University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Damian Dąbrowski
Student no. 439954

Heorhii Lopatin
Student no. 456366

Ivan Gechu
Student no. 439665

Krzysztof Szostek
Student no. 440011

Large language models for forecasting market behaviour

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Piotr Hofman
Instytut Informatyki

Warsaw, May 2024

Abstract

This thesis concerns research into the use of machine learning and large language models in market analysis, focusing on market predictions.

Keywords

machine learning, large language models, time series forecasting, market prices

Thesis domain (Socrates-Erasmus subject area codes)

11.4 Artificial Intelligence

Subject classification

Computing methodologies $\rightarrow$ Neural networks

Tytuł pracy w języku polskim

Duże modele językowe w przewidywaniu zmian rynkowych

Contents

Chapter 1

Introduction

1.1. Overview

In the world of stock markets a major problem is the apparent incalculability of the complex network of factors e.g. how stock prices of one company affect those of another. As the environment of stock markets becomes more and more complex, the ability to analise and confidently predict its future becomes of crucial importance for traders, investors and researchers.

With the recent advent of generative AI and the demonstrable power of Large Language Models a question arises of if and how these can be used to accurately analise and predict time series market prices in different environments. This thesis presents our work on the subject.

The main technical challenge is the following.

Given a sequence of historical observations $X \in \mathbb{R}^{N \times T}$ consisting of N different 1-dimensional variables across T time steps, we aim to reprogram a large language model $f(\cdot)$ to understand the input time series and accurately forecast the readings at H future time steps, denoted by $\hat{Y} \in \mathbb{R}^{N \times H}$, with the overall objective to minimize the mean square errors between the expected outputs Y and predictions, i.e., $\frac{1}{H} \sum_{h=1}^{H} \|\hat{Y}_h - Y_h\|_F^2$. [17]

1.2. Outline

First we describe what datasets we were using in our experiments and how we measured the results.

Then we look at how different, simpler machine learning models deal with time series prediction.

We then describe how an LLM works and how it might be used for our purposes.

Subsequently, we discuss our own methodology; different aplied methods and techniques of input reprogramming and the use of prompts and context.

Next, we present the results we have achieved on the chosen datasets (and compare them to some other known solutions).

Finally, we speculate on the significance of our work, its potential applications in forecasting price time-series.

Chapter 2

Preliminary definitions & guidelines

2.1. Macros

2.2. Notation

- By *goal* or *problem* or *task* we mean the overall task of the thesis, which is develop a machine learning model suitable for predicting prices on the financial market, based on a history of data.

2.3. Datasets

Here we describe the various time series datasets we used for the training and testing of our models. In the below table, the columns describe the following:

- **Name:** the name of the dataset.
- **Source:** the source from where we took the dataset - most we received from AI Investments company. For others we include a link or a way to access and download the dataset.
- **Number of datapoints:** a description of how many datapoints the dataset contains, how many features each datapoints has and the overall size of the file.
- **Datapoints used:** how many of these datapoints were used in model training (these are taken from the end of the dataset).
- **Time step:** what is the time difference in the Date column between subsequent datapoints.

Datasets summary					
Dataset name	Short description	Source	Number of datapoints	Datapoints used	Time step
AAPL	Apple stock prices	Online [1]	10943	10000	1 day
BTCUSD	Rates of Bitcoin in US Dollars	AI Investments	40450	10000	1 hour
EURUSD	Rates of Euro in US Dollars	AI Investments	117397	10000	1 hour
GBPCAD	Rates of GB Pound in Canadian Dollars	AI Investments	117423	10000	1 hour
GBPTRY	Rates of GB Pound in Turkish Lires	AI Investments	35965	10000	1 hour
US500	US500 stock index	AI Investments	118023	40000	1 hour
Electricity	Electricity consumption	Online [2]	26304	20000	15 minutes

2.3.1. Explanation of the datasets

All except for the Electricity dataset are market time series - they have the following features:

- **date** - The time at which the datapoint was recorded.
- **close** - Closing price within given time interval.
- **high** - Highest price within given time interval.
- **low** - Lowest Price within given time interval.
- **open** - Opening price within given time interval.
- **volume** - Volume of stock traded.
- **adjClose** - Closing price within given time interval, modified to account for dividends, stock splits, etc., to better reflect stock value.

Our target column was the **close** column. All others were discarded.

The Electricity dataset in addition to the **date** column has 370 columns, which correspond to different energy consumption clients. Every datapoint records the energy consumption for the timestep period at each of these clients. For the target column, we chose the 127th column. This dataset serves to check whether the approach to market time-series generalises to other types of time-series.

In all experiments, the dataset used was divided into 3 parts: the first 70% was used in the testing of the model, then 10% was used in its validation, and 20% was used for testing.

2.4. Metrics

2.4.1. Mean Squared Error

The *Mean Squared Error* (MSE) is a metric used during training of machine learning model, especially in regression tasks. It quantifies how close a model's predictions are to the actual values.

Definition

MSE calculates the average of the squares of the errors. The error is the difference between the values ($\hat{y}_i$) predicted by the model and the actual values (y_i).

Calculation Steps

The metric is computed as follows.

1. For each prediction the error is computed by subtracting the predicted value from the actual value.
2. Each error is then squared in order to ensure that positive and negative errors do not cancel each other out and in order to emphasize larger errors.
3. The average of these squared errors is then computed to obtain the MSE.

Mathematical Formulation

The mathematical formula for MSE is given by:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{2.1}$$

where:

- n is the number of data points,
- y_i is the actual value for the ith datapoint,
- $\hat{y}_i$ is the predicted value for the ith datapoint,

Interpretation

- MSE is a non-negative number where a value of 0 indicates perfect predictions.
- Larger MSE values indicate worse model performance.
- MSE emphasizes larger errors due to the squaring of each term, which can be both advantageous and disadvantageous depending on the application.

2.4.2. Accuracy metric

The *Accuracy* metric measures correctness of class predictions of a model. In the context of time series prediction, it is measured as follows.

When we predict the value a_{i+k} based on values $[a_{i-n}, a_i]$ in a time series, we

1. classify the prediction $\hat{a}_{i+k}$ into a binary up/down class, based on whether the predicted value is higher or lower then the last known one:

$$\mathsf{class}(\hat{a}_{i+k}) = \begin{cases} 1, & \text{if } \hat{a}_{i+k} \geq a_i \\ 0, & \text{otherwise} \end{cases}$$

2. We then classify the value actually observed a_{i+k} in the same way.

3. Subsequently we calculate the fraction of the times when such predictions were correct, i.e. $\mathsf{class}(\hat{a}_{i+k}) = \mathsf{class}(a_{i+k})$.

4. This yields an *accuracy* score, between 0 and 1: $0 \leq \frac{\sum_{i=1}^{N}[\mathsf{class}(\hat{a}_{i+k})=\mathsf{class}(a_{i+k})]}{N} \leq 1$. [1]

[1] Here for N predictions.

Chapter 3

Other models

Here we describe four simpler machine learning models we used for comparison with LLM results: Linear Regression, Multilayer Perceptron, Convolutional Neural Network, Residual Neural Network. The descriptions include a short, technical overview of how the models work. In chapter 5 we present the results of trying to use these models to extrapolate a time series.

In the below descriptions, **overfitting** refers to a phenomenon when model learns from the training data too closely or exactly, thereby making it less generalisable to new data. See figure Figure 3.1 for an illustration of the phenomenon. Figure and description are both from Wikipedia [3].

