Deep Q Network

In Non-Deterministic Simulation

Jostein Haugland Dyrseth

April, 2019

0.1 Authorship Declaration

I, Jostein Haugland Dyrseth, confirm that this dissertation and the work presented in it are my own achievement. Where I have consulted the published work of others this is always clearly attributed; Where I have quoted from the work of others the source is always given. With the exception of such quotations this dissertation is entirely my own work; I have acknowledged all main sources of help; If my research follows on from previous work or is part of a larger collaborative research project I have made clear exactly what was done by others and what I have contributed myself; I have read and understand the penalties associated with Academic Misconduct. I also confirm that I have obtained informed consent from all people I have involved in the work in this dissertation following the School's ethical guidelines.

Signed:

Date: April 3rd 2019

Matriculation Number: 40223535

0.2 General Data Protection Regulation Declaration

Under the General Data Protection Regulation (GDPR) (EU) 2016/679, the University cannot disclose your grade to an unauthorised person. However, other students benefit from studying dissertations that have their grades attached.

The University may make this dissertation available to others, but the grade may not be disclosed.

Signed:

0.3 Acknowledgements

I would like to express my appreciation and give many thanks towards my supervisor, Dr Simon Wells for the sincere support throughout the project to discuss and plan new ideas, methods and techniques as well as giving me honest feedback, inspiration and motivation to continue even in the most difficult moments.

I also want to give thanks to fellow co-students and colleagues for their honest and objective feedback and assessment on the project.

Further, I would give gratitude towards my family for supporting me mentally over many phone calls, giving me encouragement in my endeavors. I am thankful that they support me in doing my studies abroad and that they always want me to succeed.

Abstract

The aim of the project is to investigate, research and analyse machine learning techniques and methods and more specifically deep neural networks. This knowledge will be applied to build a system from ground up to then make an agent perform in that environment or simulation. The focus then will be to observe the agent, optimize it and analyse the rewards, learning rate and fitness of the agent given different parameters.

The main findings from the project were a well structured game and simulation with an extensive set of user features and an comprehensive algorithm and method that can outperform other tested methods. Other things were the agent perception and understanding of the immediate world, a fitness function and an exploration vs exploitation trade-off. The neural network structure is to some extent configured, changed and tested to see what effect it had. Many hyper-parameters were made available to the user so that they can easily be explored.

The conclusion of the project is that there are a lot of possible machine learning tools and libraries available to do almost anything. The idea of combining neural networks and building tensor pipelines were understood and appreciated. Some of the findings are to some extent surprising.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Brief Overview

A common problem in computational intelligence and machine learning is to make precise and predictive models in order to solve specific tasks. This can generally be difficult to implement, but even more so challenging to get right as there are countless barriers to consider.

There has been an immense amount of research in deep learning in recent years and for good reason. Better hardware than ever before makes anyone with the help of an internet connection able to produce undiscovered applications and systems like never seen.

The main focus in this project will be to research and collect information and understanding on how to use neural networks in a generic environment. This will then be applied and simulated to get more familiar with the field of machine learning in general.

The final challenge of the project is to give a well defined and logical explanation as to why this is and to find variations, contrasts and characteristics amongst the models. This can further be examined, and possibly be given a good conclusion for.

Consequently, the ultimate effect is to contribute to the scientific community so that we can with our outcome, provide a more accurate explanation and a better understanding of combining several methods and techniques to artificial neural networks.

1.2 Aims & Objectives

The aim in this experimental project is to build a system that can automatically solve a specific task in a simulation. It was decided that a very simple game should be used as the core

of the simulation system. The complexity should be simple from the start and as time goes, more layers of complexity could rather be introduced to the system if I will have time.

It was decided that the simulation should be based on the classical arcade game, Snake. This was a very popular game in the 90's especially on Nokia mobile phones.

1.3 Deliverables

- Study relevant literature on ML and AI on ensemble neural networks and present it in form of a Literature Review.
- Design and build the game and simulation.
- Test the game thoroughly.
- Design and build an Agent to play the game.
- Design and build the neural network.
- Implement a good algorithm/algorithms in the agent.
- Analyse and monitor the performance of the agent.
- Observer patterns, behaviours and strategies of the agent.
- Make console user features: print score, replay, save model etc.

1.4 Target Audience

The findings from the work in the final paper can be beneficial to other computer science students, postgraduates and professionals. It will apply to the wast field of machine learning, but probably closer to the branch of deep neural networks and to some extent the field of data science. The audience can also be for science teachers and lecturers, but also private businesses.

1.5 Technical Overview

The system makes use of multiple languages, modules and packages as well as open source libraries available to the public. The system is built using the Python programming language. Some of the libraries used were TensorFlow, Keras for the neural network parts as well as numpy and pandas amongst others for data handling and data visualisation. These technologies will be used to implement the simulated environment using methods like Q-learning and Deep Q Network, with help of the Bellman Equation amongst others.

1.6 Project Scope & Boundaries

1.6.1 Scope

The work will include finding a few fairly simple both single and possibly multiplayer games to build and apply our artificial neural networks (ANN) design on. It is important to realise what games are good from an implementation, training and testing point of view. We will first see if we can implement an ANN with the cost variable being completely random. Once this is done we have to tweak the neural network so that it will create a good function for us to return an optimum from the input we feed it.

Finally the last part will focus on experimenting with making an even better collection- or ensemble neural network that will on average outperform the individual networks on a general level where the environment is non-deterministic.

Different numbers of hidden layers, different number of nodes per layer, different types of nodes, different types of strengths and/or of weight penalty and different learning algorithms. All this will be tweaked in the ANN to increase the strength and fitness as best as possible. The performance, efficiency and general findings needs to be analysed, compared and evaluated to support our initial goals.

We will also include the overall time-complexity of the program and what we possibly could have done further to both optimise it and make it more efficient to locate the global optimum on finite computing resources.

1.6.2 Boundaries

We will however not cover evolving neural networks on alternative computational platforms like con- currency nor in parallel on multicore processors, even though the latter would be very possible and even more so appropriate as we want to collect an ensemble of neural networks that will be trained individually, hence conceivably in parallel.

1.6.3 Constraints

1.7 The Importance of the Project

As this is an experimental approach for approximating a good neural network, there will be focus on different design principles of ANN to find a satisfactory model in the end to solve a particular problem in the snake game. The importance of this project is severe as there could be discovered new and fundamental theories and techniques to solve problems in an ever better fashion.

1.8 What I Hope to Learn

Some of the things I would like to learn and get a broader understanding of under the course of this project is to introduce myself to machine learning. More specifically how neural networks works under the hood. The concept is fascinating and really shows how far computer intelligence have come today.

1.9 Research

Some research, reading and consideration needs to take place before and during the work that will be undertaken to understand the particular field and it's pre-existing tools and techniques that are available. Techniques like Bagging and Boosting needs to be investigated, as well as potentially other more recent technologies that just have been published to the general public. Pre-existing knowledge and theory will be applied to new tools and libraries to build and implement the networks. The technologies that will be chosen is not yet fully identified.

1.9.1 Research Questions

Some core questions were researched in the Literature Review in the next chapter.

- What technologies and techniques can be used to achieve an optimal solution?
- Are there previously proven work that have been performing significantly?
- Can some of those researched methods be combined into an ensemble or pipeline to outperform individual methods?

These were broad questions that needed a careful approach to arrive to the right conclusion. We want to explore as wide as we can, but also to narrow it down so that we also have time to go into depth. This was the underlying plan when undertaking the literature review.

Also previously made findings, claims, questions and ethics were reviewed to familiarise the reader with a broad subject area, narrowing the domain down as time goes.

Chapter 2

Literature Review

2.1 Introduction

Before going too deep into what methods and techniques one can use in order to create more accurate results using an ensemble learner, we should cover some more general terms and concepts in AI, Machine Learning and Neural Networks as well as some of the past research and findings in this vast field. In order to achieve this, a literature review were systematically structured from a general perspective, before narrowing down the scope towards some more specific terms, concepts and ideas that were more relevant in this study.

2.2 History of AI and ML

2.2.1 Introduction

Ever since scientists have had access to modern computers (20th century) there have been countless attempts to calculate solutions to ever more complex problems. Scientists have had many breakthroughs with the help of computers in general over the decades, some of them we will cover in this paper. However, there are just as many obstacles we are facing as the field of computing is getting ever more mature. This will also lead to many philosophical questions, dilemmas and debates that must be further studied.

2.2.2 René Descartes

Some questions were: *"Will machines ever be able to think like humans?"* or *"Can a machine **be** a human"* (*The 10 most important breakthroughs in Artificial Intelligence.*). These

questions were first recorded by René Descartes, the father of modern philosophy in his book "Discourse on the Method" in 1637.

2.2.3 Alan Turing

Later, one of the most remarkable philosophical leaps was made in 1950 by mathematician and scientist Alan Turing. Turing is said by many to be the person that truly invented the computational age. He came up with a concept called the Turing Test or The Imitation Game. This game will test whether or not the any computer can fool the subject as being a human and not a computer:

> *"A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer."* (Russell, 2018)

This test has been recognized as a very important and interesting question in many debates ever since. Even though it has received a lot of criticism from both philosophers and scientists, it still stays relevant to this day. The idea came from trying to define intelligence itself, and whether or not a computer can be called intelligent if it passes the test.

He was also known for defining when a machine is theoretically capable of solving any problem, as a *Universal Machine*. This came to be known as a Turing Machine. A machine that as long as you had the program, any task could be written and solved.

2.2.4 The Earliest Computers

At the same time Turing tried to answer important philosophical questions, scientists in Manchester were working on a computer that would come to be known as The Manchester Baby. This machine was in many ways the machine that had all the fundamental components in order to be called a "modern electronic computer" as it achieved an effective random access memory.

Neither the two ruling machines at the time in the mid 1900s had this very technology (RAM), hence they were quickly being outdated as The Manchester Baby appeared to be the one advancing.

Harvard Mark 1 was also being built in the USA, being 50 feet long. This machine could make calculations in seconds, that would normally take people hours to complete (Watson,

2012). Also Turing later built a machine the "Bomb" that would be used to solve the German Navy's Enigma code during WW2.

2.3 Definition of AI

Even though the field of AI has been around from just after the WW2, the term was not coined before over a decade later in 1956. It is crucial to understand the definition of what AI is, before proceeding. This field holds a lot of sub-fields, so the term artificial intelligence is often seen as a very general umbrella term including sub-fields like Machine Learning, Natural Language Processing, Vision- & Speech Recognition and Robotics to name a few.

It's really challenging to find areas where AI is not applicable:

> *"AI is relevant to any intellectual task; it is truly a universal field."* (Russell, 2018)

Many scientists have tried to give AI a definition, and some of them are:

> *"The study of the computations that make it possible to perceive, reason, and act."* (Winston, 1992)

> *"Computational intelligence is the study of the design of intelligent agents. An agent is something that acts in an environment — it does something. Agents include worms, dogs, thermostats, airplanes, humans, organizations, and society."* (Poole, 1998)

> *"The study of mental faculties through the use of computational models."* (Charniak and McDermott, 1985)

According to the Oxford University Press, AI can be defined as the following:

> *"The theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation between languages."* (*Artificial Intelligence Oxford Definition*)

As we have gained some insight into what artificial intelligence is, some subsequent and well known issues, questions and debates will follow.

2.4 Future of Artificial Intelligence

2.4.1 Adversarial AI

As new technologies have emerged there are usually an adversarial group that warn against it, as it can bring dangerous and unforeseen consequences to society. There will always be conflicts when new and more powerful technologies emerges.

Many top AI researchers are concerned for what AI can bring in the future (*Benefits And Risks of Artificial Intelligence*). It is important that AI systems are not able to get hacked as this could result in dangerous situations in cases where pacemakers, cars and airplanes can behave in undesired ways.

As more complex AI systems gets access and autonomy over cars, planes and drones if the goal of this AI is not good enough defined, there can indeed be devastating consequences that can potentially take human lives. This is why it is so important to always consider all possibilities and research security in all computer systems.

> *"Humans don't generally hate ants, but we're more intelligent than they are – so if we want to build a hydroelectric dam and there's an anthill there, too bad for the ants."* (*Benefits And Risks of Artificial Intelligence*)

Types of AI

There are generally three levels of AI. Narrow AI, general AI and super AI. Most systems today operate very much in a narrow way only completing small and repetitive tasks, but in a very precise and efficient manner.

There is a debate amongst researchers that discuss whether or not we ever will achieve super intelligence. Some argue it is impossible while others say it will happen in our lifetime:

> *"While some experts still guess that human-level AI is centuries away, most AI researches at the 2015 Puerto Rico Conference guessed that it would happen before 2060. Since it may take decades to complete the required safety research, it is prudent to start it now."* (*Benefits And Risks of Artificial Intelligence*)

Ethics

In this study, addressing some ethical questions in regards to AI is also appropriate. Questions like: Just like we domesticate animals that we are superior to in order to meet our needs, is it ethical to let other more intelligent systems domesticate or destroy us? This very topic is addressed by Yuval Noah Harari in his book *Homo Deus*:

> *"In recent years, as people began to rethink human-animal relations, such practices have come under increasing criticism. We are suddenly showing unprecedented interest in the fate of so-called lower life forms, perhaps because we are about to become one. If and when computer programs attain superhuman intelligence and unprecedented power, should we begin valuing these programs more than we value humans? Would it be okay, for example, for an artificial intelligence to exploit humans and even kill them to further its own needs and desires? If it should never be allowed to do that, despite its superior intelligence and power, why should it be ethical for humans to exploit and kill pigs?"* (Harari, 2016)

OpenAI

OpenAI is a non-profit organisation based in San Francisco with a hundred employees where their goals are to achieve artificial general intelligence (AGI) that will outperform humans. They care about safety and want to build an open and free source of this kind to advocate and aid users to use it to help acheive their goals in AI.

> *"OpenAI's mission is to ensure that artificial general intelligence (AGI)—by which we mean highly autonomous systems that outperform humans at most economically valuable work—benefits all of humanity. We will attempt to directly build safe and beneficial AGI, but will also consider our mission fulfilled if our work aids others to achieve this outcome." ()*

2.4.2 Will we adapt with the AI?

Other experts in the domain, argue that humans will indeed develop *with* artificial intelligence systems, hence in a way becoming - or at least utilising artificial intelligence to some extent arguably through a brain interface. If this is true, the AI threat would to some extent decrease as we merge with it.

2.4.3 AI Regulation

Additional ethical debates are cases where companies with the use of AI tools advances in decision making, but are not necessarily able to portray or present exactly *why* this decision was made - the company can technically be accused of breaking the law - as they should be able to justify indeed why this decision was made. This s exactly what is seen today in advanced neural networks. No one knows exactly *why* the network behaves as it does.

As more complex AI is developing, we need to ask ourselves ever more difficult, morally challenging and ethical questions in regards to computer ethics in terms of privacy, security, safety and authenticity.

Some other questions that are heavily being debated frequently are: Should robots have rights like humans do? Should they have the same rights? What happens when the machines steals all the jobs? How can individuals have a stable economy? How can we distribute the wealth that machines are making? And, how do we as humans stay safe and in control of any intelligence system? (Bossmann, 2016b)

2.5 Future Challenges

2.5.1 Problem Complexity

Any machine learning task has a certain complexity. The complexity of a problem is the number of instructions a computer needs to compute. The complexity of any intelligence problem has often been the bottleneck for more significant breakthroughs in Machine Learning.

> *"Since the development of the digital computer in the 1940s, it has been demonstrated that computers can be programmed to carry out very complex tasks — as, for example, discovering proofs for mathematical theorems or playing chess — with great proficiency."* (Copeland, 2018)

2.5.2 Moore's Law

Here Moore's Law have played a role in predicting when certain milestones potentially could have been achieved or not. In recent years, Moore's Law have become more history than a practicality. The fact that researchers have reached the very physical limits of how small a transistor can be, has lead to a plateau in what Moore has been promising.

Back in the 1980s there were basically two components that was missing to achieve greater advancement in the field of Machine Learning. (1) available data storage, as well as (2) the raw computational power needed to calculate complex and intricate problems.

Because of this demand of ever more powerful computers, scientists now need to develop alternative designs in order to continuously increase the complexity of computers.

2.6 What Are Neural Networks

2.6.1 Overview

Neural networks are in need of powerful and robust computers, and as discussed, computers are slowly reaching their maximum physical complexity. Therefore, we want to look at other ways in which we can continue to improve our designs - often improving the software or optimising the algorithms being used.

In this paper the latter will be explored as an attempt to achieve precisely this. The goal of the study is to optimise the neural networks being used with a technique called *ensemble learning*. This technique will be covered later in this paper, as the nature of neural networks needs to be discussed.

Artificial neural networks in many ways are the same as a biological neural network in human and animal brains. The network is essentially an electrical circuit of neurons or nodes that takes an input and gives an output. The relations between these nodes can get stronger, or weaker during training. The weight between them can be positive (expiatory connection) or negative (inhibitory connections). These ANNs are used to solve computational intelligence tasks as they can "learn" just as humans do. The neurons or nodes could give binary outputs, but usually they give a scalar between -1 and 1.

2.6.2 Artificial Neural Network Model

Artificial neural networks have been around since 1943 when neurophysiologist Warren McCulloch and mathematician Walter Pitts came up with a system called threshold logic. This system was inspired by how biological brains work. These computational neural networks can be less complex and of a much smaller scale than that of a biological brain. The networks are generally used as a way to try to approximate a function to map to the solution.

A neural network is essentially built up of a set of **nodes** and **weights**. The nodes are the

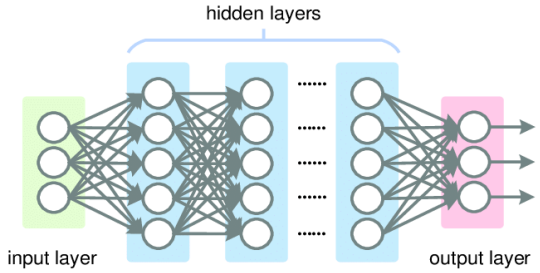


Figure 2.1: *Neural Network Model*

the basic building blocks and are usually grouped in layers as seen in Figure 2.1. A single node has multiple input values and one single output value. In a classical *dense feed-forward neural network*, any node's input values comes from all the nodes in the previous layer.

The node will add all the input values or *weights* together including a *bias*, and if that summed value is above a given threshold, it will *fire*, meaning it will send that summed value as an output to all the nodes in the next layer.

The activation function is just simply checking if the output value is above a given threshold, if it is it will be *fired*, or *activated*.

> *"The honeymoon is officially over, and neural computing has moved beyond simple demonstrations to more significant applications."* (Sharkey, 1999)

> *"It has been shown that the robustness and reliability of an ANN (artificial neural network) can be often significantly improved by appropriately combining several ANN models into an ANN ensemble. The construction of such an ensemble requires two main steps. The first is to create individual ensemble members, and the second is to find the appropriate combination of outputs from those members for producing the unique ensemble output."* (Linares-Rodriguez et al., 2013)

In this experiment (Linares-Rodriguez et al., 2013), the goal was to obtain more reliable results in regards to estimating global solar radiation from satellite images by combining the ANNs. The results from the five best models that were selected were compared. Now, after analyzing the data given, an ANN-ensemble of these individual ANNs were constructed and built.

