Machine Learning and Statistical Toy Models with Applications in Business, Economics, and Finance: A PHYS295 Research Topic

Nathan McCarley[a]
Corresponding author: Dr. Stefan Jeglinski[a]; jeglin@physics.unc.edu
[a]Physics, University of North Carolina at Chapel Hill, Chapel Hill, USA

September 14, 2023

Abstract

An important part of making financial decisions is being able to justify those decisions; doing so stems from an understanding of why those decisions would be chosen over others. While machine learning models can provide impressive predictions and classifications that aid in making said decisions, the nuances of how and why trained models behave the way they do can be quite abstract; often, the high dimensional spaces that machines classify data into have no meaning to human beings. This disconnect creates a barrier to our understanding of why a model would provide a certain prediction, causing distrust in the model's output. Creating so-called "toy models" — simple applications of complex machine learning techniques — allows one to develop further trust in those models. This is a necessary step in being able to successfully apply machine learning techniques in financial decision-making or macroeconomic forecasting, and lends itself to the application of more advanced models.

Keywords: factorization; propagation; neural networks; finance; time-series

1 Generative Adversarial Networks (GANs)

1.1 Introduction

GANs feature two neural networks that compete against each other: one that generates a synthetic data set resembling a target data set, and another that discriminates between the real versions of the data set and the "fake" versions. This artificial intelligence duo lends itself to applications such as fraud detection in order to discriminate between real credit card transactions and fraudulent ones, or generating realistic synthetic data for training financial models. Again, consider a simpler model: a basic GAN that not only generates images of squares, but also determines if an image is a square or not.

1.2 Theoretical Background

1.2.1 Initialization

Each iteration of training, the image that the Discriminator looks at shall be represented by the matrix X of dimension (n, n), where each matrix entry ranges from 0 to 1 and describes whether its corresponding pixel in the image is light or dark. To keep this example simple, $n = 4$. The matrix X is then reshaped to dimension $(n \times n, 1)$ and fed into the Discriminator.

1.2.2 Activation Function

Both the Discriminator and the Generator use the sigmoid activation function in Eq. 1 in the secondary part of their calculations (Eq. 4 and Eq. 6, respectively).

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

Performing this nonlinear transformation in succession with the linear transformations in Eq. 3 and Eq. 5 enhances the network's ability to classify and generate data. This method of layering linear and nonlinear transformations is a fundamental part of most neural networks today. This specific nonlinear transformation - sigmoid - also lends itself to backwards propagation in this model; the recursive nature of sigmoid's derivative in Eq. 2 allows for simplifications to be made in the backwards propagation equations.

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{2}$$

1.2.3 Forward Propagation

The Discriminator takes in X and applies the transformation in Eq. 3 to produce Z_D. This is done by multiplying the weight matrix W_D of dimension $(1, n \times n)$ by X, then adding the bias term b_D.

$$W_D X + b_D = Z_D \tag{3}$$

After this, the sigmoid activation function is applied in Eq. 4 to produce A_D, the Discriminator's score.

$$\sigma(Z_D) = A_D \tag{4}$$

This score reflects the level to which the discriminator believes the image is a square - its confidence level. If the confidence level is above the threshold γ, the Discriminator decides the image is real; otherwise, it decides the image is not a real square. In order to generate these images, the Generator starts out with the matrix Z of dimension $(n \times n,\ 1)$ filled with random values between 0 and 1. As seen in Eq. 5, the Generator performs element-wise multiplication by the weight matrix W_G, then performs element-wise addition with the bias matrix b_G to produce Z_G.

$$ZW_G + b_G = Z_G \tag{5}$$

Next, the sigmoid activation function is applied in Eq. 6 to yield A_G, the generator's attempt at making a square.

$$\sigma(Z_G) = A_G \tag{6}$$

Reshaping A_G back to $(n,\ n)$ allows it to be plotted against our target image for visual comparison.

1.2.4 Binary Cross-Entropy Loss

Sometimes, the input image X is a real image of a square; other times, the input is an image the Generator has come up with in an attempt to fool the Discriminator. In this model, the Discriminator goes back and forth between the real image and the images made by the Generator. Each time, the accuracy of the Discriminator and/or the Generator is measured using the error function in Eq. 7 for real images,

$$E_D = -\text{Ln}(A_D) \tag{7}$$

and the error functions in Eq. 8 and Eq. 9 for "fake" images.

$$E_D = -\text{Ln}(1 - A_D) \tag{8}$$

$$E_G = -\text{Ln}(A_D) \tag{9}$$

This can be generalized into the single equation in Eq. 10 by letting $E_X = 1$ when X is a real image, and $E_X = 0$ when X is a synthetic image from the Generator.

