Neural Networks for Time Series Prediction

1. Background

Time series prediction is used in a number of industries ranging from finance to healthcare. The ability to correctly forecast trends and events is highly desirable as it gives decision-makers an edge in developing and planning new strategies. Unfortunately, developing an accurate forecasting model is difficult due to the non-linear and non-stationary nature of most real-life datasets. Despite this challenge, researchers have developed several techniques that can be used in time series prediction.

In recent years, neural networks have become an increasingly popular area of research for many learning problems. Of these neural network architectures, Long Short Term Memory (LSTM) neural networks are the most frequently used for time series prediction. Their ability to handle long sequences of data makes them ideal for use with temporal datasets ranging over large time periods.

This project expands on the findings described in "Time Series Prediction Using LSTM Deep Neural Networks", an article written by Jakob Aungiers of Altium Intelligence [1]. The article, and its accompanying code [2], examined the use of a stacked LSTM architecture to model the closing price of the S&P 500 Equity Index over time. Three new LSTM-based architectures will be implemented in an attempt to obtain improved performance. These proposed architectures will then be evaluated using the original architecture from the article as a benchmark.

2. Dataset

The S&P 500 Equity Index from January 2000 to September 2018 will be used in this analysis. This dataset contains the daily Open, High, Low and Close prices, as well as the Volume. While the code provided with this article is well-written, much of it is designed to handle very specific types of neural network layers. This means that it cannot support the architectures proposed in this project and will not be used, unless clearly indicated in the accompanying Jupyter Notebook. One exception to this is the “DataLoader” class which is used to both load and normalize the S&P 500 data.

The closing price "is a constantly moving absolute price of the stock market" [1]. A model trained on this data without normalization would not converge. To prevent this, each training/testing window will be normalized to represent the percentage change from the start of the window. The data will then be denormalized after prediction to get an actual close price value. The following equations will be used for normalization and denormalization:

$$Normalization: n_i = \frac{p_i}{p_0} - 1$$

$$De\text{-}normalization: p_i = p_0(n_i + 1)$$

$$where\ n\ is\ the\ normalized\ list\ (window)\ of\ price\ changes$$
$$and\ p\ is\ the\ raw\ list\ (window)\ of\ adjusted\ daily\ returns$$

3. Methodology

In addition to the original architecture discussed in the article, three new LSTM architectures will be implemented. All architectures will be trained using the same process described in the article. The dataset will be split into a training set and a test set. The first 85% of the data points will be used for training and the remaining 15% will be used for testing. A sliding window of size *n* will be used in the model, where the *n* data points in the window will be used to predict the (*n+1*)th point. This also upholds the temporal nature of the dataset by ensuring that data points from the future will not be used to predict the past. For the purpose of this study, the size of the time window will be the same as that used in the article, n=49 [1].

For each of the architectures, three different optimizers (SGD, RMSProp and Adam) will be used to determine the effects of different gradient descent algorithms on the performance of the model. Mean squared error (MSE) will be used as the cost function for this analysis. MSE is the average squared difference between the predicted and actual values. This function is easily differentiable, making it ideal for gradient descent.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(predicted_i - actual_i)^2$$

$$where\ m\ is\ the\ total\ number\ of\ samples\ and\ i\ is\ the\ i^{th}\ sample$$

Each model will be trained for 100 epochs, unless otherwise stated. This value was chosen arbitrarily but was found to provide sufficient time for the models to converge. The same number of epochs was used to train all of the architectures in this study.

The SGD, RMSProp and Adam optimizers were selected for this analysis because they are three of the most commonly used algorithms. All three optimizers aim to minimize the cost function using gradient descent but they differ in how they are updated after each iteration. SGD is one of the simplest gradient descent algorithms but is slower to converge and tends to get stuck in saddle points. Its behaviour is also heavily dependent on how it is initialized. RMSProp and Adam are very similar and are adaptive algorithms. They both tend to converge faster than SGD, with Adam slightly outperforming RMSProp due to its bias-correction. In general, the best optimizer for a task depends on the dataset. The behaviour of each optimizer and how well it converges also depends on how it is initialized [3].

Hyperparameter Tuning

When implementing a neural network architecture, there are several different hyperparameters that can be changed such as the learning rate, kernel size and the number of neurons used in each hidden layer. In order to facilitate comparison, most of the hyperparameter values from the original architecture will be used and will be kept constant throughout the analysis. While modifying the

parameter values for the new architectures would likely improve their performance, the large number of variations would be difficult to evaluate and compare.

Several hyperparameters will be modified as part of this study. For simplicity, only one hyperparameter will be tuned for each architecture. The learning rate for each optimizer will be tuned using the original architecture. From here, the best learning rate for each optimizer will be used to train the proposed architectures. The optimizer learning rate is one of the most important hyperparameters for neural networks. A learning rate that is too small will result in long training times while a learning rate that is too large could result in an unstable result [3].

The second hyperparameter that will be tuned is the number of neurons in each hidden layer. This hyperparameter will only be used for the proposed LSTM autoencoder architectures and will be varied to determine the best configuration. The number of neurons within each hidden layer is important to obtain the best accuracy without overfitting the training dataset.

Finally, the number of filters and the kernel size used in the proposed CNN-LSTM architecture will be tuned. The number of filters is the number of neurons in the CNN layer, while the kernel size represents the number of timesteps that will be considered in each convolution window.

The choice of hyperparameter values is dependent on both the application as well as the dataset. For this analysis, grid search will be used to determine the best hyperparameter values from a provided list of variations. These parameters will be optimized using k-fold cross validation on the training set, where k=5. The temporal dependencies of the dataset were maintained by ensuring that for each split, the train fold is chronologically before the test fold. Figure 1 presents an example of how this was done for 3-fold cross validation [4].

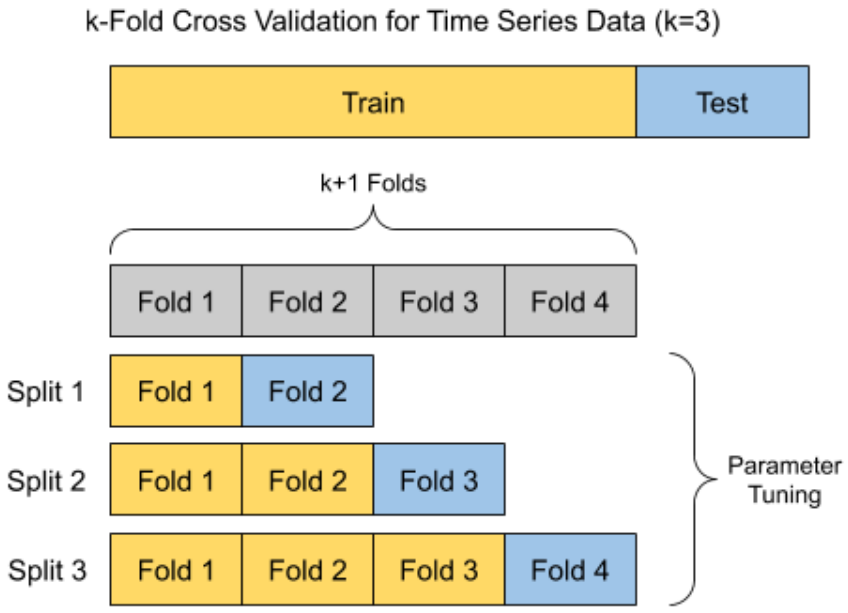


Figure 1: k-fold cross validation for time series data (where k=3)

While k-fold cross validation prevents the model from overfitting the dataset during the parameter tuning process, it is not as comprehensive as nested cross-validation. Despite this, nested cross-validation was not performed for this analysis due to time and computational constraints. When training the original architecture for 100 epochs it was found that the average training time was 30 minutes. The most computationally expensive scenario would occur when training the learning rate for each of the three optimizers using this architecture. In this case, nested cross-validation with an inner and outer loop of size five would require approximately 150 hours (3 (number of optimizers) * 4 (learning rate options) * 0.5 hours * 5 (inner loop) * 5 (outer loop)).

Performance Metrics

The performance of these architectures will be evaluated using three metrics: root mean squared error (RMSE), mean absolute percentage error (MAPE) and training time. For all three of these metrics, a lower value is more desirable and indicates better performance.

RMSE is one of the most commonly used metrics for regression tasks. It is the square root of the MSE, which is also being used as the cost function for this study. One of the advantages of both RMSE and MSE is that the errors are squared, meaning that even a small difference between the predicted and target values is penalized. While this is useful, a few bad outliers could lead to very skewed results and may not be reflective of the overall performance of the model. Despite this, RMSE (and by extension, MSE) is one of the best regression metrics in practice and will be prioritized during this analysis [5].

$$RMSE = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(predicted_i - actual_i)^2}$$

$$where\ m\ is\ the\ total\ number\ of\ samples\ and\ i\ is\ the\ i^{th}\ sample$$

MAPE is the average of the absolute percentage errors between the predicted and actual values. Historically, MAPE has been a popular metric for evaluating forecast accuracy. It is easy to interpret because it can be presented as a percentage [6]. However, MAPE has several disadvantages. It is undefined if the actual value is zero, as it would result in division by zero. MAPE is also biased towards models that under-forecast; errors due to under-forecasting are bounded whereas over-forecasting errors are not [7].

$$MAPE = \left(\frac{1}{m}\sum_{i=1}^{m}\frac{|actual_i - predicted_i|}{|actual_i|}\right) \times 100\%$$

$$where\ m\ is\ the\ total\ number\ of\ samples\ and\ i\ is\ the\ i^{th}\ sample$$

While RMSE and MAPE both measure the accuracy of the model, training time is used to determine the practicality of implementing a model. For example, a model that generates predictions with perfect accuracy but takes days to train may not be ideal for use in practice where there are time and

computational constraints. Although this is not a major constraint for this study, the training time will still be considered when assessing the performance of different architectures.

