Differential learning methods for solving fully nonlinear PDEs

William Lefebvre1 · Grégoire Loeper2 · Huyên Pham3

Received: 28 May 2022 / Accepted: 27 February 2023 / Published online: 15 March 2023 © The Author(s), under exclusive licence to Springer Nature Switzerland AG 2023

Abstract
We propose machine learning methods for solving fully nonlinear partial differential equations (PDEs) with convex Hamiltonian. Our algorithms are conducted in two steps. First, the PDE is rewritten in its dual stochastic control representation form, and the corresponding optimal feedback control is estimated using a neural network. Next, three different methods are presented to approximate the associated value function, i.e., the solution of the initial PDE, on the entire space-time domain of interest. The proposed deep learning algorithms rely on various loss functions obtained either from regression or pathwise versions of the martingale representation and its differential relation, and compute simultaneously the solution and its derivatives. Compared to existing methods, the addition of a differential loss function associated with the gradient, and augmented training sets with Malliavin derivatives of the forward process, yields a better estimation of the PDE’s solution derivatives, in particular of the second derivative, which is usually difficult to approximate. Furthermore, we leverage our methods to design algorithms for solving families of PDEs when varying terminal condition (e.g., option payoff in the context of mathematical finance) by means of

This work benefits from the financial support of the Chaire Futures of Quantitative Finance.

1 BNP Paribas Global Markets, Laboratoire de Probabilités, Statistique et Modélisation (LPSM, UMR CNRS 8001), Université Paris Cité and Sorbonne Université, Building Sophie Germain, Avenue de France, 75013 Paris, France
2 BNP Paribas Global Markets, School of Mathematics, Monash University, Clayton Campus, Melbourne, VIC 3800, Australia
3 Laboratoire de Probabilités, Statistique et Modélisation (LPSM, UMR CNRS 8001), Université Paris Cité and Sorbonne Université, Building Sophie Germain, Avenue de France, 75013, Paris Paris, France
the class of DeepOnet neural networks aiming to approximate functional operators. Numerical tests illustrate the accuracy of our methods on the resolution of a fully nonlinear PDE associated with the pricing of options with linear market impact, and on the Merton portfolio selection problem.

**Keywords** Fully nonlinear PDEs · Deep learning · Differential learning · Option pricing with market impact

**JEL Codes** C45 · C63 · G11

1 Introduction

This paper is devoted to the resolution of fully nonlinear partial differential equations (PDEs) of the form

\[
\begin{cases}
\frac{\partial u}{\partial t} + H(x, D_x u, D^2_x u) = 0, & (t, x) \in [0, T) \times \mathbb{R}^d, \\
u(T, x) &= g(x), & x \in \mathbb{R}^d,
\end{cases}
\]  

(1.1)

where the Hamiltonian \( H : \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^{d \times d} \to \mathbb{R} \cup \{\infty\} \) is a lower semi-continuous convex function with respect to the two last arguments \((z, \gamma)\), and \( g \) a measurable function on \( \mathbb{R}^d \). The numerical resolution of this class of PDE is a notorious challenging problem, and it is especially difficult to obtain a good approximation of the second spatial derivative \( D^2_x u \) of the solution in this fully nonlinear context. In the last years, significant progress has been achieved towards these challenges with several numerical methods using techniques from deep learning; see the recent surveys by Beck et al. (2020), Germain et al. (2021), and Hu and Laurière (2022): A first class of approximation algorithms, called *Physics Informed Neural Network* (PINNs) (Raissi et al., 2019), also known as *Deep Galerkin method* (DGM) (Sirignano & Spiliopoulos, 2018), directly approximates the solution to the PDE by a neural network, and its partial derivatives by automatic differentiation, by minimizing the loss function arising from the residual of the PDE evaluated on a random grid in the space-time domain. A second class of algorithms relies on the backward stochastic differential representation of the PDE in the semi-linear case by minimizing either a global loss function (see Weinan et al., 2017, and extensions in Beck et al., 2019, Ji et al., 2020, Nüskens et al., 2021), or sequence of loss functions from backward recursion (see Huré et al., 2020, and variations–extensions in Pham et al., 2021, Beck et al., 2021, Germain et al., 2021).

In this article, we consider numerical methods for fully nonlinear PDEs based on machine learning techniques that are conducted in two steps. The starting point of our approach is to rewrite the PDE (1.1) with convex Hamiltonian in its stochastic control representation form following the duality arguments of Mete Soner et al. (2013). An approximation of the associated optimal feedback control is then obtained using a neural network by a global optimization, as described in Han and Weinan (2016) and Gobet and Munos (2005). Based on this control approximation, two main approaches using neural networks are then developed to approximate the value function, hence
the solution of the initial PDE on the space-time domain, which is formulated as a conditional expectation with respect to the approximate optimal state process.

The first one, called Differential regression learning, is inspired by Huge and Savine (2020). In this paper, the authors compute conditional expectations of an option payoff in the spirit of the Longstaff–Schwartz method (Longstaff & Schwartz, 2001), by parametrizing it with a neural network and performing the regression simultaneously on the value and on the derivative of this neural network. The addition of a regression loss on the derivative, where the derivative of the network, computed by automatic differentiation, is regressed against the pathwise derivative of the conditional expectation integrand, improves the estimation of the first derivative of the conditional expectation and empirically speeds up the training by allowing to train the network on smaller batches. We adapt this method to our context. Indeed, having approximated the optimal control of the stochastic control problem associated with the PDE, the associated value function can be expressed as the conditional expectation of a running payoff of optimally controlled state trajectories, while its gradient is represented also as a conditional expectation formula by differentiation of the payoff. This representation formulae provide two loss functions that will be minimized alternately to learn by neural network approximation both the solution of the PDE and its gradient.

The second approach, called Pathwise learning, is inspired by Vidales et al. (2018), where the authors compute the conditional expectation of a payoff using the Feynman–Kac formula to derive a pathwise control variate corresponding to the hedging strategy. In their work, the derivative of the conditional expectation value, present in the hedging integral, is approximated by a neural network and optimized so as to minimize the variance of the conditional expectation estimator. This approach is analogous to the one derived in Potters et al. (2001), with the addition of machine learning techniques. In our case, given the approximation of the stochastic control associated to the PDE, a martingale representation of the payoff is derived on optimally controlled trajectories. Our first pathwise method, called Pathwise martingale learning, uses this martingale representation to train the value and the first derivative of a neural network. This method is also in the spirit of the deep BSDE method of Beck et al. (2020), but the minimization of our loss function provides directly an approximation of the solution and its gradient on the space-time domain. Our second method, called Pathwise differential learning, considers furthermore the derivative of this martingale representation, computed by automatic differentiation, which gives another loss function to be minimized to train neural networks for approximating the value function and its first and second derivatives. Such differential representation has been also considered in the recent paper (Negyesi et al., 2021) for designing a deep learning scheme with one-step loss functions as in the deep backward approach in Huré et al. (2020) for solving forward backward SDEs with new estimation and error control of the \( Z \) process. Actually, the addition of this derivative loss function permits a better approximation of the terminal condition of the PDE and improves the overall approximation of the PDE solution’s value and derivatives on the entire domain.

Finally, we leverage our deep learning algorithms for solving families of PDEs when varying the terminal condition. In other words, the input is a function \( g_K \) with parameter \( K \), and the output is the solution to the PDE with terminal condition \( g_K \). This is performed by means of the class of the DeepOnet neural networks aiming to
approximate functional operators. These networks, introduced in Lu et al. (2019), rely on a universal approximation theorem for operators (Chen & Chen, 1995), stating that a neural network with a single hidden layer can approximate accurately any nonlinear continuous operator. The DeepOnet can then be used to learn the mapping between the terminal function of a PDE and its solution.

The outline of the paper is organized as follows. In Sect. 2, we present the problem, recall the dual stochastic control representation of fully nonlinear PDEs and outlines the different methods. In Sect. 3, we present the theory of the differential regression learning method, give the expression of the losses used to train the neural network, and present the advantages of adding a loss to train the first derivative of the neural network. In Sect. 4, the Pathwise and Pathwise differential methods are developed and the expressions of the losses used to train the neural network are given. In Sect. 5, the implementation details and pseudo-codes of the different algorithms are presented along with validation tests, and we present in Sect. 6 numerical results of the three methods on portfolio selection problem (in the Merton case and in multivariate stochastic volatility models), and on the Black–Scholes with linear market impact PDE. Section 7 presents a method to solve nonlinear parabolic PDEs with parametric terminal condition $g_K$ for parameter values $K$ in a compact set. The codes of our algorithms are available on https://colab.research.google.com/drive/1xyE1U3SqiN4Hjia2d3pOsCXCDCGWRUqsH?usp=sharing. Finally, we conclude in Sect. 8 with a sum-up of our findings, a comparison of the different algorithms, and some suggestions for further developments.

Notations. We end this introduction with some notations that will be used in the sequel of the paper. The scalar product between two vectors $b$ and $z$ is denoted by $b \cdot z$, and $|\cdot|$ is the Euclidean norm. Given two matrices $A = (A_{ij})$ and $B = (B_{ij})$, we denote by $A : B = \text{Tr}(A^\top B) = \sum_{i,j} A_{ij} B_{ij}$ its inner product, and by $|A|$ the Frobenius norm of $A$. Here, $\top$ is the transpose matrix operator. $S^d$ is the set of $d \times d$ symmetric matrices with real coefficients equipped with the partial order: $\gamma_1 \leq \gamma_2$ iff $\gamma_2 - \gamma_1 \in S^d_+$, the set of positive semidefinite matrices in $S^d$.

Let $M = (M_{i_1 i_2 i_3}) \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ be a tensor of order 3. For $p = 1, 2, 3$, the $p$-mode product of $M$ with a vector $b = (b_i) \in \mathbb{R}^{d_p}$, is denoted by $M \bullet_p b$, and it is a tensor of order 2, i.e., a matrix defined elementwise as

\[
(M \bullet_1 b)_{i_2 i_3} = \sum_{i_1=1}^{d_1} M_{i_1 i_2 i_3} b_{i}, \quad (M \bullet_2 b)_{i_1 i_3} = \sum_{i_2=1}^{d_2} M_{i_1 i_2 i_3} b_{i_2}, \quad (M \bullet_3 b)_{i_1 i_2} = \sum_{i_3=1}^{d_3} M_{i_1 i_2 i_3} b_{i_3}.
\]

The $p$-mode product of a 3-rd order tensor $M \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ with a matrix $B = (B_{ij}) \in \mathbb{R}^{d_p \times d}$, also denoted by $M \bullet_p B$, is a third-order tensor defined elementwise as

\[
(M \bullet_1 B)_{i_2 i_3} = \sum_{i_1=1}^{d_1} M_{i_1 i_2 i_3} B_{1i}, \quad (M \bullet_2 B)_{i_1 i_3} = \sum_{i_2=1}^{d_2} M_{i_1 i_2 i_3} B_{2i}, \quad (M \bullet_3 B)_{i_1 i_2} = \sum_{i_3=1}^{d_3} M_{i_1 i_2 i_3} B_{3i}.
\]
\[ (M \bullet_{3} B)_{i_{1}i_{2}i_{3}} = \sum_{i_{3}=1}^{d_{3}} M_{i_{1}i_{2}i_{3}} B_{i_{3}\ell}. \]

Finally, the tensor contraction (or partial trace) of a third-order tensor \( M \in \mathbb{R}^{d_{1} \times d_{2} \times d_{3}} \) whose dimensions \( d_{p} \) and \( d_{q} \) are equal is denoted as \( \text{Tr}_{p,q} M \). This tensor contraction is a tensor of order 1, i.e., a vector, defined elementwise as

\[ (\text{Tr}_{1,2} M)_{i_{3}} = \sum_{\ell=1}^{d_{1}} M_{\ell i_{1} i_{2}}; \quad (\text{Tr}_{1,3} M)_{i_{2}} = \sum_{\ell=1}^{d_{1}} M_{i_{1} \ell i_{3}}; \quad (\text{Tr}_{2,3} M)_{i_{1}} = \sum_{\ell=1}^{d_{2}} M_{i_{1} i_{2} \ell}. \]

## 2 Dual stochastic control representation of fully nonlinear PDE

We consider a fully nonlinear partial differential equation (PDE) of parabolic type

\[
\begin{aligned}
\partial_{t} u + H(x, D_{x} u, D_{x}^{2} u) &= 0, \quad (t, x) \in [0, T) \times \mathbb{R}^{d}, \\
u(T, x) &= g(x), \quad x \in \mathbb{R}^{d},
\end{aligned}
\]  

(2.1)

where the Hamiltonian \( H : \mathbb{R}^{d} \times \mathbb{R}^{d} \times \mathbb{R}^{d \times d} \rightarrow \mathbb{R} \cup \{\infty\} \) is a lower semi-continuous convex function w.r.t. the last two arguments \((z, \gamma)\), and \( g \) a measurable function on \( \mathbb{R}^{d} \). As it is usual, we assume that \( H(x, z, \gamma) = H(x, z, \gamma^{\top}) \), and that \( \gamma \in \mathbb{S}^{d} \mapsto H(x, z, \gamma) \) is nondecreasing.

Without loss of generality, we may then assume that \( H \) is in a Bellman form

\[
H(x, z, \gamma) = \sup_{a \in A} \left[ b(x, a) \cdot z + \frac{1}{2} \sigma \sigma^{\top} (x, a) : \gamma + f(x, a) \right],
\]

(2.2)

for some measurable functions \( b : \mathbb{R}^{d} \times A \rightarrow \mathbb{R}^{d}, \sigma : \mathbb{R}^{d} \times A \rightarrow \mathbb{R}^{d \times m}, f : \mathbb{R}^{d} \times A \rightarrow \mathbb{R} \), and with \( A \) some subset of \( \mathbb{R}^{q} \). Indeed, such form may arise directly from the dynamic programming equation of a stochastic control problem. Otherwise, it can be written in this formulation by following the duality argument as in Mete Soner et al. (2013). We introduce the concave conjugate of the function \( H(x, z, \gamma) \) w.r.t. the last two variables, that is

\[
f(x, b, c) := \inf_{z \in \mathbb{R}^{d}, \gamma \in \mathbb{R}^{d \times d}} \left[ H(x, z, \gamma) - b \cdot z - \frac{1}{2} c : \gamma \right], \quad x \in \mathbb{R}^{d}, b \in \mathbb{R}^{d}, c \in \mathbb{R}^{d \times d},
\]
and notice that \( f(x, b, c) = f(x, b, c^\top) \) as \( H(x, z, \gamma) = H(x, z, \gamma^\top) \). By the Fenchel–Moreau duality relation, we then get

\[
H(x, z, \gamma) = \sup_{b \in \mathbb{R}^d, c \in \mathbb{S}^d} \left[ b \cdot z + \frac{1}{2} c : \gamma + f(x, b, c) \right] = \sup_{(b, c) \in D_f} \left[ b \cdot z + \frac{1}{2} c : \gamma + f(x, b, c) \right], \quad \text{for } x \in \mathbb{R}^d, z \in \mathbb{R}^d, \gamma \in \mathbb{S}^d,
\]

where \( D_f := \{(b, c) \in \mathbb{R}^d \times \mathbb{S}^d : f(x, b, c) > -\infty \} \subset \mathbb{R}^d \times \mathbb{S}^d \) by the nondecreasing monotonicity of \( \gamma \mapsto H(x, z, \gamma) \). By assuming that \( H \) is uniformly continuous in \( x \), we notice that the domain \( D_f \) does not depend on \( x \). Since for any \( c \in \mathbb{S}^d_+ \), there exists a unique \( s \in \mathbb{S}^d_+ \) s.t. \( c = s^2 \), the above duality relation is in the Bellman form (2.2) with \( a = (b, s) \in A = \{(b, s) \in \mathbb{R}^d \times \mathbb{S}^d_+ : f(x, b, s^2) > -\infty \} \), \( b(x, a) = b \), \( \sigma(x, a) = s \), \( f(x, a) = f(x, b, s^2) \).

