STAT5293: Pricing Derivatives with Deep Learning Final Report

Muqing Yang*[1], Bingjiang Xia†[1], and Jianfeng Chen‡[1]

Statistics Department, Columbia University

Abstract

In the fast-paced domain of computational finance, the valuation of derivatives demands models that are both precise and computationally efficient. This report presents an in-depth comparative analysis of several deep learning architectures applied to the pricing of basket options under uniform setting. This report uses Monte Carlo simulations for basket options dataset creation. We systematically investigate the accuracy, execution speed, and cost-efficiency of Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), Transformer models, Long Short-Term Memory networks (LSTM), and combined CNN-LSTM models, to determine their efficacy in predicting derivatives pricing. Our findings demonstrate a nuanced landscape where the CNN-LSTM architecture excels in accuracy due to its dual ability to process spatial and temporal information. CNNs follow closely, offering a harmonious balance of accuracy and simplicity, while LSTMs provide a reliable option with consistent, albeit varied, performance. In contrast, DNNs and Transformer models, while powerful, face challenges in slower convergence and higher susceptibility to overfitting as complexity increases. The report underscores the nuanced trade-offs between model complexity, resource allocation, and performance, guiding the selection of optimal computational strategies in the fast-paced and intricate world of financial trading.

Keywords: Deep learning, Option pricing, Transformer, CNN-LSTM, Monte Carlo Simulation.

*Email: my2842@columbia.edu
†Email: bx2218@columbia.edu
‡Email: jc6175@columbia.edu

1. Introduction

In computational finance, numerical methods are commonly used for the valuation of financial derivatives and also in modern risk management. Generally speaking, advanced financial asset models are able to capture nonlinear features that are observed in the financial markets [16]. Computational performance is of significance in real-world market derivatives transactions forecasting. In the realm of derivatives trading, the accuracy, speed, and cost-effectiveness of predictive models bear paramount significance. The ramifications of these factors extend beyond mere academic curiosity, as they directly influence the quality, timeliness, and economic viability of investment decisions. Consequently, a rigorous investigation into the comparative performance of various models in processing option transactions constitutes the central objective of this research endeavor. By systematically evaluating the efficacy of different computational approaches, this project seeks to contribute to the advancement of knowledge in the field of quantitative finance and provide actionable insights for market participants. The findings of this study have the potential to inform the development of more efficient and effective strategies for managing risk and optimizing returns in the complex and dynamic world of derivatives trading.

In this report, we embark upon a comprehensive evaluation of the performance characteristics of a diverse array of computational models, including Convolutional Neural Networks (CNN), Deep Neural Networks (DNN), Transformer architectures, Long Short-Term Memory (LSTM) networks, and CNN-LSTM, within the context of predicting options and futures under uniform setting. The primary objective of this project is to conduct a rigorous assessment of the speed, accuracy, and cost-effectiveness of these models in generating reliable forecasts for derivatives trading.

2. Model Description

2.1 Artificial Neural Network (ANN)

An ANN is a model of interconnected nodes, or neurons, structured in a way that mimics the human brain's information-processing capability. It is typically organized into multiple tiers, known as "layers," that include the input layer, one or more hidden layers, and an output layer [4][14].

The application of ANN in option pricing has a considerable history. As early as 1993, Hutchinson et al. proposed the idea of using ANNs to price and hedge derivative securities [10]. They found that, for standard call options, the trained ANN was able to achieve pricing accuracy comparable to the Black-Scholes model on out-of-sample data, and the ANN model showed advantages when dealing with more complex derivatives. Since then, many researchers have begun to explore the application of ANNs in option pricing. Yao et al. [21] used ANNs to predict the prices of Hang Seng Index options in Hong Kong and compared their performance with the traditional Black-Scholes model. The results showed that the ANN model could provide more accurate pricing results, especially for deep in-the-money/out-of-the-money options.

2.2 Deep Neural Network (DNN)

A DNN is a sophisticated iteration of an ANN, characterized by its deeper structure of hidden layers. This complexity enables a DNN to excel at tasks like regression and classification by capturing a more thorough and efficient representation of the input data.

DNNs are suitable for option pricing because they can effectively capture the complex, non-linear relationships between input variables and option prices. Culkin and Das [3] showed that DNNs outperformed traditional models like Black-Scholes in pricing options, particularly for out-of-the-money and long-maturity options. Poggio et al. [7] demonstrated that DNNs could accurately approximate the pricing functions of various options, including exotic ones. The ability of DNNs to automatically learn features from high-dimensional data makes them a powerful tool for option pricing in dynamic and complex financial markets.

2.3 Convolutional Neural Network (CNN)

CNNs represent a specialized kind of ANN known for their local connectivity and shared weights architecture, frequently employed in the field of computer science for their robust capacity to distill information. In our study, we have selected the 1D-CNN variant — a one-dimensional take on the conventional CNN model — tailored for predictive analytics in time-series data. This model adapts to the sequential nature of option market data. Utilizing a combination of convolutional and pooling operations, 1D-CNNs leverage one-dimensional convolutional layers to parse and capture temporal information. This approach enables efficient and rapid training and fitting of the 1D-CNN model to the data in question [14].

2.4 Long Short-Term Memory (LSTM)

LSTM networks, a specialized form of ANNs, are predominantly applied to time-series data for regression and classification tasks. They incorporate unique structures known as input gates (i_t), forget gates (f_t), and output gates (o_t) that enable them to retain pertinent information across time intervals. These gates effectively help LSTMs to preserve relevant historical data within the network, allowing for better predictions and addressing common training challenges such as gradient vanishing and exploding gradients [14].

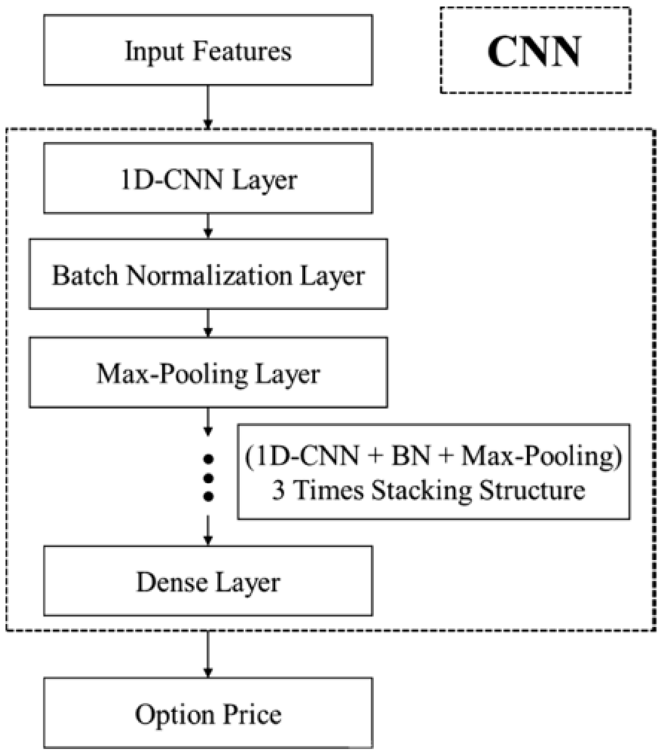
Input Features
CNN
1D-CNN Layer
Batch Normalization Layer
Max-Pooling Layer
(1D-CNN + BN + Max-Pooling)
3 Times Stacking Structure
Dense Layer
Option Price

Figure 1: Structures of CNN[14]

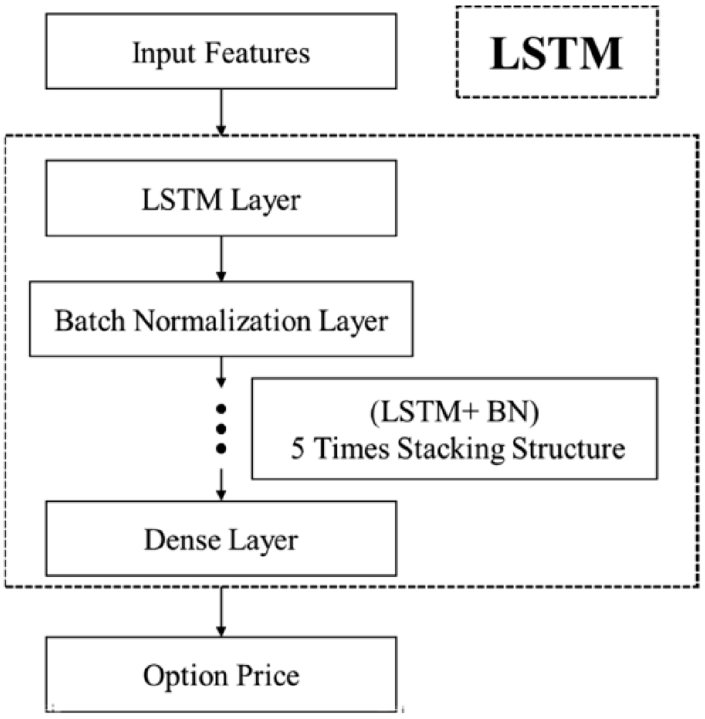

Figure 2: Structures of LSTM[14]

2.5 Transformer

The Transformer, introduced in 2017, marks a significant advancement in machine translation tasks, outperforming traditional long short-term memory (LSTM) models with its superior performance. Although it has become a mainstay in Computer Vision (CV) and Natural Language Processing (NLP), the Self-Attention mechanism of the Transformer is also well-suited for time series prediction tasks. In this study, we leverage the Transformer's robust capability of capturing complex information for predicting option prices [6].

