

AKDENİZ UNIVERSITY
DEPARTMENT OF COMPUTER ENGINEERING



*ARTIFICIAL INTELLIGENCE
PROJECT REPORT*

Temmuz Burak Yavuzer

20160808026

Contents

<i>1) Introduction</i>	3
1.1 Abstract	3
1.2 Game Definition	4
1.3 Project Definition	4
1.4 Prerequisites and Usage	5
<i>2) Problem Definition and Algorithms</i>	6
2.1 Breadth First Search.	6
2.2 Neural Network	9
<i>3) Result and Comparison</i>	10
3.1 Success rates and Statistics	10
3.2 Performance of Computation/Memory	14
3.3 Difficulties	14
<i>4) Implementation</i>	15
4.1 BFS.	15
4.2 Neural Network	16
<i>5) Conclusion and Future Works</i>	18
<i>Bibliography</i>	19

1)Introduction

1.1)Abstract

Snake Game is the common name for a video game concept where the player maneuvers a line which grows in length, with the line itself being a primary obstacle. Playing with computer this game requires artificial intelligence (AI). Although, there have been many AI algorithms implemented to Snake Game, it is still open to be improved and find an optimal strategies in choosing best move and minimizing the execution time. In this project, analysis and comparison between standard Breadth First Search and Q function by using a Neural Network Algorithms are carried out at the Snake game software written with Python in terms of length of the snake. In addition, Experiments are done by different conditions, i.e. size of the screen for Breadth First Search, size of the screen and the training size for the Neural Network Algorithm. The result of this project and experiments are Breadth First Search is faster and more accurate compare to the Q function by using a Neural Network Algorithms for small experiments.

1.2)Game Definition

This is the classic version of the most popular mobile and computer game named “SNAKE”. The main objective of this game is to feed an increasing length of a snake with food particles which are found at random positions, picking up bonus mongooses that occur at regular intervals.

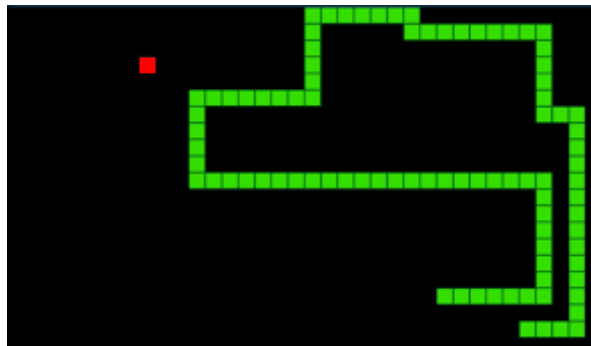


Figure 1.1: Snake Game

1.3)Project Definition

This project is a Snake game written with Python programming language which uses the Breadth First Search, Q function by using a Neural Network algorithm to find the possible best move. The pygame module is used for a graphical user interface so that player can compete with the AI on a screen gridded as 10 x 10 which is base screen size displayed on the screen. For the purpose of adding tiles and giving the algorithms to assess easily, 2D numpy array is used for the board. Pygame module is used for converting the array board into a graphical representation. The players is able to move the snake to the as using the keyboard for normal versions, but in that version that game is updated and AI tile is displayed. For Breadth First Search algorithm and Q function by using a Neural Network algorithm game will be played and displayed by the computer for 10 times. At the end there will be a average snake length will be displayed.

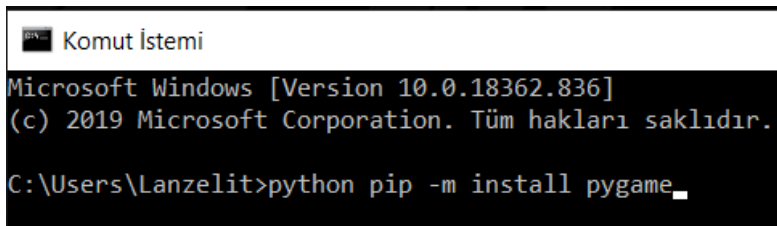
1.4) Prerequisites and Usage

Used modules and environment are listed below :

