Fast Calibration using Complex-Step Sobolev Training

by Bouazza SAADEDDINE*[†][‡]

bouazza.saadeddine2@ca-cib.com

Abstract

We present a new fast calibration technique where we propose to train neural networks to directly perform the orthogonal projection of simulated payoffs of the calibration instrument with randomized model parameters and we enrich the learning task by including path-wise sensitivities of the payoffs with respect to model and product parameters. We show that this particular instance of Sobolev training can be reformulated in a way that requires computing only (stochastic) directional derivatives and we provide a fast, memory-efficient and numerically stable approach to compute those using complex-step differentiation. Our experiment with a fixed-grid piecewise linear local volatility example demonstrates that one can get competitive price approximations without having to train the neural network on Monte Carlo prices and that both data-set construction and training can be done in reasonable time while preserving a very general framework that can be applied to a broad range of pricing models. We provide a highly optimized `C++` code based on `libtorch` which includes all the necessary extensions for AAD on holomorphic neural networks on: `https://github.com/BouazzaSE/TorchCSD`.

Keywords: complex-step differentiation, fast pricing, model calibration, neural network, sobolev training

A.M.S. subject classification: 91G20, 91G60, 91G80, 62M45

1 Introduction

With the emergence of pricing models such as rough volatility models [5] which do not have closed-form solutions for vanilla option prices and are slow to simulate using Monte Carlo methods, the need to accelerate the calibration of these models arose as one would otherwise have to repeatedly call a slow Monte Carlo pricing procedure during the calibration phase [23], which is often implemented using iterative optimization algorithms, rendering the models challenging to implement in practice. Notable recent contributions in this area involve the use of Machine Learning methods to provide fast approximations for the pricing function, and then using the *learned* approximation instead of a Monte Carlo pricer during the calibration phase, effectively accelerating the model calibration as the learned approximation is usually fast to compute. In [12] for instance, the authors propose to approximate the mapping from model and product parameters to vanilla prices using Gaussian Process regression [28]. Neural network based price approximations have also been proposed [6, 16], mainly motivated by the *Universal Approximation Theorem* [10, 18]. Another approach [15] consists in directly approximating the *inverse* function which maps prices to model parameters, effectively skipping the calibration phase altogether.

*. Laboratoire de Probabilités, Statistique et Modélisation (LPSM), Sorbonne Université et Université de Paris, CNRS UMR 8001.

†. LaMME, Université d'Evry, CNRS, Université Paris-Saclay, 91037, Evry, France.

‡. Quantitative Research GMD/GMT Crédit Agricole CIB, 92160 Montrouge.

Acknowledgments. The author's PhD research was funded by a CIFRE grant by Association Nationale de la Recherche et de la Technologie (ANRT) and Crédit Agricole CIB and was directed by Stephane CREPEY and co-supervised by Lokmane ABBAS-TURKI. The author is grateful for insightful comments and remarks by Lokmane ABBAS-TURKI, Stephane CREPEY, Pierre FICHTER, Christophe MICHEL, Vincent PORTE and Crédit Agricole GMD Quantitative Research teams.

All of these approaches suffer from the need to construct data-sets of sufficiently accurate Monte Carlo prices which can take days depending on the complexity of the pricing model. While this can be done *off-line* for general and fixed classes of pricing models and thus the time spent can be considered as a one-off upfront cost, it cannot be neglected when having to frequently deal with very custom pricing models where we may have to frequently reconstruct the pricing approximation from scratch. A slow data-set construction process also severely limits the ability to iterate effectively in research and development as one cannot afford the luxury to test out different time discretization schemes, time step sizes, variance reduction techniques or random number generators.

Different from these approaches, we propose a new fast calibration method which does not require generating data-sets of Monte Carlo prices and instead needs only realizations of payoffs corresponding to random model parameters and their path-wise sensitivities. We dub the proposed training procedure *Complex-Step Sobolev training*, which we recognize could be applied to more general problems outside of pricing model calibration and quantitative finance. We highlight the main contributions of this paper as follows:

- We propose to learn a fast vanilla pricing function using a special instance of Sobolev training by learning to orthogonally project both payoffs and path-wise sensitivities of payoffs corresponding to randomized model parameters;

- We propose a method to accelerate our Sobolev training procedure using only directional derivatives in random directions. We also show and prove how to optimally choose the distribution of this random direction such that the induced variance is minimized;

- We accelerate further the computation of the directional derivatives in an AAD-differentiable way using complex-step differentiation while attaining machine precision and preserving numerical stability;

- We give a posteriori L^2 error estimates that can be computed without having access to ground-truth prices;

- Benchmarks and a fixed-grid local volatility example demonstrating the strength of our method are provided. All the simulation codes and the necessary extensions to implement complex-step differentiation in an AAD-differentiable manner with `libtorch` have been made public on `https://github.com/BouazzaSE/TorchCSD` under a GPLv3 license.

2 Learning to Project Payoffs

Consider a stochastic risk-neutral pricing basis $(\Omega, \mathcal{A}, (\mathcal{F}_t)_{t \geq 0}, \mathbb{Q})$ and an $\mathbb{R}^d$-valued standard Brownian motion $(B_t)_{t \geq 0}$ for some $d \geq 1$. Let $f: \mathbb{R}_+ \times \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^d$ and $g: \mathbb{R}_+ \times \mathbb{R}^d \times \mathbb{R}^n \to \mathbb{R}^{d \times m}$ be two piece-wise continuously differentiable functions such that we have for all $t \geq 0$, $x, y \in \mathbb{R}^d$ and $\xi, \xi' \in \mathbb{R}^n$:

$$\begin{aligned}
\|f(t,x,\xi) - f(t,y,\xi)\| + \|g(t,x,\xi) - g(t,y,\xi)\| &\leq K\,\|x-y\| \\
\|f(t,x,\xi) - f(t,x,\xi')\| + \|g(t,x,\xi) - g(t,x,\xi')\| &\leq K\,(1+\|x\|)\,\|\xi-\xi'\| \\
\|f(t,x,\xi)\| + \|g(t,x,\xi)\| &\leq K\,(1+\|x\|)
\end{aligned}$$

for some $K>0$, where the norms are the usual Euclidean and Frobenius norms for vectors and matrices respectively. For every $\xi\in\mathbb{R}^n$, we define $(X_t^\xi)_{t\geq 0}$ to be a strong solution to the following multi-dimensional stochastic differential equation:

$$\mathrm{d}X_t^\xi = f(t, X_t, \xi)\,\mathrm{d}t + g(t, X_t, \xi)\,\mathrm{d}B_t \tag{1}$$

We assume the same deterministic initial value X_0 for all $\xi\in\mathbb{R}^n$. In a pricing context, the vector ξ represents model parameters.

Example 1. (Fixed-grid local volatility) Consider a local volatility model described by the following SDE:

$$\forall t>0, \mathrm{d}S_t = r\,S_t\,\mathrm{d}t + \sigma(t, \log(S_t))\,S_t\,\mathrm{d}B_t$$

where $r\in\mathbb{R}$, B is a standard Brownian motion and for all $t\in[t^{(i)}, t^{(i+1)}]$ and $s\in[s^{(j)}, s^{(j+1)}]$:

$$\begin{aligned}\sigma(t,s) &= \sigma_{i,j} + \frac{s-s^{(j)}}{s^{(j+1)}-s^{(j)}}(\sigma_{i,j+1}-\sigma_{i,j})\\ &\quad+\frac{t-t^{(i)}}{t^{(i+1)}-t^{(i)}}\left(\sigma_{i+1,j}-\sigma_{i,j}+\frac{s-s^{(j)}}{s^{(j+1)}-s^{(j)}}(\sigma_{i+1,j+1}-\sigma_{i,j+1}-\sigma_{i+1,j}+\sigma_{i,j})\right)\end{aligned}$$

and for all $t>0$ and $s\in\mathbb{R}$, $\sigma(t,s)=\sigma(\tau(t),\chi(s))$, where $\tau(t)=\begin{cases} t & \text{if } \exists k\colon t\in[t^{(k)}, t^{(k+1)}]\\ t^{(m)} & \text{if } \quad t\geq t^{(m)}\\ t^{(0)} & \text{if } \quad t<t^{(0)}\end{cases}$ and $\chi(s)=\begin{cases} s & \text{if } \exists k\colon s\in[s_k, s_{k+1}]\\ s^{(M)} & \text{if } \quad s\geq s^{(M)}\\ s^{(0)} & \text{if } \quad s<s^{(0)}\end{cases}$, and $0<t^{(0)}<\cdots<t^{(m)}$, $s^{(0)}<\cdots<s^{(M)}$ and $\sigma_{i,j}>0$ for all $i\in\{1,\ldots,m\}$ and $j\in\{1,\ldots,M\}$. For consistency of the interpolation formula above at the extremes, $t^{(m+1)}$ and $s^{(M+1)}$ will be taken to be any values respectively different from $t^{(m)}$ and $s^{(M)}$, their values having no impact on the value of $\sigma(t,s)$. We can again recast this model into the general form with the following mapping, with σ by construction also a function of ξ:

$$\begin{cases}\xi &:= (r, \sigma_{0,0}, \ldots, \sigma_{i,j}, \ldots \sigma_{m,M})^\top\\ X_t^\xi &:= S_t\\ f(t, X_t^\xi; \xi) &:= r\,S_t\end{cases}, \quad g(t, X_t^\xi; \xi) := \sigma(t, \log(X_t^\xi))\,X_t^\xi$$