This demonstrates just how combining models can outperform individuals. This can lessen the overall bias to any specific model, hence make the ensemble more fair and reliable.

Simulations

According to (Woodford and Plessis, 2018), performing experiments in a virtual world or a simulation, often helps speed up the process of learning artificial neural networks.

> *"The evaluation of many controllers is typically performed in a simulation instead of real-world in order to speed up the evolutionary process."* [...] (Woodford and Plessis, 2018).

2.7 Ensemble Neural Networks

Ensemble Learning is a method within the field of Machine Learning where one tries to combine several models. This can help to obtain a result or output that is less bias, more accurate, decreased variance and better predictive performance. By having people use ensemble methods many teams get's placed first in many prestigious machine learning competitions, such as the Netflix Competition, KDD 2009, and Kaggle.

> *"The idea of ensemble learning methods is to select a collection, or ensemble, of hypotheses from the hypothesis space and combine their predictions."* (Russell, 2018)

> *"Normally, as you increase the complexity of your model, you will see a reduction in error due to lower bias in the model. However, this only happens till a particular point. As you continue to make your model more complex, you end up over-fitting your model and hence your model will start suffering from high variance. "*

2.8 The NEAT Library

NEAT is a 'NeuroEvolution of Augmenting Topologies'-library for Python that uses a generic algorithm to do just this – to approximate an optimal neural network structure. The algorithm tries to evolve not only the network itself, but can also evolve the actual weights too, making this an extensive, very useful and two-in-one solution library. This was considered strongly in the project, but unfortunately TensorFlow seemd to be the winning technology for this time. However, this library was definitely added to the MoSCoW table for the future work section.

Chapter 3

Methodology

3.1 Overview

In order to proceed with the project, numerous applications, methods, languages, tools, libraries, algorithms and technologies were considered. The various methodologies were briefly described, discussed and justified in order to select the most appropriate ones in accordance with the task at hand.

3.2 The Original Plan

The original plan was to make an ensemble learner. Ensemble learning is a method for combining several trained computer models to more precisely achieve a goal. These techniques could be used by computers competing against humans in simulations or video games, but also to solve challenges in the physical world.

Originally an ensemble learner was the main aim of the project to experiment with changing the neural network structure, adapting it and making different models and then see if the ensemble would outperform the individual designs. The trained neural network ensemble would hopefully prove to run slightly more efficiently and accurately, given that the agent's environment will change or be non-discrete.

However, the scope of this was a bit too much, and so I decided to narrow it down and change it very subtly. Instead of introducing myself to the world of neural networks *before* anything else, I went "head first" into the vast field of machine learning confident that I could take on anything, and definitely this interesting piece of work. The reality is that the scope was indeed quite a bit too broad, leaving me having to narrow down the scope and pivot the

project ever so slightly in order to actively manage the project responsibly.

Some work in the Literature Review was focused on the ensemble part, however the ensemble requires several neural network models anyway, which I have achieve, but not the very last part to actually *combine* them.

3.3 Project Plan

In order to actively manage

3.4 MoSCoW Method

To visually represent the project and clearly show the goal, scope and boundaries, the well known MoSCoW method was been used. In Figure 6.1 there are four categories *must haves*, *should haves*, *could haves* and *won't haves*.

Must	Should	Could	Won't/Future
Game	Unit tests	Graphical Interface (2D)	3D Game Interface
Simulation	Find Best Game	Build Pipeline	CNN
Agent	Replay	Ensemble ANN	3D Interface
Artificial Neural Network	Analysis	TDD	GPU/TPU Compatibility
Algorithm	Manual Game Play	Step-Functionality	Multiple console output
Console Interface			Multi-Agent System
API			Interactive (Curses library)
			Tensorboard
			NEAT

Figure 3.1: *MosCow Table*

It was decided that basic components that are fundamentally necessary for the system to work were selected for the *must have* category. Objects like the game and/or the simulation itself, an agent, a neural network, an underlying algorithm, a basic console interface and an API were all selected for this category where it is absolute vital that we get produced. If we get all these components integrated and nothing else there should be at the very least a working prototype of what is being proposed. However we want to make sure that these features in this category gets produced well before the deadline as there could be unforeseen circumstances and potential risks as this is a project spanning over at least seven moths.

3.5 Gantt Chart

To be clear and precise in the planning, there were created a Gantt Chart for the project. This will help with setting up a plan that will be followed as close as possible. However there are very high chances that the project will indeed change as new things emerge. Basically the further into the future we get, the higher are the chances that something will change.

Knowing this, we will highlight the main tasks in grey and treat these as the underlying milestones we want to achieve. If something changes, it should not change the project too much when reaching the next milestone. This will reduce the possible number of ways the project can change.

The blue bars are the task-duration showing the corresponding number, title, start and due date, the estimated number of days to achieve the task and the current completion.

#	MILESTONES & TASKS	START DATE	DUE DATE	DURATION	COMPLETE
1	Project Conception and Initiation				
1.1	Project Planning	9/3/18	9/19/18	16	100%
1.2	Basic Research	9/7/18	9/23/18	16	100%
1.3	Project Deliverables	9/13/18	9/18/18	5	100%
1.4	Write Initiation Project Overview	9/13/18	9/26/18	13	100%
2	Project Definition and Planning				
2.1	Goal Setting	9/24/18	10/3/18	9	100%
2.2	Scope & Boundaries	10/1/18	10/13/18	12	100%
2.3	Constraints	10/1/18	10/13/18	12	100%
2.4	Risk & Project Control	10/7/18	10/21/18	14	100%
3	Research Phase				
3.1	Previous & Current Research	10/18/18	10/25/18	7	100%
3.2	Neural Networks	10/21/18	11/9/18	18	100%
3.3	Ensembles	10/21/18	11/14/18	23	100%
3.4	Technologies & Tools	10/26/18	11/26/18	30	100%
3.5	Draft Literature Review	10/28/18	11/30/18	32	100%
4	Game Development				
4.1	Investigate similar games	11/4/18	11/15/18	11	100%
4.2	Investigate game design	11/12/18	11/24/18	12	100%
4.3	Make Class Diagram	11/14/18	11/21/18	7	100%
4.4	Make Design	11/22/18	11/30/18	8	100%
4.5	Implement Game	11/26/18	1/21/19	55	100%
4.7	Game Unit Testing	1/16/19	1/29/19	13	100%
5	Neural Network				
5.1	Agent Implementation	1/10/19	1/25/19	15	100%
5.2	Neural Network Implementation	1/17/19	2/6/19	19	100%
5.3	Testing	2/3/19	2/12/19	9	100%
6	Analysis & Evaluation				
6.1	Implementing Evaluation Scripts	2/12/19	2/24/19	12	100%
6.2	Running Evaluation	2/13/19	3/13/19	30	100%
6.3	Document Outcomes	2/25/19	3/13/19	18	100%
7	Documentation & Report				
7.1	Inline Documentation	3/13/19	3/15/19	2	100%
7.2	Report	3/13/19	3/29/19	16	100%

Figure 3.2: *Gantt Chart: Project approach and plan of execution. The area highlighted in red is the break between the trimesters.*

The seven milestones identified as can be seen in Figure 2 are the Project Conception and Initiation, Project Definition and Planning, Research Phase, Game Development, Neural Network, Analysis & Evaluation and Documentation & Report.

3.6 Version Control

It was decided that version control should be used in this project. Most, if not all software projects today are taking advantage of the incredible features from version control. It is generally strongly advised to use version control as a central way to organise repositories - big or small - to secure it and to hold a certain professional standard. Using version control,

the project will act as a *backup solution* amongst many other features like being able to *undo*, *redo*, *branching*, *merging*, see who made a change, where, when and why amongst numerous others.

3.6.1 Git

Git is one, if not the best version control system available to date. It has the above mentioned features amongst many others. Being that I am personally familiar with Git, it was decided to use it as well as saving the repository at *GitHub*. Holding as much as possible of the project in one repository was done. However the dissertation itself was held in a separate repository liked with *Overleaf*.

3.7 Overleaf

Overleaf, formerly known as ShareLaTeX is an online web-app. The app lets anyone create and edit text files of any kind that will compile and render the text in real time to PDF format. This makes many aspects of handling a large text document a lot easier.

This online application is completely open source and free, it is very modular in that the user can include new packages and modules at any time.

Referencing was also used as a tool in this application. The reference with it's attached attributes like type of reference, name of authors, title, date of publication, journal, publisher, volume and pages cited and so on.

3.8 Software Development Methodologies

A different range of software methodologies were investigated and researched in order to come closer to a good workflow before starting the actual development.

3.8.1 Waterfall

One of the most traditional software development models is the Waterfall method as seen in Figure 3.3. The waterfall model is a sequential model and has been used in many companies earlier, and have failed due to not being sufficient enough when it comes to continually expanding an application or system. The model only matches more precise needs, it is not so

good with adapting to ever changing needs. The model has also shown to not be a great fit for projects that are of a significant size.

Waterfall works best with carefully planning and analysing the project at first and then deigning and building it, testing and correcting it before deploying it. The phases must be fully completed before one can move to the next phase and after the team have gone from one phase to another, it is generally difficult to go back to modify features or the directions as this will most likely change all the following phases too.

> *"The Waterfall model is often slow and costly due to the rigid structure and tight controls. These drawbacks led waterfall method users to the explore other software development methodologies."*

There are generally many other better models to organise and execute software development projects today, one of which leads us to the Agile methodology.

3.8.2 Agile Methodology

The Agile Methodology is an iterative and incremental method - not a sequential. Most updates happen in small incremental sprints as seen in Figure 3.3 often as short as two weeks. This makes the team more adaptive to change from clients.

The Agile methodology has a well known manifesto (Beck et al., 2001). The people behind the manifesto says that they have developed a better way to develop software. They also try to help others adapt and use their model. The manifesto states four central ideas for developers, teams and clients:

> *"Individuals and interactions over processes and tools.*
> *Working software over comprehensive documentation.*
> *Customer collaboration over contract negotiation.*
> *Responding to change over following a plan."* (Beck et al., 2001).

Agile development praises individuals and interactions over processes and tools. The team knows that humans are the creators of the tools and that the tools are just that - tools. They can fail and so can humans, but the humans are the underlying foundation of the team and should be valued more.

The method also values software that is working, clearly written and easy to understand, instead of complicated code with complicated documentation.

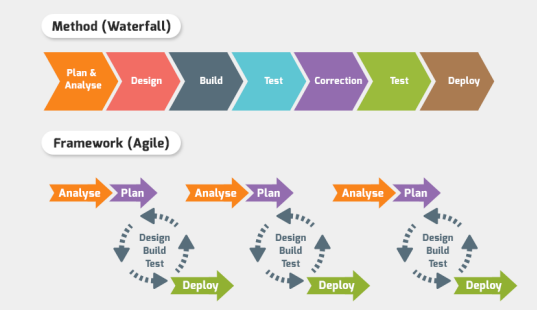


Figure 3.3: *The agile methodology as opposed to the more traditional waterfall method. Source: https://www.quickscrum.com/Article/ArticleDetails/2026/1/Is-Scrum-a-methodology*

Also, the manifesto says that teamwork and collaboration should be more valued than negotiating a good contract with the client, as this most likely could change during the time of execution.

Finally the last point is to be adaptive to change. The world we live in today is changing at a rapid phase, making project plans avoid changes and modifications along the way inevitable.

There are numerous other methodologies like Scrum, Extreme Programming, Rapid Application Development, Feature Driven Development and many more.

3.9 Project Management

3.10 Tools & Technologies

Having a clear set of deliverables and plan, we need to investigate all tools and technologies to use.

Doing research there were found an interesting and fairly new technology which was noted. This was the new Unity Machine Learning Library. Looking deeper into the documentation many aspects looked great, but the documentation was not very well structured and it seemed like the product was in very early stages. If using this library the system or simulation needs to be built in the language C# or JavaScript. These programming languages are somewhat familiar, but not a whole lot.

Other sources that were investigated were Python's extensive libraries.

Unity Unity Machine Learning Python, C Scikit-learn Tensorflow/Keras (beginner friendly,) PyTorch AWS, Google Collab

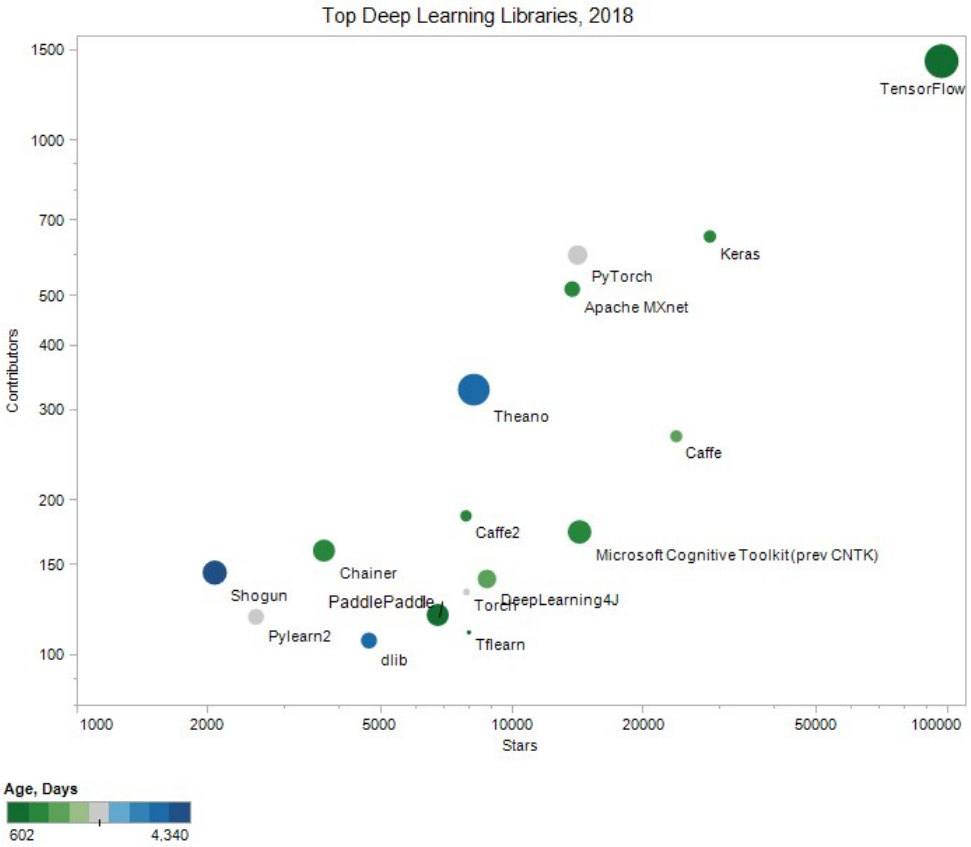


Figure 3.4: *Source: https://www.kdnuggets.com/2018/04/top-16-open-source-deep-learning-libraries.html*

3.10.1 Applications

3.10.2 Languages

3.10.3 Libraries

TensorFlow

> *"TensorFlow is an open source software library for numerical computation using dataflow graphs. Nodes in the graph represents mathematical operations, while graph edges represent multi-dimensional data arrays (aka tensors) communicated between them. The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API."*

Other libraries were also researched, but after creating a table and putting together the notes, TensorFlow was the one which was the most interesting. Tensorflow is fully GPU and TPU compatible, it is developed by Google itself (a very trusted organisation), the library is used by a huge group of people every day. It is also very beginner friendly. On the front pages of TensorFlow, there are links to tutorials, videos, beginner scripts and example implementa-

Deep Learning Libraries					
Name	**GPU/TPU**	**B.F.**	**Docs**	**Lang**	**Catch**
NEAT	Unknown	Yes	Good	Python	Genetic
Theano	GPU	No (Research)	Alright	Python	mathematical
Torch	GPU	No (Research)	Extensive, Messy	C/C++	speed
Tensorflow	GPU&TPU	Yes!	Very Good	Python	general purpose
Caffe	Unknown	OK	Good	C++/Python	speed, modularity, CNN
Scikit-Learn	Yes	OK	Good	Python	SL/UL
Unity ML	No	No	Poor	C	3D

Table 3.1: Caption2

tions of neural network models. All these features drastically reduces the *risk* of the project, knowing that potential helpful resources are not too far away.

> *"We can also easily distribute this processing across our CPU cores, GPU cores, or even multiple devices like multiple GPUs. But that's not all! We can even distribute computations across a distributed network of computers with TensorFlow. So, while TensorFlow is mainly being used with machine learning right now, it actually stands to have uses in other fields, since really it is just a massive array manipulation library."* (Kinsley, 2016)

The fact that TensorFlow is using Python is also a plus, as I have used it before and am familiar with the underlying fundamentals. Python is generally very respected in the field of Data Science as well as it is making its way into machine learning and reinforcement learning.

3.10.4 Unit Testing

unit testing

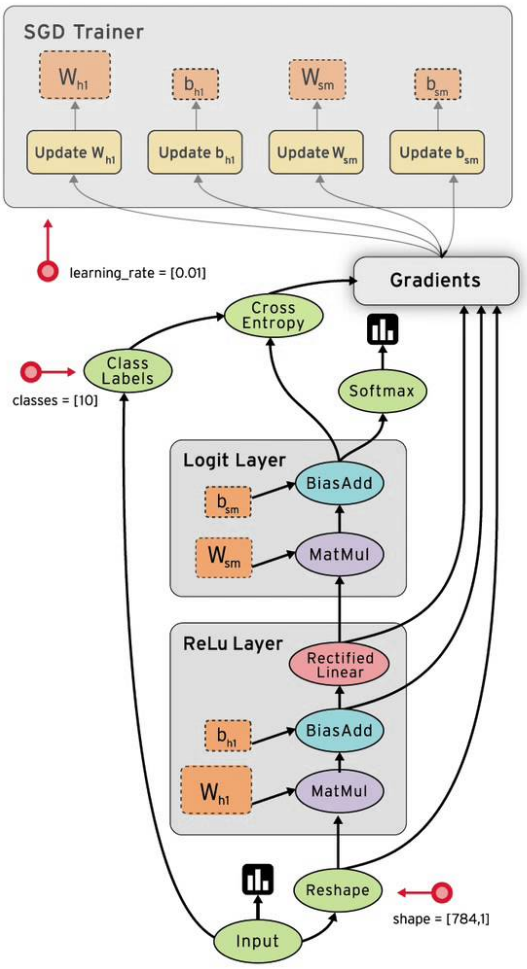


Figure 3.5: *This illustrates how data in forms of Tensors can flow in a network of connected operational nodes. The graph can send data in many directions to be processed at different locations all in a modelled pipeline before finally a prediction will be made as the resulting output.*

Chapter 4

Solution Implementation

4.1 Overview

Before going into the detail of the game, some basic and underlying guidelines and best practices were investigated.