PyCharm 2020.1.1
Python 3.6
Tensorflow 2.0.0
Numpy 1.18.4
Pygame 1.9.6
Keras 2.3.1
Random
Copy
Pickle

Possible Errors:

While trying to use the code some possible errors could occur. While using Anaconda for Python and Tensorflow , version of the Tensorflow could not work for your Python version. For example Tensorflow is not working on Python 3.7 which can be solved by downgrading the Python version to 3.6 . While using PyCharm, Pygame could not work because of the version of the PyCharm's module management which is not very stable for Pygame. Instead of that “python -m pip install pygame “ line could be run on the command prompt for solving the issue.



```
Komut İstemi
Microsoft Windows [Version 10.0.18362.836]
(c) 2019 Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\Lanzelit>python -m pip install pygame_
```

Figure 1.2: line for downloading the Pygame

2) Problem Definition and Algorithms

The problem is determining the best algorithm that is able to choose the best moves to achieve longest snake length, In this section, Breadth First Search Algorithm and Q function by using a Neural Network Algorithm are explained to determine best possible move for snake's head to the apple which will be base knowledge for future Implementation part.

2.1) Breadth First Search_[1]

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer
2. Move to the next layer

Consider the following diagram.

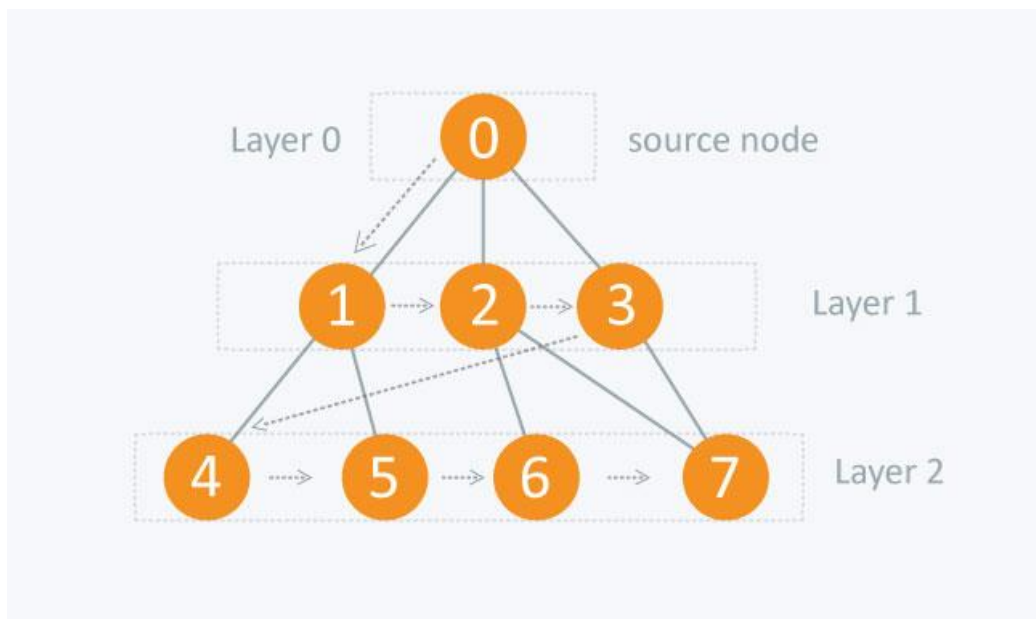


Figure 2.1: Diagram for BFS

The distance between the nodes in layer 1 is comparatively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

```
BFS (G, s)                                //where G is the graph and s is the source node
    let Q be queue.
    Q.enqueue( s ) //Inserting s in queue until all its neighbour vertices are marked.

    mark s as visited.
    while ( Q is not empty)
        //Removing that vertex from queue,whose neighbour will be visited now
        v = Q.dequeue( )

        //processing all the neighbours of v
        for all neighbours w of v in Graph G
            if w is not visited
                Q.enqueue( w )                //Stores w in Q to further visit its neighbour
                mark w as visited.
```

Figure 2.2: Pseudocode of BFS

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.