□

We assume in what follows that the calibration instruments are vanilla European calls. Let Ξ, K, T be $\mathcal{F}_0$-measurable random variables supported on respectively $\mathbb{R}^n$, $\mathbb{R}_+$ and $\mathbb{R}_+^\star$ such that K and T are mutually independent and independent of Ξ. We are interested in pricing, conditional on random model parameters Ξ, random strike K and random maturity T, a product paying $Z^\Xi = (\psi(X_T^\Xi) - K)^+$ where $\psi\colon\mathbb{R}^d\to\mathbb{R}$ is piece-wise continuously differentiable and Lipschitz continuous. If we consider for simplicity[1] a numéraire $(e^{rt})_{t\geq 0}$ with r an $\mathcal{F}_0$-measurable risk-free rate which we will assume to be the first component of Ξ, then the price is given by $\mathbb{E}[e^{-rT}Z^\Xi|\Xi, K, T]$ and satisfies:

$$\mathbb{E}[e^{-rT}Z^\Xi|\Xi, K, T] = \varphi^\star(\Xi, K, T)$$

1. This does not limit the generality of the method in any way. Numéraires with stochastic interest rates can also be used without any issues.

with

$$\varphi^\star \in \underset{\varphi \in \mathcal{B}}{\operatorname{argmin}} \mathbb{E}[(\varphi(\Xi, K, T) - e^{-rT} Z^\Xi)^2] \tag{2}$$

where $\mathcal{B}$ is the space of Borel functions $\varphi\colon \mathbb{R}^{n+2} \to \mathbb{R}$ such that $\varphi(\Xi, K, T)$ is square integrable. Here we exploited the characterization of a conditional expectation as an L^2 projection. Indeed, we have for any such φ:

$$\begin{aligned}\mathbb{E}[(\varphi(\Xi, K, T) - e^{-rT} Z^\Xi)^2] &= \mathbb{E}[(\varphi(\Xi, K, T) - \mathbb{E}[e^{-rT} Z^\Xi | \Xi, K, T])^2] \\ &\quad + \underbrace{\mathbb{E}[\operatorname{var}(e^{-rT} Z^\Xi | \Xi, K, T)]}_{\text{independent of } \varphi}\end{aligned}$$

We propose to restrict the search space in (2) to the space of neural networks with a given architecture.

Definition 1. *Let α be a non-affine continuously differentiable activation function, applied element-wise, and let $m, p, q \in \mathbb{N}^\star$. Define $\mathcal{NN}_{m,p,q,\alpha}$ to be the set of functions $\mathbb{R}^{n+2} \ni x \mapsto \phi_{p+1}(x; W, b)$ such that $W_1 \in \mathbb{R}^{m \times (n+2)}$, $W_2, \ldots, W_p \in \mathbb{R}^{m \times m}$, $W_{p+1} \in \mathbb{R}^{q \times m}$, $b_1 \ldots, b_p \in \mathbb{R}^m$, $b_{p+1} \in \mathbb{R}^q$ and:*

$$\begin{aligned}\phi_0(x; W, b) &= x \\ \phi_i(x; W, b) &= \alpha(W_i \phi_{i-1}(x; W, b) + b_i), \forall i \in \{1, \ldots, p\} \\ \phi_{p+1}(x; W, b) &= W_{p+1} \phi_p(x; W, b) + b_{p+1}\end{aligned} \tag{3}$$

$\mathcal{NN}_{m,p,q,\alpha}$ is then called the set of neural networks with p hidden layers, m neurons per hidden layer, $n+2$ inputs and q outputs, and α as its activation function.

Let $\mathcal{N}$ be such a set with $q = 1$ (*i.e.* only one output neuron). An approach to approximate the pricing function $\varphi^\star$ would then be to find $\tilde{\varphi}$ such that:

$$\tilde{\varphi} \in \underset{\varphi \in \mathcal{N}}{\operatorname{argmin}} \mathbb{E}[(\varphi(\Xi, K, T) - e^{-rT} Z^\Xi)^2]$$

where the optimization becomes parametric as the neural networks in $\mathcal{N}$ are parameterized by their weights and biases, and the optimization can be done using vanilla stochastic gradient descent (SGD) or more elaborate accelerated SGD optimizers like Adam [19]. The approximation of $\mathbb{E}[e^{-rT} Z^\Xi | \Xi, K, T]$ would then be $\tilde{\varphi}(\Xi, K, T)$.

The restriction to neural networks is mainly motivated by the following density result:

Theorem 1. (Universal Approximation Theorem for Deep Narrow Networks[18]) *Let α be a non-affine continuously differentiable activation function, $q \in \mathbb{N}^\star$ and let $K \subset \mathbb{R}^{n+2}$ be compact. Then $\bigcup_{p \in \mathbb{N}^\star} \mathcal{NN}_{q+n+4,p,q,\alpha}$ is dense in $\mathcal{C}(K, \mathbb{R}^q)$ with respect to the topology of uniform convergence.*

3 Regularizing with Sobolev Training

The learning task however will suffer from increased variance compared to price interpolation approaches and stochastic gradient descent will yield noisy updates because of gradient estimators having high variances. The high variance can be overcome by producing a large enough data-set and using large batch sizes during the SGD iterations. In addition to the potential variance problem, regularization may be needed when considering large neural networks to avoid over-fitting payoffs. This is usually [21] done by imposing generic restrictions on the neural network weights such as L^1 or L^2 regularization.

In [17], the authors propose to tackle both issues by instead adding a *regularizing* term which penalizes the error when projecting the path-wise derivatives of the payoff, which will have the effect of both artificially increasing the size of the data-set (while profiting from the common computations for the path-wise derivatives) and regularizing the learning procedure by adding constraints on the behavior of the network locally around sample points. Although the authors focused on randomizing only the initial value of the SDE and not its parameters, when recast in our calibration setting it leads to having to solve the following learning problem:

$$\tilde{\tilde{\varphi}} \in \underset{\varphi \in \mathcal{N}}{\operatorname{argmin}} \mathbb{E}[(\varphi(\Xi, K, T) - e^{-rT} Z^{\Xi})^2] + \sum_{k=1}^{n+2} \lambda_k \, \mathbb{E}[(\partial_k \varphi(\Xi, K, T) - \partial_k(e^{-rT} Z^{\Xi}))^2] \tag{4}$$

where $\lambda \in (\mathbb{R}_+^\star)^{n+2}$ and ∂_k is the partial derivative with respect to the k-th component of the concatenation of Ξ and (K, T). More precisely:

Definition 2. (Point-wise gradient of a parameterized random variable) *If $Y(\xi)$ is a real valued random variable that is parameterized by $\xi \in \mathbb{R}^n$ then we define the point-wise gradient $\nabla Y(\xi)$ of $Y(\xi)$ with respect to ξ as $\nabla Y(\xi) = (\partial_1 Y(\xi), \ldots, \partial_n Y(\xi))^\top$ where for every $i \in \{1, \ldots, n\}$ we have:*

$$\partial_i Y(\xi) := \lim_{\varepsilon \to 0} \frac{1}{\varepsilon} (Y(\xi + \varepsilon\, e_i) - Y(\xi))$$

with $(e_1, \ldots, e_n)$ being the canonical basis in $\mathbb{R}^n$, where the limit is taken point-wise over Ω.

We give below sufficient assumptions in our randomized pricing framework so that these point-wise (or *path-wise* in our case) derivatives exist and are unbiased estimators of the respective derivatives of the price. We refer the reader to [9, 13] for a more general and comprehensive treatment of path-wise sensitivities.

Assumption 1. $\lim_{\varepsilon \to 0} \frac{1}{\varepsilon} (X_T^{\xi + \varepsilon e_i} - X_T^{\xi})$ *exists with probability 1 for all $i \in \{1, \ldots, n\}$ and $\xi \in \mathbb{R}^n$.*

Assumption 2. $\mathbb{Q}(\psi(X_T^{\xi}) = K) = 0$ *for all $\xi \in \mathbb{R}^n$.*

Assumption 3. *There exists $\kappa > 0$ such that for all $\xi_1, \xi_2 \in \mathbb{R}^n$ we have*

$$\mathbb{E}[\|X_T^{\xi_1} - X_T^{\xi_2}\|] \le \kappa \, \|\xi_1 - \xi_2\|$$

Assumptions 1 and 2 in particular imply that the point-wise gradient of the call payoff with respect to model parameters exists with probability 1 and justify why it is enough to assume piece-wise differentiability. Assumption 3[2] along with the Lipschitz regularity of the positive part and of ψ implies that the call payoff is Lipschitz continuous with respect to the model parameters. All these assumptions together with the Vitali convergence theorem ensure the existence of the point-wise gradients with probability 1 and that we can interchange gradients and expectations, *i.e.*

$$\nabla \mathbb{E}[e^{-rT} Z^{\Xi} | \Xi, K, T] = \mathbb{E}[\nabla(e^{-rT} Z^{\Xi}) | \Xi, K, T] \tag{5}$$

2. We have chosen to express Assumption 3 such that we can use the Vitali convergence theorem, this is in contrast with the choice in [9, 13] to write the assumption in an almost-sure manner which allows the use of the dominated convergence theorem but makes it hard to verify it in very general pricing models. Assumption 3 can easily be verified by first bounding $\mathbb{E}[\|X_T^{\xi_1} - X_T^{\xi_2}\|^2]$ using Itô isometry and Grönwall's lemma.

The extension to the differentiation with respect to the strike and the maturity is trivial. Note that in discrete-time the payoff will be by construction not differentiable with respect to the maturity. However, one can circumvent this issue by deducing the partial derivative of the price with respect to the maturity using spatial derivatives thanks to the Fokker-Planck equation (see Appendix A).

Hence, the learning problem statement in (4) explicitly seeks to orthogonally project not only payoffs but also their path-wise sensitivities, motivated by (5), which is consistent with our calibration context as a desired feature in the price approximation is access to accurate gradients so that one can use gradient-based optimizers during the calibration phase.

Example 2 shows how path-wise derivatives can be computed in a time-discretized[3] version of our model in Example 1; one can notice in particular the amount of reusable common sub-expressions which make path-wise derivative calculations cheaper than simulating new trajectories of payoffs.

