

A-Level Computer Science NEA

Joe Harrison

Contents

1 Analysis - 4

- 1.1 Introduction - 4**
- 1.2 Current AI Systems - 4**
- 1.3 AI in Games - 8**
- 1.4 Hardware AI Platforms - 10**
- 1.5 Issues with AI Systems - 12**
- 1.6 My AI system - 12**
- 1.7 Design Objectives -13**
- 1.8 End Users - 14**

2 Design - 15

- 2.1 Overview - 15**
- 2.2 Snake - 15**
- 2.3 Q-Learning - 17**
- 2.4 Neural Networks - 19**
- 2.5 Deep Q-Networks - 22**
- 2.6 Network Architecture - 24**
- 2.7 Development Tools - 25**
- 2.8 CLI Design - 26**
- 2.9 Leaderboard - 27**
- 2.10 Data Tables - 28**

3 Code Listing - 36

4 Testing - 50

- 4.1 CLI - 50**
- 4.2 Architecture - 50**
- 4.3 Compute Issues - 51**
- 4.4 Training and Testing - 52**
- 4.5 Leaderboard and Human Interface Testing - 59**

5 Evaluation - 64

5.1 Issues with Neural Network - 64

5.2 Other Potential Improvements - 65

5.3 Evaluation of Success - 65

Appendices

A References and image credits - 68

B Test screenshots - 70

C Full code listing - 74

1 Analysis

1.1 Introduction

My project is an investigation into neural networks, artificial intelligence (AI) and machine learning, with the goal of creating a simple neural network-based AI system. More specifically, I intend to research some applications of AI to find out about how it is being used in computer science and other industries to gain some inspiration about a system that I could develop myself. The primary goal of developing this system is to investigate and learn about neural-network based AI: it's capabilities, functionality, and the theory, mathematics and algorithms that drive it.

The aim of my analysis is to gain some insight into the basics of artificial intelligence, machine learning and neural networks, some of the systems that employ these techniques, the people that use these systems, and the issues associated with them. The purpose of this is to gain a basic understanding of how some of these much more complex systems operate, and find out about some of the techniques used in these systems to provide some inspiration as to what kind of system I will develop, and the algorithms and techniques I will need to use to develop it.

The details of my system will be discussed and design objectives set out towards the end of the analysis process.

1.2 Current AI systems and their users and applications

AI has such a wide variety of current and theoretical applications. It is already implemented in many ways in computer systems, and lots of cutting-edge research is being done to create new technologies to revolutionise industries.

Self-driving cars are possibly the most prominent example of AI research currently being done, due in part to the implications the technology has on the automotive industry. Companies such as Apple, Google, Nvidia and Tesla are all investing heavily in research and development, with varying degrees of success. Generally speaking, self-driving cars use sensors to gain an idea of their environment and surroundings, and then process this through some software to decide about what to do.



Figure 1.1 – Tesla's network of sensors that forms part of its autopilot system

Tesla uses a software system known as 'Autopilot.' Eight cameras and 12 ultrasound sensors positioned around the vehicle build up a 360-degree image of the environment at 250

meters of range, and a front facing radar provides visibility through heavy rain, fog and dust. All this data is fed into a deep neural network developed and trained by Tesla which gives the computer an understanding of the environment around it which it can interpret, and then act based upon this data. [1]

Training a neural network capable of processing this amount of complex data at speeds fast enough to make sense of an environment moving at 70+ mph is no small task: huge amounts of data and processing power is needed. A paper published by Nvidia “End to End Learning for Self-Driving Cars” gives some insight into how systems like these are developed. A type of neural network was used called a CNN, or convolutional neural network, which is used to optimise pattern recognition in images. The network is capable of taking images as input, recognising patterns and objects, and then assigning actions to them. The data required to train the network was collected by human-driven cars with cameras placed in the same positions as on the autonomous vehicle.

A wide variety of road types and conditions were used to help minimise bias in the network.

Approximately 72 hours of footage was collected at the time of the paper’s publication.

The weights of the network were trained to minimise error between the command output by the network, and the command of the human driver. The first layer of the network was hard-coded to perform image normalisation, to process the images from the footage into a format that could be fed to the network. The 5 convolutional layers performed feature extraction, with exact feature map sizes chosen through extensive testing and experimentation, and the final 3 fully connected layers were designed to act as a controller: to take the features mapped by the convolutional layers and decide what to do based upon the presence of those features. However, due to the nature of a neural network it is not possible to determine exactly the function of each layer.

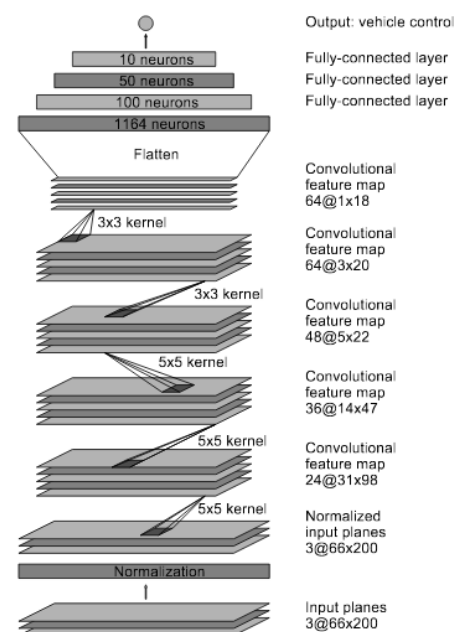


Figure 1.2 – Nvidia’s DNN designed for driving a car

The conclusion of their research was that CNNs are able to learn the task of following a lane or road in a diverse number of road and weather conditions, without manual processing or labelling of footage with less than a hundred hours of training data. This is a massive achievement, demonstrating the power provided by CNNs and that they can be applied to self-driving vehicle technology. However, more work is needed to improve and verify the robustness of the network, which is understandably important when it comes to handing over control of a car to a piece of software. [2]

At the moment perhaps the most advanced autonomous car systems available to consumers are those offered by Tesla, which are capable of many things including automated steering, braking and acceleration within a lane, navigating complex

environments, calculating optimal routes, and self-parking. However, a disclaimer on their website states “Current Autopilot features require active driver supervision and do not make the vehicle autonomous.” The features and technology are all there built into the vehicle, but more testing and refinements are needed before the vehicles can be declared fully autonomous and operated as such. [1]

For the moment, such technology is targeted only at those able to afford luxury vehicles with a starting price of \$35,000 USD for the most basic of models, all the way up to \$140,000 for the most expensive models. As the technology progresses and becomes more mainstream, it is hoped that it will become more affordable so that it is accessible to a wider range of users

Obviously, the impact of a system capable of driving a car to the same or better level than a human would have massive impact. The applications of it would be many, such as fleets of taxis, delivery vans, and military vehicles which would all no longer require drivers. All commercial driving jobs would be eliminated, which could raise some ethical concerns, but save millions for companies that rely on humans to drive. Humans may never have to operate a vehicle manually again, representing the biggest leap in transportation since the invention of the petrol engine.

Another application of artificial intelligence is in computer graphics and rendering. This is a field that is advancing at a rapid rate, with the ‘Graphics Turing Test’ being passed a long time ago, meaning it is possible that images and worlds artificially generated are indistinguishable from real ones. Image processing is a task well suited to AI as it is not one with a clean algorithmic solution.

Nvidia has developed a variety of AI based tools based around graphics and rendering. NGX technology uses deep neural networks to accelerate and enhance graphics rendering, and provide a variety of useful tools including: removing unwanted objects from images, and using AI to computer generate alternatives automatically; interpolating frames in a video to provide smoother or slow motion video; and upscaling resolution by interpreting images and intelligently placing new pixels, as oppose to current upscaling systems which simply stretch out existing pixels. These are all ground-breaking techniques that have not previously been possible to automate or have required manually coding. Using neural network-based AI makes the tasks a lot simpler to implement and the results much more accurate.

DLSS, or deep learning super sampling, is another technology developed designed to be used in real time video game rendering to upscale, enhance and sharpen frames to provide in-game visuals of a higher quality. It works by extracting features from the frame using a deep neural network, and then reconstructing them at a higher quality. This can be used to either increase resolution of a game while its being played and rendered at a lower resolution, or to sharpen and enhance details at the same resolution, all in real time. It provides image quality and results similar to that of traditional techniques, with much faster performance. It also provides more intelligence in image processing. For example, TAA, or temporal anti-aliasing, seeks to dampen the effects of temporal aliasing, an effect that occurs when the sampling rate of a scene is too low for motion to appear smooth. TAA

lessens this effect by attempting to blindly follow the motion vector of the object, making the movement smoother, but often blurry. DLSS can more intelligently analyse scenes, removing the effects of motion blur caused by this effect and giving much crisper animations.

To achieve this level of performance, another deep neural net engineered by Nvidia is used. To train their network, they rendered thousands of reference images using the best available rendering techniques such as 64x super

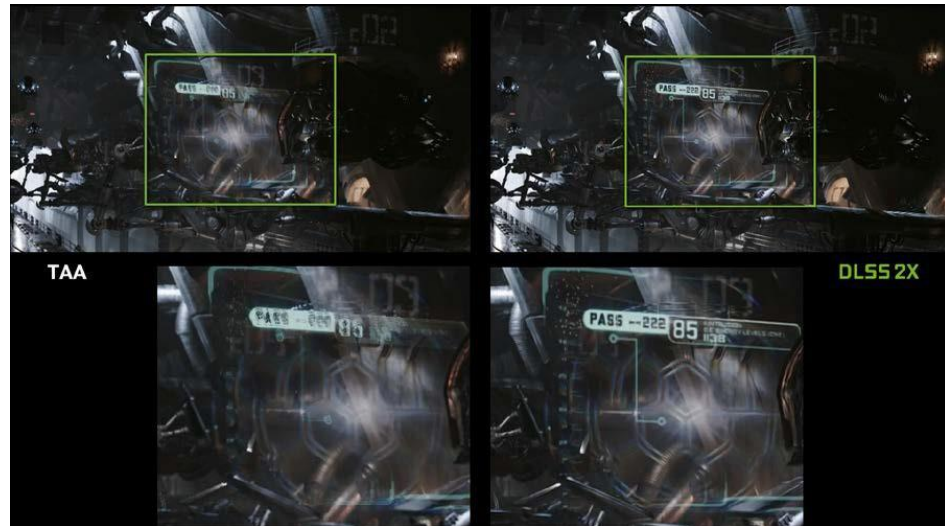


Figure 1.3 – Improvements provided by AI super sampling techniques

sampling, meaning that instead of shading each pixel once, the pixel is shaded 64 times and then combined producing an image with ideal detail and anti-aliasing quality. The same frame is then rendered by the network, and the error is calculated as numerical differences between the pixel values, and the backpropagation done according to this value with respect to the weights of the network. Other image processing neural nets are trained in a similar manner, with the network comparing the image it produces to an image edited or rendered to a gold standard, and using these images to improve its accuracy and performance. [3]

Image processing is typically an application that computers have struggled with, as it is a tricky optimisation problem on data that computers cannot easily gain a broad understanding of: a computer cannot easily tell what an image is of or extract details from it. AI has provided breakthroughs in computer vision and image rendering, significantly speeding up existing processes and creating new ones not previously thought possible.

The users and applications of such a technology are again, extensive. Gaming systems can take advantage of the real time DLSS to improve image quality, providing a much higher quality gaming experience using lower processing power, and therefore at a lower cost, which would have extensive appeal to anyone with an interest in gaming. Visual and creative arts professionals may be more interested in the NGX technologies, which provide incredibly powerful tools that replaces processes that would previously have to have been done manually, for example, object removal and infilling.

These types of artificial intelligence are bleeding edge technology, so as such would be far more complex and require far more time and resources than are at my disposal to implement. However, the insight they provide into the research and techniques used in

industry currently, as well as the power provided by neural networks, is invaluable and will aid me in my research going forward, and also assist when designing my system.

1.3 AI in games

During my research I came across an area of AI that particularly interested me – gaming. Comparing AI performance in video games with that of humans has always been a benchmark of the capabilities of AI. Many techniques have been used in the past to implement or emulate game AI, however most of them have been based on fixed decision-making trees or processes that essentially employ a long chain of if statements. Recent breakthroughs in neural network-based machine learning techniques and algorithms have allowed game AI to become more dynamic and intelligent, approaching and even exceeding human level behaviour in many games.

A research team at Google-owned AI company DeepMind published a paper in January 2015 titled “Human Level Control Through Deep Reinforcement Learning,” which detailed breakthrough techniques in using neural networks and reinforcement learning to play games, specifically classic Atari 2600 games such as Pong, Space Invaders and Breakout. The team created a new kind of reinforcement learning-based neural network, termed a deep Q-network, which can take a high-dimensional input, process it and decide on an action to take. What this essentially means is that given only images of the frames of the game it is playing and the current score, the system can learn to play the game.

The network works by taking 4 consecutive frames of the game, and passing them to the network as a tensor (n-dimensional array). Similar to Nvidia’s self-driving car network, 3 convolutional layers extract detail from the frames, and then 2 fully connected layers decide upon the action to take based upon these features. The network has an output layer, with one output node for each action available on an Atari joystick control. The network receives no other data about the game it is playing, except the frames of the game as it is being played. The same network was used for all the games with the same hyperparameters, demonstrating massive versatility: something previously very difficult to achieve with neural networks.

The versatility comes from the nature of the deep Q-network, which works by evaluating the quality of each action taken by the agent within the game environment as a numerical value, and approximating this quality, or ‘Q’ value using the network. The network outputs a Q-value for each possible action, based upon the possible score it could gain, and the action with the highest Q-value associated with it at that point is taken. The network does not need to know anything about the game, only the score so that it can evaluate the Q-value, and the state of the game environment so it can learn how to increase this score based upon its actions.

The performance and versatility of the system was considered a breakthrough, and techniques pioneered by the DeepMind team as part of their research have become very widely used within other AI systems. The image to the right illustrates the performance of the network. The scores are represented as a normalised percentage, 0% being the score achieved through selecting random actions, and 100% being the score achieved by professional game testers. [4]

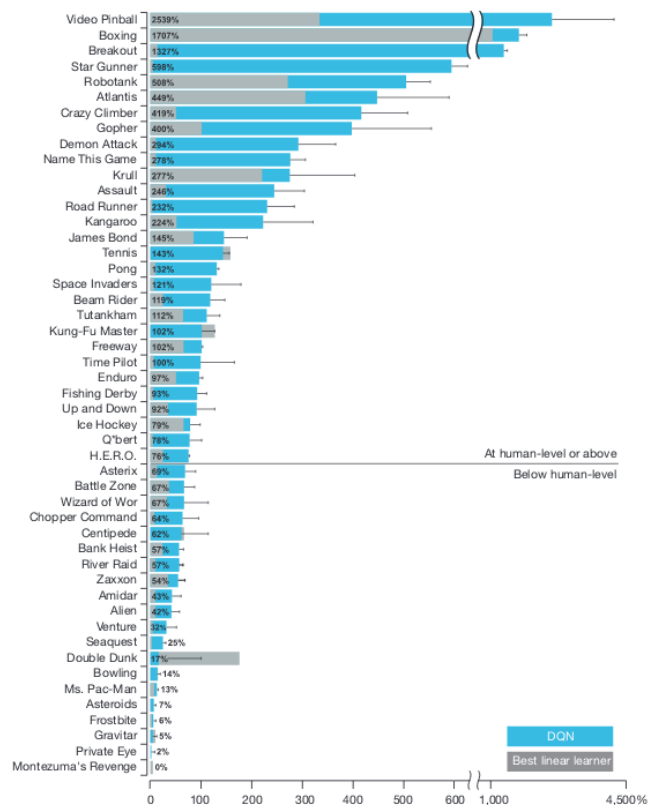


Figure 1.4 - The performance of the same neural network in many arcade games against human players

DeepMind's researchers have since progressed from Atari games to much more complex games. StarCraft 2 is one of the most popular, but also challenging real-time strategy games, and also a very popular E-sport with a large pro scene. The huge complexity of the game drew researchers to it as a challenge. In January 2019, DeepMind published a blog post on their AlphaStar system, the first AI system to defeat top professional players. Grzegorz "MaNa" Komincz, one of the best and well-respected StarCraft players in the world, was beaten 5-0 in a series of full matches under professional conditions.

Until recently, AI has struggled to cope with the complexity of StarCraft, despite success in other popular games such as Mario, Quake 3, and Dota 2. What little success had been had was achieved through imposing restrictions on maps or game rules, or using manually coded algorithms for parts of the game. Instead, AlphaStar plays the game with no restrictions or rule changes using a deep neural network trained partly by supervised learning, and partly by reinforcement learning techniques similar to those detailed above with the Atari system.

AlphaStar's deep neural network takes input data from the raw game interface, and outputs a series of actions to be taken within the game. The network was initially trained by watching human-played games, which allowed the network to learn the basics of the game and begin to develop some initial simple strategies which allowed it to defeat some less skilled human players.

This initial agent was then used to create multiple reinforcement learning agents which all played in a league against each other. This is similar to how humans experience and learn to play StarCraft, and allows the agents to experience a larger range of strategies. As the league

progressed, the agents developed and refined their strategies, and also learned how to counter other common strategies. Each agent was given a unique objective to encourage diversity in the league, for example which other competitors to focus on beating, or specific strategies to develop on, specified through internal biases that motivated how each agent plays.

The training was performed on a large cluster of Google cloud TPUs (Tensor Processing Units) running many thousands of parallel instances of StarCraft. The league was run for 14 days, giving nearly 200 years of real-time play per agent. The final agent was made by combining the most effective strategies of the agents from the league to gain the optimal mixture of strategy, which ran on a single desktop GPU.

AlphaStar's performance did not come from its superior speed: in fact, it performed actions at a much lower rate than the professionals it played against. Instead, it's superior ability to macro and micro-manage in game units and resources, as well as the ability to gain a large-scale picture of the game map instead of having to refocus its attention constantly like a human would, contributed to its success. [5]

While playing video games is fun, it does not really have much direct practical application. However, it is the techniques behind AlphaStar and the Atari DQN that may be widely applied to solve more important problems. AlphaStar was capable of modelling incredibly long sequences of actions from games lasting up to an hour, proving that AI can make reliable predictions over long periods of time: an incredibly useful skill that can be applied to a wide range of problems such as weather forecasting and financial planning. The versatility of the Atari DQN demonstrates that neural networks do not have to be specialised to a single very specific purpose, and that by simplifying and generalising input parameters, a single network can be applied to a much wider variety of tasks. These experiments have shown how some of the classical issues with machine learning are being overcome, setting a precedent for rapid advancement in the field of AI research.

The research being done by DeepMind interested me particularly as I have an interest in gaming, and the technology behind it. It also demonstrates that a solution to something which seems to have little practical application can provide so much insight into many real-world problems. This presents the idea of perhaps implementing a system similar to the Atari DQN using the same algorithms and techniques, albeit simplified to something more suited to a beginner, and coding perhaps a simple arcade game for the agent to play as well. Deepmind's methodology in testing and evaluating the performance of their systems is also useful information, as it demonstrates the industry standards on benchmarking AI, and will provide a starting point for me in testing the performance of my system.

1.4 Hardware AI platforms

One of the major issues plaguing neural network-based AI at the moment is processing power. AlphaStar took a week to train, and that was running on the most powerful hardware

available to the team. Neural networks are made up of many identical nodes or ‘neurons’, and as such are highly parallel by nature. This provides many ways to mitigate the issue of runtime, through specialised hardware designed to accelerate the specific matrix operations through parallel processing.

In 2016, Google announced the release of their TPUs, Tensor Processing Units. These are hardware chips designed specifically for use with their machine learning framework, TensorFlow. The bulk of neural network processing is done with matrices, and Google’s TPUs contain multiple MXUs, matrix multiplication units, which are designed from a hardware level to accelerate and massively parallelise dense vector and matrix operations, providing huge speedups over standard CPU processing. TPUs are available in the cloud, making them widely accessible and a powerful tool to anyone looking to work with complex neural networks with long training times. [6]

For smaller scale networks, prototyping and testing, Nvidia’s latest GPU architectures, Turing and Volta, are designed from the ground up to accelerate matrix operations with tensor cores, a new hardware accelerator developed by Nvidia for accelerating machine learning workloads. Nvidia GPUs have been used for machine learning for a long time, due mainly in part to their CUDA programming language, which enables and simplifies parallelising workloads to run on GPUs, but the introduction of tensor cores has provided further acceleration to these workloads.[7]

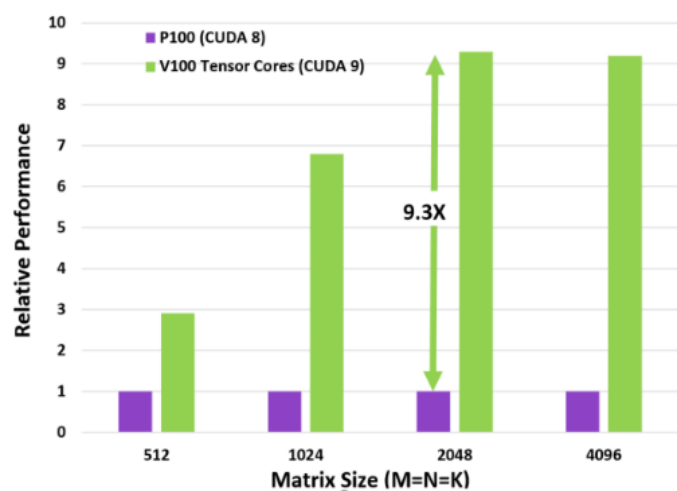


Figure 1.5 - Matrix multiplication performance of Volta tensor cores over Pascal

These hardware solutions have helped lower the previously high barrier to entry for working with neural networks, as the processing power required can now be acquired much cheaper, either through cloud instances or physical hardware. It has also reduced the time taken for training more complex models that were not previously possible.

Nvidia’s CUDA platform interested me as I own a Nvidia GPU. Training time and processing power was one of the concerns I had going into this project, but with some research into my specific hardware, I have found that many machine learning frameworks including Google’s TensorFlow provide support for my hardware, meaning that I will have the processing power required to build and run a simple neural network very efficiently.

1.5 Issues with current AI systems

The technical issues with AI at the moment are perhaps the most obvious. Processing power is one that has already been touched upon, with complex neural networks requiring weeks of processing time and incredibly powerful machines to train. One of the main issues is hyperparameter optimisation. A hyperparameter is defined as anything not trained automatically by the algorithm that has to be defined by the programmer such as network architecture, learning rates and activation functions. Tiny changes in these can have massive impacts on the performance of the network, and there are a massive number of combinations so it can be difficult to find a true optimum. Often people rely on defaults, but this can cause issues as it is rare that they are near the optimum. Lots of tweaking and experimentation is required which can be time consuming.