Figure 2.3: Traversing Process

2.2) Neural Network ^[2]

Neural networks are the workhorses of deep learning. They are trying to accomplish the same thing as any other model — to make good predictions.

Neural networks are multi-layer networks of neurons (the blue and magenta nodes in the chart below) that we use to classify things, make predictions, etc. Below is the diagram of a simple neural network with five inputs, 5 outputs, and two hidden layers of neurons.

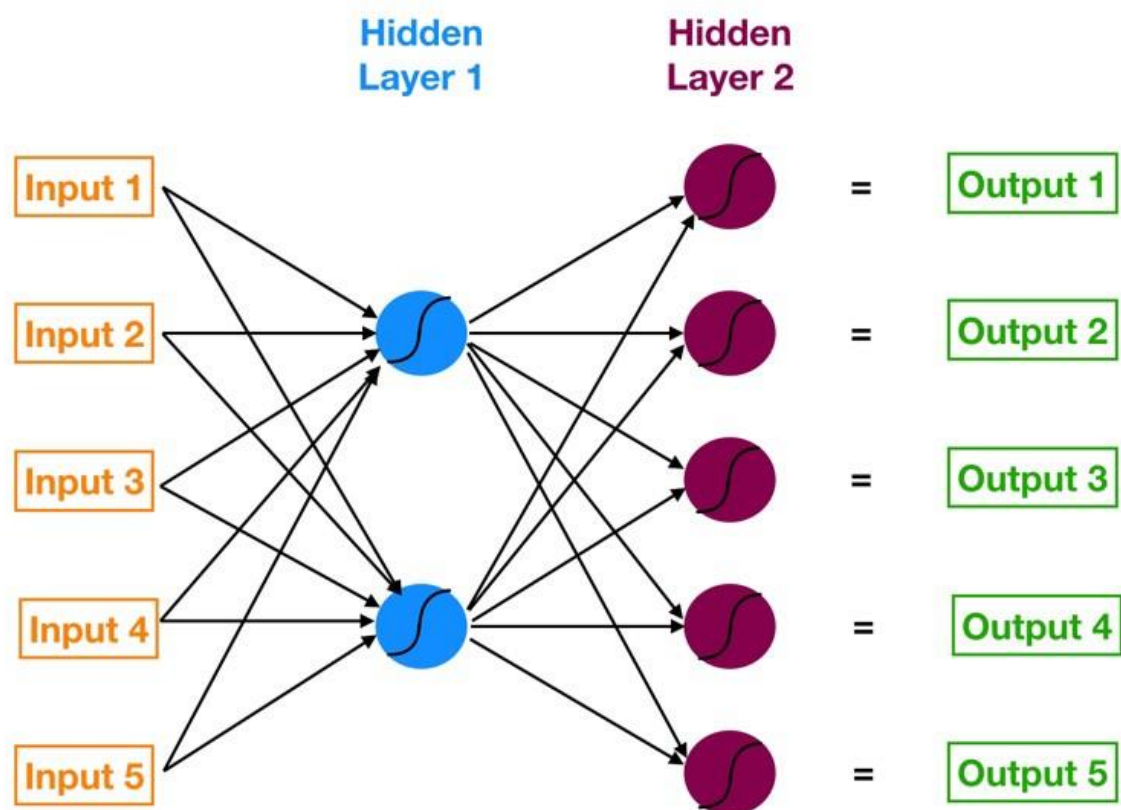


Figure 2.4: Neural Network with two hidden layers

Starting from the left, we have:

1. The input layer of our model in orange.
2. Our first hidden layer of neurons in blue.
3. Our second hidden layer of neurons in magenta.
4. The output layer (a.k.a. the prediction) of our model in green.

The arrows that connect the dots shows how all the neurons are interconnected and how data travels from the input layer all the way through to the output layer. What exactly is a neural network trying to do? Like any other model, it's trying to make a good prediction. We have a set of inputs and a set of target values — and we are trying to get predictions that match those target values as closely as possible.



