SWINBURNE UNIVERSITY OF TECHNOLOGY

SCHOOL OF SCIENCE, COMPUTING AND ENGINEERING TECHNOLOGIES

---------***--------

COS30018 – Intelligence System

Financial Prediction Using Ensemble Learning and Transfer Learning

Lecturer: Dr. Pham Thi Kim Dung

Group Code: Group 2

Student Name	Student ID
Bui Thanh Thao	104170172
Pham Minh Viet	104848767
Nguyen Dang Huy	103836512
Nguyen Tran Yen Binh	104188492

Due Date: 7th April 2025

Table of Contents

1. INTRODUCTION

The stock market represents one of the most dynamic and complex financial systems in the world, with stock prices reflecting countless variables including company performance, market sentiment, economic indicators, and global events. For many years, traders, investors, and financial institutions have sought to maximize their investment strategies and reduce risks by making accurate predictions about changes in stock prices. This study examines historical stock data from January 1, 2012, to December 12, 2024, to forecast stock values for some of the most significant technological companies in the world, including Apple, Amazon, Microsoft, and Google. In order to identify the distinctive trends and behaviors of each company's market performance, our method entails doing individual stock studies. The study compares several sophisticated time series prediction models, including Long Short-Term Memory networks (LSTM), Gated Recurrent Units (GRU), and Probabilistic Gated Recurrent Units (P_GRU). Furthermore, we present and assess three improved versions of the P_GRU model (1.1, 1.2, and 1.3), which include architectural enhancements intended to improve prediction precision and computational effectiveness while simulating the intricate, non-linear dynamics of stock price fluctuations.

2. RELATED WORKS

2.1. LSTM

Long Short-Term Memory (LSTM) networks represent a specialized architecture of recurrent neural networks (RNNs) designed to address the vanishing gradient problem inherent in traditional RNNs (Hochreiter & Schmidhuber, 1997). LSTMs have proven to be more adept at identifying temporal patterns and long-term dependencies in time series data when it comes to stock market prediction (Fischer & Krauss, 2018). Because of its unique memory cell with input, output, and forget gates, which allows for selective information retention and forgetting, the LSTM architecture is especially well-suited for financial time series forecasting, where past trends have a substantial impact on future price movements. LSTM models have been successfully used in a number of studies to predict stock prices, with lower error rates than conventional neural networks and statistical techniques (Nelson et al., 2017). The temporal aspect of stock market data, which shows both short-term volatility and long-term patterns, is ideally matched with LSTM's capacity to process sequential data of different lengths while preserving information over longer periods. In real-world stock prediction applications, LSTM models have proven their adaptability in managing multivariate time series inputs by successfully incorporating a variety of

features beyond price data, such as technical indicators, trading volume, and even sentiment analysis from financial news. Additionally, these models have demonstrated a strong ability to detect recurrent market trends and regime shifts, both of which are essential for precise forecasting in times of market turbulence (Siami-Namini et al., 2019).

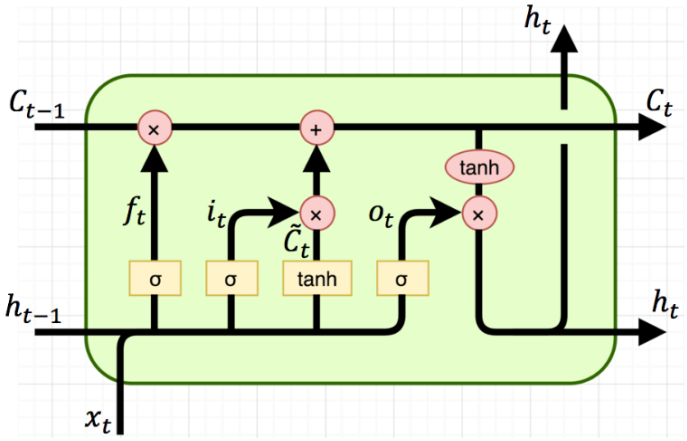


Figure 1. The Architecture of a LSTM cell

2.2. GRU

Gated Recurrent Units (GRU) provide a simplified substitute for LSTM networks while achieving similar results in sequence modeling tasks. GRUs have been popular in stock price prediction because of their ability to handle time series data effectively and their computational efficiency. Compared to LSTM models, the GRU design requires fewer parameters and captures pertinent temporal relationships to its two gates—reset and update gates—that regulate input flow through the network. GRUs are especially well-suited for high-frequency trading applications where computational speed is essential because of their simpler structure, which allows for quicker training without a noticeable loss in predictive accuracy. Comparable accuracy has been reported by empirical investigations comparing GRU and LSTM for stock market forecasting, with GRU models using significantly less processing power (Chung et al., 2014). Additionally, GRU-based models have proven to be resilient to changes in the market regime and robust when dealing with noisy financial data—two prevalent issues in stock price prediction. GRU models have demonstrated remarkable capacity to capture short-term market microstructure patterns and order flow dynamics that influence price formation when used for intraday stock price forecasting. In real-world trading systems where prediction latency is a crucial factor, their smaller computational footprint also makes it possible for more thorough hyperparameter optimization and ensemble techniques, which improve forecasting performance (Fischer & Krauss, 2018).

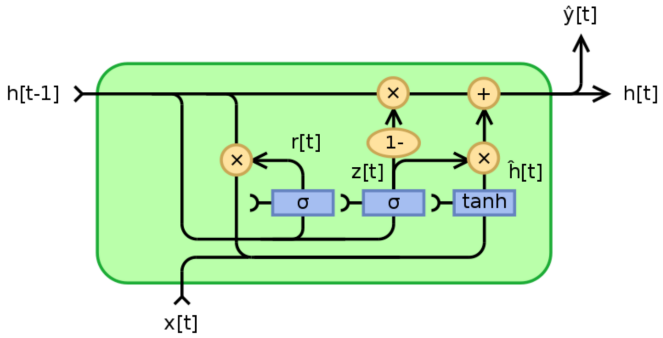


Figure 2. The Architecture of a GRU cell

2.3. TRANSFER LEARNING

Transfer learning creates high-performance models using data from different domains when training data is scarce. This method builds on the foundations of a pre-trained model by utilizing knowledge acquired from one task to improve performance in adjacent activities. The method improves accuracy while drastically lowering the amount of data needed. It is possible to modify stock prediction algorithms that were developed for well-established markets to fit new markets with less historical data. Significant increases in forecast accuracy are demonstrated by this cross-market information transfer. By enabling models to learn broad market behaviors from industry leaders before fine-tuning for particular companies with comparable business models but less trading experience, transfer learning has completely changed sector-based stock research.

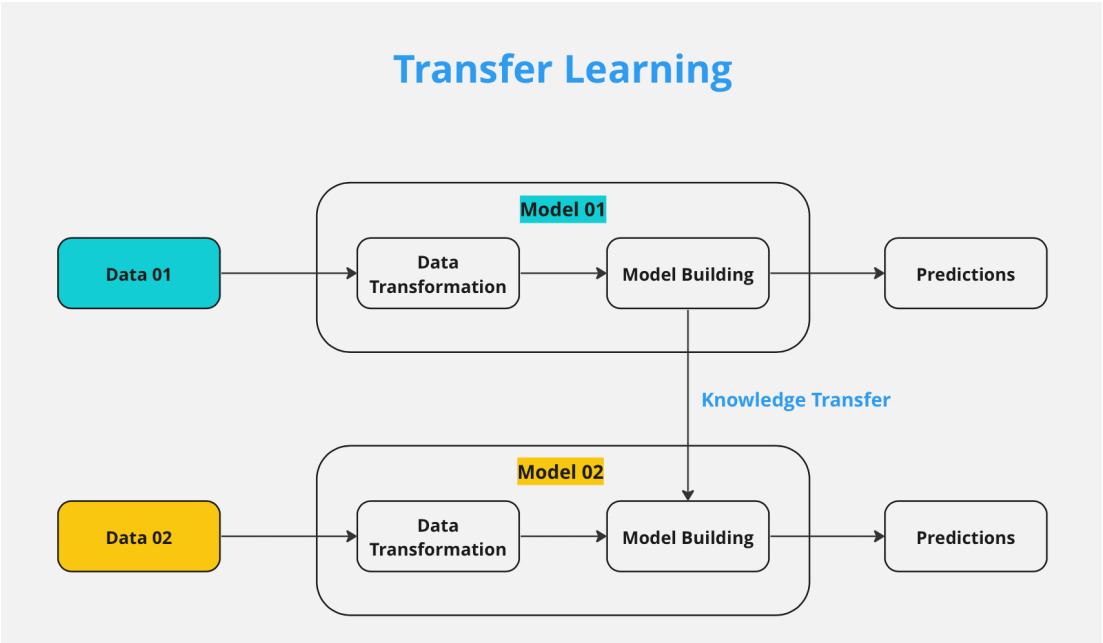


Figure 3. Transfer Learning

2.4. TENSORFLOW PROBABILITY

TensorFlow Probability (TFP) represents a specialized library within the TensorFlow ecosystem dedicated to probabilistic reasoning and statistical analysis (Dillon et al., 2017). TFP provides a framework for creating models for stock price prediction that specifically take uncertainty into consideration. Deterministic techniques sometimes ignore this essential component of financial markets. Bayesian neural networks and probabilistic state space models that capture the stochastic character of stock price fluctuations can be implemented efficiently thanks to TFP. Models can forecast whole probability distributions rather than just single point estimates thanks to its support for distributional outputs, which offers insights into potential future events and the hazards that go along with them.

Recent applications of TFP in financial markets have demonstrated effectiveness in quantifying both inherent market randomness and model limitations (Cheng et al., 2020), enabling more sophisticated risk assessment and decision-making in quantitative trading strategies.

3. DATASETS AND FEATURES

3.1. THE TRAINING DATASET

We used Yahoo Finance's API (https://yfinance-python.org/) to download the cryptocurrency data of Bitcoin. The downloaded data has the Date column and the Open, High, Low, Close, and Volume (OHLCV) columns. For this training process, we use OHLCV columns of Bitcoin prices, and the Date column is as the index of the data.

Open	High	Low	Close	Volume
99271.710938	99302.578125	99089.382812	99089.382812	0
98857.171875	99131.789062	98857.171875	99054.585938	162512896
98476.664062	98893.039062	98455.226562	98893.039062	273063936
98559.101562	98559.101562	98368.984375	98498.945312	288464896
98652.921875	98652.921875	98441.812500	98640.492188	271204352
...	...	...	...	...
83421.445312	83421.445312	83157.210938	83157.210938	161677312
82786.664062	82893.367188	82697.546875	82893.367188	157683712
82964.664062	83003.562500	82571.101562	82708.914062	1288732672
83099.585938	83099.585938	82754.781250	82889.679688	456544256
83131.296875	83162.945312	83131.296875	83162.945312	0

Figure 4. A preview of some rows in the downloaded Bitcoin data

```
data.describe()
```

	Open	High	Low	Close	Volume
count	17280.000000	17280.000000	17280.000000	17280.000000	1.728000e+04
mean	89243.210209	89293.345669	89193.356845	89243.743005	1.771284e+08
std	6176.240373	6171.766816	6181.227769	6177.385377	4.013057e+08
min	76901.398438	76985.281250	76653.531250	76808.101562	0.000000e+00
25%	83967.673828	84005.693359	83920.695312	83966.240234	0.000000e+00
50%	87004.476562	87065.265625	86950.460938	87007.718750	0.000000e+00
75%	96222.101562	96254.753906	96192.654297	96226.746094	1.543068e+08
max	102514.171875	102514.171875	102247.468750	102420.531250	8.659169e+09

Figure 5. Statistical description of the downloaded Bitcoin dataset

3.2. STOCK DATASET FOR TRANSFER LEARNING

For the transfer learning, we downloaded the stock data of the big 4 tech firms, including Apple, Amazon, Google, and Microsoft via yfinance. Due to the scope of our project, which is predicted based on historical stock prices and prediction for the short-term, we decided to use daily time-series stock data, which are known as daily OHLCV stock data, which includes the Open, High, Low, Close, and Volume columns for each day. We downloaded the data from 01-01-2012 to 12-12-2024 for each company. In this project, we will try split the data by 2 different dates. In the first experiment, we will try to train on stock data from 2012 to 2023 and predict data in 2024. In the second experiment, we only train on stock data from 2012 to 2022, and the test data will be 2023 and 2024 data.

This is the explanation of 6 columns in the dataset:

- **Date**: The date the stock data was recorded.
- **Close**: The closing price of the stock for the day.
- **High**: The highest price the stock reached during the day.
- **Low**: The lowest price the stock reached during the day.
- **Open**: The opening price of the stock for the day.
- **Volume**: The total number of shares traded during the day.

