CLASSIFICATIONS AND PREDICTIONS OF A SIMPLIFIED POKER GAME

Submitted in fulfillment of the requirements for the degree of Bachelor of Computer Science, Honours

AUTHOR: ARMAN KOCHARYAN
SUPERVISOR: DR ANTHONY WHITE
HONOURS PROJECT (COMP 4905A)

Carleton University, 1125 Colonel By Dr, Ottawa, ON K1S 5B6

DECEMBER 16, 2018

Abstract:

The problem this project is trying to solve is the ability to predict an outcome of a simplified poker game

To solve this problem the following approach was taken – a neural network was trained to classify poker hands and the hand odds of the opposing player were calculated by finding all the possible hand combinations. These values were then generated as datasets for multiple poker games and used to train a new neural network to predict a win or a loss.

Experimentation was performed on data that was downloaded from the UCI Machine Learning Repository as well as data that was generated by the project source code. The Code implementation of this project can be found in the following public GitHub repository https://github.com/armankocharyan/Honours-Project/tree/master/Data%20Generator

Although the results were good, it stands that the neural network can be improved by training it with more valid data. The predictions can also be improved by implementing other AI methods and approaches such as decision trees and reinforcement learning.

Acknowledgements

I would like to thank Dr. White for being my supervisor for this project. Dr White has given me continues guidance and support since the start of the project. I would also like to thank the department of Computer Science and all my previous professors and instructors for providing me with the knowledge that I have acquired in the last four years. Finally, I would like to thank all the online educators that released countless of free material that helped me throughout university and this project.

Table of Contents

Abstract: ii
Acknowledgements iii
List of Figures vi
List of Tables vii
List of Equations viii
1.0 Introduction 1
  1.1 General 1
  1.2 Scope 1
  1.3 Related Work 1
2.0 Background 2
  2.1 Overview 2
  2.2 Neural Network 2
    2.2.1 Structure 2
    2.2.2 Feedforward 2
    2.2.3 Backpropagation 3
  2.3 Definitions 8
  2.3 Function Definitions and Operations 9
3.0 Methodology 10
  3.1 Overview 10
  3.2 How cards are represented in data sets? 10
  3.3 How poker hands are represented in data sets? 11
  3.4 Can a neural network learn to classify a poker hand? 12
  3.5 Can a feedforward neural network predict a poker game? 13
    3.5.1 How to generate training and testing data for our neural network? 13

3.5.2 How to train our neural network to predict game outcomes? 15

4.0 Experimentation 17

4.1 Overview 17

4.2 Experiment 17

4.3 Setup 17

4.4 Design 18

4.4.3 Hand Classification Network 18

4.3.4 Game Predicting Neural Network 22

4.4 Code Implementation Process 26

4.4.1 Neural Network Code Implementation 26

4.4.2 Data Generator Code Implementation 28

5 Discussion 29

5.1 Overview 29

5.2 Results 29

5.2.1 Hand Classifier 29

5.2.2 Game predictor 31

6 Future Implementations 33

References 34

List of Figures

Figure 1 Two input perceptron 3
Figure 2 graph of f(x) = x^2 + 8 and rate of change at point (1, 9) 4
Figure 3 Iteratively finding the minimum of a function........ 5
Figure 4 Hand Classification Neural Network 12
Figure 5 Structure of the game predicting neural network........ 16
Figure 6 Classification Network 19
Figure 7 Game Classification Neural Network 23
Figure 8 Initial Hand Classifier accuracy rate 29
Figure 9 Outcome of the new Hand Classifying Network 30
Figure 10 Initial Game Predicting neural network........ 31
Figure 11 Prediction Accuracy of the new Game Predicting Network 32

List of Tables

Table 1 Minimizing table values.. 5
Table 2 Poker card rank (lowest to highest) 10
Table 3 Poker card suit (lowest to highest) 10
Table 4 Poker hands with their respective cards and weights 11
Table 5 Other Player hand odds (data obtained from a randomly generated game) 14

List of Equations

Equation 1 Chain rule 5

Equation 2 Mean Squared Error loss function 6

Equation 3 Sigmoid activation function 6

Equation 4 Combination formula 13

Equation 5 Softmax activation function 21

1.0 Introduction

1.1 General

Poker is a game of probability and randomness. People all over the world play different versions of the game competitively and for fun. This has inspired engineers and computer scientists to try to build machines that would play and beat the best poker players in the world. The incentive of this project is to design a system that would identify a poker hand given 5 cards and predict a win or a loss. It is assumed that the readers of this report are familiar with basic poker hands and rules.

1.2 Scope

Most of the popular poker game versions are very complex. They involve multiple players, limitless betting and are heavily human factor dependent. This project was built around a simpler version of a poker game, involving only two players. Each player is dealt 2 random cards from a 52-card deck, and 5 cards are put on the table that may be shared among both players. The player with the best hand wins the game.

1.3 Related Work

This project was inspired by the creators of the DeepStack algorithm in university of Alberta [Moravčík, Schmid, Burch, Lisý, Morrill, Bard, Davis, Waugh, Johanson and Bowling, 2017] Deep stack is an application of heuristic search methods. It was implemented to play a heads-up no-limit Texas hold'em poker, where it played and defeated 11 professional poker players. This project has been also inspired by the Classification-Neural-Network tutorial made by Julian Clayton [Clayton, 2018]. The Classification-Neural-Network provides help designing classification neural networks using python with TensorFlow.

2.0 Background

2.1 Overview

This section covers the tools and the ideas that were used to implement this project. The reader should have a good understanding of the principles described in this section before proceeding with the rest of the paper.

2.2 Neural Network

Artificial neural networks were inspired by biological neural networks [Aggarwal, 2018]. The purpose of a neural network system is to learn to complete tasks by examining samples of data. These samples of data are collected from none-automated task-specific processes. Neural Networks are used in many Medical Diagnosis, Credit Rating, Voice and Picture recognizing applications [Patterson and Gibson, 2017]. In this project two feedforward neural network were implemented and trained using back propagation.

2.2.1 Structure

Neural Networks consist of layers of nodes that are connected to one another [Aggarwal, 2018]. Each node can be represented as a function, where it takes in single or multiple input(s) and produces output(s). The first layer of a neural network is known as the input layer (also known as a placeholder), this layer gets fed samples of raw data. The last layer is the output layer, this layer holds the values that were predicted by the neural network. The middle layers are known as the hidden layers.

2.2.2 Feedforward

A feedforward neural network is a system where data flows in one direction [Patterson and Gibson, 2017]. To understand how a feedforward neural network works, let's look at a two-input perceptron (See Figure 1). Each connection between the nodes in the input layer and the node in the output layer is assigned a weight value. To calculate the output node value (o_0), the sum of the bias value (b_0) is added to the sum of the product of each input node (i_0, i_1), with its respective weight value (w_0, w_1). This value is then distributed into a defined range by an activation function (in this case the range is between 0 and 1 because of the sigmoid activation function).

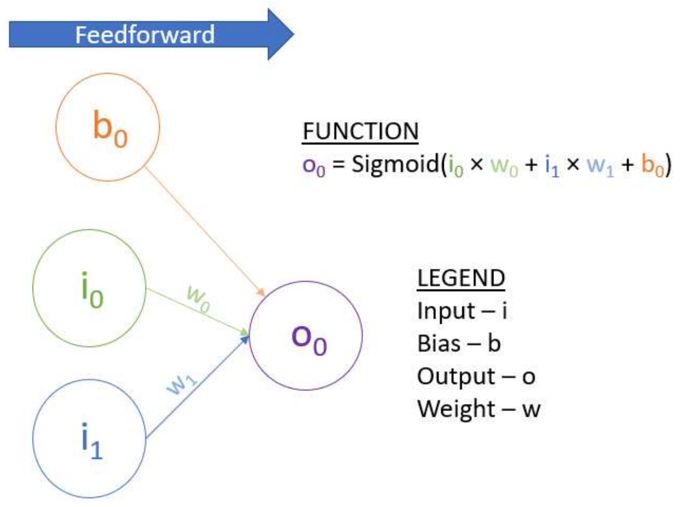


Figure 1 Two input perceptron

2.2.3 Backpropagation

Initially the weight and the bias values are completely randomized by us, which means the output values that the neural network will predict, are most likely not going to be correct. Since we know the target values of each data input, we can calculate the total Error by using a loss function. Root Mean Square, Squared Error or absolute error are all examples of loss functions designed to calculate the total error (E_{total}) between the predictions and the target values. Therefore, minimizing the total error in our neural network would improve output predictions. The process of updating each weight and bias value with respect to the total error, is known as backpropagation [Aggarwal, 2018].

We can find the rate of change of a function at a specific point by finding the derivative of the function and solving it at that point [Adams and Essex, 2017]. Let's look at an example in Figure 2.

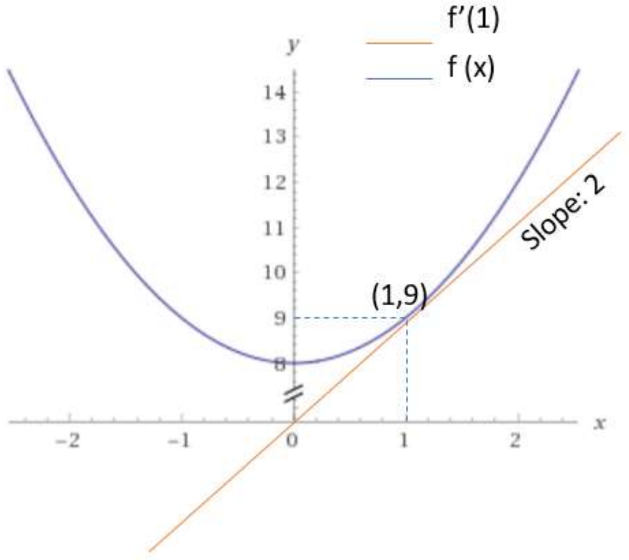


Figure 2 graph of f(x) = x^2 + 8 and rate of change at point (1, 9)

We can find the minimum of the function f(x) by finding it's derivative f'(x), setting it to zero and solving for x. This process seems trivial for a function like f(x), but for more complex functions that have multiple inputs it might not be an easy task.

