OR Spectrum (2022) 44:947–965
https://doi.org/10.1007/s00291-022-00672-1

ORIGINAL ARTICLE

Derivatives of feed-forward neural networks and their application in real-time market risk management

Antal Ratku[1,2] · **Dirk Neumann**[2]

Received: 26 March 2021 / Accepted: 25 February 2022 / Published online: 21 March 2022

Abstract

Market risk management of financial derivatives requires the efficient calculation of their price sensitivities with respect to changes in market factors. This paper shows how a deep feed-forward neural network which has been trained for pricing derivative instruments can be efficiently used to calculate these sensitivities as well. The proposed method is a fast and easily implementable alternative approach to automatic differentiation, and it simultaneously calculates all the first- and second-order derivatives of a multilayer feed-forward neural network with respect to its input features. The paper quantifies the performance improvement of the proposed method over a recent, publicly available implementation of automatic differentiation for a wide range of network sizes. The number of input parameters in these networks corresponds to those of commonly used financial models with stochastic volatility. The numerical accuracy of the proposed sensitivity calculations is demonstrated with a case study, calculating price sensitivities of European options under stochastic volatility. While the paper focuses on financial applications, the results presented herein are applicable to all deep feed-forward neural networks with sufficiently smooth activation functions.

Keywords Investment analysis · Risk analysis · Artificial intelligence · Machine learning · Sensitivity analysis · Pricing

✉ Antal Ratku
antal@antalratku.com

Dirk Neumann
dirk.neumann@is.uni-freiburg.de

[1] Chair for Information Systems Research, University of Freiburg, Rempartstrasse 16, 79098 Freiburg, Germany

[2] Allianz Global Investors GmbH, Seidlstraße 24-24a, 80335 Munich, Germany

1 Introduction

The financial community has increasingly embraced the use of artificial intelligence for pricing and hedging derivative instruments. Although these two problems are closely related to each other from a financial perspective, the artificial intelligence methods used to tackle them are usually very different. On the one hand, the recent research around hedging derivatives mainly focuses on deriving an optimal strategy using recurrent neural networks and reinforcement learning (see, e.g., Buehler et al. (2019) and Kolm and Ritter (2019), respectively). On the other hand, the pricing problem is usually tackled by using deep feed-forward neural networks to replace computationally expensive pricing models. This paper positions itself between these two areas and shows how a deep feed-forward neural network that has been trained for pricing can also be used for the efficient calculation of first- and second-order price sensitivities, which are essential for model-based hedging decisions.

The benefit of using deep feed-forward neural networks for pricing problems lies in their universal approximating capability (Hornik et al. (1989)): after being trained with the help of supervised learning, such networks are able to approximate the original pricing function with arbitrary accuracy. Furthermore, the simple mathematical operations involved in calculating the network's output enable a significant speed-up in computation times compared to the original pricing problem. Not surprisingly, the associated performance improvement is the most obvious for financial models which require either Monte Carlo simulations or computationally expensive numerical integration to arrive at the price of the derivative instrument.

The applicability of neural networks to replace derivatives pricing functions has been demonstrated for a wide range of commonly used financial models. As some of the very first research contributions in this field, Hutchinson et al. (1994) and Anders et al. (1998) use neural networks for option pricing in the Black-Scholes world. More recently, Liu et al. (2019) and Horvath et al. (2020) show that feed-forward neural networks can even be used in financial models which assume stochastic or rough volatility. Even though most of the research articles are concerned with pricing European style options on equity indexes, the same techniques are also applicable to other derivative instruments, such as American options on single name instruments (see, e.g., Gaspar et al. (2020)). For a comprehensive overview of literature using neural networks for options pricing, see Ruf and Wang (2020).

In practical applications, determining the fair price of a derivative instrument is usually only one part of the pricing problem. After a derivative has been traded, its value might be sensitive to changes in a wide range of market factors, resulting in potentially unwanted market risk exposures. Therefore, it is often necessary to calculate and monitor the price sensitivities of the instrument with respect to these market parameters. The first-order price sensitivities imply how the value of the instrument would change when market parameters evolve in isolation. At the same time, the second-order price sensitivities are essential to judge the effect of larger movements in market parameters, as well as the joint impact of a simultaneous move in multiple market factors.

Given the increasing speed of trading, a growing number of financial use-cases require that these sensitivities are not only calculated accurately, but also as fast as possible. For instance, monitoring the market risk exposures of a continuously changing trading book in real-time, or performing large scale scenario analyses on derivatives portfolios are two of the many applications carried out on an ongoing basis at trading and quantitative portfolio management firms. For these applications, using neural networks can provide the required performance benefits while hardly sacrificing any accuracy. As proven by Hornik et al. (1990), deep feed-forward neural networks are not only capable of approximating a given function with arbitrary accuracy, but also its first- and higher-order derivatives. Therefore, using the techniques outlined in the rest of this paper, they can be efficiently used to support real-time risk analytics and trading decisions.

To show how to calculate the sensitivities from a neural network that has been trained for pricing, Sect. 2 provides analytic expressions for a multilayer feed-forward neural network's Jacobian and Hessian matrices with respect to its input parameters. The calculations in this section generalize the results of Dimopoulos et al. (1995) to network architectures which are commonly used for derivatives pricing problems. In contrast to the widely used automatic differentiation approach for calculating sensitivities of neural networks, the proposed approach requires only common matrix operations. Therefore, it is fast, easy to implement in practical applications and only requires dependence on the most basic mathematical libraries. Furthermore, the presented methods enable the simultaneous calculation of all first- and second-order price sensitivities of multiple instruments, making them particularly useful in risk monitoring of portfolios of derivatives.

Supporting this argument, Sect. 3 shows with numerical experiments that the methods outlined in Sect. 2 provide a significant performance improvement over the commonly used approach to calculate sensitivities of neural networks. In applications where a deep neural network is used for pricing, automatic differentiation would commonly be used to calculate these sensitivities with respect to the market parameters as the network's input features. Deep learning frameworks, such as *PyTorch* (Paszke et al. (2019)), provide direct access to this approach given its widespread use for network training. Therefore, the calculation speed of the proposed approach is benchmarked against a recent version of the automatic differentiation implementation in *PyTorch*. The performance comparison focuses on networks whose number of input parameters corresponds to that of commonly used derivatives pricing models.