Example 2. (Path-wise sensitivities in a fixed-grid local volatility model) Assume an Euler-Maruyama scheme for the log-dynamics of the model described in Example 1:

$$\hat{s}_{i+1} = \hat{s}_i + \left(r - \frac{1}{2}\sigma(t_i, \hat{s}_i)^2\right) h + \sigma(t_i, \hat{s}_i)\sqrt{h}\, G_{i+1} \quad , \forall i \in \{0, \ldots, I-1\}$$

for some time-grid $0 = t_0 < \ldots < t_i = i\,h < \ldots < t_I = T$ with constant step size $h > 0$, a sequence of independent standard Gaussian variables $(G_i)_{i \geq 1}$, and random $\mathcal{F}_0$-measurable local volatility nodes $(\sigma_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq M}}$ and random $\mathcal{F}_0$-measurable risk-free rate r. The discrete process $(\hat{s}_i)_{0 \leq i \leq I}$ is then a time-discretized approximation of $(\log(S_t))_{t \geq 0}$ at the discrete time steps $t_0, \ldots, t_I$. Let $i \in \{0, \ldots, I-1\}$. By differentiating both the LHS and RHS in the discretized equation above and setting $\alpha_{i+1} = \sqrt{h}\,(G_{i+1} - \sigma(t_i, \hat{s}_i)\sqrt{h})\,\partial_s \sigma(t_i, \hat{s}_i)$, one can write[4]:

$$\begin{aligned} \partial_r \hat{s}_{i+1} &= \partial_r \hat{s}_i + h + \alpha_{i+1}\, \partial_r \hat{s}_i \\ \partial_{\sigma_{u,v}} \hat{s}_{i+1} &= \partial_{\sigma_{u,v}} \hat{s}_i + \alpha_{i+1}\, \partial_{\sigma_{u,v}} \hat{s}_i \end{aligned}$$

for all $1 \leq u \leq m$ and $1 \leq v \leq M$. It then remains to evaluate $\partial_s \sigma(t_i, \hat{s}_i)$, for which we defer the calculations to Appendix B.

□

This class of learning problems, where one is given not only values of the function to be approximated (or, in our case, the random variable to be projected) but also partial derivatives was, to our knowledge, first formulated by [11] who then coined the name *Sobolev training* and has been ubiquitous in Machine Learning research involving physical phenomena [32, 35], where the learning is augmented by either values of the partial derivatives, or relationships between partial derivatives using domain-specific PDEs.

3. The assumptions 1, 2 and 3 also apply to the discrete case and the notion of path-wise differentiation and the unbiasedness (*i.e.* Equation (5)) of the path-wise derivatives readily extend to time-discretized models.

4. This approach to computing derivatives by iterating through time in a forward way is more attractive when considering an implementation on GPUs. Indeed, at any time-step, all the calculations in Example 2 are done locally and registers and GPU caching can be used efficiently. This is in opposition to the traditional backward AAD where one would have to write to the global memory at each time-step and then later read from those same memory locations again which severely hurts performance in GPU implementations.

4 Complex-step Sobolev Training

4.1 Restricting to Stochastic Directional Derivatives

Evaluating the partial derivatives of the neural network in (4) can be a time-consuming process depending on how the computation is done. We give two classical approaches with which those derivatives can be computed exactly. The calculations outlined here form the backbone of modern *algorithmic differentiation* and we refer the reader to [4] for a more comprehensive survey and more technical discussions and to [31] for a reference in the context of computing sensitivities of the price of financial derivatives.

Assume $\mathcal{N} = \mathcal{NN}_{m,p,1,\alpha}$ for some $m, p \in \mathbb{N}^\star$ and a non-affine twice continuously differentiable activation function α and let $\varphi \in \mathcal{N}$, *i.e.* there exist $W_1 \in \mathbb{R}^{m\times(n+2)}$, $W_2, \ldots, W_p \in \mathbb{R}^{m\times m}$, $W_{p+1} \in \mathbb{R}^{1\times m}$, $b_1 \ldots, b_p \in \mathbb{R}^m$, $b_{p+1} \in \mathbb{R}$ such that $\varphi(x) = \phi_{p+1}(x; W, b)$ for all $x \in \mathbb{R}^{n+2}$. Denoting by $J_\cdot$ and ∇ the jacobian matrix and gradient with respect to x and by α' the derivative of the activation function α (applied element-wise), we have:

$$\begin{array}{lll} J_{\phi_0}(x;W,b) & = & I_{n+2} \\ J_{\phi_i}(x;W,b) & = & \operatorname{diag}(\alpha'(W_i\,\phi_{i-1}(x;W,b)+b_i))\,W_i\,J_{\phi_{i-1}}(x;W,b), \forall i \in \{1,\ldots,p\} \\ \nabla\phi_{p+1}(x;W,b) & = & J_{\phi_p}(x;W,b)^\top W_{p+1}^\top \end{array} \tag{6}$$

This naturally defines the so-called *forward* approach to compute the gradient $\nabla\phi_{p+1}(x;W,b)$. Indeed one starts with $J_{\phi_0}(x;W,b)$ and recursively computes $J_{\phi_i}(x;W,b)$ given $J_{\phi_{i-1}}(x;W,b)$ for all $i \in \{1,\ldots,p\}$ to deduce $\nabla\phi_{p+1}(x;W,b)$ at the end, effectively performing the product of jacobian matrices from left to right.

Another approach would be to introduce $\delta_0, \ldots, \delta_p$ as follows:

$$\begin{array}{lll} \delta_0(x;W,b) & = & W_{p+1} \\ \delta_i(x;W,b) & = & \delta_{i-1}(x;W,b)\operatorname{diag}(\alpha'(W_{p-i+1}\,\phi_{p-i}(x;W,b)+b_{p-i+1}))\,W_{p-i+1}, \\ & & \hfill \forall i \in \{1,\ldots,p\} \\ \nabla\phi_{p+1}(x;W,b) & = & \delta_p(x;W,b)^\top \end{array} \tag{7}$$

This defines a *backward*, or *reverse*, approach to compute the gradient $\nabla\phi_{p+1}(x;W,b)$, since we are in effect doing a product of jacobian matrices from right to left.

The major difference between both approaches lies in the fact that the backward approach consists, at least in theory, of less arithmetic operations for the same result as the forward approach. Indeed, by counting in terms of m, n and p the additions and multiplications involved in the matrix operations[5] and the evaluations of the derivative of the activation function and discounting the evaluations of pre-activations[6], one can show that the forward approach performs $\Theta(m^2\,n\,p)$ operations[7] while the backward approach performs only $\Theta(m^2\,p)$ operations. Indeed, during the forward approach, one iteratively computes products of full jacobian matrices and more precisely matrix-matrix products, while in the backward approach one computes only vector-matrix products given that the neural network has only one output neuron.

5. One should pay attention to the fact that the multiplication of an arbitrary square matrix A with a diagonal matrix B on the left (resp. on the right), *i.e.* $A\,B$ (resp. $B\,A$), should be implemented by multiplying the i-th column (resp. row) of A by the i-th entry on the diagonal of B instead of using generic matrix multiplication. This is taken into account in the complexity calculations.

6. *i.e.* the calculation of the terms $W_i\,\phi_{i-1}(x;W,b)+b_i$ which are assumed to have already been computed during a forward pass to compute the neural network's outputs.

7. Given two real-valued sequences $(u_n)_n$ and $(v_n)_n$, we say that $u_n = \Theta(v_n)$ if $u_n = \mathcal{O}(v_n)$ and $v_n = \mathcal{O}(u_n)$.

However, this complexity analysis neglects the memory occupation and time spent during loads and stores from and in memory that are necessary for the backward phase as one has to store the pre-activations of each layer during the forward phase in order to later load them and use them during the backward computation. The iterates $\delta_0, \ldots, \delta_p$ in the backward computations are also stored in memory so that another backward differentiation with respect to the weights of the neural network (and not its inputs), more commonly called *back-propagation*, can be performed in order to be able to make an SGD step towards solving (4). This second layer of backward differentiation is done automatically by all major neural network libraries using *adjoint algorithmic differentiation* (AAD) [1, 8, 27]. This memory cost, both in occupied space and in access times, is further exacerbated by the need to compute the gradient $\nabla \phi_{p+1}(x; W, b)$ separately for each point x in the sample when performing empirical risk minimization.

In [11], the authors point out that it is possible to reformulate the loss function in such a way that one would need only directional derivatives instead of full gradients in order to address the previous computational issue. We propose a more general result in the same spirit.

Proposition 1. *Let u be an L^2-integrable random vector supported on $\mathbb{R}^{n+2}$ with zero-mean components such that $\operatorname{cov}(u) = \operatorname{diag}(\lambda_1, \ldots, \lambda_{n+2})$ and assume that u is independent of Ξ, K, T, Z^{Ξ}. We have:*

$$\mathbb{E}[(u^\top \nabla \varphi(\Xi, K, T) - u^\top \nabla(e^{-rT} Z^{\Xi}))^2] = \sum_{k=1}^{n+2} \lambda_k \, \mathbb{E}[(\partial_k \varphi(\Xi, K, T) - \partial_k(e^{-rT} Z^{\Xi}))^2] \qquad (8)$$

Proposition 1 suggests that one can perform stochastic gradient descent to solve (4) by computing only one directional derivative along a random direction instead of having to compute full gradients during each SGD iteration. The authors in [11] suggested to draw u from a uniform distribution on the unit sphere but no discussion of the motivation behind this choice was provided. We give the optimal distribution among those verifying the assumptions of Proposition 1 when seeking to minimize the additional variance created by randomizing the direction of differentiation:

Proposition 2. *Denote $\ell := \nabla \varphi(\Xi, K, T) - \nabla(e^{-rT} Z^{\Xi})$ and $\mathcal{Z} = \sigma(\Xi, K, T, Z^{\Xi})$. Under the assumptions of Proposition 1, u minimizes $\mathbb{E}[\operatorname{var}((u^\top \ell)^2 | \mathcal{Z})]$ iff:*

$$u \sim \left(\sqrt{\lambda_1}\, R_1, \ldots, \sqrt{\lambda_{n+2}}\, R_{n+2}\right)$$

where $R_1, \ldots, R_N$ are i.i.d Rademacher variables, i.e. $\mathbb{Q}(R_i = -1) = \mathbb{Q}(R_i = 1) = \frac{1}{2}$.

