Developing Artificial Intelligence Agents to Manipulate Quantum Entanglement

This thesis is submitted in partial fulfillment of the requirements

for the degree of

M.Sc. in Artificial Intelligence

by

Pavel Tashev

(student ID: 26060)

Department of Computer Informatics
Faculty of Mathematics and Informatics
Sofia University

October 22, 2022

Approved by

Thesis Supervisor:	Marin Bukov, PhD
Co-advisor:	Prof. Dr. Maria Nisheva
defense date:	31.10.2022 – 04.11.2022

Certificate of Originality

*This is to certify that this thesis, entitled "**Developing Artificial Intelligence Agents to Manipulate Quantum Entanglement**", submitted in partial fulfillment of the requirements for the degree M.Sc. in Artificial Intelligence at Sofia University, is my own work and contains original results obtained by myself with the support and assistance of my supervisor.*

In particular, I certify that results and ideas obtained by third parties that are described, published, or authored elsewhere, are to a sufficient extent referred to, properly cited, and acknowledged in the thesis, in accordance with the requirements for respecting intellectual property, and the academic ethics and standards.

I am aware that, should plagiarism or scientific misconduct be established, the Thesis Committee reserves the right to reject this thesis, while the Committee for Academic Ethics at the Ministry of Science and Education has the legal right to retroactively revoke the degree conferred.

I declare that the results contained in the present thesis have not been submitted at any other university, institute, or higher-education institution in (partial) fulfillment of the requiements for an academic degree.

Sofia
October 22, 2022

Pavel Tashev
student ID: 26060
Faculty of Mathematics
and Informatics
Sofia University

Abstract

In this work we study the problem of manipulating quantum entanglement of multi-qubit quantum systems using deep reinforcement learning. We begin by introducing the field of RL (Reinforcement learning) in Chap. 2. We give an overview of the most important concepts and definitions establishing the terminology of RL. We formalise the notions of agent and environment, and we provide a full derivation of an algorithm for learning using trial-and-error – the PG (Policy gradient) algorithm. Different modifications allowing for variance reduction are derived and discussed. Addressing the exploration-exploitation dilemma we use a regularization method known as *entropy regularization*. A thorough investigation regarding the usage of this method in Monte Carlo policy gradient algorithms is given in App. B. Finally, an objective function unifying both variance reduction and entropy regularization is derived. In addition to RL (Reinforcement learning) we also discuss IL (Imitation learning) methods, focusing on the use of the algorithm BC (Behavioural cloning). Supervised learning algorithms, such as BC, learn from a training set of labelled examples. We discuss the need for labelled data sets and we propose to use a classical search algorithm *Beam search* for generating training examples.

In Chap. 3 we introduce the field of quantum mechanics and discuss the differences between classical an quantum computers. We define the notion of a *qubit* and we describe how quantum operations are performed using quantum gates. We also show how multi-qubit quantum systems are simulated on a classical computer and discuss the issue of exponential growth. One of the most important properties of quantum systems – quantum entanglement, is introduced and a procedure calculating the degree of entanglement is given.

In Chap. 4 the problem of manipulating quantum entanglement is introduced. We give a formal definition of a 'disentanglement' and we show a formula for measuring how close a given quantum state is to a disentangled state. In this chapter we also study the issue of finding a locally optimal gate, that would apply a maximum reduction to the entanglement of a quantum state. We show how quantum states with a small number of qubits ($n = 2$ and $n = 3$) are disentangled, and finally we give a formal definition of the problem studied in this work.

A model for the given problem using the framework of RL (Reinforcement learning) is described in Chap. 5. The task is posed as an optimization problem and the different components in a Reinforcement learning setting – the agent and the environment, are described in detail. In addition, a procedure for running simulations in parallel is described.

In Chap. 6 we present the results that we obtained from training different artificial intelligence agents to disentangle quantum states. We give a detailed explanation of the experimental set-up and we follow with a discussion of the results. An approach for com-

bining supervised learning and reinforcement learning is proposed and the results from its application are discussed.

Finally, in Chap. 7, we finish with a discussion about the contribution of this work to the fields of QC (Quantum computing) and RL (Reinforcement learning), and we make recommendations for future work on the topic.

The code for running the experiments described in this work can be found at:
https://github.com/cacao-macao/entanglement-control.

Разработване на интелигентни агенти за манипулиране на квантово вплитане

Дипломна работа представена за частично покриване на изискванията за образователно-квалификационна степен

магистър "Изкуствен Интелект"

от

Павел Златков Ташев

(факултетен номер: 26060)

Катедра Компютърна Информатика
Факултет по Математика и Информатика
Софийски университет „Св. Климент Охридски“

22 октомври 2022 г.

Одобрена от

ръководител: д-р Марин Буков

консултант: проф. д-р. Мария Нишева

дата на защитата: 31.10.2022 – 04.11.2022

Декларация за оригиналност и автентичност

Аз, ***Павел Златков Ташев****, студент от Факултет по Математика и Информатика на Софийски университет „Св. Климент Охридски“, декларирам, че представената от мен за защита дипломна работа на тема:* ***„Разработване на интелигентни агенти за манипулиране на квантово вплитане”****, за присъждане на образователно-квалификационна степен магистър "Изкуствен Интелект" е оригинална разработка и съдържа оригинални резултати, получени при проведени от мен научни изследвания (с подкрепата и/или съдействието на научния ми ръководител).*

Декларирам, че резултатите, които са получени, описани и/или публикувани от други учени, са надлежно и подробно цитирани, при спазване на изискванията за защита на авторското право и на академичната етика и стандарти.

Уведомен/а съм, че в случай на констатиране на плагиатство или недостоверност на представените научни данни, Комисията по защитата е в правото си да я отхвърли, а Комисията по академична етика към Министерство на образованието и науката е в законното си правото да анулира придобитата образователно-квалификационна степен.

Декларирам, че настоящият труд не е представян пред други университети, институти и други висши училища за придобиване на образователна и/или научна степен.

София
22 октомври 2022 г.

Павел Златков Ташев
факултетен номер: 26060
Факултет по Математика
и Информатика
Софийски университет
„Св. Климент Охридски“

Абстракт

В тази работа изучаваме проблема за манипулиране на квантовото вплитане на многокюбитови квантови системи с помощта на дълбоко обучение с утвърждение. Започваме с въвеждането на областта на RL (Reinforcement learning) в глава 2. Правим преглед на най-важните понятия и дефиниции, и установяваме терминологията на RL. Формализираме понятията агент и среда и предоставяме извеждане на алгоритъм за обучение чрез метода "проба-грешка алгоритъмът PG (Policy gradient). Изведени са и са обсъдени различни модификации, позволяващи намаляване на дисперсията. За решаване на проблема "изследване-експлоатация"използваме метод за регуларизация, известен като *регуларизация на ентропията*. Задълбочено изследване относно използването на този метод в Монте Карло градиентните алгоритми на полицата е дадено в Прил. B. Накрая е изведена целева функция, обединяваща както намаляването на дисперсията, така и регулирането на ентропията. В допълнение към RL (Reinforcement learning), обсъждаме и методи от типа IL (Imitation learning), като се фокусираме върху използването на алгоритъма BC (Behavioural cloning). Алгоритмите за обучение с учител, като например BC, се обучават от набор от анотирани примери. Обсъждаме необходимостта от анотирани данни и предлагаме да се използва класически алгоритъм за търсене *Търсене в лъч* за генериране на примери за обучение.

В глава 3 представяме областта на квантовата механика и обсъждаме разликите между класическите и квантовите компютри. Даваме определение на понятието *qubit* и описваме как се извършват квантови операции с помощта на квантови опратори. Показваме също как многокубитови квантови системи се симулират на класически компютър и обсъждаме въпроса за експоненциалния растеж. Въвежда се едно от най-важните свойства на квантовите системи - квантовото вплитане, и се дава процедура за изчисляване на степента на вплитане.

В глава 4 е представен проблемът за манипулиране на квантовото вплитане. Дава се формално определение на ‘разплитане’ и се показва формула за измерване на това колко близо е дадено квантово състояние до разплетено състояние. В тази глава се разглежда и въпросът за намиране на локално оптимален оператор, който да осигурява максимално намаляване на вплитането на квантовото състояние. Показваме как се разплитат квантови състояния с малък брой кюбити ($n = 2$ и $n = 3$) и накрая даваме формално определение на проблема, изследван в тази работа.

Модел за дадения проблем, използващ рамката на RL (Reinforcement learning), е описан в глава 5. Задачата е поставена като оптимизационен проблем и различните компоненти в условията на Reinforcement learning - агентът и средата, са описани подробно. Освен това е описана процедура за паралелно провеждане на симулации.

В глава 6 представяме резултатите, които сме получили от обучението на различни

агенти с изкуствен интелект за разплитане на квантови състояния. Даваме подробно обяснение на експерименталната постановка, а след това обсъждане на резултатите. Предложен е подход за комбиниране на обучение с учител и обучение с утвърждение и са обсъдени резултатите от неговото прилагане.

Накрая, в глава 7, завършваме с обсъждане на приноса на тази работа към областите на QC (Quantum computing) и RL (Reinforcement learning) и даваме препоръки за бъдеща работа по темата.

Кодът, с който могат да бъдат повторени експериментите от тезата, може да бъде намерен на следния адрес:
https://github.com/cacao-macao/entanglement-control.

Acknowledgments

I would like to express my gratitude to my supervisor Dr. Marin Bukov, without whom this work would have not been possible. I have benefited greatly from his wealth of knowledge in the fields of both Quantum Mechanics and Reinforcement Learning. And I also had the pleasure to be introduced to the world of research. His invaluable advises, novel ides, helpful guidance and constructive critiques helped this work and are greatly appreciated.

I would also like to thank Prof. Maria Nisheva for the overall support during the writing of this thesis.

Very special thanks to my teammate Stefan Petrov, who was an indispensable part of this project. His dedication and commitment to this cause have helped me greatly in writing this thesis.

All of the experiments provided in this work would not have been possible without the support of the BG05M2OP001-1.001-0004 UNITE project ('Universities for Science, Informatics and Technology in the eSociety'). Thank you for the provided computational resources!

Contents

List of Figures

List of Tables

List of Codes

List of Acronyms

AI Artificial intelligence 4, 5

BC Behavioural cloning iii, 18, 19, 75

DPL Direct policy learning 19

FCNN Fully-connected neural network 77

i.i.d. Independent and identically distributed 18, 19

IDA* Iterative-deepening A^* search 21

IL Imitation learning iii, 6, 17, 18, 68, 77

MDP Markov decision process 9, 53, 67, 78

MDPs Markov decision processes 4

NLP Natural language processing 71

NN Neural network 77

PG Policy gradient iii, 6, 13, 16, 68–70, 77

QC Quantum computing iv

RL Reinforcement learning iii, iv, 4–9, 13, 17, 53, 54, 57, 67, 68, 75, 76

RNN Recurrent neural network 6, 67

Chapter 1

Introduction

1.1 The quantum spin

In the beginning of the last century two physicists conceived and performed an experiment that questioned the concepts of classical mechanics and revolutionized our understandings of microscopic phenomena. The experiment, conceived by Otto Stern in 1921, and carried out by him in collaboration with Walther Gerlach in 1922, provided the first incontrovertible evidence that individual atoms have a **quantized** magnetic moment.

The experiment (see. Fig 1.1) consisted of firing a narrow beam of silver atoms through a vertical magnetic field. Since the atom as a whole carries angular momentum, when passing through a magnetic filed it is deflected by a corresponding amount. After leaving the magnetic field the atoms were projected on a detector screen illustrating the size of the deflection. Given that the atoms are randomly oriented we would expect to observe the atoms being deflected along a continuous vertical line. However, unlike classical magnets, the atoms were all deflected either upward or downward by the same amount, and were thus projected in two discrete points on the screen.

Building on top of the findings of Planck, Einstein, Bohr, de Broglie and many others, it had become clear to most physicists that classical mechanics could not fully describe the world of atoms. The correct interpretation of this result was far more subtle and a new theory needed to be developed. Thus, in the beginning of 1925 modern quantum mechanics was founded. The atom's magnetism is produced by a property of the electron called *spin*, which is a property of the quantum world an has no classical analog. Today the concept of *quantum spin* provides an analog to the classical bit and is central for the foundation of quantum computing.

1.2 Superposition and entanglement

Uncertainty is one of the central notions in quantum mechanics. Even if we know everything about the internal representation of a given quantum particle, we still cannot predict with certainty the outcome of a given experiment, e.g. to measure the direction of the spin. Instead, we can calculate a statistical distribution of the possible results. This gives rise to an important question disturbing both physicists and philosophers. What was the state of the particle, *spin-up* or *spin-down*, just before the measurement was

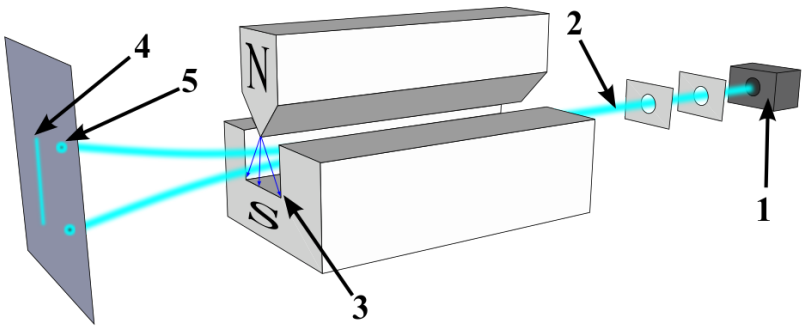


Figure 1.1: The Stern-Gerlach experiment. Silver atoms travelling through an inhomogeneous magnetic field, and being deflected up or down depending on their spin; (1) furnace, (2) beam of silver atoms, (3) inhomogeneous magnetic field, (4) classically expected result, (5) observed result. Figure adapted from [1].

made? One proposition is that the particle was in a well-defined specific state, however the theory of quantum mechanics could not determine the state with certainty. Some additional information, known as *hidden variables*, needed to be provided for a complete description of physical reality. Einstein and other advocates of this proposition claimed that quantum mechanics is an incomplete theory. The other answer, which is the most widely accepted one, known as the *Copenhagen interpretation*, states that the particle is in both states simultaneously, i.e., it is in a *superposition*. It is the act of measurement that *collapsed* the system into the state that is observed, and any other measurement immediately after the first would yield the same result.

Another peculiar property of quantum mechanics is *entanglement*. When two particles are entangled, one particle cannot be fully described without considering the other, because they are defined by a single wave function; that is to say, they are not individual particles but are an inseparable whole. Moreover, measuring one of the particles would collapse the entire state. This means that we do not have to measure the other particle, its state will already be known to us.

Using the property of entanglement, Albert Einstein, Boris Podolsky and Nathan Rosen published a paper in 1935 [2] arguing that quantum mechanics was not a complete theory, but had to be supplemented with additional variables addressing *locality* and *causality*. The paper introduced a thought experiment imagining two particles that were entangled into a single quantum state. Once separated, the two particles are still described by the same wave function, and measuring one of the particles will instantaneously determine the state of the second particle, no matter what the distance between them. The experiment is cited as a paradox, because it appears to violate one of the central tenets of relativity: transmitting information faster than the speed of light violates the principle of causality. Einstein dubbed it *'spooky action at a distance'* and postulated the existence of hidden

variables as yet unknown local properties of the system which should account for the discrepancy.

Einstein, Podolsky and Rosen did not imply that quantum mechanics was incorrect, they only claimed that it is an incomplete description and there is more to the whole story. However, in 1964 a paper was published by John Bell [3] proving that *any* local hidden variable theory was incompatible with quantum mechanics. Thus, if the EPR paradox was correct, then quantum mechanics is not only incomplete, but in fact wrong. On the other hand, if quantum mechanics is correct, then there can be *no* hidden variable theory. Many experiments were performed to test Bell's theory and all were in excellent agreement with the predictions of quantum mechanics, thus, demonstrating that nature is fundamentally non-deterministic. Probably the most important experimental evidence were reported by Aspect [4], who was awarded the Nobel prize in Physics 2022 for pioneering quantum information science.

1.3 Controlling entanglement and current research

Today, building quantum computers is quickly shaping to be the most important undertaking of the 21st century. Instead of using electricity to represent a binary logical states, or bits (0 and 1), quantum computing uses the quantum properties of particles to represent two-level quantum mechanical systems, known as *qubits*. Operations on qubits are performed using the quantum analog of logic gates – *quantum gates*.

Being fundamental to quantum computing, the property of entanglement and the processes of creating and destroying it are of paramount importance to be studied. Pursuing the creation of quantum entanglement, Kraus and Cirac showed in their paper [5] which are the optimal quantum operations for entangling a quantum system. When applied these operations will transition the system into a state of maximum entanglement. Researching the reverse process of entanglement, Terno [6] and later Tor [7] studied the problem of disentangling states, that is transforming a state of two subsystems into a separable product state. Their findings show that a universal disentangling machine is impossible and only a state-dependent machine can be designed.

In light of these findings in this work we try to design a state-dependent disentnagling machine that is optimal in the sense of the number operations that it produces. Inspired by Kraus and Cirac, we search for a quantum operation that would transform a system into a state of minimum entanglement. Then, in order for a given quantum state to be disentangled, multiple of these operations need to be applied. However, choosing the optimal sequence, that would produce the least number of operations, is a hard combinatorial task. Solving that task requires exploring a space exponential with respect to the number of qubits in the system.

1.4 History of artificial intelligence

For decades researchers have been trying to design intelligent systems trying to mimic human intelligence. The planning skills of a human simply cannot be encoded in an algorithm. Problems such as playing chess and Go, or even driving a car, have been long standing challenges to computers due to the sheer number of available actions that can be

taken. Even the most powerful super-computers cannot process all possible combinations of actions in order to select the correct path that should be taken. Choosing an action and planning for the impact it would have many steps down the road, was a problem too hard for declarative algorithms and a new paradigm in programming was founded.

In the summer of 1956 John McCarthy, Marvin Minsky, Claude Shannon and Nathaniel Rochester organised a two month workshop at Dartmouth bringing together researchers interested in the study of intelligence. Despite not leading to any breakthroughs, this workshop introduced the idea of AI (Artificial intelligence) to the scientific community effectively creating a new research area in the field of computer science. At first (1969–1986) researchers were focusing on developing *expert systems*, programs encoding human knowledge capable of solving real-world problems using an algorithm known as *resolution*. Until the mid 80s the emphasis, was less on learning and more on representing and encoding human knowledge, and reasoning using mathematical logic. However, building and maintaining expert systems for complex domains was a difficult task. Systems could not learn from experience and expert knowledge had to be continuously encoded. Moreover, in the face of uncertainty expert systems simply broke down. Around 1987 this led to the beginning of a new era in AI. Researchers turned to probability rather than using Boolean logic, and machine learning instead of hand-coding explicit knowledge. Algorithms such as *Hidden Markov models* [8] and *Bayesian networks* [9] were developed using probabilistic reasoning and based on rigorous mathematical theory. The most important aspect of these algorithms was that they allowed to be trained on a large corpus of data, thus steadily improving their performance.

The advancements in computing power and the dramatic increase of available storage space have facilitated the creation of very large data sets. This led to the development of learning algorithms specifically designed to take advantage of the presence of such data sets. These algorithms, known as neural networks, were conceived back in the early stages of AI, but it was not until enough data was available before their true capabilities were discovered. Starting from 2011 a third revolution in the field of AI has been underway. Neural network models of evermore complex architectures and exponentially growing number of parameters have been introduced, and tasks that were thought to be far beyond the reach of machines have been solved.

1.5 Reinrofcement learning for optimizing combinatorial tasks

The modern field of RL (Reinforcement learning) was conceived as a sub-field of AI in the late 1980s. A major contribution was Richard Sutton's work connecting three major threads into a unified field of research. The first thread, originating in the psychology of animal learning, was concerned with learning by trial and error. The second thread was the mathematical field of operations research and more importantly the theory of MDPs (Markov decision processes). The third thread was the study of temporal-difference learning methods, exemplified by Arthur Samuel's famous checkers playing program [10] designed in 1952. Further, in 1989 the Q-learning algorithm, developed by Chris Watkins [11], fully brought together the threads of optimal control and temporal-difference.

One of the most impressive early applications of RL was Gerald Tesauro's program, *TD-Gammon* [12] (1994). The program was a successor of Samuel's checkers player and learned to play the game of backgammon reaching the level of the world's strongest

grandmasters. The algorithm required little more knowledge than the rules of the game, and using learning from self-play managed to achieve professional level of play.

Another famous system using RL is *IBM Watson* [13]. Developed by a team of IBM to play the popular TV game *Jeopardy!*, Watson won first prize in an exhibition match against human champions. Although the the system was mainly a demonstration of the advancements in the field of natural language question-answering, a sophisticated decision-making strategy was implemented for the critical parts of the game. Except answering questions, the rules of the game dictated that contestants sometimes had to decide on how much money they want to bet that they will answer correctly the next question. Whenever Watson hat to place a bet, it chose its bet by performing an elaborate calculation maximizing its probability to win the game from the current state. In these situations humans simply cannot match the equity and confidence estimates performed by Watson. The effectiveness of this wagering strategy was well beyond the capabilities of human players and was an important contributor to Watson's impressive performance.

Probably the pinnacle of RL is the introduction of the algorithm *AlphaGo* [14, 15]. The program solved a grandstanding challenge in AI mastering the ancient game of Go and defeating the 18-time world champion Lee Sedol, winning 4 out of 5 games – a feat that was considered by many to be decades into the future. The search space for Go is significantly larger compared to the previous examples. There are about 250 legal actions per move and games tend to involve close to 150 moves. These numbers make the search space larger than the observable universe, however this is not the main factor that makes Go difficult. The major problem is that it is extremely hard to define a good position evaluation function, that would adequately predict the outcome of the game given the current position. The team at DeepMind made an incredible effort combining deep neural networks, supervised learning, Monte Calro tree search, and reinforcement learning, and producing one of the most sophisticated AI algorithms. The remarkable success of *AlphaGo* has been a driving force in the field of AI and has shined a new light on Reinforcement learning.

As we can see, solving combinatorial problems with enormous (or even infinite) state spaces is in fact a well studied problem. Thus, in this work we try to apply different algorithms from the field of Artificial intelligence to solve the intractable problem of controlling quantum entanglement.

Chapter 2

Machine learning based optimisation

In this chapter we discuss different learning algorithms for solving optimization problems. In Sec. 2.1.1 we give a short description of RL (Reinforcement learning) and we introduce the most important concepts and definitions needed to understand this work. In Sec. 2.1.3 we define what a policy is and how it can be used to make decisions. PG (Policy gradient), the algorithm used for learning in this work, is introduced in Sec. 2.1.4, and different variants for improvement are given in Sec. 2.1.5, 2.1.6. For a thorough review of the matter the reader is referred to [16, 17]. In Sec. 2.2 we describe a supervised algorithm for training agents known as IL (Imitation learning) and in Sec. 2.3 we introduce two classical search algorithms for solving combinatorial search problems.

2.1 Reinforcement learning

The field of machine learning can broadly be divided into *supervised learning*, *unsupervised learning*, and *reinforcement learning*. The type of learning heavily depends on the type of data we have. Supervised learning is learning from a training set of labeled examples. Unsupervised learning is typically about finding patterns hidden in collections of unlabelled data. Reinforcement learning is learning from data generated by an agent acting in an environment.

In this work we focus on using RL (Reinforcement learning) and the reason for this is twofold. On the one hand learning from data requires having large datasets of labeled examples, which in our case is something that we do not have. To our knowledge the problem studied in this work has always been attacked from the theoretical side, and this is (to the best of our knowledge) the first attempt at designing a learning algorithm to produce a solution.

On the other hand the problem at hand is sequential, but it is also inherently Markovian. This means that the usual tool for sequential problems – a RNN (Recurrent neural network), is not suitable in our case since it is known to be unable to handle Markovian processes [18]. Thus, we will try to develop an RL algorithm for solving the problem.

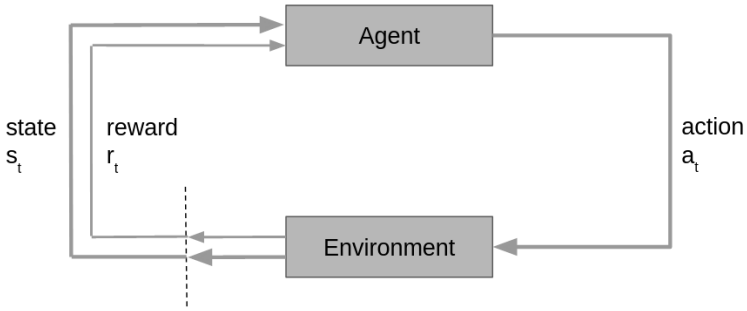


Figure 2.1: Interaction loop between agent and environment. The figure shows the interaction feedback loop between the agent and the environment. At every step the agent observes the state of the environment s_t and takes an action a_t. The environment transitions into a new state s_{t+1} and emits a reward signal r_{t+1}.

2.1.1 Agents and environments

Reinforcement learning is the study of agents and how they learn by trial and error. The key components of RL are the **agent** and the **environment**. The environment is the representation of the world that the agent lives in and interacts with. It might be a simulation or it might be a real physical environment. The interaction of the agent with the environment is described with a (possibly infinite) feedback loop. At every step of the loop, the agent observes the **state** of the environment, and based on that observation decides on an **action** to take. Every step of the loop is called a *time step*, and one run of the loop from start to finish is called an *episode*.

When the environment is acted upon it changes its state, and also emits a reward signal indicating how good or bad the new state is. Usually, it is assumed that the new state of the environment depends only on the last observed state and the action taken by the agent. This is known as a *Markov property* and is discussed in Sec. 2.1.2.

The goal of the agent is to maximise its total sum of rewards, called **return**, accumulated throughout the interaction loop. The agent is not told which actions to take, and instead discovers which actions are most promising by trying them out. A single action determines the immediate reward, but it also determines the next state, and through that it affects all subsequent rewards. As stated by Sutton & Barto in Ref. [16], these two characteristics – trial-and-error search, and delayed reward – are the two most important distinguishing features of reinforcement learning.

In order to describe the algorithms developed and used in this thesis, we first need to establish the terminology used in reinforcement learning. The problem of reinforcement learning is formalised using ideas from dynamical systems theory [19].

2.1.2 States, actions and rewards

We will start by formalising the notion of a state. As mentioned earlier, during the agent-environment feedback loop, the agent observes the state of the environment, and decides on its actions based on that state. All the knowledge needed to make inference is compressed in the state.

Formally, a **state** s is a complete description of the current representation of the environment. A state is represented by a real-valued (or a complex-valued, see Sec. 3.2) vector, matrix, or higher-order tensor. The current state s at time step t is denoted by s_t. For example, in a mechanical simulation the state could consist of the positions, velocities, and accelerations of an object along the x, y, z axes:

$$s_t = \begin{bmatrix} x_t, & y_t, & z_t, & v_{x,t}, & v_{y,t}, & v_{z,t}, & a_{x,t}, & a_{y,t}, & a_{z,t} \end{bmatrix}.$$

The **state space** $\mathcal{S}$ is the set of all valid states of the environment

$$s_t \in \mathcal{S}.$$

The actions that the agent can choose to take depend entirely on the problem we are solving, and the capabilities of our agent. However, in some states of the environment there may be a restriction on a given action.

An **action space** $\mathcal{A}(s)$ is the set of all valid actions in a given state of the environment. The action a selected by the agent at time step t while in state s_t is a_t, where

$$a_t \in \mathcal{A}(s_t).$$

The **reward** r is a signal passed from the environment to the agent. At each time step t, the reward is a scalar that depends on the current state of the environment, the action just taken, and the next state of the environment. More formally, the reward signal can be expressed as the output of a reward function:

$$\begin{aligned} &\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}, \\ &r_{t+1} = \mathcal{R}(s_t, a_t, s_{t+1}). \end{aligned}$$

The reward signal is used to formalise the idea of a goal. RL is based on the **reward hypothesis**, stating that all goals can be described by the maximisation of an expected cumulative reward.

The sum of rewards over all time steps of an episode is called the **return**. A formal description of the return is needed, because we are going to formulate the RL problem as an optimisation problem, where the objective we are trying to maximise is the return:

$$\mathcal{R}(\tau) = \sum_{t=0}^{T} r_{t+1}. \tag{2.1}$$

Here τ represents the trajectory (see Eq. (2.7)) of the agent during a full episode, and T is the total number of time steps in that episode (T may be ∞). Note that we are using an overloaded notation for the reward function $\mathcal{R}$. However, this is the standard notation used in most references, and it is also used in this work.

In the case of infinite agent-environment feedback loops, the return may grow indefinitely, and thus finding an optimisation procedure might be difficult or even impossible. To solve the problem of dealing with infinite quantities, we may add a **discount factor** γ to the reward obtained at every time step. The **discounted return** is the sum of all rewards obtained by the agent, but discounted by how far off in the future they were obtained:

$$\mathcal{R}(\tau) = \sum_{t=0}^{T} \gamma^t r_{t+1}. \tag{2.2}$$

Obviously, setting $\gamma = 1$, gives us back the undiscounted return.

A more general quantity is the sum of rewards received after time step t, denoted G_t:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots = \sum_{k=0}^{T} r_{t+k+1}, \tag{2.3}$$

or in case of discounted rewards:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k r_{t+k+1}. \tag{2.4}$$

Using Eqs. (2.2), (2.3) and (2.4), and setting $t = 0$ it can be seen that the return is equal to G_0, where

$$G_0 = \mathcal{R}(\tau) = \sum_{t=0}^{T} \gamma^t r_{t+1}. \tag{2.5}$$

Returns at successive time steps are related to each other recursively:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \\ &= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \cdots) \\ &= R_{t+1} + \gamma G_{t+1}. \end{aligned} \tag{2.6}$$

As stated earlier, when the environment is acted upon, its state changes and the new state depends only on the previous state and the action taken by the agent. Thus, the environment in an RL problem is assumed to obey the *Markov property*. This assumption allows us to rigorously formulate the reinforcement learning problem as a MDP (Markov decision process).

An MDP is a 5-tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, p_0\}$, where:

- $\mathcal{S}$ is the state space;
- $\mathcal{A}$ is the action space;
- $\mathcal{R}$ is the reward function;
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function;
- p_0 is the initial state distribution

The transition probability function defines the *dynamics* of the MDP. Assuming we are in state s_t and we take an action a_t, then $\mathcal{P}(s_t, a_t, s_{t+1})$ (or $\mathcal{P}(s_{t+1}|s_t, a_t)$) is the probability of transitioning into state s_{t+1}. The term MDP is referring to the fact that the transition function of the environment obeys the *Markov property*: transitions depend only on the current state and the selected action. No prior history is relevant.

Running the agent-environment feedback loop from beginning to end will produce one episode. At each time step t of that episode the agent observes the current state of the environment s_t and based on that state it decides on an action $a_t \in \mathcal{A}(s_t)$. As a consequence of taking that action the state of the environment changes to the new state $s_{t+1} \sim \mathcal{P}(\cdot|s_t, a_t)$, and the environment emits a reward $r_{t+1} = \mathcal{R}(s_t, a_t, s_{t+1})$, which is received by the agent.

This episode gives rise to a sequence of *(state, action, reward)* triples, and this sequence is called a **trajectory**:

$$\tau = (s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2..., s_{T-1}, a_{T-1}, r_T, s_T), \tag{2.7}$$

where s_0 is sampled from the distribution p_0 and T is the total number of time steps.

2.1.3 Policy and value functions

During each step of the agent-environment interaction loop, the agent must choose an action based on the current state on the environment. The rule that the agent uses when deciding on which action to take is called a **policy**. Assuming the action space is discrete (continuous action spaces are not considered in this thesis), a policy π is formally described as a function that maps each state to a vector of probabilities. The probabilities produced by the policy function rank each possible action by its likelihood to be selected:

$$\pi : \mathcal{S} \to [0, 1]^d,$$

where d is the number of actions in the action space $d = |\mathcal{A}(s_t)|$.

The quantity $\pi(s_t)$ yields a probability distribution over the action space $\mathcal{A}(s_t)$. The probability of choosing action a_t is written as $\pi(a_t, s_t)$ or $\pi(a_t|s_t)$. Ideally, what we want is *good* actions to receive high probability and *bad* actions to receive low probability. Intuitively, good actions should lead to good states, and the notion of *good* is defined in terms of the future rewards that can be expected. A **value function** is used to estimate the future rewards that can be expected. A value function is a function that maps each state to an estimate of the expected return when starting in that state.

