UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Faculty of Engineering, Built Environment and Information Technology

EAI 320

Report: Practical 3

Artificial neural network

Name and Surname	Student Number	Signature
H. van der Westhuizen	18141235	[illegible]

Contents

List of Figures

1 Introduction

This report will discuss the implementation and results gathered by a Artificial Neural Network (ANN). The ANN is designed to beat a group of specialised bots as well as the genetic algorithm from practical 2 by training against a large CSV data set. The ANN is programmed on python 3 and should make use of common ANN algorithms such as gradient decent, feed-forward propagation and backward propagation.

1.1 Aim of the Artificial neural network

The ANN has 1 main goal and 5 desirable peripheral goals. The main aim of the ANN is to win as many matches as possible against the other specialised bots. The other 5 sub goals are: the ANN should reach a specified fitness within a reasonable amount of time; the ANN should also not produce too much over head; win as many rounds as possible; determine the best values for the synaptic weights and determine a optimal topology.

1.2 Artificial neural network requirements

The ANN must make use of common ANN development and training algorithms. Techniques such gradient decent, feed-forward and back propagation must be utilised when determining the best values for the weights. The ANN weight values must be determined by a Python program that uses the CSV data set.

1.3 Expected outcome

The ANN is expected to produce the values for the weights connecting the node layers with a maximum average error of 0.1 within a reasonable amount of time. The ANN is expected to win more than 92% of matches against the pool of specialised bots.

2 Theoretical background

2.1 What is a Artificial neural network

Neural networks are learning models that are inspired by the brain. Like organic brains an ANN has neurons that are connected to each other through input dendrites and output axons. The neurons are interconnected with synapses (weights). These synapses cause the activation of one neuron to effect the activation level in the other neuron (connected to the axon).

The goal of an ANN is to receive inputs from the input layer, feed the activation level of each neuron through the synapses to the next neuron layer. The process is repeated until the neurons in the output layer are assigned a predicted value.

Once the values for the output neurons have been determined they are compared to pre-recorded finding to see the level of error within the ANN. This error is then used to adjust the weights connecting the neurons to produce better results in the future.

2.2 Activation function (Sigmoid function)

When developing an ANN the values of the neurons should be bounded in some form. The sigmoid function allows the neuron values to be bounded between 0 and 1. The value of the neurons are bounded to ensure a neuron does not diverge and tend to ∞. The activation function also helps to ensure that local minimum areas are avoided.

$$S(x) = \frac{1}{1 + e^{-z}} \tag{1}$$

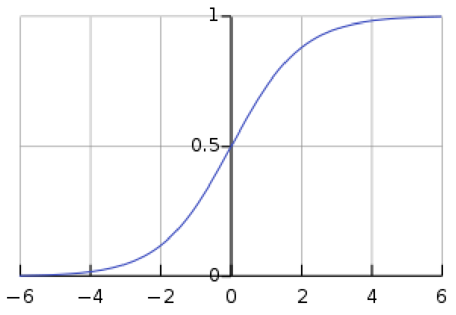

Figure 1: Sigmoid function

2.3 Gradient decent

Gradient descent is and optimisation algorithm. It's aim in a ANN is to minimise the the error function by adjusting the the weight values. This ensures the error function is iteratively moved in the direction of steepest descent, effectively using the negative gradient.

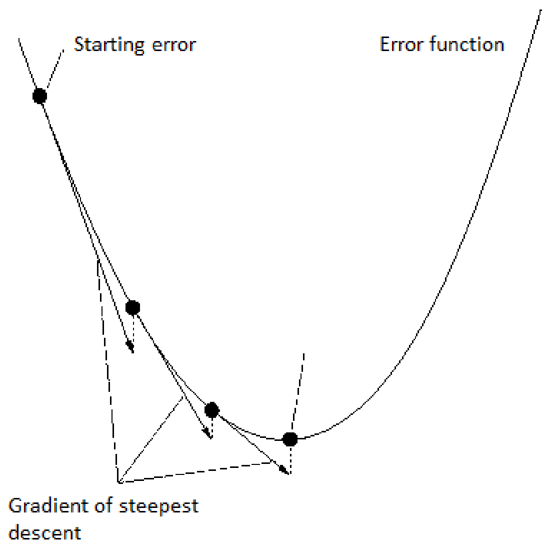

Figure 2: Gradient descent illustration

As figure 2 suggests the negative gradient adjustment decreases the error quickly in the beginning and decreases in level of adjustment as the error approaches the global minimum.

3 Methodology

The program uses basic ANN algorithms such as feed-forward, error functions, gradient decent, sigmoid functions and back propagation to determine the correct weight values, as well as determining the smallest average error value.

3.1 Overall program structure

The program is created with modulation and ease of manipulation in mind. The parameters that the ANN uses should be changed easily to test different variable configurations.The program is also split into 4 main parts: neuron and weight initialisation; error function and sigmoid declaration; feed-forward propagation and back propagation.

Firstly the neurons within the input, hidden and output layers are declared. The input neurons are initialised using the CSV's history and the output layer's neurons are initialised with the CSV's responses. After the initial neuron values are determined the weights connecting the neurons within the different layers are given random values between 0 and 1.

Once the initialisation stage is complete the back propagation stage begins. During this stage the input layers are propagated forward, one CSV history at a time. Once the input neurons propagate into the output layer the determined feed-forward values are compared with the CSV's response, stored in the output layer. The difference between the feed-forward output values and true response values are then fed into an error function. The error function is then used to determine the required level of weight value adjustment. This adjustment is done through the use of a gradient descent optimisation function. This process is then repeated until the difference between the feed forward values and true response values are sufficiently small e.g 0.01 or 0.1 average error across the total iterations.

3.2 Network topology

The final ANN topology consists out of 4 neuron layers: one input later, two hidden layers and one output layer. The input layer consists of 13 neurons (12 input neurons and 1 bias neuron); both hidden layers have 13 neurons (12 propagating neurons and 1 bias neuron). The final output layer consists out of 3 neurons. Generally it is recommended to choose a hidden layer size that is greater than the input layer to decrease the risk of compression.Although the larger then hidden layer become to larger the overhead. All in all 13 total neurons per hidden layer is sufficient for the one hot input layer system used.

