Deep Neural Network Application on Wave Equation

CSE Semester Project
Mathematics

Anya-Aurore Mauron
Sciper: 295924

Supervisor: Prof. Marco Picasso

Spring 2022

Abstract

The 1D wave equation describes the physical phenomena of mechanical waves or electromagnetic waves. We consider this equation with an initial condition which is a linear combination of sinusoidal functions, where the weights depend on some instances of i.i.d random variables following a uniform distribution.

The aim of this study is to approximate the solution of the wave equation at the final time given the instances of the random variables, employing a deep neural network. In order to achieve this task, a multi-layer perceptron is used, coupled with the grid-search method. The performance of the machine learning algorithm is studied with respect to the dimensions of the input and of the output of the model, and is compared to the performance of a finite difference scheme.

In the end, we found that the neural network achieves to approximate the solution well. However, the architecture of the model depends highly on the input/output dimensions.

Contents

1 Introduction

Given $u_0(x) : (0,1) \to \mathbb{R}$, let $u(x,t) : (0,1) \times (0,1) \to \mathbb{R}$ be the solution of the wave equation:

$$\begin{aligned} &\frac{\partial^2 u}{\partial t^2}(x,t) - \frac{\partial^2 u}{\partial x^2}(x,t) = 0 && (x,t) \in (0,1) \times (0,1), \\ &u(0,t) = u(1,t) = 0 && t \in (0,1), \\ &u(x,0) = u_0(x) \quad \text{and} \quad \frac{\partial u}{\partial t}(x,0) = 0 && x \in (0,1). \end{aligned} \tag{1}$$

If u_0 is extended as an odd 2-periodic function, the solution u is given by D'Alembert formula:

$$u(x,t) = \frac{1}{2}\left(u_0(x+t) + u_0(x-t)\right). \tag{2}$$

Denote by $\mathcal{P}$ the space of i.i.d random variables following a uniform distribution on the intervals $[-1,1]^{N_\mu}$. Assume the initial condition is given by

$$\begin{aligned} u_0(x,\boldsymbol{\mu}) : (0,1) \times [-1,1]^{N_\mu} &\to \mathbb{R} \\ (x,\boldsymbol{\mu}) &\mapsto \sum_{j=1}^{N_\mu} \frac{\mu_j}{j^K} \sin(j\pi x) \end{aligned} \tag{3}$$

where $\boldsymbol{\mu} = (\mu_1, ..., \mu_{N_\mu})^\top \in [-1,1]^{N_\mu}$ is an instance of $\mathcal{P}$. The power $K \in \mathbb{N}$ represents the smoothness of the function $u_0(x,\boldsymbol{\mu})$; the bigger it is, the smoother the function (see figure 1). Then, we solve the wave equation numerically. Let $N_h \in \mathbb{N}^*$ be the number of space steps, $h = \frac{1}{N_h+1}$ the space step, $N_\tau \in \mathbb{N}^*$ the number of time steps and $\tau > 0$ the time step. Then let

$$\boldsymbol{u}_{N_h}(\boldsymbol{\mu}) = \left(u_1^{N_\tau}(\boldsymbol{\mu}), ..., u_{N_h}^{N_\tau}(\boldsymbol{\mu})\right)$$

be the finite difference solution, where $u_i^{N_\tau}(\boldsymbol{\mu})$ is an approximation of $u(x_i,\boldsymbol{\mu}) := u(x_i, t_{N_\tau}; \boldsymbol{\mu})$ at $x_i = ih$ and at time $t_{N_\tau} = N_\tau \tau$, for $i = 1, \ldots, N_h$.

The aim of this paper is to approximate the mapping $\boldsymbol{\mu} \to \boldsymbol{u}_{N_h}(\boldsymbol{\mu})$ using a deep neural network (DNN). Given a given DNN with weights and biases $\boldsymbol{\Theta} \in \mathbb{R}^{N_{DNN}}$, let $\boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta})$ be an approximation of $\boldsymbol{u}_{N_h}(\boldsymbol{\mu})$. Let $u_{DNN}(x,\boldsymbol{\mu};\boldsymbol{\Theta})$ be $\mathbb{P}_1$ on each interval $[x_i, x_{i+1}]$ for $i = 0, \ldots, N_h$, such that $u_{DNN}(x_i,\boldsymbol{\mu};\boldsymbol{\Theta}) = \boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta})_i$. On the border, we suppose that $u_{DNN}(0,\boldsymbol{\mu};\boldsymbol{\Theta}) = u_{DNN}(1,\boldsymbol{\mu};\boldsymbol{\Theta}) = 0$. We assess the error between $u(x,\boldsymbol{\mu})$ and $u_{DNN}(x,\boldsymbol{\mu};\boldsymbol{\Theta})$ by computing the multi-dimensional integral

$$\frac{1}{2^{N_\mu}} \int_{[-1,1]^{N_\mu}} \int_0^1 \left(u(x,\boldsymbol{\mu}) - u_{DNN}(x,\boldsymbol{\mu};\boldsymbol{\Theta})\right)^2 \mathrm{d}x \, \mathrm{d}\boldsymbol{\mu}. \tag{4}$$

In order to approximate it numerically, we use the *Monte-Carlo method* to compute the integral with respect to $\boldsymbol{\mu}$, by selecting N_M replicas $\{\boldsymbol{\mu}^1, \ldots, \boldsymbol{\mu}^{N_M}\}$, instances of $\mathcal{P}$. Then, we discretize with respect to x, by taking N_h equidistant points x_i, as mentioned above. Hence we can approximate the difference between the 2 solutions by:

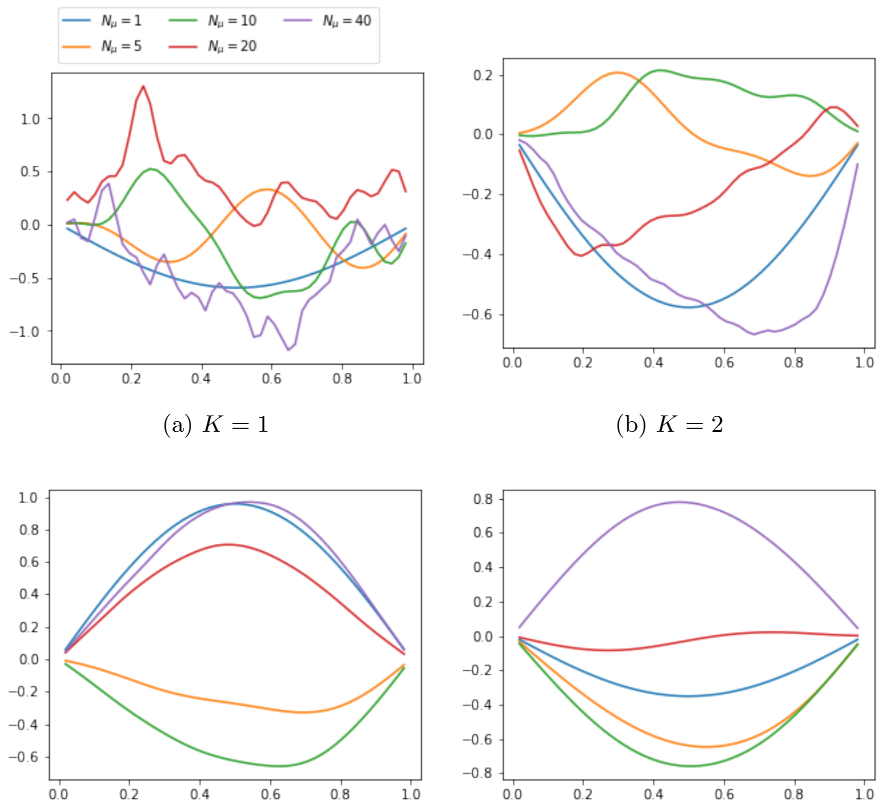
$$\frac{1}{N_M} \left\|\mathbf{u}(\boldsymbol{\mu}) - \mathbf{u}_{N_{DNN}}(\boldsymbol{\mu};\boldsymbol{\Theta})\right\|^2 \tag{5}$$

with the norm $\|\mathbf{u}(\boldsymbol{\mu})\|^2 = \frac{1}{N_h+1} \sum_{i=1}^{N_M} \sum_{j=1}^{N_h} u(x_j,\boldsymbol{\mu}^i)^2$. Given this norm, we would like to find the neural network that fits the finite difference solution the best. In other words, we want to find $\boldsymbol{\Theta}^* \in \mathbb{R}^{N_{DNN}}$ such that

$$\|\boldsymbol{u}_{N_h}(\boldsymbol{\mu}) - \boldsymbol{u}_{DNN}(\boldsymbol{\mu},\boldsymbol{\Theta}^*)\|^2 \leq \|\boldsymbol{u}_{N_h}(\boldsymbol{\mu}) - \boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta})\|^2$$

for all $\boldsymbol{\Theta} \in \mathbb{R}^{N_{DNN}}$.

