



HAL
open science

Neural networks-based backward scheme for fully nonlinear PDEs

Huyen Pham, Xavier Warin, Maximilien Germain

► **To cite this version:**

Huyen Pham, Xavier Warin, Maximilien Germain. Neural networks-based backward scheme for fully nonlinear PDEs. *SN Partial Differential Equations and Applications*, 2021, 2, 10.1007/s42985-020-00062-8 . hal-02196165v3

HAL Id: hal-02196165

<https://hal.science/hal-02196165v3>

Submitted on 10 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Neural networks-based backward scheme for fully nonlinear PDEs *

Huyên PHAM[†]

Xavier WARIN[‡]

Maximilien GERMAIN[§]

This version: December 7, 2020

Abstract

We propose a numerical method for solving high dimensional fully nonlinear partial differential equations (PDEs). Our algorithm estimates simultaneously by backward time induction the solution and its gradient by multi-layer neural networks, while the Hessian is approximated by automatic differentiation of the gradient at previous step. This methodology extends to the fully nonlinear case the approach recently proposed in [HPW20] for semi-linear PDEs. Numerical tests illustrate the performance and accuracy of our method on several examples in high dimension with non-linearity on the Hessian term including a linear quadratic control problem with control on the diffusion coefficient, Monge-Ampère equation and Hamilton-Jacobi-Bellman equation in portfolio optimization.

Key words: Neural networks, fully nonlinear PDEs in high dimension, backward scheme.

MSC Classification: 60H35, 65C20, 65M12.

1 Introduction

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

$$\begin{cases} \partial_t u + f(., ., u, D_x u, D_x^2 u) = 0, & \text{on } [0, T] \times \mathbb{R}^d, \\ u(T, .) = g, & \text{on } \mathbb{R}^d, \end{cases} \quad (1.1)$$

with a non-linearity in the solution, its gradient $D_x u$ and its hessian $D_x^2 u$ via the function $f(t, x, y, z, \gamma)$ defined on $[0, T] \times \mathbb{R}^d \times \mathbb{R} \times \mathbb{R}^d \times \mathbb{S}^d$ (where \mathbb{S}^d is the set of symmetric $d \times d$ matrices), and a terminal condition g .

The numerical resolution of this class of PDEs is far more difficult than the one of classical semi-linear PDEs where the nonlinear function f does not depend on γ . In fact, rather few methods are available to solve fully nonlinear equations even in moderate dimension.

- First based on the work of [Che+07], an effective scheme developed in [FTW11] using some regression techniques has been shown to be convergent under some ellipticity conditions later removed by [Tan13]. Due to the use of basis functions, this scheme does not permit to solve PDE in dimension greater than 5.
- A scheme based on nesting Monte Carlo has been recently proposed in [War18]. It seems to be effective in very high dimension for maturities T not too long and linearities not too important.
- A numerical algorithm to solve fully nonlinear equations has been proposed by [BEJ19] based on the second order backward stochastic differential equations (2BSDE) representation of [Che+07] and global deep neural networks minimizing a terminal objective function, but no test on real fully nonlinear case is given. This extends the idea introduced in the pioneering papers [EHJ17; HJE18], which were the first serious works for using machine learning methods to solve high dimensional PDEs.
- The Deep Galerkin method proposed in [SS18] based on some machine learning techniques and using some automatic differentiation of the solution seems to be effective on some cases. It has been tested in [AA+18] for example on the Merton problem.

*This work is supported by FiME, Laboratoire de Finance des Marchés de l’Energie, and the ”Finance and Sustainable Development” EDF - CACIB Chair.

[†]LPSM, Université de Paris, CREST-ENSAE & FiME pham at lpsm.paris

[‡]EDF R&D & FiME xavier.warin at edf.fr

[§]EDF R&D, LPSM, Université de Paris mgermain at lpsm.paris

In this article, we introduce a numerical method based on machine learning techniques and backward in time iterations, which extends the proposed schemes in [VSS18] for linear problems, and in the recent work [HPW20] for semi-linear PDEs. The approach in these works consists in estimating simultaneously the solution and its gradient by multi-layer neural networks by minimizing a sequence of loss functions defined in backward induction. A basic idea to extend this method to the fully nonlinear case would rely on the representation proposed in [Che+07]: at each time step t_n of an Euler scheme, the Hessian $D_x^2 u$ at t_n is approximated by a neural network minimizing some local L_2 criterion associated to a BSDE involving $D_x u$ at date t_{n+1} and $D_x^2 u$. Then, the pair $(u, D_x u)$ at date t_n is approximated/learned with a second minimization similarly as in the method described by [HPW20]. The first minimization can be implemented with different variations but numerical results show that the global scheme does not scale well with the dimension. Instability on the $D_x^2 u$ calculation rapidly propagates during the backward resolution. Besides, the methodology appears to be costly when using two optimizations at each time step. An alternative approach that we develop here, is to combine the ideas of [HPW20] and the splitting method in [Bec+19] in order to derive a new deep learning scheme that requires only one local optimization during the backward resolution for learning the pair $(u, D_x u)$ and approximating $D_x^2 u$ by automatic differentiation of the gradient computed at the previous step.

The outline of the paper is organized as follows. In Section 2, we briefly recall the mathematical description of the classical feedforward approximation, and then derive the proposed neural networks-based backward scheme. We test our method in Section 3 on various examples. First we illustrate our results with a PDE involving a non-linearity of type $uD_x^2 u$. Then, we consider a stochastic linear quadratic problem with controlled volatility where an analytic solution is available, and we test the performance and accuracy of our algorithm up to dimension 20. Next, we apply our algorithm to a Monge-Ampère equation, and finally, we provide numerical tests for the solution to fully nonlinear Hamilton-Jacobi-Bellman equation, with non-linearities of the form $|D_x u|^2/D_x^2 u$, arising in portfolio selection problem with stochastic volatilities.

2 The proposed deep backward scheme

Our aim is to numerically approximate the function $u : [0, T] \times \mathbb{R}^d \mapsto \mathbb{R}$, assumed to be the unique smooth solution to the fully nonlinear PDE (1.1) under suitable conditions. This will be achieved by means of neural networks approximations for u and its gradient $D_x u$, relying on a backward scheme and training simulated data of some forward diffusion process. Approximations of PDE in high dimension by neural networks have now become quite popular, and are supported theoretically by recent results in [Hut+18] and [DLM20] showing their efficiency to overcome the curse of dimensionality.

2.1 Feedforward neural network to approximate functions

We denote by d_0 the dimension of the input variables, and d_1 the dimension of the output variable. A (deep) neural network is characterized by a number of layers $L + 1 \in \mathbb{N} \setminus \{1, 2\}$ with m_ℓ , $\ell = 0, \dots, L$, the number of neurons (units or nodes) on each layer: the first layer is the input layer with $m_0 = d$, the last layer is the output layer with $m_L = d_1$, and the $L - 1$ layers between are called hidden layers, where we choose for simplicity the same dimension $m_\ell = m$, $\ell = 1, \dots, L - 1$.

A feedforward neural network is a function from \mathbb{R}^{d_0} to \mathbb{R}^{d_1} defined as the composition

$$x \in \mathbb{R}^d \mapsto A_L \circ \varrho \circ A_{L-1} \circ \dots \circ \varrho \circ A_1(x) \in \mathbb{R}. \quad (2.1)$$

Here A_ℓ , $\ell = 1, \dots, L$ are affine transformations: A_1 maps from \mathbb{R}^{d_0} to \mathbb{R}^m , A_2, \dots, A_{L-1} map from \mathbb{R}^m to \mathbb{R}^m , and A_L maps from \mathbb{R}^m to \mathbb{R}^{d_1} , represented by

$$A_\ell(x) = \mathcal{W}_\ell x + \beta_\ell,$$

for a matrix \mathcal{W}_ℓ called weight, and a vector β_ℓ called bias term, $\varrho : \mathbb{R} \rightarrow \mathbb{R}$ is a nonlinear function, called activation function, and applied component-wise on the outputs of A_ℓ , i.e., $\varrho(x_1, \dots, x_m) = (\varrho(x_1), \dots, \varrho(x_m))$. Standard examples of activation functions are the sigmoid, the ReLu, the Elu, tanh.

All these matrices \mathcal{W}_ℓ and vectors β_ℓ , $\ell = 1, \dots, L$, are the parameters of the neural network, and can be identified with an element $\theta \in \mathbb{R}^{N_m}$, where $N_m = \sum_{\ell=0}^{L-1} m_\ell(1+m_{\ell+1}) = d_0(1+m) + m(1+m)(L-2) + m(1+d_1)$ is the number of parameters. We denote by $\mathcal{N}_{d_0, d_1, L, m}$ the set of all functions generated by (2.1) for $\theta \in \mathbb{R}^{N_m}$.

2.2 Forward-backward representation

Let us introduce a forward diffusion process

$$X_t = X_0 + \int_0^t \mu(s, X_s) ds + \int_0^t \sigma(s, X_s) dW_s, \quad 0 \leq t \leq T, \quad (2.2)$$

where μ is a function defined on $[0, T] \times \mathbb{R}^d$ with values in \mathbb{R}^d , σ is a function defined on $[0, T] \times \mathbb{R}^d$ with values in \mathbb{M}^d the set of $d \times d$ matrices, and W a d -dimensional Brownian motion on some probability space $(\Omega, \mathcal{F}, \mathbb{P})$ equipped with a filtration $\mathbb{F} = (\mathcal{F}_t)_{0 \leq t \leq T}$ satisfying the usual conditions. The process X will be used for the simulation of training data in our deep learning algorithm, and we shall discuss later the choice of the drift and diffusion coefficients μ and σ , see Remark 2.3.

Let us next denote by (Y, Z, Γ) the triple of \mathbb{F} -adapted processes valued in $\mathbb{R} \times \mathbb{R}^d \times \mathbb{S}^d$, defined by

$$Y_t = u(t, X_t), \quad Z_t = D_x u(t, X_t), \quad \Gamma_t = D_x^2 u(t, X_t), \quad 0 \leq t \leq T. \quad (2.3)$$

By Itô's formula applied to $u(t, X_t)$, and since u is solution to (1.1), we see that (Y, Z, Γ) satisfies the backward equation:

$$Y_t = g(X_T) - \int_t^T [\mu(s, X_s) \cdot Z_s + \frac{1}{2} \text{tr}(\sigma \sigma^\top(s, X_s) \Gamma_s) - f(s, X_s, Y_s, Z_s, \Gamma_s)] ds - \int_t^T \sigma^\top(s, X_s) Z_s \cdot dW_s, \quad 0 \leq t \leq T. \quad (2.4)$$

Remark 2.1 This BSDE does not uniquely characterize a triple (Y, Z, Γ) contrarily to the semilinear case (without a non-linearity with respect to Γ) in which proper assumptions on the equation coefficients provide existence and uniqueness for a solution couple (Y, Z) . In the present case at least two options can be used to estimate the Γ component:

- Rely on the 2BSDE representation from [Che+07] which extends the probabilistic representation of [PP90] for semilinear equations to the fully nonlinear case. It is the approach used by [BEJ19] with a global large minimization problem, as in [HJE18].
- Compute the second order derivative by automatic differentiation. This is the point of view we adopt in this paper together with a local approach solving several small optimization problems. In this way, we provide an extension of [HPW20] to cover a broader range of nonlinear PDEs.

□

2.3 Algorithm

We now provide a numerical approximation of the forward backward system (2.2)-(2.4), and consequently of the solution u (as well as its gradient $D_x u$) to the PDE (1.1).

