Technical Report

Using tensor techniques to solve stochastic equations
CentraleSupélec : Pole Projet INFO-Q

Pole Projet INFO-Q
Bouadjadja I.
Davion C.
Kahla A.

Abstract

Many Backward Forward Stochastic Equations (BSDE) doesn't have a know analytical solution, however, they arise in many fields, from biology to finance. Hence, knowing their solution is critical. The most common ways to obtain said solutions is to approximate them with numerical methods that stem from discretization schemes for partial differentiable equations, or more recently, with neural networks. The trade-off of these methods is that precision comes at the cost of both time and space. This is a direct result of intuitive simple matrix multiplication. In this report, we will be going over said methods and provide some insights about finding which steps take the most time or space, before suggesting possible optimization with a quantum inspired tool that are tensor networks. We will be going over mainly two different approaches: the first approach consists of finding an approximate solution with neural networks and then optimizing the numbers of trainable parameters by integrating tensor networks into the architecture. The second approach consists of using a very specific tensor network to model the solution of a BSDE [6] and apply it to a discretization scheme. In this second method, no learning is involved. Practical applications will be made using equations stemming from the world of finance, like European option pricing equations.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.1.1 Definition and notations . . . 3
1.1.2 Tensor operation . . . 3
1.2 BSDE . . . 4
1.2.1 Some stochastic process theory reminders . . . 4
1.2.2 Definition . . . 5
1.2.3 A variant for our application . . . 5

2 Neural Network **6**
2.1 Motivation . . . 6
2.2 The architecture and the Black-Scholes-Barenblatt equation . . . 6
2.3 The matrix product operator . . . 8
2.4 Factorizing the hidden layers . . . 8

2.5 Convergence and possible optimisation 10
2.6 Influence of the dimension of the problem 12

3 Tensor Train **12**
3.1 Motivation 12
3.2 The ALS algorithms 12
3.3 Results 14

4 Conclusion **14**

1 Introduction

1.1 Motivation

As stated before, the main time and space heavy part in both method is the usage of matrices. Dense neural network can easily reach millions of parameters and the computation power scales directly with said number. Hence making economy on storage space by sharply diminishing the amount of needed parameters welcome. The main take away is that when using matrices, some type of redundant information is stored within it. For example, if a matrix can be diagonalized, the action of said matrix, seen as an transformation of the n^{th} dimensional space, can be stored with n parameters rather than n^2. Hence, one can find a way to design a more compact storage of the same amount of information.

However, a matrix that is diagonalizable is a rare occurrence (even though, technically, they are dense within the matrix space) and requires one to find eigenvalues, which is not always cost efficient in terms of computational power. We nevertheless have something weaker, but that applies to all matrices and requires less computational power called the **Singular Value Decompostion** (SVD). The following tools, tensors, provides us with exactly what we need, as they translate well the SVD.

1.1.1 Definition and notations

For a more complete course on tensors and tensor networks see J. Biamonte's the lecture notes [2].
A **tensor** is the generalization of vectors and matrices. Formally, a tensor is a finite table of numbers, usually from the real or complex fields, arranged over an arbitrary number of axis.

$$(a_{i_1 i_2 \dots i_k}), i_p \in [1, N_p]$$

Said number k is called the **rank** of a tensor. Hence, a tensor of rank zero is a scalar, a tensor of rank one is a vector and a tensor of rank two is a matrix. See below 3.

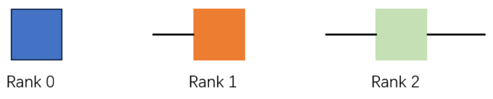


Figure 1: Example of tensors

This graphical representation is known as the Penrose notation [2].

1.1.2 Tensor operation

The main operator that we will use in this article is the tensor contraction. Let $\mathbf{A}_{ijk}$ and $\mathbf{B}_{iql}$ be two tensors. The contraction of $\mathbf{A}$ and $\mathbf{B}$ over axis i is the tensor $\mathbf{C}$ where each element is the following

$$\mathbf{C}_{jkql} = \sum_i A_{ijk} B_{iql}$$

$\mathbf{C}$ is represented by connecting the corresponding axis, see Fig. 2.