In this paper, we first present the Newmark scheme to compute $\boldsymbol{u}_{N_h}(\boldsymbol{\mu})$ and study its behaviour. Then, we present the Monte Carlo method in order to justify the approximation (5). After that, a short introduction of DNN is given, with some details about the implementation. Finally, we compute the approximation of the difference (5) in between the exact solution and the solution given by the DNN, and study its performance with respect to different parameters.



(a) $K = 1$ (b) $K = 2$

(c) $K = 3$ (d) $K = 4$

Figure 1: Initial condition $u_0(x, \boldsymbol{\mu})$ with respect to x (horizontal axis) and for different N_μ and K (see legend).

2 Finite difference approximation of the 1D wave equation

Let N_h, $N_\tau > 0$, $h = \frac{1}{N_h+1}$, $x_i = ih$ for $i = 0, 1, ..., N_h + 1$ and let $\tau > 0$ be the time step, $t_n = n\tau$ for $n = 0, 1, ..., N_\tau$. Let u_i^n be an approximation of the solution $u(x_i, t_n)$ (2). We set for all $i = 1, .., N_h$:

$$\begin{aligned} u_i^0 &= u_0(x_i), \\ u_i^1 &= u_i^0 + \frac{\tau^2}{2} \frac{u_{i-1}^0 - 2u_i^0 + u_{i+1}^0}{h^2}, \end{aligned} \tag{6}$$

and, for $n = 1, \ldots, N_\tau - 1$, for all $i = 1, \ldots, N_h$:

$$\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\tau^2} - \frac{u_{i-1}^n - 2u_i^n + u_{i+1}^n}{h^2} = 0. \tag{7}$$

This scheme is called the *Newmark scheme* [6]. Especially, for the case where

$$u_0(x) = \sin(j\pi x) \qquad \text{with} \qquad j \in \mathbb{N}^*, \tag{8}$$

this scheme gives $u_i^n = \alpha_n \sin(j\pi ih)$. The coefficients α_n are defined recursively as

$$\begin{aligned} \alpha_0 &= 1, \\ \alpha_1 &= 1 - \frac{\tau^2}{h^2} \left(1 - \cos(j\pi h)\right), \\ \alpha_n &= 2\alpha_1 \alpha_{n-1} - \alpha_{n-2} \quad \text{for} \quad n > 1. \end{aligned} \tag{9}$$

Then, by induction on n, α_n can be re-written as

$$\alpha_n = \frac{1}{2}(s_+^n + s_-^n)$$

for $n \geq 0$, with

$$s_+ = \alpha_1 + \sqrt{\alpha_1^2 - 1} \qquad \text{and} \qquad s_- = \alpha_1 + \sqrt{\alpha_1^2 - 1}.$$

Now, narrowing our problem to $h = \tau$, one can prove with the above that:

$$\alpha_n = \cos(j\pi n\tau) \tag{10}$$

(see details in appendix A). Finally, this leads to the final expression of the Newmark scheme

$$u_i^n = \cos(j\pi n\tau)\sin(j\pi ih).$$

Note that this approximation is exact on (x_i, t_n) ($i = 1, \dots, N_h$ and $n = 0, 1, \dots, N_\tau$), namely

$$u(x_i, t_n) = \cos(j\pi t_n)\sin(j\pi x_i) = u_i^n.$$

This result will be useful in our study, since $u_0(x)$ is a linear combination of sinusoidals.

2.1 Convergence of the error $u - u_{N_h}$

We study the convergence of the difference in time and space for $\boldsymbol{\mu}$ an instance of $\mathcal{P}$. Consider the Newmark scheme (6) - (7). It is stable if the following CFL condition is satisfied:

$$\tau \leq h$$

The proof can be found in ”Introduction à l'analyse numérique” [5]. Moreover, it can be shown that if the CFL condition is satisfied and if u is smooth enough, the scheme is of order 2. More precisely:

$$\sqrt{\sum_{i=1}^{N_h}\left(u(x_i, 1) - u_i^{N_\tau+1}\right)^2} \leq Ch^2 \tag{11}$$

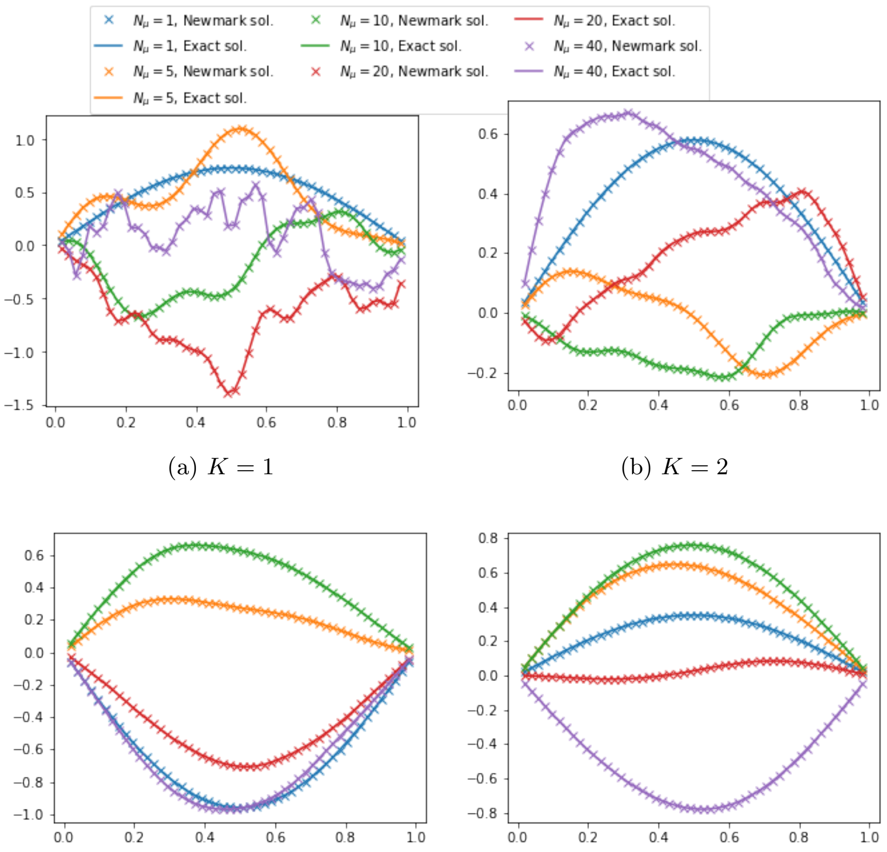
with C a constant independent of h and τ.

2.2 Numerical validation of the convergence

We implement the Newmark scheme when the initial condition u_0 is given by (3). We set the parameters $\boldsymbol{\mu}$ as an instance of $\mathcal{P}$. First, consider the case $\tau = h$ and notice that the initial condition is a linear combination of sinus functions. As the Newmark scheme is a linear iterative scheme and since it is exact on sinusoidal like (8), we have that the approximation is in fact exact in this case too. After implementation for different N_μ, N_h and K, we validate this result, as the difference in between the exact solution and the approximated one were of order 10^{-13} to 10^{-15} (machine error). Figure 2 presents the exact and approximated solutions for various parameters N_μ and K at the final time, with $N_h = N_\tau = 50$.

