1 Introduction

Partial differential equations (PDEs) are often used to describe financial assets and other items in finance and other fields. But as new modeling problems have come up, new groups of equations, which include Kolmogorov partial differential equations, were additionally made. The **"curse of dimensionality"** makes it hard to find explicit solutions for most PDEs, especially in high dimensions. In order to solve this problem, our major goal is to employ deep neural networks to solve PDEs without the curse of dimensionality.

First we consider a stochastic differiential equation depend on a terminal condition:

$$X_t^x = x + \int_0^t \mu(X_s)\ ds + \int_0^t \sigma(X_s)\ dW_s, \qquad 0 \leq t \leq T \tag{1.1}$$

Using a time-discretization of the stochastic differential equation and straightforward Monte Carlo techniques, this problem can be solved numerically. Using the above approach, it is possible to develop efficient algorithms for determining the solutions of the Kolmogorov equations. These equations belong to a class of linear parabolic partial differential equations.

$$\begin{cases} \frac{\partial f}{\partial t}(t,x) = \frac{1}{2}\,\mathrm{Trace}_{\mathbb{R}^d}\left(\sigma(x)[\sigma(x)]^*\left(\mathrm{Hess}_x f\right)(t,x)\right) + \langle \mu(x), \left(\nabla_x f\right)(t,x)\rangle_{\mathbb{R}^d} \\ f(0,x) = \varphi(x) \end{cases} \tag{1.2}$$

for a fixed space-time point we can have an explicit form of the solution, due to Feynman-Kac formula:

$$f(T,x) = \mathbb{E}[\varphi(X_T^x)] \tag{1.3}$$

To put it another way, if we desire to determine the solutions at a lot of different points in space, we would have to simulate paths and computing the expectations for each of these places separately. As a result, the amount of work to do on the computer goes up in a way that is related to both the size of the space and the length of each interval. This means that such method can't be used in the real world because it would be too expensive to compute as the

space dimensions and gaps get bigger.
However, it has been recently shown that deep neural networks trained with stochastic gradient descent can overcome the curse of dimensionality [3, 6], making them a popular choice to solve this computational challenge in the last few years.
Our work is to improve such an algorithm, and find not only a solution in a fixed space-time point but in a subset of $\mathbb{R}^d$, hence we are looking for this quantity:

$$[a, b]^d \ni x \mapsto f(T, x) \tag{1.4}$$

To leverage deep neural networks, our approach is to reframe our problem as an optimization task. By setting up our problem as an optimization task, meaning having a function that we want to optimize and such function is often called the objective function. The objective function represents the goal we want to achieve or the criterion we want to optimize, and we show that the function above is the solution of the following optimization problem:

$$\min_{F \in C([a,b]^d, \mathbb{R})} \mathbb{E}[|F(X) - Y|^2] \tag{1.5}$$

with $(X, Y) = (X, \varphi(X_t^x))$ of independent samples drawn from the distribution of the data X this part will be discussed in chapter 2.
By employing a temporal discretization method, like the Euler-Maruyama scheme, we can obtain independent samples from the data distribution. These samples are represented as $(x_i, y_i)_{i=1}^m$, where m denotes the number of samples. Each sample is generated by simulating the stochastic differential equation at discrete time points, we arrive at the empirical Learning problem

$$\min_{F \in \mathrm{H}} \frac{1}{m} \sum_{i=1}^{m} |F(x_i) - y_i|^2 \tag{1.6}$$

over some space H. This thesis aims to solve a problem using machine learning algorithms, in particular deep neural networks. By combining different discretization, we aim to reduce computational costs and will be treated in chapter 4. However, for the comprehensive understanding of the reader, we will try to delve into understanding what is deep neural networks all about, particularly from a mathematical learning theory perspective and even from the practical point of view, this exploration will extend throughout the 3th chapter concluding with some examples.

2 Mathematical Learning Problem

2.1 Main Problem

Let $T \in (0, \infty), d \in \mathbb{N}$, let $\mu : \mathbb{R}^d \to \mathbb{R}^d$ and $\sigma : \mathbb{R}^d \to \mathbb{R}^{d \times d}$ be Lipschitz continuous functions, let $\varphi : \mathbb{R}^d \to \mathbb{R}$ be a function, and let $f = (f(t,x) \in C^{1,2}([0,T] \times \mathbb{R}^d, \mathbb{R})$ be a function with at most polynomially growing partial derivatives which satisfies for every $t \in (0,T], x \in \mathbb{R}^d$ that $f(0,x) = \varphi(x)$ and

$$\frac{\partial f}{\partial t}(t,x) = \frac{1}{2} \operatorname{Trace}_{\mathbb{R}^d}\left(\sigma(x)[\sigma(x)]^T \left(\operatorname{Hess}_x f\right)(t,x)\right) + \langle \mu(x), \left(\nabla_x f\right)(t,x) \rangle_{\mathbb{R}^d}. \tag{2.1}$$

Our goal is to approximately calculate the function $\mathbb{R}^d \ni x \mapsto f(T,x) \in \mathbb{R}$ on some subset of $\mathbb{R}^d$. To fix ideas we consider real numbers $a, b \in \mathbb{R}$ with $a < b$ and we suppose that our goal is to approximately calculate the function $[a,b]^d \ni x \mapsto f(T,x) \in \mathbb{R}$.