$$E = -E_X \text{Ln}(A_D) - (1 - E_X)\text{Ln}(1 - A_D) \tag{10}$$

1.2.5 Backwards Propagation

Fake Image:

$$\frac{\partial E}{\partial W_D} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial W_D} \tag{11}$$

$$= \frac{1}{1 - A_D}\sigma'(Z_D)A_G^T \tag{12}$$

$$= A_D A_G^T \tag{13}$$

$$\frac{\partial E}{\partial b_D} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial b_D} \tag{14}$$

$$= \frac{1}{1 - A_D}\sigma'(Z_D) \tag{15}$$

$$= A_D \tag{16}$$

$$\frac{\partial E}{\partial W_G} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial A_G}\frac{\partial A_G}{\partial Z_G}\frac{\partial Z_G}{\partial W_G} \tag{17}$$

$$= -\frac{1}{A_D}\sigma'(Z_D)W_D^T\sigma'(Z_G)Z \tag{18}$$

$$= (A_D - 1)W_D^T\sigma'(Z_G)Z \tag{19}$$

$$\frac{\partial E}{\partial b_G} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial A_G}\frac{\partial A_G}{\partial Z_G}\frac{\partial Z_G}{\partial b_G} \tag{20}$$

$$= -\frac{1}{A_D}\sigma'(Z_D)W_D^T\sigma'(Z_G)1 \tag{21}$$

$$= (A_D - 1)W_D^T\sigma'(Z_G) \tag{22}$$

Real Image:

$$\frac{\partial E}{\partial W_D} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial W_D}$$

$$= -\frac{1}{A_D}\sigma'(Z_D)X^T$$

$$= (A_D - 1)X^T$$

$$\frac{\partial E}{\partial b_D} = \frac{\partial E}{\partial A_D}\frac{\partial A_D}{\partial Z_D}\frac{\partial Z_D}{\partial b_D}$$

$$= -\frac{1}{A_D}\sigma'(Z_D)1$$

$$= (A_D - 1)$$

1.2.6 Update Parameters

$$W_D = W_D - \left(\alpha \times \frac{\partial E}{\partial W_D}\right) \tag{23}$$

$$b_D = b_D - \left(\alpha \times \frac{\partial E}{\partial b_D}\right) \tag{24}$$

$$W_G = W_G - \left(\alpha \times \frac{\partial E}{\partial W_G}\right) \tag{25}$$

$$b_G = b_G - \left(\alpha \times \frac{\partial E}{\partial b_G}\right) \tag{26}$$

1.3 Methods

We start with the target image — what the generator is ultimately trying to replicate.

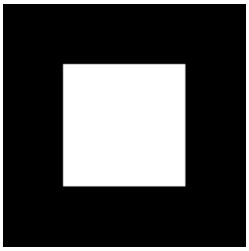

Figure 1: A real image of a square in the 4×4 pixel environment; this is our target image.

Using a learning rate $\alpha = 0.05$, confidence threshold $\gamma = 0.8$, and running for 3000 iterations, both the discriminator and generator were trained. Each iteration, the following process happened:

- Run the Discriminator on the real image of a square

- Update Discriminator weight and bias matrices
- Have the Generator produce an image
- Run the Discriminator on this generated image of a square
- Update Discriminator and Generator weight and bias matrices.

1.4 Results and Discussion

Every 20 iterations, the current version of the Generator's output was captured and stored in Fig. 2. The final image in the sequence — the 400$^{\text{th}}$ iteration, was relatively where the Generator's output passed the eye test.

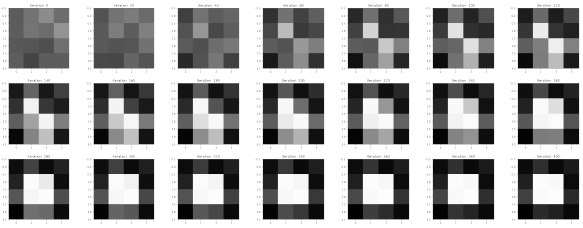

Figure 2: Starting with the initial random output of the untrained Generator, we see the learning process as the images visually become more like our interpretation of a square. The final image in the sequences corresponds to the 400$^{\text{th}}$ iteration.

