Differential Machine Learning – Appendix 4
Supervised Learning without Supervision:
Wide and Deep Architecture and Asymptotic Control

Brian Huge
brian.huge@danskebank.dk

Antoine Savine
antoine.savine@danskebank.dk

May 2, 2020

Introduction

Modern deep learning is very effective at function approximation, especially in the differential form presented in the working paper. But training of neural networks is a nonconvex problem, without guaranteed convergence to the global minimum of the cost function or close[1]. Neural networks are usually trained under close human supervision and it is hard to execute it reliably behind the scenes.

This is of particular concern in Derivatives risk management, where automated procedures cannot be implemented in production without strong guarantees. Vast empirical evidence, that modern training heuristics (data normalization, Xavier-Glorot initialization, ADAM optimization, one-cycle learning rate schedule...) often combine to converge to acceptable minima, is not enough. Risk management is not built on faith but on mathematical guarantees.

In this appendix, we see how and to what extent guarantees can be established for training neural networks with a special architecture called 'wide and deep', also promoted by Google in the context of recommender systems.

We also discuss asymptotic control, another key requirement for reliable unsupervised implementation.

1 Wide and Deep Learning

1.1 Wide vs Deep

Classic regression (which we call *wide learning* for reasons apparent soon) finds an approximation $\hat{f}$ of a target function $f : \mathbb{R}^n \to \mathbb{R}$ as a linear combination of a predefined set of p basis functions ϕ_j of inputs x in dimension n:

$$\hat{f}(x; w) = \sum_{j=1}^{p} w_j \phi_j(x) = \phi(x)\, w$$

by projection onto the space of functions spanned by the basis functions ϕ_j. With a training set of m examples given by the matrix X of shape $m \times n$, with labels stacked in a vector Y of dimension m, the p learnable weights w_j are estimated by minimization of the mean squared error (MSE), itself an unbiased estimation of the distance $||\hat{f} - f||^2$ in L^2:

[1] It is also prone to overfitting, so generalization is not guaranteed even with minimum MSE on the training set. Differential machine learning considerably helps, as abundantly coommented in the working paper and other appendices.

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

It is immediately visible that the objective MSE is convex in the weights w. The optimization is well defined with a unique minimum, easily found by canceling the gradient of the MSE wrt w, resulting in the well known *normal equation*:

$$\hat{w} = \left(\Phi^T \Phi \right)^{-1} \Phi^T Y$$

where Φ is the $m \times p$ matrix stacking basis functions of inputs in its row vectors:

$$\Phi^{(i)} = \phi \left(X^{(i)} \right)$$

Let us call *input dimension* the dimension n of x and *regression dimension* the dimension p of ϕ. In low (regression) dimension, the normal equation is tractable but subject to numerical trouble when the matrix $\Phi^T \Phi$ is near singular. This is resolved by SVD regression (see appendix 3), a safe implementation of the projection operator so the problem is still convex and analytically solvable. In high dimension, the inversion or SVD decomposition may become intractable, in which case the argmin of the MSE is found numerically, e.g. by a variant of gradient descent. Importantly, the problem remains convex so numerical optimizations like gradient descent are guaranteed to converge to the unique minimum (modulo appropriate learning rate schedule).

This is all good, but it should be clear that the practical performance of classic regression is highly dependent on the relevance of the basis functions ϕ_j for the approximation of the true function f, mathematically measured by the L^2 distance between the true function and the space spanned by the basis functions, of which the MSE is an estimate.

One strategy is pick a vast number of basis functions ϕ_j so that their combinations approximate *all* functions to acceptable accuracy. For example, the set of all *monomials* of x of the form:

$$\phi_j (x) = \prod_{j=1}^{n} x_j^{k_j} \text{ such that } \sum_{j=1}^{n} k_j \leq K$$

are dense in $L^2 (\mathbb{R}^n)$ so polynomial regression has the *universal approximation* property: it approximates all functions to arbitrary accuracy by growing degree K. Regardless, this strategy is almost never viable in practice due to the *course of dimensionality*. Readers may convince themselves that the number of monomials of degree up to K in dimension n is:

$$\frac{(n+K)!}{n!K!}$$

and grows exponentially in the input dimension n and polynomial degree K. A cubic regression in dimension 20 has $1,771$ monomials. A degree 7 polynomial regression has $888,030$. In most contexts of practical relevance, dimension of this magnitude is both computationally intractable and bound to overfit training noise, even when dimension n was previously reduced with a meaningful method like differential PCA (see appendix 2). The same arguments apply to all other basis of functions besides polynomials: Fourier harmonics, radial kernels, cubic splines etc. They are all affected by the same curse and only viable in low dimension.