Proof. We have:

$$\begin{aligned}
\operatorname{var}((u^\top \ell)^2 | \mathcal{Z}) &= \mathbb{E}[(\ell^\top (u\, u^\top - \operatorname{diag}(\lambda))\, \ell)^2 | \mathcal{Z}] \\
&= \mathbb{E}\left[\left(\sum_{i=1}^{n+2} \ell_i^2\, (u_i^2 - \lambda_i) + 2 \sum_{1 \le i < j \le n+2} \ell_i\, \ell_j\, u_i\, u_j\right)^2 \middle| \mathcal{Z}\right]
\end{aligned}$$

Denote $A := \sum_{i=1}^{n+2} \ell_i^2\, (u_i^2 - \lambda_i)$ and $B := \sum_{1 \le i < j \le n+2} \ell_i\, \ell_j\, u_i\, u_j$. We then have:

$$\operatorname{var}((u^\top \ell)^2 | \mathcal{Z}) = \mathbb{E}[A^2 + 4\, B^2 + 4\, A\, B | \mathcal{Z}]$$

and

$$
\begin{aligned}
A^2 &= \sum_{i=1}^{n+2} \ell_i^4 (u_i^2 - \lambda_i)^2 + 2 \sum_{1\le i<j\le n+2} \ell_i^2 \ell_j^2 (u_i^2 - \lambda_i)(u_j^2 - \lambda_j) \\
B^2 &= \sum_{1\le i<j\le n+2} \ell_i^2 \ell_j^2 u_i^2 u_j^2 + \sum_{\substack{1\le i<j\le n+2 \\ 1\le \imath<\jmath\le n+2 \\ (i,j)\neq(\imath,\jmath)}} \ell_i \ell_\imath \ell_j \ell_\jmath u_i u_\imath u_j u_\jmath \\
A\,B &= \sum_{r=1}^{n+2} \sum_{1\le i<j\le n+2} \ell_r^2 \ell_i \ell_j (u_r^2 - \lambda_r) u_i u_j
\end{aligned}
$$

Notice that whenever $i \neq j$, we have:

$$
\begin{aligned}
\mathbb{E}[\ell_i^2 \ell_j^2 (u_i^2 - \lambda_i)(u_j^2 - \lambda_j)|\mathcal{Z}] &= \ell_i^2 \ell_j^2 \mathbb{E}[u_i^2 - \lambda_i]\, \mathbb{E}[u_j^2 - \lambda_j] = 0 \\
\mathbb{E}[\ell_i^2 \ell_j^2 u_i^2 u_j^2|\mathcal{Z}] &= \ell_i^2 \ell_j^2 \lambda_i \lambda_j
\end{aligned}
$$

Let $i, \imath, j, \jmath$ be integers such that $1 \le i < j \le n+2$ and $1 \le \imath < \jmath \le n+2$ and $(i, j) \neq (\imath, \jmath)$. We have the following:

- if $i = \imath$ and $j \neq \jmath$: $\mathbb{E}[\ell_i \ell_\imath \ell_j \ell_\jmath u_i u_\imath u_j u_\jmath|\mathcal{Z}] = \ell_i^2 \ell_j \ell_\jmath \mathbb{E}[u_i^2]\, \mathbb{E}[u_j]\, \mathbb{E}[u_\jmath] = 0$
- if $i \neq \imath$ and $j = \jmath$: $\mathbb{E}[\ell_i \ell_\imath \ell_j \ell_\jmath u_i u_\imath u_j u_\jmath|\mathcal{Z}] = \ell_i \ell_\imath \ell_j^2 \mathbb{E}[u_i]\, \mathbb{E}[u_\imath]\, \mathbb{E}[u_j^2] = 0$

Let r be an integer such that $1 \le r \le n+2$. We can distinguish the following cases:

- if $r \neq i$ and $r \neq j$: $\mathbb{E}[\ell_r^2 \ell_i \ell_j (u_r^2 - \lambda_r) u_i u_j|\mathcal{Z}] = \ell_r^2 \ell_i \ell_j \mathbb{E}[u_r^2 - \lambda_r]\, \mathbb{E}[u_i]\, \mathbb{E}[u_j] = 0$
- if $r = i$: $\mathbb{E}[\ell_r^2 \ell_i \ell_j (u_r^2 - \lambda_r) u_i u_j|\mathcal{Z}] = \ell_r^3 \ell_j \mathbb{E}[(u_r^2 - \lambda_r) u_r]\, \mathbb{E}[u_j] = 0$
- if $r = j$: $\mathbb{E}[\ell_r^2 \ell_i \ell_j (u_r^2 - \lambda_r) u_i u_j|\mathcal{Z}] = \ell_r^3 \ell_i \mathbb{E}[(u_r^2 - \lambda_r) u_r]\, \mathbb{E}[u_i] = 0$

Hence,

$$
\operatorname{var}((u^\top \ell)^2|\mathcal{Z}) = \sum_{i=1}^{n+2} \ell_i^4 \mathbb{E}[(u_i^2 - \lambda_i)^2] + 4 \sum_{1\le i<j\le n+2} \ell_i^2 \ell_j^2 \lambda_i \lambda_j \tag{9}
$$

which is minimized iff $u_i^2 = \lambda_i$ a.s. for all $i \in \{1, \ldots, n+2\}$. Assume this is verified. Since u is centered, we have $\mathbb{Q}(u_i = \sqrt{\lambda_i}) = \mathbb{Q}(u_i = -\sqrt{\lambda_i}) = \frac{1}{2}$ for every $i \in \{1, \ldots, n+2\}$ and this concludes the proof. □

Remark 1. The expectation of the conditional variance in (9) also happens to be the additional variance caused by the introduction of a random direction of differentiation. Indeed, reusing the notation of Proposition 2 and setting $\tilde{\ell} := \varphi(\Xi, K, T) - e^{-rT} Z^\Xi$, we have:

$$
\operatorname{var}(\tilde{\ell}^2 + (u^\top \ell)^2) = \operatorname{var}(\tilde{\ell}^2) + \operatorname{var}((u^\top \ell)^2) + 2 \operatorname{cov}(\tilde{\ell}^2, (u^\top \ell)^2)
$$

Using the tower property one can show that:

$$
\operatorname{cov}(\tilde{\ell}^2, (u^\top \ell)^2) = \operatorname{cov}(\tilde{\ell}^2, \mathbb{E}[(u^\top \ell)^2|\mathcal{Z}])
$$

We also have from the total variance formula that:

$$
\operatorname{var}((u^\top \ell)^2) = \mathbb{E}[\operatorname{var}((u^\top \ell)^2|\mathcal{Z})] + \operatorname{var}(\mathbb{E}[(u^\top \ell)^2|\mathcal{Z}])
$$

Hence:

$$
\operatorname{var}(\tilde{\ell}^2 + (u^\top \ell)^2) = \operatorname{var}(\tilde{\ell}^2 + \mathbb{E}[(u^\top \ell)^2|\mathcal{Z}]) + \mathbb{E}[\operatorname{var}((u^\top \ell)^2|\mathcal{Z})]
$$

and the conclusion is immediate by noticing that:

$$\tilde{\ell}^2 + \mathbb{E}[(u^\top \ell)^2 | \mathcal{Z}] = (\varphi(\Xi, K, T) - e^{-rT} Z^\Xi)^2 + \sum_{k=1}^{n+2} \lambda_k \left(\partial_k \varphi(\Xi, K, T) - \partial_k (e^{-rT} Z^\Xi)\right)^2$$

Hence, the computational gain provided by differentiating along random directions instead of computing full gradients comes at the cost of a necessarily higher variance. In this regard, Proposition 2 helps reduce this additional variance as much as possible by judiciously choosing the distribution of u. By choosing the optimal distribution, the added variance is then:

$$\mathbb{E}[\mathrm{var}((u^\top \ell)^2 | \mathcal{Z})] = 4 \sum_{1 \le i < j \le n+2} \lambda_i \lambda_j \mathbb{E}[\ell_i^2 \ell_j^2]$$

□

One can then compute the directional derivatives using a forward approach, yielding similar complexity to the backward approach in terms of the arithmetic operations, while performing less memory accesses as no jacobians need to be stored in the forward approach. Indeed, one can iterate jointly on the forward approach and the calculation of the neural network output and share pre-activation values between both computations. Although the same directional derivatives computation can be done using the backward approach, even in terms of solely the arithmetic operations it is sub-optimal to do so. Figure 1 shows that when the number of inputs is greater than the number of outputs (*i.e.* $d_0 > d_2$ in the figure), vector-matrix multiplications from left to right perform less work and are thus faster.

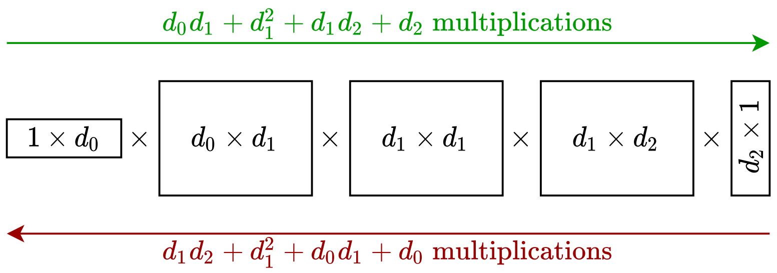

Figure 1. Jacobian multiplications when differentiating a neural network with 3 hidden layers having d_0 inputs, d_1 neurons per hidden layer and d_2 outputs, with respect to its inputs along a fixed direction.

