Monte Carlo learning

Daniel Roth daniel-roth@posteo.com

Abstract

The aims of this project can be classified into two primary categories.

- Firstly, to develop a flexible framework for regression-based neural network training tasks, we will investigate two primary classes of data input/generation methods. The first method involves using general file inputs, allowing for the use of precomputed data during training. The second method involves implementing individual functions that generate data on-the-fly during training, providing greater adaptability and customization.
- Secondly, we aim to investigate the properties of neural networks concerning the use of sampled or approximated results as labels for randomly selected inputs during training. We will refer to this approach as Monte Carlo learning, which will be discussed in detail in section 2. Unlike conventional machine learning techniques that use a finite test set, Monte Carlo learning can generate an unlimited amount of data, enabling neural networks to be evaluated on a continuous range of inputs.

To demonstrate these concepts, we will present a series of examples with increasing complexity in the first section. The code for each example will be made available on GitHub. Our objective is to gain a comprehensive understanding of Monte Carlo learning and its potential applications in neural network training tasks.

Contents

1. Example: Cumulative density function: Random selection versus random inputs as training data

To begin, we consider a one-dimensional training set and focus on the cumulative density function (CDF) Φ of a standard normal distribution. Our goal is to train a neural network that can approximate this function effectively. For simplicity, we limit the input domain and study the network training for a restricted range of inputs. Therefore, we aim to replace the function

$$\Phi : [-4, 4] \to [0, 1],$$

with a neural network

$$\mathcal{N} : [-4, 4] \to [0, 1].$$

As a reference, the integral $\int_{-4}^{4} \Phi(x)\mathrm{d}x = 4$ (and 4.01 for $[-4.01, 4.01]$).

In the following four examples, which include code provided in the repository, we will examine various approaches for selecting training data and analyze their impact on the neural network's performance. By using identical network architectures and training hyperparameters across all examples, we will gain insights into the influence of the integration error of (2) on neural network training.
The implementations of these examples can be found in the GitHub repo, and a Google Colab executable link with all examples is available Colab executable

For further code explanations and general remarks on using the framework, please refer to section 3

1.1 Code example: non-randomized

In the first training example, we discretize the input interval using equidistant steps with different step sizes and train the network with the resulting (increasingly sorted) arrays. This approach, while simple, has its limitations.
When the training batches are taken without randomization, the network starts with the smallest input numbers, leading to a skewed representation of the data. In this case, since the cumulative density function (CDF) of small inputs is zero, the network initially learns a constant value for these inputs.

As the training progresses and the inputs increase, the network should ideally adapt to the changing outputs. However, due to the lack of randomization in the training data, the network may have already overfit the constant value it learned earlier, making it harder to adjust to the new patterns in the data.
This issue is exacerbated by the fact that the CDF of the inputs tends towards one as the inputs increase, leading to a significant shift in the output distribution. The network, having overfit the constant zero value from earlier inputs, struggles to adapt to this new distribution and ultimately fails to learn the desired function.
Observation:

- The final trained network is constantly zero, indicating that the network fails to learn the desired function due to the non-randomized training approach, which causes overfitting and hinders the network's ability to adjust to changing input patterns.

1.2 Code example: randomized

There are several approaches to overcome the issue observed in the first example. For instance, using smaller learning rates can help prevent the network from overfitting the constant value it learns from the initial inputs. Alternatively, employing other deterministic batch partition methods may ensure a more diverse representation of the data in each batch, allowing the network to learn the desired function more effectively.
The simplest approach, however, is to randomize the training set. By shuffling the input data before creating the training batches, the network is exposed to a more representative mix of inputs and corresponding outputs during each training step. This helps prevent overfitting and ensures that the network learns the underlying patterns in the data more effectively.
It is worth noting, however, that randomizing the training set may result in the loss of certain properties of the quadrature rule (which will be discussed later in this document). Despite this potential drawback, randomization has proven to be a useful technique for improving the learning process in many situations.
Observation:

- The randomization effectively resolves the issue observed in the first example. By exposing the network to a diverse mix of inputs throughout the training process, the network is better able to learn the desired function and adapt to the changing input patterns.

1.3 Code example: random-sampled inputs

In this example, rather than generating the training set through equidistant discretization of the input interval, we opt for an approach that directly uses randomly sampled inputs from the interval, along with their respective calculated outputs. This method allows for a more diverse and representative sample of the input space, further aiding the network in learning the desired function.
Observations:

- Upon comparing the second (randomized) and third (randomly sampled) examples, there are no visible differences in the learning outcomes. This suggests that both

methods effectively address the issue of overfitting and help the network learn the underlying patterns in the data.