4. Original Architecture

The original architecture presented in the article is simple stacked LSTM network consisting of 3 LSTM layers followed by a single fully connected layer. This architecture also uses dropout after the first and last LSTM layers. In this case, 20% of the layer outputs are randomly selected and dropped in order to prevent overfitting during training. The use of dropout after the first and last LSTM layers in the network is repeated for the proposed architectures. For this analysis, the dropout rate was not tuned. It was kept at 20% for all architectures.

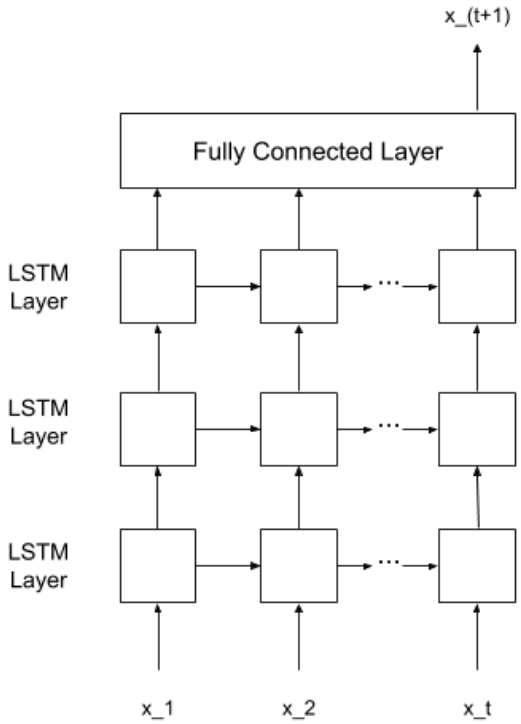


Figure 2: Original stacked LSTM architecture.

In order to establish a baseline for comparison with the new architectures, the best learning rate for each of the three optimizers (SGD, RMSProp and Adam) was determined using grid search with 5-fold cross validation. While most of the architectures were trained for 100 epochs, this grid search was only run for 50 epochs due to time constraints. The average MSE score for each optimizer and learning rate is shown in Figure 3 below. Only four different learning rate values were tested: 0.01, 0.001, 0.0001 and 1e-05.

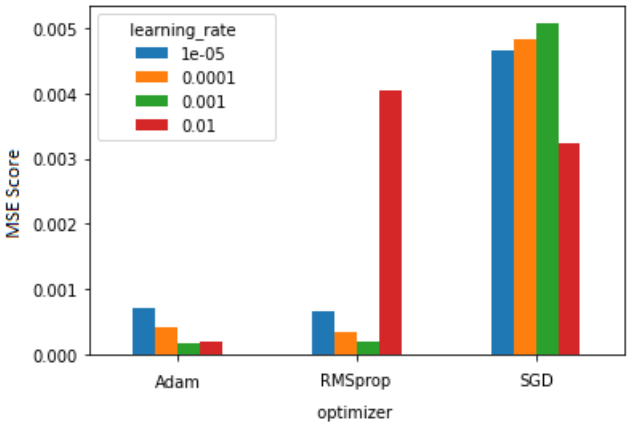


Figure 3: Results of using grid search to find optimal learning rate for each optimizer.

Based on the results of grid search, the optimal learning rates for SGD, Adam and RMSProp are 0.01, 0.001 and 0.001 respectively. Similarly, the graphs in Figures 4, 5 and 6 show the learning curves for each of the optimizer and learning rate combinations. These plots were generated separately, by training the model for 100 epochs and using the entire training and test datasets. In general, these plots confirm that performing grid search for 50 epochs, while not ideal, is sufficient for the model to reach convergence and obtain the best learning rate. However, this was not true for RMSProp.

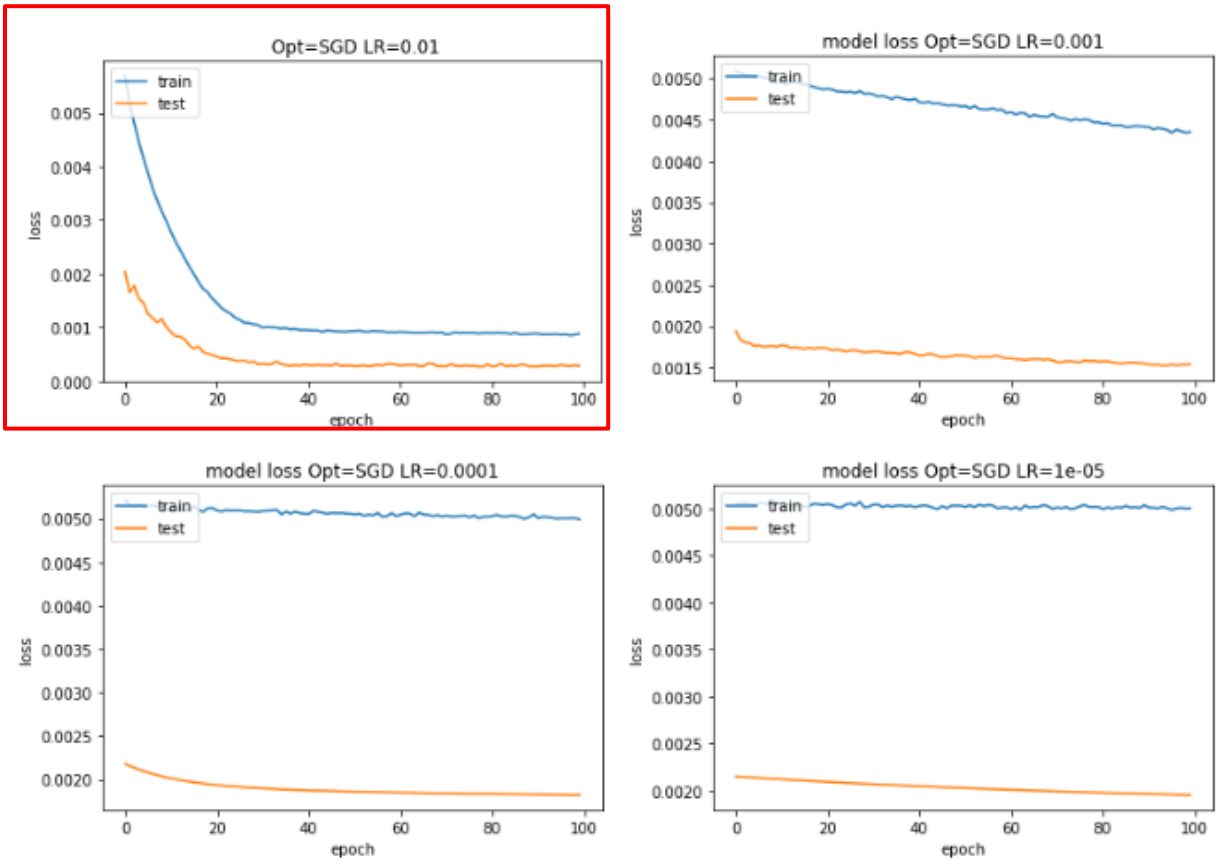


Figure 4: Learning curves for SGD optimizer using various learning rate values. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal learning rate as identified via grid search.