Then, we check that the solution has convergence of order 2 with $\tau \leq h$. In this case, we set $K = 2$ and $N_\mu = 5$. Several number of space steps are taken: $N_h \in \{8,\ 16,\ 32,\ 64,\ 128,\ 256\}$. The time step is set such that $\tau = \frac{h}{4}$ and $N_\tau = 4(N_h + 1)$ (so that $\tau < h$). From figure 3, we conclude that the convergence is indeed of order 2.

From now on, we will always take $N_h = N_\tau$ and $\tau = h$.



(a) $K = 1$ (b) $K = 2$

(c) $K = 3$ (d) $K = 4$

Figure 2: Exact solution and approximated solution given by the Newmark scheme at the final time, with respect to x and several N_μ and K, when $\tau = h$, $N_h = N_\tau = 50$.

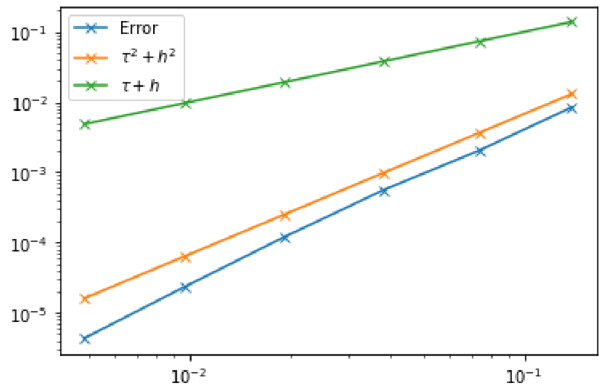


Figure 3: Error (11) with respect to $\tau + h$, where $\tau = \frac{h}{4}$, $N_\tau = 4(N_h + 1)$, $\boldsymbol{\mu}$ instances of $\mathcal{P}$, $N_\mu = 5$ and $K = 2$.

3 Monte-Carlo Method

This section follows [3]. Consider a random variable Z that is the output quantity of a stochastic model. The goal is to compute its expectation

$$m = \mathbb{E}[Z].$$

The Monte Carlo method consists of generating N_M i.i.d replicas $Z^{(1)}, ..., Z^{(N_M)}$ of Z and estimating m by a *sample mean estimator*

$$\hat{m}_{N_M} = \frac{1}{N_M}\sum_{i=1}^{N_M} Z^{(i)}.$$

We assume that $\delta(Z) = \sigma^2 < +\infty$. The sample mean estimator $\hat{m}_{N_M}$, which we also call the *Monte Carlo estimator*, has the following properties. First, it is *unbiased*, i.e. $\mathbb{E}[\hat{m}_{N_M}] = m$:

$$\mathbb{E}[\hat{m}_{N_M}] = \mathbb{E}[\frac{1}{N_M}\sum_{i=1}^{N_M} Z^{(i)}] = \frac{1}{N_M}\sum_{i=1}^{N_M} \mathbb{E}[Z^{(i)}] = m$$

where the expectation here is taken with respect to the distribution of the samples $Z^{(1)}, ..., Z^{(N_M)}$. Moreover, we have $\delta(\hat{m}_{N_M}) = \frac{\sigma^2}{N_M}$:

$$\begin{aligned}\delta(\hat{m}_{N_M}) &= \mathbb{E}[(\hat{m}_{N_M} - \mathbb{E}[\hat{m}_{N_M}])^2] = \mathbb{E}\left[\left(\frac{1}{N}\sum_{i=1}^{N_M}(Z^{(i)} - m)\right)^2\right] \\ &= \frac{1}{N_M^2}\sum_{i,j=1}^{N_M} \mathbb{E}\left[(Z^{(i)} - m)(Z^{(j)} - m)\right] \\ &= \frac{1}{N_M^2}\sum_{i=1}^{N_M} \mathbb{E}\left[(Z^{(i)} - m)^2\right] + \frac{1}{N_M^2}\sum_{i\neq j}^{N_M} \mathbb{E}\left[(Z^{(i)} - m)(Z^{(j)} - m)\right] \\ &= \frac{\sigma^2}{N_M}.\end{aligned} \tag{12}$$

The last equality is implied by $\mathbb{E}\left[(Z^{(i)} - m)^2\right] = \sigma^2 \ \forall i$ since $Z^{(i)}$ are i.i.d., and

$$\mathbb{E}\left[(Z^{(i)} - m)(Z^{(j)} - m)\right] = \mathbb{E}\left[(Z^{(i)} - m)\right]\mathbb{E}\left[(Z^{(j)} - m)\right] = 0$$

since $Z^{(i)}$ and $Z^{(j)}$ are independent. Then, by the Strong Law of Large Numbers, we have the almost sure convergence $\hat{m}_{N_M} \xrightarrow{N_M\to\infty} m$ a.s., since $\mathbb{E}[Z] = m$. By the Central Limit Theorem (CLT), we have the asymptotic normality

$$\frac{\sqrt{N_M}(\hat{m}_{N_M} - m)}{\sigma} \xrightarrow{d} N(0, 1)$$

since $\delta(Z) < +\infty$ (where $\xrightarrow{d}$ means convergence in distribution). Furthermore, using the CLT, we construct the *asymptotic* $1 - \alpha$ *confidence interval*, which is an interval with coverage probability $1 - \alpha$:

$$I_{\alpha,N_M} = \left[\hat{m}_{N_M} - c_{1-\alpha/2}\frac{\sigma}{\sqrt{N_M}}, \hat{m}_{N_M} + c_{1-\alpha/2}\frac{\sigma}{\sqrt{N_M}}\right]$$

with c_α the α-quantile of the normal distribution satisfying $\Phi(c_\alpha) = \alpha$ and Φ the cumulative distribution function of a standard normal random variable. This means that

$$P(\mu_N \in I_{\alpha,N_M}) \xrightarrow{N_M\to\infty} 1 - \alpha.$$

Indeed, this result follows from

$$\begin{aligned} P(\mu_N \in I_{\alpha,N_M}) &= P\left(\hat{m}_{N_M} - c_{1-\alpha/2}\frac{\sigma}{\sqrt{N_M}} \leq \mu_N \leq \hat{m}_{N_M} + c_{1-\alpha/2}\frac{\sigma}{\sqrt{N_M}}\right) \\ &= P\left(-c_{1-\alpha/2} \leq \frac{\sqrt{N_M}(\mu_N - \hat{m}_{N_M})}{\sigma} \leq c_{1-\alpha/2}\right) \\ &\xrightarrow{N_M\to\infty} 2(1-\alpha) - 1 = 1-\alpha. \end{aligned}$$

In other words, the error $|m - \hat{m}_{N_M}|$ satisfies

$$|m - \hat{m}_{N_M}| \leq c_{1-\alpha/2}\frac{\sigma}{\sqrt{N_M}}$$

with probability $1-\alpha$, asymptotically. So the CLT shows that the Monte Carlo error is of order $1/\sqrt{N_M}$. This is a very slow convergence rate, but it holds under quite weak assumptions ($\delta(Z) < +\infty$). However, the previous error estimate and confidence interval are not practical, as the variance σ^2 is usually unknown. We replace it by the *sample variance estimator* computed using the same sample $Z^{(1)}, ..., Z^{(N_M)}$

$$\hat{\sigma}^2_{N_M} = \frac{1}{N_M - 1}\sum_{i=1}^{N_M}\left(Z^{(i)} - \hat{m}_{N_M}\right)^2.$$

It is also an unbiased estimator and converges almost surely to σ^2. Then $\frac{\sigma}{\hat{\sigma}_{N_M}} \to 1$ a.s. and

$$\frac{\sqrt{N_M}(\hat{m}_{N_M} - m)}{\hat{\sigma}_{N_M}} = \frac{\sigma}{\hat{\sigma}_{N_M}}\frac{\sqrt{N_M}(\hat{m}_{N_M} - m)}{\sigma} \xrightarrow{d} N(0,1).$$

Therefore a *computable* asymptotic confidence interval is given by

$$\hat{I}_{\alpha,N_M} = \left[\hat{m}_{N_M} - c_{1-\alpha/2}\frac{\hat{\sigma}_{N_M}}{\sqrt{N_M}}, \hat{m}_{N_M} + c_{1-\alpha/2}\frac{\hat{\sigma}_{N_M}}{\sqrt{N_M}}\right]. \tag{13}$$

Finally, this leads to $P\left(\mu_N \in \hat{I}_{\alpha,N_M}\right) \xrightarrow{N_M\to\infty} 1-\alpha$.