Date	Close	High	Low	Open	Volume
2012-01-03	16.591755	16.660076	16.266609	16.280821	146912940
2012-01-04	16.663319	16.712440	16.472319	16.582279	114445440
2012-01-05	16.432173	16.555849	16.362855	16.509970	131184684
2012-01-06	16.208012	16.456860	16.202276	16.435664	107608284
2012-01-09	15.520813	16.132709	15.490143	16.120242	232671096
...	...	...	...	...	...
2024-12-05	172.442368	175.858451	172.132725	175.159255	21356200
2024-12-06	174.510010	174.879581	171.663266	171.833070	21462400
2024-12-09	175.369995	176.259995	173.649994	173.960007	25389600
2024-12-10	185.169998	186.360001	181.050003	182.850006	54813000
2024-12-11	195.399994	195.610001	184.850006	185.309998	67894100

3257 rows × 5 columns

Figure 6. A preview of some rows in the downloaded Google data

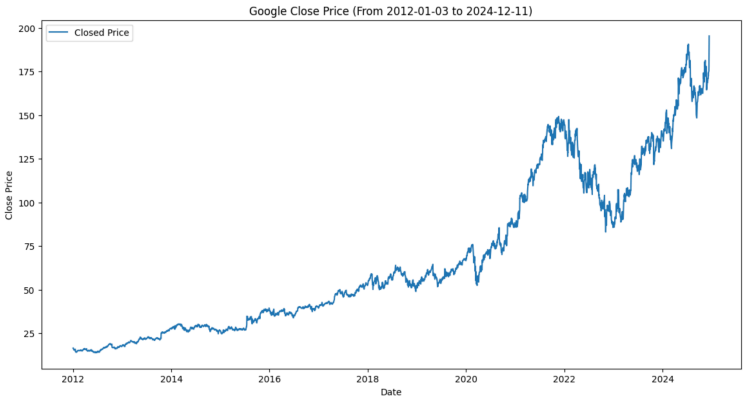


Figure 7. Visualisation of Close Price column in the dataset

4. METHODOLOGY

In this project, we used the **Ensemble Learning and Transfer Learning methods** to predict the stock prices of four major technology firms, including Apple, Amazon, Google, and Microsoft. Specifically, we applied a new method using probabilistic gated recurrent units (**P-GRU**), as discussed in the paper by Golnari, Komeili, and Azizi (2024). This method adds probabilistic features to the model, allowing for the creation of probability distributions for the predicted values. We also compared this method with other models, such as **GRU**, long short-term memory (**LSTM**), and various variations of these models (including **simple, time-distributed, and bidirectional models**). To enhance the model's effectiveness, we employed a custom **callback function that tracks the R2-score to identify the best model weights** based on validation data. Additionally, we used a **transfer learning** approach, applying a pre-trained BTC (Bitcoin) model **to predict the stock price** of one of the four tech companies mentioned above. As a result, a separate model was created for each company's stock price. Finally, to enhance the model performance, we have **tried the 3 cases for feature engineering** for this model, including adding technical indicators, using graph features, and combining 2 methods.

The remainder of Section 4 will discuss our method in more detail, following the order in which we approached this project. There are **two main periods** in our process: the first is **Ensemble Learning**, and the second is **Transfer Learning**.

4.1. ENSEMBLE LEARNING

4.1.1. Data Preprocessing

In the first period, we downloaded the BTC cryptocurrency data using yfinance, as mentioned in Section 3. After downloading the data, we plotted some graphs to visualize it. Then, we proceeded to the Data Preprocessing step.

4.1.1.1. Normalization

Instead of using the usual Min-Max Scaler for the financial price prediction problem, we used a more advanced formula, which is an **enhanced version of the Min-Max Scaler** that **includes the parameters α and β** .

$$x_{norm} = \frac{x - (\alpha \times x_{min})}{(\beta \times x_{max}) - (\alpha \times x_{min})} \text{ with } 0 < \alpha \leq 1, 1 \leq \beta$$

In this formula, x represents the raw data, x_{min} is the minimum value, and x_{max} is the maximum value. The parameters α and β play important roles in the normalization process. α helps adjust the data by shifting it within a certain range, while β controls how much the data is scaled. By modifying these values, the data's range and distribution can be fine-tuned to fit the needs of the machine learning models.

The purpose of using α and β in normalization is to ensure that no single feature overwhelms the model's predictions due to differences in the data's scale (e.g., large spikes or small fluctuations). Additionally, choosing appropriate values for α and β helps manage extreme values and fluctuations in the data, allowing the model to learn effectively without being unduly affected by outliers or periods of high volatility.

4.1.1.2. Data Splitting

After the cryptocurrency data is normalized, it is split into three parts: training (60%), validation (20%), and testing (20%). This division helps the model learn from one set of data during training while also checking its performance on new, unseen data through the validation and test sets. The training set is used to teach the model, while the validation set helps adjust settings, prevents overfitting, and measures the model's performance using a custom R2-score callback. During each training cycle, the model's accuracy is tracked with this callback, and the best model weights are selected. This step provides a reliable way to measure how well the model can generalize. Finally, the test set is used to check how the model performs on completely new data, giving a solid indication of how well it can generalize.

4.1.2. Models

For this Ensemble Learning period, we trained 12 deep learning models, including probabilistic ones, with various architectures using BTC price data. These 12 models include both GRU and LSTM variants such as GRU, LSTM, time-distributed, bidirectional, probabilistic, and simple models. After training those 12 models, we utilized the best-performing model as a pre-trained model.

Models Evaluation Results

```
df = pd.DataFrame.from_dict(models_test_result, orient = 'index')
df
```

	r2_score	mean_absolute_percentage_error	explained_variance_score
best_BTCPred_bi_gru_prob	0.941158	0.005342	0.996794
best_BTCPred_bi_gru_simple	0.996185	0.001050	0.996505
best_BTCPred_bi_gru_time_dist	0.992929	0.001396	0.993473
best_BTCPred_gru_prob	0.986427	0.002301	0.995893
best_BTCPred_gru_simple	0.992489	0.001623	0.996168
best_BTCPred_gru_time_dist	0.981016	0.002452	0.986008
best_BTCPred_bi_lstm_prob	0.772377	0.010700	0.996598
best_BTCPred_bi_lstm_simple	0.995959	0.001060	0.996038
best_BTCPred_bi_lstm_time_dist	0.989632	0.001787	0.992039
best_BTCPred_lstm_prob	0.991640	0.001763	0.996754
best_BTCPred_lstm_simple	0.992021	0.001653	0.995452
best_BTCPred_lstm_time_dist	0.988667	0.001852	0.990504

Figure 8. Evaluation Results of 12 models in the Ensemble Learning phase

To add probabilistic features to the GRU and LSTM models, we **integrate TensorFlow Probability** into their structure. This integration allows us to use the probabilistic approach provided by TensorFlow Probability, capturing the uncertainties in the model's predictions. **These probabilistic models use the negative log-likelihood as their loss function**, while the **other GRU and LSTM models use mean squared error (MSE)**, which measures the average squared difference between the predicted and actual target values.

In the following sub-parts, we will go through the structure of 12 models.

4.1.2.1. GRU (gru_simple)

For this GRU model, we **constructed a Simple Gated Recurrent Unit (GRU) model with Regularization and Batch Normalization**. In detail, it uses a Gated Recurrent Unit (GRU) layer with 256 units and applies L1 regularization to prevent overfitting by penalizing large weights. The model also includes batch normalization after each layer to stabilize training by normalizing the activations. Following the GRU layer, a dense layer with 64 units and a sigmoid activation function is applied, followed by another batch normalization layer. The final output layer consists of a single unit with a sigmoid activation function, providing a probability score for predictions. The model is compiled using the RMSprop optimizer with a learning rate of 1e-3 and uses mean absolute error (MAE) as the loss function to assess the model's accuracy.

Simple GRU Model with Regularization and Batch Normalization

```
[ ] def gru_simple(shape = None, name = None):
        """
        Description:
        - Construct a Simple Gated Recurrent Unit (GRU) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Simple GRU model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg)(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)
        h = Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'gru_simple_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 9. Code of the simple GRU model

```
Model: "gru_simple_btcusd"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 close_price (InputLayer)    [(None, 6, 1)]            0

 gru (GRU)                   (None, 256)               198912

 batch_normalization (BatchNormalization)  (None, 256)  1024

 dense (Dense)               (None, 64)                16448

 batch_normalization_1 (BatchNormalization)  (None, 64)  256

 dense_1 (Dense)             (None, 1)                 65

 batch_normalization_2 (BatchNormalization)  (None, 1)  4

 activation (Activation)     (None, 1)                 0

=================================================================
Total params: 216709 (846.52 KB)
Trainable params: 216067 (844.01 KB)
Non-trainable params: 642 (2.51 KB)
```

Figure 10. Summary of the simple GRU model

4.1.2.2. Bidirectional GRU (bi_gru_simple)

For this Bidirectional GRU model, we constructed a Bidirectional Simple Gated Recurrent Unit (GRU) model with regularization and batch normalization. The code is similar to the simple GRU model. **However**, **instead of using the regular GRU layer "GRU()" like the basic GRU model, this bidirectional GRU uses "Bidirectional(GRU())"**, which processes the input data in both forward and backward directions. Being bidirectional, this model has the potential to better capture information from both past and future time steps in sequential data.

Bidirectional Simple GRU Model with Regularization and Batch Normalization

```
[ ]  def bi_gru_simple(shape = None, name = None):
         """
         Description:
         - Construct a Bidirectional Simple Gated Recurrent Unit (GRU) model with regularization and batch normalization.

         Parameters:
         - shape (tuple): Input shape for the model.
         - name (str): Name to identify the model.

         Returns:
         - tf.keras.Model: Compiled Bidirectional Simple GRU model.
         """
         # Set up L1 regularization
         rg = tf.keras.regularizers.l1(l1 = 1e-3)

         # Define model architecture
         I = Input(shape = shape, name = 'close_price')
         h = Bidirectional(GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg))(I)
         h = BatchNormalization()(h)
         h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
         h = BatchNormalization()(h)
         h = Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg)(h)
         h = BatchNormalization()(h)
         O = Activation('sigmoid')(h)

         # Compile the model
         m = Model(inputs = I, outputs = O, name = f'bi_gru_simple_{name}')
         m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
         return m
```

Figure 11. Code of the Bidirectional simple GRU model

Model: "bi_gru_simple_btcusd"

Layer (type)	Output Shape	Param #
close_price (InputLayer)	[(None, 6, 1)]	0
bidirectional (Bidirectional)	(None, 512)	397824
batch_normalization (BatchNormalization)	(None, 512)	2048
dense (Dense)	(None, 64)	32832
batch_normalization_1 (BatchNormalization)	(None, 64)	256
dense_1 (Dense)	(None, 1)	65
batch_normalization_2 (BatchNormalization)	(None, 1)	4
activation (Activation)	(None, 1)	0

Total params: 433029 (1.65 MB)
Trainable params: 431875 (1.65 MB)
Non-trainable params: 1154 (4.51 KB)

Figure 12. Summary of the Bidirectional simple GRU model

4.1.2.3. Probabilistic GRU (gru_prob)

For this Probabilistic GRU model, we built upon the GRU Simple model by introducing a probabilistic output. The **model architecture is similar to the gru_simple model**, with the inclusion of a GRU layer with 256 units, L1 regularization, and batch normalization. **However**, instead of producing a single predicted value, **the output is a probabilistic distribution**. This is achieved using DistributionLambda, which generates a Normal distribution where the mean (loc) is derived from the model's predictions and the scale is calculated using a softplus function applied to the predicted values. This change allows the model to provide uncertainty estimates along with predictions. Additionally, the **gru_prob model uses a custom loss function based on the negative log-likelihood of the predicted distribution**, in contrast to the mean absolute error used in the gru_simple model.

Probabilistic GRU Model with Regularization and Batch Normalization

```
[ ] def gru_prob(shape = None, name = None):
        """
        Description:
        - Construct a Probabilistic Gated Recurrent Unit (GRU) model with regularization, batch normalization,
        and distribution output. The distribution is defined as a Normal distribution with the predicted values (loc)
        and a scale parameter derived from a softplus function applied to the predicted values. This approach is employed
        for probabilistic predictions in the context of the model.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Probabilistic GRU model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg)(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)

        # Output a probabilistic distribution using DistributionLambda
        O = DistributionLambda(lambda t: Normal(loc = t, scale = 1e-2 * tf.math.softplus(1e-2 * t), validate_args = True, allow_nan_stats = False))(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'gru_prob_{name}')
        m.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3), loss = lambda y, p: -p.log_prob(y), metrics = None)
        return m
```

Figure 13. Code of the Probabilistic GRU model

4.1.2.4. Bidirectional Probabilistic GRU (bi_gru_prob)

In this Bidirectional Probabilistic GRU model, the architecture is still the same as the above gru_prob architecture. We just **changed a little bit by using a Bidirectional wrapper around the GRU layer**, allowing it to process input data in both forward and backward directions. This enables the model to capture information from both past and future time steps, improving its ability to understand temporal dependencies compared to the unidirectional GRU Probabilistic model.

Bidirectional Probabilistic GRU

```
[ ] def bi_gru_prob(shape = None, name = None):
        """
        Description:
        - Construct a Bidirectional Probabilistic Gated Recurrent Unit (GRU) model with regularization, batch normalization,
        and distribution output. The distribution is defined as a Normal distribution with the predicted values (loc)
        and a scale parameter derived from a softplus function applied to the predicted values. This approach is employed
        for probabilistic predictions in the context of the model.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Bidirectional Probabilistic GRU model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = Bidirectional(GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg))(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)

