Eötvös Loránd University

Faculty of Informatics

Adaptability of neural networks between games of different genres

Kitlei Róbert

Assistant lecturer

Andrey Khasanov

Software Information Technology MSc

Department of Programming Languages and Compilers

Budapest, 2018

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

THESIS TOPIC DECLARATION

Student:

Name: **Andrey Khasanov** Code: **ANKTAAI.ELTE / VXDW14**

Type: **full-time student** Course: **Software Information Technology BSc**

Supervisor:

Name: Róbert Kitlei

Affiliation with address: ELTE Faculty of Informatics
Department of Programming Languages and Compilers
1117 Budapest, Pázmány Péter sétány 1/C.

Status and qualification: Assistant Lecturer, MSc in Computer Science

Title of the Thesis work:

Implementation of a platform game

Topic of the Thesis work:

Developing an implementation of the classic game "Prince of Persia" in Java using libgdx library to create executables for Android, Windows and other operating systems.

Contents

1. Introduction

1.1. Abstract

The goal of this thesis is to train a neural network for one task and then apply the trained neural network on the other task. This will be achieved by expressing the second task in terms of the first.

In the first part of the work, several artificial neural networks will be created and trained for the Snake game and the networks' properties, such as weights and biases, will be compared to their results. Some of the networks will be trained only briefly, while others will have an extensive training.

In the second part, the best performing networks will play the Maze game and the results will be compared to two other networks, one of which was exclusively trained for the Maze game and the other was trained for the Snake game first and for the Maze game second.

Some networks will have different structures, such as different number of layers and different number of neurons in them, but there will also be similarly structured networks, which will be trained with a varying number of test games to observe if more data and training bring the best results.

1.2. Snake game history

The Snake game is the classic video game concept where the player plays for a line that represents the snake which grows in length after consuming other points. The game's main difficulty is staying alive without touching the walls or itself. This concept was originally seen in the arcade maze game Blockade developed by Gremlin and published by Sega in October 1976. [1]

There are many variations of The Snake Game, both single player and multiplayer, with the most known being Nibbler, Tron, Slither.io and Nokia Snake.

Soon after its creation in 1978 Snake, which was programmed by Peter Trefonas for TRS-80 microcomputer and Apple II, was instantly widespread on arcade machines, personal computers and other platforms due to its ease of implementation and replayability and in 1996 Next Generation magazine ranked it number 41 on their "Top Hundred Games of All Time." [2]

In 1998 Nokia resurged the Snake popularity after preloading it as an app on their monochrome mobile phones. This helped the game tremendously as it was open to a much larger audience than before and people could play the game to spend their time in queues and bus stops. Snake was associated with a Nokia app ever since.

1.3. Artificial neural network history

The original idea of a neural network was described in 1943 by neuropsychologist Warren McCulloch and mathematician Walter Pitts in their "A logical calculus of the ideas immanent in nervous activity", which was on how neurons in the brain might work. The writers have modeled a simple neural network using electrical circuits. [3]

Later in 1949, Donald Hebb introduced his theory, also known as "Hebb's Postulate", about the neural bases of learning in his book "Organization of Behavior", which has indicated that neural connections are enhanced every time they are being used and if two nerves fire at the same time, the connection between them becomes stronger. [4] Thus, learning is not done by brain passively, which would be an incredible feat, but rather it is a process during which the cellular structure of the brain is altered permanently. This theory has a classic status within science and is backed by recent research. [5]

In 1959, Bernard Widrow and Marcian Hoff invented the Widrow-Hoff least mean squares filter (LMS) adaptive algorithm that was the basis for the ADALINE and MADALINE (Multiple ADAptive LINear Elements) networks. MADALINE was the first neural network to ever be applied to a real-world problem. It is an adaptive filter that eliminates the echoes on phone lines, which is still in commercial use. [6]

The success of ADALINE and MADALINE networks unfortunately did more harm than good to neural network research, particularly due to high expectations and outrageous promises on the network capabilities and severe limitations of hardware back then. Unfulfilled promises and increasing popularity of von Neumann architecture led to neural network research funding being heavily reduced.

There were a few advances in the field, however it was not until 1982 when John Hopfield presented "Neural networks and physical systems with emergent collective computational abilities" paper to the National Academy of Sciences. In this work he proposed using bidirectional connections between neurons, since previously the neural networks have only used single "forward" direction. [7] This has led to an invention of an associative neural network, also known as Hopfield network, which consists of long-term and short-term memories and the new training is done using the short-term memory.

In the same year, Reilly, Cooper and Elbaum have published "A neural model for category learning" that used the "hybrid" multiple layer neural network and in 1986 Rumelhart and McClelland described the use of parallel distributed processing (connectionism) in the neural network. [8] This is now known as a backpropagation network and, unlike hybrid networks, it uses more than a couple of layers, however that makes its training much slower, as it needs many more iterations over the input to produce results.

Nowadays, networks with multiple hidden layers, also known as deep neural networks (DNN), have achieved state-of-the-art performance on computer vision problems and the current goal is

to reduce both the model size and computational cost of the networks to allow a more widespread deployment. [9]

1.4. Neural networks in games

Neural networks are quite a rare occurrence in games as the latter are usually fully scripted and use 30-year old artificial intelligence (AI) technology, such as A* and finite state machines instead. [10] There are exceptions to this, of course, such as the 1996 game Creatures, where the player assumes control over the small furry animals and teaches them how to behave. These creatures are using the neural networks to learn. Another example would be the 2013 Forza Motorsport 5 by Microsoft where the neural networks learn how the human players control the in-game vehicles and react to events, such as crashes or overtakes by other players and then the trained networks, also known as drivatars, play versus human players for more realistic competition. [11]

On the other hand, the neural networks are quite commonly used for playing the games instead. One of the most famous examples of this is AlphaGo, a Google DeepMind's DNN which in the span of 2016-2017 won official matches against two world champions of Go game (ranked 1st and 4th at the time) to become the first Go program, not only to surpass the amateur level, but also to reach the professional nine dan level in the game. Even though nine dan is the highest possible level in Go, many top Go players believe that AlphaGo is above that. [12]

1.5. Personal motivation

One of the limitations of neural networks is the fact that the networks are problem-specific, and a trained network cannot be applied to another task after it learned to solve one. [13,14]

In this work, however, I aim to create and train a neural network that can play a game of one genre and then apply it onto a game of another genre to show that it is possible to reuse the network, instead of developing a new one altogether.