3.1 Monte Carlo to approximate integrals

Consider a stochastic model $Z = \psi(X_1, ..., X_d)$ with $\psi : \mathbb{R}^d \to \mathbb{R}$ bounded and $\boldsymbol{X} = (X_1, ..., X_d)$ a random vector with joint probability density function $f : \mathbb{R}^d \to \mathbb{R}_+$. Then

$$\mathbb{E}[Z] = \int_{\mathbb{R}^d} \psi(x_1, ..., x_d) f(x_1, ..., x_d)\,\mathrm{d}x_1 ... \,\mathrm{d}x_d = \int_{\mathbb{R}^d} \psi(\boldsymbol{x}) f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}.$$

A Monte Carlo approximation of $m = \mathbb{E}[Z]$ consists of generating N_M i.i.d replicas $X^{(i)} \sim f$ and computing $\hat{m}_{N_M} = \frac{1}{N_M}\sum_{i=1}^{N_M}\psi(\boldsymbol{X}^{(i)}) \approx \int_{\mathbb{R}^d}\psi(\boldsymbol{x})f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$.

This method is very useful for approximating high dimensional integrals. Indeed, the usual quadrature formula (e.g. the mid-point rule) is of order $O(N^{-2/d})$, which depends on the number of dimensions d, whereas the Monte Carlo method is of order $O(N^{-1/2})$. Therefore, for $d > 4$, the Monte Carlo method is favoured compared to the quadrature formula.

3.2 Numerical validation of the Monte Carlo method

We compute two integrals with the Monte Carlo method, in order to experiment with that method and check its properties. The first integral is a simple generic integral, and the second one is the integral of the exact solution of the wave equation.

3.2.1 Integration of $\exp\{(-\frac{1}{2}x^2)\}$

We check the convergence of the Monte-Carlo method on the integral

$$\int_{-2}^{2} \exp\left\{\left(-\frac{1}{2}x^2\right)\right\} \mathrm{d}x. \tag{14}$$

In that case, define $Z = \psi(x) = \exp\{(-\frac{1}{2}x^2)\}$ where $\psi : \mathbb{R} \to \mathbb{R}$ is bounded, with x a random vector with joint probability density function $f : \mathbb{R} \to \mathbb{R}$ defined by $f(x) = \frac{1}{4}\mathbf{1}\{x \in [-2,2]\}$, which is equal to $1/4$ if x belongs to $[-2,2]$, and 0 otherwise. Then we compute the integral

$$\int_{-2}^{2} \exp\left\{\left(-\frac{1}{2}x^2\right)\right\} \mathrm{d}x = 4\int_{\mathbb{R}} \psi(x)f(x)\,\mathrm{d}x = 4\mathbb{E}[Z]$$

by varying the number of samples N_M, we generate N_M i.i.d replicas $X^{(i)}$, and compute

$$4\hat{m}_{N_M} = \frac{4}{N_M}\sum_{i=1}^{N_M} \psi(X^{(i)}).$$

We repeat this procedure N_{trials} times and generate N_{trials} replicas $\hat{m}_{N_M}^{(1)}, ..., \hat{m}_{N_M}^{(N_{trials})}$. Notice that N_{trials} is fixed and does not depend on N_M. Here we set $N_{trials} = 200$ in the implementation as we will see that the convergence is good enough. Then we compute an approximation of $\mathbb{E}[\hat{m}_{N_M}]$

$$\hat{m}_{N_{trials}} = \frac{1}{N_{trials}}\sum_{i=1}^{N_{trials}} \hat{m}_{N_M}^{(i)}$$

and an approximation of the variance $\delta(\hat{m}_{N_M})$

$$\hat{\sigma}^2_{N_{trials}} = \frac{1}{N_{trials}-1}\sum_{i=1}^{N_{trials}} \left(\hat{m}_{N_M}^{(i)} - \hat{m}_{N_{trials}}\right)^2.$$

We verify the convergence of $\hat{\sigma}^2_{N_{trials}}$ is in $O(1/N_M)$, using equation (13). On figure 4, the convergence

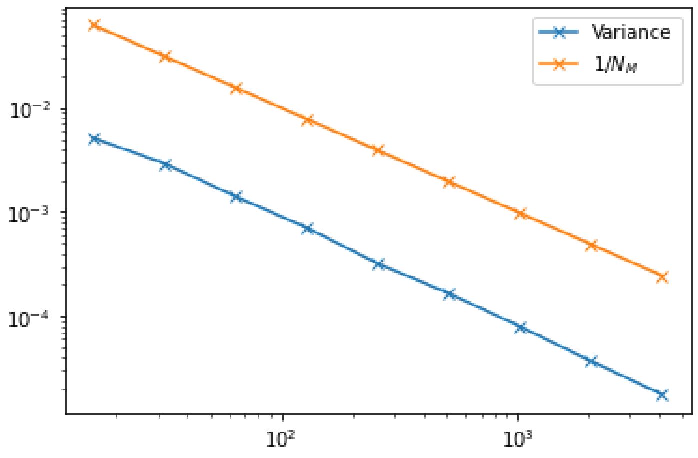


Figure 4: Convergence of $\hat{\sigma}^2_{N_{trials}}$ with respect to N_M, the number of replicas.

of the approximation of the variance is indeed in $O(1/N_M)$.

3.2.2 Integration of the solution

Let $u_0(x, \boldsymbol{\mu})$ be the initial condition as in equation (3), with $K = 1$. The solution at the final time $u(x, \boldsymbol{\mu})$ is given by D'Alembert formula (2). We want to compute the integral:

$$\begin{aligned} I &= \frac{1}{2^{N_\mu}} \int_{[-1,1]^{N_\mu}} \int_{[0,1]} |u(x, \boldsymbol{\mu})|^2 \,\mathrm{d}x \,\mathrm{d}\boldsymbol{\mu} \\ &= \mathbb{E}\left(|u(x, \boldsymbol{\mu})|^2\right). \end{aligned} \tag{15}$$

The analytical result of the integral is

$$I = \frac{1}{6} \sum_{j=1}^{N_\mu} \frac{1}{j^2}.$$

We implement an approximation of the integral using Monte-Carlo, for the variable $(x, \boldsymbol{\mu}) \in [0, 1] \times [-1, 1]^{N_\mu}$. Later, we will use a quadrature formula for approximating the integral in space. Here, however, we do a numerical validation of the Monte-Carlo method. Similarly to the previous integral, we can write I as:

$$I = \int_{\mathbb{R} \times \mathbb{R}^{N_\mu}} \psi(x, \boldsymbol{\mu}) f(x, \boldsymbol{\mu}) \,\mathrm{d}x \,\mathrm{d}\boldsymbol{\mu}$$

with $\psi(x, \boldsymbol{\mu}) = |u(x, \boldsymbol{\mu})|^2$ and the density function $f(x, \boldsymbol{\mu}) = \frac{1}{2^{N_\mu}} \mathbf{1}\{(x, \boldsymbol{\mu}) \in [0, 1] \times [-1, 1]^{N_\mu}\}$. We obtain the sample mean estimator

$$\hat{m}_{N_M} = \frac{1}{N_M} \sum_{i=1}^{N_M} \left|u(x_i, \boldsymbol{\mu}^i)\right|^2$$

with replicas $(x_1, \boldsymbol{\mu}^1), ..., (x_{N_M}, \boldsymbol{\mu}^{N_M})$. We set the number of parameters $N_\mu = 5$. We verify that the value $\hat{m}_{N_M}$ tends to the exact value I, with a confidence interval given by equation (13), where $\alpha = 0.05$ (in order to have an exactitude of 95%). We approximate $\mathbb{E}[\hat{m}_{N_M}]$ by generating N_{trials} replicas $\hat{m}_{N_M}^{(1)}, ..., \hat{m}_{N_M}^{(N_{trials})}$, and estimate the exact value I by the empirical mean

$$\hat{m}_{N_{trials}} = \frac{1}{N_{trials}} \sum_{i=1}^{N_{trials}} \hat{m}_{N_M}^{(i)}.$$

Moreover, we use the approximation of the variance

$$\hat{\sigma}^2_{N_{trials}} = \frac{1}{N_{trials} - 1} \sum_{i=1}^{N_{trials}} \left(\hat{m}_{N_M}^{(i)} - \hat{m}_{N_{trials}}\right)^2. \tag{16}$$

To set the number of trials N_{trials}, we make several experiments by varying N_{trials} and check when the convergence is smooth enough. Here we have a good result with $N_{trials} = 200$. On figure 5, we see that $\hat{m}_{N_{trials}}$ indeed tends to I.