It is well known that the solution to the PDE (2.1) with a Hamiltonian \( H \) as in (2.2) admits the stochastic representation

\[
u(t, x) = \sup_{\alpha \in \mathcal{A}} \mathbb{E} \left[ g(X_t^l, x, \alpha) + \int_0^T f(X_t^l, x, \alpha_s) ds \right], \quad (t, x) \in [0, T] \times \mathbb{R}^d, (2.3)
\]

where \( X = X_t^l, x, \alpha \) is solution to the stochastic differential equation

\[
dX_s = b(X_s, \alpha_s) ds + \sigma(X_s, \alpha_s) dW_s, \quad t \leq s \leq T, \quad X_t = x, (2.4)
\]
on a filtered probability space \( (\Omega, \mathcal{F}, \mathbb{P} = (\mathcal{F}_t)_{0 \leq t \leq T}, \mathbb{P}) \) along with a \( m \)-dimensional Brownian motion \( W \), and the control \( \alpha \in \mathcal{A} \) is a pair of \( \mathbb{F} \)-progressively measurable processes valued in \( A \), satisfying suitable integrability conditions for ensuring under some Lipschitz assumptions on the coefficients \( b, \sigma \) that the SDE (2.4) admits a unique strong solution.

Problem (2.3) is a standard stochastic control problem with controlled Markov state process \( X \) governed by (2.4), and it is well known that when it exists the optimal control \( \hat{\alpha} \in \mathcal{A} \) is in closed-loop (or feedback) form, that is

\[
\hat{\alpha}_s = \hat{\alpha}(s, \hat{X}_s^l), \quad t \leq s \leq T, \quad (t, x) \in [0, T] \times \mathbb{R}^d,
\]

for some measurable function \( \hat{\alpha} : [0, T] \times \mathbb{R}^d \rightarrow A \subset \mathbb{R}^q \), where \( \hat{X} \) is the state process controlled by \( \hat{\alpha} \). From the time consistency of the stochastic control problem (2.3), we notice that this feedback form \( \hat{\alpha} \) does not depend on the starting point \( (t, x) \in [0, T] \times \mathbb{R}^d \) of the value function. Furthermore, the value function \( u \) is given by the conditional expectation

\[
u(t, x) = \mathbb{E} \left[ g(\hat{X}_T) + \int_0^T f(\hat{X}_s, \hat{\alpha}(s, \hat{X}_s)) ds \big| \hat{X}_t = x \right]. (2.5)
\]
Our resolution method for approximating a solution to (2.1) on the whole domain $[0, T] \times \mathbb{R}^d$ is performed in two steps:

1. First, following the deep learning approach in Han and Weinan (2016), we shall use neural networks functions $a_\theta : [0, T] \times \mathbb{R}^d \rightarrow A \subset \mathbb{R}^q$, to approximate the optimal feedback control $\hat{b}$, by maximizing over parameters $\theta$ the objective function

$$J(\theta) = \mathbb{E}\left[g(X^\theta_T) + \int_0^T f(X^\theta_t, a_\theta(t, X^\theta_t))dt\right], \quad (2.6)$$

where $X^\theta$ solves

$$dX^\theta_t = b(X^\theta_t, a_\theta(t, X^\theta_t))dt + \sigma(X^\theta_t, a_\theta(t, X^\theta_t))dW_t, \quad 0 \leq t \leq T,$$

with initial condition $X^\theta_0$ distributed according to some law $\mu_0$ on $\mathbb{R}^d$. We denote by $\theta^*$ the “optimal parameter” that maximizes $J(\theta)$, and set $a^* = a_{\theta^*}$. We denote by $X^* = X^\theta^*$ an approximation of the optimal state process $\hat{X}$. For the numerical implementation, we discretize in time the process $X^\theta$ and the integral over $f$ in (2.6), and apply a stochastic gradient ascent algorithm based on samples of $X^\theta$. The pseudo-code is presented in Algorithm 1.

2. Once we get an approximation of the optimal feedback control, we could in principle compute $u(t, x)$ from the Feynman–Kac representation (2.5) by Monte Carlo simulations of $X^*$. However, with the purpose of solving the PDE (2.1) on the whole domain, this has to be performed for every point $(t, x) \in [0, T] \times \mathbb{R}^d$, which is not feasible in practice. Instead, we apply three types of differential learning methods for approximating simultaneously the value function $u$, as well as its derivative: (i) the first one, called differential regression learning, is directly inspired from the original approach in Huge and Savine (2020), and gives an approximation of $u$ and its first derivative $D_x u$ from the minimization of two loss functions based on least-square regressions, (ii) the second one in the spirit of Potters et al. (2001), Vidalas et al. (2018), which uses a contingent claim hedging strategy as a Monte Carlo control variate, approximates the value function and its first derivative from the minimization of a single loss function based on the martingale representation in (2.5), and is referred to as pathwise martingale learning method; (iii) the third one, called pathwise differential learning, provides in addition an accurate approximation of the second derivative $D^2_x u$ of $u$. We develop these three methods and present their pseudo-codes in the next sections.

Notice that since the neural network $a^*$ is by nature a suboptimal feedback policy, the approximation computed in the second step provides a lower bound for the value function $u$ solution to the PDE.


3 Differential regression learning

From the conditional expectation representation (2.5), and its fundamental characterization property as an $L^2$-regression, we have

$$u(t, \hat{X}_t) = \arg\min_{v_t} \mathbb{E}|\hat{Y}_T^t - v_t(\hat{X}_t)|^2(\hat{X}_t), \quad \text{for all } t \in [0, T],$$

where the target payoff is

$$\hat{Y}_T^t = g(\hat{X}_T) + \int_t^T f(\hat{X}_s, \hat{a}(s, \hat{X}_s))ds, \quad t \in [0, T],$$

and the argmin is taken over measurable real-valued functions $v_t$ on $\mathbb{R}^d$ s.t. $v_t(\hat{X}_t)$ is square-integrable.

This suggests to use a class of neural networks (NN) functions $\vartheta_\eta$ on $[0, T] \times \mathbb{R}^d$, with parameters $\eta$, for approximating the value function $u$, and a loss function

$$\hat{L}_{val}(\eta) = \mathbb{E}\left[\int_0^T \left|\hat{Y}_T^t - \vartheta_\eta(t, \hat{X}_t)\right|^2 dt\right] \simeq \mathbb{E}\left[\int_0^T \left|Y_T^{*, t} - \vartheta_\eta(t, X_T^*)\right|^2 dt\right] =: L_{val}(\eta),$$

where

$$Y_T^{*, t} = g(X_T^*) + \int_t^T f(X_s^*, a^*(s, X_s^*))ds, \quad t \in [0, T].$$

As pointed out in Huge and Savine (2020), the training of the loss function $L_{val}$ in (3.2) would require a vast number of samples (often of order millions) to learn accurate approximation of the value function, and is furthermore prone to overfitting. Indeed, by training a neural network to minimize $L_{val}$, we would obtain a function which interpolates the random points generated during training. This comes with two shortcomings. First, a large number of training samples are needed to get satisfactory values of the solution and a good enough generalization to untrained domains. Second, the functions obtained by this method are usually noisy. If we are interested in the derivatives of the PDE solution, as it is the case in finance for example, where greeks are computed to hedge contingent claims, the solution computed might not be accurate enough. Some standard methods, such as Ridge and Lasso penalizations, allow to reduce overfitting but come at the cost of adding bias and an arbitrary penalty and do not ensure that the derivative of the network will be a good approximation of the derivative of the PDE solution. To circumvent these issues, and following the idea in Huge and Savine (2020), we propose to consider furthermore the learning of the derivative of the value function. This method relies on pathwise differentiation of the target payoff $\hat{Y}_T^t$ for deriving the gradient of the value function (see Chapter 7 in
Glasserman, 2013)

\[ D_x u(t, \hat{X}_t) = \mathbb{E}\left[ \hat{Z}_T^t \mid \mathcal{F}_t \right], \quad t \in [0, T], \tag{3.4} \]

where \( \hat{Z}_T^t = D_x \hat{X}_t \hat{Y}_T^t \) (recall that \( \hat{Y}_T^t \) is a function of \( \hat{X}_t \) as the control \( \hat{a} \) is a feedback function of \( t \) and \( \hat{X}_t \)). This suggests to complete the learning of the value function together with its derivative by considering furthermore the loss function

\[ \hat{L}_{der}(\eta) = \mathbb{E}\left[ \int_0^T \left| \hat{Z}_t^t - D_x \hat{a}^\eta(t, \hat{X}_t) \right|^2 dt \right] \]

\[ \simeq \mathbb{E}\left[ \int_0^T \left| Z_T^t - D_x \hat{a}^\eta(t, X_t^*) \right|^2 dt \right] =: L_{der}^*(\eta), \]

where \( Z_T^t = D_X^* Y_T^* \) valued in \( \mathbb{R}^d \), is obtained by automatic differentiation as

\[ Z_T^t = (D_X X_T)^\top D_x g(X_T) + \int_0^T (D_X X_s)^\top D_x f^a^*(s, X_s) ds, \quad t \in [0, T], \tag{3.5} \]

where we denote by \( f^a^*(t, x) = f(x, a^*(t, x)) \), and assuming that \( g \) and \( f \) are continuously differentiable. Notice that \( D_x f^a^* = D_x f + (D_t a^*)^\top D_a f \), where the derivatives \( D_x a^* \) of the approximate optimal feedback control in the class of neural networks can be efficiently computed by automatic differentiation. Here, to alleviate notations, we have dropped the superscript * for the state \( X = X^* \). Actually, when \( g \) and \( f \) are only piecewise-differentiable, the above relation still holds when the marginal law of \( X_s \) is absolutely continuous with respect to Lebesgue measure on \( \mathbb{R}^d \), which is satisfied under nondegeneracy conditions on the diffusion coefficients (see Theorem 2.3.2 in Nualart, 1995).

We recall that the flow derivative of the optimal state process, valued in \( \mathbb{R}^{d \times d} \), is solution to the SDE (see, e.g., Protter, 2005)

\[ D_X X_s = I_d + \int_t^s D_x b^a^*(r, X_r) D_X X_r dr + D_x \sigma^a^*(r, X_r) D_X X_r dW^j_r, \quad s \leq t \leq T, \tag{3.6} \]

where we denote by \( b^a^*(t, x) = b(x, a^*(t, x)) \), \( \sigma^a^*(t, x) = \sigma(x, a^*(t, x)) \), and use the Einstein summation convention over the repeated index \( j = 1, \ldots, d \), with \( \sigma^a^* \) (resp. \( \sigma^j \)) the \( j \)-th column of the matrix \( \sigma^a^* \) (resp. \( \sigma^j \)). Notice that \( D_x b^a^* = D_x b + D_a b D_x a^* \), and \( D_x \sigma^a^*_j = D_x \sigma_j + D_a \sigma_j D_x a^* \).

**Remark 3.1** We have an alternative representation (3.4) for the gradient of \( v \), which avoids smoothness assumptions on the coefficients. It is expressed with \( \hat{Z}_T^t \) given by...
\[ Z_t = g(\hat{X}_T) \hat{H}_T^T + \int_T^T f(\hat{X}_s, \hat{a}(s, \hat{X}_s)) \hat{H}_s^T ds, \quad t \in [0, T]. \]  

(3.7)

with the so-called Malliavin weights \( \hat{H}_s^T, t \leq s \), given by

\[ H_s^T = \frac{1}{s-t} \int_t^s \sigma^{-1}(t, \hat{X}_t) \sigma^{-1}(r, \hat{X}_r) D_{\hat{X}_t} \hat{X}_r \sigma(t, \hat{X}_r) dW_r, \]

where \( \sigma^{-1} = \sigma^\top (\sigma \sigma^\top)^{-1} \) is the right-inverse of the matrix \( \sigma \) assumed to be of full rank. Therefore, in the loss function \( L^*_{\text{der}} \), instead of \( Z_{T}^{*, T} \) as in (3.5), we can use alternately \( Z_{T}^{*, T} \) as in (3.7), with \( \hat{X} \) approximated by \( X^* \), and \( \hat{a} \) approximated by \( a^* \).

To illustrate the interest of learning the derivative of the value function, we plot in Fig. 1 the results obtained by learning the value and the derivative (Differential regression learning) or by learning just the value (Simple learning) of the call option price with market impact (see the application presented in Sect. 6.3).

As a reference, we compute the option price on chosen points \((t, x) \in \mathbb{R}_+ \times \mathbb{R}\) by Monte Carlo, as explained in Sect. 5.4.

In the differential regression learning method, we use a combination of the loss \( L^*_{\text{val}} \) and \( L^*_{\text{der}} \) in the training of the neural networks for approximating the value function and its derivative. We train alternately the value and the derivative of the network by taking a gradient step to minimize \( L^*_{\text{val}} \) every even number of epochs and a gradient step to minimize \( L^*_{\text{der}} \) every odd number of epochs. An alternative method would be to minimize a convex combination of \( L^*_{\text{val}} \) and \( L^*_{\text{der}} \) with weights \( w_{\text{val}} \) and \( w_{\text{der}} = 1 - w_{\text{val}} \). These weights could be chosen by performing a grid search or a random search, such as advocated in Bergstra and Bengio (2012) for the choice of neural network hyperparameters. Some theoretical arguments could also be derived to choose these weights, as it has been done in van der Meer et al. (2021) to optimally choose the weights of the losses associated with different constraints of a PDE when using Physics Informed Neural Networks (Raissi et al., 2019). Our approach proves to be effective, as shown by the numerical results in Sects. 6.1 and 6.3, and avoids the need to chose a value for this additional hyperparameter. The algorithmic implementation and the pseudo-codes are described in Sect. 5.2.

