School of Computer Science and Information Technology |
University College Cork (ucc.ie)

Generative Adversarial Networks for time series forecasting

David Shanahan

Supervisor: Dr James Doherty

April 24, 2024

FYP for
Data Science and Analysis

Abstract

Financial Time series Data is incredibly hard to model due to the non-linear complexity of financial markets. This study explores the potential of deep learning in capturing market dynamics, proposing a Generative Adversarial Network which positions Long Short Term Memory networks as the generator and Convolutional Neural Networks as the discriminator, leveraging their strengths to capture the trends and patterns in the data. The research examines the use of variants for GANs, with Wassterstein GAN (WGAN) and Metropolis Hastings GAN (MHGAN). WGAN uses the Wassterstein distance as a loss function with the goal of enhancing training stability while MHGAN incorporates Metropolis Hastings sampling into GANs, refining the sample selection process. The investigation into GAN variants, as contrasted with the LSTM model, underscores the potential benefits of employing GANs for time series forecasting. This comparison aims to highlight the robustness of GANS in a domain that is notoriously hard to predict.

Declaration of originality

Declaration of Originality In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

I hereby declare that:

• this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;

• with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;

• with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: David William Shanahan. . . . . .

Date: 24/04/2024 . . . . . .

Acknowledgements

Thanks Everyone!

I would like to thank by supervisor Dr James Doherty for support and guidance throughout the project and Prof. Gregory Provan for the group tutorial that helped build my understanding of the topic. I would also like to thank James Delaney for helping me get set up on the GPU computer in the Western Gateway Building.

I would also like to thank my family and classmates for support and encouragement throughout the project.

Contents

List of Figures

List of Tables

1 Introduction

Artificial intelligence in finance has been a research area of increasing interest for the last few decades, especially in areas like stock trading. Predicting market fluctuations, studying consumer behavior, and analyzing stock price dynamics are examples of how investors can use machine learning for stock trading. The prediction of stock prices offers substantial rewards to investors and analysts who can forecast market trends accurately. However, the inherent volatility and non-linearity of the stock market, influenced by a myriad of factors, make this task very difficult. Stock market prediction is essentially a time series forecasting problem. Time series problems add the complexity of temporal dependencies between observations. Traditional time series forecasting methods like ARIMA suffer from problems like assuming linear relationship and excluding more complex distributions [11] and often fall short due to their inability to capture complex market dynamics [12].

Owing to recent advancement in deep learning techniques, deep learning models are gradually replacing traditional learning models as the choice for price forecasting. Deep learning neural networks are able to automatically learn complex mappings from inputs to outputs and support multiple inputs and outputs [13]. These are powerful features that offer a lot of promise for time series forecasting, particularly on problems with complex-nonlinear dependencies, multivariate inputs, and multi-step forecasting. Through its ability to model high-level abstractions in data, deep learning offers a nuanced understanding of market dynamics, enabling more accurate and robust forecasting models. Deep learning models such as Convolutional neural networks (CNN) and Long Short term memory (LSTM) have shown great accuracy in time series forecasting [14].

LSTMs, a class of recurrent neural networks, excel in capturing temporal dependencies, a crucial aspect of time series data like stock prices. Their unique architecture allows them to remember information over long sequences, making them ideally suited for modeling the sequential patterns observed in financial markets [15]. CNN can learn to recognise and extract important features from past data points and use these features to make predictions about future values. CNNs approach forecasting by using 1D convolution for prediction. This makes sense because a 1D convolution on a time series is roughly computing its moving average or using digital signal processing terms, applying a filter to the time series. By

applying convolutional layers to historical stock price data, CNNs can identify patterns that are not immediately apparent, capturing underlying trends and cycles that influence future prices [16].

Generative Adversarial networks (GANs) are an approach to generative modeling that can be used in the context of stock price prediction. GANs can generate synthetic yet plausible market data using historical data to produce future stock prices. It comprises two neural networks, a generator and a discriminator, that are trained simultaneously in an adversarial manner. The generator is trying to generate data that is indistinguishable from the true data, while the discriminator tries to differentiate between the real and generated data, identifying features that distinguish the real from the artificial [5]. If we were to think of the generator as an art forger, crafting imitations of the art that could be mistaken for the originals and the discriminator as an art detective, whose job it is to scrutinise each painting to determine whether the painting is real or fake. As the forger refines their work based on the detective's evaluation, the forgeries become better and better until the detective only has a 50/50 chance of identifying whether a painting is genuine or fabricated.

Our approach uses Long Short Term Memory network as a generator, selected for its capabilities in capturing long term dependencies within time series data. Convolutional Neural Network serves as the discriminator, for its effectiveness in identifying and distinguishing important features from the data.

Wasserstein GAN introduces an alternative loss function based on the wasserstein distance. This modification aims to improve training stability, addressing some critical issues like mode collapse, thereby fostering the generation of higher quality data that provides more stable training [6]. Another variant of GAN is the Metropolis Hastings GAN, which incorporates Metropolis Hastings sampling into the GAN architecture. In this model, the discriminator is used after initial training process to select which generator samples to accept for use. This integration enhances the selection process of generated samples, refining the model's capability to learn the underlying data distribution and enhancing its predictive capabilities for stock price movements [7].

The core objective of this research is to harness the power of Generative Adversarial networks and its variants in modeling and predicting the closing stock price movements using Google stock data from 2010 to 2022. This study evaluates the predictive accuracy of these models compared to the LSTM model. By examining the GAN and its variants, WGAN and MHGAN, the research aims to contribute better understanding to the use of GANs in the field of Time Series forecasting.

The ensuing sections will guide the reader through the Related work and Background Theory, Dataset and Feature engineering, Methodology, Results and Conclusions, offering a comprehensive exploration of the process employed in this study.

2 Related Work

Time series forecasting is a statistical technique used to predict future values based on past data points. Classical algorithms like Autoregression Integrated Moving Average (ARIMA) works great on time series data that is linear and stationary, but may not perform as well with nonlinear and non-stationary data like those in the stock market. In the 2018 paper 'A Comparison of ARIMA and LSTM in Forecasting Time Series' by Siami-Namini et al [15], it advocates for the superiority of LSTM over ARIMA in time series forecasting. The authors conducted empirical studies comparing ARIMA and LSTM models using various financial and economic time series datasets. In the paper the forecasting challenges associated with financial time series data are highlighting the unpredictable economic trends and incomplete information. It asks the question whether traditional models like ARIMA have similar accuracy and precision scores compared to deep learning based forecasting methods. LSTM is chosen due to its ability in preserving and training the features over a longer period of time. The study finds that the LSTM gives an average reduction in error rates of between 84-87 percent when compared with the ARIMA model. It concludes that despite the higher computational expense of deep learning methods like the LSTM, the increase in predictive accuracy they offer when compared to traditional stochastic models justifies the increased computational cost.

The exploration of the use of GANs for stock market prediction was investigated by Zhang et al in the 2018 paper 'Stock Market Prediction based on Generative Adversarial networks' [17]. The paper introduces a unique GAN framework where the LSTM network is used as the generator, and a Multi-Layer Perceptron is used as the discriminator. The method aims to forecast the daily closing price of stocks by generating data distributions similar to actual stock market data. The generator is designed by LSTM because of its ability in processing time series data . It mines data distributions from historical stock data, using several financial indicators to predict future closing prices. The discriminator evaluates whether the given data is real or generated by the LSTM generator. It aims to create a differentiable function that classifies the input data. MLP with three hidden layers is used to process the input data, outputting 0 for fake data and 1 for real data using a sigmoid activation function. The results indicate that the proposed GAN model outperforms

traditional models like LSTM, ANN and SVR in predicting the closing stock prices. The proposed GAN architecture performs better on error metrics such as MAE, RMSE and MAPE. The paper demonstrates the potential of using GANs for more accurate stock market predictions. The authors suggest that their models could be further improved by incorporating more significant financial factors to enhance the accuracy of trend and price predictions.

Ricardo Alberta Carrillo Romero also explored the use of 'Generative Adversarial network for Stock Market Price Prediction' [18], comparing its effectiveness with traditional deep learning models and the ARIMA model. The GAN architecture used has a three layer dense network as a generator and a three layer Convolutional Neural Network as the discriminator. The study evaluates the models based on their accuracy in training, validation and testing phases. The results show us that the Deep LSTM model performed best on the training and validation accuracy and the GAN model performed best on the test accuracy showing its ability to generalise well to unseen data. The paper concluded that 'there are no significant differences with GAN and models traditionally used as LSTM'. They also mention the possibility of exploring other GAN architectures like WGAN and MHGAN, for potentially improved prediction accuracy.

Boris Banushev published a github repository where he uses Generative Adversarial Network to predict stock price movements for Goldman Sachs [1]. The architecture uses a LSTM as the generator and a CNN as the discriminator. The project uses a vast array of input data which include historical trading data and technical indicators, as well as sentiment analysis of the market, fourier transforms for extracting trend direction, stacked autoencoders for high-level features and Autoregressive integrated moving average for stock function approximation. GANs can be used to generate data for future values that will have similar distribution as the historical data that we already have. The use of the LSTM as the generator is down to the fact it is able to keep track of all previous data points and can capture the temporal patterns that develop over time. CNNs excel at feature extraction, which makes them well-suited for distinguishing real data from fake data. Boris experiments with an approach that integrates the Wasserstein GAN and the Metropolis Hastings GAN and uses Reinforcement learning to optimize the hyperparameters. After every 200 epochs, an error statistic is recorded and used as a reward value for the reinforcement learning algorithm that will then decide whether to change or keep the set of hyperparameters. If it decides it will update the hyperparameters it will call bayesian optimisation library that will give the next best expected the next optimal set of hyperparameters based on expected performance improvements. The final plot below 2.1 is run for 10 episodes of 200 epoch and we can see that it has the ability to capture the Goldman Sachs stock closing price quite well.

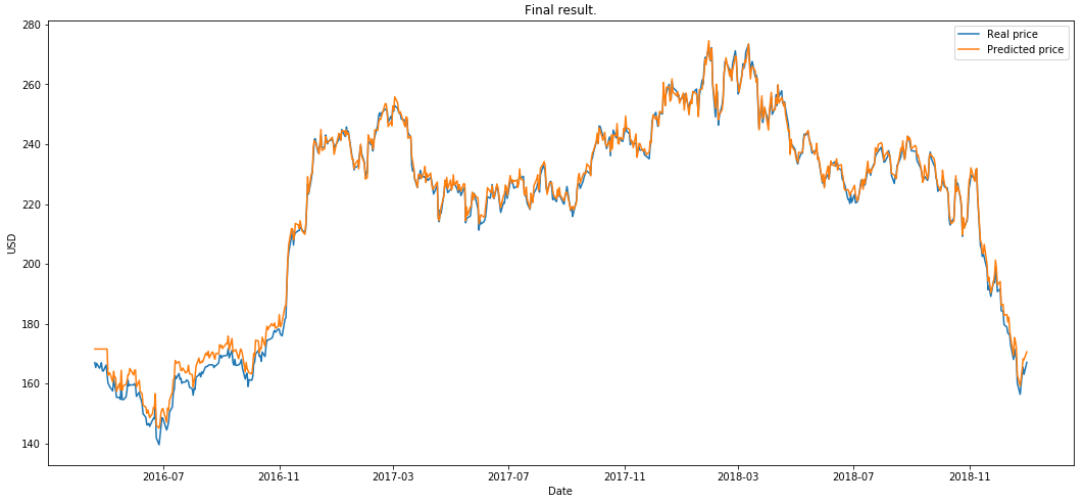
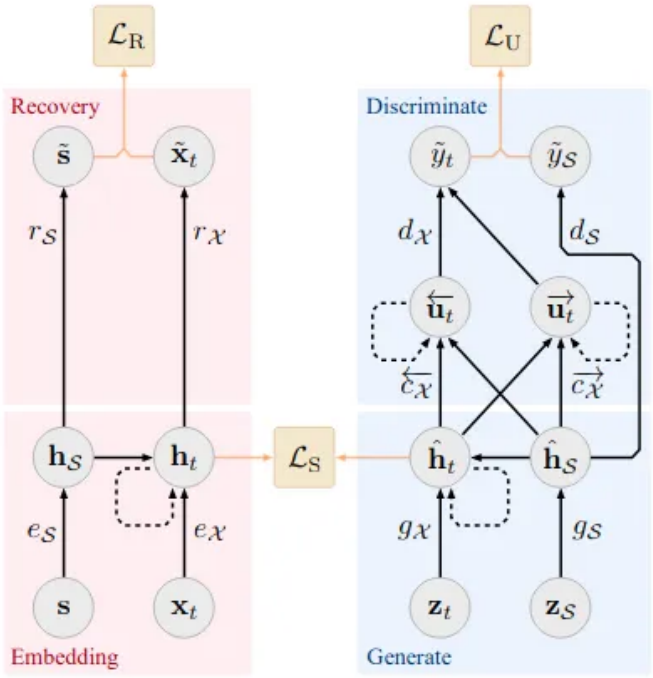


Figure 2.1: Predicted vs actual closing stock price [1]

A good generative model for time series data should preserve temporal dynamics of the data, in the sense that new sequences should respect the original structure of the data across time. Yoon et al proposed a new GAN architecture in their paper 'Time-series Generative Adversarial Networks' [2]. They introduced a novel framework for generating realistic time series data. Unlike other GAN architectures where an unsupervised adversarial loss is used, TimeGAN introduces a supervised loss which helps the model to capture time conditional distribution within the data by using the original data as a supervision. This dual approach enables the model to learn not just from the adversarial process but also from the actual temporal transitions observed in the real data. This helps in better capturing the conditional distribution $p(x_t|x_{1:t-1})$, which is crucial for preserving temporal dynamics.

The model introduces an embedding network that maps high-dimensional data to lower-dimensional latent space. This process facilitates the generator in learning and replicating the temporal dynamics. TimeGAN ensures co-optimization of the embedding and generation networks, promoting the generation of sequences that adhere to the temporal correlations observed in the training data.

We can see in the model below 2.2. The embedding block is where the original data are transformed into latent space representation. The recovery block attempts to map the latent space representation back to the original data space, attempting to reconstruct the original time series data from the embedding. The generator block is tasked with producing new time series data and the discriminate block differentiates between real and generated data. L_S is the supervised loss and guides the embedding and generation to preserve the temporal dynamics of the data.



(a) TimeGAN

Figure 2.2: Time GAN Framework [2]

In the paper, TimeGAN is tested on a dataset with varied characteristics such as sinusoidal sequence, stock prices, energy consumption and event timing data. The TimeGAN outperforms other GAN architectures like RCGAN, WaveGAN and others in Discriminator Score and Predictive score with the paper concluding that 'TimeGAN demonstrates consistent and significant improvements over state-of-the-art benchmarks in generating realistic time-series data' [2].

3 Theoretical Background

3.1 LSTM

Long Short Term Memory (LSTM) are in essence a type of Recurrent Neural Network(RNN) introduced by Hochreiter and Schmidhuber in 1997 to improve upon the RNN [19]. LSTM is well suited for sequence prediction and improves the RNNs ability at capturing long term dependencies which are crucial in Time series forecasting. In traditional neural networks, inputs and outputs are assumed to be independent of each other. A RNN is a type of artificial neural network where the output from the previous step is fed as input to the current step which is helpful for tasks like time series analysis.

The most important feature of RNN is its hidden state, which retains information from the previous inputs, allowing the network to maintain a form of 'memory' [20]. This memory enables the RNN to make use of historical data, effectively giving the network a sense of 'time' as it processes sequences. It uses the same parameters for each step of the sequence, reducing model complexity and ensuring consistent processing across inputs. For each element in the sequence, the RNN combines the new input with its current state to calculate the new state, which will be passed on to the next time step. The update of the hidden state allows RNNs to model temporal dependencies, as each new state is a reflection of past information combined with new input, enabling the network to make informed predictions or decisions based on accumulated knowledge. This feature is particularly valuable when working with time series data.

Training an RNN involves unfolding the network over time steps, creating a chain of replicated networks, each corresponding to a time step in the sequence. This unfolding transforms the RNN into a deep network for the purposes of training, where each layer represents a distinct time step in the sequence . During backpropagation through time (BPTT), gradients are calculated across this extended network structure. This process is crucial as it allows the RNN to learn dependencies across different time steps, making it possible for the network to understand and remember the sequence's context which is essential for tasks where the order and relationship of events over time significantly influence the outcome.

