EXPLORATION OF DEEP LEARNING ARCHITECTURES FOR TIME SERIES FORECASTING IN RETAIL SALES

Elliot H. Ha
Pratt School of Engineering
Duke University
Durham, NC 27708, USA
elliot.ha@duke.edu

ABSTRACT

This study evaluates Long Short-Term Memory (LSTM) and Transformer deep learning models for forecasting retail sales time series at Corporación Favorita, an Ecuadorian retailer. After an exploratory data analysis and preprocessing, both models are developed and compared. The results reveal that the LSTM model surpasses the Transformer in loss performance, demonstrating its efficacy in sales forecasting. This research provides insights into effective deep learning approaches for retail time series forecasting, aiding in better inventory management and customer satisfaction.

1 INTRODUCTION

Time series forecasting lies at the heart of many modern problems today. It is thus no surprise that time series forecasts are both relevant and critically important to the everyday grocery store; with a behind-the-scenes war being waged every day to tip the delicate balance of overstocking versus understocking products and goods in the local retailer's favor. Overstocking products, especially perishable ones, leads to an unsold surplus for the store and potential food waste, which negatively impacts profits. On the other hand, understocking products similarly leads to lost revenue, and perhaps more importantly, unsatisfied customers. For Corporación Favorita, an Ecuadorian-based retailer, being able to find this perfect balance between the two has prompted the question of whether or not deep learning architectures may be applied to retail data over a time series.

This paper aims to first provide two implementations of deep learning architectures for the specific problem of predicting unit sales for the many items sold at different Favorita retail stores, based off of historical data and trends. The first of which will be explored will be a standard Long Short-Term Memory (LSTM) model architecture, and the second of which will be a Transformer model architecture adapted from the ”Attention is All You Need” by Vaswani et al. Vaswani et al. (2017) paper for time series forecasting purposes. Finally, this paper will aim to provide a comparison between the two architectures, noting their respective strengths, weaknesses, and overall efficacy in evaluating time series forecasts.

This paper will investigate the effectiveness of Long Short-Term Memory (LSTM) and Transformer models in forecasting retail sales time series. It aims to highlight the potential of the LSTM model in outperforming the Transformer in terms of loss performance, underscoring its utility for accurate sales prediction in retail environments like Corporación Favorita. The study will demonstrate how these models can enable retailers to forecast product demand more precisely, leading to optimized inventory management, enhanced customer satisfaction, and reduced surplus, ultimately contributing to increased profitability.

2 BACKGROUND

The Transformer model, introduced in ”Attention is All You Need” by Vaswani et al. Vaswani et al. (2017), represents a significant shift in sequence learning, particularly for natural language processing. Its core is the self-attention mechanism, enabling the model to process input sequences

Table 1: Train data (3,000,888 entries)

FEATURE	DTYPE	MIN	MAX	MEAN	STD DEV	MISSING
id	*int*	0	$3,000,887$	$1,500,438$	$866,000$	0
date	*str*	2012-12-31	2017-08-14	2015-04-24	NaN	0
store_nbr	*int*	0	54	27.5	15.6	0
family	*str*	NaN	NaN	NaN	NaN	0
sales	*float*	0	$124,717$	357.78	$1,102$	0
onpromotion	*int*	0	741	2.6	12.6	0

Table 2: Test data (28,512 entries)

FEATURE	DTYPE	MIN	MAX	MEAN	STD DEV	MISSING
id	*int*	$3,000,888$	$3,029,399$	$3,015,143$	$8,230$	0
date	*str*	2017-08-15	2017-08-30	2017-08-23	NaN	0
store_nbr	*int*	0	54	27.5	15.6	0
family	*str*	NaN	NaN	NaN	NaN	0
onpromotion	*int*	0	646	6.97	20.7	0

in parallel and capture long-range dependencies, a departure from the sequential processing of RNNs and LSTMs. The architecture comprises an encoder and decoder, each with multiple layers of multi-head self-attention and feed-forward networks. Positional encodings are employed to retain sequence order information. For time series forecasting, the Transformer's ability to handle long-range dependencies and its efficient parallel processing make it a promising candidate, suggesting potential advancements in forecasting accuracy and efficiency.