Then, we check the convergence of the variance as in equation (12). We compute the variance once again by generating $N_{trials} = 200$ replicas $\hat{m}_{N_M}^{(1)}, ..., \hat{m}_{N_M}^{(N_{trials})}$, and compute an approximation of the variance $\hat{\sigma}^2_{N_{trials}}$ as in equation (16). On figure 6, the variance is plot with respect to the number of replicas N_M and the number of parameters N_μ. The convergence of the variance is indeed in $O(1/N_M)$, as expected.

To summarize our results, the exact integrals I and the values $\hat{m}_{N_{trials}}$ and $\hat{\sigma}^2_{N_{trials}}$ for $N_M = 1024$ are placed in table 1 with respect to various N_μ. That way, we notice that $\hat{m}_{N_{trials}}$ is a good approximation of I, for the different N_μ considered. Moreover, the variances are reasonable with respect to I.

To conclude, with this approximation of the integrals (14) and (15), we have shown that the Monte Carlo is indeed effective to compute integrals. In particular, it is very useful in the case of multidimensional integrals, like in the second computed integral, as it has a better convergence than the quadrature formula.

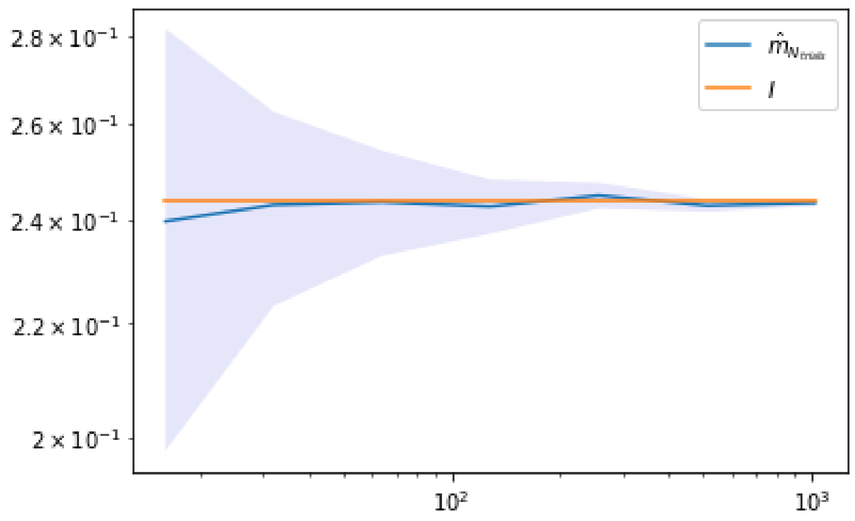


Figure 5: Convergence of $\hat{m}_{N_{trials}}$ with respect to the number of samples N_M ($N_\mu = 5$). Error bars are given by $c_{1-\alpha/2}\frac{\hat{\sigma}_{N_{trials}}}{\sqrt{N_M}}$, where $\alpha = 0.05$.

	N_μ					
	1	5	10	15	20	25
I	0.167	0.244	0.258	0.263	0.266	0.267
$\hat{m}_{N_{trials}}$	0.167	0.245	0.259	0.264	0.265	0.267
$\hat{\sigma}^2_{N_{trials}}$	$8.78 \cdot 10^{-5}$	$2.11 \cdot 10^{-4}$	$2.34 \cdot 10^{-4}$	$2.59 \cdot 10^{-4}$	$2.67 \cdot 10^{-4}$	$2.68 \cdot 10^{-4}$

Table 1: Summary of the results, for $N_M = 1024$ and $N_{trials} = 200$ replicas.

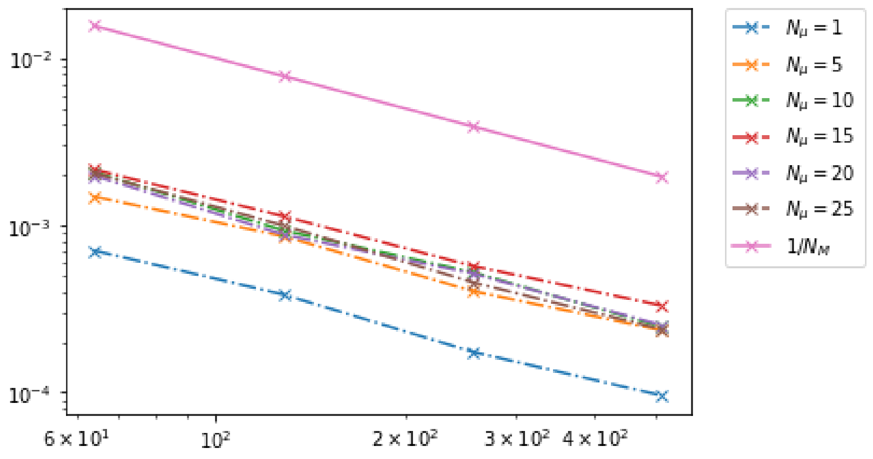


Figure 6: Convergence of $\hat{\sigma}^2_{N_{trials}}$ with respect to N_M (horizontal axis), for different number of parameters N_μ (legend).

3.3 Approximation of $\frac{1}{2^{N_\mu}} \int_{[-1,1]^{N_\mu}} \int_0^1 \big(u(x,\boldsymbol{\mu}) - u_{DNN}(x,\boldsymbol{\mu};\boldsymbol{\Theta})\big)^2 \,\mathrm{d}x\,\mathrm{d}\boldsymbol{\mu}$

As mentioned in the introduction, we want to assess the error between u and $\boldsymbol{u}_{DNN}(\boldsymbol{\mu},\boldsymbol{\Theta})$ with integral (4). To compute a numerical approximation of the integral, we use the Monte-Carlo method to compute the integral with respect to $\boldsymbol{\mu}$, by selecting N_M replicas $\{\boldsymbol{\mu}^1,...,\boldsymbol{\mu}^{N_M}\}$, instances of $\mathcal{P}$:

$$\frac{1}{N_M}\sum_{i=1}^{N_M}\int_0^1 \big(u(x,\boldsymbol{\mu}^i) - u_{DNN}(x,\boldsymbol{\mu}^i;\boldsymbol{\Theta})\big)^2\,\mathrm{d}x.$$

Then we discretize with respect to x by taking N_h equidistant points $x_i = ih$, for $i = 1,...,N_h$, with $h = \frac{1}{N_h+1}$. Hence

$$\begin{aligned}
&\frac{1}{N_M}\sum_{i=1}^{N_M}\int_0^1 \big(u(x,\boldsymbol{\mu}^i) - u_{DNN}(x,\boldsymbol{\mu}^i;\boldsymbol{\Theta})\big)^2\,\mathrm{d}x\\
&= \frac{1}{N_M}\sum_{i=1}^{N_M}\sum_{j=0}^{N_h}\int_{x_j}^{x_{j+1}} \big(u(x,\boldsymbol{\mu}^i) - u_{DNN}(x,\boldsymbol{\mu}^i;\boldsymbol{\Theta})\big)^2\,\mathrm{d}x\\
&\approx \frac{1}{N_M}\sum_{i=1}^{N_M}\sum_{j=1}^{N_h} h\big(u(x_j,\boldsymbol{\mu}^i) - u_{DNN}(x_j,\boldsymbol{\mu}^i;\boldsymbol{\Theta})\big)^2\\
&= \frac{1}{N_M}\frac{1}{N_h+1}\sum_{i=1}^{N_M}\sum_{j=1}^{N_h} \big(u(x_j,\boldsymbol{\mu}^i) - u_{DNN}(x_j,\boldsymbol{\mu}^i;\boldsymbol{\Theta})\big)^2,
\end{aligned}$$

