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

EAI 320

INTELLIGENT SYSTEMS

PRACTICAL 3

Name and Surname	Student Number	Signature
L.M. Burgess	18015001	

By submitting this assignment I confirm that I have read and am aware of the University of Pretoria's policy on academic dishonesty and plagiarism and I declare that the work submitted in this assignment is my own as delimited by the mentioned policies. I explicitly declare that no parts of this assignment have been copied from current or previous students' work or any other sources (including the internet), whether copyrighted or not. I understand that I will be subjected to disciplinary actions should it be found that the work I submit here does not comply with the said policies.

Contents

List of Figures

1 Introduction

Artificial Neural Networks are a form of Artificial intelligence used for optimized learning tasks with large data where possible patterns may also exist. The network may be used after learning to identify items (make approximations) from the data it learns from. The network may also be used to make predictions from the data it learns. A network is considered to be deep learning on the amount of hidden-layers present. The algorithm followed by the network is called back propagation. A Neural Network which may learn how to play rock, paper, scissors is implemented.

2 Aim

To investigate, design, implement, and optimize an Artificial Neural Network (NN) with back propagation and arrays which may be used to learn and play game of rock, paper, scissors (RPS). Research will include resources found online [1] and the prescribed textbook section on neural networks. Design of the neural network will follow the prescribed textbook's algorithm and the specification document [2]. Implementation will follow the specification of using back propagation with optimizations observed from resources online and targets prescribed. The specifications being in terms of run time and accuracy. Optimization observes the learning rate, number of hidden layers, the error rate, and number of epochs (learning iterations) in order to find the best operation of the neural network. The format the input and output data is formatted in with the choice of transfer function also of consideration. The final test being the performance of an online learning-gameplay-session against another agent with run time and win-rate being observed. The run time desired to be as low as possible and the win rate as high as possible.

3 Implementation and design

The textbook algorithm, [3] found in appendix A, gives the basic design implementation of the neural network. A neural network consists of two parts: forward and back propagation.

Forward propagation follows the process:

1. Receive input data (A) at a node
2. Multiply A with the corresponding weights (W)
3. This generates the next layers input data (IN)
4. IN is normalized using the activation function (A)
5. The normalized data is the input data (A) for the next layer

The back propagation process follows the process:

1. From the output layer with input data (A) calculate the cost (error) from the correct input A*: Cost = (A* - A)
2. Calculate the minimum change (Epsilon) using the derivative of the activation function and ”IN” from forward propagation
3. Multiply epsilon by the cost to determine the output Delta
4. The delta is then propagated back a layer from the output, opposite in relative direction to forward propagation
5. Determine epsilon for the layer
6. Multiply Delta by the sum of the weights connected to the forward propagation side and epsilon
7. The Delta is then propagated back until it has reached the second last layer

The weights are then updated by multiplying the calculated delta at each layer with the learning rate and the input data to the node A. The result is added to the connected weights of the node.

3.1 Input and output data

The neural network will receive input data in the format of two previous rounds. One round consists of a move by agent X (the neural network) and agent Y (opposing agent). The Python library, Numpy, and it's matrices are used to generate the data processing format. Input and output data is represented using a one-hot-encoding format. The object played will be set to ”1” and the remaining objects are set to ”0” at the input. The output will have values in the range between zero and one.

3.1.1 Input data

The input data will be either be a definite one or zero. From one data sample of two rounds, twelve input points are then provided. The format being RPS,RPS, RPS, RPS where one of the objects is one for each RPS and the rest zero. Multiple samples are able to be read in this format. The one-hot encoding scheme was chosen as only the correct option is shown as correct or progress forward. If a win-draw-lose ”1, 0, -1” scheme was chosen the network may learn slower and weighting is given to the network drawing the game. This may be less beneficial as instead of training the network to win, there is learning benefit to drawing. The network may also have to unlearn if the option of ”-1” is given. It is thus more beneficial to the use of a true-false system of ”1,0”.

3.1.2 Output data

The output data is not a definite one or zero. The neural network learns from the input data and will make a prediction or approximation from what is has learned. The output objects will thus have non-zero values. Through the fine tuning process of back propagation the objects will approach one or zero for each set of RPS. A helper function (in appendix C) then converts the output approximations into a set of moves to be played. The helper function operates on a tolerance for determining which object is determined (1) and which objects are not (0).

3.2 Cost, Loss, error

The next move to be played by the opponent is also provided with the two previous rounds as input. The ply is converted to a definite one-hot move set as well. The move is compared against the predicted move and the cost function may then be developed to calculate the error and loss used for back propagation and performance interpretation. The results are stored in vectors.

3.2.1 Loss

After each epoch (training iteration) the loss is calculated as:

$$Loss = \frac{1}{2} * \sum (t_k - y_k)^2 \tag{1}$$

The loss acts as a performance measure of the neural network's ability to learn the data presented and predict the most suitable move to be played. "t" is the target or provided move which will beat the opposing agent's next move. It is also processed in one-hot format. "y" is the predicted move by the neural network. "k" being the node or object in the RPS set being observed.

3.2.2 Cost

Cost is measured after each epoch as:

$$Cost = t_k - y_k \tag{2}$$

The cost is implemented with the back-propagation algorithm. It is the error in predicting the output moves of the neural network.

3.2.3 Error Convergence Criterion

The error convergence criterion is used to terminate the learning process of a data sample set early. The early termination prevents the neural network from over learning or fine tuning too much to the current data set. The neural network requires to strike a balance between being general and specialized in its prediction and learning process. If the the neural network over learns, it will struggle to apply itself in learning another differing data sample set. The predictions may also be too specific then. Too specific a set of predictions may lead to poor performance and results. Being more general allows the neural network to quickly adjust to a new data set and make mostly accurate results. A more specialized neural network will however make better predictions for the current data set. The error is calculated as prescribed in the specification:

$$\Delta Error = \left| Loss_t - Loss_{t-1} \right| \tag{3}$$

When this criterion is met the program will terminate the current learning batch. However upon refactoring of code and investigation the error criterion was changed to also incorporate:

$$\Delta Error = Loss_t - Loss_{t-1} \tag{4}$$

Where termination can occur on:

$$\Delta E > 0 \tag{5}$$

This was added for two main reasons:

1. The neural network may begin learning the data slowly if a small learning rate is applied. This may lead to an issue where the error is small in the beginning and the program terminates early. No learning may even take place then.
2. With the absolute value being used the indication as to whether the neural network is learning or not learning well, thus over learning is vague. If the difference in error is negative it means the neural network is learning as the loss (relative mean error) is reducing. If it is positive the neural network is no longer learning well and be learning to perform worse.

The two will be used together with the error a relatively small number to ensure early termination does not occur.

3.3 Hidden layers

The hidden layers of a neural network come with many purposes and sizes. Each layer can be designed to process a specific characteristic of the input. It may also refine aspects of the sample data inputted, observing different aspects at each layer. A hidden network of two layers also allows for discontinuities to be accounted for in the input data. Too many hidden layers may also make problems. Too many hidden layers may skew the output from any ”noise” or outlying data being identified as an aspect of importance within the data. The final implementation with online uses two layers. Too many layers and too many nodes in a layer reduce performance and timing dramatically. Different amounts of layers and nodes will be examined in the results and discussion.

The default size used by the agent for the number of hidden nodes in one layer is:

$$HN = \frac{N_{samples}}{4} \tag{6}$$

3.4 Weights

The weights act as the links between layers. The weights are generated randomly upon first generation, and are then continuously re-used and fine tuned for the training session. The weights consist of vectors made up of the dimensions of the two layers it connects. The weights used in the default online learning application consist of three vectors:

- Input to Hidden connection
- Hidden to Hidden connection
- Hidden to Output connection

The weights are randomly generated to a number between one and minus one to six significant figures. This ensures a safe large exploration of the solution space from randomization for initializing the data for training purposes. Once the weights have been fine tuned to an acceptable level observed from the loss and specialization to data, the weights may be stored in a readable file (see appendix B and C). Training then does not need to begin from randomization then. The benefit of this is the ability to perform and make better predictions from the beginning.

3.5 Activation function

The activation function acts to normalize data submitted to a specific range within the domain of the function. It converts the data from continuous to discrete. The sigmoid function will return values in the range of ”0” and ”1”. This fits the encoding of one-hot-encoding. This is beneficial as values may become larger (or smaller) than the encoding accepts due to multiplication with the weight values in forward propagation and multiplication with the learning rate in back propagation. The sigmoid thus ”squashes” or represses the value into a range which is usable by the program. If encoding using ”-1, 0, 1” is used the tanh function will be beneficial. The tanh function will repress value to a range between ”-1 and 1” in it’s domain.

The Sigmoid function:

$$\Delta(z) = \frac{1}{1 + e^{-z}} \tag{7}$$

The derivative of the sigmoid function:

$$\Delta'(z) = \frac{1}{1 + e^{-z}} * \left(1 - \frac{1}{1 + e^{-z}}\right) \tag{8}$$

3.6 Bias

The bias acts an extra node in each layer except the output, where there is no bias. The purpose of the bias is to stimulate the learning data. Learning data may be inconsistent or be missing a component required for approximating the data well or correctly. The bias exists with a fixed value of one and has its own set on weights connected to the next layer. No weights are connected to it from the previous layer.

4 Optimization

4.1 Matrix multiplication

Instead of using loops for most multiplication operations within the neural network, matrix multiplication may occur. This is a much faster operation at a low time cost as the size of the array grows compared to running loops to perform the same operation. The input data is multiplied with the linked weights to the hidden layer, the hidden layer with the weights to the next hidden layer, and the second hidden layer to the output layer.

4.2 Saving to CSV and hardcoding

The weights after a training session may be saved to a CSV from their respective and reloaded upon the next training session. This allows the neural network to not need to relearn from initialization each time. This is especially useful for when the neural network is implemented as an agent as it may not necessarily lose as many starting rounds when learning the opponents moves. Hard coding speeds the program even further as input-output operations are relatively slow to hard coded values stored within the program on RAM already.

4.3 Number of data sets trained with and epochs

The number of data sets containing data samples to train with affects how fine tuned a neural network may be. Paired with the number of epochs (training iterations) the neural network is able to strike a good balance between generalization and specialization. The more data with less epochs may lead to more generalization and vice versa. The more epochs given, the more time the neural network has to learn a set of data and specialize to it.

4.4 Learning rate on the training process and convergence criterion

The learning rate affects how quickly the data may approach values depending on how small or big the constant is set to. A small learning rate makes the network approach the minimum (correct prediction) slower but more accurately. If too small however the neural network will fail to learn at all as the propagated change to update the weights will be too small and insignificant. Larger values allow the learning rate to approach faster but come with the caveat that the large value may make it go past the minimum away from the correct approximation if too big. The neural network will not learn or minimize the error then. The value requires fine tuning and testing for a safe size for speed. The convergence criterion allows the neural network to learn consistently and constructively. If the rate is too large and the learning diverges it will stop the learning batch.

5 Online learning and Game-playing

The agent is trained off the data of the neural network to play against other RPS agents. The agent learns from the games played against the other agents and fine tunes it's next move set using the neural network. The predicted moves may then be used to beat the other agent.

5.1 Terminology

- One match consists of N rounds
- One round is where each agent performs a single move (RPS)

5.2 Goal

The agent is required to perform one hundred matches consisting of ten thousand rounds in one session. The agent is desired to win as many rounds as possible in as fast a run time as possible. It is required to be able to run in less than ten minutes or six hundred seconds. It will be required to perform back propagation after N rounds. Where N may be any number between two and a hundred.

5.3 Implementation

The agent is implemented using hard coded weights which were optimized and manually chosen before being hard coded. The neural network was trained off the sample data and examined for the most optimal weights for any sample size one hundred or less. This allows the most optimal read speed for the weights used as CSV is too slow and when multi-threading deadlock occurs.

Upon startup of the agent for the match, two implementations may occur: the first is the number of rounds before back propagation begins is a fixed number for N (faster) and the other is a random number generated for each match. The N chosen will then generate N random moves to begin with. This is due to the fact no opposing data exists yet as the opposing agent is unknown and has not played yet. No predictions may then occur. The agent plays until all moves are exhausted. Forward propagation then occurs to receive the next play set half the size of N. The new play set is played and the opposing data is collected as the cost.

Back propagation now occurs with the first set of N data points and opposing data points with the newly collected cost. The top N-half of the original data randomly generated is then discarded. The second half is shifted up, and the predicted output and cost are appended as the second set of data points. Forward propagation occurs again where new data is predicted and the process repeats. Thus only N rounds suffer from random play and the agent has the benefit of the neural network for the rest of the match.

6 Results

6.1 Neural Network

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed. The best score being the figure reaching ”100” for moves correctly predicted. Some tests also include gameplay against another agent for run time performance tests. Error is made a small number in order to observe the trend of generalization to specialization and over-training.

6.1.1 Number of epochs

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed.

- The number of hidden layers is two
- Training sample size ”N” is one hundred
- One hundred and fifty sample sets
- Learning rate is ”0.8”
- Weights are initialized randomly with a seed value of ”1”
- Hidden nodes Sample size / 8
- Error is 0.00001

Epochs examined:
One:

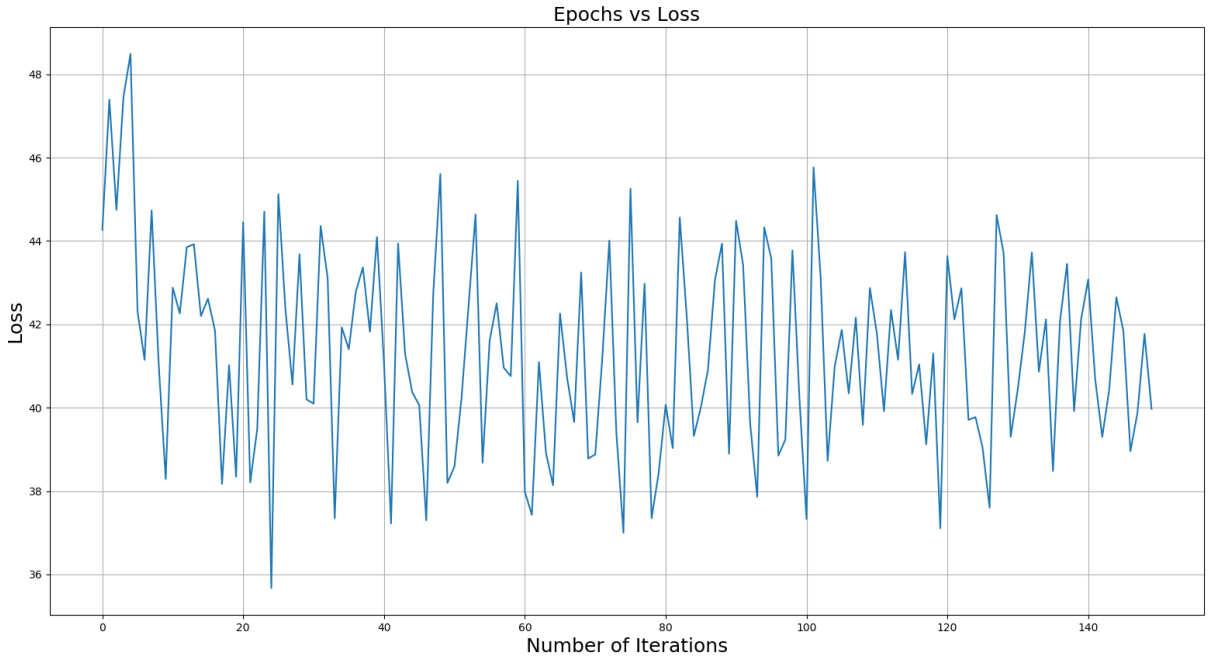

Figure 1: Loss

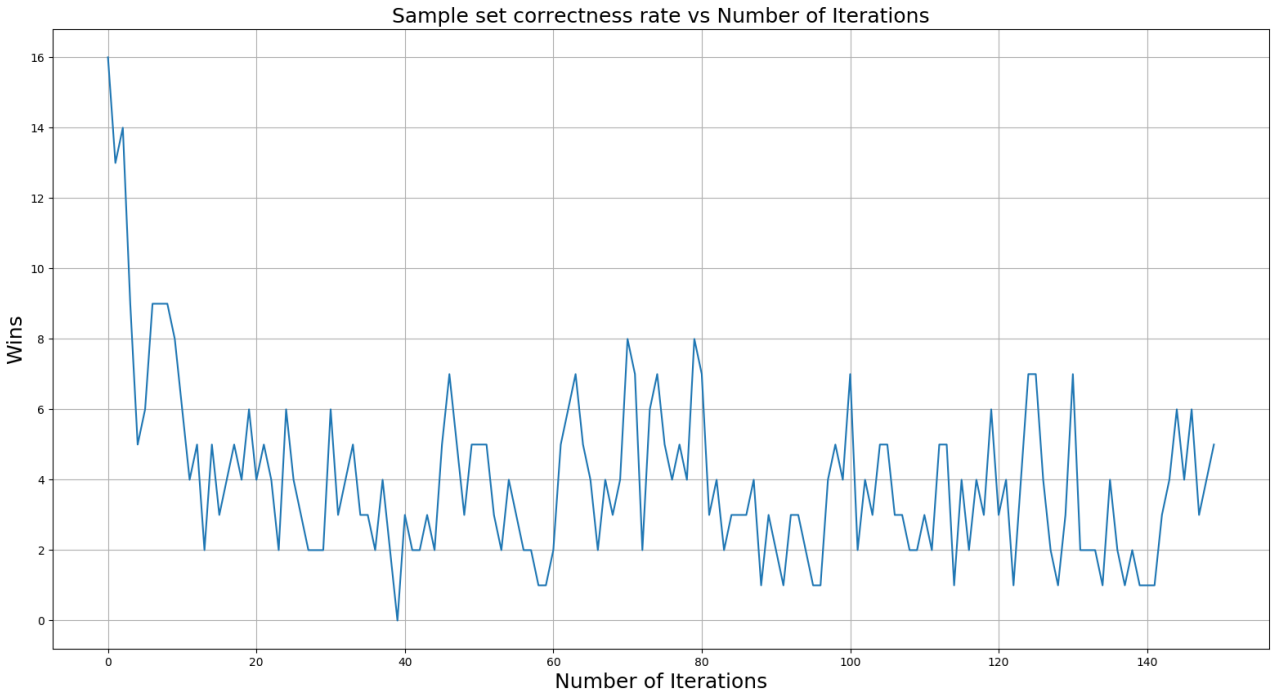

Figure 2: Moves correctly predicted

Five:

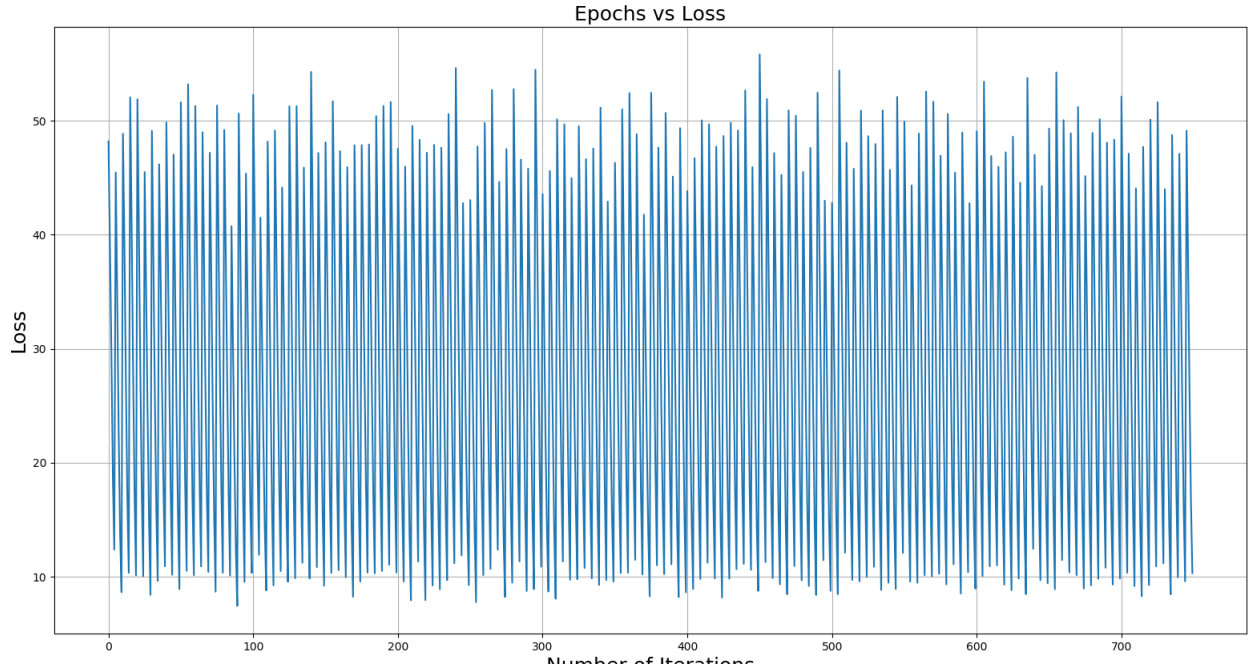

Figure 3: Loss

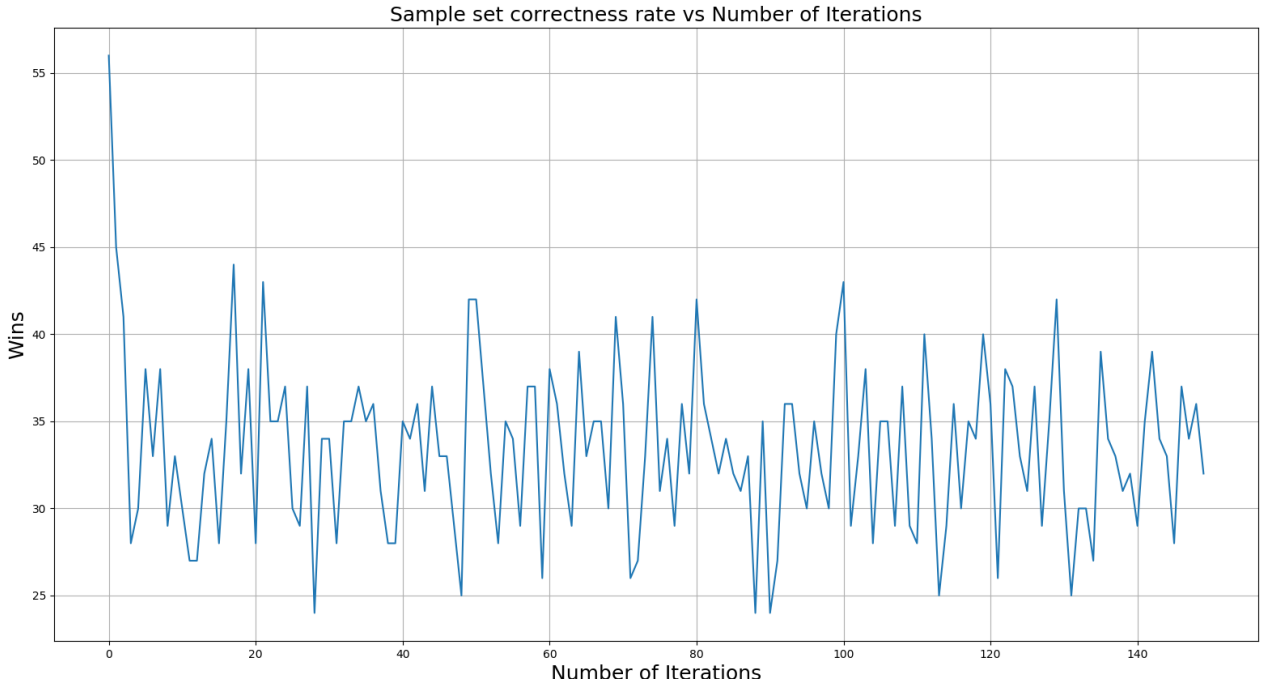

Figure 4: Moves correctly predicted

Ten:

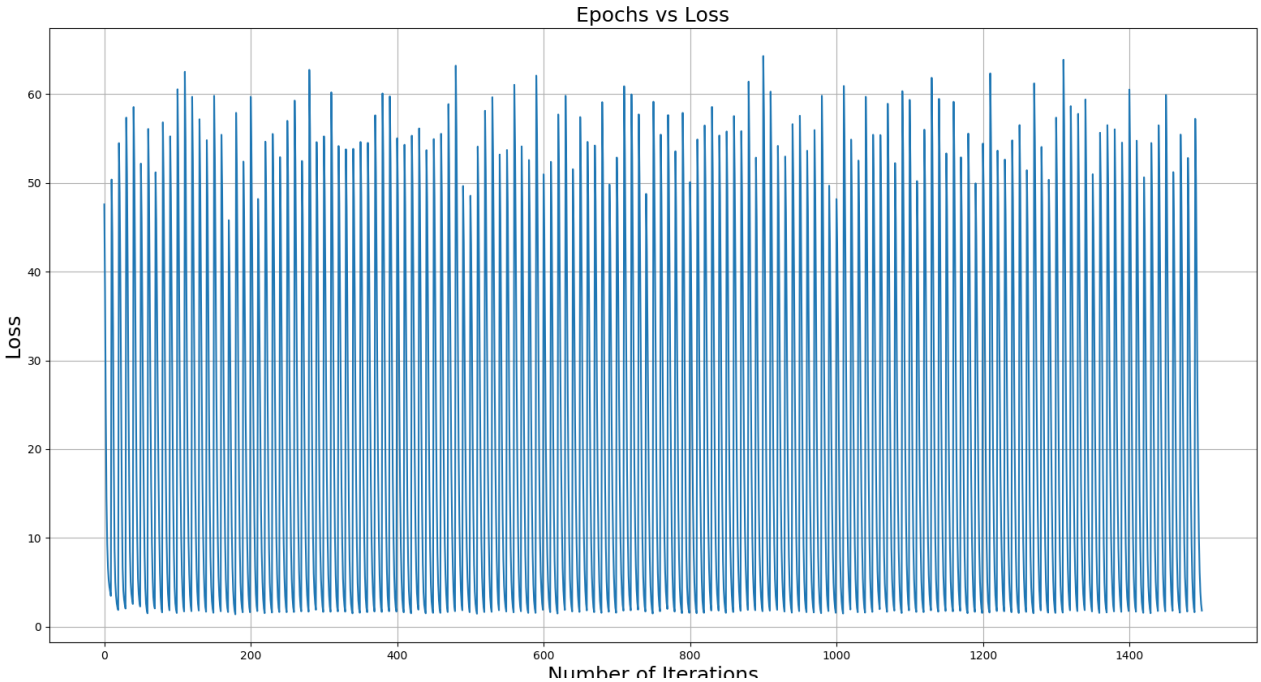

Figure 5: Loss

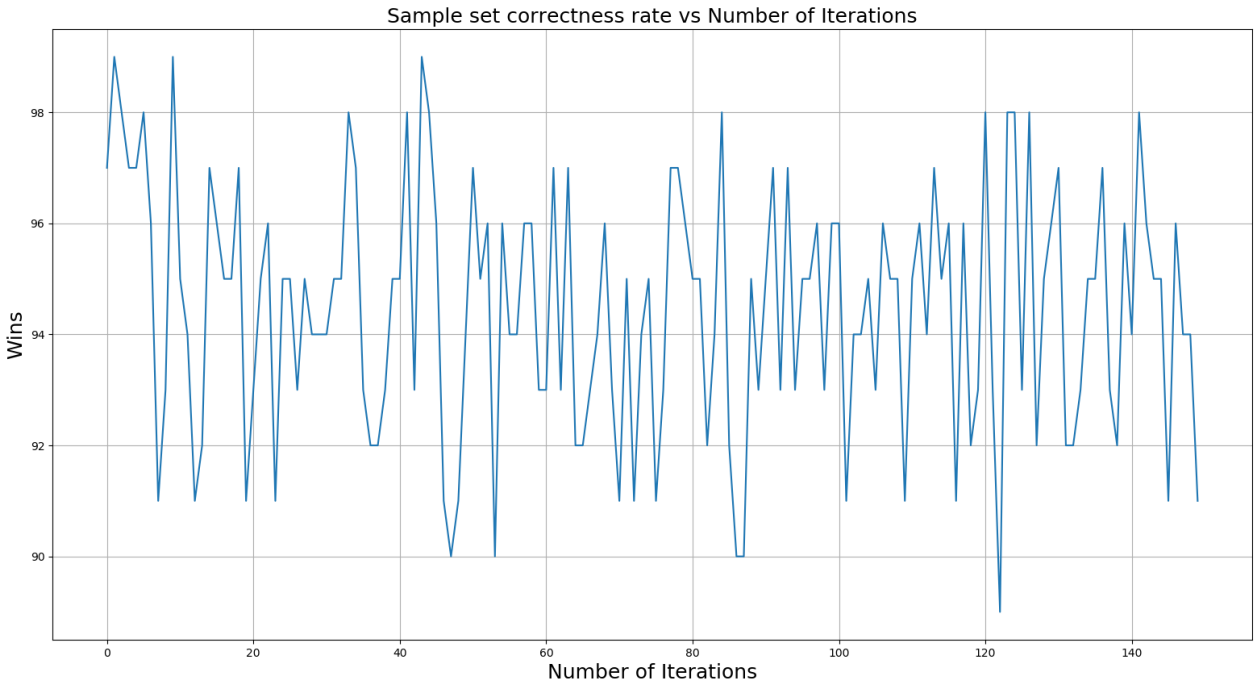

Figure 6: Moves correctly predicted

Fifteen:

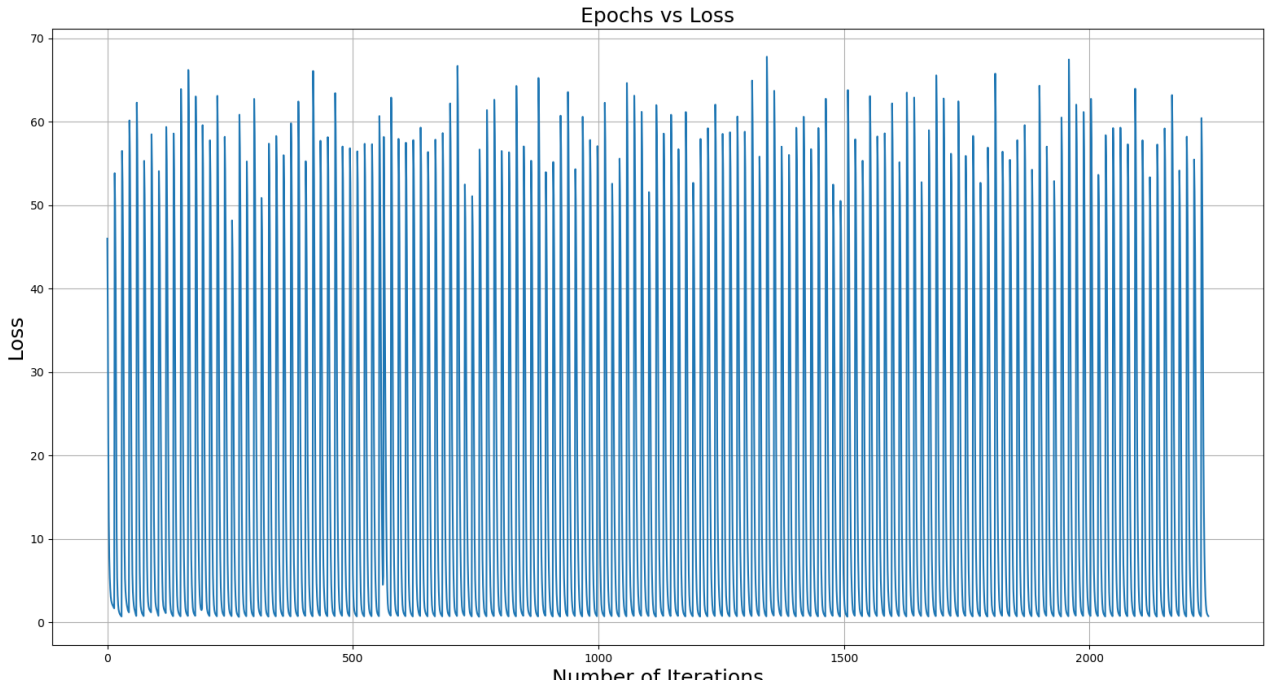

Figure 7: Loss

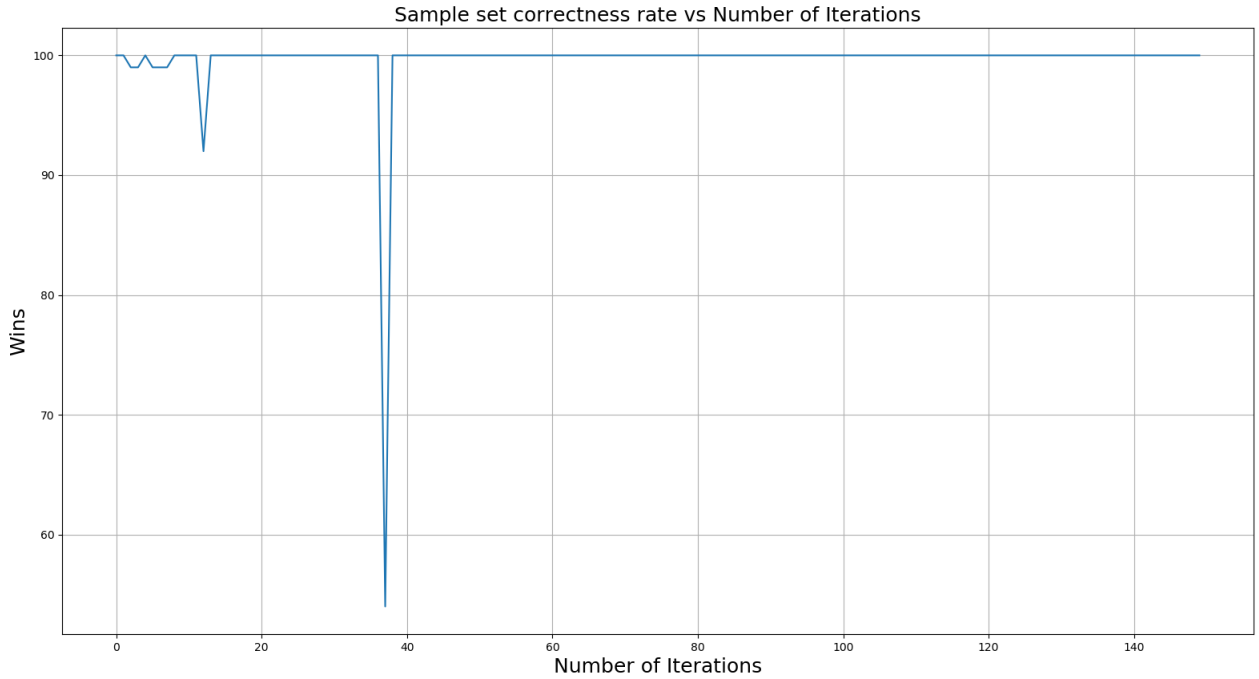

Figure 8: Moves correctly predicted

Twenty:

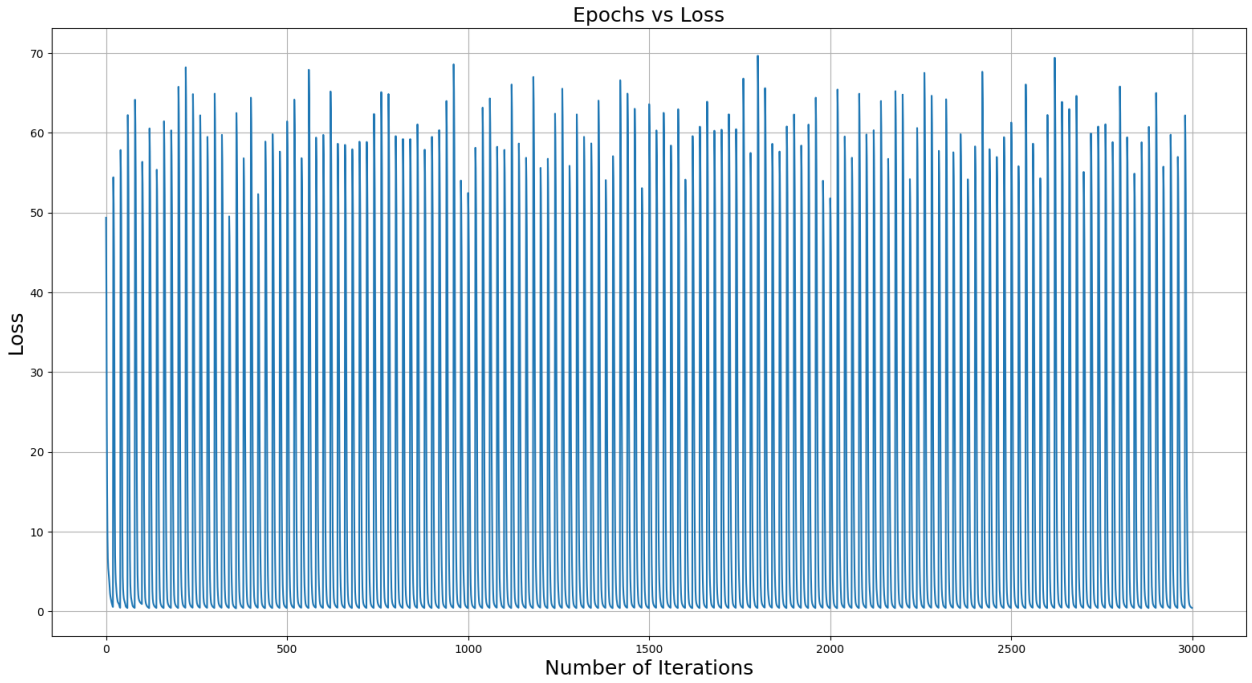

Figure 9: Loss

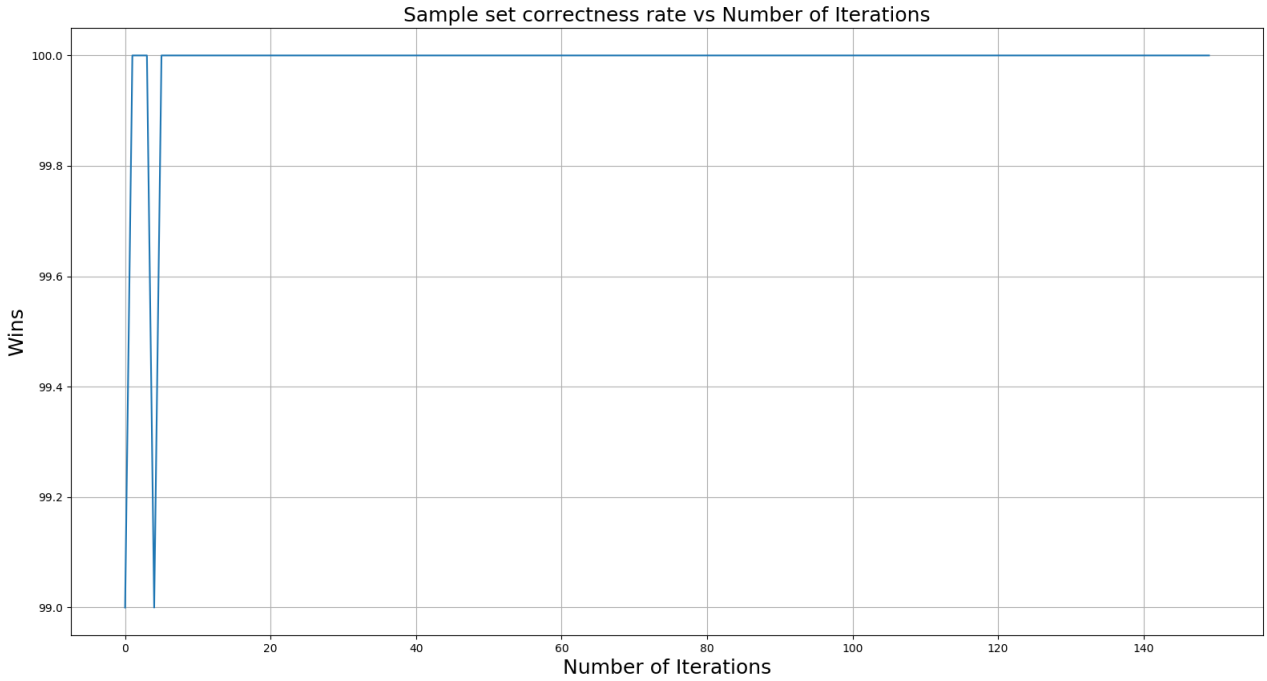

Figure 10: Moves correctly predicted

Thirty:

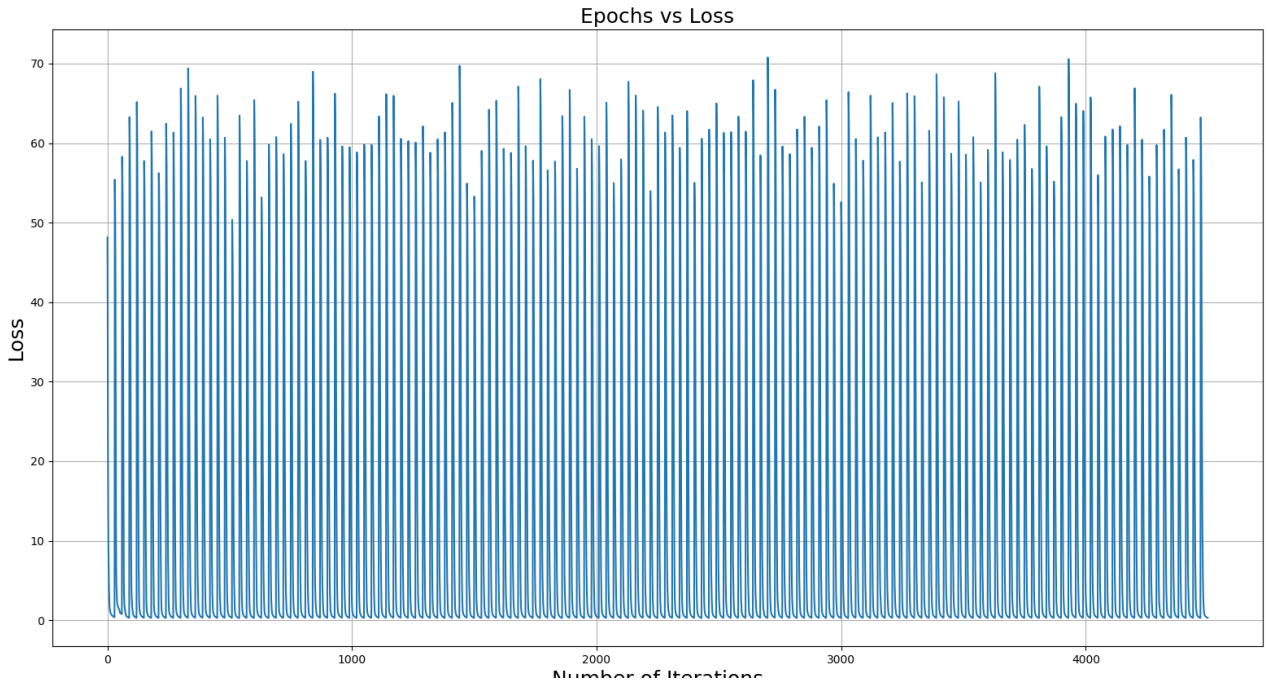

Figure 11: Loss

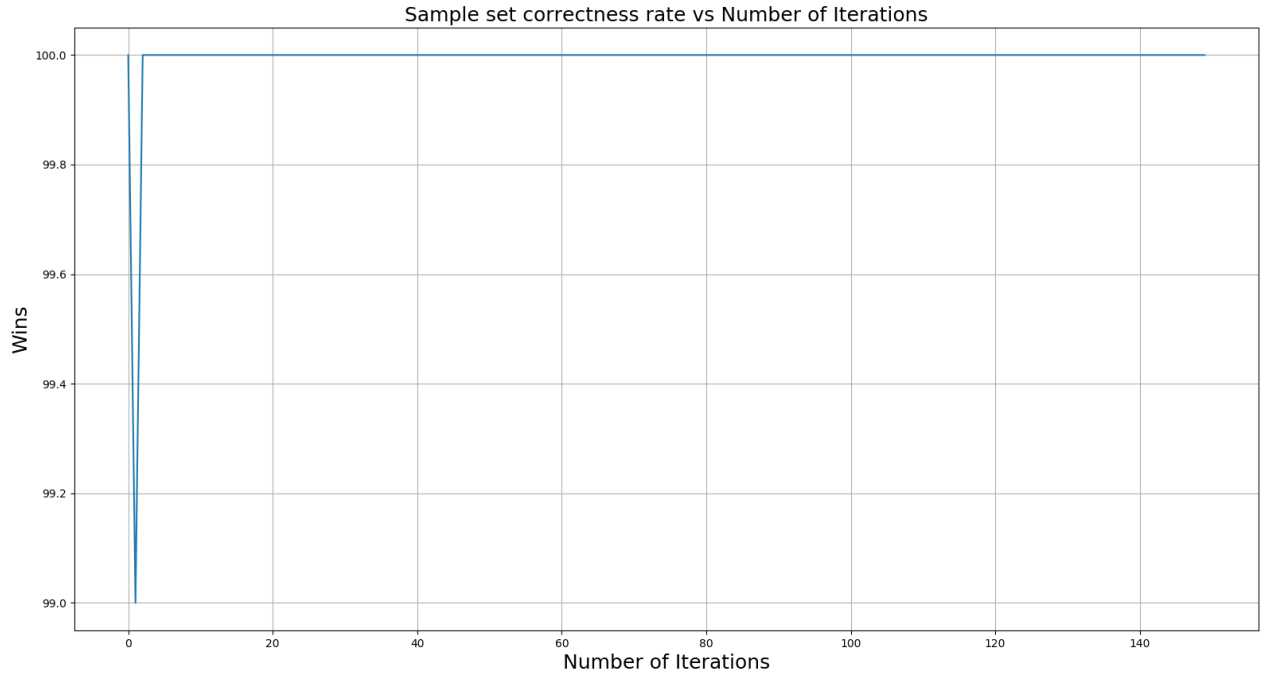

Figure 12: Moves correctly predicted

6.1.2 Number of hidden layers

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed.

- The number of nodes in a layer is Sample size /8
- Training sample size is sixty
- One hundred and fifty sets which are the same for each test
- Learning rate is ”0.8”
- Weights are initialized randomly with a seed value of ”1”
- Ten epochs
- Error is 0.00001

One, two, three, four and five hidden layers are observed: One layer:

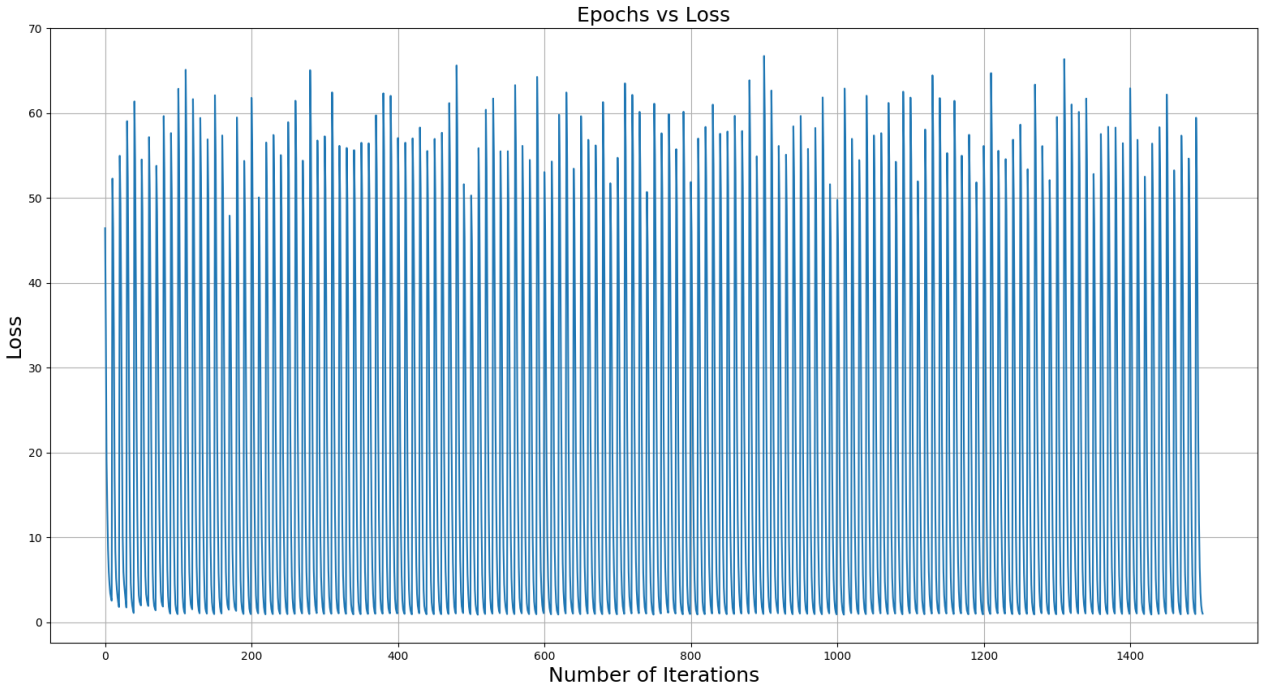

Figure 13: Loss for neural network with one hidden layer

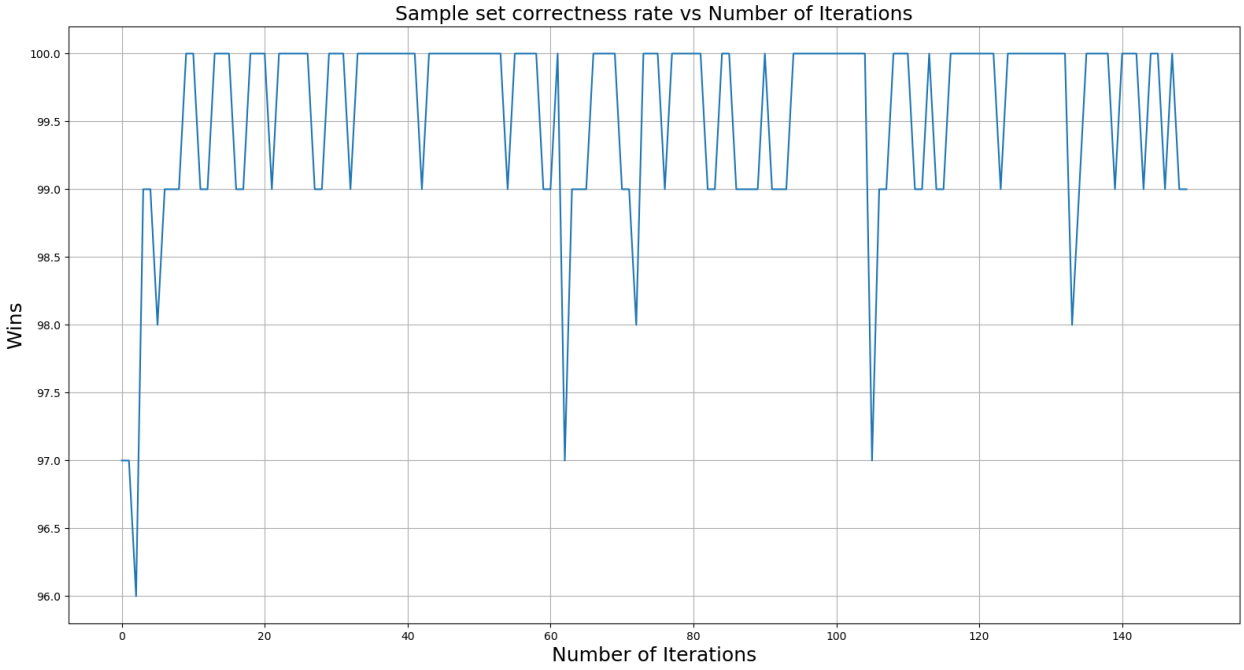

Figure 14: Moves correctly predicted for neural network with one hidden layer

Two layers:

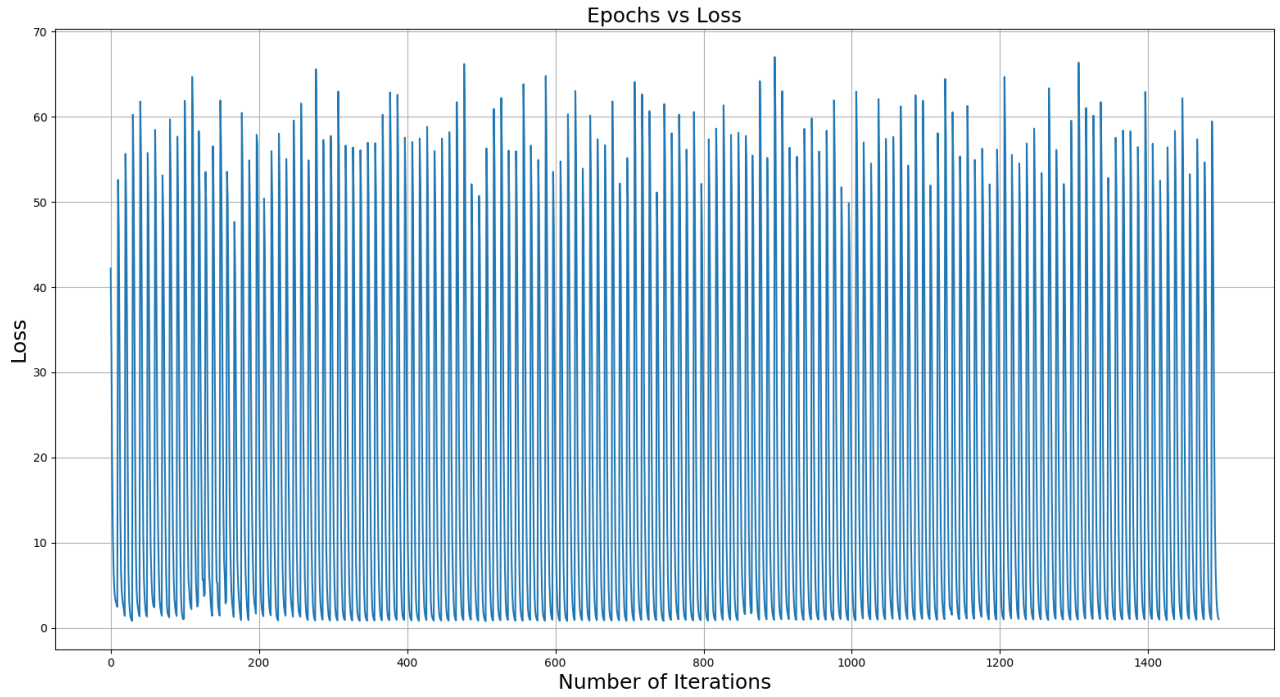

Figure 15: Loss for neural network with two hidden layers

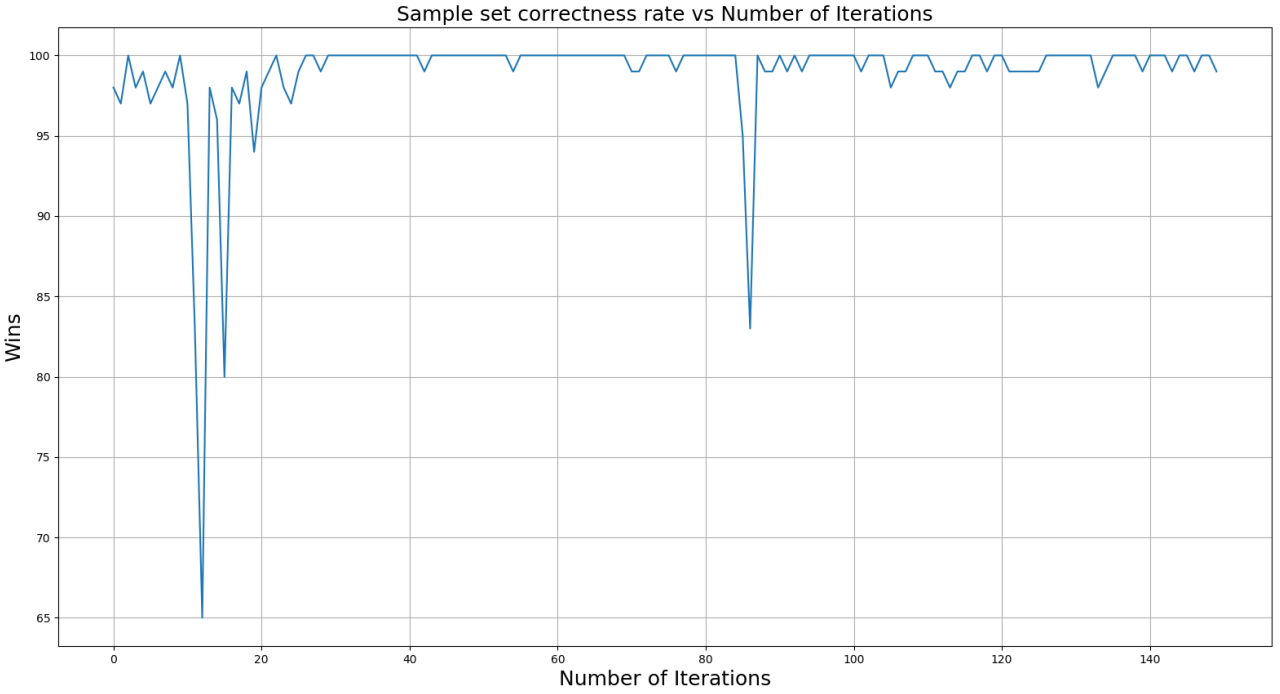

Figure 16: Moves correctly predicted for neural network with two hidden layers

Three layers:

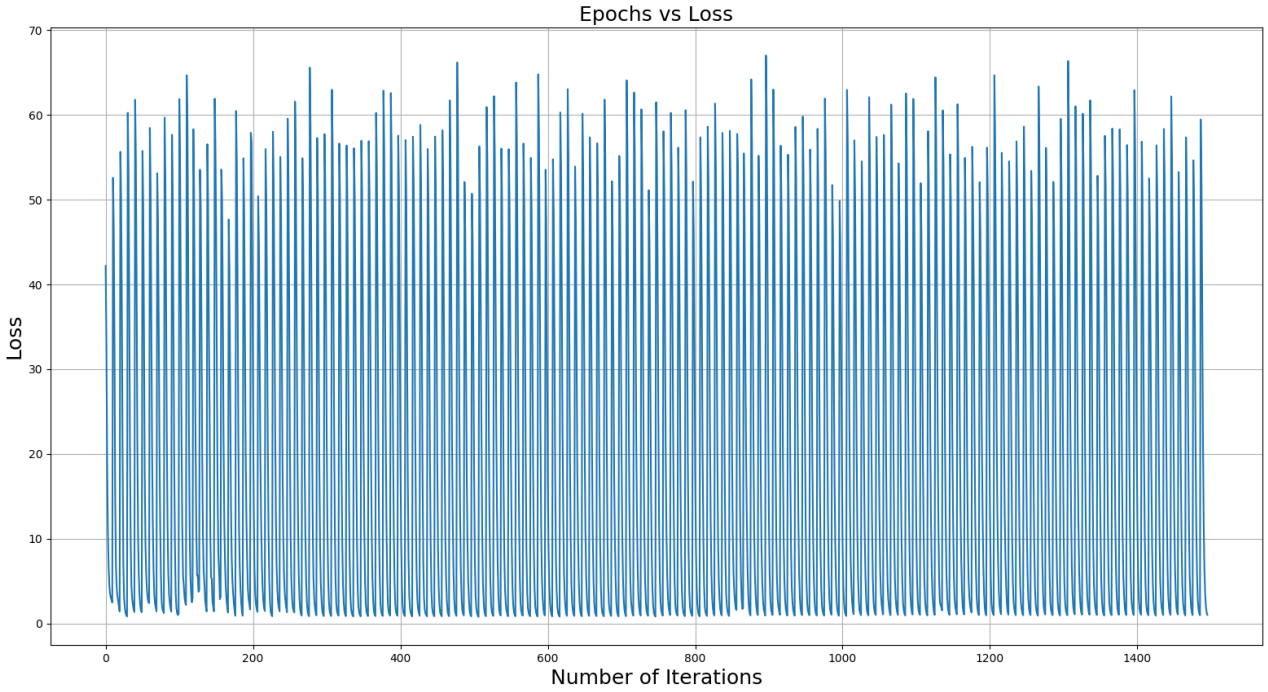

Figure 17: Loss for neural network with three hidden layers

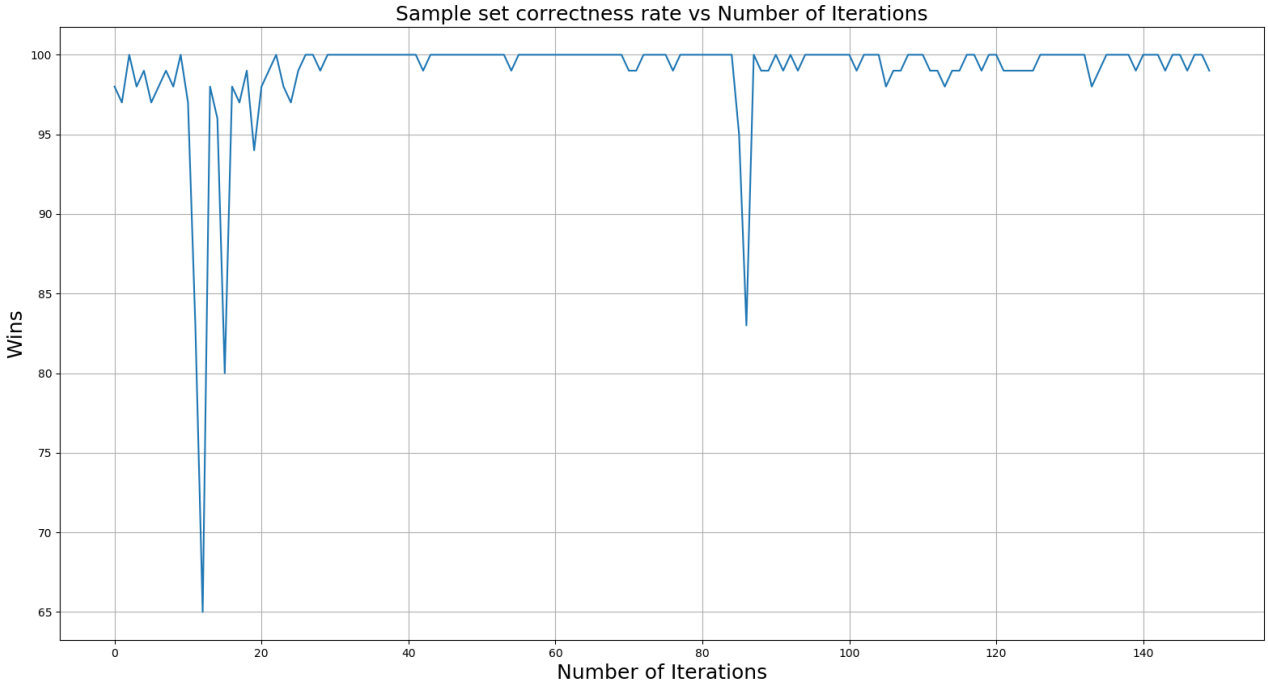

Figure 18: Moves correctly predicted for neural network with three hidden layers

Four layers:

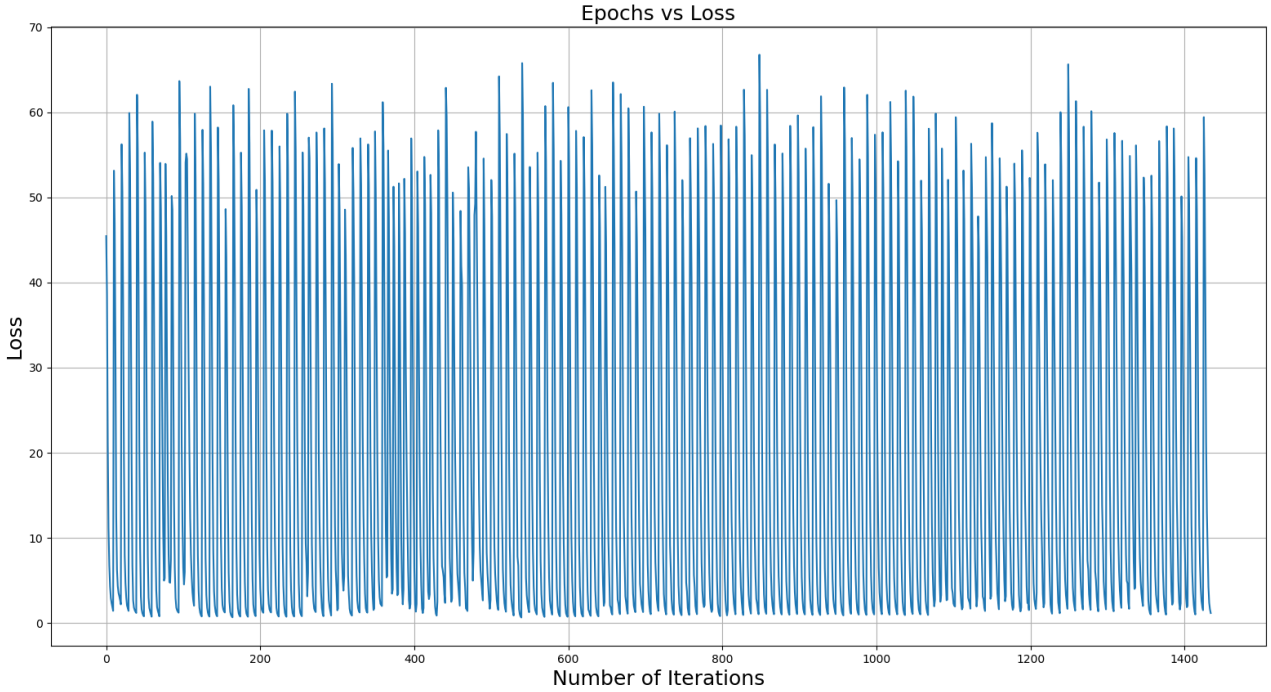

Figure 19: Loss for neural network with four hidden layers

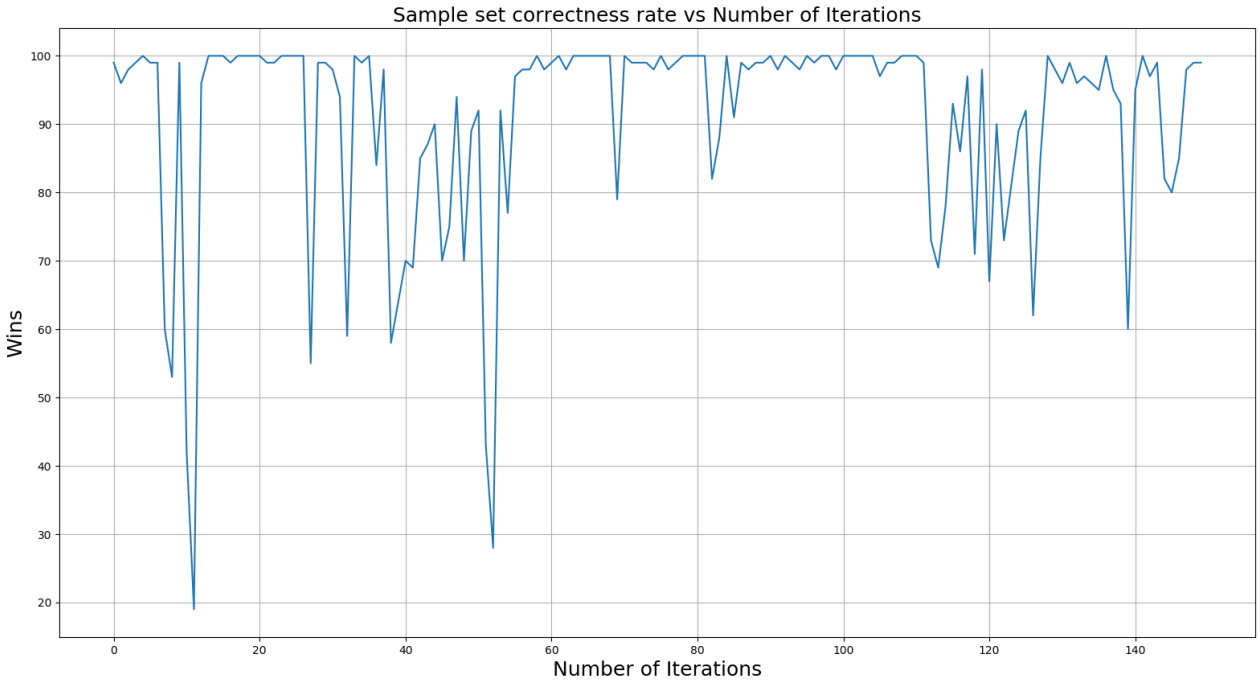

Figure 20: Moves correctly predicted for neural network with four hidden layers

Five layers:

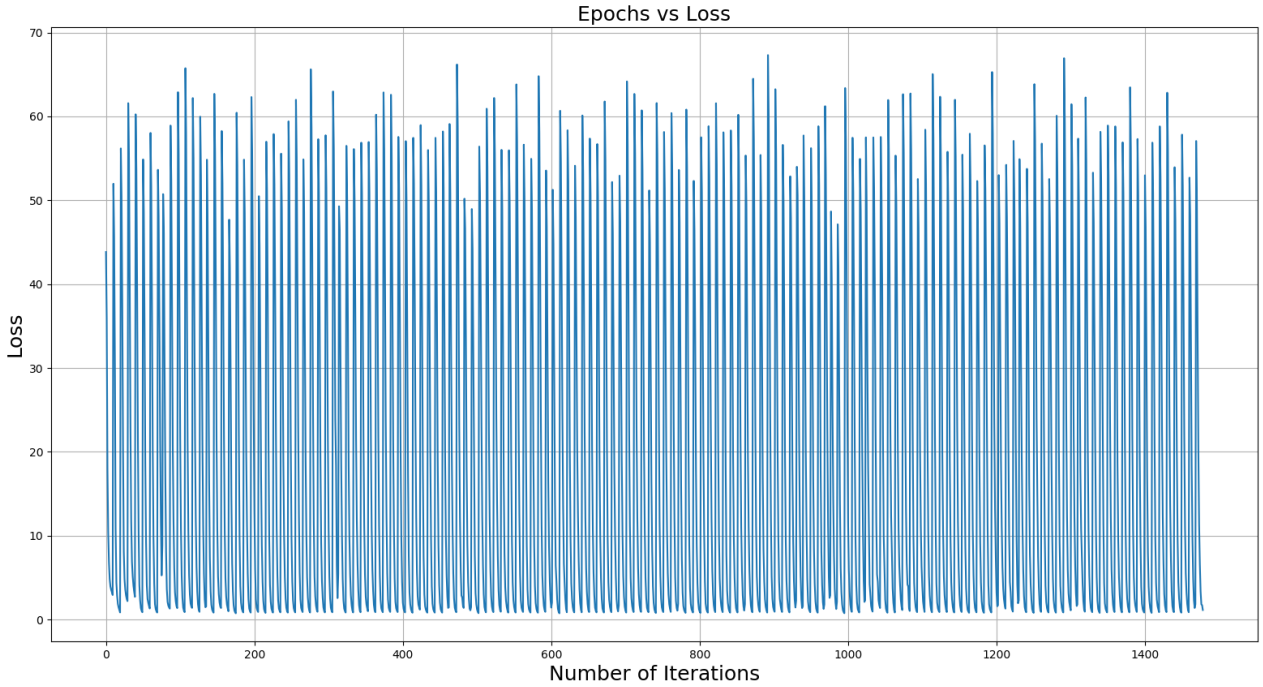

Figure 21: Loss for neural network with five hidden layers

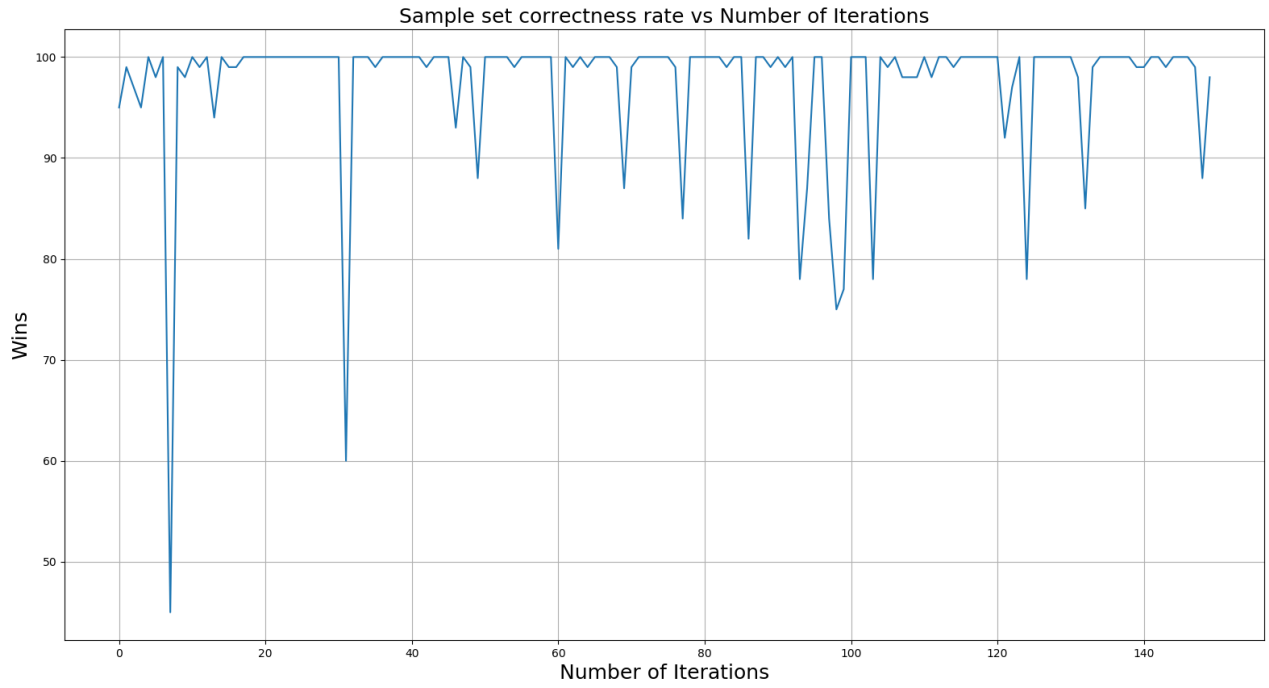

Figure 22: Moves correctly predicted for neural network with five hidden layers

6.1.3 Number of nodes in hidden layer

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed. One match of ten thousand rounds is observed for run time.

6.1.4 Number of hidden layers

- The number of hidden layers is two
- Training sample size ”N” is ”100”, ”40” for Agent test
- One hundred and fifty sample sets
- Learning rate is ”0.8”
- Weights are initialized randomly with a seed value of ”1”
- Ten epochs
- Error is 0.00001
- One match of one thousand rounds against only rock
- The number of rounds before back propagation is forty

Node sizes per layer examined: One:

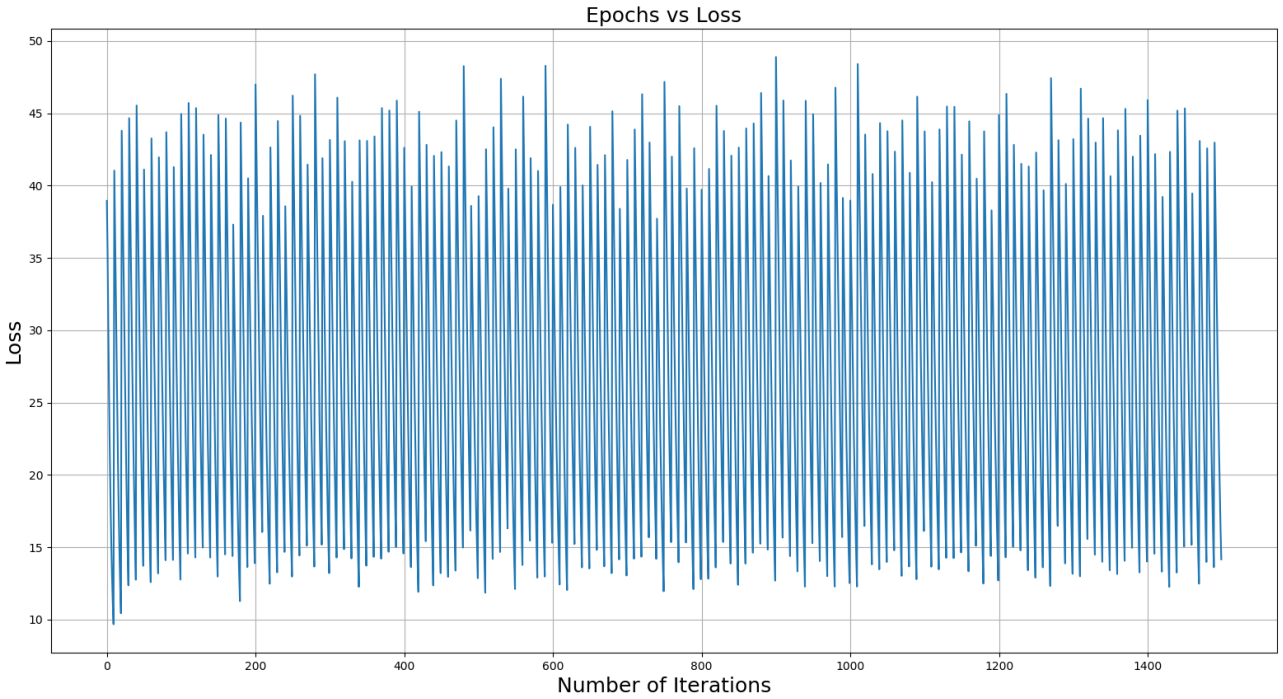

Figure 23: Loss

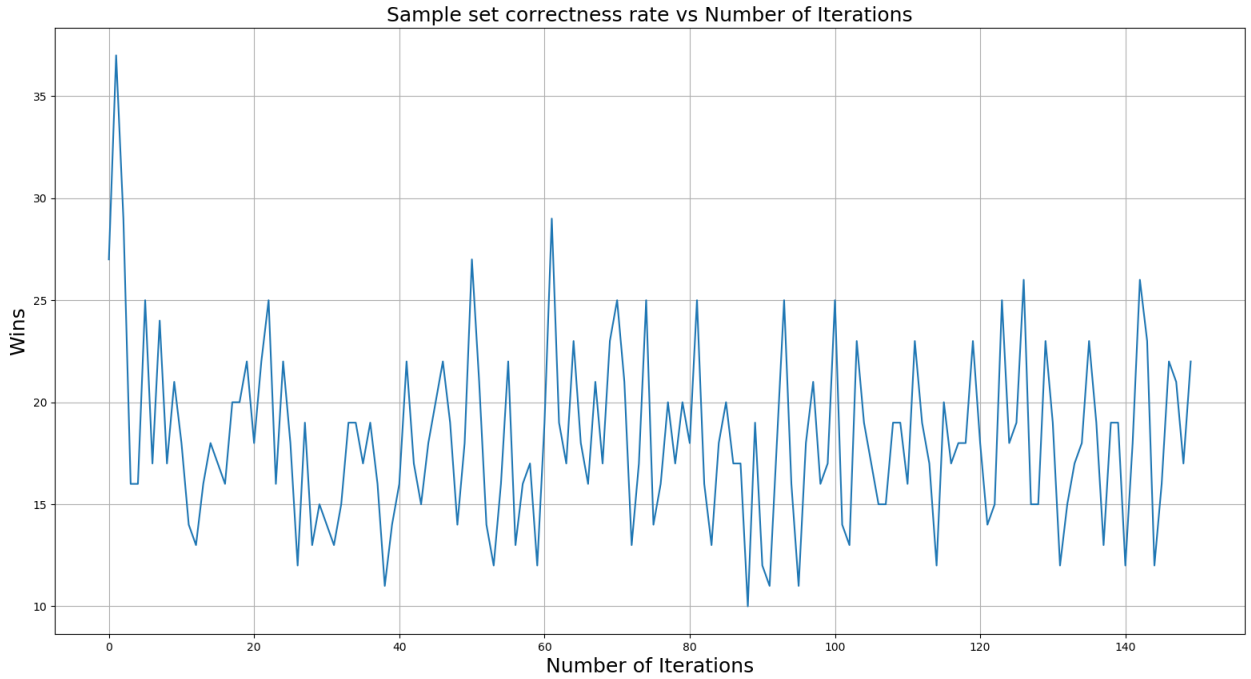

Figure 24: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
1 matches run
total run time: 0.84 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.3% of rounds (13 of 1000)
    avg score -944.0, net score -944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 95.7% of rounds (957 of 1000)
    avg score 944.0, net score 944.0
```

Figure 25: Run time

N/8:

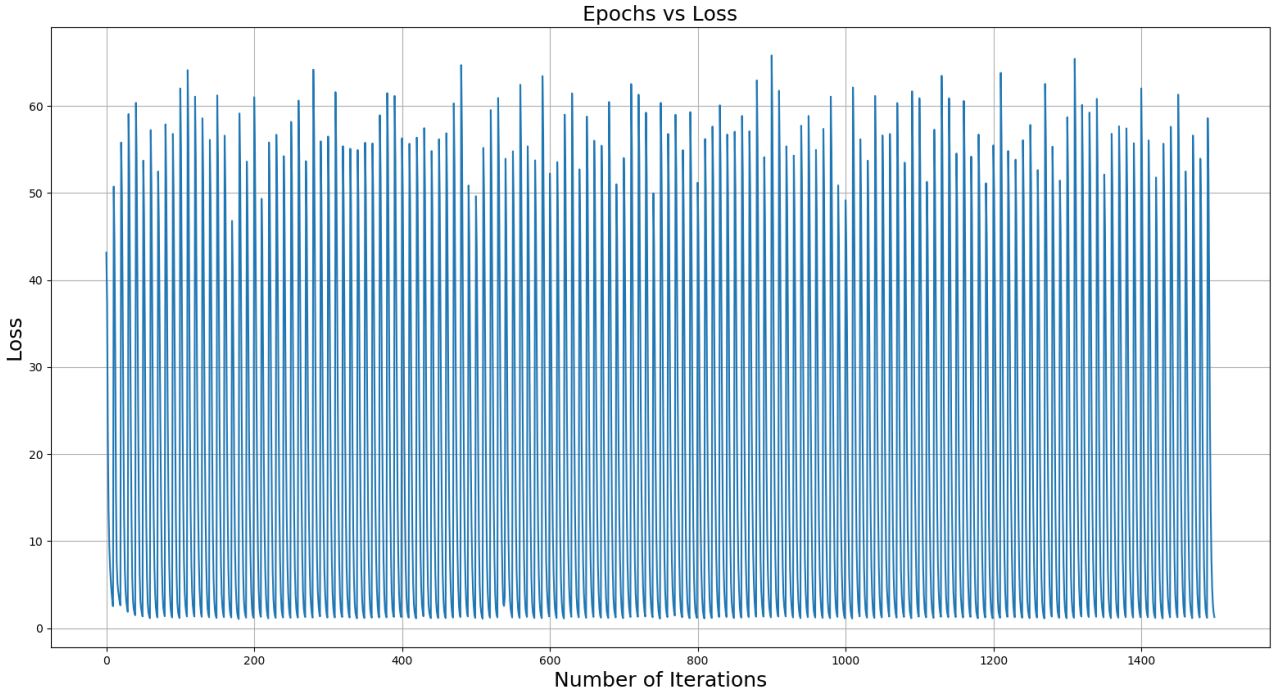

Figure 26: Loss

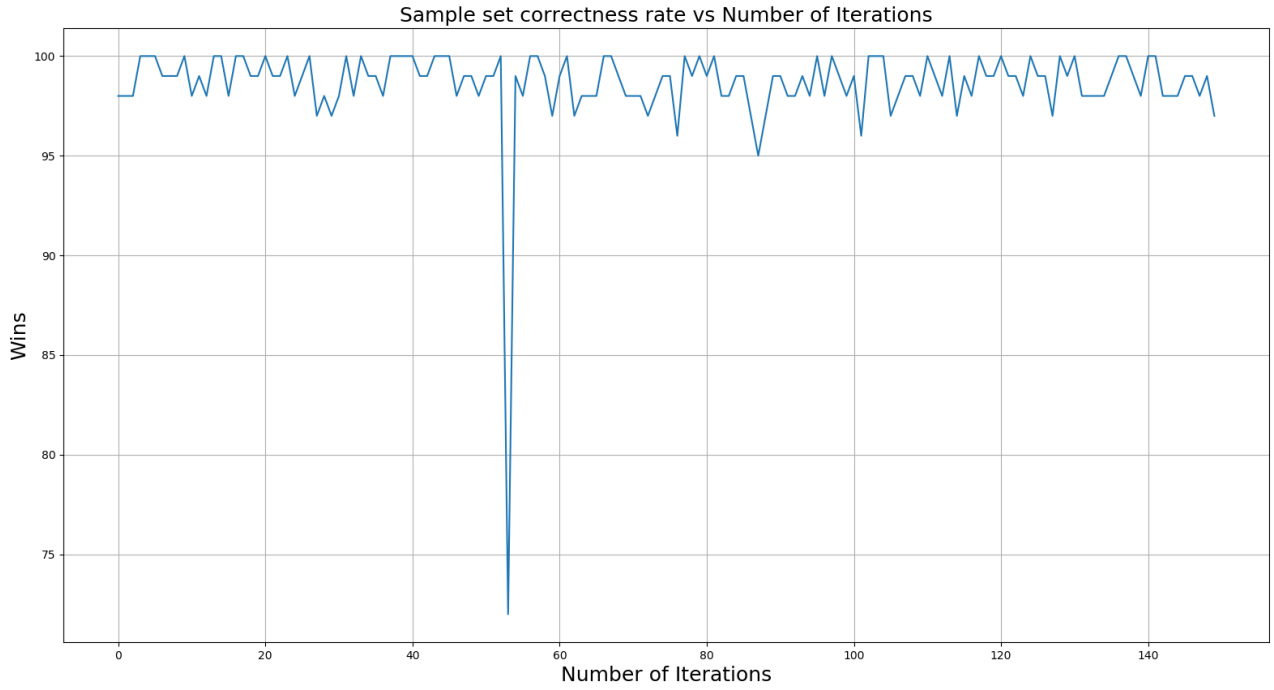

Figure 27: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
1 matches run
total run time: 0.88 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.6% of rounds (16 of 1000)
    avg score -944.0, net score -944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 96.0% of rounds (960 of 1000)
    avg score 944.0, net score 944.0
```

Figure 28: Run time

N/4:

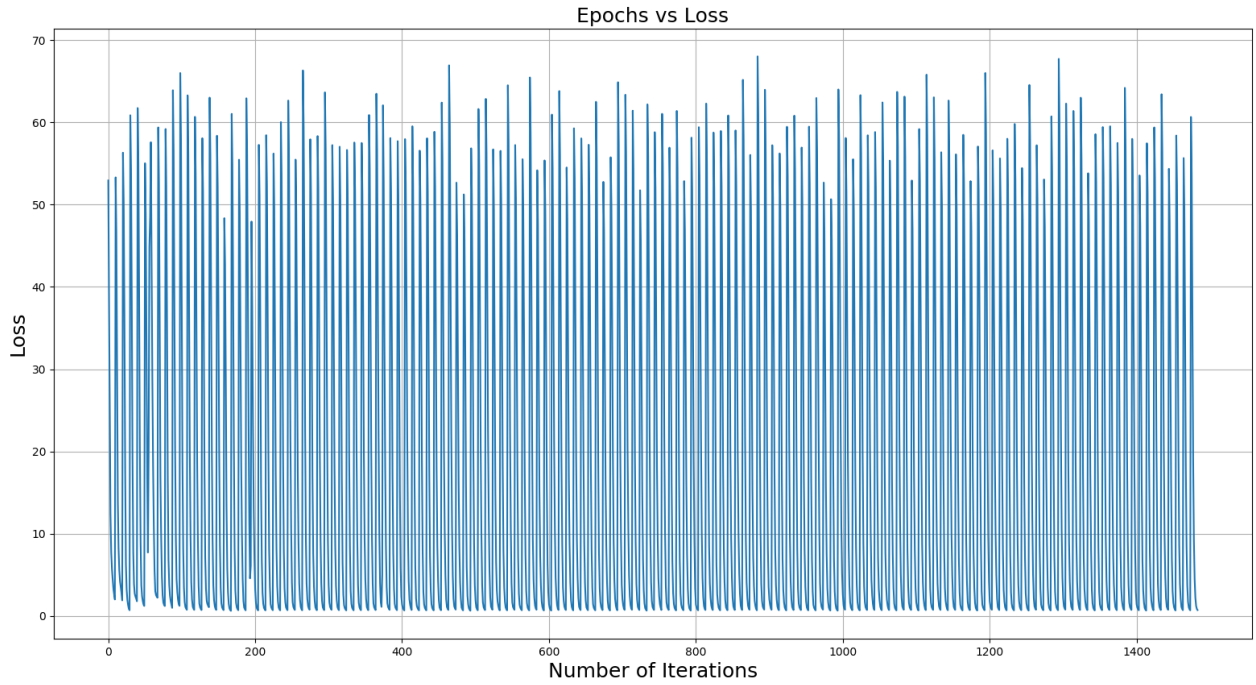

Figure 29: Loss

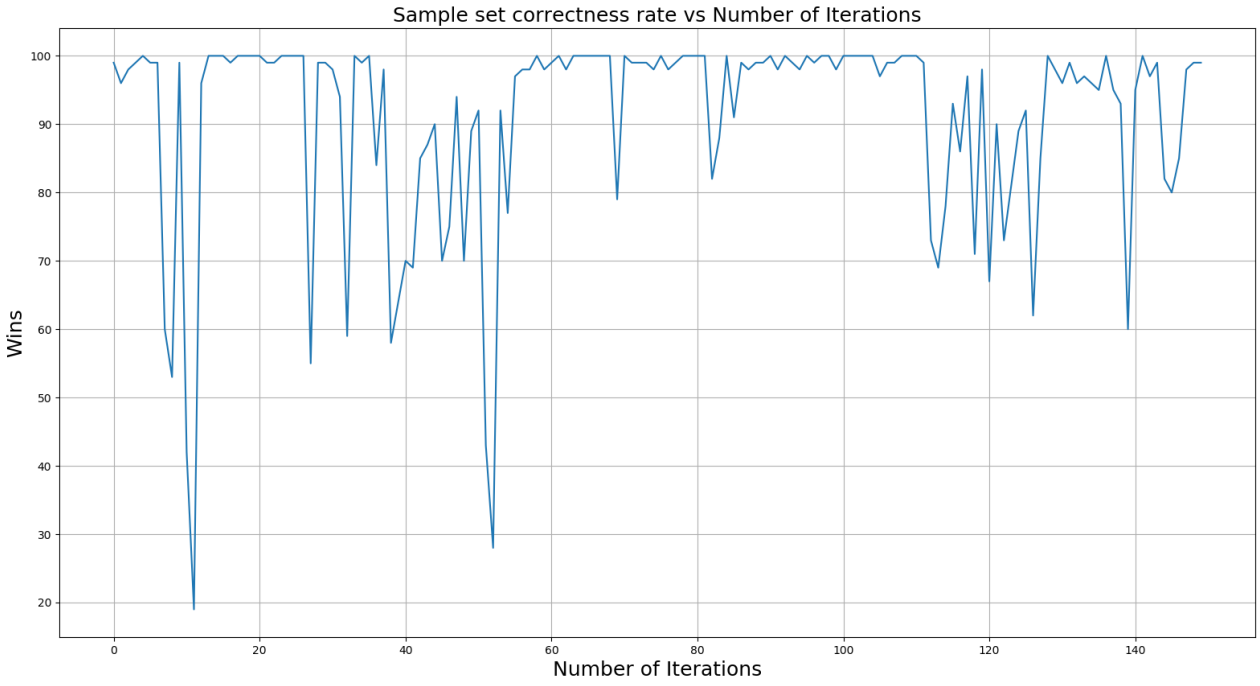

Figure 30: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overf
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 0.96 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.6% of rounds (16 of 1000)
    avg score -942.0, net score -942.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 95.8% of rounds (958 of 1000)
    avg score 942.0, net score 942.0
```

Figure 31: Run time

N/2:

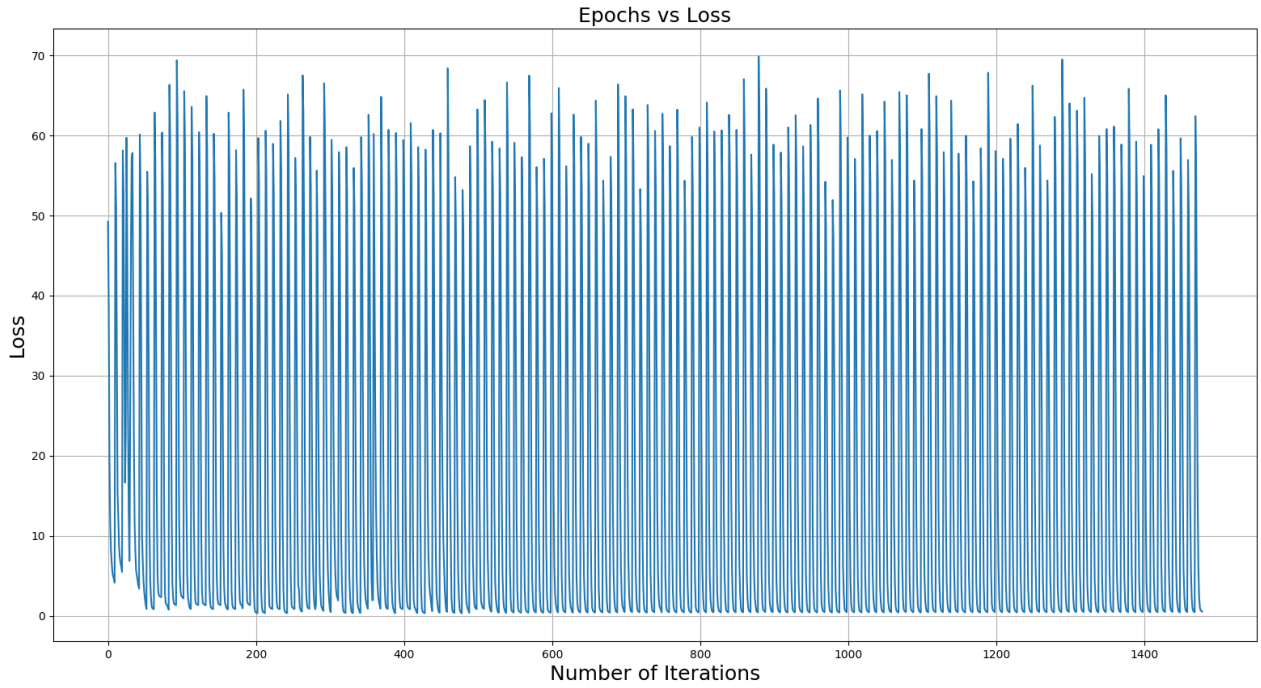

Figure 32: Loss

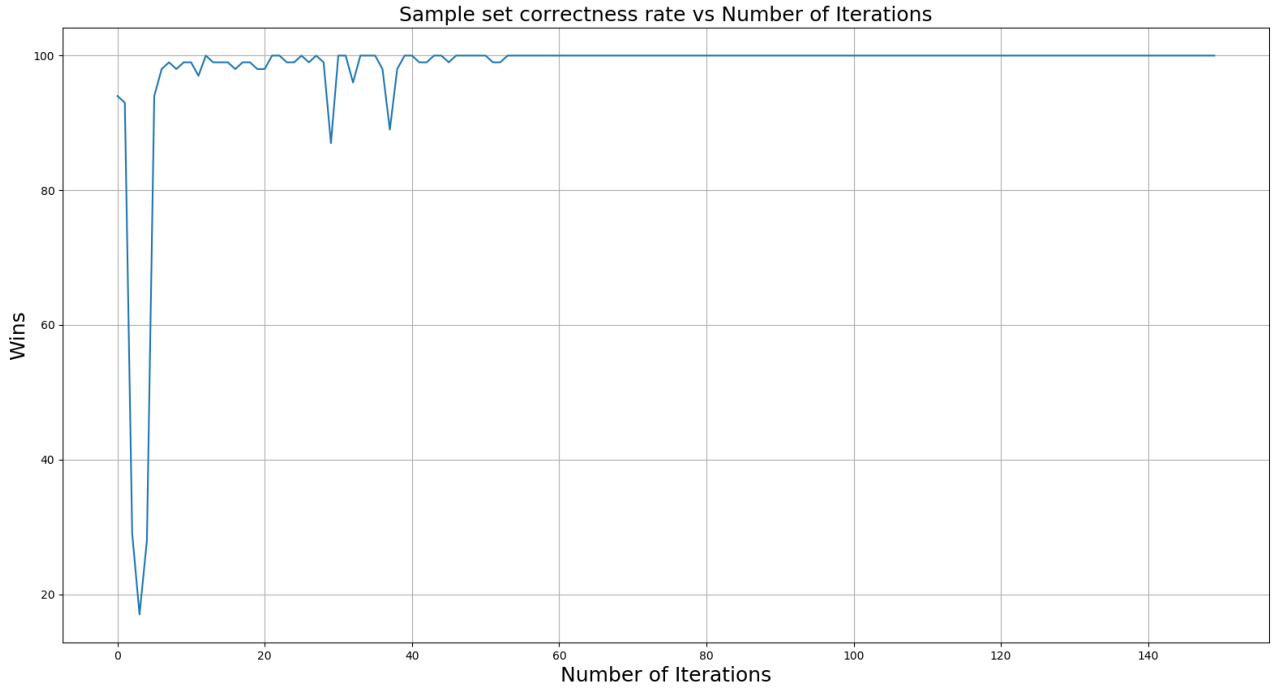

Figure 33: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overf
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.17 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.5% of rounds (15 of 1000)
    avg score -941.0, net score -941.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 95.6% of rounds (956 of 1000)
    avg score 941.0, net score 941.0
```

Figure 34: Run time

N:

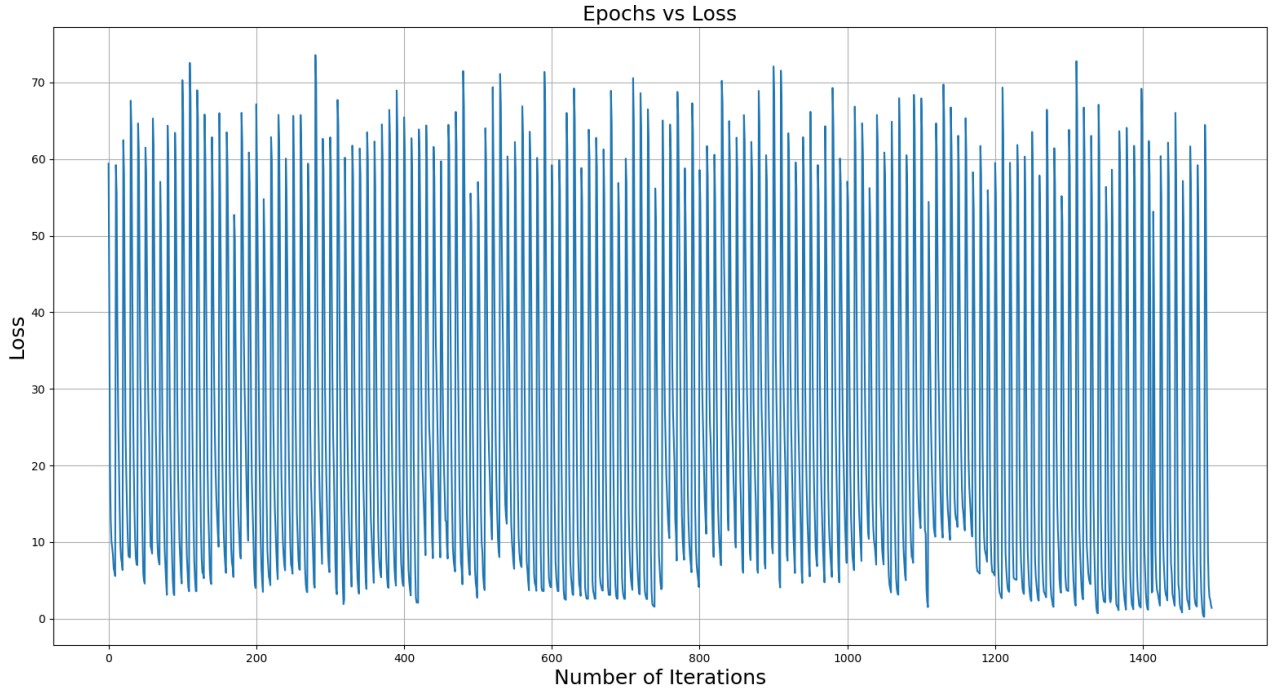

Figure 35: Loss

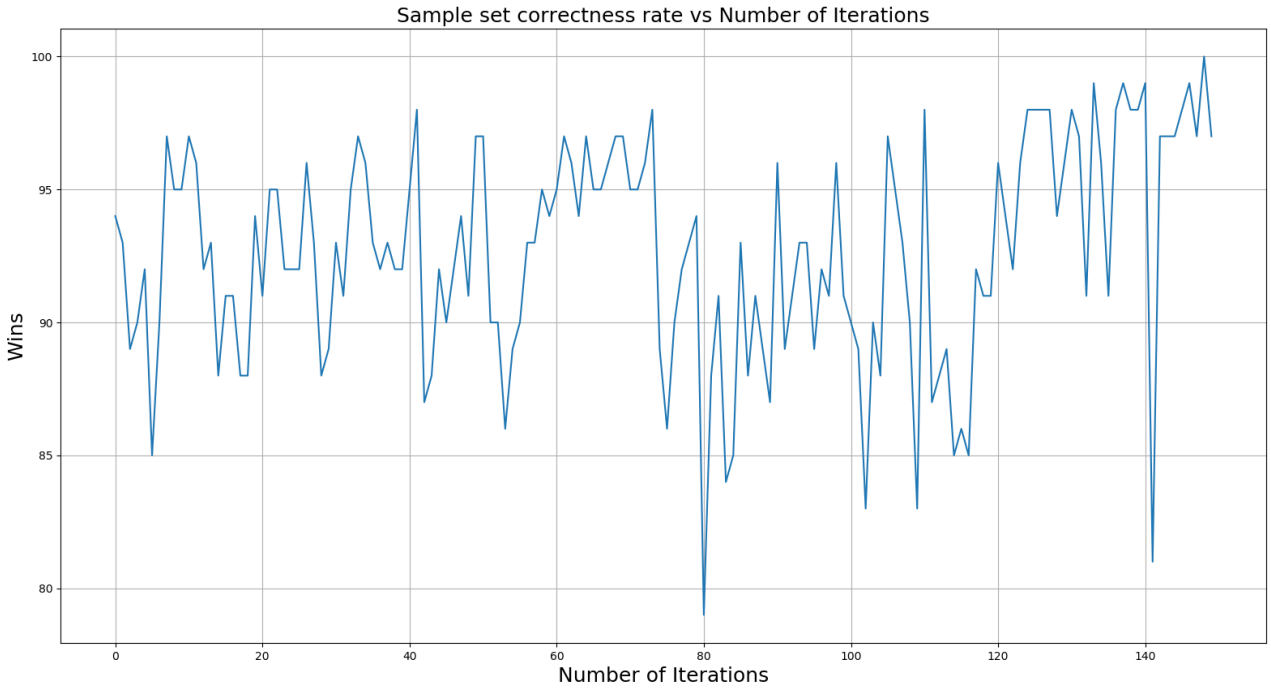

Figure 36: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: over
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.41 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.5% of rounds (15 of 1000)
    avg score -941.0, net score -941.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 95.6% of rounds (956 of 1000)
    avg score 941.0, net score 941.0
```

Figure 37: Run time

2N:

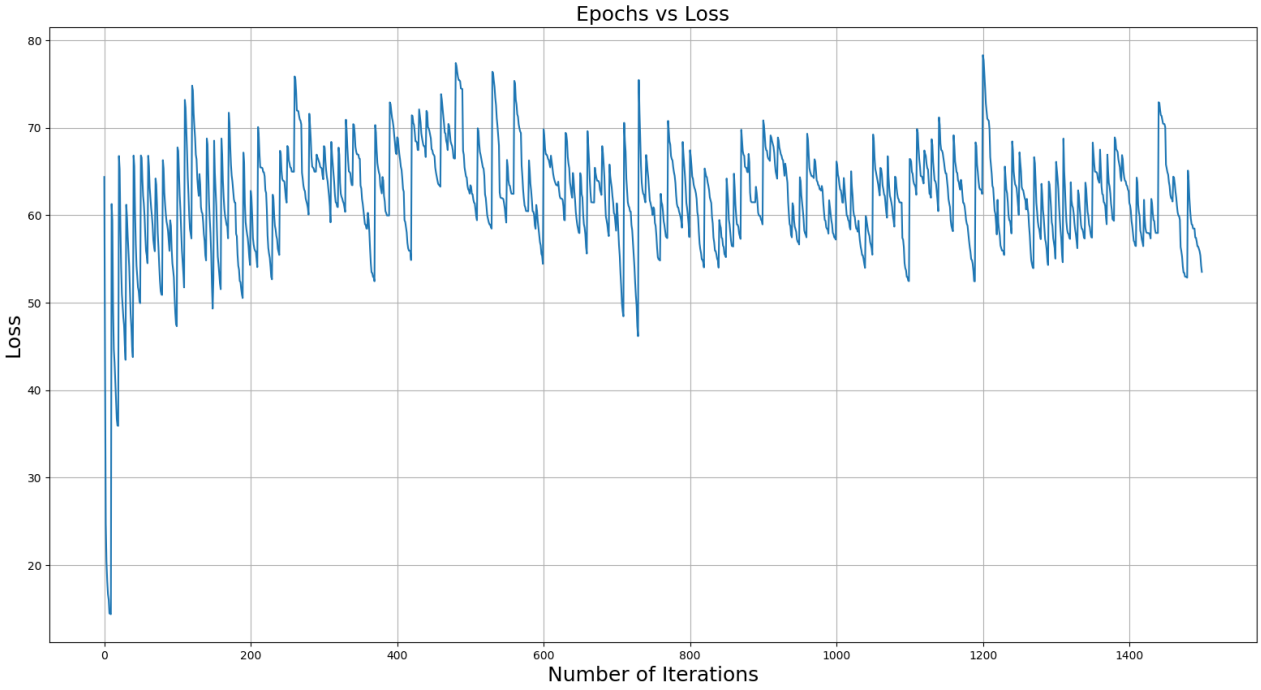

Figure 38: Loss

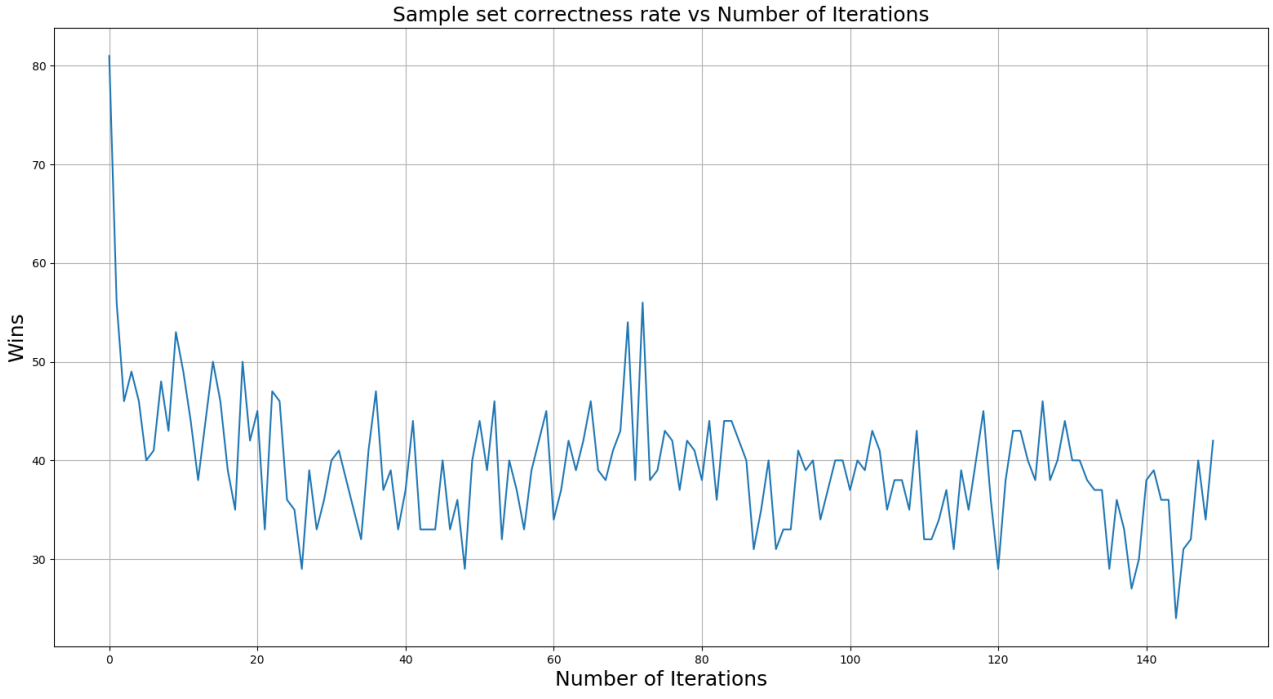

Figure 39: Moves correctly predicted for N/8 nodes

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 1000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overf
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.91 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.6% of rounds (16 of 1000)
    avg score -894.0, net score -894.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 91.0% of rounds (910 of 1000)
    avg score 894.0, net score 894.0
```

Figure 40: Run time

6.1.5 Learning rate

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed. One match of ten thousand rounds is observed for run time.

- The number of hidden layers is two
- Training sample size ”N” is sixty
- One hundred and fifty sample sets
- Number of hidden layer nodes is Sample sets / 8
- Weights are initialized randomly with a seed value of ”1”
- Ten epochs
- Error is 0.00001
- One match of ten thousand rounds against only rock
- The number of rounds before back propagation is forty

Learning rates examined: LR = 0.01:

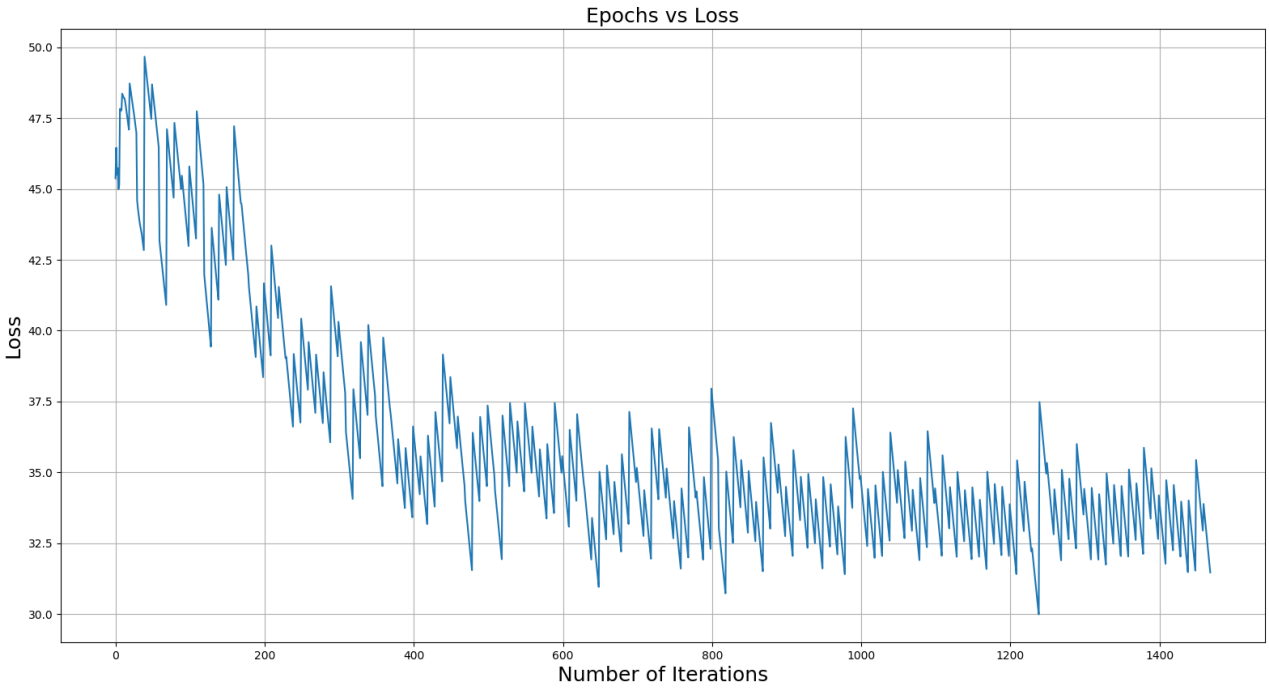

Figure 41: Loss

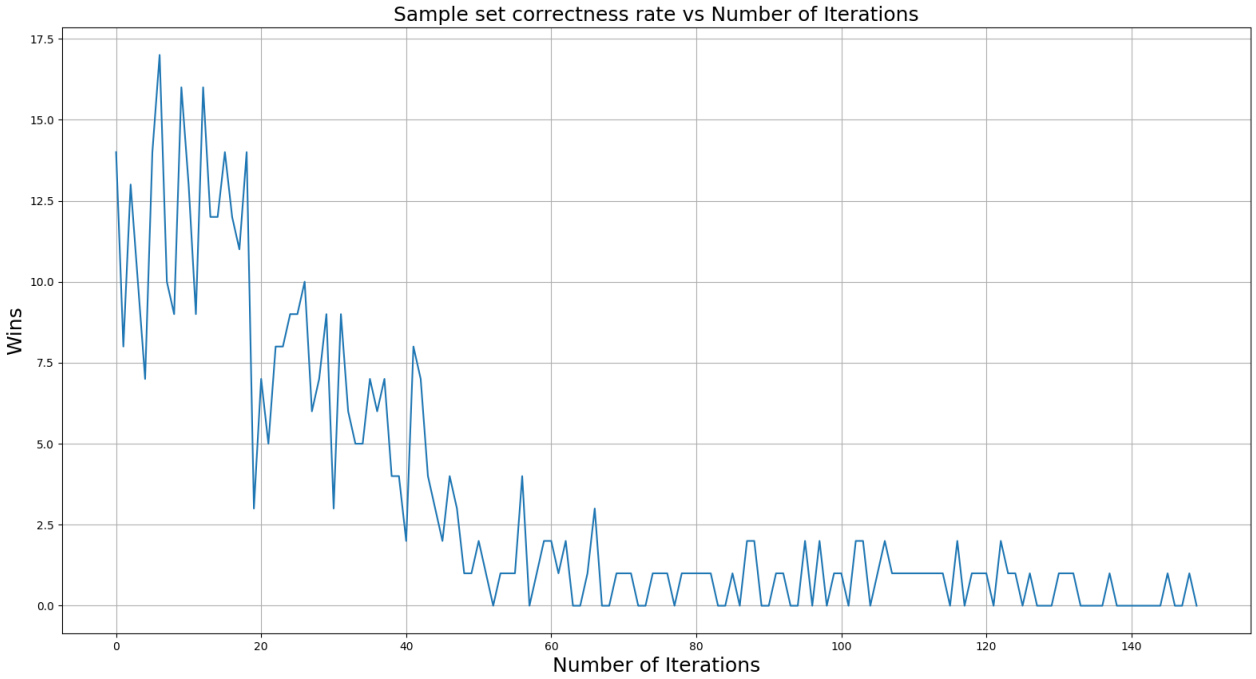

Figure 42: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
1 matches run
total run time: 4.12 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 1.1% of rounds (111 of 10000)
    avg score -9370.0, net score -9370.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 94.8% of rounds (9481 of 10000)
    avg score 9370.0, net score 9370.0
```

Figure 43: Run time

LR = 0.1:

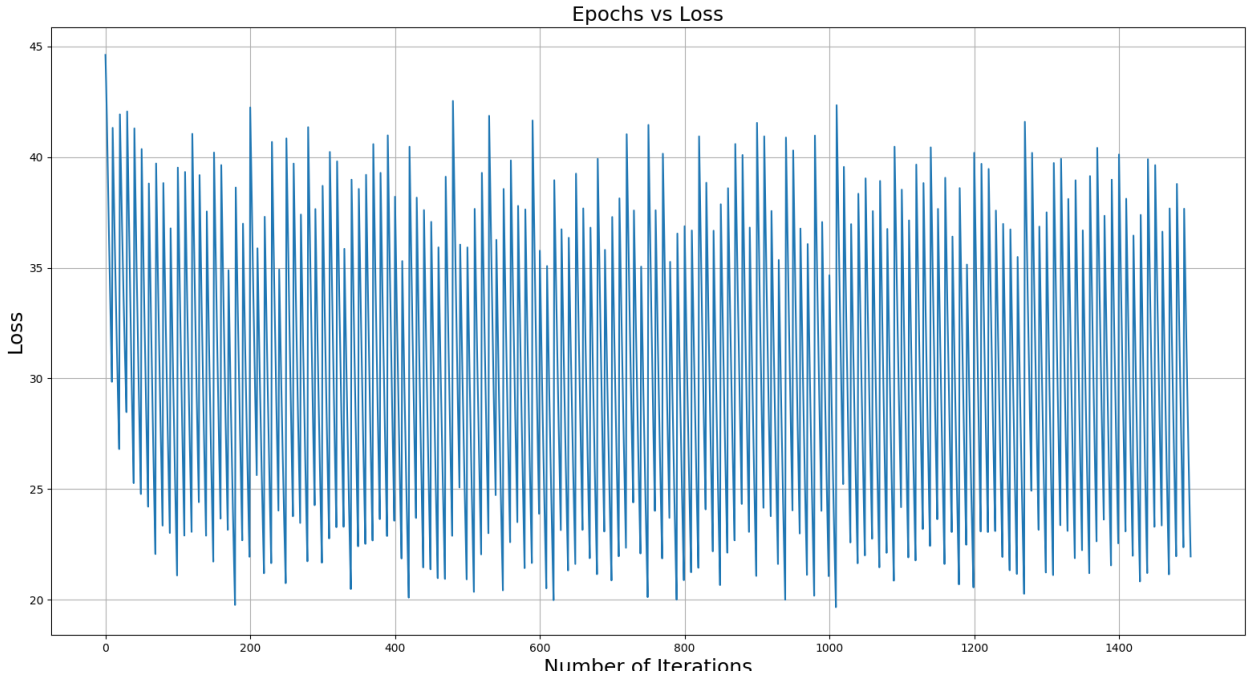

Figure 44: Loss

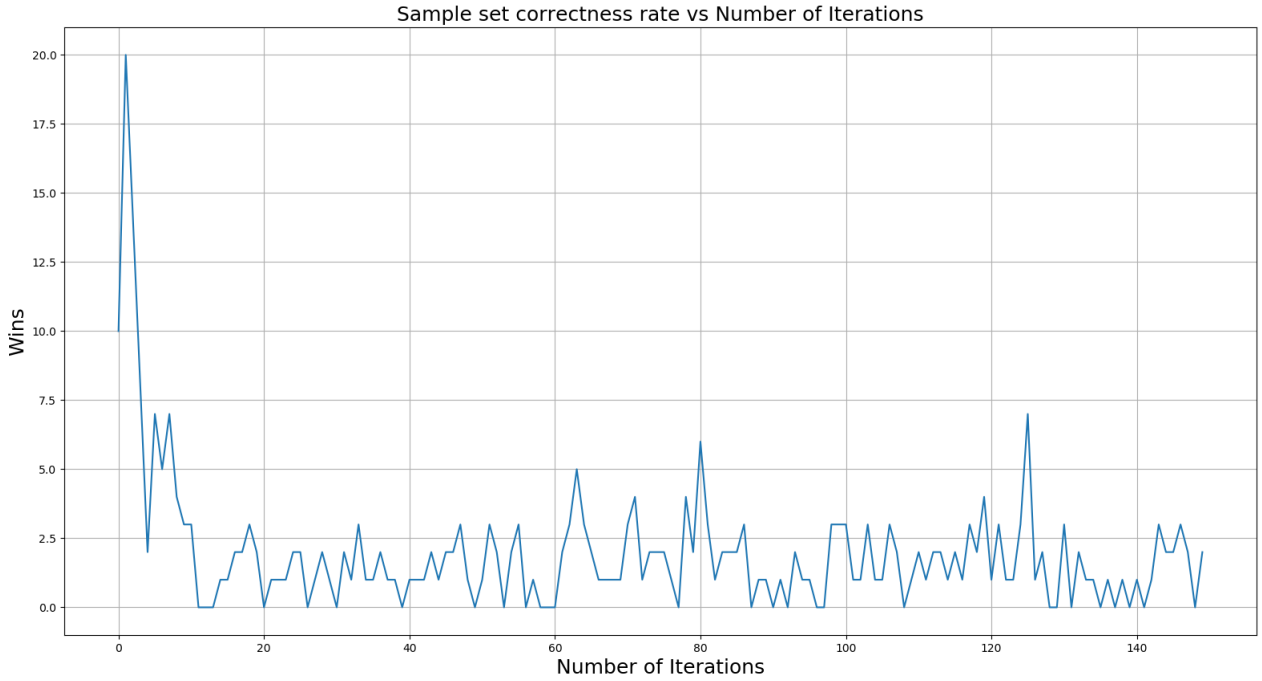

Figure 45: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
1 matches run
total run time: 2.37 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (23 of 10000)
    avg score -9899.0, net score -9899.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.2% of rounds (9922 of 10000)
    avg score 9899.0, net score 9899.0
```

