Differential Machine Learning

Brian Huge
brian.huge@danskebank.dk

Antoine Savine
antoine.savine@danskebank.dk

Written January 2020 Published May 2020

Introduction

Differential machine learning (ML) is an extension of supervised learning, where ML models are trained on examples of not only inputs and labels but also *differentials of labels to inputs.*

Differential ML is applicable in situations where high quality first order derivatives wrt training inputs are available. In the context of financial Derivatives and risk management, *pathwise differentials* are efficiently computed with automatic adjoint differentiation (AAD). Differential machine learning, combined with AAD, provides extremely effective pricing and risk approximations. We can produce fast pricing analytics in models too complex for closed form solutions, extract the risk factors of complex transactions and trading books, and effectively compute risk management metrics like reports across a large number of scenarios, backtesting and simulation of hedge strategies, or regulations like XVA, CCR, FRTB or SIMM-MVA.

The article focuses on differential *deep* learning (DL), arguably the strongest application. Standard DL trains neural networks (NN) on punctual examples, whereas differential DL teaches them *the shape* of the target function, resulting in vastly improved performance, illustrated with a number of numerical examples, both idealized and real world. In the online appendices, we apply differential learning to other ML models, like classic regression or principal component analysis (PCA), with equally remarkable results.

This paper is meant to be read in conjunction with its companion GitHub repo `https://github.com/differential-machine-learning`, where we posted a TensorFlow implementation, tested on Google Colab, along with examples from the article and additional ones. We also posted appendices covering many practical implementation details not covered in the paper, mathematical proofs, application to ML models besides neural networks and extensions necessary for a reliable implementation in production.

The authors thank Ove Scavenius and their colleagues of Superfly Analytics, Danske Bank's quantitative research department, for thoughtful suggestions and comments. The advanced numerical results of section Section 3 were produced with the proprietary implementation in Danske Bank's production risk management system. We also thank Bruno Dupire, Jesper Andreasen and Leif Andersen for a substantial contribution through meaningful review and sometimes heated discussions, all of which resulted in a considerable improvement of the contents.

Pricing approximation and machine learning

The approximation of pricing functions is a persistent challenge in quantitative finance. In the early 1980s, researchers already experimented with moment matching approximations for Asian and Basket options, or Taylor expansions for stochastic volatility models. Iconic results were derived in the 1990s, including Hagan's SABR formula [13] or Musiela's swaption pricing formula in the Libor Market Model [5], and new results are being published regularly, either in traditional form ([1] [3]) or by application of advances in machine learning.

Traditionally, pricing approximations were derived by hand, but, in the recent years, automated techniques borrowed from the fields of artificial intelligence (AI) and ML attracted a lot of attention. The general format is to approximate asset pricing functions of a set of inputs x (market variables, path-dependencies, model and instrument parameters), with a function $\hat{f}(x; w)$ subject to a collection of weights w, learned from a training set of m examples of inputs (each a vector of dimension n) paired with labels (or 'targets', typically real numbers), by minimization of a cost function (often the mean squared error between predictions and targets).

For example, the recent works of [23] and [17] trained neural networks to price European calls[1], respectively in the SABR model and the 'rough' volatility family of models [9]. Their training sets included a vast number of examples, labeled by ground truth prices, computed by numerical methods. This approach essentially interpolates prices in parameter space, and works with computationally expensive training sets to produce approximations reusable in many contexts. Effective neural approximations make sophisticated models like rough Heston practically usable e.g. to simultaneously fit SP500 and VIX smiles, see [11].

ML models generally learn approximations from training data alone, without knowledge of the generative simulation model or financial instrument. Although performance may be considerably improved on a case by case basis with contextual information (such as the nature of the transaction), the most powerful, general ML implementations achieve accurate approximation from data alone.

The trained approximation computes prices (and risks, by differentiation wrt inputs) with (near) analytic speed.

Function approximation in general, and by ML in particular, is also getting traction in the context of multiple scenario risk reports, backtesting or regulations like XVA, CCR, FRTB or SIMM-MVA, where the values and risk sensitivities of Derivatives trading books are repeatedly computed in many different market states. A correct approximation could execute the repeated computations with analytic speed and resolve considerable computation bottlenecks.

However, the precedent approach is not viable in this context. With the computational complexity of numerical ground truth prices, the generation of the training set is unreasonably expensive, only to learn the value of a specific Derivatives book for the computation of one risk report or regulation, at a given moment in time. Fortunately, the classic Least Square Method (LSM, also known as 'Longstaff-Schwartz') of [22] and [6] offers an alternative to simulate entire training sets for a cost comparable to *one* computation of the price by Monte-Carlo. Instead of ground truth prices, labels correspond to *one* sample of the payoff (sum of cash-flows after some horizon date), drawn from its correct conditional distribution, computed for the cost of *one* Monte-Carlo path. Because samples are independent, unbiased estimates of the conditional expectation, approximators trained on LSM datasets still converge to the true pricing function. This result was formally demonstrated in the founding papers for linear combinations of fixed basis functions, and carries over to arbitrary approximators with universal approximation property, including neural networks. We posted a general proof in the online appendices.

Modern deep learning models, trained on LSM datasets, approximate prices more effectively than classic linear models, especially in high dimension. The extension of LSM to deep learning was explored in many recent works like [21], along with evidence of a considerable improvement, in the context of Bermudan options, although the conclusions carry over to arbitrary schedules of cash-flows. Neural networks effectively resolve the long standing *curse of dimensionality*. Contrarily to classic regression models, they are resilient in high dimension, therefore particularly relevant in the context of complex transactions and trading books, see e.g. [4].

Training with derivatives

We found that the performance of modern deep learning remains insufficient in the context of complex transactions or trading books, where a vast number of training examples (often in the hundreds of thousands or millions) is necessary to learn accurate approximations, so the training set cannot be simulated in reasonable time, even under LSM. Training on samples is prone to overfitting, and unrealistic numbers of training examples are necessary even in the presence of regularization. Finally, risk sensitivities converge considerably slower than values and often remain blatantly wrong, even with hundreds of thousands of examples.

We resolve these problems by training ML models on datasets *augmented with differentials* of labels wrt inputs:

[1] Early exploration of neural networks for pricing approximation [19] or in the context of Longstaff-Schwartz for Bermudan and American options [14] were published over 25 years ago, although it took modern deep learning techniques to achieve the performance demonstrated in the recent works.

$$x^{(i)} \ , \ y^{(i)} \ , \ \frac{\partial y^{(i)}}{\partial x^{(i)}}$$

This simple idea, along with the adequate training algorithm, allows ML models to learn accurate approximations even from small LSM datasets, making ML viable in the context of trading books and regulations.

When learning from ground truth labels, the input $x^{(i)}$ is one example parameter set of the pricing function. If we were learning Black and Scholes' pricing function, for instance, (without using the formula, as if we didn't know it)[2], $x^{(i)}$ would be one possible set of values for the initial spot price, volatility, strike and expiry (ignoring rates or dividends). The label $y^{(i)}$ would be the (ground thruth) call price computed with these inputs (by Monte-Carlo or Finite Difference Methods since we don't know the formula), and the derivatives labels $\partial y^{(i)} / \partial x^{(i)}$ would be the Greeks.

