Title

Deep Learning and Machine Learning Models for High-Frequency Stock Price Prediction and Inference

Author

Zhang, Yuelong

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Deep Learning and Machine Learning Models for High-Frequency Stock Price Prediction and Inference

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Statistics

by

Yuelong Zhang

2024

ABSTRACT OF THE THESIS

Deep Learning and Machine Learning Models
for High-Frequency Stock Price Prediction and Inference

by

Yuelong Zhang
Master of Science in Statistics
University of California, Los Angeles, 2024
Professor George Michailidis, Chair

This thesis investigates the use of deep learning and machine learning models for high-frequency stock price prediction and inference. By using models such as Long Short-Term Memory (LSTM) networks, Convolutional LSTMs (CLSTM), and Transformer architectures, this work evaluates the predictive performance of these models in both single-step and multi-step stock price prediction tasks. The models are trained on various datasets, including those with technical indicators, sentiment analysis, and the US Dollar Index, along with Fourier-transformed features for improved feature engineering. The results demonstrate that Transformer-based models, particularly those added with convolutional layers, outperform LSTM-based models in capturing long-term dependencies and making accurate predictions over extended time periods. Additionally, the Fourier-transformed features enhances overall models performance by revealing underlying periodic patterns in stock prices. This research contributes to the growing literature on stock price prediction and inference by offering insights into model architectures and feature engineering techniques that improve the accuracy of financial forecasting.

The thesis of Yuelong Zhang is approved.

Frederic R. Paik Schoenberg

Yingnian Wu

George Michailidis, Committee Chair

University of California, Los Angeles

2024

To my family, whose unwavering support and encouragement have guided me throughout my life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

The stock market is a financial market where investors aim to maximize returns by buying and selling stocks and their derivatives. Stock prices are influenced by many factors, both short-term and long-term, including news, market sentiment, historical stock data, and macroeconomic indicators. The combination of these factors contributes to the volatility and unpredictability of stock prices.

Stock price prediction involves forecasting the future values of stock prices based on historical data and other relevant information. Traditional methods for stock price prediction include statistical models like ARIMA, GARCH, and linear regression. However, these models sometimes fail to capture the nonlinear patterns and complex relationships in financial time series data. By introducing machine learning and deep learning in the stock price prediction, there has been a paradigm shift in how stock prices are predicted. Advanced models such as RNN related neueral networks and Transformer related models have shown promising results in capturing the long term and short term temporal dependencies and complicated patterns present in high-frequency financial data.

This thesis aims to explore the use of machine learning and deep learning models to predict and infer on stock prices and stock statistics. By analyzing the performance of different model architectures, this thesis tries to contribute to the growing body of literature on stock price prediction and inference.

CHAPTER 2

Literature Review

2.1 Prediction Task and Inference Task

In the context of machine learning and deep learning, prediction is the process of using a trained model to predict an output based on input data. This is the final step in the model where the model applies what it has learned from training data to new data. However, for Inference, is to draw conclusions about the underlying relationships and dependencies within the data. It involves understanding which features are important, the effect of each feature, and the causal relationships.

Deep learning often performs better in prediction tasks compared with traditional machine learning models [LBH15]. However, machine learning models have more interpretability than deep learning models.

2.2 Optimization

The training of machine learning or deep learning models can be framed as an optimization problem. The objective is to find the model parameters that minimize a loss function, which measures the difference between the predicted outputs and the target values in the training data.

Given a dataset $\{(x_i, y_i)_{i=1}^N\}$, where x_i represents the ith example and y_i the corresponding target outputs, we try to solve:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L\left(y_i, f\left(x_i; \theta\right)\right),$$

where $f(x_i; \theta)$ is the model's prediction for input x_i with model parameters θ, and L is the loss function (e.g., mean squared error for regression or cross-entropy loss for classification).

One approach is to find the parameters that globally minimize the loss function. In certain cases, such as linear regression and logistic regression, we can derive an analytic solution directly because the loss functions are convex. However, for more complex models, especially those with high-dimensional parameter spaces, deriving an analytic solution is often infeasible. The loss function may be non-convex and involve many local minima and saddle points, making it difficult to find the global minimum analytically.

2.2.1 Gradient-Based Optimization Methods

To overcome this challenge, gradient-based optimization algorithms are used in training due to their effectiveness in high-dimensional spaces typical of deep learning models. These methods iteratively update the model parameters in the direction that minimizes the loss function based on the calculated gradients.

The basic gradient descent algorithm updates the model parameters in the opposite direction of the gradient of the loss function with respect to the parameters[Bis06]:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} L\left(\theta_t\right),$$

where $\nabla_{\theta} L\left(\theta_t\right)$ is the gradient and η is the learning rate, controlling the step size of each update.

2.2.1.1 Second-Order Methods

In addition to gradient, second-order methods incorporate second derivatives to use the curvature of the loss surface. This can lead to more precise parameter updates and potentially faster convergence [NW06].

Newton's Method Newton's method uses both the gradient and the Hessian matrix (the matrix of second-order partial derivatives) to update the parameters:

$$\theta_{t+1} = \theta_t - \eta H^{-1}(\theta_t) \nabla_\theta L\left(\theta_t\right)$$

where $H(\theta_t)$ is the Hessian matrix evaluated at θ_t. By using curvature of the loss function, Newton's method can make more precise steps towards the minimum.

Challenges with Hessian-Based Methods Despite their potential advantages, Hessian-based methods come with significant computational challenges [GBC16]:

- Computation and Memory Cost: Computing the Hessian matrix and its inverse is computationally expensive, especially for models with a large number of parameters. Storing the Hessian matrix also requires a large amount of memory, which is $O(k^2)$, where k is the number of parameters for the model f. In the context of deep learning neural networks, there are millions of parameters, so the storage of the Hessian matrix may exceed the available GPU memory (e.g., CUDA GPU memory), making it impractical to store and manipulate the matrix directly.
- Ill-Conditioning: The Hessian matrix may be singular or ill-conditioned; therefore, it is hard to invert the matrix.

2.2.2 Advanced Optimization algorithm

Momentum Method Momentum methods use past gradients to smooth out updates, helping to navigate ravines and accelerate in consistent gradient directions. The parameter update procedures are:

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta L(\theta_t) \\
\theta_{t+1} &= \theta_t - v_t \\
v_0 &= 0
\end{aligned}
$$

where:

- v_t is the velocity (momentum term) at time step t.
- γ is the momentum coefficient (e.g., 0.9).
- η is the learning rate, just like in gradient method.
- $\nabla_\theta L(\theta_t)$ is the gradient of the loss function with respect to parameters θ at time t.

AdaGrad AdaGrad [DHS11] adapts the learning rate for each parameter individually based on historical gradients, performing larger updates for infrequent parameters:

$$
\begin{aligned}
G_0 &= 0 \\
\theta_0 &= \text{initial parameters} \\
g_t &= \nabla_\theta L(\theta_t) \\
G_t &= G_{t-1} + g_t \odot g_t \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t
\end{aligned}
$$

where:

- g_t is the gradient at time t.
- G_t is the sum of the squares of the gradients up to time t (accumulated squared gradients).
- θ_t represents the parameters at time t.
- ϵ is a small constant to prevent division by zero (e.g., 10^{-8}).
- η is the learning rate.

RMSProp RMSProp [TH12] modifies AdaGrad by using a moving average of squared gradients to prevent the learning rate from decaying too quickly:

$$
\begin{aligned}
E[g^2]_0 &= 0 \\
\theta_0 &= \text{initial parameters} \\
g_t &= \nabla_\theta L(\theta_t) \\
E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t \odot g_t \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \odot g_t
\end{aligned}
$$

where:

- g_t is the gradient at time t.
- $E[g^2]_t$ is the exponentially weighted moving average of the squared gradients.
- γ is the decay rate (e.g., 0.9).
- ϵ is a small constant to prevent division by zero (e.g., 10^{-8}).
- η is the learning rate.
- θ_t represents the parameters at time t.

Adam (Adaptive Moment Estimation) Adam [KB15] combines momentum and RMSProp by maintaining running averages of both gradients and their squared values, providing an adaptive learning rate:

$$
\begin{aligned}
\theta_0 &= \text{initial parameters} \\
m_0 &= 0 \\
v_0 &= 0 \\
g_t &= \nabla_\theta L(\theta_t) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t
\end{aligned}
$$

where:

- g_t is the gradient at time t.
- m_t is the first moment estimate (mean of the gradients).
- v_t is the second moment estimate (uncentered variance of the gradients).
- β_1 and β_2 are the exponential decay rates for the moment estimates (e.g., 0.9 and 0.999).
- $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected estimates.
- ϵ is a small constant to prevent division by zero (e.g., 10^{-8}).
- η is the learning rate.