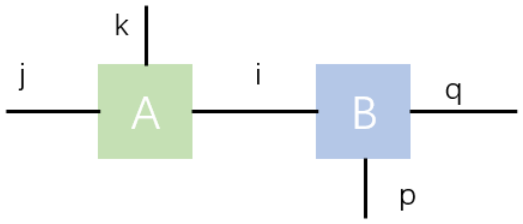


Figure 2: Example of a contraction

One may immediately notice that rank of both tensors get added then subtracted by 2 and that if the axis that we choose to contract over isn't of the same size, than the operation is not possible.
The purpose of this specific operation is to generalize the matrix vector product, the matrix-matrix product and the scalar product.

Example Let's do the contraction of a rank 2 tensor (matrix) with a rank 1 tensor (vector).

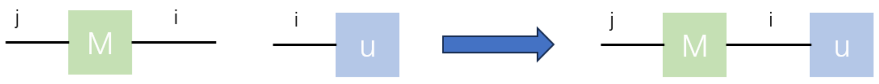


Figure 3: Example of a matrix vector contraction

Apply the formula above and the resulting rank 1 tensor (as it should be) sees its j^{th} component become :

$$[Mu]_j = \sum_i M_{ij} u_i$$

Which is exactly the result of a traditional matrix vector product.

1.2 BSDE

1.2.1 Some stochastic process theory reminders

In this section we recall some elementary results of stochastic process theory. If one is already familiar with the results, they may proceed to the next section.

- A **filtration**, usually noted $(\mathcal{F}_t)_{t\in\mathbb{R}^+}$, is an indexed family where for all $t \in \mathbb{R}^+$, $\mathcal{F}_t$ is a σ-algebra and for all $s > t$, $\mathcal{F}_t \subset \mathcal{F}_s$.

- A Gaussian process is a collection of processes $(X_t)_{t\in\mathcal{T}}$ such that for all n the vector $(X_{t_1}, ...X_{t_n})$ is Gaussian. This allows us to completely describe the process with only two functions via the following theorem.

  Theorem Let $\mathcal{T}$ be any set, $\mu : \mathcal{T} \to \mathbb{R}$ and $\Sigma : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$ be two functions. Assume that Σ is symmetric, positive definite. Then there exists a Gaussian process indexed by $\mathcal{T}$ such that

  1. $E[X_t] = \mu(t)$

  2. $Cov(X_s, X_t) = \Sigma(s,t)$

  For the proof and further details, see [5] or any lecture on Gaussian process theory.

- The **Brownian motion** is a Gaussian process indexed by $\mathbb{R}^+$ with $\mu = 0$ and $\Sigma(s,t) = \min(s,t)$

1.2.2 Definition

A Backward Stochastic Differential Equation is a stochastic differential equation with a terminal condition. Formally, we set $T > 0$ a terminal time, $(B_t)_{t<T}$ a standard Brownian motion, $\mathcal{F}_t$ a filtration defined by $\mathcal{F}_t = \sigma(B_s : s \leq t)$ and ξ a random variable that is $\mathcal{F}_T$-measurable. This choice of filtration ensures that each state depends only on past states. We then consider the following equations :

$$dY_t = f(t, \omega, Y_t, Z_t) - Z_t dB_t, \quad Y_T = \xi$$

which is the differential form. One can write the equivalent integral form :

$$Y_t = \xi + \int_t^T f(t, \omega, Y_s, Z_s) dB_s - \int_t^T Z_s dB_s$$

A solution consists of a pair of processes (Y_t, Z_t). The function f is set to be know beforehand and to satisfy the necessary conditions.

The link between parabolic PDEs and BSDEs is not covered by this report. For more details, the curious reader may refer to [4] (for an in-depth introduction to BSDEs) or [7], on which most of this section's contents are based.