4.2 Faster Directional Derivatives with Complex-step Differentiation

Notice that for a given direction u, the directional derivative $u^\top \nabla \varphi(\Xi, K, T)$ can also be approximated using a finite difference along the direction u at the cost of two[8] evaluations for both the output of the network and its directional derivative and would have to perform less work than an exact directional derivative with the forward approach given that the latter will have the additional overhead of the diagonal matrix multiplication in (6), which has quadratic complexity in the number of hidden units, while the forward pass using (3) doesn't.

8. or three when using a centered finite difference approximation.

However, the finite difference method suffers from round-off errors when implemented using finite precision arithmetic because of the substraction involved. In particular, the approximation can become unstable as shown in the example of Figure 2 and can fail for step sizes that are less than $\sim\sqrt{\epsilon}$ ($\sim\sqrt[3]{\epsilon}$ if using the central finite difference method) where ϵ is the machine epsilon [30]. This is further exacerbated by the fact that computations on the GPU are preferably done in single precision (where the machine epsilon is of the order of 10^{-7}) as switching to double precision (the machine epsilon becoming then $\sim 10^{-16}$) comes with a performance penalty. This greatly limits the degree to which one can reduce the approximation error by reducing the step size if one wants to preserve the computational advantage[9] of single precision, especially on GPUs.

In [22, 33], it is shown that if the function that is being differentiated admits an analytic extension, then these numerical issues can be overcome using a so-called *complex-step differentiation* instead of a finite difference approximation. We first present the result in scalar form, which can easily be verified using a Taylor expansion for holomorphic functions (we refer the reader to [20] for a classic treatment of complex analysis):

Proposition 3. (Complex-step differentiation) *Let $x\in\mathbb{R}$ and let $F:\mathbb{R}\to\mathbb{R}$ be thrice differentiable on a neighborhood V of x and assume that F can be extended analytically on V. By identifying F with its analytic extension on V, we have:*

$$\frac{1}{\varepsilon}\,\mathrm{Im}(F(x+\mathrm{i}\,\varepsilon)) = F'(x) + \mathcal{O}(\varepsilon^2)$$

where i *is the imaginary unit and* Im *is the imaginary part operator.*

Notice that the approximation in Proposition 3 does not involve a difference, hence being less subject to the round-off errors encountered in finite difference implementations, and is of order two, having thus the same order as a central finite difference but with better numerical stability. In practice, the complex step size can be taken to be as small as the machine epsilon, yielding an approximation that is almost indistinguishable from the exact value of the derivative in finite precision. In the example of Figure 2, the absolute error is of the order of the machine epsilon when ε is sufficiently small. Switching to holomorphic functions on $\mathbb{C}^{n+2}$, and assuming that the neural network φ admits an analytic extension with which we identify it, we can finally write:

$$u^\top\nabla\varphi(x) = \frac{1}{\varepsilon}\,\mathrm{Im}(\varphi(x+\mathrm{i}\,\varepsilon\,u)) + \mathcal{O}(\varepsilon^2)$$

where Im is applied element-wise. Notice that the output of the neural network can be deduced immediately without performing any additional evaluation, with the same order of error:

$$\varphi(x) = \mathrm{Re}(\varphi(x+\mathrm{i}\,\varepsilon\,u)) + \mathcal{O}(\varepsilon^2)$$

with Re being the real part operator applied element-wise. Hence both values and directional derivatives can be computed at the same time with low approximation error.

9. Not only are double-precision arithmetics, in theory, twice as slow as single-precision arithmetics, recent NVidia GPUs give even more advantage to single-precision (albeit in a specialized TensorFloat32 format) with speedups up to ×8 with respect to regular single-precision (*i.e.* floats or FP32) on the A100 GPU thanks to *Tensor cores*, at least according to NVidia's official specifications [25]. Of course, in addition to the computational speed, there is also the difference in memory storage, as double-precision will necessarily use twice as much memory as single-precision, and also involve twice as many memory exchanges which can be penalizing in situations that are memory-bound.

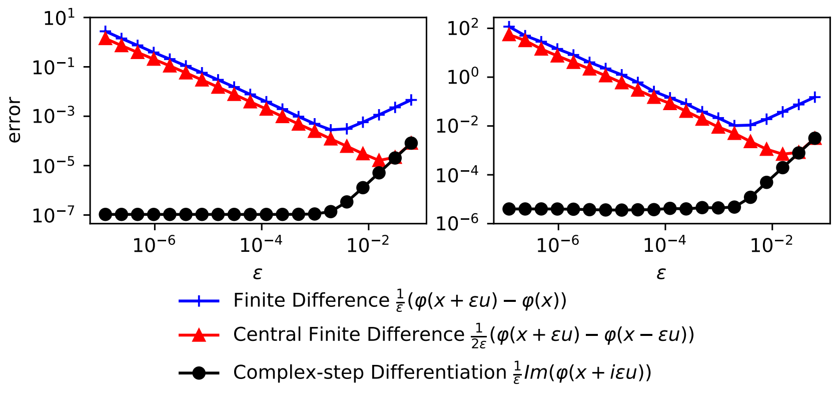

Figure 2. Sample average of the absolute value of absolute (**left**) and relative (**right**) errors, when approximating the directional derivative of a randomly initialized neural network φ, with 28 inputs, 6 hidden layers, 112 hidden units per layer and a Softplus activation, with respect to its inputs, using each of the finite difference, central finite difference and complex-step differentiation methods. The average is done over an i.i.d sample of $2^{14}=16384$ errors each corresponding to a network input vector x with components drawn independently from $\mathcal{U}([-\sqrt{3},\sqrt{3}])$ and a direction u drawn as in Proposition 2 with $\lambda_1=\cdots=\lambda_{28}=1$. Plots are in log-log scale.

In practice, a neural network can be rendered analytic by choosing activation functions that can be extended analytically, since the affine layers admit trivial analytic extensions.

Example 3. (Analytic Softplus activation) Consider the Softplus activation function:

$$\alpha(x)=\log(1+\exp(x)), \quad \forall x\in\mathbb{R}$$

and which is applied element-wise when supplied with a vector in its input. It can be extended naturally to a holomorphic function by extending log with $z\mapsto\log|z|+i\,\mathrm{Arg}(z)$ on $\mathbb{C}\setminus\{0\}$, with Arg being the principal argument on $(-\pi,\pi]$, and exp with $x+\mathrm{i}\,y\mapsto\exp(x)\,(\cos(y)+i\sin(y))$ on $\mathbb{C}$. Hence, except where undefined[10], we can use the following[11] extension for all $x,y\in\mathbb{R}$:

$$\alpha(x+\mathrm{i}y)=\frac{1}{2}\log(1+\exp(2x)+2\exp(x)\cos(y))+\mathrm{i}\,\mathrm{atan2}(\exp(x)\sin(y),1+\exp(x)\cos(y))$$

where $\mathrm{atan2}(y,x)=\mathrm{Arg}(x+\mathrm{i}\,y)$ and is implemented in most math libraries. One then extends α to vectors in $\mathbb{C}^{n+2}$ by applying it element-wise.

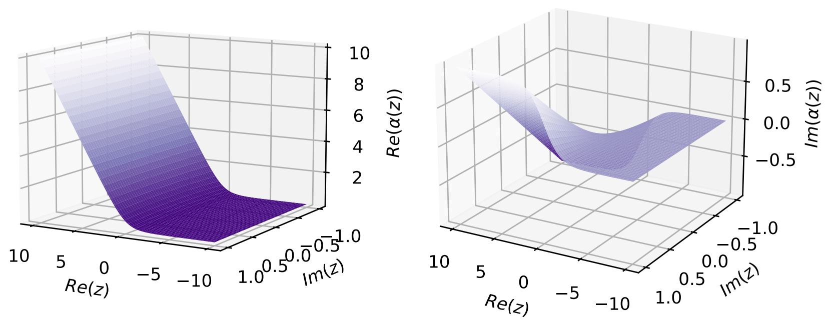

Figure 3. An analytic extension of the Softplus activation. **Left:** real part, **right:** imaginary part.

10. which is not an issue as the set of such points is of measure zero.

11. One still needs to treat overflow issues, caused by the exponential, based on the sign of x in the implementation.

□

The main difficulty then resides in implementing support for complex-valued inputs for neural networks in such a way that the library's automatic differentiation with respect to the network weights remains possible. At the time of this writing, this is done in `pytorch` for instance using the notion of Wirtinger differentiation [7] which is more general than holomorphic differentiability. However, we chose to specialize in holomorphic functions as these enjoy the Cauchy-Riemann property, which we recall below in the scalar case, as opposed to general Wirtinger differentiable functions.

Theorem 2. (Cauchy-Riemann equations) *Let $v: z = x + \mathrm{i}\, y \mapsto v_0(x, y) + \mathrm{i}\, v_1(x, y)$ be defined in a neighborhood V of $z_0 = x_0 + \mathrm{i}\, y_0$ and assume v is holomorphic on V. Then the partial derivatives $\partial_x v_0$, $\partial_y v_0$, $\partial_x v_1$ and $\partial_y v_1$ exist at (x_0, y_0) and we have:*

$$\begin{aligned}\partial_x v_0(x_0, y_0) &= \partial_y v_1(x_0, y_0)\\ \partial_y v_0(x_0, y_0) &= -\partial_x v_1(x_0, y_0)\end{aligned}$$

The Cauchy-Riemann equations in particular allow one to compute only two partial derivatives (*e.g.* the partial derivatives of the real part of the output with respect to the real and imaginary parts of the input) and deduce the other two partial derivatives with at most a change of sign. This in particular allows us to directly hard-code[12] the fact that one needs to store only two partial derivatives and use those to immediately get all four partial derivatives (of the real and imaginary parts of the output with respect to the real and imaginary parts of the inputs) at all intermediate layers during the back-propagation, while Wirtinger differentiation requires in principle keeping track of all four partial derivatives as the Cauchy-Riemann equations are not necessarily verified for a Wirtinger differentiable function.

