Machine learning in Finance Lab Report

Honglin Qian, honglin6@illinois.edu
Yixuan Huo, yhuo9@illinois.edu
Junhong Huang, jh136@illinois.edu
Bohan Li, bohanli5@illinois.edu

2024-12-05

Introduction

The pricing of derivatives is a cornerstone of modern financial markets, enabling efficient risk management, informed decision-making, and fair valuation across a wide range of instruments. Accurate pricing models are essential for hedging, portfolio management, and speculative trading. However, American options, with their embedded early exercise feature, introduce complexities that make traditional pricing methods computationally intensive. The overarching goal is to evaluate the computational efficiency and pricing accuracy of neural networks relative to traditional tree-based models, advancing the field of derivative pricing.

To address the computational challenges inherent in training neural networks with real-world data, we propose a novel approach that leverages synthetic data generated using the Heston stochastic volatility model. This method ensures a controlled environment for experimentation and facilitates an efficient exploration of the NN model. The input to the neural network comprises two components: (1) basic option features, including the moneyness, strike price, risk-free rate, time to maturity, and dividend yield; and (2) Heston model parameters, including long-term variance θ, mean reversion speed κ, volatility of volatility σ, correlation between asset price and variance ρ, and initial variance v_0, calibrated using market data. By bounding these parameters through Heston calibration, we enhance the accuracy and realism of the generated inputs, thereby improving the model's predictive capability.

This project explores the pricing of American options using the Heston stochastic volatility model integrated with machine learning techniques. The workflow begins with calibrating the Heston model using European option data to estimate the underlying stochastic volatility dynamics, where Heston parameters are optimized to fit historical market data. The calibrated parameters are then applied to price American options using both the NN model and traditional methods. Various calibration approaches, including cost function minimization and iterative optimization, ensure accurate replication of volatility dynamics. These parameters form the foundation for the subsequent modeling stages.

Data feature engineering is central to the project, focusing on constructing a robust dataset for model training. This step involves generating synthetic American option prices by varying key input features such as the moneyness, strike price, risk-free rate, volatility parameters, and time to maturity. The dataset also includes derived metrics like implied volatility, ensuring a comprehensive representation of the pricing landscape. Numerical methods, such as QuantLib-based finite difference schemes, are employed to validate and enhance the dataset's accuracy.

Machine learning models are deployed to approximate option prices efficiently. Tree-based models, including Random Forest, Gradient Boosted Trees, Decision Trees, and LightGBM, are benchmarked to evaluate feature importance and prediction accuracy. Additionally, deep learning models are explored for capturing the non-linear dynamics of the Heston model. Architectures such as Artificial Neural Networks (ANN), Gradient Boosted Neural Networks, Long Short-Term Memory (LSTM) networks, and Convolutional Neural Networks (CNN) are implemented. Each model is tuned for optimal performance using hyperparameter optimization techniques, ensuring high accuracy and generalization.

The project underscores the potential of combining stochastic volatility models with modern machine learning methods to achieve computational efficiency and robust pricing accuracy. This hybrid framework has significant implications for real-time financial modeling and derivative pricing.

Contents

1 Baseline Model - Finite Difference **4**
- 1.1 Finite Difference . . . 4
  - 1.1.1 Grid Discretization . . . 4
  - 1.1.2 Finite Difference Approximation . . . 4
  - 1.1.3 Boundary and Terminal Conditions . . . 5
  - 1.1.4 Early Exercise Adjustment for American Options . . . 5
  - 1.1.5 Algorithmic Summary . . . 6
- 1.2 American Option Pricing under Heston Model using Explicit Finite Difference Method . . . 6

2 Calibration **8**
- 2.1 Objective and Motivation . . . 8
  - 2.1.1 Calibration of Derivative Pricing Models . . . 8
  - 2.1.2 European Options for Calibration . . . 8
  - 2.1.3 Focus on At-the-Money (ATM) Options . . . 8
- 2.2 Data Processing . . . 8
- 2.3 Calibration Functions . . . 9
- 2.4 Differential Evolution Optimization . . . 10
- 2.5 Calibration Output . . . 10

3 Feature Input **12**
- 3.1 Data Generation . . . 12
- 3.2 Finite Difference Pricing . . . 12
- 3.3 Multiprocessing . . . 13
- 3.4 Result . . . 14

4 Machine Learning Models **15**
- 4.1 Tree Models . . . 15
  - 4.1.1 Decision Tree - `DecisionTreeRegressor` . . . 15
  - 4.1.2 Random Forest - `RandomForestRegressor` . . . 15
  - 4.1.3 Gradient Boost Decision Tree - `GradientBoostingRegressor` . . . 15
  - 4.1.4 LightGBM - `lgb.LGBMRegressor` . . . 16
  - 4.1.5 Tree Model Parameter Optimization . . . 16
  - 4.1.6 Tree Model Result and Conclusion . . . 18
- 4.2 Deep Learning Models . . . 20
  - 4.2.1 Artificial Neural Network . . . 20
  - 4.2.2 ANN Architecture and Result . . . 21
  - 4.2.3 Gradient Boost Neural Network . . . 26
  - 4.2.4 Gradient Boost Neural Network Architecture and Result . . . 27
  - 4.2.5 Long-short Term Memory Model . . . 29
  - 4.2.6 LSTM Architecture and Result . . . 30
  - 4.2.7 Convolutional Neural Networks . . . 31
  - 4.2.8 CNN Architecture and Result . . . 32

5 Results and Conclusions **35**

1 Baseline Model - Finite Difference

1.1 Finite Difference

In this part, we present the mathematical principles of the explicit finite difference method for pricing American-style options using the Black-Scholes Partial Differential Equation (PDE). The approach discretizes time and space, applies central difference approximations for spatial derivatives, and enforces early exercise conditions at each time step.

Model Setup

Consider an American option with strike price K, maturity T, and underlying asset price S. Let $V(S,t)$ be the value of the option. Under the risk-neutral measure, $V(S,t)$ satisfies the Black-Scholes PDE:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r-q)S\frac{\partial V}{\partial S} - rV = 0, \tag{1}$$

where r is the risk-free interest rate and σ is the volatility of the underlying asset.
For an American option, there is an additional constraint due to the possibility of early exercise. For an American put:

$$V(S,t) \geq \max(K-S, 0). \tag{2}$$

For an American call:

$$V(S,t) \geq \max(S-K, 0). \tag{3}$$

1.1.1 Grid Discretization

Spatial Grid

Choose a sufficiently large maximum asset price $S_{\max}$ so that at $S_{\max}$ the option value is either negligible (for puts) or easily approximated (for calls). Define the spatial step:

$$\Delta S = \frac{S_{\max}}{N}.$$

The spatial grid points are:

$$S_i = i\Delta S, \quad i = 0, 1, \ldots, N.$$

Time Grid

Divide the time interval $[0, T]$ into M equal steps:

$$\Delta t = \frac{T}{M}, \quad t_j = j\Delta t, \quad j = 0, 1, \ldots, M.$$

We know the terminal payoff at $t = T$, and we will step backwards to $t = 0$.

1.1.2 Finite Difference Approximation

Approximate the derivatives in the PDE at the grid point (S_i, t_j) as follows.

Temporal Derivative

Use a backward difference in time:

$$\frac{\partial V}{\partial t} \approx \frac{V_i^j - V_i^{j-1}}{\Delta t},$$

where $V_i^j \approx V(S_i, t_j)$.

Spatial Derivatives

Use central differences for the first and second spatial derivatives:

$$\frac{\partial V}{\partial S} \approx \frac{V_{i+1}^j - V_{i-1}^j}{2\Delta S},$$

$$\frac{\partial^2 V}{\partial S^2} \approx \frac{V_{i+1}^j - 2V_i^j + V_{i-1}^j}{(\Delta S)^2}.$$

Substituting into the PDE

The Black-Scholes PDE becomes:

$$\frac{V_i^j - V_i^{j-1}}{\Delta t} + \frac{1}{2}\sigma^2 S_i^2 \frac{V_{i+1}^j - 2V_i^j + V_{i-1}^j}{(\Delta S)^2} + rS_i \frac{V_{i+1}^j - V_{i-1}^j}{2\Delta S} - rV_i^j = 0.$$

Rearrange for V_i^{j-1}:

$$V_i^{j-1} = V_i^j - \Delta t \left[\frac{1}{2}\sigma^2 S_i^2 \frac{V_{i+1}^j - 2V_i^j + V_{i-1}^j}{(\Delta S)^2} + rS_i \frac{V_{i+1}^j - V_{i-1}^j}{2\Delta S} - rV_i^j \right].$$

This is the explicit finite difference scheme: knowing V_i^j, we can directly compute V_i^{j-1}.

1.1.3 Boundary and Terminal Conditions

Terminal Condition

At maturity $t = T$:

$$V_i^M = \max(\Phi(S_i), 0),$$

where $\Phi(S)$ is the payoff function, e.g. for a put $\Phi(S) = K - S$, for a call $\Phi(S) = S - K$.

Boundary Conditions

At $S = 0$, for a put $V(0,t) \approx K$, and for a call $V(0,t) = 0$.
At $S = S_{\max}$, for a put $V(S_{\max}, t) \approx 0$, and for a call $V(S_{\max}, t) \approx S_{\max} - Ke^{-r(T-t)}$. These conditions are applied at each time step.

1.1.4 Early Exercise Adjustment for American Options

After computing V_i^{j-1} using the explicit scheme, enforce the American early exercise condition. For an American put:

$$V_i^{j-1} \leftarrow \max(V_i^{j-1}, K - S_i).$$

For a call:

$$V_i^{j-1} \leftarrow \max(V_i^{j-1}, S_i - K).$$

This ensures that at each time step, the option value does not violate the early exercise constraint.

Stability Considerations

The explicit method can be conditionally stable and often requires a small Δt. A typical stability constraint is:

$$\Delta t \leq \frac{(\Delta S)^2}{\sigma^2 S_{\max}^2}.$$

To achieve accurate and stable results, a sufficiently fine time and space discretization must be chosen.

1.1.5 Algorithmic Summary

1. Set up the grid: choose $S_{\max}$, N, M, compute ΔS, Δt.
2. Initialize terminal values: $V_i^M = \max(\Phi(S_i), 0)$.
3. For $j = M, M-1, \ldots, 1$:
   (a) Apply boundary conditions at $S = 0$ and $S = S_{\max}$.
   (b) Compute V_i^{j-1} using the explicit scheme.
   (c) Enforce the early exercise condition:
   $$V_i^{j-1} = \max(V_i^{j-1}, \text{Immediate Exercise Value}).$$
4. After reaching $j = 0$, $V(S_0, 0)$ is the current option value (if S_0 is the current price).