- It is worth noting that in practice, it can be beneficial to ensure that outliers and/or border cases are included in the training set (rather than the test set) through special pre-processing of the data. By exposing the network to these extreme cases during training, it is better equipped to handle a wider range of inputs and provide more accurate predictions across the entire input space.

1.4 Code example: computing training data during the training

Given the considerable storage requirements of large training sets (e.g., over 20 MB for 10^5 inputs), we introduce an alternative strategy: sampling inputs and computing outputs on-the-fly during the training process. This is achieved through a new training data class tailored specifically for the cumulative density function (CDF).
By generating data dynamically during training, we eliminate the need to store extensive training data between training steps, effectively reducing the training process to just one epoch. This approach not only conserves memory but also allows for a more efficient and flexible learning process..
Observations:

- The dynamic generation of training data in each training step alleviates the need for multiple epochs, streamlining the training process and reducing the risk of overfitting.
- In this example, we already employed 10^4 inputs per training step. With 10^2 training steps, this results in a total of 10^6 training samples, showcasing the potential of this method for handling large amounts of data more efficiently.
- It is essential to consider the trade-offs of this approach, such as the potential loss of certain properties of the quadrature rule (as mentioned earlier). However, the benefits of improved efficiency and adaptability in the learning process may outweigh these drawbacks in many situations.

This example marks the initial foray into Monte Carlo learning, a novel approach designed to address the limitations of conventional methods.

2. Monte Carlo learning

In the previous section, we observed the similarities between randomizing the training set and randomly selecting inputs when the output generating function is known.
Upon comparing deterministic and randomized or random inputs, there doesn't seem to be a significant difference in the learning outcomes. As our project focuses on high-dimensional problems, where deterministic approaches could lead to complications due to the curse of dimensionality, we will here concentrate on cases involving randomly chosen inputs.
Building on this foundation, we will explore not only using randomly selected inputs but also incorporating approximations of the output. Specifically, we will employ Monte Carlo sampled approximations, paving the way for the Monte Carlo learning approach. The subsequent studies are motivated by the following objectives:

- To better understand the impact of noise (resulting from, for example, measurements) on the training process when dealing with a fixed and finite training set. Gaining insights in this direction could help improve the training process.
- To address scenarios where the evaluation of the output function is numerically expensive in practice, and we aim to replace the evaluation of the function with a neural network evaluation (e.g., to save real-time compile time). Expensive numerical approximations could lead to infeasibly long training processes. Using only approximations of the outputs might result in significant time savings.

First, in the next subsection, we will define mathematically what we mean by replacing a function with a neural network. Next, we will present several training examples that demonstrate this concept in action. Finally, in a third subsection, we will briefly explore the statistical properties that enable us to use randomly sampled outputs when training the neural network.

2.1 Neural network function approximation:

Consider a function of the form:

$$f : Y \to \mathbb{R},$$

for a specified training domain Y. We want to train a neural network with an architecture such that it is also a function of the form:

$$\mathcal{N} : Y \to \mathbb{R},$$

in such a way that it can be used to replace f. Specifying the second point of above remarks, consider the following gedankenexperiment:

- Consider that the evaluation of f is numerically expensive and therefore very time-consuming. In practice, if you have to evaluate the function on a regular basis (e.g. for new calibrated input parameters), there is usually not much more to do than improving the algorithm as much as possible, or paying for more/better hardware leading to ongoing costs.
- We wish to replace the evaluation of f with the evaluation of a neural network $\mathcal{N}$. While we will need many evaluations of the function f within the training process of the neural network (to generate training data), the training process has to be done only once and if this worked properly the costly numerical algorithm can be replaced by a fast neural network evaluation.

In this work, we are interested in looking at the impacts of different choices as training data. Neural network training consists of many facets, but from the Monte Carlo point of view, we are interested in the impact of the quality of the loss function approximation to the whole training process. While there are different choices for a proper loss function, we'll consider, the canonical choice, the mean-squared-error (MSE):

$$\text{MSE} := \frac{1}{n} \sum_{i=1}^{n} \left(f(Y_i) - \mathcal{N}(Y_i) \right)^2 . \tag{1}$$

Within each training step of a neural network training the loss function is evaluated (for different batches within each epoch) and the weights and/or biases are adjusted (with a respective learning rate) in such a way to minimize the loss function. For now, we'll assume that the following basic concepts of neural network training are known: batches, weights, bias, learning rate, back-propagation, epochs. Since Monte Carlo simulation is a technique for numerical integration, we may also interpret the issue as an approximation of the mean integrated squared error (MISE)

$$\text{MISE} := \int_Y (f(x) - \mathcal{N}(x))\,\mathrm{d}x. \tag{2}$$

Hence, assume the choice of training data is connected to the approximation of the MSE, respectively the MISE. Then, we may study to what extent the well known Monte Carlo results may apply to the training process of a neural network.