My personal goal is to familiarize myself with the neural networks playing the games and later apply that knowledge in game development field to create an alternative to the current AI technology and make the techniques of scripting and AI cheating obsolete. This will also make the AI controlled game characters' actions be more human-like and give AI the ability to learn from its mistakes and adjust accordingly.

I hope that this will bring us one step closer to more realistic artificial intelligence, which in turn will provide a better gaming experience to the players.

2. Preparation

2.1. Abstract concepts

2.1.1. Neural network

So far there was a lot of history of neural networks in this work, but there was no explanation on what the neural network is.

A neural network is a massively parallel distributed processor made up of simple processing units, which has a natural propensity for storing experiential knowledge and making it available for us. It resembles the brain in two respects: the knowledge is acquired by the network from its environment through a learning process and then stored using interneuron connection strengths, known as synaptic weights. [15]

There are two types of neural networks: biological neural networks and artificial neural networks (ANN). The biological neural networks could be any group of connected biological nerve cells, known as neurons. An example of such a network is our brain.

An artificial neural network, on the other hand, is a mathematical structure designed to simulate biological networks. ANN is formed of three different types of layers: input layer, hidden layer and output layer. Each layer consists of one or more neurons, where each neuron represents a relatively simple function that takes an input and produces an output. The neurons of an input layer are passive, meaning they do not modify the data used for training, while the neurons of hidden and output layers are active instead. A visual representation of an arbitrary ANN with one hidden layer can be seen in Figure 1.

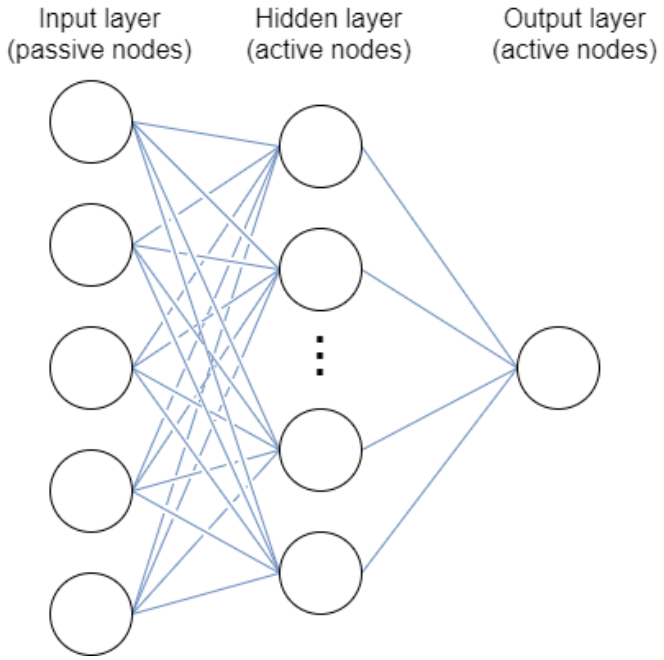

Figure 1. An arbitrary artificial neural network with one hidden layer

When the data is provided to an ANN for training, each value in an input layer is duplicated and each neuron in the hidden layers receives a copy. This is called a fully interconnected structure. After that, the values are multiplied by a set of predetermined numbers, also known as the weights, and then weighted inputs are added to produce a single number. This can be expressed as

$$y = f(x) = \sum_{i=1}^{n} x_i w_i + b$$

where x is the input, w is the weight and b is the bias, which is used to adjust the output. The number is passed through a non-linear function, also known as an activation function or a transfer function, which could be a sigmoid, rectifier, tan$h$ or any other activation function. This function takes a value between $-\infty$ and $+\infty$ and produces an output within the limits of the used function, which specifies if the neuron should fire or not. [16]

This work uses the state-of-the-art rectified linear unit (ReLU) as the activation function for a couple of reasons. First is the ease of calculation compared to other activation functions, since ReLU is a simple $f(x) = \max(0, x)$, while, for example, sigmoid uses expensive operations, such as exponentials: $f(x) = \frac{1}{1+e^{-x}}$. The other reason to use ReLU is because it returns the value of 0 more often than sigmoid or tan$h$, which makes the neural network sparser, since fewer neurons fire, which more realistically simulates a biological neural network and allows for the smaller number of calculations needed.

The training of an ANN can be supervised, unsupervised or a mixture of both. Unsupervised training needs a lot more data, but the data can be "unlabeled", which means it can lack categorization. This eliminates the need to label the data by hand, however, that also means that the network should have an effective algorithm to use the unlabeled data efficiently. The problem with this approach is that the accuracy of the said algorithm's output cannot be evaluated. However, this way of training is closer to the true artificial intelligence and can produce the results which humans do not take into consideration due to pre-existing biases.

Supervised learning is done using "labeled" data. The input is presented to the network as the pairs of data and desired outputs. The goal of this approach is to analyze the training data and produce an accurate function, which can be used both for expected and unexpected inputs to get reasonable outputs. In this work, the neural network training will be supervised.

2.1.2. Backpropagation

During the ANN training, there will be a discrepancy between the expected and the actual outputs. This discrepancy is an error, which will be used to update the neuron weights for a better result.

It is not possible to get around this issue and choose "perfect" neuron weights during the initialization step of the neural network due to how the weights are calculated. A common

practice for choosing the initial weights since 1986, when Rumelhart and McClelland published their classic “Parallel Distributed Processing”, is to generate random non-zero numbers, which are very small. [17] As the weights are randomly generated, they will not produce the best immediate results and will need to be adjusted. Here is when backpropagation comes in.

Backpropagation, also known as backward propagation of errors or a feedback step, is a method of calculating an error made by each neuron in a network after the data is processed, to then send the values to an optimization method, such as gradient descent, which will adjust the neuron’s weight accordingly. This process requires multiple iterations, also known as epochs, to reach its highest potential to reduce the difference between the expected and the actual output of the network. It is most commonly used in deep neural networks.

2.1.3. Optimization algorithms

As it was mentioned above in section 2.1.2, the optimization algorithm adjusts the ANN’s neuron weights to produce more accurate results. There are two types of such algorithms: first order optimization algorithms and second order optimization algorithms, where the order specifies the derivative order.