1.2 American Option Pricing under Heston Model using Explicit Finite Difference Method

Heston Model PDE

The Heston stochastic volatility model is described by the following stochastic differential equations (SDEs):

$$\begin{aligned} dS_t &= \mu S_t\, dt + \sqrt{v_t} S_t\, dW_t^S, \\ dv_t &= \kappa(\theta - v_t)\, dt + \sigma\sqrt{v_t}\, dW_t^v, \\ \langle dW_t^S, dW_t^v \rangle &= \rho\, dt. \end{aligned}$$

The corresponding partial differential equation (PDE) for the option price $C(S, v, t)$ is given by:

$$\frac{\partial C}{\partial t} + \frac{1}{2} v S^2 \frac{\partial^2 C}{\partial S^2} + \rho\sigma v S \frac{\partial^2 C}{\partial S \partial v} + \frac{1}{2}\sigma^2 v \frac{\partial^2 C}{\partial v^2} + rS\frac{\partial C}{\partial S} + \kappa(\theta - v)\frac{\partial C}{\partial v} - rC = 0.$$

Finite Difference Discretization

We discretize the PDE using a finite difference grid with S, v, and t dimensions:

$$S_i = i\Delta S, \quad v_j = j\Delta v, \quad t_n = n\Delta t.$$

The derivatives are approximated as:

$$\frac{\partial C}{\partial S} \approx \frac{C_{i+1,j}^n - C_{i-1,j}^n}{2\Delta S}, \quad \frac{\partial^2 C}{\partial S^2} \approx \frac{C_{i+1,j}^n - 2C_{i,j}^n + C_{i-1,j}^n}{\Delta S^2}.$$

$$\frac{\partial C}{\partial v} \approx \frac{C_{i,j+1}^n - C_{i,j-1}^n}{2\Delta v}, \quad \frac{\partial^2 C}{\partial v^2} \approx \frac{C_{i,j+1}^n - 2C_{i,j}^n + C_{i,j-1}^n}{\Delta v^2}.$$

The time derivative is approximated as:

$$\frac{\partial C}{\partial t} \approx \frac{C_{i,j}^{n+1} - C_{i,j}^n}{\Delta t}.$$

Substituting these into the PDE yields the explicit update formula:

$$C_{i,j}^{n+1} = C_{i,j}^n + \Delta t \cdot \mathcal{L}(C_{i,j}^n),$$

where $\mathcal{L}(C)$ represents the discrete operator for the spatial derivatives.

American Option Early Exercise Condition

To account for the early exercise feature of American options, we apply the following condition at each time step:

$$C_{i,j}^n = \max\left(C_{i,j}^n, \text{Payoff}(S_i)\right),$$

where the payoff is given by $\max(K - S_i, 0)$ for a put option or $\max(S_i - K, 0)$ for a call option.

Boundary and Initial Conditions

The boundary conditions are:

$$C(0, v, t) = \max(K, 0), \quad C(S_{\max}, v, t) = 0, \quad C(S, v, T) = \max(K - S, 0).$$

Numerical Algorithm Workflow

1. **Initialize the grid and boundary conditions:**
   - **Initial value:** At expiration $t = T$, the option value is given by $C(S, v, T) = \max(K - S, 0)$.
   - **Boundary conditions:** Option value at extreme S or v.
2. **Time-stepping iteration:**
   - For each time step t_n, compute C^{n+1} using the explicit finite difference method.
   - Check if each grid point satisfies the early exercise condition, and apply corrections if necessary.
3. **Output the result:**
   - The option value $C(S_0, v_0, 0)$ at the current time $t = 0$.

2 Calibration

2.1 Objective and Motivation

2.1.1 Calibration of Derivative Pricing Models

Calibration is the process of aligning a theoretical pricing model with observed market data to ensure its accuracy and relevance. In the context of the Heston model, calibration involves determining the parameters that best replicate the observed prices of market-traded options. This step is critical because the performance of any pricing model depends on how well it reflects the actual market dynamics. Without calibration, the outputs of a pricing model may deviate significantly from real-world prices, reducing its utility in applications such as hedging and risk management.

The Heston model introduces five key parameters: long-term variance θ, mean reversion speed κ, volatility of volatility σ, correlation between asset price and variance ρ, and initial variance v_0. These parameters collectively describe the stochastic behavior of volatility and its impact on option prices. Calibration ensures that the values of these parameters are consistent with the implied volatility observed in the market.

Implied volatility serves as a bridge between observed market prices and theoretical models. It represents the market's expectations of future volatility and is inferred from the prices of traded options. Backing out implied volatility is essential because it provides a common language to compare options across strikes and maturities. In this project, implied volatility underpins the calibration of the Heston model, ensuring that the synthetic data generated for training the Neural Network accurately reflects market behavior.

2.1.2 European Options for Calibration

European options are used for Heston calibration due to their computational simplicity and shared implied volatility surface with American options. Unlike American options, European options can only be exercised at maturity, eliminating the need to account for early exercise behavior during calibration. This streamlines the computational process, allowing for faster and more efficient calibration without compromising the relevance of the resulting parameters. Importantly, the shared implied volatility surface between European and American options ensures that the calibrated parameters can be reliably applied to American options.

2.1.3 Focus on At-the-Money (ATM) Options

At-the-money (ATM) options are the focus of this calibration process due to their high sensitivity to implied volatility and their liquidity in the market. ATM options are typically the most traded and provide the clearest signal about the market's expectations of volatility. Their pricing is less influenced by bid-ask spreads or other market inefficiencies, making them a reliable foundation for calibration. Additionally, ATM implied volatility serves as a starting point for constructing the volatility surface, a critical input for the Heston model.

2.2 Data Processing

This function processes options data to prepare it for modeling, particularly for tasks such as volatility surface construction or model calibration. It filters, cleans, and structures the data to extract key inputs like spot prices, expiration dates, implied volatilities (IV), and strike prices.

Filtering Criteria:

- **Days to Expiration (DTE):** Must be greater than or equal to 365 (long-dated options only).
- **Moneyness:** Includes options with moneyness between 0.8 and 1.2 (close to at-the-money options).
- **Quote Date:** Data is filtered for a specific `QUOTE_DATE` provided as input.

- **Implied Volatility (C_IV):** Only includes rows where call IV is greater than 0.
- **Missing Data:** Rows with missing values are removed.

The function then organizes call IV data into a clean, structured format (grid) for further analysis, along with the corresponding spot price and strike prices.

```python
def data_processing(data, date):
    data['MONEYNESS'] = data['UNDERLYING_LAST']/data['STRIKE']

    filtered_data = data[
        (data['DTE'] >= 365)
        & (data['MONEYNESS'] >= 0.8)
        & (data['MONEYNESS'] <= 1.2)
        & (data['QUOTE_DATE'] == date)
        & (data['C_IV'] > 0)
        ]

    filtered_data = filtered_data.dropna()

    spot = float(filtered_data['UNDERLYING_LAST'].unique()[0])

    #step 1: Convert unique_expire_dates to QuantLib Date format
    expiration_dates = [Date(date.day, date.month, date.year) for date in pd.to_datetime(filtered_data['EXPIRE_DATE'].unique())]

    #iv_data: Each row in data is a different expiration time, and each column corresponds to various strikes as given in strikes.
    # Create a pivot table for IV data

    #step 2: Pivot table: Rows are expiration dates, columns are strikes, values are IV
    iv_data_pivot = filtered_data.pivot_table(
        index='EXPIRE_DATE',
        columns='STRIKE',
        values='C_IV',
    )

    iv_data_cleaned = iv_data_pivot.dropna(axis=1, how='any')

    iv_data = iv_data_cleaned.to_numpy()

    #step 3: choose strike
    strikes = iv_data_cleaned.columns.to_numpy()

    return spot, expiration_dates, iv_data, strikes
```

Figure 1: Illustration of Data Processing Workflow

2.3 Calibration Functions

1. `setup_helpers`

The `setup_helpers` function creates calibration tools (`HestonModelHelper`) and grid data for the Heston model. It takes inputs such as the pricing engine, expiration dates, strikes, implied volatility (IV) data, and term structures for risk-free and dividend yields. For each combination of expiration dates and strike prices, it generates a `HestonModelHelper` object, which facilitates the calibration of the model parameters to observed market data (e.g., IV). These helpers are then stored in a list for calibration, and the corresponding grid of expiration-strike pairs is stored for later analysis.

2. `cost_function_generator`

The `cost_function_generator` function creates a cost function for calibrating the Heston model. It takes the Heston model and the list of `HestonModelHelper` objects as inputs. The generated cost function adjusts the Heston model parameters, computes the calibration error for each helper, and returns either a normalized total error (for optimization) or individual errors. This allows the calibration process to minimize the difference between model prices and market prices.

3. `calibration_report`

The `calibration_report` function generates a report to evaluate the performance of the Heston model calibration. It compares the market prices with the model-predicted prices for each option in the grid, calculating relative errors and the average absolute error. If the `detailed` parameter is enabled, it prints a comprehensive table showing strikes, expiration dates, market values, model values, and relative errors for each option, providing insights into the calibration quality.

4. `setup_model`

The `setup_model` function initializes the Heston model and its pricing engine. It creates a `HestonProcess` using inputs such as risk-free and dividend yield term structures, the spot price, and initial conditions for the model parameters (θ, κ, σ, ρ, and v_0). The process is then wrapped into a `HestonModel` object, which represents the stochastic volatility dynamics. The model is paired with an `AnalyticHestonEngine` to enable pricing of options under the Heston framework. This setup forms the foundation for both calibration and option pricing.