2.2 Example: Geometric-Brownian motion and option pricing

Consider the Black-Scholes model and a European call option. It is well known that the fair price of the option is given by the following formula:

$$C(S_0, K, T, r, \sigma) = S\Phi(d_1) - Ke^{-rT}\Phi(d_2), \tag{3}$$

where

$$d_1 = \frac{\ln(\frac{S_0}{K}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}},$$

$$d_2 = d_1 - \sigma\sqrt{T}.$$

For simplicity, in this section, we will restrict our studies to examples where the analytical closed solution is known.
Since we can solve the geometric Brownian motion without discretization error through the following equation:

$$S(T) = S_0 e^{(r - \frac{1}{2}\sigma^2)T + \sigma\sqrt{T}Z}, \tag{4}$$

where Z is a standard normal random variable, we obtain an unbiased Monte Carlo estimator of the fair value of the option given by:

$$C_{MC}(S_0, K, T, r, \sigma) = e^{-rT}\frac{1}{N}\sum_{i=1}^{N}\max(S_i(T) - K, 0). \tag{5}$$

Here, $S_i(T)$ represents the i-th simulated asset price at maturity T, and N is the number of Monte Carlo samples.
We aim to train a neural network to replace the parameter-to-price mapping, which is given by the Black-Scholes model.
We will present some more background information in appendix A.1.

2.2.1 Code example: B.S. closed solution plus noise

In this example, we investigate the performance of a neural network when trained on data with varying noise levels, specifically with noise variances of 0, 1, and 30. The purpose of this experiment is to demonstrate the ability of the neural network to learn and adapt to the underlying structure of the data, even in the presence of significant noise. This is a critical aspect of the Monte Carlo learning approach, as it utilizes approximations of the output function, which can be inherently noisy. The example leverages the Black-Scholes (B.S.) closed solution as a benchmark for assessing the performance of the neural network.
The experiment begins with the generation of training data, where the inputs are randomly selected, and the corresponding outputs are computed using the B.S. closed solution. Noise with varying levels of variance (0, 1, and 30) is then added to the outputs to simulate the impact of real-world measurement errors or approximation inaccuracies. This dataset, comprising both inputs and noisy outputs, is used to train the neural network.
Upon training, the neural network is evaluated on its ability to learn the underlying structure of the data and make predictions that align with the B.S. closed solution, despite the presence of noise. The results demonstrate that the neural network is indeed capable of finding a solution that closely approximates the B.S. closed solution, even when trained on data with significant noise.
This example highlights the robustness of neural networks in handling noisy data and showcases the potential benefits of employing the Monte Carlo learning approach in scenarios where the evaluation of the output function is computationally expensive or time-consuming. By using approximations of the outputs, we can save valuable resources and time while still achieving satisfactory results.

2.2.2 Code example: Geometric closed path solution

In this example, we build upon the previous experiment by altering the method of generating the training data. Instead of using the Black-Scholes (B.S.) closed solution, we now utilize sampled path solutions, which are unbiased since the solution of the geometric Brownian motion is known. To generate these paths, we apply the payoff function on the simulated paths, which results in a different noise distribution than the previous example. In this case, the noise is not symmetrically normally distributed around the expectation but follows a log-normal distribution with some degree of truncation due to the application of the payoff function.
Upon training the neural network with this new dataset, the performance is found to be not as good as when using the B.S. closed solution. However, when we increase the sample size per training step, as illustrated in the second picture, the network's performance improves, showcasing the ability of the neural network to adapt to the underlying data structure when provided with more information.
To further enhance the network's performance, we employ a variance reduction technique called importance sampling, specifically tailored for the European call option. As shown in the third picture, this method reduces the number of samples required per training step, resulting in a more efficient learning process.
This example highlights the importance of sample size and variance reduction techniques in training neural networks, particularly when dealing with noisy data or non-symmetrically

distributed noise. By optimizing these factors, we can improve the network's ability to learn and adapt to the underlying data structure, even when it deviates from the ideal noise distribution assumed in the previous example.

2.2.3 Code example: 5-dim training interval

In this next example, we extend the previous experiments by considering a higher-dimensional scenario, specifically a 5-dimensional case. The objective is to demonstrate the ability of neural networks to learn and adapt in the presence of multi-dimensional inputs, as well as to showcase the relevance of the Monte Carlo learning approach for handling more complex problems.

As before, we generate training data by simulating paths based on the geometric Brownian motion. In the 5-dimensional case, each input now represents a vector of five values. The corresponding outputs are computed by applying the payoff function on the simulated paths, resulting in log-normal distributed noise with some degree of truncation.

