

MATURA THESIS

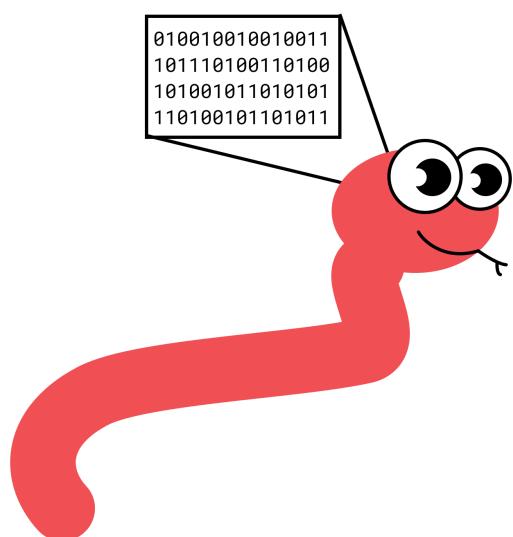
Teaching bots to play a game using artificial intelligence

APPLIED DEEP REINFORCEMENT
LEARNING IN A REALTIME GAME

By Shivram Sambhus
shivram.sambhus/at/gmail.com
Gymnasium Oberwil
Class 4e

Supervisor
Mr. Stefan Greising
Co-supervisor
Mr. Jonas Gloor

September 2022



Foreword

From a young age, I was interested in software-related topics. I learned how to program for the web and how to create scripts and mini-projects in Python. I tried my hand at various fields, but games and a subfield of machine learning called reinforcement learning particularly intrigued me. So I decided to try my hand at this area and familiarize myself with it a bit. As with learning and familiarizing myself with most topics, I researched the Internet to learn more about these new areas that had sparked my interest. I discovered how the use of games or simulations enables the application of reinforcement learning to real-world problems. Reinforcement learning can produce complex but intelligent strategies and behaviours. Examples include self-driving vehicles and recommender systems. But what intrigued me much more was the groundbreaking work of OpenAI. They had developed an AI called OpenAI five that plays the computer game Dota 2. Dota 2 is a famous multiplayer online battlefield computer game developed and distributed by Valve. This game is an exceptionally complicated role-playing and strategy game where two groups of 5 players each try to wipe out each other's base. OpenAI 5, which defeated the world champions in the Dota 2 tournament, was a testament to the power of reinforcement learning and caught the attention of the AI community by storm.

I was interested in doing something with computer science as my Matura thesis. The sheer amount of new advancements and impressive projects I had seen on the internet led me to an idea of developing a project encompassing three areas: website development, backend development and machine learning. So I came up with the idea of creating a browser-based real-time multiplayer snake game while building computer-controlled players who could effectively play this game using artificial intelligence.

During this project, I learned about various technologies and concepts applicable in real-world, while strengthening my foundations in programming and mathematics. I hope to give readers a brief insight into the world of computer science, game development, artificial intelligence and reinforcement learning.

Acknowledgements

First of all, I would like to thank my thesis supervisor Mr. Stefan Greising from the Gymnasium Oberwil for his technical support and guidance for which I remain always grateful. I am thankful to the extraordinary community on the Internet without whom I would not have been able to complete this project. I would also like to thank the people who contributed to the open-source projects and libraries I used. Finally, I would also like to thank my family for their support and encouragement during this project.

Contents

1	Introduction	5
2	Game	6
2.1	What is a bot?	6
2.2	Rules of the game	7
2.3	Multiplayer game	7
2.3.1	What is a client?	7
2.3.2	Structure and design of the game	7
2.3.2.1	Game Class	7
2.3.2.2	Snake Class	8
2.3.3	Communication	8
2.3.3.1	What are WebSockets?	8
2.3.3.2	Why use WebSockets?	8
2.3.4	Parallelism	9
2.3.4.1	What are threads?	9
2.4	Implementation	9
2.4.1	Game Server	10
2.4.1.1	What is Golang?	10
2.4.1.2	Why use Golang?	10
2.4.1.3	What is garbage collection?	10
2.4.1.4	Static Typing	11
2.4.1.5	Strong Typing	11
2.4.1.6	Compilation	11
2.4.2	Game Client	11
2.4.2.1	What is SvelteJS?	11
2.4.2.2	How was the game client created?	12
2.4.2.3	Rendering the game	12
2.4.3	Summary	12
3	Artificial Intelligence	13
3.1	History of Artificial Intelligence	13
3.2	Applications of Artificial Intelligence	15
3.2.1	Recommendation and Personalization Systems	15
3.2.2	Healthcare	15
3.2.3	Content Moderation	15
3.2.4	Maps and Navigation	15
3.3	Classification of Artificial Intelligence	16
3.3.1	Weak and Strong AI	16
3.3.1.1	Weak AI	16
3.3.1.2	Strong AI	16
3.3.2	4 tiers of Artificial Intelligence	16
3.3.2.1	Reactive Machines	17
3.3.2.2	Limited Memory	17

3.3.2.3	Theory of Mind	17
3.3.2.4	Self-Awareness	17
3.3.3	Subfields of Machine Learning	18
3.3.4	AI vs Machine learning vs Deep learning	18
3.3.5	Machine Learning subfields	18
3.3.5.1	Supervised learning	19
3.3.5.2	Unsupervised learning	19
3.3.5.3	Ensemble learning	19
3.3.5.4	Deep learning	20
3.3.5.5	Reinforcement learning	20
4	Deep Learning	21
4.1	Neural Networks	21
4.1.1	Single Perceptron	21
4.1.2	Multilayer Perceptron	22
4.2	Activation Functions	22
4.2.1	Binary Step Function	22
4.2.2	Sigmoid Function	23
4.2.3	Rectified Linear Unit (ReLU)	24
4.2.4	Leaky ReLU	25
4.2.5	Optimal Activation Functions	25
4.3	Loss Function	26
4.3.1	Mean Squared Error (MSE)	26
4.3.2	Adaptive Loss Functions	27
4.4	Optimization	28
4.4.1	Gradient Descent	28
4.4.2	Other Optimization Algorithms	28
4.4.3	Brute Force Optimization	28
4.4.4	Learning Rate	29
4.5	Essence of Deep Neural Networks	30
5	Reinforcement Learning	31
5.1	Introduction	31
5.2	Formalization of the Problem	31
5.3	Bellman Equation	33
5.4	Overview of RL Algorithms	33
5.4.1	Model-based RL	33
5.4.1.1	Value iteration	34
5.4.1.2	Policy iteration	35
5.4.2	Model-free RL	35
5.4.2.1	Q-Function	35
5.4.2.2	Monte Carlo Learning	35
5.4.2.3	Temporal Difference Learning	36
5.4.2.4	Q-Learning	37
5.4.2.5	Off-Policy vs. On-Policy	37
5.4.2.6	Summary	37
5.5	Deep Reinforcement Learning	38
5.5.1	Deep Policy Networks	38
5.5.2	Deep Q-Networks	38
5.5.3	Deep Dueling Q-Networks	39
5.5.4	Actor Critic Network	39
5.5.5	Advantage Actor Critic Network	40
5.5.6	Summary	41
6	Development of the RL Agent: AI and Results	42
6.1	Technologies	42

6.1.1	OpenAI Gym	42
6.1.2	Stable Baselines 3	42
6.1.3	PyTorch	43
6.1.4	TensorBoard	43
6.1.5	Matplotlib	43
6.1.6	Celluloid	43
6.1.7	NumPy	43
6.2	Local environment	43
6.3	Procedure and Initial Conditions	44
6.4	Experiments	45
6.4.1	Baseline experiment	45
6.4.2	DQN Baseline experiment	45
6.4.3	DQN with Framestacking	45
6.4.4	DQN with observation normalization	45
6.4.5	DQN with observation normalization and framestacking	46
6.4.6	DQN with observation normalization, framestacking and a larger network	47
6.4.7	Scheduled Learning Rate	47
6.4.8	Dynamic Exploration Rate	48
6.4.9	Observation Space Reduction by Feature Extraction	48
6.4.10	Advantageous Actor Critic (A2C)	49
6.4.11	Proximal Policy Optimization (PPO)	49
6.5	Results	50
6.5.1	Why did the DQN algorithm outperform the other algorithms?	50
6.5.2	Optimized Agent	51
6.6	Conclusion	52
6.6.1	Answering the problem statement	52
7	Discussion	53

Chapter 1

Introduction

People have long been fascinated by the possibility of building machines that mimic the human brain. The term "artificial intelligence" was coined by John McCarthy. Together with other computer scientists, he organized a conference in 1956 called the Dartmouth Summer Research Project on Artificial Intelligence. From these beginnings came the fields of machine learning, deep learning and deep reinforcement learning.

In this paper, I will convert the popular arcade game Snake into a browser-based multiplayer variant. Then, I will then create computer-controlled players (bots) that can play this game using reinforcement learning and deep learning.

The specific problem statement of this project is: *How to train bots with reinforcement learning to play a multiplayer game with the shortest possible training time?*

In the first part of this paper I will explain how I developed the game. In the next part, I will briefly discuss the history, applications, and types of artificial intelligence. After that, I will delve deeper and explain the concepts of reinforcement learning and deep learning. After that, I will explain how I trained the bots to play the game. Finally, I will end with a summary of the project and a discussion of the results.

Chapter 2

Game

The first part of this project is the development of the multiplayer video game "Snake". There are two main parts in the game of this project:

1. An online game server that runs the game and communicates with the browser game client.
2. An offline game that serves as an environment for training the RL bots.

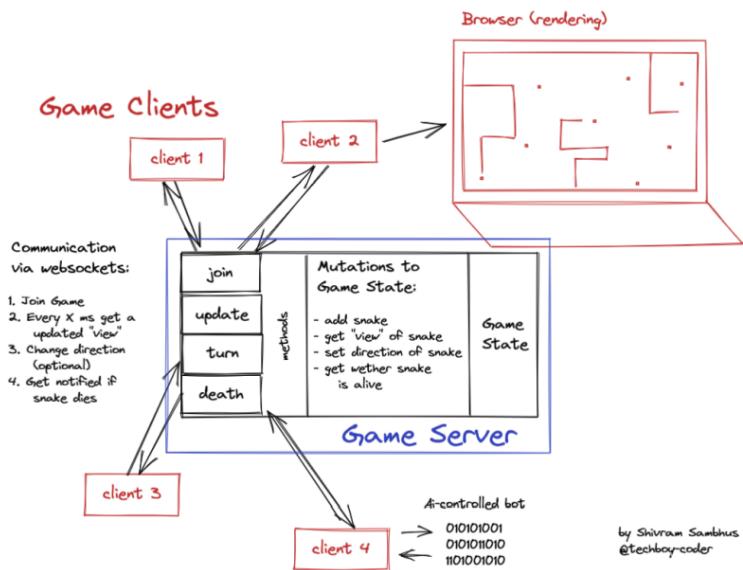


Figure 2.1: Overview of full game and RL bots

In the following sections I will explain what bots are, what the game rules are, describe the architecture of the multiplayer game and finally explain how I implemented the game in code.

2.1 What is a bot?

A bot is a player that is directly controlled by a computer. In real world, they are often used to improve the game experience for human players. For example, if there are too few players, bots pretending to be real players can make the game more interesting for real human players.

2.2 Rules of the game

The snake game version of this project is a multiplayer video game where the player controls a snake that consists of a series of blocks in a grid-like playing field. Each snake has only a limited view of the playing field, which is the area in which the snake can move. The snake can move in any direction, but not in the opposite direction of the previous movement. The snake tries to eat food blocks to grow longer, its body or the body of another snake being the obstacle it must not bump into, as well as the walls surrounding the playing field. The snakes compete to eat food in order to grow longer and obtain a higher rank. Snakes that die are partially converted into food and removed from the playing field, which encourages the snakes to kill other snakes.

2.3 Multiplayer game

Multiplayer games operate with a centralized game state stored on the game server. This allows a correct, secure, and synchronized representation of the game state to be sent to each client. Each client can send new actions (e.g. move the player) to the server. After all actions are collected from all clients, the server runs the game logic or environment physics that uses the current game state, actions, and game rules to update the game state.



Figure 2.2: Game logic & update

After each update, either the entire game state or a modified representation (e.g. limited view of the playing field; night vision; ...) of it can be sent to the clients.

2.3.1 What is a client?

A client (in computer science) is a program that uses services provided by other programs. In the case of the game client, the game client requests game data from the game server and provides input actions to the game server. The client can use the data received from the game server for various purposes. A game client can be a browser game client (e.g. the one created for this project) or a console game client (e.g. PS4 or Xbox) that provides interfaces for humans to play the game. However, game clients can also serve other purposes, such as providing an interface for a program or AI (e.g. a chess engine that impersonates a human in an online chess tournament) to play the game.¹

2.3.2 Structure and design of the game

In this section, I will explain the structure and design of the game. All entities in this game will be encapsulated in a class-like structure. This allows the game to be easily extended and modified in the future.

2.3.2.1 Game Class

My game class needs to keep track of all current snakes in the game and the entire playing field. The game state is represented as a 2D array of cells. This data structure was chosen for its simplicity and

¹ *What Is a Client?*

fast read and write operations (explained later). Each cell can either be empty, contain a snake, contain food, or contain a wall, each represented by a different number. The game class has some functions assigned to it. These functions are needed to run the game and to update the game state. They include the following:

updateSnakes()

This function updates snakes by calling their movement function, and eliminates them if they are dead (which is determined by the snake's movement function).

list(func(x,y))

These functions determine the state of the cell at a given position in the field. These functions set the state of the cell at a given position in the field, ensuring that the constraints² are met.

fieldsAround(x,y)

This function returns the state of the cells around a given position in the field, which is then passed to the clients.

2.3.2.2 Snake Class

Each snake is represented by a class. This class contains the current direction, a temporary direction to track direction changes (will be explained later), the snake's head coordinates, a linked list (will be explained later) with the snake's body coordinates, a boolean value to store whether it is alive or not, and finally a score to track the snake's performance. There are also some functions assigned to the snake class. These functions are needed to update the snakes position, change its direction, make it grow when it takes food, and eliminate it when it dies. They include the following functions:

move()

This function updates the snake's position by setting the current direction to the current temporary direction (see `setDirection`), moving the last item in the body list to the top of the list, and then updating the head position. This function also checks for collisions with food, walls, and snakes.

setDirection(direction)

This function checks if the new direction is valid (not the opposite direction of the current direction & in the range of valid directions), and if so, sets it to the temporary direction (since many update requests try to change the direction between game update steps).

createBody()

This function is called when eating, and adds a body to the snake.

toFood()

This function is called when the snake dies, and randomly converts any part of the snake's body into food.

2.3.3 Communication

The communication between the game server and the game client is done via WebSockets.

2.3.3.1 What are WebSockets?

WebSocket is the name of a Transmission Control Protocol (TCP)-based network protocol that can be used to establish bidirectional connections between web applications and WebSocket servers. Once a WebSocket connection is established, data can flow in both directions without the need for additional requests or procedures to establish the connection.³

2.3.3.2 Why use WebSockets?

WebSockets make information exchange more efficient and faster. Unlike pure HTTP communication, no client-side request is required to transfer new data from the server. The underlying TCP connec-

²Such as: input coordinates x and y cannot be larger than the field itself

³“WebSocket”.

tion persists throughout WebSocket communication. WebSocket can be used to implement real-time applications like games between a client and a server.⁴

2.3.4 Parallelism

My game server has 3 main tasks:

1. Update the game state.
2. Send the game state to the clients.
3. Listen for new actions from the clients.

Some of these tasks must run in parallel to ensure that the game state is always up to date, that the game does not get stuck if a client is slow or abruptly disconnects, that updates are registered by all clients, and that updates to the game are sent to each client. All of these tasks are blocking tasks, which means that they wait until all other tasks have finished before they can continue. This is a problem because the game is constantly being updated and waiting for new actions at the same time, so no new actions can be received while the game is being updated. For this reason, these two tasks are run in parallel on separate threads. See image 2.3 for a visualization of the threads.

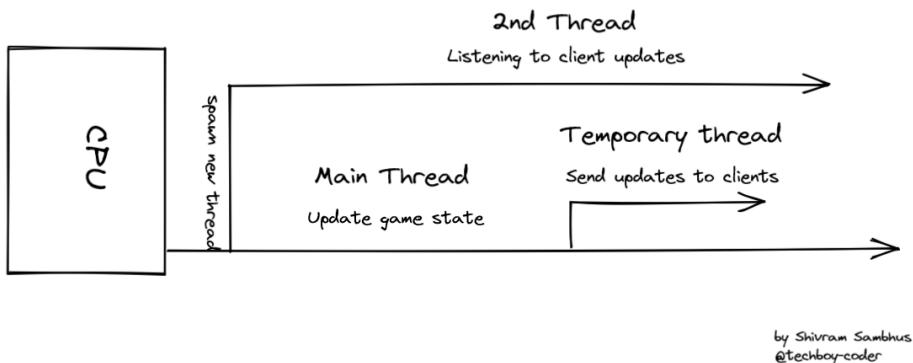


Figure 2.3: Visualization of the threads for my realtime snake game

2.3.4.1 What are threads?

A thread is a collection of instructions that are to be executed by the CPU separately from other processes. For example, a program may have open threads waiting for a network request so that the main program can perform other tasks.⁵

In this case, we have a thread that updates the game state and sends it to the clients, and a thread that waits for new actions from the clients. These two threads are independent of each other and run in parallel.

2.4 Implementation

In order to have a working game, the game structure and design needed to be translated into code that could be executed on a computer. I will explain how the game server and clients were created in the following sections.

⁴Blog, *When to Use a HTTP Call Instead of a WebSocket (or HTTP 2.0)*.

⁵What Is a Thread?

2.4.1 Game Server

The game server is a program that runs on a computer and is responsible for updating the game state and sending the game state to the clients while waiting for new actions from the clients. The game server must be:

1. Be able to handle many TCP connections.
2. Be able to perform calculations to update the game state.
3. Ideally easy to learn, use, maintain and extend.

For the game server I chose to use Golang.