### 4 Pathwise learning

#### 4.1 Pathwise martingale learning

This approach is based on the martingale representation related to relation (2.5), which leads by Itô’s formula to the equation

\[ \hat{Y}_T^I = u(t, \hat{X}_t) + \int_T^T (D_X u(s, \hat{X}_s))^\top \sigma^\hat{a}(s, \hat{X}_s) dW_s, \quad t \in [0, T], \]

(4.1)
Fig. 1 Value function values (first line) and derivatives (second line) obtained by Differential Learning (navy curve) (Algorithm 2), Simple learning (blue dashed curve) and Monte Carlo (red dots) plotted as functions of $x$, for fixed values of $t$

where we recall that $\hat{Y}_t^T$ is given in (3.1), and denote $\sigma(\hat{a}(t,x))$. This suggests to use a class of neural networks (NN) functions $\vartheta_\eta$ on $[0,T] \times \mathbb{R}^d$, with parameters $\eta$, for approximating the value function $u$, and a loss function

$$\hat{L}_{mar}(\eta) = \mathbb{E} \left[ \int_0^T |\hat{Y}_t^T - \vartheta(\eta(t, \hat{X}_t)) - \int_t^T (D_x \vartheta(\eta(s, \hat{X}_s))) \sigma(\hat{a}(s, \hat{X}_s))dW_s |^2 dt \right]$$

$$\simeq \mathbb{E} \left[ \int_0^T |Y^*,t^T - \vartheta(\eta(t, X^*_t)) - \int_t^T (D_x \vartheta(\eta(s, X^*_s))) \sigma_*(s, X^*_s)dW_s |^2 dt \right]$$

$$=: L^{*}_{mar}(\eta).$$

Recall that $Y^*$ is the process defined in (3.3) when using the approximate optimal neural network $a^*$. Alternately, we can use two classes of neural networks: one $\vartheta_\eta$ from $[0,T] \times \mathbb{R}^d$ into $\mathbb{R}$, with parameters $\eta$, for the approximation of $u$, and a second one $Z_\delta$ from $[0,T] \times \mathbb{R}^d$ into $\mathbb{R}^d$, with parameters $\delta$, for the approximation of $D_x u$. We then consider a loss function

$$\bar{L}^{*}_{mar}(\eta, \delta) := \mathbb{E} \left[ \int_0^T |Y^*,t^T - \vartheta(\eta(t, X^*_t)) - \int_t^T Z_\delta(s, X^*_s) \sigma_*(s, X^*_s)dW_s |^2 dt \right].$$
Notice that compared to the deep BSDE approach in Han et al. (2017), which considers a loss function from the misfit between the l.h.s (the target) and r.h.s. of (4.1) at time 0, namely

$$\tilde{L}^*_{DBSDE}(y_0, \delta) := \mathbb{E}\left[|Y_T^*, 0 - y_0 - \int_0^T \mathcal{Z}^\delta(s, X_s^*)^\top \sigma^a^*(s, X_s^*)dW_s|^2\right].$$

our loss functions $L^*_{mar}$ or $\tilde{L}^*_{mar}$ take into account the misfit between the l.h.s and r.h.s. of (4.1) at any time $t \in [0, T]$, since our goal is to approximate the solution $u$ (and its derivative) on the whole domain $[0, T] \times \mathbb{R}^d$ (and not only at time $t = 0$).

**Remark 4.1** From the martingale representation theorem (4.1), the unique minimizer to the loss function $\tilde{L}^*_{mar}$ is obtained in theory when $\vartheta^\eta$ is equal to $u$ and $\mathcal{Z}^\delta$ is equal to the gradient of $u$. Therefore, by considering two distinct neural networks, we get more flexibility to learn $u$ and its gradient, hence more chance to achieve a very small loss function compared to the case where we constrain the gradient relation between the neural networks.

### 4.2 Pathwise differential learning

We can further compute the pathwise derivative in the martingale representation relation (4.1) to obtain a second estimator linking the first and second derivatives of $u(t, x)$. Indeed, by El Karoui et al. (1997), we have

$$D\hat{X}_t^Y = D_x u(t, \hat{X}_t) + \int_t^T \left( D_x \sigma^\hat{a}(s, \hat{X}_s) \bullet_3 D\hat{X}_t^Y \hat{X}_s \right) \bullet_1 D_x u(s, \hat{X}_s) + \sigma^\hat{a}(s, \hat{X}_s)^\top D^2_x u(s, \hat{X}_s) D\hat{X}_t \hat{X}_s \right)^\top dW_s, \quad t \in [0, T].$$

(4.2)

This suggests to use a class of neural networks (NN) functions $\vartheta^\eta$ on $[0, T] \times \mathbb{R}^d$, with parameters $\eta$, for approximating the value function $u$, and a loss function

$$L^*_{dermar}(\eta) = \mathbb{E}\left[ \int_0^T \left| Z_T^{*, t} - D_x \vartheta^\eta(t, X_t) \right|^2 \right]$$

where we recall that $Z_T^{*, t}$ is given in (3.5), and we omit the superscript $*$ in the approximation of the optimal state process $X = X^*$. Alternately, we can use three classes of neural networks: one $\vartheta^\eta$ from $[0, T] \times \mathbb{R}^d$ into $\mathbb{R}$, with parameters $\eta$, for the approximation of $u$, a second one $\mathcal{Z}^\delta$ from $[0, T] \times \mathbb{R}^d$ into $\mathbb{R}^d$, with parameters $\delta$, for the approximation of $D_x u$, and a third one $\Gamma$ from $[0, T] \times \mathbb{R}^d$ into $\mathbb{S}^d$, with
parameters $\epsilon$, for the approximation of $D_{XX}u$, and consider a loss function

$$
\tilde{L}_{\text{dermar}}^{*}(\delta, \epsilon) = \mathbb{E}\left[ \int_{0}^{T} \left| Z_{T}^{*} - Z^{\delta}(t, X_{t}) \right| - \int_{t}^{T} \left( [D_{s}\sigma^{a*}(s, X_{s}) \bullet_{3} D_{X_{t}}X_{s}] \bullet_{1} Z^{\delta}(s, X_{s}) + \sigma^{a*}(s, X_{s})^{\top} \Gamma_{\epsilon}(s, X_{s}) D_{X_{t}}X_{s}^{\top} dW_{s} \right)^{2} \right].
$$

Again, the argument for using three distinct neural networks comes from the uniqueness of $u$, its gradient, and its Hessian in the martingale representation equations (4.1) and (4.2).

**Remark 4.2** The different versions of the algorithms, using either one neural network and its derivatives computed by automatic differentiation, or one neural network to represent the value of the PDE solution $u$ and different neural networks to represent its derivatives, have both their advantages and their weaknesses. The versions of the algorithm using only one neural network and its derivatives to represent $u$ and its spatial derivatives, which are the one implemented and numerically tested in this article, have the advantage of empirically converging faster and thus yielding more accurate solutions when a small number of Monte Carlo trajectories are used (in the order of 8000 in the case of our article). However, since only one neural network is used, there is less flexibility to learn $u$ and its derivative than in the method where multiple neural networks are used. Empirically, while using multiple neural networks converges slower and yields less accurate results when a small number of samples are used, it has more chance to achieve a null loss function when the number of samples is large.

As with the Differential regression learning method presented in Sect. 3, the neural network can be trained either by minimizing a convex combination of the losses $L^{*}_{\text{mar}}$ and $L^{*}_{\text{dermar}}$ or by minimizing these losses individually. During our numerical experiment, we minimized these two losses individually, but contrary to the algorithm used in the Differential regression learning, for each epoch, a gradient step was made to minimize $L^{*}_{\text{mar}}$, and then, another one was made to minimize $L^{*}_{\text{dermar}}$.

**Remark 4.3** This optimization scheme was found to be more effective when optimizing the neural network parameters for the Pathwise differential learning. For the Differential regression learning method, making a gradient step on only one of these two losses at each epoch gave better results. The difference between these two optimization schemes lies in the fact that if both the losses $L^{*}_{\text{mar}}$ and $L^{*}_{\text{dermar}}$ are optimized during an epoch, these two losses are computed using the “old” network weights $\eta$, then these weights are modified two times, first by making a gradient step to minimize $L^{*}_{\text{mar}}$, and then by taking another gradient step to minimize $L^{*}_{\text{dermar}}$, as written below

One epoch:

$$
\eta' \leftarrow \eta - \nabla_{\eta}L^{*}_{\text{mar}}(\eta), \quad \eta'' \leftarrow \eta' - \nabla_{\eta}L^{*}_{\text{dermar}}(\eta).
$$
When the network is optimized by minimizing alternatively one of these two losses at each epoch, as in the Differential regression method, one of the losses, say $L^*_{mar}$, is computed using the “old” weights $\eta$. A gradient step is then made to minimize this loss and obtain new network weights $\eta'$, which are then used in the next epoch to compute the loss $L^*_{mar}$ and make the next gradient step, as written below:

One epoch:
$$\eta' \leftarrow \eta - \nabla_\eta L^*_{mar}(\eta),$$

Next epoch:
$$\eta'' \leftarrow \eta' - \nabla_\eta L^*_{dermar}(\eta').$$

5 Algorithms

In this section, we detail the implementation of the methods presented above. Our algorithms are discretized in time for the training of the processes and for the integrals that appear in the loss functions, and which are approximated by Riemann sums. In the sequel, we are then given a mesh grid $T_N = \{0 = t_0 < t_1 < \ldots < t_N = T\}$ of $[0, T]$ with $\Delta t_n = t_{n+1} - t_n$, for $n = 0, \ldots, N - 1$.

Our codes are written in Python and we use the Tensorflow library to implement the neural networks and compute the derivatives present in our calculations by auto-differentiation (AAD).

5.1 Approximation of the optimal control

As a first step, we consider a neural network $a_\theta$ from $[0, T] \times \mathbb{R}^d$ into $A \subset \mathbb{R}^q$ for the approximation of the feedback control, and the associated discretized state process

$$X_{t_{n+1}}^\theta = X_{t_n}^\theta + b(X_{t_n}^\theta, a_\theta(t_n, X_{t_n}^\theta)) \Delta t + \sigma(X_{t_n}^\theta, a_\theta(t_n, X_{t_n}^\theta)) \Delta W_{t_n}, \quad n = 0, \ldots, N - 1,$$

starting from $X_0 \sim \mu_0$ (probability distribution on $\mathbb{R}^d$), and where $\Delta W_{t_n} = W_{t_{n+1}} - W_{t_n}$. As in Huré et al. (2021), to constrain the output of the neural network $a_\theta$ to be in the control space $A$, we define a custom activation function $\sigma_A$ for the output layer of the network. This activation function $\sigma_A$ is chosen depending on the form of the control space $A$. When $A = \mathbb{R}^q$, $\sigma_A$ is equal to the identity function. When the control space if of the form $A = \prod_{i=1}^q \left[a_i, \infty\right)$, one can take the component-wise ReLU activation function (possibly shifted and scaled); when $A = \prod_{i=1}^q \left[a_i, b_i\right]$, for $a_i \leq b_i$, $i = 1, \ldots, q$, one can take the component-wise sigmoid activation function (possibly shifted and scaled). For the numerical experiments presented here, we used the ELU (Exponential Linear Unit) activation function, defined as
Fig. 2  Structure of the neural network used to approximate the optimal control and value function

\[ ELU(x) = \begin{cases} 
    x & x > 0 \\
    \alpha(e^x - 1) & x \leq 0,
\end{cases} \]

with parameter \( \alpha \) for the hidden layers.

The structure of the neural networks used to approximate the feedback control is represented in Fig. 2. It is composed of two dense feed-forward sub-networks composed of two layers of \( n \) neurons, taking, respectively, the time \( t \) and the state \( x \) as input. The outputs of these two sub-networks, which are in \( \mathbb{R}^n \), are then concatenated and inputed in a third dense network composed of two layers of \( n \) neurons and a last layer of \( q \) neurons which outputs the approximation of the control in \( \mathbb{R}^q \). This structure adds more flexibility compared to the network structure usually implemented, where the time and state variables are directly concatenated and passed through a dense feed-forward network. It allows to use different activation functions in each sub-network and adapts well to situations where the network is used to approximate a function which has very different behaviors in its time and state variables. The same structure is used for the neural networks used to approximate the value function by Differential regression learning or Pathwise learning. In the applications presented in this article, we used neural networks with \( n = 50 \) neurons per layer and \( q = 1 \).

**Remark 5.1** For the approximation of the optimal control for the example of the Merton portfolio selection, presented in Sect. 6.1, the neural network used had the same architecture as the one described above but used 20 neurons per layer and hyperbolic tangent activations functions. This configuration proved to work better empirically, as the exploding gradient problem was encountered while training a network using 50 neurons and ELU activation function.

For the training of the neural network control \( a_\theta \), we use a batch of \( M \) independent trajectories \( \{x_{t_n}^{m,\theta}, t_n \in T_N\}, m = 1, \ldots, M, \) of \( \{X_t^{\theta}, t_n \in T_N\} \), and apply a stochastic
The gradient ascent method to the empirical gain function

\[ J_M(\theta) = \frac{1}{M} \sum_{m=1}^{M} \left[ g(X_T^{m,\theta}) + \sum_{n=0}^{N-1} f(X_{tn}^{m,\theta}, a_\theta(t_n, X_{tn}^{m,\theta})) \Delta t_n \right]. \]

The pseudo-code is described in Algorithm 1. The output of this algorithm yields a parameter \( \theta^* \), and so, an approximation of the optimal feedback control with \( a^* = a_{\theta^*} \), and of the associated optimal state process with \( X^* = X_{\theta^*} \). In the sequel, to alleviate notations, we shall omit the superscript \( * \), and simply denote \( a \) and \( X \).

