School of Computing
FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

Creating and Implementing AI BOT using neural networks to perform in a given environment

Shreyas Milind Bhalerao

Submitted in accordance with the requirements for the degree of
MSc Advanced Computer Science (Artificial Intelligence)

2022/2023

The candidate confirms that the following have been submitted*:*

Items	**Format**	**Recipient(s) and Date**
Project Report	*Report*	*SSO (25/08/23)*
Source code	*URL :* *https://github.com/Shreyascreate/MSc_Final_dissertation*	*Supervisor, assessor* *(25/08/23)*

Type of Project: Exploratory Software

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

Shreyas Milind Bhalerao
(Signature of student)

Summary

This project is an engaging endeavour focused on recreating the timeless appeal of the traditional Snake game made with the use of Pygame library and implementing an AI Bot to control the Snake. This is achieved by creating a neural network which has its roots in Reinforcement learning principles, particularly in the Q-learning algorithm. This algorithm guides the snakes by giving incentives on performing actions which in result helps the snake to reach its goals. The neural network consist of 1 input layer, 2 hidden layers and 1 output layer. The neural network takes 11 inputs which helps the neural network to understand the surroundings and based on these inputs the neural network guides the snake to the goal.

The main goal of the project is to revive the nostalgia of the iconic arcade snake game but to add a layer of complexity and a touch of modern neural networks through AI driven automated gameplay. This report introduces the relevant research required by the project followed by creation and implementation of the neural network into the snake game and finally experimentations on the variations of the neural networks while analysing other designs of neural networks for achieving the specified goals.

This project will use a DQN architecture, as the main model for achieving its goal. The DQN architecture uses memory buffers to store its experiences, which it then uses in two types of buffers, short-term-memory, and long-term memory. The short-term memory uses the models' most recent experiences to train the model, whereas the long-term memory can be visualised as the cumulative of experiences of model over many iterations. To procure aforementioned techniques, the DQN architecture employs Bellman equation which calculates the reward for the current step and next steps based on the previous experiences of the model.

The project will also evaluate and compare various variations of the neural networks during the testing phase, to evaluate the performance of the chosen model with others and also will be glancing over the strengths and weaknesses of each model compared.

Acknowledgements

Firstly, I would like to express my gratitude towards my supervisor, Dr. Martin Callaghan for his constant support, patience, and encouragement throughout the process. His advice during the meeting gave me a lot of insights and showed me various angles in which things can be done. His knowledge gave me a lot of insights that helped me during my research.

My sincere thanks also go to my assessor, Dr. Mark Walkley, for his valuable feedback and recommendations during the process. He helped me by showing various ways in which my project could be improved and by recommending various tests that could be done to improve my report.

Finally, I would like to thank my family and friends who helped me during these stressful and frustrating times, they always encouraged me and stood by me whenever I felt nervous. They are the ones who kept me going and motivated me throughout the process.

Table of Contents

List of Tables

List of Abbreviations

NPC	Non-Playable Character
AI	Artificial Intelligence
CPU	Central processing unit
FSM	Finite state machine
DQN	Deep Q-Network
PIP	Python Package Installer
GPU	Graphics processing unit
ReLU	Rectified linear unit
Adam	Adaptive Moment Estimation

Chapter 1
Introduction

This Chapter begins with an explanation of the motivation behind this project, followed by the aim and objectives. It helps to get the concept and define the problem for this project. Next, it discusses the possible solutions for our problem, making us aware of the various ways to tackle such problems. Finally, it addresses the deliverables and academic relevance of the project and concludes with an overview of the rest of the report.

1.1 Motivation

Classic arcade games like Snake Game evoke a sense of nostalgia and recollection of fond memories, giving a glimpse into the gaming of the past. However, in an era of rapid technology, there is a captivating opportunity to infuse our cherished classic games with modern AI tools like neural networks and give them a modern essence. The integration of AI-driven gameplay plays as the central motivation to creating this project. In a world where artificial intelligence is reshaping various industries, the gaming domain provides a remarkable insight to show its potential. Today a lot of organisations related to gaming use AI to achieve various tasks, creating bots for the game to recreate non-human characters or NPC which helps the companies to recreate the feel of the real world.

The project shows the potential that modern neural networks hold to a certain extent, with a demonstration in the form of its application on the classic snake game and will also help understand the other applications of such form. The surge in AI has people excited and welcoming to make the digital world more real-like. According to a report by PocketGamer.biz "More than 78% of gamers would talk to NPCs more if they were smarter" (Aaron Astle, Pocket Gamer, 2023). This project will be created, keeping the inputs as human as possible to capture the feelings of the mass gamers.

1.2 AIM

This project aims to develop a neural network that can play a snake game, by giving the game inputs depending on the environment while keeping the inputs as humanly as possible to achieve its goal, which is to eat the food. Since it replicates human input reinforcement learning deep-Q learning is more suitable than traditional machine learning algorithms like decision trees.

The Project will be using the deep-Q network's ability to train its neural network using the replay memory buffer. The DQN uses two types of memory to form a replay memory; short-term memory and long-term memory. The short-term memory also “Replay Memory” stores the model’s most recent experiences, whereas the long-term memory can be thought of as, a cumulative of the model’s experiences over many iterations. To achieve this DQN uses the Bellman equation, which calculates the Q-values which are basically the new moves based on the model’s previous experience and the rewards the model gets.

1.3 Objectives

To achieve the aim of this project the following objectives must be completed:

1. Set up the environment of our game to automate it.
2. Study the game and the way it handles inputs.
3. Based on the game design the input of the neural network.
4. Create and develop the neural network.
5. Test the neural network and compare the variations of the network to better suit the implementation.
6. Fine-tune the best neural network of the variations.

1.4 Possible Solution

This project can be tackled in various methods like neural networks, decision trees, etc. The mentioned two are the most compatible ones. The fundamental difference between these two implementations is the neural network works on a probabilistic approach whilst the decision trees work on a deterministic approach. The differing approaches make them viable for different implementations. A deterministic

approach will always disregard any other data that does not suit its current situation, thus resulting in a loss of important data. A probabilistic approach works based on probability generated in the current situation and will not disregard any data that it thinks is not important at the given moment. A deterministic approach is more viable in a scenario where the path of the NPC is predetermined or has a set number of options to choose from, which doesn't consider changes in the environment.

A probabilistic approach is viable in a scenario where the outcome is not predetermined or there are not a set number of outcomes, and the environment is constantly changing. Modern games that are mimicking today's world are increasingly implementing a probabilistic approach for the same reason as the situation in an environment can change and to mimic the real world the NPCs must react accordingly thus making a deterministic approach less practicable. More complex environments can deploy both methods simultaneously on different parts of environments like an NPC crashing a car; where an NPC can be controlled by a neural network whereas the physics of the crashed car can be controlled by a decision tree. As the NPC would react to the situation in a humanly manner, where they will address to the changing conditions, they will be needing a probabilistic approach, however, since the car can only react in a certain fixed ways and will just be dented in a particular manner, it can be handled by a deterministic approach. But in the case of this project since we are just replicating a human input, the most feasible way to implement this is by neural networks.

1.5 Deliverables

This project will produce the following deliverables:

1. A fine-tuned neural network that reacts to its changing environments and will produce humanlike inputs to the snake game, built using the PyTorch library of Python.
2. Design for the neural network implemented in the game.
3. Code for the neural network and the implemented game.
4. Final project report

1.6 Academic Relevance

To achieve the project's main aim, knowledge from several fields is required. The main required knowledge was acquired from COMP5625M Deep Learning which helped in acquiring the essential proficiency of neural networks. The module helped

in get a clear concept of the workings of neural networks and their implementation. In addition, it helped the author in procurement of the understanding of ways in which neural network makes decision; how different types of neural network approach the same problems in different ways and what significance the architecture of neural network hold. COMP5611M Machine Learning provided the understanding of different ways in which these types of problems can be tackled for example decision trees. Moreover, it also provided an understanding of what different fundamentals neural networks and decision trees apply to reach a goal state and how they differ. COMP5930M Scientific Computation helped me understand the fundamental algorithms like Gradient descent, Newton method, etc. which neural networks apply to make decisions for a given set of inputs. It also gave an in-depth understanding of the difference the algorithms make when applied to the same scenario which helped in understanding different techniques a neural network can apply to get desired results.

1.7 Report Structure

The following section provides an overview of how this project was managed and gives a step-by-step explanation of how this project was created. Chapter 2 discusses the overall background research required for this project to be completed which includes the base knowledge, the fundamental ideas, and different types of neural networks. It also examines the past research done on these topics and considers' their relevance to this project. Chapter 3 analyses the overall system design which includes software requirements for the project and will also discuss the implemented software in this project. Moreover, this chapter also talks through the system design that is being used in this project by explaining the chosen design of the neural network. Chapter 4 will discuss about the Implementation of the chosen design and software in this project and will justify the choices made during the creation of the project. Chapter 5 will discuss the results procured during the creation of software and will compare it to the other variations on neural networks. Finally, Chapter 6 will provide the conclusions obtained from this project and will glance over the future scope of the project and possible expansions that can be done to the project.

Chapter 2
Background Research

2.1 Literature Survey

The literature survey begins with a summary of the history and current state of AI in games. Then it discusses about the previous research and articles that were published considering the use of AI in games and how it has impacted the gaming industry. The mentioned section provides the research and articles in a section form and gives an overview of the contents.

2.1.1 History and current state

The roots of AI in video games can be found as early as 1980s in the games like “PAC-MAN” where a simple decision tree was being used to control the movements of ghosts in the game. Most of the early games implemented in the 1980s and 1990s were mostly using decision trees or simple if and else statements to give the user a feel of computer-controlled bots to make the game more engaging. Some of the first modern implementations of AI in games can be seen in games like Need for Speed, Gran Turismo, etc. More sophisticated AI models were being used to control NPCs' cars and gave them a more real race-like feel while accommodating every player regardless of their skill group, as it infused different AI models for different skill groups. Today AI is being used for almost everything in game worlds, for example, games like Grand Theft Auto use AI to control the NPCs to replicate the world, they also use AI to smartly render the world, reducing the load on the CPU and saving memory while doing so. AI has come a long way in the last 40 years they started with simple decision trees to sophisticated neural networks to replicate human responses to a situation or the current environment.

The implementation of modern AI techniques in games has made the game worlds more engaging and feel more realistic regardless of the genre and the style but there is still a lot of potential for AI in-game as the industry is a long way from reaching the true potential of AI.

2.1.2 Previous Research

2.1.2.1 Artificial Intelligence in Video Games(Andrew Mikhailov, Experfy, 2020)

The article written by Andrew Mikhailov for Experfy in the year 2020 provides an overview of how artificial intelligence is being used in video games and environments to provide players with a much more realistic and immersive experience. It starts by providing a brief history and evolution that was done in the field of AI from a simple rule-based system that followed predefined scrips and behaviours, to a very complex neural network and machine learning algorithms that will adapt to the changes in environments and situations. Further, it discusses the challenges and opportunities that AI provides to video game design and development, like balancing the difficulty and believability of the AI-controlled NCPs as allies and opponents, enhancing the interactivity and personalization of the game world and the player's experience. It also discusses the future application of AI in video games, such as how AI-generated content, like levels, items, and characters based on the main frame of the game can create a dynamic story-telling and can impact the narratives of the games based on the dynamic and changing environments and can change according to the players' response and choices. Moreover, it provides a brief look over, how AI can adjust gameplay and difficulty according to the player's skill level or preferences; and how AI can create social and emotional NPCs that can interact with the player naturally and expressively making the player much more immersed in the game. Finally, the author adds, how AI can create immersive environments that can simulate realistic sensations with the help of virtual reality. The article concludes that AI is a powerful tool for video game development, and it will continue to shape the future of the industry and will provide a much more realistic and immersive experience to the players.