- θ_t represents the parameters at time t.

2.3 Deep Learning Models

By using various innovative layers and stacking layers together, deep neural networks can learn the complex structures and relationships within the data, and therefore are good at handling complex tasks such as image recognition, natural language processing, and time-series forecasting.

2.3.1 Common Layers in Neural Network

2.3.1.1 Pooling Layer

Pooling layers are used to reduce the spatial dimensions (width and height) of the input feature maps, thereby decreasing the number of parameters and computation in the training. The two most common types of pooling are max pooling and average pooling. The parameter of the maxpooling is p, s, where p is the height and width of pooling, and s is the stride of the pooling. For maxpooling, the output $y_{i,j} = \max_{0 \le m < p,\ 0 \le n < p} x_{s \cdot i + m, , s \cdot j + n}$; for average pooling, the ouput $y_{i,j} = \frac{1}{p^2} \sum_{m=0}^{p-1} \sum_{n=0}^{p-1} x_{s \cdot i + m, , s \cdot j + n}$

2.3.1.2 Dropout Layer

Dropout layers help prevent the model from overfitting to the training data by randomly setting a fraction of input units to zero during training [SHK14]. This regularization method makes the network have redundant representations and improves generalization to new data.

Dropout works by generating a binary mask $\mathbf{r} = [r_1, r_2, \ldots, r_n]$, where each r_i is independently sampled from a Bernoulli distribution: $r_i \sim \text{Bernoulli}(p)$ where p is the probability of keeping the unit. The dropout output is $\tilde{\mathbf{x}} = \mathbf{r} \odot \mathbf{x}$

2.3.1.3 Normalization Layer

Normalization layers help stabilize and accelerate the training of deep neural networks by standardizing the inputs [IS15]. One common normalization technique is Batch Normalization, which works by computing the mean μ_B and variance σ_B^2 of the inputs over a mini-batch:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2, \quad \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where m is the number of samples in the mini-batch and ϵ is a small constant added for numerical stability. We also have learnable parameters γ (scale) and β (shift): $y_i = \gamma\hat{x}_i + \beta$.

2.3.1.4 Fully Connected Layer

Fully connected layers are fundamental components in neural networks where each neuron in the layer is fully connected to every neuron in the previous layer. This structure allows the network to learn linear relationships between inputs and outputs $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$ where $\mathbf{W}$ and $\mathbf{b}$ are learnable parameters.

2.3.1.5 Activation Layer

Activation layers introduce non-linearity into neural networks, making them to learn complex patterns in data. Without those activation functions, neural networks would be limited to learning only linear mappings. There are a few commonly used activation functions:

Common activation functions include:

- Sigmoid Function:

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

- Hyperbolic Tangent (Tanh) Function:

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Rectified Linear Unit (ReLU):

$$f(x) = \max(0, x)$$

2.3.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a class of deep neural networks specifically designed to process data with a grid-like data, such as images [LBB98, LBH15]. A typical CNN architecture comprises multiple layers, including convolutional layers, pooling layers, dropout layers, fully connected layers, and activation layer. The convolutional layers use a set of learnable filters (kernels) that convolve across the input data to produce feature maps.

The convolution operation for a single output feature map is defined as:

$$y_{i,j}^{(k)} = f\left(\sum_{c=1}^{C}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} x_{i+m,,j+n}^{(c)} \cdot w_{m,n}^{(c,k)} + b^{(k)}\right),$$

where:

- $y_{i,j}^{(k)}$ is the output at position (i, j) in the k-th feature map.
- $x_{i+m,,j+n}^{(c)}$ is the input at position $(i + m, j + n)$ in the c-th input channel.
- $w_{m,n}^{(c,k)}$ is the weight of the kernel at position (m, n) connecting the c-th input channel to the k-th feature map.
- $b^{(k)}$ is the bias term for the k-th feature map.
- $f(\cdot)$ is the activation function, such as ReLU.
- C is the number of input channels(for example, image with RGB representation has three chaneels)
- M and N are the height and width of the kernel, respectively.

2.3.3 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of neural networks used to model sequential data by incorporating temporal dependencies [Elm90]. They are widely used in natural language processing, speech recognition, and time-series analysis.

A RNN nueral network keeps a hidden state at each time t that is updated at each time step based on the current input and the previous hidden state:

$$\mathbf{h}_t = \phi\left(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h\right),$$

where:

- $\mathbf{h}_t$ is the hidden state at time t and $\mathbf{x}_t$ is the input at time t.
- $\mathbf{W}_{xh}$ and $\mathbf{W}_{hh}$ are weight matrices.
- $\mathbf{b}_h$ is the bias vector, and $\phi(\cdot)$ is the activation layer.

The vanilla RNN has a problem of vanishing and exploding gradients during training, making it difficult to capture long-term data structure. This limitation restricts its usage in prediction for long time sequences.

2.3.3.1 Long Short-Term Memory (LSTM)

The Long Short-Term Memory (LSTM) network addresses the gradient explosion and vanishing problem by introducing memory cells and gating mechanisms [HS97].

An LSTM cell comprises an input gate $\mathbf{i}_t$, a forget gate $\mathbf{f}_t$, an output gate $\mathbf{o}_t$, and a cell state $\mathbf{c}_t$:

$$
\begin{aligned}
\mathbf{i}_t &= \sigma\left(\mathbf{W}_{xi}\mathbf{x}_t + \mathbf{W}_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i\right), \\
\mathbf{f}_t &= \sigma\left(\mathbf{W}_{xf}\mathbf{x}_t + \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f\right), \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o\right), \\
\mathbf{g}_t &= \tanh\left(\mathbf{W}_{xg}\mathbf{x}_t + \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{b}_g\right), \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t, \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh\left(\mathbf{c}_t\right),
\end{aligned}
$$

$\sigma(\cdot)$ is the sigmoid function, and $\odot$ denotes element-wise multiplication.

2.3.3.2 Attention Mechanism

The attention mechanism allows models to adjust weights and focus on specific parts of the input sequence when generating each part of the output sequence [BCB15].

The attention weights between the query vector $\mathbf{q}$ and the key vectors $\mathbf{k_i}$ are calculated as:

$$
\alpha_{\mathbf{q},\mathbf{k_i}} = \text{softmax}\left(A(\mathbf{q}, \mathbf{k_i})\right) = \frac{\exp\left(A(\mathbf{q}, \mathbf{k_i})\right)}{\sum_{j=1}^{N} \exp\left(A(\mathbf{q}, \mathbf{k_j})\right)}
$$

where A is the alignment function $A : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, measuring the similarity between query and key, and N is the number of key-value pairs in the input sequence.

2.3.4 Transformer

Introduced by Vaswani et al. [VSP17], the Transformer architecture marked a paradigm shift in natural language processing by relying on self-attention and eliminating the need for recurrence and convolutions. This approach makes Transformers to achieve state-of-the-art results across a wide range of Natural Language Processing tasks.

2.3.4.1 Encoder

The Transformer encoder has stacked layers, each containing two main layer: a multi-head self-attention layer and a position-wise feed-forward layer. The self-attention mechanism allows the model to weigh the relevance of different x_i in the input sequence when encoding a particular x_k.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

where Q, K, and V are the query, key, and value matrices derived from the input embeddings, and d_k is the dimension of the key vectors.

Multi-Head Attention To capture different types of relationships and interactions, the Transformer also uses multi-head attention, which runs multiple attention mechanisms in parallel. If the input data have a total dimension of d_{model}, the multi-head attention mechanism splits these dimentsions into h heads, and each head has a dimension of $d_k = \frac{d_{\text{model}}}{h}$.

The multi-head attention is computed as:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\, W^O, W^O \in \mathbb{R}^{h \times d_k \times d_{\text{model}}}$$

$$\text{head}_i = \text{Attention}(QW_i^Q,\ KW_i^K,\ VW_i^V), \quad W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad .$$

By concatenating the outputs of all attention heads, the model combines information from multiple representation subspaces. The total dimension after concatenation is $h \times d_k$, which matches the original input dimension d_{model}.

2.3.4.2 Decoder

The Transformer decoder also has stacked layers, with three main components in each layer: a masked multi-head self-attention layer, an encoder-decoder attention layer, and a position-

wise feed-forward layer. The masking in the decoder ensures that the model only attends to earlier positions in the output sequence during training.

2.4 Machine learning model

In contrast to deep learning models, which are difficult to interpret due to their large number of parameters and complex network architectures, traditional machine learning models are more interpretable.