Algorithm 1: Deep learning scheme to solve the stochastic control problem (2.3)

**Result:** A set of optimized parameters \( \theta^* \).

**Initialize the learning rate \( l \) and the neural network \( a_\theta \);**

**Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_0 < t_1 < \ldots < t_N = T \) with time steps \( (\Delta t_n)_{n=0,\ldots,N-1} \);**

**Generate a batch of \( M \) starting points \( X_0 \sim \mu_0 \) and Brownian increments \( (\Delta W_{tn})_{n=0,\ldots,N-1} \) in \( \mathbb{R}^d \);**

**for each epoch do**

**for each batch element \( m \) do**

Compute the trajectory \( (X_{tn}^{m,\theta})_{n=0,\ldots,N} \) through the scheme

\[ X_{tn+1}^{m,\theta} = X_{tn}^{m,\theta} + b(X_{tn}^{m,\theta}, a_\theta(t_n, X_{tn}^{m,\theta})) \Delta t_n + \sigma(X_{tn}^{m,\theta}, a_\theta(t_n, X_{tn}^{m,\theta})) \Delta W_{tn}^{m}, \]

from the generated starting point \( X_0^{m} \) and Brownian increments \( (\Delta W_{tn}^{m})_{n=0,\ldots,N-1} \);

**end**

Compute the batch loss

\[ J_M(\theta) = \frac{1}{M} \sum_{m=1}^{M} \left[ g(X_T^{m,\theta}) + \sum_{n=0}^{N-1} f(X_{tn}^{m,\theta}, a_\theta(t_n, X_{tn}^{m,\theta})) \Delta t_n \right]. \]

Compute the gradients \( \nabla_\theta J_M(\theta) \);

Update \( \theta \leftarrow \theta - l \nabla_\theta J_M(\theta) \);

**end**

**Return:** The set of optimized parameters \( \theta^* \);

5.2 Differential regression learning algorithm

We consider a neural network \( \vartheta^N \) from \([0, T] \times \mathbb{R}^d \) into \( \mathbb{R} \) for the approximation of the value function. The derivatives \( D_x g, D_x f, D_x b, D_x \sigma, D_x a, \) and \( D_x \vartheta^N \) that appear in the differential regression learning methods are computed straightforwardly by auto-differentiation. Concerning the flow derivative of the approximate optimal state process, it is computed by time discretization of (3.6), which can be efficiently obtained by storing the one-step derivatives

\[ DX_{tn} X_{tn+1} = I_d + D_x b^a(t_n, X_{tn}) \Delta t_n + \sum_{j=1}^{d} D_x \sigma^a_j(t_n, X_{tn}) \Delta W_{tn}^j, \quad n = 0, \ldots, N - 1, \]
and then use the chain rule
\[ DX_{tn}X_{tp} = DX_{tn}X_{tn+1} \cdots DX_{tp-1}X_{tp}, \quad \text{for } n < p \in [0, N]. \]

The target payoff and its derivative are then computed as
\[
Y^n_T = g(X_T) + \sum_{p=n}^{N-1} f^{a^n}(t_p, X_{tp}) \Delta t_p, \quad n = 0, \ldots, N,
\]
\[
Z^n_T = (DX_{tn}X_T)^\top D_x g(X_T) + \sum_{p=n}^{N-1} \left( DX_{tn}X_{tp} \right)^\top D_x f^{a^n}(t_p, X_{tp}) \Delta t_p,
\]
with the convention that the above sum over \( p \) is zero when \( n = N \).

For the training of the neural network \( \vartheta^\eta \), we use a batch of \( M \) independent samples \((x_{tn}, y_{tn}^m, z_{tn}^m)\), \( m = 1, \ldots, M \), of \((X_{tn}, Y^n_T, Z^n_T)\), \( n = 0, \ldots, N \), and apply stochastic gradient descent for the minimization of the mean squared error functions
\[
MSE_{\text{val}}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} \left| y_{tn}^m - \vartheta^\eta(t_n, x_{tn}^m) \right|^2 \Delta t_n
\]
\[
MSE_{\text{der}}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} \frac{1}{\|z_{tn}^m\|^2} \left| z_{tn}^m - D_x \vartheta^\eta(t_n, x_{tn}^m) \right|^2 \Delta t_n.
\]

Here, as in Huge and Savine (2020) (Appendix 2), we normalize the derivative loss by the \( L_2 \) norm of the target derivative computed along the batch dimension \( \|z_{tn}^m\|^2 := \left( \sum_{m=1}^{M} z_{tn}^m \right)^2 \).

The pseudo-code is described in Algorithm 2.

**Remark 5.2** Notice that in the above expressions of the mean squared errors, the neural network is trained from the time \( t_0 = 0 \) up to time \( t_{N-1} < T \) and is thus not trained on the terminal condition of the PDE (2.1). This choice is justified by the fact that the neural network is already implicitly trained to fit the terminal condition at time \( T \), since the terminal function \( g \) appears in the losses. Furthermore, we observed that the regularity of the terminal function affects the performance of the neural network. If the terminal function is at least of class \( C^1 \), no problem arises as the regularity of the solution to the PDE (2.1) is the same on the domain \([0, T) \times \mathbb{R}^d\) and on the terminal domain \([T] \times \mathbb{R}^d\). If we use a neural network \( \vartheta^\eta \) of regularity \( C^1 \), we will then be able to approximate the PDE solution on the entire domain \([0, T] \times \mathbb{R}^d\). However, if the terminal condition’s regularity is less than \( C^1 \), it will be difficult for the neural network to approximate the PDE solution on the entire domain \([0, T] \times \mathbb{R}^d\). Indeed, the solution of parabolic PDEs is often smoother than its terminal (or initial) condition; thus, the neural network will have to approximate a function that has a continuous first derivative on the domain \([0, T) \times \mathbb{R}^d\) and a discontinuous first derivative on \([T] \times \mathbb{R}^d\). If the neural network used is not \( C^1 \), which is the case for a network with ReLU.
activation functions for example, the network will give a good approximation of the terminal condition but will give a worst fit of the solution on the domain $[0, T) \times \mathbb{R}^d$, and particularly of its derivatives. On the contrary, if the neural network used is $C^1$, which is the case when the ELU activation function is used, the network will give a good approximation of the solution on $[0, T) \times \mathbb{R}^d$ but will give a worse approximation of the terminal condition. The difficulty thus comes from the fact that we try to obtain a solution on the entire domain of the PDE. As the terminal condition is known and the quantity of interest is the solution of the PDE (2.1) on the domain $[0, T) \times \mathbb{R}^d$, we choose to use a $C^1$ neural network trained on this domain.

5.3 Pathwise learning algorithms

We consider a neural network $\vartheta^\eta$ from $[0, T] \times \mathbb{R}^d$ into $\mathbb{R}$ for the approximation of the value function. For the training of this neural network, we use a batch of $M$ independent samples $(x_{tn}, y_{tn}, \Delta w_{tn}^m)$, $m = 1, \ldots, M$, of $(X_{tn}, Y_{tn}, \Delta W_{tn})$, $n = 0, \ldots, N$, and apply stochastic gradient descent for the minimization of the mean squared error function.
\[ MSE_{mar}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} |y_{T}^{m, t_{n}} - \vartheta(\eta, x_{t_{n}}^{m})\] 

\[- \sum_{p=n}^{N-1} (D_{x} \vartheta(\eta, x_{t_{p}}^{m}))^{\top} \sigma(x_{t_{p}}^{m}, a^{*}(x_{t_{p}}^{m})) \Delta u_{t_{p}}^{m} |^{2} \Delta t_{t_{n}}. \]  

(5.1)

The pseudo-code for the pathwise martingale learning is described in Algorithm 3.

**Algorithm 3:** Deep learning scheme for Pathwise martingale learning with 1 NN

**Result:** A set of optimized parameters \( \eta^{*} \);

Initialize the learning rate \( l \), the neural networks \( \vartheta(\eta) \);

Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_{0} < t_{1} < ... < t_{N} = T \) with time steps \( (\Delta t_{n})_{n=0,...,N-1} \);

Generate a batch of \( M \) starting points \( X_{0} \sim \mu_{0} \) and Brownian increments \( (\Delta W_{t_{n}})_{n=0,...,N} \) in \( \mathbb{R}^{d} \);

for each batch element \( m \) do

Compute the trajectory \( (x_{t_{n}}^{m})_{n=0,...,N} \) through the scheme

\[ x_{t_{n+1}}^{m} = x_{t_{n}}^{m} + b^{*}(t_{n}, x_{t_{n}}^{m}) \Delta t_{n} + \sigma^{*}(t_{n}, x_{t_{n}}^{m}) \Delta W_{t_{n}}^{m}, \]

from the generated starting point \( x_{t_{0}}^{m} \), Brownian increments \( (\Delta W_{t_{n}}^{m})_{n=0,...,N-1} \) and previously trained control \( a = a_{\theta}^{*} \);

Compute the value target \( (y_{T}^{m, t_{n}})_{n=0,...,N} \);

end

for each epoch do

Compute, for every batch element \( m \), the integral

\[ \sum_{n=0}^{N-1} |y_{T}^{m, t_{n}} - \vartheta(\eta, x_{t_{n}}^{m}) - \sum_{p=n}^{N-1} (D_{x} \vartheta(\eta, x_{t_{p}}^{m}))^{\top} \sigma(x_{t_{p}}^{m}, a^{*}(x_{t_{p}}^{m})) \Delta W_{t_{p}}^{m}|^{2} \Delta t_{t_{n}}; \]

Compute the batch loss \( MSE_{mar}(\eta) \);

Compute the gradient \( \nabla_{\eta} MSE_{mar}(\eta) \);

Update \( \eta \leftarrow \eta - l \nabla_{\eta} MSE_{mar}(\eta) \);

end

Return: The set of optimized parameters \( \eta^{*} \);

Alternately, we can use another neural network \( \mathcal{Z}_{\delta} \) from \( [0, T] \times \mathbb{R}^{d} \) into \( \mathbb{R}^{d} \) for the approximation of the gradient of the solution, and then use the mean squared error function

\[ \tilde{MSE}_{mar}(\eta, \delta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} |y_{T}^{m, t_{n}} - \vartheta(\eta, x_{t_{n}}^{m})|^{2} \Delta t_{t_{n}}; \]

The pseudo-code for the pathwise martingale learning with two neural networks is described in Algorithm 7 in Appendix A.

For the differential version of this algorithm, we also use a neural network \( \vartheta(\eta) \) from \( [0, T] \times \mathbb{R}^{d} \) into \( \mathbb{R} \) for the approximation of the value function that we train using the same batch of \( M \) independent samples and applying stochastic gradient descent for the minimization of both the mean squared error functions defined in (5.1) and the
following one:

\[ \text{MSE}_{\text{dermar}}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} c_{T}^{m,n} - D_{x} \vartheta^{\eta}(t_{n}, x_{n}^{m}) \]

\[ - \sum_{p=n}^{N-1} \left( [D_{x} \sigma a^{*}(t_{p}, x_{t_{p}}^{m}) \bullet_{3} D_{x_{t_{n}}} x_{t_{p}}^{m}] \bullet_{1} D_{x} \vartheta^{\eta}(t_{p}, x_{t_{p}}^{m}) \right. \]

\[ + \sigma a^{*}(t_{p}, x_{t_{p}}^{m})^\top D_{xx} \vartheta^{\eta}(t_{p}, x_{t_{p}}^{m}) D_{x_{t_{n}}} x_{t_{p}}^{m} \bigg) \Delta w_{t_{p}}^{m} \bigg|^{2} \Delta t_{n}. \]

The pseudo-code is described in Algorithm 4.

**Algorithm 4:** Deep learning scheme for Pathwise differential learning with 1 NN

**Result:** A set of optimized parameters \( \eta^{*} \);

Initialize the learning rate \( l \), the neural networks \( \vartheta^{\eta} \);

Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_{0} < t_{1} < \ldots < t_{N} = T \) with time steps \( \Delta t_{n} = 0, \ldots, N-1 \);

Generate a batch of \( M \) starting points \( X_{0} \sim \mu_{0} \) and Brownian increments \( \Delta W_{t_{n}} = 0, \ldots, N \) in \( \mathbb{R}^{d} \);

for each batch element \( m \) do

Compute the trajectory \( (x_{t_{n}}^{m})_{n=0,..,N} \) through the scheme

\[ x_{t_{n+1}}^{m} = x_{t_{n}}^{m} + a^{*}(t_{n}, x_{t_{n}}^{m}) \Delta t_{n} + \sigma a^{*}(t_{n}, x_{t_{n}}^{m}) \Delta w_{t_{n}}^{m}, \]

from the generated starting point \( x_{t_{0}}^{m} \), Brownian increments \( \Delta w_{t_{n}}^{m} = 0, \ldots, N-1 \) and previously trained control \( a = a^{*} \);

Compute the value and derivative targets \( (x_{t_{n}}^{m,n})_{n=0,..,N} \) and \((x_{t_{n}}^{m,n})_{n=0,..,N} \);

end

for each epoch do

Compute, for every batch element \( m \), the integral

\[ \sum_{n=0}^{N-1} c_{t_{n}}^{m} - \vartheta^{\eta}(t_{n}, x_{t_{n}}^{m}) - \sum_{p=n}^{N-1} \left( D_{x} \vartheta^{\eta}(t_{p}, x_{t_{p}}^{m}) \right) \sigma a^{*}(t_{p}, x_{t_{p}}^{m}) \Delta w_{t_{p}}^{m} \Delta t_{n}; \]

Compute the batch loss \( \text{MSE}_{\text{mar}}(\eta) \);

Compute the gradient \( \nabla_{\eta} \text{MSE}_{\text{mar}}(\eta) \);

Update \( \eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{mar}}(\eta) \);

Compute, for every batch element \( m \), the integral

\[ \sum_{n=0}^{N-1} c_{t_{n}}^{m} - D_{x} \vartheta^{\eta}(t_{n}, x_{t_{n}}^{m}) - \sum_{p=n}^{N-1} \left( D_{x} \vartheta^{\eta}(t_{p}, x_{t_{p}}^{m}) \right) \sigma a^{*}(t_{p}, x_{t_{p}}^{m}) \Delta w_{t_{p}}^{m} \Delta t_{n}; \]

Compute the batch loss \( \text{MSE}_{\text{dermar}}(\eta) \);

Compute the gradient \( \nabla_{\eta} \text{MSE}_{\text{dermar}}(\eta) \);

Update \( \eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{dermar}}(\eta) \);

end

Return: The set of optimized parameters \( \eta^{*} \);

Alternately, in addition to the NN \( \vartheta^{\eta} \) for \( u \), we can use neural networks \( \mathcal{Z}^{\delta} \) and \( \Gamma^{\epsilon} \) for the gradient and the Hessian that we train with the loss function

\[ \text{MSE}_{\text{dermar}}(\delta, \epsilon) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} c_{t_{n}}^{m,n} - \mathcal{Z}^{\delta}(t_{n}, x_{t_{n}}^{m}) \]

\[ - \sum_{p=n}^{N-1} \left( [D_{x} \sigma a^{*}(t_{p}, x_{t_{p}}^{m}) \bullet_{3} D_{x_{t_{n}}} x_{t_{p}}^{m}] \bullet_{1} \mathcal{Z}^{\delta}(t_{p}, x_{t_{p}}^{m}) \right. \]

\[ + \sigma a^{*}(t_{p}, x_{t_{p}}^{m})^\top \Gamma^{\epsilon}(t_{p}, x_{t_{p}}^{m}) D_{x_{t_{n}}} x_{t_{p}}^{m} \bigg) \Delta w_{t_{p}}^{m} \bigg|^{2} \Delta t_{n}. \]
where \( D_{x_{tn}} x_{tn}^m \) denotes the flow derivative of the approximate optimal state process at time \( t_p \) w.r.t the state at time \( t_n \) along the path \( m \) of the batch.

The pseudo-code for this version using three neural networks is described in Algorithm 8 in Appendix A.

### 5.4 Validation tests

The deep learning methods described above provide an approximation \( \vartheta \) of the solution \( u \) to the PDE, which relies up-front on the approximation \( a^\theta \) of the optimal control \( \hat{a} \) arising from the dual stochastic control representation.