1.2.3 A variant for our application

Let B_t be a real valued Brownian motion and T a terminal time. Just like the ordinary equation, we can have coupled BSDEs with a caveat. We call a coupled backward forward differential equation

$$\begin{aligned}
&dX_t = \mu(t, X_t, Y_t, Z_t)\, dt + \sigma(t, X_t, Y_t)\, dB_t, \quad t \in [0,T],\\
&X_0 = \xi,\\
&dY_t = \varphi(t, X_t, Y_t, Z_t)\, dt + Z_t' \sigma(t, X_t, Y_t)\, dB_t, \quad t \in [0,T),\\
&Y_T = g(X_T).
\end{aligned}$$

Functions μ, σ, g and ξ are known beforehand and and we seek the appropriate processes X_t, Y_t, and Z_t. In [1], one can find the proof that this equation is linked with a PDE of the form

$$\frac{\partial u}{\partial t} = f(t, x, u, \nabla u, \nabla^2 u)$$

where f is determined in term of μ, σ, g. Applying Ito formula (see [3]), we have that

$$Y_t = u(t, X_t), \quad Z_t = \nabla Y_t$$

Again, refer to actual papers on the topic for more in depth details.

2 Neural Network

2.1 Motivation

In essence, Backward Stochastic Differential Equations (BSDEs) involve finding solutions within the class of stochastic processes, which can be viewed as random functions. Since neural networks are known for their ability to approximate functions universally, we draw inspiration from physics-informed neural networks, which can predict solutions to ordinary differential equations (ODEs). We apply a similar approach to BSDEs by expressing the problem in the form:

$$F(Y_t, Z_t, B_t) = 0$$

We then use a discretization scheme to approximate the solution and train a neural network to predict the function $t \to Y_t$. Essentially, our goal is to determine the best neural network through the training process to approximate this function.

2.2 The architecture and the Black-Scholes-Barenblatt equation

Set d (unless specified otherwise, it will be set to be 100) to be the dimension of the problem, we have the coupled BSDE (see [8]):

$$\begin{aligned} dX_t &= \sigma diag(X_T) dB_t, \quad t \in [0, T], \\ X_0 &= \xi, \\ dY_t &= r(Y_t - Z_t' X_t)dt + \sigma Z_t' diag(X_T) dB_t, \quad t \in [0, T), \\ Y_T &= ||X_T||^2. \end{aligned}$$

Where $\xi = (1, 0.5, 1, 0.5....) \in \mathbb{R}^d$ (we see here that d needs to be even).
The BSDE above is associated to the Black–Scholes–Barenblatt PDE :

$$\frac{\partial u}{\partial t} = -\frac{1}{2} Tr(\sigma^2 diag(X_t)^2 \nabla^2 u) - r(u - (\nabla u)' x)$$

With terminal condition $u(T, x) = ||x||^2$.
This equation admits the explicit solution

$$u(t, x) = \exp\left((r + \sigma^2)(T - t)\right) ||x||^2$$

The neural network is then tasked with predicting $u(t, X_t) = Y_t$, with linear layers with **sine** activation functions.

Using the loss function provided in [8],

$$\sum_{m=0}^{M}\sum_{n=0}^{N-1}|Y_m^{n+1}-Y_m^n-\Phi_m^n\Delta t^n-(Z_m^n)'\Sigma_m^n\Delta B_m^n|^2+\sum_{m=1}^{M}|Y_m^N-g(X_m^N)|^2$$