Figure 2.5: Logistic regression (with only one feature) implemented via a neural network

Logistic regression (with only one feature) implemented via a neural network

This is a single feature logistic regression expressed through a neural network. To see how they connect we can rewrite the logistic regression equation using our neural network color codes.

$$\text{Sigmoid}(B_1 * X + B_0) = \text{Predicted Probability}$$

Figure 2.5: Logistic regression equation

Logistic regression equation

X (in orange) is our input, the lone feature that we give to our model in order to calculate a prediction.

B1 (in turquoise, a.k.a. blue-green) is the estimated slope parameter of our logistic regression — B1 tells us by how much the Log_Odds change as X changes. Notice that B1 lives on the turquoise line, which connects the input X to the blue neuron in Hidden Layer 1.

B0 (in blue) is the bias — very similar to the intercept term from regression. The key difference is that in neural networks, every neuron has its own bias term (while in regression, the model has a singular intercept term).

The blue neuron also includes a sigmoid activation function (denoted by the curved line inside the blue circle).

And finally we get our predicted probability by applying the sigmoid function to the quantity ($B1 \cdot X + B0$).

A super simple neural network consists of just the following components:

A connection (though in practice, there will generally be multiple connections, each with its own weight, going into a particular neuron), with a weight “living inside it”, that transforms your input (using $B1$) and gives it to the neuron.

A neuron that includes a bias term ($B0$) and an activation function (sigmoid in our case).

And these two objects are the fundamental building blocks of the neural network. More complex neural networks are just models with more hidden layers and that means more neurons and more connections between neurons. And this more complex web of connections (and weights and biases) is what allows the neural network to “learn” the complicated relationships hidden in our data.

3) Result and Comparison

3.1) Success rates and Statistics

Experiments are carried out by running computer versus computer implementation at Snake Game program with various different conditions. 15 game is played for each conditions in different screen size for both of the algorithms and different number of training size for Neural Network. The details of the snake length are tabulated which is shown below. Initial snake size is 1×1 and the apple's size is always 1×1

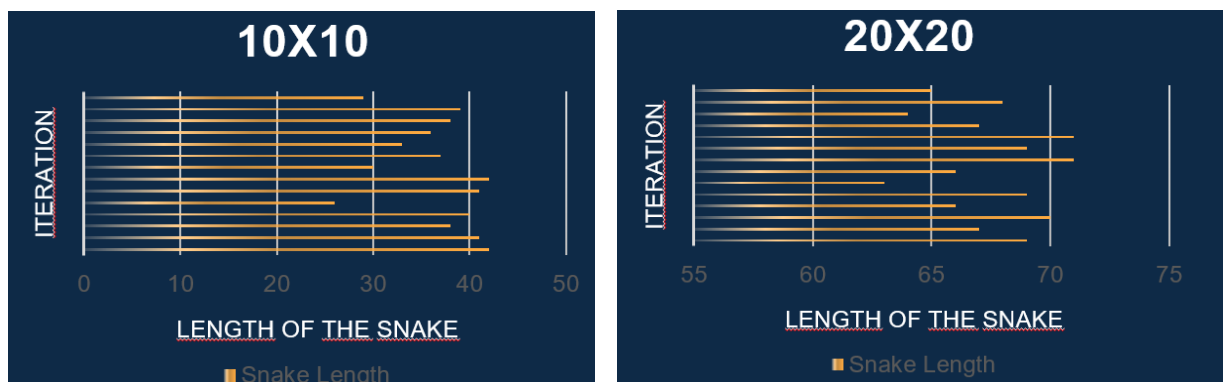


Figure 3.1: BFS average snake length for different screen size

When we look at the tables that shown above, we can see that snake lengths are about 35 and 67 which is pretty good score for a game that. Because of the characteristics of the Breadth First Search, AI most of the time makes a great decisions for the 10x10 and the 20x20 screen size, score still great for a basic approach like that but there is a flaw in that. Breadth First Search have to look every states to see which one is better, the speed of the decision is really slow when we compare with the 20x20 to 10x10 because when the new apple is more likely spawns very far from the snake this makes the decision pretty late when we compare with the 10x10.