More formally, a *value function under a policy* π, denoted $v_\pi(s)$, is the expected return when starting in s and always taking actions suggested by the policy π:

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

where $\mathbb{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy π.

Having a value function we could compare actions by comparing the values of the states that these actions would lead to. However, doing this requires knowing the transition function of the environment. In case we have no access to the dynamics of the environment we could use an **action-value function**. An action-value function is a function that maps state-action pairs to an estimate of the expected return when performing that given action in that given state.

Similarly, a more formal description is that the *action-value function under a policy* π, denoted $q_\pi(s, a)$, is the expected return when starting in s, taking action a, and thereafter

acting according to the policy π:

$$q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R},$$
$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

Many times we would like to generate a trajectory by letting an agent act in the environment under a given policy π. We will call this process **policy rollout**. During policy rollout the agent acts according to a fixed policy π starting from an initial state s_0 and continuously transitioning to new states until a termination condition is reached. The result of the policy rollout is a complete trajectory – the ordered collection of states, actions, and rewards observed during the process.

Since we are concerned with maximising the expected return, we would like to find a policy such that at every step the action is taken that would lead to the best next state. Such a policy is known as an **optimal policy** π^*. The notions of optimal value function and optimal action-value function can also be defined in terms of the optimal policy. It should be noted that while there may be several optimal policies, the optimal value function and the optimal action-value function are unique.

The **optimal value function**, denoted $v^*(s)$, is the expected return when starting in s, and acting optimally:

$$v^*(s) = \max_\pi \mathbb{E}_\pi[G_t | S_t = s].$$

The **optimal action-value function**, denoted $q^*(s, a)$, is the expected return when starting in s, taking action a, and thereafter acting optimally:

$$q^*(s, a) = \max_\pi \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

There is an important connection between the optimal policy π^* and the optimal action-value function $q^*(s, a)$. The optimal action from $\pi^*(a|s)$, for a given state s, can be directly obtained from the optimal action-value function q^* by:

$$\pi^*(a|s) = \operatorname*{argmax}_{a'} q^*(s, a').$$

Thus, having the optimal action-value function, the optimal policy can easily be recovered. This is useful in certain tasks where it might be easier to learn the q-function, rather than learning directly the policy function.

Value functions satisfy a recursive relationship inherited from the relationship for the return in Eq. (2.6). These equations are known as the **Bellman optimality equations** and they state that the value of the state s is equal to the (discounted) value of the next state s' drawn from the transition probability distribution, plus the reward returned by the environment:

$$\begin{aligned} v^*(s) &= \max_\pi \mathbb{E}_\pi[G_t | S_t = s] \\ &= \max_\pi \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \max_a \mathbb{E}_{s' \sim \mathcal{P}}[\mathcal{R}(s, a, s') + \gamma v^*(s')], \end{aligned} \tag{2.8}$$

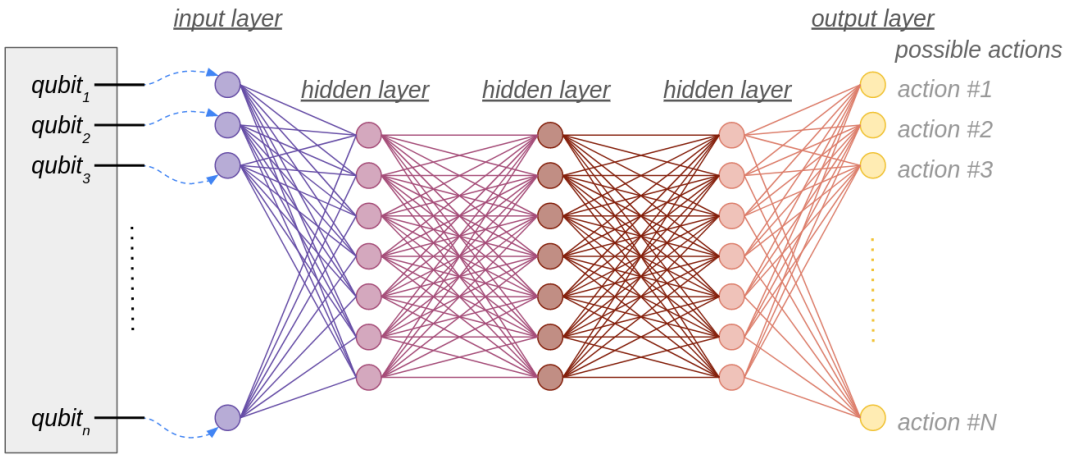


Figure 2.2: Function approximation with a deep neural network. The function takes as input the state of the environment s_t, and outputs a vector of probabilities assigned to each of the possible actions.

$$\begin{aligned} q^*(s,a) &= \max_\pi \mathbb{E}_\pi[G_t|S_t=s, A_t=a] \\ &= \max_\pi \mathbb{E}_\pi[r_{t+1} + G_{t+1}|S_t=s, A_t=a] \\ &= \mathbb{E}_{s'\sim\mathcal{P}}[\mathcal{R}(s,a,s') + \gamma \max_{a'} q^*(s',a')], \end{aligned} \tag{2.9}$$

where $\mathbb{E}_{s'\sim\mathcal{P}}[\cdot]$ denotes the expected value of a random variable given that the next state s' is sampled from the transition probability distribution $\mathcal{P}(\cdot|s,a)$.

One way to find an optimal policy is to solve the Bellman optimality equations. However, this requires an exhaustive search. We need to look at all possible outcomes, computing their respective probabilities of occurrence and summing their associated rewards. It is usually not feasible to solve the Bellman optimality equations and compute the optimal policy. In many cases there are far more states than we could ever visit, and in the case of continuous state spaces there is an infinite number of states. In such cases the value functions must be approximated using a parametrised function representation. This means that the output of the function depends on a set of parameters (e.g., weights and biases of a neural network). Function parameters will be denoted by θ, and function approximators are written with a subscript representing the parameters of the function:

$$\begin{aligned} &v_\theta(s), \\ &q_\theta(s,a), \\ &\pi_\theta(a|s). \end{aligned}$$

Fig. 2.2 shows an example approximation of a policy function π_θ using a deep fully-connected neural network. The input to the neural network will be the current state of the environment s_t, and the output produced will be a probability distribution over the action space. Each possible action will be assigned a probability score indicating how likely it is to be selected. Note that the output of the function depends on the weights θ of the model. In order for the neural network to be a good approximation of the actual optimal policy, we will need to train it by showing it numerous examples and adjusting the weights so that the model will fit those examples. A detailed procedure for training the neural network model is given in Sec. 2.1.4.

2.1.4 Policy gradient

Identifying **what** to learn in RL is a very important question and a critical branching point. Algorithms usually try to learn one of the following:

- policy function (π)
- value function (V-function)
- action-value function (Q-function)

In this thesis we focus on learning a policy $\pi(a_t|s_t)$ mapping a state s_t to a distribution over the action space $\mathcal{A}(s_t)$.

PG (Policy gradient) is a policy optimization method that uses a parametrised function representation of the policy π_θ and tries to optimise the parameters θ by performing gradient ascent updates. The updates are computed based on the gradient of a performance objective $J(\theta)$ with respect to the parameters θ. This type of optimization method is **on-policy**, meaning that updates must use data collected while acting with the most recent version of the policy. It is also **offline**, meaning that the episode has to end before the update can take place.

The performance objective that we aim to maximise is the expected return:

$$J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\mathcal{R}(\tau)\Big], \tag{2.10}$$

where the trajectory τ is sampled from the distribution $\mathcal{P}_\theta$ defined by the policy π_θ and the transition function $\mathcal{P}$. To sample a trajectory we actually have to sample a state $s_t \sim \mathcal{P}$ and an action $a_t \sim \pi_\theta$ for every time step t. The probability of a trajectory τ under a policy π_θ is given by:

$$\mathcal{P}_\theta(\tau) = p_0(s_0)\prod_{t=0}^{T}\pi_\theta(a_t|s_t)\mathcal{P}(s_{t+1}|s_t,a_t).$$

To optimise the parameters of the policy we will use a method based on the gradient of the objective with respect to the policy parameters θ. Since we are trying to *maximise* the objective our update method will approximate gradient ascent on the function $J(\theta)$:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\theta),$$

where α is the **learning rate** (or the *step size*). The parameter α is a hyper-parameter that must be fine-tuned during training.

Note that the objective function given in Eq. (2.10) is actually a function of the parameters θ. To see this we have to express the expectation as an integral over all possible trajectories:

$$J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\mathcal{R}(\tau)\Big] = \int \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau.$$

Although we have an analytic expression for the gradient of the objective, this is not very helpful. Since in general we have no access to the transition function $\mathcal{P}$ and to the reward function $\mathcal{R}$, there is no way to compute the gradient from this expression. Instead,

what we would like to do is express the gradient as an expectation over the probability distribution of the trajectories. To do this we will use the *policy gradient theorem*:

$$\nabla_\theta J(\theta) = \int \nabla_\theta \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau, \qquad (\text{since } \nabla_\theta \mathcal{R}(\tau) = 0)$$

$$\nabla_\theta J(\theta) = \int \mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau,$$

$$(\text{since } \nabla_\theta \mathcal{P}_\theta(\tau) = \mathcal{P}_\theta(\tau)\frac{\nabla_\theta \mathcal{P}_\theta(\tau)}{\mathcal{P}_\theta(\tau)} = \mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau))$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\nabla_\theta \log \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\Big],$$

$$\log \mathcal{P}_\theta(\tau) = \log \mathcal{P}(s_0) + \sum_{t=0}^{T}\Big[\log \pi_\theta(a_t|s_t) + \log \mathcal{P}(s_{t+1}|s_t, a_t)\Big],$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\mathcal{R}(\tau)\right]. \qquad (\text{since } \nabla_\theta \log \mathcal{P}(\cdot) = 0)$$

Having the gradient expressed as an expectation means that we can approximate it using samples. We can collect a set of trajectories $\mathcal{D} = \{\tau_i\}_{i=1,\dots,N}$, obtained using the agent-environment interaction loop. For each trajectory the agent uses the policy π_θ to choose actions. Then the policy gradient can be approximated by:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\mathcal{R}(\tau_i), \tag{2.11}$$

where N is the number of trajectories collected in $\mathcal{D}$. This method for calculating the gradient of the objective, also known as *Monte Carlo Policy Gradient*, gives rise to a pseudo-objective that we are trying to optimise:

$$J_{\text{pseudo}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\mathcal{R}(\tau_i). \tag{2.12}$$

The function given by Eq. (2.12) is usually called a pseudo-objective, because it does not evaluate the metric that we aim to optimize. Our goal is to maximize the expected return and this loss function is only useful to evaluate the gradient of our objective at the current parameters, with data generated by the current parameters.

2.1.5 Variance reduction

As a Monte Carlo method this method is of high variance and thus produces slow learning. In order to obtain a low variance estimate of the gradient we would need a huge amount of sample trajectories. To reduce the variance of the approximation, a simple trick is

used. For any probability distribution P_θ parametrised by θ we have the following:

$$\begin{aligned}\mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] &= \int P_\theta(x)\nabla_\theta \log P_\theta(x)\mathcal{D}x \\ &= \int P_\theta(x)\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)}\mathcal{D}x \\ &= \int \nabla_\theta P_\theta(x)\mathcal{D}x \\ &= \nabla_\theta 1 = 0. \end{aligned} \tag{2.13}$$

An immediate consequence of Eq. (2.13) is that for any function b, which does not depend on the action a_t, we have:

$$\mathbb{E}_{a_t\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)] = 0. \tag{2.14}$$

This allows us to add or subtract $b(s_t)$ from the expression in Eq. (2.11) without changing it in expectation. Any function b used in this way is called a **baseline**:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\left(\mathcal{R}(\tau_i) - b(s_t)\right). \tag{2.15}$$

The optimal baseline can be derived by minimising the variance of the analytic expression of the gradient. Detailed derivation of the formula is given in Ref. [20]:

$$b = \frac{\mathbb{E}[\nabla_\theta \log^2 \pi_\theta(\tau)\mathcal{R}(\tau)]}{\mathbb{E}[\nabla_\theta \log^2 \pi_\theta(\tau)]}.$$

The optimal baseline can be seen as the expected return when starting in s_0, weighted by the square of the log-probability of the policy. However, in practice, the type of baseline used is simply the expected return of the trajectory:

$$b = \mathbb{E}\Big[\mathcal{R}(\tau)\Big] \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T} r_{t+1}.$$

This baseline has the intuitive property of centering the returns of the trajectories in $\mathcal{D}$, so that trajectories that are better than average have a positive return, and trajectories that are worse than average have a negative return.

Another way to reduce variance is to notice that rewards obtained in previous time steps should have no effect on current actions. Agents should only reinforce actions based on future consequences and not on past experiences. Noting that previous rewards are not a function of the current action, and plugging into Eq. (2.14), it can be seen that:

$$\mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=0}^{t} r_{t'+1}\right] = 0.$$

Thus, in Eq. (2.11), instead of $\mathcal{R}(\tau) = \sum_{t=0}^{T} r_{t+1}$ we could use the *'reward-to-go'*, $\sum_{t'=t}^{T} r_{t'+1}$, without changing the expectation:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\sum_{t'=t}^{T} r_{t'+1,i}\right]. \tag{2.16}$$

This version of the equation is informally called *reward-to-go policy gradient*.

Putting together the ideas of *'reward-to-go'* and *'baseline'* we arrive at the following expression for the policy gradient:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\left(\sum_{t'=t}^{T} r_{t+1,i} - b(s_t)\right)\right]$$

The baseline used in this work is the expected value function of state s_t, approximated using the collected set of trajectories:

$$b(s_t) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t'=t}^{T} r_{t'+1,i}. \tag{2.17}$$

2.1.6 Entropy regularization

The trade-off between exploration and exploitation is one of the most important challenges in reinforcement learning. Agents should prefer actions yielding high rewards (exploit), but also, to find such actions, they have to try new actions that have not been selected before (explore).

When learning a policy with PG, exploration is ensured by the fact that the policy outputs a probability distribution over the action space and actions are selected in a non-deterministic manner. However, an action producing *some* positive reward is reinforced and thus the probability of selecting that action in the future is increased, which in turn reinforces the action again. It might happen that the agent will always select this very same action, while there could exist another action yielding a much higher return.

To encourage exploration we use a method called **entropy regularization** described in Ref. [21, 22]. The reward returned by the environment at every time step is augmented by an additional term - the entropy of the policy.

$$r_t^* = r_t + \beta^{-1} H(\pi_\theta(\cdot|s_t)),$$

where $H(\pi_\theta(\cdot|s_t))$ represents the entropy of the probability distribution of the actions that is proposed by the policy π_θ for the state s_t, and is computed as:

$$H(\pi_\theta(\cdot|s_t)) = -\sum_{a\in\mathcal{A}(s_t)} \pi_\theta(a|s_t)\log\pi_\theta(a|s_t) = -\mathrm{E}_{a\sim\pi_\theta}\Big[\log\pi_\theta(a|s_t)\Big], \tag{2.18}$$

and β is the inverse temperature factor that controls the level of regularization. Choosing $\beta = 0$ would apply a maximum level of regularization resulting in a purely stochastic policy, while choosing $\beta = \infty$ effectively removes the regularization.

Combining the techniques for variance reduction (*'reward-to-go'* and *'baseline'*) together with entropy regularization yields the following expression for the gradient of the RL objective:

$$\nabla_\theta J(\theta) \sim \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T}\nabla_\theta \log\pi_\theta(a_{t,i}|s_{t,i})\times \right.$$
$$\left.\times\left[\sum_{t'=t}^{T} r_{t'+1,i} - \beta^{-1}\sum_{t=0}^{T}\log\pi_\theta(a_{t,i}|s_{t,i}) - b(s_t)\right]\right], \tag{2.19}$$

where $b(s_t)$ is the baseline computed as follows:

$$b(s_t) = \frac{1}{N}\sum_{i=1}^{N} r_{t+1,i} - \beta^{-1}\log\pi_\theta(a_{t,i}|s_{t,i}).$$

Since our policy π_θ is parametrised by a neural network we would like to use an automatic differentiation software package [23, 24] to backpropagate the derivative of the objective $\nabla J(\theta)$ in order to update the weights of the model. To use auto-differentiation we would need to define a loss function such that its gradient is equal to the policy gradient. Considering the function taken by integrating Eq. (2.19) we can see that this function has the nice property that its gradient is equal to the policy gradient when the (s_t, a_t) pairs are collected while acting with the current policy:

$$J_{\text{pseudo}}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left[\sum_{t=0}^{T}\log\pi_\theta(a_{t,i}|s_{t,i})\times \right.$$
$$\left.\times\left[\sum_{t'=t}^{T} r_{t'+1,i} - \frac{1}{2}\beta^{-1}\sum_{t=0}^{T}\log\pi_\theta(a_{t,i}|s_{t,i}) - b(s_t)\right]\right]. \quad (2.20)$$

A detailed derivation of the formulas from Eqs. (2.19) and (2.20) can be found in Sec. B.

2.2 Imitation learning

In RL we are concerned with learning a policy that the agent can use to decide on which action to take, for any state of the environment. Most RL algorithms work by trial-and-error – the agent runs multiple episodes and progressively optimises its policy to obtain higher returns. However, if we are provided with a set of high-return trajectories that the agent should follow, we could instead use IL (Imitation learning) to learn a policy function.

In IL, instead of trying to learn the policy by optimising for the expected return with gradient ascent, we are provided with a set of sample trajectories by a knowledgeable expert. These sample trajectories are called *demonstrations*. The agent then tries to learn a policy that imitates the actions taken in the demonstrations, by maximizing a similarity measure. In general, this approach is useful when we have access to an expert at training time, and we can query the expert for more demonstrations or for evaluation.

Note that, given the state of the environment s_t, our policy defines a probability distribution over the action space $\pi_\theta(\cdot|s_t)$, and the expert produces one ground-truth action that should be taken, e.g., $a_k \in \mathcal{A}(s_t)$. The output of the expert can also be represented as a probability distribution, namely a 'one-hot' vector where all the values are set to zero, and only the value at the index k is set to unity:

$$\tilde{\pi}(\cdot|s_t) = \big(0, 0, \cdots, 0, 1, 0, \cdots, 0\big).$$

During training we would like the probability distribution produced by our neural network model to approximate the probability distribution produced by the expert. Thus, we will define a similarity measure between the two distributions, and we will try to maximize that measure by performing gradient ascent on it.

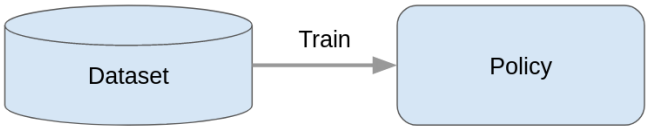


Figure 2.3: Behavioural cloning pipeline. The process for training an agent consists of two steps: 1) collecting a training set, and 2) training the policy using supervised learning with the loss function defined in Eq. (2.21)

The most commonly used similarity measure between probability distributions is the *Kullback-Leibler divergence*:

$$D_{KL}(\tilde{\pi}||\pi_\theta) = \sum_{a \in \mathcal{A}(s_t)} \tilde{\pi}(a) \log \frac{\tilde{\pi}(a)}{\pi_\theta(a)}.$$

In our case $\tilde{\pi}(a)$ is equal to zero in all but one cases, meaning that these terms will vanish from the sum leaving us with:

$$D_{KL}(\tilde{\pi}||\pi_\theta) = -\tilde{\pi}(a_k) \log \pi_\theta(a_k) = -\log \pi_\theta(a_k),$$

where a_k is the action selected by the expert.

Note that the KL divergence actually measures the difference between the two distributions, thus, we need to perform gradient descent instead. Using this function as a similarity measure gives rise to the well-known *cross-entropy loss* function:

$$J(\theta) = -\frac{1}{B} \sum_{i=1}^{B} \log \pi_\theta(a_i), \tag{2.21}$$

where B is the batch size for performing stochastic gradient descent.

The two most common approaches to applying IL are: *behavioural cloning* and *data aggregation*. We will briefly describe each of the two.

2.2.1 Behavioural cloning

The simplest form of IL is BC (Behavioural cloning). This form of IL tries to learn a policy using simple supervised learning. First, the set of trajectories provided by the expert is divided into state-action pairs, which form the training dataset. Then, supervised learning is applied by drawing random state-action pairs from the training data and back-propagating the loss signal. During training we treat the data points from the training set as i.i.d. (Independent and identically distributed). This assumption, however, is violated since the states and actions in a single trajectory are not at all i.i.d.. Nevertheless, in some cases BC works perfectly fine even though the i.i.d. assumption is not valid [25, 26].

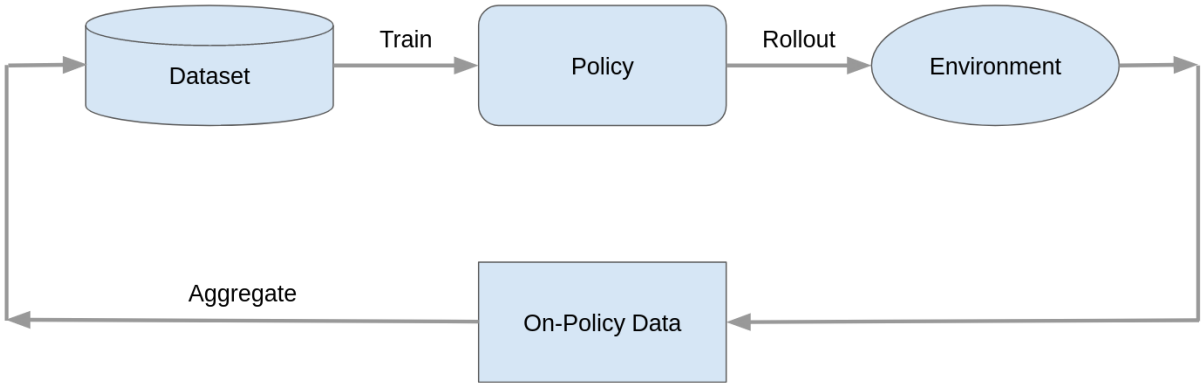


Figure 2.4: Data aggregation pipeline. The process of training an agent consists of four steps: 1) collecting a training set, 2) training a policy using supervised learning, 3) performing policy rollout to generate new data, 4) using an expert to label the newly generated data, and 5) aggregating the new training examples into the old training set and repeat.

2.2.2 Direct policy learning with data aggregation

As we mentioned earlier, the data points in the training set are not i.i.d.. Leaving aside the fact that this assumption is not valid, there is another more subtle problem with BC. Another assumption made for the training and testing data is that the data points in the test set are drawn from the same distribution as the training set. The validity of this assumption allows us to estimate the out-of-sample performance of our classifier by using the in-sample performance. Thus, minimising the in-sample performance results in minimising the out-of-sample performance. However, the distribution of the training data (p_{data}) is usually not the same as the distribution of the data during testing (p_{π_θ}). During testing the agent acts according to the learned policy, and small deviations from the demonstrated behaviour would force the agent into new states, thus modifying the distribution over the states in the test set. This problem is called a *distribution mismatch*.

A way to fix the issue is to change the data so that p_{data} equals p_{π_θ}. This approach is known as data aggregation, and the corresponding algorithm is called **DAgger**. DPL (Direct policy learning) with data aggregation is an improved version of BC; however, for this method to work we must have access to an expert that we can query at training time. First we train the policy on the initial training set $\mathcal{D}_i$. After that we collect trajectories by rolling out the trained policy, and we ask the expert to label every state of the trajectories. This way we collect a new training set $\mathcal{D}_\pi$. Finally, we aggregate the initial training set and the newly labelled training set $\mathcal{D}_{i+1} = \mathcal{D}_i \cup \mathcal{D}_\pi$, and we repeat the process. A detailed description of this algorithm can be found in Ref. [27].

2.3 Search algorithms

In a fully observable, deterministic, and known environment, the solution to any problem is a fixed sequence of actions, called a **path**. An **optimal solution** would be the path with the lowest cost. In our problem, that would be the shortest path. In this setting, the optimal solution could be found with a **state-space search**. The state space can be

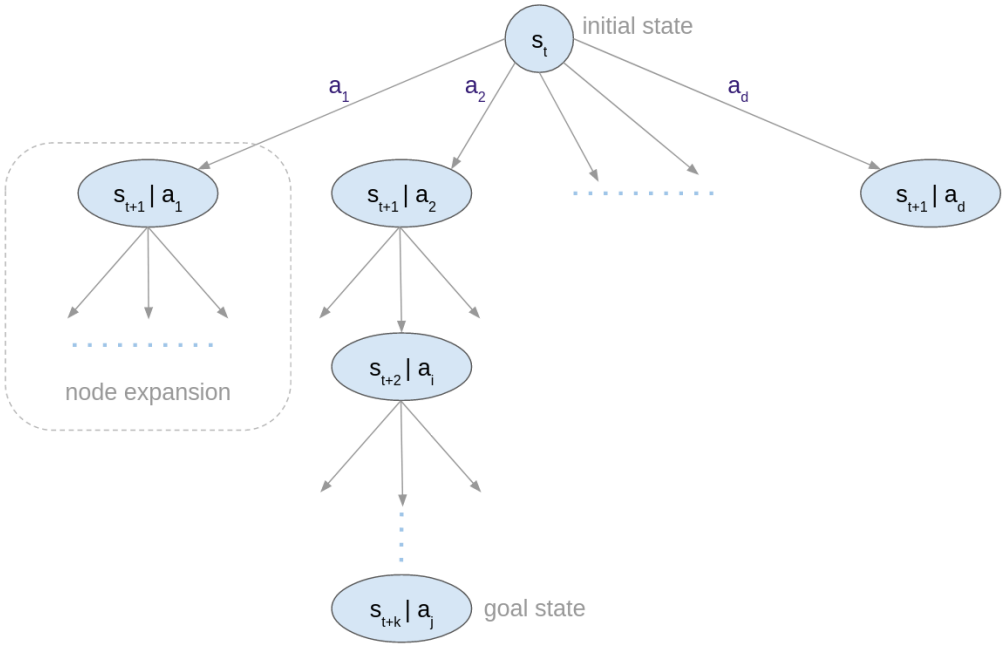


Figure 2.5: State-space tree search algorithm. Starting from the initial state, at every step we select one node and consider all possible actions $\{a_1, a_2, ..., a_d\}$ that can be taken from that node. Different search algorithm use different procedures for selecting the next node to be expanded. The search process continues until the goal node is reached.

represented as a graph. The states s_t are represented as the vertices of the graph, and the actions a_t transitioning one state to another are represented as directed edges.

A **search algorithm** works by superimposing a search tree over the state-space graph (see Fig. 2.5). The algorithm examines the various paths formed from the initial state, and tries to find a path that reaches a goal state. The initial state of the problem is placed at the root of the tree. A node is expanded by considering the available actions for that state, and generating a new **child** node for each of the states resulting after taking each of the actions. The set of all non-expanded nodes is called a **fringe** (see Fig.2.6. A search algorithm proceeds by iteratively selecting a node for expansion from the fringe (starting with the root), then generating all child nodes of that node, and finally adding the newly generated nodes to the fringe.

Basic search methods rely on systematically exploring the entire search-space until a solution is found. Heuristic search methods use a heuristic function to evaluate any node n of the tree and to guide the search towards more promising paths.

The performance of search algorithms can be evaluated in four ways:

- **Completeness:** is the algorithm guaranteed to find a solution when one exists, and to correctly report failure otherwise?
- **Optimality:** does the algorithm find an optimal solution out of all solutions?
- **Time complexity:** the time it takes to find a solution, usually measured in the number of expanded nodes;
- **Space complexity:** the memory needed to perform the search.

In this thesis two common search algorithms are used. These are A^* *search* and *beam search*. We will describe how these algorithms operate and compare their performance.

2.3.1 A* search

A^* **search** is the most common informed search algorithm. The algorithm uses an evaluation function $f(n)$ to evaluate any node n in the fringe, and always expands the node with the minimum value. The evaluation function consists of two parts:

$$f(n) = g(n) + h(n)$$

where:

- $g(n)$ is the path cost from the initial node to the node n;
- $h(n)$ is the estimated cost of a path from node n to a goal node.

The A^* algorithm is complete and, depending on the heuristic function, it could be optimal. The key properties that the heuristic function must have in order for A^* to be optimal are:

- admissibility: it never overestimates the cost to reach the goal (it is optimistic);
- consistency: for every node n and every successor n' of n generated by an action a, we have: $h(n) \leq \text{cost}(n, a, n') + h(n')$.

A complete description of the A^* algorithm can be found in Ref. [28].

The main issue with A^* is its use of memory. All evaluated and unexpanded nodes are kept in the fringe and during the search process the fringe grows exponentially large. However, this problem can be overcome by using a variant of A^* called IDA* (Iterative-deepening A^* search).

Another problem is that A^* expands a lot of nodes, i.e., it takes a lot of time to find a solution. To find the solution, the algorithm must expand all nodes that have cost lower than the optimal cost $f(n) \leq C^*$, and in general there are exponentially many such nodes. However, if we are willing to accept solutions that are sub-optimal, but are good enough according to some extra criterion, we can run A^* with an **inadmissible heuristic** – one that may overestimate. This way we risk missing the optimal solution, but the heuristic could guide the search to a solution while expanding fewer nodes.

2.3.2 Beam search

Global search algorithms are usually inefficient in large state spaces because they have to explore the entire state space to find the optimal solution. Informed search algorithms such as A^* use heuristics to prune the search tree; however, only paths that are certain not to be optimal are pruned. This is a problem because we are still left with an exponential amount of work to do. This type of pruning reduces the problem multiplicatively, not exponentially.

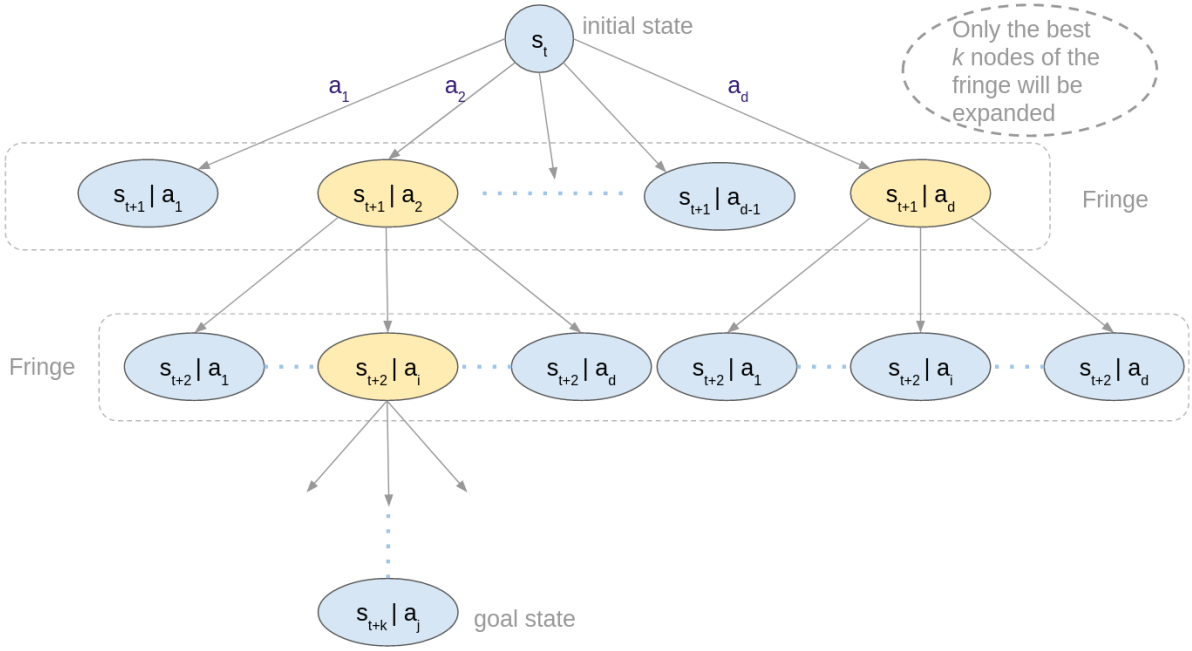


Figure 2.6: Beam search algorithm. At every step we consider a set of nodes called the *fringe* which will be expanded. Initially only the starting state is in the fringe. At every iteration we select the top k nodes from the fringe and expand them. After that a new fringe containing all the newly expanded nodes is created. This process continues iteratively until a goal node is reached.

Local search algorithms, on the other hand, are usually very efficient and quickly find solutions to the problem. However, these solutions are not guaranteed to be optimal. During the process of pruning, local search algorithms prune more aggressively and, thus, a path leading to the optimal solution might be pruned, without ever being considered. Obviously there is a trade-off between optimality and efficiency. If we are required to find an optimal solution at all costs, then local search will not do the job. Local search algorithms, however, use little memory and can often find a good near-optimal solution for a reasonable amount of time.