Yields the following result, on the Black-Scholes-Barenblatt Equation

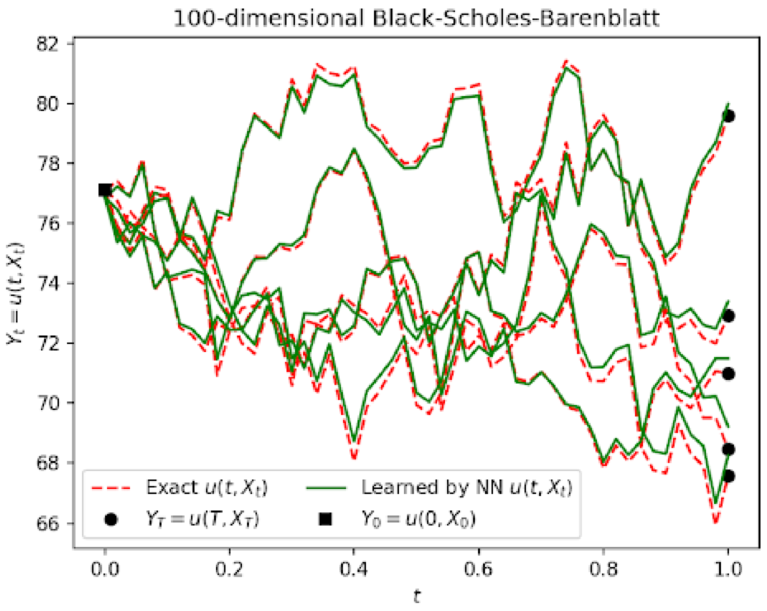


Figure 4: Results of NN

The relative error, calculated with the formula below

$$\textbf{Err} = \text{avg}_{\text{time}}\frac{||Y_{\text{pred}}-Y_{\text{ref}}||_2}{||Y_{\text{ref}}||_2\cdot\mathbb{1}_{\{Y_{\text{ref}}\neq 0\}}+\mathbb{1}_{\{Y_{\text{ref}}=0\}}}$$

yields 0.5% as well as a training time of $5,800$ seconds, with architecture of $(D,256)-3\times(256,256)-(256,1)$. Accounting for bias, we have roughly $200,000$ parameters into our network.
Note : We take into account the training time because if some parameters of the equations were changed, a complete re-training will be required.

2.3 The matrix product operator

The matrix product operator is a specific tensor network to represent what is essentially a series of **SVDs** operated on the matrix.

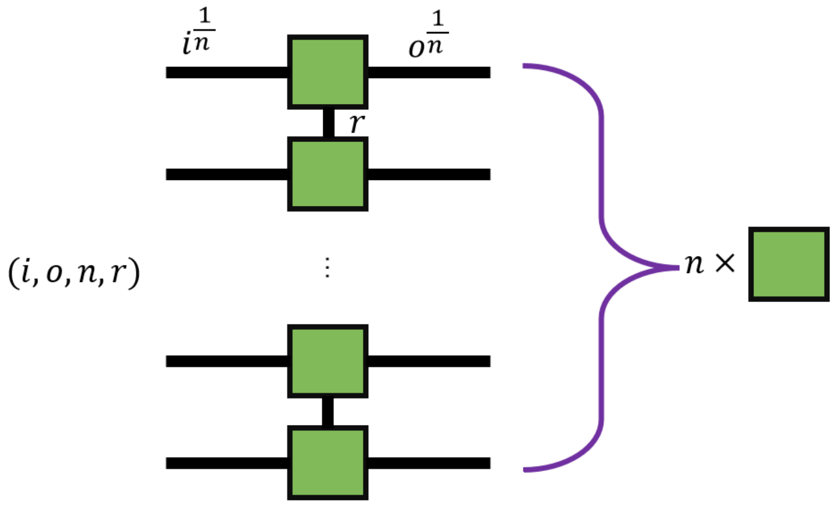


Figure 5: Notation for the MPO

2.4 Factorizing the hidden layers

The main issue with dense neural network is the numbers of parameters. Each dense matrix provides a quadratic scaling of the number of parameters. However, by factorizing a matrix into MPO states, we can drastically reduce the number of parameters.

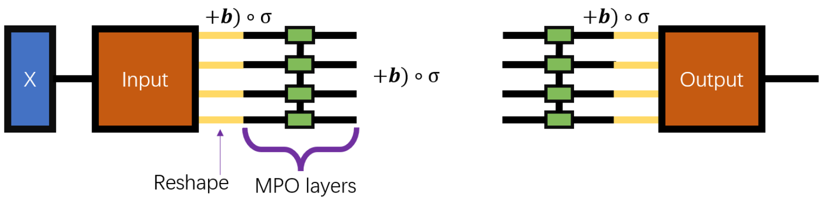


Figure 6: Neural Network with MPO

The main upside of this method is that we can still see this as an neural network, thus any method about neural networks can be applied to it. The question to explore is then if it is worth the effort, namely in terms of time cost and precision. The figure below provide the results of the neural networks with MPO inside.

In practice, weights are direcly stored as MPOs. The same error as above was calculated to be 0.6%, with a training time of 9,000 seconds. The chosen architecture for this plot was $(D, 8) - 3 \times (8, 8, 3, 4) - (8, 1)$, which roughly had 1000 parameters.