Guidelines & Best Practices

The Unix community have over the years since the start in the late 1970s, developed a philosophy for its operating system and its developers. The philosophy includes ideas like modular design, simplicity and reusability. An inventor of Unix tools (McIlroy, 1978), wrote the forewords a journal article under the AT&T Corporation (founded by Scottish Graham Bell) says this:

> *(i) Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.*
> *(ii) Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
> *(iii) Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.*
> *(iv) Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.*

Doug McIlroy is pointing out many interesting and powerful concepts here. Some of his

ideas are making the output of one program the input of another, building small, but capable, modular and reusable designs and to use tools to test your system in an early age and not being afraid to throw away the clumsy parts if they don't do the job.

Other characteristics that have been pointed out by other Unix members are clarity over cleverness, design for simplicity; add complexity only where you must, write a big program only when nothing else will do, design for visibility to make inspection and debugging easier (Bossmann, 2016a).

> *Software is said to be robust when it performs well under unexpected conditions which stress the designer's assumptions, as well as under normal conditions.* (Bossmann, 2016a)

This will be followed up with tests in the next chapter.

Also, in Python there is a a well know philosophy or a set of principles which is called The Zen of Python. This was published by Tim Peters and the 19 rules are as follows:

> *Beautiful is better than ugly.*
> *Explicit is better than implicit.*
> *Simple is better than complex.*
> *Complex is better than complicated.*
> *Flat is better than nested.*
> *Sparse is better than dense.*
> *Readability counts.*
> *Special cases aren't special enough to break the rules.*
> *Although practicality beats purity.*
> *Errors should never pass silently.*
> *Unless explicitly silenced.*
> *In the face of ambiguity, refuse the temptation to guess.*
> *There should be one– and preferably only one –obvious way to do it.*
> *Although that way may not be obvious at first unless you're Dutch.*
> *Now is better than never.*
> *Although never is often better than *right* now.*
> *If the implementation is hard to explain, it's a bad idea.*
> *If the implementation is easy to explain, it may be a good idea.*
> *Namespaces are one honking great idea – let's do more of those!*

Python also has a set of Python Enhancement Proposals (PEPs) which is a number of pro-

posals that is being accepted or rejected by the elected body. These are also guidelines for developers to follow to make the design of systems hold a certain standard.

Although not all rules and principles can or will be followed to the very core at all times, they are sources that any programmer should strive to follow. Most of these are being used under the development of this project as a means to follow best practices.

4.2 Introduction

In order to achieve the prepared deliverables for this project, there needs to be implemented several modules in the system. Some of the modules are:

- Environment (Game)
- Snake (Agent)
- Brain (Neural Network)
- Communicator (API)

The environment in our system is the game or simulation, the brain is the artificial neural network, the snake is an Agent and the communicator is what is called an API.

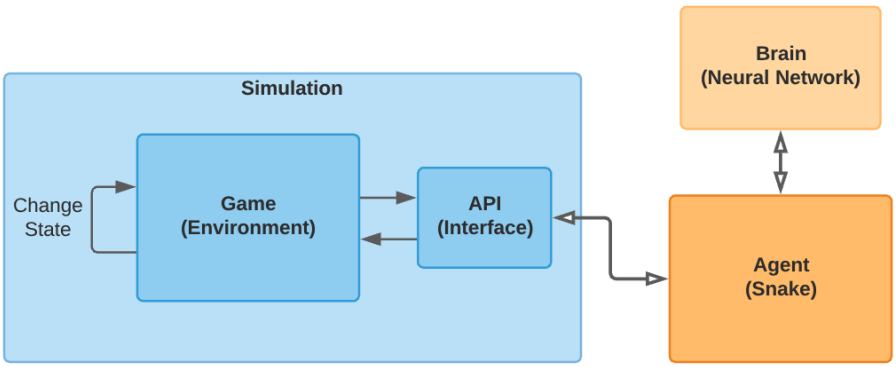


Figure 4.1: *The environment is the game, also holding an interface. This let's ideally any kind of agent influence or alter the environment.*

Ideally, we want to construct this simulation so that it is compatible with any kind of agent and also any kind of environment. That is why it is important to follow certain best practices and design principles. This is also how OpenAI's *Gym* environments are designed.

Basically, any game can be broken down to a series of discrete states. The simulation is not going to try to sense the real world, using sensors like cameras or microphones, but rather have virtual sensors that is being mimicked in a computer system environment or game. OpenAi quotes:

These environments have a shared interface, allowing you to write general algorithms. ()

OpenAI's gym environments all have a *step* function. This function will, return four values: the *discrete state* of the environment in a given snapshot, a *reward*, a *done* variable, and a *info* dictionary.

It was decided that this project should hold similar design-structure and logic, and hold these variables similar to the OpenAI environment.

In order to make this system, we are dependent on the environment that the agent can act upon, meaning this should to be implemented before any other module, or at the very least along with the other parts.

There were set off time to research multiple implemented snake games that could work for this purpose. Multiple games were considered strongly. See references (TODO: include refs). However, after all consideration, it was decided that the game needed to be implemented from scratch. The justification for this is manifold:

- being able to provide flexible and sensible data to the neural network
- being able to communicate between the environment and the neural network
- step-by-step design, not continuous
- games found had unclear and unreadable code
- being able to fully debug and inspect data

To be able to provide flexible and appropriate data to the network, one should have this in mind while designing the game from the ground up. Many of the games that was inspected had some data mined (state), but not necessarily for instance every *state* of every *move* of every *game*, therefore this was not flexible enough. Also some games provided irrelevant data and data that was not required.

To be able to communicate between the game and the network, one often needs an interface. This could have been implemented in any snake game, but was rather designed along with the game itself to make sure all design principles were followed.

The game needed to have a step-by-step design instead of a continuous classical game-loop. What is mean by this is: the system don't need to worrie about finding a move in a sertain timeframe. Rather, the CPU can take it's time to find the optimal action as well as train the agent incrementally before performing the action. Also, there is no need to visually see the

game while being played, as there could be inplemented replay and step by step functionality, leaving the system to calculate and train as quick as possible.

If a later scenario, if an interactive game is needed for humans to play as well, another version can be made.

4.3 Snake Game Design & Implementation

Snake is a classic arcade game, that was very popular in the 90s on mobile phones, but originated in 1976 by Germlin Industries.

4.3.1 Game Rules and Goal

- Eating an apple will make the snake grow
- Hitting the walls will end the game
- Hitting itself will end the game

The goal of the game is to eat as many apples as possible to increase the score, without hitting the walls or itself. The game is technically *solved* when the snake fills up the whole grid, leaving nowhere for the apple to spawn.

Although the snake game is being used as a base, to build our simulation, some rules can changed to meet our interests. Ideally we want to move beyond the game itself, and make it more of a simulation where we even can introduce multiple agents, essentially making a multi-agent system, where we can monitor their behaviours.

4.4 The Game Implementation

We want to make the game object as logical as possible to hold relevant data that we can later retrieve to analyse and improve the performance between the games. The game should hold all game states from every single step that the game-loop creates. Each game should also hold a Board, Apple and a Snake. It will also hold data like: *score* and *game_steps* and have methods to update, get the current state and check for food.

When structuring the game, it is important to know what data from the game we need to give to the neural network. Being able to provide what the neural network wants is vital.

4.4.1 Game Loop

Because the simulation is doing the same things over and over again, there needs to be implemented a loop. In many games, this is referred to as the *game-loop*. In out game-loop we want the game to do three main tasks over and over again:

- Get action from agent
- Update the game state
- Draw the game

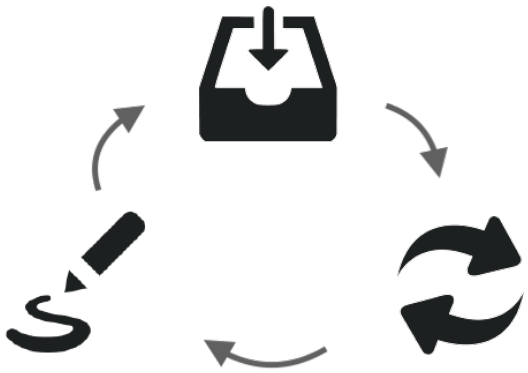


Figure 4.2: *Get action from agent, update the game state and finally draw the game.*

Right before the loop however, the game gets initialized, and it will run the very first or initial update as seen in algorithm 1 on line 2-3. Then inside the game-loop the game will get the previous state based on the *previous action*. It will then request and receive an *action* from the neural network based on that *previous state*. This will interface with the agent object - which ideally could live on a separate server somewhere. Then the agent executes the *action* onto the environment by updating the game (line 7). It also gets the *current state*. On line 9 it gets the *reward*, and updates the neural network's *weights* by doing the training.

It will lastly check if it has reached the maximum number of *game steps*. This is to avoid the snake to spin in circles. And finally the *epsilon* value will be multiplied by its *hyperparameter factor*, which will decrease the *epsilon-value*, hence making the agent less likely to make a random action in the following loop-iterations.

To put it other terms, when the agent must do an action onto the environment it will request a direction to receive from its *brain component* (neural network). Once the agent holds a current direction it wants to move in which is relative to the snake ('forward', 'right' or 'left'), it will act upon the environment by executing that move command i.e. *update('forward')*. The game will update itself and continue forever, as long as the agent don't move outside the board, move into itself or reaches the maximum *game steps* limit.

Algorithm 1: Game Loop

Result: Game

```
1  while not games > MAX_GAMES do
2      game ← Game()
3      game.initial_update()
4      while not game_over do
5          prev_state ← game.get_state(prev_action)
6          action ← agent.get_action(prev_state)
7          game.update(action)                          // perform a move
8          curr_state ← game.get_state(action)
9          reward ← snake.get_reward(done)
10         agent.train()                          // train the neural network
11         if steps > INTERATIONS then
12             game_over ← True
13         if epsilon > MIN_EPSILON then
14             epsilon ← epsilon * EPSILON_FACTOR
15     steps++
16     agent.replay_new()
```

If the current game happens to come to an end, the system will immediately add that game to the *all_games* list, which is a property of the simulation object. The simulation will then immediately start a new game unless the achieved number of games are reached (line 1) in algorithm 1. When the number of iterations or games are reached, the system will end.

On ending the system, the simulation will save the list of *all_games* to a database, save the *model* that the neural network have trained (weights-database), save information to a csv file which later can be displayed as a graph. This will give good insight into the simulation. It will finally print an overview to the console as well as to a log-file.

4.4.2 Class Diagram

Modular Design

We want to follow the guidelines from the Unix Philosophy described above, making the simulation modular, scaleable and generic. This will make the system more compliant with concepts like cloud computing and distributed systems.

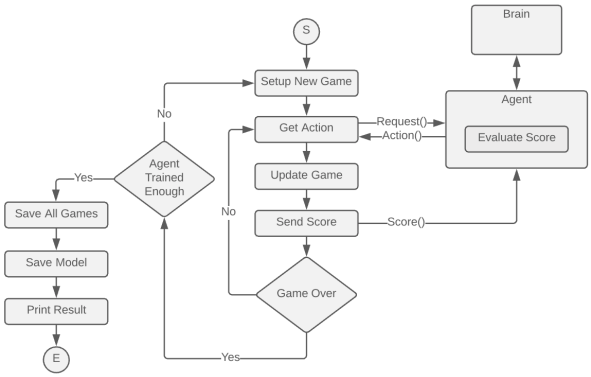


Figure 4.3: *Game Loop*

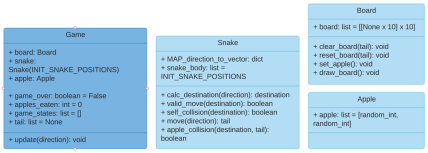


Figure 4.4: *The Game is holding a Board, a Snake and an Apple. There is also a Simulation class and an Agent Class.*

4.4.3 Get State

We want to be able to get the state of each and every step of the game. This will finally be the *input* to the neural network. We make a method *get_state()* in the game-class. This makes a list *state* of six boolean values (True/False) to make it readable. Then before feeding the list, the values will be converted to pure numbers, 1s or 0s.

The data the snake needs to know about is only the dangers and the apple. If it knows something about these objects and can distinguish them, it will hopefully be able to make predictions in regards to where to move, which is going to be the output of the neural network; the action. The action will be the relative direction the snake will move in, from the snakes perspective.

Deciding how and what data or information from the game should be extracted and put into the neural network is important. We want the snake to build knowledge or rules from the information we feed it. Ideally the snake should not get any more information than what is fair. The snake should not be able to perceive the whole board. It should not even see exactly where the walls are, or where the apple is.

The snake should only be able to see the *dangers* that are immediately next to him, however for the apples it is able to locate from any distance as long as the snake is either facing that direction or is crossing the horizontal or vertical line as shown in Figure 4.6.

The input is as follows:

Figure 4.5: *The current state is an example from the second image in Figure Figure 4.6. Here there are* ***immediate*** *dangers in front and to its left as well as indicating that there is an apple to its right.*

- (1) Is there any danger immediately in front of the snake?
- (2) Is there any danger immediately to the right of the snake?
- (3) Is there any danger immediately to the left of the snake?
- (4) Is there any food at some location to its front?
- (5) Is there any food at some location to its right?
- (6) Is there any food at some location to its left?

In total there are six binary booleans which will be encoded into a list of True or False, or more specifically 1 or 0, as this is what the neural network wants. Later in this project we can look into maybe passing in scalar values, instead of just binary values, as that could lead to more accurate predictions.

4.4.4 Check for Food

For each step of the game the game have to tell the snake if there is either any food or danger in it's proximity or vision. We want to let the snake have limited vision, which will hopefully increase the interest of the system, not giving the snake any kind of super powers. We want to let the snake only see from its perspective. If a human were to play snake however, the human would in fact have 100% vision of the full board, which arguably is unfair. The snake literally lives in a 2D world and can only see in its world in first person, at least that is what we want for this system, making the whole design more like a simulation than a game.

In this figure, the first three values of the state represents the danger and the following three represents the if there are any apples.

4.5 Neural Network Implementation

> *"When it is not in our power to determine what is true, we ought to act in accordance with what is most probable."* (Descartes, 2018)

The artificial neural network to be used in this project will *only* predict future actions, and

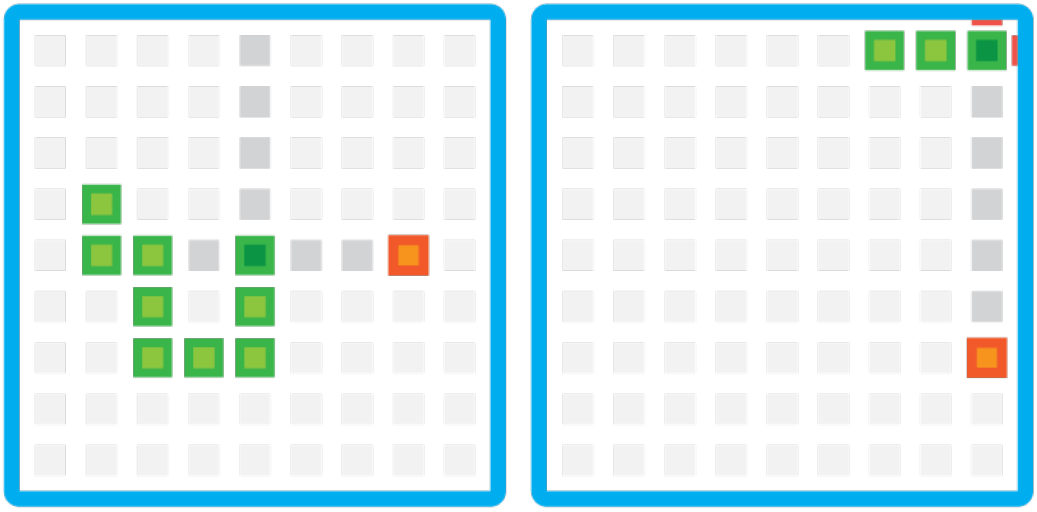


Figure 4.6: *This figure illustrates the vision of the snake, checking for dangers and for food in* ***front****, to its* ***right*** *and to its* ***left****. The first state (left) will result in a state of: [0, 0, 0, 0, 1, 0,]. The second state will result in a state of: [1, 0, 1, 0, 1, 0].*

will never *truly* understand the absolute optimal solution ever. It will only *approximate* it. The solution or *model* which is the result, is simply just a function or a set of weights, which we will come back to.

Now that we have the environment implemented, our next step is to implement the Agent and it's Brain (neural network). The Brain of the Agent is the system that is responsible for predicting and providing the best actions to the agent to let it act upon the environment.

When developing ML software which are very computational heavy, it is important to value the overall performance of the system, and try to best optimise it. The better the performance, the faster the training can occur. Different designs can change the complexity in different orders of magnitude. We also want to test this later on a GPU and a distributed TPU cloud service to (amongst other things) profile the performance and identify bottlenecks. We can later target these potential bottlenecks and search for ways to optimize these.

4.5.1 Reinforcement Learning

So as discussed, both supervised and unsupervised learning is all about pattern recognition. Classifying or clustering data, predicting for instance what an object is on a given image, given some possible labels. However, in reinforcement learning, it is more about trial-and-error over labels. Or to be more precise it is about extracting *labels* from the environment in real-time (unsupervised), to feed the agent, as opposed to feeding the agent from a pure pre-made database (supervised).

Just like anyone learns to ride the bicycle, reinforcement learning agents learn the same way,

by trial and error or feedback from the environment. If one falls off the bicycle it probably hurts, hence receiving a negative reward or penalty, contrariwise if one succeed or manages to both steer the bicycle whiles giving speed one receives a positive reward. The is really how simple reinforcement learning is when we break down all parts in the simulation.

So, instead of using labeled data or unlabeled data, we want to use *rewards* to train our agent. The reward given to the agent is always a result of the *state-action* pair present in the environment. However, the reward with the state-action pair could be seen as *unlabeled data*.

We also want to make the agent's algorithm as *general* as possible, making it close to solving other similar board games in the future if need be. This is often more challenging and time consuming to develop.

A paper from Google's DeepMind (Mnih et al., 2015), is stating that a team of researchers have been able to make an agent learn in different Atari games with complete different environments. This is arguably one of the first steps towards *general artificial intelligence*. Basically the agent was able to play 50 different Atari Games as well as achieve superhuman performance in all of them. The new algorithm they come up with was named **Deep Q Network**, where the agent is essentially just watching the screen (with a convolutional neural network - CNN) and then making predictions about the rules and what to achieve in order to get the highest accumulated future reward. This started a new door of possibilities essentially mixing more traditional reinforcement learning methods *with* deep learning using neural networks - getting best of both worlds. All this also makes it arguably a beginner friendly method as complexity can further be added. And worst case scenario there would be something to fall back on.

Reinforcement learning interests us more than *supervised learning*, as supervised learning needs lots of training- and testing data - which we ideally want to avoid. This training-data is often made by having humans play the game many times, accumulating a data-set. Or it is made from a more traditional AI-algorithm bot which is essentially playing the game to collect the training data, to then be trained by the neural network to try and mimic that AI-bot, which will never really achieve any higher score than that of the player itself. This solution is arguably not as clever as achieving a pure reinforcement learning agent fully modeling the world and making the rules by himself without the use of pre-collected data.