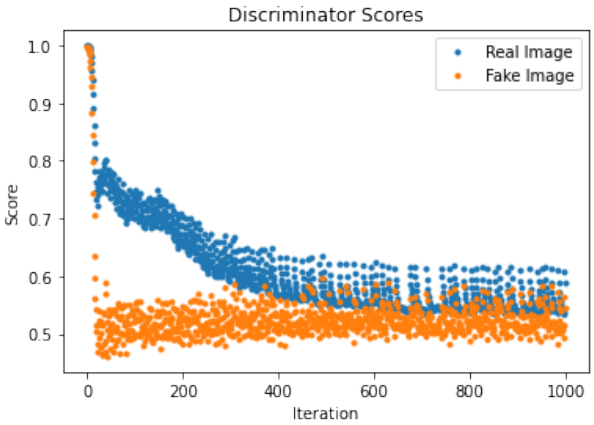

Figure 3: Discriminator scores at each iteration for both the real image and the generated image.

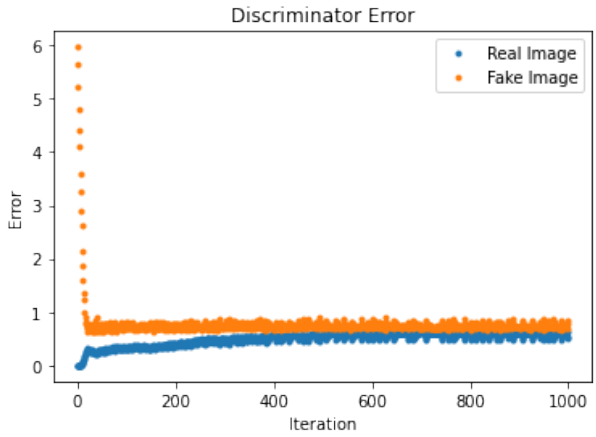

Figure 4: The Discriminator's error at each iteration for both the real image and the generated image.

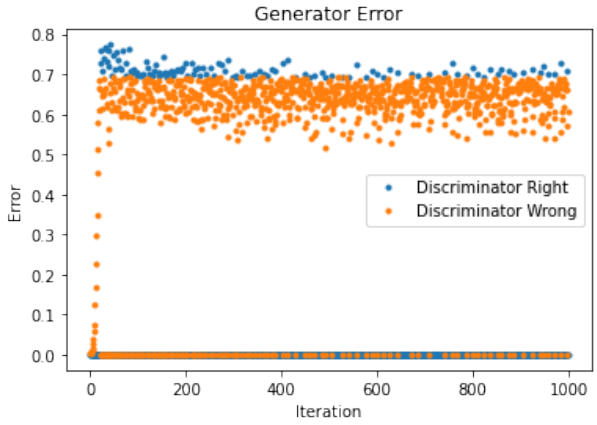

Figure 5: The Generator's error at each iteration, classified by whether the Discriminator was able to correctly classify the generated images.

1.5 Documents

Google Colab Notebook

2 Genetic Algorithms

2.1 Theoretical Background

Genetic algorithms emulate characteristics of evolution in an attempt to find the optimal solution to a problem. First, they start with a population representing potential solutions as the first generation. Next, they use a fitness function to evaluate the fitness of each member of the generation. This is similar to a loss function that judges potential solutions in other algorithms; instead of finding a set of parameters that minimizes the error, we are finding the set that maximizes fitness. Then, using the fitness scores, they probabilistically select members of the current generation to pass on their genes. Solutions (or population members) with more fit attributes have a higher chance of surviving. After the gene pool is chosen, a combination of random

crossovers and mutations produce part of the next generation of solutions. The remaining part of that generation will be direct copies of the top solutions from the current generation, so that those are preserved in case the crossovers and mutations destroy the best solutions. This process is repeatedly done until the solution population converges on the most fit solutions — or genetic diversity stops decreasing.

2.2 Methods

2.2.1 Initialization

In this experiment, each member of the population has a chromosome consisting of five genes with values equal to either 0 or 1. Visually, 1s are represented by black squares, and 0s by white squares. Each generation has a population size of 50 members; Fig. 6 shows the first generation with each member's chromosome displayed across its own row.

Figure 6: The first generation's randomly generated genetics with 5 chromosomes per member (columns) and 50 members per generation (rows). White squares are 0s, black squares are 1s.

2.2.2 Fitness Function

The fitness of each member is just the number of 1s, or black squares, that appear in its chromosome. This means that a chromosome of uniform 1s/black squares is the optimal chromosome; this can be seen in Fig. 7. Using a simple fitness function such as this makes it easy for humans to understand what the result should look like, making it easier to judge the algorithm using a clear visual benchmark.

Figure 7: Intuitively, the best possible genome a member could have — all 1s. This is what a successful algorithm should end up producing.