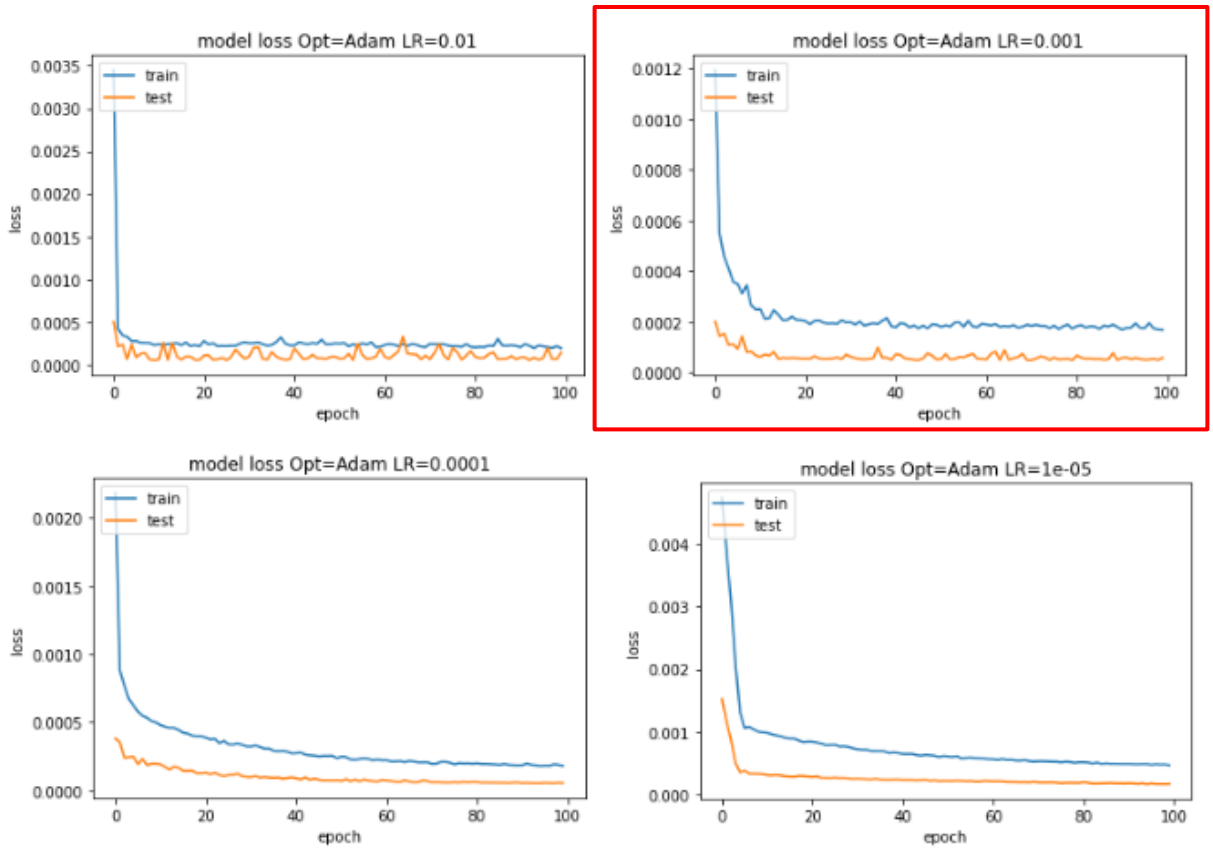


Figure 5: Learning curves for Adam optimizer using various learning rate values. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal learning rate as identified via grid search.

The learning curves for the RMSProp optimizer (Figure 6) were particularly interesting. The performance of the optimizer for the highest and lowest learning rates were clearly sub-par. A learning rate of 0.01 resulted in underfitting while a learning rate of 1e-05 caused very slow convergence. The performance of RMSProp with a learning rate of 0.001 was very similar to that of an optimizer of 0.0001. In fact, it had the best performance until it reached 90 epochs, at which point the MSE loss began to increase once again.

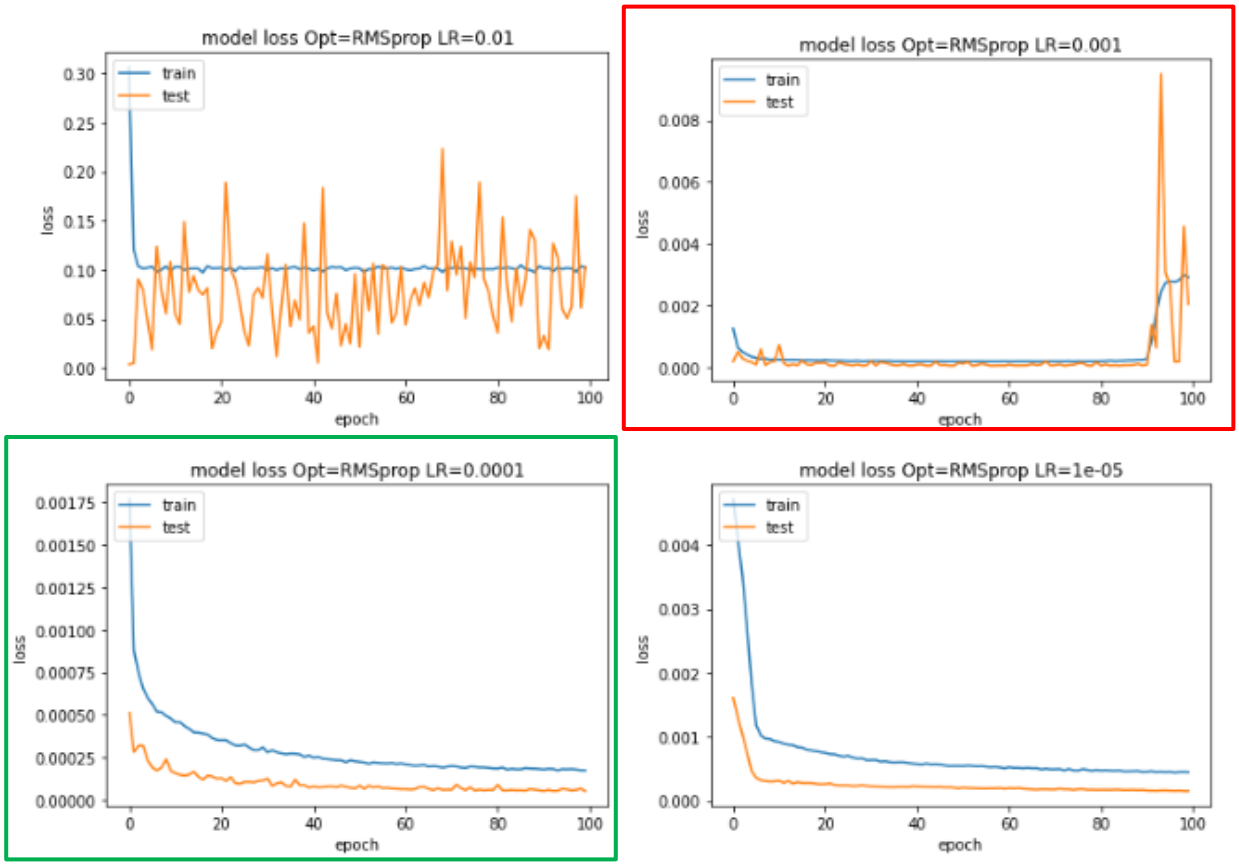
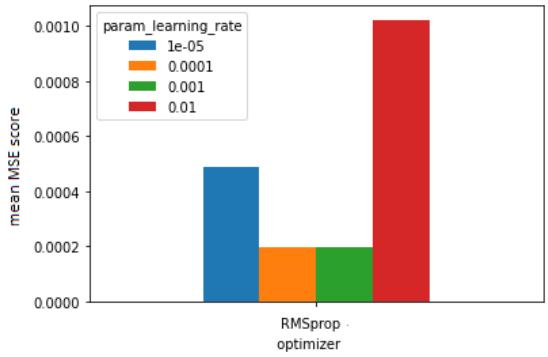


Figure 6: Learning curves for RMSProp optimizer using various learning rate values. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal learning rate as identified via grid search for 50 epochs. The green box indicates the optimal learning rate identified via grid search for 100 epochs.

Since all of the models would be trained for 100 epochs, grid search was repeated for RMSProp using 100 epochs rather than 50 in order to verify the ideal learning rate for this optimizer. From here it was found that the ideal learning rate for 100 epochs was 0.0001.



learning_rate	mean_test_score	std_test_score	mean_score_time
0.01	0.001024	0.000788	1.350635
0.001	0.000199	0.000184	1.672123
0.0001	0.000195	0.000120	1.926943
1e-05	0.000486	0.000337	2.413277

Figure 7: Results of using grid search with 100 epochs to find optimal learning rate for RMSProp.

While grid search was repeated for RMSProp, it was not repeated for the other optimizers. The learning curves for SGD and Adam indicate that the performance of the model does not change significantly after

more than 50 epochs. This is a potential limitation of this analysis because the learning curves were obtained after a single run and may not reflect the average case.

After analysis it was found that the best learning rates for SGD, Adam and RMSProp are 0.01, 0.001 and 0.0001 respectively. Using the best learning rates for each optimizer, the performance of the optimizer was evaluated by comparing the resulting closing price predictions against the actual values. The predicted and actual values are shown in the figure below, as well as the performance metric values for each optimizer.

Metric	Adam	RMSprop	SGD
RMSE	0.00738	0.00765	0.0165
MAPE	36.8	43.1	100
Time (min)	35.6	35.6	35.2

Figure 8: S&P 500 predictions using original architecture. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

The training time for all three optimizers was very similar. The difference in performance is primarily seen in the RMSE and MAPE values. The SGD optimizer had the worst performance while Adam and RMSProp performed well and had similar RMSE values. As discussed previously, the Adam and RMSProp optimizers are both adaptive algorithms although Adam tends to outperform RMSProp.

5. LSTM Autoencoder

Autoencoders are often used to identify the most important features within a dataset. It is expected that this property, combined with the LSTMs' ability to handle sequential data, will improve the performance of the model [8]. By allowing the model to focus on the important features, the resulting predictions will be more accurate.

Two different architectures were implemented: a shallow autoencoder and a deep autoencoder. In both architectures, dropout was used after the first and last layers, following the same design as the original architecture.

Shallow Autoencoder

The shallow autoencoder was implemented with two LSTM layers. One layer acting as the encoder and a second layer acting as the decoder. This architecture is also shown in the diagram below.