We start from a time grid $\pi = \{t_i, i = 0, \dots, N\}$ of $[0, T]$, with $t_0 = 0 < t_1 < \dots < t_N = T$, and time steps $\Delta t_i := t_{i+1} - t_i$, $i = 0, \dots, N-1$. The time discretization of the forward process X on π is then equal (typically when μ and σ are constants) or approximated by an Euler scheme:

$$X_{t_{i+1}} = X_{t_i} + \mu(t_i, X_{t_i}) \Delta t_i + \sigma(t_i, X_{t_i}) \Delta W_{t_i}, \quad i = 0, \dots, N-1,$$

where we set $\Delta W_{t_i} := W_{t_{i+1}} - W_{t_i}$ (by misuse of notation, we keep the same notation X for the continuous time diffusion process and its Euler scheme). The backward SDE (2.4) is approximated by the time discretized scheme

$$Y_{t_i} \simeq Y_{t_{i+1}} - [\mu(t_i, X_{t_i}) \cdot Z_{t_i} + \frac{1}{2} \text{tr}(\sigma \sigma^\top(t_i, X_{t_i}) \Gamma_{t_i}) - f(t_i, X_{t_i}, Y_{t_i}, Z_{t_i}, \Gamma_{t_i})] \Delta t_i - \sigma^\top(t_i, X_{t_i}) Z_{t_i} \cdot \Delta W_{t_i},$$

that is written in forward form as

$$Y_{t_{i+1}} \simeq F(t_i, X_{t_i}, Y_{t_i}, Z_{t_i}, \Gamma_{t_i}, \Delta t_i, \Delta W_{t_i}), \quad i = 0, \dots, N-1, \quad (2.5)$$

with

$$F(t, x, y, z, \gamma, h, \Delta) := y - \tilde{f}(t, x, y, z, \gamma) h + z^\top \sigma(t, x) \Delta, \quad (2.6)$$

$$\tilde{f}(t, x, y, z, \gamma) := f(t, x, y, z, \gamma) - \mu(t, x) \cdot z - \frac{1}{2} \text{tr}(\sigma \sigma^\top(t, x) \gamma).$$

The idea of the proposed scheme is the following. Similarly as in [HPW20], we approximate at each time t_i , $u(t_i, \cdot)$ and its gradient $D_x u(t_i, \cdot)$, by neural networks $x \in \mathbb{R}^d \mapsto (\mathcal{U}_i(x; \theta), \mathcal{Z}_i(x; \theta))$ with parameter θ that are learned optimally by backward induction: suppose that $\hat{\mathcal{U}}_{i+1} := \mathcal{U}_{i+1}(\cdot; \theta_{i+1}^*)$, $\hat{\mathcal{Z}}_{i+1} := \mathcal{Z}_{i+1}(\cdot; \theta_{i+1}^*)$ is an

approximation of $u(t_{i+1}, \cdot)$ and $D_x u(t_{i+1}, \cdot)$ at time t_{i+1} , then θ_i^* is computed from the minimization of the quadratic loss function:

$$\hat{L}_i(\theta) = \mathbb{E} \left| \hat{\mathcal{U}}_{i+1} - F(t_i, X_{t_i}, \mathcal{U}_i(X_{t_i}; \theta), \mathcal{Z}_i(X_{t_i}; \theta), D\hat{\mathcal{Z}}_{i+1}(\mathcal{T}(X_{t_{i+1}})), \Delta t_i, \Delta W_{t_i}) \right|^2$$

where \mathcal{T} is a truncation operator such that $\mathcal{T}(X)$ is bounded for example by a quantile of the diffusion process and $D\hat{\mathcal{Z}}_{i+1}$ stands for the automatic differentiation of $\hat{\mathcal{Z}}_{i+1}$. The idea behind the truncation is the following. During one step resolution, the estimation of the gradient is less accurate at the edge of the explored domain where samples are rarely generated. Differentiating the gradient gives a very oscillating Hessian at the edge of the domain. At the following time step resolution, these oscillations propagate to the gradient and the solution even if the domain where the oscillations occur is rarely attained. In order to avoid these oscillations, a truncation is achieved, permits to avoid that the oscillations of the neural network fit in zone where the simulations propagate scarcely to areas of importance. This truncation may be necessary to get convergence on some rather difficult cases. Of course this truncation is only valid if the real Hessian does not varies too much.

The intuition for the relevance of this scheme to the approximation of the PDE (1.1) is the following. From (2.3) and (2.5), the solution u to (1.1) should approximately satisfy

$$u(t_{i+1}, X_{t_{i+1}}) \simeq F(t_i, X_{t_i}, u(t_i, X_{t_i}), D_x u(t_i, X_{t_i}), D_x^2 u(t_i, X_{t_i}), \Delta t_i, \Delta W_{t_i}).$$

Suppose that at time t_{i+1} , $\hat{\mathcal{U}}_{i+1}$ is an estimation of $u(t_{i+1}, \cdot)$. Recalling the expression of F in (2.6), the quadratic loss function at time t_i is then approximately equal to

$$\begin{aligned} \hat{L}_i(\theta) \simeq & \mathbb{E} \left| u(t_i, X_{t_i}) - \mathcal{U}_i(X_{t_i}; \theta) + (D_x u(t_i, X_{t_i}) - \mathcal{Z}_i(X_{t_i}; \theta))^\top \sigma(t_i, X_{t_i}) \Delta W_{t_i} \right. \\ & \left. - \Delta t_i [\tilde{f}(t_i, X_{t_i}, u(t_i, X_{t_i}), D_x u(t_i, X_{t_i}), D_x^2 u(t_i, X_{t_i})) - \tilde{f}(t_i, X_{t_i}, \mathcal{U}_i(X_{t_i}; \theta), \mathcal{Z}_i(X_{t_i}; \theta), D\hat{\mathcal{Z}}_{i+1}(\mathcal{T}(X_{t_{i+1}})))] \right|^2. \end{aligned}$$

By assuming that \tilde{f} has small non-linearities in its arguments (y, z, γ), say Lipschitz, possibly with a suitable choice of μ, σ , the loss function is thus approximately equal to

$$\hat{L}_i(\theta) \simeq (1 + O(\Delta t_i)) \mathbb{E} |u(t_i, X_{t_i}) - \mathcal{U}_i(X_{t_i}; \theta)|^2 + O(\Delta t_i) \mathbb{E} |D_x u(t_i, X_{t_i}) - \mathcal{Z}_i(X_{t_i}; \theta)|^2 + O(|\Delta t_i|^2).$$

Therefore, by minimizing over θ this quadratic loss function, via stochastic gradient descent (SGD) based on simulations of $(X_{t_i}, X_{t_{i+1}}, \Delta W_{t_i})$ (called training data in the machine learning language), one expects the neural networks \mathcal{U}_i and \mathcal{Z}_i to learn/approximate better and better the functions $u(t_i, \cdot)$ and $D_x u(t_i, \cdot)$ in view of the universal approximation theorem for neural networks. The rigorous convergence of this algorithm is postponed to a future work.

To sum up, the global algorithm is given in Algo 1 in the case where g is Lipschitz and the derivative can be analytically calculated almost everywhere. If the derivative of g is not available, it can be calculated by automatic differentiation of the neural network approximation of g .

Algorithm 1 Algorithm for fully non-linear equations.

- 1: Use a single deep neural network $(\mathcal{U}_N(\cdot; \theta), \mathcal{Z}_N(\cdot; \theta)) \in \mathcal{N}_{d,1+d,L,m}$ and minimize (by SGD)

$$\begin{cases} \hat{L}_N(\theta) := \mathbb{E} |\mathcal{U}_N(X_{t_N}; \theta) - g(X_{t_N})|^2 + \frac{\Delta t_{N-1}}{d} \mathbb{E} |\mathcal{Z}_N(X_{t_N}; \theta) - Dg(X_{t_N})|^2 \\ \theta_N^* \in \arg \min_{\theta \in \mathbb{R}^{N_m}} \hat{L}_N(\theta). \end{cases}$$

- 2: $\hat{\mathcal{U}}_N = \mathcal{U}_N(\cdot; \theta_N^*)$, and set $\hat{\mathcal{Z}}_N = \mathcal{Z}_N(\cdot; \theta_N^*)$

- 3: **for** $i = N - 1, \dots, 0$ **do**

- 4: Use a single deep neural network $(\mathcal{U}_i(\cdot; \theta), \mathcal{Z}_i(\cdot; \theta)) \in \mathcal{N}_{d,1+d,L,m}$ for the approximation of $(u(t_i, \cdot), D_x u(t_i, \cdot))$, and compute (by SGD) the minimizer of the expected quadratic loss function

$$\begin{cases} \hat{L}_i(\theta) := \mathbb{E} \left| \hat{\mathcal{U}}_{i+1}(X_{t_{i+1}}) - F(t_i, X_{t_i}, \mathcal{U}_i(X_{t_i}; \theta), \mathcal{Z}_i(X_{t_i}; \theta), D\hat{\mathcal{Z}}_{i+1}(\mathcal{T}(X_{t_{i+1}})), \Delta t_i, \Delta W_{t_i}) \right|^2 \\ \theta_i^* \in \arg \min_{\theta \in \mathbb{R}^{N_m}} \hat{L}_i(\theta). \end{cases} \quad (2.7)$$

- 5: Update: $\hat{\mathcal{U}}_i = \mathcal{U}_i(\cdot; \theta_i^*)$, and set $\hat{\mathcal{Z}}_i = \mathcal{Z}_i(\cdot; \theta_i^*)$.
-

Remark 2.2 Several alternatives can be implemented for the computation of the second order derivative. A natural candidate would consist in choosing to approximate the solution u at time t_i by a neural network \mathcal{U}_i

and estimate Γ_i as the iterated automatic differentiation $D_x^2 \mathcal{U}_i$. However, it is shown in [HPW20] that choosing only a single neural network for u and using its automatic derivative to estimate the Z component degrades the error in comparison to the choice of two neural networks \mathcal{U}, \mathcal{Z} . A similar behavior has been observed during our tests for this second order case and the most efficient choice was to compute the derivative of the \mathcal{Z} network. This derivative can also be estimated at the current time step t_i instead of t_{i+1} . However this method leads to an additional cost for the neural networks training by complicating the computation of the automatic gradients performed by Tensorflow during the backpropagation. It also leads numerically to worse results on the control estimation, as empirically observed in Table 5 and described in the related paragraph "Comparison with an implicit version of the scheme". For this reason, we decided to apply a splitting method and evaluate the Hessian at time t_{i+1} . For this reason, we decided to apply a splitting method and evaluate the Hessian at time t_{i+1} . \square

Remark 2.3 The diffusion process X is used for the training simulations in the stochastic gradient descent method for finding the minimizer of the quadratic loss function in (2.7), where the expectation is replaced by empirical average for numerical implementation. The choice of the drift and diffusion parameters are explained in Section 3.1. \square

3 Numerical results

We first construct an example with different non-linearities in the Hessian term and the solution. We graphically show that the solution is very well calculated in dimension $d = 1$ and then move to higher dimensions. We then use an example derived from a stochastic optimization problem with an analytic solution and show that we are able to accurately calculate the solution. Next, we consider the numerical resolution of the Monge-Ampère equation, and finally, give some tests for a fully nonlinear Hamilton-Jacobi-Bellman equation arising from portfolio optimization with stochastic volatilities.

3.1 Choice of the algorithm hyperparameters

We describe in this paragraph how we choose the various hyperparameters of the algorithm and explain the learning strategy.

- **PARAMETERS OF THE TRAINING SIMULATIONS:** the choice of the drift coefficient is typically related to the underlying probabilistic problem associated to the PDE (for example a stochastic control problem), and should drive the training process to regions of interest, e.g., that are visited with large probability by the optimal state process in stochastic control. In practice, we can take a drift function $\mu(\cdot)$ equal to the drift associated to some a priori control. This choice of control could be an optimal control for a related problem for which we know the solution, or could be the control obtained by the first iteration of the algorithm. The choice of the diffusion coefficient σ is also important: large σ induces a better exploration of the state space, but as we will see in most of examples below, it gives a scheme slowly converging to the solution with respect to the time discretization and it generates a higher variance on the results. Moreover, for the applications in stochastic control, we might explore some region that are visited with very small probabilities by the optimal state process, hence representing few interest. On the other hand, small σ means a weak exploration, and we might lack information and precision on some region of the state space: the solution calculated at each time step is far more sensitive to very local errors induced by the neural network approximation and tends to generate a bias. Therefore a trade off has to be found between rather high variance with slow convergence in time and fast convergence in time with a potential bias. We also refer to [NR20] for a discussion on the role of the diffusion coefficient.

In practice and for the numerical examples in the next section, we test the scheme for different σ and by varying the number of time steps, and if it converges to the same solution, one can consider that we have obtained the correct solution. We also show the impact of the choice of the diffusion coefficient σ .