Using our custom holomorphic implementation in `C++` yields speed-ups between $\times 1.4$ and $\times 2$ compared to `PyTorch`'s Wirtinger-based back-propagation[13] in our benchmarks of back-propagation times in 5.2. We also get speed-ups between $\times 3.6$ and $\times 7.6$ compared to an exact calculation of directional derivatives using the forward approach.

5 Numerical Case-study: Fixed-grid Local Volatility

5.1 Setup of The Experiments

We consider the problem of fitting a 5×5 fixed-grid local volatility model, as described in Example 1, on observed quotes of call and put prices. We set $X_0 = 1$ and for the local volatility grid we use a time grid $\left\{\frac{1}{12} + \frac{(2 - 1/12)\, k}{4}\right\}_{0 \leq k \leq 4}$ and a spatial grid $\left\{\log(0.25) + \frac{\log(2.1/0.25)\, k}{5}\right\}_{0 \leq k \leq 4}$.

12. We do this using the `C++ libtorch` library by reimplementing the needed neural network blocks, such as activations and layers, by inheriting from the `torch::autograd::Function` class and exploiting the Cauchy-Riemann property in our custom `torch::autograd::Function::backward` implementation.

13. We used PyTorch's *just-in-time* (JIT) compilation mechanism when benchmarking the vanilla complex implementation on Python, which removes in practice most of the Python overhead. Hence the speed-up is mostly explained by our specialization to holomorphic functions and not merely by switching to `C++`. The switch to `C++` was needed only because our custom implementation using CUDA kernels could not, at the moment of this writing, be used with the JIT mechanism, thus severely penalizing it on Python.

In this example then, we have $d=1$, $\psi=\mathrm{Id}$ and $n=26$ corresponding to 25 local volatility nodes and the risk-free rate.

We generate $2^{23}\approx 8$ million Monte Carlo samples of (Ξ, K, T, Z) assuming a time discretization with an Euler-Maruyama scheme. We sample each factor as in Table 1, which essentially amounts to seeking an approximation of the pricing function for parameters in the described ranges. Since the simulations are fast (see 5.2), we also use a modest variance reduction by replacing the payoffs with an average over 32 realizations[14] of the payoff conditional on the same parameter realization, since both have the same expectation conditional on (Ξ, K, T, Z).

Factor	**Description**	**Distribution**
r	risk-free rate	$\mathcal{U}([0, 0.05])$
K	strike price	$\mathcal{U}([0.25, 2.1])$
T	maturity	$\mathcal{U}([0.05, 2.5])$
$\sigma_{i,j}$	local volatility at node (i,j)	$\mathcal{U}([0.1, 2.0])$

Table 1. Distributions of product and model parameters.

We used a neural network with $p=6$ layers, $m=56$ hidden units per hidden layer, an analytic Softplus activation and, because we are specializing in call prices, $q=2$ outputs constrained to be valued on $[0,1]$, on which an inner-product with $(X_0, -e^{-rT}K)^\top$ is then performed to get an estimate of the call price. More precisely, let $\mathcal{N} = \mathcal{NN}_{56,6,2,\alpha}$ and denote by $[.]_k$ the k-th coordinate of its argument (with k zero-indexed). Define:

$$\mathcal{H} := \{\mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \ni (\xi, k, \tau) \mapsto X_0\, s([\phi(\xi, k, \tau)]_0) - e^{-[\xi]_0 \tau} k\, s([\phi(\xi, k, \tau)]_1) : \phi \in \mathcal{N}\}$$

where $s: \mathbb{R} \ni x \mapsto \frac{1}{1+\exp(-x)}$. Then the minimization is carried over $\mathcal{H}$ instead of $\mathcal{N}$. This helps ensure proper scaling for the price approximation with no need to rescale the outputs by hand and is motivated by the following observation:

$$\mathbb{E}[e^{-rT} Z^{\Xi} | \Xi, K, T] = X_0 \underbrace{\mathbb{E}\left[e^{-rT} \frac{X_T^{\Xi}}{X_0} \mathbb{1}_{\{X_T^{\Xi} \geq K\}} | \Xi, K, T\right]}_{\in[0,1]} - e^{-rT} K\, \mathbb{Q}(X_T^{\Xi} \geq K | \Xi, K, T)$$

5.2 Execution Times and Benchmarks

Unless otherwise stated, the experiments are done on a single Nvidia V100 GPU, although one could readily extend our implementation to multi-GPU training using lock-free approaches such as Hogwild training [29]. The simulations are fast as no pricing is being performed and path-wise sensitivities of the payoffs are computed with the forward approach described in Example 2. Both are implemented on GPU in `CUDA` and take $\sim$1 min 20 secs.

14. From a GPU programming perspective, a size of 32 is attractive, especially for Nvidia GPUs since a warp is of size 32 [26], as it gives the possibility to have threads inside the same warp work on the same model and product parameters and thus prevent thread divergences that would otherwise occur if for example two threads of the same warp worked on two different realizations of the maturity T. It also allows to compute the empirical average over the 32 conditional realizations using only warp intrinsics and without resorting to shared memory or synchronization barriers.

To demonstrate the effectiveness of our approach in terms of speed and memory footprint, we reported in tables 2 and 3 the execution time and memory usage when performing one iteration of back-propagation, *i.e.* the library's AAD with respect to network weights, through a neural network with 5 layers and respectively 64 and 128 neurons per layer, using a loss function involving either a sum of errors of each derivative of the network with respect to its inputs (*i.e.* similarly to the second sum in (4)) or one error involving a random directional derivative (*i.e.* as in the LHS of (8)). The last two rows correspond to our proposed complex-step differentiation approach and refer to respectively using `pytorch`'s Wirtinger-based AAD for complex functions and our custom `C++` implementation specializing in holomorphic functions. For the sake of accuracy, the other rows are all implemented in `C++` but very similar timings were also achieved using simply `pytorch`'s *just-in-time* compilation mechanism in plain Python.

Training for 300 epochs, with 512 batch iterations per epoch (amounting in total to 153600 SGD iterations), takes ∼13mins (compared to at least hours for equivalent accuracy when training using full gradients). Evaluating 1024 prices along with their 28 derivatives during inference takes ∼5ms on an Nvidia T4 GPU, which is very appealing for model calibration routines.

	Time	**Cumul. speedup**	**Mem. usage**
Full gradients, backward	17.01(0.40)		2365.68
Full gradients, forward	16.56(0.48)	×1.03	1330.74
Exact directional derivatives	6.42(0.14)	×2.65	433.07
CSD directional derivatives	2.60(0.19)	×6.54	139.74
CSD directional derivatives, `C++`	**1.75(0.27)**	**×9.72**	**90.20**

Table 2. Time spent (in ms), cumulative speed-ups and memory usage (in MB) during 1 iteration of back-propagation for different differentiation procedures for a neural network with 64 neurons/layer and 5 hidden layers, assuming 28-dimensional input and scalar outputs.

	Time	**Cumul. speedup**	**Mem. usage**
Full gradients, backward	59.65(1.00)		8834.49
Full gradients, forward	48.42(0.48)	×1.23	3169.55
Exact directional derivatives	18.86(0.16)	×3.16	1343.92
CSD directional derivatives	5.17(0.04)	×11.54	276.92
CSD directional derivatives, `C++`	**2.49(0.28)**	**×23.95**	**178.00**

Table 3. Time spent (in ms), cumulative speed-ups and memory usage (in MB) during 1 iteration of back-propagation for different differentiation procedures for a neural network with 128 neurons/layer and 5 hidden layers, assuming 28-dimensional input and scalar outputs.

5.3 Validation Without Ground-truth Values

Even though we don't compute *ground-truth* prices, *e.g.* using a dedicated Monte-Carlo simulation for each set of model and product parameters, as we are only simulating payoffs (or a conditional sample average over a very small sample), one can still estimate an L^2 distance to the ground-truth prices by following a *twin-simulation* procedure introduced in [2]. The idea is to notice that if φ is our neural network, then:

$$\begin{aligned}\mathbb{E}[(\varphi(\Xi,K,T)-\mathbb{E}[e^{-rT}Z^{\Xi}|\Xi,K,T])^2] &= \mathbb{E}[\varphi(\Xi,K,T)\,(\varphi(\Xi,K,T)-e^{-rT}\,(Z^{\Xi,1}+Z^{\Xi,2}))] \\ &\quad +\mathbb{E}[e^{-2rT}Z^{\Xi,1}Z^{\Xi,2}]\end{aligned}$$

where $Z^{\Xi,1}$ and $Z^{\Xi,2}$ are two conditionally independent copies of Z^{Ξ} given (Ξ, K, T). Hence, via two sub-simulations conditional on each realization of the model and product parameters, one can estimate the L^2 distance between the neural network approximation and the ground-truth price without ever computing the latter. Assuming a sample of size N independent of the trajectories which were used for training, we can estimate this distance using the following unbiased estimator:

$$\mathrm{MSE}_N^{\mathrm{twin}} := \frac{1}{N}\sum_{i=1}^{N} \varphi(\Xi^{(i)}, K^{(i)}, T^{(i)})\left(\varphi(\Xi^{(i)}, K^{(i)}, T^{(i)}) - e^{-r^{(i)}T^{(i)}}\left(Z^{\Xi^{(i)},1} + Z^{\Xi^{(i)},2}\right)\right) \\ + e^{-2r^{(i)}T^{(i)}} Z^{\Xi^{(i)},1} Z^{\Xi^{(i)},2}$$

where $((\Xi^{(i)}, K^{(i)}, T^{(i)}))_{1\le i\le N}$ is an i.i.d sample of (Ξ, K, T) and for each $i \in \{1, \ldots, N\}$, $Z^{\Xi^{(i)},1}$ and $Z^{\Xi^{(i)},2}$ are two conditionally independent copies of Z^{Ξ} given $(\Xi, K, T) = (\Xi^{(i)}, K^{(i)}, T^{(i)})$ and the sample $\left(\left(\Xi^{(i)}, K^{(i)}, T^{(i)}, Z^{\Xi^{(i)},1}, Z^{\Xi^{(i)},2}\right)\right)_{1\le i\le N}$ is independent of the sample that was used for the training.

