Exeter Mathematics Certificate

Machine Learning and Neural Networks

Demonstrated through learning to play Snake

Orson Hart
Exeter Mathematics School

EXETER MATHS SCHOOL
FREEDOM FOR THINKERS

Contents

1. Introduction

Machine Learning is a type of artificial intelligence which 'learns' how to accomplish a task, without being explicitly told how to complete it. [1] Instead a machine learning algorithm learns by processing large amounts of data. This is useful for tasks which would be easy for a human to do, but would be hard for them to explicitly describe exactly how to do them to to a computer. Examples of uses of machine learning include image classification and self-driving cars. [2]

The task that I want to achieve is to implement an algorithm which will learn to play Snake, starting from having no knowledge of how to play the game to eventually being able to achieve high scores. This algorithm will make use of a neural network, a system which creates links between inputs and outputs similarly to the operation of the human brain. Completing this task will allow me to build a comprehensive understanding of how machine learning works, and I will be able to use these skills in future projects.

2. Neural Networks

2.1 What is a Neural Network?

Neural Networks are simplistic algorithmic representations of the way that the human brain works, which can take some abstract input data and, by connecting to neurons inside the network, evaluate the correct response to this data. A neuron will have different impacts on other neurons, and the neural network will have to run many times in order to optimise the amount of impact that the neurons have on each other. [3]

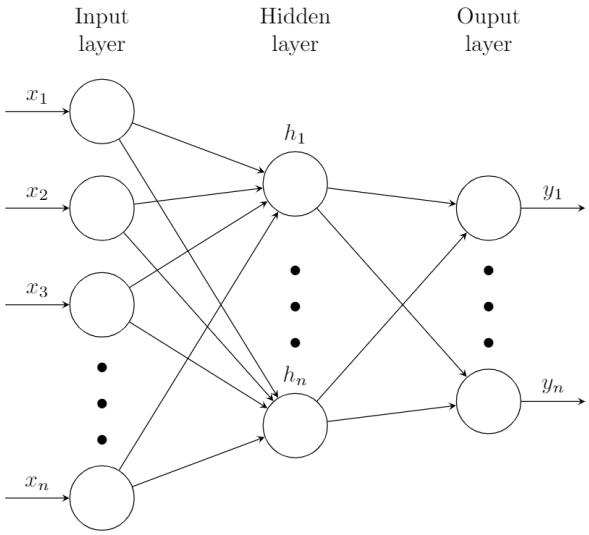

Figure 2.1: **A visual representation of a neural network**

A Neural Network is made up of 3 main components:

- The Input Layer
- The Hidden Layers
- The Output Layer

The Input Layer is the name given to the set of data points that is fed into the neural network.

The Hidden Layers are sets of interconnected nodes that will change based on the values of the nodes in the input layer. The number of hidden layers and number of nodes per hidden layer can be customised to balance accuracy and speed of the network. How the hidden nodes change based on the input nodes is based on parameters known as weights and biases. A weight is a coefficient of the sum of the values of the preceding nodes, and a bias is an added constant. For example, if the sum of the values of the preceding nodes was x, then the value passed to the next node would be $wx + b$, where w and b are the associated weight and bias [3]. These weights and biases change as the neural network operates in order to create a more effective output.

The Output Layer is the name given to the set of nodes which determine possible decisions. Once the input neurons have been processed and sent through the hidden layers, the network gives values to the output neurons, representing the best decision to make as a result of the input data. The value of an output neuron represents the probability that a set of input data corresponds to that particular output. Therefore, the output neuron which has the highest value has the highest probability of being correct, thus representing the final evaluation of the network.

2.2 Training a Network

A neural network can be "trained" to optimise the weights and biases by giving it a large amount of input data linked to the corresponding correct output. The network can use this data to select parameters for links between neurons, and find weights and biases in a way that has the best correspondence between inputs and outputs. In order to quantify how good a network is performing, a cost function is used [3]. This is also referred to as the loss function or the mean squared error, and evaluates how far from the true value the neural network's output is. To optimise a neural network, the cost function is minimised.

$$C(w, b) = \frac{1}{2n} \sum_{x} |y(x) - a|^2. \tag{2.1}$$

where:

w is the set of weights

b is the set of biases

n is the number of inputs

x is the set of inputs

$y(x)$ is the set of outputs the network is trying to match

a is the set of outputs that the network evaluates when given x as an input

In order to minimise the cost function, the network undergoes a process known as *Backpropagation* [4]. Simply put, this process finds the gradient of the cost function with respect to a weight and bias, and finds a local minimum across the set of weights and biases, which is used as an approximation for the global minimum. This determines the values of each weight and bias to be used to create an optimised network.

3. Creating a Snake Game

3.1 What is Snake?

Snake is a game in which the player controls a snake moving around cells on a grid. The snake is made up of a head and a body, and the body will follow in a line the previous positions of the head. The snake dies if it touches its own body or the edge of the grid, and will increase in length if the snake eats fruit which appears randomly around the grid. The aim of the game is to get the snake as long as possible without dying.

In order to train a neural network to play Snake, I first had to actually program the game itself. In order to do this I used the Python library pygame, as it is simple and allowed me to quickly create a Snake game. Python also has libraries used for machine learning, which would be useful further into the project.