Beam search is a kind of local search algorithm. Thus, beam search is neither complete, nor optimal. However, local search algorithms use little memory and can often find a good near-optimal solution for a reasonable amount of time. Like A^*, beam search is also an informed search algorithm, which means that it also uses an evaluation function $f(n)$.

In this work we will be using beam search which is a kind of a local search algorithm. The working process of beam search is shown in Fig. 2.6.

Note that A^* uses an evaluation function that tries to predict the length of the path from the current node to the goal node. In order for A^* to work properly this function must have the properties of *admissibility* and *consistency*. However, designing a non-trivial consistent heuristic is a very hard task and that is why researchers are exploring different ways of applying inconsistent heuristics [29].

Beam search on the other hand is designed to work using an evaluation function that does not need to be consistent. The algorithm uses a scoring function $s(n)$ that assigns a score to every node. Nodes that are closer to the goal should be scored higher, however

this is not a necessary condition.

The function $s(n)$ simply assigns a score to every node, with the aim to assign higher scores to nodes that are closer to the goal, and lower scores to nodes that are further away from the goal. Note, however, that this is not a necessary condition and the score function is not perfect. It might assign a lower score to a node that is actually along the optimal path. The algorithm would then prune that node, thus losing the optimal path. If we had access to a perfect scoring function we would not need to perform any kind of search and would just expand only the best node at every step.

The algorithm works by limiting the size of the fringe to a fixed size k, called the beam size. At every step the algorithm expands **all** of the nodes in the fringe, instead of picking one. All of the generated successors are evaluated using the evaluation function $s(n)$, and the k best successors are kept in the fringe. The k best successors represent the k best paths the algorithm continues to extend while trying to find a solution. High-cost paths are abandoned and only low-cost paths are expanded in the next iteration.

Chapter 3

Quantum computing prerequisites

In this chapter we introduce the field of quantum mechanics, defining the most important quantities that are used in this work. We start off by revisiting classical computers in Sec. 3.1. Qubits – the quantum version of classical bits are introduced in Sec. 3.2. Multi-qubit quantum systems are described in Sec. 3.3 and in Sec. 3.4 we show how operations are performed on a quantum computer. One of the most important properties of quantum systems – quantum entanglement; is examined in Sec. 3.5 and 3.6. For a thorough review of the subject of Quantum Mechanics the reader is referred to [30, 31].

3.1 Classical computers

The simplest unit of computation is the classical bit. Classical bits take one of two values – 0 or 1. All the algorithms and data structures are entirely built up from classical bits by clever manipulations with classical logic gates (AND, OR, XOR, etc.).

The quantum version of the classical bit is called a **qubit**. When measured, it also can only give an output of 0 or 1. We will later describe what we mean by *'measurement'*, but it is important to point out that the internal structure of the qubit is much more complicated that 0s and 1s. Qubits can also be manipulated in new ways that can only be described by the laws of quantum mechanics. Thus, new gates exit, called **quantum gates** that will be used.

3.1.1 Bits

A classical bit is a way of describing a system whose set of states is of size two. We usually write these two possible states as 0 and 1, or as F and T.

We could also represent the two states in vector form:

$$\mathrm{F} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathrm{T} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

This is simply a one-hot vector representation, where the position of the 1 indicates in which state the system currently is. Using this representation of a bit, we could describe

one byte (8-bits) as follows:

$$11010011 = \text{TTFTFFTT} = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix}. \tag{3.1}$$

An 8-bit system is fully described by listing the state in which each of the eight bits currently is. Thus, the number of parameters needed to describe the full system, using this representation, grows linearly with the size of the system. However, if we try to represent the entire 8-bit system as a one-hot vector it will be a 256-dimensional vector with a single 1 positioned at the index of the current state of the system. We can see that trying to represent states in this form results in an exponential growth:

$$11010011 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \begin{matrix} 00000000 \\ 00000001 \\ \vdots \\ 11010011 \\ \vdots \\ 11111111 \end{matrix}.$$

More formally, the one-hot vector of a classical system can be formed by taking the *tensor product* of the one-hot vectors representing the individual bits. Considering a two-bit system, the tensor product is applied as follows:

$$10 = \text{TF} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ 1 \times \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}.$$

Although this representation is not used when working with classical computers, it would be useful for us before transitioning to quantum systems. We will see that, when we try to simulate quantum systems on a classical computer, we cannot simply list the state of each individual qubit. Rather, we will be compelled to write them in this tensor-product form.

3.1.2 Gates

Bits are manipulated by applying logic gates. When our system is represented using a one-hot vector, we can see that we could also represent logic gates using a matrix form. The application of a logic gate to a system is then mathematically modelled by multiplying the system state vector from the left by the gate matrix. For example, logical NOT operates on a single bit and can be represented as a 2×2 matrix. Logical AND, on the other hand, operates on two bits and outputs a single bit. Thus, we could represent that operation using a 2×4 matrix:

$$\text{NOT} = \begin{matrix} \text{F} & \text{T} \\ \end{matrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{matrix} \text{F} \\ \text{T} \end{matrix}, \quad \text{AND} = \begin{matrix} \text{FF} & \text{FT} & \text{TF} & \text{TT} \\ \end{matrix} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} \text{F} \\ \text{T} \end{matrix}.$$

We can see that the matrix form of a logic gate consists of 0s and 1s and actually corresponds to the truth table that would represent this operation. Then, inverting an F-bit into a T-bit would be written as:

$$\begin{matrix} & & \text{F} & & \text{T} \\ \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & = & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{matrix}.$$

AND-ing two bits (T&F = F) would be written as:

$$\begin{matrix} & & & & \text{TF} & \\ \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} & = & \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \text{F} \end{matrix}.$$

One gate of special importance to classical computers is the NAND gate, because it has the property of functional completeness. That is, every other boolean function can be implemented using only a combination of NAND gates. The NAND gate corresponds to applying an AND gate followed by a NOT gate. Applying multiple gates in succession can be represented by chaining matrix multiplications from the left:

$$\text{NAND} = \text{NOT} \times \text{AND} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}.$$

It should be noted that chaining the AND and the NOT gate is possible, because the size of the output of the AND gate matches the size of the input for the NOT gate. However if we would like to first apply the NOT gate to one of the bits and then apply the AND gate, things get a little different. Applying a NOT gate only on the first bit of a two-bit system corresponds to leaving the second bit unchanged by applying an identity gate on it. To see how this operation would be applied to the entire system state vector, we can first apply the NOT and Id gates to the separate bits and after that get the tensor product. Mathematically, this can be expressed as follows:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}.$$

Using the distributive property of the tensor product we can see that this corresponds to:

$$\left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right) \left(\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} \otimes \begin{pmatrix} b_0 \\ b_1 \end{pmatrix} \right).$$

Thus, we can act on the entire system with the tensor product between the NOT gate and the Id gate:

$$\text{NOT} \otimes \text{Id} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & 1 \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\ 1 \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} & 0 \times \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix},$$

$$\begin{matrix} & & & & \text{TF} \\ \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & & & & \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \end{matrix} = \begin{matrix} \\ \\ \\ \\ \text{FF} \\ \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \end{matrix}.$$

A nice property of the NOT gate is that the number of input bits is the same as the number of output bits. In the last example we can see that applying the NOT $\otimes$ Id gate leaves the size of the system unchanged. This property makes chaining operations somewhat easier as we do not need to keep track of the number of bits in our system. As we will see later, all quantum gates have this property.

In general, if we have a system of n bits and we would like to apply gate X_1 on the first k_1 bits, and gate X_2 on the next k_2 bits, and so on, and finally apply gate X_m on the last k_m bits ($k_1 + k_2 + \cdots + k_m = n$), then we need to consider the tensor product $X_1 \otimes X_2 \otimes ... \otimes X_m$. As mentioned earlier, when working with quantum systems we will have to use the full tensor-product form to mathematically represent the system. Thus, applying one gate on one subset of the qubits and another gate on the other subset will require using the tensor product between these gates. We will elaborate more on that in Sec 3.4.3.

3.2 Quantum bits of information (qubits)

3.2.1 From classical to quantum

Classical bits always have a well-defined state – they are either 0 or 1, either F or T, but they cannot be both. However, in the quantum world things are different. The famous Stern-Gerlach experiment [32] was explained by proposing that electrons have an entirely new property associated with them called *spin*. Spin is a property of the quantum world with no classical analogue, although it bears some formal relation to angular momentum. When we measure the spin of a particle in a given direction it can only be found in two states – it either spins anti-clockwise (spin up) or clockwise (spin down), w.r.t. the measurement direction. The result of the measurement depends on the initial state of the particle, which is described by a two-dimensional complex vector:

$$\psi = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \begin{matrix} up \\ down \end{matrix},$$

where $\alpha, \beta \in \mathbb{C}$.

The squares of the absolute values $|\alpha|^2$ and $|\beta|^2$ determine the probabilities that the system collapses in each of the two possible states after measurement. In order to ensure that the total probability is unity we must have $|\alpha|^2 + |\beta|^2 = 1$. The numbers α and β are known as the complex *amplitudes* of the state ψ. The name comes from the fact that ψ behaves mathematically as a complex wave when its time evolution is studied.

We can see that the state vector of a qubit ψ can be decomposed as:

$$\psi = \begin{bmatrix}\alpha \\ \beta\end{bmatrix} = \alpha \begin{bmatrix}1 \\ 0\end{bmatrix} + \beta \begin{bmatrix}0 \\ 1\end{bmatrix}.$$

We say that our quantum state is in a *superposition* between the two *basis* states. It is in both states simultaneously, and only when measured, it collapses to one of the two basis states.

In quantum mechanics *bra-ket* notation is used to denote vectors that represent quantum states. Thus, we can write the previous equation using **ket**s as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,,$$

where:

$$|0\rangle = \begin{bmatrix}1 \\ 0\end{bmatrix}, \quad |1\rangle = \begin{bmatrix}0 \\ 1\end{bmatrix}.$$

For the given spin directions (up and down), the vectors $|0\rangle$ and $|1\rangle$ are the canonical basis vectors of the vector-space $\mathbb{C}^2$.

Every ket vector has a corresponding **bra** which augments the vector space with an inner product. The bra of a ket vector $|\psi\rangle$ is the row vector arrived at by taking the hermitian conjugate:

$$\langle\psi| = |\psi\rangle^\dagger = \begin{bmatrix}\bar{\alpha}, & \bar{\beta}\end{bmatrix},$$

where $\bar{\alpha}$ and $\bar{\beta}$ denote the complex conjugates of α and β respectively.

The inner product between two vectors $|\phi\rangle$ and $|\psi\rangle$ is denoted as $\langle\phi|\psi\rangle$, and is evaluated by performing a matrix multiplication between a conjugated row vector and a column vector. The result is a complex number known as the *transition amplitude*. The transition amplitude allows us to determine how likely it is for a state to transition into another state upon measurement. The probability p of state $|\psi\rangle$ transitioning into state $|\phi\rangle$ is equal to the square of the absolute value of the transition amplitude. Thus:

$$\begin{aligned} A &= \langle\phi|\psi\rangle = \begin{bmatrix}\bar{\gamma}, & \bar{\delta}\end{bmatrix} \begin{bmatrix}\alpha \\ \beta\end{bmatrix} = \bar{\gamma}\alpha + \bar{\delta}\beta, \\ p(|\psi\rangle &\rightsquigarrow |\phi\rangle) = |A|^2. \end{aligned}$$

For example, when measuring the spin along the z-direction, we would like to compute how likely it is that our system transitions into a spin-up state. The transition amplitude is given by taking the inner product between $|0\rangle$ and $|\psi\rangle$, and the probability for this transition is calculated by taking the square of the absolute value of this transition amplitude:

$$\begin{aligned} A &= \langle 0|\psi\rangle = \begin{bmatrix}1, & 0\end{bmatrix} \begin{bmatrix}\alpha \\ \beta\end{bmatrix} = \alpha, \\ p(|\psi\rangle &\rightsquigarrow |\uparrow\rangle) = |A|^2 = |\alpha|^2. \end{aligned}$$

3.2.2 Observables and measurement

The state of a particle is described by a two-dimensional complex vector, but we have no access to the components of the vector. We can only observe the direction of the

spin (up or down) when a measurement is performed. Thus, observables are the real physical quantities that are actually possible to be observed. In quantum mechanics to every physical observable there corresponds a *hermitian* operator.

To understand what this means let us consider as an example the spin operator along the z direction:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

We will shortly show how this operator is arrived at. One property that hermitian operators have is that their eigenvalues are all real numbers. The reason why this is important is because the eigenvalues of the operator are the only values that can be observed as a result of the measurement. The eigenvalues of the operator Z are $\lambda_0 = 1$ and $\lambda_1 = -1$, which corresponds to measuring the spin of the particle along the direction of the $+z$ or $-z$ axis.

In order to understand how to calculate which of the possible values of an observable is measured, we need to describe two differences that are present between classical systems and quantum systems:

- In classical mechanics the act of measurement would leave the system in whatever state it was. However, in quantum mechanics states get perturbed and modified as a result of measuring. In fact, performing a measurement *collapses* the quantum system into one of the eigenvectors of the observable operator;
- In classical mechanics the result of a measurement can be predicted with certainty if we know the state of the classical system. In quantum mechanics, a measurement is inherently a non-deterministic process, only the probabilities of observing the possible values can be calculated.

If the result of measuring the state $|\psi\rangle$ with the operator Ω is the eigenvalue λ_i, then the system state collapses into the eigenvector $|e_i\rangle$ corresponding to the measured eigenvalue. In order to calculate the probability of measuring any of the possible values, we can calculate the probability of our current state $|\psi\rangle$ transitioning into the corresponding eigenvectors. Going back to our example, the probability of measuring our particle in the spin-up state, that is measuring λ_0, is equal to the probability of our state transitioning into the eigenvector $|e_0\rangle$ corresponding to the spin-up state.

The eigenvectors of an observable operator form an orthonormal basis for the state space of our quantum system. Therefore, we can express $|\psi\rangle$ as a linear combination in this basis:

$$|\psi\rangle = c_0 |e_0\rangle + c_1 |e_1\rangle + ... + c_{n-1} |e_{n-1}\rangle . \tag{3.2}$$

Now it is obvious that the values $|c_0|^2, |c_1|^2, ..., |c_{n-1}|^2$ give exactly the probabilities that we measure the corresponding eigenvalues of the observable operator, since $c_i = \langle e_i|\psi\rangle$. From this we can conclude that the state $|\psi\rangle$ can be expressed in the basis of the observable operator as:

$$|\psi\rangle = \langle e_0|\psi\rangle |e_0\rangle + \langle e_1|\psi\rangle |e_1\rangle + ... + \langle e_{n-1}|\psi\rangle |e_{n-1}\rangle .$$

Finally, using the eigen decomposition property we can write any hermitian operator Ω as:

$$\Omega = \sum_i \lambda_i |e_i\rangle \langle e_i| .$$

Using this formula we can reconstruct the representation of our spin operator in the z direction. We know that our basis vectors $|0\rangle$ and $|1\rangle$ are eigenvectors of the spin operator, because when measured particles will collapse into one of these two states. And we also know that our eigenvalues are $\lambda_0 = 1$ and $\lambda_1 = -1$, because we can only measure whether the spin is along or opposite the z direction. Thus, for the spin operator we have:

$$Z = |0\rangle\langle 0| - |1\rangle\langle 1| = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix} - \begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0 & 1\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & -1\end{bmatrix}$$

3.2.3 The Bloch sphere

Note that using Euler's formula for representing complex numbers, $\alpha = re^{i\varphi}$, we can express any quantum state as:

$$\begin{aligned}|\psi\rangle &= \alpha|0\rangle + \beta|1\rangle = \\ &= e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right) \\ &= e^{i\gamma}|\phi\rangle,\end{aligned}$$

where

$$|\phi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle.$$

Now suppose that we try to measure the states $|\psi\rangle$ and $|\phi\rangle$ using some observable operator Ω and let $|x\rangle$ be an arbitrary eigenvector of Ω. Then, for the probability of measuring the eigenvalue corresponding to $|x\rangle$ we have:

$$|\langle x|\psi\rangle|^2 = |\langle x|e^{i\gamma}|\phi\rangle|^2 = \left|\begin{bmatrix}\bar{x_1} & \bar{x_2}\end{bmatrix}\begin{bmatrix}e^{i\gamma}\alpha\\e^{i\gamma}\beta\end{bmatrix}\right|^2 = \tag{3.3}$$

$$= |e^{i\gamma}|^2|\langle x|\phi\rangle|^2 = \tag{3.4}$$

$$= |\langle x|\phi\rangle|^2. \tag{3.5}$$

The variable γ is known as *global phase*. Two states that only differ by global phase cannot be distinguished from one another. The reason for this is that they behave identically during measurements and measurement is the only way we can extract information from qubits.

Thus, we can safely ignore the factor $e^{i\gamma}$ when representing quantum states. Hence, the statevector of any qubit can be described using only two real variables θ, φ:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle,$$

where

$$0 \leq \theta \leq \pi, \quad 0 \leq \varphi < 2\pi.$$

If we interpret θ and φ as spherical coordinates we can plot any single qubit state on the surface of a unit sphere, known as the *Bloch sphere*, as shown in Fig. 3.1. The Bloch sphere is very useful for visualising the state of a single qubit; however, it must be kept in mind that there is no simple generalization of the Bloch sphere for multiple qubits.

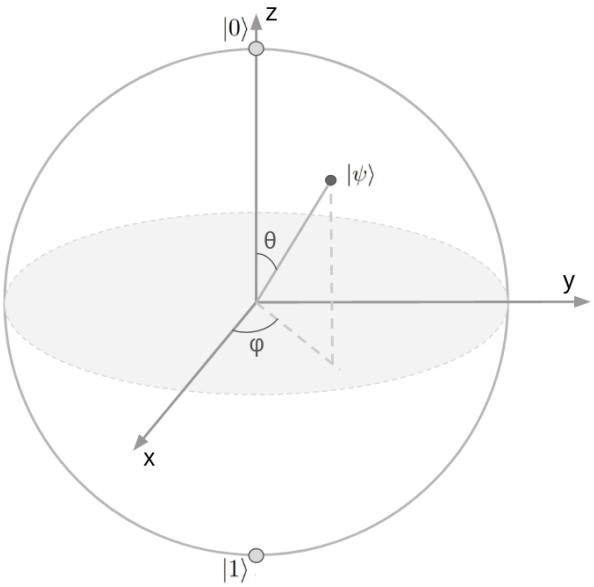


Figure 3.1: The Bloch sphere. Geometrical representation of a single qubit state vector. The north and south poles show the standard basis vectors $|0\rangle$ and $|1\rangle$.

3.3 Multi-qubit systems

3.3.1 Hilbert spaces

The state spaces of quantum system are *Hilbert spaces*, that is they are complex inner product spaces that are complete. In digital quantum computing we will only deal with finite dimensional spaces. However, in quantum mechanics the state space of a quantum system may be infinitely dimensional, and satisfies additional technical restrictions.

In order to combine quantum systems one has to use the tensor product. We know that a single qubit is an element of $\mathbb{C}^2$. Thus, the state of a system consisting of n qubits is an element of $\underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes ... \otimes \mathbb{C}^2}_{\text{n times}}$.

As an example, the state vector of a 2 qubit system is an element of $\mathbb{C}^2 \otimes \mathbb{C}^2 = \mathbb{C}^4$. There are four computational basis vectors for the $\mathbb{C}^4$ space denoted $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. These are formed by taking the tensor product between the basis vectors in the $\mathbb{C}^2$ space,

namely:

$$|00\rangle = |0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix},$$

$$|01\rangle = |0\rangle \otimes |1\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\1\\0\\0\end{bmatrix},$$

$$|10\rangle = |1\rangle \otimes |0\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\\1\\0\end{bmatrix},$$

$$|11\rangle = |1\rangle \otimes |1\rangle = \begin{bmatrix}0\\1\end{bmatrix} \otimes \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}.$$

Any 2-qubit state exists in a superposition of these four states, thus the state vector of such a system can be written as:

$$|\psi\rangle = \alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle\,, \tag{3.6}$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{C}$. Similar to the case of a single qubit, the squares of the absolute values of the amplitudes determine the probabilities that our system collapses to each of the basis states upon measurement. We therefore have again the normalization condition:

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1.$$

Analogously, the basis vectors for the $\mathbb{C}^{2^n} = \mathbb{C}^2 \otimes \mathbb{C}^2 \otimes ... \otimes \mathbb{C}^2$ space can be derived by performing the tensor products between the basis vectors of the individual spaces. And thus, the state vector of any n-qubit system can be expressed as:

$$|\psi\rangle = c_0\,|e_0\rangle + c_1\,|e_1\rangle + ... + c_{2^n-1}\,|e_{2^n-1}\rangle\,,$$

where $|e_0\rangle\,, |e_1\rangle\,, ..., |e_{2^n-1}\rangle$ are the basis vectors of the $\mathbb{C}^{2^n}$ space.

3.3.2 Quantum entanglement

We know that performing a measurement on a quantum system collapses the system into one of the basis vectors of the observable operator. Thus, all of the qubits collapse to a well-known state, either $|0\rangle$ or $|1\rangle$. However, for a multi-qubit system we could measure any subset of the qubits. For example, in our two-qubit system from Eq. (3.6) if we measure only the first qubit we would receive a value of 0 with probability $|\alpha|^2 + |\beta|^2$, which corresponds to our system collapsing into a superposition of $|00\rangle$ and $|01\rangle$. After this measurement our collapsed state would be:

$$|\psi'\rangle = \frac{\alpha}{\sqrt{|\alpha|^2 + |\beta|^2}}\,|00\rangle + \frac{\beta}{\sqrt{|\alpha|^2 + |\beta|^2}}\,|01\rangle\,. \tag{3.7}$$

Important two qubit quantum states are the *Bell* states. One of them is given in Eq.(3.8)

$$|\psi_{\text{Bell}}\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{3.8}$$

The state $|\psi_{\text{Bell}}\rangle$ has the property that the measurement outcomes of the separate qubits are correlated. If we measure the first qubit and after that we measure the second qubit, then the second measurement will *always* produce the same output as the first measurement. Observe that if the result of measuring the first qubit is 0, then applying Eq. (3.7) yields that our state collapses to the basis state $|00\rangle$, and thus measuring the second qubit must produce the result 0 again. The same reasoning applies if the result of measuring the first qubit is 1. In this case we say that the states are **entangled**. As it turns out representing entangled quantum systems by listing the state vectors of each qubit separately, as for the classical system in Eq. (3.1), is impossible. The individual states of the system are intimately related to one another and the only way to represent the state vector of the system is to use the full tensor product.

To see why this is true assume that we could list the qubits of the Bell state separately. Thus, assume that there exist $\alpha, \beta, \gamma, \delta \in \mathbb{C}$ such that:

$$\begin{aligned} |q_0\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}, \quad |q_1\rangle = \begin{bmatrix} \gamma \\ \delta \end{bmatrix}, \\ |\psi_{\text{Bell}}\rangle = |q_0\rangle \otimes |q_1\rangle = \begin{bmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix}. \end{aligned} \tag{3.9}$$

From Eq. (3.9) we can see that we are left with the following system of equations, which has no solution:

$$\begin{cases} \alpha\gamma = \frac{1}{\sqrt{2}} \\ \alpha\delta = 0 \\ \beta\gamma = 0 \\ \beta\delta = \frac{1}{\sqrt{2}} \end{cases}$$

Thus, the state $|\psi_{\text{Bell}}\rangle$ cannot be represented by listing each of its qubit state vectors separately.

In general, quantum states that can be separated into the tensor product of constituent subsystems are called *product* states, or also *separable* states. Quantum states that cannot be separated into the tensor product of any subsystems are called *entangled* states.

3.4 Quantum gates

Computations on a classical computer are conducted by applying logic gates to manipulate the system. Analogously, computations on a quantum computer are conducted by

applying quantum gates to the state of the quantum system.

3.4.1 Unitary gates

As we already said, the action of a gate on a specific state is found by multiplying the state vector $|\psi\rangle$, which represents the quantum state, by the operator matrix that represents the gate. The state vector of any quantum state is a unit vector from the Hilbert space $\mathbb{C}^{2^n}$, where n is the number of qubits in the system. Applying a quantum gate produces a new quantum state, which, in order to be a valid quantum state, must be of unit length. Thus, operations on a qubit system must preserve the norm:

$$\|U|\psi\rangle\| = \||\psi\rangle\|,$$

which is equivalent to:

$$\langle\psi U^\dagger|U\psi\rangle = \langle\psi|U^\dagger U|\psi\rangle = \langle\psi|\psi\rangle\,. \tag{3.10}$$

Equation (3.10) is valid if and only if the matrix U is a unitary matrix, i.e.,

$$UU^\dagger = U^\dagger U = I. \tag{3.11}$$

From here we see that every quantum gate is in fact a unitary matrix. This also implies that noise-free operations performed with quantum gates are reversible. If a given quantum gate U is applied to a quantum system, then the matrix $U^\dagger$ represents the quantum gate that would undo our previous operation:

$$\begin{aligned} U|\psi\rangle &= |\phi\rangle\,, \\ U^\dagger|\phi\rangle &= U^\dagger U|\psi\rangle = |\psi\rangle\,. \end{aligned}$$

In a reversible computation no information is ever erased, because the input can always be recovered from the output.

3.4.2 Single-qubit gates

A quantum gate is an operator that acts on the qubits, and is represented by a unitary matrix. Single-qubit operators are 2×2 matrices, and applying the operator is done by multiplying the qubit state vector with the operator matrix. The result of the multiplication is the state vector of the quantum system that results from applying this quantum gate.

Three very important single-qubit gates are the *Pauli* matrices:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

They occur in the Pauli equation, which is a formulation of the Schroedinger equation for spin-$\frac{1}{2}$ particles, which takes into account the interaction of the particle's spin with an external electromagnetic field.

Each Pauli matrix is Hermitian, and together with the identity matrix I, the Pauli matrices form a basis for the real vector space of 2×2 Hermitian matrices. Thus, any

2×2 Hermitian matrix can be represented as a linear combination of these four matrices using only real coefficients.

Using the Bloch sphere (see Fig. 3.1) we can visually represent the action of qubit gates on a single qubit. The action of every unitary 2×2 matrix changes the amplitudes of our qubit and thus changes the spherical coordinates θ and φ used to represent the qubit on the Bloch sphere. Suppose a single qubit is represented by the vector $\vec{h}$ on the Bloch sphere. Then we could apply a rotation operator $R(\gamma)$ that rotates this vector by an angle γ radians along a given axis. This rotation operator is defined in terms of the Pauli matrices. For example, performing a rotation on the quantum state by γ radians along the x axis can be done by the operator:

$$R_x(\gamma) = \exp\left(-i\frac{\gamma}{2}X\right) = \cos\frac{\gamma}{2}I - i\sin\frac{\gamma}{2}X.$$

Similarly we could define rotation operators along the y and z axes:

$$R_y(\gamma) = \exp\left(-i\frac{\gamma}{2}Y\right) = \cos\frac{\gamma}{2}I - i\sin\frac{\gamma}{2}Y,$$
$$R_z(\gamma) = \exp\left(-i\frac{\gamma}{2}Z\right) = \cos\frac{\gamma}{2}I - i\sin\frac{\gamma}{2}Z.$$

Note that the Pauli matrices are themselves unitary operators, which means that they are valid single-qubit quantum gates and define the following effects on a qubit state:

- Bit-flip
$$X|0\rangle = |1\rangle, \qquad X|1\rangle = |0\rangle,$$
- Phase-flip
$$Z|0\rangle = |0\rangle, \qquad Z|1\rangle = -|1\rangle,$$
- Bit-Phase-flip
$$Y|0\rangle = i|1\rangle, \qquad Y|1\rangle = -i|0\rangle.$$

Finally, note that any quantum gate can be represented as rotating the vector on the Bloch sphere along some axis. Applying a rotation along an arbitrary axis $\vec{n} = (n_x, n_y, n_z)$ is done by the rotation operator:

$$R_n(\gamma) = \cos\frac{\gamma}{2}I - i\sin\frac{\gamma}{2}(n_xX + n_yY + n_zZ). \tag{3.12}$$

3.4.3 Multi-qubit gates

For quantum computing one of the most important features of a multi-qubit quantum state is the presence of entanglement between individual qubits. We would like to be able to modify the degree of entanglement of such systems. However, when applying single-qubit gates, we only manipulate individual qubits without changing the degree of entanglement of that qubit with the rest of the system. In order to create or destroy entanglement we need to apply qubit gates that act simultaneously on more that one qubit, that is, we need to apply multi-qubit quantum gates [33].

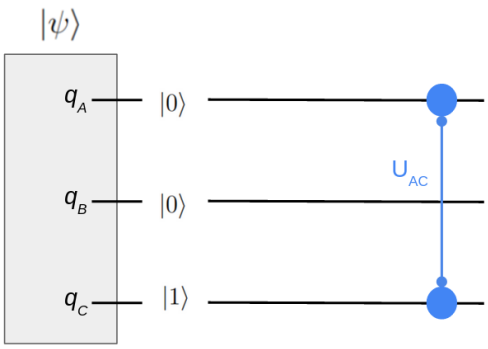


Figure 3.2: Quantum circuit for a 3-qubit quantum state. Qubits A and B are prepared in state $|0\rangle$, while qubit C is prepared in state $|1\rangle$. The quantum gate U is applied to qubits A and C.

It should be noted that not all multi-qubit gates modify the degree of entanglement of a quantum state. For example, we could form a two qubit gate by simultaneously applying two single-qubit gates on two separate qubits. Consider applying gate U_1 on the first qubit and applying gate U_2 on the second qubit of a two qubit system. This can be done by applying the tensor product $U_1 \otimes U_2$ to the entire system:

$$U_A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, U_B = \begin{bmatrix} e & f \\ g & h \end{bmatrix},$$
$$U = U_A \otimes U_B = \begin{bmatrix} ae & af & be & bf \\ ag & ah & bg & bh \\ ce & cf & de & df \\ cg & ch & dg & df \end{bmatrix}.$$

Thus, we can form two-qubit gates by taking the tensor product of single-qubit gates. However, these gates still only operate on individual qubits. Qubit gates that change the entanglement of the quantum system are such gates that cannot be written as the tensor product of single-qubit gates. These gates are called *entangling* gates.

One question that arises is how to apply a gate on a subset of the bits that are not consecutive. In Sec. 3.1.2 we showed that if we apply a k_1-bit gate to the first k_1 classical bits, and a k_2-bit gate to the next k_2 classical bits, and so on, then we need to take the tensor product between these gates and apply the resulting matrix on the state vector. Now, suppose that we have an n-qubit system and apply a two-qubit gate that acts on two qubits that are not next to each other. In order to see how we can achieve this let us consider a simple example. Ultimately, we would like to come up with a procedure for applying multi-qubit gates to arbitrary quantum sates. However, in order to make the discussion more comprehensible, in what follows we will consider a product quantum state.