Regression is only viable in practice when basis functions are carefully selected with handcrafted rules applied to contextual information. One example is the classic Longstaff-Schwartz algorithm (LSM) of 2001, originally designed for the regression of the continuation value of Bermudan options in the Libor Market Model (LMM) of Musiela and al. (1995). The Markov state of LMM is high dimensional and includes all forward Libor rates up to a final maturity, e.g. with 3m Libors up to maturity 30y, dimension is $n = 120$. To regress the value of a Bermudan swaption in such high dimension is hopeless. Instead, classic implementations regress on

low dimensional features (i.e. a small number of nonlinear functions) of the state, called *regression variables*. It is known that Bermudan options on call dates are mainly sensitive to the swap rates to maturity and to the next call (assuming deterministic volatility and basis). Instead of attempting regression in dimension 120, effective implementations of LSM simply regress on those two functions of the state, effectively reducing regression dimension from 120 to 2.

This is very effective, but it takes prior knowledge of the generative model. We can safely apply this methodology because we know that this is a Bermudan option in a model with deterministic volatility and basis. Careful prior study determined that the value mainly depends on two regression variables, the two swap rates, so we could hardcode the transformation of the 120 dimensional state into a 2 dimensional regression vector as part of LSM implementation. This all fails when the transaction is not a standard Bermudan swaption, or with a different simulation model (say, with stochastic volatility, when volatility state is another key regression variable). The methodology cannot be applied to arbitrary schedules of cash flows, simulated in arbitrary models. In practice, *regression cannot learn from data alone*. This is why it doesn't qualify as *artificial intelligence* (AI). Regression is merely a fitting procedure. Intelligence lies in the selection of basis functions, which is performed by hand and hardcoded as a set of rules. In fact, the whole thing can be seen as a neural network (NN) with fixed, nonlearnable hidden layers, as shown in Figure 1.