What the goal is in our case, is to optimize the game as much as possible, hence we want to look at what options we have for reinforcement learning, but first we will look at one of the most important and central concept in reinforcement learning: the Markov Decision Process.

Markov Decision Process

Looking closer at the Markov Decision Process in Figure 4.7, we can actually understand that both the agent and the environment are indeed just functions for one another. The environment takes an action as an input and outputs both a state and a reward. The agent takes this data (state and reward) and gives back an action again. A San Francisco based company, Skymind says this:

> *"Environments are functions that transform an action taken in the current state into the next state and a reward; agents are functions that transform the new state and reward into the next action. We can know the agent's function, but we cannot know the function of the environment."* (Skymind, 2018)

It is important to understand that the environment-function is indeed the function that the agent is trying to predict or mimic, hence this is the unknown function that we will try to approximate.

In any Markov Decision Process there are a few important concepts, which we will look closer at:

- Search Space
- Exploration vs. Exploitation
- State
- Reward
- Action
- Model
- Policy

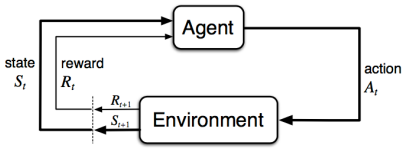


Figure 4.7: Markov Decision Process: The environment is often referred to as a black-box. This just means that the environment function is unknown and the only thing we can see is the input and output.

Any update that the agent takes upon the environment, will occur sequentially over time, feeding back the state of the environment and the reward to the agent as seen in Figure 4.7.

> *"Reinforcement learning represents an agent's attempt to approximate the environment's function, such that we can send actions into the black-box environment*

that maximize the rewards it spits out." (Skymind, 2018).

Search Space

The 3D graph in Figure 4.8 is a visual representation of what could be the *search space* of the environment; all possible states the agent can be in. The vertical z-axis is the reward given by the current state and action. The agent is often referred to as being *greedy* since it always wants to optimise and achieve the best reward possible.

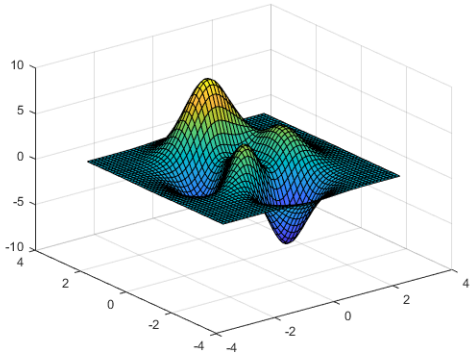


Figure 4.8: Search Space: The Snake agent must learn the correct policy in order to always return the maximum reward of the search space. If the policy happen to be perfect, the policy will map directly to the terrain. Source: https://www.mathworks.com/help/matlab/visualize/changing-surface-properties.html

Exploration vs. Exploitation

Exploration is to look in unexplored regions of the *search space* (often very or completely random actions) and *exploitation* is to focus the search in a particular region of the search space (often predicted actions).

In reinforcement learning it is often important to *explore* the environment first to let the agent get a basic understanding of what *state-action* pairs gives a good reward and what *state-action* pairs can give back a negative reward or a penalty. Later, as the agent collects more and more data, it can start to trust itself more and more. Controlling this there were introduced an *epsilon* value to the agent, to balance exploration vs. exploitation. This will be further discussed later.

This strategy will hopefully make the agent learn quickly. This is why it is normal to often set the whole matrix or model of initial weights to be completely random numbers, and then for each step of the game-loop the agent will tweak these values to optimise its policy.

State

As already discussed in the game-section, the state is just a representation of what the snake can perceive in each snapshot of the game.

The current state of the game will be sent from the environment to the agent. This is what the agent will feed the *Brain*, hence making it interpret the best action to take for the agent. The action will then be sent back to the environment, and so it goes.

Reward

The reward will be calculated by the environment, giving feedback to the agent on how well it did given the state it was in. If the agent hits the apple it gets a high reward (+10), and if the agent hits the wall it gets a low penalty (-10).

Later in the project, more complex rewards can be introduced. For instance to let the agent get a small reward for moving closer to the apple. It could get the score of the apple divided by the current length to the apple. The agent could also get a small negative reward for going in a repetitive circle, not exploring the environment.

Action

The action is returned by the agent either being random or predicted. The action is *relative* to the snake's direction on the board and can always only be "*forward*", "*right*", or "*left*".

Model

The returning value from the neural network will be a model. The model is essentially just the set of weights from the neural network which was initialised and changed throughout the simulation, this can also be referred to as a *function*, *solution* or *policy*.

Curriculum Learning

Curriculum Learning is when the environment changes between each episode, to make the agent learn more efficiently. In the start, the task is very easy to solve while gradually changing to a more difficult task between the episodes. This can automatically be controlled by hyper-parameters in the game. An example in the snake game could be to increase the size

Rank/Dimensions	Math Entity
0	Scalar (magnitude only)
1	Vector (magnitude and direction)
2	Matrix (table of numbers)
3	3-Tensor (cube of numbers)
n	n-Tensor (cube of cubes)

Table 4.1: Caption

of the board, spawn more apples or initialize the snake with an increasing number of body segments.

> *"It is often difficult for agents to learn a complex task at the beginning of the training process. Curriculum learning is the process of gradually increasing the difficulty of a task to allow more efficient learning."* (Juliani, 2017)

> *"Humans and animals learn much better when the examples are not randomly presented but organized in a meaningful order which illus- trates gradually more concepts, and gradu- ally more complex ones."* (Bengio et al., 2009)

Tensor

A tensor is simply a multi-dimensional array of numbers. To get more perspective: a *scalar* is a single real number (0-Dimensions), a *vector* is a set of numbers (1-Dimension), a *matrix* is a set of vectors (2-Dimensions) and a *tensor* is a set of matrices (N-Dimensions).

Tensors are what TensorFlow is dealing with. The input is always going to be a tensor of data. In our case we only feed it single scalar numbers to each input. The tensor object in TensorFlow is accessed using the `tf.Tensor` call.

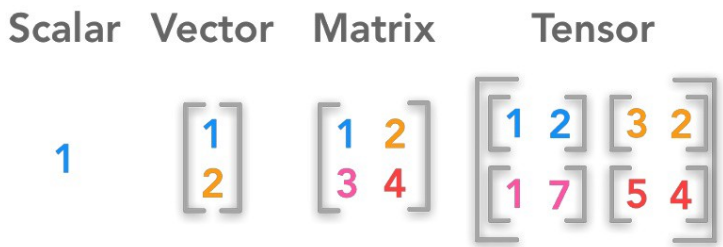


Figure 4.9: *Tensor*

4.5.2 Q-learning

Before going into the Deep Q Learning we want to inspect the traditional Q-learning algorithm.

> *"Q-learning is a simple way for agents to learn how to act optimally in controlled Markovian domains. It amounts to an incremental method for dynamic programming which imposes limited computational demands. It works by successively improving its evaluations of the quality of particular actions at particular states."* (Watkins Dayan, 1992)

The Q-learning algorithm is a reinforcement learning algorithm. Its overall goal is to find an optimal policy in order to act upon the environment in a desired manner to always try to return the most accumulated future reward. The algorithm uses a matrix or a look-up table often referred to as the *Q-table* - Q for *quality*. Each row in the table represents the state with each column representing the actions. Each value in the table is referred to as a *Q-value*. The Q-values represent the *reward* that the agent will receive by taking that action. These values can be initialized randomly between 0 and 1 or just simply as zero.

The concepts most often in Q-learning are:

- **Q-table** (the policy)
- **Q-value** (reward from a state-action pair)
- **Epsilon** (tradeoff between taking a random or predicted action)
- **Discount Rate** (tradeoff between short or long-term reward)
- **Target Value** (the final Q-table)
- **Learning Rate** (how aggressively to apply new knowledge)

Q-table & Q-values

The Q-table is a simple matrix. The rows are **all** possible *states* (search space), and the columns are the *actions*. This forms a set of entries: state-action pairs, which are the *Q-values*. These will be initialized randomly, and updated as the algorithm approximate the solution.

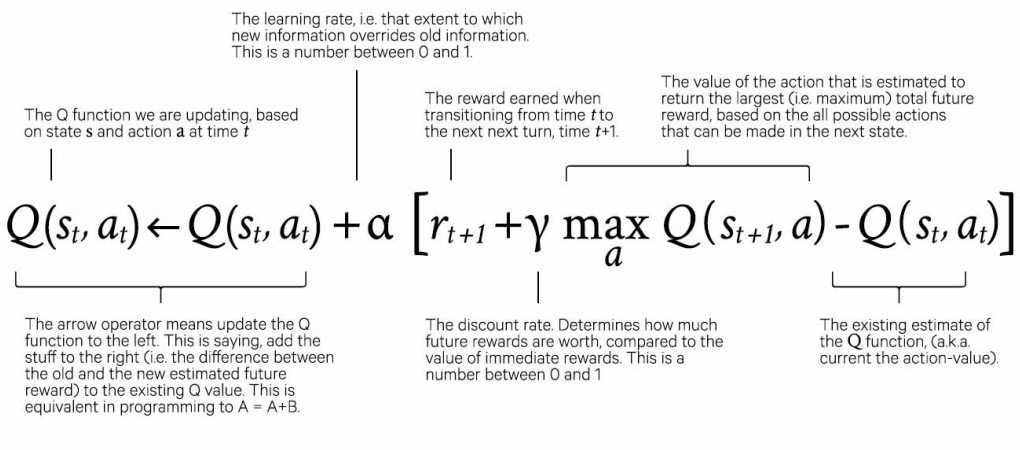


Figure 4.10: *Q-learning Algorithm:* Source: https://randomant.net/reinforcement-learning-concepts/

Epsilon

The *epsilon* factor makes it so that the agent don't predict actions as frequently in the very beginning. This is because in the start we want to value random decisions more, simply to *explore* the environment and find what actions lead to what rewards. Later, as the agent learns more of the policy, the epsilon will decrease in order to favor more the actual knowledge the agent has (make predictions) over random actions. This is what is know as the exploration vs. exploitation trade-off.

Discount Rate

The discount rate is just a scalar factor that is applied to each reward. Since the environment is unpredictable or non-deterministic, this value is added on each reward in an exponential fashion, as the rewards further into the future is more likely to diverge. So, the more into the future the reward is, the less we take it into account.

Target Value

The target value is the value we always want to achieve. This value is simply the best possible sequence of rewards we can get being in a given state.

> *"The maximum future reward for this state and action, is the immediate reward plus the maximum future reward for the next state. This is also called the Bellman Equation"* (youtube video: https://www.youtube.com/watch?v=79pmNdyxEGo)

Learning Rate

This factor is simply how fast we want the agent to apply the new knowledge it is learning. This is often referred to as the α alpha-learning rate, and in practice often is a constant of about 0.1 or 0.2. This means it will value mostly what it already knows, but reapply or override this a little bit.

4.5.3 OpenAI Gym: Frozen Lake

To familiarize ourselves, the OpenAI-gym library was used. OpenAI has an extensive library of a range of both discrete and continuous games available for developers to use. The environments are made purposely to apply ML techniques with a shared interface. One of the environments they have are very similar to the problem domain we are having.

FrozenLake Environment

Some of the ideas in this section were used by another github repository: https://github.com/the-computer-scientist/OpenAIGym/blob/master/QLearningIntro.ipynb

FrozenLake-v0 is a classical atari game where the problem is to simply move a node (in our case the snake head) on a grid towards a randomly selected location (where the apple is).

This is a very good isolated scenario that was used to get to understand the q-learning algorithm more in depth. The rules are straight forward:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

Figure 4.11: *The rules of the game.*

As briefly mentioned, in any gym environment there will be some very central variables that will be fetched once updating the environment:

```
# .step() returns the four most central variables
next_state, reward, done, info = env.step(action)
```

A basic algorithm were implemented with an Agent class. The agent gets initialized before starting the *simulation* loop of the system. The state also gets initialized as well as *done* and

steps. Then the main loop get started. Here the action will get fetched from the *get_action()* method given the state of the environment. We also get the four variables *next_state*, *reward*, *done*, *info* from the step() function from the gym API. Holding these variables, we now can train the *q-table* of the agent.

After the training some variables gets updated and some gets printed to the console, following with rendering the game-state which can be seen in Figure 4.12.

```
agent = QAgent(env)

total_reward = 0
for episode in range(10000):
    state = env.reset()
    done = False
    steps = 0
    while not done:
        action = agent.get_action(state)
        next_state, reward, done, info = env.step(action)
        agent.train((state, action, next_state, reward, done))

        state = next_state
        total_reward += reward

        print("state:", state, "action:", action, "episode:", episode, "steps:", \
            steps, "total_reward:", total_reward, "epsilon:", agent.epsilon)
        env.render()

        print(agent.q_table)

        time.sleep(.01)
        clear_output(wait=True)
        steps += 1
```

```
s: 3 a: 1 ep: 275 st: 7 tot_rew: 11.0 eps: 0.06304904523214558
  (Down)
SFFF
FHFH
FFFH
HFFG
```

Figure 4.12: *Running FrozenLake in the console. Here, the agent have not found the optimal policy yet and is deciding to go* ***down*** *and is unfortunately ending the game by going into a hole.*

The Q-learning Function

As has been mentioned already the goal or quality-value we want to approximate is the maximum possible future accumulated reward. The function can be described as this:

$$Q(s_t, a_t) = r_{t+1} + r_{t+2} + \ldots + r_{t+n} \tag{4.1}$$

Or, a better way to express what we want to achieve is to express it **recursively**. This time we also include the discount rate γ (gamma) which has already been covered. Further, we state that the next future reward is going to be the maximum of all potential four rewards possible.

$$Q(s_t, a_t) = r_{t+1} + \gamma max(Q(s', a')) \tag{4.2}$$

The number of possible future rewards that needs to be iterated over is: $action\_size^t$, making the algorithm exponentially increase the number of calculations. However since the board is not more than 4x4, this should not be any issue what so ever. The $action\_size$ is the number of all possible actions. In this case the agent can go four directions: *up*, *down*, *right*, *left*. Time t is the number of *time-steps* into the future.

The Agent Model

In order to train the agents *policy* we need to build a model to hold that policy, or since we don't include a neural network right now we just initialize a simple matrix to hold all states which again holds all actions. For each step we inspect the *q-table* (*policy* or *model*) as seen in Figure 4.13.

```
Observation Space:  Discrete(16)
Action Space:  Discrete(4)
Action size:  4
State Size: 16
state: 0 action: 0 episode: 0 steps: 0 total_reward: 0.0 epsilon: 1.0
  (Left)
SFFF
FHFH
FFFH
HFFG
[[2.63359069e-05 4.64107711e-05 7.30664274e-05 7.66017117e-05]
 [9.87206531e-05 5.06509205e-06 8.13210229e-05 3.43919137e-05]
 [7.04781978e-05 6.46484738e-05 4.83796193e-05 6.67231814e-06]
 [5.95522701e-05 6.40746210e-05 7.17872895e-05 5.18202844e-05]
 [2.08894764e-05 4.84431817e-05 1.79750346e-05 7.85696934e-05]
 [3.62501363e-05 5.64563787e-05 1.97852579e-05 3.70533281e-05]
 [5.72075652e-05 1.50190357e-05 5.31200464e-05 9.16307306e-05]
 [6.32569991e-05 6.39119135e-05 1.95639187e-05 2.35080753e-05]
 [6.39647379e-05 2.57867406e-05 7.84170135e-05 6.27378423e-06]
 [9.79242896e-05 7.19187256e-05 5.95502769e-05 2.99993294e-05]
 [7.76779227e-05 9.32508370e-05 6.67988662e-05 5.02094450e-05]
 [1.82543090e-05 7.61537841e-05 6.41641166e-05 5.61962354e-05]
 [8.99449126e-05 5.24815764e-05 6.60628045e-05 8.14757655e-05]
 [6.91717181e-05 7.28534478e-05 3.48918618e-05 3.67268254e-05]
 [9.30155069e-05 4.84882435e-05 6.50029096e-05 2.81792503e-06]
 [7.17063967e-05 5.41464239e-05 5.79978886e-05 9.00783403e-05]]
```

Figure 4.13: *Each **q-value** (cell) gets initialized randomly. Each item in the matrix is a state with each item in each state being an action (up, down, right, left).*

In order to build this simple model we use the numpy and random library in python:

```
1 def build_model(self):
2     self.q_table = 0.0001 * np.random.random([self.state_size, self.action_size])
```

Training The FrozenLake Agent

Our **hypothesis** is like mentioned in Figure 4.14: to expect the agent to find an optimal policy exponentially (most likely) after a certain number of time-steps, whiles also optimizing the number of steps, where lower is better.

```
reward: 0.0 state: 14 action: 1 episode: 14 steps: 12 total_reward: 1.0 epsilon: 0.8687458127689781
reward: 0.0 state: 14 action: 2 episode: 14 steps: 13 total_reward: 1.0 epsilon: 0.8687458127689781
reward: 0.0 state: 14 action: 2 episode: 14 steps: 14 total_reward: 1.0 epsilon: 0.8687458127689781
reward: 1.0 state: 15 action: 2 episode: 14 steps: 15 total_reward: 2.0 epsilon: 0.8600583546412883
Success!
reward: 0.0 state: 0 action: 0 episode: 15 steps: 0 total_reward: 2.0 epsilon: 0.8600583546412883
reward: 0.0 state: 1 action: 1 episode: 15 steps: 1 total_reward: 2.0 epsilon: 0.8600583546412883
reward: 0.0 state: 5 action: 2 episode: 15 steps: 2 total_reward: 2.0 epsilon: 0.8514577710948754
Fell into a hole.
```

Figure 4.14: *Here the agent arrives to the **goal-state** (G) after 15 steps. After finding a better policy the number of steps will hopefully decrease, as the absolute shortest possible path should only be six moves.*

Now that we have stated the hypothesis, we obviously need to test it ans observe if this assumption indeed is correct or not. In order to do that and test our hypothesis we need to collect the data from the q-learning simulation that has been built. Like mentioned we use numpy to save the data in *numpy-arrays*, and then use matplotlib to plot our data and visually represent it in a graph.

So, to test this we turn up the number of *episodes* to 300. This is a very high number and should give us at least an indication of what is going on.