- **PARAMETERS OF TRUNCATION:** Given the training simulations X , we choose a truncation operator \mathcal{T}_p indexed by a parameter p close to 1, so that $\mathcal{T}_p(X_t)$ corresponds to a truncation of X_t at a given quantile ϕ_p . In the numerical tests, we shall vary p between 0.95 and 0.999.

- **PARAMETERS OF THE OPTIMIZATION ALGORITHM OVER NEURAL NETWORKS:** In the whole numerical part, we use a classical Feedforward network using layers with m neurons each and a tanh activation function, the output layer uses an identity activation function. At each time step the resolution of equation (2.7) is achieved using a mini-batch with 1000 training trajectories. The training and learning rate adaptation procedure is the following:

- Every 40 inner gradient descent iterations, the loss is checked on 10000 validation trajectories.
- This optimization sequence is repeated with 200 outer iterations for the first optimization step at date $t_N = T$ and only 100 outer iterations at the dates t_i with $i < N$.

- An average of the loss calculated on 10 successive outer iterations is performed. If the decrease of the average loss every 10 outer iterations is less than 5% then the learning rate is divided by 2.

The optimization is performed using the Adam gradient descent algorithm, see [KB14]. Notice that the adaptation of the learning rate is not common with the Adam method but in our case it appears to be crucial to have a steady diminution of the loss of the objective function. The procedure is also described in [CWNMW19] and the chosen parameters are similar to this article. At the initial optimization step at time $t_N = T$, the learning rate is taken equal to $1E - 2$ and at the following optimization steps, we start with a learning rate equal to $1E - 3$.

During time resolution, it is far more effective to initialize the solution of equations (2.7) with the solution $(\mathcal{U}, \mathcal{Z})$ at the next time step. Indeed the previously computed values at time step t_{i+1} are good approximations of the processes at time step t_i if the PDE solution and its gradient are continuous. All experiments are achieved using Tensorflow [Aba+15]. In the sequel, the PDE solutions on curves are calculated as the average of 10 runs. We provide the standard deviation associated to these results. We also show the influence of the number of neurons on the accuracy of the results.

3.2 A non-linearity in uD_x^2u

We consider a generator in the form

$$f(t, x, y, z, \gamma) = y\text{tr}(\gamma) + \frac{y}{2} + 2y^2 - 2y^4 e^{-(T-t)},$$

and $g(x) = \tanh\left(\frac{\sum_{i=1}^d x_i}{\sqrt{d}}\right)$, so that an analytical solution is available:

$$u(t, x) = \tanh\left(\frac{\sum_{i=1}^d x_i}{\sqrt{d}}\right) e^{-\frac{T-t}{2}}.$$

We fix the horizon $T = 1$, and choose to evaluate the solution at $t = 0$ and $x = 0.5\frac{\mathbf{1}_d}{\sqrt{d}}$ (here $\mathbf{1}_d$ denotes the vector in \mathbb{R}^d with all components equal to 1), for which $u(t, x) = 0.761902$ while its derivative is equal to 1.2966. This initial value x is chosen such that independently of the dimension the solution is varying around this point and not in a region where the tanh function is close to -1 or 1 .

The coefficients of the forward process used to solve the equation are (here \mathbf{I}_d is the identity $d \times d$ -matrix)

$$\sigma = \frac{\hat{\sigma}}{\sqrt{d}} \mathbf{I}_d, \quad \mu = 0,$$

and here the truncation operator is chosen equal to

$$\mathcal{T}_p(X_t^{0,x}) = \min \{ \max[x - \sigma\sqrt{t}\phi_p, X_t^{0,x}], x + \sigma\sqrt{t}\phi_p \},$$

where $\phi_p = \mathcal{N}^{-1}(p)$, with \mathcal{N} is the CDF of a unit centered Gaussian random variable.

In the numerical results, we take $p = 0.999$ and $m = 20$ neurons. We first begin in dimension $d = 1$, and show in Figure 1 how u , $D_x u$ and $D_x^2 u$ are well approximated by the resolution method.

On Figure 2, we check the convergence, for different values of $\hat{\sigma}$ of both the solution u and its derivative at point x and date 0. Standard deviation of the function value is very low and the standard deviation of the derivative still being low.

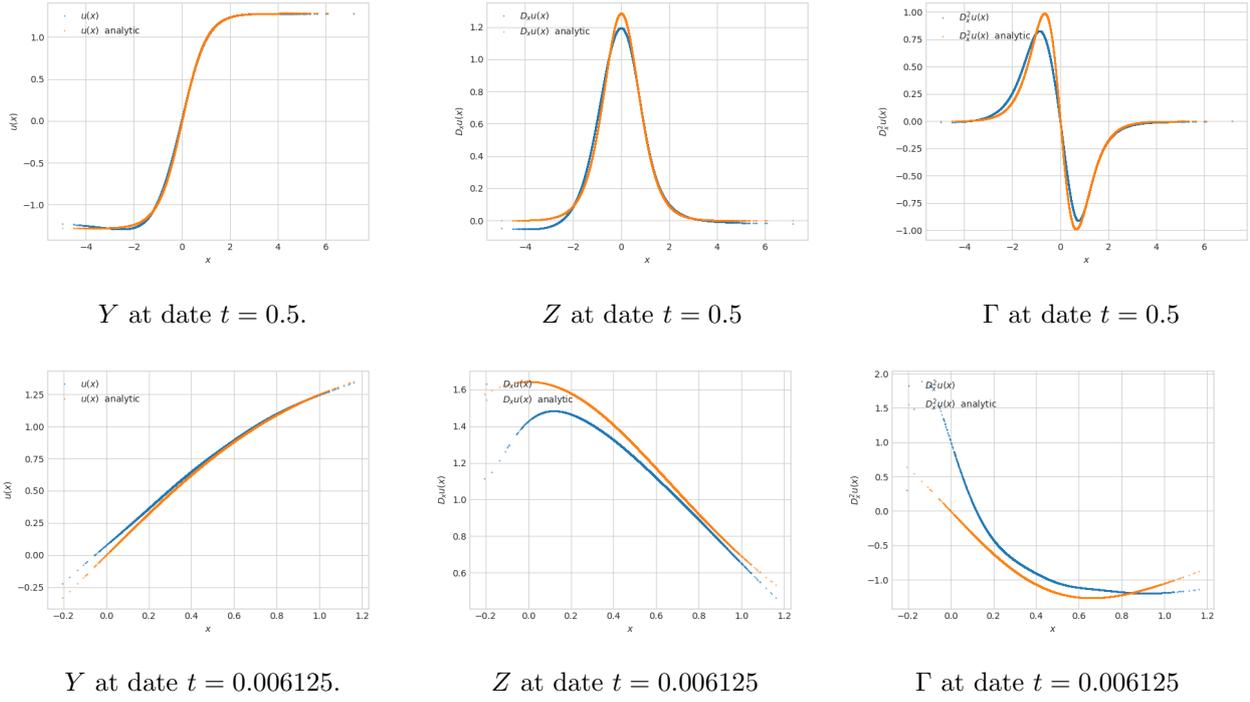


Figure 1: A single valuation run for test case one 1D using 160 time steps, $\hat{\sigma} = 2.$, $p = 0.999$, 20 neurons, 2 layers.

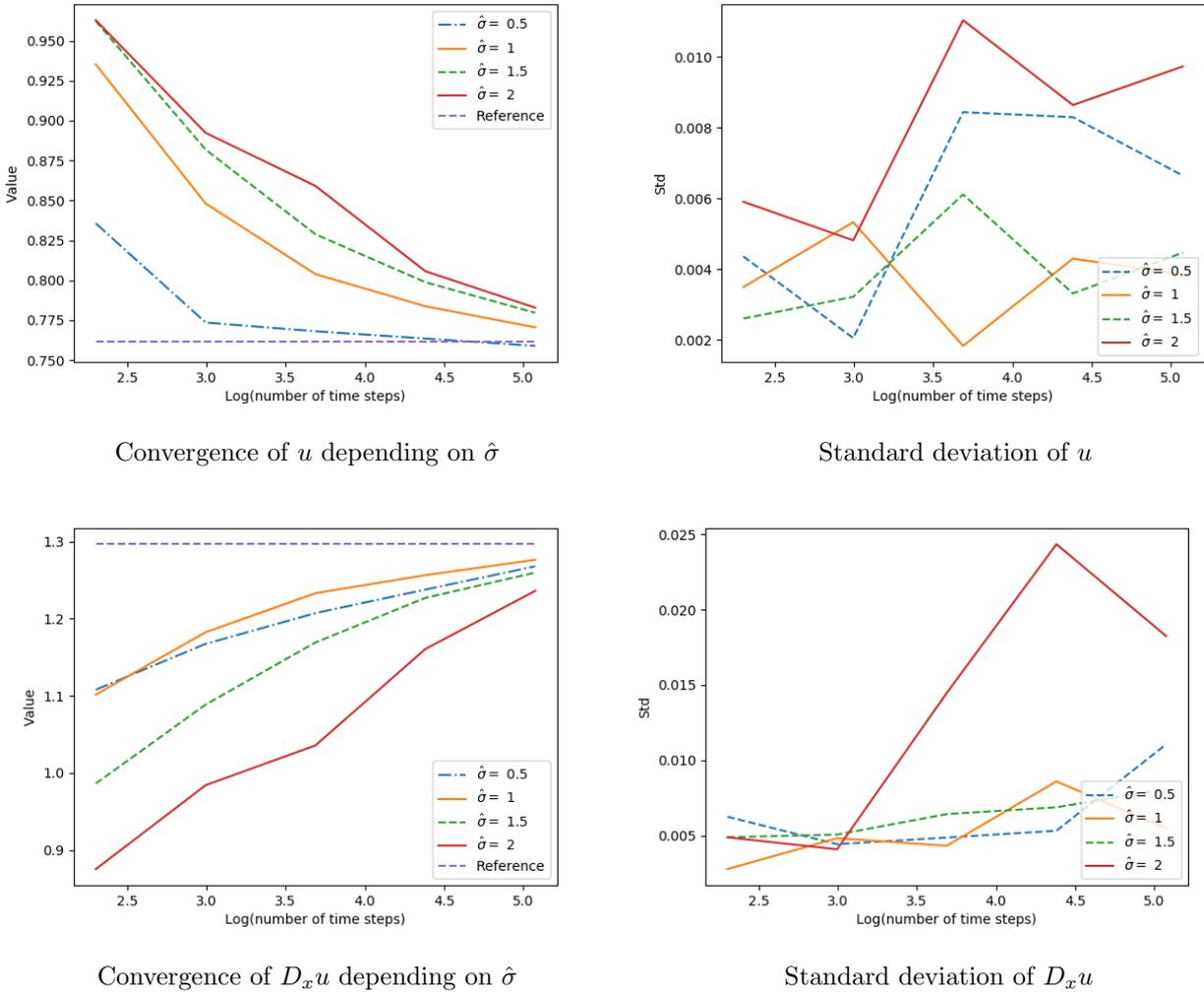
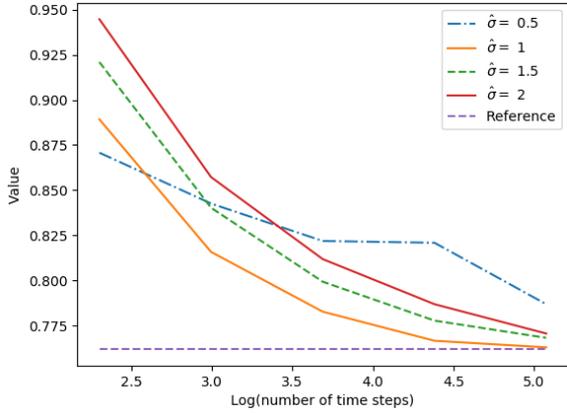
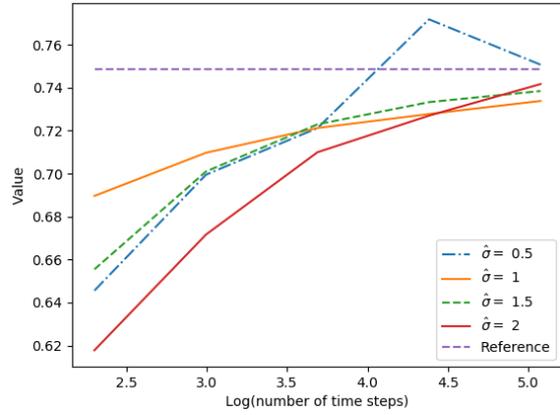


Figure 2: Convergence in 1D of the case one, number of neurons par layer equal to 20, 2 layers, $p = 0.999$.