where the third line is an approximation of a finite integral with a basic quadrature rule. Moreover, the sum over j starts at 1, because we supposed that $u_{DNN}(x_0,\boldsymbol{\mu}^i;\boldsymbol{\Theta}) = u(x_0,\boldsymbol{\mu}^i) = 0\ \forall i$. In short we can approximate the difference in between the 2 solutions by:

$$\frac{1}{2^{N_\mu}}\int_{[-1,1]^{N_\mu}}\int_0^1 \big(u(x,\boldsymbol{\mu}) - u_{N_{DNN}}(x,\boldsymbol{\mu};\boldsymbol{\Theta})\big)^2\,\mathrm{d}x\,\mathrm{d}\boldsymbol{\mu} \approx \frac{1}{N_M}\left\|\mathbf{u}(\boldsymbol{\mu}) - \mathbf{u}_{N_{DNN}}(\boldsymbol{\mu};\boldsymbol{\Theta})\right\|^2 \tag{17}$$

with the norm $\|\mathbf{u}(\boldsymbol{\mu})\|^2 = \frac{1}{N_h+1}\sum_{i=1}^{N_M}\sum_{j=1}^{N_h} u(x_j,\boldsymbol{\mu}^i)^2$.

4 Neural Network

4.1 Introduction

This introduction to Neural Network (NN) follows [1]. It presents the basis to Machine Learning, but restrains to the content that is used in the specific problem we are considering in this study. In particular, we restric ourselves to neural networks called *multi-layer perceptron.* For further readings, see [2].

Consider the *training set* composed of pairs of input $\boldsymbol{x}^i \in \mathbb{R}^{N_{input}}$ and target $\boldsymbol{y}^i \in \mathbb{R}^{N_{output}}$:

$$\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \dots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}.$$

We would like to find a function $f : \mathbb{R}^{N_{input}} \to \mathbb{R}^{N_{output}}$ such that $f(\boldsymbol{x}^i)$ provides an approximation $\hat{\boldsymbol{y}}^i$ of the target $\boldsymbol{y}^i$ for each $i = 1, \dots, N$. In other words, we would like that for any input, the function f provides an output as close as possible to the target answer. In particular, we represent the function f as a response to a *multi-layer perceptron.* It is defined with a depth $L \in \mathbb{N}^*$, a width $M \in \mathbb{N}^*$, some weights

$$\begin{aligned} &\boldsymbol{w}^{(1)} \in \mathbb{R}^M \times \mathbb{R}^{N_{input}}, \\ &\boldsymbol{w}^{(2)}, \dots, \boldsymbol{w}^{(L-1)} \in \mathbb{R}^M \times \mathbb{R}^M, \\ &\boldsymbol{w}^{(L)} \in \mathbb{R}^{N_{output}} \times \mathbb{R}^M, \end{aligned} \tag{18}$$

some biases

$$\begin{aligned} &\boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L-1)} \in \mathbb{R}^M \\ &\boldsymbol{b}^{(L)} \in \mathbb{R}^{N_{output}} \end{aligned} \tag{19}$$

and activation functions

$$\begin{aligned} &\sigma^{(l)} : \mathbb{R}^M \to \mathbb{R}^M \quad \text{for} \quad l = 1, \dots, L-1, \\ &\sigma^{(L)} : \mathbb{R}^{N_{output}} \to \mathbb{R}^{N_{output}}. \end{aligned}$$

Then, the output $\hat{\boldsymbol{y}}^i$ is obtained recursively. Set $\boldsymbol{a}^{(0)} = \boldsymbol{x}^i$ and iterate:

$$\forall l = 1, ..., L \quad \begin{cases} \boldsymbol{s}^{(l)} = \boldsymbol{w}^{(l)} \cdot \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)} \\ \boldsymbol{a}^{(l)} = \sigma^{(l)}(\boldsymbol{s}^{(l)}). \end{cases} \tag{20}$$

The first operation can be represented as in figure 7. Each component $\boldsymbol{w}^{(1)}_{1,j}$ (on the black line) multiplies $\boldsymbol{a}^{(0)}_j$ represented by the green neuron on the left end of the line. Then all the products $\boldsymbol{w}^{(1)}_{1,j} \cdot \boldsymbol{a}^{(0)}_j$ are summed up with the bias $\boldsymbol{b}^{(1)}_1$, which gives $\boldsymbol{s}^{(1)}_1$. After that, the activation function $\sigma^{(1)}$ is evaluated in $\boldsymbol{s}^{(1)}_1$, which results in the blue neuron $\boldsymbol{a}^{(1)}_1$. Now, the blue neuron column can be seen as the vector $\boldsymbol{a}^{(1)} = [\boldsymbol{a}^{(1)}_1, \dots, \boldsymbol{a}^{(1)}_m]^\top$. It is called a *layer* of the neural network. In particular, if $1 \le l \le L-1$, then $\boldsymbol{a}^{(l)}$ is called a *hidden layer.* Furthermore, diagram 7 can be extended to all layers $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \dots, \boldsymbol{a}^{(L)}$, which compose a *neural network*, see figure 8.

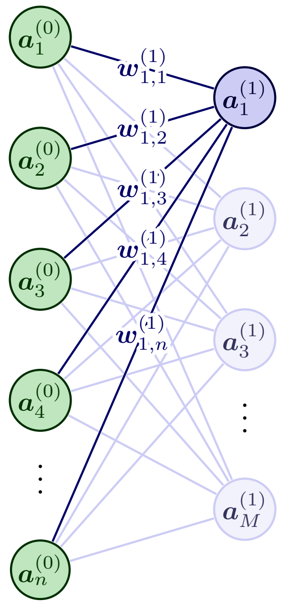
It represents the operations from an input $\boldsymbol{x}$ to an output $\hat{\boldsymbol{y}} = \boldsymbol{a}^{(L)}$. For more than 2 layers, we call the model a *deep* neural network.

More formally, let N_{DNN} be the total number of weights and biases components:

$$N_{DNN} := \overbrace{MN_{input} + (L-2)M^2 + MN_{output}}^{\text{total number of weights comp.}} + \overbrace{(L-1)M + N_{output}}^{\text{total number of biases comp.}} .$$

Define $\boldsymbol{\Theta} = (\boldsymbol{w}^{(1)}, \dots, \boldsymbol{w}^{(L)}, \boldsymbol{b}^{(1)}, \dots, \boldsymbol{b}^{(L)}) \in \mathbb{R}^{N_{DNN}}$ the weights and biases, as in (18)-(19). The neural network f is a function defined as:

$$\begin{aligned} f : \mathbb{R}^{N_{input}} \times \mathbb{R}^{N_{DNN}} &\to \mathbb{R}^{output} \\ (\boldsymbol{x}, \boldsymbol{\Theta}) &\mapsto \hat{\boldsymbol{y}}. \end{aligned}$$



$$= \sigma^{(1)} \overbrace{\left(\boldsymbol{w}_{1,1}^{(1)} \boldsymbol{a}_1^{(0)} + \ldots + \boldsymbol{w}_{1,n}^{(1)} \boldsymbol{a}_n^{(0)} + \boldsymbol{b}_1^{(1)} \right)}^{\boldsymbol{s}_1^{(1)}}$$
$$= \sigma^{(1)} \left(\sum_{i=1}^{n} \boldsymbol{w}_{1,i}^{(1)} \boldsymbol{a}_i^{(0)} + \boldsymbol{b}_1^{(0)} \right)$$

Figure 7: First layer of a neural network. In our case, the input layer has dimension $n = N_{input}$.

It sends $(\boldsymbol{x}, \boldsymbol{\Theta})$ to the output $\hat{\boldsymbol{y}}$ via the iterative system (20).