3 METHODOLOGY

As an overview of the methodology that we will use in approaching this time series forecasting problem, our solution will be broken up into four main steps.

First, we will perform exploratory data analysis on our data, specifically looking out for the structure of the data while considering how we can adapt it towards being suitable for time series prediction models such as our LSTM and Transformer. We will also analyze the overall quality and characteristics of each dataset that we choose to use, in order to conclude by making an informed decision as to which direction it is best to go to preprocess the data for our time series models.

Second, we will perform the appropriate preprocessing steps that we have decided best suit our task for time series forecasting. This will necessarily include transforming the data somehow into sequence data, as is typical in time series models, as well as additional preprocessing steps that are typical for deep learning tasks such as normalization.

Next, we will start with building the Long Short-Term Memory model for our preprocessed data. We will train the model on our processed training data and evaluate it across all entries in the testing data, as we are missing sales data in the testing data set.

Finally, we will repeat the process from building our LSTM for our Transformer model adapted to time series forecasting. After training, we will use the model to predict the testing data sales values. This will allow us to compare the efficacy of both models on the same task.

3.1 EXPLORATORY DATA ANALYSIS

To begin our exploratory data analysis, we will focus on three datasets that we have been given. These will be train.csv, test.csv, and oil.csv. This section aims to explore the structure of the data and provide some insights via typical exploratory data analysis techniques.

Table 3: Oil data (1,218 entries)

FEATURE	**DTYPE**	**MIN**	**MAX**	**MEAN**	**STD DEV**	**MISSING**
date	*str*	2012-12-31	2017-08-30	2015-05-02	NaN	0
dcoilwtico	*float*	26.2	111	67.7	25.6	43

Starting with the training data, whose general characteristics are displayed in Table 1, we first note that there are no missing values across any feature, but most notably the date, which tells us that we will be able to make consistent sequences in time without having to preprocess further. We also note that the wide range and high standard deviation indicate significant variability in sales, possibly due to differences in store locations, product families, and other factors. As well as this, the presence of sales as low as 0 might indicate no sales for some items on certain days or data recording practices that include days/items with no sales. Finally, with the maximum value of the onpromotion feature of 741, as well as the mean of 2.6 and standard deviation of 12.6, this suggests that promotions are not equally distributed across products or stores, with some items experiencing heavy promotion while others have none.

Moving onto the testing data, displayed in Table 2, it is shown again that there are no missing values across any feature, with the standout again being the date, as the minimum value of the testing date is the date directly after the training data ends. This suggests that we will be able to iteratively predict sales values for our testing dates in an autoregressive manner, using the output prediction at one time step to build our input sequence for the next time step.

Finally, the oil data displayed in Table 3 shows missing values for the dcoilwtico feature, which represents the closing price of oil, a notable variable due to the importance of the resource to Ecuador's economy, and thus, tangentially, the average spending power of an Ecuadorian. By looking into the data more, it is noted that these missing dates all occur on weekend dates, suggesting the data source is likely a market that operates on weekdays only. This needs to be considered in any time series analysis to avoid skewed interpretations.

To summarize the overview on the datasets, we have presented three datasets with little to no missing values or outliers. In addition, we identify two important categorical variables across the data in the form of the store_nbr and family, and three continuous features in the sales, onpromotion, and dcoilwtico features. In order to build inputs suitable for time sequences, we will need to consider how to combine the categorical information into sequences with the continuous features.

3.1.1 APPROACH TO SEQUENCE BUILDING

Approaching our training data, it is immediately of note that there are many more entries than there should be dates, if each date was given a distinct entry immediately suitable for time series problems. The training data records sales numbers for dates for any of the 54 distinct store numbers, and if a specific store had even a single sale on that specific date, then there are rows outlining each of the 33 families of products, as well as there sales numbers, which tend to be sparse due to unlikelihood of selling at least one item from every single family on a given date. Due to the variability in our data's structure, i.e., there can be as little as 1 store and as such 33 rows of data corresponding to a date, to as many as 54 stores having sales, with each store getting 33 rows in the dataset outlining the sales numbers of the products that they sold that date, this presents a significant challenge in deciding how best to preprocess this data in order to make it suitable as input to time series models, i.e., make contiguous time step sequences that are consistent in the data that they represent.