At last, Sect. 4 demonstrates the practical applicability of the proposed sensitivity calculations with a case study. The basis of this case study is a multilayer feed-forward neural network, which has been trained to approximate the pricing function of European options under stochastic volatility. The weights and activation functions of this network are used in the methods from Sect. 2 to calculate the first- and second-order price sensitivities for a wide range of parameter combinations. The sensitivities of the network are shown to match the analytic values with a very high accuracy.

The derivations in Sect. 2 and the case study in Sect. 4 calculate the price sensitivities on a single position level. In case the sensitivities of a portfolio of instruments are of interest, the proposed approach can be used to calculate the sensitivities

of each instrument in isolation, followed by an aggregation step to portfolio level. This requires weighting the sensitivities of the instruments with the corresponding exposures.

The primary conclusion of this paper is that the potential of feed-forward neural networks in replacing traditional derivatives pricing methods goes beyond simply determining the fair prices of derivatives. The very same neural networks can be used to efficiently and accurately calculate the first- and higher-order sensitivities of the instrument's price, which make them appealing in performance critical real-time financial applications. They eliminate the performance bottleneck which is inherent in complex derivatives pricing models, making the dependence on sometimes unrealistic but computationally simple models obsolete.

2 Jacobian and Hessian matrices of the output of feed-forward neural networks

Multilayer feed-forward neural networks belong to the most elementary tools of deep learning. These networks take a set of input features and pass them through a chain of transformations to produce the network output. The transformation steps are most commonly organized as layers, and each layer usually consists of a linear combination of its inputs, followed by the application of a non-linear activation function. As the transformation steps are executed one after another, the output of one layer serves as the input for the following one. This ensures that the calculations *feed forward* by flowing from input features towards the output, without recurrence. The intermediate layers preceding the output of the network are commonly referred to as hidden layers.

Mathematically, let $N : \mathbb{R}^n \to \mathbb{R}^m$ be a feed-forward neural network with L hidden linear layers and corresponding activation functions. Given the (row) vector of input features x_{l-1}, the output of the l^{th} hidden layer, x_l, is a (row) vector given by

$$x_l = F_l(x_{l-1} W_l^T + b_l), \tag{1}$$

where W_l and b_l are the weights and bias of the l^{th} linear layer, respectively, and F_l is the corresponding activation function, applied element-wise to its input. As long as the selected activation functions are bounded and non-constant, such feed-forward network architectures are able to approximate functions and their derivatives with arbitrary accuracy (Hornik (1991)).

To achieve this approximation, the weights and biases of each layer need to be determined with the help of supervised learning. The process of finding the appropriate layer weights and biases is referred to as *network training*. For an extensive introduction to deep neural networks, as well as a detailed overview of the techniques commonly used for their training, see Goodfellow et al. (2016).

Once a feed-forward neural network has been trained to approximate a given function, the evaluation of its output for a set of input features is very efficient, as it usually only consists of vector-matrix multiplications and elementwise applications of activation functions on vectors. This makes feed-forward neural networks

particularly attractive in use cases where computation speed is of essence, but at the same time it is hindered by the use of computationally expensive functions. Replacing these functions with their neural network approximations can provide arbitrary accuracy, while significantly increasing the computation performance.

As shown in this section, the layer transformations can not only be used to calculate the network's output, but also can be directly applied for the calculation of the network's derivatives. The feed-forward nature of the transformations enables the use of the generalized chain rule, which reduces the computation of the network's derivatives to a series of elementary matrix operations. These operations allow for the calculation of the network derivatives with respect to all input features simultaneously, leading to the efficient calculation of the Jacobian and Hessian matrices.

Let x_0 be the n-vector of input features, and x_L the m-vector output of the network. The first-order derivatives of the network output with respect to the input features at $x = x_0$ are given by the network's Jacobian as

$$\mathbf{J}N(x)\big|_{x=x_0} = \begin{bmatrix} \frac{\delta N_1(x)}{\delta x_1} & \cdots & \frac{\delta N_1(x)}{\delta x_n} \\ \vdots & \ddots & \\ \frac{\delta N_m(x)}{\delta x_1} & & \frac{\delta N_m(x)}{\delta x_n} \end{bmatrix}_{x=x_0}.$$

The second-order derivatives of the output with respect to the input features at $x = x_0$ are expressed by an $n \times m \times n$ array of Hessians, whose j^{th} slice along the second dimension corresponds to the $n \times n$ Hessian of the j^{th} output variable with respect to the input features:

$$\mathbf{H}N_j(x)\big|_{x=x_0} = \begin{bmatrix} \frac{\delta^2 N_j(x)}{\delta x_1^2} & \cdots & \frac{\delta^2 N_j(x)}{\delta x_1 \delta x_n} \\ \vdots & \ddots & \\ \frac{\delta^2 N_j(x)}{\delta x_n \delta x_1} & & \frac{\delta^2 N_j(x)}{\delta x_n^2} \end{bmatrix}_{x=x_0}.$$

As long as the activation functions in each hidden layer l are twice differentiable, both the Jacobian and the Hessians at $x = x_0$ can be expressed in terms of the first- and second-order derivatives of the activation functions, evaluated at x_0, as well as the weights of the linear layers. Dimopoulos et al. (1995) derive the Jacobian of a network with non-scalar output as a matrix product, as well as the gradient and Hessian of a network with a single hidden layer and scalar output. The methods presented below extend these results and generalize them for networks with multiple hidden layers and non-scalar network outputs.

With this respect, the purpose of the following derivations overlaps with that of Laue et al. (2018), who develop a framework for the efficient calculation of higher-order derivatives of matrix and tensor expressions using automatic differentiation and Ricci calculus. At the same time, the results below are targeted specifically at feed-forward neural network architectures, and only rely on elementary matrix operations. As a consequence, they are concise, particularly easy to implement, and efficient to use in practical applications.

The following derivations calculate the Jacobian and array of Hessians of a network with L hidden layers and m output variables. The notation assumes that the output layer of the network corresponds to the L^{th} hidden layer.