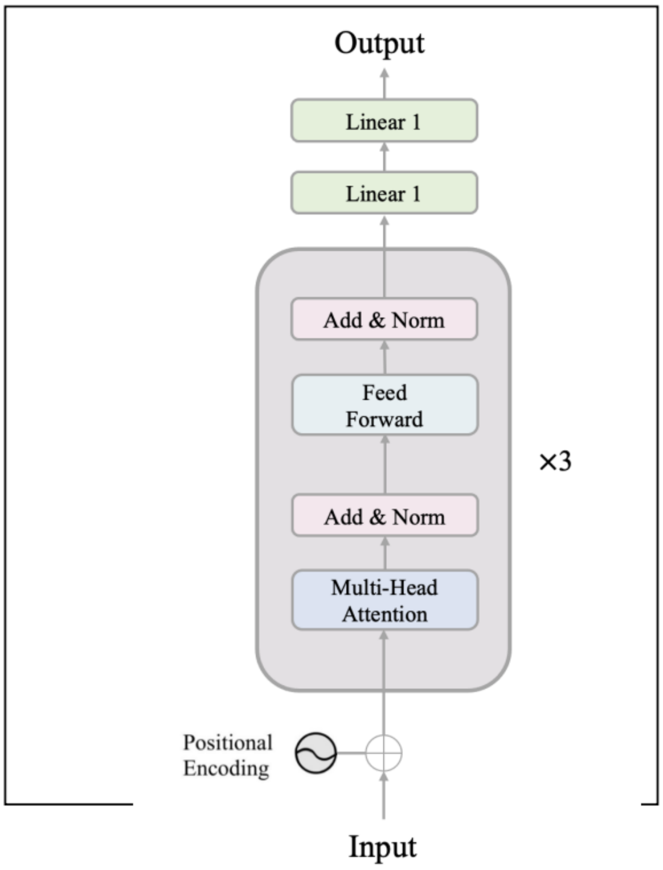

Figure 3: Structures of Transformer model[6]

Transformers are well-suited for option pricing tasks due to their ability to capture long-range dependencies and model sequential data. Gu et al. [5] demonstrated that Transformer-based models outperformed traditional methods and other deep learning models in predicting option prices, showcasing their effectiveness in capturing the complex dynamics of financial markets. Liu et al. [15] further showed that Transformers could accurately forecast implied volatility surfaces, a crucial component in option pricing. The self-attention mechanism in Transformers allows them to focus on relevant information across time series data, making them a powerful tool for option pricing in the face of market volatility and rapidly changing conditions.

2.6 CNN-LSTM

CNN-LSTM is a type of deep neural network that combines convolutional neural networks and long short-term memory networks [2]. CNN is particularly effective at processing grid-like structure data and LSTM is a type of recurrent neural network designed to process sequences of data. The combination of CNN-LSTM allows the model to effectively process data that has both spatial and temporal dimensions, which makes it ideal for tasks such as video classification. The input data is firstly passed through a series of CNN layers. The role of CNN here is to extract a set of features from it. The output representation of each frame of the input data. After feature extraction, the sequence of features produced by CNN is formed into a temporal sequence. Then the sequence of features is then fed into the LSTM layers. The LSTM processes the sequence one element at a time, with the memory cells maintaining information across the frames. This architecture allows CNN-LSTM to effectively capture the spatial features and temporal dependencies of input data.

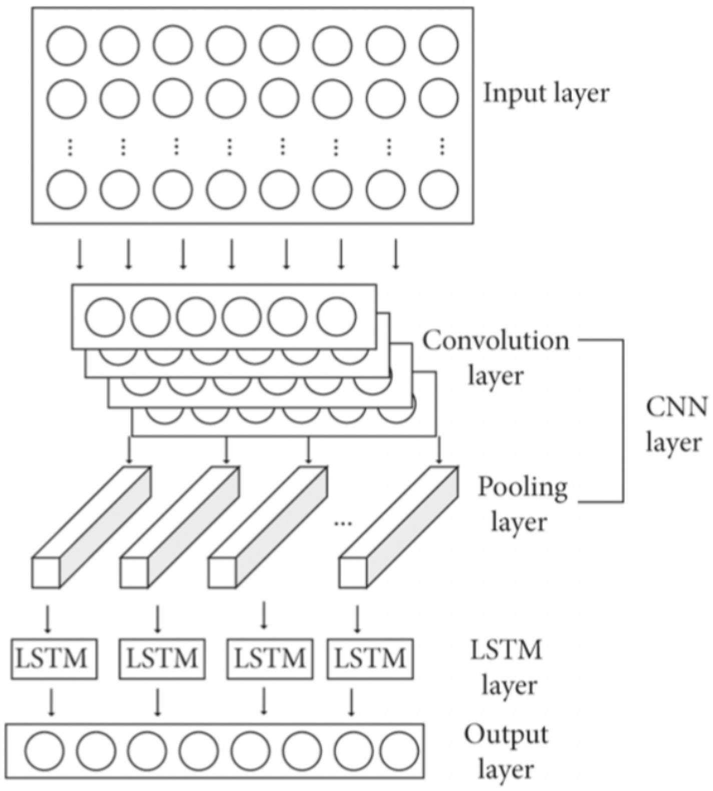

Figure 4: Structures of CNN-LSTM[18]

3. Theories

3.1 ANN/DNN

An ANN consists of input layers, hidden layers, and an output layer. Each layer in the ANN consists of neurons and synapses that can transmit information. A neuron is the basic computational unit of the neural network and performs a weighted sum of inputs and outputs through an activation function. The input layer accepts inputs and transmits it to the hidden layers. This process can be represented as

$$z^{[1]} = W^{[1]}x + b^{[1]} \tag{1}$$

where $z^{[1]}$ is the input to the hidden layer, $W^{[1]}$ is the weight matrix, x is the input vector, and $b^{[1]}$ is the bias vector. Then an activation function is applied on $z^{[1]}$,

$$a^{[1]} = g^{[1]}(z^{[1]}) \tag{2}$$

where $a^{[1]}$ is the output from the hidden layer's activation function $g^{[1]}$. Then the output from the hidden layer is seen as the input to the output layer and then goes through the activation function in the output layer to generate the final output similar to the equation (1) and (2).

3.2 CNN

CNNs makes use of several layers to process and transform the input data through convolutions and pooling. The convolutional layers apply a convolution operation to the input,

$$Z^{[l]} = W^{[l]} * A^{[l-1]} + b^{[l]} \tag{3}$$

where $*$ denotes the convolution operation, $W^{[l]}$ is the weight matrix, $A^{[l-1]}$ is the activation from the previous layer, $b^{[l]}$ is the bias, and $Z^{[l]}$ is the convolved feature. Then non-linear activation functions are applied after each convolution operation to introduce non-linear structure. After applying the activation function, a pooling layer is introduced,

$$P^{[l]} = \text{MaxPool}(A^{[l]}) \tag{4}$$

The pooling layer can reduce the dimensionality of the data but retains the most important information. After one or more convolutional and pooling layers, fully connected layers are introduced to classify the feature extracted by the convolutions into desired output.

3.3 LSTM

LSTMs are a special variant of recurrent neural networks. LSTMs have a chain-like structure. The core component of LSTMs is its cell state and the three gates to regulate the information flow. The forget gate (f_t) decides what information is to be discarded from the cell state. The input gate (i_t) decides which values will update the cell state. Finally, the output gate (o_t) decides what the next hidden state would be. The operations within an LSTM cell at time step t can be described by the following equations:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{5}$$

The forget gate decides which information to throw away through a sigmoid function. It looks at the previous hidden state h_{t-1} and the current input x_t.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{6}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{7}$$

The input gate updates the cell state by first deciding which values will be updated and the creating a vector of new candidate values $\tilde{C}_t$ that could be added to the state through *tanh*.

