uc3m | Universidad Carlos III de Madrid

Master Degree in Statistics for Data Science

2024-2025

Master Thesis

Attention To Markets

Transformer-Based Forecasting of Realized Volatility from High-Frequency Stock Returns

---

Nicolas Manuel Bühringer

Prof. Dr. Daniel Peña

Madrid, September 15, 2025

AVOID PLAGIARISM

The University uses the Turnitin Feedback Studio program within the Aula Global for the delivery of student work. This program compares the originality of the work delivered by each student with millions of electronic resources and detects those parts of the text that are copied and pasted. Plagiarizing in a TFM is considered a **Serious Misconduct**, and may result in permanent expulsion from the University.

Abstract

This master thesis explores the use of a transformer encoder for forecasting time series with an application to the prediction of next-day realized volatility from high-frequency stock data. Using minute returns for 28 Dow Jones Industrial Average stocks (2010–2019), the model is trained on prior-day sequences of 380 returns and predicts log realized volatility. Against naïve and HAR baselines, the Transformer consistently improves on naïve and is broadly competitive with HAR. Diebold–Mariano tests assess statistical significance across stocks. Additional analysis includes the interpretation of attention heatmaps of intra-day market movements. These findings motivate integrating covariates and adaptations of the standard transformer architecture in future work.

Contents

List of Figures

List of Tables

1. Introduction

Time series analysis has long relied on models which are explicitly designed for sequential data like ARIMA models. These models process data step by step but often exhibit limitations in capturing long range dependencies. Moreover, statistical methods like ARIMA require a strong methodological understanding of time series dynamics for model specification and are inherently limited to modeling linear relationships in the data.

In recent years, deep learning techniques have proven to be powerful across various domains while at the same time not relying on too many strict assumptions as classical statistical methods. Feedforward neural networks are the basic version of a neural network with information flowing in one direction from input to output through so-called neurons in hidden layers. The transformer architecture proposed by Vaswani et al., (2017) has revolutionized the realms of natural language processing by shifting away from sequential processing. Instead, they employ a mechanism called self-attention which allows for parallelization in the training process and thus faster overall training speeds. Naturally, research looked for ways to employ this method to model time series data. Both Amazon and Google have presented transformer-based time series forecasting models which in case of Amazon is used for their demand forecasting (Eisenach et al., 2022; Lim et al., 2019).

Another field with high demand for time series forecasts is the financial sector, with volatility modeling being of particular importance. Volatility directly affects areas such as risk management, financial asset pricing and portfolio management. When volatility can be estimated with greater accuracy, institutions are better equipped to measure the risks linked to their investments. In addition, volatility serves as a direct variable in the valuation of derivatives like options since the premium is largely determined by the expectations about the future variability of the underlying asset. Therefore, dependable models for volatility forecasting are essential in modern finance, enabling decision making and supporting efficient markets. (Poon & Granger, 2003)

Classical methods for modeling and forecasting volatility were first developed by Engle (1982) who introduced the Autoregressive Conditional Heteroscedasticity (ARCH) model,

framing volatility as a dynamic process driven by available information. Bollerslev (1986) later extended this framework to the Generalized ARCH (GARCH) model, which remained the benchmark in both academia and industry for decades. More recently, Corsi (2009) presented the Heterogeneous Autoregressive model which is based on daily, weekly and monthly components of volatility following the Heterogeneous Market Hypothesis. It states that groups of investors make decisions based on different investment horizons and volatility emerges as a combination of these actions (Müller et al., 1997).

This thesis will attempt to employ a transformer-based model for univariate time series forecasting. The next-day realized volatility of 28 stocks of the Dow Jones Industrial Average will be forecasted using as input data the minutely returns of the preceding day, with the overall dataset spanning nine years of minutely returns. The evaluation on the test set will be compared against a naïve baseline by using the realized volatility of the previous day as a forecast and against a HAR model as presented by Corsi (2009).

The remainder of this thesis is structured as follows: Section 2 gives an overview over the nature of time series data and how it was modeled in the past. Section 3 introduces the transformer architecture and attention mechanism used in language models. Section 4 discusses challenges in using transformers for time series data and presents a literature review of transformer-based models for time series forecasting. Section 5 defines realized volatility and outlines the methodology of the experimental study and model architecture. Section 6 presents the out-of-sample results. Section 7 discusses the findings and highlights areas for future research.

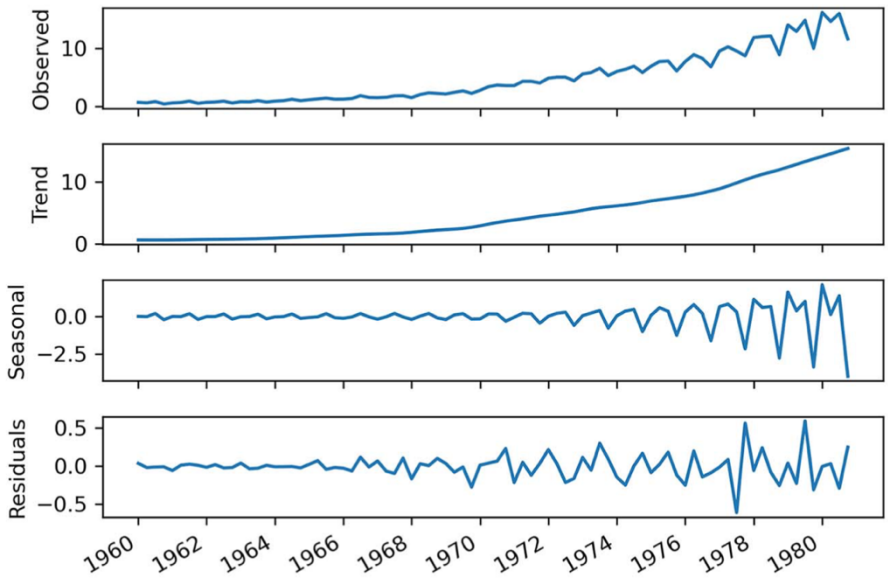
2. A Brief History of Time Series Forecasting

2.1 Statistical Methods

Univariate time series data refers to a sequence of observations collected at consistent time intervals. They reflect how a certain phenomenon evolves over time. Each data point corresponds to the value or state of the observed variable at a specific point in time. By analyzing the temporal structure of this data, patterns like trends or seasonal and cyclic effects can be

identified. This is often essential in order to forecast future values. Separating a time series into trend, seasonal and residual parts is called a decomposition and can be seen in Figure 1.

Figure 1: Decomposition of a time series (Peixeiro, 2022)



It was one of the first approaches proposed for time series analysis. The first graph of figure 1 shows the recorded values of the quarterly earnings of Johnson and Johnson from 1960 to 1980. The series is decomposed as the addition of Trend, Seasonal and Residuals from the graphs below. The second graph shows the trend and can be defined as the underlying slow-moving changes of the time series – in this case positive developments in the quarterly earnings due to an increasing trend line. The third graph shows the seasonal variation which is the deviation from the trend line during a fixed time period cycle. During one year, earnings start low, increase and decrease again towards the end of the year. The fourth graph shows the residuals which are the difference of the observed series and the sum of trend and seasonal components. Thus, it is the part that cannot be explained by any of the components. These are usually random errors in the data which are not possible to predict. Over time, more powerful methods for time series have been proposed, including ARIMA models, state space models and nonlinear models. (Peixeiro, 2022)

Box & Jenkins (1970) proposed the Autoregressive Integrated Moving Average Model in 1970 which predicts future values by utilizing the autocorrelation in the underlying data. It

uses linear combinations of past values (AR) and past error terms (MA) to predict values. It can be expanded to account for seasonal effects (S) through the SARIMA model. These models are simple and intuitive which enable them to identify basic patterns in data (Kim et al., 2025). Other methods for time series analysis have been proposed including state space models and nonlinear models. State space models represent a time series through latent state variables which capture unobserved components like trend or seasonality often estimated with the Kalman filter. Nonlinear models allow dynamics to change across regimes or depend on past volatility which makes them suitable for asymmetric behavior or structural breaks. (Peña & Tsay, 2021)

2.2 Machine Learning Models

Compared to traditional statistical models, machine learning models are highly effective at modeling nonlinear patterns in the data without relying on too strict assumptions. However, it is important to note that most machine learning methods were originally developed for independent and identically distributed tabular data. To apply them to time series forecasting, the temporal structure must first be transformed into a supervised learning problem. This is usually achieved by constructing lagged input features and additional engineered covariates like rolling averages or calendar effects. In this way, the forecasting task becomes a regression or classification problem on engineered features.

A decision tree is a machine learning model that recursively splits data into tree-like structures for classification or prediction. Although easy to interpret due to clear decision splits, they are very prone to overfitting. This can be addressed by combining several trees into so called random forests which reduce variance by averaging over multiple trees trained on bootstrapped samples (Breiman, 2001). In the context of time series, decision trees and random forests can only forecast if lagged values and derived features are explicitly included. Thus, their performance depends strongly on feature engineering and the choice of input window.

Support vector machines employ the use of a hyperplane that separates data points by maximizing the margin while allowing for small misclassifications within a defined threshold (Cortes et al., 1995). They are particularly effective for high-dimensional problems through

nonlinear kernels for complex relationships. In time series tasks, they can be trained on lagged input vectors or transformed time series features.

Gradient boosting machines, and namely XGBoost, have found wide adoption in the data science community, especially in competitions on platforms like Kaggle. They are an extension of the gradient boosting framework by combining several optimization techniques to increase performance and efficiency (Chen & Guestrin, 2016; Friedman, 2001). These models iteratively build ensembles of so-called weak learners like decision trees by fitting to the residuals of previous iterations. Thus, they are able to capture complex structures in the data. In time series forecasting competitions such as the M4 and M5 (Makridakis et al., 2020, 2022), boosting methods with engineered lag features and calendar covariates have consistently ranked among the top-performing approaches.

