lesson-2-1

May 14, 2025

Run in Google Colab

1 Deep Learning for Option Pricing

1.1 Introduction

The problem of this first part of our lessons can be stated simply as follow. Let's say, from a very general point of view, that we have a contingent claim $\mathcal{V}$ that depends on D parameters:

$$\mathcal{V}(\mathbf{x}),\, \mathbf{x} = \{\S_\infty, \dots, \S_\mathcal{D}\} \in \mathbb{R}^\mathcal{D}$$

We want to **approximate the pricing function with a neural network**. Remember that we can always consider the NN as a sort of mapping function

$$\Phi : \mathbb{R}^D \to \mathbb{R}$$

trained to compute prices given a point in $\mathbb{R}^D$ representing a particular set of parameters.

The use of a NN as a pricing functions has a number of advantages. the first is that in this way we are able to compute efficiently thousands of prices in a small amount of time, even when the derivative contract has complicated conditions and when the model is complex. This comes with the downside that the neural network may introduce systematic errors that could affect our estimation of the sensitivities in a number of ways.

The second advantage is that instead of training the network on the model parameters, which in general are not observable, we could train the network ***using data that is directly observable in the market***. For example quotes and trades by market participants provide points on the volatility surface. Interpolating between these points as necessary, a trader can derive a reasonable estimate of the implied volatility appropriate for any new plain vanilla European or American option that is of interest. Plain vanilla options are therefore not priced using a model. They are simply priced to be consistent with the market.

The volatility surface derived from the Black–Scholes–Merton model is a convenient interpolation tool for doing this.

Exotic options are generally not as actively traded as plain vanilla options and, as a result, a model is required for pricing. A variety of different models are used in practice. Two conditions that traders would like the model to satisfy are:

- A. The stochastic behavior assumed for the underlying asset price should correspond reasonably well to its observed behavior, and
- B. The volatility surface derived from the model should be reasonably consistent with the volatility surface used to price plain vanilla options.

Two categories of models that are used in practice can be distinguished. The models in the first category focus on condition A by assuming a process for the asset price that is roughly consistent with its observed behavior. The models have parameters that can be chosen to provide an approximate fit to the current volatility surface. Models in the second category focus on condition B and are designed to be exactly consistent with the current volatility surface. Many different models in the first category involving stochastic volatility and jumps have been proposed. Examples of stochastic volatility models are Hull and White (1987) and Heston (1993). Merton (1976) proposed a model that overlays Black-Scholes-Merton model with jumps. Bates (1996) adds jumps to Heston (1993). Madan et al (1998) propose a variance–gamma model where there are only jumps. More recently, rough volatility models where the process for volatility is non-Markov have been suggested by authors such as Gatheral et al (2018).

The second category of models are referred to as local volatility models. The original onefactor local volatility model was suggested by Dupire (1994) and Derman and Kani (1994). It has been extended by authors such as Ren et al (2007) and Saporito et al (2019). Local volatility models by design satisfy condition B.

The usual approach to implementing models in the first category is to choose model parameters to fit the volatility surface as closely as possible. This approach, which we refer to as the ***"model calibration approach"*** or ***MCA***. A drawback of the approach is that some of the points on the volatility surface are likely to be more important than others for any particular exotic option that is considered. It is of course possible to vary the weights assigned to different points on the volatility surface according to the instrument being valued. However, it is difficult to determine in advance what these weights should be. As a result, the points are usually given equal weight when model parameters are determined.

Neural networks can be used in conjunction with MCA to provide fast pricing once model parameters have been determined. Consider an exotic option that is valued using Monte Carlo simulation. As a first step, it is necessary to devote computational resources to creating a data set relating model parameters and exotic option parameters to the price. The pricing model can then be replicated with a neural network. Once this has been done, valuation is several orders of magnitude faster than Monte Carlo simulation because it involves nothing more than a forward pass through the neural network.

In this first notebook we will look at a simple example of this approach. For the sake of simplicity we will consider an example based on the Black and Scholes model for the pricing of a simple European option.

In the second part of these lessons we'll see how to train a neural network on market data (volatility surface) using the heston model as the underlying model.

Specifically, in this notebook:

- We generate a large **synthetic dataset** of option parameters (maturity, strike-to-spot ratio, and volatility) using **Latin Hypercube Sampling (LHS)**, to cover the typical range of market conditions.

- We compute exact Black-Scholes option prices for this synthetic data, building a training set that approximates the theoretical pricing function.
- We construct and train a **fully connected feed-forward neural network** (three hidden layers, ReLU activation) to approximate the mapping from option features to prices.
- We evaluate the performance of the network using Mean Absolute Error (MAE) and Mean Squared Error (MSE) metrics, ensuring consistency with the training objectives discussed in the lectures.
- We save the trained model for later use in fast, real-time pricing applications.

This exercise serves as the first, simpler example before extending the approach to more complex models (e.g., Heston stochastic volatility) and full VFA using implied volatility smiles, as discussed in later parts of the course.

1.2 The Black and Scholes Model

We consider an underlying process $S(t)$ described by the sde

$$dS(t) = a(S,t)dt + b(S,t)dW \tag{1}$$

A scenario is a set of values $\hat{S}^j(t_i)$, $i = 1, \dots, I$ that are an approximation to the $j-th$ realization,$S^j(t_i)$, of the solution of the sde evaluated at times $0 \le t_i \le T$, $i = 1, \dots, I$. A scenario is also called a trajectory. A trajectory can be visualized as a line in the state-vs-time plane describing the path followed by a realization of the stochastic process (actually by an approzimation to the stochastic process).

The Black and Scholes model assumes a market in which the tradable assets are:

1. A risky asset, whose evolution is driven by a geometric brownian motion

$$dS = \mu S dt + \sigma S dw \Rightarrow S(T) = S(t_0)e^{(\mu - \frac{1}{2}\sigma^2)(T-t_0)+\sigma[w(T)-w(t_0)]} \tag{2}$$

2. The money market account, whose evolution is deterministic

$$dB = Brdt \Rightarrow B(T) = B(t_0)e^{r(T-t_0)} \tag{3}$$

The value for a non-dividend-paying underlying stock in terms of the Black–Scholes parameters is:

$$C(S_t, t) = N(d_1)S_t - N(d_2)Ke^{-r(T-t)} \tag{4}$$

$$P(S_t, t) = N(-d_2)Ke^{-r(T-t)} - N(-d_1)S_t \tag{5}$$

where

$$d_1 = \frac{1}{\sigma\sqrt{T-t}}\left[\ln\left(\frac{S_t}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)(T-t)\right] \tag{6}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \tag{7}$$

For both:

- $N(\cdot)$ is the cumulative distribution function of the standard normal distribution
- $T-t$ is the time to maturity (expressed in years)
- S_t is the spot price of the underlying asset
- K is the strike price
- r is the risk free rate (annual rate, expressed in terms of continuous compounding)
- σ is the volatility of returns of the underlying asset

1.2.1 Simpler formulation for vanilla options

A different form for plain vanilla put options

Up until now, we described the pricing of instruments with general payoffs and in particular of vanilla options. In the latter case, starting from the expression of the vanilla put option price

$$P_t = \mathbb{E}\left[e^{-r(T-t)}(K-S_T)^+\right]$$

Rewrite the payoff as:

$$(K-S_T)^+ = S_t\left(\frac{K}{S_t}-\frac{S_T}{S_t}\right)^+$$

because multiplying inside and outside by S_t is the same since S_t is not a random variable but a known value from the filtration $\mathcal{F}_t$.

Thus:

$$P_t = \mathbb{E}\left[e^{-r(T-t)}S_t\left(\frac{K}{S_t}-\frac{S_T}{S_t}\right)^+\right] = S_t\,\mathbb{E}\left[e^{-r(T-t)}\left(\frac{K}{S_t}-\frac{S_T}{S_t}\right)^+\right]$$

$$\Rightarrow P_t = S_t\mathbb{E}\left[\left(e^{-r(T-t)}\frac{K}{S_t}-M_T\right)^+\right]$$

where $M_T = e^{-r(T-t)}S_T/S_t$ is the stochastic process of a martingale with the same volatility of S_t and $M_t = 1$. By comparing this expression with the one from the Black Scholes formula, we notice that we can in general rewrite the price of a vanilla put option as

$$P_(S_t, K, T, \sigma, r) = S_tP(1, e^{-r(T-t)}K/S_t, T, \sigma, 0)$$

From this we can see that we can calculate the price of any vanilla put option by calculating the value of a simpler put option with $r=0$, $S_t=1$ and $e^{-r(T-t)}K/S_t$ in place of the strike price. The exact same argument can be made for a vanilla call option. For simplicity, in the rest of this work we will always consider $r=0$, since we don't lose any generality when the risk-free rate is independent of the asset price. —

What is an Asset-or-Nothing Option?

- An **asset-or-nothing** option is a type of exotic option that **pays the value of the underlying asset** at maturity if a certain condition is met (and zero otherwise).
- For a **put** asset-or-nothing:
  - You receive the asset S_T **only if** $S_T \leq K$ (strike).
- For a **call** asset-or-nothing:
  - You receive the asset S_T **only if** $S_T > K$.

Payoff for an asset-or-nothing put:

$$\text{Payoff} = S_T \cdot \mathbf{1}_{\{S_T \leq K\}}$$

What is a Cash-or-Nothing Option?