Training the neural network with this 5-dimensional dataset poses additional challenges, as the complexity of the problem has increased. Nevertheless, the network is still capable of learning the underlying structure of the data, albeit at a slower rate or with a higher requirement for computational resources.

This example underscores the versatility of neural networks and the Monte Carlo learning approach when dealing with higher-dimensional problems. By employing appropriate techniques and adjusting the sample size, we can tackle more complex scenarios while maintaining satisfactory performance, highlighting the potential of these methods for a wide range of applications.

2.3 Monte Carlo Learning: Finding the objective function using sample outputs

Now, we want to briefly discuss the underlying statistical property that allows for using sampled outputs during the training process.
Consider replacing the following simplified map: initial asset price S_0 to fair price C given by $C : Y = [100, 110] \to \mathbb{R}$, with $S_0 \in Y$ (e.g., for a European option given by eq. (3)), while the remaining parameters (K, T, r, σ) remain fixed and constant.
Now, wanting to replace C within the training interval Y, we search for the function f that minimizes the following expression:

$$\inf_{f \in \mathcal{C}(Y,\mathbb{R})} \int_Y (C(u) - f(u))^2 \, \mathrm{d}u \tag{6}$$

Let us now replace the asset price to fair price map by an expectation. Using the martingale approach, we know that the here studied pricing map can be re-formulated to

$$\begin{aligned} C(u) &= \int_{\mathbb{R}} \phi(z) \max\left(u e^{(r-\frac{1}{2}\sigma^2)T + \sigma\sqrt{T}z} - K, 0\right) \mathrm{d}z \\ &=: \int_{\mathbb{R}} P_u(z) \, \mathrm{d}z = \mathbb{E}[P_u] \end{aligned}$$

During each Monte Carlo learning training step, we randomly select N inputs u_i, with $i = 1, ..., N$. For each of these inputs we compute one sample output $P_{u_i}(z_i)$, using the solution of the geometric Brownian motion (see eq. (5)) and a standard normal random sample z_i, for $i = 1, \ldots, N$.
Now to bring things together: From, e.g., proposition 2.2 of Beck et al. (2019), we know that under certain assumptions, there exists a a neural network $N : Y \to \mathbb{R}$ and a unique continuous function f such that

$$\inf_{f \in \mathcal{C}(Y,\mathbb{R})} \int_Y (P_u - f(u))^2 \, \mathrm{d}u = \int_Y (P_u - N(u))^2 \, \mathrm{d}u \tag{7}$$

and it holds for every $u \in Y$ that

$$N(u) = \int\limits_{\mathbb{R}} P_u \, \mathrm{d}u = \mathbb{E}\left[P_u\right]. \tag{8}$$

Hence, Monte Carlo learning allows us to train a function that minimizes the expected loss and achieves the best possible approximation of the true relationship between input and output variables. By leveraging the statistical insights from the conditional expectation, Monte Carlo learning provides a robust and efficient framework for learning complex relationships in high-dimensional problems.

3. Code documentation: Version 1.0

The project consists of three main classes that work together to generate synthetic data and train a deep learning model on it. These three classes are DataImporter/TrainingDataGenerator, Neural_Approximator, and TrainingSettings. Each class has its own specific purpose, and they all work together to achieve the desired outcome of training a deep learning model on synthetic financial data.

3.1 Training data

In the Monte Carlo learning framework, there are two main methods for generating or selecting training data: the `DataImporter` class and the `TrainingDataGenerator` class. These classes serve different purposes and can be used in different situations.
The `DataImporter` class is used for importing data from external files, such as CSV files, and providing a way to access that data within the framework. Here is an example of using the `DataImporter` class:

```
Generator = DataImporter()
Generator.set_path(os.path.join(mainDirectory, 'src', 'Examples', ...
    'CumulativeDensitiyFunction', 'cdf_deterministic_data.csv'))
Generator.set_inputName('x')
Generator.set_outputName('CDF(x)')
```

The `TrainingDataGenerator` class, on the other hand, is designed for generating synthetic training and test data based on specific functions or distributions. The `CDF` class is an example of a subclass that inherits from `TrainingDataGenerator`. Here's an example of using the `CDF` class:

```
Generator = CDF()
Generator.set_inputName('x')
Generator.set_outputName('CDF(x)')
```

Both methods have their own advantages and can be chosen based on the specific requirements of a given project. The `DataImporter` is particularly useful when working with real-world data, while the `TrainingDataGenerator` is convenient when synthetic data is needed for testing or modeling purposes.