2.1.2.2 Application of Neural Network for Intelligent Video Game Character Artificial Intelligence (Simon D. Mikulcik, Eastern Kentucky University, 2016)

The research paper by Simon D. Mikulcik investigates the application of neural networks for creating intelligent video game characters artificial intelligence (AIs). The research paper begins by providing existing techniques that video game development companies use to implement AI in their games, such as FSM (Finite state machines), behaviour trees, decision trees, etc. These techniques are mainly based on some predefined rules and conditions that will impact the state of the game

characters. Further, it discusses the use of neural networks in video games, in doing tasks like pathfinding, learning, decision making, etc. and gives some examples of games that use neural networks for certain tasks in the games such as “Black & White”, “Creatures” and “NERO”. The paper then describes a project which was made to study and analyse the neural network-controlled characters. The game used is a top-down 2-D shooter that has two teams zombies and soldiers, in which the zombies are controlled by a feedforward neural network that takes input from the game environment such as distance to the soldiers and the angle to the soldiers, and outputs the movement decision and attacking. The soldiers are controlled by a recurrent neural network that takes the input of the environment from its memory like the distance to the zombies, the soldier's health, and the angle to the zombies. Then the paper evaluates the results by comparing the performance and behaviour of the two different types of neural networks with a rule-based behaviour system. It uses several metrics to measure the performance of the neural networks such as accuracy, kill ratio, survival time, and unpredictable actions made by the neural network. Finally, it concludes that the neural network-based characters can provide much more realistic behaviours and can adapt to the changing environments more realistically, it discusses some challenges as well which the neural network-based character poses like training, balancing, and debugging.

2.1.2.3 How to teach AI to play Games: Deep Reinforcement Learning (Mauro Comi, Towards Data Science, 2018)

This article explains how to teach an AI to play games using deep reinforcement learning, which combines deep neural networks with reinforcement learning. One of the algorithms that the article covers is DQN, which stands for Deep Q-Network, which learns the value of each state-action pair. The neural network predicts expected rewards by taking a certain action for each given state. When the state changes the neural network updates the value function by using the bellman equation shown in Figure 2.1, which states that the value of the state-action pair is equal to the immediate reward plus the discounted value for the next state-action pair. The issue with the DQN is that the algorithm stores the results in a table format which is a problem when considering a larger state and action spaces, this is solved by using a deep neural network that approximates the value function. The neural network is trained by a technique called experience replay where the neural network uses the memory that it learns from the previous iterations which are in a buffer and are randomly sampled to update the neural network weights.

$$NewQ(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max\, Q'(s',a') - Q(s,a)]$$

New Q-Value | Current Q-Value | Learning rate | Reward | Discount rate | Maximum predicted reward, given new state and all possible actions

Figure 2.1: Bellman equation (Mauro Comi, 2019)

Then the article implements the DQN with a neural network in an example snake game, where it uses a Deep Q-Network to control the snake in the example game. The game uses a neural network that has 1 input layer, 3 hidden layers of size of 120 neurons and 1 output layer of size 3 and uses the Bellman equation to approximate the size of the value function. Then they trained the neural network for 150 games and according to the observations the snake at early iterations performed poorly and was able to score only 2-3 points per game, but as the neural network started to learn it developed a good strategy and by the end of its 150 iterations it was able to score 45 points (this is demonstrated in figure 2.2)

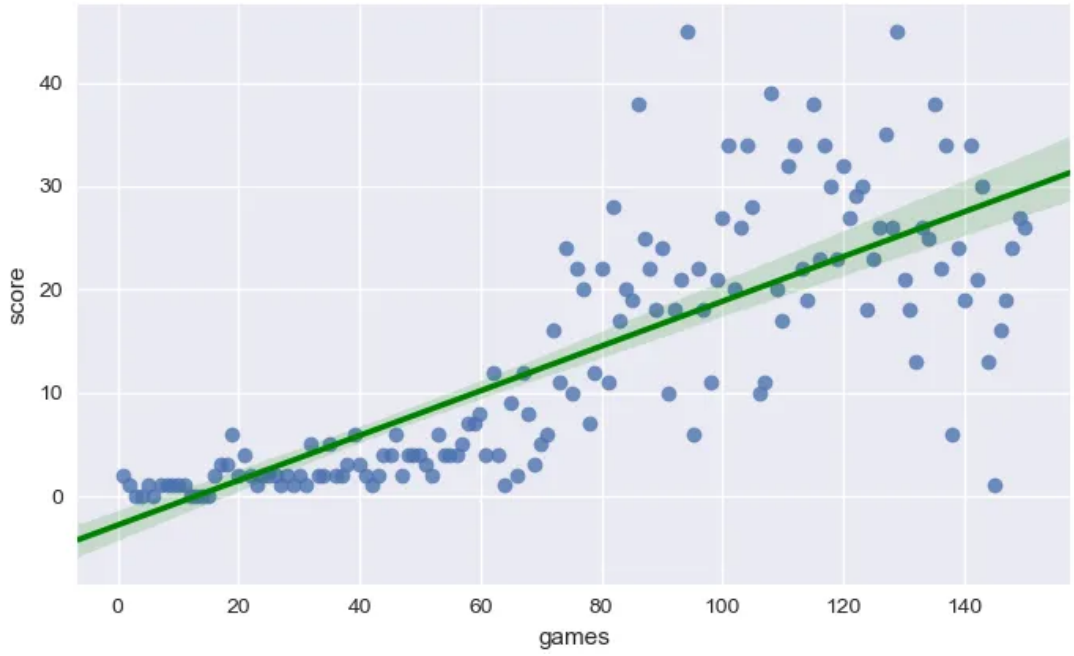

Figure 2.2: Progress over the iterations of games (Mauro Comi, 2019)

Finally, the article concludes with an explanation of the result and also introduces the idea of using Double Deep Q-Learning instead of Deep Q-Learning as it can perform better and can give more precise convergence.

2.3 Methods and Techniques

This section briefly explains the available techniques that are being used to solve similar types of problems, which explains some of the techniques that can be used for this project.

A project where human input is being automated can be achieved through “Decision trees”, "Reinforcement Learning", "Policy Gradient Methods", "Q-Learning", "Imitation learning" etc. Most popular of these are decision trees, reinforcement learning applied in Q-Learning (DQN) and imitation learning. The decision trees follow a deterministic approach that follows a predefined set of rules or paths, without considering the changes in the environment, or will not consider any other data that is available on the algorithm during a certain state. A reinforcement learning applied with Q-Learning commonly known as Deep Q-Learning or DQN uses a neural network to get the data of surroundings and generates an approximation of the value of the state-action pair. This helps the neural network to generate combinations of possibilities that can be applied to create an optimal strategy for the neural network. DQN also uses its memory to learn from its previous iterations and uses them in buffers by randomly sampling them to train the network which helps the neural network to converge more accurately. Imitation Learning is a method where the network imitates a human or expert and replicates the movements or inputs made by them. It's a technique where instead of learning from scratch through trial and error, the algorithm leverages the data from existing expert knowledge to accelerate the learning process and improve performance. This is achieved by a neural network also referred to as a "policy network" or "imitation network", which is trained to directly map states to the actions based on the expert's actions in the dataset. The challenge with these types of networks is that the network can end up in a state where there is no data point in the experts' database thus resulting in poor performance. Another major challenge that these types of neural networks face is that the quality of the expert greatly influences the success of imitation learning, so if the expert's behaviour is suboptimal then the results generated by the network will be suboptimal degrading the quality of the overall network.

Overall, these are the potential solutions that can be used to solve the kind of problems that this project is aiming to tackle. Every solution has its positives and negatives, giving every solution a unique advantage over the other respective methods.

2.4 Choice of Methods

The methods discussed in section 2.3 provide a good set of options to choose from, but for this project, the most optimal solution that fits the chosen environment is DQN. DQN provides a unique set of advantages in this environment, when compared to others, by allowing the neural network to learn on itself to navigate through the space and find the optimal path towards the goal. While decision trees and imitation learning are also very compelling options, decision trees since being deterministic in nature, disregard data that they think are not optimal or useful while considering for output. Since they also follow a set of rules to generate outputs or predictions and do not adapt according to the changing environment, they do not comply with the aim of this project "of keeping them as human as possible", as humans do not make decisions based on a certain set of rules and changing environments play a vital role in the decision-making which decision trees do not consider. Imitation learning though makes up for the disadvantages posed by decision trees, like they make decisions based on the changing environment, and they do not use a set of rules for generating outputs or making decisions, they pose a different disadvantage of having an expert database on which they train. This makes them repetitive in nature, in addition, if the network gets into a stage where there is no data point in the expert database while doing behavioural cloning, the performance of the network can drop significantly. Since this project is based on the fact that the snake food can appear randomly and cannot be predetermined, the performance drop can be a significant threat to the consistency of the neural network. These problems can be solved by DQN as DQN does not rely on a predetermined set of rules or regulations while making decisions and it does not require a pre-existing expert dataset to train on as DQN trains itself and can also adapt to the changing environment while keeping the consistency during the random placement of the goals. DQN is the optimal option which aligns with the aim of this project of "automating the input for the snake while keeping the inputs as human as possible".

Chapter 3
Software Requirements and System Design

3.1 Software Requirements

This section will cover and discuss the software and libraries that are required for this project and will further discuss the usage and a brief background of the software. This section will provide us with the required knowledge and understanding of required technologies for this project.

3.1.1 Python

Python is a high-level, versatile, easy-to-read programming language created and designed by 'Guido van Rossum' in the year 1991. It is a multi-paradigm programming language as it fully supports object-oriented and structured-oriented programming. The versatility of this programming language is what helps developers add more functions to the language by adding modules to their codes and using it for development.

Other alternative languages were also considered for this project, for instance, Java (Java, 2023), C (JTC1, 2021), C++ (ISOCPP, 2023), etc. However due to the abundance of documentation, forums, tools, and modules available for Python, plus the additional support to the new standards and new methods, Python was preferred for this project rather than the other alternatives. The popularity of Python can be seen in Figure 3.1 from a study published by Sarah Feldman for 'Statista' (Sarah Feldman, Statista, 2019). The survey was done based on Google searches done for programming language tutorials and this was quoted in the study "This is a 5 percent increase when compared to a year ago, showing continued and growing interest in the language" (Sarah Feldman, Statista, 2019). The popularity and the continued support of the language by the core Python developers and module developers made Python the ideal language.