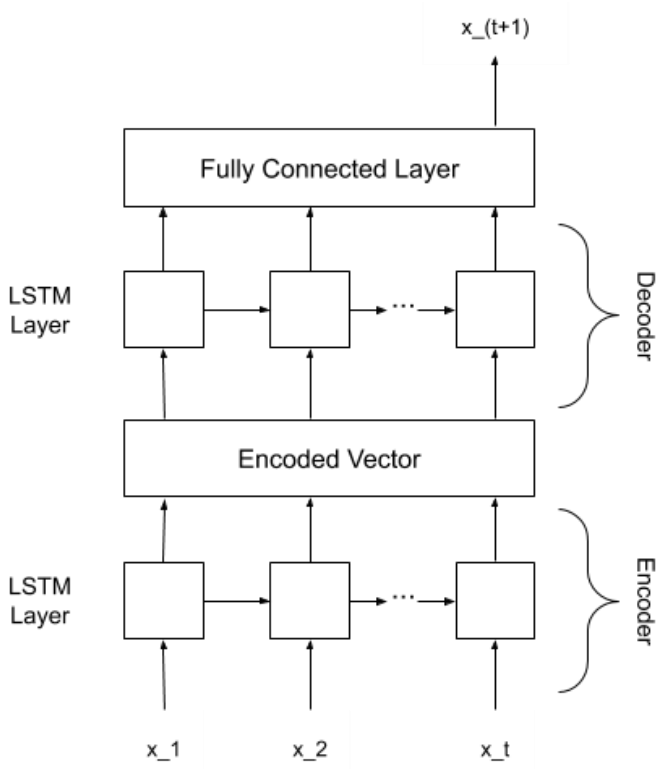
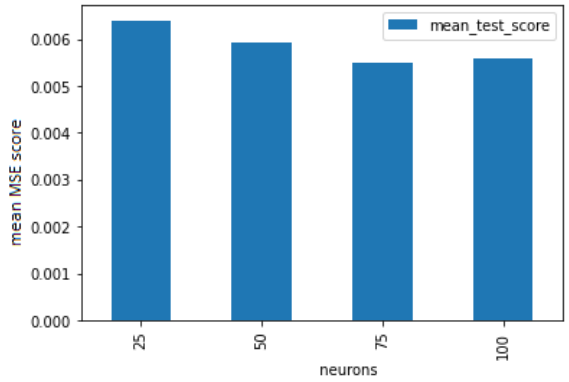


Figure 9: Shallow LSTM autoencoder architecture.

The number of neurons in the layers were tuned using grid search and 5-fold cross validation for 100 epochs. For simplicity, only the Adam optimizer was used to tune this hyperparameter. The results are summarized in the diagram below. Four different configurations were considered: 25, 50, 75 or 100 neurons. After using grid search, results showed that having 75 neurons in each of the hidden layers had the best performance. This number of neurons had the lowest MSE after cross validation, as shown in Figure 10 below. The learning curves for each of the neuron combinations can also be found in Figure 11. These curves were generated using the full train and test data and were not generated during grid search.



neurons	mean_test_score	std_test_score	mean_score_time
25	0.006393	0.001776	8.383675
50	0.005939	0.001654	9.355790
75	0.005489	0.001387	9.994865
100	0.005584	0.001884	11.012161

Figure 10: Results of using grid search to determine number of hidden layer neurons in shallow LSTM autoencoder.

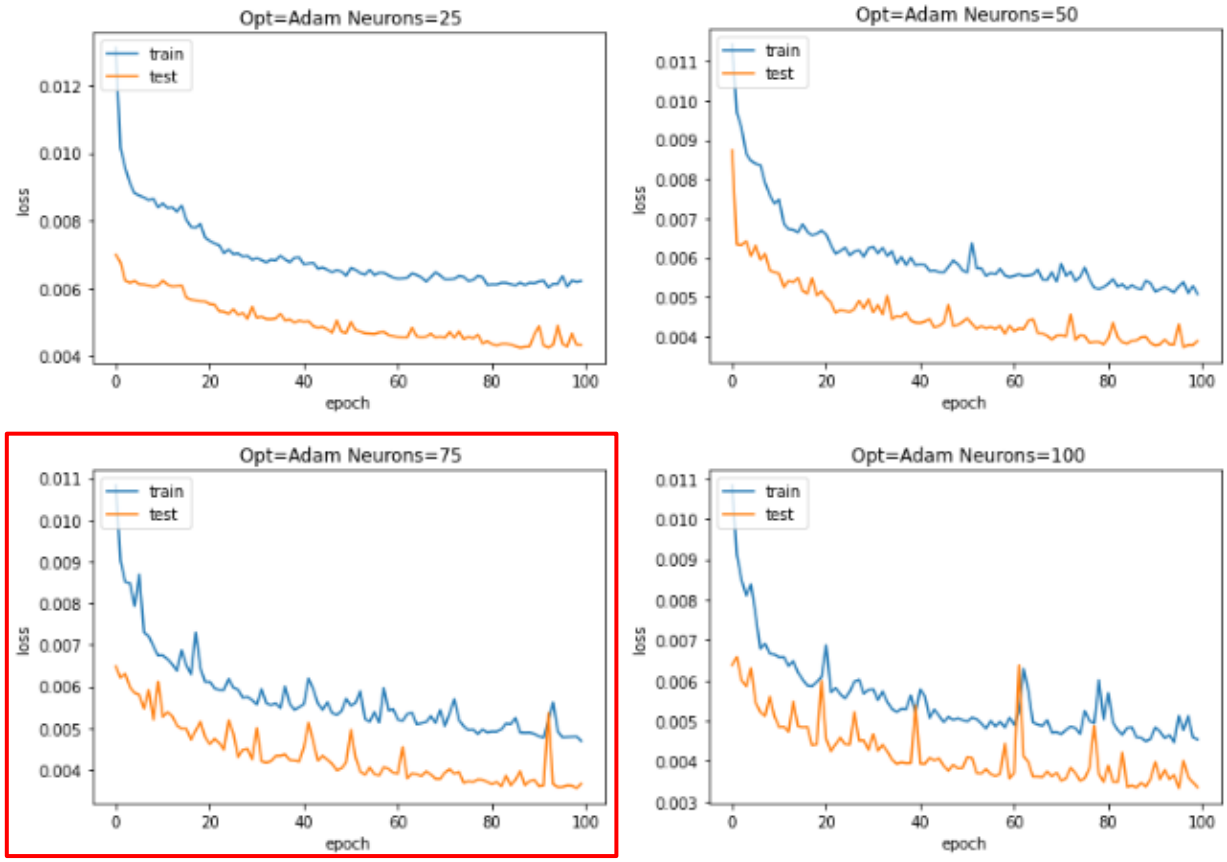


Figure 11: Learning curves after tuning number of neurons in each hidden layer for the shallow LSTM autoencoder. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal number of neurons as identified via grid search.

Using the results above, the experiment was repeated using all three optimizers for a shallow LSTM autoencoder with 75 neurons in each hidden layer. The performance of the model for each optimizer is shown in the figures below.

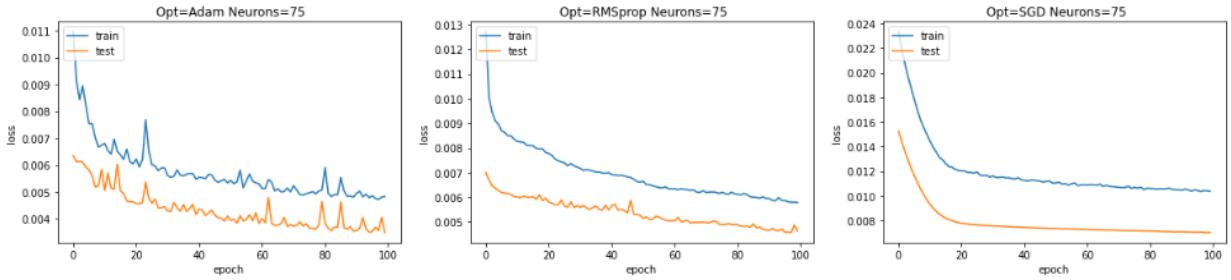


Figure 12: Learning curves showing MSE loss while training the shallow LSTM autoencoder using the Adam, RMSProp and SGD optimizers.

Metric	Adam	RMSprop	SGD
RMSE	0.01542	0.01925	0.02275
MAPE	83.06041	107.01989	78.83110
Time (min)	18.63086	18.20436	17.97906

Figure 13: S&P 500 closing price predictions for shallow LSTM autoencoder. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

Once again, the Adam optimizer outperformed both RMSProp and SGD in terms of RMSE. However, the SGD optimizer had the lowest MAPE value indicating that the absolute difference between the actual and predicted values was generally low, but there may have been many outliers that were captured when calculating RMSE.

Deep Autoencoder

The deep autoencoder was implemented with four LSTM hidden layers. The first two layers act as the encoder while the last two layers perform the decoding operation.

For this architecture, only the number of neurons in the middle LSTM layers were varied. Three different possibilities were considered for these layers: 25, 50 or 75 neurons. This was tuned using grid search and 5-fold cross validation over 100 epochs. As before, only the Adam optimizer was used. The number of neurons in the first and last hidden layer were kept constant at 100 neurons, the same as in the original architecture. While this limits the analysis and may prevent a truly optimal architecture from being found, it was done to limit the number of variations due to computational and time constraints.