To measure the closeness of $f(\boldsymbol{x}^i; \boldsymbol{\Theta})$ with $\boldsymbol{y}^i$, we introduce a *loss* function $l : \mathbb{R}^{N_{output}} \times \mathbb{R}^{N_{output}} \to \mathbb{R}$. A famous loss is the *Mean Squared Error* (MSE) :

$$l(\hat{\boldsymbol{y}}^i, \boldsymbol{y}^i) = \text{MSE}(\hat{\boldsymbol{y}}^i, \boldsymbol{y}^i) = \frac{1}{N_{output}} \sum_{j=1}^{N_{output}} (\hat{\boldsymbol{y}}_j^i - \boldsymbol{y}_j^i)^2$$

where $\boldsymbol{y}_j^i$, respectively $\hat{\boldsymbol{y}}_j^i$, is the jth component of vector $\boldsymbol{y}^i \in \mathbb{R}^{N_{output}}$, respectively $\hat{\boldsymbol{y}}^i \in \mathbb{R}^{N_{output}}$.

As we want to find the best function $f(\cdot; \boldsymbol{\Theta})$ whose output $\hat{\boldsymbol{y}}^i = f(\boldsymbol{x}^i; \boldsymbol{\Theta})$ is the closest to the target $\boldsymbol{y}^i$ for each pair $i = 1, \ldots, N$, we minimize the loss in between each pair $(\hat{\boldsymbol{y}}^i, \boldsymbol{y}^i)$ with respect to the parameters $\boldsymbol{\Theta}$. It can be formulated as an optimization problem: find $\boldsymbol{\Theta}^* \in \mathbb{R}^{DNN}$ such that for $i = 1, \ldots, N$:

$$l\left(f(\boldsymbol{x}^i; \boldsymbol{\Theta}^*), \boldsymbol{y}^i\right) \leq l\left(f(\boldsymbol{x}^i; \boldsymbol{\Theta}), \boldsymbol{y}^i\right) \quad \text{for all} \quad \boldsymbol{\Theta} \in \mathbb{R}^{DNN}.$$

From there, we loop through all pairs $(\hat{\boldsymbol{y}}^i, \boldsymbol{y}^i)$ and minimize the quantity $l(\hat{\boldsymbol{y}}^i, \boldsymbol{y}^i)$ with respect to $\boldsymbol{\Theta}$. However, looping through individual pairs is computationally consuming. That is why small groups of pairs $\{(\hat{\boldsymbol{y}}^1, \boldsymbol{y}^1), \ldots, (\hat{\boldsymbol{y}}^B, \boldsymbol{y}^B)\}$ are rather taken when optimizing. The small group is called a *batch*.

With such optimization problems, there exist a lot of techniques to find minima numerically. It can be solved with the famous *gradient descent* or *stochastic gradient descent*, under certain assumptions on $f(\cdot; \boldsymbol{\Theta})$. A more advanced algorithm is the *ADAM optimizer*, which is an adaptive learning rate optimization algorithm.

More details about how to choose the best batch size B and optimization techniques for training deep models can be found in chapter 8 of *Deep Learning* from Goodfellow et al. [2].

Until now, it was presented how the neural network f was trained so that its outputs $\hat{\boldsymbol{y}}^1, \ldots, \hat{\boldsymbol{y}}^N$ were close to the corresponding targets $\boldsymbol{y}^1, \ldots, \boldsymbol{y}^N$. This first optimization part is called the *training* part, with the corresponding *training set* $\{(\boldsymbol{x}^1, \boldsymbol{y}^1), \ldots, (\boldsymbol{x}^N, \boldsymbol{y}^N)\}$. Then, we are interested in measuring how well the machine learning algorithm performs on new data. That is why we introduce a *test set* of data, which is separate from the data used for training. The performance of the trained function $f(\cdot; \tilde{\boldsymbol{\Theta}})$ is

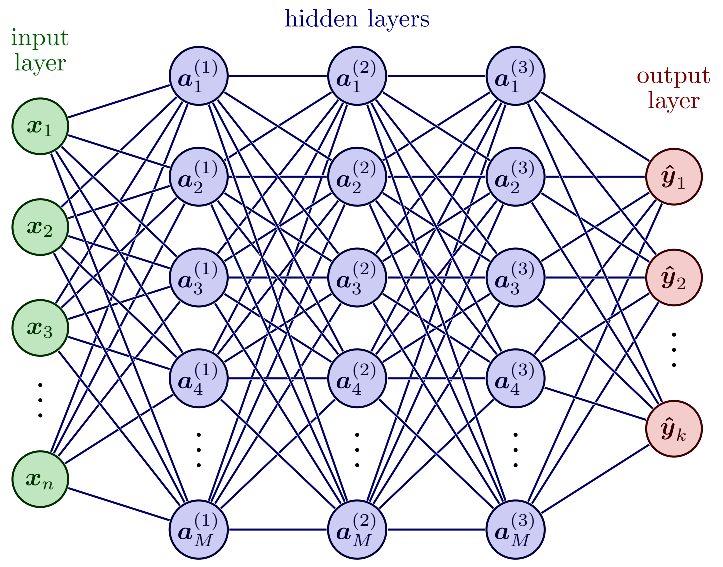


Figure 8: General representation of a neural network. Here, the number of hidden layers is 3. In our case, the input layer has dimension $n = N_{input}$ and the output layer has dimension $k = N_{output}$.

measured on the test set $\{(\tilde{\boldsymbol{x}}^1, \tilde{\boldsymbol{y}}^1), \dots (\tilde{\boldsymbol{x}}^{N_{test}}, \tilde{\boldsymbol{y}}^{N_{test}})\}$ by the *test loss*:

$$\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} l(f(\tilde{\boldsymbol{x}}^i; \tilde{\boldsymbol{\Theta}}), \tilde{\boldsymbol{y}}^i). \tag{21}$$

To conclude with the introduction, figure 9 summarizes the whole procedure of training and testing a neural network.

