Imperial College London

Department of Computing

Deep Learning of High-dimensional Partial Differential Equations

Author:
Alexis Laignelet

Supervisor:
Dr Panos Parpas

Submitted in partial fulfillment of the requirements for the MSc degree in Machine Learning of Imperial College London

September 2019

Abstract

Partial Differential Equations (PDE) have always been of great interest, both in the industrial world and in the research one. These equations can be found in a variety of fields, such as structural dynamics, fluid mechanics (Navier-Stokes), electromagnetism (Maxwell), and financial mathematics (Black-Scholes). Solving a PDE is often challenging, and strongly depends on the boundary conditions. If, as in most cases, there are no closed form solutions, numerical techniques are often used, like finite differences, or Monte Carlo simulations. Their main drawback, however, is that they do not scale well in high dimensions. This requires either to have large computational resources or to limit the size of the problem to be solved.
This project suggests an alternative technique relying on deep learning to solve PDEs in high dimensions. In particular, this report focuses on the Black-Scholes equation, widely used in finance to price derivatives. The proposed implementation is capable of finding the solution efficiently enough to also allow for additional terms, which are usually discarded for complexity reasons, to be added to the equation. This could have sizeable implications in the banking world, where, for example, many banks were severely damaged in 2008 as a consequence of ditching the impact of the risk of default from this equation.
Moreover, after building the aforementioned algorithm, the project focused on developing ways for optimising it that go beyond the standard computational optimisation techniques and that are more tailored to this specific task. The final optimisation algorithm developed was based on solving sub proximal problems, and performed widely better than the other benchmark techniques.

Acknowledgments

I truly thank Dr Panos Parpas for his availability, confidence and his wise advice during the whole project. His expertise in the field were of great help and he managed to leave me some space to look into details by myself while also guiding me to go in the right direction.

Contents

Chapter 1

Introduction

Conventional numerical techniques to solve Partial Differential Equation (PDE) suffer from the curse of dimensionality. This problem appears because they rely on spatio-temporal grids which can quickly become computationally expensive. For example, doing Monte Carlo simulation requires to sample exponentially from a probability distribution to estimate an expectation. The same curse of dimensionality appears for finite differences, where every dimension has to be discretised. The number of points needed to define a particular region of the space grows exponentially with the dimension (Figure 1.1).

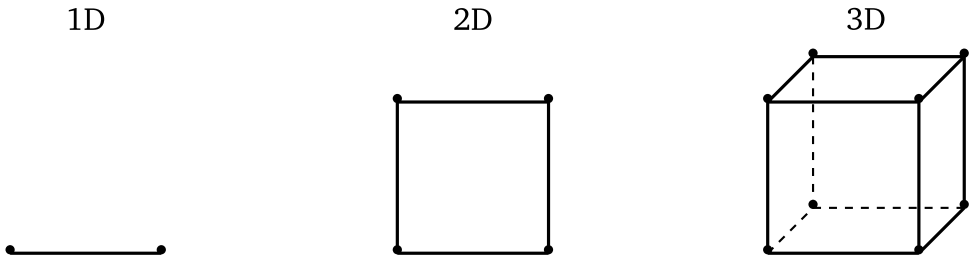

Figure 1.1: Illustration of the curse of dimensionality with the points needed to define a particular region within 1D, 2D and 3D spaces. In this case, 2^d points, with d the dimension, are needed.

This report suggests a different approach to solve these equations: using deep learning. This not only it gives an accurate solution to the equation, but it also scales very well with the dimension of the problem. However, deep learning cannot be applied directly to PDEs. Instead, the partial differential equation has to be converted into a set of forward and backward Stochastic Differential Equation (SDE) before deep learning can be used.
Once the forward and backward SDEs are set, a neural network (Raissi (2018b), Han et al. (2017), Henry-Labordere (2017), Beck et al. (2019), Sirignano and Spiliopoulos (2018)), is used to approximate the unknown solution. A lot of space is left in the choice of the architecture (Han et al. (2018), Weinan et al. (2017), Han et al. (2016)). The automatic differentiation from the deep learning library (PyTorch or TensorFlow) is then applied to compute the gradient through the neural network. There is no conventional dataset to train the model. The training set is composed

of sequences over time based on randomly generated Brownian motions. Learning on thousands of paths enables the model to then produce an accurate prediction for paths coming from the same distribution.
From an ethical point of view, there is no private data involved in this project. However, from a big-picture perspective, developing a way to solve PDEs more efficiently can involve serious ethical considerations, given that PDEs are used in countless industries, and could therefore have questionable applications such as, for example, military/terrorist use.
The following report focuses particularly on the Black-Scholes equation, and how deep learning can help to solve the equation in high dimensions. The goal is to solve the equation as fast as possible and with the best possible accuracy by training a neural network.
The first part of the report gives the necessary mathematical background to derive the forward and backward stochastic differential equations from the initial partial differential equation. A special focus is made on the Black-Scholes equation. For very simple cases, a closed formed solution is derived. The second part suggests the use of neural networks to solve the problem. A practical implementation is developed, and tests are conducted to ensure the correct behaviour of the model. Then, several stochastic gradient descent based techniques are explored and tested on toy examples to have an intuition on how it could help the resolution of the main problem. The same approach is applied to implicit methods, including proximal backpropagation. The last part of the report focuses on the numerical experiments done using the above techniques, applied to the main problem.

Chapter 2

Background

This chapter covers the mathematical material required to be able to build the neural network in the next chapter . It starts from the very fundamental concept of Brownian motion, moves to forward-backward stochastic equations, and then focuses on the Black-Scholes equation.

2.1 Brownian motion

2.1.1 Definition

A Brownian motion or a Wiener process constitutes the very fundamental of the following theory. It can be seen as the limit when $\delta t \to 0$ of a symmetric random walk with equal probabilities to go up or down (well described in Higham (2004)). Let N be the number of periods of time δt. A random walk can be defined as this additive process:

$$\begin{aligned} z(t_{k+1}) &= z(t_k) + \epsilon(t_k)\sqrt{\delta t} \\ t_{k+1} &= t_k + \delta t \end{aligned}$$

for $k = 0, 1, 2, ..., N$, where $z(0) = 0$ and the disturbance follows a standard normal distribution: $\epsilon(t_k) \sim \mathcal{N}(0, 1)$.

2.1.2 Properties

A Brownian motion can also be defined by its properties:

1. For all $j < k$ we have $z(t_k) - z(t_j) \sim \mathcal{N}(0, t_k - t_j)$,

2. For all $t_{k_1} < t_{k_2} \leq t_{k_3} < t_{k_4}$ the random variables $z(t_{k_2}) - z(t_{k_1})$ and $z(t_{k_4}) - z(t_{k_3})$ are independent,

3. $z(t_0) = 0$ with probability 1.

Normal distribution

The difference random variable defined by $z(t_k) - z(t_j)$ for $j < k$ is normally distributed: $z(t_k) - z(t_j) \sim \mathcal{N}(0, t_k - t_j)$.
The expectation is:

$$\begin{aligned}
\mathbb{E}[z(t_k) - z(t_j)] &= \mathbb{E}\left[\sum_{i=j}^{k-1} \epsilon(t_i)\sqrt{\delta t}\right] \\
&= \sqrt{\delta t}\left(\sum_{i=j}^{k-1} \mathbb{E}[\epsilon(t_i)]\right) \\
&= 0
\end{aligned}$$

The variance is:

$$\begin{aligned}
\mathrm{Var}[z(t_k) - z(t_j)] &= \mathbb{E}\left[\left(\sum_{i=j}^{k-1} \epsilon(t_i)\sqrt{\delta t}\right)^2\right] - \left(\mathbb{E}\left[\sum_{i=j}^{k-1} \epsilon(t_i)\sqrt{\delta t}\right]\right)^2 \\
&= \mathbb{E}\left[\left(\sum_{i=j}^{k-1} \epsilon(t_i)\sqrt{\delta t}\right)^2\right] \\
&= \mathbb{E}\left[\sum_{i=j}^{k-1} \epsilon(t_i)^2 \delta t\right] + \mathbb{E}\left[\sum_{\substack{i,i'=j \\ i \neq i'}}^{k-1} \epsilon(t_i)\epsilon(t_{i'})\delta t\right] \\
&= \mathbb{E}\left[\sum_{i=j}^{k-1} \epsilon(t_i)^2 \delta t\right] \\
&= \delta t\left(\sum_{i=j}^{k-1} \mathbb{E}[\epsilon(t_i)^2]\right) \\
&= \delta t\left(\sum_{i=j}^{k-1} \mathrm{Var}[\epsilon(t_i)] + \mathbb{E}[\epsilon(t_i)]^2\right) \\
&= \delta t(k - j) \\
&= t_k - t_j
\end{aligned}$$

Non overlapping intervals

If $t_{k_1} < t_{k_2} \leq t_{k_3} < t_{k_4}$, then the random variable $z(t_{k_2}) - z(t_{k_1})$ and $z(t_{k_4}) - z(t_{k_3})$ are independent.
This is because these differences are made up with different uncorrelated ϵ's.

Non differentiability

A Wiener process is not differentiable with respect to time. An intuition can be given by the following calculation. For times s and t such that $s < t$:

$$\begin{aligned}\mathbb{E}\left[\left(\frac{z(s)-z(t)}{s-t}\right)^2\right] &= \frac{1}{(s-t)^2}\mathrm{Var}[z(s)-z(t)] \\ &= \frac{s-t}{(s-t)^2} \\ &= \frac{1}{s-t} \xrightarrow[s-t\to 0]{} \infty\end{aligned}$$

2.2 Ito's lemma

In this section, a mathematical intuition is given on how to derive Ito's lemma.
Let f a function of two variables: $f(t, X_t)$, with X_t defined as an Ito process. This means:

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t$$

with μ and σ two continuous functions of (t, X_t), and W_t a Brownian motion.
Applying the Taylor expansion up to the second order terms:

$$df(t, X_t) \approx \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X_t}dX_t + \frac{1}{2}\frac{\partial^2 f}{\partial t^2}dt^2 + \frac{1}{2}\frac{\partial^2 f}{\partial X_t^2}dX_t^2 + \frac{\partial^2 f}{\partial t \partial X_t}dtdX_t$$

The idea is to evaluate the order of each term according to (Table 2.1) to make sure this expansion is relevant:

	dt	dX_t
dt	0	0
dX_t	0	dt

Table 2.1: Order analysis.

Taking only first-order terms:

$$df(t, X_t) \approx \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X_t}dX_t + \frac{1}{2}\frac{\partial^2 f}{\partial X_t^2}dX_t^2$$

Substituting X_t in the Taylor expansion:

$$df(t, X_t) \approx \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X_t}(\mu dt + \sigma dW_t) + \frac{1}{2}\frac{\partial^2 f}{\partial X_t^2}(\mu dt + \sigma dW_t)^2$$

As for the term in dX_t^2, we have to analyze the order of the different terms when developing (Table 2.2):

	dt	dW_t
dt	0	0
dW_t	0	dt

Table 2.2: Order analysis.

So we keep only the term in dW_t since it is the only term with first order magnitude.

$$df(t, X_t) = \frac{\partial f}{\partial t}dt + \frac{\partial f}{\partial X_t}(\mu dt + \sigma dW_t) + \frac{1}{2}\frac{\partial^2 f}{\partial X_t^2}\sigma^2 dt$$

And then, reorganizing the terms, leads to Ito's lemma:

$$df(t, X_t) = \left(\frac{\partial f}{\partial t} + \frac{\partial f}{\partial X_t}\mu + \frac{1}{2}\frac{\partial^2 f}{\partial X_t^2}\sigma^2\right)dt + \frac{\partial f}{\partial X_t}\sigma dW_t$$

2.3 Stochastic differential equation

2.3.1 Definition

A stochastic differential equation can be defined by a Brownian motion W_t, and μ and σ two continuous functions of (t, X_t):

$$X_t = X_0 + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s$$

An equivalent formulation is:

$$dX_t = \mu(t, X_t)dt + \sigma(t, X_t)dW_t \tag{2.1}$$

Under certain conditions, the existence and uniqueness of the solution can be proven. The function $\mu(t, X_t)$ is usually called the 'drift' and represents the general trend of the stochastic process, whereas the other term corresponds to the 'stochastic volatility'.
The above formula (Equation 2.1) can be written as:

$$X_{t+1} = X_t + \mu(t, X_t)dt + \sigma(t, X_t)\sqrt{t}\epsilon$$
$$\epsilon \sim \mathcal{N}(0, 1)$$

As a simple example, taking scalar values for $\mu(t, X_t)$ and $\sigma(t, X_t)$ leads to the following stochastic processes (Figure 2.1).

2.3.2 Feynman-Kac formula

The Feynman-Kac formula (Pham (2015), Van Casteren (2007)) establishes a link between partial derivative equations and stochastic differential equations (Ludvigsson (2013)).

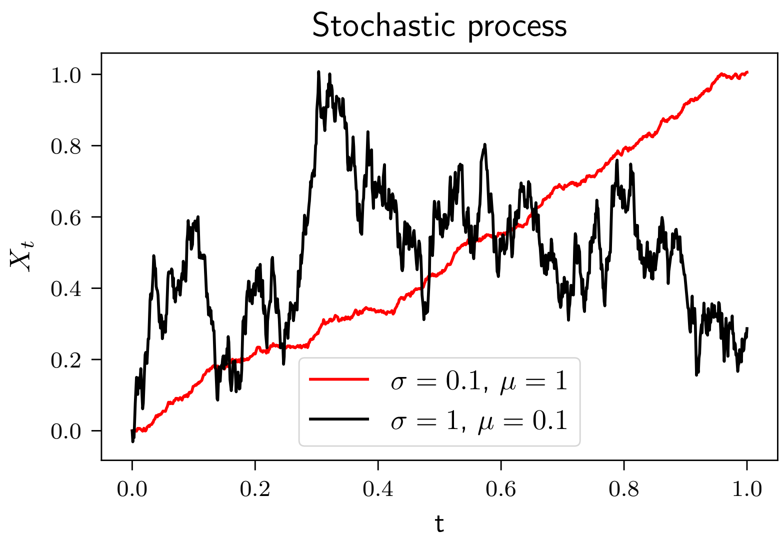

Figure 2.1: Stochastic processes for different values of drift and volatility.

Let μ, σ and ϕ such that:

$$\frac{\partial f}{\partial t} + \mu(t,x)\frac{\partial f}{\partial x} + \frac{1}{2}\sigma^2(t,x)\frac{\partial^2 f}{\partial x^2} = 0$$

with the final condition $f(x,T) = \phi(x)$.
It can be shown that the solution of the above equation is:

$$f(x,t) = \mathbb{E}[\phi(X_T)|X_t = x]$$

Where (X_t) is the solution of the SDE:

$$dX_t = \mu(t,X_t)dt + \sigma(t,X_t)dW_t$$

To get to this result let consider the Ito process $Y_t = f(t,X_t)$ when f is solution of the PDE.

$$\begin{aligned}dY_t =& \frac{\partial f}{\partial x}(t,X_t)dX_t + \frac{\partial f}{\partial t}(t,X_t)dt + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(t,X_t)dX_t \\ =& \frac{\partial f}{\partial x}(t,X_t)(\mu(t,X_t)dt + \sigma(t,X_t)dW_t) + \frac{\partial f}{\partial t}(t,X_t)dt + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(t,X_t)\sigma^2(t,X_t)dt \\ =& \left(\frac{\partial f}{\partial x}(t,X_t)\mu(t,X_t) + \frac{\partial f}{\partial t}(t,X_t) + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(t,X_t)\sigma^2(t,X_t)\right)dt+ \\ & \frac{\partial f}{\partial x}(t,X_t)\sigma(t,X_t)dW_t\end{aligned}$$

Since f verifies the PDE, the terms in dt cancel out, and then:

$$dY_t = \frac{\partial f}{\partial x}(t,X_t)\sigma(t,X_t)dW_t$$

Integrating this equation from t to T:

$$Y_T - Y_t = \int_t^T \frac{\partial f}{\partial x}(t, X_t)\sigma(t, X_t)dW_t$$

And then, taking the expectation, since the Brownian motion has a null expectation:

$$\mathbb{E}[Y_T] - \mathbb{E}[Y_t] = \mathbb{E}[f(X_t, T)|X_t = x] - \mathbb{E}[f(X_t, t)|X_t = x] = 0$$

So, finally

$$f(x,t) = \mathbb{E}[f(X_t, t)|X_t = x] = \mathbb{E}[f(X_t, T)|X_t = x] = \mathbb{E}[\phi(X_t)|X_t = x]$$

This is usually where the Monte Carlo simulation is done (Bouchard and Touzi (2004)), in order to evaluate this expectation. Instead of doing this, we derive the forward and backward equations, and make the neural network learn Y_t.

2.3.3 Forward equation

The forward equation is given by:

$$X_t = X_0 + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s$$

Where μ and σ are two continuous functions of (t, W_t), with W_t defined as a Brownian motion.