There are also many ethical issues surrounding AI. For example, Microsoft made a tweetbot AI named Tay, a twitter account set up to learn how to interact with people based upon how they interact with it. It didn't take long for people to start tweeting racist and sexist comments at it, and as a result of what it had learned from these interactions, the AI decided to tweet out calling for genocide. Obviously, this is an issue – but does the problem lie with Microsoft's poor AI design, or is it with the internet for teaching the bot to behave as it did? [8] Further issues involve racial bias in facial recognition software, and the legal responsibilities surrounding autonomous vehicles in the event of injury or death in crashes.

However, the issue perhaps most pertinent in this situation is education. It took a lot of research to be able to find any comprehensive resources aimed at beginners that goes into the technical detail required to be able to design and program a neural network from scratch, and even then, much of it was poorly documented or incomplete. Many articles or papers about AI such as the ones by Nvidia and Deepmind are written with the assumption that the reader already has a good level of technical knowledge, and as such are inaccessible to beginners interested in the subject.

1.6 My AI System

The Deepmind research into game AI interested me particularly, so I decided to take inspiration from their research and develop an AI to play one of my favourite arcade games, snake. Snake is one of the simplest arcade games around, but is still complex enough that designing an AI to play it provides enough of a task.

Another game I considered was pong, but decided against it because it's very simple and any neural network would be outperformed by some simple code just matching the position of the ball. I also considered Pac-Man as another one of my favourites, but coding a Pac-Man game in Python from scratch would take up much of the time, when I want the focus to be on the AI playing the game.

I will also be using the Deep-Q reinforcement learning techniques pioneered by Deepmind and applying them to my AI system. This technique interested me as it has many parallels to the way humans learn, and the mathematical modelling behind the algorithm is also

fascinating, so by using a Deep-Q network design for my neural network I hope to learn more about this algorithm that is being used at the forefront of computer science research.

As I also have access to GPU hardware with hardware level-optimisations for deep learning, I will run the neural network using TensorFlow's CUDA interface to investigate the advantages that parallel compute and hardware acceleration can provide to this application.

1.7 Design objectives

The specific system design objectives for my snake game and neural-network based reinforcement learning algorithm are outlined below.

1.To write a snake game.

- 1.1 The game should consist of a player or computer-controlled snake on a grid-based game board that moves forward, and can turn left or right.
- 1.2 The game should randomly generate an apple that the snake can eat.
- 1.3 When the apple is eaten the snake should grow by a length of one unit, the score should increase by 1 point, and another apple should be placed on the game board.
- 1.4 The game should reset when the head of the snake touches either its body or the edge of the game board.
- 1.5 The score should be visible by the player for example in the form of a counter in the corner of the screen.
- 1.6 The game should be designed in such a way that it is playable by both a human and AI

2.To design and implement a neural network-based artificial intelligence to play the game.

- 2.1 Deep-Q machine reinforcement learning algorithms should be employed to train a neural network to play the game.
- 2.2 The network should take some information about the state of the game environment as input.
- 2.3 The network should return some information that determines what action the snake should take.
- 2.5 The network should be trained so that it can play the snake game to a decent standard.
- 2.6 The network should be able to be saved to a file when finished training, so that it can be loaded back into the system to play the game.

3. The neural network should be tested to evaluate its performance.

- 3.1 The performance of the network as it trains should be tracked to investigate how a neural network learns and makes decisions.
- 3.2 Data should also be collected on the trained network as it plays the game to evaluate its performance and behaviour
- 3.3 The performance of the AI should be compared to that of a human to evaluate the effectiveness of the machine learning system.

4. The game should save high scores to a file, such that a leaderboard is created.
 - 4.1 The leaderboard should be viewable and display the scores, name of the person, who got the score and the date they achieved it on, ordered by score.
 - 4.2 The scoreboard should show the top scores for all players, as well as the top scores for each individual player
 - 4.3 Scores should be saved by both human and by AI players, so that it can be used to compare the performance of the two.
5. The network should be run on a GPU to investigate compute performance of GPU vs CPU for neural network applications.

1.8 End users

As my project does not revolve around developing a system with any practical application, there isn't an intended end user as there would usually be. In fact, the whole idea of AI revolves around removing the need for a user in the first place. Developing an AI that performs any useful task with accuracy is a project far beyond the scope of A-level, so my design objectives lend themselves more to an investigation into how neural networks and machine learning work two reasons: for my own educational purposes, as it is an area that interests me; and to demonstrate the functions and capabilities of AI so that the ideas behind my system could perhaps be used as a starting point for larger systems.

As this project is more investigative in nature, one of the main goals of it is education. I am developing this system for myself in the sense that its purpose is not to perform some set task, but for me to learn about neural network-based artificial intelligence. Most of the design choices I make will be with this as the primary objective. I have already mentioned how education and barrier to entry is the most relevant issue to this situation, and the primary issue I will need to overcome with this project. My project could also serve as a starting point to other students similar to me, with no prior knowledge looking to develop AI or reinforcement learning-based systems.

2 Documented Design

2.1 Overview

As outlined in my objectives, the program needs to be able to perform a few different functions, all centred around the snake game. An AI needs to be able to train itself to play the game, and then play it, as well as a human being able to play it. A leaderboard also needs to be created and interacted with. As this will require lots of different functions, it makes sense to encapsulate all the subroutines and variables associated with a specific part of the code within python classes. This way, the code is much cleaner and more readable, and also the functionality that is required can be initialised and called as needed from its class. Full class listings are below, along with an ER diagram showing their interactions.

As all the functionality will be implemented with a single code file, a way for the user/player to choose what classes to call is needed, so a simple user interface will be designed as well. details as to the design of this interface are outlined below.

Also outlined below is the basics of the algorithm that underpins the snake game, and some details as to how the leaderboard scores will be saved as a file. There is also a technical explanation of how the Q-learning and neural networks work on a fundamental level, and the design of my neural network architecture and algorithms. All designs outlined are just outlines, and the exact details of implementation will be finalised during coding.

2.2 Snake

Before any machine learning can be done, there needs to be something to learn.

Snake is a classic arcade game originating from the 1976 game blockade, but was popularised when a version came preloaded onto Nokia mobile phones in the late 90s. Since then there has been many iterations of the game, and it is one of the most well-known arcade games, as well as my favourite.

Snake is very simple; the player moves a snake around a board trying to collect apples. When the snake eats an apple, it grows longer. When the snake collides with itself or the wall, it dies. This is very simple game logic makes the game fairly easy to code. The core game loop is shown in the flowchart in figure 2.1.

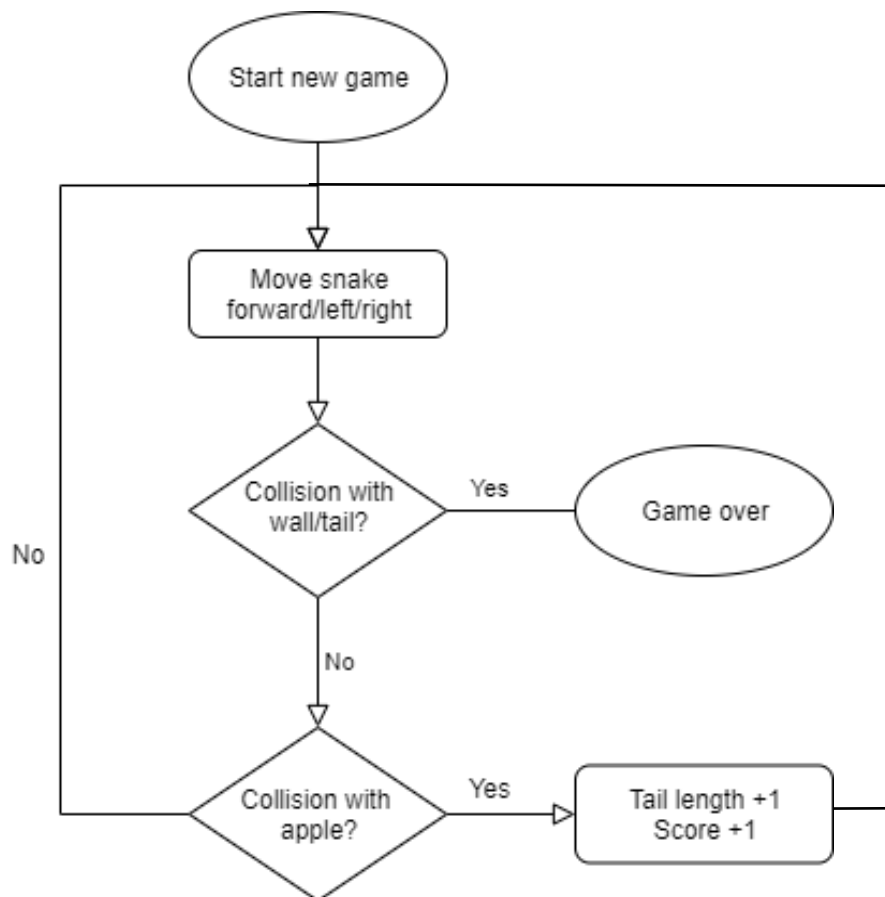


Figure 2.1 - Core snake game loop

Keeping the game simple allows for my focus to be on designing the AI and not having to worry about the snake code too much.

The graphics for the game will be implemented using a grid. A matrix data structure will be used to represent the game grid with the number at each position in the matrix representing what is there. This makes collision detection very simple, as the only check that needs to be carried out is what number is at the given position in the matrix. It also provides a layer of abstraction internally within the code, as the rest of the code can interact with the display and draw the graphics by simply modifying entries within the matrix, while the class representing the grid handles all the pygame rendering logic for drawing to the game window.

The game will be programmed in an object-oriented fashion, with objects representing the snake, apple, game grid/window, and an object for the main loop to run in and pull all the bits together. This also makes for ease of use when designing the machine learning algorithm so parts of the game can be interacted with through calls to the objects encapsulating them.

2.3 Q-learning

To train the snake, I will be using the Q-learning algorithm, a kind of reinforcement learning. This algorithm was outlined in the paper published by DeepMind that I discussed as part of my analysis.

Reinforcement learning works by observing the actions of an agent in an environment, and punishing it or rewarding it based upon its actions. In our case, the agent is the snake, and the environment is the game grid. The actions of the snake can be modelled by a Markov Decision Process, or MDP. MDPs are used for formalising sequential decision making, and provide the basis for most problems involving reinforcement learning.

We have a few components of our MDP:

Agent – The snake, the thing that makes the decisions

Environment – The game board with the snake and apple on it

State – A way of describing the environment formally

Action – The movement of the agent, or snake in this case

Reward – The consequences, positive or negative, of our actions. In this case, the score.

At each step, the agent selects an action based upon its state, transitions to a new state based upon this action, and receives a reward (or not). This sequence happening in a loop creates a trajectory, and the goal of the agent is to create a trajectory which maximises its cumulative rewards.

To formalise these ideas mathematically, at each time step t , the agent observes the state of the environment $S_t \in S$, where S is the set of all states. Based upon this observation, an action $A_t \in A$ is taken, where A is the set of all actions. Time is then incremented to $t = t + 1$, and a numerical reward R_t is given to the agent based upon the state-action pair (s_t, a_t) . This happens over and over in a loop, nicely illustrated by figure 2.2.

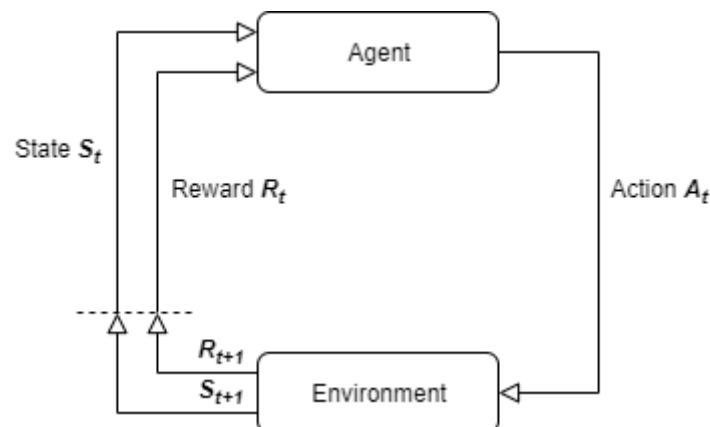


Figure 2.2 – Markov Decision Process for reinforcement learning

Going back to the idea of maximising rewards, the goal of our agent is to navigate the MDP in such a way to get the largest possible cumulative reward. This means it wants to know what reward it is going to get from its immediate action, and then based upon that predict possible future rewards. Predicting the future is no small task, so we need to do some more maths.

Cumulative reward can be described as the sum of all rewards given to the agent at each time step:

$$P_t = R_{t+1} + R_{t+2} + R_{t+3} + R_{t+4} \dots$$

Assuming time goes on indefinitely, as we don't know when it will stop, this is an infinite sum and as such the cumulative reward P_t doesn't really mean much. To solve this, we will introduce the discount factor γ . Obviously, future rewards are less important than immediate rewards- the snake eating an apple right now is better than the snake maybe eating an apple in 20 steps time – and we should adjust our sum to account for this. Where $0 < \gamma < 1$, we can redefine our sum of rewards as:

$$P_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \gamma^3 \cdot R_{t+4} \dots$$

You can see from the sum how gamma is being used to discount future rewards and place heavier emphasis on immediate rewards. A bit of rearranging and this sum collapses nicely:

$$P_t = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+2} + \gamma^3 \cdot R_{t+3} \dots$$

$$P_t = R_{t+1} + \gamma \cdot (R_{t+2} + \gamma \cdot R_{t+3} + \gamma^2 \cdot R_{t+4} \dots)$$

$$P_t = R_{t+1} + \gamma \cdot P_{t+1}$$

This infinite sum yields a finite result which is quite intuitive and describes nicely how discounted cumulative reward works. It is also recursive – To know cumulative rewards at time t , it must know cumulative rewards at time $t + 1$, and to know that it must know cumulative rewards at time $t + 2$, and so on. Obviously, we cannot know the rewards in the future as we cannot calculate it, but that's where neural networks come in, which will be explained later.

What determines how the agent acts – in this case how the snake moves - is known as a policy, which will measure 'how good' it is for an agent to take an action from a given state. A policy employs a value function, which is a function of state-action pairs that estimate the cumulative future reward from taking that action from that state. The function mapping actions to these values is known as the Q function, and it's return the Q value (Q for quality). The optimal Q function, which provides maximum rewards is denoted Q^* . The critical property of Q^* is that it must satisfy the following equation:

$$Q^*(s, a) = R_{t+1} + \gamma \cdot \max_{a'} Q^*(s', a') \quad \begin{array}{l} s, s' \in S \\ a, a' \in A \end{array}$$

This equation is known as the bellman optimality equation. In English, it says that for a state-action pair (s, a) , the expected return on taking action a from state s is equal to the

immediate reward R_{t+1} , which will be a known value, plus the maximum possible discounted reward from taking the next action a' from the next state s' . This means that the Q^* function should ideally, for each state-action pair, return the maximum possible discounted future reward for taking that action from that state: The Q-value. The process for finding the Q^* function, and hence these Q-values, is the appropriately named Q-learning algorithm. Note the recursive nature of this Q^* function, it calls itself.

The Q-learning algorithm is iterative, updating the Q-values for each state-action pair using this bellman optimality equation until the values (hopefully) converge to their optimum, Q^* . For a simpler situation with only a handful of states, the state-action pairs can be stored in a matrix, known as a Q-table, where the dimensions of the table are number of actions x number of states, and each entry in the matrix represents the Q-value of taking that action from that state.

At each time step, the agent selects the action from the table with the highest value for the state it is currently in, and gains a reward based upon that action, as defined in our MDP. The Q-table entry is then updated using the bellman optimality equation $Q^*(s, a) = R_{t+1} + \gamma \cdot \max Q^*(s', a')$, where R_{t+1} is the reward just given, and $\max Q^*(s', a')$ is the maximum reward available from taking the optimal action at the state just moved to - obtained from the Q-table. As the agent starts out it won't be very good as the Q-values are usually all initialised to random numbers, so the actions it's taking are also random. But as the agent navigates the environment and works out what actions yield high rewards from what states, and what actions yield negative rewards (punishments), the values in the Q-table converge to their optima.

A table storing Q-values is good for environments with a small number of states, but when that number of states gets very large, a Q-table is no longer practical, which is where neural networks come in.

2.4 Neural Networks

A neural network is made up of several layers of perceptrons, or 'neurons', each of which may be modelled as follows:

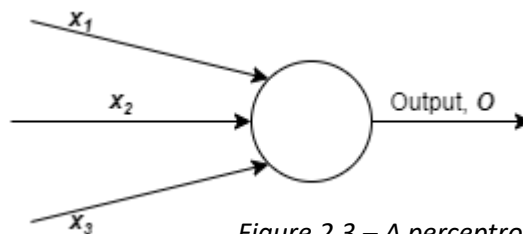


Figure 2.3 – A perceptron

x_1, x_2, x_3 are numerical floating-point inputs, output O is also a float. Each input also has a weight w_i associated with it, which represents the importance of that input to the output of the neuron. The neuron then computes the sum of all the inputs times their weights: $\sum w_i x_i$. This could be simplified if we represent the weights and inputs as two single vectors w and x respectively, making our sum simply the scalar product of the two: $w \cdot x$. The neuron also has another parameter, the bias b . This is another number that adjusts the threshold of the

neuron, making it easier or harder for the neuron to activate. Including the bias in our sum makes it $w \cdot x + b$. However, it is not this sum that is output by the neuron, it is first passed to a function known as an activation function, some common ones being the sigmoid function $\sigma(x) = (1 + e^{-x})^{-1}$ which looks like this:

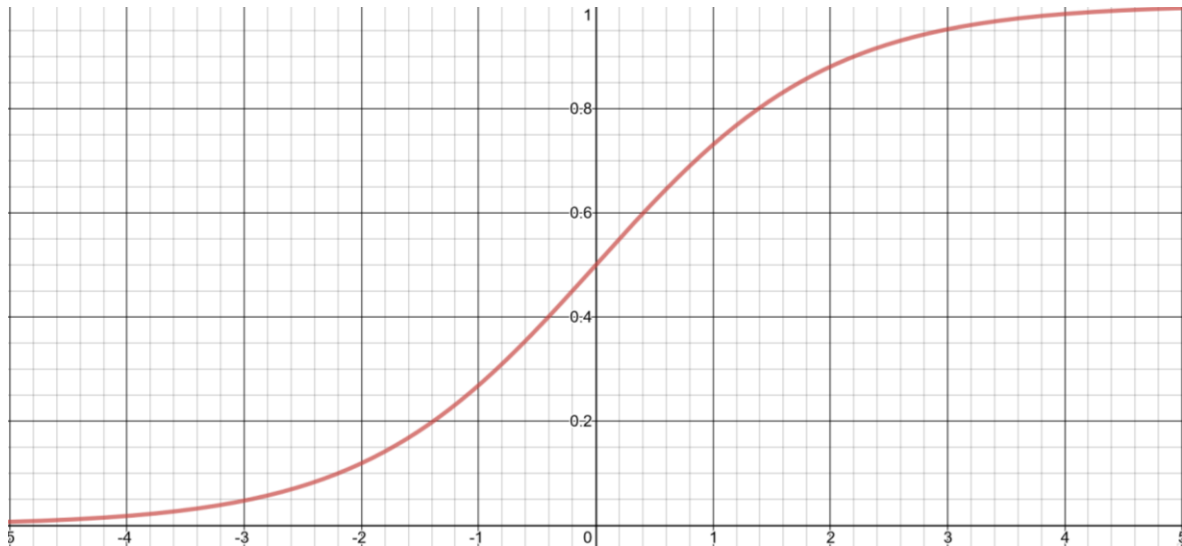


Figure 2.3 – The sigmoid function

And the rectified linear unit function, or ReLU, which is simply $\text{ReLU}(x) = \max(0, x)$. It looks like this:

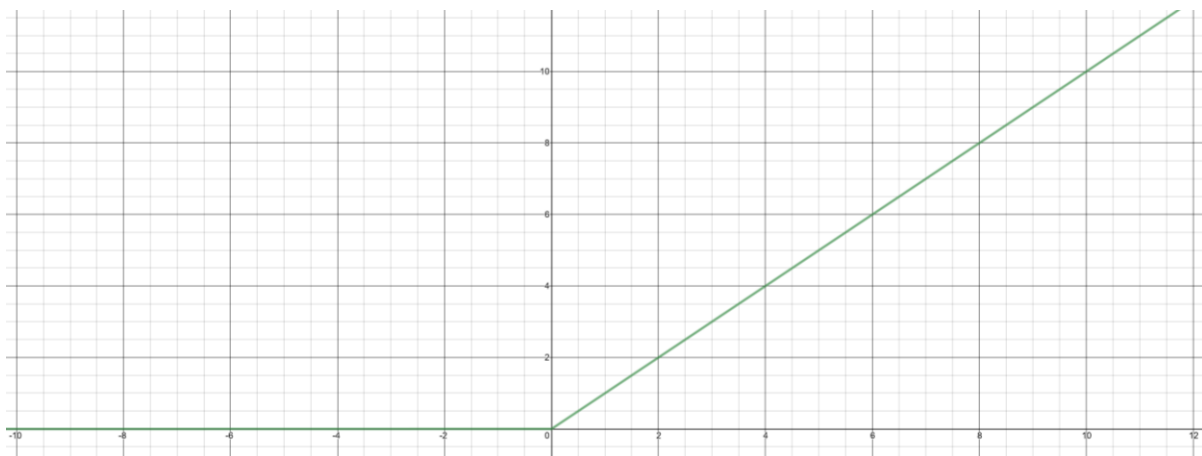


Figure 2.4 – The ReLU function

The reason for these activation functions is to make sure that small changes in the input do not have dramatic effects on output. The advantages and disadvantages of various activation functions may be debated, but for our purposes I will focus on ReLU as it is simple, and what is used in my code. The final output O of a ReLU neuron is.

$$O = \text{ReLU}(w \cdot x + b)$$

One neuron alone does not make a neural network. Many of them are connected in layers like so:

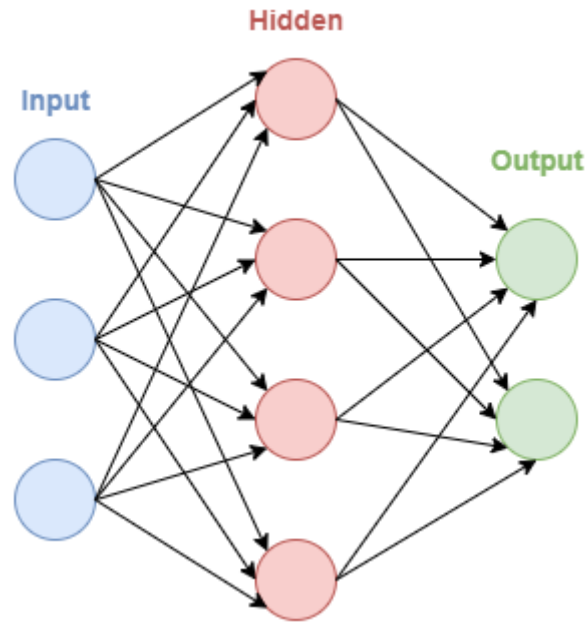


Figure 2.5 – A fully-connected, linear neural network.