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{8}$$

The old state, C_{t-1}, is updated to the new state, C_t, by utilizing the forget gate to determine which portion of the old state can remain and the input gate with the candidate values decides which new information can be added.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{9}$$

$$h_t = o_t * \tanh(C_t) \tag{10}$$

Equations (9) and (10) show how the output gate decides what the next hidden state should be. The state is filtered by the output gate and then go through tanh to push the values to be between -1 and 1.

3.4 Transformer

Transformer is a type of neural network that is well-suited for handling sequential data through its self-attention mechanism to weight the significance of each part of the input differently, which is the key innovation of the transformer. The Transformer model consists of encoder-decoder structure. The encoder comprises a stack of identical layers and each layer has a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. The decode also consists of several identical layers, but with an additional third sub-layer that can perform multi-head attention on the encoder's output. The attention can be seen as mapping a query and a set of key value pairs to an output and the output is computed as a weighted sum of the values,

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{11}$$

where K are the keys, V are the values, and Q is the query. The multi-head attention mechanism extends the single attention models by linearly projecting the queries, keys, and values,

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \tag{12}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

This allows the attention function to be applied in parallel to produce outputs and then concatenate the outputs.

3.5 CNN-LSTM

CNN-LSTM is a hybrid neural network model that combines CNN and LSTM to leverage the spatial hierarchy of features learned from CNN with the ability of sequence processing of LSTM. CNN-LSTM is particularly useful for tasks that require recognition and interpretation of sequential data. A CNN is applied to the input at time t, X_t, at first,

$$C_t = \text{CNN}(X_t) \tag{13}$$

where C_t is the feature vector. Then the sequence of CNN outputs $C_1, C_2, \ldots, C_t$ is fed into LSTM to produce final output.

3.6 Activation Functions

Activation functions are a core component in neural networks that introduce non-linearity to the system. Common activation functions used in above architectures include,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{14}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{15}$$

$$\text{ReLU}(x) = \max(0, x) \tag{16}$$

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_k e^{x_k}} \tag{17}$$

3.7 Evaluation Metric (Loss Function)

Mean Squared Error (MSE) is a measure of difference between values predicted by a model and the actual values. Higher MSE value indicates worse performance. It is calculated as

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{18}$$

where n is the number of data points, $\hat{y}_i$ is the predicted value, and y_i is the actual value.

4. Data Description

4.1 Data Generation

A derivatives valuation model is a function that takes inputs, including market data and specific trade terms, and produces a single output: the value of the derivative. Techniques such as Monte Carlo simulation are necessary to simulate this option value. Simple European stock options can be valued using just five inputs: the strike price of an option, the current stock price, the time to expiration, the risk-free rate, and the volatility.

During our project, a method is proposed for generating input data for deep learning models in basket option pricing based on Monte Carlo simulation [4]. The method comprehensively utilizes statistical distributions such as Beta distribution, log-normal distribution, and uniform distribution to simulate and generate key input variables required for basket option pricing, including the forward prices of underlying assets, volatilities, correlation matrices between the assets in the basket, and option maturity times. We set each basket of options containing 6 stocks.

Specifically, we first use the Beta distribution to generate elements of the correlation matrix and control the distribution characteristics of correlations by adjusting shape parameters to simulate various common asset correlation structures. Then, the log-normal distribution is used to generate the forward prices of underlying assets, conforming to the distribution characteristics of financial asset prices. Volatilities are simulated using the uniform distribution, reflecting the uncertainty of volatility. Furthermore, the option maturity times are generated using the square root of the uniform distribution, which aligns with the actual distribution of option maturity times. Finally, the generated input variables are combined into complete simulation data [4].

By repeating the above steps, a large number of simulated data samples can be obtained for training and testing deep learning option pricing models. This method proposed in our project integrates financial domain knowledge and statistical simulation techniques, enabling the generation of training data that conforms to actual characteristics and is sufficient in quantity.

In terms of training set generation, the following 28 key parameters are set when generating training data for basket option pricing using Monte Carlo simulation:

- The mean and standard deviation of the normal distribution are 0.5 and 0.25, respectively, which are used to generate the forward prices of 6 stocks (6 parameters).
- The stock volatilities are generated from a uniform distribution on [0, 1] (6 parameters).
- The option maturities follow a uniform distribution on the interval [1, 4], and the results are squared (1 parameter).

- The parameters of the Beta distribution are $\alpha = 5$ and $\beta = 2$, which are used to generate a 6-dimensional correlation matrix ($6 \times 5/2 = 15$ parameters).
- The strike price of the options is set to 100.
- The risk-free rate of the options is set to 0.05.
- The sample size n_samples is set to 100,000.

We utilize the generated option data as the input dataset for our model to price a European call option on a worst-of basket with six underlying stocks:

$$V = max(0, min(Stock1, Stock2, Stock3, Stock4, Stock5, Stock6) - K)$$

For each basket of options, we employ 100,000 steps of Monte Carlo paths to simulate the generation of option prices, which are then used as the output dataset for the model. This simulation process ensures precision and randomness in price generation, providing high-quality training and testing data for the deep learning model.

It takes about 50 minutes (2969 seconds) for the Monte Carlo simulation throughout 100,000 basket options. The previous research mentions that generating 50 million data with each basket option simulates a same 100,000 steps takes about a week, which is actually faster than our generation [4].

4.2 Data Preprocessing

To facilitate better learning and generalization by the neural network, it is necessary to standardize the input data. We use the MinMaxScaler to scale forward prices, volatilities, and option prices parameters to a range of [0, 1]. This process aids in optimizing the performance of gradient descent algorithms and avoids numerical stability issues during training due to large differences in data scale and magnitude.

We randomly split the dataset into 70% training data and 30% testing data.

5. Model Implementation

5.1 Implementation Details

In the implementation phase of our project, we developed various deep learning models with configurations of 1, 3, and 5 layers/blocks with 128 neurons/filters per layer to compare the performance of different models with different architectures. These models are constructed using TensorFlow Keras library. We standardized our training process by setting the batch size to 256 and training each model for 50 epochs to compare the performance across different models under a consistent setting. For optimization, we