2.3.4 Backward equation

Based on Gobet (2016) and Perkowski (2011), let:

$$\begin{aligned}
&X_t = X_0 + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s \\
&\frac{\partial u}{\partial t}(t,x) + \mu(t,x)\frac{\partial u}{\partial x}(t,x) + \frac{1}{2}\sigma(x,t)^2\frac{\partial u^2}{\partial x^2}(t,x) + g(t, x, u(t,x), \nabla u(t,x)) = 0
\end{aligned}$$

Then, the stochastic processes (X, Y, Z) define as:

$$\begin{aligned}
Y_t &= u(t, X_t) \\
Z_t &= \nabla u(t, X_t)\sigma(t, X_t)
\end{aligned}$$

satisfy the following:

$$\begin{aligned}
X_t &= X_0 + \int_0^t \mu(s, X_s)ds + \int_0^t \sigma(s, X_s)dW_s \\
Y_t &= f(X_T) + \int_t^T g(s, X_s, Y_s, Z_s[\sigma(s, X_s)]^{-1})ds - \int_t^T Z_s dW_s
\end{aligned}$$

The second equation is called the backward equation. This result can be shown starting from:

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

And applying the Ito's formula on u and X:

$$\begin{aligned}dY_s =& du(s, X_s)\\ =& \left(\frac{\partial u}{\partial t}(s, X_s) + \mu(s, X_s)\frac{\partial u}{\partial X_s}(s, X_s) + \frac{1}{2}\sigma(X_s, s)^2\frac{\partial u^2}{\partial X_s^2}(s, X_s)\right) ds+\\ &\nabla u(s, X_s)\sigma(s, X_s)dW_s\\ =& - g(s, X_s, u(s, X_s), \nabla u(s, X_s))ds + \nabla u(s, X_s)\sigma(s, X_s)dW_s\end{aligned}$$

By integration between $s = t$ and $s = T$:

$$u(T, X_T) = Y_t - \int_t^T g(s, X_s, u(s, X_s), \nabla u(s, X_s))ds + \int_t^T Z_s dW_s$$

Since $u(T, .) = f(.)$:

$$Y_t = f(X_T) + \int_t^T g(s, X_s, Y_s, Z_s[\sigma(s, X_s)]^{-1})ds - \int_t^T Z_s dW_s$$

2.4 Black-Scholes equation

In the previous section, we have seen how the PDE, the forward and the backward stochastic equations are linked.
The goal is to solve the Black-Scholes equation in high dimensions. The usual methods (Monte Carlo or finite differences) used to solve PDEs will not work well due to the curse of dimensionality already mentioned. The use of forward and backward SDE is mentioned in Gobet (2016), but Y_t is only evaluated by a simple linear regression. A more sophisticated method would be to use a neural network to approximate the function $u(t_n, x)$ and automatic differentiation to evaluate $Du(t_n, x)$. This is what we suggest to explore in the following.

2.4.1 Partial differential equation

Based on Ito's Lemma we derive the Black-Scholes equation.
First, the following portfolio P is defined:

$$P = V + \delta S$$

This portfolio contains 1 option V and δ stocks. The stock price S follows a stochastic process:

$$dS = \mu S dt + \sigma S dW$$

Here, the functions $\mu(t,S) = \mu S$ with μ a constant, and $\sigma(t,S) = \sigma S$ with σ constant. We know from Ito's lemma that:

$$dV = \left(\frac{\partial V}{\partial t} + \mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}\right)dt + \sigma S\frac{\partial V}{\partial S}dW$$

So:

$$\begin{aligned}dP &= \left(\frac{\partial V}{\partial t} + \mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}\right)dt + \sigma S\frac{\partial V}{\partial S}dW + \delta(\mu S dt + \sigma S dW)\\ \iff dP &= \left(\frac{\partial V}{\partial t} + \mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + \delta\mu S\right)dt + \left(\sigma S\frac{\partial V}{\partial S} + \delta\sigma S\right)dW\end{aligned}$$

By choosing the number of stocks so that we eliminate the randomness (the Brownian motion W) in the previous equation:

$$\delta = -\frac{\partial V}{\partial S}$$

This leads to:

$$\begin{aligned}dP &= \left(\frac{\partial V}{\partial t} + \mu S\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} - \frac{\partial V}{\partial S}\mu S\right)dt\\ dP &= \left(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}\right)dt\end{aligned}$$

This becomes a non stochastic portfolio and so its value has to be the same has if it was on a bank account with a risk free interest rate r. This means:

$$dP = rPdt$$

Taking the previous expression and this one, leads to:

$$\left(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}\right)dt = rPdt$$

Then, replacing P by $V + \delta S$:

$$\begin{aligned}\left(\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2}\right)dt &= r(V + \delta S)dt\\ &= r\left(V - \frac{\partial V}{\partial S}S\right)dt\end{aligned}$$

Reorganizing the terms, leads to the Black-Scholes equation:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2\frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0$$

2.4.2 Closed-form solution

The main idea to derive a closed-from solution is to transform the Black-Scholes equation into the heat equation. The different transformations are independent of the derivative type, which means, they do not affect the terminal condition $V(T, X_T)$ stating the payoff of the derivative.
Starting from the partial differential equation:

$$\frac{\partial V}{\partial t} + rS\frac{\partial V}{\partial S} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} = rV$$

The first step is to apply the following transformations, with K the strike price:

$$\begin{aligned} x &= \ln\left(\frac{S}{K}\right) \\ \tau &= T - t \\ Z(x,\tau) &= V(Ke^x, T-\tau) \end{aligned}$$

The derivatives become:

$$\begin{aligned} \frac{\partial V}{\partial t} &= -\frac{\partial Z}{\partial \tau} \\ \frac{\partial V}{\partial S} &= \frac{1}{S}\frac{\partial Z}{\partial x} \\ \frac{\partial^2 V}{\partial S^2} &= -\frac{1}{S^2}\frac{\partial Z}{\partial x} + \frac{1}{S^2}\frac{\partial^2 Z}{\partial x^2} \end{aligned}$$

The resulting PDE for the function $Z(x,\tau)$ is:

$$\begin{aligned} &-\frac{\partial Z}{\partial \tau} + rS\frac{1}{S}\frac{\partial Z}{\partial x} + \frac{1}{2}\sigma^2 S^2\left(-\frac{1}{S^2}\frac{\partial Z}{\partial x} + \frac{1}{S^2}\frac{\partial^2 Z}{\partial x^2}\right) - rZ = 0 \\ \iff &\frac{\partial Z}{\partial \tau} + \left(\frac{\sigma^2}{2} - r\right)\frac{\partial Z}{\partial x} - \frac{1}{2}\sigma^2\frac{\partial^2 Z}{\partial x^2} + rZ = 0 \end{aligned}$$

The second step is to transform the above equation into the heat equation. We introduce the new function $u(x,\tau) = A(x,\tau) = e^{\alpha x + \beta\tau} Z(x,\tau)$, where the real number α and β are chosen so that the transformed PDE for u is the heat equation. The derivatives are the following:

$$\begin{aligned} \frac{\partial Z}{\partial \tau} &= A\left(\frac{\partial u}{\partial \tau} - \beta u\right) \\ \frac{\partial Z}{\partial x} &= A\left(\frac{\partial u}{\partial x} - \alpha u\right) \\ \frac{\partial^2 Z}{\partial x^2} &= A\left(\alpha^2 u - 2\alpha\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\right) \end{aligned}$$

Putting back the expression of the derivatives in the EDP:

$$
\begin{aligned}
&\frac{\partial u}{\partial \tau} - \beta u + \left(\frac{\sigma^2}{2} - r\right)\left(\frac{\partial u}{\partial x} - \alpha u\right) - \frac{1}{2}\sigma^2\left(\alpha^2 u - 2\alpha\frac{\partial u}{\partial x} + \frac{\partial^2 u}{\partial x^2}\right) + ru = 0 \\
\iff &\frac{\partial u}{\partial \tau} + \left(\alpha\sigma^2 + \frac{\sigma^2}{2} - r\right)\frac{\partial u}{\partial x} - \frac{1}{2}\sigma^2\frac{\partial^2 u}{\partial x^2} + \left((1+\alpha)r - \beta - \frac{\alpha^2\sigma^2 + \alpha\sigma^2}{2}\right)u = 0
\end{aligned}
$$

The idea is to cancel the terms $\frac{\partial u}{\partial x}$ and u. This means:

$$
\begin{aligned}
&\left\{\begin{array}{l} \alpha\sigma^2 + \frac{\sigma^2}{2} - r = 0 \\ (1+\alpha)r - \beta - \frac{\alpha^2\sigma^2+\alpha\sigma^2}{2} = 0 \end{array}\right. \\
\iff &\left\{\begin{array}{ll} \alpha & = \frac{r}{\sigma^2} - \frac{1}{2} \\ \beta & = \frac{r}{2} + \frac{\sigma^2}{8} + \frac{r^2}{2\sigma^2} \end{array}\right.
\end{aligned}
$$

The function $u(x,\tau)$ is solution of the PDE:

$$
\frac{\partial u}{\partial \tau} - \frac{\sigma^2}{2}\frac{\partial^2 u}{\partial x^2} = 0
$$

The heat equation admits the following solution (Green formula):

$$
u(x,\tau) = \frac{1}{\sqrt{2\sigma^2\pi\tau}}\int_{-\infty}^{\infty} e^{-\frac{(x-s)^2}{2\sigma^2\tau}}u(s,0)ds
$$

It corresponds to the convolution between the fundamental solution and the function $g(s) = u(s,0)$.

2.4.3 Example on a call option

For a call option, the terminal condition is: $V(S,T) = \max(0, S-K)$. This means:

$$
\begin{aligned}
u(x,0) &= e^{\alpha x}Z(x,0) \\
&= e^{\alpha x}V(Ke^x, T)
\end{aligned}
$$

So:

$$
u(x,0) = \left\{\begin{array}{ll} Ke^{\alpha x}(e^x - 1) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{array}\right.
$$

Putting back this expression of $u(s,0)$ in the Green formula:

$$
\begin{aligned}
u(x,\tau) &= \frac{1}{\sqrt{2\sigma^2\pi\tau}}\int_0^{\infty} e^{-\frac{(x-s)^2}{2\sigma^2\tau}}Ke^{\alpha s}(e^s - 1)ds \\
&= K\left(\frac{1}{\sqrt{2\sigma^2\pi\tau}}\int_0^{\infty} e^{-\frac{(x-s)^2}{2\sigma^2\tau}}e^{(\alpha+1)s}ds - \frac{1}{\sqrt{2\sigma^2\pi\tau}}\int_0^{\infty} e^{-\frac{(x-s)^2}{2\sigma^2\tau}}e^{\alpha s}ds\right)
\end{aligned}
$$

$$= K\left(e^{(1+\alpha)x+\frac{1}{2}\sigma^2\tau(1+\alpha)^2}\Phi\left(\frac{x+\sigma^2\tau(1+\alpha)}{\sigma\sqrt{\tau}}\right) - e^{\alpha x+\frac{1}{2}\sigma^2\tau\alpha^2}\Phi\left(\frac{x+\sigma^2\tau\alpha}{\sigma\sqrt{\tau}}\right)\right)$$

Where Φ is the distribution function of a normalised normal distribution:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{x} e^{-\frac{s^2}{2}}ds$$

Let us simplify the expression.

$$\begin{aligned}(1+\alpha)x+\frac{1}{2}\sigma^2\tau(1+\alpha)^2-(\alpha x+\beta\tau) =&(1+\alpha)x+\frac{1}{2}\sigma^2\tau(1+\alpha)^2-\alpha x-(1+\alpha)r\tau+\\ &\frac{\sigma^2\alpha^2\tau}{2}+\frac{\alpha\sigma^2\tau}{2}\\ =&x+\sigma^2\tau(\frac{1}{2}+\frac{3\alpha}{2}+\alpha^2)-(1+\alpha)r\tau\\ =&x+\sigma^2\tau\left(\frac{r^2}{\sigma^4}+\frac{r}{2\sigma^2}\right)-\left(\frac{1}{2}+\frac{r}{\sigma^2}\right)r\tau\\ =&x\end{aligned}$$

Similarly for the second term:

$$\begin{aligned}\alpha x+\frac{1}{2}\sigma^2\tau\alpha^2-(\alpha x+\beta\tau) &= \sigma^2\tau\alpha^2-(1+\alpha)r\tau+\frac{\alpha\sigma^2\tau}{2}\\ &= -r\tau\end{aligned}$$

So:

$$\begin{aligned}u(x,\tau) &= Ke^{\alpha x+\beta\tau}\left(e^x\Phi(d_1)-e^{-r\tau}\Phi(d_2)\right)\\ d_1 &= \frac{x+\sigma^2\tau(1+\alpha)}{\sigma\sqrt{\tau}}\\ d_2 &= d_1-\sigma\sqrt{\tau}\end{aligned}$$

This leads to the option price:

$$\begin{aligned}V(S,t) &= S\Phi(d_1)-Ke^{-r(T-t)}\Phi(d_2)\\ d_1 &= \frac{\ln(\frac{S}{K})+(r+\frac{\sigma^2}{2})(T-t)}{\sigma\sqrt{T-t}}\\ d_2 &= d_1-\sigma\sqrt{T-t}\end{aligned}$$

For a specific value of $K = 6$, $T = 1$, $\sigma = 1$ and $r = 0.1$ the results are the following (Figure 2.2).
For a given stochastic process, we can compute the price of the option call associated to the asset (Figure 2.3).

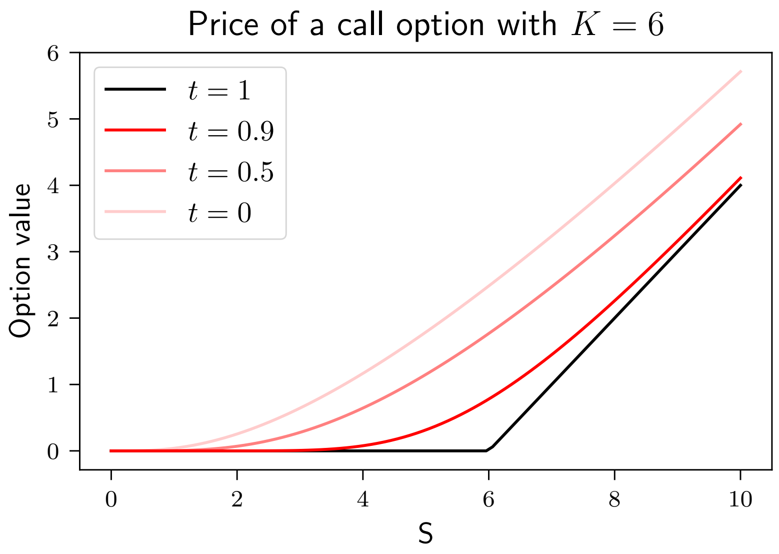

Figure 2.2: Price of a call option for $K = 6$ at different times, function of the price of the underlying asset.

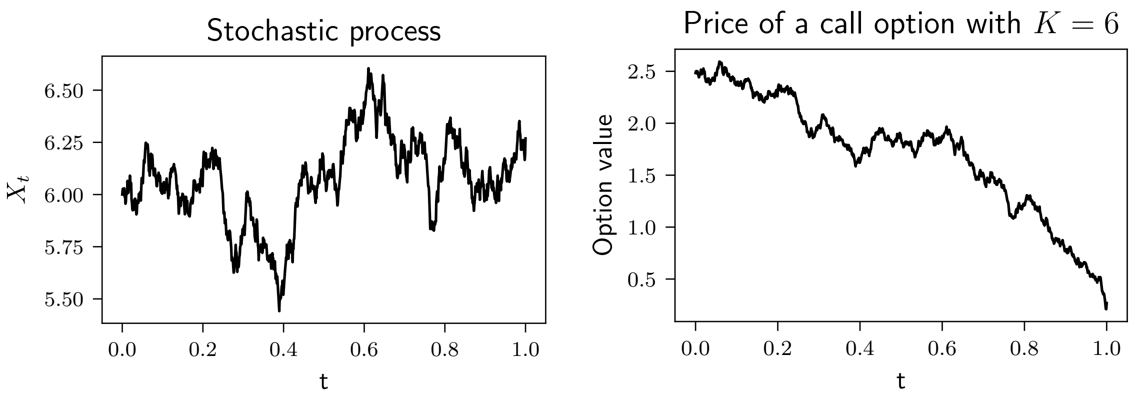

Figure 2.3: Stochastic process of the underlying asset, and call option value through time.

Chapter 3

Forward-Backward Stochastic Neural Network

The aim of this part is to explain how the neural network (denoted forward-backward stochastic neural network in the following) is built. This chapter covers the initial forward-backward stochastic equations used to formulate the problem in a way a neural network can take part. It goes through the choices made for the loss function, the architecture of the neural network itself and presents a few experimental results after a first training session.

3.1 Model

3.1.1 Forward-backward equations

The problem can be described by the following set of forward-backward equations, with the defined terminal condition (Bender and Denk (2007)):

$$\begin{cases} dX_t &= \mu(t, X_t, Y_t, Z_t)dt + \sigma(t, X_t, Y_t)dW_t \\ X_0 &= \xi \\ dY_t &= \phi(t, X_t, Y_t, Z_t)dt + Z_t^T \sigma(t, X_t, Y_t)dW_t \\ Y_T &= g(X_T) \end{cases}$$

First of all, the characteristics of the stochastic process X_t has to be defined. As done in Raissi (2018b), no drift is taken into account ($\mu = 0$), and the volatility is a constant σ that multiplies $\operatorname{diag}(X_t)$. This determines the characteristic of the path of the underlying assets X_t. Then, the function ϕ is defined as $\phi(t, X_t, Y_t, Z_t) = r(Y_t - Z_t^T X_t)$. This leads to the following set of equations:

$$\begin{cases} dX_t &= \sigma \operatorname{diag}(X_t)dW_t \\ X_0 &= \xi \\ dY_t &= r(Y_t - Z_t^T X_t)dt + \sigma Z_t^T \operatorname{diag}(X_t)dW_t \\ Y_T &= g(X_T) \end{cases}$$

The above forward-backward equation is equivalent to the Black-Scholes equation, with:

$$\begin{aligned} Y_t &= u(t, X_t) \\ Z_t &= \nabla u(t, X_t)\sigma(t, X_t) \end{aligned}$$

This leads to (Raissi (2018b)):

$$u_t = -\frac{1}{2}\operatorname{Tr}(\sigma^2 \operatorname{diag}(X_t^2) D^2 u) + r(u - (Du)^T x)$$

In the following, and for the sake of simplicity, the function g is chosen so that: $g(x) = \|x\|^2$. In our case, this means the norm of the vector X_t of dimension D at $t = T$.
Applied to a stochastic process, the result is the following (Figure 3.1):

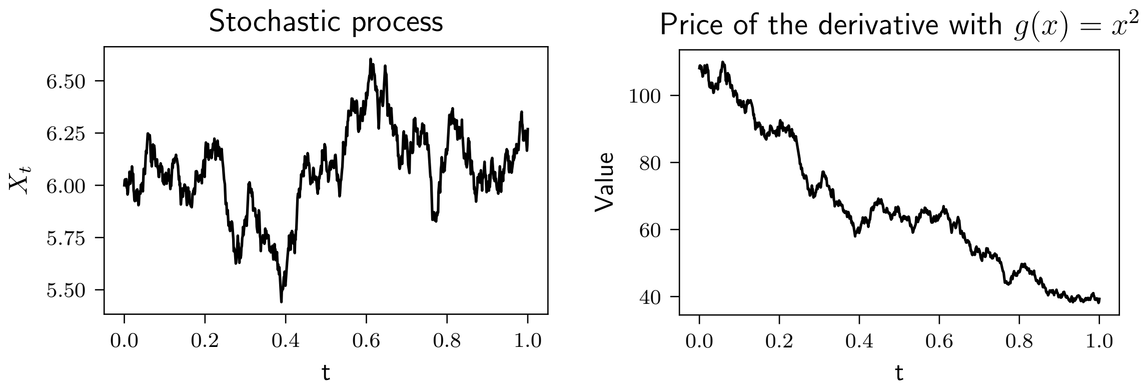

Figure 3.1: Stochastic process of the underlying asset, and value of the derivative through time with terminal condition $g(x) = x^2$.