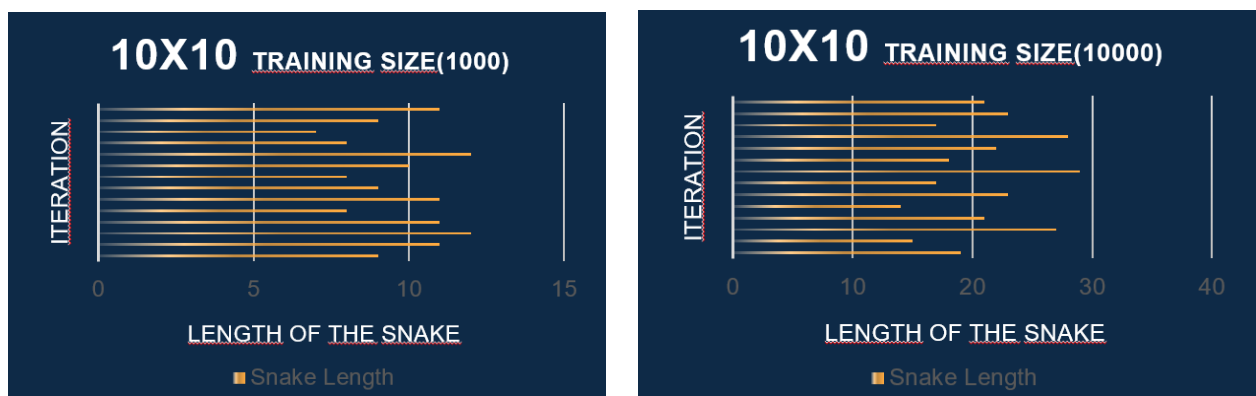


Figure 3.2: NN average snake length for different training size

When we look at the Neural Network approach for the 10x10 screen size, average length of the snake is about 10 and 20, we also see that average length is very low when we compare with the Breadth First Search, because the training part only take 1 or 10 minutes which is really small time for a training part. If we increase the training part we get pretty good result for this small partion of time.

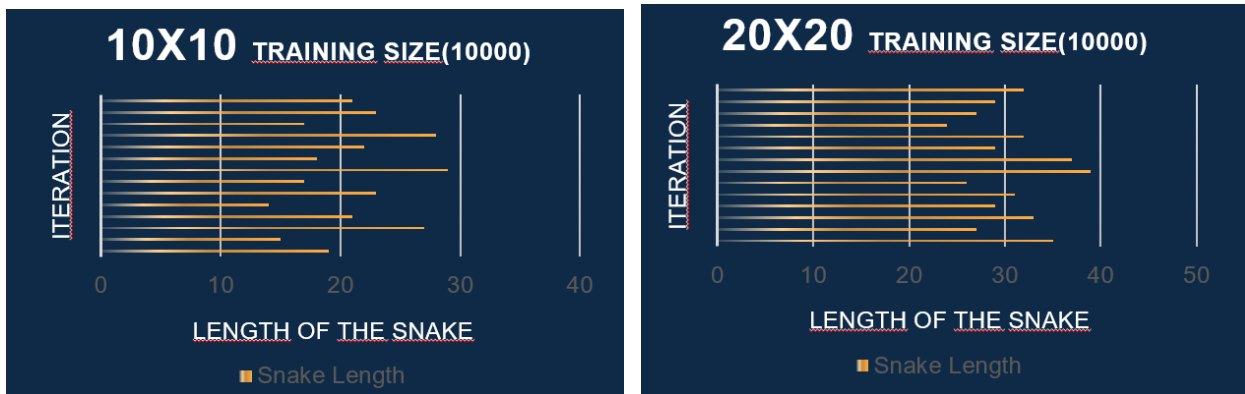


Figure 3.3: NN average snake length for different screen size

This time we will compare the length of the snake for same training size but different screen size. When we look at the results 10x10's average length is about 20 while 20x20's average length is about 30. We can see a good increase for the snake length for about %50, but when it comes to the training time, 20x20's training time took 4 times more than 10x10's training which is about 40 minutes which is relatively high for same training size. But this is because of the there is too much space that snake's head should look at to decide to where to go.