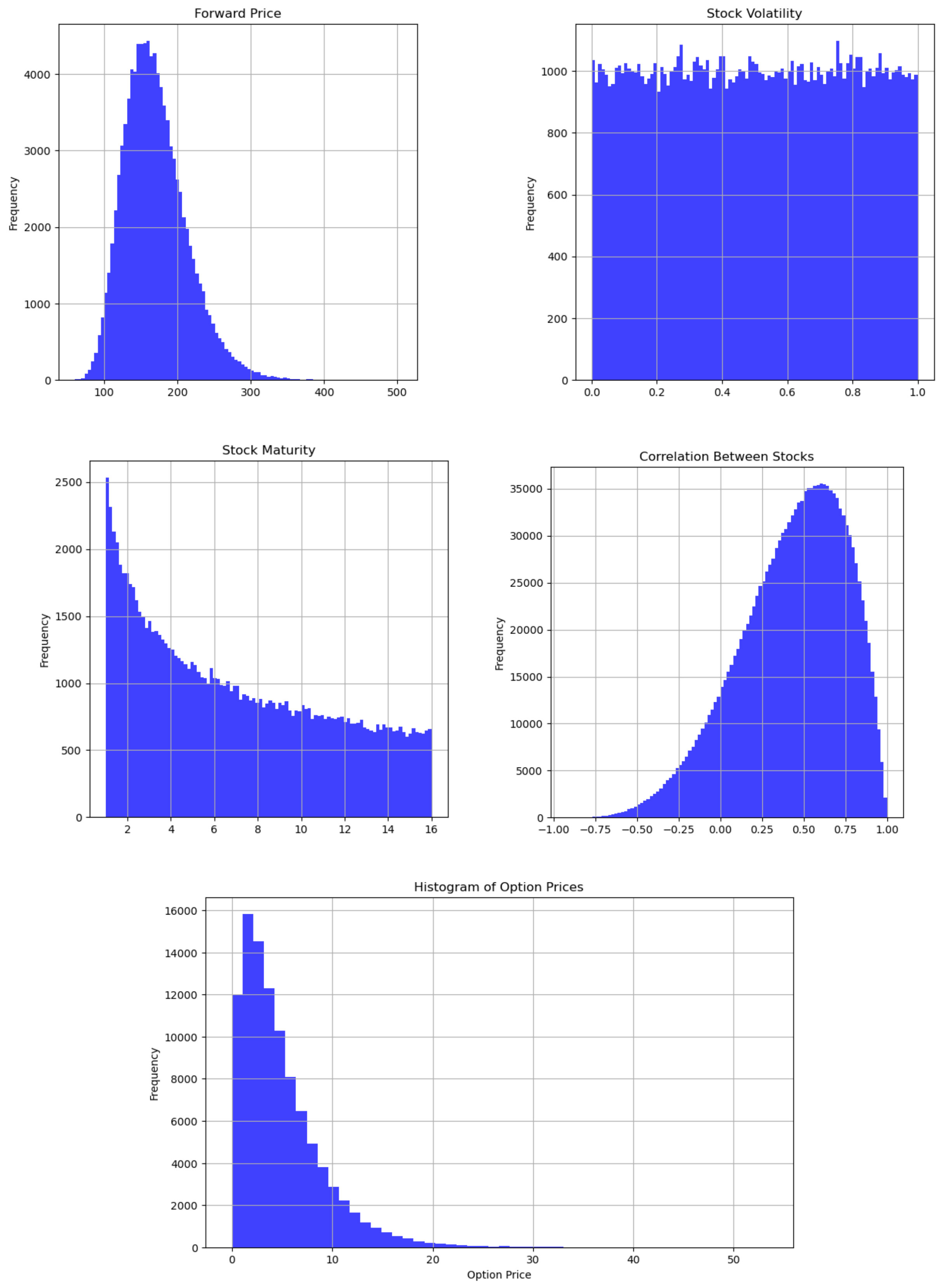

Figure 5: Illustrative counts of the input variable distributions for training and testing the option pricing model, showing the distribution of parameters. The distinct diagrams depict the following: (a) the market price of stocks; (b) the volatilities of the stocks; (c) the term to maturity of the option, expressed in year; (d) the correlation metric for a pair of stocks; (e) the simulated value of the option basket.

employed the Adam optimizer with learning rate 0.001, b1 of 0.9, and b2 of 0.999. For the activation functions, we used the default activation function, for DNN and CNN and for LSTM. These consistent setups allowed us to fairly compare the performance of each model.

The implementation details for DNN, CNN, and LSTM are relatively simple. The more complex model CNN-LSTM employs the TimeDistributed wrapper to apply convolution and pooling layers across each time step of input data after the convolutional and pooling layers. The purpose of TimeDistributed wrapper is to flatten the convoluted data then pass it to the LSTM layers. All LSTM layers except the last one return sequences, which allows for the continuation of sequence processing. The final LSTM layer outputs a single vector which is then fed into a Dense layer to produce the final output.

Algorithm 1 CNN-LSTM Model Construction and Compilation

```
 1: procedure CNN-LSTM Model
 2:     Input: Input shape ($n_{\text{timesteps}}, n_{\text{features}}, 1$)
 3:     Initialize model with Sequential API
 4:     Add InputLayer with specified input shape
 5:     Add TimeDistributed layer with Conv1D(filters=128, kernel_size=1, activation='relu')
 6:     Add TimeDistributed layer with MaxPooling1D(pool_size=1)
 7:     Add TimeDistributed layer with Flatten()
 8:     Add LSTM layer (128 units, activation='tanh', return_sequences=False)
 9:     Add Dense output layer (1 unit)
10:     Define optimizer: Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999)
11:     Compile model with optimizer, loss='mean_squared_error'
12: end procedure
13: Output: Compiled CNN-LSTM model ready for training
```

The implementation details of the MultiHeadAttention and TransformerBlock classes along with a model-building function used the Tensorflow Keras API. The MultiHeadAttention Layer starts with an Initialization layer which initializes the dimensionality of the model and the number of attention heads. The Splitting Heads reshapes and transposes the input tensor to prepare for multi-head attention, which dives the model's dimensions into multiple heads. Then during the forward pass, queries, keys, and values are first projected using dense layers and then split into multiple heads and processed through scaled dot-product attention. In the TransformerBlock Layer, after the input undergoes multi-head attention, the output then passes through the feed-forward network, followed by another round of normalization and summation, which produces the final output of the block [20].

Algorithm 2 MultiHead Attention and Transformer Block

```
1: procedure MultiHeadAttention($d_{\text{model}}$, num_heads)
2:     assert $d_{\text{model}}$%num_heads == 0
3:     depth ← $d_{\text{model}}$/num_heads
4:     wq, wk, wv ← Dense($d_{\text{model}}$), Dense($d_{\text{model}}$), Dense($d_{\text{model}}$)
5:     dense ← Dense($d_{\text{model}}$)
6: end procedure
7: function SplitHeads($x$, batch_size)
8:     $x$ ← reshape($x$, (batch_size, −1, num_heads, depth))
9:     return transpose($x$, [0, 2, 1, 3])
10: end function
11: function Call($v, k, q$, batch_size)
12:     $q$ ← wq($q$); $k$ ← wk($k$); $v$ ← wv($v$)
13:     $q, k, v$ ← SplitHeads($q$, batch_size), SplitHeads($k$, batch_size), SplitHeads($v$, batch_size)
14:     matmul_qk ← matmul($q, k$, transpose_b = $True$)
15:     dk ← cast(shape($k$)[−1], float32)
16:     scaled_attention_logits ← matmul_qk/$\sqrt{\text{dk}}$
17:     attention_weights ← softmax(scaled_attention_logits, axis = −1)
18:     $output$ ← matmul(attention_weights, $v$)
19:     $output$ ← transpose($output$, [0, 2, 1, 3])
20:     $output$ ← reshape($output$, (batch_size, −1, $d_{\text{model}}$))
21:     return dense($output$)
22: end function
23: procedure TransformerBlock($d_{\text{model}}$, num_heads, $dff$, rate)
24:     att ← MultiHeadAttention($d_{\text{model}}$, num_heads)
25:     ffn ← [Dense($dff$, activation =′ $relu$′), Dense($d_{\text{model}}$)]
26:     layernorm1, layernorm2 ← LayerNormalization($\epsilon = 1e - 6$), LayerNormalization($\epsilon = 1e - 6$)
27:     dropout1, dropout2 ← Dropout(rate), Dropout(rate)
28: end procedure
29: function BuildModel(input_shape, $d_{\text{model}}$, num_heads, $dff$, rate, layers)
30:     $inputs$ ← Input(shape = (input_shape, ))
31:     $x$ ← Dense($d_{\text{model}}$)($inputs$)
32:     for $i = 1$ to layers do
33:         $x$ ← TransformerBlock($d_{\text{model}}$, num_heads, $dff$, rate)($x$, training = $True$)
34:     end for
35:     $outputs$ ← Dense(1)($x$)
36:     return $outputs$
37: end function
```

5.2 Practical Difficulties

5.2.1 Difficulties Encountered in Hardware/Library

During the training process, in order to minimize the training time, each of us implemented and trained different models separately. However, we discovered that the performance of the same model on the same dataset can be significantly different on two different computers. This might be due to differences in hardware. The type of GPU, CPU, and even the version of these devices can affect how the computations in the training process are handled, which might lead to variations in results [11]. In addition, mismatch in software versions, especially in Tensorflow version can cause different behavior in training dynamics [1]. In order to fix this issue and ensure all the training and computation are done under the same setting, all of the final result of our project is produced on the computer of one of us.

5.2.2 Difficulties Encountered in Transformer Implementation

The Transformer model, with its encoder-decoder architecture and attention mechanism, was being trained on a large corpus of text data for a machine translation task. We initially attempted to utilize large batch sizes (512 and 1024) to accelerate the training process and potentially improve convergence. However, due to the model's complexity and the memory requirements of such batch sizes, we encountered GPU memory limitations on the available NVIDIA Tesla V100 GPUs with 16GB memory each. This experience highlighted the memory limitations of training complex models like Transformers with large batch sizes. It emphasized the importance of carefully considering hardware resources and exploring memory-efficient techniques during the training process [20]. In future work, we plan to investigate distributed training across multiple GPUs and explore model architectures with lower memory footprints while maintaining comparable performance [19][12].

6. Results

6.1 Accuracy

General findings and insights could be found according to the test loss of various models with different layer configurations. Firstly, all models generally exhibit a decrease in loss over epochs, demonstrating learning and improvement. Additionally, models with more layers tend to start with a higher initial loss, likely due to increased complexity and the need to learn more parameters. Moreover, most models seem to converge towards similar loss values, although the rate and smoothness of convergence vary.