Proposition 1 *The Jacobian of the output of a single layer with respect to its inputs is given by*

$$\frac{\delta x_l}{\delta x_{l-1}} = \left[\left(F_l'\right)^T J_l \right] \circ W_l, \tag{2}$$

where F_l' is the first derivative of the activation function in layer l, applied elementwise to the (row) vector $x_{l-1}W_l^T + b_l$, J_l is a row vector of ones, whose size corresponds to the size of the input vector x_{l-1}, and $A \circ B$ is the elementwise (Hadamard) product of matrices A and B of the same dimension.

Proof Using Expression (1), the q^{th} element of x_l, x_l^q is given by

$$\begin{aligned} x_l^q &= \left[F_l(x_{l-1}W_l^T + b_l) \right]_q \\ &= F_l(x_{l-1}\left[W_l^T\right]_q + b_l), \end{aligned} \tag{3}$$

where $\left[W_l^T\right]_q$ is the q^{th} column of the transposed weight matrix W_l^T. Differentiating (3) with respect to input feature p gives

$$\begin{aligned} \frac{\delta x_l^q}{\delta x_{l-1}^p} &= \left(F_l'\right)_q \left[W_l^T\right]_{p,q} \\ &= \left(F_l'\right)_q \left[W_l\right]_{q,p}, \end{aligned} \tag{4}$$

where $\left[W_l\right]_{q,p}$ is the $(q,p)^{th}$ element of the weight matrix W_l. Calculating each element of the Jacobian using Expression (4) yields

$$\begin{aligned} \frac{\delta x_l}{\delta x_{l-1}} &= \begin{bmatrix} \left(F_l'\right)_1 \left[W_l\right]_{1,1} & \cdots & \left(F_l'\right)_1 \left[W_l\right]_{1,n} \\ \vdots & \ddots & \\ \left(F_l'\right)_m \left[W_l\right]_{m,1} & & \left(F_l'\right)_m \left[W_l\right]_{m,n} \end{bmatrix} \\ &= \left[\left(F_l'\right)^T J_l \right] \circ W_l. \end{aligned} \tag{5}$$

□

Corollary 1 *The Jacobian of the entire network at $x = x_0$ is given by*

$$\mathbf{J}N(x)\big|_{x=x_0} = \prod_{l=0}^{\curvearrowright L-1} \left\{ \left[\left(F_{L-l}'\right)^T J_{L-l} \right] \circ W_{L-l} \right\}. \tag{6}$$

Corollary 1 is identical up to transposition to the results of Dimopoulos et al. (1995). The proof is straightforward using the chain rule and Proposition 1:

Proof Let x_0 and x_L be the vector of input features and outputs of network N. Then

$$\begin{aligned}\mathbf{J}N(x)\big|_{x=x_0} &= \frac{\delta x_L}{\delta x_0} \\ &= \frac{\delta x_L}{\delta x_{L-1}} \cdots \frac{\delta x_1}{\delta x_0} \\ &= \overset{\curvearrowright}{\prod_{l=0}^{L-1}} \frac{\delta x_{L-l}}{\delta x_{L-l-1}} \\ &= \overset{\curvearrowright}{\prod_{l=0}^{L-1}} \left\{ \left[\left(F'_{L-l}\right)^T J_{L-l} \right] \circ W_{L-l} \right\}.\end{aligned} \tag{7}$$

□

Corollary 1 can be used in a straightforward way to calculate the Jacobian of sub-networks:

Corollary 2 *The partial derivatives of the outputs of layer q with respect to the inputs of layer p, with $1 \leq p \leq L$ and $p \leq q \leq L$, are given by*

$$\mathbf{J}N^{p,q}(x)\big|_{x=x_0} = \overset{\curvearrowright}{\prod_{l=L-q}^{L-p}} \left\{ \left[\left(F'_{L-l}\right)^T J_{L-l} \right] \circ W_{L-l} \right\}. \tag{8}$$

Proof This follows from the proof of Corollary 1. □

For $1 \leq p \leq L$, define $\mathbf{J}N^{p,p-1}(x)\big|_{x=x_0} = I_{p-1}$ and $\mathbf{J}N^{L+1,L}(x)\big|_{x=x_0} = I_L$, where I_{p-1} and I_L are identity matrices, whose sizes are equal to the sizes of the input vector to layer p, x_{p-1}, and the output vector x_L, respectively. Otherwise, $\mathbf{J}N^{p,q}(x)$ is undefined.

The second-order derivatives of the network's output at $x = x_0$ are expressed as an $n \times m \times n$ array, whose (a, b, c) element corresponds to $\frac{\delta N_b(x)}{\delta x_a \delta x_c}$. The j^{th} slice of this array along the first dimension is given by

$$\mathbf{T}_j N(x)\big|_{x=x_0} = \begin{bmatrix} \frac{\delta^2 N_1(x)}{\delta x_1 \delta x_j} & \cdots & \frac{\delta^2 N_1(x)}{\delta x_n \delta x_j} \\ \vdots & \ddots & \\ \frac{\delta^2 N_m(x)}{\delta x_1 \delta x_j} & & \frac{\delta^2 N_m(x)}{\delta x_n \delta x_j} \end{bmatrix}_{x=x_0}. \tag{9}$$

Proposition 2 *The matrix of partial derivatives in Expression (9) can be expressed as*

$$\mathbf{T}_j N(x)\big|_{x=x_0} = \sum_{l=1}^{L} \boldsymbol{\Phi}_l^{Post} \boldsymbol{\Phi}_{l,j} \boldsymbol{\Phi}_l^{Pre}, \tag{10}$$

with

$$\begin{aligned}
\boldsymbol{\Phi}_l^{Post} &= \mathbf{J}N^{l+1,L}(x)\big|_{x=x_0}, \\
\boldsymbol{\Phi}_l^{Pre} &= \mathbf{J}N^{1,l-1}(x)\big|_{x=x_0}, \\
\boldsymbol{\Phi}_{l,j} &= \left\{ \left[\left(F_l''\right)^T \circ M_j \right] J_l \right\} \circ W_l,
\end{aligned}$$

where M_j is the j^{th} column of the matrix $M = W_l \boldsymbol{\Phi}_l^{Pre}$.