When learning from LSM samples, the input $x^{(i)}$ is one example state. In the Black and Scholes example, $x^{(i)}$ would be the spot price at some future date $T_1 \geq 0$, called *exposure date* in the context of regulations, or *horizon date*, picked on path i. The label $y^{(i)}$ would be the payoff of a call expiring on a later date T_2, picked on that same path. The exercise is to learn a function of S_{T_1} approximating the value of the T_2 call, priced at T_1. In this case, the differential labels $\partial y^{(i)} / \partial x^{(i)}$ are the *pathwise derivatives* of the payoff at T_2 wrt the state at T_1, computed on path i. In Black and Scholes:

$$\frac{\partial y^{(i)}}{\partial x^{(i)}} = \frac{\partial \left(S_{T_2}^{(i)} - K \right)^+}{\partial S_{T_1}^{(i)}} = \frac{\partial \left(S_{T_2}^{(i)} - K \right)^+}{\partial S_{T_2}^{(i)}} \frac{\partial S_{T_2}^{(i)}}{\partial S_{T_1}^{(i)}} = 1_{\left\{ S_{T_2}^{(i)} > K \right\}} \frac{S_{T_2}^{(i)}}{S_{T_1}^{(i)}}$$

We further review pathwise derivatives in Section 2.2.

Learning from ground truth labels is slow, but the learned function is reusable in many contexts. This is the correct manner to learn e.g. European option pricing functions in stochastic volatility models. Learning from LSM samples is fast, but the learned approximation is a function of the state, specific to a given financial instrument or trading book, under a given calibration of the model. This is how we can quickly approximate the value and risks of complex transactions and trading books, e.g. in the context of regulations. Differential labels vastly improve performance in both cases.

Differentials classically intervene as constraints in various *interpolation* methods. Differential *regularization* has been applied to model fitting (e.g. regression or calibration of financial models), with penalties generally imposed on the *norm* of differentials, e.g. the size of second order differentials, expressing a preference for linear functions. Our proposition is different. We do not to express *preferences*, we enforce differential *correctness*, measured by proximity to *differential labels.* An application of differential labels was independently proposed in the recent [7], in the context of high dimensional semi linear partial differential equations. Our algorithm is general. It applies to either ground truth learning (closely related to interpolation) or sample learning (related to regression). It consumes derivative sensitivities for ground truth learning or pathwise differentials for sample learning. An exact computation of the differential labels is achieved for very little computation cost with automatic adjoint differentiation (AAD).

Effective differential labels with AAD

Differential ML consumes differential labels $\partial y^{(i)} / \partial x^{(i)}$ in an augmented training set. The differentials must be *accurate* or the optimizer might get lost chasing wrong targets, and they must be computed quickly, even in high dimension, for the method to be applicable in realistic contexts. Conventional differentiation algorithms like finite differences fail on both counts. It is the promise of AAD to automatically compute the differentials of arbitrary computations, with analytic accuracy, for a computation cost proportional to *one* functional evaluation of the price, irrespective of dimension[3].

[2] See the online notebook for a complete implementation.

[3] It takes the time of 2 to 5 evaluations in practice to compute thousands of differentials with an efficient implementation, see [28].

AAD was introduced to finance in the ground breaking 'Smoking Adjoints' [12]. It is closely related to backpropagation, which powers the core of modern deep learning and has largely contributed to its recent success. In finance, AAD produces risk reports in real time, including for exotic books or XVA. In the context of Monte-Carlo or LSM, AAD produces exact pathwise differentials for a very small cost. AAD made differentials massively available in quantitative finance. Besides evident applications to instantaneous calibration or real-time risk reports, the vast amount of information contained in differentials may be leveraged in creative ways, see e.g. [24] for an original application. Differential ML is another strong application.

In the interest of brevity, we refer to [28] for a comprehensive description of AAD, including all details of how training differentials were obtained in this study, or the video tutorial [25], which explains the main ideas in 15 minutes.

The paper is structured as follows. In Section 1 we introduce notations to describe feedforward neural networks and backpropagation. The core algorithm of the paper is described in Section 1.2 and continues with training in Section 2. Section 3 shows numerical results in various textbook and real-world situations[4]. Section 4 extends the algorithm to multiple outputs and higher order derivatives.

1 Differential deep learning

Because our methodology learns from data alone, it is completely general and applicable to arbitrary schedules of cash-flows, e.g. as described with a cash-flow scripting language, simulated in arbitrary models. This section describes the training algorithm in the context of feedforward neural networks, although it carries over to NN of arbitrary complexity in a straightforward manner. At this stage, we assume the availability of a training set augmented with differentials, and are not concerned with how they were produced in the first place, wither analytical, numerical, estimated from empirical data or computed by AAD.

1.1 Notations

1.1.1 Feedforward equations

In this section, we introduce notations for standard feedforward networks. Define the input (row) vector $x \in \mathbb{R}^n$ and the predicted value $y \in \mathbb{R}$. For every layer $l = 1, \dots, L$ in the network, define a scalar 'activation' function $g_{l-1} : \mathbb{R} \to \mathbb{R}$. Popular choices are relu, elu and softplus, with the convention $g_0(x) = x$ is the identity. The notation $g_{l-1}(x)$ denotes elementwise application. We denote $w_l \in \mathbb{R}^{n_{l-1} \times n_l}, b_l \in \mathbb{R}^{n_l}$ the weights and biases of layer l.

The network is defined by its feedforward equations:

$$\begin{aligned} z_0 &= x \\ z_l &= g_{l-1}\left(z_{l-1}\right) w_l + b_l \quad , l = 1, \dots, L \\ y &= z_L \end{aligned} \tag{1}$$

where $z_l \in \mathbb{R}^{n_l}$ is the row vector containing the n_l pre-activation values, also called *units* or *neurons*, in layer l. Figure 1 illustrates a feedforward network with $L = 3$ and $n = n_0 = 3, n_1 = 5, n_2 = 3, n_3 = 1$, together with backpropagation.

1.1.2 Backpropagation

Feedforward networks are easily differentiated due to their simple mathematical structure. This is known as backpropagation and generally applied to compute the derivatives of some some cost function (we discuss it

[4] Some of which are replicated in the TensorFlow notebook on our GitHub.

shortly) wrt the weights and biases for optimization. For now, we are not interested in those differentials, but in the differentials of the *predicted* value $y = z_L$ wrt the *inputs* $x = z_0$. Recall, for pricing approximation, inputs are states and predictions are prices, hence, these differentials are risk sensitivities (*Greeks*), obtained by differentiation of the lines in (1), in the reverse order:

$$\begin{aligned} \bar{z}_L &= \bar{y} = 1 \\ \bar{z}_{l-1} &= \left(\bar{z}_l w_l^T\right) \circ g'_{l-1}\left(z_{l-1}\right) \quad , l = L, \ldots, 1 \\ \bar{x} &= \bar{z}_0 \end{aligned} \tag{2}$$

with the *adjoint* notation $\bar{x} = \partial y / \partial x, \bar{z}_l = \partial y / \partial z_l, \bar{y} = \partial y / \partial y = 1$ and $\circ$ is the elementwise (Hadamard) product.