Overall, these machine learning models are often used as strong benchmarks in time series forecasting tasks. They provide competitive performance with usually straightforward model complexity. Their main limitation lies in the fact that they do not natively model temporal dependence but require careful transformation of the series into lagged features in contrast to deep learning architectures which aim to learn sequential dependencies directly.

2.3 Deep Learning Models

A comprehensive summary of the field of deep learning and neural networks can be found in the work of Goodfellow et al., 2016 and more recently by Prince, 2024.

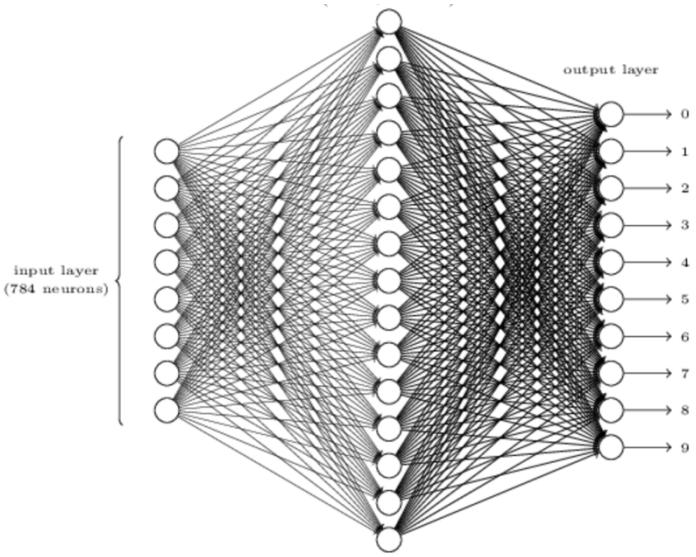
The groundwork on neural networks was laid by McCulloch & Pitts (1943) and Rosenblatt (1958). They introduced the term of the perceptron – an artificial node which mimics neurons in the human brain by capturing only a tiny aspect of the provided data. These perceptrons are connected in multiple layers which is called a multilayer perceptron (MLP). Although their origin was as early as the 1950s, MLPs are still a core component of state-of-the-art language models like GPT-5. Based on the Universal Approximation Theorem, these networks are able to closely model any continuous function (Hornik et al., 1989). However, research initially stalled since they were unable to find a suitable algorithm to train more complex networks. Everything changed when Rumelhart et al. (1986) developed the

backpropagation algorithm which to this day is used to train neural networks. It is the method of how the network automatically improves itself during the training process. (Prince, 2024)

Since these Multilayer Perceptrons are still a vital element of modern language models, this section first introduces standard Multilayer Perceptrons and their terminology through the example of handwritten digit recognition by LeCun et al. (1998) before discussing variants used in time series problems.

2.3.1. Standard Multilayer Perceptron

Figure 2: A neural network (Prince, 2024)



A multilayer perceptron as depicted in figure 2 is made up of several layers of interconnected perceptrons where each perceptron will learn a small aspect of the task at hand. In figure 2, the leftmost layer is called the input layer with one perceptron for each dimension in the input data. Here, the handwritten digits are small greyscale images of 28x28 pixels which means 784 input perceptrons. The rightmost layer is the output layer whose size is defined by the prediction task. For regression tasks, the output layer will only consist of one perceptron while in classification problems it contains one perceptron for each class in the dataset. Thus, for classifying digits, the output layer has ten perceptrons – one for each digit. Since the layers in between are neither input nor output they are called hidden layers with networks usually containing many of them. (Nielsen, 2015)

Figure 2 shows that all perceptrons are connected to all other perceptrons in adjacent layers. The value of one perceptron is calculated as a weighted sum of connected preceding perceptrons plus a bias term. Since this would lead to a model consisting of a series of linear transformations, non-linearity is introduced by passing the weighted sum through an activation

function like the Rectified Linear Unit[1] function. Otherwise, the model would result in one linear transformation regardless of its depth. The values of the weights and biases are parameters which are learned during the training phase of the network. The magnitude of difference between the actual class labels and the predictions of the network is determined by the loss function which is a measure of how well the network parameters are tuned. In time series tasks, common loss functions are the mean squared error or mean absolute error which the training process tries to minimize. The algorithm to tune the parameters based on true labels is called backpropagation. For efficient computation of the necessary changes of the weights and biases during training, the data is divided into mini batches. After all mini batches are processed, one training epoch is completed. Networks are trained over many of these epochs until they converge which means that additional training would only lead to minimal performance improvement. (Nielsen, 2015; Prince, 2024)

2.3.2. Multilayer Perceptron Variants in Time Series

To address the limitations of standard multilayer perceptrons for time series data, several specialized variants were developed.

Recurrent Neural Networks (RNN) introduced by Elman (1990) were designed specifically to process sequential data. In RNNs, the output from a previous time step is fed back as input to the current time step through a hidden state, allowing the model to propagate information across the sequence. However, standard RNNs show notable limitations. For long sequences, they can suffer from the vanishing gradient problem during backpropagation through time. Here, gradients shrink as they propagate backward, preventing information from early time steps from influencing later ones. This severely hinders the model's ability to learn long-term dependencies. Additionally, because each time step's computation depends on the previous one, RNNs must be executed sequentially, making them difficult to parallelize efficiently. This results in high computational cost and slow training times. (Kim et al., 2025)

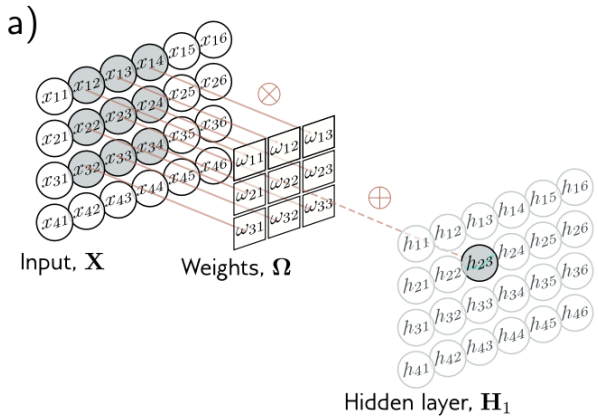
To remedy these issues, the Long Short-Term Memory (LSTM) network was introduced by Hochreiter & Schmidhuber (1997) followed by the simpler Gated Recurrent Unit (GRU) proposed by Cho et al. (2014). Both architectures employ gating mechanisms and memory

[1] $ReLU(x) = max(0, x)$

cells that control the flow of information. This enables the model to retain relevant context over longer horizons while discarding unimportant information which mitigates the vanishing gradient problem.

A different MLP variant emerged with the introduction of convolutional layers alongside the standard fully connected layers. These types of models were first studied by Fukushima (1980) when proposing the Neocogniton model and later popularized by LeCun et al. (1998) with LeNet and Krizhevsky et al. (2012) with the AlexNet architecture.

Figure 3: A convolutional layer (Prince, 2024)



A convolution is a mathematical operation that, in this context, can be understood as applying a kernel (a small matrix of weights) that slides over the input grid. The result is called a feature map where each entry is a weighted sum of neighboring input values as seen in figure 3. Thus, it can capture local features while keeping the number of trainable parameters manageable since the same kernel weights are shared across different regions of the data grid. Initially developed for image recognition, CNNs were subsequently employed for time series problems which can be thought of as a one-dimensional grid.

The success of these MLP variants started a renaissance in both RNN and CNN approaches for time series tasks with many models building on their architecture. However, while successfully adapting the architecture for the vanishing gradient problem, the inherently sequential nature of RNNs still prevented efficient parallelization during training. As a result, models faced memory constraints with longer sequence lengths which limited batch sizes. This limitation was not effectively addressed until the emergence of the transformers architecture. (Vaswani et al., 2017)

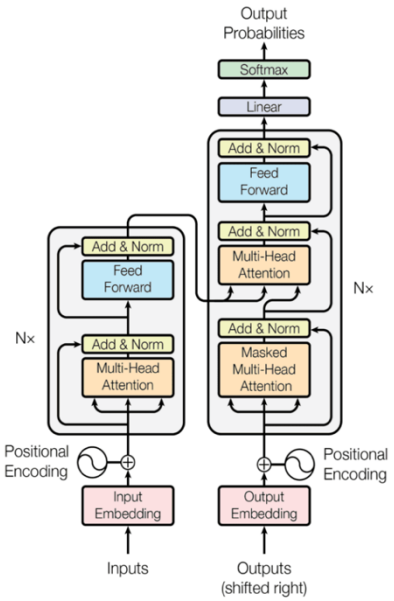
From a chronological perspective, Transformers would naturally be introduced here as the next major advancement in time series modeling. However, since they form the central

research focus of this thesis, transformers and their use cases will first be thoroughly introduced and then discussed in the following sections.

3. Transformers and Large Language Models

A completely different branch of deep learning surrounds the task of language modeling. This area was revolutionized by the 2017 paper Attention Is All You Need which introduced an encoder-decoder architecture called a transformer as seen in figure 4. On a very high level, this branch of models can be segmented into encoder-decoder models as initially presented by Vaswani et al. (2017) and subsequently used for sequence-to-sequence translation tasks and models that only use an encoder or decoder part of the transformer. Encoder-only models are typically used for text understanding or classification models while decoder-only models are used for generation tasks like GPT-3.

Figure 4: Transformer architecture (Vaswani et al., 2017)



At their core, both univariate time series forecasting and language modeling can be framed as sequence prediction problems. In time series analysis, the objective is to predict the next value of a process

$$x_{t+1} \sim p(x_{t+1} \mid x_1, \dots, x_t)$$

given a history of past observations. In language modeling, the similar task is to predict the next token of a sentence

$$y_{t+1} \sim p(y_{t+1} \mid y_1, \dots, y_t)$$

where each token y_t is represented as a high-dimensional embedding vector. Both tasks involve conditioning on sequential context to forecast the next element. Yet, there is a conceptual difference: in univariate time series, each input is originally a scalar, whereas in language models each input is a vector of features which encodes semantic information of the token in

the embedding space. This raises the question if architectures designed to capture contextual dependencies in language can also uncover nonlinear dependencies in time series that go beyond linear correlations.