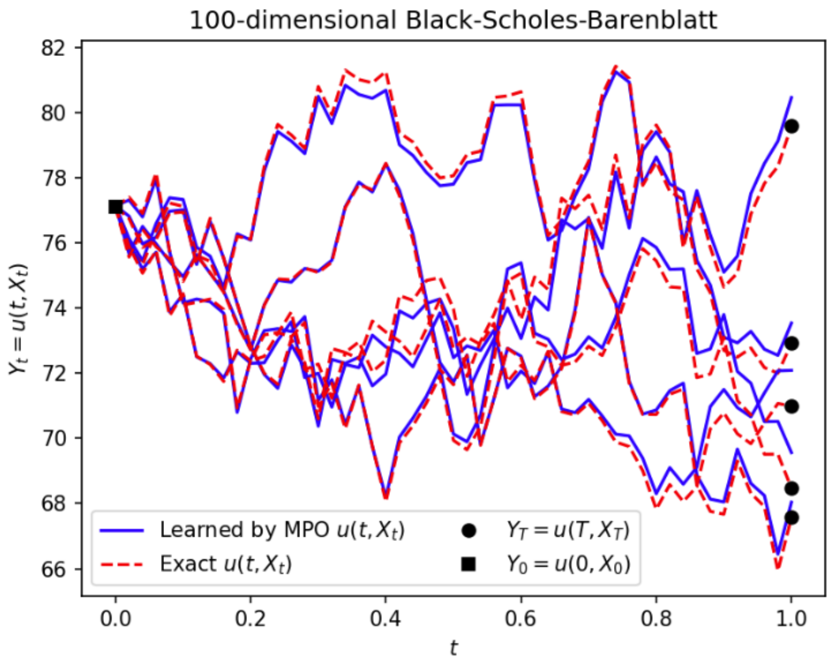


Figure 7: Results with MPOs

2.5 Convergence and possible optimisation

Previous results showcased that apart from the drastic decrease in the number of parameters, the MPO network yield roughly the same precision, at the cost of nearly double the training time. Furthermore, if we examine the rate at which the loss functions are decreasing, we see that the $(8, 8, 3, 4)$ MPO neural network (refered as MPO in this section) doesn't really add much value.

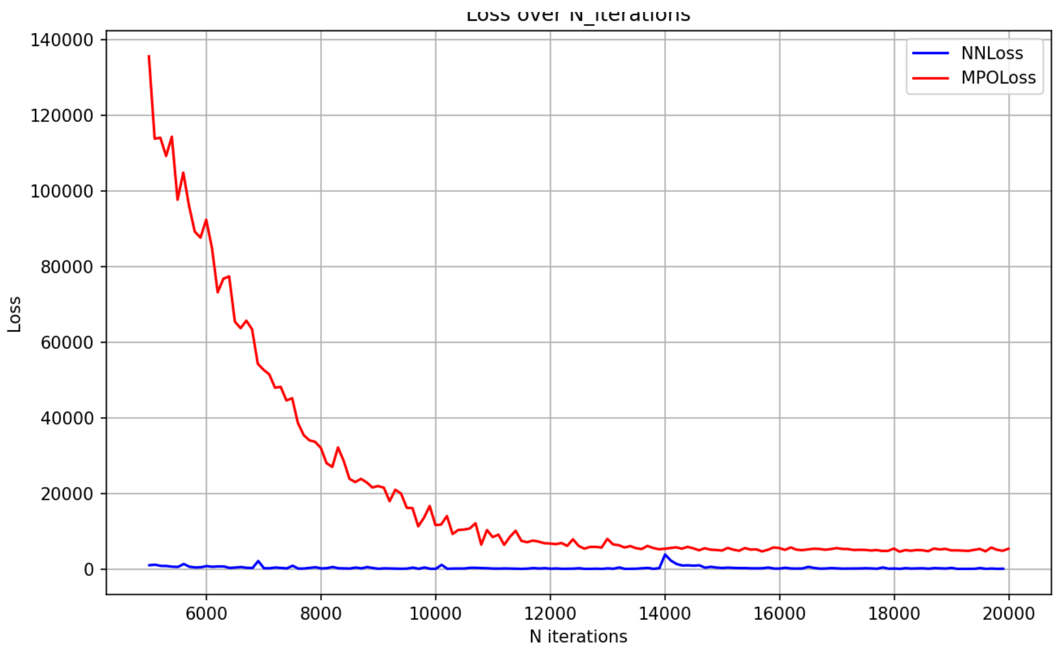


Figure 8: Loss functions for NN and MPOs as a function of the number of iterations