There are a couple of different approachs to consider. First, for our time series models, we could build a single model handling all combinations that could potentially learn global patterns applicable to all stores and families. This would be done with an embedding vector for categorical information, with each date necessarily having to encode all 54*33 = 1782 possible store_nbr, family combinations of sales data as an upper bound. However, such a model might struggle with the high variability and sparsity across combinations. It may fail to capture the unique characteristics of each specific store-family pair. Second, we could consider using feature engineering to solve this problem by encoding the categorical features as one-hot vectors and using them as inputs to a different model before our time series model to extract a condensed feature representation. However, this would significantly increase the feature space as we would be adding 87 features to our space, and could lead to a very sparse and high-dimensional problem, making the model prone to overfitting and increasing computational complexity. Finally, we can choose to build a separate model for each store_nbr, family combination. This allows for tailored learning specific to each combination's sales pattern, as well as handling sparsity as a dedicated model might be better suited to learn from the limited data available. This would solve our data structure approach as we would be able to filter the dataset for specifically the categorical combination at hand, which will naturally lend itself towards singular time step increments, as it will be impossible for a given store_nbr, family combination to occur twice in one day. Although the computational complexity of this approach is excessive with building a separate model for each combination, it equals the computation needed to compute a similarly sized embedding for each date in order to get our time steps in a sequential order, with this approach's advantages stemming from how it intrinsicly deals with sparsity in data and its granularity in learning each combination's patterns. Finally, by building a model for each categorical combination pairing, this inherently utilizes the categorical information in the model, allowing us to simply use the remaining features, all continuous, as the input features for our input sequences.

To summarize, the decision to build a separate model for each store_nbr, family pairing is driven by the need to address the unique sales patterns and data sparsity issues inherent in each combination. While alternative approaches like a single unified model or embedding layers could provide a more elegant and scalable solution, they come with their own set of challenges, particularly in effectively capturing the variability across different store-family combinations. The chosen approach aims to balance model complexity, data sparsity, and the need for customized predictions for each unique combination.

3.2 Data Preprocessing

To proceed with data preprocessing, it is important to bear in mind the final shape of input that will be passed to the models. As this is a time series problem, we will be using sequences of our data of shape $(sequence\_length, num\_features)$. As these will be batched, our final input will end up being tensors of shape $(batch\_size, sequence\_length, num\_features)$.

3.2.1 Creating Subset Data

First, I handled the oil dataset, merging it with both the training and testing dataset with a left-merge on the date variable. To handle the missing weekend dates, I interpolated the merged datasets with a limit direction in both directions, as this would allow for dates to be filled that do not have immediately following data values, such as dates before the weekend starts. Then, I obtained a list of the unique entries for both the store_nbr and families, of which are shared across the training and testing datasets. With a double-nested loop, it becomes possible to select a subset of the larger training data with values only for this specific store_nbr and family, which results in us having sequential time data with step sizes of 1.

3.2.2 Preprocessing Subset Data

As a reminder, the keys of each subset of data from the merged training dataset were (id, date, store_nbr, family, sales, onpromotion, dcoilwtico). As we are handling the categorical variables store_nbr and family via our approach to data, we drop this, the id, and set the date as the index of the dataframe. We are left with three continuous variables of sales, onpromotion, and dcoilwtico for the training data for this subset for the store_nbr, family combination. As we are not guaranteed to have sales for this categorical combination for each date, we fill in missing dates with zero values. Finally, we apply a Min-Max Normalization scaling to the data.

3.2.3 Creating Sequences

Our preprocessed subset of data now has shape $(num_entries, num_features)$, where each entry is a date, and our features are the normalized continuous features. We want to create sequences where each token is a time step, and each token has the full feature dimensionality. In order to do so, we iterate through the dataset up until the seq_lengthth index from the last point. This allows us to take a slice of the dataframe from our current index to our current index plus the sequence

Table 4: LSTM hyperparameters

HYPERPARAMETER	**VALUE**	**DESCRIPTION**
$hidden\_dim$	128	The hidden dimension of the model
num_layers	3	The number of LSTM layers
$num\_features$	3	The number of continuous features
$output\_dim$	1	The output sales value from the feature dimension
$batch\_size$	32	The batch size for the DataLoader

length, which gives us our sequence of shape $(seq_length, num_features)$. Additionally, we take take the $index + seq_length$ value from the 'sales' column. This will be our target label, or the 'next-date' sales value we want to predict. We iterate through the dataset in this manner, appending each sequence and label to their respective arrays, and finally returning them as NumPy float arrays. From these, we create a TensorDataset, and then DataLoader with a parameter of $shuffle = True$. This DataLoader will give us inputs, $(batch_size, seq_length, num_features)$.