3.2 Implementing Game Framework

```
import pygame
pygame.init()
```

Then, using pygame we can create a window to display the game. I also implemented a grid-based system, as this is what the Snake game will run on.

```
# Initialise configuration variables.
width = 20
height = 15
cell = 20

BLACK = (12, 16, 18)
RED = (255, 58, 59)
GREEN = (102, 254, 76)

# Create screen object.
screen = pygame.display.set_mode((width * cell,height * cell))
pygame.display.set_caption("Snake")

# Display screen.
screen.fill(BLACK)
pygame.display.update()
```

Next, I created a class for the Snake and the Apple. A class is a way of grouping together associated procedures and variables.

```
class Snake():
    def __init__(self):
        # Create empty list of positions.
        self.pos = []

        # Populate position list.
```

```
        self.pos.append(pygame.Vector2(2,0))
        self.pos.append(pygame.Vector2(1,0))
        self.pos.append(pygame.Vector2(0,0))

        # Set direction.
        self.direction = pygame.Vector2(1,0)

class Apple():
    def __init__(self):
        # Set position to a random position on the board.
        self.pos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
```

The Snake object will hold a list of vectors to represent the position of each segment in the snake, and a single vector to represent the direction in which it is going. When a snake is created, it will start at the top left of the screen and be moving to the right.

The Apple object will hold a single vector which will represent its position on the screen.

Next, the game needs a loop which will contain code that will execute each frame, and draw it to the screen.

```
def playGame():
    # Create snake and apple object.
    snake = Snake()
    apple = Apple()

    # GAME LOOP
    play = True
    while play:
        # Run at 10fps.
        clock.tick(10)
        # Clear the screen.
        screen.fill(BLACK)

        # Draw apple.
        pygame.draw.rect(screen, GREEN, (apple.pos.x * cell, apple.pos.y * cell,cell,cell))

        # Draw all snake segments.
        for snakepos in snake.pos:
            pygame.draw.rect(screen, RED,(snakepos.x * cell,snakepos.y * cell,cell,cell))

        # Update the display.
        pygame.display.update()

playGame()
```

This draws the snake and the apple to the screen, and by using the pygame function `clock.tick(10)`, the game logic and display will update 10 times per second. The result is shown in Figure 3.1:

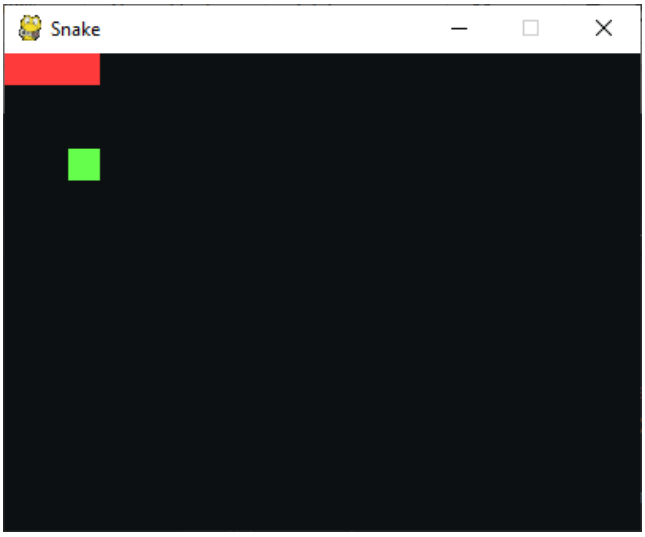

Figure 3.1: **A screenshot from my initial implementation of the Snake game**

3.3 Implementing Game Logic

This is the majority of the setup needed to make a Snake game. Next I needed to implement the logic of Snake, which is as follows:

1. The snake must move in the chosen direction.
2. If the snake collides with itself or the wall then it is game over.
3. If the snake collides with an apple then the apple moves and the length of the snake increases.

1. Movement
As I decided to implement both the snake's position and its direction as vectors, I could use vector addition to change the position of the head. To change the position of the rest of the body, I looped backwards through the list of Snake segments and propagated their position vectors backwards.

```
def moveSnake(self):
        # Move body.
        for i in range(len(self.pos)-1,0,-1):
            self.pos[i].x = self.pos[i-1].x
            self.pos[i].y = self.pos[i-1].y

        # Move head.
        head = self.pos[0]
        head.x += self.direction.x
        head.y += self.direction.y
```

2. Game Over
This is a simple check to see if the position of the snake is outside the board, or if there is more than one occurrence of the same position vector within the snake.

```
def checkGameOver(self):
        for pos in self.pos:
            if pos.x < 0 or pos.x >= width:
                return True
            if pos.y < 0 or pos.y >= height:
                return True
            if self.pos.count(pos) > 1:
                return True
        return False
```

3. Eating Apples
First, I check if the location of the snake's head is the same as the location of the apple and, if so, change the location of the apple to one that is not occupied by the snake and add a segment to the snake.

```
# If snake eats apple
        if snake.pos[0] == apple.pos:
            # Set apple position to new random position not occupied by snake.
            newApplePos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
            while newApplePos in snake.pos:
                newApplePos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
            apple.pos = newApplePos

            # Add new snake segment.
            snake.pos.append(pygame.Vector2(0,0))
```

All that was left was to allow the user to change the direction of the Snake using their keyboard, and add a counter to keep track of the number of moves and score. Examples are shown in Figure 3.2

