Monte Carlo learning

Daniel Roth daniel-roth@posteo.com

Abstract

In this document, we will study the training process of a neural network regressions. I.e., we are interested in replacing costly numerical algorithms with the evaluation of a neural network. This is different to the general machine learning considerations, since 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 not only on 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 during the training process of a neural network for regression purpose.

Therefore, we will start with examples on which we will build more complex examples. For all mentioned examples the code will be provided in a GitHub repository. The yield of this work is to give an understandable introduction to this new arising topic from the viewpoint of Monte Carlo simulation.

Within this work, we will consider examples where there are algorithms or analytical formulas given to obtain a solution up to an accuracy of ϵ, for all considered inputs. The working question will be how to replace these algorithms or analytical formulas through a neural network and how to train such a network efficiently.

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):

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

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)

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

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. Exciting questions we want to tackle in this work are:

- effect of variance reduction
- using samples instead of accurate results
- curse of dimensionality

2. Examples

2.1 CDF

For starters, we consider a one-dimensional input set for the cumulative density function Φ of a standard normal distribution. 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 $[-4.01, 4.01]$)

2.2 Training examples

1. For the first training, we equidistantly discretise the input interval with different step sizes and train the network with this training set. Comments:

   – 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. Comments:

   – The randomization leads to feasible results

3. For the third training example, we directly use random sampled inputs over the interval. Commments:

   – 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: Comments

   – 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.

3. Introduction: PDE and SDE

If you don't want to go deep into the formulas (or don't want to), I would still recommend you to understand the relationship between PDE and SDE. E.g. I made this image couple of time ago for a lecture:

Kommentar: add picture from diss

For e.g. a simple European option we have two main tools to get a price approximation (let's assume we don't know the closed solution):

In the stochastic world: Monte Carlo simulation

In the deterministic world: Finite differences

Coming from this side: Since both methods work and deliver the same result, there has to be a connection between SDE's and PDE's, which is stated in "Linking SDEs to PDEs: Kolmogorov Forward Equation Derivation" in the article above... (of course we were now running backwards from the solution to the equations)

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 dimensionalty

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).

4. Finding the minimium while using payoff samples

From Multilevel Monte Carlo learning: `https://arxiv.org/abs/2102.08734v1`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

$$\inf_{f \in \mathcal{C}([0,1]^d, \mathbb{R})} \int_{[0,1]^{d+H}} \Big(P^h(u) - f(u) \Big)^2 \, \mathrm{d}u = \int_{[0,1]^{d+H}} \Big(P^h(u) - N(u) \Big)^2 \, \mathrm{d}u \qquad (4.1)$$

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

$$N(u) = \int_{[0,1]^H} P^h(u)\,\mathrm{d}u = \mathbb{E}\left[P^h(u)\right]. \tag{4.2}$$

Remark 4.1 *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* (**??**)*.*