As the dimension increases, we have to increase the value of $\hat{\sigma}$ of the forward process. In dimension 3, the value $\hat{\sigma} = 0.5$ gives high standard deviation in the result obtained as shown on Figure 3, while in dimension 10, see Figure 4, we see that the value $\hat{\sigma} = 1$ is too low to give good results. We also clearly notice that in 10D, a smaller time step should be used but in our test cases we decided to consider a maximum number of time steps equal to 160.

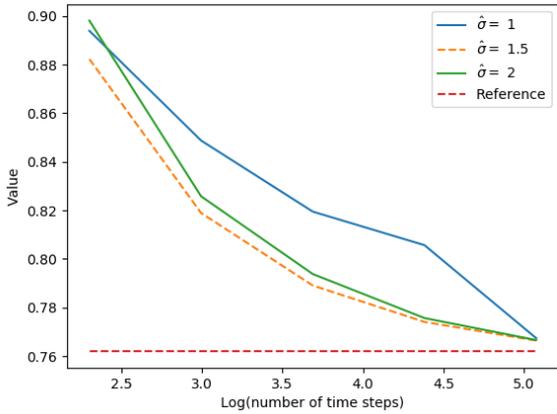


Convergence of u depending on $\hat{\sigma}$

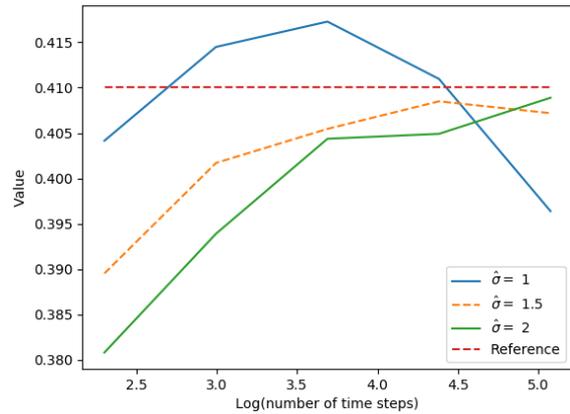


Convergence of $D_x u$ (first component) depending on $\hat{\sigma}$

Figure 3: Convergence in 3D of the case one, number of neurons par layer equal to 20, 2 layers, $p = 0.999$.



Convergence of u depending on $\hat{\sigma}$



Convergence of $D_x u$ depending on $\hat{\sigma}$ (first component)

Figure 4: Convergence in 10D of the case one, number of neurons par layer equal to 20, 2 layers, $p = 0.999$.

On this simple test case, the dimension is not a problem and very good results are obtained in dimension 20 or above with only 20 neurons and 2 layers.

3.3 A linear quadratic stochastic test case.

In this example, we consider a controlled process $\mathcal{X} = \mathcal{X}^\alpha$ with dynamics in \mathbb{R}^d according to

$$d\mathcal{X}_t = (A\mathcal{X}_t + B\alpha_t)dt + D\alpha_t dW_t, \quad 0 \leq t \leq T, \quad \mathcal{X}_0 = x,$$

where W is a real Brownian motion, the control process α is valued in \mathbb{R} , and the constant coefficients $A \in \mathbb{M}^d$, $B \in \mathbb{R}^d$, $D \in \mathbb{R}^d$. The quadratic cost functional to be minimized is

$$J(\alpha) = \mathbb{E} \left[\int_0^T (\mathcal{X}_t^\top Q \mathcal{X}_t + \alpha_t^2 N) dt + \mathcal{X}_T^\top P \mathcal{X}_T \right],$$

where P, Q are non negative $d \times d$ symmetric matrices and $N \in \mathbb{R}$ is strictly positive.

The Bellman equation associated to this stochastic control problem is:

$$\begin{aligned} \frac{\partial u}{\partial t} + \inf_{a \in \mathbb{R}} [(Ax + Ba) \cdot D_x u + \frac{a^2}{2} \text{tr}(DD^\top D_x^2 u) + x^\top Q x + Na^2] &= 0, \quad (t, x) \in [0, T) \times \mathbb{R}^d, \\ u(T, x) &= x^\top P x, \quad x \in \mathbb{R}^d, \end{aligned}$$

which can be rewritten as a fully nonlinear equation in the form (1.1) with

$$f(t, x, y, z, \gamma) = x^\top Q x + Ax \cdot z - \frac{1}{2} \frac{|B^\top z|^2}{\text{tr}(DD^\top \gamma) + 2N}.$$

An explicit solution to this PDE is given by

$$u(t, x) = x^\top K(t)x,$$

where $K(t)$ is non negative $d \times d$ symmetric matrix function solution to the Riccati equation:

$$\dot{K} + A^\top K + KA + Q - \frac{KBB^\top K}{N + D^\top KD} = 0, \quad K(T) = P.$$

We take $T = 1$. The coefficients of the forward process used to solve the equation are

$$\sigma = \frac{\hat{\sigma}}{\sqrt{d}} \mathbf{I}_d, \quad \mu(t, x) = Ax.$$

In our numerical example we take the following parameters for the optimization problem:

$$A = \mathbf{I}_d, \quad B = D = \mathbf{1}_d, \quad Q = P = \frac{1}{d} \mathbf{I}_d, \quad N = d$$

and we want to estimate the solution at $x = \mathbf{1}_d$.

In this example, the truncation operator (indexed by p between 0 and 1 and close to 1) is as follows:

$$\mathcal{T}_p(X_t^x) = \min \left\{ \max \left[x e^{\hat{A}t} - \sigma \sqrt{\frac{e^{2\hat{A}t} - \hat{\mathbf{1}}}{2\hat{A}}} \phi_p, X_t^x \right], x e^{\hat{A}t} + \sigma \sqrt{\frac{e^{2\hat{A}t} - \hat{\mathbf{1}}}{2\hat{A}}} \phi_p \right\},$$

where $\phi_p = \mathcal{N}^{-1}(p)$, \hat{A} is a vector so that $\hat{A}_i = A_{ii}$, $i = 1, \dots, d$, $\hat{\mathbf{1}}$ is a unit vector, and the square root is taken componentwise.

On Figure 5 we give the solution of the PDE with $d = 1$ using $\hat{\sigma} = 1.5$ obtained for two dates: at $t = 0.5$ and at t close to zero. We observe that we have a very good estimation of the function value and a correct one of the Γ value at date $t = 0.5$. The precision remains good for Γ close to $t = 0$ and very good for u and $D_x u$.

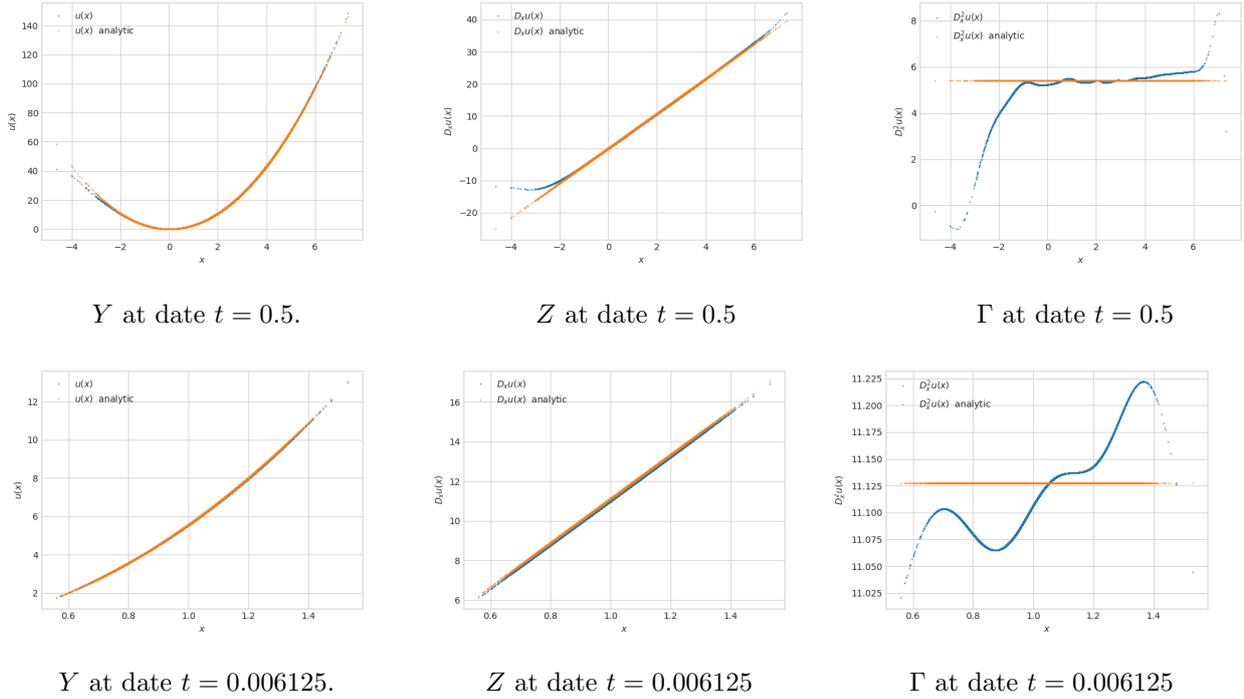


Figure 5: Test case linear quadratic 1D using 160 time steps, $\hat{\sigma} = 1.5$, $p = 0.999$, 100 neurons.

On Figure 6, we give the results obtained in dimension $d = 1$ by varying $\hat{\sigma}$. For a value of $\hat{\sigma} = 2$, the standard deviation of the result becomes far higher than with $\hat{\sigma} = 0.5$ or 1.

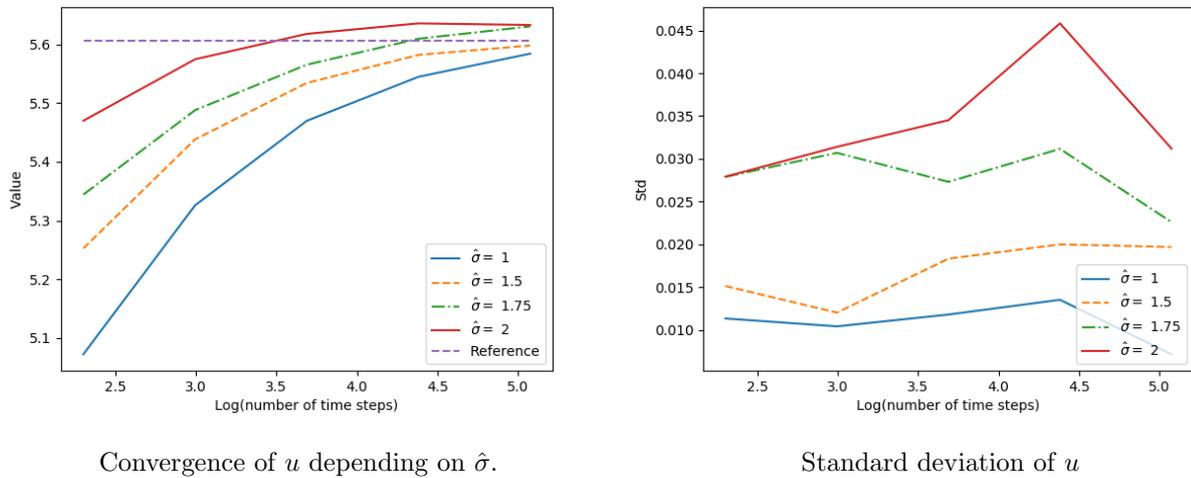
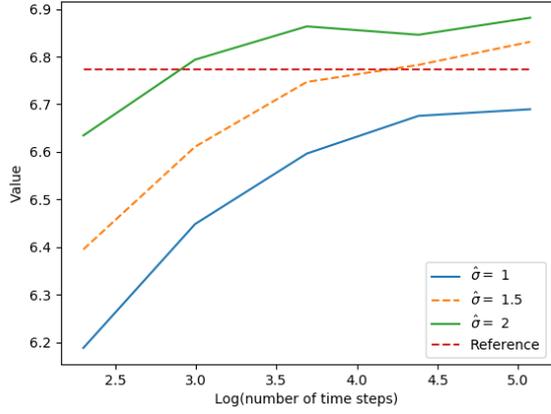
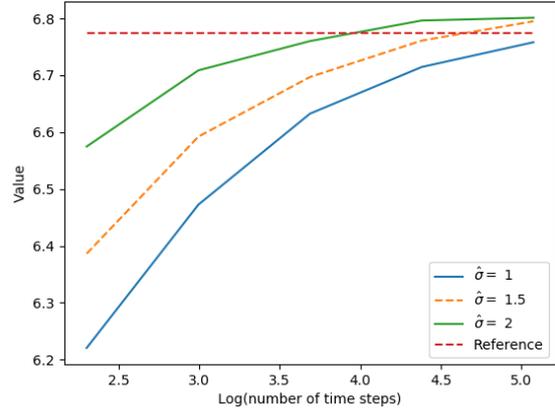


Figure 6: Convergence in 1D of the linear quadratic case, number of neurons par layer equal to 50, 2 layers, $p = 0.999$.

