CS3105 Practical 2 Report

Alasdair Macindoe

September 4, 2017

Introduction

Outline

The aim of this was to build a feed forward artificial neural network capable of playing Tic Tac Toe using error back propagation. In this submission I have build a strictly feed forward artificial neural network using a sigmoid activation function, back propagation, semi-supervised learning and an analysis for various neural network topologies and parameters.

Acknowledgements

I would like to acknowledge the explanations given in the lectures, including the pseudocode on error back propagation. Additionally I would like to acknowledge the explanations and pseudcode in Russell, Norvig's Artificial Intelligence Third Edition, and finally also acknowledge the explanations and pseudcode given at `https://www.cse.unsw.edu.au/~cs9417ml/MLP2/`. These were all instrumental in the completion of this project.

Layout and Execution

Please read `README.md` for instruction on how to install and execute various parts of this submission. This section will cover the overall layout of the submission, full explanations of each file as well as any relevant data and graphs.

Raw data can be located in `data` folder. Sample networks (both in Pickle and JSON) are in `networks` folder. And finally graphs can be located in `graphs` folder.

Data Formats

I have typically chosen to use pickle[1] for data because, in my opinion, it is the best way for Python specific data to be transferred. When required, or in the

[1] `https://docs.python.org/3/library/pickle.html`

interests of readability, I have stored some data as JSON. Intuitively any data stored in a `.pickle` file is not human readable and is the output of Python's pickle library, and any data stored in `.json` file is human readable-ish produced by Python's JSON library and should be able to be parsed by any JSON parser. If given the choice between JSON and pickle I suggest the pickle because it is easier to test that it works because it is one line to import in Python.

Future works

Whilst I am very happy with my submission, given more time I would like to have implemented a perfect playing Tic Tac Toe UI to compare my networks against and use to create fully unsupervised training sets.

Testing

Due to the nature of this practical it is hard to test whether the neural network itself is working correctly. There is a test suite provided which requires `PyTest` to run that tests various aspects of the code to ensure good code coverage. View README.md for more information.

Neural Network Topology

In the folder `networks` you will find several sample networks:

1. 9-9-9-network.json
2. 9-81-9-network.json
3. 9-4-8-1-network.json

With each number referring to the number of neurons in each layer, as an example a 1-2-3-4 network would have an input layer of 1 node, and two hidden layers of 2 and 3 nodes with an output layer of 4 nodes. These networks have all been assigned random initial weights for all their weightings with the exception of the input nodes that always have input weights 1. All of these are pre-training.

Note: In the analysis part I create other networks which are also included in the `networks` folder can have a slightly different naming system. For the experimental part these are of limited use which is covered more in the analysis section.

All of these can be created from within `game.py` using the following snippet of code:

```
g = Game()
nn1 = g.create_nn()
nn1.new_initial_layer(9, 9)
nn1.new_random_layer(4, 9)
nn1.new_random_layer(8, 4)
nn1.new_random_layer(9, 8)
```

This creates a random 9-8-4-9 network for use within the game.

9-9-9 Network

The most basic network design. The input is the current board state and the output is a weighting for each of the positions on the board, i.e the highest value is the best node. Note: The best move may be a move that has already been made, effectively meaning that the best move is not the move at all [2]. Obviously

[2]`https://youtu.be/MpmGXeAtWUw?t=75.com`

the game logic takes this into account and will always make a move if there is a legal move to make.

This, in my opinion, is the most basic strategy. It considers that at all points in the game certain locations are inherently just more "valuable", and will attempt to make moves there first. I believe that the downside to this strategy is that there is only a small amount of consideration made for what locations have been played so far.

9-81-9 Network

At the opposite end of the spectrum is this. It considers all 9 inputs in each of the 81 nodes ($9x9 = 81$), effectively building a neural network that learns each individual move and possible move through the training data.

I believe that this should give the best results since it does literally consider each state, but it is probably the slowest of them all due to the mass of calculations.

9-4-8-9 Network

This is a theory, which I will investigate later, but I believe that this can model Tic Tac Toe as well as the `9-81-8` network could. Since the board is symmetrical there are four possible strongest locations (the edges) for the first move and after that there are eight (described in Appendix 0.1 possible combinations that lead to victory in Tic Tac Toe.