Therefore, we take a different approach called gradient descent [Zill and Wright, 2011]. If we subtract our slope times a learning rate from a random x-value for multiple iterations, we will approach the function minimum (see Figure 3). When the slope of the tangent line equals to a value that is close to zero, we could approximate the minimum of the function (see Table 1).

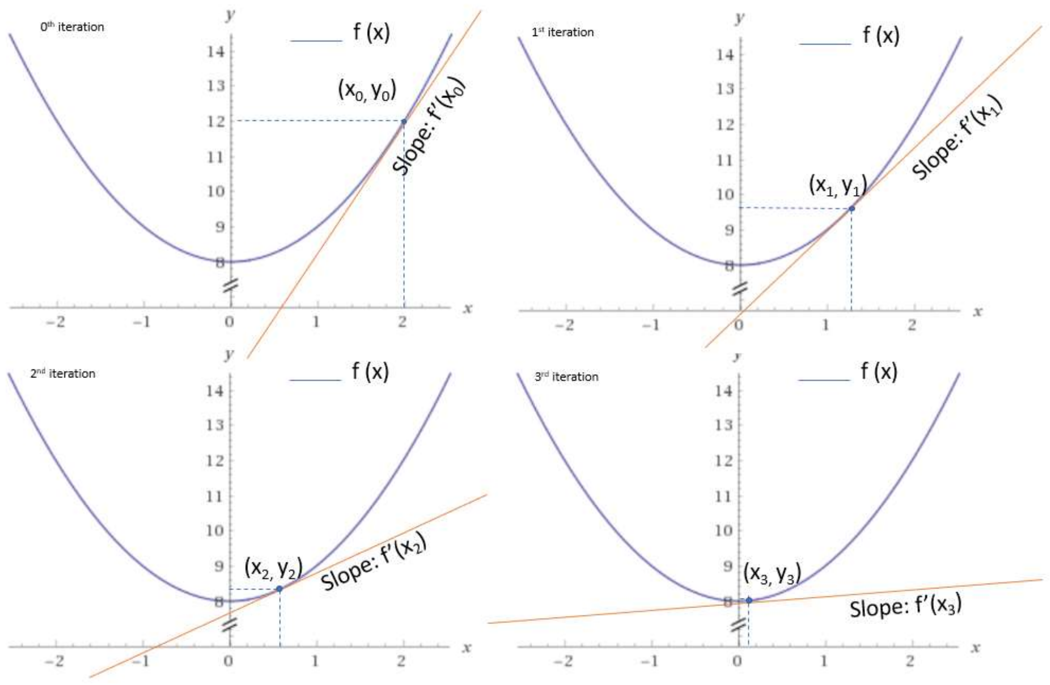


Figure 3 Iteratively finding the minimum of a function

Table 1 Minimizing table values

Iteration	X-Values	Y-Values	Comment
0	x_0	y_0 ($f(x_0) = y_0$)	We select a random x-value (in this case x_0)
1	x_1	y_1 ($f(x_1) = y_1$)	$x_1 = x_0 – f'(x_0) \times$ learning rate
2	x_2	y_2 ($f(x_2) = y_2$)	$x_2 = x_1 – f'(x_1) \times$ learning rate
3	x_3	y_3 ($f(x_3) = y_3$)	$x_3 = x_2 – f'(x_3) \times$ learning rate

We take a similar approach with our neural network. We use the chain rule (see Equation 1) to partially differentiate our loss function with respect to each weight and bias [Adams and Essex, 2017]. Once we do this, we multiply each respective weight and bias derivate with a learning rate and subtract them each original value

$$IF\ z = f(y)\ AND\ y = g(x), THEN\ \frac{\partial z}{\partial x} = \left(\frac{\partial z}{\partial y}\right)\left(\frac{\partial y}{dx}\right) = f'(y)g'(x) = f'\big(g(x)\big)g'(x)$$

Equation 1 Chain rule

To demonstrate this process on our perceptron (see Figure 1) we will use the squared error function (Equation 2) as our loss function:

$$E_{total} = \sum ½(target - output)^2$$

Equation 2 Mean Squared Error loss function

We proceed:

$$E_{total} = ½(target_{O_0} - o_0)^2$$

We know that output (o_0) equates to the output of an activation (see Equation 3) function taking in net of all the weights and biases in the input layer:

$$\sigma(x) = \frac{1}{1 + e^{-(x)}}$$

Equation 3 Sigmoid activation function

We proceed:

$$o_0 = sigmoid(net_{O_0})$$
$$o_0 = 1/(1 + e^{-net_{O_0}})$$

Net of the input layer:

$$net_{o_0} = i_0 \times w_0 + i_1 \times w_1 + b_0$$

Now we need to partially differentiate E_{total} with respect to w_0, w_1, and b_0. We do this by applying the chain rule:

$$\delta E_{total}/_{\delta w_0} = \frac{\delta E_{total}}{\delta o_0} \times \frac{\delta o_0}{\delta net_{O_0}} \times \frac{\delta net_{O_0}}{\delta w_0}$$

Let's break down each derivate separately:

$$\delta E_{total}/_{\delta o_0} = -\left(target_{o_0} - o_0\right)$$

$$\delta o_0 /_{\delta net_{O_0}} = o_0 \times (1 - o_0)$$

$$\delta net_{O_0} /_{\delta w_0} = i_0 \times w_0^{-1}$$

Putting it all together:

$$\delta E_{total}/_{\delta w_0} = \left[-\left(target_{O_0} - o_0\right)\right] \times [\, o_0 \times (1 - o_0)] \times [i_0 \times w_0^{-1}]$$

We repeat this step for w_1, and b_0:

$$\delta E_{total}/_{\delta w_1} = \frac{\delta E_{total}}{\delta o_0} \times \frac{\delta o_0}{\delta net_{O_0}} \times \frac{\delta net_{O_0}}{\delta w_1}$$

$$\delta E_{total}/_{\delta b_0} = \frac{\delta E_{total}}{\delta o_0} \times \frac{\delta o_0}{\delta net_{O_0}} \times \frac{\delta net_{O_0}}{\delta b_0}$$

We come up with new weight and bias values once we find our partial derivates by multiplying them with a learning rate and subtracting them from our initial values

$$w_0^{new} = w_0 - (learning\ rate) \times (^{\delta E_{total}}/_{\delta w_0})$$

$$w_1^{new} = w_1 - (learning\ rate) \times (^{\delta E_{total}}/_{\delta w_1})$$

$$b_0^{new} = b_0 - (learning\ rate) \times (^{\delta E_{total}}/_{\delta b_0})$$

We do this process for multiple iterations with different data sets until our total error (E_{total}) is minimized

2.3 Definitions

This section defines key variables and ideas that were used to implement the project

Bias

A randomly generated constant value(s) used in a neural network to predict outcomes. The bias values usually get tweaked during the training process of the neural network

Derivate

Measures the rate of change of a function value with respect to change in its input.

Hidden Layer

All the nodes in a neural network that do not belong in the input layer or the output layer, are in the hidden layers. The number of hidden layers vary on the design of the neural network

Learning Rate

A proportional constant value that is chosen to adjust each weight and bias value during training

Testing Data

Input and target values for a neural network. This data is used to test the neural network once it is trained

Training Data

Input and target values for a neural network. This data is used to train the neural network

Weight

A value that serves as a connection between nodes. Weighs are used in a process to output predictions. These weight values change during the training process of a neural network

2.3 Function Definitions and Operations

This section talks about some of the functions that were used during the project implementation.

Activation Function

An activation function defines the range of an output given a set of inputs. (sigmoid, softmax, inverse square)

Loss Function

Functions that evaluate how well a prediction is compared to its respective target value

Gradient Descent

An iterative optimization algorithm that finds a minimum of a function, by moving in proportional steps towards the functions negative gradient vector

Matrix Operations

List of mathematical procedures that can be applied to a matrix, such as addition, subtraction, multiplication and others.

Combination Function

Number of subset combinations in a set

3.0 Methodology

3.1 Overview

This section breaks down a list of questions and answers that unravel the methodology behind classifying poker hands and predicting wins and losses. The answers will explain the thought process behind the creation of the datasets and algorithms.

3.2 How cards are represented in data sets?

A poker card consists of two attributes, a rank and a suit. Poker decks consist of 52 cards, with each card having a unique suit and rank. There are 13 ranks and 4 suits in total (See Tables 2 and 3).

Table 2 Poker card rank (lowest to highest)

Rank	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King	Ace

Table 3 Poker card suit (lowest to highest)

Suit	Clubs	Diamonds	Hearts	Spades

Knowing this we can represent cards in lists of integers, with two adjacent elements making a single card. First integer being a suit (1 – 4) and the other being a rank (1 – 13).

For example:

Three of clubs can be presented as:

Cards = {1, 3}

Four of diamonds and 2 of spades can be presented as

Cards = {2, 4, 4, 2}

3.3 How poker hands are represented in data sets?

A poker hand consists of 5 cards that are dealt from a shuffled deck. There are 10 possible hands with one outweighing the other (See Table 4).

Table 4 Poker hands with their respective cards and weights

Hand	Cards	Weight
Royal Flush	{3, 13, 3, 12, 3, 11, 3, 10, 3, 9}	9
Straigh Flush	{3, 9, 3, 8, 3, 7, 3, 6, 3, 5}	8
Four of Kind	{3, 9, 1, 9, 2, 9, 4, 9, 3, 5}	7
Full House	{3, 9, 1, 9, 2, 9, 4, 5, 3, 5}	6
Flush	{3, 9, 3, 2, 3, 10, 3, 12, 3, 5}	5
Straight	{3, 2, 2, 3, 1, 4, 4, 5, 3, 6}	4
Three of Kind	{3, 2, 2, 2, 1, 2, 4, 5, 3, 6}	3
Two Pair	{3, 2, 2, 2, 1, 9, 4, 6, 3, 6}	2
Pair	{3, 2, 2, 2, 1, 9, 4, 13, 3, 6}	1
High Card	{3, 10, 2, 2, 1, 9, 4, 13, 3, 6}	0

By giving weight to our hands (0 – 9) we can represent a poker hand with 11 elements in a list. With first 10 elements representing our five cards and the last element representing the weight of the Hand.