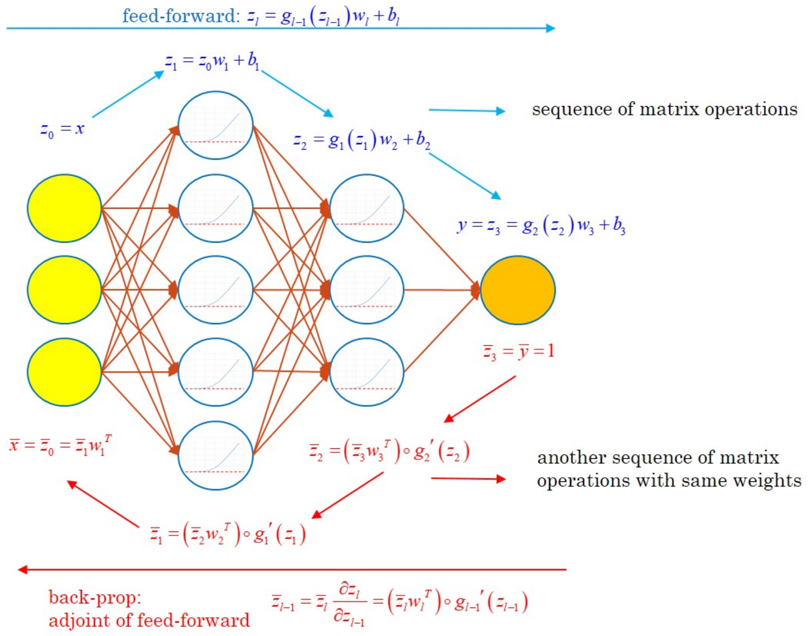

feed-forward: $z_l = g_{l-1}(z_{l-1}) w_l + b_l$
$z_1 = z_0 w_1 + b_1$
$z_0 = x$
$z_2 = g_1(z_1) w_2 + b_2$
sequence of matrix operations
$y = z_3 = g_2(z_2) w_3 + b_3$
$\bar{z}_3 = \bar{y} = 1$
$\bar{x} = \bar{z}_0 = \bar{z}_1 w_1^T$
$\bar{z}_2 = (\bar{z}_3 w_3^T) \circ g_2'(z_2)$
another sequence of matrix operations with same weights
$\bar{z}_1 = (\bar{z}_2 w_2^T) \circ g_1'(z_1)$
back-prop: adjoint of feed-forward
$\bar{z}_{l-1} = \bar{z}_l \frac{\partial z_l}{\partial z_{l-1}} = (\bar{z}_l w_l^T) \circ g_{l-1}'(z_{l-1})$

Figure 1: feedforward neural network with backpropagation

Notice, the similarity between (1) and (2). In fact, backpropagation defines a second feedforward network with inputs $\bar{y}, z_0, \ldots, z_L$ and output $\bar{x} \in \mathbb{R}^n$, where the weights are shared with the first network and the units in the second network are the adjoints of the corresponding units in the original network.

Backpropagation is easily generalized to arbitrary network architectures, as explained in a vast amount of deep learning literature. It may even be generalized to arbitrary computations unrelated to deep learning or AI, in which case we call it AD. See our video tutorial [25] for details. Backpropagation can also be implemented to differentiate computations automatically, behind the scenes. ML frameworks like TensorFlow implicitly invoke it in training loops. The most general incarnation of automated backpropagation is called AAD.

1.2 Twin networks

Derivatives from feedforward networks form another neural network, efficiently computing risk sensitivities in the context of pricing approximation. Since the adjoints form a second network, we might as well use them for training and should expect significant performance gain:

1. The network is explicitly trained on derivatives and always returns solid Greeks.

2. The network doesn't learn *points* but *shapes*, which is far more efficient.

To make the approximation of differentials part of the network, we continue the feedforward equations (1) after the computation of the prediction y, through the backpropagation layers (2). We call it *twin network*. Its first half is the standard feedforward network, its second half is its mirror image, which feedforward equations are the adjoint equations of the first network, with shared weights. A mathematical description is given by concatenation of (1) and (2), illustrated in Figure 2:

$$\begin{aligned} z_0 &= x \\ z_l &= g_{l-1}\left(z_{l-1}\right) w_l + b_l \quad ,l = 1, \ldots, L \qquad (3) \\ y &= z_L \\ \bar{z}_L &= \bar{y} = 1 \\ \bar{z}_{l-1} &= \left(\bar{z}_l w_l^T\right) \circ g'_{l-1}\left(z_{l-1}\right) \quad ,l = L, \ldots, 1 \qquad (4) \\ \bar{x} &= \bar{z}_0 \end{aligned}$$

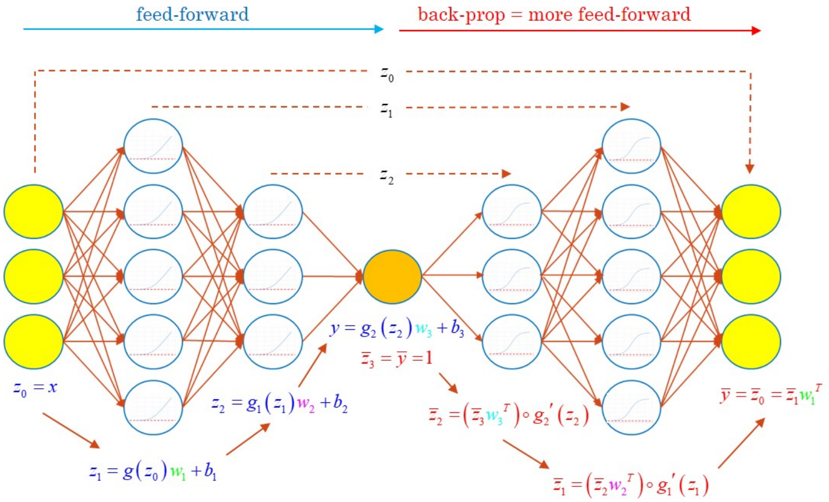

feed-forward
back-prop = more feed-forward
z_0
z_1
z_2
$z_0 = x$
$z_1 = g(z_0) w_1 + b_1$
$z_2 = g_1(z_1) w_2 + b_2$
$y = g_2(z_2) w_3 + b_3$
$\bar{z}_3 = \bar{y} = 1$
$\bar{z}_2 = (\bar{z}_3 w_3^T) \circ g_2'(z_2)$
$\bar{z}_1 = (\bar{z}_2 w_2^T) \circ g_1'(z_1)$
$\bar{y} = \bar{z}_0 = \bar{z}_1 w_1^T$

Figure 2: twin network

The twin network has $2L$ layers where weights w_l are shared between layers l and $2L + 1 - l$. E.g. layer 1 computing $z_1 = z_0 w_1 + b_1$ and layer $2L$ computing $\bar{z}_0 = \bar{z}_1 w_1^T$ share the weight matrix w_1. Also notice, that the values z_l computed in the first L layers are reused for the computation of the adjoints $\bar{z}_l$ in the last L layers through $g'_{l-1}\left(z_{l-1}\right)$. We differentiate activation functions explicitly in the twin network, hence, our activation functions must be C^1, which rules out e.g. relu.

The evaluation of the twin network with equations (3) and (4) returns a predicted value y, and its differentials $\bar{x}$ to the n_0 inputs x, for twice the computation complexity of value prediction alone. In the context of pricing, a trained twin approximates prices and risk sensitivities given inputs (or states) in a combined, and particularly efficient manner.