2.2 Probabilistic Solution

Let $(\Omega, \mathcal{F}, \mathbb{P})$ be a probability space with filtration $(\mathcal{F}_t)_{t \in [0,T]}$ generated by a standard Brownian motion $(W_t)_{t \in [0,T]}$. Moreover, let $(X_t^x)_{t \in [0,T]} : [0,T] \times \Omega \to \mathbb{R}$ be a time-homogeneous Itô diffusion process that is $(X_t^x)_{t \in [0,T]}$ is the solution to the following Itô SDE

$$X_t^x = x + \int_0^t \mu\left(X_s^x\right) ds + \int_0^t \sigma\left(X_s^x\right) dW_s \quad \mathbb{P} \text{ - as.} \tag{2.2}$$

by using the **Feynman-Kac formula** that there exists a unique solution $f = (f(t,x))_{(t,x) \in [0,T] \times \mathbb{R}^d} \in C^{1,2}\left([0,T] \times \mathbb{R}^d, \mathbb{R}\right)$ such that for every $x \in \mathbb{R}^d$ it holds that

$$f(T,x) = \mathbb{E}\left[f\left(0, X_T^x\right)\right] = \mathbb{E}\left[\varphi\left(X_T^x\right)\right] \tag{2.3}$$

2.3 Formulation as a minimization problem

In this first step, we intend to exploit the L^2-minimization property of the expectation of a real-valued random variable. That is, for any random variable X with $\mathbb{E}\left[|X|^2\right] < \infty$ and for any $y \in \mathbb{R}$ it holds that there exists a unique real number $z \in \mathbb{R}$ such that

$$\mathbb{E}\left[|X - z|^2\right] = \inf_{y \in \mathbb{R}} \mathbb{E}\left[|X - y|^2\right]$$

As shown in the paper, this can be extended from random variables to **random fields** 2.3.1. Thus, we can formulate the solution to the PDE 2.1 as given by 2.7 as the following minimisation problem

$$\mathbb{E}\|\varphi\left(\mathbb{X}_T\right) - f(T,\xi)|^2\Big] = \inf_{\nu \in C\left([a,b]^d, \mathbb{R}\right)} \mathbb{E}\left[|\varphi\left(\mathbb{X}_T\right) - \nu(\xi)|^2\right] \tag{2.4}$$

where $\xi \sim U\left([a,b]^d\right)$. As we prove in this work, this minimization problem is uniquely solved by

$$[a,b]^d \ni x \mapsto f(T,x) = \mathbb{E}\left[\varphi\left(X_T^x\right)\right] \in \mathbb{R}. \tag{2.5}$$

In other words, we leverage equation 2.7 to formulate a minimization problem that admits a unique solution by the function $[a,b]^d \ni x \mapsto f(T,x) \in \mathbb{R}$. To accomplish this, we first revisit the L^2-minimization property of the expectation for a real-valued random variable. We then extend this minimization concept to certain random fields. Subsequently, we apply Proposition 2.3.1 to random fields within the framework of the Feynman-Kac representation 2.7, leading to the derivation of Proposition 2.3.2.

Proposition 2.3.2 introduces a minimization problem for which the function $[a,b]^d \ni x \mapsto f(T,x) \in \mathbb{R}$ stands as the unique global minimizer.

Our proof of Proposition 2.3.2 builds upon the elementary auxiliary results found in Lemmas [2.3.2–2.3.5]. To ensure completeness, we try provide the proofs and references that guides reader to good understanding in this context.

We first define what a random field really mean, random field s simply a stochastic process, taking values in a Euclidean space, and defined over a parameter space of dimensionality at least one. Mathematicaly says:

5 Examples

Remark 5.0.1

For the following examples we uses SGD optimization algorithm using the following parametre in the the context of 4.3:
Let $(\gamma_m)_{m\in\mathbb{N}} \subset (0,\infty)$*, and assume that for all* $m \in \mathbb{N}$*,* $x \in \mathbb{R}^\rho$*,* $\Phi_m \in (\mathbb{R}^\rho)$ *such that* $\nu = \rho$*,*

$$\Psi_m(x, \Phi_m) = \Phi_m, \tag{5.1}$$

and

$$\Phi_m(x) = \gamma_m x. \tag{5.2}$$

Then it holds for all $m \in \mathbb{N}$ *that*

$$\Theta_{m+1} = \Theta_m - \gamma_{m+1} (\nabla \phi^m)(\Theta_m). \tag{5.3}$$

5.1 Geometric Brownian Motion

we set the numerical setting:

Let $r = \frac{1}{20}$, $\mu = r - \frac{1}{10} = -\frac{1}{20}$, $\sigma_1 = \frac{1}{10} + \frac{1}{200}$, $\sigma_2 = \frac{1}{10} + \frac{2}{200}$, …, $\sigma_{100} = \frac{1}{10} + \frac{100}{200}$.
Assume for every $s, t \in [0,T]$, $x = (x_1, x_2, \ldots, x_d)$, $w = (w_1, w_2, \ldots, w_d) \in \mathbb{R}^d$, $m \in \mathbb{N}_0$ that $N = 1$, $d = 100$, $\varphi(x) = \exp(-rT)\max\left[\max_{i\in\{1,2,\ldots,d\}} x_i - 100, 0\right]$, and

$$H(s,t,x,w) = \left(x_1 \exp\left(\left(\mu_1 - \frac{|\sigma_1|^2}{2}\right)(t-s) + \sigma_1 w_1\right), \ldots, x_d \exp\left(\left(\mu_d - \frac{|\sigma_d|^2}{2}\right)(t-s) + \sigma_d w_d\right)\right), \tag{5.4}$$

Assume that $\xi^{0,1} : \Omega \to \mathbb{R}^d$ is continuous and uniformly distributed on $[90, 110]^d$. Let $f = (f(t,x))_{(t,x)\in[0,T]\times\mathbb{R}^d} \in C^{1,2}([0,T]\times\mathbb{R}^d, \mathbb{R})$ be an at most polynomially growing function satisfying for every $t \in [0,T]$ and $x \in \mathbb{R}^d$ that $u(0,x) = \varphi(x)$.

$$\frac{\partial f}{\partial t}(t,x) = \frac{1}{2}\sum_{i=1}^{d} |\sigma_i x_i|^2 (\frac{\partial^2 f}{\partial x_i^2}(t,x)) + \sum_{i=1}^{d} \mu_i x_i (\frac{\partial f}{\partial x_i}(t,x)) \tag{5.5}$$

The Feynman-Kac formula 4.1.1 shows that for every standard Brownian motion $W = (W^{(1)}, \ldots, W^{(d)})$ $[0,T]\times\Omega \to \mathbb{R}^d$ and every $t \in [0,T]$, $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, it holds that

$$f(t,x) = \mathbb{E}\left[\varphi\left(x_1 \exp\left(\sigma_1 W_t^{(1)} + (\mu_1 - \frac{|\sigma_1|^2}{2})t\right), \ldots, x_d \exp\left(\sigma_d W_t^{(d)} + (\mu_d - \frac{|\sigma_d|^2}{2})t\right)\right)\right]. \tag{5.6}$$

In this case we use the **Relu** activation function along with 200 neuron in each hidden layer and we used 2 hidden layers, and we set the learning rate of SGD 5.0.1 as $\gamma_m = 10^{-3}\mathbb{1}_{[0,1000]}(m) + 10^{-4}\mathbb{1}_{(1000,2000]}(m) + 10^{-5}\mathbb{1}_{[2000,\infty)}(m)$ and $J_m = 256$ and all weights in the neural network are initialized by means of the Xavier initialization.

Figure 5.1: Loss on Training and Validation data for the Feynman-Kac solver in case of PDE (5.5)

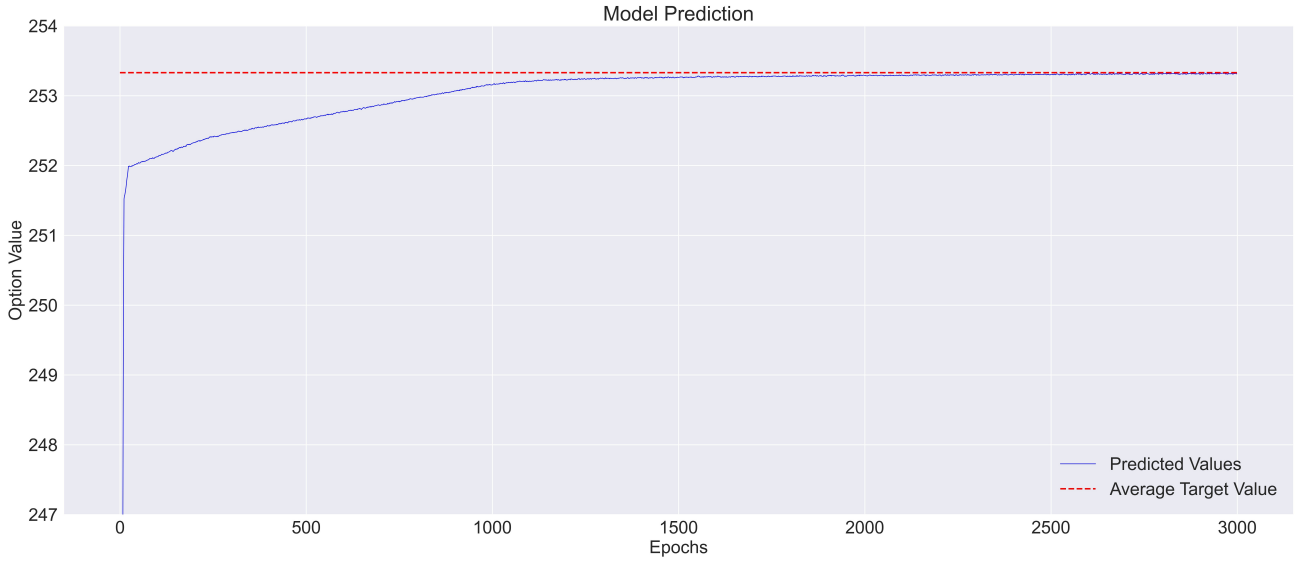

Figure 5.2: Model prediction over epoches of PDE (5.5)

