Trading at the Close: Deep Learning Models

Hugo Brunlid
KTH Royal Institute of Technology
brunlid@kth.se

Alex Lövgren
KTH Royal Institute of Technology
alexlov@kth.se

Jacob Gustavsson
KTH Royal Institute of Technology
jacgu@kth.se

Abstract

Predicting financial markets is a challenge that millions of people face every day. The Efficient Market Hypothesis (EMH) suggests that asset prices fully reflect all available information, implying that future price movements are largely unaffected by historical data. This project explores financial market prediction using deep learning models, with datasets from the Kaggle competition *Trading at the Close*. Five deep learning models were implemented — Neural Network, Recurrent Neural Network, Convolutional Neural Network, Long Short-Term Memory, and Transformer model. Despite limited predictive performance, results are aligned with top competition submissions, highlighting the difficulty in predicting financial markets and providing support for the EMH.

1 Introduction

Predicting financial markets is a challenging and intriguing task due to their complex and dynamic nature. The Efficient Market Hypothesis (EMH) posits that asset prices fully incorporate all available information, making it impossible to infer patterns from historical price data. Nonetheless, market inefficiencies and behavioral factors can create opportunities for predictive modeling. There examples of top quantitative hedge funds being able to consistently find patterns in the market, delivering returns outperforming the market year-on-year.

In this project, we aim to explore the intersection of financial markets and machine learning by experimenting with multiple deep learning models and feature engineering methods. The problem formulation and datasets are derived from the Kaggle competition *Trading at the Close*. Our approach is inspired by the top 10 submissions on the leaderboard [4].

Initially, feature engineering was conducted to select the most significant features for the analysis. A total of five deep learning models were implemented using the PyTorch library in Python. These models include a Neural Network (NN), Recurrent Neural Network (RNN), Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), and a Transformer model. An ensemble model is developed to leverage the strengths of each individual model.

Despite the ensemble model's predictive performance being relatively poor, the results are consistent with the top submissions in the competition. This outcome may underscore the inherent challenges of predicting financial markets based on historical price data, thereby supporting the Efficient Market Hypothesis.

2 Background

2.1 Efficient Market Hypothesis

The Efficient Market Hypothesis (EMH) posits that financial markets efficiently reflect all available information, making it impossible for investors to consistently outperform the market on a risk-adjusted basis. According to EMH, asset prices always incorporate relevant information, so stock prices are fair and represent their true value [2].

Eugene F. Fama argues that new information is quickly integrated into stock prices, rendering both technical and fundamental analysis ineffective for consistently achieving superior returns. Stock price movements are thus unpredictable, following a "random walk," since they only change in response to unforeseen information [5].

2.2 Related work

Although there is a lot of material discussing stock predictions using deep learning, the exact problem at hand is rather unique, hence the motive for a Kaggle competition [4]. As such, the overall inspiration for the project as well as some high-level architecture primarily stems from various implementations on the leaderboard of the competition.

The solution by GitHub user `nimashahbazi`, placing 7:th in the competition and winning $25,000, has been used as the main inspiration for feature engineering and model selection [6]. Due to the inherent complexity in understanding the underlying data, many of `nima`:s suggestions for feature engineering are used in the final models. Further, to be able to benchmark the project results against solutions in the leaderboard despite using a slimmed dataset, `nima`:s models were trained and evaluated using the same features train and test data set as our final models.

Another solution by Kaggle user `hyd` inspired a final ensemble model with the aim of leveraging strengths of different models [3]. He also shared some common pitfalls that helped us navigate which paths to avoid, amongst other things avoiding using too large transformer models.

3 Data

The dataset used in the project was a dataset of 5,237,981 data points containing historic daily 10-minute closing auctions for stocks traded on the NASDAQ stock exchange, retrieved from Kaggle [4]. The 10 numerical and the single categorical features can be found in Table 2 in `Appendix` along with a description of each feature.