Figure 46: Run time

LR = 0.8:

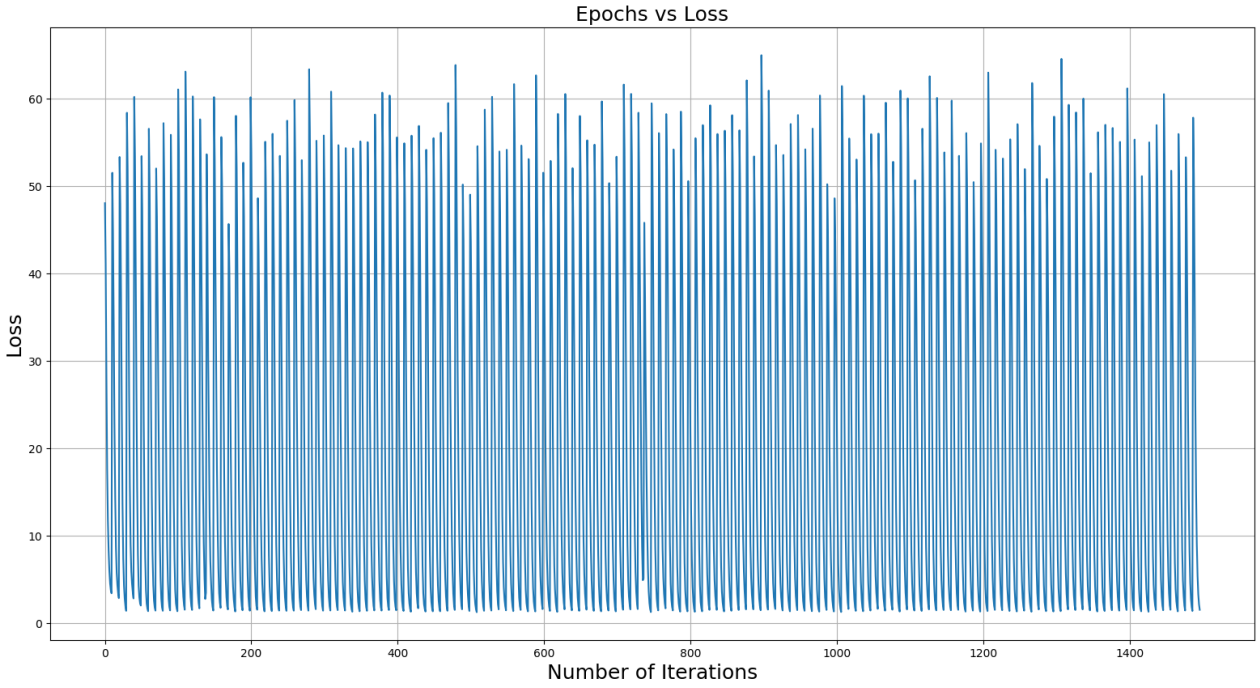

Figure 47: Loss

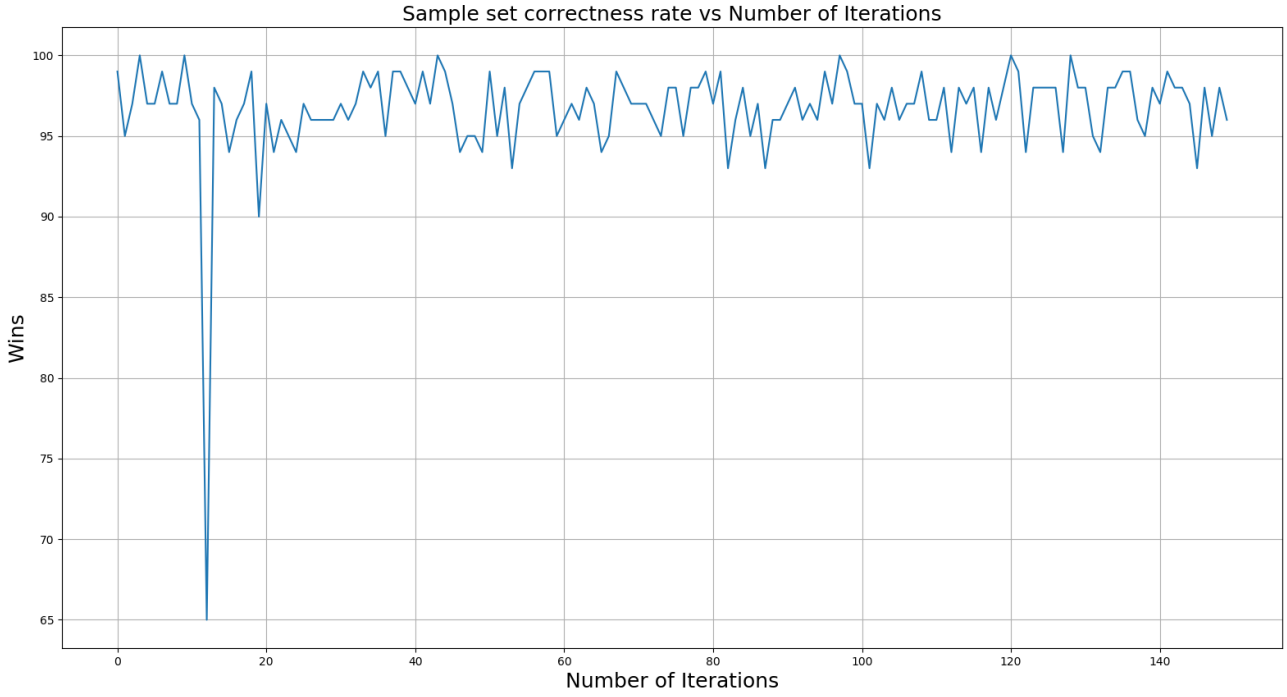

Figure 48: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
1 matches run
total run time: 1.76 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (16 of 10000)
    avg score -9944.0, net score -9944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.6% of rounds (9960 of 10000)
    avg score 9944.0, net score 9944.0
```

Figure 49: Run time

LR = 5:

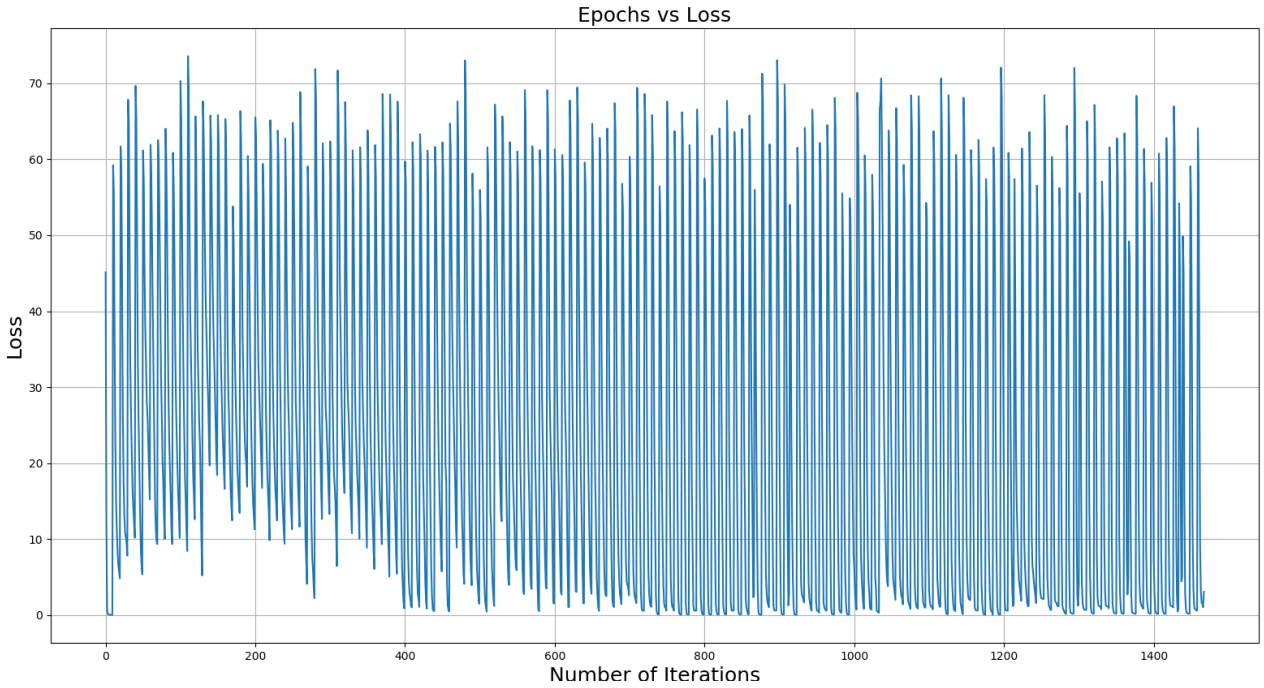

Figure 50: Loss

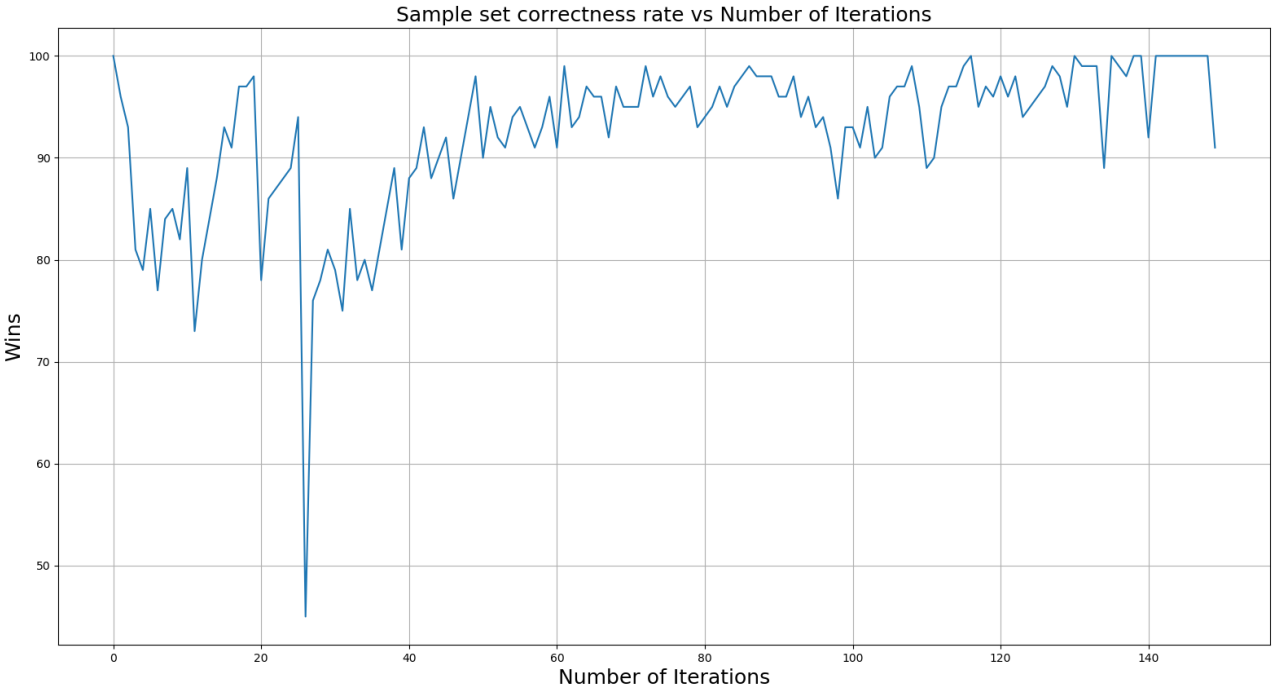

Figure 51: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overfl
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.38 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (16 of 10000)
    avg score -9944.0, net score -9944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.6% of rounds (9960 of 10000)
    avg score 9944.0, net score 9944.0
```

Figure 52: Run time

LR = 10:

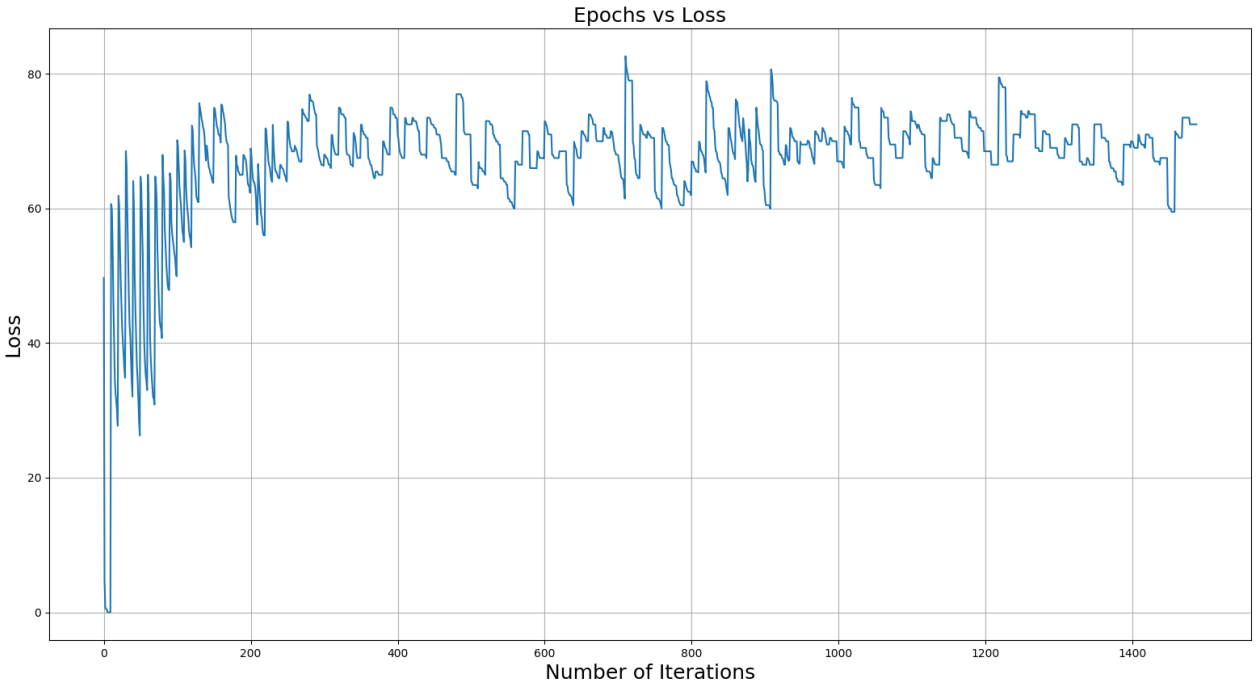

Figure 53: Loss

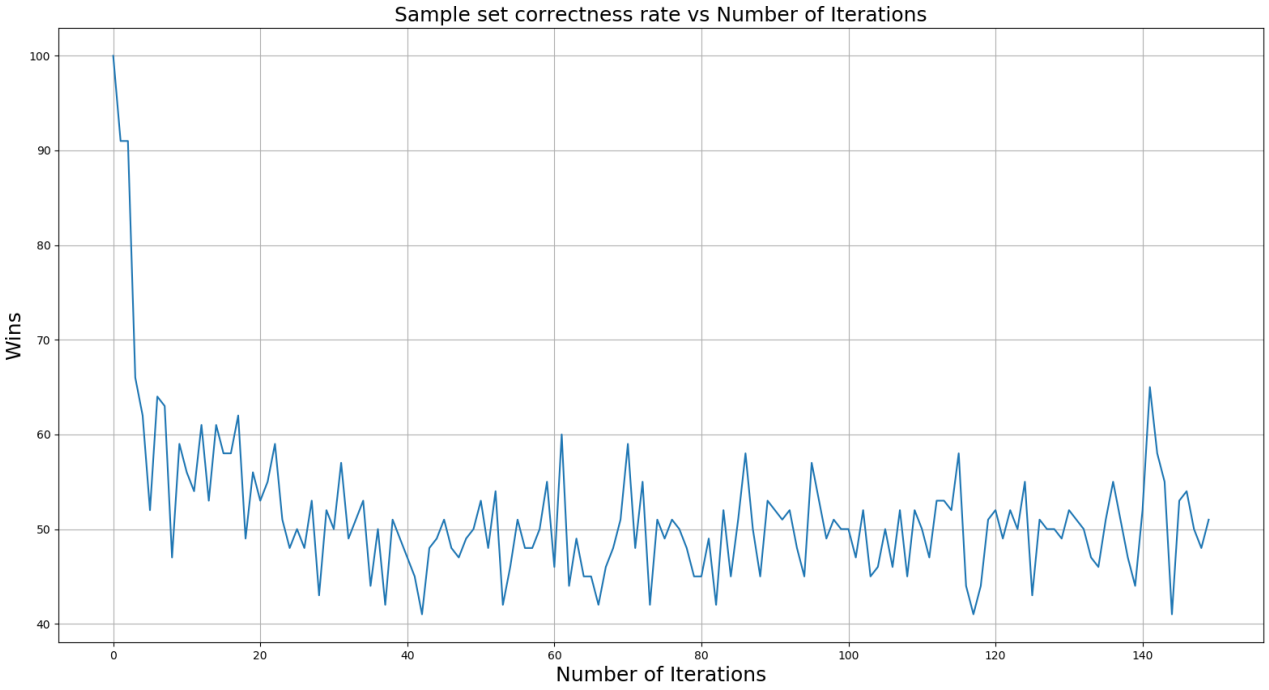

Figure 54: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overfl
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.36 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (16 of 10000)
    avg score -9944.0, net score -9944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.6% of rounds (9960 of 10000)
    avg score 9944.0, net score 9944.0
```

Figure 55: Run time

LR = 25:

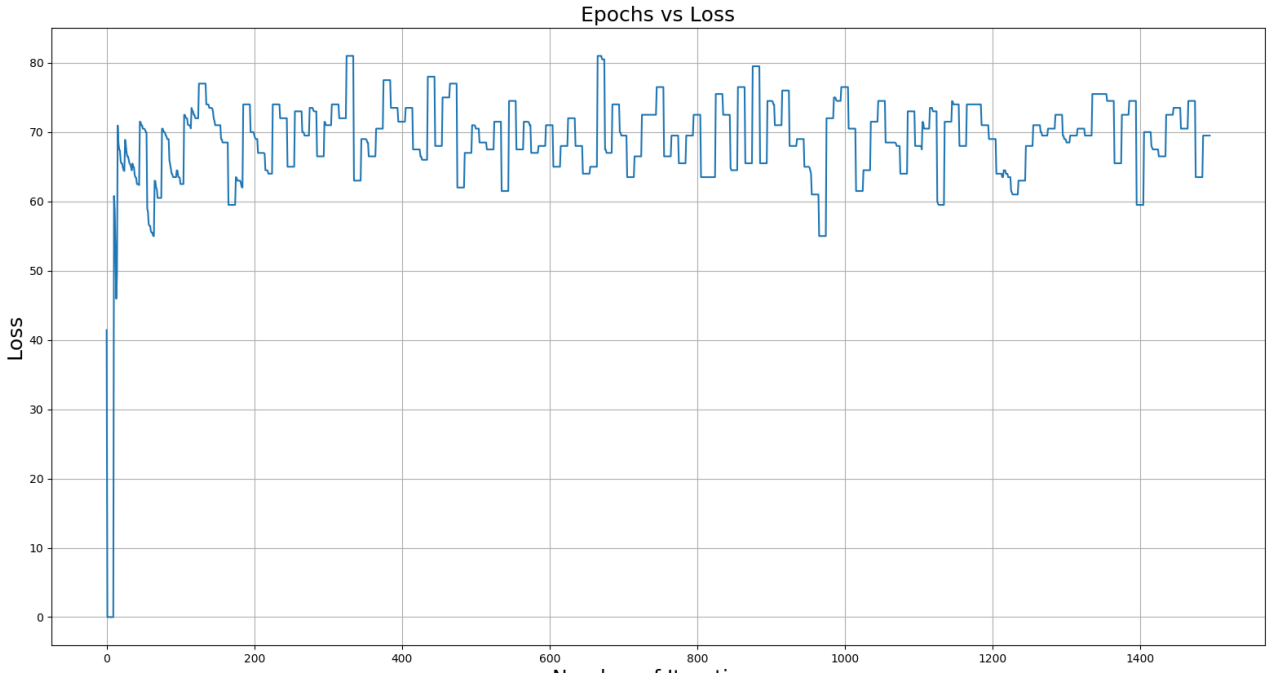

Figure 56: Loss

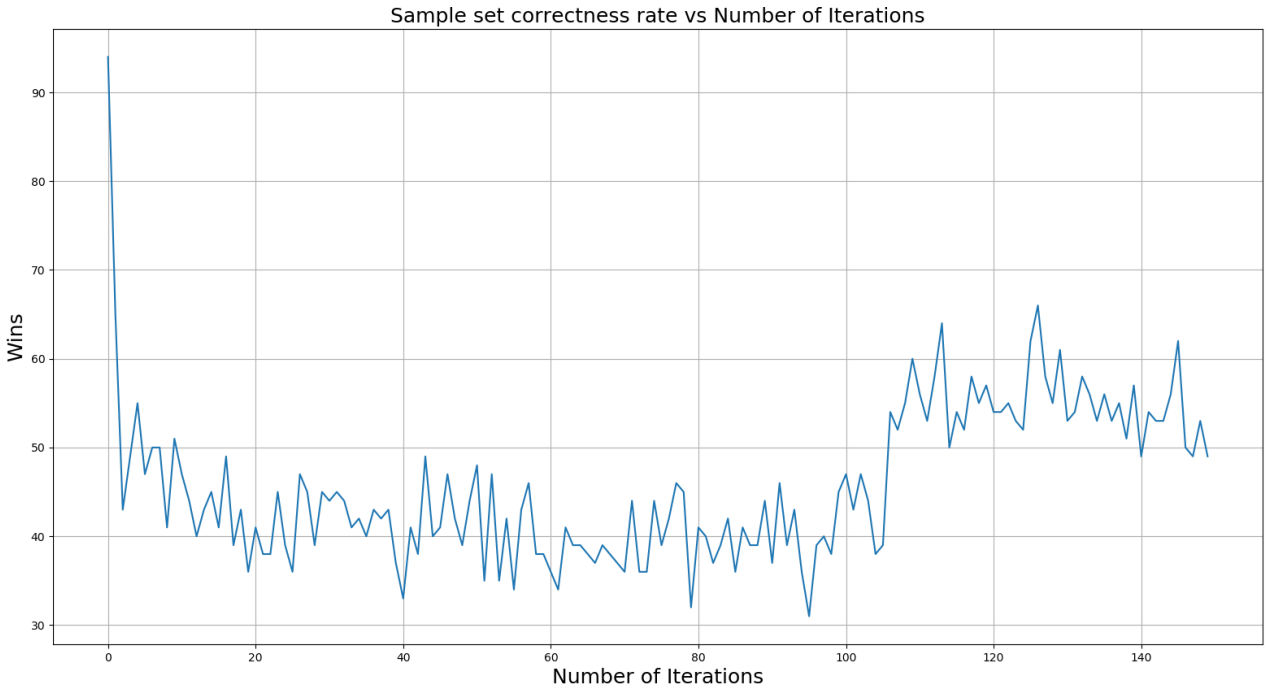

Figure 57: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overfl
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.30 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (16 of 10000)
    avg score -9944.0, net score -9944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.6% of rounds (9960 of 10000)
    avg score 9944.0, net score 9944.0
```

Figure 58: Run time

LR = 100:

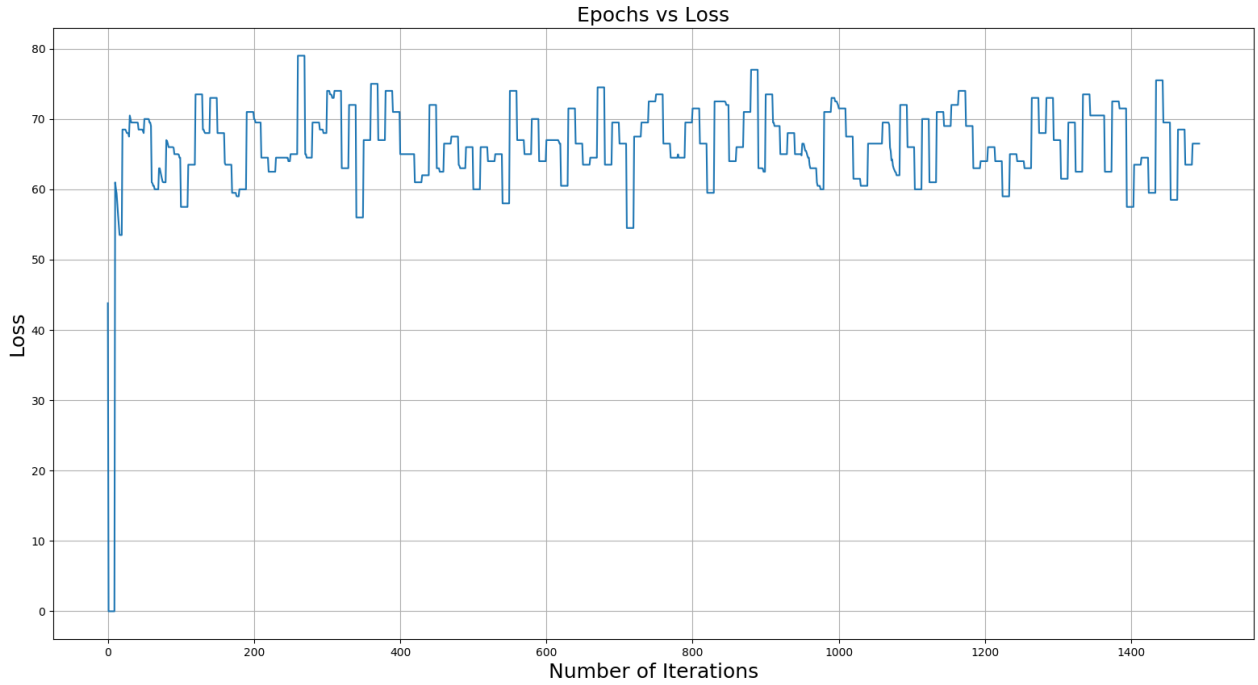

Figure 59: Loss

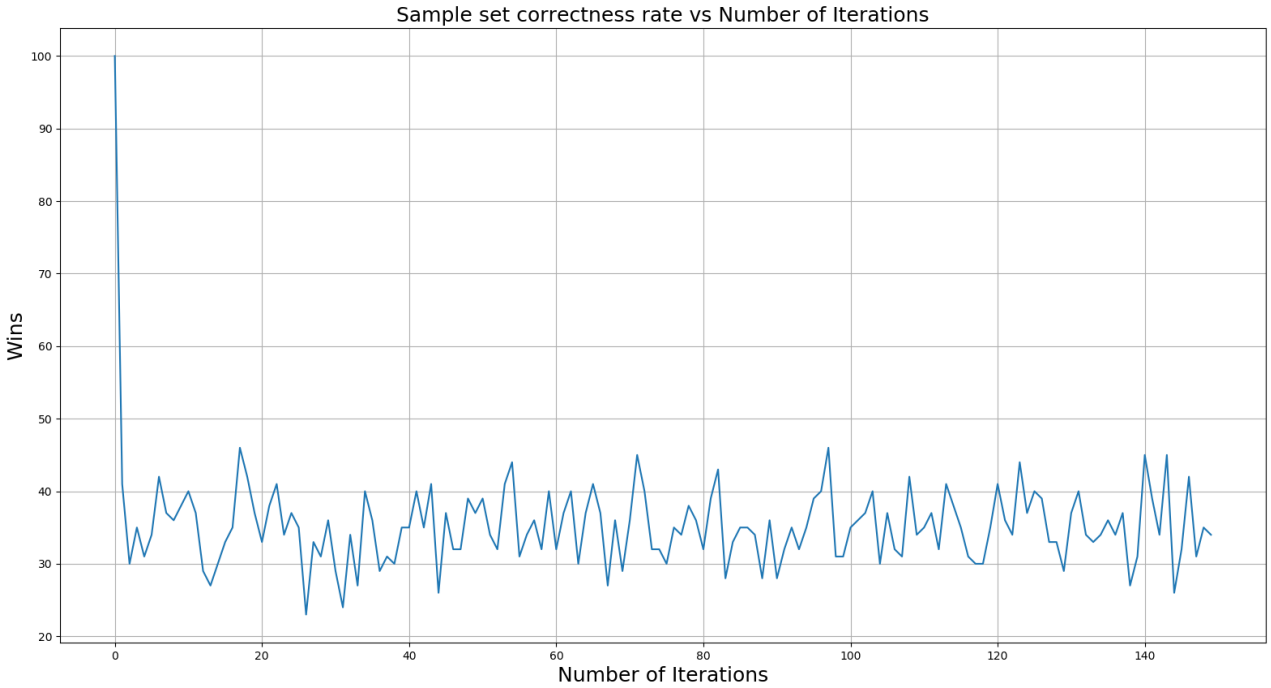

Figure 60: Moves correctly predicted

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 1 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 1 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overfl
  sum = 1 / (1 + np.e**-z)
1 matches run
total run time: 1.30 seconds

only_rock.py: won 0.0% of matches (0 of 1)
    won 0.2% of rounds (16 of 10000)
    avg score -9944.0, net score -9944.0