We can test and validate the convergence and accuracies of these approximations as follows. On the one hand, as in the DGM and PINN methods, see Sirignano and Spiliopoulos (2018) and Raissi et al. (2019), we can compute the losses \( L_{res} \) and \( L_{term} \), associated, respectively, with the residual and with the terminal condition of the partial differential equation (2.1)

\[
L_{res} := \frac{1}{|T||\chi|} \sum_{t \in T, x \in \chi} \left| \partial_t \vartheta^\eta + H(x, D_x \vartheta^\eta, D_x^2 \vartheta^\eta) \right|^2,
\]

\[
L_{term} := \frac{1}{|\chi|} \sum_{x \in \chi} \left| \vartheta^\eta(T, x) - g(x) \right|^2,
\]

where the time grid \( T \) is composed of times that were not used during the network training, so as to verify the generalization of the solution obtained, and \( \chi \) is a bounded space grid in \( \mathbb{R}^d \). As the state diffusion \( (X_{\theta t}^\eta)_{0 \leq t \leq T} \) is not bounded, the bounds of the space grid are fixed arbitrarily depending on the domain of interest. Alternatively, these bounds could be chosen based on the distribution of the values attained during the computation of the diffusion scheme of \( (X_{\theta t}^\eta)_{0 \leq t \leq T} \).

On the other hand, by noting that the optimal control should satisfy the optimality condition

\[
D_a b(x, \hat{a}) D_x u(t, x) + \frac{1}{2} Tr_{1,2} \left( D_a \sigma \sigma^\top (x, \hat{a}) \bullet_2 D_{xx} u(t, x) \right) + D_a f(x, \hat{a}) = 0,
\]

we can check the accuracy the approximation \( a^\theta \) of the optimal control by computing the following loss:

\[
L_{optim} = \frac{1}{|T||\chi|} \sum_{t \in T, x \in \chi} \left| D_a b(x, a^\theta) D_x \vartheta^\eta(t, x) \right|^2
\]

\[
+ \frac{1}{2} Tr_{1,2} \left( D_a \sigma \sigma^\top (x, a^\theta) \bullet_2 D_{xx} \vartheta^\eta(t, x) \right)
\]

\[
+ D_a f(x, a^\theta) \right|^2,
\]

on the same grid \( T \times \chi \) as before.
Another validation method, which is more graphical, consists in approximating numerically the optimal control as described in Sect. 5.1 and then computing the value function and its first derivative for some chosen points \((t, x)\) by Monte Carlo simulations

\[
\vartheta_{MC}(t, x) = \frac{1}{M} \sum_{m=1}^{M} \left[ g(x_{t_N}^{m,t,x}) + \sum_{p=n}^{N-1} f(x_{t_p}^{m,t,x}, a_\theta(t_p, x_{t_p}^{m,t,x})) \Delta t_p \right],
\]

\[
D_x \vartheta_{MC}(t, x) = \frac{1}{M} \sum_{m=1}^{M} \left[ \left( D_x x_{t_N}^{m,t,x} \right)^\top D_x g(x_{t_N}^{m,t,x}) \right.
\]

\[
+ \sum_{p=n}^{N-1} \left( \left( D_x x_{t_p}^{m,t,x} \right)^\top D_x f(x_{t_p}^{m,t,x}, a_\theta(t_p, x_{t_p}^{m,t,x})) \right.
\]

\[
+ \left( D_x a_\theta(t_p, x_{t_p}^{m,t,x}) D_x x_{t_p}^{m,t,x} \right)^\top D_a f(x_{t_p}^{m,t,x}, a_\theta(t_p, x_{t_p}^{m,t,x})) \Delta t_p \right],
\]

with \(t = t_n\). We then plot these Monte Carlo points alongside the value functions obtained using neural networks to check that the machine learning methods described in the previous sections are able to approximate the value function corresponding to the approximated optimal control \(a_\theta\).

6 Numerical examples

We illustrate on some financial examples the numerical results obtained from the implementation of the algorithms described in the previous section.

6.1 Example of Merton portfolio selection

We consider the Bellman equation

\[
\begin{cases}
\partial_t u + \sup_{\alpha \in \mathbb{R}} \left[ axb D_x u + \frac{1}{2} a^2 x^2 \sigma^2 D^2_x u \right] = 0, & (t, x) \in [0, T) \times (0, \infty), \\
u(T, x) = g(x), & x \in (0, \infty),
\end{cases}
\]

which arises from the Merton portfolio selection problem where an agent invests a proportion \(\alpha = (\alpha_t)\) of her wealth \(X = X^\alpha\) in a stock following a Black–Scholes model with rate of return \(b \in \mathbb{R}\), and constant volatility \(\sigma > 0\). The controlled wealth dynamics is then governed by

\[
dX_t = X_t \alpha_t b dt + X_t \alpha_t \sigma dW_t,
\]

and the goal of the investor is to maximize over \(\alpha\) her expected terminal wealth \(\mathbb{E}[g(X_T)]\), with \(g\) some utility function, i.e., concave and nondecreasing, on \((0, \infty)\).
When the utility function $g$ is of power type, i.e., $g(x) = x^\gamma / \gamma$, for some $\gamma < 1$, $\gamma \neq 0$, it is well known that the optimal control is constant equal to

$$\hat{a} = \frac{b}{\sigma^2 (1-\gamma)},$$

while the value function is explicitly given by

$$u(t, x) = e^{\rho(T-t)} g(x), \quad \text{with} \quad \rho = \frac{b^2}{2\sigma^2} \frac{\gamma}{1-\gamma}.$$

These closed-form expressions serve as benchmarks for comparing our results computed by the differential learning algorithms.

To check that the value function approximation obtained is a lower bound of the true one, we compute in Table 1 the difference between the closed-form value function and the estimation of the value function on points $(t, x)$ obtained by computing the expectation (2.5) by Monte Carlo on $1e^6$ trajectories controlled by the Deep learning approximation of the optimal control. We compute the value functions on a grid $t \in \{0, 0.5, 0.9\}$, $x \in \{1e^{-2}, 0.5, 0.75, 1, 1.25, 1.5, 2\}$ with parameters $b = 0.2, \sigma = 0.2$ and power utility with exponent $\gamma = 0.5$ and present in the table the difference between the closed form value and the Monte Carlo approximation. For clarity of presentation, we present the results averaged over $t$ in this table.

We compute, in Table 2, the residual losses defined in (5.2) for the NN $\vartheta^\eta$ obtained by the various deep learning methods: the differential learning scheme (Algorithm 2), the pathwise martingale learning with 1 NN (Algorithm 3), and the pathwise differential learning with 1 NN (Algorithm 4). Since two gradient steps are performed during each training epoch of the Pathwise differential learning method, we indicate the training time for 500 epochs for this method, whereas the training time of the two other methods is indicated for 1000 epochs. For each of these algorithms, 8192 starting points and Brownian trajectories are used to train the neural networks. We also provide the training time for each of these algorithms. On this table, we see that the Pathwise differential learning method is the slowest to train but yields the best results in terms of residual and terminal losses. We see that for all three methods, the difference between the residual loss only and the sum of residual and terminal loss is small, meaning that with all three methods the neural network managed to fit the terminal condition of the PDE during the training.

We plot the value function $\vartheta^\eta(t, x)$ and its derivatives $\partial_x \vartheta^\eta(t, x)$ and $\partial_{xx} \vartheta^\eta(t, x)$ for fixed values $t = 0, t = 0.5, t = 0.9$, and for parameter values $b = 0.2, \sigma = 0.2$.

### Table 1

Difference between the closed form value function and the value computed by Monte Carlo on $1e^6$ trajectories on points $x \in \{1e^{-2}, 0.5, 0.75, 1, 1.25, 1.5, 2\}$ and averaged over times $t \in \{0, 0.5, 0.9\}$ for the Merton problem with parameters $b = 0.2, \sigma = 0.2$ and power utility with exponent $\gamma = 0.5$

| $x$ | Difference closed - MC |
|-----|------------------------|
| $1e^{-2}$ | $2.699e^{-4}$ |
| $0.5$ | $1.811e^{-3}$ |
| $0.75$ | $2.208e^{-3}$ |
| $1$ | $2.543e^{-3}$ |
| $1.25$ | $2.839e^{-3}$ |
| $1.5$ | $3.106e^{-3}$ |
| $2$ | $3.582e^{-3}$ |
Table 2  Residual and boundary losses computed on a 1002x1002 time and space grid with $t \in [0, 0.9]$ and $x \in [1e^{-2}, 2]$ for the Merton problem with parameters $b = 0.2, \sigma = 0.2,$ and power utility with exponent $\gamma = 0.5$

|                      | Diff. regr. learning | Path. 1NN | Path. diff. 1NN |
|----------------------|----------------------|-----------|-----------------|
| Residual loss        | $1.538e^{-1}$        | $1.752e^{-1}$ | $7.872e^{-2}$   |
| Residual loss + terminal loss | $1.548e^{-1}$ | $1.758e^{-1}$ | $7.894e^{-2}$   |
| Training time        | 274 s                | 297 s     | 525 s           |

Fig. 3  Value function $\vartheta^\eta$ and its first and second derivative obtained by Differential regression Learning (Algorithm 2) for the Merton problem with parameters $b = 0.2, \sigma = 0.2,$ and power utility with exponent $\gamma = 0.5$, plotted as functions of $x$, for fixed values of $t$

$\gamma = 0.5$, and compare it with the closed-form solution of the problem. Figure 3 corresponds to the Differential regression learning method, Fig. 4 corresponds to the pathwise martingale learning, while Fig. 5 corresponds to the pathwise differential learning method. These graphs are coherent with the results presented in Table 2. We see that the Pathwise differential provides the best fit, in particular for the first and second derivatives of the solution, followed by the Differential regression method. The Pathwise method provides a good fit for the value of the PDE solution but does not manage to fit very well its first and second derivatives for small values of $x$.  

\( \vartheta^\eta \)
Fig. 4 Value function $\vartheta$ and its first and second derivative obtained by Pathwise learning (Algorithm 3) for the Merton problem with parameters $b = 0.2$, $\sigma = 0.2$, and power utility with exponent $\gamma = 0.5$, plotted as functions of $x$, for fixed values of $t$.

6.2 A multi-dimensional portfolio selection problem

We consider a multi-dimensional extension of the Merton problem where an agent can invest an amount $\alpha$ of her wealth in $n$ stocks with price process governed by a stochastic volatility model modeled by an Ornstein–Uhlenbeck process. This leads to a Bellman equation on $[0, T] \times \mathbb{R} \times \mathbb{R}^n$ (hence in dimension $d = n + 1$) in the form

$$
\begin{align*}
\partial_t u + \sup_{(a_i)_i \in \mathbb{R}^n} \left\{ \sum_{i=1}^n a_i \sigma(v_i) \lambda_i(v_i) D_{x_i} u + \frac{1}{2} a_i^2 \sigma^2(v_i) D_{x_i}^2 u \\
+ \kappa_i (\theta_i - v_i) D_{v_i} u + \frac{1}{2} \nu_i^2 D_{v_i}^2 u \right\} &= 0, \\
\forall (t, x, v = (v_i)_i) \in [0, T] \times \mathbb{R} \times \mathbb{R}^n,
\end{align*}
$$

for $(t, x, v = (v_i)_i) \in [0, T] \times \mathbb{R} \times \mathbb{R}^n$. Here, $\sigma(.)$ is a positive volatility coefficient, e.g., $\sigma(v) = e^v$, $\lambda_i(\cdot)$, $i = 1, \ldots, n$, are the risk premia of the stock, and $\kappa_i, \theta_i, \nu_i, i = 1, \ldots, n$, are positive constants. Notice that we assumed here for simplicity that there is no correlation between stocks and volatility.

We shall test our algorithms in the case of an exponential utility function $g(x) = -e^{-\eta x}$, with $\eta > 0$, for which the optimal feedback control is explicitly given by $\hat{a} = (\hat{a}_i)_i$ with
Fig. 5 Value function $\vartheta^H$ and its first and second derivative obtained by Pathwise differential learning (Algorithm 4) for the Merton problem with parameters $b = 0.2$, $\sigma = 0.2$, and power utility with exponent $\gamma = 0.5$, plotted as functions of $x$, for fixed values of $t$.

\[
\hat{a}_i(t, x, v) = \frac{\lambda_i(v_i)}{\eta \sigma(v_i)}, \quad (t, x, v = (v_i)_i) \in [0, T] \times \mathbb{R} \times \mathbb{R}^n, \quad i = 1, \ldots, n,
\]

and when the risk premia are in linear form: $\lambda_i(v_i) = \lambda_i v_i$, for $\lambda_i > 0$, for which the value function solution to the Bellman PDE is explicitly given by $u(t, x, v) = U(x)w(t, v)$ with

\[
w(t, v) = \exp \left( - \sum_{i=1}^n \left[ \frac{\phi_i(t) v_i^2}{2} + \psi_i(t) v_i + \chi_i(t) \right] \right), \quad (t, v) \in [0, T] \times \mathbb{R}^n,
\]

\[
\phi_i(t) = \lambda_i^2 \frac{\sinh(\hat{k}_i(T - t))}{\rho_i(t)}, \quad \psi_i(t) = \lambda_i^2 \frac{\kappa_i \theta_i}{\hat{k}_i} \frac{\cosh(\hat{k}_i(T - t)) - 1}{\rho_i(t)}, \quad \chi_i(t) = \frac{1}{2} \ln \frac{\rho_i(t)}{\hat{k}_i} - \frac{1}{2} \kappa_i(T - t)
\]

\[
- \lambda_i^2 \frac{(\kappa_i \theta_i)^2}{\hat{k}_i^2} \left[ \frac{\sinh(\hat{k}_i(T - t))}{\rho_i(t)} - (T - t) \right] - \lambda_i^2 \frac{(\kappa_i \theta_i)^2 \kappa_i}{\hat{k}_i^3} \frac{\cosh(\hat{k}_i(T - t)) - 1}{\rho_i(t)},
\]
with \( \rho_i(t) = \kappa_i \sinh(\hat{k}_i(T - t)) + \hat{k}_i \cosh(\hat{k}_i(T - t)) \), and \( \hat{k}_i = \sqrt{\kappa_i^2 + \nu_i^2} \).

We present the results obtained in Table 3 and Figs. 6 and 7. We solve numerically this problem using the Differential regression learning method for the following numbers of assets and parameter values:

- **Four assets** \((n = 4, d = 5)\), \(\lambda = (1.5 1.1 2.0.8), \theta = (0.1 0.2 0.3 0.4), \nu = (0.2 0.15 0.25 0.31), \kappa = (1.0 1.8 1.1 1.3), \eta = 0.5\),
- **Seven assets** \((n = 7, d = 8)\), \(\lambda = (1.5 1.1 2.0.8 0.5 1.7 0.9), \theta = (0.1 0.2 0.3 0.4 0.25 0.15 0.18), \nu = (0.2 0.15 0.25 0.31 0.4 0.35 0.22), \kappa = (1.0 1.8 1.1 1.3 0.95 0.99 1.02), \eta = 0.5\).