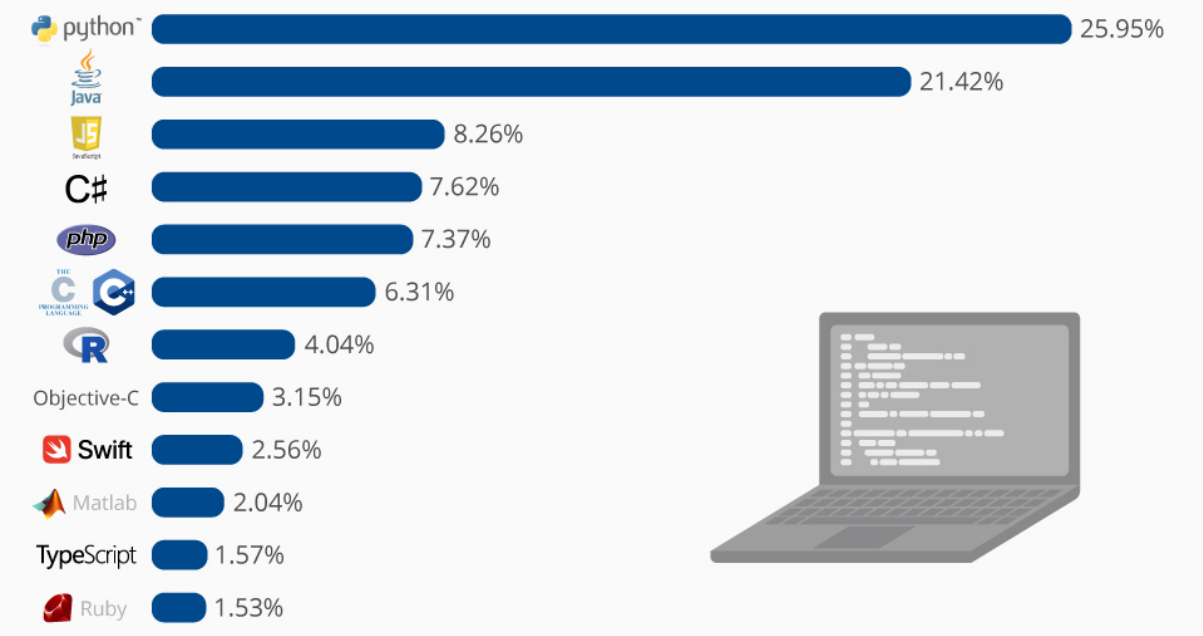

Figure 3.1: Most popular programming language (Statista, 2019)

This project also uses Python as the base language for its development and will be using various modules to add functionalities, which will be discussed further. This project will be using a stable release of Python version 3.9.10 (python, 2022).

3.1.2 Anaconda

Anaconda (Anaconda, 2023) is a distribution of Python programming language which is popular amonst machine learning developers, analysts, data scientists. This module includes essential Python libraries with additional libraries like NumPy (NumPy, 2023) and Pandas (Pandas, 2023) and packaging manager, which are essential in machine learning and data analysis. Anaconda also allows to manage various Python environments for development, which helps to keep the work-environment clean for each project.

This project will be using Anaconda for its major packages and environment handling as it provides more functionality and a cleaner workspace for the project. This project will be using Anaconda version 4.11.0 (Anaconda, 2022) as it stable version supported by major libraries and also supports Apple Silicon M1 (Apple, 2023) as the workdesk is based on the mentioned architecture.

3.1.3 PIP

Python Package Installer also known as PIP (PyPI, 2023) is a command line package installer for Python used to install and manage Python libraries. It allows

users to install Python packages easily from PyPI or any other source, in addition, it also allows its users to upgrade or uninstall Python packages, while handling their dependencies for their Python project. PIP is considered an essential tool for maintaining clean and organised Python development environments.

3.1.4 Pygame

Pygame (Pygame, 2023) is a cross-platform library for creating 2-D video games and multimedia applications using Python programming language. It provides various functionalities that are required to make video games such as graphics, sound, keyboard inputs to the game, etc. This package will be used by this project to run the base snake game. There are various alternatives for creating 2-D games in Python like Python Arcade (Python Arcade, 2023), and Cocos2D (Cocos, 2023). However due to the wide use of Pygame game, ease of use and simplified mechanisms, it was preferred to use Pygame instead of the alternatives.

3.1.5 PyTorch

PyTorch (PyTorch, 2023) is an open-source machine learning developed by Facebook's AI Research lab, it is used for Deep Learning due to the optimised tensor library. PyTorch provides an essential tool required for the development of neural networks as the library offers features like tensor computation which uses its powerful tensor computation library, allowing developers and users to do complex computation tasks easily. PyTorch also provides a 'Dynamic Computational Graph' which allows developers to modify the computation graph during runtime making debugging and experimenting much easier.

Other libraries like TensorFlow (TensorFlow, 2023) and scikit-learn (scikit-learn, 2023), were also considered for use in this project, but because of TensorFlow's dependencies issue and complex nature, it was not optimal for this project. Though TensorFlow is one of the most used and is well supported, but during the time of making this project there was no official support of TensorFlow to Apple silicon-based Macs, further restricting the usage. Scikit-learn being one of the easiest-to-use libraries does not provide the functionality and performance when compared to PyTorch in terms of GPU Acceleration which affects the runtime of the algorithm. Therefore, PyTorch was used in this project, to develop and handle the neural network.

3.2 System Design

The system design involves representations of neural network designs including the design behind the inputs given to the snake and the design for the actual neural network which is used in this project. As the main aim of this project was to automate the traditional snake game, the traditional snake game was built from the reference taken from GitHub (GitHub, 2023) repository of Patrick Loeber (Patrick Loeber, 2020)

3.2.1 Input for Neural Network

The input for the neural network used in this project is based on the current state of the snake in the game, by considering the current position of the snake, the dangers that surround the snake and the position of the food (goal) in the current state.

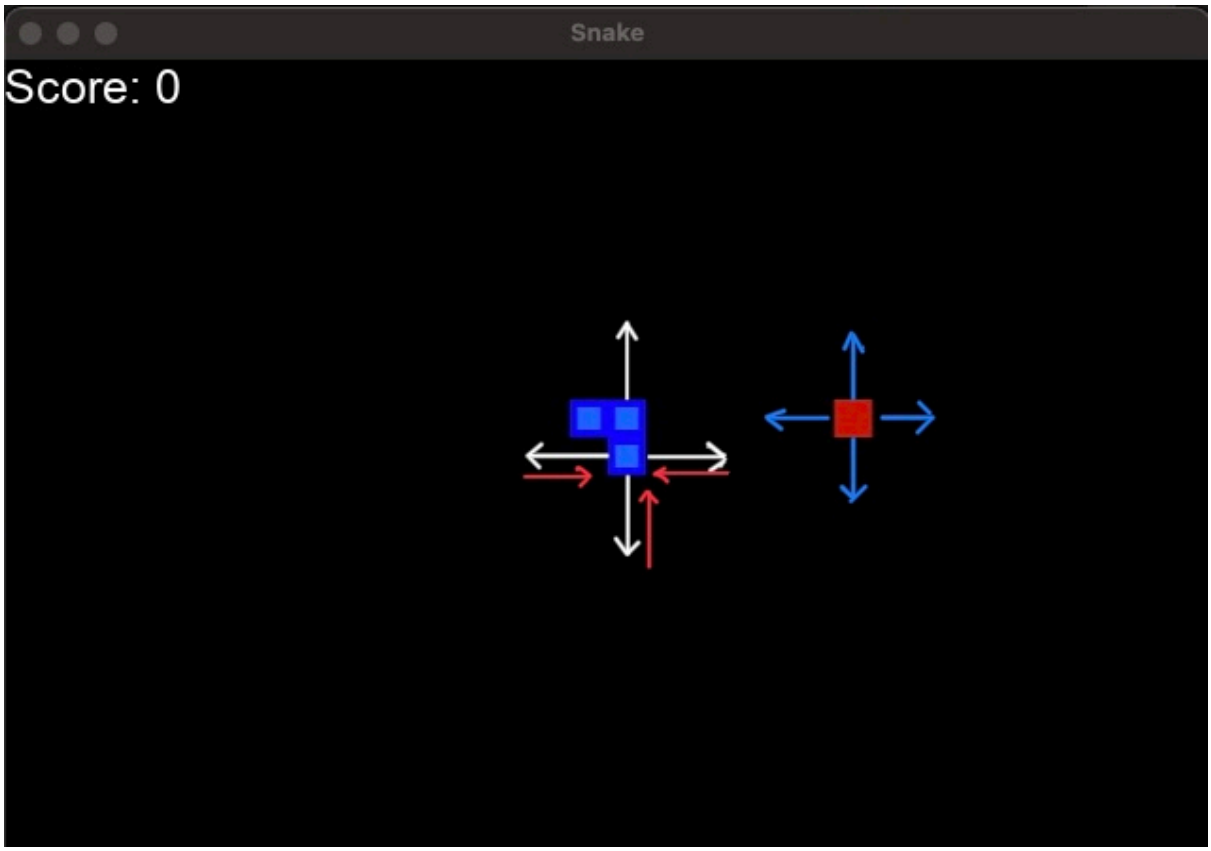

Figure 3.2: Data collection from the snake game

When referred to Figure 3.2 (data collection from the snake game), the food position and the snake can be observed in the frame, and the boundaries of the frame can be considered as the physical boundaries of the play area of the snake. To get the spatial information of the snake, 4 position pointers are created, marked as white

arrows in Figure 3.2. The white arrows denote the possible motion in which the snake can move. The location of food is denoted by blue arrows in Figure 3.2, these are the possible relative positions of food, while taking the snakes' head as the reference point. These reference points allow the snake to locate the food's direction at any given instance. The red arrows in Figure 3.2 denote the possible dangers that can end the game either by colliding with the boundary or into itself, this will also allow the snake to keep track of its body. One thing that can be noticed is that there is no need for danger at the back of the snake; because if the snake can not to turn 180 degrees, as the game uses the CPUs' clock to register an input, and only a single input can be registered per clock, so it eliminates the possibility of snake turning into its own body. In total, 11 parameters need to be considered while automating the game. These 11 pointers will provide values in boolean that will help the neural network to get the spatial awareness required for it to automate the movement of the snake.

There are multiple ways in which spatial awareness can be obtained like, the entire snake can be tracked, making the whole body of the snake a pointer to get the relative positions of danger or to get the food position data. This method though better on paper as we get more spatial information at each instance can be computationally expensive and can result in inefficiency. As the snake can grow in length, at the initial stages of the game, the method mentioned above will not have much difference, but as the game proceeds it will become computationally too demanding, thus resulting in a decrease in performance. This makes tracking the whole snake inefficient. The other method that can be implemented is by tracking food by the angle to the snake's head, rather than the traditional methods of Boolean values. This will allow the snake to get the precise location of the food and danger but will cause the snake to move in a staircase-style manner as though the values of the danger and food are not in Boolean, but the snake cannot move diagonally causing it to make rapid straight and horizontal movements. This does not fit the aim of the project; therefore, traditional Boolean methods were chosen instead of these two as they fit the needs of the project more precisely and are not too costly to operate.

3.2.2 Design of neural network

The main neural network is designed based on DQN architecture which has its roots in reinforcement learning. The use of DQN helps the neural network to converge more precisely, by allowing the neural network, to learn from the replay memory.