2.2.3 Gene Pool Selection

Selection is simple; the fitness score of each member is compared to a random integer value between 0 and 5. If the member's fitness score is higher, the member is selected for reproduction. This is repeated for all members to form the next gene pool. To help in the crossover step later on, it was ensured that there would be an even number of chromosomes in the selection pool. Although a more advanced algorithm might normalize the fitness scores, keeping it simple in this application makes it easier to follow.

2.2.4 Single Point Crossover

To produce the offspring of the selection pool, two parents are chosen and their chromosomes are examined such as in Fig. 8. The rows carry the gene values.

Figure 8: Two parent chromosomes before crossover.

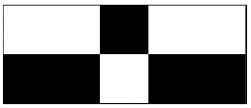

Figure 9: Two resultant child chromosomes after the crossover swaps the last two genes of the chromosome.

At a random point in the chromosome, a cut is made and the remaining part of each parent's chromosome is swapped. In Fig. 9, the first three genes are kept, and the last two genes are swapped. The parents — with fitness scores of 3 and 2 — produced offspring with fitness scores of 4 and 1. This is an improvement, as the better combination of the parent genes will likely survive and pass on more copies, while the less fit combination will likely be eliminated from the pool.

2.2.5 Top Solution Preservation

2.2.6 Mutation

To fill the rest of the next generation and get the population up to 50, parents are randomly chosen from the selection pool and mutation is performed on them. At a random index in their chromosome, the existing value is swapped for the other possible value — 1s turn to 0s, and 0s turn to 1s. This mutated copy is then added to the next generation, and the process repeats as needed.

2.3 Results and Discussion

The initial generation followed by the first 15 generations in Fig. 10 visually demonstrate the quick convergence of the algorithm. Knowing ahead of time what to look for using Fig. 7, examination yields confirmation of expectations.

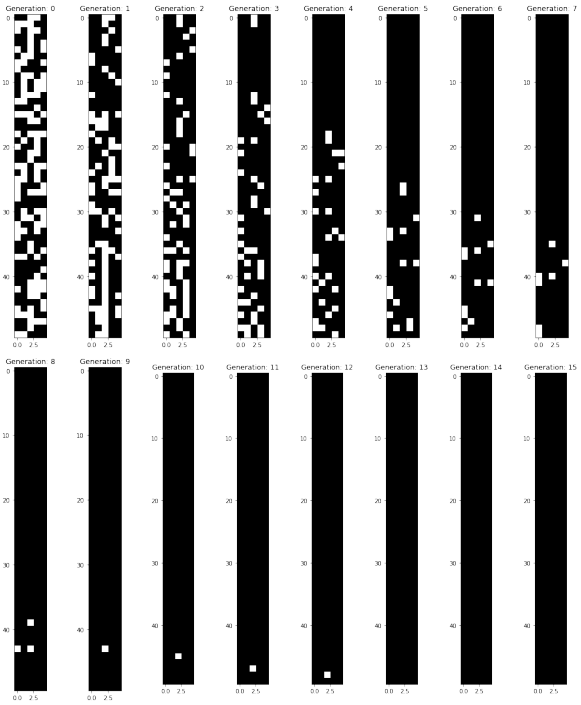

Figure 10: Changing genomes yield fewer occurrences of white squares throughout subsequent generations.

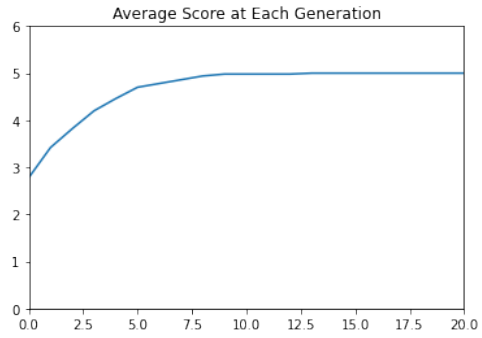

Figure 11: Average score at each generation.

2.4 Documents

Google Colab Notebook

3 Long Short-Term Memory (LSTM) Neural Networks

3.1 Introduction

One of the requirements for making educated decisions in a marketplace is the ability to accurately predict future behaviors. More specifically, if a company that sells coffee wants to estimate how much supply to buy in order to meet the demand of the coming months, it is important to know how demand will fluctuate based on the current trends. By looking at past demand and demand-impacting factors as a group of time-series data, we can use Artificial Neural Networks (ANNs) to learn the trends and cycles; this allow us to estimate future demand for a given point in time.

