Using an ANN to Analyze the Effectiveness of the Martingale Gambling System

by:

Jackson A. Crowley

Mercer University

CSC 499 Undergraduate Research

Fall 2024

Abstract

The Martingale system is a strategy that involves doubling a 1$ bet each time you lose to guarantee eventually winning back your dollar. This project attempts to use an Artificial Neural Networks (ANN) to run through customly made, realistic input and output data. A custom simulation was developed to emulate real-world gambling scenarios, where it would be given an input of number of runs to do, base bet, and total funds and use a 50/50 chance to determine how much money they would have by the end of it. These datasets were then fed into an ANN to measure how well the ANN could guess the output based on the input. This can assess the feasibility of optimizing the strategy because if the ANN can reliably guess outputs, it can then be optimized based on those expected outputs. My final results show that the ANN can accurately guess the results of a run of this system, based on the inputs. There were problems along the way that may contribute to bad results such as poor weighting and redundancies in the inputs and outputs, but the performance is still impressive. These results offer insight solely into the predictability of the Martingale System, but not the effectiveness. This project attempts to give a computational perspective on a theoretically flawed gambling system by using an ANN.

Table of Contents

Introduction

As I was reading a physics textbook, I stumbled across the Martingale system of gambling for the first time. A seemingly foolproof way to gain money, you start with a 50/50 bet of 1$ and if you lose it bet 2$. This way you earned back the one dollar lost, and still gain a dollar for that run. You keep doubling the bet if you lose, and keep repeating earning 1$ each time. Why it fails, and why it was noted in the physics textbook, is eventually you will lose many, many times in a row. If you happen to lose 17 times in a row, which is improbable but if you play long enough it will happen, then you would need 131,000$ just to bail yourself out and earn that 1$ margin for the run.

The physicist knows this won't work in practice, but I am not a physicist. I wanted to test it in a more practical environment. I started with a simple simulation that a user can go through, where they pick how much money they start with , and then get to choose how much they bet, how much they start with, and how long they keep going as if they are actually playing it.

This got me familiar with the math and basic methods this code would need to generate outputs and inputs. I made a program that ran the code on just computer inputs, and gave it randomly generated inputs, and I then had the inputs and outputs for what different runs of the martingale system would look like for varying amounts of base bets, runs, and starting wallets.

I had a file of inputs and corresponding outputs. Now is the fun part. Running those inputs and outputs through an ANN I made to see any patterns, and possibly deduce an optimal way of using this strategy in relation to the time it takes to make these bets and your starting wallet. Further on is my many, many struggles with ANN's and eventually final results that will clarify the question of if the Martingale system of gambling would work in a real environment.

Research Methodology

This was a computational project that attempted to get a set of quantitative data. This quantitative data will determine how accurately an ANN can predict the outcomes of different gambling sessions based on how long they play, the starting money, and the base bet. How well they do is not measured by this ANN, only how likely the ANN was to guess how well they did. Therefore the results of this project would not show this gambling system is good, more that it can be consistently predicted. This however is crucial information for it being good.

To measure this I would need to build an ANN that can take in a data set and run it through a training algorithm so that it can get good at predicting the results. After this I will need to give it a different set of inputs to see if the things it learned would work on data other than the training data. I will be using java (through vscode) to make all of these programs and it will be run on my computer which has an i9 9700k intel cpu which allows me to easily work with large sets of data.

I believe that the ANN will get decently good at predicting the results, as there are certain formulas that you could follow to get the odds of someone losing, but because it is based on random chance it will never be perfect

Implementation

To go about this project I first had to make a random input generator that fills up an empty text file. The parameters for the input are as follows:

starting money, base bet, number of bets to do

Where $1000 >$ starting money > 0 startingmoney/10 > basebet $100 >$ numberofbets > 0

This lets us mimic a typical run of this strategy with just three numbers. I then designed a program that takes in these three numbers, and gives a return for how the gambling ended up. If it would end on a loss based on the number of bets I had it continue until it had recovered the money. The outputs were in this tuple:

number of runs, percent gained, percent gained per time, highest buyout over starting, highest buyout over bet, binary for failed run

I put these tuples into a large array of inputs, and a large array of outputs. The ANN would judge things based on this.

The only tools and dependencies I used for this were Java's IOException for error handling, Arrays, and file readers. I made the neuron class, used it to make the layers in the layer class, had another file for the training data class and a final file for all of the sigmoid functions and other math equations. I also had the main file that ran all of the code, and other than that just the random input generator and output creator were all I needed.