The 11-parameter gathered from the game, which has already been discussed in section 3.2.1, is passed to the neural network as the input layer. The demonstration of the input can be seen in Figure 3.3 (design of the neural network) where the 11 inputs are, direction up, direction down, direction right, direction left, food up, food, down, food right, food left, danger left, danger right and danger straight.

The core design of the neural network follows a 1-2-1 design, where there is 1 input layer, 2 hidden layers and 1 output layer, and the sizes considered for the input layer is 11 as previously discussed, the size for both the hidden layers are 121 neurons and the size of the output layer is 3. The three output neurons denote the three possible directions in which the snake can turn, left, right or straight.

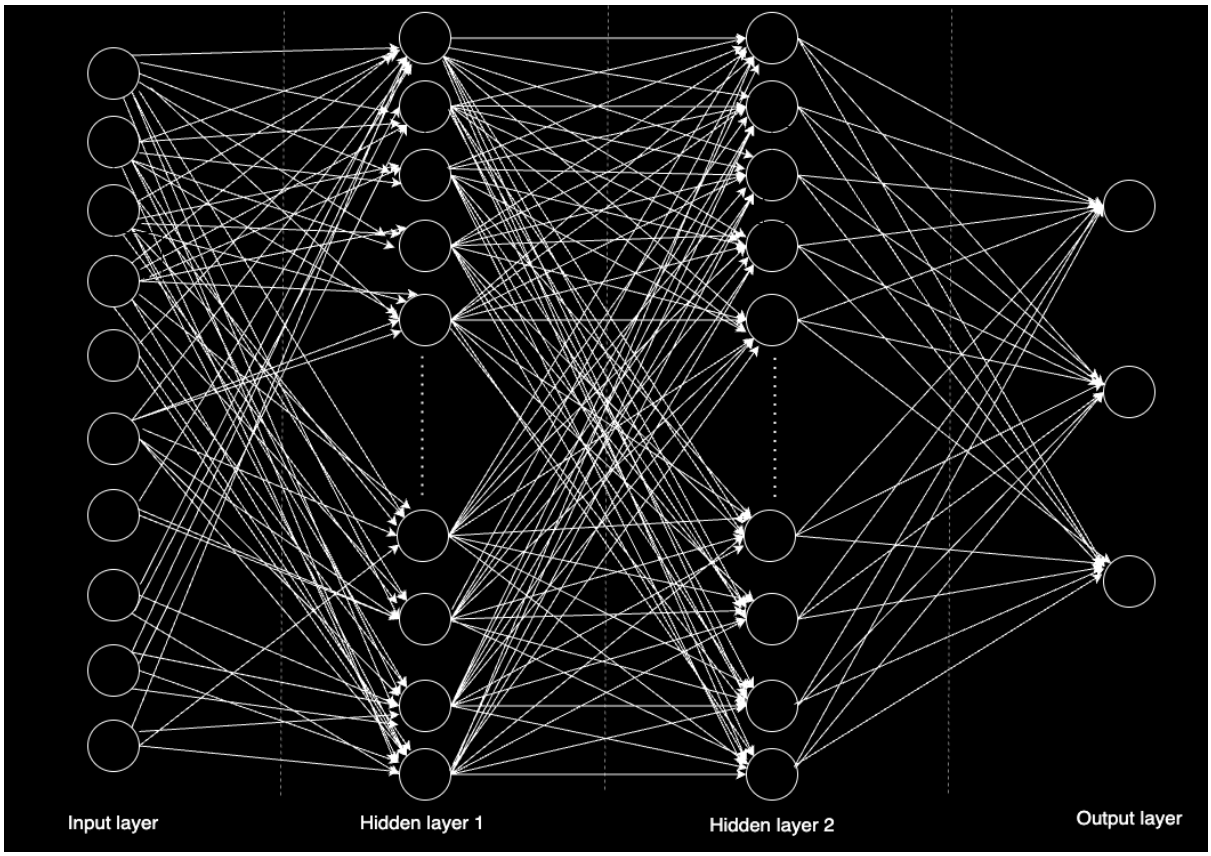

Figure 3.3: Design of neural network

The neural network will be using the Bellman equation for calculating the reward (refer to section 2.1.2.3 and figure 2.1); which calculates the current reward and discounted next reward, which will help the neural network to maximize the reward. In the case of this project, a discount rate of 0.9 is applied to the network, also known as gamma.

DQN architecture which can be referred from Figure 3.4 (DQN architecture using short-term and long-term memory), which is inspired by Arwa, Erick and Folly,

Komal's Techniques for Optimal Power Control in Grid-Connected Microgrids (Arwa, Erick and Folly, Komal's, 2020), uses both short term and long-term memory to improve learning and decision making. Short-term memory also known as the experience reply is where the neural network stores its experience in batches which are randomly sampled during the training. This helps reduce biases by breaking the correlation between two consecutive experiences. Long-term memory can be visualised as a collection of the memories that the neural network learns during the training over various iterations. The long-term memory enables the model to make informed choices while predicting the actions, based on the learnings from the previous experiences. DQN uses the neural network to calculate the Q values also known as the reward value for the network to help it keep the target same. The target Q value is the expected cumulative future rewards for the model that it can achieve by taking a particular action in the given state. For the visualisation of the mentioned process, Figure 3.4 can be referred to.

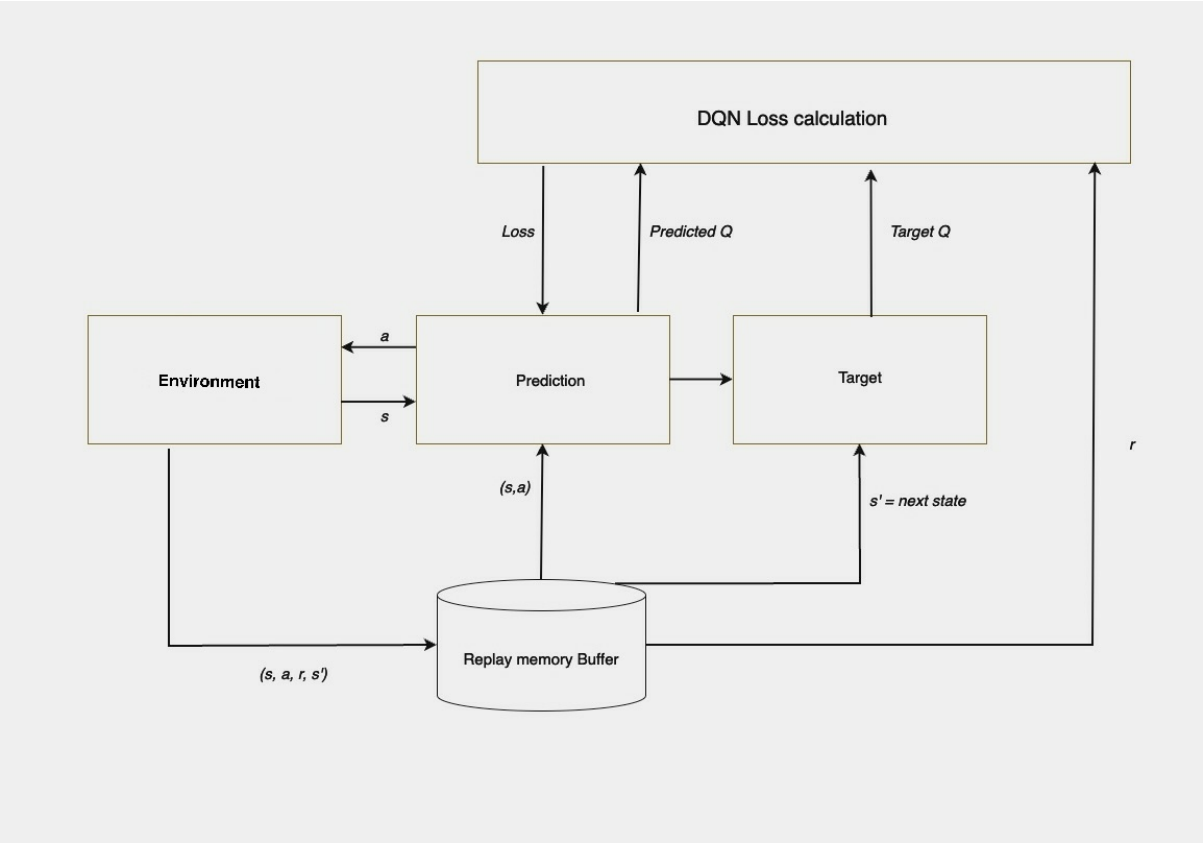

Figure 3.4: DQN architecture using short-term and long-term memory

The project uses MSE loss as the loss function for the neural network as it provides various advantages over other loss functions, such as its sensitivity towards outliers, as the values are squared this allows it to eliminate any large outliers, proving beneficial for the convergence. Another advantage that MSE loss provides is its ease

of optimization, since the MSE is a smooth convex function, it allows the neural network to calculate the gradient easily, which makes backpropagation for the neural network simpler during the training process.

Figure 3.5: Total parameters in the neural network

Referring to Figure 3.5 (Total parameters in the neural network) will help understand the number of parameters which are trainable, in the used neural network, eventually helping in the debugging and fine-tuning of the neural network. This helps in understanding the architecture of the neural network more closely by presenting the exact trainable parameters present in a neural network. For example, while referring to Figure 3.5, it can be observed that the first linear layer of the neural network has 1,452 total parameters, the second layer has 14,762 total parameters and the third layer of the neural network has 366 total parameters, giving the total parameters present in the neural network to be 16,580, out of which trainable parameters are 16,580 and non-trainable parameters are 0.

The results presented above and in Figure 3.5 were obtained by using the torchinfo (torchinfo, Tyler Yep, GitHub, 2023) library as, PyTorch does not provide built-in functionality for visualising a neural network similar to TensorFlow's (TensorFlow, 2023) model.summary(), which provides similar results obtained in the figure 3.5.

Chapter 4
Implementation

This chapter discusses the implementation of the designs discussed in the previous chapter (refer to Chapter 3.2). This chapter studies, one of the most important deliverables, which is the algorithm used in this project. Implementation will be focused on two areas, first the implementation of the neural network itself and second the implementation of the neural network into the snake game.

This project will be using a python's virtual environment created using Anaconda, which is previously been discussed in section 3.1.2. This will help in organising the libraries required for the project. A fresh environment can be created with the help of Anaconda by giving the "conda create –name {environment name} {python version}" (Anaconda, 2023) command in the terminal when in the base environment of Anaconda. The virtual environment can be accessed by giving "conda activate {environment name}" (Anaconda, 2023) in the terminal.

The base snake game requires Pygame, random, Enum and NumPy libraries to run. The random, Enum and NumPy libraries are preinstalled libraries when using a virtual environment from Anaconda, but the Pygame library needs to be installed first. To install the Pygame library give the "pip install pygame" (PyPI, 2023) command in the terminal when the virtual environment is active. The important thing to notice here is that the activation of the correct virtual environment is crucial, as failing to do so will result in mishandling of the library and the library will not be accessible to the environment.

4.1 Implementation of Neural Network

The design of the neural network is a fundamental aspect of the performance of the project, this is covered in detail in section 3.2.2. The neural network consists of 11 neurons in input layers, 2 hidden layers, each containing 121 neurons, and 3 neurons in the output layers (please refer to figure 3.2). The implementation of the neural network can be further divided into two subsections, first, the implementation of the actual design of the neural network which is previously discussed, and the second section discusses about the implementation of the training of the neural network.