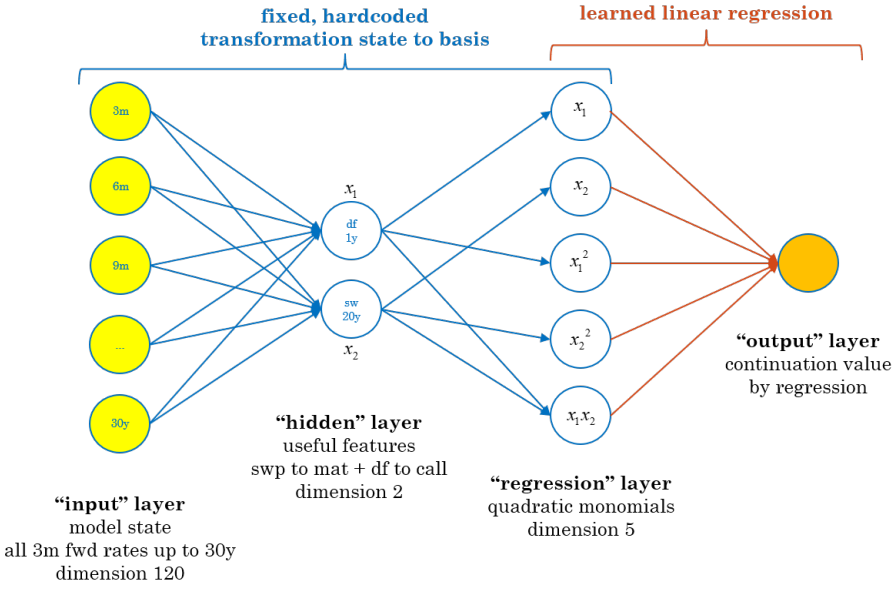

Figure 1: LSM regression as NN with fixed hidden layers

By contrast, neural networks are 'intelligent' constructs, capable of learning from data alone. NN are extensions of classic regression. In fact, they are identical to regression safe for one crucial difference: NN internalize the selection of basis functions and *and learn them from data*. For example, consider a NN with 4 hidden layers of 20 softplus activated units. The output layer is a classic regression over the basis functions identified in the *regression layer* (the last hidden layer). When the NN is trained by minimization of the MSE, the hidden weights learn to encode the 20 'best' basis functions among the (very considerable) space of functions attainable with the deep architecture. Optimization finds the best basis functions in the sense of the MSE, which itself approximates the distance between the true function and the space spanned by the basis functions. Hence, training a neural network really boils down to finding the appropriate, low dimensional regression space, often called *feature extraction* in machine learning (ML).

The strong similarity of NN to regression is illustrated on Figure 2 where we also see the one major difference: hidden layers are no longer fixed, they have learnable connection weights.

This is what makes NN so powerful, e.g. for solving high dimensional problems in computer vision or natural language processing. NN cruise through the curse of dimensionality be *learning* the limited dimension space that

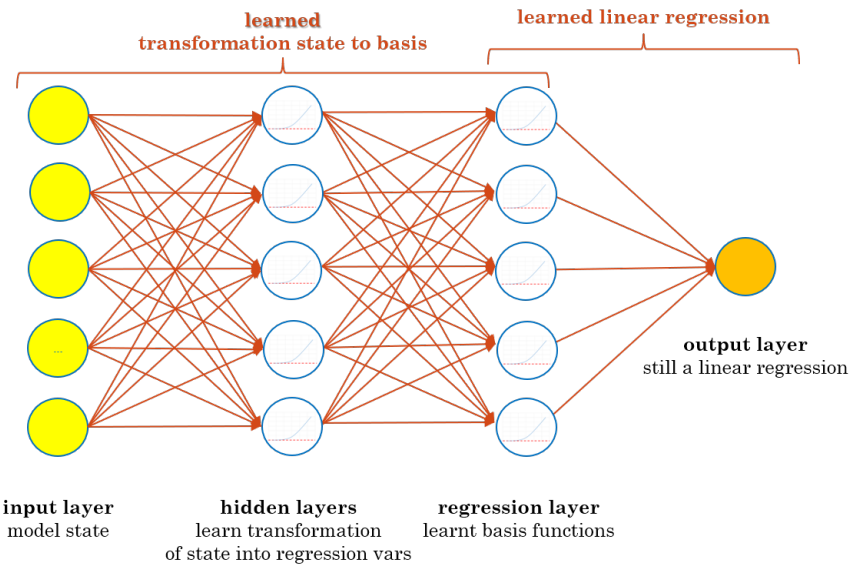

Figure 2: NN with two learnable hidden layers

best approximates the target function. They learn from data alone without handcrafted rules based on prior knowledge or specific to a given context. In particular, NN effectively approximate high dimensional functions from data alone, in finance or elsewhere, and they may even outperform regression on handcrafted features, i.e. find better basis functions from data than those extracted by hand from prior knowledge.

In return, training a neural network is famously *not* a convex problem. NN generally include many learnable connection weights ($1224 + 20n$ with 4 hidden layers of 20 units) and the MSE function of the weights has been shown to be of complicated topology, including multiple local minima and saddle points. There famously exists no algorithm guaranteed to find the global minimum in finite time. Despite extremely active research resulting in powerful *heuristic* improvements, training NN remains an art as much as a science. NN are often trained by hand over long periods where engineers slowly tweak architecture and hyperparameters until they obtain the desired behaviour in a given context. Modern training algorithms 'generally' converge to 'acceptable' minima, but such terms don't cut ice in mathematics, and they shouldn't in risk management either.

In order to automate training and implement its automatic execution, behind the scenes and *without human supervision*, we need sufficient mathematical guarantees. While it may look at first sight as a hopeless endeavour, we will see that the analysis performed in this paragraph allows to combine NN with regression in a meaningful and effective manner to establish important worst case guarantees.