        # Output a probabilistic distribution using DistributionLambda
        O = DistributionLambda(lambda t: Normal(loc = t, scale = 1e-2 * tf.math.softplus(1e-2 * t), validate_args = True, allow_nan_stats = False))(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'bi_gru_prob_{name}')
        m.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3), loss = lambda y, p: -p.log_prob(y), metrics = None)
        return m
```

Figure 14. Code of the Bidirectional Probabilistic GRU model

```
Model: "bi_gru_prob_btcusd"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 close_price (InputLayer)    [(None, 6, 1)]            0

 bidirectional (Bidirectional)  (None, 512)            397824

 batch_normalization (BatchNormalization)  (None, 512)  2048

 dense (Dense)               (None, 64)                32832

 batch_normalization_1 (BatchNormalization)  (None, 64)  256

 distribution_lambda (DistributionLambda)  ((None, 64),
                                            (None, 64))  0

=================================================================
Total params: 432960 (1.65 MB)
Trainable params: 431808 (1.65 MB)
Non-trainable params: 1152 (4.50 KB)
_________________________________________________________________
```

Figure 15. Summary of the Bidirectional Probabilistic GRU model

4.1.2.5. Time Distributed GRU (gru_time_dist)

We used the **same architecture as the Simple GRU model**. **However**, we made several changes in the Time Distributed GRU model. First, we **set the GRU layer with return_sequences=True**, allowing it to output sequences rather than just the final state. To process each time step independently, we **applied the TimeDistributed wrapper to the Dense layers**. This enables the model to handle sequential data more effectively by making predictions at each time step. Additionally, we **added a Flatten layer** to reshape the output **before** applying the **final sigmoid activation**. **Unlike** the Simple GRU model, which uses a **single Dense layer for the final output**, the **Time Distributed GRU** model

includes multiple time-distributed Dense layers, along with **batch normalization** applied throughout the model. These modifications allow the model to better handle sequential tasks.

Time Distributed GRU Model with Regularization and Batch Normalization

```
[ ] def gru_time_dist(shape = None, name = None):
        """
        Description:
        - Construct a Time Distributed Gated Recurrent Unit (GRU) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Time Distributed GRU model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg, return_sequences = True)(I)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = Flatten()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'gru_time_dist_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 16. Code of the Time Distributed GRU model

4.1.2.6. Bidirectional Time Distributed GRU (bi_gru_time_dist)

Similar to other Bidirectional versions, we used the **same architecture with the original model (gru_time_dist)** and applied a Bidirectional GRU layer by **adding a Bidirectional wrapper around the GRU layer**, which allows the model to process the input data in both forward and backward directions, capturing context from both past and future time steps.

Bidirectional Time Distributed GRU Model with Regularization and Batch Normalization

```
[ ] def bi_gru_time_dist(shape = None, name = None):
        """
        Description:
        - Construct a Bidirectional Time Distributed Gated Recurrent Unit (GRU) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Bidirectional Time Distributed GRU model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = Bidirectional(GRU(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg, return_sequences = True))(I)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = Flatten()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'bi_gru_time_dist_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 17. Code of the Bidirectional Time Distributed GRU model

4.1.2.7. LSTM (lstm_simple)

For this LSTM model, we constructed a simple Long Short-Term Memory (LSTM) model with Regularization and Batch Normalization. In detail, this model is built with Long Short-Term Memory (LSTM) layers for handling sequential data. The model starts with an **LSTM layer** containing 256 units, followed by L1 regularization and batch normalization to prevent overfitting and ensure better generalization. After the LSTM layer, a **Dense layer** with 64 units and a **sigmoid activation function** is applied, followed by another batch normalization layer. A final **Dense layer** with 1 unit is used to produce the output, and batch normalization is applied once more before the final **sigmoid activation**. The model is compiled using the **RMSprop optimizer** with a learning rate of 1e-3 and **mean absolute error (MAE)** as the loss function. This architecture is designed to process time-series data effectively while preventing overfitting through regularization and batch normalization.

Simple LSTM Model with Regularization and Batch Normalization

```
[ ]  def lstm_simple(shape = None, name = None):
         """
         Description:
         - Construct a simple Long Short-Term Memory (LSTM) model with regularization and batch normalization.

         Parameters:
         - shape (tuple): Input shape for the model.
         - name (str): Name to identify the model.

         Returns:
         - tf.keras.Model: Compiled Simple LSTM model.
         """
         # Set up L1 regularization
         rg = tf.keras.regularizers.l1(l1 = 1e-3)

         # Define model architecture
         I = Input(shape = shape, name = 'close_price')
         h = LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg)(I)
         h = BatchNormalization()(h)
         h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
         h = BatchNormalization()(h)
         h = Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg)(h)
         h = BatchNormalization()(h)
         O = Activation('sigmoid')(h)

         # Compile the model
         m = Model(inputs = I, outputs = O, name = f'lstm_simple_{name}')
         m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
         return m
```

Figure 18. Code of the simple LSTM model

```
Model: "lstm_simple_btcusd"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 close_price (InputLayer)    [(None, 6, 1)]            0

 lstm (LSTM)                 (None, 256)               264192

 batch_normalization (BatchNormalization)  (None, 256)  1024

 dense (Dense)               (None, 64)                16448

 batch_normalization_1 (BatchNormalization)  (None, 64)  256

 dense_1 (Dense)             (None, 1)                 65

 batch_normalization_2 (BatchNormalization)  (None, 1)  4

 activation (Activation)     (None, 1)                 0

=================================================================
Total params: 281989 (1.08 MB)
Trainable params: 281347 (1.07 MB)
Non-trainable params: 642 (2.51 KB)
```

Figure 19. Summary of the simple LSTM model

4.1.2.8. Other 5 variants of LSTM model: Bidirectional LSTM (bi_lstm_simple), Probabilistic LSTM (lstm_prob), Bidirectional Probabilistic LSTM (bi_lstm_prob), Time Distributed LSTM (lstm_time_dist), and Bidirectional Time Distributed LSTM (bi_lstm_time_dist)

The methods we used to modify the structure of the lstm_simple model to create the other 5 variants of the LSTM models are the same as those used to create the 5 variants of the GRU model. You can refer to the previous sections to see the explanation of how each variant was created.

Bidirectional Simple LSTM Model with Regularization and Batch Normalization

```
[ ] def bi_lstm_simple(shape = None, name = None):
        """
        Description:
        - Construct a Bidirectional Simple Long Short-Term Memory (LSTM) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Bidirectional Simple LSTM model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = Bidirectional(LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg))(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)
        h = Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'bi_lstm_simple_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 20. Code of the Bidirectional simple LSTM model

Probabilistic LSTM Model with Regularization and Batch Normalization

```
[ ] def lstm_prob(shape = None, name = None):
        """
        Description:
        - Construct a Probabilistic Long Short-Term Memory (LSTM) model with regularization, batch normalization, and distribution output.
        The distribution is defined as a Normal distribution with the predicted values (loc) and a scale parameter derived from a softplus
        function applied to the predicted values. This approach is employed for probabilistic predictions in the context of the model.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Probabilistic LSTM model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg)(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)

        # Output a probabilistic distribution using DistributionLambda
        O = DistributionLambda(lambda t: Normal(loc = t, scale = 1e-2 * tf.math.softplus(1e-2 * t), validate_args = True, allow_nan_stats = False))(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'lstm_prob_{name}')
        m.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3), loss = lambda y, p: -p.log_prob(y), metrics = None)
        return m
```

Figure 21. Code of the Probabilistic LSTM model

Bidirectional Probabilistic LSTM Model with Regularization and Batch Normalization

```
[ ] def bi_lstm_prob(shape = None, name = None):
        """
        Description:
        - Construct a Bidirectional Probabilistic Long Short-Term Memory (LSTM) model with regularization, batch normalization,
        and distribution output. The distribution is defined as a Normal distribution with the predicted values (loc)
        and a scale parameter derived from a softplus function applied to the predicted values. This approach is employed
        for probabilistic predictions in the context of the model.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Bidirectional Probabilistic LSTM model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = Bidirectional(LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg))(I)
        h = BatchNormalization()(h)
        h = Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg)(h)
        h = BatchNormalization()(h)

        # Output a probabilistic distribution using DistributionLambda
        O = DistributionLambda(lambda t: Normal(loc = t, scale = 1e-2 * tf.math.softplus(1e-2 * t), validate_args = True, allow_nan_stats = False))(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'bi_lstm_prob_{name}')
        m.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3), loss = lambda y, p: -p.log_prob(y), metrics = None)
        return m
```

Figure 22. Code of the Bidirectional Probabilistic LSTM model

Time Distributed LSTM Model with Regularization and Batch Normalization

```
[ ]  def lstm_time_dist(shape = None, name = None):
        """
        Description:
        - Construct a Time Distributed Long Short-Term Memory (LSTM) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Time Distributed LSTM model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg, return_sequences = True)(I)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = Flatten()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'lstm_time_dist_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 23. Code of the Time Distributed LSTM model

Bidirectional Time Distributed LSTM Model with Regularization and Batch Normalization

```
[ ]  def bi_lstm_time_dist(shape = None, name = None):
        """
        Description:
        - Construct a Bidirectional Time Distributed Long Short-Term Memory (LSTM) model with regularization and batch normalization.

        Parameters:
        - shape (tuple): Input shape for the model.
        - name (str): Name to identify the model.

        Returns:
        - tf.keras.Model: Compiled Bidirectional Time Distributed LSTM model.
        """
        # Set up L1 regularization
        rg = tf.keras.regularizers.l1(l1 = 1e-3)

        # Define model architecture
        I = Input(shape = shape, name = 'close_price')
        h = Bidirectional(LSTM(units = 256, kernel_regularizer = rg, bias_regularizer = rg, recurrent_regularizer = rg, return_sequences = True))(I)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 64, activation = 'sigmoid', kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = TimeDistributed(Dense(units = 1, kernel_regularizer = rg, bias_regularizer = rg))(h)
        h = BatchNormalization()(h)
        h = Flatten()(h)
        O = Activation('sigmoid')(h)

        # Compile the model
        m = Model(inputs = I, outputs = O, name = f'bi_lstm_time_dist_{name}')
        m.compile(optimizer = tf.keras.optimizers.RMSprop(learning_rate = 1e-3), loss = 'mae', metrics = None)
        return m
```

Figure 24. Code of the Bidirectional Time Distributed LSTM model

4.1.3. Optimizing performance using R2-score tracking

To improve the model's performance, a custom callback was added that tracks the R2-score. This callback monitors the training process, tracks the R^2 score, and saves the best model based on R^2 improvement on the validation dataset. If the R^2-score is higher than the previously observed best, the model's weights are saved. This callback successfully identified the best model weights based on validation data, helping to avoid overfitting and improve the model's ability to generalize to new unseen data besides capturing the most appropriate model weights. Using this callback made the models predictions more accurate and dependable.

```
Epoch: 01/20 | 0:00:24 | Price R2 score: 0.0512 | loss: 1395.3964 | Model Weights Changed And Best Model Saved
87/87 [==============================] - 1s 6ms/step
Epoch: 02/20 | 0:00:07 | Price R2 score: 0.2459 | loss: 192.1724 | Model Weights Changed And Best Model Saved
87/87 [==============================] - 0s 6ms/step
Epoch: 03/20 | 0:00:07 | Price R2 score: -12.8226 | loss: 20.8850 | Model Weights Not Changed
87/87 [==============================] - 1s 9ms/step
Epoch: 04/20 | 0:00:08 | Price R2 score: 0.9586 | loss: 7.4593 | Model Weights Changed And Best Model Saved
87/87 [==============================] - 0s 5ms/step
Epoch: 05/20 | 0:00:07 | Price R2 score: 0.9737 | loss: 8.2693 | Model Weights Changed And Best Model Saved
87/87 [==============================] - 1s 6ms/step
Epoch: 06/20 | 0:00:07 | Price R2 score: 0.1555 | loss: 5.4655 | Model Weights Not Changed
87/87 [==============================] - 1s 9ms/step
Epoch: 07/20 | 0:00:07 | Price R2 score: 0.8364 | loss: 4.4964 | Model Weights Not Changed
87/87 [==============================] - 0s 6ms/step
Epoch: 08/20 | 0:00:07 | Price R2 score: -0.0651 | loss: 4.7612 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 09/20 | 0:00:07 | Price R2 score: -0.5308 | loss: 4.3423 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 10/20 | 0:00:07 | Price R2 score: -1.9638 | loss: 4.5321 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 11/20 | 0:00:07 | Price R2 score: -0.1478 | loss: 3.9055 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 12/20 | 0:00:08 | Price R2 score: -0.0303 | loss: 3.7934 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 13/20 | 0:00:07 | Price R2 score: -0.6795 | loss: 3.4898 | Model Weights Not Changed
87/87 [==============================] - 1s 6ms/step
Epoch: 14/20 | 0:00:08 | Price R2 score: 0.9758 | loss: 3.5544 | Model Weights Changed And Best Model Saved
87/87 [==============================] - 0s 6ms/step
Epoch: 15/20 | 0:00:08 | Price R2 score: 0.0656 | loss: 2.8666 | Model Weights Not Changed
```

Figure 25. An example of how the custom callback works is that the model weights are updated when the R2 score improves compared to the previous best.

4.1.4. Models Evaluation and Visualization

After training the 12 models, we proceeded to the model evaluation step. We stored the evaluation metrics for each model on the test set in a dictionary once the training was complete. The image below shows the results of each model during this Ensemble Learning period:

Models Evaluation Results