In Table 4, we list the estimates we obtain for our method along with alternative approaches using only payoffs, *i.e.* approaches not seeking to project path-wise sensitivities.

	$\sqrt{\mathrm{MSE}_N^{\mathrm{twin}}}$	stdev of $\mathrm{MSE}_N^{\mathrm{twin}}$
Projecting both payoffs and path-wise sensitivities	$\mathbf{4.9 \cdot 10^{-4}}$	$1.7 \cdot 10^{-6}$
Projecting only payoffs, same # of samples	$7.06 \cdot 10^{-3}$	$1.7 \cdot 10^{-6}$
Projecting only payoffs, $2\times$ as many samples	$4.30 \cdot 10^{-3}$	$1.7 \cdot 10^{-6}$
Projecting only payoffs, $4\times$ as many samples	$1.42 \cdot 10^{-3}$	$1.7 \cdot 10^{-6}$

Table 4. $\mathrm{MSE}_N^{\mathrm{twin}}$ estimates for different approaches for projecting payoffs.

5.4 Calibration Example

We consider the problem of fitting our fixed-grid local volatility nodes by using the price approximation learned by our neural network in the calibration phase. More precisely, we seek local volatility nodes $(\sigma_{i,j})_{\substack{1\le i\le m \\ 1\le j\le M}}$ such that:

$$(\sigma_{i,j})_{i,j} \in \underset{(\sigma_{i,j})_{i,j}\in[0.1,2]^{[\![1,m]\!]\times[\![1,M]\!]}}{\operatorname{argmin}} \sum_{l=1}^{L} \left(\varphi((r, \sigma_{0,0}, \ldots, \sigma_{i,j}, \ldots \sigma_{m,M}), k^{(l)}, \tau^{(l)}) - p_{\mathrm{mkt}}^{(l)}\right)^2$$

where we assume that we have access to L market prices of vanilla calls $p_{\mathrm{mkt}}^{(1)}, \ldots, p_{\mathrm{mkt}}^{(L)}$ corresponding to strikes $k^{(1)}, \ldots, k^{(L)}$ and maturities $\tau^{(1)}, \ldots, \tau^{(L)}$, and for simplicity of the experiment a flat term structure of risk-free rates.

Using the same grid of size 5×5 for the local volatility as previously, we solve the above calibration problem using Tesla's stock (Nasdaq:`TSLA`) as our underlying on 2022/02/14 using a vanilla gradient descent with respect to the local volatility nodes. We show in Figure 4 the smiles resulting from the model parameters obtained through the neural network pricing proxy. Under given model parameters, the smiles are constructed using a slow Monte-Carlo pricing to ensure a validation that is independent of the neural network approximation. We obtain close to perfect fits when compared to the implied volatilities of the market prices.

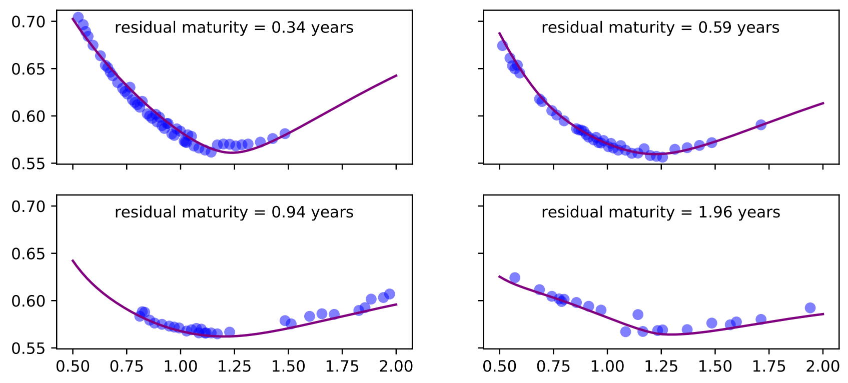

Figure 4. Fit of TSLA implied volatility smiles on 2022/02/14 using a 5×5 local volatility model calibrated using our proxy pricer. **Blue dots:** market prices, **purple curve:** implied volatility smile of fitted local volatility model, ***x*-axis:** moneyness, ***y*-axis:** implied volatility levels.

Although this is not the goal of the present article, notice that the flexibility to specify for instance custom grids for the local volatility, and easily by extension stochastic-local volatility models [14, 34], gives a new possibility to regularize calibration problems by enforcing parsimony directly at the grid level.

The calibration approach is entirely orthogonal to the procedure presented in this paper. One can imagine more elaborate calibration procedures based on Levenberg-Marquardt or other algorithms. The main idea remains that inside the calibration procedure, the pricing function is replaced by the approximation given by our neural network, and, whenever derivatives are needed, these can be computed either explicitly or using automatic differentiation.

Appendix A Derivatives with respect to time to maturity in discrete time models

Let $\xi \in \mathbb{R}^n$ and $r, \tau, k > 0$. In our general SDE (1) we will assume one single spatial dimension in X_t (*i.e.* $d = 1$) and we will consider the payoff of a vanilla call with time to maturity τ and assume risk-free rate r. Let $h(x) = (x - k)^+$. For every $t > 0$, let $x \mapsto p(t, x)$ be the probability density of X_t^ξ. Then, for all $x \in \mathbb{R}$, the density p follows the following Fokker-Planck equation at (τ, x):

$$\partial_\tau p(\tau, x) = -\partial_x (f(\tau, x, \xi)\, p(\tau, x)) + \frac{1}{2} \partial_{xx} (g(\tau, x, \xi)^2\, p(\tau, x)) \tag{10}$$

We will assume in the following that X_τ^ξ is at least L^3-integrable. In this case the Lipschitz regularity of f and g in their second argument yields:

$$\begin{aligned} \lim_{x \to +\infty} x\, f(\tau, x, \xi)\, p(\tau, x) &= 0 \\ \lim_{x \to +\infty} g(\tau, x, \xi)^2\, p(\tau, x) &= 0 \\ \lim_{x \to +\infty} x\, \partial_x (g(\tau, x, \xi)^2\, p(\tau, x)) &= 0 \end{aligned}$$

Then using integration by parts, we have:

$$\int_{-\infty}^{+\infty} h(x)\,\partial_x(f(\tau,x,\xi)\,p(\tau,x))\,\mathrm{d}x \;=\; e^{-r\tau}\int_{-\infty}^{+\infty}(x-k)\,\partial_x(f(\tau,x,\xi)\,p(\tau,x))\,\mathrm{d}x$$
$$= -e^{-r\tau}\mathbb{E}\big[\,f(\tau,X_\tau^\xi,\xi)\,\mathbb{1}_{\{X_\tau^\xi\geq k\}}\big] \tag{11}$$

and

$$\int_{-\infty}^{+\infty} h(x)\,\partial_{xx}(g(\tau,x,\xi)^2\,p(\tau,x))\,\mathrm{d}x \;=\; e^{-r\tau}\int_{-\infty}^{+\infty}(x-k)\,\partial_{xx}(g(\tau,x,\xi)^2\,p(\tau,x))\,\mathrm{d}x$$
$$= e^{-r\tau} g(\tau,k,\xi)^2\,p(\tau,k)$$

We also have via the Breeden-Litzenberger formula that $p(\tau,k)=\partial_{kk}\mathbb{E}[(X_\tau^\xi-k)^+]$, thus:

$$\int_{-\infty}^{+\infty} h(x)\,\partial_{xx}(g(\tau,x,\xi)^2\,p(\tau,x))\,\mathrm{d}x = e^{-r\tau} g(\tau,k,\xi)^2\,\partial_{kk}\mathbb{E}[(X_\tau^\xi-k)^+] \tag{12}$$

Notice now that we have:

$$\partial_\tau(e^{-r\tau}\mathbb{E}[(X_\tau^\xi-k)^+]) = -r\,e^{-r\tau}\mathbb{E}[(X_\tau^\xi-k)^+] + e^{-r\tau}\int_k^{+\infty}(x-k)\,\partial_\tau p(\tau,x)\,\mathrm{d}x \tag{13}$$

Plugging the expressions of (11), (12) and (13) into the Fokker-Planck equation in (10), we get:

$$\partial_\tau(e^{-r\tau}\mathbb{E}[(X_\tau^\xi-k)^+]) \;=\; -r\,e^{-r\tau}\mathbb{E}[(X_\tau^\xi-k)^+] + e^{-r\tau}\mathbb{E}\big[\,f(\tau,X_\tau^\xi,\xi)\,\mathbb{1}_{\{X_\tau^\xi\geq k\}}\big]$$
$$+\frac{1}{2}e^{-r\tau} g(\tau,k,\xi)^2\,\partial_{kk}\mathbb{E}[(X_\tau^\xi-k)^+] \tag{14}$$

Finally notice that for any $0<h<\tau$, applying the tower property and the Breeden-Litzenberger formula conditionally at $\tau-h$ we have:

$$\begin{aligned}\partial_{kk}\mathbb{E}[(X_\tau^\xi-k)^+] &= \mathbb{E}[\partial_{kk}\mathbb{E}[(X_\tau^\xi-k)^+|X_{\tau-h}^\xi]]\\ &= \mathbb{E}[p^h(\tau,k|X_{\tau-h}^\xi)]\end{aligned}$$

where $x\mapsto p^h(\tau,x|x')$ is the density of X_τ^ξ conditional on $X_{\tau-h}^\xi=x'$.