Each neuron in the previous layer passes its output to each neuron the next layer as an input, with each of those connections having a weight associated with it. This multi-layer model allows networks to make quite complex decisions, as multiple layers break down the decision-making process. Deciding how many layers a network should have, how many neurons per layer, and what activation function to use is all part of building a network. These variables are known as hyperparameters, as they are programmer-defined and not computed as part of training.

Another hyperparameter is the cost, or loss, function. The loss function of a network computes a numerical difference between the actual output of the network, and what we want the output of the network to be. For example, the mean squared error (MSE) loss function calculates the mean of the squared error of the output of every neuron in the output layer.

$$MSE = \frac{1}{n} \sum_{i=0}^n (x_i - \hat{x}_i)^2$$

Training a neural network is the process of deciding what weights and biases should be associated with each connection, and is an incredibly complex processing involving lots of calculus which is far beyond the scope of this project. In simple terms, the partial derivative of the cost function with respect to every parameter in the network is computed by an algorithm known as backpropagation, which uses the chain rule to compute the rate of change of the cost function with respect to every weight and bias in the network. These derivatives tell us how the cost function, and therefore the network's output changes as the parameters of the network change, and therefore can be used to alter the weights and biases to minimise the cost function using a gradient descent algorithm.

Gradient descent uses the values computed by backpropagation to minimise the cost function iteratively: steps are run at fixed intervals when training the network, and the amount by which they change the weights - the learning rate - is also defined when designing the network. These are two more examples of hyperparameters.

Many gradient descent steps are required, ranging from a few hundred to in the millions for more complex networks to achieve any degree of accuracy, and all the calculations required for a neural network and gradient descent are very computationally expensive, which is why training can take a long time.

Obviously, this is a simplified explanation of the basics of how a neural network functions: there are many more low-level optimisations in neural network engines such as TensorFlow which help improve performance, but the overview given here provides enough understanding to be able to work with neural networks using a high-level API like Keras.

2.5 Deep Q-Networks

The way neural networks come into Q-learning is that they can be used to approximate the Q-values discussed earlier. Instead of a Q-table storing the value for each action from each state, a network is used where the input vector describes the state of the environment, and the output vector gives an estimated Q-value for each possible action from that state (recall that Q-values give the maximum possible discounted future reward of taking a given action from a given state). This kind of network is termed a Deep Q-Network, or DQN

The DQN will need a cost function before it can work. The output of the network will be the predicted Q-value, but we need an optimal Q-value to compare this with before we can calculate the error, or loss, in the network. Taking the squared error of these two values gives $Cost = (Q - Q^*)^2$, where Q^* is the optimal Q-value

As the optimal Q-value is not a fixed target - we don't actually know it - it would also have to be predicted by the network. This causes an issue, as the network ends up trying to optimise itself to a target that is a) constantly changing and b) predicted by itself, so we end up with the network effectively chasing its own tail, causing the training to be very unstable and rarely result in convergence. There is a solution to this, a technique known as a Double Deep-Q learning, where two separate networks are used: one 'active' network which is constantly updated and used for training and controlling the agent, and one 'target' network which is used solely for predicting these optimal Q-values, with every so often the target net being updated to mirror the active net. This helps reduce the issue of constantly changing targets by providing semi-constant optimal values for the active net to work towards, and when the active net gets close to those values, they are re-calculated to be more (hopefully) more optimal.

There are a few more tricks we can use to increase the accuracy of the DQN. The first is experience replay. Instead of training the network on each action as it happens, each 'experience' is stored, and these experiences randomly sampled from each time we want to run a gradient descent step. The experiences are stored as a tuple describing what happened at each time step and giving the network the data it needs to train. The random

sampling of actions helps decorrelate the training so that it is not training in the cycles that tend to happen in games, as this can destabilise training and reduce the accuracy of the network. The experiences are stored in a queue of fixed size called replay memory, where as new data is added, old data leaves. This ensures that the data in the replay memory represents the most recent experiences, which will hopefully be of a higher quality, i.e. yield more reward, as the network has trained more.

The second technique is known as 'epsilon greedy', which helps resolve the common issue of exploration vs exploitation. When the agent is learning it may quickly come across what it believes to be an optimal strategy, and continue to exploit this strategy, unaware that a more optimal strategy is available as it has not explored enough. To solve this issue, a parameter ϵ is used to introduce some randomness into the agent's actions to help it explore more unknown strategies it may not have explored on its own. Each time the agent goes to take an action, it has probability $(1 - \epsilon)$ of using the neural network to choose an action (exploitation), or has probability ϵ of choosing an action randomly (exploration). The parameter decays over time as the agent is trained- it is assumed that optimal policy is converged towards with more training, so exploration is no longer needed as exploitation should always yield the better results.

With all these techniques, this makes the final training algorithm look something like this:

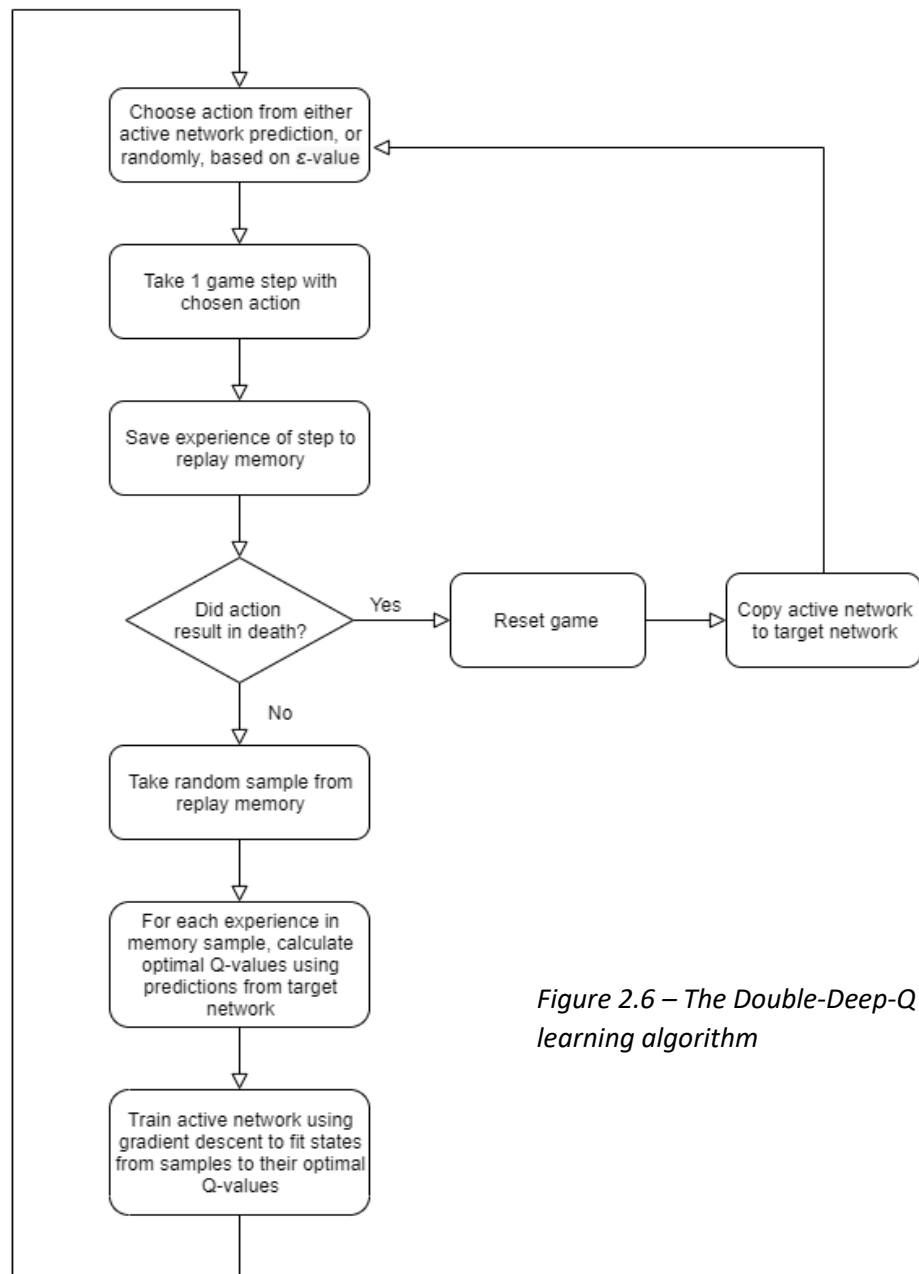


Figure 2.6 – The Double-Deep-Q learning algorithm

2.6 Network Architecture

There are many ways in which a DQN could be built to play snake. One of the most common, and probably best way involves a convolutional neural network, the kind of network that Nvidia and Deepmind used for their autonomous vehicle and game AI. The entire game grid would be passed to the network as a matrix flattened to a vector and the network would use that as it's environment state, and then train on that. Convolutional neural networks are more complex to design and implement, take longer to train, and I don't fully understand how they work. For these reasons, a simple, linear neural network will be used.

As a simpler network is being used, using the entire game grid as the input vector is impractical because the network would not be complex enough to process the data. Instead, specific information about the environment will be provided to the network.

There are a few ways in which information could be extracted from the environment, however I decided to go with the following: x-y vectors will be used to give the snake's direction, and the position of the apple relative to the snake. 3 more values will also be used to indicate what is to the immediate left, right, or front of the snake, and set to negative one to indicate an obstacle, zero for nothing, and one for an apple. The length of the snake's tail will also be given.

This makes the total size of the network's input 7: obstacle left, obstacle right, obstacle front, position x, position y, direction x, direction y, tail length.

The output layer of the network will be of size 3, with each value being the estimated Q-value for going forward, left or right. 1 hidden layer of 12 neurons will be used. The activation function for this layer will be the ReLU function already discussed, and TensorFlow's built in 'Adam' optimiser for gradient descent.

2.7 Development Tools

For machine learning, Python is the industry standard for many reasons:

- Many libraries that make extending the language easy
- Ability to wrap C/C++ code for performance
- Ease of use

All the functionality being discussed obviously doesn't come built in as part of Python's standard library, so a few third-party libraries will be relied on. For programming the game, Pygame is the obvious choice as it provides a rich but simple API for creating game graphics that I am already very familiar with, having used it in the past.

For the neural network, Google's open-source TensorFlow library will be used. TensorFlow provides all the backend necessary for building and training neural networks, and as such is used widely within industry. Much of TensorFlow is written in C/C++ and Nvidia CUDA, making it highly optimised and much faster than raw Python.

CUDA is a compute platform written by NVidia for parallel GPU computation. As already discussed, the nature of the neural networks lends itself nicely to parallel computation, and Tensorflow can integrate with CUDA to use a GPU as an accelerator. cuDNN is the CUDA deep neural network library that Tensorflow utilises for this.

TensorFlow comes bundled with Keras, a high-level machine-learning API designed to make working with neural networks simple. It is widely used by both hobbyists and in industry because it is easy to use, but can be very powerful where necessary. Keras can be installed standalone and used with multiple machine learning libraries as backends, but the version of Keras that comes bundled with TensorFlow (`tf.keras`) provides seamless integration

between the two so Keras can be used for most purposes, but pure TensorFlow code can be dropped in where necessary.

Much numerical processing is also required, so NumPy has been used as it provides many mathematical functions, and NumPy arrays are also much faster than raw Python because it is written mostly in C. NumPy is used by TensorFlow and Keras for backend processing, so it is necessary for use with those libraries anyway.

For clarity, the exact versions of all relevant libraries and software tools used are given below. Development and testing were done across two machines, the details of which are also below. This information is relevant to the discussion of compute performance later on. The laptop GPU was not utilised for compute as CUDA is a proprietary NVidia platform, and as such is not available on Intel GPUs.

	Laptop	Desktop
CPU	Intel i5-8265U 4 cores, 8 threads 1.6GHz base clock 3.9GHz boost clock 6 MB cache	Intel i5-7500 4 cores, 4 threads 3.4 GHz base clock 3.8 GHz boost clock 6 MB cache
GPU	Intel integrated – UHD 620	Nvidia RTX 2060 Super
RAM	8GB	16 GB
OS	Windows 10 1909	Windows 10 1909
Python version	3.6.8	3.6.8
Tensorflow version	1.15.0	1.15.0
CUDA version	-	10.0
cuDNN version	-	7.4.2

2.8 CLI design

An interface is needed for the user to select how to run the program. Implementing a GUI for just this purpose would incur an unnecessary amount of performance overhead, and would be overkill for just choosing options. As such, a simple CLI will be implemented, with command line flags indicating what the program should do. Plus, the command line is superior anyway.

Command line arguments are passed when running the program from the command line in the following way:

```
> python ai-snake.py <option> [arguments]
```

One of five options will need to be selected:

Option	Flag	Arguments	Purpose
Train	-t	Number of games to train on, default 1000	Train a new model and save it to a file

Play	-p	None	Play the game the old-fashioned way with a keyboard
Run	-r	Number of games to run, defaults to 1	Run a game using a trained model
Leaderboard	-l	Player who's scores to view, defaults to all	View the leaderboard of either all-time, or a specific player
Debug	-d	None	Launches the program in an interactive console and calls no functions, useful for debugging

This interface is simple, but also covers all the functionality of the code, and the extra arguments allow a degree of flexibility, making it very powerful. It also allows for easy extensibility should any more functionality needed to be added to the interface in future.

2.9 Leaderboard

The leaderboard will be stored on disk as a json file, the reasons for which are outlined in the table below.

Solution/file type	Advantages	Disadvantages
Plain Text	Easily human readable	Difficult to parse Requires strict formatting- no inherent formatting rules
XML	Well structured More featured than JSON	Needs parsing Verbose
Python Pickle	Very easy to work with in python	Not human readable Only works with python code
JSON	Well structured Supports multiple data types Easy to work with in python	None applicable to this application
SQL database	Well organised, powerful	Completely overkill for this application

JSON makes the data well-organised and structured, and can convert directly to python objects so is easy to work with.

The leaderboard will store the top 20 scores of all time of players of all names, as well as keep track of the top 10 scores per player. The top-level object will store the 'all' leaderboard, which will be an array of arrays, with each array consisting of the player's name, score and date at which it was achieved. The top-level object will also store each individual leaderboard with the player's name as the key, and value as an array of arrays, each array storing score and date (unlike the 'all' category, name is not needed as all scores are for the same player). An example JSON file is shown in figure 2.7.

```

1  {
2      "all": [["ai", 49, "13/11/19, 09:18"],
3              ["steve", 39, "01/10/19, 17:49"],
4              ["joe", 21, "01/10/19, 17:40"]],
5      "ai": [[49, "13/11/19, 09:18"],
6              [41, "01/10/19, 17:19"]],
7      "joe": [[21, "01/10/19, 17:40"],
8              [17, "02/02/20, 13:42"]]
9  }

```

Figure 2.7 – Example leaderboard file

The code to handle modifying the leaderboard will have to be carefully written and well-tested to ensure that this format is properly maintained, with scores being added to the necessary sections and added/removed appropriately. This will pay off however, as it will mean the code will need to do little processing in sorting the leaderboard or scores, as it will already be sorted and organised.

2.10 Class/Data Tables

Grid

Provides the interface with the grid, and handles all the drawing/rendering logic. Other classes can call methods of this class to draw onto the game grid.

Methods

Name	Parameters	Return Value	Explanation
init	Pygame display object, square size of the grid as an integer	None	The constructor method – initialises attributes and provides an initial call to draw the gridlines onto the blank window passed to it.
Draw	None	None	Subroutine to draw the lines grid onto the pygame window.
Update	None	None	Subroutine to draw the coloured squares onto the grid from the matrix storing their locations.
Clear	None	None	Subroutine to reinitialise the matrix back to all zeroes.
Set square	Tuple representing a coordinate pair (x, y), Integer value to be set	None	Sets the value in the matrix at (x, y) equal to the integer value passed. Different integer values represent different objects i.e. 2 is an apple, 1 is snake, and also represents what colour to set them to.
Get Val	Tuple representing coordinate pair	Integer value	Returns the value at that coordinate in the matrix, -1 if the coordinate is out of range.

Attributes

Name	Type	Explanation
Rows	Int	The number of rows and columns in the grid as an integer, used when calculating positions for drawing gridlines and coloured squares onto pygame window.
Matrix	NumPy 2d array of ints	The grid values as an NxN matrix, represents the game board.
Display	Pygame display object	Pygame display object for drawing onto.
Screen Size	Int	The square size of the pygame display window in pixels, used when calculating position of lines and rectangles drawn onto display.

Snake

Represents the snake player. All information about the snake's position, as well as methods to draw and move it are encapsulated here.

Methods

Name	Parameters	Return Value	Explanation
init	Grid object	None	Constructor class: initialises attributes.
Draw	None	None	Draws the snake head and body onto the grid by setting the relevant matrix values through calls to grid.setsq().
Move	Direction as a unit vector (x, y)	None	Moves the snake forward by one square in the direction passed.

Attributes

Name	Type	Explanation
Grid	Grid object	For the snake to interact with to move around and draw itself.
Position	Tuple (x, y)	Coordinate pair representing head position.
Tail Len	Int	The length of the snake's tail, also the length of the Body list.
Body	Python List	Stores the coordinates of all the snake's body pieces.

Apple

Represents the apple. All information about the apple's position, as well as methods to draw and randomly reposition it are encapsulated here.

Methods

Name	Parameters	Return Value	Explanation
init	Grid object	None	Constructor method: initialises attributes and initially positions apple randomly through call to <code>reposition()</code> .
Reposition	None	None	Randomly calculates apple position.
Draw	None	None	Draws apple onto game board through call to <code>grid.setsq()</code> .

Attributes

Name	Type	Explanation
Grid	Grid object	For the apple to interact with to draw and move itself.
Position	Tuple (x, y)	Coordinate pair representing position.

Game

The main game class. Contains instances of all the components of the game: snake, apple, grid. Instead of running the game in a loop, the `step()` method moves the game forward by one step, which makes it easier for the AI to deal with the game one step at a time.

Methods

Name	Parameters	Return Value	Explanation
init	Number of grid rows and columns as an int, screen size in pixels as an int	None	Constructor method: initialises attributes, initialises pygame module and functionality, provides initial call to <code>new_game()</code> .
Get state	None	Tuple representing state of game environment	Calculates if there are any obstacles surrounding the snake's head, the direction vector to the apple from the snake's head, and the snake's direction, and returns all this information as a tuple which is used to train the neural network.
Draw score	None	None	Draws the text onto the pygame window to show the player their current score.
New game	None	None	Initialises new game by resetting score, clearing grid etc. Creates new instances of Apple and Snake classes, and draws them onto grid.
Step	Integer representing what action to take.	Tuple with values representing consequences of game step	Progresses game by one time step. Moves snake, does collision detection, redraws and updates display. Returns tuple that says if an apple was eaten, if the snake died, and the current score.

Attributes

Name	Type	Explanation
Display	Pygame display object	The object representing the window so the game can draw onto it.
Font	Pygame font engine object	Pygame needs a separate object for drawing text onto windows, the score is drawn as characters onto the window through this object.
Grid	Grid object	Represents the game board on which the game is played.
Score	Int	The player's current score.
Done	Bool	Is the player dead?.
Direction	Tuple (x, y)	Current direction the snake is pointing in.
Prev Direction	Tuple (x, y)	The direction the snake was pointing in at the last time step.
Snake	Snake object	Object representing the snake through which the snake is moved around and drawn.
Apple	Apple object	Object representing the apple through which the apple is drawn and positioned.

Play Game (inherits from Game)

Adds a clock to the game class so it maintains a constant framerate and has a delay between game steps.

Methods

Name	Parameters	Return Value	Explanation
init	Number of grid rows and columns as an int, screen size in pixels as an int, framerate as an int	None	Constructor method: calls the constructor of the superclass and also initialises speed attribute.
New Game	None	None	Calls super new game function but also initialises pygame clock object so the game runs at a set framerate.
Step	Action	Tuple representing environment state	Calls super action function and returns same values as it, but also adds a delay provided by a call to clock.tick() so the game does not run as fast as the computer will run it, so that it can be played by a human.

Attributes

Name	Type	Explanation
Speed	Int	The speed at which the game runs in frames per second
Clock	Pygame clock object	The object which controls framerate and provides delay

Leaderboard

Contains all the functionality required for dealing with the leaderboard.

Methods

Name	Parameters	Return Value	Explanation
init	Filename of leaderboard file as a string	None	Constructor method: opens the json file passed as a parameter and loads the file into memory as a python dict. If the file does not exist, it creates one.
Test	Name of person and their score	Boolean	Tests to see if a score would be inserted onto the scoreboard or if it should be discarded. Returns True if the score should be discarded as it does not beat a top score already on the board.
Add	Name of person and their score	None	Adds a score to the leaderboard loaded into memory.
Display	Name of player to print scores for, "all" by default	None	Displays the scores for the player passed as an argument in descending order.
Save	None	None	Saves the python dict in memory back to disk in json format.

Attributes

Name	Type	Explanation
Filename	Str	The name of the JSON file on disk
Scores	Dict	The scoreboard saved in memory as a dict.

DQPlayer

Runs the game with a neural network loaded from file to control the snake.

Methods

Name	Parameters	Return Value	Explanation
init	Filename of saved model	None	Constructor method: loads the saved neural network into memory, also initialises leaderboard for saving scores.
Act	Tuple representing	Action for the snake to	Passes the given tuple to the neural network to predict what action to take based upon its.

	environment state	take an int either 0,1 or 2	environment. Return what action the neural network decides to take.
Play	Number of games to play, max steps (default 5000)	List of scores and steps per episode for data analysis	Runs the game. Initialises an instance of play game class, and runs a loop getting environment state, calculating best action using act() and taking action at each time step by interacting with game class. Runs a fixed number of games which is passed as parameter.

Attributes

Name	Type	Explanation
Net	Keras network object	The neural network.
Leaderboard	Leaderboard object	For saving the AIs scores to the leaderboard.
Loop Threshold	Int	Maximum steps the snake can take without scoring before the game intervenes, to stop it infinitely looping.

Human Player

This class runs the game in such a way that it can be played by a human

Methods

Name	Parameters	Return Value	Explanation
init	None	None	Constructor method: initialises game object as attribute.
Game over	None	None	Called when the player dies, displays a game over message and saves score to leaderboard.
Play	None	None	Runs the main game loop, taking user keyboard input and acting upon it at each time step.

Attributes

Name	Type	Explanation
Game	Play game object	The game object which contains most of the functionality for actually playing the game.
Score	Int	The current score.