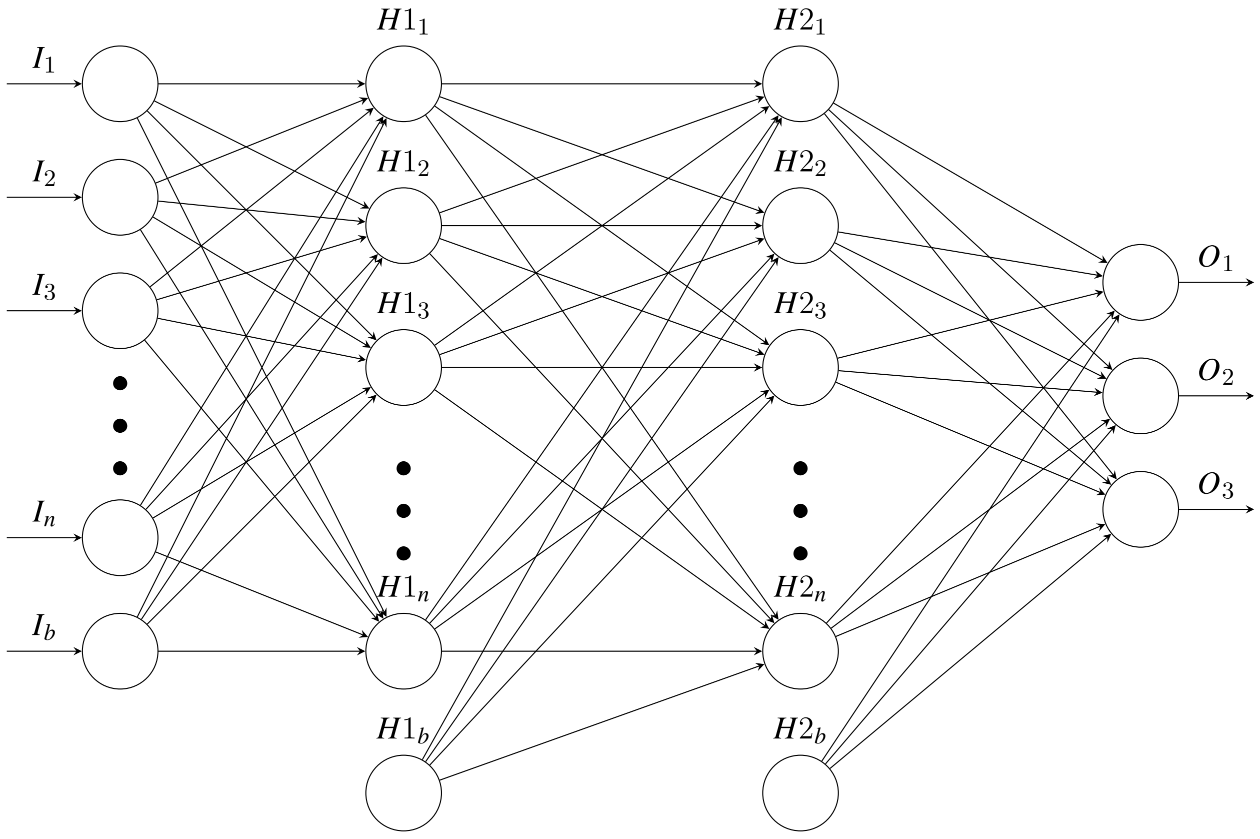

Figure 3: Artificial neural network topology

3.3 Neuron and synapse initialisation

The input layer is initialised by using a one hot encoding method. The input layer receives the history of each CSV entry in the form [X_{t-2},Y_{t-2},X_{t-1},Y_{t-1}], whereas the X is the agents play and Y is the opponents play. The history is then converted into a one hot encoding method. For example RSPS is converted into [1,0,0,0,0,1,0,1,0,0,0,1]. R is represented by [1,0,0], P by [0,1,0] and S [0,0,1]. Finally each entry is appended 1 to account for the positive bias. This is repeated until all the CSV history entries are encoded and appended to the Input layer array whereas n = 1 million in this case:

$$InputLayer_{(12,n)} = \begin{bmatrix} I[0]_1 & I[0]_2 & \cdots & I[0]_n \\ I[1]_1 & I[1]_2 & \cdots & I[1]_n \\ I[2]_1 & I[2]_2 & \cdots & I[2]_n \\ \vdots & \vdots & \ddots & \vdots \\ I[12]_1 & I[12]_2 & \cdots & I[12]_n \end{bmatrix}$$

The output layer is recorded and encoded in a similar fashion. Each CSV response is converted and appended to the array:

$$OutputLayer(3,n) = \begin{bmatrix} O[0]_1 & O[0]_2 & \cdots & O[0]_n \\ O[1]_1 & O[1]_2 & \cdots & O[1]_n \\ O[2]_1 & O[2]_2 & \cdots & O[2]_n \end{bmatrix}$$

The hidden layers are randomly given a value between 0 and 1 except for the 13^{th} node that is assigned the bias value of 1.

$$HiddenLayer = \begin{bmatrix} H_1, & H_2, & H_3, & ... & ,H_{12}, & 1 \end{bmatrix}$$

Lastly the connection between the input layer and first hidden layer as well as the connections between the two hidden layers consists out of 156 weights, whereas the connection between the output and second hidden layer consists out of 39 weights. Each weight is initialised with a random value between 0 and 1.

Weight layer 1 and 2:

$$WeightLayer_{(12,11)} = \begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,11} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,11} \\ \vdots & \vdots & \ddots & \vdots \\ W_{12,0} & W_{12,1} & \cdots & W_{12,11} \end{bmatrix}$$

Weight layer 3:

$$WeightLayer_{(13,3)} = \begin{bmatrix} W_{0,0} & W_{0,1} & \cdots & W_{0,2} \\ W_{1,0} & W_{1,1} & \cdots & W_{1,2} \\ \vdots & \vdots & \ddots & \vdots \\ W_{12,0} & W_{12,1} & \cdots & W_{12,2} \end{bmatrix}$$

3.4 Activation function

The sigmoid function is the activation function that ensures the values of the neurons stay between 0 and 1 as forward propagation takes place. This is done by substituting the dot product of the current layer array with the weight layer (connected at the axon end) within the sigmoid function presented in in Equation 1. This ensure each neuron has some effect on the system and no single node can collapse the ANN by diverging.

3.5 Error function

The error function calculates the relative difference between the output values that are pre-recorded (true output values) and the output values determined by the ANN's feed forward algorithm. The function is given below:

$$Error = \sum_{i=1}^{3} \frac{(O_T[i] - O[i])^2}{2} \tag{2}$$

This error value is used by the back propagation algorithm to determine the level of weight adjustment needed to reach the minimum error efficiently.

3.6 Feed-forward propagation

In essence feed forward propagation is the act of determining the activation levels of each node in the proceeding layer; due to the dot product of the current layer of neurons and the connecting weights between the layers. The result of the dot product is then substituted into the activation function to ensure the neurons fall within the required boundaries. The 4 layer ANN follows the following procedure:

$$Z1 = Input\ Layer \cdot Weight\ Layer_1$$
$$Hidden\ Layer_1 = Sigmoid(Z1)$$
$$Z2 = Hidden\ Layer_1 \cdot Weight\ Layer_2$$
$$Hidden\ Layer_2 = Sigmoid(Z2)$$
$$Z3 = Hidden\ Layer_2 \cdot Weight\ Layer_3$$
$$Output\ Layer = Sigmoid(Z3)$$

Once the output layer's neuron values are determined the error function can be calculated. In the case of the RPS game the output value determined by the feed forward function is played as an output response. Also as the procedure indicated, the input layer has an effect on the output layer due to the fact that the neurons values in the input layer are propagated forward in each function calculation.

3.7 Gradient decent and derivative calculation

When determining the gradient decent the effect of each layer as well as neuron on the output needs to be known. This can be done by using linear algebra, the derivative of each neuron and weight value can be determined.In this case the effect of each weight on the output is required since they need to be change to reduce the overall error within the network. The derivative equations can be represented as follows.

$$\frac{dE}{dW_3} = \frac{dE}{dO}\frac{dO}{dZ_3}\frac{dZ_3}{dW_3}$$

$$\frac{dE}{dW_2} = \frac{dE}{dO}\frac{dO}{dZ_3}\frac{dZ_3}{dH_2}\frac{dH_2}{dZ_2}\frac{dZ_2}{dW_2}$$

$$\frac{dE}{dW_1} = \frac{dE}{dO}\frac{dO}{dZ_3}\frac{dZ_3}{dH_2}\frac{dH_2}{dZ_2}\frac{dZ_2}{dH_1}\frac{dH_1}{dZ_1}\frac{dZ_1}{dW_1}$$

The procedure of back propagation is prevalent in the equations above, since each gradient calculation shows the dept of effect a value has on the error. When doing back propagation 2 additional parameters are required an Array that holds the error (E as seen above) of each output neuron and a derivative of the sigmoid function. These parameters are determined by:

$$\frac{dS}{dz} = \frac{e^{-z}}{(1+e^{-z})^2} \tag{3}$$

$$E = \frac{(O_T[0] - O[0])^2}{2} + \frac{(O_T[1] - O[1])^2}{2} + \frac{(O_T[2] - O[2])^2}{2} \tag{4}$$

The procedure for determining the negative gradient is linear as seen above. This allows the ANN to change the weight values through basic array multiplication and transposition. The 4 layer ANN implementation follows the following procedure.

Determining the new weights connecting the second hidden layer and and the output layer, each value from the output layer is substituted into the sigmoid derivative function, this forms a new array that is multiplied with the error array given by Equation 4 this subsequently forms a new array called derivativeYSig. After this the dot product of derivativeYSig and hidden layer 2 is found. Finally the array determined by the dot product is transposed re-orientated to ensure the new weight values can be substitutes back into the ANN.

Since the system follows the rules of linear algebra the weights for the upper levels are determined in the same way.

3.8 Back prorogation

Back propagation is an ANN's algorithm that allows it to change the weight values between the layer by increasing or decrease the values of the weight determined by the amount calculated through gradient decent. The system works as follows. Feed-forwarded propagation takes place, the error array is created by utilising the error function and gradient decent. Once the gradient decent algorithm has determined the best adjustment for each weight the original weights are altered by adding the adjustment. This adjustment can be multiplied by a constant n. The larger n is the greater the amount of change the weight will receive. This process is repeated for each CSV training entry until the average error has been reduced to a desired amount or enough iterations (CSV training samples) have been processed.

The goal of back propagation is to improve the accuracy of the feed forward output value prediction.

4 Results

When measuring the results there are a few parameters of interest. Firstly the effect of the n constant on the rate of convergence; secondly the effect on the speed of convergence due to the number of layers as well as the number of neurons per hidden layer and the win percentage of the ANN.