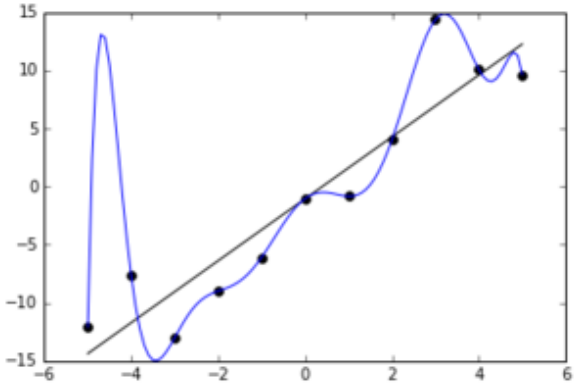

Figure 3.1: Noisy (roughly linear) data is fitted to a linear function and a polynomial function. Although the polynomial function is a perfect fit, the linear function can be expected to generalize better: if the two functions were used to extrapolate beyond the fitted data, the linear function should make better predictions.

3.1. Linear regression

Linear regression [4] is a statistical method used for predicting the value of a continuous outcome variable based on one or more input features. It models the relationship between the input features and the continuous outcome by fitting a linear equation to observed data.

In the context of predicting time series, an input vector x_i is a subseries of datapoints of length seq_len, for which the output y_i is a prediction of the actual value of the datapoint following x_i. Therefore, in the description below, the number of features of an input vector is $d = seq\_len \cdot k$, where k is the number of features of an individual datapoint.

With that in mind, the general formulation of the linear regression model is as follows:

Given a set of n observations $\{(x_i, y_i)\}_{i=1}^n$, where $x_i \in \mathbb{R}^d$ represents the feature vector and $y_i \in \mathbb{R}$ represents the continuous outcome, the linear regression model predicts the value of y as

$$\hat{y} = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_d x_{i,d},$$

where β_0 is the intercept term, and $\beta_1, \beta_2, \ldots, \beta_d$ are the coefficients corresponding to the d features.

The model is trained by minimizing the residual sum of squares (RSS), which measures the discrepancy between the observed values y_i and the predicted values $\hat{y}_i$. The RSS is given by

$$RSS(\beta) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} \left(y_i - (\beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + \cdots + \beta_d x_{i,d})\right)^2.$$

To find the optimal parameters β, the RSS is minimized using analytical methods such as the normal equation or numerical optimization techniques such as gradient descent.

3.2. Multilayer Perceptron

The *Multilayer Perceptron* (MLP) [6] is a *feedforward neural network*, that is made up of multiple layers of nodes in a directed graph, where each node from one layer is connected to all the nodes from the previous one. MLPs are widely used in pattern recognition, classification, and regression problems due to their ability as networks to model complex nonlinear relationships in the input data. An MLP consists of an input layer of neurons, one or more hidden layers, and an output layer. Each node, except for those in the input layer, is a neuron that uses a nonlinear activation function to combine inputs from the previous layer and an additional *bias term*.

3.2.1. Structure of an MLP

An MLP is made up of the following components:

- **Input Layer:** The first layer of the network, which receives the input data to be processed. Each neuron in this layer represents a feature of the input data.
- **Hidden Layers:** One or more layers that perform computations on the inputs received and pass their output to the next layer. The neurons in these layers apply activation functions to their inputs to introduce nonlinearity.
- **Output Layer:** The final layer that produces the output of the network.

3.2.2. Forward Propagation

The process of computing the output of an MLP is called forward propagation. In this process, the input data is passed through each layer of the network, transforming the data as it moves through. The output of each neuron is computed as follows:

$$a_j^{(l)} = \phi\left(\sum_i w_{j,i}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right), \tag{3.1}$$

where

- $a_j^{(l)}$ is the activation of the j-th neuron in the l-th layer;
- ϕ denotes the activation function, e.g. ReLU (section 3.3.1), softmax;
- $w_{j,i}^{(l)}$ represents the weight from the i-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer;
- $b_j^{(l)}$ is the bias term for the j-th neuron in the l-th layer;
- $a_i^{(l-1)}$ is the activation of the i-th neuron in the $(l-1)$-th layer.

3.2.3. Backpropagation and Training

To train an MLP, the backpropagation algorithm is used. This algorithm adjusts the weights and biases of the network to minimize the difference between the actual output and the expected output. The process involves computing the gradient of a loss function with respect to each weight and bias in the network, and then using these gradients to update the weights and biases in the direction that minimizes the loss. The loss function measures the error between the predicted output and the actual output. The update rule for the weights is given by

$$w_{j,i}^{(l)} \leftarrow w_{j,i}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{j,i}^{(l)}}, \tag{3.2}$$

where

- η is the learning rate. If it is too small, the model will train very slowly and may get stuck in local minima. If it is too large, it might not converge.
- $\frac{\partial \mathcal{L}}{\partial w_{j,i}^{(l)}}$ is the partial derivative of the loss function $\mathcal{L}$ with respect to the weight $w_{j,i}^{(l)}$.

Similar updates are made for the biases.

Through iterative training involving forward propagation, loss calculation, and backpropagation, the MLP learns to approximate the function that maps data inputs to desired predictions.

3.3. Convolutional neural network

Convolutional Neural Networks (CNNs) [7] are a class of deep neural networks, highly effective for analyzing visual imagery. They employ a mathematical operation called convolution, which allows them to efficiently process data in a grid-like topology, such as images.

3.3.1. Architecture of Convolutional Neural Networks

A typical CNN architecture comprises several layers that transform the input image to produce an output that represents the presence of specific features or class labels. The layers found in a CNN are:

Convolutional Layer

The convolutional layer applies a set of learnable filters to the input. Each filter activates specific features at certain spatial positions in the input. Mathematically, the convolution operation of each filter is defined as follows:

$$f(x,y) = (g * h)(x,y) = \sum_c \sum_m \sum_n g_c(m,n) \cdot h_c(x-m, y-n)$$

where $f(x,y)$ is the output, g is the filter, h is the input image, and $*$ denotes the convolution operation. x and y range over output image dimensions; m and n range over the filter dimensions.

Following convolution, an **activation function** is applied to introduce non-linearity into the model. The Rectified Linear Unit (ReLU) is commonly used:

$$\phi(x) = \max(0, x).$$

For time series forecasting, we use 1D-convolutional neural network, which means that filter has 1 row.

Pooling Layer

The pooling layer reduces the spatial dimensions (width and height) of the input volume for the next convolutional layer.

Fully Connected Layer

Towards the end of the network, fully connected layers are used, where each input node is connected to each output by a learnable weight.

A simple diagram of the layers can be seen on figure Figure 3.2 (from [8]). [1]

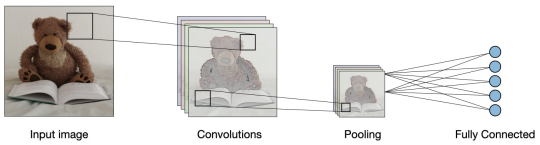

Figure 3.2: A simple diagram illustrating the different layers of the network working on an example input image.