Initially, we can observe the performance of various models on a single layer. Figure 5 illustrates the test loss trend of five distinct 1-layer models over the course of training

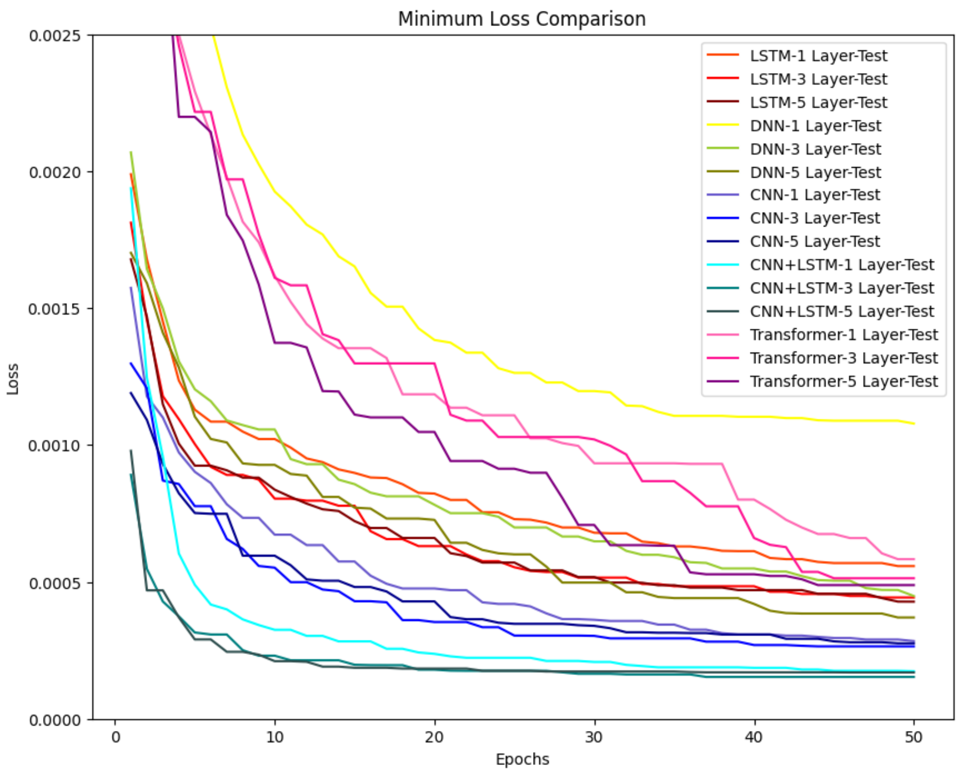

Figure 6: Minimum Test Loss in Different Models with Varying Layers

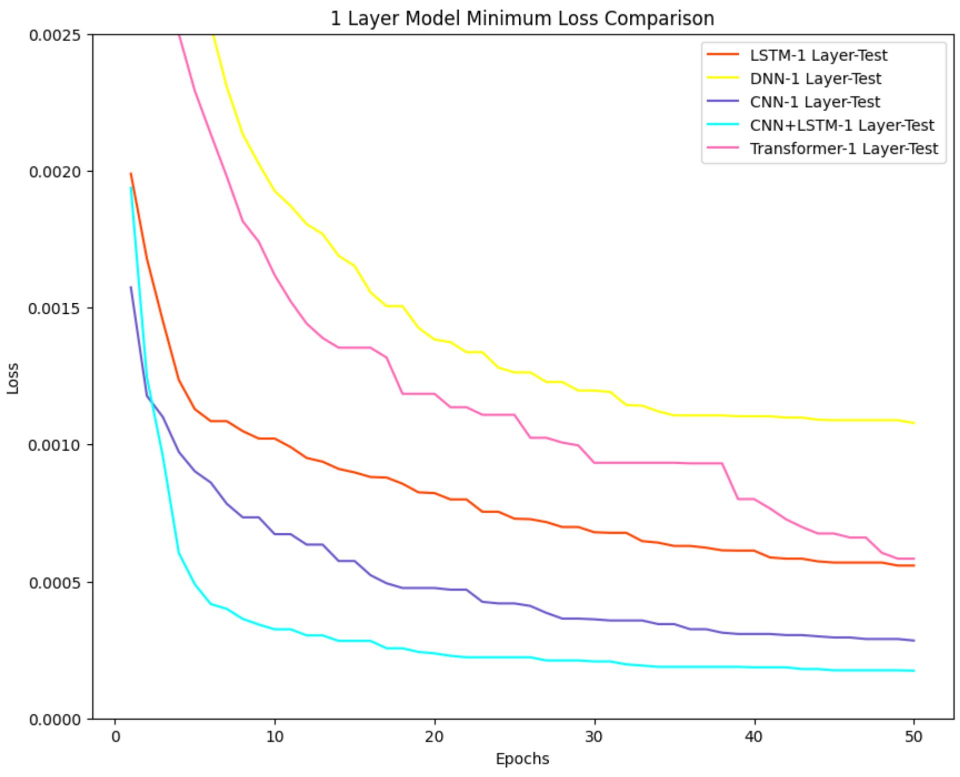

Figure 7: Minimum Test Loss in Different Models with 1 Layer

epochs.

While the CNN-LSTM model demonstrates the strongest performance with rapid initial convergence, its susceptibility to fluctuations suggests that even this hybrid approach might require additional depth for optimal stability and generalization.The CNN model also exhibits strong performance, highlighting the importance of spatial feature extraction even in a single-layer architecture. The LSTM model, while achieving decent convergence, experiences more fluctuations, indicating the presence of temporal dependencies but potentially requiring additional layers for effective capture. Surprisingly, the Transformer model, known for its strength in handling long-range dependencies, exhibits slower convergence and a higher final test loss compared to CNN and CNN-LSTM. This suggests that a single-layer Transformer might not be sufficient to effectively learn the data representation for this task. The DNN model, with the slowest convergence and highest final test loss, clearly demonstrates the limitations of a single-layer architecture for capturing the complexities of the data, likely resulting in underfitting.

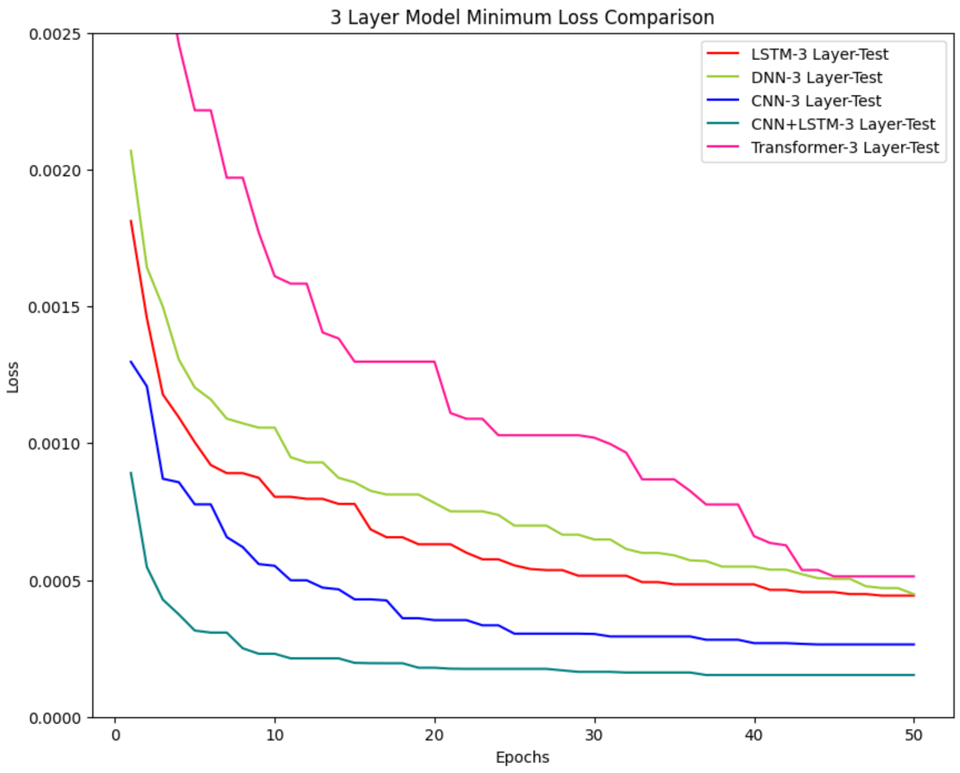

Figure 8: Minimum Test Loss in Different Models with 3 Layer

In addition, let us examine the efficacy of different models across three layers. The CNN-LSTM model demonstrates superior performance, achieving rapid convergence to a low and stable test loss, indicating its effectiveness in capturing both spatial and temporal features within the data. The CNN model also exhibits strong performance with a relatively smooth loss curve, highlighting the importance of spatial feature extraction for this specific problem. Its slightly higher final test loss compared to the CNN-LSTM model suggests that temporal dependencies might also play a role, albeit a secondary one. The LSTM model demonstrates decent convergence but with more fluctuations in the loss curve, indicating potential challenges in optimization or a higher sensitivity to the specific characteristics of the data. Both the DNN and Transformer models exhibit slower convergence and higher final test loss values, with noticeable fluctuations that could indicate overfitting. This suggests that these architectures might not be optimally suited for capturing the underlying patterns in the data or may require further hyperparameter tuning and regularization techniques.

