Pricing and Hedging American-Style Options with Deep Learning

Xinyi Ke, Xiaoxi Zhang

May 2024

Abstract

This report explores the pricing and hedging of Bermudan max-call options, which allow early exercise and depend on the maximum value of a basket of assets.

We address the valuation by formulating the exercise strategy as a stopping problem. Utilizing neural networks, we train a decision function that maps from a multi-dimensional space to exercise decisions, effectively determining the optimal exercise time for the option. The robustness of the estimated optimal value is enhanced through statistical methods, including the law of large numbers and the central limit theorem, providing a reliable confidence interval for the option's value.

Additionally, we explore two primary methods for hedging: direct minimization of the mean squared error between the hedging portfolio and the option's payoff, and delta-hedging, which is refined either through finite differences or neural network predictions of delta values. Each method leverages deep learning to approximate critical functions and minimize the hedging error, demonstrating the effectiveness of integrating machine learning into complex derivative hedging.

As for numerical experiments, we specifically focusing on the Transformer, Multilayer Perceptron (MLP), and Convolutional Neural Network (CNN) models. Utilizing simulated stock price dynamics generated by Geometric Brownian Motion (GBM), the study aims to develop an optimal stopping strategy to maximize expected rewards.

The optimization problem is tackled by training individual models at each timestep in reverse order, utilizing information from the subsequent timesteps to minimize the negative estimated expected reward, which serves as the loss function for that specific timestep. These individual models are then concatenated into a sequence model.

Results highlight the superiority and robustness of the Transformer model in the task, despite potential computational efficiency challenges for real-time applications. Conversely, the MLP offers speed advantages, while the CNN exhibits competitive performance with moderate computational demands. However, Graph Neural Networks (GNNs) yielded suboptimal results with the simplistic approach employed.

Keywords: Bermudan max-call options, Pricing, Hedging, Neural Network, Optimal Stopping, GBM, MLP, CNN, Transformer

1 Theory

1.1 Optimal Stopping Strategy

Due to the nature of American options, we have to determine the exercise date with the largest value of options rather than only on the expiration date. In order to do that, we consider an option that can be exercised at discrete time $0 = t_0 < t_1 < \cdots < t_N = T$. Define a discrete-time Markov process $X = (X_n)_{n=0}^N$ valued in $\mathbb{R}^d$ on a filtered probability space $(\Omega, \mathcal{F}, \mathbb{F} = (\mathcal{F}_n)_{n=0}^N, \mathbb{P})$, where $\mathcal{F}_n$ is the smallest σ-field generated by $X_0, X_1, \ldots, X_n$ and $\mathbb{P}$ is the pricing measure. And denote a stopping time as $\tau : \Omega \to \{0, 1, \ldots, N\}$, to be clear, for any $n \in 0, 1, \ldots, N$ the event $\{\tau = n\}$ is belonged to $\mathcal{F}_n$. If holder exercises at time t_n, the option yields a payoff can be captured by a measurable function $g(n, X_n) : \{0, 1, \ldots, N\} \times \mathbb{R}^d \to [0, \infty)$. Then our optimal stopping strategy equals to solve the problem

$$V = \sup_{\tau \in \mathcal{T}} E[g(\tau, X_\tau)], \tag{1}$$

where $\mathcal{T}$ is the set of all stopping times τ and we assume that function g is satisfied with some conditions to make sure the problem above is well defined.
Optimally stopping the discrete-time Markov Process X can be transformed into making a $0-1$ stopping decisions whether or not stopping at time t_n, which is a measurable function $f_n(X_n) : \mathbb{R}^d \to \{0,1\}, n = 0,1,\ldots,N$. Then, we consider the dual stopping problem

$$V_n = \sup_{\tau \in \mathcal{T}_n} E[g(\tau, X_\tau)] \, n = 0,1,\ldots,N, \tag{2}$$

where $\mathcal{T}_n$ is the set of all stopping times which satisfies $n \leq \tau \leq N$. According to a well-known result, to compute an approximation solution to our original problem, it is sufficient to consider the stopping problem below

$$\tau_n = \sum_{m=n}^{N} m f_m(X_m) \prod_{j=n}^{m-1} (1 - f_j(X_j)), \tag{3}$$

which defines a stopping time τ_n in $\mathcal{T}_n$.

1.2 Confidence Interval of the Optimal Value

After training the stopping decision function f on the neural network, we can get the output of the optimal stopping time τ (3). Furthermore, the optimal value $V_0 = \sup_{\tau \in \mathcal{T}} E[g(\tau, X_\tau)]$ can be estimated by generating the confidence interval of $E[g(\tau, X_\tau)]$. Recall that we already simulated K_p sample paths for stochastic process $\{X_n^k\}_{n=0}^N$ for $k = 0,1,\ldots,K_p$. By the law the large numbers, the consistent estimator for $E[g(\tau, X_\tau)]$ is

$$\hat{V} = \frac{1}{K_p} \sum_{k=1}^{K_p} g(\tau_k, X_{\tau_k}^k). \tag{4}$$

And then by the central limit theorem, $\hat{V}$ follows the normal distribution where the sample variance is

$$\hat{\sigma}^2 = \frac{1}{K_p - 1} \sum_{k=1}^{K_p} \left(g(\tau_k, X_{\tau_k}^k) - \hat{V} \right)^2. \tag{5}$$

According to the results above, we can create the $1-\alpha$ confidence interval for $E[g(\tau, X_\tau)]$ which is

$$\left[\hat{V} - z_{\alpha/2} \frac{\hat{\sigma}}{\sqrt{K_p}}, \hat{V} + z_{\alpha/2} \frac{\hat{\sigma}}{\sqrt{K_p}} \right], \tag{6}$$

where $z_{\alpha/2}$ is the $\alpha/2$ quantile of standard normal distribution.

