Differential Machine Learning – Appendix 1
Learning Prices from Samples

Brian Huge
brian.huge@danskebank.dk

Antoine Savine
antoine.savine@danskebank.dk

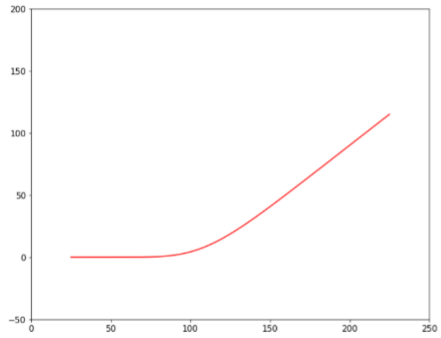
May 4, 2020

Introduction

When learning Derivatives pricing and risk approximations, the main computation load belongs to the simulation of the training set. For complex transactions and trading books, it is not viable to learn from examples of ground truth prices. True prices are computed numerically, generally by Monte-Carlo. Even a small dataset of say, 1000 examples, is therefore simulated for the computation cost of 1000 Monte-Carlo pricings, a highly unrealistic cost in a practical context. Alternatively, *sample* datasets *a la* Longstaff-Schwartz (2001) are produced for the computation cost of *one* Monte-Carlo pricing, where each example is not a ground truth price, but one sample of the payoff, simulated for the cost of one Monte-Carlo path. This methodology, also called LSM (for Least Square Method as it is called in the founding paper) simulates training sets in realistic time and allows to learn pricing approximations in realistic time.

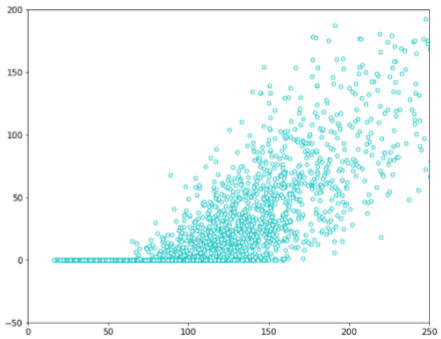
This being said, we now expect the machine learning model to learn correct pricing functions without having ever seen a price. Consider a simple example: to learn the pricing function for a European call in Black and Scholes, we simulate a training set of call payoffs $Y^{(i)} = \left(S_{T_2}^{(i)} - K\right)^+$ given initial states $X^{(i)} = S_{T_1}^{(i)}$. The result is a random looking cloud of points $X^{(i)}, Y^{(i)}$, and we expect the machine to learn from this data the correct pricing function given by Black and Scholes' formula.

learn this



$$E\left[\left(S_{T_2} - K\right)^+ \middle| S_{T_1}\right] = S_{T_1} N(d_1) - K N(d_2)$$

from this



$$\left\{S_{T_1}^{(i)}, \left(S_{T_2}^{(i)} - K\right)^+\right\}$$

It is not given at all, and it may even seem somewhat magical, that training a machine learning model on this data

should converge to the correct function. When we train on ground true prices, we essentially interpolate prices in input space, where it is clear and intuitive that arbitrary functions are approximated to arbitrary accuracy by growing the size of the training set and the capacity of the model. In fact, the same holds with LSM datasets, and this appendix discusses some important intuitions and presents sketches of mathematical proof of why this is the case.

In the first section, we recall LSM in detail[1] and frame it in machine learning terms. Readers familiar with the Longstaff-Schwartz algorithm may browse through this section quickly, although skipping it altogether is not recommended, this is where we set important notations. In the second section, we discuss universal approximators, formalize their training process on LSM samples, and demonstrate convergence to true prices. In the third section, we define pathwise differentials, formalize differential training and show that it too converges to true risk sensitivities.

The purpose of this document is to explain and formalize important mathematical intuitions, not to provide complete formal proofs. We often skip important mathematical technicalities so our demonstrations should really be qualified as 'sketches of proof'.