Agent_NN.py: won 100.0% of matches (1 of 1)
    won 99.6% of rounds (9960 of 10000)
    avg score 9944.0, net score 9944.0
```

Figure 61: Run time

6.1.6 Number of learning sample sets

The computed Loss per epoch and the number of moves correctly predicted per sample set over multiple training sets are observed.

- The number of hidden layers is two
- Training sample size ”N” is sixty
- Learning rate is set to ”0.8”
- Number of hidden layer nodes is Sample sets / 8
- Weights are initialized randomly with a seed value of ”1”
- Ten epochs
- Error is 0.00001

Batch sizes examined examined: Size = 1:

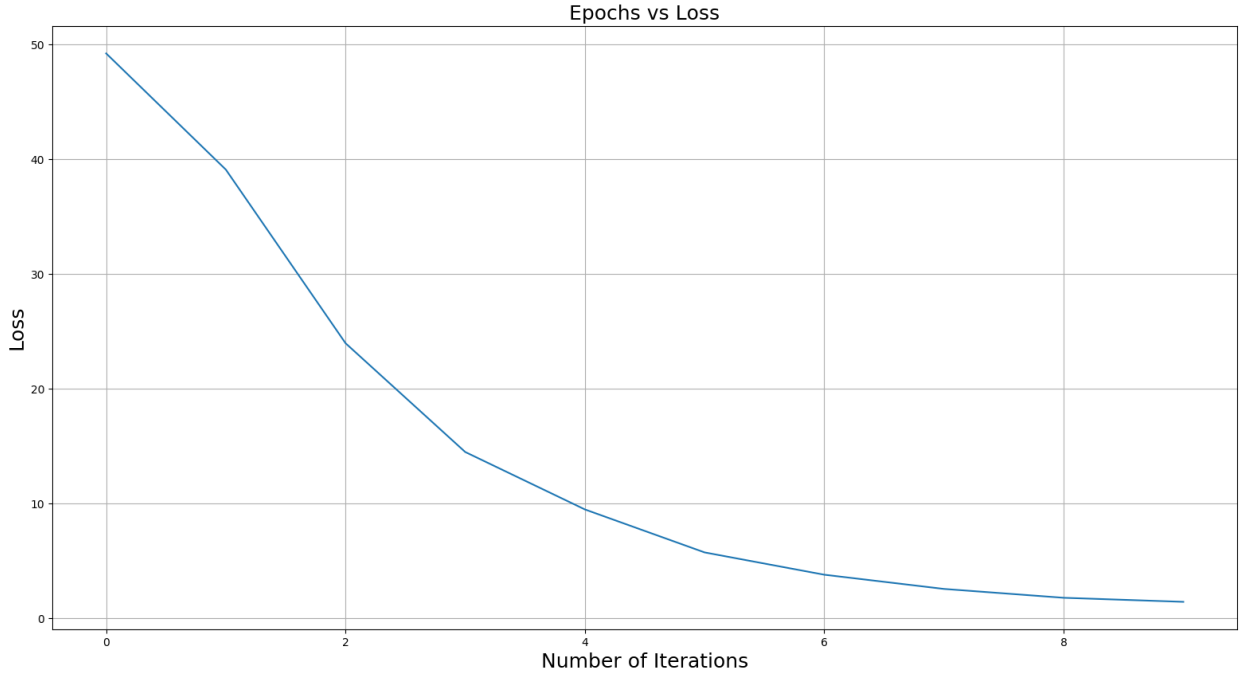

Figure 62: Loss

Size = 10:

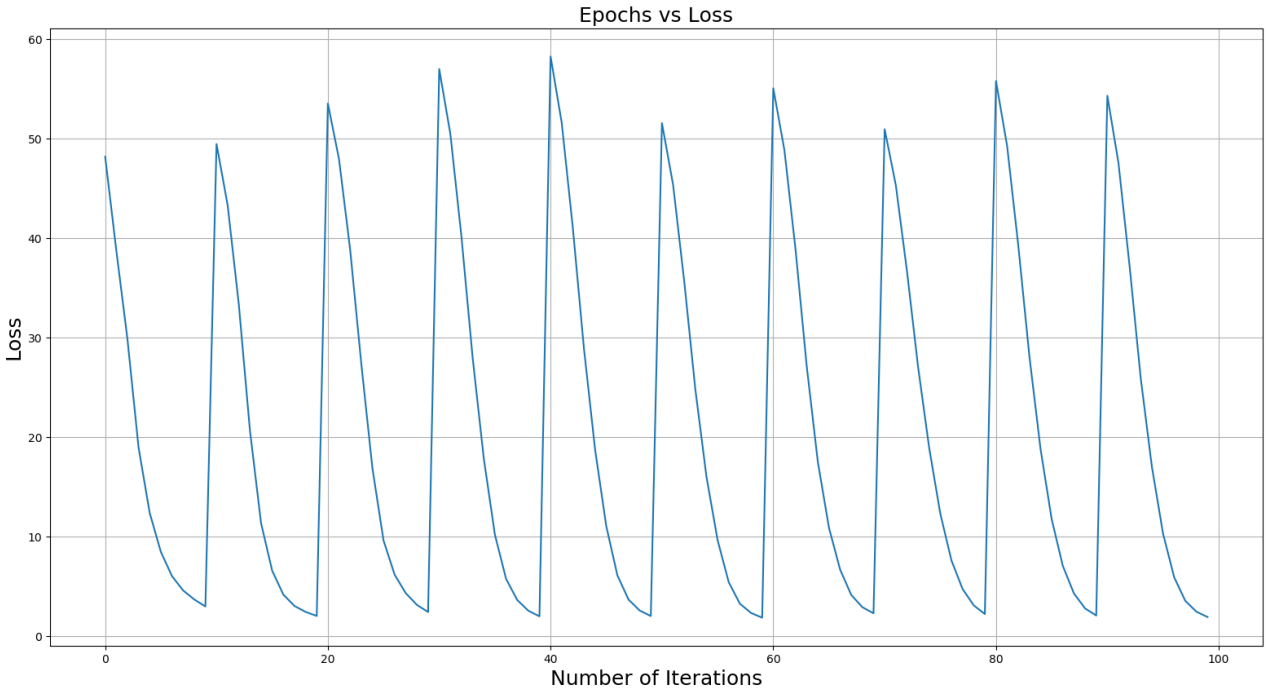

Figure 63: Loss

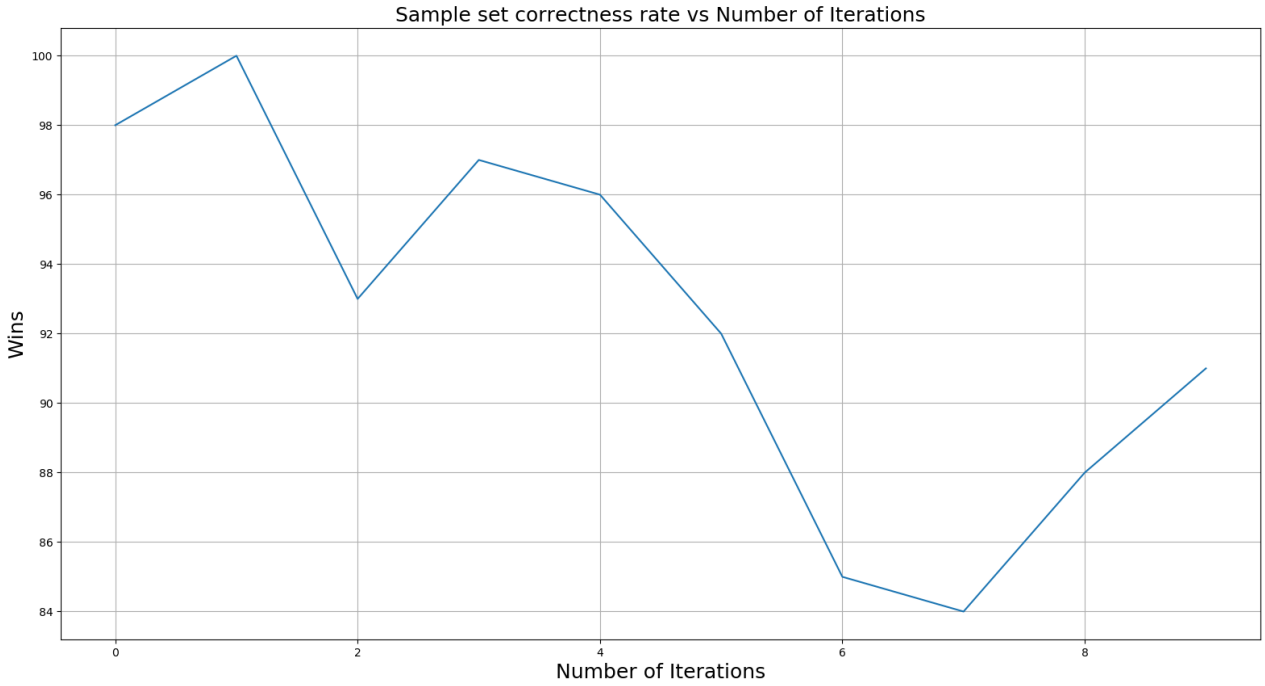

Figure 64: Moves correctly predicted

Size = 50:

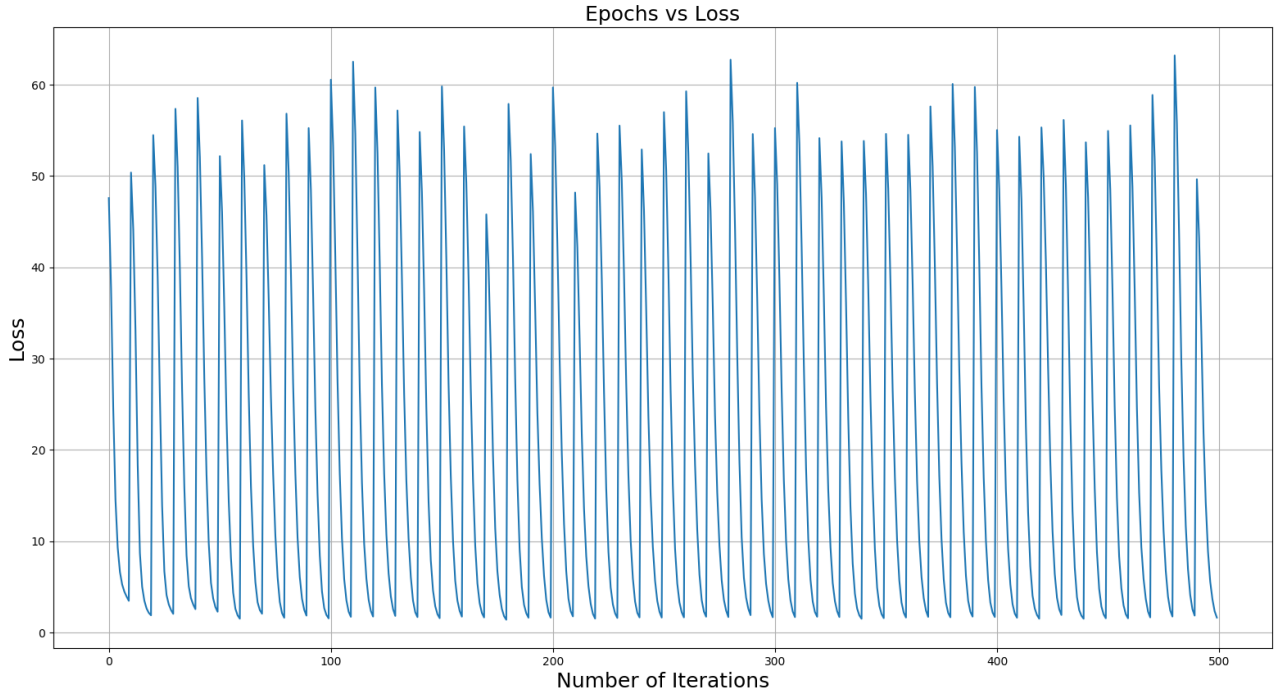

Figure 65: Loss

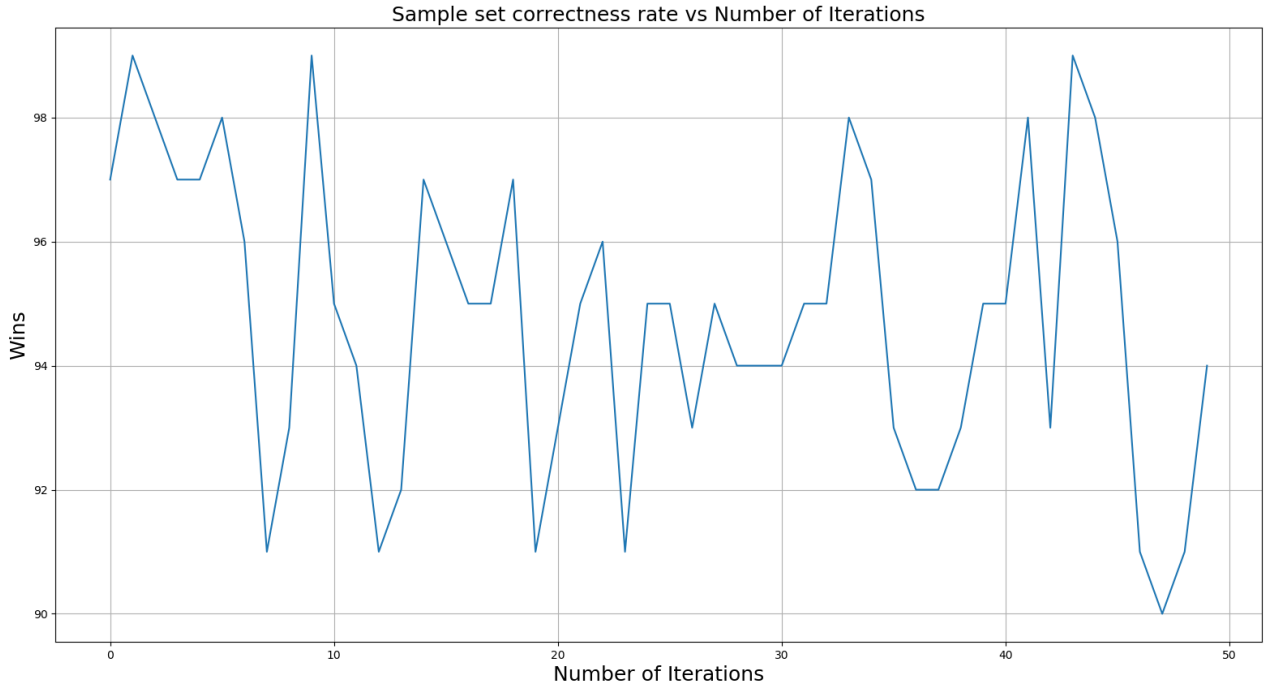

Figure 66: Moves correctly predicted

Size = 100:

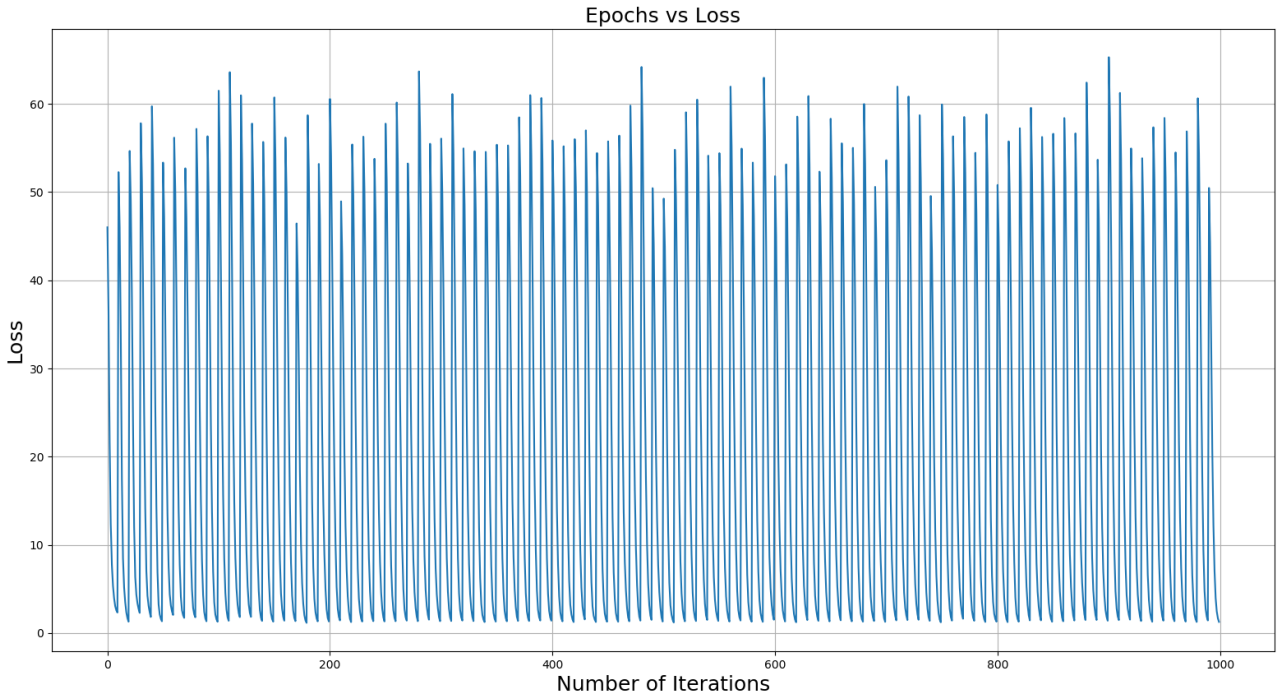

Figure 67: Loss

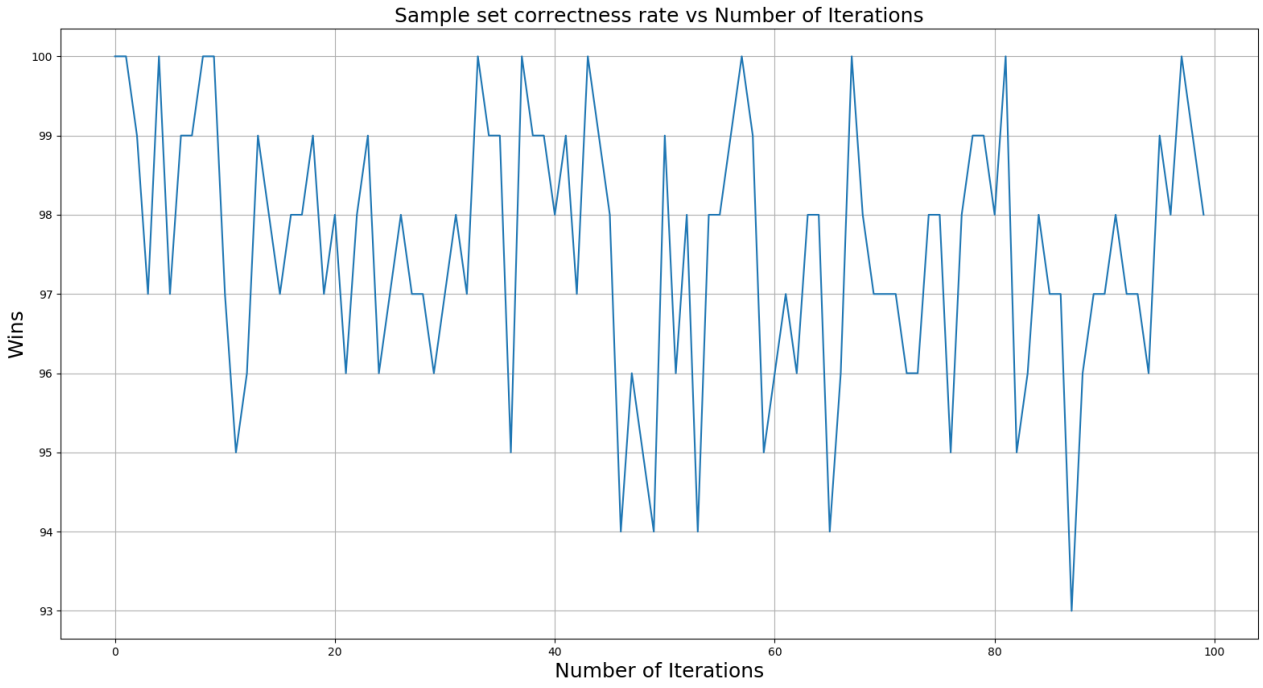

Figure 68: Moves correctly predicted

Size = 150:

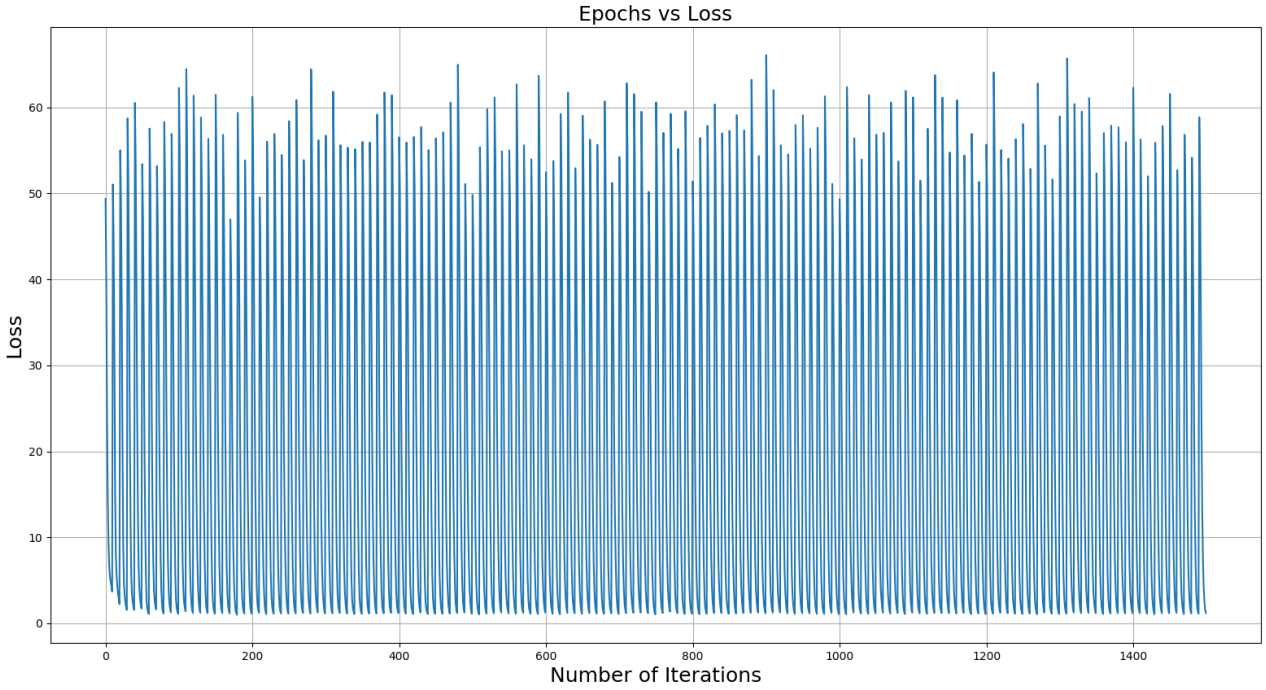

Figure 69: Loss

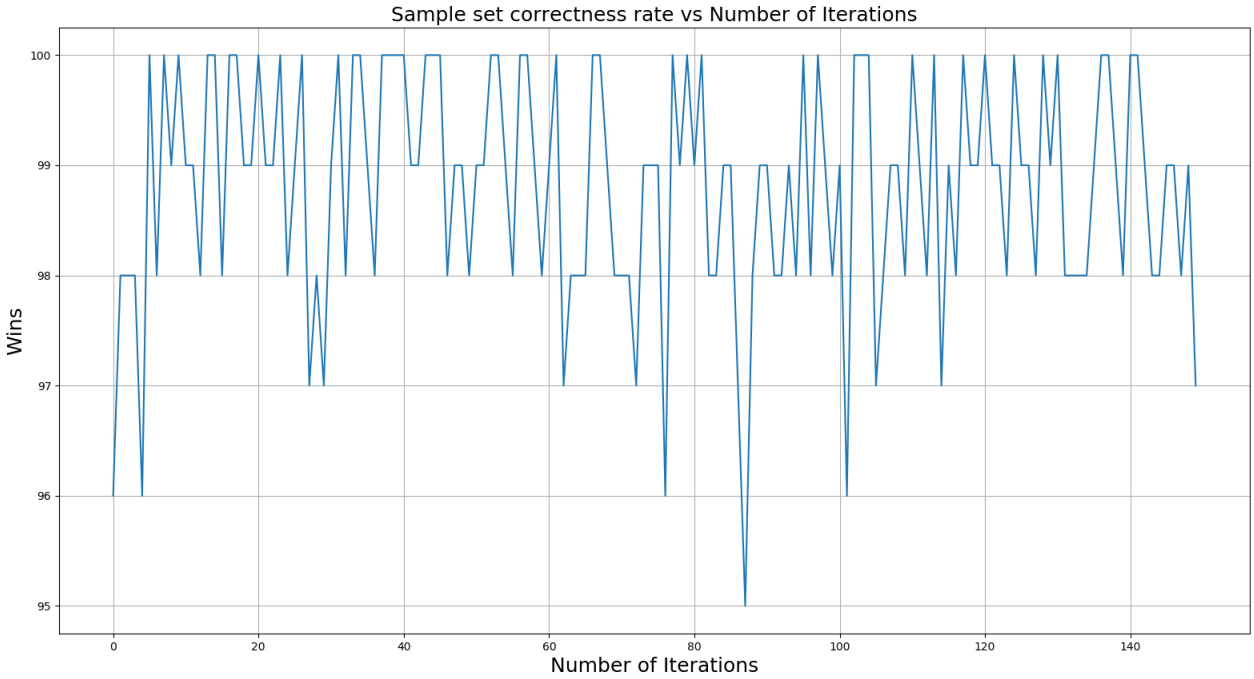

Figure 70: Moves correctly predicted

Size = 500:

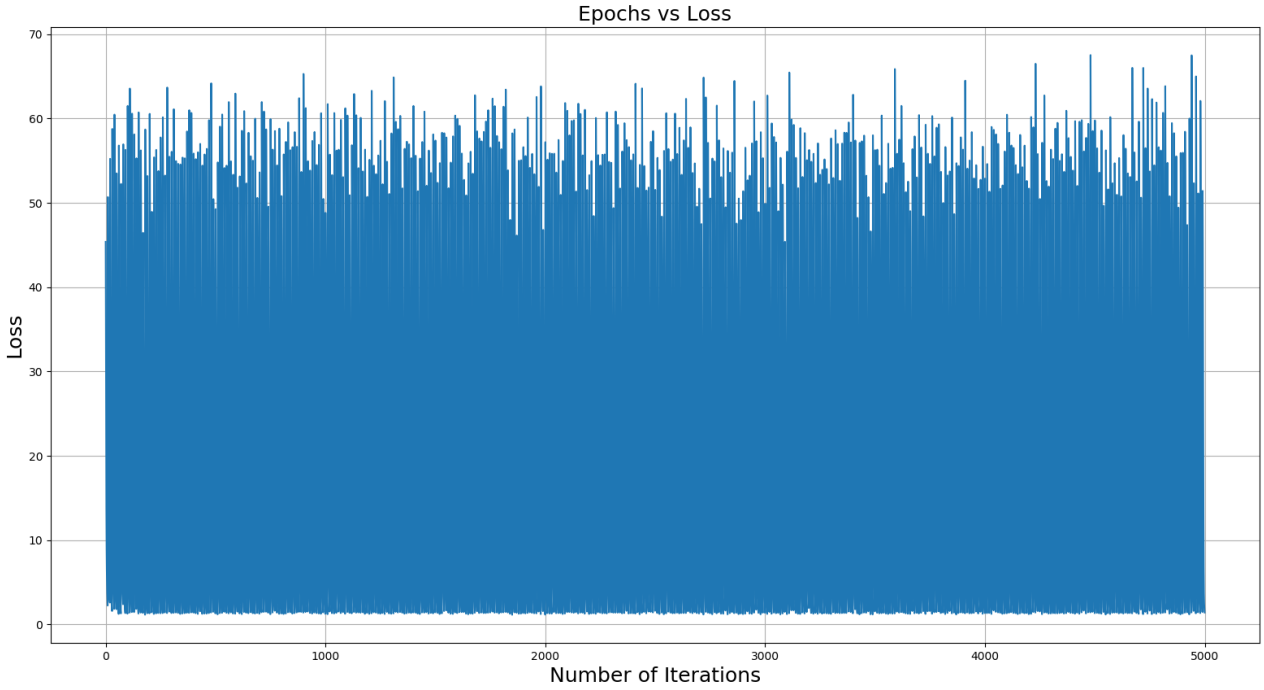

Figure 71: Loss

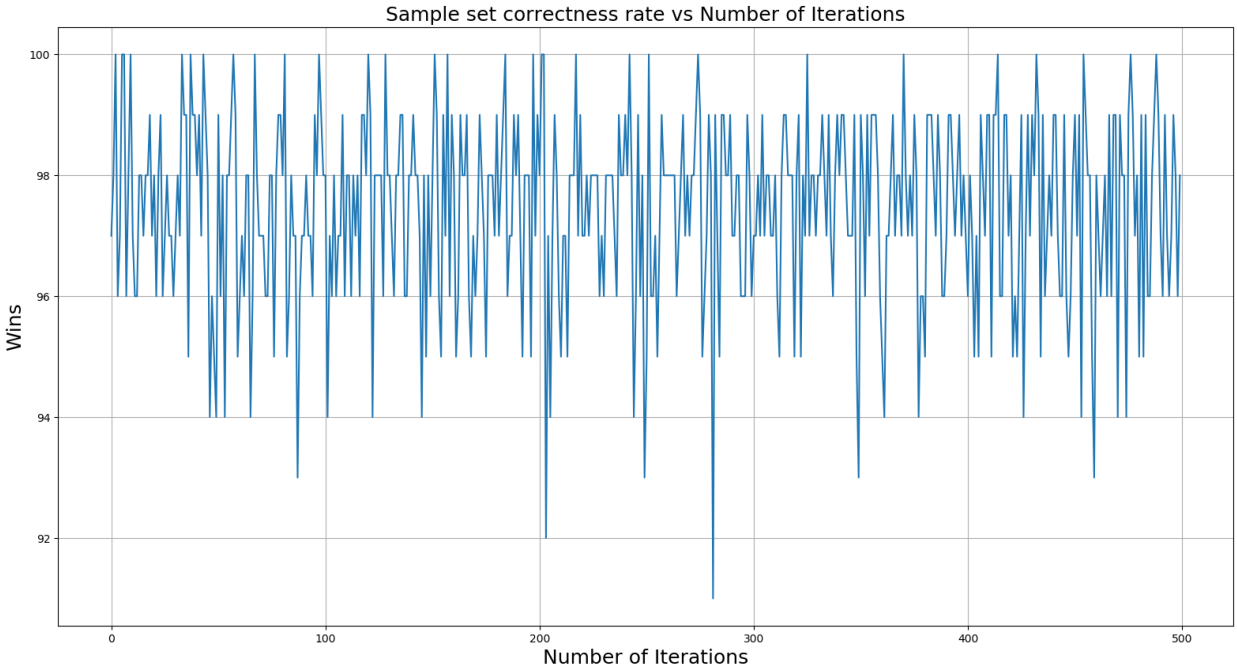

Figure 72: Moves correctly predicted

6.2 Convergence criterion

- The number of hidden layers is two
- Training sample size is 100
- Learning rate is set to ”0.83”
- Number of hidden layer nodes is Sample sets / 8
- Weights are randomly generated
- Ten epochs per a session of back propagation
- Sample set size is ”150”

Error = 0.00001:

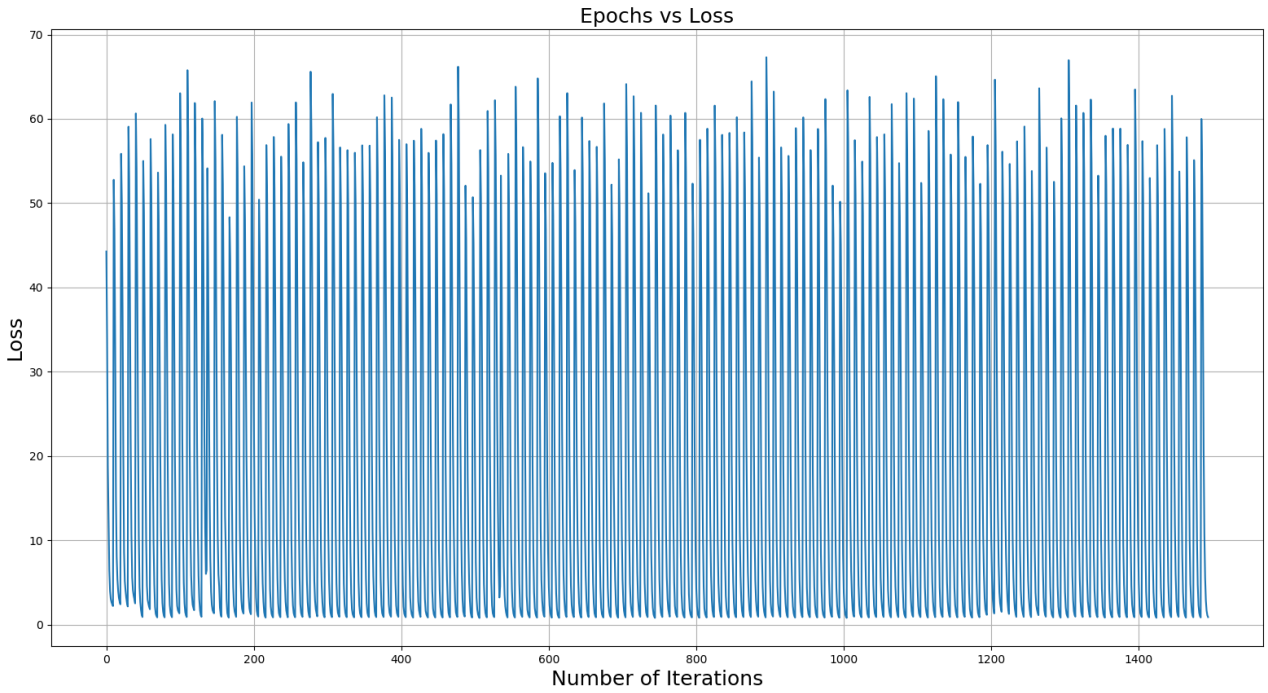

Figure 73: Loss

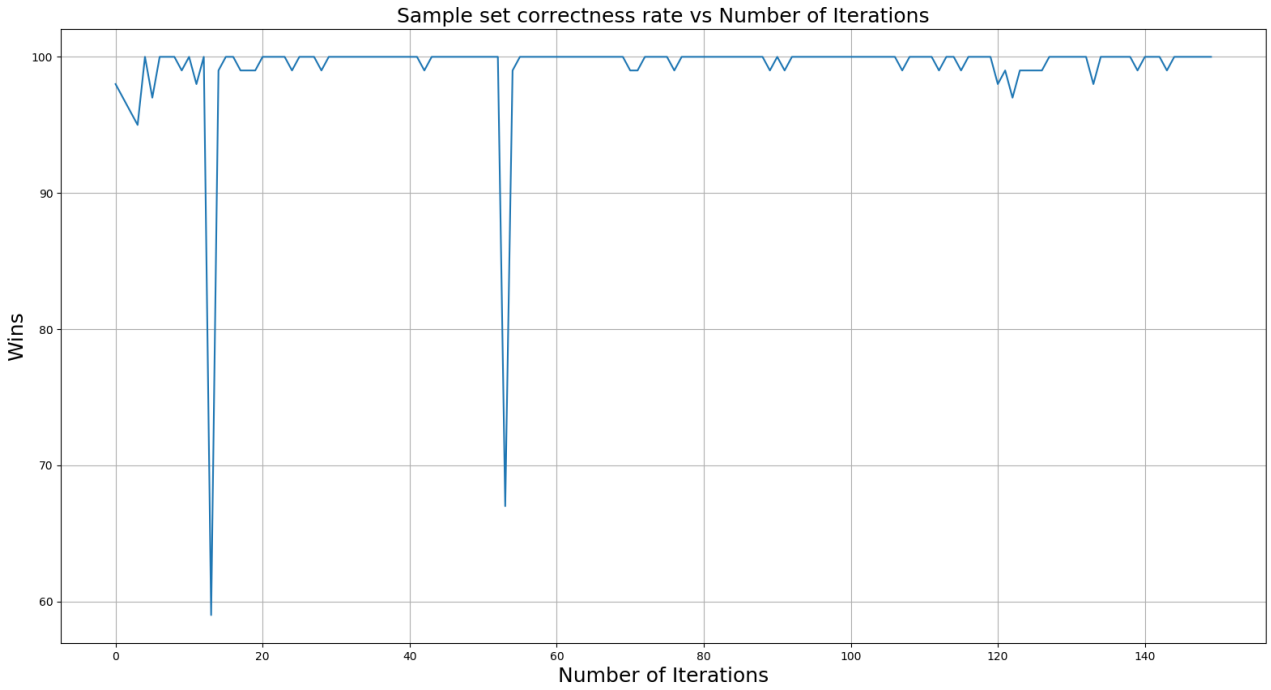

Figure 74: Moves correctly predicted

Error = 0.0001:

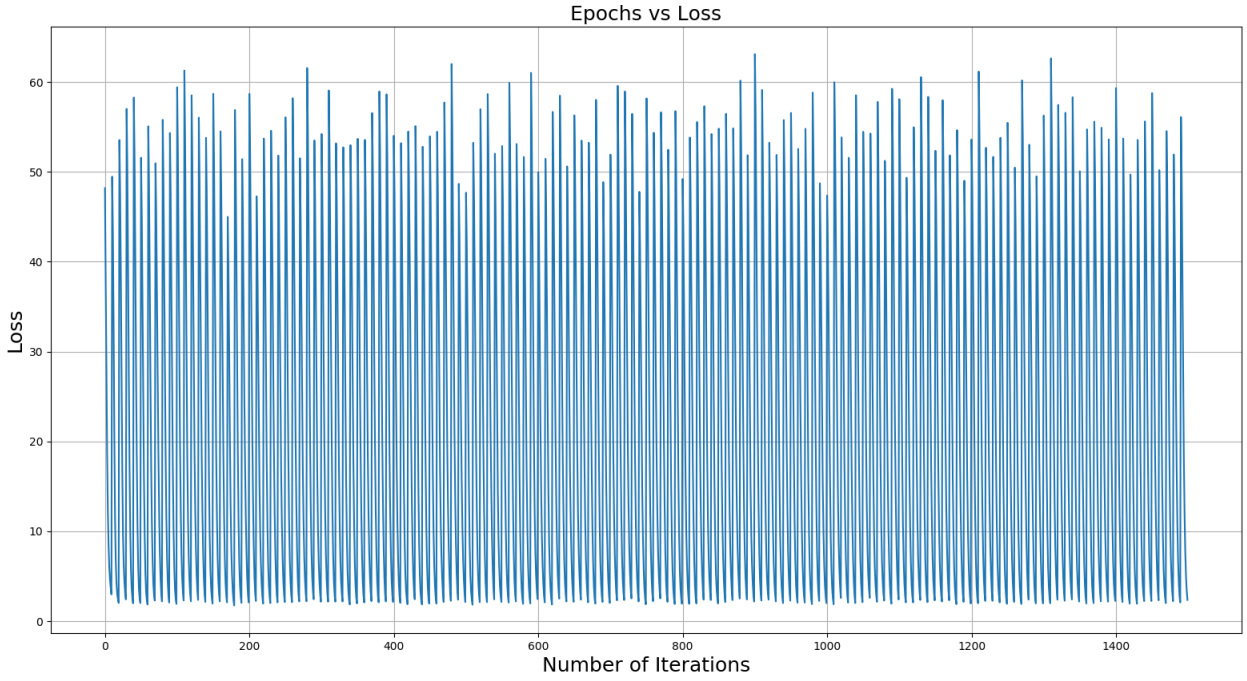

Figure 75: Loss

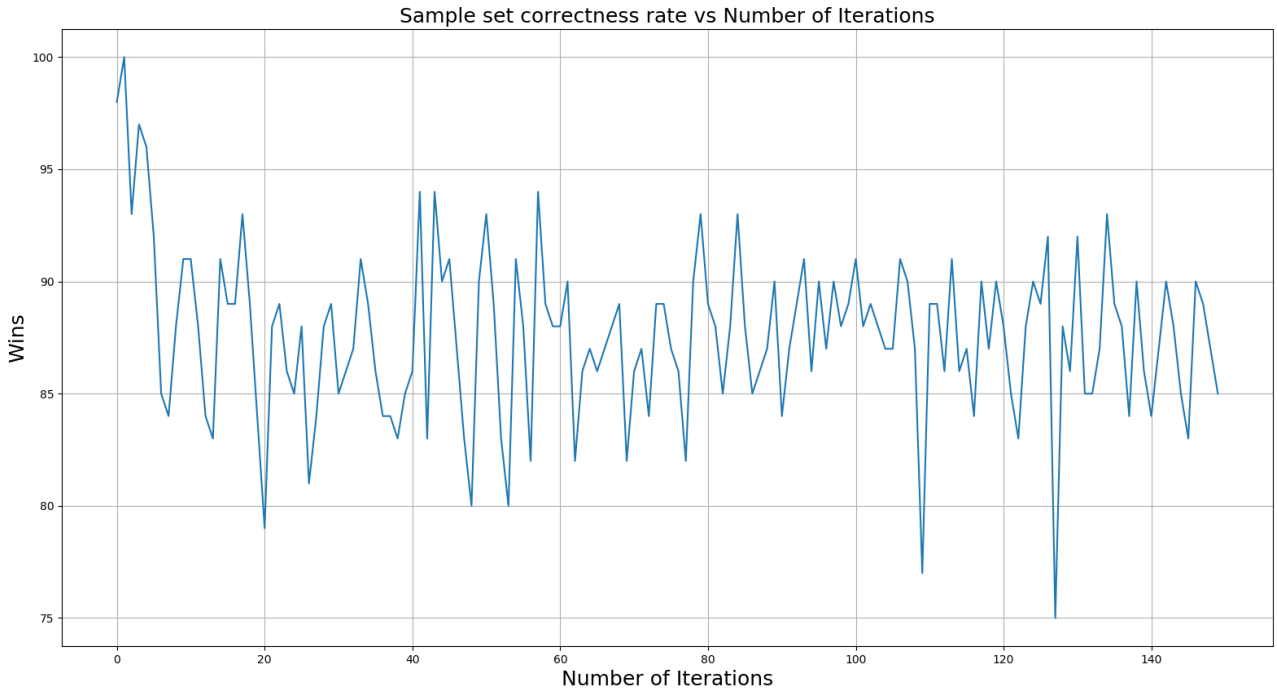

Figure 76: Moves correctly predicted

Error = 0.001:

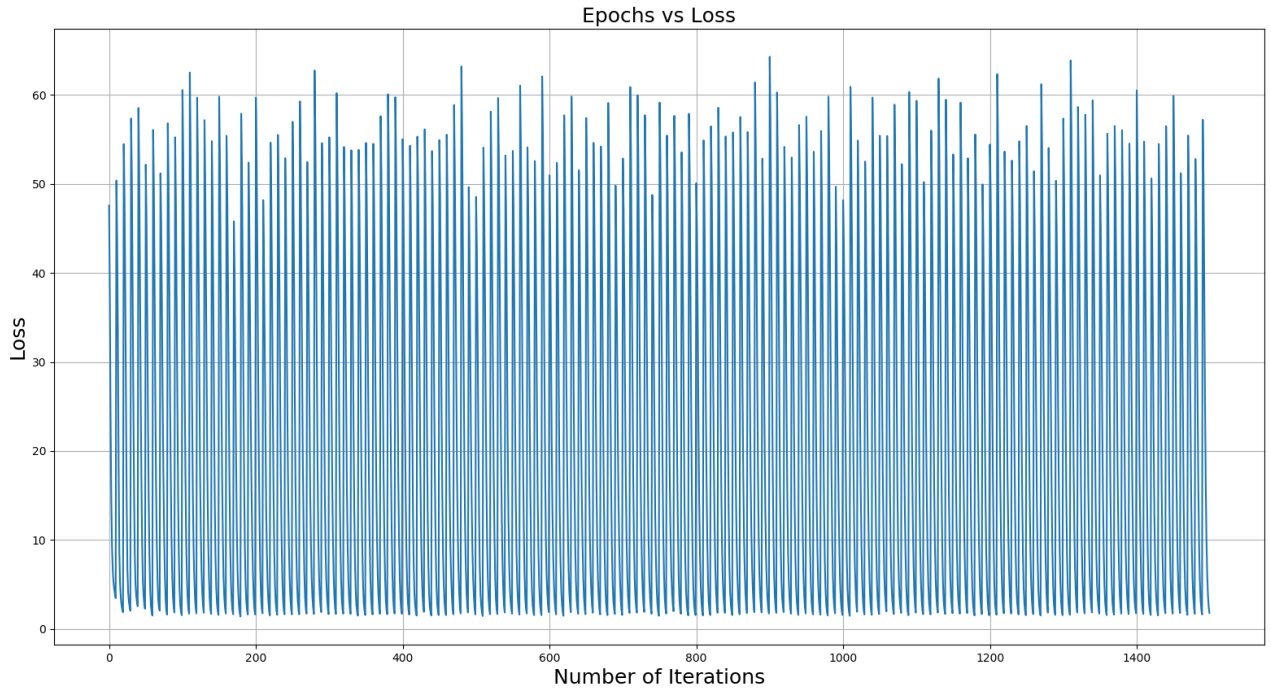

Figure 77: Loss

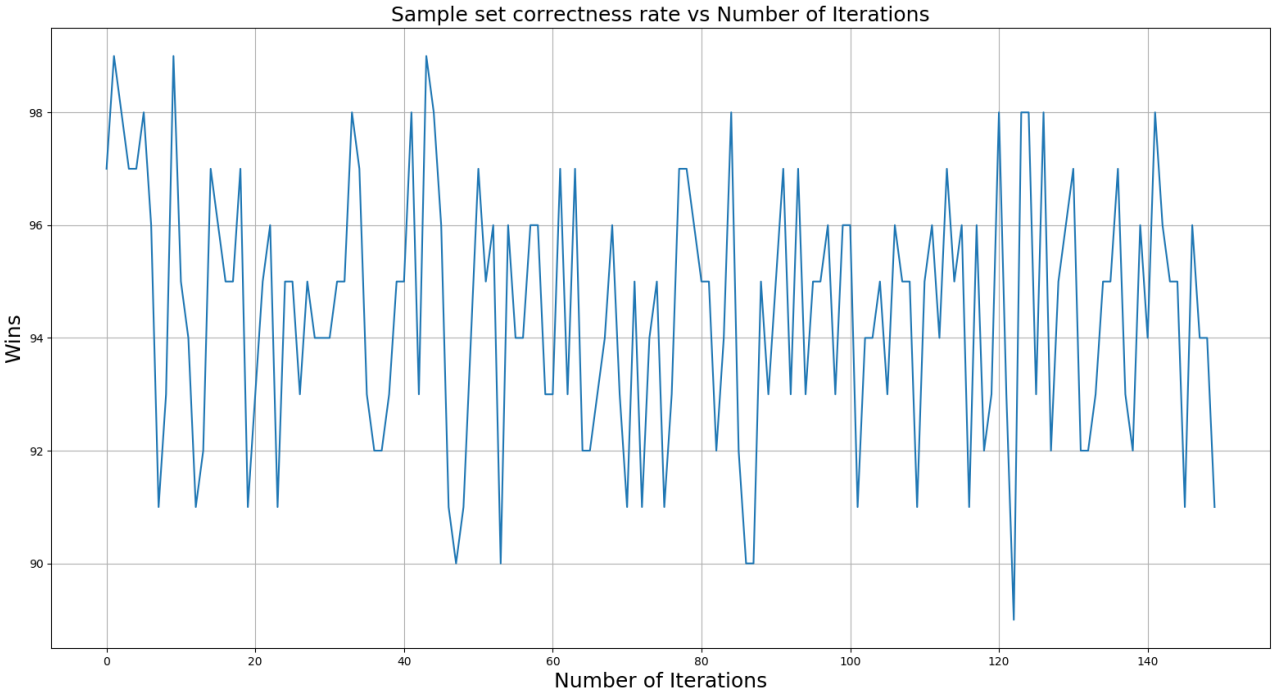

Figure 78: Moves correctly predicted

Error = 0.01:

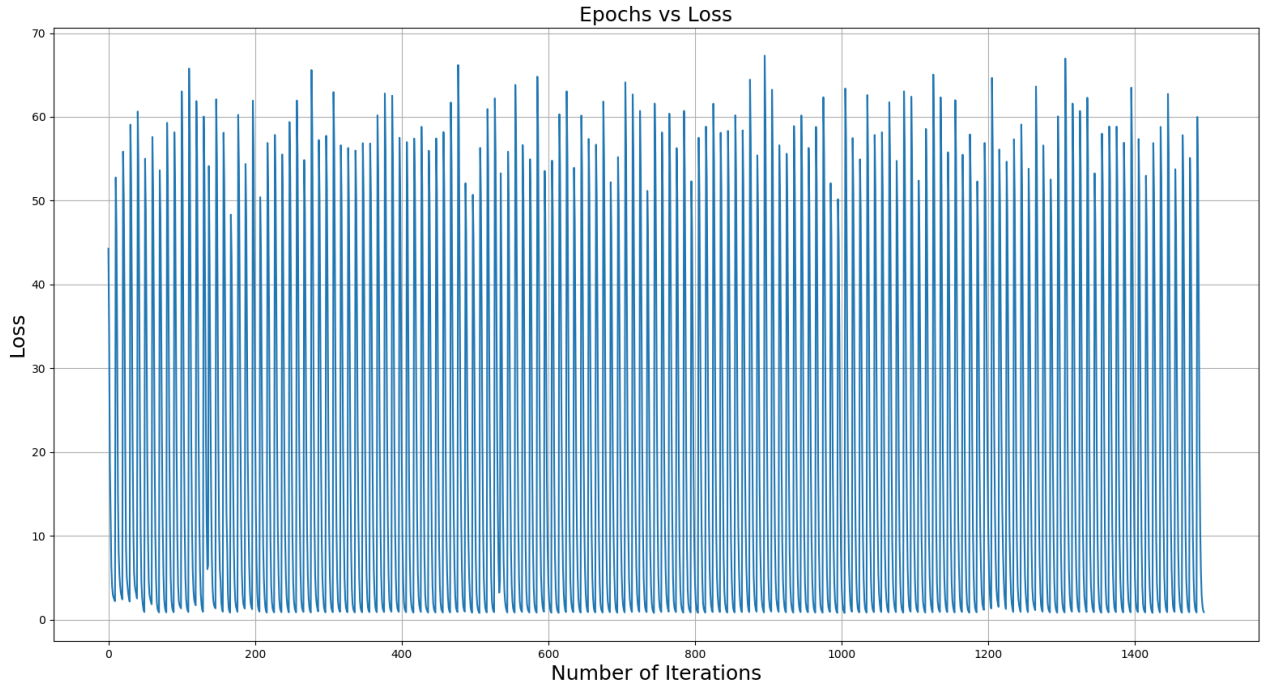

Figure 79: Loss

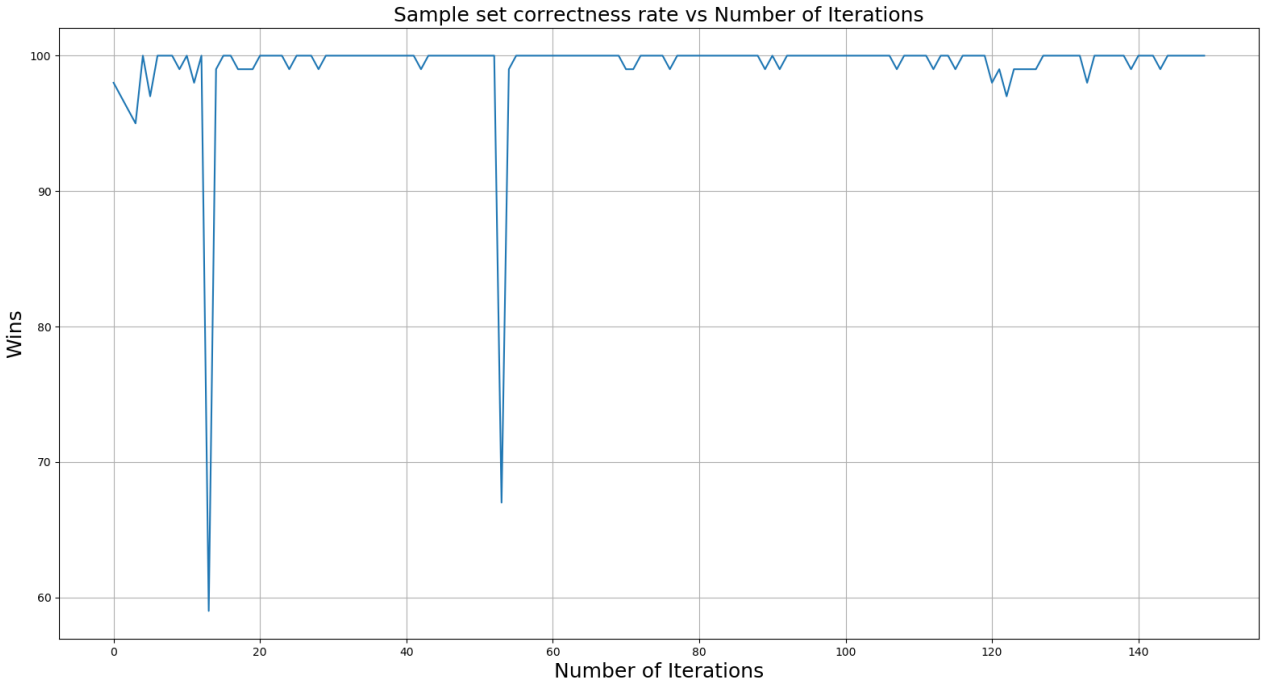

Figure 80: Moves correctly predicted

Error = 0.1:

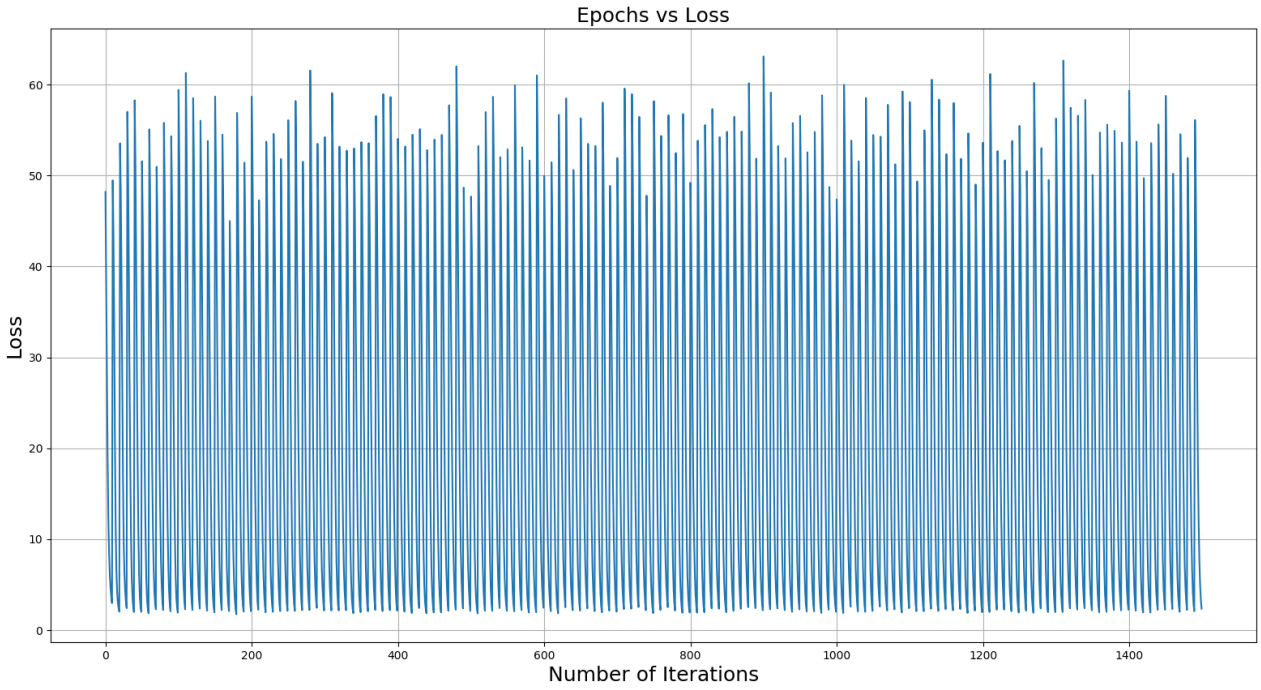

Figure 81: Loss

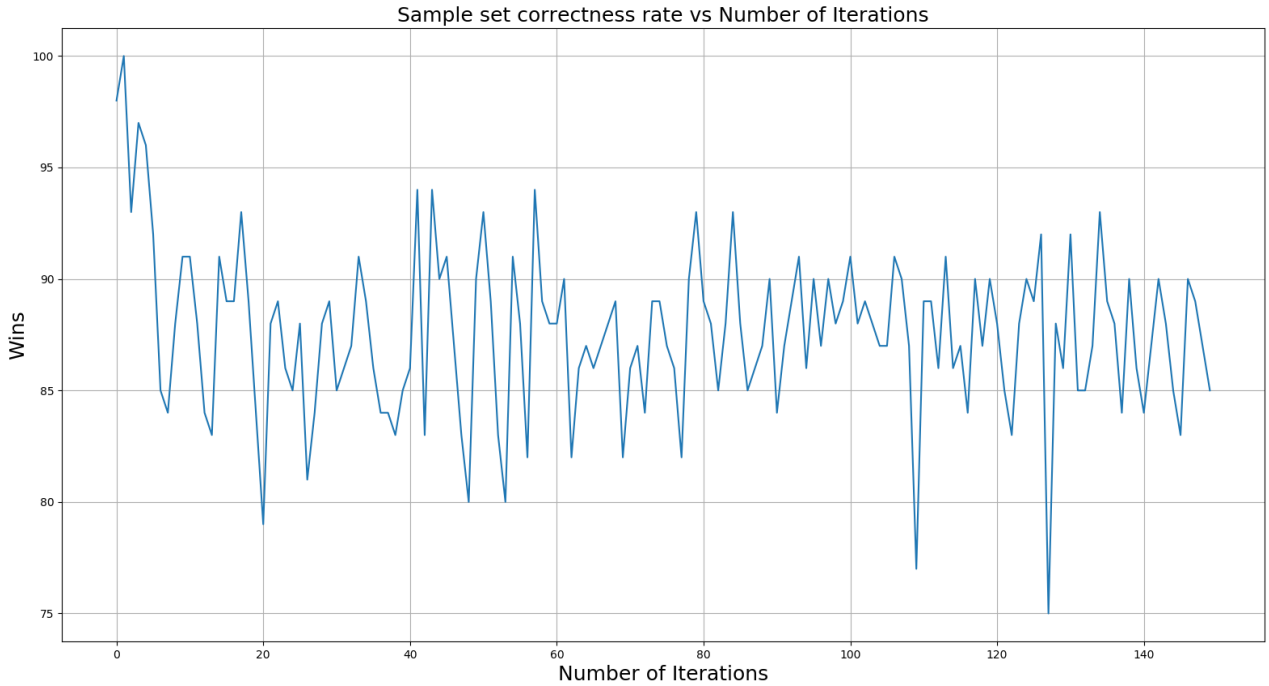

Figure 82: Moves correctly predicted

6.2.1 Final implementation performance

- The number of hidden layers is two
- Training sample size is 100, reduced to 40 for Agent
- Learning rate is set to ”0.83”
- Number of hidden layer nodes is Sample sets / 8
- Weights are randomly generated
- Ten epochs per a session of back propagation
- Error is 0.00001
- Sample set size is ”150”

Final:

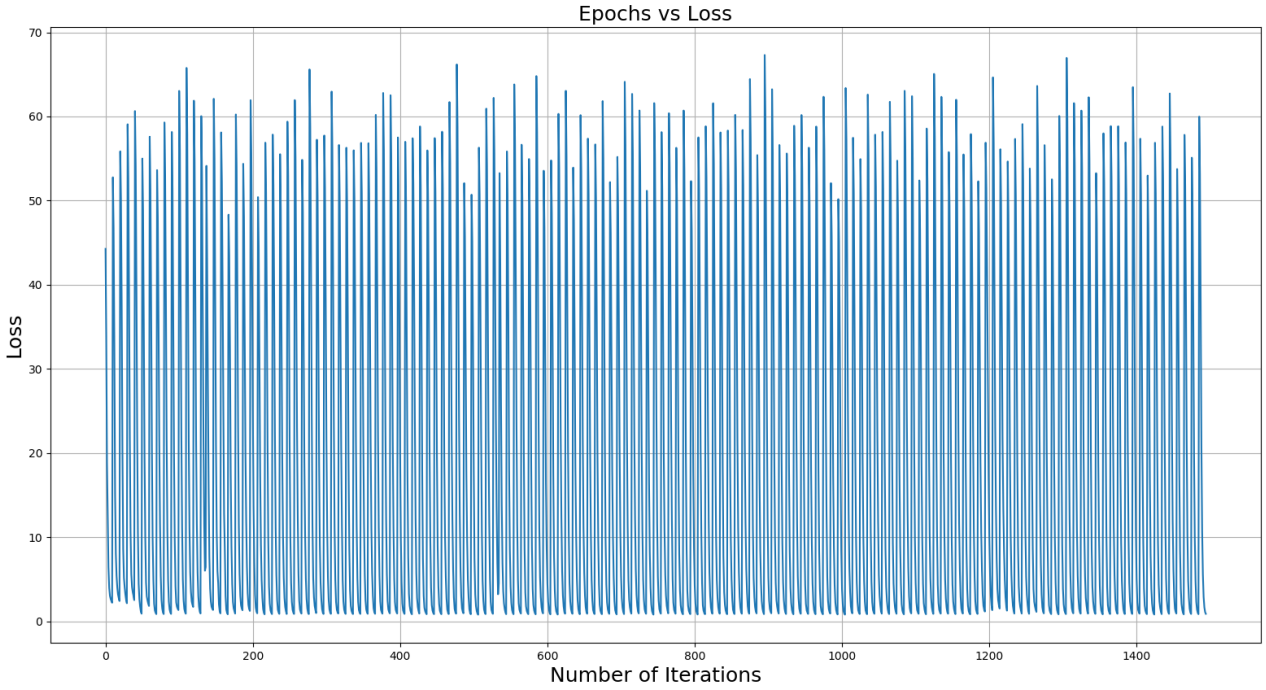

Figure 83: Loss

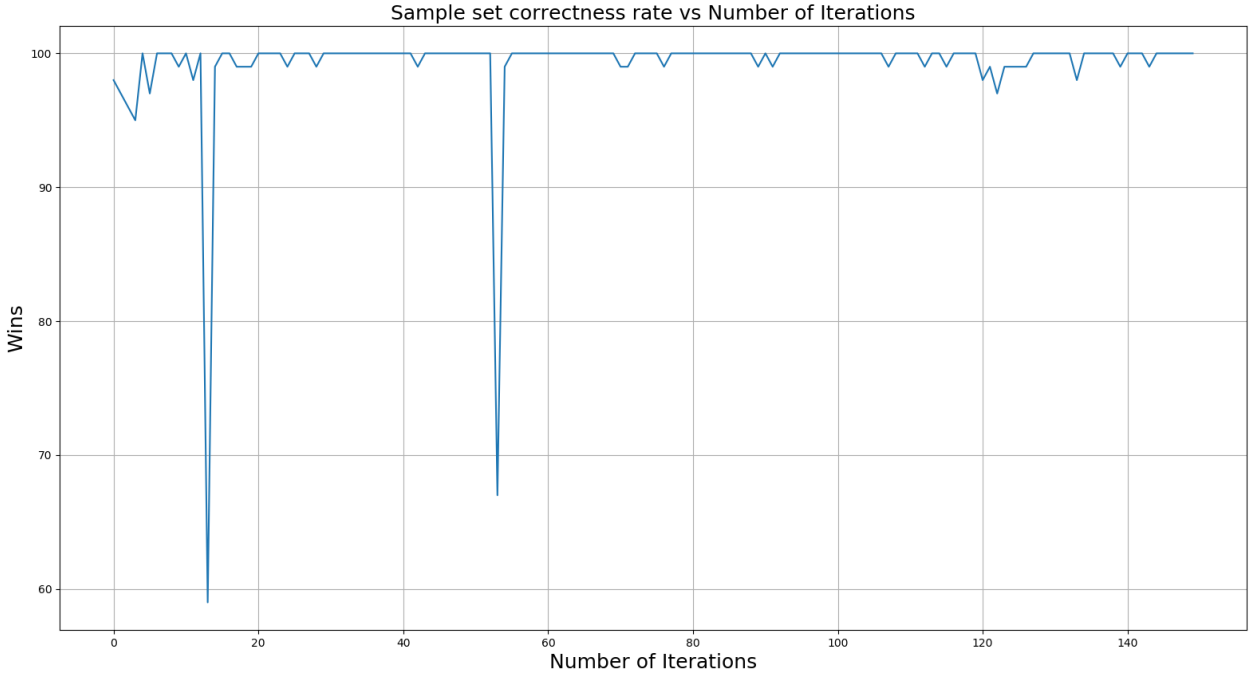

Figure 84: Moves correctly predicted

6.3 Agent and Gameplay

- The number of hidden layers is two
- Training sample size ”N” is examined, default is N = 40
- Learning rate is set to ”0.83”
- Number of hidden layer nodes is Sample sets / 8
- Weights are hard coded
- Ten epochs per a session of back propagation
- Error is 0.00001, unless specified

6.3.1 General performance

Performance is observed playing against only-paper with one hundred matches of ten thousand rounds each:

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 4 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 61.11 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (2400 of 1000000)
    avg score -9933.0, net score -993300.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (995700 of 1000000)
    avg score 9933.0, net score 993300.0
```

