# Predicting Option Pricing Using Encoder-Only Transformers

**Rhythm Satav**
Carnegie Mellon University
Pittsburgh, PA 15213
rsatav@andrew.cmu.edu

**Alex Lyons**
Carnegie Mellon University
Pittsburgh, PA 15213
alyons@andrew.cmu.edu

## Abstract

The Black-Scholes Equation is a differential equation that represents a call option pricing model. It computes call option prices by taking in the strike price, stock price, time maturity, interest rate, and volatility. In practice, implied volatility is used since volatility can't be directly observed, rather, can only be derived inversely by observing option prices from the market. Previous work uses feedforward neural networks to predict call option pricing, and various architectures to compute the implied volatility needed as input. Our work uses an encoder-only transformer to compute the implied volatility given previous strike prices, stock prices, time maturities, and interest rates in the form of a time series transformed into embeddings, and we input the volatility into a forward neural network along with the other Black-Scholes inputs to predict call option pricing. Using the transformer in addition to the feedforward model yielded performance improvement, providing evidence that transformers are useful for computing useful implied volatilities.

## 1 Introduction

A call option is a financial contract that gives a holder the right, not obligation, to buy an asset at a specified strike price. A common call option pricing model is the Black-Scholes model that calculates the theoretical price of options represented by the following formula:

$$C = S_0 N(d_1) - X e^{-rT} N(d_2), \tag{1}$$

where:

$$d_1 = \frac{\ln\left(\frac{S_0}{X}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}},$$
$$d_2 = d_1 - \sigma\sqrt{T},$$

and

- $C$: Call option price,
- $S_0$: Current price of the underlying asset,
- $X$: Strike price of the option,
- $r$: Risk-free interest rate,
- $T$: Time to expiration (also known as time maturity),
- $\sigma$: Volatility of the asset's returns,
- $N(\cdot)$: Cumulative distribution function of the standard normal distribution.

One useful piece of information that financial analysts are interested in is how a call option changes over time according to the Black-Scholes equation. Specifically, since the volatility changes over time, researchers are interested in how the change in volatility over time can affect a contract's call option price. Financial researchers have attributed time maturity and strike price as the main factors that affect volatility [4][7]. Therefore, many option forecasting models rely on creating volatility forecasting models (i.e. predicting the volatility given historical data), and then use the predicted volatility to compute the predicted option call price.

Therefore, we aimed to experiment with methods that either forecasted the call option price directly, or forecast the volatility and use that to compute the call option price.

## 2 Background

For our baseline, we implemented a standard deep neural network in order to directly predict the call option price. Specifically, we inputted the time maturity, interest rate, strike price, implied volatility, and stock price into a deep neural network, used the call option price as the target, and then use mean squared error loss in order to optimize the model. We performed this on two datasets: A synthetic dataset generated by directly computing call option prices using the Black-Scholes model, and a Yahoo finance dataset scraped from the web.

Both models showed lowering and converging train and validation losses, meaning they both were correctly training on the data. However, while the synthetic dataset was simple enough to yield a low final loss for the trained model, the Yahoo finance dataset model yielded a final loss of around 3000, which was very high.

Therefore, for the final project we had two goals. The first was to collect a new dataset that was large enough for a model to train on, but not artificially simple like the synthetic dataset. The second goal was to design and train a model to achieve a low loss on that dataset.

## 3 Related Work

Different model architectures that require different assumptions about the data have been used in the past. For example, neural networks have been used to directly compute the option pricing [1]. Specifically, the time maturity and a calculated price are inputted into a neural network that computes the option pricing. Consequently, the model doesn't take into account any changes over time.

Additionally, past models that experiment with volatilities utilize historical time-series as their inputs [7]. In order to incorporate the change in parameters of the option pricing over time, several series models have been tested. Both recurrent neural networks (RNN's) and long-short term memory networks (LSTM's) have been experimented with by accumulating data over many days [2]. Specifically, they use the LSTM to predict volatility, and input the volatility along with stock price, strike price, and time maturity into an MLP to predict the option pricing. These models yielded improved performance, but still suffer from several problems. They lose information seen at the beginning of the sequence over time, inhibiting the models' ability to use information across the full scope of the time series. Additionally, they input each point in the series sequentially, which loses explicit information on the different amounts of time that could have passed in between each data point.