4.1 Analysis of Python plots

During the testing of the ANN many interesting characteristics of a typical ANN became evident. These characteristics effected many aspects of the program, some of these affects altered the speed of convergence, increased the programs running over head,caused long term stagnation and even resulted in non-convergence.

In order to show how different variable effected the ANN a benchmark needed to be set that will act as the control in terms of variable values. After many iterations it became evident that the best benchmark variable values are:

n constant = 3
Termination error = 0.1
Number of hidden layers = 3
Number of neurons per hidden layer = 12 each

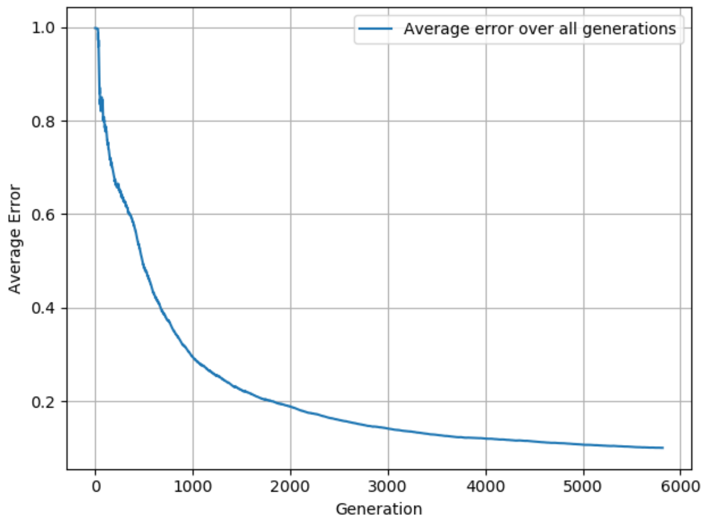

Figure 4: Bench mark graph that plots the Average error against the current generation amount

As Figure 4 indicate the average error reduces to 0.1 within about 5800 generations. This rate of convergence is extremely fast since only 6% of the CSV is needed, this might be an issue if the CSV is not populated randomly and certain areas have bias history and response pairs. But with the configurations provided above the convergence is very quick, it causes average overhead which is mitigated by the speed of convergence and the *lin* function nature of the graph indicates the n value does not over shoot the global minimum. When analysing the affect each variable should be altered individually keeping the rest of the configurations the same.

When moving in order the affect of the n constant should be calculated and graphed.

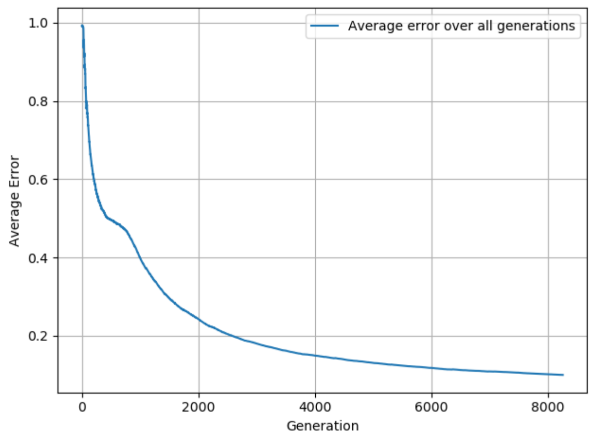

Figure 5: Average error vs generation count when n = 1

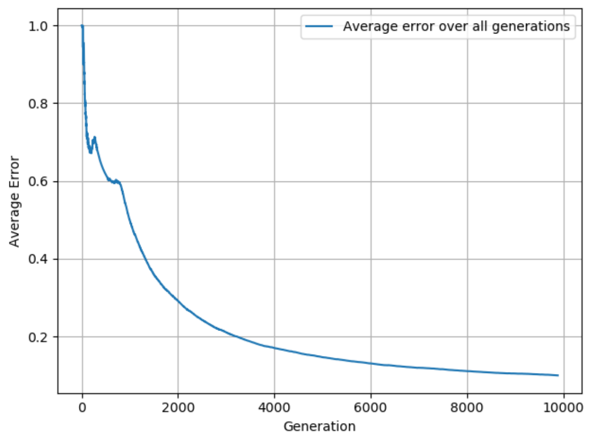

Figure 6: Average error vs generation count when n = 6

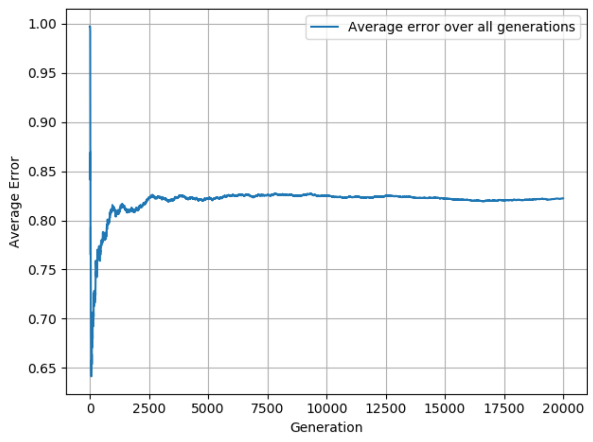

Figure 7: Average error vs generation count when n = 9

When consulting figures 5, 6 and 7 the n value has two main effects. Firstly n can cause the average error to increase instead of decrease when the derivative is added to the weight. This is caused when the new weight overshoots the global minimum. If n is small the likely hood of overshoot is mitigated as seen in Figure 5. Whereas Figure 7 shows that the change to the weights can be so drastic it does not even converge and gets stuck in a local minimum. The speed of convergence is decreases as n decreases since the steps to the global minimum is more precise.

Next the termination error can be increased or decreased. The effect of this is obvious the higher the error tolerance the faster the convergence and the lower the error tolerance the slower the convergence. Using the CSV data set the global minimum lies at about an average error of 0.05 if the terminate error is below this it will not converge.

Although the amount of hidden layers the ANN has is not as obvious:

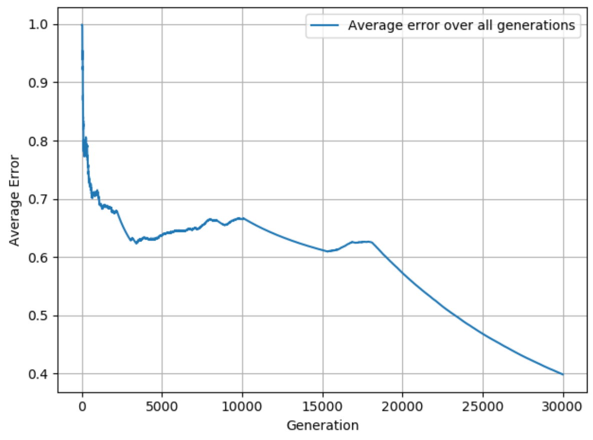

Figure 8: Average error vs generation count when Layers = 1

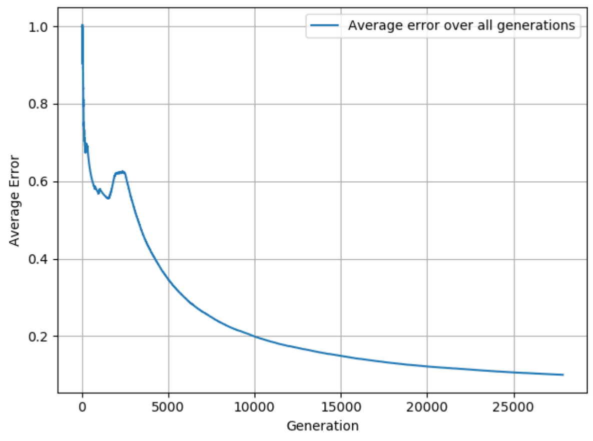

Figure 9: Average error vs generation count when Layers = 3

When examining Figure 8 and 9 it is clear to see that 2 layers allow for faster convergence. When considering a ANN with a single hidden layer the system can not make fine adjustments easily, thus if the weight values are changed they might overcompensate for an error. The level of convergence must also increase if only one hidden layer is present, since less layers mean greater abstraction and lower level of fine tuning. This is indicated by the average error of 0.4 at 30000 generations. A ANN with 3 hidden layers should allow for a slightly better global minimum but the fact that the inputs are not complex causes a feedback loop from the layers. This creates noise (as seen by the spike at the 5000 generation mark) within the ANN ultimately causing slower convergence.

Lastly the number of neurons per hidden layer must be considered.