This section explains how transformers emerged in language modeling and introduces key terminology for the next section to discuss their time series use cases.

3.1 Tokens and Embeddings

Before looking at any type of language model, a way to convert text into a numeric representation must be found. Tokens and word embeddings are a key concept of large language models. A given sentence is broken down into smaller pieces called tokens which consist of words or parts of words. Each token will be represented by a vector in *n*-dimensional space in the embedding. GPT-3 for example uses 12,288 dimensions for its embeddings space.

There are different tokenization methods which share the same goal of finding an efficient set of tokens to encode a text. Initially, full words were used as tokens which comes with the downside of missing embeddings for unseen words in the training set. Even a slight difference like *decoration* vs. *decor* would cause a problem. Subword tokenization attempts to solve this by splitting words into tokens. For example, the noun *decoration* is split into the token *decor* and the token for the suffix *-ation*. One further complexity level are character tokens which facilitate text representation even further but are more difficult to model since for the same input word many more tokens are used than for less complex tokenization. From the model's perspective, text is represented as a sequence of discrete tokens. (Alammar & Grootendorst, 2024)

Finding a vector representation of the tokens is called an embedding. An early example is word2vec by Mikolov et al., 2013. First, a sliding window is used on text documents to create training examples. A window size of two would mean that two neighbor tokens on each side of the central token are being used as neighbors resulting in four training examples. Then, a neural network is trained to classify two tokens as neighbors by giving these training examples a value of 1 and so-called negative examples of random token pairs a value of 0. If the model is incorrect during training, it adjusts the initially random vectors of the two presented tokens, so it improves the probability of correctly classifying future token pairs. This results

in the model settling on an embedding where vector directions keep semantic meaning. Subtracting the vector of the word *woman* from that of *man* will result in a similar vector as subtracting *aunt* from *uncle.* The resulting vector of the subtraction can be interpreted as encoding gender information in the embedding space. The similarity of vectors is represented by their dot product. It is positive when two vectors point in the same direction and 0 when they are perpendicular. Hence, in the embedding, the vectors for *tower* and *dome* will have a high positive dot product. However, this method creates a static embedding space for its tokens. (Alammar & Grootendorst, 2024)

The main improvement of language models with transformers compared to previous attempts of working with language modeling problems is that they do not rely on static vectors for their embeddings. Instead, they can adjust the embedding of a token based on its context. Considering these two sentences:

She opened a new account at the Santander bank in Madrid.

She had a picnic on the bank of Manzanares River.

The word *bank* would have the same embedding vector in a static embedding in both sentences even though its meaning is completely different based on the context. Transformers extract this contextual information and, in this case, would adjust the embedding of bank toward a financial subspace or toward a geographical subspace (e.g. associated with water or Madrid) depending on context.

3.2 Building Blocks of Language Models

A language model like GPT-3 can be understood as systems designed to predict the next token in a sequence given its preceding context. During text generation, each predicted token is appended to the input sequence, and the extended sequence is passed through the model again to produce the next prediction. This autoregressive process continues until an end-of-sequence token is produced and or a predefined length is reached.

The internal workflow of a language model can be summarized as follows: the input text is first decomposed into tokens by a tokenizer which are then mapped into vector embeddings. These embeddings are processed by a stack of transformer blocks, each of which enriches

the representation of a token with contextual information from the surrounding sequence. At the final stage, the hidden representation of each token is projected into the model's vocabulary space. The vocabulary space is the set of all tokens the model can generate which typically consists of tens of thousands of entries. A learned linear transformation (often called the unembedding matrix) maps each contextualized embedding to a vector of raw scores, one for every vocabulary token. These scores are then converted into a probability distribution over the vocabulary, from which the next token is sampled. (Alammar & Grootendorst, 2024)

Each transformer block consists of two principal components: a multi-head attention mechanism, which incorporates contextual information from other tokens in the sequence, and a feedforward neural network, often referred to as a multilayer perceptron (MLP). The attention mechanism enables the model to dynamically weight relevant parts of the input context while the MLP allows the network to store and transform information in higher-dimensional feature space. Together, these components enable the model to generalize beyond seen inputs by interpolating between patterns learned during training. The following subsections introduce these building blocks as seen in figure 4.

3.2.1. Positional Encoding

As transformers do not employ recurrence or convolution, they lack an inherent notion of the sequence order. Hence, positional encodings are added to the token embeddings ensuring that information about the absolute or relative position of the token is available to the model. These encodings have the same dimensionality as the embedding space and can be either learned or fixed.

Kamath et al. (2022) state four desirable properties for these positional encodings:

1. "Unique encoding value for each time-step (word in the sentence).
2. Consistent distances between two time-steps across sentences of various lengths
3. Encoding results are generalized independent of the length of the sentence
4. The encoding is deterministic."

In the original transformer architecture, Vaswani et al. (2017) proposed fixed sine and cosine functions that fulfill these requirements and are defined as:

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

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

with *pos* the position index in the token sequence (rows in the *PE*-matrix) and *i* the embedding dimension (columns in the *PE*-matrix) from 0 to $d_{model}/2 - 1$. Assuming an input representation of $X \in R^{n \times d}$ with a *d*-dimensional embedding for *n* tokens in a sequence, the positional encodings are added to the input matrix as *X* + *PE* using the above defined positional embedding matrix $PE \in R^{n \times d}$ of the same shape. Therefore, each embedding dimension is assigned a sine or cosine wave of a different wavelength ranging from 2π up to $10000 \cdot 2\pi$. (Vaswani et al., 2017). The logic behind these functions is that they generate smooth and continuous signals at different frequencies across the embedding dimensions. Each position in the sequence is mapped to a unique combination of sine and cosine values meaning no two positions share the same pattern. For small *i*, short wavelengths encode short-range position information while for large *i*, long wavelengths encode long-range position information. Since sine and cosine are periodic and orthogonal, their dot products preserve information about relative distances between position. This property enables the attention mechanism to infer if two tokens are close together or far apart, even though the transformer processes all tokens in parallel. (Kamath et al., 2022)

Later architectures mostly adopted learned positional encodings, in which vectors are treated as parameters optimized during training. Vaswani et al. (2017) already noted that learned and fixed encodings yield comparable performance. Most recent large language models have moved beyond additive position encodings and instead employ alternatives such as Rotary Position Embeddings (RoPE). RoPE applies a position dependent rotation to the key and query vectors who will be introduced in the next section. This allows the model to extrapolate more effectively to sequences longer than those seen during training. (Su et al., 2023)

3.2.2. The Attention Layer

The attention mechanism constitutes the core operation of transformer architectures. Its objective is to produce enriched vector representations by weighting information from other tokens in the sequence according to their contextual relevance. The relevant matrices of this

calculation are called the Query (Q) and Key (K) projection matrices and are usually in lower dimensional space than the embedding (e.g. 128 for GPT-3). These matrices contain learnable parameters during training of the model. Given the sentence '*On a sunny day, he walked through Madrid.'* one Query can be interpreted as asking if any adjectives influence the noun *day*. The Query matrix is multiplied with each token vector in the sentence resulting in one vector for each token in lower dimensional Query space which could for example encode the idea of looking for preceding adjectives. The Key matrix is also multiplied with each token and can be seen as the answer to the Query indicating if a token is an adjective. If one token is important for the context of another token, the Query and Key vectors align in their space meaning they have a positive dot product. In this case the dot product of *sunny* and *day* would be high since it directly influences what type of day it is – the embedding of *day* attends to *sunny*. Then, all the dot products of possible token combinations of the sequence are normalized for numerical stability by dividing by the square root of the number of dimensions in the Key space (d_k) and turned into a probability distribution using a softmax function. The softmax function:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

transforms a set of scores $(\mathrm{z}_1, \ldots, \mathrm{z}_\mathrm{n})$ into a probability distribution. By exponentiating the scores and normalizing by their sum, softmax ensures that all outputs are non-negative and sum to one. The probability distribution of the dot product between Query and Key matrix gives a numerical score on how much a previous token is relevant for the current token. This score is multiplied with the Value matrix (V). To adjust the embedding of the current token, the rescaled vectors of the Value matrix are summed up and added to each initial embedding vector. The whole process from Query and Key to Value is called an attention head and represented in the following compact formula.

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

To model different types of contextual updating, not just of adjectives to nouns, one attention block consists of multiple attention heads each with unique Key, Query and Value matrices.

GPT-3 uses 96 attention heads in one layer whose proposed changes are added to the initial embedding of the token. (Alammar & Grootendorst, 2024; Brown et al., 2020)

3.2.3. Multilayer Perceptron

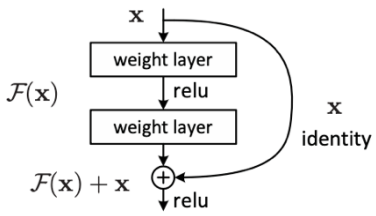
A substantial portion of the trainable weights in large language models such as GPT-3 reside in the feedforward neural networks, also known as multilayer perceptrons (MLPs), which follow the attention layers. The function of the MLP is to transform the contextualized embeddings produced by attention into higher-dimensional representations that capture complex features of the input sequence.

Formally, the input of the MLP is the output of the attention mechanism for each token. The MLP operates independently on each token embedding and typically consists of two linear transformations separated by a non-linear activation function. In GPT-3, the activation function used is the Gaussian Error Linear Unit (GELU), a smooth variant of the Rectified Linear Unit (RELU). Concretely, the embedding is first projected into a much higher-dimensional space (49,152 dimensions in GPT-3), passed through the non-linearity, and subsequently projected back to the original embedding dimension (12,228). This expansion-compression structure allows the network to model complex interactions between features while retaining the original dimensionality for downstream processing. Each neuron in the expanded layer effectively learns to respond to specific patterns in the token representation, enabling the model to encode abstract relationships that go beyond the immediate context captured by attention. Finally, the MLP output for each token is added back to its input, producing a further enriched embedding that is passed to the next layer of the transformer. (Brown et al., 2020)

Having introduced the attention mechanism and feedforward MLP, the following sections will turn to additional architectural components, such as residual connections, normalization and dropout, that are crucial for stable and efficient training of large-scale transformer models.