First order algorithms minimize the error using the gradient values with respect to the parameters, with an example being the gradient descent algorithm. The regular (batch) gradient descent will perform an update after calculating the gradient of the whole data, which makes it very slow and inefficient if the data is too large. There is another variant of gradient descent called stochastic gradient descent (SGD), which is more popular, as it calculates the gradient for each training input data. The SGD algorithms which are most commonly used are momentum, Nesterov accelerated gradient, AdaGrad, AdaDelta and Adam, with the last being used in this work.

The second order optimization algorithms depend on the second order derivatives and, thus, are not that popular now, due to high cost of computation in terms of time and memory. However, they have a potential to outperform the first order algorithms if their derivates are calculated in advance. Second order algorithms include Newton’s method, Quasi-Newton and Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm.

2.1.4. Adam optimization algorithm

Adam, short for adaptive moment estimation, is a method which computes the adaptive learning rates for different parameters. The learning rate is a measure of how fast the network arrives to the final solution, so if the learning rate is too high, the network might fluctuate around the solution without reaching it or miss it completely (diverge). If the learning rate is too low, however, the network will take a very long time to reach the final solution (converge), so the best practice is too find a good middle ground.

Adam was introduced in 2015 as a SGD method that only requires first-order gradients with little memory requirement and it was designed to combine the advantages of two other popular

optimization algorithms: AdaGrad, which works well with the sparse gradients an Root mean square propagation (RMSProp), which does well in realistic settings. [18]

At this moment Adam is a state-of-the-art algorithm that compares favorably to other SGD methods, as it can be seen on Figure 2.

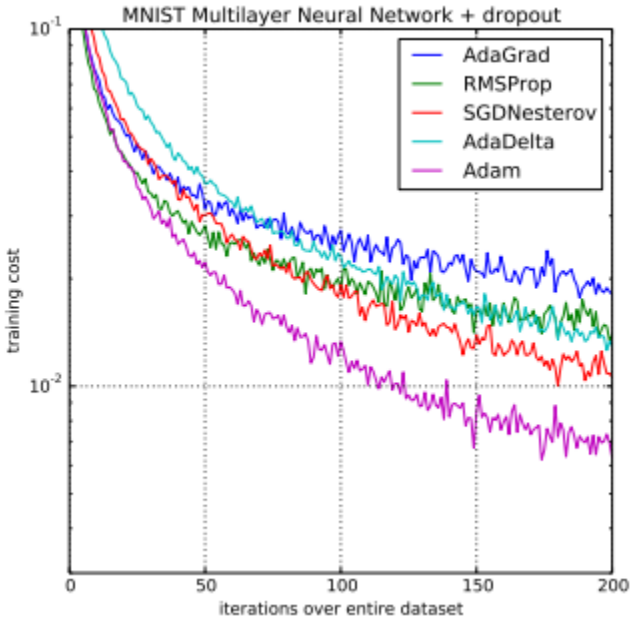

Figure 2. Comparison of Adam to other optimization algorithms training a multilayer neural network. Taken from Adam: A Method for Stochastic Optimization, 2015 *[18]*

2.2. Software

This work uses Python 3 programming language [19] and the publicly available libraries which can be found on "PyPI – the Python Package Index." [20]

2.2.1. TensorFlow

TensorFlow [21] is an open-source library for numerical computation using data flow graphs. The graph nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. TensorFlow is available on all major operating systems, including Android and iOS. It was originally developed by Google Brain team for the purposes of conducting machine learning and deep neural networks research internally, but was released to public in 2015.

2.2.2. TFLearn

TFLearn [22] is a modular and transparent deep learning library for distributed machine learning built on top of TensorFlow. It features a higher-level API for implementing deep neural networks, as well as helper functions to train the TensorFlow graphs and offers easy graph visualization. It was created by Aymeric Damien in 2016.

2.2.3. NumPy

NumPy [23] is the fundamental Python package for scientific computations which is used massively. It contains N-dimensional array objects, tools for integrating C/C++ and Fortran code, as well as the implementations of high-level mathematical functions. The ancestor of NumPy, Numeric, was created by Jim Hugunin, which was later competing with another package, Numarray. Numarray was faster while doing operations with large arrays and Numeric was faster with small ones. In 2005, Travis Oliphant created NumPy to unify the community around a single package which had the best of both worlds.

3. Applying a neural network to the Snake game

This part of research concerns the Snake game and the experiments will be conducted on how modifying the settings and parameters affects the performance of the neural networks. First, the networks with a varying number of initial test cases will be compared in the Section 3.4.1, and then their neuron weight values will be discussed in the Section 3.4.3. After that the neural networks with the same total number of neurons across the different number of layers will be compared in the Section 3.5 and it will be seen if the deep neural networks perform better than the shallow ones.

The performance will be measured by the scores the neural networks achieve in their 5,000 test games and, after that, the best performing networks will be tested on the Maze game to see if they can obtain similar results.

3.1. Snake game implementation

The Snake game source code was taken from the internet source [24], and extensively modified to allow extra features, such as manual play and command-line arguments. The playing field is 20x20 characters and the field borders are impassable walls. The score is shown at the top of the playground. The characters used to print the snake are ‘O’ for the head and ‘o’ for the body, while ‘x’ marks an apple location.

A screenshot of the game can be seen below.