2.4 Differential Evolution Optimization

The `calibrate_heston_model` function calibrates the parameters of the Heston stochastic volatility model $(\theta, \kappa, \sigma, \rho, v_0)$ using the Differential Evolution algorithm. The calibration process minimizes the error between market-observed option prices or implied volatilities and the model's predictions. The function begins by constructing `HestonModelHelper` objects through `setup_helpers`, which compare the model prices against market data. Next, it generates a cost function using `cost_function_generator`, which computes the calibration error. Differential Evolution is then applied to optimize the parameters within user-defined bounds over a maximum number of iterations. After optimization, the function updates the model parameters, computes the average absolute calibration error via `calibration_report`, and returns the optimized parameters along with a summary of the calibration results.

```python
def calibrate_heston_model(model,
                           engine,
                           yield_ts,
                           dividend_ts,
                           spot,
                           expiration_dates,
                           strikes,
                           iv_data,
                           calculation_date,
                           bounds,
                           max_iter=200,
                           norm=True):
    """
    Calibrate the Heston model using Differential Evolution.

    Parameters:
        model: The Heston model object to calibrate.
        engine: The engine used for pricing within the model.
        yield_ts: The yield term structure.
        dividend_ts: The dividend term structure.
        spot: The spot price of the underlying asset.
        expiration_dates: List of option expiration dates.
        strikes: List of option strike prices.
        data: Observed market option prices.
        calculation_date: The valuation date for calibration.
        bounds: Parameter bounds for the optimization [(theta_min, theta_max), ...].
        max_iter: Maximum iterations for differential evolution.
        norm: Whether to normalize the cost function or not.

    Returns:
        dict: A dictionary with calibration results:
            - 'params': Calibrated model parameters [theta, kappa, sigma, rho, v0].
            - 'error': Calibration error.
            - 'summary': Summary of the calibration process.
    """
    heston_helpers, grid_data = setup_helpers(engine, expiration_dates, strikes, iv_data, calculation_date, spot, yield_ts, dividend_ts)

    # scipy.optimize.differential_evolution 默认是以 最小化 目标函数为优化方向的。也就是说，它会尝试找到目标函数 (cost_function) 的最小值。
    cost_function = cost_function_generator(model, heston_helpers, norm=norm)

    sol = differential_evolution(cost_function, bounds, maxiter=max_iter)

    optimized_params = sol.x  # Extract optimized parameters from the solution, # theta, kappa, sigma, rho, v0 = optimized_params

    error = calibration_report(heston_helpers, grid_data)

    summary = [["SciPy DE1", error] + list(optimized_params)]

    return optimized_params, DataFrame(summary, columns=["Name", "Avg Abs Error","Theta", "Kappa", "Sigma", "Rho", "V0"], index=['']*len(summary))
```

Figure 2: Illustration of Differential Evolution Workflow

2.5 Calibration Output

The `calibration` function is designed to perform parameter calibration for the Heston stochastic volatility model. It begins by setting the calculation date (`calculation_date`), which serves as the evaluation reference point for time-dependent computations. Then, it initializes the risk-free rate and dividend yield term structures using flat forward curves. Afterward, the function sets up the Heston model and its pricing engine using `setup_model`. The calibration process is performed by invoking `calibrate_heston_model`, which minimizes the error between the model prices and market-observed implied volatilities using Differential Evolution. The function concludes by returning a summary of the

optimized parameters and the calibration error. The parameter bounds used during optimization are detailed in the table below:

Parameter	Description	Bounds
θ	Long-term variance	$[0.01, 1.0]$
κ	Mean reversion speed	$[0.1, 15]$
σ	Volatility of volatility	$[0.01, 1.0]$
ρ	Correlation between asset price and variance	$[-0.9, 0.9]$
v_0	Initial variance	$[0.1, 1.0]$

```python
def calibration(date, spot, expiration_dates, iv_data, strikes):
    # set up pricing initialization
    day_count = Actual365Fixed()

    calculation_date = Date(date.day, date.month, date.year)  # 定义了计算的基准日期。这个日期通常用于计算贴现因子和其他时间相关的金融参数。


    Settings.instance().evaluationDate = calculation_date
    dividend_yield = QuoteHandle(SimpleQuote(0.0))

    risk_free_rate = 0.05
    dividend_rate = 0.0156

    yield_ts = YieldTermStructureHandle(FlatForward(calculation_date, risk_free_rate, day_count))
    dividend_ts = YieldTermStructureHandle(FlatForward(calculation_date, dividend_rate, day_count))

    # calibration
    model, engine = setup_model(yield_ts, dividend_ts, spot)

    bounds = [(0.01, 0.5), # theta
              (0, 5), # kappa
              (0.01,1.0), # sigma
              (-0.9, 0.9),  # rho
              (0, 0.5)] # v0


    optimized_params, result = calibrate_heston_model(model, engine, yield_ts, dividend_ts, spot, expiration_dates, strikes, iv_data, calculation_date, bounds)

    return result

```

Figure 3: Illustration of calibration Workflow

The `date_list` is constructed to select the last available `QUOTE_DATE` for each month. This is achieved by grouping the dataset by month using the `dt.to_period('M')` method, which converts each date into its corresponding month period, and then applying the `.tail(1)` method to retrieve the most recent date within each group. By doing so, the `date_list` ensures that only the final `QUOTE_DATE` of each month is included. This approach is particularly useful for financial modeling, as it captures a representative snapshot of market conditions at the different month, capture the overall volatility surface in the calibration.

Name	Avg Abs Error	Theta	Kappa	Sigma	Rho	V0
SciPy DE	1.172766	0.090408	0.163527	0.311938	-0.857725	0.033519
SciPy DE	1.118656	0.068540	0.371457	0.376333	-0.873383	0.042899
SciPy DE	0.959806	0.197273	0.095271	0.385767	-0.895675	0.039161
SciPy DE	3.257900	0.245116	0.069387	0.384070	-0.887355	0.034842
SciPy DE	1.611970	0.163629	0.194981	0.580281	-0.859366	0.036494
SciPy DE	1.418260	0.093429	0.598107	0.925724	-0.723515	0.027153
SciPy DE	1.766805	0.282570	0.087387	0.411160	-0.891599	0.017256
SciPy DE	2.312549	0.064432	1.011648	0.758951	-0.844866	0.016878
SciPy DE	7.035678	0.052616	1.524657	0.884623	-0.752249	0.036571
Mean/Error Range	**2.2949388**	**[0.0526, 0.2826]**	**[0.0694, 1.5247]**	**[0.3119, 0.9257]**	**[-0.8957, -0.7235]**	**[0.0169, 0.0429]**

Table 1: Calibration Results for Heston Model Using SciPy Differential Evolution, with Error and Parameter Ranges

3 Feature Input

3.1 Data Generation

The `data_generation` function performs the following tasks:

1. **Parameter Sampling:** Random values are sampled for the following variables:
   - Strike price (K) from a uniform distribution over $[90, 110]$.
   - Time to maturity (T) from a uniform distribution over $[1/255, 1]$.
   - Heston model parameters:
     - κ (mean reversion speed),
     - ρ (correlation coefficient),
     - σ (volatility of variance),
     - θ (long-term variance),
     - v_0 (initial variance),

     using ranges defined in the provided `para_data`.

   The table below summarizes the parameters used in the `data_generation` function and their respective ranges:

Parameter	Symbol	Range	Description
Initial Stock Price	S_0	Given	Initial price of the underlying asset
Strike Price	K	[90, 110]	Uniformly sampled strike price
Time to Maturity	T	$[1/255, 1]$	Uniformly sampled time to maturity (in years)
Risk-Free Rate	r	Given	Risk-free interest rate
Dividend Yield	q	Given	Dividend yield of the underlying asset
Mean Reversion Speed	κ	$[\texttt{Kappa}_{\min}, \texttt{Kappa}_{\max}]$	Uniformly sampled mean reversion speed
Correlation Coefficient	ρ	$[\texttt{Rho}_{\min}, \texttt{Rho}_{\max}]$	Uniformly sampled correlation coefficient
Volatility of Variance	σ	$[\texttt{Sigma}_{\min}, \texttt{Sigma}_{\max}]$	Uniformly sampled volatility of variance
Long-Term Variance	θ	$[\texttt{Theta}_{\min}, \texttt{Theta}_{\max}]$	Uniformly sampled long-term variance
Initial Variance	v_0	$[\texttt{V0}_{\min}, \texttt{V0}_{\max}]$	Uniformly sampled initial variance
Option Type	–	`put/call`	Specifies whether the option is a put or call

2. **Feller Condition:** The sampled parameters are checked against the relaxed Feller condition:

$$2\kappa\theta > \sigma^2$$

   Any parameter set failing this condition is excluded.

3. **Option Value Computation:** For valid parameter sets, the value of an American option is calculated using the `american_option_finite_difference` method. This includes:

$$\text{Inputs: } S_0, K, T, r, q, \theta, \kappa, \sigma, \rho, v_0, \text{option type}$$

4. **Data Storage:** Valid results, including input parameters and the computed option value, are stored in a Pandas DataFrame. The data is saved as a CSV file in the directory:

`./data/raw_data/<option_type>/b_raw_data_part<n>.csv`

3.2 Finite Difference Pricing

The function `american_option_finite_difference` computes the price of an American option using the Heston model and the finite difference method. It performs the following steps:

- Initializes an American-style vanilla option with early exercise rights based on the given parameters: spot price (S_0), strike price (K), maturity (T), risk-free rate (r), and dividend yield (q).

- Configures the Heston model to simulate the stochastic dynamics of the underlying asset and its variance using parameters: initial variance (v_0), mean reversion speed (κ), long-term variance (θ), volatility of variance (σ), and correlation (ρ).
- Sets up a finite difference grid with predefined time, price, and variance steps and applies the Hundsdorfer numerical scheme for solving the associated partial differential equation.
- Calculates the option's net present value (NPV) using the `FdHestonVanillaEngine` in QuantLib.
- Returns the computed option price, or -1 if an error occurs during the calculation.

This function is a practical tool for pricing American options under the Heston framework using numerical methods.

```python
def american_option_finite_difference(s0, K, T, r, q, theta, kappa, sigma, rho, v0, option_type):
    '''
    s0: Spot price at t=0
    m: Moneyness
    T: Time to maturity of the option
    r: Risk-free interest rate
    q: Dividend yield of the underlying asset
    v0: Initial volatility of the Heston model
    theta: Mean reversion speed of the Heston model
    kappa: Mean reversion level of the Heston model
    sigma: Volatility of the Heston model
    rho: Correlation between the asset price and volatility processes
    option_type: Type of the option ('call' or 'put')
    '''

    ## Convert option type
    if option_type.lower() == 'call':
        option_type_ql = ql.Option.Call
    elif option_type.lower() == 'put':
        option_type_ql = ql.Option.Put
    else:
        raise ValueError("Invalid option type. Choose 'call' or 'put'.")

    # Set up the option
    today = ql.Date.todaysDate()
    expiry_date = today + ql.Period(f"{int(T * 365)}d")
    am_exercise = ql.AmericanExercise(earliestDate=today, latestDate=expiry_date)

    option = ql.VanillaOption(ql.PlainVanillaPayoff(option_type_ql, K), am_exercise)

    # Set up the Heston model
    heston_process = ql.HestonProcess(
        ql.YieldTermStructureHandle(ql.FlatForward(today, r, ql.Actual365Fixed())),
        ql.YieldTermStructureHandle(ql.FlatForward(today, q, ql.Actual365Fixed())),
        ql.QuoteHandle(ql.SimpleQuote(s0)),
        v0, kappa, theta, sigma, rho
    )

    heston_model = ql.HestonModel(heston_process)

    # Finite difference grid settings
    tGrid, xGrid, vGrid = 100, 100, 50
    dampingSteps = 0
    fdScheme = ql.FdmSchemeDesc.Hundsdorfer()

    # Calculate the option price using the Heston model with Finite Difference
    heston_engine = ql.FdHestonVanillaEngine(heston_model, tGrid, xGrid, vGrid, dampingSteps, fdScheme)
    option.setPricingEngine(heston_engine)

    try:
        return option.NPV()
    except:
        return -1.
```

Figure 4: Illustration of Finite Difference Pricing Workflow

3.3 Multiprocessing

The function `multi_processing_generation` efficiently generates large datasets for option pricing by leveraging parallel processing with multiple CPU cores. It distributes the workload across the specified number of cores, with each core independently executing the `data_generation` function using distinct parameter sets. This parallelism significantly reduces computation time, making it ideal for handling resource-intensive tasks like generating financial modeling data. The function ensures proper resource management and saves the results in separate files for each core. The key steps are as follows:

- **Parameter Setup:** The total workload is divided into smaller tasks, with each core assigned a unique task ID n.
- **Parallel Processing:** A multiprocessing pool with n_{cores} cores is created. Each core independently executes the `data_generation` function using its specific parameter set:

$$\text{Parameters: } (N, \texttt{para\_data}, S_0, r, q, \texttt{option\_type}, n)$$

- **Workload Distribution:** The `pool.map` method ensures the tasks are executed concurrently, leveraging all available cores.
- **Output Handling:** Each core saves its generated data to a separate file, identified by the unique task ID n.
- **Resource Management:** The use of the `with` block ensures that resources are properly released after processing.