1 LSM datasets

1.1 Markov States

1.1.1 Model state

First, we formalize the definition of a LSM dataset. LSM datasets are simulated with a Monte-Carlo implementation of a dynamic pricing model. Dynamic models are parametric assumptions of the diffusion of a *state vector* S_t, of the form:

$$dS_t = \mu\left(S_t, t\right) dt + \sigma\left(S_t, t\right) dW_t$$

where S_t is a vector of dimension n_0, $\mu\left(s,t\right)$ is a vector valued function of dimension n_0, $\sigma\left(s,t\right)$ is a matrix valued function of dimension $n_0 \times p$ and W_t is a p dimensional standard Brownian motion under the *pricing measure*. The number n_0 is called the *Markov dimension* of the model, the number p is called the *number of factors*. Some models are non-diffusive, for example, jump diffusion models *a la* Merton or rough volatility models *a la* Gatheral. All the arguments of this note carry over to more general models, but in the interest of concision and simplicity, we only consider diffusions in the exposition. Dynamic models are implemented in Monte-Carlo simulations, e.g. with Euler's scheme:

$$S^{(i)}_{T_{j+1}} = S^{(i)}_{T_j} + \mu\left(S^{(i)}_{T_j}, T_j\right)\left(T_{j+1} - T_j\right) + \sigma\left(S^{(i)}_{T_j}, T_j\right)\sqrt{T_{j+1} - T_j} N^{(i)}_j$$

where i is the index of the path, j is the index of the time step and the $N_j\left(i\right)$ are independent Gaussian vectors in dimension p.

The definition of the state vector S_t depends on the model. In Black and Scholes or local volatility extensions *a la* Dupire, the state is the underlying asset price. With stochastic volatility models like SABR or Heston, the bi-dimensional state $S_t = (s_t, \sigma_t)$ is the pair (current asset price, current volatility). In Hull and White / Cheyette interest rate models, the state is a low dimensional latent representation of the yield curve. In general Heath-Jarrow-Morton / Libor Market models, the state is the collection of all forward rates in the yield curve.

We call *model state* on date t the state vector S_t of the model on this date.

[1] Omitting the recursive part of the algorithm, specific to early exerciseable Derivatives.

1.1.2 Transaction state

Derivatives transactions also carry a state, in the sense that the transactions evolve and mutate during their lifetime. The state of a barrier option depends on whether the barrier was hit in the past. The state of a Bermudan swaption depends on whether it was exercised. Even the state of a swap depends on the coupons fixed in the past and not yet paid. European options don't carry state until expiry, but then, they may exercise into an underlying schedule of cashflows.

We denote U_t the transaction state at time t and n_1 its dimension. For a barrier option, the transaction state is of dimension one and contains the indicator of having hit the barrier prior to t. For a real-world trading book, the dimension n_1 may be in the thousands and it may be necessary to split the book to avoid dimension overload. The transaction state is simulated together with the model state in a Monte-Carlo implementation. In a system where event driven cashflows are scripted, the transaction state U_t is also the script state, i.e. the collection of variables in the script evaluated over the Monte-Carlo path up to time t.

1.1.3 Training inputs

The exercise is to learn the pricing function for a given transaction or a set of transactions, in a given model, on a given date $T_1 \geq 0$, sometimes called the *exposure date* or *horizon date*. The price evidently depends on both the state of the model S_{T_1} and the state of the transaction U_{T_1}. The concatenation of these two vectors $X_{T_1} = [S_{T_1}, U_{T_1}]$ constitute the *complete Markov state* of the system, in the sense that the true price of transactions at T_1 are deterministic (but unknown) functions of X_{T_1}.

The dimension of the state vector is $n_0 + n_1 = n$.

The training inputs are a collection of examples $X^{(i)}$ of the Markov state X_{T_1} in dimension n. They may be sampled by Monte-Carlo simulation between today ($T_0 = 0$) and T_1, or otherwise. The distribution of X in the training set should reflect the intended use of the trained approximation. For example, in the context of value at risk (VAR) or expected loss (FRTB), we need an accurate approximation in extreme scenarios, hence, we need them well represented in the training set, e.g. with a Monte-Carlo simulation with increased volatility. In low dimension, the training states $X^{(i)}$ may be put on a regular grid over a relevant domain. In higher dimension, they may be sampled over a relevant domain with a low discrepancy sequence like Sobol. When the exposure date T_1 is today or close, sampling X_{T_1} with Monte-Carlo is nonsensical, an appropriate sampling distribution must be applied depending on context.

1.2 Pricing

1.2.1 Cashflows and transactions

A cashflow CF_k paid at time T_k is formally defined as a T_k measurable random variable. This means that the cashflow is revealed on or before its payment date. In the world described by the model, this is a *functional* of the path of the state vector S_t from $T_0 = 0$ to the payment date T_k and may be simulated by Monte-Carlo.