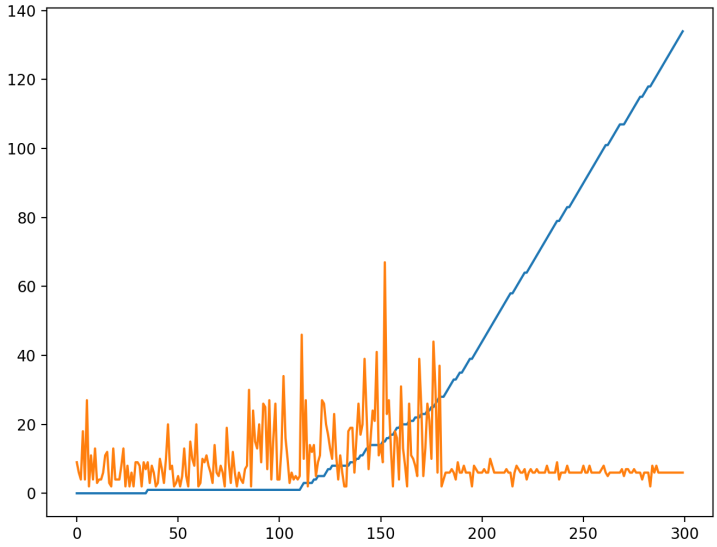


Figure 4.15: *[Blue: reward] [Orange: steps]. Observing the graph one can easily see that the policy gets optimised and the number of steps gets lowered to six moves eventually.*

Analyzing the data one can observe that the optimal policy is found after roughly 180 steps. After this time most of the runs had a positive reward, hence arrived in the goal state. Some of the runs after this point went into a whole, but the majority did not, which is a good

achievement. The number of steps had a high variance, but eventually got minimised to six moves, which we initially were hoping for.

4.5.4 Deep Q With A Neural Network

Some of the ideas used in were built on this repository: https://github.com/maurock/snake-ga

Making the same system, but this time letting the agent hold a neural network model - and not a q-matrix-table like we initially did, we import TensorFlow into our system and reconstruct the build_model() method.

```
1  def build_model(self):
2          tf.reset_default_graph()
3          self.state_in = tf.placeholder(tf.int32, shape=[1])
4          self.action_in = tf.placeholder(tf.int32, shape=[1])
5          self.target_in = tf.placeholder(tf.float32, shape=[1])
6
7          self.state = tf.one_hot(self.state_in, depth=self.state_size)
8          self.action = tf.one_hot(self.action_in, depth=self.action_size)
9
10         self.q_state = tf.layers.dense(self.state, units=self.action_size, name='q_table')
11         self.q_action = tf.reduce_sum(tf.multiply(self.q_state, self.action), axis=1)
12
13         self.loss = tf.reduce_sum(tf.square(self.target_in - self.q_action))
14         self.optimizer = tf.train.AdamOptimizer(self.learning_rate).minimize(self.loss)
```

On lines 3-5 a placeholder is initialized to hold the state-input, action-input and target-input to be fed to the network. As seen when inspecting the placeholders object they essentially are what TensorFlow calls a *Tensor*: `<tf.Tensor 'Placeholder:0' shape=(1,) dtype=int32>` object, which is what it requires to be fed. These all have a shape of a 1D-vector or array.

OneHot Encoded

However in TensorFlow giving networks input, they need to be of a particular type which is called *onehot*. This is an encoding that can be easily converted with the built in one_hot() function provided. The *depth* of the vector is now going to be the same as the whatever the *state_size* and *action_size* is. So right before they will go through the network they will look like something shown in Figure 4.16.

On line 10 in the build_model() method, a dense hidden layer is declared which is between the input and output layer. The dense layer means that all nodes in this layer are fully connected

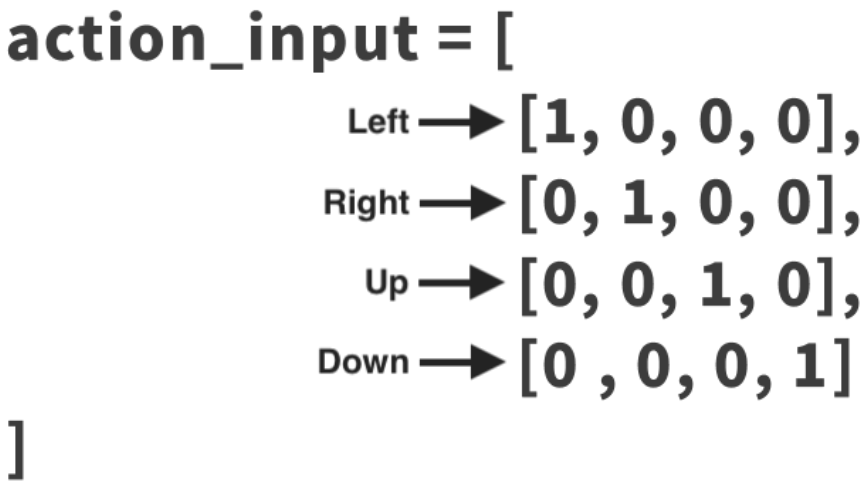


Figure 4.16: *Onehot encoded Tensor-Placeholder of the action input data. The Placeholder is a Tensor object that is being used as a handle to later act as a pipeline or opening to the network.*

with all other nodes in the network. The number of unints or nodes are set to four, or the same as the *action_size*. The number of nodes can be explored more, but for now this is a fine number to test the model with.

On line 13 a loss variable is declared. This is simply trying to reduce the sum between the prediction and the actual target. Then finally an optimizer is initialized using the AdamOptimizer.

In Figure 4.17 we can see that the new model is performing a lot better.

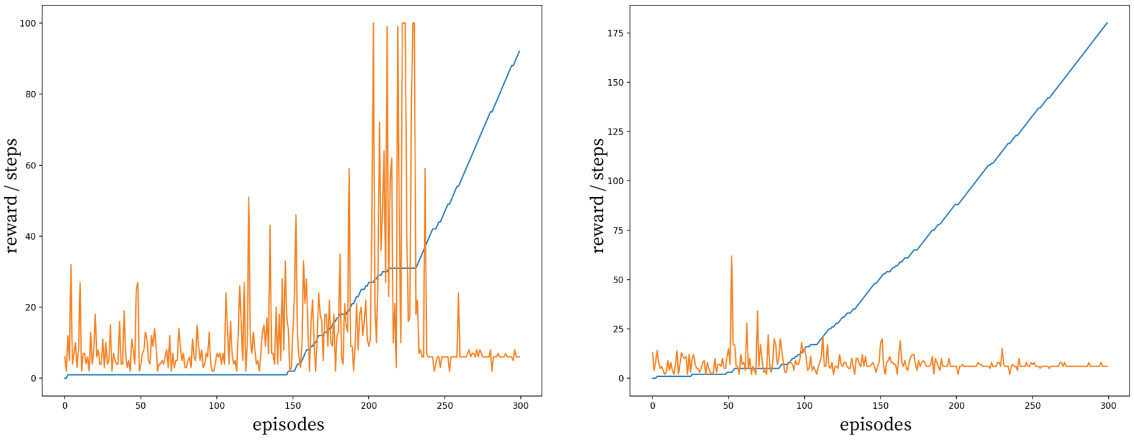


Figure 4.17: *[Blue: reward] [Orange: steps]. Observing the new graph one can easily see that the policy gets optimised a lot quicker this time. The number of steps gets lowered to six moves roughly after 200 moves compared to 250 moves for the original one. A good and optimal path or policy is found arguable just after 100 steps.*

The new model seem to be a lot more efficiant as it is able to find an optimal policy a lot qucker. This model will not only change each cell only once per step, but multiple cells, hence this is why it is a lot quicker, which is taken care of by TensorFlow.

Conclusion FrozenLake

This experiment let us build first a q-learning algorithm, a q-table model and an agent to test some concepts in a smaller isolated environment. We then introduced a neural network model to act as the weights of the q-table. What have been learnt will be applied to the snake environment so that the system can be implemented in a similar fashion as the OpenAI gym environments and the models that have been constructed.

4.6 Snake Agent

So, back to the snake environment, we implement the agent with all the parameters needed as already mentioned previously. The variables that the agent should hold is *current_reward*, *gamma*, *short_memory*, *agent_target*, *agent_predict*, *learning_rate*, *epsilon*, *memory*, *random_move*, *random_moves*, *model*. Also the methods that needs to be implemented are: *neural_network()*, *get_action()*, *train_short_memory()* and *replay_new()*.

```
def neural_network(self, weights=None):
        model = Sequential()
        model.add(Dense(activation="relu", input_dim=NUM_OF_INPUTS, units=120))
        model.add(Dropout(0.15))
        model.add(Dense(activation="relu", units=120))
        model.add(Dropout(0.15))
        model.add(Dense(activation="relu", units=120))
        model.add(Dropout(0.15))
        model.add(Dense(activation="softmax", units=3))
        model.compile(loss='mse', optimizer=Adam(self.learning_rate))

        if weights:
            model.load_weights(weights)

        return model
```

```
def get_action(self, prev_state, prev_direction, possible_actions, game_counter):
        """returns a random or predicted action (forward, right, left)"""
        if random.uniform(0, 1) < self.epsilon:
            current_direction = possible_actions[randint(0, 2)]
            self.random_move = True
            self.random_moves += 1
        else:
            prediction = self.model.predict(prev_state.reshape((1, NUM_OF_INPUTS))) # reshape:
            current_direction = possible_actions[np.argmax(prediction[0])]
            self.random_move = False
        return current_direction # aka action

    def train_short_memory(self, prev_state, action, reward, current_state, game_over):
```

```
        target = reward
        if not game_over:
            target = reward + GAMMA * np.amax(self.model.predict(current_state.reshape((1, NUM_
        target_future = self.model.predict(prev_state.reshape((1, NUM_OF_INPUTS)))
        target_future[0][np.argmax(action)] = target
        self.model.fit(prev_state.reshape((1, NUM_OF_INPUTS)), target_future, epochs=1, verbose

    def replay_new(self, memory):

        if len(memory) > STEPS_INTO_FUTURE:
            minibatch = random.sample(memory, STEPS_INTO_FUTURE)
        else:
            minibatch = memory

        for state, action, reward, current_state, game_over in minibatch:
            target = reward
            if not game_over:
                target = reward + GAMMA * np.amax(self.model.predict(np.array([current_state])))
            target_future = self.model.predict(np.array([state]))
            target_future[0][np.argmax(action)] = target
            self.model.fit(np.array([state]), target_future, epochs=1, verbose=0)
```

Chapter 5

Testing & Evaluation

5.1 Testing

Testing the system or simulation is profoundly important, especially if the scale of the system will increase in the future. Ideally setting up a build pipeline would be of interest and is included in the MoSCoW table, however this must be done in the future because of limited time. The build pipeline could be set up so that whenever we want to deploy the system, a script is activated. This script will run our simulation with our predefined tests, and depending on if the tests passed or not it gets saved and deployed to our platform of choice, the web, or other servers.

5.1.1 Test Driven Development

Test Driven Development (TDD) were investigated both in the beginning, but also later in the project. The idea – as seen Figure 5.1 – of test driven development is to mock up a carefully created, specific and narrow single test, and then afterwards implement that feature or method in the system. Finally one will test that feature to make sure it passes before continuing with implementing the next test and feature pair. The system will not be continued before absolute all tests are passed, to make sure the system is as strong and reliable as possible.

In the book from (Beck, 2003) Beck sums up the iterative steps or cycle to follow a test driven development. The steps that are stated are:

- Add a little test
- Run all tests and fail
- Make a little change

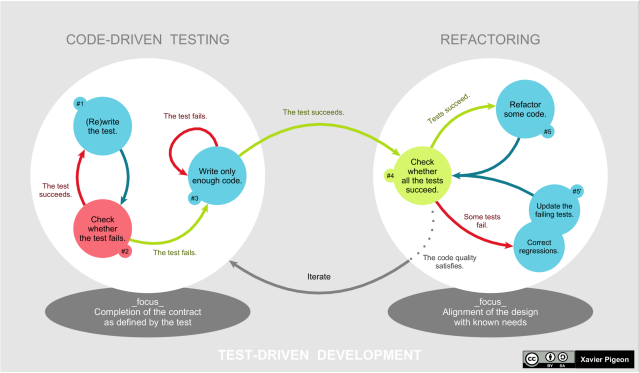


Figure 5.1: *TDD.*

- Run the tests and succeed
- Refactor to remove duplication

5.1.2 Unit Testing

To make unit tests in Python, a new separate file needs to be created in the project folder. We call this *test_methods.py*. In this file we need to import the module *unittest*, before any tests can be written.

Normally the tests needs to be grouped inside classes, which will inherit from the *unittest.TestCase*. These are made so that any developer can make test cases or scenarios relating to their system. Many methods were tested from the system. And one of them were to calculate the destination on a grid given a direction of the snake.

Calculate Destination

To test the calculate-destination method in the system, we need to explain how it works first. A *diff_xy* is initialized to hold the difference in both x and y as a vector [x, y]. To find this the direction gets mapped to a static MAP dictionary. This in turn, returns four possible directions as a *vector*, in this case (up) it returns [-1, 0]. The corresponding values will ten be summed to find the absolute destination on the grid – depending on where the snake is – with help of some syntactic sugar and the built in *zip()* method on line 5.

```
1 def calc_destination(self, direction):
2     """add diff_xy to snake_head"""
3     diff_xy = self.MAP[direction]
4     snake_head = self.snake_body[0]
5     destination = [a_i + b_i for a_i, b_i in zip(diff_xy, snake_head)]
6     return destination
```

On line 1 we define the test method: *test_calc_destination_up()*. Then on line 2-3 the snakes head is set to be [2, 2]. We then run the actual method *calc_destination()*, pass it the direction *up*, and finally (line 5) we assert that the return of the method or the destination is indeed [1, 2], where the first number represent the vertical x-axis and the second number represents the horizontal y-axis. Essentially this verifies that the snake have moved up on cell on the grid. This was tested for all directions possible to cover all possible scenarios.

```
1 def test_calc_destination_up(self):
2     game.snake.snake_body = [[2, 2]]
3     snake_head = game.snake.snake_body[0]
4     destination = game.snake.calc_destination('up')
5     self.assertEqual(destination, [1, 2])
```

Testing the *get_close_sight()* were also done. This method return the immediate neighbours. This is to check for danger in the system. The method is as follows:

```
1  def get_close_sight(self, direction):
2      '''returns the immediate neighbours'''
3      neighbours = []
4      if direction == 'up':
5          neighbours.append(self.calc_destination('up')) # forward
6          neighbours.append(self.calc_destination('right')) # right
7          neighbours.append(self.calc_destination('left')) # left
8      elif direction == 'down':
9          neighbours.append(self.calc_destination('down')) # forward
10         neighbours.append(self.calc_destination('left')) # right!
11         neighbours.append(self.calc_destination('right')) # left!
12     elif direction == 'right':
13         neighbours.append(self.calc_destination('right')) # forward
14         neighbours.append(self.calc_destination('down')) # right
15         neighbours.append(self.calc_destination('up')) # left
16     elif direction == 'left':
17         neighbours.append(self.calc_destination('left')) # forward
18         neighbours.append(self.calc_destination('up')) # right
19         neighbours.append(self.calc_destination('down')) # left
```

A combination of both absolute and relative directions in the systme can be confusing, but these functions are carefully designed to deal with that, always both printing the relative and absolute direction to the console so the user can know what is going on. This is also well documented in the code.

This is using the previously tested *calc_destination()* method, to then be appended on a list of neighbours depending on whet absolute direction the snake is currently traveling in. Again, the test is covering all possible directions, but this example is the case of moving towards the *left*:

```
def test_get_close_sight_left(self):
    neighbours = game.snake.get_close_sight('left')
    self.assertEqual(
        neighbours, [
            game.snake.calc_destination('left'),
            game.snake.calc_destination('up'),
            game.snake.calc_destination('down')
        ]
    )
```

Again, all tests passed as seen in Figure 5.2.

```
(env)
josdyr at JosBookPro in Snake_Game on master [!?$]
$ python -m unittest -v test_methods.py
Using TensorFlow backend.
test_calc_destination_down (test_methods.TestMethods) ... ok
test_calc_destination_left (test_methods.TestMethods) ... ok
test_calc_destination_right (test_methods.TestMethods) ... ok
test_calc_destination_up (test_methods.TestMethods) ... ok
test_get_close_sight_down (test_methods.TestMethods) ... ok
test_get_close_sight_left (test_methods.TestMethods) ... ok
test_get_close_sight_right (test_methods.TestMethods) ... ok
test_get_close_sight_up (test_methods.TestMethods) ... ok

----------------------------------------------------------------------
Ran 8 tests in 0.001s

OK
```

Figure 5.2: *The test script runs and outputs an OK, signalling all 8 tests were passed correctly.*

We can conclude that the system after testing it, is indeed a lot more reliable, hence we can trust that the system should work as intended when in the future we want to release it as production code. Later, when the system will be furthered developed and the developer(s) are refactoring the system, they can exclude possible beliefs of why the system should not happen to work. There were some parts of the system that could not be tested fully, because of a limiting time-frame. Also, techniques like test driven development and making a build pipeline would have been more strictly followed and further investigated in the future to be able to fully trust and rely on the system.

5.2 Evaluation

After training the snake for a good number of times (est: 30min), the snake became fairly good. The snake first seemed to learn how to avoid the walls, hence being able to play or be inside the current game for a lot longer. It basically learned that whenever there is a wall, obstacle or danger it should try to go into another direction which don't have any danger. This is obviously very basic for humans, but for the snake's neural network it needs to fully learn this from absolute scratch. Also, recall that the snake can *only* see part of the world at

all times, unlike any human playing the game which they can perceive the whole world. All things considered, this indeed resulted in a great achievement, and probably the most central and most difficult achievement of them all so far.

As seen in Figure 5.3 we run the simulation for 100 episodes and the grid size to 5x5, we can see that only after 12 runs the agent eats 3 apples and after 67 runs it eats 8.

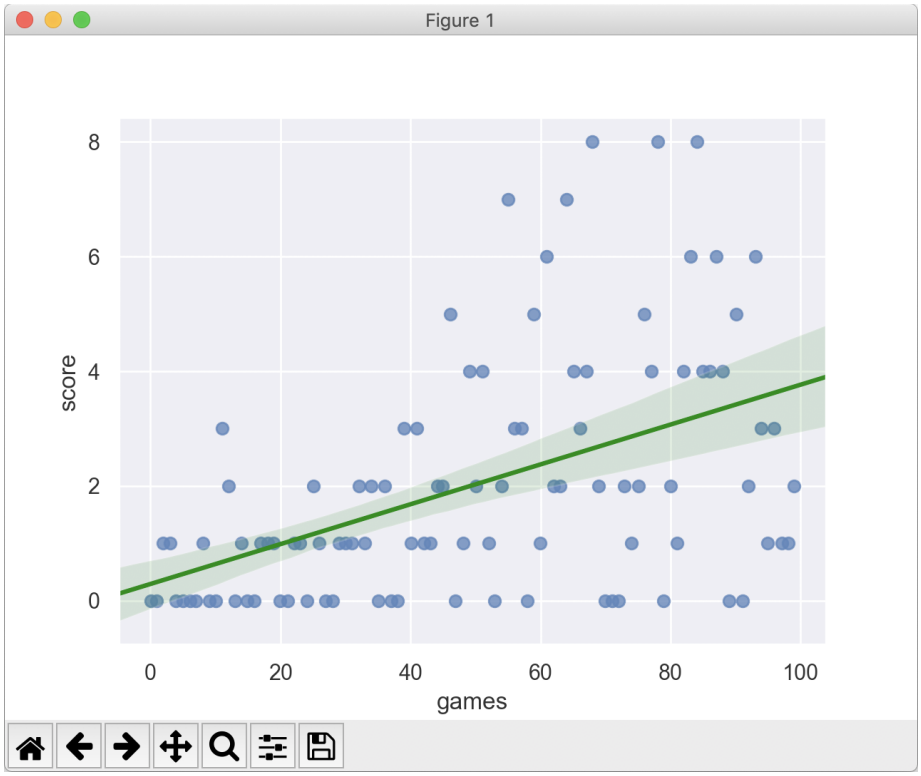


Figure 5.3: *Result*

We also apply a *linear regression* line (green line) on the diagram to visually represent the learning rate and get the *slope* or learning speed of the system. This slope can be used later to compare different runs between each other to compare the speed of them, adjusting things like the neural network and hyper-parameters.