2.4.1 Extreme Gradient Boosting Tree

As introduced by [CG16], the Extreme Gradient Boosting Tree is an ensemble learning model that combines both decision tree algorithms and gradient boosting. Extreme gradient boosting tree model is based on tree methods, and optimizes the loss function with the gradient boosting method.

Tree Method

Decision tree is a basic machine learning algorithm used for both classification and regression. Decision Tree is a tree with a root node, which represents the entire dataset, multiple internal node, where data is flowed throught based on certain feature values, and leaf nodes, whcih are terminal nodes that give final results.

Decision tree partitions the dataset into subsets where most of the examples either belong to specific classes for classification tasks or have mean values that are close to the target values.

The decision tree algorithm works by recursively selecting the optimal feature and threshold for splitting the data at each node. For classificaiton tasks, the selection is guided by metrics such as the Gini index and entropy, which measures the quality of each possible split

to find the best split. The Gini index [BFO84] is defined as $\mathbf{Gini} = 1 - \sum_{i=1}^{C} p_i^2$ where p_i is the proportion of examples belonging to certain class i in particular node, and C is toal number of classes. The Entropy metric is introduced by [Qui86]: $\mathbf{E} = \sum_{i=1}^{C} -p_i \log_2(p_i)$. For regression task, the metric is chosen as Mean Square Error as described in [BFO84]: $MSE(t) = \frac{1}{N_t} \sum_{i \in t} (y_i - \bar{y}_t)^2$ where t is the node in the tree.

For Gini index, the best split is the one that results in lowest Gini index. For a split node t, the Gini gain can be defined as

$$\text{Gini}(t) - (\frac{N_{\text{left}}}{N}\text{Gini}(t_{\text{left}}) + \frac{N_{right}}{N}\text{Gini}(t_{\text{right}}))$$

where N_{left} and N_{right} are the number of samples in the left and right child nodes, and t_{left} and t_{right} is the splitted left child node and right child node. For Entropy, the best split is the one that gives highest information gain, which is defined as following:

$$IG(T, A) = E(T) - \sum_{v \in Values(A)} \frac{|T_v|}{|T|}\text{Entropy}(T_v)$$

where T is set of training samples, T_v is the subset of T and attribute A has value v, and $|\dot{|}$ denotes the number examples in the set. For regression task with MSE as metric, the split is chosen to maximize the Mean Sqaure Error Reduction:

$$MSE(t) - (\frac{N_{left}}{N}MSE(t_{left}) + \frac{N_{right}}{N}MSE(t_{right}))$$

Gradient Boosting Method

According to [HTF09], the model $\hat{f}(x)$ mean expected squared error can be decomposed into two terms:

$$E\left[\left(\hat{f}(x) - f(x)\right)^2\right] = \left(Bias\left[\hat{f}(x)\right]\right)^2 + Var\left[\hat{f}(x)\right] + \sigma^2$$

where bias measures the miss due to the simplicity of the models, and variance measures the model's poor performance because of lack of generalization.

To reduce both bias and variance, the boosting method is introduced. Boosting method combines weak learners to form a strong learner, where each subsequent weak learner focuses

on the errors of the previous weak learners. Gradient Boosting builds models iteratively by optimizing a loss function using gradient descent [Fri01].

Extreme Gradient Boosting Tree Method

The Extreme gradient boosting tree algorithm [CG16] is an improvement alogrithm to the Gradient Boosting Tree. It mainly introduces following features

- Second-Order Optimization: Use of Hessian in the optimization process as mentioned in 2.2.1.1
- Regularization: XGBoost adds both L_1 and L_2 Regularization terms to the loss function.
- Parallel Processing and Scalability: XGBT is designed to leverage parallel computation, allowing it to be trained efficiently using multi-core CPU processing or accelerated with GPU parallel computing.

CHAPTER 3

Dataset Introduction

To simplify the prediction and inference tasks, we focus exclusively on the SPY(S&P 500 ETF), using a time frequency of one minute, and denotes this time interval as t.

3.1 Lookback Timestamp

In order to incorporate more information in time series, we choose $T = 1000$ time interval lookback, that is, include previouse 1000 minutes stock features, resulting in X has shape of $(N, T, D) = (N, 1000, D)$

3.2 Time interval

In this research, we uses two datasets: a half-year dataset from September 1, 2023, to March 15, 2024, and a full-year dataset from January 1, 2023, to March 15, 2024. Our goal is to evaluate the model's performance on both shorter and longer time interval to understand how the duration of data influences models prediction performance.

3.3 Single-step prediction

For single step prediction, the output y has only single dimension $d = 1$. Therefore, y has shape of $(N, 1)$

3.3.1 Basic Data (D_1)

We use stock data with following features:

- **current price**: the SPY price at time t
- **volumn**: total number of shares traded for SPY during time t
- **high price**: the highest price traded during time t
- **lower price**: the lowest stock price traded during time t
- **transactions**: the number of transactions traded during time t

Feature Engineering In addition to the basic features, we also use simple data manipulation to generate new features. For each existing feature, we calculate the mean and variance with a sliding window of 15 timestamps. At each time t, the mean m_t and variance v_t are defined as following:

$$m_t = \frac{1}{15}\sum_{i=t-15}^{t-1} x_i, \quad v_t = \frac{1}{15}\sum_{i=t-15}^{t-1}(x_i - m_t)^2.$$

3.3.2 Data with Technical Indicator (D_2)

In addition to the features mentioned in Basic Data (D_1), we also introduce several technical indicators commonly used by traders:

Simple Moving Average (SMA): Smooths price data by averaging over N periods:

$$\text{SMA}_t = \frac{1}{N}\sum_{i=t-N+1}^{t} x_i$$

Exponential Moving Average (EMA): Averages prices with more weight on recent data:

$$\text{EMA}_t = \alpha \cdot x_t + (1-\alpha)\cdot \text{EMA}_{t-1}, \quad \alpha = \frac{2}{N+1}$$

Relative Strength Index (RSI): Measures momentum, oscillating between 0 and 100:

$$\text{RSI}_t = 100 - \left(\frac{100}{1 + RS_t}\right), \quad RS_t = \frac{\text{Avg Gain}}{\text{Avg Loss}}$$

Moving Average Convergence Divergence (MACD): Shows momentum using moving averages:

- **MACD Value**: Difference between 12-period and 26-period EMAs:

$$\text{MACD Value}_t = \text{EMA}_{12,t} - \text{EMA}_{26,t}$$

- **MACD Signal**: 9-period EMA of MACD Value:

$$\text{MACD Signal}_t = \text{EMA}_{9,t}(\text{MACD Value})$$

- **MACD Histogram**: Difference between MACD Value and Signal:

$$\text{MACD Histogram}_t = \text{MACD Value}_t - \text{MACD Signal}_t$$

3.3.3 Dataset with sentiment analysis and Dollar Index (D_3)

In addition to Data with Technical Indicator (D_2), we also introduce two additional features: dollar index and sentiment indicator.

Dollar index In order to capture macroeconomic information, we also include Dollar Index (DXY). The Dollar Index measures the value of the United States dollar relative to a basket of foreign currencies. Typically, when the Dollar Index is strong, it reflects increased investor confidence in the US economy.

News Sentiment Indicator At each timestamp t, multiple news related U.S stock market occur globally. While some news can boost the confidence of the market, others may introduce panic. To capture this information, we also include the binary news sentiment

indicator(0 for negative, 1 for neutral, and 2 for positive), which is generated using BERT [DCL19]. We used a pretrained BERT model and finetuned it on a manually labeled stock news dataset.

3.4 Sequence Prediction

3.4.1 Sequence Data (D_4)

Intead of predicting a single point, we want to predict the next $T = 10$ timestamp stock price. For each input x_i, the target y has shape of $(10, 1)$ representing the stock price for the next 10 timestamps. The features and shape of input X are same as Dataset with sentiment analysis and Dollar Index (D_3).

3.4.2 Sequence Data with Fourier Transformation (D_5)

We introduce a new dataset, D_5, where we perform Discrete Fourier Tranformation on the five basic features:current price, high price, low price, volume, and transactions. The DFT represents a time-series signal as a sum of sinusoids, capturing both amplitude and phase information for each frequency component. For a time series $x[n]$ of length N, the DFT is defined as:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn}, \quad \text{where } k = 0, 1, \ldots, N-1,$$

where $X[k]$ is a complex number expressed as $X[k] = \mathrm{Re}[X[k]] + j\,\mathrm{Im}[X[k]]$.

For each input feature x at each timestamp t, we extract the amplitude $A = \sqrt{\mathrm{Re}[X[k]]^2 + \mathrm{Im}[X[k]]^2}$ and phase $\theta = \arctan\left(\frac{\mathrm{Im}[X[k]]}{\mathrm{Re}[X[k]]}\right)$ from the DFT. For a single feature, this process adds two new features: amplitude and phase. Therefore, we add ten more features into our dataset D_5.