3.4. Residual Neural Network

A Residual Neural Network (ResNet) is a type of deep learning model specifically designed to mitigate the vanishing gradient problem, where through backpropagation only the last few layers of the network get trained. ResNets introduce skip connections, also known as residual connections, which allow the input of a layer to be directly added to the output of a subsequent layer. This is mathematically expressed as

$$\mathbf{y} = \mathcal{F}(\mathbf{x}, \{W_i\}) + \mathbf{x},$$

[1] In the image: the result of the convolution layer is a stack of output images, one for each filter.

where $\mathbf{y}$ is the output of the layer, $\mathcal{F}$ represents the residual mapping to be learned by the layer, $\mathbf{x}$ is the input, and $\{W_i\}$ denote the weights of the layers.

The primary advantage of this architecture is its ability to facilitate the training of deeper networks by preserving the gradient flow through the network during backpropagation. This is achieved by allowing the backpropagation to bypass one or more layers, reducing the risk of gradient vanishing (where update gradients become exponentially small during backpropagation, slowing down training). As a result, ResNets can be trained to much depths than CNNs, and can have more layers.

Chapter 4

Large Language Model

In this chapter we present the basic theory behind Large Language Models. We then introduce the LLaMA family of models, which we've been using. Subsequently, we describe the embedding technique we've used and we present our results.

4.1. Vocabulary

The following vocabulary is used:

- **Token** is a basic unit of text data a language model processes - usually words, subwords, punctuation marks etc.
- **Tokenization** is a process of breaking down input data into tokens.
- **Language model** is a *probabilistic model* of a natural language. Probabilistic - meaning that given some input text data, it's job is to predict the future token. [11]
- **Supervised learning** is a type of a method of training machine learning models where every input is supplied with the output the model is expected to produce. The model then matches its own output with the expected output to correct its own behaviour.
- **Pretraining** is a process of training the language model on a corpus of data
- **Fine-tuning** is a process of adapting a pretrained language model to a specific task (e.g. mathematics, poetry) by training it on a smaller, task-specific dataset.
- **Token embedding** is a mapping of tokens to high-dimensional vectors of real numbers. This mapping is expected to have the property that tokens similar in meaning are close in the output space. See [10].
- **Context length** is the maximal size of input tokens a large language model can process at any one time.
- **Cross-modality data** is data that combines multiple modalities, i.e. text, image, audio, video, etc. In particuliar, combination of text description and time series is cross-modality data.

4.2. Overview

A Large Language Model (LLM) [12] is a language model that is pretrained on a large collection of data (usually billions of tokens). These models utilize deep learning techniques, particularly neural networks, which consist of interconnected neuron layers that process information sequentially, one by one. The predominant architecture underpinning most contemporary LLMs is the Transformer (see below), notable for its self-attention mechanism that enables the model to assess the importance of different tokens in a sentence irrespective of the order the tokens are in.

The training of an LLM involves pretraining it with a large, diverse corpus of text, during which it adjusts its internal parameters to minimize the difference between its predictions and the actual data. This process of supervised learning equips the model with a probabilistic understanding of the language (its patterns, semantics, syntax, knowledge of the world inherent in a language), enabling it to predict a continuation of a given piece of input text.

Once trained, LLMs can perform a variety of language-based tasks such as translation, summarization, question answering, and text generation. It can then be fine-tuned to perform better on a specific task, e.g. write poetry, give cooking advice, write programming code, etc. We will see how such a model deals with analysing and predicting time series data.

4.3. The Transformer

The Transformer is a type of neural network architecture introduced in the seminal 2017 paper *"Attention is All You Need"* by Vaswani et al. [13]. It has since become foundational for many natural language processing (NLP) models due to its efficiency and effectiveness in handling data sequences, such as text.

4.3.1. Components

The Transformer (fig Figure 4.1) consists of the following components:

1. **Input Embedding**: Converts input tokens into vectors.
2. **Positional Encoding**: Adds positional information to the embeddings to retain the order of the sequence. This encoding is another high-dimensional vector.
3. **Encoder** (Figure 4.1 the left):
   - Consists of $N = 6$ layers[1].
   - Each layer has two sub-layers:
     - **Multi-Head Attention**: Applies attention mechanism over the input. For each token calculates the weight or importance of over tokens in the surrounding context (*Attention*). Does so independently with h 'heads'[2], each calculating different semantic relantionships (*Multi-Head*). The results are concatenated and linearly transformed into size of one output (as if from one head).
     - **Feed Forward**: Applies a fully connected feed-forward neural network.
   - **Add & Norm**: Residual connections followed by layer normalization after each sub-layer.

[1] [13], section 3.1
[2] In [13], $h = 6$

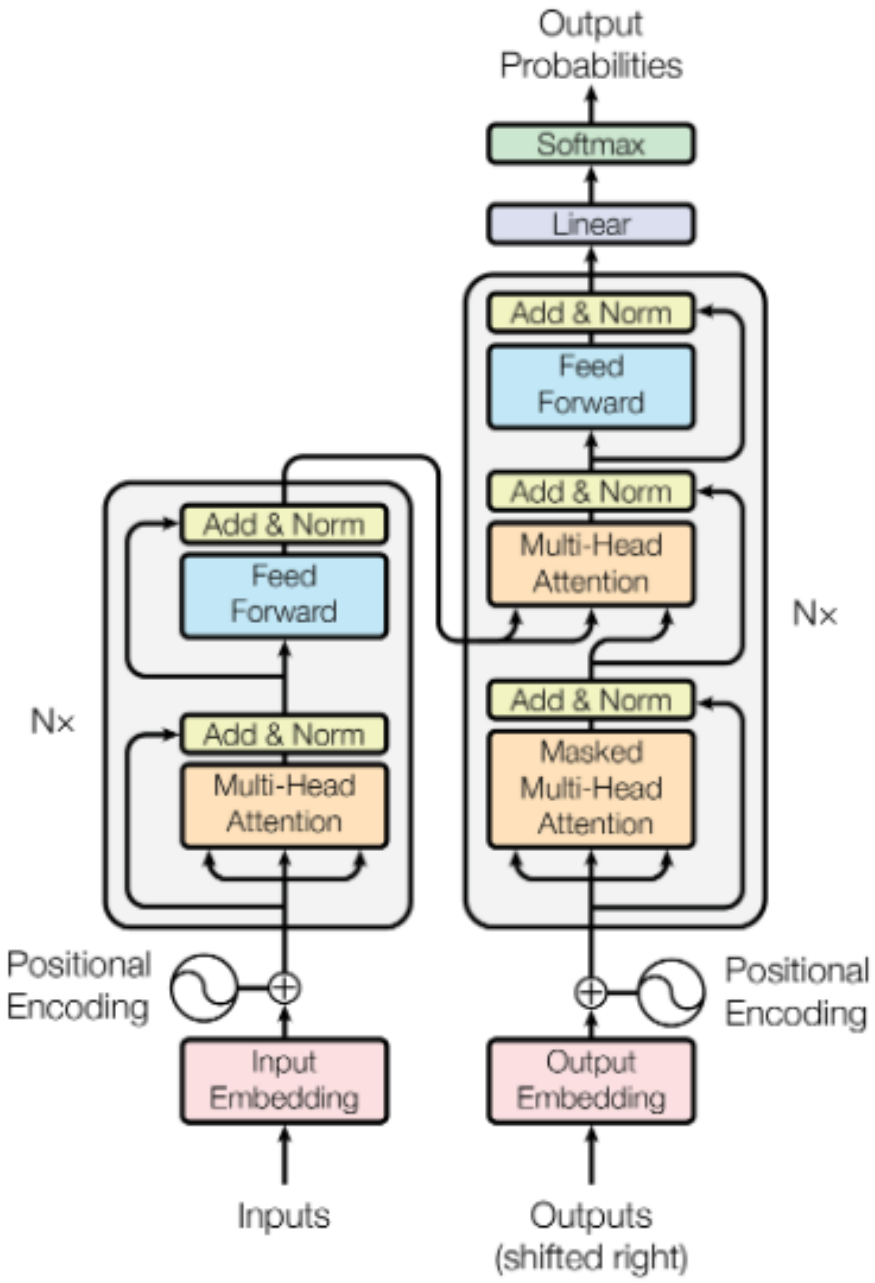

Figure 4.1: The Transformer model architecture (figure from [13]). Both inputs and outputs are embedded and concatinated ($\oplus$) with their positional encodings - those are then fed into encoder and decoder respectivaly. The left part is one encoder layer. The whole encoder is made up of N of these layers stacked (output of one is input of the next). The output of the last encoder layer is used in all N decoder layers (on the right), which are likewise stacked. The output of the decoder (stack of decoder layers) is used for prediction.