Proof Observe that $\mathbf{T}_j N(x)\big|_{x=x_0} = \frac{\delta}{\delta x_j} \mathbf{J}N(x)\big|_{x=x_0}$. Applying the generalization of the product rule to $\mathbf{J}N(x)\big|_{x=x_0}$, the form of Expression (10), as well as the definitions of $\boldsymbol{\Phi}_l^{Post}$ and $\boldsymbol{\Phi}_l^{Pre}$ are trivial.

The term $\boldsymbol{\Phi}_{l,j}$ stands for the differentiated term of the summand l:

$$\begin{aligned}
\boldsymbol{\Phi}_{l,j} &= \frac{\delta}{\delta x_j} \left\{ \left[\left(F_l'\right)^T J_l \right] \circ W_l \right\}_{x=x_0} \\
&= \left\{ \frac{\delta \left(F_l'\right)^T}{\delta x_j}\bigg|_{x=x_0} \right\} J_l \circ W_l \\
&= \left\{ \left[\left(F_l''\right)^T J_l \circ W_l \right] \overset{\curvearrowright}{\prod_{k=L-(l-1)}^{L-1}} \left\{ \left[\left(F_{L-k}'\right)^T J_{L-k} \right] \circ W_{L-k} \right\} \right\}_j J_l \circ W_l \\
&= \left\{ \left[\left(F_l''\right)^T J_l \circ W_l \right] \boldsymbol{\Phi}_l^{Pre} \right\}_j J_l \circ W_l \\
&= \left\{ \left[\left(F_l''\right)^T J_l \right]_j \circ \left[W_l \boldsymbol{\Phi}_l^{Pre} \right]_j \right\} J_l \circ W_l \\
&= \left\{ \left[\left(F_l''\right)^T \circ M_j \right] J_l \right\} \circ W_l.
\end{aligned} \tag{11}$$

□

As long as the activation functions in each hidden layer l of the network N are twice differentiable, and the first- and second-order derivatives are available in analytic form, Expressions (6) and (10) can be evaluated using a single forward pass on the network. This evaluation yields the neural network's representation of the originally approximated function's derivatives with respect to its input parameters. If the neural network's activation functions satisfy the smoothness requirements by Hornik (1991), these representations can be made arbitrarily accurate by training an appropriately wide and deep network architecture.

Using a piecewise linear activation function in some of the hidden layers should generally not impact the calculation of Expressions (6) and (10). For instance, even

though the derivatives of the *ReLU* activation function are not defined at $x = 0$, they are commonly treated as 0 by convention, which can also be applied in the calculations of F_l' and F_l'' above.

Nevertheless, the usage of piecewise linear activation functions might impact the abilities of the neural network to accurately approximate the derivatives of the function. This is easy to recognize when using Expression (10) to calculate the Hessian of a neural network with *ReLU* activation functions in each layer. Treating the second derivative of *ReLU* at $x = 0$ as zero by convention, the second derivative of the activation function is zero for every input. Therefore, in the network under consideration $\boldsymbol{\Phi}_{l,j}$ from Expression (10) is a zero matrix for every layer l, and consequently the Hessian of the network is also zero for every input, irrespective of the function approximated by the network.

3 Performance comparison with automatic differentiation

Automatic differentiation and the derivations in Sect. 2 can both be used to calculate the analytic derivatives of feed-forward neural networks. However, the two approaches arrive at the derivatives in largely different ways, making them appealing for different use cases. As automatic differentiation most frequently represents the mathematical expressions as directed graphs, it keeps track of the operations performed at every node, as well as their derivatives. Therefore, it can be conveniently used in applications where the derivatives at every network node need to be calculated, such as during network training. On the contrary, the approach in Sect. 2 focuses purely on calculating the derivatives of the network output with respect to its input parameters. As a consequence, it is appealing to use with calibrated neural networks in applications where the derivatives of the approximated function are of interest. The dependence on only elementary matrix operations make this method easy to implement and fast to evaluate and therefore useful in performance critical calculations. This section focuses on the performance aspect of calculating the Jacobian and Hessians of the network outputs with respect to its input features, and shows the improvements of computation time from using Expressions (6) and (10) over the automatic differentiation approach for selected network architectures.

Table 1 compares the median computation time of the network Jacobian using Corollary 1 with the standard autograd implementation in *PyTorch* v.1.7.0 (Paszke et al. (2019)). These median computation times (in milliseconds) are calculated on the CPU based on 10 000 random initializations for each of a variety of network sizes (I, H, O), where $I \in \{8, 9, 11, 14, 19\}$ is the size of the input feature vector, $O \in \{4, 16\}$ is the size of the network output vector, and $H \in \{32, 64, 128, 256\}$ is the size of the output vectors of the four fully connected internal hidden layers. The activation functions in the four internal and one output layers are *sigmoid*, *tanh*, *sigmoid*, *tanh* and *sigmoid*, respectively.

The selected sizes of the input feature vector correspond to the number of parameters in widely used financial mathematical models for derivatives pricing. 8 and 9 parameters are frequently used in the Heston model (Heston (1993)), without and including a continuous dividend yield, respectively. Different variations of stochastic

Table 1 Median Jacobian computation times (in milliseconds) on the CPU for selected network architectures using *PyTorch* v.1.7.0 (*PT*) and Expression (6)

In I	Out O	Internal Layer Output Sizes H							
		32		64		128		256	
		PT	(6)	*PT*	(6)	*PT*	(6)	*PT*	(6)
8	4	1.420	0.797	1.441	0.802	1.933	1.136	2.123	1.517
	16	4.520	0.860	4.494	0.974	6.255	1.217	7.254	1.610
9	4	1.720	0.948	1.547	0.880	2.242	1.282	2.489	1.639
	16	4.401	0.826	4.500	0.995	6.655	1.243	7.612	1.712
11	4	1.516	0.822	1.490	0.812	2.034	1.248	2.331	1.621
	16	4.497	0.849	4.616	0.994	6.755	1.262	7.725	1.690
14	4	1.451	0.842	1.493	0.854	2.066	1.234	2.338	1.618
	16	4.497	0.856	4.569	0.994	6.578	1.247	7.624	1.670
19	4	1.482	0.828	1.491	0.846	2.064	1.226	2.360	1.638
	16	4.612	0.833	4.578	1.000	6.671	1.258	7.773	1.680

I represents the number of the network's input features. O shows the size of the network's output vector. H stands for the size of the output vectors of the fully connected internal hidden layers

volatility models with jumps in the underlying and the volatility processes commonly use 11, 14 or 19 input parameters (e.g., Duffie et al. (2000)).