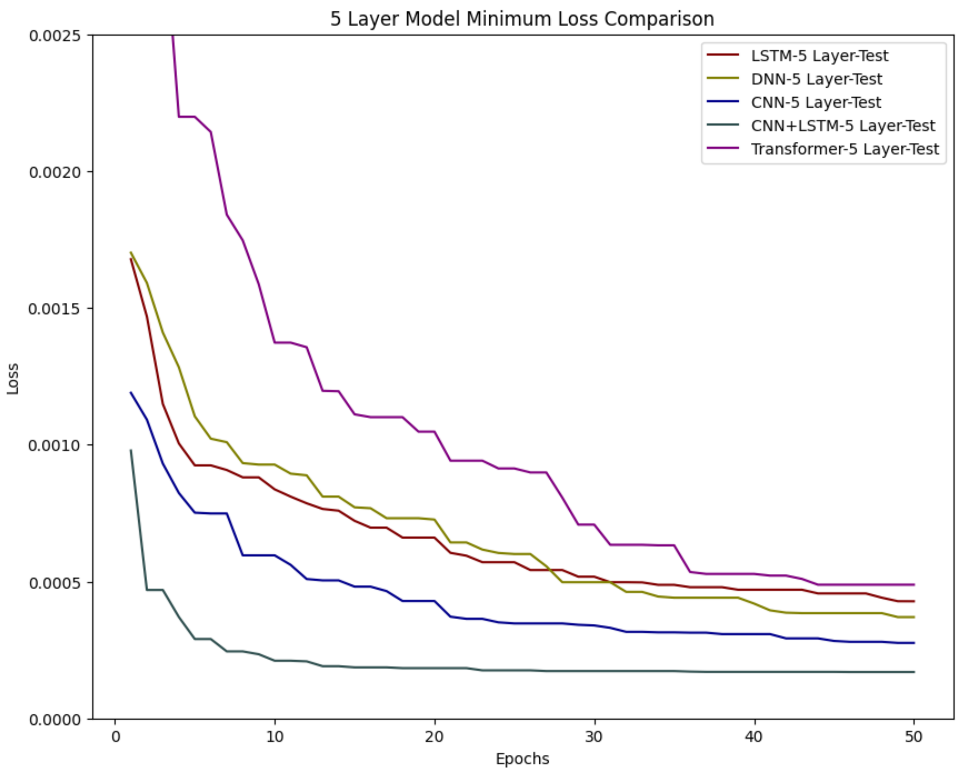

Figure 9: Minimum Test Loss in Different Models with 5 Layer

Interestingly, the overall performance trends observed in the 5-layer models closely mirror those seen in the 3-layer models. The CNN-LSTM model maintains its position as the top performer, effectively capturing both spatial and temporal features within the data. Similarly, the CNN model continues to demonstrate strong performance, highlighting the importance of spatial feature extraction. The LSTM model also exhibits decent convergence but with more fluctuations, suggesting the presence of temporal dependencies, albeit less dominant than spatial features. However, the increase in model complexity from 3 to 5 layers appears to have a more pronounced negative impact on the DNN and Transformer models. Their slower convergence, higher final test loss values, and increased fluctuations suggest a higher susceptibility to overfitting with added layers.

6.2 Speed

LSTM and DNN exhibit a marginal increase in training time as the number of layers scales up, suggesting a linear or sub-linear relationship between model complexity and computational cost, indicative of relatively efficient scaling with increased depth. CNN displays minimal variance in training times across different layer counts, implying that the training duration for CNNs is less sensitive to the depth of the model, potentially due to the parameter-sharing characteristic of convolutions that mitigates the impact of

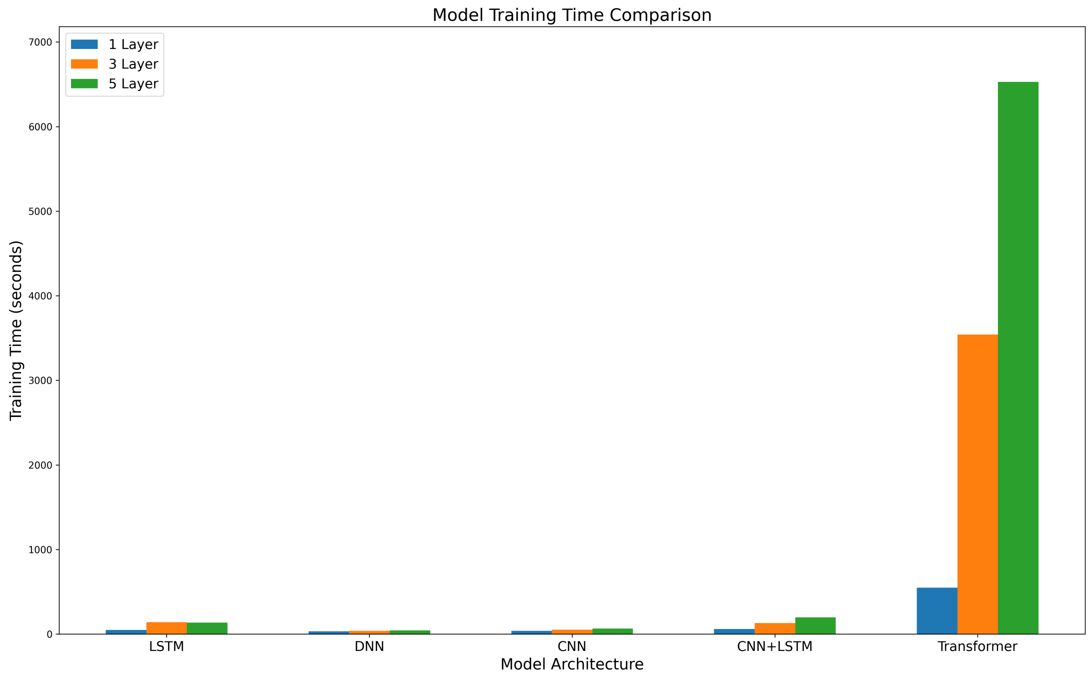

Figure 10: Training Time in Different Models with Varying Layers

additional layers. The hybrid CNN+LSTM architecture shows an incremental rise in training times with added layers, yet the increase remains modest, hinting at effective integration of spatial and temporal feature extraction while controlling computational expenses. In contrast, the Transformer architecture stands out with significantly higher training times, especially at the five-layer configuration, where it surpasses all other models by a considerable margin.

Table 1: Summary of MSE and Speed Comparison

	DNN	CNN	LSTM	CNN-LSTM	Transformer
1 Layer/Block					
MSE	0.00108	0.00030	0.00060	**0.00017**	0.00063
Speed (sec)	32.2348	39.2719	48.8501	59.8997	547.5491
3 Layers/Blocks					
MSE	0.00045	0.00027	0.00045	**0.00015**	0.00065
Speed (sec)	38.2244	51.6900	140.6393	129.7185	3541.8487
5 Layers/Blocks					
MSE	0.00037	0.00030	0.00042	**0.00017**	0.00061
Speed (sec)	42.7045	64.4978	135.5918	196.8069	6529.0911

Figure 11 visualizes the relationship between training time and mean squared error (MSE) across various model architectures with different numbers of layers or blocks. There is a clear trend indicating that as model complexity increases with more layers/blocks, the training time correspondingly escalates, often substantially. Transformer models exhibit the longest training times, followed by CNN-LSTM, LSTM, CNN, and DNN architec-