3.2.4. Residual Connections

Figure 5: Residual Connection (He et al., 2015)



Increasing model depth often improves representational capacity but very deep neural networks are prone to training instabilities such as the vanishing gradient problem or degrading accuracy. He et al. (2015) observed that, in some cases, shallower models actually learn better than their deeper counterparts when optimization becomes unstable. To address this issue, they propose the use of residual connections which add an identity mapping between layers also called skip connections. In practice, the residual block computes the sum of two components: the transformation branch, which applies nonlinear operations, and the skip branch, which passes the input forward unchanged. These residual connections allow for larger gradients to be propagated to earlier layers stabilizing the training of deep networks. The addition operation in residual blocks distributes gradients to both the transformation branch and the skip branch. This guarantees that even if gradients shrink in the main transformation branch, a direct gradient signal still flows through the identity mapping. Residual connections can also be interpreted as a mechanism for adaptively determining effective network depth. Layers that contribute positively to optimization are retained through the transformation branch while layers that do not improve performance can be effectively bypassed. In this sense, skip connections act as a structural regularization technique enabling deep architectures to be trained efficiently without requiring explicit knowledge of the optimal number of layers for a given task. (He et al., 2015)

3.2.5. Normalization Layer

Another technique that facilitates the training of deep neural networks is normalization. In machine learning, preprocessing input data is a standard practice as it often has a strong impact on final model performance. Standardizing features to have zero mean and unit variance is one of the most common techniques. A similar principle applies to neural networks: training with unnormalized activations can hinder convergence under gradient descent. Over the past decade, several normalization methods have been proposed. The most widely adopted are batch normalization by Ioffe & Szegedy (2015) and most recently layer normalization by Ba et al. (2016). Batch normalization computes statistics (mean and variance) across each

feature dimension within a mini batch and it was instrumental in enabling the training of deeper architectures. In contrast, layer normalization normalizes activations across the feature dimension of each input independently which makes it invariant to batch size. It is defined as:

$$y = \frac{x - E[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \cdot \gamma + \beta$$

where *E[x]* and *Var[x]* denote the mean and variance of a single input instance, ϵ is a small constant ensuring numerical stability, and γ, β are learnable scale and shift parameters. Since layer normalization does not rely on batch level statistics, it is highly suited for sequence models like RNNs and Transformers where batch sizes can vary. Therefore, modern large language models almost universally adopt layer normalization instead of batch normalizations. In the original transformer, Vaswani et al. (2017) employed a post-layer normalization setup where layer normalization was applied after the attention and feed-forward sublayers. However, this design was found to cause training instabilities in deeper networks. First analyzed in detail by Xiong et al. (2020), they proposed to instead use a pre-layer normalization setup. This significantly improves optimization stability and has become the de facto standard in architectures like GPT-2, GPT-3 (Brown et al., 2020) and subsequent LLMs. (Zhang et al., 2021)

3.2.6. Dropout Layer

Dropout, proposed by Srivastava et al. (2014), is a widely used regularization technique to prevent overfitting in neural networks. It randomly deactivates a fraction of activations during training which prevents the network from relying too heavily on specific neurons. This allows for redundancy in feature representations and improves generalization on unseen data since at each epoch a slightly different version of the network is trained. Within transformers, dropout is commonly employed in attention weights and feedforward layers which contributes to the overall robustness of the model. (Zhang et al., 2021)

3.2.7. Prediction

GPT-3 consists of 96 transformer blocks stacked on top of each other. Each of the blocks contains a multi-head attention mechanism with 96 attention heads and a feedforward-MLP. During text generation, an input sequence is first tokenized and embedded, then successively processed through all 96 layers, yielding contextualized representations for each position in the sequence. To predict the next token, the representation of the final position in the sequence is projected onto the vocabulary space using an unembedding matrix which produces a vector of raw scores known as logits. The logits are turned into a probability distribution over the vocabulary by applying the softmax function. In practice, always picking the token with the highest probability (greedy decoding) does not lead to the best output for most cases since outputs seem repetitive or lack diversity. Instead, several decoding strategies are employed: temperature scaling modifies the sharpness of the resulting probability distribution which encourages either more deterministic or diverse outputs. Top-k sampling restricts sampling to the k most likely tokens and top-p samples from the smallest set of tokens whose cumulative probability exceeds p. This introduces controlled randomness into the token generation process. The selected token is appended to the input sequence, and the extended sequence is fed back into the model to generate the next token. This autoregressive process continues until an end-of-sequence token is produced or a predefined length is reached. (Zhang et al., 2021)

3.3 Training

Previous chapters discussed the building blocks of a transformer and how a language models generates tokens using trained weights and biases. These parameters are learned in a phase called pretraining which takes up the majority of the computational cost. The objective of training is next-token prediction. Given a sequence of tokens $(x_1, x_2, \dots, x_{t-1})$, the model is trained to maximize the probability of the next token x_t This is typically achieved by minimizing the cross-entropy loss:

$$L = -\log P\left(x_t \mid x_1, x_2, \dots, x_{t-1}\right)$$

which penalizes the model for assigning low probabilities to the correct token and rewards high probability assignments to it. The weights and biases inside of the language model are

then optimized using backpropagation through time combined with stochastic gradient descent and its variants like Adam (Kingma & Ba, 2014). One crucial fact is that no manual labeling is required since training data is drawn from large text corpora like Wikipedia, books or news outlets. Thus, the task is self-supervised since the training labels are simply the next words in the sequence itself. This property enables scaling to massive datasets which is essential for training modern large language models. (Zhang et al., 2021)

After pretraining, models are often further adapted in a stage called fine-tuning. While pretraining exposes the model to a broad knowledge of language and world facts, fine-tuning aligns the model to specific objectives and domains. For example, reinforcement learning from human feedback refines the model to produce outputs that are helpful, safe and aligned with user expectations. Thus, pretraining provides the general linguistic foundation while fine-tuning shapes the model's behavior for targeted applications. (Zhang et al., 2021)

4. Transformer Use in Time Series Forecasting

4.1 Challenges

Adapting transformer architectures to time series forecasting introduces several unique challenges. A primary concern is the computational complexity of self-attention. In the original formulation (Vaswani et al., 2017), self-attention scales quadratically with sequence length, $\mathcal{O}(N^2)$, since every query attends to every key. In real-world forecasting applications such as high-frequency financial markets or energy systems, input sequences can easily extend to tens of thousands of time steps, making standard self-attention prohibitively demanding in terms of both memory and computation.

A second challenge arises from the relative importance of individual tokens. In natural language, most words in a sentence contribute substantially to predicting the next token. In contrast, time series often consist of highly granular observations where successive data points may carry only marginally distinct information. Nevertheless, it is generally not possible to

determine in advance which time lag steps will be most informative for forecasting future values, making adaptive mechanisms essential.

A third challenge concerns the representation of temporal order. Whereas in language modeling positional encodings serve to indicate the sequential order of tokens within a sentence, time series exhibit different structures like seasonality, cyclic behavior and multi-scale dependencies. Standard sinusoidal positional encodings may therefore be insufficient, as they capture only relative token positions within the input window rather than absolute or hierarchical temporal structure.

To address these limitations, various architectural modifications to the transformer have been proposed in recent years. The following literature review surveys key contributions and methodological innovations in adapting transformers to the domain of time series forecasting.

4.2 Literature Review

Several adaptations of the transformer architecture have been proposed to address the specific challenges of time series forecasting outlined above. These approaches can be broadly grouped according to whether they focus on computational efficiency, the selective use of informative tokens or the representation of temporal structure.

LogSparse Transformer (Li et al., 2020): To mitigate the quadratic computational cost of standard self-attention, Li et al. propose a logarithmic sparsity pattern where each query attends only to a subset of keys that are logarithmically spaced across the sequence. This reduces the computational complexity to $\mathcal{O}(N \log N)$ which enables the model to process longer input sequences than the standard transformer. Importantly, this design ensures that both local dependencies (captured by closely spaced keys) and long-range dependencies (captured by logarithmically distant keys) are preserved. This directly addresses the challenge of scalability in long-sequence time series forecasting.

Reformer (Kitaev et al., 2020): The Reformer further reduces complexity through locality-sensitive hashing, which approximates the attention mechanism by grouping tokens with similar representations. This reduces both computational and memory requirements to

$\mathcal{O}(N\ log\ N)$. By efficiently handling long input sequences, the Reformer addresses the scalability challenge while maintaining the model's ability to capture global context.

Informer (H. Zhou et al., 2020): The Informer introduces ProbSparse self-attention, which selects only the most informative queries based on a sparsity criterion that can be computed before constructing the full attention matrix. This significantly reduces the number of pairwise interactions, improving computational efficiency and enabling longer input sequences. By focusing attention on the most relevant parts of the series, the Informer also addresses the challenge of identifying which observations are most influential in predicting future values when token importance is uneven across time steps.

TemporalFusionTransformer (Lim et al., 2019): Unlike the above methods, which primarily address scalability, the Temporal Fusion Transformer (TFT) focuses on incorporating richer temporal structure and interpretability. It integrates static covariates like store location, known future inputs (e.g., holidays, calendar events) and dynamic historical covariates through a combination of attention and gating mechanisms. This allows the model to highlight which variables are most relevant at different forecast horizons, thus addressing both the challenge of representing temporal order and the need for interpretability in real world forecasting tasks.

FEDformer (T. Zhou et al., 2022): FEDformer introduces frequency-domain attention, where dependencies are modeled in the Fourier space rather than directly in the time domain. This not only reduces the computational burden of self-attention but also improves the ability to capture periodicity and seasonal cycles, which are central to many time series domains such as traffic, energy, and weather forecasting. FEDformer therefore directly addresses the challenge of representing multi-scale temporal patterns.

PatchTST (Nie et al., 2022): PatchTST adapts ideas from vision transformers by segmenting contiguous subsequences into patches, which serve as the model's tokens. This reduces the effective sequence length while allowing the model to learn richer local structures within each patch. By compressing the token space, PatchTST addresses computational scalability, and by capturing localized patterns, it improves representation of temporal dependencies beyond those available in pointwise embeddings.