Analogous to the comparisons in Table 1, Table 2 compares the median CPU computation times of the network Hessians using Proposition 2 with the standard *PyTorch* autograd implementation. The calculations for Proposition 2 leverage the efficient tensor operations implemented in *PyTorch* and calculate all slices of the array of Hessians simultaneously.

Given that the Hessian calculation using the *PyTorch* autograd method is limited to scalar functions, only $O = 1$ is used for the comparison. Further, the networks in the Hessian computations contain only two internal hidden layers, with activation

Table 2 Median Hessian computation times (in milliseconds) on the CPU for selected network architectures using *PyTorch* v.1.7.0 (*PT*) and Expression (10)

In I	Out O	Internal Layer Output Sizes H							
		32		64		128		256	
		PT	(10)	*PT*	(10)	*PT*	(10)	*PT*	(10)
8	1	3.481	1.887	3.623	1.979	4.893	2.265	5.298	3.190
9	1	3.897	1.888	3.994	1.979	5.394	2.441	5.953	3.349
11	1	4.653	1.938	4.661	2.017	6.685	2.444	6.997	3.529
14	1	5.777	1.966	5.811	2.003	8.053	2.806	8.972	3.914
19	1	7.644	1.972	7.636	2.081	11.005	2.948	11.696	4.248

I represents the number of the network's input features. O shows the size of the network's output vector. H stands for the size of the output vectors of the fully connected internal hidden layers

Table 3 Median Jacobian computation times (in milliseconds) on the GPU for selected network architectures using *PyTorch* v.1.7.0 (*PT*) and Expression (6)

In I	Out O	Internal Layer Output Sizes H							
		32		64		128		256	
		PT	(6)	*PT*	(6)	*PT*	(6)	*PT*	(6)
8	4	4.852	2.025	4.833	1.980	4.803	1.897	4.322	2.113
	16	14.134	1.654	13.096	1.521	13.718	1.778	14.483	2.102
9	4	4.106	1.759	4.304	1.795	4.312	2.003	4.395	2.272
	16	14.268	1.778	14.496	1.813	13.660	1.320	14.386	2.188
11	4	4.112	1.763	4.169	1.856	4.328	1.896	4.693	2.133
	16	15.587	1.804	14.442	1.766	13.638	2.020	14.490	2.209
14	4	4.289	1.903	4.158	1.771	4.187	1.794	4.283	2.277
	16	14.717	1.751	14.283	1.834	13.965	1.931	15.073	2.288
19	4	4.387	1.801	4.373	1.832	4.491	1.893	4.772	2.191
	16	14.562	1.740	13.780	1.204	14.841	1.532	15.912	2.334

I represents the number of the network's input features. O shows the size of the network's output vector. H stands for the size of the output vectors of the fully connected internal hidden layers

Table 4 Median Hessian computation times (in milliseconds) on the GPU for selected network architectures using Expression (10) and *PyTorch* v.1.7.0 (*PT*)

In I	Out O	Internal Layer Output Sizes H							
		32		64		128		256	
		PT	(10)	*PT*	(10)	*PT*	(10)	*PT*	(10)
8	1	10.875	5.793	10.633	5.408	10.929	6.079	11.112	6.790
9	1	12.641	5.441	11.699	5.099	12.075	5.754	11.154	6.396
11	1	14.252	5.982	14.847	5.758	15.267	5.420	13.503	6.976
14	1	16.735	5.806	18.620	5.720	18.445	5.564	17.847	6.066
19	1	22.692	5.841	23.163	6.099	22.632	6.146	22.908	5.930

I represents the number of the network's input features. O shows the size of the network's output vector. H stands for the size of the output vectors of the fully connected internal hidden layers

functions *sigmoid* and *tanh*, respectively. The activation function in the output layer is *sigmoid*.

While Tables 1 and 2 present the performance comparisons on the CPU, Tables 3 and 4 show the results for the respective comparisons on the GPU.

The calculations for the performance comparisons in Tables 1, 2, 3 and 4 were run on a commercial notebook with Intel Core i7 CPU (2.8GHz), 32GB of memory and NVIDIA GeForce GTX 1050 Ti GPU, running Windows Subsystem for Linux with Ubuntu 20.04 Linux distribution. A reference implementation for the performance comparisons and the application of Expressions (6) and (10) can be found under https://github.com/antalratku/nn_deriv.git.

Given that both automatic differentiation as well as Expressions (6) and (10) calculate the analytic derivatives of the network, their output should be theoretically identical. However, due to the rounding errors arising from machine precision, one could expect to observe very slight differences. In the calculations above, the Jacobian and Hessian values calculated with Expressions (6) and (10) are equal to those calculated with *PyTorch* up to an absolute precision of 1.4e-9.

The analysis in this section refrains from using the finite differences methods for the calculation of the network sensitivities due to two practical reasons. Firstly, while both of the presented methods give the exact derivative of the neural network, the finite differences method only approximates it with the help an arbitrarily chosen step size ϵ. This can lead to significant numerical instabilities, especially in case of the second-order network derivatives. Secondly, to calculate the network sensitivities with respect to *all* input features, the finite differences method requires at least two forward-passes on the network for each input parameter, making it computationally expensive for networks with many input features.

The comparisons in Tables 1, 2, 3, 4 show that Expressions (6) and (10) can significantly speed up the sensitivity calculations compared to the automatic differentiation approach. The performance benefits are the most obvious for the Hessian calculations, where using Expression (10) yields very consistent computation times across input feature counts. A further benefit of using Expressions (6) and (10) is that they are not restricted to scalar functions, but are applicable to a generic deep feed-forward neural network $N : \mathbb{R}^n \to \mathbb{R}^m$ with twice differentiable activation functions.