1.2 Wide *and* deep

While regression is often opposed to deep learning in literature, a natural approach is to combine their benefits, by regression on both learnable *deep* units and fixed *wide* units, as illustrated in Figure 3. Mathematically, the ouput layer is a linear regression on the concatenation of the deep layer $z^{[L-1]}$ (the last hidden layer of the deep network) and a set of fixed basis function ϕ (a.k.a. the wide layer):

$$\hat{f}(x; w) = z^{[L-1]}(x; w_{hidden})\, w_{deep} + \phi(x)\, w_{wide}$$

and it immediately follows that the differentials of predictions wrt inputs are:

$$\frac{\partial \hat{f}(x; w)}{\partial x_j} = \frac{\partial z^{[L-1]}(x; w_{hidden})}{\partial x_j} w_{deep} + \frac{\partial \phi(x)}{\partial x_j} w_{wide}$$

where $\partial z^{[L-1]}/\partial x$ are differentials of the deep network computed by backpropagation (actually a platform like TensorFLow can compute the whole gradient of the wide and deep net behind the scenes) and $\partial\phi/\partial x$ are the known derivatives of the fixed basis functions. Hence, the architecture is trivially implemented by:

1. Add a classic regression term $\phi(x)\, w_{wide}$ to the output of the deep neural network.
2. Adjust the gradients of output wrt inputs (given by backpropagation through the deep network) by $(\partial\phi(x)/\partial x_j)\, w_{wide}$ (or leave it to TensorFlow).

An implementation in code is given in Geron's textbook, second ed. chapter 10.