TimeGPT (Garza et al., 2023): In contrast to architectural modifications, TimeGPT represents a paradigm shift toward foundation models for time series. By pretraining a large transformer on diverse temporal datasets before fine-tuning for specific tasks, TimeGPT mirrors the evolution of pretrained models in natural language processing and computer vision. This approach addresses multiple challenges simultaneously: scalability is handled through large-scale pretraining infrastructure, token importance is implicitly learned across heterogeneous datasets and temporal structure is generalized through exposure to a wide range of periodic and non-periodic patterns.

While numerous adaptations of the transformer have been proposed to address scalability, token importance and temporal structure, this thesis adopts a baseline transformer model in order to evaluate its performance in forecasting realized volatility of high-frequency financial time series. This allows to establish a benchmark and assess the suitability of standard transformer architectures in this domain, before considering more specialized variants in future research.

5. Experimental Study

Having introduced the theoretical foundations of transformers and surveyed recent applications to time series forecasting, this chapter presents the experimental study. Specifically, we employ a transformer-based approach to forecast the next-day realized volatility for 28 stocks of the Dow Jones Industrial Average (DJIA). The following sections describe the dataset, preprocessing steps, and experimental setup.

5.1 Dataset

The dataset consists of intraday price data for 28 DIJA stocks spanning January 4th, 2010 to December 31st, 2019 with price observations recorded at one-minute intervals between 9:35:00 am and 3:55:00 pm. This yields 381 observations per day across 2,516 trading days, resulting in 958,596 data points per stock. From the one-minute closing prices, logarithmic returns were computed as input for modeling.

For model development, each stock's time series was divided into training, validation, and testing subsets following a 70%–15%–15% chronological split. The earliest 70% of trading days were assigned to the training set, the subsequent 15% to the validation set, and the most recent 15% to the test set. Chronological splitting ensures that the validation and test sets reflect genuinely unseen data, thereby avoiding information leakage from the future into the past. The validation set was used to monitor the transformer training process, while the test set provided an out-of-sample evaluation against baseline models.

5.2 Realized Volatility

Volatility represents the hidden factor that drives fluctuations in an asset's returns over time. Since it cannot be observed directly, researchers rely on proxy variables that closely model the actual volatility process. In continuous time, the ideal measure of volatility is the quadratic variation of the return process as shown by Andersen et al. (2006). With the increasing availability of high-frequency financial data, it has become possible to approximate this theoretical construct by realized volatility, which can be interpreted as an ex-post measure of the volatility accumulated over a trading day. (Poon & Granger, 2003)

Formally, the realized volatility of day *t* is defined as the square root of the sum of squared intraday log returns:

$$RV_t = \sqrt{\sum_{i=1}^{M} r_{t,i}^2}$$

where *M* is the number of intraday returns per trading day and

$$r_{t,i} = \log(P_{t,i}) - \log(P_{t,i-1})$$

denotes the intraday log return of the stock price series *P*. Here, $P_{t,i}$ is the observed stock price at minute *i* of day *t* ($t = 1, \dots, D;\ i = 1, \dots, M + 1$). Thus, each trading day provides *M* intraday returns from $M + 1$ observed prices. In the dataset used for this thesis, the sample of each stock consists of D = 2,516 trading days with M = 380 intraday returns per day.

As shown by Barndorff-Nielsen & Shephard (2002), realized volatility is an unbiased and, under suitable sampling conditions when $M \longrightarrow \infty$, consistent estimator of actual volatility. It has two important advantages: (i) it can be computed directly from high-frequency returns without requiring an explicit model and (ii) it allows the rich information in intraday data to be incorporated into volatility forecasting (Andersen et al., 2006). For these reasons, realized volatility provides a natural target variable for evaluating the performance of transformer-based forecasting models in this study.

5.3 Preprocessing

The following preprocessing was performed for each stock in the dataset. First, one-minute logarithmic returns were computed from closing prices within each trading day. For day *t*, the intraday log return at minute *m* is defined as

$$r_{t,m} \quad m = 1,2, \dots ,380.$$

Using these intraday returns, the realized volatility of day *t*, denoted RV_t, was calculated according to the definition introduced in Section 5.2.

Next, the dataset was divided into training, validation, and test subsets, and rolling input–output sequences were constructed. Each input sequence consisted of all intraday returns from day $t-1$, with the prediction target being the realized volatility of the following day *t*:

$$x_t = \left(r_{t-1,1},\ r_{t-1,2},\ \dots,\ r_{t-1,380}\right) \ \longrightarrow \ RV_t$$

Following common practice in the volatility forecasting literature (e.g., Andersen et al., 2006; Corsi, 2009), the logarithm of realized volatility was taken as the prediction target. This transformation yields a distribution that is closer to Gaussian and improves training robustness by reducing the impact of extreme volatility spikes. Further, to isolate the model's ability to improve upon the naïve forecast, the model was trained on the residual of the naïve forecast $R_t = RV_t - RV_{t-1}$. However, test errors were almost identical compared to optimizing the model on the direct log RV forecast. This observation will be discussed in section 7.

For the final model, the train and validation sequences from all stocks were concatenated into large datasets, respectively. The intraday return series in each split was standardized using the mean and standard deviation estimated from the training set.

5.4 Model Architecture

The forecasting model is based on a transformer encoder adapted from the original architecture of Vaswani et al. (2017) and implemented as a compact architecture tailored to univariate, high-frequency time series. Let $x_t \in R^{L \times d}$ denote the input sequence for day *t*, where *L* is the number of intraday time steps in the context window and *d* the feature dimension of each token (*d* = 1 for minutely log-returns). Although various combinations of sequence length and input dimensions through patches of return minutes were tested, the lowest test set errors were observed with *L* = 380 and *d* = 1. This means, to forecast the next-day log realized volatility, only the 380 log returns of the previous day are used as input sequence. A table of tested alternative inputs can be found in the appendix.

The model first projects each scalar input to a d_{model}-dimensional latent space via a learnable linear embedding ($d_{model} = 64$), adds learned positional embeddings, and prepends a classification token (CLS) whose hidden state serves as a pooled summary of the sequence. The embedded sequence is then processed by a stack of transformer encoder layers with pre-layer normalization for optimization stability, multi-head self-attention, residual connections and dropout. Concretely, the encoder consists of N = 2 identical layers. Each layer applies (i) multi-head self-attention to contextually reweight token representations, followed by (ii) a feedforward MLP with hidden size $d_{ff} = 256$. A multiplier of $4d_{model}$ of the embedding dimension for the feedforward MLP is recommended by literature (Vaswani et al., 2017). Residual connections and dropout $(p_{drop} = 0.1)$ are used in both sublayers. After the final encoder layer, the hidden state of the CLS token is passed through a prediction head to produce a single scalar output $\widehat{y_t}$, which corresponds to the next-day log realized volatility residual.

5.5 Training

The final model was trained on the pooled multi-stock dataset using the AdamW optimizer with learning rate 10^{-3} and weight decay 10^{-2}, mini batches of size 128, and a maximum of 50 epochs. The loss function is the mean squared error (MSE) between the model output and the target, optimized with mixed precision (AMP) on an NVIDIA Tesla T4 GPU. Training used early stopping with a patience of 10 epochs based on validation MSE and restores the best-performing weights. After concatenation across stocks, the training tensor has 49,280 one day sequences of 380 log return minutes while the validation and test tensors each have 10,528 one day sequences. The training process took 7 minutes and 50 seconds and was ended by early stopping at epoch 20.

6. Results

This section presents the out-of-sample results of the test set. After training the model on all stock data, each stock test set was forecasted individually. Two baselines were employed: A naïve baseline which uses yesterday's value as forecast for today and a HAR model as proposed by Corsi (2009). Formally, the naïve forecast is defined as

$$N_t = RV_{t-1}$$

And the HAR model is built by the three features for daily, weekly and monthly realized volatility as:

$$RV_{t-1}^{(d)} = RV_{t-1}, \quad RV_{t-1}^{(w)} = \frac{1}{5}\sum_{i=1}^{5} RV_{t-i}, \quad RV_{t-1}^{(m)} = \frac{1}{22}\sum_{i=1}^{22} RV_{t-i}$$

The HAR forecast is then generated as:

$$\widehat{RV_{HAR,t}} = \widehat{\beta_0} + \widehat{\beta_d}\, RV_{t-1}^{(d)} + \widehat{\beta_w}\, RV_{t-1}^{(w)} + \widehat{\beta_m}\, RV_{t-1}^{(m)}$$

To not lose days in the test set by having to build the 22-day rolling average first, the HAR features of the test set were built from the validation data in order to have a full forecast of the test period for each stock.

Since the transformer model was trained to predict the residual of the naïve forecast $R_t = RV_t - N_t$, the forecast of the transformer will be $\widehat{R_t}$. To evaluate this prediction against the two baseline predictions, the naïve forecast is added back to the predicted residual to produce $\widehat{RV_{Transformer,t}} = N_t + \widehat{R_t}$.