The target variable measures weighted average price movement of the stock over the next 60 seconds in 0.01% increments. It is defined as the difference between the 60-second future change in the stock's WAP and the WAP of a synthetic NASDAQ index, or more specifically

$$Target = \left(\frac{StockWAP_{t+60}}{StockWAP_t} - \frac{IndexWAP_{t+60}}{IndexWAP_t} \right) * 10000. \tag{1}$$

The dataset consisted of 200 stocks over 481 dates, mapped by `stock_id` and `date_id` respectively, and much of the initial preprocessing of data included creating sequences of data mapped to the correct `stock_id` and `date_id`. Over model iterations, the preprocessing and feature engineering became more sophisticated, explained further in Sec. 4. The dataset was for the most parts intact but contained some missing values, e.g. the target variable in which 88 values were missing. These missing values were simply handled by setting them equal to the median of that variable.

4 Methods and experiments

All of the code in this project was written in `Python`, making use of data modeling libraries such as `NumPy` and `Pandas`, data processing tools as `sklearn` and `CatBoost`. Deep learning models were implemented using the machine learning library `PyTorch`.

4.1 Initial preprocessing

An early discovered limitation was the incapability to handle the entire dataset due to time taken when training models. Due to the sequential nature of the data, we chose to remove data based on `date_id` with the aim of not losing valuable sequences of data. The final data set were cut at a threshold of `date_id`:s larger than 300, slimming down the total dataset to 3,258,035 data points. Out of these data points, 2,710,455 ($\sim 83.3\%$) were used for training whereas the remaining 547,580 ($\sim 16.6\%$) were used as test data.

The categorical variable was processed by creating embedded vectors, converting the 55 unique timestamps/categories into embedded vectors of size 10. This approach was chosen over one-hot encoding due to its greater efficiency, which was necessary given the limitations in our computational power. The embedding allowed the model to interpret the time until close as a state, providing a more nuanced representation of the temporal data.

4.2 Baseline models

A baseline model was created to establish a baseline for comparison with more complex models. For this model, training was done over two epochs with no feature engineering.

4.2.1 Neural network (NN)

The constructed NN consisted of 2 hidden layers with 64 and 32 nodes each, with ReLu as activation function. The data used at first was preprocessed only by making the categorical variable *seconds in bucket* embedded, using the embedding function in pytorch.

Training the NN on unnormalized data results in high initial losses, evident by an initial peak and slow decay in smooth loss. To address this, the data was normalized and the NN retrained. The normalization enables more effective training from the start. However, the increasing smooth loss over time indicates that, despite better initialization, the model still struggles to train effectively. The final MAE on the test set for the model trained on unnormalized data was 6.7892, compared to 6.7916 for the normalized data. Thus, while normalization eases initial learning, it makes little difference to the final performance.

4.3 Feature Engineering

Drawing inspiration from several contributors on Kaggle, it was seen that feature engineering tended to have a great impact on the model results, especially in combination with CatBoost feature importance for feature selection [3]. An important note is that the computing power available to us was far from the that of the top contributors on Kaggle, limiting the amount of features that could be implemented.

In addition to the more obvious normalization of input data, the feature engineering process was a large part of the preprocessing for the more complex models. A total of 159 unique features were created, categorized into:

- Feature pair ratios, products, sums and differences
- Feature pair imbalances
- Feature pair deviations within seconds
- Feature cumulative sums
- Lagging of target and input features

Using `CatBoost`, feature importance was calculated for each of the 159 features, resulting in a ranked list of features which is visualized in Fig. 1 in `Appendix`.

In order to determine how many features to include in the final model, several models were trained using the same LSTM architecture and hyperparameters, but with a differing amount of features. A range of the most important features were used in the grid search where models were evaluated on the MAE. Just as with the reduction of the data set size, the range of searched values were restricted due to the increased time and computation power needed when training with many features. In the end, the models were trained using the 40 most important features along with the original ones for a total of 1 categorical and 44 numerical features.

4.4 More complex models

To address the sequential nature of the data, 4 more complex models were developed. They all utilized mean average error (MAE) as loss function, and the Adam optimizer with initial learning rate of $\eta = 0.001$. This is motivated by its characteristic of feature-individual learning rates and ability to not decay learning rates too fast [1].

All models below were tested both without and with dropout using various dropout rates. During the runs, no large and systematic differences could be seen between running with or without dropout, neither in the loss plots or final test MAE. This was common for all models and was assumed to stem from (1) the fact that features already were narrowed down to the few "most important" (2) the fact that the data was noisy by nature and thus already providing a fair amount of regularization. However, the final models were trained with the dropout-rate that yielded the smallest test MAE.