4.1.1 Implementation of the design of the Neural Network

```python
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F




class Linear_QN(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super().__init__()
        self.linear1 = nn.Linear(input_size, hidden_size1)
        self.linear2 = nn.Linear(hidden_size1, hidden_size2)
        self.linear3 = nn.Linear(hidden_size2, output_size)

    def forward(self, x):
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)

        return x

    def save(self):
        filename = './model.pth'
        torch.save(self.state_dict(), filename)

```

Figure 4.1: Code for implementation of the design of neural network

Figure 4.1(Code for implementation of the design of neural network), provides the code for implementation of the neural network's design. As previously discussed, this project will be using PyTorch (refer to section 3.1.5), and the functionalities of PyTorch need to be added to the code base. To achieve that, first, the library needs to be installed as the PyTorch library is not preinstalled in the environment. For installing the library give the "pip install torch torchvision" (PyTorch, 2023) command in the terminal. Once the library is acquired, import the library in the code. To get the functionalities of the neural network, a sub-library of torch called "torch.nn" is required, which provides the basic functions required for creating neural networks like the "Linear Layers", "Loss Functions", etc. An activation function is also required for the neural network to decide whether the neuron should be activated or not, by introducing non-linearity into the calculation of the weighted sum. This helps the neural network to solve the problem of vanishing gradient. This project will use the ReLU activation function which stands for “Rectified linear unit”. This function has similar computation as that of Sigmoid and Tanh but is computationally less expensive than the others, hence making it one of the most widely used activation functions. Figure 4.2 represents the graph of the ReLU function R(z) = max(0, z), which was acquired from AILEPHANT (AILEPHANT, 2023), where the function R(z) gives an output when the z is positive, or gives 0 otherwise, while the range of the z

can lie from 0 to infinity. To acquire the ReLU function, "torch.nn.functional" needs to be imported to the code base.

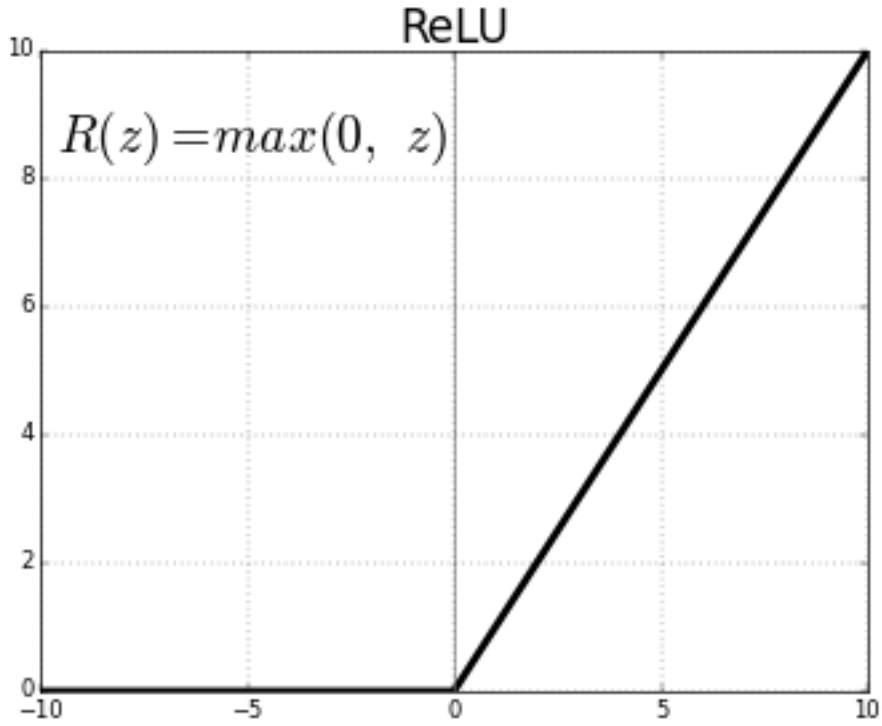

Figure 4.2: ReLU function graph and its equation (AILEPHANT, 2023)

Once the required libraries are installed and imported to the code base, the implementation of the designed neural network can be started. First, a class which will define the neural network needs to be initiated. This project will be using "Linear_QN" as the name of the class, and the "nn.module" needs to be passed in the definition of the class, for the class to access the full functionalities of the "torch.nn", for clear implementation refer to Figure 4.1. Then the structure of the neural network needs to be defined, by defining each layer of the neural network. This project uses the linear layer structure for the neural network layers, to implement this use the syntax "linear_layer = nn.Linear({input layer size}, {output layer size}), this has to be repeated till the decided number of layers is achieved, this project uses a 1-2-1 structure so to implement these layers, 3 linear layers are required. Once the 3 linear layers are acquired, a forward function needs to be defined, which directs the flow of the neural network and helps the neural network in deciding by applying the activation function. So as previously mentioned, the ReLU function needs to be applied to the layers that are concerned with the decision-

making, in this case, layers 1 and 2, will require ReLU(application of the function can be referred from Figure 4.1). Since neural networks are interconnected and the next layer requires the output from the previous layer, a variable 'x' is defined which is passed and updated at each linear layer and once it is obtained from the output layer or final linear layer it is returned by the function.

Finally, the states and values acquired from the neural network need to be stored in a '.pth' file, so that the neural network can refer to the data obtained during the training, a save function is created that will map each layer to its parameter tensor. This is achieved by using the PyTorch function named "torch.save", following the syntax as "torch.save(self.state_dict(), filename)" (PyTorch, 2023), where the state_dict is a Python dictionary that maps each layer to its parameter tensor. For the exact implementation, Figure 4.1 can be referred to.

4.1.2 Implementation of the training of neural network

```
28  class Trainer:
29      def __init__(self, model):
30          self.lr = 0.01
31          self.gamma = 0.9
32          self.model = model
33          self.optimiser = optim.Adam(model.parameters(), lr = self.lr)
34          self.loss = nn.MSELoss()
35
36      def train_step(self, state, action, reward, next_state, game_over):
37          state = torch.tensor(state, dtype= torch.float)
38          action = torch.tensor(action, dtype= torch.long)
39          reward = torch.tensor(reward, dtype= torch.float)
40          next_state = torch.tensor(next_state, dtype= torch.float)
41
42          if len(state.shape) == 1:
43              state = torch.unsqueeze(state, 0)
44              next_state = torch.unsqueeze(next_state, 0)
45              action = torch.unsqueeze(action, 0)
46              reward = torch.unsqueeze(reward, 0)
47              game_over = (game_over, )
48
49          pred = self.model(state)
50
51          target = pred.clone()
52          for idx in range(len(game_over)):
53              Q_new = reward[idx]
54              if not game_over[idx]:
55                  Q_new = reward[idx] + self.gamma * torch.max(self.model(next_state[idx]))
56
57              target[idx][torch.argmax(action[idx]).item()] = Q_new
58          self.optimiser.zero_grad()
59          loss = self.loss(target, pred)
60          loss.backward()
61
62          self.optimiser.step()
```

Figure 4.3: Code for implementation of trainer for neural network

Figure 4.3 (Code for implementation of trainer for neural network), provides the code for training the neural network created in the previous section. To create a trainer, first, a trainer class needs to be created, the exact implementation can be referred from Figure 4.1. After the creation of class, define an initialisation function that will take the neural network model as an input; this can be done by using the syntax "def

__init__(self, model)", this will enable the function to automatically initialise whenever the training class is called.

This project uses the "Adam" optimiser, short for Adaptive Moment Estimation, which is a widely used optimisation algorithm as it uses adaptive learning rates for each parameter whilst maintaining moving averages of the past squared gradients. This makes Adam versatile and faster while converging due to the variable learning rate for each parameter. For loss function, this project uses MSELoss, which is previously discussed in detail in section 3.2.2 (refer to section 3.2.2).

To perform training at each step a "train_step" function is created which takes the current state of the game, the action performed, the current reward of the state, the next state of the game and the game over as, (which tells the algorithm if the game is over or not) as input. Since each of the values except the game-over value, needs to be in a tensor, it must be converted. This can be done using the syntax "x = torch.tensor(x, dtype = torch.float)" (PyTorch, 2023), this will convert the normal values into tensor values, and the dtype argument will help maintain the original datatype of the x, the exact implementation of this syntax for this project can be seen in Figure 4.3. The thing to notice when converting into tensors is the results obtained from the torch.tensor() will be a multi-dimensional tensor, which can cause dimensional errors during the training process. To avoid this situation, the tensors need to be "unsqueezed". "Unsqueezed", adds an extra dimension to the tensor, which makes the matrix multiplication possible during the training process. This can be achieved by using the syntax "x = torch.unsqueeze(input, dimension)" (PyTorch, 2023), this will return the values of the tensor in (1,x) format. Another thing to notice is to put the game-over value in a tuple, like "game_over = (game_over,)" to eliminate the possibility of dimension error during multiplication.

Further, get predictions from the model, and create a copy of the predictions into a variable to calculate the Q values for the reward using the Bellman equation (the Bellman equation is discussed in detail in section 2.1.2.3 and the Bellman equation can be referred from Figure 2.1). To calculate the reward first it needs to be checked if the game is over or not, this can be done using for-loop, by parsing through the game_over variable. If the game has not ended then, the current reward for the given index is the reward, (Q = current reward), when calculating for the next reward, the current reward is added to gamma or the discount rate, multiplied by the maxima of next predicted Q values (Q = current reward + gamma[discount rate] * max(next predicted new values)). The implementation of this step for this project can be seen in Figure 4.3. Now insert the obtained Q value into the target at the current index.

Finally, set the optimiser gradient to zero by using "optimiser.zero_grad()" (PyTorch, 2023), this helps in doing backpropagation by explicitly setting the gradients to zero during the training phase, so that the parameters update correctly, otherwise the gradient obtained will be a combination of the old gradient and new gradient.

4.2 Implementation of neural network in Snake Game

This section will discuss the implementation of the neural network and training model created in the previous sections 4.1.1 and 4.1.2 into the snake game using the DQN techniques. The DQN architecture is discussed briefly in the sections 3.2.2 and 2.4.

The implementation of neural networks in the snake game can be divided into two sections. The first section will discuss the implementation of functions that will collect the data from the game and will also help in the training of the neural network. The second section will discuss the implementation of the training function which will actually run the training function built in the previous section 4.1.2 and will also provide the replay memory buffer and a long-term memory buffer. The architecture of this is discussed in brief in section 3.2.2 and the visualisation can be seen in the figure 3.3.