2.4.1.1 What is Golang?

Golang (or Go) is an open source programming language developed in 2009 by Robert Griesemer, Rob Pike and Ken Thompson at Google. Go compiles like C/C++ and is static, but has garbage collection and is strongly typed.⁶

2.4.1.2 Why use Golang?

I chose Go because of its simplicity, consistency, power, readability and concurrency. Along with a large community of developers using Go, and a large number of tutorials and the popularity of this language, it would be practical to learn and use Go. Go is also used by some of Google's production systems, and the popular open source projects Docker and Kubernetes are also programmed in Go.⁷



Figure 2.4: Golang Logo and Mascot⁸

2.4.1.3 What is garbage collection?

Garbage collection is the mechanism that frees the memory that is no longer needed. In some programming languages, this task is left to the programmer. When you need memory (to store data), you request it from the operating system, and when you no longer need it, you release it. This puts a lot of responsibility on the programmer; if you don't free the memory, a memory leak will occur. If you accidentally keep a reference to freed memory, a dangling pointer is created, which can be a security vulnerability.⁹

A garbage collector is a system that does all these tasks for you. Whenever you create an object, the program automatically keeps track of all references to that object. When the object is no longer referenced anywhere, it automatically frees it so that the memory is no longer occupied. This way the user (mostly) doesn't have to worry about memory resources.¹⁰

⁶“Go (Programming Language)”.

⁷ *What Is Go?*

⁸start et al., *Golang Logo » Open Up The Cloud*

⁹“Garbage (Computer Science)”.

¹⁰ *Garbage Collection Comp Sci Wiki - Google Search.*

2.4.1.4 Static Typing

Static typing means that the variable types in the program are known at compile time. Dynamic typing, on the other hand, means that the value types are checked during execution. A badly typed operation can cause the program to stop or report an error at runtime.¹¹

2.4.1.5 Strong Typing

Strong typing generally means that there are no loopholes in the type system, while weak typing means that the type system can be subverted (invalidating all guarantees).¹²

2.4.1.6 Compilation

Compilation is the creation of an executable program from code written in a compiled programming language. Compilation allows the computer to run and understand the program without the programming software used to create it.¹³

2.4.2 Game Client

The game client, as explained above, is a program that runs on the players computer and is responsible for displaying the game on the screen and processing user input. The requirements for the game client in this project are:

1. Light, which means that the program is easy on system resources.
2. Simple, which means that there is no need to download any software.
3. Cross-platform, which means that the program can be run on any computer with little or no changes.

Using a web browser to run and render the game client is the best choice for this part of the project because it is very lightweight and easy to use, the player enters the game simply by visiting the website instead of downloading the game, and it is cross-platform. For the development of the game client, I chose SvelteJS, a JavaScript framework that is very easy to use and has a very small footprint (meaning it's easy on system resources).

2.4.2.1 What is SvelteJS?

Svelte is a component-oriented JavaScript library like React and VueJS.¹⁴ Like most popular frontend JS libraries, Svelte provides a lightweight framework with powerful features and syntax sugars that make developers' jobs easier. The main difference with Svelte is the engine in the library, as Svelte is primarily a compiler.¹⁵



Figure 2.5: Svelte JS Logo¹⁶

¹¹ *What's the Difference between Being Statically versus Strongly Typed?*

¹² *Ibid.*

¹³ *What Is Compile?*

¹⁴ *Svelte • Cybernetically Enhanced Web Apps.*

¹⁵ *Svelte, Why so Much Hype ?*

¹⁶ M. Gupta, *Getting Started with Svelte*

2.4.2.2 How was the game client created?

The game client is a single page application which either renders the welcome screen or the game screen. In the background the client handles connecting to the server, sending and receiving data, and updating the game state.¹⁷

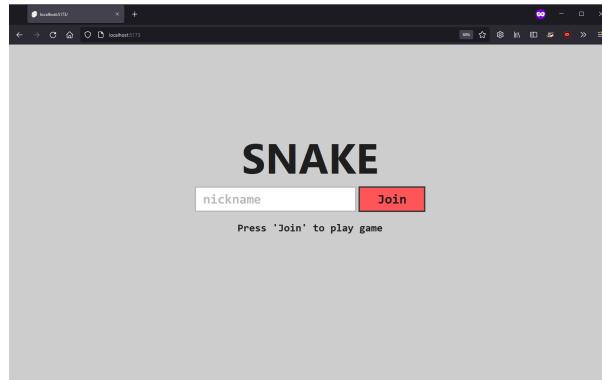


Figure 2.6: Snake Game Homepage in Firefox

2.4.2.3 Rendering the game

The most important part of the data that the client receives from the server as an update is the representation of the game (view of the snake). This view is sent as a 2D array of numbers, where each number represents the type of cell (e.g. 0 = empty, 1 = food, 2 = snake, -1 = wall). The rendering is done using the Canvas element. The Canvas element is part of HTML5 and enables dynamic, scriptable rendering of 2D shapes and bitmap images. It is a low-level procedural model that updates a bitmap.¹⁸ HTML5 Canvas also helps in creating 2D games (which is why we are using it).¹⁹

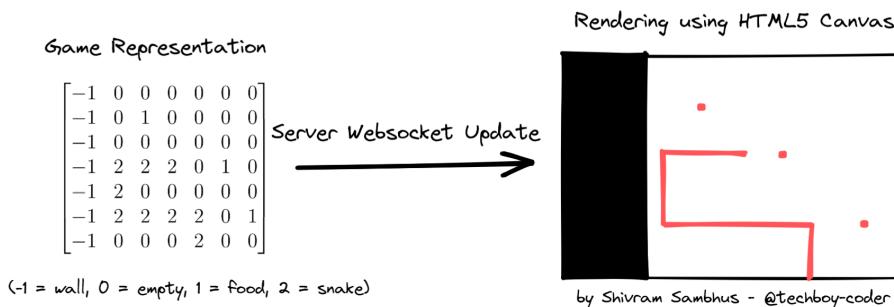


Figure 2.7: Visualization of the rendering of the game from raw data

2.4.3 Summary

In this chapter, I have described and explained the requirements and implementation of the game server and browser-based game client. The product of this chapter is a fully functional real-time multiplayer game that can be deployed and run on any computer.

¹⁷See image 2.6 for viewing the browser client homepage

¹⁸“Canvas Element”.

¹⁹See image 2.7 for an example of how raw data is rendered via the canvas element

Chapter 3

Artificial Intelligence

Artificial intelligence (AI) refers to technologies that enable computers to help humans solve tasks that require intelligence. An important subset of AI is "machine learning," which has its origins in statistical and data-driven methods. In machine learning and deep learning, which we will discuss later, a computer autonomously learns how to recognize patterns and regularities in data sets.

Such methods can provide valuable results, especially when dealing with very large or complex data sets or problems. Artificial intelligence methods complement the creativity of researchers and often provide surprising suggestions that had not been thought of before. AI is an attempt to transfer human learning and thinking to computers and give them intelligence. Rather than being programmed for a specific purpose, an AI can use machine learning to find answers and solve problems on its own.

In the following sections, we will take a look at the history of artificial intelligence, where AI is being used today, and the different types of AI that exist.

3.1 History of Artificial Intelligence

The beginnings of artificial intelligence can be traced back over decades. The question of whether computers can be intelligent in the sense in which we speak of human intelligence has been at the heart of computer science as a discipline from the very beginning. In 1950, Alan Turing proposed the "imitation game", a concept that later became known as the "Turing test"¹. According to him, a computer could be called "intelligent" if it could not be distinguished from another human in a conversation.²

In the summer of 1956, scientists gathered for a conference at Dartmouth College in New Hampshire. They believed that aspects of learning and other features of human intelligence could be simulated by machines. Programmer John McCarthy proposed the term "artificial intelligence" to describe this. By the time of the conference, the world's first artificial intelligence program, Logic Theorist, had been written and could prove several dozen mathematical theorems.³

In 1966, German-American computer scientist Joseph Weizenbaum of the Massachusetts Institute of Technology (MIT) invented a computer program called "ELIZA" that communicates with humans. Using scripts, "ELIZA" simulates various human communicators, such as a psychotherapist.⁴ In the 1980s, a form of artificial intelligence called "expert systems" was adopted by companies around the world, and knowledge became the focus of artificial intelligence research. Expert systems are computer programs that use formulas, rules, and a knowledge base to generalize knowledge in a particular area. For example, artificial intelligence had entered the medical field, where it is used to aid in diagnosis and treatment.

¹The Turing Test is a method in artificial intelligence that can be used to determine whether or not a computer is capable of thinking like a human. The test is named after Alan Turing. *What Is the Turing Test?*

²"Turing Test".

³"ELIZA".

⁴Ibid.