However, if we change the architecture, to $(D, 64) - 3 \times (64, 64, 3, 4) - (64, 1)$ (referred as MPO64), which has 2100 parameters, we see significant improvement on the convergence rate :

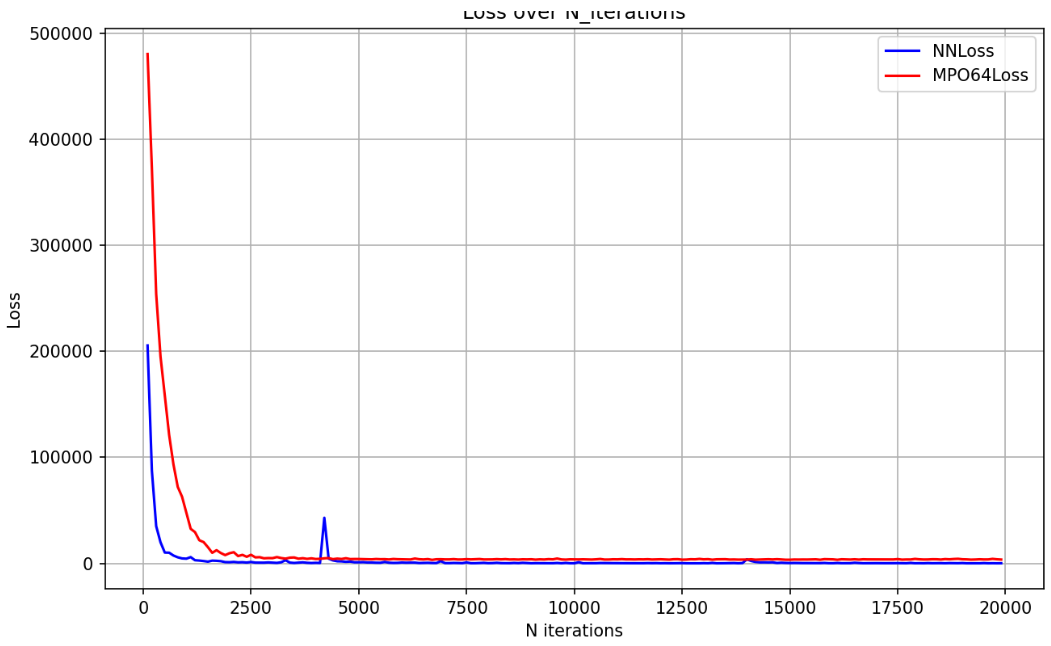
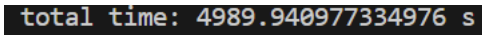


Figure 9: Loss function of MPO64 and NN

Furthermore, optimizing the computation for the MPO contraction can be yield a massive decrease in training time, 4, 000 seconds less than the precedent architecture :



```
total time: 4989.940977334976 s
```

Figure 10: Training time for optimized MPO64

A good optimization, including possible parallelizations, as well as an efficient implementation, could further reduce training time, but this remains to be implemented and tested.

2.6 Influence of the dimension of the problem

Below is the table of what we found when changing the dimension of the input layer. For the sake of time, we lessen the learning rate and the number of iterations.
Since we are comparing the relative precision of different types of networks, the learning rate is artificially set a bit higher to 0.01 instead of of 0.001 and the number of iterations is set to $10,000$

Type	Dimension	Architecture	Number of parameters	Error
NN	100	$(101,256) : (256,256) \times 3 : (256,1)$	222720	8.08%
MPO	100	$(101,256) : (256,256,2,8) \times 3 : (256,1)$	38912	7.5%
NN	4	$(5,16) : (16,16) \times 3 : (16,1)$	929	9.24%
MPO	4	$(5,16) : (16,16,2,4) \times 3 : (16,1)$	496	1.4%$*$

Table 1: Influence of dimension

These are pure empirical observations but can provide ideas for further testing.

* This is a one-time anomaly and we don't have an explanation for it.