We can verify that the function $u(x,t) = e^{(r+\sigma^2)(T-t)}g(x)$ is solution of the Black-Scholes equation:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -(r+\sigma^2)e^{(r+\sigma^2)(T-t)}g(x) \\ \frac{\partial u}{\partial x} &= e^{(r+\sigma^2)(T-t)}g'(x) \\ \frac{\partial^2 u}{\partial x^2} &= e^{(r+\sigma^2)(T-t)}g''(x) \end{aligned}$$

And then:

$$\begin{aligned} -(r+\sigma^2)g(x) + rxg'(x) + \frac{1}{2}\sigma^2 x^2 g''(x) &= -(r+\sigma^2)x^2 + 2rx^2 + \sigma^2 x^2 \\ &= rx^2 \end{aligned}$$

Which verifies the equation. The function $u(x,t) = e^{(r+\sigma^2)(T-t)}g(x)$ is then the true function that will enable us to evaluate the performance of the neural network.

3.1.2 Euler-Maruyama scheme

Applying the discretisation scheme to the previous set of equations, gives:

$$
\begin{aligned}
\Delta W_n &\sim \mathcal{N}(0,\, \Delta t_n) \\
X_{n+1} &\approx X_n + \mu(t_n, X_n, Y_n, Z_n)\Delta t_n + \sigma(t_n, X_n, Y_n)\Delta W_n \\
Y_{n+1} &\approx Y_n + \phi(t_n, X_n, Y_n, Z_n)\Delta t_n + (Z_n)^T \sigma(t_n, X_n, Y_n)\Delta W_n
\end{aligned}
\tag{3.1}
$$

3.1.3 Neural network

Based on the previous set of forward-backward stochastic equations, the idea is to train a neural network to predict Y_n from the inputs (t_n, X_n). The general architecture of the network is shown in Figure 3.2.

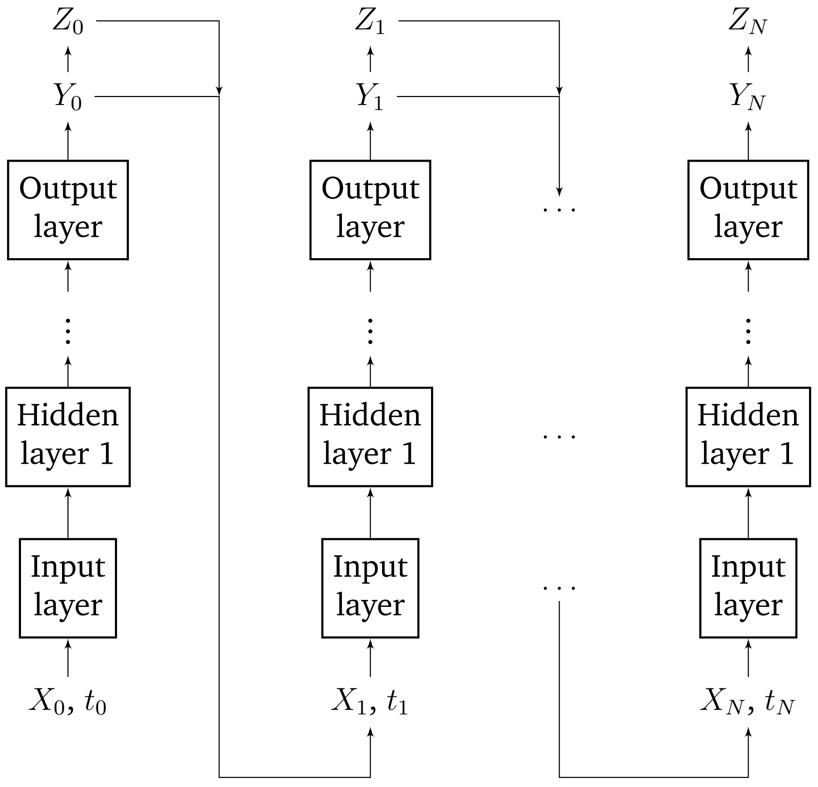

Figure 3.2: Architecture of the FBSNN

At each time step, X_t is computed by generating Brownian motions. Y_t is the prediction of the neural network. Then, Z_t is computed by taking the derivative of Y_t with respect to X_t, which means to compute the gradient of the neural network with respect to its inputs. This part is done using automatic differentiation from the deep learning packages in Python.

3.1.4 Loss function

The neural network takes (t_n, X_n) as an input and predicts Y_n. From Raissi (2018b), the loss function is designed such that it compares the prediction of the neural network (Y_{n+1} in the loss function below) with the actual value from the previously mentioned discretisation scheme. The number of time steps in the discretisation is N, whereas M is the batch size:

$$\sum_{m=1}^{M}\sum_{n=0}^{N-1}\left|Y_{n+1}^m - Y_n^m - \phi(t_n, X_n^m, Y_n^m, Z_n^m)\Delta t_n - (Z_n^m)^T\sigma(t_n, X_n^m, Y_n^m)\Delta W_n^m\right|^2 + \sum_{m=1}^{M}|Y_N^m - g(X_N^m)|^2 \tag{3.2}$$

As mentioned, the first term represents the difference between the next true value and the evaluation using the Euler-Maruyama scheme. The last term corresponds to the terminal condition, which means the last prediction is compared both to the result of the discretisation scheme and the terminal condition. As a reminder:

$$\begin{aligned} X_{n+1}^m &= X_n^m + \mu(t_n, X_n^m, Y_n^m, Z_n^m)\Delta t_n + \sigma(t_n^m, X_n^m, Y_n^m)\Delta W_n^m \\ Y_{n+1}^m &= u(t_n, X_n^m) \\ Z_{n+1}^n &= Du(t_n, X_n^m) \end{aligned} \tag{3.3}$$

In this machine learning problem, unlike usual ones, there is no concept of overfitting, since the whole point of the neural network is to find a solution that verifies the set of forward-backward stochastic equations. The solution, by definition, will then generalise to every cases.

3.2 Implementation

The starting point of the project is a TensorFlow 1.x implementation of the neural network named 'FBSNNs' available on the GitHub repository of Maziar Raissi Raissi (2018a).
The first challenge is to reproduce the results with our own PyTorch implementation. This task has two main objectives. First, implementing the code requires a good understanding of the mathematical concepts behind. Second, PyTorch appears to be more intuitive in general, and looks more promising to add new features and optimise the general implementation.

3.2.1 Architecture

The architecture of the neural network described in Raissi (2018b) is represented in Figure 3.3.
It is composed of 4 hidden layers with $\sin$ as an activation function. Each hidden layer has 256 neurons. The input layer is of dimension $D + 1$, composed by a vector

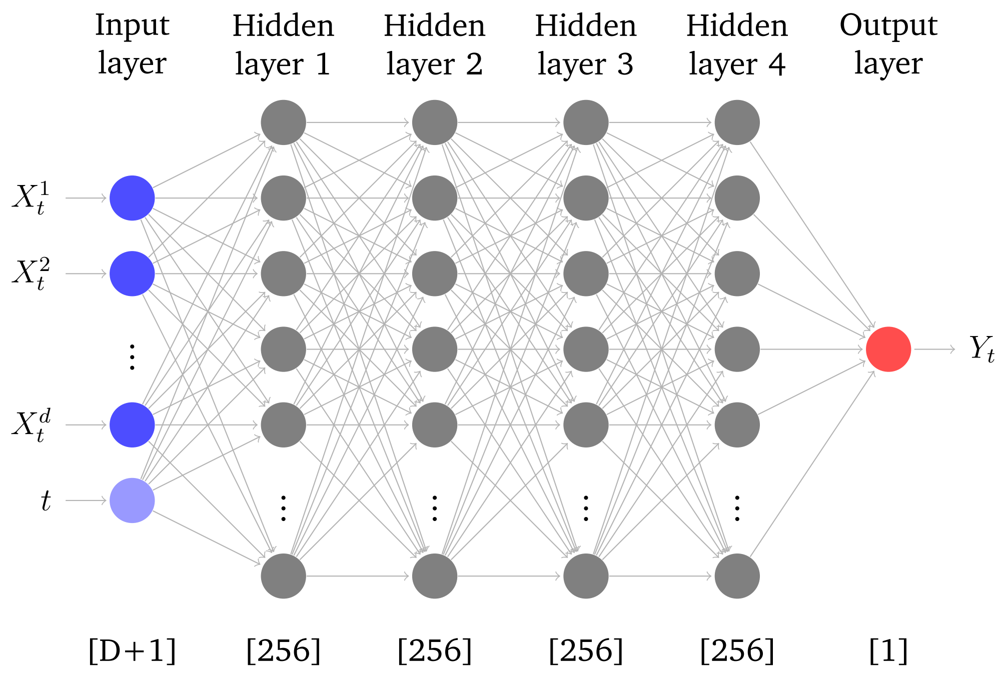

Figure 3.3: Architecture of the neural network

X_t of dimension D and a scalar t for the current time. The output layer is the prediction Y_t.

3.2.2 Pseudo code

As a reminder, the neural network tries to learn Y_t by optimising the loss function taking the sum over time of the difference between the prediction and the value computed by the Euler-Maruyama scheme. The derivative Z_t is then computed using automatic differentiation (autograd) thought the neural network (see Algorithm 1).

3.2.3 Testing the results

The main difference between TensorFlow and PyTorch is the way the graph is computed. Basically, it gathers the information about all the variables used in the neural network, linked by the operation between them. In PyTorch, the graph is computed dynamically whereas TensorFlow generates a static graph.
To correctly compare the two architectures, the randomness of the two different implementations is deleted. Since the code fundamentally relies on random processes, it implies the following:

- a random seed is placed on the generator of the Brownian motion,
- a random seed is placed on the initialization of the weights,
- the gradient descent is a batch gradient descent with a fixed learning rate.

```
Result: Train the model
Initialise attributes;
for number of iterations do
    generate Brownian motion $W$;
    generate initial value $X_0$;
    for number of time steps do
        inputs = $(t_n, X_n)$;
        compute the true value $\widetilde{Y_n}$; (see equation 3.1)
        $Y_n$ = model(input);
        $Z_n$ = autograd($Y_n$);
        compute $X_{n+1}$; (see equation 3.1)
        compute loss function; (see equation 3.2)
    end
    update weights;
end
```

Algorithm 1: Training algorithm

The comparison is done with a simple architecture, but with enough trajectories, time snapshots and dimensions, so the model is representative of the general behavior. The different graphs of the Figure 3.4 show the results after 2000 iterations. The parameter used are the following: $M = 10$, $N = 50$, $D = 10$, 2 hidden layers of 50 neurons, SGD with learning rate of 1e-5, $\sin$ activation.
Without any randomness, the results are the same between the two implementations, apart from very small numerical errors. After 2000 iterations, the training loss value is 10.9801 in PyTorch for 10.9804 in TensorFlow.
These tests ensure that our PyTorch implementation of the neural network is reliable and can serve as a base to further optimisations.

3.3 First experimental results

3.3.1 Training session

In this section, a more complex example is tested. The parameters are the following: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons each. The loss function is optimized using Adam with a learning rate sequence as described: 20k iterations at 1e-3, 30k at 1e-4, 30k at 1e-5 and another 20k at 1e-6 . The results are shown in the Figure 3.5.
We can also visualize the price of the derivative for the assets X of dimension $D = 100$, that matches the terminal condition: $Y_T = \|X_T\|^2$ (Figure 3.6).

3.3.2 Loss function

In analyzing the training loss, huge jumps can be noticed when the learning rate is 1e-3, during the first 20k iterations (see Figure 3.7 for details).

Changing the activation function for the ReLu one reduces drastically the noise as shown in Figure 3.8, but does not optimize efficiently the loss function since for the same number of iterations it decreases to 10^3 instead of 10^2. Even after a large number of iterations, the ReLu function does not enable the loss function to decrease to an acceptable small value.

3.3.3 Time efficiency

The PyTorch implementation appears to be slower than the TensorFlow one. So far all the experiments were conducted on Google Colab, either using Tesla K80 or Tesla T4 GPU. Time recorded over 100 iterations shows the following distribution within a single iteration (see Figure 3.9)
As expected, the time to compute fetch mini-batch is similar in both implementations since it only involves the creation of tensors. For the forward and backward, the stacked bars show substantial differences.
The 'forward + autograd' part is divided into two part: a constant part (negligible when the number of time snapshots is high) and a variable part dependent on N (number of time steps) which forms a loop. Within the loop itself, there are the 'forward' part to calculate Y_t and the automatic differentiation part to calculate Z_t (Figure 3.10).
What is interesting to notice is the fact the time does not change too much with the batch size (in our case the number of trajectories). This means, we can expect to have slightly better results in increasing the number of trajectories without impacting too much the computational expense, even if this notion is often debated.
Also, several tests were conducted by running the algorithm on more powerful GPU's. To do this, virtual machines were created using Google Cloud Computing. This platform enabled us to use Tesla P100 and V100 which are supposed to be more efficient. Practically speaking, the differences were minor and did not show any significant improvement. After monitoring the activity of the GPU's, we realised that only a small percentage was used. It could be an option to try to take the full advantage of powerful GPU's, or to do parallel calculations.

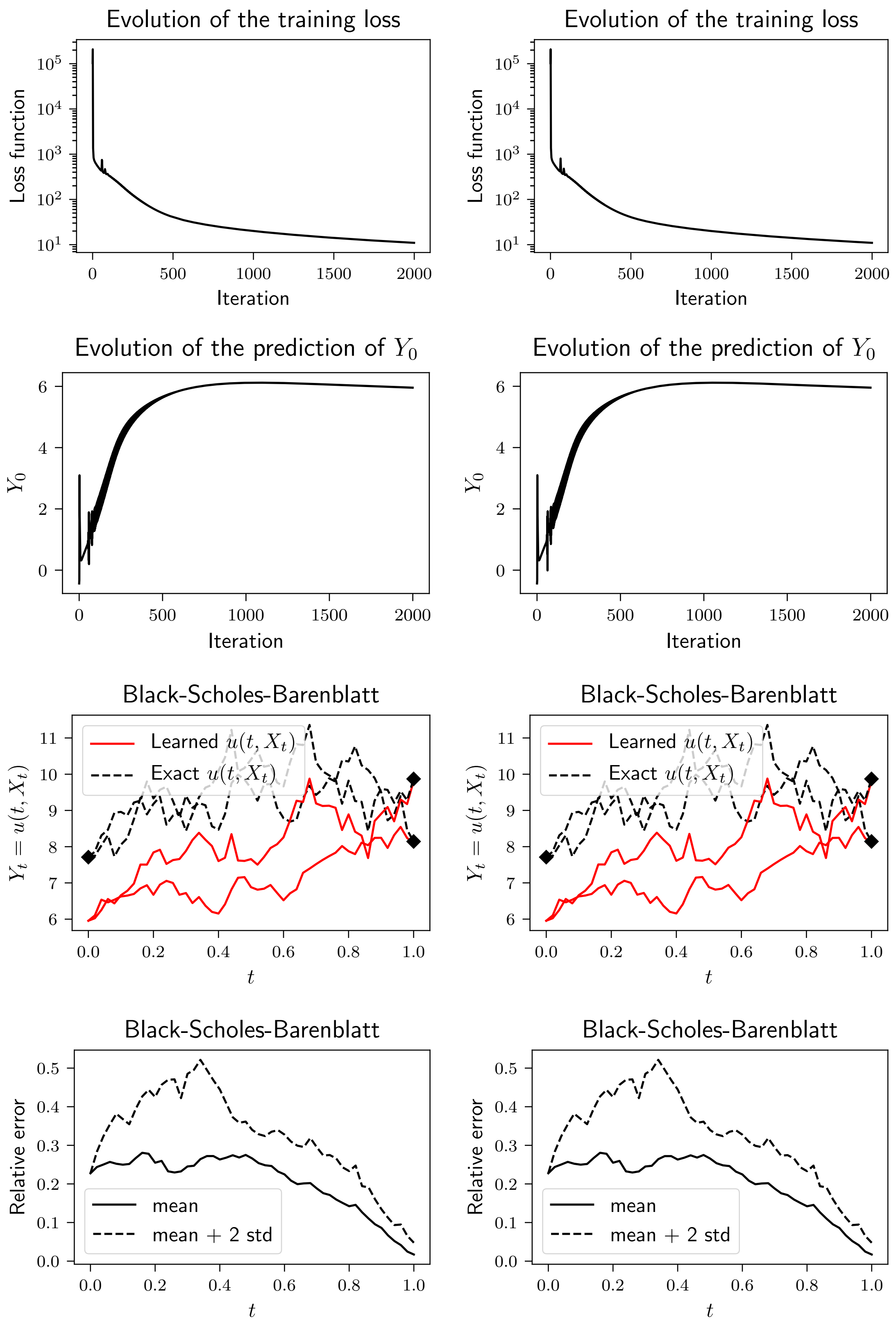

Figure 3.4: Comparison in between PyTorch (left) and TensorFlow (right) implementation for training loss, Y_0 prediction, learned solutions and error after 2000 iterations.