For the complex models, there were much more computational power needed, especially for the transformer. Therefore, only two layers of the complex models were employed, followed by 4-5 dense layers. All the models were trained over 15 epochs, both with and without feature engineering.

As will be seen in the training loss plots, the loss increases and decreases during training. Validation loss is computed using the "next" 20 days yet to be seen by the model. Early stopping may be considered, but due to the importance of seeing all available data up to the point of prediction, this approach was not pursued. Although a model with a small loss may perform very well on the training data for a short period of time, it is unlikely that it will generalize well to the test data due to the stochastic nature of stock movements.

4.4.1 Recurrent Neural Network (RNN)

One significant limitation of the NN was its inability to handle patterns and the sequential nature of the dataset, which could be critical for predicting stock prices based on historic data. To address this, a model with memory was needed, leading to the implementation of a RNN.

RNNs are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. For our experiments, we set the RNN window to 3, meaning the model considers three consecutive time steps when making predictions. After training the model on the normalized data, the RNN shows improved performance as compared to the NN.

4.4.2 Long Short-Term Memory (LSTM)

The LSTM architecture implemented for this project is designed to leverage LSTM layers to capture temporal patterns in the input data. The architecture consists of several key components.

The numerical input data and the embedded categorical data are concatenated and processed together through the LSTM layers. In the final model, the LSTM component is composed of two LSTM layers, each with a hidden size of 64. These layers are designed to maintain and update a hidden state over time, enabling the model to capture temporal dependencies in the data.

The output from the LSTM layers is taken from the last timestep, which is then passed through a series of dense layers. The final model includes five dense layers with sizes 512, 256, 128, 64, and 32, each followed by batch normalization, dropout of 0.3, and SiLU activation. A final linear layer is used to produce the output of the model, which in this case is a single value prediction.

The results obtained from the LSTM showed slightly better performance compared to the RNN model, albeit very similar development of training loss and final loss on the test set.

4.4.3 ConvNet

The ConvNet architecture implemented for this project is designed to leverage convolutional layers to capture spatial and temporal patterns in the input data. The architecture consists of several key components.

The numerical input data and the embedded categorical data are processed separately through convolutional layers. Two sets of convolutional layers were experimented with, using kernel sizes of 2 and 3, respectively. In the final model, each convolutional block consists of two convolutional layers, each followed by batch normalization, ReLU activation, and dropout. The number of filters

used in the convolutional layers is set to 16, and a dropout ratio of 0.4 is applied to prevent overfitting. Shortcut connections are included in the convolutional blocks to facilitate better gradient flow and to preserve the original input features.

The outputs from the convolutional layers are concatenated and flattened into a single vector, which serves as the input to the dense layers. The dense layers are designed to further process the features extracted by the convolutional layers. In the final model, the architecture includes five dense layers with sizes 512, 256, 128, 64, and 32, each followed by batch normalization, dropout, and SiLU activation. Again, a final linear layer is used to produce the output of the model.

4.4.4 Transformer

The Transformer architecture implemented for this project is designed to leverage self-attention mechanisms to capture complex dependencies in the input data. The architecture consists of several key components.

The numerical input data and the embedded categorical data are concatenated and processed together through the Transformer layers. In the final model, the Transformer component is composed of two Transformer Encoder layers, each with 8 attention heads and a feedforward dimension of 64. These layers are designed to capture intricate patterns in the data by attending to different parts of the input sequence, enabling the model to understand temporal and contextual relationships.

The output from the Transformer layers is taken from the last timestep, which is then passed through a series of dense layers. The final model includes five dense layers with sizes 256, 128, 64, and 32, each followed by batch normalization, dropout of 0.1, and ReLU activation. A final fully-connected layer is used to produce the output of the model.

Transformers excel at capturing long-range dependencies and contextual information within the data. Unlike traditional sequential models like RNNs and LSTMs, which process data in a fixed order, Transformers use self-attention mechanisms to weigh the importance of different parts of the input sequence dynamically. This allows them to better understand and utilize the relationships between distant elements in the sequence. Consequently, Transformers are particularly effective for tasks involving complex patterns and large amounts of data, such as natural language processing, time series forecasting, and image analysis.