2 Hedging

In our hedging instrument, there are one bank account and $e \in \mathbb{N}$ financial assets. Given a fixed number M, A time grid is established with points $0 = u_1 < u_2 < \cdots < u_{NM}$ such that $u_{nM} = t_n$ for all n. To be more specific, t_n indicates the specific times when the hedging strategy will be evaluated or adjusted, such as the end of the trading day. While u_m represents the finer, more frequently points in time when the process X is observed or could be observed. This could be every second, every minute, or at any discrete or continuous interval.

On this time grid, we define the filtration $\mathbb{H} = (\mathcal{H}_m)_{m=0}^{NM}$ such that $\mathcal{H}_{nM} = \mathcal{F}_n$ and the following $\mathbb{H}$-Markov Process $(Y_m)_{m=0}^{NM}$ such that $Y_{nM} = X_n$ for all n. The discounted value of a Markov process at time u_m is $P_{u_m} = p_m(Y_m)$ with measurable function $p_m : \mathbb{R}^d \to \mathbb{R}^e$. A sequence $h = (h_m)_{m=0}^{NM-1}$ with measurable function $h_m : \mathbb{R}^d \to \mathbb{R}^e$ characterizes a hedging strategy, in specific holdings in $P_{u_m}^1, \ldots, P_{u_m}^e$. To make the

hedging strategy self-financing, money is dynamically borrowed from or deposited in bank account, then the portfolio at time u_m is

$$(h \cdot P)_{u_m} = \sum_{j=0}^{m-1} h_j(Y_j) \cdot (P_{j+1}(Y_{j+1} - P_j(Y_j)) = \sum_{j=0}^{m-1} \sum_{i=1}^{e} h_j^i(Y_j) \cdot (P_{j+1}^i(Y_{j+1} - P_j^i(Y_j)). \tag{7}$$

We choose the gap between two data point $\Delta = t_{n+1} - t_n$ as a day. First, we construct a hedging from time 0 to t_1, then we can compute the hedging until time t_2 and so on.

There are two ways to generate hedging strategy. One way is to directly minimize the MSE between the total hedging portfolio value and the actual payoff at time t_1. The value of option at time t_1 is $V_{t_1}^{\theta_1} = v^{\theta_1}(X_1)$ with $v^{\theta_1}(x) = g(t_1, x) \vee c^{\theta_1}(x)$, where $c^{\theta_1}(x) : \mathbb{R}^d \to \mathbb{R}$ minimizes the mean squared distance $E[(g(t_2, x) - c^{\theta_1}(x))^2]$. In order to determine the hedging position $h_m, m = 0, 1, \ldots, M-1$, we try to minimize the mean square error

$$E\left[\left(\hat{V} + (h \cdot P)_{t_1} - V_{t_1}^{\theta_1}\right)^2\right]. \tag{8}$$

In order to do that, we approximate h_m with neural network $h_\lambda : \mathbb{R}^d \to \mathbb{R}^e$ with the same form in the pricing and our aim is trying to find parameters $\lambda_0, \ldots, \lambda_{M-1}$ which minimize

$$\sum_{k=1}^{K_H} \left(\hat{V} + \sum_{m=0}^{M-1} h_{\lambda_m}(y_m^k) \cdot (p_{m+1}(y_{m+1}^k) - p_m(y_m^k)) - v^{\theta_1}(y_M^k) \right)^2. \tag{9}$$

Another way is delta-hedging. Same to the pricing we did before, first we simulate all the price paths and determine the option value V^* at each optimal stopping point τ^* for each path. There are also two ways to calculate the delta. We would use the option values calculated at different prices to apply the finite difference method. For the stock price S^*, choose two points close to S^*, $S^* + \Delta S^*$ and $S^* - \Delta S^*$, where ΔS^* is a small change in the stock price. Using neural network to approximate pricing model

$$V = E\left[e^{r\frac{nT}{N}}(\max_{1 \le i \le d} S_n^i - K)^+\right], \tag{10}$$

where S is stock price and V is option value. Then apply the finite difference method for the derivative

$$\Delta = \frac{V(S^* + \Delta S^*) - V(S^* - \Delta S^*)}{2\Delta S^*}. \tag{11}$$

Or we could construct a neural network with the delta output, using the loss function

$$L(\theta_1) = \sum_{n=0}^{N-1} \frac{(\Delta S_{n+1} Y_{n+1}^{\theta_1} - \Delta V_{n+1})^2}{T}, \tag{12}$$

where Y^{θ_1} is the delta output.