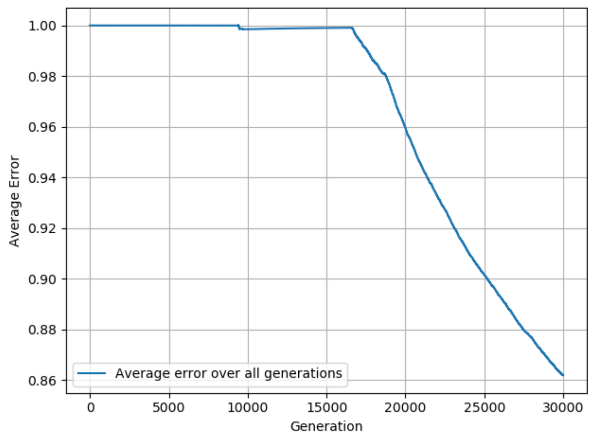

Figure 10: Average error vs generation count when Neurons per hidden layer = 6

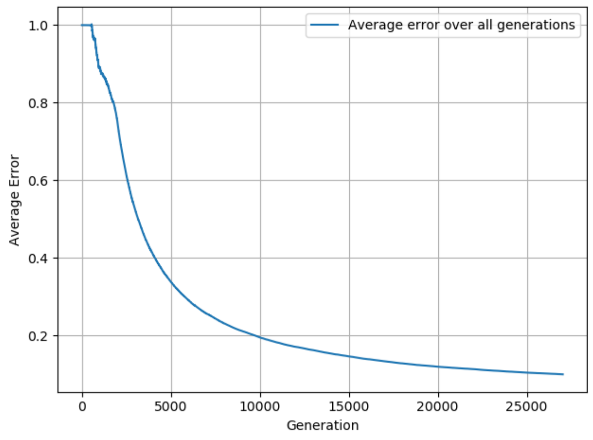

Figure 11: Average error vs generation count when Layers = 16

When the hidden layer is smaller than the input layer compression takes place, this compression can corrupt the ANN inputs and make it difficult to create strong relationships between neurons. If the compression is to much it causes the system to diverge as seen by Figure 10. When the hidden layers are larger than the input layer they allow for greater fine tuning of neuron relations. Overall more neurons per hidden layer will cause greater overhead and possible a smaller global minimum even though convergence is slower.

5 Match results

```
D:\EAIProjects\Prac3>rpsrunner.py -m 100 -r 1000 Task2.py
    only_rock.py,only_paper.py,only_scissors.py,beat_previous
    .py,beat_common.py
Pool 1: 1 bots loaded
Pool 2: 5 bots loaded
Playing 100 matches per pairing.
Running matches in 8 threads
500 matches run
total run time: 16.51 seconds

only_rock.py: won 0.0% of matches (0 of 100)
    won 0.0% of rounds (0 of 100000)
    avg score -1000.0, net score -100000.0

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.0% of rounds (0 of 100000)
    avg score -999.0, net score -99900.0

only_scissors.py: won 0.0% of matches (0 of 100)
    won 0.1% of rounds (100 of 100000)
    avg score -998.0, net score -99800.0

beat_previous.py: won 0.0% of matches (0 of 100)
    won 30.5% of rounds (30500 of 100000)
    avg score -389.2, net score -38922.0

beat_common.py: won 0.0% of matches (0 of 100)
    won 20.0% of rounds (20000 of 100000)
    avg score -600.0, net score -60000.0

Task2.py: won 87.8% of matches (439 of 500)
    won 89.8% of rounds (449222 of 500000)
    avg score 797.2, net score 398622.0

D:\EAIProjects\Prac3>rpsrunner.py -m 100 -r 1000 Task3.py
    only_rock.py,only_paper.py,only_scissors.py,beat_previous
    .py,beat_common.py
Pool 1: 1 bots loaded
Pool 2: 5 bots loaded
Playing 100 matches per pairing.
Running matches in 8 threads
500 matches run
total run time: 65.44 seconds

only_rock.py: won 0.0% of matches (0 of 100)
```

```
    won 0.0% of rounds (0 of 100000)
    avg score -1000.0, net score -100000.0

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.0% of rounds (0 of 100000)
    avg score -999.0, net score -99900.0

only_scissors.py: won 0.0% of matches (0 of 100)
    won 0.1% of rounds (100 of 100000)
    avg score -998.0, net score -99800.0

beat_previous.py: won 0.0% of matches (0 of 100)
    won 1.1% of rounds (1118 of 100000)
    avg score -976.9, net score -97694.0

beat_common.py: won 0.0% of matches (0 of 100)
    won 20.0% of rounds (20000 of 100000)
    avg score -600.0, net score -60000.0

Task3.py: won 100.0% of matches (500 of 500)
    won 95.7% of rounds (478612 of 500000)
    avg score 914.8, net score 457394.0

D:\EAIProjects\Prac3>rpsrunner.py -m 100 -r 1000 Task3.py
   agent.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 8 threads
100 matches run
total run time: 19.58 seconds

agent.py: won 0.0% of matches (0 of 100)
    won 21.9% of rounds (21943 of 100000)
    avg score -504.5, net score -50452.0

Task3.py: won 100.0% of matches (100 of 100)
    won 72.4% of rounds (72395 of 100000)
    avg score 504.5, net score 50452.0
```

The results above are gathered by playing against the five specified bot using the ANN trained by CSV file. Then the same bots are played against the ANN that trains online. The first ANN only won 87% of the matches whereas the smarter ANN that trains as it plays wins 100% of the matches. The online ANN also beats the genetic algorithm 100% of the time.Although if the amount of rounds are reduced per match the online ANN does worse (still better than the non adaptive one), this is due to the ANN needing several generations to fine tune the new weights needed to consistently beat the other bots. This is why the main aim of the ANN must be to converge to a small enough average error as fast as possible.

6 Discussion

Throughout the report the underlying workings of an Artificial Neural Network has been discussed and demonstrated. During the practical it became evident that the ANN greatly effected by various parameters.

The learning rate constant η increase the amount of generations needed to reduce the error value to a specified amount if it is small.Although when η is to large it will cause divergence since the original weight is adjusted to much ever generation, making small adjustment impossible.

The activation function allows the neuron values to be bounded and adds a non-linear component within the ANN. The sigmoid function bounds the neurons within 0 and 1 , while also allowing for fine adjustments.

Moreover, the amount of hidden layer and neurons per layer play a large part.Then the hidden layers are smaller than the input layer the data is compressed, this creates less depth and may allow for divergence. When their is only 1 hidden layer the system takes much longer to converge and may increase the global minimum that can be achieved. Whereas an ANN with 3 hidden layers create noise in less complex systems.

7 Conclusion

The report demonstrates the planning, design and implementation process when using an ANN algorithm to solve a problem. The result obtained from the graph as well as the match win rates ensure the ANN topology is sufficient. The weights gathered by doing back propagation with the CSV data set act as a good base line when playing against other bots.Although when the ANN did online back propagation to adjust the hard coded weight values, it improved greatly wining 100% of matches instead of 87%. As stated in the match results the main aim of the ANN must be to converge to a small enough average error as fast as possible. The practical goals have been achieved and a sufficient ANN has been programmed and implemented.

8 References

[1]Khan, R., 2020. Nothing But Numpy: Understanding Creating Neural Networks With Computational Graphs From Scratch - Kdnuggets. [online] KDnuggets. Available at: <https://www.kdnuggets.com/2019/08/numpy-neural-networks-computational-graphs.html> [Accessed 13 May 2020].

[2]Matplotlib.org. 2020. Sample Plots In Matplotlib — Matplotlib 3.2.1 Documentation. [online] Available at: <https://matplotlib.org/3.2.1/tutorials/introductory/sample$_{p}lots.html >$ $[Accessed 13 May 2020]$.

[3]MissingLink.ai. 2020. 7 Types Of Activation Functions In Neural Networks: How To Choose?. [online] Available at: <https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/> [Accessed 13 May 2020].