```
[ ] df = pd.DataFrame.from_dict(models_test_result, orient = 'index')
    df
```

	r2_score	mean_absolute_percentage_error	explained_variance_score	mean_squared_log_error	mean_poisson_deviance	max_error	mean_absolute_error	root_mean_squared_error
best_BTCPred_bi_gru_prob	0.994287	0.001439	0.995967	0.000003	0.289840	1414.234375	121.425583	156.297287
best_BTCPred_bi_gru_simple	0.995014	0.001240	0.995368	0.000003	0.252035	1284.539062	105.051178	146.021637
best_BTCPred_bi_gru_time_dist	0.977130	0.003160	0.991213	0.000013	1.148573	1295.000000	269.284698	312.722382
best_BTCPred_gru_prob	0.996183	0.001111	0.996275	0.000002	0.192650	1270.710938	94.227470	127.761360
best_BTCPred_gru_simple	0.994455	0.001356	0.995760	0.000003	0.279116	1438.312500	115.081215	153.986984
best_BTCPred_gru_time_dist	0.942237	0.004724	0.955488	0.000033	2.866858	1571.789062	405.842194	496.990662
best_BTCPred_bi_lstm_prob	0.991655	0.001839	0.996113	0.000005	0.421206	1250.421875	156.123047	188.901062
best_BTCPred_bi_lstm_simple	0.992874	0.001596	0.993709	0.000004	0.356988	1169.187500	136.027802	174.563354
best_BTCPred_bi_lstm_time_dist	0.989870	0.001815	0.990297	0.000006	0.510304	1218.242188	154.173065	208.125259
best_BTCPred_lstm_prob	0.993715	0.001527	0.996117	0.000004	0.317509	1269.687500	129.581970	163.943314
best_BTCPred_lstm_simple	0.993479	0.001512	0.993571	0.000004	0.328636	1205.539062	128.357361	166.980026
best_BTCPred_lstm_time_dist	0.989367	0.001857	0.989410	0.000006	0.538095	1250.000000	157.133194	213.228287

Figure 26. Models Evaluation Results on BTC data

Then, we plotted the prediction outputs on graphs for visualization. The result images of each model will be discussed in detail in Section 4: Results and Discussion.

4.2. TRANSFER LEARNING

After completing the first phase, we used the best-performing model as a pre-trained model and transferred it to predict prices for our stock price data (Apple, Amazon, Google, and Microsoft) in this second phase. This transfer learning approach leverages the knowledge gained from the pre-trained model's experience with BTC data and applies it to forecast stock prices. By utilizing this transferred knowledge, the model improves its performance on new datasets, demonstrating the effectiveness of transfer learning in predicting stock prices.

4.2.1. Data Preprocessing

Before the data preprocessing step, we downloaded the stock data of four major tech companies, including Apple, Amazon, Microsoft, and Google, using yfinance. We used the notebook 00 in the submitted "notebook" folder to download and rename the columns of these data files. The downloaded data is stored in four CSV files in the "raw" folder, with each CSV file containing the stock data for one company.

4.2.1.1. Feature Engineering

In order to enhance the model's performance in this second phase, we tried three different feature engineering approaches. The first approach involved adding Technical Indicators. The second approach involved adding Graph features, specifically the Collective Influence columns. The third approach combined both the first and second approaches.

4.2.1.1.1. Case 1: Adding 5 Technical Indicators into raw data

a. Simple Moving Average (SMA_20)

The purpose of this function is to smooth out short-term fluctuations by averaging closing prices over 20 days to benefit prediction by capturing long-term trend; helps the model understand momentum.

$$\text{SMA}_{20} = \frac{1}{20}\sum_{i=t-19}^{t} P_i$$

Example: If the closing prices for the last 5 days are [100, 102, 101, 99, 98], the 5-day SMA would be (100+102+101+99+98)/5 = 100. To help prediction by smoothing out daily price ups and downs, allowing the model to focus on the broader trend over time. By averaging the closing prices over 20 days, it reveals whether the stock is generally moving up, down, or staying steady à long-term patterns and momentum.

b. Stochastic Oscillator (%K, %D)

The purpose of this function is to indicate momentum and overbought/oversold conditions by showing how strong the current price is compared to its recent range. If the price is near the high end of its 50-day range, it suggests strong momentum; if it's near the low end, it may indicate weakness. The %K line captures this position, and the %D line smooths it out for better trend detection. This helps the model identify potential turning points—like when a stock might be overbought or oversold—improving its ability to forecast short-term price movements.

%K = ((Close - Low_50) / (High_50 - Low_50)) * 100;

%D = SMA(%K, 30-day)

Code:

```
low_50 = data["Low"].rolling(window=50).min()
high_50 = data["High"].rolling(window=50).max()
data["Stochastic_%K"] = ((data["Close"] - low_50) / (high_50 - low_50)) * 100
data["Stochastic_%D"] = data["Stochastic_%K"].rolling(window=30).mean()
```

Example: If the 50-day low is 90, the 50-day high is 110, and the current close is 100, then

$$\%K = ((100 - 90) / (110 - 90)) * 100 = 50.$$

This means

- A %K of 50 means the current price is exactly halfway between the 50-day high and low.
- If the %K were closer to 100, it would mean the stock is trading near its recent high (potentially overbought).
- If it were closer to 0, it would mean the stock is near its recent low (potentially oversold).

c. MACD & Signal Line

This function is to highlight the changes in trend direction and strength.

$$MACD = EMA\_12 - EMA\_26; Signal = EMA\_9 \text{ of } MACD$$

Code:

```
short_ema = data["Close"].ewm(span=12, adjust=False).mean()
long_ema = data["Close"].ewm(span=26, adjust=False).mean()
data["MACD"] = short_ema - long_ema
data["MACD_Signal"] = data["MACD"].ewm(span=9, adjust=False).mean()
```

Example: If the 12-day EMA is 105 and the 26-day EMA is 100, then MACD = 5. The Signal line is the 9-day EMA of MACD values. 12-day EMA = 105: This is the average closing price over the past 12 days, weighted to give more importance to recent prices. 26-day EMA = 100: This is the longer-term average, also weighted.

$$MACD = 105 - 100 = 5$$

The positive value means recent prices are moving up faster than the longer-term trend-signaling upward momentum. The Signal Line is then calculated by averaging recent MACD values over the past 9 days, helping detect when momentum is accelerating or slowing.

If the MACD crosses above the Signal Line, it often signals a bullish trend. If it crosses below, it may indicate a bearish trend, making this a valuable tool for predicting turning points in price direction.

These specific values (12, 26, 9) were introduced by Gerald Appel, who developed the MACD in the late 1970s. They effectively capture short- and medium-term price movements.

They give reliable signals across many types of stocks and time periods.

4.2.1.1.2. Case 2: Adding Graph-based features: 7 Collective Influence (CI) columns into raw data

Besides using Technical Indicators, we also experimented with Graph-based features. Specifically, we added 7 Collective Influence (CI) columns into the raw data. To extract these 7 CI columns, we used the Price Graph method, which was introduced by Wu et al. (2022). The 7 Collective Influence columns added to the raw data represent node weights, which are extracted from the visibility graphs (VGs) created from the time series stock data input.

Based on the OHCLV data, the visibility graph (VG) algorithm is applied to transform stock price time series data, specifically using OHCLV (Open, High, Close, Low, Volume) data. This transformation converts the time series into a price graph, where each data point from the original series becomes a node in the graph. Two nodes are connected by an edge if they can "see" each other, meaning no other points interrupt a direct line between them. This is the core idea of a visibility graph (VG), which captures relationships between data points based on their visibility and helps measure how quickly information can propagate through the series. The VG algorithm is commonly used in financial time series analysis because it is not influenced by algorithmic parameters, and it maps the time series into scale-free graphs, making it particularly suitable for modeling stock price movements. The resulting graph illustrates how stock prices evolve over time, with edges representing periods when price changes occur without sudden jumps. Once the time series is converted into a price graph, the Collective Influence (CI) algorithm is employed to evaluate the importance of each node in the graph. Unlike traditional methods that focus on local features like node degree, the CI algorithm assesses the overall structure of the graph to determine the relative influence of each data point. The CI algorithm measures node weights, capturing the significance of each point in the series based on its position within the broader graph. By combining these two methods, we are able to extract meaningful structural information from the stock price data and use it to better understand the underlying patterns in the financial time series.

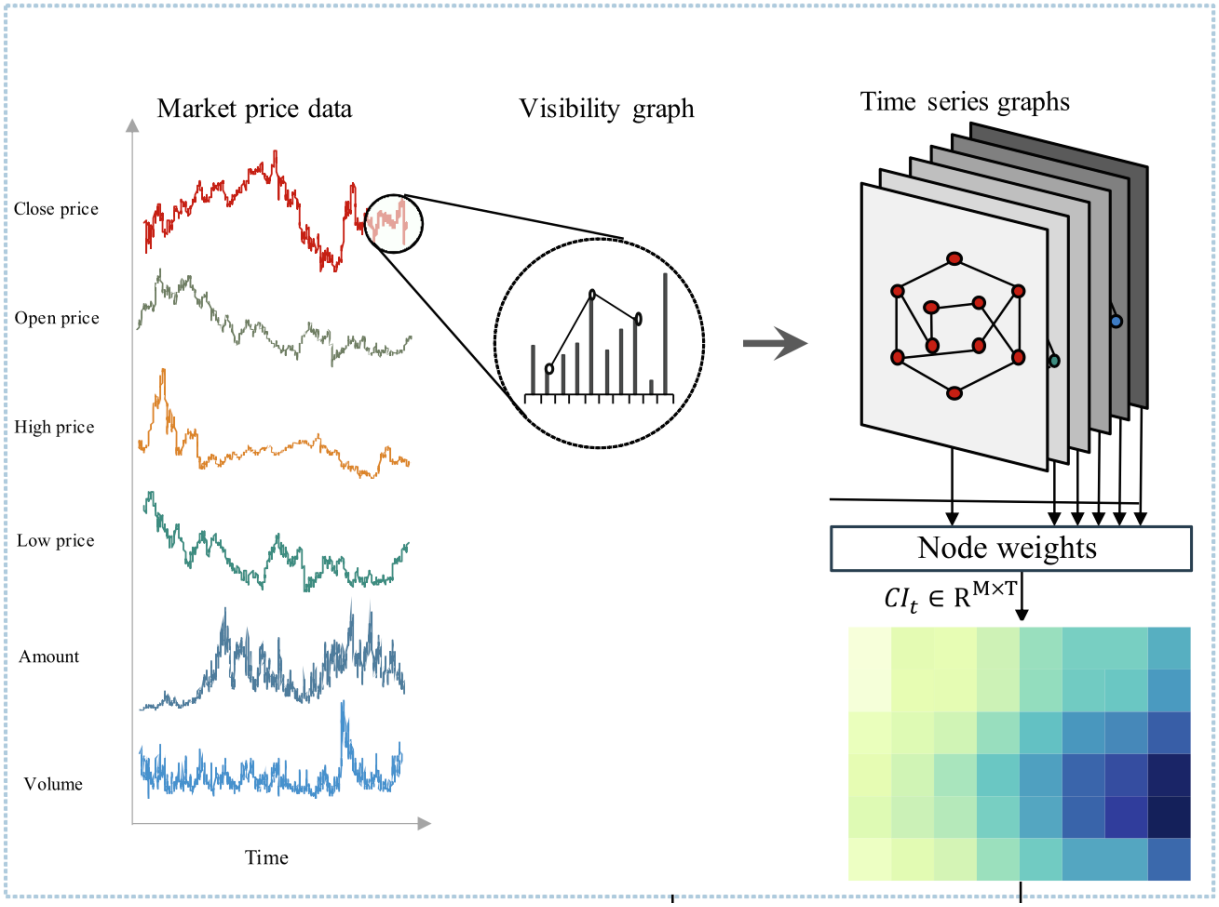
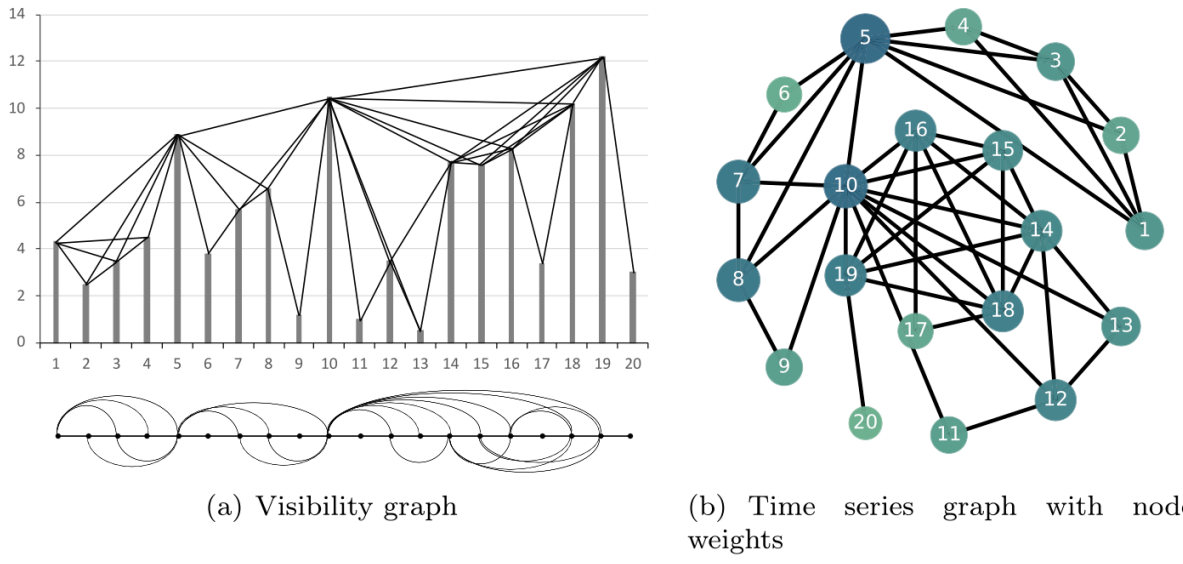


Figure 27. Computational flow of the Price Graph framework (Wu et al., 2022). The node weights are represented as CI_T ∈R^(M × T) , where M is the number of stock quote data (six in this case), T is the length of the lookback window, and E is the embedding size



(a) Visibility graph

(b) Time series graph with node weights

Figure 28. In Figure a, an example of the VG algorithm is shown, where a time series with 20 data points (T=20) is transformed into a VG. Figure b then shows the same time series graph, but with the Collective Influence (CI) applied to the node weights.

In the original Price Graph paper by Wu et al. (2022), T, the length of the lookback window, is set to 20, meaning 20 days. However, we experimented with various window sizes in this project, including T = 20, T = 10, and T = 7. When using a window size of 7 days, the results were better than with other window sizes. Therefore, we chose to use T = 7, resulting in the creation of 7 CI columns.

4.2.1.1.3. Case 3: Adding 5 Technical Indicators and 7 CI columns into raw data

In this third case, we combined the methods from case 1 and case 2, expecting it to yield the best results. To create the dataset for this third case, we used the raw data downloaded in notebook 00 from the "notebook" folder. We then used it as input for notebook 01 to add 7 columns to the raw dataset. After that, we took the resulting CSV file from that notebook, stored in the “data/price_graph” folder, and processed it through notebook 02 to add 5 technical indicator columns. The final CSV output was saved in the “data/indicators_and_graph” folder. We used this final CSV output file as the data in the transfer learning process to test the results of case 3.

Date	Open	High	Low	Close	Volume	ci1	ci2	ci3	ci4	ci5	ci6	ci7	SMA_20	Stochastic_%K	Stochastic_%D	MACD	MACD_Signal
2012-02-02	9.086000443	9.097000122	8.840000153	8.982500076	174726000	0	4	4	3	3	3	3	9.206824923	31.22447947	42.58088641	0.164518282	0.167589944
2012-02-03	9.383999825	9.395000458	9.094499588	9.141500473	162410000	0	12	6	0	6	6	6	9.22404995	44.20410975	27.60544594	0.154440465	0.164960048
2012-02-06	9.156999588	9.328000069	9.145999908	9.31400013	106200000	7	7	0	4	4	6	0	9.244574976	58.28572096	44.57143673	0.154767048	0.162921448
2012-02-07	9.209500313	9.246999741	9.102999687	9.132499695	102078000	0	4	6	6	0	8	8	9.244299984	43.46935201	48.65306091	0.144304820	0.159198123
2012-02-08	9.274000168	9.324500084	9.145500183	9.24750042	109532000	8	16	16	0	16	12	12	9.25392499	52.85717066	51.53741454	0.141113685	0.155581272
2012-02-09	9.248999596	9.284500122	9.088000298	9.225000381	143784000	8	8	0	6	6	6	6	9.266074991	51.020432	49.11565155	0.136552492	0.151775516
2012-02-10	9.277000427	9.381500244	9.126000404	9.170999527	115942000	0	0	8	8	0	8	8	9.276074982	46.61219692	50.16326652	0.129052116	0.147230836
2012-02-13	9.579500198	9.625	9.284000397	9.358499527	121428000	0	18	18	0	18	18	0	9.304474974	61.91832651	53.18365181	0.132554395	0.144295547

Figure 29. Some rows in the "indicators_ci_Amazon_20120101_to_20241212.csv" file

4.2.1.2. Normalization

To normalize the stock data, we used the same method as in the Ensemble Learning phase, which involves an advanced version of Min-Max Scaling that incorporates parameters α and β into the Min-Max scaler. A detailed explanation of the formula can be found in Section 4.1.1.1. Normalization.

$$x_{norm} = \frac{x - (\alpha \times x_{min})}{(\beta \times x_{max}) - (\alpha \times x_{min})} \text{ with } 0 < \alpha \leq 1, 1 \leq \beta$$

4.2.1.3. Data Splitting

In this phase, we split the training and test sets based on a specific date, conducting two experiments. In Experiment 1, we split the training and test sets on 01-01-2024, using 11 years of data for training and testing on 1 year. In Experiment 2, we split the training and test sets on 01-01-2023, using 10 years of data for training and testing on 2 years. After splitting the data into training and test

sets, we further divided the training set into train and validation sets. The train set consists of 80% of the training data, while the validation set contains the remaining 20%.

4.2.2. Transfer Learning and Fine-Tuning

After identifying best effective model by training 12 models in the Ensemble Learning phase, in this Transfer Learning phase, we used the best model by **freezing all layers fixed except the last one**. This approach locks the layers to capture the best features from the input data, while training only the final layer with a new dataset to produce the most accurate results based on the features obtained from the frozen layers.

Transfer Model for Probabilistic Prediction with Fine-Tuning (trained model on BTC price data)