- A **cash-or-nothing** option is an exotic option that **pays a fixed amount of cash** (e.g., 1) if a certain condition is met, and nothing otherwise.
- For a **put** cash-or-nothing:
  - You receive a fixed cash payment (e.g., 1) **only if** $S_T \leq K$.
- For a **call** cash-or-nothing:
  - You receive the cash payment **only if** $S_T > K$.

Payoff for a cash-or-nothing put:

$$\text{Payoff} = \mathbf{1}_{\{S_T \leq K\}}$$

How do they relate to plain vanilla options?

- A **plain vanilla European put** can be **decomposed** into a combination of:
  - a **cash-or-nothing put** (giving a fixed amount K),
  - minus an **asset-or-nothing put** (giving the asset S_T).

Formula:

$$(K - S_T)^+ = K \cdot \mathbf{1}_{\{S_T \leq K\}} - S_T \cdot \mathbf{1}_{\{S_T \leq K\}}$$

meaning:

- You get K cash if $S_T \leq K$ (cash-or-nothing),
- and you "lose" S_T if $S_T \leq K$ (asset-or-nothing).

Thus, the **price** of a European put can be built as:

$$\text{Put Price} = K \times \text{Cash-or-Nothing Put Price} - S_0 \times \text{Asset-or-Nothing Put Price}$$

where the second term is evaluated correctly at time 0 under risk-neutral probability.

Why is this decomposition useful?

- **Simplifies** pricing, especially when using forwards,
- **Numerically stable** for small volatility or short maturities,
- Helps in **building other exotic products** (because cash-or-nothing and asset-or-nothing are building blocks).

1.2.2 Function for Option Pricing

The following functions implement:

- **Asset-or-nothing** options (using `__npan__`)
- **Cash-or-nothing** options (using `__npcn__`)
- Pricing of **European calls and puts** (using these two building blocks).

They are **NumPy-enabled**, meaning they work not only for scalars but also for arrays (vectors of inputs) — efficient for Monte Carlo simulations or grid evaluations.

They use the fact that: - Forward prices F are more convenient when pricing European options, - $F = S_0 e^{rT}$ is the **forward price**, - Normal Black-Scholes assumptions are made: lognormal asset dynamics, constant volatility σ, constant rate r.

[46]:
```
import sys
import math
import time
import numpy             as np
import pandas            as pd
import matplotlib.pyplot as plt

from scipy.stats            import norm
from sklearn.model_selection import train_test_split
```

Safe Operation

In the following numerical implementation you will find some tricks to **safely handle very small volatility values** in computations, which is very important in the context of option pricing or Gaussian kernel calculations. Here's a detailed explanation of each step:

```
s = sgma * np.sqrt(T)
```

- This computes the **standard deviation of returns** over time horizon T, where `sgma` is the instantaneous volatility (could be a scalar or array).
- `s` now represents the **effective volatility** over the option's time to maturity.

```
mask = np.where(s < 1.e-8, 1, 0)
MASK = np.where(s >= 1.e-8, 1, 0)
```

- These lines create two **boolean masks**:
  - `mask` identifies the entries in `s` where the volatility is **very close to zero** (numerically, below 10^{-8}).
  - `MASK` is the complementary mask — it identifies the entries where volatility is **not negligible**.
- These masks are typically used to apply **special-case logic** when volatility is nearly zero, which could otherwise cause **division by zero** or **numerical instability**.

```
s = np.where(mask, 1., s)
```

- This line **modifies** `s` to ensure safe computation:
  - Wherever the volatility is **too small**, it **replaces it with** `1.0`, purely as a placeholder.

  - This prevents any downstream calculation from dividing by zero (or near-zero), while the use of the `mask` later allows to **undo or bypass** these artificial values where needed.
  - The value `1.0` is arbitrary and acts as a **neutral denominator** — it won't affect results if properly masked out later.

Why This Matters

In computational finance, very small volatility values can:

- Lead to **division by zero** (e.g., when computing $\frac{x-\mu}{\sigma}$)
- Cause **exploding gradients** or **NaNs** in neural network training
- Trigger **singular matrices** or unstable numerical kernels

This pattern is a **safe-guarding technique**: it keeps the computation stable while flagging special cases for controlled handling.

```
[47]: #
      # NumPy-enabled Asset-or-Nothing option function
      #
      def __npan__(Fw, T, sgma, k):
          """
          Computes the asset-or-nothing put probability for given inputs.

          Parameters:
          - Fw   : Forward price (F = S0 * exp(r*T))
          - T    : Time to maturity
          - sgma : Volatility (sigma)
          - k    : Strike price

          Returns:
          - Asset-or-nothing probability, elementwise if inputs are arrays
          """
          s = sgma * np.sqrt(T)  # Standard deviation over time T
          # Define masks: 'mask' for very small volatility cases, 'MASK' for normal␣
       ↪cases
          mask = np.where(s < 1.e-8, 1, 0)
          MASK = np.where(s >= 1.e-8, 1, 0)
          # Avoid division by zero in degenerate cases by setting s artificially to 1␣
       ↪where needed
          s = np.where(mask, 1., s)
          # Compute the d_minus term
          dm = (np.log(k / Fw) - 0.5 * s * s) / s
          # Compute asset-or-nothing probability (normal CDF of d_minus)
          an = norm.cdf(dm)
          # Handle degenerate cases explicitly
          m1 = np.logical_and(mask, Fw <= k)
          res = np.where(m1, 1., 0.)          # If Fw <= k and volatility ~0,␣
       ↪probability = 1
          res += np.where(MASK, an, 0.0)       # Otherwise use normal CDF
```

```
    print(res.shape)
    return res
#
# NumPy-enabled Cash-or-Nothing option function
#
def __npcn__(Fw, T, sgma, k):
    """
    Computes the cash-or-nothing put probability for given inputs.

    Parameters:
    - Fw   : Forward price (F = S0 * exp(r*T))
    - T    : Time to maturity
    - sgma : Volatility (sigma)
    - k    : Strike price

    Returns:
    - Cash-or-nothing probability, elementwise if inputs are arrays
    """
    s = sgma * np.sqrt(T)  # Standard deviation over time T
    # Define masks: 'mask' for very small volatility cases, 'MASK' for normal␣
 ↪cases
    mask = np.where(s < 1.e-8, 1, 0)
    MASK = np.where(s >= 1.e-8, 1, 0)
    # Avoid division by zero
    s = np.where(mask, 1., s)
    # Compute the d_plus term
    dp = (np.log(k / Fw) + 0.5 * s * s) / s
    # Compute cash-or-nothing probability (normal CDF of d_plus)
    cn = norm.cdf(dp)
    # Handle degenerate cases explicitly
    m1 = np.logical_and(mask, Fw <= k)
    res = np.where(m1, 1., 0.)            # If Fw <= k and volatility ~0,␣
 ↪probability = 1
    res += np.where(MASK, cn, 0.0)        # Otherwise use normal CDF
    return res
#
# Forward-based European put option pricing
#
def np_fw_euro_put(F, T, sgma, k):
    """
    Prices a European put option given forward price.

    Parameters:
    - F    : Forward price
    - T    : Time to maturity
    - sgma : Volatility (sigma)
    - k    : Strike price
    Returns:
    - Forward-based European put price
    """
    # Put price as the combination:
    # put = K * (cash-or-nothing probability) - F * (asset-or-nothing␣
 ↪probability)
    return k * __npcn__(F, T, sgma, k) - F * __npan__(F, T, sgma, k)
#
# Forward-based European call option pricing
#
def np_fw_euro_call(F, T, sgma, k):
    """
    Prices a European call option given forward price.

    Parameters:
    - F    : Forward price
    - T    : Time to maturity
    - sgma : Volatility (sigma)
    - k    : Strike price

    Returns:
    - Forward-based European call price
    """
    # Use put-call parity at forward measure:
    # call = put + (F - K)
    return np_fw_euro_put(F, T, sgma, k) + F - k
#
# Spot-based European put option pricing
#
def np_euro_put(So, r, T, sigma, k):
    """
    Prices a European put option starting from spot price.

    Parameters:
    - So    : Spot price at time 0
    - r     : Risk-free rate
    - T     : Time to maturity
    - sigma : Volatility (sigma)
    - k     : Strike price

    Returns:
    - Spot-based European put price
    """
    # First convert spot price to forward price
    Fw = np.exp(r * T) * So
    # Compute forward-based put and discount it back to today
    return np.exp(-r * T) * np_fw_euro_put(Fw, T, sigma, k)

#
# Spot-based European call option pricing
#
def np_euro_call(So, r, T, sigma, k):
    """
    Prices a European call option starting from spot price.

    Parameters:
    - So    : Spot price at time 0
    - r     : Risk-free rate
    - T     : Time to maturity
    - sigma : Volatility (sigma)
    - k     : Strike price

    Returns:
    - Spot-based European call price
    """
    # First convert spot price to forward price
    Fw = np.exp(r * T) * So
    # Compute forward-based call and discount it back to today
    return np.exp(-r * T) * np_fw_euro_call(Fw, T, sigma, k)
```

[48]:

```
def test__npan__():
    """
    Test suite for the __npan__ function.
    """
    # Test 1: Zero volatility, Fw < k -> should return 1.0
    assert np.isclose(__npan__(90, 1.0, 0.0, 100), 1.0), "Test 1 failed"

    # Test 2: Zero volatility, Fw > k -> should return 0.0
    assert np.isclose(__npan__(110, 1.0, 0.0, 100), 0.0), "Test 2 failed"

    # Test 3: Non-zero volatility, known input
    result = __npan__(100, 1.0, 0.2, 100)
    expected = norm.cdf((np.log(100/100) - 0.5 * 0.2**2) / (0.2 * np.sqrt(1.0)))
    assert np.isclose(result, expected), "Test 3 failed"

    # Test 4: Vectorized input
    Fw_test = np.array([90, 100, 110])
    T_test = np.array([1.0, 1.0, 1.0])
    sigma_test = np.array([0.0, 0.01, 0.2])
    k_test = np.array([100, 100, 100])
    result = __npan__(Fw_test, T_test, sigma_test, k_test)
    assert result.shape == (3,), "Test 4 shape failed"
    assert result[0] == 1.0, "Test 4 value[0] failed"
    assert 0.0 < result[1] < 1.0, "Test 4 value[1] failed"
    assert 0.0 < result[2] < 1.0, "Test 4 value[2] failed"

    print("All tests passed!")


# Run the test function
test__npan__()
```

```
()
()
()
(3,)
All tests passed!
```

1.3 Training Data Generation

To train the network to approximate the pricing function, we need a training dataset. For the network approximating the Black Scholes pricing function, we created one by drawing $T, K/S_0$ and σ using the Latin Hypercube Sampling (LHS) (see below) within the ranges usually found in real market.