3.3 Long Short-Term Memory (LSTM) Model

We start discussion of the Long Short-Term Memory Model in the context of its usual use-case: natural language processing tasks. Here, the objective is to predict the next word, a categorical feature, by having the model outputting a softmax probability distribution over the vocabular and adjusting it with the cross-entropy loss, which will give us the next word if we take the argmax of the output probability vector. In our specific time series use case, this changes to wanting to predict the next date's sales value, a continuous feature, so we can skip several steps in the usual LSTM's workflow such as the embedding layers and creating a probability distribution over data.

Instead, both our LSTM and Transformer will utilize the Root Mean Squared Log Error, mathematically,

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

Where n is the total number of instances, $\hat{y}_i$ is the predicted value for instance i, y_i is the actual value for instance i, and log represents the natural logarithm. This is a form of loss that incurs a larger penalty for underestimation of results than overestimation. This is important in the store sales prediction case as it is much more profitable and useful for a store to have a surplus of a product than a deficit.

3.3.1 Model Architecture

Before we discuss our LSTM's architecture, we will define the hyperparameters as follows in Table 4:

With the context out of the way, we start with an input from the DataLoader of shape $(batch\_size, seq\_len, num\_features)$. We can input this directly to the LSTM layer with internal weights initialized to zeros with the current batch size. For further LSTM layers, of which we will have 3, the internal weights will be set to the weights from the previous layer. As our LSTM layer outputs an output of shape $(batch\_size, seq\_len, hidden\_dim)$ in addition to the tuple of hidden and cell states, it is from this that we want to obtain our final sales prediction. We can do so by slicing this output along the sequence dimension to get the last time step of shape $(batch\_size, hidden\_dim)$, which represents the LSTM's output representation of the next date's sales values. Finally, we send this through a Fully Connected Layer mapping from $hidden\_dim$ to an $output\_dim = 1$ and return this as our normalized sales prediction along with the last internal weights from the last LSTM layer.

Table 5: Transformer hyperparameters

HYPERPARAMETER	**VALUE**	**DESCRIPTION**
$model\_dim$	512	The hidden dimension of the model. Originally d_k
num_layers	6	The number of encoder and decoder blocks
num_heads	8	The number of heads for multi-head attention
$num\_features$	3	The number of continuous features
$forward\_expansion$	4	The scaling factor in the feed forward network
N_enc	15	The input sequence length for the Encoder
N_dec	1	The input sequence length for the Decoder
$output\_dim$	1	The output sales value from the feature dimension
$batch\_size$	32	The batch size for the DataLoader

3.4 Transformer Model

We will now begin discussion of the Transformer architecture used in this time series forecasting problem. It will be divided into two parts: an overview of the Encoder, and how it ties into the Decoder to produce output. Beforehand, we will define the hyperparameters as follows in Table 5:

3.4.1 Transformer Encoder

The transformer starts with an input sequence from the DataLoader of shape $(batch\_size, N\_enc, num\_features)$. We send this to a Linear Layer in order to project the input into the model space, $(batch\_size, N\_enc, model\_dim)$. We also create a positional embedding of similar size and add to this projected input. This represents the input to the first Encoder block.

Each Encoder block is defined by two parts. First, a Multi-Head Attention block with a subsequent Add & Norm block. Then, a Feed Forward block with a subsequent Add & Norm block. Connected to each of these Add & Norm blocks are skip connections from the previous input.