Recently, transformers have been used to better capture data across an entire sequence [3]. They utilize attention, which allows the model to find connections across all data points in the sequence, no matter how far apart they are in their ordering. Consequently, these have started to be utilized to predict option pricing [4]. Following the financial research that shows volatility is dependent on time maturity and strike price, they convert their input data into a surface where the x and y coordinates are the time maturity and strike price, and the z coordinates are the volatilities[7]. Then, they interpolate it into a 2D matrix, and input this into a convolutional transformer to predict volatility. Then, they input the volatility along with the stock price, strike price, and time maturity directly into the original Black Scholes equation to predict the option pricing. However, the model's positional encoding still doesn't explicitly take into account the case where data points in the series are different amounts of time apart, and additionally we believe the overall architecture can be simplified from 2D. Additionally, their loss function requires using implied volatilities, which is data our final model won't use.

## 4 Methods

### 4.1 Data Curation

In order to sufficiently train a large enough model, we required a larger dataset than our previous Yahoo dataset, while at the same time representing realistic data, unlike the former synthetic dataset. Therefore, we instead used official data purchased from Cboe, a financial data company that curates and sells large amounts of historical financial data. We purchased 3 weeks of options data from the S&P 500 from October 2024. Notably, this contained the stock price, strike price, time maturity, and end of day bid and ask prices for 388393 contracts, with the interest rate assumed to be constant among all of them. The dataset didn't contain the call option price or implied volatility. However, we estimated the call option price by averaging the bid and ask price as was done in previous work [2]. The lack of implied volatility data meant we had to predict the call option price directly, instead of being able to predict the volatility and then plugging the result into the Black-Scholes model.

Next, in order to format the data as a forecasting problem we created series of contracts to use as training data points. To do so, we grouped together contracts that had the same time maturity and strike price, according to the financial research stating that those two affect volatility the most [7]. Then, we ordered each contract within the series according to the contract date. Contracts with a specific time maturity and strike price weren't available on all days, so series had different lengths according to how many days a contract with that maturity and strike price was available.

In total, our dataset contained 22608 separate series. Since each series could have lengths from 1 to 16, we compared the counts of each series length, as shown in Figure 1. The majority of series have length only 1, meaning they were single data points.

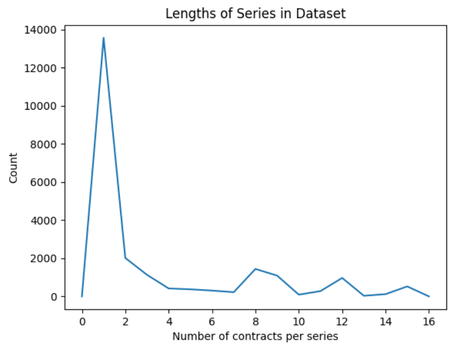

Figure 1: The number of series in the dataset with each possible length (maximum 16 days).

### 4.2 Core Model Architecture

A summary of our core model architecture is shown in Figure 2.

We start of by pre-processing our time-series data by converting the raw data into embeddings via an EmbeddingMLP. This was accomplished using a feed-forward neural network with 2 connected layers and a ReLU activation function to introduce non-linearity. Notably, we also included what position in the series each set of inputs when inputting them into the EmbeddingMLP. The point of this is to act as an implicit positional encoding, since positional encodings have been shown to improve transformer performance.

Moving forward, we implemented an encoder-only transformer class that learned the volatility from the previously computed embeddings. For predicting the volatility at time step $T$ in the series, the embeddings from time steps $1..T-1$ are then passed into the transformer encoder which contains a self-attention mechanism that can identify dependencies between timestamps, aiding in the eventual volatility prediction. We decided to use a relatively small Encoder-Only Transformer with 3 layers and 2 heads, given that there is little input data and the data isn't very complex. The final volatility was extracted from the final element in the encoder output. We did this since in theory the new volatility should depend mostly on the most recent data compared to earlier data in the time series, and therefore getting the output from the final element in the output sequence would bias the volatility towards recent data in the series. Note that since the time series had different lengths, we used added zeros to make them the same length and used masking within the transformer to adjust accordingly.