3.2 Theoretical Background

3.2.1 LSTMs

LSTMs are a specific type of Recurrent Neural Network (RNN) that take an input sequence - often as a vector or tensor - and then form an output sequence or vector. It does so by learning to apply its 3 main processing techniques (Gates):

- Forget Gate: how much to rely on recent data points versus older data points in the time series,
- Input Gate: what to interpret from the data at each time step,
- Output Gate: what information to pass on to the future time steps.

This lends itself to making predictions based off of time-series data; looking at a broad view of the economy, one could use macroeconomic factors (GDP, manufacturing, growth across sectors, employment, inflation, and CPI) to predict the future of the economy by those metrics. This idea will be revisited, but first a basic LSTM must be tested to ensure confidence in the model's prediction abilities.

To make sure the predictions of the LSTM match the actual future values, it is helpful to perform an analysis on an intuitive function: the sine wave. Looking at its graph, it is easy to determine which direction the function will go next due to its simple and repetitive behavior. By training the LSTM on samples that contain the last n values of the sine function at a given point in time, the model can then make predictions for where the sine wave will go at whichever location in time is chosen.

3.2.2 Model Specifics

1. Create sample data of a sine wave
2. Scale data using StandardScaler
3. Restructure data into X and Y
4. Split data into training sets and testing sets
5. Create LSTM Model (Sequential)
6. Fit Model
7. Make predictions and evaluate them

3.3 Methods

3.4 Results and Discussion

3.5 Documents

Google Colab Notebook

4 Matrix Factorization

4.1 Theoretical Background

Recommendation algorithms factor a partially-populated matrix to predict the values of matrix elements that have not yet been filled in. An application of this method is the Netflix Recommendation Algorithm, which attempts to predict the movies people will like based on their perception of the various "features" associated with movies in general (genres for example), and the amount the movie in question displays each of those "features". The algorithm uses error functions and gradient descent to find the right values of the "features" to appear in the factors of the matrix. Matrices with fully-entered data can also be factorized, usually for describing each input's perception to the primary underlying factors that categorize the data. An example factorization recommendation algorithm and mathematical derivation are provided.

$$R = PQ \tag{27}$$

$R \in \mathbb{R}^{u \times d}$, $P \in \mathbb{R}^{u \times k}$, and $Q \in \mathbb{R}^{k \times d}$.

4.1.1 Forward Propagation

Matrix multiply the current versions of P and Q

$$PQ = \hat{R} \tag{28}$$

$$\hat{R} \to \hat{R}_0 \tag{29}$$

$$E = \hat{R}_0 - R \tag{30}$$

$$C = E^2 = (\hat{R}_0 - R)^2 \tag{31}$$

4.1.2 Backwards Propagation

$$\frac{\partial C}{\partial P} = \frac{\partial C}{\partial \hat{R}} \frac{\partial \hat{R}}{\partial P}$$

$$= -2EQ^T \tag{32}$$

$$\frac{\partial C}{\partial Q} = \frac{\partial C}{\partial \hat{R}} \frac{\partial \hat{R}}{\partial Q}$$

$$= -2EP^T \tag{33}$$

4.2 Methods

4.3 Results and Discussion

4.4 Documents

Google Colab Notebook

5 Markov Chains

5.1 Theoretical Background

By modeling a series of connected websites as a series of nodes between which users can travel, we can model the population at each node using the entries of a column vector, and we can represent the flow between nodes with a matrix of transition probabilities detailing the likelihood of switching to one website from another. Repeatedly applying the transition matrix to the vector of distributions will - in the right cases - yield a steady-state distribution. This distribution shows which nodes users spend the most time on, granting us the ability to rank these nodes; this is the basis for Google's original PageRank algorithm. Similarly, one could model traffic flowing through highways in this manner, or even track the market share of companies as brand loyalty varies; this section of the project will model the latter option.

5.2 Methods

5.3 Results and Discussion

5.4 Documents

Google Colab Notebook

6 Extensions

6.1 Introduction

References

1. Sak, H., Senior, A., & Beaufays, F. (2014). Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. arXiv preprint arXiv:1402.1128.
2. Wang, Yang. (2020). A Mathematical Introduction to Generative Adversarial Nets (GAN).
3. Y. -X. Wang and Y. -J. Zhang, "Nonnegative Matrix Factorization: A Comprehensive Review," in IEEE Transactions on Knowledge and Data Engineering, vol. 25, no. 6, pp. 1336-1353, June 2013, doi: 10.1109/TKDE.2012.51.
4. Melanie Mitchell. 1998. An Introduction to Genetic Algorithms. MIT Press, Cambridge, MA, USA.
5. Mahfuz, Fariha, "MARKOV CHAINS AND THEIR APPLICATIONS" (2021). Math Theses. Paper 10.