By using the DFT, we can identify key periodicities and reduce noise by filtering out high-frequency components, therefore focusing on more meaningful patterns in the data [YZC23].

3.5 Data Preprocessing

We train the neural network by using the gradient based method, and use the backpropagation to calculate the gradient. During backpropagation in neural networks, gradients are propagated backward through multiple layers to update the model's parameters. This process can lead to problems of gradient vanishing or exploding, which will make training divergent. Therefore, we need to normalize the numerical input data by performing min-max scaling on each feature, defined as:

$$\hat{X}\cdot j = \frac{X\cdot j - \min(X_{\cdot j})}{\max(X_{\cdot j}) - \min(X_{\cdot j})}$$

This scaling transforms the features to a common range, between 0 and 1, increasing numerical stability during training and helping to prevent gradient-related issues.

3.6 Dataset Summary

Table 3.1: Dataset Summary

Dataset	**X**	**y**	**Description**
D_1	$(N_{time}, 1000, 10)$	$(N, 1)$	Basic stock feature
D_2	$(N_{time}, 1000, 16)$	$(N, 1)$	D1 + Technical indicator
D_3	$(N_{time}, 1000, 18)$	$(N, 1)$	D2 + Dollar index + Sentiment
D_4	$(N_{time}, 1000, 18)$	$(N, 10)$	Sequence Prediction
D_5	$(N_{time}, 1000, 28)$	$(N, 10)$	Sequence Prediction + Fourier Transformation

CHAPTER 4

Single-Step SPY Prediction

4.1 Problem Formantion

In the context of single-step SPY prediction, our goal is to predict the stock price at the next timestamp using historical data. The input to our model, denoted as $\mathbf{X}$, is a three-dimensional tensor with shape $N \times T \times D$, where N is the number of data samples, T is the number of previous timestamps considered, and$D$ is the number of features available at each timestamp.

The output of the model $\mathbf{y}$ with shape $N \times 1$ is the predicted stock price at the next timestamp for each x_i.

The loss function for our model is mean squared loss, which is defined as following:

$$\text{MSE}(\mathbf{X}, \mathbf{y}) = \frac{1}{N} \sum_{i=1}^{N} (f(\mathbf{x}_i) - y_i)^2$$

where:

- $f(\mathbf{x}_i)$ denotes the model's prediction for the i-th input sample $\mathbf{x}_i$,
- y_i is the true stock price for the i-th sample.

4.2 Model Introduction

4.2.1 Convolutional Neural Network

The input data has a shape of (N, T, D), which can be interpreted as a 2D image with time is the width and features is the height. This new interpretation of the time series data fit to the usage of Convolutional Neural Networks (CNNs) in computer vision for processing image data. Therefore, we used a CNN architecture with three convolutional layers with max pooling and batch normalization in each layer, and the final layer is a fully connected layer.

4.2.2 Long Short-Term Memory (LSTM) Network

LSTM are inherently good at capturing temporal dependencies because of their design, which uses the encoded context information from previous time intervals. Moreover, LSTMs address a common issue found in traditional RNNs—the vanishing gradient problem—by incorporating gates that manage information flow. This capability enables them to preserve data across extensive time sequences. Therefore, we used a single-layer LSTM, and a fully connected layer to project hiddent states to final output.

4.2.3 Convolutional LSTM

A challenge in stock prediction in our dataset is a shortage of features. Even though we performed the featuer engineering on the dataset, there is still a lack of features. To address this problem, we reshaped the input data to $(N, D, T, 1)$ and applied convolutional operations to the input features using a kernel size of $(1, 1)$ with zero padding. This approach generates new features from the existing ones. After a single convolutional layer, the data has the shape (N, T, C), where C represents the number of filters in the convolutional layer. The model then proceeds with an LSTM layer and a fully connected layer.

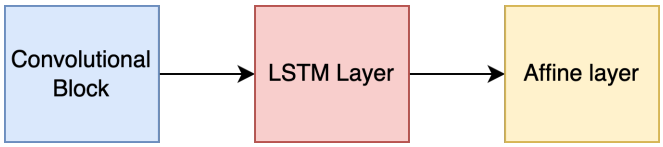


Figure 4.1: CLSTM Model Structure

4.2.4 Transformer (Encoder-Only)

The Transformer architecture addresses the limited parallelism inherent in RNN-based models while effectively managing long-range temporal dependencies, thereby enhancing scalability for this task. Unlike the original implementation [VSP17], which uses the ReLU activation function, we use the tanh activation function to mitigate the vanishing gradient problem associated with negative values. Furthermore, prior to the multihead attention layer, we include a linear layer to project the input data X to the specified dimension.

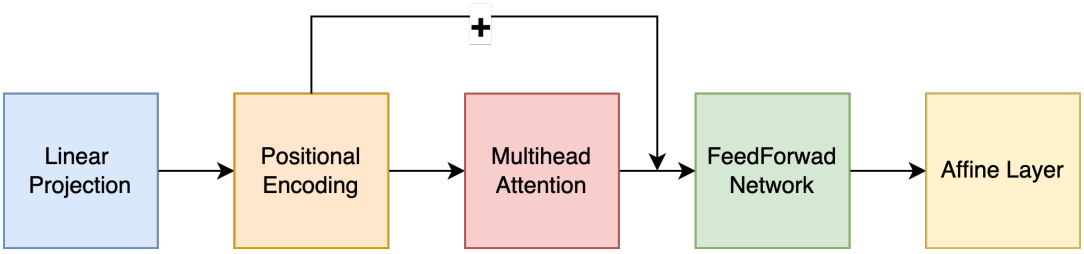


Figure 4.2: Tranformer(Encoder-only) Model Structure

4.2.5 Convolutional Transformer (Encoder-Only)

Similar to the approach in Section 4.2.3, we added a convolutional layer before the Transformer architecture to perform feature engineering instead of using the linear projection.

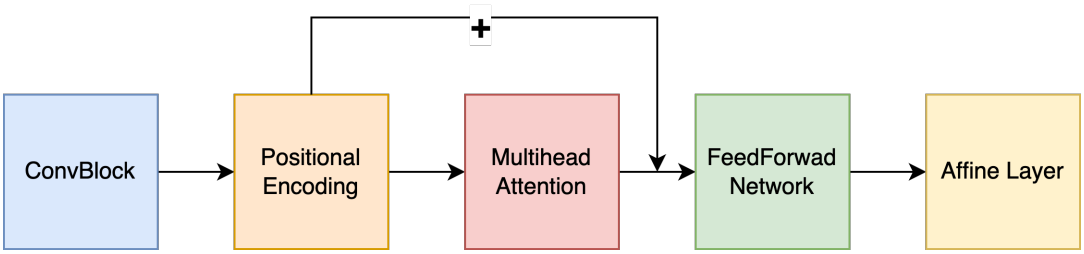


Figure 4.3: ConvTranformer(Encoder-only) Model Structure

4.3 Training Setup

Data Splits The dataset is partitioned into training, validation, and test sets in an 80:10:10 ratio, respectively. To maintain the temporal dependencies in the data, we split the dataset without shuffling.

Model Parameters

- **Convolutional Neural Network (CNN) Parameters:**
  - Kernel Sizes: $(5, 3)$ and $(3, 1)$
  - Max Pooling Sizes: $(5, 3)$ and $(3, 1)$
  - Number of Filters: 15 and 30
- **Long Short-Term Memory (LSTM) Parameters:**
  - Hidden Size: $3 \times D$
- **Convolutional LSTM (ConvLSTM) Parameters:**
  - Convolutional Block Filter Size: $2 \times D$
  - LSTM Block Hidden Size: $C_{out} \times 2$, where C_{out} is the output feature size from the convolutional block
- **Transformer Parameters:**

  - Linear Projection Output Size: $L_{out} = 2 \times D$
  - Multi-Head Attention: 4 heads
  - Feedforward Network Dimension: $4 \times L_{out}$
- **Convolutional Transformer Parameters:**
  - Convolutional Block Output Size: $C_{out} = 2 \times D$
  - Multi-Head Attention: 4 heads
  - Feedforward Network Dimension: $4 \times C_{out}$

4.4 Experiment Results

4.4.1 Half Year Mean Square Error Results

Model	D_1	D_2	D_3
CNN	3.90×10^{-4}	3.7×10^{-4}	8.80×10^{-4}
LSTM	5.81×10^{-6}	6.98×10^{-6}	2.82×10^{-6}
CLSTM	1.48×10^{-6}	1.32×10^{-6}	3.95×10^{-5}
Transformer	1.89×10^{-5}	1.72×10^{-6}	6.4976×10^{-5}
CTransformer	1.61×10^{-5}	1.63×10^{-6}	3.9395×10^{-5}

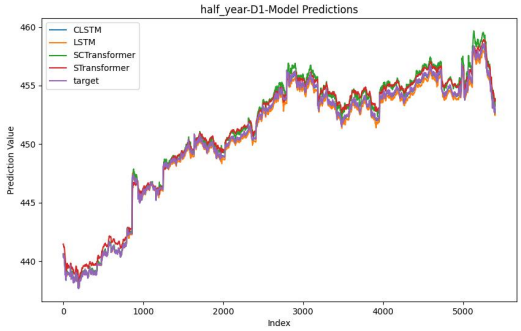
Table 4.1: Half Year Test loss of models on Datasets D1, D2, and D3.