Nine Inputs/Outputs

All of these networks have 9 inputs and 9 outputs. I have considered 9 inputs because it is important for a network to have access to the entire state of the universe rather than to just react with incomplete information, further since Tic Tac Toe is a complete information game I believed that having fewer than 9 inputs would be an unecessary handicap on my networks.
I have considered only 9 outputs because I don't see any intuitive way to not have 9 outputs. The networks each take in the state of the Tic Tac Toe board and output the strength of each location as the new position to play.

0.0.1 Error back propagation

In `data/9-9-9-pre-train.json` you can find a JSON representation of a the 9-9-9 neural network, and in `data/9-9-9-post-train.json` you can find the same network after the 10 game training set has been applied to it.

Analysis and Experimentation

Comparison

Throughout I have, unless otherwise stated, compared my neural networks to a basic AI which just chooses a random legal move, I done this because I do not believe that we can consider an AI to be "good" if it under performs in comparison to random actions.

Again, unless otherwise stated, these networks have been ran using a learning rate of 2 and an alpha value of 0.5. Additionally each network as been considered over 2500 games, and recreated (with random initial values) and retrained (using constant data which is covered later) 50 times to give an average performance for this data. The computer is NAUGHT and always moves first. At a later point these are varied but it will be clear in advance when any deviations from this set up happen.

Semi-supervised Training

In the folder entitled `data` you will find the following files:

1. ex-2.pickle
2. ex-10.pickle
3. ex-20.pickle
4. ex-30.pickle
5. ex-40.pickle

Which contains data created using methods found within `game.py`. In effect what happens is that a random (legal) board is generated and I was asked what move I would make for a that board. Then both the start board and the board after my move was added to these files. This allows us to train the network with specific data and varying amounts of data to see the effect.

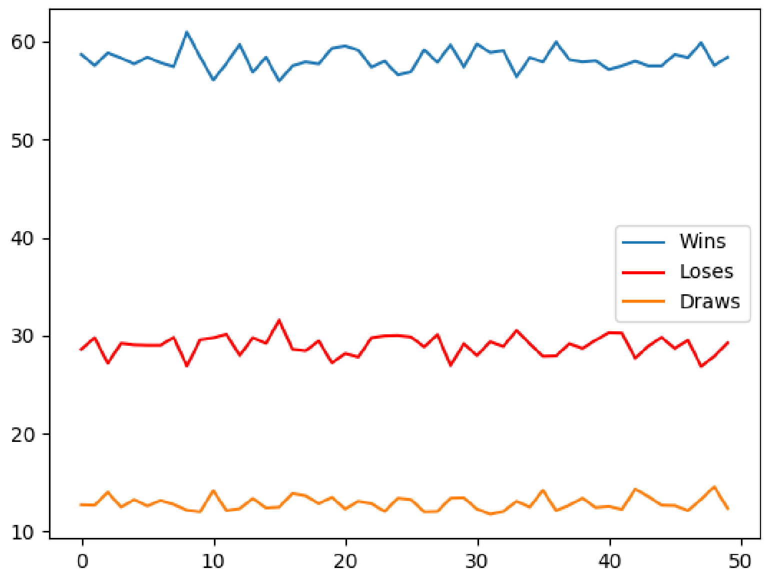

Figure 1: Random vs Random Baseline

Data Output

Where ever possible I have attempted to store the data output from the execution of my program in the folder `data` encoded with `pickle` for good scientific process. This will allow the recreation of any graphs I have in my report unless otherwise stated.

Comparison: Random vs Random

By allowing two random moves to compete we get a base case to compare my networks to. By moving first a random AI will win approximately 58% of the time and only loses 29% of the time (as graphed in 1). The raw data is stored in `data/r-v-r-2500-50.pickle` and can be verified.

Neural Networks vs Random

Each neural network was pit against the random AI and their data can be founded in the `data` folder, likewise graphs can be found in the `graphs` folder. No training was done to the networks. Networks are as defined above (and