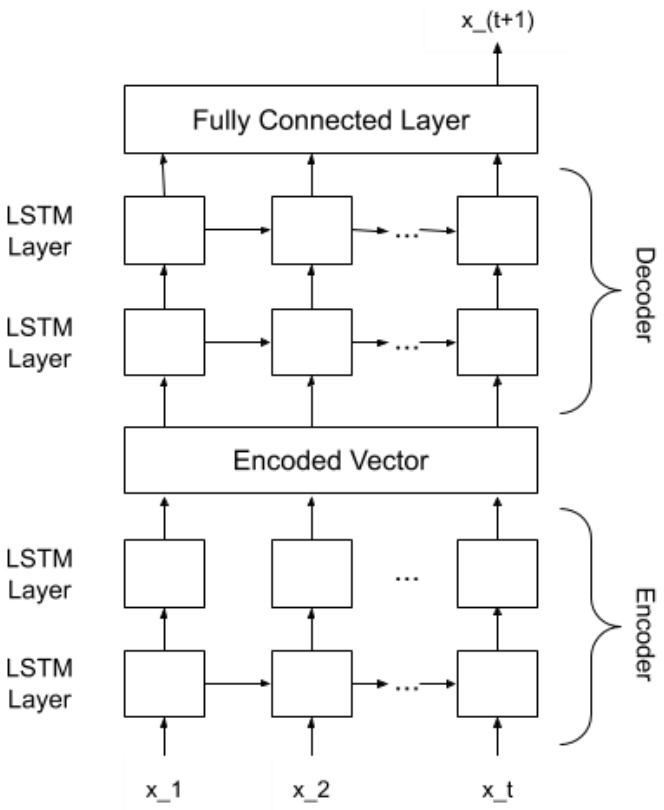
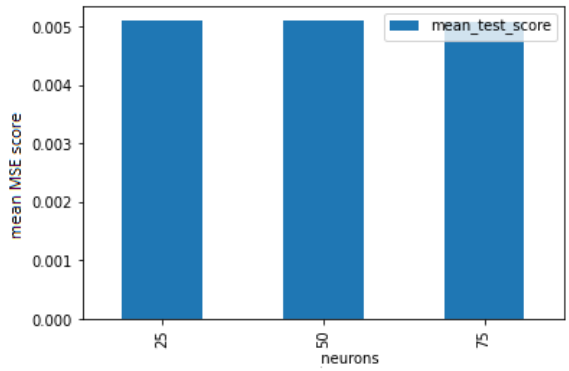


Figure 14: Deep LSTM autoencoder architecture.

After using grid search it was found that the neuron variations that were considered did not have a significant impact on the performance of the model. The architecture using 75 hidden layer neurons was selected as it had the lowest average MSE and a relatively low standard deviation. The learning curves for these variations were also used to confirm the performance.



neurons	mean_test_score	std_test_score	mean_score_time
25	0.005097	0.001801	16.043657
50	0.005088	0.002017	17.944959
75	0.005066	0.001860	18.537731

Figure 15: Results of using grid search to determine number of hidden layer neurons in deep LSTM autoencoder.

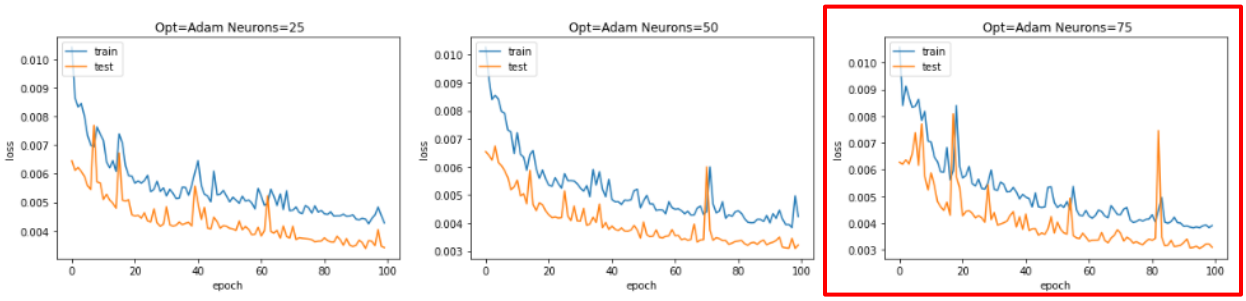


Figure 16: Learning curves after tuning number of neurons in each hidden layer for the deep LSTM autoencoder. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal number of neurons as identified via grid search.

From here, the model was run using all three optimizers. The learning curves and resulting predictions are shown in Figures 17 and 18. Once again, the Adam optimizer was found to outperform both RMSProp and SGD with respect to RMSE. As with the shallow architecture, SGD had the lowest MAPE value; however, it's MAPE score was not much lower than that of the Adam optimizer. Overall, the Adam optimizer was found to have the best performance.

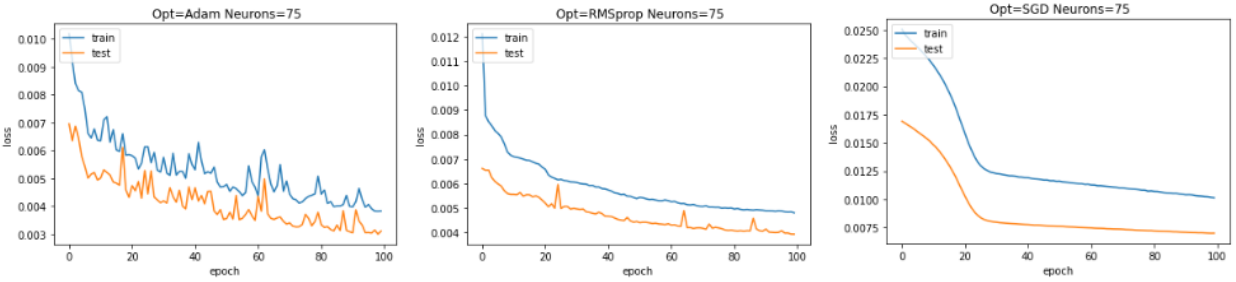


Figure 17: Learning curves showing MSE loss while training the deep LSTM autoencoder using the Adam, RMSProp and SGD optimizers.

Metric	Adam	RMSprop	SGD
RMSE	0.01417	0.01544	0.03428
MAPE	94.36997	102.55308	92.37433
Time (min)	43.65564	43.34419	43.10764

Figure 18: S&P 500 closing price predictions for deep LSTM autoencoder. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

It is clear that this architecture does not perform as well as the baseline. This is especially true when looking at the performance of the version using the SGD optimizer. Despite the low MAPE for this optimizer, it's RMSE is the highest. Similarly, the learning curve for SGD shows that it flattens after approximately 20 epochs and does not improve from there. It is likely that the performance of all three

optimizers would improve if their hyperparameters were tuned for this specific architecture; however, that was not done for this analysis.

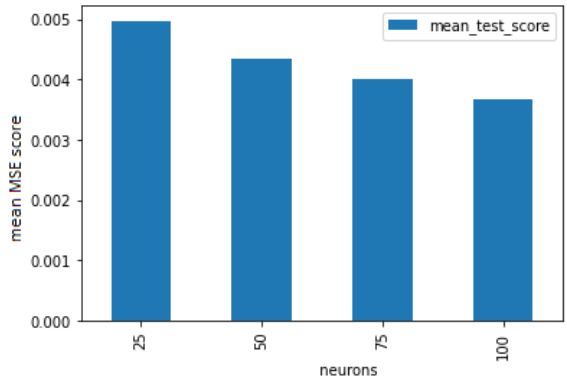
6. Bidirectional LSTM Autoencoder

The bidirectional LSTM (BLSTM) autoencoder is similar to the architecture from the previous section. In fact, the only difference between the two is that for a BLSTM autoencoder, the input will be read both forwards and backwards for each LSTM layer. Although this will take longer to train, reading the data in multiple ways could provide the network with additional context and may result in better predictions.

Both the shallow and deep architecture used for the LSTM autoencoder were implemented for this section. The diagrams for these architectures are shown in Figure 9 and 14.

Shallow Autoencoder

A shallow architecture was implemented using 2 LSTM layers, with the first layer being used to encode the input and the second layer acting as the decoder (see Figure 9). The number of hidden layer neurons for this architecture were tuned using grid search and 5-fold cross validation for 100 epochs using only the Adam optimizer. Four different hidden layer options were tested: 25, 50, 75 or 100 neurons. For this architecture, the number of neurons was found to have a significant impact on the performance of the model. In this case, having 100 neurons was found to have the best performance as it had the lowest average MSE score from grid search. This is shown in the figures below. Figure 19 is a summary of the results of grid search and Figure 20 shows the learning curves for each configuration. As discussed previously, the results for these figures were generated separately.



neurons	mean_test_score	std_test_score	mean_score_time
25	0.004975	0.001689	9.245716
50	0.004337	0.001664	10.465553
75	0.004011	0.001524	11.144907
100	0.003683	0.001349	12.575767

Figure 19: Results of using grid search to determine number of hidden layer neurons in shallow BLSTM autoencoder.