Finally, the volatility is inputted along with the other inputs from time step $T$ (i.e. the time maturity, stock price, strike price, and interest rate) into the PriceMLP, which is a 4-layer feed-forward neural network with ReLU activations. The output of this network is the predicted call options pricing for time step $T$. This was compared against the ground truth call options pricing (which was estimated using the bid and ask prices), and the error was backpropagated via mean squared error. The idea behind this model is to replace the traditional Black-Scholes equation, given that it has the same inputs and output. Also, note that architecturally, the PriceMLP is the same as the baseline, except the baseline doesn't use any volatilities since they weren't in the dataset. By doing so, in the discussion and analysis we were able to show that including the Encoder-Only Transformer improved performance compared to just using the PriceMLP, signifying that using historical data (i.e. the time series) was useful.

Thus, in summary the full architecture for predicting the call options price at time step $T$ consists of using the time series from time steps $1...T-1$ to create embeddings via the EmbeddingMLP, feeding the embeddings into the Encoder-Only Transformer to compute the volatility, and then inputting volatility along with the other inputs at time step $T$ being into the PriceMLP to predict the call options price.

Additionally, during training we experimented with different hyperparameters to maximize full architecture performance. Through testing different learning rates, we found the model losses initially don't greatly change, and then suddenly drop and continue to greatly fluctuate at the new low. Therefore, we used a learning rate scheduler to use a higher learning rate to reach the drop in loss quicker, and then lower the learning rate after the drop to provide more stable optimization.

## 4.3 Auxiliary Model Architectures

One architecture change we experimented with was replacing the PriceMLP with the Black-Scholes model directly, as was done in the convolutional transformer [4]. Specifically, the output of the Encoder-Only Transformer was fed directly into the pre-defined Black-Scholes equation (along with the other corresponding inputs) to compute the predicted call options pricing. The idea behind this is that the error would backpropagate from the loss through the Black-Scholes equation, and update the other submodels accordingly. However, as will be explained in more detail in the Discussion and Analysis section, the complexity of the Black-Scholes equation likely made the gradients vanish, which prevented model training.

Another architecture change we experimented with was adding an additional positional encoding in addition to the implicit one from the EmbeddingMLP. Before passing in the embeddings into the transformer encoder, the positional encoding is applied to them as well as a scaling factor of the square root of the embedding size. This scaling factor ensures that the positional encoding is scaled to the same magnitude as the input embeddings, making sure that the positional encoding still has reasonable influence alongside the original input embeddings. The idea behind doing this was to test how well the EmbeddingMLP was able to incorporate positional information into the embeddings. As will be explained later, the additional positional embedding didn't greatly change performance.

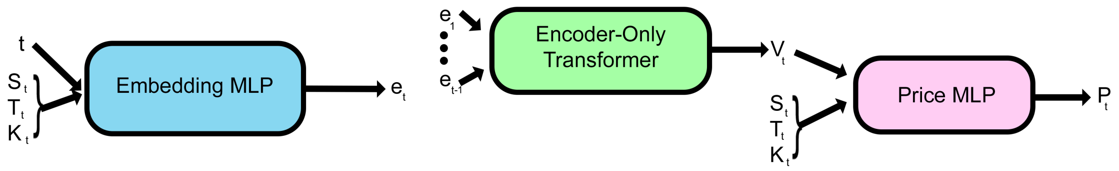

Figure 2: Diagram showing the full architecture of our core model.

Table 1: Experimental Results

| Model | Final Training MSE | Testing Final MSE |
|---|---|---|
| Baseline | 1669.165 | 1786.958 |
| Encoder-Only Architecture | 4894.655 | 4985.956 |
| Full Architecture | **44.249** | **52.075** |
| Full Architecture w/ Pos. Enc. | 50.907 | 58.274 |

## 5 Results

### 5.1 Training and Evaluation Methods

To evaluate our models, we split our dataset into training, validation, and testing data with a 70%, 10%, 20% split. As is standard, we didn't tune or train our models using any of the testing data. Each model was trained for 500 epochs, unless the loss became constant in which case we performed early stopping. We used mean squared error (MSE) to train and analyze each model, given that it provided smooth training and makes it easier to compare to other models that used the same error metric.