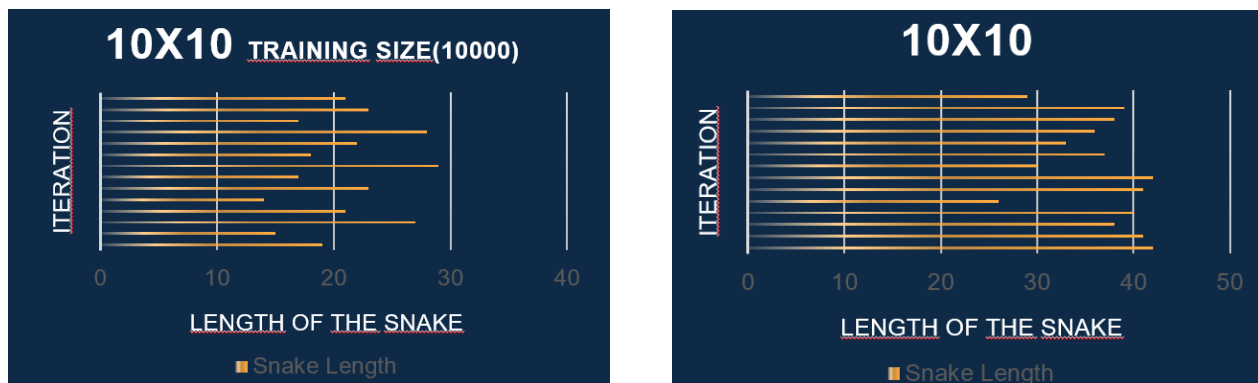


Figure 3.3: Same screen size, different approach

This time we are comparing the same screen size for 2 different approach. In my opinion this is more critical then the other comparisons because this is quite good comparison for 2 different approaches. When we look at the results, Breadth First Search looks like more successful for small experiment but when it comes to the big experiments like which would took 1 day or more to train the Neural Network, Breadth First Search have no chance to beat the Neural Network. In my opinion the biggest problem for the Breadth First Search is BFS very static and always stuck at some point but this is not same for the

Neural Network, because when we look at the tables for Neural Network we always see that Neural Network increases but this is not valid for Breadth First Search because of its own structure. But we can't put aside that Breadth First Search is much more easy to implement. It is just like double side of the sword, pros and cons.

3.2) Performance of Computation / Memory

In this section, experiments made between Breadth First Search & Neural Network for different screen size, Breadth First Search & Neural Network for same screen size, Neural Network & Neural Network for different screen size are compared with their execution time and used memory.

In the first experiment Performance of Computation is very valid for a simple and common method like this. Both of two had great results for snake length which were 35 vs 67 but when it comes to the Usage of the Memory, the BFS with bigger screen size was terrible because there was 4 times more space to look to decide which was about 10 seconds to 40 seconds to complete one game. Because of the BFS implementation structure still it is good results.

In the second Experiment for Performance of Computation is quite different from each other. Neural Network which is trained for 10000 iteration had reached 21 snake length on the other hand BFS had 35 snake length which is much better. Neural Network created a file about 43MB to train itself on the other hand Breadth First Search not needed something like that. Breadth First Search finished in 10 seconds while Neural Network finished in 2 seconds which is quite good domination for Neural Network against BFS.

In the last Experiment for Performance of Computation, Both of the Neural Networks had a good results. Neural Network with 1000 iteration had reached 12 snake length while other Neural Network with 10000 iteration had reached 25 snake length. Neural Network with 10000 iteration created a file about 43MB to train itself in 10 minutes on the other one created a file about 4 MB to train itself in 1 minutes.