4.2.1 Implementation of Training Functions

```
Class Worker:
    Initialize:
        Initialize n_games = 0, epsilon = 0, gamma = 0.9
        #MAX_MEMORY is 100,000
        Initialize memory as deque with maximum length MAX_MEMORY
        Initialize model using Linear_QN(11, 121, 121, 3)
        Initialize trainer using Trainer(model, lr=LR, gamma=gamma)

    Function get_state(game):
        Compute head and surrounding points
        Compute danger indicators and move directions
        Compute food location indicators
        Return state as np.array

    Function save_state(state, action, reward, next_state, game_over):
        Append (state, action, reward, next_state, game_over) to memory

    Function train_long_memory():
        If length of memory > BATCH_SIZE:
            mini_sample = random sample of BATCH_SIZE from memory
        Else:
            mini_sample = memory
        Extract states, actions, rewards, next_states, dones from mini_sample
        Call trainer.train_step with states, actions, rewards, next_states, dones

    Function train_short_memory(state, action, reward, next_state, game_over):
        Call trainer.train_step with state, action, reward, next_state, game_over

    Function get_action(state):
        Calculate epsilon based on n_games
         epsilon = 80 - number of games
        If random value < epsilon:
            Choose a random move
        Else:
            Get model's prediction for the state
            Choose the move with the highest predicted value
```

Figure 4.4: Pseudocode for the implementation of training functions

The implementation of the training function in this project is mainly concerned with defining the functions that are going to be needed during the training process. Figure 4.4 (Pseudo Code for the implementation of the training functions) gives the idea of the code flow and the code structure of the implementation. To properly organise the code a class named 'Worker' is created, which when called can access the required method. For the initialisation the variable concerned with the number of games is assigned zero, the epsilon which is also used to keep the game random is also assigned zero, and as previously discussed, the gamma also known as the discount rate for the Bellman equation is 0.9. To store the memory, deque (Python, 2023) is being used, with the allowance of 100,000 as the memory. Finally, initialise the model and the trainer function previously created into the function to make it accessible for the class.

```
def get_state(self, game):
        head = game.snake[0]
        xpt_l = Point(head.x - BLOCK_SIZE, head.y)
        xpt_r = Point(head.x + BLOCK_SIZE, head.y)
        xpt_u = Point(head.x, head.y - BLOCK_SIZE)
        xpt_d = Point(head.x, head.y + BLOCK_SIZE)

        dir_l = game.direction == Direction.LEFT
        dir_r = game.direction == Direction.RIGHT
        dir_u = game.direction == Direction.UP
        dir_d = game.direction == Direction.DOWN

        state = [

            (dir_r and game.is_collision(xpt_r)) or
            (dir_l and game.is_collision(xpt_l)) or
            (dir_u and game.is_collision(xpt_u)) or
            (dir_d and game.is_collision(xpt_d)),

            (dir_r and game.is_collision(xpt_d)) or
            (dir_l and game.is_collision(xpt_u)) or
            (dir_u and game.is_collision(xpt_r)) or
            (dir_d and game.is_collision(xpt_l)),

            (dir_l and game.is_collision(xpt_d)) or
            (dir_r and game.is_collision(xpt_u)) or
            (dir_u and game.is_collision(xpt_l)) or
            (dir_d and game.is_collision(xpt_r)),

            dir_l,
            dir_r,
            dir_u,
            dir_d,

            game.food.x < game.head.x,
            game.food.x > game.head.x,
            game.food.y < game.head.y,
            game.food.y > game.head.y
        ]
```

Figure 4.5: Code for get_state function

To acquire information about the current state of the game, a 'get_state' function is created, the code reference can be seen in Figure 4.5, where the function defines 4 added with a block size (20 pixels) of external points around the snakes' head, which will help the snake to sense the dangers around. Then another four variables are created, depicting the snakes' direction of motion. The state variable defines a base state in the form of 11 Boolean values, the 11 values are briefly discussed in section 3.2.1. After creating a 'get_state' function a function to save the current state is created named 'save_state'. The function saves the current state of the game in the format previously discussed to the memory.

To train the neural network from the long memory a 'train_long_memory' function is created. The function checks if the size of the memory exceeds the batch size or not. If the size of the memory does exceed the batch size, random samples of the batch size are taken from the memory. If the size of memory does not exceed the batch size the entire memory is used as samples for the training. Once the samples are acquired, the required elements are extracted from the samples, like states, actions, rewards, next states, and game over, and are sent to the trainer for the neural network. To train from short memory a function called 'train_short_memory' is created, which directly sends the current state to the trainer.

Finally, to acquire the next move for the snake, a function get_action is created. This function helps the snake to explore during the initial stages of the game. This is done by using the epsilon, which was previously initialised. The epsilon is updated after every game, in the case of this project, 'epsilon = 80 – number of games'. Once the epsilon is calculated, it is compared with a random integer between 0 to 200 and if the random integer is smaller than the epsilon, a random move will be performed, otherwise, a prediction from the model is acquired to perform the move.

4.2.2 Implementation of the training function

```
Function train():
    Initialize plot_scores and plot_mean_scores as empty lists
    Initialize total_score = 0, record = 0
    Create an Worker instance called worker
    Create a SnakeGame instance called game
    While True:
        Get old state using worker.get_state(game)
        Get move using worker.get_action(state_old)
        Perform the move in the game and get reward, game_over, and score
        Get new state using worker.get_state(game)
        Train short memory using worker.train_short_memory
        Remember the experience using worker.save_state
        If game_over:
            Reset the game
            Increment n_games
            Train long memory using worker.train_long_memory
            If score > record, update record and save the model
            Print game number, score, and record
            Append score to plot_scores and update total_score
            Calculate mean score and append to plot_mean_scores
            Plot scores and mean scores using plot function

If __name__ == '__main__':
    Call train()
```

Figure 4.6: Pseudocode for the training function

Figure 4.6 (Pseudo code for the training function) presents the pseudo-code for the training function which trains the neural network. For initial initialisation create two empty lists to keep track of the scores and the mean of the score for the given

instance. Then create an instance of the Worker class so that, it is possible to access the functions created in that class. Also, create an instance of the actual snake game to access the attributes of the snake and to be able to control the game.

Once the required attributes are acquired, an endless loop is created to keep the training going, by using "While True". To proceed further the current state of the game is required, this is done by using the 'get_state' function created in the worker class, and based on the state the action is also determined through the 'get_action' function.

```
64      def play_step(self, action):
65          self.frame_iteration += 1
66          # 1. collect user input
67          for event in pygame.event.get():
68              if event.type == pygame.QUIT:
69                  pygame.quit()
70                  quit()
71
```

Figure 4.7 Code for play step function

The acquired move is then performed in the game using the 'play_step' function available in the game (the code for this can be seen in Figure 4.7), this provides the worker with the values of reward, game-over status, and score value. Based on these outcomes a new state is derived from the 'get_state' function, which is stored, this is done to help the models' short-term memory which helps the neural network to learn from the immediate outcomes. These results are also stored in the memory for the model to learn when it is using long-term memory. To store the results, the 'save_state' function is used to store the results in the memory.

```
41      def reset(self):
42          # init game state
43          self.direction = Direction.RIGHT
44
45          self.head = Point(self.w/2, self.h/2)
46          self.snake = [self.head,
47                        Point(self.head.x-BLOCK_SIZE, self.head.y),
48                        Point(self.head.x-(2*BLOCK_SIZE), self.head.y)]
49
50          self.score = 0
51          self.food = None
52          self.place_food()
53          self.frame_iteration = 0
```

Figure 4.8: Code for reset function

Once the initial game is completed, the game is then reset using the 'reset' function available in the game (the code for this can be seen in Figure 4.8), with the increment in the number of games variable. The reset function resets the game to its initial stage, by placing the snake in its default direction and resets the score to 0.

After the initial training based on the short-term memory is done the neural network is then trained on the results stored in the memory using the long-term memory function 'train_long_memory'. As previously discussed, long-term memory uses the memory in randomly sampled mini-batches and trains the model based on the previous outcomes. If the current game score is higher than the total high score of previous games, then the variable concerned with the high score is then updated to the current high score and the current model is saved in a '.pth' file. To plot the graph of the models' performance the current score is appended to the 'plot_scores' list, and the mean of all scores is calculated and then again appended to the 'plot_mean_scores' list.

```
6   def plot(scores, mean_scores):
7       display.clear_output(wait=True)
8       display.display(plt.gcf())
9       plt.clf()
10      plt.title('Total games to Score')
11      plt.xlabel('Games')
12      plt.ylabel('Score')
13      plt.plot(scores, color = 'black')
14      plt.plot(mean_scores, color = 'red')
15      plt.ylim(ymin=0)
16      plt.text(len(scores)-1, scores[-1], str(scores[-1]))
17      plt.text(len(mean_scores)-1, mean_scores[-1], str(mean_scores[-1]))
18      plt.show(block=False)
19      plt.pause(.1)
20
```

Figure 4.9: Code for plotting function

After the scores are accumulated, the list is then used to plot the graph of the model's performance using a Python library called "Matplotlib" (Matplotlib, 2023). This is achieved by creating a plotting function, the code for which can be seen in Figure 4.9. The function plots the number of games to score in the graph where the X axis represents the total number of games played and the Y axis represents the scores of the game. The entire process of training is repeated and again to enhance the decision-making of the model.

Chapter 5
Software Testing and Evaluation

This chapter will discuss the testing and evaluating results acquired from testing the implemented model. The implemented model will be compared with two different architecture models, based on the consistency of the model, the high score it achieves, and the mean scores it produces. The single hidden layer model, two hidden layers model and three hidden layers model are the three models that are going to be compared. This chapter will be divided into 4 sections, the first, second and third will be the observations during the testing of all models followed by the comparison in the last section.