Table 1 shows a summary of the results across the different models we experimented with. We will go into depth about the results of each model grouping them into two categories: Models that were essential for the core ideas behind this paper, and models for testing auxiliary experiments. Then, we will compare our results to prior work.

### 5.2 Baseline and Core Models

Since the baseline from the proposal report was trained on the Yahoo dataset, it needed to be retrained on the new dataset. After the baseline model was retrained on the new dataset, the final training and testing losses were 1669.165 and 1786.958 respectively. Figure 3 shows the training and validation loss of the baseline model after it was retrained on the new dataset. Both losses quickly decreased in the early epochs, and then quickly converged to their final losses. As expected, the curves look similar to those found in the proposal report using the Yahoo dataset given that the data in the new dataset should be similar to that of the Yahoo dataset, except larger.

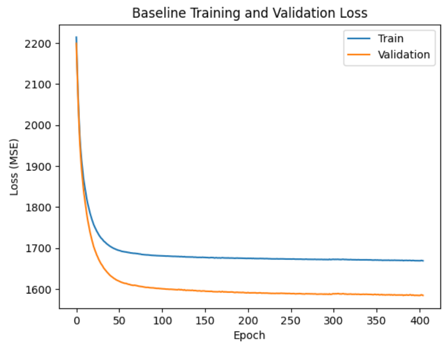

Figure 3: The training and validation losses from the baseline model, which is only the PriceMLP.

The encoder-only architecture is the model where the output from the encoder-only transformer was directly used to compute the call option pricing. After training, the final training and testing losses were 4894.655 and 4985.956 respectively. The training and validation losses over time, as shown in Figure 4, show a similar behavior to the baseline. Specifically, they drop quickly initially, and then level out to the final losses. According to the final losses, this model performed worse than the baseline, which isn't in align with expectations. Since more data was used as input (i.e. the whole series) to the encoder-only architecture, it was expected that it would perform better.

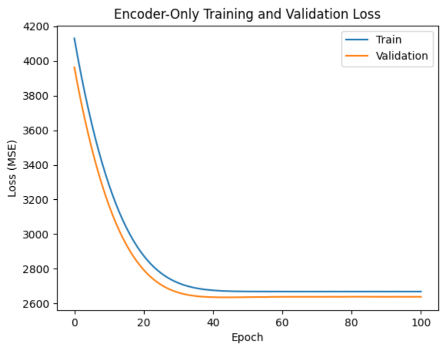

Figure 4: The training and validation losses from using only the Encoder-Only Transformer to predict the call options price.

The full architecture as shown in Figure 2 is essentially the output of the encoder-only architecture fed into the baseline model, which was expected to merge the benefits of both models. The benefit of the encoder-only architecture is that it uses data over several days, and the benefit of the baseline is that it places emphasis on the parameters (strike price, maturity, etc) most important to predicting the desired call option pricing. Accordingly, the full architecture model achieved the lowest training and testing losses of 44.249 and 52.075 respectively. Figure 5 show the training and validation loss of our final optimized model. Unlike the two previously described curves, more complex behavior is shown. First, the losses slowly decrease until approximately epoch 200. Then, they rapidly drop until epoch 250, where the first iteration of learning rate lowering occurs to smooth results. Then it slowly decreases until epoch 350, where the learning rate lowers more to yield convergence on the best losses. As described, the losses were expected to be lower than the two previous models, and this expectation was met.

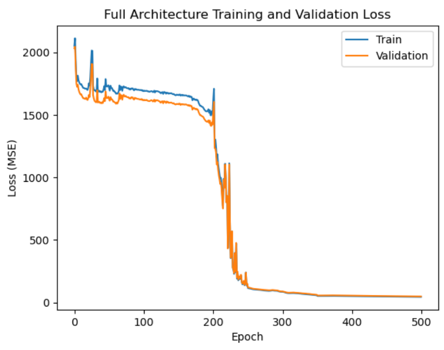

Figure 5: The training and validation losses from the full architecture model, i.e. the combination of the Encoder-Only Transformer and PriceMLP.