3 Neural Networks

3.1 Optimal stopping Strategy

We want to find an optimal stopping strategy for the Markov process $(X_n)_{n=0}^N$ based on a sequence of functions $\{f_n(X_n)\}_{n=0}^N, f_n : \mathbf{R}^d \to \{0, 1\}$. Constructing a sequence of neural networks of the form $f^{\theta_n} : \mathbf{R}^d \to \{0, 1\}$ with parameters $\theta_n \in \mathbf{R}^q$, to approximate f_n. So that τ_{n+1} can be approximated by

$$\sum_{k=n}^{N} k f^{\theta_k}(X_k) \prod_{j=n}^{k-1} (1 - f^{\theta_j}(X_j)), \tag{13}$$

We train the parameters via a multi-layer, feed-forward neural network that outputs probabilities in the interval $(0,1)$ and transform the results into $0-1$ stopping decisions. For $\theta \in \{\theta_0, ...\theta_N\}$ we introduce $F^\theta : \mathbf{R}^d \to (0,1)$ which is a neural network of the form:

$$F^\theta = \varphi \circ a_I^\theta \circ \varphi_{q_{I-1}} \circ a_{I-1}^\theta \circ ... \circ \varphi_{q_1} \circ a_1^\theta, \tag{14}$$

where

- $I, q_1, q_2, \ldots, q_{I-1}$ are positive integers specifying the depth of the network and the number of nodes in the hidden layers (if there are any).
- $a_1^\theta : \mathbf{R}^d \to \mathbf{R}^{q_1}, \ldots, a_{I-1}^\theta : \mathbf{R}^{q_{I-2}} \to \mathbf{R}^{q_{I-1}}$ and $a_I^\theta : \mathbf{R}^{q_{I-1}} \to \mathbf{R}$ are affine functions of the form:

$$a_i^\theta(x) = A_i x + b_i, i = 1, \ldots, I, \tag{15}$$

  where the components of the parameter $\theta \in \mathbf{R}^q$ of F^θ consist of the entries of the matrices $A_1 \in \mathbf{R}^{q_1 \times d}, \ldots, A_{I-1} \in \mathbf{R}^{q_{I-1} \times q_{I-2}}, A_I \in \mathbf{R}^{1 \times q_{I-1}}$ and the vectors $b_1 \in \mathbf{R}^{q_1}, \ldots, b_{I-1} \in \mathbf{R}^{q_{I-1}}, b_I \in \mathbf{R}$.

  So the dimension of the parameter space is :

$$q = \begin{cases} d+1 & if I = 1 \\ 1 + q_1 + \ldots + q_{I-1} + dq_1 + \ldots + q_{I-2}q_{I-1} + q_{I-1} & if I \geq 2 \end{cases} \tag{16}$$

- For $j \in \mathbf{N}$, $\varphi_j : \mathbf{R}^j \to \mathbf{R}^j$ is the component-wise ReLU activation function given by $\varphi_j(x_1, \ldots, x_j) = (x_1^+, \ldots, x_j^+)$.
- $\varphi : \mathbf{R} \to (0,1)$ is the standard logistic function $\varphi(x) = e^x/(1+e^x)$.

Hence at each time step, we are aiming to find the function f^θ that maximizes

$$E[g(n, X_n)f(X_n) + g(\tau_{n+1}, X_{\tau_{n+1}})(1 - f(X_n))] \tag{17}$$

3.2 Estimate Optimal Value V

When there are K_p independent paths of Markov process $(X_n)_{n=0}^N$, the sample paths are denoted $(x_n^k)_{n=0}^N$ for $k = 1, \ldots, K_p$. Since we must stop at time N, we set $f^{\theta_N}(x_N^k) \equiv 1, \forall m$ and back recursively compute θ_n for $n = N-1, \ldots, 0$.

At time step n along path m, the realized reward is:

$$r_n^k(\theta_n) = g(n, X_n^k)F^\theta(X_n^k) + g(\bar{\tau}_{n+1}^k, X_{\bar{\tau}_{n+1}^k}^k) \tag{18}$$

(17) can be approximated by

$$\frac{1}{K_p} \sum_{k=1}^{K_p} r_n^k(\theta_n) \tag{19}$$

Thus the reward function (19) can be optimized via a gradient descent algorithm with respect to θ_n.

Once at time zero, $n = 0$, the current optimal stopping time:

$$\bar{\tau}^k = \sum_{n=1}^{N} n f^{\theta_n}(X_k^n) \prod_{j=1}^{n-1} (1 - f^{\theta_j}(X_j^k)) \tag{20}$$

The corresponding estimate of $\mathbf{E}g(\hat{\tau}, X_{\hat{\tau}})$ is:

$$\hat{V} = \frac{1}{K_p} \sum_{k=1}^{K_p} g(\hat{\tau}_k, y_{\hat{\tau}_k}^k) \tag{21}$$

4 Our Models

4.1 MLP

A multi-layer Perception (MLP) also called feedforward neural network is a type of artificial neural network. It consists of multiple layers of neurons, including an input layer, one or more hidden layers, and an output layer. Each neuron in a layer is connected to all neurons in the subsequent layer. They are connected by applying an activation function on the neuron associated weight and bias. This function is crucial as it introduces non-linearity into the model, enabling the MLP to learn more complex patterns in the data. Common activation functions include: ReLU, Sigmoid, Tanh. Then MLPs use backpropagation for learning to adjust the weights based on the loss function we define. They are particularly good at handling complex patterns and non-linear data, making them useful for tasks such as classification, regression, and even option pricing in financial analyses.