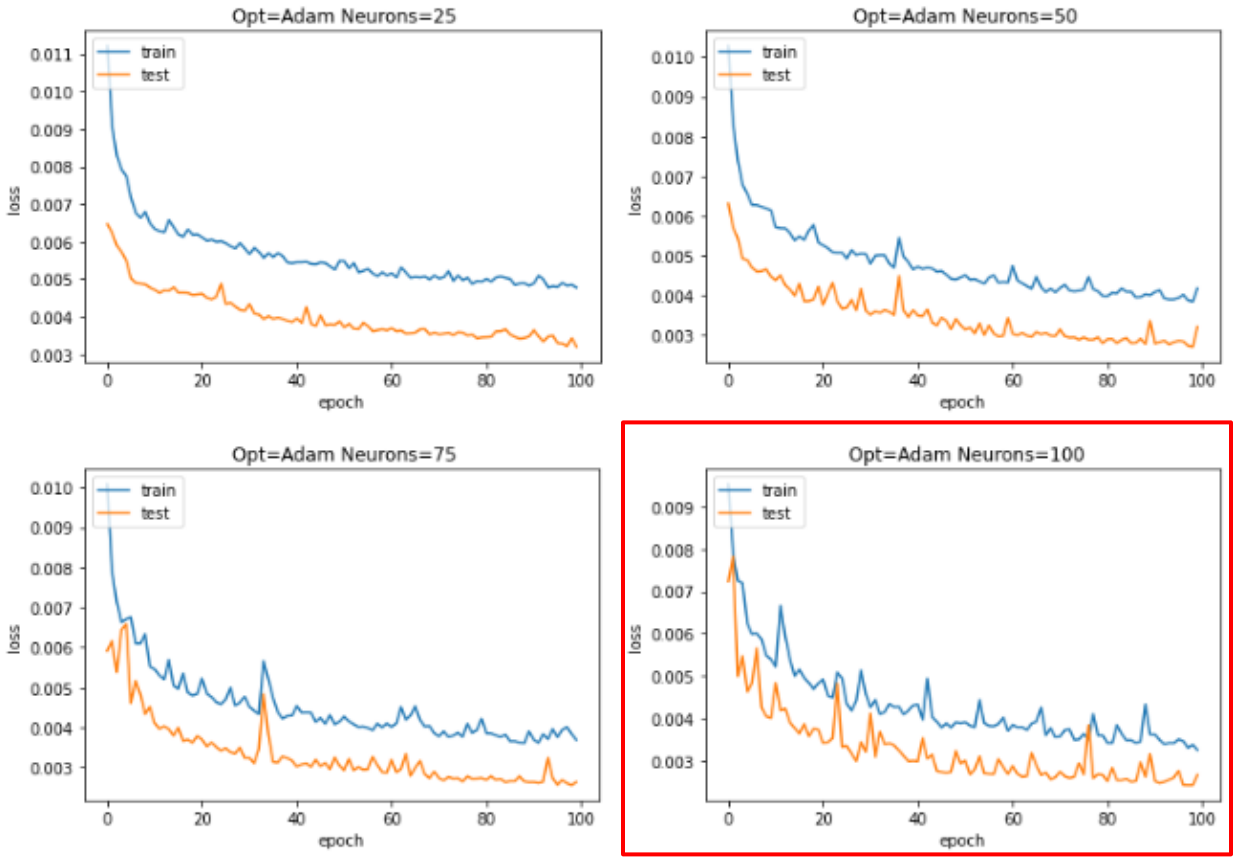


Figure 20: Learning curves after tuning number of neurons in each hidden layer for the shallow BLSTM autoencoder. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal design as identified via grid search.

Using the results above, the experiment was repeated for all three optimizers. These results are summarized in the figures below. Unlike the regular LSTM autoencoders from the previous section, the shallow BLSTM autoencoder was clearly found to have the best performance when using the Adam optimizer. The SGD optimizer performed poorly for this architecture. When looking at the SGD learning curve from Figure 21, the training loss started to flatten after around 20 epochs and did not improve much from there. This is likely due to poor hyperparameter tuning for this optimizer. While the ideal learning rate was obtained for the original architecture, the poor performance of SGD for this architecture indicates that the same learning rate should not be used here.

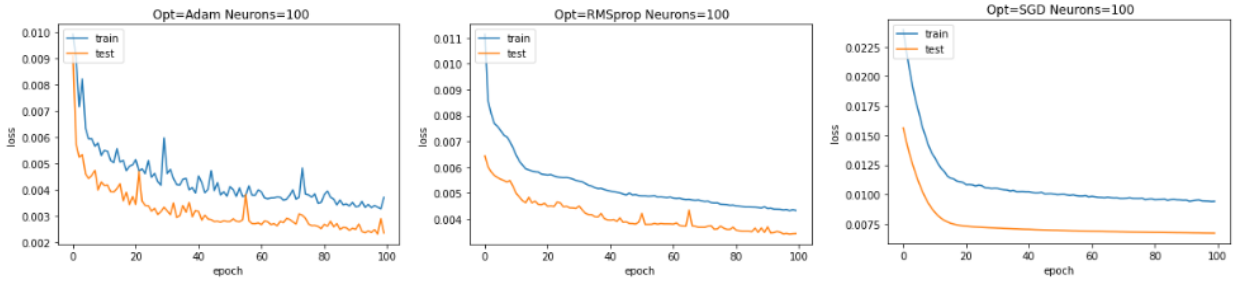


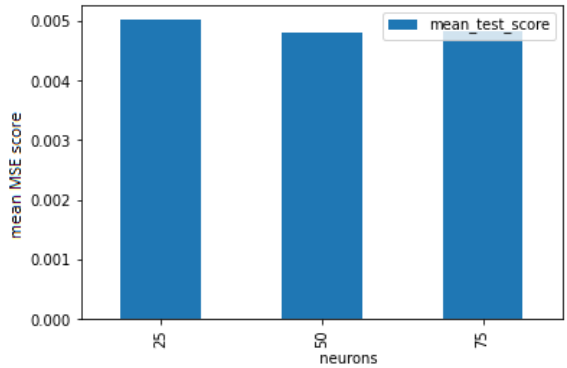
Figure 21: Learning curves showing MSE loss while training the shallow BLSTM autoencoder using the Adam, RMSProp and SGD optimizers.

Metric	Adam	RMSprop	SGD
RMSE	0.01397	0.01411	0.03204
MAPE	70.65497	95.66663	86.54612
Time (min)	44.72592	43.74166	38.67274

Figure 22: S&P 500 closing price predictions for shallow BLSTM autoencoder. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

Deep Autoencoder

The deep architecture was implemented using 4 LSTM hidden layers (see Figure 14). The number of hidden layer neurons in the middle LSTM layers were tuned using grid search and 5-fold cross validation for 100 epochs using only the Adam optimizer. Three different hidden layer options were tested: 25, 50 or75 neurons. As with the previous deep LSTM autoencoder, the number of neurons on the outer LSTM layers were kept at 100 neurons. Although this could limit the performance of the architecture, it was done to keep the number of possible variations manageable. The results of grid search, as well as the learning curves, are shown in the figures below. There was no significant difference when varying the architecture of the middle hidden layers. While the architecture with 50 neurons was found to have the lowest MSE, it was not much lower than the architecture with 75 neurons; however, it had a higher standard deviation. As a result of this, the architecture with 75 neurons in the hidden layers was selected as the optimal architecture.



neurons	mean_test_score	std_test_score	mean_score_time
25	0.005020	0.001799	6.466082
50	0.004796	0.001969	7.237960
75	0.004817	0.001828	8.273328

Figure 23: Results of using grid search to determine number of hidden layer neurons in deep BLSTM autoencoder.

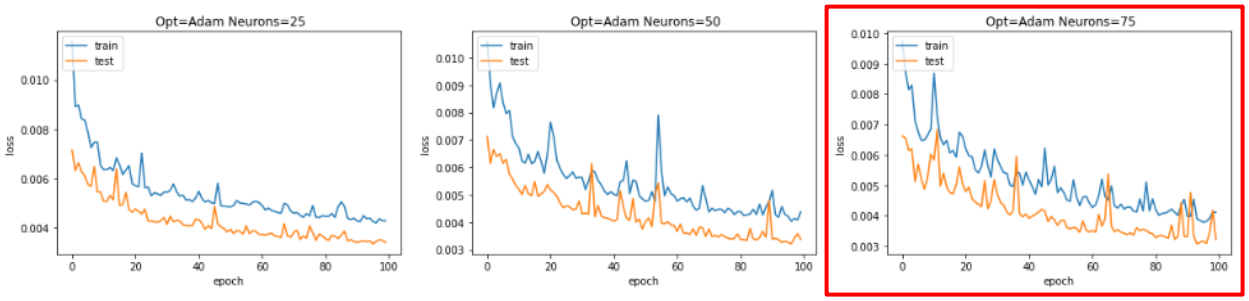


Figure 24: Learning curves after tuning number of neurons for the deep BLSTM autoencoder. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal design as identified via grid search.

From here, the three optimizers were evaluated for the deep BLSTM autoencoder architecture. As before, the Adam optimizer and the RMSProp optimizers were found to have similar performance. In this case, the RMSProp optimizer slightly outperformed Adam in terms of RMSE. However, the difference between the two scores is very small. Once again, the SGD optimizer performed badly. When looking at the S&P 500 predictions from Figure 26, the predictions generated by the model using the SGD optimizer barely follow the overall shape of the true data.

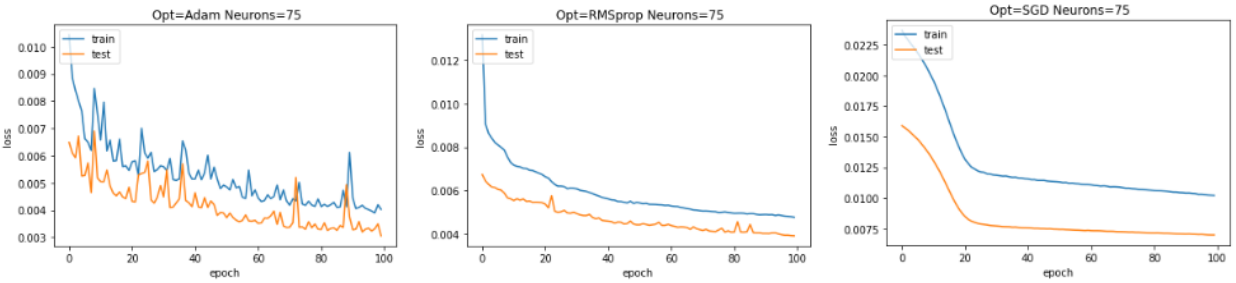


Figure 25: Learning curves showing MSE loss while training the deep BLSTM autoencoder using the Adam, RMSProp and SGD optimizers.