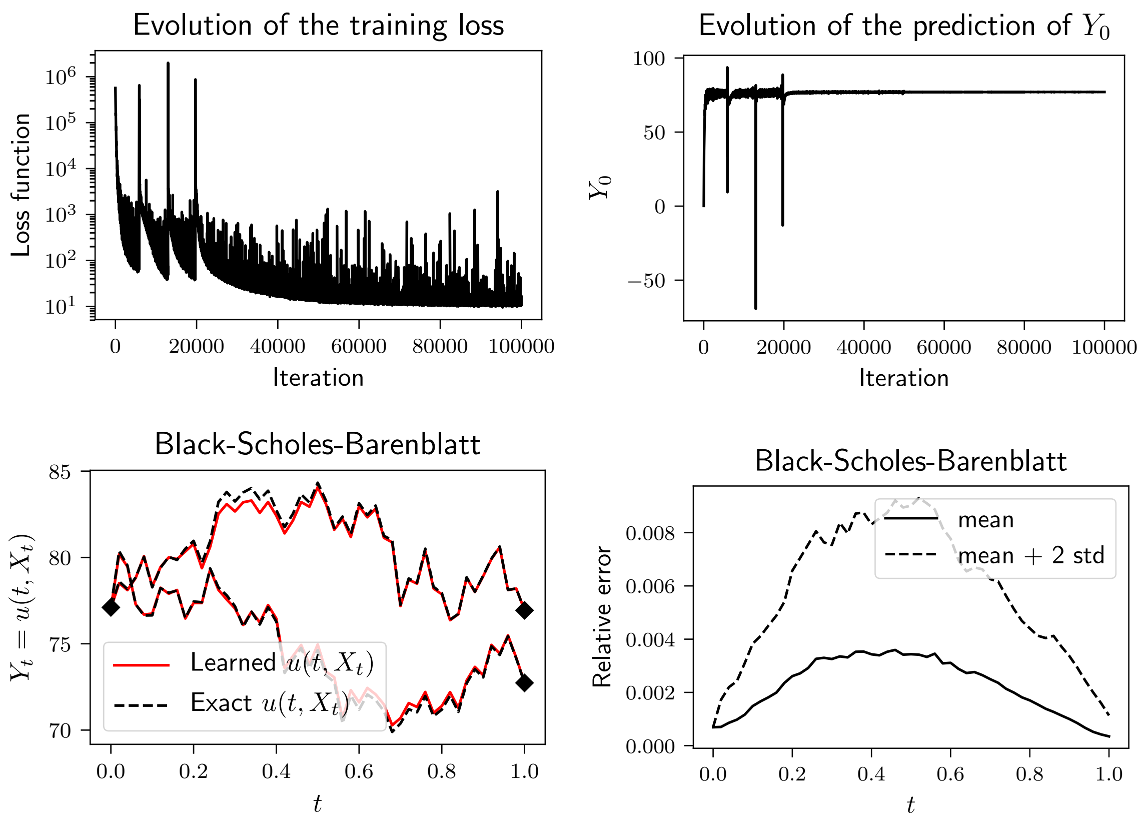

Figure 3.5: Training loss over 100k iterations with the following parameters: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons, Adam with learning rate according to: 20k iterations at 1e-3, 30k at 1e-4, 30k at 1e-5 and another 20k at 1e-6, $\sin$ activation function.

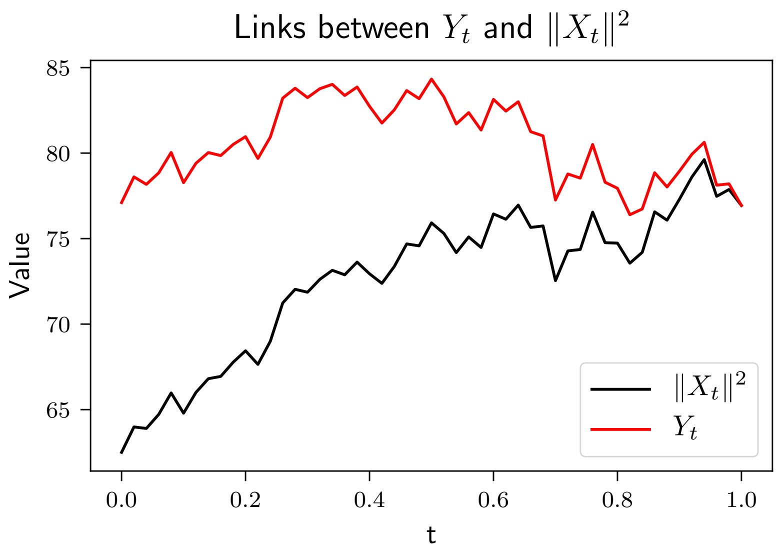

Figure 3.6: Comparison between Y_t and $\|X_t\|^2$, at the end of the training.

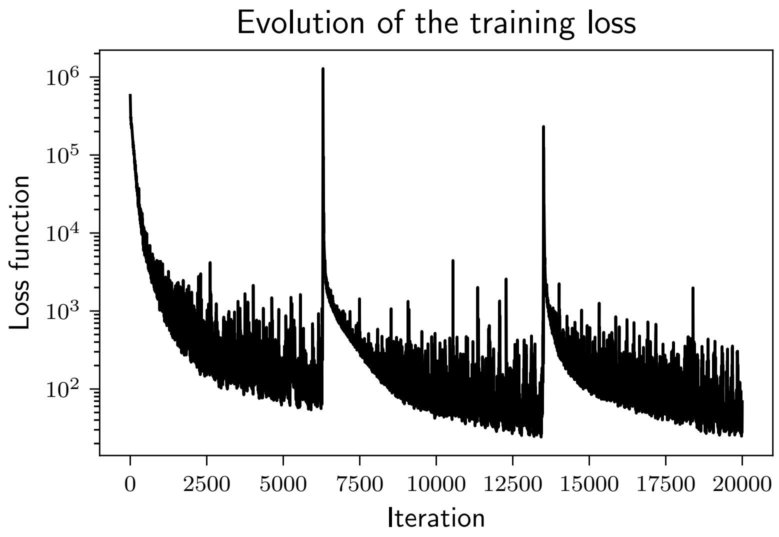

Figure 3.7: Training loss over 20k iterations with the following parameters: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons, Adam with learning rate of 1e-3, sin activation function

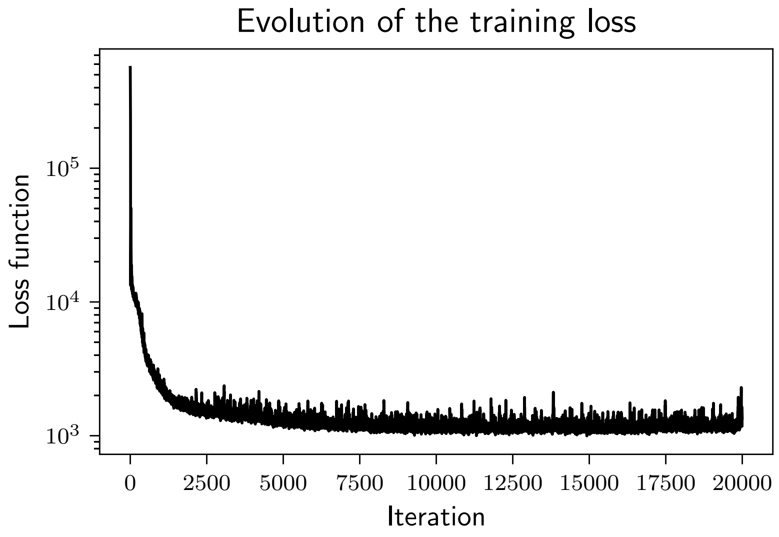

Figure 3.8: Training loss over 20k iterations with the following parameters: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons, Adam with learning rate of 1e-3, ReLu activation function.

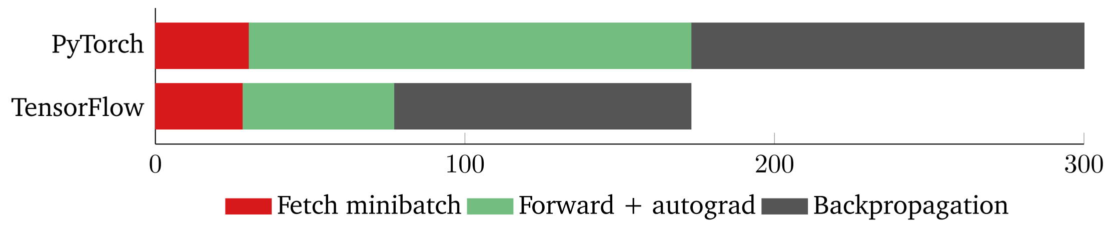

Figure 3.9: Time comparison done on a average of 100 iterations on Google Colab using Tesla K80 with the following parameters: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons, Adam with learning rate of 1e-3, $\sin$ activation function. Time in milliseconds.

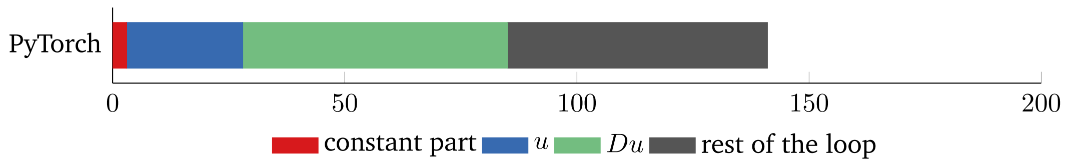

Figure 3.10: Time distribution within 'Forward + autograd' done on a average of 100 iterations run on Google Colab using Tesla K80 with the following parameters: $M = 100$, $N = 50$, $D = 100$, 4 hidden layers of 256 neurons, Adam with learning rate of 1e-3, sin activation function. Time is in millisecond.

Chapter 4

Stochastic Gradient Descent methods

From the conducted experiments, the main challenge seems to converge to a proper set of weights, in a reasonable amount of time. This is where the optimiser takes place. In the following, different techniques are explained and then tested on the forward-backward stochastic neural network problem. The very first algorithm is Stochastic Gradient Descent (SGD), and the two next are based on it. Even today, SGD algorithms are still of great interest (Vaswani et al. (2019), Toulis et al. (2016), Ruder (2016)).

4.1 Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD) is the simplest gradient descent based method, especially in our case, since one batch corresponds to the entire dataset. It can be formulated as this:

$$\theta_t = \theta_{t-1} - \tau \nabla L(\theta_{t-1})$$

Where θ is the parameters to optimise, and L the loss function.
The only hyperparameter in this algorithm is τ, the learning rate. Usually, there is a trade off between converging quickly (with high values) and avoiding divergence (not to high values). Let f a convex function such that:

$$f(x) = \frac{1}{2}x^2$$

SGD applied to this problem provides the following results (Figure 4.1).
This example is based on a convex function which is often not the case of the loss function. Let f the following non-convex function:

$$f(x) = (x - 0.5)^3 + 0.5\sin(16x + 4) + (2x)^4 - (3.5x + 0.2)^2$$

On a non-convex function, the main problem of this algorithm is that is converges to a local minimum, as shown in the following example (Figure 4.2):

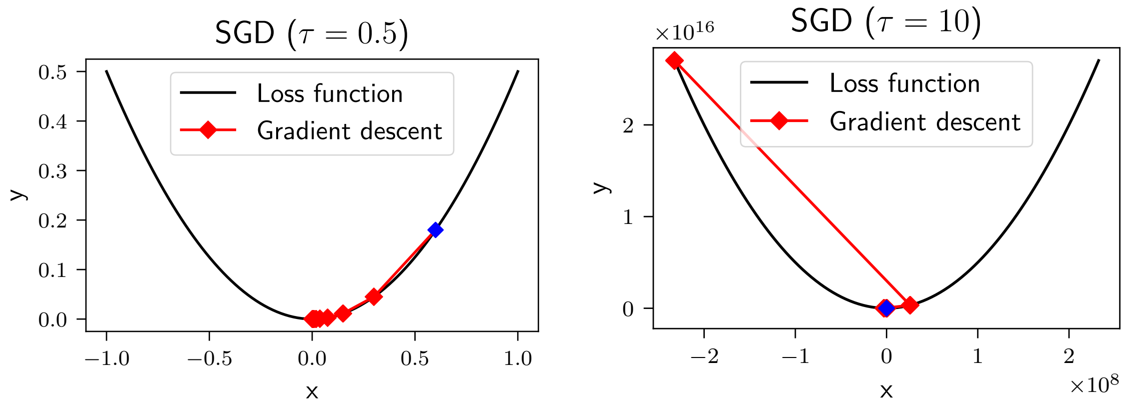

Figure 4.1: Stochastic Gradient Descent applied on a convex function, for $\tau = 0.5$ and $\tau = 10$. On the left, the algorithm converges to the solution. On the right side, the gradient descent diverges

4.2 Stochastic Gradient Langevin Dynamics

An additional term can be added to the previously described algorithm. The Stochastic Gradient Langevin Dynamics (SGLD) can be defined as this (well described in Kantas et al. (2019), and Brosse et al. (2018)):

$$\theta_t = \theta_{t-1} - \tau \nabla L(\theta_{t-1}) + \sqrt{\frac{2}{\beta}}\epsilon$$

Where ϵ is a standard Gaussian vector: $\epsilon \sim \mathcal{N}(0, 1)$.
This extra term introduces noise to the vanilla SGD. The new hyperparameter β adjusts the capacity of the gradient descent to explore the space (Figure 4.3). It is particularly important for non-convex loss function, as it may help to escape local minima.
With this algorithm, the new challenge is to wisely choose β so that the space is well explored, and a good level of convergence is kept.

4.3 Continuous Tempering Langevin Dynamics

An extension to the previous algorithm would be to make β change during the training. A first intuition could be to choose β dependent of the number of iterations, so that the exploration of the space is encouraged at the beginning, before switching to a convergence phase, with a greater value of β.
This leads to the famous 'simulated annealing' model (Pan and Jiang (2015)), which can be expressed this way:

$$\theta_t = \theta_{t-1} - \tau \nabla L(\theta_{t-1}) + \sqrt{\frac{2}{\beta(t)}}$$

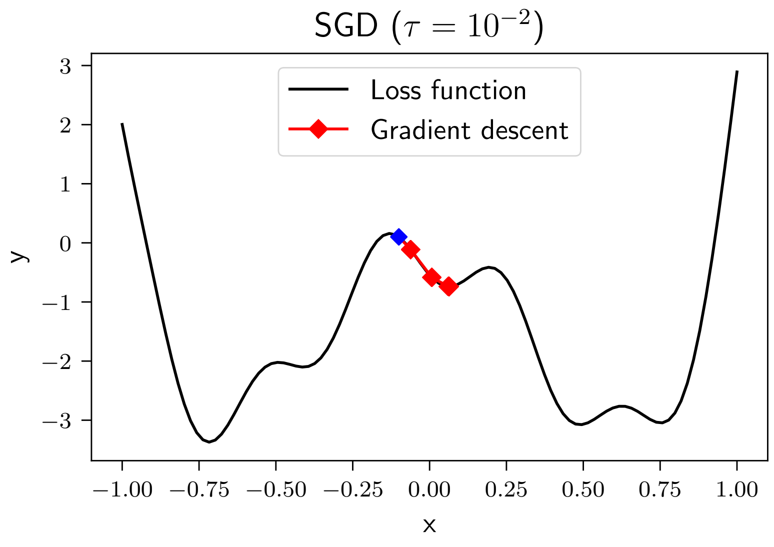

Figure 4.2: Stochastic Gradient Descent applied on a non-convex function. It converges to a local minimum.

Where:

$$\beta(t)^{-1} = k_B T(t)$$
$$T(t) = \frac{c}{\log(2+t)}$$

Here, k_B is the Boltzmann constant, and c a well chosen constant. This model can be even more sophisticated, and this is what is developed in (Ye et al. (2017)). Two phases are defined: sampling and optimisation.