This function is particularly useful for computationally intensive tasks like generating financial modeling data, as it reduces runtime by effectively utilizing parallelism.

3.4 Result

The code filters out extreme values from the finite difference computed option prices by retaining only those within the central 90% of the data distribution. Specifically:

Filter Data: Retain only the rows where the values of `option` lie within these bounds:

$$q_{0.05} \leq \texttt{option} \leq q_{0.95}.$$

This process removes the extreme 5% of values from both ends of the distribution, focusing on the central 90% of the data and reducing the influence of outliers.

The following table outlines the input fields used in the machine learning model for option pricing:

- M : The ratio of the current stock price S_0 to the strike price K, representing the relative position of the stock price to the option's strike price.
- T**:** Time to maturity (in years).
- r**:** Risk-free interest rate.
- q**:** Dividend yield of the underlying asset.
- v_0**:** Initial variance of the Heston model.
- θ**:** Long-term variance (mean reversion level).
- κ**:** Mean reversion speed of variance.
- σ**:** Volatility of variance in the Heston model.
- ρ**:** Correlation between the underlying asset price and its variance.
- **call:** Computed option price using the finite difference method.

4 Machine Learning Models

4.1 Tree Models

4.1.1 Decision Tree - `DecisionTreeRegressor`

The decision tree uses a recursive binary structure to minimize the Mean Squared Error (MSE) during splits.

Objective Function

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y})^2$$

Split Criterion

$$\Delta = \text{MSE}_{\text{parent}} - \left(\frac{N_L}{N}\text{MSE}_{\text{left}} + \frac{N_R}{N}\text{MSE}_{\text{right}}\right)$$

Where:

- N_L, N_R are the number of samples in the left and right child nodes, respectively.

Leaf Prediction

$$\hat{y}_{\text{leaf}} = \frac{1}{N_{\text{leaf}}}\sum_{i\in\text{leaf}} y_i$$

4.1.2 Random Forest - `RandomForestRegressor`

Random Forest aggregates multiple decision trees trained on bootstrapped datasets.

Prediction: For M trees, the prediction for a sample x is the mean of all tree outputs:

$$\hat{y} = \frac{1}{M}\sum_{m=1}^{M} h_m(x)$$

Where $h_m(x)$ is the prediction from the m-th tree.

Features selection in splits: At each split, only a random subset of features k (e.g., $k = \sqrt{\text{total features}}$) is considered.

4.1.3 Gradient Boost Decision Tree - `GradientBoostingRegressor`

Gradient Boosting minimizes a loss function iteratively using decision trees as base learners.

Objective function:

$$L(y, F(x)) = \sum_{i=1}^{N}\ell(y_i, F(x_i))$$

Residual fitting:

At iteration m, compute the negative gradient of the loss:

$$r_i^{(m)} = -\left[\frac{\partial\ell(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$$

Fit a new tree $h_m(x)$ to approximate $r_i^{(m)}$.

Model update:

$$F_m(x) = F_{m-1}(x) + \eta \cdot h_m(x)$$

Where η is the learning rate.

4.1.4 LightGBM - `lgb.LGBMRegressor`

LightGBM optimizes GBDT with histogram-based splitting and a leaf-wise growth strategy.

Histogram-based splitting:

Continuous features are discretized into k bins. The split gain is calculated as:

$$\text{Gain} = \frac{\left(G_L^2/H_L\right) + \left(G_R^2/H_R\right) - \left(G^2/H\right)}{2} - \lambda$$

Where:

- G and H are the first and second-order gradient sums.
- G_L, H_L, G_R, H_R are for left and right child nodes.
- λ is a regularization term.

Leaf-wise growth:

LightGBM grows the leaf with the maximum split gain first, unlike GBDT's level-wise growth.

Objective function:

Using a second-order Taylor expansion for the loss function:

$$\mathcal{L}(F) \approx \sum_{i=1}^{N} \left[g_i h(x_i) + \frac{1}{2} h_i h(x_i)^2 \right] + \Omega(h)$$

Where:

- g_i: first-order gradient.
- h_i: second-order gradient.

4.1.5 Tree Model Parameter Optimization

1. Define the Objective Function

The tuning process starts by defining an **objective function** that Optuna will minimize. This includes:

- Defining the hyperparameter search space for the model.
- Splitting the data into training and validation sets.
- Training the model with sampled hyperparameters.
- Computing the validation mean squared error (MSE) as the objective value:

$$\text{val\_mse} = \frac{1}{n} \sum_{i=1}^{n} \left(y_{\text{val},i} - \hat{y}_{\text{val},i} \right)^2$$

2. Hyperparameter Sampling

Optuna uses different methods to sample hyperparameters:

- **Categorical parameters:** Sampled using `trial.suggest_categorical`.
- **Integer parameters:** Sampled within a range using `trial.suggest_int`.
- **Float parameters:** Sampled within a continuous range using `trial.suggest_float`.

For example, a parameter space with bounds $[a, b]$ for a float:

$$\text{param} = \texttt{trial.suggest\_float("param", a, b)}$$

3. Train and Evaluate the Model

- The dataset is split into training and validation sets using `train_test_split`.
- The model is trained with the sampled hyperparameters.
- Validation predictions are made, and the MSE is calculated:

$$\hat{y}_{\text{val}} = \text{model.predict}(X_{\text{val}})$$

$$\text{val\_mse} = \text{mean\_squared\_error}(y_{\text{val}}, \hat{y}_{\text{val}})$$

4. Optimize with Optuna

Optuna iteratively optimizes the objective function:

- **Search direction:** Minimize the validation MSE.
- **Number of trials:** The number of parameter configurations to test is specified by `n_trials`.
- The study saves the best hyperparameters:

$$\text{best\_params} = \text{study.best\_params}$$

5. Output the Best Hyperparameters

The best parameters are printed and returned:

$$\text{Best parameters for the model: } \texttt{study.best\_params}$$

Parameter Search Space:

Model	Parameter	Range/Values
Random Forest	`n_estimators`	100 – 500 (integer)
	`max_depth`	5 – 30 (integer)
Gradient Boosting	`n_estimators`	100 – 500 (integer)
	`max_depth`	3 – 15 (integer)
	`learning_rate`	0.01 – 0.2 (float)
Decision Tree	`max_depth`	5 – 30 (integer)
	`min_samples_split`	2 – 10 (integer)
	`min_samples_leaf`	1 – 5 (integer)
LightGBM	`n_estimators`	100 – 500 (integer)
	`max_depth`	5 – 30 (integer)
	`learning_rate`	0.01 – 0.2 (float)
	`num_leaves`	10 – 50 (integer)

Table 3: Hyperparameter Search Space for Different Models

```python
def optuna_tune_model(dataframe, model_name, model_class, param_space, n_trials=10):
    """
    Use Optuna to tune hyperparameters for a single model.

    Parameters:
        dataframe (pd.DataFrame): The data containing features and target variable.
        model_name (str): Name of the model.
        model_class: The model class (e.g., RandomForestRegressor).
        param_space (dict): Parameter space for Optuna to explore.
        n_trials (int): Number of trials for Optuna to perform.

    Returns:
        dict: The best parameters for the model.
    """
    def objective(trial):
        params = {}
        for key, values in param_space.items():
            if isinstance(values, list):
                params[key] = trial.suggest_categorical(key, values)
            elif isinstance(values, tuple) and len(values) == 2:
                if isinstance(values[0], int):  # Use int range
                    params[key] = trial.suggest_int(key, *values)
                else:  # Use float range
                    params[key] = trial.suggest_float(key, *values)
            else:
                raise ValueError(f"Invalid parameter space for {key}: {values}")

        X = dataframe.iloc[:, :-1] # feature
        y = dataframe.iloc[:, -1] # real price
        X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2)

        if model_name == "LightGBM":
            model = model_class(verbose=-1, **params)  # Suppress LightGBM output
        else:
            model = model_class(**params)

        model.fit(X_train, y_train)
        y_val_pred = model.predict(X_val)
        val_mse = mean_squared_error(y_val, y_val_pred)

        return val_mse

    # Create and optimize the Optuna study
    study = optuna.create_study(direction="minimize")
    study.optimize(objective, n_trials=n_trials)

    print(f"Best params for {model_name}: {study.best_params}")
    return study.best_params
```

Figure 5: Illustration of Tree Model Parameter Optimization Workflow

Model	Parameter	Value
Random Forest	n_estimators	139
	max_depth	19
Gradient Boosting	n_estimators	367
	max_depth	5
	learning_rate	0.0928
Decision Tree	max_depth	19
	min_samples_split	5
	min_samples_leaf	3
LightGBM	n_estimators	430
	max_depth	17
	learning_rate	0.1676
	num_leaves	48

Table 4: Best Hyperparameters for Different Models

4.1.6 Tree Model Result and Conclusion

For each tree model, we train in different data size. The dataset sizes range from approximately 0.09 to 0.91, representing the variation in the scale of data used for model training and testing. This range

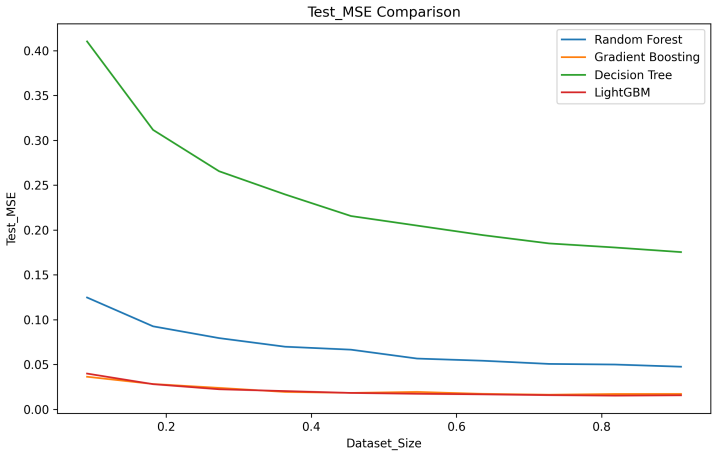
allows for evaluating model performance across small to relatively larger datasets.