DQTrainer

This class contains all the functionality for training the neural network and running the double deep Q learning algorithm. Most of the attributes here are hyperparameters for the

Q-learning algorithm. Functionality is also included to load a network from a file and continue training it.

Methods

Name	Parameters	Return Value	Explanation
init	None	None	Constructor method: initialises all the attributes, mostly hyperparameters for the ML algorithm, and builds the two neural networks, saving them as attributes also.
Build net	None	Tensorflow neural network object	Builds a neural network with a set architecture and returns it.
Update target	None	None	Copy the weights of the active network to the target network.
Remember	Tuple of the experience of one game step	None	Add an experience to the replay memory so it can be later used for training. Also decay the epsilon value.
Act	Environment state tuple	Int representing action to take	Based on the current epsilon value, either take a random action, or pass the tuple to the neural network that then decides the action to take.
Learn	None	None	Take a random sample from the replay memory and use it to train the active network to fit the target Q values calculated by the target net.
Train	Number of training episodes to run, and max steps per episode as ints	List of scores and steps at each episode for data analysis	Run the training algorithm, calling the learn() function at each time step to train the model, and update target() at each episode to update the target network. Initialises an instance of the Game class for use as the learning agent's environment.

Attributes

Name	Type	Explanation
State size	int	The size of the vector/tuple representing the environment state.
N actions	Int	The number of different actions the snake can take.
Gamma	Float	Discount factor for Q-learning.
Epsilon	Float	For epsilon-greedy exploitation vs exploration.
E decay	Float	Rate at which epsilon exponentially decays.
Batch size	Int	Number of items to sample from replay memory each time learn() is called.
Min mem length	Int	Minimum length of replay memory for training to start.
Replay mem	Python deque	The experience replay memory.

Active net	Keras Neural Network Object	The active neural network for training and taking actions.
Target net	Keras Neural Network Object	Target neural network for computing optimal Q* values.

An entity relationship diagram showing the interactions between classes is shown in figure 2.8. Most of the class interactions are through composition, meaning that the instances of the child classes are created inside and belong to the instances of parent classes. The exception to this being the aggregation between the snake, grid and apple classes where the instance of the grid object belongs to the game object, but a reference to this instance is stored within the snake and apple classes such that they may interact directly with the grid.

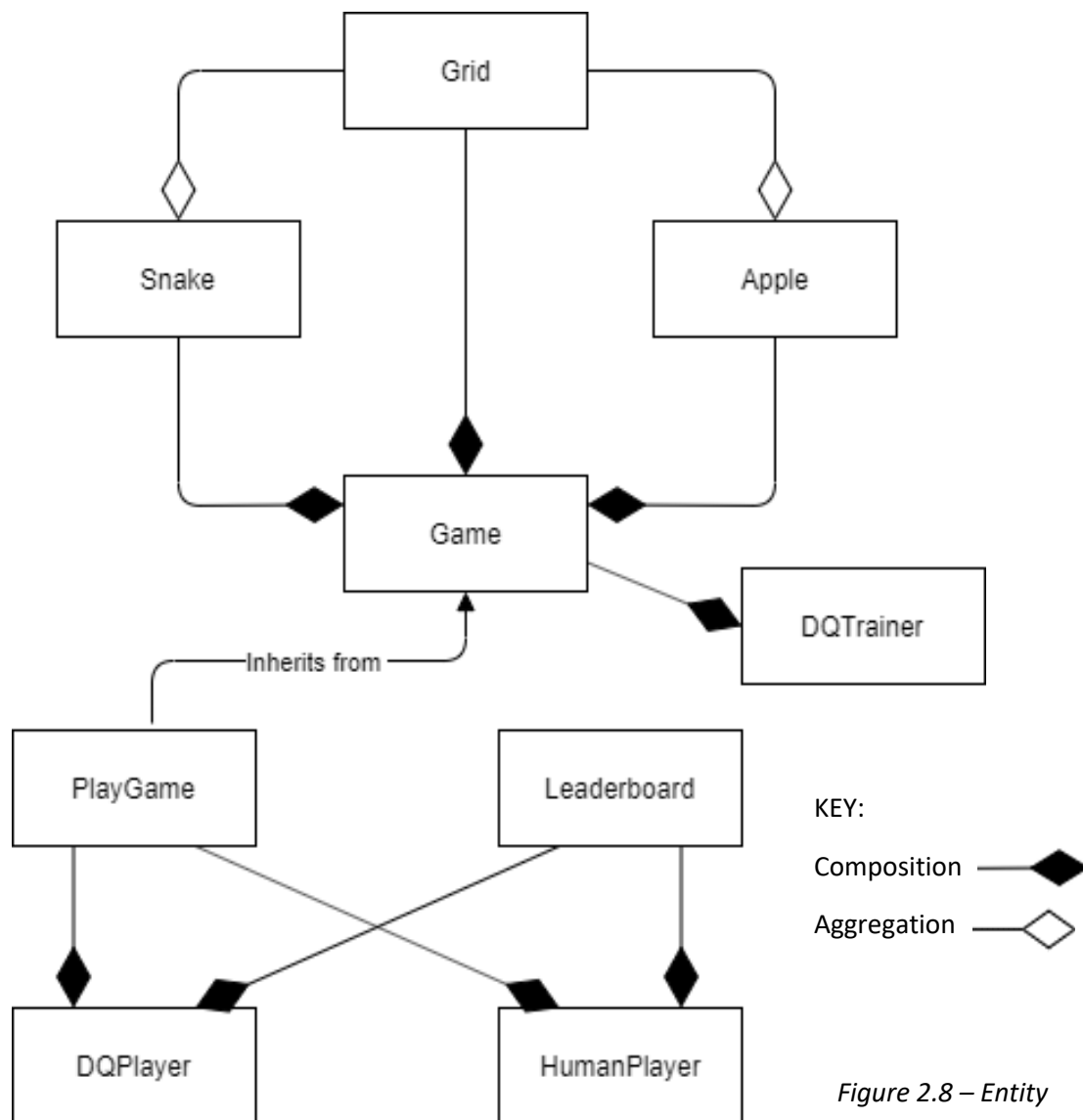


Figure 2.8 – Entity relationship diagram

3 Code Listing

This section of the report contains the Python code for the ai-snake.py file. This is the only code file that is a part of the solution, and covers all functionality required. The code is separated into sections, each individually screenshotted and annotated with more detailed explanation, on top of code comments giving more succinct explanation where appropriate. Some screenshots are shown with sections collapsed, so that entire classes can be shown in one image. The collapsed sections are also given in full, but in separate screenshots with their own annotations. Line numbers are included in screenshots for clarity. The full code is given unannotated and uncommented in Appendix C, for reference.

```
1  from os import environ
2  environ["PYGAME_HIDE_SUPPORT_PROMPT"] = "hide"
3  environ["CUDA_VISIBLE_DEVICES"] = "-1"
4  environ["TF_CPP_MIN_LOG_LEVEL"] = '2'
5
6  import pygame                #graphics and input
7  import numpy as np           #arrays and numerical processing
8  from tensorflow import keras #AI
9  from collections import deque #replay memory data structure
10 from datetime import datetime #timestamping
11 from time import sleep        #delays
12 import json                   #leaderboard file handling
13 import pickle as pkl          #saving python data to be loaded into other scripts
14
```

Imports and environment variables. The three environment variables are needed to be set before the libraries they relate to are imported, hence why they are above the rest of the imports. Line 2 is to hide the pygame welcome message that is usually printed when importing pygame, line 3 is to tell Tensorflow I have no GPU set up so that the code runs on the CPU, and line 4 is to stop Tensorflow printing all the debug information, and only print errors. Deprecation warnings cannot be muted, hence why some info is still printed when running the code.

```
15  SCREEN_SIZE = 600
16  ROWS = 30
17  FONT = "comicsansms"
18  FONT_SIZE = 18
19  APPLE_REWARD = 100
20  DEATH_PENTALTY = -10
21  HUMAN_SPEED = 10
22  AI_SPEED = 100
23  DISCOUNT_FACTOR = 0.995
24  L_FILE = "leaderboard.json"
25  M_FILE = "AI-final.h5"
26  M_LEFT = ((0, 1),
27            | (-1, 0))
28  M_RIGHT = ((0, -1),
29            | (1, 0))
30
```

These are the global constants used by the code. Some are at the top so they can be easily edited without searching for them, and some because they are used in multiple places. All the values seen are the values used in testing.

The last two are the matrix constants for rotating vectors. Matrices can be used to define transformations, and these two define rotations by 90° left and right respectively. They are backwards to the standard rotation matrices, but that is because pygame uses a backwards y-axis, as shown in figure 3.1. These are used for calculating vectors perpendicular to the direction vector of the snake throughout the code.

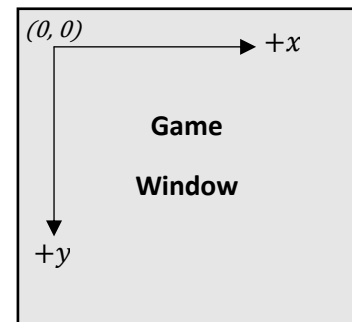


Figure 3.1 – Pygame axes

```

32 class Grid:
33     def __init__(self, display, rows):
34         self.rows = rows
35         self.matrix = np.zeros((self.rows, self.rows)) #grid data structure
36
37         self.display = display #pygame display object
38         self.screen_size = self.display.get_width()
39         self.draw()
40
41     def draw(self):
42         for i in range(self.rows):
43             start = (self.screen_size / self.rows * i, 0) #start point of line
44             end = (self.screen_size / self.rows * i, self.screen_size) #end point
45             pygame.draw.line(self.display, (0, 0, 0), start, end) #draw
46         for i in range(self.rows):
47             start = (0, self.screen_size / self.rows * i)
48             end = (self.screen_size, self.screen_size / self.rows * i)
49             pygame.draw.line(self.display, (0, 0, 0), start, end)
50
51     def update(self):
52         #render the grid values onto the screen as squares
53         for i in range(self.rows):
54             for j in range(self.rows):
55                 colour = self.matrix[i][j]
56                 if colour == 0: # empty
57                     colour = (255, 255, 255)
58                 elif colour == 1: # body
59                     colour = (0, 255, 0)
60                 elif colour == 2: # head
61                     colour = (0, 128, 0)
62                 elif colour == 4: # apple
63                     colour = (255, 0, 0)
64                 pygame.draw.rect(self.display, colour,
65                                 (self.screen_size / self.rows * i + 1, #position(x)
66                                 self.screen_size / self.rows * j + 1, #position(y)
67                                 self.screen_size / self.rows - 1,      #width
68                                 self.screen_size / self.rows - 1))      #height
69
70     def clear(self):
71         #reset grid back to all zeros
72         self.matrix = np.zeros((self.rows, self.rows))
73
74     def setsq(self, xy, val):
75         #setter function for grid values
76         self.matrix[xy] = val
77
78     def getval(self, xy):
79         #getter function for grid values
80         if (xy[0] < 0) or (xy[1] < 0) or (xy[0] > self.rows - 1) or (xy[1] > self.rows - 1):
81             return -1 #if out of range
82         else:
83             return int(self.matrix[xy])
84

```

This is the grid class. Most of it is self-explanatory and there isn't much complicated going on here, except for the draw method:

```
41     def draw(self):
42         for i in range(self.rows):
43             start = (self.screen_size / self.rows * i, 0) #start point of line
44             end = (self.screen_size / self.rows * i, self.screen_size) #end point
45             pygame.draw.line(self.display, (0, 0, 0), start, end) #draw
46         for i in range(self.rows):
47             start = (0, self.screen_size / self.rows * i)
48             end = (self.screen_size, self.screen_size / self.rows * i)
49             pygame.draw.line(self.display, (0, 0, 0), start, end)
50
```

This function is responsible for drawing the lines of the grid onto the game window. The first loop iterates over every vertical row of the grid, calculating the start and end points of each line using the values for the size of the screen and the number of rows passed at instantiation. The second loop does the same, but horizontally. The point of making these calculations at runtime is so that any value can be given for the number of rows or screen size and the program can draw the grid as required. The same principal applies to lines 64-68, the pygame API call that draws the coloured squares onto the screen.

```
78 class Snake:
79     def __init__(self, grid):
80         self.grid = grid # grid object
81         self.pos = (int(grid.rows / 2), int(grid.rows / 2)) # head coordinate
82         self.tail_len = 3
83         self.body = [] # coords of body squares
84
85     def draw(self):
86         self.grid.setsq(self.pos, 2) # head
87         for i in self.body:
88             self.grid.setsq(i, 1) # body
89
90     def move(self, direction):
91         self.body.append(tuple(self.pos)) # add current position to body
92         self.body = self.body[-(self.tail_len):] # trim body to correct length
93         self.pos = (self.pos[0] + direction[0], self.pos[1] + direction[1]) # move
94
95 class Apple:
96     def __init__(self, grid):
97         self.grid = grid
98         self.pos = (0, 0)
99         self.repos() # randomly position on creation
100
101     def repos(self):
102         # randomly generate coordinates
103         self.pos = (np.random.randint(0, self.grid.rows), np.random.randint(0, self.grid.rows))
104         if self.grid.getval(self.pos) != 0: #if randomly selected coords have snake in it
105             self.repos() #recursion ooooh
106
107     def draw(self):
108         self.grid.setsq(self.pos, 4)
109
110
111
```

The snake and apple classes are also fairly simple, and do not require any further explanation.

```

119 class Game:
120     def __init__(self, rows, screen_size):
121         # initialise pygame, window and fonts and start new game
122         pygame.init()
123         self.display = pygame.display.set_mode((screen_size, screen_size))
124         pygame.display.set_caption("snek")
125         self.display.fill((255, 255, 255))
126
127         pygame.font.init()
128         self.font = pygame.font.SysFont(FONT, FONT_SIZE)
129
130         self.grid = Grid(self.display, rows)
131         self.new_game()
132
133 > def get_state(self): ...
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185 > def draw_score(self):
186     # draw the score onto the screen
187     self.display.blit(self.font.render(str(self.score), True, (0, 0, 0)), (5, -2))
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227

```

The game class is shown with some methods collapsed, individually explained below. The new game method sets up/resets the variables required for each new game, so that multiple games can be run without having to reinstantiate the game class every time the snake dies, lending itself to being run multiple times in a row by an AI.

```

133 def get_state(self):
134     # return state of environment, for AI to use
135     item_left = self.grid.getval((self.snake.pos[0] + self.direction[1], self.snake.pos[1] - self.direction[0]))
136     item_right = self.grid.getval((self.snake.pos[0] - self.direction[1], self.snake.pos[1] + self.direction[0]))
137     item_front = self.grid.getval((self.snake.pos[0] + self.direction[0], self.snake.pos[1] + self.direction[1]))
138
139     if item_left == 0:
140         obstacle_left = 0
141     elif item_left == 4:
142         obstacle_left = 1
143     else:
144         obstacle_left = -1
145
146     if item_right == 0:
147         obstacle_right = 0
148     elif item_right == 4:
149         obstacle_right = 1
150     else:
151         obstacle_right = -1
152
153     if item_front == 0:
154         obstacle_front = 0
155     elif item_front == 4:
156         obstacle_front = 1
157     else:
158         obstacle_front = -1
159
160     # position of apple relative to snake x and y
161     pos_x = (self.snake.pos[0] - self.apple.pos[0])
162     pos_y = (self.apple.pos[1] - self.snake.pos[1])
163
164     return (obstacle_left, obstacle_right, obstacle_front, pos_x, pos_y, self.direction[0], -self.direction[1], self.snake.tail_len)
165

```

The get state method calculates all the information needed to describe the state of the environment to the neural network, as defined in the design section, and returns this information as a tuple. Lines 135-137 determine what objects surround the snake's head, and then the conditional expressions in lines 139-158 convert this information to a 0 for empty, 1 apple, -1 for wall or snake tail. Lines 161 and 162 calculate the position vectors of the apple from the snake's head. The reason for the position and direction vectors being negated in the ways they are is because the already mentioned strange coordinate system of pygame. The way it is implemented means that the snake sees the apple in the normal cartesian way, with positive y being up. This isn't really necessary; it just makes debugging less confusing for me.

```

184     def step(self, action):
185         # function to progress the game by one step, based upon action passed to it
186         if action == 0: # carry on
187             self.direction = self.prev_direction
188         elif action == 1: # turn right
189             self.direction = tuple(np.dot(M_RIGHT, self.direction))
190         elif action == 2: # turn left
191             self.direction = tuple(np.dot(M_LEFT, self.direction))
192
193         # change snake direction and move
194         self.snake.move(self.direction)
195         self.prev_direction = self.direction
196
197         # check if snake died based upon the movement taken
198         if self.grid.getval(self.snake.pos) == -1:
199             self.done = True
200         else:
201             for n in self.snake.body:
202                 if n == self.snake.pos:
203                     self.done = True
204
205         # if dead, reset game
206         apple_got = False
207         if self.done:
208             return apple_got, True, self.score
209
210         # if snake at the apple, make a new apple
211         if self.snake.pos == self.apple.pos:
212             self.apple.repos()
213             self.score += 1
214             self.snake.tail_len += 1
215             apple_got = True
216
217         # draw and update everything
218         self.grid.clear()
219         self.snake.draw()
220         self.apple.draw()
221         self.grid.update()
222         self.draw_score()
223         pygame.display.update()
224         return apple_got, self.done, self.score
225

```

The function of the step method is explained by the inline comments. As discussed in design, the step function is designed to be called in a loop so that the AI can also operate within that loop. The return statement gives information about the step taken to the calling function, so that action can be taken based upon the consequences of the step.


```

227 class play_game(Game):
228     def __init__(self, rows, screen_size, speed):
229         super().__init__(rows, screen_size)
230         self.speed = speed
231
232     def new_game(self):
233         super().new_game()
234         self.clock = pygame.time.Clock()
235
236     def step(self, action):
237         a = super().step(action)
238         self.clock.tick(self.speed)
239         return a
240

```

The play game class, that inherits from game. Just wraps a clock around the game class to control speed, as already explained.

```

242 class Leaderboard:
243     def __init__(self, filename):
244         self.filename = filename
245         try: # try to open leaderboard
246             with open(self.filename, "r") as f:
247                 self.scores = json.load(f)
248         except FileNotFoundError: # if doesnt exist, create it, then open it
249             with open(self.filename, "w") as f:
250                 json.dump({"all": []}, f)
251             with open(self.filename, "r") as f:
252                 self.scores = json.load(f)
253
254     def test(self, name, score):
255         # returns True if score worth adding
256         if name not in self.scores: # create them if they dont exist
257             return True
258         elif len(self.scores["all"]) < 20: # if less than 20 scores exist add it
259             return True
260         elif len(self.scores[name]) < 10: # if less than 10 scores exist for them add it
261             return True
262         elif self.scores["all"][19][1] < score: # if the score is higher than the bottom of the leaderboard add it
263             return True
264         elif self.scores[name][9][0] < score: # if the score is higher than the bottom of their leaderboard add it
265             return True
266         else:
267             return False # else just dont bother
268
269 > def add(self, name, score):...
288
289 > def display(self, name="all"):...
304
305     def save(self):
306         with open(self.filename, "w") as f:
307             json.dump(self.scores, f)
308

```

The leaderboard class, shown with some methods collapsed. Upon instantiation, the leaderboard file is opened and saved into memory as a python dict by the `json.load()` function, which converts JSON to python objects. The test method is a helper function used to check if a score would go onto a leaderboard, tested by various conditional statements explained by the inline comments. This method is called by the other methods of the class. The save function is not called automatically, and must be called to commit the changes in memory back to disk.

```

260     def add(self, name, score):
261         name = name.lower()
262         if name == "all":
263             raise ValueError("name cant be 'all'")
264         # add to personal scores list
265         if self.test(name, score):
266             time = datetime.now().strftime("%d/%m/%y, %H:%M ")
267             if name in self.scores:
268                 self.scores[name].append((score, time)) # add
269                 self.scores[name].sort(key=lambda x: x[0], reverse=True) # sort
270                 self.scores[name] = self.scores[name][:10] # chop
271             else:
272                 self.scores[name] = [[score, time]]
273
274         # add to high scores
275         self.scores["all"].append((name, score, time)) # add
276         self.scores["all"].sort(key=lambda x: x[1], reverse=True) # sort
277         self.scores["all"] = self.scores["all"][:20] # chop
278

```

The add function is responsible for making entries to the leaderboard. The score is timestamped by line 266, then appended to the player's scores. The entries are then sorted by score thanks to the anonymous function, and trimmed to only include 10 entries. If the player is new however, the entry can just be added. Lines 275-277 perform the same function, but for the all-time scores. The reason for the strict checks and maintaining the correct format is so that the leaderboard file is always sorted and only contains the data it needs to, making loading the file (and debugging) much simpler.

```

279     def display(self, name="all"):
280         name = name.lower()
281         if name not in self.scores:
282             raise ValueError("That player does not exist")
283         if len(self.scores[name]) == 0:
284             print("No Entries!")
285             return
286         if name == "all":
287             print(" ---- Top Scores All Time ----")
288             for n, s, d in self.scores["all"]:
289                 print("{:<5} - {:<2d} - {}".format(n, s, d))
290         else:
291             print(" ---- Top Scores For " + name.capitalize() + " ----")
292             for s, d in self.scores[name]:
293                 print("{:<2d} - {}".format(s, d))
294

```

The display function prints the leaderboard to the console. If there is an issue with the name, an exception is thrown, and the function exits. If the leaderboard is empty, a message is printed. Different logic is needed for printing the all-time scores and the personal scores, as different information is handled. The format function is utilised for specifying fixed width printing to make the board line up and look good when printed.

```

300 class DQPlayer:
301     def __init__(self, filename):
302         self.net = keras.models.load_model(filename)
303         self.l = Leaderboard(L_FILE)
304         self.loop_threshold = 300
305
306     def act(self, state):
307         Q_vals = self.net.predict(np.array([state]))[0] # pass state to net
308         action = np.argmax(Q_vals) # pick largest Q-val
309         return action, list(Q_vals)
310
311 > def play(self, n_episodes):...
361
362

```

The DQPlayer class which plays the game with a trained neural net. Line 302 loads the saved neural network from file, and line 303 loads the leaderboard. This is done as part of the constructor method, so that scores can be added as the snake plays over multiple games. The act method takes a state and returns an action. The list of Q value is also returned to that the loop detection system can work with them, if necessary.

```

311 def play(self, n_episodes):
312     env = play_game(ROWS, SCREEN_SIZE, AI_SPEED)
313     tracker = []
314     for e in range(n_episodes):
315         dead = False
316         state = env.get_state()
317         steps = 0
318         prev_score = 0
319         steps_since_score = 0
320         second_Qs = []
321         target_Q = None
322 >         while not dead: # main game loop...
349
350             tracker.append((score, steps))
351             print("episode {:<3d} survived {:<4d} steps and scored {:<2d}".format(e, steps, score))
352             env.new_game()
353             self.l.add("AI", score)
354
355             print(str(n_episodes) + " episodes complete")
356
357             tracker = np.array(tracker)
358             score, steps = np.hsplit(tracker, 2)
359             print("max score of {:2d}".format(np.max(score)))
360             print("mean score of {:2.1f}".format(np.mean(score)))
361             print("median score of {:2.1f}".format(np.median(score)))
362             self.l.save()
363

```

Shown here is the play method, with the main game loop collapsed. Lines 315-321 are run every episode/game before it starts. These lines are mainly just initialising variables needed to run the game. Lines 318-321 are variables needed for detecting and dealing with the snake looping. Lines 350-353 are run at the end of every episode, starting a new game and adding the score to leaderboard. Note the use of the list tracker, initialised in line 313, which keeps track of the scores as the game is run so the summary statistics can be calculated in lines 357-361. The list is initialised as a vanilla python list, as NumPy arrays are of a fixed length and cannot be appended onto. It is converted to a NumPy array for faster calculation in line 357. Line 362 is the important function call to write the leaderboard back to disk.

```

322     while not dead: # main game loop
323         pygame.event.pump()
324         action, Q_vals = self.act(state)
325
326         if target_Q in Q_vals:
327             action = Q_vals.index(target_Q)
328             target_Q = None # reset loop detection
329             second_Qs = []
330         _, dead, score = env.step(action)
331
332         if score == prev_score:
333             steps_since_score += 1
334         else:
335             steps_since_score = 0
336         prev_score = score
337
338         state = env.get_state()
339         steps += 1
340
341         if steps_since_score > self.loop_threshold:
342             second_Qs.append(sorted(Q_vals)[1]) # get second highest Q-value
343             if steps_since_score > self.loop_threshold + 100:
344                 target_Q = np.max(second_Qs)
345         if steps_since_score > self.loop_threshold * 5:
346             dead = True
347

```

The pygame call in line 323 is necessary so that the process can continue to respond to OS system calls so that Windows does not think the process is dead and try to kill it. Line 324 calculates the Q-values and determines the best action to take, and line 330 takes that action.