4.4.5 Ensemble

Finally, an ensemble model is created to leverage the strengths of each individual model. In this approach, predictions are made on the test data using each of the models, and the final prediction is obtained by averaging these individual predictions. This method helps to mitigate the weaknesses of any single model, leading to improved generalization to unseen test data. By combining aspects of multiple models, the ensemble approach typically results in more robust and accurate predictions.

4.5 Results

Combining more sophisticated models designed for sequential data along with feature engineering and longer training time yields promising results. After training for 15 epochs, their generalization ability is evaluated using mean absolute error (MAE) for both original and feature engineered data. While feature engineering was extensive, the model seemed to find many of the designed patterns itself, providing limited improvements. Increasing model complexity from RNN to Transformers also seems to improve performance.

Histograms over the predictions of the models as well as the target were plotted, (see Fig. 2) showing that the predictions of all models were consistently less dispersed than the actual target. Predicting around zero yields on average a lower MAE, incentivising the models to pursue that approach. Even finding the correct sign is a feat when predicting movements of financial markets, with the ensemble model correctly predicting 303,387 out of 547,580 (55.41%).

Table 1: Mean Average Error Loss on Test Data Set

Model	MAE (no FE)	MAE (with FE)
RNN	5.972889	5.950669
LSTM	5.940858	5.931735
ConvNet	5.948735	5.936930
Transformer	5.939970	5.919141
Ensemble	5.935134	5.909938

5 Discussion

5.1 Comparing with Kaggle Leaderboard

Comparing the model performance with Kaggle top submissions, no large differences in MAE was seen during training serving as an indication that the model architectures are satisfactory. A larger data set, more features and longer training time would enhance the predictive power of the models. Competitors were able to retrain their models for weeks, which was not attainable within the scope of this assignment. Longer training seems to consistently decrease the loss, from ~ 6.45 loss on test set after 2 epochs to ~ 5.93 after 15 epochs. More time and compute will likely improve performance.

5.2 Generalization to Test Data

Stock price forecasting in financial markets is conditionally dependent on all information available at the current time of prediction. During training, it is therefore not viable to use test data far in the future disconnected from current hidden state of the model. Therefore, we adopted an approach where the model predicts the subsequent chronologically ordered 20 batches. For example, using information from January for predicting movements in November is not as useful as for predicting February. After training the model using data from the first 250 days, the subsequent unseen 50 days were predicted as test set.

5.3 Alternative Data Selection

An alternative approach to splitting the dataset could involve splitting by stocks rather than by dates. This method could potentially enable the model to train for longer periods, providing more data and enhancing its ability to identify intricate patterns. However, this introduces the dilemma of training on "future" data. If the training and test datasets share the same start and end dates, the model might learn patterns from the entire market during that period, limiting its generalization to unseen data.

Moreover, dividing stocks into groups based on characteristics like trading volume, price, or industry, and training on one group at a time, might help the model learn more precise patterns. Stocks within the same industry or with similar market capitalizations might display comparable behaviors. While this can improve performance on similar stocks, it is important to remember that this also restrict the model's ability to generalize.

6 Conclusion

Our findings indicate that combining sophisticated models designed for sequential data, along with feature engineering and longer training times, yields promising results. Despite extensive feature engineering, the models independently identified many patterns, providing limited improvements. However, increasing model complexity, using larger datasets and extending training times improved performance. Comparing model performance with top Kaggle submissions showed no significant differences in mean absolute error (MAE), suggesting that the deep learning models are satisfactory.

If anything, our results underscore the inherent difficulty in predicting financial markets, providing support for the Efficient Market Hypothesis. If the opposite was true, we would all be billionaires.

References

[1] D. P. Kingma and J. L Ba. 2015. *Adam: A Method For Stochastic Optimization*. International Conference on Learning Representations (ICLR).

[2] Fama, E. F. 1970. *Efficient Capital Markets: A Review of Theory and Empirical Work.* The Journal of Finance, **25**(2), 383-417.