For example:

A flush can be represented as:

$$Hand = \{3, 9, 3, 2, 3, 10, 3, 12, 3, 9, 5\}$$

A Pair can be represented as:

$$Hand = \{3, 2, 2, 2, 1, 9, 4, 13, 3, 6, 1\}$$

Data with the structure described above was downloaded from UCI machine learning repository [Cattral and Oppacher, 2007].

3.4 Can a neural network learn to classify a poker hand?

Given the definitions that we made for poker cards and hands, and the data that we have gathered from the Machine learning repository we may construct a neural network that can classify a poker hand.

The neural network would consist of ten input nodes and ten output nodes (see Figure 4). The number of weights and biases would depend on the structure of the neural network (number of hidden layers, number of nodes in each layer, etc.). The values of the weights and biases would be randomly generated and stored. The network can then proceed to start learning the poker hands, by feeding its input layer with first ten elements of our dataset and using the last element of the dataset as a target value. After many iterations and many datasets the network should minimize the total error and learn to classify poker hands. More data sets can then be used to test the classification accuracy.

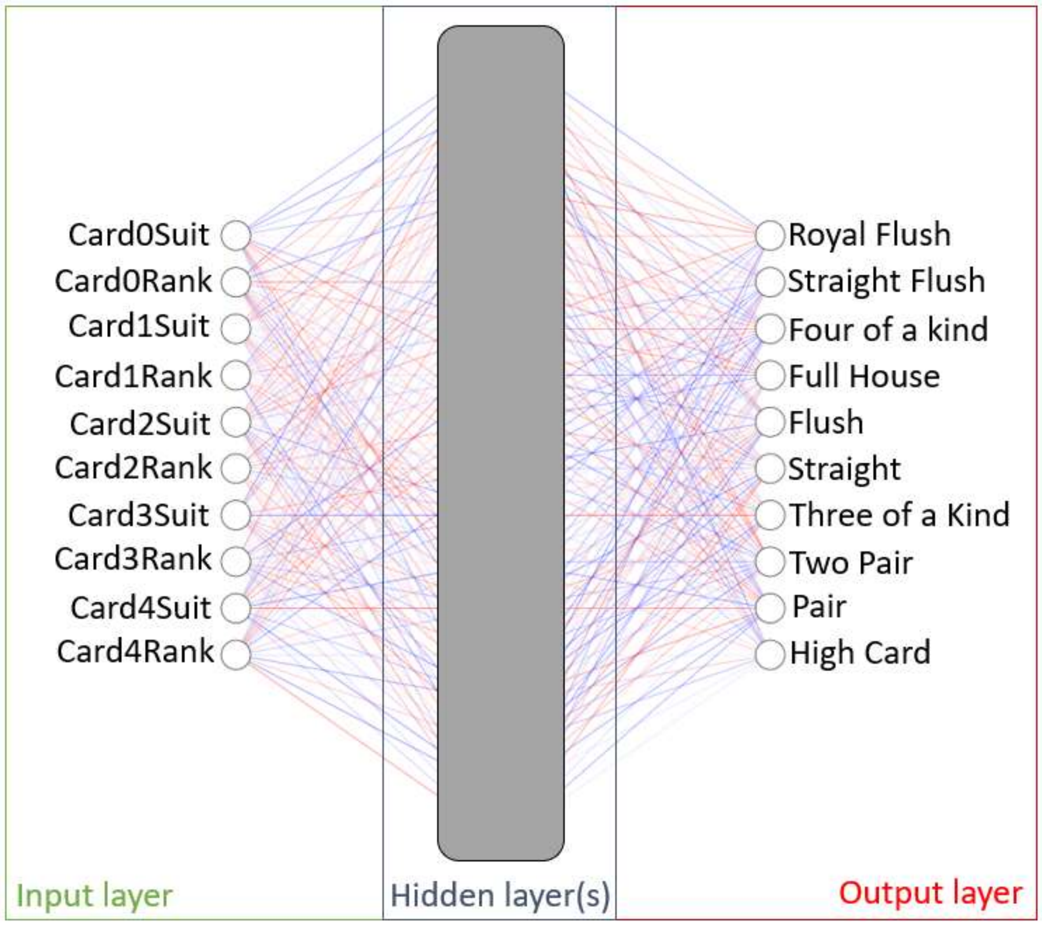


Figure 4 Hand Classification Neural Network

3.5 Can a feedforward neural network predict a poker game?

Remark:

Each poker game involves two players, AIPlayer and otherPlayer. Each player is dealt 2 cards from a 52-card generated poker deck. Five cards are put on the table and are shared between the two players. Player that has the best hand from the seven available cards wins the game.

AIPlayer needs to predict whether the game can be won or lost given only the information that is available to it. Following is the information that is available to the AIPlayer:

- Two cards that were dealt to him
- Five shared cards on the desk
- Size of the deck at any point of the game
- All the cards in the deck before it is dealt
- AIPlayers best possible hand

We are going to break down this question into two parts:

3.5.1 How to generate training and testing data for our neural network?

We will be using the otherPlayers hand odds as our input data.

AIPlayer can calculate the hand odds of the otherPlayer with the available information given to him using the combination formula [Simmons, 2017].

$$\binom{n}{r} = \frac{n!}{r!\,(n-r)!}$$

Equation 4 Combination formula

Combination formula returns the number of possible combinations of subsets length *r* from a set of *n* elements. Initially we know that the deck size is 52, but after dealing AIPlayer two cards, and putting five shared cards on the table, we know that the deck size is reduced to 45. From this we can find all the two cards that could have possible been dealt to the otherPlayer.

$\binom{n}{r} = \binom{45}{2} = 990$

Now we need find all the possible hands that AI can have taking consideration these possibilities:

- Selecting 3 cards from the shared 5 cards and using 2 of its own
- Selecting 4 cards from the shared 5 cards and using 1 of its own
- Selecting all 5 cards of the shared 5 cards and not using any of its own cards

The sum of all these combinations would give us all the possible hands the otherPlayer can have.

$$Total = \binom{45}{2}\binom{5}{3} + \binom{45}{1}\binom{5}{4} + \binom{45}{0}\binom{5}{5}$$

$Total =$ (990)(10) + (45)(5) + (1)(1)

$Total =$ 9900 + 225 + 1

$Total =$ 10126

Having the set of all possible otherPlayer cards, we may now find the odds of the otherPlayers hand by classifying and keeping track of each possible otherPlayer hand (see Table 5).

Note: we may do this by reformatting our hands into appropriate datasets and using our classification neural network.

Table 5 Other Player hand odds (data obtained from a randomly generated game)

Total	Royal Flush	Straight Flush	Four of a Kind	Full House	Flush	Straight	Three of a kind	Two Pair	One Pair	High Card
10126 (100%)	0 (0%)	0 (0%)	3 (0.0296%)	27 (0.267%)	0 (0%)	56 (0.553%)	297 (2.93%)	693 (6.84%)	4767 (47.1%)	4283 (42.3%)

3.5.2 How to train our neural network to predict game outcomes?

Using the logic described in sections 3.4 and 3.5.5 we can generate many poker games and create datasets. To find the target value for each game we can choose a random hand out of the 10126 generated hands and compare it with AIPlayers hand. The target value is set to 1 in the outcome is a win and 0 if the outcome is a loss.

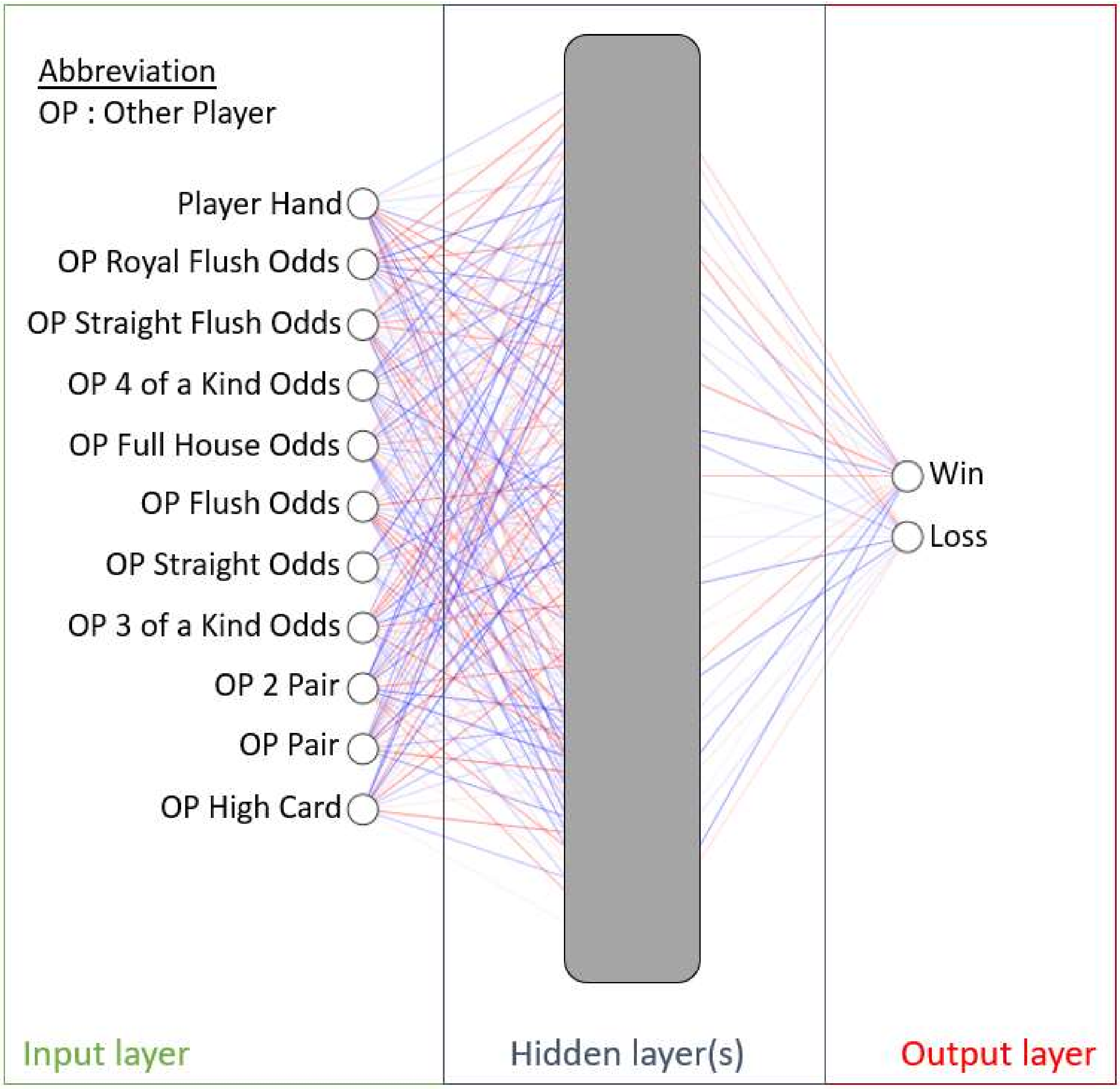
A prediction dataset can be a list with 12 elements, with first element being AIPlayers hand (acquired from the hand classification neural network). The next ten elements are other players hand odds (see Table 5). The last element is the target value that is used to train and test the neural network.