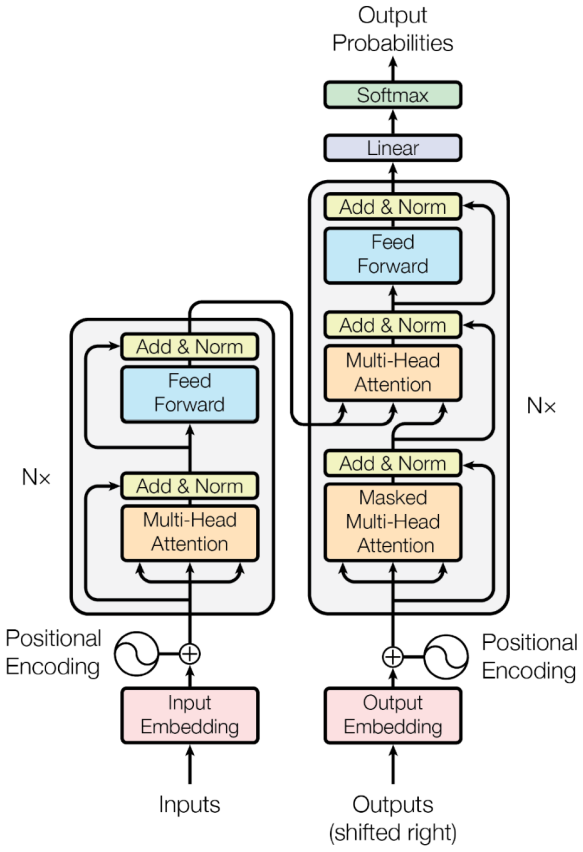


Figure 1: Transformer

4.2 Transformer

The Transformer architecture follows an encoder-decoder structure which can produce output of variable length depending on the input data. The Transformer encoder's self-attention mechanism calculates the output C using the formula $\mathrm{MA}(\cdot)$ with $Q = K = V = X$, where Q, K, and V represent the queries, keys, and values matrices respectively, and $\mathbf{X}$ is the input matrix with positional encoding applied. The dimensionality d_k and d_v is typically d/h, with h being the number of attention heads. Each encoder layer also includes a two-part feedforward neural network with ReLU and linear transformations. Layer normalization is applied

after both the self-attention and the feedforward network. The encoder is a stack of such layers, usually six in total, and it produces a separate output for each input word. The Transformer decoder architecture is formulated as follows: The self-attention mechanism computes $C_1 = \text{MA}(\cdot)$ where $Q = K = V = Y$, and Y is the input with positional encoding applied. Layer normalization is then applied to the self-attention output: $\text{LayerNorm}(C_1 + Y)$. This is followed by cross-attention, $C_2 = \text{MA}(\cdot)$ with $Q = Y$, $K = V = X$, where X is the encoder output. Another layer normalization follows: $\text{LayerNorm}(C_2 + \text{output of the previous sublayer})$. The decoder then applies a two-layer feedforward neural network (FNN) to each position, followed by layer normalization on the FNN output. These processes are encapsulated within each decoder layer, and multiple such layers are stacked to construct the complete decoder unit.

4.3 CNN

In a Convolutional Neural Network (CNN), data is processed as a 3D array with dimensions for height, width, and channels. The convolution operation performed by filters or kernels is defined as:

$$(I * K)(i, j) = \sum_c \sum_m \sum_n I_c(m, n) \cdot K_c(i - m, j - n), \tag{22}$$

where I is the input, K is the filter, i, j index spatial coordinates, m, n index the filter size, and c indexes the channels. This convolution is applied uniformly across the spatial domain, leveraging weight sharing to reduce parameter count and enhance feature extraction efficiency. As the network layers deepen, each neuron's receptive field expands, enabling the capture of increasingly complex features through a process known as hierarchical feature engineering. This structure makes CNNs highly efficient for extracting features from a dataset and while they are normally used in a 2-dimensional space for image processing, they could be used in a 1-dimensional space for time series.

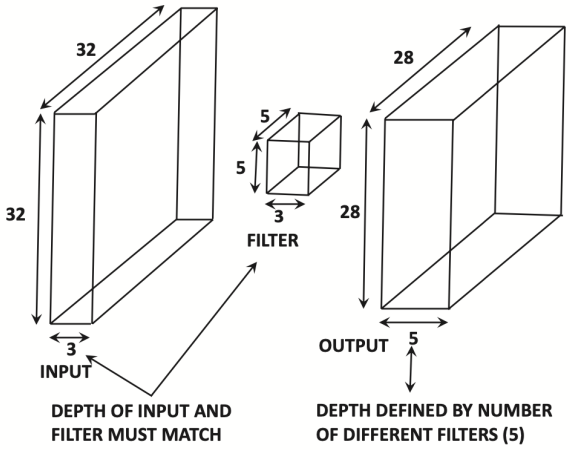


Figure 2: Convolutional Neural Network

5 Experiments

Firstly, we delve into a case study involving a Bermudan max-call option, employing our own Python implementation based on the methodology outlined in [Becker et al., 2018]. Subsequently, we examine another scenario centered around optimizing the stopping time of a fractional Brownian motion.

5.1 Dataset

Geometric Brownian motion (GBM) is a mathematical model widely used to describe the behavior of stock prices. In our setting, we use this model to simulate the stochastic dynamics of asset prices. It is represented by the stochastic differential equation and the initial condition:

$$\begin{cases} dS_t = \mu S_t dt + \sigma S_t dW_t, \\ S_0 = s_0. \end{cases} \tag{23}$$

where μ represents the drift and σ represents the volatility of the stock. The drift indicates the direction and the rate of the expected value of the stock price going up or down over the time. If μ is positive, the average trend of the stock price is upward, and if μ is negative, it trends downward. The volatility reflects the degree of randomness in the price fluctuations. A higher σ means more uncertainty in price, whereas a lower σ indicates less uncertainty in price. Assuming that we are in the no-arbitrage environment, then drift μ is replaced by the risk-free rate r. After solving this SDE, the equation for stock price S_t at time t is

$$S_t = s_0 \exp\left(r - \frac{1}{2}\sigma^2 t + \sigma\omega_t\right) \tag{24}$$

In this report, we utilize GBM to simulate stock price dynamics, denoted as S, with a data dimension of $(N = 9, K_{\mathrm{p}} = 6000, d = 10)$. This simulated data serves as an input for a neural network used to develop an optimal stopping strategy. Here, N represents the number of time steps, K_{p} denotes the number of paths, and d signifies the number of financial securities involved. During our numerical experiments, we establish a baseline model with a constant stock price, which is generated using the following parameters: time $T = 3$, strike price $K = 100$, volatility $\sigma = 0.2$, dividend rate $\delta = 0.1$, initial stock price $S_0 = 90$, and risk-free interest rate $r = 0.05$.