2 Differential training

2.1 With ground truth labels

Assume for now that we are training with ground truth labels, given by a function $f : \mathbb{R}^n \to \mathbb{R}$, which the network is meant to approximate by a function $\hat{f}\left(x; \{w_l, b_l\}_{l=1,\ldots,L}\right)$. Differential training is best understood in this context, although it identically applies to sampled labels, as we see shortly. The training set consists in m sample row vectors $x^{(1)}, \ldots, x^{(m)}$ with labels $y^{(1)} = f\left(x^{(1)}\right)$, …, $y^{(m)} = f\left(x^{(m)}\right)$. We also provide derivatives labels $\bar{x}^{(1)} = \partial f\left(x^{(1)}\right) / \partial\left(x^{(1)}\right)$, … , $\bar{x}^{(m)} = \partial f\left(x^{(m)}\right) / \partial\left(x^{(m)}\right)$.

As is customary with deep learning, we stack training data in matrices, with examples in rows and units in columns:

$$X = \begin{bmatrix} x^{(1)} \\ \vdots \\ x^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times n} \quad Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(m)} \end{bmatrix} \in \mathbb{R}^{m} \quad \bar{X} = \begin{bmatrix} \bar{x}^{(1)} \\ \vdots \\ \bar{x}^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times n}$$

Notice, the equations (3) and (4) identically apply to matrices or row vectors. Hence, the evaluation of the twin network computes the matrices:

$$Z_l = \begin{bmatrix} z_l^{(1)} \\ \vdots \\ z_l^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times n_l} \quad \text{and} \quad \bar{Z}_l = \begin{bmatrix} \bar{z}_l^{(1)} \\ \vdots \\ \bar{z}_l^{(m)} \end{bmatrix} \in \mathbb{R}^{m \times n_l}$$

Training consists in finding weights and biases such that the network predictions on the examples are close to the labels compute with the function f. This is achieved by minimization of a cost function in the weights and biases:

$$\min_{w_l, b_l} C\left(\{w_l, b_l\}_{l=1,\ldots,L}\right)$$

Standard regression cost is the mean squared error in values, without penalty for wrong derivatives. Differential training combines the cost of errors in values **and** derivatives:

$$C\left(\{w_l, b_l\}_{l=1,\ldots,L}\right) = \underbrace{\frac{1}{m}\left(Z_L - Y\right)^T \left(Z_L - Y\right)}_{\text{average cost of values}} + \underbrace{\lambda \frac{1}{m} \text{tr}\left[\beta^T \left(\bar{Z}_0 - \bar{X}\right)^T \left(\bar{Z}_0 - \bar{X}\right) \beta\right]}_{\text{average cost of derivatives}}$$

where tr is the trace of a matrix and β is the diagonal matrix of the weights, in the combined cost, of the derivative errors in the columns of $\bar{Z}_0 - \bar{X}$. The magnitude of the values Y and differentials $\bar{X}$ may be very different. For example, a 1y at-the-money call on a stock worth 10000 is worth order of 800 with volatility 20, with delta of order 0.5 and vega of order 40. Evidently, errors on such heterogeneous amounts cannot be simply added, and a sensible normalization is *essential* in any practical implementation. Weighting may also be generalized with a non diagonal matrix β e.g. to consider errors of directional, mutually orthonormal differentials. We posted a detailed appendix on GitHub, where we also introduce *differential PCA*.

This is the same cost function as the vanilla network, with an additional penalty for wrong differentials. Since the learnable parameters are unchanged, differential training acts as a regularizer, although of a very different nature than classical regularizers, which express *preferences*, e.g. on the size of parameters or derivatives, whereas differential training enforces their *correctness*. Standard regularizers cannot inject additional information in the training process, thereby helping correct the shape of the trained approximation. They only reduce the effective capacity of the model with arbitrary constraints, hereby introducing a bias. Differential training uses *additional* information contained in the differentials, hereby reducing variance *without bias*. We formalize and quantify these notions in the online appendices.

It follows, in particular, that the algorithm has little sensitivity to the hyperparameter λ, balancing the relative weight of prediction errors and regularization penalties. The excessive sensitivity to λ of standard regularizers considerably limits their practical application. Not with differential training: with infinite λ, we train of differentials alone, disregarding values, and still converge to the correct function, modulo an additive constant.

In fact, because differential training injects meaningful information computed for very little cost, it is more closely related to *data augmentation*, which, in turn, may be seen as a more effective form of regularization.

2.2 With sampled labels

As long as all the cash-flows are differentiable functions of market variables, all of the above carries over to LSM datasets with sampled payoffs in labels.

With non differentiable cash-flows like digitals or barriers, pathwise derivatives are not well defined. This is a well known problem, since Monte-Carlo risk sensitivities cannot be computed either. For this reason, traders always *smooth* cash-flows, representing digitals as tight call spreads and hard barriers as soft barriers. Smoothing is standard industry practice, it is even sometimes embedded and enforced in modern systems and cash-flows scripting languages[5].

With smooth(ed) payoffs, the conditional expectation and differentiation commute, so pathwise differentials are independent, unbiased estimates of the true risks, just like payoffs are independent, unbiased estimates of the true prices[6]. By the same argument, then, to train universal approximators by minimization of derivatives errors converges to an approximation with the correct differentials, i.e. the true pricing function, modulo an additive constant. We develop and formalize the argument in the online appendices. While it permits to train universal approximators on derivatives alone, the best numerical results are obtained in practice by combining values and derivatives errors in the cost function.

3 Numerical results

In this section, we show some numerical results and compare the performance of the twin network with its standard feedforward counterpart. We picked three examples from relevant textbook and real-world situations, where the purpose of the neural network is to learn pricing and risk approximations from small LSM datasets.

We kept neural architecture constant for all the examples, with 4 hidden layers of 20 softplus-activated units. Our neural networks are trained on mini-batches of normalized data, with the ADAM optimizer and a one-cycle learning rate schedule. All the details are given in the online notebook. Standard feedforward networks are regularized with cross-validated early stopping, considerably improving their performance. Twin nets don't require additional regularization besides derivatives labels. A differential training set takes 2-5 times longer to simulate with AAD, and training twin nets takes twice longer than standard nets, but we are going to see that they perform up to thousandfold better on small datasets.

3.1 Basket options

The first (textbook) example is a basket option in a correlated Bachelier model, for 7 assets[7]

$$dS_t = \sigma \, dW_t$$

[5] See [26] for an overview of smoothing, along with a general formalism drawn from similarities with fuzzy logic.

[6] A general proof of commutation of differentials with the conditional expectation is found in Dupire's Functional Ito Calculus [8].

[7] This example is reproduceable in the online notebook, where the number of assets is configurable, and the covariance matrix and basket weights can be re-generated randomly.

where $S_t \in \mathbb{R}^7$. The task is to learn the pricing function of a 1y call option on a basket, with strike 110 and initial basket value of 100 (all asset prices start at 100 without loss of generality, and weights are normalized to sum to 1). Evidently, the basket price is also Gaussian in this model, hence, the correct price is given by Bachelier's formula with the current basket, and we can easily measure the performance of the approximation. This example is of particular interest because, although the input space is 7 dimensional, we know from maths that pricing is really 1 dimensional. Can the network learn this property from data?