Num of iterations	Train Loss	Validation Loss	Runtime (seconds)
0	48882.2304	21144.8085	4.6031
1000	0.4280	0.4391	607.11
1500	0.3307	0.3544	1559.201
2000	0.3174	0.3305	3483.820
2500	0.3006	0.3184	3854.704
3000	0.2804	0.2991	4213.682

Table 5.1: Simulations of the Feynman-Kac solver in the case of PDE (5.5)

After the training process, we check our model with train, test, and validation data to see if it fits well and can be used for data it hasn't seen before:

	Accuracy	Mean Absolute Error (MAE)	Mean Squared Error (MSE)
Training Data	99.0912	0.1505	0.0358
Validation Data	99.0589	0.1681	0.0440
Test Data	99.0912	0.1678	0.0444

Table 5.2: Different metrics of the Feynman-Kac solver in the case of PDE (5.5)

5.2 The multi-asset Black-Scholes model

we set the numerical setting:
Let $r = \frac{1}{20}, \mu = r - \frac{1}{10} = -\frac{1}{20}, \beta_1 = \frac{1}{10} + \frac{1}{200}, \beta_2 = \frac{1}{10} + \frac{2}{200}, \ldots, \beta_{100} = \frac{1}{10} + \frac{100}{200}, Q = (Q_{i,j})_{(i,j)\in\{1,2,\ldots,100\}}, \Sigma = (\Sigma_{i,j})_{(i,j)\in\{1,2,\ldots,100\}} \in \mathbb{R}^{100\times 100}, \varsigma_1, \varsigma_2, \ldots, \varsigma_{100} \in \mathbb{R}^{100}$, assume for every $s, t \in [0,T], x = (x_1, x_2, \ldots, x_d), w = (w_1, w_2, \ldots, w_d) \in \mathbb{R}^d, m \in \mathbb{N}_0, i, j, k \in \{1, 2, \ldots, 100\}$ with $i < j$ that $N = 1, d = 100, Q_{k,k} = 1, Q_{i,j} = Q_{j,i} = \frac{1}{2}, \Sigma_{i,j} = 0, \Sigma_{k,k} > 0, \Sigma\Sigma^* = Q$ So we can see that for initializing the corolation the vector ξ_i that the inner product model the correlation matrix we need to find the explicit form of Σ, we propose the scholesky decomposition to do so ([14], Theorem 4.2.5]) and of course the computing of such will be delivered in the code python in the last remaine section, $\varsigma_k = (\Sigma_{k,1}, \ldots, \Sigma_{k,100}), \varphi(x) = \exp(-\mu T)\max\{110 - \left[\min_{i\in\{1,2,\ldots,d\}} x_i\right], 0\}$, and

$$\begin{aligned} H(s,t,x,w) = &\left(x_1 \exp\left(\left(\mu - \frac{1}{2}\left\|\beta_1\varsigma_1\right\|^2_{\mathbb{R}^d}\right)(t-s) + \langle\varsigma_1, w\rangle_{\mathbb{R}^d}\right), \ldots,\right. \\ &\left. x_d \exp\left(\left(\mu - \frac{1}{2}\left\|\beta_d\varsigma_d\right\|^2_{\mathbb{R}^d}\right)(t-s) + \langle\varsigma_d, w\rangle_{\mathbb{R}^d}\right)\right), \end{aligned} \tag{5.7}$$

Assume that $\xi^{0,1} : \Omega \to \mathbb{R}^d$ are continuous uniformly distributed on $[90, 110]^d$, and let $u = (u(t,x))$ for $t \in [0,T]$ and $x \in \mathbb{R}^d \in C^{1,2}([0,T]\times\mathbb{R}^d, \mathbb{R})$ be a function that satisfies, for every $t \in [0,T]$ and $x \in \mathbb{R}^d$, the initial condition $u(0,x) = \varphi(x)$ and the following partial differential equation:

$$\frac{\partial u}{\partial t}(t,x) = \frac{1}{2}\sum_{i,j=1}^{d} x_i x_j \beta_i \beta_j \langle\varsigma_i, \varsigma_j\rangle_{\mathbb{R}^d} \frac{\partial^2 u}{\partial x_i \partial x_j}(t,x) + \sum_{i=1}^{d} \mu_i x_i \frac{\partial u}{\partial x_i}(t,x). \tag{5.8}$$