4. **Decoder** (Figure 4.1, on the right):

- Also consists of $N = 6$ layers[3].
- Each layer has three sub-layers:
  - **Masked Multi-Head Attention**: As in Encoder, but masked, i.e. attention results for future tokens are disarded - .
  - **Multi-Head Attention**: Applies the multi-head attention mechanism to encoder output.
  - **Feed Forward**: Applies a fully connected feed-forward neural network.
- **Add & Norm**: Residual connections followed by layer normalization after each sub-layer.

5. **Output Embedding**: Converts decoder output tokens into semantic vector space.

6. **Linear & Softmax Layer**: Maps the decoder's output to the probability distribution over the target vocabulary. The most probable token is the output. [4]

4.4. LLaMA model

4.4.1. Introduction

LLaMA or *Large Language Model Meta AI* [14] is a collection of large language models developed by Meta AI.

4.4.2. Features

- **Model Variants:** LLaMA is available in various sizes, offering flexibility for deployment in different environments. These variants range from models of 7 billion parameters in size up to models of 65 billion parameters in size. [5]

- **Training Data:** The model has been trained on 1.4 trillion tokens of data from several sources, including CommonCrawl and Github, Wikipedia (Table 1. in [14]). It therefore has an enormous and domain diverse range of input data.

- **Accessibility:** The code that can be used to run the model has been publicly released under the open-source GPLv3 license [15].

4.4.3. LLaMA-2

LLaMA-2 [16] is an improved version of LLaMA, with similar model sizes. It has the same architecture as LLaMA-1, but was trained on a much larger set of data (2 trillion tokens). It also has doubled context length of 4096 tokens. For our experiments, we were using LLaMA-2.

[3] [13], section 3.1

[4] This may seem contrary to popular experience using e.g. ChatGPT, where given the same input, it may not necessarily output the same result. However, such tools may have random seeds for each interaction session and also may take into account the context of the conversation.

[5] LLaMa-3, which came out in April 2024, has size possibilities of 8 billion and 70 billion parameters

4.5. Time-series Embedding

We now present the technique we've used for using a vanilla (not fine-tuned) LLaMA-2 model to predict time series data. The main idea involves using a framework around a frozen LLM (i.e. one that is not changed during the training process) that transforms input time series data into a text representation the LLM can then work on. Its output is then converted into a prediction. The idea is due to an article by Jin et al. (2024) [17].

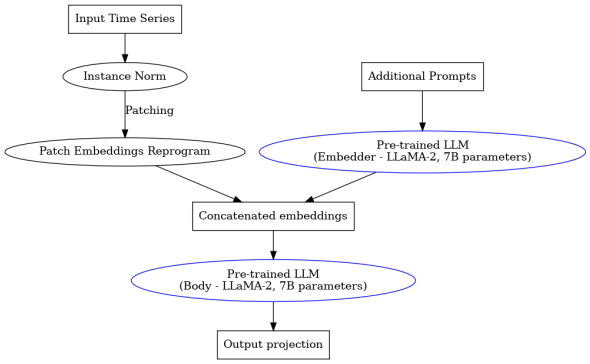

Figure 4.2: Simplified model of the embedding framework. The input time series is normalized and patched (broken down into chunks). These patches are reprogrammed into embeddings, and concatenated with embeddings generated from additional prompts by a pre-trained LLM. The combined embeddings are then processed by the LLM to produce the final forecast output.

The following three paragraphs come from the article.[17]

We consider the following problem: given a sequence of historical observations $X \in \mathbb{R}^{N \times T}$ *consisting of* N *different 1-dimensional variables across* T *time steps, we aim to reprogram a large language model* $f(\cdot)$ *to understand the input time series and accurately forecast the readings at* H *future time steps, denoted by* $\hat{Y} \in \mathbb{R}^{N \times H}$ *, with the overall objective to minimize the mean square errors between the expected outputs* Y *and predictions, i.e.,* $\frac{1}{H} \sum_{h=1}^{H} \|\hat{Y}_h - Y_h\|_F^2$.

The method encompasses three main components: (1) input transformation, (2) a pre-trained and frozen LLM, and (3) output projection. Initially, a multifeature time series is partitioned into N *unifeature time series, which are subsequently processed independently (Nie et al., 2023) [18]. The* i*-th series is denoted as* $X(i) \in \mathbb{R}^{1 \times T}$ *, which undergoes normalization, patching, and embedding prior to being reprogrammed with learned text prototypes to align the source and target modalities. Then, we augment the LLM's time series reasoning ability by prompting it together with the transformed series to generate output representations, which are projected to the final forecasts* $\hat{Y}^{(i)} \in \mathbb{R}^{1 \times H}$ *.*

We note that only the parameters of the lightweight input transformation and output projection are updated, while the backbone language model is frozen. In contrast to vision-language and other multimodal language models, which usually fine-tune with paired cross-modality data, this use of model is directly optimized and becomes readily available with only a small set of time series and a few training epochs, maintaining high efficiency and imposing fewer resource constraints compared to building large domain-specific models from scratch or fine-tuning them.

4.6. Our methodology

In the current and the next few sections we will share the details of our implementation. The goal is to further elaborate on the idea presented in the previous section and describe the specifics and challenges of our approach.

Our model has the same architecture as the one presented in the one described in the previous section. We will expand on each of the main components of the model and explain how it is trained.

4.6.1. Input Transformation

Data Preprocessing

As it was previously mentioned, the input to the model is a vector of floating-point numbers representing prices over time. Consequently, we use only use the target feature from the time series data. This implies that, from the datasets described earlier, all columns except the target column are discarded. Following this, the data is normalized.

Each entry in the updated dataset comprises two features. The first feature is a vector of length **seq_len** consisting of consecutive prices sampled at intervals of **seq_step** values from the original dataset. The second feature is the target feature, which represents the next **pred_len** prices, also sampled using the same step size.

Embedding

The embedding in LLaMa2 involves transforming human language into a format that the model can understand and process. The process converts text into a sequence of tokens and later vectors, which are essentially numerical representations of words. First, the text is tokenized. Each token is then mapped to a corresponding vector of floating-point numbers, creating the embedding. The embedding is supposed to capture semantic information about the tokens, allowing the model to understand the meaning of a word in a specific context.