⁵ibid.

```
Welcome to
      EEEEEE  LL     IIII    ZZZZZZ  AAAAAA
      EE     LL     II     ZZ     AA     AA
      EEEEEE  LL     II     ZZZ    AAAAAAAA
      EE     LL     II     ZZ     AA     AA
      EEEEEE  LLLLLL  IIII   ZZZZZZ  AA     AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU: Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU: They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU: Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU: He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU: It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

Figure 3.1: Example of an ELIZA conversation⁵

In 1997, artificial intelligence first attracted public attention. IBM’s artificial intelligence chess machine Deep Blue defeated world chess champion Garri Kasparov in a tournament. This is considered a historic success for machines in a field previously dominated by humans.⁶

In the 2000s, the world of artificial intelligence began to develop rapidly thanks to technology companies such as Google, Facebook, Apple and others, as they began to invest heavily in artificial intelligence and related research and scientific talent. This was largely thanks to the advanced computing power of computers, which enabled the widespread use of artificial intelligence. One example is the artificial intelligence developed by IBM called Watson, which understands our language and can answer difficult questions very quickly. In 2011, Watson finally managed to beat human contestants on the American TV quiz show Jeopardy and impress the public.⁷ In 2016, Google’s artificial intelligence "AlphaGo" caused a sensation when it managed to beat the world’s best Go player, which until then had been considered almost impossible due to the game’s high complexity.⁸ Since the 2010s, AI has been gaining momentum

Figure 3.2: AlphaGo vs. Lee Sedol (the world’s best Go player)⁹

and popularity. The world is now using artificial intelligence to solve problems that are not directly related to human intelligence.

⁶IBM100 - Deep Blue.

⁷"History of Artificial Intelligence".

⁸AlphaGo.

⁹Hern, "AlphaGo"

3.2 Applications of Artificial Intelligence

In this section, we will take a look at some applications of artificial intelligence. Artificial intelligence already plays an important role in most people's daily lives. Often, we are not even aware of how often we already use AI in our daily lives. In the following part we will look at some examples.

3.2.1 Recommendation and Personalization Systems

Intelligent recommendation and personalization systems use artificial intelligence to learn more and more about our behavior and tailor recommendations to our interests. This creates a personalized experience that constantly learns and improves through our own online behavior. Examples include Netflix suggesting new shows and Amazon showing us which products might match our last purchase. The algorithms of Google and the likes work in a similar way, providing us with an optimized web experience. Here, AI serves as quality control and guarantees high-quality content at the top of the search results.¹⁰

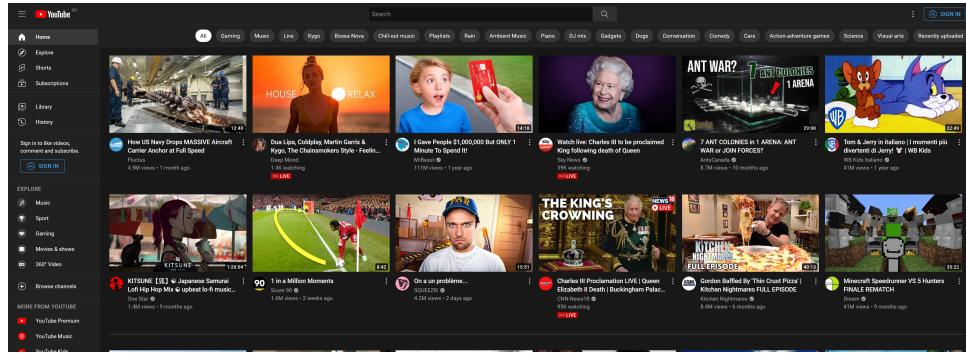


Figure 3.3: Example of YouTube's recommendation system (YouTube algorithm)¹¹

3.2.2 Healthcare

Proper diagnosis of disease requires years of medical training. But even then, diagnosis is often a laborious and time-consuming process. In many areas, the demand for experts far exceeds the available supply. This, in turn, increases the pressure on doctors and not infrequently delays life-saving patient diagnostics. Machine learning, especially deep learning algorithms, have recently made great strides in automatically diagnosing diseases using scans, images and patient data.¹²

3.2.3 Content Moderation

In social media or news forums, moderation is a major challenge. Identifying and removing trolls, insults, and prohibited content is often beyond the scope of human manpower. AI-powered systems can automatically analyze and classify potentially dangerous content, thereby improving the speed and efficiency of the whole moderation process.¹³

3.2.4 Maps and Navigation

Before the use of artificial intelligence, route planning was very tedious for humans. Thanks to Google Maps and other services, trip planning is no longer a problem. The artificial intelligence behind the apps helps plan the optimal route, and user data helps predict traffic jams and problems. Traffic problems

¹⁰ Top Applications of Artificial Intelligence (AI) in 2022.

¹¹ YouTube

¹²Rangaiah, Artificial Intelligence in Healthcare.

¹³Darbinyan, Council Post.

are predicted. User experience is at the heart of the application.¹⁴ Similar technology is behind Uber and other ride-sharing apps, helping to identify the optimal route and the right vehicle.¹⁵

We have looked at just a few examples. There is much more AI in our daily lives than meets the eye. It is used to help us, to make our lives easier and safer, or simply to learn.

3.3 Classification of Artificial Intelligence

In this section, we will take a look at the two different ways in which artificial intelligence can be classified.

3.3.1 Weak and Strong AI

The first way to classify AI is to divide it into: weak and strong artificial intelligence.

3.3.1.1 Weak AI

Weak AI is a type of artificial intelligence that is specifically designed to focus on a particular task and appear very intelligent. It lacks creative capabilities and has no apparent ability to learn on its own in a universal sense.¹⁶

A very good example of weak AI is Apple's Siri, which is backed by the Internet as a powerful database. Siri appears to be very intelligent, as it can carry on conversations with real people and even make a few little remarks and jokes, but in reality it functions in a very narrow, predetermined way. However, the 'limits' of its function are evident in the inaccurate results when it engages in conversations for which it is not programmed.¹⁷

Robots used in manufacturing can also appear very intelligent because of their precision and the fact that they perform very complex actions that appear incomprehensible to the normal human mind. But that is the extent of their intelligence; they know what to do in the situations for which they have been programmed and do not have the ability to decide what to do beyond that. Even an artificial intelligence that is capable of machine learning can only learn and apply what it has learned to the extent for which it has been programmed.¹⁸

3.3.1.2 Strong AI

On the other hand, the realization of strong AI is not yet within reach: The goal of the strong AI concept is for natural and artificial intelligence carriers (e.g. humans and robots) to build a common understanding and trust when working in the same field of action.

For example, efficient human-machine collaboration could be learned and enabled. A strong AI can autonomously recognize and define tasks and acquire and build knowledge about the corresponding application domain to do so. It investigates and analyzes problems to find an adequate solution which can also be new or creative.¹⁹

It is still disputed whether the development of such intelligence is even possible. However, the majority of researchers now agree that powerful AI will be developed, but there is no consensus on when this will happen. However, a time span of 20 to 40 years is considered realistic.²⁰

3.3.2 4 tiers of Artificial Intelligence

The second way to classify AI is to divide it into four tiers. The four tiers are: reactive machines, limited memory, theory of mind, and self-awareness.

¹⁴ *A Smoother Ride and a More Detailed Map Thanks to AI.*

¹⁵ *Uber AI in 2019.*

¹⁶ experience et al., *Difference Between Strong and Weak AI / Difference Between.*

¹⁷ *What Is Weak Artificial Intelligence (Weak AI)?.*

¹⁸ *Weak vs. Strong AI.*

¹⁹ experience et al., *Difference Between Strong and Weak AI / Difference Between.*

²⁰ *What's the Difference between Being Statically versus Strongly Typed?*

3.3.2.1 Reactive Machines

Reactive machines AI is the first and most basic form of artificial intelligence. These machines have no memory or perception of the world or time and are designed to perform a single task depending on the current situation.

An example of reactive AI is Deep Blue, IBM's chess-playing supercomputer that defeated international grandmaster Garri Kasparov in the late 1990s.²¹ Deep Blue can identify pieces on a chessboard and knows how they move. It can make predictions about what moves might follow for him and his opponent. And it can choose the most optimal moves among the possible ones. But it has no knowledge of the past and no memory of what happened before. Apart from the rarely used chess-specific rule of not repeating the same move three times, Deep Blue ignores everything that happened before the present moment. It simply looks at the pieces on the chessboard and chooses from the possible next moves. In this type of intelligence, the computer perceives the world directly and responds to what it sees. It does not rely on an internal concept of the world.²²

3.3.2.2 Limited Memory

Unlike reactive machines, AIs with limited memory are able to apply collected data from past situations to current events and incorporate it into their decisions. Machines with limited memory must relate what they have learned to current events in order to make a decision.

Modern self-driving cars are based on this system. The computer in these cars knows from memory how cars normally drive, what people or cyclists look like, and what the traffic rules are. At the same time, it watches its surroundings for obstacles such as other cars, trees or people that complete its picture of the world.

This is the most common form of artificial intelligence today: it is used by major technology companies such as Google, Facebook and Amazon for tasks such as speech recognition, image recognition and translation. It is also used in selfdriving cars and in many other applications.²³

3.3.2.3 Theory of Mind

These artificial intelligence systems represent a very advanced technology that is able to interpret the environment and the things in it. This type of artificial intelligence should be able to understand people's feelings, thoughts, beliefs and expectations and interact in a social environment. This requires a deep understanding of how humans and other living things in the environment change their feelings and behavior.²⁴

R2-D2 from Star Wars is an example of the intelligent robot theory, as he was able to understand the fear and anger of other humans and react fearfully or angrily in certain situations.

This type of AI still presents a challenge. Researchers struggle to mimic what happens in the human brain. Social norms, emotions, and instincts create many variables that are difficult to replicate in a machine.²⁵

3.3.2.4 Self-Awareness

The highest form of artificial intelligence describes machines that have consciousness and self-awareness and are sentient. In addition, these machines should be able to show a desire for certain things and recognize their own feelings. At this level, computer thinking reaches the level of human consciousness with a complete perception of the world, human feelings, intentions and reactions.²⁶

This AI will move from "I think" to "I know I think." Robots with this AI will be as intelligent or even more intelligent than humans and will be able to do all the normal tasks we do today if not better. This

²¹ IBM100 - Deep Blue.

²² Understanding the 4 Types of Artificial Intelligence (AI).

²³ Understanding the Four Types of Artificial Intelligence.

²⁴ Johnson, 4 Types of Artificial Intelligence.

²⁵ Understanding the 4 Types of Artificial Intelligence (AI).

²⁶ Understanding the Four Types of Artificial Intelligence.

form of AI does not yet exist, but experts agree that this invention, if ever realized, will be one of the greatest milestones in the history of artificial intelligence.²⁷

3.3.3 Subfields of Machine Learning

In this section, we will clarify some terms and then take a look at the different subfields of machine learning.

3.3.4 AI vs Machine learning vs Deep learning

In social media and news sites, the terms artificial intelligence, machine learning, and deep learning have become confused. To avoid confusion, we will clarify some terms in this subsection.

Artificial intelligence is the ability of a computer system to mimic human cognitive functions such as learning and problem solving. In AI, a computer system uses mathematics and logic to simulate the thinking that humans use to learn from new information and make decisions.²⁸

Machine learning is a subset of AI. It uses mathematical data models such as regression, clustering, decision trees, and various other models that allow a computer to learn without direct instructions. In this way, a computer system can continuously learn and improve based on experience.

Deep learning is a subset of machine learning. Deep learning works in a similar way, which is why the two terms are often confused. Deep Learning algorithms are based on neural networks (which we will discuss in more detail later).²⁹

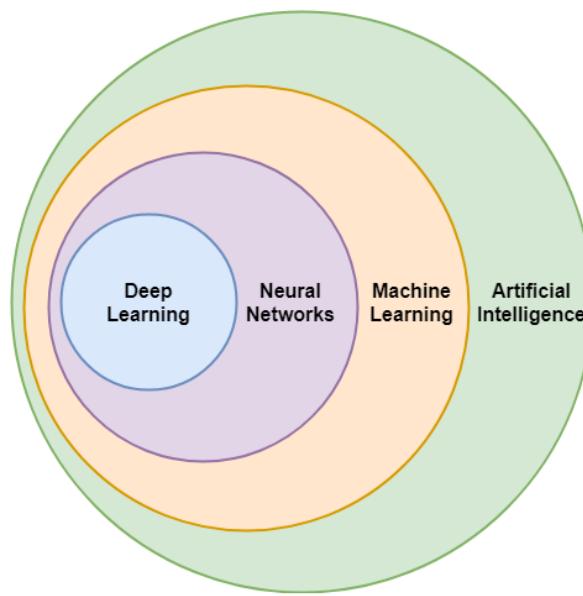


Figure 3.4: Artificial Intelligence subtopics³⁰

3.3.5 Machine Learning subfields

Now, that we have distinguished artificial intelligence from machine learning, we'll take a look at the different machine learning subfields. There are 5 main machine learning subfields: Supervised learning, Unsupervised learning, Ensemble learning, Reinforcement learning, and Deep learning.³¹

²⁷Computer, *What If AI Becomes Self-Aware?*

²⁸Understanding The Difference Between AI, ML, And DL.

²⁹Kalavala, *AI, ML and DL*.

³⁰Silvan, *What Is the Difference between Artificial Intelligence, Machine Learning and Deep Learning?*

³¹“Outline of Machine Learning”.

³²Trekhleb, *Homemade Machine Learning in Python*

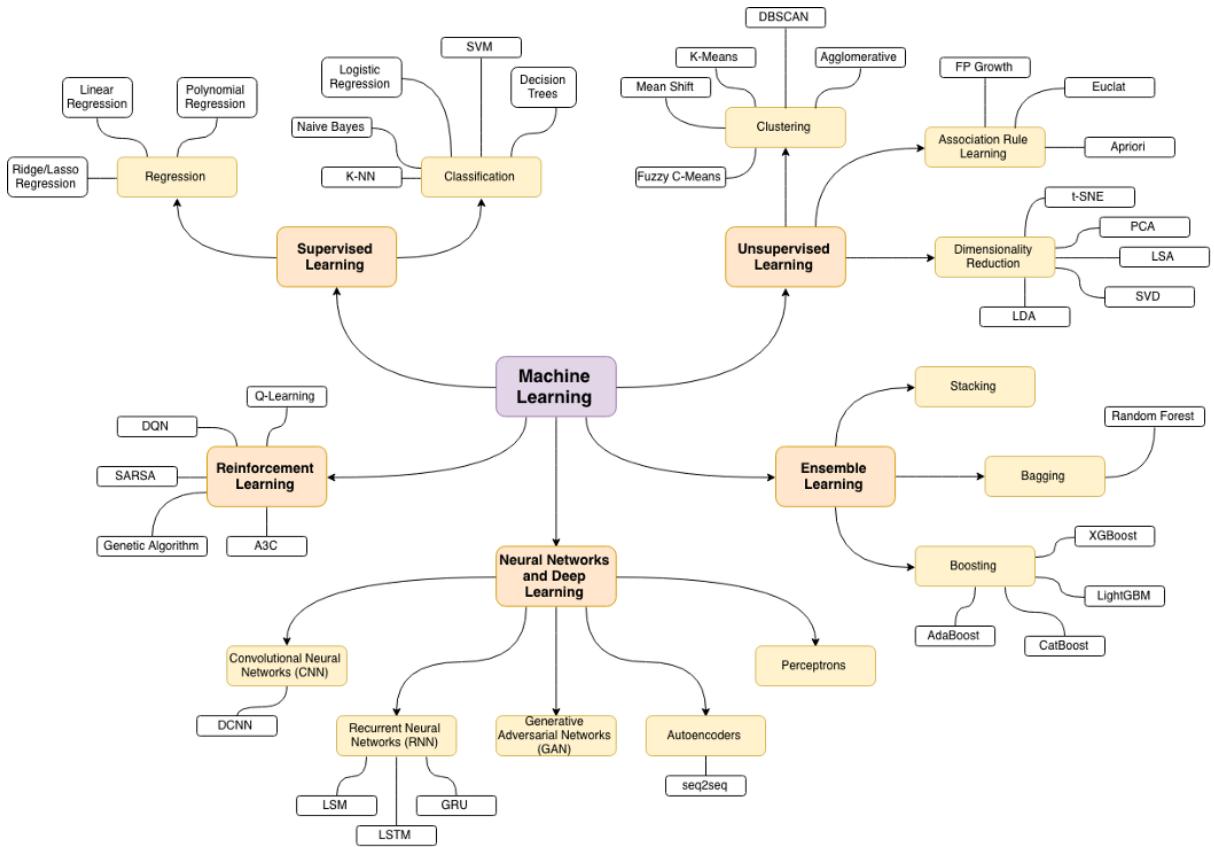


Figure 3.5: Overview of the different machine learning subfields³²

3.3.5.1 Supervised learning

Supervised learning is a machine learning technique in which the algorithm is presented with a data set where the target variable is already known. The algorithm learns relationships and dependencies in the data that explain these target variables. After training, the quality of the prediction is evaluated, and then the learned patterns are applied to unknown data to produce forecasts and predictions.³³

Supervised learning algorithms include regression (e.g. linear regression) and classification algorithms (e.g. support vector machines or decision trees).

3.3.5.2 Unsupervised learning

Unsupervised learning methods are trained in the absence of existing knowledge. Unlike supervised learning methods, there is no data sample with labels, instead, the model must recognize the model itself. This type of method is often used for clustering tasks, such as classifying existing data into groups. This is the case, for example, in marketing, where existing customers are classified into meaningful groups and then advertisements are created specifically for those groups.³⁴ Unsupervised learning algorithms include clustering (e.g. K-means) and dimensionality reduction (e.g. principal component analysis).³⁵

3.3.5.3 Ensemble learning

Ensemble learning models use multiple models to combine the results of the models. This is done, for example, by averaging the results of the models. An example of ensemble learning is the random forest algorithm, which combines the results of multiple decision trees.³⁶

³³Yalçın, *4 Machine Learning Approaches That Every Data Scientist Should Know*.

³⁴Trekhleb, *Homemade Machine Learning in Python*.

³⁵Deep Learning vs. Machine Learning.

³⁶Yalçın, *4 Machine Learning Approaches That Every Data Scientist Should Know*.

3.3.5.4 Deep learning

Deep Learning is currently one of the most exciting areas of research in machine learning. It is based on the use of artificial neural networks. Artificial neural networks are algorithms modeled on the biological model of the human brain. They are used to recognize patterns, interpret text, or help us form clusters and classify objects on images. Of course, like any machine learning algorithm, a deep learning algorithm is trained using data. Artificial neural networks are often very complex, making it difficult to interpret individual decisions. Since Deep Learning plays an important role in this project, we will look at it in more detail in a later section.³⁷

3.3.5.5 Reinforcement learning

Reinforcement learning is another type of learning method. In this case, the model is trained with the help of a reward. This is necessary when there is no given data set and the model has to learn from self-generated data. This approach is mostly used to train AI for playing video games or interacting with dynamic systems. Such models are also used in other fields such as robotics or finance. Since the focus of this paper is on reinforcement learning, we will discuss it in more detail later in this paper.³⁸

In this chapter, we have taken a general look at the history and the different types and subfields of artificial intelligence. In the next chapter, we will take a closer look at Deep Learning and Reinforcement Learning.

³⁷ What Is Deep Learning? / How It Works, Techniques & Applications - MATLAB & Simulink.

³⁸ What Is Reinforcement Learning?

Chapter 4

Deep Learning

We will now look at Deep Learning in more detail, as it plays an important role in our project. As explained above, Deep Learning is a subfield of machine learning based on the use of neural networks. Just like the biological neural network, the artificial neural network continuously learns and updates its knowledge and understanding of the environment based on the experiences it has encountered. Each deep learning algorithm usually consists of the following components: a neural network, an activation function, a loss function, and an optimization algorithm.

4.1 Neural Networks

A neural network is a mathematical model of the human brain. It consists of a series of interconnected neurons. Each neuron, much like a biological neuron, can store a state or value and has a series of incoming and outgoing connections. To understand how a multilayer (or deep) neural network works, we need to start small and approach slowly. We will start with a simple perceptron and build our knowledge from there to a multilayer perceptron.

4.1.1 Single Perceptron

A single perceptron is the basic building block from which most further multilayer perceptrons (or deep neural networks) are constructed. The perceptron computes a linear regression for a given input vector \mathbf{x} using a weight vector \mathbf{w} and a bias vector \mathbf{b} , which is then passed through an activation function ϑ to produce an output \hat{y} .¹ The perceptron shown in the figure above is a feedforward neural network.