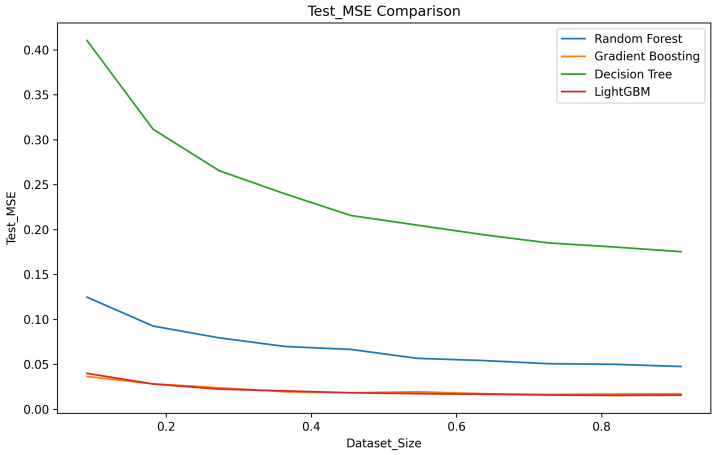
Among the tree models, LightGBM consistently outperforms others in both training and testing metrics. It achieves the lowest average Test MSE (0.02180) and Test MAE (0.12218), indicating high predictive accuracy and generalization ability.

Gradient Boosting achieves competitive performance, with a low Test MSE (0.01999) and Test MAE (0.11067). However, GBDT is really time-consuming compared with LightGBM, so we choose LightGBM as our best tree model for later comparison with deep learning models.

Model	**Avg. Train MSE**	**Avg. Train MAE**	**Avg. Test MSE**	**Avg. Test MAE**
Random Forest	0.01014	0.07498	0.09188	0.19532
Decision Tree	0.03052	0.13534	0.25817	0.38218
Gradient Boosting	0.01287	0.08784	0.01999	0.11067
LightGBM	0.00503	0.05521	0.02180	0.12218

Model	**Average Training Time (s)**
Random Forest	55.058
Decision Tree	0.535
Gradient Boosting	101.106
LightGBM	2.811





4.2 Deep Learning Models

4.2.1 Artificial Neural Network

Artificial Neural Networks (ANNs) are computational models designed to learn complex mappings from inputs to outputs. A basic feedforward neural network, consisting of layers of neurons, can approximate various functions given sufficient capacity and training.

Network Architecture

- Input: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layers: Transform the input through weights and nonlinear activations.
- Output: $\mathbf{y}$, the final prediction of the network.

If we have L layers, let $\mathbf{h}^{(0)} = \mathbf{x}$, and for each layer l (where $1 \leq l \leq L$):

$$\mathbf{z}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{h}^{(l)} = \sigma(\mathbf{z}^{(l)})$$

The final output is:

$$\mathbf{y} = \mathbf{h}^{(L)}$$

Here, $W^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for layer l.

Nonlinear Activation Functions

To capture nonlinear relationships, an activation function $\sigma(\cdot)$ is applied element-wise:

$$\sigma(z) = \begin{cases} \frac{1}{1+e^{-z}} & \text{(Sigmoid)} \\ \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} & \text{(Tanh)} \\ \max(0, z) & \text{(ReLU)} \end{cases}$$

Loss Function

To train the network, we measure the discrepancy between predictions $\mathbf{y}$ and targets $\mathbf{t}$ using a loss function $\mathcal{L}(\mathbf{y}, \mathbf{t})$.
For regression (Mean Squared Error):

$$\mathcal{L} = \frac{1}{2}\|\mathbf{y} - \mathbf{t}\|^2$$

For classification (Cross-Entropy):

$$\mathcal{L} = -\sum_c t_c \log y_c$$

Training (Backpropagation and Gradient Descent)

To find optimal parameters $\{W^{(l)}, \mathbf{b}^{(l)}\}$, we minimize $\mathcal{L}$.

1. **Forward pass**: Compute $\mathbf{y}$ and $\mathcal{L}$.
2. **Backward pass**: Compute gradients using the chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$$

3. **Update parameters**:

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial W^{(l)}}, \quad \mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$$

Here, η is the learning rate.

Summary

Artificial Neural Networks are powerful function approximators. By stacking linear transformations and nonlinear activations, and optimizing parameters to minimize a loss function, they can learn complex mappings from input data to target outputs.

4.2.2 ANN Architecture and Result

Module Setup

The `NeuralNet` module is a feedforward neural network designed for regression tasks. The input layer is configured with a dimensionality of 10, corresponding to the number of features in the input data.

The network architecture incorporates multiple hidden layers, each equipped with a Rectified Linear Unit (ReLU) activation function, which introduces non-linearity into the model. Additionally, a dropout layer is employed after each hidden layer as a regularization technique to prevent overfitting. Dropout works by randomly deactivating a fraction of neurons during training, ensuring that the model does not become overly dependent on specific neurons and enhancing its generalization to unseen data. The final output layer consists of a single neuron and does not apply any activation function, allowing the output to take on any real value.

The `NeuralNet` module is as follows:

```python
class NeuralNet(nn.Module):
    def __init__(self, input_dim, n_layers=2, hidden_units=128, dropout_rate=0.3):
        """
        Initialize the neural network architecture.

        Parameters:
        - input_dim (int): Number of input features.
        - n_layers (int): Number of hidden layers.
        - hidden_units (int): Number of neurons in each hidden layer.
        - dropout_rate (float): Dropout rate for regularization.
        """
        super(NeuralNet, self).__init__()
        layers = []
        for i in range(n_layers):
            input_dim_layer = input_dim if i == 0 else hidden_units
            layers.append(nn.Linear(input_dim_layer, hidden_units))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout_rate))
        layers.append(nn.Linear(hidden_units, 1))  # Output layer for regression
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

Figure 6: ANN model setup

Data Preprocessing

The `data_processing` function first identifies and addresses missing values within the dataset. The function separates the features and target variable. The 10 features includes moneyness, K, T, r, q, v0, theta, kappa, sigma, and rho, while the target variable is designated as the call prices calculated by the Heston Model. The data is split into training, validation, and testing sets using an 8:1:1 ratio.

A robust scaling step is applied to standardize the data, ensuring that the features are on a similar scale and resistant to the influence of outliers. Notably, the validation and test sets are scaled using the mean and variance computed from the training set, preventing future information leakage. The target variable is also scaled in a similar manner

The dimensions of the datasets are as follows:

- Training Set (80% of total dataset): $(87,964,10), (87,964)$
- Validation Set (10% of total dataset): $(10,995,10), (10,995)$
- Test Set (10% of total dataset): $(10,996,10), (10,996)$

Grid Searching for the Best Model Configuration

The grid search function evaluates all possible combinations of hyperparameters to identify the optimal configuration for training a neural network. For each combination, a new instance of the neural network model is constructed. We use Mean Squared Error (MSE) as the loss function, and the Adam optimizer to update model weights based on specified learning rates and weight decay values. The Adam optimizer (Adaptive Moment Estimation) combines the advantages of momentum (taking the exponential average of the past gradients) and RMSProp (normalize the learning rate based on the recent gradients), achieving a more stable and faster convergence. DataLoaders are created for both training and validation datasets, ensuring shuffled batches for training and sequential batches for validation.

The model is trained over a predefined number of epochs, during which the `train` function computes the loss for each batch, applies backpropagation to calculate gradients, and updates the weights to minimize the loss. After training, the model's performance is assessed on the validation dataset through MSE, calculated using the `evaluate_model` function. The model yielding the lowest validation MSE is selected as the best-performing model. The function returns the optimal hyperparameters.

The 36 hyperparameter combinations for grid search are as follows:

Hyperparameter	**Values Tested**
Learning Rate	$[1 \times 10^{-3}, 1 \times 10^{-4}]$
Batch Size	$[256, 512, 1024]$
Number of Layers (n_layers)	5
Hidden Units	$[64, 128, 256]$
Dropout Rate	$[0.1, 0.3]$
Weight Decay ($L2$)	1×10^{-3}

Table 5: Hyperparameter Grid for Model Search

The `train`, `evaluate_model`, and the `grid_search` functions are as follows:

```python
def train(model, train_loader, criterion, optimizer, device):
    model.train()
    total_loss = 0.0
    for batch_data, batch_targets in train_loader:
        batch_data = batch_data.to(device)
        batch_targets = batch_targets.to(device)
        batch_data = batch_data.unsqueeze(1)
        batch_targets = batch_targets.unsqueeze(1)
        outputs = model(batch_data)
        loss = criterion(outputs, batch_targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    return avg_loss
```

Figure 7: Train function

The best model configuration from the grid search is as follows:

```python
def evaluate_model(model, test_loader, criterion):
    model.eval()
    predictions, true_values = [], []
    total_loss = 0.0
    with torch.no_grad():
        for x_test, y_test in test_loader:
            outputs = model(x_test).squeeze()
            y_test = y_test.squeeze()
            predictions.extend(outputs.tolist())
            true_values.extend(y_test.tolist())
            loss = criterion(outputs, y_test)
            total_loss += loss.item()
    mse = mean_squared_error(true_values, predictions)
    mae = mean_absolute_error(true_values, predictions)
    r2 = r2_score(true_values, predictions)
    avg_loss = total_loss / len(test_loader)
    return mse, mae, r2, avg_loss
```

Figure 8: Evaluation function

Hyperparameter	**Value**
Learning Rate	0.0001
Batch Size	256
Number of Layers (n_layers)	5
Hidden Units	256
Dropout Rate	0.1
Weight Decay ($L2$)	0.001

Table 6: Best Hyperparameter Configuration

Validation Loss for the best configuration: 0.000518

Retrain the Model after grid searching

The `retrain_model` function is designed to fine-tune a neural network using the best hyperparameters identified during the grid search. The function employs the Mean Squared Error (MSE) loss function, and Adam optimizer as before stated. Training and testing datasets are wrapped into PyTorch DataLoader objects, enabling mini-batch processing, shuffling, and efficient data access during training and evaluation.

The training process is conducted over 20 epochs. During each epoch, the model is first trained on the training dataset using the train function, which computes and records the average training loss for that epoch (one detail here is that we use a scaler to fit the training targets and reverse the scaled result to get the evaluation matrix in a more real scale). The model's performance is then evaluated on the test dataset using the `evaluate_model` function. This evaluation computes the Mean Squared Error (MSE), Mean Absolute Error (MAE), R-squared (R^2), the average test loss, and the computation time.

After all epochs are completed, the function returns a dictionary containing the retrained model, loss values across epochs, final evaluation metrics (MSE, MAE, R^2), and the average training time per epoch.

The retrain function is as follows:

```python
def grid_search(hyperparameter_grid, train_dataset, val_dataset, input_dim, device, n_epochs):
    """
    Perform grid search to find the best hyperparameter combination.
    Args:
        hyperparameter_grid (dict): Dictionary of hyperparameter lists to search.
        train_dataset (Dataset): Training dataset.
        val_dataset (Dataset): Validation dataset.
        input_dim (int): Number of input features.
        device (torch.device): Device to use for training and evaluation.
        n_epochs (int): Number of epochs for training each model.

