Monte Carlo learning

Daniel Roth daniel-roth@posteo.com

Abstract

The purpose of this documentation is the study of the training process of neural network regressions. I.e., we are interested in replacing costly numerical algorithms with the evaluation of a neural network. This is in such a way different to the general machine learning considerations, that we will be able to generate an unlimited amount of data. This difference will also changes our training evaluation: We may test the neural regression on an interval of inputs and hence it is not restricted to a finite test set.
We will give an introduction to Monte Carlo learning, i.e., the idea of using sampled or approximated results as labels (for random selected inputs) during the training process of a neural network for regression purpose. The yield of this work is to give an understandable introduction to this new arising topic from the viewpoint of Monte Carlo simulation. We want to explain and show examples for the following three topics:

1. Random selection and random inputs as training data
2. Curse of dimensionality within the loss function approximation
3. Approximation of outputs as training data

For all mentioned examples the code are provided on GitHub.

1. Introduction

Regression formulation:

For simplicity, let's consider a function of the following form:

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

with the training domain Y. We want to train a neural network

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

in such a way that it can be used to replace f. Later, we will show explicit examples where this replacement might be of interest in practice. For now, we want to motivate the above considerations through 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 lost 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)

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

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.

While in this work, we will define Monte Carlo learning as the approach of using random sampled inputs and only approximations of the output, the author believes that studying this approach may help to understand neural network training in general.
For high-dimensional problems, trying to first generate training data and store them in files will sooner or later lead to infeasible file sizes. Under these circumstances, it is necessary to be able to compute training data on the fly during the neural network training process.

1.1 Introductory example: Random selection and random inputs as training data

For starters, we consider a one-dimensional training set and the cumulative density function Φ of a standard normal distribution as the function we want train. For simplicity, we cut the input domain and only study the training for inputs, hence we want to replace

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

with a neural network

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

For reference, $\int_4^4 \Phi(x)\,\mathrm{d}x = 4$ (and 4.01 for $[-4.01, 4.01]$).

In the following four examples, for which codes are given in the repo, we will study different approaches for the selection of training data and their impact on the goodness of the neural network. Furthermore, since we use equal network architectures and training hyperparameters in each example, we will get first impressions of the influence of the integration error of (2) on the neural network training.

1. For the first training, we equidistantly discretise the input interval with different step sizes and train the network with this training set.
   Implementation and results can be viewed in GitHub link
   Observations:
   - We see that it doesn't work properly. Since we don't do a randomization of the training set the batches contain nearly the same outputs for all of the inputs.
2. There are different possibilities to overcome the issue of the first example. E.g., one may use different deterministic quadrature rules for each batch, or a predefined quadrature rule with different disretizations.

   The easiest approach to overcome this issue is to randomize the training set. However, here certain properties of the quadrature rule may get lost.
   Implementation and results can be viewed in GitHub link
   Observations:
   - The randomization leads to feasible results
3. For the third training example, we directly use random sampled inputs over the interval.
   Implementation and results can be viewed in GitHub link
   Observations:
   - Comparing the second and third examples, there seem to be not much of a difference. There seems to be not much of a difference between deterministic chosen and randomized versus random inputs. Since we won't study approaches for

deterministically choosing the inputs and since we will be interested in higher dimensional problems, where the deterministic approach leads to further problems (curse of dimensionality), we will focus on the case of randomly chosen inputs.

4. Since we consider randomly chosen inputs and since for only 10^5 the file size exceeds 20 MB, we'd like to introduce the following approach: Sample inputs and compute outputs during the training. In the following we compute a batch directly during each training step:
   Implementation and results can be viewed in GitHub link
   Observations:

   – In this example we already used 10^4 inputs per training step, which by multiplying with the 10^2 trainings steps, results in 10^6 training samples.
   – Since not storing the training data locally, a second epoch would require the re-computation of the outputs, with according random seeds.
   – However, since epochs might amplify over-fitting, we will use new samples, or in other words one epoch, but with a possible infinite amount of training data.

1.2 Introductory example: Curse of dimensionality within the loss function approximation

In the last section, we briefly studied the effect of different training data selection/generation. In this section, we want to show the curse of dimensionality of numerical integration using some simple examples.
The curse of dimensionality refers to the challenges that arise when working with high-dimensional domains (e.g. training data).

There are different aspects occurring if we face high-dimensional data. First of all, the used statistical or machine learning technique might not be applicable for high-dimensional data. For example, it should be mentioned that some techniques may suffer from overfitting, which can lead to poor generalization and poor performance on new data.

Here, however, we want to investigate the curse of dimensionality within numerical integration and its effect on the neural network training by e.g. considering that we are able to neglect other errors than the numerical integration error in (2).