In Figure 3 we have trained a number of networks and used them to predict values and derivatives in 1024 independent scenarios, with current baskets on the horizontal axis and the option price/deltas on the vertical axis (we show 2 of the 7 derivatives), compared to the correct results computed with Bachelier's formula. The twin network was trained on only 1024 examples in the figure (although standard errors are also shown for larger datasets). Classical networks were trained on 1024 (1k), 8192 (8k) and 65536 (64k) samples, with cross-validation and early stopping. As we see, the twin network performs better for values and a lot better for derivatives. In particular, it learned that the option price and deltas are a fixed function of the basket, as evidenced by the thinness of the approximation curve. The classical network doesn't learn this property well, even with 64k examples. It overfits training data and predicts different values or deltas for different scenarios on the 7 assets with virtually identical baskets. Even with 64k data, the vanilla network performs worse than the twin network with 1k data, especially for the approximation of deltas.

Figure 4 compares standard errors of classical and twin approximations with standard Monte-Carlo (MC) errors[8]. Monte-Carlo is a primary pricing and risk management framework, and its error is generally deemed acceptable. We see that the standard error of the twin network stands below MC error for small datasets (although the network standard errors are sensitive to random seed and empirically representative up to a multiplicative factor around 2). The last line, produced with 8 *million* samples, verifies that both networks converge to similar low errors, on values and derivatives (*not* zero: there is an inherent bias to ML models of finite capacity). The twin network converges hundreds of times faster, in the sense that its performance on small datasets is comparable to the classical network trained on hundreds of times larger datasets.

3.2 Worst-of autocallables

As a second (real-world) example, we approximate an exotic instrument, a four-underlying version of the popular worst-of autocallable trade, in a more complicated model, a collection of 4 correlated local volatility models *a la* Dupire:

$$dS_t^j = \sigma_j \left(t, S_t^j \right) dW_t^j \quad j = 1, \ldots, 4$$

where $dW_t^j dW_t^k = \rho_{jk}$.

The example is relevant, not only due to the popularity of these exotic trades, but also because of path-dependence, barriers and massive final digitals, all of which are known to break ML models.

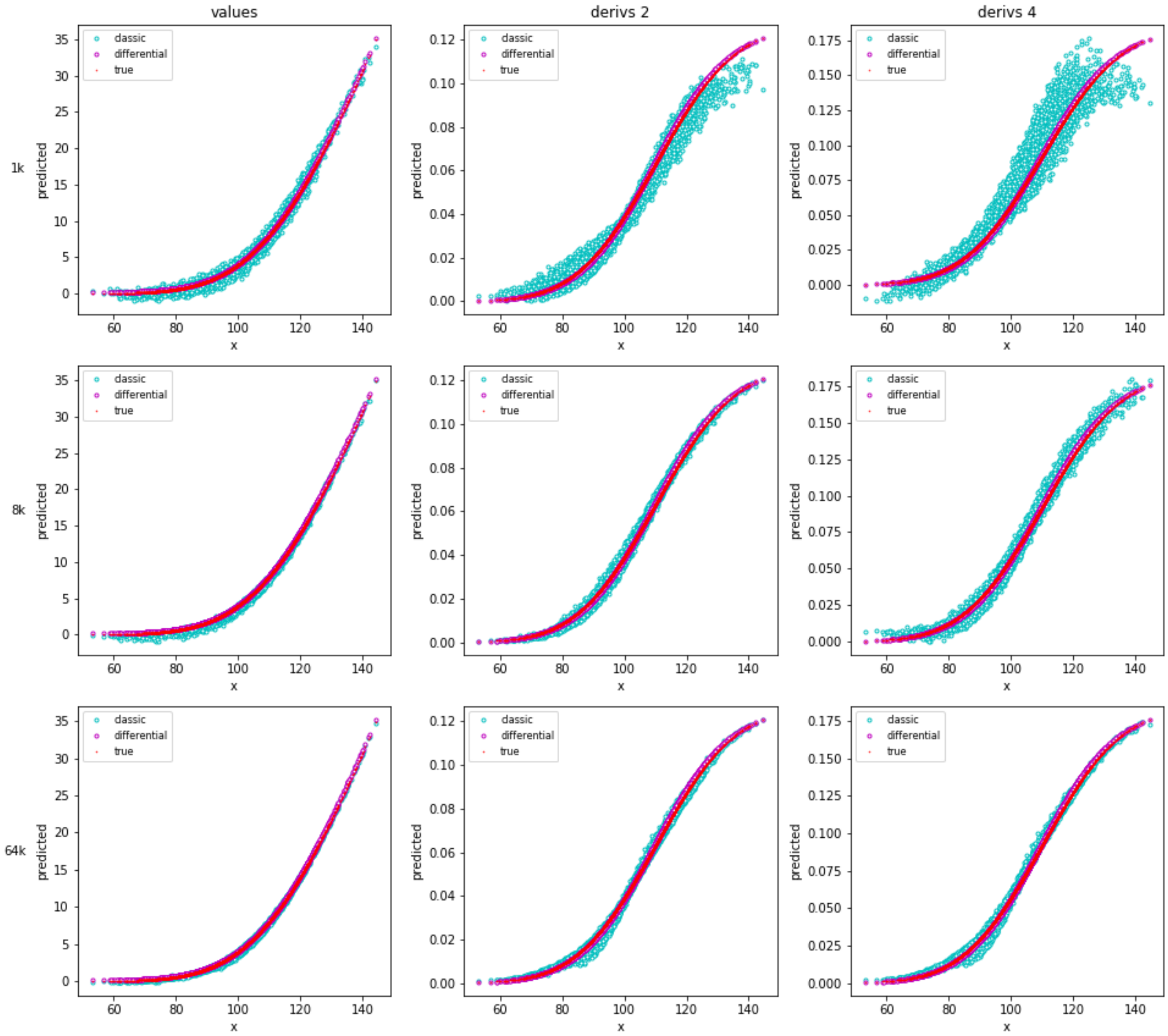
We do not have a closed form solution for reference, so performance is measured against nested Monte-Carlo simulations (a *very* slow process). In Figure 5, we show prediction results for 128 independent examples, with correct numbers on the horizontal axis, as given by the nested simulations, and predicted results on the vertical axis. Performance is measured by distance to the 45deg line.

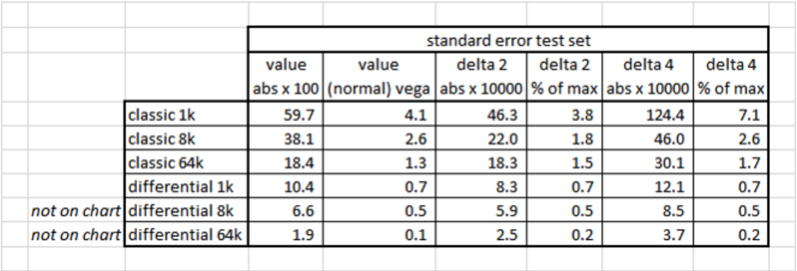
The classical network is trained on 32768 (32k) samples, without derivatives, with cross-validation and early stopping. The twin network is trained on 8192 (8k) samples with pathwise derivatives produced with AAD.