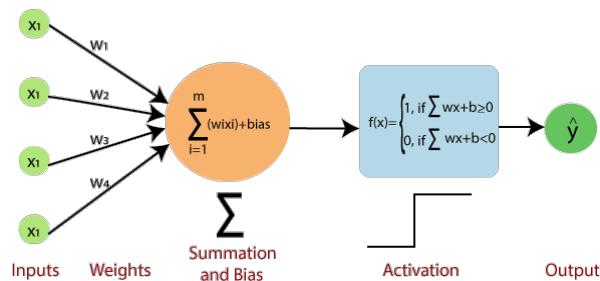


Figure 4.1: Single layer perceptron²

However, there are also variants such as the recurrent neural network (RNN) or long-term memory neural networks (LSTM) that can be used to model time-dependent data.

¹"Perceptron".

²Single Layer Perceptron in TensorFlow - Javatpoint

$$y = \vartheta\left(\sum_{i=1}^n x_i w_i + b\right) = \vartheta(\mathbf{x} \cdot \mathbf{w} + b) \quad (4.1)$$

Computation of a single perceptron

This simple perceptron is the basic framework. This building block can only perform linear regression if the weights and bias are set correctly (which we will look at in the "Optimization" section). It is not capable of solving more complex problems. To solve more complex tasks we need to have a look at the activation function of the perceptron.

4.1.2 Multilayer Perceptron

Multilayer perceptrons are a combination of simple perceptrons. Depending on the task, they can use different activation functions and different variants of the perceptron. The most common type of multilayer perceptron is the feedforward neural network.³

4.2 Activation Functions

An activation function is a mathematical function used in a neural network to activate the neurons and introduce nonlinearity by transforming the inputs. They are also called transfer functions because they can transform (normalize, randomize, ...) the individual perceptron outputs. The reason why we need an activation function is that a neural network consisting of several simple perceptrons could only compute a linear regression without such a function.⁴

Let's see why this happens when we don't use an activation function. Let's look at the following example. We will combine 2 simple perceptrons into a multilayer feedforward neural network.

First layer takes input(x_1) and multiplies it with weight(w_1) and adds bias(b_1):

$$z_1 = w_1 \cdot x_1 + b_1$$

There is no activation function, meaning the output is the same as the input:

$$\begin{aligned} \vartheta(z_1) &= z_1 \\ x_2 &= \vartheta(z_1) \end{aligned}$$

The output of the first layer is the input of the second layer:

$$z_2 = w_2 \cdot x_2 + b_2 = w_2 \cdot \vartheta(z_1) + b_2 = w_2 z_1 + b_2$$

We can rewrite the equation above to a new linear equation:

$$\begin{aligned} z_2 &= w_2 \cdot (w_1 \cdot x_1 + b_1) + b_2 = w_2 * w_1 \cdot x_1 + w_2 b_1 + b_2 \\ \text{output} &= z_2 = (w_2 * w_1) \cdot x_1 + (w_2 \cdot b_1 + b_2) \end{aligned}$$

We see that a non-existent or linear activation function results in the neural network being able to compute only linear functions, regardless of how many layers your neural network has.⁵ To allow modeling of more complex functions, we need to use a nonlinear activation function.⁶

4.2.1 Binary Step Function

Let's take a look at the simplest nonlinear activation function: the binary step function. The binary step function is a very simple threshold based activation function. It takes a value as input and returns either 0 or 1. The function is defined as follows: This activation function is sometimes used in the output

³: "Multilayer Perceptron".

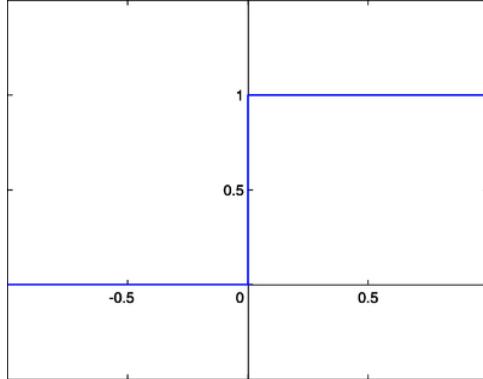
⁴ chowdhury, *Demystifying Activation Functions in Neural Network*.

⁵ DeepLearningAI, *Why Non-linear Activation Functions (C1W3L07)*.

⁶ *Activation Functions in Neural Networks*.

$$\vartheta(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases} \quad (4.2)$$

Binary step function formula

Figure 4.2: Binary step function in a graph⁷

node of a binary classifier. It is not used in hidden layers of a neural network. The reason is that the binary step function is not differentiable. The idea behind this is that we need to be able to calculate how weight changes affect the output of the neural network. If the activation function is not differentiable, we cannot calculate the derivative of the function, and thus we cannot calculate the gradient of the neural network. (We will look at the gradient in the "Optimization" section.)⁸

So let's take a look at other differentiable activation functions.

4.2.2 Sigmoid Function

The sigmoid function is a very common activation function. It is defined as follows:

The sigmoid

$$\vartheta(x) = \frac{1}{1 + e^{-x}} \quad (4.3)$$

Sigmoid function formula

function takes the input value and assigns it a value between zero and one. Although the sigmoid function is suitable for many regression and classification tasks, it has some disadvantages. The main disadvantage is that the sigmoid function goes into saturation and kills the gradients. For very small input values, the gradient of the sigmoid function is very close to 0, and for a very large value, it is close to 1. This means that the gradient is very small, and therefore the neural network weights are not updated much (see the "Optimization" section). This can lead to a situation where the neural network cannot learn anything. This is called the vanishing gradient problem.¹⁰

Another activation function with a similar problem is the Hyperbolic Tangent activation function. Both functions suppress information and have increasingly smaller gradients for small or large values. One way to solve this problem is to use a different activation function that does not squash information as much as the sigmoid function.¹¹

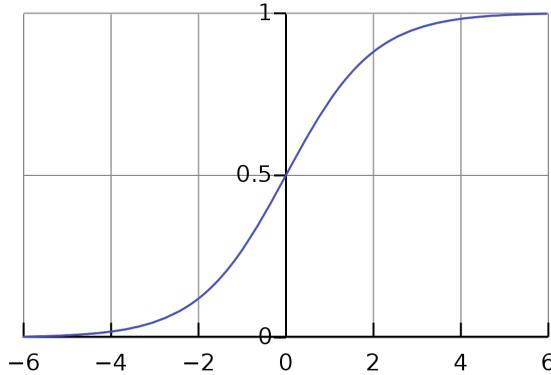
⁷chowdhury, *Demystifying Activation Functions in Neural Network*

⁸SHARMA, *Activation Functions in Neural Networks*.

⁹"Sigmoid Function"

¹⁰Activation Functions in Neural Networks /12 Types & Use Cases/.

¹¹Wang, *The Vanishing Gradient Problem*.

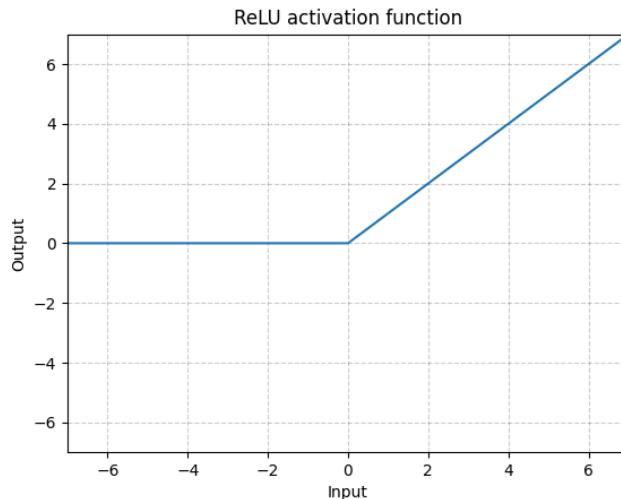
Figure 4.3: Sigmoid activation function on a graph⁹

4.2.3 Rectified Linear Unit (ReLU)

The rectified linear unit is a very popular activation function. It is defined as follows:

$$\vartheta(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (4.4)$$

ReLU function formula

Figure 4.4: ReLU activation function on a graph¹²

It takes the input and returns the input if it is greater than 0, otherwise it returns 0. This means that the ReLU activation function does not suppress the information as much as the sigmoid function. Even though the ReLU activation function is not continuous at 0, it is differentiable. This means that we can calculate the gradient of the ReLU activation function for positive inputs or return 0 for negative inputs. This solves the vanishing gradient problem, but introduces a new problem: the dying ReLU problem.¹³

¹²ReLU — PyTorch 1.12 Documentation

¹³Activation Functions in Neural Networks [12 Types & Use Cases].

The ReLU activation function allows unaffected neurons to not be updated. Only neurons that are activated are updated (see the "Optimization" section). This means that a neuron that is not activated will never be updated. This can lead to a situation where the neural network cannot learn anything. This is called the dying ReLU problem. The absolute loss of information is not as high as in the sigmoid function, but it is still a problem.¹⁴

4.2.4 Leaky ReLU

The leaky ReLU activation function is a variant of the ReLU activation function. It is defined as follows:

$$\vartheta(x) = \begin{cases} ax & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases} \quad (4.5)$$

Leaky ReLU function formula with $a < 1$

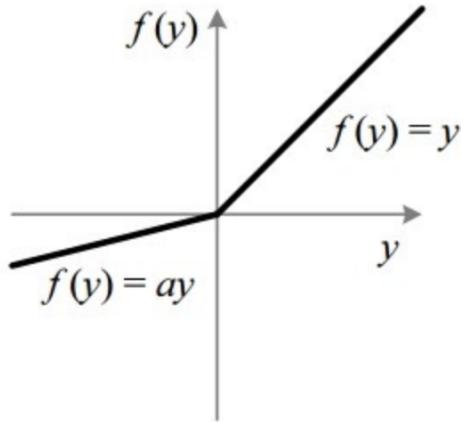


Figure 4.5: Leaky ReLU on a graph¹⁵

The leaky ReLU activation function differs from the ReLU activation function in that it does not return 0 for negative inputs, but a small value. This means that the leaky ReLU activation function does not suppress the information as much as the sigmoid function and also does not have the problem of the dying ReLU.¹⁶

However, there is a problem with both the ReLU and leaky ReLU activation functions: the exploding gradient problem. The exploding gradient problem occurs when the gradient of the activation function is very large. This means that the weights of the neural network are updated very strongly. This can result in the neural network being unable to learn anything. This is called the exploding gradient problem. The absolute loss of information is not as high as with the sigmoid function, but it is still a problem. One could trim the gradient to a certain value, but this too would suppress the information and lead to a problem similar to the dying ReLU problem.

4.2.5 Optimal Activation Functions

Currently, there is no perfect activation function. Each activation function has its advantages and disadvantages. The best activation function depends on the task and the data. In practice, you try different activation functions and see which one works best or give the desired outputs.

¹⁴Leung, *The Dying ReLU Problem, Clearly Explained*.

¹⁵Papers with Code - Leaky ReLU Explained

¹⁶Leung, *The Dying ReLU Problem, Clearly Explained*.

¹⁷SHARMA, *Activation Functions in Neural Networks*

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
Tanh		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU) [2]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [3]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

Figure 4.6: Overview of common activation functions and their derivatives¹⁷

4.3 Loss Function

Neural networks or perceptrons themselves are not capable of learning or knowing how right or wrong they are in the training phase. They simply compute values from their inputs and return an output. It is the job of the loss function to determine how right or wrong the network is. The loss function is a function that takes the predicted value and the actual value as input and returns a value indicating how wrong the prediction was. The loss function is used to train the network. The network is trained by minimizing the loss function. The loss function is also called the "cost function" or "objective function".

4.3.1 Mean Squared Error (MSE)

There are many different loss functions. The most common loss function is the mean squared error (MSE). The MSE is a measure of the average of the squares of the errors. The MSE is calculated by taking the difference between the predicted value and the actual value squared. The MSE is then calculated from the average of all squared errors. The MSE is a good loss function for many regression problems.¹⁸ The MSE is calculated as follows:

While the MSE is well suited for many regression problems, it has some problems in practice. The MSE is not very robust to outliers. If you have some non-distributed outliers in your data, the MSE will be very high because you are taking the sum of squares of the errors. This may mean that the network does not generalize well. For this reason, there are several loss functions. Examples include: mean absolute

¹⁸Frost, *Mean Squared Error (MSE)*.

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (4.6)$$

Mean squared error; y_i is the actual value and \hat{y}_i is the predicted value

error (MAE), Huber loss, quantile loss, and many others.¹⁹

4.3.2 Adaptive Loss Functions

A new method that does not require you to manually figure out which loss function is appropriate for your problem is that of adaptive loss functions. An adaptive loss function is a loss function that automatically adapts to the problem.²⁰

$$f(x, \alpha, c) = \frac{|\alpha - 2|}{\alpha} \cdot \left(\left(\frac{x}{c} \right)^2 + 1 \right)^{\alpha/2} - 1 \quad (4.7)$$

Adaptive loss function formula²¹

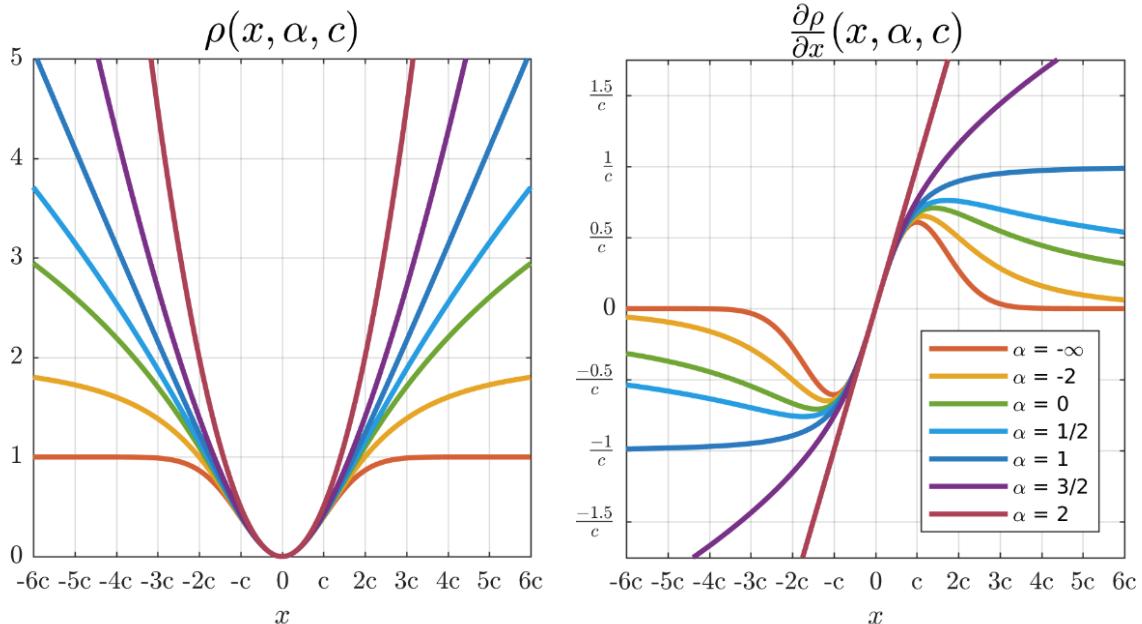


Figure 4.7: General (adaptive) loss function (left) and its gradient (right) for different values of its shape parameter α . Several values of α reproduce existing loss functions: L2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -\infty$)²²

α controls the robustness of the loss, c controls the scaling of the loss, and x is the difference between the predicted value and the actual value. The adaptive loss function is a generalization of many loss

¹⁹Common Loss Functions in Machine Learning / Built In.

²⁰Barron, A General and Adaptive Robust Loss Function.

²²ibid.

functions, but they are not yet widely used in practice. However, they represent a promising new development.²³

4.4 Optimization

Optimization is possibly the most important key component in machine learning, as it controls the entire learning or training process. The sub-steps of optimization sound complex at first, but they are usually performed fully automatically by computers. Ultimately, the goal of our machine learning model is to minimize our cost function to ensure a correct fit to a given observation. When the model adjusts its weights and biases, it uses the cost function to figure out how much it is off by. But how should the weights and biases be changed to reduce this loss?²⁴

Imagine we are at a point in a 2D world with mountains and valleys. Our task is to find the lowest point in the world. We can spawn anywhere, but we can only see what is in our immediate vicinity. So how do we find our way to the desired (lowest or highest, depending on the problem) point? Let's greedily go ahead and just take steps in the direction with the steepest slope. But this might lead us to a local and small valley (called local minima). This is a better position than our original position, but we want to tweak it a bit in order to find a better position. So one option would be to scout the surrounding area first to get a better feel for the landscape around us. One could also scout distant locations without knowing what lies in between, hoping to discover more suitable positions. Optimization in neural networks addresses a very similar problem, but in a much higher-dimensional space and with higher complexity.

4.4.1 Gradient Descent

The most common optimization algorithm is gradient descent. The idea is that we "trace" the error through the network during a phase called "backpropagation". We start with the output layer and compute the error. Then we compute the error for the previous layer and so on until we reach the input layer. To find out to what extent the weights and biases need to be adjusted, we calculate (using derivatives and the chain rule²⁵) the gradient of the cost function with respect to the weights and biases. The gradient is a vector pointing in the direction of the largest increase in the cost function. The algorithm then adjusts the weights and biases in the opposite direction of the gradient. This process is repeated until the cost function is minimized. Once this is done, we can use the trained model for predictions.²⁶

4.4.2 Other Optimization Algorithms

Depending on the type of objective function, there are different approaches to solving the optimization problem. For some functions (continuous and differentiable), optima can be determined via the derivatives of the function (necessary and sufficient criterion for gradients). However, for a large parameter space, this problem cannot be solved in a mathematically closed way. Instead, gradient descent methods can be used here, in which the optimum is approached step by step. Starting from a starting position, the gradient is alternately calculated with the current parameters, and a change of the parameters in the direction of the gradient is made until no more improvement can be achieved. For most neural networks the current gradient is only estimated stochastically (for a part of the data). Accordingly, fewer predictions and gradient calculations are needed. This speeds up the stepwise optimization.²⁸

4.4.3 Brute Force Optimization

If no gradients can be determined, one can try to search the parameter space systematically. Well-known representatives of this idea are the interval bisection method, the golden section method and the Nelder-

²³Bhattacharyya, *The Most Awesome Loss Function ?*

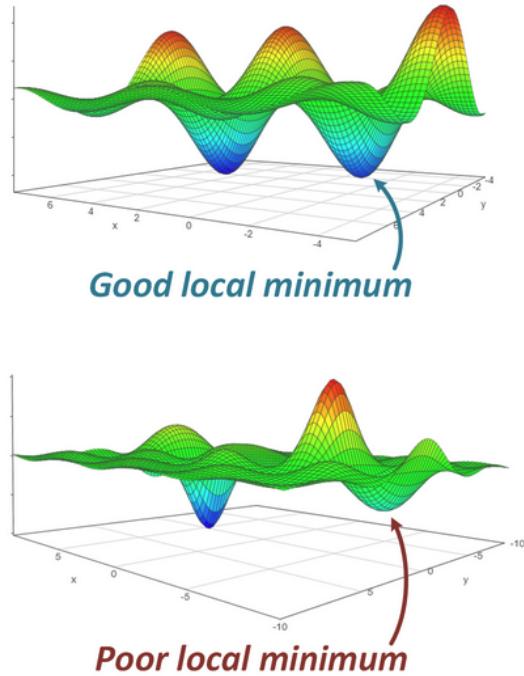
²⁴A. Gupta, *A Comprehensive Guide on Deep Learning Optimizers*.

²⁵which states that the derivative of a function is the product of the derivative of the inner function and the derivative of the outer function: $\frac{\partial}{\partial w} f(g(x)) = \frac{\partial f}{\partial g} \frac{\partial g}{\partial x}$

²⁶12.3. *Gradient Descent — Dive into Deep Learning 1.0.0-Alpha1.Post0 Documentation*.

²⁷Hardt, *Understanding Optimization in Deep Learning by Analyzing Trajectories of Gradient Descent*

²⁸12. *Optimization Algorithms — Dive into Deep Learning 1.0.0-Alpha1.Post0 Documentation*.

Figure 4.8: Good vs poor local minimum²⁷

Mead method. They iteratively partition the parameter space and try to find lower and lower values for the objective function.³¹

Gradient descent and search methods, on the other hand, usually find only local minima, depending on the selected starting point. With multithreading methods, several optimizations can be performed in parallel. Thus, several local optima can be found, among which a global optimum can also be found. Evolutionary algorithms, which are based on the theory of evolution, follow a similar idea. These methods gradually change (generations) a group (population) of parameter assignments (individuals). Through certain mutation and recombination procedures, new assignments are generated, a part of which (depending on their usefulness) "survives" in the next generation.³²

4.4.4 Learning Rate

As explained in this section, optimization is about figuring out how to change the weights and biases of our neural network to reduce the loss function during the backpropagation phase. One of the most important parameters used to determine how much to change the weights and biases is the learning rate. If the learning rate is too high, the weights and biases may change too much and the gradient descent may not converge. If the learning rate is too low, it may take too long for the neural network to converge. While the learning rate is usually set manually, there are also methods to slowly change the learning rate during training. This is called an adaptive learning rate.³³

³¹"Brute-Force Search".

³²*Evolutionary Optimization Algorithms / Wiley*.

³³*Reducing Loss*.

³⁴*Setting the Learning Rate of Your Neural Network*.

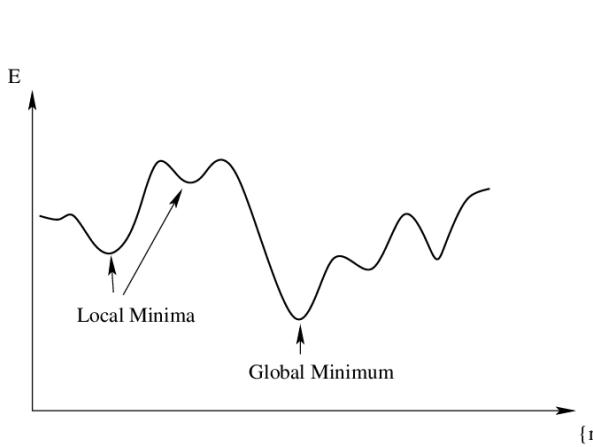
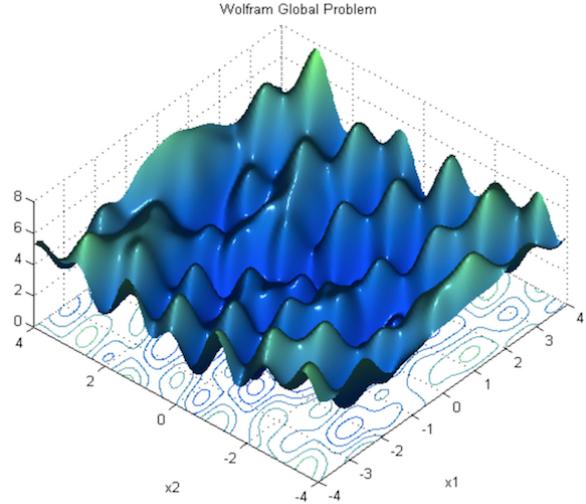
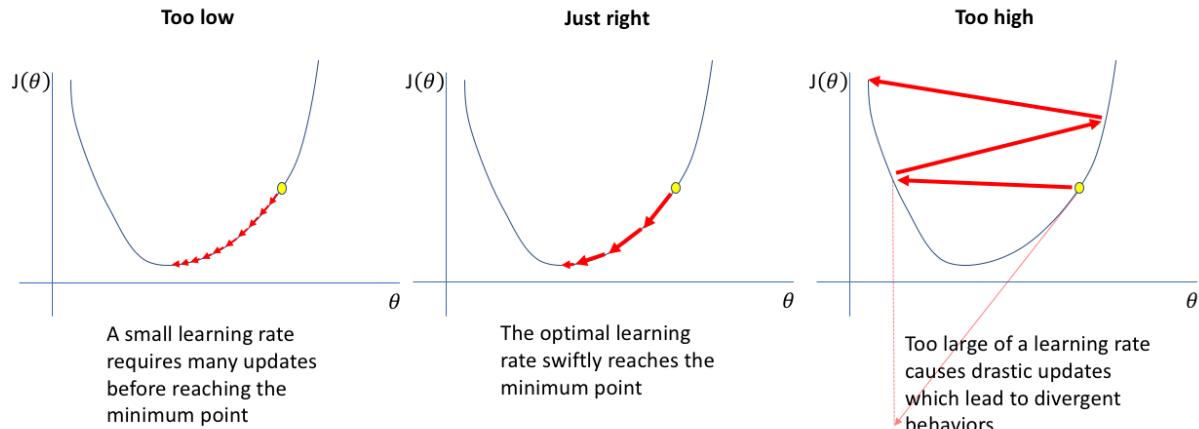
Figure 4.9: Global vs local minimum²⁹

Fig. 1. A Typical Energy Landscape Depicting Position of Several Local...

Figure 4.10: Example function with two parameters.³⁰

Stewart, *Neural Network Optimization*

Figure 4.11: Optimization algorithms try to find the global minimum efficiently in a much higher-dimensional and complex space.

Figure 4.12: How different learning rates affects gradient descent³⁴

4.5 Essence of Deep Neural Networks

Let us briefly summarize what has been described in the above pages. Neural networks are a class of machine learning models inspired by the human brain. They are composed of neurons that are connected to each other. Each neuron has a weight and a bias. Optimization algorithms such as gradient descent, along with parameters such as learning rate, figure out how to optimize the weights and biases to reduce the loss function and thus improve the model's predictions.