3.1.1 DataImporter

The DataImporter class is responsible for importing data from external files and providing a way to access that data within the Monte Carlo learning framework. This class is used by all training data generators in the project.

The DataImporter class has the following attributes:

```
_path = None
_trainTestRatio = 0.8
_randomized = True
```

```
4  _inputName = None
5  _outputName = None
6  _trainingSetSizes = [100, 1000, 10000]
7  _dataSeed = None
```

The `path` attribute specifies the location of the CSV file containing the data. The `trainTestRatio` attribute determines the ratio of the data to be used for training versus testing. The `randomized` attribute determines whether the data should be shuffled before splitting into training and testing sets. The `inputName` and `outputName` attributes specify the names of the columns in the CSV file corresponding to the input and output variables, respectively. The `trainingSetSizes` attribute is a list of integers specifying the number of data points to use for training at each iteration of the Monte Carlo algorithm. Finally, the `dataSeed` attribute is used to set the random seed for shuffling the data.
The `DataImporter` class also has the following methods for setting the attributes:

```
1      def set_path(self, path):
2          self._path = path
3
4      def set_trainTestRatio(self, ratio):
5          self._trainTestRatio = ratio
6
7      def set_randomized(self, randomized):
8          self._randomized = randomized
9
10     def set_inputName(self, inputName):
11         self._inputName = inputName
12
13     def set_outputName(self, outputName):
14         self._outputName = outputName
15
16     def set_trainingSetSizes(self, trainingSetSizes):
17         self._trainingSetSizes = trainingSetSizes
18
19     def set_dataSeed(self, seed):
20         self._dataSeed = seed
```

The following code shows an example of how to use the `DataImporter` class:

```
1  Generator = DataImporter()
2  Generator.set_path(os.path.join(mainDirectory, 'src', 'Examples', ...
       'CumulativeDensitiyFunction', 'cdf_deterministic_data.csv'))
3  Generator.set_inputName('x')
4  Generator.set_outputName('CDF(x)')
5  Generator.set_trainTestRatio(0.8)
6  Generator.set_randomized(False)
7  Generator.set_trainingSetSizes([100,1000,10000])
8  Generator.set_dataSeed(1)
```

In this example, a `DataImporter` object is created and the path to the CSV file is set. The input and output variable names are set to "x" and "CDF(x)", respectively. The ratio of training data to total data is set to 0.8, and the data is not randomized before splitting.

The training set sizes are set to [100, 1000, 10000], and the random seed for shuffling the data is set to 1.

3.1.2 TrainingDataGenerator

The `TrainingDataGenerator` class is an abstract class for generating training and test data for various applications. It defines the basic structure and methods that all training data generator classes should implement. In this section, we will discuss the basic structure of the `TrainingDataGenerator` class and show an example of a derived class called `CDF`, which generates training and test data for the cumulative distribution function (CDF).

The `TrainingDataGenerator` class has the following attributes:

```
_differential = None
_trainingMethod = TrainingMethod.GenerateDataDuringTraining
_inputName = 'x'
_outputName = 'y'
_dataSeed = 1
_testSeed = 1
```

The `differential` attribute is a flag indicating whether or not to use differential training. The `trainingMethod` attribute defines the method used for generating the training data. The `inputName` and `outputName` attributes specify the names of the input and output variables, respectively. The `dataSeed` and `testSeed` attributes are used to set the random seeds for generating the training and test data.

The `TrainingDataGenerator` class also has the following methods:

```
def set_nTest(self, value):
self._nTest = value

def set_inputName(self, inputName):
self._inputName = inputName

def set_outputName(self, outputName):
self._outputName = outputName

def set_dataSeed(self, dataSeed):
self._dataSeed = dataSeed

def set_testSeed(self, testSeed):
self._testSeed = testSeed

def trainingSet(self, m, trainSeed=None):
raise NotImplementedError("Subclass must implement abstract method")

def testSet(self, num, testSeed=None):
raise NotImplementedError("Subclass must implement abstract method")
```

The `trainingSet()` and `testSet()` methods are abstract methods that must be implemented by the subclasses of `TrainingDataGenerator`.