Lines 332-336 keep track of how many steps have been taken since the snake last scored a point, with the purpose of detecting if the snake has got stuck in a loop. Lines 341-345 deal with this by running when the snake goes long enough without scoring, by beginning to log the second-best Q-values, instead of the best, for 100 consecutive steps. After these 100 steps, a target Q-value is selected as the highest Q-value (most favourable action) in that 100 steps. After this value is selected, lines 326-329 are triggered by searching for this value to come up again in the loop the snake is stuck in, and taking the action it corresponds to when it does. The idea behind this is that when the snake calculates that it's optimal action is to spin in an infinite loop, and this infinite loop is broken by selecting another action the snake would not take on its own, but ensuring this action is still one of high quality, i.e. wont (hopefully) instantly kill it. If the snake is still unable to break out of a loop after this, for example, in the case that the loop detection just puts the snake in another loop, the snake is killed after a certain time.

```

364 class HumanPlayer:
365     def __init__(self):
366         self.game = play_game(ROWS, SCREEN_SIZE, HUMAN_SPEED)
367         self.score = 0
368
369     def game_over(self):
370         self.game.display.blit(self.game.font.render("G A M E   O V E R", True, (0, 0, 0)), (230, 220))
371         pygame.display.update()
372         sleep(2)
373         self.game.display.blit(self.game.font.render("Score - " + str(self.score), True, (0, 0, 0)), (255, 250))
374         pygame.display.update()
375         sleep(2)
376         pygame.quit()
377         name = input("enter your name >>>")
378         l = Leaderboard(L_FILE)
379         l.add(name, self.score)
380         l.save()
381
382 > def play(self): ...
412

```

The Human Player class runs the game in a loop, providing a keyboard interface for playing the game. The constructor method initialises the game environment, and then the play method is designed to be called separately. The game over method displays a game over message and the final score on the window when the player dies, and then prompts the user for their score in the console and saves it to the leaderboard.

```

381     def play(self):
382         dead = False
383         keypress = None
384         while not dead:
385             for event in pygame.event.get():
386                 if event.type == pygame.QUIT: # event handling
387                     exit()
388                 if event.type == pygame.KEYDOWN:
389                     if event.key == pygame.K_a:
390                         keypress = (-1, 0)
391                     if event.key == pygame.K_d:
392                         keypress = (1, 0)
393                     if event.key == pygame.K_w:
394                         keypress = (0, -1)
395                     if event.key == pygame.K_s:
396                         keypress = (0, 1)
397
398             # convert keypress to direction
399             left = tuple(np.dot(M_LEFT, self.game.direction))
400             right = tuple(np.dot(M_RIGHT, self.game.direction))
401             if keypress == right:
402                 action = 1
403             elif keypress == left:
404                 action = 2
405             else:
406                 action = 0
407
408             _, dead, self.score = self.game.step(action)
409             if dead:
410                 self.game_over()
411

```

The main game loop is very simple: lines 385-396 poll the pygame event queue for any input, and if it finds a keypress, records it. Line 387 is run when the game is quit by pressing

the actual close button on the window. Lines 399-400 calculate the directions to the left and right of the snake by matrix transformations, as discussed earlier. If the keypress indicates either the left or right direction, the relevant action is taken. Otherwise, the snake just continues forward.

```
413 class DQTrainer:
414     def __init__(self, model=None):
415         self.state_size = 8
416         self.n_actions = 3
417
418         self.gamma = DISCOUNT_FACTOR # discount factor
419         # e-greedy
420         self.e_decay = -0.00013
421
422         self.batch_size = 64
423         self.min_mem_length = 1000
424         # replay memory deque
425         self.replay_mem = deque(maxlen=10000)
426
427         if model:
428             self.epsilon = 0
429             self.active_net = keras.models.load_model(model)
430         else:
431             self.epsilon = 1
432             self.active_net = self.build_net()
433
434         self.target_net = self.build_net()
435
436     def build_net(self):
437         # define the network architecture for Keras
438         net = keras.models.Sequential()
439         net.add(keras.layers.Dense(12, input_dim=self.state_size, activation="relu"))
440         net.add(keras.layers.Dense(self.n_actions, activation="linear"))
441         net.compile(loss="mse", optimizer="Adam")
442         return net
443
444     def update_target(self):
445         # update the target network to be equal to the active network
446         self.target_net.set_weights(self.active_net.get_weights())
447
448     def remember(self, state, action, reward, next_state, done):
449         # add an action taken to the memory, and decay the epsilon to increase exploitation with each action taken
450         self.replay_mem.append((state, action, reward, next_state, done))
451         self.epsilon *= np.exp(self.e_decay)
452
453     def act(self, state):
454         # choose an action either randomly or by network
455         if np.random.rand() <= self.epsilon:
456             action = np.random.randint(self.n_actions)
457         else:
458             Q_vals = self.active_net.predict(np.array([state]))[0]
459             action = np.argmax(Q_vals)
460         return action
461
462 > def learn(self): ...
504
505 > def train(self, n_episodes, max_steps=1000): ...
551
552
```

The class responsible for training the neural net. The constructor method stores many of the hyperparameters for training as attributes. The build net method implements the architecture defined as part of the design documentation. Line 438 declared a sequential neural network, the next line adds a fully-connected linear layer with 12 neurons, input dimension 8 (state size) and activation ReLU, and the next line adds the output layer with size 3 (n actions) and linear activation. Line 441 compiles the net with mean squared error for the cost/loss function, and the 'Adam' algorithm as the optimiser.

```

462 def learn(self):
463     if len(self.replay_mem) > self.min_mem_length: # only run if enough data gathered
464
465         # grab a random sample and unpack it into lists and arrays
466         batch = np.array([self.replay_mem[x] for x in np.random.randint(len(self.replay_mem), size=self.batch_size)])
467
468         states = np.zeros((self.batch_size, self.state_size))
469         next_states = np.zeros((self.batch_size, self.state_size))
470         actions, rewards, dones = [], [], []
471
472         for n in range(self.batch_size):
473             states[n] = batch[n, 0]
474             actions.append(batch[n, 1])
475             rewards.append(batch[n, 2])
476             next_states[n] = batch[n, 3]
477             dones.append(batch[n, 4])
478
479         # predict Q values of actions from s from active net
480         target = self.active_net.predict(states)
481         # predict Q values of actions from s' from active net
482         target_next_active = self.active_net.predict(next_states)
483         # predict Q values of actions from s' from target net
484         target_next_target = self.target_net.predict(next_states)
485
486         for n in range(self.batch_size):
487             # get maximum Q value at s' from target model
488             if dones[n]:
489                 target[n][actions[n]] = rewards[n]
490                 # if action directly caused a failure give it its negative reward instead of predicted one, so network learns not to fail
491                 # this can be done in this case as there will be no more future rewards, so bellman equation below doesnt apply
492                 # cannot be done for actions that caused positive rewards as future after that has to be taken into account
493
494             else:
495                 a = np.argmax(target_next_active[n]) # select most favourable action from s' using active net
496                 target[n][actions[n]] = rewards[n] + self.gamma * target_next_target[n][a]
497                 # set Q value for action taken equal to max discounted future reward from target net
498                 # target_next_target[n][a] is the Q-value for the action taken from s'
499                 # this makes target[n][actions[n]] the Q-value for the action taken from s, equal to the known reward plus estimated future reward
500
501         # train the batch, fit states to target Q values
502         self.active_net.fit(states, target, batch_size=self.batch_size, epochs=1, verbose=0)

```

This is the method that implements the Double-Deep-Q learning algorithm. I left detailed comments in the code while writing it as I had a lot of trouble writing and understanding this part, so they shall serve to explain what's going on here. s is used here to denote the current state from experience, and s' the next state. The 'optimal' Q values are all calculated into the array 'target', which is what is passed to the network to compute loss. Note how only the Q-value for the action taken is calculated, and the rest are left as they were predicted by the network, and passed back to it. This gives the network a loss of zero for the other two Q-values, so that the network is only trained on the one that was calculated by the bellman equation in line 496. The optimal Q values for the other two cannot be calculated as they were not acted on, so the next states/rewards for them are not known.

```

504 def train(self, n_episodes, max_steps=1000):
505     env = Game(ROWS, SCREEN_SIZE)
506     data = []
507     for e in range(n_episodes):
508         dead = False
509         score = 0
510         state = env.get_state()
511         steps = 0
512
513         # play episode
514         while True:
515             pygame.event.pump()
516
517             action = self.act(state) # choose an action and take it
518             apple, dead, _ = env.step(action)
519             if dead:
520                 reward = DEATH_PENTALTY
521             elif apple:
522                 reward = APPLE_REWARD
523                 score += 1
524             else:
525                 reward = 0
526             next_state = env.get_state()
527
528             self.remember(state, action, reward, next_state, dead)
529
530             state = next_state
531             steps += 1 # move to next step
532
533             self.learn() # train model at each step
534
535             if dead or (steps >= max_steps):
536                 self.update_target() # copy the active network to the target network
537                 env.new_game() # start new game
538                 print("episode {0:<4d} survived {1:<4d} steps scored {2:<2d} memory len {3:<4d} epsilon {4:<6.9f}"
539                       .format(e, steps, score, len(self.replay_mem), self.epsilon))
540                 data.append((e, steps, score, self.epsilon))
541                 break
542
543         # save trained model
544         print("{} episodes complete, saving model...".format(n_episodes))
545         timestamp = datetime.now().strftime("%d-%m-%H-%M")
546         self.target_net.save("model-" + timestamp + ".h5")
547         with open("data-{}.pkl".format(timestamp), "wb") as f:
548             pickle.dump(data, f)
549

```

This is the method that runs the main training loop. Lines 505-506 initialise the game environment and list to store data. Lines 508-511 are run before each new game starts, reinitialising all the variables needed for running the game and training. Lines 515-541 are the main game loop. An action is chosen by the `act()` method and then passed to the game environment to be taken. The consequences of that action are evaluated by lines 519-525, calculating rewards as necessary. The experience is added to the replay experience memory by line 528, and then line 533 calls the training algorithm to train the snake. The active network is trained at every game step so that the snake is always improving, and also can train out of any loops it gets stuck in. This frequency of training is offset by the small batch size of only 64. When the snake dies, the active net is copied back into the target net so the target net can give better Q^* approximations, and a new game is started. Lines 544-548 are run when training is finished, saving the model and training diagnostic data to disk.


```

552 if __name__ == "__main__":
553     import argparse
554     parser = argparse.ArgumentParser(description="AI Snake")
555     group = parser.add_mutually_exclusive_group(required=True)
556     group.add_argument("-t", "--train", type=int, metavar="n_games", nargs="?", const="1000",
557                        help="train a new model, specify number of games to train with, default 10000")
558     group.add_argument("-r", "--run", type=int, metavar="n_games", nargs="?", const="1",
559                        help="run some games with a trained model, specify number of games to run, default 3")
560     group.add_argument("-p", "--play", action="store_true", help="play the game yourself")
561     group.add_argument("-l", "--leaderboard", type=str, metavar="player name", nargs="?", const="all",
562                        help="view the leaderboard,specify players scores to view, displays all by default")
563     group.add_argument("-d", "--debug", action="store_true", help="run in interactive mode for debugging")
564     args = parser.parse_args()
565     if args.play:
566         p = HumanPlayer()
567         p.play()
568     elif args.leaderboard:
569         l = Leaderboard(L_FILE)
570         try:
571             l.display(name=args.leaderboard)
572         except ValueError:
573             print("that player does not exist!")
574     elif args.run:
575         r = DQPlayer(M_FILE)
576         r.play(args.run)
577     elif args.train:
578         t = DQTrainer()
579         t.train(args.train)
580     elif args.debug:
581         import code
582         code.interact(local=locals())
583

```

This is the information that handles creating the CLI. Notice line 551, which means that this code is only run if this script is run, and not imported into or called by another file. Argparse is used to create a CLI, then add all the arguments to it with their respective options. They are added to a mutually exclusive group so the parser will only accept one argument. Lines 561-578 all instantiate and call the respective parts of the code, dependent upon which argument is passed.

4 Testing

4.1 CLI

Before testing any of the rest of the program, it makes sense to test that the interface works. Most of the error handling and input checking is handled by the python argparse module, so it should work mostly as intended. Tabled below is a series of inputs to the program from the command line, their purposes, and their results.

The tests were all run by executing the command

```
> python ai-snake.py <input>
```

at the command line. The screenshots of the tests and their outputs are listed in appendix B.

No.	Input	Expected outcome	Actual outcome
1	None	Error, no arguments were passed	As expected
2	"test"	Error, unrecognised input	As expected
3	-h	Prints help info	As expected
4	-p	Launches app in human player mode, terminal prompts for player's name on death	As expected
5	-t	Launches game with default number of training games (1000) Prints Tensorflow debug info and starts training	As expected
6	-d	Launches interactive python console with the program's classes and constants all pre-defined.	As expected
7	--debug	Same as above	As expected
8	-r	Runs the default (1) number of games played by the ai	As expected
9	-r 3	Runs 3 games played by the ai	As expected
10	-r help	Error, "help" is not an integer	As expected
11	-p 5	Error, unrecognised argument for -p	As expected
12	-t 5	Runs 5 games to train a model, then saves model to file.	As expected
13	-v	Error, unrecognised argument	As expected
14	-l	Leaderboard is called, but returns message about being empty (no data has been added to it at this point)	As expected

Everything works as intended, mainly because I tested as I went along. However, these final tests still serve as a good double-check.

4.2 Architecture

Designing a neural network is a very iterative process: there are many hyperparameters such as batch size, number of hidden layers and neurons, input parameters etc. All these parameters interact in such a way that can very drastically affect the AI's performance. There are millions of possible combinations of these values, so finding the one that is

optimal can take a long time, especially when it can take an hour before the training network shows any performance that is representative of anything.

The architecture detailed in the design section is the final architecture arrived at after a very long time testing different architectures and tuning hyperparameters. There are a few interesting things that I noticed during this process:

- When it comes to number of hidden layers and neurons- less is often more. The initial network I started with had 2 hidden layers: one of 16 followed by one of 8 neurons. This made training take a long time, and didn't actually improve the performance. If anything, the network performed better when I trimmed it down to one layer of 12 neurons, and it also trained much faster, sometimes converging to ideal behaviour after only a few hundred episodes.
- One of the biggest things that made a difference to the snake's behaviour was the rewards. A reward of 100 is given for an apple, and -10 for death. These may seem out of proportion, but because getting an apple is a much more distant thing than death, the reward must be high enough that when discounted as a future reward due to the discount factor, it is still a big enough reward to target. A good abstraction of this concept is that the higher the value of the reward, the further into the future the snake can see it. The death reward does not have to be too high, as death is often very immediate for the snake. This does however mean that the snake often manoeuvres itself into positions it cannot get out of without dying, like wrapping itself up in its own tail. However, this kind of behaviour is hard to avoid, especially when in a situation where the snake's tail lies in the path to an apple, and the snake sees the reward for the apple as higher than the punishment for dying.
- The discount factor had to be very close to 1 for the snake to do anything useful: a value of 0.995 was eventually arrived at. The discount factor is essentially the snake's ability to 'see the future,' and as for a game like this most rewards lie far into the future, it would make sense that this is very high. This value seemed to have a large impact on the snake's behaviour, changing it by even ± 0.001 resulted in a snake that just spun in circles, only scoring a couple of points.
- The main behaviour that it was difficult to get the snake to avoid was spinning in circles, especially during the early stages of training. The immediate strategy the snake discovered was to not die, and that the best way of doing this was to spin in circles. It took some time for the snake to discover that eating apples yielded a much higher reward than spinning in circles, helped partially by the ϵ -greedy technique implemented to help the snake explore different strategies. Some models never did make it past this stage, however.

4.3 Compute issues

As already discussed during the analysis of my investigation, I intended to utilise the hardware matrix-multiplication accelerators on my Nvidia RTX GPU to make training the networks much faster. However, during training I noticed that running the training

algorithm on my desktop computer's CPU was actually much faster. This prompted me to do some investigation.

Figure 4.1 shows a graph generated by me using Tensorflow of the time taken to multiply two randomly generated square rank-2 tensors of varying sizes using my desktop GPU, desktop CPU and laptop CPU. As you can see the CPU time increases exponentially, while GPU is very low and remains almost constant for lower matrix sizes, demonstrating the power of GPU compute for this application.

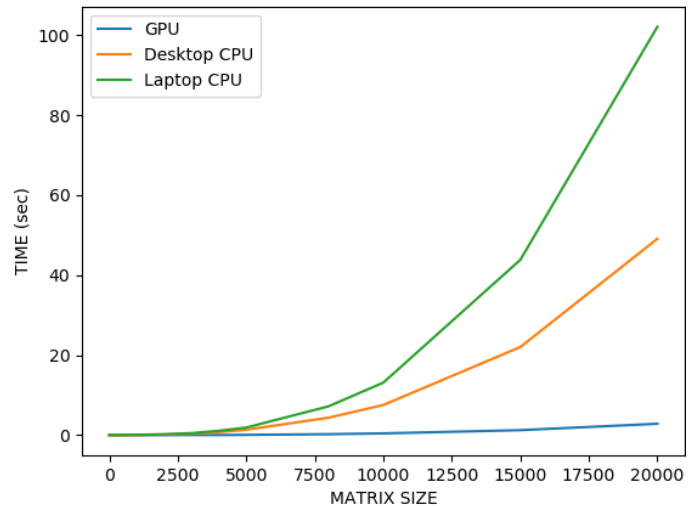


Figure 4.1 – CPU vs GPU performance

However, the reason I was not able to access this kind of performance with my neural network was because the tensors involved in a 3-layer network with only a handful of neurons per layer are relatively small. While the GPU is able to perform the raw calculations quicker than my CPU, the overhead in setting up the computation, transferring the data from system memory to GPU memory, and transferring it back made performing the computation faster on CPU.

This bottleneck could perhaps be made less of an issue with faster memory or better system interconnects, but those resources were not available. Increasing the batch size of the training from 64 would also help negate this issue, but this would be detrimental to the performance of the snake as training too much in one step can cause the snake to learn too quickly and adjust strategy too dramatically, instead of learning from a variety of experiences over time as it slowly improves its performance.

For this reason, all the training was done on the desktop CPU, which was still faster than the laptop as it could sustain higher clocks for the longer periods required for training the models. This was a shame as hardware research formed a large part of my analysis, but such things are the nature of more investigative projects.

4.4 Training and Testing

After I had arrived at the final architecture, I trained 5 models using it and the results varied massively. All 5 models exhibited the intended behaviour of going after apples, but some much more successfully than others. All models were trained for 1000 episodes, and then tested for another 5000.

Occasionally during testing, the snake would get stuck in a loop that it was unable to break out of. To solve this, after a certain number of steps taken without a score, the snake would detect that it was stuck in a loop and take the action of the second highest Q-value instead of the first, which would usually break it out of a loop. In the case that it didn't, the snake would be killed if it went a further number of steps without a score. During training, this wasn't applied, and the snake was simply killed after 1000 steps regardless.

Data for each model is given and explained below. The first graph for each model shows how the score changed as the model trained, with the average score per 5 episodes being plotted to reduce noise in the data and make the graph more comprehensible. The second shows the distribution of scores as the model trained, with the red curve as a normal approximation to the data.

Alpha

Alpha was one of the worst performing out of the five models, with a mean score of only 9.4. The data is skewed heavily towards the lower scores, indicating very poor performance. This is reflected in the training data, with the network never really scoring consistently highly. Interestingly, this model did not have any problems with looping, it just wasn't very good at avoiding its own tail.