There are some disadvantages to the traditional RNN that the LSTM addresses. During backpropagation, RNNs suffer from the vanishing gradient problem. This is where gradients can shrink exponentially as the sequence length increases, leading to very small updates to the weights of the network [21]. This can make it difficult for the RNN to learn and maintain information from early inputs in the sequence, affecting its ability to capture long-term dependencies. They also by design have a short memory, which means it struggles to remember information from the distant past in a sequence.

The LSTM introduces gate mechanisms - the input, forget, and output gate- that regulate the flow of information in a more refined manner than traditional RNNs. These gates allow the network to decide which information to keep or discard at each time step, mitigating the vanishing gradient problem in RNNs by preventing the accumulation of irrelevant information that can dilute the gradient [19]. These gates work in tandem to maintain a constant flow of information through the cell, allowing LSTM to update and carry information across much longer sequences than traditional RNNs. The key feature of LSTMs is the cell state, which runs straight down the entire chain of the network, with only minor linear interactions. This design allows the network to transport information across many time steps effectively, enabling it to remember long term dependencies. The figure 3.1 shows the structure of the network.

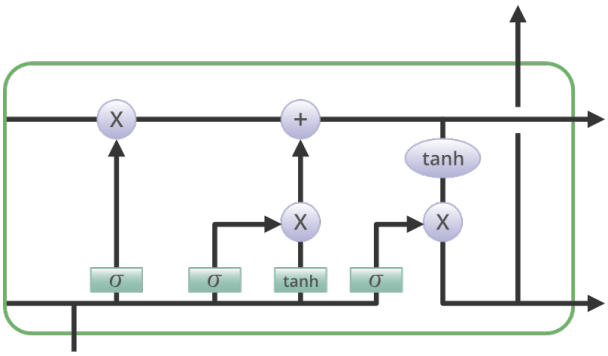


Figure 3.1: LSTM network [3]

The forget gate in LSTM allows the cell to selectively remember or forget information, which gives it a more dynamic and controllable memory mechanism. This selective memory is vital for tasks where not all past information is relevant to future predictions. This is significant improvement over traditional RNNs, which have a more rigid and less controllable memory structure. As seen in figure 3.2, two inputs X_t and h_{t-1} are fed to the gate and multiplied with weight matrices followed by the addition of bias. The result is passed through an activation function which gives a binary output. This is represented in the formula below where f_t represents the forget gate at time step t, σ is the sigmoid activation function, W_f is the weight matrix for the forget gate, h_{t-1} is the hidden state vector from the previous time step, x_t is the input vector at the current time step with the brackets [ ] representing a concatenation between h_{t-1} and x_t and finally b_f is the bias vector.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

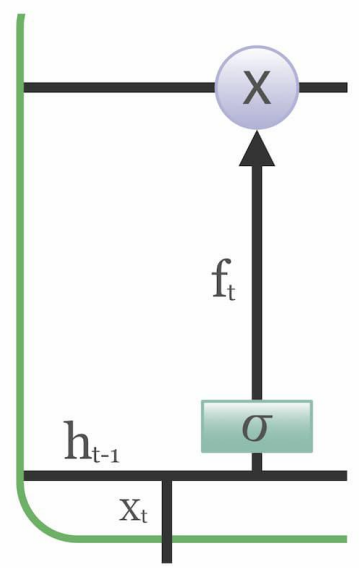


Figure 3.2: The forget gate mechanism in an LSTM cell. [3]

The input gate in an LSTM network is responsible for deciding what new relevant information will be added to the cell state. First, information is regulated using the sigmoid function, which is applied to a combination of the current input x_t and the previous hidden state h_{t-1} to decide what new information is important enough to be remembered. The output of the sigmoid function i_t, acts as a filter that allows certain parts of information to pass through. Concurrently as seen in figure 3.3, a vector $\tilde{C}_t$ is created using tanh function that gives an output from -1 to +1, which contains all the possible updates derived from the current and past information. At last, the values of the vector $\tilde{C}_t$ and the regulated values i_t are multiplied to obtain the useful information that is allowed to update the cell state.

$$i_t = (W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = (W_c \cdot [h_{t-1}, x_t] + b_c)$$

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

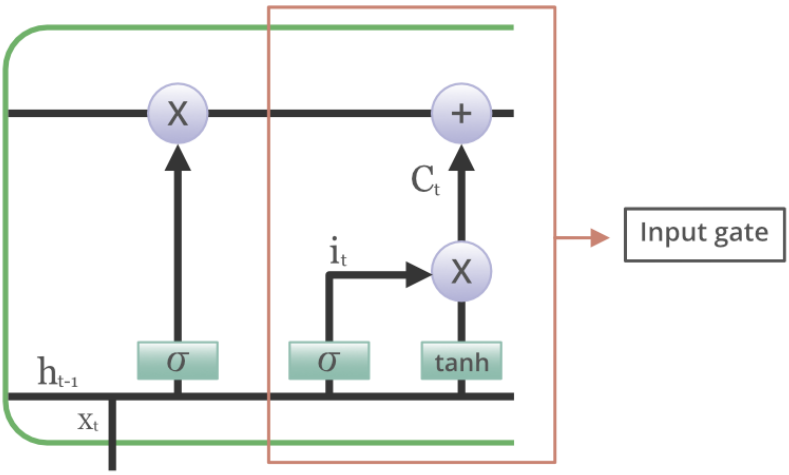


Figure 3.3: The input gate mechanism in an LSTM cell. [3]

The output gate has the tasks of extracting useful information from the current cell state to be presented as output. First, the cell state C_t is passed through a tanh function in order to normalize the values of the cell state to be between -1 and 1. Then, the output gate itself uses a sigmoid function applied to the inputs X_t and h_{t-1}, giving the output o_t, which decides which parts of the cell state should be outputted. Finally, the tanh-processed cell state and the sigmoid gate's output o_t are multiplied. This product is then forwarded to the next time step as the hidden state.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

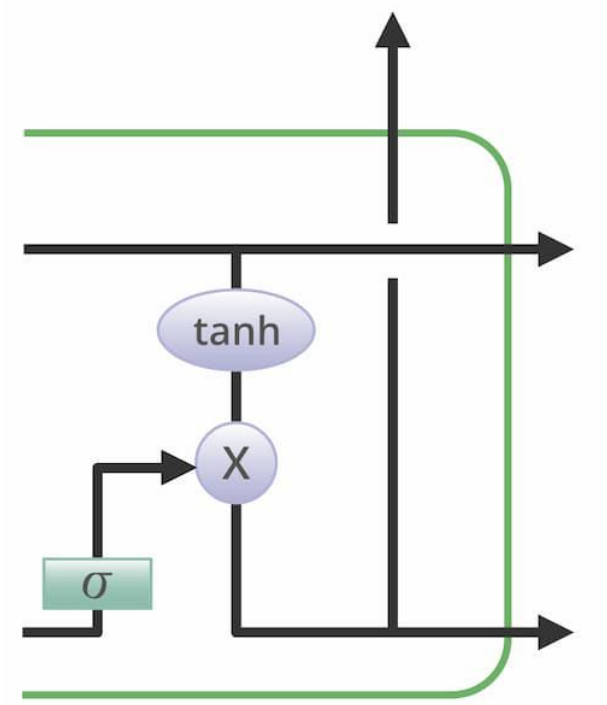


Figure 3.4: The output gate mechanism in an LSTM cell. [3]

3.2 CNN

Convolutional neural networks were first introduced by Yann Lecun in 1998 for recognition of handwritten digits [22]. The key innovation was the use of convolutional layers, pooling layers and fully connected layer for image classification. In 2012, AlexNet revolutionized deep learning by introducing an innovative architecture that leveraged the increased computing power available. This architecture featured deeper network layers with a larger number of filters, stacked convolutional layers, max pooling, dropout, data augmentation, and the ReLU activation function. These enhancements significantly improved performance, enabling AlexNet to win the ImageNet Challenge 2012 [23].

A convolutional neural network (CNN) is an advanced type of neural network that evolves from fully connected networks, specifically designed to process data in a grid-like structure. Unlike fully connected networks where every neuron is connected to every neuron in the next layer, CNNs introduce a specialized architecture that efficiently handles spatial data. A CNN typically has three layers, a convolutional layer, a pooling layer, and a fully connected layer. The convolutional layer applies filters to the input data to extract features, the pooling layer downsamples the data to reduce computation and the fully connected layer makes the final prediction.

The convolutional layer forms the cornerstone of the CNN. This layer utilizes a set of learnable parameters otherwise known as a kernel, which are small but extend through the full depth of the input volume [24]. By sliding these filters across the width and height of the input, the network learns to identify patterns and features. During the forward pass, the kernel slides across the height and width of the input, computing dot products between the filter and the input at each spatial position as shown in figure 3.5. This produces a two-dimensional activation map that gives the response of the kernel at each spatial position of the image. The sliding size of the kernel is called a stride. The output size of the convolutional layer is the feature map. If we have an input of size W x W x D [4] and D_{out} number of kernels with spatial size of F with stride S and amount of padding P, then the size of the feature map can be determined by

$$W_{out} = \frac{(W - F + 2P)}{S} + 1$$

Pooling layers follow convolutional layers and serve to progressively reduce the spatial size of the representation by deriving a summary statistic of the nearby outputs, which decreases the required amount of computation and weights [24]. These layers simplify the information in the feature map while retaining essential features detected by the convolutional layers. Max pooling, a common pooling technique, shrinks the feature map by dividing it into regions and retaining only the maximum value from each region as shown in figure 3.6. If we

have a feature map of size W x W x D [4], a pooling kernel of spatial size F and stride S, then the size of output volume can be determined by

$$W_{out} = \frac{W - F}{S} + 1$$

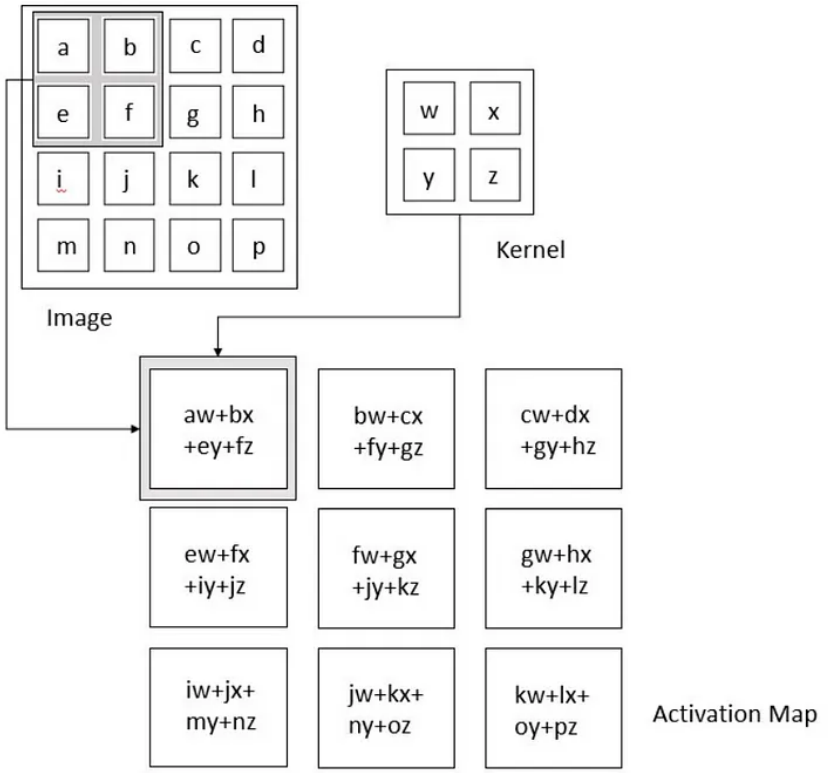


Figure 3.5: Convolutional operation [4]

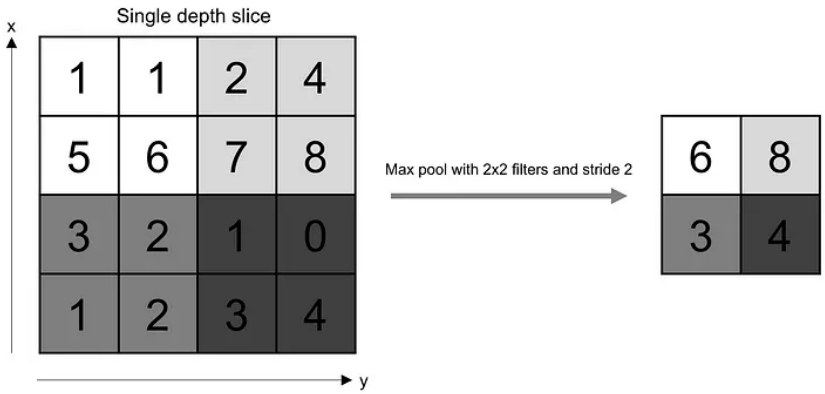


Figure 3.6: pooling operation [4]

The fully connected layer, typically positioned after convolutional and pooling layers, integrate the learned features into a form suitable for output. They have neurons with a

relationship to every neurons in the preceding and succeeding layer. The final layer often has the same number of neurons as the target class, producing the networks output prediction.

Since convolution is a linear operation and the data may be far from linear, non linearity layers are often placed directly after the convolutional layer to introduce non-linearity to the activation map. The rectified linear unit (ReLU) computes the function $f(k) = max(0, k)$. However, when neurons only receive negative input, the output of the ReLU function is zero. As a result, during the backpropagation process, there is no gradient flowing through the neuron, and its weights do not get updated. To mitigate this, Leaky ReLU introduces a small gradient for negative inputs, preserving some activity in the neurons [25].

3.3 Generative Adversarial Networks

Generative Adversarial Networks (GANs) are an approach to generative modeling introduced by Goodfellow et al in a 2014 paper titled 'Generative Adversarial Nets' [5]. This approach involves a process where two neural networks, a generator (G) and a discriminator (D), are trained simultaneously in a game-theoretic approach. These networks engage in a duel, where the generator aims to create data indistinguishable from genuine data, while the discriminator tries to accurately distinguish between real and generated data. Deep Generative models are a class of machine learning algorithms that are capable of generating new data samples that resemble the training data. Generative modeling is an unsupervised learning task that involves identifying and learning the patterns within the input data so that the model can produce new data points that could feasibly belong to the original data. GANs are a way of training a generative model by framing the problem as a supervised learning problem, where the generator and discriminator improve in tandem through their adversarial interaction, driving the system towards producing highly realistic data. The training continues until the discriminator model is fooled about half the time, meaning the generator model is generating convincing data.

At the heart of GAN architecture is the interplay between the generator and the discriminator, trained competitively to outsmart each other. The generator starts by sampling noise from a predefined distribution, commonly a Gaussian distribution. This noise, represented as vector z, serves as the input to the generator to produce synthetic data. The input vector is randomly drawn and seeds the generative process. After training, points in this vector space will correspond to points in the problem domain, forming a compressed representation of the data distribution. Latent variables within this space, which are influential yet not directly observable, encapsulate important features of the domain. The discriminator functions as a classifier that attempts to distinguish between real data drawn from the training set and fake data generated by the Generator. During the training process,

the discriminator guides the generator to produce more realistic data, while it simultaneously becomes better at detected fakes.

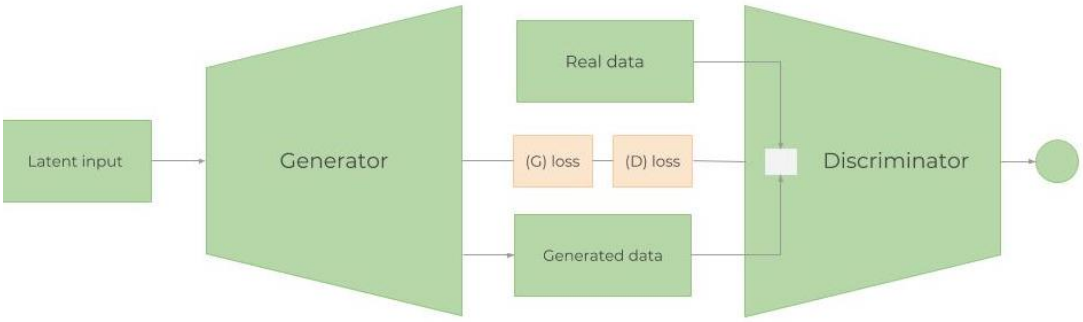


Figure 3.7: GAN architecture [1]

Training process of a GAN is a zero sum non cooperative game called minimax, where one player is trying to maximise their chances of winning while the other player is trying to minimise the chances of the first player winning, a concept central to the GAN framework as explained by Goodfellow et al [5]. In game theory, the GAN model converges when the discriminator and the generator reach a Nash equilibrium. The value Function is trying to capture the competition between the two players and measures how well they are performing. The optimal point in this function is our Nash equilbrium.