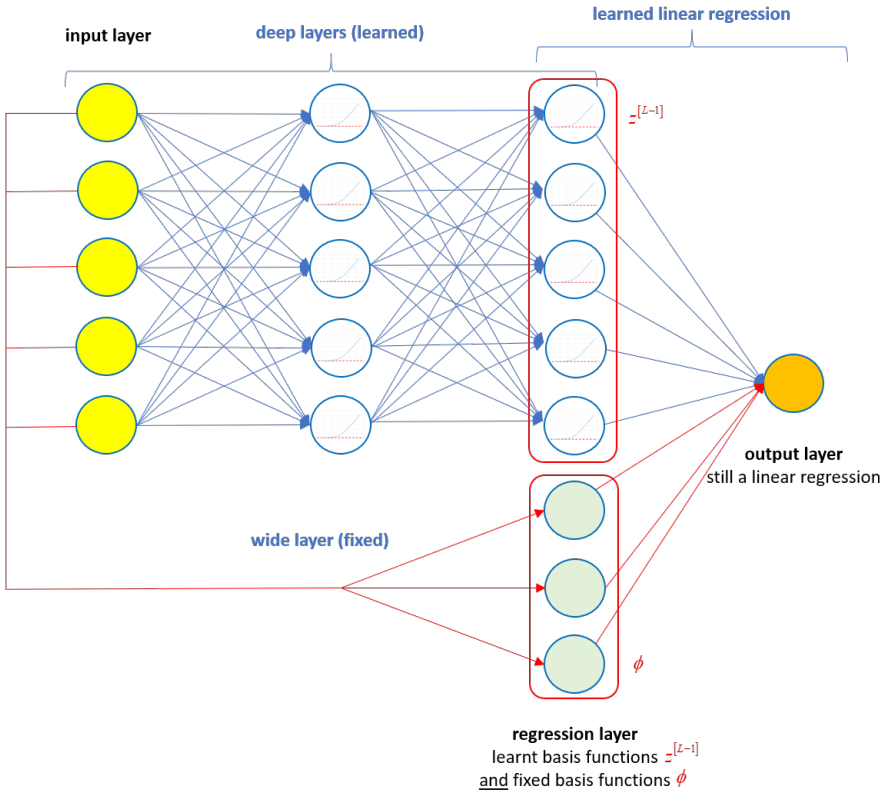

Figure 3: Wide and deep architecture

The idea is natural, and it is certainly not new. It was popularized by Google under the name "Wide and Deep Learning", in the context of recommender systems (`https://arxiv.org/abs/1606.07792`), although from a different perspective. Specifying a number of fixed regression functions in the wide layer should help training by restricting search for additional basis functions in the deep layers to dissimilar functions. For example, when the wide layer is a copy of the input layer x ($\phi = id$), it handles all linear functions of x and specializes the deep layers to a search for nonlinear functions (since another linear function in the deep regression layer would not reduce the MSE). In other terms, Google presented the wide and deep architecture as a training improvement, and it may well be that it does improve performance significantly with very deep, complex architectures. In our experience, the improvement is marginal with the simple architecture sufficient for pricing function approximation, but the wide and deep architecture still has a major role to play, because it provides guarantees and allows a safe implementation of automated training without supervision.

It is general wisdom that minimization of the MSE with NN doesn't offer any sort of guarantee. This is not entirely correct, though. Consider the MSE as a function of the connection weights of the output layer alone.

This is evidently a convex function. In fact, since the output layer is exactly a linear regression on the regression layer, the optimal weights are even given in closed form by the normal equation:

$$\hat{w}_{\text{output}} = \left(z^{[L-1]^T} z^{[L-1]}\right)^{-1} z^{[L-1]^T} Y$$

or its SVD equivalent (see appendix 3). Recall, while numerical optimization may not find the global minimum, it is still guaranteed to converge to a point with uniformyl zero gradient. In particular, training converges to a point where the derivatives of the MSE to the *output* connection weights are zero. And since the MSE is convex in *those* weights, *the projection onto the basis space is optimal.* Training may converge to 'bad' basis functions, but the approximation *in terms of these basis functions* is always as good as it can be. It immediately follows that, with a deep and wide architecture, we have a meaningful worst case guarantee: the approximation is least as good as a linear regression on the wide units. In practice, we get an orders of magnitude better performance from the deep layers, but it is the worst case guarantee that gives us permission to train without supervision. In practice, convergence may be checked by measuring the norm of the gradient, or, optimization may be followed by an analytic implementation of the normal equation wrt the combined regression layer (ideally in the SVD form of appendix 3).

Of course, the worst case guarantee is only as good as the choice of the wide functions. An obvious choice is a straightforward copy of the input layer. The wide layer handles all linear functions of the inputs, hence the worst case result is a linear regression. Another strategy is also add the squares of the input layers, and perhaps the cubes, depending on dimension, but not the cross monomials, which would bring back the curse of dimensionality.

A much more powerful wide layer may be constructed in combination with differential PCA (see appendix 2), which reduces the dimension of inputs and orders them by relevance, in a basis where differentials are orthogonal. This means that the input column X_1 affects targets most, followed by X_2 etc. Because inputs are presented in a relevant hierarchy, we may build a meaningful wide layer with a richer set of basis functions applied to the most relevant inputs. For example, we could use all monomials up to degree 3 on the first two inputs (10 basis functions), monomials of degree less than two on the next three inputs (another nine basis functions), and the other $n-5$ inputs raised to power 1, 2 and maybe 3 (up to $3n-15$ additional functions). Because of the differential PCA mechanism, a plain regression on these basis functions bears acceptable results by itself, especially with differential regression (see appendix 3), and this is only the *worst case* guarantee, with orders of magnitude better average performance.

2 Asymptotic control

2.1 Elementary asymptotic control

2.1.1 Enforce linear asymptotics

Another important consideration for unsupervised training is the performance of the trained approximation on *asymptotics*. This is particularly crucial for risk management applications like value at risk (VAR), expected loss (EL) or FRTB, which focuses on the behaviour of trading books in extreme scenarios. Asymptotics are hard because they are generally learned from little to no data in edge scenarios. In other terms, the asymptotic behaviour of the approximation is an *extrapolation* problem and reliable extrapolation is always harder, for instance, polynomial regression absolutely cannot be trusted.

As always, we want to control asymptotics from data alone and not explore methods based on prior knowledge of the correct asymptotics. For instance, a European call is known to have flat left asymptotic and linear right asymptotic with slope 1. If we know that the transaction is a European call, the correct asymptotics could be enforced by a variety of methods, see e.g. Antonov and Piterbarg for cutting edge. But that only works when we know for a fact that we are approximating the value of a European call, and we are only interested in general algorithms without other knowledge than a simulated dataset.