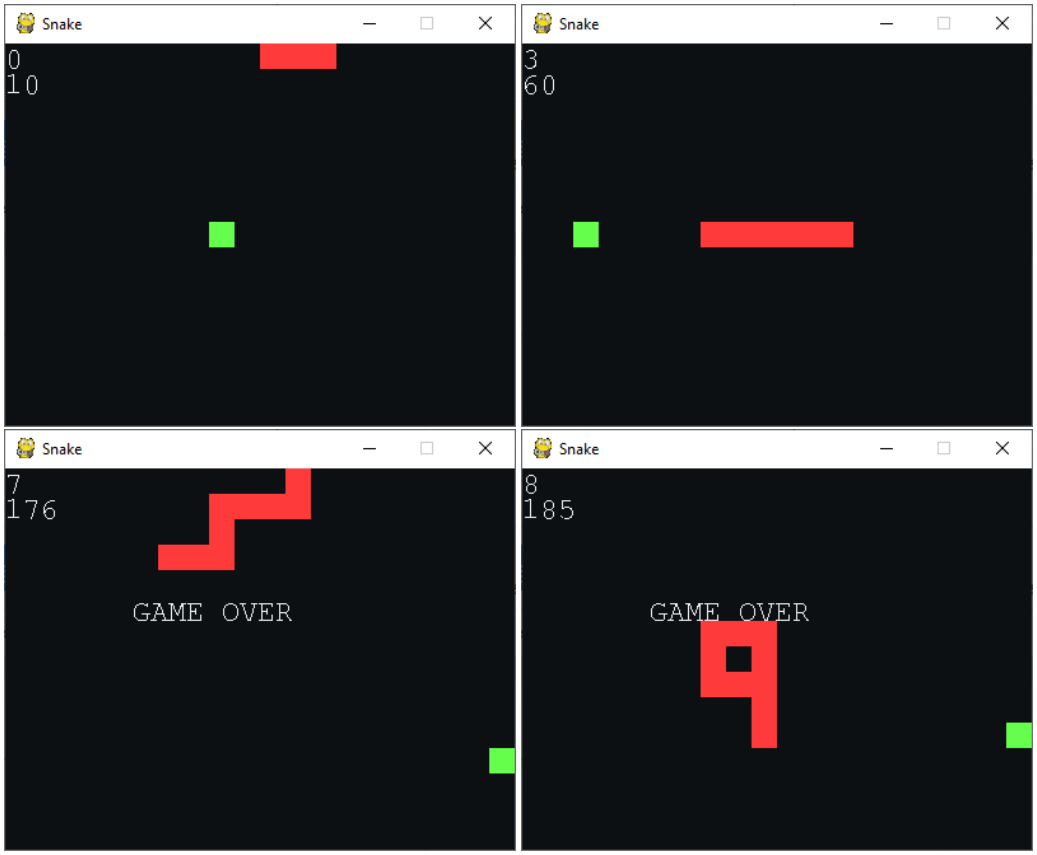

Figure 3.2: **Examples of Snake gameplay**

4. Neural Network Design

4.1 Input and Output Nodes

A Neural Network must take some abstract inputs which it will then use to predict what to do next, where every input must be normalised to have a value between 0 and 1, inclusive. In order to keep the project simple, I wanted to limit the number of observations in a data sample. I decided on the following observations.

These observations keep the snake alive:

- Can the snake move forwards without dying?
- Can the snake move left without dying?
- Can the snake move right without dying?

These observations move the snake towards food:

- Is the food in front of the snake?
- Is the food on the left of the snake?
- Is the food on the right of the snake?
- Is the food *directly* ahead of the snake?

All of these are Boolean variables (take the value of either True or False), so can be represented as either a 0 or 1, which means they can be used as input nodes to the neural network.

The neural network will have 3 output nodes:

- Should the snake turn left?
- Should the snake keep going forwards?
- Should the snake turn right?

Whichever output node has the highest value (i.e - the most effective move), will be the move the snake will make.

4.2 Hidden Layers

The internal structure of a neural network can be changed to balance accuracy against computation time. A higher number of layers and nodes per layer will generally increase the accuracy of a neural network, although using a very large number of nodes will dramatically increase processing time whilst providing little benefit to the accuracy of epothe network.

I will use two hidden layers consisting of 10 nodes as this should provide a good level of accuracy whilst keeping computation time relatively low. As my neural network has a relatively low number of input and output nodes, a large number of hidden nodes is not required.

4.3 TensorFlow

I will be using the TensorFlow library for Python which allows the creation of neural networks to be streamlined. Included with the TensorFlow library is a library called Keras, which is an open source neural network library which works in conjunction with TensorFlow to enable fast experimentation with deep neural networks. As Keras focuses on being user-friendly, modular, and extendible - it will be ideal to work with during my project.

4.4 Generating and Processing Training Data

In order for a neural network to associate an input state with an output state, it needs a large amount of training data. In some scenarios, there is a large amount of training data already available - but for a Snake neural network this would require someone to manually play many of games of Snake.

Instead, it would be more efficient and effective to randomly play games of Snake in order to generate training data for the network. When making a move, the state of the snake before the move can be stored, and once a random move has been made, it can be evaluated whether it has been a beneficial move or not. If the random move it made was beneficial, then the observations and the associated move can be added as to the set of training data for the neural network to learn from.