1.3.1 Sampling Utilities

Latin Hypercube Sampling

- Imagine you want to generate random points inside a box — for example, two variables x_1 and x_2 each between 0 and 1.
- A **normal random sampling** (pure Monte Carlo) just picks points completely at random: some areas might get many points, other areas might get none — it's messy.
- **Latin Hypercube Sampling** is smarter:
  - It **divides** the range of each variable into **equal intervals** (like slicing into 10 slices).
  - Then it **forces** the random samples to pick **exactly one point in each slice** for each variable.
  - In 2D, it's like making sure that every row and every column has exactly one point.

So, **LHS spreads the points out much more evenly** across the space.

Very simple visual:

Imagine a 10×10 grid:

- **Pure random sampling** → points can cluster in one corner, and leave other areas empty.
- **Latin hypercube sampling** → each **row and each column** gets exactly one point, so the points are "well distributed" across the grid.

(That's why it's called "hypercube" — in many dimensions — and "Latin" — like a Latin square where each number appears exactly once per row and column.)

Advantages of Latin Hypercube Sampling:

Feature	Latin Hypercube Sampling (LHS)	Pure Random Sampling
Spread of points	Much more uniform	Random and often clustered
Coverage of space	Guaranteed: each interval is sampled	Might miss regions entirely
Accuracy of estimates (e.g., integration)	Better with fewer points	Needs many more points for similar accuracy
Efficiency	High	Lower (more samples needed)

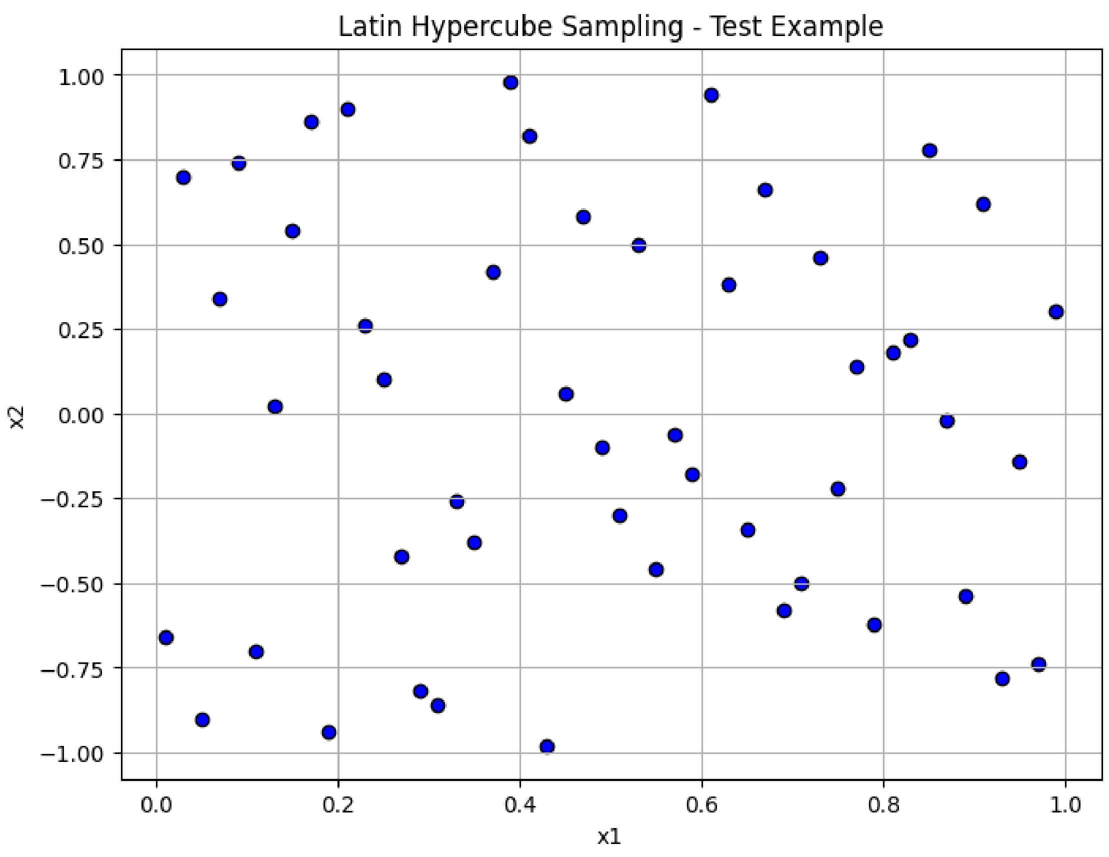
In short:

- Latin Hypercube makes sure you **cover all parts** of your input space,
- It **avoids gaps** and **reduces clustering**,
- It gives you **better, more reliable** results **with fewer samples**.

[49]:
```
from smt.sampling_methods import LHS

def lhs_sampling(rand, NUM, bounds=None):
    """
    Generate a Latin Hypercube Sampling (LHS) over specified variable bounds.

    Parameters:
    - rand   : (unused in this version) random seed or generator, placeholder
    - NUM    : number of samples to generate
    - bounds : dictionary where each key is a variable name and each value is a␣
↪2-element list [lower_bound, upper_bound]

    Returns:
    - X : Pandas DataFrame containing the generated samples, one column per␣
↪variable
    """
    # Convert dictionary keys into a list
    # Note: list(bounds) gives the list of dictionary's keys
    kw = list(bounds)
    # Initialize an empty array to store bounds as (lower, upper) pairs
    limits = np.empty(shape=(0, 2))
    # Build the array of bounds by concatenating each variable's limits
    for k in kw:
        # Each 'bounds[k]' is assumed to be a list or array like [lower_bound,␣
↪upper_bound]
        limits = np.concatenate((limits, [bounds[k]]), axis=0)
    # Create the LHS (Latin Hypercube Sampling) object with the bounds
    sampling = LHS(xlimits=limits)
    # Generate NUM random samples within the specified limits
    x = sampling(NUM)
    # Initialize an empty DataFrame to store samples in readable format
    X = pd.DataFrame()
    # For each variable, fill the corresponding column in the DataFrame
    for n in range(len(kw)):
        tag = kw[n]         # Variable name (key)
        X[tag] = x[:, n]    # Assign the nth sampled column to the variable

    return X
```

[50]:
```
# Now, create a simple test example

# 1. Define variable bounds
bounds = {
    'x1': [0.0, 1.0],    # Variable x1 ranges from 0 to 1
    'x2': [-1.0, 1.0],   # Variable x2 ranges from -1 to 1
}

# 2. Number of samples
NUM_SAMPLES = 50

# 3. Call the sampling function
samples = lhs_sampling(rand=None, NUM=NUM_SAMPLES, bounds=bounds)

# 4. Display the samples
print(samples.head())

# 5. Plot the samples to visualize the distribution
plt.figure(figsize=(8,6))
plt.scatter(samples['x1'], samples['x2'], c='blue', edgecolors='k')
plt.title('Latin Hypercube Sampling - Test Example')
plt.xlabel('x1')
plt.ylabel('x2')
plt.grid(True)
plt.show()
```

```
     x1    x2
0  0.99  0.30
1  0.81  0.18
2  0.19 -0.94
3  0.77  0.14
4  0.69 -0.58
```



1.3.2 Sample Generator

We assume $r = q = 0, S_0 = 1$ and create a data set of NUM observations by randomly sampling from uniform distributions for the other five inputs to the Black-Scholes-Merton formula. The lower and upper bounds of the uniform distributions are as indicated in the `bounds` dictionary. For each set of parameters sampled, we calculate the Black-Scholes-Merton price using equations (4) and (5).