4.4.2 Full Year Mean Square Error Results

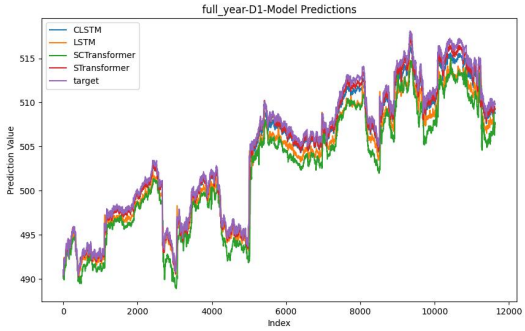
Model	D_1	D_2	D_3
CNN	1.22×10^{-2}	7.92×10^{-4}	2.83×10^{-4}
LSTM	1.83×10^{-4}	1.27×10^{-4}	2.80×10^{-5}
CLSTM	3.90×10^{-5}	1.10×10^{-5}	3.41×10^{-4}
Transformer	1.40×10^{-5}	2.70×10^{-5}	2.00×10^{-5}
CTransformer	3.17×10^{-5}	8.00×10^{-6}	3.00×10^{-6}

Table 4.2: Full Year Test loss of models on Datasets D1, D2, and D3.

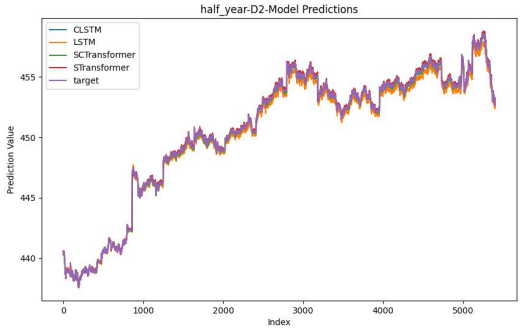
4.4.3 Model Prediction Comparison with different dataset and time interval



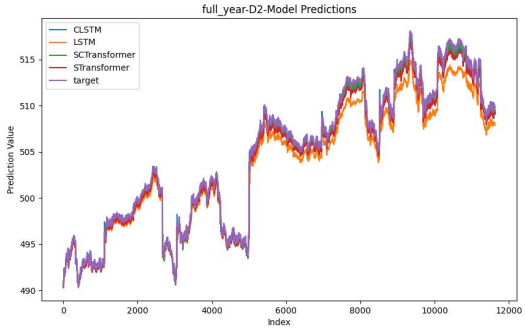
(a) D1-Multi-Model-Predictions



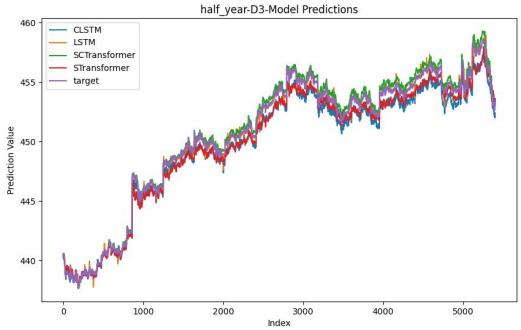
(b) Full-D1-Multi-Model-Predictions



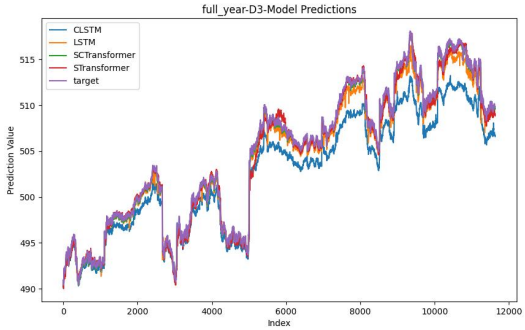
(c) Half-D2-Multi-Model-Predictions



(d) Full-D2-Multi-Model-Predictions



(e) Half-D3-Multi-Model-Predictions



(f) Full-D3-Multi-Model-Predictions

Figure 4.4: Model Performance on D1, D2, and D3

4.5 Discussion

4.5.1 Model Discussion

4.5.1.1 CNN

The half year loss table and full year loss table demonstrates that the CNN architecture is not suitable for handling time series data with long dependencies. The inherent limitations of the convolution operation, particularly its sparse interactions across the receptive fields, makes the architecture's untable to capture long-term temporal structure within the input data.

4.5.1.2 LSTM

LSTM is good at capturing relatively long-term temporal relationships, making a significant improvement over the CNN architecture. As shown in the half year loss table and full year loss table, LSTM demonstrates a notable peroformance improvement compared to the CNN framework.

4.5.1.3 CLSTM

While the CNN architecture may not good in capturing long-term temporal relationships, it is useful at identifying local feature interactions at each timestamp t. By performing convolution operations on input features at each timestamp, the CNN architecture can generate new features, potentially offering a better representation of the original data.

The observed enhancement in performance within D_2 and D_1 can be attributed, in part, to the CNN architecture's feature generation in this dataset, where it achieved more than 50% reduction in loss. We also observe the CLSTM model improved performane over the LSTM model in the result table.

To reduce the risk of the convolutional layer failing to produce valuable features or generating irrelevant information, the filter size is set to double the number of original features. This approach makes sure that, in the worst-case scenario, the original features can at least be identity-mapped to the convolutional output, thereby minimizing potential information loss in the convolutional layer.

4.5.1.4 Transformer

As shown in the half year loss table and full year loss table, the transformer architecture significantly outperforms the LSTM-related models in Full year dataset. Transformer models use self-attention mechanisms that allow for direct connections between distant data in a sequence. This ability to attend to all positions in the sequence simultaneously enables the transformer to model complex dependencies without the constraints of sequential processing. Furthermore, transformers are more scalable than LSTM models. Due to parallel structure in the attention, transformer models can be trained and perform inference at scale.

4.5.1.5 CTransformer

By using a new convolutional layer, jus like CLSTM, the CNN architecture generate new features, potentially offering a better representation of the original data. As shown in the half year loss table and full year loss table, the performance of CTransformer improved compared with the simple Transformer only archietecture. It is worth noting that CTransformer reaches lowest MSE across all models and datasets on full year data.

4.5.2 Dataset Discussion

As shown in the half-year loss table (see half year loss table), the model with the lowest mean square error of 1.32×10^{-6} is the LSTM model trained on dataset D_2. In contrast, for the full-year data (see full year loss table), the model with the lowest mean square error of

3.00×10^{-6} is the CTransformer model trained on dataset D_3.

Model Performance

For the short-term time series data, LSTM-based models outperform the Transformer architecture. LSTM related models perform better because LSTMs capture temporal relations, and their hidden and cell states are enough for modeling short-term time series patterns. However, when predicting long-term data such as a full year, LSTM models struggle to capture long-term dependencies and patterns effectively. Therefore, Transformer related models outperform LSTM models in this task due to their ability to handle long-range dependencies through self-attention mechanisms.

Dataset Differences

The performance of models trained on datasets D_2 and D_3 is better than that of models trained on dataset D_1. This indicates that the usage of technical indicators, sentiment analysis, and macroeconomic analysis contributes to the prediction of stock prices.

For the shorter time span, the best model performance is achieved using dataset D_2, while for the longer time interval, the best performance is obtained using dataset D_3. We can conclude that in the short term, macroeconomic effects and sentiment patterns may not contribute as significantly to stock price prediction. However, over longer periods, these factors provide valuable information, therefore improving the model's performance.

CHAPTER 5

Multiple Steps Prediction

5.1 Problem Formation

In the context of multi-step SPY prediction, our goal is to predict the stock price at the next 10 timestamp using historical data. The input to our model, $\mathbf{X}$, is a three-dimensional tensor with shape of $N \times T \times D$, where N is number of data, T is the number of timestamps, and$D$ is the number of features at each timestamp.