We use predefined methods to perform the embedding in order to provide the model with instructions and additional information about the nature of the dataset and the upcoming input. The details about the prompt will be described in a separate section. The embedded prompt is later used as a part of the input to the model.

Patching

Patches are small, contiguous segments of the original data sequence, representing local windows of information. By patching we mean a preprocessing of the input feature that is meant to prepare it for the model. It is composed of 2 main steps, which are Patch Embedding and Reprogramming. Considering that the weights of the LLM are frozen, patching plays a crucial role in training the model.

Patch Embedding The method is used to transform raw input data into a format suitable for sequence modeling. It begins by padding the input through repetition to ensure that patches can be extracted without losing boundary information. Afterwards, we divide the input into overlapping patches, each capturing a segment of the sequence. The patches are then projected into a lower-dimensional space using a 1D convolutional layer, which captures local patterns within a single patch. This provides additional information about the data.

Additionally, dropout is applied to the embedded patches in order to prevent overfitting. The resulting sequence, now embedded and regularized, is used as the input for subsequent layers in the model.

Reprogramming The purpose of this step is to enchance the LLM's ability to understand the input data. At the moment the input may be seen as random to the model, so in this step we combine the input vector and the instructions embeddings mentioned in the previous section. A linear transformation with unforzen weights is applied to both. Then, multi-head attention is applied to the input using the instructions embeddings.

Finally, the reprogrammed embeddings are reshaped and passed through the output projection layer, which transforms them back into the appropriate dimensionality for the model's subsequent layers. This allows the LLM to understand the connection between the instructions and the input vector.

4.6.2. Body and Output Projection

Two sequences of vectors received from the patching and embedding layers are concatenated and fed into the LLaMa as a single sequence. The output is then projected to the forecasted values using a linear layer.

4.6.3. Training Process

The model is trained to minimize the MSE between the predicted and actual values. It is trained and evaluated in epochs. After each epoch if the model's performance has improved compared to all previous epochs, the weights are saved. The training process is stopped once the model has not improved over a given number of epochs.

4.7. Model Parameters

In this study, we employed a set of distinctive parameters for our model training:

- **seq_len** – This parameter defines the number of records in one prompt to the model.
- **pred_len** – This parameter specifies the number of records to predict.
- **seq_step** – This parameter determines the interval at which records are selected during iteration. For example, if *seq_step* is set to 4, the records would be indexed as 0, 4, 8, and so on. This means that instead of processing every single record sequentially, you only process every fourth record, like 0, 4, 8, etc., and then 1, 5, 9, etc. This method helps in reducing noise caused by hourly fluctuations, significantly improving the accuracy of the results.
- **learning_rate** – This parameter controls the speed of learning. Lower values mean slower, more stable learning, while higher values mean faster, but potentially unstable, learning.
- **lradj** - This parameter specifies the strategy for adjusting the learning rate during training. Each strategy modifying the learning rate according to a predefined pattern or schedule.

- **patience** This parameter is used to determine the number of epochs with no improvement after which training will be stopped.
- **n_heads** - number of heads in the multi-head attention mechanism.
- **d_ff** - number of neurons in the Reprogramming layer.

4.7.1. Types of lradj

- **type1**: The learning rate is halved after each epoch.
- **type2**: The learning rate changes at specific epochs to predefined values:
  - Epoch 2: 5×10^{-5}
  - Epoch 4: 1×10^{-5}
  - Epoch 6: 5×10^{-6}
  - Epoch 8: 1×10^{-6}
  - Epoch 10: 5×10^{-7}
  - Epoch 15: 1×10^{-7}
  - Epoch 20: 5×10^{-8}
- **type3**: The learning rate remains the same for the first epoch and then decreases by 30% after each subsequent epoch.
- **PEMS**: The learning rate decreases by 10% after each epoch.
- **TST**: The learning rate is adjusted according to the scheduler's last learning rate.
- **constant**: The learning rate remains constant throughout the training process.

4.7.2. Impact on Results

The effectiveness of these parameters is highly contingent upon the specific dataset in use. For datasets characterized by high volatility and a high frequency of records, a smaller *seq_step* is preferable. Conversely, for data that remains relatively stable over time, a larger *seq_step* is necessary to prevent the model from merely replicating the last observed value.

Our experimentation with various proportions between *seq_len* and *pred_len* revealed that optimal results were achieved when *pred_len* was approximately one-fourth of *seq_len*. This finding is intuitive, as it ensures the indicators retain their significance. A more detailed discussion on this can be found in the Prompt Engineering section.

Due to constraints in time and resources, we were unable to identify a universally optimal ratio for all datasets. Nevertheless, we believe that such a golden ratio exists and can be discovered with further research. More detailed information on this can be found in the Results and Conclusion section.

4.7.3. Overfitting Concerns

To avoid reducing the number of records available for training, we leveraged the *seq_step* parameter in our data loaders. Rather than using every *seq_step* value, we trained the model with sequences such as 0, 4, 8, 12, etc., followed by 1, 5, 9, 13, and so on (if *seq_step* was set to 4).

This approach, however, led to overfitting, particularly with a high *seq_ len* of 200, a *seq_ step* of 12, and a *pred_ len* of 40. For illustration, consider feeding the model data from the past 20 weeks and predicting the next 4 weeks (one month). Our currency datasets contain 24 records per day over 5 days a week.

In the initial iterations, our model achieved an accuracy of 89% on the training data in the first epoch. This high accuracy was due to the similarity of the training data sequences. Essentially, we presented price data for the last 5 months divided into 200 records and then predicted the price for the next month. We then slightly shifted the data forward by 1 hour, resulting in nearly identical sequences being fed to the model. Consequently, the model could easily predict the next month +1 hour, given its similarity to the previous predictions.

However, this approach failed during validation, as the model's accuracy drastically dropped when applied to completely unseen test data. This highlights the importance of diversifying training sequences to avoid overfitting and improve the model's generalization capabilities. Further exploration and refinement of these parameters are necessary to develop a robust predictive model.

4.8. Prompt Engineering

The foundational concept behind leveraging a Large Language Model (LLM) lies in its extensive knowledge about the world. Our objective was to determine optimal strategies to harness this knowledge, thereby enhancing our model's forecasting accuracy. Essentially, we aimed to bypass fine-tuning and instead focus on crafting the most effective task descriptions to elicit accurate predictions from the outset.

Initially, we experimented with simply providing sequences of numbers, similar to our approach in other models. However, it became apparent that the model was not trained to understand numerical series in this way. Just as a random person would not understand a sequence of numbers without context, the model's responses often included random numbers, and for longer sequences, it tended to frequently repeat the last number.

Recognizing the need for a more sophisticated approach, we turned to autocorrelation analysis to identify recurring patterns within the data. Here's a detailed explanation of how it works:

1. **Fourier Transformation**: We apply the Fourier transform to the input data x_{enc} to convert it from the time domain to the frequency domain. This helps in identifying patterns by analyzing the frequency components of the data.

Let $X(f)$ be the Fourier transform of x_{enc}. The transformation is defined as:

$$X(f) = \sum_{t=0}^{N-1} x_{\text{enc}}(t) e^{-i2\pi ft/N}$$

In our case, the input data is permuted and transformed:

$$q_{\text{fft}} = \text{FFT}(x_{\text{enc}}) \quad \text{and} \quad k_{\text{fft}} = \text{FFT}(x_{\text{enc}})$$

2. **Cross-Spectral Density**: We compute the cross-spectral density by multiplying the Fourier-transformed data with the complex conjugate of itself. This step helps in identifying the strength and phase relationship between different frequencies.

$$R(f) = q_{\text{fft}}(f) \cdot k^*_{\text{fft}}(f)$$

where $k^*_{\text{fft}}(f)$ denotes the complex conjugate of $k_{\text{fft}}(f)$.