As an example, consider the following 3-qubit product state given in Fig. 3.2. We apply the following quantum gate U to the state $|\psi\rangle$ acting on qubits A and C:

$$U = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}_{AC},$$
$$|\psi\rangle = |0_A 0_B 1_C\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}_A \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}_B \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix}_C.$$

What we do is permute the qubits, so that the first and the third qubits are next to each other, apply the matrix, and then transform back the result undoing the first permutation. But how exactly is the state vector modified when permuting two qubits? To see this, let us compare the state vectors of the original state $|\psi\rangle$ and the permuted state $|\psi'\rangle$. We reshape the original state vector of the system and the state vector of the permuted system into tensors of shape $2 \times 2 \times 2$. Comparing the $2 \times 2 \times 2$ tensors will allow us to find a pattern that corresponds to swapping two qubits:

$$|\psi\rangle = |0_A 0_B 1_C\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}_A \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}_B \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix}_C = \begin{bmatrix} 1_A \times 1_B \times 0_C \\ 1_A \times 1_B \times 1_C \\ 1_A \times 0_B \times 0_C \\ 1_A \times 0_B \times 1_C \\ 0_A \times 1_B \times 0_C \\ 0_A \times 1_B \times 1_C \\ 0_A \times 0_B \times 0_C \\ 0_A \times 0_B \times 1_C \end{bmatrix},$$

$$|\psi\rangle = \begin{array}{|cc|cc|} \hline & & 0_A \times 1_B \times 0_C & 0_A \times 0_B \times 0_C \\ & & 0_A \times 1_B \times 1_C & 0_A \times 0_B \times 1_C \\ \hline 1_A \times 1_B \times 0_C & 1_A \times 0_B \times 0_C & & \\ 1_A \times 1_B \times 1_C & 1_A \times 0_B \times 1_C & & \\ \hline \end{array}$$

Figure 3.3: Tensor form of quantum state $|\psi\rangle$

$$|\psi'\rangle = |0_B 0_A 1_C\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}_B \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}_A \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix}_C = \begin{bmatrix} 1_B \times 1_A \times 0_C \\ 1_B \times 1_A \times 1_C \\ 1_B \times 0_A \times 0_C \\ 1_B \times 0_A \times 1_C \\ 0_B \times 1_A \times 0_C \\ 0_B \times 1_A \times 1_C \\ 0_B \times 0_A \times 0_C \\ 0_B \times 0_A \times 1_C \end{bmatrix}.$$

$$|\psi'\rangle = \begin{array}{|cc|cc|} \hline & & 0_B \times 1_A \times 0_C & 0_B \times 0_A \times 0_C \\ & & 0_B \times 1_A \times 1_C & 0_B \times 0_A \times 1_C \\ \hline 1_B \times 1_A \times 0_C & 1_B \times 0_A \times 0_C & & \\ 1_B \times 1_A \times 1_C & 1_B \times 0_A \times 1_C & & \\ \hline \end{array}$$

Figure 3.4: Tensor form of quantum state $|\psi'\rangle$

Looking at the two states we see that swapping qubits A and B actually corresponds to transposing the horizontal axis with the depth axis. Indexing the axes we start from the outermost one moving inwards. The depth axis chains two 2×2 matrices, thus, this is the outermost axis and is indexed as axis 0. Next, the horizontal axis chains two 2-dimensional vectors, and thus, is indexed as axis 1. Finally, the vertical axis chains scalars

and is indexed axis 2. Swapping qubit A with qubit B can be described as transposing axis 0 with axis 1.

In general, for an n-qubit system, if we want to swap qubit i with qubit j, what we need to do is reshape the state vector into a $\underbrace{2 \times 2 \times ... \times 2}_{\text{n times}}$ tensor, transpose axis i with axis j, and reshape the tensor back again into a 2^n-dimensional vector.

In this work use only two-qubit quantum gates. Note that the set of all two-qubit gates forms a set of *universal quantum gates*. That is, any operation possible on a quantum computer can be reproduced as a finite sequence of gates from the set (see [34]).

3.5 Density operators

3.5.1 Pure and mixed states

Recall from Eq. (3.2) that a quantum system state vector is given by:

$$|\psi\rangle = \sum_i c_i |e_i\rangle .$$

We have what is called a *pure state*. This means that we can say with statistical certainty that our system is in state $|\psi\rangle$, i.e., we have a complete knowledge of the system state. In practice, however, quantum systems are hard to isolate, and hence often are entangled with their environment. As a consequence, when we get our system, we cannot be certain about its state. Instead, we have to describe the system as a probabilistic combination of all the pure states it could possibly have been prepared in:

$$|\psi\rangle = \begin{cases} |\psi_0\rangle , & \text{with probability} \quad p_0 \\ |\psi_1\rangle , & \text{with probability} \quad p_1 \\ \dots & \\ |\psi_k\rangle , & \text{with probability} \quad p_k \end{cases} ,$$

where $\{|\psi_i\rangle\}$ are pure states and $\{p_i\}$ are the probabilities associated with each state. The sum of the probabilities of all possible states is equal to unity. The state $|\psi\rangle$ is written as $\{p_i, |\psi_i\rangle\}$ and is called an *ensemble of pure states* or a *mixture of pure states*. More rigorously: a *mixed quantum state* is a statistical ensemble of pure states.

As we discussed above, one situation where mixed states could arise is when the preparation of the system is not fully known, and thus one must deal with a statistical ensemble of possible preparations. Another situation is when one wants to describe a quantum system which is entangled with another system. As mentioned in Sec. 3.3, describing an entangled system by listing the state vectors of each of the qubits is impossible. We will shortly see that the state of each qubit can actually be described as an ensemble of pure states.

Consider the Bell state from Eq. (3.8):

$$|\psi_{\text{Bell}}\rangle = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle .$$

Recall that the result of measuring the first qubit in the computational basis has a $\frac{1}{2}$ chance of being 0 and $\frac{1}{2}$ chance of being 1. Recall also that, for this Bell state, measuring the first qubit results in collapsing the second qubit as well. Suppose we perform a measurement on the first qubit, but we do not know the result. The state of the second qubit fully depends on the measurement of the first qubit, and has a $\frac{1}{2}$ chance of being $|0\rangle$ and $\frac{1}{2}$ chance of being $|1\rangle$. Note that this does not mean that our second qubit is in a superposition of the two states $|0\rangle$ and $|1\rangle$. Being in a superposition means that the particle is in both states simultaneously, and only upon measurement collapses into one of the states. Rather, our particle has collapsed into one of the two states, but we do not know which one. Thus, the state of the second qubit can be described as a probabilistic mixture of the two states $|0\rangle$ and $|1\rangle$:

$$|q_2\rangle = \begin{cases} |0\rangle\,, & \text{with probability } \frac{1}{2} \\ |1\rangle\,, & \text{with probability } \frac{1}{2} \end{cases}. \tag{3.13}$$

3.5.2 The density matrix

Until now the state of a quantum system was described using a state vector. Alternatively, the state can also be described using the *density matrix*. This approach is equivalent to the state vector approach; however it is much more convenient to describe mixed quantum states using their density matrix. The density matrix ρ for a mixed state $\{p_i, |\psi_i\rangle\}$ is given by:

$$\rho = \sum_i p_i \, |\psi_i\rangle \, \langle\psi_i| \,. \tag{3.14}$$

Using Eq. (3.14) we can see that the density matrix of a pure state is given by:

$$\rho = \sum_i p_i \, |\psi_i\rangle \, \langle\psi_i| = |\psi\rangle \, \langle\psi| \,. \tag{3.15}$$

This equation allows us to give another definition a pure state. A pure state is any state that has a density matrix of rank 1, i.e., $\rho = |\psi\rangle \, \langle\psi|$.

In order for a matrix to be a valid density matrix it has to be a positive semi-definite hermitian matrix with a unit trace:

$$\rho^\dagger = \rho, \tag{3.16}$$
$$\rho v = \lambda v \iff 0 \leq \lambda, \tag{3.17}$$
$$\mathrm{tr}(\rho) = 1, \tag{3.18}$$

where v is an eigenvalue of the matrix ρ.

Note that from Eqs. (3.2) and (3.14) we have:

$$\rho_{uv} = \sum_i p_i c_u c_v^\dagger = \left(\sum_i p_i c_u^\dagger c_v \right)^\dagger = \rho_{vu}^\dagger,$$

thus the density matrix is hermitian.

Additionally, for any quantum state $|\phi\rangle$ we have:

$$\langle\phi| \, \rho \, |\phi\rangle = \sum_i p_i \, \langle\phi|\psi_i\rangle \, \langle\psi_i|\phi\rangle = \sum_i p_i \Big(\, \langle\phi|\psi_i\rangle \, \Big)^2 \geq 0.$$

And for the trace of the density matrix we have:

$$\mathrm{tr}(\rho) = \sum_j \rho_{jj} = \sum_i \sum_j p_i |c_i^{(j)}|^2 = 1.$$

To see that the density matrix formulation is equivalent to the state vector approach, we can show that all quantum mechanical formulas and equations can be reformulated in terms of the density matrix:

- Using Eq. (3.15) we can express the density matrix of the quantum system in the computational basis:

$$\begin{aligned} |\psi\rangle &= \sum_i c_i |e_i\rangle , \\ \rho = |\psi\rangle \langle\psi| &= \sum_{ij} c_i c_j^\dagger |e_i\rangle \langle e_j| . \end{aligned} \tag{3.19}$$

- The density matrix of a product state is arrived at by simply taking the tensor product between the density matrices of the individual states:

$$\rho_{|\psi\phi\rangle} = \rho_{|\psi\rangle} \otimes \rho_{|\phi\rangle}.$$

- Suppose we would like to apply a gate U to a mixed quantum state $\{p_i, |\psi_i\rangle\}$. The evolution of the density matrix can be described as:

$$\rho = \sum_i p_i |\psi_i\rangle \langle\psi_i| \xrightarrow{U} \sum_i p_i U |\psi_i\rangle \langle\psi_i| U^\dagger = U\rho U^\dagger. \tag{3.20}$$

- Recall that the probability of observing a given eigenvalue λ_j of a measurement operator is given by the square of the absolute value of the transition amplitude between our state vector and the eigenvector $|e_j\rangle$ of the measurement operator corresponding to that eigenvalue. Thus, for a pure state $|\psi\rangle$ we have:

$$\begin{aligned} A &= \langle e_j|\psi\rangle \\ p(\lambda_j) &= |A|^2 = |\langle e_j|\psi\rangle|^2 \\ &= \langle e_j|\psi\rangle \langle e_j|\psi\rangle^\dagger \\ &= \langle e_j|\psi\rangle \langle\psi|e_j\rangle = \langle e_j| \rho |e_j\rangle . \end{aligned} \tag{3.21}$$

  This formula generalizes as well to mixed states described using their density matrix:

$$\begin{aligned} p(\lambda_j) &= \sum_i p_i |\langle e_j|\psi_i\rangle|^2 \\ &= \sum_i p_i \langle e_j|\psi_i\rangle \langle\psi_i|e_j\rangle \\ &= \langle e_j| \sum_i p_i |\psi_i\rangle \langle\psi_i|e_j\rangle \\ &= \langle e_j| \rho |e_j\rangle . \end{aligned} \tag{3.22}$$

Looking at Eqs. (3.19), (3.21) and (3.22) we can see that measuring a state $|\psi\rangle$ in the standard basis results in λ_i with probability $p(\lambda_i) = \rho_{ii} = \langle e_i|\rho|e_i\rangle$. Thus, the diagonal entries of the density matrix give the probabilities of our system to collapse onto each of the standard basis vectors.

3.5.3 Partial trace

As mentioned earlier, the subsystem of a composite quantum system can be described using the density matrix formalism. In this case the matrix describing the subsystem is called a *reduced density matrix* or a *reduced density operator*. Suppose we have a composite system AB described by the density matrix ρ_{AB} and we would like to arrive at a description for the subsystem A, that is the reduced density matrix ρ_A.

In order to do this we need to use a mapping called the *partial trace* over subsystem B:

$$\rho_A = \mathrm{tr}_B(\rho_{AB}). \tag{3.23}$$

The definition of the partial trace is given by the following formula:

$$\mathrm{tr}_B(\rho_{AB}) = \sum_i \Big(\mathrm{I}_A \otimes \langle i|_B \Big) \rho_{AB} \Big(\mathrm{I}_A \otimes |i\rangle_B \Big), \tag{3.24}$$

where $\{|i\rangle_B\}_i$ is any orthonormal basis for the Hilbert space of subsystem B.

From the definition in Eq. (3.24) we can deduce the following:

- for any two vectors $|a_1\rangle$ and $|a_2\rangle$ from the state space of system A, and any two vectors $|b_1\rangle$ and $|b_2\rangle$ from the state space of system B we have:

$$\begin{aligned} \mathrm{tr}_B(|a_1 b_1\rangle \langle a_2 b_2|) &= |a_1\rangle \langle a_2| \, \mathrm{tr}(|b_1\rangle \langle b_2|) \\ &= |a_1\rangle \langle a_2| \, \langle b_1|b_2\rangle \, ; \end{aligned} \tag{3.25}$$

- the partial trace is linear, that is for any two systems ρ_{A_1} and ρ_{A_2} we have:

$$\mathrm{tr}_B(\rho_{A_1} + \rho_{A_2}) = \mathrm{tr}_B(\rho_{A_1}) + \mathrm{tr}_B(\rho_{A_2}). \tag{3.26}$$

As an example, let us consider again the Bell state as a quantum system composed of two separate particles (subsystems):

$$|\psi\rangle_{\text{Bell}} = \frac{1}{\sqrt{2}} |00\rangle + \frac{1}{\sqrt{2}} |11\rangle$$

In Eq. (3.13) we tried to reason what would be a possible description for one of the qubits of the Bell state. We did this by pretending to measure the first qubit, but without looking at the result of the measurement. The partial trace operator does exactly the same thing. Intuitively, we would like our reduced density matrix ρ_A to represent a mixed quantum state that would reflect the outcome of all different possible measurements of the subsystem B with their respective probabilities of occurring.

Calculating the reduced density matrix for qubit 2 we need to trace out qubit 1 using the

partial trace:

$$\begin{aligned}\rho_{\text{Bell}} &= |\psi\rangle_{\text{Bell}} \langle\psi|_{\text{Bell}} = \frac{1}{\sqrt{2}}\Big(|00\rangle + |11\rangle \Big)\frac{1}{\sqrt{2}}\Big(\langle 00| + \langle 11| \Big) \\ &= \frac{|00\rangle \langle 00| + |00\rangle \langle 11| + |11\rangle \langle 00| + |11\rangle \langle 11|}{2},\end{aligned}$$

$$\begin{aligned}\rho_2 &= \text{tr}_1(\rho_{\text{Bell}}) \\ &= \frac{1}{2}\Big[\text{tr}_1\Big(|00\rangle \langle 00| \Big) + \text{tr}_1\Big(|00\rangle \langle 11| \Big) + \text{tr}_1\Big(|11\rangle \langle 00| \Big) + \text{tr}_1\Big(|11\rangle \langle 11| \Big)\Big] \\ &= \frac{1}{2} |0\rangle \langle 0| + \frac{1}{2} |1\rangle \langle 1| .\end{aligned}$$

We can see that using the partial trace we arrive at the same expression for the second qubit as in Eq. (3.13).

In order to understand why the reduced density matrix of a subsystem is given by the partial trace, let us see what requirements have to be satisfied by a correctly formulated reduced density matrix. Consider a composite system ρ_{AB}, and let ρ_A be the reduced density matrix describing the subsystem A. Suppose we would like to measure some observable M on subsystem A (e.g., the spins of all the qubits in system A). Thus, M is the observable operator measured on the subsystem A. Let M' denote the corresponding observable operator for the same measurement on the composite system ρ_{AB}. We then have:

$$M' = M \otimes I_B.$$

If ρ^A is a correct description of our subsystem, then performing the two measurements must produce the same measurement statistics. We therefore have:

$$\text{tr}\Big(M\rho_A\Big) = \text{tr}\Big(\big(M \otimes I_B\big)\rho_{AB}\Big).$$

It turns out that this equation is satisfied by $\rho^A = \text{tr}_B\Big(\rho_{AB}\Big)$, and the partial trace is the unique function having this property.

3.6 Quantifying quantum entanglement

To quantify the entanglement of a quantum state we will use the notion of entropy. Entropy is a key concept of information theory. It measures the level of uncertainty in the state of a physical system.

Given a pure state $\rho_{AB} = |\psi\rangle_{AB} \langle\psi|_{AB}$, we would like to measure the amount of entanglement between the two subsystems A and B. Note that, as we showed in Sec. 3.5, each of the subsystems is an ensemble of quantum states described by a reduced density matrix. Thus, we could measure the level of uncertainty that is present in each of the subsystems; that is, we measure the entropy of the ensemble state.

Given a random variable X, the entropy of X measures the amount of uncertainty about X before we learn its value. Alternatively, we could say that the entropy of X quantifies

how much information we gain, on average, when we learn the value of X. The entropy of a random variable is defined to be a function of the probabilities of the different possible values the random variable takes, i.e., it is a function of a probability distribution $p_1, p_2, ..., p_k$:

$$H(X) = -\sum_x p_x \log(p_x),$$

where x are all the possible values for the random variable X, and $p_x = \mathrm{P}(X = x)$ is the probability of X having value x.

Entropy measures the uncertainty associated with a classical probability distribution. Extending the definition of entropy from probability distributions to matrices will allow us to calculate the entropy of a quantum state described by a density matrix:

$$S(\rho) = -\mathrm{tr}(\rho \log \rho). \tag{3.27}$$

Given a pure state $\rho_{AB} = |\psi\rangle_{AB} \langle\psi|_{AB}$, the degree of entanglement between A and B is given by calculating the entropy of the subsystem A (or B). Using the formula from Eq. (3.23) we obtain the density matrix for the subsystem A, and using formula Eq. (3.27) we can calculate the entropy of that system:

$$S_{\mathrm{ent}} = -\mathrm{tr}(\rho_A \log \rho_A) = -\mathrm{tr}(\rho_B \log \rho_B).$$

Chapter 4

The multi-qubit system disentangling problem

In this chapter we introduce the problem of disentangling a quantum system. A definition of disentanglement is given in Sec. 4.1. In Sec. 4.2 we show a practical implementation of how quantum gates are applied. In Sec. 4.3 we give a procedure for computing a quantum gate that applies the maximum local reduction to the entanglement. Quantum systems of 2-qubit and 3-qubit quantum states have exact solutions, which are shown in Sec. 4.4. Finally, in Sec. 4.5 we introduce the problem that we will be solving in this work.

4.1 Fully separable system

The problem that we will try to solve in this work is to come up with an algorithm for reducing the entanglement of a quantum state, i.e., we would like to **disentangle** a quantum state.

In Sec. 3.6 we saw that given a pure quantum state $|\psi\rangle$ we can decompose the state into two subsystems A and B, and we can calculate the entanglement between these two systems. A different decomposition of the state into two different subsystems C and D would yield a different value for the degree of entanglement between these two subsystems.

If for a given decomposition the entanglement is zero, then our initial quantum state can be represented as a product state between the states of the two subsystems:

$$|\psi\rangle_{AB} = |\psi\rangle_A \otimes |\psi\rangle_B \iff S(\rho_A) = S(\rho_B) = 0.$$

As an example, consider a 4-qubit state, and let subsystem $A = \{1, 2\}$ consist of qubits 1 and 2, and subsystem $B = \{3, 4\}$ consist of qubits 3 and 4. Suppose that the entanglement between subsystems A and B is equal to zero, i.e., $S(\rho_A) = 0$. This means then, that our initial 4-qubit state can be separated into two 2-qubit states.

As another example, let $A = \{1\}$, $B = \{2, 3, 4\}$, and $S(\rho_A) = 0$. This means that qubit 1 is not entangled with the other qubits and can be separated from the rest of the system.

Continuing in this fashion, we can define a **fully separable state** to be such quantum state that for an arbitrary decomposition into two subsystems A and B the entanglement

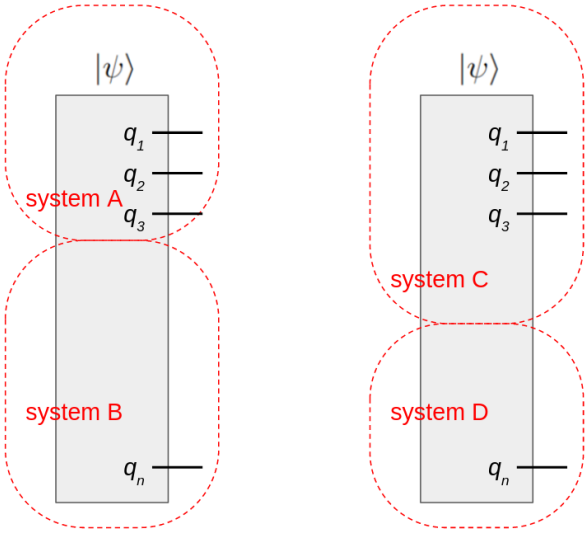


Figure 4.1: Decomposition of a quantum system into subsystems. There are multiple ways to decompose a quantum system into sub-systems. If the entanglement between subsystems A and B is zero, then the quantum system is in a product state between these two systems.

between these two systems is zero. Or, put in other words, any group of qubits can be separated from the rest of the system. Another way to describe a fully separable state is to say that an n-qubit state is fully separable if and only if it is a product state of n one-qubit states:

$$|\psi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle \otimes ... \otimes |\phi_n\rangle \,,$$

where $|\phi_i\rangle$ is a single-qubit state $\forall i$.

Thus, in order for an n-qubit state to be fully separable, we need to have the entanglement between the subsystems $A = \{i\}$ and $B = \{1, 2, ..., i-1, i+1, ..., n\}$ to be equal to zero, for all $i = 1, 2, ..., n$. In order to have a measure for how close a quantum state to a fully separable state is we define the *average entanglement* of a quantum state:

$$S_{\text{avg}}(|\psi\rangle) = -\frac{1}{N}\sum_{i=1}^{N} \text{tr}(\rho_{\{i\}} \log(\rho_{\{i\}}). \tag{4.1}$$

It should be noted that as the number of qubits in the system grows, perfectly separating a qubit from the system becomes notoriously hard. Thus, when we talk about separating a qubit from the system, we will be aiming at reducing the entanglement of that qubit with the rest of the system below a given threshold ϵ. We will consider a quantum state to be disentangled if all qubits are separated up to ϵ from the rest of the system. For the purposes of this work we set the threshold value to $\epsilon = 10^{-3}$.

The task at hand is to come up with an algorithm that for an arbitrary quantum state can produce a sequence of quantum gates to be applied to that state such that the resulting quantum state is disentangled.

4.2 Applying quantum gates

In Sec. 3.4 we mentioned that in order for a gate to be able to modify the degree of entanglement between qubits in a quantum state, this gate must be a multi-qubit gate that is not a tensor product of single-qubit gates. In this work we will only consider applying two-qubit quantum gates to modify entanglement. In fact it can be shown that any multi-qubit gate can be simulated by applying multiple two-qubit gates.

Recall, that applying a quantum gate to a state is described by performing matrix multiplication between the quantum gate and the state vector:

$$|\psi\rangle \xrightarrow{U} |\psi'\rangle\,, \quad U\,|\psi\rangle = |\psi'\rangle\,.$$

Working with an n-qubit state we can see that this results in a vector-matrix multiplication of size 2^n. The complexity of performing such an operation is $O(2^{2n})$. Note, however, that we want to apply a two-qubit gate, say $V_{4\times4}$. Thus, the gate U will be arrived at by taking the tensor product between V and the identity operator for the rest of the system:

$$U = V \otimes I_{2^{n-2}}.$$

We can take advantage of this and re-arrange the matrix multiplication in order to reduce the cost of computing. Let

$$V = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix}.$$

Then, after expressing the matrix U in block from, we arrive at the following for the vector-matrix multiplication:

$$U\,|\psi\rangle = \begin{bmatrix} aI_{2^{n-2}} & bI_{2^{n-2}} & cI_{2^{n-2}} & dI_{2^{n-2}} \\ eI_{2^{n-2}} & fI_{2^{n-2}} & gI_{2^{n-2}} & hI_{2^{n-2}} \\ iI_{2^{n-2}} & jI_{2^{n-2}} & kI_{2^{n-2}} & lI_{2^{n-2}} \\ mI_{2^{n-2}} & nI_{2^{n-2}} & oI_{2^{n-2}} & pI_{2^{n-2}} \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ q_{2^n-2} \\ q_{2^n-1} \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} Q^1_{2^{n-2}} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} Q^2_{2^{n-2}} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} Q^3_{2^{n-2}} \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} Q^4_{2^{n-2}} \end{matrix}.$$

If we imagine decomposing the state vector $|\psi\rangle$ into 4 blocks of size 2^{n-2} we arrive at:

$$U\,|\psi\rangle = \begin{bmatrix} aI_{2^{n-2}} & bI_{2^{n-2}} & cI_{2^{n-2}} & dI_{2^{n-2}} \\ eI_{2^{n-2}} & fI_{2^{n-2}} & gI_{2^{n-2}} & hI_{2^{n-2}} \\ iI_{2^{n-2}} & jI_{2^{n-2}} & kI_{2^{n-2}} & lI_{2^{n-2}} \\ mI_{2^{n-2}} & nI_{2^{n-2}} & oI_{2^{n-2}} & pI_{2^{n-2}} \end{bmatrix} \begin{bmatrix} [Q_1] \\ [Q_2] \\ [Q_3] \\ [Q_4] \end{bmatrix} = \begin{bmatrix} [aQ_1 + bQ_2 + cQ_3 + dQ_4] \\ [eQ_1 + fQ_2 + gQ_3 + hQ_4] \\ [iQ_1 + jQ_2 + kQ_3 + lQ_4] \\ [mQ_1 + nQ_2 + oQ_3 + pQ_4] \end{bmatrix}_{2^n}. \tag{4.2}$$

Note that the final result is a vector represented in block form. However, looking at Eq. (4.2), we can see that if we simply reshape the state vector into a matrix of shape

$4 \times 2^{2^n-2}$, then we can perform a much cheaper matrix-matrix multiplication. Finally, the resulting matrix has to be reshaped back into a 2^n-dimensional vector:

$$|\psi\rangle \xrightarrow{\text{reshape}} \begin{bmatrix} q_0 & \cdots & q_{2^{n-2}-1} \\ q_{2^{n-2}} & \cdots & q_{2^{n-1}-1} \\ q_{2^{n-1}} & \cdots & q_{2^{n-1}+2^{n-2}-1} \\ q_{2^{n-1}+2^{n-2}} & \cdots & q_{2^n-1} \end{bmatrix} \begin{matrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{matrix} = \Psi, \tag{4.3}$$

$$V\Psi = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{bmatrix} = \begin{bmatrix} aQ_1 + bQ_2 + cQ_3 + dQ_4 \\ eQ_1 + fQ_2 + gQ_3 + hQ_4 \\ iQ_1 + jQ_2 + kQ_3 + lQ_4 \\ mQ_1 + nQ_2 + oQ_3 + pQ_4 \end{bmatrix}_{4\times 2^{n-2}},$$

$$V\Psi \xrightarrow{\text{reshape}} U|\psi\rangle.$$

Simulating the application of quantum gates using the procedure from Eq. (4.3) results in sixteen **scalar-vector** multiplications of size 2^{n-2}. Thus, the complexity of performing this operation is $O(16 \times 2^{n-2}) \in O(2^n)$. The complexity class is still exponential, which is normal considering the fact that we are trying to simulate a quantum system. However, the reduction from $O(2^{2n})$ to $O(2^n)$ is enormous and results in a massive speed up when applied in practice.

4.3 Locally optimal gates

Since we are only concerned with applying two-qubit gates, at each step our algorithm needs to produce the indices of the qubits, $\{i, j\}$, to which a gate is applied. We need to specify the quantum gate itself as well.

Applying a quantum gate to qubits i and j of a quantum state $|\psi\rangle$ can modify the entanglement of only these two qubits, and the amount with which the entanglement is modified depends on the specific gate that is selected. Given that the goal is to reduce the entanglement to zero, it seems natural to try to find a gate that applies a maximum reduction to the entanglement of each of the two qubits.

Starting with the reduced density matrix for the subsystem $A = \{i, j\}$, ρ_A (Eq. (3.23)), we show an algorithm for arriving at a gate that is guaranteed to reduce the entanglement of at least one of the qubits.

First, let us consider what happens when a two-qubit quantum gate U is applied to our quantum state $|\psi\rangle$:

$$|\psi'\rangle = (U \otimes I_B)|\psi\rangle,$$

where $B = \{1, 2, ...n\} \backslash \{i, j\}$ is the subsystem of all the other qubits except i and j.

Then, using Eq. (3.24) for the reduced density matrix of subsystem A of the new quantum

state we get:

$$\begin{aligned}
\rho' &= (U \otimes I_B) \left|\psi\right\rangle \left\langle\psi\right| (U \otimes I_B)^\dagger \\
\rho'_A &= \mathrm{tr}_B(\rho') \\
&= \sum_i \left(\mathrm{I}_A \otimes \left\langle i \right|_B \right) \rho' \left(\mathrm{I}_A \otimes \left| i \right\rangle_B \right) \\
&= \sum_i \left(\mathrm{I}_A \otimes \left\langle i \right|_B \right) (U \otimes I_B) \rho (U \otimes I_B) \dagger \left(\mathrm{I}_A \otimes \left| i \right\rangle_B \right) \\
&= \sum_i U \left(\mathrm{I}_A \otimes \left\langle i \right|_B \right) \rho \left(\mathrm{I}_A \otimes \left| i \right\rangle_B \right) U^\dagger \\
&= U \rho_A U^\dagger
\end{aligned}$$

Thus, when considering the evolution of the reduced density matrix ρ_A, we can see that applying a quantum gate U to qubits i and j will transform ρ_A into $U\rho_A U^\dagger$.

What we did is to come up with a quantum gate U such that it reduces the entanglement between the qubit i and the rest of the system $B' = \{1, 2, ..., i-1, i+1, ..., n\}$. To calculate the entanglement between the systems $A = \{i\}$ and B' we need to trace out qubit j from ρ_A:

$$\begin{aligned}
\rho_i &= \mathrm{tr}_j(\rho_A), \\
S(\rho_i) &= -\sum_k \eta_k \log \eta_k,
\end{aligned}$$

where $\{\eta_k\}$ are the eigenvalues of the reduced density matrix ρ_i.

Following Eq. (3.18) the sum of the eigenvalues is equal to unity. We can see that having the eigenvalues approximately equal provides for the highest entanglement between the systems A and B', while having all the mass concentrated in one eigenvalue provides for the lowest entanglement.

Note that the matrix ρ_A is diagonalizable and can be written as:

$$D = P \rho_A P^\dagger,$$

where:

$$D = \begin{bmatrix} \lambda_0 & 0 & 0 & 0 \\ 0 & \lambda_1 & 0 & 0 \\ 0 & 0 & \lambda_2 & 0 \\ 0 & 0 & 0 & \lambda_3 \end{bmatrix},$$

is a diagonal matrix constructed from the eigenvalues λ_i of ρ_A, and P is the matrix composed of the eigenvectors of ρ_A.

The reduced density matrix for the qubit i is then:

$$\rho_i = \mathrm{tr}_j(D) = \begin{bmatrix} \lambda_0 + \lambda_1 & 0 \\ 0 & \lambda_2 + \lambda_3 \end{bmatrix},$$

and the entanglement is computed as:

$$S(\rho_i) = -(\lambda_0 + \lambda_1) \log(\lambda_0 + \lambda_1) - (\lambda_0 + \lambda_1) \log(\lambda_0 + \lambda_1).$$

The eigenvalues λ_0 and λ_1 effectively represent qubit i, while λ_2 and λ_3 – qubit j. We modify the eigenvalues in such a way that the mass is shifted toward qubit i effectively reducing the entanglement between systems A and B'. The modification is applied in the form of a swap matrix that arranges the eigenvalues in ascending order: We can see that having the shifting the mass

$$D^* = SDS^\dagger = \begin{bmatrix} \lambda_i & 0 & 0 & 0 \\ 0 & \lambda_j & 0 & 0 \\ 0 & 0 & \lambda_k & 0 \\ 0 & 0 & 0 & \lambda_l \end{bmatrix}, \qquad \lambda_i \leq \lambda_j \leq \lambda_k \leq \lambda_l.$$

The gate that is applied to qubits i and j will be:

$$U = PS, \tag{4.4}$$

where P is the matrix composed of the eigenvectors of the reduced density matrix ρ_A, and S is a swap matrix that arranges the eigenvalues in ascending order. This is a valid quantum gate since it is a product of two Hermitian matrices.

The effect of this gate will not modify the eigenvalues of the reduced density matrix ρ_A; instead, it will simply ensure that they are arranged in ascending order. This property will ensure that the entanglement $S(\rho_i)$ between qubit i and the rest of the quantum system is decreased.

Note that for the entanglement $S(\rho_j) = S(\mathrm{tr}_i(\rho_A))$ between qubit j and the rest of the quantum system we cannot guarantee that it will be reduced. Instead of the swap matrix S, there are different Hermitian matrices that could be applied that shift the mass towards one or the other qubits. During our experiments, however, we found that the matrix S applies the maximum amount of reduction to the joint quantity $S(\rho_i)+S(\rho_j)$. We have no proof that the gate given in Eq. (4.4) is the gate that applies the maximum reduction to the entanglement between qubit i and subsystem B'. However, based on the experiments that we performed we are inclined to make the following:

Conjecture 1 *The two-qubit quantum gate given in Eq. (4.4) is locally optimal in a sense that when applied to qubits $\{i, j\}$ of an n-qubit quantum state $|\psi\rangle$ it will apply the maximum reduction of entanglement between qubit $\{i\}$ and the rest of the system $B' = \{1, 2, ..., i-1, i+1, ..., n\}$.*

Thus, in this work the gate given in Eq. (4.4) is used.