3.1.3 CDF

The `CDF` class is an example of a subclass derived from `TrainingDataGenerator`. It generates training and test data for the cumulative distribution function (CDF). The class has additional attributes specific to the CDF:

```
_mean = 0.0
_vol = 1.0
```

The `mean` and `vol` attributes represent the mean and volatility of the CDF, respectively.
The `CDF` class also overrides the `trainingSet()` and `testSet()` methods to generate the training and test data for the CDF:

```
def trainingSet(self, m, trainSeed=None):
np.random.seed(trainSeed)
b = 4.01
a = -4.01
randomInputs = (b - a) * np.random.random_sample(m) + a
cdfVector = norm.cdf(randomInputs, self._mean, self._vol)
return randomInputs.reshape([-1, 1]), cdfVector.reshape([-1, 1]), None

def testSet(self, num, testSeed=None):
np.random.seed(testSeed)
b = 4.0
a = -4.0
testInputs = np.linspace(a, b, num)
cdfVector = norm.cdf(testInputs, self._mean, self._vol)
return testInputs.reshape([-1, 1]), cdfVector.reshape([-1, 1]), None, None
```

The `trainingSet()` method generates a training set of size `m` by drawing random inputs uniformly between `a` and `b`. It then computes the CDF values for these inputs using the `norm.cdf()` function from the SciPy library. The method returns a tuple containing the reshaped random inputs and CDF values.
The `testSet()` method generates a test set of size `num` by creating a linearly spaced array of inputs between `a` and `b`. It then computes the CDF values for these inputs using the `norm.cdf()` function from the SciPy library. The method returns a tuple containing the reshaped test inputs and CDF values.
To work with the `CDF` class, create an instance and set its attributes using the appropriate methods. Afterward, configure the neural network structure and hyperparameters:

```
Generator = CDF()
Generator.set_inputName('x')
Generator.set_outputName('CDF(x)')
#2. Set train settings
TrainSettings.set_min_batch_size(1)
TrainSettings.set_test_frequency(10)
TrainSettings.set_nTest(2000)
TrainSettings.set_samplesPerStep(1000)
TrainSettings.set_trainingSteps(10)
```

In this example, a `CDF` object is created, and the input and output variable names are set to "x" and "CDF(x)", respectively. Next, the neural network structure and hyperparameters are configured using the `TrainSettings` class. The minimum batch size is set to 1, the test frequency is set to 10, the number of test data points is set to 2000, the samples per step are set to 1000, and the number of training steps is set to 10.

3.2 Neural_Approximator

The Neural_Approximator class is responsible for creating, preparing, building, and training a neural network for function approximation. This class is utilized in the Monte Carlo learning framework to train a neural network on the imported data.
The Neural_Approximator class has the following attributes:

```
_lam = None
_hiddenNeurons = None
_hiddenLayers = None
_activationFunctionsHidden = None
_activationFunctionOutput = None
_weight_seed = None
_biasNeuron = None
_Generator = None
x_raw = None
y_raw = None
dydx_raw = None
graph = None
session = None
x = None
y = None
x_mean = None
y_mean = None
x_std = None
y_std = None
dy_dx = None
lambda_j = None
```

The `lam` attribute determines the balance cost between values and derivatives. The `hiddenNeurons` and `hiddenLayers` attributes define the number of neurons in each hidden layer and the number of hidden layers, respectively. The `activationFunctionsHidden` attribute specifies the activation function to be used in the hidden layers, and the `activationFunctionOutput` attribute specifies the activation function for the output layer. The `weight_seed` attribute sets the random seed for initializing weights. The `biasNeuron` attribute determines whether to use a bias neuron.
The Neural_Approximator class has the following methods:

```
def set_Generator(self, Generator):
def set_lam(self, lam):
def set_hiddenNeurons(self, hiddenNeurons):
def set_hiddenLayers(self, hiddenLayers):
def set_activationFunctionsHidden(self, activationFunctionsHidden):
def set_activationFunctionOutput(self, activationFunctionOutput):
```

```
def set_weight_seed(self, weight_seed):
def set_biasNeuron(self, biasNeuron):
def initializeAndResetGraph(self):
def initializeData(self, x_raw=None, y_raw=None, dydx_raw=None):
def prepare(self, dataSize, nTest=None):
def storeNewDataAndNormalize(self, x_raw, y_raw, dydx_raw, dataSize):
def build_graph(self, lam, hiddenNeurons, hiddenLayers, ...
    activationFunctionsHidden, activationFunctionOutput, weight_seed, ...
    biasNeuron):
def train(self, description, TrainingSettings, reinit=True, ...
    callback=None, callback_epochs=[], xTest=None, yTest=None):
def predict_values(self, x, isTraining=True):
def predict_values_and_derivs(self, x):
```

Here's an example of how to use the Neural_Approximator class:

```
Regressor = Neural_Approximator()
Regressor.set_Generator(Generator)
Regressor.set_hiddenNeurons(50)
Regressor.set_hiddenLayers(2)
Regressor.set_activationFunctionsHidden(tf.nn.sigmoid)
Regressor.set_activationFunctionOutput(tf.nn.sigmoid)
```

In this example, a Neural_Approximator object is created and configured with the given Generator object. The number of neurons in each hidden layer is set to 50, and the number of hidden layers is set to 2. The activation functions for the hidden layers and the output layer are set to the sigmoid function.