```
[ ] def transfer_model(transfer_model_path = None, transfer_model_name = None, name = 'best_TRXPred_gru_prob'):
        """
        Description:
        - Load a pre-trained model, freeze layers, and create a new model for transfer learning with fine-tuning.

        Parameters:
        - transfer_model_path (str): Path to the directory containing the pre-trained model.
        - transfer_model_name (str): Name of the pre-trained model file.
        - name (str): Name for the new transfer model.

        Returns:
        - tf.keras.Model: Compiled transfer model with fine-tuning for probabilistic prediction.
        """
        # Default values for transfer model path and name
        if transfer_model_path is None:
            transfer_model_path = 'best_BTCPred_gru_prob'
        if transfer_model_name is None:
            transfer_model_name = 'best_BTCPred_gru_prob'

        # Load the pre-trained model
        m = load_model(f'{transfer_model_path}/{transfer_model_name}.keras', safe_mode = False)

        # Freeze layers except the last one
        for layer in m.layers[:-3]:
            layer.trainable = False

        # Set a new name for the transfer model
        m._name = name

        # Compile the transfer model with a specified optimizer and loss function (NLL)
        m.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-3), loss = lambda y, p: -p.log_prob(y), metrics = None)
        return m
```

Figure 30. Code of Transfer Model to Transfer Learning

5. RESULTS & DISCUSSION

This section provides a thorough study of our experimental findings, starting with an explanation of the evaluation measures used to gauge model performance and moving on to a thorough examination of prediction results for various models and businesses.

5.1. OUTCOME ASSESSMENT

To quantitatively evaluate the performance of our prediction models, we employed three complementary metrics that provide different perspectives on prediction accuracy.

5.1.1. Mean Absolute Error (MAE)

MAE measures the average magnitude of errors in a set of predictions, without considering their direction. It represents the average over the test sample of the absolute differences between prediction and actual observation:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

5.1.2. Root Mean Square Error (RMSE)

RMSE represents the square root of the second sample moment of the differences between predicted values and observed values:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

RMSE is especially helpful for discovering models that avoid major prediction deviations since it assigns a comparatively higher weight to huge errors. This is particularly helpful when predicting stock prices because significant mistakes can lead to significant losses.

5.1.3. Mean Absolute Percent Error (MAPE)

MAPE expresses accuracy as a percentage of the error, providing a measure of prediction accuracy that is scale-independent:

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

This metric is particularly useful for comparing prediction performance across different stocks with varying price ranges.

5.2. RESULTS