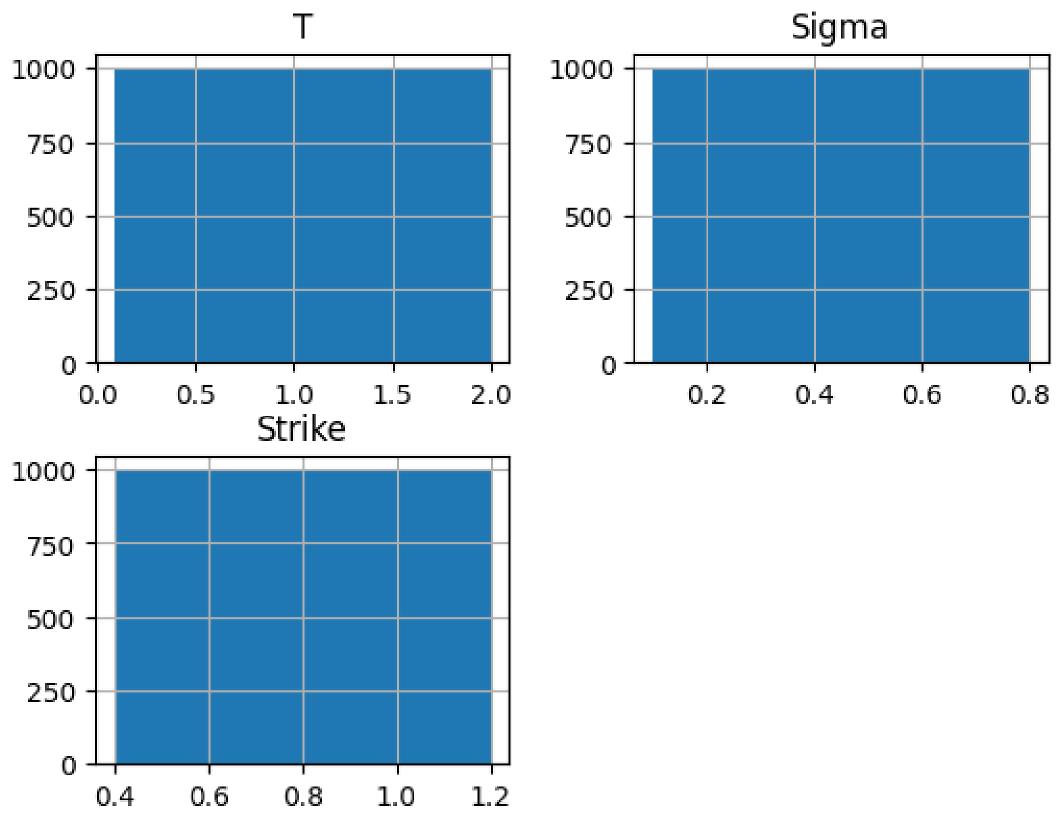
[51]:
```
# For the neural network calculating the vanilla option prices in the Black␣
↪Scholes model,
# the features used are the time to maturity T, the moneyness (simply defined as␣
↪'Strike')
# and the asset volatility sigma.
#
# Lower and upper boundaries for each parameter
#
bounds = {  "T"      : [1./12., 2.00]
          , "Sigma" : [ .1   ,   .80]
          , "Strike": [ .4    , 1.20]
         }
#
```

```
# Number of Observations
#
NUM = 100000
#
# Random number generator
#
seed = 42
rand = np.random.RandomState()
#
# Latin Hypercube Sampling
#
xDF = lhs_sampling(rand, NUM, bounds=bounds)
xDF.head()
```

```
[51]:           T     Sigma    Strike
      0  1.406495  0.272505  0.527228
      1  0.582155  0.114578  1.124932
      2  0.589765  0.588240  0.980652
      3  0.187111  0.391301  0.639588
      4  0.312174  0.621658  0.445804
```

```
[52]: xDF.hist(bins=100)
```

```
[52]: array([[<Axes: title={'center': 'T'}>, <Axes: title={'center': 'Sigma'}>],
              [<Axes: title={'center': 'Strike'}>, <Axes: >]], dtype=object)
```



The for each set of randomly generated parameters, we calculate the theoretical **call** price according to the Black and Scholes model:

```
[53]: def gen(NUM, pars):

          S0     = np.full(NUM, 1.0, dtype = np.double)
          r      = np.full(NUM, 0.0, dtype = np.double)
          price  = np_euro_call(S0, r, pars["T"], pars["Sigma"], pars["Strike"])

          df = pd.DataFrame(pars)
          df["Price"] = price

          return df
```

```
[54]: __tStart = time.perf_counter()
      df =  gen(NUM, xDF)
      __tEnd = time.perf_counter()
      print("@ %-34s: elapsed %.4f sec" %("GEN", __tEnd - __tStart) )

      df.head()
```

```
(100000,)
```

```
@ GEN                               : elapsed 0.0402 sec
```

[54]:
```
          T     Sigma    Strike     Price
0  1.406495  0.272505  0.527228  0.474848
1  0.582155  0.114578  1.124932  0.003817
2  0.589765  0.588240  0.980652  0.186807
3  0.187111  0.391301  0.639588  0.360586
4  0.312174  0.621658  0.445804  0.554973
```

[55]:
```
X_train, X_test = train_test_split(df, test_size=0.33, random_state=42)
print(len(X_train), len(X_test))
X_train.head()
```

```
67000 33000
```

[55]:
```
              T     Sigma    Strike     Price
59428  1.809090  0.167967  0.503644  0.496410
34957  0.269317  0.553044  1.141180  0.064171
4264   1.670669  0.539940  0.434044  0.590927
53791  0.708790  0.527795  0.593532  0.426389
82114  0.488469  0.206684  0.575772  0.424230
```

[56]:
```
X_test.head()
```

[56]:
```
              T     Sigma    Strike     Price
75721  0.782715  0.272890  0.439676  0.560338
80184  0.767535  0.792122  1.092564  0.239891
19864  0.129630  0.181911  0.908204  0.093753
76699  1.467617  0.550055  0.946020  0.281797
92991  1.334160  0.364464  1.166364  0.109128
```

To make the illustration more interesting, we can add a random error to each calculated price. The random error is normally distributed with a mean of zero and a standard deviation of $\epsilon \cdot (P_{max} - P_{min})$ where ϵ is a scale parameter defined by the user.

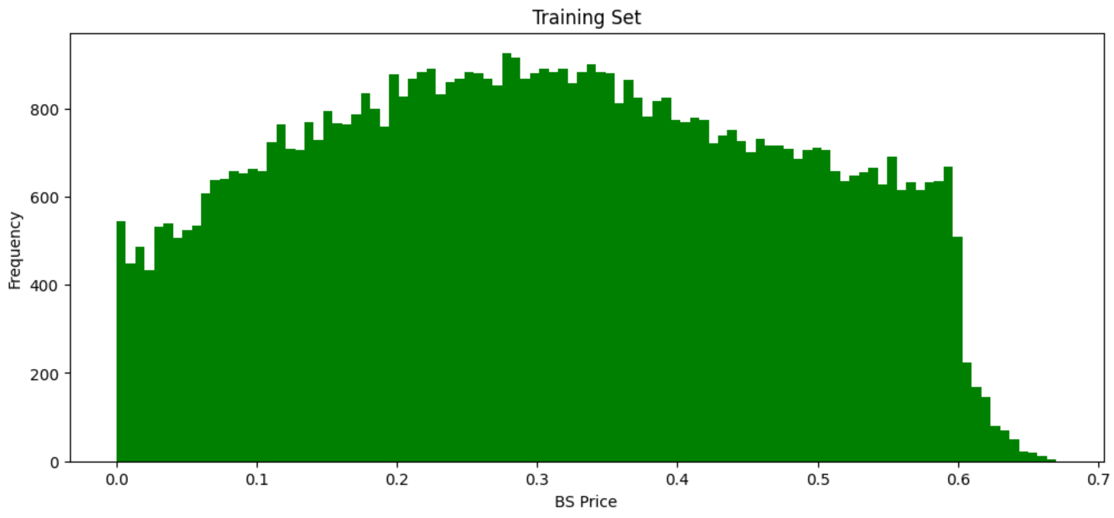
[57]:
```
def add_noise(rand, Xv, eps):
    X  = Xv.copy()
    xl = np.min(X["Price"])
    xh = np.max(X["Price"])

    xi = rand.normal( loc = 0.0, scale = eps*(xh-xl), size=X.shape[0])
    X["Price"] += xi
    return X
```

[58]:
```
EPS = 0.0
if EPS > 0.0:
    X_train = add_noise(rand, X_train, EPS)
```

```
[59]: y_train = X_train['Price']
      y_test  = X_test['Price']
```