Each Multi-Head Attention block takes the input of shape $(batch\_size, N\_enc, model\_dim)$. To split into the different number of heads, we send this input through a Linear Layer to get shape $(batch\_size, N\_enc, model\_dim/num\_heads)$. For the Encoder solely, we do this three separate times in order to get our Key, Value, and Query matrices each of these shape. We then reshape each of these into $(batch\_size, num\_heads, N\_enc, model\_dim/num\_heads)$ with the num_heads in the second dimension to better parallelize processing.

From here, we calculate attention via the formula Attention$(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$ Vaswani et al. (2017). Explicitly, QK^T gives us a shape of $(batch\_size, num\_heads, N\_enc, N\_enc)$. Applying a softmax and normalizing across the third dimension by $\sqrt{d_k}$, which, in this case, is $model\_dim$, will keep this shape. Applying the last matrix multiplication with the values matrix of shape $(batch\_size, num\_heads, N\_enc, model\_dim/num\_heads)$ gives us $(batch\_size, num\_heads, N\_enc, model\_dim)$, which we reshape into $(batch\_size, N\_enc, num\_heads \cdot model\_dim/num\_heads) = (batch\_size, N\_enc, model\_dim)$. Finally, we send this to a Linear layer of the same output dimension, and this will be the output from our first Multi-Head Attention block.

From previously, we add our original input from the positional embedding of the same shape, and then apply Layer Normalization to this output. Now, we send this to a Feed Forward network with two Linear layers and a ReLU non-linearity function in between. The first Linear layer projects the input from $(batch\_size, N\_enc, model\_dim)$ to $(batch\_size, N\_enc, forward\_expansion \cdot model\_dim)$, and then ReLU is applied, and then the second Linear layer brings the input back to $model\_dim$. We send this to another Add & Norm block where we add the input to the Feed Forward network to our current input, and then apply Layer Normalization again. This output will be that of our very first Encoder block. We use this as the input to our next Encoder block and we repeat this for $num\_layers = 6$ total Encoder blocks. Thus, our Encoder is finished.

Table 6: LSTM Loss Performance

COUNT	MEAN	STD DEV	MIN	25%	50%	75%	MAX
1782	0.128488	0.092307	0.000000	0.068375	0.103600	0.159650	0.456200

Table 7: Transformer Loss Performance

COUNT	MEAN	STD DEV	MIN	25%	50%	75%	MAX
1782	0.167399	0.103450	0.000000	0.090125	0.148350	0.232575	0.596400

3.4.2 TRANSFORMER DECODER

The architecture of our Transformer's Decoder is largely similar to that of our Encoder, with three important distinctions. The first is that the input to the Decoder is no longer a sequence of shape $(batch\_size, N\_enc, num\_features)$, representing the sequence across all features for $N\_enc = 15$ time steps, but rather it is of shape $(batch\_size, N\_dec, output\_dim)$, where $N\_dec = 1$, as we are trying to predict the single next-date sales output value. Similarly, the final output of our decoder block will be of equal shape to this as we want to use our predicted outputs as part of our next sequence inputs. Second, the first multi-head attention block in the Decoder is distinct from those in the Encoder as it uses a lower triangular mask in order to make future tokens relative to our current token values close to negative infinity, thus forbidding our model from learning from these future values. This is because the actual predicting of values happens in the Decoder, and as such, we cannot learn from these future values. The idea behind the implementation is that if we are trying to mask our attention scores, which for our decoder, will be of shape $(batch\_size, num\_heads, N\_dec, N\_dec)$, then we want to make a square matrix tensor of ones of shape (N_dec, N_dec), take the lower triangle of this, and expand it into the decoder shape in order to apply it as a mask over the attention scores. Finally, the last difference between the Decoder and Encoder is that there is a second block of multi-head attention, but it is distinct in that it uses the output from the last Encoder block as its inputs for the Key and Value matrices. The Query matrix comes from the previous masked multi-head attention block.

Altogether, we start with an input of shape $(batch\_size, N\_dec, output\_dim)$, and similarly to the Encoder input, project it to the model dimension and add a positional encoding. Our input to the first Decoder block is thus $(batch\_size, N\_dec, model\_dim)$. Our output from each Decoder block will be of similar shape, and the output from the very last Decoder block will be sent to a Linear layer in order to project it into our output dimension for our sales prediction.