Below is the code which I implemented for randomly changing the direction of the snake, determining if the spaces surrounding the snake were clear.

```
# Functions change direction of snake.
    def turnLeft(self):
        self.direction = self.direction.rotate(-90)
    def turnRight(self):
        self.direction = self.direction.rotate(90)

    # Determine if a spot on the board is clear.
    def isClearSpot(self, spot):
        if spot in self.pos:
            return False
        if spot.x < 0 or spot.x >= width:
            return False
        if spot.y < 0 or spot.y >= height:
            return False
        return True

    # Returns if the spot in front of the snake is clear.
    def isClearAhead(self):
        return self.isClearSpot(self.pos[0] + self.direction)
    # Returns if the spot to the left of the snake is clear.
```

```
    def isClearLeft(self):
        return self.isClearSpot(self.pos[0] + self.direction.rotate(-90))
    # Returns if the spot to the right of the snake is clear.
    def isClearRight(self):
        return self.isClearSpot(self.pos[0] + self.direction.rotate(90))

    # Moves the snake in a random direction (left, straight, right).
    def makeRandomMove(self):
        choice = random.randint(0,2)
        if choice == 0:
            self.turnLeft()
        elif choice == 2:
            self.turnRight()
        self.move()
        return choice
```

The neural network learns from the data by cycling through the full training dataset a number of times. Each cycle is known as an epoch [5]. Each epoch processes the data in a new random order, which will allow the network to run better when presented with previously unseen data.

5. Neural Network Implementation

5.1 Keeping the Snake Alive

Initially I developed a neural network which kept the snake alive. My training data consisted of 3 observations

- Is the spot in front of the snake clear?
- Is the spot to the left of the snake clear?
- Is the spot to the right of the snake clear?

I decided to add a move to the training dataset if, once the move has been made, the snake remained alive. Below is the code used to generate a data sample. If the program is trying to generate data then it will store the direction that the snake randomly moved in to include in the training sample, otherwise it will predict what the best move should be based on the generated model.

```
# Create a sample from the known data.
        sample = [int(snake.isClearLeft()), int(snake.isClearAhead()), int(snake.isClearRight())]
        if generate_data == True:
            # Create a corresponding label to the move which is made.
            label = [0,0,0]
            label[snake.makeRandomMove()] = 1
        else:
            # Predict what to do based on the data sample.
            prediction = model.predict([sample])
            direction = np.argmax(prediction)
            # Change the direction of the Snake based on the output of the model.
            if direction == 0:
                snake.turnLeft()
            elif direction == 2:
                snake.turnRight()
            # Move the snake.
            snake.move()
```

The following code adds the recorded observations to the training datset if the snake survived.

```
# Add the sample and label to the training dataset if the snake has not died.
        if generate_data == True and play == True:
            train_samples.append(sample)
            train_labels.append(label)
```

These final lines of code are used to generate the data, format it in such a way that it can be processed by a neural network, create a neural network and then train it on the training data.

```
# Gather data from 100 random games.
for _ in range(100):
    playGame(generate_data=True,fps=None)
```

```
# Convert training data to numpy arrays.
train_samples = np.array(train_samples)
train_labels = np.array(train_labels)

# Create a model using keras.
model = keras.Sequential([
    # 3 input nodes.
    keras.layers.Flatten(input_shape=(3,)),
    # 10 hidden nodes.
    keras.layers.Dense(10, activation="relu"),
    # 3 output nodes.
    keras.layers.Dense(3, activation="softmax")
    ])

# Compile the model
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

# Play the game using the generated model.
while True:
    model.fit(x=train_samples,y=train_labels,epochs=3)
    playGame(generate_data=False, fps=10)
```

When executed, this code creates a neural network which keeps the snake alive.

```
Train on 1514 samples
Epoch 1/3
1514/1514 [==============================] - 1s 408us/sample - loss: 0.6474 - accuracy: 0.6651
Epoch 2/3
1514/1514 [==============================] - 0s 59us/sample - loss: 0.6397 - accuracy: 0.6651
Epoch 3/3
1514/1514 [==============================] - 0s 59us/sample - loss: 0.6362 - accuracy: 0.6651
```

Figure 5.1: **An example output of training this neural network.**

In Figure 5.1 it can be seen that the loss function is very high, averaging 0.64 across 3 epochs, and the accuracy was consistently 0.6651. In fact, using this model, no matter the number of games played, the number of hidden nodes, or the number of epochs used, the accuracy could not reach over 0.6667 This was a limitation of the small number (3) of input nodes.
The network would occasionally generate a model which would not keep the snake alive, although once the same training data was processed again by a new network, it could be used to keep the snake alive indefinitely. Due to the random nature of generating training data, different datasets would create different patterns of movement by the snake. These included:

- Circling around the edge of the screen in a straight line
- Circling around the edge of the screen whilst moving towards and away from the edge.
- Spinning in a tight circle
- Moving vertically, turning around at the top and the bottom of the screen.

These different patterns of movement show how different training datasets can cause different behaviour on a system, and even if they all achieve the same goal (staying alive), they do so in different ways.

5.2 Getting the Snake to Eat Food

Once I developed a neural network which could keep the snake alive, I was able to modify it to enable the snake to eat food. To do this, I needed to add new observations to the sample, which described the location of

the food relative to the snake. The two observations I added were:

- If the food was in front of the snake.
- If the food was on the left of the snake, the right of the snake, or directly in front of the Snake.

To record these observations, I used the fact that I had implemented the snake's direction as a vector, and that I had stored the position of elements on the screen as vectors, to find the angle between the snake's movement and the food, which allowed me to determine the above points.
I used the dot product (5.1) to calculate the required angle, where **A** represents the direction vector of the snake, and **B** represents the vector from the head of the snake to the food.