### 5.3 Auxiliary Models

One aspect of our model that we analyzed was if adding Pytorch's positional encoding to the embeddings would improve performance. Our expectation was that performance wouldn't greatly vary, because as described in the methods one of the purposes of the EmbeddingMLP was to act as a positional encoder. In align with expectations, the final training and testing losses of 50.907 and 58.274 were very similar to those from the Full Architecture model. Additionally, as shown in Figure 6, the behavior of the training and validation losses was also very similar.

Another aspect of our model that we attempted to test was if the model would perform better if we used the predicted volatilities to compute the call option pricing directly using the Black-Scholes equation, as was done in the convolutional transformer model [4]. However, the losses never changed,

Table 2: Cross-Dataset Results

| Model | Final Training MSE | Testing Final MSE |
|---|---|---|
| Full Architecture | 44.249 | 52.075 |
| Stanford LSTM [2] | 30.61 | 30.97 |
| Stanford MLP [2] | 7.70 | 15.21 |

signifying the model wasn't training properly. This went against expectations, since the convolutional transformer model was successfully trained in previous research.

### 5.4 Comparison to Prior Work

Table 2 shows a cross-dataset comparison between our best model and two models from Stanford that also used mean square error for their analysis [2]. By cross-dataset comparison, it means that the losses computed for their models was taken from a different dataset than ours, given that we don't have access to their data. In terms of the final losses, both of their models achieved a lower training and testing loss. There are key differences between the two datasets that could explain our model's under-performance. First, their training dataset included volatilities, unlike ours, which would likely make it easier to predict future volatilities. Second, as described in their paper each of their series fully contained 20 days worth of data, whereas in our data, as shown earlier, a majority series contain only 1 day worth of data. Both of these qualities would provide useful information for predicting future volatilities, which our dataset wouldn't allow our model to do with such detail.

## 6 Discussion and Analysis

### 6.1 Baseline and Core Models

According to the results, the encoder-only architecture performed the worst, the baseline performed better, but the combination of the two into the final full architecture performed the best.

One of the key assumptions behind our model design was that it should be similar to the Black-Scholes equation, given that is has been historically used to compute the call options pricing. Based on the fact that the baseline performed better than the encoder-only architecture, it signifies that the most important part of the full architecture is the PriceMLP. This also aligns with previous work done in the Stanford models, in that the losses for the MLP only model were lower than the LSTM losses as shown in Table 2 [2]. Thus, this provides evidence for our assumption that the Black-Scholes model is important, since these MLP's model the Black-Scholes model by using the same inputs and output.

Another one of the assumptions behind our model design was that the PriceMLP could act as a substitute for the Black-Scholes equation when predicting the call options pricing. However, due to limitations with our dataset and model, we were unable to directly compare the PriceMLP performance to the Black-Scholes equation. One limitation is that our dataset doesn't contain volatilities, meaning we wouldn't be able to plug in the values required for the Black-Scholes equation into the PriceMLP to compare performance. One alternative solution would be to plug in the volatilities computed from the Encoder-Only transformer, and plug these values into both the PriceMLP and Black-Scholes equation and compare the results. However, one limitation of our model that prevents this is that when we tried to put constraints on the computed volatility, the model training failed. Specifically, the Black-Scholes equation requires that the volatility be positive, and typically between 0 and 1 [8]. When we tried to use Sigmoid activation to apply the constraint, the losses stopped lowering, signifying a vanishing gradient problem. When we tried to use ReLU, the model trained and didn't affect performance, but there were many volatilities that were either 0 or much greater than 1, which both simulate unrealistic conditions that the Black-Scholes model would assume. Thus, we weren't able to directly compare the PriceMLP results against the Black-Scholes model results because there weren't suitable volatilities available. To overcome this for future research, one would likely either pick a new dataset with volatilities included, or add new constraints in model training to enforce the computed volatilities be between 0 and 1.