Comparing the performance figures across the CPU and the GPU makes it apparent that the GPU does not provide a clear performance benefit over the CPU for the selected network architectures. This can be explained on the one hand by the network dimensions, and on the other hand by the performance comparison methodology. First, the networks have relatively few input features, and the dimensions of their hidden layers are rather small. While these network dimensions are generally sufficient to accurately approximate derivatives pricing functions, they do not justify the need for large-scale parallelization provided by the GPU. Second, during each of the 10 000 iterations of the performance comparison only one realization of the input vector was fed into the neural network. This approach is largely consistent with derivatives pricing models, where one common set of model parameters is used for determining the price of multiple instruments. Therefore, the practical requirements somewhat counteract the parallelization capabilities of the GPUs, which usually excel at operating on large *batches* of input data, consisting of multiple, independent input vectors.

4 Case study of numerical accuracy

The efficient calculation of the price sensitivities of financial derivative instruments is essential in applications such as market risk monitoring, the continuous supervision of trading limits, large-scale scenario analysis of a portfolio of derivatives, or the implementation of quantitative trading strategies. These sensitivities not only

help the trader identify potentially unwanted sources of market risk, but can also provide an estimate of how the market risk exposure would change with regards to changes in market parameters. Only with the help of such information can the trader decide, how and when to hedge a certain exposure to market factors.

Traditionally, when complex financial models are applied to price derivative instruments, the sensitivity calculations can pose a computational bottleneck for large trading books. Therefore, often rather simple pricing models are preferred for such calculations, even though they might not capture the dynamics and stylized facts of the market factors accurately. This dependence on overly simplistic models can be significantly reduced by approximating more complex ones with deep feed-forward neural networks. By relying on the approximation capabilities and evaluation simplicity of these networks, one can largely eliminate the computational burden of the more complex pricing models that use Monte Carlo simulations or numerical integration, while maintaining a very high pricing accuracy. As outlined in Section 2, the usability of this pricing approach can be further enhanced by combining the feed-forward neural network with Eqs. 6) and (10) for efficient price sensitivity calculations.

The following case study demonstrates how accurately a feed-forward neural network can approximate the price sensitivities of European options, while providing the performance improvements described in Sec. 3.

4.1 Accuracy of sensitivity approximations

As the basis of the case study, a deep feed-forward neural network is trained to approximate the pricing function of the Heston model (Heston (1993)) for a European call option. The output of the network is the option premium, expressed as a percentage of the underlying value S. The input features to the network are the five parameters of the Heston model that describe the dynamics of the volatility, $\{\kappa, \theta, \sigma, \rho, v_0\}$, the continuously compounded risk-free and dividend rates, r and d, respectively, the remaining time-to-expiry τ of the option, as well as its spot moneyness m. The spot moneyness of a call option with strike price K and underlying value S is defined as $m = \frac{K}{S}$. Accordingly, in the money call options with strike price $K < S$ have a spot moneyness below 1, while out of the money call options have a spot moneyness above 1.

As shown in Sect. 2, the same neural network that has been trained to approximate the price of a call option given some input parameters can also be used to calculate the sensitivities of the option price with respect to these parameters. It is worth noting that, similarly to the approximated option price, the price sensitivities calculated from the neural network are also only approximations of those from the original pricing model. However, the accuracy of these approximations can be largely controlled for by selecting an appropriate network architecture and training method.

The neural network selected for the following demonstration contains three internal hidden layers of 128 nodes each, with activation functions *tanh*, *sigmoid*, *tanh*, respectively. The output layer contains a single node with a *sigmoid*

Table 5 Lower and upper bounds for the uniformly sampled input parameters

	κ	θ	σ	ρ	v_0	m	τ	r	d
Lower bound	0.1	0.01	0.1	−1.0	0.0015	0.1	0.0027	−0.1	0.0
Upper bound	10.0	1.0	2.0	1.0	1.5	2.4	3.0	0.1	0.2

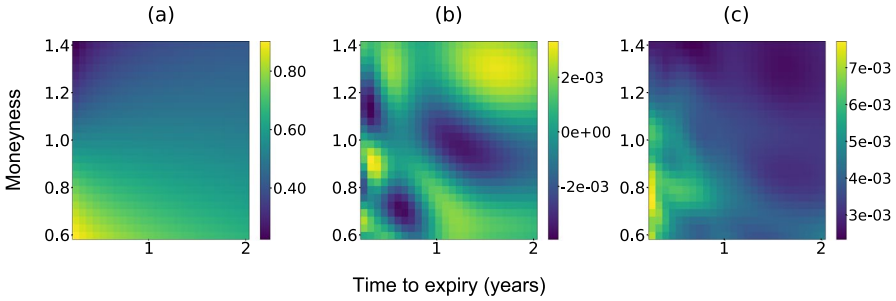

Fig. 1 Delta approximation. **a** average Heston deltas. **b** average difference between the neural network deltas and the analytic deltas. **c** standard deviation of the differences between the neural network deltas and the analytic deltas

activation function. The network is trained on 16 million randomly selected parameter combinations as inputs. Each parameter is sampled from a uniform distribution with lower and upper bounds as presented in Table 5, and for each parameter combination the satisfaction of the Feller condition is ensured. The targets are the call option premia corresponding to each parameter combination, calculated with the Heston pricing function.

Following the network training step, 10 000 combinations of Heston parameters as well as risk-free and dividend rates are selected at random. The parameters are sampled uniformly, with the same lower and upper bounds as in Table 5, and ensuring the satisfaction of the Feller condition. For each parameter combination, the first- and second-order sensitivities are calculated using Expressions (6) and (10) for a range of options with spot moneyness m spanning from 0.6 to 1.4, and remaining time-to-expiration τ from 0.25 to 2 years.

Figures 1, 2, 3 and 4 compare the sensitivities calculated from the trained neural network using Expressions (6) and (10) with the analytic sensitivities for the same parameter combinations and same options. For the calculation of the analytic sensitivities, see Rouah (2013).