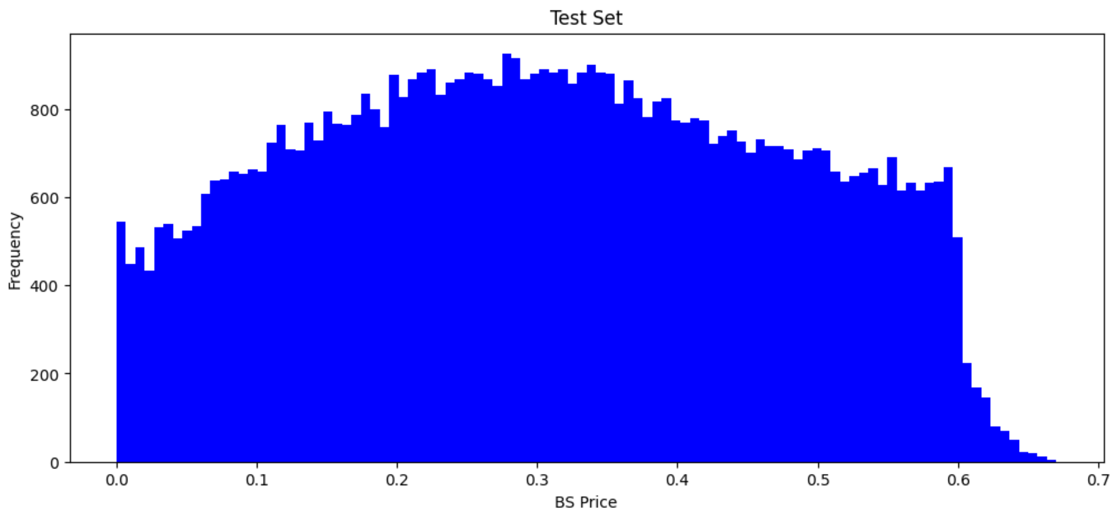
```
[60]: ax=y_train.plot(kind='hist', bins=100, grid=False, figsize=(12,5),␣
       ↪color='green', title = 'Training Set')
      ax.set_xlabel('BS Price')
```

```
[60]: Text(0.5, 0, 'BS Price')
```



```
[61]: ax=y_train.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='blue',␣
       ↪title = 'Test Set')
      ax.set_xlabel('BS Price')
```

```
[61]: Text(0.5, 0, 'BS Price')
```



[62]:
```
X_train = X_train.drop(['Price'], axis=1)
X_test  = X_test.drop(['Price'], axis=1)
X_train.head()
```

[62]:
```
              T     Sigma    Strike
59428  1.809090  0.167967  0.503644
34957  0.269317  0.553044  1.141180
4264   1.670669  0.539940  0.434044
53791  0.708790  0.527795  0.593532
82114  0.488469  0.206684  0.575772
```

1.3.3 Scaling Data

[63]:
```
from sklearn.preprocessing import StandardScaler

scaling = False

if scaling:
    # Scale the train set
    scaler = StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)

    # Scale the test set
    scaler = StandardScaler().fit(X_test)
    X_test = scaler.transform(X_test)

    # Transform into pd DataFrame
    X_train = pd.DataFrame(X_train)
    X_test = pd.DataFrame(X_test)
```

1.4 Create the Model

1.4.1 Neural Network Architecture

We use mean absolute error (mae) as the cost function. The neural network has two hidden layers and a decreasing number of neurons per layer. The relu activation function is used.

[64]:
```
from keras.models import Sequential
from keras.layers import Dense
```

The input shape

What flows between layers are tensors. Tensors can be seen as matrices, with shapes. Shapes are consequences of the model's configuration. Shapes are tuples representing how many elements an array or tensor has in each dimension.

Ex: a shape (30,4,10) means an array or tensor with 3 dimensions, containing 30 elements in the first dimension, 4 in the second and 10 in the third, totaling 30*4*10 = 1200 elements or numbers.

In Keras, the input layer itself is not a layer, but a tensor. It's the starting tensor you send to the first hidden layer. This tensor must have the same shape as your training data.

Ex: if you have 30 images of 50x50 pixels in RGB (3 channels), the shape of your input data is (30,50,50,3). Then your input layer tensor, must have this shape.

Since the input shape is the only one you need to define, Keras will demand it in the first layer.

[65]:
```
def model_builder( inputShape = (1,)):
    # Initialize the constructor
    model = Sequential()
    # Start from the first hidden layer, since the input is not actually a layer␣
 ↪
    # But inform the shape of the input, with inputShape elements.
    model.add(Dense(200, activation='relu', input_shape=inputShape))
    # Add one more hidden layer
    model.add(Dense(200, activation='relu'))
    # Add one more hidden layer
    model.add(Dense(200, activation='relu'))
    # Add an output layer
    model.add(Dense(1))
    # End model construction

    # Model output shape
    #print("model.output_shape: %s" %(str(model.output_shape)))
    # Model summary
    #print("Model.summary"); model.summary()
    # Model config
    #print("Model.config"); model.get_config()

    model.compile(loss='mse', optimizer='rmsprop', metrics=['mae'])
    return model
```

[68]:
```
model = model_builder( inputShape = (X_train.shape[1],))
#model.summary()
```

1.5 Train the Model

[22]:
```
def show_scattered( y, t, TAG, ax = None):
    #x      = model.predict(X)
    #y      = np.ravel(x)
    xMin = min(t)
    xMax = max(t)
    v    = np.arange(xMin, xMax, (xMax-xMin)/100.)
    diff   = np.fabs(y - t)
    print("@ %-24s: E[y-t]: %.6f Std(y-t): %.6f" %( TAG, np.mean(diff), np.
↪std(diff)))
    if ax == None: return

    ax.plot( y, t, ".")
    ax.plot( v, v, color="red")
    ax.set_title("%s mae=%8.4f, std=%8.4f" %(TAG, np.mean(diff), np.std(diff)))
    ax.set_xlabel("predicted")
    ax.set_ylabel("target")
```

[23]:
```
def display_nn_results( model, X_train, X_test, t_train, t_test, resFile=None):

    fig, ax = plt.subplots(1,2, figsize=(12,6))
    fig.suptitle("Scattered plots")
    #
    # The numpy module ravel of NumPy provides a function, called numpy.ravel,␣
↪which is used to change
    # a 2-dimensional array or a multi-dimensional array into a contiguous␣
↪flattened array. The returned
    # array has the same data type as the source array or input array.
    #
    y_train  = np.ravel(model.predict(X_train))
    show_scattered( y_train, t_train, "InSample", ax = ax[0])
    diff     = np.fabs(y_train - t_train)
    delta    = y_train - t_train
    RES      = pd.DataFrame({"predicted": y_train, "target": t_train, "err": diff,␣
↪"delta": delta})
    h1       = RES['delta']
    RES.to_csv("res_in_sample.csv", sep=',', float_format="%.6f", index=True)
    print("@")

    y_test  = np.ravel(model.predict(X_test))
    show_scattered( y_test , t_test, "OutOfSample", ax= ax[1])
    diff     = np.fabs(y_test-t_test)
    delta    = y_test - t_test
    RES      = pd.DataFrame({"predicted": y_test, "target": t_test, "err": diff,␣
↪"delta": delta})
    h2       = RES['delta']
    RES.to_csv("res_ou_sample.csv", sep=',', float_format="%.6f", index=True)
    print("@")

    if resFile != None:
        plt.savefig(resFile, format="png")
        print("@ %-12s: results saved to '%s' "%("Info", resFile))
    plt.show()
    score = model.evaluate(X_test, t_test, verbose=1)
    print('Score:'); print(score)

    h1.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='blue',␣
 ↪title = 'err distribution')
    h2.plot(kind='hist', bins=100, grid=False, figsize=(12,5), color='red')
```

```
[24]: print(X_train.shape)
      print(X_test.shape)
      print(y_train.shape)
      print(y_test.shape)
```

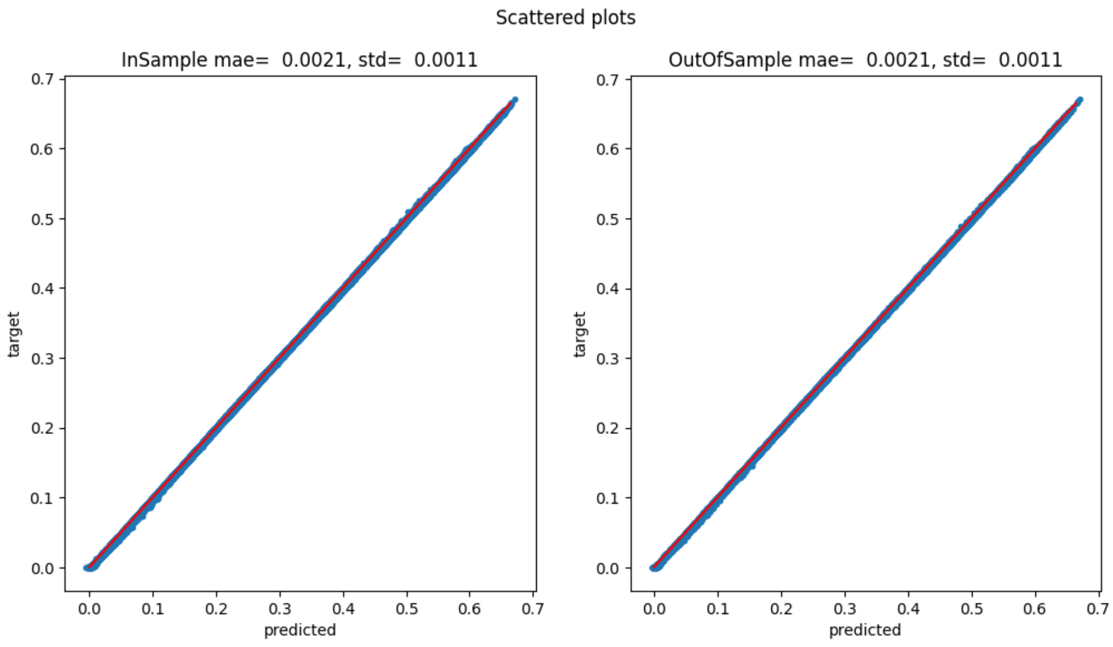
```
(67000, 3)
(33000, 3)
(67000,)
(33000,)
```

```
[25]: frames = [X_train, X_test]
      X = pd.concat(frames)

      frames = [y_train, y_test]
      Y = pd.concat(frames)
```