4.3.1 Definition

It is generally accepted that sharp minima lead to poor generalisation, and on the contrary, flat minima often generalise better. Based on this result, and viewing the problem from a Bayesian perspective, flat minima can be assimilated to 'fat' mode in the probability distribution over the parameters. This means, the sampling phase is here to find the 'fat' modes, which concentrate most of the mass of the distribution. However, we can imagine these modes to be isolated from each other, which will require stochastic approximation techniques to overcome this issue.
The model suggests to start with the the Stochastic Gradient Langevin Dynamics:

$$\begin{cases} \theta_t & = \theta_{t-1} + \tau r_{t-1} \\ r_t & = r_{t-1} - \tau \nabla L(\theta_t) + \sqrt{\frac{2}{\beta}}\epsilon \end{cases}$$

The above problem can be written this way:

$$\begin{cases} d\theta & = r dt \\ dr & = -\nabla L(\theta) dt + \sqrt{\frac{2}{\beta}}\epsilon \end{cases}$$

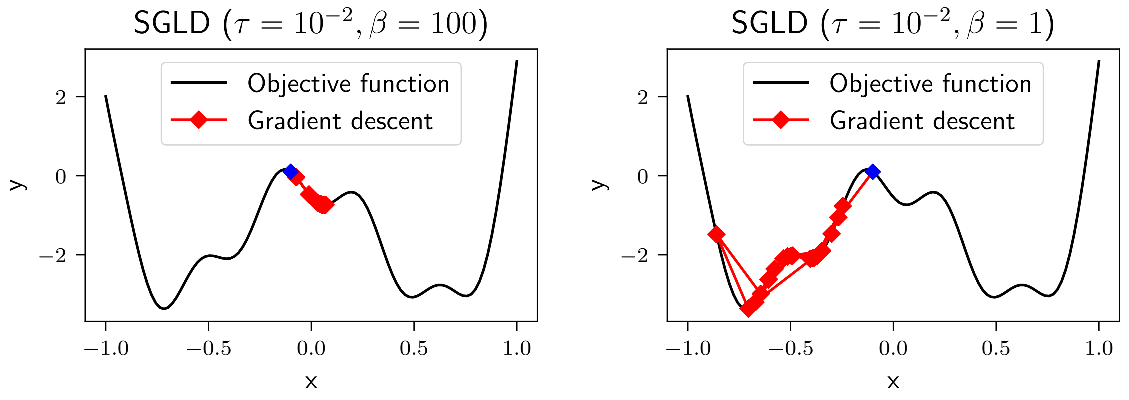

Figure 4.3: Stochastic Gradient Langevin Dynamics applied on a non-convex function, for $\beta = 1$ and $\beta = 100$. On the left, the algorithm converges to the local minimum. On the right side the space is more explored, and a better minimum is reached

With the usual learning rate equals to τ^2. A friction coefficient is then added:

$$dr = -\nabla L(\theta)dt - \gamma r dt + \sqrt{\frac{2}{\beta}}\epsilon$$

From a gradient descent perspective, it correspond to a momentum of $1 - \tau\gamma$. Inspired by temperature dynamics, to enable more effective space exploration, we introduce a function $\beta(\alpha)$, where α verifies the the two last equations:

$$\begin{cases} d\theta & = r dt \\ dr & = -\nabla L(\theta)dt - \gamma r dt + \sqrt{\frac{2\gamma}{\beta(\alpha)}}\epsilon \\ d\alpha & = r_\alpha dt \\ dr_\alpha & = h(\theta, r, \alpha)dt - \gamma_\alpha r_\alpha dt + \sqrt{2\gamma_\alpha} d\epsilon_\alpha \end{cases}$$

α is now the augmented variable which rules the inverse temperature $\beta(\alpha)$, and γ_α the corresponding friction coefficient. The function h links both the parameters θ we want to find, and the variable α, and is defined as:

$$h(\theta, r, \alpha) = -\frac{\partial}{\partial\alpha}H(\theta, r, \alpha, r_\alpha)$$
$$H(\theta, r, \alpha, r_\alpha) = g(\alpha)H(\theta, r) + \phi(\alpha) + r_\alpha^2/2$$

So, it leads to:

$$h(\theta, r, \alpha) = -\partial g(\alpha)H(\theta, r) - \partial\phi(\alpha)$$

The function $\phi(\alpha)$ is defined such that its gradient applies a force to the augmented variable α so it stays in a certain interval.

$$\partial\phi(\alpha) = \begin{cases} 0 & \text{if } |\alpha| \leq \delta' \\ C & \text{otherwise.} \end{cases}$$

As for the function $g(\alpha)$, the temperature scaling function, it can be built as a piece-wise polynomial function:

$$g(\alpha) = \begin{cases} 1 & \text{if } |\alpha| \leq \delta \\ 1 - S(3z^2(\alpha) - 2z^3(\alpha)) & \text{if } \delta < |\alpha| < \delta' \\ 1 - S & \text{if } |\alpha| \geq \delta' \end{cases}$$

Where $z(\alpha) = \frac{|\alpha| - \delta}{\delta' - \delta}$. An example of function $g(\alpha)$ is given in Figure 4.4.

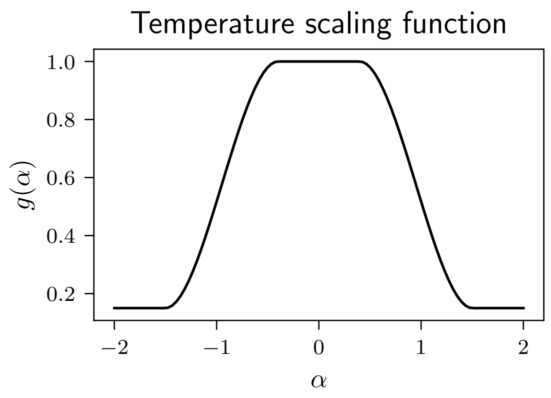

Figure 4.4: Temperature scaling function $g(\alpha)$ for Continuous Tempering Langevin Dynamics algorithm.

Both the functions $g(\alpha)$ and $\partial\phi(\alpha)$ make the variable α stay close to zero, which means $g(\alpha) = 1$. When the variable α goes greater or below δ', $g(\alpha)$ goes to 0.2. This increases the exploration, and puts more weight to the Gaussian. But, when it appends, the gradient of $\phi(\alpha)$ pulls α back to zero.
Then an additional term $V_b(\alpha)$ is added, to reflect the meta-dynamics that biases the extended Hamiltonian, which means:

$$H(\theta, r, \alpha, r_\alpha) = g(\alpha)H(\theta, r) + \phi(\alpha) + r_\alpha^2/2 + V_b(\alpha)$$

This additional term $V_b(\alpha)$ evolves this way:

$$V_{b,t}(\alpha) = V_{b,t-1}(\alpha) + w \exp\left(-\frac{(\alpha - \alpha_{t-1})^2}{2\sigma^2}\right)$$

The algorithm is fully explain in the paper (Ye et al. (2017)) and is synthesised below (Algorithm 2).
This algorithm involves a lot of hyperparameters, which gives the user a certain flexibility on how to use this gradient descent technique. However, these defaults values provide a good starting point:

$$\gamma = \frac{1 - c_m}{\tau}, \text{ with } c_m \in [0, 1]$$
$$\gamma_\alpha = \frac{1}{\tau}$$

```
Result: Optimise the loss function
Initialise attributes:
$r_0 \sim \mathcal{N}(0, I)$, $\alpha_0 = 0$, $r_{\alpha,0} \sim \mathcal{N}(0, 1)$ and $V_{b,0}(\alpha_0) = 0$
for number of iterations do
    if iteration $t < L_s$ then
        This is the exploration regime:
        Sample $\epsilon \sim \mathcal{N}(0, 1)$ and $\epsilon_\alpha \sim \mathcal{N}(0, 1)$
        $\theta_t = \theta_{t-1} + \tau r_{t-1}$
        $r_t = (1 - \tau\gamma)r_{t-1} - \tau \nabla L(\theta_t) + \sqrt{\frac{2\tau\gamma}{g(\alpha_{t-1})}}$
        $\alpha_t = \alpha_{t-1} + \tau r_{\alpha,t-1}$
        $r_{\alpha,t} = (1 - \tau\gamma_\alpha)r_{\alpha,t-1} + \tilde{h}(\theta_t, r_t, \alpha_t)\tau + \sqrt{2\tau\gamma_\alpha}\epsilon_\alpha$
        $\tilde{h}(\theta_t, r_t, \alpha_t) = -\partial g(\alpha_t)\tilde{H}(\theta_t, r_t) - \partial\phi(\alpha_t) - \frac{V_{b,t}(\alpha_{k^*+1}) - V_{b,t}(\alpha_{k^*})}{\frac{2\delta'}{K}}$
        where $k$ is the bin in which $\alpha_t$ is located
        $V_{b,t}(\alpha) = V_{b,t-1}(\alpha) + w \exp\left(-\frac{(\alpha - \alpha_t)^2}{2\sigma^2}\right)$
    else
        This is the optimisation regime:
        $\theta_t = \theta_{t-1} + \tau r_{t-1}$
        $r_t = (1 - \tau\gamma)r_{t-1} - \tau \nabla L(\theta_t)$
    end
end
```

Algorithm 2: CTLD

$$\begin{aligned} \sigma &= 0.04 \\ C &= \delta'/\tau^2 \\ w &= \frac{20}{\tau^2 L_s K},\ K = 300 \end{aligned}$$

4.3.2 Example

This gradient descent algorithm can be tested on the six hump camel function, used in Kantas et al. (2019):

$$\Phi(x, y) = \left(4 - 2.1x^2 + \frac{x^4}{3}\right)x^2 + xy + (-4 + 4y^2)y^2$$

The global minima are located in (0.0898, 0.7126) and (0.0898, 0.7126). It also has local minima as shown in Figure 4.5.
A good example is provided bellow, where the algorithm visits different local minima before finding the global one (Figure 4.6). It successfully escapes the first local minima (-1.6071, -0.5687), and then the second located in (-1.7036, 0.7961).
This experiment shows promising results, especially because the hyperparameters have not been particularly tuned, so we can expect the algorithm to perform even better. On the other hand, the large number of hyperparameters makes CTLD hard to set up, and to tailor to a particular problem.

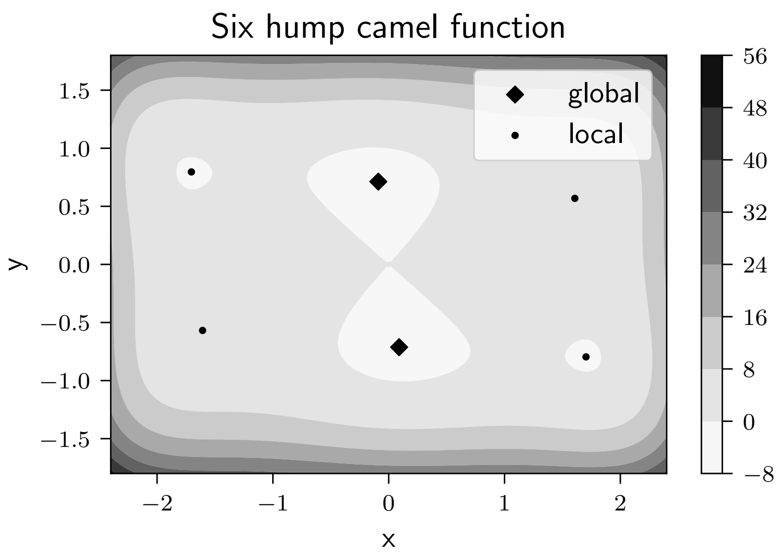

Figure 4.5: The six hump camel function, with its global and local minima.

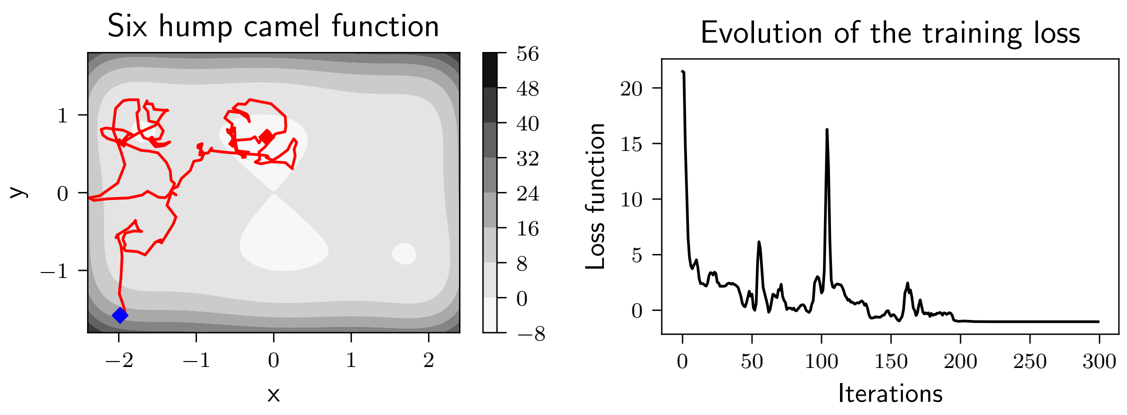

Figure 4.6: Ideally, the algorithm is able to visit several local minima, before finding the global one, and stays in it. Convergence of the algorithm after 300 iterations, with the following parameters: $\tau = 0.05$, $L_s = 200$, $c_m = 0.5$, $\delta' = 1.5$, $\delta = 0.4$, $S = 0.85$ and the defaults parameters given above.

Chapter 5

Implicit methods

The gradient descent techniques mentioned before are all explicit methods. In the following, we explore implicit methods, and try to give an understanding of how they can be valuable options (Toulis et al. (2014), Toulis and Airoldi (2014)). Also, the solution of the sub problem does not have to be exact to already provides good results (Li et al. (2017)).

5.1 Implicit scheme

5.1.1 Definition

The above techniques are said to be 'explicit', which means:

$$\theta_t = \theta_{t-1} - \tau \nabla L(\theta_{t-1})$$

An implicit scheme is when the gradient of the loss function is evaluated with the updated parameters θ_t:

$$\theta_t = \theta_{t-1} - \tau \nabla L(\theta_t)$$

This problem can be written the following way. It is called 'proximal' of the function.

$$\theta_t = \arg\min_{\theta} \left\{ L(\theta) + \frac{1}{2\tau} \|\theta - \theta_{t-1}\|^2 \right\}$$

Since θ_t minimize the above expression, this means the gradient is equal to zero:

$$\begin{aligned} &\nabla L(\theta_t) + \frac{1}{\tau}(\theta - \theta_{t-1}) = 0 \\ \iff &\theta_t = \theta_{t-1} - \tau \nabla L(\theta_t) \end{aligned}$$

So the main point is to solve the min problem defined above.

5.1.2 Example on a convex function

The power of the implicit scheme can be shown by a simple example. Let L be a simple convex function: $L(\theta) = \frac{1}{2}\|\theta\|^2$.
The explicit scheme gives:

$$\begin{aligned} \theta_t &= \theta_{t-1} - \tau \nabla L(\theta_{t-1}) \\ &= \theta_{t-1} - \tau \theta_{t-1} \\ &= (1-\tau)\theta_{t-1} \end{aligned}$$

Which means:

$$\theta_t = (1-\tau)^t \theta_0$$

Assuming $\tau > 0$, there are different regimes, depending on the value of τ:

- $0 < \tau < 1 \iff 0 < 1-\tau < 1$: converges to 0,
- $\tau = 1 \iff 1-\tau = 0$: gives 0,
- $1 < \tau < 2 \iff -1 < 1-\tau < 1$: converges to 0 with oscillations,
- $\tau = 2 \iff 1-\tau = -1$: oscillates between θ_0 and $-\theta_0$
- $\tau > 2 \iff 1-\tau < -1$: diverges.

The implicit scheme gives:

$$\begin{aligned} & \theta_t = \theta_{t-1} - \tau \nabla L(\theta_{t-1}) \\ \iff & \theta_t = \theta_{t-1} - \tau \theta_t \\ \iff & \theta_t(1+\tau) = \theta_{t-1} \\ \iff & \theta_t = \frac{\theta_{t-1}}{1+\tau} \\ \iff & \theta_t = \frac{\theta_0}{(1+\tau)^t} \end{aligned}$$

Still assuming $\tau > 0$, there is only one regime:

- $0 < \tau \iff 0 < \frac{1}{1+\tau} < 1$: converges to 0.

This simple example gives us a first intuition on the robustness of the algorithm depending on the value of τ (see Figure 5.1). We can also notice that, for $\tau \ll 1$, the two methods are roughly equivalent as $\theta_t \approx (1-\tau)\theta_{t-1}$.

5.1.3 Example on a non-convex function

If we consider again the previous example of the non-convex function defined by:

$$f(x) = (x-0.5)^3 + 0.5\sin(16x+4) + (2x)^4 - (3.5x+0.2)^2$$

For a sufficiently small value of τ, the additional term $\frac{1}{2\tau}\|\theta_t - \theta\|^2$ tends to make the function convex (Figure 5.2).

5.1.4 Ill-conditioned problems

Let consider the following problem:

$$L(X) = \frac{1}{2}\|AX - b\|^2$$

To easily be able to choose a particular condition number for the problem, the matrix A is built from its Singular Value Decomposition (SVD):

$$A = U\Sigma V^T$$

Where U and V are orthogonal matrices, and Σ has its diagonal composed of the singular values, such that:

$$\begin{aligned}
&U \in \mathbb{R}^{m\times m}\\
&V \in \mathbb{R}^{n\times n}\\
&\Sigma = \text{diag}(\sigma_1, ..., \sigma_p) \in \mathbb{R}^{m\times n}, \text{ with } p = \min(m, n)
\end{aligned}$$

Assuming σ_1 is the smallest singular value, and σ_p the largest, the condition number is defines as:

$$\kappa = \frac{\sigma_p}{\sigma_1}$$

In our case, the eigenvalues of A^TA correspond to the squares of the singular values:

$$\text{eig}(A^TA) = \{\sigma_1^2, ..., \sigma_p^2\}$$

For the explicit scheme, the update of X is done this way:

$$\begin{aligned}
X_t &= X_{t-1} - \tau A^T(AX_{t-1} - b)\\
\iff X_t &= (I - \tau A^TA)X_{t-1} + \tau A^Tb
\end{aligned}$$

With the closed-form solution for the optimiser X^*:

$$X^* = (A^TA)^{-1}A^Tb$$

The implicit scheme gives the following:

$$\begin{aligned}
&X_t = \arg\min_X \left\{\frac{1}{2}\|AX - b\|^2 + \frac{1}{2\tau}\|X - X_{t-1}\|^2\right\}\\
\iff &\frac{\partial}{\partial X}\left(\frac{1}{2}\|AX - b\|^2 + \frac{1}{2\tau}\|X - X_{t-1}\|^2\right)(X_t) = 0\\
\iff &A^T(AX_t - b) + \frac{1}{\tau}(X_t - X_{t-1}) = 0\\
\iff &X_t = (\tau A^TA + I)^{-1}(\tau A^Tb + X_{t-1})
\end{aligned}$$

Numerically, the results are the following (Figure 5.3) with the same random initialisation.
The implicit scheme finds the right solution, with a final loss function of 0. For the explicit scheme, the loss function does not go below 5e+3. This is because the condition number is very large ($\kappa = 100$), and the updates are equally large for components associated to small and large singular values.
In this case, it is possible to have a closed-form solution for X_t. Usually, it is not possible (the condition number is $\approx 10^7$ for CIFAR 10 according to Frerix et al. (2017)), and then X_t has to be approximated: conjugate gradient, gradient descent. Both methods require several inner iterations to have an evaluation of X_t.
The implicit scheme enable τ to take a large range of values. This is not the case for the explicit scheme, where there are constraints on the value to ensure a proper convergence. In the example above, the learning rate is set to be $\tau = \frac{2}{\sigma_1^2+\sigma_p^2}$ (Nesterov (2018)).
This section on the implicit scheme is the key step to understand how the proximal backpropagation is done in the next section.