Neural networks are universal function approximators. This means that they can approximate any function by approximating a polynomial Taylor series. This is why neural networks are so powerful.

Chapter 5

Reinforcement Learning

In this section, we will take an in-depth look at reinforcement learning (RL). This area is very important because the techniques and methods of RL are used in many areas such as robotics, video games, autonomous vehicles and also in this project to solve the problem statement.

5.1 Introduction

RL describes numerous individual methods in which an algorithm or machine learning agent learns strategies independently. The goal is to maximize reward in a simulation environment. Within this simulation environment, the computer performs an action and then receives feedback. The machine-learning agent receives no information in advance about which action is most promising and must determine its own course of action in a trial-and-error process.

Instead, the computer receives rewards at various points in time that influence its strategies. Through these events, the machine learning agent learns to estimate the sequence of certain actions within the simulation environment. The whole thus forms the basis for the machine-learning agent to develop long-term strategies and maximize rewards in the same process. Several components come into play in decision making. How do we gather experience? How do we make decisions? How do we learn from our decisions? How do we know if we are making the right decisions? These are all questions that RL seeks to answer. To develop algorithms for RL, we need to define the problem systematically.

5.2 Formalization of the Problem

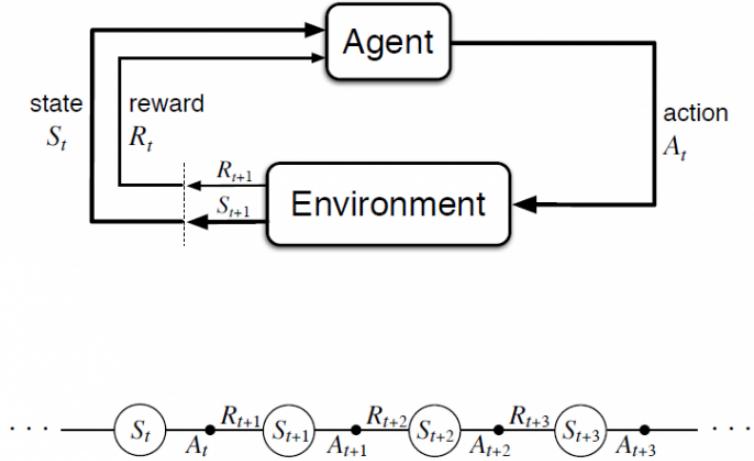
Conventionally, an RL problem is modeled as a Markov decision process. In mathematics, a Markov decision process (MDP) is a discrete-time stochastic control process. This means that the process is governed by a stochastic process (a random process for the agent) and that updates must be made at each point in time. MDP provides a mathematical framework for modeling decision making in situations where outcomes are partially random and partially controlled by a decision maker such as in the real world and many RL environments.

An MDP must satisfy the Markov property, which means that the future state of the system depends only on the current state and not on the sequence of events that led to the current state. A state S_t is *Markov* if $\mathbb{P}(S_{t+1}|S_t, S_{t-1}, \dots, S_0) = \mathbb{P}(S_{t+1}|S_t)$.¹ In simpler terms, the future state of the system depends only on the current state and not on the sequence of events that led to the current state and is thus memoryless.

Once this Markov property is satisfied we can model a MDP using four variables. A state space S , an action space A , a transition function P and a reward function R . The state and action space are a set of

¹Jagtap, *Understanding Markov Decision Process (MDP)*.

²Choudhary, *Dynamic Programming In Reinforcement Learning*

Figure 5.1: Markov decision process illustrated²

all possible states or actions of the system. The transition function (also called probability function) P describes the probability of transitioning from one state to another given an action. The reward function R describes the reward received by the agent for taking an action in a given state.³

Figure 5.1 shows how the MDP is modeled. The agent receives a state from the environment that depends solely on the previous state (Markov property) and on the action that the agent performed and with which the environment transitions to the next state (can be stochastic). The agent receives a reward for its actions. In addition to the variables described above for RL problems, we also define other variables and functions that are important for the RL problem.

In most RL problems, we want to maximize the expected cumulative reward G_t . To tune how much weight we place on future rewards, we add a discount factor. If the discount factor is 0, we only care about the immediate reward which will lead to a greedy strategy. If the discount factor is 1, we care about all future rewards which will lead to a long-term strategy. The discount factor is usually set to 0.9 or 0.99.⁴ In this equation T is the current time step, k is the number of steps in the future and γ is

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (5.1)$$

Expected cumulative reward

the discount factor (0 being greedy and 1 being long-term).

The strategy of our agent is called a policy and denoted by π . A strategy can be deterministic or stochastic and can be represented as a mapping from a state space s to an action space a .⁵

So now let's formalize the problem that RL is trying to tackle. We have an agent that is in a state s and wants to maximize the expected cumulative reward G_t . The agent can take an action a and transition to a new state s' . The agent receives a reward R for taking action a in state s . The agent's strategy is a policy π that maps states to actions. The agent's goal is to find the optimal policy π^* that maximizes the expected cumulative reward G_t .

³Silver, “Lecture 2: Markov Decision Processes”.

⁴blackburn, *Introduction to Reinforcement Learning*.

⁵“Markov Decision Process”.

⁶Silver, “Lecture 2: Markov Decision Processes”

$$\pi^* = \arg \max_{\pi} \mathbb{E}[G_t | \pi] \quad (5.2)$$

Optimal policy which the agent is trying to find⁶

5.3 Bellman Equation

The Bellman equation is a dynamic programming equation (algorithms that use recursive functional approaches to solve problems) used for discrete-time optimization problems such as the MDP optimization problem.⁷ The Bellman equation attempts to answer the following questions:

- If an agent is in a state and it is assumed that the agent performs the best possible action in each substep, what long-term reward can I expect?
- What is the value of a particular state?

$$V(s) = \max_a \left(R(s, a) + \gamma \sum_{s'} P(s', r | s, a) V(s') \right) \quad (5.3)$$

Bellman equation for stochastic environments⁸

In this equation $R(s, a)$ is the immediate reward for taking action a in state s , $P(s', r | s, a)$ is the probability of transitioning from state s to state s' given action a and reward r , $V(s)$ is the value of state s and γ is the discount factor.

The Bellman equation presented here applies to stochastic environments. It states that the value of any state is the immediate reward of that state together with the discounted expected reward of all the next states, maximized over all possible actions. Reason being that we maximize over a and then sum over s' .⁹

5.4 Overview of RL Algorithms

We're now at a point where using the concepts from MDP and Bellman we can start to look at different RL approaches which are then used in Deep RL. The first distinction we can make is between model-based and model-free RL. Model-based RL assumes that the environment is known and the agent can use this knowledge to plan ahead. Model-free RL assumes that the environment is unknown and the agent has to learn the environment by interacting with it.

5.4.1 Model-based RL

Model-based RL attempts to circumvent the problem of lack of prior knowledge about the environment by allowing the agent to construct a fictitious representation of its environment. This means that we have a model of the transition function (or likelihood function) that describes how the environment transitions from one state to another ($P(s', s, a) = Pr(s_{t+1} | s_t = s, a_t = a)$) and a model of the reward function that describes how the agent is rewarded for its actions ($R(s, a) = Pr(r_{t+1} | s_t = s, a_t = a)$). Using this model, the agent can then plan ahead and find the optimal strategy. With this strategy, the agent can then interact with the environment and update the model. This process is repeated until the agent finds the optimal strategy.

⁷“Bellman Equation”.

⁸Tanwar, *Bellman Equation and Dynamic Programming*

⁹Skowster the Geek, *Bellman Equation Basics for Reinforcement Learning*.

Given here a model of the environment as an MDP (since we assume that the environment is an MDP), Bellman equations can be used to solve the problem. There are two main approaches to this: Value Iteration and Policy Iteration.

5.4.1.1 Value iteration

Value iteration is an algorithm that iteratively improves an estimate of the value function. The value function is a function that gives the expected value of a given state. Using the rewritten Bellman equation, we can write the value function as a recursive function for a known reward: The value function

$$V(s) = \max_a \mathbb{E} \left(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a \right) \quad (5.4)$$

Value function for a known reward

is the discounted reward of this and all future rewards (for the best action). The value function can be rewritten as: We can now replace the last sum with the value function for the next state $V(s')$ and

$$V(s) = \max_a \mathbb{E} \left(r_0 + \sum_{t=1}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a \right) \quad (5.5)$$

Value function rewritten to include the reward of the next state

obtain a recursive function: This is called the Bellman optimality condition. It states that the value

$$V(s) = \max_a \mathbb{E} (r_0 + \gamma V(s') | s_0 = s, a_0 = a) \quad (5.6)$$

Bellman optimality condition: Value function rewritten as a recursive function

function is the expected reward of the current state plus the discounted value of the next state. This is the same as the Bellman equation, but now we have defined the value function as a function of strategy π .

This is very useful because now one can use dynamic programming (devide and concur recursive algorithms) to solve the problem. One can start with the value function of the last state and work backwards to the first state. This is called backward induction. Or one can start with the value function of the first state and work forward to the last state. This is called forward induction. This approach is better than a pure brute force approach because the value function only needs to be calculated once for each state.

A simpler approach is to use a random value function, which is then iteratively improved. We create a value table that contains randomly initialized "bad estimates" for the value function of each possible state. With such a value table, we can technically evaluate the value function of each state, even if it is wrong. Now, if we let our agent make the best move (using our current strategy derived from the bad value function), we can update the value function of the state we are in. We can then repeat this process until the value function converges (which is why this method is called value "iteration"). This algorithm is an "on-policy" algorithm (algorithms that evaluate and improve the same strategy used to select actions). This method is only suitable for environments with a small number of states. For larger environments, we can use Deep RL methods (which we will discuss later).¹⁰

¹⁰Steve Brunton, *Model Based Reinforcement Learning*.

Once we have found a sufficiently good, i.e., reasonably good, approximation to the value function, as well as an incorrigible change in the old and new value functions, we can use the value function to find the optimal strategy: $\pi = \arg \max_{\pi} \mathbb{E}(r_0 + \gamma V(s'))$.

5.4.1.2 Policy iteration

Policy iteration is a very similar approach to value iteration. It is a two-step process (unlike the 1 step process of value iteration). We first create a random policy and then we use the policy to find the value function the way we did in value iteration. Once we have found a sufficiently good value function, we can then use the value function to find the optimal policy. So effectively we're "locking" the policy, finding out the value function, then "locking" the value function and finding out the policy. This method typically converges faster than value iteration. However, it is more computationally expensive than value iteration. This method is also "on-policy".¹¹

5.4.2 Model-free RL

Model-free RL is a more practical approach to RL. It assumes that the environment is unknown and the agent must learn the environment by interacting with it. This means that we do not have a model for the transition function or the reward function. This means that we cannot plan ahead and must learn the environment by interacting with it. While this is more practical, it is also more computationally intensive because we have to learn the environment by interacting with it. There are two main approaches to this: On-Policy and Off-Policy.

5.4.2.1 Q-Function

Before we jump into the various different model-free RL algorithms, we need to talk about the Q-function. The Q-function (it can be thought of as a quality function) is a function that gives the joint quality or value of performing an action in a given state, given the probabilities. Using the Q-function the Bellman

$$Q(s, a) = \sum_{s'} P(s'|s, a) (R(s, a, s') + \gamma V(s')) \quad (5.7)$$

Quality function¹²

equation can be rewritten as $V(s) = \max_a Q(s, a)$. The policy function of our agent (stated above) $\pi(a|s) = \arg \max_a Q(s, a)$ is the action that maximizes the Q-function for a given state s . The reason we're creating this function is because it encompasses both the immediate reward and the discounted value of the state. This allows us to not need a model of the environment. This is why this method is called a model-free approach. It's a very useful approach because it allows us to use RL in environments where we don't have a model of the environment. Through a given Q-function, both the value function ($V(s) = \max_a Q(s, a)$) and the policy function ($\pi(a|s) = \arg \max_a Q(s, a)$) can be derived.¹³

5.4.2.2 Monte Carlo Learning

The first algorithm we will look at is the Monte Carlo learning algorithm, which executes random moves and learns to approximate the value or Q function by playing episodes. In Monte Carlo learning, we try to approximate the value function $V(s)$ or the Q function $Q(s, a)$ by running a large number of episodes and averaging the rewards.¹⁴ The reward function R_{Σ} is a discounted reward function of the complete episode. The value function is now defined as a recursive function which allows us to use environment episode (completed game play) observations to update our value function. The Q function can be updated in a similar way. In these two cases, we can approximate our function using a neural network. The loss

¹¹Steve Brunton, *Model Based Reinforcement Learning*.

¹²Steve Brunton, *Q-Learning*

¹³Shyaliaka, *A Beginners Guide to Q-Learning*.

¹⁴Team, *Reinforcement Learning*.

$$R_{\sum} = \sum_{t=0}^n \gamma^t r_t \quad (5.8)$$

Monte Carlo reward function

$$V_{\text{new}}(S_t) = V_{\text{old}}(S_t) + \frac{1}{n}(R_{\sum} - V_{\text{old}}(S_t)) \quad (5.9)$$

Monte Carlo value function approximation

$$Q_{\text{new}}(S_t, A_t) = Q_{\text{old}}(S_t, A_t) + \frac{1}{n}(R_{\sum} - Q_{\text{old}}(S_t, A_t)) \quad (5.10)$$

Monte Carlo Q function approximation

function would be the last part of the above equation (for value function: $\frac{1}{n}(R_{\sum} - V_{\text{old}}(S_t))$; for Q function: $\frac{1}{n}(R_{\sum} - Q_{\text{old}}(S_t, A_t))$). During the neural network training, the old function should converge against the new function ($V_{\text{new}}(S_t)$ or $Q_{\text{new}}(S_t, A_t)$). In principle this algorithm should converge given enough episodes.

5.4.2.3 Temporal Difference Learning

Temporal difference (TD) learning is an algorithm that also uses episodes to learn the value function. However, it does not use the entire episode to update the value function. Instead, a single step of the environment is used to update the value function. This is referred to as the TD(0) algorithm. The TD algorithm is a model-free algorithm, which means that it does not require a model of the environment. The TD algorithm can be written as follows:

$$V(s) = \mathbb{E}[R(s, a, s') + \gamma V(s')] \quad (5.11)$$

TD value function

To approximate the value function, we can write it in a recursive form together with a learning rate α : Thus, the loss function can be viewed as the TD target estimator ($R_{\sum} = r_k + \gamma V_{\text{old}}(S_{k+1})$) minus

$$V_{\text{new}}(S_t) = V_{\text{old}}(S_t) + \alpha(r_t + \gamma V_{\text{old}}(S_{t+1}) - V_{\text{old}}(S_t)) \quad (5.12)$$

TD value function approximation

the old value function ($V_{\text{old}}(S_t)$) with a learning rate α . You will notice that this is very similar to the Monte Carlo algorithm. The difference is that the TD algorithm uses a single step in the environment to update the value function, while the Monte Carlo algorithm uses the entire episode to update the value function.

TD(0) looks one step ahead, while Monte Carlo looks at the entire episode. But we don't have to stop here. We can look more than one step into the future. TD(n) looks n steps ahead. The TD(n) algorithm can be written as follows: What does n mean? n represents the number of steps we look forward in the environment. As n approaches infinity, the TD(n) algorithm approaches the Monte Carlo algorithm.

$$V_{\text{new}}(S_t) = V_{\text{old}}(S_t) + \alpha(r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V_{\text{old}}(S_{t+n}) - V_{\text{old}}(S_t)) \quad (5.13)$$

TD(n) value function approximation

TD(n) algorithms are on-policy algorithms, meaning they must choose the best perceived action to converge. The TD algorithm must act greedily. We can modify the TD algorithm to be off-policy, which means we can choose suboptimal actions. This is called a Q-learning algorithm.¹⁵

5.4.2.4 Q-Learning

Q-learning is an algorithm that is almost identical to the TD(0) algorithm. The only difference is that the TD(0) algorithm uses the value function to update the value function, while the Q-learning algorithm uses the Q-function to update the Q-function. The Q learning algorithm can be written as follows: This equation, which is similar to the TD equation above, states that the new Q-function updates itself

$$Q_{\text{new}}(S_t, A_t) = Q_{\text{old}}(S_t, A_t) + \alpha(r_t + \gamma \max_{a'} Q_{\text{old}}(S_{t+1}, a') - Q_{\text{old}}(S_t, A_t)) \quad (5.14)$$

Q learning algorithm

by comparing its estimated Q-value with the actual Q-value obtained (reward and Q-value of the next state). Similar to model-based algorithms, we can approximate the Q-value by randomly initializing the Q-function (by creating a table of states and Q-values) and then iteratively updating our Q-values by interacting with the environment.¹⁶