A transaction is a collection of cashflows $CF_1, ..., CF_K$. A European call of strike K expiring at T is a unique cashflow, paid at T, defined as $(s_T - K)^+$. A barrier option also defines a unique cashflow:

$$1_{\{max(s_u, 0 \leq u \leq T) < B\}} (s_T - K)^+$$

An interest rate swap defines a schedule of cashflows paid on its fixed leg and another one paid on its floating leg. Scripting conveniently and consistently describes all cashflows, as functional of market variables, in a language purposely designed for this purpose.

A netting set or trading book is a collection of transactions, hence, ultimately, a collection of cashflows. In what follows, the word 'transaction' refers to arbitrary collection of cashflows, maybe netting sets or trading books. The payment date of the last cashflow is called the *maturity* of the transaction and denoted T_2.

1.2.2 Payoffs

The payoff of a transaction is defined as the *discounted sum of all its cashflows*:

$$\pi = \sum_{k=1}^{K} CF_k$$

hence, the payoff is a T_2 measurable random variable, which can be sampled by Monte-Carlo simulation.

For the purpose of learning the pricing function of a transaction on an exposure date T_1, we only consider cashflows *after* T_1, and discount them to the exposure date. In the interest of simplicity, we incorporate discounting to T_1 in the functional definition of the cashflows. Hence:

$$\pi = \sum_{T_k > T_1} CF_k$$

The payoff is still a T_2 measurable random variable. It can be sampled by Monte-Carlo simulation *conditional to state* $X_{T_1} = [S_{T_1}, U_{T_1}]$ *at* T_1 by seeding the simulation with state X_{T_1} at T_1 and simulating up to T_2.

1.2.3 Pricing

Assuming a complete, arbitrage-free model, we immediately get the price of the transaction from the fundamental theorem of asset pricing:

$$V_{T_1} = E\left[\pi | F_{T_1}\right]$$

where expectations are taken in the pricing measure defined by the model and F_{T_1} is the filtration at T_1 (loosely speaking, the information available at T_1). Since by assumption $X_{T_1} = [S_{T_1}, U_{T_1}]$ is the complete Markov state of the system at T_1:

$$V_{T_1} = E\left[\pi | X_{T_1}\right] = h\left(X_{T_1}\right)$$

Hence, the true price is a deterministic (but unknown) function h of the Markov state.

1.2.4 Training labels

We see that the price corresponding to the input example $X^{(i)}$ is:

$$V^{(i)} = E\left[\pi | X_{T_1} = X^{(i)}\right]$$

and that its computation, in the general case, involves averaging payoffs over a number of Monte-Carlo simulations from T_1 to T_2, all identically seeded with $X_{T_1} = X^{(i)}$. This is also called *nested simulations* because a set of simulations is necessary to compute the value of each example, the initial states having themselves been sampled somehow. If the initial states were sampled with Monte-Carlo simulations, they are called *outer simulations*. Hence, we have *simulations within simulations*, an extremely costly and inefficient procedure[2].

Instead, for each example i, we draw one single payoff $\pi^{(i)}$ from its distribution conditional to $X_{T_1} = X^{(i)}$, by simulation of one Monte-Carlo path from T_1 to T_2, seeded with $X^{(i)}$ at T_1. The labels in our dataset correspond to these random draws:

[2] Although, we can use nested simulations as a reference to measure performance, as we did in the working paper, sections 3.2 and 3.3. In production, nested simulations may be used to regularly double check numbers.

$$Y^{(i)} \xleftarrow{sample} \pi_{T_2} | \left\{ X_{T_1} = X^{(i)} \right\}$$

Notice (dropping the condition to $X_{T_1} = X^{(i)}$ to simplify notations) that, while labels no longer correspond to true prices, they are *unbiased* (if noisy) estimates of true prices.

$$E\left[Y^{(i)}\right] = E\left[\pi^{(i)}\right] = V^{(i)}$$

in other terms:

$$Y^{(i)} = V^{(i)} + \epsilon^{(i)} = h\left(X^{(i)}\right) + \epsilon^{(i)}$$

where the $\epsilon^{(i)}$ are independent noise with $E\left[\epsilon(i)\right] = 0$. This is why universal approximators trained on LSM datasets converge to true prices despite having never seen one.