9 Appendix

Task 1 Code:

```python
# Author:          H. van der Westhuizen
# Date:            26 April 2020
# Revision:        12 May 2020
# Artificial neural network task 3

import numpy as np
import csv
import random
import matplotlib.pyplot as plt


beat = {"R": "P", "P": "S", "S": "R"}


def init_Layers_Weights():
    """This function initialised the weights and layer
       values"""
    # The hidden layers are initialised to 1. And the input
        and output layers are created
    for i in range(hiddenLayerSize+1):
        if i < 13:
            inputLayer0.append([])
            weight1.append([])
        weight2.append([])
        weight3.append([])
        hiddenLayer1.append(1)
        hiddenLayer2.append(1)
        z1.append(1)
        z2.append(1)
        for j in range(hiddenLayerSize):
            if i < 13:
                weight1[i].append(random.random())
            weight2[i].append(random.random())
        for k in range(3):
            weight3[i].append(random.random())
    for i in range(3):
        outputLayer3.append([])
        z3.append(1)
    with open("data1.csv") as csv_data:
        # Each entry from the CSV is read in one at a time.
        read_CSV = (csv.reader(csv_data, delimiter=','))
        overallPosition = 0
        for row in read_CSV:
            CSV_History = list(row[0])
            CSV_Response = list(row[1])[0]
            for x in outputLayer3:
                x.append(0)
            # The input history is encoded into a 1 hot encoding format.
            inputA = histCheck(CSV_History)
            for i in range(13):
                inputLayer0[i].append(inputA[i])
            if CSV_Response == 'R':
                outputLayer3[0][overallPosition] = 1
            elif CSV_Response == 'P':
                outputLayer3[1][overallPosition] = 1
            else:
                outputLayer3[2][overallPosition] = 1
            overallPosition = overallPosition + 1


def sigmoid(weight_node):
    """The dot product values of the current layer and proceeding weight array is substituted
    into the sigmoid function"""
    sigmoid_weight_Node = 1 / (1 + np.exp(-weight_node))
    return sigmoid_weight_Node


def sigmoidDerivative(sigValue):
    """The dot product values of the current layer and preceding weight array is substituted
        into the derivative of the sigmoid function"""
    sigDerivative = (np.exp(-sigValue)) / pow((1 + np.exp(-sigValue)), 2)
    return sigDerivative


def errorFunction(true_output):
    """This method calculates the error of the feed forward output value and true response value"""
    totalError = 0
    for i in range(3):
        totalError += pow((true_output[i] - feedforwardOutputValue[i]), 2)
    return totalError / 2


def feedForward(input_Entry):
    global feedforwardOutputValue
    global hiddenLayer1
    feedforwardOutputValue = []
    # The first hidden layer values are obtained by doing
    # the dot product of the input layer and proceeding weight layer
    z_x = np.dot(input_Entry, weight1)
    for i in range(hiddenLayerSize):
        z1[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer1[i] = sigmoid(z_x[i])
    # The second hidden layer values are obtained by doing
    # the dot product of the first hidden layer and proceeding weight layer
    z_x = np.dot(hiddenLayer1, weight2)
    for i in range(hiddenLayerSize):
        z2[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer2[i] = sigmoid(z_x[i])
    # The output layer values are obtained by doing
    # the dot product of the second hidden layer and proceeding weight layer
    z_x = np.dot(hiddenLayer2, weight3)
    for i in range(3):
        z3[i] = z_x[i]
        feedforwardOutputValue.append(sigmoid(z_x[i]))


def histCheck(hist):
    """The input from the oponent is processed here into
    1 hot encoding"""
    histA = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    y = 0
    for i in hist:
        if i == 'R':
            histA[y * 3] = 1
        elif i == 'P':
            histA[y * 3 + 1] = 1
        else:
            histA[y * 3 + 2] = 1
        y += 1
    return histA


def derivative(realInput, realOutput):
    derSigZ1 = []
    derSigZ2 = []
    derSigZ3 = []
    derivativeL1 = []
```

```python
    derivativeL2 = []
    derivativeL3 = []
    derivativeOutput = []
    derivativeYSig = []
    derivativeH1H2 = []
    # The two hidden layer and output layer values are substituted into
    # into the derivative of the sigmoid function
    for i in range(hiddenLayerSize + 1):
        derSigZ1.append(sigmoidDerivative(z1[i]))
        derSigZ2.append(sigmoidDerivative(z2[i]))
    for i in range(3):
        derSigZ3.append(sigmoidDerivative(z3[i]))
        derivativeOutput.append(-(realOutput[i] - feedforwardOutputValue[i]))  # An error array is created.
        # The output derivative is multiplied with the error array.
        derivativeYSig.append(derSigZ3[i] * derivativeOutput[i])
        # The dot product of the derivativeYSig and the second hidden layer is found
        derivativeL3.append(np.dot((derivativeYSig[i]), hiddenLayer2))
    weight3Change = np.transpose(derivativeL3)  # The dot product is transposed to ensure the weight array
    # is correctly orientated
    # The procedure is repeated all the way until the weights between the input layer and
    # first input layer is achieved.
    derivativeYSig = np.dot(weight3, np.transpose(derivativeYSig))
    derivativeYSig = np.transpose(derivativeYSig)
    for i in range(hiddenLayerSize):
        derivativeH1H2.append(derivativeYSig[i] * derSigZ2[i])
        derivativeL2.append(np.dot((derivativeH1H2[i]), hiddenLayer1))
    weight2Change = np.transpose(derivativeL2)
    derivativeH1H2 = np.dot(weight2, np.transpose(derivativeH1H2))
    derivativeH1H2 = np.transpose(derivativeYSig)
    for i in range(hiddenLayerSize):
        derivativeL1.append(np.dot((derSigZ1[i] * derivativeH1H2[i]), realInput))
    weight1Change = np.transpose(derivativeL1)
    return weight1Change, weight2Change, weight3Change
```

```


def BackPropagation():
    global weight1, weight2, weight3
    csvEntry = 0
    averageError = 1
    errorArray = []
    average = []
    counter = 0
    if input == "":
        while averageError > terminateError:
            counter= counter+1
            realInput = []
            realOutput = []
            for i in range(13):
                # The history of the match is gathered from CSV
                realInput.append(inputLayer0[i][csvEntry])
            for i in range(3):
                # The response of the opponent is gathered from CSV.
                realOutput.append(outputLayer3[i][csvEntry])
            feedForward(realInput)
            error = errorFunction(realOutput)
            errorArray.append(error)
            weight1Change, weight2Change, weight3Change = derivative(realInput, realOutput)
            # The weights are changed by subtracting the weight derivative from the original weight.
            weight1 = weight1 - constant * weight1Change
            weight2 = weight2 - constant * weight2Change
            weight3 = weight3 - constant * weight3Change
            csvEntry += 1
            averageError = 0
            # The average error is calculated by dividing the total error
            # of each generation by the current generation count
            for i in errorArray:
                averageError += i
            averageError = averageError / csvEntry
            average.append(averageError)
        return average
    else:
        feedForward(histCheck(history))
        if input == "R":
            realOutput = [1,0,0]
        elif input == "S":
            realOutput = [0, 1, 0]
        else:
            realOutput = [0, 0, 1]
        error = errorFunction(realOutput)
        errorArray.append(error)
        weight1Change, weight2Change, weight3Change = derivative(realInput, realOutput)
        weight1 = weight1 - constant * weight1Change
        weight2 = weight2 - constant * weight2Change
        weight3 = weight3 - constant * weight3Change

input = ""
if __name__ == "__main__":
    if input == "":
        # If the match starts the Neural Network is trained against the
        # csv data to establish the weights of the connections.
        smart = False
        terminateError = 0.1  # This value determines when the Average error of the NN
        # is small enough to give a accurate prediction.
        hiddenLayerSize = 12 # This value determines the number of nodes in the Hidden Layer.
        constant = 3 # This is a constant that determines the rate of gradient decent.
        # Initialise Layers 1 input layer 2 hidden layers and 1 output layer
        inputLayer0 = []
        hiddenLayer1 = []
        hiddenLayer2 = []
        outputLayer3 = []
        # Weights of the connections between layers
        weight1 = []  # Between Input Layer and Hidden Layer 1
        weight2 = []  # Between Hidden Layer 1 and Hidden Layer 2
        weight3 = []  # Between Hidden Layer 2 and Output Layer
        # Sigmoid value of each neuron in a given Layer
        z1 = []  # HiddenLayer 1 sigmoid value array
        z2 = []  # HiddenLayer 2 sigmoid value array
        z3 = []  # OutputLayer sigmoid value array
        # Feed forward values
        # The values of the RPS output layer calculated
            through feed forward.
        feedforwardOutputValue = []
        history = ["R","R","R","R"]  # The history is
            initialised
        init_Layers_Weights()  # THe Layers and weights of
            the NN needs to be created.
        BackPropagation()
        # x2 = np.linspace(1, len(y), len(y))
        # plt.plot(x2,y,label="Average error over all
            generations",color='black')
        # plt.xlabel('Generation')
        # plt.ylabel('Average Error')
        # plt.legend()
        # plt.show()
```