5.2.1 Observing the Snakes Behaviour

Later as the snake is able to stay inside each game longer, it allows him to explore more and learn what its ultimate objective is: to go towards the apples. It seems to learn this, however it is not perfect. Most times it is indeed going towards the apple, but then suddenly it *could* decide to go a complete different direction *or* even decide to just go towards the wall (for fun / for exploration). This is happening because the algorithm is obviously training the snake's neural network, and we have not yet made a testing session for the snake to be evaluated. So the reason it is all of a sudden changing direction or deciding to go into a wall is because the algorithm is sometimes choosing a *complete random* direction which has

nothing to do with the neural network, hence we can not blame the neural network for this mistake. Acknowledging this, we need to change something. But what?

There are multiple directions and ways to go from here. As we have concluded, the algorithm is making it so that the snake fails and ends the game - which we obviously want to avoid. However we want it to *fail* during testing so that it can learn the *rule* of what makes him fail, so it can avoid that during real life testing - in the final weights of the model.

Having this in mind, we can conclude that there is not really any fault in the system, however we need to test the snake's performance. We print out the *cause of death* over the *game count*, to inspect how well it is doing.

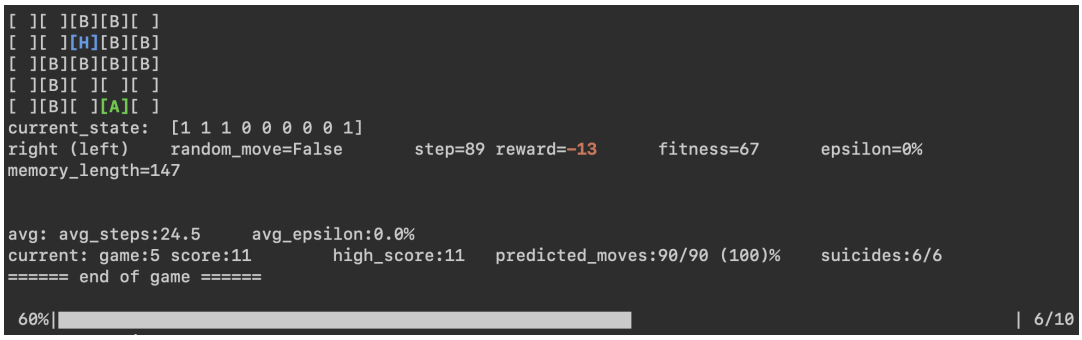


Figure 5.4: *The snake is predicting and trying intentionally to go to its **left** (relative) our **right** (absolute), however that leads to a **suicide**, which is displayed in the lower right corner. In 6 games out of 6, the snake died because of a **predicted move** (random_move=False), hence taking a **suicide**. The conclusion is that the neural network is not **perfect** in any way unfortunately.*

We could say that since the snake intentionally decides to take suicide or die 6 times out of 6, the neural network is not in well enough trained yet - or maybe it is somehow overfitted. This needs to be further investigated.

The Snake Loves Apples

Looking closer at how the snake behaves, we can immediately see that the snake likes apples - a lot. Even when the apple is somewhere in front of him - but his tail is in between - the snake still tries to go toward the apple, but obviously dies by hitting his own tail.

This could be because the snake have not *learned* properly yet how to avoid this scenario. The snake was trained on a relative small board-size 5 by 5. But when increasing the board size it is clear that the snake do not know how to behave properly. Knowing this, we try to use *curriculum learning* which is previously discussed, to increase the board size slowly, so that the agent learns first how to handle small isolated environments, and then we increase the size to also train it on a bigger field.

Maybe the snake should not be able to see the food from this angle, however this could be

Figure 5.5: *Here we can see that even though there is a danger (tail) to its front (our right), the snake will insist on going forward to try to catch the apple. The current state of the game here is: [1, 0, 0, 1, 0, 0] which show that there is a danger in front of him, however there is an apple to its front as well, so the snake chooses to try to get that apple.*

improved upon in the future.

Spirals

Another behaviour that was found in the snake, was the tendency to circle in big loops around the grid – and then when it saw an apple it immediately goes in, eats it and goes back to the outer loop – adopting arguably a very good strategy. Recall that there is no such thing as trying to avoid number of steps. Doing many *steps* could be argued as a very good thing, knowing that the snake knows exactly how to avoid the walls. Later, there will be introduced a micro-penalty for each step.

However, sometimes the snake also tended to go into a sort of a spiral. This was noted, and funnily enough this is what snakes sometimes do when they are threatened or in other stress related situations.

Exploring The Grid

When inspecting the moves of the snake, another interesting behavior were found. Sometimes, the snake started spinning into circles for a good while. Eve though we have a max step-count terminator of 500, the snake found it intriguing to spin for long periods, before continuing. This was evaluated, and another feature were added to the system in hope of an increase in exploration.

A new hyper-parameter were set: *anti_exploration_count* for lack of a better name. This is set to 10 by default. This parameter's main idea is to avoid making the snake re-visit the 10

most recently visited cells in the grid of the board. If however the snake does a re-visit, it will get a small penalty of -3 to avoid this kind of policy. After implementing this feature, the snake did indeed avoid going into circles and explored the board a lot more.

We also print and color code the step, reward and fitness values to get a stronger impression and understanding of the system. We also calculate the slope of the learning curve to inspect the *learning-rate*.

5.3 Neural Network Exploration

An initial plan was set up to try to find a good structure for the neural network and to try to compare them to evaluate and understand what works best, now that we have a working system which can give us average fitness and a learning rate.

5.3.1 Number of Units

Although we are not going to try to approximate the absolute optimal neural network model, some investigation and effort were made to find out what some configurations were ideal. Recall that the NEAT Library would be excellent to use here although this was unfortunately decided to not be used in this project.

The number of *units* (neurons/nodes) being one of them. However, when considering how many hyper-parameters there are in the system, testing all of them, finding correlations and compare between them becomes unbelievably challenging and more so time consuming. This is where the NEAT Library comes in, although this was decided to not utilised as discussed earlier.

Four simulation objects were run and created each having 50 episodes/games to cover a decent and reliable result. The results were interesting, and even more so difficult to understand. The number of units that were firstly tested were 30, 60, 90 and 120. The first test indicated that the latter one – 120 units – were the best one of them. However another test were run too, to test it again for reliability, and this time it was decided to up the range to 60, 90, 120 and 150 to check it it goes even higher.

Will the learning rate and fitness increase even more if we add some more units? We utilised the remote server as seen in Figure 5.7, and disabled all form of sleeping and printing. All these things considered, seeded up the testing process a lot.

The snake does not seem to develop an underlying strategy for how to play the game. It

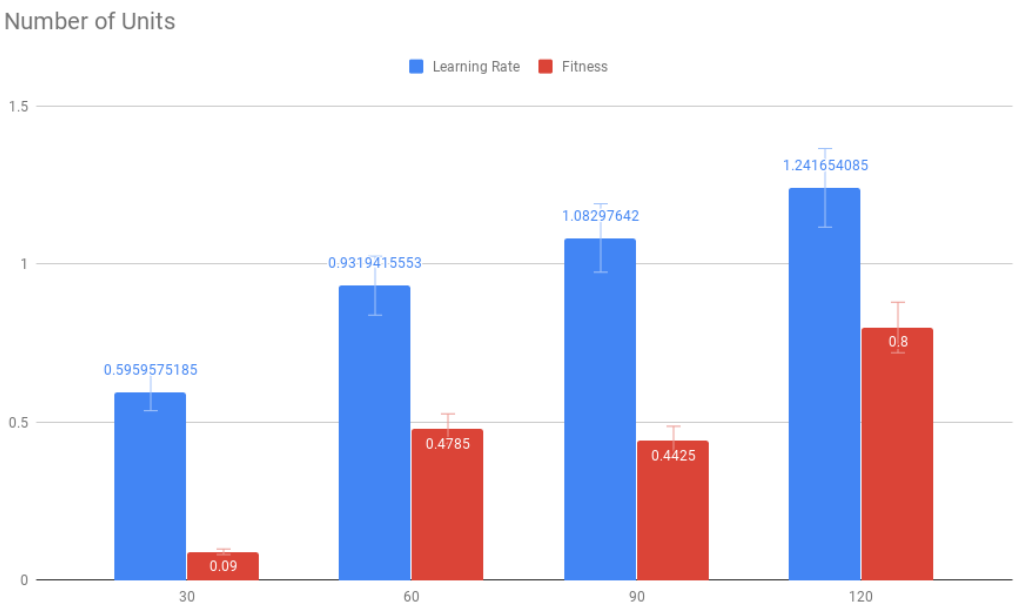


Figure 5.6: *The number of units have some level of correlation. The best one being 120 units. The overall learning rate and fitness is better. Will it increase even more?*

Number of Units					
Run	**Units**	**Board Size**	**Learning Rate**	**Intercept**	**Fitness**
1	30	6	0.59595751	-0.6176014	0.15
2	60	6	0.93194155	-0.4215668	47.85
3	90	6	1.08297641	-0.6286035	44.25
4	120	6	1.2416540	-0.7285167	80.0

Table 5.1: Caption2

seems to be very reactive to what he perceives and not. The overall time spent to find a good policy is arguably lower quicker than expected. The number of nodes that were found were roughly 120 nodes, although no good explanation were found for why this is exactly.

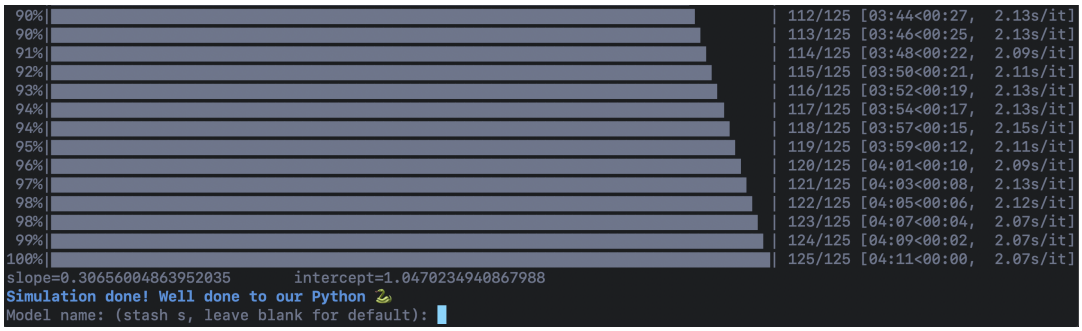


Figure 5.7: *Running the tests in print-free version on our configured AWS server.*

Chapter 6

Conclusion & Future Work

As briefly mentioned, the agent learns the policy surprisingly quick, sometimes covering nearly all cells. Interesting patterns were found like circling in big rings and spirals. It loves food even when his tail is stopping him. The snake is good with exploring the grid evenly as there is a penalty for revisiting cells. Visually printing the data for each simulation-object helped immensely with evaluating the system.

6.1 Future Work

Some technologies and methods would be looked further into if more time were present. NEAT the generic algorithm library to optimise neural networks is probably the foremost.

Future work like 3D Graphics to the game, GPU/TPU compatibility, use of the Curces-library as well as Tensorboard and Multi-Agent System Implementation. Also distributed systems where we could demonstrate server to server communication via the interface, as well as ouptting to multiple consoles and log files. This were initially planned, but again time was limiting.

Convolutional Neural Network (CNN) is another one. Making the agent function similarly, by seeing a 3D pixel image. The three dimensions are the horizontal pixels, vertical pixels and the RGB color dimension. This is so that we can approach a general method to play any game achieving similar results like What Googles' researchers did like mentioned.

Other tests that would have been interesting to conduct, given more time are:

- Change Number of Layers in the Network
- Change activation function

Must	Should	Could	Won't/Future
Game	Unit tests	Graphical Interface (2D)	3D Game Interface
Simulation	Find Best Game	Build Pipeline	CNN
Agent	Replay	Ensemble ANN	3D Interface
Artificial Neural Network	Analysis	TDD	GPU/TPU Compatibility
Algorithm	Manual Game Play	Step-Functionality	Multiple console output
Console Interface			Multi-Agent System
API			Interactive (Curses library)
			Tensorboard
			NEAT

Figure 6.1: *MosCow Table*

- Change reward system or the fitness function
- Change the environment during execution (curriculum learning)
- Give a tiny reward per step

Some concepts and ideas were investigated and implemented in an immense and extensive fashion, where the documentation might not reflect all this as accurately as I wanted. However, I am basically very happy with the result of the project itself and the results that have been given.

References

.

URL: https://gym.openai.com/docs/.

Artificial Intelligence Oxford Definition. URL: http://www.oxfordreference.com/view/10.1093/oi/authority.20110803095426960.

Beck, Kent (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Beck, Kent et al. (2001). "Manifesto for agile software development". In:

Bengio, Yoshua et al. (2009). "Curriculum learning". In: *Proceedings of the 26th annual international conference on machine learning*. ACM, pp. 41–48.

Bossmann, J. (2016a). *Basics of the Unix Philosophy*. URL: http://www.catb.org/~esr/writings/taoup/html/ch01s06.html.

— (2016b). *Top 9 ethical issues in artificial intelligence*. URL: https://www.weforum.org/agenda/2016/10/top-10-ethical-issues-in-artificial-intelligence/.

Charniak, Eugene and Drew McDermott (1985). *Introduction to Artificial Intelligence*. Tech. rep. Addison-Wesley.

Copeland, B. (2018). *Artificial Intelligence*. URL: https://www.britannica.com/technology/artificial-intelligence.

Descartes, René (2018). *Discourse on the Method*. Perennial Press.

Harari, Yuval Noah (2016). *Homo Deus: A Brief History of Tomorrow*. Random House.

Juliani, Arthur (2017). *Introducing: Unity Machine Learning Agents Toolkit*.

Kinsley, Harrison (2016). *Introduction to Deep Learning with TensorFlow*. URL: https://pythonprogramming.net/tensorflow-introduction-machine-learning-tutorial/.

Life Institute, Future of. *Benefits And Risks of Artificial Intelligence*. URL: https://futureoflife.org/background/benefits-risks-of-artificial-intelligence/?cn-reloaded=1.

Linares-Rodriguez, Alvaro et al. (2013). "An artificial neural network ensemble model for estimating global solar radiation from Meteosat satellite images". In: *Energy* 61, pp. 636

–645. URL: http://www.sciencedirect.com/science/article/pii/S0360544213007597.

McIlroy, M. D. (1978). "UNIX Time-Sharing System: Foreword". In: *Bell System Technical Journal* 57.6, pp. 1899–1904. DOI: 10.1002/j.1538-7305.1978.tb02135.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.1538-7305.1978.tb02135.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/j.1538-7305.1978.tb02135.x.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, p. 529.

O'Malley, J. *The 10 most important breakthroughs in Artificial Intelligence.* URL: https://www.techradar.com/news/the-10-most-important-breakthroughs-in-artificial-intelligence.

Poole (1998). *Computational intelligence: a logical approach*.

Russell, S. N. (2018). *Artificial Intelligence: A modern approach.*

Sharkey, A. J. (1999). "Combining Artificial Neural Nets: Ensemble and Modular Multi-Net Systems". In: *Springer*.

Skymind (2018). *A Beginner's Guideto Deep Reinforcement Learning*. URL: https://skymind.ai/wiki/deep-reinforcement-learning.

Winston, P. H. (1992). *Artificial Intelligence (Third edition)*.

Woodford, Grant W and Mathys C du Plessis (2018). "Robotic snake simulation using ensembles of artificial neural networks in evolutionary robotics". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM, pp. 173–180.

Appendix

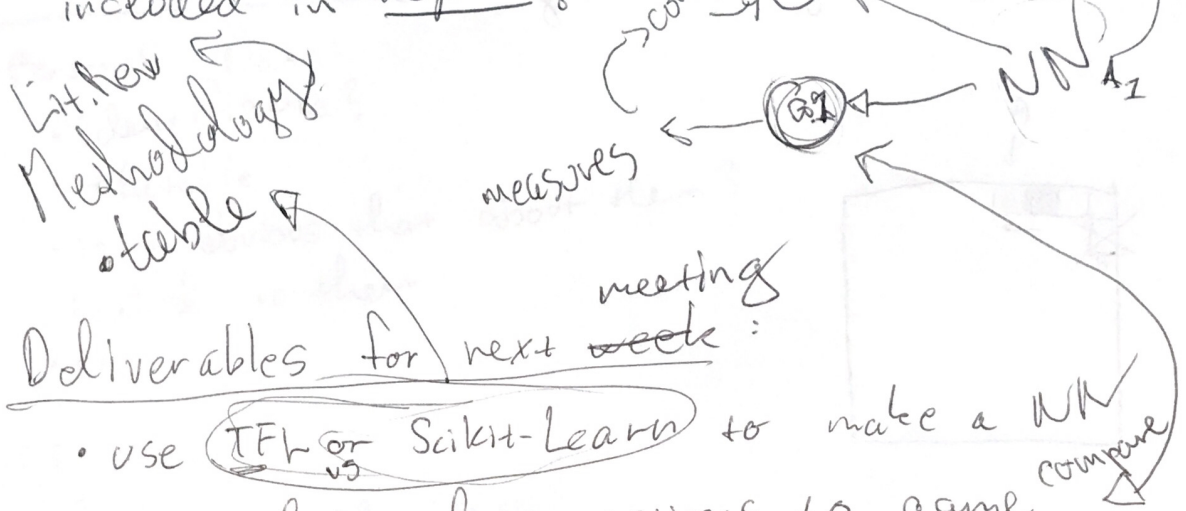
Source Code: https://github.com/josdyr/Snake_Game

#	MILESTONES & TASKS	START DATE	DUE DATE	DURATION	COMPLETE
1	Project Conception and Initiation				
1.1	Project Planning	9/3/18	9/19/18	16	100%
1.2	Basic Research	9/7/18	9/23/18	16	100%
1.3	Project Deliverables	9/13/18	9/18/18	5	100%
1.4	Write Initiation Project Overview	9/13/18	9/26/18	13	100%
2	Project Definition and Planning				
2.1	Goal Setting	9/24/18	10/3/18	9	100%
2.2	Scope & Boundaries	10/1/18	10/13/18	12	100%
2.3	Constraints	10/1/18	10/13/18	12	100%
2.4	Risk & Project Control	10/7/18	10/21/18	14	100%
3	Research Phase				
3.1	Previous & Current Research	10/18/18	10/25/18	7	100%
3.2	Neural Networks	10/21/18	11/9/18	18	100%
3.3	Ensembles	10/21/18	11/14/18	23	100%
3.4	Technologies & Tools	10/26/18	11/26/18	30	100%
3.5	Draft Literature Review	10/28/18	11/30/18	32	100%
4	Game Development				
4.1	Investigate similar games	11/4/18	11/15/18	11	100%
4.2	Investigate game design	11/12/18	11/24/18	12	100%
4.3	Make Class Diagram	11/14/18	11/21/18	7	100%
4.4	Make Design	11/22/18	11/30/18	8	100%
4.5	Implement Game	11/26/18	1/21/19	55	100%
4.7	Game Unit Testing	1/16/19	1/29/19	13	100%
5	Neural Network				
5.1	Agent Implementation	1/10/19	1/25/19	15	100%
5.2	Neural Network Implementation	1/17/19	2/6/19	19	100%
5.3	Testing	2/3/19	2/12/19	9	100%
6	Analysis & Evaluation				
6.1	Implementing Evaluation Scripts	2/12/19	2/24/19	12	100%
6.2	Running Evaluation	2/13/19	3/13/19	30	100%
6.3	Document Outcomes	2/25/19	3/13/19	18	100%
7	Documentation & Report				
7.1	Inline Documentation	3/13/19	3/15/19	2	100%
7.2	Report	3/13/19	3/29/19	16	100%

Figure 2: *Gantt Chart*

What I have done this week

- focusing on integrating my own Snake game
  - full control over game and API
- also noted down some SO-concepts to be included in report!