    Returns:
        dict: Best hyperparameters and corresponding performance metrics.
    """
    best_mse = float('inf')
    best_params = None
    best_model = None
    # implementing grid search
    for params in product(*hyperparameter_grid.values()):
        hyperparams = dict(zip(hyperparameter_grid.keys(), params))
        print(f"Training with hyperparameters: {hyperparams}")

        model = NeuralNet(input_dim=input_dim,
                          n_layers=hyperparams['n_layers'],
                          hidden_units=hyperparams['hidden_units'],
                          dropout_rate=hyperparams['dropout_rate']).to(device)

        criterion = nn.MSELoss()
        optimizer = torch.optim.Adam(model.parameters(),
                                     lr=hyperparams['learning_rate'],
                                     weight_decay=hyperparams['weight_decay'])
        # Create DataLoaders
        train_loader = DataLoader(train_dataset, batch_size=hyperparams['batch_size'], shuffle=True)
        val_loader = DataLoader(val_dataset, batch_size=hyperparams['batch_size'], shuffle=False)
        # Train the model
        for epoch in range(n_epochs):
            train_loss = train(model, train_loader, criterion, optimizer, device)
            print(f"Epoch {epoch + 1}/{n_epochs} - Train Loss: {train_loss:.4f}")
        # Evaluate the model
        mse, mae, r2, _ = evaluate_model(model, val_loader, criterion)
        print(f"MSE: {mse}, MAE: {mae}, R2: {r2}")
        # Update best parameters
        if mse < best_mse:
            best_mse = mse
            best_params = hyperparams
            best_model = model
    print(f"\nBest Hyperparameters: {best_params} with MSE: {best_mse}")
    return best_params
```

Figure 9: Grid Searching

Results and Analysis

The results demonstrate the neural network's high accuracy and computational efficiency in american option pricing task. The model achieves a very low test Mean Squared Error (MSE) of 0.0206, indicating that the average squared difference between predicted and actual values is minimal. Additionally, the test Mean Absolute Error (MAE) of 0.0897 highlights that the average magnitude of errors is small, showing precise predictions. The model's R^2 score of 0.9987 reveals that it explains over 99.87% of the variance in the target variable, showcasing its ability to capture complex relationships in the data effectively.

In terms of computational efficiency, the model demonstrates an average training time per epoch of 76.12 seconds using CPU, which is manageable even for iterative optimization processes over multiple epochs. More impressively, the average training time per data point is 8.65×10^{-4} seconds, highlighting the neural network's remarkable efficiency in handling individual samples. The computational advantage of Neural Network when dealing with large dataset is significantly outperforms traditional tree-based models. Overall, these results highlight the neural network's superior predictive accuracy and computational efficiency.

We use SHAP value to analyze the feature importance. The figure 13 shows that the time to maurity T has the biggest impact on the model output, followed by moneyness, K and the Heston parameters.

```python
def retrain_model(best_params, train_dataset, test_dataset, input_dim, device, n_epochs, scaler):
    final_model = NeuralNet(input_dim=input_dim,
                            n_layers=best_params['n_layers'],
                            hidden_units=best_params['hidden_units'],
                            dropout_rate=best_params['dropout_rate']).to(device)

    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(final_model.parameters(),
                                 lr=best_params['learning_rate'],
                                 weight_decay=best_params['weight_decay'])

    train_loader = DataLoader(train_dataset, batch_size=best_params['batch_size'], shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=best_params['batch_size'], shuffle=False)
    train_losses = []
    test_losses = []
    epoch_times = []
    for epoch in range(n_epochs):
        start_time = time.time()
        train_loss = train_inverse_scale(final_model, train_loader, criterion, optimizer, device, scaler)
        mse, mae, r2, test_avg_loss = evaluate_model_inverse_scale(final_model, test_loader, criterion, scaler)
        end_time = time.time()
        epoch_time = end_time - start_time
        epoch_times.append(epoch_time)