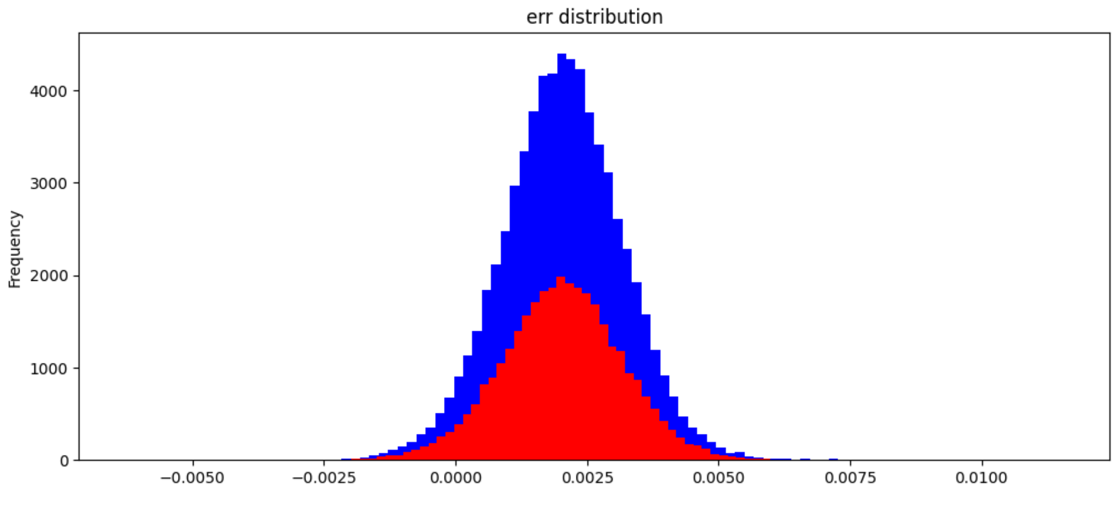
```
[26]: # Fit the model
      history = model.fit(X, Y, validation_split=0.33, epochs=25, verbose=0)
```

```
[27]: import warnings
      warnings.simplefilter('ignore')

      display_nn_results(model, X_train, X_test, y_train, y_test, resFile='training.
       ↪png')
```

```
2094/2094  3s 2ms/step
@ InSample                : E[y-t]: 0.002077 Std(y-t): 0.001076
@
1032/1032  2s 2ms/step
@ OutOfSample             : E[y-t]: 0.002075 Std(y-t): 0.001081
@
@ Info        : results saved to 'training.png'
```



```
1032/1032  3s 3ms/step -
loss: 5.4276e-06 - mae: 0.0021
Score:
[5.474354111356661e-06, 0.0020752165000885725]
```



1.5.1 Access Model Training History in Keras

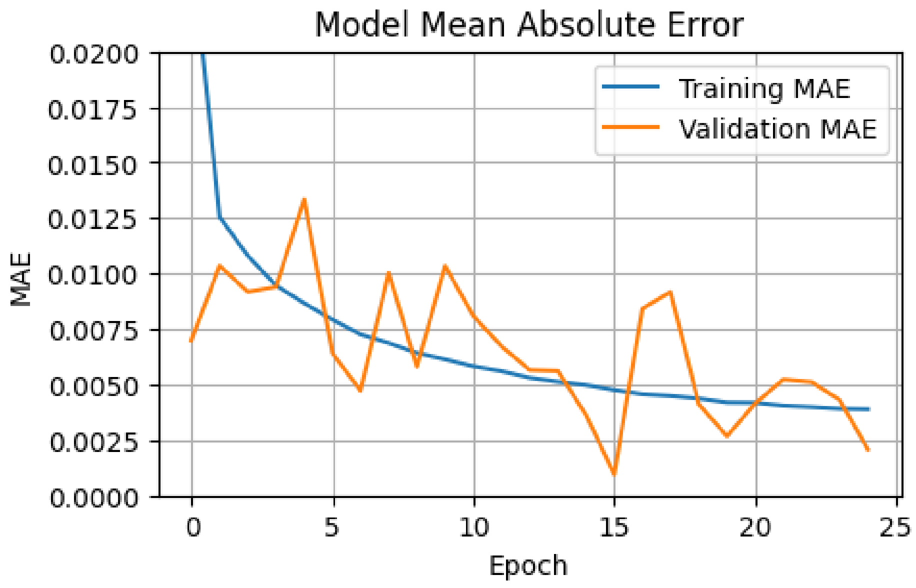
Keras provides the capability to register callbacks when training a deep learning model. One of the default callbacks that is registered when training all deep learning models is the **History callback**.

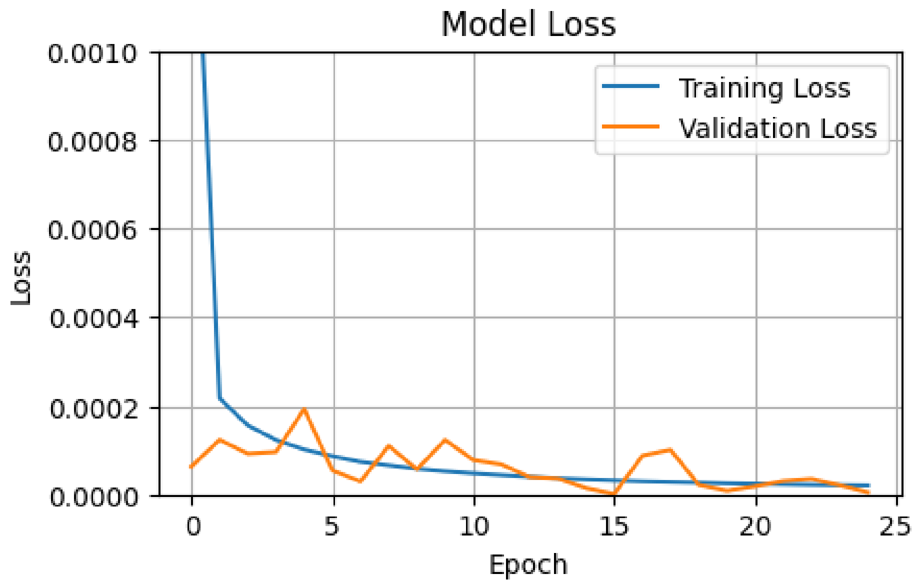
It records training metrics for each epoch. This includes the loss and the accuracy (for classification problems) as well as the loss and accuracy for the validation dataset, if one is set.

The history object is returned from calls to the fit() function used to train the model. Metrics are stored in a dictionary in the history member of the object returned.

For example, you can list the metrics collected in a history object using the following snippet of code after a model is trained:

[28]:
```
# Set fixed y-axis limits
Y_MIN = 0.0     # Typically zero for loss/MAE
Y_MAX_1 = 0.02    # Example: 2% MAE, you can adapt based on your expected errors
Y_MAX_2 = 0.001

# Plot MAE
plt.figure(figsize=(5,3))
plt.plot(history.history['mae'], label='Training MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.title('Model Mean Absolute Error')
plt.ylabel('MAE')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)
plt.ylim([Y_MIN, Y_MAX_1])  # <- Fix y-axis
plt.show()

# Plot Loss
plt.figure(figsize=(5,3))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)
plt.ylim([Y_MIN, Y_MAX_2])  # <- Fix y-axis
plt.show()
```



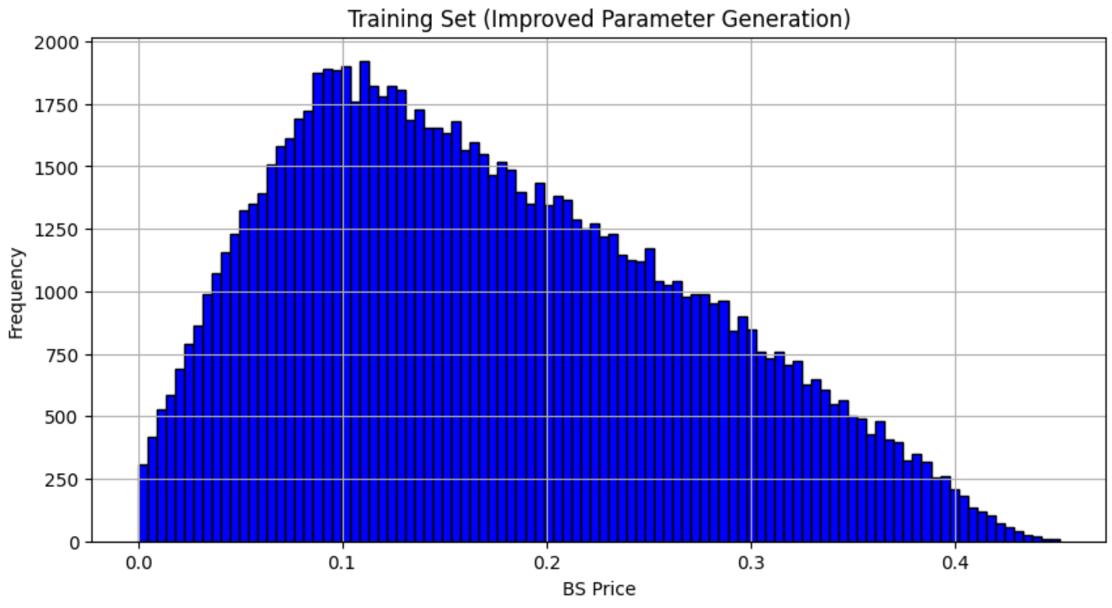


1.6 A Better Approximation

[29]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from smt.sampling_methods import LHS
from scipy.stats import norm
```

```
# Improved LHS sampling
def improved_lhs_sampling(NUM):
    """
    Improved parameter generation based on moneyness, volatility, maturity, and␣
 ↪rate.

