory-bounded dynamic programming for decentralized pomdps. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, page 344–351, 2007.

D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning*, pages 387–395, 2014.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016.

S. P. Singh and R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine learning*, 1996.

E. Soria, F. Schiano, and D. Floreano. Distributed predictive drone swarms in cluttered environments. *IEEE Robotics and Automation Letters*, 2021.

R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2018.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 1999.

S. Thrun, W. Burgard, and D. Fox. *Probabilistic Robotics*. MIT Press, 2005.

G. Vásárhelyi, C. Virágh, G. Somorjai, T. Nepusz, A. E. Eiben, and T. Vicsek. Optimized flocking of autonomous drones in confined environments. *Science Robotics*, 2018.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

L. Weng. Policy gradient algorithms. *lilianweng.github.io*, 2018. URL https://lilianweng.github.io/posts/2018-04-08-policy-gradient/.

R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 1992.

A. Wong, T. Bäck, A. V. Kononova, and A. Plaat. Deep multiagent reinforcement learning: Challenges and directions. *Artificial Intelligence Review*, 2023.

J. Xiao, G. Yuan, J. He, K. Fang, and Z. Wang. Graph attention mechanism based reinforcement learning for multi-agent flocking control in communication-restricted environment. *Information Sciences*, 2023.

C. Xu, J. T. Springenberg, M. Equi, A. Amin, A. Esmail, S. Levine, and L. Ke. RL token: Bootstrapping online RL with vision-language-action models, 2026. URL https://www.pi.website/research/rlt.

C. Yan, C. Wang, H. Zhou, X. Xiang, X. Wang, and L. Shen. Multi-agent reinforcement learning with spatial–temporal attention for flocking with collision avoidance of a scalable fixed-wing UAV fleet. *IEEE Transactions on Intelligent Transportation Systems*, 2024.

A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. URL https://D2L.ai.

D. Zhang, C. Yu, F. Xue, and Q. Zhang. Learning efficient flocking control based on Gibbs random fields. *IEEE*, 2025.

C. Zhu, M. Dastani, and S. Wang. A survey of multi-agent deep reinforcement learning with communication. *Autonomous Agents and Multi-Agent Systems*, 2024.

B. Zitkovich, T. Yu, S. Xu, P. Xu, T. Xiao, F. Xia, J. Wu, P. Wohlhart, S. Welker, A. Wahid, et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. In *Conference on Robot Learning*, pages 2165–2183, 2023.