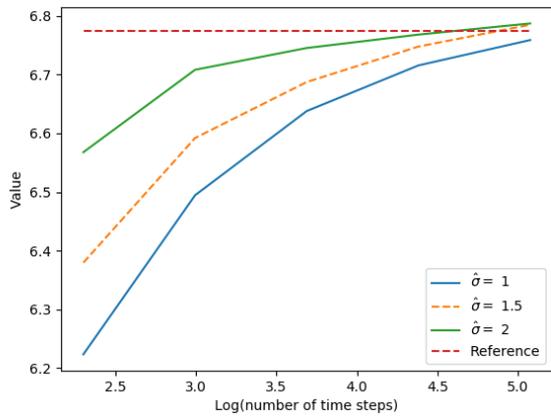
On Figure 7, for $d = 3$, we take a quite low truncation factor $p = 0.95$ and observe that the number of neurons to take has to be rather high. We have also checked that taking a number of hidden layers equal to 3 does not improve the results.



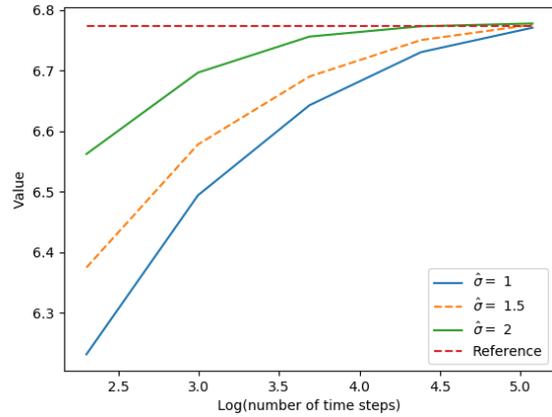
10 neurons



20 neurons



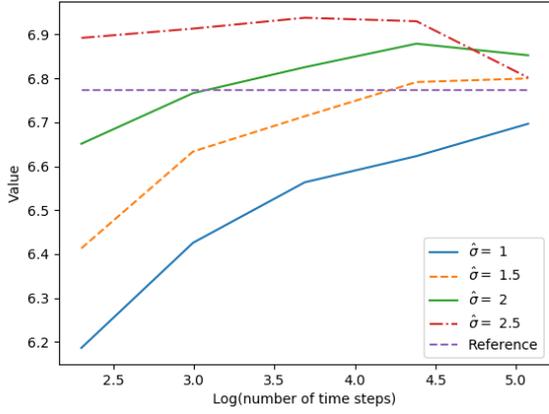
30 neurons



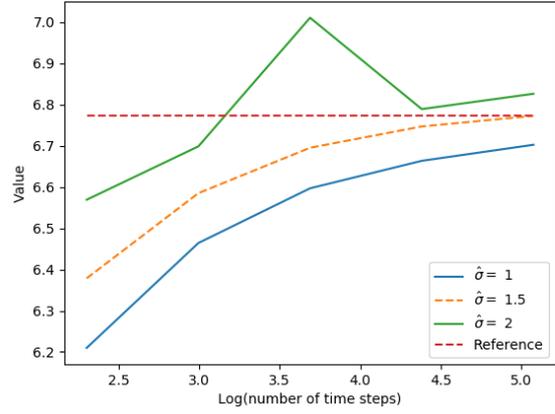
50 neurons

Figure 7: Convergence in 3D of the linear quadratic case, 2 layers, testing the influence of the number of neurons, truncation $p = 0.95$.

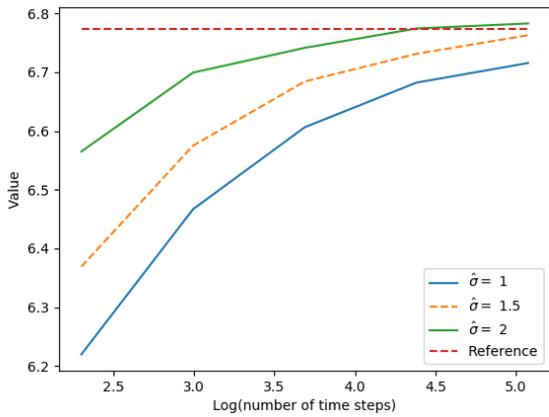
On Figure 8, for $d = 3$, we give the same graphs for a higher truncation factor. As we take a higher truncation factor, the results are improved by taking a higher number of neurons (100 in the figure below).



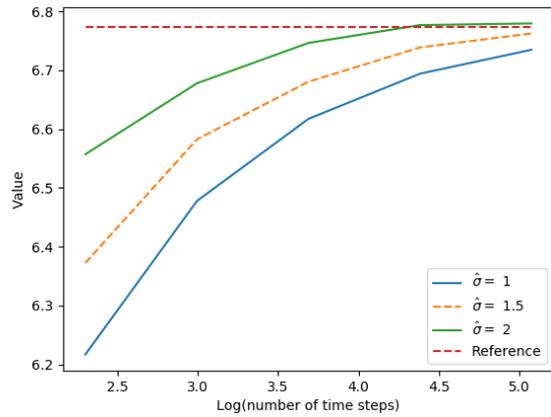
10 neurons



20 neurons



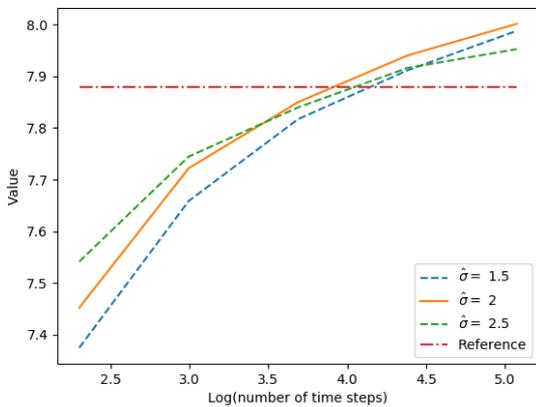
50 neurons



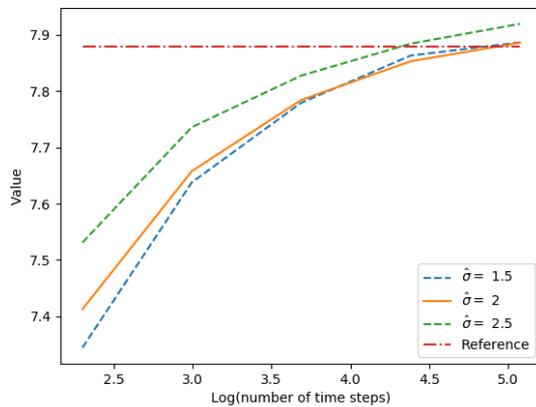
100 neurons

Figure 8: Convergence in 3D of the linear quadratic case, 2 layers, testing the influence of the number of neurons, truncation $p = 0.99$.

On Figure 9, we observe in dimension 7 the influence of the number of neurons on the result for a high truncation factor $p = 0.999$. We clearly have a bias for a number of neurons equal to 50. This bias disappears when the number of neurons increases to 100.



Convergence with 50 neurons



Convergence with 100 neurons

Figure 9: Convergence in 7D of the linear quadratic case, 2 layers, $p = 0.999$.

On Figure 10, for $d = 7$, we check that influence of the truncation factor appears to be slow for higher dimensions.

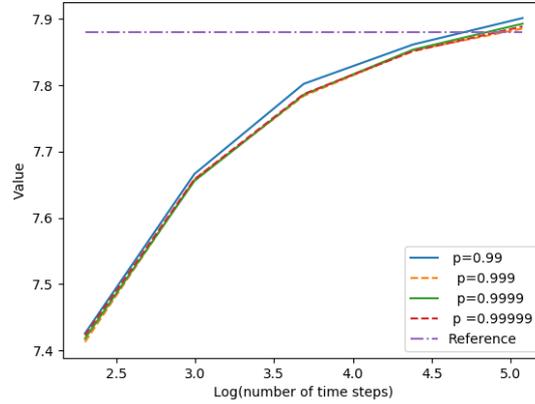


Figure 10: Function value convergence in 7D of the linear quadratic case with 2 layers, 100 neurons, testing p , using $\hat{\sigma} = 2$

Finally, we give results in dimension 10, 15 and 20 for $p = 0.999$ on Figures 11, 12. We observe that the number a neurons with 2 hidden layers has to increase with the dimension but also that the increase is rather slow in contrast with the case of one hidden layer as theoretically shown in [Pin99]. For $\hat{\sigma} = 5$ we had to take 300 neurons to get very accurate results.

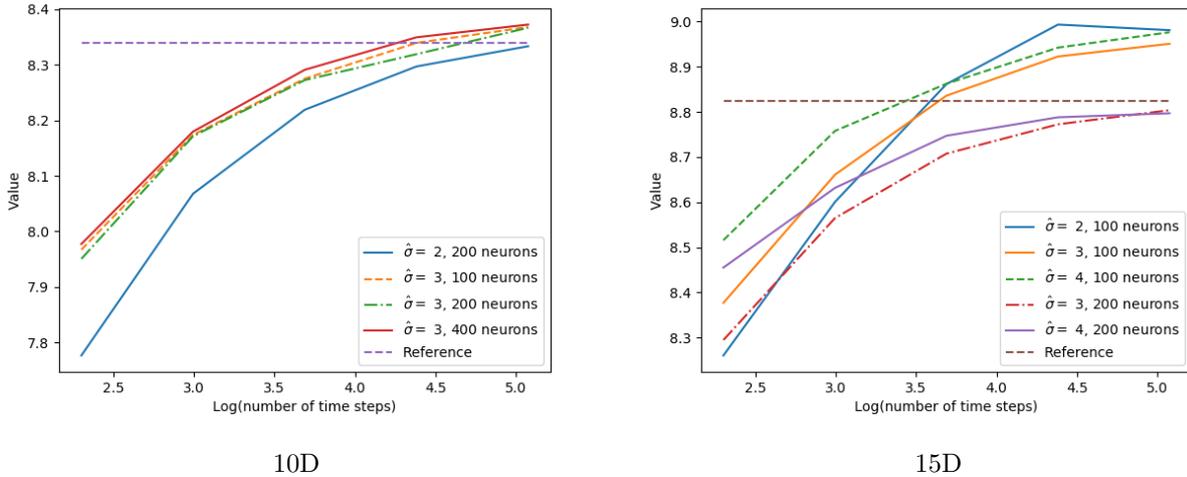


Figure 11: Function value convergence in 10D and 15D of the linear quadratic case with 2 layers, $p = 0.999$.