Metric	Adam	RMSprop	SGD
RMSE	0.01477	0.01458	0.03750
MAPE	92.44022	95.97817	99.36575
Time (min)	42.47589	42.51026	43.25428

Figure 26: S&P 500 closing price predictions for shallow BLSTM autoencoder. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

7. CNN-LSTM Hybrid Architecture

While the previous architectures only used LSTM layers, the final proposed architecture is a CNN-LSTM hybrid. CNN models tend to perform well when working with spatial and sequential data, hence their frequent use in problems involving images as well as natural language processing. For this analysis, two CNN layers will be used along with a single LSTM layer. The CNN layers will be used to extract the dominant features of the input dataset while the LSTM layer will be used to interpret the temporal behaviour of the features [9]. A diagram depicting the architecture of this neural network can be seen below.

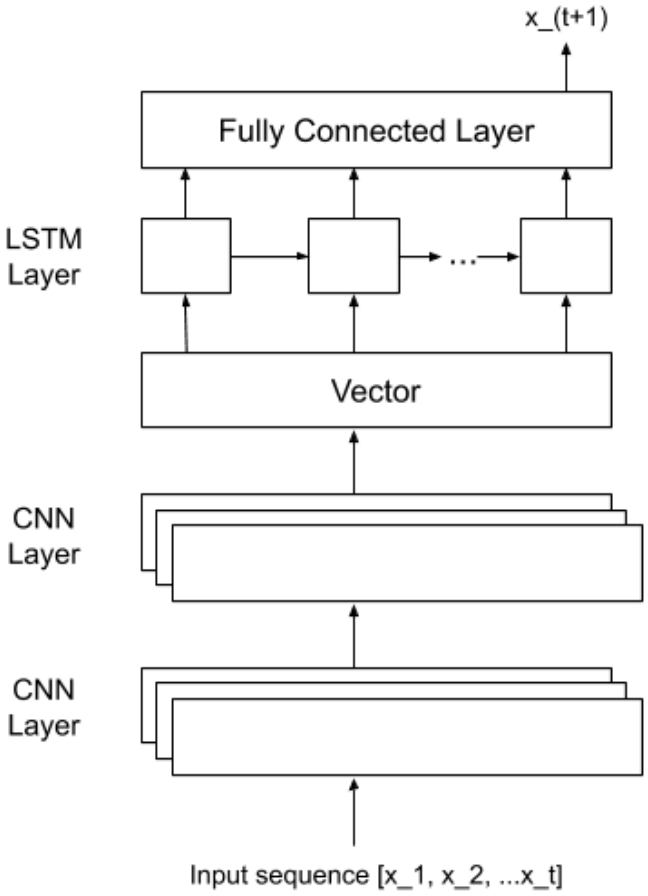


Figure 27: CNN-LSTM hybrid architecture.

For this architecture, the LSTM layer was configured to have 100 neurons, similar to the original architecture. The number of filters and kernel size for the CNN portion of the design was tuned, with both CNN layers being given the same parameter values. Grid search was used to tune these hyperparameters. Six different filter sizes were considered: 5, 10, 25, 50, 75 and 100; and four different kernel sizes were evaluated: 1, 3, 5 and 10. The results of grid search are shown in Figure 28. It was found that the networks with 5 filters performed best, regardless of the kernel size. When choosing the kernel size, a size of 1 and 3 were found to have very similar average MSE scores; however, the kernel size of 3 had a lower standard deviation and was chosen for this analysis.

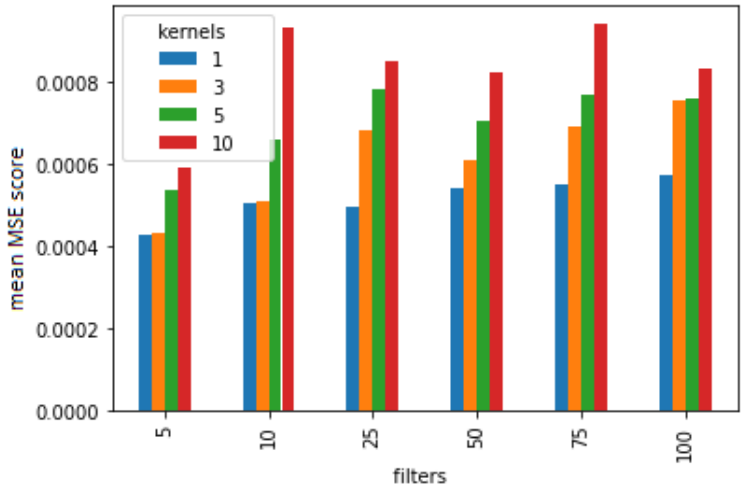


Figure 28: Results of using grid search to determine number of filters and kernel size for CNN-LSTM architecture.

As before, the learning curves for each configuration was generated. Only the learning curves for the architectures with 5 filters are shown in Figure 29.

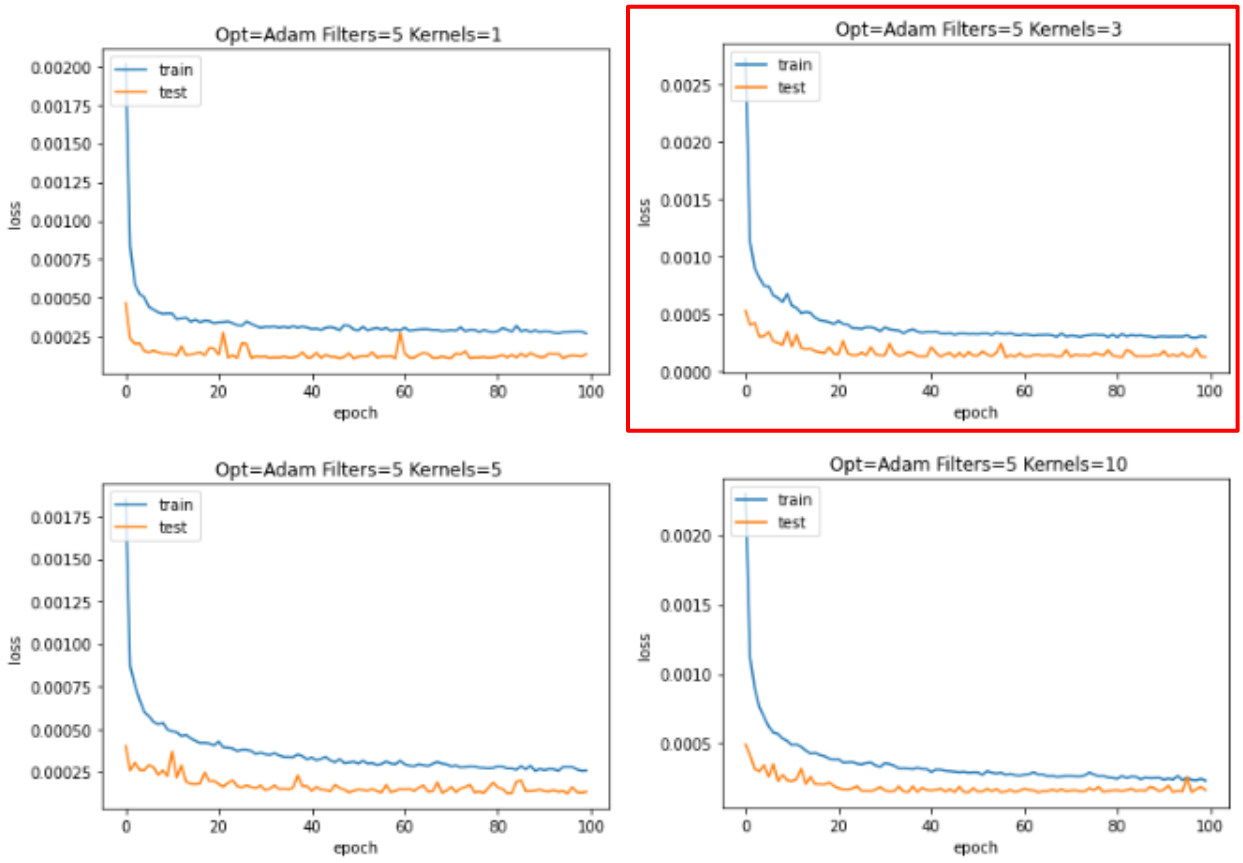


Figure 29: Learning curves after tuning number of neurons for the CNN-LSTM architecture. Each curve shows the MSE loss over 100 epochs. The red box indicates the optimal design as identified via grid search.

From here, the CNN-LSTM architecture was trained using all three optimizers. The Adam optimizer was found to outperform both the RMSProp and the SGD optimizers in terms of both RMSE and MAPE. The training time for all three optimizers was very similar.