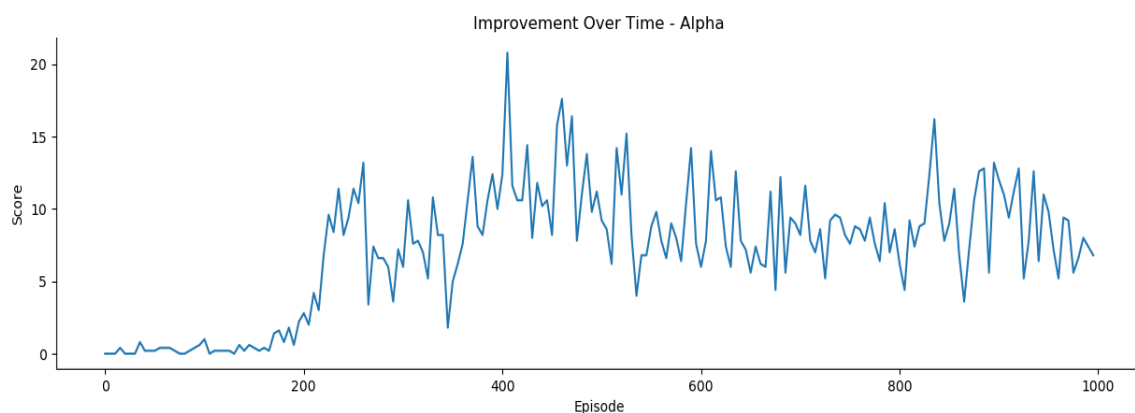


Figure 4.2

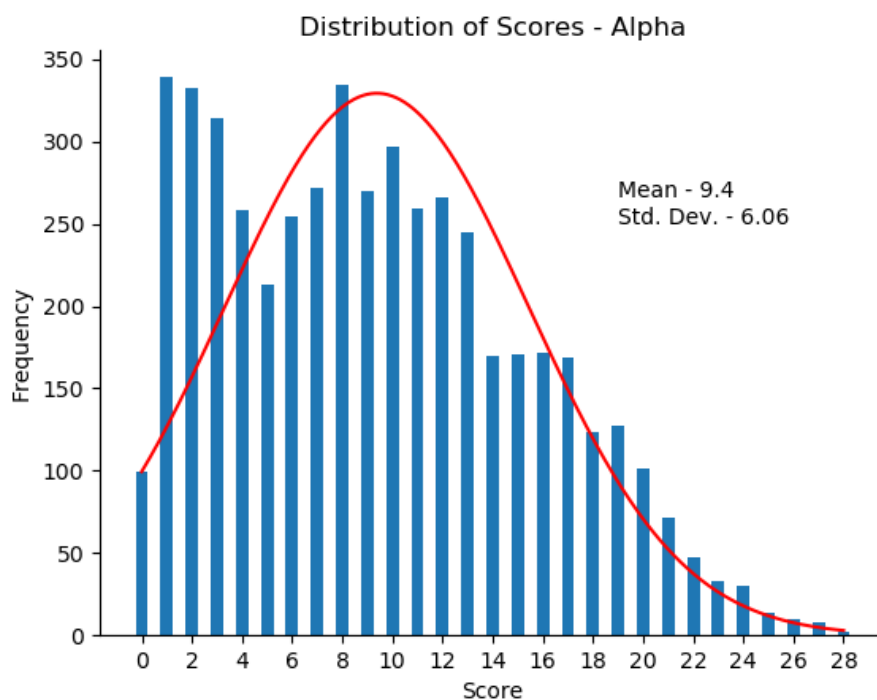


Figure 4.3

Beta

This model was one of the best performing ones with a mean score of 24.5. It was consistently scoring highly, being able to avoid its own tail well. It had a few issues with looping but was able to break out of them with the loop detection. However, the distribution of the scores was strange, with the snake scoring 21-23 and 31 much more frequently than any other score. A possible explanation for this could be the activation of the input neuron that gives the snake's length causing strange behaviour that causes the snake to die when its tail is at these specific lengths. The high performance of this model is also demonstrated in the training graph where it scores consistently, compared to, for example, alpha, which dips much lower and is much less consistent in training.

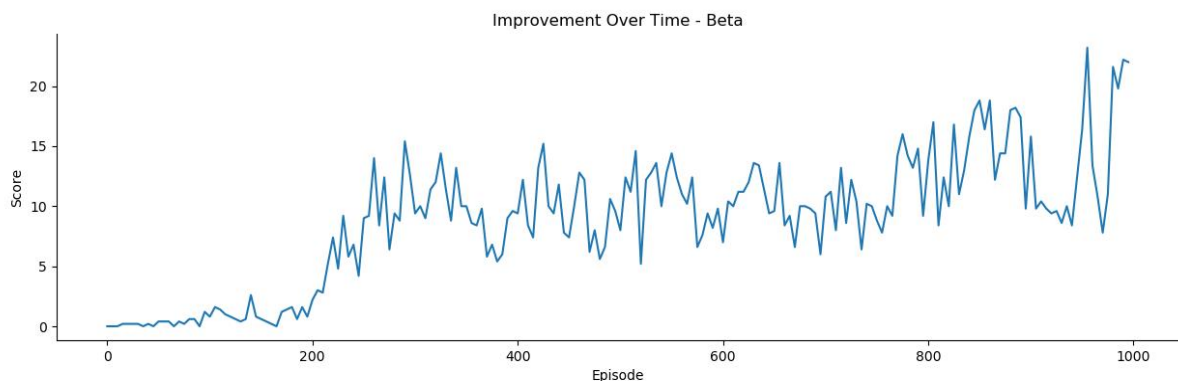


Figure 4.4

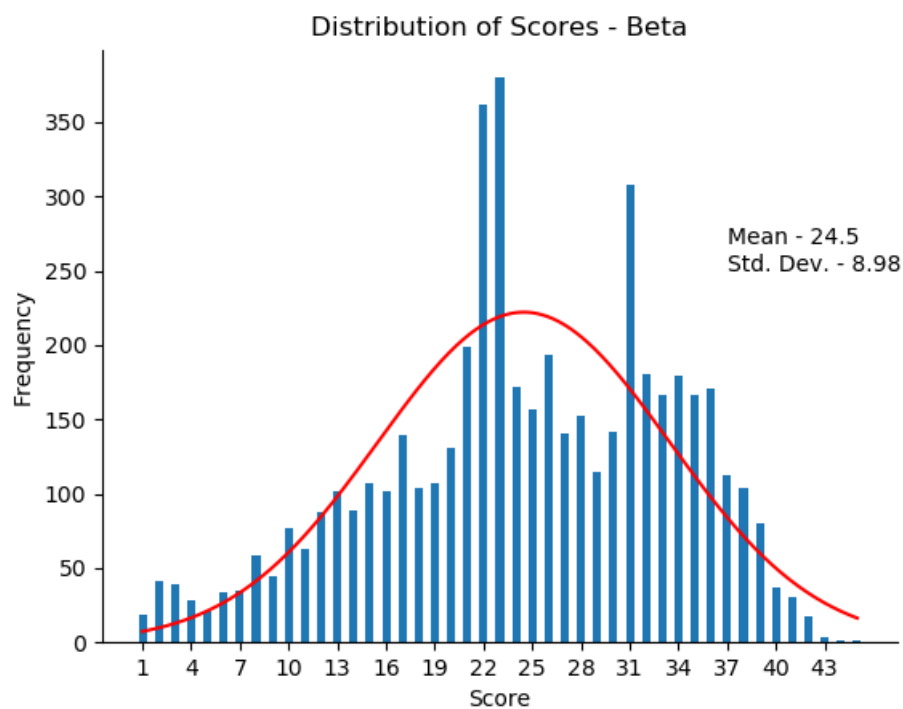


Figure 4.5

Charlie

While still performing better than alpha, Charlie was still pretty bad. It had a big problem with looping, getting stuck spinning in the same very specific pattern most rounds, and when breaking out of it just got stuck back in another loop again. As can be seen from the graph, it didn't have much trouble getting to a score of around 12-17, which accounted for much of its scores, but getting past around this mark was rare. Interestingly, the training graph for this model appears to show consistently high results around the 600 mark, then drops off quite suddenly and climbs back up again, demonstrating a particularly unstable training pattern.

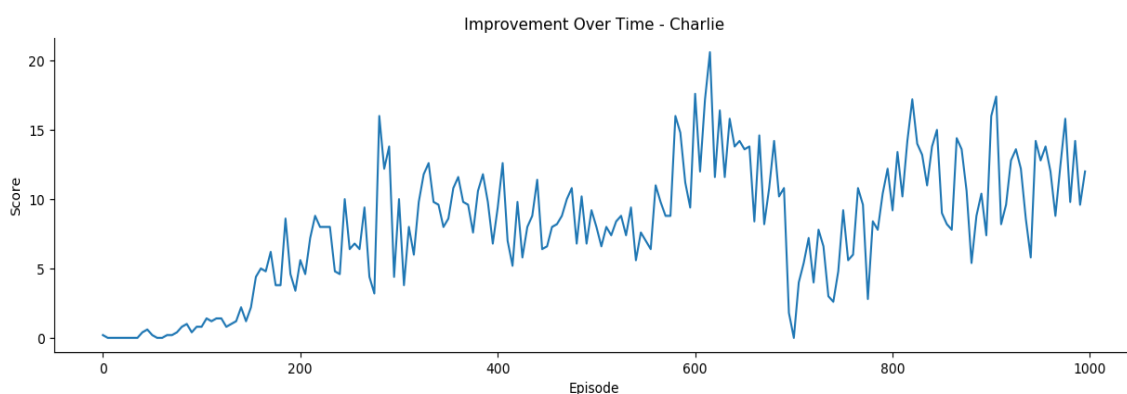


Figure 4.6

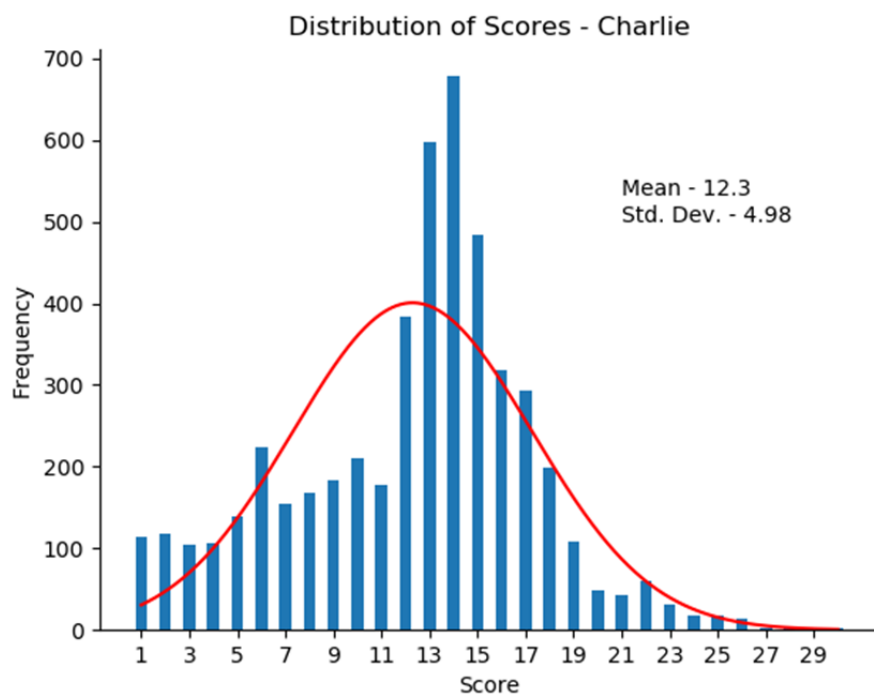


Figure 4.7

Delta

This model performed at consistent rate with an even distribution of scores, which is a favourable quality, but didn't score very highly. While it has a lower mean score than alpha, there was much less skew in the data. This model also had an issue with looping, although less so than Charlie as it had more success in breaking out of loops. The training pattern for this model was particularly unstable with the score spiking to some lows but not very many highs: the peak score was around 12.

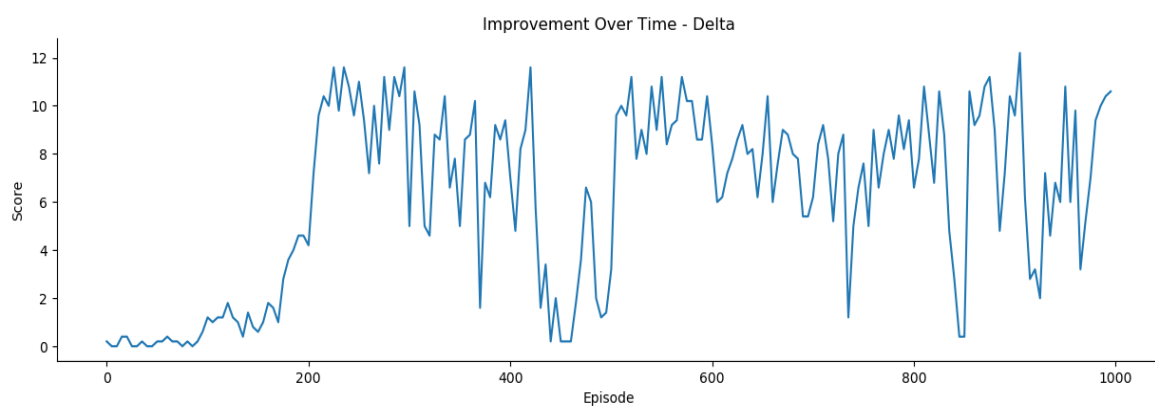


Figure 4.8

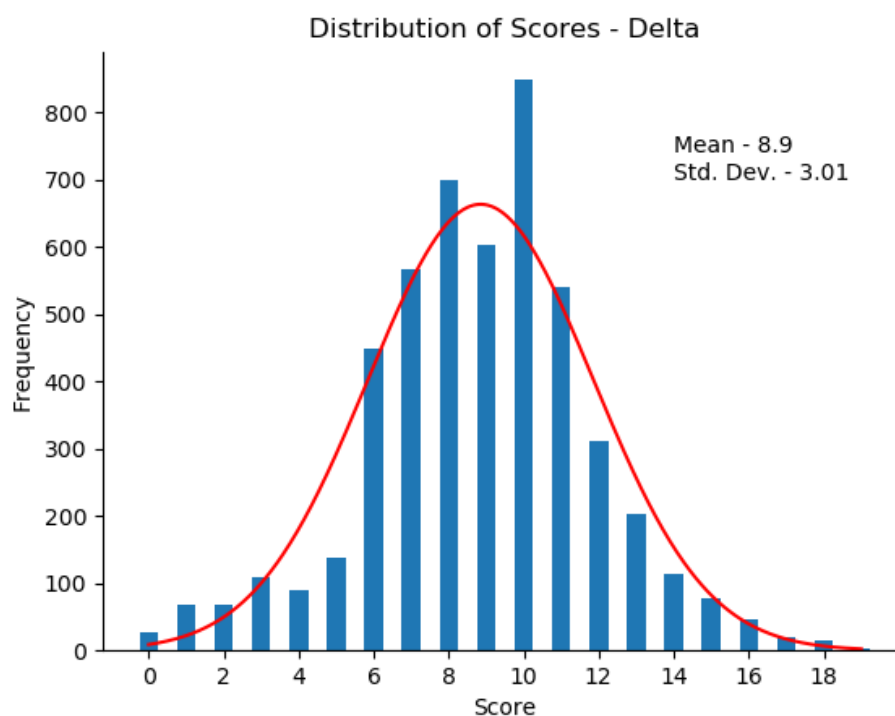


Figure 4.9

Echo

Echo was also one of the best performing models with a high mean score of 22.9. It also had no issues with looping, and has an even distribution of scores, meaning it doesn't exhibit any strange behaviour at any certain points unlike beta. Also, the training graph shows a sharp increase in score around 900 steps and maintains that high score, showing that the training ended with consistent high results and the model was saved in this state, meaning this high performance is what was carried over to the saved model.

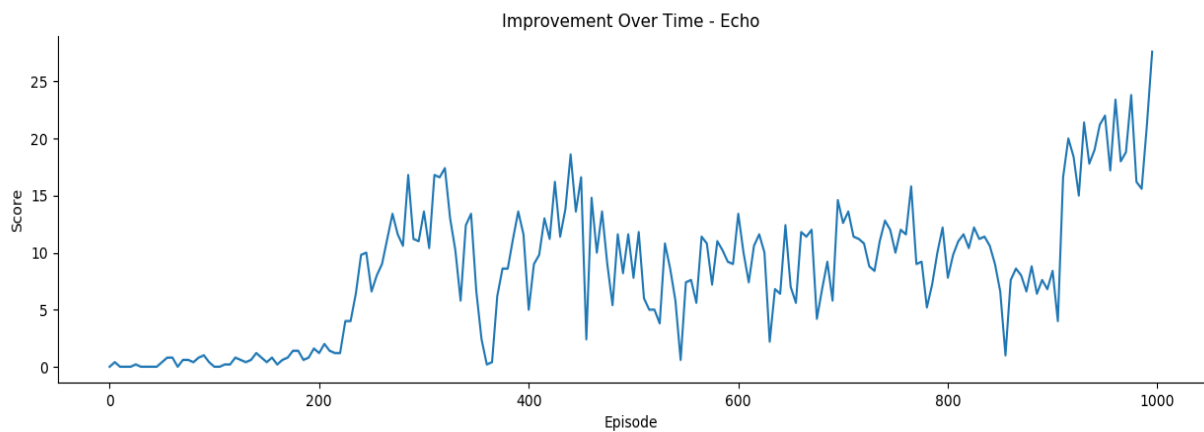


Figure 4.10

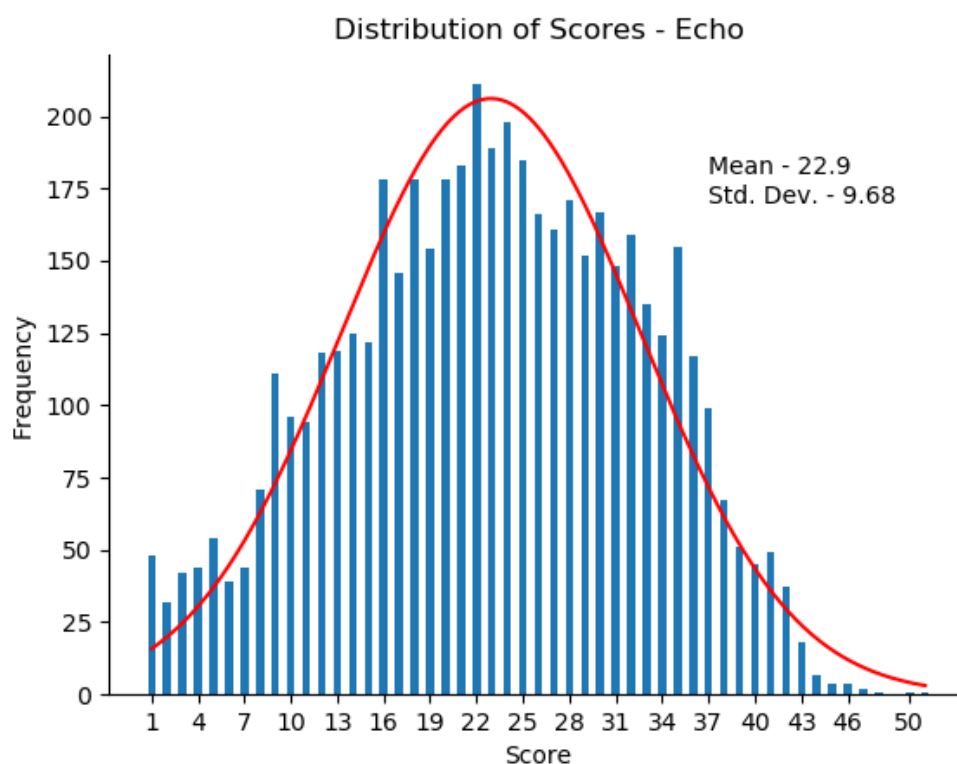


Figure 4.11

Observations

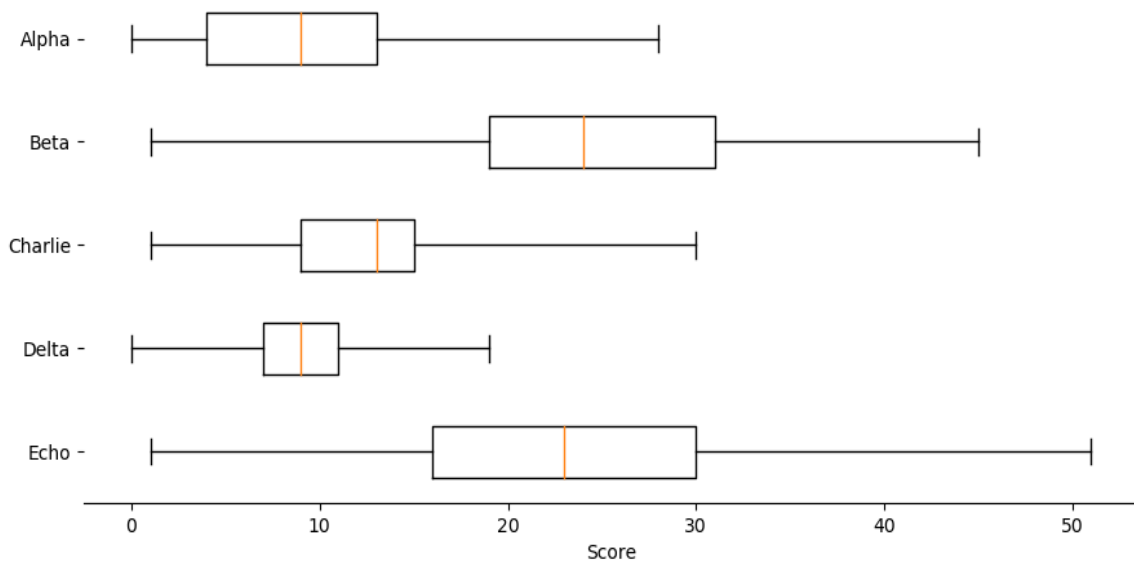


Figure 4.12 – Distribution of AI performances

Figure 4.12 shows the variation in the performance of the models, despite being trained under identical conditions. There are a few reasons which could serve to provide some explanation for this, including one of the inherent issues with DDQ networks.

All of the training graphs show a large amount of instability during training, which is to be expected due to the random nature of the game. However, an almost cyclical pattern can be seen in some of the graphs with periods of scoring high, and then low. This is due to the unstable nature of deep-Q networks. Recall that the loss of the network is calculated as the mean squared error of the predicted Q values, and the optimal Q^* values calculated by the target network. As the Q^* values are also estimated by a neural network, it is essentially optimising to itself, causing it to be effectively 'chasing its own tail.' This is alleviated by the Q^* values being calculated by another network, the entire point of a double DQN, but it can still be an issue. Because the target network is only a copy of the active network, updated every time the snake dies, the two networks are very similar which is where this problem arises.

This cyclical pattern introduces an element of randomness into the training, as to whether the 1000-episode training period will finish on a high, or a low. Models beta and echo both finish on highs, whereas alpha finishes on a low, which explains their respective performances.

The random nature of the gradient descent algorithm also explains why the performance and behaviour of the models are so different despite all being trained with the exact same parameters. The role of gradient descent is to optimise the cost function by finding its minima, which is what trains the neural network.