$dataSet = \{0.3, 0.42, 0.47, 0.068, 0.029, 0.0055, 0, 0.0027, 0.0003, 0, 0, 1\}$

Once we generate thousands of datasets, we may construct our neural network which could predict the game result (see Figure 5)

The number of weights and biases would depend on the structure of the neural network (number of hidden layers, number of nodes in each layer, etc.). The values of the weights and biases would be randomly generated and stored in matrices. The Neural Network can learn to predict game wins and losses by iterating through many samples of data and optimizing its weight and bias values.

Figure 5 Structure of the game predicting neural network



4.0 Experimentation

4.1 Overview

This section covers the implementation of the project. The poker game project consists of two parts, hand classification of a poker hand, and game predictions. We will start with covering all the tools that were used during development and then proceed with the setup and requirements. In the design section we will cover the structures of the neural networks.

4.2 Experiment

Following software and tools were used for the project development:

- Windows 10 64-bitAtom text editor
- Python version 3.6.7
    - TensorFlow version 1.12.0
    - CSV 0.0.11
- Windows Command Prompt
- Windows File System
- Eclipse Desktop IDE
    - Apache maven 4.0.0
    - Junit 4.12
    - Java Development Kit 1.8.0_191
    - Java Runtime Environment 1.8.0_191
- NN-SVG (visualization tool)

4.3 Setup

The project was developed on a windows 10 operating system. All the neural networks were made with python and written in the Atom text editor. This project can only be run on python versions 3.5 or 3.6 due to TensorFlow module not being compatible with newer versions. The data generator is written in java and can be ran as an Eclipse Maven project. For more details please check the ReadMe file in the project github repository.

4.4 Design

This section breaks down the design of the hand classification and game prediction neural networks.

4.4.3 Hand Classification Network

The structure of the hand classification neural network (see Figure 6) consists of:

- Input layer: 10 nodes
- Hidden layer0: 300 nodes
- Hidden layer1: 300 nodes
- Output layer: 10 nodes

Since this neural network is large, we break down the nodes into matrices and use matrix operations to come up with our output [Lang, 2011].

Starting from the input layer we can put all the input nodes in a 1×10 matrix. For all the first bias values we create a 1 × 300 matrix, due to hidden layer0 containing 300 nodes. For the weight values we create a 10 × 300 matrix, due to our input layer having 10 nodes and our hidden layer0 having 300.

Following are the matrices that are used to find the values for hidden layer0 nodes:

$$I_0 = \begin{bmatrix} i_0^{(0)} \\ i_1^{(0)} \\ i_2^{(0)} \\ i_3^{(0)} \\ i_4^{(0)} \\ i_5^{(0)} \\ i_6^{(0)} \\ i_7^{(0)} \\ i_8^{(0)} \\ i_9^{(0)} \end{bmatrix} \qquad b_0 = \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ b_2^{(0)} \\ b_3^{(0)} \\ \vdots \\ b_{295}^{(0)} \\ b_{296}^{(0)} \\ b_{297}^{(0)} \\ b_{298}^{(0)} \\ b_{299}^{(0)} \end{bmatrix} \qquad W_{hid} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,9} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,9} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,9} \end{bmatrix}$$

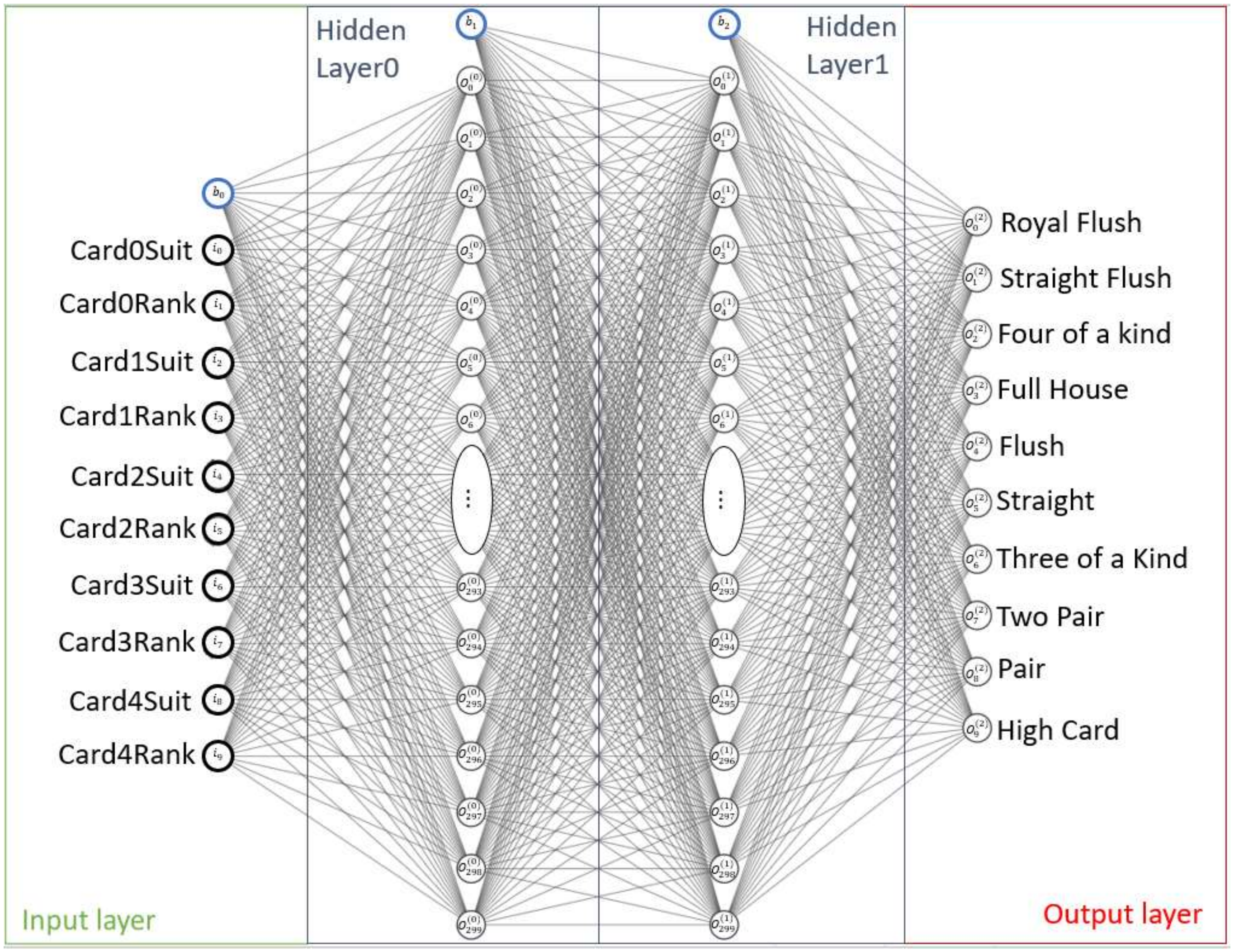


Figure 6 Classification Network

We can calculate the values in each node in the hidden layer0 by doing the following matrix operations:

$$hiddenLayer0 = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,9} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,9} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,9} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_9 \end{bmatrix} + \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ \vdots \\ b_{299}^{(0)} \end{bmatrix}\right)$$

Where:

$$O_n^{(0)} = \sigma(w_{(n,0)} i_1^{(0)} + w_{(n,1)} i_1^{(0)} + \cdots + w_{(n,9)} i_9^{(0)} + b_n^{(0)})$$

We use all the nodes in the hidden layer0 as our new input. We then create a new 1 × 300 bias matrix and new 300 × 300 weight matrix:

$$I_1 = \begin{bmatrix} i_0^{(1)} \\ i_1^{(1)} \\ i_2^{(1)} \\ i_3^{(1)} \\ \vdots \\ i_{295}^{(1)} \\ i_{296}^{(1)} \\ i_{297}^{(1)} \\ i_{298}^{(1)} \\ i_{299}^{(1)} \end{bmatrix} = \begin{bmatrix} o_0^{(0)} \\ o_1^{(0)} \\ o_2^{(0)} \\ o_3^{(0)} \\ \vdots \\ o_{295}^{(0)} \\ o_{296}^{(0)} \\ o_{297}^{(0)} \\ o_{298}^{(0)} \\ o_{299}^{(0)} \end{bmatrix} \qquad b_1 = \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_{295}^{(1)} \\ b_{296}^{(1)} \\ b_{297}^{(1)} \\ b_{298}^{(1)} \\ b_{299}^{(1)} \end{bmatrix} \qquad W_{hidden1} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix}$$

We can now calculate the output value in the hidden layer1 by using the following matrix operations:

$$hiddenLayer1 = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{299} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ \vdots \\ b_{299}^{(1)} \end{bmatrix}\right)$$

Where:

$$O_n^{(1)} = \sigma(w_{(n,0)} i_0^{(1)} + w_{(n,1)} i_1^{(1)} + \cdots + w_{(n,299)} i_{299}^{(1)} + b_n^{(1)})$$