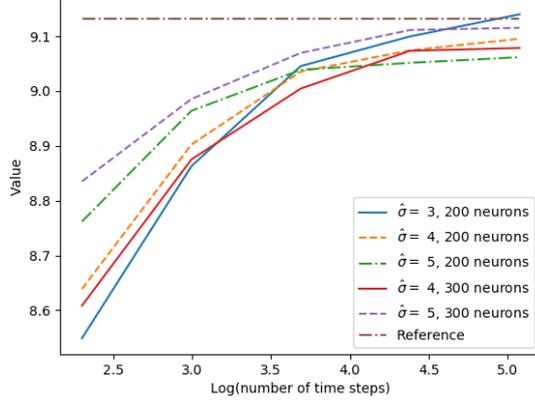


Figure 12: Function value convergence in 20D of the linear quadratic case with 2 layers, $p = 0.999$.

3.4 Monge-Ampère equation

Let us consider the parabolic Monge-Ampère equation

$$\begin{cases} \partial_t u + \det(D_x^2 u) = h(x), & (t, x) \in [0, T] \times \mathbb{R}^d, \\ u(T, x) = g(x), \end{cases} \quad (3.1)$$

where $\det(D_x^2 u)$ is the determinant of the Hessian matrix $D_x^2 u$. It is in the form (1.1) with

$$f(t, x, \gamma) = \det(\gamma) - h(x).$$

We test our algorithm by choosing a C^2 function g , then compute $G = \det(D_x^2 g)$, and set $h := G - 1$. Then, by construction, the function

$$u(t, x) = g(x) + T - t,$$

is solution to the Monge-Ampère equation (3.1). We choose $g(x) = \cos(\sum_{i=1}^d x_i / \sqrt{d})$, and we shall train with the forward process $X = x_0 + W$, where W is a d -dimensional Brownian motion. On this example, we use neural networks with 3 hidden layers, $d + 10$ neurons per layer, and we do not need to apply any truncation to the forward process X . Actually, we observe that adding a truncation worsens the results. For choosing the truncation level, we first test the method with no truncation before decreasing the quantile parameter p . In the Monge-Ampère case the best results are obtained without any truncation. It may be caused by the oscillation of the Hessian.

The following table gives the results in dimension $d = 5, 15$, and for $T = 1$.

Dimension d	Averaged value	Standard deviation	Relative error (%)	Theoretical solution
5	0.37901	0.00312	0.97	0.382727
15	0.25276	0.00235	1.17	0.255754

Table 1: Estimate of $u(0, x_0 = 1_5)$ on the Monge Ampere problem (3.1) with $N = 120$. Average and standard deviation observed over 10 independent runs are reported.

3.5 Portfolio selection

We consider a portfolio selection problem formulated as follows. There are n risky assets of uncorrelated price process $P = (P^1, \dots, P^n)$ with dynamics

$$dP_t^i = P_t^i \sigma(V_t^i) [\lambda_i(V_t^i) dt + dW_t^i], \quad i = 1, \dots, n,$$

where $W = (W^1, \dots, W^n)$ is a n -dimensional Brownian motion, $b = (b^1, \dots, b^n)$ is the rate of return of the assets, $\lambda = (\lambda^1, \dots, \lambda^n)$ is the risk premium of the assets, σ is a positive function (e.g. $\sigma(v) = e^v$ corresponding to the Scott model), and $V = (V^1, \dots, V^n)$ is the volatility factor modeled by an Ornstein-Uhlenbeck (O.U.) process

$$dV_t^i = \kappa_i [\theta_i - V_t^i] dt + \nu_i dB_t^i, \quad i = 1, \dots, n, \quad (3.2)$$

with $\kappa_i, \theta_i, \nu_i > 0$, and $B = (B^1, \dots, B^n)$ a n -dimensional Brownian motion, s.t. $d \langle W^i, B^j \rangle = \delta_{ij} \rho_{ij} dt$, with $\rho_i := \rho_{ii} \in (-1, 1)$. An agent can invest at any time an amount $\alpha_t = (\alpha_t^1, \dots, \alpha_t^n)$ in the stocks, which generates a wealth process $\mathcal{X} = \mathcal{X}^\alpha$ governed by

$$d\mathcal{X}_t = \sum_{i=1}^n \alpha_t^i \sigma(V_t^i) [\lambda_i(V_t^i) dt + dW_t^i].$$

The objective of the agent is to maximize her expected utility from terminal wealth:

$$\mathbb{E}[U(\mathcal{X}_T^\alpha)] \leftarrow \text{maximize over } \alpha$$

It is well-known that the solution to this problem can be characterized by the dynamic programming method (see e.g. [Pha09]), which leads to the Hamilton-Jacobi-Bellman for the value function on $[0, T) \times \mathbb{R} \times \mathbb{R}^n$:

$$\begin{cases} \partial_t u + \sum_{i=1}^n [\kappa_i(\theta_i - v_i) \partial_{v_i} u + \frac{1}{2} \nu_i^2 \partial_{v_i}^2 u] = \frac{1}{2} R(v) \frac{(\partial_x u)^2}{\partial_{xx}^2 u} + \sum_{i=1}^n [\rho_i \lambda_i(v_i) \nu_i \frac{\partial_x u \partial_{xv_i}^2 u}{\partial_{xx}^2 u} + \frac{1}{2} \rho_i^2 \nu_i^2 \frac{(\partial_{xv_i}^2 u)^2}{\partial_{xx}^2 u}] \\ u(T, x, v) = U(x), \quad x \in \mathbb{R}, v \in \mathbb{R}^n, \end{cases}$$

with a Sharpe ratio $R(v) := |\lambda(v)|^2$, for $v = (v_1, \dots, v_n) \in (0, \infty)^n$. The optimal portfolio strategy is then given in feedback form by $\alpha_t^* = \hat{\alpha}(t, \mathcal{X}_t^*, V_t)$, where $\hat{\alpha} = (\hat{\alpha}_1, \dots, \hat{\alpha}_n)$ is given by

$$\hat{\alpha}_i(t, x, v) = -\frac{1}{\sigma(v_i)} \left(\lambda_i(v_i) \frac{\partial_x u}{\partial_{xx}^2 u} + \rho_i \nu_i \frac{\partial_{xv_i}^2 u}{\partial_{xx}^2 u} \right), \quad (t, x, v = (v_1, \dots, v_n)) \in [0, T) \times \mathbb{R} \times \mathbb{R}^n,$$

for $i = 1, \dots, n$. This Bellman equation is in the form (1.1) with

$$f(t, x, y, z, \gamma) = \sum_{i=1}^n [\kappa_i(\theta_i - v_i) z_i + \frac{1}{2} \nu_i^2 \gamma_{ii}] - \frac{1}{2} R(v) \frac{z_0^2}{\gamma_{00}} - \sum_{i=1}^n [\rho_i \lambda_i(v_i) \nu_i \frac{z_0 \gamma_{0i}}{\gamma_{00}} + \frac{1}{2} \rho_i^2 \nu_i^2 \frac{(\gamma_{0i})^2}{\gamma_{00}}],$$

for $x = (x, v) \in \mathbb{R}^{n+1}$, $z = (z_0, \dots, z_n) \in \mathbb{R}^{n+1}$, $\gamma = (\gamma_{ij})_{0 \leq i, j \leq n} \in \mathbb{S}^{n+1}$, and displays a high non-linearity in the Hessian argument γ .

The truncation operator indexed by a parameter p is chosen equal to

$$\mathcal{T}_p(X_t^{0,x}) = \min \{ \max[x + \mu t - \sigma \sqrt{t} \phi_p, X_t^{0,x}], x + \mu t + \sigma \sqrt{t} \phi_p \},$$

where $\phi_p = \mathcal{N}^{-1}(p)$, \mathcal{N} is the CDF of a unit centered Gaussian random variable. We use neural networks with 2 hidden layers and $d + 10$ neurons per layer. We shall test this example when the utility function U is of exponential form: $U(x) = -\exp(-\eta x)$, with $\eta > 0$, and under different cases for which closed-form solutions are available:

- (1) *Merton problem.* This corresponds to a degenerate case where the factor V , hence the volatility σ and the risk premium λ are constant, so that the generator of Bellman equation reduces to

$$f(t, x, y, z, \gamma) = -\frac{1}{2} |\lambda|^2 \frac{z^2}{\gamma}, \quad (t, x, y, z) \in [0, T] \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}, \quad (3.3)$$

with explicit solution given by:

$$u(t, x) = e^{-(T-t) \frac{|\lambda|^2}{2}} U(x), \quad \hat{\alpha}_i = \frac{\lambda_i}{\eta \sigma}.$$

We train with the forward process

$$X_{k+1} = X_k + \lambda \Delta t_k + \Delta W_k, \quad k = 0, \dots, N, X_0 = x_0.$$

- (2) *One risky asset:* $n = 1$. A quasi-explicit solution is provided in [Zar01]:

$$u(t, x, v) = U(x) w(t, v), \quad \text{with } w(t, v) = \left\| \exp \left(-\frac{1}{2} \int_t^T R(\hat{V}_s^{t,v}) ds \right) \right\|_{L^{1-\rho^2}}$$

where $\hat{V}_s^{t,v}$ is the solution to the modified O.U. model

$$d\hat{V}_s = [\kappa(\theta - \hat{V}_s) - \rho \nu \lambda(\hat{V}_s)] ds + \nu dB_s, \quad s \geq t, \hat{V}_t = v.$$

We test our algorithm with $\lambda(v) = \lambda v$, $\lambda > 0$, for which we have an explicit solution:

$$w(t, v) = \exp\left(-\phi(t)\frac{v^2}{2} - \psi(t)v - \chi(t)\right), \quad (t, v) \in [0, T] \times \mathbb{R},$$

where (ϕ, ψ, χ) are solutions of the Riccati system of ODEs:

$$\begin{aligned} \dot{\phi} - 2\bar{\kappa}\phi - \nu^2(1 - \rho^2)\phi^2 + \lambda^2 &= 0, & \phi(T) &= 0, \\ \dot{\psi} - (\bar{\kappa} + \nu^2(1 - \rho^2)\phi)\psi + \kappa\theta\phi &= 0, & \psi(T) &= 0, \\ \dot{\chi} + \kappa\theta\psi - \frac{\nu^2}{2}(-\phi + (1 - \rho^2)\psi^2) &= 0, & \chi(T) &= 0, \end{aligned}$$

with $\bar{\kappa} = \kappa + \rho\nu\lambda$, and explicitly given by (see e.g. Appendix in [SZ99])

$$\begin{aligned} \phi(t) &= \lambda^2 \frac{\sinh(\hat{\kappa}(T-t))}{\hat{\kappa} \cosh(\hat{\kappa}(T-t)) + \bar{\kappa} \sinh(\hat{\kappa}(T-t))} \\ \psi(t) &= \lambda^2 \frac{\kappa\theta}{\hat{\kappa}} \frac{\cosh(\hat{\kappa}(T-t)) - 1}{\hat{\kappa} \cosh(\hat{\kappa}(T-t)) + \bar{\kappa} \sinh(\hat{\kappa}(T-t))} \\ \chi(t) &= \frac{1}{2(1-\rho^2)} \ln \left[\cosh(\hat{\kappa}(T-t)) + \frac{\bar{\kappa}}{\hat{\kappa}} \sinh(\hat{\kappa}(T-t)) \right] - \frac{1}{2(1-\rho^2)} \bar{\kappa}(T-t) \\ &\quad - \lambda^2 \frac{(\kappa\theta)^2}{\hat{\kappa}^2} \left[\frac{\sinh(\hat{\kappa}(T-t))}{\hat{\kappa} \cosh(\hat{\kappa}(T-t)) + \bar{\kappa} \sinh(\hat{\kappa}(T-t))} - (T-t) \right] \\ &\quad - \lambda^2 \frac{(\kappa\theta)^2 \bar{\kappa}}{\hat{\kappa}^3} \frac{\cosh(\hat{\kappa}(T-t)) - 1}{\hat{\kappa} \cosh(\hat{\kappa}(T-t)) + \bar{\kappa} \sinh(\hat{\kappa}(T-t))}, \end{aligned}$$

with $\hat{\kappa} = \sqrt{\kappa^2 + 2\rho\nu\lambda\kappa + \gamma^2\lambda^2}$. We train with the forward process

$$\begin{aligned} \mathcal{X}_{k+1} &= \mathcal{X}_k + \lambda\theta\Delta t_k + \Delta W_k, \quad k = 0, \dots, N-1, \quad \mathcal{X}_0 = \mathbf{x}_0, \\ V_{k+1} &= V_k + \nu\Delta B_k, \quad k = 0, \dots, N-1, \quad V_0 = \theta. \end{aligned}$$