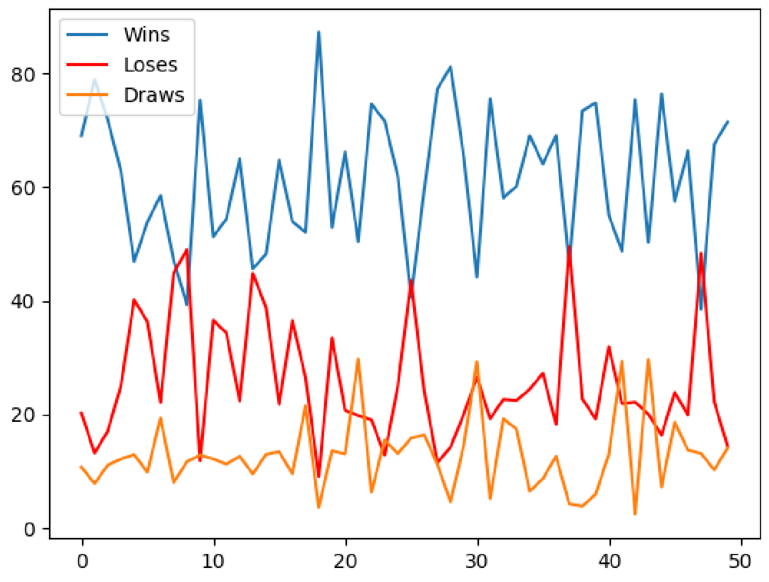

Figure 2: Neural Network (Untrained) vs Random Baseline

identified using their number).
Note: If the sum is not 100% it is from rounding.

1. Neural Network 1 - nn1-v-r-2500-50.pickle / nn1-base.png / 62%W, 26%, 13%D

2. Neural Network 2 - nn2-v-r-2500-50.pickle / nn2-base.png / 56%W, 31%, 12%D

3. Neural Network 3 - nn3-v-r-2500-50.pickle / nn3-base.png / 61%W, 26%, 14%D

These results are roughly as expected. The neural networks perform on average as good as the random choice AI does, but unlike the random choice AI there is a huge standard deviation of the results (as demonstrated by graph 2) which is explainable. Without training the networks are effectively the same as the random choice AI but instead of having an equal distribution of moves they favour moves which they have been randomly assigned higher weightings to. If these random weightings favoured edge moves then they are more likely to win. I would imagine that the standard deviation would be reduced with training.

Trained Neural Networks vs Random

Using the same training data in each instance I have generated the data for each neural network again.

20 Games

1. Neural Network 1 - nn1(20)-v-r-2500-50.pickle / nn1-20train.png / 62%W, 26%, 12%D
2. Neural Network 2 - nn2(20)-v-r-2500-50.pickle / nn2-20train.png / 59%W, 29%, 13%D
3. Neural Network 3 - nn3(20)-v-r-2500-50.pickle / nn3-20train.png / 61%W, 27%, 12%D

Training with 20 games seems to have very little effect, with only NN2 showing any real changes, but even then they were small (3% increase in wins).

40 Games

1. Neural Network 1 - nn1(40)-v-r-2500-50.pickle / nn1-40train.png / 61%W, 27%, 11%D
2. Neural Network 2 - nn2(40)-v-r-2500-50.pickle / nn2-40train.png / 60%W, 28%, 12%D
3. Neural Network 3 - nn3(40)-v-r-2500-50.pickle / nn3-40train.png / 57%W, 27%, 14%D

Conclusion

Overall training data seems to have no positive effect. I imagine that this is because of the parameters preventing it from converging.

Adaptation of Parameters

For this section I am just going to consider Neural Network 2 since it seems to have been most impacted by training. I reran the experiment with the same parameters as above, but used 100 randomly generated weights each 5000 times with no training. I then stored the best performing network in `networks/max-nn2.pickle` and `networks/max-nn2.json`. The raw data is then stored in `data/nn1-v-r-5000-100.pickle`.
This network won approximately 86 percent of the time.

Reproducibility

Step 1 is to check that this result was not because the random moves were bad but because the network chooses wisely. I reran the experiment using only that network 100,000 times to get the results. This network still won 85% of the time, only lost 8% of the time and drew the rest.

Training

Now that I know the network is not actually a good network it would be interesting to see the effects of training on this network. In this case I have reduced the amount of games down to 25,000 and used the 40 game training data set.

Interestingly after training on the 40 game data set it performs worse than it had done, dropping to 66/24/10.

Variance of Learning Rate

Since NN2 seems to perform worse after learning I will try to adapt the learning rate to see if that changes the performance. Right now the learning rate is 2. To perform this experiment I have reduced the amount of games down to 3,500.

The raw data can be found in `data/nn2-v-r-2500-50-alpha.pickle`. Alpha was varied in 0.1 increments from -2.0 to 1.9.

The results are plotted in figure 4, unfortunately no matter what the learning does not ever out perform the unlearned network.