5.2 Proximal backpropagation

5.2.1 Definition

In this section, we suggest the use of the proximal backpropagation technique (Frerix et al. (2017) (variations in Fagan and Iyengar (2018)). The general idea of proximal backpropagation is to replace the explicit gradients usually computed for backpropagation by taking implicit steps to update the network. The proximal mapping of a function f if defined as:

$$\text{prox}_{\tau f}(y) = \arg\min_{x} \left\{ f(x) + \frac{1}{2\tau}\|x-y\|^2 \right\}$$

By definition of a minimizer:

$$\begin{aligned} & f(x_t) + \frac{1}{2\tau}\|x_t - x_{t-1}\|^2 \leq f(x_{t-1}) + \frac{1}{2\tau}\|x_{t-1} - x_{t-1}\|^2 \\ \iff & f(x_t) + \frac{1}{2\tau}\|x_t - x_{t-1}\|^2 \leq f(x_{t-1}) \\ \iff & f(x_t) \leq f(x_{t-1}) \end{aligned}$$

This means, the sequence of $f(x_t)$ decreases for any $\tau > 0$.
The main idea behind proximal backpropagation is to solve all the linear sub problems by using a implicit scheme. Let us first define a neural network (Figure 5.4).
The tensors z_i and a_i are set to be the intermediate results of a forward pass through the neural network. The tensor z_i comes after the linear transformation ϕ (multiplication by the weights of the layer), whereas a_i are outputs of the activation function σ. This means:

$$z_i = \phi(\theta_i, a_{i-1}) = a_{i-1}\theta_i$$

$$a_i = \sigma(z_i)$$

The last layer update is done explicitly:

$$\begin{aligned} a_{L-2}^{k+1} &= a_{L-2}^{k} - \tau \nabla_{a_{L-2}} L(\phi(\theta_{L-1}^{k}, a_{L-2}^{k})) \\ \theta_{L-1}^{k+1} &= \theta_{L-1}^{k} - \tau \nabla_{\theta_{L-1}} L(\phi(\theta_{L-1}^{k}, a_{L-2}^{k})) \end{aligned}$$

For all the other layers, the update is done this way:

$$\begin{aligned} z_l^{k+1} &= z_l^k - \sigma'(z_l^k)(\sigma(z_l^k) - a_l^{k+1}) \\ a_{l-1}^{k+1} &= a_{l-1}^{k} - \nabla \left(\frac{1}{2} \|\phi(\theta_l, .) - z_l^{k+1}\|^2 \right) (a_{l-1}^k) \end{aligned}$$

Then, the weights of the network are updated according to:

$$\theta_l^{k+1} = \arg\min_{\theta} \left\{ \frac{1}{2} \left\| \phi(\theta, a_{l-1}^k) - z_l^{k+1} \right\|^2 + \frac{1}{2\tau} \|\theta - \theta_l^k\|^2 \right\}$$

This update, which actually has a closed-from solution, is done for every intermediate layers. Elements of proof are given in Frerix et al. (2017).
The main idea here is to solve individual linear problems, as shown in the previous section. All the sub problems are actually of the form:

$$L(X) = \frac{1}{2} \|AX - b\|^2$$

With:

$$\begin{aligned} X &= \theta \\ A &= a_{l-1}^k \\ b &= z_l^{k+1} \end{aligned}$$

The closed-form solution for the corresponding implicit equation has been calculated in the previous section and is given by:

$$X_t = (\tau A^T A + I)^{-1} (\tau A^T b + X_{t-1})$$

These sub problems are solved according to an implicit scheme, as its efficiency, especially on ill-conditioned problems, has been demonstrated. For the activation function 'layer' though, the usual explicit scheme is done, as by construction, it cannot be transform to a linear problem.
Even if the closed-form solution is known for these sub problems, it may be faster to approximate it by taking a certain number of steps with the conjugate gradient solver, or enough steps so that the gradient reaches a value below a certain threshold.

5.2.2 Example

An easy example is used to demonstrate the capacity of this technique. The idea is to generate a 2D dataset. It is built from the following function, on which noise is added:

$$
\begin{aligned}
&f(x) = (x-2)\sin(2x) \\
&y = f + \epsilon
\end{aligned}
$$

Where $\epsilon \sim \mathcal{N}(0,1)$. It leads to the dataset shown in Figure 5.5. The prediction is done with a small neural network, with two hidden layers of 10 neurons each (Figure 5.5).
As expected, the grid search shows better results for the exact proximal backpropagation (Figure 5.6). When only one iteration is done within the conjugate gradient algorithm, the convergence is slower. As for the influence of the hyperparameter τ, large values (100, 1000) seems to provide good results.

5.2.3 Application to the FBSNN

This section explains how to use the proximal backpropagation in the case of the forward-backward stochastic neural network, as it differs from more conventional neural networks. As a reminder, at each epoch, a new batch is generated, and the loss is computed with N evaluations through the neural network, corresponding to the number of time steps.
The challenging part is to construct the different tensors $z_{i,j}$ and $a_{i,j}$ (at layer i and time j), which correspond to the intermediate results of the neural network. In our case, at each iteration, N predictions are made, so a_i and z_i have to contain all the evaluations through time. One idea could be to take the average, to avoid storing all these intermediate results and manipulate simpler objects. Unfortunately, this straightforward implementation does not provide any good result.
The following Figure 5.7 shows the different variables that have to be stored.
Once the different values are stored, tensors containing all the values through time of a particular intermediate results are built. So we end up with tensors z_i containing all the $z_{i,j}$, and a_i containing all the value $a_{i,j}$. As a reminder, the update of a_{L-2} is done this way:

$$a_{L-2}^{k+1} = a_{L-2}^{k} - \tau \nabla_{a_{L-2}} L(\phi(\theta_{L-1}^{k}, a_{L-2}^{k}))$$

This requires to compute the gradient of the loss function with respect to all the $a_{L-2,j}$. In term of implementation, this is done using automatic differentiation. The different gradients are then stacked together. The same process is used to compute the gradient of the loss function with respect to the weights of the last layer θ_{L-1}, since the update is done the following way:

$$\theta_{L-1}^{k+1} = \theta_{L-1}^{k} - \tau \nabla_{\theta_{L-1}} L(\phi(\theta_{L-1}^{k}, a_{L-2}^{k}))$$

Then, the regular update scheme is done through the layers for z_l and a_{l-1} which enable the update of θ_l by the implicit scheme, until each θ is updated.

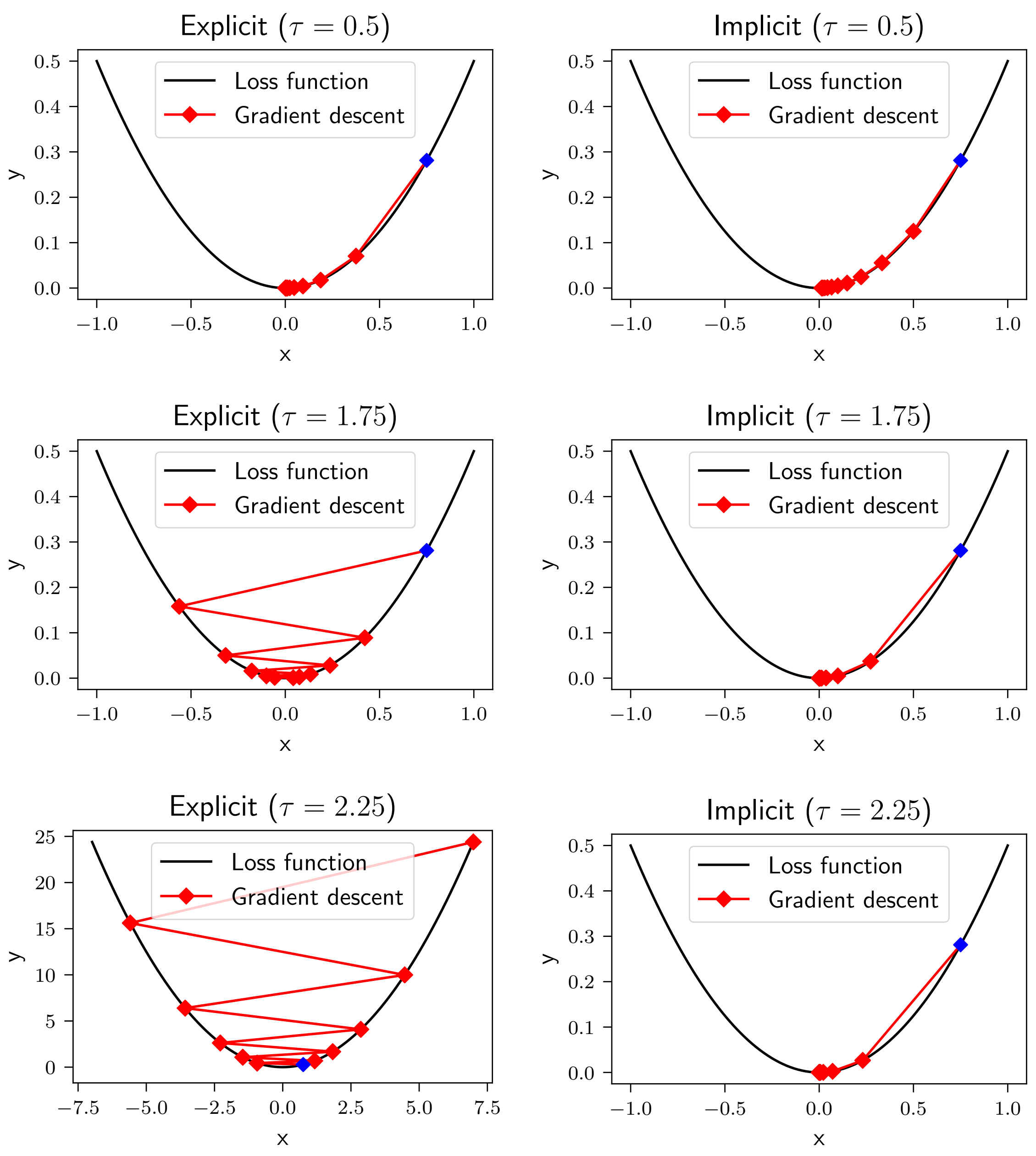

Figure 5.1: Comparison between explicit and implicit scheme for different learning rate: $\tau = 0.5$, $\tau = 1.75$ and $\tau = 2.25$ on the loss function $L(x) = \frac{1}{2}x^2$.

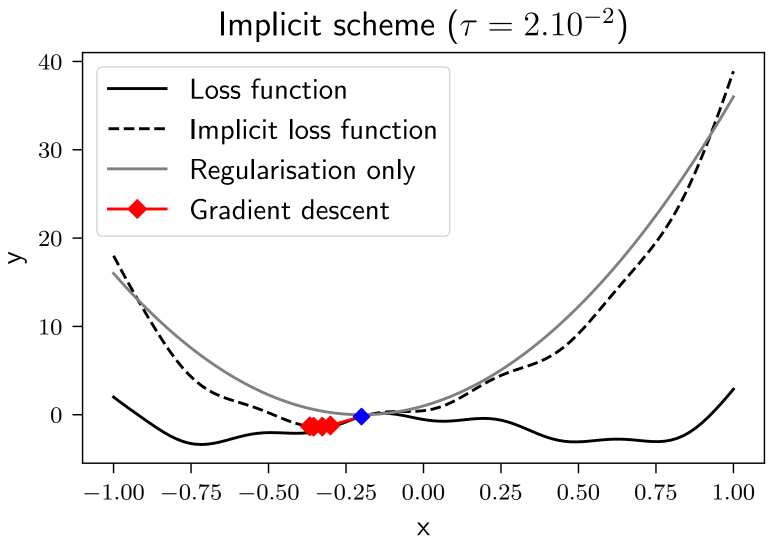

Figure 5.2: Implicit scheme on a non convex function

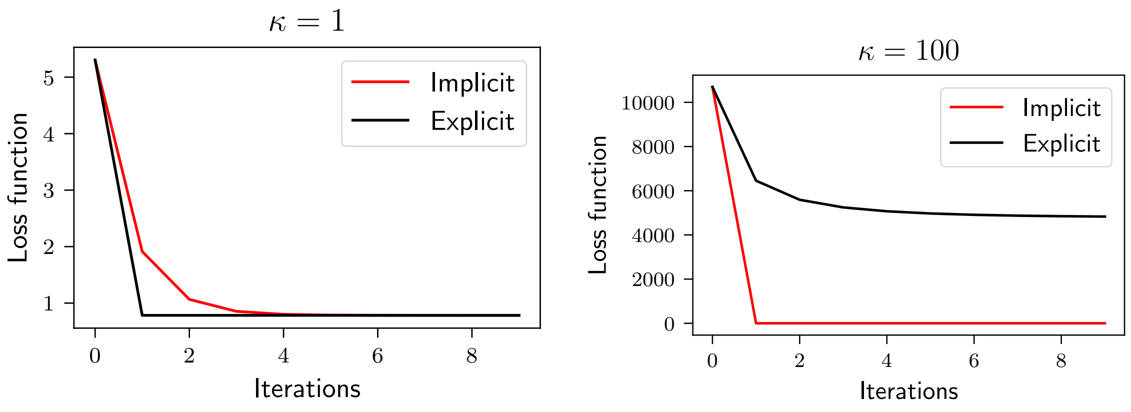

Figure 5.3: Comparison in between explicit and implicit scheme for $\frac{1}{2}\|AX - b\|^2$ with $m = 30$, $n = 20$, for $\kappa = 1$ and $\kappa = 100$.

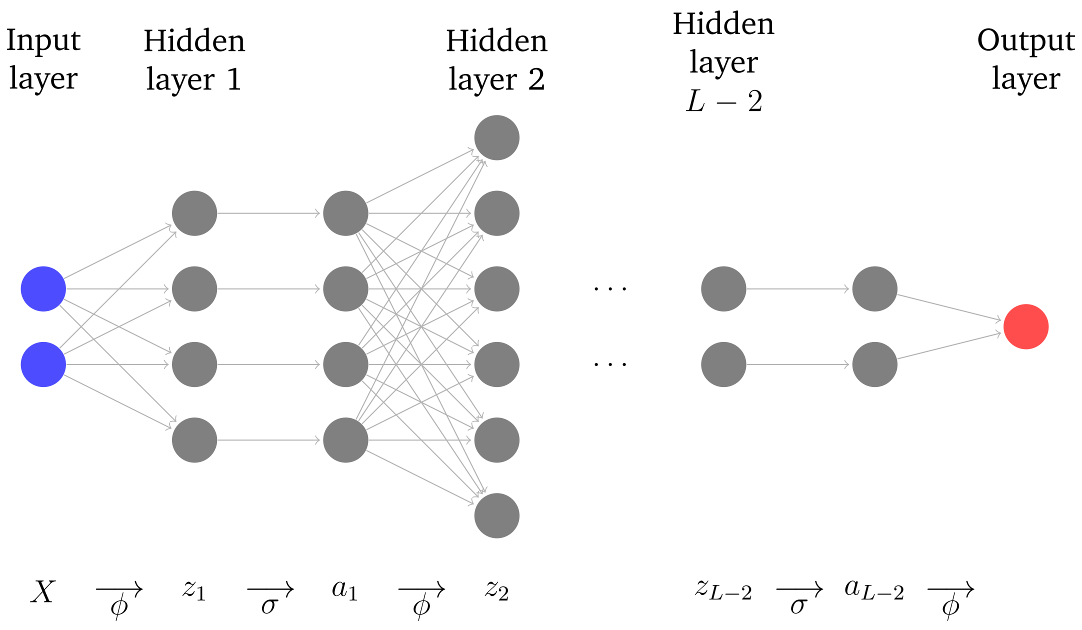
Input layer
Hidden layer 1
Hidden layer 2
Hidden layer $L-2$
Output layer
$X \xrightarrow[\phi]{} z_1 \xrightarrow[\sigma]{} a_1 \xrightarrow[\phi]{} z_2$
$z_{L-2} \xrightarrow[\sigma]{} a_{L-2} \xrightarrow[\phi]{}$

Figure 5.4: Architecture of the neural network

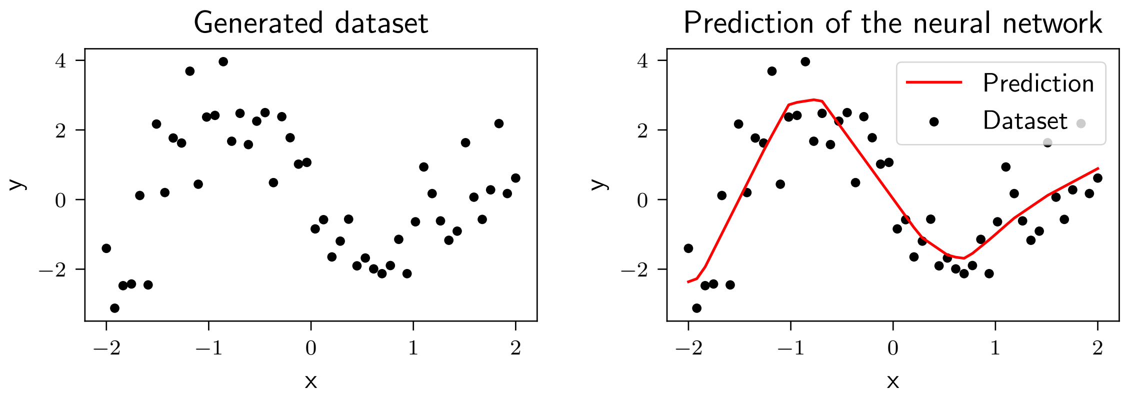

Figure 5.5: Generated dataset and predicted function.