(3) *No leverage effect, i.e., $\rho_i = 0$, $i = 1, \dots, n$.* In this case, there is a quasi-explicit solution given by

$$u(t, \mathbf{x}, v) = U(\mathbf{x})w(t, v), \quad \text{with } w(t, v) = \mathbb{E}\left[\exp\left(-\frac{1}{2}\int_t^T R(V_s^{t,v})ds\right)\right], \quad (t, v) \in [0, T] \times \mathbb{R}^n, \quad (3.4)$$

where $V^{t,v}$ is the solution to (3.2), starting from v at time t . We test our algorithm with $\lambda_i(v) = \lambda_i v_i$, $\lambda_i > 0$, $i = 1, \dots, n$, $v = (v_1, \dots, v_n)$, for which we have an explicit solution given by

$$\begin{aligned} w(t, v) &= \exp\left(-\sum_{i=1}^n \left[\phi_i(t)\frac{v_i^2}{2} + \psi_i(t)v_i + \chi_i(t)\right]\right), \quad (t, v) \in [0, T] \times \mathbb{R}^n, \\ \phi_i(t) &= \lambda_i^2 \frac{\sinh(\hat{\kappa}_i(T-t))}{\kappa_i \sinh(\hat{\kappa}_i(T-t)) + \hat{\kappa}_i \cosh(\hat{\kappa}_i(T-t))} \\ \psi_i(t) &= \lambda_i^2 \frac{\kappa_i \theta_i}{\hat{\kappa}_i} \frac{\cosh(\hat{\kappa}_i(T-t)) - 1}{\hat{\kappa}_i \cosh(\hat{\kappa}_i(T-t)) + \kappa_i \sinh(\hat{\kappa}_i(T-t))} \\ \chi_i(t) &= \frac{1}{2} \ln \left[\cosh(\hat{\kappa}_i(T-t)) + \frac{\kappa_i}{\hat{\kappa}_i} \sinh(\hat{\kappa}_i(T-t)) \right] - \frac{1}{2} \kappa_i(T-t) \\ &\quad - \lambda_i^2 \frac{(\kappa_i \theta_i)^2}{\hat{\kappa}_i^2} \left[\frac{\sinh(\hat{\kappa}_i(T-t))}{\hat{\kappa}_i \cosh(\hat{\kappa}_i(T-t)) + \kappa_i \sinh(\hat{\kappa}_i(T-t))} - (T-t) \right] \\ &\quad - \lambda_i^2 \frac{(\kappa_i \theta_i)^2 \kappa_i}{\hat{\kappa}_i^3} \frac{\cosh(\hat{\kappa}_i(T-t)) - 1}{\hat{\kappa}_i \cosh(\hat{\kappa}_i(T-t)) + \kappa_i \sinh(\hat{\kappa}_i(T-t))}, \end{aligned}$$

with $\hat{\kappa}_i = \sqrt{\kappa_i^2 + \nu_i^2 \lambda_i^2}$. We train with the forward process

$$\begin{aligned} \mathcal{X}_{k+1} &= \mathcal{X}_k + \Delta W_k, \quad k = 0, \dots, N-1, \quad \mathcal{X}_0 = \mathbf{x}_0, \\ V_{k+1}^i &= V_k^i + \nu_i \Delta B_k^i, \quad k = 0, \dots, N-1, \quad V_0^i = \theta_i, \end{aligned}$$

with $\langle W, B^i \rangle_t = 0$.

Merton Problem. We take $\eta = 0.5$, $\lambda = 0.6$, $T = 1$, $N = 120$, and $\sigma(v) = e^v$. We plot the neural networks approximation of $u, D_x u, D_x^2 u, \alpha$ (in blue) together with their analytic values (in orange). For comparison with Figures 6 and 7, we report the error on the gradient and the initial control. In practice, after empirical tests, we choose $p = 0.98$ for the truncation.

	Averaged value	Standard deviation	Theoretical value	Relative error (%)
$u(0, x_0 = 1)$	-0.50561	0.00029	-0.50662	0.20
$D_x u(0, x_0 = 1)$	0.25081	0.00088	0.25331	0.99
$\alpha(0, x_0 = 1)$	0.83552	0.02371	0.80438	3.87

Table 2: Estimate of the solution, its derivative and the optimal control at the initial time $t = 0$ in the Merton problem (3.3). Average and standard deviation observed over 10 independent runs are reported.

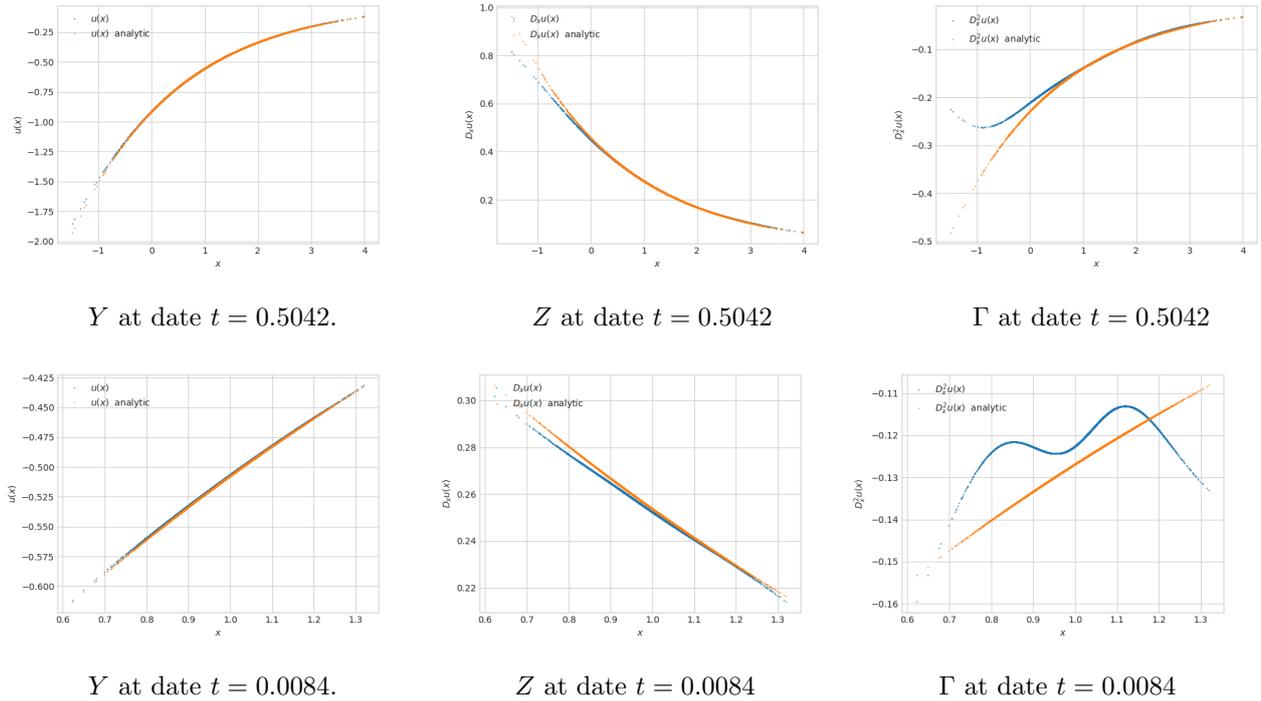


Figure 13: Estimates of the solution and its derivatives on the Merton problem (3.3) using 120 time steps.

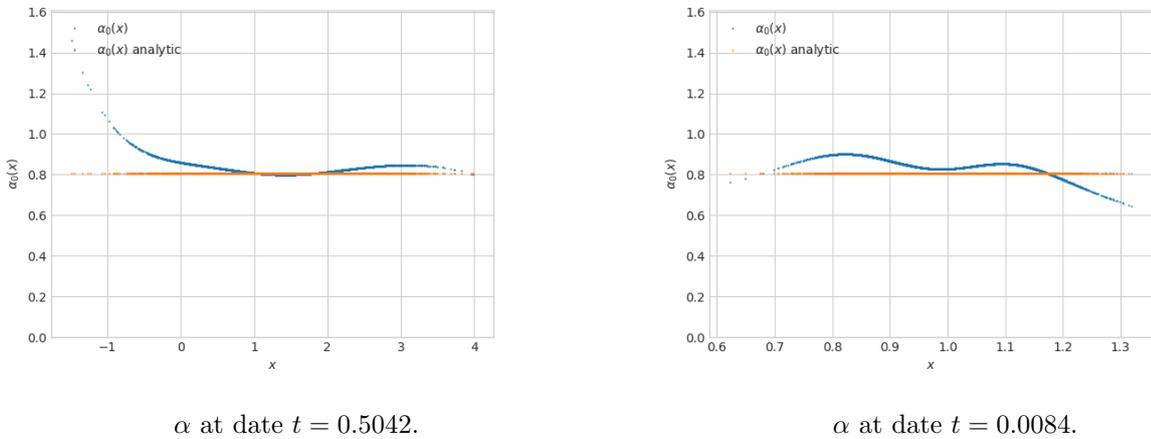


Figure 14: Estimates of the optimal control α on the Merton problem (3.3).

One asset ($n = 1$) in Scott volatility model. We take $\eta = 0.5$, $\lambda = 1.5$, $\theta = 0.4$, $\nu = 0.4$, $\kappa = 1$, $\rho = -0.7$. For all tests we choose $T = 1$, $N = 120$, and $\sigma(v) = e^v$. In practice, after empirical tests, we choose

$p = 0.98$ for the truncation.

Averaged value	Standard deviation	Relative error (%)
-0.53431	0.00070	0.34

Table 3: Estimate of $u(0, x_0 = 1, \theta)$ on the One Asset problem with stochastic volatility ($d = 2$). Average and standard deviation observed over 10 independent runs are reported. The exact solution is -0.53609477 .

No Leverage in Scott model. In the case with one asset ($n = 1$), we take $\eta = 0.5$, $\lambda = 1.5$, $\theta = 0.4$, $\nu = 0.2$, $\kappa = 1$. For all tests we choose $T = 1$, $N = 120$, and $\sigma(v) = e^v$. In practice, after empirical tests, we choose $p = 0.95$ for the truncation.

Dimension d	Averaged value	Standard deviation	Relative error (%)	Theoretical solution
2	-0.49980	0.00073	0.35	-0.501566
5	-0.43768	0.00137	0.92	-0.441765
8	-0.38720	0.00363	1.96	-0.394938
10	-0.27920	0.05734	1.49	-0.275092

Table 4: Estimate of $u(0, x_0 = 1, \theta)$ on the No Leverage problem (3.4). Average and standard deviation observed over 10 independent runs are reported.

In the case with four assets ($n = 4$, $d = 5$), we take $\eta = 0.5$, $\lambda = (1.5 \ 1.1 \ 2. \ 0.8)$, $\theta = (0.1 \ 0.2 \ 0.3 \ 0.4)$, $\nu = (0.2 \ 0.15 \ 0.25 \ 0.31)$, $\kappa = (1. \ 0.8 \ 1.1 \ 1.3)$.

In the case with seven assets ($n = 7$, $d = 8$) we take $\eta = 0.5$, $\lambda = (1.5 \ 1.1 \ 2. \ 0.8 \ 0.5 \ 1.7 \ 0.9)$, $\theta = (0.1 \ 0.2 \ 0.3 \ 0.4 \ 0.25 \ 0.15 \ 0.18)$, $\nu = (0.2 \ 0.15 \ 0.25 \ 0.31 \ 0.4 \ 0.35 \ 0.22)$, $\kappa = (1. \ 0.8 \ 1.1 \ 1.3 \ 0.95 \ 0.99 \ 1.02)$.