It is important to note that the Q learning algorithm also allows for suboptimal actions. This is because the Q-learning algorithm uses the maximum Q-value of the next state to update the Q-value of the current state. The new Q-value includes the reward and the maximum Q-value of the next state, regardless of which action was chosen, thus allowing exploration and transfer learning.¹⁷

5.4.2.5 Off-Policy vs. On-Policy

We have seen that there are two main types of RL algorithms: off-policy and on-policy. Which one should we use? The answer is that it depends on the problem.

Off-policy algorithms allow for suboptimal actions. On-policy algorithms do not allow suboptimal actions. Off-policy algorithms are more flexible, while on-policy algorithms are more stable and online. In general, off-policy algorithms are better suited for problems with a large state space and learn faster, while on-policy algorithms are better suited for problems with a small state space and are more stable because they do not allow suboptimal actions.¹⁸

5.4.2.6 Summary

In this section, we have seen that RL algorithms can be categorized into two main types: model-based and model-free. Model-based algorithms use a model of the environment to learn the value function independently or together with the strategy. Model-free algorithms do not use a model, such as the time difference of the environment, to learn the value function. Algorithms can be divided into two main types: Off-Policy and On-Policy. Off-policy algorithms allow suboptimal actions, while on-policy algorithms do not allow suboptimal actions. These algorithms are theriotic algorithms that work in simple environments such as tic-tac-toe. However, these algorithms are unsuitable for complex environments with large state spaces. In the next section, we will see how Deep Learning can be used to solve these problems.

¹⁵Steve Brunton, *Q-Learning*.

¹⁶Shyaliaka, *A Beginners Guide to Q-Learning*.

¹⁷Steve Brunton, *Q-Learning*.

¹⁸Causevic, *A Structural Overview of Reinforcement Learning Algorithms*.

5.5 Deep Reinforcement Learning

Deep reinforcement learning (DRL) is a subfield of reinforcement learning that uses deep neural networks to approximate functions, as they are capable of learning complex functions as universal function approximators. The use of neural networks in classical RL algorithms has several massive advantages.

First, unlike tabular value pairs, neural networks can represent many more states. Think of an 8x8 checkerboard as our state space. Such an environment would have (conservatively estimated) 10^{120} states.¹⁹ (That's more states than there are atoms in the universe.²⁰) That's a huge state space, and it's impossible to represent all those states in tabular form. However, neural networks can represent this state space with ease. This is because the entire state space can be represented with 64 input nodes in a single layer. This greatly facilitates the solution of complex RL problems.

Another major advantage of neural networks is automatic feature extraction. Neural networks extract helpful, low-dimensional patterns by finding intrinsic relationships in the data. In chess, for example, this would mean identifying motives or tactics. This and the ability to develop complex strategies through the breadth and depth of the network make neural networks very powerful and a great tool for solving complex RL problems.

5.5.1 Deep Policy Networks

The simplest and most straightforward way to use Deep Learning in RL is to approximate the best strategy π using a neural network. The difficulty is in determining the loss function. If one takes only the negative reward (or something similar) as the loss function, the network will not learn in the long run and will only learn greedy actions. This will not lead to good performance and is not practical in many environments because of the sparse rewards. What you need to do is to include the future rewards in our approximation equation: This equation states that our neural networks with parameters (weights

$$\text{Policy Network: } \pi_\theta(s, a) \Rightarrow \theta_{\text{new}} = \theta_{\text{old}} + \alpha \delta_\theta R_{\Sigma, \theta} \quad (5.15)$$

Deep Policy Network algorithm

and biases) θ should be updated by the gradient of expected return $R_{\Sigma, \theta}$ with respect to parameters θ . This is a very powerful equation, but it is not easy to implement. The problem is that the expected return $R_{\Sigma, \theta}$ is not known. So this is where different algorithms come into play. For example, if one represents this expected future return by a value function, one is dealing with an Actor-Critic algorithm. Methods such as the Actor-Critic algorithm (AC; and A2C, A3C), the Trust Region Policy Optimization algorithm (TRPO), and the Proximal Policy Optimization algorithm (PPO) are all similar algorithms. These algorithms are called policy gradient algorithms because they use the gradient of expected return to update the policy network.²¹

5.5.2 Deep Q-Networks

Deep Q-networks (DQN) are a type of DPN that use a neural network to approximate the Q-function. These algorithms (and their variations) have been used to solve many complex RL problems, such as Atari games²², robotics, and finance. The DQN algorithm can be written as follows: You will notice that this is almost the same equation as the Q learning algorithm. The basic loss function (which you can plug into other functions like MSE) is the same as the Q learning algorithm: $l = (r_t + \gamma \max_{a'} Q_{\text{old}}(S_{t+1}, a') - Q_{\text{old}}(S_t, A_t))$. Typically the neural network in DQNs uses MSE $L = l^2 = (r_t + \gamma \max_{a'} Q_{\text{old}}(S_{t+1}, a') - Q_{\text{old}}(S_t, A_t))^2$ as the loss function. The difference is that the Q-learning algorithm uses a Q-table (with state-action and value pairs) to update the Q-function, while the DQN

¹⁹“Shannon Number”.

²⁰ Which Is Greater? The Number of Atoms in the Universe or the Number of Chess Moves? / National Museums Liverpool.

²¹ AI, Reinforcement Learning Algorithms — an Intuitive Overview.

²² Mnih et al., Playing Atari with Deep Reinforcement Learning.

$$Q_{\text{new}}(S_t, A_t) = Q_{\text{old}}(S_t, A_t) + \alpha(r_t + \gamma \max_{a'} Q_{\text{old}}(S_{t+1}, a') - Q_{\text{old}}(S_t, A_t)) \quad (5.16)$$

Deep Q-Network algorithm

algorithm uses a neural network to approximate the Q-function. The DQN algorithm is an off-policy algorithm, which means that it can choose suboptimal actions. This is because the DQN algorithm uses the maximum Q-value of the next state to update the Q-value of the current state. The new Q-value contains the reward and the maximum Q-value of the next state, regardless of which action was chosen, enabling exploration and transfer learning.²³

5.5.3 Deep Dueling Q-Networks

Deep Dueling Q-networks (DDQN) is a advantageous type an extension of the DQN algorithm which uses policy gradients. The main difference is that the DDQN algorithm uses two neural networks to approximate the Q-function. One network is used to estimate the state value function $V(s)$, and the other network is used to estimate the advantage function $A(s, a)$. The Q-function is then calculated as follows: ²⁵

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{|A|} \sum_{a'} A(s, a')) \quad (5.17)$$

DDQN Q-function

$$L_{\text{actor}} = \frac{1}{2}(r_t + \gamma V(s_{t+1}) - V(s_t))^2 \quad (5.18)$$

DDQN actor loss function

$$L_{\text{critic}} = \frac{1}{2}(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))^2 \quad (5.19)$$

DDQN critic loss function derived from the policy gradient equation²⁴

The DDQN is now training two separate neural networks, one to estimate the value of being in a state and one to estimate the relative advantage of taking an action. This decoupling technique is very powerful and allows the DDQN to learn the value of being in a state without having to learn the effect of each action at each state. This is a very powerful technique that can be used to solve complex RL problems.²⁶

5.5.4 Actor Critic Network

The actor critic network (ACN) is a type of policy gradient algorithm that uses a neural network to approximate the value function $V(s)$ and the policy function $\pi(s)$ at the same time. Actor-critic methods are TD methods that have a separate memory structure to explicitly represent the policy independent of the value function. The idea is that we do policy iteration and value iteration at the same time.

²³Steve Brunton, *Overview of Deep Reinforcement Learning Methods*.

²⁵Steve Brunton, *Overview of Deep Reinforcement Learning Methods*

²⁶Yoon, *Double Deep Q Networks*.

The "actor" tries to find the best policy, and the "critic" tries to find the best value function.²⁷ The ACN algorithm can be written as follows: The policy function is using the temporal difference error

$$V_{\text{new}}(S_t) = V_{\text{old}}(S_t) + \alpha(r_t + \gamma V_{\text{old}}(S_{t+1}) - V_{\text{old}}(S_t)) \quad (5.20)$$

Critic approximation algorithm

$$\pi_{\text{new}}(S_t) = \pi_{\text{old}}(S_t) + \alpha(r_t + \gamma V_{\text{old}}(S_{t+1}) - V_{\text{old}}(S_t)) \quad (5.21)$$

Actor approximation algorithm

as the loss function, and the value function is using the same loss function as the DQN algorithm. The ACN algorithm is an on-policy algorithm, which means that it can only choose optimal actions. This is because the ACN algorithm uses the value function of the next state to update the value function of the current state. The new value function contains the reward and the value function of the next state, which is only possible if the optimal action was chosen. The ACN algorithm is a very powerful algorithm that can be used to solve complex RL problems.²⁸

The loss of the actor is the negative log probability of the action taken because the actor is trying to maximize the probability of taking the optimal action. The loss of the critic is the mean squared error of the value function. The actor and critic are trained at the same time, and the actor is trained to maximize the critic's loss. This is called the actor-critic method. The actor-critic method is a very powerful algorithm that can be used to solve complex RL problems.²⁹

5.5.5 Advantage Actor Critic Network

The advantage actor critic network (A2CN) is a type of ACN that uses a neural network to approximate the advantage function $A(s, a)$ and the policy function $\pi(s)$ at the same time. It has a similar idea to the DDQN algorithm, where the A2CN algorithm uses two neural networks to approximate the advantage function and the policy function. The A2CN algorithm can be written as follows: The policy function

$$A_{\text{new}}(S_t, A_t) = A_{\text{old}}(S_t, A_t) + \alpha(r_t + \gamma \max_{a'} A_{\text{old}}(S_{t+1}, a') - A_{\text{old}}(S_t, A_t)) \quad (5.22)$$

Advantage approximation algorithm

$$\pi_{\text{new}}(S_t) = \pi_{\text{old}}(S_t) + \alpha(r_t + \gamma \max_{a'} A_{\text{old}}(S_{t+1}, a') - A_{\text{old}}(S_t, A_t)) \quad (5.23)$$

Policy approximation algorithm

is using the temporal difference error as the loss function, and the advantage function is using the same loss function (derived from policy gradient equation) as the DDQN algorithm. The A2CN algorithm is an on-policy algorithm, which means that it can only choose optimal actions. This is because the A2CN algorithm uses the advantage function of the next state to update the advantage function of the current state. The new advantage function contains the reward and the advantage function of the next

²⁷ 6.6 Actor-Critic Methods.

²⁸ Causevic, *A Structural Overview of Reinforcement Learning Algorithms*.

²⁹ Karunakaran, *The Actor-Critic Reinforcement Learning Algorithm*.

state, which is only possible if the optimal action was chosen. The A2CN algorithm is a very powerful algorithm that can be used to solve complex RL problems.³⁰

5.5.6 Summary

In this section we have taken a brief look at some DRL algorithms. There are many more methods and algorithms in this area, however this does not fit into the scope of this paper. What should be taken away from this is that classical RL algorithms can be strengthened by neural networks and that many algorithms can be derived from each other and from equations such as the policy gradient equation or the Bellman equation. This is a very powerful technique that can be used to solve complex RL problems.³¹

³⁰Steve Brunton, *Overview of Deep Reinforcement Learning Methods*.

³¹Causevic, *A Structural Overview of Reinforcement Learning Algorithms*.

Chapter 6

Development of the RL Agent: AI and Results

We are now at a point where we know enough about DRL to begin developing our AI bots. In this chapter, I will describe the practical development of our AI. The goal of this thesis is to develop a bot (computer controlled player) that uses knowledge from artificial intelligence and RL to play our game. I have developed a web-based real-time game (see the "Game" chapter) for which I want to develop a bot.

In the first part, I discuss the technologies used, then I explain what systems and experiments were performed to find good algorithms, and finally the best bot created is described and analyzed hvgvhv

6.1 Technologies

Lets take a look at the different technologies and tools I used. In order to create the AI-controlled bot, I needed to use a relatively efficient language with great ML support and a large community. I chose Python¹ because it is the most widely used language in the field. Below, I'll go over some of the libraries I used.

6.1.1 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing RL algorithms. It supports teaching agents in everything from walking to playing games like pong or pinball. It is a great tool for testing and comparing different RL algorithms. It also provides a large number of environments in which to test the algorithms. I used the OpenAI Gym library to create my own custom environment that met its API requirements. This allowed me to connect to with other libraries that are compatible with the OpenAI Gym library.²

6.1.2 Stable Baselines 3

This library provides a large number of state-of-the-art RL algorithms. It is a useful tool for testing and comparing different RL algorithms. I used it to train and test, tune and compare different RL algorithms. Additionally, Stable Baselines 3 offers a wide range of useful wrappers and functions around OpenAI Gym, such as Framestacking, Normalizing, and Logging.³

¹Van Rossum and Drake, *Python 3 Reference Manual*.

²Brockman et al., *OpenAI Gym*.

³Raffin et al., "Stable-Baselines3: Reliable Reinforcement Learning Implementations".

6.1.3 PyTorch

PyTorch is an open source ML library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook’s AI Research lab. I used PyTorch under the hood of Stable Baselines 3 to create and train the neural networks.⁴

6.1.4 TensorBoard

Tensorboard is a visualization tool for ML experiments. It allows you to track and visualize metrics such as loss and accuracy, visualize the model graph, view histograms of weights, biases, or other tensors as they change over time, project embeddings to a lower dimensional space, and much more. I used this to visualize the training process metrics such as the reward, episode length and loss.⁵

6.1.5 Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. I used Matplotlib to visualize the game as a heatmap.⁶

6.1.6 Celluloid

Celluloid is a Python library for creating animations. I used Celluloid to create the animations of the game from Matplotlib heatmaps.⁷

6.1.7 NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things a powerful N-dimensional array object and useful linear algebra, Fourier transform, and random number capabilities. I used NumPy extensively in the environment to handle the game state and the observation space.⁸

6.2 Local environment

Now that we’ve taken a look at the technologies used, let’s look at how I developed a system that allows the AI bots to train. Since I want to train our AI in an efficient way, I needed to be able to run through the game quickly. Doing this via network calls to the server is very inefficient. In general, network response times vary between 100 and 200 milliseconds⁹, which is too slow for our purposes. Doing 100k network calls would take more than 2.5 hours. Also, our game runs at about 60 frames per second, which means we would be limited to 60 network calls per second, which in turn would cause 100k game iterations to take more than 3.2 hours. 100k game iterations is a very small number in RL.

In order to train the AI efficiently, I decided to create an exact copy of the game (without the entire network) for local training. This way, I can play through the game as fast as my computer can. This is a very common practice in RL, where the environment is simulated locally.

The local environment is modular, so it is easy to extend and replace components. To allow our agent to train against other players (e.g., agents with random rules), I created a central instance where all games are stored and managed. This way I can easily create multiple games and run them in parallel. Whenever an environment is instantiated, it participates in a game of the central control instance. Each player or the corresponding environment instance can only have a limited view of the entire environment.

The environment (which is technically an abstract wrapper around the actual game) can modify the observation and state space as well as the reward function in a limited way. In this way, we can adjust the environments to optimize the learning process.

⁴Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library”.

⁵Abadi et al., “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”.

⁶Hunter, “Matplotlib: A 2D graphics environment”.

⁷Kvam, *Celluloid*.

⁸Harris et al., “Array programming with NumPy”.

⁹8 Ways to Effectively Reduce Server Response Time.

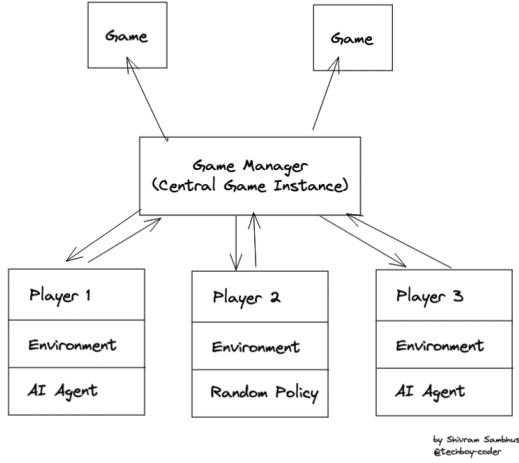


Figure 6.1: Local architecture of the game

6.3 Procedure and Initial Conditions

In order to systematically develop our AI, I have conducted some mini-experiments that show how certain approaches work and how they can be improved. These experiments are described in the following sections. In order to compare different approaches, I had to create uniform initial conditions. These include the environment reward function, the viewing distance (which defines the observation space), the number of randomly moving players (to simulate other bad players), the size of the field, the food in that field, and finally the number of training steps each agent is allowed to use.

Let's start with the reward function for all our environments. This equation states that if the agent dies,

$$\delta_t = (score_t - score_{t-1}) * 10 + \delta_{t-1} * 0.2 - 0.25$$