5.1 Performance of two Hidden Layers Model

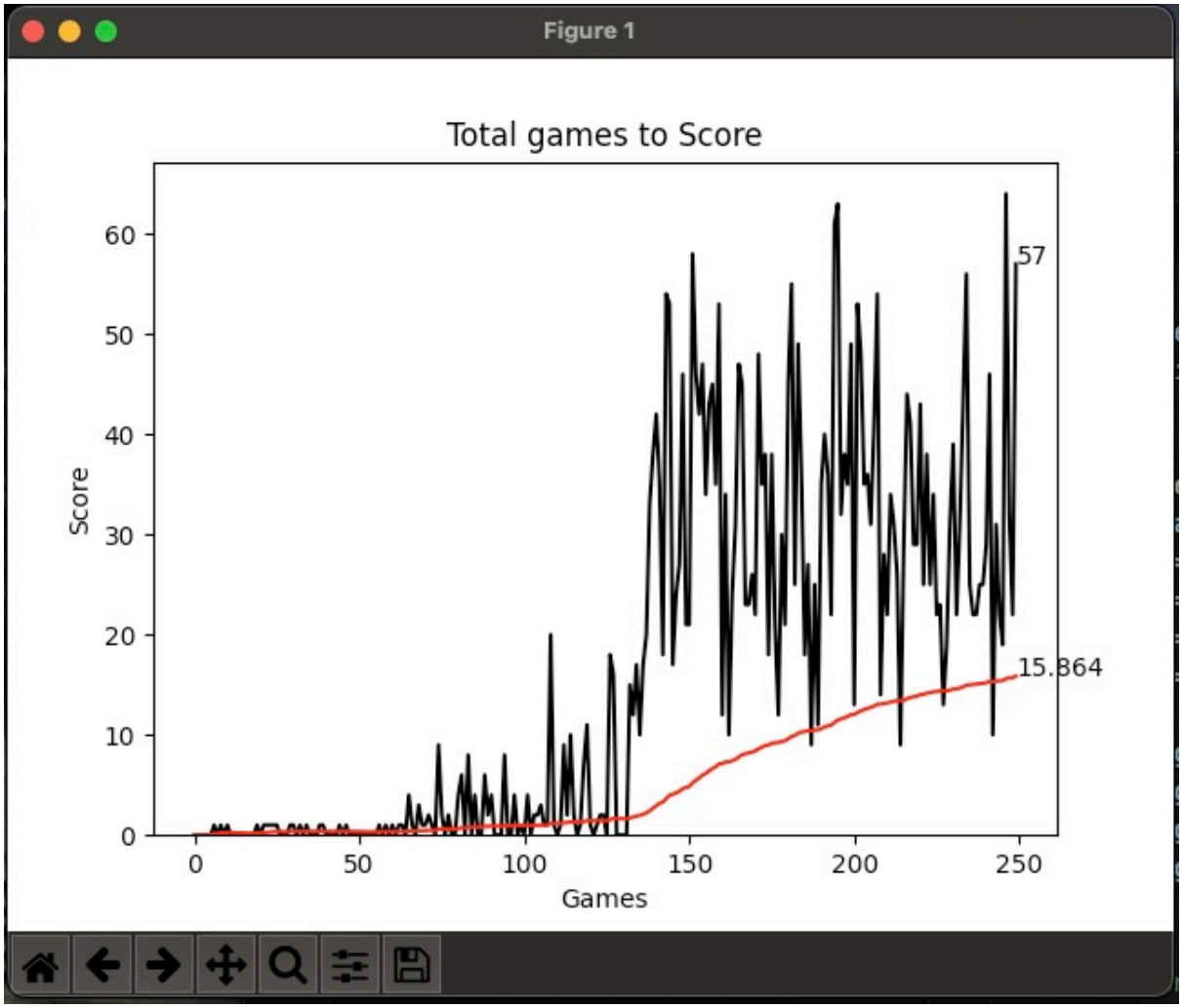

Figure 5.1: Performance of two hidden layers of neural network

The two hidden layers model is the main model that is being used in this project. The size of the two hidden layers in the model is 121 neurons each, the design of the model is discussed in depth in section 3.2.2. The testing was done on this model for 250 iterations. Figure 5.1 (Performance of two hidden layers model) depicts the visualised behaviour of the model. When referencing Figure 5.1, it can be observed that, during the starting phase of the training, the model did not perform well, which is expected behaviour as the model is forced to explore more during the early stages. Due to the introduction of epsilon condition which forces the model to perform random movements in the early stages. The model does not cross the 10-point mark and is not consistent, the model frequently scores 0 points. After around 100 games it starts to increase its consistency and increases its high score to 21 points. The model starts to consistently score above 10 points, still, the model has not perfected its strategy yet as it still occasionally scores 0 points in between. After around the 125th game mark, the model starts to perform better as it starts to use its experiences using long-term memory to perfect its strategy. Due to this, the model starts to consistently score above 40 points.

The model performs consistently after the 150-game mark, as on the 150th game the model scores a new high score of 56. There are no instances where the performance dips below 10 points. The model over the span of 250 games, averaged its score of 15.864 points which can be considered good, considering that the model did not start to perform well unit game number 110. The highest score achieved by this model is 64 points, achieved in 246th game.

The noticeable behaviour observed during the testing of this model is that, occasionally the snake would coil itself and the game would end, since the snake would hit his own body, or there would be instances where the snake would get stuck in an endless loop, where it would follow its tail, till the game would quit itself due to timeout. This behaviour would hamper the consistency of the model, resulting in low scores. After the 125th game, the endless loop behaviour would reduce resulting in a better outcome, but the coiling behaviour though reduced in many iterations, would persist, and would mainly occur when the size of the snake would be very large. Due to the coiling behaviour, the model could not gain more points, resulting in a lower high score.

5.2 Performance of single Hidden Layer Model

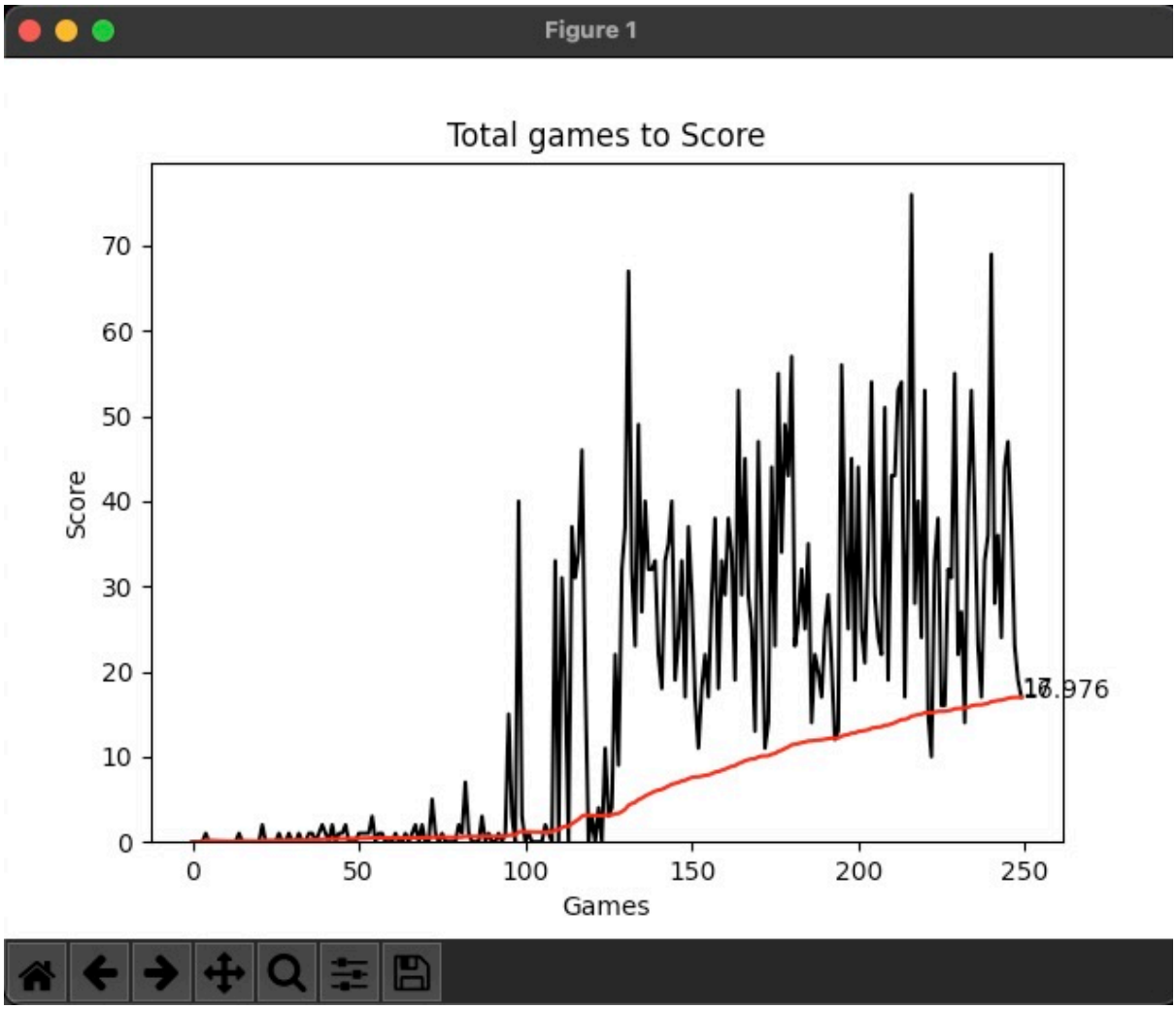

Figure 5.2: Performance of the single hidden layer of neural network

This is the first of the two models to which the main neural networks' model is going to be compared. This model has 1 input layer with 11 neurons in it, 1 hidden layer with 121 neurons and 1 output layer with 3 neurons. This testing was done for 250 iterations. Figure 5.2 (Performance of the single hidden layer model), helps in visualising the performance of this model. The model performs poorly till the 94th game, which is again as expected as the training is designed in that way, due to the addition of epsilon. The first peak in the model is achieved in the 98th game by scoring 41 points.

The model starts to perform more consistently after the 118th game, from where the performance does not drop below 10 points. Though the performance is much better when compared to the start of the training, major drops can be still seen in the performance of the model. During the span of the 250 games, the model performs well achieving a high score of 74 in the 226th game and has a mean score of 16.976.

The noticeable observation during the training was that the snake would repeatedly get stuck in an endless loop of following its tail until the game ended due to the time out. This behaviour was seen throughout the training lowering the scores and creating drops in the performance, which are visible in Figure 5.2. The snake would rarely coil itself and end the game, but this behaviour could also be seen as well. Further dropping the efficiency of the model.

5.3 Performance of three Hidden Layers Model

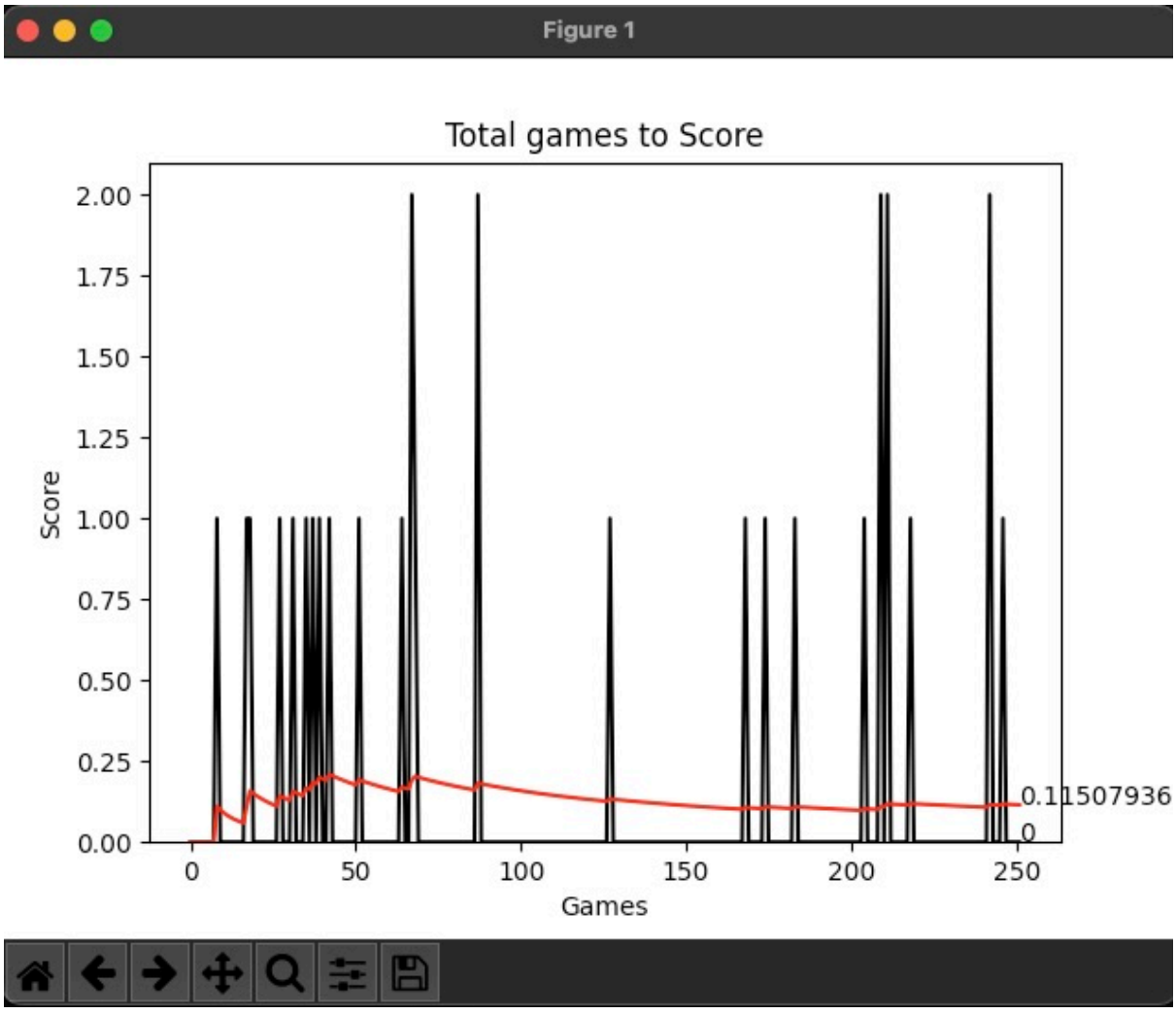

Figure 5.3: Performance of three hidden layers of neural network

This is the second neural network to which the main neural network is going to be compared. This neural network has 1 input layer consisting of 11 neurons, 3 hidden layers each consisting of 121 neurons and 1 output layer consisting of 3 neurons. The testing of this model was performed for 250 iterations. Figure 5.3 (Performance of three hidden layers neural network) helps to visualise the performance of the