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$$

In GANs the generator tries to minimise this function by getting better at fooling the discriminator while the discriminator wants to maximise this function by improving its ability to distinguish between the real and fake data. D(x) represents the probability that the x came from the actual data rather than the generators distribution p_g. The discriminator D is trained to maximise the probability of assigning the correct label to both training samples and samples from g. Simultaneously, the generator G is trained to minimise $log(1 - D(G(z))$.

Optimizing this value function involves an iterative approach using minibatch stochastic gradient descent to update the weights of the discriminator and the generator in an alternating fashion. The training alternates between optimizing D for a few steps and then performing one step of G optimization. For a specified k steps, the discriminator is updated by sampling a minibatch of noise samples from the noise prior $p_g(z)$ and real data examples from the data generating distribution $p_{\text{data}}(x)$. The discriminator parameters are updated by ascending its stochastic gradient. This gradient is computed based on the discriminators ability to correctly classify real data as real or generated data as fake. The update aims to

maximise the sum of the log probability of correctly identifying real data as real and the log probability of correctly identifying fake data as fake.

After training the discriminator for k steps, the generator is updated by sampling a new minibatch of m noise samples from the noise prior $p_g(z)$. The gradient of the generator parameters is computed based on the generator ability to produce data that the discriminator classifies as real. The generator's goal is to maximise the probability that the discriminator makes a mistake by classifying fake data as real.

The minimax game played between the generator and discriminator has a global optimal for $p_g = p_{\text{data}}$. This optimal discriminator D^* is derived as a function of the probability distribution of p_{data} and p_g. When p_{data} equals p_g, the global minimum of the training criterion is achieved, meaning the generator replicates the data so that the discriminator is equally likely to guess real or fake, signifying the generator's success in producing realistic data. The figure 3.8 shows the process of the two networks converging.

$$D_G^*(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$$

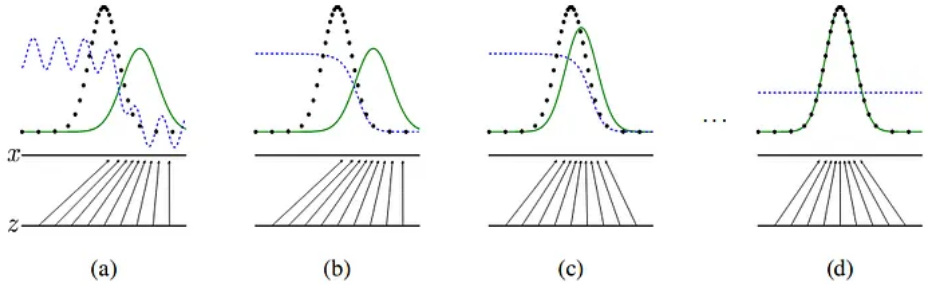


Figure 3.8: Convergence of GAN [5]

A challenge that arises in training GANs is the issue of non-convergence to the equilibrium. Non-convergence is a situation where the training does not stabilize, resulting in the generator and discriminator continually oscillating without reaching a point where the discriminator is fooled about half the time. Instead of reaching the equilibrium, the network enters a cycle of endless adaptation. Non-convergence can significantly impact the performance and utility of GANs. It can lead to the generation of low-quality data that does not accurately reflect the diversity or distribution of the training set. Several factors contribute to non-convergence, including the choice of loss functions, the architecture of the neural networks, and the training methodology. Imbalances in the training rates of the generator and discriminator can exacerbate this issue, leading to a scenario where neither network improves relative to the other over time.

One common issue is that early in the training, when G's output is poor, D can easily distinguish fake data, leading to a situation where term $log(1 - D(G(z))$ saturates. The gradient for the generator will also vanish which makes the gradient descent optimization very slow. Instead an alternative strategy is used to train G to maximise log D(G(z)) which provides stronger gradients for G, especially early in learning, facilitating better learning dynamics.

Another problem with GANs is mode collapse, which occurs when the generator finds a particular output that consistently fools the discriminator, it might focus on producing only that output or similar variations, leading to mode collapse. The generator should produce diverse output reflecting the data distribution. However, in the case where G is trained extensively without updates to D, it will converge to find the optimal data x^* that is most effective at decieving D, where $x^* = argmax_x D(x)$. This scenario lead to the mode collapse to a single point in the data distribution and the gradient with respect to input noise z approaches zero.

When training is restarting in the discriminator, the most effective way to detect fake data is to detect this single mode. Since the generator has reduced the influence of the input noise, the gradient from the discriminator will likely push the single point around for the next most vulnerable mode. Consequently, the generator produces such an imbalance of modes in training that it deteriorates its capability to generate diverse output. This results where both networks end up overfitting, each exploiting the short term weakness of the opponent and leading to a scenario where the model does not converge. This issue can make GANs less effective, as they fail to capture the full diversity of the training data, affecting the quality and variability of generated samples.

3.4 Wasserstein GAN

Wasserstein GANs (WGAN) is an approach to improving the stability and performance of traditional GANs, introduced by Arjovsky et al in 2017 [6]. An alternative loss function is used called the Wasserstein distance also known as the earth mover distance, that provides a more natural and meaningful metric for measuring the discrepancy between the real and fake data distributions. The introduction of the EM distance addresses the notorious training instability of GANs, including the mitigation of mode collapse, and facilitates a more reliable training process.

The Earth Mover distance is a measure of the distance between two probability distributions. It quantifies the minimum amount of work required to move from one distribution to another [26]. For the real data distribution P_r and the generated distribution P_g is defined

as the greatest lower bound for any transport plan.

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Gamma(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma} [\|x - y\|]$$

$\Gamma(P_r, P_g)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively P_r and P_g. $\gamma(x, y)$ indicates how much mass must be transported from x to y in order to transform the distributions P_r into the distribution P_g. The EM distance then is the cost of the optimal transport plan.

It can be shown how simple sequences of probability distributions converge using the Earth Mover's (EM) distance, but do not converge under other divergences such as the Kullback-Leibler (KL) and Jensen-Shannon (JS) divergences. Given a uniform distribution Z from 0 to 1 and a probability distribution P_0 represented by the points (0, Z), we can show the measures of another distribution P_θ that is parallel to P_0 and θ units away from it. Below we have the distance and divergence measure for EM, JS and KL.

$$W(P_0, P_\theta) = |\theta|$$

$$JS(P_0, P_\theta) = \begin{cases} \log 2 & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0, \end{cases}$$

$$KL(P_\theta \parallel P_0) = KL(P_0 \parallel P_\theta) = \begin{cases} +\infty & \text{if } \theta \neq 0, \\ 0 & \text{if } \theta = 0, \end{cases}$$

When θ approaches 0, the sequence converges to P_0 under EM distance, but does not converge at all under either the JS or KL divergences due to the non-overlapping nature of the distributions unless $\theta = 0$. The image 3.9 shows the smoother gradient compared to the GAN which fills areas with diminishing or exploring gradients.

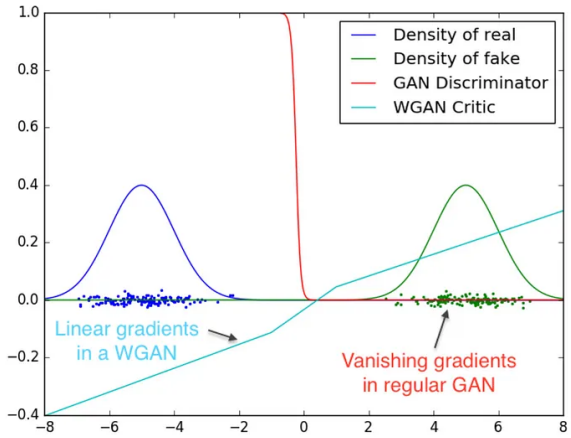


Figure 3.9: Optimal discriminator and critic when learning to differentiate two Gaussians. [6]

This helps solve the issues faced in a traditional GANs like non-convergence, diminishing gradient and mode collapse. We can see above how providing a continuous measure of distance that reflects the actual effort to align distributions, the EM distance facilitates convergence in scenaries where traditional divergences fail. Traditional measures might give uninformative gradients, which hinder the training process. In contrast, the EM distance provides a gradient that reflects the actual distance the generator needs to move, ensuring more consistent and effective learning updates, preventing the gradient diminishing issue. Also if the generator captures even a single mode of the target distribution closely enough, it might minimise the divergence without capturing other modes. The EM distance encourages the generator to spread its output to reduce the transportation cost across the entire distribution space, promoting diversity in the generated samples and combating mode collapse.

The Wasserstein distance can be very useful for the optimization of machine learning models like GANs. However, the infimum in the wasserstein distance equation is highly intractable [6], meaning that it is challenging to solve. This is because the infimum requires optimization over the space of all possible joint distributions, making the optimization process challenging. The Kantorovich-Rubinstein duality provides a useful way to compute the Wasserstein distance by optimizing over a family of 1-Lipschitz functions, f, which are functions that have gradients with absolute value at most 1 everywhere.

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} E_{x\sim\mathbb{P}_r}[f(x)] - E_{x\sim\mathbb{P}_\theta}[f(x)]$$

To solve this optimization problem we need to find a 1-lipschitz function that follows the constraint $|f(x_1) - f(x_2)| \leq |x_1 - x_2|$ and maximises $E_{x\sim P_r}[f(x)] = E_{x\sim P_g}[f(x)]$ [6]. In the original Wasserstein Gan paper there is a theorem that sounds us that there exists an optimal function f maximises the difference in expected values in the equation $E_{x\sim P_r}[f(x)] = E_{x\sim P_g}[f(x)]$ and moreover, it also shows a way to calculate the gradient of the wasserstein distance with respect to the parameter θ by backpropagation.

$$\nabla_\theta W(P_r, P_\theta) = -\mathbb{E}_{z\sim p(z)}[\nabla_\theta f(g_\theta(z))]$$

The Lipschitz constraint is important, as it ensures that the discriminator, referred to as the critic in their paper, does not give overly large gradients, which could destabilize the training process. To ensure that f is a 1-lipschitz function, WGAN applies weight clipping to restrict the maximum weight value in f. The weights of the discriminator must be within a certain range controlled by the hyperparameters c.

$$w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$$

$$w \leftarrow \text{clip}(w, -c, c)$$

For the training process, a learning rate, a clipping parameter, number of iterations of the critic per generator iteration and initial critic and generator parameters are defined. The training loop continues until the generator parameters have converged. Inside this loop, another loop starts which runs number of iterations of the critic per generator iteration to update the critic. For each iteration, a batch of real data is sampled from the real distribution and a batch of prior samples is taken from the noise distribution. The critic's gradient is computed based on the difference in the critics evaluation of real data and generated data.

$$\nabla_w \frac{1}{m} \sum_{i=1}^{m} \left[f(x^{(i)}) - f(G(z^{(i)})) \right]$$

The critic's weights are updated by moving in the direction of the gradient. This step minimises the Wasserstein distance. The weights are then clipped to ensure they stay within the range dictated by the clipping parameter which ensures the lipschitz continuity. After the critic is updated, a new batch of prior samples is sampled and the generators gradient is computed. The generator's parameters are updated in the direction that maximises the critics loss as we want the generator to fool the critic.

$$\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} f(G(z^{(i)}))$$

The biggest problem in this method is that weight clipping can be problematic way to enforce the lipschitz constraint, in particular when the hyperparameter c is not tuned correctly. If the clipping parameter is too high, it can slow down learning, making it harder to train the critc till optimal. If the clipping parameter is too low, it may lead to the vanishing gradients with deeper networks or when batch normalization is not used.

To fix this problem Gulrajani et al introduced the gradient penalty to the WGAN instead of the weight clipping to enforce the lipschitz constraint in their paper 'Improved Training of Wasserstein GANs' [27]. The WGAN-GP modifies the critic loss function by adding a gradient penalty term. This term penalizes the model if the gradient norm of the critic's output with respect to its input deviates from 1. For a function to be 1-Lipschitz, it should have gradients with norms at most 1 everywhere. Points interpolated between the real and generated data should have a gradient norm of 1 for f.

WGAN-GP penalizes the model if the gradient norm moves away from its target norm value 1. The loss function for the critic consists of the original critic loss plus the gradient penalty. The penalty term is the expected value of $(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2$ scaled by the lambda

parameter.

$$L = \mathbb{E}_{\tilde{x}\sim P_g}[D(\tilde{x})] - \mathbb{E}_{x\sim P_r}[D(x)] + \lambda \mathbb{E}_{\hat{x}\sim P_{\hat{x}}}\left[(\|\nabla_{\hat{x}} D(\hat{x})\|_2 - 1)^2\right]$$

The gradient penalty lead to more stable GAN training because it avoids the problems with weight clipping such as vanishing gradient. It helps the critic to better approximate the Wasserstein distance by ensuring that the function it learns is smooth.

In the WGAN-GP algorithm the critic and generator parameters are updated iteratively until the generator's parameters have converged. For a specified number of critic iterations, the critic's parameters are updated by sampling real data and generated fake data, then interpolating between real and fake data using a random number ϵ. The critics loss which includes the gradient term is then computed. The critics parameter is then updated using the Adam optimizer. After critic updates, the generator's parameter are updated by sampling from a new batch of input noise and computing the generators loss. The generator's parameters are updated using the Adam optimizer and this cycle is repeated until the generator parameter has converged.

3.5 Metropolis Hastings GAN

Metropolis Hastings GAN is an approach to GANs introduced by the Uber Engineering team in a paper titled 'Metropolis Hastings Generative Adversarial Networks' [7], in which combines aspects of Markov Chain Monte Carlo and GANs. When we normally train a GAN, we use the discriminator for the sole purpose of better training the generator. This approach deploys a sampling method where samples are drawn from the latent space of the generator using the Metropolis- Hastings algorithm with the discriminator choosing which generator samples to accept. It utilizes the discriminator after the initial training to create a wrapper around the generator for improved sampling. This selective process ensures that even with an imperfect generator, the resulting samples more accurately reflect the true data distribution. The discriminator effectively acts as quality controller on the generator's output.

The Metropolis Hastings algorithm is a Markov-Chain Monte-Carlo (MCMC) method for sampling from a distribution that is difficult to directly sample from. MCMC are a family of algorithms that uses Markov Chains to perform Monte Carlo estimates. Monte Carlo is a technique for sampling from a probability distribution and using those samples to approximate desired quantity. A Markov Chain is a process that exhibits Markov Property. This property makes an assumption that the probability of jumping from one state to the next state depends only on the current state and not on the sequence of previous states that lead to this current state. Given the state at a particular step along the chain, we can decide on the next state by seeing the probability distribution of states over the next step. A

Markov Chain reaches a stationary distribution if after enough steps in the chain, we have fixed transition probabilities between states. By starting with a random distribution and employing a Markov chain that is ergodic and where each sample is dependent only on the previous sample, allows for the exploration of the distribution space. These algorithms are designed to eventually converge to the target distribution, from which samples can be drawn [28].