As evaluation metric, the root mean squared error was used. For each forecast, it was calculated as:

$$\mathrm{RMSE}_{\mathrm{Naive}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(RV_t - N_t)^2} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}R_t^2}$$

$$\mathrm{RMSE}_{\mathrm{HAR}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(RV_t - \widehat{RV_{HAR,t}}\right)^2}$$

$$\mathrm{RMSE}_{\mathrm{Transformer}} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(RV_t - \widehat{RV_{Transformer,t}}\right)^2}$$

with $T = 376$ sequences in the test set for each of the 28 stocks. For each forecasting method, the average RMSE across the 28 stocks is computed as

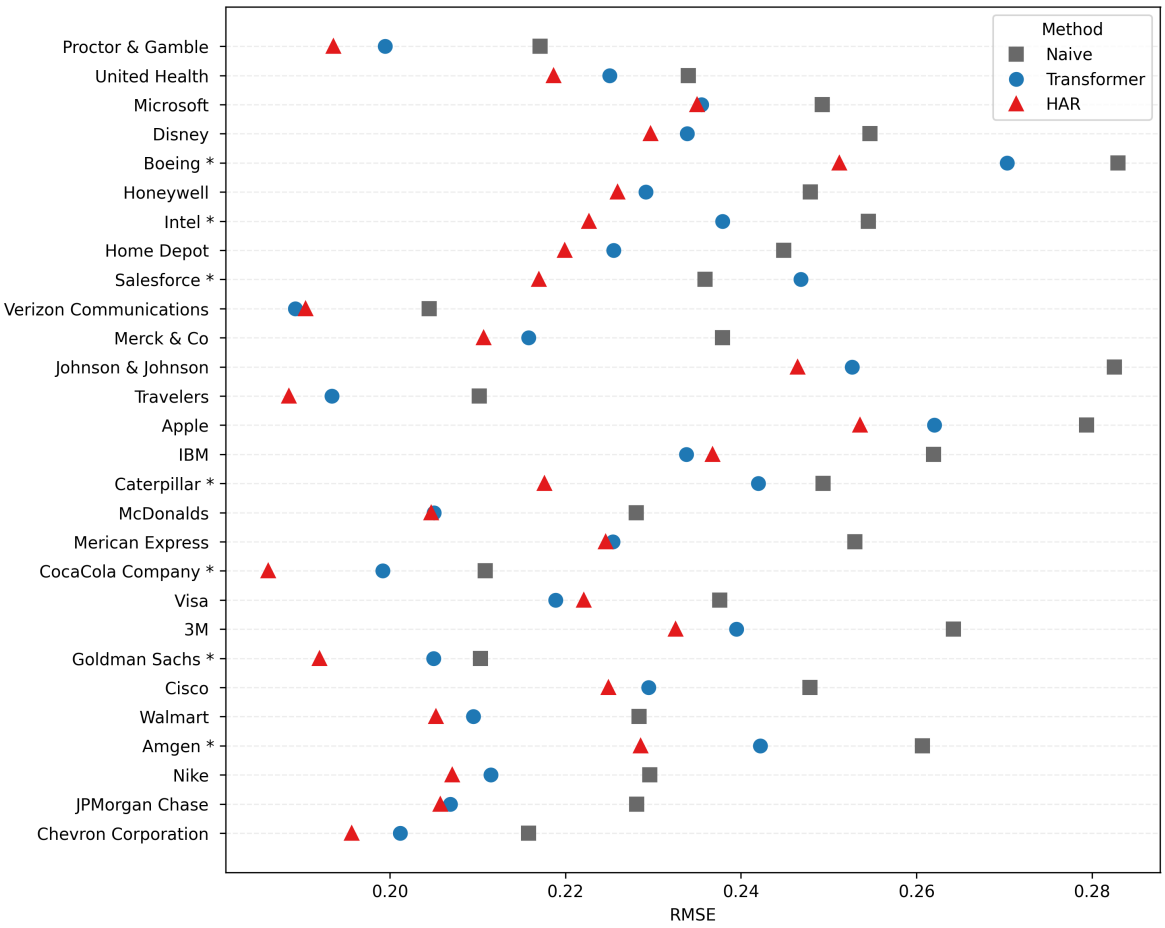
$$\overline{\mathrm{RMSE}} = \frac{1}{28}\sum_{i=1}^{28}\mathrm{RMSE_i}$$

To assess whether differences in forecast errors are statistically significant, the Diebold–Mariano (DM) test is applied for pairwise model comparisons (naïve vs. transformer; HAR vs. transformer). It compares two forecast methods over the entire test horizon by examining the time series of their loss differences. The null hypothesis of the Diebold–Mariano test is equal predictive accuracy: the two forecasts have the same expected loss. Thus, a small p-value suggests a statistically significant difference in accuracy.

6.1 RMSE Out-of-Sample Results

For each of the 28 stocks, three forecasts were computed: Naïve, HAR, and transformer. Performance is evaluated using RMSE on the log realized-volatility scale. Per-stock RMSEs are reported in appendix table 1; their scatter plot appears in figure 6 below. The transformer improves upon the Naïve baseline for 27/28 stocks (the exception is Salesforce). In direct RMSE comparisons against HAR, the transformer is superior for 3/28 stocks.

Figure 6: RMSE out-of-sample results

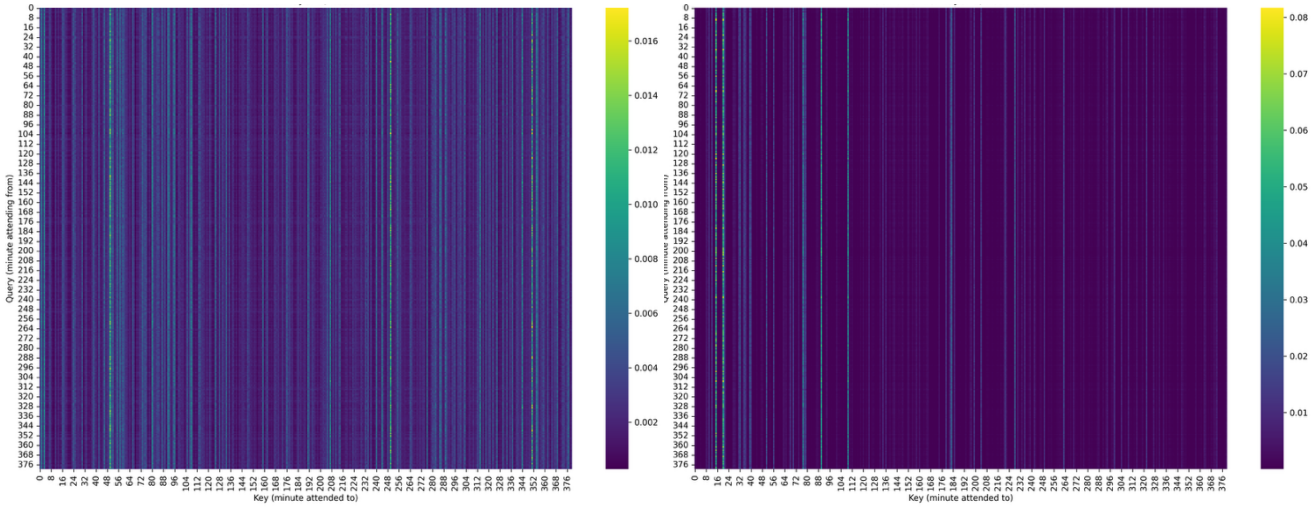


To assess whether these differences in the time series are meaningful rather than noise, Diebold-Mariano tests were conducted per stock (naïve vs. transformer; HAR vs. transformer). Only 7/28 transformer–HAR comparisons show a statistically significant difference at the 5% level. These stocks are marked with an asterisk in figure 6. For naïve-transformer, 23/28 show a statistically significant difference at the 5% level. Full Diebolt-Mariano results appear in the appendix in table 2.

6.2 Attention Maps

To probe how the transformer applies attention to intraday returns, the self-attention scores from the trained model are visualized in a heatmap. For a given layer and head, the attention matrix assigns, for each query minute (rows), a probability distribution over key minutes (columns) after softmax as described in 3.2.2. The heatmaps in figure 7 show for the stock Visa, test day 103 (CLS token omitted, axes index minutes 0–379) the attention scores from the two heads of layer two. Darker or brighter values indicate the relative strength of the attention assigned to a given key by a given query. Note the different scales of attention for each day.

Figure 7: Visa test day 103 attention heatmaps of layer 2, head 1(left) and head 2 (right)



Findings:

- Head 1 of layer 2 (left) exhibits diffuse attention, distributing weight across broad spans of the day. The many vertical bars of attention distributed across many minutes suggests the model is learning repeating intraday patterns (open, midday, close effects across minutes).
- Head 2 of layer 2 (right) is sparser and more selective: attention concentrates on a small set of columns, indicating a few minutes that are especially informative for summarizing the day. These can be interpreted as anchoring minutes which serve as context for the entire day. This specialization suggests complementary roles of one

head providing global context and the other focusing on salient events/spikes on that day.

At the same time, attention maps must not be mistaken with measures of causal or predictive importance. High attention weights do not imply that the corresponding minute directly causes or strongly influences the model's final prediction. Instead, attention weights describe where the model looks for information but not how that information is used (e.g. whether it contributes positively or negatively to the predicted return).

These visualizations highlight a strength unique to transformers in this setting: learned context selection from raw intraday data without hand-crafted features. Further attention heatmaps can be found in the appendix in figures 8 and 9.

7. Discussion and Conclusion

This thesis investigated the use of a transformer-based model to forecast next-day realized volatility from high-frequency intraday returns. The objective was to assess if transformers, originally developed for language modeling, can be effectively applied to this financial context and how they compare against established benchmarks like the HAR model (Corsi, 2009). The results indicate that while the transformer consistently outperformed the naïve benchmark, the predictions of the HAR model showed lower RMSE test errors across stocks. However, most of the forecasts of transformer and HAR do not show a statistically significant difference on a 5% level.

Two factors may explain these findings: First, volatility is known to show strong persistence and volatility clustering (Andersen et al., 2006). Given this property, much of the predictive signal is contained in the previous day's volatility while minute-by-minute returns are highly noisy (Liu et al., 2015). This likely explains why extending the input to multiple days of returns worsened test errors and why more complex architectures with larger embeddings, additional layers or more attention heads did not yield improvements. The similarity in test errors between direct forecasts and residual forecasts further suggests that the transformer

may implicitly perform residual training. In the direct forecast setting, it could have optimized its weights to sum the input minute returns to approximate the naïve forecast, then adjusting based on intraday trading patterns. This mechanism would explain why the model surpasses the naïve benchmark while remaining broadly in line with HAR forecasts which also incorporate long-horizon realized volatility averages as predictors next to the previous day's value.