3. **Inverse Fourier Transformation**: We then apply the inverse Fourier transform to convert the data back to the time domain. This results in the autocorrelation function, which shows how the data correlates with itself at different lags (time shifts).

$$r(t) = \text{IFFT}(R(f))$$

4. **Mean Autocorrelation**: We calculate the mean of the autocorrelation across all data points to get a single autocorrelation function that represents the overall pattern in the data.

$$\bar{r}(t) = \frac{1}{M} \sum_{m=1}^{M} r_m(t)$$

where M is the number of data points and $r_m(t)$ represents the autocorrelation for each data point m.

5. **Top-k Features**: Finally, we select the top-k lags with the highest autocorrelation values. These lags represent the steps at which the data shows the most significant recurring patterns.

We identify the top-k lags by finding the indices of the highest values in the mean autocorrelation function:

$$\text{lags} = \text{argsort}(\bar{r}(t))[:k]$$

By using this method, we were able to identify and select the steps with the highest autocorrelation, effectively capturing the most significant patterns in the data.

This method significantly improved our accuracy, particularly for datasets with clear seasonal patterns. For example, in many countries, weather is cyclical (seasons), and depending on the season and temperature, electricity consumption shows correlations. Over the course of a year, electricity usage typically increases during the winter months and decreases during the summer. Similarly, daily patterns show higher usage during daylight hours and lower usage at night.

However, our primary goal was to predict market movements, especially in the forex market, which lacks such strong and clear seasonal patterns in shorter time frames. While financial markets do exhibit some seasonal trends, these are not as pronounced or consistent. Forex market movements and stock prices, for instance, tend to show more long-term growth trends and variability, rather than oscillating within a fixed range or displaying consistent short-term patterns.

To address this challenge, we explored various analytical tools commonly used in trading. Numerous indicators are designed to forecast future prices, based on factors such as moving averages, trading volume, and price trends. We decided to incorporate three widely-used indicators: Relative Strength Index (RSI) [1], Moving Average Convergence Divergence (MACD) [2], and Bollinger Bands (BBANDS) [3]. These indicators, already familiar to the model due to its pre-existing knowledge, significantly enhanced its predictive capabilities.

Bibliography

[1] J. Welles Wilder Jr. *New Concepts in Technical Trading Systems*. Trend Research, 1978.

[2] Gerald Appel. *The Moving Average Convergence Divergence Trading Method*. 1979.

[3] John Bollinger. *Bollinger on Bollinger Bands*. McGraw-Hill, 2002.

4.8.1. Indicators Used

- **Relative Strength Index (RSI)**: This momentum oscillator measures the speed and change of price movements. It oscillates between 0 and 100 and is typically used to identify overbought or oversold conditions in a market. The RSI is calculated using the formula:

$$RSI = 100 - \frac{100}{1 + RS}$$

where RS (Relative Strength) is the average gain of up periods during the specified time frame divided by the average loss of down periods during the specified time frame:

$$RS = \frac{\text{Average Gain}}{\text{Average Loss}}$$

Here, the **average gain** refers to the mean of all positive price changes over the specified period, while the **average loss** refers to the mean of all negative price changes over the same period. These are calculated as follows:

  - **Up Periods**: Time intervals during which the price increases from the previous closing price.
  - **Down Periods**: Time intervals during which the price decreases from the previous closing price.

The calculations are as follows:

  - **Average Gain**: The average of all gains (positive changes in price) during the up periods within the specified time frame.
  - **Average Loss**: The average of all losses (negative changes in price) during the down periods within the specified time frame.

For example, with a time frame of 14 days, the average gain is calculated by summing all the gains over the 14 days and dividing by 14. Similarly, the average loss is calculated by summing all the losses over the 14 days and dividing by 14.

These values are then used to calculate the RS, and subsequently the RSI, to determine whether the market is potentially overbought (typically RSI > 70) or oversold (typically RSI < 30).

- **Moving Average Convergence Divergence (MACD)**: This trend-following indicator shows the relationship between two moving averages of a security's price. The MACD is calculated by subtracting the 26-period Exponential Moving Average (EMA) from the 12-period EMA:

$$\text{MACD} = \text{EMA}_{12} - \text{EMA}_{26}$$

Additionally, a 9-period EMA of the MACD, called the "signal line," is plotted on top of the MACD to function as a trigger for buy and sell signals:

$$\text{Signal Line} = \text{EMA}_9(\text{MACD})$$

The MACD histogram, which represents the difference between the MACD and the signal line, is often used to identify potential buy and sell points:

$$\text{MACD Histogram} = \text{MACD} - \text{Signal Line}$$

- **Bollinger Bands (BBANDS)**: These volatility bands are placed above and below a moving average. Volatility is based on the standard deviation, which changes as volatility increases and decreases. Bollinger Bands consist of three lines:
  - Middle Band: a simple moving average (SMA) of the security's price over N periods:

$$\text{Middle Band} = \text{SMA}_N$$

  - Upper Band: the middle band plus k times the standard deviation (σ) over the same period:

$$\text{Upper Band} = \text{SMA}_N + k\sigma$$

  - Lower Band: the middle band minus k times the standard deviation (σ) over the same period:

$$\text{Lower Band} = \text{SMA}_N - k\sigma$$

    where N is the number of periods for the moving average and k is a multiplier, typically set to 2.

By incorporating these indicators, we enhanced the model's ability to predict market movements. The RSI helps identify overbought and oversold conditions, the MACD provides insights into price trends and momentum, and Bollinger Bands measure market volatility. This comprehensive approach allowed us to capture various aspects of market behavior, leading to improved predictive performance.

Incorporating these indicators yielded a substantial improvement in our model's performance. The accuracy of our predictions increased by over 2% on certain datasets, highlighting the value of integrating domain-specific indicators and leveraging the LLM's pre-existing knowledge. Detailed results will be discussed in the subsequent sections.

4.9. Possible Improvements

4.9.1. Underlying LLM

There are a few different options when it comes to open source LLMs. We have used a 7 billion parameters LLaMa 2 model, however there are more powerful models available. For example, the 70b LLama 2 or the 70b LLaMa 3 score 40-60% better on nearly all benchmarks, which could mean an inprovement in our model's performance with such models used as the underlying LLM, however it would require significantly more computational resources as well as additional code that would allow us to run the program on multiple GPUs. The problem is that a single graphic card simply wouldn't have enough memory to train the model.

4.9.2. Larger Training Set

The model was trained on datasets which contained less than 40000 records. It takes us approximately 12 hours to train the model on a dataset of this size. Training the model on a larger dataset would allow to fit more complex patterns in the data which could potentially be developing over a span of multiple years.