Four significant IT companies—Apple, Microsoft, Google, and Amazon—are compared in our experimental study of several sophisticated deep learning architectures for stock price prediction. We provide a thorough examination of performance indicators for every model and business combination.

*Table 1. Performance Comparison of our methods compared with traditional LSTM and GRU (split_date = **2024**-01-01)*

Company	Metric	LSTM	GRU	P_GRU	P_GRU 1.1	P_GRU 1.2	P_GRU 1.3
Apple	MAE	5.26	6.14	2.52	2.29	2.26	2.31
	RMSE	6.70	7.52	3.27	3.38	3.05	3.41
	MAPE	2.53	2.94	1.22	1.13	1.11	1.14
Amazon	MAE	5.28	5.17	2.44	2.34	2.55	2.43
	RMSE	7.03	6.73	3.30	3.40	3.44	3.43
	MAPE	2.82	2.77	1.34	1.30	1.39	1.32
Microsoft	MAE	9.15	12.15	4.16	4.32	4.00	4.41
	RMSE	11.21	14.22	5.56	5.66	5.34	5.93
	MAPE	2.18	2.87	1.00	1.04	0.96	1.06
Google	MAE	5.25	5.3	2.28	2.10	2.05	2.21
	RMSE	6.31	6.32	3.05	3.11	2.90	3.19
	MAPE	3.17	3.22	1.41	1.30	1.26	1.36

- **P_GRU variants significantly outperform** traditional models (LSTM and WGAN) on all metrics and companies.
- P_GRU 1.2 is the best-performing variant overall, consistently achieving the lowest MAE, RMSE, and MAPE across most datasets.
- **LSTM and WGAN** trail far behind in predictive accuracy, with WGAN especially underperforming on Amazon and Microsoft.

Table 2. Performance Comparison of our methods compared with traditional LSTM and GRU (split_date = 2023-01-01)

Company	**Metric**	**LSTM**	**GRU**	**P_GRU**	**P_GRU 1.1**	**P_GRU 1.2**	**P_GRU 1.3**
Apple	MAE	7.52	5.72	2.79	2.01	2.12	3.63
	RMSE	9.2	7.22	3.64	2.87	2.85	4.51
	MAPE	3.87	2.90	1.46	1.09	1.14	1.91
Amazon	MAE	5.2	4.92	2.19	2.18	2.12	2.12
	RMSE	7	6.41	2.92	3.05	2.86	2.98
	MAPE	2.78	3.12	1.49	1.51	1.45	1.46
Microsoft	MAE	11.19	13.64	5.61	4.65	3.99	4.31
	RMSE	13.47	16.61	7.1	5.88	5.25	5.69
	MAPE	3.05	3.55	1.56	1.29	1.14	1.22
Google	MAE	5.3	5.08	2.54	1.87	1.88	2.05
	RMSE	6.34	6.40	3.23	2.65	2.63	2.29
	MAPE	3.25	3.51	1.83	1.35	1.38	1.47

- All P_GRU variants dramatically outperform LSTM in every metric across all companies.
- **P_GRU 1.2** and **P_GRU 1.1** emerge as the strongest overall performers, offering the **lowest average MAE, RMSE, and MAPE**.
- **P_GRU 2.0** shows mixed performance—solid on Google but **weaker** on Apple

5.2.1. P_GRU with basic stock indicators

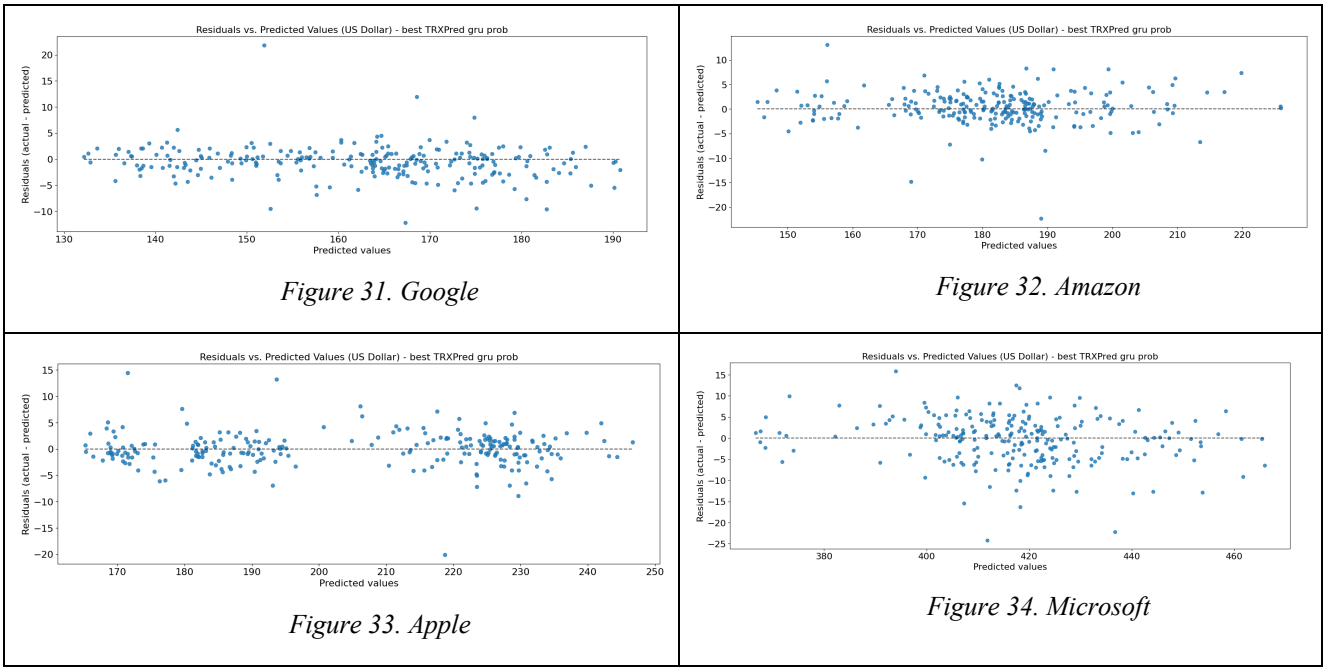


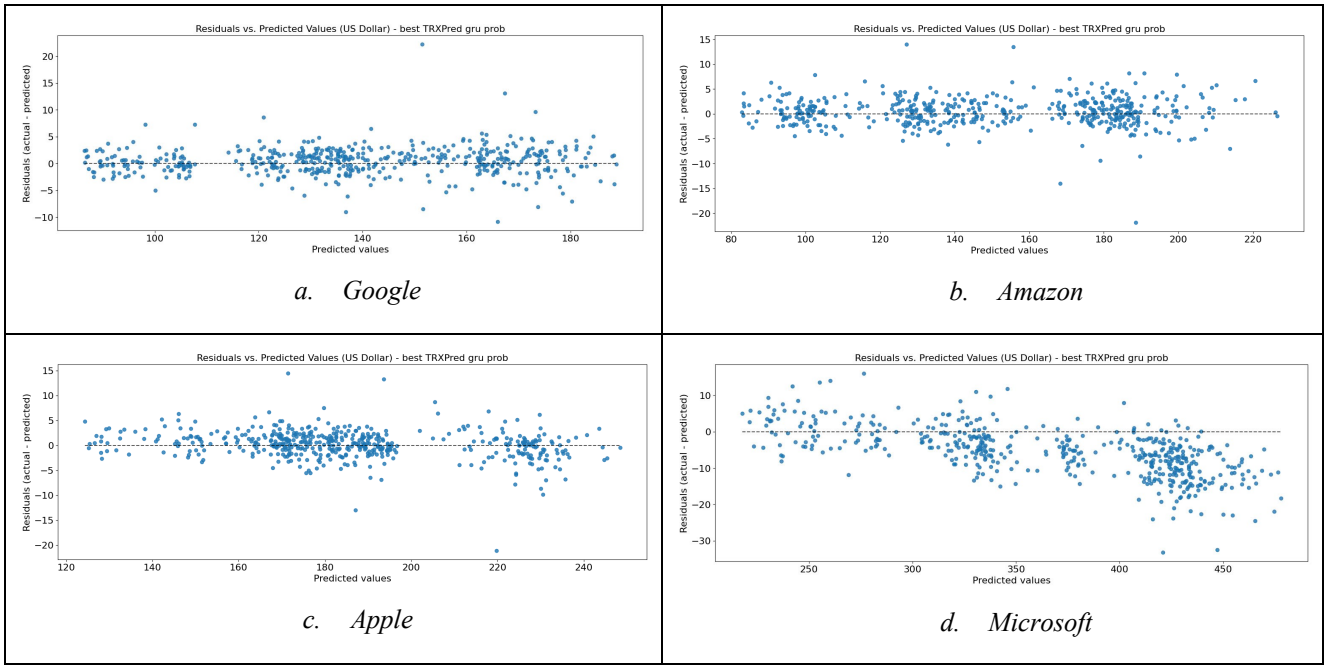
Figure 31. Google

Figure 32. Amazon

Figure 33. Apple

Figure 34. Microsoft

Figure 35. Residual Analysis for Technology Companies 2024



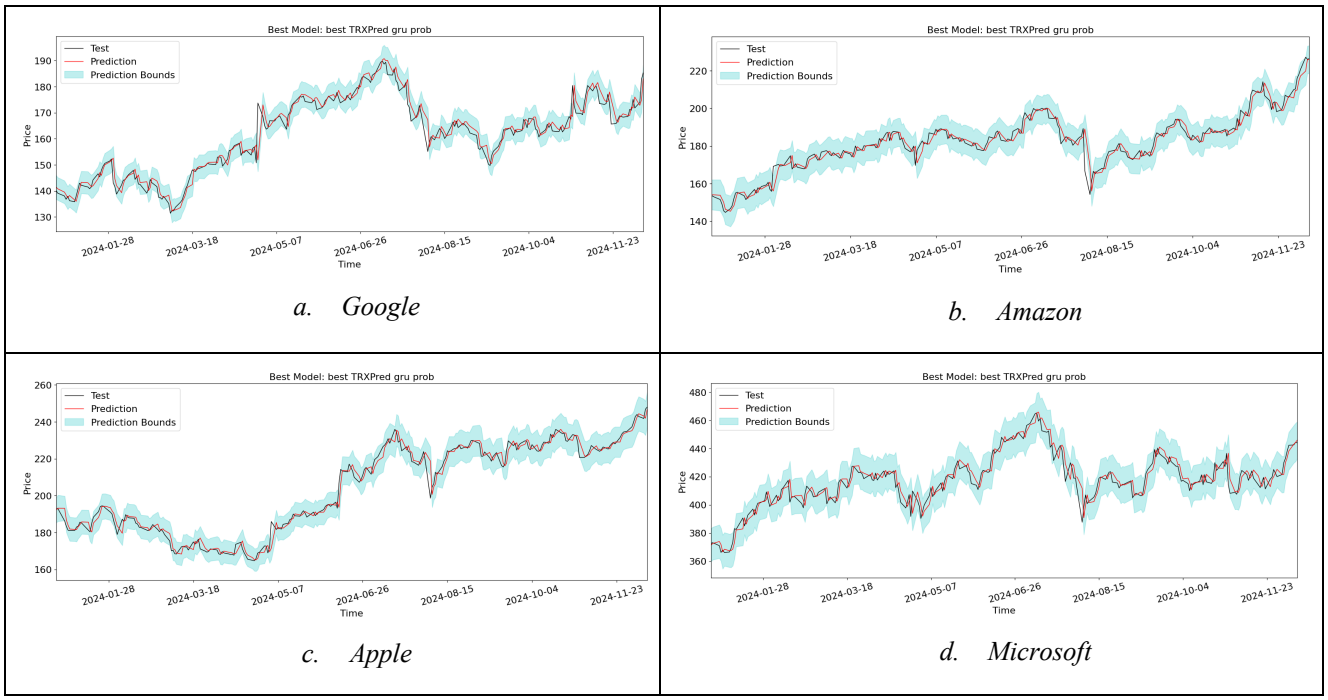
a. Google

b. Amazon

c. Apple

d. Microsoft

Figure 36. Residual Analysis for Technology Companies 2023-2024



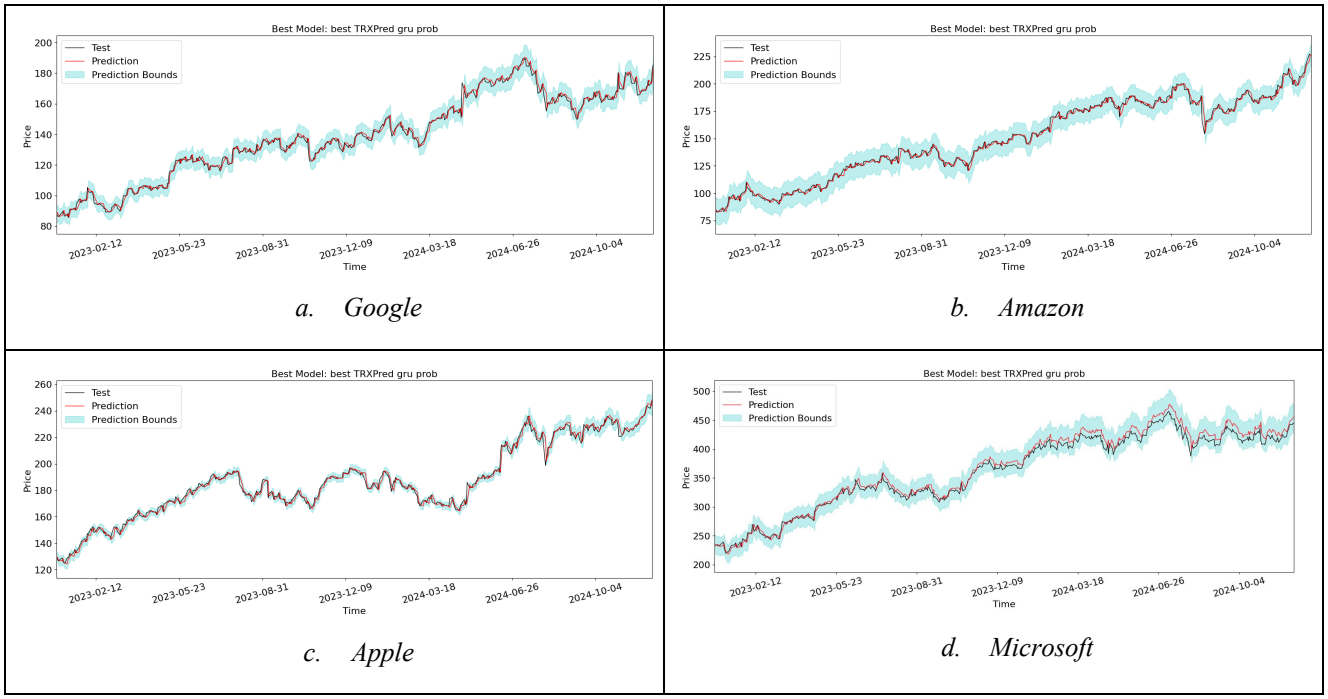
a. Google

b. Amazon

c. Apple

d. Microsoft

Figure 37. Difference of Prediction and Truth values Visualizations 2024



a. Google *b. Amazon*

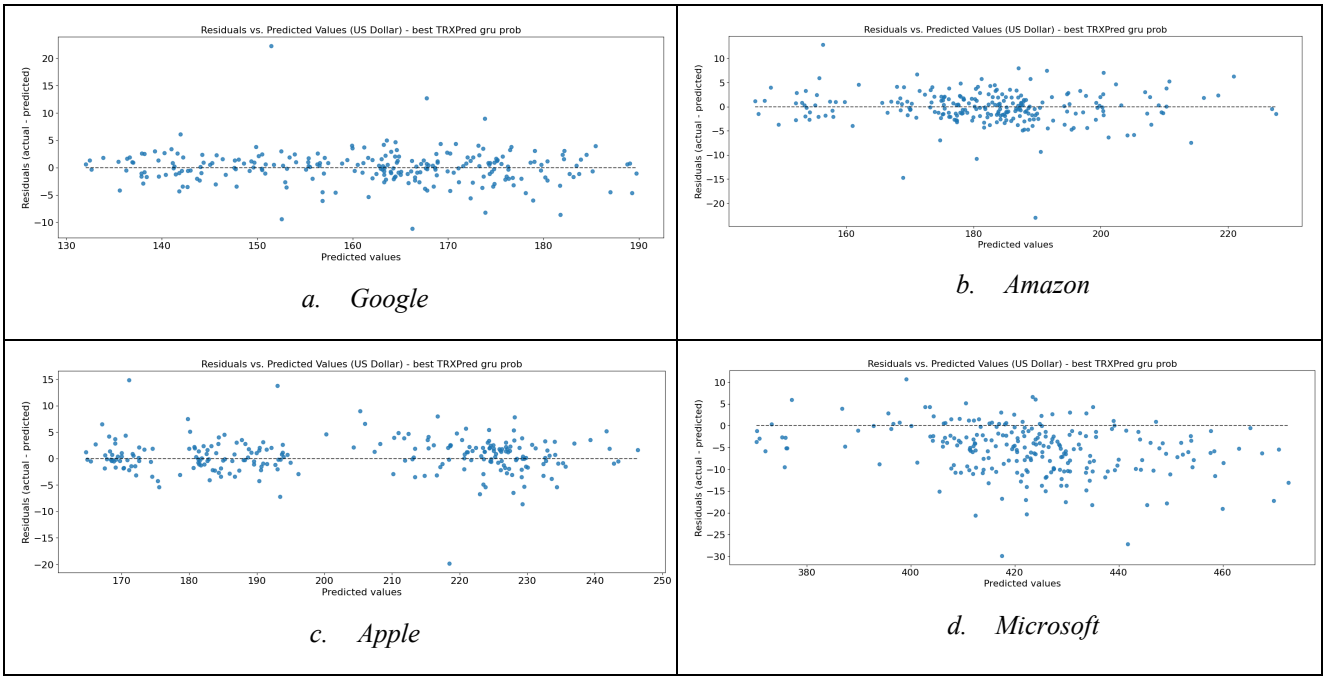
c. Apple *d. Microsoft*

Figure 38. Difference of Prediction and Truth values Visualizations 2023-2024

All four of the largest IT companies show outstanding prediction performance using P_GRU after adding some stock indicators. Time series forecasts accurately capture both long-term trends and short-term oscillations, demonstrating a striking alignment between projected and actual stock prices throughout 2024.