$$R = \begin{cases} \delta_t & \text{if alive} \\ -20 & \text{if dead} \end{cases} \quad (6.1)$$

Equation for the reward function

he receives a deduction of 20. If the agent lives, he receives a reward equal to the difference between the current score and the previous score multiplied by 10. Reward gained at the previous timestep attenuated by a factor (here 0.2) is then added so as to encourage activities like eating. Lastly we add an asymptotic penalty of -0.25. This ensure that the agent is penalized (in the long run) for not eating long enough. This equation is a good starting point for the reward function, but can be modified as needed.

The visibility d is the number of units in each direction that the agent can see. The observation space would be $(2d + 1) * (2d + 1)$, since the agent can see in all directions. The observation space is the input to the neural network. The observation space is a 2D array of flattened integers. The integers represent the type of object at that position. The integers are as follows: -1 is a wall, 0 is empty, 1 represents food, and 2 represents a snake body unit.

The number of players moving randomly is the number of players not controlled by the agent. These players are controlled by a random strategy. This is done to simulate other bad players and make the game more challenging. For the following experiments, I used 4 randomly moving players.

The field size is the size of the field on which the game is played. The field is a square, so the field size is the length of one side. The field size is 20 units. Food is placed randomly on the playing field. The number of food pieces is also 20, which gives a food density of 5%.

The number of training steps is the number of steps the agent is allowed to take. This is the number of times the agent can choose an action. The number of training steps is 200k, which means that the agent can choose an action 200k times (when the game ends, a new game is started). Then the performance is measured over 200 games (each game lasts until the agent dies). This is done to get a more accurate measure of the agent's performance.

6.4 Experiments

I conducted experiments on the following topics: environment, observation, network architecture, hyper-parameters, and algorithm types. The experiments are described in the following sections.

6.4.1 Baseline experiment

The first experiment I performed was to create a random agent. This agent simply chooses a random action from the set of possible actions. This is a very simple approach, but it is a good basis for comparison with other approaches. The random agent is a good basis because it is very simple to implement and is a good indicator of how simple the environment would be.

Results: An agent with a random strategy received an average reward of -9.7 with a standard deviation of 17.5. This is a reward very close to the reward for dying, which is -20, but a bit more since it is possible to stumble across food randomly. This is a good starting point to compare other approaches.

6.4.2 DQN Baseline experiment

In this and the next experiments, a DQN is used. DQNs are a very common approach and will be used later to compare different parameters. In order to compare different approaches, I once again needed a baseline. The baseline for this experiment is a DQN with a network architecture of 64x64 fully connected neurons and an input dimension of $17 \times 17 = 289$ (derived from the 8 units of visual distance in each direction) with a ReLU activation function, a learning rate of 0.0001, and the Adam Optimizer (a commonly used algorithm based on gradient descent).

Results: The DQN with the above architecture received an average reward of 40.02 after 200k training steps (or iterations) with a standard deviation of 60.41. This is certainly an improvement over the random agent.

6.4.3 DQN with Framestacking

In this experiment, I used the same DQN as in the previous experiment, but I added framestacking. Framestacking is a technique that allows the agent to see the previous n frames. This allows the agent to see the difference between the previous state and next state. This is useful for the agent to learn the dynamics of the game. In this experiment, I used $n = 2$. This in turn increases the observation space to $2 \times 17 \times 17 = 578$.

Results: By doubling the observation space by adding framestacking one could have either expected an increase in performance due to increased information or a decrease in performance due to increased complexity and observation space. In this case, the agent received an average reward of 42.74 with a standard deviation of 65.75. This is a very small increase in performance, but can be considered negligible due to the small sample size and unknown variance.

6.4.4 DQN with observation normalization

In this experiment, I used the same DQN as in the previous experiment (without framestacking), but I added observation normalization. Observation normalization is a technique that normalizes the observation space. This was done by subtracting the average mean and dividing by the standard deviation. In our case, the average mean and standard deviation are calculated using the moving average and moving standard deviation. An important reason for normalizing the values is to avoid problems like exploding, disappearing or vanishing gradients. This is done to make the observation space more stable and to

$$\begin{aligned}\mu &= \frac{1}{N} \sum_{i=1}^N x_i \\ \sigma &= \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2} \\ x_{norm} &= \frac{x - \mu}{\sigma}\end{aligned}\tag{6.2}$$

Equation for observation normalization (N is the number of observations from which the mean and standard deviation are calculated, x is the observation, μ is the mean, σ is the standard deviation, and x_{norm} is the normalized observation)

stabilize the learning process. In this experiment, I used the mean and standard deviation of the entire observation space.

Results: The agent received an average reward of 45.96 with a standard deviation of 73.22. Once again this is only a small increase in performance.

6.4.5 DQN with observation normalization and framestacking

In this experiment, I combined the use of framestacking and observation normalization.

Results: This time the results did improve much more than in the previous experiments. The agent received an average reward of 73.27 with a standard deviation of 94.00. We can see that the combination of framestacking and observation normalization is very effective.

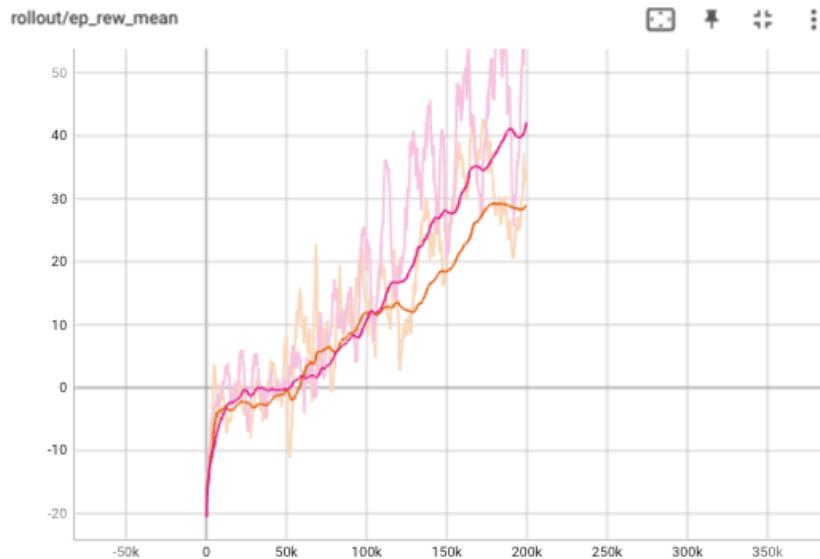


Figure 6.2: Comparison of the baseline DQN with the DQN with framestacking and observation normalization. (Orange: Baseline DQN, Red: DQN with framestacking and observation normalization)

6.4.6 DQN with observation normalization, framestacking and a larger network

In this experiment, I used the same DQN as in the previous experiment, but I increased the size of the network. The network architecture is now 512x512 fully connected neurons. This is a very large network, but it is still a fully connected network. This is done to see if a larger network can improve the performance of the agent by increasing the ratio of input nodes to hidden nodes.

Results: This change in network architecture did improve the performance of the agent by a good amount. The agent received an average reward of 89.21 with a standard deviation of 107.75.

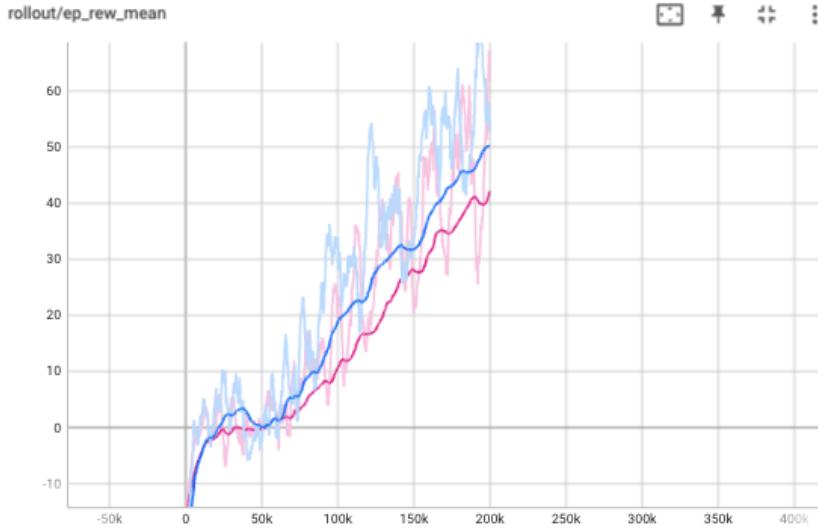


Figure 6.3: Comparison of different network architectures for DQN with framestacking and observation normalization. (Orange: 64x64, Blue: 512x512)

6.4.7 Scheduled Learning Rate

In this experiment, I used the same DQN as in the previous experiment, but I added a scheduled learning rate. A scheduled learning rate is a learning rate that changes over time. This allows the neural network to make bigger gradient descent steps at the beginning of the training process and smaller gradient descent steps at the end of the training process. This is done to make the learning process more stable and to avoid the agent getting stuck in a local minimum. In this experiment, I used a learning rate that starts at 0.001 and approaches 0 at the end of the training process. This is done by using the following equation:

$$\begin{aligned}
 & \text{Progress: } p = 1 \rightarrow 0 \\
 & \text{Initial learning rate: } \alpha_0 = 0.001^{1/3} \\
 & \text{Decay amount: } \delta = 0.00001^{1/3} \\
 & \text{Learning rate: } (\alpha = (\alpha_0 - \delta) \cdot p)^3
 \end{aligned} \tag{6.3}$$

Scheduled learning rate for DQN with framestacking and observation normalization

Results: This change in learning rate did improve the performance of the agent. The agent received an average reward of 98.72 with a standard deviation of 113.22.

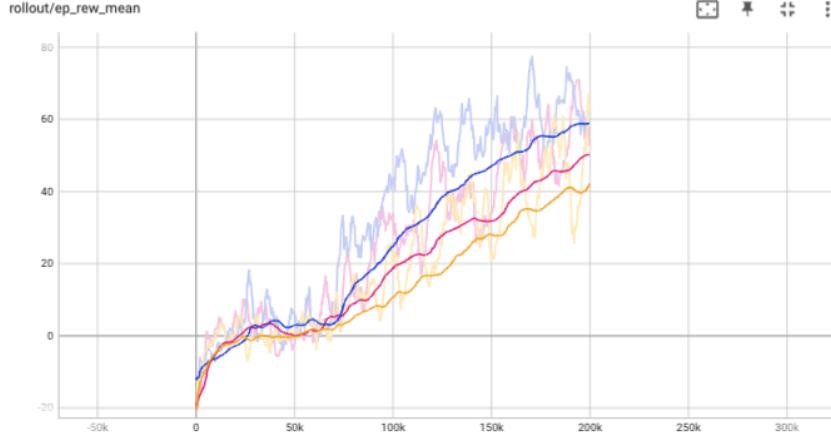


Figure 6.4: Comparison of small DQN (64x64, Yellow), big DQN (512x512, Red) and big DQN with scheduled learning rate (512x512, Blue)

6.4.8 Dynamic Exploration Rate

In this experiment, I used the same DQN as in the previous experiment, but I added a dynamic exploration rate. A dynamic exploration rate is an exploration rate that changes over time. This allows the agent to explore more at the beginning of the training process and less at the end of the training process. This allows the agent to find a good balance between exploration and exploitation. In this experiment, I used an exploration rate that starts at 0.5 and approaches 0.0001 at the end of the training process.

Results: This change in exploration rate did improve the performance of the agent. The agent received

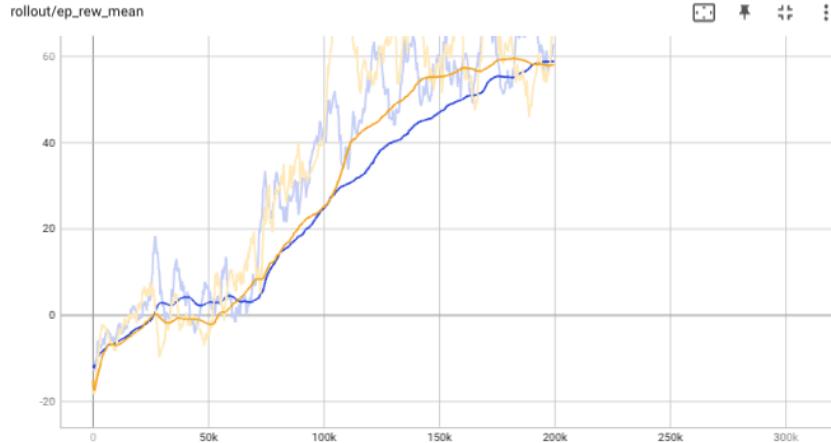


Figure 6.5: Comparison of DQN without dynamic exploration rate and DQN with dynamic exploration rate. (Blue: DQN without dynamic exploration rate, Yellow: DQN with dynamic exploration rate)

an average reward of 105.83 with a standard deviation of 263.87.

6.4.9 Observation Space Reduction by Feature Extraction

In this experiment, I decided to change the way the AI can observe the environment. Currently, the AI observes the environment in a 17x17 grid with its current position in the center. This is a very large observation space. I made the observation space smaller using feature extraction. Feature extraction is a technique that can be used to reduce the dimensionality of the observation space. I switched from

a grid-based observation to a direction and distance-based observation. I manually extracted the type of object and the distance to the object for the 8 directions top, bottom, left, right, top-left, top-right, bottom-left, and bottom-right. This reduced the observation space from 17×17 to 8×2 for any grid-based observation. This is a very large reduction in observation space. The purpose of this is to determine if a smaller observation space can improve agent performance by reducing the amount of information the agent must process and learn from.

Results: This change in observation space improved the agents performance substantially. The agent received an average reward of 182.3 with a standard deviation of 94.84. We can see that feature extraction has really boosted the performance of our AI. We'll be using this observation space for the rest of the experiments since it is the most efficient and effective observation space currently observed. In the following experiments, I'll be using the same environment architecture as in this experiment but test out different DRL algorithms.



Figure 6.6: Comparison of different observation spaces for DQN with framestacking and observation normalization. (Black: $17 \times 17 \times 2$, Blue: $8 \times 2 \times 2$)

6.4.10 Advantagous Actor Critic (A2C)

In this experiment, I used the same environment as in the previous experiment, but I used the Advantagous Actor Critic (A2C) algorithm. A2C is a policy gradient algorithm that uses a neural network to approximate the value function. It is a combination of value and policy iteration based methods (see RL theory chapter), which allows for more stability in the network by decoupling the value and policy networks. I used the A2C algorithm with 512×512 architecture and 8×2 observation space. I used the same hyperparameters as in the previous experiment. The only difference is that I used the A2C algorithm instead of the DQN algorithm.

Results: This change of algorithm decreased the agents performance substancially. The agent received an average reward of 66.33 with a standard deviation of 84.65. We can see that A2C is not a good algorithm for this environment.

6.4.11 Proximal Policy Optimization (PPO)

In this experiment, I used the same environment as in the previous experiment, but I used the Proximal Policy Optimization (PPO) algorithm. PPO is a on-policy policy gradient algorithm. Here too, I use the same parameters as in the previous experiment.

Results: This change of algorithm decreased the agents performance similar to the previous experiment. The agent received an average reward of 72.77 with a standard deviation of 62.31. We can see that PPO is not a good algorithm for this feature extracted environment.

6.5 Results

I have conducted many experiments and found that the best algorithm for this environment is the DQN algorithm with the following hyperparameters: 512x512 architecture, 8x2 observation space, frame stacking, observation normalization, dynamic exploration rate, and scheduled learning rate.

Table 6.1: Comparison of different methods and algorithms (trained for 200k steps)

Experiment	Description	Reward	Relative	Change
Random Policy	Randomly choosing actions	-9.6	-5.27%	Base
Basic DQN	DQN with 64x64 architecture, 17x17 observation space	40.02	21.94%	Improvement
DQN with Frames-tacking	DQN with 64x64 architecture, 17x17x2 observation space, frames-tacking	42.74	23.44%	Improvement
DQN with Observation Normalization	DQN with 64x64 architecture, 17x17 observation space, observation normalization	45.96	25.21%	Improvement
DQN with Observation Normalization and Frames-tacking	DQN with 64x64 architecture, 17x17x2 observation space, observation normalization, framestacking	73.27	40.19%	Improvement
+ larger network	DQN with 512x512 architecture, 17x17x2 observation space, observation normalization, framestacking	89.21	48.94%	Improvement
+ scheduled learning rate	DQN with 512x512 architecture, 17x17x2 observation space, observation normalization, framestacking, scheduled learning rate	113.22	62.11%	Improvement
+ dynamic exploration rate	DQN with 512x512 architecture, 17x17x2 observation space, observation normalization, framestacking, dynamic exploration rate	105.83	58.05%	No Improvement
+ manual feature extraction	DQN with 512x512 architecture, 8x2x2 observation space, observation normalization, framestacking, dynamic exploration rate	182.3	100%	Best
Replace DQN with A2C	A2C with 512x512 architecture, 8x2x2 observation space, observation normalization, framestacking, dynamic exploration rate	66.33	36.39%	Deterioration
Replace DQN with PPO	PPO with 512x512 architecture, 8x2x2 observation space, observation normalization, framestacking, dynamic exploration rate	72.77	29.92%	Deterioration

6.5.1 Why did the DQN algorithm outperform the other algorithms?

We saw that the (relatively) simple DQN algorithm outperformed more sophisticated algorithms such as A2C and PPO. Let us consider some possible reasons for this: First, we are dealing with a relatively small observation space ($8 * 2 * 2 = 32$) and a relatively small discrete action space (4). DQN's primitive exploration and exploitation mechanism is sufficient for this environment. The other algorithms are more complex and require more data to learn from. Therefore, the DQN algorithm outperforms the other algorithms. Second, the DQN algorithm is an off-policy algorithm, unlike the two on-policy algorithms

(A2C and PPO) can learn from exploratory and previous experiences. DQN is more sample efficient.¹⁰ Another reason why other algorithms did not perform as well as DQN is that those algorithms are very sensitive to hyperparameters. The DQN algorithm is more robust to setting hyperparameters.¹¹

6.5.2 Optimized Agent

Since this algorithm outperforms all others in this environment, I will use this algorithm to train the final optimized agent. I trained this agent for 5 million steps. The agent received an average reward of 791.34 with a standard deviation of 142.56. Training this AI took about 4.5 hours on a computer with Intel CPU i5-8250U processor with 1.6 GHz. The following figure shows the reward over time for this agent. Figure 6.7 shows the average reward over time for this agent.

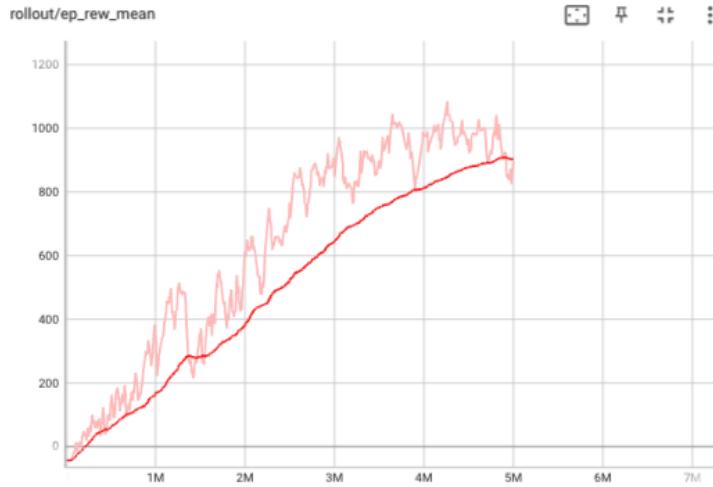


Figure 6.7: Reward over time for the final optimized agent

If you look at how this agent acts visually (see figure 6.8), you can confidently say that it is a good agent that is able to play the game at a pretty good level. Due to the observation limitation, the agent cannot survey the entire playing field and develop a long-term strategy.

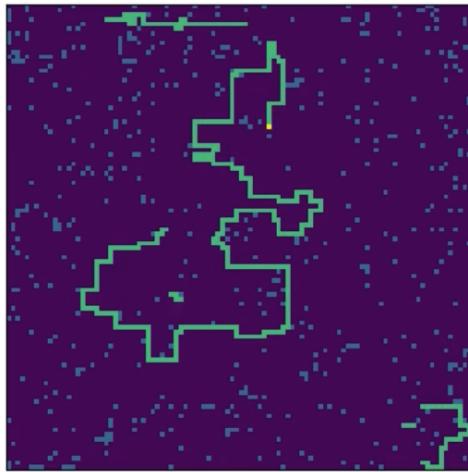


Figure 6.8: The final optimized agent (yellow head) playing a local multiplayer game

¹⁰Tewari, *Which Reinforcement Learning-RL Algorithm to Use Where, When and in What Scenario?*

¹¹Krishna, “COMPARISON OF REINFORCEMENT LEARNING ALGORITHMS”.

6.6 Conclusion

In this project, I created an environment for the multiplayer game Snake. Then I tested different DRL algorithms and different hyperparameters in mini-experiments to find out which combination of algorithm and hyperparameters is most effective for this environment: the following algorithms: DQN, A2C and PPO, different techniques such as frame stacking, normalization of observations, scheduled learning rate, dynamic exploration rate and reduction of observation space by manual feature extraction, network architectures: 64x64 and 512x512 and the following observation spaces: 17x17x2, 8x2x2. I found that the DQN algorithm with a 512x512 network architecture, an 8x2x2 observation space, observation normalization, frame stacking, and dynamic exploration rate was the best algorithm for this environment, achieving an average reward of 791.34 with a standard deviation of 142.56 after 5 million training steps. This agent was successfully able to play the game at a good level.

Video of the final optimized agent: <https://youtu.be/Hu7CmQLyZvc>

6.6.1 Answering the problem statement

The problem for this project was: How can bots be trained with RL to play a multiplayer game in the shortest possible time? There is no universal answer to this question. It depends heavily on the game or the environment. Given my computational and time constraints, I systematically tested different approaches to find out what works and what doesn't. In this way, I was able to reduce training time and increase performance. In this project case, I can summarize the ideas that worked:

- Use robust (not sensitive to hyperparameters) algorithms like DQN.