Metropolis Hastings algorithm was first developed in 1953 by Metropolis et al [29] for the use in particle physics and then later in 1970 Hastings generalized the algorithm towards statistical applications [30]. The algorithm associated with a target density π, requires the choice of a conditional density q called the candidate [31]. The transition from the value of the Markov chain $X^{(t)}$ at time t and its value at time t + 1 proceeds via the following transition steps. Start with an initial point $X^{(t)} = x^{(t)}$ from the target distribution. At each step t, generate a candidate point Y_t from a proposal distribution $q(Y_t|X^{(t)})$. This proposal distribution can be any distribution from which you can sample. The choice of the proposal distribution q is crucial and can significantly affect the convergence of the algorithm. Compute the acceptance probability $\rho(x^{(t)}, Y_t)$ which is the probability of moving from the current point $x^{(t)}$ to the candidate point Y_t. The probability is calculated as

$$\rho(x^{(t)}, Y_t) = \min\left(\frac{\tilde{\pi}(x^{(t)}) \cdot q(Y_t|x^{(t)})}{\tilde{\pi}(Y_t) \cdot q(x^{(t)}|Y_t)}, 1\right)$$

$\tilde{\pi}(x)$ is the unnormalized target density at x. $q(y|x)$ is the probability of proposing y given x from the proposal distribution. The candidate is accepted with probability $\rho(x^{(t)}, Y_t)$ and $X^{(t+1)}$ is set to equal Y_t. The candidate is rejected with probability 1- $\rho(x^{(t)}, Y_t)$ and $X^{(t+1)}$ is set to equal $x^{(t)}$. These steps are repeated and after a large number of iterations the distribution of the values $X^{(t)}$ will approximate the target distribution π [31].

The integration of the Metropolis Hastings algorithm into the Generative Adversarial Network improves the generators ability to converge to the real data probability distribution. In a GAN, global optimum is reached when the generator distribution and the discriminator distribution equal the data distribution. $p_D = p_{data} = p_G$. If the discriminator is optimal for a imperfect generator, then $p_D = p_{data} \neq p_G$. Using MCMC, we can enhance the sampling process from p_D to iteratively refine the samples generated by G to more closely reflect the true data distribution p_{data}. Given a perfect discriminator D and an imperfect generator G, the exact samples from the true distribution p_{data} can be obtained by generating multiple samples from G and using D to accept or reject these samples based on their likelihood under the target distribution p_{data}. In figure 3.10 it shows how samples from the generator are input into the Metropolis Hastings Selector, which operates based on evaluations made by the discriminator.

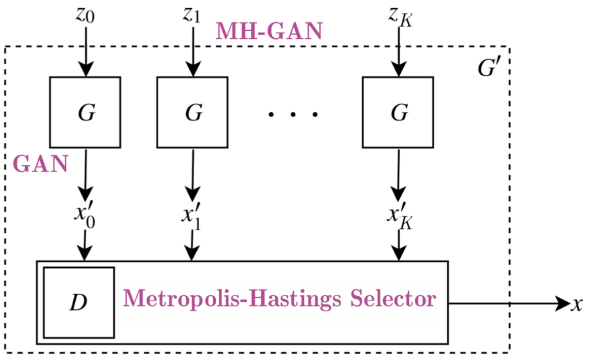


Figure 3.10: MHGAN is essentially a selector from multiple draws of G [7]

The goal of the MHGAN is to sample from the distribution p_D using the discriminator scores. This is achieved by setting the target distribution $p^* = p_D$ and the proposal distribution $q = p_G$. The ratio of $\frac{p_D}{p_G}$ is computed entirely from the discriminator scores.

$$\frac{p_D}{p_G} = \frac{1}{D^{-1} - 1}$$

The acceptance probability $\alpha(x_0, x_k)$ for transitioning from a current sample x_0 to a new sample x_k is determined using the discriminator scores. This probability is calculated as the minimum of 1 and the ratio of the probabilities of x_k and x_0, as derived from the discriminator scores.

$$\alpha(x_0, x_k) = \min\left(1, \frac{D(x_k)^{-1} - 1}{D(x_0)^{-1} - 1}\right).$$

If the discriminator D is perfect, then $p_D = p_{data}$, meaning that the samples obtained would ideally be from the data distribution p_{data}. This process ensures that even if the generator is not perfect, the sampling mechanism can still produce samples that closely represent the true data distribution. Calibration of the discriminator's probabilities ensures that the probabilities output by D accurately reflect the true likelihood of the samples being real or fake. Calibration can be tested by comparing the expected output with the probabilities provided by D. It can be accessed how well D's output aligns with actual outcomes. The calibration of D is crucial for the accuracy of the density ratios used in the Metropolis Hastings acceptance probability. In MCMC methods, the initial samples might not follow the target distribution, especially if the starting point is far from the equilibrium. The method circumvents the usual burn-in issues common in MCMC methods by initializing the Markov chain with a real data sample. This approach leverages the detailed balance property to ensure that the chain's marginal distribution at each step closely follows p_D.

The MHGAN algorithm takes as input a generator G, a calibrated discriminator D, and a set of real samples. It starts by selecting a random real sample x_0 and assigning it to x. This step is crucial for initializing the Markov Chain with a sample from the target distribution, aiming to mitigate the burn-in issue typically encountered in MCMC methods. For k iterations, a new sample x' is drawn from G and a number from the distribution

Uniform(0,1) is also drawn. The acceptance probability $\frac{D(x_k)^{-1}-1}{D(x_0)^{-1}-1}$ is calculated on the discriminator's evaluation of x and x'. If the number drawn from the uniform distribution is less than or equal to the acceptance probability, the algorithm updates x to be x'. This is where the Metropolis Hastings criterion decides whether to accept or reject the new sample generated by G. After K iterations, if x remains the initial real sample x_0, the algorithm restarts with a new sample from G. This helps avoid the situation where the chain gets stuck at the initial real sample. The final output if the sample x from the modified generator, which is a refined version of the original generator G.

4 Dataset

4.1 Introduction

When working with time series data, the old saying of garbage in, garbage out still holds particularly true. The effectiveness of a Generative Adversarial Network (GAN) in forecasting stock prices relies on the quality and preprocessing of the input data. This section delves into our dataset, focusing on our target stock prices and exploring additional features to augment our model's predictive power. Our data preprocessing pipeline begins with cleaning the data to ensure high quality inputs. After that, meaningful features were extracted that include technical indicators such as moving average and momentum. These indicators are pivotal for the model to grasp trends and patterns in the stock market. Furthermore, advanced methods like Fourier transforms to capture cyclical behaviors and Variational Autoencoders to generate new features. To standardize the input data, we apply normalization techniques, ensuring that all features contribute equally to the model's training process. This step is vital for preventing any single feature from disproportionately influencing the model due to scale differences. The data is also structured into sequential batches that represent historical windows of stock price, which is crucial for the temporal dynamics of stock data.

4.2 Dataset Description

The historical data for Google's stock was downloaded from Yahoo finance using yfinace which is a python library that allows to download historical market data. The data spans from 1st January, 2010, to 1st January, 2023. This dataset encapsulates daily trading information, including opening prices, the highest prices during the trading day, the lowest price, and closing price, along with trading volume, dividends issued and stock splits. This data enables us to dissect google's stock behaviour across varying market conditions, aiding in the development of out models.

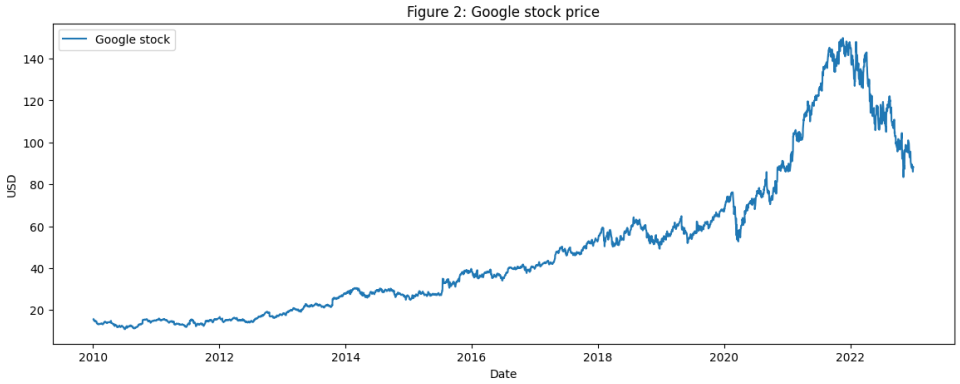


Figure 4.1: Google Closing stock price 2010-2022

4.3 Correlated Assets

Google is one of the biggest companies in the world and obviously doesn't operate in a isolated world. It depends on and interacts with external factors, including competitors and the wider global economy. Companies like Amazon, Apple and Microsoft are technological giants similar in size to google. Their stock movements can indicate trends within the tech sector and the broader market due to their significant market capitalizations. A change in movement for one of these stocks may provide valuable insights into sector-specific trends and can indicate a change in movement of price for google stock.

Market Indices provide a snapshot of market sentiment and overall economic health. The NASDQA is weighted towards technology stock, making it relevant for forecasting tech stocks like google. The S&P 500 and NYSE offer a broader perspective of the economy, offering insights into a wide range of sectors. The VIX is the volatility index and quantifies market risk and investors' sentiments by measuring the expected near-term volatility conveyed by S&P 500 stock index option prices. It is a key indicator of market sentiment and uncertainty.

The assets chosen all have over 0.94 correlation with Google closing stock price except for the volatility index suggesting that there is a not a linear relationship between the two. The VIX may influence the closing price in other non linear ways that cannot be measured in a traditional correlation test and give the model information about market conditions and sentiment in times of volatility. Incorporating a range of correlated assets provides the model with a holistic view of the financial markets, capturing both direct and indirect influence on the google stock price.

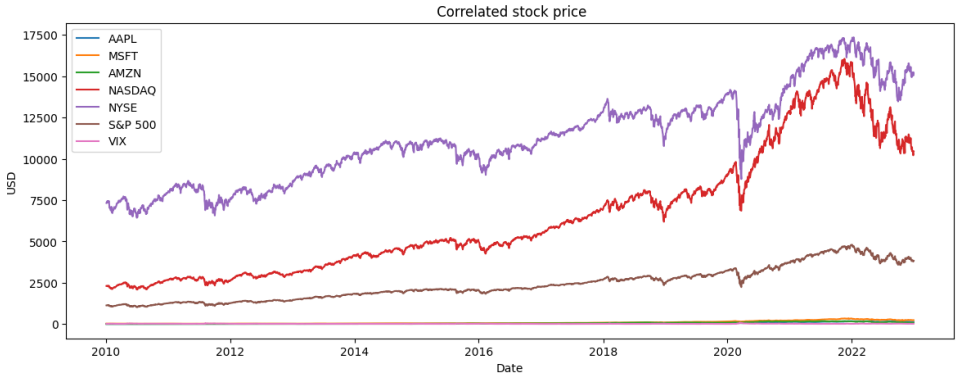


Figure 4.2: Correlated assets 2010-2022

4.4 Data Cleaning

Data cleaning plays a pivotal role in the preprocessing phase of a machine learning project. Incorrect or Irrelevant data can lead to misleading insights, rendering the forecasting model unreliable. Data cleaning ensures the model is trained on meaningful, accurate, and relevant information, paving the way for more dependable prediction [32].

In our dataset, the features 'Dividends' and 'Stock Splits' were removed due to their high counts of zeros suggesting these features are sparse for the given time period. Sparse features often contribute more noise than valuable information, potentially detracting from the model's predictive power [33].

Converting the Date column into Python date objects ensures consistency in date formatting. This consistency is vital for time series analysis, where the model's ability to recognize and interpret the sequence and intervals of data points can greatly influence its accuracy. The dataset index was also reset to facilitate easier manipulation of the data frame. The incorporation of technical indicators enriches the dataset with historical trend data. However, this addition introduces null values for initial data points where there isn't enough historical data to perform the calculations e.g the first 20 days for a 21-day moving average will be blank. The removal of these rows with null values ensures data point consistency, enhancing the analysis's reliability by ensuring every data point is complete.

4.5 Feature Engineering

Feature Engineering is a critical step in the development of machine learning models, involving the creation of new features or the transformation of existing features or the selection of relevant information from the raw data. This process aims to enhance model performance by improving data representation, making it easier to detect underlying patterns and relationships [34].

The process of feature engineering can involve several steps:

- **Feature Creation:** This step involves constructing new features from existing ones, applying a mix of mathematical operations, leveraging domain-specific knowledge, or integrating various features. A prime example is the utilization of technical indicators in financial modeling, designed to reveal market trends and help predict future movements.
- **Feature Transformation:** This is the process of transforming the features into a more suitable representation for the machine learning model. Transforming features can include scaling numeric data through normalization or encoding categorical variables.
- **Feature Extraction:** Often used in the context of dimensionality reduction, feature extraction involves creating a smaller set of new features that capture the most important information from the original set of features. Variational autoencoders can be used to restructure the data into a lower dimensional space, highlighting intrinsic structure and patterns that might not be obvious in the original high-dimensional space.

4.5.1 Technical Indicators

Technical indicators are essential tools in the analysis of financial markets since the 18^{th} century, providing insights into trends, momentum, volatility, and trading patterns [35]. Indicators such as moving averages help smooth out the data to reveal underlying trends, momentum helps measure the speed and change of price movements and bollinger bands help measure market volatility. These indicators help to make informed decisions by analyzing past and present price action to forecast future movements.

- **7-day and 21-day Moving Averages(ma7 and ma21):** Moving averages are foundational in identifying the direction of the trend. A moving average smooths out price data by creating a constantly updated average price. The 7-day moving average offers a glimpse into recent price movements, being more reactive to short-term changes. In contrast, the 21-day moving average provides a broader view, smoothing out price fluctuations over a longer period [36]. Crossovers of different moving averages can signal changes in trend, with the ma7 crossing above the ma21 indicating an uptrend, and a cross below suggesting a downtrend [37].
- **Exponential Moving Average (ema):** EMA is a type of moving average that places a greater emphasis on recent price data than the simple moving average. This makes the EMA more responsive to new information and quicker to react than the simple moving average. The EMA can be used to determine trend direction and

potential reversal points [38].

- **Moving Average Convergence Divergence(MACD):** The MACD is calculated by subtracting the 26-day Exponential Moving Average (EMA) from the 12-day EMA. The 12-day EMA is a faster, more sensitive average, while the 26-day EMA is slower and less sensitive. By comparing two EMA, it highlights how two moving averages converge or diverge over time. A positive MACD can indicate upward momentum while conversely, a negative MACD suggests downward momentum [39]. The MACD is used to spot changes in the strength, direction, momentum, and duration of a trend in a stock's price.
- **Bollinger Bands (upper-band and Lower-band):** Bollinger Bands measure market volatility and provide insights into overbought or oversold conditions. The bands widen during periods of increased volatility and tighten during less volatile times [40]. The upper band is calculated as a 20-day simple moving average plus twice the 20-day standard deviation of price closes. The lower band is the 20-day moving average minus twice the 20-day standard deviation.
- **Momentum:** The Momentum indicator measures the rate of change in stock prices. Momentum is used to identify the strength of a price movement. Positive Momentum indicate an upward trend, while negative values indicate a downward trend [41]. It can be used to identify overbought and oversold conditions, with the expectation that the price will revert to the mean. It is calculated by taking the current closing price, divides it by 100, and then subtracts 1, yielding a fraction that reflects the movement of the price in comparison to a benchmark price of 100

The figure 4.3 shows the technical indicators talked about above that are used to analyze the closing stock price trends for Google. The top chart displays the closing price for Google over the last 400 days along with the short-term and medium-term moving averages (MA 7 and MA 21), and the Bollinger Bands that indicate volatility. The MA7, shown by the green dashed line, closely follows the daily price movements, reflecting recent price trends and providing insight into short-term market sentiment. The MA21, shown by a red dashed line, smooths out the price over a longer period, offering a view of the intermediate trend. The bollinger bands, shown by the shaded area between the two turquoise blue blue lines, illustrate the level of volatility in the stock price.

The lower chart illustrates the MACD and Momentum indicators, which help identify the strength and direction of the stock's price movement. The MACD, shown by the dashed line, oscillates above and below zero, which indicates when the short-term momentum of the stock is diverging from its longer-term momentum. The momentum indicator, shown by the solid line, is constantly above zero line with a few dips below.