In each of the figures, subplot (a) presents the average Heston sensitivities by moneyness and time-to-expiry with respect to selected market factors over the 10 000 random parameter combinations. Subplot (b) shows the average difference between the Heston sensitivities calculated from the neural network and the analytic sensitivities. At last, subplot (c) presents the standard deviation of the differences between the Heston sensitivities calculated from the neural network and their analytic counterparts by moneyness and time-to-expiry.

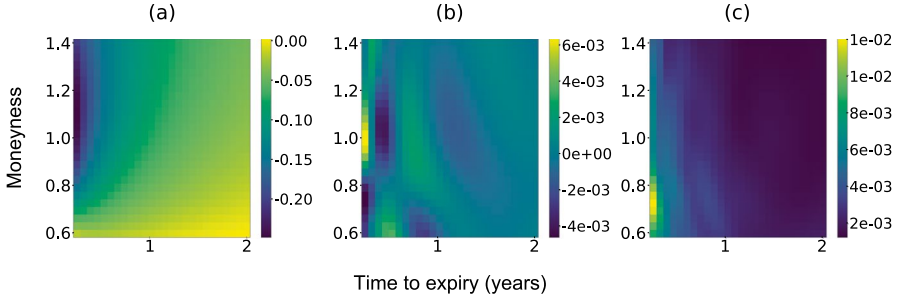

Fig. 2 Theta approximation. **a** average Heston thetas. **b** average difference between the neural network thetas and the analytic thetas. **c** standard deviation of the differences between the neural network thetas and the analytic thetas

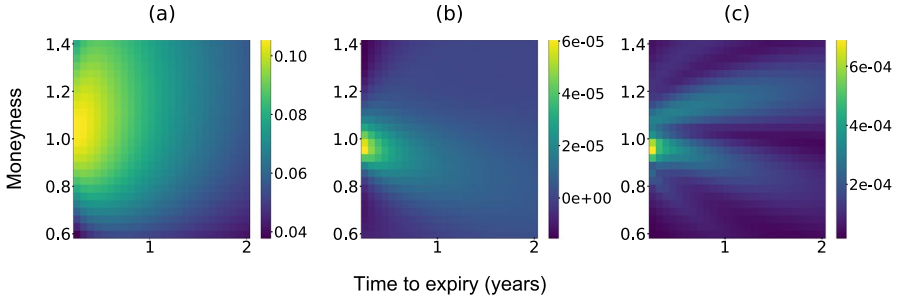

Fig. 3 Vega approximation. **a** average Heston vegas with respect to v_0. **b** average difference between the neural network vegas and the analytic vegas. **c** standard deviation of the differences between the neural network vegas and the analytic vegas

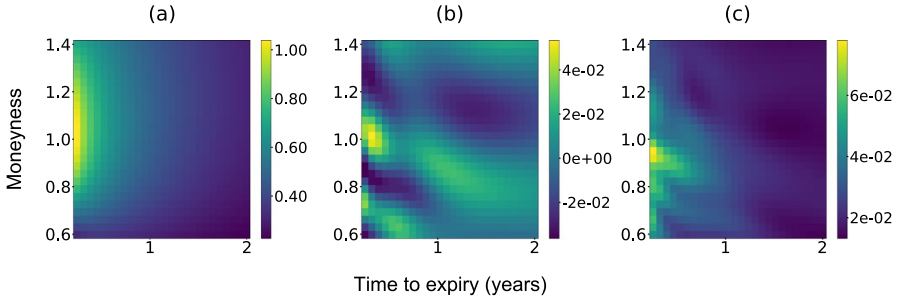

Fig. 4 Gamma approximation. **a** average Heston gammas. **b** average difference between the neural network gammas and the analytic gammas. **c** standard deviation of the differences between the neural network gammas and the analytic gammas

Figure 1 presents the approximation results for the option delta. As the delta of a call option shows how the price of the option would change for a small move in the option's underlying, it is often considered one of the most important first-order sensitivities of the option price. For instance, an option delta of 0.3 means

that a 1 unit increase in the underlying's value would lead to an approximately 0.3 unit increase in the option's price.

Subplot (a) of Figure 1 shows the average call option deltas by time-to-expiry and moneyness over the 10 000 randomly selected Heston parameter combinations. As shown on this subplot by the yellow area, deep in the money call options with short time-to-expiry behave very much like the underlying itself, and therefore have a delta close to 1. On the other hand, deep out-of-the-money call options with short time-to-expiry have a lower probability of becoming in-the-money at expiration, and therefore react with much less sensitivity to moves in the underlying instrument, resulting in low delta values.

Subplot (b) of the same figure presents the average difference between the option deltas calculated using the neural network and the analytic option deltas, over the 10 000 randomly selected Heston parameter combinations. The average differences between the two sensitivity calculation methods for most combinations of time-to-expiry and moneyness are below 0.003 in absolute terms, which demonstrates the very high accuracy of the network approximations. This is further supported by subplot (c) of Fig.1, which shows that not only the average of the delta differences, but also their standard deviations are generally very low. This suggests that the neural network can accurately approximate the semi-analytic pricing function's derivatives for a wide range of input parameters, and therefore generalizes well for the Heston model. The highest standard deviations of the delta differences are shown for the in-the-money call options with very short time-to-expiry, which is consistent with the high level of the option delta itself for these combinations of moneyness and time-to-expiry.

Figure 2 compares the theta of the option calculated from the neural network with the analytic theta. The theta of an option shows its first-order price sensitivity with respect to the passage of time, and the corresponding shortening of the option's time-to-expiry. While subplot (a) of Fig. 2 shows the well-known pattern that out-of-the-money call options with very short time-to-expiry are the most exposed to a loss of value due to the passage of time, subplots (b) and (c) demonstrate that the theta approximations with neural networks are highly accurate across all time-to-expiry and moneyness combinations.

Similarly, accurate approximations of the option vega are shown on Fig. 3. In this analysis, the vega of the option is defined with respect to the Heston parameter v_0, and it represents the sensitivity of the option price with respect to changes in the instantaneous volatility.