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}||\mathbf{B}| \cos \theta \tag{5.1}$$

I used the same method of playing games whilst making random moves to generate training data. To decide whether a sample should be added to the training dataset, I determined whether, after making a random move, the snake stayed alive and also got closer to the food. I calculated the distance to the food using *Pythagoras' theorem*, however, as the snake is not free to move in all directions - and is constrained to moving along a grid - it would have been equally valid to calculate the *Taxicab distance* between the two points. [6]
Below are the functions used to find the distance between two position vectors, and the angle between two direction vectors.

```
# Find the distance between two position vectors.
def distanceBetween(p1, p2):
    x = abs(p1.x - p2.x)
    y = abs(p1.y - p2.y)
    d = np.sqrt(x**2 + y**2)
    return d

# Find the angle between two direction vectors.
def angleBetween(v1,v2):
    ang = v1.angle_to(v2)
    if ang > 180:
        ang = ang-360
    if ang <= -180:
        ang = ang+360
    return ang
```

Below is the code used to create the data required for the sample:

```
# Store angle between snake and food.
        ang = angleBetween(apple.pos-snake.pos[0],snake.direction)

        # Store whether the food is in front of the snake or not.
        if abs(ang) < 90:
            food_ahead = 1
        else:
            food_ahead = 0

        # Store the direction of the food relative to the Snake. (-1 RIGHT, 0 STRAIGHT, 1 LEFT)
        food_direction = np.sign(ang)

        # Store the distance between the snake and food before move has been made.
        initial_distance = distanceBetween(snake.pos[0], apple.pos)

        # Create a sample from the known data.
        sample = [int(snake.isClearLeft()), int(snake.isClearAhead()), int(snake.isClearRight()),
    food_ahead, food_direction]
```

Below is the code which determines if a data sample should be added to the training dataset.

```
# Calculate distance between snake and apple after random move.
        new_distance = distanceBetween(snake.pos[0], apple.pos)

        # If the snake has not died, and has gotten closer to the food, add the data sample
        # to the set of training samples.
        if generate_data == True and play == True and new_distance < initial_distance:
            train_samples.append(sample)
            train_labels.append(label)
```

The full code listing can be found in the appendix.

```
Epoch 1/5
5975/5975 [==============================] - 1s 144us/sample - loss: 0.4898 - accuracy: 0.7278
Epoch 2/5
5975/5975 [==============================] - 0s 62us/sample - loss: 0.3960 - accuracy: 0.7782
Epoch 3/5
5975/5975 [==============================] - 0s 59us/sample - loss: 0.3518 - accuracy: 0.7791
Epoch 4/5
5975/5975 [==============================] - 0s 59us/sample - loss: 0.3321 - accuracy: 0.7758
Epoch 5/5
5975/5975 [==============================] - 0s 62us/sample - loss: 0.3213 - accuracy: 0.7829
```

Figure 5.2: **An example output of training this neural network.**

I used a sample of 1000 training games which generated 5975 training samples, and trained across 5 epochs. The loss function was reduced to 0.32, and the accuracy of the network increased to 0.78, which is a noticeable improvement over the performance of the initial network which exclusively tried to keep the snake alive. This improvement is due to the increased number of observations, and the stricter criteria for choosing samples to add to the training set.
Overall, this implementation of a neural network was successful as the snake would move towards food to achieve higher scores. A more detailed breakdown of results can be found in the next chapter.

6. Results

An analysis of how the network performs when trained on different amounts of training data.

6.1 Loss and Accuracy

I decided to compare how the size of a training data set can impact performance on the network. To do this, I trained neural networks on data sets from 100, 250, 500, 1000, and 2500 randomly played games, and compared the average score and distribution of scores across 100 games played using the network.
To keep the comparisons fair, I trained the network over the same number of epochs (5). Looking at the loss functions and accuracy of the network after training on the data sets across 5 epochs, the results are shown in Figure 5.3.

Number of Training Games	Loss	Accuracy
100	0.4996	0.7398
250	0.3479	0.7955
500	0.3404	0.7779
1000	0.3126	0.7894
2500	0.3093	0.7820

Figure 6.1: **Table showing how the number of training games affects loss and accuracy**

These results show that as the amount of training data increases, the loss of the network decreases and the accuracy increases.

6.2 Comparison of Average Scores

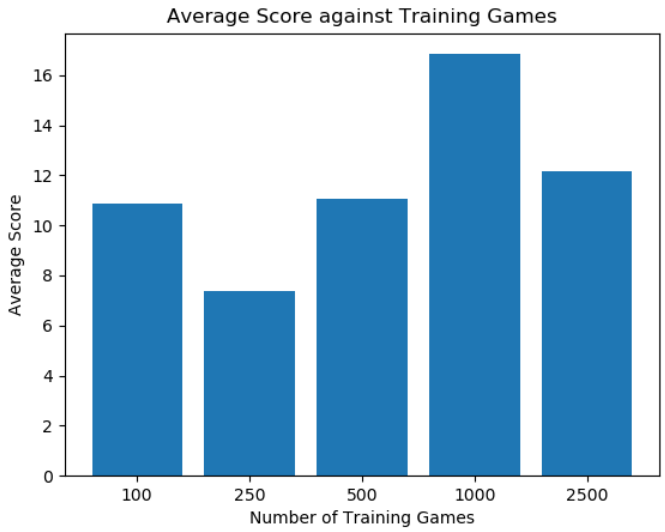

Figure 6.2: **The average score across 100 games against the number of training games**

This shows that, whilst the trend of the average score is that it increases as the number of training games increases, there is not a consistent growth, and sometimes there is a noticeable decrease in the average score - for example from 100 to 250 and 1000 to 2500. This could be because the random games that the training set takes its training data from did not produce enough data, or that the random parameters of the game causes skewed results, for example due to the random placement of fruit in the grid.
The case from 1000 to 2500 could also be a result of overfitting [7], where patterns of data within a dataset cause the network becomes biased towards certain behaviours. For example, in this case the training dataset might not have had enough examples of the snake avoiding its own tail, which would cause the network to prioritise moving towards food over avoiding obstacles, leading to lower scores on average.