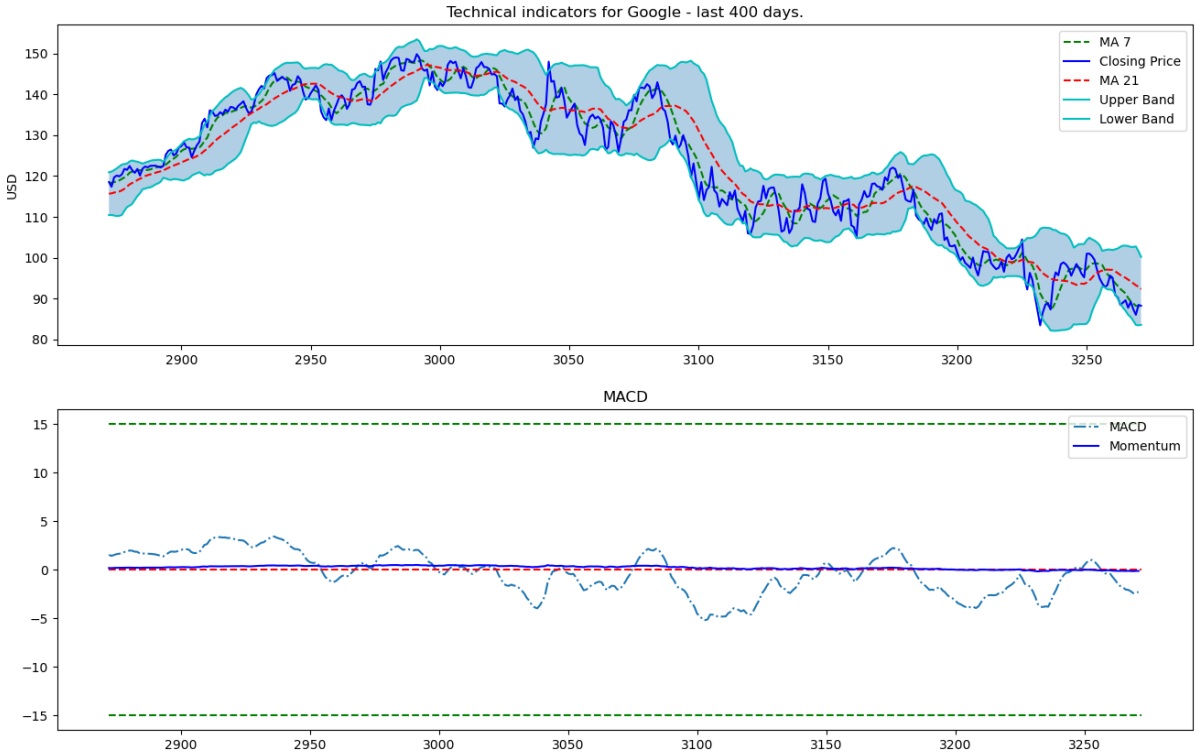


Figure 4.3: Technical Indicators for Google's Stock over the Last 400 Days

4.5.2 Fourier Transform

Time series is a sequence of data captured at regular periods of time and a common challenge is identifying underlying patterns as they are not easily discernible due to noise or irregular fluctuations. The Fourier Transform decomposes functions depending on time into functions depending on frequency. It is able to break down complex signals into their constituent components, providing insights into the cyclical behavior of a time series. This frequency based function is called the signals spectrum. The decomposition process involves representing the original time domain signal as a sum of sinusoidal functions of varying magnitudes, frequencies, and phase shifts. The Fourier Transform $F(\omega)$ of a time domain function $f(t)$ is given by

$$F(\omega) = \int_{-\infty}^{\infty} f(t) \cdot e^{-i\omega t}\, dt$$

here $e^{-i\omega t}$ represents a complex exponential function, which can be expressed as cosine and sine components using Euler's formula. The parameter ω represents angular frequency, and the exponential function oscillates at that frequency [42].

Our approach applies the Fast Fourier Transform (FFT), an algorithm used to compute Fourier transform more efficiently [43], to the closing prices of google stock data. It computes the frequency components of the time series, converting the data from the time

domain into the frequency domain. The transformation reveals the various cycles that contributes to the overall trend and fluctuations in the stock price. Once the FFT is applied, Magnitude and phase angle, which are key components for each frequency component, are calculated. The magnitude represents the contribution of each frequency to the signal's strength, while the phase angle indicates the starting point of each sinusoidal component within the cycle. These attributes provide understanding into the significance and timing of the underlying cycles within the stock price data.

To investigate the impact of these frequencies, we reconstruct the signal using a select number of components. This process involves zeroing out all but a few of the lowest frequency components on both ends of the spectrum, effectively acting as a low-pass filter. By retaining only 3, 6, 9 and 100 components, we can observe how the stock prices main trend and significant cycles are represented with varying levels of detail. The inverse FFT is then applied to these filtered frequency components, converting them back into time domain. The result is a series of smoothed signals that approximate the original time series.

The figure 4.4 displays the original closing prices alongside the reconstructed signals. It shows the progressive addition of detail as more frequency components are included. With 3 components, the signal captures the broadest trend, while including 100 components brings the reconstruction closer to the original data, retaining more of the short-term fluctuations.

The Fourier Transform can help isolate the most influential cycles, potentially improving the accuracy of the models. The reconstructed signals are integrated into the original dataset, enriching it with additional features that represent different frequency approximations of the stock price movement. This addition can help the model account for long term trends and key cyclical patterns.

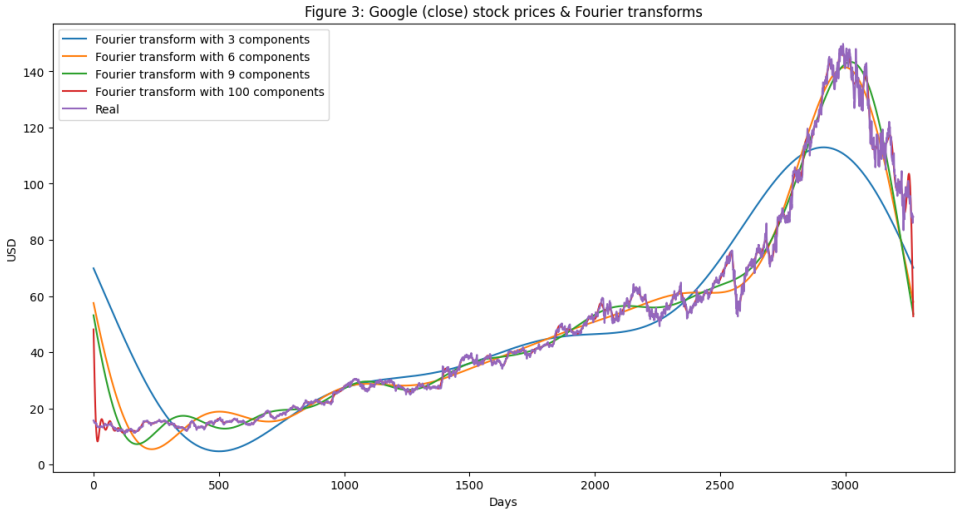


Figure 4.4: Fourier Transforms

4.5.3 Autoencoder

An Autoencoder is a type of neural network used for unsupervised learning to discover underlying correlations among data and represent data in a smaller dimension. An Autoencoder learns to compress the input data into a lower dimensional code and then reconstruct the output from this representation. An Autoencoder is made up of an Encoder, Bottleneck and Decoder. The encoder is the part of the network which takes in the input an produces a lower dimensional encoding. The Bottleneck represents the feature reducing layer that captures the compressed knowledge of the input data. The Decoder aims to reconstruct the input data from the latent space representation, aiming to recreate the input data as accurately as possible. To achieve this we aim to minimise the loss function which is the reconstructed loss. This is calculated by the error between the input and the reconstruction output [8].

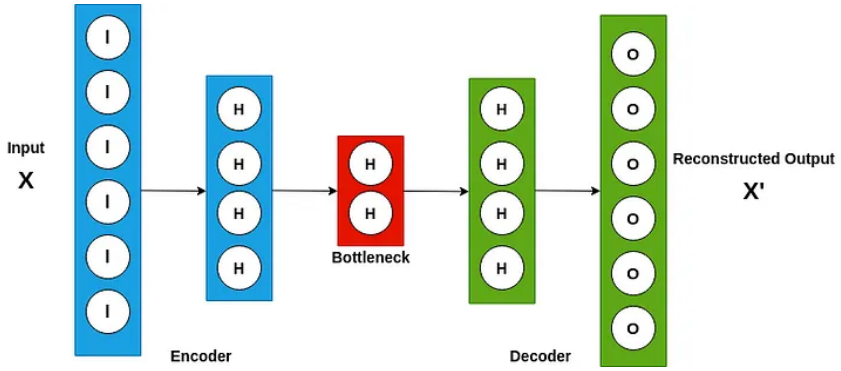


Figure 4.5: Autoencoder Architecture [8]

In this paper, we are going to use a variant of Autoencoders called Variational Autoencoder (VAE). Unlike standard Autoencoders, which aim to compress and reconstruct data as accurately as possible, VAEs focus on learning the probability distribution parameters, i.e. mean and variance, allowing them to generate new data points similar to the input. The encodings distribution is regularised during training in order to ensure that its latent space has good properties allowing us to generate some new data. The encoder takes input data and produces a probability distribution in the latent space and then the decoder samples from the probability distribution in the latent space to reconstruct the input data [44].

The loss function is this instance is made up of two terms the reconstruction loss and KL Divergence, which measures how one probability distribution diverges from another. In the context of a VAE, it ensures that the latent variables of the input are distributed in a way that closely approximates a desired distribution. Sampling from the distribution is a non-differentiable operation that prevents the direct application of backpropagation in the model. The reparameterization trick is used to help with this issue [45]. It works by adjusting the configuration of the sampling process, treating the random sampling as a noise term. In the case of a Gaussian distribution, the noise term is modeled as a standard normal

distribution. The noise term is then treated as independent from the other parameters in the model. This approach makes the sampling process differentiable, as the randomness is isolated to the noise allowing the gradients to flow through the mean and standard deviation during backpropagation.

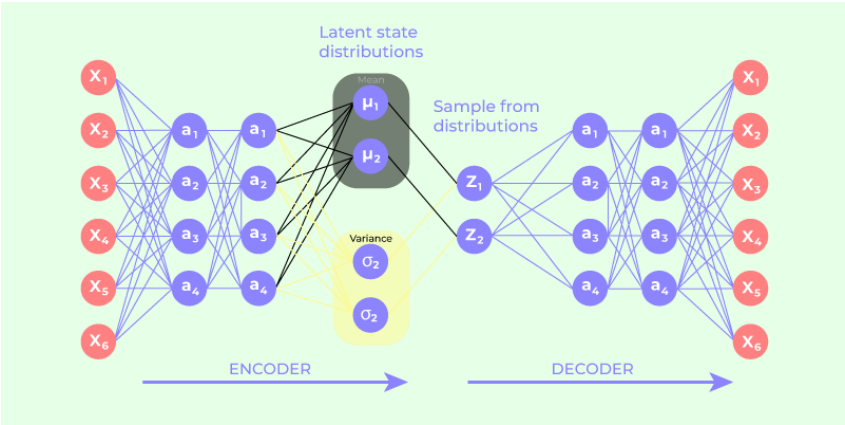


Figure 4.6: Variational Autoencoders [9]

In the implementation, a VAE is designed which defines the model architecture and the training procedure. A configuration list was used to define the size and the layers in the network and the dimensionality of the latent space. The encoder is constructed a series of linear layers using the dimensions of the configuration list with a Relu activation function. The distribution in the latent space is defined by two linear layers to output the mean and log variance. The decoder mirrors the encoder but in reverse, mapping the latent space back to the original input space. The final layer used a sigmoid activation to ensure output between zero and one, which aligns with the normalised data. The reparameterization trick is applied allowing gradients to flow. The output returns the latent representation, the mean and the log variance which gives us the parameters of the learned representation. For the training process, batches of the data are iterated over a number of epochs. For each batch, it performs a forward pass through the model, calculates the loss, performs backpropagation, and updates the model parameters. The loss is comprised of the KL divergence for latent space regularization and the binary cross-entropy is used for the reconstruction loss.

The dimensional reduction of the input data to the latent space serves as compressed feature set that captures the most important information from the original data. These features are learned during training to maximise the likelihood of the input data given the latent representation. This results in a set of features that are good at capturing the underlying factors of structure and variation in the data. Adding the latent space as a feature, allows the model to learn these underlying distribution of the data.

4.5.4 Normalization

Scaling and normalization are essential for preprocessing numerical data. Scaling refers to a broad set of methods aimed at adjusting the range of numerical values within a dataset. Its primary goal is to maintain the inherent distribution of the data, thus helping convergence in the model. By equalizing the influence of features with varying magnitudes, scaling ensures that no single feature disproportionately affects the model's outcome, which is vital for algorithms that depend on gradient-based optimization [46]. Normalization, a subset of scaling techniques, involves rescaling data to a range, usually between 0 and 1. This process guarantees that all features contribute equally to the model's analysis, without distorting the original distribution shape of the data. Different features can be measured in different units and scale and without normalization, features with higher magnitudes and wider ranges can dominate those with smaller scales during model training, leading to biased models that don't accurately represent the underlying data.

The MinMaxScaler is a specific tool for normalization that rescales the data to a specified range, typically between 0 and 1. The process involves fitting the scalar to the data, which calculates the minimum and maximum values for each feature and then transforming the data to scale it within the [0, 1] range. This scaling method operates by transforming each feature individually, subtracting the minimum value of the feature and then dividing by the range of the feature [47].

$$X_{\text{scaled}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}}$$

The MinMaxScaler from scikit-learn's preprocessing module is adeptly applied to normalize the data. Two MinMaxScaler objects are created, one for scaling the input features and another for scaling the target variable. The scalars are fitted to the time series data and scales the variables to a range between 0 and 1, based on the minimum and maximum values of each column. The scaled data is then integrated back into the original dataset, replacing the data not yet scaled. The target scalar is serialized and saved to a file using picker. This allows the scalar to be reloaded in order to inversely transform scaled predictions back to their original scale

4.6 Data Structure

Time series data is inherently sequential, with each data point potentially influenced by preceding ones. Unlike machine learning models with non sequential data, which assume independence between observations, time series data requires a method to incorporate temporal dependencies. In order to use supervised machine learning on time series data you

need to re-frame the data [48].

Transforming a time series dataset into a format suitable for supervised learning, involves restructuring the data by using previous time steps as input variables and use the next time step as the output variable. This method allows us to use the value at a given time step to predict the value at the next.

The sliding window approach utilizes previous time steps also known as lags to predict future values. In this framework, we designate the previous time steps as the input and the subsequent time step as the output. This pairing creates a direct link between past and future values, essential for forecasting. The figure 4.7 shows the window technique as it moves over different windows of time. It shows a window of size 4 from t-3 to t, meaning the model is going to map the information contained in the window to make prediction at point t+1 [10].

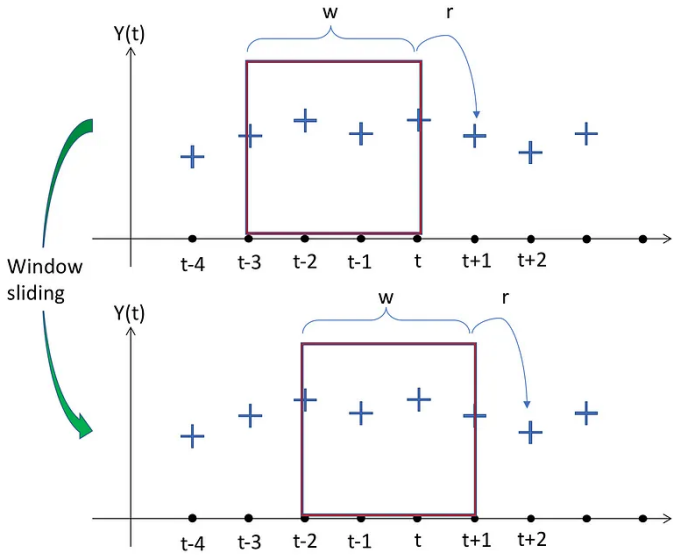


Figure 4.7: Sliding Window [10]

The term lag refers to the number of time steps back from the current observation that are used as inputs to predict the next step. It is one of the cornerstones of time series analysis. The size of the lag determines how many past observations are considered in predicting the future value, essentially setting the window size of our sliding window approach is setting the number of past observation we will use [48]. By employing the sliding window technique, we effectively convert the time series data into a sequence of input-output pairs, which preserves the order between the observations helping the model capture the temporal dependencies .