1.3 Introductory example: Approximation of outputs as training data

1.3.1 Finding the minimium while using payoff samples

From Multilevel Monte Carlo learning: Link
For, e.g., $d = H = 1$, this leads to $2M$ needed random numbers per iteration step.
Furthermore, from, e.g., proposition 2.2 of **?**, we know that under certain assumptions, there exists a a neural network $N : [0,1]^d \to \mathbb{R}$ and a unique continuous function f such that

$$\inf_{f \in \mathcal{C}([0,1]^d, \mathbb{R})} \int_{[0,1]^{d+H}} \Big(P^h(u) - f(u) \Big)^2 \, \mathrm{d}u = \int_{[0,1]^{d+H}} \Big(P^h(u) - N(u) \Big)^2 \, \mathrm{d}u \tag{3}$$

and it holds for every $u \in [0,1]^d$ that

$$N(u) = \int\limits_{[0,1]^H} P^h(u) \, \mathrm{d}u = \mathbb{E}\left[P^h(u) \right]. \tag{4}$$

Remark 1.1. In other words, the function minimizing the right-hand side of (**??**) could as well minimize the right-hand side of (**??**). I.e., for the training process of a neural network, both Monte Carlo estimators of **??** and **??** lead to unbiased results. With respect to the error assumptions, this justifies (**??**). However, one should keep in mind that the Monte Carlo estimator of **??** could be more efficient, even though, depending on the choice of N and M. The apparent reason for this is that even though the norm choice does not apply a bias, e.g., in (**??**), its respective Monte Carlo estimators' quality will affect the loss functions' error, e.g., in (**??**).

2. SDE vs PDE

2.1 Financial instrument pricing

The fair value of a financial derivative depends on the expected future development of the underlying. Usually, the underlyings' development is presumed to be given by a stochastic differential equation. Then, the fair price is commonly given as an expectation or a partial differential equations' solution. While in the case of an expectation, an integration problem has to be computed, partial differential equations usually result in a large linear system which needs to be solved. The connection between these approaches is shown in the Feynman-Kac theorem. As explained above, under specific models, a closed solution may exist for the price of a derivative. If no closed solution can be derived, the price of the derivative has to be computed after suitable discretization as the solution of the resulting discrete problem. The following brief organisational chart in Figure 1 gives a short overview of the common procedure.

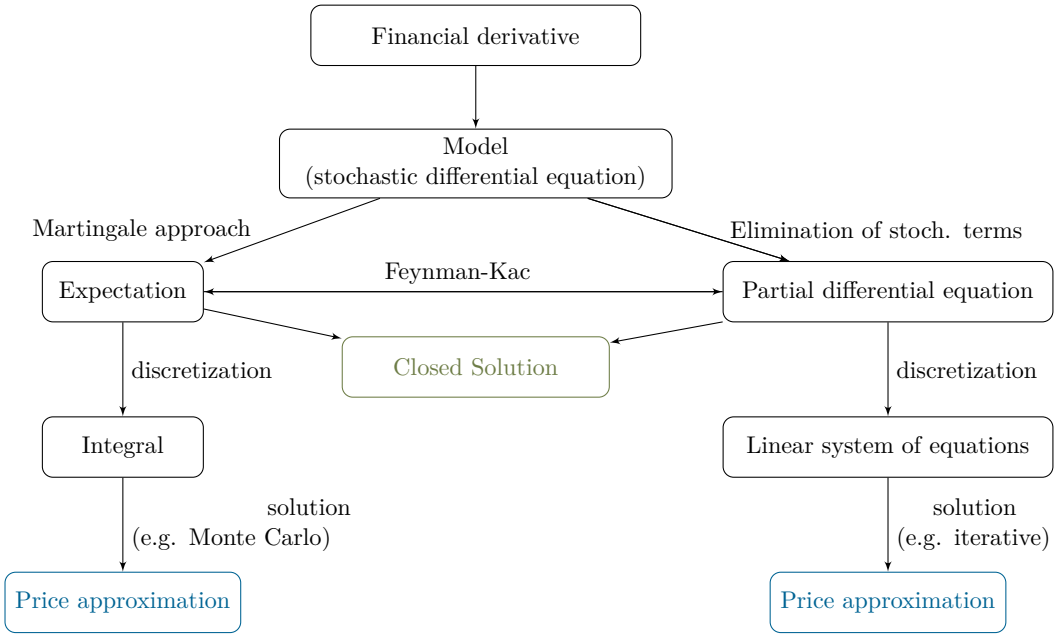

Figure 1: Organization of the common procedure for the valuation of financial derivatives.

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.

2.2 Further thoughts: PDE and SDE

Why is this connection so important? Because in the different worlds (I name them deterministic and stochastic), we have very different method properties. Some crucial properties are:
Deterministic algorithms in general converge faster
Deterministic algorithms suffer under the curse of dimensionality (you won't be able to integrate over 100 dimensions, because the cost will explode exponentially with respect to dimension). Hence, you won't be able to compute the price of an index option with a deterministic approach
Stochastic algorithms have converge slowly
Stochastic algorithms don't suffer under the curse of dimensionality.
Hence, in practice it is not directly clear which path we have to take to get a good approximation.

How is this stuff related with deep learning?
I think the most essential part to get a better understanding on deep learning is to understand the connection between the loss function and the hyper-parameters:
Let's take the mean-squared-error (MSE): The MSE is a discrete version of the mean-integrated-error
Hence, a good approximation of the loss function might be interpreted as numerical integration. Numerical integration can be done with a determinstic approach (quadrature rule, e.g. equidistant grid) or a stochastic approach (Monte Carlo simulation, e.g. by sampling random inputs).