To compute these solutions, we used neural networks with the same architecture as before, represented in Fig. 2, with an additional sub-network taking the stochastic volatility \(v\) as an input. We used a network with 20 neurons per layer using \(\tanh\) activation function to approximate the optimal control and a network with 50 neurons per layer using \(\text{ELU}\) activation function to approximate the PDE solution. To solve this multi-dimensional problem involving two independent Brownian motions, we use a slightly modified version of the Differential regression learning Algorithm 2. In addition to the loss associated with the value and the one associated with the derivative of the PDE solution w.r.t the variable \(x\), we consider a third loss associated to the derivatives of the PDE solution w.r.t each component of the stochastic volatility \(v\).

More precisely, during the training, instead of generating a batch of the two Brownian motions and training the network on this same batch for all epochs, we generate the multi-dimensional Brownian motion associated with the stochastic volatility one at the start of the training, and we generate a new batch of the Brownian motion associated with the assets every three epochs. We chose to generate new Brownian motions every three epochs to make a gradient step on each component of the loss with the same realizations of Brownian motions.

In Table 3, we show the total loss (sum of residual and terminal losses) of the network computed on a 102x102 grid of variables \((t, x)\), for fixed volatility value \(v = \theta\). We also show the relative error between the value function estimated by the neural network and the solution in closed form. We compute this relative error on the point \((t = 0, x = 1, v = \theta)\) in the case of four and seven assets.

In Figs. 6 and 7, we represent the value, first and second derivatives of the network and of the closed form solution, w.r.t the variable \(x\), for fixed values of \(t\) and \(v = \theta\). On these graphs, we can see that the solution obtained tends to fit the true solution better as time \(t\) approaches the terminal time \(T = 1\) and that the value and derivatives are better approached for values of \(x > 1\). In our numerical tests, augmenting the dimension from four to seven does not impact the accuracy of this numerical method for the estimation of the value and first derivative of the PDE solution. Although the approximation for the value function is still good, we note that the errors for the derivatives are less accurate compared to the one-dimensional case.
Table 3 Residual and boundary losses computed on a 102x102 time and space grid with \( t \in [0, 0.9] \), \( x \in [0, 2] \) and \( v = \theta \), and relative error between the network approximation and the closed form solution computed at point \((t = 0, x = 1, v = \theta)\) for the multi-dimensional Merton problem with stochastic volatility with terminal function \( g(x) = -e^{-x} \)

| Total loss | Relative error \( \mathcal{V}^\eta \) (%) |
|------------|----------------------------------|
| \( n = 4 \) | \( 1.056e^{-2} \) | 1.765 |
| \( n = 7 \) | \( 1.944e^{-2} \) | 1.374 |

Fig. 6 Value function \( \vartheta^\eta \) and its first and second derivative obtained by Differential regression learning (Algorithm 2) for the multi-dimensional Merton problem with stochastic volatility with \( n = 4 \) assets and terminal function \( g(x) = -e^{-x} \), plotted as functions of \( x \), for fixed values of \( t \) and volatility \( v = \theta \)

6.3 Example of the Black–Scholes model with linear market impact

We consider the option pricing problem with linear market impact as studied in Loeper (2018), which leads to a nonlinear Black–Scholes (BS) equation in the form (2.1) with \( g \) the option payoff and an Hamiltonian \( H \) given on \((0, \infty) \times \mathbb{R}\) by

\[
H(x, \gamma) = \begin{cases} 
\frac{1}{2} \sigma^2 x^2 \gamma, & \text{if } \lambda x^2 \gamma < 1 \\
\infty, & \text{otherwise,}
\end{cases} \quad (6.1)
\]

 Springer
Fig. 7 Value function $\vartheta^n$ and its first and second derivatives obtained by Differential regression learning (Algorithm 2) for the multi-dimensional Merton problem with stochastic volatility with $n = 7$ assets and terminal function $g(x) = -e^{-x}$, plotted as functions of $x$, for fixed values of $t$ and volatility $v = \theta$

where $\sigma > 0$ is the volatility in the BS model, and $\lambda$ is a nonnegative constant related to the linear market impact. Notice that $H$ can be written in Bellman form as

$$H(x, \gamma) = \sup_{a \geq 0} \left[ \frac{1}{2} ax^2 \gamma - \frac{1}{2\lambda} (\sqrt{a} - \sigma)^2 \right],$$

which corresponds to the dual stochastic control representation of the option price $u$ as

$$u(t, x) = \sup_\alpha \mathbb{E} \left[ g(X_T^{t, x, \alpha}) - \frac{1}{2\lambda} \int_t^T (\sqrt{\alpha_s} - \sigma)^2 ds \right], \quad (6.2)$$

where $X = X^{t, x, \alpha}$ is governed by the controlled dynamics

$$dX_s = X_s \sqrt{\alpha_s} dW_s, \quad t \leq s \leq T, \quad X_t = x > 0,$$

with a control process $\alpha$ valued in $\mathbb{R}_+$.  

We shall apply the various differential learning methods to this problem for two examples of option payoff.
6.3.1 Closed-form solution for a logarithmic terminal cost

We first consider the toy example where the option payoff $g$ is logarithmic: $g(x) = \ln x$. Indeed, in this case, we can check that the solution to the pricing PDE (2.1) with $H$ as in (6.1) is given in closed-form by

$$u(t, x) = \ln(x) - \frac{\sigma^2}{2(1 + \lambda)} (T - t),$$

while the optimal control to the dual stochastic control representation (6.2) is constant equal to

$$\hat{a}(t, x) = \left(\frac{\sigma}{1 + \lambda}\right)^2.$$

To check that the value function approximation obtained is a lower bound of the true one, we compute in Table 4 the difference between the closed form value function and the estimation of the value function on points $(t, x)$ obtained by computing the expectation (2.5) by Monte Carlo on $1e^6$ trajectories controlled by the Deep Learning approximation of the optimal control. We compute the value functions on a grid $t \in \{0, 0.5, 0.9\}, x \in \{1e^{-2}, 0.5, 0.75, 1, 1.25, 1.5, 2\}$ with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$ and present in the table the difference between the closed form value and the Monte Carlo approximation. For clarity of presentation, we present the results averaged over $t$ in this table.

In Table 5, we compute the residual losses defined in (5.2) for the NN $\vartheta_\eta$ obtained by the various deep learning methods: the differential learning scheme (Algorithm 2), the pathwise martingale learning with 1 NN (Algorithm 3), and the pathwise differential learning with 1 NN (Algorithm 4). We also provide the training time for 500 epochs for each of these algorithms. On this table, we see that the Pathwise learning methods yield better results than the Differential regression learning methods. The difference between the residual loss only and the sum of the residual and terminal loss is small in all three methods, meaning that both the PDE solution’s derivatives and terminal condition have been learned by the neural network.

We plot the value function $\vartheta_\eta(t, x)$ and its derivatives $\partial_x \vartheta_\eta(t, x)$ and $\partial_{xx} \vartheta_\eta(t, x)$ for fixed values $t = 0, t = 0.5, t = 0.9$, parameter $\sigma = 0.2$ and linear market impact factor $\lambda = 5e^{-3}$, and compare it with the closed-form solution of the problem. Figure 8 corresponds to the Differential regression learning method, Fig. 9 corresponds to the pathwise martingale learning, while Fig. 10 corresponds to the pathwise differential

| Table 4 | Difference between the closed form value function and the value computed by Monte Carlo on $1e^6$ trajectories on points $x \in \{1e^{-2}, 0.5, 0.75, 1, 1.25, 1.5, 2\}$ and averaged over times $t \in \{0, 0.5, 0.9\}$ for Black–Scholes problem with linear market impact factor $\lambda = 5e^{-3}$ and parameter $\sigma = 0.2$ |
|-----------------------------------------------|
| $x = 1e^{-2}$ | $x = 0.5$ | $x = 0.75$ | $x = 1$ | $x = 1.25$ | $x = 1.5$ | $x = 2$ |
| Difference closed - MC | $2.673e^{-4}$ | $2.042e^{-4}$ | $2.058e^{-4}$ | $2.066e^{-4}$ | $2.099e^{-4}$ | $2.147e^{-4}$ | $2.300e^{-4}$ |
Table 5  Residual and boundary losses computed on a 1002x1002 time and space grid with \( t \in [0, 0.9] \) and \( x \in [0.1, 2] \) for a terminal logarithmic payoff, with parameter \( \sigma = 0.2 \) and linear market impact factor \( \lambda = 5e^{-3} \).

|                      | Diff. regr. learning | Path. INN | Path. diff. INN |
|----------------------|----------------------|-----------|-----------------|
| Residual loss        | 2.046e\(^{-3}\)      | 3.484e\(^{-4}\) | 6.644e\(^{-4}\) |
| Residual loss + terminal loss | 2.179e\(^{-3}\)      | 3.864e\(^{-4}\) | 6.758e\(^{-4}\) |
| Training time (500 epochs) | 163 s               | 130 s     | 525 s           |

Fig. 8  Value function \( \vartheta_{\eta} \) and its first and second derivatives obtained by Differential Regression Learning (Algorithm 2) for a logarithmic option payoff, with parameter \( \sigma = 0.2 \) and linear market impact factor \( \lambda = 5e^{-3} \), plotted as functions of \( x \), for fixed values of \( t \).

Learning method. On these graphs, we can see that the three methods, as well as the Monte Carlo values \( \vartheta_{MC} \) and \( D_x \vartheta_{MC} \), yield good approximations of the PDE solution and its derivatives. Notice, however, that the pathwise learning (see Fig. 9) does not provide a good approximation of the second derivative on the boundary points of the grid, namely the points that were not explored by the simulations, but when combining with the differential learning (see Fig. 10), it greatly improves the approximation of the second derivative.
Fig. 9 Value function $\varphi^\eta$ and its first and second derivatives obtained by Pathwise Learning (Algorithm 3) for a logarithmic option payoff, with parameter $\sigma = 0.2$ and linear market impact factor $\lambda = 5 e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

6.3.2 Call-option type terminal condition

We now consider a usual call option payoff with strike $K = 1$, and hence, a function $g$ is equal to: $g(x) = \max(x - 1, 0)$.

Again, we compute in Table 6 the residual losses defined in (5.2) for the NN $\varphi^\eta$ obtained by the various deep learning methods: the differential learning scheme (Algorithm 2), the pathwise martingale learning with 1 NN (Algorithm 3), and the pathwise differential learning with 1 NN (Algorithm 4). We also provide the training time for each of these algorithms. On this table, we see that the Differential regression learning and the Pathwise differential methods yield better results than the “simple” Pathwise method. Despite having the lowest residual loss, the Pathwise method gives the biggest terminal loss. This shows that, while the time and second space derivatives of the neural network give a low residual loss corresponding to the Hamiltonian (6.1), the network does not manage to fit the terminal function. This phenomenon is also present, to a lesser extent, in the approximation given by the Differential regression learning method. Out of the three methods, the Pathwise differential yields the smallest residual and terminal losses.

We plot again the value function $\varphi^\eta(t, x)$ and its derivative $\partial_x \varphi^\eta(t, x)$ for fixed values $t = 0, t = 0.5, t = 0.9$, parameter $\sigma = 0.3$, and linear market impact.
Fig. 10 Value function $\vartheta^I$ and its first and second derivatives obtained by Pathwise Differential Learning (Algorithm 4) for a logarithmic option payoff, with parameter $\sigma = 0.2$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

Table 6 Residual and boundary losses computed on a $1002 \times 1002$ time and space grid with $t \in [0, 0.9]$ and $x \in [0.1, 2]$ for a terminal call-option payoff $g(x) = \max(x - 1, 0)$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$.

|                          | Diff. regr. learning | Path. 1NN | Path. diff. 1NN |
|--------------------------|----------------------|-----------|-----------------|
| Residual loss            | $2.998e^{-4}$        | $2.756e^{-4}$ | $2.283e^{-4}$   |
| Residual loss + terminal loss | $3.972e^{-4}$ | $1.004e^{-3}$    | $2.538e^{-4}$   |
| Training time            | $262$ s              | $299$ s    | $584$ s         |
|                          | $1000$ epochs        | $1000$ epochs | $500$ epochs    |

factor $\lambda = 5e^{-3}$, and compare it with the Monte Carlo estimation obtained. Figure 11 corresponds to the Differential regression learning method, and Fig. 12 corresponds to the pathwise martingale learning, while Fig. 13 corresponds to the Pathwise differential learning method. Graphically, the results of the Differential regression learning and the Pathwise differential learning methods are very close to the points obtained by Monte Carlo estimation for the value and the first derivative. The Pathwise methods do not give a good approximation of the value and the derivatives for values of $x$ smaller than 1. The difference of performance between the Pathwise and the Differential pathwise methods is analogous to the one observed between Differential regression learning and
Fig. 11 Value function $\vartheta^q$ and its derivative obtained by Differential Regression Learning (Algorithm 2) for a call option with strike 1, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

Fig. 12 Value function $\vartheta^q$ and its derivative obtained by Pathwise Learning (Algorithm 3) for a call option with strike 1, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

“simple” regression learning in Fig. 1, demonstrating the interest of adding a regression term for the derivative of the neural network.
In line with the works in Vidales et al. (2018), Glau and Wunderlich (2020), Remlinger et al. (2022), our next goal is to design a Machine Learning method allowing us to directly obtain a solution of problem (2.3) for a parametric terminal condition $g_K$, for every value of the parameter $K \in \mathbb{R}^P$ in a compact set. In other words, we aim to learn the operator that maps the payoff function parameter $K$ to the solution of the PDE with terminal condition $g_K$. We want to be able to train the neural networks once and for all on a selection of parameter values and obtain a network which takes a couple $(t, x)$ and the parameter value $K$ and outputs the solution of the problem. In Glau and Wunderlich (2020), the authors solve parametric PDEs using a variant of highway networks (Srivastava et al., 2015), which are feed-forward networks where each dense layer has an additional parameter called gate which allows the layer to output a combination of the unmodified input and of the output of the affine and activation operations, alleviating the vanishing gradient problem and allowing to train deeper networks. Their network takes the time, space, and PDE parameter as input and is trained in the spirit of the Deep Galerkin method. The PDE residuals is computed from the neural network on a random time, space, and parameter grid and is minimized to approximate the PDE solution. In Vidales et al. (2018), the authors give four methods to solve parametric PDEs using a fully connected network taking the time, space, and PDE parameter as input and minimizing a loss averaged over random parameter values. In this article, we follow the methodology developed in Remlinger et al. (2022) by relying on a class of neural networks, called DeepONet, presented in Lu et al. (2019), and aiming to approximate functional operators. This method is based

7 Further step: resolution for parametric terminal functions

7.1 Theory and network structure
on the following universal approximation theorem for operator, due to Chen and Chen (1995).