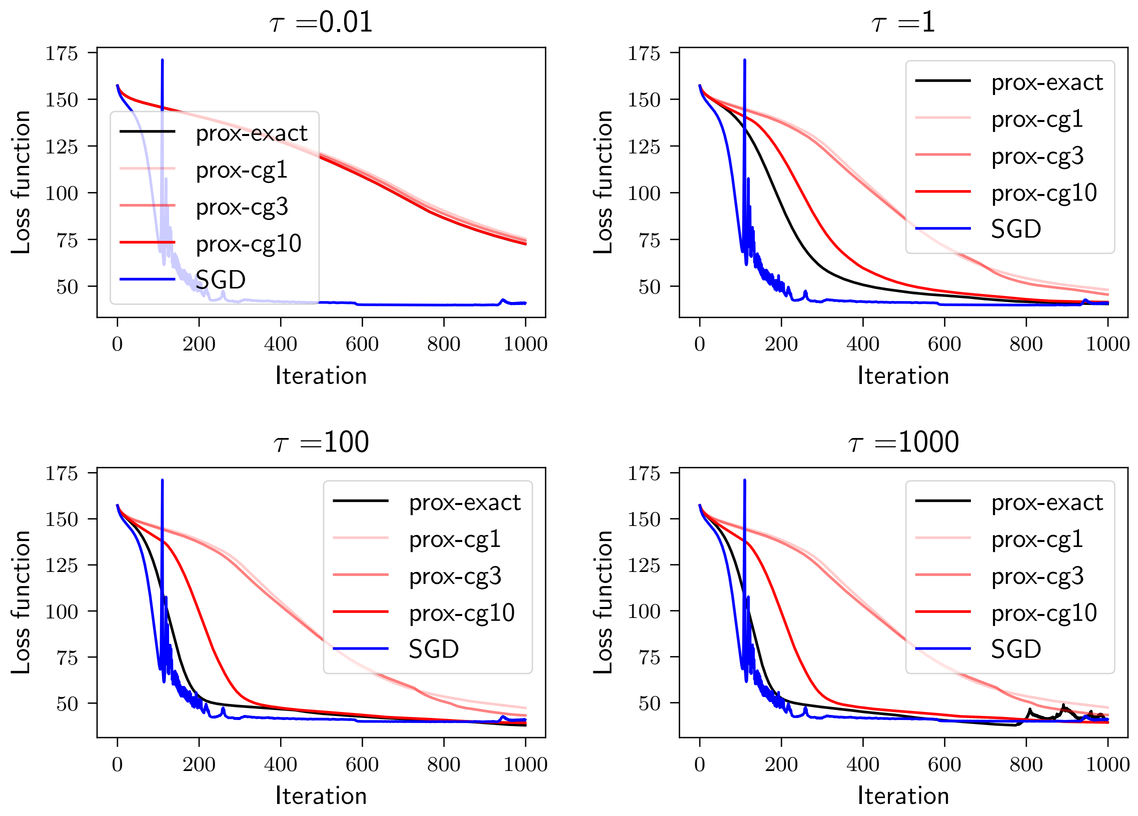

Figure 5.6: Comparison within different proximal backpropagation for learning rates: $\tau = 0.01$, $\tau = 1$, $\tau = 100$ and $\tau = 1000$.

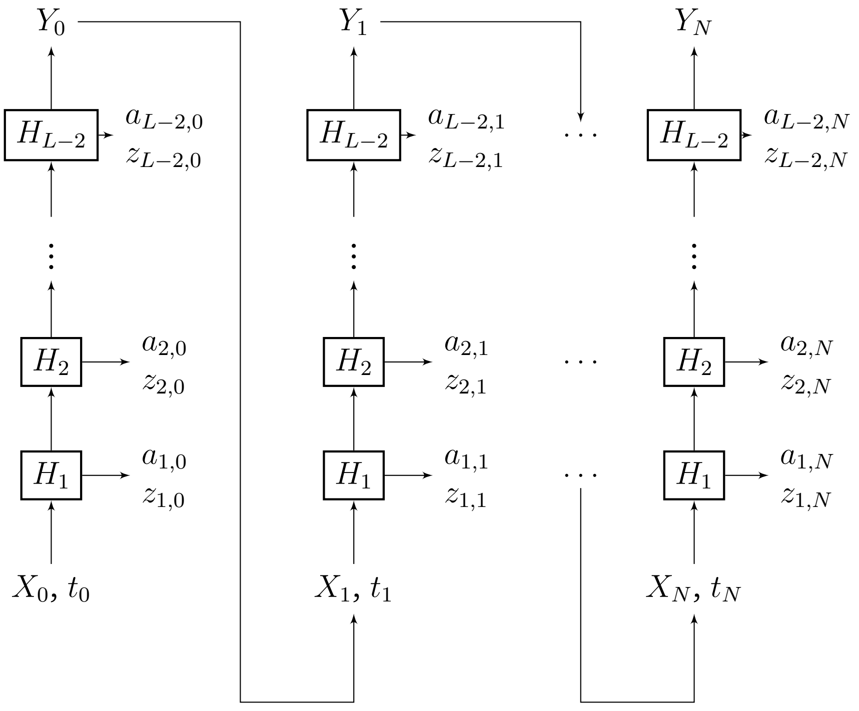

Figure 5.7: Architecture of the forward-backward stochastic neural network, with values of $z_{i,j}$ and $a_{i,j}$ stored to compute proximal backpropagation.

Chapter 6

Simulations

In this chapter, the different methods seen in the previous chapter are applied to the main problem. To save time, a lighter problem is defined. The number of trajectories is limited to $M = 10$, the number of time steps is $N = 50$ and the dimension is $D = 10$. The neural network is smaller as well: only 2 hidden layers of 50 neurons each. The number of weights to optimise is 3201 (instead of 223745 for the original problem). The weights initialization is done with the same random seed to ensure a good comparison between the different algorithms. For time reasons, only 1000 iterations are done, and so the initialization may affect the performance of the algorithms.

6.1 Stochastic Gradient Descent

This vanilla version of the SGD enables us to have a base model for comparison. The Figure 6.1 shows the results after 1000 iterations.
Quantitatively speaking, the mean, standard deviation and minimum of the loss function over the last 100 iterations are indicated in the Table 6.1. They serve as quantitative metrics on top of the qualitative visualisation of the paths. Also, the error computed at the last iteration is provided. The relative error is defined as:

$$\sqrt{\frac{(Y_{test} - Y_{pred})^2}{Y_{test}^2}}$$

The 'error mean' which is represented in the figures corresponds to the means of the error over the different paths. The term 'std' represents the standard deviation of the mean over the different paths.

Table 6.1: Loss function on the last 100 iterations, and error at the last iteration.

	loss (mean)	loss (std)	loss (min)	error
SGD	81.1	86.5	18.3	0.075 ($\pm$ 0.035)

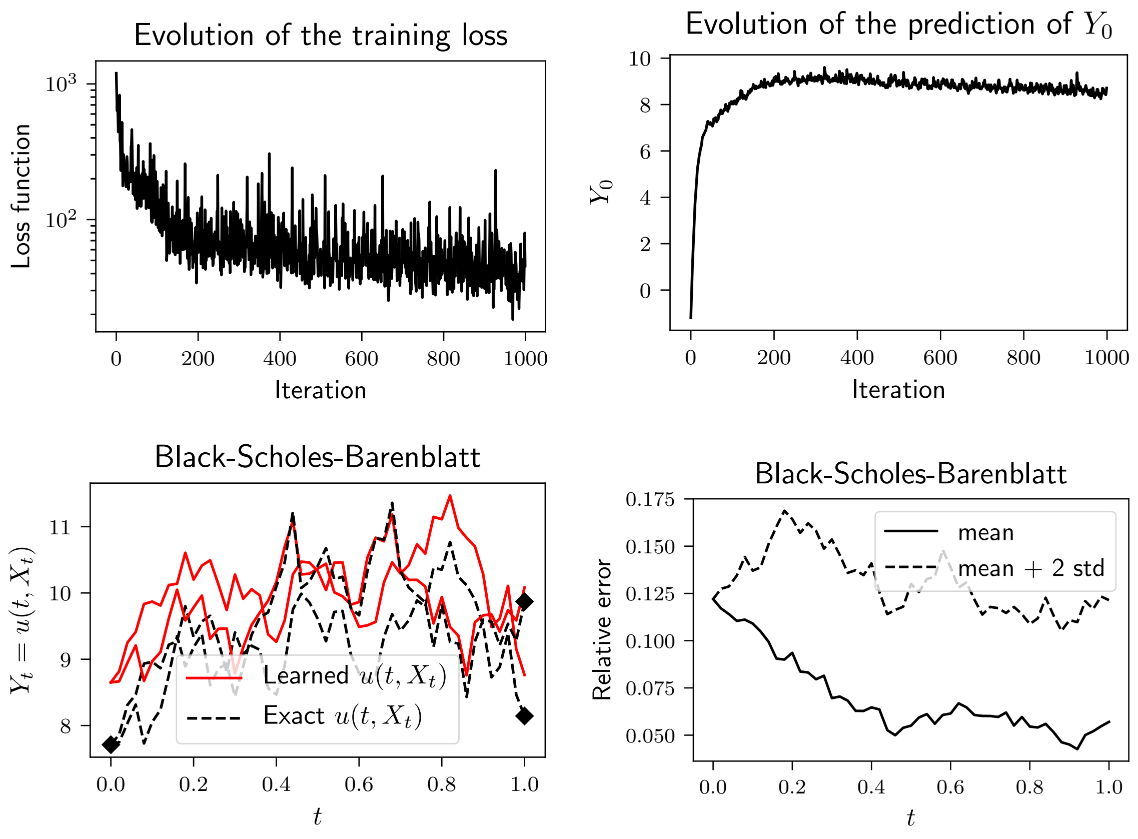

Figure 6.1: Training loss over 1000 iterations with the following parameters: $M = 10$, $N = 50$, $D = 10$, 2 hidden layers of 50 neurons, SGD with learning rate 1e-4, $\sin$ activation function.

6.2 Stochastic Gradient Lanevin Dynamics

The idea here is to evaluate how the algorithm performs, depending on the value of β. As a reminder, this hyperparameter is representative of the trade off between exploration and convergence. Two values are tested: $\beta = 10$ and $\beta = 1000$, with the same learning rate as before 1e-4.
Visually, the paths look more convincing with the larger value of β (Figure 6.2), which is also confirm by the metrics from (Table 6.2).

Table 6.2: Loss function on the last 100 iterations, and error at the last iteration.

	loss (mean)	loss (std)	loss (min)	error
SGD	81.1	86.5	18.3	0.075 ($\pm$ 0.035)
SGLD ($\beta = 10$)	84.4	87.1	18.4	0.099 ($\pm$ 0.049)
SGLD ($\beta = 1000$)	81.3	86.5	18.4	0.077 ($\pm$ 0.036)

It seems that SGLD with a large value of β, which means a small impact of the standard Gaussian vector, provides similar results to SGD in this case. For a small value of β the exploration might take the lead over the convergence which could explain that the results are not as good as for a large value. Again, in SGLD this hyperparameter is fixed through time, and does not allow to have an exploration phase followed by a convergence phase like CTLD does.

6.3 Continuous Tempering Langevin Dynamics

This algorithm provides an exploration phase, and a convergence phase. The challenge here is to correctly adjust the level of exploration. This first step enables the algorithm to visit different places, and potentially local minima. For the convergence step, the algorithm can be written this way:

$$\begin{aligned} & \theta_t = \theta_{t-1} + \tau r_{t-1} \\ \iff & \theta_t = \theta_{t-1} + \tau((1-\tau\gamma)r_{t-2} - \tau\nabla L(\theta_{t-1})) \\ \iff & \theta_t = \theta_{t-1} - \tau^2\nabla L(\theta_{t-1}) + \tau(1-\tau\gamma)r_{t-2} \\ \iff & \theta_t - \theta_{t-1} = -\tau^2\nabla L(\theta_{t-1}) + (1-\tau\gamma)(\theta_{t-1} - \theta_{t-2}) \end{aligned}$$

Which means this corresponds to SGD with a learning rate of τ^2 and a momentum of $1 - \tau\gamma$. To stay consistent with the previous experiments, τ is chosen to be equal to 1e-2 so the learning rate is still 1e-4. Conducting the same experiment with CTLD leads to the following results (Figure 6.3 and Table 6.3).

Table 6.3: Loss function on the last 100 iterations, and error at the last iteration.

	loss (mean)	**loss (std)**	**loss (min)**	**error**
SGD	81.1	86.5	18.3	0.075 (± 0.035)
CTLD ($L_s = 200$)	58.0	100.3	1.7	0.070 (± 0.070)

This algorithm has a lot of hyperparameters to tune, and as a first approach, recommended values are preferred for most of them.
Even with default values for the hyperparameters, this algorithm provides good results, and significantly better than the ones obtained with a vanilla Stochastic Gradient Descent. This is promising in a sens that we could expect improvement by correctly tuning the hyperparameters.

6.4 Implicit scheme

The implicit scheme is tested with two values of τ (0.01 and 100), for 2 and 10 inner iterations. The visual results are shown in Figure 6.4, whereas the quantitative ones are available in Table 6.4.
As expected, increasing the number of inner iterations gives better results. However, this conclusion has to be nuanced since it also increases the computation time. We have to keep in mind that the goal here is not to transfer the main optimisation problem to the inner problem.
However, even if we still optimize the loss function using a gradient descent algorithm, this one acts on a more convex function, so potentially with better properties than the original one.

Table 6.4: Loss function on the last 100 iterations, and error at the last iteration for explicit scheme and implicit one (I).

	loss (mean)	**loss (std)**	**loss (min)**	**error**
Explicit SGD	81.1	86.5	18.3	0.075 (± 0.035)
I ($\tau = 0.01$, $n = 2$)	54.2	64.6	8.4	0.024 (± 0.013)
I ($\tau = 0.01$, $n = 10$)	13.2	33.4	0.2	0.015 (± 0.021)
I ($\tau = 100$, $n = 2$)	54.2	65.1	7.3	0.024 (± 0.012)
I ($\tau = 100$, $n = 10$)	12.9	29.3	0.2	0.017 (± 0.021)

6.5 Proximal backpropagation

This algorithm relies on two hyperparameters. The first one is the learning rate τ to directly update a_{L-2} and θ_{L-1} which are respectively the output of the penultimate layer and the weights of the last layer. This update of a_{L-2} is then used to compute the update of z_{L-2}, which is used to update a_{L-3}, and then θ_{L-2}, etc. The update of the weights θ_l is done using and implicit scheme, where the second hyperparameter τ_θ is used.
The Figure 6.5 shows the results for $\tau = 10^{-2}$, and $\tau_\theta = 0.01$. The results are the following (Table 6.5).

Table 6.5: Loss function on the last 100 iterations, and error at the last iteration for explicit SGD scheme and proximal backpropagation.

	loss (mean)	**loss (std)**	**loss (min)**	**error**
Explicit SGD	81.1	86.5	18.3	0.075 (± 0.035)
P ($\tau = 10^{-2}$, $\tau_\theta = 0.01$)	65.9	80.5	13.2	0.029 (± 0.024)
P ($\tau = 10^{-2}$, $\tau_\theta = 1$)	60.3	74.8	6.7	0.049 (± 0.044)
P ($\tau = 10^{-2}$, $\tau_\theta = 100$)	77.9	125.8	8.1	0.142 (± 0.076)
P ($\tau = 10^{-3}$, $\tau_\theta = 0.01$)	112.9	70.8	36.3	0.203 (± 0.102)
P ($\tau = 10^{-3}$, $\tau_\theta = 1$)	95.4	69.7	32.5	0.086 (± 0.055)
P ($\tau = 10^{-3}$, $\tau_\theta = 100$)	78.2	68.8	21.0	0.047 (± 0.037)

These results demonstrate the large capacities of this technique. In particular, the case with $\tau = 10^{-2}$ and $\tau_\theta = 0.01$ leads to a relative error of 0.029 on the last iteration, which is less than half the value of the one obtained by an explicit SGD. It is also possible to use a larger learning rate without suffering from a diverging loss function. Also, the computation time per iteration is comparable to the one from the other algorithm we have seen so far. This makes this technique particularly interesting to solve this problem. Finally, large values of τ_θ speed up a bit the convergence, but we advise not using too high values as it tends to introduce noisy predictions.