[3] hyd. 2024. *Discussion - 1st place solution*. Kaggle. https://www.kaggle.com/competitions/optiver-trading-at-the-close/discussion/487446.[Accessed 2024-05-14]

[4] Kaggle. 2023. *Optiver - Trading at the Close.* Kaggle. https://www.kaggle.com/competitions/optiver-trading-at-the-close/overview.[Accessed 2024-05-13]

[5] Malkiel, B. G. 2003. *The Efficient Market Hypothesis and Its Critics.* Journal of Economic Perspectives, 17(1), 59-82.

[6] nimashahbazi. 2024. *optiver-trading-close.* Github. https://github.com/nimashahbazi/optiver-trading-close/tree/master.[Accessed 2024-05-14]

Appendix

A. Original Features

Table 2: Feature Names and Descriptions

Variable Name	Type	Description
imbalance_size	Num	The amount unmatched at the current reference price (in USD).
imbalance_buy_sell_flag	Num	An indicator reflecting the direction of auction imbalance: buy-side imbalance (1), sell-side imbalance (-1), no imbalance (0).
reference_price	Num	The price where paired shares are maximized, imbalance is minimized, and distance from bid-ask midpoint is minimized.
matched_size	Num	The amount that can be matched at the current reference price (in USD).
far_price	Num	The crossing price that maximizes the number of shares matched based on auction interest only.
near_price	Num	The crossing price that maximizes the number of shares matched based on auction and continuous market orders.
bid_price	Num	Price of the most competitive buy level in the non-auction book.
ask_price	Num	Price of the most competitive sell level in the non-auction book.
bid_size	Num	The dollar notional amount on the most competitive buy level in the non-auction book.
ask_size	Num	The dollar notional amount on the most competitive sell level in the non-auction book.
wap	Num	The weighted average price in the non-auction book.
seconds_in_bucket	Cat	The number of seconds elapsed since the beginning of the day's closing auction, always starting from 0.

B. Feature Importance

After performing feature engineering on the entire training data set, we are left with 159 unique features of categorical or numerical qualities. CatBoost was used to identify the most influential features for predicting the `target`. Here are the top 30 most influential features, although the top 40 were used together with the original data for training the subsequent models.

#	Feature	Importance
1	seconds_in_bucket	4.5497
2	deviation_from_median_price_wap_difference_eng	3.7769
3	date_id	3.0926
4	price_wap_difference_eng_cumsum	2.8596
5	ask_price_bid_price_imb_pr_	2.7965
6	spread_eng	2.7011
7	weighted_imbalance_eng_cumsum	2.4616
8	weighted_imbalance_eng	2.2149
9	deviation_from_median_wap	2.1201
10	deviation_from_median_weighted_imbalance_eng	1.9810
11	deviation_from_median_ask_size_bid_size_imb_sz_	1.9771
12	bid_size_matched_size_imb_sz__cumsum	1.8744
13	deviation_from_median_ask_price	1.6942
14	bid_ask_ratio	1.6069
15	ask_size_matched_size_imb_sz_cumsum	1.5918
16	deviation_from_median_near_price	1.5645
17	deviation_from_median_bid_price	1.5606
18	deviation_from_median_imbalance_size_matched_size_imb_sz_	1.5486
19	spread_eng_cumsum	1.4908
20	deviation_from_median_matched_size	1.4875
21	bid_price	1.4443
22	ask_price_reference_price_imb_pr_	1.3973
23	ask_size_bid_size_imb_sz_	1.3674
24	weighted_imbalance_eng_diff_lag1	1.3663
25	imbalance_ratio	1.2923
26	bid_price_wap_imb_pr_	1.2073
27	deviation_from_median_reference_price	1.1983
28	bid_price_reference_price_imb_pr_	1.1367
29	bid_price_near_price_imb_pr_	1.1330
30	matched_size_to_total_size_ratio_eng_cumsum	1.0983