Now we may compute the output values. To do this we generate new matrices for our final layer. We use all the nodes in the hidden layer1 as our new input. Our output layer has 10 nodes we generate a 1 × 10 bias matrix. We also generate a new weight matrix size of 300 × 10.

$$I_2 = \begin{bmatrix} i_0^{(2)} \\ i_1^{(2)} \\ i_2^{(2)} \\ i_3^{(2)} \\ \vdots \\ i_{295}^{(2)} \\ i_{296}^{(2)} \\ i_{297}^{(2)} \\ i_{298}^{(2)} \\ i_{299}^{(2)} \end{bmatrix} = \begin{bmatrix} o_0^{(1)} \\ o_1^{(1)} \\ o_2^{(1)} \\ o_3^{(1)} \\ \vdots \\ o_{295}^{(1)} \\ o_{296}^{(1)} \\ o_{297}^{(1)} \\ o_{298}^{(1)} \\ o_{299}^{(1)} \end{bmatrix} \quad b_2 = \begin{bmatrix} b_0^{(2)} \\ b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \\ b_4^{(2)} \\ b_5^{(2)} \\ b_6^{(2)} \\ b_7^{(2)} \\ b_8^{(2)} \\ b_9^{(2)} \end{bmatrix} \quad W_{hidden2} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{9,0} & w_{9,1} & \cdots & w_{9,299} \end{bmatrix}$$

We can now calculate the output nodes for our neural network. We use a softmax (see Equation 5) instead of a sigmoid for our last layer because we need to classify and choose between ten alternatives (ten poker hands). Softmax allows us to get a probability distribution which can be applied cross entropy loss function.

$$f(x_j) = \frac{e^{x_j}}{\Sigma_i e^{x_i}}$$

Equation 5 Softmax activation function

$$output = softmax\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{9,0} & w_{9,1} & \cdots & w_{9,299} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{299} \end{bmatrix} + \begin{bmatrix} b_0^{(2)} \\ b_1^{(2)} \\ \vdots \\ b_9^{(2)} \end{bmatrix} \right)$$

Where:

$$O_n^{(2)} = softmax(w_{(n,0)} i_0^{(2)} + w_{(n,1)} i_1^{(2)} + \cdots + w_{(n,299)} i_{299}^{(2)} + b_n^{(2)})$$

4.4.4 Game Predicting Neural Network

The structure of the game prediction neural network (see Figure 7) consists of:

- Input layer: 11 nodes
- Hidden layer0: 300 nodes
- Hidden layer1: 300 nodes
- Hidden layer2: 300 nodes
- Output layer: 2 nodes

Since this neural network is large, we break down the nodes into matrices and use matrix operations to predict our output.

Starting from the input layer we can put all the input nodes in a 1×11 matrix. For all the first bias values we create a 1 × 300 matrix, due to hidden layer0 containing 300 nodes. For the weight values we create a 11 × 300 matrix, due to our input layer having 11 nodes and our hidden layer0 having 300.

Following are the matrices that are used to find the values for hidden layer0 nodes:

$$I_0 = \begin{bmatrix} i_0^{(0)} \\ i_1^{(0)} \\ i_2^{(0)} \\ i_3^{(0)} \\ i_4^{(0)} \\ i_5^{(0)} \\ i_6^{(0)} \\ i_7^{(0)} \\ i_8^{(0)} \\ i_9^{(0)} \\ i_{10}^{0} \end{bmatrix} \qquad b_0 = \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ b_2^{(0)} \\ b_3^{(0)} \\ \vdots \\ b_{295}^{(0)} \\ b_{296}^{(0)} \\ b_{297}^{(0)} \\ b_{298}^{(0)} \\ b_{299}^{(0)} \end{bmatrix} \quad W_{hidd} \quad = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,10} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,10} \end{bmatrix}$$

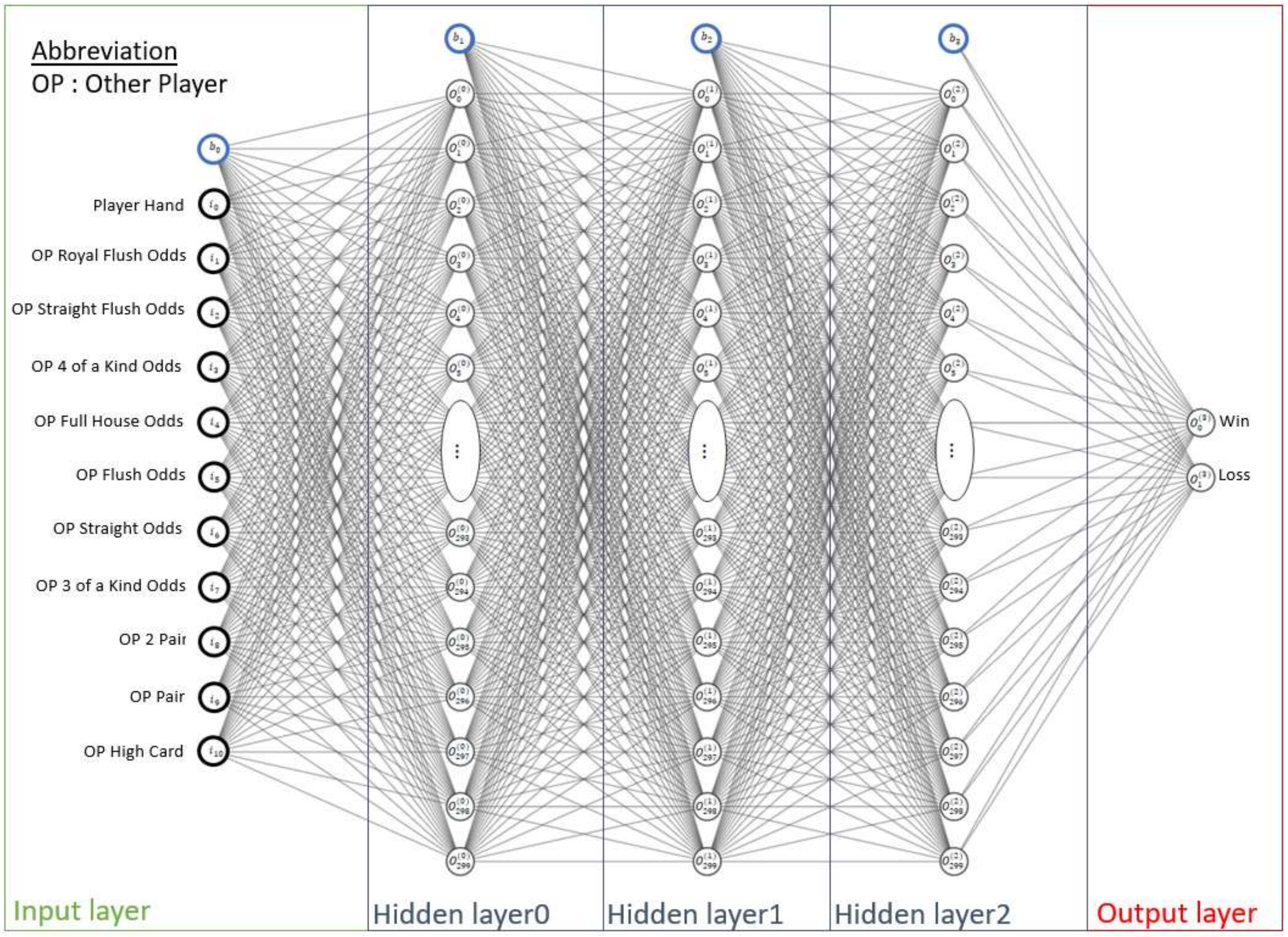


Figure 7 Game Classification Neural Network

We can calculate the values in each node in the hidden layer0 by doing the following matrix operations:

$$hiddenLayer0 = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,10} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,10} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,10} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{10} \end{bmatrix} + \begin{bmatrix} b_0^{(0)} \\ b_1^{(0)} \\ \vdots \\ b_{299}^{(0)} \end{bmatrix}\right)$$

Where:

$$O_n^{(0)} = \sigma(w_{(n,0)} i_1^{(0)} + w_{(n,1)} i_1^{(0)} + \cdots + w_{(n,10)} i_{10}^{(0)} + b_n^{(0)})$$

We use all the nodes in the hidden layer0 as our new input. We then create a new 1 × 300 bias matrix and new 300 × 300 weight matrix.

$$I_1 = \begin{bmatrix} i_0^{(1)} \\ i_1^{(1)} \\ i_2^{(1)} \\ i_3^{(1)} \\ \vdots \\ i_{295}^{(1)} \\ i_{296}^{(1)} \\ i_{297}^{(1)} \\ i_{298}^{(1)} \\ i_{299}^{(1)} \end{bmatrix} = \begin{bmatrix} o_0^{(0)} \\ o_1^{(0)} \\ o_2^{(0)} \\ o_3^{(0)} \\ \vdots \\ o_{295}^{(0)} \\ o_{296}^{(0)} \\ o_{297}^{(0)} \\ o_{298}^{(0)} \\ o_{299}^{(0)} \end{bmatrix} \quad b_1 = \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ \vdots \\ b_{295}^{(1)} \\ b_{296}^{(1)} \\ b_{297}^{(1)} \\ b_{298}^{(1)} \\ b_{299}^{(1)} \end{bmatrix} \quad W_{hidde} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix}$$

We can calculate the values in each node in the hidden layer1 by doing the following:

$$hiddenLayer1 = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{299} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ \vdots \\ b_{299}^{(1)} \end{bmatrix}\right)$$

Where:

$$O_n^{(1)} = \sigma(w_{(n,0)} i_0^{(1)} + w_{(n,1)} i_1^{(1)} + \cdots + w_{(n,299)} i_{299}^{(1)} + b_n^{(1)})$$

We use all the nodes in the hidden layer1 as our new input. We then create a new 1 × 300 bias matrix and new 300 × 300 weight matrix.