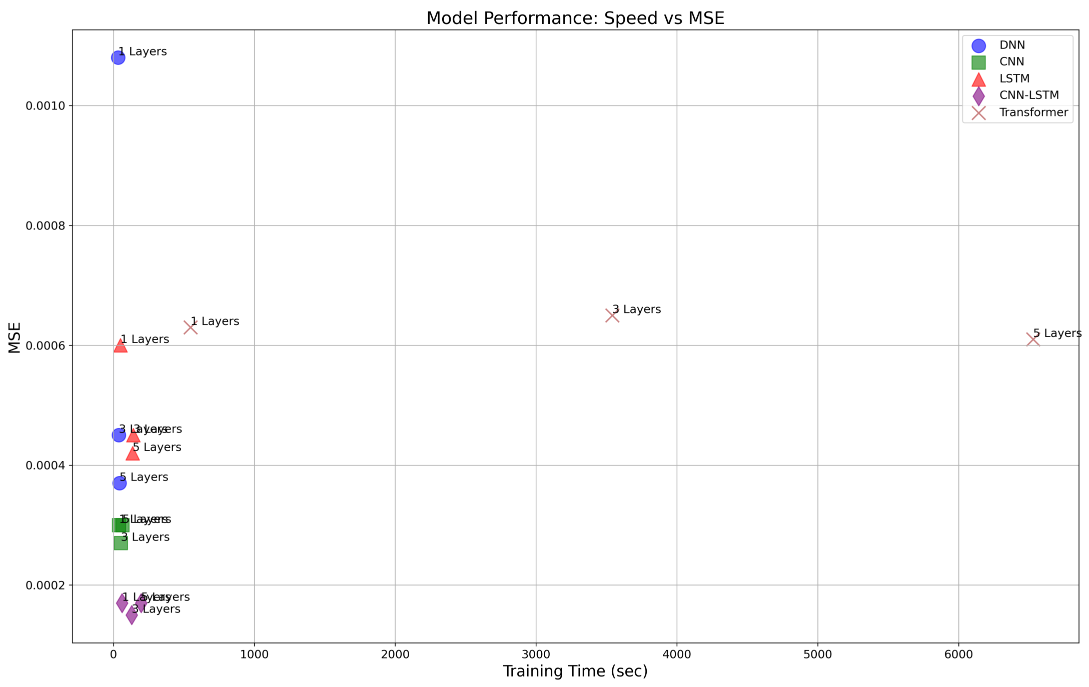

Figure 11: Relationship between Speed and MSE

tures respectively. Notably, CNN-LSTM models with 3 and 5 layers/blocks achieve the lowest MSE, closely trailed by their CNN counterparts with equivalent layer counts. Conversely, DNN and LSTM models generally demonstrate slightly higher MSE compared to CNN and CNN-LSTM architectures with the same layer configurations. While Transformer models display relatively high MSE despite their prolonged training durations, suggesting a potential trade-off between speed and accuracy inherent to this architecture. Furthermore, some variation in MSE and training time exists even among models with identical layer counts, implying that architectural nuances and hyperparameter tuning can significantly impact performance metrics.

The CNN-LSTM is the outlier that performs best in the speed-acc trade-off when it comes to the relationship between cost and MSE. Compared to other models, the total parameter range of 1,901,441 - 2,493,825 is significantly higher. Therefore, despite having the highest acc, the large number of parameters would result in high memory and computation costs. As a result, CNN is the ideal model to replace it since, regardless of layer composition, CNN consistently tends toward the coordinate axis' origin, which is the optimal result we are looking for. CNN-LSTM and LSTM come next. DNN follows, and at last, the transformer.

6.3 Cost

In terms of Figure 12, LSTM and CNN models exhibit a clear increase in neuron count with additional layers, reflecting the need for greater complexity to capture intricate patterns in data. In contrast, DNNs maintain a remarkably low neuron count even with increasing layers, highlighting their parameter efficiency. The hybrid LSTM+CNN model

Table 2: Summary of MSE and Cost Comparison

	DNN	CNN	LSTM	CNN-LSTM	Transformer
1 Layer/Block					
MSE	0.00108	0.00030	0.00060	**0.00017**	0.00063
Parameters	961	20,353	80,513	1,901,441	202,113
3 Layers/Blocks					
MSE	0.00045	0.00027	0.00045	**0.00015**	0.00065
Parameters	5,153	53,377	343,681	2,197,633	598,657
5 Layers/Blocks					
MSE	0.00037	0.00030	0.00042	**0.00017**	0.00061
Parameters	21,729	86,401	606,849	2,493,825	995,201

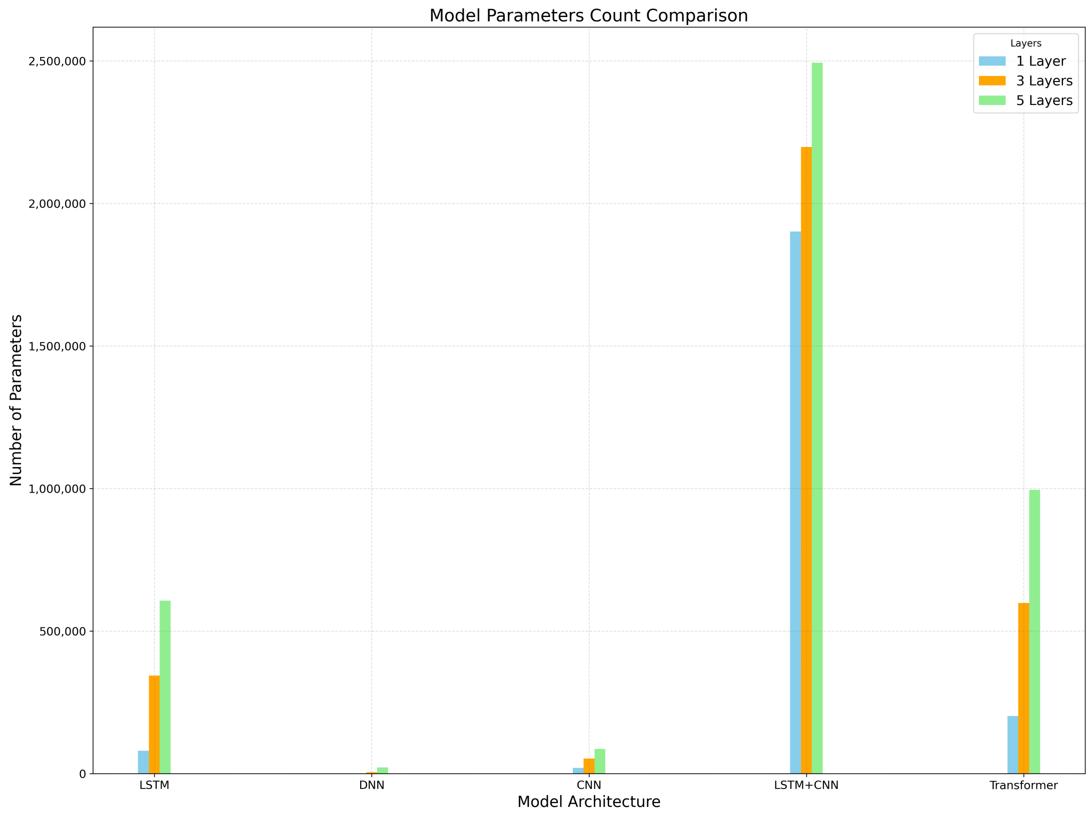

Figure 12: Model Complexity in Different Models with Varying Layers

stands out with the highest neuron count, indicating its substantial complexity due to the combination of convolutional and recurrent layers. Interestingly, the Transformer model, known for its powerful performance, demonstrates a relatively lower neuron count compared to the LSTM+CNN model, likely due to its efficient use of attention mechanisms.

Figure 13 delineates the relationship between total model parameters, an indicator of model complexity, and MSE, a metric for prediction accuracy, across various model architectures with differing layer configurations. Increase of the number of layers leads to a substantial escalation in the total parameter count for each model type, signifying higher