Table 3: Feature Importance Table

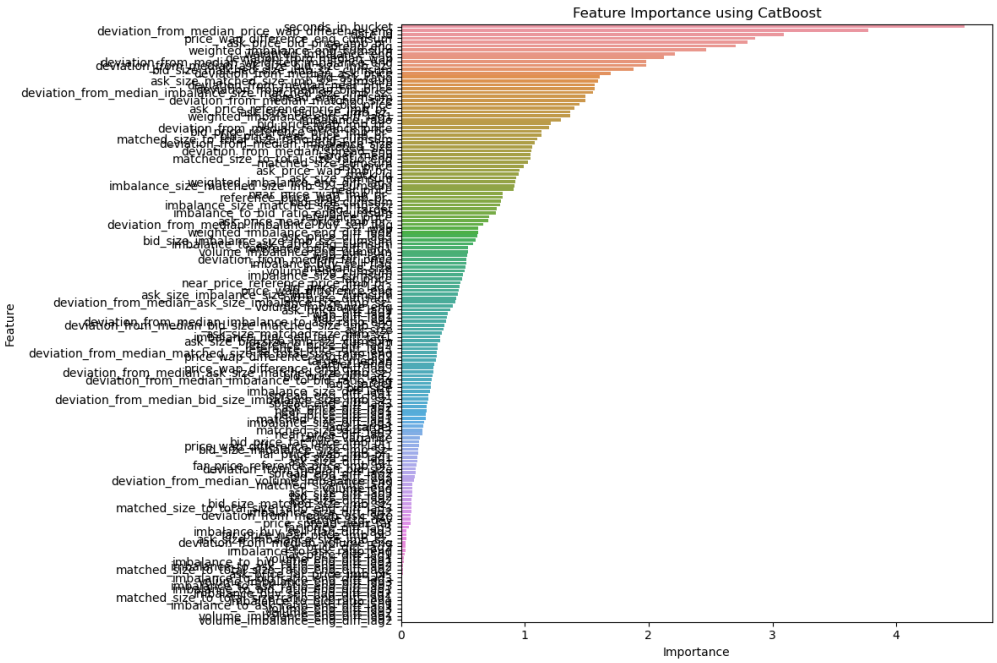
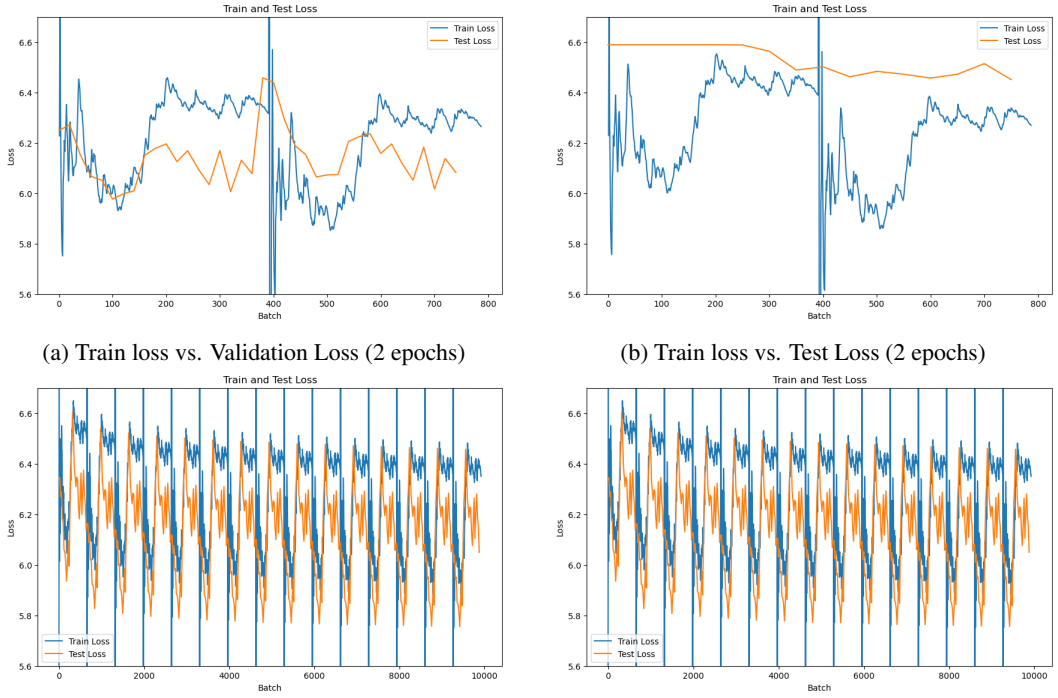


Figure 1: Results from CatBoost Feature Selection

C. Loss Plots

During training of the models the running loss was computed for each batch. The validation set consisted of 20 batches directly sequential to the current model state, although yet unseen by the model, and was computed every 20 batches. We see very small deviations between model characteristics during training, with some performing slightly better than the others. Sometimes the validation loss is lower than the training loss, which is likely due to unpredictable market movements that alter the state of the model for the worse.