Second, this thesis highlights limitations of transformers for this specific forecasting task. Although attention matrices can be visualized, their scores do not provide direct measures of feature importance which limits interpretability from an economic perspective. Furthermore, most successful applications of transformers in industry focus on multivariate forecasting and long-term prediction horizons where the flexibility of transformers is most advantageous. In contrast, the univariate, next-day point forecast setting of this thesis may not fully leverage the strengths of this method, leaving traditional statistical models as strong competitors.

The limitations of this thesis lie primarily in its scope and methodological design. First, the hyperparameter search was intentionally restricted and a more systematic exploration may have yielded stronger results. Second, setting up and training transformer models is considerably more complex than implementing HAR models, both in terms of design and computation demand. Although the use of cloud computing resources (Google Colab) reduced training time, practical challenges remain such as managing GPU memory, data loaders and optimizer configurations. Third, the architecture implemented was a basic encoder-only transformer, similar in design to Vaswani et al., (2017), chosen to provide a clean benchmark evaluation. While this allowed for a straightforward comparison, more advanced transformer variants developed specifically for time series forecasting were not examined in depth. Of these, only the idea of patching inputs (Nie et al., 2023) was tested. However, aggregating minute returns into patches of various lengths (5, 10, 20 minutes, see appendix table 3) did not improve forecasts beyond the naïve baseline and was therefore not pursued further.

In conclusion, this thesis provides a case study of applying a transformer to realized volatility forecasting with high-frequency financial data. While the approach improves on the naïve forecasts, it does not consistently outperform the HAR model. Future research could extend the model by incorporating HAR-style covariates as additional input streams or experiment with modern transformer variants tailored for time series.

8. References

Alammar, J., & Grootendorst, M. (2024). *Hands-On Large Language Models: Language Understanding and Generation*. O'Reilly Media

Andersen, T. G., Bollerslev, T., Christoffersen, P. F., & Diebold, F. X. (2006). Volatility And Correlation Forecasting. *Handbook of Economic Forecasting*, *1*, 777–787. https://doi.org/10.1016/S1574-0706(05)01015-3

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). *Layer Normalization*. http://arxiv.org/abs/1607.06450

Barndorff-Nielsen, O. E., & Shephard, N. (2002). Econometric Analysis of Realized Volatility and its Use in Estimating Stochastic Volatility Models. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, *64*(2), 253–280. https://doi.org/10.1111/1467-9868.00336

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics*, *31*(3), 307–327. https://doi.org/10.1016/0304-4076(86)90063-1

Box, G. E. P., & Jenkins, G. M. (1970). *Time Series Analysis: Forecasting and Control*. Holden-Day.

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*(1), 5–32. https://doi.org/10.1023/A:1010933404324

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., … Amodei, D. (2020). *Language Models are Few-Shot Learners*. http://arxiv.org/abs/2005.14165

Chen, T., & Guestrin, C. (2016). *XGBoost: A Scalable Tree Boosting System*. https://doi.org/10.1145/2939672.2939785

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 1724–1734. https://doi.org/10.3115/v1/d14-1179

Corsi, F. (2009). A simple approximate long-memory model of realized volatility. *Journal of Financial Econometrics*, *7*(2), 174–196. https://doi.org/10.1093/jjfinec/nbp001

Cortes, C., Vapnik, V., & Saitta, L. (1995). Support-Vector Networks Editor. In *Machine Leaming* (Vol. 20). Kluwer Academic Publishers.

Eisenach, C., Patel, Y., & Madeka, D. (2022). *MQTransformer: Multi-Horizon Forecasts with Context Dependent and Feedback-Aware Attention*. http://arxiv.org/abs/2009.14799

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211. https://doi.org/10.1207/s15516709cog1402_1

Engle, R. F. (1982). Autoregressive Conditional Heteroscedasticity with Estimates of the Variance of United Kingdom Inflation. *Econometrica*, *50*(4), 987. https://doi.org/10.2307/1912773

Friedman, J. H. (2001). Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, *29*(5), 1189–1232. https://doi.org/10.1214/aos/1013203451

Fukushima, K. (1980). Biological Cybernetics Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position. In *Biol. Cybernetics* (Vol. 36).

Garza, A., Challu, C., & Mergenthaler-Canseco, M. (2023). *TimeGPT-1*. http://arxiv.org/abs/2310.03589

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). *Deep Residual Learning for Image Recognition*. http://arxiv.org/abs/1512.03385

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/NECO.1997.9.8.1735

Hornik, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, *2*(5), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Ioffe, S., & Szegedy, C. (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. http://arxiv.org/abs/1502.03167

Kamath, U., Graham, K. L., & Emara, W. (2022). *Transformers for Machine Learning: A Deep Dive* (1st ed.). Chapman & Hall/CRC. https://www.routledge.com/Chapman--

Kim, J., Kim, H., Kim, H., Lee, D., & Yoon, S. (2025). *A Comprehensive Survey of Deep Learning for Time Series Forecasting: Architectural Diversity and Open Challenges*. http://arxiv.org/abs/2411.05793

Kingma, D. P., & Ba, J. (2014, December 22). *Adam: A Method for Stochastic Optimization*. http://arxiv.org/abs/1412.6980

Kitaev, N., Kaiser, Ł., & Levskaya, A. (2020). *Reformer: The Efficient Transformer*. http://arxiv.org/abs/2001.04451

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *ImageNet Classification with Deep Convolutional Neural Networks*. http://code.google.com/p/cuda-convnet/

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-Based learning Applied to Document Recognition. *Proceedings of the IEEE*, *86*(11).

Li, S., Jin, X., Xuan, Y., Zhou, X., Chen, W., Wang, Y.-X., & Yan, X. (2020). *Enhancing the Locality and Breaking the Memory Bottleneck of Transformer on Time Series Forecasting*. http://arxiv.org/abs/1907.00235

Lim, B., Arik, S. O., Loeff, N., & Pfister, T. (2019). *Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting*. http://arxiv.org/abs/1912.09363

Liu, L. Y., Patton, A. J., & Sheppard, K. (2015). Does anything beat 5-minute RV? A comparison of realized measures across multiple asset classes. *Journal of Econometrics*, *187*(1), 293–311. https://doi.org/10.1016/j.jeconom.2015.02.008

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2020). The M4 Competition: 100,000 time series and 61 forecasting methods. *International Journal of Forecasting*, *36*(1), 54–74. https://doi.org/10.1016/j.ijforecast.2019.04.014

Makridakis, S., Spiliotis, E., & Assimakopoulos, V. (2022). M5 accuracy competition: Results, findings, and conclusions. *International Journal of Forecasting*, *38*(4), 1346–1364. https://doi.org/10.1016/j.ijforecast.2021.11.013

McCulloch, W. S., & Pitts, W. (1943). A Logical Calculus Of The Ideas Immanent In Nervous Activity. *Bulletin Of Mathematical Biophysics*, *5*.

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). *Efficient Estimation of Word Representations in Vector Space*. http://arxiv.org/abs/1301.3781

Müller, U. A., Dacorogna, M. M., Davé, R. D., Olsen, R. B., Pictet, O. V, & Von Weizsäcker, J. E. (1997). Volatilities of different time resolutions-Analyzing the dynamics of market components. In *Journal of Empirical Finance* (Vol. 4).

Nie, Y., Nguyen, N. H., Sinthong, P., & Kalagnanam, J. (2022). *A Time Series is Worth 64 Words: Long-term Forecasting with Transformers*. http://arxiv.org/abs/2211.14730

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press. http://neuralnetworksanddeeplearning.com

Peixeiro, M. (2022). *Time Series Forecasting in Python*.

Peña, D., & Tsay, R. S. (2021). *Statistical Learning for Big Dependent Data* (1st ed.). Wiley.

Poon, S.-H., & Granger, C. W. J. (2003). Forecasting Volatility in Financial Markets: A Review. In *Journal of Economic Literature: Vol. XLI*.

Prince, S. J. D. (2024). *Understanding Deep Learning*. http://udlbook.com.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533A0

Srivastava, N., Hinton, G., Krizhevsky, A., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. In *Journal of Machine Learning Research* (Vol. 15).

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2023). *RoFormer: Enhanced Transformer with Rotary Position Embedding*. http://arxiv.org/abs/2104.09864

Vaswani, A., Brain, G., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention Is All You Need*.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). *On Layer Normalization in the Transformer Architecture*. http://arxiv.org/abs/2002.04745

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). *Dive into Deep Learning*. https://arxiv.org/abs/2106.11342

Zhou, H., Zhang, S., Peng, J., Zhang, S., Li, J., Xiong, H., & Zhang, W. (2020). *Informer: Beyond Efficient Transformer for Long Sequence Time-Series Forecasting*. http://arxiv.org/abs/2012.07436

Zhou, T., Ma, Z., Wen, Q., Wang, X., Sun, L., & Jin, R. (2022). *FEDformer: Frequency Enhanced Decomposed Transformer for Long-term Series Forecasting*. http://arxiv.org/abs/2201.12740

Appendix

Table 1: RMSE Out-of-Sample Results

	RMSE		
Stock	Naive	Transformer	HAR
Proctor & Gamble	.2171	.1994	.1936
United Health	.2340	.2251	.2187
Microsoft	.2493	.2355	.2350
Disney	.2547	.2339	.2297
Boeing	.2829	.2703	.2512
Honeywell	.2479	.2291	.2259
Intel	.2545	.2379	.2227
Home Depot	.2449	.2255	.2199
Salesforce	*.2359*	.2468	.2170
Verizon Communications	.2045	**.1892**	.1904
Merck&Co	.2379	.2158	.2107
Johnson & Johnson	.2826	.2526	.2465
Travelers	.2102	.1934	.1885
Apple	.2794	.2621	.2536
IBM	.2619	**.2338**	.2368
Caterpillar	.2493	.2420	.2176
McDonalds	.2281	.2050	.2047
American Express	.2530	.2254	.2246
CocaCola Company	.2108	.1992	.1861
Visa	.2376	**.2189**	.2221
3M	.2642	.2395	.2326
Goldman Sachs	.2103	.2050	.1920
Cisco	.2479	.2295	.2249
Walmart	.2284	.2095	.2053
Amgen	.2607	.2422	.2286
Nike	.2296	.2115	.2071
JPMorgan Chase	.2281	.2069	.2058
Chevron Corporation	.2158	.2012	.1957
Average	.2415	.2245	.2174

Note: Naïve errors with better performance over the transformer in italic while transformer errors with a better performance over the HAR model in bold.