The sliding window technique reorganizes time series data into a structure that can be used in machine learning. It treats each pair as an independent instance, embedding temporal

relationships within each sequence. This allows models to learn from historical observations to predict future events.

The sliding window function used in this paper, iterates through the dataset, starting from the index equal to the window size. This ensures that there is enough previous observations to create a meaningful sequence for each data point. The window parameter determines the number of past observations used to predict the next value in the series. It does this by extracting a subset of data points starting from the current position i and extending back the length of the window parameter. This creates a sequence of past observations that serve as input features for the model. This implementation transforms the original time series data into a structured format for machine learning models. In this study the window size is set to 3 with a single response variable, creating a one step ahead forecast.

Splitting the data into training, testing and validation set is essential for a model's accuracy and ability to apply what it learns to new data. The Google close price serves as the target variable.

1. Training Set comprised of 70% of the dataset, trains the model, allowing it to learn the relationships between various features and the Close price.
2. Validation Set comprised of 15% of the data, is positioned between the training and testing sets. The validation set is pivotal for tuning the model's hyperparameters and preventing overfitting. It offers a feedback mechanism to adjust the model's complexity before final evaluation.
3. Testing Set is final 15% of the dataset and assesses the model's predictive power on data it has never encountered, providing insight into its real-world applicability and accuracy.

The distribution of the ensures that the model is trained extensively, fine-tuned appropriately, and evaluated accurately. Importantly, the split is executed chronologically to preserve the time series data temporal dependencies, ensuring that the model's predictive accuracy is tested on future data, reflecting a realistic scenario where past data is used to predict future outcomes and preventing data leakage.

5 Methodology

5.1 Introduction

In this section, we explore the methodologies behind LSTM and GAN architectures and training processes, as well as the GAN variants, WGAN and MHGAN. The LSTM model serves both as a baseline and as the generator component across all GAN architectures discussed. The GAN models leverage CNNs as discriminators to assess the generated data. Detailed discussions will cover the architecture of the models, including the number of layers and number of neurons on each layer, as well as the training processes used in the different models and hyperparameter tuning strategies employed to optimize the model efficiency.

5.2 Packages

The following packages are used to create and build the models.

- Pandas is imported for data manipulation and analysis, facilitating the handling of our dataset.
- NumPy provides comprehensive support for scientific computing in Python and is essential for processing numerical data.
- PyTorch is an open source machine learning library that supports the construction and training of neural networks.

  < torch.nn a submodule of PyTorch provides the building blocks for neural networks such as components like layers, activation functions and loss functions. >

  < Dataloader and TensorDataset are also packages from PyTorch that help in loading the data, making it easier to iterate over batches of data.>
- Matplotlib is a popular library for creating visualizations in python and is used to plot the loss functions and results.
- Scikit-learn is employed for its preprocessing capabilities

< preprocessing.MinMaxScaler is used for scaling and unscaling features >

< metrics.mean squared error used to evaluate the evaluate the performance of the model >

< isotonic.isotonicRegression provides a method for non linear regression technique and is used in the MHGAN training process. >

- math provides access to mathematical functions, supporting various operations that might be necessary for processing data

5.3 LSTM

In this research, the LSTM model serves as baseline against which we evaluate the performance of the GAN models. The LSTM acts as our generator, designed to generate forecasts for the data resembling the stock price movements in Google.

5.3.1 Architecture

A Vanilla LSTM is a model that has a single hidden layer but previous works have shown that a deep network provides much better results [18]. The increase in layers add levels of abstraction of input observations over time [49]. Each layers of LSTM can be thought of as learning a different level of abstraction of the data. With multiple layers, the network can potentially capture dependencies at various time scales. The lower layers might focus on short term dependencies, while higher layers might capture longer-term relationships. These enhanced learning capabilities make it very suitable at capturing the data distribution for Google stock price. The generator class uses nn.Module from PyTorch to implement custom layers for the LSTM model. Using nn.LSTM creates an instance of an LSTM layer where you can define the input size to the layer and the number of features in the hidden state. The first layer of the LSTM model has an input size defined by the number of features in the data and outputs 1024 features. The second layer takes 1024 features and reduces them to 512 features. The final LSTM reduces that to 256 features. This configuration acts as a funnel where the first layer captures a wide range of patterns from data. As the data moves through the network, the next layers with fewer neurons focus on more abstract, high level-representation facilitating the extraction of complex, high-level features from the input data.

The last LSTM layer produces an output that contains the learned representations of the input data. This output is passed through a series of linear layers that reduces the dimensionality of the features. These layers progressively downscale the from 256 features to 128, and then 64, and finally to a single value. This single value represents the generated

output at each time step. These linear layers allow to refine the extracted information extracted by the LSTM layers to a single prediction value.

In a deep network the risk of overfitting is much higher. A single model can be used to simulate having a large number of different networks by randomly dropping out nodes during training [50]. This is dropout and is a technique used for regularization of the model. The dropout layer is defined with a rate of 0.2 to reduce overfitting by randomly setting a fraction of the input units to 0 during training meaning that each neuron has a 20chance of being temporarily dropped in each training phase.

The forward pass processes the input data through the network. It defines and initializes hidden states and cell states for each layer of the LSTM network as zero tensors along with the feature dimension corresponding to the size of each layer's hidden state. Initializing these states to zero provides a neutral starting point, implying no prior knowledge before processing the sequence The data flows through the network with the input data fed into the model in the first layer along with the initialised hidden and cell states. A dropout layer is used a after each LSTM layer to prevent overfitting by randomly zeroing some of the elements of the output tensors with a probability of 0.2 during training. The output of the last LSTM layer is then passed through the first linear layer reducing its feature size to 128. This is then passed on through the next two linear layers where the feature size is reduced to 64 then finally to a single value which represents the generated data.

5.3.2 Training Process

The generator model is created with a specified input size that represents the number of features in the model. The mean squared error loss function is one of the most common loss function is machine learning [51] and is used to quantify the difference between the model's prediction and actual target values. It measures the average squared difference between the two values, guiding the model to improve its predictions iteratively. The ADAM optimizer is a popular algorithm for first-order gradient-based optimization of stochastic objective functions [52]. It is selected for adjusting the model weights, initialized with the specified learning rate. Adam is favoured for its adaptive learning rate properties, making it suitable for data with noise or sparse gradients.

A DataLoader organises the training data into batches controlled by the batch size without shuffling to preserve the sequential nature of the data, which is crucial for time series analysis. Within each epoch, the model undergoes training where it predicts the output given the input sequences. The loss is computed and updates the model parameters based on these gradients. Notably, gradients are reset to zero at the beginning of processing each batch to avoid incorrect accumulation from previous iterations. Loss values are tracked and accumulated throughout an epoch to monitor the model's performance and convergence.

The aggregated loss is presented at the end of each epoch, offering insights into the model's learning progress over time.

5.4 Generative Adversarial Networks

The Generative Adversarial networks designed here used the traditional GAN framework using LSTM as our Generator and CNN as our discriminator and train them in an adversarial approach.

5.4.1 Architecture

GAN extends the LSTM model using the same architecture for its Generator, utilizing the LSTM ability to capture the long term relationships in sequential data.

CNNs excel at feature extraction, making them effective discriminators capable of distinguishing between real and fake data. Similar to LSTM, by stacking multiple convolutional and pooling layers, CNNs can learn increasingly complex features, leading to high performance as was shown with the breakthrough of AlexNet [23].

The discriminator is comprised of three 1D convolutional layers. These layers are designed to process sequential data, making them suitable for time series input. The first convolutional layer has 4 input channels and 32 output channels, with a kernel size of 3, a stride of 1 and padding set to same ensuring the output has the same length as the input. The second layer increases the channel depth from 32 to 64, using the same kernel size, stride, and padding. The third layer increases the channel depth to 128. As the channel depth increases, the network gains the ability to detect a larger variety of features. This enhances the networks capacity to discriminate between the generated and real data.

Each convolutional layer is followed by a LeakyReLU activation function with a negative slope of 0.01. This function introduces non-linearity, facilitating the network to learn complex patterns and maintain gradient flow even when the unit is not active [25]. Following the linear layers, ReLU and sigmoid activations are utilized, where the sigmoid is used at the output to squash the final value into a range of 0 to 1, denoting the probability that the input is real.

After the convolutional processing, the data is flattened before being passed through a series of linear layers. The first linear layer transforms the flattened data to a size of 220, followed by batch normalization to stabilize learning. 'Improved Techniques for Training GANs' [53] talks about the benefits of using batch normalization for optimization in a GAN. The second layer maintains this same structure while the final layer reduces the dimension to one, producing a single scalar value, representing the discriminators score.

In the forward pass the input is sequentially passed through the convolutional layers with LeakyReLU activation. The output is flattened and processed through the linear layers with LeakyReLU and ReLU activation. The final scalar value, obtained after passing through a sigmoid activation function, represents the discriminator's confidence that the input is from the real dataset.

5.4.2 Training process

The use of the binary cross entropy for the loss function closely resembles the value function $\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log(1 - D(G(z)))]$ used in Goodfellow et al original GAN paper [5]. The GAN utilizes the binary cross entropy Loss from PyTorch's nn module. This loss function is suitable for the binary classification task the discriminator performs. It measures the difference between the predicted probabilities and the target true probabilities, optimizing the discriminator to accurately classify real and fake data.

As the Generator and Discriminator are working in adversarial training with the generator trying to minimise the loss function and discriminator trying to maximise it, two separate ADAM optimizer are initialized for the generator and the discriminator. By setting distinct learning rates and beta parameters, the training process allows for control over each parameters updates.

Training occurs across specified epochs, where each processes data in batches from the DataLoader. The discriminator is first trained on real data with associated labels set to 1, representing the real class. It then computes the loss based on its predictions. Subsequently, the discriminator is trained on fake data generated by the generator. This fake data is labeled as 0, denoting the fake class, to compute the loss. The total loss for the discriminator is the sum of losses from real and fake data, guiding the discriminator to improve its classification accuracy.

The generator's objective is to produce data that the discriminator will classify as real. Hence, during generator training, the fake data is treated as real to compute the generator's loss. This loss assesses the generator's success in fooling the discriminator, promoting the generator's ability to produce increasingly convincing fake data.

After computing the losses, backpropagation is performed to calculate the gradients for both models. The optimizer for the discriminator updates its weights to reduce the classification loss, enhancing its ability to distinguish between real and fake data.

After each epoch, the average losses for the generator and discriminator are calculated and displayed, offering insights into the GAN's learning ability and the competition between the generator and discriminator.

5.5 Wasserstein GAN

WGAN is a variant of GAN, using a novel loss function to train the model in the Wasserstein distance. This new loss function changes the architecture and training process of the model compared to GAN.

5.5.1 Architecture

The architecture for the Generator remains the same as the original GAN utilizing the LSTM model but there are some changes to the Discriminator architecture.

The architecture of the discriminator in the context of a Wasserstein GAN (WGAN) retains the fundamental structure of a typical GAN discriminator but incorporates specific modifications to align with the Wasserstein objective. These adjustments are essential to harness the benefits of the WGAN framework, which aims to provide a more stable training process and a more meaningful measure of distance between the distribution of real data and the generated data.

Gulrajani et al discuss the absence of critic batch normalization in the Improved Training of Wasserstein GANs [27]. The WGAN discriminator avoids using batch normalization because the training process involves a gradient penalty, which requires penalizing the critic's gradient norm for each input independently to adhere to the Lipschitz constraint. The batch-wise correlation introduced by batch normalization undermines this penalization mechanism, causing the gradient for a single input to be inadvertently influenced by other inputs within the same batch.

As discussed in the original Wasserstein GAN paper [6], WGAN discriminator does not apply a sigmoid activation function at the output, unlike traditional GANs. In WGANs, the discriminator does not classify inputs as real or fake but instead provides a score that represents the realness or fakeness of the input. This scoring is unbounded, which is a significant departure from the binary output in standard GANs. The unbounded nature of the discriminator's output facilitates the computation of the wasserstein distance, enhancing the stability and reliability of the training process.

5.5.2 Training process

Unlike GANs that use Binary Cross-Entropy, WGAN-GP employs Wasserstein loss, calculated by subtracting the discriminators average score on fake data from its average score on real data gives us the same $W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} E_{x\sim\mathbb{P}_r}[f(x)] - E_{x\sim\mathbb{P}_\theta}[f(x)]$ as in the original GAN paper [6]. This loss provides a more meaningful and stable training process, as it correlates directly with the generator's quality. WGAN-GP also incorporates a gradient

penalty to enforce the Lipschitz constraint and ensure stable training.

The WGAN also implements a gradient penalty in the training process as introduced in 'Improved Training of Wasserstein GANs [27]. This penalty is used for enforcing the Lipschitz constraint, ensuring stable training and addressing issues like mode collapse. The gradient penalty function blends between real and fake samples and computes the gradient of the discriminator output with respect to these blended samples. The penalty is the mean squared difference between the norms of these gradients and 1.

The discriminator is updated more frequently than the generator as shown in the WGAN algorithm [6]. This is controlled by the critic iterations parameter, ensuring that the discriminator accurately evaluates the samples before each generator update. During each epoch, the discriminator loss incorporates the gradient penalty, reflecting both its ability to differentiate real from fake and adhere to the Lipschitz constraint. The generator's loss is computed at a controlled frequency, aligning with the critic iteration schedule. This loss is negative of the discriminator's mean output for the fake data, encouraging the generator to produce samples that are deemed real by the discriminator.

For the discriminator, the gradient of the loss is computed and used to update the model's parameters. The optimizer's state is reset at the beginning of each iteration to prevent accumulation of gradients from previous steps. The output of loss is score representing the realness of the input data, not confined to the 0,1 range like the GAN.

5.6 Metropolis Hastings GAN

MHGAN builds on the foundation of GAN by using the same architecture and integrates the Metropolis Hastings algorithm. It uses the discriminator to sample from the generator after the initial GAN training in order to enhance the sample diversity and training stability.

5.6.1 Training process

The MHGAN improves upon the training of the GAN by integrating a Metropolis Hastings sampling approach introduced in 'metropolis-hastings generative adversarial networks' [7]. A MHGAN class is initialized and inherits a standard GAN framework, setting up the generator and discriminator, loss function and optimizer. Just like the training in the GAN, the discriminator employs a binary cross entropy loss to gauge the accuracy in classification, and the generator employs it to optimize for deceiving the discriminator.

The MHGANs unique feature is the integration of the Metropolis Hastings algorithm, giving us a sampling method that enhances the generator's output based on the discriminator feedback. Initial training follows the GAN methodology, where the discriminator assesses

both real and generated samples, and the generator updates based on the discriminator's evaluations. Once that training is finished, the discriminator, which is normally thrown away after the initial training, is used to refine the generated samples, aiming for a closer approximation to the real data distribution. Essentially, an initial phase to establish a baseline performance followed by a fine tuning phase employing the Metropolis Hastings sampling.

A Generator samples function is used to sample from the generator model. The score sample function evaluates the discriminator's confidence in distinguishing real and fake data. It computes the discriminator's output for the provided samples, and returns the score without calculating gradients. The calibrate discriminator function serves to calibrate the discriminator's outputs to align more closely with the true labels of the training data. It generates samples from the generator, concatenates these samples with the real data and uses the discriminator to score these concatenated samples. It fits an Isotonic Regression model to align the discriminator's scores with the actual labels. Isotonic Regression is a non-parametric calibration technique that assumes the underlying relationship is non-decreasing, thereby preserving the order of the scores while adjusting them. It is particularly effective for 'Transforming classifier scores into accurate multiclass probability estimates' [54], which is our case refines the discriminator's output.

The mh sample function implements the Metropolis Hastings sampling. The function generates and evaluates data samples, utilizing the calibrated discriminator as a metric for quality control. At each iteration, a new sample is generated and compared with an existing baselines using the discriminator's calibrated scores. It uses the Metropolis Hastings criteria as seen in the MHGAN paper $\alpha(x_0, x_k) = \min\left(1, \frac{D(x_k)^{-1}-1}{D(x_0)^{-1}-1}\right)$ [7] to determine whether a new sample should replace the baseline. It collects the samples that progressively align more closely with the true data distribution.