2 Machine learning with LSM datasets

2.1 Universal approximators

Having simulated a training set of examples $X^{(i)}, Y^{(i)}$ we proceed to train approximators, defined as functions $\hat{h}\left(x, w\right)$ of the input vector x of dimension n, parameterized by a vector w of *learnable weights* of dimension d. This is a general definition of approximators. In classic regression, w are the regression weights, often denoted β. In a neural network, w is the collection of all connection matrices $W^{[l]}$ and bias vectors $b^{[l]}$ in the multiple layers $l = 1, ..., L$ of the network.

The *capacity* of the approximator is an informal measure of both its computational complexity and its ability to approximate functions by matching discrete sets of datapoints. A classic formal definition of capacity is the Vapnik-Chervonenkis dimension, defined as the largest number of arbitrary datapoints the approximator can match exactly. We settle for a weaker definition of capacity as the number d of learnable parameters, sufficient for our purpose.

A *universal* approximator is one guaranteed to approximate any function to arbitrary accuracy when its capacity is grown to infinity. Examples of universal approximator include classic linear regression, as long as the regression functions form a complete basis of the function space. Polynomial, harmonic (Fourier) and radial basis regressors are all universal approximators. Famously, neural networks are universal approximators too, a result known as the Universal Approximation Theorem.

2.2 LSM approximation theorem

Training an approximator means setting the value of its learnable parameters w in order to minimize a *cost function*, generally the mean square error (MSE) between the approximations and labels over a training set of m examples:

$$w = argmin_w MSE = \frac{1}{m} \sum_{i=1}^{m} \left[\hat{h}\left(X^{(i)}, w\right) - Y^{(i)}\right]^2$$

The following theorem justifies the practice of training approximators on LSM datasets:

> A universal approximator $\hat{f}$ trained by minimization of the MSE over a training set $X^{(i)}, Y^{(i)}$ of independent examples of Markov states at T_1 coupled with conditional sample payoffs at T_2, converges to the true pricing function

$$h(x) = E[\pi | X_{T_1} = x]$$

when the size m of the training set and the capacity d of the approximator both grow to infinity.

We provide a sketch of proof, skipping important mathematical technicalities to highlight intuitions and important properties.

First, notice that the training set consists in m independent, identically distributed realizations of the couple X, Y where X is the Markov state at T_1, sampled from a distribution reflecting the intended application of the approximator, and $Y|X$ is the conditional payoff at T_2, sampled from the pricing distribution defined by the model and sampled by conditional Monte-Carlo simulation.

Hence, the true pricing function h satisfies:

$$h(X) = E[Y|X]$$

By definition, the conditional expectation $E[Y|X]$ is the function of X closest to Y in L^2:

$$E[Y|X] \equiv min_{g \in L^2(X)} ||g(X) - Y||_2^2$$

Hence, pricing can be framed as an optimization problem in the space of functions. By universal approximation property:

$$min_w ||\hat{h}(X, w) - Y||_2^2 \to min_{g \in L^2(X)} ||g(X) - Y||_2^2$$

when the capacity d grows to infinity, and:

$$MSE \to ||\hat{h}(X, w) - Y||_2^2$$

when m grows to infinity, by assumption of an IID training set, sampled from the correct distributions. Hence:

$$\hat{h}(X, min_w MSE) \to h(X)$$

when both m and d grow to infinity. This is the theoretical basis for training machine learning models on LSM samples, and it applies to all universal approximators, including neural networks. This is why regression or neural networks trained on samples 'magically' converge to the correct pricing function, as observed e.g. in our demonstration notebook with European calls in Black and Scholes and basket options in Bachelier. The theorem is general and equally guarantees convergence for arbitrary (complete and arbitrage-free) models and schedules of cashflows.

3 Differential Machine Learning with LSM datasets

3.1 Pathwise differentials

By definition, pathwise differentials $\partial\pi/\partial X_{T_1}$ are T_2 measurable random variables equal to the gradient of the payoff at T_2 wrt the state variables at T_1.

For example, for a European call in Black and Scholes, pathwise derivatives are equal to:

$$\frac{\partial \pi}{\partial S_{T_1}} = \frac{\partial (s_{T_2} - K)^+}{\partial s_{T_1}} = \frac{\partial (s_{T_2} - K)^+}{\partial s_{T_2}} \frac{\partial s_{T_2}}{\partial s_{T_1}} = 1_{\{s_{T_2} > K\}} \frac{s_{T_2}}{s_{T_1}}$$

In a general context, pathwise differentials are conveniently and efficiently computed with automatic adjoint differentiation (AAD) over Monte-Carlo paths as explained in the founding paper *Smoking Adjoints* (Giles and Glasserman, Risk 2006) and the vast amount of literature that followed. We posted a video tutorial explaining the main ideas in 15 minutes `https://www.youtube.com/watch?v=IcQkwgPwfm4`.