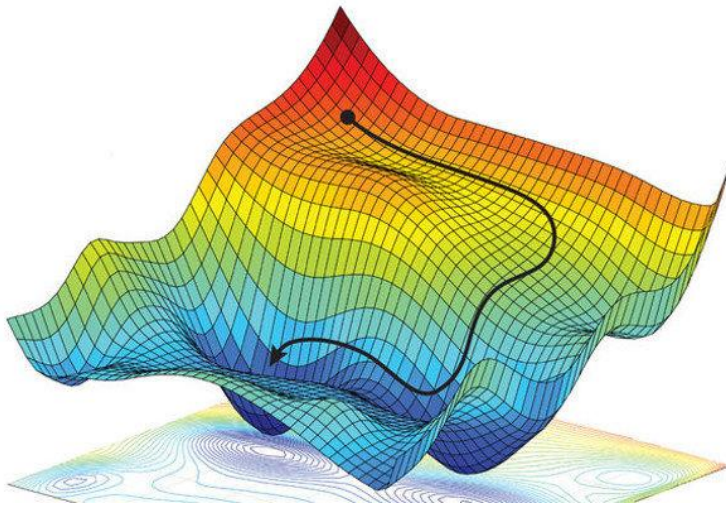


Figure 4.13 – A graphical representation of gradient descent in 3D space

Finding the minimum value of a single-variable function is simple calculus, but when every weight in the network needs to be optimised, the function becomes more complex. The analogy often used is rolling a ball down a hill, with the intent of finding the lowest valley at the bottom, as can be seen in the image. However, there are many local minima in this valley, and the ball could fall into any one of them, thinking it is the global minimum. There is the

added issue with a deep-Q network in that the valley is constantly changing shape, as the Q^* value is approximated by the network, which means the ball has no idea what is going on and is just trying to do its best. The ball could fall into a minimum, only for that minimum to disappear and a new one appear next to it, explaining the unstable, cyclical nature of the training graphs.

The final choice for the model to be used during the rest of the testing is echo, due to its high scores, but also well-distributed scores, which indicates predictable, good behaviour; unlike beta which despite a slightly higher mean score had issues around certain score points, and also a slight habit of looping. Epsilon also had a higher maximum score, indicating better potential to score very highly.

4.5 Leaderboard & Human Interface Testing

The rest of the system consists the human playable game, and the leaderboard to keep track of scores. To test both of these components I just played 50 rounds of snake, using the leaderboard to keep track of scores as I play. This will serve a double purpose, as the data collected while playing can also be used to compare the performance of a human player to AI.

Screenshotting every frame of the game as I play to prove it works is impractical, but I can report that over the course of testing everything mostly worked as intended, with the exceptions to this discussed below. The system that usually causes the main issues with a game like this is collision detection, but as the entire game was written using a matrix as the underlying data structure for the grid, this was no issue as it just involved checking the value at the position the snake was trying to move to.

I discovered a quirk where the apple would occasionally be placed on top of the snake's tail, despite the game checking that the square was empty before selecting coordinates for the apple. The reason for this was to do with the order in which the game executes steps, and which things are rendered: the apple was being placed before the snake was moved in

accordance with the human input, so it was putting it in an empty space, and then the snake was moving over it, and because apples are above snake tail in the expression that renders the matrix values onto the window, the apple would replace a section of snake tail. This was fixed by moving function calls around, so the apple was repositioned after the snake was moved.

Another bug in the program pertains to movement and framerate. Every game step, the game checks for input, and then moves the snake based on the input. The issue with this is that if two key presses are given during the same game step, only one will be detected. This can be alleviated by making the game step shorter, but doing so makes the snake move too fast for a human to keep up. The current game step length ($1/13^{\text{th}}$ of a second) makes the game input feel smooth without making the snake too fast, however slight inconsistencies in timing can still sometimes make the input feel off.

The data for the games I played is graphed below. As the quantity of data is much lower (50 games) a bar chart didn't work, so a histogram is given.

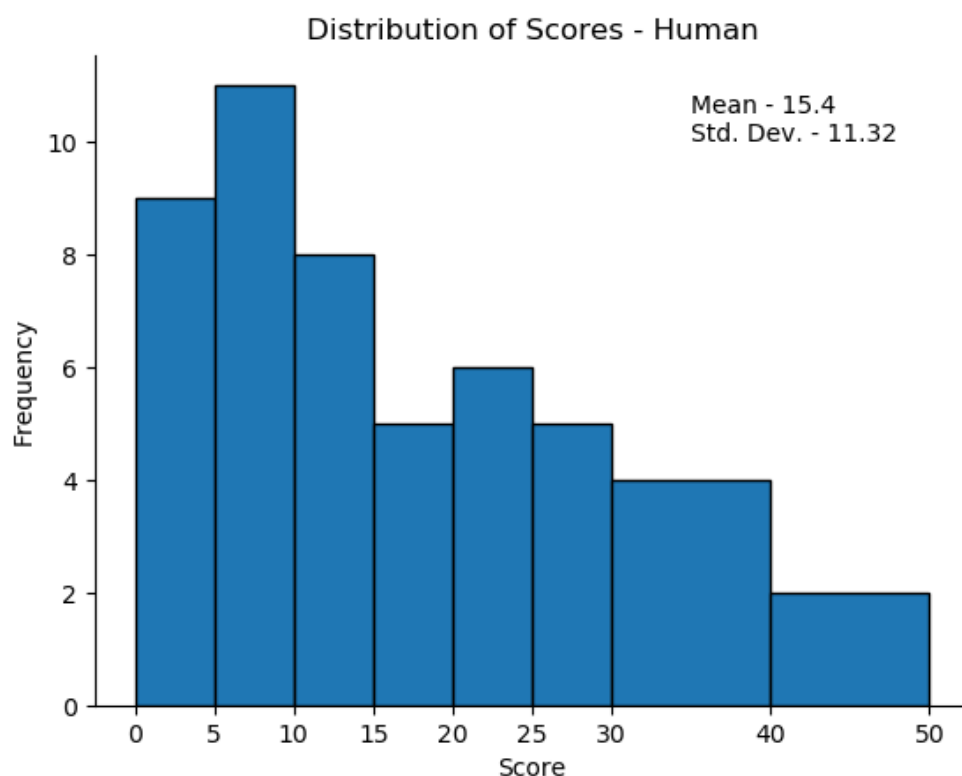
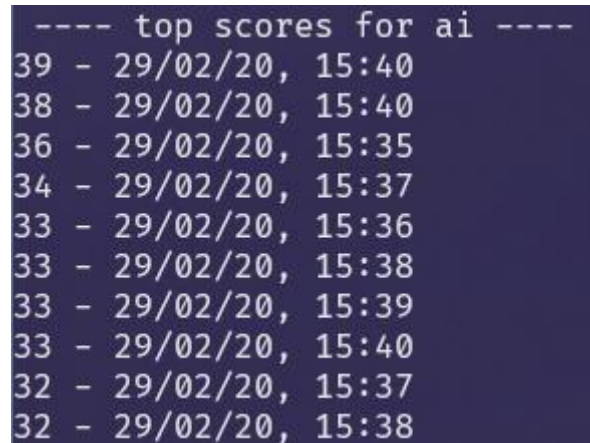


Figure 4.14 – Results of 50 games of snake

As can be seen, I was outperformed by the AI by a significant margin. Although I was better than some of the models shown earlier, the final AI had a higher mean score than me by 8 points. The amount of data collected on human performance is small, so no detailed conclusions can be drawn, but it can still be said that the AI has achieved above human-level performance in this case.

For a fair comparison, 50 games were also ran as the AI and recorded on the leaderboard. Another tester (Tim) was also drafted to play a handful of games to help test the leaderboard, their scores are also shown. Use of the program to list these scores, and the commands ran to obtain the output, is demonstrated below.

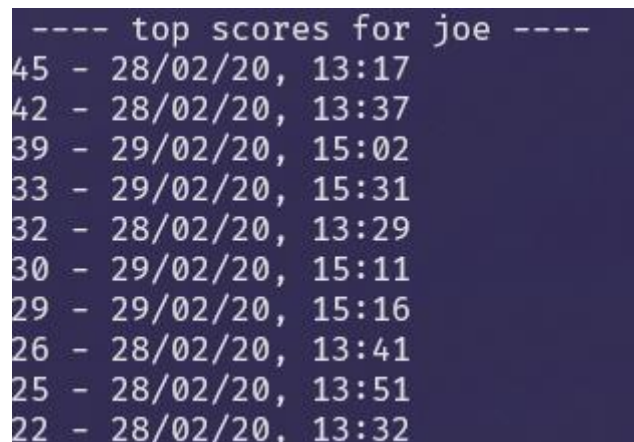
AI scores: `python ai-snake.py -l ai`



```
---- top scores for ai ----
39 - 29/02/20, 15:40
38 - 29/02/20, 15:40
36 - 29/02/20, 15:35
34 - 29/02/20, 15:37
33 - 29/02/20, 15:36
33 - 29/02/20, 15:38
33 - 29/02/20, 15:39
33 - 29/02/20, 15:40
32 - 29/02/20, 15:37
32 - 29/02/20, 15:38
```

Figure 4.14 – Screenshot of terminal output

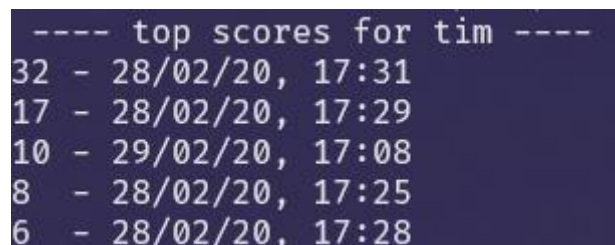
My (Joe's) scores: `python ai-snake.py -l joe`



```
---- top scores for joe ----
45 - 28/02/20, 13:17
42 - 28/02/20, 13:37
39 - 29/02/20, 15:02
33 - 29/02/20, 15:31
32 - 28/02/20, 13:29
30 - 29/02/20, 15:11
29 - 29/02/20, 15:16
26 - 28/02/20, 13:41
25 - 28/02/20, 13:51
22 - 28/02/20, 13:32
```

Figure 4.15 – Screenshot of terminal output

Tim's scores: `python ai-snake.py --leaderboard tim`



```
---- top scores for tim ----
32 - 28/02/20, 17:31
17 - 28/02/20, 17:29
10 - 29/02/20, 17:08
8 - 28/02/20, 17:25
6 - 28/02/20, 17:28
```

Figure 4.16 – Screenshot of terminal output

All scores: `python ai-snake.py -l`

```
---- top scores all time ----
joe  - 45 - 28/02/20, 13:17
joe  - 42 - 28/02/20, 13:37
joe  - 39 - 29/02/20, 15:02
ai   - 39 - 29/02/20, 15:40
ai   - 38 - 29/02/20, 15:40
ai   - 36 - 29/02/20, 15:35
ai   - 34 - 29/02/20, 15:37
joe  - 33 - 29/02/20, 15:31
ai   - 33 - 29/02/20, 15:36
ai   - 33 - 29/02/20, 15:38
ai   - 33 - 29/02/20, 15:39
ai   - 33 - 29/02/20, 15:40
joe  - 32 - 28/02/20, 13:29
ai   - 32 - 29/02/20, 15:37
ai   - 32 - 29/02/20, 15:38
tim  - 32 - 28/02/20, 17:31
ai   - 31 - 29/02/20, 15:34
ai   - 31 - 29/02/20, 15:39
joe  - 30 - 29/02/20, 15:11
ai   - 30 - 29/02/20, 15:35
```

Figure 4.17 – Screenshot of terminal output

As can be seen from the screenshots, the leaderboard works as intended. The entries in the all-time scores are consistent with those for each individual player. The all-time shows 20 scores, while each player shows only 10 scores (with the exception of Tim, who only played 5 games, so all 5 are shown).

Notice that the top of the leaderboard is dominated by me, and not the AI. In the 50 games recorded on the leaderboard, the AI had a max score of only 39, compared to its max of 51. This indicates the more consistent performance of the ai around it's mean score, whereas my scores were more scattered, and I was able to get very high (>40) scores more frequently than the AI. This attribute can also be determined from the summary stats of the data: the AI had a mean of 23 and standard deviation of 9.7, whereas I had a lower mean of 15 but higher standard deviation of 11.3. Of course, consistent scores about a mean from a computer and more variance in human data is an expected result, but interesting nonetheless: human inconsistency can lead to higher maximum (but also lower minimum) scores.

For completeness of testing, the result of the command `python ai-snake.py -l Steve` is shown. The player 'Steve' has never played the game, and as such should not exist on the leaderboard.



```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -l Steve
that player does not exist!
```

Figure 4.18 – Screenshot of terminal output

With the results of this test, it can be concluded that the system works as expected in its entirety.

5 Evaluation

5.1 Issues with Neural Network

While the performance of the neural network was still good, it was nowhere near perfect. I have seen many examples of neural networks trained to play the same game achieving much higher scores, with some even obtaining a perfect score, where the snake fills the board entirely. Most of these networks were using convolutional neural networks, CNNs. CNNs are much better at extracting detail from images, and are used in image recognition and automatic vehicle applications, as discussed as part of my analysis. As such, one would be well suited to my system, as it would be able to extract detail from the matrix data structure representing the game board.

Another issue with having such a simple network architecture is that it is limited in the amount of data it is able to process and how sophisticated its decision-making process can be. This issue is best represented by the snake taking actions that lead to it getting wrapped up in its own tail, and therefore inevitably dying. Despite the Q-learning supposedly algorithm being able to consider future consequences, these are only estimates and the lower negative future penalty of death is often offset by the high immediate reward of an apple. The limited input that the network takes means it is tricky for it to predict these kinds of situations. This issue could be overcome by providing the network with more information, by either passing the entire game grid as a flattened matrix to a much more complex linear neural network, or by employing a convolutional neural network, as discussed.

There are also more issues to be considered with a neural network than just architecture. Each activation function has its own merits and downfalls, and ReLU is no exception. ReLU was chosen because it is what is most commonly used, not very computationally expensive, and usually converges faster due to its constant gradient (recall how gradient descent uses derivatives with respect to activations to optimise the network's cost function). Because ReLU also outputs zero for all negative values, the network is sparsely activated: not all neurons are always firing, which results in less noise in the network and more concise models.

However, this can also become a problem. If the activation of a neuron becomes zero, as the function has no gradient for all negative values, it is likely to stay stuck at zero. This is known as the "dying ReLU" problem, which results in large parts of the network doing nothing and staying doing nothing. This is a particular issue in my case, as the training was observed to be rather unstable, a neuron could erroneously end up with zero activation, and then be unable to change recover from it. Also, there is a lot of negative feedback in the network, with negative rewards for death, so neurons are likely to 'die', which could provide some insight as to why the network was so bad at predicting death.

There are some variants of ReLU which help to overcome this problem. Leaky ReLU has a slight gradient of 0.01 for negative values, meaning the network is able to dig itself out of holes, albeit still with some difficulty. Exponential linear unit, ELU, also has a gradient for negative values, but uses the function $\alpha(e^x - 1)$ when $x < 0$. The gradient of ELU approaches 0 as the input value approaches $-\infty$, combining the favourable qualities of ReLU and leaky ReLU. Lowering the learning rate of the network can also help to mitigate dead ReLUs.

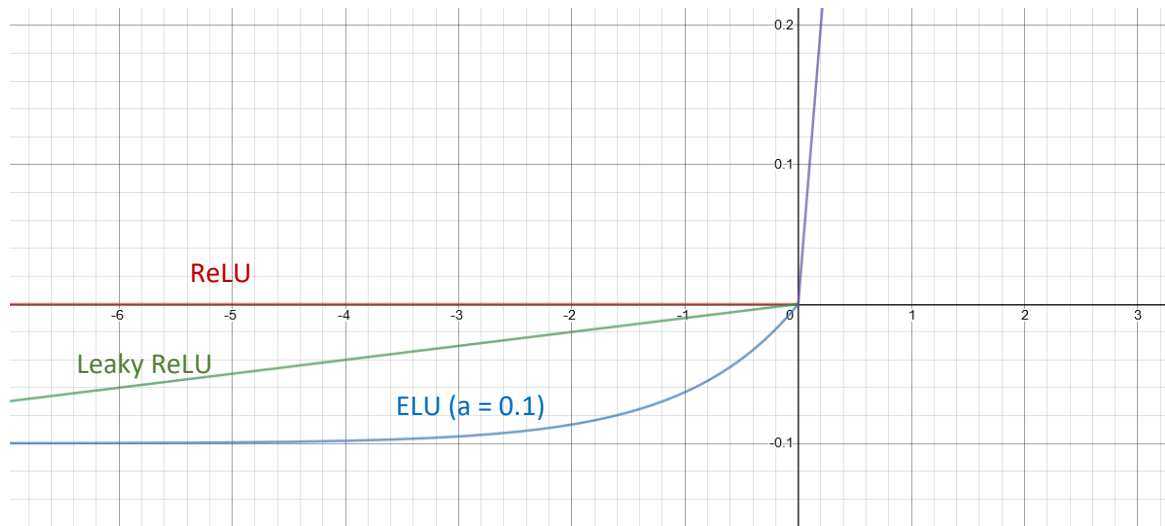


Figure 5.1 – Variations on ReLU

5.2 Other Potential Improvements

The obvious potential improvements to the system have already been discussed, with the neural network being the main aspect. The other architectures or activation functions discussed would have most likely lead to improved results, but the main reasons they weren't implemented in the first place are time constraints, and also my own lack of in-depth understanding of how they work. As one of the main motivations for this project was education, I felt it more appropriate (and more rewarding) to implement something I understood than to write code that I didn't understand the theory behind.

There are also further optimisations that can be made, specific to deep-Q networks. Prioritised experience replay is an extension to replay memory that weights the experiences based upon how relevant or important they are to optimising the network. This could have been implemented by including more experiences which lead to immediate consequences in the memory samples.

Another variation on deep-Q networks is duelling deep-Q networks (DDQN). DDQNs work by decomposing the $Q^*(s, a)$ function into two functions: a value function $V(s)$, which determines how good it is to be in state s ; and an action function $A(s, a)$, which determines the advantage gained by taking action a from state s .

$$Q^*(s, a) = V(s) + A(s, a)$$

This is implemented by splitting the neural network into two, one for each function, and then combining their values at the end. This technique decouples the estimation, and so allows the agent to determine if a state is good to be in or not without having to determine what actions it can take from that state. This leads to much faster models, as well as more accurate Q-values.

Testing other combinations of network hyperparameters such as rewards, learning rate, discount factor, numbers of neurons/layers. would also have likely yielded improved results. There is a near-infinite number of combinations of these parameters, and no given way to optimise each one, so I did the best I could with the time and resources available.

The other main issue already discussed is that of the keyboard input being tied to framerate. Solving this would require moving the snake forward, polling for input and the framerate of the game all to be independent of each other. A potential solution would be to have a very high framerate, such that tying input to it is not an issue, and then moving the snake forward automatically every few frames. However, early testing of this fix meant that the snake moved faster when driven by input than when moved automatically. Other solutions involved rearchitecting significant portions of the code, including how the AI interfaced with the game, so time constraints necessitated that this issue be left unresolved.

Much of the theory behind neural networks is quite complex, and is way beyond A-level standard, especially some of the vector calculus involved in training a neural network. Although I feel I have fulfilled the purpose of investigating neural network-based machine learning and AI, especially in learning about some of the more in depth theory behind Q-learning and neural network architecture, I have treated the gradient descent and backpropagation algorithms as a sort of 'black box', in that I don't fully understand the mathematics behind how they work: I have just used them by calling `keras.models.fit()` on my data. For this reason, I don't feel as if I have entirely fulfilled my objective of learning about neural networks, as these algorithms are core to their function. Gaining a better understanding of them would go a long way towards this goal, and also by providing more insight into how the network is trained, I could tweak things more to improve performance.

5.3 Evaluation of Success

Going back to the design objectives set out at the start of this project for the functionality of the system, I have been successful in fulfilling all of them. The snake game, AI, and leaderboard both meet all design criteria, and the testing data collected with the 5 different models allowed me to evaluate performance and investigate behaviour as I set out to do. The only design objective not fulfilled was that of designing the system to utilise GPU compute.

The motivation behind this objective was to investigate how hardware acceleration can provide a performance boost in neural network applications, and this was done and discussed as part of testing, finding that my specific network architecture and design was faster to run on CPU due to the overhead incurred in moving the data and initialising it on the GPU was higher than the speedup provided by hardware acceleration, but that for larger computations the GPU was indeed faster. In truth, there is no difference in writing code for

CPU or GPU when using Keras, all the GPU acceleration is done automatically if one is detected. In the code, an environment variable was set to hide the GPU from TensorFlow so that the CPU would be used instead. As this design objective was intended to promote investigation, I still feel it was a success.

The main success of course lies in the performance of the AI. Testing lead to the conclusion that the AI was achieving above human (my) performance, with a higher mean score of 8 points. Of course, more data on human performance in snake games could be collected to draw a more reliable conclusion, but I still believe this is an achievement. Designing an AI to beat a human was not one of my design objectives because I was not certain how possible it would be with my level of knowledge, and it wasn't the point anyway. The point was to learn about how it beats a human: whether it actually did or not was secondary.

Appendix A – References and Image Credits

Sources for the references in the analysis section, as well as the credits for every image that isn't a code screenshot is listed below.

References

[1] Tesla Autopilot

tesla.com/autopilot

[2] End to End Learning for Self-Driving Cars, Nvidia Corporation, April 2016

arxiv.org/pdf/1604.07316v1.pdf

[3] Nvidia Turing GPU Architecture, Nvidia Corporation, 2018

nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[4] Human Level Control Through Deep Reinforcement Learning, DeepMind, Jan 2015

deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning

[5] AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, DeepMind, Jan 2019

deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

[6] Cloud Tensor Processing Units, Google, August 2019

cloud.google.com/tpu/docs/tpus

[7] Tesla V100 GPU Architecture, Nvidia Corporation, August 2017

images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[8] “Tay, Microsoft's AI chatbot, gets a crash course in racism from Twitter,” The Guardian, March 2016

theguardian.com/technology/2016/mar/24/tay-microsofts-ai-chatbot-gets-a-crash-course-in-racism-from-twitter

Images

Figure 1.1 – [1]

Figure 1.2 – [2]

Figure 1.3 – [3]

Figure 1.4 – [4]

Figure 1.5 – [7]

Figure 2.1 – Drawn using draw.io

Figure 2.2 – Drawn using draw.io

Figure 2.3 – Drawn using desmos

Figure 2.4 – Drawn using desmos

Figure 2.5 – Drawn using draw.io
Figure 2.6 – Drawn using draw.io
Figure 2.7 – Screenshot from Visual Studio Code
Figure 2.8 – Drawn using draw.io
Figure 3.1 – Drawn using Word
Figure 4.1 – Generated using Tensorflow and Matplotlib
Figures 4.2-4.12 – Drawn using Matplotlib
Figure 4.13 - Spatial Uncertainty Sampling for End-to-End Control, Alexander Amin
Figures 4.14 – 4.18 – Screenshots from Windows Terminal
Figure 5.1 – Drawn using desmos

Draw.io is a free, online, open-source diagramming tool – drawio-app.com

Desmos is a free online graphing calculator – desmos.com

Matplotlib is a Python 2D plotting library, available on PyPI – matplotlib.org

Appendix B – Test Screenshots

1:

```
PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)
ai-snake.py: error: one of the arguments -t/--train -r/--run -p/--play -l/--leaderboard -d/--debug is required
```

2:

```
PS C:\Users\josep\Dropbox\Code\AI-snake> python ai-snake.py test
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)
ai-snake.py: error: one of the arguments -t/--train -r/--run -p/--play -l/--leaderboard -d/--debug is required
```

3:

```
PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -h
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)

AI Snake

optional arguments:
  -h, --help            show this help message and exit
  -t [n_games], --train [n_games]
                        train a new model, specify number of games to train
                        with, default 10000
  -r [n_games], --run [n_games]
                        run some games with a trained model, specify number of
                        games to run, default 3
  -p, --play            play the game yourself
  -l [player name], --leaderboard [player name]
                        view the leaderboard,specify players scores to view,
                        displays all by default
  -d, --debug          run in interactive mode for debugging
```

4:

```
PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -p
enter your name >>>
```

The game launched in another window, the enter your name prompt popped up after the game ended.