    Returns:
    - DataFrame with sampled parameters
    """
    bounds = {
        'moneyness': [0.9, 1.1],  # S0/K ratio
        'Sigma'    : [0.1, 0.8],  # volatility
        'T'        : [1./12., 2.0],  # time to maturity
        'r'        : [0.0, 0.0], # interest rate
    }

    # List of keys
    kw = list(bounds)

    # Build limits array
    limits = np.array([bounds[k] for k in kw])

    # LHS sampling
    sampling = LHS(xlimits=limits)
    x = sampling(NUM)

    # Create DataFrame
    X = pd.DataFrame()
    for n in range(len(kw)):
        tag = kw[n]
        X[tag] = x[:, n]

    return X

# Simple Black-Scholes call price function (assuming K=1)
def bs_call_price(S0, K, r, sigma, tau):
    if sigma * np.sqrt(tau) < 1e-10:
        return max(S0 - K * np.exp(-r * tau), 0.0)
    d1 = (np.log(S0 / K) + (r + 0.5 * sigma ** 2) * tau) / (sigma * np.sqrt(tau))
    d2 = d1 - sigma * np.sqrt(tau)
    call_price = S0 * norm.cdf(d1) - K * np.exp(-r * tau) * norm.cdf(d2)
    return call_price

# Now, generate samples
NUM_SAMPLES = 100000
samples = improved_lhs_sampling(NUM_SAMPLES)
```

```
# Assume S0 = 1 for simplicity
samples['K'] = 1.0 / samples['moneyness']

# Compute Black-Scholes call prices
samples['Price'] = samples.apply(lambda row: bs_call_price(1.0, row['K'],␣
↪row['r'], row['Sigma'], row['T']), axis=1)

# Plot the histogram of generated call prices
plt.figure(figsize=(10, 5))
plt.hist(samples['Price'], bins=100, color='blue', edgecolor='k')
plt.title('Training Set (Improved Parameter Generation)')
plt.xlabel('BS Price')
plt.ylabel('Frequency')
plt.grid(True)
plt.show()
```



[30]:
```
samples.head()
```

[30]:
```
   moneyness     Sigma         T    r         K     Price
0   1.029135  0.437677  1.837745  0.0  0.971690  0.244392
1   0.987707  0.729668  0.431122  0.0  1.012446  0.184338
2   1.063941  0.212214  1.935111  0.0  0.939902  0.146348
3   1.096487  0.787145  1.386561  0.0  0.912004  0.386810
4   0.963755  0.447532  0.549840  0.0  1.037608  0.116281
```

[31]:
```
X_train_2, X_test_2 = train_test_split(samples, test_size=0.33, random_state=42)
print(len(X_train_2), len(X_test_2))
X_train_2.head()
```

```
67000 33000
```

[31]:
```
       moneyness     Sigma         T    r         K     Price
59428   0.915435  0.613062  0.811465  0.0  1.092377  0.184241
34957   1.000457  0.204353  1.739880  0.0  0.999543  0.107414
4264    0.948323  0.304789  0.681573  0.0  1.054493  0.077867
53791   1.046995  0.603920  0.218621  0.0  0.955114  0.133639
82114   1.001341  0.453511  0.947817  0.0  0.998661  0.175274
```

[32]:
```
y_train_2 = X_train_2['Price']
y_test_2  = X_test_2['Price']
```

[33]:
```
X_train_2 = X_train_2.drop(['Price', 'moneyness', 'r'], axis=1)
X_test_2  = X_test_2.drop(['Price', 'moneyness', 'r'], axis=1)
X_train_2.head()
```

[33]:
```
          Sigma         T         K
59428  0.613062  0.811465  1.092377
34957  0.204353  1.739880  0.999543
4264   0.304789  0.681573  1.054493
53791  0.603920  0.218621  0.955114
82114  0.453511  0.947817  0.998661
```

[34]:
```
print(X_train_2.shape)
print(X_test_2.shape)
print(y_train_2.shape)
print(y_test_2.shape)
```

```
(67000, 3)
(33000, 3)
(67000,)
(33000,)
```

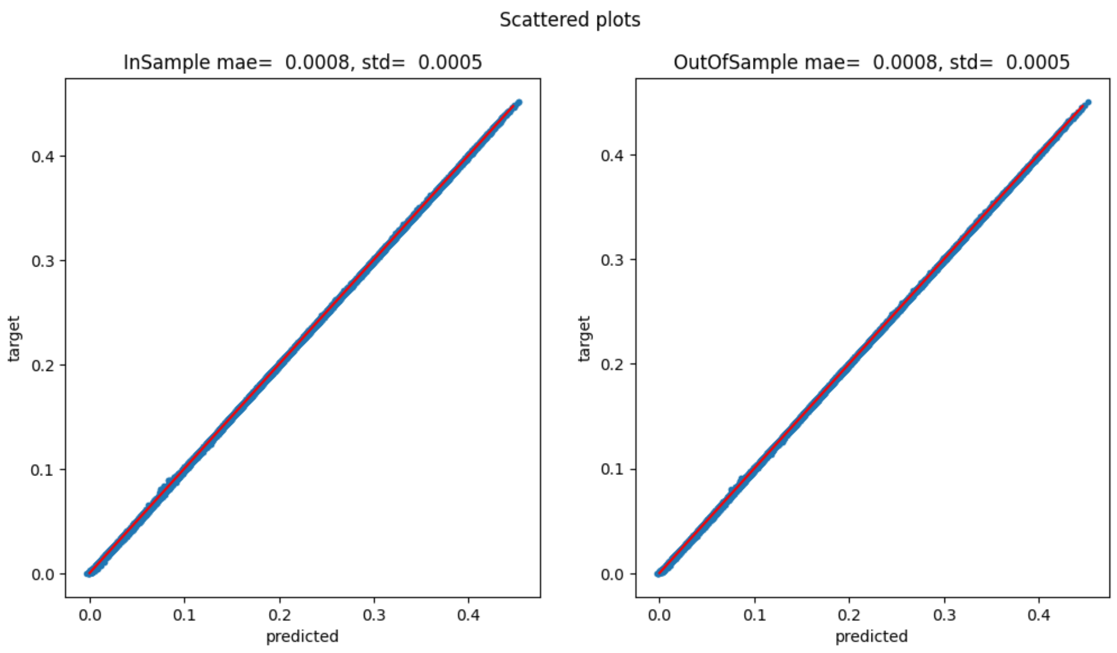
[35]:
```
frames = [X_train_2, X_test_2]
X_2 = pd.concat(frames)

frames = [y_train_2, y_test_2]
Y_2= pd.concat(frames)
```

[36]:
```
# Fit the model
history = model.fit(X_2, Y_2, validation_split=0.33, epochs=25, verbose=0)
```

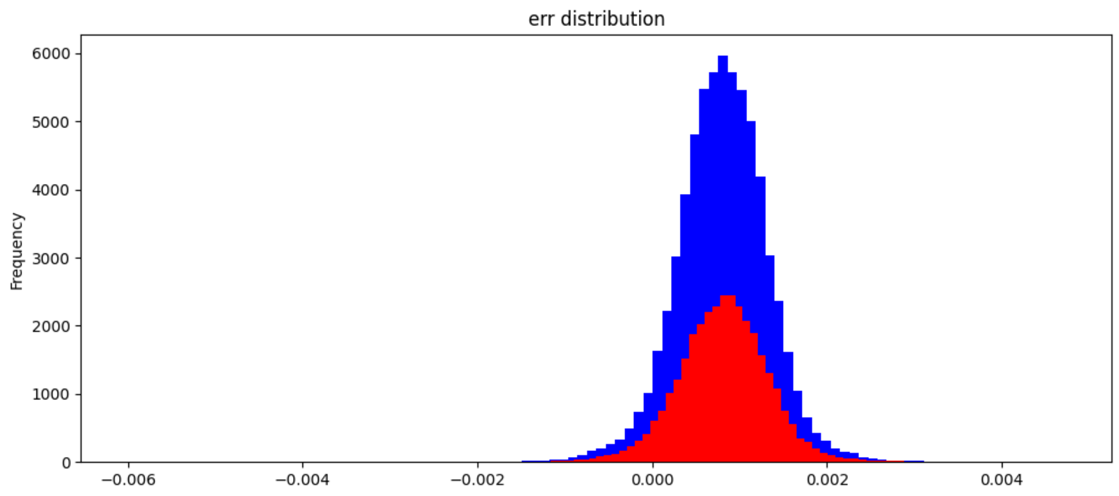
[37]:
```
import warnings
warnings.simplefilter('ignore')
```