3.3) Difficulties

Breadth First Search algorithm is a method that commonly used for many applications. And also Breadth First Search algorithm is one of the most popular algorithm for games. The main reason why it is that popular that is because it is pretty simple to understand its approach and implement it. Therefore, after the implementation, it can be a quite effective AI for small size experiments.

I think Neural Network is more advanced and more complicated than Breadth First Search algorithm in point of understanding and implementing it. And because I had not a knowledge for Neural Networks at first , I had the start from the bottom but it was not same for the Breadth First Search because I had a couple experience with the Breadth First Search before. I thought It should not be so difficult to understand Neural Network. However, It was challenging for me to get the approach that I am completely new.

5) Implementation

4.1) BFS

```
def BreathFirstSearch(size_x, size_y, b_size, visited, apple, start):
    choices = queue.Queue(maxsize=_0)
    choices.put(start)
    prtNode = {}
    while choices.empty() is False:
        look = choices.get()
        if look == apple: break
        elif look in visited: pass
        else:
            newCh = []
            newCh.append([look[0] + b_size, look[1]])
            newCh.append([look[0] - b_size, look[1]])
            newCh.append([look[0], look[1] + b_size])
            newCh.append([look[0], look[1] - b_size])
            visited.append(look)
            for i in newCh:
                if i[0] < 0 or i[1] < 0 or i[0] >= size_x or i[1] >= size_y or i in visited: pass
                else:
                    prtNode[str(i)] = look
                    choices.put(i)
    final_way = getWay(prtNode, start, apple)
    return final_way
```

4.1 Figure : Main Breath First Search Function

Function shown in figure 4.1 is the main function for Breath First Search algorithm. This function is for choosing the best move for the snake AI. The BFS method to get the way from the starting point of the snake to the apples position. It is look the nodes are visited or not. If it is the apple then we break the look , If it is not visited then looking the other nodes with the order or first in first out(FIFO).

```
def getWay(prtNode, start, apple):
    way = [apple]
    while way[-1] != start:
        way.append(prtNode[str(way[-1])])
    way.reverse()
    return way
```

4.2 Figure : Main Breath First Search Function

Based on the parent list to generate the path from starting point of the snake to the end point which is the position of the apple. It takes 3 parameters: prtNode: parent node, start: starting point of the snake, apple: position of the apple.

4.2) Neural Network

```
f=open("train_data.txt", "rb")
train_data = pickle.load(f)
X = np.array([i[0] for i in train_data]).reshape(-1, 5)
Y = np.array([i[1] for i in train_data]).reshape(-1, 1)

model = Sequential()
model.add(Dense(25, input_dim=5, activation='relu'))
model.add(Dense(1, activation='linear'))
model.compile(loss='mean_squared_error', optimizer='adam')
#model = load_model('my_model.h5')
model.fit(X, Y, epochs=20, batch_size=128, verbose=2)
model.save('my_model.h5')
```

4.3 Figure : Main Breath First Search Function

This is the Neural Network class that I implemented. I created sequential model object, named object. I added model which has 5 input parameters, 25 neuron in the first hidden layer with and activation Rectified Linear Unit. For compile part, I chose Adaptive Moment Estimation for optimizer because it is very popular nowadays. While I was trying to learn keras I wanted to learn the newest and the popular parts. To train the data we needed fit method. In this method batch size as 128 which is example number to help me to train the data. For the epochs I chose 20 which is basically it ensures that the dataset passes all of the artificial neural networks.


```

def trainSnake(times = 40000):
    train_data = []
    for i in range(times):
        print(i)
        pygame.event.pump()
        failed = False
        lead_x = 70
        lead_y = 70
        snake = NeuralNetwork_Snake(showScreen, screenx, screeny, snakeImg, lead_x, lead_y)
        apple = Apple(showScreen, screenx, screeny, b_size, appleImg, snake.snake_list)
        apple_x, apple_y = apple.apple_pos()
        act = "up"
        state = snake.state([apple_x, apple_y], act)
        former_distance = snake.distance([apple_x, apple_y])
        while not failed:
            apple_x, apple_y = apple.apple_pos()
            snake.update_snake_list(apple_x, apple_y)
            distance = snake.distance([apple_x, apple_y])
            score = 0
            if snake.is_alive() is False:
                failed = True
                score = -1
            showScreen.fill(white)
            if snake.eaten is True: apple.update_apple_pos(snake.snake_list)
            if snake.eaten is True or distance < former_distance: score = 1
            train_data.append([np.array(state), score])
            act = random.choice(moves)
            apple_x, apple_y = apple.apple_pos()
            former_distance = snake.distance([apple_x, apple_y])
            state = snake.state([apple_x, apple_y], act)
            snake.set_direction(act)