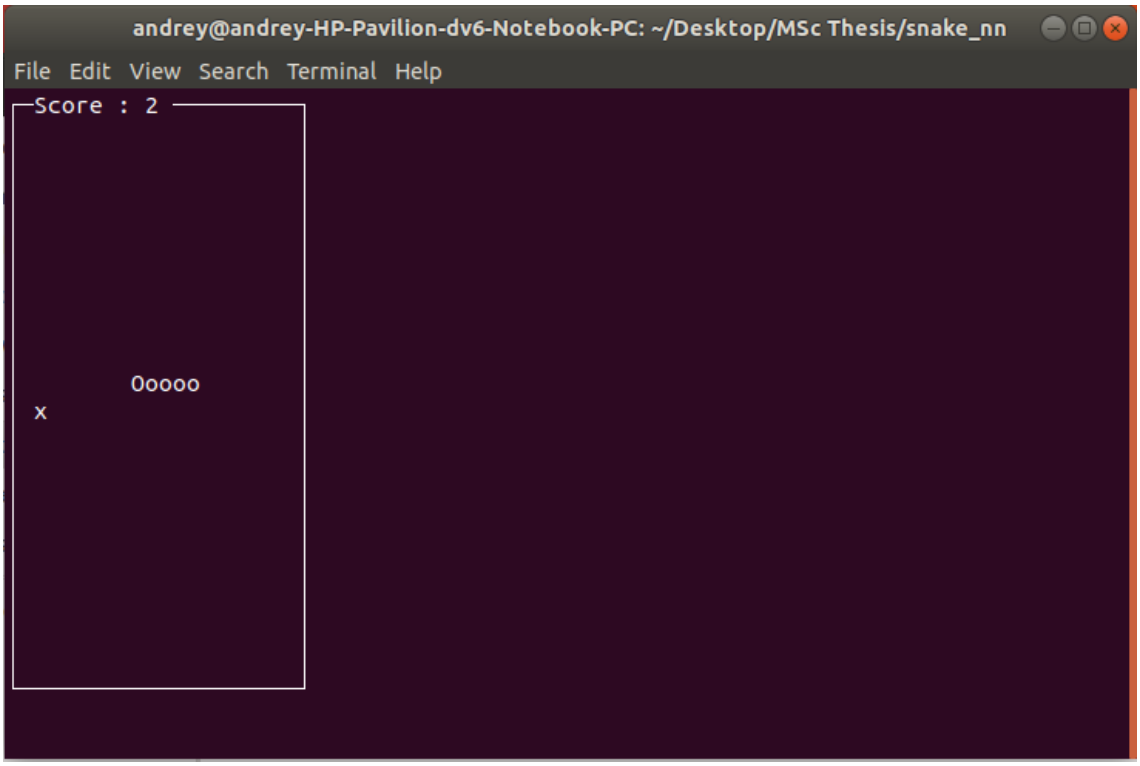

Figure 3. The Snake game written in Python using included Curses library

3.2. Neural network structure

3.2.1. Input layer

The input layer receives the data, which it later passes on to the following layers without any changes. This data will be a list of five values, with first three values specifying if there are obstacles immediately to the left, front and right of the snake. The values can be either one (1) or zero (0), where zero means that there is no obstacle in specified direction and one means that the obstacle is there.

The fourth value of an input is an angle between snake's current direction vector and direction to an apple. This angle will be normalized and divided by 180 degrees, so that the resulting value ranges from -1 to 1.

The last value will be the direction the snake is suggested to take, which can be one of the three predetermined values: -1 specifies a turn to the left, 0 is to continue moving in the current direction and 1 is a turn to the right. This is what makes the neural network training supervised, as the network is told what outputs are expected for certain inputs.

3.2.2. Hidden layer

The hidden layer is the first layer where the values are expected to be modified, however the results of this layer will be sent to another hidden layer or to an output layer for additional calculations and, thus, will remain hidden outside the network, hence the name.

Since the neural network created for playing the Snake game does not have to be overly complicated, there was a decision to only have one single hidden layer with 25 neurons in it, with ReLU being the activation function.

The other networks, however, will consist of multiple hidden layers with a varying number of neurons in each, where the number of neurons per layer will be specified.

3.2.3. Output layer

The output layer produces the final results on the neural network and in our case the return value will be one of the three choices: -1 if the snake did not survive a turn, 0 if the snake has survived, but the direction of the snake is wrong in relation to the apple and 1 if the snake has survived and chose the correct direction in its last move.

This layer contains only one neuron and uses the default linear activation function.

3.2.4. Code implementation

Described neural network is trivial to implement in Python using TFLearn package. The code is as follows:

```
network = input_data(shape=[None, 5, 1], name='input')
network = fully_connected(network, 25, activation='relu')
network = fully_connected(network, 1, activation='linear')
network = regression(network, optimizer='adam',
learning_rate=0.01, loss='mean_square')
model = tflearn.DNN(network)
```

3.3. Neural network training

The training of the neural network is a time and memory consuming process depending on the amount of games to train with. In our case, we will use different networks, which will be trained with a varying number of games and then the results of these networks will be compared after 5,000 test games are played by each.

The games are generated automatically, which means that each turn of the game will be chosen randomly out of three possible directions (left, front, and right), as going backwards is impossible, and if the snake is alive after the previous turn, new turn is generated until the game is over.

Since the computers used for generation of these initial sets of games were not designed to contain a big amount of data in the memory at the same time (there is more than 5,000,000 randomly generated moves for 100,000 games only), the training part of the network was designed to have an ability to load an existing network and continue training it with a new set of data. At first, however, this feature was absent, and the network was trained from scratch every time the training process was activated. This quickly became a problem due to the limitations of computer memory, as the network could not be trained with more than 100,000 games, due to a memory error. Therefore, the network with 1,000,000 initial games was trained during 10 different training sessions of 100,000 games each.

Time spent on training the networks, as well as their performance comparison over the course of 5,000 test games, is in the Table 1. The networks are single-layered and have 25 hidden layer neurons, except for two-layered and three-layered networks, which have 100 neurons per each hidden layer, and every network was trained with a different number of initial games. For the network with 1,000,000 training data the time was recorded over the course of 10 training sessions and calculated later. Time spent on generation of training data set is omitted, however it was the main reason of the training slowing down tremendously. It is also important to note that the game score is the number of apples picked up by the snake and not the final length of it, which would be equal to the score plus three (the size of the snake at the start of the game).

3.4. Network comparison based on an initial number of games

3.4.1. General results

Table 1. Trained neural networks performance comparison

Number of training games	1,000	10,000	100,000	1,000,000	100,000 (two-layered)	100,000 (three-layered)
Time spent on the 1st epoch (s)	4.401	36.105	362.135	3636.172	602.469	791.910
Time spent on the 2nd epoch (s)	3.402	35.220	378.288	3843.051	614.335	820.464
Time spent on the 3rd epoch (s)	4.144	36.470	376.790	3861.501	617.867	824.834
Total training time (s)	11.947	107.795	1117.213	11340.724	1834.671	2437.208
Average steps in 5,000 games	279.8218	247.0770	426.6786	400.9064	549.5052	602.5854
Average score in 5,000 games	19.3770	16.9704	28.7722	27.3626	35.5392	38.2772
Worst score	1	1	4	2	6	5
Best score	53	67	75	64	94	89

As it can be seen from the table, the number games to train from is an important distinction between the networks and their performances. It is expected that the less games there are, the worse the network performs, however, there are two interesting performance drops, with the first being from 1,000 games to 10,000 and the second from 100,000 games to 1,000,000. This could be attributed to the fact that the set of 5,000 games is a very small sample, however, the tests were run many times and the results were consistent across all runs.