```
display_nn_results(model, X_train_2, X_test_2, y_train_2, y_test_2,␣
 ↪resFile='training_2.png')
```

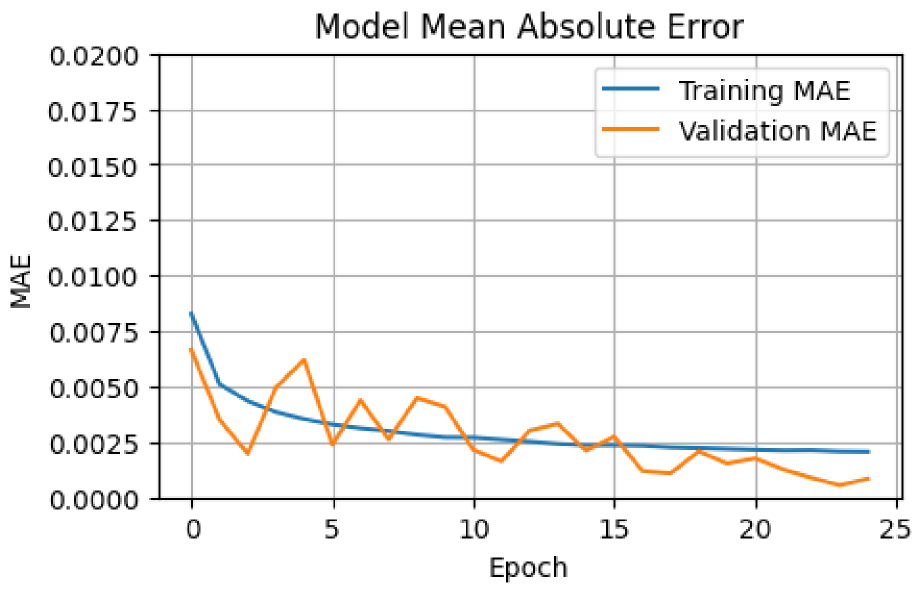
```
2094/2094  3s 2ms/step
@ InSample                : E[y-t]: 0.000845 Std(y-t): 0.000469
@
1032/1032  2s 2ms/step
@ OutOfSample             : E[y-t]: 0.000849 Std(y-t): 0.000471
@
@ Info        : results saved to 'training_2.png'
```

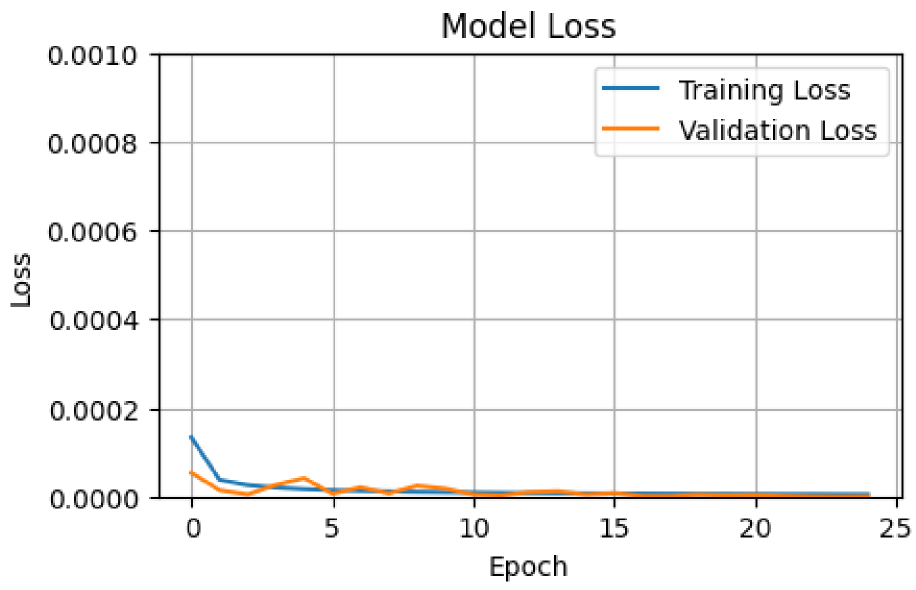


```
1032/1032  4s 4ms/step -
loss: 9.4269e-07 - mae: 8.4959e-04
Score:
[9.429476222067024e-07, 0.0008494004723615944]
```



[38]:
```
# Set fixed y-axis limits
Y_MIN = 0.0      # Typically zero for loss/MAE
Y_MAX_1 = 0.02    # Example: 2% MAE, you can adapt based on your expected errors
Y_MAX_2 = 0.001

# Plot MAE
plt.figure(figsize=(5,3))
plt.plot(history.history['mae'], label='Training MAE')
plt.plot(history.history['val_mae'], label='Validation MAE')
plt.title('Model Mean Absolute Error')
plt.ylabel('MAE')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)
plt.ylim([Y_MIN, Y_MAX_1])  # <- Fix y-axis
plt.show()

# Plot Loss
plt.figure(figsize=(5,3))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend()
plt.grid(True)
plt.ylim([Y_MIN, Y_MAX_2])  # <- Fix y-axis
plt.show()
```





1.7 Saving the Model

[39]:
```
TAG         = 'bs-0100000-050-T'
mdlDir      = "model_%s.keras" %(TAG)


model.save(mdlDir)
```

```python
print("@ %-24s: model saved to '%s'" %("Info", mdlDir))
```

```
@ Info                    : model saved to 'model_bs-0100000-050-T.keras'
```

1.8 Appendix

1.8.1 Why is $ M_T $ a martingale?

$ M_T $ is defined as:

$$M_T := \frac{e^{-r(T-t)} S_T}{S_t}$$

Now we check if $ (M_u)_{u $\geq t\}\$isamartingaleundertherisk-neutralmeasure\$\mathbb{Q}\$$.

Recall:

Under $ Q $, the discounted stock price $ (e^{-r(u-t)} S_u) $ is a martingale.

That is:

$$\mathbb{E}^{\mathbb{Q}} \left[e^{-r(u'-t)} S_{u'} \,\middle|\, \mathcal{F}_u \right] = e^{-r(u-t)} S_u \quad \text{for all} \quad t \leq u \leq u'$$

Thus, if we define:

$$M_u := \frac{e^{-r(u-t)} S_u}{S_t}$$

then:

$$\mathbb{E}^{\mathbb{Q}} \left[M_{u'} \,\middle|\, \mathcal{F}_u \right] = \mathbb{E}^{\mathbb{Q}} \left[\frac{e^{-r(u'-t)} S_{u'}}{S_t} \,\middle|\, \mathcal{F}_u \right] = \frac{1}{S_t} \mathbb{E}^{\mathbb{Q}} \left[e^{-r(u'-t)} S_{u'} \,\middle|\, \mathcal{F}_u \right]$$

but since $ e^{-r(u'-t)} S_{u'} $ is a martingale:

$$= \frac{1}{S_t} e^{-r(u-t)} S_u = e^{-r(u-t)} \frac{S_u}{S_t} = M_u$$

Thus:

$$\mathbb{E}^{\mathbb{Q}}[M_{u'} | \mathcal{F}_u] = M_u$$

meaning $ (M_u)_{u $\geq t\}\$isa$**martingale**.

1.8.2 Why do we say "martingale with zero interest rate"?

Under the **risk-neutral measure** \$ Q \$, it is a **basic fact** from financial theory that:

$$\left(e^{-r(u-t)}S_u\right)_{u\geq t} \quad \text{is a martingale.}$$

Why?

Because under \$ Q \$, \$ S_u \$ follows:

$$dS_u = rS_u\,du + \sigma S_u\,dW_u^{\mathbb{Q}}$$

and the process \$ e^{-r(u-t)} S_u \$ has exactly the drift term that cancels the risk-free rate \$ r \$.

When you compute \$ d(e^{-r(u-t)} S_u) \$ using Itô's lemma, you get:

$$d(e^{-r(u-t)}S_u) = e^{-r(u-t)}\sigma S_u\,dW_u^{\mathbb{Q}}$$

no drift term → it's a local martingale → and under mild conditions on \$ S_u \$, it's a true martingale.

Thus:

$$\mathbb{E}^{\mathbb{Q}}\left[e^{-r(u'-t)}S_{u'} \mid \mathcal{F}_u\right] = e^{-r(u-t)}S_u \quad \text{for all} \quad t \leq u \leq u'$$

Now, \$ M_u \$ is just \$ e^{-r(u-t)} S_u \$ normalized by \$ S_t \$

We have:

$$M_u = \frac{e^{-r(u-t)}S_u}{S_t}$$

Since \$ S_t \$ is **known** at time \$ t \$ (deterministic from the viewpoint of \$ t \$), dividing by \$ S_t \$ is just scaling by a constant.

Thus, if \$ e^{-r(u-t)} S_u \$ is a martingale, **then \$ M_u \$ is a martingale too** (because multiplying/dividing by a constant preserves the martingale property).

\$ M_u \$ behaves like the **price process** of an asset that:

- starts at \$ M_t = 1 \$,
- has **no drift**,
- only has volatility \$ σ\$,
- evolves according to a pure geometric Brownian motion **without any risk-free rate adjustment**.

Meaning: it's as if the underlying asset \$ M_u \$ evolves in a market where \$ r = 0 \$.

Specifically, \$ M_u \$ satisfies a **pure diffusion** SDE:

$$dM_u = \sigma M_u \, dW_u^{\mathbb{Q}}$$

(no $ r $ term, no drift, only randomness from $ dW_u^Q $).

In that world:

- The discounted price is the same as the non-discounted price (because $ r=0 $),
- You don't need to adjust cash flows by $ e^{-r(T-t)} $.

Thus, **$ M_u $ is a martingale in a world where the interest rate is 0**.

```
[ ]:
```