4.10. Results

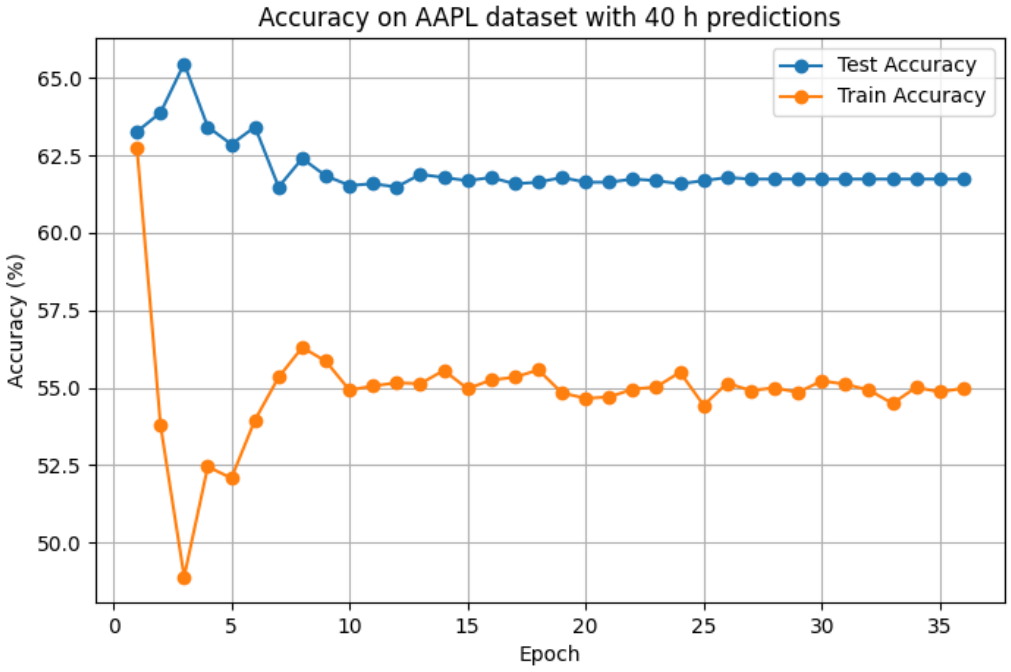

Figure 4.3

The predictability of datasets varied significantly. Moreover, splitting the datasets into different periods for training, validation, and testing purposes also influenced the performance outcomes. Models trained on smaller datasets often yielded the most interesting results, as the data to be predicted was temporally closer to the oldest training data.

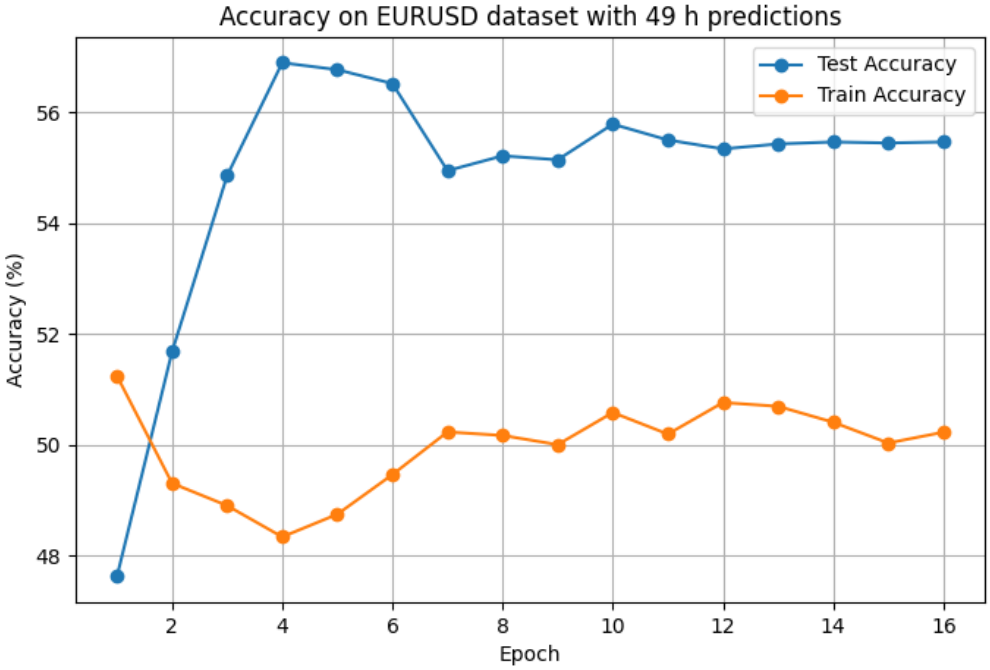

Figure 4.4

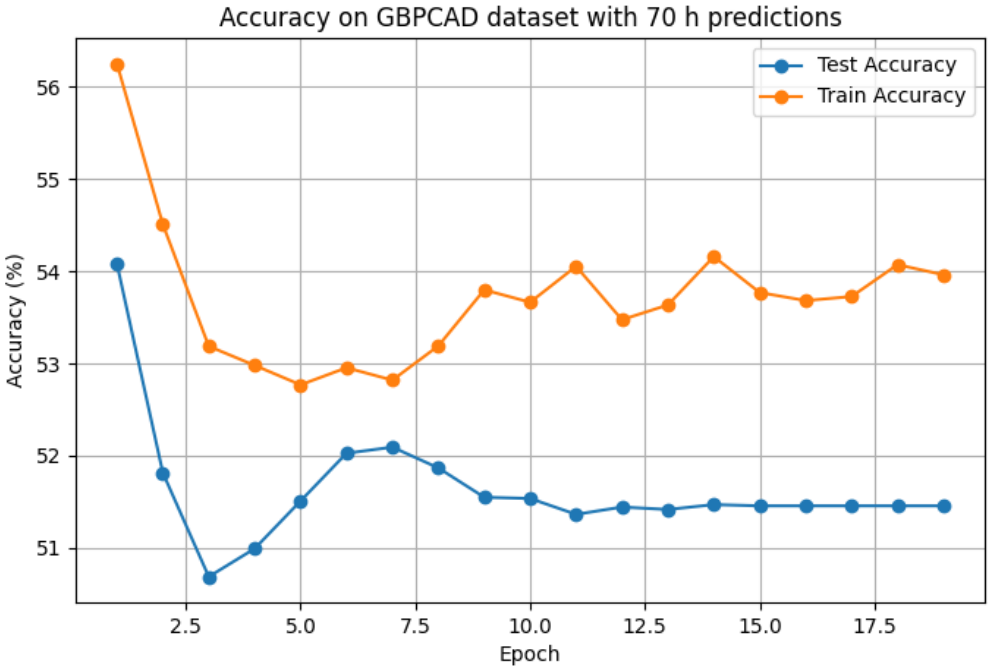

Figure 4.5

Due to limited resources, specifically two computers with high-performance graphics cards, training a dataset of 100,000 records took approximately 2-3 days. Given the substantial