The first drop in performance is probably because 10,000 initial games is too small of a data set and the network did not train enough to obtain good or at least better results than its predecessor. It could also mean that the training took a wrong turn during the process and the quality of the network went down, but then went up again once there was more data to train from in a 100,000 initial games network.

The reason of the second performance drop is a phenomenon known as the overtraining. Overtraining happens when there are too many iterations done on the training data and the network simply memorizes the input values and their expected results, which increases the network's performance over the training set of data, but worsens its ability to generalize on the new data. In other words, the network does well when it replays the game scenarios it has trained on, but has difficulties playing the game otherwise.

After studying the data for the single-layered networks, it was observed that the data set of 100,000 games produces the best trained neural network and the decision was made to train the multi-layered networks with the same amount of games in hopes to create the best possible deeper networks.

Out of all the studied networks, the neural network of three layers produced the best results, while the two-layered network was the second best, but it was interesting to see that even though three-layered network was better on average, it still could not beat any of the high scores of the two-layered network.

The performance rise of the deeper networks, however, is not surprising at all, because the general approach in neural network creation is to have multiple layers, with some state-of-the-art networks having over 150 of them! [25]

Since operations which are performed by the network's neurons are very simple, combining them allows for much more complex calculations to be made possible. This, in turn, allows for a higher level of abstraction in deeper layers, where every next layer will recognize more and more features of the data set. An example is the image recognition, where the first layer of the neural network will only recognize the very simple things like edges, second layer will be trained to recognize more difficult shapes, such as ellipses and circles, third layer will be able to differentiate between ears and eyes, while the fourth layer will already learn very complex features, like human faces or other objects.

After considering the success of the deeper neural networks, there was a natural decision to study the networks with a constant total number of neurons across the varying number of hidden layers later in the paper to see if the deepest network does, in fact, perform the best.

3.4.2. Game strategies

One of the interesting things to talk about is the different strategies that some of the neural networks had. As it was expected before the training, the networks with 1,000 and 10,000 initial set of games were using a diagonal turn strategy. An example of what that means can be seen in Figure 4 and Figure 5. This strategy was always used if an angle between the snake head and an apple was 45°. When the angle was different, then instead of moving in a perfect diagonal line, the networks were skipping a turn to get to 45° sooner, but they also never skipped more than one turn at a time.

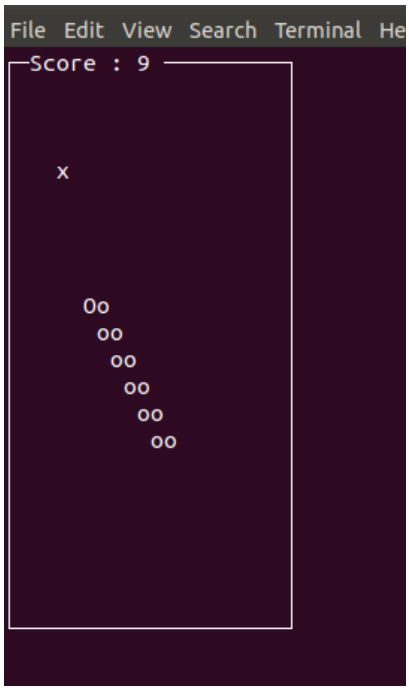

Figure 4. **Diagonal turns strategy**

Figure 5. **Diagonal turns with skips**

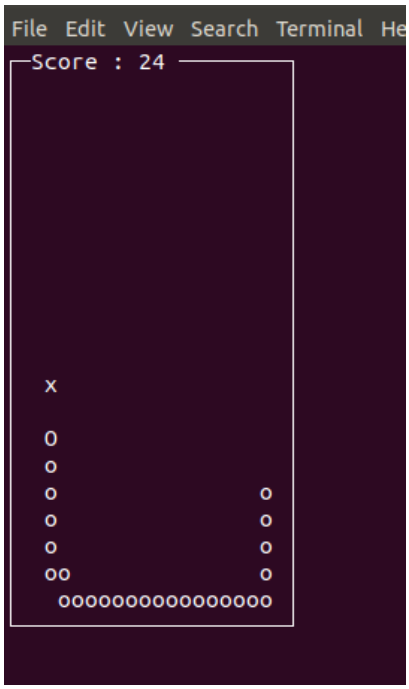

Figure 6. **Transition from diagonal turns to straight turns after more training**

As it can be seen in Figure 6, when the network was trained with more games, such as the 100,000 and 1,000,000 games networks, it was transitioning from diagonal turn strategy to the straight turns, however, some of the occasional moves it made resembled the original diagonal turn strategy.

The complete shift from the diagonal turns strategy in favor of the straight turns finally happened once the networks became deeper. Both two-layered and three-layered networks were using this strategy, which is visualized in Figure 7.

Several people were asked to play the Snake game to compare the strategies used by the neural networks and the humans, and every human player used the straight turns strategy right from the start as well, which the deeper networks have managed to discover by themselves.

The biggest difference between the way the humans played the game compared to the neural networks was that humans were able to tell if they were going to enter an area that was obstructed by their own body, which was not possible to get out from in a limited number of moves, such as the game state shown in Figure 8. In these situations, humans were able to take preliminary measures to avoid the problematic states of the game. Most players did it by going in the opposite direction to the apple first and then, when the hazard of entering an unwinnable position was gone, continuing playing the game normally.