3 Tensor Train

3.1 Motivation

In this section we forego the training aspect and focus on a local method. If we set $t_1, t_2...t_n$ a time-mesh, then the BSDE becomes using Euler–Maruyama approximation :

$$X_{n+1} = X_n + b(X_n, t_n)\Delta t + \sigma(X_n, t_n)\xi_{n+1}\sqrt{\Delta t}$$

and so, with Ito's lemma [3]

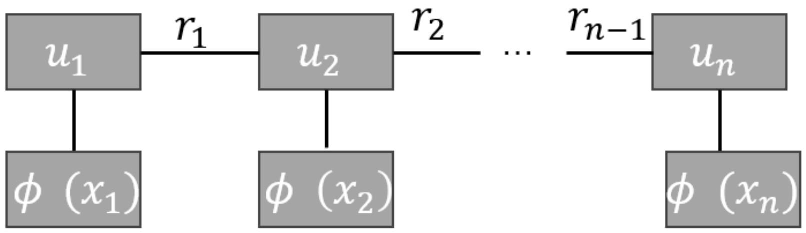
$$Y_{n+1} = Y_n + h_{n+1}\Delta t + Z_n \cdot \xi_{n+1}\sqrt{\Delta t}$$

Where $A_n = A_{t_n}$ for all processes, and ξ_n follows a Gaussian distribution of mean 0 and variance Δt. Note that $\Delta t = t_{n+1} - t_n$, we let go of an additional superscript because most of the time, the time mesh is regular.
Following the idea of [6], we are trying to find the best tensor network to approximate Y_i for all i, working backwards from the terminal condition $Y_N = g(X_N)$

3.2 The ALS algorithms

Following [6], where they chose to represent the solution V of the BSDE in the form of a tensor train contracted with feature vectors :

$$\phi(x_i) = \begin{bmatrix} \phi_1(x_i) \\ \phi_2(x_i) \\ \vdots \\ \phi_m(x_i) \end{bmatrix} \tag{1}$$

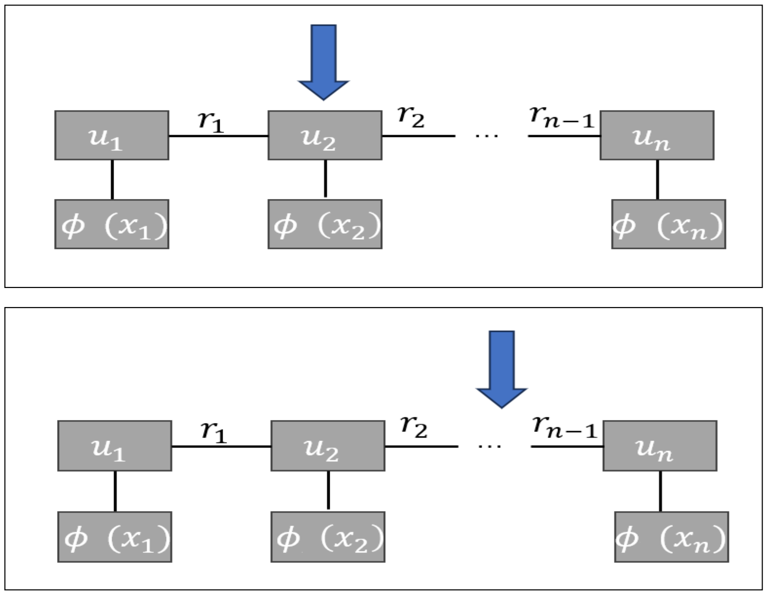


Via back propagation, the instance V_n is found by minimizing the following quantity

$$E[(V_n - h(X_n) - V_{n+1})^2] \tag{2}$$

This quantity is minimized by running any usual minimizing algorithms on the train, with input variable u_i and the rest of the u_j set. The pseudo-code is given as follows :

```
Algorithm 1: ALS
    Data: The number of tensors $N$, base functions $(\phi_i)_{1\leq i\leq N}$, rank of the tensor train
          $(r_1, r_2...r_{N-1})$
    Result: The solution $V$ at instant $n$ of the BSDE, represented by tensor train
    Initialisation: Set $u_i$ randomly, with size according to the rank, for all $1 \leq i \leq N$
    for $i = 1$ to $N$ do
        $u_i \leftarrow$ Minimization of (2) with respect to $(\phi_i)_{1\leq i\leq N}$
```



3.3 Results

We use this model with the simple stochastic process $dX_t = dB_t$. Hence $dV_t = \nabla_X V \cdot dB_t$ and we want to minimize $E[(V_n(X_n) - V_{n+1}(X_{n+1}))^2]$. Since we minimize with respect to only one tensor of the train at each time, this is a simple linear regression which is easy to implement. We solve for :

$$n = 2$$

$$\phi = \begin{bmatrix} x \mapsto 1 \\ x \mapsto x \\ x \mapsto x^2 \end{bmatrix} \tag{3}$$

We have the following plots.