[8] In Bachelier

$$stderr = \sigma \sqrt{\frac{T}{N}} \sqrt{\left(1 - 2dn\left(d\right) + d^2 N\left(-d\right)\right) N\left(d\right) + \left(d - n\left(d\right)\right) n\left(d\right)}$$

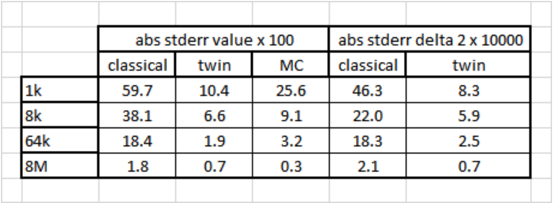
where n and N are the standard normal density and distribution, $d = (S - K) / \left(\sigma\sqrt{T}\right)$ where S is the current basket, 100 in our example, K, T are the strike and maturity, 110 and 1 for us, and σ is the basket volatility, in our case 20 and N is the number of Monte-Carlo simulations. Hence, our Monte-Carlo error is $8.2/\sqrt{N}$.





		standard error test set					
		value abs x 100	value (normal) vega	delta 2 abs x 10000	delta 2 % of max	delta 4 abs x 10000	delta 4 % of max
	classic 1k	59.7	4.1	46.3	3.8	124.4	7.1
	classic 8k	38.1	2.6	22.0	1.8	46.0	2.6
	classic 64k	18.4	1.3	18.3	1.5	30.1	1.7
	differential 1k	10.4	0.7	8.3	0.7	12.1	0.7
not on chart	differential 8k	6.6	0.5	5.9	0.5	8.5	0.5
not on chart	differential 64k	1.9	0.1	2.5	0.2	3.7	0.2

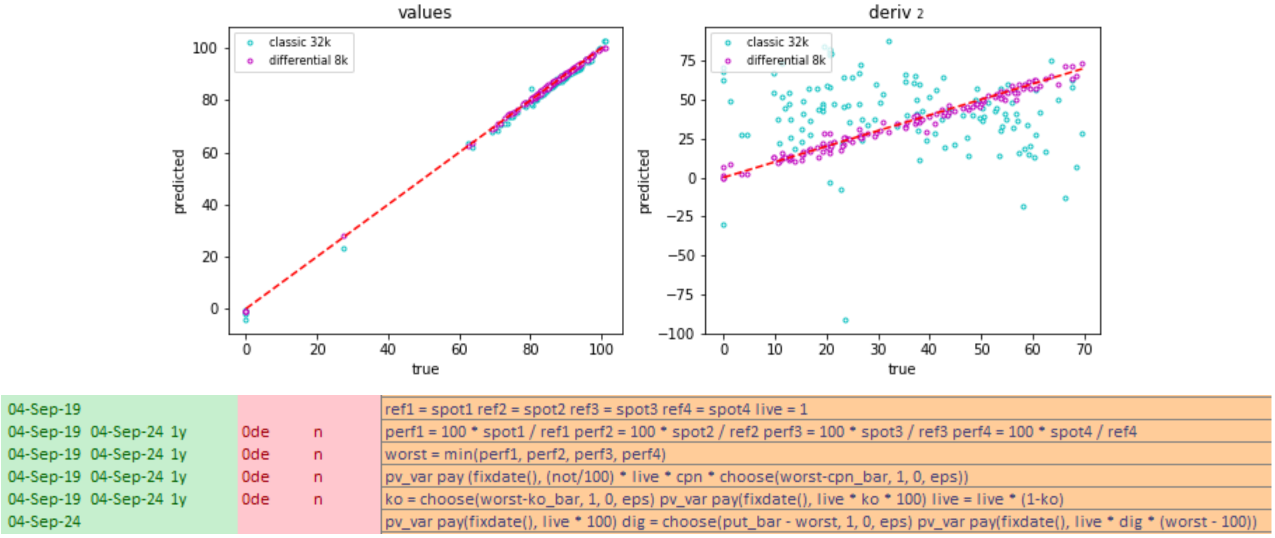
Figure 3: basket option in Bachelier model, dimension 7



	abs stderr value x 100			abs stderr delta 2 x 10000	
	classical	twin	MC	classical	twin
1k	59.7	10.4	25.6	46.3	8.3
8k	38.1	6.6	9.1	22.0	5.9
64k	18.4	1.9	3.2	18.3	2.5
8M	1.8	0.7	0.3	2.1	0.7

Figure 4: approximation error and Monte-Carlo error in Bachelier basket

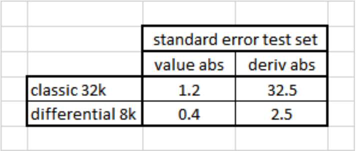
Both sets were generated in around 0.4 sec in Superfly, Danske Bank's proprietary derivatives pricing and risk management system.



04-Sep-19					ref1 = spot1 ref2 = spot2 ref3 = spot3 ref4 = spot4 live = 1
04-Sep-19	04-Sep-24	1y	0de	n	perf1 = 100 * spot1 / ref1 perf2 = 100 * spot2 / ref2 perf3 = 100 * spot3 / ref3 perf4 = 100 * spot4 / ref4
04-Sep-19	04-Sep-24	1y	0de	n	worst = min(perf1, perf2, perf3, perf4)
04-Sep-19	04-Sep-24	1y	0de	n	pv_var pay (fixdate(), (not/100) * live * cpn * choose(worst-cpn_bar, 1, 0, eps))
04-Sep-19	04-Sep-24	1y	0de	n	ko = choose(worst-ko_bar, 1, 0, eps) pv_var pay(fixdate(), live * ko * 100) live = live * (1-ko)
04-Sep-24					pv_var pay(fixdate(), live * 100) dig = choose(put_bar - worst, 1, 0, eps) pv_var pay(fixdate(), live * dig * (worst - 100))

Figure 5: worst-of-four autocallable with correlated local volatility models – twin network trained on 8k samples vs vanilla network trained on 32k samples

Figure 5 shows the results for the value and the delta to the second underlying, together with the script for the instrument, written in Danske Bank's Jive scripting language. Note that the barriers and the digitals are explicitly smoothed with the keyword 'choose', as discussed in the previous section. It is evident that the twin network with only 8k training data produces a virtually perfect approximation in values, and a decent approximation on deltas. The classical network also approximates values correctly, although not on a straight line, which may cause problems for ordering, e.g. for expected loss or FRTB. Its deltas are essentially random, which rules them out for approximation of risk sensitivities, e.g. for SIMM-MVA. The standard errors are given in Figure 6.



	standard error test set	
	value abs	deriv abs
classic 32k	1.2	32.5
differential 8k	0.4	2.5

Figure 6: worst-of-four autocallable – standard errors

For comparison, the Monte-Carlo pricing error is 0.2 with 8k paths, around half the standard error of the twin net trained on 8k samples. Again, the estimation of neural network standard errors is accurate modulo multiplicative factor 2. What matters is that the error of the twin net is of the same order as Monte-Carlo. The error on the classical net, with 4 times the training size, is larger for values and order of magnitude larger for differentials.

3.3 Derivatives trading books

For the last example, we picked a real netting set from Danske Bank's portfolio, including single and cross currency swaps and swaptions in 10 different currencies, eligible for XVA, CCR or other regulated computations. Simulations are performed in Danske Bank's model of everything (the 'Beast'), where interest rates are simulated each with a four-factor version of Andreasen's take on multi-factor Cheyette [2], and correlated between one another and with forex rates.