As mentioned in the previous section, the task is to come up with a sequence of two-qubit quantum gates to be applied such that the system results in a disentangled quantum state. We now have a procedure for determining a locally optimal two-qubit quantum gate $U(|\psi\rangle, i, j)$ to be applied to a given quantum state $|\psi\rangle$ given the pair of qubits to which we want to apply it.

Given a quantum state $|\psi\rangle$ and a pair of qubits $\{i, j\}$ to which to apply a quantum gate we now have a procedure for determining a locally optimal gate $U(|\psi\rangle, i, j)$. This means that all we have to do at each step is to choose a pair of qubits to which the gate should be applied. Thus, for an n-qubit quantum state, at each step we have to choose one from $V_n^2 = n(n-1)$ different pairs of qubits, and after that apply the resulting optimal gate to that pair. We are considering all possible variations because choosing the pairs $\{i, j\}$

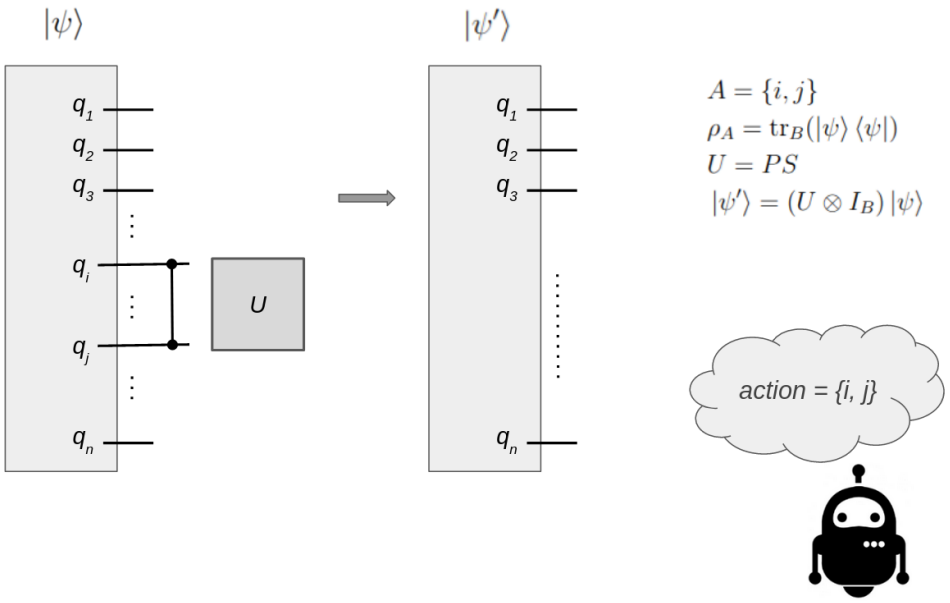


Figure 4.2: Action selection and action application. After the agent selects the two qubits $\{i, j\}$, we calculate the locally optimal quantum gate using Eq. (4.4) and apply it to those qubits.

and $\{j, i\}$ are two different actions. The ordering effectively indicates towards which of the two qubits, i or j, the entanglement is shifted.

Note that the procedure for calculating the optimization gate depends entirely on the fact that there exists an ordering between the eigenvalues $\lambda_0, ..., \lambda_3$ of ρ_A. It may be the case, however, that we arrive at a maximally mixed state and the reduced density matrix is proportional to the identity matrix, $\rho_A \propto I$. In this case applying the quantum gate calculated using Eq. (4.4) will have no effect on the entanglement of the quantum sate. Nevertheless, this would mean that there exists a different a qubits, say $\{k, l\}$, for which the mass of the eigenvalues is maximally shifted towards one of the qubits. This effect is due to the so-called *Monogamy of entanglement*.

4.4 Exact solutions

To provide an algorithm for disentangling a quantum state, at every step of the sequence we will apply the quantum gate given by Eq. (4.4) to the selected qubits i and j.

4.4.1 Two-qubit states

Before considering how we could solve multi-qubit systems with a large number of qubits, let us first examine the most basic case, which is a quantum state of two qubits. There is only one action we can take and that is to apply the optimal gate to qubits 0 and 1.

The formula for the locally optimal gate in Sec. 4.3 was derived by considering an arbitrary mixed quantum state with density matrix ρ. However, in this case we have a pure state. This means that the eigenvalues of the density matrix ρ are $\lambda_0 = 1, \lambda_1 = \lambda_2 = \lambda_3 = 0$. For the entanglement between the two qubits after applying the optimization gate we

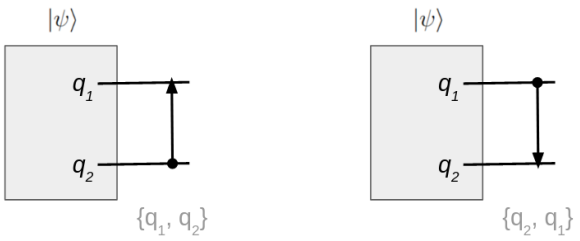


Figure 4.3: Solutions for 2-qubit quantum states.

have:

$$S(\rho_1) = 0.$$

From here we can see that a two-qubit quantum state can actually be disentangled with the help of a single quantum gate. Thus, there are two different solutions to the problem in this case:

4.4.2 Three-qubit states

In the case of a three-qubit state we need to apply the locally optimal quantum gate to a pair of qubits, say i and j. Thus, we consider the reduced density matrix ρ_A that describes the subsystem of qubits $A = \{i, j\}$. Since we are tracing out only one qubit, subsystem A will be an ensemble of two two-qubit states:

$$|\psi\rangle_A = \begin{cases} |\psi_0\rangle\,, & \text{with prob. } p_0 \text{ equal to the prob. of measuring qubit } k \text{ as } 0; \\ |\psi_1\rangle\,, & \text{with prob. } p_1 \text{ equal to the prob. of measuring qubit } k \text{ as } 1. \end{cases}$$

Hence, for the reduced density matrix we have:

$$\rho_A = p_0 \left|\psi_0\right\rangle \left\langle\psi_0\right| + p_1 \left|\psi_1\right\rangle \left\langle\psi_1\right|.$$

Thus, our reduced density matrix has two non-zero and two zero eigenvalues. Once again, for the entanglement between the two qubits after applying the gate we have:

$$S(\rho_i) = 0.$$

This means that applying a single quantum gate would separate qubit i from qubit j. However, this actually means that one of the two qubits will be separated from the entire system. To see why this is true let us assume that after the gate is applied both qubits i and j are still entangled to qubit k. Then our quantum state $|\psi\rangle$ is not a product state. But if we trace-out qubit k we will arrive at a product state, since qubits i and j are disentangled, which is a contradiction.

Following that we can apply the optimization quantum gate for the remaining two qubits and, with this, disentangle the entire quantum state. Thus, we can see that we can disentangle any three-qubit quantum state with applying just two quantum two-qubit gates. It is obvious from the discussion so far that the choice of qubit is irrelevant, which means that there are:

$$V_3^2 \times C_3^2 = \frac{3!}{1!}\frac{3!}{2!1!} = 18,$$

different solutions to the problem. Here V_n^k gives all the k-element variations from an n-element set, and C_n^k gives all the k element combinations from an n-element set.

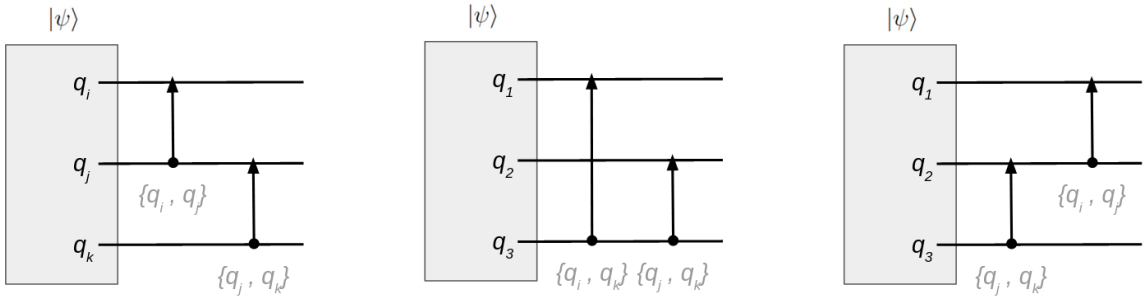


Figure 4.4: Solutions for 3-qubit quantum states. The figure shows only a general pattern for arriving at a solution for 3-qubit quantum states. All solutions to the 3-qubit quantum state are permutations of the solutions given on the figure.

4.5 The optimization problem

As mentioned in Sec. 4.1 the problem that we will try to solve is to come up with an algorithm for disentangling arbitrary quantum states. That is, the algorithm must produce a sequence of quantum gates which disentangles the initial quantum state.

In Sec. 4.3 we proposed a procedure for calculating the matrix representation of the quantum gate to be applied, given the indices of the qubits to which it will be applied. Thus, our algorithm needs to produce a sequence of decisions, where each decision is a pair of indices indicating the two qubits for which a gate should be calculated and applied at the given step.

Note that applying the quantum gate calculated using Eq. (4.4) guarantees that the entanglement in the entire quantum system is reduced. Thus, we could simply try to naively disentangle the system by selecting qubit pairs at random and applying the respective quantum gates. As it turns out (see Sec. 6.1), this will yield a solution to the problem, albeit not a very efficient one. Most of the quantum gates selected following this naive approach, even though reducing the entanglement, are completely unnecessary and can be skipped. Thus, we will be interested in finding a sequence that is (near-)optimal in terms of length.

The problem that we want to solve can be formulated as follows:

Problem 1 *Find an algorithm such that, given an arbitrary quantum state $|\psi\rangle$, the algorithm produces a sequence of actions, reducing the entanglement of every qubit $\{S(\rho_1), S(\rho_2), ..., S(\rho_n)\}$ (Eq. (3.27)) of the state below a given threshold $\epsilon = 10^{-3}$. The actions that are produced are pairs of indices indicating the two qubits to which a locally optimal quantum gate (Eq.(4.4)) should be applied. The sequence produced by the algorithm must be (near-)optimal in terms of its length.*

Chapter 5

Controlling quantum entanglement using deep reinforcement learning

In this chapter we describe how we model the problem in the framework of RL (Reinforcement learning). In Sec. 5.1 we pose the problem as an optimization problem. In Sec. 5.2 we describe how we set up the reinforcement learning environment. An optimization procedure for the agent-environment loop using parallelism is given in Sec. 5.3. And in Sec. 5.4 a definition of an agent is given.

5.1 Optimization using learning

As explained in Sec. 4.3 our algorithm produces a sequence of actions, where each action gives the indices of the two qubits to which we apply the calculated quantum gate using the procedure from Eq. (4.4).

The problem that we try to solve is to optimize the number of actions taken to disentangle a quantum state. Tree search algorithms such as A^* are known to be able to produce an optimal solution to such kind of combinatorial problems; however these algorithms run exponentially slow (w.r.t. the sequence length). Another downside of tree search algorithms is that they are not able to generalize at all. Thus, knowing the solution to one quantum state does not help in any way when searching for the solution for another very similar quantum state. Search algorithms are discussed in Sec. 6.2.

In this thesis we show that we can leverage the generalization capabilities of neural networks. However, in order to train a network using supervised learning we need training data. That is, we need to know the solutions to a finite amount of quantum states, and then train the network on these solutions. In general, we do not have access to training data with optimal solutions. Nevertheless, we can generate near-optimal solutions using a local search algorithm and train a neural network on these solutions. This approach is further discussed in Sec. 6.3.

The results from training using supervised learning show that the model is not generalizing well to out-of-data examples. In addition, due to the sequential nature of the problem, the performance of the model on the disentangling task is poor. Another approach that we take is to treat the problem as a MDP (Markov decision process) and solve it using RL (Reinforcement learning). Using RL is better suited for learning to

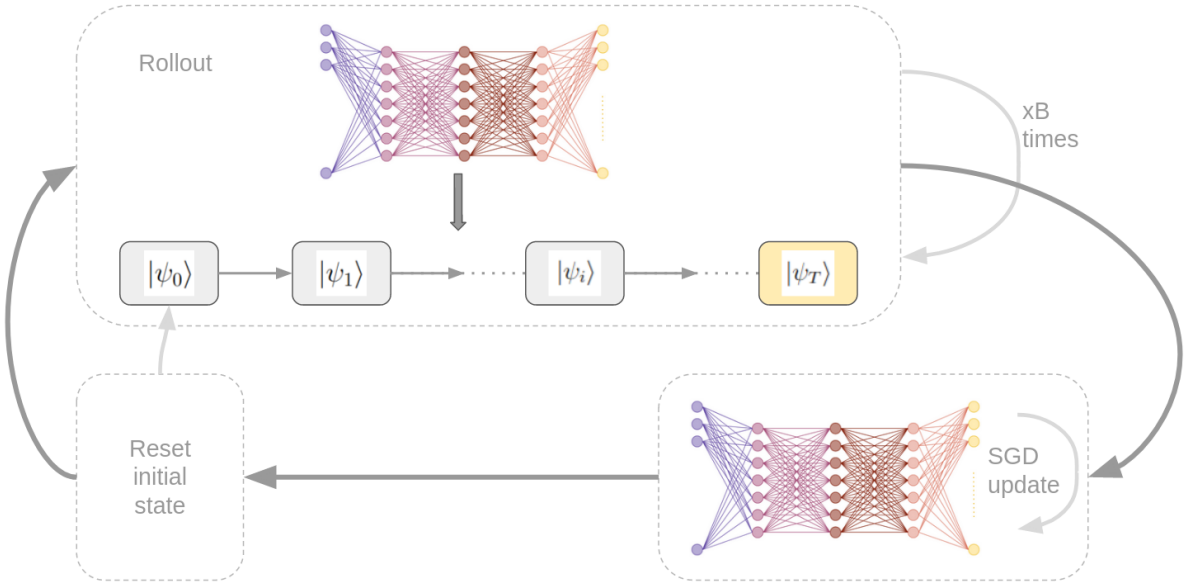


Figure 5.1: RL Agent training loop. The procedure starts with resetting the environment state. After that a number of rollout simulations are performed and the generated episodes are saved. Finally, the agent updates the policy parameters using the policy gradient update rule described in Sec. 2.1.4

solve sequential problems and greatly improves the results from the training. Training using RL is discussed in Sec. 6.4.

5.2 Reinforcement learning setup

The process of disentangling a quantum state is modelled using an agent-environment loop. We have an agent (e.g., a neural network, or a tree search algorithm) interacting with the environment (the quantum state) and trying to achieve a fixed goal – to disentangle the state. Using this setup we test and compare agents trained using different approaches. We examine different approaches for training the neural network model and we compare the different models by running an agent-environment loop and observing how well the agent behaves. The two key components – the agent and the environment, are now described in detail. Comparisons of different agents is given in Sec. 6.

5.2.1 Simulation environment

The environment for our problem is a physical simulation of a quantum system. As explained in Sec. 3.3, we model the state of a quantum system using tensors defined over the set of complex numbers – a quantum state of n qubits is a $\underbrace{\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes ... \otimes \mathbb{C}^2}_{\text{n times}}$ tensor. Applying quantum gates is modelled by performing matrix multiplication between the current state and the quantum gate.

Fig. 5.1 shows the training process using the simulation environment. At the beginning

```
    class Environment:
        def state(self):
            """Return the state of the environment."""
            // ...
            return self.state

        def actions(self):
            """Return a list with the actions."""
            // ...
            return self.actions

        def step(self, action):
            """Perform the selected action and transition the
            environment to the resulting state.
            """
            // ...
            return (next_state, reward, done)

        def reset(self, action):
            """Reset the quantum state to be ready for a new
            agent-environment loop.
            """
            // ...
            return self.state

        def entanglement(self):
            """Return the average entanglement of the
            current state.
            """
            // ...
            return entanglement
```

Code 5.1: Interface for the environment object. The environment simulates a quantum system and is used for training an RL agent. The concrete implementation of the environment follows the interface shown. An implementation of the environment can be found on https://github.com/cacao-macao/entanglement-control.

of every rollout the state of the environment is reset to the initial state $|\psi_0\rangle$. Usually, environments have a well defined initial state. A given environment might have only one possible or multiple initial states. In our case the environment has no specific initial state. Given that the goal is to have an agent that can disentangle an arbitrary quantum state, it seems natural the initial state of the environment to be simply a random quantum state.

At every iteration of the agent-environment rollout the agent will select an action a_t to be applied, that is the agent will select the indices of the two qubits $\{i, j\}$ to which it wants to act. After the indices are selected the environment will calculate the vector form of the quantum gate to be applied using the Eq. (4.3) and it will apply that gate to the selected qubits. After the action is applied, the state $|\psi_t\rangle$ of the environment will be changed to the new state $|\psi_{t+1}\rangle$ resulting from the application of the quantum gate.

After the agent performs multiple rollouts the weights of the policy are updated using the procedure described in Sec. 2.1.4.

In order to keep track of the progress of the agent, during the rollout we query the environment for the value of the entanglement of the current state. Since the entanglement

is a mathematical property of the quantum state, we implemented a procedure that calculates the value using Eq. (4.1) and returns it. An overview of the interface of the environment is given in Code 5.1.

5.2.2 State space

As explained in Sec. 3.2, quantum states are represented by tensors of complex numbers. However, most frameworks for training deep neural network models [23, 24] are implemented to operate on real numbers. For this reason we implemented a procedure for bi-directional mapping $\mathcal{M}$ between complex tensors and real tensors. The procedure works by simply concatenating the real part and the imaginary part of the complex tensor. Given, a complex tensor:

$$C = A + iB,$$

we construct the real mapping by stacking together the tensors A and B:

$$\mathcal{M}(C) = \begin{bmatrix} A \\ B \end{bmatrix}.$$

Note that, according to Eq. (3.3) states that differ by a global phase are physically equivalent. Thus, they cannot be distinguished from one another by any physical measurement. The agent, however, observes the internal representation of the quantum state and will be able to distinguish between states that differ by a global phase. For this reason states are 'phase-normalized' before they are input through the neural network. Thus, every quantum state is shifted by a global phase so that the component along the $|00\ldots0\rangle$ standard basis vector is a real number:

$$\begin{aligned} |\psi\rangle &= c_0 |00...00\rangle + c_1 |00\ldots01\rangle + \cdots + c_{2^n-1} |11\ldots1\rangle \\ \gamma &= -\mathrm{Arg}(c_0) \\ |\psi\rangle_{\mathrm{norm}} &= e^{i\gamma} |\psi\rangle , \end{aligned}$$

where $\mathrm{Arg}(c_0) = \arctan \dfrac{\mathrm{Re}(c_0)}{\mathrm{Im}(c_0)}$.

5.2.3 Action space

The actions to be selected by the agent are pairs of indices indicating which two qubits should the calculated quantum gate be applied on. Thus, the action space is the set of all variations of two qubit indices. For a quantum state of n qubits this means that the action space is $O(n^2)$. The downside of using this action space is that it increases polynomialy with the size of the system. However, for relatively small quantum states (e.g. 10-15 qubits) this is manageable.

Note that the action space consists of all *variations* instead of all *combinations*. This is due to the fact that selecting indices (i, j) is different from selecting indices (j, i), as discussed in Sec. 4.3.

5.2.4 Reward

The reward function that we use is to penalize the agent for every action that is performed with a reward of -1, and we give a positive reward only when the state is disentangled. The value for the positive reward is discussed in Sec. 6.4. This type of reward function directly specifies our goal – disentangle the quantum state while using as few actions as possible. However, rewards that carry any meaningful information are relatively rare, because only when a state is disentangled does the agent receive positive feedback. If, for example, the agent chooses a few good actions, but then fails to disentangle the state, the entire sequence of actions will be penalized. In order to solve this problem we decided to train with longer episodes, thus allowing the agent to achieve the reward more frequently.

When selecting the rewards for this problem we experimented with a few different variants. We tried to make the reward a function of the current average entanglement of the quantum state (see the definition of average entanglement in Eq. (4.1)). Using the entanglement of the system directly as feedback reward would provide for a very efficient training of the agent. The reward achieved at every step carries information how good was the action that was taken.

The first function that we tested was:

$$\mathcal{R}(s_t) = -S_{\text{avg}}(s_t).$$

What we saw was that the agent was learning how to disentangle the system up to some point and after that it could not learn how to continue reducing the entanglement. The reason for this was the fact that the reward was proportional to the entanglement, and when the entanglement of the state was small any further reductions yielded small rewards. To fix this issue we came up with a non-linear function relation between the entanglement and the reward:

$$\mathcal{R}(s_t) = -\log S_{\text{avg}}(s_t).$$

Using this function we managed to train an agent to disentangle quantum states, however we saw that often longer disentangling sequences were yielding higher returns values than shorter sequences. Thus, using this reward function would induce bias in the training of the agent, favoring reducing the average entanglement faster instead of disentangling the state faster.

5.3 Parallel simulation

During RL training, the agent performs numerous interactions with the environment via the agent-environment loop. At every iteration of the loop the system state is reset and the agent starts the interaction producing a sequence of *(state, action)* pairs (called an episode). In order for the agent to improve the policy model it needs to perform a huge amount of episodes, thus the runtime complexity of this procedure should be sufficiently optimized.

Note that generating interaction episodes is inherently parallel as there is no relation between two different agent-environment loops. In addition to that, the training algorithm described in Sec. 2.1.4 performs multiple interaction episodes before every update of the

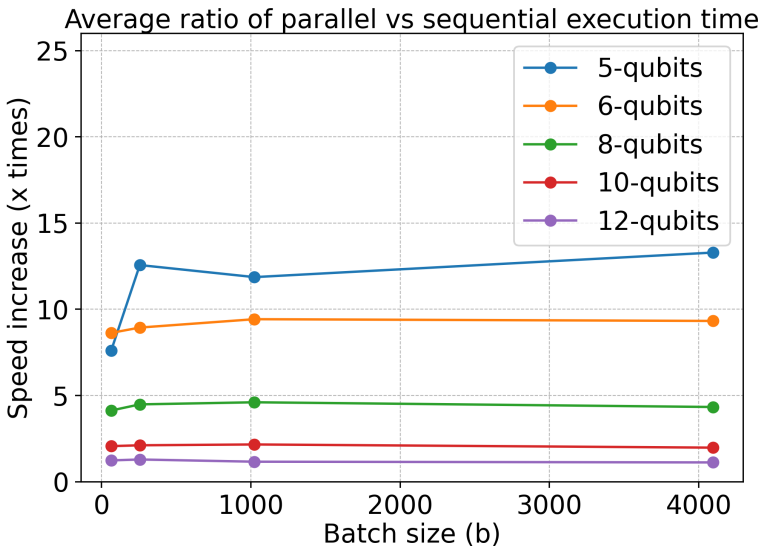


Figure 5.2: Comparison of parallel and sequential rollout. Parallel execution is performed with batches of different sizes. The average time to execute a single action on a single state is calculated for both parallel and sequential execution and the ration is plotted along the y-axis. Using parallel execution leads to 5x- to 10x-increase in the runtime. The test was performed on an Intel-Core i7-11800H processor.

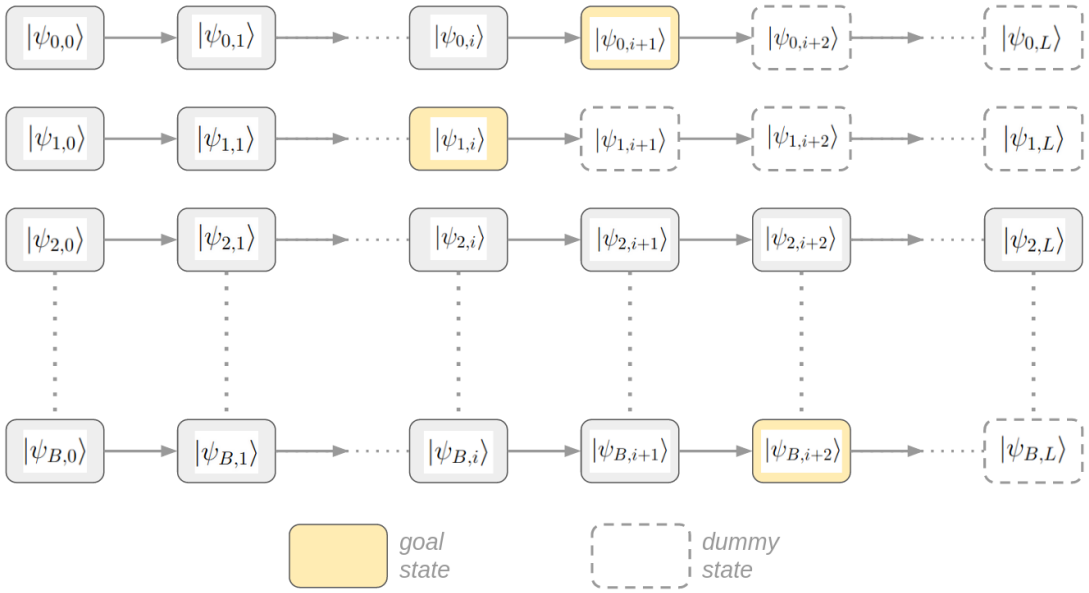


Figure 5.3: Multiple agent-environment loops in parallel. Every state visited during rollout is stored in a rectangular tensor form (see Eq. (3.3)). The states are batched together in a 2-dimensional table. All episodes in the batch must have the same length. Episodes that finish earlier are padded with a dummy state.

policy of the agent. Thus, we run different agent-environment loops in parallel, and once all episodes finish we perform a synchronous update of the parameters of the policy.

Recall that, executing an action boils down to applying linear algebraic operations, thus the application of multiple actions on different quantum states is simply batched together

```
    class Policy:
        def get_action(self, state):
            """Given the current state return a probability
            distribution over the action space.
            """
            // ...
            return probs
```

Code 5.2: Interface for the policy object. The policy processes the input quantum state and produces an action with which the agent should act. An implementation of the policy can be found on https://github.com/cacao-macao/entanglement-control.

and executed in parallel on different cores of the CPU or the GPU.

The quantum state simulator environment is designed to simulate a batch of quantum states, each state having the same number of qubits. The environment also supports parallel execution of actions on all the states in the batch. A simple comparison between the execution times of the parallel and the sequential environments on a CPU is shown on Fig. 5.2 revealing a 5x- to 10x-increase in computation times. Note that deep learning frameworks provide the possibility to forward batches through the neural network out of the box. Thus, our agents can simply forward the entire batch of quantum states through their policy model.

One downside of using a batched environment is that episodes running in parallel must have the same length. This means that even if one of the quantum states of the batch is disentangled, the agent must keep selecting actions for that state until the entire batch of episodes is finished. However, if the episode length needed to disentangle one of the states from the batch is too long, then this causes a computational overhead that is proportional to the batch size. To solve this problem we decided to use episodes with fixed lengths instead of running episodes until the state is disentangled. A discussion on selecting a suitable value for the episode length can be found in Sec. 6.1.

5.4 Agent

An agent is modelled as a composite entity consisting of a policy object and an environment object. The policy is an implementation of the prescription given in Sec. 2.1.3. It takes as input a quantum state and produces a probability distribution over the action space. The interface for the policy is shown in Code 5.2.

In our case the policy is a neural network model that is trained in order to improve its performance and ultimately produce optimal sequences for disentangling quantum states. In order to improve the policy the agent implements a training procedure that performs agent-environment loops and updates the policy. Different algorithms for updating the policy are given in Sec. 2.1.4, 2.2. When interacting with the environment, the agent queries its policy in order to select the action with which to act. The interface of the agent is shown in Code 5.3.

```
    class Agent:
        env = Environment()
        policy = Policy()

        def rollout(self, steps):
            """Perform a rollout of length `steps` or until the
            quantum system is disentangled. Use the current policy
            to select actions.

            Return the states and actions produced during the
            episode rollouts.
            """
            // ...
            return states, actions

        def train(self, num_iters, steps, hyperparameters_config):
            """Train the policy for `num_iters` iterations.
            At every iteration perform a rollout of length
            `steps` and update the weights of the policy.
            """
```

Code 5.3: Interface for the agent object. The agent is composed of a policy and an environment object. The agent trains the policy by performing agent-environment loops and updating the weights of the model. An implementation of the agent can be found on https://github.com/cacao-macao/entanglement-control.

Chapter 6

Artificial intelligence agents: training and results

In this section we give an overview of the different artificial intelligence agents that we tested and the results that we achieved. In Sec. 6.1 we discuss the most basic agent – the random agent. Section 6.2 provides a discussion about a classical search algorithm agent. Training an agent using supervised learning is discussed in Sec. 6.3 and training using reinforcement learning is discussed in Sec. 6.4. In Sec. 6.5 we propose an approach for pre-training agents combining the ideas of supervised learning and reinforcement learning.

6.1 Random agent

The first algorithm that we will discuss is the random agent. A random agent is an agent that uses a policy which selects actions from the action space uniformly at random. Due to the design of the quantum gate calculated using the procedure from Sec. 4.3, selecting actions at random actually yields a solution to the problem. However, as we will see shortly, this solution produces a lot of redundant actions, thus, significantly extending the length of the solution sequence.

The performance of the random agent is studied in order to understand the complexity of the problem and to try to come to a conclusion about the expected episode lengths. Figure 6.1 shows how the entanglement of a quantum state changes when the agent follows the uniform random policy. The experiment considers quantum states of $n = 5, 6, 8, 10, 12$ qubits. For every value of n the agent is given a batch of $b = 2048$ quantum states of n qubits and the value of the average entanglement $\frac{1}{b}\sum_{i=1}^{b} S_{\text{avg}}(|\psi_i\rangle)$ of the entire batch is plotted for every step of an episode rollout. We can see that an agent selecting actions at random is perfectly capable of disentangling a quantum state. However, the number of actions required to reach a certain threshold for the entanglement grows exponentially with the system size.

The analysis of Fig. 6.1 shows that the number of actions needed to disentangle a quantum system grows exponentially with the number of qubits: $T \in O(e^n)$. Although the random agent is very inefficient in the action selection, more sophisticated algorithms also yield solutions that are likely in the exponential class $O(e^n)$. This leads us to the following:

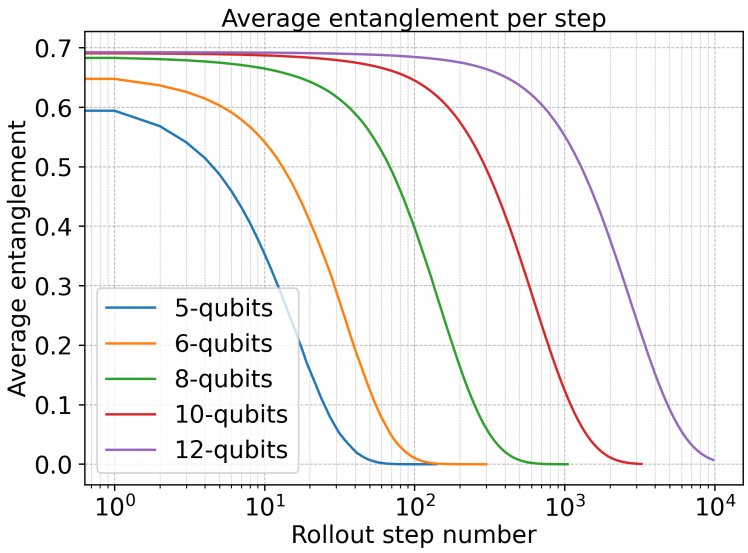


Figure 6.1: Random agent performance. This figure shows the average entanglement at every step of an episode rollout for quantum states of different sizes. Episode lengths increase exponentially with the size of the system. The number of steps is shown along the x-axis and is plotted on a logarithmic scale. For every system size the values are averaged over a batch of $b = 2048$ different states of that size.

Conjecture 2 *The length of the optimal solution T_{opt} to the disentangling problem 1 is exponential with respect to the number of qubits.*

As explained in Sec. 5.3 we have to select a suitable value for the episode length when using a batch of quantum states for parallel rollouts. Note that at the begining of the training procedure our the policy of the agent is initialized with random weights. Thus, the behaviour of the agent is similar to the random agent. The process of selecting the most suitable episode length is iterative and is based on trial and error. As a first iteration for the episode lenght we select the average number of steps needed by the random agent to disentangle a quantum system.

6.2 Search agent

Given the combinatorial nature of the problem at hand a class of algorithms suitable for solutions are tree-search algorithms. A search agent is an agent that uses a tree-search procedure to find a sequence of actions that disentangles a quantum state. We call the resulting sequence a **plan**. During the agent-environment loop, instead of querying a policy, our search agent selects the next action from the plan that was constructed beforehand. Note that using a plan during agent-environment loop is only possible for deterministic environments. In case the environment is non-deterministic (due to noise for example), using the plan from the tree-search algorithm is not certain to lead to a solution.