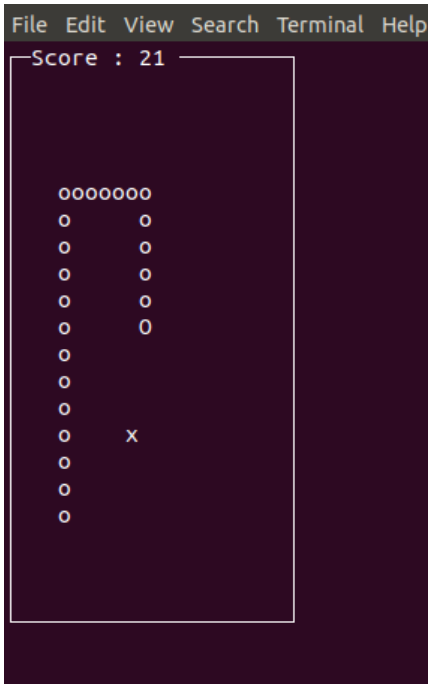

Figure 7. **Straight turns strategy in deeper networks**

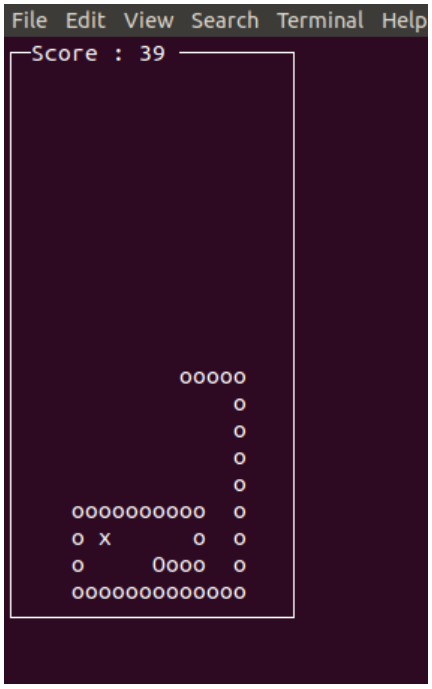

Figure 8. **Unwinnable state avoidable by humans**

This skill, however, did not help humans obtain higher scores than the neural networks, with their best scores being in the between 20 or 35 points only, while the networks were getting the results that were several times better consistently. But, on the other hand, if the networks knew how to avoid situations like this, it would give them an even bigger advantage over the human players, which would lose any ability to compete with the networks at any point of the game, like it already has happened with the Go game, as mentioned earlier in Chapter 1.4 on page 6.

3.4.3. Weights and biases

Every neuron has its own weight value for every input and since there are five input values in our networks, every neuron has five different weights, which, if we are taking our single-layered network of 25 neurons, is 125 values. Biases, however, are not different for every input value, thus making them to be a total of 25 values.

Considering that the structure of the multiple layered networks and the single layered ones is different (they have different number of neurons per layer, too), the best way to compare the weights and biases is to compare them across the networks of the similar structure. Another limitation is that 125 values is too many to compare, so only the values for one input variable will be compared across the networks. There is no importance in which variable is chosen, but let us choose the one that specifies if there is an obstacle directly to the left side of the snake, which is the second parameter of the input.

Table 2. Single-layered network weights for one of the input parameters

1,000 initial games	10,000 initial games	100,000 initial games	1,000,000 initial games
0.072406173	-0.047996681	-0.01248554	-0.01248554
-0.538972318	-0.847055733	-2.822388411	-2.618926287
-0.877597332	-1.344542503	0.273763716	0.843500078
-1.143232465	-2.159810543	-3.354897499	-3.400511503
0.042361051	-0.030023403	-0.006273229	-0.006273229
-0.769199014	-0.2972067	-0.2972067	-0.2972067
-0.939412951	-0.939412951	-0.939412951	-0.939412951
0.075344317	-0.021229422	-0.001764486	-0.001764486
0.252286732	0.384311885	0.530116677	0.505924523
-0.487262487	-1.483316064	0.331819355	0.331819355
-1.04176867	-2.099506855	-2.499260902	-2.499260902
-0.188621491	-0.001163005	0.128729463	-0.046580069
-0.094241835	0.080227323	0.411132187	0.258867204
0.046578322	-0.021678241	-0.170884013	-0.071125247
-1.098333478	-1.802851558	-1.802851558	-1.802851558
-0.003732676	-0.003732676	-0.003732676	-0.003732676
-1.042869568	0.008858151	0.008858151	0.008858151
-0.29927665	0.107646152	0.086139977	0.086139977
-0.13634105	0.007103892	0.160483778	0.151804492
0.009423577	0.009423577	0.009423577	0.009423577
-0.080396853	-0.247389868	-0.437185228	-0.419417709
-0.062357765	-0.062357765	-0.062357765	-0.062357765
-0.129455134	-0.000063382	-0.000063382	-0.000063382
-0.750602961	-0.919643402	-1.939302325	-1.992216945
-0.023972072	0.034327365	0.034327365	0.034327365

The weights for four single-layered networks can be seen in the Table 2. The weights that did not change throughout the additional trainings are highlighted. An interesting observation is that if the neuron weight value did not change for one of the input parameters, they did not change for other parameters and the bias values of those neurons remained unchanged, too. Also, another observation is that if the value remained the same from one network to the other, it remained constant throughout all the networks, as if the calculated value was “perfect”.

When the neural networks are being trained for prolonged amount of time, it is expected that at some point the differences between consequent networks weights start converging to zero, because the network comes closer and closer to its final state where additional trainings will not have much influence over the performance of the network. Since 1,000,000 initial games network was calculated across 10 different training sessions of 100,000 games each, it is possible to compare the weight value changes from one session to the other, as all the network states were recorded.

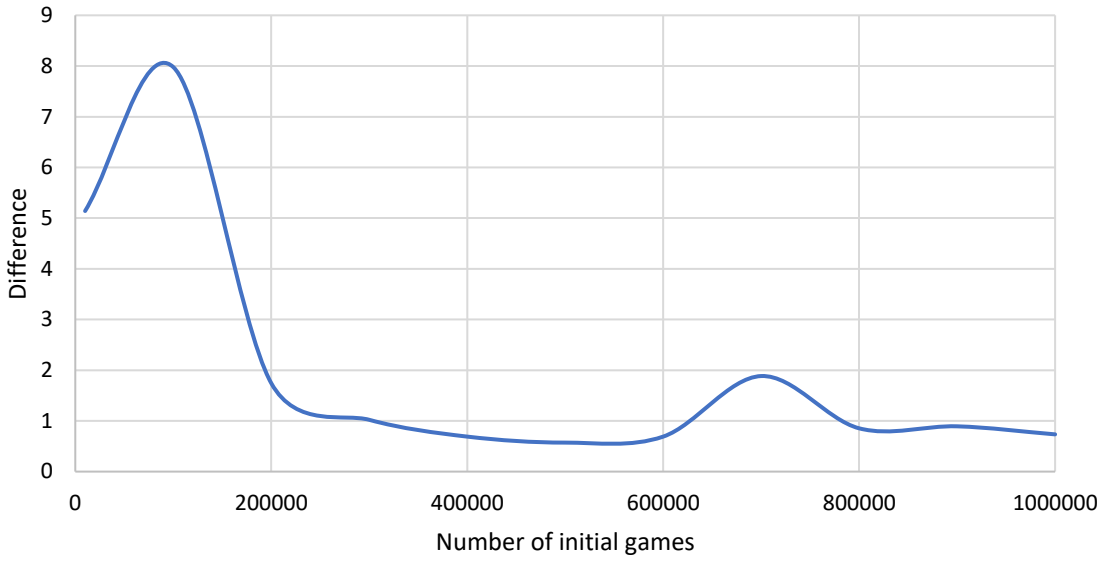

Figure 9. The weights differences converge to zero with every subsequent training