5.2 Model Setup

The stock price data, denoted as $\{X_n\}_{n=0}^{N}$, has a dimensionality of $(N = 9, K_{\mathrm{p}} = 6000, d = 10)$. Our objective is to train a sequence function $\{f_n(X_n)\}_{n=0}^{N}$, where $f_n : \mathbf{R}^d \to \{0, 1\}$, that maximizes $\mathbf{E}[g(\hat{\tau}, X_{\hat{\tau}})]$, which can be approximated by (21).

To achieve this goal, we initialize $f_N = 1$, and then train $\{f_n(X_n)\}_{n=0}^{N-1}$ separately from $n = N - 1$ to 0, and concatenate the models at the end.

During each individual model training process, our input data has dimensions (K_{p}, d), where we interpret K_{p} observations, each with d features. This optimization problem aims to find a strategy that maximizes value. In other words, at each time step, we aim to minimize the negative estimated expected reward for that timestep, which in our case is the negative of (19). Therefore, we define it as our loss function.

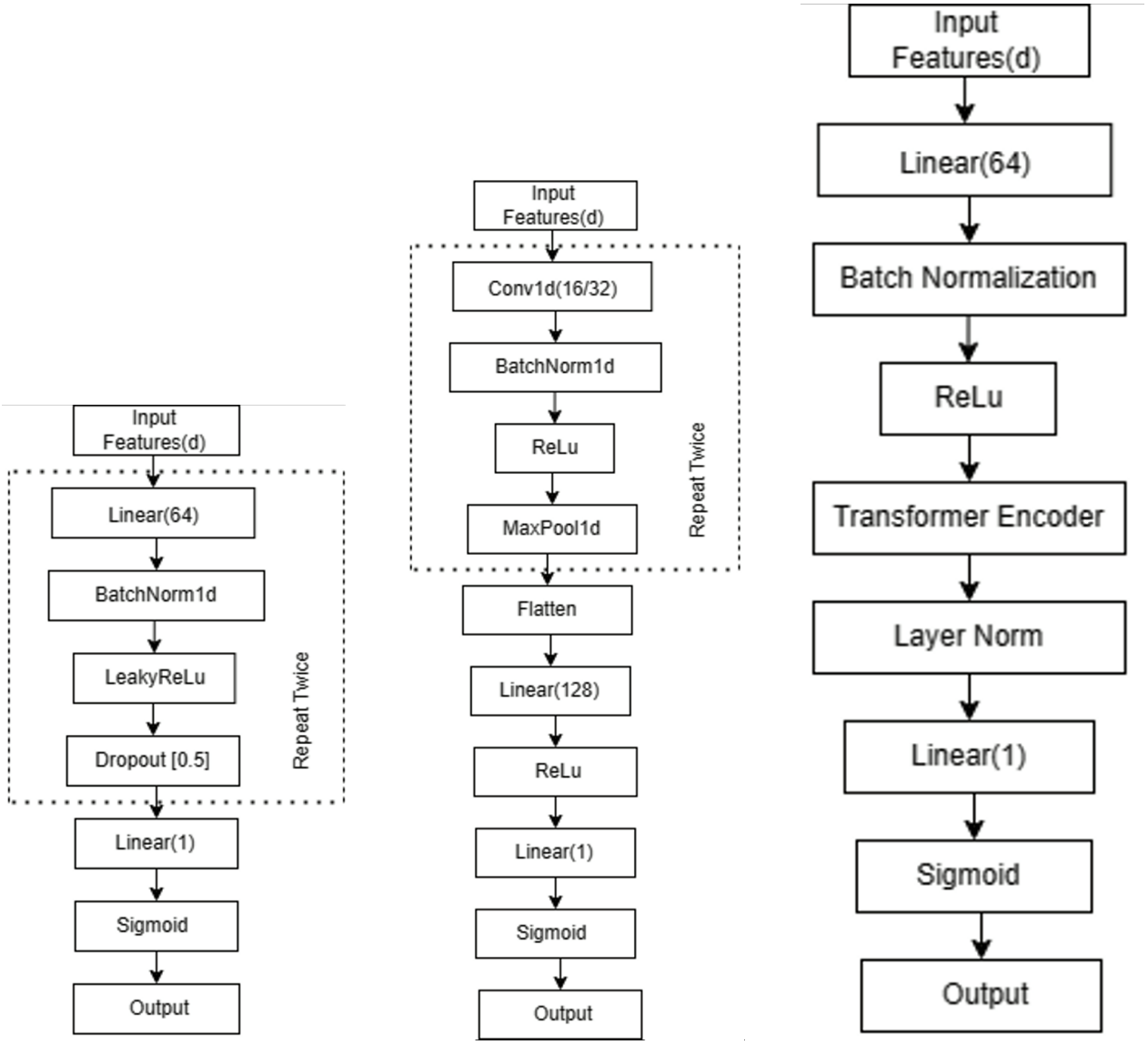
$$\frac{-1}{K_p} \sum_{k=1}^{K_p} g(n, X_n^k) f^\theta(X_n^k) + g(\bar{\tau}_{n+1}^k, X_{\bar{\tau}_{n+1}^k}^k) \tag{25}$$