6.3 Distribution of Scores

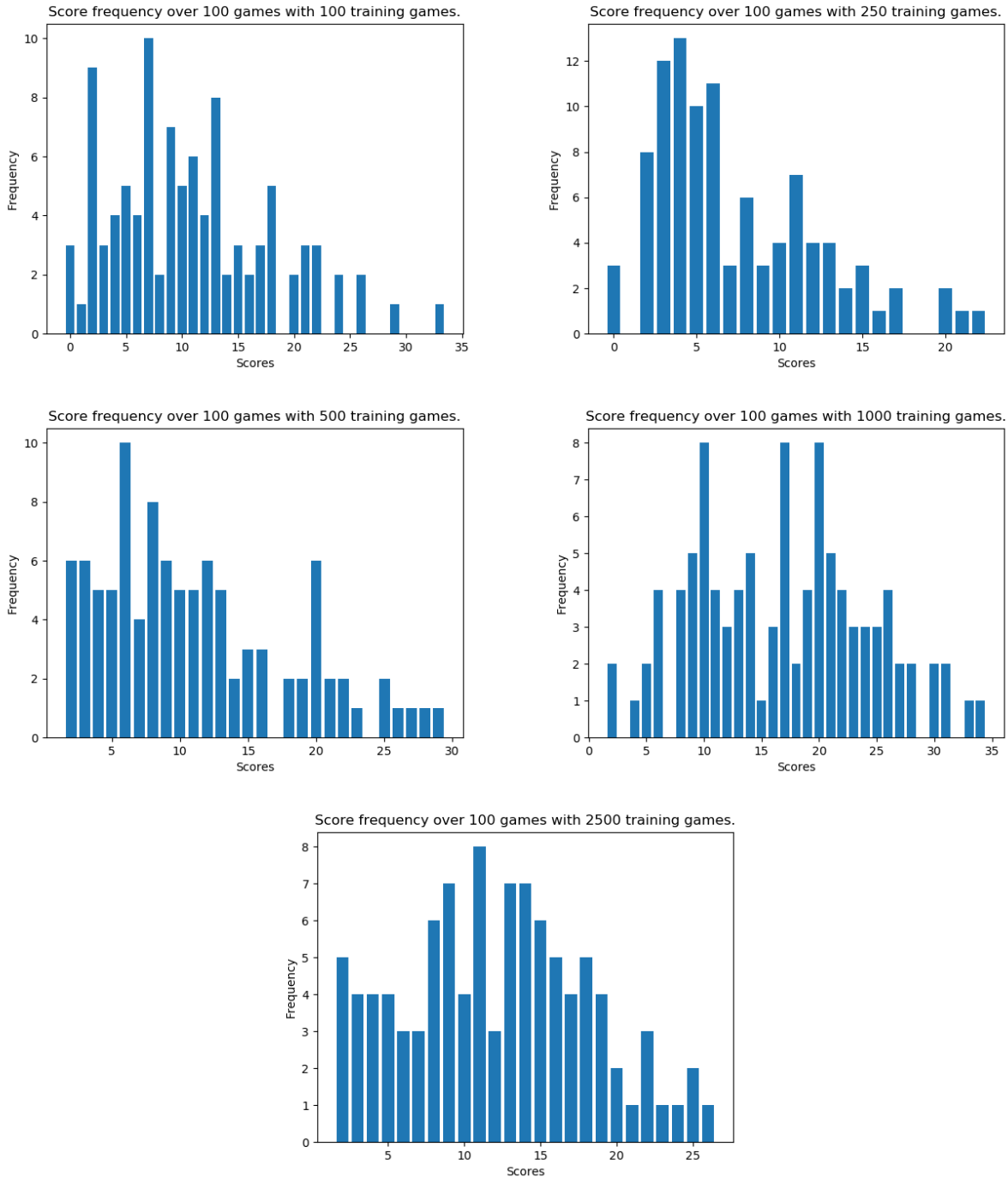

Figure 6.3: **Distribution of scores across different number of training games**

The pattern of the data shown in Figure 6.3 is that generally, as the number of training games increase, spread of the data decreases and the network more consistently achieves the same result. This can be seen when comparing the data for 100 and 1000 training games, as the score distribution from 100 training games shows a positive skew, whereas the score distribution from 1000 training games has produced a more symmetric distribution.

The most symmetric distribution of scores is when the network has been trained across 2500 random games, although this increase in consistency has decreased the overall average score. It can also be noted that the only time that the network has scored 0 points was when the network was trained across 100 random games. This can only happen by the snake hitting the wall rather than its own tail, and the network learnt to overcome this behaviour once it was trained across more games. The network trained across 100 random games was also the most erratic network, as although it achieved this low score of 0, it also achieved the second highest high score (33), only one point behind the overall highest (34, achieved by the network trained across 1000 games).

A pattern I did notice, which is not reflected in these results, is that when using a lower number of training games (less than 500), the snake would move in a diagonal zig-zag to try and reach the food, whereas with an increased number of training games, the snake moved in longer, straight lines. Due to the constraints of moving along a grid, both techniques would produce paths of equal length.