This is an important example, because it is representative of how we want to apply twin nets in the real world. In addition, this is a stress test for neural networks. The Markov dimension of the four-factor non-Gaussian Cheyette model is 16 per currency, that is 160 inputs, 169 with forexes, and over 1000 with all the path-dependencies in this real-world book. Of course, the value effectively only depends on a small number of combinations of inputs, something the neural net is supposed to identify. In reality, the extraction of effective risk factors is considerably more effective in the presence of differential labels (see our online appendix), which explains the results in Figure 7.

Figure 7 shows the values predicted by a twin network trained on 8192 (8k) samples with AAD pathwise derivatives, compared to a vanilla net, trained on 65536 (64k) samples, all simulated in Danske Bank's XVA system. The difference in performance is evident in the chart, as is virtual perfection of the approximation given by the twin net, trained on only 8k examples. As with the previous example, the predicted values for an independent set of 128 examples are shown on the vertical axis, with correct values on the horizontal axis. The 'correct' values for the chart were produced with nested Monte-Carlo overnight. The entire training process for the twin network (on entry level GPU), including the generation of the 8192 examples (on multithreaded CPU), took a few seconds on a standard workstation.

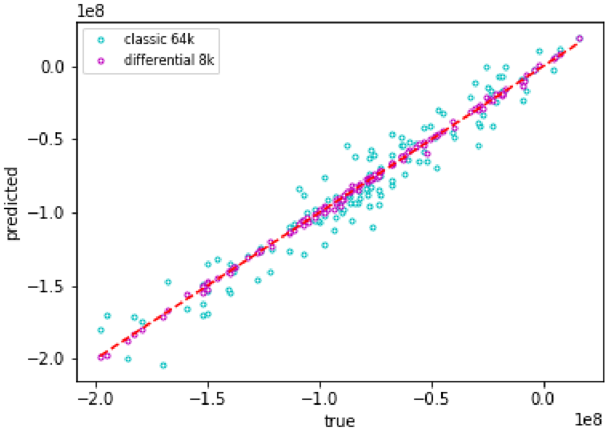


Figure 7: real-world netting set – twin network trained on 8k samples vs classical net trained on 64k samples

We have shown in this figure the predicted values, not derivatives, because we have too many of them, often wrt obscure model parameters like accumulated covariances in Cheyette. For these derivatives to make sense, they must be turned into market risks by application of inverse Jacobian matrices [27], something we skipped in this exercise.

Standard errors are 12.85M with classical 64k and 1.77M with differential 8k, for a range of 200M for the 128 test examples, generated with the calibrated hybrid model. On this example too, twin 8k error is very similar to the Monte-Carlo pricing error (1.70M with 8k paths). Again in this very representative example, the twin network has the same degree of approximation as a classic Monte-Carlo pricer.

4 Extensions

We have presented our algorithm in the context of single value prediction to avoid confusions and heavy notations. In this section, we discuss two advanced extensions, allowing the network to simultaneously predict multiple values, as well as higher order derivatives.

4.1 Multiple outputs

One innovation in [17] is to compute calls of multiple strikes and expiries in a single network, exploiting correlation and shared factors, and encouraging the network to learn global features like non-arbitrage conditions. The idea may be further generalized by making predictions a *parameterization* of the volatility surface, like the parameters of Gatheral's popular SVI structure [10]. Note that the original idea in [17] is indeed a particular case, where the parameters are the values of interpolation knots.

We can combine our approach with this idea by an extension of the twin network to compute multiple predictions, meaning $n_L > 1$ and $y = z_L \in \mathbb{R}^{n_L}$. The adjoints are no longer well defined as vectors. Instead, we now define them as directional differentials wrt some specified linear combination of the outputs yc^T where $c \in \mathbb{R}^{n_L}$ has the coordinates of the desired direction in a row vector:

$$\bar{x} = \frac{\partial yc^T}{\partial x}, \bar{z}_l = \frac{\partial yc^T}{\partial z_l}, \bar{y} = \frac{\partial yc^T}{\partial y} = c$$

Given a direction c, all the previous equations are identically applicable, where the boundary condition for $\bar{y}$ in the backpropagation equations is no longer the number 1, but the row vector c. For example, $c = e_1$ means that adjoints are defined as derivatives wrt the first output y_1. We can repeat this for $c = e_1, \ldots, e_n$ to compute the derivatives of all the outputs wrt all the inputs $\partial y/\partial x \in \mathbb{R}^{n_L \times n}$, i.e the Jacobian matrix. Written in matrix terms, the boundary is the identity matrix $I \in \mathbb{R}^{n_L \times n_L}$ and the backpropagation equations are written as follows:

$$\begin{aligned} \bar{z}_L &= \bar{y} = I \\ \bar{z}_{l-1} &= \left(\bar{z}_l w_l^T\right) \circ g'_{l-1}\left(z_{l-1}\right), \quad l = L, \ldots, 1 \\ \bar{x} &= \bar{z}_0 \end{aligned}$$

where $\bar{z}_l \in \mathbb{R}^{n_L \times n_l}$. In particular, $\bar{x} \in \mathbb{R}^{n_L \times n}$ is (indeed) the Jacobian matrix $\partial y/\partial x$. To compute a full Jacobian, the theoretical order of calculations is n_L times the vanilla network. Notice however, that the implementation of the multiple backpropagation in the matrix form above on a system like TensorFlow automatically benefits from CPU or GPU parallelism. Therefore, the additional computation complexity will be experienced as sublinear.

4.2 Higher order derivatives

The twin network can also predict higher order derivatives. For simplicity, revert to the single prediction case where $n_L = 1$. The twin network predicts $\bar{x}$ as a function of the input x. The network, however, doesn't know anything about derivatives. It just computes numbers by a sequence of equations. Hence, we might as well consider the prediction of derivatives as multiple outputs.

As previously, in what is now considered a multiple prediction network, we can compute the adjoints of the outputs $\bar{x}$ of this network. These are now *the adjoints of the adjoints*:

$$\bar{\bar{x}} \equiv \frac{\partial \bar{x} c^T}{\partial x} \in \mathbb{R}^n$$

in other terms, the Hessian matrix of the value prediction y. Note that the original activation functions must be C^2 for this computation.

When training on call prices it is important to obey arbitrage conditions as noted in [20]. With this technique we are able to train directly on the arbitrage conditions $\partial call/\partial T > 0, \partial call/\partial K < 0, \partial^2 call/\partial K^2 > 0$ coming from the model where T, K are expiry and strike respectively. We refer to [18] for further details and applications of higher order derivatives. The computation load of a full Hessian is of order n times the original network. These additional calculations generate a lot more data, one value, n derivatives and $n\,(n+1)$ second order derivatives for the cost of $2n$ times the value prediction alone. In a parallel system like TensorFlow, it will be also be experienced as sublinear.

We can extend this argument to arbitrary order q, with the only restriction that the (original) activation functions are C^q. This is the exact same restriction as in the Universal approximation theorem including derivatives (see [16]), which states that a a single layer feedforward network *with C^q activation* can approximate f *and all of its derivatives up to order q* asymptotically, with arbitrary accuracy. The twin network may be seen as a proof. To see this, start with $q = 1$. By the Universal approximation theorem (without derivatives, see [15]), both the classical network (1) and the twin network defined by (3) and (4) converge to the same function f. We observe from the twin network that also the derivatives of f converge hence this is also true for the simple feedforward network. We can construct networks matching derivatives of any order, which completes the proof.