Table 1: Hyperparameters for MLP, CNN, Transformer

	Threshold	Learning Rate	$\hat{V}$
MLP	0.4354	0.08697	27.7584
CNN	0.4354	0.08697	27.9600
Transformer	0.8096	0.00167	27.9901

We implemented the proposed method using PyTorch on CPU. Utilizing deep learning models involves tuning dozens of hyperparameters related to the model structure and optimizer. To efficiently explore this

hyperparameter space, we employed random cross-validation. In our setup, we defined a parameter space comprising the threshold (ϵ) and learning rate. These parameters were randomly sampled within specified ranges to construct various parameter combinations. Specifically, we uniformly sampled the threshold (ϵ) from the range $[0, 1]$, while the learning rate was generated on a logarithmic scale within the range $[0.001, 0.1]$. Through 10 iterations of random sampling cross-validation, we selected the optimal parameter configuration by comparing the estimated optimal value $\hat{V}$. The best hyperparameters we discovered, along with their corresponding estimated optimal value $\hat{V}$, are summarized in Table 1.



(a) MLP (b) CNN (c) Transformer

Figure 3: Workflow: the number in parentheses represents the output dimension of that specific layer, the number in square brackets represents the parameter of that layer.

Using the parameters that we trained our models (MLP, 1D-CNN, Transformer), the workflows are illustrated in Figure 3. The input data for the three models have the same dimensionality of (K_p, d).

In Section 4.1, we elucidated how MLP can find an optimal stopping strategy. In the numerical experiment section, we configured two linear layers for the MLP. The first layer transforms the input dimension from d to 64, while the second layer does not alter the data dimension. Each linear layer is followed by a batch

normalization layer and a dropout layer with a drop rate of 0.5, aimed at mitigating gradient vanishing. We also employ Leaky ReLU as the activation function. Finally, we use a linear layer with only one neuron to transform the data to output dimension 1, and apply the sigmoid function to scale it to the range 0 to 1, representing the probability.

The 1D-CNN model comprises two convolutional layers, each extracting 16 and 32 feature maps from the input data, respectively, using a kernel size of 3, a stride of 1, and padding of 1. Following each convolutional layer, batch normalization and ReLU activation functions are applied. Subsequently, max-pooling operations with a size of 2, stride of 2, and zero padding are performed to reduce the spatial dimension of the data while retaining important information. Finally, a flatten layer is employed to transform the data into a 1D vector, after which the operations remain consistent with those of the MLP model.

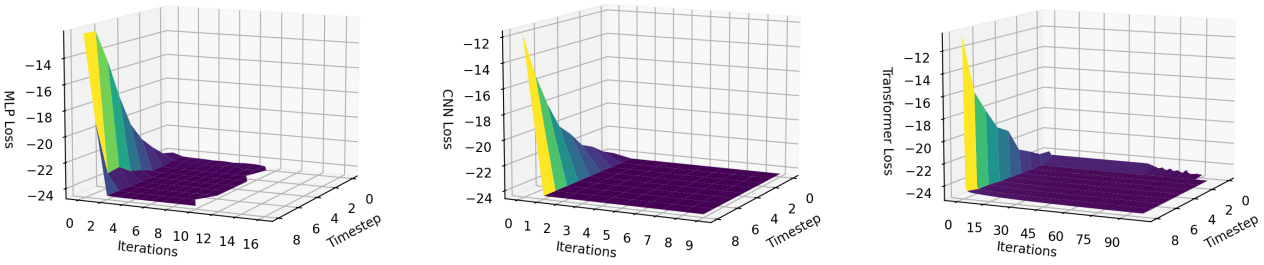
The transformer model differs from the MLP in that it includes a transformer encoder and layer normalization layer. The transformer encoder, a key component of the transformer architecture, allows the model to capture complex dependencies within the data, facilitating better feature extraction and representation learning. Additionally, the layer normalization layer is employed to stabilize the training process by normalizing the activations of each layer. These components enhance the model's ability to process sequential data and extract meaningful features.

We split the dataset into three parts: train, validation, and test, each containing 64%, 16%, and 20% of the data, respectively. During the training process, we utilize an early stopping strategy, which halts the training process if the validation loss remains non-decreasing for 10 epochs.

5.3 Performance

The performance of these models was measured in terms of the loss reduction over iterations at each timesteps in figure 4. The experimental results indicate that all three neural network architectures are capable of learning and improving their pricing approximations. The CNN demonstrated a rapid initial decrease in loss, potentially suggesting its suitability for tasks where early convergence is critical. The MLP also showed a promising start with a similar decline in loss, which indicates its capability in capturing relevant features quickly.

In contrast, the Transformer Network exhibited a more gradual but consistent decrease in loss, suggesting that it may be learning more complex dependencies within the data. This could be particularly advantageous in scenarios where the pricing structure is intricate and requires a nuanced understanding of long-range dependencies or sequential data.