The differences between the neural network weights after N initial games can be seen in Figure 9. The values do start converging to zero after 200,000 set of games, as expected. There is an abnormal increase in the weights difference going from 600,000 to 700,000 initial games, which can be attributed to the network overtraining phenomenon, but after this irregularity the values continue converging to zero.

3.5. Network comparison based on layers

Previously in this work it was observed that the three-layered network produced the best results out of all the compared networks and there was an assumption that the deeper networks should produce the best results.

Since the number of layers will be different across all networks, there was a decision to keep the total number of neurons the same, split through the layers equally. The chosen number was 36, because it is divisible by most numbers from 1 to 9. Thus, the single-layered network has 36 neurons in its only layer, the two-layered network has two layers of 18 neurons each and so on. In case the neurons cannot be split across the layers equally, each layer contains the quotient of the total neuron number with the divisor being the number of layers. Thus, in the five-layered network, first four layers contain 7 neurons, while the last one contains 8.

Each network performance is in the Table 3, which is similarly structured to the previous comparison table, however, the training time was omitted, as it was equal to 320 seconds per epoch across all the neural networks, since they all contain the same total number of neurons overall.

The average scores of the networks are also present in the Figure 10 for an easier visual comparison.

Table 3. Trained multiple-layered neural networks comparison

Layers number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Average steps in 5,000 games	373.48	443.63	323.97	207.95	198.69	200.61	228.57	214.4	∞
Average score in 5,000 games	25.812	30.289	22.514	14.972	14.397	14.684	16.532	15.683	0
Worst score	4	2	2	1	1	1	2	1	0
Best score	63	67	67	44	43	48	47	45	0

As it can be seen both from the table and the chart, the single-layered and two-layered networks have a strong start, but the subsequent networks have a large performance drop compared to the first two. The last nine-layered network with four neurons per each layer was only able to learn one goal of the Snake game, which is staying alive, but it completely missed the other important objective of picking the fruits up to gain an increase both in the snake length and the score.

There is a small improvement shown by seven-layered network, but this improvement is so little it can be neglected due to a rather small test sample of 5,000 games.

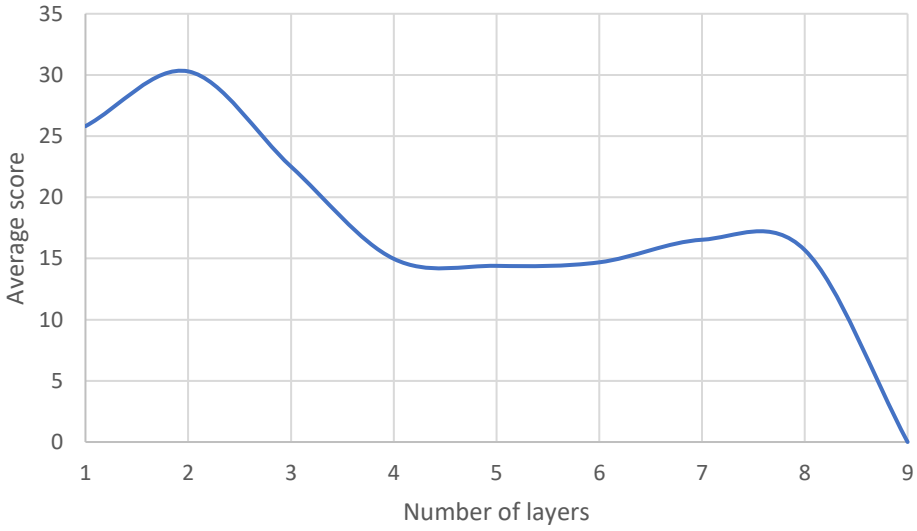

Figure 10. Average scores maintained by the networks with total same number of neurons

This shows that the previous assumption of the deeper networks being better than the shallow ones is only partially correct. Even though the two-layered network did perform better than the single-layered one, after that there was an inverse correlation between the number of layers and the scores that the networks managed to achieve on average. This observation, however, does not necessarily mean that the deeper networks are worse than the shallow ones, since we had the same total number of neurons in our networks, while in the real-life scenarios we would have additional neurons to work with.

Thus, the deeper neural networks success in the Section 3.4.1 can be attributed to the fact that the deeper the network was, the larger total number of neurons it contained.

4. Applying a pre-trained network to a different game

This part of research is about applying the pre-trained networks, which performed the best in the Snake game, onto a Maze game. The experiments will be conducted on how these pre-trained networks perform in the new game and in the Section 4.3 the results will be compared to the networks which were exclusively trained for the Maze game. After that the best performing networks from the previous chapter will be trained against the Maze game and the results will be compared again in the Section 4.4.

The training process will be the same as the one used in the previous chapter.

4.1. Maze game implementation

Just like in the Snake game, the playing field is 20x20 characters and the field borders, as well as the obstacles on the field, are impassable walls. The score is shown at the top of the playground as well. The character used to print the player is ‘o’, while an ‘x’ marks an exit location.

The maze itself is generated by the Kruskal algorithm and then the neighboring cells of the start and the end points of the maze are cleared of walls to have a higher chance of the maze being solvable. While there is still a small possibility of the maze not being solvable, there was a decision not to change that, as this makes it for a bigger challenge for the networks.

Figure 11. Maze game built on top of the Python Curses library

Originally, the score was incrementing by one after every successful turn that the player managed to survive, but this quickly turned out to be an abysmal design decision, as every trained network was moving in circles to obtain infinitely high scores, without considering the objective of reaching the exit.