**Theorem 7.1** Suppose that $\sigma$ is a continuous non-polynomial function, $X$ is a Banach space, $K_1 \subset X$, $K_2 \subset \mathbb{R}^d$ are two compact sets in $X$ and $\mathbb{R}^d$, respectively, $V$ is a compact set in $C(K_1)$, $G$ is a nonlinear continuous operator, which maps $V$ into $C(K_2)$. Then, for any $\epsilon > 0$, there are positive integers $n$, $p$, $m$, constants $c_{i}^{k}$, $\xi_{ij}^{k}$, $\theta_{i}^{k}$, $\zeta_{k} \in \mathbb{R}$, $w_{k} \in \mathbb{R}^{d}$, $x_{j} \in K_{1}$, $i = 1, ..., n$, $k = 1, ..., p$, $j = 1, ..., m$, such that

$$
\left\| G(u)(y) - \sum_{k=1}^{p} \sum_{i=1}^{n} c_{i}^{k} \sigma \left( \sum_{j=1}^{m} \xi_{ij}^{k} u(x_{j}) + \theta_{i}^{k} \right) \sigma \left( w_{k} y + \zeta_{k} \right) \right\| < \epsilon,
$$

holds for all $u \in V$ and $y \in K_2$.

The network used in Lu et al. (2019) is composed of two sub networks, the branch net, which takes the terminal function estimated on a fixed number of points called sensors as input, and the trunk net, which takes the time and space coordinates as input. In our case, as in Remlinger et al. (2022), the branch net takes the parametric terminal function estimated on a grid of sensors, and will be trained for random values of the function’s parameter. We represent the structure of this neural network in Fig. 14.

As our numerical methods proceed in two stages, by first approximating the optimal control of problem (2.3) with a neural network and then approximating the associated value function with another network, we again use a DeepONet to approximate the
optimal control and then another one to approximate the value function \( u \) by differential learning.

While the authors of Lu et al. (2019) train a DeepONet in a supervised manner, we train our network in an **unsupervised way**.

As before, we start by training the control network \( a_\theta \) in order to approximate the optimal control of problem. We use a batch of \( M \) independent trajectories \( \{x^m_{t_n}, K_m, \theta\} \), \( t_n \in T_N \), \( m = 1, \ldots, M \), of \( \{X^K_{t_n}, t_n \in T_N\} \), where the \( K \) superscript denotes that the trajectory is driven by a control with input parameter \( K \), and \( M \) random parameter values \( K_m \) randomly sampled from a distribution \( \mu_K \) with compact support in \( \mathbb{R}^P \), and apply a stochastic gradient ascent method to the empirical gain function

\[
J_M(\theta) = \frac{1}{M} \sum_{m=1}^{M} \left[ g(x^m_T, K_m, \theta) + \sum_{n=0}^{N-1} f(x^m_{t_n}, K_m, \theta, a_\theta(t_n, x^m_{t_n}, K_m, K_m)) \delta t_n \right].
\]

The pseudo-code is described in Algorithm 5. The output of this algorithm yields a parameter \( \theta^* \), and so an approximation of the optimal feedback control with \( a^*_\theta = a_\theta^* \), and of the associated optimal state process with \( X^* = X^{\theta^*} \). In the sequel, to alleviate notations, we shall omit the superscript \( * \), and simply denote \( a \) and \( X \).

**Algorithm 5:** Deep learning scheme to solve the stochastic control problem (2.3)

- **Result:** A set of optimized parameters \( \theta^* \);
- Initialize the learning rate \( l \) and the neural network \( a_\theta \);
- Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_0 < t_1 < \ldots < t_N = T \) with time steps \( \Delta t_n \);
- Generate a batch of \( M \) starting points \( X_0 \sim \mu_0 \), Brownian increments \( \Delta W_{t_n} \) in \( \mathbb{R}^d \) and parameter values \( K = \mu_K \);
  - for each batch element \( m \) do
    - Compute the trajectory \( (x^m_{t_n}, K_m, \theta)_{n=0, \ldots, N} \) through the scheme
      \[
      x^m_{t_{n+1}} = x^m_{t_n} + b(x^m_{t_n}, K_m, \theta, u_0(t_n, x^m_{t_n}, K_m)) \Delta t_n + \sigma(x^m_{t_n}, K_m, \theta, u_0(t_n, x^m_{t_n}, K_m)) \Delta W_{t_n},
      \]
    - Compute the trajectory from the generated starting point \( x^m_{t_0} \), Brownian increments \( \Delta W_{t_n} \) and parameter \( K_m \);
  - end
- for each epoch do
  - Compute the batch loss
    \[
    J_M(\theta) = \frac{1}{M} \sum_{m=1}^{M} \left[ g(x^m_T, K_m, \theta) + \sum_{n=0}^{N-1} f(x^m_{t_n}, K_m, \theta, a_\theta(t_n, x^m_{t_n}, K_m, K_m)) \delta t_n \right]
    \]
  - Compute the gradients \( \nabla_\theta J_M(\theta) \);
  - Update \( \theta \leftarrow \theta - l \nabla_\theta J_M(\theta) \);
  - end
- **Return:** The set of optimized parameters \( \theta^* \);
From this optimal control approximation, the value function is then approximated through the Differential regression learning algorithm presented in Sect. 5.2 modified to train the network for different values of the terminal function parameter $K$. The target payoff and its derivative are then computed as

$$Y^K_{m,n} = g(X^K_{m,n}, K_m) + \sum_{q=n}^{N-1} f^a(t_q, X^K_{m,n}) \Delta t_q, \quad n = 0, \ldots, N,$$

$$Z^K_{m,n} = (DX X^K_{m,n})^\top D_x g(X^K_{m,n}, K_m) + \sum_{q=n}^{N-1} (DX X^K_{m,n})^\top D_x f^a(t_q, X^K_{m,n}) \Delta t_p,$$

with the convention that the above sum over $q$ is zero when $n = N$. For the training of the neural network $\vartheta^\eta$, we use a batch of $M$ independent samples $(x_{m,n}, y^K_{m,n}, z^K_{m,n})$, $m = 1, \ldots, M$, of $(X^K_{m,n}, Y^K_{m,n}, Z^K_{m,n})$, $n = 0, \ldots, N$ and $M$ random parameter values $K^m$ randomly sampled from a distribution $\mu_{K}$ with compact support in $\mathbb{R}^P$, and apply stochastic gradient descent for the minimization of the mean squared error functions

$$MSE_{val}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} \frac{1}{\Delta t_n} |y_{m,K^m,n} - \vartheta^\eta(t_n, x_{m,K^m,n}, K_m)|^2 \Delta t_n,$$

$$MSE_{der}(\eta) = \frac{1}{M} \sum_{m=1}^{M} \sum_{n=0}^{N-1} \frac{1}{\|z^K_{m,n}\|^2} |z^K_{m,K^m,n} - D_x \vartheta^\eta(t_n, x_{m,K^m,n}, K_m)|^2 \Delta t_n.$$

The pseudo-code is described in Algorithm 6.
Algorithm 6: Deep learning scheme for Differential regression learning

**Result:** A set of optimized parameters $\eta^*$.

Initialize the learning rate $\theta$; the neural networks $\theta$; $\eta$; 
Generate an $[0, T]^{N+1}$-valued time grid $0 = t_0 < t_1 < \ldots < t_N = T$ with time steps $(\Delta t_n)_{n=0,\ldots,N-1}$; 
Generate a batch of $M$ starting points $X_0 \sim \mu_0$, Brownian increments $(\Delta W_{tn})_{n=0,\ldots,N}$ in $\mathbb{R}^d$ and parameter values $K \sim \mu_K$; 

for each batch element $m$

Compute the trajectory $(x_{tn}^{m,K_m})_{n=0,\ldots,N}$ through the scheme

$$x_{tn+1}^{m,K_m} = x_{tn}^{m,K_m} + b^n(t_n, x_{tn}^{m,K_m}) \Delta t_n + \sigma^n(t_n, x_{tn}^{m,K_m}) \Delta W_{tn}^m,$$

from the generated starting point $x_{t_0}^{m,K_m}$, Brownian increments $(\Delta W_{tn}^m)_{n=0,\ldots,N-1}$, parameter $K_m$ and previously trained control $a = a_0^*$; 

Compute the value and derivative targets $(y_T^{m,K_m,t_n})_{n=0,\ldots,N}$ and $(z_T^{m,K_m,t_n})_{n=0,\ldots,N}$; 

end

for each epoch do

If Epoch number is even then

Compute, for every batch element $m$, the integral $\sum_{n=0}^{N-1} (y_T^{m,K_m,t_n} - \theta \eta(t_n, x_{tn}^{m,K_m}, K_m))^2 \Delta t_n$; 

Compute the batch loss $\text{MSE}_{\text{val}}(\eta)$; 

Compute the gradient $\nabla_{\eta} \text{MSE}_{\text{val}}(\eta)$; 

Update $\eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{val}}(\eta)$; 

end

else

Compute, for every batch element $m$, the integral $\sum_{n=0}^{N-1} (z_T^{m,K_m,t_n} - D_x \theta \eta(t_n, x_{tn}^{m,K_m}, K_m))^2 \Delta t_n$; 

Compute the batch loss $\text{MSE}_{\text{der}}(\eta)$; 

Compute the gradient $\nabla_{\eta} \text{MSE}_{\text{der}}(\eta)$; 

Update $\eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{der}}(\eta)$; 

end

end

Return: The set of optimized parameters $\eta^*$.

7.2 Application to the Black–Scholes model with linear market impact

Using this DeepONet based algorithm, we revisit the resolution of the nonlinear Black–Scholes equation presented in Sect. 6.3. In this section, we use the Algorithms 5 and 6 to solve the PDE for a terminal function corresponding to a call option payoff $g(x, K) = \max(x - K, 0)$ with parameter (or strike) $K \in \mathbb{R}^*_+$. 

The branch net of the DeepONet used to approximate the control as a function of the terminal function $g_K$ is a standard feed-forward network composed of two layers with 50 neurons and use the $\tanh$ activation function. The trunk net has the same structure as the network used in the previous sections and represented in Fig. 2. It is composed of two sub-networks taking, respectively, the time $t$ and state $x$ as an input and each composed of two layers of 50 neurons using the $\tanh$ activation function. The outputs of these two sub-networks are concatenated into a vector of $\mathbb{R}^{100}$ which passes through two additional layers of 50 neurons using $\tanh$ activation. The outputs of the branch net and the trunk net, which have the same dimension, are then combined through a dot product whose output passes through a layer of one neuron using $\tanh$ activation and a layer of one neuron using the Parametric ReLU activation function ensuring, as explained in Sect. 5.1, that the control obtained belongs to the control space $A$. 

\begin{algorithm}[H]

\textbf{Result:} A set of optimized parameters $\eta^*$.

Initialize the learning rate $\theta$; the neural networks $\theta$; $\eta$; 
Generate an $[0, T]^{N+1}$-valued time grid $0 = t_0 < t_1 < \ldots < t_N = T$ with time steps $(\Delta t_n)_{n=0,\ldots,N-1}$; 
Generate a batch of $M$ starting points $X_0 \sim \mu_0$, Brownian increments $(\Delta W_{tn})_{n=0,\ldots,N}$ in $\mathbb{R}^d$ and parameter values $K \sim \mu_K$; 

for each batch element $m$

Compute the trajectory $(x_{tn}^{m,K_m})_{n=0,\ldots,N}$ through the scheme

$$x_{tn+1}^{m,K_m} = x_{tn}^{m,K_m} + b^n(t_n, x_{tn}^{m,K_m}) \Delta t_n + \sigma^n(t_n, x_{tn}^{m,K_m}) \Delta W_{tn}^m,$$

from the generated starting point $x_{t_0}^{m,K_m}$, Brownian increments $(\Delta W_{tn}^m)_{n=0,\ldots,N-1}$, parameter $K_m$ and previously trained control $a = a_0^*$; 

Compute the value and derivative targets $(y_T^{m,K_m,t_n})_{n=0,\ldots,N}$ and $(z_T^{m,K_m,t_n})_{n=0,\ldots,N}$; 

end

for each epoch do

If Epoch number is even then

Compute, for every batch element $m$, the integral $\sum_{n=0}^{N-1} (y_T^{m,K_m,t_n} - \theta \eta(t_n, x_{tn}^{m,K_m}, K_m))^2 \Delta t_n$; 

Compute the batch loss $\text{MSE}_{\text{val}}(\eta)$; 

Compute the gradient $\nabla_{\eta} \text{MSE}_{\text{val}}(\eta)$; 

Update $\eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{val}}(\eta)$; 

end

else

Compute, for every batch element $m$, the integral $\sum_{n=0}^{N-1} (z_T^{m,K_m,t_n} - D_x \theta \eta(t_n, x_{tn}^{m,K_m}, K_m))^2 \Delta t_n$; 

Compute the batch loss $\text{MSE}_{\text{der}}(\eta)$; 

Compute the gradient $\nabla_{\eta} \text{MSE}_{\text{der}}(\eta)$; 

Update $\eta \leftarrow \eta - \nabla_{\eta} \text{MSE}_{\text{der}}(\eta)$; 

end

end

Return: The set of optimized parameters $\eta^*$.
Fig. 15 Control $a_0$ obtained by global method with DeepONet (Algorithm 5) for a call option with strike $K = 1$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

The DeepONet used to approximate the value function $u$ shares the same structure. The branch and trunk net have the same structures as the ones used in the control DeepONet with the same number of layers and neurons per layer and with Swish activation function, defined as

$$Swish(x) = \frac{x}{1 + e^{-x}}.$$  

After the dot product, the output also passes through a layer composed of one neuron using Swish activation function and a last layer of one neuron using no activation function.

Remark 7.2 For the control DeepONet, we used the tanh activation function instead of the ELU activation used in the previous sections as we encountered loss divergences during the training of the DeepONet with ELU activation. Since the tanh activation is bounded, the problem was resolved using this function.

For the value DeepONet, we used the Swish activation function instead of the ELU activation used in the previous sections as it empirically gave better results. As the ELU function’s second derivative is discontinuous, a kink was observed on the value function’s derivative we obtained with the DeepONet. This effect was not present when performing the “simple” regression with a standard network in the previous sections, probably because the regression problem is simpler and the true value function fitted with a better accuracy. Since the Swish activation is of class $C^\infty$, we obtained better results, without kinks, using this function.