3.3 TrainingSettings

The `TrainingSettings` class is responsible for managing various training settings required for different training methods in the machine learning framework. It provides methods to set and access the training configurations.

The `TrainingSettings` class has the following attributes:

```
_learning_rate_schedule = [(0.0, 0.01), (0.2, 0.001), (0.4, 0.0001), ...
    (0.6, 0.00001), (0.8, 0.000001)]
_epochs = 1
_batches_per_epoch = 1
_min_batch_size = 20
_madeSteps = 0
_testFrequency = 1000
_nTest = 1000
_mcRounds = 1
_samplesPerStep = 1000
_trainingSteps = 100
_useExponentialDecay = False
```

These attributes represent various configurations such as learning rate schedule, number of epochs, batches per epoch, minimum batch size, test frequency, number of test cases, Monte

Carlo rounds, samples per step, training steps, and whether to use exponential decay or not.
The `TrainingSettings` class also has the following methods for setting the attributes and accessing them:

```
def increaseMadeSteps(self)
def useExponentialDecay(self, initial_learning_rate, decay_rate, decay_steps)
def set_epochs(self, value)
def set_fileName(self, value)
def set_learning_rate_schedule(self, value)
def set_batches_per_epoch(self, value)
def set_min_batch_size(self, value)
def set_test_frequency(self, value)
def set_nTest(self, value)
def set_mcRounds(self, value)
def set_trainingSteps(self, trainingSetSizes)
def set_samplesPerStep(self, trainingSetSizes)
```

The following code shows an example of how to use the `TrainingSettings` class:

```
TrainSettings = TrainingSettings()
TrainSettings.set_epochs(20)
TrainSettings.set_batches_per_epoch(10)
TrainSettings.set_learning_rate_schedule([(0.0, 0.001), (0.333, 0.0001), ...
    (0.666, 0.0001)])
TrainSettings.set_test_frequency(100)
TrainSettings.set_nTest(2000)
TrainSettings.set_samplesPerStep(20000)
TrainSettings.set_trainingSteps(1000)
TrainSettings.useExponentialDecay(0.01, 0.5, 100)
```

In this example, a `TrainingSettings` object is created and various attributes are set such as epochs, batches per epoch, learning rate schedule, test frequency, number of test cases, samples per step, and training steps. Additionally, exponential decay is configured with the initial learning rate, decay rate, and decay steps.

4. Literature:

The framework described in this documentation was motivated by the recent work on Multilevel Monte Carlo learning by Gerstner et al. Gerstner et al. (2021). In this article, the authors propose a novel approach to solving partial/stochastic differential equations (PDEs/SDEs) using a combination of deep Learning and multilevel Monte Carlo methods. Specifically, they use a multiple neural network to learn the solution of the SDEs level estimators.
The underlying idea of using deep learning to solve PDEs was introduced by Beck et al. Beck et al. (2018). In their paper, the authors propose a method based on deep neural networks to solve both stochastic differential equations.
The code structure of classes used in this framework was adapted from the GitHub repository by Huge and Savine. The repository provides a collection of Jupyter notebooks illustrating various aspects of differential machine learning. The code was modified and extended to suit the specific needs of this project.

References

Christan Beck, Arnulf Jentzen, and Benno Kuckuck. Full error analysis for the training of deep neural networks. *arXiv preprint arXiv:1910.00121*, 2019.

Christian Beck, Sebastian Becker, Philipp Grohs, Nils Jaafari, and Arnulf Jentzen. Solving stochastic differential equations and kolmogorov equations by means of deep learning. *arXiv preprint arXiv:1806.00421*, 1, 2018.

Tobias Gerstner, Bastian Harrach, Daniel Roth, and Martin Simon. Multilevel monte carlo learning. *arXiv preprint arXiv:2102.08734*, 2021.

Appendix A. Appendix: Background option pricing and the curse of dimensionality

A.1 Financial instrument pricing

The valuation of a financial derivative is reliant on the anticipated future behaviour of the underlying asset. Typically, the underlying asset's behaviour is modelled using a stochastic differential equation. Consequently, the fair price of the derivative is often represented as either an expectation or a solution to a partial differential equation. The former requires the computation of an integration problem, while the latter often yields a large linear system that needs to be solved. The relationship between these two approaches is expounded in the Feynman-Kac theorem. In certain models, a closed-form solution can be obtained for the derivative price. However, if no closed-form solution exists, the derivative price must be computed by discretizing the problem and solving the resulting discrete problem. A concise overview of the typical methodology is presented in the organizational chart in Figure 1.