```

4.5 Figure : trainSnake Function

Function shown in 4.6 is the main function for training snake via Neural Network. This function basically stores the training data first, then use this data to train Neural Network. While doing that also print the game number that currently training. Based on the direction, we can work out x, y changes to update the snake, after the snake moves we get new distance to the apple. As a default, score is zero. If snake eats the apple, it will create a new random apple. If snake eats the apple or moves closer to the apple score will be 1. Last 6 line is using random method to move the snake to the apple.

```

def test(times = 10):
    model = load_model('my_model.h5')
    s = []
    for i in range(times):
        print(i)
        pygame.event.pump()
        failed = False
        lead_x = 70
        lead_y = 70
        snake = NeuralNetwork_Snake(showScreen, screenx, screeny, snakeImg, lead_x, lead_y)
        apple = Apple(showScreen, screenx, screeny, b_size, appleImg, snake.snake_list)
        while not failed:
            apple_x, apple_y = apple.apple_pos()
            snake.update_snake_list(apple_x, apple_y)
            if snake.is_alive() is False:
                failed = True
                s.append(snake.snake_length)
            showScreen.fill(white)
            if snake.eaten is True:
                apple.update_apple_pos(snake.snake_list)
            apple_x, apple_y = apple.apple_pos()
            allPredictions = {}
            for act in moves:
                state = snake.state([apple_x, apple_y], act)
                allPredictions[act] = model.predict(np.array(state).reshape(-1, 5))[0][0]
            act = max(allPredictions, key=allPredictions.get)
            snake.set_direction(act)

```

4.5 Figure : Test Function

Function shown in 4.6 is the function for using the Neural Network to play the snake game. Firstly we should load the module that we had during the training snake process. S is the list to store the length of the each game. Snake's default direction is right. Basically the function uses Neural Network model to get action max Q.

5) Conclusion and Future Works

In this project, Breadth First Search and Q function by using a Neural Network Algorithms are implemented into Snake Game and they are compared over their snake's length. Based on the analysis results done on the algorithms, the following conclusions are found: a. While the snake length extremely different for the conditions, performance increases when the training size increased for the Neural Network while Breadth Search Algorithm reach pretty much the same length. b. Based on experiments it is observed that the snake length increased and the speed of the snake is decreased as the screen size increased for Breadth First Search. c. Games played Breadth First Search and Q function by using a Neural Network Algorithms with same screen size, Breadth First Search showed better results for small experiments. The length achieved by BFS, was around 30 while Neural Networks snake length for 1000 training size was around 12. d. The execution time of BFS on average is much faster than Neural Network algorithm with 1000 training size. So, as a conclusion,

It is better to prefer to use Neural Network for large experiments.

Results obtained by the experiments shows that Breadth First Search is more static while we compare with the different conditions.

If we are looking for future implementations, our choice definitely would be Neural Networks.

To optimize the Artificial Intelligence more in Snake Game, Deep Q Network strategy can be implemented. This strategy indicates that is better for that can deal with huge screen size which makes the training fast.

Bibliography

[1] <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/>

<https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/>

[2] <https://towardsdatascience.com/understanding-neural-networks-19020b758230>

<https://pathmind.com/wiki/neural-network>

<https://medium.com/turkce/keras-ile-derin-ogrenmeye-giris-40e13c249ea8>

<https://pathmind.com/wiki/neural-network>