The most important piece of code might be the way in which I measure the first percentage, and the second. The snippet follows:

```
int howmuch = 1000;
int numberofiter = 100;
CreateTrainingData(howmuch);
System.out.println("Total amount of data working with: " + howmuch);
System.out.println("Output before training");
float summedup = 0;
for(int i = 0; i < (tDataSet.length)/numberofiter; i++) {
    forward(tDataSet[i].data);
```

```
        summedup += layers[2].neurons[0].value;}
summedup = summedup/((tDataSet.length)/numberofiter);
System.out.println("Average accuracy: " + summedup);

int pertraining = howmuch/numberofiter;
train(numberofiter, 0.1f, pertraining);
System.out.println("Output after training");
summedup = 0;
 for(int i = (tDataSet.length - pertraining); i < tDataSet.length; i++)
        {forward(tDataSet[i].data);
        summedup += layers[2].neurons[0].value;}
summedup = summedup/((tDataSet.length)/numberofiter);
System.out.println("Average accuracy: " + summedup);
```

First of all, this lets me easily change how much data I am actually pulling from the input.txt and output.txt which is prefilled with tens of thousands of entries. Secondly it lets me easily change how many training iterations I want. Originally, I would get a .999 post accuracy for all of the data regardless of learning rate, data size or anything else. That was when I realized that the data I was running the final test of the post training accuracy was in the training data pool. My solution was to still take in all the data you will need for the ANN all at once, but split it into pieces based off how big you want a training iteration to be. The first chunk will be the initial data it runs through for the pre training accuracy. The middle chunks will all go towards training, leaving the last chunk unseen by the ANN up until this point so that it can be used for the post training test. This solution was very nice as I wouldnt have to keep digging into the I/O files every time I needed new unseen data for the ANN to process.

Other than that the most important thing is the learning rate. Since my problem isn't that complex, I can get by with smaller data sizes which means I can use a slightly larger learning rate. To see the effects I recorded each data size and number of training iterations with .1, ,05, and .01. I heard that .1-.01 is a good range for simple, relatively small data.

I was unsure of how many layers to use which may affect the data. I feel that it was a bit too many inputs/outputs for no inner layer, but almost too simple to have an inner layer. I decided on going with an inner layer but I am not sure if it was the correct decision and it may make my results less reliable.

Results and Analysis

Training Data Size = 100

Learning rate Below	Iterations —---->	5	10	20	25
.01 LR	Pre Accuracy	.510	.391	.479	.813
	Post Accuracy	.957	.963	.972	.984
.05 LR	Pre Accuracy	.517	.581	.500	.642
	Post Accuracy	.973	.979	.982	.981
.1LR	Pre Accuracy	.702	.542	.423	.463
	Post Accuracy	.982	.989	.985	.991

Training Data Size = 1000

Learning rate Below	Iterations —---->	10	25	50	100
.01 LR	Pre Accuracy	.496	.588	.156	.360
	Post Accuracy	.988	.991	.994	.993
.05 LR	Pre Accuracy	.535	.353	.664	.656
	Post Accuracy	.989	.992	.996	.9998
.1LR	Pre Accuracy	.779	.615	.661	.552
	Post Accuracy	.992	.992	.994	.999

The data I collected tends to show a positive trend of a higher learning rate yielding better results. One reason I think this is true is because the smaller data set and relatively simple problem it's solving allows for a higher learning rate without loss of accuracy. Also the more you divide the data into chunks to process at a time, the better it seems to run. It is going through each of these chunks a set number of times, so the smaller you make the chunks the more it goes through data, even if it is the same data.

One fault of mine might lie with the input and outputs being too similar and relatable, and with the weighting. The number of runs will always directly tie to the time spent, and there are similar redundancies. The other problem was with the weighting, where I kept everything the same, but there are certain things that are more important. If you are doing a high volume of runs you are much more likely to strike it out, but it has the same weight as everything else while going through the neurons of the layers.

Overall I think this shows that an ANN can predict with strong accuracy how a run will turn out based on the inputs. This makes sense as you could have a function that tells you the odds of if someone loses everything based on the base bet, their wallet, and number of runs. Similarly you could take a guess at how much they would make as you could easily find out the average number of runs it takes to earn your 1$ back and just use multiplication. You could specifically change a node to represent these things and it would do it quite well.

Conclusion

While the Martingale system is flawed from a theoretical view, in practice it can be easily predicted. Since we can easily predict it we could optimize it and see if it truly is flawed in the real world and if we would ever need to bail ourselves out for 130,000$. The ANN I built successfully predicted results with a 98-99% accuracy in some cases. I still need to continue this to remove any room for doubt that comes in obvious redundancies in the inputs and outputs making the outputs obvious, and issues with weighting. However, even with these the results offer valuable insight on emulating the Martingale system and the predictability of it. Other than facing the already stated problems, a direction I would like to take this project in is a hosted website that offers random users a way to gamble using the martingale system. The inputs and outputs of their runs could then be fed into a database that I could continue to run an optimized ANN on to see if these results hold true for more real trials. This project taught me a lot on what an ANN is, how to build one, and how to give it data and mess with the settings. My progress on this was much slower than I was hoping. At the beginning of this project I didn’t know what an ANN was, and debugging an ANN is no small feat. Along with having to re-familiarize myself with basics of coding I haven’t touched in a while, I feel there is a lot left to do. I hope to pursue this project more in the future so I can eventually one day test it for myself at the casino.

References

“Neural Network from Scratch in Java“ *YouTube*, uploaded by Deep Learning with Yacine, 27 Dec. 2018, https://www.youtube.com/watch?v=1DIu7D98dGo.

Note- This was the template of a basic ANN in java that I used for my project and a very helpful learning resource.