Conclusion

Throughout our analysis we have seen that 'learning the correct shape' from differentials is crucial to the performance of regression models, including neural networks, in such complex computational tasks as the pricing and risk approximation of complex Derivatives trading books. The *unreasonable effectiveness* of what we called 'differential machine learning' permits to accurately train ML models on a small number of simulated examples, *a la* Longstaff-Schwartz, in realistic time. This makes the twin networks applicable to real-world problems, including regulations and risk reports with multiple scenarios, the empirical test error of twin networks being of comparable magnitude to Monte-Carlo pricing error.

The methodology learns from data alone and is applicable in very general contexts, with arbitrary schedules of cash-flows, scripted or not, and arbitrary simulation models. The algorithm also applies to many families of approximations, including classic linear combinations of fixed basis functions, and neural networks of arbitrary complex architecture. It consumes differentials of labels wrt inputs. In finance, the differentials are obtained with AAD, in the same way we generate Monte-Carlo risk reports, with analytic accuracy and very little computation cost.

One of the main benefits of twin networks is their ability to learn effectively from small datasets. Differentials inject meaningful additional information, eventually resulting in *better* results with small datasets of 1k to 8k examples than can be obtained otherwise with training sets orders of magnitude larger. Learning effectively from small datasets is critical in the context of regulations, where the pricing approximation must be learned quickly, and the expense of a large training set cannot be afforded.

The penalty enforced for wrong differentials in the cost function also acts as a very effective regularizer, superior to classical forms of regularization like Ridge, Lasso or Dropout, which enforce arbitrary penalties to mitigate overfitting, whereas differentials meaningfully augment data. Standard regularizers are very sensitive to the regularization strength λ, a manually tweaked hyperparameter. Differential training is virtually insensitive to λ, because, even with infinite regularization, we train on derivatives alone and still converge to the correct approximation, modulo an additive constant.

We posted on GitHub two further applications of differential ML: differential regression and differential PCA. In the context of classic regression, differentials act as a very effective regularizer. Like Tikhonov regularization, differential regularization is analytic and works SVD. Unlike Tikhonov, differential regularization is *unbiased.* Differential PCA, unlike classic PCA, is able to extract from data the principal risk factors of a given transaction, and it can be applied as a preprocessing step to safely reduce dimension.

Differential training also appears to stabilize the training of neural networks, and improved resilience to hyperparameters like network architecture, seeding of weights or learning rate schedule was consistently observed, although to explain exactly why is a topic for further research.

Standard machine learning may often be considerably improved with contextual information not contained in data, such as the nature of the relevant features from knowledge of the transaction and the simulation model. For example, we know that the continuation value of a Bermudan option on call date mainly depends on the swap rate to maturity and the discount rate to the next call. We can learn pricing functions much more effectively with hand engineered features. But it has to be done manually, on a case by case basis, depending on the transaction and the simulation model. Differential machine learning learns *better* from data alone, the vast amount of information contained in the differentials playing a role similar, and often more effective, to manual adjustments from contextual information.

Differential machine learning is similar to data augmentation in computer vision, a technique consistently applied in that field with documented success, where multiple labeled images are produced from a single one, by cropping, zooming, rotation or recoloring. In addition to extending the training set for a negligible cost, data augmentation encourages the ML model to learn important invariances. Similarly, derivatives labels, not only increase the amount of information in the training set, but also encourage the model to learn the *shape* of the pricing function.

References

[1] M. S. Alexandre Antonov, Michael Konikov. Mixing sabr models for negative rates. *Risk*, 2015.

[2] J. Andreasen. Back to the future. *Risk*, 2005.

[3] D. Bang. Local stochastic volatility: shaken, not stirred. *Risk*, 2018.

[4] B.Huge and A. Savine. Deep analytics: Risk management with ai. Global Derivatives, 2019, also available on deep-analytics.org.

[5] A. Brace, D. Gatarek, and M. Musiela. The market model of interest rate dynamics. *Mathematical Finance*, 7(2):127–154, 1997.

[6] J. F. Carriere. Valuation of the early-exercise price for options using simulations and nonparametric regression. *Insurance: Mathematics and Economics*, 19(1):19–30, 1996.

[7] Q. Chan-Wai-Nam, J. Mikael, and X. Warin. Machine Learning for semi linear PDEs. *arXiv e-prints*, page arXiv:1809.07609, Sept. 2018.

[8] B. Dupire. Functional it calculus. *Quantitative Finance*, 19(5):721–729, January 2019.

[9] J. Gatheral. Rough volatility: An overview. Global Derivatives, 2017.

[10] J. Gatheral and A. Jacquier. Arbitrage-free svi volatility surfaces. *Quantitative Finance*, 14(1):59–71, 2014.

[11] J. Gatheral, P. Jusselin, and M. Rosenbaum. The quadratic rough heston model and the joint s&p 500/vix smile calibration problem. *Risk*, May 2020.

[12] M. Giles and P. Glasserman. Smoking adjoints: Fast evaluation of greeks in monte carlo calculations. *Risk*, 2006.

[13] P. S. Hagan, D. Kumar, A. S. Lesniewski, and D. E. Woodward. Managing smile risk. *Wilmott Magazine*, 1:84–108, 2002.

[14] M. B. Haugh and L. Kogan. Pricing american options: A duality approach. *Operations Research*, 52(2):258–270, 2004.

[15] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[16] K. Hornik, M. Stinchcombe, and H. White. Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. *Neural Networks*, 3(5):551–560, 1990.

[17] B. Horvath, A. Muguruza, and M. Tomas. Deep learning volatility, 2019.

[18] B. Huge. Going in the right direction. Danske Bank working paper available from www.researchgate.net, 2018.

[19] J. M. Hutchinson, A. W. Lo, and T. Poggio. A nonparametric approach to pricing and hedging derivative securities via learning networks. *The Journal of Finance*, 49(3):851–889, 1994.

[20] A. Itkin. Deep learning calibration of option pricing models: some pitfalls and solutions. *Risk*, April 2020.

[21] B. Lapeyre and J. Lelong. Neural network regression for bermudan option pricing, 2019.

[22] F. A. Longstaff and E. S. Schwartz. Valuing american options by simulation: A simple least-square approach. *The Review of Financial Studies*, 14(1):113–147, 2001.

[23] W. A. McGhee. An artificial neural network representation of the sabr stochastic volatility model, 2018.

[24] A. Reghai, O. Kettani, and M. Messaoud. Cva with greeks and aad. *Risk*, 2015.

[25] A. Savine. Aad and backpropagation in machine learning and finance, explained in 15min. https://www.youtube.com/watch?v=IcQkwgPwfm4.

[26] A. Savine. Stabilize risks of discontinuous payoffs with fuzzy logic. Global Derivatives, 2015.

[27] A. Savine. From model to market risks: The implicit function theorem (ift) demystified. *SSRN preprint*, 2018. Available at SSRN: https://ssrn.com/abstract=3262571 or http://dx.doi.org/10.2139/ssrn.3262571.

[28] A. Savine. *Modern Computational Finance: AAD and Parallel Simulations*. Wiley, 2018.