In the case with nine assets ($n = 9$, $d = 10$), we take $\eta = 0.5$, $\lambda = (1.5 \ 1.1 \ 2. \ 0.8 \ 0.5 \ 1.7 \ 0.9 \ 1. \ 0.9)$, $\theta = (0.1 \ 0.2 \ 0.3 \ 0.4 \ 0.25 \ 0.15 \ 0.18 \ 0.08 \ 0.91)$, $\nu = (0.2 \ 0.15 \ 0.25 \ 0.31 \ 0.4 \ 0.35 \ 0.22 \ 0.4 \ 0.15)$, $\kappa = (1. \ 0.8 \ 1.1 \ 1.3 \ 0.95 \ 0.99 \ 1.02 \ 1.06 \ 1.6)$.

Hamilton-Jacobi-Bellman equation from portfolio optimization is a typical example of full-nonlinearity in the second order derivative, and the above results show that our algorithm performs quite well up to dimension $d = 8$, but gives a high variance in dimension $d = 10$.

Comparison with an implicit version of the scheme. As explained in Remark 2.2, an alternative option for the estimation of the Hessian is to approximate it by the automatic differentiation of the current neural network for the Z component. It corresponds to the replacement of $D\hat{\mathcal{Z}}_{i+1}(\mathcal{T}(X_{t_{i+1}}))$ by $D\mathcal{Z}_i(\mathcal{T}(X_{t_i}); \theta)$ in (2.7). An additional change has to be made to the method for it to work. At the last optimization step (for time step $t_0 = 0$), we notice empirically that the variable Γ_0 is not able to properly learn the initial Hessian value at all. Therefore for this last step we use variables Y_0, Z_0 and an explicit estimation of the second order derivative given by $D\hat{\mathcal{Z}}_1(\mathcal{T}(X_{t_1}))$. We see in Table 5 that the results for the Merton problem are very similar to the ones from Table 2 for the splitting scheme but with a worse estimation of the Hessian and optimal control (the error is multiplied by around 1.5). When we tested this implicit scheme on the Monge Ampere problem we also faced computational problems during the optimization step of Tensorflow. The numerical computation of the gradient of the objective function for the backpropagation step, more precisely for the determinant part, often gives rise to matrix invertibility errors which stops the algorithm execution. For these two reasons, we focused our study on the explicit scheme.

	Average	Std	True value	Relative error (%)
$u(0, x_0 = 1)$	-0.50572	0.00034	-0.50662	0.18
$D_x u(0, x_0 = 1)$	0.25091	0.00067	0.25331	0.95
$\alpha(0, x_0 = 1)$	0.85254	0.01956	0.80438	5.99

Table 5: Estimate of the solution, its derivative and the optimal control at the initial time $t = 0$ in the Merton problem (3.3) with implicit estimation of the Hessian. Average and standard deviation (Std) observed over 10 independent runs are reported

Comparison with the 2BSDE scheme of [BEJ19]. We conclude this paper with a comparison of our algorithm with the global scheme of [BEJ19], called Deep 2BDSE. The tests below concern the Merton

problem (3.3) but similar behavior happens on the other examples with stochastic volatilities. This scheme was implemented in the original paper only for small number of time steps (e.g. $N = 30$). Thus we tested this algorithm on two discretizations, respectively with $N = 20$ and $N = 120$ time steps, as shown in Figure 15, for $T = 1$ where we plotted the learning curve of the Deep BSDE method. These curves correspond to the values taken by the loss function during the gradient descent iterations. For this algorithm the loss function to minimize in the training of neural networks is defined as the mean L^2 error between the generated Y_N value and the true terminal condition $g(X_N)$. We observe that for this choice of maturity $T = 1$ the loss function oscillates during the training process and does not vanish. As a consequence the Deep 2BSDE does not converge in this case. Even when decreasing the learning rate, we noticed that we cannot obtain the convergence of the scheme.

However, the Deep 2BSDE method does converge for small maturities T , as illustrated in Table 6 with $T = 0.1$ and different values for the number of time steps N . Nevertheless, even if the value function is well approximated, the estimation of the gradient and control did not converge (the corresponding variance is very large), in comparison with our scheme whereas the gradient is very well approximated and the control is quite precise. We also have a much smaller variance in the results. Table 7 shows the results obtained by our method with $T = 0.1$ in order to compare it with the performance of [BEJ19]. It illustrates the limitations of the global approach and justifies our introduction of a local method.

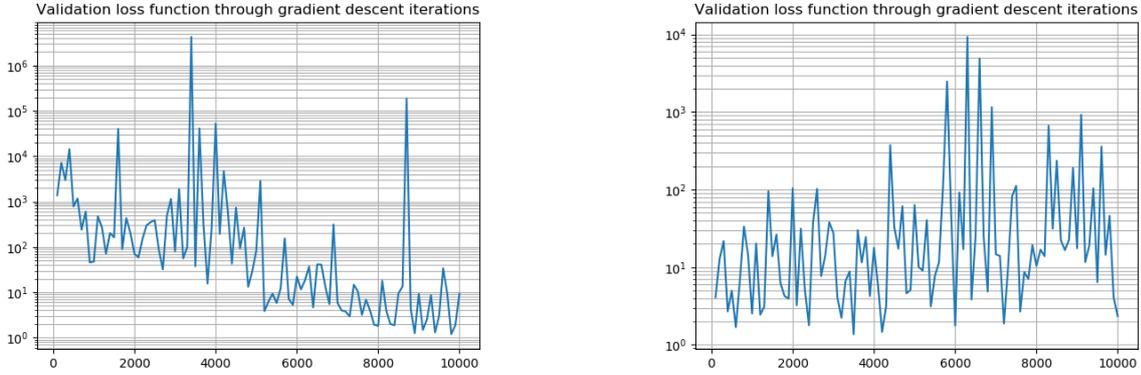


Figure 15: Learning curve in logarithmic scale for the scheme [BEJ19] on the Merton problem (3.3) with $N = 20$ times steps on the left and $N = 120$ time steps on the right. The maturity is $T = 1$. 10000 gradient descent iterations were conducted.

	N	Averaged value	Standard deviation	Theoretical value	Relative error (%)
$u(0, x_0 = 1)$	5	-0.60667	0.01588	-0.59571	1.84
$u(0, x_0 = 1)$	10	-0.59841	0.02892	-0.59571	0.45
$u(0, x_0 = 1)$	20	-0.59316	0.04251	-0.59571	0.43
$D_x u(0, x_0 = 1)$	5	0.09668	0.25630	0.29786	67.54
$D_x u(0, x_0 = 1)$	10	0.03810	0.44570	0.29786	93.36
$D_x u(0, x_0 = 1)$	20	0.07557	0.55030	0.29786	74.63
$\alpha(0, x_0 = 1)$	5	-0.15243	0.61096	0.80438	118.95
$\alpha(0, x_0 = 1)$	10	0.59971	1.97906	0.80438	25.44
$\alpha(0, x_0 = 1)$	20	0.28385	0.43775	0.80438	64.71

Table 6: Estimate of the solution, its derivative and the optimal control at the initial time $t = 0$ in the Merton problem (3.3) with maturity $T = 0.1$ for the [BEJ19] scheme. Average and standard deviation observed over 10 independent runs are reported.

	N	Averaged value	Standard deviation	Theoretical value	Relative error (%)
$u(0, x_0 = 1)$	5	-0.59564	0.01136	-0.59571	0.01
$u(0, x_0 = 1)$	10	-0.59550	0.00037	-0.59571	0.04
$u(0, x_0 = 1)$	20	-0.59544	0.00054	-0.59571	0.04
$D_x u(0, x_0 = 1)$	5	0.29848	0.00044	0.29786	0.21
$D_x u(0, x_0 = 1)$	10	0.29842	0.00084	0.29786	0.19
$D_x u(0, x_0 = 1)$	20	0.29785	0.00054	0.29786	0.001
$\alpha(0, x_0 = 1)$	5	0.82322	0.01014	0.80438	2.34
$\alpha(0, x_0 = 1)$	10	0.85284	0.07565	0.80438	6.02
$\alpha(0, x_0 = 1)$	20	0.84201	0.09892	0.80438	4.68

Table 7: Estimate of the solution, its derivative and the optimal control at the initial time $t = 0$ in the Merton problem (3.3) with maturity $T = 0.1$ for our scheme. Average and standard deviation observed over 10 independent runs are reported.

References

- [AA+18] A. Al-Aradi et al. “Solving Nonlinear and High-Dimensional Partial Differential Equations via Deep Learning”. In: *arXiv:1811.08782* (2018).
- [Aba+15] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [Bec+19] C. Beck et al. “Deep splitting method for parabolic PDEs”. In: *arXiv:1907.03452* (2019).
- [BEJ19] C. Beck, W. E, and A. Jentzen. “Machine Learning Approximation Algorithms for High-Dimensional Fully Nonlinear Partial Differential Equations and Second-order Backward Stochastic Differential Equations”. In: *J. Nonlinear Sci.* 29.4 (2019), pp. 1563–1619.
- [Che+07] P. Cheridito et al. “Second-order backward stochastic differential equations and fully nonlinear parabolic PDEs”. In: *Communications on Pure and Applied Mathematics* 60.7 (2007), pp. 1081–1110.
- [CWNMW19] Quentin Chan-Wai-Nam, Joseph Mikael, and Xavier Warin. “Machine learning for semi linear PDEs”. In: *Journal of Scientific Computing* 79.3 (2019), pp. 1667–1712.
- [DLM20] J. Darbon, G. Langlois, and T. Meng. “Overcoming the curse of dimensionality for some Hamilton–Jacobi partial differential equations via neural network architectures”. In: *Res Math Sci* 7 (2020).
- [EHJ17] W. E, J. Han, and A. Jentzen. “Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations”. In: *Communications in Mathematics and Statistics* 5.4 (2017), pp. 349–380.
- [FTW11] A. Fahim, N. Touzi, and X. Warin. “A probabilistic numerical method for fully nonlinear parabolic PDEs”. In: *The Annals of Applied Probability* (2011), pp. 1322–1364.
- [HJE18] J. Han, A. Jentzen, and W. E. “Solving high-dimensional partial differential equations using deep learning”. In: *Proceedings of the National Academy of Sciences of the United States of America* 115.34 (2018), pp. 8505–8510.
- [HPW20] Côme Huré, Huyên Pham, and Xavier Warin. “Deep backward schemes for high-dimensional nonlinear PDEs”. In: *Mathematics of Computation* 89.324 (2020), pp. 1547–1579.
- [Hut+18] M. Hutzenthaler et al. “Overcoming the curse of dimensionality in the numerical approximation of semilinear parabolic partial differential equations”. In: *arXiv:1807.01212* (2018).
- [KB14] D. P. Kingma and J. Ba. *Adam: A Method for Stochastic Optimization*. Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: <http://arxiv.org/abs/1412.6980>.
- [NR20] N. Nüsken and L. Richter. “Solving high-dimensional Hamilton-Jacobi-Bellman PDEs using neural networks: perspectives from the theory of controlled diffusions and measures on path space”. In: *arXiv:2005.05409* (2020).
- [Pha09] H. Pham. *Continuous-time Stochastic Control and Optimization with Financial Applications*. Vol. 61. SMAP. Springer, 2009.
- [Pin99] A. Pinkus. “Approximation theory of the MLP model in neural networks”. In: *Acta numerica* 8 (1999), pp. 143–195.

- [PP90] E. Pardoux and S. Peng. “Adapted solution of a backward stochastic differential equation”. In: *Systems & Control Letters* 14.1 (1990), pp. 55–61.
- [SS18] J. Sirignano and K. Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364.
- [SZ99] R. Schöbel and J. Zhu. “Stochastic volatility with an Ornstein-Uhlenbeck process and extension”. In: *Review of Finance* 3.1 (1999), pp. 23–46.
- [Tan13] X. Tan. “A splitting method for fully nonlinear degenerate parabolic PDEs”. In: *Electronic Journal of Probability* 18 (2013).
- [VSS18] M. Sabate Vidales, D. Siska, and L. Szpruch. “Unbiased deep solvers for parametric PDEs”. In: *arXiv:1810.05094v2* (2018).
- [War18] X. Warin. “Monte Carlo for high-dimensional degenerated Semi Linear and Full Non Linear PDEs”. In: *arXiv:1805.05078* (2018).
- [Zar01] T. Zariphopoulou. “A solution approach to valuation with unhedgeable risks”. In: *Finance and Stochastics* 5 (2001), pp. 61–82.