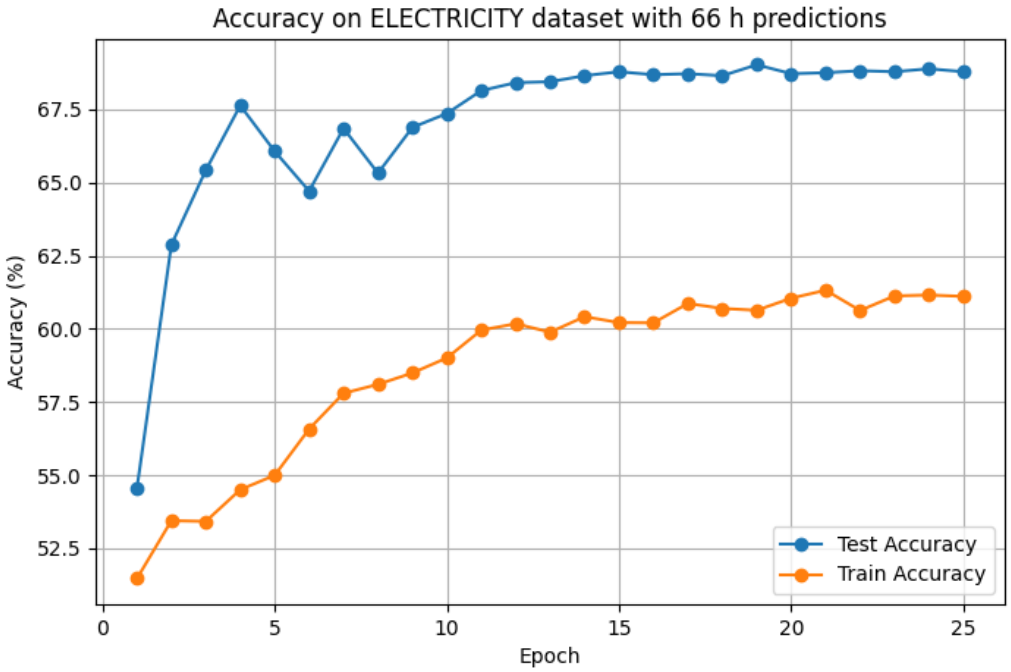

Figure 4.6

size and resource demands of large language models, we were unable to test all interesting combinations, leaving room for further exploration.

One key observation was that using a short prediction length (pred_len) of 4 or fewer, despite the goal of predicting price increases or decreases, proved suboptimal. The mean squared error (MSE)-based loss function did not optimize well compared to longer sequences. Conversely, with very long prediction lengths (over 60), the model struggled due to the complexity of the task. The best results were obtained with a pred_len around 10.

Another noteworthy point was the use of hourly datasets. Predicting values for the upcoming hours was nearly impossible due to a high degree of randomness, likely influenced by speculative decisions and the actions of individuals who may not fully understand market dynamics. However, predictions over longer periods showed more promise, leading to the introduction of the seq_step parameter. By skipping several records, we achieved more predictable sequences.

The learning rate (*lr*) between 0.01 and 0.0001, with the *lradj* parameter set to type1 or type3, yielded the best results. Starting with values too low prevented the model from achieving good results, while too high a learning rate damaged the pretrained LLM's performance from the outset. Additionally, not decreasing the learning rate over epochs resulted in suboptimal training outcomes.

In summary, several key insights emerged from our experiments:

- **Dataset Size and Temporal Proximity**: Models trained on smaller datasets, where prediction data was temporally closer to the training data, performed better.
- **Prediction Length (pred_len)**: A pred_len around 10 was optimal. Shorter lengths did not allow the MSE-based loss function to optimize well, while longer lengths were too complex for the model.

- **Hourly Data Prediction**: Predicting hourly values was challenging due to randomness, but longer periods with seq_step parameter adjustments improved predictability.
- **Learning Rate and Adjustment (lradj)**: Learning rates between 0.01 and 0.0001 with type1 or type3 *lradj* settings were most effective. Lower starting rates hindered performance, and higher rates damaged pretrained capabilities without proper reduction over epochs.

These findings highlight the importance of tuning prediction length, learning rates, and the use of appropriate parameters for different datasets to achieve optimal forecasting performance. Further exploration of combinations and settings is necessary to fully understand and leverage the capabilities of large language models in predictive tasks.

Chapter 5

Main results

In this chapter we present the results in the following way: for each dataset described in chapter 3, we present one table. The table presents accuracies each model achieves at predicting the price `Prediction Timestep` time into the future. Next to the name of each dataset is a tuple of the form (`pred_len`, `seq_step`) with which parameters the models were trained.

5.1. Method

For each dataset, we have experimented with a few different pairs (`pred_len`, `seq_step`), and chose the one, where the LLM performed best on the final epoch, i.e. had highest accuracy. Then, we've trained the other 4 smaller models with these parameters, to see how they compare.

5.2. Results

Dataset model results						
Dataset name	Prediction timestep	Linear regression	MLP	CNN	ResNet	LLM
AAPL (40, 1)	40 days	63.55%	65.42%	34.92%	34.92%	54.98%
BTCUSD (5, 2)	10 hours	48.54%	46.66%	45.31%	45.56%	50.54%
EURUSD (7,7)	49 hours	49.27%	47.47%	53.48%	49.87%	55.46%
GBPCAD (10, 7)	70 hours	53.03%	49.08%	53.67%	43.93%	51.45%
GBPTRY (5, 2)	10 hours	54.79%	58.64%	41.97%	40.93%	54.65%
Electricity (6, 11)	66 hours	57.81%	66.87%	65.31%	50.62%	68.78%
US500 (10, 1)	10 hours	42.54%	44.00%	43.49%	41.12%	53.37%

5.3. Failed attempts

Chapter 6

Conclusion

Bibliography

[1] https://finance.yahoo.com/quote/AAPL/history/

[2] https://archive.ics.uci.edu/dataset/321/electricityloaddiagrams20112014

[3] https://en.wikipedia.org/wiki/Overfitting

[4] Kuchibhotla, A. K., Brown, L. D., Buja, A. & Cai, J. (2019). All of linear regression. arXiv preprint arXiv:1910.06386.

[5] Steinwart, I. & Christmann, A. (2006). Estimating conditional quantiles with the help of the pinball loss. arXiv preprint arXiv:math/0612817.

[6] (Add complete citation when available).

[7] Simonyan, K. & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv preprint arXiv:1511.08458.

[8] Amidi, A. & Amidi, S. (n.d.). Convolutional Neural Networks cheatsheet. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

[9] He, K., Zhang, X., Ren, S. & Sun, J. (2015). Deep Residual Learning for Image Recognition. arXiv preprint arXiv:1512.03385.

[10] Jurafsky, D. & Martin, J. H. (n.d.). Token Embeddings. Retrieved from https://web.stanford.edu/~jurafsky/slp3/6.pdf.

[11] Li, Z., Li, J. & Liu, X. (2023). Efficient Language Models with Dynamic Token Dropping. arXiv preprint arXiv:2303.18223.

[12] Zhang, Z., Li, X. & Yang, W. (2023). An Introduction to Large Language Models. arXiv preprint arXiv:2304.00612.

[13] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. & Polosukhin, I. (2017). Attention is All You Need. arXiv preprint arXiv:1706.03762.

[14] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M. A., Lacroix, T., ... & Jegou, H. (2023). LLaMA: Open and Efficient Foundation Language Models. arXiv preprint arXiv:2302.13971.

[15] Meta AI. (2023). LLaMA GitHub Repository. Retrieved from https://github.com/meta-llama/llama.

[16] Touvron, H., Martin, X., Stone, A., Albert, P., Almahairi, A., Babaei, Y., ... & Jegou, H. (2023). LLaMA 2: Open Foundation and Fine-Tuned Chat Models. arXiv preprint arXiv:2307.09288.

[17] Wang, Y., Xu, J. & Lin, J. (2023). Reprogramming Large Language Models with Synthetic Data. arXiv preprint arXiv:2310.01728.

[18] Nie, Y., Nguyen, N. H., Sinthong, P. & Kalagnanam, J. (2023). A time series is worth 64 words: Long-term forecasting with transformers. In International Conference on Learning Representations.