The Feynman-Kac formula 4.1.1 shows that for every standard Brownian motion $W = (W^{(1)}, \ldots, W^{(d)})$ $[0,T]\times\Omega \to \mathbb{R}^d$ and every $t \in [0,T]$, $x = (x_1, \ldots, x_d) \in \mathbb{R}^d$, it holds that

$$\begin{aligned} u(t,x) = \mathbb{E}\Big[\varphi\Big(x_1 \exp\Big(\left\langle\varsigma_1, W_t^{(1)}\right\rangle_{\mathbb{R}^d} + \left(\mu_1 - \frac{\left\|\beta_1\varsigma_1\right\|^2_{\mathbb{R}^d}}{2}\right)t\Big), \ldots,\\ x_d \exp\left(\left\langle\varsigma_d, W_t^{(d)}\right\rangle_{\mathbb{R}^d} + \left(\mu_d - \frac{\left\|\beta_d\varsigma_d\right\|^2_{\mathbb{R}^d}}{2}\right)t\right)\Big)\Big]. \end{aligned} \tag{5.9}$$

In this case we use the **tanh** activation function along with 200 neuron in each hidden layer and we used 2 hidden layers, and we set the learning rate of SGD 5.0.1 as $\gamma_m = 10^{-3}\mathbb{1}_{[0,100]}(m) + 10^{-4}\mathbb{1}_{(100,200]}(m) + 10^{-5}\mathbb{1}_{[200,\infty)}(m)$ and $J_m = 256$ and all weights in the neural network are

Figure 5.3: Loss on Training and Validation data for the Feynman-Kac solver in case of PDE (5.8)

initialized by means of the Xavier initialization.

Num of iterations	Train Loss	Validation Loss	Runtime (seconds)
0	700.854	36.047	15.807
100	0.0054	0.0053	1387.413
200	0.0044	0.0043	2798.796
300	0.0035	0.0034	4263.372
400	0.0028	0.0028	5678.744
500	0.0024	0.0023	7200.931

Table 5.3: Simulations of the Feynman-Kac solver in the case of PDE (5.8)

	Accuracy	Mean Absolute Error (MAE)	Mean Squared Error (MSE)
Training Data	87.1290	0.0389	0.0024
Validation Data	86.8121	0.0387	0.0024
Test Data	86.8121	0.0388	0.0024

Table 5.4: Different metrics of the Feynman-Kac solver in the case of PDE (5.8)

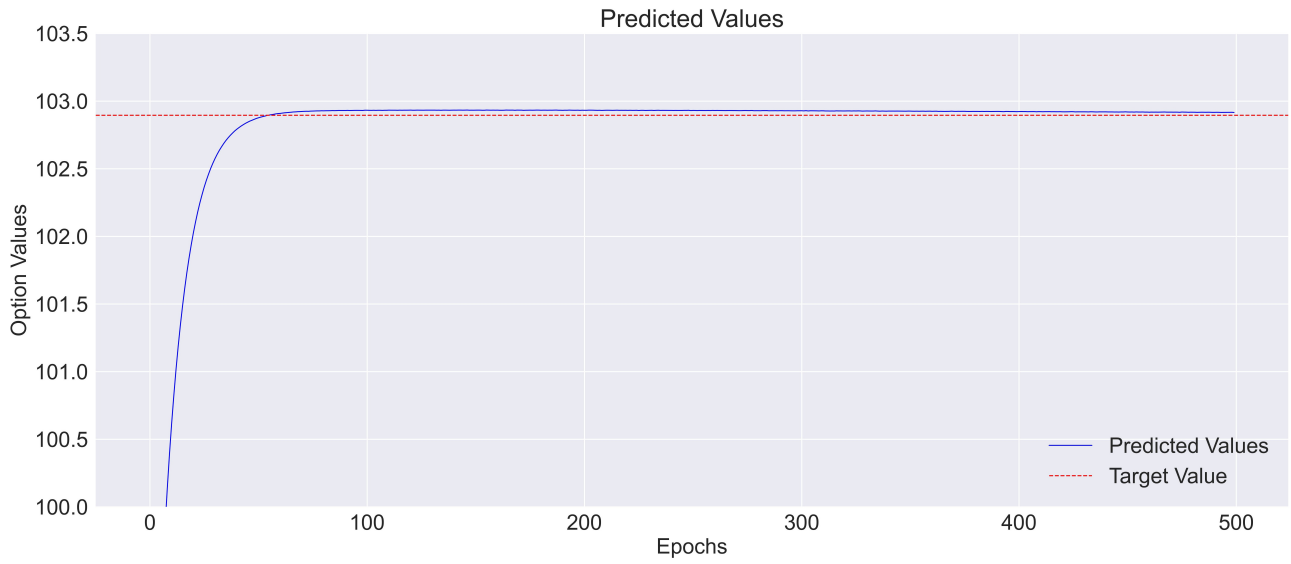

Figure 5.4: Model prediction over epoches of PDE (5.8)

5.3 Python code sources

The numerical experiments detailed below were executed using Python with TensorFlow on a Windows PC featuring an intel $i5-10310U$ CPU operating at 2.21 Gigahertz (GHz) and equipped with 16 gigabytes (GB) of 2666 Megahertz (MHz) double data rate type four random-access memory (DDR4-RAM).