Thus, there was a decision to give negative points once the network repeats the same turn more than three times and if that does not help the network change its strategy, the game is declared lost, once the same turn was repeated more than seven times. If the maze is solved, however, the final score of the game is set to 1, which is done to simplify the counting of the solved mazes.

To apply the trained neural networks from the previous chapter, the Maze game was expressed in terms of the Snake game. Thus, an apple became an exit location, while the obstacles on the field are the continuation of the snake body and the player is simply a snake head. This made it possible for the networks to apply their knowledge of the previous game to play the new one.

The screenshot of the Maze game can be seen in the Figure 11.

4.2. Neural network structure

The structure of the pre-trained neural networks is the same as before, as they were not changed for the new game. The newly trained networks are going to contain up to three hidden layers.

4.2.1. Input layer

The input layer is the same as it was in the previous chapter; there are five arguments passed to the network, where the first three specify if there are obstacles immediately next to the player, the fourth value is a normalized angle to the exit divided by 180 degrees and the fifth value is the direction that the player is suggested to take, which makes the neural networks training in this chapter supervised, too.

4.2.2. Hidden layer

The number of neurons per each hidden layer will be specified for every neural network in this chapter. ReLU will remain the activation function that all the neurons use.

4.2.3. Output layer

The output layer contains only one neuron returns one of the three values, like before: -1 if the player did not survive a turn, 0 if the player has survived, but the direction of the player is wrong in relation to the exit and 1 if the player has both survived and chosen the correct direction in their last move. Linear activation function is still the one used by the hidden layer neuron.

4.3. Networks comparison

The pre-trained networks that were chosen to be tested with the Maze game are the single-layered network of 25 neurons that was trained over the set of 100,000 initial games from the Section 3.4, the two- and three-layered networks of 100 neurons per layer from the Section 3.4, and the two-layered network of 18 neurons per layer from the Section 3.5.

The networks that were exclusively trained with the Maze game were chosen to be of the same structure as the pre-trained networks: the single-layered network of 25 neurons and two- and three-layered networks of 100 neurons per layer. It was decided not to implement the two-layered network of 18 neurons per layer, as there was already a two-layered network with a larger number of neurons.

Every new network was trained with a set of 100,000 initial games and the results that were achieved by both pre-trained and new networks are in the Table 4. Since it is possible to lose with the score of 0, if no turns were repeated more than three times and it is also possible to have a negative score throughout the game and still solve the maze in the end, which sets the final score to 1, the average score statistic is omitted from the table, as it is rather meaningless.

4.3.1. General comparison

Number of layers	1[1]	2 (100 neurons per layer)[1]	3[1]
Average steps in 10,000 games	23.7222		
Total mazes solved	400		

[1] Pre-trained network

5. Bibliography

[1] Gerard Goggin, *Global Mobile Media*. London: Taylor & Francis, 2010.

[2] Imagine Media, "Top Hundred Games of All Time," *Next Generation*, vol. 2, no. 21, pp. 55-56, 1996.

[3] Warren S McCulloch and Walter Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115-133, 1943.

[4] Donald Hebb, *The Organization of Behavior*. New York: Wiley & Sons, 1949.

[5] Richard Webster, *Why Freud Was Wrong: Sin, Science and Psychoanalysis*. Oxford: The Orwell Press, 2005.

[6] Prashan Premaratne, *Human Computer Interaction Using Hand Gestures*. Singapore: Springer Science & Business Media, 2014.

[7] John Joseph Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 79, no. 8, pp. 2554-2558, April 1982.

[8] David Everett Rumelhart and James McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*. Cambridge: MIT Press, 1986.

[9] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos, "Deep Learning with Low Precision by Half-wave Gaussian Quantization," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Honolulu, 2017, pp. 5406-5414.

[10] Georgios N Yannakakis, "Game AI Revisited," in *Proceedings of the 9th conference on Computing Frontiers*, New York, 2012, pp. 285–292.

[11] Tsaipei Wang and Keng-Te Liaw, "Driving style imitation in simulated car racing using style evaluators and multi-objective evolution of a fuzzy logic controller," in *IEEE Conference on Norbert Wiener in the 21st Century*, Boston, 2014, pp. 1-7.

[12] Fei-Yue Wang et al., "Where does AlphaGo go: from church-turing thesis to AlphaGo thesis and beyond," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 2, pp. 113-120, 2016. [Online]. http://www.ieee-jas.org/EN/abstract/article_145.shtml

[13] Albert Nigrin, *Neural Networks for Pattern Recognition*. Cambridge: MIT Press, 1993.

[14] Konstantin Bournayev, "Neural-Network Based Physical Fields Modeling Techniques," in *Computer Science -- Theory and Applications*, vol. 3967, St. Petersburg, 2006, pp. 393-402.

[15] Simon Haykin, *Neural Networks: A Comprehensive Foundation*, 1st ed. Upper Saddle River, NJ: Prentice Hall PTR, 1994.

[16] Steven W Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego: California Technical Publishing, 1997.

[17] Karl Gustafson and Guido Sartoris, "Assigning Initial Weights in Feedforward Neural Networks," in *IFAC Proceedings Volumes*, vol. 31, Rio Patras, 1998, pp. 1053-1058.

[18] Diederik P Kingma and Jimmy Lei Ba, "Adam: A Method for Stochastic Optimization," in *Proceedings of the International Conference on Learning Representations (ICLR)*, San Diego, 2015.

[19] Python Software Foundation. The official home of the Python Programming Language. [Online]. https://www.python.org

[20] Python Software Foundation. PyPI - the Python Package Index : Python Package Index. [Online]. https://pypi.python.org/pypi

[21] TensorFlow. TensorFlow - An open-source software library for Machine Intelligence. [Online]. https://www.tensorflow.org

[22] Aymeric Damien. TFLearn | TensorFlow Deep Learning Library. [Online]. http://tflearn.org

[23] NumPy. NumPy - Fundamental package for scientific computing with Python. [Online]. http://www.numpy.org

[24] Slava Korolev. (2017, July) Neural network to play a Snake game. Towards Data Science. [Online]. https://towardsdatascience.com/today-im-going-to-talk-about-a-small-practical-example-of-using-neural-networks-training-one-to-6b2cbd6efdb3

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, "Deep Residual Learning for Image Recognition," Microsoft Research, 2015. [Online]. https://arxiv.org/abs/1512.03385v1