The training model function defines the training process for the MHGAN model. It iterates through the epoch training both the generator and discriminator. If the check type return true, it indicates a special training mode that involves calibrating the discriminator and generating samples based on Metropolis Hastings method. It also tracks the losses throughout the training process, updating their weights accordingly. The mh training function orchestrates the overall MHGAN training process. First the GAN undergoes standard training, then the Metropolis Hastings enhanced training is conducted.

5.7 Hyperparameter Tuning

Batch size determines the number of samples that will be propagated through the network at one time before updating the internal model parameters. An increase in batch size lowers

performance of the model but a larger batch size lowers the cost of training as fewer updates to the network means less training time [55]. It is a trade-off between efficiency and memory usage.

Learning rate defines the adjustment in the weights of our network with respect to the loss gradient descent. It controls the step size at each iteration while moving toward a minimum of the loss function. It determines how fast or slow we will move toward the optimal weights. If the learning rate is very large we will skip the optimal solutions and if it is too small we will need too many iterations to converge to the best values [56].

An epoch is one complete pass through the entire dataset, essentially one epoch is complete when the model has processed all the batches and updated its parameter based on the calculated loss. Number of epoch specifies how many times the entire dataset will be passed forward and backward through the neural network. When the number of epochs is too large, the training model learns patterns that are specific to sample data, making it incapable to generalize to new data by overfitting the training data. To mitigate overfitting, the model should be trained for an optimal number of epochs [57].

Lambda gradient penalty is the penalty coefficient used to weight the gradient penalty term in the Wasserstein GAN. This idea of gradient penalty in WGAN-GP is to enforce a constraint such that the gradients of the critics output with respect to the inputs to have unit norm and lambda decides how much of this penalty to apply [58].

Metropolis Hastings epochs decides how long the Metropolis Hastings algorithm will sample from the generator and decide whether to accept or reject the sample based on the acceptance probability.

An optimization procedure involves defining a search space , which can be visualized as a multidimensional landscape where each axis corresponds to a different hyperparameter of the model. Each point within the space represents a unique combination of hyperparameter values, essentially defining a singular model configuration. The goal of the optimization procedure is to find a vector that results in the best performance of the model after learning [59]. Random search defines a search space as a bounded domain where hyperparameter values can vary, then randomly selecting points to sample different configurations. Grid search defines a search space as a grid of hyperparameters values with each node representing a different combination of hyperparameter values. It then evaluates every position in the grid for a comprehensive examination of the grid.

Bergstra et al show in 'Random search for hyper-parameter optimization', that randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid [60]. While grid search is exhaustive as it searches over all the search space, Random search over the same domain is able to find models that are as good or better within a small fraction of

the computation time.

For the hyperparameter optimization of our models we define a search space for each of the hyperparameters. For each trial, a new instance of our models is initialized with the randomly chosen hyperparameters. This model undergoes training on our dataset, following which it is evaluated in terms of its predictive accuracy on a separate validation dataset. The mean squared error is calculated between the true values and predicted values and serves as the performance metric for the model configuration. It then selects the model configuration with the lowest MSE and outputs the parameters of this model. This optimal configuration is reported, providing insight into the most effective hyperparameter values for our model within the explored search space.

6 Results

6.1 Training Loss

This section looks at the loss functions for the four models after optimizing the models hyperparmeters. Here are the models training loss.

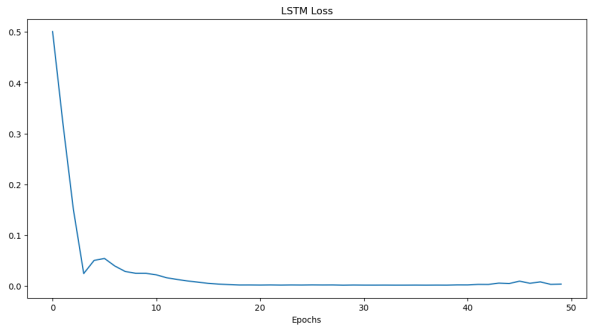


Figure 6.1: LSTM training loss

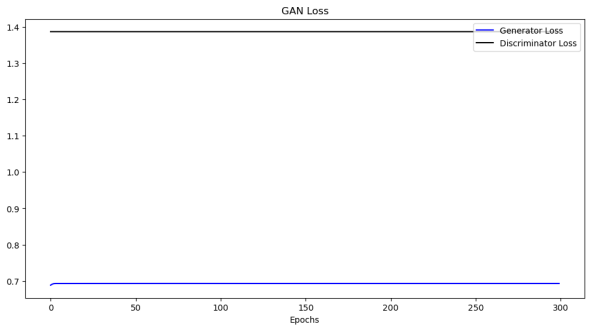


Figure 6.2: GAN training loss

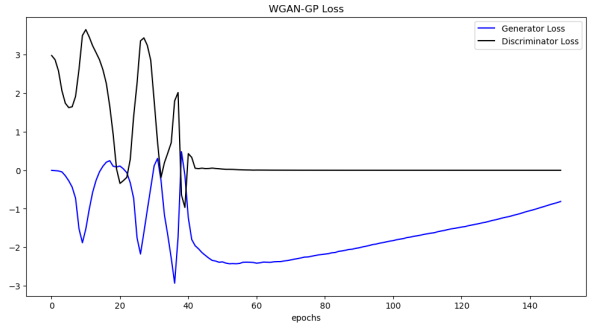


Figure 6.3: WGAN training loss

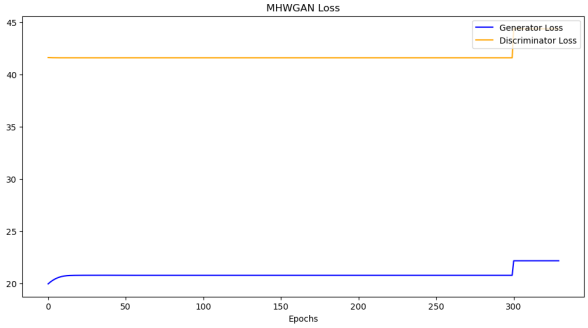


Figure 6.4: MHGAN training loss

- Figure 6.1: LSTM Training Loss shows a sharp increase and then decrease at the start of training and then the loss plateaus with some small movements up and down. This indicates learning progress through the initial number of epochs and then stabilization after that.
- Figure 6.2: GAN Training Loss shows the graph with the two unusual straight loss function for both the discriminator in black and generator in blue. This indicated that the GAN is not learning as expected. This would suggest that both the Generator and

Discriminators performance does not significantly improve or worsen as the training progresses. In GAN training training we would expect the losses to oscillate between each other.

- Figure 6.3: WGAN Training Loss displays oscillating lines representing the discriminator and generator losses. The discriminator's loss fluctuates significantly, suggesting it is repeatedly being challenged and improving in response to the generator. The generator loss decreases over time, with some fluctuation up and down, which hints at incremental improvements in generating realistic data. The two losses begin to converge at the end which may indicate that an equilibrium has been reached.
- Figure 6.4: Paralleling the GAN, the MHGAN shows a relative stability in training loss. However, during the Metropolis Hastings sampling phase the loss jumps for both the discriminator and the generator, which may indicate the model's attempt to refine its generative output.

6.2 Training Results

Here are the training Results after making a 1 day ahead prediction on the training data reveals the models ability to capture the market dynamics.

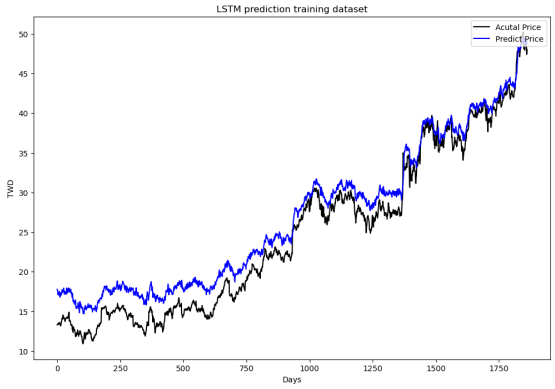


Figure 6.5: LSTM training result

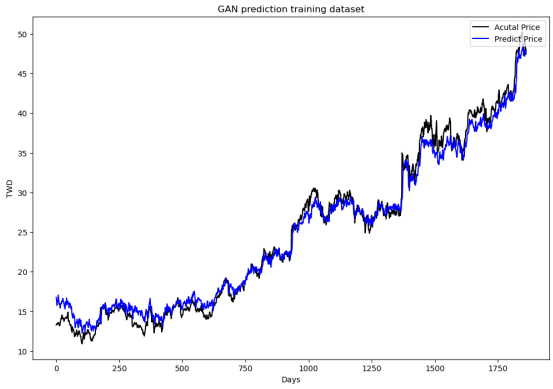


Figure 6.6: GAN training result

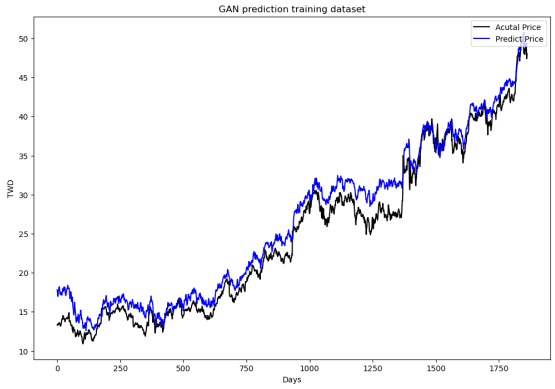


Figure 6.7: WGAN training result

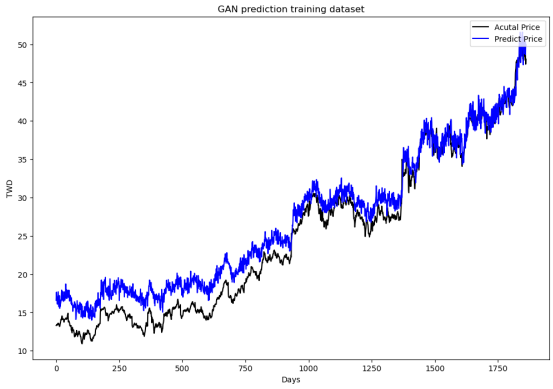


Figure 6.8: MHGAN training result

Figures 6.5; 6.6; 6.7; 6.8; compares the actual training data and the predicted closing prices for each of the four models on the training data, indicating the performance of the model on the data it has been trained on. Each model's predictions closely resemble the actual closing prices demonstrating their capability in learning the general trend of the price movement. Table 6.1 shows the errors scores for these models on the training data using Root Mean Square Error (RMSE), Mean Absolute Error (MAE) and Mean Absolute Percentage Error (MAPE). The table of error results also backs up the statement that the predictions closely resemble the actual closing price ranging from 1.24 to 1.75 RMSE.

Table 6.1: Comparison of the different models on training data

Models	RMSE	MAE	MAPE
LSTM	1.72	1.329	0.054
GAN	1.24	0.99	0.0467
WGAN	1.625	1.346	0.066
MHGAN	1.751	1.483	0.0818

6.3 Test Results

Here are the test Results after making a 1 day ahead prediction on the test data showing the actual data (black) versus the predicted values (blue) along with a table 6.2 of the overall test errors.

- Figure 6.9: shows the LSTM model's performance on test data. The predicted line closely follows the movement of the actual data but struggles with market fluctuations, particularly where the actual data shows sharp rises or drops.
- Figure 6.10: the predicted prices for the GAN model show a rough correlation with the actual prices but an overall RMSE score of 4.975 highlights the models limited capabilities to generalize well on the test data. It seems to also struggle with capturing the fluctuations of the data which may be indicated by the overfitting on the training data and inability to generalise to new data.
- Figure 6.11: demonstrates that the WGAN model more accurately follows the overall market trend and outperforms both the GAN and LSTM models in capturing market volatility. The training stability brought in by the new loss function and the use of a gradient penalty is evident as it is better able to capture market dynamics compared to the GAN and LSTM achieving overall lower errors scores at 2.412 RMSE.
- Figure 6.12: predictions for teh MHGAN model seem to effectively track the stock price movements, showing slight improvements upon the GAN. The reduction in error

from the GAN model of 4.975 to 3.213 shows the potential of the Metropolis Hastings sampling in producing better results.

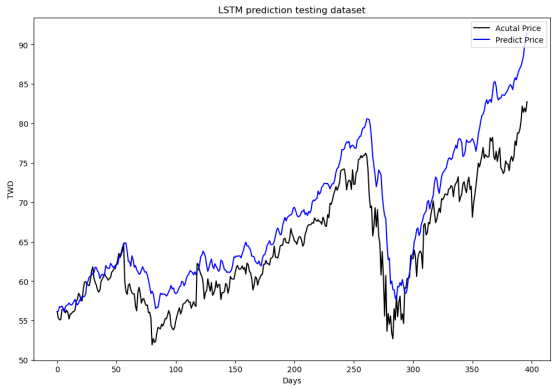


Figure 6.9: LSTM test result

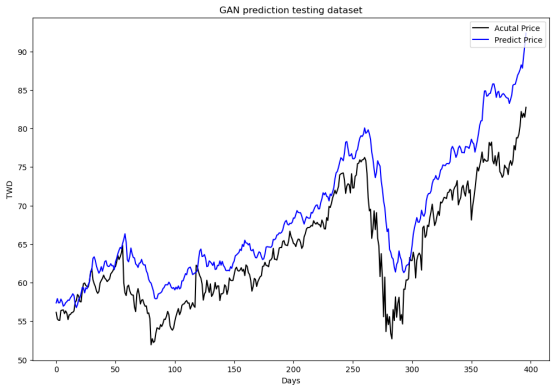


Figure 6.10: Gan Testing Result

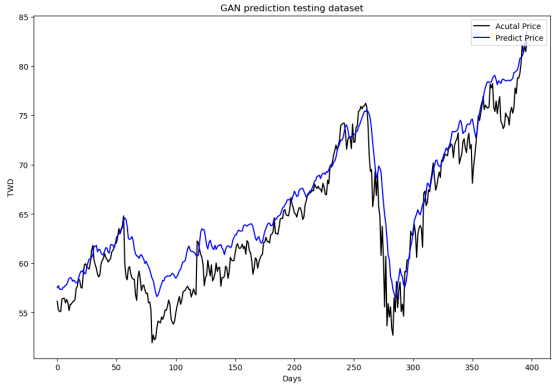


Figure 6.11: WGAN test result

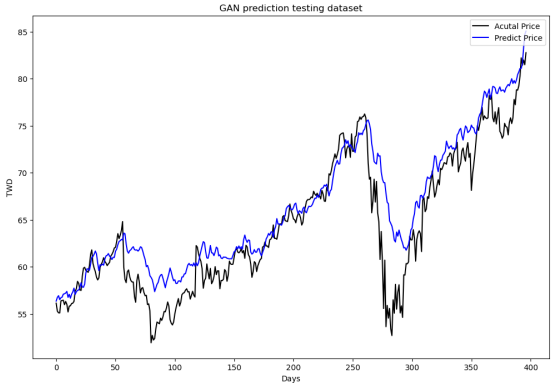


Figure 6.12: MHGAN test result

Table 6.2: Comparison of the different models on test data

Models	RMSE	MAE	MAPE
LSTM	4.378	3.603	0.055
GAN	4.975	4.142	0.064
WGAN	2.412	1.804	0.029
MHGAN	3.213	2.304	0.037

The following is a comparison of the models based on the errors scores obtained by subsampling the test data and generating predictions on those subsets of the test data. T-tests are performed on the array of errors from each model to determine whether there is a significant difference between the two models.

6.3.1 LSTM vs GAN

H0: no statistically significant difference between the errors between the LSTM and GAN

H1: there is a statistically significant difference between the errors between the LSTM and GAN

Test Statistic = -0.504

p-value = 0.622

As the p-value is greater than the significance level of 0.05 we Fail to reject the H0 that there is no statistically significant difference between the errors between the LSTM and GAN.

This result demonstrates that the GAN does not have a statistically better performance compared to the LSTM.

6.3.2 LSTM vs WGAN

H0: no statistically significant difference between the errors between the LSTM and WGAN

H1: there is a statistically significant difference between the errors between the LSTM and WGAN

Test Statistic = 2.19

p-value = 0.046

As the p-value is less than the significance level of 0.05 we reject the H0 that there is no statistically significant difference between the errors between the LSTM and WGAN.