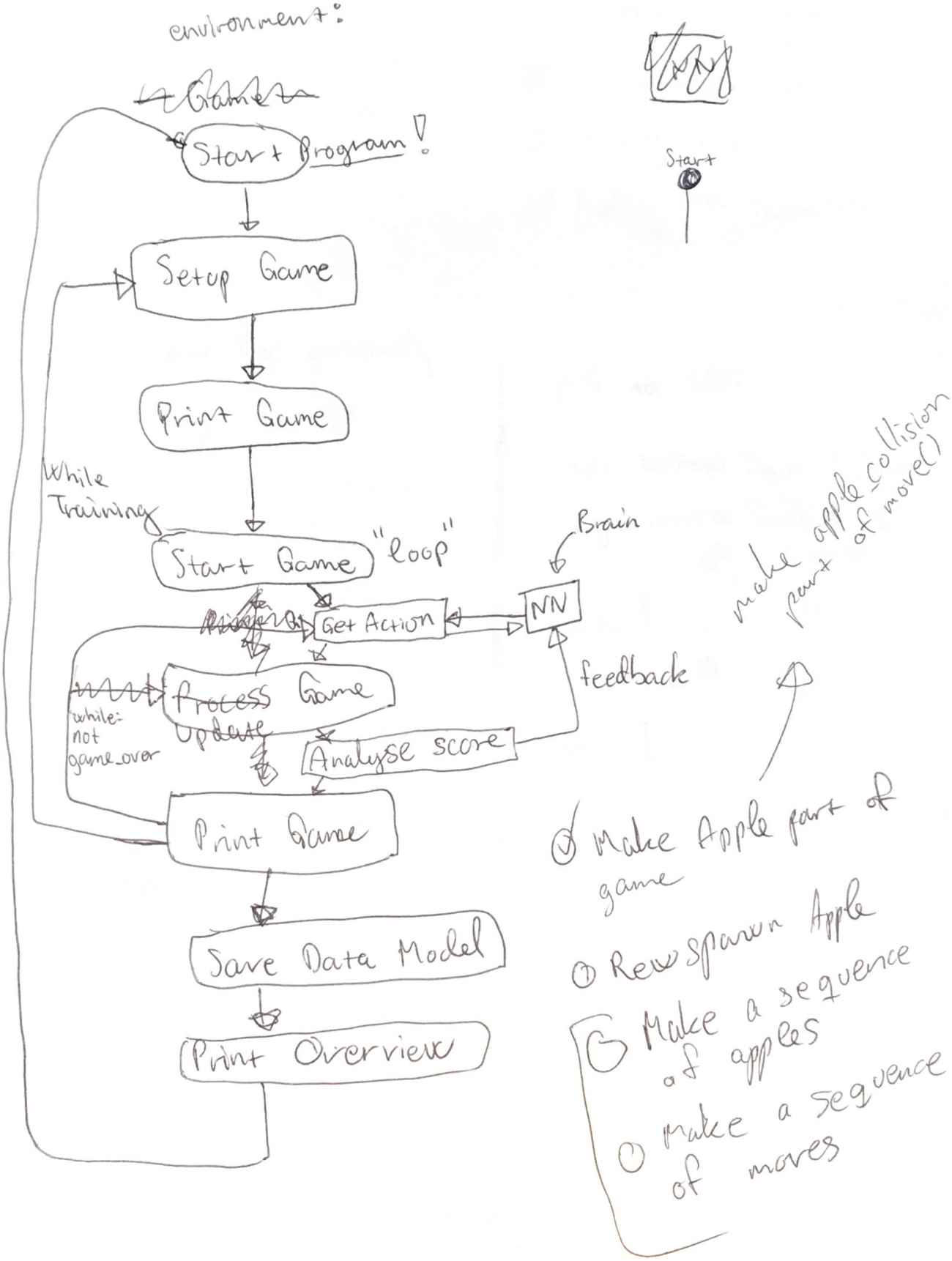


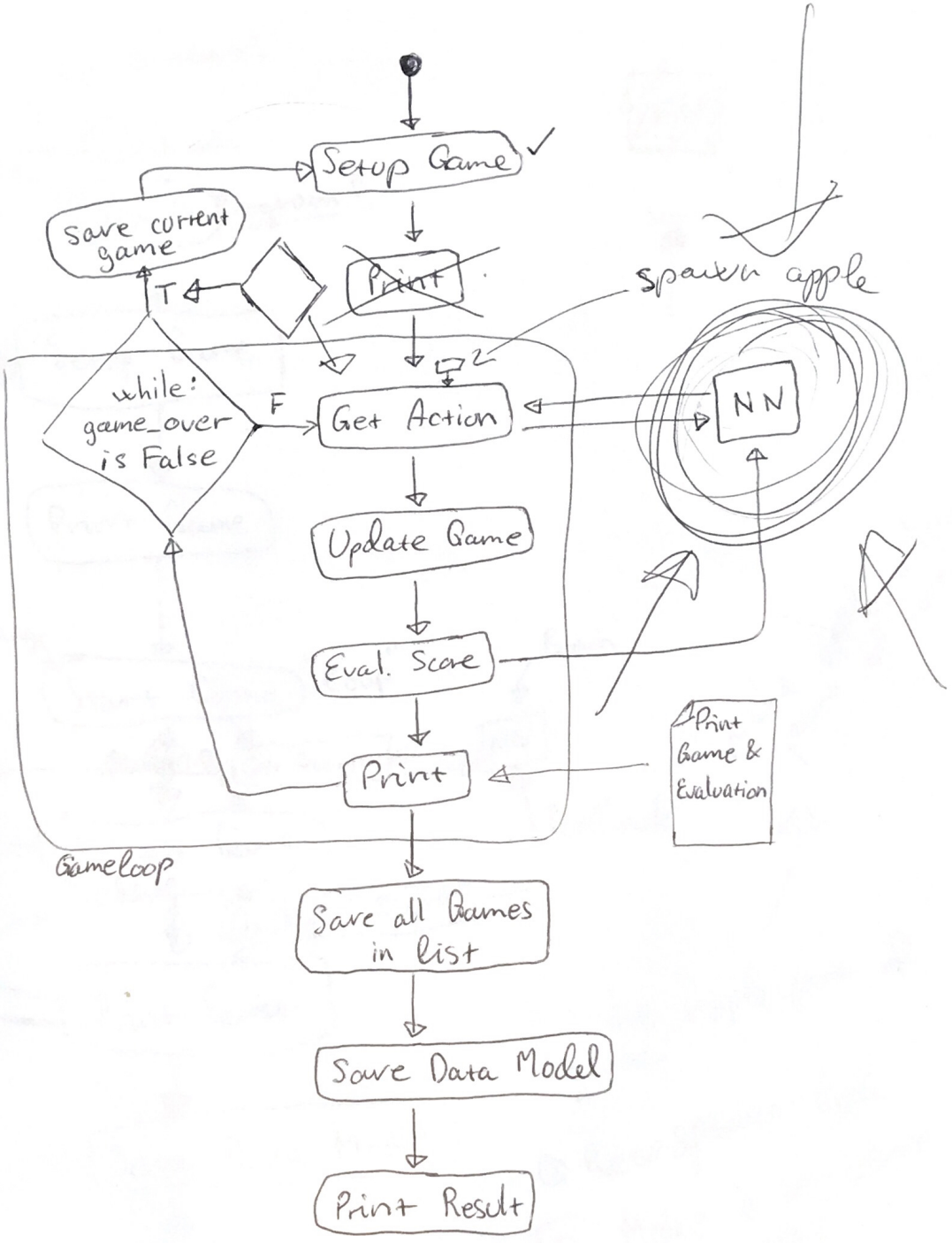
Deliverables for next ~~week~~ meeting:

- use TFL or vs Scikit-Learn to make a NN
- send random actions to game
- close the "loop" -> provide first prototype!
- fill in some sections in report
- Add: & prev. implementation to game

NEAT

Traditional AI vs NN
- trade offs

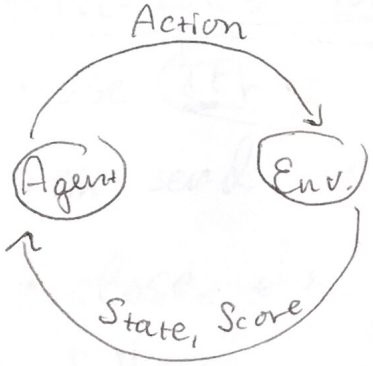

environment:
Start Program!
Setup Game
Print Game
While Training
Start Game
"loop"
Brain
Get Action
NN
Process Game
Update
while: not game_over
Analyse score
feedback
Print Game
Save Data Model
Print Overview
make apple_collision part of move()
Start
Make Apple part of game
Respawn Apple
Make a sequence of apples
Make a sequence of moves


Setup Game
Save current game
T
Print
spawn apple
while: game_over is False
F
Get Action
NN
Update Game
Eval. Score
Print
Print Game & Evaluation
Gameloop
Save all Games in list
Save Data Model
Print Result

- [x] read "getting started w gym", 15m
- [x] do FrozenLake tutorial, 1h
- [ ] get_score()

- [x] reformat mac
- [x] set up mac
- [x] linux commands
- [x] vir envs
- [x] tf. tut
- [x] fix env in proj.
- [x] ~~Kite tutorial~~
- [x] Atom setup

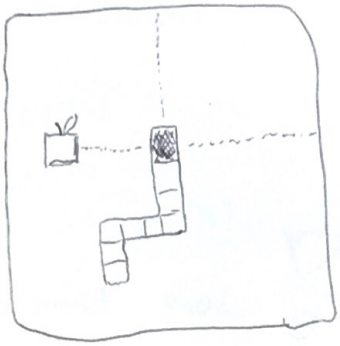
Markov Decision Process

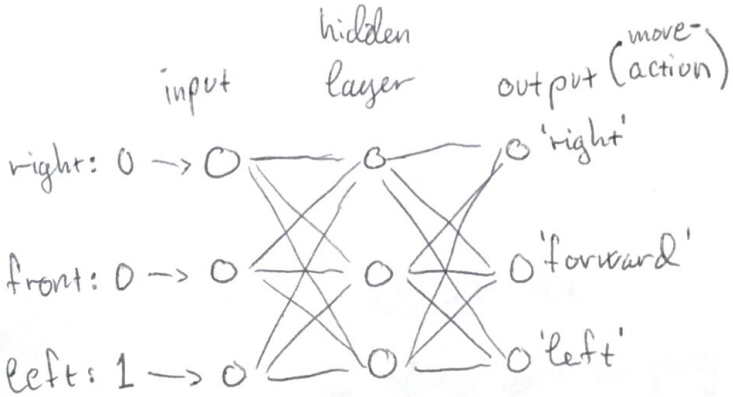


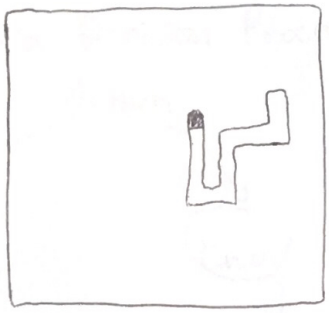
RL

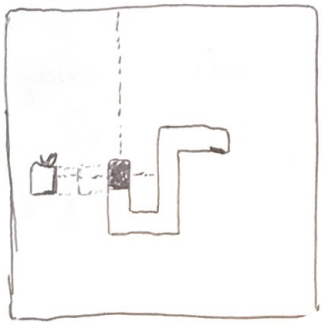
- we need to explore the environment first, and allocate a set of states with their dedicated scores.
- Agent learns to map "state-action" pairs to rewards!
- A policy maps a state to an action

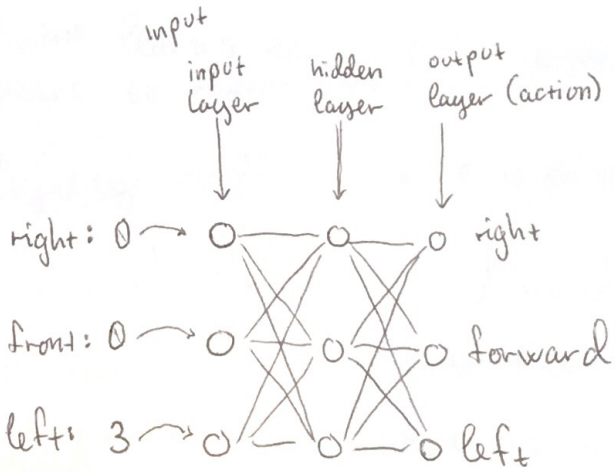
- At beginning the coefficient might be initialized randomly, getting feedback, it can ~~change the~~ adjust the weights & improve it's interpretation of state-action pairs.




input
hidden layer
output (move-action)
right: 0
front: 0
left: 1
'right'
'forward'
'left'



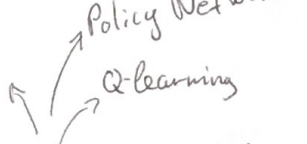



input
input layer
hidden layer
output layer (action)
right: 0
front: 0
left: 3
right
forward
left

Plan: Thu 14 Feb 12:00

- work on report!
  - fix references
  - fix styling
  - look over literature review
  - fill in on methodology

- read 2 white papers!
  - Tech Note: Q-learning



Q: can I write "about" concepts outside the ~~lit~~/tech review?

Q: Is it ok that I refer to many webpages as long as I have also a good amount of "proper" sources?

$Q[s, a]$ = immediate reward + discounted reward (discounted & optimal future reward)

scalar

$= r + \gamma \max(Q(s', a'))$

Policy: $\pi(s)$ = what is the policy/action when in state s?

we take advantage of our $Q[s, a]$ (Q-table) to figure that out!

$\pi(s) = \text{argmax}_a (Q[s, a])$:

↳ argmax will step through all actions in a given state, and return the maximum value/reward/quality of that state.

↳ ~~We will eventually~~ After running Q-learning for long enough, we will eventually converge to the optimal policy: $\pi^*(s)$ $Q^*[s, a]$

learning rate: usually 0.2 -> [0, 1]

$Q'[s, a] = (1-\alpha) Q[s, a] + \alpha \cdot$ improved estimate

Q Learning:

expected long-term reward

$$Q(s,a) = E[r]$$

State, action

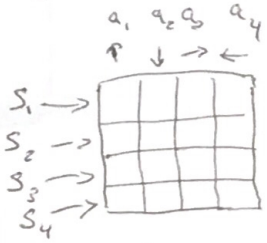
$$Q(s_t, a_t) = \gamma^0 r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots$$

$\gamma^0 = 1$; Discount rate

target value

$$Q(s_t, a_t) = r_{t+1} + \gamma \max(Q(s_{t+1}))$$

discount rate

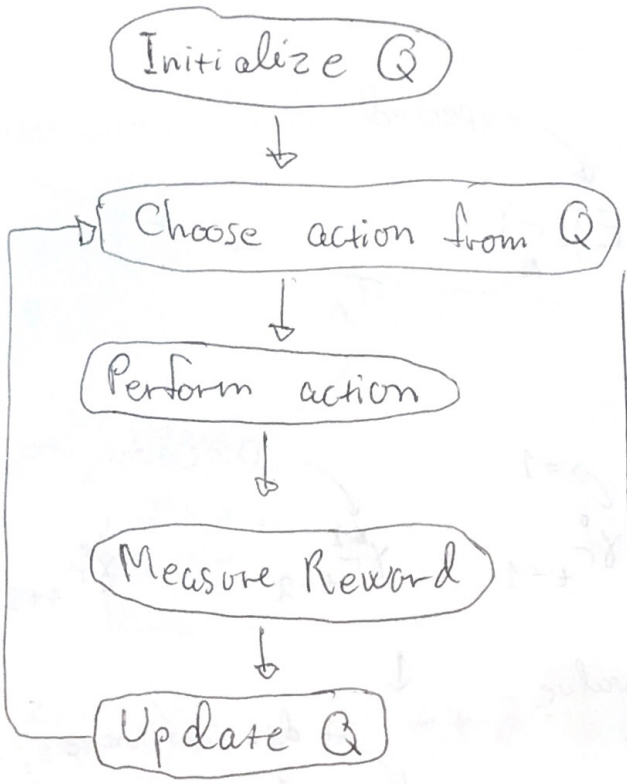


in each element in our table

- We will start with random values, and make small ajustments as we go. The algorithm will slowly learn the Q-values.

learning rate

$$q(s,a) \leftarrow q(s,a) + \alpha(\text{target} - q(s,a))$$



Model Based learning vs Model Free learning

↓

Not having any concrete "model"/percept. of the env.

$Q[s_1, a_1]$ = immediate reward + discounted reward

$= r + \gamma Q[s_2, a_2]$ full lookahead → Monte Carlo Technique

$= r + \gamma r + \gamma Q[s_3, a_3]$ step look ahead → TD (temporal differe.

$= r + r + \ldots$ How many steps is best to look ahead?

= the sequence of all rewards

vs.