Alpha Variance in Sigmoid

The trained network performed strongest at around 0.1 learning rate, with everything else constant I have considered varying the alpha rate in the sigmoid function to see if that will improve my network's learning. Until now that alpha has been constant at 0.1.
Once more it was varied from -2.0 to 1.9. Once more raw data can be located in `data/nn2-v-r-3500-50-alpha-sigmoid.pickle`.

This actually peaks at 86 percent win rate with alpha rate = 0.6. In fact as can be shown in `graphs/ar-wr-ext.png` increasing alpha rate will not guarantee an increased win rate. It seems that Alpha = 0.6 is a (hopefully global) minimum.

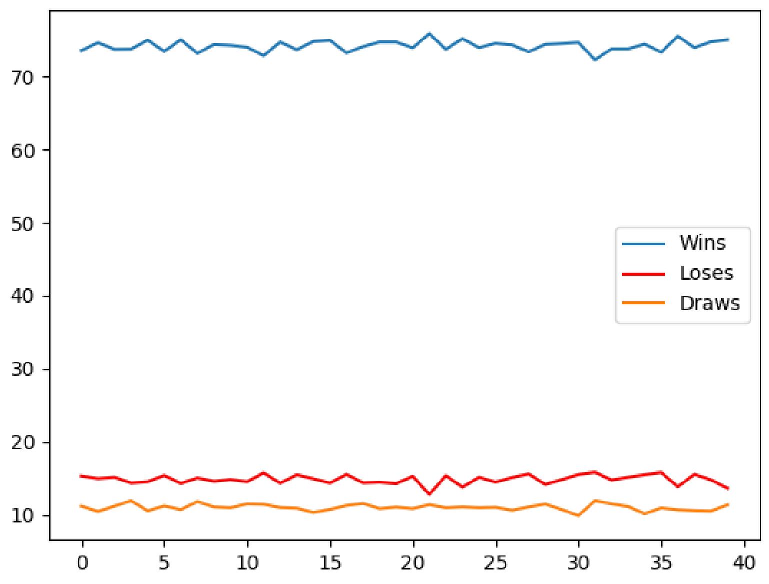

Figure 3: Effect of Learning Rate on win rate. LR is = -2.0 + index/10

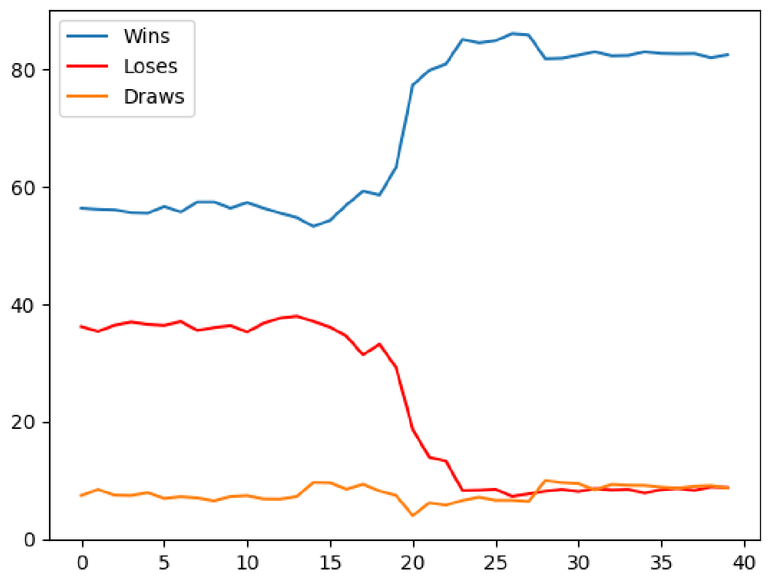

Figure 4: Effect of Alpha on win rate. Alpha is = -2.0 + index/10

Conclusion

I have managed to achieve and train a neural network (located in `networks/best-nn.pickle` or `networks/best-nn.json`) that achieves an impressive 92 percent win/draw rate against random move AI.
I done this through a combination of extensive experimentation, randomly generated default weightings and semi-supervised training.

Appendix

0.1 Methods to win

Consider Tic Tac Toe:

0	1	2
3	4	5
6	7	8

Then the possible ways to win are the following:

1. 0, 1, 2
2. 3, 4, 5
3. 6, 7, 8
4. 0, 3, 6
5. 1, 4, 7
6. 2, 5, 8
7. 0, 4, 8
8. 2, 4, 6