From this result it can said that the Wassterstein GAN does statistically improve the performance on the data compared to the LSTM. This aligns with the overall errors where the RMSE MAE and MAPE all fall in the WGAN compared to the LSTM.

6.3.3 LSTM vs MHGAN

H0: no statistically significant difference between the errors between the LSTM and MHGAN

H1: there is a statistically significant difference between the errors between the LSTM and MHGAN

Test Statistic = 1.29

p-value = 0.217

As the p-value is greater than the significance level of 0.05 we Fail to reject the H0 that there is no statistically significant difference between the errors between the LSTM and

MHGAN.

Although the RMSE for MHGAN compared LSTM falls from 4.378 to 3.213, it is concluded that from this result that the MHGAN does not have a statistically better performance compared to the LSTM.

6.3.4 GAN vs WGAN

H0: no statistically significant difference between the errors between the WGAN and GAN

H1: there is a statistically significant difference between the errors between the WGAN and GAN

Test Statistic = 2.611

p-value = 0.02

As the p-value is less than the significance level of 0.05 we reject the H0 that there is no statistically significant difference between the errors between the WGAN and GAN.

It can conclude that from this result that the Wassterstein GAN does have a statistically better performance compared to the GAN. This aligns with the overall test RMSE which reduces from 4.975 to 2.412.

6.3.5 GAN vs MHGAN

H0: no statistically significant difference between the errors between the MHGAN and GAN

H1: there is a statistically significant difference between the errors between the MHGAN and GAN

Test Statistic = 1.74

p-value = 0.104

As the p-value is greater than the significance level of 0.05 we Fail to reject the H0 that there is no statistically significant difference between the errors between the MHGAN and GAN.

This result shows us that the Metropolis Hastings GAN does not statistically improve the performance compared to the traditional GAN although it should be mentioned that the overall RMSE falls from 4.975 to 3.213 which may indicate some improvement.

7 Conclusion

7.1 Discussion of the Results

The GAN model did not exhibit significant improvements in predictive accuracy over the LSTM during testing. This underperformance could be attributed to potential issues in model training stability, as indicated by the models failure to converge as shown in figure 6.2. In GANs training the generator and discriminator engage in a adversarial loop, with the generator aiming to fool the discriminator and the discriminator tries to counter this by better distinguishing between the real and fake data. Ideally, this results in oscillating losses that gradually lead to convergence. However, the lack of movement in the losses suggests that neither the generator nor the discriminator were effectively learning and adapting during training. The lack of adversarial training could be the reason that the GAN model fails to improve upon the LSTM model. The training process used may need to be improved and reviewed in order to facilitate proper training between the generator and discriminator.

While the Metropolis Hastings GAN does reduce overall errors compared to the LSTM and GAN models, these improvement are not statistically significant. This limitation might stem from the ineffectiveness of the original GAN's discriminator during training, which impacts its role in the Metropolis Hastings sampling process. If the imperfect discriminator cannot accurately assess the candidate in the Metropolis Hastings Sampling, it cannot effectively guide the generator towards producing more realistic outputs. Although there is not a significant improvement in the performance of the MHGAN model, the observed reduction in overall test error scores indicates a potential improvement in the model's ability to generalize to test data. Future iterations could benefit from enhancing discriminator training in the GAN model.

The WGAN with its gradient penalty and Wasserstein loss function, showcased greater stability in training and achieves the lowest overall test errors across the four models. It significantly outperformed both the LSTM and traditional GAN in predictive accuracy showing that the new architecture does have many benefits to use. Unlike the traditional GAN, the WGAN's training losses displayed dynamic oscillations which indicate effective adversarial learning illustrating the generator and discriminator ability to successfully adapt

and respond to each other's strategies. .The success of the WGAN underscores the potential of this architecture in managing the inherent challenges that can be seen in GAN training.

7.2 Future Works

Ultimately, the study's findings highlight the potential of GAN-based models in time-series forecasting. The WGAN, with its gradient penalty and Lipschitz enforcement, emerges as a particularly stable and promising model and the MHGAN, through its use of the Metropolis-Hastings sampling, shows promise in enhancing the model's performance. The combination of the two architectures with the integration of the Metropolis Hastings sampling into Wassterstein GAN framework can be looked at in further research. Such a hybrid model would be able to capitalize on the strenghts of both the WGAN and the MHGAN to produce better predictions on the data. Further research, may be able to test would the Metropolis Hastings Wassterstein GAN improve upon the performance of the models that we can looked at in the study.

This study focuses on one-step ahead forecasting but a multi step ahead prediction could also be investigated. By adjusting the sliding window for example using a lag size of 4 with window from t-3 to t and the targets set at t+1 and t+2 gives us the ability to make two step ahead forecast. This approach would require adjustments to the model's architecture and training process to accommodate the additional step in the forecast. The output layer would need to be modified to predict for two time steps simultaneously while the loss functions may need to be adjusted to account for accuracy at two future points instead of one. Implementing these changes would allow us to assess how well the models can handle extended predictions and manage the increased complexity associated with multi step forecasting.

The study's use of the Random Search could be replaced by other hyperparameter optimization techniques such as Bayesian Optimization. Finding the best hyperparameters can be challenging like exhaustive search being computationally expensive and the method used in this paper Random Search may sometimes be inefficient. Bayesian Optimization is a strategy for finding the minimum or maximum of a function and is particularly useful in hyperparameter optimization as it builds a probabilistic model of the function's landscape and the performance of the machine learning model as a function of the hyperparameters and it uses this model to make decisions about which pints in the hyperparameter space to try next. This was explored in the 2019 paper 'Hyperparameter Optimization for Machine Learning Models Based on Bayesian Optimization' by Wu et al [61], and its adoption could significantly enhance model performance in further reseach of these models.

Further research may delve into other architectures and loss functions, potentially exploring newer variants of these generative models. As the field of machine learning continues to evolve rapidly, the exploration of these advanced techniques will likely unlock further improvements in the predictive performance of financial time-series models. As discussed in the related works, the TimeGAN proposed by Yoon et al could be a model architecture that could be investigated further. This architecture was specifically tailored for time series and has shown promising results in their paper [2]. Investigating this model may provide insights into their efficacy compared to the models assessed in this study in regards to time-series analysis capabilities .

7.3 Conclusion

This study's exploration of various generative models for financial time series forecasting has yielded promising results, shows the capability of deep learning techniques in capturing the complex dynamics of financial markets. The use of a LSTM as a generator and a CNN as a discriminator within the GAN framework effectively captures the trends and fluctuations of Google's stock price, highlighting the strength of this architecture.

The investigation highlights that among the GAN variants, the Wasserstein GAN demonstrated its superior performance, as it significantly enhances training stability and model reliability, which are crucial in the volatile nature of stock price movements. The study shows the potential of the Metropolis Hastings GAN which refines sample selection through Metropolis Hastings sampling, although did not show significant improves over the GAN.

The comparative analysis with traditional LSTM models reveals the capabilities of GANs to offer a robust and accurate predictions in a sector known for its unpredictability. This research paves the way for further investigations into combining and refining these models, aiming to improve and expand their predictive power in financial time series forecasting.

8 Appendix

https://github.com/david2211shan/FYPUCC

Bibliography

[1] B. Banushev, "Using the latest advancements in ai to predict stock market movements," 2018, https://github.com/borisbanushev/stockpredictionai/blob/master/readme.md#42-metropolis-hastings-gan-and-wasserstein-gan-.

[2] M. v. d. S. J.Yoon, D.Jarrett, "Time-series generative adversarial networks," *Advances in Neural Information Processing Systems 32 (NeurIPS 2019)*, 2019.

[3] aakarshachug, "Deep learning | introduction to long short term memory," https://www.geeksforgeeks.org/deep-learning-introduction-to-long-short-term-memory/.

[4] M. Mishra, "Convolutional neural networks, explained," https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.

[5] M. B. D.-F. S. A. Y. Ian J. Goodfellow, J.Pouget-Abadie†, "Generative adversarial nets," 2014.

[6] L. Martin Arjovsky, S.Chintala, "Wasserstein gan," *arXiv:1701.07875v3*, 2017.

[7] E. Y.-J. R.Turner, J.Hung, "Metropolis-hastings generative adversarial networks," *arXiv:1811.11357*, 2017.

[8] A. Roy, "Introduction to autoencoders," 2020, https://towardsdatascience.com/introduction-to-autoencoders-7a47cf4ef14be.

[9] pawangfg, "Variational autoencoders," 2023, https://www.geeksforgeeks.org/variational-autoencoders/.

[10] P. Ruiz, "Ml approaches for time series," 2019, https://towardsdatascience.com/ml-approaches-for-time-series-4d44722e48fe.

[11] M. E. M. V. B. Emmanuel, L. Borja, "Chapter 1: Autoregressive integrated moving average (arima)," 2023, https://phdinds-aim.github.io/time_series_handbook/01_AutoRegressiveIntegratedMovingAverage/01_AutoRegressiveIntegratedMovingAverage.html.

[12] C. K. A. A. Ariyo Adebiyi, A. Oluyinka Adewumi, "Comparison of arima and artificial neural networks models for stock price prediction," *Journal of Applied Mathematics Volume 2014*, 2014.

[13] J. Brownlee, "Deep learning for time series forecasting," 2017, https://machinelearningmastery.com/deep-learning-for-time-series-forecasting/#:~:text=Traditionally%2C%20time%20series%20forecasting%20has%20been%20dominated%20by,to%20outputs%20and%20support%20multiple%20inputs%20and%20outputs.

[14] B. G. J. Cristian, "Deep learning for time-series analysis," *arXiv:1701.01887v1*, 2017.

[15] M. v. d. S. J.Yoon, D.Jarrett, "A comparison of arima and lstm in forecasting time series," *17th IEEE International Conference on Machine Learning and Applications*, 2018.

[16] J. Brownlee, "Using cnn for financial time series prediction," 2021, https://machinelearningmastery.com/using-cnn-for-financial-time-series-prediction/.

[17] J. D. S. W. Y. W. JKang Zhanga, Guoqiang Zhonga, "Stock market prediction based on generative adversarial network," *2018 International Conference on Identification, Information and Knowledge in the Internet of Things, IIKI 2018*, 2018.

[18] R. A. C. Romero, "Generative adversarial network for stock market price prediction," 2018.

[19] J. S.Hochreiter, "Long short-term memory," *Neural computation,9(8), 1735-1780*, 1997.

[20] J. Brownlee, "An introduction to recurrent neural networks and the math that powers them."

[21] Y. R.Pascanu, T.Mikolov, "On the difficulty of training recurrent neural networks," *arXiv:1211.5063*, 2003.

[22] Y. L. L. B. Y. B. P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE, vol. 86, no. 11,*, 1998.

[23] G. E. H. A.Krizhevsky, I.Sutskever, "Imagenet classification with deep convolutional neural networks," 2012.

[24] R. N. K. O'Shea, "An introduction to convolutional neural networks," *arXiv:1511.08458v2*, 2015.

[25] J. C. Olamendy, "Understanding relu, leakyrelu, and prelu: A comprehensive guide," https://medium.com/@juanc.olamendy/understanding-relu-leakyrelu-and-prelu-a-comprehensive-guide-20f2775d3d64.

[26] J. Hui, "Gan — wasserstein gan wgan-gp," https://jonathan-hui.medium.com/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490.

[27] M. V. A. I.Gulrajani, F.Ahmed, "improved training of wasserstein gans." *arXiv:1704.00028*, 2017.

[28] S. Agrahari, "Monte carlo markov chain (mcmc), explained," https://towardsdatascience.com/monte-carlo-markov-chain-mcmc-explained-94e3a6c8de11.

[29] M. N. R. A. H. T. E. T. N. Metropolis, A. W. Rosenbluth, "Equation of state calculations by fast computing machines," *J. Chem. Phys. 21, 1087–1092 (1953)*, 1953.

[30] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika, Vol. 57, No. 1 (Apr., 1970)*, 1970.

[31] C. Robert, "The metropolis-hastings algorithm," *arXiv:1504.01896v3*, 2016.

[32] utsavgoel, "Ml | overview of data cleaning," 2024, https://www.geeksforgeeks.org/data-cleansing-introduction/.

[33] R. R. X. Li, Y. Wang, "A survey on sparse learning models for feature selection," *IEEE Transactions on Cybernetics*, 2020.

[34] nikhilbhoi973, "What is feature engineering?" 2023, https://www.geeksforgeeks.org/what-is-feature-engineering/.

[35] S. Dongrey, "Study of market indicators used for technical analysis," *International Journal of Engineering and Management Research*, 2022.

[36] T. Zhang, "Trend following system for stock index trading," 2016, https://ssrn.com/abstract=2732889.

[37] P. Nabriya, "Generating trade signals using moving average(ma) crossover strategy — a python implementation," 2020, https://towardsdatascience.com/making-a-trade-call-using-simple-moving-average-sma-crossover-strategy-python-implementation-2996

[38] M. B. PERRY, "The exponentially weightedmoving average," 2010.

[39] A. Khatua, "An application of moving average convergence and divergence (macd) indicator on selected stocks listed on national stock exchange (nse)," 2016, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2872665.

[40] J. Bollinger, "Using bollinger bands," 1992.

[41] Y. Otsuka and T. Hasuike, "Effectiveness of momentum indicators to improve accuracy of stock price prediction for large-captial stocks," *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*, 2018.

[42] K. Omar, "Deconstructing time series using fourier transform," 2020, https://medium.com/@khairulomar/deconstructing-time-series-using-fourier-transform-e52dd535a44e.

[43] U. Oberst, "The fast fourier transform," *SIAM Journal on Control and Optimization*, 2007.

[44] J. Rocca, "Understanding variational autoencoders (vaes)," 2019, https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.

[45] S. Paul, "Reparameterization trick in variational autoencoders," 2020, https://towardsdatascience.com/reparameterization-trick-126062cfd3c3.

[46] Shivanipickl, "What is feature scaling and why does machine learning need it?" 2023, https://medium.com/@shivanipickl/what-is-feature-scaling-and-why-does-machine-learning-need-it-104eedebb1c9.

[47] J. Brownlee, "How to use standardscaler and minmaxscaler transforms in python," 2020, https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/.

[48] ——, "Time series forecasting as supervised learning," 2020, https://machinelearningmastery.com/time-series-forecasting-supervised-learning/.

[49] ——, "Stacked long short-term memory networks," https://machinelearningmastery.com/stacked-long-short-term-memory-networks/.

[50] ——, "A gentle introduction to dropout for regularizing deep neural networks," https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/.

[51] P. AI, "Understanding the 3 most common loss functions for machine learning regression," https://towardsdatascience.com/understanding-the-3-most-common-loss-functions-for-machine-learning-regression-23e0ef3e14d3.

[52] J. L. B. D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv:1412.6980v9*, 2014.

[53] W. Z. V. C. A. R. X. C. T. Salimans, I. Goodfellow, "Improved techniques for training gans," *arXiv:1606.03498v1*, 2016.

[54] C. E. B. Zadrozny, "Transforming classifier scores into accurate multiclass probability estimates," *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[55] Devansh, "How does batch size impact your model learning," https://medium.com/geekculture/how-does-batch-size-impact-your-model-learning-2dd34d9fb1fa.

[56] A. Rakhecha, "Understanding learning rate," https://towardsdatascience.com/https-medium-com-dashingaditya-rakhecha-understanding-learning-rate-dd5da26bb6de.

[57] manmayi, "Choose optimal number of epochs to train a neural network in keras," https://www.geeksforgeeks.org/choose-optimal-number-of-epochs-to-train-a-neural-network-in-keras/.

[58] A. Sankar, "Demystified: Wasserstein gan with gradient penalty(wgan-gp)," https://towardsdatascience.com/demystified-wasserstein-gan-with-gradient-penalty-ba5e9b905ead.

[59] J. Brownlee, "Hyperparameter optimization with random search and grid search," https://machinelearningmastery.com/hyperparameter-optimization-with-random-search-and-grid-search/.

[60] Y. B. J. Bergstra, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, 2012.

[61] H. Z. L. X. H. L. S. D. J. Wu, X. Chen, "Hyperparameter optimization for machine learning models based on bayesian optimization," *Journal of Electronic Science and Technology Volume 17 Issue 1*, 2019.