We also plot the test loss against the training loss to see how it develops over time. Since it is never seen by the model, the predictive accuracy is much lower than for the validation loss. Although, it does decrease very slowly as the number of epochs rise. Since the loss plots from all models are almost identical, only results from the LSTM model are shown below.



(a) Train loss vs. Validation Loss (2 epochs)

(b) Train loss vs. Test Loss (2 epochs)

(c) No Feature Engineering (15 Epochs)

(d) Feature Engineering (15 Epochs)

Figure 2: LSTM Model Training Progression

D. Visualizing Predictions

To evaluate the performance of the models against the true labels, predictions are made on the test set and plotted in histograms. We plot 6 histograms, corresponding to: ConvNet, LSTM, RNN, Transformer, Ensemble and Targets. Note that the absolute values of the actual targets are way larger than the predicted values. Standardizing the targets before predicting could be a possible solution, but since even the signs are difficult to predict (55.41% correct) this is not a viable option.

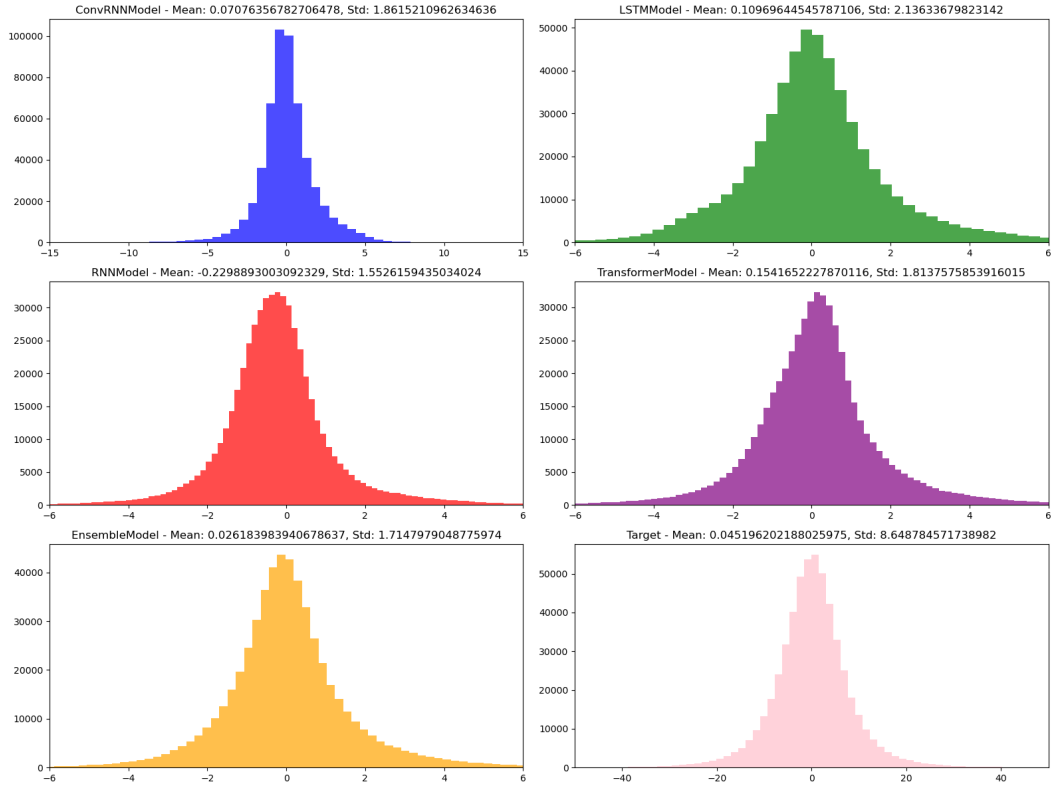


Figure 3: Histograms of test set predictions