- Reduce the observation space and complexity of the environment.
- Add input normalization to avoid optimization problems.
- Use frame stacking to provide more past information (memory) for the agent to learn from
- Have a ratio between the dimensionality of the inputs and the number of neurons in the network that is not too large or too small
- Have a reward function that rewards optimal behavior

By implementing these aspects, I was able to train an agent who was able to play the game at a good level after 4.5 hours of training, which is a very good result considering the training time for RL problems.

Chapter 7

Discussion

The goal of this project was to develop a multiplayer Snake game for which AI-controlled bots would be created in an efficient manner. While developing the video game, I was first able to learn networking, game design, and architecture concepts, as well as deepen my Golang and web stack programming experience. In the next part, I explored Deep Learning and Reinforcement Learning and implemented a simple environment. Through trial and error and some research, I was able to initially port my game to a Python version and create simple AI bots. However, these were not very powerful. I began to further optimize my code and environment. I managed to create a bot that worked well, but the way the problem was solved can be further discussed. Although Deep Reinforcement Learning methods have proven to be very effective in many areas, they are not the only way to solve this problem.

Since the rules of the game are simple, one could also use a simple heuristic approach. For example, one could use a simple greedy algorithm to find the shortest path to food while avoiding other players and obstacles. Such an algorithm would not require much computational power and would be much simpler and easier to transfer than a deep learning approach. Another approach to such a game would be an evolutionary algorithm. One could use a genetic algorithm like NEAT to evolve a population of agents and select the best ones. Another idea would be to use data from an already trained AI or human player and teach a more complex AI through transfer learning to learn from the simpler AI's data and then improve the AI's performance. This would be a very interesting approach because it reduces the overall computation and training time.

These are certainly interesting ideas, but they are beyond the scope of this project. Given the time, computational, and data constraints, it was not possible for me to implement these ideas. However, I believe that these approaches could be explored in the future. I feel that in spite of its narrow scope and limitations, this project was a successfull learning experience. Future opportunities for improvement and expansion are available.

List of equations

4.1	Computation of a single perceptron	22
4.2	Binary step function formula	23
4.3	Sigmoid function formula	23
4.4	ReLU function formula	24
4.5	Leaky ReLU function formula with $a < 1$	25
4.6	Mean squared error; y_i is the actual value and \hat{y}_i is the predicted value	27
4.7	Adaptive loss function formula ¹	27
5.1	Expected cumulative reward	32
5.2	Optimal policy which the agent is trying to find	33
5.3	Bellman equation for stochastic environments	33
5.4	Value function for a known reward	34
5.5	Value function rewritten to include the reward of the next state	34
5.6	Bellman optimality condition: Value function rewritten as a recursive function	34
5.7	Quality function	35
5.8	Monte Carlo reward function	36
5.9	Monte Carlo value function approximation	36
5.10	Monte Carlo Q function approximation	36
5.11	TD value function	36
5.12	TD value function approximation	36
5.13	TD(n) value function approximation	37
5.14	Q learning algorithm	37
5.15	Deep Policy Network algorithm	38
5.16	Deep Q-Network algorithm	39
5.17	DDQN Q-function	39
5.18	DDQN actor loss function	39
5.19	DDQN critic loss function derived from the policy gradient equation	39
5.20	Critic approximation algorithm	40
5.21	Actor approximation algorithm	40
5.22	Advantage approximation algorithm	40
5.23	Policy approximation algorithm	40
6.1	Equation for the reward function	44
6.2	Equation for observation normalization (N is the number of observations from which the mean and standard deviation are calculated, x is the observation, μ is the mean, σ is the standard deviation, and x_{norm} is the normalized observation)	46
6.3	Scheduled learning rate for DQN with framestacking and observation normalization	47

¹Barron, *A General and Adaptive Robust Loss Function*.

List of Figures

2.1	Overview of full game and RL bots	6
2.2	Game logic & update	7
2.3	Visualization of the threads for my realtime snake game	9
2.4	Golang Logo and Mascot	10
2.5	Svelte JS Logo	11
2.6	Snake Game Homepage in Firefox	12
2.7	Visualization of the rendering of the game from raw data	12
3.1	Example of an ELIZA conversation	14
3.2	AlphaGo vs. Lee Sedol (the world's best Go player)	14
3.3	Example of Youtube's recommendation system (Youtube algorithm)	15
3.4	Artificial Intelligence subtopics	18
3.5	Overview of the different machine learning subfields	19
4.1	Single layer perceptron	21
4.2	Binary step function in a graph	23
4.3	Sigmoid activation function on a graph	24
4.4	ReLU activation function on a graph	24
4.5	Leaky ReLU on a graph	25
4.6	Overview of common activation functions and their derivatives	26
4.7	General (adaptive) loss function (left) and its gradient (right) for different values of its shape parameter α . Several values of α reproduce existing loss functions: L2 loss ($\alpha = 2$), Charbonnier loss ($\alpha = 1$), Cauchy loss ($\alpha = 0$), Geman-McClure loss ($\alpha = -2$), and Welsch loss ($\alpha = -\inf$)	27
4.8	Good vs poor local minimum	29
4.9	Global vs local minimum	30
4.10	Example function with two parameters	30
4.11	Optimization algorithms try to find the global minimum efficiently in a much higher-dimensional and complex space.	30
4.12	How different learning rates affects gradient descent	30
5.1	Markov decision process illustrated	32
6.1	Local architecture of the game	44
6.2	Comparison of the baseline DQN with the DQN with framestacking and observation normalization. (Orange: Baseline DQN, Red: DQN with framestacking and observation normalization)	46
6.3	Comparison of different network architectures for DQN with framestacking and observation normalization. (Orange: 64x64, Blue: 512x512)	47
6.4	Comparison of small DQN (64x64, Yellow), big DQN (512x512, Red) and big DQN with scheduled learning rate (512x512, Blue)	48
6.5	Comparison of DQN without dynamic exploration rate and DQN with dynamic exploration rate. (Blue: DQN without dynamic exploration rate, Yellow: DQN with dynamic exploration rate)	48

6.6	Comparison of different observation spaces for DQN with framestacking and observation normalization. (Black: 17x17x2, Blue: 8x2x2)	49
6.7	Reward over time for the final optimized agent	51
6.8	The final optimized agent (yellow head) playing a local multiplayer game	51

Bibliography

12. *Optimization Algorithms — Dive into Deep Learning 1.0.0-Alpha1.Post0 Documentation.*
https://d2l.ai/chapter_optimization/index.html.
- 12.3. *Gradient Descent — Dive into Deep Learning 1.0.0-Alpha1.Post0 Documentation.*
https://d2l.ai/chapter_optimization/gd.html.
- 6.6 *Actor-Critic Methods.* <http://www.incompleteideas.net/book/ebook/node66.html>.
- 8 *Ways to Effectively Reduce Server Response Time.*
<https://datadome.co/learning-center/how-to-reduce-server-response-time/>.
- A *Smoother Ride and a More Detailed Map Thanks to AI.*
<https://blog.google/products/maps/google-maps-101-ai-power-new-features-io-2021/>. May 2021.
- Abadi, Martin et al. “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems”. In: (), p. 19.
- Activation Functions in Neural Networks.* Jan. 2018.
- Activation Functions in Neural Networks [12 Types & Use Cases].*
<https://www.v7labs.com/blog/neural-networks-activation-functions>,
<https://www.v7labs.com/blog/neural-networks-activation-functions>.
- AI, SmartLab. *Reinforcement Learning Algorithms — an Intuitive Overview.* Feb. 2019.
- AlphaGo.* <https://www.deepmind.com/research/highlighted-research/alphago>.
- Barron, Jonathan T. *A General and Adaptive Robust Loss Function.* Apr. 2019. arXiv: 1701.03077 [cs, stat].
- “Bellman Equation”. In: *Wikipedia* (May 2022).
- Bhattacharyya, Saptashwa. *The Most Awesome Loss Function ?*
<https://towardsdatascience.com/the-most-awesome-loss-function-172ffc106c99>. Sept. 2020.
- blackburn. *Introduction to Reinforcement Learning : Markov-Decision Process.*
<https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da>. May 2022.
- Blog, Windows Developer. *When to Use a HTTP Call Instead of a WebSocket (or HTTP 2.0).*
<https://blogs.windows.com/windowsdeveloper/2016/03/14/when-to-use-a-http-call-instead-of-a-websocket-or-http-2-0/>. Mar. 2016.
- Brockman, Greg et al. *OpenAI Gym.* June 2016. arXiv: 1606.01540 [cs].
- “Brute-Force Search”. In: *Wikipedia* (Nov. 2021).
- “Canvas Element”. In: *Wikipedia* (June 2022).
- Causevic, Siwei. *A Structural Overview of Reinforcement Learning Algorithms.*
<https://towardsdatascience.com/an-overview-of-classic-reinforcement-learning-algorithms-part-1-f79c8b87e5af>. Aug. 2021.
- Choudhary, Ankit. *Dynamic Programming In Reinforcement Learning.* Sept. 2018.
- chowdhury, sourajit roy. *Demystifying Activation Functions in Neural Network.* Jan. 2021.
- Common Loss Functions in Machine Learning / Built In.*
<https://builtin.com/machine-learning/common-loss-functions>.
- Computer, Express. *What If AI Becomes Self-Aware?* Dec. 2021.
- Darbinyan, Rem. *Council Post: The Growing Role Of AI In Content Moderation.*
<https://www.forbes.com/sites/forbestechcouncil/2022/06/14/the-growing-role-of-ai-in-content-moderation/>.
- Deep Learning vs. Machine Learning: Deep Dive.*
<https://www.educative.io/blog/deep-vs-machine-learning>.

- DeepLearningAI. *Why Non-linear Activation Functions (C1W3L07)*. Aug. 2017.
- “ELIZA”. In: *Wikipedia* (Sept. 2022).
- Evolutionary Optimization Algorithms / Wiley*.
<https://www.wiley.com/en-us/Evolutionary+Optimization+Algorithms-p-9780470937419>.
- experience, nature He believes everyone is a learning et al. *Difference Between Strong and Weak AI / Difference Between*.
- Fig. 1. A Typical Energy Landscape Depicting Position of Several Local...*
https://www.researchgate.net/figure/A-typical-energy-landscape-depicting-position-of-several-local-minima-which-indicate-the_fig3_45900660.
- Frost, Jim. *Mean Squared Error (MSE)*. Nov. 2021.
- “Garbage (Computer Science)”. In: *Wikipedia* (May 2022).
- Garbage Collection Comp Sci Wiki - Google Search*.
<https://www.google.com/search?q=garbage+collection+comp+sci+wiki>.
- “Go (Programming Language)”. In: *Wikipedia* (Sept. 2022).
- Gupta, Ayush. *A Comprehensive Guide on Deep Learning Optimizers*. Oct. 2021.
- Gupta, Mayank. *Getting Started with Svelte*. Aug. 2020.
- Hardt, Moritz. *Understanding Optimization in Deep Learning by Analyzing Trajectories of Gradient Descent*. <http://offconvex.github.io/2018/11/07/optimization-beyond-landscape/>.
- Harris, Charles R. et al. “Array programming with NumPy”. In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL:
<https://doi.org/10.1038/s41586-020-2649-2>.
- Hern, Alex. “AlphaGo: Its Creator on the Computer That Learns by Thinking”. In: *The Guardian* (Mar. 2016). ISSN: 0261-3077.
- “History of Artificial Intelligence”. In: *Wikipedia* (Sept. 2022).
- Hunter, J. D. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.
- IBM100 - Deep Blue*. <http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/>. CTB14. Mar. 2012.
- Jagtap, Rohan. *Understanding Markov Decision Process (MDP)*.
<https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150>. Feb. 2021.
- Johnson, Jonathan. *4 Types of Artificial Intelligence*.
<https://www.bmc.com/blogs/artificial-intelligence-types/>.
- Kalavala, Bala. *AI, ML and DL: What's the Difference?* Aug. 2022.
- Karunakaran, Dhanoop. *The Actor-Critic Reinforcement Learning Algorithm*. Sept. 2020.
- Krishna, Velivila Vamsi. “COMPARISON OF REINFORCEMENT LEARNING ALGORITHMS”. In: (), p. 30.
- Kvam, Jacques. *Celluloid*. Sept. 2022.
- Leung, Kenneth. *The Dying ReLU Problem, Clearly Explained*.
<https://towardsdatascience.com/the-dying-relu-problem-clearly-explained-42d0c54e0d24>. Sept. 2021.
- “Markov Decision Process”. In: *Wikipedia* (Aug. 2022).
- Mnih, Volodymyr et al. *Playing Atari with Deep Reinforcement Learning*. Dec. 2013. arXiv: 1312.5602 [cs].
- “Multilayer Perceptron”. In: *Wikipedia* (Sept. 2022).
- “Outline of Machine Learning”. In: *Wikipedia* (Sept. 2022).
- Papers with Code - Leaky ReLU Explained*. <https://paperswithcode.com/method/leaky-relu>.
- Paszke, Adam et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- “Perceptron”. In: *Wikipedia* (Aug. 2022).
- Raffin, Antonin et al. “Stable-Baselines3: Reliable Reinforcement Learning Implementations”. In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL:
<http://jmlr.org/papers/v22/20-1364.html>.
- Rangaiah, Mallika. *Artificial Intelligence in Healthcare: Applications and Threats / Analytics Steps*.
<https://www.analyticssteps.com/blogs/artificial-intelligence-healthcare-applications-and-threats>.

- Reducing Loss: Learning Rate / Machine Learning.*
<https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate>.
- ReLU — PyTorch 1.12 Documentation.*
<https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html>.
- Setting the Learning Rate of Your Neural Network.* <https://www.jeremyjordan.me/nn-learning-rate/>. Mar. 2018.
- “Shannon Number”. In: *Wikipedia* (Sept. 2022).
- SHARMA, SAGAR. *Activation Functions in Neural Networks*.
<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>. July 2021.
- Shyalika, Chathurangi. *A Beginners Guide to Q-Learning*.
<https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c>. July 2021.
- “Sigmoid Function”. In: *Wikipedia* (Sept. 2022).
- Silvan. *What Is the Difference between Artificial Intelligence, Machine Learning and Deep Learning?* Aug. 2019.
- Silver, David. “Lecture 2: Markov Decision Processes”. In: *Markov Processes* (), p. 57.
- Single Layer Perceptron in TensorFlow - Javatpoint*.
<https://www.javatpoint.com/single-layer-perceptron-in-tensorflow>.
- Skowster the Geek. *Bellman Equation Basics for Reinforcement Learning*. Sept. 2018.
- start, Lou Bichard Hey I’m Lou! I’m a Cloud Software Engineer From London I. created Open Up The Cloud to help people get their et al. *Golang Logo » Open Up The Cloud*.
<https://openupthecloud.com/wp-content/uploads/2020/01/Golang.png>.
- Steve Brunton. *Model Based Reinforcement Learning: Policy Iteration, Value Iteration, and Dynamic Programming*. Jan. 2022.
- *Overview of Deep Reinforcement Learning Methods*. Jan. 2022.
 - *Q-Learning: Model Free Reinforcement Learning and Temporal Difference Learning*. Jan. 2022.
- Stewart, Matthew. *Neural Network Optimization*.
<https://towardsdatascience.com/neural-network-optimization-7ca72d4db3e0>. July 2020.
- Svelte • Cybernetically Enhanced Web Apps*. <https://svelte.dev/>.
- Svelte, Why so Much Hype ?* <https://dev.to/zenika/svelte-why-so-much-hype-2k61>.
- Tanwar, Sanchit. *Bellman Equation and Dynamic Programming*. Jan. 2022.
- Team, Towards AI. *Reinforcement Learning: Monte-Carlo Learning – Towards AI*.
- Tewari, Ujjwal. *Which Reinforcement Learning-RL Algorithm to Use Where, When and in What Scenario?* <https://medium.datadriveninvestor.com/which-reinforcement-learning-rl-algorithm-to-use-where-when-and-in-what-scenario-e3e7617fb0b1>. Apr. 2020.
- Top Applications of Artificial Intelligence (AI) in 2022*.
<https://www.interviewbit.com/blog/applications-of-artificial-intelligence/>. Jan. 2022.
- Trekhleb, Oleksii. *Homemade Machine Learning in Python*.
<https://medium.datadriveninvestor.com/homemade-machine-learning-in-python-ed77c4d6e25b>. Feb. 2019.
- “Turing Test”. In: *Wikipedia* (Sept. 2022).
- Uber AI in 2019: Advancing Mobility with Artificial Intelligence*.
<https://www.uber.com/blog/uber-ai-blog-2019/>. Dec. 2019.
- Understanding the 4 Types of Artificial Intelligence (AI)*.
<https://www.linkedin.com/pulse/understanding-4-types-artificial-intelligence-ai-bernard-marr>.
- Understanding The Difference Between AI, ML, And DL: Using An Incredibly Simple Example*.
<https://www.advancinganalytics.co.uk/blog/2021/12/15/understanding-the-difference-between-ai-ml-and-dl-using-an-incredibly-simple-example>.
- Understanding the Four Types of Artificial Intelligence*.
<https://www.govtech.com/computing/Understanding-the-Four-Types-of-Artificial-Intelligence.html>. Nov. 2016.
- Van Rossum, Guido and Fred L. Drake. *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 1441412697.
- Wang, Chi-Feng. *The Vanishing Gradient Problem*.
<https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484>. Jan. 2019.
- Weak vs. Strong AI*. <https://www.citibeats.com/center-for-knowledge/weak-strong-ai>.
- “WebSocket”. In: *Wikipedia* (Aug. 2022).

- What Is a Client?* <https://www.computerhope.com/jargon/c/client.htm>.
- What Is a Thread?* <https://www.computerhope.com/jargon/t/thread.htm>.
- What Is Compile?* <https://www.computerhope.com/jargon/c/compile.htm>.
- What Is Deep Learning? / How It Works, Techniques & Applications - MATLAB & Simulink.*
<https://www.mathworks.com/discovery/deep-learning.html>.
- What Is Go? An Intro to Google's Go Programming Language (Aka Golang).*
<https://aclougdguru.com/blog/engineering/what-is-go-an-intro-to-googles-go-programming-language-aka-golang>. May 2021.
- What Is Reinforcement Learning? Working, Algorithms, and Uses.*
- What Is the Turing Test?* <https://www.techtarget.com/searchenterpriseai/definition/Turing-test>.
- What Is Weak Artificial Intelligence (Weak AI)? - Definition from Techopedia.*
<http://www.techopedia.com/definition/31621/weak-artificial-intelligence-weak-ai>.
- What's the Difference between Being Statically versus Strongly Typed?*
<https://www.quora.com/What's-the-difference-between-being-statically-versus-strongly-typed>.
- Which Is Greater? The Number of Atoms in the Universe or the Number of Chess Moves? / National Museums Liverpool.* <https://www.liverpoolmuseums.org.uk/stories/which-greater-number-of-atoms-universe-or-number-of-chess-moves>.
- Yalçın, Orhan G. *4 Machine Learning Approaches That Every Data Scientist Should Know.*
<https://towardsdatascience.com/4-machine-learning-approaches-that-every-data-scientist-should-know-e3a9350ec0b9>. Feb. 2021.
- Yoon, Chris. *Double Deep Q Networks.*
<https://towardsdatascience.com/double-deep-q-networks-905dd8325412>. July 2019.
- YouTube.* <https://www.youtube.com/>.

Originality statement

I hereby declare that I have written this thesis independently and have not used any sources, aids or assistants other than those indicated. All text passages in the paper that have been taken verbatim or in spirit from sources are marked as such.