This being said, it is generally considered fair game that pricing functions have linear asymptotics, with an unknown slope to be estimated from data. For instance, it is common practice to enforce a zero second derivative boundary condition when pricing with finite difference methods (FDM)[2]. Linear asymptotics are guaranteed for neural networks as long as the activations are asymptotically linear. This is the case e.g. for common RELU, ELU, SELU or softplus activations, but not sigmoid or tanh, which asymptotics are flat, hence, to be avoided for pricing approximation[3].

Figure 4 compares the asymptotics of polynomial and neural approximations for a call price in Bachelier's normal model, obtained with our demonstration notebooks *DifferentialML.ipynb* and *DifferentialRegression.ipynb* on `https://github.com/differential-machine-learning/notebooks` (8192 training examples). The trained approximation is voluntarily tested on an unreasonably wide range of inputs in order to highlight asymptotic behaviour. Unsurprisingly, polynomial regression terribly misbehaves whereas neural approximation fares a lot better due to linear extrapolation. The comparison is of course unfair. The performance of the neural net is only due to linear asymptotics, which can be equally enforced for linear regression[4]. The point here is that we want to enforce linear asymptotics[5], something given with neural networks doable with some effort with regression.

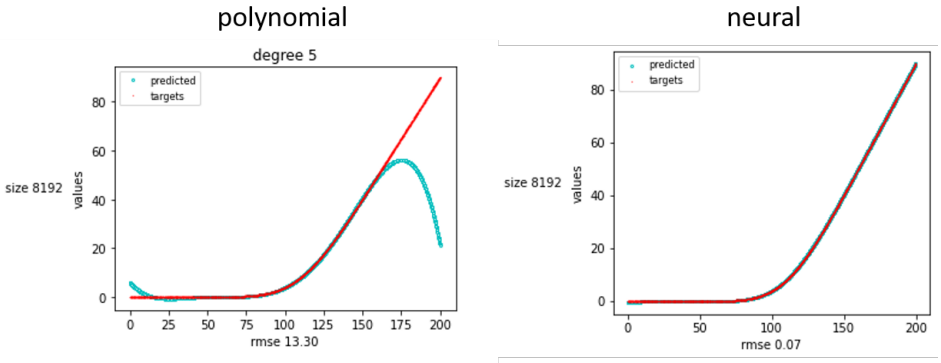

Figure 4: Asymptotics in a polynomial and neural approximation of a European call price

2.1.2 Oversample extreme scenarios

Enforcing linear asymptotics is not enough, we must also learn the slope from data. What makes it difficult is the typical sparsity of data on the edges of the training domain, especially when training data is produced with Monte-Carlo, and noisy labels e.g. from LSM simulations certainly don't help.

By far, the easiest walkaround is to simulate a larger number of edge examples. Recall, we may sample training inputs in any way we want. It is only the labels that must be computed in complete agreement with the pricing model, either by conditional sampling (sample labels) or conditional expectation (ground truth labels). Hence, we sample training examples over a domain and with a distribution reflecting the intended use of the trained approximation. In applications where asymptotics are important, we want many training examples in edge scenarios.

When training examples are sampled with Monte-Carlo simulations, it is particularly simple to oversample extreme scenarios *by increasing volatility from today to the horizon date* in the generative simulations. We implemented this simple method in our demonstration notebooks. As expected, increasing volatility to horizon

[2]Although overreliance on this common assumption may be dangerous: the Derivatives industry lost billions in 2008 on variance swaps and CMS caps, precisely due to nonlinear asymptotics.

[3]Recall that *differential* deep learning requires C^1 activation, ruling out RELU and SELU and leaving only the very similar ELU or softplus among common activations.

[4]By cropping basis functions with linear extrapolation outside of a given domain.

[5]While maintaining awareness that not all transactions are linear on edges.

date effectively resolves asymptotic behaviour, as illustrated on Figure 5 for degree 5 polynomial regression of the Bachelier call price.