4 RESULTS

Both models were trained on the same data sets with a batch size of 32, number of epochs of 10, learning rate of 0.001, the 'Adam' optimizer, a 'MultiStepLR' scheduler, and trained on 'CUDA'.

4.1 LONG SHORT-TERM MEMORY MODEL

When submitted to the Kaggle competition, a score of 3.02217 was obtained for the final testing data submission CSV file.

4.2 TRANSFORMER MODEL

When submitted to the Kaggle competition, a score of 3.9873 was obtained for the final testing data submission CSV file.

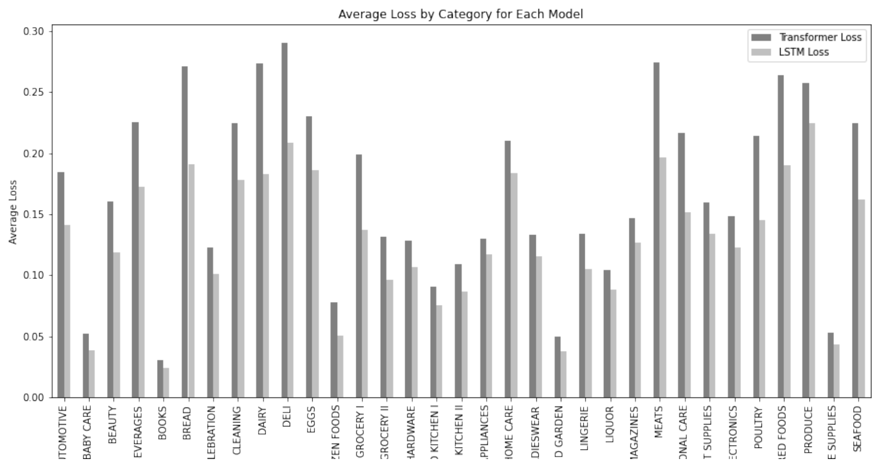

Figure 1: Average loss by category for the transformer vs LSTM model

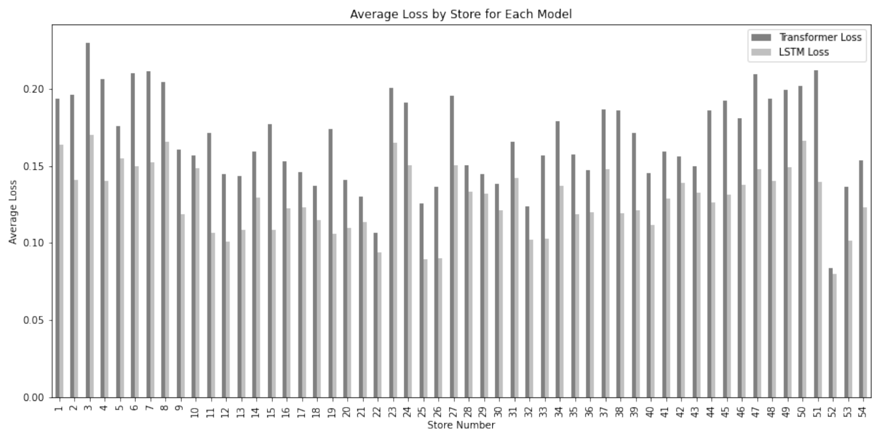

Figure 2: Average loss by store for the transformer vs LSTM model

5 Conclusion

Based on the presented results, it is evident that in the context of the evaluated dataset or task, the LSTM model outperforms the Transformer model in terms of loss performance. The statistical analysis reveals that the LSTM not only maintains a lower average loss but also exhibits a tighter distribution of loss values, as indicated by its lower standard deviation. Additionally, the LSTM shows more favorable results in both the median and the maximum loss recorded, suggesting its consistency in maintaining lower loss values across various segments of the dataset. This is further compounded visually in Figures 1 and 2, with the Transformer having higher values across all stores and categories. While these findings highlight the LSTM's efficiency in this specific scenario, it's crucial to note that this conclusion is based solely on loss metrics. For a thorough assessment, other performance indicators and the model's applicability to the task at hand should also be taken into account. Nonetheless, in the realm of loss minimization, the LSTM model demonstrates a clear advantage over the Transformer model for this particular application.

REFERENCES

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.