Geometric Brownian Motion

```
from keras.models import Sequential
from keras.layers import Dense, BatchNormalization, Activation, Input
from tensorflow.keras.initializers import GlorotUniform
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import tensorflow as tf
import pandas as pd
from sklearn.metrics import r2_score
from keras.callbacks import Callback
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
import time
# Set data type
DTYPE='float32'
#DTYPE='float64'
tf.keras.backend.set_floatx(DTYPE)
print('TensorFlow version used: {}'.format(tf.__version__))
```

```
# Final time
T = tf.constant(1., dtype=DTYPE)

# Spatial dimensions
dim = 100

# Domain-of-interest at t=0
a = 90 * tf.ones((dim), dtype=DTYPE)
b = 110 * tf.ones((dim), dtype=DTYPE)

# Interest rate
r = tf.constant(1./20, dtype=DTYPE)

# Drift
mu = tf.constant(-1./20, dtype=DTYPE)

# Strike price
K = tf.constant(100., dtype=DTYPE)

# Diffusion/volatility
sigma = 1./10 + 1./200*tf.range(1, dim+1, dtype=DTYPE)

# Define terminal condition, i.e., payoff at maturity
def fun_g(x):
    return tf.exp(-r*T) * tf.maximum(tf.reduce_max(x, axis=1, keepdims=True)
   , K)
```

```

def draw_X(num_samples, a, b):
    dim = a.shape[0]

    X0 = a + tf.random.uniform((num_samples, dim), dtype=DTYPE) * (b-a)
    xi = tf.random.normal(shape=(num_samples, dim), dtype=DTYPE)
    XT = X0 * tf.exp( (mu- tf.square(sigma)/2) * T + sigma * tf.sqrt(T) * xi
   )

```

```
    # Return simulated paths
    return tf.stack([X0, XT], 2)
```

```

def init_model(dim, activation='relu',
               num_hidden_neurons=200,
               num_hidden_layers=2,
               initializer=GlorotUniform()):
    model = Sequential()
    model.add(Input(shape=(dim,)))
    model.add(BatchNormalization(epsilon=1e-6))

    # Create a fixed number of hidden layers
    for _ in range(num_hidden_layers):
        model.add(Dense(num_hidden_neurons,
                        activation=None,
                        use_bias=False,
                        kernel_initializer=initializer))
        model.add(BatchNormalization(epsilon=1e-6))
        model.add(Activation(activation))

    model.add(Dense(1,
                    activation=None,
                    use_bias=False,
                    kernel_initializer=initializer))
    model.add(BatchNormalization(epsilon=1e-6))
    return model
```

```

n_test = 80000
X = draw_X(n_test, a, b)
Xtest = X[:, :, 0]
Xtest = tf.convert_to_tensor(Xtest, dtype=DTYPE)
Ytest = tf.zeros((Xtest.shape[0], 1), dtype=DTYPE)  # Create an initial
    Ytest tensor filled with zeros
b_size = 32
mc_samples = 4000

def mc_step(y):

    Xi = tf.random.normal(shape=(b_size, dim), dtype=DTYPE)
    upd = tf.exp((mu - tf.square(sigma) / 2) * T + sigma * tf.sqrt(T) * Xi)
    XT = tf.reshape(Xtest, shape=[n_test, dim, 1]) * tf.transpose(upd)
```

```
    expected_payoff = tf.reduce_sum(tf.reshape(fun_g(XT), [n_test, b_size]),
    axis=1, keepdims=True)
    return y + expected_payoff / (b_size * mc_samples)


for i in range(mc_samples):
    Ytest = mc_step(Ytest)
```

```
Ytest = np.array(Ytest)
df = pd.DataFrame(Xtest)
df['Ytest'] = Ytest
X_train_val, X_test, y_train_val, y_test = train_test_split(df.drop(columns
   =['Ytest']), df.loc[:, 'Ytest'], test_size=0.2)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val,
   test_size=0.5)
```

```

class PredictionHistory(Callback):
    def __init__(self, X_val):
        super().__init__()
        self.X_val = X_val
        self.predictions = []

    def on_epoch_end(self, epoch, logs={}):
        self.predictions.append(self.model.predict(self.X_val))
```

```
# Instantiate the callback
pred_history = PredictionHistory(X_val)

# Fit the model with the custom callback
start_time = time.time()
history = model.fit(X_train, y_train, batch_size=128, epochs=3000,
   validation_data=(X_val, y_val), callbacks=[pred_history])
end_time = time.time()
total_time = end_time - start_time


# Print the total time
print(f"Total time for model fitting: {total_time/3600:.3f} hours")

#The following code is for saving data traning in csv file.
# Convert the training history to a DataFrame
#history_df = pd.DataFrame(history.history)
```

```

# Add a column for the time taken for training
#history_df['training_time'] = total_time

# Save the history to a CSV file
#history_df.to_csv('training_history2.csv', index=False)
```

```
# Get the predictions for all epochs
all_epoch_predictions = np.array(pred_history.predictions)
plt.figure(figsize=(25, 10))
# Csalculate the average across all data points for each epoch
average_predictions = np.mean(all_epoch_predictions, axis=1)

# Plot the average predictions over epochs
plt.plot(average_predictions[1:])
plt.title('Average Model Predictions Over Epochs ')
plt.xlabel('Epoch')
plt.ylabel('Average Predicted Value')
plt.show()
```

```
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np
from sklearn.metrics import mean_absolute_error
# Assuming you have already trained your model and obtained predictions for
    both validation and test data
y_pred_val = model.predict(X_val)  # Predictions for validation data
y_pred_test = model.predict(X_test)  # Predictions for test data
y_pred_train = model.predict(X_train)  # Predictions for training data


# Calculate R-squared for both sets
r_squared_val = r2_score(y_val, y_pred_val)
r_squared_test = r2_score(y_test, y_pred_test)
r_squared_train = r2_score(y_train, y_pred_train)


# Calculate MAE for all three datasets
mae_val = mean_absolute_error(y_val, y_pred_val)
mae_test = mean_absolute_error(y_test, y_pred_test)
mae_train = mean_absolute_error(y_train, y_pred_train)

# Calculate MSE for all three datasets
mse_val = mean_squared_error(y_val, y_pred_val)
```

```
mse_test = mean_squared_error(y_test, y_pred_test)
mse_train = mean_squared_error(y_train, y_pred_train)



print(f"Accuracy on Validation Data: {r_squared_val * 100}")
print(f"Accuracy on Test Data: {r_squared_test * 100}")
print(f"Accuracy on Train Data: {r_squared_test * 100}")

print(f"Mean Absolute Error (MAE) on Validation Data: {mae_val:.4f}")
print(f"Mean Absolute Error (MAE) on Test Data: {mae_test:.4f}")
print(f"Mean Absolute Error (MAE) on Training Data: {mae_train:.4f}")

print(f"Mean Squared Error (MSE) on Validation Data: {mse_val:.4f}")
print(f"Mean Squared Error (MSE) on Test Data: {mse_test:.4f}")
print(f"Mean Squared Error (MSE) on Training Data: {mse_train:.4f}")
```