7. Evaluation

7.1 Limitations

Due to the structure of the data samples in the neural network, the snake could not 'see' obstacles which were not directly adjacent to it, whilst it could locate food from across the board. This meant the snake would often move towards food without avoiding its own tail in front of it, causing the snake to die.
If I were to continue the project further, I would add more observations to my samples in order to create a more comprehensive system for reaching higher scores. I would also want to introduce a genetic algorithm system to optimise a network, where parameters networks which perform well are "bred" and mutated in order to create new, better performing networks. [8]
Analysis of the performance of the networks was also limited due to the speed at which games could be played, as each move had to be processed by the network before it could be made. This could be calculated at a stable rate of 10 moves per second, although it is unclear if this performance could be matched if the number of observations in the data sample, and therefore complexity of the network, increased. A more detailed analysis of the effects of the size of the training dataset would require more examples of the performance of the network when trained across a wider range of datasets.

7.2 Conclusion

In conclusion, the project was a success as I was able to produce a neural network which could consistently produce good scores in Snake and I learnt a lot about how neural networks work, and what is required for effective performance.
I was also able to identify patterns of behaviour exhibited by the network when trained across different sized data sets, and that the patterns of data within a data set can greatly influence the behaviour of a network trained on it. For example, a larger data set will not always produce more effective results if the observations within that set are repetitive and do not accurately represent the full range of possible situations.

References

[1] Michael Neilsen
http://neuralnetworksanddeeplearning.com/about.html
Accessed 2020-02-20.

[2] Stanford University
https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Applications/index.html
Accessed 2020-02-20.

[3] Michael Neilsen
http://neuralnetworksanddeeplearning.com/chap1.html
Accessed 2020-02-04.

[4] Michael Neilsen
http://neuralnetworksanddeeplearning.com/chap2.html
Accessed 2020-02-04.

[5] DeepAI
https://deepai.org/machine-learning-glossary-and-terms/epoch
Accessed 2020-02-18.

[6] Wikipedia
https://en.wikipedia.org/wiki/Taxicab_geometry
Accessed 2020-02-18.

[7] Piotr Skalski
https://towardsdatascience.com/preventing-deep-neural-network-from-overfitting-953458db800a
Accessed 2020-02-20.

[8] Ahmed Gad
https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b
Accessed 2020-02-22.

[9] 3Blue1Brown's Neural Network Playlist
https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
Accessed 2019.

[10] TheAILearner
https://theailearner.com/2018/04/19/snake-game-with-deep-learning/
Accessed 2019.

[11] Peter Binggeser
https://becominghuman.ai/designing-ai-solving-snake-with-evolution-f3dd6a9da867
Accessed 2019.

[12] Slava Korolev
https://towardsdatascience.com/today-im-going-to-talk-about-a-small-practical-example-of-using-neural-networks-training-one-to-6b2cbd6efdb3
Accessed 2019.