While Figs. 1, 2 and 3 focus on first-order price sensitivity approximations, Figure 4 presents the same analysis for the option gamma, which is the second-order option price sensitivity with respect to changes in the underlying. As a consequence, gamma can also be interpreted as the sensitivity of the option delta with respect to changes in the underlying, and therefore it is often used to judge how frequently a trader would need to adjust an option portfolio to keep its overall delta close to a target level. In general, at-the-money options with short time-to-expiry show the highest gammas, which is also demonstrated in subplot (a) of Fig. 4. Subplots (b) and (c) present the means and the standard deviations of the differences between the option gammas calculated using the neural network,

as well as the analytically calculated gammas over the 10 000 randomly selected Heston parameter combinations. Both of these subplots highlight how accurately a neural network is able to approximate even higher-order price sensitivities.

The analyses presented in Figs. 1, 2, 3 and 4 underscore the applicability of feed-forward neural networks for the efficient and accurate calculation of option price sensitivities. Even though the neural network used for these examples was trained to approximate only the pricing function of the option, the same network can also be used to accurately approximate the first- and higher-order derivatives of this function. Combining these approximation capabilities with the performance benefits from using Expressions (6) and (10) can enable the use of realistic but complex derivatives pricing models in performance critical applications, by eliminating the computational bottleneck inherent in them.

5 Conclusion

This article explores the use of deep feed-forward neural networks for efficiently calculating price sensitivities of financial derivatives. The analytic results proposed for this purpose are straightforward to implement for a network that has been trained to approximate a derivatives pricing function.

Besides providing very accurate approximations of the price sensitivities with respect to market factors, the proposed approach also delivers all the first- and second-order sensitivities simultaneously within milliseconds. The approach is even faster than a recent implementation of automatic differentiation, which is demonstrated by numerical experiments for calculating the Jacobian and Hessian matrices of different network architectures.

The approximation accuracy and high performance evaluation make the proposed approach particularly appealing in time-critical use-cases, such as real-time risk monitoring of derivatives portfolios or high-frequency trading strategies. Future research could apply the proposed sensitivity calculations not only to managing the risk of derivatives, but also to improve the calibration of financial mathematical models to observed market prices of derivatives. Advances in this area would enable trading firms to determine the fair value of financial instruments in a more accurate way and thereby contribute to the overall efficiency of capital markets.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of interest The authors declare that they have no conflict of interest. The authors did not receive support from any organization for the submitted work. Opinions expressed in this paper are those of the authors. Antal Ratku is employed by Allianz Global Investors GmbH and receives no financial or non-financial benefits form his employer for the publication of this manuscript. The code used for the analysis in Sect. 3 is available in Jupyter Notebook format under Antal Ratku's public GitHub repository https://github.com/antalratku/nn_deriv.

References

Anders U, Korn O, Schmitt C (1998) Improving the pricing of options: a neural network approach. J forecast. 17(5–6):369–388

Buehler H, Gonon L, Teichmann J, Wood B (2019) Deep hedging. Quantit. Fin. 19(8):1271–1291. https://doi.org/10.1080/14697688.2019.1571683

Dimopoulos Y, Bourret P, Lek S (1995) Use of some sensitivity criteria for choosing networks with good generalization ability. Neural Process Lett. 2(6):1–4. https://doi.org/10.1007/BF02309007

Duffie D, Pan J, Singleton K (2000) Transform analysis and asset pricing for affine jump-diffusions. Econometrica 68(6):1343–1376. https://doi.org/10.1111/1468-0262.00164

Gaspar RM, Lopes SD, Sequeira B (2020) Neural network pricing of american put options. Risks 8(3):73. https://doi.org/10.3390/risks8030073

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MIT press

Heston SL (1993) A closed-form solution for options with stochastic volatility with applications to bond and currency options. Rev Fin Stud. 6(2):327–343. https://doi.org/10.1093/rfs/6.2.327

Hornik K (1991) Approximation capabilities of multilayer feedforward networks. Neural Netw. 4(2):251–257

Hornik K, Stinchcombe M, White H (1989) Multilayer feedforward networks are universal approximators. Neural netw. 2(5):359–366. https://doi.org/10.1016/0893-6080(89)90020-8

Hornik K, Stinchcombe M, White H (1990) Universal approximation of an unknown mapping and its derivatives using multilayer feedforward networks. Neural Netw. 3(5):551–560. https://doi.org/10.1016/0893-6080(90)90005-6

Horvath B, Muguruza A, Tomas M (2020) Deep learning volatility: a deep neural network perspective on pricing and calibration in (rough) volatility models. Quantit. Fin. 21(1):11–27

Hutchinson JM, Lo AW, Poggio T (1994) A nonparametric approach to pricing and hedging derivative securities via learning networks. The J Fin. 49(3):851–889. https://doi.org/10.1111/j.1540-6261.1994.tb00081.x

Kolm PN, Ritter G (2019) Dynamic replication and hedging: a reinforcement learning approach. The J Financ Data Sci. 1(1):159–171. https://doi.org/10.3905/jfds.2019.1.1.159

Laue S, Mitterreiter M, Giesen J (2018) Computing higher order derivatives of matrix and tensor expressions. In: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (eds.) Advances in Neural Information Processing Systems 31, pp. 2750–2759. Curran Associates, Inc. http://papers.nips.cc/paper/7540-computing-higher-order-derivatives-of-matrix-and-tensor-expressions.pdf

Liu S, Oosterlee CW, Bohte SM (2019) Pricing options and computing implied volatilities using neural networks. Risks 7(1):16. https://doi.org/10.3390/risks7010016

Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, Killeen T, Lin Z, Gimelshein N, Antiga L, Desmaison A, Kopf A, Yang E, DeVito Z, Raison M, Tejani A, Chilamkurthy S, Steiner B, Fang L, Bai J, Chintala S (2019) Pytorch: An imperative style, high-performance deep learning library. In: H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlche-Buc, E. Fox, R. Garnett (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019). http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

Rouah FD (2013) The Heston Model and Its Extensions in Matlab and C. John Wiley & Sons

Ruf J, Wang W (2020) Neural networks for option pricing and hedging: a literature review. J Comput Finan, Forthcom. https://doi.org/10.21314/jcf.2020.390