The multi-asset Black-Scholes model

```
# Final time
T = tf.constant(1., dtype=DTYPE)

# Spatial dimensions
dim = 100

# Domain-of-interest at t=0
a = 90 * tf.ones((dim), dtype=DTYPE)
b = 110 * tf.ones((dim), dtype=DTYPE)

# Interest rate
r = tf.constant(1./20, dtype=DTYPE)

# Drift
mu = tf.constant(-1./20, dtype=DTYPE)

# Strike price
K = tf.constant(100., dtype=DTYPE)

# Diffusion/volatility
beta_i = 1./10 + 1./200*tf.range(1, dim+1, dtype=DTYPE)

```

```
# Define terminal condition, i.e., payoff at maturity
# Define the payoff function phi
def phi(x):
    return tf.exp(-mu * T) * tf.maximum(110 - tf.reduce_min(x, axis=1,
    keepdims=True), 0)
```

```
# Define a correlation matrix Q
Q = np.ones([dim, dim]) * 0.5
np.fill_diagonal(Q, 1.)

# Perform Cholesky decomposition on Q to get sigma (volatility) and
    sigma_norms
L = np.linalg.cholesky(Q)
sigma_norms = tf.constant(np.linalg.norm(L, axis=0), dtype=DTYPE)
sigma = tf.constant(L, dtype=DTYPE)
```

```
def calculate_H(num_samples, a, b):
    """ Function to calculate H for num_samples many pairs of uniformly
    drawn starting
    values X_0 and end points X_T of a stochastic process X with zero drift
    and
    constant scalar diffusion. Starting points are drawn uniformly from the
    hypercube [a,b] \\subset \mathbb{R}^d. """
    # Create a 100x100 matrix with diagonal elements of 1 and off-diagonal
    elements of 1/2

    dim = a.shape[0]
    X0 = a + tf.random.uniform((num_samples, dim), dtype=DTYPE) * (b - a)
    Xi = tf.random.normal(shape=(num_samples, dim), dtype=DTYPE)
    XT = X0 * tf.exp((mu - 1/2 * (beta_i * sigma_norms) ** 2) * T +  tf.
    matmul(tf.sqrt(T)* Xi, sigma))

    # Stack the list of XT values along a new dimension to create the final
    output

    return tf.stack([X0, XT],2)
```

```
n_test1 = 200000
X1 = calculate_H(n_test1, a, b)
Xtest1 = X1[:, :, 0]
Xtest1 = tf.convert_to_tensor(Xtest1, dtype=DTYPE)
Y = tf.zeros((Xtest1.shape[0], 1), dtype=DTYPE)  # Create an initial Ytest
    tensor filled with zeros
```

```
b_size = 32
mc_samples1 = 4000

def mc_samples2(y):
    Xi = tf.random.normal(shape=(b_size, dim), dtype=DTYPE)
    upd = tf.exp((mu - 1/2 * (beta_i * sigma_norms) ** 2) * T +  tf.matmul(
   tf.sqrt(T)* Xi, sigma))
    XT = tf.reshape(Xtest1, shape=[n_test1, dim, 1]) * tf.transpose(upd)
    expected_payoff = tf.reduce_sum(tf.reshape(phi(XT), [n_test1, b_size]),
   axis=1, keepdims=True)
    return y + expected_payoff / (b_size * mc_samples1)

for i in range(mc_samples1):
    Y = mc_samples2(Y)
```

```
Y = np.array(Y)
df = pd.DataFrame(Xtest1)
df['Y'] = Y
X_train_val1, X_test1, y_train_val1, y_test1 = train_test_split(df.drop(
   columns=['Y']), df.loc[:, 'Y'], test_size=0.2)
X_train1, X_val1, y_train1, y_val1 = train_test_split(X_train_val1,
   y_train_val1, test_size=0.5)
```

```
def init_model(dim, activation='tanh',
               num_hidden_neurons=200,
               num_hidden_layers=3,
               initializer=GlorotUniform()):
    model = Sequential()
    model.add(Input(shape=(dim,)))
    model.add(BatchNormalization(epsilon=1e-6))

    # Create a fixed number of hidden layers
    for _ in range(num_hidden_layers):