Another key assumptions behind our model was that using historical data to calculate the volatility (via the Encoder-Only transformer) would improve the PriceMLP performance. The fact that performance of the baseline (which is architecturally equivalent to the PriceMLP) was worse than the performance of the full architecture model provides evidence that this assumption was true. Note that this portrays the opposite relationship to that found in the Stanford results, where the more complicated LSTM performed worse than the simpler MLP [2]. This suggests that the architecture for computing the volatility is important, and specifically that an Encoder-Only transformer actually improves PriceMLP performance instead of det. Therefore, future research should explore more architectures for computing the volatility, since that is the bottleneck when trying to use historical data to predict future volatility.

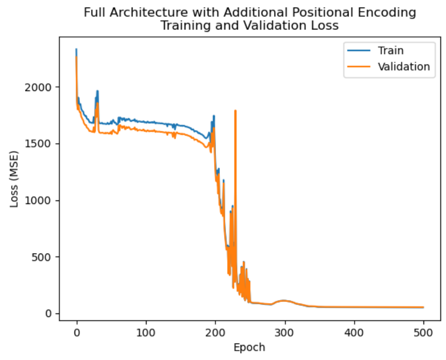

Figure 6: Full architecture training and validation losses when adding the explicit positional encoding to the Encoder-Only Transformer embeddings.

### 6.2 Auxiliary Models

One notable point of discussion is why our model failed when trying to compute the call option pricing by using the Black-Scholes model directly, while the convolutional transformer model was able to do so. Upon analysis, it was found that the gradients from the loss to the input volatilities vanished when backpropagating through the Black-Scholes equation. This was discovered by systematically removing functions within the Black-Scholes equation, and discovering that the losses started to decrease when the complexity of the function was sufficiently decreased, especially after removing the standard cumulative distribution functions. Furthermore, one key difference that likely allowed the convolutional transformer to overcome this issue was that their model had an additional L1 loss comparing the actual and predicted volatilities, which didn't require the Black-Scholes equation. Given that their dataset included volatilities, and that there isn't a closed-form solution for the volatilities given the other parameters of the Black-Scholes equation, our model wouldn't be able to train using the same additional L1 loss. Therefore, this difference provides a possible explanation for why our model failed and theirs successfully trained, and to improve our model, we would need data that includes volatilities.

Another notable point of discussion is that the full architecture model performed approximately the same as the full architecture with an additional positional encoding. One of the key assumptions behind our model's architecture was that by passing in the day as a value in the EmbeddingMLP before inputting the embedding into the transformer, it would act as an implicit positional encoding. The fact that performance stays the same after adding an additional positional encoding implies that this assumption is true since adding a redundant positional encoding would lead to no significant performance change. This suggests that in order to improve performance via an additional positional encoding, the encoding method would have to be more complex than a simple MLP can compute, or else it would be made redundant by the EmbeddingMLP. Therefore, a route of future research would be to experiment with adding more complex positional encodings.

## References

[1] Santos, D. D. S., & Ferreira, T. A. E. (2024). Neural Network Learning of Black-Scholes Equation for Option Pricing. arXiv preprint arXiv:2405.05780.

[2] Ke, A., & Yang, A. (n.d.). Option Pricing with Deep Learning. Stanford. https://cs230.stanford.edu/projects_fall_2019/reports/26260984.pdf

[3] Vaswani, A. (2017). Attention is all you need. Advances in Neural Information Processing Systems.

[4] Kim, S., Yun, S. B., Bae, H. O., Lee, M., & Hong, Y. (2024). Physics-informed convolutional transformer for predicting volatility surface. Quantitative Finance, 24(2), 203-220.

[5] Shannonn, C. (2018). Black-Scholes NAS [Software]. GitHub. https://github.com/cshannonn/blackscholes_nas/tree/master

[6] Lisle, D. (2024). Black Scholes Options Pricing [Software]. GitHub. https://github.com/dreamchef/Black-Scholes-options-pricing

[7] Medvedev, N., & Wang, Z. (2022). Multistep forecast of the implied volatility surface using deep learning. Journal of Futures Markets, 42(4), 645-667.

[8] Can Volatility Be Greater than 1? - Macroption. (2024). Macroption.com. https://www.macroption.com/can-volatility-be-greater-than-1/