The first algorithm that we test is a brute-force search algorithm. However, running brute-force search is only possible for very small problems, and thus we are able to arrive at a solution only for quantum systems of size $n = 3, 4$ qubits. In the case of $n = 3$ qubits

we verified the results that we derived in Sec. 4.4.2.

The solution to a 4-qubit quantum state was quite interesting and deserves another work dedicated specifically to that problem. We found that any 4-qubit quantum state can be disentangled using a sequence of at most 5 actions. We are convinced that a mathematical proof can be given showing how these solution sequences are produced. However, this is beyond the scope of this work.

Note that the number of all possible T-step sequences of actions is d^T, where $d = |\mathcal{A}|$ is the number of different actions. For our problem the number of actions is equal to $n(n-1)$, where n is the number of qubits in the quantum state. Thus, in order for brute-force search to find an optimal solution, it has to check $\sum_{i=1}^{T} n^i(n-1)^i \sim n^{2T}$ different sequences. Thus, brute-force runs in $O(n^{2T})$ time. For a 4-qubit state we saw that the required sequence length is $T = 5$. This means that the search algorithm tried around $4^{10} = 1048576$ different trajectories before it found the solution. In the case of a quantum state of 5 qubits the number of actions needed to disentangle the state is around $T \sim 16$. For the case of a quantum state of 6 qubits this number is around $T \sim 50$. These numbers were experimentally observed using a local search algorithm as described in Sec. 6.2.

# of qubits	$n = 4$	$n = 5$	$n = 6$
# of steps	$T = 5$	$T = 16$	$T = 50$
# of sequences	1.05×10^6	2.33×10^{22}	6.53×10^{77}

Table 6.1: Expected number of different trajectories. The number of trajectories a brute-force search algorithm needs to try in order to find an optimal disentangling sequence. This number grows exponentially with the sequence length and super-exponentially with the system size n.

Examining Fig. 6.1 we conclude that the expected length of an optimal disentangling sequence grows exponentially with the number of qubits. This implies that the total number of possible trajectories that need to be checked grows super-exponentially. Table 6.1 shows the number of trajectories that brute-force search needs to try in order to solve quantum systems of sizes $n = 4, 5, 6$ qubits. With an optimistic assumption that checking whether a sequence is a solution takes 10^{-4} seconds, it is obvious that solving even a 5-qubit state is infeasible using brute-force search.

In order to overcome the super-exponential time complexity needed to run brute-force, we tried running a local search algorithm. We tested *beam search* using a simple scoring function that scores quantum states based on their current entanglement (see Eq. (4.1)):

$$\text{score}(|\psi\rangle) = S_{\text{avg}}(|\psi\rangle).$$

A description of the beam search algorithm is given in Sec. 2.3.2. The algorithm progressively expands multiple trajectories guided by the scoring function. Paths that have high scores continue to be expanded and paths that have low scores are abandoned. The number of expanded paths k is kept constant and this allows the algorithm to run more efficiently than a brute-force search algorithm.

Figure 6.2 shows how the algorithm works. At every step each of the k paths is expanded. For the current state in every path we perform each of the possible actions from the action set. The number of possible actions is $n(n-1)$; thus, this produces a set of $kn(n-1)$

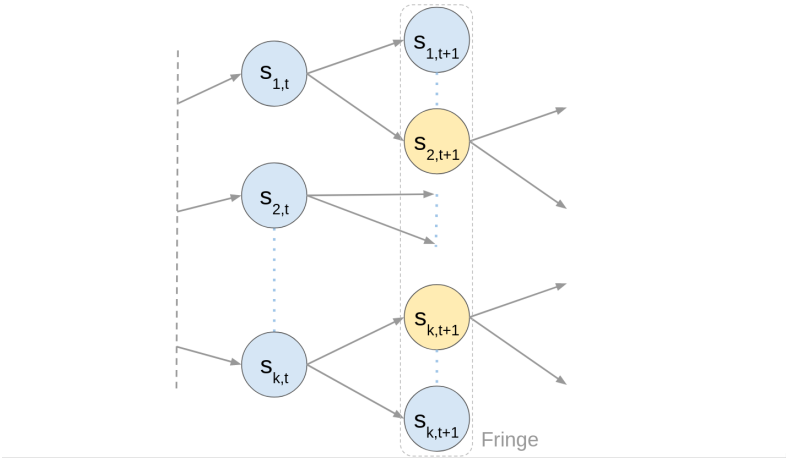


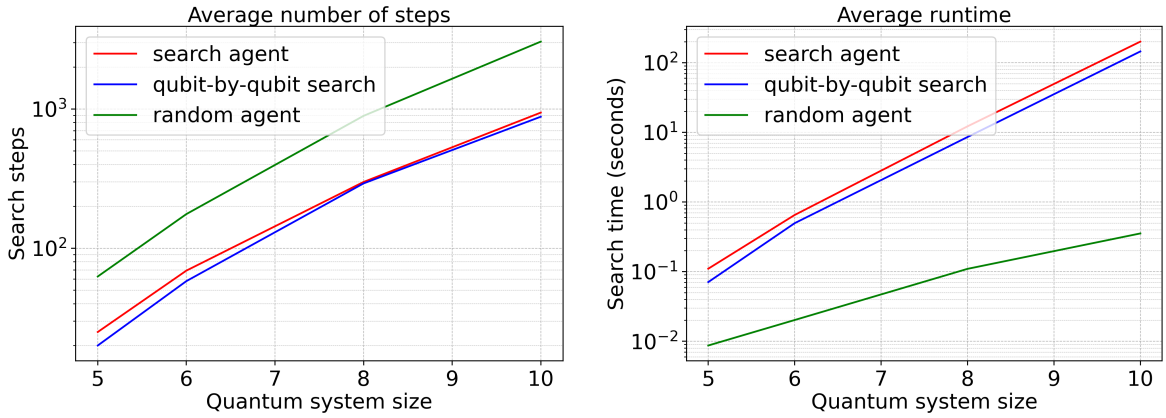
Figure 6.2: Functioning of beam search algorithm. At every step we extend the k-best paths. Current states are expanded by performing each of the possible actions producing a fringe of $kn(n-1)$ states. The algorithm continues by selecting the k-best states (colored yellow) from the fringe.

new states, called the fringe. The algorithm continues by selecting the k-best states from the fringe using the scoring function. From here we can see that beam search needs to explore $kn(n-1)T$ different trajectories before arriving at a solution, where T is the length of the solution sequence produced. Thus, beam search runs in $O(Tn^2)$ time, linear with respect to the sequence length.

It should be noted that guiding the search using the average entanglement of the entire state as a score function is not the best solution and a very simple improvement can be made. Note that if we could disentangle one of the qubits from the rest of the system then this qubit can no longer influence the other qubits and we can focus on disentangling only the rest of the system. Thus, instead of aiming to disentangle the entire system as a whole, we could run the beam search algorithm with the goal to disentangle only one specific qubit. The score function that we use is then the entanglement of that qubit with the rest of the system. When the qubit is disentangled we can simply run another beam search procedure on the rest of the system, again focusing on one specific qubit, and so on until the entire system is disentangled qubit-by-qubit.

A graph showing the running time of beam search for quantum systems with different numbers of qubits is shown in Fig. 6.3b. The figure compares the running times of beam search using the two different heuristic functions to the running time of a random agent. A comparison between the solution sequence lengths produced is given in Fig. 6.3a. From the two figures it can be seen that using the modified heuristic produces better (shorter) solutions, while also running faster. However, the length of the solutions grows exponentially with respect to the number of qubits, which lead us to Conjecture 2. We can also see that the random agent runs much faster than the beam search. In fact, the runtime of the random agent is $O(T)$ because action selection is performed for constant time at every step of the trajectory. However, the length of the solution sequence produced by the random agent is a magnitude higher.

Even though the search agent manages to find near-optimal solutions to our problem, it is obvious from Fig. 6.3b that the time needed to produce a solution sequence continues to grow exponentially with the number of qubits, reaching up to 100 seconds for 10-qubit states. We want to produce a policy as good as the search agent, but we want the policy



(a) (b)

Figure 6.3: Comparison between search agent and random agent. The algorithms are run until the average entanglement of the quantum system reaches a threshold value $\epsilon = 10^{-3}$. The search agent was run with two different scoring functions. The first uses $s(|\psi\rangle) = S_{\text{avg}}(|\psi\rangle)$ – states are scored using their average entanglement. The second uses $s(|\psi\rangle) = S(\rho_1)$ – states are scored using the entanglement of the first qubit. When the first qubit is disentangled, we recursively run another 'beam search' procedure disentangling the rest of the system. (a) Average number of actions that a random search agent and tree search agents use to disentangle quantum systems with different numbers of qubits. (b) Average run time that a random search agent and tree search agents use to disentangle quantum systems with different numbers of qubits.

to run in a time-frame comparable with the random agent. In order to do that in the following sections we explore how to train a neural network model as a policy.

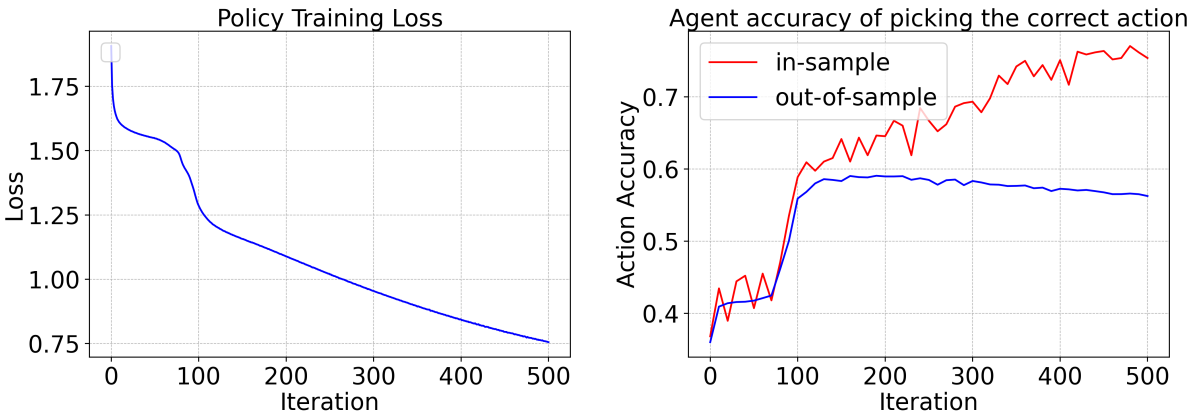
6.3 Imitation learning agent

The most basic agent using a neural network model as a policy is the imitation learning agent. The model is trained using behaviour cloning as explained in Sec. 2.2.

The most important ingredient to train a supervised learning model is a dataset of good examples. To train the model we ceate a dataset, where each example data point consists of a quantum state and the corresponding action that should be taken from that state, i.e., a *(state, action)* pair. We generate a dataset of examples using the search agent: given an arbitrary quantum state the search agent produces a solution sequence for disentangling that state, and we then divide that sequence into single step *(state, action)* pairs.

Using the search agent we can disentangle any quantum state; however, as we saw in Fig. 6.3b, the time to disentangle a state grows exponentially with the length of the produced solution. Having a neural network trained using the solutions produced by the search agent we aim at arriving at a model that produces solution sequences of the same near-optimal length as the search agent, but running in time linear with respect to the solutions it produces.

For the purpose of this work we train a neural network policy for disentangling quantum states of $n = 5$ qubits. To generate the dataset, we use the search agent with the modified *'qubit-by-qubit'* heuristic. The average length of the solution sequences produced by the



(a) (b)

Figure 6.4: Training an imitation learning agent on 5-qubit states. Training is performed on a dataset of 1.9×10^6 examples using batches of size $b = 2048$. The optimization procedure uses the *Adam* update rule with a learning rate parameter of $\text{lr} = 10^{-4}$. (a) Average value of the loss during each iteration of training, computed using Eq. (2.19), (b) In-sample and the out-of-sample accuracy of the agent. The action accuracy shows how well the model approaches the solutions provided by the search agent. The figures show that after the initial improvement the model performance increases very slowly. The results can be reproduced using the hyper-parameters from Table A.1.

search agent for 5-qubit states is $T = 19$. Thus, starting out with a set of 10^5 initial states, the search agent generates a dataset of approximately 1.9×10^6 data points.

To disentangle a 5-qubit state we experiment with different neural network architectures for the model, starting with a small fully-connected network with one hidden layer and increasing the size gradually. Satisfactory results are achieved with a fully-connected network with three hidden layers of sizes $[4096, 4096, 512]$ respectively. The model is trained using the *Adam* [35] optimizer. In each iteration we run through the entire dataset drawing at random batches of size $b = 2048$. For every data point pair in the batch (s_t, a_t) consisting of a state and the corresponding action, we run the state through the policy network producing a probability distribution over the action set. We then compute the data point loss as the *Kullback-Leibler divergence* between the produced probability distribution and the true distribution. The total loss at each step is computed as the average data point loss over the entire batch. The value of the loss is then back-propagated using the Adam update rule with learning rate of $\text{lr} = 10^{-4}$. The average loss at each iteration is shown in Fig. 6.4a.

During training we test the agent accuracy at every 100 iterations. The in-sample accuracy is tested on a batch of $b_{\text{in}} = 1024$ quantum states drawn at random from the training set. The out-of-sample accuracy is tested on a fixed test set of $D_{\text{test}} = 10000$ quantum states. To test the accuracy we compare whether the action scored with the highest probability by the policy matches the action from the dataset. The in-sample and out-of-sample action accuracies are plotted in Fig. 6.4b.

We can see from the panels in Fig. 6.4 that the agent learns well on the training set, achieving an accuracy of $\sim 75\%$ and continually reducing the value of the loss. However, the out-of-sample performance peaks at iteration 100 and from there on only decreases. Thus, the model does not generalize well and after iteration 100 starts overfitting the

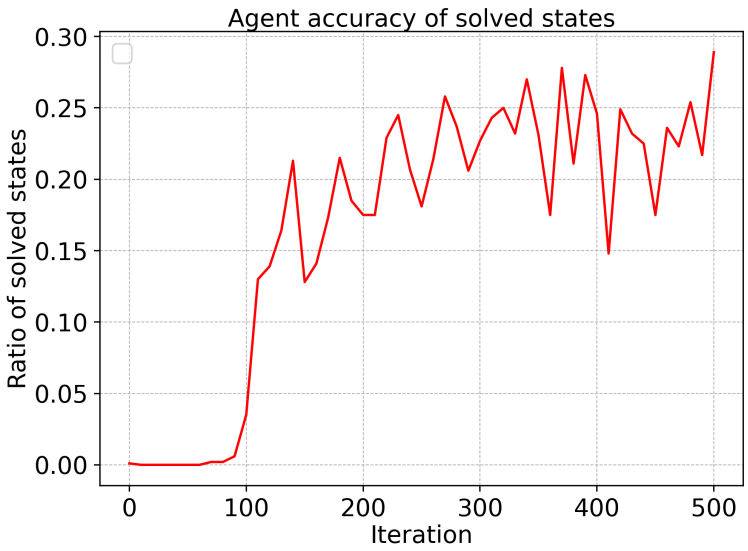


Figure 6.5: Imitation learning agent success rate when tested to disentangle a 5-qubit quantum state. The plot shows the percentage of states, for which the agent managed to reduce the entanglement below $\epsilon = 10^{-3}$. The test is performed every 100 iterations on a set of $b = 1000$ randomly generated quantum states. The agent runs for $T = 30$ steps and after that the entanglement of the quantum system is measured. We can see that the accuracy of solved states is significantly lower than the action accuracy shown on Fig. 6.4b. The results can be reproduced using the hyper-parameters from Table A.1.

training data.

Another test that we performed was to test the agent to disentangle randomly generated quantum states. The test consists of running the agent for 30 steps on a set of 1000 randomly initialized quantum states and recording the ratio of the states that were successfully disentangled. The results of the test are shown in Fig. 6.5. We can see that during the first ~ 150 iterations the agent greatly increases its performance; however, the ratio of the solved states still remains fairly low at $\sim 30\%$. Although the out-of-sample action accuracy of the agent is around $\sim 60\%$, disentangling the entire quantum state requires selecting the correct action multiple times, which results in the low performance on this test.

From the discussion so far we see that the model is capable of learning to achieve a great improvement in the early stages of the training process. However, after $\sim 100 - 150$ iterations the model starts overfitting the training data and its performance stops improving, and even declines.

There are several reasons for the lack, however, the main reason why the agent fails to learn to disentangle quantum states is that the problem is inherently sequential. We need to perform the training on entire sequences rather than splitting each sequence into single steps. One way to do that is to re-design the architecture of the neural network and use a RNN (Recurrent neural network). Note, however, that choosing an action only depends on the current state of the quantum system and does not depend on the history of the trajectory. This implies that we can instead model the problem as an MDP and train an agent using RL.

6.4 Policy-gradient agent

Given that the IL agent is not able to learn how to disentangle quantum states we decided to train an RL agent. The training procedure uses the PG (Policy gradient) algorithm as described in Sec. 2.1.4 and the quantum state is simulated using an environment model as described in Sec. 5.2. Again, for the purpose of this work, we will be considering only 5-qubit quantum states. We will use the same three-hidden-layers architecture in order to directly compare the PG agent with the IL agent.

As discussed in Sec. 5.3, we have to select a suitable value for the length of the episodes, as the model of the environment that we use performs simultaneous parallel rollout of multiple episodes. During training we experimented with several different values of the episode length T starting with $T = 19$, which is equal to the number of steeps needed for the search agent to disentangle a 5-qubit quantum state, and increasing the value gradually. The best results during training and subsequent testing were achieved using $T = 40$.

Another thing that needs to be selected is the scale of the rewards that the agent receives at every step of the agent-environment interaction loop. The penalty is set to $r_{\text{penalty}} = -1$, and for the reward achieved when the state is disentangled again we experimented with several different values $r_{\text{solution}} = 0, 1, 10, 100$. The best results were achieved with the value $r_{\text{solution}} = 100$. Using a value that is larger than the episode length is important because it instructs the agent at the early stages of learning that solving the problem is much more important than coming up with a very short solution sequence.

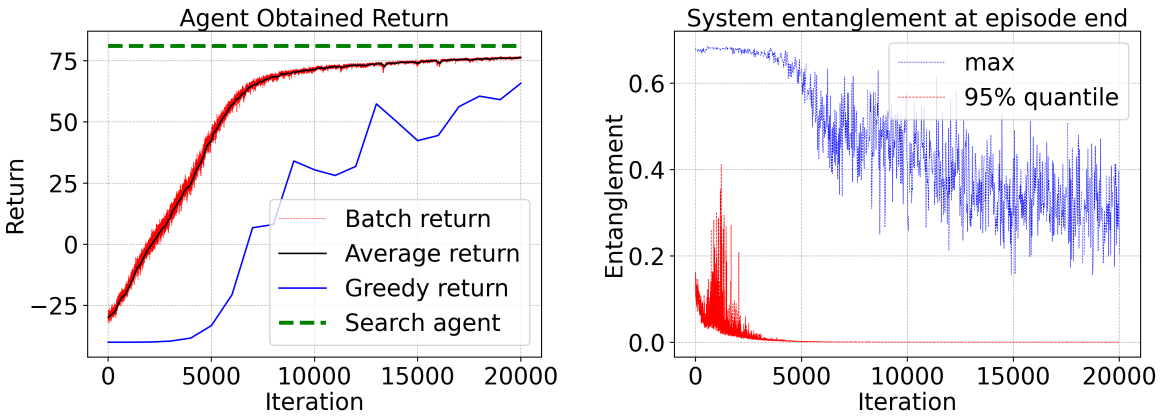
The most important metric to track during RL training is the return (see Eq. 2.2) obtained by the agent, and how that value compares with the target return that we are aiming for. Note that in this set-up our most sophisticated agent – which needs on average $T = 19$ steps to disentangle a 5-qubit quantum state, would achieve a return value of:

$$\mathcal{R}_{\text{search}}(\tau) = 19 \times (-1) + 100 = 81.$$

It should also be mentioned that initially we tried training the agent using the vanilla policy gradient procedure; however we quickly discovered that the policy was converging to a delta-function after a few hundred iterations and the training procedure was very unstable to adjustments of the hyper-parameters. In order to solve that issue we extended the policy gradient algorithm with an entropy-based regularization term (see Sec. 2.1.6). Adding entropy-based regularization incentives the policy to remain probabilistic, thus, encouraging exploration. For a thorough treatment of the topic of entropy-based regularization see App. B.

The training procedure follows the process from Fig .5.1. At every iteration a batch of $b = 1024$ episodes are rolled out in parallel for a duration of $T = 40$ steps. During rollout a quantum state is fed to the neural network policy and from the produced probability distribution an action is sampled. Once the rollout has finished the expected return for each state is calculated. The gradient of the policy is then calculated using Monte-Carlo approximation as explained in Sec.2.1.4 and back-propagated again using the *Adam* update rule with learning rate of $\text{lr} = 10^{-3}$. In addition, every 1000 iterations the agent is tested by running a "greedy" rollout where actions are not selected probabilistically, but rather the action with the highest score is always returned.

Note that the iterations during policy gradient training are very different from the itera-



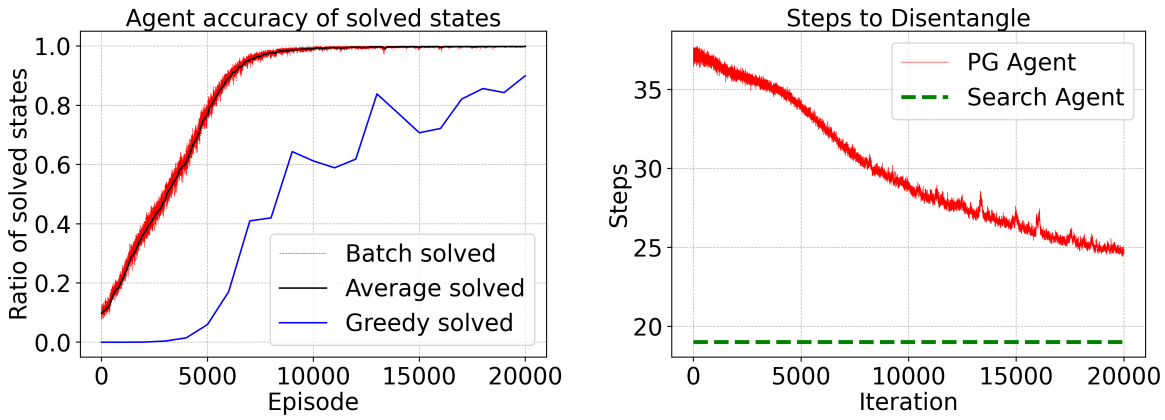
(a) (b)

Figure 6.6: Training a policy gradient agent on 5-qubit states. Training is performed by rolling out a batch of $b = 1024$ episodes for $T = 40$ steps, and calculating the policy gradient using Monte-Carlo approximation (see Sec. 2.1.4). The optimization procedure uses the *Adam* update rule with a learning rate parameter of $\mathrm{lr} = 10^{-3}$. (a) Average return accumulated by the agent during each iteration. The model successfully learns sequences of actions that result in higher returns. (b) Maximum and 95 percentile entanglement over the batch at the end of each episode. It can be seen that the agent learns how to disentangle almost all quantum states, except for a small fraction of outliers. The results can be reproduced using the hyper-parameters from Table A.1.

tions during imitation learning training. While in imitation learning in a single iteration we run through the entire dataset of 1.9×10^6 examples, here the agent observes only around $1024 \times 40 \sim 4 \times 10^4$ examples in a single iteration. Thus, we expect the number of iterations needed for training the PG agent to be much higher.

In Fig. 6.6a the accumulated returns during training and "greedy" testing are plotted. The average return achieved by the agent already approaches the results from the search agent, showing that the model successfully learns to find patterns and make decisions that have an impact on the episode future. In Fig. 6.6b are shown the maximal and the 95 percentile values of the entanglement measured over the states in the batch at the final step of the episode in each iteration. We can see that the agent quickly learns how to reduce the entanglement of more than 95 percent of the quantum states that it is given, leaving entangled only a small fraction of outlier states.

During training we also kept track of the percentage of solved states as well as the actual number of steps needed to disentangle each state. Looking at Fig. 6.7a we see that the agent needs about 10^4 iterations in order to learn how to successfully disentangle nearly 100% of the states given. In Fig. 6.7b we see that after the agent learns how to disentangle quantum states using sequences of length $T = 40$ it starts to explore other sequences searching for better solutions further reducing the number of steps needed. This behaviour is expected because of the way the reward function is designed. Note that the return achieved by the agent is centered by subtracting the baseline average return as given in Eq. (2.17). Thus, in case the agent manages to solve all states in the batch, the rewards achieved at the end of each episode are canceled out by the baseline, leaving only the accumulated penalties to score each episode. Thus, at the beginning of the training process learning is dominated by the final reward achieved when the state is disentangled. At the later stages learning is dominated by the length of the solution



(a) (b)

Figure 6.7: Performance of a policy gradient agent on 5-qubit states. (a) Percentage of solved states averaged over the entire batch. The agent needs around 10^4 iterations to learn how to disentangle nearly all of the quantum states. (b) Number of steps needed by the agent to disentangle a 5-qubit quantum state. The number of steps are averaged over the entire batch. We see that the agent continues to improve by producing shorter sequences after every iteration. The results can be reproduced using the hyper-parameters from Table A.1.

sequence.

Using the Policy gradient algorithm we successfully managed to train a neural network model that is capable of disentangling 5-qubit quantum states. Running the model and disentangling a quantum state requires only $O(T)$ time, where T is the length of the solution sequence produced. This is true because action selection is performed in constant $O(1)$ time with respect to the number of qubits. Note, however, that training the model requires a lot of iterations and takes a lot of time. Although, the required time depends heavily on the available hardware, training an agent to solve a 5-qubit quantum state for 2×10^4 iterations on a TeslaT4 GPU takes 19 hours.

6.5 Pre-trained agent

Despite the results of the PG agent, the solution is not scalable even to 6- or 8-qubit systems. The main reason for this is the amount of time it takes for the agent to train.

An approximate estimate of the time required to train an agent is given in Table 6.2. From Fig. 6.1 we can see that the random agent needs approximately $T = 100$ steps to disentangle a 6-qubit quantum state and $T = 400$ steps to disentangle an 8-qubit quantum state. We ran the policy gradient training procedure using these set-up configurations (e.g., $\{n = 6, T = 100\}$ and $\{n = 8, T = 500\}$) to obtain the time needed to run 10^3 iterations. In this calculation we assume that training on 6-qubit systems would need 5x increase of the number of iterations compared to 5-qubit systems, i.e., 10^5 iterations. We again assume that training on 8-qubit system would need 5x increase of the number of iterations compared to 6-qubit systems, i.e., 5×10^5 iterations. We should point out that these numbers are just an educated guess and are not backed up by any experimental data.

# iters	$n = 5,\ T = 40$	$n = 6,\ T = 100$	$n = 8,\ T = 400$
10^3	1 hour	1.5 hours	2.5 hours
2×10^4	19 hours		
10^5		150 hours	
5×10^5			1250 hours

Table 6.2: Expected time to train a policy gradient agent. This table shows the expected time in hours that would be needed to train a PG agent to disentangle 5-, 6-, and 8-qubit quantum systems respectively. The running times are calculated assuming that training an agent to disentangle a 6-qubit system would need 10^5 iterations, and training an agent to disentangle an 8-qubit system would need 5×10^5 iterations. The needed amount of time to train an agent grows exponentially meaning that this solution is not appropriate for even slightly larger quantum systems.

Looking at the table we see that the expected time to train an agent to disentangle an 8-qubit quantum system using the same algorithm and the same hardware would require around 52 days! Obviously we would need to come up with a better solution in order to solve larger quantum systems.

One way to speed up training is to reduce the number of iterations needed to train the agent to solve quantum states and focus mainly on optimizing the number of steps. Looking at Fig. 6.7, in the case of 5-qubit quantum states, almost half of the training time is dedicated to finding a policy that could solve all states in the batch, and only after that does the agent begin to improve on that policy by reducing the number of steps it takes.

To achieve this we apply an idea that is common in NLP (Natural language processing) and computer vision tasks and that is *transfer learning* [36]. Usually large neural network models are pre-trained on some general task such as sentence completion [37] in the case of NLP or image classification [38, 39] in the case of computer vision, and after that they are fine-tuned to the specific task which needs to be solved.

In our case, instead of starting the reinforcement learning procedure from scratch, we actually use the trained policy of the IL agent as the initial policy with which we apply reinforcement learning. In other words, instead of starting with a policy with randomly initialized weights, we start with a pre-trained policy that was trained on data generated by the search agent and we 'fine-tune' that policy using reinforcement learning. One important detail that we should keep in mind is that reinforcement learning works exactly because the agent starts off randomly and explores different solutions, and gradually improves on each iteration. If our pre-trained policy is not 'stochastic enough', then during the reinforcement learning procedure our agent will not be exploring sufficiently, thus, leading to bad results. However, if our pre-trained policy is 'too stochastic', then it will behave as if there was no pre-training at all. We see that pre-training has to strike a good balance between exploration and exploitation, once again underlining the importance of this trade-off.

Having this in mind we need to select a suitable amount of iterations for which to run imitation learning before continuing training with the policy gradient algorithm. However, the number of iterations is not a quantity that can be easily interpreted and extrapolated

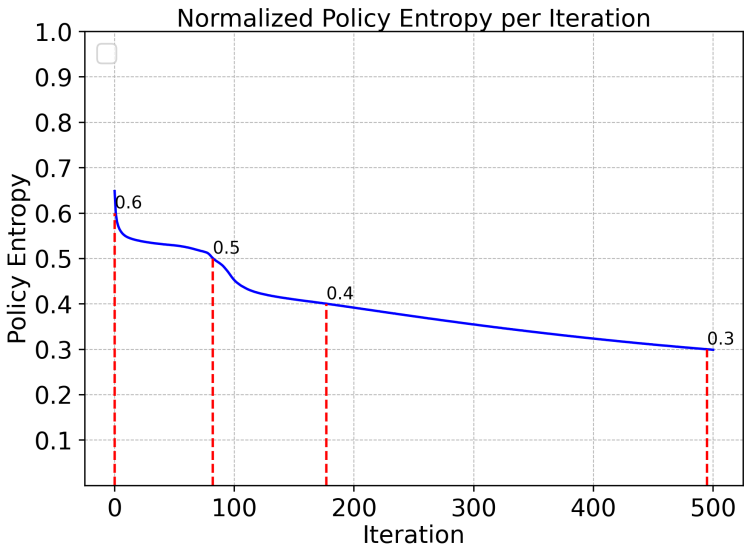
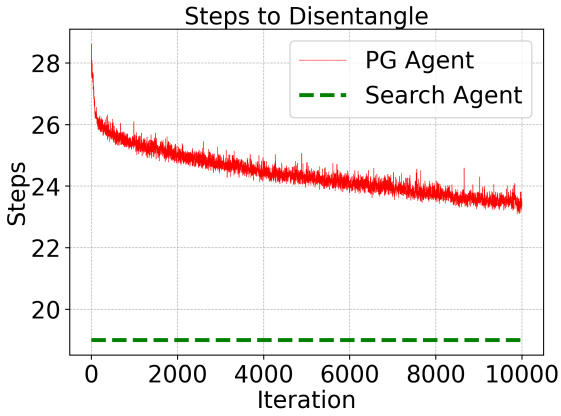


Figure 6.8: Normalized policy entropy during imitation learning training. The results from this figure are taken from the training process explained in Sec. 6.3. The normalized policy entropy measure is averaged over the batch of examples. Training starts with a randomly initialized policy with high entropy measure. During the training process the entropy of the policy decreases as the policy becomes less probabilistic. Red dashed vertical lines show the number of iterations needed to reach certain values of the normalized entropy measure $\{0.3, 0.4, 0.5\}$. Using the results from this figure we select the number of iterations for pre-training. The results can be reproduced using the hyper-parameters from Table A.1.