$$I_2 = \begin{bmatrix} i_0^{(2)} \\ i_1^{(2)} \\ i_2^{(2)} \\ i_3^{(2)} \\ \vdots \\ i_{295}^{(2)} \\ i_{296}^{(2)} \\ i_{297}^{(2)} \\ i_{298}^{(2)} \\ i_{299}^{(2)} \end{bmatrix} = \begin{bmatrix} o_0^{(1)} \\ o_1^{(1)} \\ o_2^{(1)} \\ o_3^{(1)} \\ \vdots \\ o_{295}^{(1)} \\ o_{296}^{(1)} \\ o_{297}^{(1)} \\ o_{298}^{(1)} \\ o_{299}^{(1)} \end{bmatrix} \quad b_2 = \begin{bmatrix} b_0^{(2)} \\ b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \\ \vdots \\ b_{295}^{(2)} \\ b_{296}^{(2)} \\ b_{297}^{(2)} \\ b_{298}^{(2)} \\ b_{299}^{(2)} \end{bmatrix} \quad W_{hidden2} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix}$$

We can calculate the values in each node in the hidden layer1 by doing the following matrix operations:

$$hiddenLayer2 = \sigma\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \\ \vdots & \vdots & \ddots & \vdots \\ w_{299,0} & w_{299,1} & \cdots & w_{299,299} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{299} \end{bmatrix} + \begin{bmatrix} b_0^{(2)} \\ b_1^{(2)} \\ \vdots \\ b_{299}^{(2)} \end{bmatrix}\right)$$

Where:

$$O_n^{(2)} = \sigma(w_{(n,0)} i_0^{(2)} + w_{(n,1)} i_1^{(2)} + \cdots + w_{(n,299)} i_{299}^{(2)} + b_n^{(2)})$$

Now we can compute the output values. To do this we generate new matrices for our final layer. Since our output layer has 10 nodes in, we generate a 1 × 2 bias matrix. We use all the nodes in the hidden layer1 as our new input. We also generate a new weight matrix size of 300 × 2.

$$I_3 = \begin{bmatrix} i_0^{(3)} \\ i_1^{(3)} \\ i_2^{(3)} \\ i_3^{(3)} \\ \vdots \\ i_{295}^{(3)} \\ i_{296}^{(3)} \\ i_{297}^{(3)} \\ i_{298}^{(3)} \\ i_{299}^{(3)} \end{bmatrix} = \begin{bmatrix} o_0^{(2)} \\ o_1^{(2)} \\ o_2^{(2)} \\ o_3^{(2)} \\ \vdots \\ o_{295}^{(2)} \\ o_{296}^{(2)} \\ o_{297}^{(2)} \\ o_{298}^{(2)} \\ o_{299}^{(2)} \end{bmatrix} \quad b_3 = \begin{bmatrix} b_0^{(3)} \\ b_1^{(3)} \end{bmatrix} \quad W_{hidde} \quad = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \end{bmatrix}$$

We can now calculate the output nodes for our neural network. We use a softmax instead of a sigmoid for our last layer because we need to classify and choose between two alternatives. Like the hand classification network, softmax will allow us to get a probability distribution which can be applied to cross entropy loss function.

$$output = softmax\left(\begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{0,299} \\ w_{1,0} & w_{1,1} & \cdots & w_{1,299} \end{bmatrix} \times \begin{bmatrix} i_0 \\ i_1 \\ \vdots \\ i_{299} \end{bmatrix} + \begin{bmatrix} b_1^{(3)} \\ b_2^{(3)} \end{bmatrix}\right)$$

Where:

$$O_n^{(3)} = softmax(w_{(n,0)} i_0^{(3)} + w_{(n,1)} i_1^{(3)} + \cdots + w_{(n,299)} i_{299}^{(3)} + b_n^{(3)})$$

4.4 Code Implementation Process

This section talks about how code was implemented for our neural networks and data generator. It goes over the functions that were implemented as well as some of the functions that were used from imported libraries

4.4.1 Neural Network Code Implementation

The Classification neural network and the Hand prediction network were both implemented on python using the TensorFlow library

TensorFlow is a dataflow library that allows us to:

- Create matrices and vectors of any number of dimensions
- Allows data to from one layer to another (node to node)
- A library full of matrix operations and math operations

The input layer and the output layers are both set as TensorFlow *placeholders* in both of our neural networks. Placeholders are matrices and vectors that data can be fed to.

The weight and bias matrices are both generated using the TensorFlow *variable* and *random_normal* methods. The *variable* function creates the matrix and the *random_normal* populates the matrices with random numbers with a normal distribution.

From there on each layer was calculated using the TensorFlow's built in activation functions, *tensorflow.nn.sigmoid* for the nodes in the hidden layers and *tensorflow.nn.softmax* for the outer layers.

The loss function that we used to calculate the total error in both the classification and the game prediction networks is the root mean squared. We created this function in a scope using basic tensor flow math libraries such as *tensorflow.reduce_sum*, *tensorflow.subtract*, *tensorflow.square*, *tensorflow.sqrt*.

TensorFlow has functions it its train class that can be used to train the neural network. We used gradient descent to minimize our cost function so we used the tf.train.GradientDescentOptimizer().minimize().

The data was parsed from testing and training files with the csv python library. Csv.reader method was used to store the data in rows and columns. The data was then converted into vectors and matrices for our neural networks

For the full implementation for both neural networks please check the python code at:

https://github.com/armankocharyan/Honours-Project/tree/master/Neural%20Networks

4.4.2 Data Generator Code Implementation

We created a series of functions that identify poker hands, score given hands and generate games for our game prediction neural network. We have unit tested these functions using Junit framework. For the code implementation of please check:

https://github.com/armankocharyan/Honours-Project/tree/master/Data%20Generator

We created a *getCombination* method in our Combination class, that given a list and a size, statically returns a list of the combinations of subsets with given size. Using the combination formula and poker specific methods that is implemented in our Combination and PokerStrategy classes, we generated over a million poker games and outputted them in testing and training data files with the following format:

$\{0.1,0.42534071,0.47076833,0.06843769,0.02933044,0.00316018,0,0.0026664,0.00029627,0,0,0\}$

This file then was then moved to the neural network directory and was used to train and test the game prediction neural network.

5 Discussion

5.1 Overview

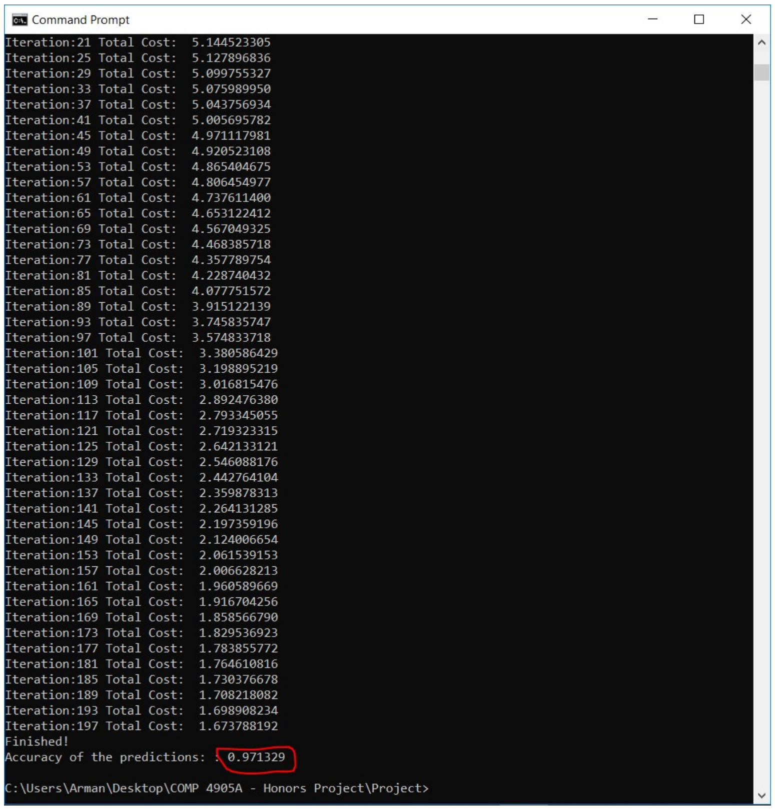
This section talks about the results of our experimentation. The reader will also get an idea of how the structure of the neural networks were improved throughout the implementation.

5.2 Results

This section is broken into the results of the hand classifying neural network and the results of the game predicting networks