Full Code Listing

```
# Import required modules.
import pygame, random, time
import tensorflow as tf
from tensorflow import keras
import numpy as np

# Initialise Pygame.
pygame.init()

# Create font object.
myfont = pygame.font.SysFont("monospace", 24)
# Create clock object.
clock = pygame.time.Clock()

# Initialise colour constants.
WHITE = (236, 240, 241)
BLACK = (12, 16, 18)
RED = (255, 58, 59)
GREEN = (102, 254, 76)

# Initialise configuration variables.
width = 20
height = 15
cell = 20

# Create screen object.
screen = pygame.display.set_mode((width * cell,height * cell))
pygame.display.set_caption("Snake")

# Snake object.
class Snake():
    def __init__(self):
        # Create empty list of positions.
        self.pos = []
        self.addSegment = False

        # Populate posiiton list.
        self.pos.append(pygame.Vector2(2,0))
        self.pos.append(pygame.Vector2(1,0))
        self.pos.append(pygame.Vector2(0,0))

        # Set direction.
        self.direction = pygame.Vector2(1,0)

    # Functions change direction of snake.
    def turnLeft(self):
        self.direction = self.direction.rotate(-90)
    def turnRight(self):
        self.direction = self.direction.rotate(90)

    # Determine if a spot on the board is clear.
    def isClearSpot(self, spot):
        if spot in self.pos:
            return False
        if spot.x < 0 or spot.x >= width:
            return False
        if spot.y < 0 or spot.y >= height:
            return False
        return True
```

```
60
61      # Returns if the spot in front of the snake is clear.
62      def isClearAhead(self):
63          return self.isClearSpot(self.pos[0] + self.direction)
64      # Returns if the spot to the left of the snake is clear.
65      def isClearLeft(self):
66          return self.isClearSpot(self.pos[0] + self.direction.rotate(-90))
67      # Returns if the spot to the right of the snake is clear.
68      def isClearRight(self):
69          return self.isClearSpot(self.pos[0] + self.direction.rotate(90))
70
71      # Moves the snake in a random direction (left, straight, right).
72      def makeRandomMove(self):
73          choice = random.randint(0,2)
74          if choice == 0:
75              self.turnLeft()
76          elif choice == 2:
77              self.turnRight()
78          self.move()
79          return choice
80
81      # Function moves snake.
82      def move(self):
83          # If the snake has eaten an apple on the previous move...
84          if self.addSegment == True:
85              # Append a segment to the body.
86              self.pos.append(pygame.Vector2(0,0))
87              # Reset the flag.
88              self.addSegment = False
89
90          # Propogate the positions of the snake segments backwards.
91          for i in range(len(self.pos)-1,0,-1):
92              self.pos[i].x = self.pos[i-1].x
93              self.pos[i].y = self.pos[i-1].y
94
95          # Move head.
96          head = self.pos[0]
97          head.x += self.direction.x
98          head.y += self.direction.y
99
100     # Function returns True if the snake goes off the board or hits itself.
101     def checkGameOver(self):
102         for pos in self.pos:
103             if pos.x < 0 or pos.x >= width:
104                 return True
105             if pos.y < 0 or pos.y >= height:
106                 return True
107             if self.pos.count(pos) > 1:
108                 return True
109         return False
110
111 # Apple object.
112 class Apple():
113     def __init__(self):
114         # Set position to a random position on the board.
115         self.pos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
116
117 # Display text at the specified position.
118 def displayText(text, pos):
119     label = myfont.render(text, False, WHITE)
120     screen.blit(label, pos)
121     pygame.display.update()
```

```
122
123 # Find the distance between two position vectors.
124 def distanceBetween(p1, p2):
125     x = abs(p1.x - p2.x)
126     y = abs(p1.y - p2.y)
127     d = np.sqrt(x**2 + y**2)
128     return d
129
130 # Find the angle between two direction vectors.
131 def angleBetween(v1,v2):
132     ang = v1.angle_to(v2)
133     if ang > 180:
134         ang = ang-360
135     if ang <= -180:
136         ang = ang+360
137     return ang
138
139 # Initialsie lists of training samples and labels.
140 train_samples = []
141 train_labels = []
142
143 def playGame(generate_data, fps=None):
144     # SETUP
145     # Create snake and apple object.
146     snake = Snake()
147     apple = Apple()
148
149     # Initialise instance variables.
150     score = 0
151     moves = 0
152
153     # GAME LOOP
154     play = True
155     while play:
156         # Run at specifed fps.
157         if fps != None:
158             clock.tick(fps)
159         # Clear the screen.
160         screen.fill(BLACK)
161
162         # Increment moves.
163         moves = moves + 1
164
165         # For every pygame event...
166         for event in pygame.event.get():
167             # If quit then quit.
168             if event.type == pygame.QUIT:
169                 pygame.quit()
170
171         # Store angle between snake and food.
172         ang = angleBetween(apple.pos-snake.pos[0],snake.direction)
173
174         # Store whether the food is in front of the snake or not.
175         if abs(ang) < 90:
176             food_ahead = 1
177         else:
178             food_ahead = 0
179
180         # Store the direction of the food relative to the Snake. (-1 RIGHT, 0 STRAIGHT, 1 LEFT)
181         food_direction = np.sign(ang)
182
183         # Store the distance between the snake and food before move has been made.
```

```
        initial_distance = distanceBetween(snake.pos[0], apple.pos)

        # Create a sample from the known data.
        sample = [int(snake.isClearLeft()), int(snake.isClearAhead()), int(snake.isClearRight()),
    food_ahead, food_direction]
        if generate_data == True:
            # Create a corresponding label to the move which is made.
            label = [0,0,0]
            label[snake.makeRandomMove()] = 1
        else:
            # Predict what to do based on the data sample.
            prediction = model.predict([sample])
            direction = np.argmax(prediction)
            # Change the direction of the Snake based on the outcome of the model.
            if direction == 0:
                snake.turnLeft()
            elif direction == 2:
                snake.turnRight()
            # Move the snake.
            snake.move()

        # If snake eats an apple..
        if snake.pos[0] == apple.pos:
            # Set apple position to new random position not occupied by snake.
            newApplePos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
            while newApplePos in snake.pos:
                newApplePos = pygame.Vector2(random.randint(0,width-1),random.randint(0,height-1))
            apple.pos = newApplePos

            # Add new snake segment next move.
            snake.addSegment = True

            # Increment score.
            score = score+1

        # If the snake dies
        if(snake.checkGameOver()):
            play = False

        # Draw apple.
        pygame.draw.rect(screen, GREEN, (apple.pos.x * cell, apple.pos.y * cell,cell,cell))

        # Draw all snake segments.
        for snakepos in snake.pos:
            pygame.draw.rect(screen, RED,(snakepos.x * cell,snakepos.y * cell,cell,cell))

        # Update score and moves display every tick.
        displayText(str(score),(0,0))
        displayText(str(moves),(0,20))

        # Update the display.
        pygame.display.update()

        # Calculate distance between snake and apple after random move.
        new_distance = distanceBetween(snake.pos[0], apple.pos)

        # If the snake has not died, and has gotten closer to the food, add the data sample
        # to the set of training samples.
        if generate_data == True and play == True and new_distance < initial_distance:
            train_samples.append(sample)
            train_labels.append(label)

```

```
# Get how many games to play from the user.
training_games = int(input("How many training games?: "))

# Gather data from playing multiple random games.
for _ in range(training_games):
    playGame(generate_data=True,fps=None)

# Convert training data to numpy arrays.
train_samples = np.array(train_samples)
train_labels = np.array(train_labels)

# Create a model using keras.
model = keras.Sequential([
    # 5 input nodes.
    keras.layers.Flatten(input_shape=(5,)),
    # 1st hidden layer of 10 nodes.
    keras.layers.Dense(10, activation="relu"),
    # 2nd hidden layer of 10 nodes.
    keras.layers.Dense(10, activation="relu"),
    # 3 output nodes.
    keras.layers.Dense(3, activation="softmax")
    ])

# Get how many epochs to fit the model across.
epochs = int(input("How many epochs?: "))
# Complie the model, optimising for accuracy.
model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
# Fit the training data to the model.
model.fit(x=train_samples,y=train_labels,epochs=epochs)

# Play the game using the generated model.
while True:
    playGame(generate_data=False, fps=10)
```