(a) MLP (b) CNN (c) Transformer

Figure 4: Training Loss over iterations at each timestep for MLP, CNN and Transformer

In the report, we employ $\hat{V}$ as the primary evaluation metric for the models. Since $\hat{V}$ is assessed on the test dataset, it can effectively represent a model's generalization ability. Our objective is to discover a strategy that maximizes $\mathbf{E}[g(\tau, X_\tau)]$, which is approximated by $\hat{V}$. Additionally, through $\hat{V}$, we can derive confidence intervals for $\mathbf{E}[g(\tau, X_\tau)]$, providing insight into the robustness of the model.

Table 2 illustrates the average training and validation losses, computational efficiency, $\hat{V}$, and the corresponding confidence interval.

According to the table, the MLP was the most time-efficient model during training, with an average time

Table 2: Model performance comparison

	MLP	CNN	Transformer
Average Training Time	0.0771	0.5176	0.1569
Average Training Loss	-24.5314	-24.3638	-24.7232
Average Validation Loss	-22.9279	-23.7938	-24.0966
Average Validation Time	0.0044	0.00969	0.0609
$\hat{V}$	26.2216	27.6445	27.7811
2.5th Percentile of $E[g(\tau, X_t)]$	24.8683	26.1646	26.3672
97.5th Percentile of $E[g(\tau, X_t)]$	27.5749	29.1244	29.1949

of 0.0771 seconds. The CNN was the most time-consuming during training. However, in terms of average training loss, the Transformer outperformed all other models, suggesting it was able to fit the training data most effectively. Despite the Transformer's higher training loss, it achieved the best validation loss, indicating a stronger generalization to unseen data. The CNN closely followed, which also displayed robust generalization performance.

In terms of computational efficiency during validation, the MLP was again the fastest, whereas the Transformer was substantially slower. This could be a consideration for real-time applications where inference speed is critical.

When examining the valuation metric $\hat{V}$, which represent the overall evaluation of the models, the Transformer model achieved the highest value, closely followed by the CNN. This indicates that both models are capable of producing more accurate and robust pricing predictions.

The statistical robustness assessed by the confidence interval of $\mathbf{E}g(\tau, X_\tau)$ presumably reflecting the confidence interval of the expected payoff—was highest for the Transformer and CNN models, suggesting that their pricing approximations are more reliable.

Conclusively, the Transformer model stands out as the most capable in terms of loss metrics and robustness but at the cost of computational efficiency. The MLP, while not matching the performance metrics of the Transformer, offers a compelling alternative given its superior speed, both during training and validation. The CNN represents a balanced option, demonstrating good performance and moderate computational demands.

Table 3: Estimated Optimal Value with Different Random Seed

	MLP	CNN	Transformeer
Seed=21	27.6615	27.6183	27.7287
Seed=42	26.2216	27.6445	27.7811
Seed=63	28.8233	28.8233	28.8233
Seed=84	28.3642	28.4518	28.3910

Based on table 3, it is evident that different random seeds lead to varying estimates of the optimal value. While the MLP model exhibits considerable fluctuations in estimated optimal values across different seeds, the CNN and Transformer models demonstrate greater stability. Specifically, the MLP model shows sensitivity to initial randomization, with significant variations observed between seeds, indicating lower stability. In contrast, both the CNN and Transformer models display consistent trends across different seeds, with minimal fluctuations in estimated optimal values. Consequently, in terms of stability, the CNN and Transformer models appear to be more reliable compared to the MLP model.

6 Difficulties and Suggestions

In the report, the decision to model each timestep individually and create a sequence model instead of building a single comprehensive model is based on recognizing the temporal dependencies inherent in the data. By treating each timestep separately, the model can effectively capture the nuanced variations and

evolving patterns over time, thereby improving its capacity for optimization. Specifically, the loss function (refer to Equation 25) at each timestep is determined by information available after that particular timestep, making it uniquely tailored to the characteristics of each timestep.

Here we consider graph neural network(GNN) :

Instead of traditional machine learning methods which use graphic structured data to a matrix or vector, Graph Neural Networks (GNNs) are designed to preserve the graph structured nature by utilizing nodes to represent entities and edges to depict relationships. They operate through a message-passing mechanism where nodes aggregate and update information from their neighbors, leveraging neural networks to refine node states. Given a graph $G = (V, E)$ where V is the set of nodes and E is the set of edges, each node v in the graph updates its state by the following process:

$$h_v^{(l+1)} = f\left(h_v^{(l)}, g\left(\{h_u^{(l)} \mid u \in \mathcal{N}(v)\}\right)\right) \tag{26}$$

Here, $h_v^{(l)}$ denotes the state of node v at layer l, and f and g are differentiable function mappings, typically implemented as neural networks. The function g is an aggregation function (e.g., sum, mean, or max), which combines the features of the neighboring nodes, and f is an update function that integrates the node's previous state with the aggregated information to generate a new state. GNNs can model the relationships and interdependencies between different stocks or assets in a portfolio. By capturing these complex relationships,it can help optimize portfolio allocations to minimize risk and maximize returns.

Considering each stock price path as a node in a graph and defining their relationships based on various factors such as similarity, correlation, or common patterns in price fluctuations among stocks. This method allows for the representation of complete paths rather than individual time points, capturing complex dependencies and patterns among stocks, potentially leading to better insight into the underlying dynamics and patterns of the stock market.

However, in this report, we only try a simplified GNN approach, that is remain the basic structure of our baseline model, and considering a single time point. Each node would represent the price of a particular stock at specific time point. The relationships between nodes are random setted. Not surprisingly, the results of this experiment is not promising.