5.2.1 Hand Classifier

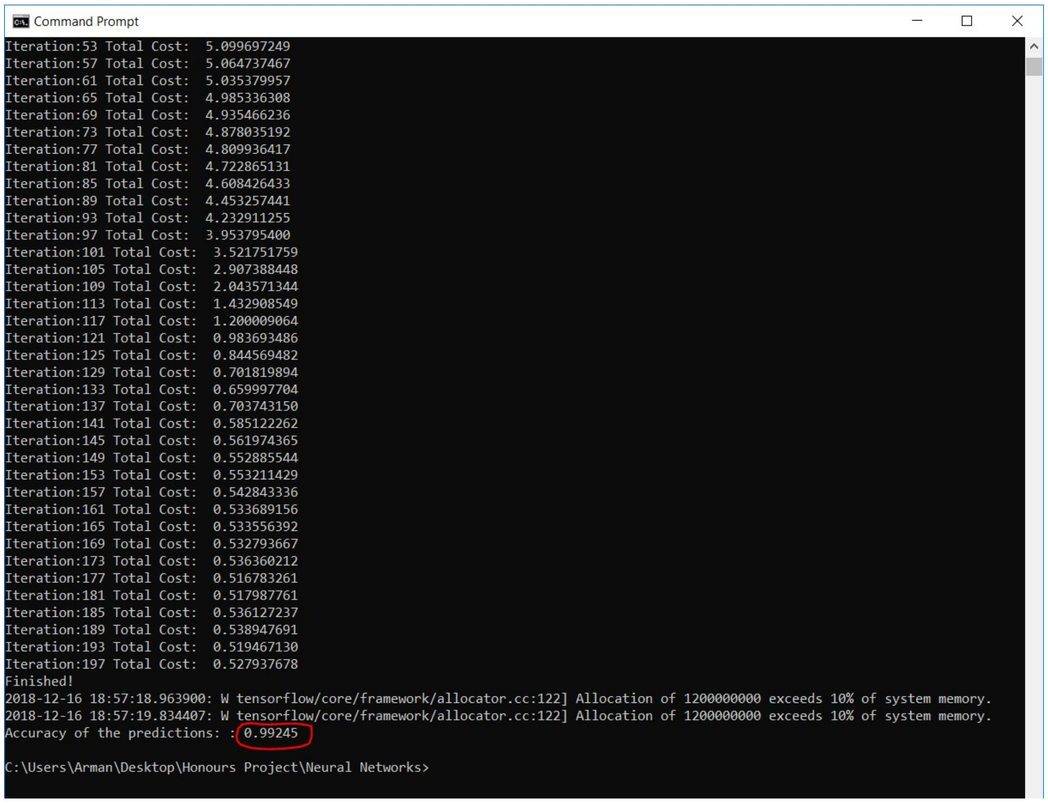
Using the data acquired from UCI machine learning repository we trained our neural network for 200 iterations at 1.15% learning rate. The initial neural network that was designed had one hidden layer consisting of 200 nodes. This neural network learned to classify hands at approximately 97.13% accuracy and total error of 1.67 (see Figure 8)



```
Iteration:21 Total Cost:  5.144523305
Iteration:25 Total Cost:  5.127896836
Iteration:29 Total Cost:  5.099755327
Iteration:33 Total Cost:  5.075989950
Iteration:37 Total Cost:  5.043756934
Iteration:41 Total Cost:  5.005695782
Iteration:45 Total Cost:  4.971117981
Iteration:49 Total Cost:  4.920523108
Iteration:53 Total Cost:  4.865404675
Iteration:57 Total Cost:  4.806454977
Iteration:61 Total Cost:  4.737611400
Iteration:65 Total Cost:  4.653122412
Iteration:69 Total Cost:  4.567049325
Iteration:73 Total Cost:  4.468385718
Iteration:77 Total Cost:  4.357789754
Iteration:81 Total Cost:  4.228740432
Iteration:85 Total Cost:  4.077751572
Iteration:89 Total Cost:  3.915122139
Iteration:93 Total Cost:  3.745835747
Iteration:97 Total Cost:  3.574833718
Iteration:101 Total Cost:  3.380586429
Iteration:105 Total Cost:  3.198895219
Iteration:109 Total Cost:  3.016815476
Iteration:113 Total Cost:  2.892476380
Iteration:117 Total Cost:  2.793345055
Iteration:121 Total Cost:  2.719323315
Iteration:125 Total Cost:  2.642133121
Iteration:129 Total Cost:  2.546088176
Iteration:133 Total Cost:  2.442764104
Iteration:137 Total Cost:  2.359878313
Iteration:141 Total Cost:  2.264131285
Iteration:145 Total Cost:  2.197359196
Iteration:149 Total Cost:  2.124006654
Iteration:153 Total Cost:  2.061539153
Iteration:157 Total Cost:  2.006628213
Iteration:161 Total Cost:  1.960589669
Iteration:165 Total Cost:  1.916704256
Iteration:169 Total Cost:  1.858566790
Iteration:173 Total Cost:  1.829536923
Iteration:177 Total Cost:  1.783855772
Iteration:181 Total Cost:  1.764610816
Iteration:185 Total Cost:  1.730376678
Iteration:189 Total Cost:  1.708218082
Iteration:193 Total Cost:  1.698908234
Iteration:197 Total Cost:  1.673788192
Finished!
Accuracy of the predictions: : 0.971329

C:\Users\Arman\Desktop\COMP 4905A - Honors Project\Project>
```

Figure 8 Initial Hand Classifier accuracy rate

We then added an extra hidden layer and increased the number of nodes in each hidden layer to 300. This slowed down the training process and even displayed a message stating that the weight tensors have allocated more than 10% of system memory. The prediction accuracy did go up to 99.245 % and the cost value went down to 0.527 (see Figure 9).



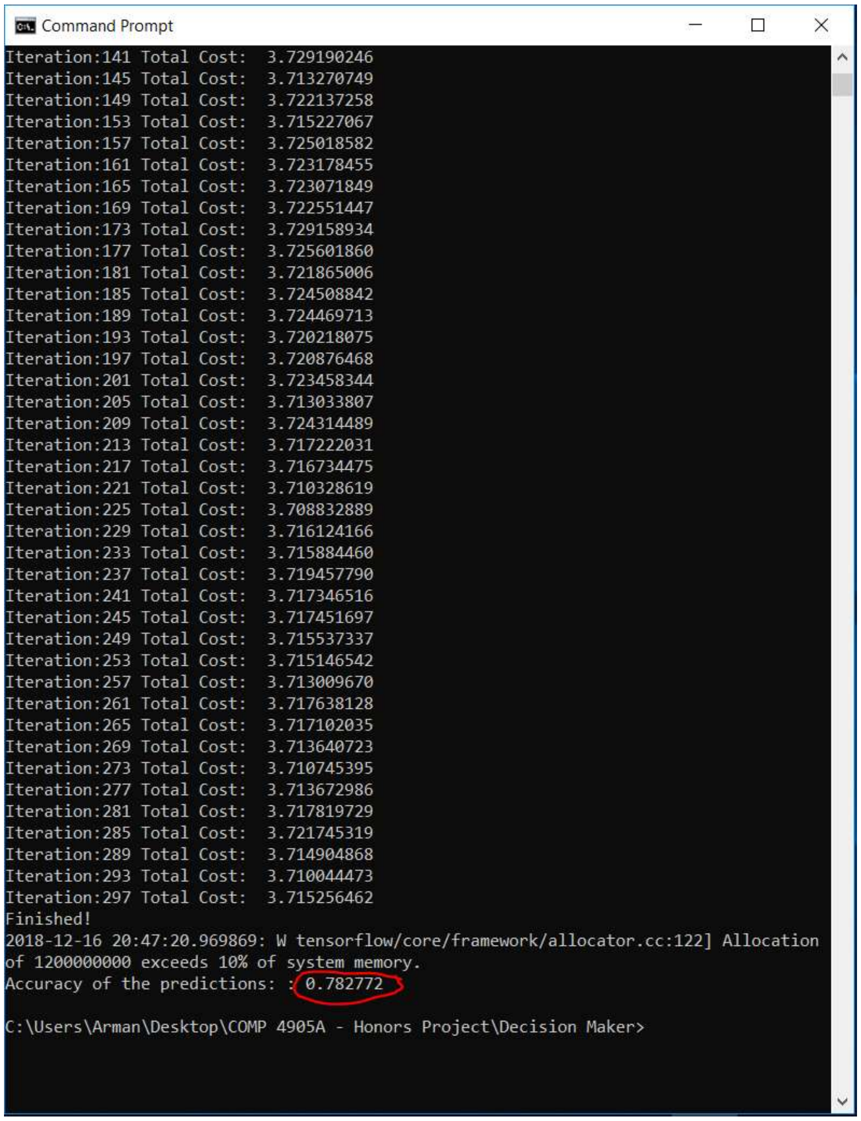
```
Iteration:53 Total Cost:  5.099697249
Iteration:57 Total Cost:  5.064737467
Iteration:61 Total Cost:  5.035379957
Iteration:65 Total Cost:  4.985336308
Iteration:69 Total Cost:  4.935466236
Iteration:73 Total Cost:  4.878035192
Iteration:77 Total Cost:  4.809936417
Iteration:81 Total Cost:  4.722865131
Iteration:85 Total Cost:  4.608426433
Iteration:89 Total Cost:  4.453257441
Iteration:93 Total Cost:  4.232911255
Iteration:97 Total Cost:  3.953795400
Iteration:101 Total Cost:  3.521751759
Iteration:105 Total Cost:  2.907388448
Iteration:109 Total Cost:  2.043571344
Iteration:113 Total Cost:  1.432908549
Iteration:117 Total Cost:  1.200009064
Iteration:121 Total Cost:  0.983693486
Iteration:125 Total Cost:  0.844569482
Iteration:129 Total Cost:  0.701819894
Iteration:133 Total Cost:  0.659997704
Iteration:137 Total Cost:  0.703743150
Iteration:141 Total Cost:  0.585122262
Iteration:145 Total Cost:  0.561974365
Iteration:149 Total Cost:  0.552885544
Iteration:153 Total Cost:  0.553211429
Iteration:157 Total Cost:  0.542843336
Iteration:161 Total Cost:  0.533689156
Iteration:165 Total Cost:  0.533556392
Iteration:169 Total Cost:  0.532793667
Iteration:173 Total Cost:  0.536360212
Iteration:177 Total Cost:  0.516783261
Iteration:181 Total Cost:  0.517987761
Iteration:185 Total Cost:  0.536127237
Iteration:189 Total Cost:  0.538947691
Iteration:193 Total Cost:  0.519467130
Iteration:197 Total Cost:  0.527937678
Finished!
2018-12-16 18:57:18.963900: W tensorflow/core/framework/allocator.cc:122] Allocation of 1200000000 exceeds 10% of system memory.
2018-12-16 18:57:19.834407: W tensorflow/core/framework/allocator.cc:122] Allocation of 1200000000 exceeds 10% of system memory.
Accuracy of the predictions: : 0.99245

C:\Users\Arman\Desktop\Honours Project\Neural Networks>
```

Figure 9 Outcome of the new Hand Classifying Network

From this we can conclude that the network was improved by increasing the number of hidden layers. We may also conclude that the data that was provided by the UCI Machine learning Repository is valid.