Thus, going back to random model parameters, thanks to (14) one can write the derivative of the price with respect to time to maturity as a conditional expectation involving either only one second spatial derivative, or no derivatives at all if one knows the conditional density $p^h(\tau,.|.)$ of $X_\tau^\xi|X_{\tau-h}^\xi$. The latter can in practice be replaced, given sufficiently small h, with the closed-form available in discrete time given that increments in an Euler scheme for example are Gaussian (or log-normal if one discretizes the log-dynamics). We refer to works such as [3] for a quantification of the error when using the density of the discrete solution instead of that of the continuous time solution.

We used this approach in our implementation for the derivative with respect to time to maturity, by considering this integrand:

$$r\,e^{-r\tau}\left(-(X_\tau^\xi-k)^+ + f(\tau,X_\tau^\xi,\xi)\,\mathbb{1}_{\{X_\tau^\xi\geq k\}} + \frac{1}{2}g(\tau,k,\xi)^2\,p^h(\tau,k|X_{\tau-h}^\xi)\right)$$

as a *fictitious* path-wise sensitivity with respect to time to maturity.

Appendix B Differentiation of the local volatility function in Example 2

We resume here the calculations of Example 2. We have:

- If there exist $p \in \{1, \ldots, m-1\}$ and $q \in \{1, \ldots, M-1\}$ such that $t^{(p)} \leq t_i < t^{(p+1)}$ and $s^{(q)} \leq \hat{s}_i < s^{(q+1)}$, then:

$$\partial_{u,v}\sigma(t_i, \hat{s}_i) = \begin{cases} \dfrac{(t^{(p+1)} - t)(s^{(q+1)} - \hat{s}_i)}{(t^{(p+1)} - t^{(p)})(s^{(q+1)} - s^{(q)})} & \text{if} \quad (u,v) = (p,q) \\ \dfrac{(t - t^{(p)})\,(s^{(q+1)} - \hat{s}_i)}{(t^{(p+1)} - t^{(p)})\,(s^{(q+1)} - s^{(q)})} & \text{if} \quad (u,v) = (p+1,q) \\ \dfrac{(t^{(p+1)} - t)\,(\hat{s}_i - s^{(q)})}{(t^{(p+1)} - t^{(p)})\,(s^{(q+1)} - s^{(q)})} & \text{if} \quad (u,v) = (p,q+1) \\ \dfrac{(t - t^{(p)})\,(\hat{s}_i - s^{(q)})}{(t^{(p+1)} - t^{(p)})\,(s^{(q+1)} - s^{(q)})} & \text{if} \quad (u,v) = (p+1,q+1) \\ 0 & \text{otherwise} \end{cases}$$

- If $t \geq t^{(m)}$ and there exists $q \in \{1, \ldots, M-1\}$ such that $s^{(q)} \leq \hat{s}_i < s^{(q+1)}$, then:

$$\partial_{u,v}\sigma(t_i, \hat{s}_i) = \begin{cases} \dfrac{s^{(q+1)} - \hat{s}_i}{s^{(q+1)} - s^{(q)}} & \text{if} \quad (u,v) = (m,q) \\ \dfrac{\hat{s}_i - s^{(q+1)}}{s^{(q+1)} - s^{(q)}} & \text{if} \quad (u,v) = (m,q+1) \\ 0 & \text{otherwise} \end{cases}$$

- If $t < t^{(0)}$ and there exists $q \in \{1, \ldots, M-1\}$ such that $s^{(q)} \leq \hat{s}_i < s^{(q+1)}$, then:

$$\partial_{u,v}\sigma(t_i, \hat{s}_i) = \begin{cases} \dfrac{s^{(q+1)} - \hat{s}_i}{s^{(q+1)} - s^{(q)}} & \text{if} \quad (u,v) = (0,q) \\ \dfrac{\hat{s}_i - s^{(q+1)}}{s^{(q+1)} - s^{(q)}} & \text{if} \quad (u,v) = (0,q+1) \\ 0 & \text{otherwise} \end{cases}$$

- If $s \geq s^{(M)}$ and there exists $p \in \{1, \ldots, m-1\}$ such that $t^{(p)} \leq t_i < t^{(p+1)}$, then:

$$\partial_{u,v}\sigma(t_i, \hat{s}_i) = \begin{cases} \dfrac{t^{(p+1)} - t}{t^{(p+1)} - t^{(p)}} & \text{if} \quad (u,v) = (p,M) \\ \dfrac{t - t^{(p)}}{t^{(p+1)} - t^{(p)}} & \text{if} \quad (u,v) = (p+1,M) \\ 0 & \text{otherwise} \end{cases}$$

- If $s < s^{(0)}$ and there exists $p \in \{1, \ldots, m-1\}$ such that $t^{(p)} \leq t_i < t^{(p+1)}$, then:

$$\partial_{u,v}\sigma(t_i, \hat{s}_i) = \begin{cases} \dfrac{t^{(p+1)} - t}{t^{(p+1)} - t^{(p)}} & \text{if} \quad (u,v) = (p,0) \\ \dfrac{t - t^{(p)}}{t^{(p+1)} - t^{(p)}} & \text{if} \quad (u,v) = (p+1,0) \\ 0 & \text{otherwise} \end{cases}$$

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al, *TensorFlow: a system for Large-Scale machine learning*, in 12th USENIX symposium on operating systems design and implementation (OSDI 16), 2016, pp. 265–283.

[2] L. Abbas-Turki, S. Crépey, and B. Saadeddine, *Hierarchical simulation for learning with defaults*. Working paper available on https://www.lpsm.paris/pageperso/crepey, 2021.

[3] V. Bally and D. Talay, *The law of the euler scheme for stochastic differential equations: ii. convergence rate of the density*, (1996).

[4] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: a survey*, Journal of Marchine Learning Research, 18 (2018), pp. 1–43.

[5] C. Bayer, P. Friz, and J. Gatheral, *Pricing under rough volatility*, Quantitative Finance, 16 (2016), pp. 887–904.

[6] C. Bayer and B. Stemper, *Deep calibration of rough stochastic volatility models*, arXiv preprint arXiv:1810.03399, (2018).

[7] P. Bouboulis, *Wirtinger's calculus in general hilbert spaces*, arXiv preprint arXiv:1005.5170, (2010).

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, *JAX: composable transformations of Python+NumPy programs*. 2018.

[9] M. Broadie and P. Glasserman, *Estimating security price derivatives using simulation*, Management science, 42 (1996), pp. 269–285.

[10] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Mathematics of control, signals and systems, 2 (1989), pp. 303–314.

[11] W. M. Czarnecki, S. Osindero, M. Jaderberg, G. Swirszcz, and R. Pascanu, *Sobolev training for neural networks*, Advances in Neural Information Processing Systems, 30 (2017).

[12] J. De Spiegeleer, D. B. Madan, S. Reyners, and W. Schoutens, *Machine learning for quantitative finance: fast derivative pricing, hedging and fitting*, Quantitative Finance, 18 (2018), pp. 1635–1643.

[13] P. Glasserman, *Monte Carlo methods in financial engineering*, vol. 53, Springer, 2004.

[14] J. Guyon and P. Henry-Labordere, *The smile calibration problem solved*, Available at SSRN 1885032, (2011).

[15] A. Hernandez, *Model calibration with neural networks*, Available at SSRN 2812140, (2016).

[16] B. Horvath, A. Muguruza, and M. Tomas, *Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models*, Quantitative Finance, 21 (2021), pp. 11–27.

[17] B. N. Huge and A. Savine, *Differential machine learning*, Available at SSRN 3591734, (2020).

[18] P. Kidger and T. Lyons, *Universal approximation with deep narrow networks*, in Conference on learning theory, 2020, PMLR, pp. 2306–2327.

[19] D. P. Kingma and J. Ba, *Adam: a method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[20] S. Lang, *Complex analysis*, vol. 103, Springer Science & Business Media, 2003.

[21] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, nature, 521 (2015), pp. 436–444.

[22] J. R. Martins, P. Sturdza, and J. J. Alonso, *The complex-step derivative approximation*, ACM Transactions on Mathematical Software (TOMS), 29 (2003), pp. 245–262.

[23] R. McCrickerd and M. S. Pakkanen, *Turbocharging monte carlo pricing for the rough bergomi model*, Quantitative Finance, 18 (2018), pp. 1877–1886.

[24] T. Nguyen-Thien and T. Tran-Cong, *Approximation of functions and their derivatives: a neural network implementation with applications*, Applied Mathematical Modelling, 23 (1999), pp. 687–704.

[25] NVIDIA Corporation, *Nvidia a100 datasheet*. 2020.

[26] NVIDIA Corporation, *Programming guide: CUDA toolkit documentation*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2020. Accessed: 2020-04-28.

[27] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al, *Pytorch: an imperative style, high-performance deep learning library*, Advances in neural information processing systems, 32 (2019).

[28] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006.

[29] B. Recht, C. Re, S. Wright, and F. Niu, *Hogwild!: a lock-free approach to parallelizing stochastic gradient descent*, Advances in neural information processing systems, 24 (2011).

[30] T. Sauer, *Numerical analysis*, Addison-Wesley Publishing Company, 2011.

[31] A. Savine, *Modern computational finance: AAD and parallel simulations*, John Wiley & Sons, 2018.

[32] H. Son, J. W. Jang, W. J. Han, and H. J. Hwang, *Sobolev training for physics informed neural networks*, arXiv preprint arXiv:2101.08932, (2021).

[33] W. Squire and G. Trapp, *Using complex variables to estimate derivatives of real functions*, SIAM review, 40 (1998), pp. 110–112.

[34] Y. Tian, Z. Zhu, G. Lee, F. Klebaner, and K. Hamza, *Calibrating and pricing with a stochastic-local volatility model*, The Journal of Derivatives, 22 (2015), pp. 21–39.

[35] N. N. Vlassis and W. Sun, *Sobolev training of thermodynamic-informed neural networks for interpretable elasto-plasticity models with level set hardening*, Computer Methods in Applied Mechanics and Engineering, 377 (2021), p. 113695.