The output of model is $\mathbf{y}$ with shape $N \times 10 \times 1$, and the loss function is mean squared loss, which is defined as following:

$$\text{MSE}(\mathbf{X}, \mathbf{y}) = \frac{1}{N \times 10} \sum_{i=1}^{N} (\mathbf{f}(\mathbf{x}_i) - \mathbf{y_i})^T (\mathbf{f}(\mathbf{x}_i) - \mathbf{y_i})$$

where:

- $\mathbf{f}(\mathbf{x}_i)$ denotes the model's prediction for the i-th input sample $\mathbf{x}_i$, which has shape of $(10, 1)$
- $\mathbf{y_i}$ is the true stock price sequence for the i-th sample, which also has shape of $(10, 1)$

5.2 Model Introduction

5.2.1 CLstmEncoderDecoder

5.2.1.1 CLstm Encoder Decoder

We use an encoder-decoder architecture with ConvLSTM layers to predict sequence data. The encoder encodes the input sequence $\mathbf{X}$ and outputs context vector in the final hidden state of the LSTM encoder. The decoder gets the most recent w inputs, which is $\mathbf{X}_{T-w+1}, \mathbf{X}_{T-w+2}, \ldots, \mathbf{X}_T$. Those inputs form a new input for the LSTM decoder, which has shape of (N, w, D). The LSTM decoder then produces the output sequence by including the previous context vector and iteratively feeding each previously generated value back into the model for the next time step.

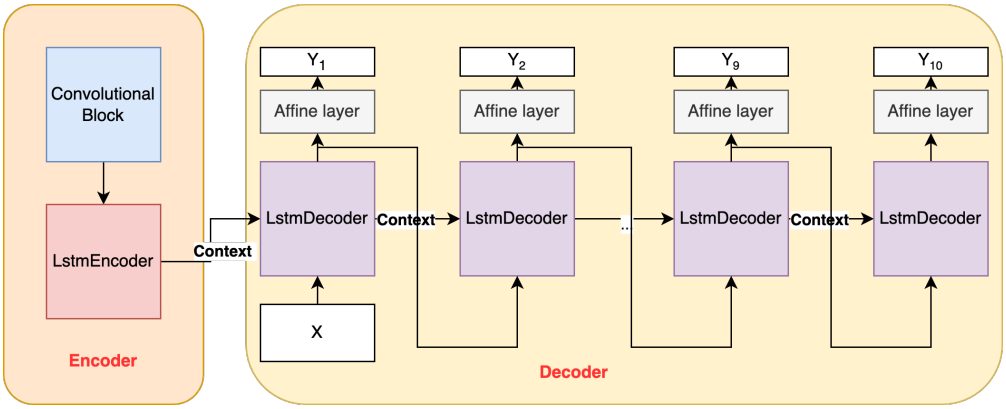


Figure 5.1: LstmEncoderDecoder

5.2.1.2 CLstm Encoder Decoder with Key Padding

Instead of using LstmDecoder output, this model uses the affine layer output, which is the predicted price, and added the padding for the LstmDecoder input to denote that some of the features are missing because predicted features may not be representative of true features like transactions and volume before.

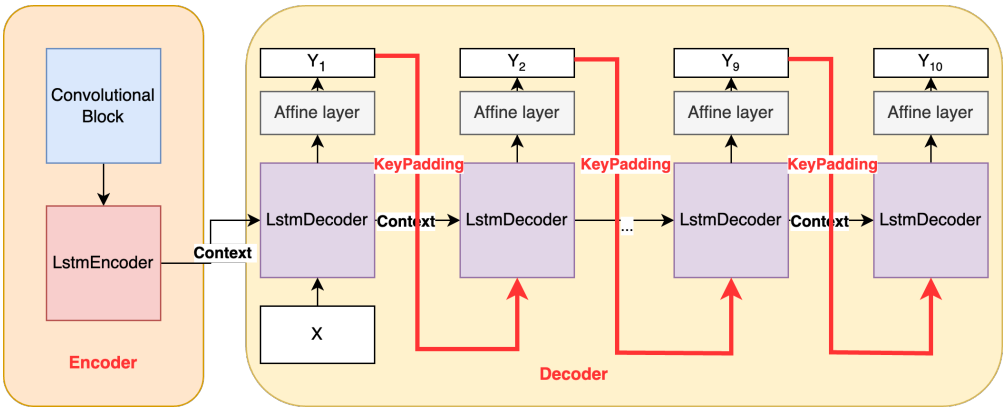


Figure 5.2: LstmEncoderDecoderKeyPadding

5.2.1.3 Lstm Encoder Decoder with Attention

For this variant of ClstmEncoderDecoder, we introduce the Attention mechanism: the Bahdanau Attention Mechanism [BCB14].

Bahdanau Attention Mechanism The Bahdanau Attention Mechanism allows the decoder to focus on relevant parts of the input sequence. For each decoding step t and ith hidden state, the alignment scores $e_{t,i}$ are calculated as:

$$e_{t,i} = \mathbf{v}^\top \tanh(\mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \mathbf{s}_{t-1}), \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_k \exp(e_{t,k})}$$

where $\mathbf{h}_i$ are the encoder hidden states, $\mathbf{s}_{t-1}$ is the decoder hidden state from the previous time step, $\mathbf{W}_1, \mathbf{W}_2, \mathbf{v}$ are learnable parameters, and $\alpha_{t,i}$ is the attention weight.

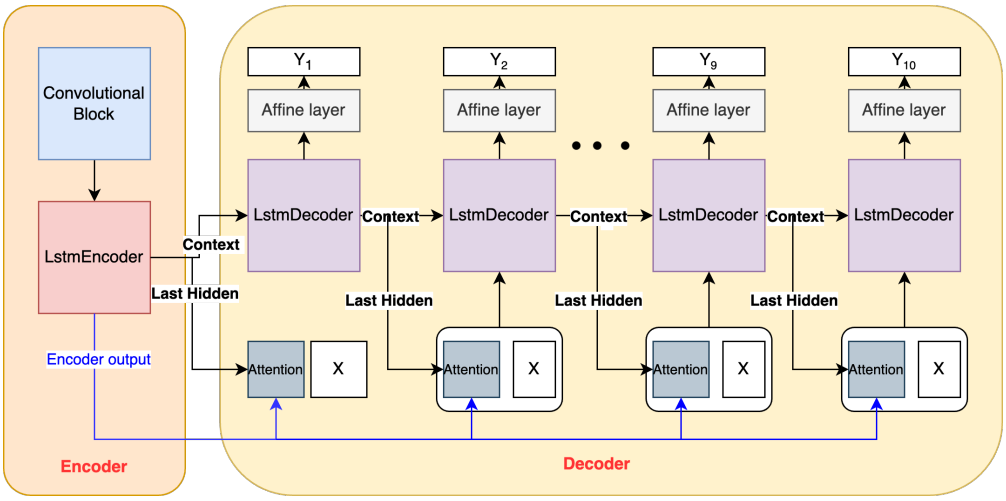


Figure 5.3: CLstmEncoderDecoderAttention

5.2.2 Transformer (Encoder-Decoder Structure)

Positional Encoding In the Transformer architecture, the positional encoding is used to add information about the relative or absolute position of the tokens in the sequence. Since Transformers do not have a built-in notion of order, positional encodings are added to the input embeddings to provide positional information.

The positional encoding is defined as a function of the position pos and the dimension i:

$$PE_{(pos,2t)} = \sin\left(\frac{pos}{10000^{\frac{2t}{d_{\text{model}}}}}\right)$$

$$PE_{(pos,2t+1)} = \cos\left(\frac{pos}{10000^{\frac{2t}{d_{\text{model}}}}}\right)$$

where:

- pos is the position of the token in the sequence.
- t is the dimension along which the positional encoding is computed.

- d_{model} is the dimension of the model (i.e., the size of the input embeddings).