Test	Statistics
$\hat{V}$	6.0087
Confidence Interval	(5.6611,6.3563)
Average Training Time	0.0065
Average Training Loss	-16.0687
Average Validation Time	-15.9889
Average Validation Loss	0.0071
Seed=21	5.6768
Seed=42	6.0087
Seed=63	5.9694
Seed=84	6.2572

Table 4: GNN Experiment Results

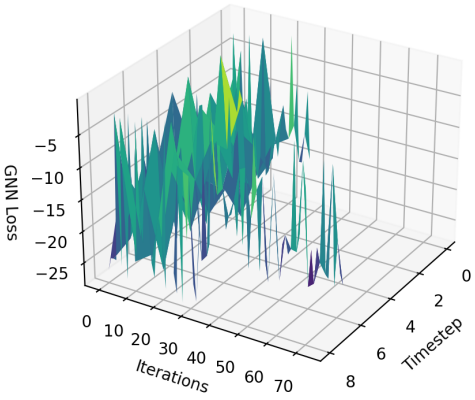


Figure 5: GNN Loss over Iterations at each timestep

Table 4 and Figure 5 indicate that the GNN appears to be the least suitable model for this pricing approximation task based on the provided metrics. Although its computational efficiency was high, both its training and validation losses were significantly higher than those of the other models, and it exhibited a much lower $\hat{V}$.

The GNN may not be suitable for the pricing approximation task due to several factors. Despite each node representing a path and possessing interconnections, the complexity of pricing data may not be adequately captured by this simplified representation. Pricing tasks typically involve intricate relationships between

multiple variables, necessitating richer feature representations than the d-dimensional features assigned to each node in the GNN. Additionally, while GNNs are theoretically applicable to graph data, the structure of pricing data may not align seamlessly with the graph structures that GNNs are designed to handle, potentially leading to suboptimal performance.

However, considering each stock price path as a node, rather than a specific time, and employing a more comprehensive relationship method during the experiment could still make GNN worthy of consideration. Therefore, further exploration and refinement of data representation, model architecture, and parameter tuning may be necessary to enhance the GNN's suitability for the task.

7 Experiments for the parameters

7.1 The Volatility

The experiment conducted assessed the neural network's performance in pricing approximation under varying levels of volatility, represented by sigma values σ ranging from 0.1 to 1. The valuation metric, denoted as $\hat{V}$, increased with the rise in σ, suggesting a positive correlation between the model's output and the level of input volatility.

At lower volatility $\sigma = 0.1$, the model predicted pricing at approximately 3.996. As σ increased, $\hat{V}$ values rose, indicating the model's recognition of higher risk and dynamic pricing environments. An anomaly was observed between $\sigma = 0.6$ and $\sigma = 0.7$, where $\hat{V}$ decreased slightly, warranting further investigation. The substantial increase at higher sigma levels implies heightened sensitivity of the model to volatility, though the decrease at $\sigma = 1$ suggests a potential non-linear relationship.

The results demonstrate the importance of volatility in pricing models and the capability of neural networks to capture these dynamics, despite irregularities which require additional examination. Such findings can inform risk management and pricing strategies in volatile financial markets.

7.2 The Risk-Free Rate

This study aimed to utilize a neural network to approximate pricing under different levels of the risk-free rate r, ranging from 0.01 to 0.145. The results indicated a clear upward trend in the estimated valuation $\hat{V}$ as r increased.

Commencing at $r = 0.01$, the valuation was approximately 15.97, correlating with a lower expected return on risk-free assets. With a rise in r, $\hat{V}$ consistently increased, reaching approximately 68.04 at $r = 0.145$. The network's sensitivity to the risk-free rate reflects an understanding of the direct relationship between r and asset pricing.

The neural network's outputs align with financial theory, where a higher risk-free rate suggests higher present values in discounted cash flow models. The model's performance confirms its ability to capture the fundamental dynamics between the risk-free rate and valuation, proving useful for predicting changes in valuation due to fluctuations in r, and serving as a tool for investment decisions and risk management.

7.3 Enhance for the parameters space

In our paper, we introduced a novel neural network framework that effectively approximates the value of Bermudan max-call options using a minimal set of parameters—time and stock price(1) . Moving forward, we can investigate the potential of expanding our model's parameter space to include additional variables commonly used in option pricing, such as moneyness, maturities, and strike prices. The future research could compare the performance of our streamlined model against traditional models that employ a wider range of parameters, and also evaluate the trade-offs between model complexity and accuracy, computational efficiency, and the adaptability of our approach to varying market conditions.

8 Conclusion

The Transformer model emerges as the most proficient option for optimal stopping tasks among the models evaluated in this study. Despite its superior performance in terms of both performance metrics and robustness, its computational efficiency poses a concern for real-time applications. Conversely, the MLP offers a compelling alternative due to its speed, albeit with slightly lower performance levels compared to the Transformer. The CNN stands as a balanced option, demonstrating competitive performance while maintaining moderate computational demands.

While Graph Neural Networks (GNNs) were explored as a potential approach, the simplistic method employed in this study, where each stock price path is represented as a node with randomly set relationships, yielded suboptimal results. This suggests that the complexity of pricing data may not be adequately captured by this simplistic representation. Further exploration and refinement of data representation, model architecture, and parameter tuning may be necessary to unlock the full potential of GNNs in pricing approximation tasks.

Overall, the findings underscore the importance of considering trade-offs between model performance, computational efficiency, and robustness when selecting the most suitable model for optimal stopping tasks in financial markets. Continued research and experimentation with different model architectures and techniques are crucial for advancing the state-of-the-art in this domain.

9 Code availability

$https://github.com/Mukkke/Deep\_optimal\_stopping\_hedging.git$