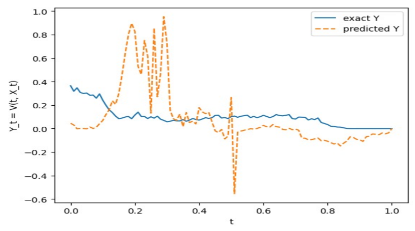


Figure 11: 10 simulations

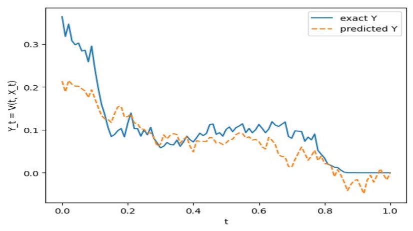


Figure 13: 50 simulations

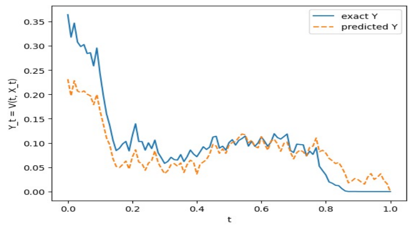


Figure 12: 100 simulations

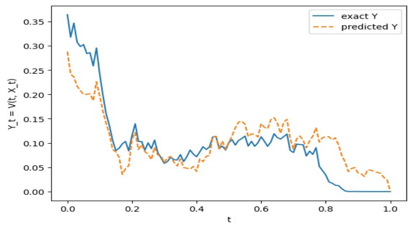


Figure 14: 1000 simulations

4 Conclusion

In this report, we have studied two different method to solve BSDE's using the quantum inspired technique that is tensor networks.
The first approach is to use neural network to find the best approximation of the function $t \to Y_t$. We compared the classical linear network with one where the weights matrixs was factorised into an MPO states. Using the Black-Scholes Barenblatt equation was a reference, we found that using a MPO network can yields similar results in terms of precision with a significant decrease in parameters, but efficiency like training time and convergence rate heavily depends on the chosen architecture and the code implementation.
The second approach is using a tensor network that we call tensor train to approximate each instance of Y_n, the value taken by the solution at the n-th step of the time-mesh. Using the ALS, which essentially is a coordinate by coordinate minimization on a chosen basis, we have see that the precision depends on the numbers of simulation. However, this method heavily depends on optimisation of the code below. Hence, further testing is necessary.

All in all, tensor network could be an interesting alternative to study in neural network in general, as any matrix can be factorized into the MPO. However, it is then much more delicate to find the correct architecture to yield to the correct results.

References

[1] F. Antonelli. Backward-forward stochastic differential equations, 1993.

[2] Jacob Biamonte. Lectures on quantum tensor networks, 2020.

[3] P. Cheridito, H. M. Soner, N. Touzi, and N. Victoir. Second-order backward stochastic differential equations and fully nonlinear parabolic pdes. *Communications on Pure and Applied Mathematics*, 60:1081–1110, 2007.

[4] Pardoux E. *Backward stochastic differential equations and viscosity solutions of systems of semilinear parabolic and elliptic PDEs of second order.* Springer, stochastic analysis and related topics vi edition, 1998.

[5] Erick Herbin. Processus aléatoire à temps discret, 2020. Lecture notes of Centrale-Supélec, course of Advanced Probabilities.

[6] Richter L., Sallant L., and Nusken N. Solving high-dimensional parabolic pdes using the tensor train format, 2021. arXiv:2102.11830v2 [stat.ML] 17 Jul 2021.

[7] Nicolas Perkowski. Backward stochastic differential equations: an introduction.

[8] Maziar Raissi. Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations, 2018.