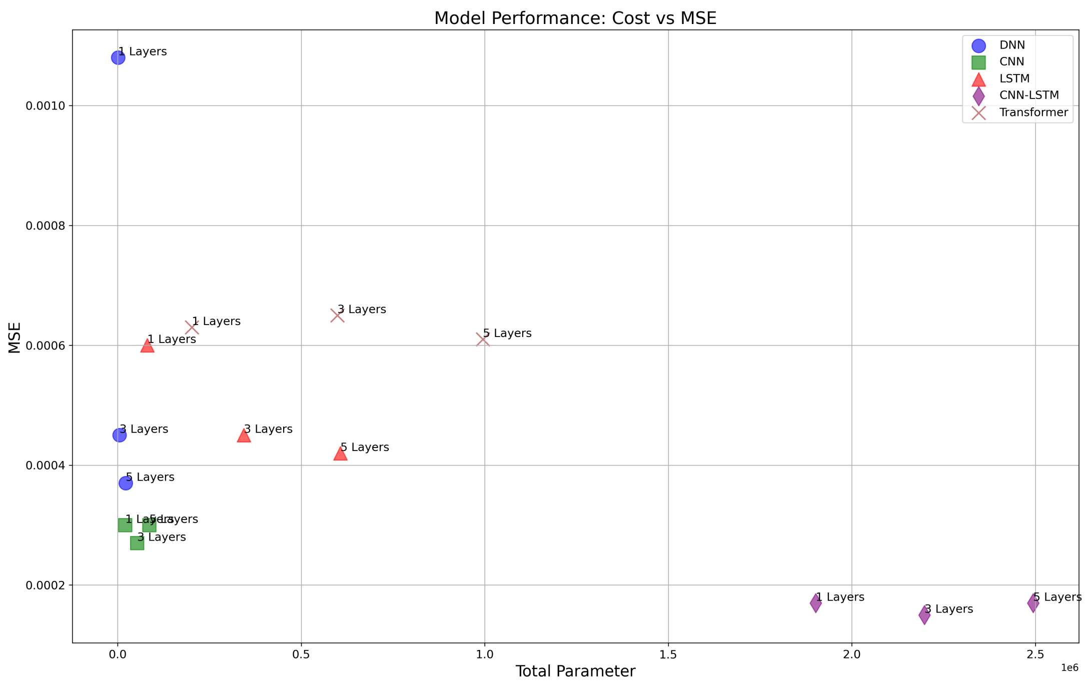

Figure 13: Relationship between Cost and MSE

model complexity. Transformer models exhibit the highest parameter counts, followed by CNN-LSTM, LSTM, CNN, and DNN architectures, respectively. Notably, CNN-LSTM models with 3 and 5 layers achieve the lowest MSE, closely trailed by their CNN counterparts with equivalent layer counts. In contrast, DNN and LSTM models generally demonstrate slightly higher MSE compared to CNN and CNN-LSTM architectures with the same layer configurations. While Transformer models display relatively high MSE despite their substantial model complexity, suggesting that excessive complexity beyond a certain threshold may not necessarily translate to improved performance. Furthermore, some variation in MSE and total parameter count exists even among models with identical layer counts, implying that architectural nuances and hyperparameter tuning can significantly impact performance metrics.

7. Conclusion

7.1 Primary Findings

In conclusion, this project embarked on a comprehensive exploration of deep learning models for basket option pricing under a uniform setting, utilizing Monte Carlo simulations for data generation. Through rigorous evaluation of accuracy, speed, and cost across various architectures with different layer configurations, several key findings emerged as follows.

- CNN-LSTM consistently outperformed other models in terms of accuracy: Its ability to capture both spatial and temporal features proved highly effective for this task.

This aligns with the findings of Liu Xiong (2022), highlighting the effectiveness of combining convolutional and recurrent layers for capturing complex dependencies in basket option pricing [17].

- CNN demonstrated strong performance, highlighting the importance of spatial feature extraction: While slightly less accurate than CNN-LSTM, it offered a good balance between accuracy and complexity.

- LSTM achieved decent convergence but with more fluctuations, suggesting the presence of temporal dependencies: Additional layers or hyperparameter tuning might further improve its performance.The ability of LSTMs to handle time series data makes them a potential candidate for basket option pricing, as demonstrated by Hochreiter Schmidhuber (1997) [8].

- DNN and Transformer models exhibited slower convergence and higher test loss, especially with increased layers: They might be more susceptible to overfitting or require further optimization for this specific task. DNNs, as foundational deep learning models, may be limited by their simpler structure (LeCun et al., 2015) [13], while the complexity of Transformers may necessitate more extensive training data and computational resources (Vaswani et al., 2017) [20].

- Speed and cost varied significantly across models: DNNs were the most efficient, while Transformers were the most computationally expensive. CNNs offered a good balance between speed and accuracy.

- CNN and CNN-LSTM achieve the best performance when 3 layers are implemented while when increasing the number of layers, DNN, LSTM, and Transformer show better performance. This shows that CNN and CNN-LSTM require less complexity to achieve best performance while DNN, LSTM, and Transformer need more careful hyperparameter tuning.

Overall, the choice of the optimal model depends on the specific priorities and constraints. If accuracy is paramount and computational resources are available, CNN-LSTM emerges as the top contender. For scenarios where efficiency and interpretability are crucial, DNNs offer a viable alternative. CNNs provide a good balance between accuracy and speed, making them suitable for various applications.

7.2 Future Exploration Suggestions

We suggest future research can explore the performance of these models on real-market data. While the simulated data provides a great starting point to compare the prediction performance and speed, real market data often present more complexity due to factors such as market volatility and many external economic influences. The added complexity can significantly affect the model performance, which highlights the need for further studies to verify the robustness of our findings in a more dynamic market environment.

In addition, while our research covers a wide range of deep learning models, including the basic DNN, recurrent type neural networks (LSTM), convolution neural networks, state-of-art transformer, and hybrid model (CNN-LSTM), there are still other newly introduced models that worth to take a look. For example, Hun [9] introduced Exponential Levy Neural Network for option pricing. This model combines ANN and the exponential Levy process to take advantage of nonparametric deep learning models and traditional parametric models. This model is designed to handle the overfitting issues of deep learning models and the parameter estimation issues of Levy process. However, it is still valuable to conduct empirical experiments on these newly introduced hybrid models under uniform setting with other benchmarks models to verify its efficiency and accuracy.

References

[1] Martín Abadi et al. '{TensorFlow}: a system for {Large-Scale} machine learning'. In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Ali Agga et al. 'CNN-LSTM: An efficient hybrid deep learning architecture for predicting short-term photovoltaic power production'. In: *Electric Power Systems Research* 208 (2022), p. 107908.

[3] Robert Culkin & Sanjiv R Das. 'Machine learning in finance: the case of deep learning for option pricing'. In: *Journal of Investment Management* 15.4 (2017), pp. 92–100.

[4] Ryan Ferguson & Andrew Green. 'Deeply learning derivatives'. In: *arXiv preprint arXiv:1809.02233* (2018).

[5] Shihao Gu, Bryan Kelly & Dacheng Xiu. 'Empirical asset pricing via machine learning'. In: *The Review of Financial Studies* 33.5 (2020), pp. 2223–2273.

[6] Tingyu Guo & Boping Tian. 'The Study of Option Pricing Problems based on Transformer Model'. In: *2022 International Conference on Information Science and Communications Technologies (ICISCT)*. IEEE. 2022, pp. 1–5.

[7] JB Heaton, Nicholas G Polson & Jan Hendrik Witte. 'Deep learning in finance'. In: *arXiv preprint arXiv:1602.06561* (2016).

[8] Sepp Hochreiter & Jürgen Schmidhuber. 'Long short-term memory'. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[9] Jeonggyu Huh. 'Pricing options with exponential Lévy neural network'. In: *Expert Systems with Applications* 127 (2019), pp. 128–140.

[10] James M Hutchinson, Andrew W Lo & Tomaso Poggio. 'A nonparametric approach to pricing and hedging derivative securities via learning networks'. In: *The journal of Finance* 49.3 (1994), pp. 851–889.

[11] Norman P Jouppi et al. 'In-datacenter performance analysis of a tensor processing unit'. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[12] Zhenzhong Lan et al. 'Albert: A lite bert for self-supervised learning of language representations'. In: *arXiv preprint arXiv:1909.11942* (2019).

[13] Yann LeCun, Yoshua Bengio & Geoffrey Hinton. 'Deep learning'. In: *nature* 521.7553 (2015), pp. 436–444.

[14] Longyue Liang & Xuanye Cai. 'Time-sequencing European options and pricing with deep learning–Analyzing based on interpretable ALE method'. In: *Expert Systems with Applications* 187 (2022), p. 115951.

[15] Minghao Liu et al. 'Gated transformer networks for multivariate time series classification'. In: *arXiv preprint arXiv:2103.14438* (2021).

[16] Shuaiqiang Liu, Cornelis W Oosterlee & Sander M Bohte. 'Pricing options and computing implied volatilities using neural networks'. In: *Risks* 7.1 (2019), p. 16.

[17] Yan Liu & Xiong Zhang. 'Option pricing using lstm: A perspective of realized skewness'. In: *Mathematics* 11.2 (2023), p. 314.

[18] Wenjie Lu et al. 'A CNN-LSTM-based model to forecast stock prices'. In: *Complexity* 2020 (2020), pp. 1–10.

[19] Mohammad Shoeybi et al. 'Megatron-lm: Training multi-billion parameter language models using model parallelism'. In: *arXiv preprint arXiv:1909.08053* (2019).

[20] Ashish Vaswani et al. 'Attention is all you need'. In: *Advances in neural information processing systems* 30 (2017).

[21] Jingtao Yao, Yili Li & Chew Lim Tan. 'Option price forecasting using neural networks'. In: *Omega* 28.4 (2000), pp. 455–466.