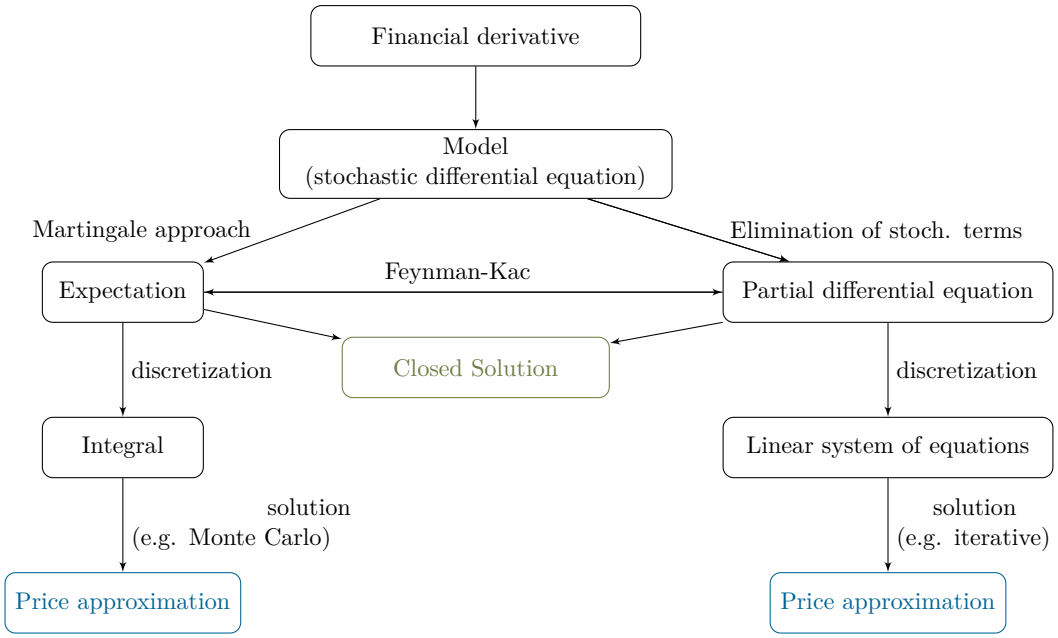

Figure 1: Organization of the common procedure for the valuation of financial derivatives.

The martingale approach is one of the main principles for option pricing and corresponds to the case where the fair price is given as an expectation. The martingale approach states that the fair price of an option is the discounted expectation of the payoff under the risk-neutral probability distribution of the underlying economic factors. However, the discretization of the expectation in many relevant cases is characterised by large or even infinite-dimensional integrals since the number of dimensions depends on the number of independent stochastic factors. E.g., consider that the number of stochastic factors for the above-introduced barrier options is equivalent to the number of observations dates. This is quite intuitive for discretely monitored barrier options: From observation date to

observation date, we have a certain probability whether the underlyings' stock crosses the barrier or not - each leading to an independent stochastic factor. However, in practice, there are barrier options observed daily within over a year, leading to a high-dimensional integration problem. For continuously monitored barrier options, these considerations lead to an infinite-dimensional integral.
For the trapezoidal rule, the effort of evaluating an integral for a required accuracy on a d-dimensional domain increases exponentially. This difficulty is the so-called *curse of dimension*. Unless the smoothness increases with the dimension, the order of convergence becomes slow even in moderate dimensions. However, there exists an algorithm that can break the curse of dimension. E.g., if the integrand satisfies specific smoothness conditions, the *sparse grid method* does not suffer from the curse of dimension. Monte Carlo methods are independent of the smoothness and dimension of the problem, but the convergence rate is quite low. Monte Carlo methods are randomised algorithms that evaluate the integrand at a set of (pseudo-)randomly chosen points. Monte Carlo methods remain the preferred approach for the simulation of more complex stochastic models and are used extensively in computational finance.

A.2 Curse of dimensionality

The importance of stochastic differential equations (SDEs) and partial differential equations (PDEs) in the field of quantitative finance and option pricing methods has been established. The interrelation between these two types of equations is crucial due to their divergent method properties. While deterministic algorithms demonstrate faster convergence, they are restricted by the curse of dimensionality, where computational expenses grow exponentially with increased dimensions. Conversely, stochastic algorithms do not fall prey to the curse of dimensionality, but they converge slowly. In practical scenarios, choosing the optimal algorithm for efficiency is challenging, and it depends on the specific problem at hand.
In this study, we explored the potential of deep learning algorithms as an alternative to expensive option pricing approximations. The loss function in a deep learning algorithm can be construed as a numerical integration problem, which allows us to evaluate the effectiveness of Monte Carlo learning.