Task 2 Code:

```python
# Author:        H. van der Westhuizen
# Date:          26 April 2020
# Revision:      12 May 2020
# Artificial neural network task 3

import numpy as np

beat = {"R": "P", "P": "S", "S": "R"}


def init_Layers_Weights():
    """This function initialised the weights and layer values"""
    # The hidden layers are initialised to 1. And the input and output layers are created
    for i in range(hiddenLayerSize + 1):
        if i < 13:
            inputLayer0.append([])
        hiddenLayer1.append(1)
        hiddenLayer2.append(1)
        z1.append(1)
        z2.append(1)
    for i in range(3):
        outputLayer3.append([])
        z3.append(1)


def sigmoid(weight_node):
    """The dot product values of the current layer and proceeding weight array is substituted
    into the sigmoid function"""
    sigmoid_weight_Node = 1 / (1 + np.exp(-weight_node))
    return sigmoid_weight_Node


def errorFunction(true_output):
    """This method calculates the error of the feed forward output value and true response value"""
    totalError = 0
    for i in range(3):
        totalError += pow((true_output[i] - feedforwardOutputValue[i]), 2)
    return totalError / 2


def feedForward(input_Entry):
    global feedforwardOutputValue
    global hiddenLayer1
    feedforwardOutputValue = []
    # The first hidden layer values are obtained by doing
    # the dot product of the input layer and proceeding weight layer
    z_x = np.dot(input_Entry, weight1)
    for i in range(hiddenLayerSize):
        z1[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer1[i] = sigmoid(z_x[i])
    # The second hidden layer values are obtained by doing
    # the dot product of the first hidden layer and proceeding weight layer
    z_x = np.dot(hiddenLayer1, weight2)
    for i in range(hiddenLayerSize):
        z2[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer2[i] = sigmoid(z_x[i])
    # The output layer values are obtained by doing
    # the dot product of the second hidden layer and proceeding weight layer
    z_x = np.dot(hiddenLayer2, weight3)
    for i in range(3):
        z3[i] = z_x[i]
        feedforwardOutputValue.append(sigmoid(z_x[i]))


def histCheck(hist):
    """The input from the oponent is processed here into
    1 hot encoding"""
    histA = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    y = 0
    for i in hist:
        if i == 'R':
            histA[y * 3] = 1
        elif i == 'P':
            histA[y * 3 + 1] = 1
        else:
            histA[y * 3 + 2] = 1
        y += 1
    return histA


if input == "":
    # If the match starts the Neural Network is trained against the
    # csv data to establish the weights of the connections.
    terminateError = 0.1  # This value determines when the Average error of the NN
    # is small enough to give a accurate prediction.
    hiddenLayerSize = 12  # This value determines the number of nodes in the Hidden Layer.
    constant = 2  # This is a constant that determines the rate of gradient decent.
    # Initialise Layers 1 input layer 2 hidden layers and 1 output layer
    inputLayer0 = []
    hiddenLayer1 = []
    hiddenLayer2 = []
    outputLayer3 = []
    # Weights of the connections between layers
    weight1 = []  # Between Input Layer and Hidden Layer 1
    weight2 = []  # Between Hidden Layer 1 and Hidden Layer 2
    weight3 = []  # Between Hidden Layer 2 and Output Layer
    # Sigmoid value of each neuron in a given Layer
    z1 = []  # HiddenLayer 1 sigmoid value array
    z2 = []  # HiddenLayer 2 sigmoid value array
    z3 = []  # OutputLayer sigmoid value array
    # Feed forward values
    # The values of the RPS output layer calculated through feed forward.
    feedforwardOutputValue = []
    history = ["P", "P", "P", "P"]  # The history is initialised
    hist2 = ["P", "P", "P", "P"]
    init_Layers_Weights()  # THe Layers and weights of the NN needs to be created.
    weight3 = np.array([[-0.66119589, -2.62168913, 1.28753527],
                        [-0.54914063, -0.53361302, 0.16285548],
                        [-5.42022181, 1.01593939, 0.5977465],
                        [4.24692756, -4.43153178, -3.81746169],
                        [3.21176844, -0.0651893, -3.29182633],
                        [-1.01017338, 4.41552022, -4.24360004],
                        [0.83573932, -3.45781449, -0.89367652],
                        [0.46254662, -1.00306656,
                            -1.10126725],
                        [2.23213326, 1.28048257,
                            -1.59004906],
                        [-5.26523416, 3.71264524,
                            -2.97257957],
                        [-0.30874854, -2.52921068,
                            1.79196534],
                        [-0.16745571, -3.00073281,
                            1.96624231],
                        [-1.32971075, -2.06794795,
                            1.18232026]])
    weight2 = np.array([[7.68405662e-01, -9.53466642e-01,
        1.94733613e-01, -3.39512944e+00
                            , -2.14046221e+00, -7.63354517e
                                +00, -1.40275517e+00,
                                -1.40245078e+00
                            , -4.19840654e+00, -4.48134340e
                                +00, 2.30838981e-01,
                                3.74663204e-01],
                        [-2.31084713e-01, 2.73427773e-01,
                            4.66244919e-01, -9.15691642e-01
                            , 9.58068515e-01, 3.40126795e
                                -01, 1.24797104e-01,
                                -2.25726888e-01
                            , 4.95350666e-01, 1.45719836e
                                +00, 2.39908909e-01,
                                -5.97692982e-01],
                        [1.78572939e+00, -4.50192424e-02,
                            3.00081427e+00, -1.62312804e+00
                            , -5.47610004e+00, -1.46926021e
                                +00, -2.69324064e+00,
                                -2.11021688e+00
                            , -5.02639747e+00, 1.51588573e
                                -01, 2.14007289e+00,
                                2.57545065e+00],
                        [-2.41388381e+00, -9.38038253e-01,
                            -5.54352393e+00, 7.18248741e+00
                            , 3.30146650e+00, -6.96217333e
                                -01, 1.33711025e+00,
                                8.52179751e-01
                            , -1.03453924e+00, -5.24436177e
                                +00, -3.20986880e+00,
                                -3.74288105e+00],
                        [-1.02045569e+00, -1.88166683e-01,
                            -2.85775220e+00, 1.27556457e+00
                            , 3.92617465e+00, 2.54540074e
                                +00, 2.44713508e+00,
                                5.90954062e-01
                            , 3.20327798e+00, 2.14925710e
                                +00, -1.37724439e+00,
                                -2.06512462e+00],
                        [-3.39139400e+00, -1.47541555e+00,
                            7.40972619e-02, -1.12336401e+00
                            , -3.78357028e+00, 4.71341628e
                                +00, -2.07710847e+00,
                                -1.41113125e+00
                            , 1.27462327e+00, 2.47695172e
                                +00, -3.28531945e+00,
                                -3.86399366e+00],
                        [-1.11471584e+00, -7.70370362e-01,
                            -3.05917575e+00, 8.24168140e-01
                            , 1.97579413e+00, -4.64425606e
                                +00, 2.46525596e+00,
                                1.01164728e+00
                            , 1.43161384e+00, -1.76540736e
                                +00, -8.54083761e-01,
                                -8.58890589e-01],
                        [-3.12286573e-01, 2.31536265e-02,
                            -1.19700166e+00, 2.23695385e+00
                            , 2.56426091e+00, -2.23158030e
                                +00, 1.50223943e+00,
                                1.02197000e+00
                            , 1.12226782e+00, -1.39236346e
                                +00, -3.44778855e-01,
                                -4.69763179e-01],
                        [-2.33825366e+00, -2.84021619e-01,
                            -5.40514256e-01, -5.82999215e-01
                            , 2.71109815e+00, 3.33411780e
                                +00, 5.44211543e-01,
                                -9.86746212e-02
                            , 3.29460398e+00, 4.05063007e
                                +00, -1.90230997e+00,
                                -2.42112670e+00],
                        [-1.17751619e+00, -3.52305361e-01,
                            9.75973072e-01, -2.44416893e+00
                            , -1.60097209e+00, 2.86424859e
                                +00, -4.73806086e+00,
                                -1.17507898e+00
                            , -1.06984604e-01, 7.23908928e
                                +00, -1.73144071e+00,
                                -1.36338694e+00],
                        [9.41895714e-01, 6.40450737e-01,
                            6.58636070e-01, 1.84901980e-01
                            , -7.08104212e-02, -5.25914196e
```

```
                          -01, -4.15903045e-03, 2.23715659e-01
                          , 6.87451974e-01, -1.15524107e+00, 7.94593309e-01, 5.57405789e-01],
                         [9.62784442e-01, 5.09850936e-01, 2.73105307e-01, -1.50338247e-02
                          , 2.08457009e-01, -1.14310748e+00, 9.24092187e-01, 3.08066312e-01
                          , -3.99201417e-01, -2.59403760e+00, 7.52373384e-01, 9.40688309e-01],
                         [-2.14503250e+00, -1.62464726e+00, -6.94241718e-01, -1.67024948e+00
                          , -3.10496489e+00, -7.76307796e-01, -2.68124744e+00, -2.47769461e+00
                          , -1.96429579e+00, -2.69714573e+00, -1.83044393e+00, -1.77305169e+00]])
    weight1 = np.array([[2.99924288, -0.44149766, 0.03982674, -0.1663377, 0.18566107, -3.54008005
                          , 2.12048221, 0.8404493, 0.03558765, -0.85593692, 0.73750741, 0.70157011],
                         [-0.63366395, -0.53944985, -5.54004924, -0.34517938, 0.53883013, 4.63010985
                          , 2.24138047, 0.31708712, 0.20515264, -0.36277893, -2.92935238, -3.49246788],
                         [-1.76529636, -0.05919467, 5.45850311, -0.30196933, -1.21922974, -0.85779947
                          , -4.2310419, -2.81554697, -0.15159771, -0.58026279, -0.26085204, 0.19631896],
                         [-2.88791608, -1.10289099, -1.33460407, 4.4219026, -1.64983038, 0.56098888
                          , -0.03441273, 0.82798142, -2.2804557, -2.15235744, -0.5457546, -1.52942235],
                         [0.05048957, -0.36804721, 0.25788167, -2.31944686, 4.03459938, -0.22220648
```

```
          , 1.49693493, -1.68033154,
             4.0028309, 4.01150028,
             -1.63680223, -1.40084954],
[3.23101253, -0.09605562,
     0.88378099, -2.68478309,
     -3.0539961, -0.58217472
          , -0.6642017, -0.86020587,
             -2.13644658, -3.47349498,
             0.82338614, 0.48022317],
[-1.97930261, -0.67741582,
     -2.87683163, -1.90509889,
     1.15357772, 5.91235943
          , -2.59373341, -0.62217443,
             1.19890231, 3.11263842,
             -1.55930005, -2.03256122],
[1.81007145, -0.59759986,
     6.14105815, -2.50433948,
     -3.19388547, -3.06165146
          , -0.62561388, -1.15905964,
             -2.92984022, -1.89276358,
             -0.20564918, 0.16510137],
[0.78413344, -0.49012334,
     -3.26793201, 3.15125581,
     2.70920364, -2.9732939
          , 4.39255516, 0.94641698,
             1.8186331, -2.0147975,
             -0.33329003, -0.19057018],
[-3.18825245, -1.07567207,
     -4.7834374, 4.8641528,
     -1.01855443, 0.40557983
          , 2.1428049, 2.07907256,
             -2.26092704, -3.81843193,
             -1.19410066, -1.13191659],
[0.20118096, 0.21671913,
     -0.15883531, -3.82087476,
     3.90474989, 2.92290059
          , 1.10643825, -0.85122975,
             4.25444478, 5.664621,
             -1.24695799, -2.10199566],
[3.32592053, -0.24487379,
     4.54742655, -2.7140907,
     -3.46988805, -3.94154171
          , -2.34927043, -1.54774446,
             -2.65518284, -3.50634339,
             1.29303225, 2.14069558],
[-0.27933558, -2.41520988,
     -1.07906326, -2.26755423,
                                        -0.90533425, -1.44066524
                                        , -0.4668636, -2.65846686,
                                        -1.34989581, -2.20271189,
                                        -2.48841759, -2.29751592]])
    output = "P"
else:
    history.pop(0)
    history.append(input)
    for i in range(4):
        hist2[i] = history[i]
    feedForward(histCheck(history))
    if max(feedforwardOutputValue) == feedforwardOutputValue[0]:
        output = "P"
    elif max(feedforwardOutputValue) == feedforwardOutputValue[1]:
        output = "S"
    else:
        output = "R"
    history.pop(0)
    history.append(output)
```