5.2.2 Game predictor

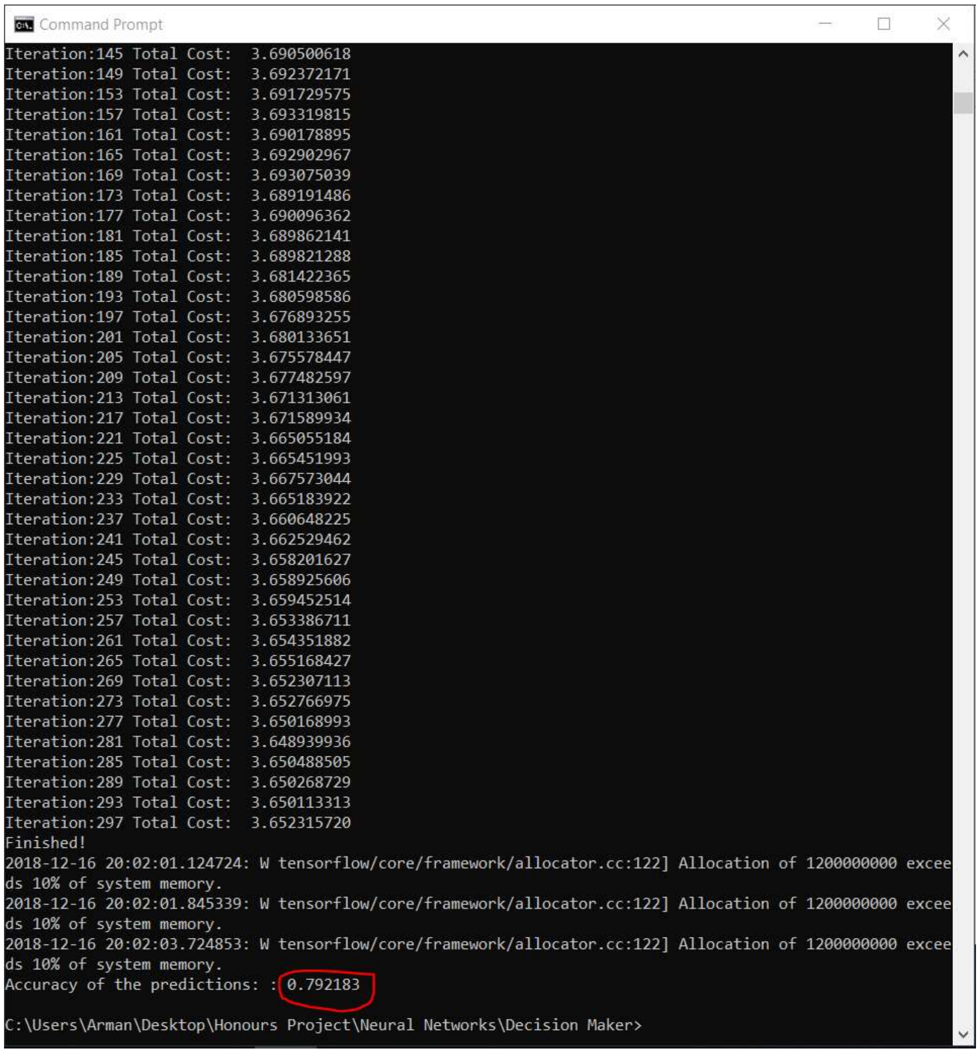
Using the data that was generated by the Java poker game generator we initially trained our game predicting neural network for 300 iterations at 2.5% learning rate. The neural network structure had one hidden layer consisting of 200 nodes. This neural network learned to predict games at approximately 78.27% with total error value of 3.715 (see Figure 10).



```
Iteration:141 Total Cost:  3.729190246
Iteration:145 Total Cost:  3.713270749
Iteration:149 Total Cost:  3.722137258
Iteration:153 Total Cost:  3.715227067
Iteration:157 Total Cost:  3.725018582
Iteration:161 Total Cost:  3.723178455
Iteration:165 Total Cost:  3.723071849
Iteration:169 Total Cost:  3.722551447
Iteration:173 Total Cost:  3.729158934
Iteration:177 Total Cost:  3.725601860
Iteration:181 Total Cost:  3.721865006
Iteration:185 Total Cost:  3.724508842
Iteration:189 Total Cost:  3.724469713
Iteration:193 Total Cost:  3.720218075
Iteration:197 Total Cost:  3.720876468
Iteration:201 Total Cost:  3.723458344
Iteration:205 Total Cost:  3.713033807
Iteration:209 Total Cost:  3.724314489
Iteration:213 Total Cost:  3.717222031
Iteration:217 Total Cost:  3.716734475
Iteration:221 Total Cost:  3.710328619
Iteration:225 Total Cost:  3.708832889
Iteration:229 Total Cost:  3.716124166
Iteration:233 Total Cost:  3.715884460
Iteration:237 Total Cost:  3.719457790
Iteration:241 Total Cost:  3.717346516
Iteration:245 Total Cost:  3.717451697
Iteration:249 Total Cost:  3.715537337
Iteration:253 Total Cost:  3.715146542
Iteration:257 Total Cost:  3.713009670
Iteration:261 Total Cost:  3.717638128
Iteration:265 Total Cost:  3.717102035
Iteration:269 Total Cost:  3.713640723
Iteration:273 Total Cost:  3.710745395
Iteration:277 Total Cost:  3.713672986
Iteration:281 Total Cost:  3.717819729
Iteration:285 Total Cost:  3.721745319
Iteration:289 Total Cost:  3.714904868
Iteration:293 Total Cost:  3.710044473
Iteration:297 Total Cost:  3.715256462
Finished!
2018-12-16 20:47:20.969869: W tensorflow/core/framework/allocator.cc:122] Allocation
of 1200000000 exceeds 10% of system memory.
Accuracy of the predictions:  0.782772

C:\Users\Arman\Desktop\COMP 4905A - Honors Project\Decision Maker>
```

Figure 10 Initial Game Predicting neural network

Deciding to take the same approach as we did with the hand classifying network, two extra hidden layers were added. Just like for the hand classifying network, the training process slowed down, with the prompt once again stating that over 10% of system memory were allocated by each weight tensor. But unlike the hand classifying network the accuracy prediction only improved by one percent. The newer neural network learned to predict games at approximately 79.21% with total error value of 3.65 (see Figure 11), increasing by only 1 percent from its predecessor.



```
Iteration:145 Total Cost:  3.690500618
Iteration:149 Total Cost:  3.692372171
Iteration:153 Total Cost:  3.691729575
Iteration:157 Total Cost:  3.693319815
Iteration:161 Total Cost:  3.690178895
Iteration:165 Total Cost:  3.692902967
Iteration:169 Total Cost:  3.693075039
Iteration:173 Total Cost:  3.689191486
Iteration:177 Total Cost:  3.690096362
Iteration:181 Total Cost:  3.689862141
Iteration:185 Total Cost:  3.689821288
Iteration:189 Total Cost:  3.681422365
Iteration:193 Total Cost:  3.680598586
Iteration:197 Total Cost:  3.676893255
Iteration:201 Total Cost:  3.680133651
Iteration:205 Total Cost:  3.675578447
Iteration:209 Total Cost:  3.677482597
Iteration:213 Total Cost:  3.671313061
Iteration:217 Total Cost:  3.671589934
Iteration:221 Total Cost:  3.665055184
Iteration:225 Total Cost:  3.665451993
Iteration:229 Total Cost:  3.667573044
Iteration:233 Total Cost:  3.665183922
Iteration:237 Total Cost:  3.660648225
Iteration:241 Total Cost:  3.662529462
Iteration:245 Total Cost:  3.658201627
Iteration:249 Total Cost:  3.658925606
Iteration:253 Total Cost:  3.659452514
Iteration:257 Total Cost:  3.653386711
Iteration:261 Total Cost:  3.654351882
Iteration:265 Total Cost:  3.655168427
Iteration:269 Total Cost:  3.652307113
Iteration:273 Total Cost:  3.652766975
Iteration:277 Total Cost:  3.650168993
Iteration:281 Total Cost:  3.648939936
Iteration:285 Total Cost:  3.650488505
Iteration:289 Total Cost:  3.650268729
Iteration:293 Total Cost:  3.650113313
Iteration:297 Total Cost:  3.652315720
Finished!
2018-12-16 20:02:01.124724: W tensorflow/core/framework/allocator.cc:122] Allocation of 1200000000 excee
ds 10% of system memory.
2018-12-16 20:02:01.845339: W tensorflow/core/framework/allocator.cc:122] Allocation of 1200000000 excee
ds 10% of system memory.
2018-12-16 20:02:03.724853: W tensorflow/core/framework/allocator.cc:122] Allocation of 1200000000 excee
ds 10% of system memory.
Accuracy of the predictions:  : 0.792183

C:\Users\Arman\Desktop\Honours Project\Neural Networks\Decision Maker>
```

Figure 11 Prediction Accuracy of the new Game Predicting Network

From this result we can conclude that the data that was produced by the game generator is not as valid as the data that was acquired from UCI. The problem with the generated data is that the value for each hand is abstract. An ace in a high card hand is valued the same as a king in a high card hand.

6 Future Implementations

The neural networks built for this project were designed for a simplified version of a poker game. It does not take into consideration of betting sizes, checking, or playing against more than two players. Future implementation can be done for more complex poker games. Reinforcement learning and Decision trees can be built to account for the human factor and limitless betting.

To improve the current game prediction neural network the data generator can be modified to output a score for each hand. In the current version each hand type has the same value, which makes the data not as valid since a pair of kings would beat a pair of jacks. Giving a specific score to each hand would validate our data more.

References

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning: A Textbook*. Cham: Springer Nature.

3Blue1Brown. (2017, October 05). Retrieved December 17, 2018, from https://www.youtube.com/watch?v=aircAruvnKk

Cattral, R. and Oppacher, F. (2007). UCI Machine Learning Repository [https://archive.ics.uci.edu/ml/datasets/Poker+Hand]. Ottawa, ON: Carleton University, Department of Computer Science

J. C. (2018, August 2). *Classification-Neural-Network*[Scholarly project]. Retrieved October 3, 2018, from https://github.com/julianClayton/Classification-Neural-Network

Lang, S. (2011). *Linear algebra*. New York: Springer.

Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M., Bowling, M. (2017). DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. *Science,356*(6337), 508-513. doi:10.1126/science.aam6960

Patterson, J., & Gibson, A. (2017). *Deep learning: A practitioner's approach*. Sebastopol (CA): O'Reilly.

Shiffman, D. (2012). *The nature of code*. Lexington.

Simmons, B., BS. (2017, July 19). Combination Formula. Retrieved December 3, 2018, from http://www.mathwords.com/c/combination_formula.htm

Wansbrough, W. D. (1912). *The ABC of the differential calculus*. Manchester: Technical Publishing.

Zill, D. G., & Wright, W. S. (2011). *Multivariable calculus*. Sudbury, MA: Jones and Bartlett.