neural network. Keeping Figure 5.3 as the reference, the performance of the model is consistently poor throughout the testing. The model scores a maximum of 2 points in the entire testing phase, with an average score of 0.115 points.

During the observation of the snakes' behaviour, it was noticed that the snake would always get stuck in an endless loop where it would follow the snake's tail till the game would timeout. This behaviour resulted in very low scores, and even if the snake scored points, the snake would eventually go in an endless loop, resulting in the timeout of that iteration. The training started with the random moves being performed, which is the expected behaviour, but as the training progressed the snakes' actions were repetitive, and the aforementioned behaviour would persist.

5.4 Comparing the performance of all the models

The testing of all three models was performed under identical conditions to get the most accurate results for comparison. Fine-tuning of the two models would result in better performance, but to compare the models in identical conditions and using the same parameters, the base tuning of all the models was kept the same. The testing of all three models was done for exactly 250 iterations/ number of games.

The performance of the single hidden layer model and the double hidden layers model is similar when referred to in Figure 5.2 and Figure 5.1. Both the models start similarly with random movements and the models would stabilise, around the 125th game. The single hidden layer model would score 74 points with an average of 16.976 points, whereas the double hidden layer model would score 64 points with an average of 15.864 points. The difference between the two models would be visible in the behaviour of the snake during the training. In the single hidden layer model, the snake would go in an endless loop following its tail, creating drops in the graph of performance. This behaviour was significantly reduced in the double hidden layers model, in this model the drops were created because the snake would coil itself, trapping the head, but this behaviour could only be seen when the size of the snake gets too large.

The performance of the double hidden layer models and triple hidden layer model was similar in the beginning 50 iterations of the testing but would drastically change after that. The two-hidden-layers model was able to perfect its strategy as the testing progressed, but the three-hidden-layers model failed to do so. Resulting in a very low score and average compared to the double hidden layers model. Rendering the triple hidden layers model is not an optimal choice for it to be considered as the main model for this project.

While considering the main model for this project, the single hidden layer and double hidden layers models were the most optimal and complacent designs. Both models have their unique advantages and cons. The single hidden layer model was capable of achieving a better high score and better average score as compared to the double layer model, on the contrary, the double hidden layered model was much more reliable and was able to reduce the issue of the snake stuck in an endless loop. The selection of the main model for this project was predicated upon a focused inquiry into the alignment of the models' behavioural variations with the overarching objectives of this research. The pivotal criterion guiding this decision-making process was the meticulous evaluation of which specific model variant best converged with the fundamental aim of this project. After scrutinising the behaviour of both the models, the double hidden layer model emerged to be having the most converging with the aim of this project. In addition, the double hidden layers model provided an ideal equilibrium between archiving high scores and the stability of the model.

Chapter 6
Conclusion and Future Work

6.1 Conclusion

This project aimed to design, build, and evaluate a neural network model that can replicate human-like input to the classic snake game, using DQN architecture. This was achieved successfully with the creation of a double hidden layers neural network which uses its own experiences learnt during the training, by using its replay buffer.

The decision to use DQN architecture was based on the background research done during the project, where previous research done on automation of the user input in video games suggested the use of hybrid models of Deep neural network and Q-Learning; these are considered to be the bases of Deep Q Network usually known as DQN architecture. DQN architecture enabled this project to pursue its aim of keeping the inputs as humanly as possible.

The design of the neural network was key to the performance of the model. Various designs of neural networks were considered as the main model for this project, which was later evaluated and compared with the main model in the evaluation and testing phase of this project (refer to Chapter 5 Section 5.4). During the coding process the libraries used like PyTorch, Pygame, etc., proved beneficial due to their wide adaptation and ease of use, providing lots of resources and documentation available. The use of Anaconda helped a lot during the coding process by providing key features like a virtual Python environment and the necessary libraries required for the development of this project.

The testing and evaluating phase of this project provided the real-world performance of the neural network design used in this project and compared it with other neural network designs. Each neural network design had its unique advantages but the choice for the double hidden layered neural network was based on its performance alignment to the aim of this project.

Finally, in conclusion, this project provided a glimpse into the field of AI in games and revived the timeless appeal of the traditional Snake game, with a touch of modern neural networks and their capabilities.

6.2 Future Work

There are several directions in which this project can be expanded, where a snake game is automated using a DQN architecture. First and foremost, the exploration strategy can be changed from the epsilon method to more robust strategies like Boltzmann exploration or noisy network. These could promote more diverse exploration strategies than using random movements for exploration, as this is a greedy approach that can be considered heavy on resources.

Another avenue in which this project can be expanded is by using the 'Double DQN' architecture. This architecture uses two separate neural networks for predicting the next action and for calculating new Q values. This architecture can solve the common issue found in the traditional Deep Q-Network of overestimation bias. The overestimation bias leads to the overestimation of Q-values, resulting in the degradation of the learning process.

Additionally, various strategies can also be applied to the replay buffer of the DQN. Where tweaking the size of the reply buffer can result in different outcomes from the model. There is also a possibility where the experience memory buffer is prioritised during the learning process which can result in enhanced learning efficiency and stability.

Lastly, running multiple models on the snake game at the same time can also be achieved; where one model controls the movement of the snake, and the other model can work as the awareness model for the snake. The awareness model will help the snake model to help predict the movements of the snake, by giving it the data about the possible surroundings for the snake in the next step. This can significantly enhance the performance of the model as it will make informed decisions for the next moves avoiding the coiling problem faced in this project.

List of References

AILEPHANT. (2017). *ReLU function - AILEPHANT*. [online] Available at: https://ailephant.com/glossary/relu-function/.

Arwa, E.O. and Folly, K.A. (2020). Reinforcement Learning Techniques for Optimal Power Control in Grid-Connected Microgrids: A Comprehensive Review. *IEEE Access*, 8(1-16), pp.208992–209007. doi:https://doi.org/10.1109/access.2020.3038735.

Astle, A. and Writer, S. (2023). *Inworld AI's report finds 99% of gamers are excited for AI NPCs*. [online] pocketgamer.biz. Available at: https://www.pocketgamer.biz/news/80847/inworld-ais-report-finds-99-of-gamers-are-excited-for-ai-npcs/ [Accessed 10 Jul. 2023].

Comi, M. (2018). *How to teach an AI to play Games: Deep Reinforcement Learning*. [online] Towards Data Science. Available at: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a [Accessed 14 Jul. 2023].

Loeber, P. (2020). *Snake game - Pygame*. [online] GitHub. Available at: https://github.com/patrickloeber/python-fun/tree/master/snake-pygame.

Mikhailov, A. (2020). *AI in Video Games: Artificial Intelligence | Experfy Insights*. [online] Experfy. Available at: https://resources.experfy.com/ai-ml/artificial-intelligence-in-video-games/ [Accessed 12 Jul. 2023].

Mikulcik, S. (2016). *Application of Neural Networks for Intelligent Video Game Character Artificial Intelligences*. [Honors Theses] Available at: https://encompass.eku.edu/cgi/viewcontent.cgi?article=1401&context=honors_theses [Accessed 12 Jul. 2023].

Python. (2023). *Python Documentation.* Available at: https://docs.python.org/3/

C++. (2023). *ISO CPP.* Available at: https://isocpp.org.

C . (2021). *C Documentation.* Available at: https://www.open-std.org/jtc1/sc22/wg14/.

Java (2023). *Java Documentation.* Available at: https://www.java.com/en/download/help/java8_manual_update_macos.html.

Feldman, S. (2019). *Infographic: The Most Popular Programming Languages*. [online] Statista Infographics. Available at: https://www.statista.com/chart/16567/popular-programming-languages/.

Python version 3.9.10. (2022). *Python Release Documentation.* Available at: https://www.python.org/downloads/release/python-3910/.

Anaconda. (2023). *Anaconda Documentation.* Available at: https://www.anaconda.com.

NumPy. (2023). *NumPy Documentation.* Available at: https://numpy.org.

Pandas. (2022). *Pandas Documentation.* Available at: https://pandas.pydata.org.

PIP. (2023). *PyPI.* Available at: https://pypi.org.

Apple. (2023). *Apple support Apple silicon.* Available at: https://support.apple.com/en-gb/HT211814.

Pygame. (2023). *Pygame documentation.* Available at: https://www.pygame.org/news.

Python Arcade. *Python Arcade Documentation.* Available at: https://api.arcade.academy/en/latest/.

Cocos. (2023). *Cocos Documentation.* Available at: https://www.cocos.com/en.

PyTorch. (2023). *PyTorch Documentation.* Available at: https://pytorch.org.

TensorFlow. (2023). *TensorFlow Documentation.* Available at: https://www.tensorflow.org.

Scikit-learn. (2023). *Scikit-learn Documentation.* Available at: https://scikit-learn.org/stable/.

GitHub. *GitHub Documentation.* Available at: https://github.com.

Yep, T. (2020). *torchinfo*. [online] GitHub. Available at: https://github.com/TylerYep/torchinfo.

pseudoeditor.com. (n.d.). (2023). *Pseudocode Editor Online - PseudoEditor*. [online] Available at: https://pseudoeditor.com/.

Duque. (2023). *Python Containers Documentation.* Available at: https://docs.python.org/3.9/library/collections.html#collections.deque.

Matplotlib. (2023). *Matplotlib Documentation*. Available at: https://matplotlib.org.

Roper, W. (2020). *Infographic: Python Remains Most Popular Programming Language*. [online] Statista Infographics. Available at: https://www.statista.com/chart/21017/most-popular-programming-languages/.

Appendix A
External Materials

The base snake game was adopted from Patrick Leober's GitHub repository.

Link: https://github.com/patrickloeber/python-fun/tree/master/snake-pygame