Q Learning:

expected long-term reward

$$Q(s,a) = E[r]$$

State, action

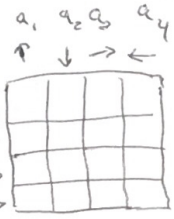
$$Q(s_t, a_t) = \gamma^0 r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots$$

$\gamma^0 = 1$; discount rate

target value

$$Q(s_t, a_t) = r_{t+1} + \gamma \max(Q(s_{t+1}))$$

discount rate



in each element in our table

- We will start with random values, and make small ajustments as we go. The algorithm will slowly learn the Q-values.

learning rate

$$q(s,a) \leftarrow q(s,a) + \alpha(\text{target} - q(s,a))$$

- [x] Make 'cause_by_death' variable avg. - rand vs. pred.
- [x] Make apple only able to spawn outside snake_body
- [ ] Make epsilon slowly approximate min_value.
- [ ] Give the snake a tiny ~~reward~~ penalty for going into a circle and a reward for exploring.
- [ ] Make it train until user-stops.
- [ ] Make script GPU/TPU compliant
- [ ] Implement pygame.
- [ ] Remove 7th-9th input (move)
- [ ] Fix image snapping in latex

- [ ] clean up the whole system.
- [ ] spell-check the report!

state [1, 2, ..., 15] = onehot = [

[1,0,0,0,0,0...0]
[0,1,0,0...0]
[0,0,1,0...0]
⋮
[0,0,0,0...1]

]

Q Learning:

expected long-term reward

$$Q(s,a) = E[r]$$

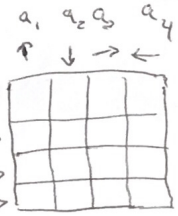
state, action

$$Q(s_t, a_t) = \gamma^0 r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots$$

$\gamma^0 = 1$; discount rate

target value; discount rate

$$Q(s_t, a_t) = r_{t+1} + \gamma \max(Q(s_{t+1}))$$



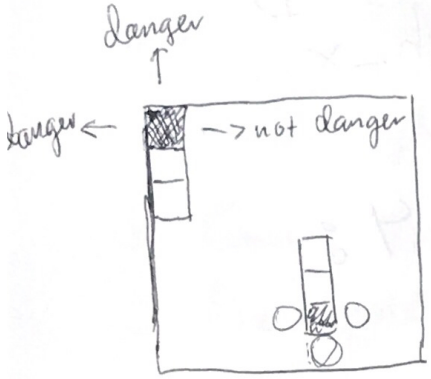
in each element in our table

- We will start with random values, and make small ajustments as we go. The algorithm will slowly learn the Q-values.

learning rate

$$q(s,a) \leftarrow q(s,a) + \alpha(\text{target} - q(s,a))$$

- [x] Do I need to feed the nn the move ?
  - yes ?
  - no ?
- [x] Include movement to state ?
  - not right now... (maybe later)
- [x] Convert list to onehot encoding !
- [ ] Fix: apple must spawn immediately !
- [x] Make state-array an np-array !
- [x] train_short_memory()
- [x] remember()
- [ ] Make unit-tests !
- [x] Fix Error Handling ! -> cause of "death" !



if 'up' -> up, right, left
if 'down' -> down, right, left
if 'left' -> left, up, down

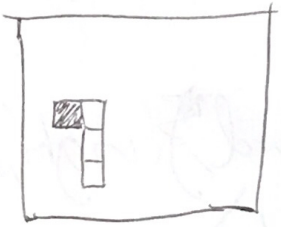
current_direction = "up"

neighbours = ~~get_neighbours(current_direction):~~

neighbours = [[-1, 0], [0, 1], [0, -1]]

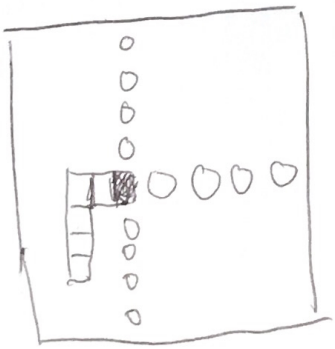
get_destination(current_direction):

get_long_sight()



~~if 'up' -> all but down~~

if 'right'



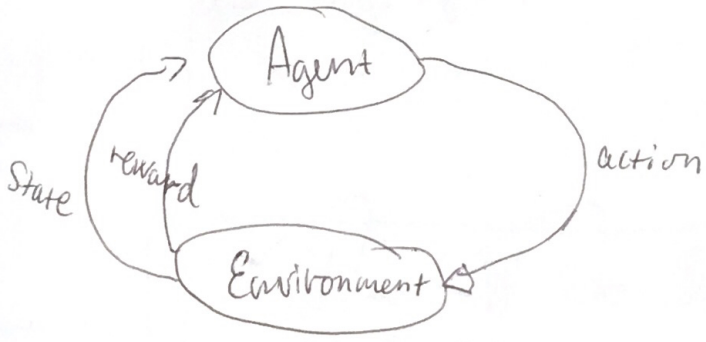
Is there a way we can teach the snake to play the game entirely on its own? without a training set or a testing set?

neural network → in RL it is called Policy Network.

RL is using ~~Policy Network~~ Policy Gradients to train the Policy Network

Loop:

1. We start off with a random network (weights, bias)
2. We feed the network a frame from the game-engine which produces an action
3. We send that action back to the game-engine
4. The game-engine is producing the next frame.



Goal is to optimize its policy, to receive as much reward as possible.

Sparse reward setting → Reward Shaping → manually designing reward function, to guide its policy

1. Initialise Agent
   1.1 Initialise neural network model
   1.2

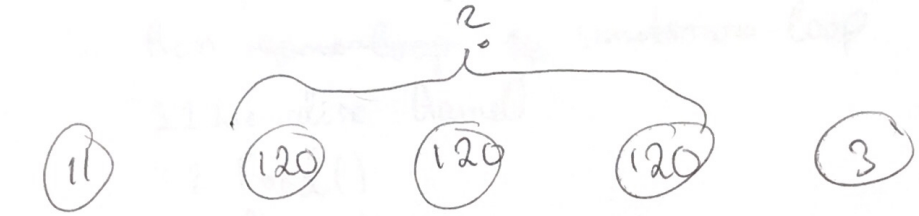
self.position = all segments?

1. Agent()
   1.1 Set parameters
   1.2. Init NN
2. Set parameters
3. Run ~~game-loop~~ simulation-loop
   3.1 Initialise Game()
   3.2 Food()
   3.3 Player()
   3.4 Run game-loop
      3.4.1 epsilon = 80 - counter_games
      3.4.2 get old state → state_old

- ✓ Initialise Q-table, 20m
- ✓ Print Q-table for each move/update, 20m
- ✓ Make network DQN - 50%
- ✓ Add epsilon
- ✓ Decrease epsilon for each game iteration

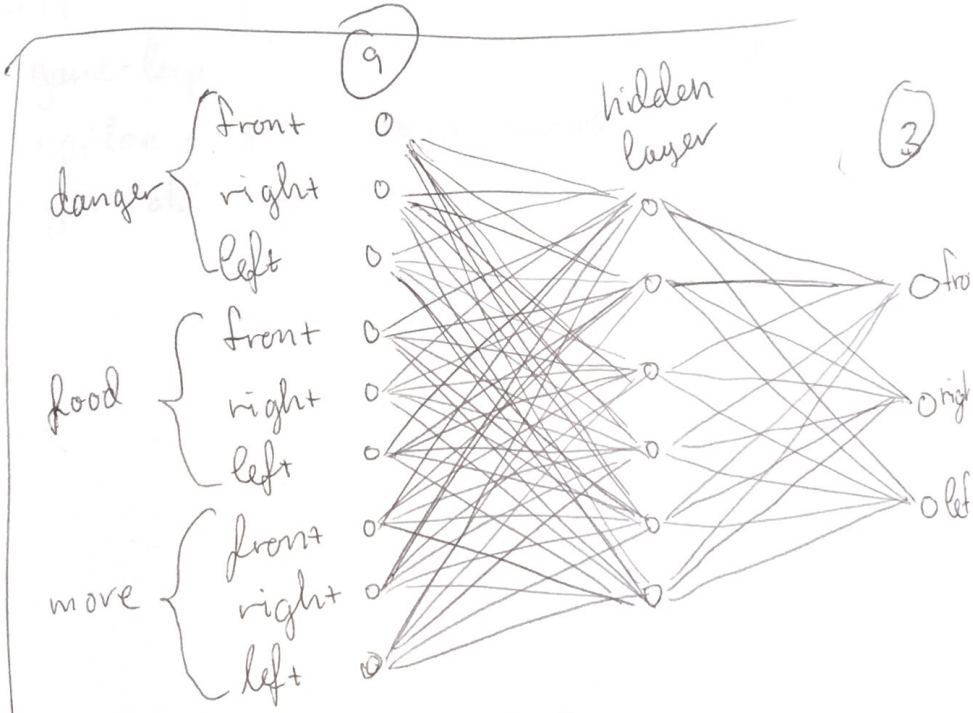
Agent, Food, Game, Player

- ✓ ~~find out what the pd.DataFrame is.~~



danger front ✓
danger right ✓
danger left ✓
move left ✓
move right ✓
move up front ✓
~~move down~~ ✗

food left ✓
food right ✓
food up front ✓
~~food down~~ ✗



- what is prediction?

- GPU/TPU - collab
- Print to multiple consoles ⟶
- API ?
- Diagrams are not "snapping" them to where I want them to be in the report.

- Testing
- Analysis
  - comparing
  - include the tables of the data.
  - here some learned
  - w fitness avg.

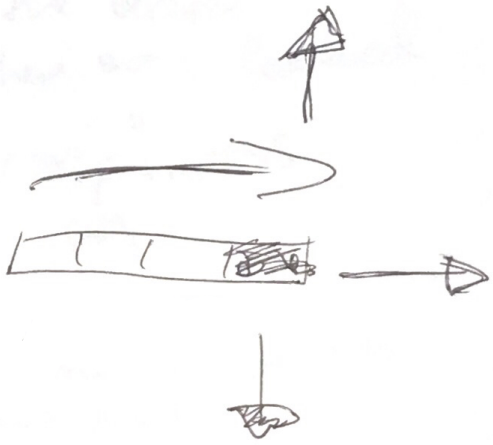
1 methodology
2 hypothisis
3 conclusion

- address some questions
- narrow the scope
- rewrite the introduction

Poster -> PowerPoint -> Infographics -> omnigraphel

Viva

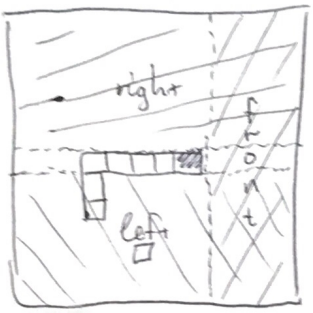
- ✓ Split up classes
- ✓ Reread article
- ✓ Check out 'to_categorical' -> onehot encoder
- ✓ Check if move is correct?! ~~it is?!~~ but absolute



-> Experience Replay

It is good to learn the rules, with little input data?!

- Give the snake more intel? (like in the snake-ga - repo...)?
  - should I do that, is it even fair



- Check prev_move vs. curr_move in orig. design.

- Make epsilon slowly approximate min-value dependent on the nr_iterations!

- Give a tiny penalty for going into a loop/circle → give reward for exploring

- Make it train until user-stop!

Start → End
80% → 10%

- ○ Tensorflow GPU
- ✓ End: print avg. steps, avg. eps., avg predicted_moves etc... avg-score
- ✓ Pass in ~~mc~~ current move? - see if it improves? - not sure?
- ✓ Save output 'plot' to folder (downloadable)
- ○ Print to ~~two~~ consoles
- ✓ Make equivilent epsilon function
- ✓ Fix epsilon

- ✓ Print current state for each move
- ✓ Make a step-out
- ✓ Run - ~~over~~ overwrite the model.
- ✓ Adjust the gamma=0.9

it is ower fitted!

Project Diary

Tasks:\\
[v] finding out what made the snake die\\
[v] give a small penalty for going into a small circle - not explore\\

    [?] fix sleep debug bug
    [v] learn how to take in arguments in python

    [v] if cell is being explored twice or more within the 10 last moves, then give a small penalty (-5) - explore
    [v] print reward and step number for each step
    [v] color code the reward

    [ ] make every 10th game play without training to be the set that is printed in the end (to get an honest test-feedback)
    [v] calculate regression line - degree of learning rate
    [v] make a version where we don't print out anything except a status-bar

    [v] put color on apple and snake head
    [v] print accumulated reward, aka fitness
    [ ] print average fitness per game
    [!] prompt user to make a new model, override model, or stash model
    [ ] make console output a table

[ ] make epsilon decrease depending on the nr of iterations\\
[ ] testing the snake performance\\
[ ] changing the neural network structure, activation, nr of layers, nr of nodes, I/O\\

[ ] include a table and diagram of a simulation output w avg. fitness score.

Sources to be researched:
* Why Can a Machine Beat Mario but not Pokemon?*
* article*: https://towardsdatascience.com/why-can-a-machine-beat-mario-but-not-pokemon-ff61313187e1
* src*: https://pastebin.com/ZZmSNaHX
* Ensemble learners*: https://www.youtube.com/watch?v=Un9zObFjBH0
* scikit-learn*:
general: http://scikit-learn.org/stable/
ensemble: http://scikit-learn.org/stable/modules/ensemble.html

* simplest neural network*: https://medium.com/technology-invention-and-more/how-to-build-a-simple-neural-network-in-9-lines-of-python-code-cc8f23647ca1
* Is it possible to combine two neural networks into one?* https://www.quora.com/Is-it-possible-to-combine-two-neural-networks-into-one
* Improving Predictions with Ensemble Model*: https://www.datasciencecentral.com/profiles/blogs/improving-predictions-with-ensemble-model
* But what*/is/*a neural network?* https://www.youtube.com/watch?v=aircAruvnKk
* Boxcar 2D:* http://www.boxcar2d.com/about.html
* matplotlib or matlab:* https://matplotlib.org or https://uk.mathworks.com/products/matlab.html
* Snake NN*: https://towardsdatascience.com/today-im-going-to-talk-about-a-small-practical-example-of-using-neural-networks-training-one-to-6b2cbd6efdb3
* Combining Artificial Neural Nets:* https://books.google.no/books?hl=en&lr=&id=_-rjBwAAQBAJ&oi=fnd&pg=PA1&ots=wsyvKDs7jE&sig=uFYMdQWmAOMoiL85PJnECfNf3aU&redir_esc=y#v=onepage&q&f=false

Log
* Mon 8 Oct 11:00 Meeting*
* include gantt cart
* table of requirements (moscow method)
    * features that will, should, and could be built in
* are there templates for the literature review: it is on sharelatex
* show evidence of active project management in final report

* Mon 15 Oct 11:00 Meeting*
general discussion in regards to report.
General outline:
    * Intro
    * Lit.Rev.
    * Methodology
    * Solution and Implementation
    * Evaluation and Testing

    * Conclusion and Future Work

* Mon 22 Oct 12:00 Meeting*
Deliver to Simon W for next week:
* a list of games people have been used in the past

Other deliverables:
* simple api between the NN and the game
* short script to move the agent
* what have been used in the past
* Remember to begin narrowing down what to read for the literature review
* Send skype id
* Get resources from friend

* Fri 2 Nov 13:00 - Goal*
* look at youtube-resource, 15m
* make random food generator, 20m
* continue design
* email, 12m

* Mon 5 Nov 11:40 Meeting*
Game â†□â†’ API: ML â†□â†’ Python ML
* [ ] Make a â€œrandomâ€□ prototype (to make random actions)
    * [x] get familiar with ML-toolkit/do tutorial
    * [ ] implement the ML-toolkit into snake game
    * [ ] add the â€œrandomâ€□ NN

* Mon 12 Nov 11:40 Meeting*

* Meeting With Second Marker: Simon P: 7th December, 11am*
Worth checking out the NEAT Library.
This grows the network and stops while a good solution has arrived.
Helps with not over or under fitting the model.
Every dissertation (Literature review) should have a good set of *research questions*.

In the methodology:
* justify your method (RL, NEAT, method of implementation (PPO, etc â€¦ and the customisation))
    * discussion in regards to over-, under fitting

Approach:
* Start off with one black-box NN (only input layer and output layer)
* customise the NN (penalty & reward)
* run training to produce a model (samples=30)
* check that it performs better than the black box NN
* run the same training again to get another model (samples=30)
* try to ensamble these two models together
* gather data and visualise
* run the same training a third time, but this time leave the number of runs in the game (samples) to 100
* gather data and visualise
* see which one of the ensamble or the deep NN performs better.

NOTES:
* what perspective will the snake have while playing?
      * from the players perspective?
      * or from the actual snakes perspective?
* what steps are needed to create the game?
      * maybe create a simple 2d plane and let a cube-agent move towards a goal-node
      * and build the snake game up by iterating and adding tiny components one by one
What Iâ€™ve learned, How the Learning Environment is connected to the Pthon API:
During training, all the medics (agents) in the scene send their observations to the Python API through the External Communicator (this is the behavior with an External Brain). The Python API processes these observations and sends back actions for each medic (agent) to take. During training these actions are mostly exploratory to help the Python API learn the best policy (optimal mapping from observations to actions) for each medic (agent). Once training concludes, the learned policy for each medic can be exported. Given that all our implementations are based on TensorFlow, the learned policy is just a TensorFlow model file. Then during the inference phase, we switch the Brain type to Internal and include the TensorFlow model generated from the training phase. Now during the inference phase, the medics still continue to generate their observations, but instead of being sent to the Python API, they will be fed into their (internal, embedded) model to generate the optimal action for each medic to take at every point in time.
/source: ML-agents toolkit on github/
To summarize: our built-in implementations are based on TensorFlow, thus, during training the Python API uses the observations it receives to learn a TensorFlow model. This model is then embedded within the Internal Brain during inference to generate the optimal actions for all Agents linked to that Brain. Note that our Internal Brain is currently experimental as it is limited to TensorFlow models and leverages the third-party TensorFlowSharp library.
Curriculum Learning, When we think about how reinforcement learning actually works, the learning signal is reward received occasionally throughout training. The starting point when training an agent to accomplish this task will be a random policy. That starting policy will have the agent running in circles, and will likely never, or very rarely achieve the reward for complex environments. Thus by simplifying the environment at the beginning of training, we allow the agent to quickly update the random policy to a more meaningful one that is successively improved as the environment gradually increases in complexity. In our example, we can imagine first training the medic when each team only contains one player, and then iteratively increasing the number of players (i.e. the environment complexity). The ML-Agents toolkit supports setting custom environment parameters within the Academy. This allows elements of

the environment related to difficulty or complexity to be dynamically adjusted based on training progress.
Reseach: PPO
Weâ€™re releasing a new class of reinforcement learning algorithms, *Proximal Policy Optimization (PPO)*, which perform comparably or better than state-of-the-art approaches while being much simpler to implement and tune. PPO has become the default *reinforcement learning*algorithm at OpenAI because of its ease of use and good performance. - OpenAI
A great algorithm, which is easy to implement and tune
Reinforcement Learning in Unity https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design.md
The ML-Agents toolkit uses a reinforcement learning technique called [Proximal Policy Optimization (PPO)](https://blog.openai.com/openai-baselines-ppo/) . PPO uses a neural network to approximate the ideal function that maps an agentâ€™s observations to the best action an agent can take in a given state. The ML-Agents PPO algorithm is implemented in TensorFlow and runs in a separate Python process (communicating with the running Unity application over a socket).

* Note:*if you arenâ€™t studying machine and reinforcement learning as a subject and just want to train agents to accomplish tasks, you can treat PPO training as a/black box/. There are a few training-related parameters to adjust inside Unity as well as on the Python training side, but you do not need in-depth knowledge of the algorithm itself to successfully create and train agents. Step-by-step procedures for running the training process are provided in the [Training sectio](https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md)