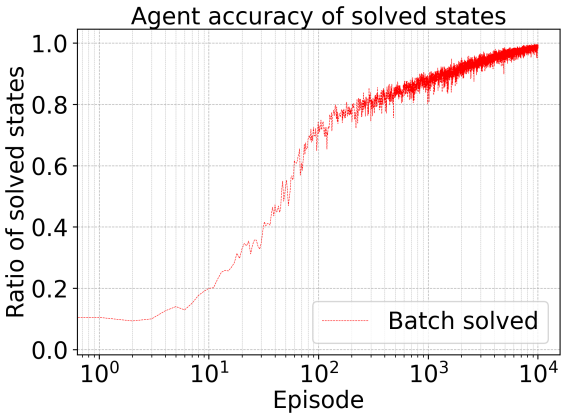
for agents trying to disentangle quantum states with different number of qubits, or even trying to disentangle quantum states with the same number of qubits but using different hyper-parameters. Instead, we will keep track of the entropy of the policy $H(\pi_\theta(\cdot|s))$ (see Eq. (2.18)) to decide on a suitable value beyond which we should switch from imitation learning to reinforcement learning. Note however, that even the entropy of the policy is not a measure that is easily transferred between problems of different number of qubits. The entropy of the policy depends on the size of the action space, which in turn varies with the size of the quantum state. Thus, we introduce the notion of a *normalized entropy measure*. We normalize the entropy in such a way that the uniform policy, having the highest entropy value, has a normalized entropy measure of 1.0:

$$\mathcal{H}_\pi = -\frac{H(\pi_\theta(\cdot|s_t))}{d}, \tag{6.1}$$

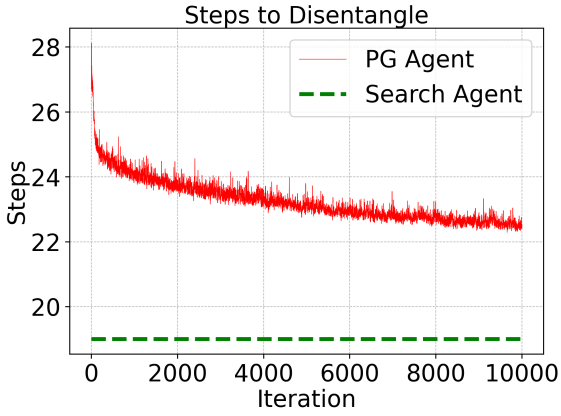
where $d = |\mathcal{A}(s_t)|$ is the size of the action space.



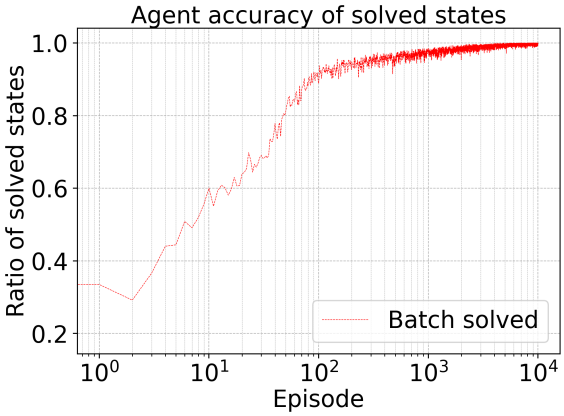
(a) pre-training for 100 iterations



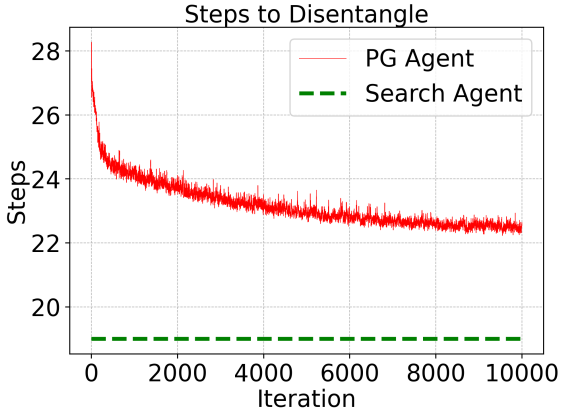
(b) pre-training for 100 iterations



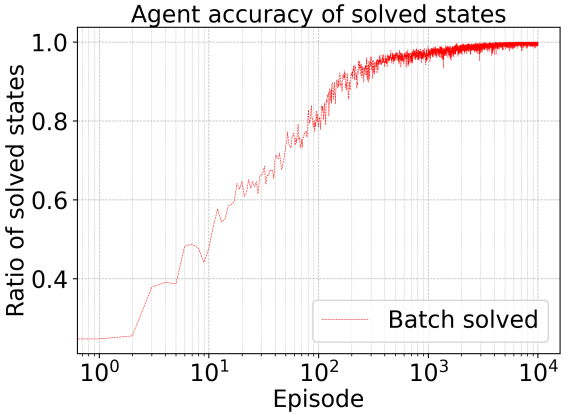
(c) pre-training for 200 iterations



(d) pre-training for 200 iterations

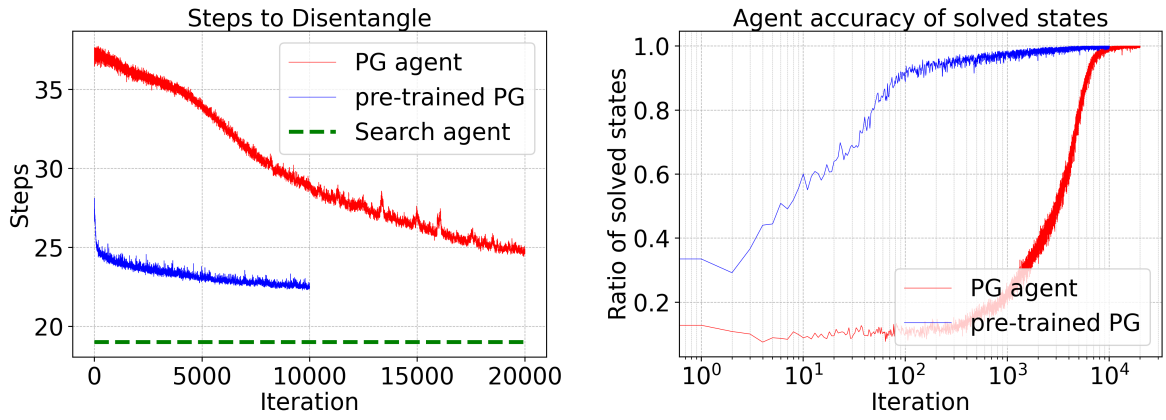


(e) pre-training for 500 iterations



(f) pre-training for 500 iterations

Figure 6.9: Policy gradient with pre-training. The figures show how the performance of the agents on disentangling a 5-qubit state improve during training. All pre-trained agents greatly outperform non-pretrained policy gradient training (see Fig. 6.7 for results of non-pretrained policy gradient agent, and Fig. 6.10 for comparison between the two). The figures showing the ratio of solved states are plotted on a logarithmic scale along the x-axis. We can see that the agent reaches above 90% success rate in just 100 iterations. The results with pre-training for 200 and 500 iterations are very similar, indicating that pre-training until a normalized entropy measure for the policy of 0.4 is reached is sufficient. The results can be reproduced using the hyper-parameters from Table A.1.



(a) (b)

Figure 6.10: Comparison between training a policy gradient agent with and without pre-training. The results of training without pre-training are taken from Fig. 6.7. The results of training with pre-training are taken from Figs. 6.9c, 6.9d. We see that policy gradient with pre-training achieves better results while requiring much less training time. The pre-trained model almost instantly learns to solve all quantum states in the batch and focuses entirely on improving the number of steps. The results can be reproduced using the hyper-parameters from Table A.1.

The normalized entropy of the policy averaged over the batch during imitation learning is shown on Fig. 6.8. We can see that during the training process the entropy of the policy decreases as the policy becomes less and less probabilistic. In the figure we see the number of iterations needed to reach certain values of the normalized entropy measure $\{0.3, 0.4, 0.5\}$. A checkpoint of the policy parameters was saved at every 100 iterations and the saved models that were closest to achieving these values were selected to be trained with policy gradient.

We ran three separate policy gradient training procedures with a pre-trained model after 100, 200 and 500 iterations. The results of the trainings are shown on Fig. 6.9. When training with a pre-trained model we decided to decrease the episode length to $T = 30$ steps. Note that the value of $T = 40$ was selected given that the agent starts out with a randomly initialized policy. In the current setting, however, the agent starts with a pre-trained policy and according to the results in Fig. 6.5 it is able to solve $20 \sim 30\%$ of the states it sees with $T = 30$ steps. The rewards that are used for training with a pre-trained model are the same as before, but the learning rate parameter is reduced to $\text{lr} = 5 \times 10^{-5}$.

An analysis of Fig. 6.9 shows that pre-trained agents almost instantly start solving all the quantum states in the batch and the training process focuses entirely on improving the number of steps. Note that the figures showing the ratio of solved states are plotted on a logarithmic scale on the x-axis. The results also show that pre-training for 200 and for 500 iterations yield the same results, and are slightly better than training for 100 iterations. This leads to the conclusion that pre-training to a normalized entropy measure for the policy of $\mathcal{H}_\pi = 0.4$ is sufficient.

In Fig. 6.10 we see a comparison between the metrics during the training processes of a policy gradient agent with and without pre-training. The pre-trained model not only trains much faster, but it also achieves a shorter sequence length.

Chapter 7

Outlook

Quantum entanglement is arguably the most important property of multi-qubit quantum systems. It has no classical analog and it is the driving force behind the power of quantum computing. Without this property quantum computers would have been computationally equivalent to a classical computer. To manipulate and compute quantum entanglement, however, we need to perform quantum simulations. And with quantum computers still largely unavailable, we need to run these simulations on classical computers.

It is due to entanglement that the simulation of quantum systems on a classical computer is a process of exponential complexity, in terms both of time and resources. Simulating a 60-qubit quantum state on a classical computer was considered to be impossible even on the most powerful super-computers available and it was only in 2022 when a 61-qubit quantum state simulation was successfully performed [40]. Controlling quantum entanglement is an even harder task. We have no formula that tells us what quantum gates to apply and to which qubits to apply them to in order to bring a quantum system's entanglement to a specified quantity. Thus, we need to perform numerous simulations, exploring different combinations of quantum actions, before we come to a solution.

In this work we set out to tackle the problem of controlling quantum entanglement in an efficient manner. We ask the question whether the internal representation of a quantum state provides any information regarding how the state can be disentangled into individual qubits. And if, indeed, there is a pattern, can we train an intelligent agent to learn that pattern, and in turn provide an efficient manner for controlling entanglement?

Based on the results of the experiments that were performed we can conclude that a pattern does in fact exist and that learning is feasible. We saw that using RL (Reinforcement learning) agents can learn which is the correct sequence of quantum operations that needs to be applied in order to disentangle 5-qubit quantum systems. We even went further and showed how other forms of learning (e.g. BC (Behavioural cloning)) can be used to enhance the performance of RL agents in the form of pre-training.

Nevertheless, it is safe to say that we have barely just scratched the surface of the problem of controlling quantum entanglement. First, given that the complexity to simulate quantum systems on a classical computer grows exponentially with respect to the number of qubits, it still remains to be seen whether this solution can be applied to quantum systems with larger numbers of qubits. It may be the case that other, more sophisticated RL algorithms, need to be applied and tested in order for the learning process to be improved. There have been a lot of improvements proposed in recent research, introducing

algorithms such as *TRPO* [41], *PPO* [42], *GAE* [43] and *Soft-Actor Critic* [21].

On the topic of transfer learning, we should point out that using supervised learning is the most basic form of pre-training. A number of recent works have been published [44, 45, 46, 47] focusing on pre-training using unsupervised learning. With this approach the agent is allowed a period of unsupervised interaction with the environment, during which it tries to build an improved representation of the state space. We expect that improving the pre-training procedure would reduce even further the training time for fine-tuning, which is essential for disentangling quantum systems with larger number of qubits.

Finally, we have to mention that uncertainty is at the heart of quantum mechanics. We cannot measure the values of complementary physical quantities with an arbitrary certainty. A fundamental limit is imposed to the accuracy of such measurements. Thus, in our simulations we have to account for this fact, and allow our models to tolerate non-deterministic behaviour. An idea for a future improvement is to have our RL agents operate on an observable of the quantum state instead of on the internal representation.

Obviously, there is still much more work to be done before arriving at agents that can control quantum entanglement in large quantum systems. However, this work lays the groundwork for many future investigations. We have split the process of evolving quantum systems into two tasks and we have designed an algorithm for each of them. We proposed a procedure for determining a locally optimal quantum gate that applies the maximum reduction to the quantum entanglement. Next, we trained an RL agent that given a quantum state can select the qubits to which an action should be applied. Using this set-up further research can be carried out exploring different opportunities for improvement and ultimately harnessing the power of quantum computing.

Appendix A

Hyper-parameters

Agent	NN	Opt	lr	T	β^{-1}
IL	FCNN	Adam	10^{-4}	-	-
PG	FCNN	Adam	10^{-3}	40	10^{-2}
pre-trained PG	FCNN	Adam	5×10^{-5}	30	10^{-2}

Table A.1: Hyper-parameters used for training. The table shows all the hyper-parameters used when performing the experiments.

FCNN (Fully-connected neural network) – fully connected neural network with three hidden layers of sizes 4096, 4096 and 512 respectively. The input to the neural network is a vector of size 2^{n+1} where n is the number of qubits in the quantum system. The output is a vector of size $n(n-1)$.

Adam – Adam optimizer with parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Appendix B

Entropy-based exploration for Monte-Carlo policy gradient methods

Researching the topic of entropy regularization in Monte Carlo policy gradient algorithms was the subject of a pre-thesis course project submitted in partial fulfillment of the requirements for the degree **M.Sc. in Artificial Intelligence**.

B.1 Introduction

Reinforcement learning is the study of agents and how they learn by trial and error. The key components of reinforcement learning are the agent and the environment. The interaction of the agent with the environment is described with a feedback loop as shown on Fig. B.1. At every step of the loop, the agent observes the state of the environment, and based on that observation decides on an action to take. When the environment is acted upon, it changes its state according to a specified transition function. The transition function outputs a probability distribution for the new state and that distribution depends only on the last observed state and the action taken by the agent. The environment also emits a scalar reward that can be described as a function of the previous state, the action taken, and the new state. One run of the loop from start to finish is called an episode and the goal of the agent is to maximise its total sum of rewards, called return, accumulated throughout the interaction loop. The agent is not told which actions to take, and instead discovers which actions are the most promising by trying them out.

In order to study and describe the algorithms used in reinforcement learning we need to rigorously formulate the problem as a MDP (Markov decision process). An MDP is a 5-tuple $\{\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, p_0\}$, where:

- $\mathcal{S}$ is the set of all possible states of the environment (could be infinite);
- $\mathcal{A}$ is the set of all possible actions that the agent can take. We will assume that the action space is discrete and finite;
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, where $\mathcal{R}(s, a, s')$ is the immediate reward received by the agent after transitioning from state s to state s' due to action a;
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition probability function, where $\mathcal{P}(s'|s, a)$ is the probability of transitioning from state s to state s' due to action a;

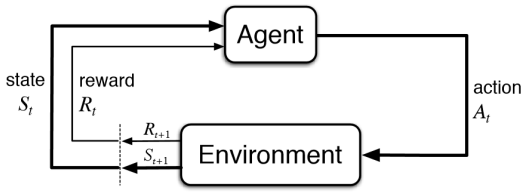


Figure B.1: Interaction loop between agent and environment. This figure shows the interaction feedback loop between the agent and the environment. At every step the agent observes the state of the environment s_t and takes an action a_t. The environment transitions into a new state s_{t+1} and emits a reward signal r_{t+1}.

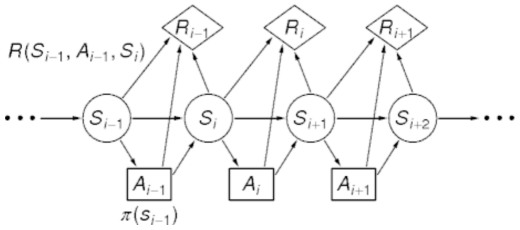


Figure B.2: Markov Decision Process. This figure shows a graphical representation of a Markov decision process. Arrows in the graph represent conditional dependence.

- and p_0 is the initial state distribution over the state space $\mathcal{S}$.

A graphical representation of a Markov Decision Process is shown on Fig. B.2.

In general we have no access to the transition function $\mathcal{P}$ and to the reward function $\mathcal{R}$, as we do not know the inner workings of the problems we are trying to solve (e.g. playing Atari or autonomous driving). To succeed at the task of maximizing the return, the agent must try different actions and progressively start preferring those actions that yield higher returns (i.e. it must exploit). However, to find actions that maximize the return, the agent must try new actions that it has not tried before (i.e. it must explore). This gives rise to the so called *exploration-exploitation dilemma* [48]. The trade-off between exploration and exploitation is one of the most important challenges in reinforcement learning. This feature of reinforcement learning does not arise in other kinds of learning (e.g. supervised or unsupervised learning).

To balance between exploration and exploitation we could add some stochasticity to the behaviour of the agent. While the agent mostly exploits its previous choices, choosing an action at random from time to time would cause the agent to explore actions that it would not normally choose. The idea of adding some stochasticity to the behaviour of the agent to increase exploration has been studied extensively and different methods have been proposed [49, 50].

An idea proposed by Williams and Peng in Ref. [22] is to modify the reward at every time step by augmenting it with a bonus that depends on the stochasticity of the agent (see Sec. B.3.1). The bonus is higher if the behaviour of the agent is more stochastic and the bonus is lower if the behaviour of the agent is more deterministic. This modification is referred to as *entropy regularization* and recently gained popularity with the introduction of the soft actor-critic algorithm by Haarnoja et. al. in Ref. [21]. Most of the work done with entropy regularization estimates the return $\mathcal{R}(\tau)$ using indirect methods based on

bootstrapping. This course project focuses on applying entropy regularization to policy gradient with Monte-Carlo estimates of the return.

B.2 Monte-Carlo Policy gradient

During each step of the agent-environment loop the agent observes the current state of the environment and chooses an action to take based on that state. Reinforcement learning algorithms produce a rule, also called a **policy**, for choosing an action based on the current state of the environment. More formally a policy

$$\pi : \mathcal{S} \to [0, 1]^d$$

, where d is the number of actions in the action space $d = |\mathcal{A}(s_t)|$, is a function that maps (state, action) pairs to a probability score, where $\pi(a_t|s_t)$ is the probability of picking action a_t when observing state s_t.

Some reinforcement learning algorithms try to learn the policy function directly (e.g. REINFORCE [51], TRPO [41], PPO [42]), while other algorithms learn the policy indirectly by first learning a value function (e.g. Q-learning [11]). Both approaches have their advantages and disadvantages and the choice of algorithm greatly depends on the problem we are trying to solve.

In this course project we examine Policy Gradient which is a direct policy optimization method that tries to learn a parametrised function representation of the policy $\pi \approx \pi_\theta$. The notation π_θ means that the function π depends on a set of parameters θ (e.g. weights and biases of a neural network). Policy Gradient tries to optimise the parameters θ by performing gradient ascent updates computed based on the gradient of a performance objective $J(\theta)$ with respect to the parameters θ.

The performance objective that we aim to maximise is the expectation of the sum of rewards over all time steps

$$J(\theta) = \mathbb{E}_{s\sim\mathcal{P},a\sim\pi_\theta}\left[\sum_{t=0}^{T} r_{t+1}\right] = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\mathcal{R}(\tau)\Big], \tag{B.1}$$

where

$$\mathcal{R}(\tau) = \sum_{t=0}^{T} r_{t+1} \tag{B.2}$$

is the return of the trajectory τ containing all time steps of a single episode.

The expectation of the return is computed over the probability distribution of the trajectories, denoted by $\mathcal{P}_\theta$. To sample a trajectory we actually have to sample a state $s_t \sim \mathcal{P}$ and an action $a_t \sim \pi_\theta$ for every time step t. The probability of a trajectory τ under a policy π_θ is given by

$$\mathcal{P}_\theta(\tau) = \mathcal{P}_0(s_0)\prod_{t=0}^{T} \pi_\theta(a_t|s_t)\mathcal{P}(s_{t+1}|s_t, a_t). \tag{B.3}$$

To optimise the parameters of the policy we will iteratively perform gradient ascent updates by computing the gradient of the objective $J(\theta)$ with respect to the parameters

θ and updating the parameters,

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta),$$

where α is the learning rate (or the *step size*). The parameter α is a hyperparameter that must be fine-tuned during training. This type of optimization method is on-policy, meaning that updates must use data collected while acting with the most recent version of the policy.

To compute the gradient of the objective $J(\theta)$ note that the expectation $\mathbb{E}[\mathcal{R}(\tau)]$ can be expressed as an integral:

$$\mathbb{E}[\mathcal{R}(\tau)] = \int \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau. \tag{B.4}$$

Now taking the derivative with respect to θ and noting that the return $\mathcal{R}(\tau)$ is not a function of θ we arrive at:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\mathcal{R}(\tau)\Big] = \int \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau, \\ \nabla_\theta J(\theta) &= \int \nabla_\theta \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau. \end{aligned} \tag{B.5}$$

Although we have an analytic expression for the gradient of the objective, this is not very helpful. Since in general we have no access to the transition function $\mathcal{P}$ and to the reward function $\mathcal{R}$, there is no way to compute the gradient from this expression. Instead, what we would like to do is express the gradient as an expectation over the probability distribution of the trajectories. Having the gradient expressed as an expectation means that we can approximate it using samples. Note the identity:

$$\nabla_\theta \mathcal{P}_\theta(\tau) = \nabla_\theta \mathcal{P}_\theta(\tau)\frac{\mathcal{P}_\theta(\tau)}{\mathcal{P}_\theta(\tau)} = \mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau). \tag{B.6}$$

Substituting into Eq. (B.5) we arrive at:

$$\begin{aligned} \nabla_\theta J(\theta) &= \int \mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\mathcal{D}\tau \\ &= \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\nabla_\theta \log \mathcal{P}_\theta(\tau)\mathcal{R}(\tau)\Big]. \end{aligned} \tag{B.7}$$

The expression $\log \mathcal{P}_\theta$ is further transformed in order to get rid of unknown quantities. Substituting $\mathcal{P}_\theta$ with the expression from Eq. (B.3) and applying the product rule for logarithms we arrive at the following:

$$\begin{aligned} \log \mathcal{P}_\theta(\tau) &= \log\left[p_0(s_0)\prod_{t=0}^{T}\pi_\theta(a_t|s_t)\mathcal{P}(s_{t+1}|s_t,a_t)\right] \\ &= \log \mathcal{P}_0(s_0) + \sum_{t=0}^{T}\Big[\log \pi_\theta(a_t|s_t) + \log \mathcal{P}(s_{t+1}|s_t,a_t)\Big]. \end{aligned} \tag{B.8}$$

However, the transition function of the Markov Decsion Process does not depend on the parameters θ, and thus, its derivative w.r.t. θ must vanish. Differentiating the previous expression leads to:

$$\nabla_\theta \log \mathcal{P}_\theta(\tau) = \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t). \tag{B.9}$$

Finally, after combining Eq. (B.7) and Eq. (B.9), we arrive at an expression for the gradient of the objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t=0}^{T} r_{t+1}\right]. \tag{B.10}$$

To compute the gradient using samples we can collect a set of trajectories $D = \{\tau_i\}_{i=1,\dots,N}$, obtained using the agent-environment interaction loop. For each trajectory the agent uses the policy π_θ to choose its actions. Then the policy gradient can be approximated with:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\sum_{t=0}^{T} r_{t+1,i}, \tag{B.11}$$

where N is the number of trajectories collected in D. This method for calculating the gradient of the objective is also known as *Monte Carlo Policy Gradient* as it uses a Monte-Carlo estimate of the return.

Assuming that we have represented our policy in a way which allows us to calculate $\nabla_\theta \log \pi_\theta(a_t|s_t)$, we can compute the policy gradient using sample trajectories and take an update step. In case our policy π_θ is parametrised by a neural network, however, we might ask whether it is possible to use an automatic differentiation software package [24, 23] to backpropagate the derivative of the objective $\nabla J(\theta)$ in order to update the weights of the model. To use auto differentiation we would need to define a loss function such that its gradient is equal to the policy gradient.

Considering the function taken by integrating Eq. (B.11):

$$\begin{aligned} J_{\text{pseudo}}(\theta) &= \int \nabla_\theta J(\theta)\mathrm{d}\theta \\ &\approx \int \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\sum_{t=0}^{T} r_{t+1,i}\mathrm{d}\theta \\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\sum_{t=0}^{T} r_{t+1,i}, \end{aligned} \tag{B.12}$$

we can see that this function has the nice property that its gradient is equal to the policy gradient when the (s_t, a_t) pairs are collected while acting with the current policy.

The function given by Eq. (B.12) is usually called a pseudo objective, because it is not a loss function in the typical sense from supervised learning. Firstly, we are not sampling data independently but rather the data must be sampled using the most recent model parameters. And secondly, this function does not evaluate the metric that we aim to optimize. Our goal is to maximize the expected return and this loss function has no connection to the real objective. It is only useful to evaluate the gradient of our objective at the current parameters, with data generated by the current parameters.

B.3 Exploration with entropy regularization

When learning a policy with policy gradient, exploration is ensured by the fact that the policy outputs a probability distribution over the action space and actions are selected

in a non-deterministic manner. However, if an agent observes some state s_k and chooses an action a_k producing *some* positive reward, then that action is reinforced. Thus, the probability of selecting that action when observing state s_k is increased. This makes it more likely to reinforce that action again in the future, which in turn further increases the probability of selection. It might happen that the agent will keep selecting this action, while there could exist another action yielding a much higher return. Effectively, the policy becomes a delta-function, choosing (sub-)optimal actions without the possibility of exploring.

To see more rigorously why the policy would become a deterministic function we have to note that the value of our objective depends on the logarithm of the probabilities of the selected actions. The logarithm is an increasing function and it achieves its highest value when the probability is equal to 1. Thus, if for a given state there are multiple actions that lead to the same optimal return, then optimizing the objective would mean setting the probability for one of those actions to 1.

B.3.1 Maximum entropy principle

To address this problem and improve the exploration capability of policy gradient, a modification to the reward function was introduced by Williams and Peng in Ref. [22]. The reward received by the agent at each time step is augmented with a bonus, which is proportional to the entropy of the policy at that time step:

$$r^*_{t+1} = r_{t+1} + \beta^{-1} H(\pi_\theta(s_t)), \tag{B.13}$$

where β^{-1} is a temperature parameter that controls the level of regularization, and

$$\begin{aligned} H(\pi_\theta(s_t)) &= -\sum_{a\in\mathcal{A}} \pi_\theta(a|s_t) \cdot \log \pi_\theta(a|s_t) \\ &= -\mathbb{E}_{a\sim\pi_\theta}\Big[\log \pi_\theta(a|s_t)\Big] \end{aligned} \tag{B.14}$$

is the *entropy* of the probability distribution $\pi_\theta(s_t)$.

The idea behind this modification is to train a policy that achieves high returns while acting as randomly as possible. This way the agent learns by itself how much exploration is necessary in order to learn properly. Substituting Eqs. (B.13) and (B.14) into Eq. (B.1), we arrive at a new expression for the new performance objective that we aim to maximise:

$$\begin{aligned} J(\theta) =& \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} r^*_{t+1}\right] \\ =& \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} r_{t+1} - \beta^{-1}\sum_{t=0}^{T}\log \pi_\theta(a_t|s_t)\right]. \end{aligned} \tag{B.15}$$

It is important to note that with the new objective we are not simply searching for a policy with high entropy. Instead, we want a policy that would direct the agent into states where the policy has high entropy. In other words, we want the agent to explore states for which it has not yet learned how to act. To prove this statement we will show that optimizing the objective given in Eq (B.15) would produce a policy that actually tries to maximize the entropy of the probability distribution of the trajectories $H(\mathcal{P}_\theta)$.

Increasing $H(\mathcal{P}_\theta)$ means that the agent is less likely to repeat trajectories and more likely to explore new paths.

Consider augmenting the objective given in Eq. (B.1) with the entropy of the probability distribution of the trajectories:

$$\begin{aligned} J'(\theta) &= \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} r_{t+1}\right] + \beta^{-1}H(\mathcal{P}_\theta) \\ &= \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} r_{t+1}\right] - \beta^{-1}\mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\log \mathcal{P}_\theta(\tau)\right]. \end{aligned}$$

Replacing $\log \mathcal{P}_\theta(\tau)$ with the result from Eq. (B.8) we can see that:

$$\begin{aligned} J'(\theta) =& \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} r_{t+1}\right] \\ &- \beta^{-1}\mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\right] \\ &- \beta^{-1}\log p_0(s_0) - \beta^{-1}\sum_{t=0}^{T} \log \mathcal{P}(s_{t+1}|s_t, a_t). \end{aligned}$$

Note that the transition function $\mathcal{P}$ does not depend on the parameters θ and thus the last two additives are constants; we have:

$$J'(\theta) = J(\theta) - \text{const}.$$

Since we are looking for the values of the parameters θ that maximize the objective, then adding a constant will not change the solution. Thus, finding a policy that maximizes the objective $J(\theta)$ will in fact also maximize the objective $J'(\theta)$.

B.3.2 Derivation of Entropy-regularized policy gradient

Modifying the objective function by augmenting the rewards with the entropy of the policy at that time step means that we now have to re-derive the gradient of the objective. The reason for this is that the additional entropy term also depends on the parameters θ and this has to be accounted for when taking the derivative.

Differentiating Eq. (B.15) with respect to θ and using the linearity of the derivative we arrive at the following expression for the gradient of the modified objective:

$$\begin{aligned} \nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}&\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t=0}^{T} r_{t+1}\right] \\ &-\beta^{-1}\nabla_\theta \int \mathcal{P}_\theta(\tau)\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t)\mathcal{D}\tau. \end{aligned} \tag{B.16}$$

The first part of Eq. (B.16) comes by directly plugging in Eq. (B.10). For the second part we need to do differentiation by parts. To simplify the notation let us write:

$$\Sigma_\theta(\tau) = \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t).$$

Note that from Eq. (B.9) we also have:

$$\nabla_\theta \log \mathcal{P}_\theta(\tau) = \nabla_\theta \Sigma_\theta(\tau).$$

Focusing only on the second part of Eq. (B.16) and applying again the substitution from Eq. (B.6) we have:

$$\begin{aligned}
&\nabla_\theta \int \mathcal{P}_\theta(\tau)\Sigma_\theta(\tau)\mathcal{D}\tau = \\
&= \int \nabla_\theta \mathcal{P}_\theta(\tau)\Sigma_\theta(\tau) + \mathcal{P}_\theta(\tau)\nabla_\theta\Sigma_\theta(\tau)\mathcal{D}\tau \\
&= \int \mathcal{P}_\theta(\tau)\nabla_\theta \log \mathcal{P}_\theta(\tau)\Sigma_\theta(\tau) + \mathcal{P}_\theta(\tau)\nabla_\theta\Sigma_\theta(\tau)\mathcal{D}\tau \\
&= \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Big[\nabla_\theta\Sigma_\theta(\tau)\Big[\Sigma_\theta(\tau)+1\Big]\Big].
\end{aligned}$$

Finally for the gradient of the new objective given by Eq. (B.15) we have:

$$\begin{aligned}
\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\Bigg[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\times \\
\times\Bigg[\sum_{t=0}^{T} r_{t+1} - \beta^{-1}\sum_{t=0}^{T}\log\pi_\theta(a_t|s_t) - \beta^{-1}\Bigg]\Bigg].
\end{aligned}$$

Again, as the gradient is expressed as an expectation we can estimate it by drawing samples:

$$\begin{aligned}
\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\Bigg[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_{t,i}|s_{t,i})\times \\
\times\Bigg[\sum_{t=0}^{T} r_{t+1,i} - \beta^{-1}\sum_{t=0}^{T}\log\pi_\theta(a_{t,i}|s_{t,i}) - \beta^{-1}\Bigg]\Bigg].
\end{aligned} \tag{B.17}$$

Similar to Eq. (B.12), we need to set-up a pseudo objective by integrating the approximation of the gradient given by Eq. (B.17). We see that the expression for the policy gradient given by Eq. (B.17) consists of three terms. The first term integrates similarly to Eq. (B.12). Also it is trivial to find the anti-derivative of the last term. However, the integration of the second term deserves some attention. To perform the integration we will use the following identity:

$$\int f(\theta)\nabla_\theta f(\theta)\mathrm{d}\theta = \frac{1}{2}f^2(\theta), \tag{B.18}$$

which holds for any integrable function f.