        train_losses.append(train_loss)
        test_losses.append(test_avg_loss)
        print(f"Epoch {epoch+1}/{n_epochs} - Train Loss: {train_loss:.4f}, test Loss: {test_avg_loss:.4f}, Time:
         {epoch_time:.2f} seconds")
    avg_epoch_time = sum(epoch_times) / len(epoch_times)
    print(f"Average Epoch Time: {avg_epoch_time:.2f} seconds")
    return {
        "final_model": final_model,
        "train_losses": train_losses,
        "test_losses": test_losses,
        "test_mse": mse,
        "test_mae": mae,
        "test_r2": r2,
        "avg_epoch_time": avg_epoch_time
    }
```

Figure 10: Retrain function

Metric	**Value**
Test Mean Squared Error (MSE)	0.0206
Test Mean Absolute Error (MAE)	0.0897
Test R^2 Score	0.9987
Average Training Time / Epoch	76.12 seconds
Average Training Time / Data Point	8.65×10^{-4} seconds
Prediction Time	0.16 seconds
Average prediction Time / Data Point	1.455×10^{-5} seconds

Table 7: Model Evaluation Metrics

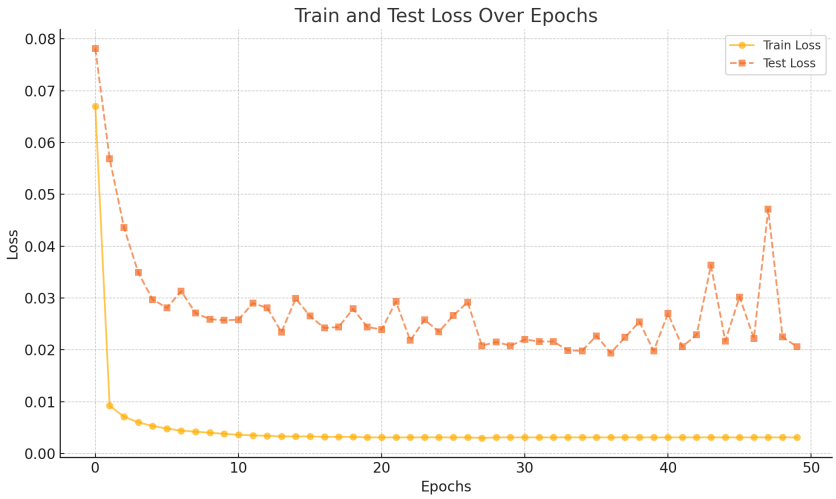


Figure 11: ANN Training and Test Loss Over Time

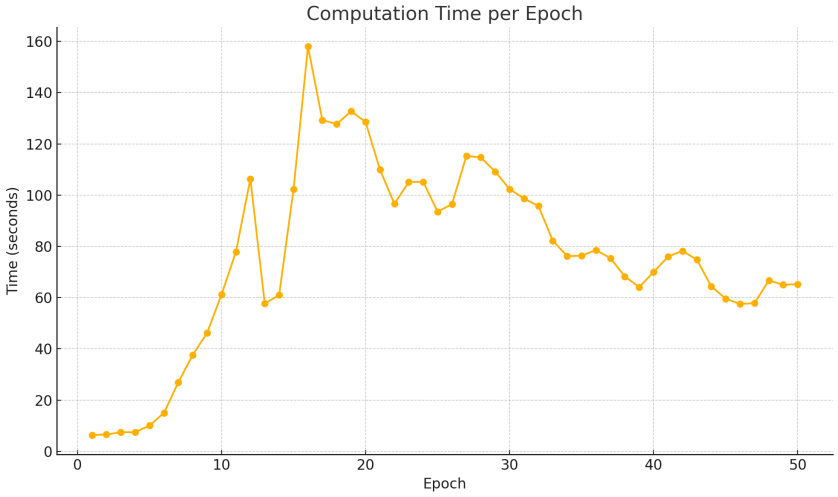


Figure 12: ANN computation time per epoch

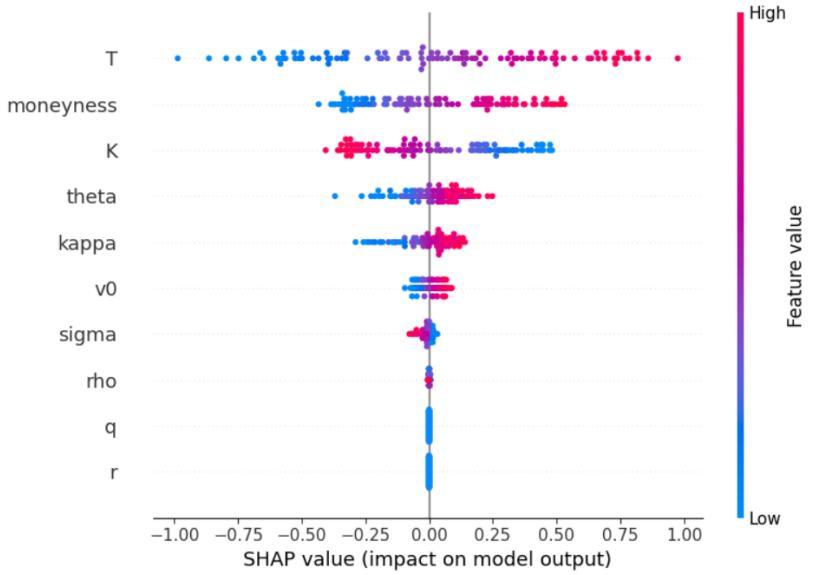


Figure 13: SHAP value analysis

4.2.3 Gradient Boost Neural Network

Gradient Boosted Neural Networks (GBNN) extend the gradient boosting framework by using neural networks as base learners instead of decision trees.

Setup

Given training data $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{N}$ and a differentiable loss function $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$, we want to build an ensemble model:

$$\hat{\mathbf{y}}^{(M)}(\mathbf{x}) = \sum_{m=0}^{M} \nu f^{(m)}(\mathbf{x}; \theta^{(m)}),$$

where each $f^{(m)}$ is a neural network learned at iteration m, and $\nu \in (0, 1)$ is a learning rate.

Algorithm Steps

1 Initialization: Start with an initial prediction:

$$\hat{\mathbf{y}}^{(0)}(\mathbf{x}) = f^{(0)}(\mathbf{x}; \theta^{(0)}).$$

2 Compute Residuals: At the m-th iteration, compute the pseudo-residuals:

$$r_i^{(m)} = - \left. \frac{\partial \mathcal{L}(\mathbf{y}_i, \hat{\mathbf{y}}_i)}{\partial \hat{\mathbf{y}}_i} \right|_{\hat{\mathbf{y}}_i = \hat{\mathbf{y}}_i^{(m-1)}}.$$

3 Train the m-th Neural Network: Fit a neural network to the residuals:

$$\theta^{(m)} = \arg\min_{\theta} \sum_{i=1}^{N} (r_i^{(m)} - f(\mathbf{x}_i; \theta))^2.$$

4 Update the Ensemble: Add the newly trained network's predictions to the ensemble:

$$\hat{\mathbf{y}}^{(m)}(\mathbf{x}_i) = \hat{\mathbf{y}}^{(m-1)}(\mathbf{x}_i) + \nu f^{(m)}(\mathbf{x}_i; \theta^{(m)}).$$

Repeat the above steps for $m = 1, \ldots, M$.

Discussion

- **Advantages:** Neural networks can model complex nonlinear relationships and adapt to various data types.
- **Challenges:** Neural networks are more computationally expensive and can easily overfit, necessitating careful regularization, architectural choices, and hyperparameter tuning.

Conclusion

Gradient Boosted Neural Networks combine the iterative residual-fitting approach of gradient boosting with the representational power of neural networks. While more complex than using decision trees, this method has the potential to yield highly flexible and accurate models if carefully managed.

4.2.4 Gradient Boost Neural Network Architecture and Result

1 Train-Validation-Test Split

- **Training Set**: 80% of the data, used for model training.
- **Validation Set**: 10%, used to tune hyperparameters and monitor overfitting.
- **Test Set**: 10%, used to evaluate final model performance.

This split ensures a balanced distribution of data and separates unseen data for testing.

2 Feature Standardization

- **StandardScaler**: Standardizes features to have a mean of 0 and a standard deviation of 1.
- Standardization is applied to X (features) only, not to y.
- This ensures faster convergence and stable training.

3 Dataset and DataLoader

The `OptionDataset` class is a custom PyTorch dataset that:

- Converts X and y to PyTorch tensors.
- Implements `len` (dataset size) and `getitem` (accessing specific samples).

This design allows seamless integration with PyTorch's `DataLoader`.

- **train_loader**: Supplies batches of training data to the model during training.
- **val_loader**: Supplies validation data in batches to evaluate performance.
- **Batch Size**: Set to 64 by default but can be adjusted.

4 Neural Network Definition

The `GradientBoostNN` is a feedforward neural network designed for regression tasks:

- **Input Layer**: Accepts the 10-dimensional feature vector.
- **Hidden Layers**:
  - Fully connected (`Linear`) layers with ReLU activation introduce non-linearity.
  - Dropout layers prevent overfitting.
  - Dimensionality decreases progressively ($hidden\_units \rightarrow hidden\_units/2$).
- **Output Layer**: A single neuron that outputs the predicted `call` price.

The `nn.Sequential` module conveniently stacks layers for simplicity and clarity in PyTorch.

5 Training and Validation

- **Inputs:**
  - `model`: The neural network.
  - `optimizer`: Optimizes model parameters (e.g., Adam, SGD).
  - `criterion`: Loss function (e.g., MSELoss for regression).
  - `train_loader`: Provides training data.
  - `val_loader`: Provides validation data.
  - `device`: Hardware to use (CPU or GPU).
  - `epochs`: Number of iterations through the entire training dataset.
- **Workflow:**
  1. **Training Phase**:
     - The model is set to `train` mode.
     - For each batch in `train_loader`:
       * Predictions are made.
       * Loss is computed using the criterion.
       * Gradients are backpropagated, and the optimizer updates model weights.
       * Accumulated loss for all batches is recorded.
  2. **Validation Phase**:
     - The model is set to `eval` mode (no weight updates, faster computation).
     - Predictions are made on the validation set.
     - Validation loss is computed and used for performance monitoring.
  3. **Best Model Saving**:
     - The best-performing model (lowest validation loss) is saved for later use.
- **Output**: Returns the best validation loss achieved.

6 Parameter Optimization

This part uses Optuna, a hyperparameter optimization framework, to search for the best hyperparameters for training a neural network. The process involves defining an objective function, which Optuna minimizes by testing various hyperparameter combinations within the specified ranges.

Hyperparameter	Range/Options	Description
`hidden_units`	[32, 64, 96, ..., 256]	Number of neurons in the first hidden layer.
`dropout_rate`	[0.0, 0.5]	Dropout rate to prevent overfitting.
`learning_rate`	10^{-4} to 10^{-2}	Learning rate for the optimizer (log-uniform).
`batch_size`	[64, 128, 256]	Batch size for training and validation.

Table 8: Hyperparameter Ranges for Optimization

Hyperparameter	Optimal Value
Hidden Units	64
Dropout Rate	0.0134
Learning Rate	0.000722
Batch Size	64

Table 9: Optimal Hyperparameters from Optuna Optimization

7 Model Evaluation

After training:

- The model is evaluated on unseen data (test set) to ensure generalization.
- Performance is typically measured using metrics such as:
  - **Mean Squared Error (MSE)**: Measures average squared difference between predicted and actual values.
  - **Mean Absolute Error (MAE)**: Measures average absolute difference between predicted and actual values.

The table below presents the final test metrics and training time:

Metric	Value
Mean Squared Error (MSE)	0.003728
Mean Absolute Error (MAE)	0.046348
Training Time	18.54 seconds

Table 10: GBNN Final Test Metrics and Training Time

4.2.5 Long-short Term Memory Model

Long Short-Term Memory (LSTM) networks are a special type of Recurrent Neural Networks (RNNs) designed to address the issue of long-term dependencies. By introducing a cell state and gated mechanisms, LSTMs effectively reduce the problems of gradient vanishing and exploding that traditional RNNs often face.

Key Variables

- $x_t \in \mathbb{R}^d$: Input at time t.
- $h_t \in \mathbb{R}^h$: Hidden state at time t.
- $c_t \in \mathbb{R}^h$: Cell state at time t.

Parameters

$$W_f, W_i, W_o, W_c \in \mathbb{R}^{h \times d}$$
$$U_f, U_i, U_o, U_c \in \mathbb{R}^{h \times h}$$
$$b_f, b_i, b_o, b_c \in \mathbb{R}^h$$

Equations

At each time step t:

1 Forget Gate

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

2 Input Gate and Candidate Cell State

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$
$$\tilde{c}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

3 Cell State Update

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

4 Output Gate and Hidden State

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$
$$h_t = o_t \odot \tanh(c_t)$$

Here, $\sigma(\cdot)$ is the sigmoid function and $\odot$ denotes element-wise multiplication.

Summary

The LSTM architecture allows the network to selectively preserve or discard information over long time periods. Through its gating mechanisms, the LSTM can better handle long-term dependencies in sequential data and mitigate issues like gradient vanishing or exploding.

4.2.6 LSTM Architecture and Result

This model uses a Long Short-Term Memory (LSTM) architecture designed for processing sequential data. The dataset consists of 109,955 data points, split into 20 percent for testing and 80 percent for training and validation. Within the training and validation split, 64 percent of the total data is used for training and 16 percent for validation. The architecture includes 2 LSTM layers followed by 1 linear layer, allowing the model to capture temporal dependencies in the data and map the learned representations to the desired output.

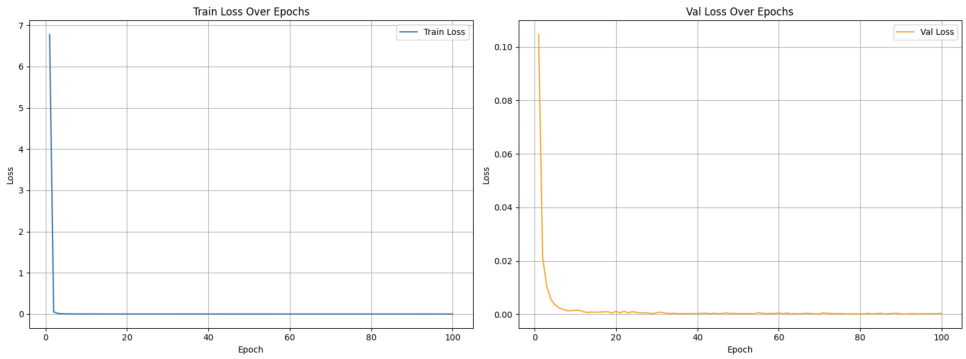
LSTMs are ideal for tasks involving sequential patterns, such as time-series forecasting, natural language processing, or anomaly detection. The LSTM layers use gates (forget, input, and output) to selectively retain or discard information, solving the long-term dependency problem. The final linear layer processes the output of the LSTM layers into a target format, making the architecture versatile for tasks like regression, classification, or future value predictions.

Additionally, we used the unsqueeze() function to add a dimension to the data, allowing it to fit the 3-dimensional input expected by the LSTM. Since our data is randomly generated, its sequence length is consequently set to 1.

LSTM parameter optimization

Optuna was employed to optimize the hyperparameters of the model, utilizing its efficient search capabilities over large parameter spaces. The hyperparameter search included the following parameters:

- **Hidden Size:** Chosen from categorical values {16, 32, 64, 128}.
- **Learning Rate:** Sampled logarithmically within the range $[1 \times 10^{-4}, 1 \times 10^{-2}]$.
- **Batch Size:** Selected from {16, 32, 64, 128}.



Metric/Parameter	Best Value
Hidden Size	64
Learning Rate	0.001424157949952588
Batch Size	64
Minimum Validation MSE	0.00019937166936208749

Table 11: Best Model Parameters and Validation MSE

LSTM Result

The ultimate LSTM result are as follows:

Metric	LSTM Model	FD Model
Test MSE	0.000202	-
Test MAE	0.011037	-
Total Runtime (seconds)	0.290052	6757.652811
Test Data Points Processed	21,991	-
Runtime per Price (seconds)	1.318959×10^{-5}	0.33788264

Table 12: Comparison of LSTM Model and FD Model Metrics and Runtime

4.2.7 Convolutional Neural Networks

CNNs are designed to process data with a grid-like topology, such as images. They use convolutional layers to efficiently extract local patterns and build hierarchical feature representations.

Convolution Operation

Given an input $\mathbf{X} \in \mathbb{R}^{H \times W \times C_{in}}$ and filters $\mathbf{W} \in \mathbb{R}^{K_H \times K_W \times C_{in} \times C_{out}}$, the convolution produces an output $\mathbf{Y} \in \mathbb{R}^{H_{out} \times W_{out} \times C_{out}}$:

$$Y_{h,w,c} = \sum_{u=1}^{K_H} \sum_{v=1}^{K_W} \sum_{c'=1}^{C_{in}} X_{h+u-1,w+v-1,c'} W_{u,v,c',c} + b_c.$$

Activation Function

A nonlinear activation such as ReLU is applied:

$$Z_{h,w,c} = \max(0, Y_{h,w,c}).$$

Pooling

Pooling layers reduce spatial dimensions. For max pooling with window $P_H \times P_W$ and stride S:

$$Z^{\text{pool}}_{h,w,c} = \max_{0 \le u < P_H, 0 \le v < P_W} Z_{hS+u,wS+v,c}.$$

Fully Connected Layers and Output

After convolution and pooling, the resulting features are flattened and passed to fully connected layers:

$$\mathbf{o} = W_{\text{fc}}\mathbf{z} + \mathbf{b}_{\text{fc}}$$

For classification, a softmax is used:

$$p(y = c|\mathbf{x}) = \frac{e^{o_c}}{\sum_{c'} e^{o_{c'}}}.$$

Loss Function and Training

Define a loss function, such as cross-entropy, and minimize it via gradient descent:

$$\mathcal{L}(\mathbf{y}, \mathbf{o}) = -\sum_{c} y_c \log p(y = c|\mathbf{x}).$$

During backpropagation:

$$W \leftarrow W - \eta \frac{\partial \mathcal{L}}{\partial W}, \quad b \leftarrow b - \eta \frac{\partial \mathcal{L}}{\partial b}.$$

Summary

CNNs leverage local connectivity, weight sharing, and hierarchical feature extraction to efficiently learn from image-like data. Through convolution, pooling, and fully connected layers, CNNs capture increasingly abstract features and excel in tasks such as image classification, object detection, and more.

4.2.8 CNN Architecture and Result

1 Data Loading

The dataset was loaded using the command `pd.read_csv("/content/processed_data.csv")` and contained a total of 109,955 samples. This indicates that the dataset provides ample training samples for a deep learning model.

2 Handling Missing Values

Missing values were identified using `data.isnull().sum()` and removed using `data = data.dropna()` to ensure the model training process is error-free.

3 Data Standardization

Features were standardized using `StandardScaler`, while target values were robustly scaled using `RobustScaler`:

```
feature_scaler = StandardScaler()
target_scaler = RobustScaler()
```

Standardized features were stored in `X_scaled`, and robust scaling reduced the influence of outliers on the target values.

4 Data Reshaping

The features were reshaped to match the input format required by `Conv1d`:

```
X_scaled = X_scaled.reshape((X_scaled.shape[0], 1, X_scaled.shape[1]))
```

A single channel was used since the data is a standard one-dimensional feature sequence.

5 Dataset Splitting

The data was split into training (70%), validation (15%), and test (15%) sets using `train_test_split`. The validation set was used to monitor the model's generalization during training and hyperparameter tuning.

6 Feature Statistics

After preprocessing, the following statistics were observed:

- Total samples: 109,955
- Number of features: 10
- Unscaled target (call price) statistics: mean ≈ 8.1, standard deviation ≈ 4.0, range ≈ 0.6 to 16.0.

7 PyTorch Dataset and DataLoader Creation

The `OptionDataset` class was defined to encapsulate the features (`X`) and targets (`y`) into a PyTorch dataset object for efficient data loading.
DataLoader Instances Separate DataLoader instances were created for the training, validation, and test sets, specifying batch size and whether to shuffle the data.

8 CNN Model Architecture

The CNN model consists of:

- Input dimensions: `(batch_size, 1, 10)`.
- Two `Conv1d` layers, each followed by `BatchNorm1d` and `MaxPool1d`.
- A `Flatten` layer followed by fully connected layers.
- Dropout with a rate of 0.5 to prevent overfitting.
- A single output neuron for regression predictions.

9 Regularization and Normalization

Dropout and batch normalization were used to improve generalization.
The `train_and_evaluate` function includes:

- Training and validation phases for each epoch.
- Mean Squared Error (MSE) loss for evaluation.
- Learning rate scheduler (`ReduceLROnPlateau`).
- Early stopping with a patience of 30 epochs.

10 Grid Search for Hyperparameter Optimization

The following hyperparameters were explored:

- Learning rates: `[0.001, 0.0005, 0.0001, 0.00005]`.
- Batch sizes: `[128, 256]`.
- Dropout rates: `[0.3, 0.4, 0.5]`.
- Optimizers: `['Adam', 'RMSprop', 'AdamW']`.

All 48 combinations of hyperparameters were tested using the same `train_and_evaluate` function. Results were stored in a DataFrame, and the combination with the lowest validation loss was selected.

11 Best Hyperparameters

The best hyperparameter combination was:

- Learning rate: 0.001
- Batch size: 256
- Dropout rate: 0.3
- Optimizer: Adam

12 Final Model Training

The final model was trained with the best hyperparameters. Early stopping ensured training was halted at optimal performance.

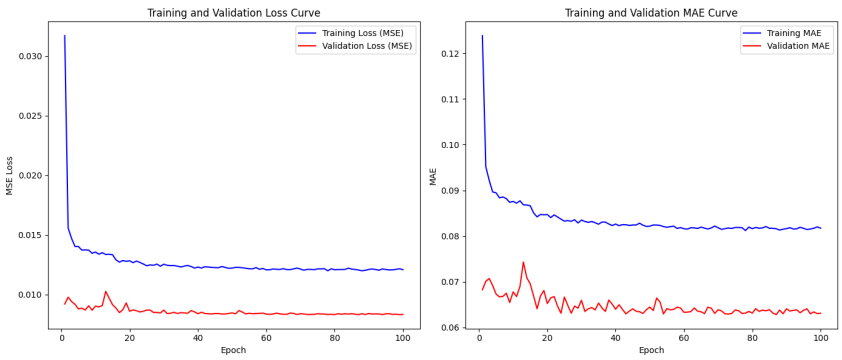
13 Training Results

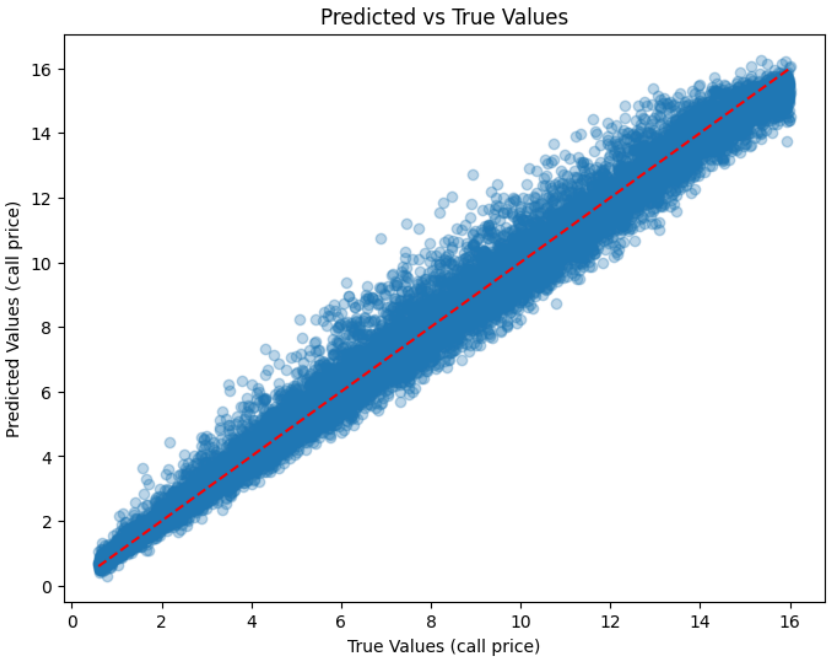
The best validation loss achieved during final training was approximately 0.008309.
The final model was evaluated on the test set with the following metrics:

Metric	**Value**
Test MSE (scaled)	0.008512
Test MAE (scaled)	0.063513
Test MSE	0.325626
Test MAE	0.392836
Test RMSE	0.570636
Total Runtime	0.16 seconds
Runtime per Price	0.000010 seconds/sample

Table 13: CNN Test Metrics for Model Performance

14 Model Result





5 Results and Conclusions

The table below summarizes the test MSE, test MAE, and training time for various models:

Model	**Test MSE**	**Test MAE**	**Predict Time (s)**
LightGBM	0.02180	0.12218	2.811
ANN	0.0206	0.0897	0.16
GBNN (Gradient Boost NN)	0.003728	0.04635	18.54
LSTM	0.000202	0.01104	0.290
CNN	0.32563	0.39284	0.16

Table 14: Consolidated Test Metrics and Prediction Times for All Models

Conclusion

The consolidated test metrics indicate that the LSTM model outperforms all other models in terms of prediction accuracy, achieving the lowest Test MSE (0.000202) and Test MAE (0.01104). Additionally, its prediction time (0.290 seconds) is reasonably low, making it a highly efficient and accurate choice for the given task. This suggests that LSTM is well-suited for scenarios where both precision and computational efficiency are critical.

The GBNN model also demonstrates strong predictive accuracy, with a low Test MSE (0.003728) and Test MAE (0.04635). However, its prediction time (18.54 seconds) is significantly longer than the other models, which could be a limitation in real-time or large-scale applications. Despite its accuracy, the high computational cost may hinder its practical utility.

ANN and LightGBM deliver a balance between prediction accuracy and efficiency. ANN, with a Test MSE of 0.0206 and a Test MAE of 0.0897, achieves reasonably good accuracy while maintaining an extremely low prediction time (0.16 seconds). Similarly, LightGBM performs moderately well in terms of accuracy, with slightly higher error metrics (Test MSE: 0.0218, Test MAE: 0.12218) but requires a longer prediction time (2.811 seconds). These models may be suitable for scenarios where computational efficiency is prioritized over optimal accuracy.

Lastly, the CNN model performs poorly in this comparison, with the highest Test MSE (0.32563) and Test MAE (0.39284). Although its prediction time is very low (0.16 seconds), the lack of accuracy makes it an unsuitable choice for this specific task.

In conclusion, the LSTM model emerges as the best-performing option due to its superior accuracy and reasonable prediction time. GBNN could be considered for applications where accuracy is paramount and computational cost is less of a concern. ANN and LightGBM provide viable trade-offs between accuracy and efficiency, while the CNN model is not recommended for this task.