Prediction boundaries are a useful tool for quantifying uncertainty, and they appropriately widen at times of increasing market volatility, like the price swings of Microsoft in June and July and the erratic trading of Amazon in May and June. Different error patterns are revealed by residual analysis: Microsoft shows the widest error dispersion; Amazon shows a clear negative trend in residuals at higher predicted values; Apple consistently performs well at mid-range prices with a slight negative bias at higher values; and Google shows the most balanced distribution. The model's temporal consistency over long time horizons is confirmed by long-term forecasting. All things considered, the FE model does a good job at capturing the intricate, non-linear dynamics of stock price fluctuations. At higher price points, it performs best for Google and Apple but struggles more with Microsoft and Amazon.

5.2.2. Feature Engineering + Price Graph



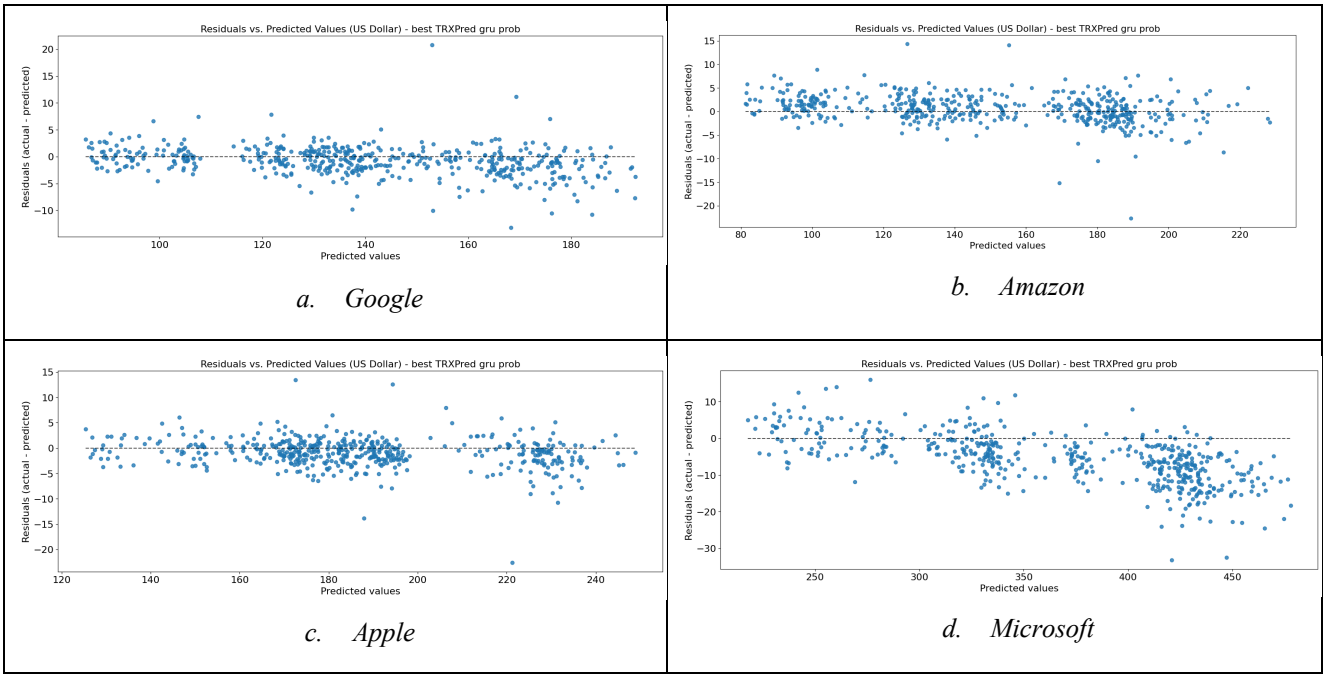
a. Google

b. Amazon

c. Apple

d. Microsoft

Figure 39. Residual Analysis for Technology Companies 2024



a. Google

b. Amazon

c. Apple

d. Microsoft

Figure 40. Residual Analysis for Technology Companies 2023-2024

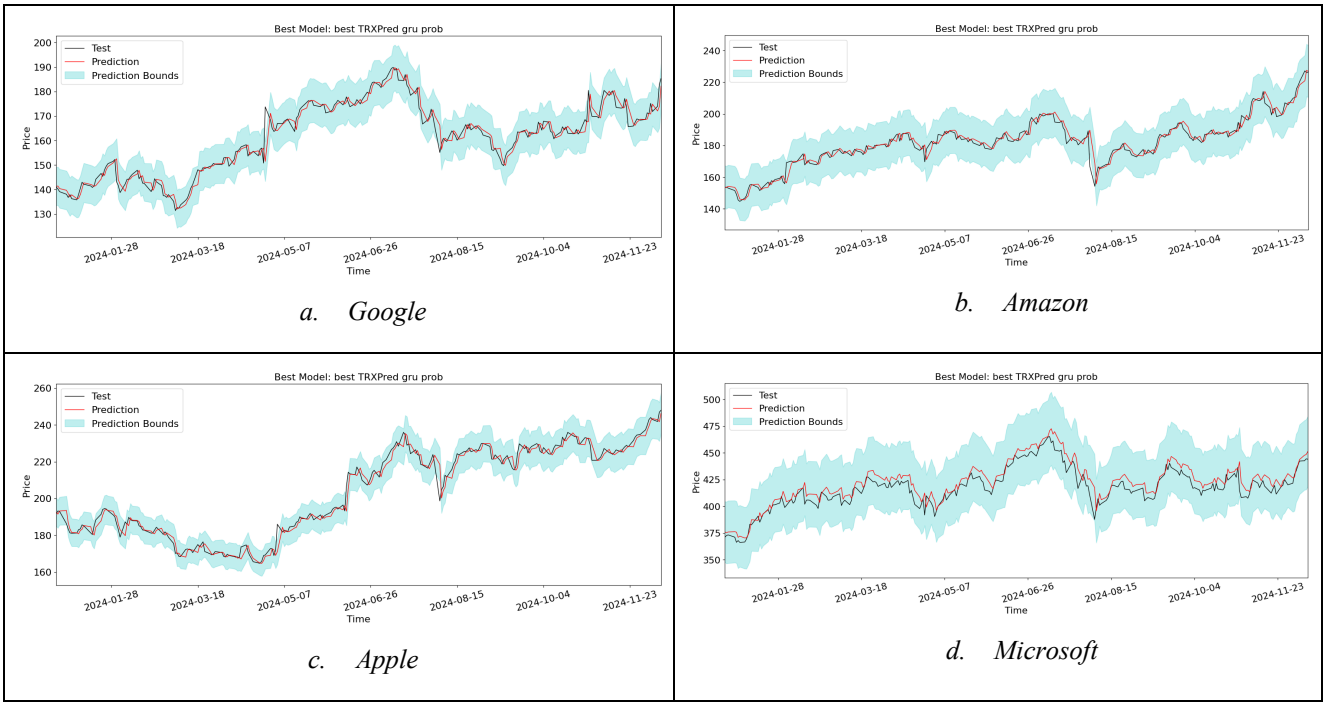


Figure 41. Time Series Prediction Visualizations 2024

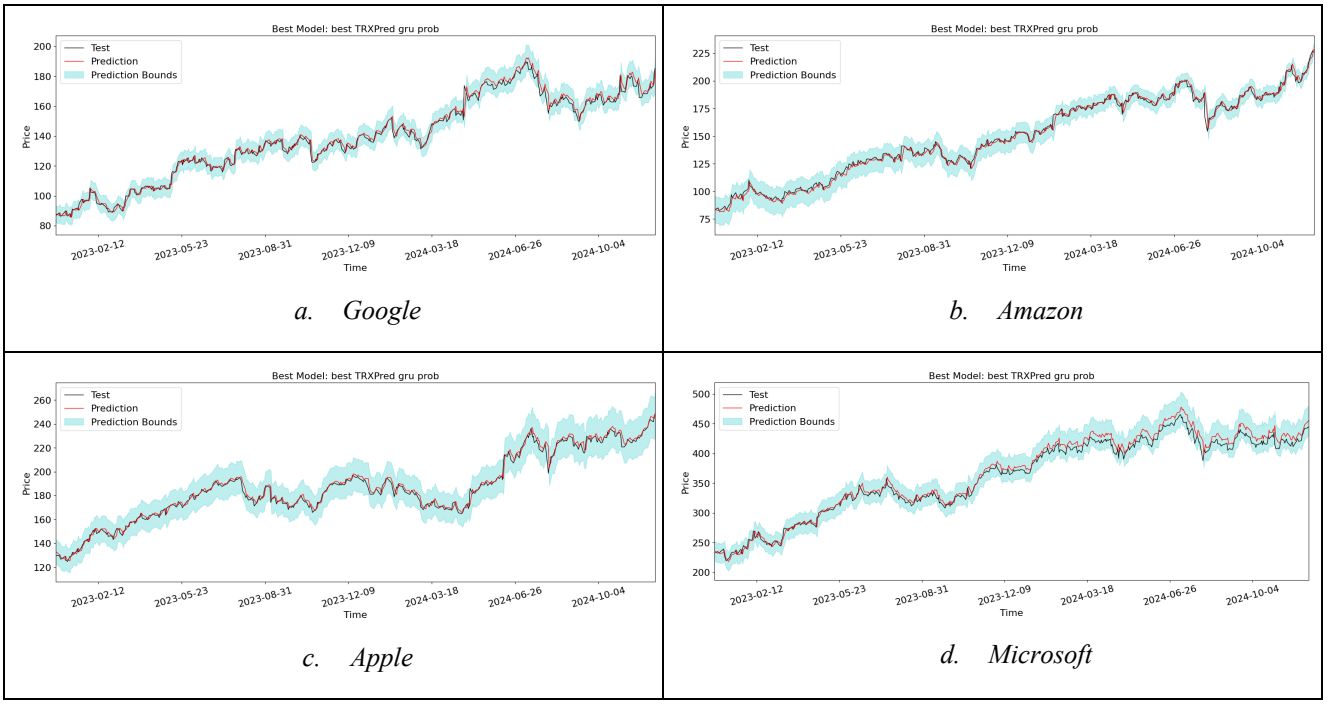


Figure 42. Time Series Prediction Visualizations 2023-2024

All four technology companies demonstrate notable advantages over the FE-only technique using the Feature Engineering + Probability Guidance (FE+PG) strategy. Throughout 2024, time series

forecasts show remarkable agreement between expected and actual stock prices, successfully capturing both long-term patterns and transient swings. Improved error patterns are revealed by residual analysis: Apple and Microsoft have more uniformly distributed residuals, particularly in mid-range pricing; Google maintains a balanced distribution with fewer outliers; and Amazon exhibits a noticeable decrease in negative bias at higher predicted values. Prediction boundaries exhibit improved flexibility, suitably varying their width in both volatile and stable market conditions. This enhanced quantification of uncertainty is especially noticeable during major market occurrences, such as the July 2024 severe corrections. Long-term projections for 2023–2024 show strong performance over long time horizons, accurately capturing general upward trends while capturing volatility patterns unique to individual companies.

6. DEPLOYING A MODEL IN A STREAMLIT WEB APP

We developed a web app to create an interactive interface for interacting with the model. In the web app, the model we used is the original **P_GRU** model, and the datasets include Apple, Amazon, Microsoft, and Google, retrieved using the yfinance Python dependency. The website was built using the Streamlit library.

Upon initialization, the program defines dictionaries for stock prices and model paths. It also creates UI elements such as a title and a dropdown button to select different stocks. When a stock is selected, the app loads the raw data from CSV files and caches it for better performance. If the dataset contains a ‘Date’ column, the app converts it to datetime format.