Applying the identity given in Eq. (B.18) to the second term of Eq. (B.17) we obtain:

$$\begin{aligned}
\int \sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t=0}^{T}\log\pi_\theta(a_t|s_t)\mathrm{d}\theta = \\
= \frac{1}{2}\sum_{t=0}^{T}\log\pi_\theta(a_t|s_t)\sum_{t=0}^{T}\log\pi_\theta(a_t|s_t).
\end{aligned}$$

For the new pseudo objective function we have:

$$J_{\text{pseudo}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\times \\ \times\left[\sum_{t=0}^{T} r_{t+1,i} - \frac{1}{2}\beta^{-1}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i}) - \beta^{-1}\right]. \quad \text{(B.19)}$$

B.3.3 Reducing variance

Monte-Carlo methods have the desirable quality of producing unbiased estimates of the quantities we are trying to estimate. However, Monte-Carlo methods produce estimates of very high variance. This is problematic as high variance estimates of the gradient will not produce good updates of the policy parameters. In order to obtain a low variance estimate of a quantity we would need a huge amount of samples which would significantly increase the time needed for learning a policy.

To reduce the variance of the policy gradient approximation given in Eq. (B.10) we will show that instead of the full trajectory return $\mathcal{R}(\tau) = \sum_{t=0}^{T} r_{t+1}$ we could use the return starting from step t, $\sum_{t'=t}^{T} r_{t'+1}$ without changing the expectation.

For any probability distribution P_θ parametrised by θ we have:

$$\begin{aligned}\mathbb{E}_{x\sim P_\theta}[\nabla_\theta \log P_\theta(x)] &= \int P_\theta(x)\nabla_\theta \log P_\theta(x)\mathrm{d}x \\ &= \int P_\theta(x)\frac{\nabla_\theta P_\theta(x)}{P_\theta(x)}\mathrm{d}x \\ &= \int \nabla_\theta P_\theta(x)\mathrm{d}x \\ &= \nabla_\theta \int P_\theta(x)\mathrm{d}x = \nabla_\theta 1 = 0. \end{aligned} \quad \text{(B.20)}$$

An immediate consequence of Eq. (B.20) is that for any function b, which does not depend on the action a_t, we have:

$$\mathbb{E}_{a_t\sim\pi_\theta}\Big[\nabla_\theta \log \pi_\theta(a_t|s_t)b(s_t)\Big] = 0. \quad \text{(B.21)}$$

Notice that, for a given episode, rewards obtained at earlier time steps should have no effect on choosing the current action: indeed, agents should only reinforce actions based on future returns and not on earlier gains. Noting that previous rewards are not a function of the current action, and plugging $r_{t'+1}$ into Eq. (B.21), it can be seen that:

$$\mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=0}^{t} r_{t'+1}\right] = 0, \quad \text{(B.22)}$$

for $t' < t$.

The expressions for the gradient and for the pseudo objective given in Eqs. (B.10) and

(B.12) would now change to:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\sum_{t'=t}^{T} r_{t'+1}\right]$$
$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\sum_{t'=t}^{T} r_{t'+1,i}.$$

A similar argument can be applied to the expression given in Eq. (B.17) which can be rewritten as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\times \right.$$
$$\left.\times\left[\sum_{t'=t}^{T} r_{t'+1} - \beta^{-1}\sum_{t'=t}^{T}\log \pi_\theta(a_{t'}|s_{t'}) - \beta^{-1}\right]\right]. \quad \text{(B.23)}$$

One problem with the expression given in Eq. (B.23) is that there is no closed form solution for the anti-derivative of this expression. And, thus, there is no straightforward way to arrive at an expression for the pseudo objective.; see Sec. B.6 for a proof of this statement.

However, let us first consider what we could do to reduce the variance of the expression given in Eq. (B.17). First of all, we can apply the variance reduction modification to the first term. Second, the second term must not be modified in order to be able to arrive at a closed form solution for the anti-derivative of the expression. Third, the last term can be omitted as it is not a function of the current action, and according to Eq. (B.21) its expectation is 0. Finally, we arrive at the following expressions for the gradient of the objective and for the pseudo objective:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\mathcal{P}_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\times \right.$$
$$\left.\times\left[\sum_{t'=t}^{T} r_{t'+1} - \beta^{-1}\sum_{t=0}^{T}\log \pi_\theta(a_t|s_{t'})\right]\right]$$
$$J_{\text{pseudo}}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\times$$
$$\times\left[\sum_{t'=t}^{T} r_{t'+1,i} - \frac{1}{2}\beta^{-1}\sum_{t=0}^{T}\log \pi_\theta(a_{t,i}|s_{t,i})\right].$$

B.4 Experiments and Results

To show how using entropy regularization improves training we will work with a simple gridworld problem. In Fig. B.3 are shown two different gridworld layouts:

- A simple symmetric 5x5 gridworld with two terminal states. Using this layout we will demonstrate how training with entropy regularization produces a non-deterministic policy;

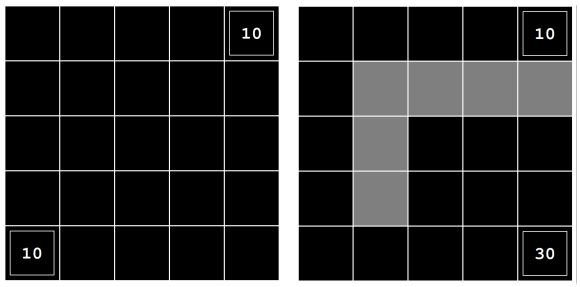


Figure B.3: Gridworld Layouts. The gridworld environment has a living reward of -1 and an exit reward for leaving the terminal state. Two layouts are considered for the experiments – SmallGrid (left) and MazeGrid (right). We use the SmallGrid layout to demonstrate how training with entropy regularization produces a non-deterministic policy. And we use the MazeGrid layout to demonstrate how training with entropy regularization produces a more optimal policy.

- A 5x5 gridworld maze with two terminal states. On this layout we will demonstrate how training with entropy regularization produces a better policy that achieves higher returns.

The cells of the grid correspond to the states of the environment. At each cell four actions are possible: *north*, *south*, *east* and *west*, which deterministically cause the agent to move one cell in the respective direction of the grid. Actions that would take the agent off the grid leave its location unchanged. Every action results in a reward of -1. Leaving the terminal state terminates the game and awards the agent an additional reward.

The states of the environment are encoded as one-hot vectors. The policy that we will train is a linear function that maps each state to a score vector and applies a softmax after that to produce a probability distribution over the action space.

$$
\begin{aligned}
s_t &= \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 & 0 & \cdots & 0 \end{bmatrix} \\
\pi_\theta(s_t) &= \text{softmax}(s_t \cdot \theta)
\end{aligned}
$$

B.4.1 SmallGrid

There are multiple policies that achieve the optimal return on the SmallGrid layout. However, we would like to arrive at a policy that achieves the optimal return while remaining as probabilistic as possible.

Figures B.4 and B.5 show the final policies learned by the agent with and without entropy regularization respectively. We can see that when training without entropy regularization the agent arrives at an optimal but deterministic policy for the problem. While training with entropy regularization the final policy is (near) optimal but remains probabilistic.

Figure B.6 shows the average return obtained by the agent as a function of the training iteration. When training without entropy regularization the agent achieves higher returns and also needs fewer iterations to arrive at a better policy. The agent quickly learns to exploit the knowledge it has for the environment.

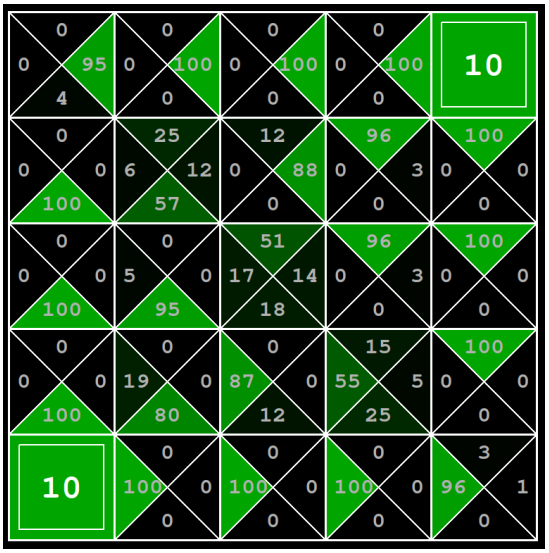


Figure B.4: Final policy for the SmallGrid layout without entropy regularization. The policy converges to a deterministic policy when training without entropy regularization. Training is performed with entropy regularization temperature of 0.

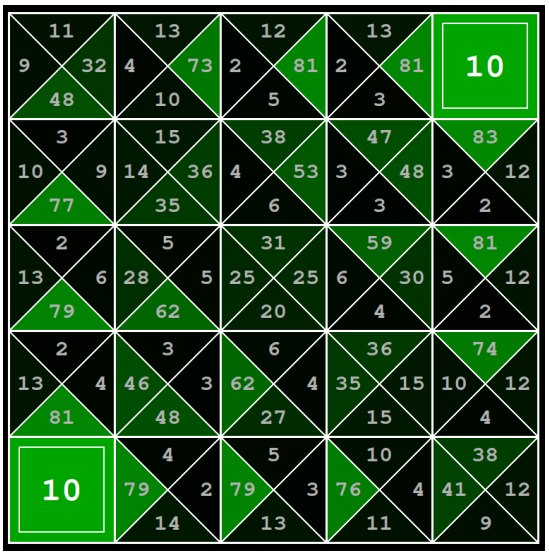


Figure B.5: Final policy for the SmallGrid layout with entropy regularization. The policy achieved when training with entropy regularization is non-deterministic. Training is performed with entropy regularization temperature of 1.

However, the downside is that the policy quickly converges to a deterministic function and the agent mostly exploits while very rarely exploring new trajectories. By examining Fig. B.7 we can see that the entropy of the policy, when training without entropy regularization, converges towards 0.0 and at 20k iterations learning effectively stops. The agent always follows the same trajectory from a given state without exploring new paths. When training with entropy regularization the entropy of the policy converges towards around 0.7, which corresponds to the entropy of the distribution $\begin{bmatrix}0.5 & 0.5 & 0.0 & 0.0\end{bmatrix}$. This could be interpreted that on average the agent is deciding between two actions and chooses based on a coin-flip.

The relationship between the average entropy of the final policy and the entropy regu-

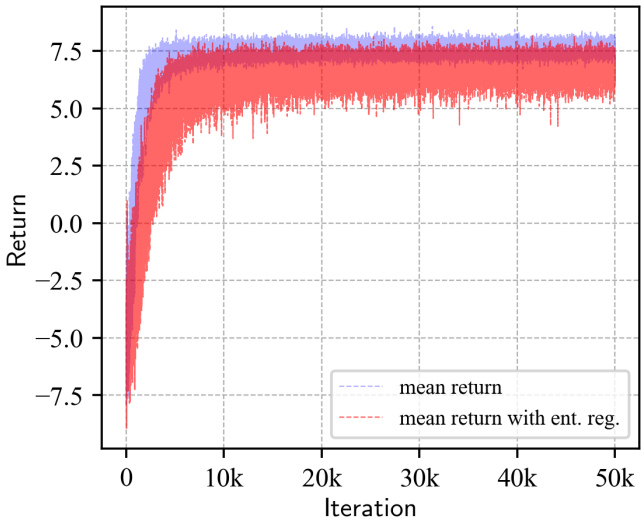


Figure B.6: Agent return averaged over the training batch. The return is calculated using eq. (B.2) without adding the entropy bonus. The value is plotted as a function of iteration count. Average returns increase faster and achieve higher results when training without entropy regularization. Training is performed with a batch size of 32 and a learning rate of 10^{-3}.

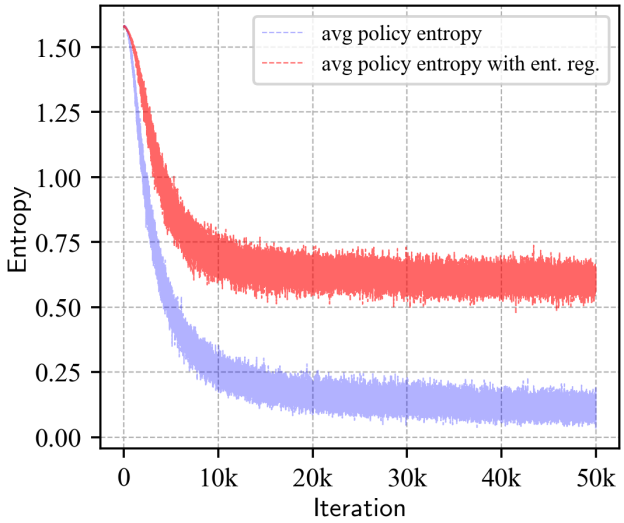


Figure B.7: Policy entropy averaged over all states of the environment. The value is plotted as a function of iteration count. When training without entropy regularization the entropy of the policy quickly converges to 0, which means the policy becomes a deterministic function. Training with entropy regularization keeps the entropy of the policy high, which means the policy remains a probabilistic function. Training is performed with a batch size of 32 and a learning rate of 10^{-3}.

larization temperature is shown on Fig. B.8. We can see that as we increase the value of the entropy regularization temperature parameter the entropy of the policy increases. if β^{-1} is chosen too low, the entropy will not play a significant role in the optimization and we may obtain a sub-optimal deterministic policy early during training as there was not enough exploration. if $\beta{-}1$ is too high, the entropy will dominate over the rewards received from the environment resulting in a purely random policy. The entropy regularization temperature parameter is yet another hyperparameter that additionally needs to

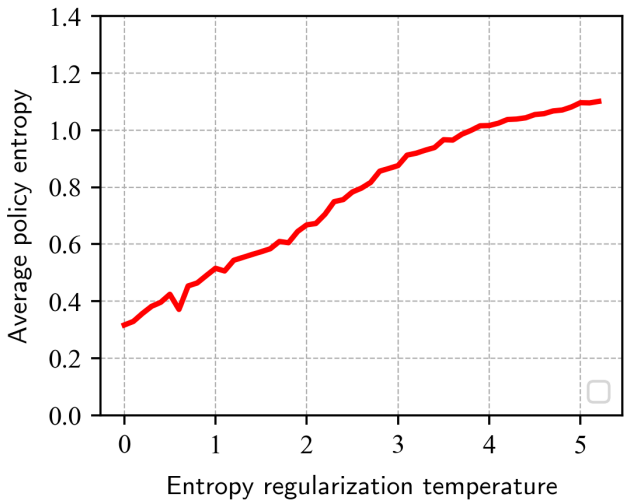


Figure B.8: Policy entropy achieved when training with different values of the entropy regularization temperature parameter. The entropy of the policy increases as we increase the value of the entropy regularization temperature parameter.

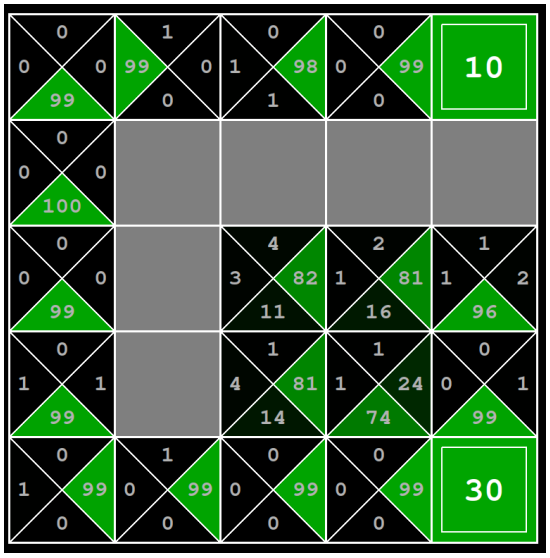


Figure B.9: Final policy for the MazeGrid layout without entropy regularization. The policy converges to a sub-optimal deterministic policy when training without entropy regularization. Training is performed with entropy regularization temperature of 0.

be scheduled. We would like to have more exploration at the beginning of training and less exploration at the end.

B.4.2 MazeGrid

To achieve the maximum return on the MazeGrid layout the agent **must** terminate at the bottom right end of the grid. The reward received from reaching this terminal state would outweigh any penalty accumulated during traversing the gridworld.

Figure B.9 shows the final policy learned by the agent when training without entropy regularization. We can see that the policy learned for the states at the top row of the

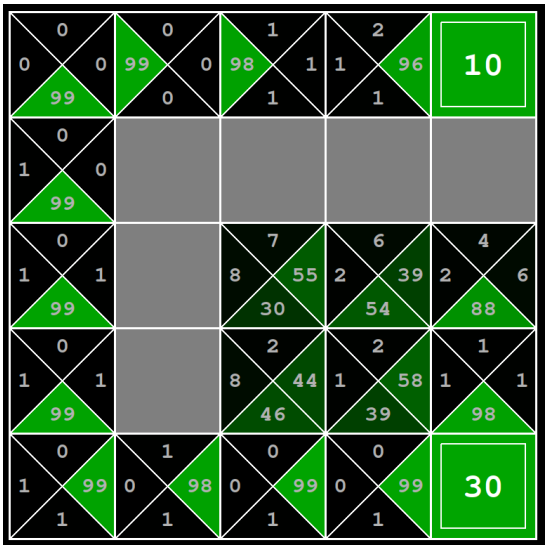


Figure B.10: Final policy for the MazeGrid layout with entropy regularization. The policy when training with entropy regularization converges to a better and non-deterministic policy achieving higher returns. Training is performed with entropy regularization temperature of 1.

grid is actually not optimal. The agent should be moving left and down towards the bottom right corner. Instead, however, the policy suggests that the agent should move right towards the upper right corner. The reason for this might be that these states are very close to the terminal state at the upper right corner. The length of the trajectory needed to terminate at the upper right is very small (1-3 steps) and the agent very often receives a positive feedback from following this path. To reach the bottom right corner, however, the agent has to travel a much longer trajectory and it reaches the terminal states very few times. This results in a much smaller amount of positive feedbacks.

Training with entropy regularization keeps the entropy of the policy high for the states for which the agent is still not certain which is the optimal action. This allows it to explore for a much longer period and, in turn, it allows it to arrive at a better policy achieving higher returns for this layout. The final policy learned by the agent when training with entropy regularization is shown in Fig. B.10.

Figure B.11 shows the average return obtained by the agent as a function of the training iteration. At first when training without entropy regularization the agent achieves higher returns. However without exploration it fails to improve its policy. Training with entropy regularization converges slower however the agent manages to find a better policy and eventually achieve a higher average return.

By examining Fig. B.12 we can see that both when training with and without entropy regularization, the entropy of the policy converges towards 0.0. This is due to the fact that for most of the states on this layout there exists a single optimal action, and thus the optimal policy must be deterministic for these states. However, training with entropy regularization allows the policy to remain non-deterministic for a longer period which in turn incentivizes the agent to explore a larger set of trajectories and eventually arrive at a better policy.

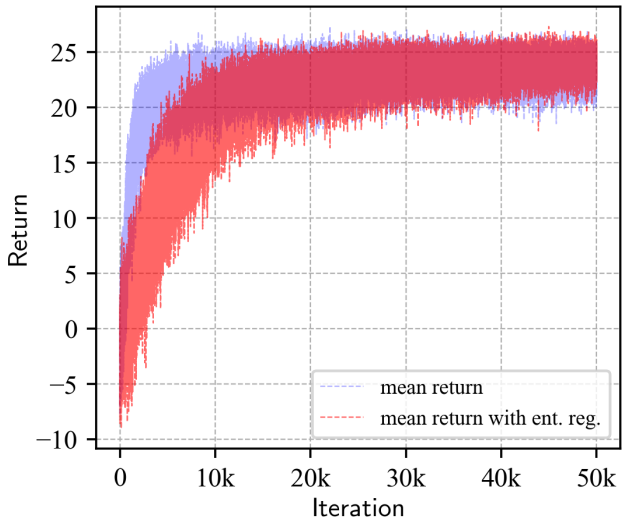


Figure B.11: Agent return averaged over the training batch. When training without entropy regularization the agent achieves sub-optimal returns. Training with entropy regularization allows the agent to explore more and achieve higher returns. The value is plotted as a function of iteration count. Training is performed with a batch size of 32 and a learning rate of 10^{-3}.

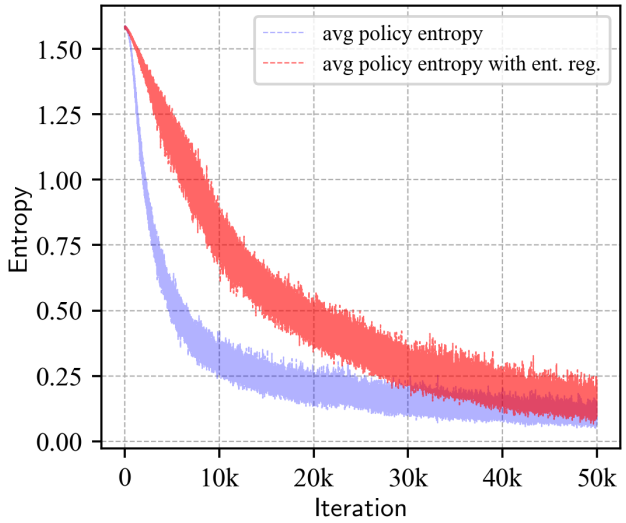


Figure B.12: Policy entropy averaged over all states of the environment. The value is plotted as a function of iteration count. Training both with and without entropy regularization the entropy of the policy converges to 0. Training with entropy regularization keeps the policy non-deterministic for a longer period allowing the agent to explore more. Training is performed with a batch size of 32 and a learning rate of 10^{-3}.

B.5 Conclusion

Policy gradient provides us with the means to train agents and find an optimal policy for problems for which we have no knowledge of the internal workings of the environment. Learning is achieved solely through trial and error.

Training with entropy regularization allows agents to arrive at solutions that remain probabilistic distributions and do not collapse into deterministic functions. This is helpful be-

cause, the policy remaining a probabilistic distribution increases the level of stochasticity and allows an agent to continue to explore even when the policy is almost optimal. If the policy becomes deterministic the agent might get stuck in a local extremum and never arrive at an optimal solution. Continuously exploring different trajectories allows agents to avoid local extrema.

Another reason for learning a stochastic policy is that, learning a deterministic policy only leads to a single optimal solution to the problem. Learning a stochastic policy forces the agent to learn many optimal solutions to the same problem: the agent is forced to learn as much information as possible from the experienced transitions.

It should also be noted that, we rarely have a Markov Decision Process. Most interesting problems are Partially Observed Markov Decision Processes (POMDPs), where the states are indirectly inferred through observations, and these observations can be probabilistic. In this setting the optimal policy to follow is a stochastic policy [52].

The idea of entropy regularization is regularly applied in bootstrapped methods [53] and the effects have been studied extensively [21, 54, 55]. However, entropy regularization can also be applied to Monte-Carlo methods in order to encourage exploration and improve performance. In this course project we show how the formula for applying entropy regularization is derived and we provide experiments to show the benefits of using it.

The idea of entropy regularization which is regularly used in bootstrapped methods can also be applied to simpler Monte-Carlo methods in order to encourage exploration and improve performance. The formula derived in Eq. (B.19) suggests that incorporating the additional regularizing term comes practically for free in terms of computational costs and the experiments described in Sec. B.4 show that using entropy regularization could improve the performance of the model in terms of optimality.

B.6 Integration

In this section we will disprove the following claim:

Claim: There exists a closed form solution for the anti-derivative of the following expression:

$$\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} \log \pi_\theta(a_{t'}|s_{t'}),$$

for any $T \in \mathbb{N}$.

Proof:
Let us denote $f_i(\theta) = \log \pi_\theta(a_i|s_i)$.We are now going to disprove the claim by providing a counter-example for the case when $T = 1$. We want to find the anti-derivative of:

$$\nabla_\theta f_0(\theta) \Big(f_0(\theta) + f_1(\theta) \Big) + \nabla_\theta f_1(\theta) f_1(\theta)$$

Assuming the anti-derivative exists and taking the integral of this expression we arrive

at:

$$F(\theta) = \int \left[\nabla_\theta f_0(\theta) \Big(f_0(\theta) + f_1(\theta) \Big) + \nabla_\theta f_1(\theta) f_1(\theta) \right] \mathrm{d}\theta,$$
$$F(\theta) = \frac{1}{2} f_0^2(\theta) + \frac{1}{2} f_1^2(\theta) + \int \nabla_\theta f_0(\theta) f_1(\theta) \mathrm{d}\theta,$$
$$\int \nabla_\theta f_0(\theta) f_1(\theta) \mathrm{d}\theta = F(\theta) - \frac{1}{2} f_0^2(\theta) - \frac{1}{2} f_1^2(\theta),$$

which is equivalent to the statement that the expression

$$\nabla_\theta f_0(\theta) f_1(\theta)$$

has a closed form solution for its anti-derivative for any two functions $f_0(\theta)$ and $f_1(\theta)$ – a contradiction

Bibliography

[1] Wikipedia. Stern–Gerlach experiment — Wikipedia, the free encyclopedia, 2022.

[2] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Phys. Rev.*, pages 777–780, 1935. doi:10.1103/PhysRev.47.777.

[3] J. S. Bell. On the einstein podolsky rosen paradox. *Physics Physique Fizika*, pages 195–200, 1964. doi:10.1103/PhysicsPhysiqueFizika.1.195.

[4] P. Grangier, G. Roger, and A. Aspect. Experimental evidence for a photon anticorrelation effect on a beam splitter: A new light on single-photon interferences. *Europhysics Letters*, pages 173–179, 1986. doi:10.1209/0295-5075/1/4/004.

[5] B. Kraus and J. I. Cirac. Optimal creation of entanglement using a two-qubit gate. *Phys. Rev. A*, page 062309, 2001. doi:10.1103/PhysRevA.63.062309.

[6] D. R. Terno. Nonlinear operations in quantum-information theory. *Phys. Rev. A*, pages 3320–3324, 1999. doi:10.1103/PhysRevA.59.3320.

[7] T. Mor. Disentangling quantum states while preserving all local properties. *Phys. Rev. Lett.*, pages 1451–1454, 1999. doi:10.1103/PhysRevLett.83.1451.

[8] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The Annals of Mathematical Statistics*, pages 1554–1563, 1966. doi:10.1214/aoms/1177699147.

[9] J. S. Denker, D. B. Schwartz, B. S. Wittner, S. A. Solla, R. E. Howard, L. D. Jackel, and J. J. Hopfield. Large automatic learning, rule extraction, and generalization. *Complex Syst.*, 1987. doi:https://www.complex-systems.com/abstracts/v01_i05_a02/.

[10] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, pages 210–229, 1959. doi:10.1147/rd.33.0210.

[11] C. J. Watkins and P. Dayan. Q-learning. *Machine Learning*, pages 279–292, 1992. doi:10.1007/BF00992698.

[12] G. Tesauro. Temporal difference learning and td-gammon. *Association for Computing Machinery*, pages 58–68, 1995. doi:10.1145/203330.203343.

[13] G. Tesauro, D. C. Gondek, J. Lenchner, J. Fan, and J. M. Prager. Simulation, learning, and optimization techniques in watson's game strategies. *IBM J. Res. Dev.*, pages 423–433, 2012. doi:10.1147/JRD.2012.2188931.

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. baker, M. Lai, A. Bolton, Y. Chen, T. P. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, pages 354–359, 2017. doi:10.1038/nature24270.

[15] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, pages 484–489, 2016. doi:10.1038/nature16961.

[16] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT Press, 2018. ISBN 9780262039246.

[17] J. Achiam. Spinning up in deep reinforcement learning (`https://spinningup.openai.com`, 2018.

[18] A. Salaün, Y. Petetin, and F. Desbouvries. Comparing the modeling powers of rnn and hmm. *ICMLA 2019: 18th International Conference on Machine Learning and Applications*, pages 1496–1499, 2019. doi:10.1109/ICMLA.2019.00246.

[19] K. J. Astrom and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008. ISBN 0691135762.

[20] L. Weaver and N. Tao. The optimal reward baseline for gradient-based reinforcement learning. *CoRR*, 2013. doi:10.48550/ARXIV.1301.2315.

[21] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, 2018. doi:10.48550/ARXIV.1801.01290.

[22] R. J. Williams and J. Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, pages 241–268, 1991. doi:10.1080/09540099108946587.

[23] 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. doi:10.5555/3454287.3455008.

[24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016. doi:10.5555/3026877.3026899.

[25] D. A. Pomerleau. ALVINN: an autonomous land vehicle in a neural network. *Advances in Neural Information Processing Systems 1*, pages 305–313, 1989. doi:10.5555/89851.89891.

[26] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to end learning for self-driving cars. *CoRR*, 2016. doi:10.48550/ARXIV.1604.07316.

[27] S. Ross, G. Gordon, and D. Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. *Proceedings of Machine Learning Research*, pages 627–635, 2011. doi:10.48550/ARXIV.1011.0686.

[28] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, 1968. doi:10.1109/TSSC.1968.300136.

[29] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang. Inconsistent heuristics in theory and practice. *Artificial Intelligence*, pages 1570–1603, 2011. doi:10.1016/j.artint.2011.02.001.

[30] N. S. Yanofsky and M. A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008. ISBN 0521879965.

[31] M. S. ANIS et al. Qiskit: An open-source framework for quantum computing (`https://qiskit.org/`). 2021. doi:10.5281/zenodo.2573505.

[32] W. Gerlach and O. Stern. Der experimentelle nachweis der richtungsquantelung im magnetfeld. *Zeitschrift für Physik*, pages 349–352, 1922. doi:10.1007/BF01326983.

[33] D. Deutsch, A. Barenco, and A. Ekert. Universality in quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, pages 669–677, 1995. doi:10.1098/rspa.1995.0065.

[34] D. P. DiVincenzo. Two-bit gates are universal for quantum computation. *Physical Review A*, pages 1015–1022, 1995. doi:10.1103/physreva.51.1015.

[35] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv*, 2014. doi:10.48550/ARXIV.1412.6980.

[36] S. Bozinovski. Reminder of the first paper on transfer learning in neural networks, 1976. *Informatica*, 2020. doi:10.31449/inf.v44i3.2828.

[37] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, 2018. doi:10.48550/ARXIV.1810.04805.

[38] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *CoRR*, 2013. doi:10.48550/ARXIV.1310.1531.

[39] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. doi:10.48550/ARXIV.1311.2524.

[40] X.-C. Wu, S. Di, E. M. Dasgupta, F. Cappello, H. Finkel, Y. Alexeev, and F. T. Chong. Full-state quantum circuit simulation by using data compression. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019. doi:10.1145/3295500.3356155.

[41] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. *International conference on machine learning*, pages 1889–1897, 2015. doi:10.48550/ARXIV.1502.05477.

[42] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. doi:10.48550/ARXIV.1707.06347.

[43] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv:1506.02438*, 2015. doi:10.48550/ARXIV.1506.02438.

[44] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine. Diversity is all you need: Learning skills without a reward function. *arXiv*, 2018. doi:10.48550/ARXIV.1802.06070.

[45] D. Yarats, R. Fergus, A. Lazaric, and L. Pinto. Reinforcement learning with prototypical representations. *arXiv*, 2021. doi:10.48550/ARXIV.2102.11271.

[46] R. Zhao, Y. Gao, P. Abbeel, V. Tresp, and W. Xu. Mutual information state intrinsic control. *arXiv*, 2021. doi:10.48550/ARXIV.2103.08107.

[47] H. Liu and P. Abbeel. Aps: Active pretraining with successor features. *arXiv*, 2021. doi:10.48550/ARXIV.2108.13956.

[48] T. L. Lai. Adaptive treatment allocation and the multi-armed bandit problem. *The Annals of Statistics*, pages 1091–1114, 1987. doi:10.1214/aos/1176350495.

[49] A. L. Strehl and M. L. Littman. An analysis of model-based interval estimation for markov decision processes. *Journal of Computer and System Sciences*, pages 1309–1331, 2008. doi:10.1016/j.jcss.2007.08.009.

[50] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos. Unifying count-based exploration and intrinsic motivation. *Advances in neural information processing systems*, 2016. doi:10.48550/ARXIV.1606.01868.

[51] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, pages 229–256, 1992. doi:10.1007/BF00992696.

[52] E. Todorov. General duality between optimal control and estimation. *2008 47th IEEE Conference on Decision and Control*, pages 4286–4292, 2008. doi:10.1109/CDC.2008.4739438.

[53] T. Haarnoja, A. Zhou, S. Ha, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. *CoRR*, 2018. doi:10.48550/ARXIV.1812.11103.

[54] B. Eysenbach and S. Levine. Maximum entropy RL (provably) solves some robust RL problems. *CoRR*, 2021. doi:10.48550/ARXIV.2103.06257.

[55] Z. Ahmed, N. L. Roux, M. Norouzi, and D. Schuurmans. Understanding the impact of entropy on policy optimization. *CoRR*, 2018. doi:10.48550/ARXIV.1811.11214.