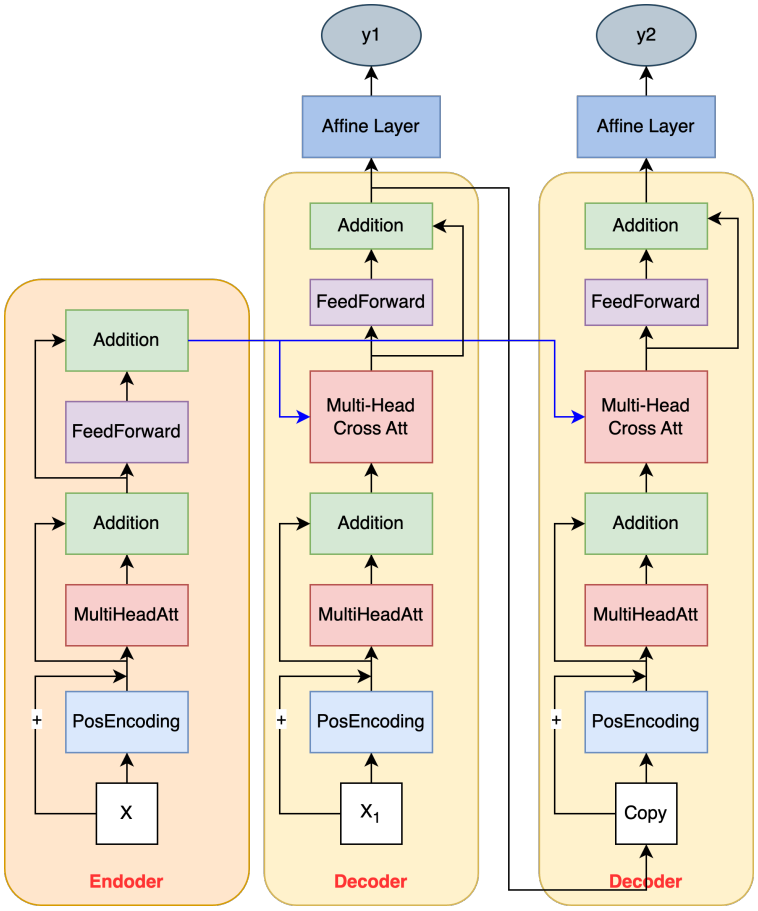


Figure 5.4: TransformerEncoderDecoder

5.3 Training Setup

Data Splits The dataset is partitioned into training, validation, and test sets in an 80:10:10 ratio, respectively. To maintain the temporal dependencies in the data, we split the dataset without shuffling.

Dataset For sequence prediction task, we used dataset D_4 and D_5. For training, we used the adam optimizer with learning rate 10^{-3}

5.4 Half Year Experiment

5.4.1 CLstmEncoderDecoder Half year experiment result

Model Hyperparameter For all of the CLstmEncoderDecoder and its variant models, the Convolutional block filter size is set to be $2 \times D$ and LSTM hidden size is $2 \times F$ where F is the filter size.

- For the CLstmEncoderDecoder, and CLstmEncoderDecoderKeyPadding, we train the models with window size $w = 1, 3, 5$
- For CLstmEncoderDecoderAttention, we train model with hyperparemter mentioned before.

Model	**D4**	**D5**
LstmEncoderDecoderAttention	0.00284	0.00108

Table 5.1: Results for LstmEncoderDecoderAttention Model

w(window size)	**D4-CLstmEncoderDecoder**	**D5-CLstmEncoderDecoder**
1	0.00848	0.00665
3	0.01192	0.00594
5	0.01850	0.01133

Table 5.2: Half Year Results for CLstmEncoderDecoder

w	D4-CLstmEncoderDecoderKeyPadding	D5-CLstmEncoderDecoderKeyPadding
1	0.01502	0.01559
3	0.01859	0.01220
5	0.01169	0.01235

Table 5.3: Half Year Results for CLstmEncoderDecoderKeyPadding

5.4.2 TransformerEncoderDecoder Half year experiment result

Model Hyperparameter For dataset D4, the TransformerEncoderDecoder has filter size of $D \times 2$ and Feedforward Neural network has size of $4 \times f$ where f is the filter size in the convolutional block. For dataset D5, we did not add the Convolutional block because we have already performed the feature enginneering with the discrete fourier transformation.

Dataset	N heads = 2	N heads = 4	N heads = 6
D4	0.00286	0.00026	0.01561

Table 5.4: TransformerEncoderDecoder for D4 with Different Numbers of Heads

Dataest	N heads = 2	N heads = 4	N heads = 7
D5	0.00138	0.00143	0.00009

Table 5.5: TransformerEncoderDecoder for D5 with Different Numbers of Heads

5.5 Full Year Experiment

As we can see in the Half year experiment, in general, models perform better in the dataset in D_5 and TransformerEncoderDecoder and CLstmEncoderDecoderAttention. Therefore, we

extend the time interval to full year with only dataset D_5 and those two models.

Dataset	**N heads = 2**	**N heads = 4**	**N heads = 7**
D5	0.00059922	0.00007840	0.00421713

Table 5.6: TransformerEncoderDecoder for D5 with Different Numbers of Heads

Dataset	**D5**
CLstmEncoderDecoderAttention	0.00284137

Table 5.7: CLstmEncoderDecoderAttention Result for D5

5.6 Discussion

5.6.1 Models Discussion

5.6.1.1 LSTM Encoder-Decoder Related Models

As shown in the CLSTM Encoder-Decoder Experiment Results, the performance of the vanilla CLSTM Encoder-Decoder model decreases as the window size w increases. This demonstrates that the window-based approach to expanding context did not work for this task. Although the CLSTM Encoder-Decoder with key padding shows improved performance at $w = 5$, its overall performance is similar to the vanilla CLSTM Encoder-Decoder. Therefore, we conclude that the window-based LSTM Encoder-Decoder method does not effectively provide sufficient context for sequence prediction.

Furthermore, the CLSTM Encoder-Decoder Experiment Results reveal that the best-performing Lstm related models is the CLSTM Encoder-Decoder with an attention layer with a test loss of 0.00284. By using the attention mechanism, the model can dynamically

use the entire sequence of previous T time steps, assigning weighted importance to each step based on their relevance to the current prediction. This dynamic weighting makes model selectively focus on the most useful parts of the input sequence.

5.6.1.2 Transformer Encoder-Decoder

As shown in the Transformer Encoder-Decoder Half Year Experiment, the model achieves best performance with $n_{heads} = 7$ on dataset D_5, whereas for D_4, the best performance is at $n_{heads} = 4$. This variation shows the transformer's ability to focus on different parts of the input, therefore increasing model performance. Also, the Transformer Encoder-Decoder model has better performance on dataset D_5 compared to dataset D_4.

5.6.1.3 Model Comparison

As demonstrated in the Sequence Full Year Experiment and the Transformer Encoder-Decoder Half Year Experiment, the Transformer Encoder-Decoder model outperforms the CLSTM Encoder-Decoder model on D_4 and D_5. This consistent better performance shows the effectiveness of the Transformer architecture in making sequence prediction.

5.6.2 Dataset Discussion

As shown in the Half Year CLstmEncoderDecoder and Half Year TransformerEncoderDecoder, by feature enginneering with the Fourier transformation to add amplitude and phase, the best performance of TransformerEncoderDecoder and CLstmEncoderDecoderAttention improved on dataset D_5 over the D_4.

CHAPTER 6

Inference

6.1 Feature Importance

We try to determine the feature importance of stock price at time t within the previous time interval $T = 1000$ and quantify how influential those features are at each timestamp t.

As introduced earlier, the input data X has N examples with T timestamps and D features. To measure feature importance at a specific timestamp, we need to quantify their influence.

6.1.1 Feature Importance Calculation

As discussed in Section 2.4.1, we use information gain at each decision tree split to assess feature importance. The importance score for a feature f is calculated as:

$$\text{Importance}(f) = \sum_{s \in S_f} \Delta I_s,$$

where S_f is the set of splits using feature f, and ΔI_s is the gain from split s.

Since the stock prediction loss function is mean square loss, and we use extreme gradient boosting tree model, the gain ΔI_s for a split s is defined as following:

$$\Delta I_s = \frac{1}{2}\left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\right) - \gamma,$$

where:

- G_L, G_R: sums of first-order gradients for the left and right child nodes,
- H_L, H_R: sums of second-order gradients for the left and right child nodes,
- λ: regularization parameter for leaf weights,
- γ: regularization term for the number of leaves.

6.1.2 Experiment Setup

We use the extreme gradient boosting model with the mean squared error loss function, as introduced earlier. We use the full-year dataset D_3 because it contains the largest number of features without Fourier transformation. Since X has the shape (N, T, D) and the input of extreme gradient boosting must be two-dimensional data, we flatten the last two dimensions into a single dimension, thus input has a shape of $(N, T \times D)$.

6.1.3 Experimental Results and Discussion

After training the extreme gradient boosting model, we obtain the following feature importance data:

Table 6.1: Feature Importance Results

Time Index(0-999)	**Feature Description**	**Score**
999	Current Price	6.969564
998	High Price	4.683924
476	EMA Value	4.675079
970	EMA Value	3.289914
999	Low Price	2.602305

As shown in the table, the most important feature is the stock price at the previous timestamp, which makes sense because most traders make decisions based on recent performance. It is worth noting that the EMA (Exponential Moving Average) values also contribute significantly. Compared with the SMA (Simple Moving Average), the EMA is more responsive to recent stock prices. This suggests that recent stock prices contribute more to the $T+1$ stock price, and that older information may not be as useful.