Table 2: Diebold-Mariano Test Results

	Naive-Transformer		HAR-Transformer	
Stock	Test Statistic	p-value	Test Statistic	p-value
Proctor & Gamble	4.268	.00003*	-1.290	.198
United Health	1.431	.153	-.947	.344
Microsoft	2.693	.007*	-.095	.924
Disney	3.886	.0001*	-1.029	.304
Boeing	1.532	.126	-2.859	.004*
Honeywell	3.947	.0001*	-.778	.437
Intel	2.432	.015*	-2.850	.005*
Home Depot	3.614	.0003*	-1.411	.159
Salesforce	-1.462	.144	-4.335	.00002*
Verizon Communications	3.037	.003*	.329	.742
Merck & Co	4.372	.00002*	-1.156	.248
Johnson & Johnson	4.518	.00001*	-1.358	.175
Travelers	3.547	.0004*	-1.183	.238
Apple	2.769	.006*	-1.960	.051
IBM	3.659	.0003*	.525	.600
Caterpillar	1.194	.233	-4.110	.00005*
McDonalds	4.445	.00001*	-.084	.933
American Express	4.154	.00004*	-.180	.857
CocaCola Company	2.879	.004*	-3.171	.002*
Visa	3.716	.0002*	.650	.516
3M	3.310	.001*	-1.511	.132
Goldman Sachs	1.132	.258	-2.767	.006*
Cisco	2.917	.004*	-1.010	.313
Walmart	3.800	.0002*	-.898	.370
Amgen	2.725	.007*	-2.456	.014*
Nike	3.076	.002*	-1.012	.312
JPMorgan Chase	4.807	0.00000*	-.288	.773
Chevron Corporation	3.380	.001*	-1.421	.156

Note: p-values smaller than 0.05 are marked with an asterisk

Table 3: Hyperparameter testing

Stocks per test	Stock	Calendar patches	Input patch length:days	HAR patches	Standardize returns	Parameters (d_inp, nl, nh, bs)	Training Time	Target variant	Naive MSE	Test MSE	Predicted Test Mean	Notes and decisions
1	WMT	False	1:1	False	False	128, 4, 3, 64	9.8 seconds	Direct log RV	0.2105	0.3955	Yes	
1	WMT	False	1:1	False	False	512, 8, 8, 64	3, 42 minutes	Direct log RV	0.2105	0.4073	Yes	
1	WMT	False	5:1	False	True	512, 8, 8, 64	1,06 minutes	Direct log RV	0.2105	0.2478	No	Standardize input returns
1	WMT	False	10:10	False	True	128, 4, 3, 64	32 seconds	Direct log RV	0.2105	0.2746	No	
1	WMT	False	20:5	False	True	128, 4, 3, 64	38 seconds	Direct log RV	0.2105	0.2229	No	
1	WMT	False	20:5	False	True	512, 8, 8, 64	42 seconds	Direct log RV	0.2105	0.2563	No	
1	WMT	False	1:1	False	True	512, 8, 8, 64	1,53 minutes	Direct log RV	0.2105	0.2009	No	Use unpatched minute returns as input (1:x input)
1	WMT	False	1:2	False	True	512, 8, 8, 64	6,55 minutes	Direct log RV	0.2105	0.1798	No	
1	WMT	False	1:3	False	True	512, 8, 8, 64	15,3 minutes	Direct log RV	0.2105	0.2157	No	
1	WMT	False	1:2	False	True	512, 8, 8, 64	7,32 minutes	Direct log RV	0.2105	0.1733	No	
1	WMT	True	1:2	False	True	512, 8, 8, 64	7,54 minutes	Direct log RV	0.2105	0.1979	No	Don’t use calendar features with minute returns as patches
1	WMT	False	1:2	1,5,22	True	512, 8, 8, 64	7,46 minutes	Direct log RV	0.2105	0.2094	No	
1	WMT	False	1:2	1,5,22	True	64, 2, 2, 64	18 seconds	Direct log RV	0.2105	0.1803	No	Don't use HAR features with minute returns as patches
1	WMT	False	1:2	False	True	64, 2, 2, 64	53 seconds	Direct log RV	0.2105	0.1706	No	Use smaller model architecture
1	WMT	False	1:2	False	True	64, 2, 2, 128	52 seconds	Direct log RV	0.2105	0.1730	No	
1	WMT	False	1:2	False	True	64, 2, 2, 128	1, 20 minutes	Residual log RV	0.2105	0.1742	No	Both target variants yield similar test errors
5	WMT	False	1:1	False	True	64, 2, 2, 128	3, 22 minutes	Residual log RV	0.2105	0.1733	No	Mean Naive MSE: 0.2283; Mean Test MSE: 0.1877
	CV								0.1841	0.1599	No	
	3M								0.2800	0.2200	No	
	JPM								0.2082	0.1688	No	
	INT								0.2591	0.2167	No	
5	WMT	False	1:2	False	True	64, 2, 2, 128	6, 48 minutes	Residual log RV	0.2105	0.1688	No	Mean Naive MSE: 0.2283; Mean Test MSE: 0.1908
	CV								0.1841	0.1588	No	Use 1:1 input for all-stock large model
	3M								0.2800	0.2322	No	
	JPM								0.2082	0.1784	No	
	INT								0.2591	0.2158	No	

Note: This table shows the input and hyperparameter variants tested before training the large transformer on all 28-stock data.

Stocks: WMT = Walmart, CV = Chevron Corporation, 3M = 3M, JPM = JPMorgan Chase, INT = Intel

Calendar: It was evaluated to add a calendar encoding as feature next to minute returns into one patch token. These encoded the day, week and month of the input sequence as features.

Input: States the input shape of returns as patch length:preceding days. Patch length = 1 corresponds to minutely input returns per day while patch length = 5 would mean 5 minutes are aggregated in one input patch token

HAR patches: It was evaluated to add HAR similar covariates into each minute patch token meaning rolling 5 day and 22 day realized volatility means.

Standardize returns: whether to standardize the input returns. Without doing so, the model always predicted the train set RV mean for every day in the test set sequence (See column Predicted test mean)

Parameters: Model hyperparameters in order of: input dimension, encoder layers, attention heads, batch size. All listed variants were tested with learning rate e-3, epochs 50 and early stopping 5

Naïve MSE, Test MSE: It has to be noted that the reported errors here are the MSE of the target being the log realized variance. For the final model and in the report, to better align with existing literature, the RMSE metric and log realized volatility as target was chosen.

Figure 8: Apple test day 349, attention heatmap of layer 2, head 1

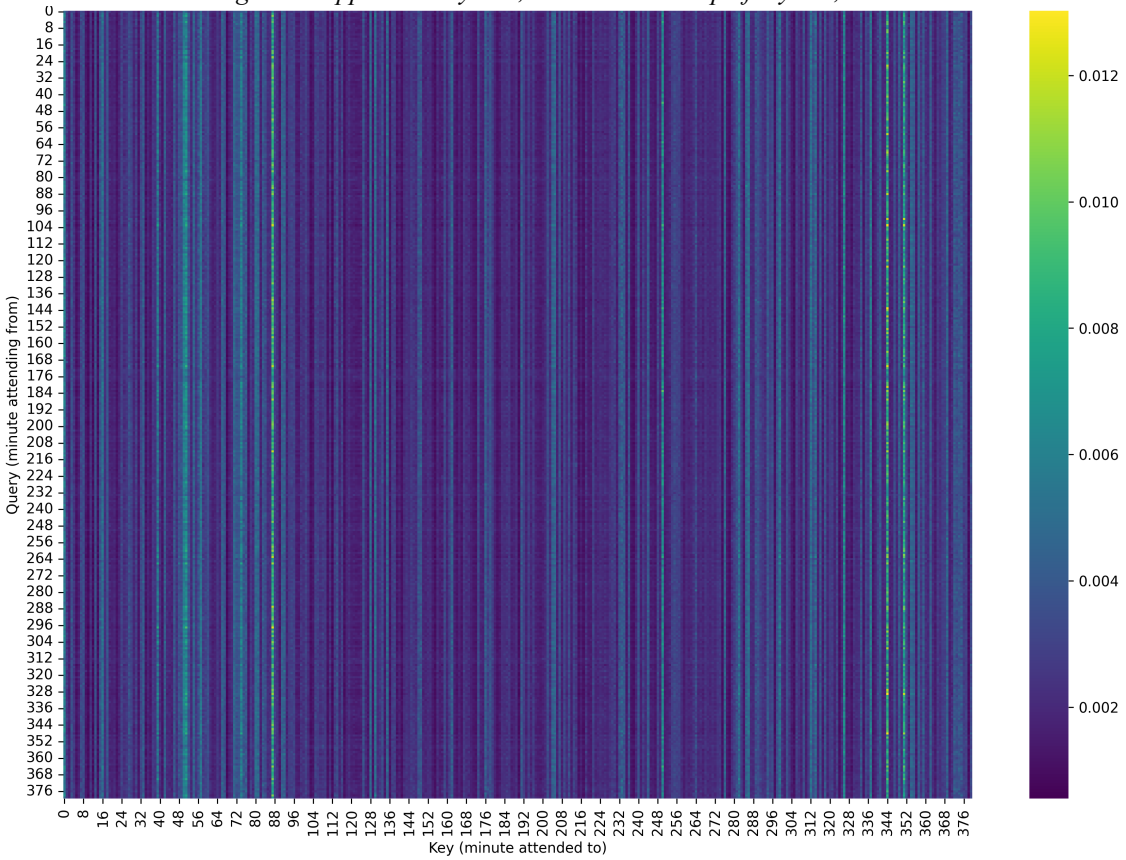


Figure 9: Apple test day 349, attention heatmap of layer 2, head 2