5:

```
PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -t
WARNING:tensorflow:From C:\Users\josep\AppData\Local\Programs\Python\Python36\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
2020-02-12 14:00:05.109353: E tensorflow/stream_executor/cuda/cuda_driver.cc:318] failed call to cuInit: UNKNOWN ERROR (303)
episode 0    survived 192  steps scored 0  memory len 192  epsilon 0.962327933
episode 1    survived 96   steps scored 0  memory len 288  epsilon 0.944027483
episode 2    survived 124  steps scored 0  memory len 412  epsilon 0.920903524
episode 3    survived 98   steps scored 0  memory len 510  epsilon 0.903029552
episode 4    survived 201  steps scored 0  memory len 711  epsilon 0.867447750
episode 5    survived 505  steps scored 0  memory len 1216 epsilon 0.784114675
episode 6    survived 450  steps scored 0  memory len 1666 epsilon 0.716626854
episode 7    survived 82   steps scored 0  memory len 1748 epsilon 0.704970021
episode 8    survived 598  steps scored 0  memory len 2346 epsilon 0.625502470
episode 9    survived 68   steps scored 0  memory len 2414 epsilon 0.617053222
episode 10   survived 115  steps scored 0  memory len 2529 epsilon 0.603022964
episode 11   survived 160  steps scored 0  memory len 2689 epsilon 0.584031710
episode 12   survived 87   steps scored 0  memory len 2776 epsilon 0.573957458
episode 13   survived 272  steps scored 1  memory len 3048 epsilon 0.543568253
```

The game launched in another window. I killed the process after a few episodes because waiting for it to finish 1000 would have taken a while.

6:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -d
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> print("hello world")
hello world
>>> dir()
['AI_SPEED', 'APPLE_REWARD', 'Apple', 'DEATH_PENALTY', 'DQPlayer', 'DQTrainer', 'FONT', 'FONT_SIZE', 'Game', 'Grid', 'HUMAN_SPEED', 'HumanPlayer', 'L_FILE', 'Leaderboard', 'M_FILE', 'M_LEFT', 'M_RIGHT', 'ROWS', 'SCREEN_SIZE', 'Snake', '__annotations__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'argparse', 'args', 'code', 'datetime', 'deque', 'environ', 'group', 'json', 'keras', 'np', 'parser', 'play_game', 'pygame', 'sleep']
>>>
```

The python dir() function returns a list of names in the current local scope.

7:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py --debug
Python 3.6.8 (tags/v3.6.8:3c6b436a57, Dec 24 2018, 00:16:47) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> import this
The Zen of Python, by Tim Peters
{
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 code has been run in both of these examples to demonstrate this is a full interactive python console.

8:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -r
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97
: calling GlorotUniform.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97
: calling Zeros.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *constraint arguments to layers.
2020-02-20 14:04:51.938202: E tensorflow/stream_executor/cuda/cuda_driver.cc:318] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
episode 0 survived 1129 steps and scored 27
1 episodes complete
max score of 27
average score of 27.0
```

The game launched in another window

9:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -r 3
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97
: calling GlorotUniform.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\init_ops.py:97
: calling Zeros.__init__ (from tensorflow.python.ops.init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the constructor
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *constraint arguments to layers.
2020-02-20 14:06:40.083194: E tensorflow/stream_executor/cuda/cuda_driver.cc:318] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
episode 0 survived 1037 steps and scored 27
episode 1 survived 529 steps and scored 1
episode 2 survived 1071 steps and scored 25
3 episodes complete
max score of 27
average score of 17.67
```

The game launched in another window

10:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -r help
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)
ai-snake.py: error: argument -r/--run: invalid int value: 'help'
```

11:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -p 5
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)
ai-snake.py: error: unrecognized arguments: 5
```


12:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -t 5
WARNING:tensorflow:From C:\Users\josep\venv\TF-venv\lib\site-packages\tensorflow_core\python\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
2020-02-20 14:10:34.319204: E tensorflow/stream_executor/cuda/cuda_driver.cc:318] failed call to cuInit: CUDA_ERROR_NO_DEVICE: no CUDA-capable device is detected
episode 0    survived 38    steps scored 0    memory len 38    epsilon 0.995072182
episode 1    survived 338   steps scored 0    memory len 376   epsilon 0.952295398
episode 2    survived 120   steps scored 0    memory len 496   epsilon 0.937554865
episode 3    survived 152   steps scored 0    memory len 648   epsilon 0.919210619
episode 4    survived 315   steps scored 0    memory len 963   epsilon 0.882329244
5 episodes complete, saving model...
```

The game launched in another window, then after 5 episodes was completed saved the 'trained' model and data to files.

13:

```
(TF-venv) PS C:\Users\josep\Dropbox\Code\ai-snake> python ai-snake.py -V
usage: ai-snake.py [-h]
                  (-t [n_games] | -r [n_games] | -p | -l [player name] | -d)
ai-snake.py: error: one of the arguments -t/--train -r/--run -p/--play -l/--leaderboard -d/--debug is required
```

14:

```
(TF-venv) PS C:\Users\josep\Dropbox\code\ai-snake> python ai-snake.py -l
No Entries!
```

Appendix C – Full code File

The code file for the ai-snake.py file is given in full, in text format here, for reference purposes. It may look slightly different to the screenshots because it had to be slightly reformatted to get into a document, so it's not exactly PEP-8 compliant, but all the lines of code are there.

```
from os import environ
environ["PYGAME_HIDE_SUPPORT_PROMPT"] = "hide"
environ["CUDA_VISIBLE_DEVICES"] = "-1"
environ["TF_CPP_MIN_LOG_LEVEL"] = '2'

import pygame
import numpy as np
from tensorflow import keras
from collections import deque
from datetime import datetime
from time import sleep
import json
import pickle as pkl

SCREEN_SIZE = 600
ROWS = 30
FONT = "comicsansms"
FONT_SIZE = 18
APPLE_REWARD = 100
DEATH_PENTALTY = -10
HUMAN_SPEED = 10
AI_SPEED = 100
DISCOUNT_FACTOR = 0.995
L_FILE = "leaderboard.json"
M_FILE = "AI-final.h5"
M_RIGHT = ((0, -1),
            (1, 0))
M_LEFT = ((0, 1),
           (-1, 0))

class Grid:
    def __init__(self, display, rows):
        self.rows = rows
        self.matrix = np.zeros((self.rows, self.rows))
        self.display = display
        self.screen_size = self.display.get_width()
        self.draw()

    def draw(self):
        for i in range(self.rows):
            start = (self.screen_size / self.rows * i, 0)
            end = (self.screen_size / self.rows * i, self.screen_size)
            pygame.draw.line(self.display, (0, 0, 0), start, end)
        for i in range(self.rows):
            start = (0, self.screen_size / self.rows * i)
            end = (self.screen_size, self.screen_size / self.rows * i)
            pygame.draw.line(self.display, (0, 0, 0), start, end)

    def update(self):
        for i in range(self.rows):
            for j in range(self.rows):
                colour = self.matrix[i][j]
                if colour == 0: # clear
                    colour = (255, 255, 255)
                elif colour == 1: # body
                    colour = (0, 255, 0)
                elif colour == 2: # head
                    colour = (0, 128, 0)
```

```

        elif colour == 4: # apple
            colour = (255, 0, 0)
            pygame.draw.rect(self.display, colour,
                             (self.screen_size / self.rows * i + 1,
                              self.screen_size / self.rows * j + 1,
                              self.screen_size / self.rows - 1,
                              self.screen_size / self.rows - 1))

    def clear(self):
        self.matrix = np.zeros((self.rows, self.rows))

    def setsq(self, xy, val):
        self.matrix[xy] = val

    def getval(self, xy):
        if (xy[0] < 0) or (xy[1] < 0) or (xy[0] > self.rows - 1) or (xy[1] > self.rows - 1):
            return -1
        else:
            return int(self.matrix[xy])

class Snake:
    def __init__(self, grid):
        self.grid = grid
        self.pos = (int(grid.rows / 2), int(grid.rows / 2))
        self.tail_len = 3
        self.body = []

    def draw(self):
        self.grid.setsq(self.pos, 2)
        for i in self.body:
            self.grid.setsq(i, 1)

    def move(self, direction):
        self.body.append(tuple(self.pos))
        self.body = self.body[-(self.tail_len):]
        self.pos = (self.pos[0] + direction[0], self.pos[1] + direction[1])

class Apple:
    def __init__(self, grid):
        self.grid = grid
        self.pos = (0, 0)
        self.repos()

    def repos(self):
        self.pos = (np.random.randint(0, self.grid.rows), np.random.randint(0, self.grid.rows))
        if self.grid.getval(self.pos) != 0:
            self.repos()

    def draw(self):
        self.grid.setsq(self.pos, 4)

class Game:
    def __init__(self, rows, screen_size):
        pygame.init()
        self.display = pygame.display.set_mode((screen_size, screen_size))
        pygame.display.set_caption("snek")
        self.display.fill((255, 255, 255))
        pygame.font.init()
        self.font = pygame.font.SysFont(FONT, FONT_SIZE)
        self.grid = Grid(self.display, rows)
        self.new_game()

    def get_state(self):
        item_left = self.grid.getval((self.snake.pos[0] + self.direction[1],
                                       self.snake.pos[1] - self.direction[0]))

```

```

        item_right = self.grid.getval((self.snake.pos[0] - self.direction[1],
                                         self.snake.pos[1] + self.direction[0]))
        item_front = self.grid.getval((self.snake.pos[0] + self.direction[0],
                                         self.snake.pos[1] + self.direction[1]))

    if item_left == 0:
        obstacle_left = 0
    elif item_left == 4:
        obstacle_left = 1
    else:
        obstacle_left = -1
    if item_right == 0:
        obstacle_right = 0
    elif item_right == 4:
        obstacle_right = 1
    else:
        obstacle_right = -1
    if item_front == 0:
        obstacle_front = 0
    elif item_front == 4:
        obstacle_front = 1
    else:
        obstacle_front = -1
    pos_x = (self.snake.pos[0] - self.apple.pos[0])
    pos_y = (self.apple.pos[1] - self.snake.pos[1])
    return (obstacle_left, obstacle_right, obstacle_front, pos_x, pos_y,
            self.direction[0], -self.direction[1], self.snake.tail_len)

def draw_score(self):
    self.display.blit(self.font.render(str(self.score), True, (0, 0, 0)), (5, -2))

def new_game(self):
    self.grid.clear()
    self.score = 0
    self.done = False
    self.direction = (0, 1)
    self.prev_direction = self.direction
    self.snake = Snake(self.grid)
    self.snake.draw()
    self.apple = Apple(self.grid)
    self.apple.draw()
    self.grid.update()
    pygame.display.update()

def step(self, action):
    if action == 0: # carry on
        self.direction = self.prev_direction
    elif action == 1: # turn right
        self.direction = tuple(np.dot(M_RIGHT, self.direction))
    elif action == 2: # turn left
        self.direction = tuple(np.dot(M_LEFT, self.direction))
    self.snake.move(self.direction)
    self.prev_direction = self.direction
    # check if snake died based upon the movement taken
    if self.grid.getval(self.snake.pos) == -1:
        self.done = True
    else:
        for n in self.snake.body:
            if n == self.snake.pos:
                self.done = True
    # if dead, reset game
    apple_got = False
    if self.done:
        return (apple_got, True, self.score)
    # if snake at the apple, make a new apple

```

```

        if self.snake.pos == self.apple.pos:
            self.apple.repos()
            self.score += 1
            self.snake.tail_len += 1
            apple_got = True
        # draw and update everything
        self.grid.clear()
        self.snake.draw()
        self.apple.draw()
        self.grid.update()
        self.draw_score()
        pygame.display.update()
        return (apple_got, self.done, self.score)

class play_game(Game):
    def __init__(self, rows, screen_size, speed):
        super().__init__(rows, screen_size)
        self.speed = speed
    def new_game(self):
        super().new_game()
        self.clock = pygame.time.Clock()
    def step(self, action):
        a = super().step(action)
        self.clock.tick(self.speed)
        return a

class Leaderboard:
    def __init__(self, filename):
        self.filename = filename
        try:
            with open(self.filename, "r") as f:
                self.scores = json.load(f)
        except FileNotFoundError:
            with open(self.filename, "w") as f:
                json.dump({"all": []}, f)
            with open(self.filename, "r") as f:
                self.scores = json.load(f)

    def test(self, name, score):
        if name not in self.scores:
            return True
        elif len(self.scores["all"]) < 20:
            return True
        elif len(self.scores[name]) < 10:
            return True
        elif self.scores["all"][19][1] < score:
            return True
        elif self.scores[name][9][0] < score:
            return True
        else:
            return False

    def add(self, name, score):
        name = name.lower()
        if name == "all":
            raise ValueError("name cant be 'all'")
        if self.test(name, score):
            time = datetime.now().strftime("%d/%m/%y, %H:%M ")
            if name in self.scores:
                self.scores[name].append((score, time)) # add
                self.scores[name].sort(key=lambda x: x[0], reverse=True) # sort
                self.scores[name] = self.scores[name][:10] # chop
            else:
                self.scores[name] = [[score, time]]
            self.scores["all"].append((name, score, time))

```

```

        self.scores["all"].sort(key=lambda x: x[1], reverse=True)
        self.scores["all"] = self.scores["all"][:20]

def display(self, name="all"):
    name = name.lower()
    if name not in self.scores:
        raise ValueError("That player does not exist")
    if len(self.scores[name]) == 0:
        print("No Entries!")
        return
    if name == "all":
        print(" ---- Top Scores All Time ----")
        for n, s, d in self.scores["all"]:
            print("{:<5} - {:<2d} - {}".format(n, s, d))
    else:
        print(" ---- Top Scores For " + name.capitalize() + " ----")
        for s, d in self.scores[name]:
            print("{:<2d} - {}".format(s, d))

def save(self):
    with open(self.filename, "w") as f:
        json.dump(self.scores, f)

class DQPlayer:
    def __init__(self, filename):
        self.net = keras.models.load_model(filename)
        self.l = Leaderboard(L_FILE)
        self.loop_threshold = 300

    def act(self, state):
        Q_vals = self.net.predict(np.array([state]))[0]
        action = np.argmax(Q_vals)
        return action, list(Q_vals)

    def play(self, n_episodes):
        env = play_game(ROWS, SCREEN_SIZE, AI_SPEED)
        tracker = []
        for e in range(n_episodes):
            dead = False
            state = env.get_state()
            steps = 0
            prev_score = 0
            steps_since_score = 0
            second_Qs = []
            target_Q = None
            while not dead:
                pygame.event.pump()
                action, Q_vals = self.act(state)
                if target_Q in Q_vals:
                    action = Q_vals.index(target_Q)
                    target_Q = None
                    second_Qs = []
                _, dead, score = env.step(action)
                if score == prev_score:
                    steps_since_score += 1
                else:
                    steps_since_score = 0
                prev_score = score
                state = env.get_state()
                steps += 1
            if steps_since_score > self.loop_threshold:
                second_Qs.append(sorted(Q_vals)[1])
                if steps_since_score > self.loop_threshold + 100:
                    target_Q = np.max(second_Qs)
            if steps_since_score > self.loop_threshold * 5:

```

```

        dead = True
        tracker.append((score, steps))
        print("episode {:<3d} survived {:<4d} steps and scored {:<2d}"
              .format(e, steps, score))
        env.new_game()
        self.l.add("AI", score)
    print(str(n_episodes) + " episodes complete")
    tracker = np.array(tracker)
    score, steps = np.hsplit(tracker, 2)
    print("max score of {:2d}".format(np.max(score)))
    print("mean score of {:2.1f}".format(np.mean(score)))
    print("median score of {:2.1f}".format(np.median(score)))
    self.l.save()

class HumanPlayer:
    def __init__(self):
        self.game = play_game(ROWS, SCREEN_SIZE, HUMAN_SPEED)
        self.score = 0
    def game_over(self):
        self.game.display.blit(self.game.font.render("G A M E    O V E R", True,
                                                       (0, 0, 0)), (230, 220))

        pygame.display.update()
        sleep(2)
        self.game.display.blit(self.game.font.render("Score - " + str(self.score),
                                                       True, (0, 0, 0)), (255, 250))

        pygame.display.update()
        sleep(2)
        pygame.quit()
        name = input("enter your name >>>")
        l = Leaderboard(L_FILE)
        l.add(name, self.score)
        l.save()

    def play(self):
        dead = False
        keypress = None
        while not dead:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    exit()
                if event.type == pygame.KEYDOWN:
                    if event.key == pygame.K_a:
                        keypress = (-1, 0)
                    if event.key == pygame.K_d:
                        keypress = (1, 0)
                    if event.key == pygame.K_w:
                        keypress = (0, -1)
                    if event.key == pygame.K_s:
                        keypress = (0, 1)
            left = tuple(np.dot(M_LEFT, self.game.direction))
            right = tuple(np.dot(M_RIGHT, self.game.direction))
            if keypress == right:
                action = 1
            elif keypress == left:
                action = 2
            else:
                action = 0
            _, dead, self.score = self.game.step(action)
            if dead:
                self.game_over()

class DQTrainer:
    def __init__(self, model=None):
        self.state_size = 8
        self.n_actions = 3

```

```

self.gamma = DISCOUNT_FACTOR
self.e_decay = -0.00013
self.batch_size = 64
self.min_mem_length = 1000
self.replay_mem = deque(maxlen=10000)
if model:
    self.epsilon = 0
    self.active_net = keras.models.load_model(model)
else:
    self.epsilon = 1
    self.active_net = self.build_net()
self.target_net = self.build_net()

def build_net(self):
    net = keras.models.Sequential()
    net.add(keras.layers.Dense(12, input_dim=self.state_size, activation="relu"))
    net.add(keras.layers.Dense(self.n_actions, activation="linear"))
    net.compile(loss="mse", optimizer="Adam")
    return net

def update_target(self):
    self.target_net.set_weights(self.active_net.get_weights())

def remember(self, state, action, reward, next_state, done):
    self.replay_mem.append((state, action, reward, next_state, done))
    self.epsilon *= np.exp(self.e_decay)

def act(self, state):
    if np.random.rand() ≤ self.epsilon:
        action = np.random.randint(self.n_actions)
    else:
        Q_vals = self.active_net.predict(np.array([state]))[0]
        action = np.argmax(Q_vals)
    return action

def learn(self):
    if len(self.replay_mem) > self.min_mem_length:
        batch = np.array([self.replay_mem[x] for x in
                           np.random.randint(len(self.replay_mem),
                           size=self.batch_size)])
        states = np.zeros((self.batch_size, self.state_size))
        next_states = np.zeros((self.batch_size, self.state_size))
        actions, rewards, dones = [], [], []
        for n in range(self.batch_size):
            states[n] = batch[n, 0]
            actions.append(batch[n, 1])
            rewards.append(batch[n, 2])
            next_states[n] = batch[n, 3]
            dones.append(batch[n, 4])
        target = self.active_net.predict(states)
        target_next_active = self.active_net.predict(next_states)
        target_next_target = self.target_net.predict(next_states)
        for n in range(self.batch_size):
            if dones[n]:
                target[n][actions[n]] = rewards[n]
            else:
                a = np.argmax(target_next_active[n])
                target[n][actions[n]] = rewards[n] + self.gamma * target_next_target[n][a]
        self.active_net.fit(states, target, batch_size=self.batch_size,
                             epochs=1, verbose=0)

def train(self, n_episodes, max_steps=1000):
    env = Game(ROWS, SCREEN_SIZE)
    data = []
    for e in range(n_episodes):

```



```

dead = False
score = 0
state = env.get_state()
steps = 0
while True:
    pygame.event.pump()
    action = self.act(state)
    apple, dead, _ = env.step(action)
    if dead:
        reward = DEATH_PENTALTY
    elif apple:
        reward = APPLE_REWARD
        score += 1
    else:
        reward = 0
    next_state = env.get_state()
    self.remember(state, action, reward, next_state, dead)
    state = next_state
    steps += 1
    self.learn()
    if dead or (steps ≥ max_steps):
        self.update_target()
        env.new_game()
        print("episode {0:<4d} survived {1:<4d} steps scored {2:<2d}
              memory len {3:<4d} epsilon {4:<6.9f}"
              .format(e, steps, score, len(self.replay_mem), self.epsilon))
        data.append((e, steps, score, self.epsilon))
        break
print("{} episodes complete, saving model...".format(n_episodes))
timestamp = datetime.now().strftime("%d-%m-%H-%M")
self.target_net.save("model-" + timestamp + ".h5")
with open("data-{}.pkl".format(timestamp), "wb") as f:
    pkl.dump(data, f)

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="AI Snake")
    group = parser.add_mutually_exclusive_group(required=True)
    group.add_argument("-t", "--train", type=int, metavar="n_games", nargs="?", const="1000",
                      help="train a new model, specify number of
                           games to train with, default 10000")
    group.add_argument("-r", "--run", type=int, metavar="n_games", nargs="?", const="1",
                      help="run some games with a trained model, specify number of
                           games to run, default 3")
    group.add_argument("-p", "--play", action="store_true", help="play the game yourself")
    group.add_argument("-l", "--leaderboard", type=str, metavar="player name", nargs="?",
                      const="all", help="view the leaderboard,specify players scores to
                           view, displays all by default")
    group.add_argument("-d", "--debug", action="store_true",
                      help="run in interactive mode for debugging")
    args = parser.parse_args()
    if args.play:
        p = HumanPlayer()
        p.play()
    elif args.leaderboard:
        l = Leaderboard(L_FILE)
        try:
            l.display(name=args.leaderboard)
        except ValueError:
            print("that player does not exist!")
    elif args.run:
        r = DQPlayer(M_FILE)
        r.play(args.run)
    elif args.train:
        t = DQTrainer()

```

```
    t.train(args.train)
elif args.debug:
    import code
    code.interact(local=locals())
```