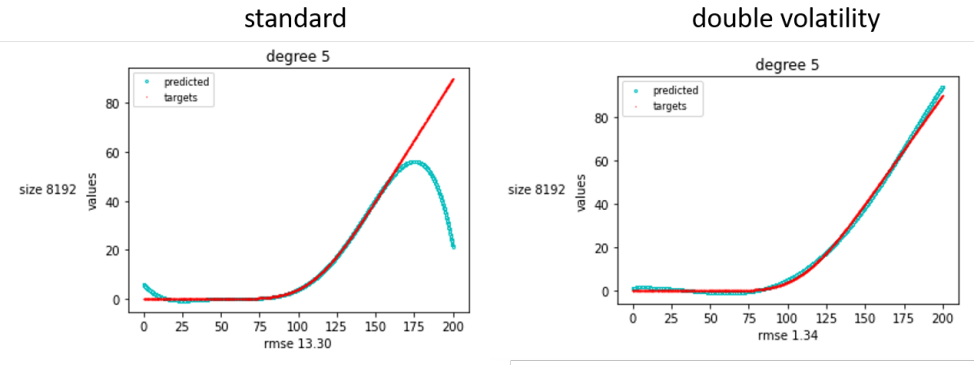

Figure 5: Fixing polynomial asymptotics with increased volatility

Note that increased volatility fixes asymptotics but deteriorates accuracy in the interior domain. By enforcing more examples on edge scenarios, we learn from fewer examples elsewhere, resulting in a loss of quality.

Further, it is fair game to increase volatility or change model parameters in any way *before horizon*, but the (conditional) simulation *after horizon date* must exactly follow the pricing model or we get biased labels. With multiple horizon dates, the simple walkaround no longer works: between two horizon dates, we cannot simultaneously simulate an increased volatility for the state variables, and the original volatility for the cashflows[6].

For these reasons, our simple walkaround is certainly not an optimal solution, but it is extremely simple to comprehend and implement, with reasonable performance for such a trivial method. The more advanced algorithms introduced next perform a lot better, but with significant implementation effort.

2.2 Advanced asymptotic control

2.2.1 Ground truth labels in edge examples

In the main article and appendix 1, we have opposed *ground truth learning*, where labels are numerically computed conditional expectations, to *sample learning* where labels are samples drawn from the conditional distribution, e.g. by simulation of one Monte-Carlo path. We concluded that ground truth learning is not viable in many contexts of practical relevance because of the computational cost of conditional expectations, and that sample learning offers a viable, consistent alternative.

The opposition between the two doesn't have to be black and white. In fact, many intermediate solutions exist. Ground truth labels are computed by averaging a large number of samples, theoretically infinity. Sample labels are (averages of) one sample each. We could as well compute labels by averaging an intermediate number N of samples, reducing variance by $\sqrt{N}$ in return for a computation cost linearly increasing in N. Notice that in the demonstration notebooks, we computed labels by averaging *two* antithetic paths. As a result, the variance of the noise is reduced by a factor two, but we can simulate half as many examples for a fixed computation load. Hence, benefits balance out but we get the additional benefit of antithetic sampling, making it worthwhile.

This realization leads to a powerful asymptotic control algorithm in the context of differential machine learning, where differential labels (gradients of labels wrt inputs) are available too. Identify a small number of edgemost examples in the training set, e.g. by Gaussian likelihood of inputs. Recall that 'extreme' scenarios mean extreme *inputs* here, irrespective of labels.

[6]One solution is to simulate Monte-Carlo paths with a form of *importance sampling* where samples are normalized by a likelihood ratio to compensate for increased volatility. However, importance sampling may raise numerical problems, since likelihood ratios between measures with different volatilities diverge in the continuous limit.

Train on sample labels for interior examples and ground truth labels, including ground truth differentials, for edgemost examples. Assign a larger weight to the extreme examples in the cost function to treat them as a constraint. The resulting approximation (provided linear asymptotics) will have the correct asymptotic behaviour beyond edge points, where intercepts and slopes are given by construction by ground truth values and gradients. Figure 6 displays 1024 inputs sampled from a bidimensional Gaussian distribution, where 16 edge examples are identified by Gaussian likelihood.