For both trainings, we use the Adam optimizer with a learning rate equal to $1e^{-3}$ and train the network on 8192 random trajectories and strike values. To test the generalization power of our method, we train the control and value neural networks on strikes randomly sampled from $\mathcal{U}([0.25, 0.75] \cup [1.5, 2])$ and test the network on strikes chosen in $[0.2, 2.1]$. We plot in Fig. 15 below the optimal control approximation obtained along for $K = 1$ along with the control approximation obtained by Algorithm 1 (denoted regular network control) which serves as a reference.
Table 7  Residual and boundary losses computed on a 102x102 time and space grid with $t \in [0, 0.9]$ and $x \in [0, 3]$ for a terminal call option payoff $g(x) = \max(x - K, 0)$ for $K \in \{0.2, 0.5, 1, 1.75, 2, 2.1\}$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$.

| $K$   | Residual loss + terminal loss |
|-------|-------------------------------|
| 0.2   | $1.918e^{-3}$                 |
| 0.5   | $1.461e^{-4}$                 |
| 1     | $1.700e^{-3}$                 |
| 1.75  | $1.162e^{-3}$                 |
| 2     | $2.002e^{-3}$                 |
| 2.1   | $1.75e^{-3}$                  |

We compute in Table 7 the residual losses defined in (5.2) for the DeepONet $\phi^\eta$ by the Differential Regression Learning scheme (Algorithm 6). We compute the residual losses on a 102x102 linearly spaced time and space grid with $t \in [0, 0.9]$ and $x \in [0, 3]$ for a terminal call option payoff for strikes $K \in \{0.2, 0.5, 1, 1.75, 2, 2.1\}$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$.

We plot below the value function $\phi^\eta(t, x)$ and its derivative $\partial_x \phi^\eta(t, x)$ obtained by Differential Regression Learning with DeepONet networks, for fixed values $t = 0$, $t = 0.5$, $t = 0.9$, parameter $\sigma = 0.3$, and linear market impact factor $\lambda = 5e^{-3}$, and compare it with the Monte Carlo estimation obtained. We plot these value functions for strike values $K \in \{0.5, 2\}$ inside the training domain and $K \in \{0.2, 1, 2.1\}$ outside the training domain.

On these graphs, we see that the estimation of the value of the value function is good and consistent with the Monte Carlo estimator for every strike, inside or outside of the training domain. The estimation of the derivative of the value function is not as good and we can see on the graphs that we get the worst results for the value of the strike closest to zero, $K = 0.2$ (Fig. 18), and for values of the strike out of the training domain, $K = 1$ (Fig. 19) and $K = 2.1$ (Fig. 20).
Fig. 17 Value function $\vartheta^m$ (first line) and its derivative (second line) obtained by Differential Regression Learning (Algorithm 6) for a terminal call option payoff with strike $K = 2$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

Fig. 18 Value function $\vartheta^m$ (first line) and its derivative (second line) obtained by Differential regression learning (Algorithm 6) for a terminal call option payoff with strike $K = 0.2$, with parameter $\sigma = 0.3$ and linear market impact factor $\lambda = 5e^{-3}$, plotted as functions of $x$, for fixed values of $t$.

8 Conclusion

We have presented three distinct algorithms, respectively, called Differential regression learning, Pathwise learning, and Differential pathwise learning. The goal of these algorithms is to approximate the value, the first and the second derivatives of the solution to fully nonlinear partial differential equations of the form (1.1). These methods can be implemented using one neural network to approximate the value of the PDE solution $u$ and the networks derivatives, computed by automatic differentiation for the approximation of the derivatives of $u$. In this article, this version of the algorithms...
was implemented, and numerical tests showed that our algorithms yield an accurate estimation of the PDE solution $u$ and its first two derivatives on the entire domain of interest. In our tests, the differential pathwise learning method was the most accurate but the slowest in terms of training time. The Differential regression learning method gave the second best approximation of the solution and was the fastest method to train. The Pathwise learning method was the second fastest method but the less accurate of the three, making this method the less interesting.

While the focus of our article was to present methods with a fast convergence, which allows us to obtain a good estimation of the PDE solution and its derivative using a
small number of samples, another version of our algorithms, using one neural network to approach the value of \( u \) and two others to approach its first and second derivatives is presented in Appendix. Although the version using multiple neural networks has a slower empirical convergence, and was less accurate in the case where a small number of training samples were used, it provides more flexibility and has theoretically more chance to achieve a null loss function in the case where the number of training samples grows.

Furthermore, our Differential regression learning algorithm was tested in a version using DeepOnet neural networks that allows us to solve nonlinear PDEs with parametric terminal condition for values of the parameter in a compact set. Numerical tests showed that this algorithm yields good estimations of the PDE solution’s value and derivatives for parameters drawn inside the parameter training set and has a good generalization power when tested on parameter values outside, but close to, the training set. A more complete investigation of DeepOnet for solving families of fully nonlinear PDEs with differential regression learning algorithms is postponed for further research.

### A Alternative algorithms using multiple neural networks

We present below Algorithm 7, which is the version of Algorithm 3 using two neural networks, described in Sect. 5.3.

**Algorithm 7:** Deep learning scheme for Pathwise martingale learning with 2 NN

**Result:** A set of optimized parameters \( \eta^*, \delta^* \);

1. Initialize the learning rate \( l \), the neural networks \( \vartheta^{\eta}, Z^{\delta} \);
2. Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_0 < t_1 < \ldots < t_N = T \) with time steps \( (\Delta t_n)_{n=0,\ldots,N-1} \);
3. Generate a batch of \( M \) starting points \( X_0 \sim \mu_0 \) and Brownian increments \( (\Delta W_{tn})_{n=0,\ldots,N-1} \in \mathbb{R}^d \);

   **for each batch element \( m \) do**
   
   - Compute the trajectory \( (x_{tn}^m)_{n=0,\ldots,N} \) through the scheme
     
     \[
     x_{tn+1}^m = x_{tn}^m + b^a(t_n, x_{tn}^m) \Delta t_n + \sigma^a(t_n, x_{tn}^m) \Delta W_{tn}^m,
     \]
   
   - from the generated starting point \( x_{t_0}^m \), Brownian increments \( (\Delta W_{tn}^m)_{n=0,\ldots,N-1} \) and previously trained control \( a = a^{\theta^*} \);
   
   - Compute the value target \( (y_{T}^{m,t_n})_{n=0,\ldots,N} \);

   **end**

   **for each epoch do**
   
   - Compute, for every batch element \( m \), the integral
     
     \[
     \sum_{n=0}^{N-1} \left| y_{T}^{m,t_n} - \vartheta^{\eta}(t_n, x_{tn}^m) - \sum_{p=n}^{N-1} (Z^{\delta}(t_p, x_{tp}^m))^{\top} \sigma^w(t_p, x_{tp}^m) \Delta W_{tp}^m \right|^2 \Delta t_n;
     \]
   
   - Compute the batch loss \( \tilde{MSE}_{mar}(\eta, \delta) \);
   
   - Compute the gradient \( \nabla_{\eta} \tilde{MSE}_{mar}(\eta, \delta) \) and \( \nabla_{\delta} \tilde{MSE}_{mar}(\eta, \delta) \);
   
   - Update \( \eta \leftarrow \eta - l \nabla_{\eta} \tilde{MSE}_{mar}(\eta, \delta), \delta \leftarrow \delta - l \nabla_{\delta} \tilde{MSE}_{mar}(\eta, \delta) \);

   **end**

**Return:** The set of optimized parameters \( \eta^*, \delta^* \);
In the same way, Algorithm 8 below is the version of Algorithm 4 using three neural networks described in Sect. 5.3.

**Algorithm 8:** Deep learning scheme for Pathwise differential learning with 3 NN

**Result:** A set of optimized parameters \( \eta^* \);

Initialize the learning rate \( l \), the neural networks \( \theta^n \);

Generate an \( \mathbb{R}^{N+1} \)-valued time grid \( 0 = t_0 < t_1 < \ldots < t_N = T \) with time steps \( (\Delta t_n)_{n=0, \ldots, N-1} \);

Generate a batch of \( M \) starting points \( X_0 \sim \mu_0 \) and Brownian increments \( (\Delta W_{tn})_{n=0, \ldots, N} \in \mathbb{R}^d \);

for each batch element \( m \) do

Compute the trajectory \( (x^n_T)_{n=0, \ldots, N} \) through the scheme

\[
x_{n+1} = x_n + \Delta x_n \Delta t_n + \sum_{m=0}^{N-1} \Delta x_m \Delta W_{tn}.
\]

from the generated starting point \( x^n_{T0} \), Brownian increments \( (\Delta W_{tn})_{n=0, \ldots, N-1} \) and previously trained control \( a = a_{q*} \);

Compute the value and derivative targets \((x^n_T, \eta^*_{n})_{n=0, \ldots, N} \) and \((x^n_{T0}, \eta^*_{n})_{n=0, \ldots, N} \);

end

for each epoch do

Compute, for every batch element \( m \), the integral

\[
\sum_{n=0}^{N-1} \frac{1}{2} \left| \frac{\partial \psi^*_{m,n}}{\partial t} \left( t_n, x^n_{Tm} \right) - \psi^*_{m,n} \right|^2 \Delta t_n.
\]

Compute the batch loss \( MSE_{mar}(\eta) \);

Compute the gradient \( \nabla \eta MSE_{mar}(\eta) \);

Update \( \eta \leftarrow \eta - \nabla \eta MSE_{mar}(\eta) \);

Compute, for every batch element \( m \), the integral

\[
\sum_{n=0}^{N-1} \frac{1}{2} \left| \left( \frac{\partial \psi^*_{m,n}}{\partial x} \right)_{t_n, x^n_{Tm}} + \Delta x_m \Delta W_{tn} \right|^2 \Delta t_n.
\]

Compute the batch loss \( MSE_{dermar}(\eta) \);

Compute the gradient \( \nabla \eta MSE_{dermar}(\eta) \);

Update \( \eta \leftarrow \eta - \nabla \eta MSE_{dermar}(\eta) \);

end

Return: The set of optimized parameters \( \eta^* \);

References

Beck, C., Hutzenthaler, M., Jentzen, A., & Kuckuck, B. (2020). An overview on deep learning-based approximation methods for partial differential equations. arXiv:2012.12348.

Beck, C., Becker, S., Cheridito, P., Jentzen, A., & Neufeld, A. (2021). Deep splitting method for parabolic PDEs. *SIAM Journal on Scientific Computing*, 43(5), A3135–A3154.

Beck, C., E. W., & Jentzen, A. (2019). Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *J. Nonlinear Sci.*, 29(4), 1563–1619.

Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(2).

Chen, T., & Chen, H. (1995). Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems. *IEEE Transactions on Neural Networks*, 6(4), 911–917.

El Karoui, N., Quenez, M.-C., & Peng, S. (1997). Backward stochastic differential applications in finance. *Mathematical Finance*, 7(1), 1–71.

Germain, M., Pham, H., & Warin, X. (2021). Neural networks-based algorithms for stochastic control and PDEs in finance. to appear in *Machine learning for financial markets: a guide to contemporary practices*.

Glasserman, P. (2013). *Monte Carlo methods in financial engineering* (Vol. 53). Springer.
Glau, K., & Wunderlich, L. (2020). The deep parametric PDE method: Application to option pricing. 
arXiv:2012.06211.

Gobet, E., & Munos, R. (2005). Sensitivity analysis using Itô-Malliavin calculus and martingales, and 
an application to stochastic optimal control. SIAM Journal on Control and Optimization, 43(5), 1676–1713.

Han, Jiequn, & Weinan, E. (2016). Deep learning approximation for stochastic control problems. Deep 
Reinforcement Learning Workshop, NIPS, arXiv:1611.07422.

Han, J., Jentzen, A., & E, Weinan. (2017). Solving high-dimensional partial differential equations using 
deep learning. Proceedings of National Academic Science USA, 115.

Hu, R., & Laurière, M. (2022). Recent developments in machine learning methods for stochastic control 
and games. hal-03656245, version 1.

Huge, B. N., & Savine, A. (2020). Differential machine learning. Available at SSRN 3591734.

Huré, C., Pham, H., Bachouch, A., & Langrené, N. (2021). Deep neural networks algorithms for stochastic 
control problems on finite horizon: Convergence analysis. SIAM Journal on Numerical Analysis, 59(1), 
525–557.

Huré, C., Pham, H., & Warin, X. (2020). Deep backward schemes for high-dimensional nonlinear PDEs. 
Mathematics of Computation, 89(324), 1547–1579.

Ji, S., Peng, S., Peng, Y., & Zhang, X. (2020). Three algorithms for solving high-dimensional fully coupled 
FBSDE through deep learning. IEEE Intelligent Systems, 35(3), 71–84.

Loeper, G. (2018). Option pricing with linear market impact and nonlinear Black-Scholes equations. 
Annals of Applied Probability, 28(5), 2664–2726.

Longstaff, F. A., & Schwartz, E. S. (2001). Valuing American options by simulation: A simple least-squares 
approach. The Review of Financial Studies, 14(1), 113–147.

Lu, L., Jin, P., & Karniadakis, G. E. (2019). Deeponet: Learning nonlinear operators for identifying 
differential equations based on the universal approximation theorem of operators. arXiv:1910.03193.

Ma, J., & Zhang, J. (2002). Representation theorems for backward stochastic differential equations. Annals 
of Applied Probability, 12(4), 1390–1418.

Mete Soner, H., Touzi, N., & Zhang, J. (2013). Dual formulation of second order target problems. The 
Annals of Applied Probability, 23(1), 308–347.

Negyesi, B., Andersson, K., & Oosterlee, C. (2021). The one step Malliavin scheme: new discretization of 
bsdes implemented with deep learning regressions. arXiv:2110.05421.

Nualart, D. (1995). The Malliavin calculus and related topics. Springer.

Nüskens, N., & Richter, L. (2021). Interpolating between BSDEs and PINNs: deep learning for elliptic and 
parabolic boundary value problems. arXiv:2112.03749.

Pham, H., Warin, X., & Germain, M. (2021). Neural networks-based backward scheme for fully nonlinear 
PDEs. SN Partial Differential Equations and Applications, 2(1), 1–24.

Potters, M., Bouchaud, J.-P., & Sestovic, D. (2001). Hedged Monte-Carlo: Low variance derivative pricing 
with objective probabilities. Physica A: Statistical Mechanics and its Applications, 289(3–4), 517–525.

Protter, P.E. (2005). Stochastic differential equations. In Stochastic integration and differential equations, 
pp. 249–361. Springer.

Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning 
framework for solving forward and inverse problems involving nonlinear partial differential equations. 
Journal of Computational Physics, 378, 686–707.

Remlinger, C., Mikael, J., & Elie, R. (2022). Robust operator learning to solve PDE. 

Srirgiano, J., & Spiliopoulos, K. (2018). Dgm: A deep learning algorithm for solving partial differential 
equations. Journal of Computational Physics, 375, 1339–1364.

Srivastava, R.K., Greff, K., & Schmidhuber, J. (2015). Training very deep networks. Advances in neural 
information processing systems, 28.

van der Meer, R., Oosterlee, C.W., & Borovykh, A. (2021). Optimally weighted loss functions for solving 
PDEs with neural networks. Journal of Computational and Applied Mathematics, p. 113887.

Vidales, M.S., Siska, D., & Szpruch, L. (2018). Unbiased deep solvers for parametric PDEs. 
arXiv:1810.05094.

Weinan, E., Han, J., & Jentzen, A. (2017). Deep learning-based numerical methods for high dimensional 
parabolic partial differential equations and backward stochastic differential equations. Communications in Mathematics and Statistics, 5(4), 349–380.
Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.