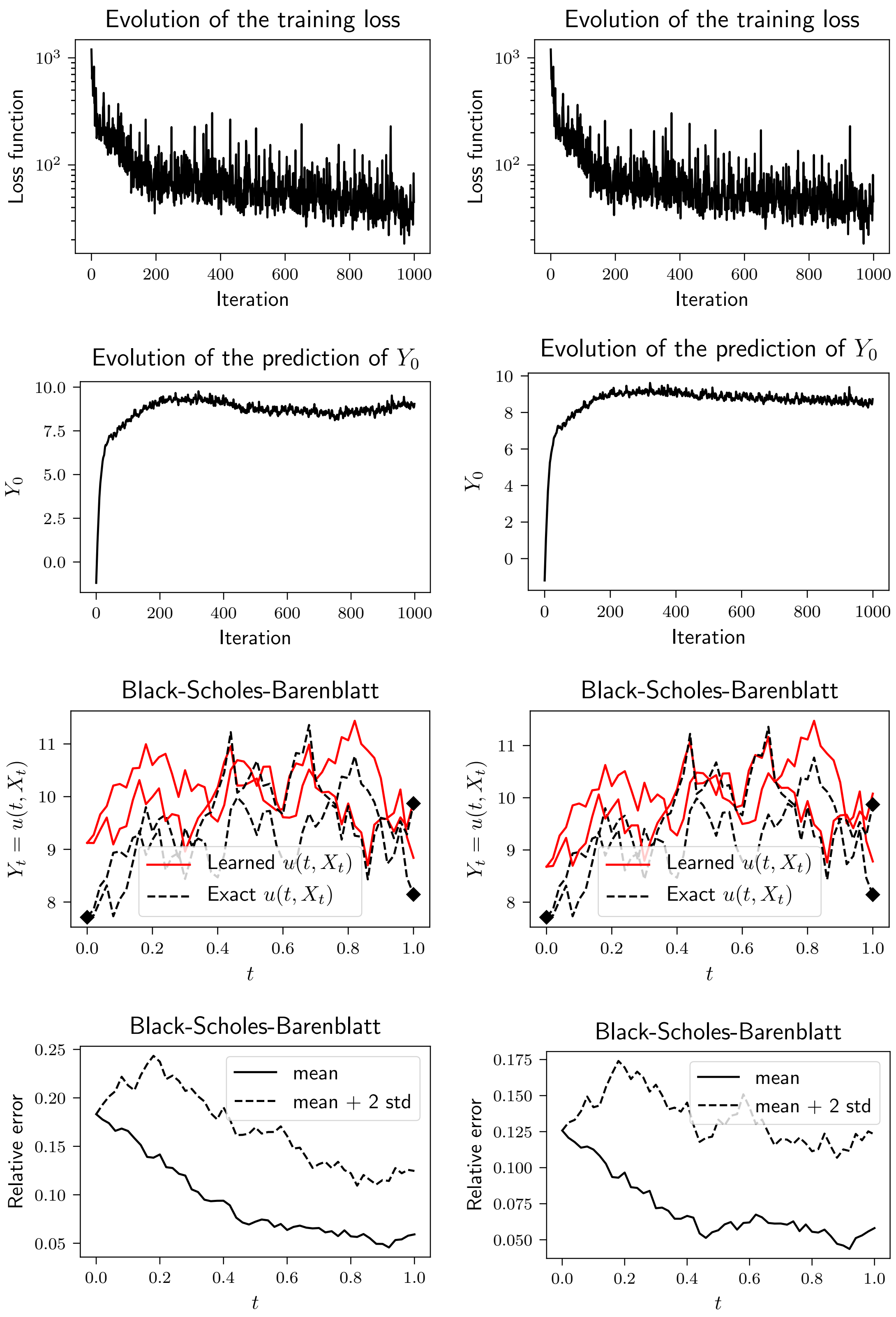

Figure 6.2: Comparison in between $\beta = 10$ (left) and $\beta = 1000$ (right) implementation for training loss, Y_0 prediction, learned solutions and error after 1000 iterations.

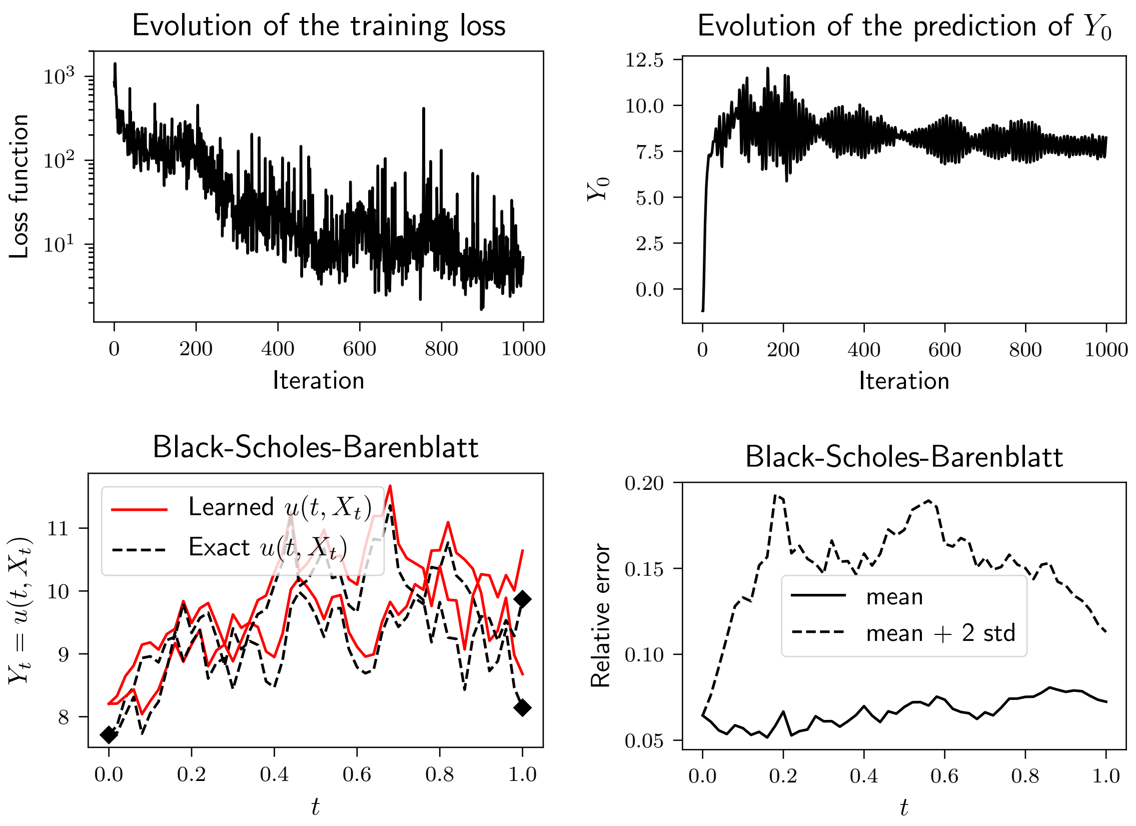

Figure 6.3: Training loss over 1000 iterations with the following parameters: $M = 10$, $N = 50$, $D = 10$, 2 hidden layers of 50 neurons, CTLD with an exploration phase of 200 iterations.

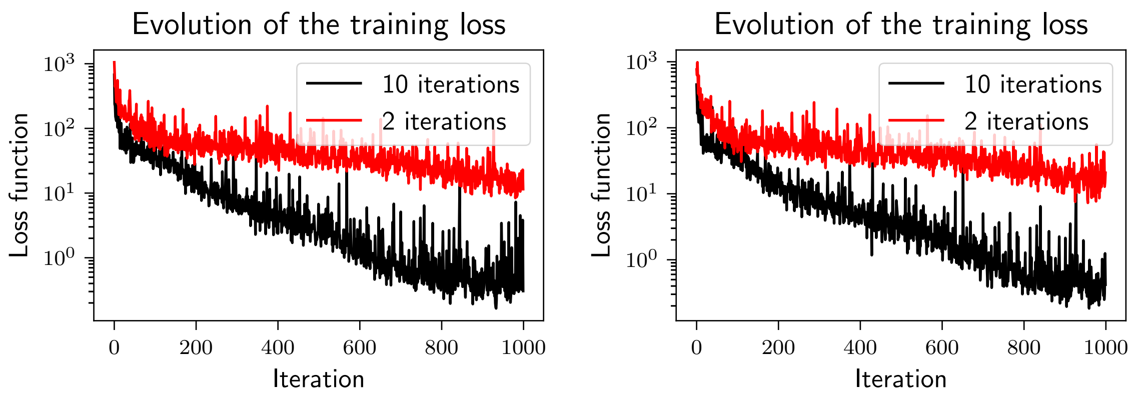

Figure 6.4: Training loss over 1000 iterations with the following parameters: $M = 10$, $N = 50$, $D = 10$, 2 hidden layers of 50 neurons, implicit with $\tau = 0.01$ (left) and $\tau = 100$ (right).

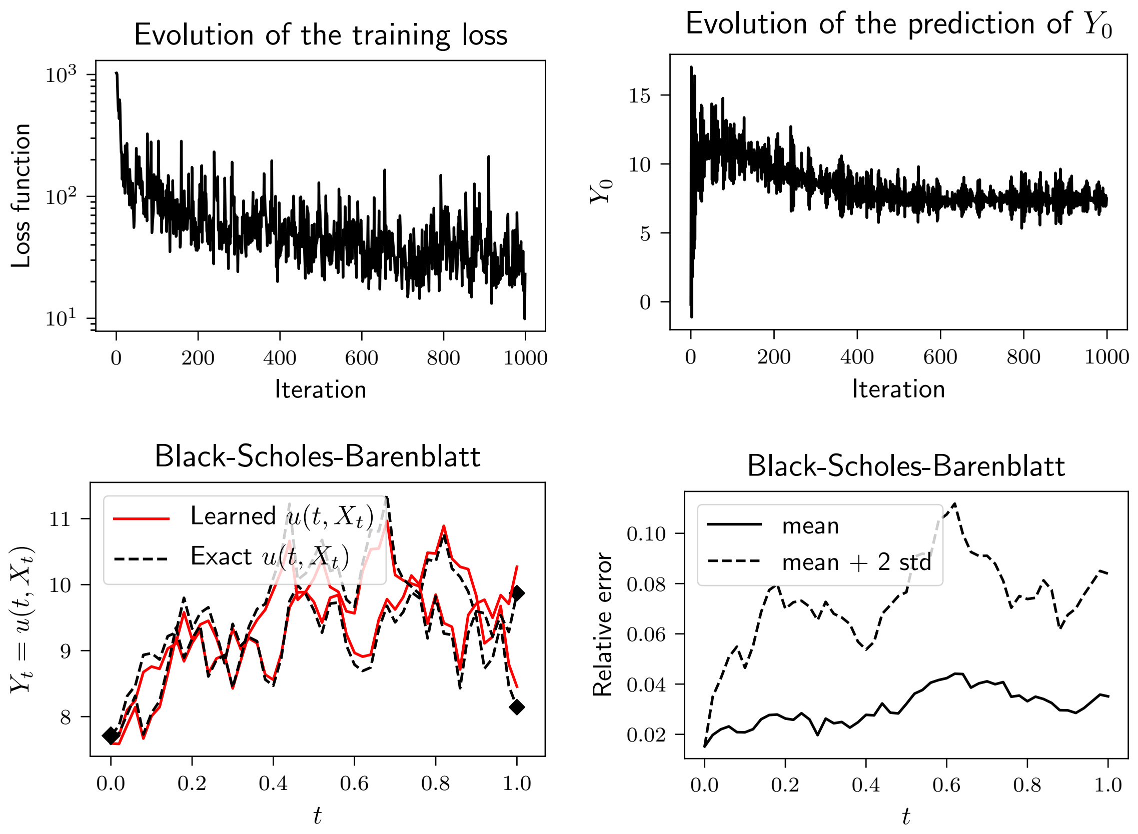

Figure 6.5: Training loss over 1000 iterations with the following parameters: $M = 10$, $N = 50$, $D = 10$, 2 hidden layers of 50 neurons, proxy with learning rate 1e-2, $\tau = 0.01$, $\sin$ activation function.

Chapter 7

Conclusion

7.1 Achievements

The first main goal of the project was to have a working implementation of the neural network applied to the Black-Scholes equation. This PyTorch implementation had to be tested by running training sessions to ensure it provided correct results. A couple of experiments were first conducted by changing the activation function and noticing the impact on the loss function, and so on the overall convergence. Also, a time analysis has been done to understand where time could be saved in the learning process.

This project focuses on optimisation techniques to solve, in the best possible way, the Black-Scholes equation. To this purpose, several gradient descent techniques have been explored and toy examples were built to conduct tests and evaluate the performance of such techniques. Starting from simple Stochastic Gradient Descent (SGD), the project then moved onto exploring more complex algorithms. Among them, an evolved version of SGD with a Langevin Dynamics terms and Continuous Tempering Langevin Dynamics based on exploration and convergence phases. This has required the implementation of different optimisers. The main reason all these techniques were tested, was to understand how the optimisers explore the parameters space.

Another idea was to analyse techniques that are supposed to provide better results on ill-conditioned problems, which is how problems usually are in reality. A raw implicit scheme has been tested, which produced very good results. A more sophisticated technique 'proximal backpropagation' was then also tested. This obtained very promising results on a toy neural network, encouraging us to test it on our main problem. We hence developed an adapted version of this algorithm to fit the forward-backward stochastic neural network. The results substantially outperformed the other algorithms tested in this project, as the algorithm manages to provide a very close solution with a limited number of iterations.

7.2 Future work

During this project, a lot of choices have been made, and starting from a wide problem, we narrowed it down to a few very specific subproblems. At this point, future work could focus either on going forward in the current direction or on taking a step back to explore more options.
As mentioned, the proximal backpropagation applied to the forward-backward stochastic neural network is the most promising technique seen during the project. The current implementation provides already great results. However, there is still room for improvement. It could be improved, for example, by making the implementation more efficient from a software engineer point of view or getting closer to the hardware to try to take full advantage of more powerful GPU's. In terms of algorithm, the proximal backpropagation could be even more tailored for the FBSNN. For now, the output of the penultimate layer and the weights of the last layer are updated with a typical explicit scheme. The other weights are updated through an implicit scheme, using an exact solution. This step could almost certainly be improved by looking for more efficient techniques to solve the implicit problem itself. This could be for instance mixing a state-of-the-art optimiser with the proxy scheme.
Regarding the architecture of the neural network itself, there are many ways in which it could be improved. The one used in this project is a unique feed-forward neural network. But we can think of more evolved architecture or even a single neural network per time step.
From a wider point of view, the original goal was to provide an efficient way to solve PDEs using deep learning. This project is very theoretical, with the aim of developing a robust tool which could be used in many scenarios. However, we focused on the Black-Scholes equation and a particular terminal condition. Several variations could be tried. For example, changing the function describing the terminal condition. Or we could try this model on call options, and also try our model on real data, and perform back-testing. There are also plenty of other PDEs that could be solved, hence bringing the potential impact of this project to various different industries and applications.

Bibliography

Beck, C., Weinan, E., and Jentzen, A. (2019). Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations. *Journal of Nonlinear Science*, 29(4):1563–1619. pages 1

Bender, C. and Denk, R. (2007). A forward scheme for backward sdes. *Stochastic processes and their applications*, 117(12):1793–1812. pages 15

Bouchard, B. and Touzi, N. (2004). Discrete-time approximation and monte-carlo simulation of backward stochastic differential equations. *Stochastic Processes and their applications*, 111(2):175–206. pages 8

Brosse, N., Durmus, A., and Moulines, E. (2018). The promises and pitfalls of stochastic gradient langevin dynamics. In *Advances in Neural Information Processing Systems*, pages 8268–8278. pages 27

Fagan, F. and Iyengar, G. (2018). Robust implicit backpropagation. *arXiv preprint arXiv:1808.02433*. pages 36

Frerix, T., Möllenhoff, T., Moeller, M., and Cremers, D. (2017). Proximal backpropagation. *arXiv preprint arXiv:1706.04638*. pages 36, 37

Gobet, E. (2016). *Monte-Carlo methods and stochastic processes: from linear to non-linear*. Chapman and Hall/CRC. pages 8, 9

Han, J. et al. (2016). Deep learning approximation for stochastic control problems. *arXiv preprint arXiv:1611.07422*. pages 1

Han, J., Jentzen, A., and Weinan, E. (2017). Overcoming the curse of dimensionality: Solving high-dimensional partial differential equations using deep learning. *arXiv preprint arXiv:1707.02568,* pages 1–13. pages 1

Han, J., Jentzen, A., and Weinan, E. (2018). Solving high-dimensional partial differential equations using deep learning. *Proceedings of the National Academy of Sciences*, 115(34):8505–8510. pages 1

Henry-Labordere, P. (2017). Deep primal-dual algorithm for bsdes: Applications of machine learning to cva and im. *Available at SSRN 3071506*. pages 1

Higham, D. J. (2004). *An introduction to financial option valuation: mathematics, stochastics and computation*, volume 13. Cambridge University Press. pages 3

Kantas, N., Parpas, P., and Pavliotis, G. A. (2019). The sharp, the flat and the shallow: Can weakly interacting agents learn to escape bad minima? *arXiv preprint arXiv:1905.04121*. pages 27, 31

Li, Q., Chen, L., Tai, C., and Weinan, E. (2017). Maximum principle based algorithms for deep learning. *The Journal of Machine Learning Research*, 18(1):5998–6026. pages 33

Ludvigsson, G. (2013). Kolmogorov equations. pages 6

Nesterov, Y. (2018). *Lectures on convex optimization*, volume 137. Springer. pages 36

Pan, H. and Jiang, H. (2015). Annealed gradient descent for deep learning. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, pages 652–661. AUAI Press. pages 27

Perkowski, N. (2011). Backward stochastic differential equations: an introduction. *Tanári jegyzet*. pages 8

Pham, H. (2015). Feynman-kac representation of fully nonlinear pdes and applications. *Acta Mathematica Vietnamica*, 40(2):255–269. pages 6

Raissi, M. (2018a). Fbsnns. `https://github.com/maziarraissi/FBSNNs`. pages 18

Raissi, M. (2018b). Forward-backward stochastic neural networks: Deep learning of high-dimensional partial differential equations. *arXiv preprint arXiv:1804.07010*. pages 1, 15, 16, 18

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*. pages 26

Sirignano, J. and Spiliopoulos, K. (2018). Dgm: A deep learning algorithm for solving partial differential equations. *Journal of Computational Physics*, 375:1339–1364. pages 1

Toulis, P., Airoldi, E., and Rennie, J. (2014). Statistical analysis of stochastic gradient methods for generalized linear models. In *International Conference on Machine Learning*, pages 667–675. pages 33

Toulis, P. and Airoldi, E. M. (2014). Implicit stochastic gradient descent for principled estimation with large datasets. *ArXiv e-prints*. pages 33

Toulis, P., Tran, D., and Airoldi, E. (2016). Towards stability and optimality in stochastic gradient descent. In *Artificial Intelligence and Statistics*, pages 1290–1298. pages 26

Van Casteren, J. A. (2007). Feynman-kac formulas, backward stochastic differential equations and markov processes. In *Functional Analysis and Evolution Equations*, pages 83–111. Springer. pages 6

Vaswani, S., Mishkin, A., Laradji, I., Schmidt, M., Gidel, G., and Lacoste-Julien, S. (2019). Painless stochastic gradient: Interpolation, line-search, and convergence rates. *arXiv preprint arXiv:1905.09997*. pages 26

Weinan, E., Han, J., and 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. pages 1

Ye, N., Zhu, Z., and Mantiuk, R. K. (2017). Langevin dynamics with continuous tempering for training deep neural networks. *arXiv preprint arXiv:1703.04379*. pages 28, 30