6.2 VaR Inference

To quantify the potential loss in trading the SPY stock, we used Value at Risk (VaR) as a metric to measure the amount of asset value at risk for each individual share. VaR is a risk management tool that estimates the maximum expected loss over a specified time period at a given confidence level. We try to estimate the 5% quantile of the SPY stock price distribution, corresponding to a 95% confidence level.

To calculate VaR, we used the quantile loss function:

$$L_\tau(y, \hat{y}) = \sum_{i=1}^{N} \max\left(\tau(y_i - \hat{y}_i),\, (1-\tau)(\hat{y}_i - y_i)\right),$$

where: y_i is the true value at time t_i, $\hat{y}_i$ is the predicted quantile value at time t_i, τ is the quantile level ($\tau = 0.05$ for 95% quantile), N is the number of observations.

The quantile loss function is used to estimate conditional quantiles by asymmetrically penalizing overestimations and underestimations: it penalizes underestimations (when $y_i > \hat{y}_i$) by a factor of τ and overestimations (when $y_i < \hat{y}_i$) by $1 - \tau$.

Since we use $\tau = 0.05$ quantile, we estimate the threshold below which only 5% of the observed returns fall, effectively capturing the extreme negative returns. This provides a statistical measure of the potential worst-case loss, which is important for risk management and strategic decision-making.

For model selection, we chose Convolutional Long Short-Term Memory (CLSTM) network. For training, we use the Adam optimizer with a learning rate of 10^{-3}, and trained the this model on dataset D_3.

6.2.1 Results

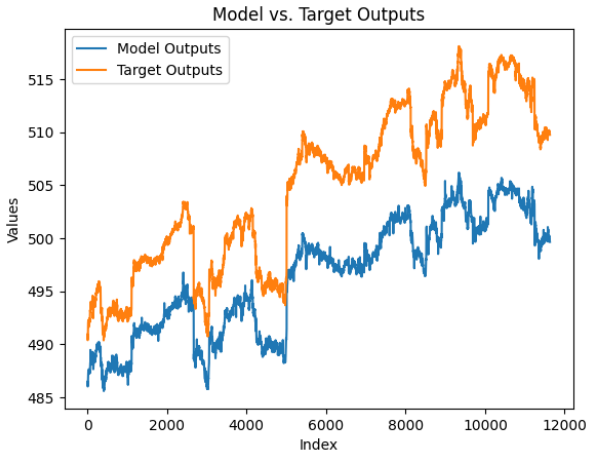


Figure 6.1: CLSTM Model Predicted Quantile

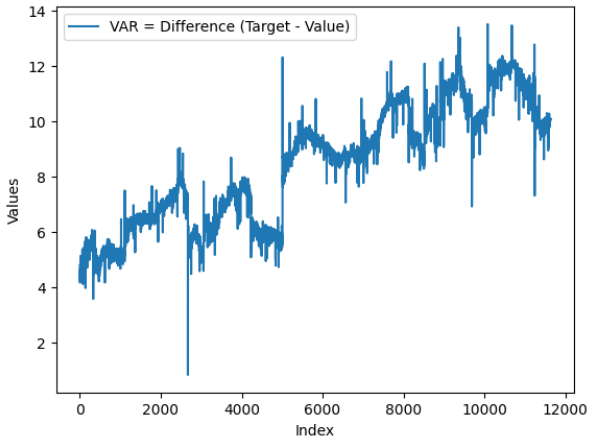


Figure 6.2: Calculated VaR over Time

6.2.2 Discussion

The analysis shows a trend where the Value at Risk (VaR) increases as stock prices of SPY rises. This positive correlation indicates that as the market value of the asset increases, the potential for significant losses also increases.

CHAPTER 7

Discussion

This thesis shows the performances of various deep learning and machine learning models in predicting and drawing inference on stock prices. The experiments confirm that models with complex architectures such as transformers, convolutional LSTMs, and attention mechanisms can outperform relative simple models like LSTMs and CNNs in terms of predictive accuracy, especially when predicting stock prices over longer periods.

7.1 Model Performance

Across both single-step and multi-step prediction tasks, the Tranformer-related models outperformed other architectures most of the time. Their ability to capture long-range dependencies through self-attention mechanisms allowed those models to learn large amount of time-series data more effectively than LSTM related models and CNN. Also, the combination of convolutional layers with LSTMs, as seen in the CLSTM model, provided improvements over basic LSTMs by introducing local feature extraction at individual timestamps before applying temporal processing. This combination of spatial and temporal features contributed to better overall predictions, especially in datasets with technical indicators (D2) and sentiment analysis (D3) for the full year data.

The introduction of attention mechanisms in the CLSTM models further improved their ability to focus on the most relevant past information, dynamically assigning importance to previous timestamps. The addition of attention made significant improvements in multi-step

prediction tasks, where predicting future prices depends on using key historical patterns.

7.2 Dataset Discussion

The technical indicators, sentiment analysis, and phase and amplitude generated by Fourier transformations are useful in improving model performance. For short-term predictions, the LSTM model performed best on datasets with basic technical indicators (D2), as it could capture local temporal dependencies. However, when extending the time period to a full year, the Transformer and CTransformer models trained on dataset D3 (which includes sentiment and macroeconomic features) outperformed other models. This suggests that macroeconomic indicators and market sentiment are important in long-term stock price prediction.

The Fourier-transformed dataset (D5) has additional improvements in performance by including underlying periodic patterns in the data. The use of amplitude and phase information from the Discrete Fourier Transform allowed the models to capture cyclical behavior in stock prices, resulting in better predictions for datasets with longer time intervals.

7.3 Limitations and Future Directions

Although the performance of models are relatively good for stock price prediction, the results suggest that the benefits of sentiment analysis and macroeconomic indicators may be context-dependent. Even through these features improved long-term predictions, they did not significantly improve short-term prediction performance; therefore, their utility varies based on the temporal interval of the analysis.

Furthermore, the performance of various models on sequence prediction dataset are not comparable to the ones in the single-step prediction, suggesting that those models did not learn and capture the underlying structure of the data well.

Future research may focus on further optimizing the models by exploring more advanced

transformer archiecture or hybrid architectures on sequence prediction. Another direction could be to incorporate more granular sentiment analysis, for example, integrating more real-time news sentiment or social media information to capture more precise market sentiment. Additionally, due to the insufficient number of features in our current dataset, we could include other macroeconomic data, such as interest rates or inflation data, to improve long-term forecasting in the future.

REFERENCES

[BCB14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate." *arXiv preprint arXiv:1409.0473*, 2014.

[BCB15] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[BFO84] Leo Breiman, Jerome Friedman, Richard Olshen, and Charles Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, USA, 1984.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[CG16] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*, pp. 785–794, 2016.

[DCL19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186. Association for Computational Linguistics, 2019.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research*, **12**:2121–2159, 2011.

[Elm90] Jeffrey L Elman. "Finding structure in time." *Cognitive science*, **14**(2):179–211, 1990.

[Fri01] Jerome H Friedman. "Greedy function approximation: a gradient boosting machine." *Annals of Statistics*, **29**(5):1189–1232, 2001.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[HS97] Sepp Hochreiter and J"urgen Schmidhuber. "Long short-term memory." *Neural computation*, **9**(8):1735–1780, 1997.

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2nd edition, 2009.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *Proceedings of the 32nd International Conference on Machine Learning*, **37**:448–456, 2015.

[KB15] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization." In *International Conference on Learning Representations*. ICLR, 2015.

[LBB98] Yann LeCun, L'eon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE*, **86**(11):2278–2324, 1998.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature*, **521**(7553):436–444, 2015.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Science & Business Media, 2006.

[Qui86] J. Ross Quinlan. "Induction of Decision Trees." *Machine Learning*, **1**(1):81–106, 1986.

[SHK14] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research*, **15**:1929–1958, 2014.

[TH12] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5—RMSProp: Divide the Gradient by a Running Average of Its Recent Magnitude." Coursera Lecture, 2012. Neural Networks for Machine Learning.

[VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008, 2017.

[YZC23] Kun Yi, Qi Zhang, Longbing Cao, Shoujin Wang, Guodong Long, Liang Hu, Hui He, Zhendong Niu, Wei Fan, and Hui Xiong. "A Survey on Deep Learning based Time Series Analysis with Frequency Transformation." *Journal of the ACM*, **37**(4):111:1–111:15, 2023.