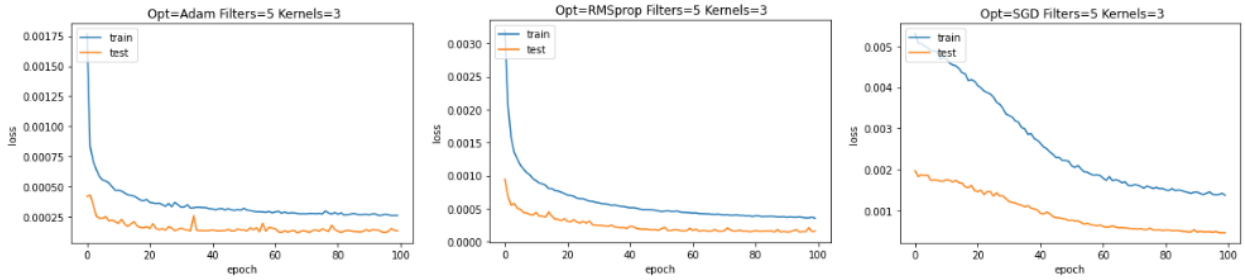


Figure 30: Learning curves showing MSE loss while training the deep BLSTM autoencoder using the Adam, RMSProp and SGD optimizers.

Metric	Adam	RMSprop	SGD
RMSE	0.01147	0.01258	0.02151
MAPE	66.16389	68.82497	138.37486
Time (min)	2.78845	2.78762	2.65141

Figure 31: S&P 500 closing price predictions for CNN-LSTM architecture. The values for the performance metrics are shown in the table on the left with the lowest values for each metric highlighted in green.

8. Comparison of Architectures

The figures below present a summary of the performance metrics for each of the architectures that were examined in this analysis. Only the results for the Adam optimizer are shown since this optimizer was found to have the best performance for all architectures.

For both RMSE and MAPE (Figures 32 and 33), the original architecture outperformed all the proposed designs by a significant margin. All three of the proposed designs aimed to improve on the performance of the original architecture by introducing some type of feature extraction. The purpose of this was to focus on the salient features of the dataset when making predictions. The S&P 500 dataset used in this analysis consisted of five features and it seems that emphasizing some of these features did not improve the predictive performance of the model.

Of the proposed architectures, the CNN-LSTM model performed the best across all three metrics. The CNN layers of this architecture captured the short-term spatial relationships in the data while the LSTM layer captured the long-term temporal patterns. By considering both short-term and long-term patterns, the CNN-LSTM architecture had better performance than the LSTM autoencoders which focussed on longer term relationships.

The CNN-LSTM model also had the shortest training time, beating the other proposed architectures as well as the original model (Figure 33). This result was not surprising given that the CNN-LSTM architecture only had one LSTM layer while the other designs had at least two. LSTM layers, as with all recurrent network layers, are inherently slow to train since each cell must be processed sequentially. In contrast, CNN layers have no such limitation and can be processed in parallel.

Finally, when comparing the LSTM autoencoders, there was no significant difference between the shallow and deep architectures in terms of RMSE. Based on this it can be concluded that deeper networks will not result in better predictions for this dataset. The use of bidirectional LSTM layers was also not found to have an impact on the predictive performance of the model.

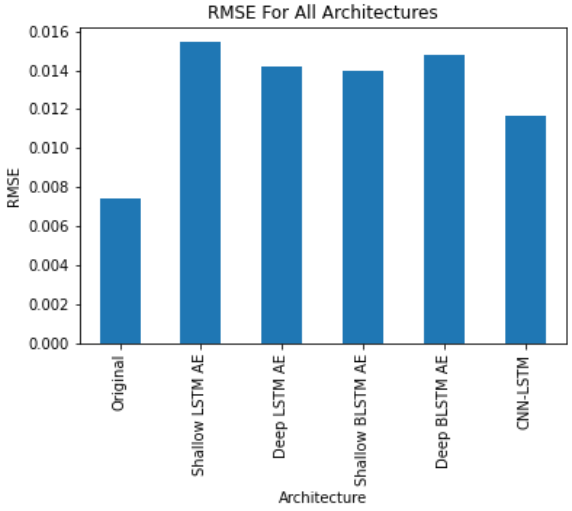


Figure 32: RMSE values for all architectures (Adam optimizer only).

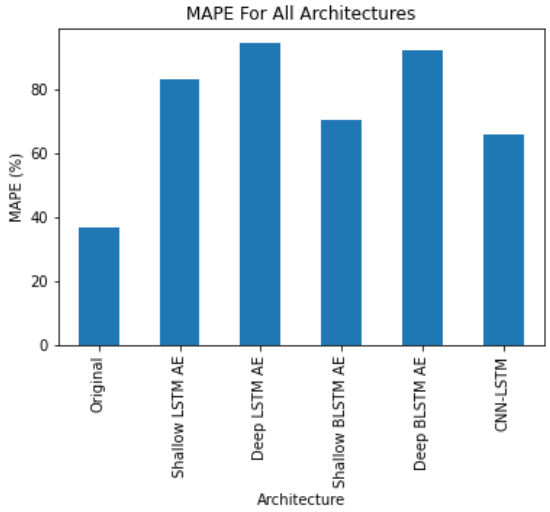


Figure 33: MAPE values for all architectures (Adam optimizer only).

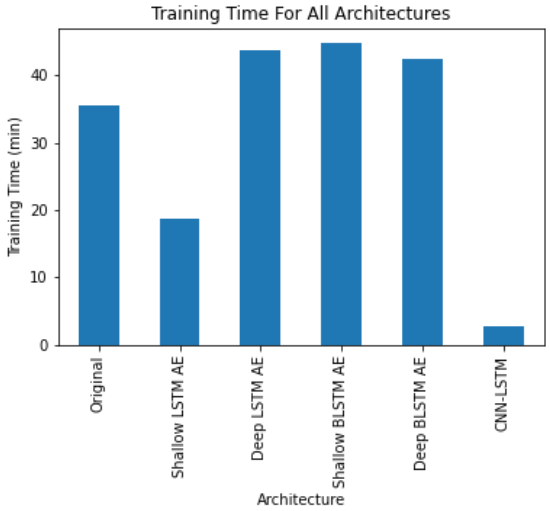


Figure 34: Training time (minutes) for all architectures (Adam optimizer only).

9. Conclusions and Future Work

From the results of this analysis, it is clear that the original architecture outperformed the proposed architectures. The proposed architectures primarily aimed to improve the predictive performance of the model by extracting the salient features of the dataset. However, for this dataset, the dimensionality reduction provided by the proposed models did not improve the overall predictions.

The SGD optimizer performed poorly for all the architectures; however, this was unsurprising. As discussed in a previous section, this optimizer takes a very basic approach to gradient descent and tends to get stuck in local minima, especially if it is not initialized well. Both the Adam and RMSProp optimizers performed well for all the architectures, although the Adam optimizer performed slightly better. This was expected since the Adam optimizer builds on the good properties of RMSProp by including bias-correction.

One limitation of this analysis was that the hyperparameters for the optimizers were not tuned for each architecture. Instead, the learning rates that were found to be ideal for the original architecture were reused. While this was done to facilitate comparison between the different networks, it also prevented the proposed architectures from reaching their full potential. This was particularly true for the SGD optimizer. The SGD optimizer had the worst performance for all the architectures and part of this was because its performance is heavily dependent on how it is initialized.

Furthermore, this analysis took a very simplistic approach and generally only tuned one parameter for each of the proposed architectures. In reality, there are a virtually infinite number of variations that could have been tested for the hyperparameters that were considered in this analysis, as well as others. For example, while the learning rate is the main hyperparameter used with optimizers, each optimizer has additional parameters that can be tuned such as the initial weights or momentum values. Also, activation functions are another hyperparameter that can have a significant impact on the performance of a neural network. Activation functions were not explored in this analysis, and the default linear activation was used for most of the neural network layers.

10. References

[1] Aungiers, Jakob. “Time Series Prediction Using LSTM Deep Neural Networks.” *Articles & Research*, Altum Intelligence, 1 Sept. 2018, www.altumintelligence.com/articles/a/Time-Series-Prediction-Using-LSTM-Deep-Neural-Networks.

[2] Aungiers, Jakob. “README.md.” *LSTM-Neural-Network-for-Time-Series-Prediction*, Github, 30 Apr. 2019, github.com/jaungiers/LSTM-Neural-Network-for-Time-Series-Prediction.

[3] Ruder, Sebastian. “An Overview of Gradient Descent Optimization Algorithms.” *Sebastian Ruder*, 19 Jan. 2016, ruder.io/optimizing-gradient-descent/.

[4] API design for machine learning software: experiences from the scikit-learn project, Buitinck *et al.*, 2013.

[5] Drakos, Georgios. “How to Select the Right Evaluation Metric for Machine Learning Models: Part 1 Regression Metrics.” *Medium*, Medium, 26 Aug. 2018, medium.com/@george.drakos62/how-to-select-the-right-evaluation-metric-for-machine-learning-models-part-1-regrression-metrics-3606e25beae0.

[6] “Mean Absolute Percentage Error (MAPE) .” *Statistics How To*, 8 Sept. 2017, www.statisticshowto.com/mean-absolute-percentage-error-mape/.

[7] Tofallis, Chris. “A Better Measure of Relative Prediction Accuracy for Model Selection and Model Estimation.” *Journal of the Operational Research Society*, vol. 66, no. 8, 2015, pp. 1352–1362., doi:10.1057/jors.2014.103.

[8] Brownlee, Jason. “How to Develop LSTM Models for Time Series Forecasting.” *Machine Learning Mastery*, 14 Nov. 2018, machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/.

[9] Kim, Tae-Young, and Sung-Bae Cho. “Predicting Residential Energy Consumption Using CNN-LSTM Neural Networks.” *Energy*, vol. 182, 2019, pp. 72–81., doi:10.1016/j.energy.2019.05.230.