Pathwise differentials are not well defined for discontinuous cashflows, like digitals or barriers. This is classically resolved by *smoothing*, i.e. the replacement of discontinuous cashflows with close continuous approximations. Digitals are typically represented as tight call spreads, and barriers are represented as *soft barriers*. Smoothing has been a standard practice on Derivatives trading desks for several decades. For an overview of smoothing, including generalization in terms of *fuzzy logic* and a systematic smoothing algorithm, see our presentation `https://www.slideshare.net/AntoineSavine/stabilise-risks-of-discontinuous-payoffs-with-fuzzy-logic`.

Provided that all cashflows are differentiable by smoothing (and some additional, generally satisfied technical requirements), the expectation and differentiation operators commute so that true risks are (conditional) expectations of pathwise differentials:

$$\text{if } h\left(x\right) = E\left[\pi | X_{T_1} = x\right] \text{ then } \frac{\partial h\left(x\right)}{\partial x} = E\left[\frac{\partial \pi}{\partial X_{T_1}} | X_{T_1} = x\right]$$

This theorem is demonstrated in stochastic literature, the most general demonstration being found in *Functional Ito Calculus*, also called *Dupire Calculus*, see Quantitative Finance Volume 19, 2019, Issue 5. It also applies to pathwise differentials *wrt model parameters*, and justifies the practice of Monte-Carlo risk reports by averaging pathwise derivatives.

3.2 Training on LSM pathwise differentials

LSM datasets consist of inputs $X^{(i)} = X_{T_1}^{(i)}$ with labels $Y^{(i)} = \pi^{(i)}$. Pathwise differentials are therefore the gradients of labels $Y^{(i)}$ wrt inputs $X^{(i)}$. The main proposition of the working paper is to augment training datasets with those differentials and implement an adequate training on the augmented dataset, with the result of vastly improved approximation performance.

Suppose first that we are training an approximator on differentials alone:

$$w = argmin_w MSE = \frac{1}{m}\sum_{i=1}^{m} || \frac{\partial \hat{h}\left(X^{(i)}, w\right)}{\partial X^{(i)}} - \frac{\partial Y^{(i)}}{\partial X^{(i)}} ||^2$$

with predicted derivatives on the left hand side (LHS) and differential labels on the right hand side (RHS). Note that the LHS *is* the predicted sensitivity $\partial \hat{h}\left(X^{(i)}\right)/\partial X^{(i)}$ but the RHS is *not* the true sensitivity $\partial h\left(X^{(i)}\right)/\partial X^{(i)}$. It is the pathwise differential, a random variable with expectation the true sensitivity and additional sampling noise.

We have already seen this exact same situation while training approximators on LSM samples, and demonstrated that the trained approximator converges to the true conditional expectation, in this case, the expectation of pathwise differentials, a.k.a. the true risk sensitivities.

The trained approximator will therefore converge to a function $\hat{h}$ with all the same differentials as the true pricing function h. It follows that on convergence $\hat{h} = h$ modulo an additive constant c, trivially computed at the term of training by matching means:

$$c = \frac{1}{m}\sum_{i=1}^{m}\left[Y^{(i)} - \hat{h}\left(X^{(i)}\right)\right]$$

Conclusion

We reviewed the details of LSM simulation framed in machine learning terms, and demonstrated that training approximators on LSM datasets effectively converges to the true pricing functions. We then proceeded to demonstrate that the same is true of differential training, i.e. training approximators on pathwise differentials also converges to the true pricing functions.

These are asymptotic results. They justify standard practices and guarantee consistence and meaningfulness of classical and differential training on LSM datasets, classical or augmented e.g. with AAD. They don't say anything about speed of convergence. In practicular, they don't provide a quantification of errors with finite capacity d and finite datasets of size m. They don't explain the vastly improved performance of differential training, consistently observed across examples of practical relevance in the working paper. Both methods have the same asymptotic guarantees, where they differ is in the magnitude of errors with finite capacity and size. To quantify those is a harder problem, explored in a future appendix.