Task 3 Code:

```python
# Author:        H. van der Westhuizen
# Date:          26 April 2020
# Revision:      12 May 2020
# Artificial neural network task 3

import numpy as np

beat = {"R": "P", "P": "S", "S": "R"}


def init_Layers_Weights():
    """This function initialised the weights and layer values"""
    # The hidden layers are initialised to 1. And the input and output layers are created
    for i in range(hiddenLayerSize + 1):
        if i < 13:
            inputLayer0.append([])
        hiddenLayer1.append(1)
        hiddenLayer2.append(1)
        z1.append(1)
        z2.append(1)
    for i in range(3):
        outputLayer3.append([])
        z3.append(1)


def sigmoid(weight_node):
    """The dot product values of the current layer and proceeding weight array is substituted
    into the sigmoid function"""
    sigmoid_weight_Node = 1 / (1 + np.exp(-weight_node))
    return sigmoid_weight_Node


def errorFunction(true_output):
    """This method calculates the error of the feed forward output value and true response value"""
    totalError = 0
    for i in range(3):
        totalError += pow((true_output[i] - feedforwardOutputValue[i]), 2)
    return totalError / 2


def feedForward(input_Entry):
    global feedforwardOutputValue
    global hiddenLayer1
    feedforwardOutputValue = []
    # The first hidden layer values are obtained by doing
    # the dot product of the input layer and proceeding
       weight layer
    z_x = np.dot(input_Entry, weight1)
    for i in range(hiddenLayerSize):
        z1[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer1[i] = sigmoid(z_x[i])
    # The second hidden layer values are obtained by doing
    # the dot product of the first hidden layer and
       proceeding weight layer
    z_x = np.dot(hiddenLayer1, weight2)
    for i in range(hiddenLayerSize):
        z2[i] = z_x[i]
        #  The layer is normalised between 0 and 1
        hiddenLayer2[i] = sigmoid(z_x[i])
    # The output layer values are obtained by doing
    # the dot product of the second hidden layer and
       proceeding weight layer
    z_x = np.dot(hiddenLayer2, weight3)
    for i in range(3):
        z3[i] = z_x[i]
        feedforwardOutputValue.append(sigmoid(z_x[i]))


def histCheck(hist):
    """The input from the oponent is processed here into
    1 hot encoding"""
    histA = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
    y = 0
    for i in hist:
        if i == 'R':
            histA[y * 3] = 1
        elif i == 'P':
            histA[y * 3 + 1] = 1
        else:
            histA[y * 3 + 2] = 1
        y += 1
    return histA


if input == "":
    # If the match starts the Neural Network is trained
       against the
    # csv data to establish the weights of the connections.
    terminateError = 0.1  # This value determines when the Average error of the NN
    # is small enough to give a accurate prediction.
    hiddenLayerSize = 12  # This value determines the number of nodes in the Hidden Layer.
    constant = 2  # This is a constant that determines the rate of gradient decent.
    # Initialise Layers 1 input layer 2 hidden layers and 1 output layer
    inputLayer0 = []
    hiddenLayer1 = []
    hiddenLayer2 = []
    outputLayer3 = []
    # Weights of the connections between layers
    weight1 = []  # Between Input Layer and Hidden Layer 1
    weight2 = []  # Between Hidden Layer 1 and Hidden Layer 2
    weight3 = []  # Between Hidden Layer 2 and Output Layer
    # Sigmoid value of each neuron in a given Layer
    z1 = []  # HiddenLayer 1 sigmoid value array
    z2 = []  # HiddenLayer 2 sigmoid value array
    z3 = []  # OutputLayer sigmoid value array
    # Feed forward values
    # The values of the RPS output layer calculated through feed forward.
    feedforwardOutputValue = []
    history = ["P", "P", "P", "P"]  # The history is initialised
    hist2 = ["P", "P", "P", "P"]
    init_Layers_Weights()  # THe Layers and weights of the NN needs to be created.
    weight3 = np.array([[-0.66119589, -2.62168913, 1.28753527],
                        [-0.54914063, -0.53361302, 0.16285548],
                        [-5.42022181, 1.01593939, 0.5977465],
                        [4.24692756, -4.43153178, -3.81746169],
                        [3.21176844, -0.0651893, -3.29182633],
                        [-1.01017338, 4.41552022, -4.24360004],
                        [0.83573932, -3.45781449, -0.89367652],
                        [0.46254662, -1.00306656,
                            -1.10126725],
                        [2.23213326, 1.28048257,
                            -1.59004906],
                        [-5.26523416, 3.71264524,
                            -2.97257957],
                        [-0.30874854, -2.52921068,
                            1.79196534],
                        [-0.16745571, -3.00073281,
                            1.96624231],
                        [-1.32971075, -2.06794795,
                            1.18232026]])
    weight2 = np.array([[7.68405662e-01, -9.53466642e-01,
        1.94733613e-01, -3.39512944e+00
                            , -2.14046221e+00, -7.63354517e
                                +00, -1.40275517e+00,
                                -1.40245078e+00
                            , -4.19840654e+00, -4.48134340e
                                +00, 2.30838981e-01,
                                3.74663204e-01],
                        [-2.31084713e-01, 2.73427773e-01,
                            4.66244919e-01, -9.15691642e-01
                            , 9.58068515e-01, 3.40126795e
                                -01, 1.24797104e-01,
                                -2.25726888e-01
                            , 4.95350666e-01, 1.45719836e
                                +00, 2.39908909e-01,
                                -5.97692982e-01],
                        [1.78572939e+00, -4.50192424e-02,
                            3.00081427e+00, -1.62312804e+00
                            , -5.47610004e+00, -1.46926021e
                                +00, -2.69324064e+00,
                                -2.11021688e+00
                            , -5.02639747e+00, 1.51588573e
                                -01, 2.14007289e+00,
                                2.57545065e+00],
                        [-2.41388381e+00, -9.38038253e-01,
                            -5.54352393e+00, 7.18248741e+00
                            , 3.30146650e+00, -6.96217333e
                                -01, 1.33711025e+00,
                                8.52179751e-01
                            , -1.03453924e+00, -5.24436177e
                                +00, -3.20986880e+00,
                                -3.74288105e+00],
                        [-1.02045569e+00, -1.88166683e-01,
                            -2.85775220e+00, 1.27556457e+00
                            , 3.92617465e+00, 2.54540074e
                                +00, 2.44713508e+00,
                                5.90954062e-01
                            , 3.20327798e+00, 2.14925710e
                                +00, -1.37724439e+00,
                                -2.06512462e+00],