        model.add(Dense(num_hidden_neurons,
                        activation=None,
                        use_bias=False,
                        kernel_initializer=initializer))
        model.add(BatchNormalization(epsilon=1e-6))
        model.add(Activation(activation))

    model.add(Dense(1,
                    activation=None,
                    use_bias=False,
```

```
                    kernel_initializer=initializer))
    model.add(BatchNormalization(epsilon=1e-6))
    return model
```

```
# Define the model
model = init_model(dim=dim)

# Learning rate schedule and optimizer
lr = tf.keras.optimizers.schedules.PiecewiseConstantDecay([750, 1500], [1e
    -3, 1e-4, 1e-5])
optimizer = tf.keras.optimizers.SGD(learning_rate=lr, momentum=0.9)
#optimizer = tf.keras.optimizers.Adam(learning_rate=lr, epsilon=1e-8)


# Compile the model
model.compile(optimizer=optimizer, loss='mean_squared_error')
model.summary()
```

```
# Train the model
start_time = time.time()
history2 = model.fit(X_train1, y_train1, batch_size=256, epochs=3000,
    validation_data=(X_val1, y_val1))
end_time = time.time()
total_time = end_time - start_time

# Print the total time
print(f"Total time for model fitting: {total_time/3600:.3f} hours")

#The following code is for saving data traning in csv file.
# Convert the training history to a DataFrame
#history2_df = pd.DataFrame(history.history)

# Add a column for the time taken for training
#history2_df['training_time'] = total_time

# Save the history to a CSV file
#history2_df.to_csv('training_history2.csv', index=False)
```

```
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np
from sklearn.metrics import mean_absolute_error

```

```
# Assuming you have already trained your model and obtained predictions for
    both validation and test data
y_pred_val1 = model.predict(X_val1)  # Predictions for validation data
y_pred_test1 = model.predict(X_test1)  # Predictions for test data
y_pred_train1 = model.predict(X_train1)  # Predictions for training data


# Calculate R-squared for both sets
r_squared_val1 = r2_score(y_val1, y_pred_val1)
r_squared_test1 = r2_score(y_test1, y_pred_test1)
r_squared_train1 = r2_score(y_train1, y_pred_train1)

# Calculate MAE for all three datasets
mae_val1 = mean_absolute_error(y_val1, y_pred_val1)
mae_test1 = mean_absolute_error(y_test1, y_pred_test1)
mae_train1 = mean_absolute_error(y_train1, y_pred_train1)

# Calculate MSE for all three datasets
mse_val1 = mean_squared_error(y_val1, y_pred_val1)
mse_test1 = mean_squared_error(y_test1, y_pred_test1)
mse_train1 = mean_squared_error(y_train1, y_pred_train1)



print(f"Accuracy on Validation Data: {r_squared_val1 * 100}")
print(f"Accuracy on Test Data: {r_squared_test1 * 100}")
print(f"Accuracy on Train Data: {r_squared_test1 * 100}")

print(f"Mean Absolute Error (MAE) on Validation Data: {mae_val1:.4f}")
print(f"Mean Absolute Error (MAE) on Test Data: {mae_test1:.4f}")
print(f"Mean Absolute Error (MAE) on Training Data: {mae_train1:.4f}")

print(f"Mean Squared Error (MSE) on Validation Data: {mse_val1:.4f}")
print(f"Mean Squared Error (MSE) on Test Data: {mse_test1:.4f}")
print(f"Mean Squared Error (MSE) on Training Data: {mse_train1:.4f}")
```