```
@st.cache_data
def load_data(file_path):
    if os.path.exists(file_path):
        data = pd.read_csv(file_path)
        if 'Date' in data.columns:
            data['Date'] = pd.to_datetime(data['Date'])
            data.set_index('Date', inplace=True)
```

Figure 43. Loading and cache dataset function

After the data loads successfully, the web page uses **Plotly**, a Python library, to plot the raw stock data based on the "Open" and "Close" values. The code checks if the dataset contains both "Open" and "Close" columns and then visualizes them on the website.

```python
def plot_raw_data():
    fig = go.Figure()
    # Check common column name patterns
    open_col = 'open' if 'open' in data.columns else 'Open'
    close_col = 'close' if 'close' in data.columns else 'Close'
    fig.add_trace(go.Scatter(x=data.index, y=data[open_col], name="Stock Open"))
    fig.add_trace(go.Scatter(x=data.index, y=data[close_col], name="Stock Close"))
    fig.layout.update(title_text='Time Series Data with Rangeslider', xaxis_rangeslider_visible=True)
    st.plotly_chart(fig)
```

Figure 44. Streamlit and plotly plot data

To calculate the model performance metrics, we saved the best model after training, along with the close_minmax_trxusd file used to normalize and denormalize the data.

- Save/Load model code: tf.saved_model.save(model, path)/ tf.saved_model.load(model_path)
- Save/Load close_minmax code: np.save(f'close_minmax_{price_pairs.lower()}.npy', close_minmax)/ close_minmax = np.load(close_minmax_path)

After loading the model and the close_minmax file, the web app preprocesses the data. First, it normalizes the data using the same close_minmax file that was used during training.

```python
Windsurf: Refactor | Explain | Generate Docstring | ×
def normalize(x_, minmax):
    return (x_ - minmax[0]) / (minmax[1] - minmax[0])

Windsurf: Refactor | Explain | Generate Docstring | ×
def denormalize(x_, minmax):
    return minmax[0] + ((minmax[1] - minmax[0]) * x_)
```

Figure 45. Normalize and denormalize function

Then, the web app prepares the prediction window. The function takes the last 6 days (analysis_duration=6), reshapes it into the proper format for model input, and returns the prepared data. The default analysis_duration is 6 because the model was trained to recognize patterns over 6 consecutive days. The prepared data is then converted into a TensorFlow tensor using tf.convert_to_tensor() for prediction.

```python
def prepare_prediction_window(close_price, analysis_duration):
    """
    Prepare the most recent data for prediction using a sliding window approach.

    Parameters:
    close_price (numpy.ndarray): Array of normalized close prices
    analysis_duration (int): Number of time steps to use for prediction (window size)

    Returns:
    numpy.ndarray: X_test data ready for prediction
    """
    X_test = []

    # Get the latest window of data for prediction
    # Using the same pattern as in the training implementation
    X_test.append(close_price[-analysis_duration:].reshape(analysis_duration))

    # Reshape testing data into the required format (samples, time steps, features)
    X_test = np.asarray(X_test).reshape((len(X_test), analysis_duration, 1))

    return X_test
```

Figure 46. Prepare prediction window for model input

After preprocessing the data and loading the model, the website proceeds to make predictions. Before predicting, it retrieves the **"serving_default"** signature from the loaded TensorFlow SavedModel using:

infer = reconstructed_model.signatures["serving_default"].

Next, the website makes a prediction using:

close_price = X_tensor (X_tensor is the preprocess data)

Then, the pred_array will be the model's probabilistic output, which is a distribution.

```python
predictions = infer(close_price=X_tensor)
if "distribution_lambda" in predictions:
    pred_array = predictions["distribution_lambda"].numpy()
```

Figure 47. Model predict based on the prepare prediction window

After obtaining the pred_array, the app calculates the mean and standard deviation of the distribution, then denormalizes both the actual and predicted values.

```
# Calculate mean and standard deviation for confidence intervals
pred_mean = pred_array.mean(axis=1)
pred_std = pred_array.std(axis=1)

# Denormalize predictions
actual_values = denormalize(normalized_close[-len(pred_mean):], close_minmax)
predicted_values = denormalize(pred_mean, close_minmax)
```

Figure 48. Model predict based on the prepare prediction window

Using Streamlit, a slider is created to let users select the test window size. The test window size defines how many recent days will be used to evaluate the model (e.g., if the test window size is 100, then MAE, RMSE, and MAPE will be computed over the last 100 days). The forecast_horizon represents how many days ahead the model is predicting; it is currently hardcoded as 1.

```
# Loop through the test window
for i in range(test_window):
    # End index for the current prediction
    end_idx = len(normalized_close) - test_window + i

    # Make a prediction using data up to end_idx
    X_test = prepare_prediction_window(normalized_close[:end_idx], analysis_duration)
    X_tensor = tf.convert_to_tensor(X_test, dtype=tf.float32)
    try:
        pred = infer(close_price=X_tensor)
        pred_mean = pred["distribution_lambda"].numpy().mean(axis=1)

        # Get actual value forecast_horizon steps ahead
        if end_idx + forecast_horizon < len(normalized_close):
            actual = normalized_close[end_idx + forecast_horizon - 1]
            predicted = pred_mean[0]

            # Denormalize values
            actual_denorm = denormalize(actual, close_minmax)[0]
            pred_denorm = denormalize(predicted, close_minmax)

            y_true_all.append(actual_denorm)
            y_pred_all.append(pred_denorm)
```

Figure 49. Using slider to determine the test window size

Next, the app calculates performance metrics including **MAE, RMSE, MAPE**, and **R^2,** and plots a graph comparing actual vs. predicted values. Finally, the app also plots the **error distribution** and **absolute percentage error distribution**. The formulas for these two metrics are shown below:

```
# Show error distribution
errors = y_true_np - y_pred_np
abs_pct_errors = np.abs((y_true_np - y_pred_np) / y_true_np) * 100
```

Figure 50. Error distribution and absolute percentage error distribution formula

7. CONCLUSION AND FUTURE WORK

In conclusion, after comparing several sophisticated deep learning architectures for stock price prediction for Apple, Microsoft, Google, and Amazon, the performance of Long Short-Term Memory (LSTM), Wasserstein Generative Adversarial Networks with Gradient Penalty (WGAN-GP), Deep Variational Autoencoders (DVA and DVAE), and Parallel Gated Recurrent Units (P_GRU) models was evaluated using Mean Absolute Error (MAE), Root Mean Square Error (RMSE), and Mean Absolute Percent Error (MAPE). The results indicated that **our approach using Ensemble Learning and Transfer Learning generally demonstrated strong prediction performance across the four technology companies**. Furthermore, this research explored the impact of feature engineering (adding stock indicators) into the P_GRU model, which led to improved performance for all four companies in 2024. This research also investigated the effect of incorporating price graph information in addition to feature engineering (adding CI columns strategy). This approach generally yielded notable advantages over the feature engineering-only method but perform slightly better than only using P_GRU with feature engineering. In summary, the results suggest that probabilistic deep learning models, particularly the improved versions of the P_GRU model enhanced with price graph T=7 outperformed older models in stock prediction for major companies.

7.1. FUTURE WORK

a. Incorporating more data

Stock price predictions should not solely rely on historical stock data due to its complex nature. Stock prices are influenced by many other factors beyond past performance. Therefore, to make the prediction more realistic and meaningful, we would want to incorporate more data such as sentiment analysis from **news** or **macroeconomic factors**.

b. Model optimization

While using LSTM, WGAN and P-GRU yielded good results, we would want to explore more advanced models or hybrid model approaches. We would try **experimenting with different stock combinations of technical indicators to find the most impactful ones for finance prediction**. Another approach should be considered is hybrid models, such as using an ensemble of LSTM and Transformer models.

c. Real-time data integration

Due to the ever-changing nature of stock price, we would explore integrating real-time stock price and real time sentiment analysis data into the model for live predictions. Having up-to-date data allows the model to reflect the most current market conditions.

8. Acknowledgement

We would like to thank Dr. Pham Thi Kim Dung, our lecturer, for her valuable feedback and advice throughout the progress of our project.

9. Contributions

Work Items		Bui Thanh Thao	Nguyen Tran Yen Binh	Pham Minh Viet	Nguyen Dang Huy
Experiment	D-Va	25%	25%	25%	25%
	DVAE	25%	25%	25%	25%
	WGAN-GP	-	10%	90%	-
Research	Find new models	26.5%	24.5%	24.5%	24.5%
Code	P-GRU original	80%	5%	5%	10%
	7 CI columns	100%	-	-	-
	Technical indicators	-	5%	-	95%
	GRU	100%	-	-	-
	Web app	-	-	100%	-
Presentation	Slide	25%	25%	25%	25%
	Video	25%	25%	25%	25%
Report		20%	65%	10%	5%
Average		**25.3%**	**24.8%**	**25%**	**24.9%**

In the first phase of the project, we tried multiple advanced models, including Diffusion Variational Autoencoders, to achieve the project goal, alongside commonly used models for time series data like LSTM and GRU. However, the results were not as high as we expected. As a result, we had to

conduct additional research and experiment with different models. The model and approach that produced the results we expected is the one we presented and submitted.

Description of Each Member's Contribution

- **Bui Thanh Thao**: Thao is the team leader. She is primarily responsible for the entire model training process. Whenever there are issues with the models, she is the one who tries to find solutions. The current approach we used for the project is one she discovered in class. From that point on, she took on the primary responsibility of training and running this original version of the model using stock data. She is also responsible for running the Price Graph model to add seven columns to the raw dataset. In both the Presentation Slide and the Report, Thao is responsible for the Methodology section. Additionally, in order to compare the results of this approach to the commonly used GRU model, she created, trained, and ran the GRU model. In the first phase of the project, she contributed equally with the other team members in testing the D-Va and DVAE models.
- **Pham Minh Viet**: Viet is a Computer Science student majoring in Software Development. Although his major is not AI, he has been very helpful to the team. He was responsible for writing the API and deploying our models to a web app for visualizing the model predictions using Streamlit, a Python UI library he had not used before. Although there was an issue where we could not run the model on the local machine and could only display output data, Viet, after receiving hints from our lecturer, fixed the issue that same night and updated the web app code to allow real-time predictions. For the video presentation, he is responsible for the Result, UI demonstration, and Conclusion sections. In the report, he wrote Section 5 (Deploy the Model) and the Conclusion. In addition to these main contributions, Viet spent time working on the WGAN model during the Experiment phase, even though it took a long time to train, in an effort to achieve the minimum MAPE. He also assisted the team with research and finding relevant papers after the results of the D-Va and DVAE models did not meet expectations.
- **Nguyen Dang Huy**: Huy has been very helpful throughout the project. His tasks involved running and tracking the results for all versions of the P-GRU model in three different cases across two experiments. He analyzed these results to evaluate the performance of each version. For Experiments 2 and 3, he also researched and applied feature engineering techniques to improve the models. In addition to his technical work, Huy wrote the final report and created slides to present our findings clearly.

- **Nguyen Tran Yen Binh**: Binh was responsible for the entire report. Although she does not have as much experience with training models as Thao and Huy, she worked diligently throughout the process. Her tasks included key responsibilities related to machine learning models, as well as report and presentation preparation. She conducted research and identified diffusion models using the "Papers with Code" platform to understand current advancements and implementations. In addition, she ran and made necessary adjustments to the DVA and DVAE models to ensure they functioned correctly and aligned with the project's goals. She also executed the P-GRU model, performing a train-test split with a 20% test ratio. Finally, she was responsible for writing a comprehensive report detailing the processes and findings, as well as creating effective slides to present the outcomes of the work.

References

Alonso, M. N., Batres-Estrada, G., & Moulin, A. (2018). *Deep Learning in Finance: Prediction of Stock Returns with Long Short-Term Memory Networks*. 251–277. https://doi.org/10.1002/9781119522225.ch13

Amin Golnari, Mohammad Hossein Komeili, & Azizi, Z. (2024). Probabilistic deep learning and transfer learning for robust cryptocurrency price prediction. *Expert Systems with Applications*, *255*, 124404–124404. https://doi.org/10.1016/j.eswa.2024.124404

Fischer, T., & Krauss, C. (2018). Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, *270*(2), 654–669.

Hayes, A. (2021, June 25). *Stochastic Oscillator*. Investopedia. https://www.investopedia.com/terms/s/stochasticoscillator.asp

jesse_jcharis. (2020, September 23). *How to Split Dataset into Training and Testing Dataset For Machine Learning*. JCharisTech. https://blog.jcharistech.com/2020/09/23/how-to-split-dataset-into-training-and-testing-dataset-for-machine-learning/

Papers with Code - Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. (2017). Paperswithcode.com. https://paperswithcode.com/paper/empirical-evaluation-of-gated-recurrent

Schlegel, A. (2024, December 12). *SMA Ribbon [A] — Indicator by AndrinSchlegel*. TradingView. https://www.tradingview.com/script/dWAn1Gk1/

Sezer, O. B., Mehmet Ugur Gudelek, & A. Murat Ozbayoglu. (2019). *Financial Time Series Forecasting with Deep Learning : A Systematic Literature Review: 2005-2019*. https://doi.org/10.48550/arxiv.1911.13288

Subramanian, B., Olimov, B., Naik, S. M., Kim, S., Park, K.-H., & Kim, J. (2022). An integrated mediapipe-optimized GRU model for Indian sign language recognition. *Scientific Reports*, *12*(1), 11964. https://doi.org/10.1038/s41598-022-15998-7

Wen, X., & Li, W. (2023). Time Series Prediction Based on LSTM-Attention-LSTM Model. *Time Series Prediction Based on LSTM-Attention-LSTM Model*, *11*, 48322–48331. https://doi.org/10.1109/access.2023.3276628