Figure 85: Agent vs Only-paper with back propagation after N random rounds, multi-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 143.21 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (2400 of 1000000)
    avg score -9933.0, net score -993300.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (995700 of 1000000)
    avg score 9933.0, net score 993300.0
```

Figure 86: Agent vs Only-paper with back propagation after N random rounds, single-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
100 matches run
total run time: 158.17 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.1% of rounds (1100 of 1000000)
    avg score -9973.0, net score -997300.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.8% of rounds (998400 of 1000000)
    avg score 9973.0, net score 997300.0
```

Figure 87: Agent vs Only-paper with back propagation after N = 10 rounds, single-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
100 matches run
total run time: 133.45 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.1% of rounds (1500 of 1000000)
    avg score -9961.0, net score -996100.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.8% of rounds (997600 of 1000000)
    avg score 9961.0, net score 996100.0
```

Figure 88: Agent vs Only-paper with back propagation after N = 20 rounds, single-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 134.13 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (2400 of 1000000)
    avg score -9932.0, net score -993200.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (995600 of 1000000)
    avg score 9932.0, net score 993200.0
```

Figure 89: Agent vs Only-paper with back propagation after N = 40 rounds, single-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 200.78 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.5% of rounds (5000 of 1000000)
    avg score -9869.0, net score -986900.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.2% of rounds (991900 of 1000000)
    avg score 9869.0, net score 986900.0
```

Figure 90: Agent vs Only-paper with back propagation after N = 65 rounds, single-threaded

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 201.98 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.6% of rounds (5700 of 1000000)
    avg score -9843.0, net score -984300.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.0% of rounds (990000 of 1000000)
    avg score 9843.0, net score 984300.0
```

Figure 91: Agent vs Only-paper with back propagation after N = 90 rounds, single-threaded

6.3.2 Only-object performance

Only object performance with the same amount of rounds and matches on a single thread with back propagation after ten, forty-five, and ninety rounds.

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_rock.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
100 matches run
total run time: 132.38 seconds

only_rock.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (1600 of 1000000)
    avg score -9944.0, net score -994400.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (996000 of 1000000)
    avg score 9944.0, net score 994400.0
```

Figure 92: Agent vs Only-rock with back propagation after N = 40 rounds

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_paper.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 133.78 seconds

only_paper.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (2400 of 1000000)
    avg score -9932.0, net score -993200.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (995600 of 1000000)
    avg score 9932.0, net score 993200.0
```

Figure 93: Agent vs Only-paper with back propagation after N = 40 rounds

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py only_scissors.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow enc
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 134.73 seconds

only_scissors.py: won 0.0% of matches (0 of 100)
    won 0.2% of rounds (2000 of 1000000)
    avg score -9944.0, net score -994400.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 99.6% of rounds (996400 of 1000000)
    avg score 9944.0, net score 994400.0
```

Figure 94: Agent vs Only-scissors with back propagation after N = 40 rounds

6.3.3 beat-common performance

Beat common performance on a single thread

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py beat_common.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow e
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 352.58 seconds

beat_common.py: won 0.0% of matches (0 of 100)
    won 32.6% of rounds (326300 of 1000000)
    avg score -2933.0, net score -293300.0

Agent_NN.py: won 100.0% of matches (100 of 100)
    won 62.0% of rounds (619600 of 1000000)
    avg score 2933.0, net score 293300.0
```

Figure 95: Beat common with back propagation after N = 40 rounds

6.3.4 Really smart bot performance

Really smart bot performance on a single thread

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py really_smart_bot.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow encoun
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 619.32 seconds

Agent_NN.py: won 23.0% of matches (23 of 100)
    won 33.0% of rounds (330138 of 1000000)
    avg score -79.0, net score -7904.0

really_smart_bot.py: won 77.0% of matches (77 of 100)
    won 33.8% of rounds (338042 of 1000000)
    avg score 79.0, net score 7904.0
```

Figure 96: really smart bot with back propagation after N = 40 rounds

6.3.5 Random play bot performance

Random play bot performance on a single thread

```
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents>rpsrunner.py -m 100 -r 10000 Agent_NN.py random_play.py
Pool 1: 1 bots loaded
Pool 2: 1 bots loaded
Playing 100 matches per pairing.
Running matches in 1 threads
C:\Users\liamb\Documents\University\2020\EAI\Practicals\Practical 3\sample_agents\Neural_Network_task_2.py:199: RuntimeWarning: overflow e
  sum = 1 / (1 + np.e**-z)
100 matches run
total run time: 373.80 seconds

Agent_NN.py: won 42.0% of matches (42 of 100)
    won 33.3% of rounds (332849 of 1000000)
    avg score -0.3, net score -32.0

random_play.py: won 55.0% of matches (55 of 100)
    won 33.3% of rounds (332881 of 1000000)
    avg score 0.3, net score 32.0
```

Figure 97: random play bot with back propagation after N = 40 rounds

7 Discussion of results

The Neural Network's performance was found to be dependent on a balance of it's hyper parameters. The parameters are required to be balanced in order for beneficial learning and predictions to take place.

The number of epochs chosen affects the amount of learning the neural network is able to perform. When too few epochs the neural network is unable to perform enough iterations to fine tune the neural network for better predictions. The sample is in no particular order to improve generalization. If the number of iterations is too low the neural network may even fail to generalize well. The weights may not have enough time to even be optimized from their random initialized value as observed in figure 2 and figure 4. However if the size becomes too large the data may begin over learning as in figure 11. This is problematic if specialization is not desired but may be avoided by the use of the convergence criterion.

The number of hidden of layers is based around the number of features or characteristics existing in the sample data and what is of importance to the neural network's learning. The number of layers enables the neural network to learn with discontinuities and linearity changes in the data. An increase in the amount of layers also may exhibit and enhance characteristics found in the data which are undesired. Noise or outliers in the data may be observed as important by the network. This skews the learning and optimizing process to the absolute correct data. Two layers may be observed as a good choice due to discontinuities in the sample data being mostly catered for then. The number of predicted moves correct improved and stayed constant when moving one layer to two. Thus two layers seems the best choice for this implementation.

The number of hidden nodes in one layer (size) is of great importance to the execution time of the neural network. The network linearly increases with the number of hidden nodes in run time. The program was unable to learn with a single node in each layer, but found successful learning with sample set size divided by eight nodes a hidden layer. The learning continued to improve as the size approached half the sample set size at the expense of a longer run time. Past half the sample set size the learning quality begins to degrade again. The loss for a layer size twice that of the sample size (figure 38) indicated an increasing loss. The agent attempts to optimize for the noise in the data in larger layer sizes. The learning thus reduces in quality with too large a layer size with the run time too long.

The learning rate is the rate chosen at which the neural network will attempt to optimize to a minimum. The learning rate is a constant which is set to a relatively small number. This is to avoid the issue whereby if too large the learning rate may push the network to move away from the minimum (correct approximation) when optimizing (learning). This may be observed from LR = ”10” (figure 53). As the learning rate increases the observed loss tends to begin increasing as well. The learning rate may however not be too small either. The learning rate is multiplied with the cost update added to the weights. If the learning rate is too small no beneficial learning will take place as the update will become too small to be useful (figure 42). No update to the weights improving the network may occur.

The number of sample sets accessed enables the network to learn and fine tune it's data to become more accurate to the real scenario the data is based on. When balanced with the number of epochs, the network is able to be generalize well if the sample set is given in no particular order. The more sets the neural network has to learn from the better as long as

the data is not biased in any way, it is always fine tuning to be more accurate and give good approximations to the data. A smaller learning rate may be more beneficial in larger data sets to enable more generalization, as the small network optimization from the learning rate is compensated by the number of data sets. The results show a tendency to learning better as the sets increase. Other factors linked to the learning rate and number of epochs affect the ability of the network to learn from many sets effectively.

The convergence criterion acts to prevent the network from over training on a single data set. If the loss begins becoming quite small or starts increasing the program may terminate the training batch (sample set) in order for the network to remain more generalized in use and approximations. The criterion thus also may decrease run time when a relatively large number is chosen for the error criterion as termination will occur after less epochs. If the criterion is too large termination will occur too early and not much learning will take place.

Observing the agent, changing the number of rounds (N) before back propagation affected the run time. The larger N, the shorter the run time. This is due to the number of back propagation operations reducing with a larger N. The number of epochs from each propagation required to run in one match is thus reduced. The run time decreases beneficially. The larger the N the more nodes which have back propagation performed on them as well. This allows more learning to take place and a larger more correct move-set to be played. A random number of rounds increases the run time as the inconsistent size may lead to more back propagations for each match. A balance with the number of epochs and hidden layer size will allow the run time to be optimal.

When the parameters are balanced well the agent's performance dominates most of the opposing agents. The Neural Network agent acquired a ninety-nine percent success rate against the only objects due to a good balance in the hyper parameters existing and the relative ease in learning that only one move will be played. This is however more of a specialization than a generalization case. The agent however preformed evenly with the really smart bot and beats- beat common. The run time of each game session is dependent on the back propagation process of the network agent and the algorithms implemented by each of the agents. The win rate is dependent of the process of the back propagation. Beyond the initial rounds which are played randomly while data is being collected, the hyper parameters and data collected determine the win rate against the other agent.

The artificial neural network is thus optimized by a variety of factors including its hyper parameters and input data format. The form of artificial intelligence is effective for and identifying and learning patterns for solving for solutions given input data. The deep learning of the neural network still requires much debate as the number of hidden layers had no broad effect on the data for this application. The neural network is however undoubtedly an effective tool for approximations and predicting data.

8 Conclusion

The neural network was successfully implemented. Back propagation performed to expectation with the network able to train and predict move sets for a game of RPS to varying degrees of accuracy based on the hyper parameters. The hyper parameters were able to be manipulated to optimize the efficiency of the network and observe different problems which may occur within the network if the parameters are not balanced. The input and output format with the transfer function were successfully implemented to be able to best realize the data of the neural network and learning process. The agent implementation with the network is also successful with the agent winning and drawing matches to varying degrees again based on the aspects of hyper parameters as well as the generalization and specialization of the agent.

9 References

[1] R. Khan, *Neural networks, nothing but numpy*, Available at https://medium.com/towards-artificial-intelligence/nothing-but-numpy-understanding-creating-neural-networks-with-computational-graphs-from-scratch-6299901091b0 (2019/06/22).

[2] W. D. Plessis, "Eai-320 practical 3 specification," Specification, 2020.

[3] P. N. Stuart Russel, *Artificial intelligence, a modern approach*, ser. Always learning. Pearson, 2010, ISBN: 1292153962.

10 Appendices

A Textbook algorithm

function BACK-PROP-LEARNING($examples, network$) **returns** a neural network
inputs: $examples$, a set of examples, each with input vector **x** and output vector **y**
$network$, a multilayer network with L layers, weights $w_{i,j}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

repeat
for each weight $w_{i,j}$ in $network$ **do**
$w_{i,j} \leftarrow$ a small random number
for each example $(\mathbf{x}, \mathbf{y})$ **in** $examples$ **do**
/* *Propagate the inputs forward to compute the outputs* */
for each node i in the input layer **do**
$a_i \leftarrow x_i$
for $\ell = 2$ **to** L **do**
for each node j in layer ℓ **do**
$in_j \leftarrow \sum_i w_{i,j} a_i$
$a_j \leftarrow g(in_j)$
/* *Propagate deltas backward from output layer to input layer* */
for each node j in the output layer **do**
$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
for $\ell = L - 1$ **to** 1 **do**
for each node i in layer ℓ **do**
$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j]$
/* *Update every weight in network using deltas* */
for each weight $w_{i,j}$ in $network$ **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
until some stopping criterion is satisfied
return $network$

Figure 98: Textbook algorithm for a Neural network with Backpropagation

B Task 1: Neural Network code

```
#!/usr/bin/env python3
import csv
import numpy as np
import matplotlib.pyplot as plt
#====================

# Refactor dates:
# 2/04/2020
# 8/04/2020
# 25/04/2020
# 28/04/2020
# 01/05/2020
# 03/05/2020
# and more dates that I am not proud to put here

#N = 1 #np.random.randint(5, 100)
weight_random_range = 1000000
test = []
wins = []
losses = []
np.random.seed(1)

#====================
# Generate data sets
# Return the 1 hot training data of a move as input feature
def last_move_1hot(move):
    if move == 'R':
        r = 1
        p = 0
        s = 0
        return r, p, s
    elif move == 'P':
        r = 0
        p = 1
        s = 0
        return r, p, s
    else:
        r = 0
        p = 0
        s = 1
        return r, p, s

# set target as winning move
def win_move(y_move):
    r, p, s = last_move_1hot(y_move)
    if r == 1:
        r = 0
        p = 1
```

```
        s = 0
        return r, p, s
    elif p == 1:
        r = 0
        p = 0
        s = 1
        return r, p, s
    else:
        r = 1
        p = 0
        s = 0
    return r, p, s


# Read CSV to generate input and testing data
def CSV_read(N, start = 0):
    d = []
    c = []
    filename = "data1.csv"
    with open(filename) as CSV:
        reader = csv.reader(CSV, delimiter=',')
        how_many = skip = 0
        for row in reader:
            if skip < start:
                skip += 1
                continue
            for i in range(4):
                r, p, s = last_move_1hot(row[0][i])
                d.append(r)
                d.append(p)
                d.append(s)

            # generate move as target which may be used to train towards
            r, p, s = win_move(row[1])
            c.append(r)
            c.append(p)
            c.append(s)
            how_many += 1
            if how_many == N:
                break

        # bias
        d.append(1)

        #print("Data read complete")

    return d, c

# ==========================================
# export data to CSV file
```

```
def export(DHW, HHW, HCW):
    #np.savetxt("Input_Hidden.csv", DHW, delimiter=",")
    #np.savetxt("Hidden_Hidden.csv", HHW, delimiter=",")
    #np.savetxt("Hidden_Output.csv", HCW, delimiter=",")
    return print("Files saved")

# generate files with the format of an array for easy copying
def gen_hard_code(DHW, HHW, HCW):
    np.set_printoptions(threshold=np.nan)
    filename = "DUMP1.txt"
    file = open(filename, "w")
    file.write(repr(DHW))
    filename = "DUMP2.txt"
    file = open(filename, "w")
    file.write(repr(HHW))
    filename = "DUMP3.txt"
    file = open(filename, "w")
    file.write(repr(HCW))
    return print("data hardcoded")

# load pre-generated weights
def Load_from_file():
    W = [line.split(',') for line in open("Input_Hidden.csv")]
    DHW = np.array(W)
    DHW.transpose()
    W = [line.split(',') for line in open("Hidden_Hidden.csv")]
    HHW = np.array(W)
    HHW.transpose()
    W = [line.split(',') for line in open("Hidden_Output.csv")]
    HCW = np.array(W)
    HCW.transpose()
    return DHW.astype('float64'), HHW.astype('float64'), HCW.astype('float6


# ========================================
# Graphing
# pyplot graphing function
def graph(x, y, xlab, ylab, title, file):
    x = np.arange(0, x, 1)
    plt.title(title, size=18)
    plt.xlabel(xlab, size=18)
    plt.ylabel(ylab, size=18)
    plt.grid(1)
    plt.plot(x, y)
    plt.show()
    if file == 1:
        plt.savefig('Loss.png', bbox_inches='tight')
    else:
        None
    return
```

```
#=========================================
# back propagation functions
# Generate weights
def weights(depth, width):
    W = np.empty([depth, width])
    W.astype(float)

    for i in range(depth):
        for j in range(width):
            rand = np.random.randint(-1*weight_random_range, weight_random_
            W[i][j] = rand
    return W


# Generate the arrays as numpy arrays for the BackProp function
def gen_arrays(D, C, Win, Wout):
    retD = np.array(D)
    retC = np.array(C)
    retWin = np.array(Win)
    retWout = np.array(Wout)
    return retD, retC, retWin, retWout


# array of errors for back propagation
def array_errors(comp, target):
    copy = target - comp
    return copy


# compute the error of the data
def compute_error(computed, target):
    sum = 0
    for i in range(np.size(computed)):
        sum += np.square(target[i] - computed[i])
    sum /= 2
    return sum


# computer convergence criterion
def convergence_criterion(Ei, Ef, error):
    if Ef - Ei > 0 or error > np.abs(Ef - Ei):
        return True
    else:
        return False


# sigmoid activation function
def activation_function(z):
```

```
    sum = 1 / (1 + np.e**-z)
    return sum


# derivative of activation function
def derivative_activation_function(z):
    sum = activation_function(z)
    sum = sum * (1 - sum)
    return sum


# using numpy matrix multiplication
def forward_layer_propagation(A, W):
    inj = np.matmul(W, A)
    Ao = activation_function(inj)
    return Ao, inj


# compute upstream gradient at output
def output_back_gradient(inj, Err):
    der = derivative_activation_function(inj)
    del_j = der * Err
    return del_j


# compute upstream gradient for nodes not the output
def back_gradient(inj, W, j_del):
    hold = []
    for i in range(W.shape[1]):
        sum = 0
        for j in range(W.shape[0]):
            sum += W[j][i]
        hold.append(sum)
    hold = np.array(hold)
    inj = derivative_activation_function(inj)
    sum = np.outer(inj, hold).transpose()
    i_del = np.matmul(sum, j_del)
    return i_del


# generate a bias array of the correct size for appending to np array
def bias_array(size):
    ret = []
    for _ in range(size):
        ret.append(1)
    return np.array(ret)


# update weights
def weight_update(W, delta, learning_rate, A):
```

```
    sum = np.outer(delta, A)
    sum *= learning_rate
    new_weights = W + sum
    return new_weights


# changes final computation of moves from decimal to moves to be played
def moves(number_of_moves, comp_data):
    tolerence = 0.6
    if number_of_moves*3 > np.size(comp_data):
        print("Not enough moves from data")
        return 1

    #if 0.4 < comp_data[0] < 0.6:
    #    return print("Data not fitted enough")

    moves = []
    i = 0
    while len(moves) != number_of_moves:
        for x in range(3):
            if comp_data[i+x] > tolerence:
                if x==0:
                    moves.append("R")
                elif x==1:
                    moves.append("P")
                elif x==2:
                    moves.append("S")
                i += 3
                break
            elif x == 2:
                moves.append("R")
                i += 3
                break

    return moves


# test against data not used to train with from CSV best score is 100/100
def real_test(target, comp):
    w = c = 0
    for i in range(np.size(target)):
        if comp[i] > 0.8:
            if target[i] == np.ceil(comp[i]):
                c += 1
            else:
                w += 1
    return w, c


# Back-propagation algorithm
```

```
def BackPropagation(dn, cn_target, dhw, hhw_1, hcw, learning_rate, epoch, e
    loss = []
    e = 0
    cn_comp = []
    for iteration in range(epoch):

        #print("Iteration: "+str(iteration+1))

        # forward propagation
        # input to hidden 1
        dn.transpose()
        hn1, in1 = forward_layer_propagation(dn, dhw)
        # add bias
        bias = bias_array(np.size(hn1[0]))
        hn1 = np.append(hn1, bias, axis=0)
        in1 = np.append(in1, [1], axis=0)

        # ================================================
        # hidden 1 to hidden layer 2
        hn2, in2 = forward_layer_propagation(hn1, hhw_1)

        # add bias
        bias = bias_array(np.size(hn2[0]))
        hn2 = np.append(hn2, bias, axis=0)
        in2 = np.append(in2, [1], axis=0)

        # ================================================

        # hidden to output
        cn_comp, in3 = forward_layer_propagation(hn2, hcw)

        # only prediction not backprop desired
        if cn_target == []:
            return cn_comp

        # compute errors
        e = compute_error(cn_comp, cn_target)
        loss.append(e)

        # check error convergence
        if len(loss) > 1:
            if convergence_criterion(loss[len(loss) - 2], loss[len(loss) -
                break

        # ================================================

        # back propagation
        # Cost calculation
        E = array_errors(cn_comp, cn_target)
```

```
        # output to hidden 2 gradient calculation
        j_del = output_back_gradient(in3, E)
        # ====================================

        # hidden 2 to hidden 1 gradient calculation
        i_del_1 = back_gradient(in3, hcw, j_del)

        # hidden 1 to input
        i_del_2 = back_gradient(in2, hhw_1, i_del_1)

        # ======================================

        # update output to hidden weights
        hcw = weight_update(hcw, j_del, learning_rate, hn2)

        # remove bias before adjusting weights of previous layer
        i_del_1 = np.delete(i_del_1, np.size(i_del_1) - 1, axis=0)

        # update hidden 2 to hidden 1 weights
        hhw_1 = weight_update(hhw_1, i_del_1, learning_rate, hn1)

        # remove bias before adjusting weights of previous layer
        i_del_2 = np.delete(i_del_2, np.size(i_del_2) - 1, axis=0)

        # update hidden in input layer weights
        dhw = weight_update(dhw, i_del_2, learning_rate, dn)

    return dhw, hhw_1, hcw, loss, cn_comp


# ===================================
# Batch training
def tests(N_samples, sets, iterate, error_stop, learn_rate, hidden_nodes):
    wins = []
    loss_total = []
    #===========================================
    D, C = CSV_read(N_samples, 0)
    Win = weights(hidden_nodes, len(D))
    Whidden_2 = weights(hidden_nodes, hidden_nodes + 1)
    Wout = weights(len(C), hidden_nodes+1)
    dn, cn, Win, Wout = gen_arrays(D, C, Win, Wout)
    # ============================================
    DHW, HHW_1, HCW, loss, CN = BackPropagation(dn, cn, Win, Whidden_2, Wou
    loss_total.extend(loss)
    wrong, correct = real_test(cn, CN)
    wins.append(correct)
    losses.append(wrong)
    #print(moves(N, CN))
    # ====================================
```

```
    for i in range(1, sets):
        print("Iteration: "+str(i))
        D, C = CSV_read(N_samples, i*N_samples)
        dn, cn, [], [] = gen_arrays(D, C, [], [])
        DHW, HHW_1, HCW, loss, CN = BackPropagation(dn, cn, DHW, HHW_1, HCW
        loss_total.extend(loss)
        wrong, correct = real_test(cn, CN)
        wins.append(correct)
        losses.append(wrong)
        #print(moves(N, CN))

    #print(CN)

    graph(len(loss_total), loss_total, "Number of Iterations", "Loss", "Epo
    graph(len(wins), wins, "Number of Iterations", "Wins", "Sample set corr
    #export(DHW, HHW, HCW)
    #gen_hard_code(DHW, HHW, HCW)
    return

# ====================================
# n, number of data sets, epochs, error, rate of learning, hidden nodes

#Epochs
#tests(100, 150, 1, 0.000001, 0.83, int(100/8))
#tests(100, 150, 5, 0.000001, 0.83, int(100/8))
#tests(100, 150, 10, 0.000001, 0.83, int(100/8))
#tests(100, 150, 15, 0.000001, 0.83, int(100/8))
#tests(100, 150, 20, 0.000001, 0.83, int(100/8))
#tests(100, 150, 30, 0.000001, 0.83, int(100/8))

# hidden layers
#tests(100, 150, 10, 0.000001, 0.83, int(100/8))

# hidden nodes
#tests(100, 150, 10, 0.000001, 0.8, 1)
#tests(100, 150, 10, 0.000001, 0.8, int(100/8))
#tests(100, 150, 10, 0.000001, 0.8, int(100/4))
#tests(100, 150, 10, 0.000001, 0.8, int(100/2))
#tests(100, 150, 10, 0.000001, 0.8, int(100))
#tests(100, 150, 10, 0.000001, 0.8, int(200))

# learning rate
#tests(100, 150, 10, 0.000001, 0.01, int(100/8))
#tests(100, 150, 10, 0.000001, 0.1, int(100/8))
#tests(100, 150, 10, 0.000001, 0.8, int(100/8))
#tests(100, 150, 10, 0.000001, 5, int(100/8))
#tests(100, 150, 10, 0.000001, 10, int(100/8))
#tests(100, 150, 10, 0.000001, 25, int(100/8))
#tests(100, 150, 10, 0.000001, 100, int(100/8))
```

```
# sample sets
#tests(100, 1, 10, 0.000001, 0.83, int(100/8))
#tests(100, 10, 10, 0.000001, 0.83, int(100/8))
#tests(100, 50, 10, 0.000001, 0.83, int(100/8))
#tests(100, 100, 10, 0.000001, 0.83, int(100/8))
#tests(100, 150, 10, 0.000001, 0.83, int(100/8))
#tests(100, 300, 10, 0.000001, 0.83, int(100/8))
#tests(100, 500, 10, 0.000001, 0.83, int(100/8))

# error
#tests(100, 150, 10, 0.000001, 0.83, int(100/8))
#tests(100, 150, 10, 0.0001, 0.83, int(100/8))
#tests(100, 150, 10, 0.001, 0.83, int(100/8))
#tests(100, 150, 10, 0.01, 0.83, int(100/8))
#tests(100, 150, 10, 0.1, 0.83, int(100/8))

# final implementation
#tests(100, 150, 10, 0.00001, 0.83, int(100/8))
```