                        [-3.39139400e+00, -1.47541555e+00,
                            7.40972619e-02, -1.12336401e+00
                            , -3.78357028e+00, 4.71341628e
                                +00, -2.07710847e+00,
                                -1.41113125e+00
                            , 1.27462327e+00, 2.47695172e
                                +00, -3.28531945e+00,
                                -3.86399366e+00],
                        [-1.11471584e+00, -7.70370362e-01,
                            -3.05917575e+00, 8.24168140e-01
                            , 1.97579413e+00, -4.64425606e
                                +00, 2.46525596e+00,
                                1.01164728e+00
                            , 1.43161384e+00, -1.76540736e
                                +00, -8.54083761e-01,
                                -8.58890589e-01],
                        [-3.12286573e-01, 2.31536265e-02,
                            -1.19700166e+00, 2.23695385e+00
                            , 2.56426091e+00, -2.23158030e
                                +00, 1.50223943e+00,
                                1.02197000e+00
                            , 1.12226782e+00, -1.39236346e
                                +00, -3.44778855e-01,
                                -4.69763179e-01],
                        [-2.33825366e+00, -2.84021619e-01,
                            -5.40514256e-01, -5.82999215e-01
                            , 2.71109815e+00, 3.33411780e
                                +00, 5.44211543e-01,
                                -9.86746212e-02
                            , 3.29460398e+00, 4.05063007e
                                +00, -1.90230997e+00,
                                -2.42112670e+00],
                        [-1.17751619e+00, -3.52305361e-01,
                            9.75973072e-01, -2.44416893e+00
                            , -1.60097209e+00, 2.86424859e
                                +00, -4.73806086e+00,
                                -1.17507898e+00
                            , -1.06984604e-01, 7.23908928e
                                +00, -1.73144071e+00,
                                -1.36338694e+00],
                        [9.41895714e-01, 6.40450737e-01,
                            6.58636070e-01, 1.84901980e-01
                            , -7.08104212e-02, -5.25914196e
```

```
                                    -01, -4.15903045e-03,
                                    2.23715659e-01
                                  , 6.87451974e-01, -1.15524107e
                                    +00, 7.94593309e-01,
                                    5.57405789e-01],
                              [9.62784442e-01, 5.09850936e-01,
                                2.73105307e-01, -1.50338247e-02
                                  , 2.08457009e-01, -1.14310748e
                                    +00, 9.24092187e-01,
                                    3.08066312e-01
                                  , -3.99201417e-01, -2.59403760e
                                    +00, 7.52373384e-01,
                                    9.40688309e-01],
                              [-2.14503250e+00, -1.62464726e+00,
                                -6.94241718e-01, -1.67024948e+00
                                  , -3.10496489e+00, -7.76307796e
                                    -01, -2.68124744e+00,
                                    -2.47769461e+00
                                  , -1.96429579e+00, -2.69714573e
                                    +00, -1.83044393e+00,
                                    -1.77305169e+00]])
        weight1 = np.array([[2.99924288, -0.44149766,
          0.03982674, -0.1663377, 0.18566107, -3.54008005
                                  , 2.12048221, 0.8404493,
                                    0.03558765, -0.85593692,
                                    0.73750741, 0.70157011],
                              [-0.63366395, -0.53944985,
                                -5.54004924, -0.34517938,
                                0.53883013, 4.63010985
                                  , 2.24138047, 0.31708712,
                                    0.20515264, -0.36277893,
                                    -2.92935238, -3.49246788],
                              [-1.76529636, -0.05919467,
                                5.45850311, -0.30196933,
                                -1.21922974, -0.85779947
                                  , -4.2310419, -2.81554697,
                                    -0.15159771, -0.58026279,
                                    -0.26085204, 0.19631896],
                              [-2.88791608, -1.10289099,
                                -1.33460407, 4.4219026,
                                -1.64983038, 0.56098888
                                  , -0.03441273, 0.82798142,
                                    -2.2804557, -2.15235744,
                                    -0.5457546, -1.52942235],
                              [0.05048957, -0.36804721,
                                0.25788167, -2.31944686,
                                4.03459938, -0.22220648
```

```
     , 1.49693493, -1.68033154,
        4.0028309, 4.01150028,
        -1.63680223, -1.40084954],
[3.23101253, -0.09605562,
   0.88378099, -2.68478309,
   -3.0539961, -0.58217472
     , -0.6642017, -0.86020587,
        -2.13644658, -3.47349498,
        0.82338614, 0.48022317],
[-1.97930261, -0.67741582,
   -2.87683163, -1.90509889,
   1.15357772, 5.91235943
     , -2.59373341, -0.62217443,
        1.19890231, 3.11263842,
        -1.55930005, -2.03256122],
[1.81007145, -0.59759986,
   6.14105815, -2.50433948,
   -3.19388547, -3.06165146
     , -0.62561388, -1.15905964,
        -2.92984022, -1.89276358,
        -0.20564918, 0.16510137],
[0.78413344, -0.49012334,
   -3.26793201, 3.15125581,
   2.70920364, -2.9732939
     , 4.39255516, 0.94641698,
        1.8186331, -2.0147975,
        -0.33329003, -0.19057018],
[-3.18825245, -1.07567207,
   -4.7834374, 4.8641528,
   -1.01855443, 0.40557983
     , 2.1428049, 2.07907256,
        -2.26092704, -3.81843193,
        -1.19410066, -1.13191659],
[0.20118096, 0.21671913,
   -0.15883531, -3.82087476,
   3.90474989, 2.92290059
     , 1.10643825, -0.85122975,
        4.25444478, 5.664621,
        -1.24695799, -2.10199566],
[3.32592053, -0.24487379,
   4.54742655, -2.7140907,
   -3.46988805, -3.94154171
     , -2.34927043, -1.54774446,
        -2.65518284, -3.50634339,
        1.29303225, 2.14069558],
[-0.27933558, -2.41520988,
   -1.07906326, -2.26755423,
                                        -0.90533425, -1.44066524
                                         , -0.4668636, -2.65846686, -1.34989581, -2.20271189, -2.48841759, -2.29751592]])
    output = "P"
else:
    history.pop(0)
    history.append(input)
    for i in range(4):
        hist2[i] = history[i]
    feedForward(histCheck(history))
    if max(feedforwardOutputValue) == feedforwardOutputValue[0]:
        output = "P"
    elif max(feedforwardOutputValue) == feedforwardOutputValue[1]:
        output = "S"
    else:
        output = "R"
    history.pop(0)
    history.append(output)
```