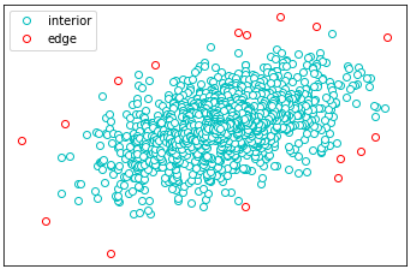

Figure 6: Interior and edge training inputs sampled from a bidimensional correlated Gaussian distribution

2.2.2 Compute ground truth labels with *one* Monte-Carlo path

Hence, we can control asymptotics effectively with ground truth labels for a small number of edge examples. Contrarily to the simple idea of oversampling extreme scenarios, this method fixes asymptotics without damaging approximation, since it doesn't modify interior examples in any way. The training set is still generated in reasonable time, since only a small number of extreme inputs gets costly ground truth labels.

However, computation cost may still be considerable compared to a standard LSM dataset. For example, say we simulate 8192 training examples, mostly with sample labels, but in the 64 least likely scenarios, we produce ground truth labels with 32768 nested Monte-Carlo paths. We simulate a total of $64 \times 32768 + 8128 = 2105280$ paths against 8192, a computation load increased by a factor 257.

It turns out that, at least in the context of financial pricing approximation from a Monte-Carlo dataset, we can actually compute ground the true edge values and risks for the cost of *one* Monte-Carlo path, and effectively fix asymptotics without additional cost.

The common assumption of linear asymptotics comes from that the value of many financial Derivatives converges to *intrinsic value* in extreme scenarios. In general terms, the intrinsic value corresponds to the payoff evaluated on the *forward scenario*, where all underlying instruments fix on their forward values, conditional to initial (extreme) state. See e.g. chapter 4 of Modern Computation Finance (Wiley, 2018) for details.

Hence, under the assumption of linear asymptotics, the value and Greeks in edge states are computed for the cost of one Monte-Carlo path, generated from the forward underlying asset prices computed from the (extreme) state variables. By hypothesis of linear asymptotics (a.k.a. asymptotically intrinsic values), this procedure computes correct prices (and Greeks) in extreme scenarios (and only in those). The practical implementation is dependent on the specifics of simulation systems. The idea is illustrated on Figure 7.

All labels are computed by sampling payoffs on one path conditional to the scenario, but for edge scenarios, we use a special inner simulation, the 'forward path', where payoffs coincide with intrinsic values and pathwise differentials coincide with true Greeks. With the constraint to match these amounts exactly in the cost function, we effectively control asymptotics without additional computational cost or damage to approximation quality.

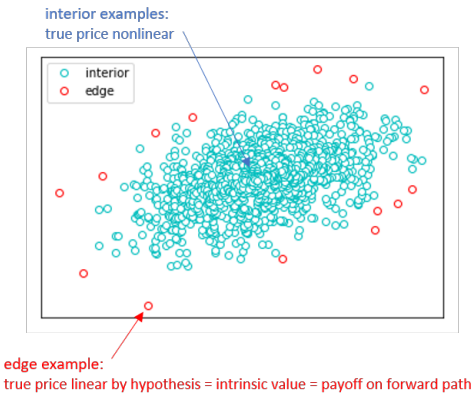

Figure 7: Computation of true prices in interior and edge examples

Conclusion

Contrarily to common wisdom, training neural networks does provide some guarantees. In particular, the regression of the output layer on the basis functions identified in the regression layer is guaranteed optimal. We built on this observation to combine classic regression with deep learning in a wide and deep architecture *a la* Google and establish worst case training guarantees. We also discussed the particular effectiveness of the wide and deep architecture when combined with differential PCA presented in appendix 2.

We also covered elementary and advanced asymptotic control algorithms, the most advanced ones, implemented with some effort, being capable of producing correct asymptotics without additional computation cost or stealing data from the interior domain. The algorithm requires differential labels and only works with differential machine learning. In financial Derivatives risk management, differential labels are easily and very efficiently produced with automatic adjoint differentiation (AAD).