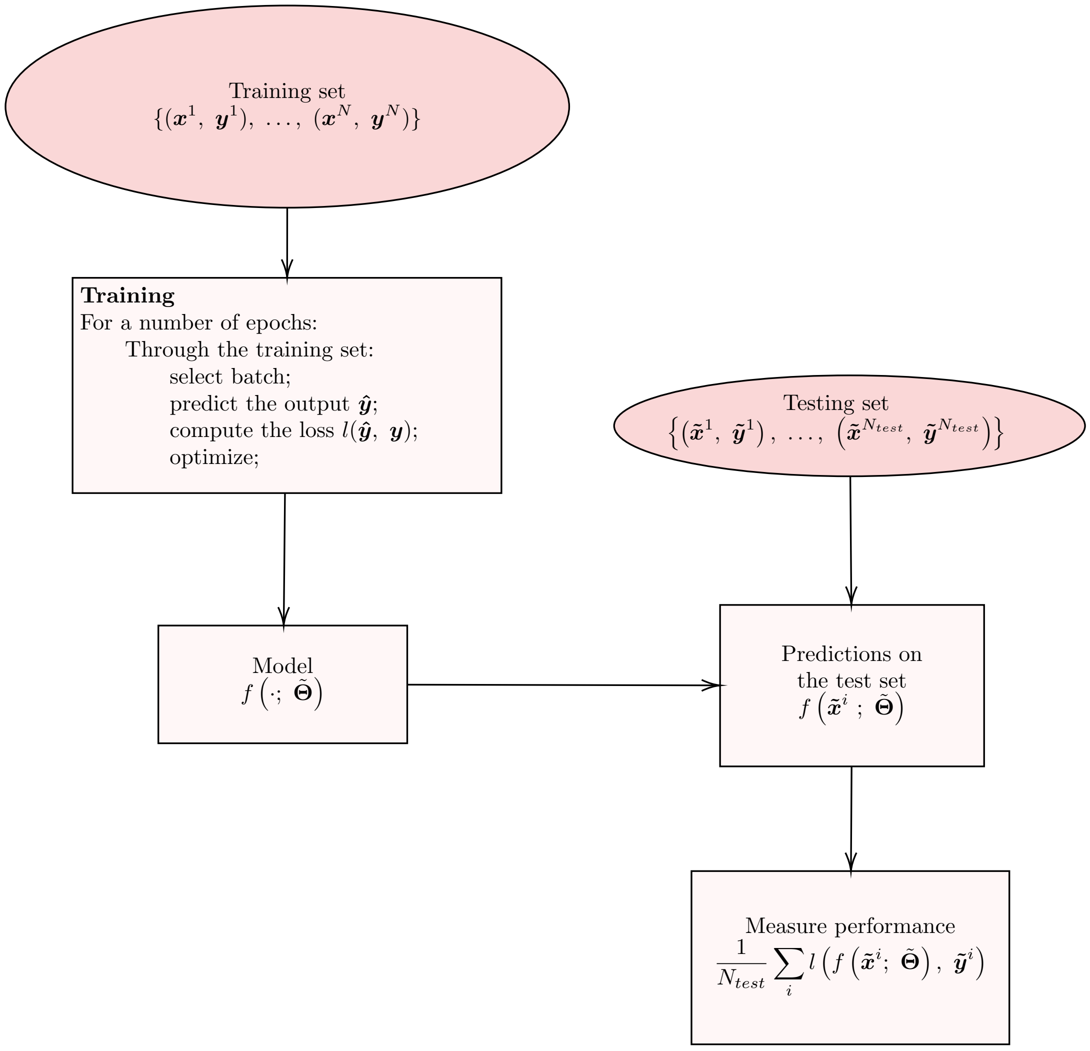

Training set
$\{(\boldsymbol{x}^1,\ \boldsymbol{y}^1),\ \ldots,\ (\boldsymbol{x}^N,\ \boldsymbol{y}^N)\}$
Training
For a number of epochs:
Through the training set:
select batch;
predict the output $\hat{\boldsymbol{y}}$;
compute the loss $l(\hat{\boldsymbol{y}},\ \boldsymbol{y})$;
optimize;
Testing set
$\left\{\left(\tilde{\boldsymbol{x}}^1,\ \tilde{\boldsymbol{y}}^1\right),\ \ldots,\ \left(\tilde{\boldsymbol{x}}^{N_{test}},\ \tilde{\boldsymbol{y}}^{N_{test}}\right)\right\}$
Model
$f\left(\cdot;\ \tilde{\boldsymbol{\Theta}}\right)$
Predictions on the test set
$f\left(\tilde{\boldsymbol{x}}^i\ ;\ \tilde{\boldsymbol{\Theta}}\right)$
Measure performance
$\frac{1}{N_{test}}\sum_i l\left(f\left(\tilde{\boldsymbol{x}}^i;\ \tilde{\boldsymbol{\Theta}}\right),\ \tilde{\boldsymbol{y}}^i\right)$

Figure 9: Diagram summarizing the training part (on the left) and the testing part (on the right) of the machine learning algorithm.

4.2 Approximation $\boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta})$ of $\boldsymbol{u}_{N_h}(\boldsymbol{\mu})$

Define the multi-layer perceptron

$$\begin{aligned} f : \mathbb{R}^{N_\mu} \times \mathbb{R}^{N_{DNN}} &\to \mathbb{R}^{N_h} \\ (\boldsymbol{\mu}, \boldsymbol{\Theta}) &\to \boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta}). \end{aligned}$$

Given $\boldsymbol{\mu}$ an instance of $\mathcal{P}$, its output $\boldsymbol{u}_{DNN}(\boldsymbol{\mu};\boldsymbol{\Theta})$ approximates $\boldsymbol{u}_{N_h}(\boldsymbol{\mu})$. Take the training set

$$\{(\boldsymbol{\mu}_1, \boldsymbol{u}_{N_h}(\boldsymbol{\mu}_1)), \dots, (\boldsymbol{\mu}_N, \boldsymbol{u}_{N_h}(\boldsymbol{\mu}_N))\}$$

where $\boldsymbol{\mu}_1, \dots, \boldsymbol{\mu}_N$ are instances of $\mathcal{P}$. The hidden layers all have the same width and have activation functions $\sigma^{(1)} = \sigma^{(2)} = \cdots = \sigma^{(L-1)} = \text{ReLU}$ (Rectified Linear Units) defined as

$$\text{ReLU}(x) := \max(0, x).$$

The last activation function is the identity map (so that the output can have negative components).

After training, define $\tilde{\boldsymbol{\Theta}} \in \mathbb{R}^{N_{DNN}}$ the optimized weights and biases. Given $\boldsymbol{\mu}$ instance of $\mathcal{P}$, $\boldsymbol{u}^{(j)}_{DNN}(\boldsymbol{\mu};\tilde{\boldsymbol{\Theta}})$ is an approximation of $u(x_j, \boldsymbol{\mu})$. We would like to compute the difference given by equation (17) between the exact solution and the trained neural network. We can rewrite is as:

$$\begin{aligned} \frac{1}{N_M} \left\| \mathbf{u}(\boldsymbol{\mu}) - \mathbf{u}_{DNN}(\boldsymbol{\mu};\tilde{\boldsymbol{\Theta}}) \right\|^2 &= \frac{1}{N_M} \frac{1}{N_h+1} \sum_{i=1}^{N_M} \sum_{j=1}^{N_h} \left(u(x_j, \boldsymbol{\mu}_i) - \mathbf{u}^{(j)}_{DNN}(\boldsymbol{\mu}_i;\tilde{\boldsymbol{\Theta}}) \right)^2 \\ &\overset{\text{(i)}}{=} \frac{1}{N_M} \frac{N_h}{N_h+1} \sum_{i=1}^{N_M} \frac{1}{N_h} \sum_{j=1}^{N_h} \left(u(x_j, \boldsymbol{\mu}_i) - \mathbf{u}^{(j)}_{DNN}(\boldsymbol{\mu}_i;\tilde{\boldsymbol{\Theta}}) \right)^2 \\ &\overset{\text{(ii)}}{=} \frac{1}{N_M} \frac{N_h}{N_h+1} \sum_{i=1}^{N_M} \text{MSE} \left(u(x_j, \boldsymbol{\mu}_i), \mathbf{u}^{(j)}_{DNN}(\boldsymbol{\mu}_i;\tilde{\boldsymbol{\Theta}}) \right). \end{aligned}$$

Equality (i) is obtained by multiplying the sum by $\frac{N_h}{N_h}$, and equality (ii) by using the definition of the MSE loss. Hence, we see that difference (17) is close to the mean of the MSE over the Monte-Carlo replicas, for N_h big enough so that $\frac{N_h}{N_h+1} \approx 1$. Consequently, given an instance $\boldsymbol{\mu}$ of $\mathcal{P}$ and when optimizing the DNN, we will use the MSE loss.

4.3 Implementation

The implementation of the deep neural network is done in python, with the library Pytorch [4]. As mentioned before, the loss function is the MSE loss. The optimization algorithm is the ADAM optimizer and the weights and biases of the neural network are initialized by default. We fix the following parameters in the training part:

- $N_\mu = 5$;
- $K = 2$ the power in the initial condition;
- $N_h = 39$ the number of space steps in the Newmark scheme;
- $h = \frac{1}{N_h+1} = 0.025$ the space step;
- $N_\tau = N_h$ the number of time steps in the Newmark scheme;
- $\tau = h$ the time step;
- number of training samples: 10'000;
- number of testing samples: 5'000;

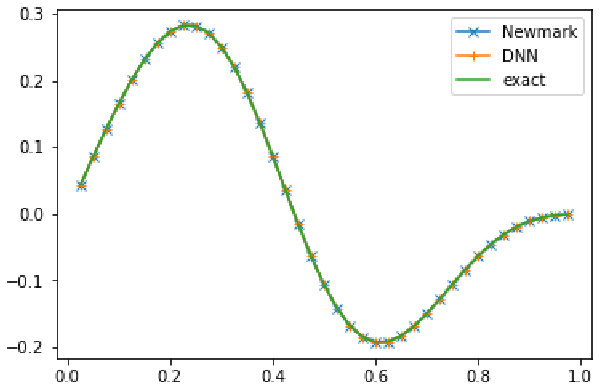


Figure 10: Exact solution and approximated solutions given by the Newmark scheme and the neural network.

- batch size: 16 (as powers of two perform better on GPUs [2]) ;
- number of epochs: 50.

We vary the following parameters:

- $HL \in \{1, 2, 3, 10, 20, 50\}$: number of hidden layers in the neural network ($HL = L - 1$),
- $M \in \{5, 10, 25, 50, 75, 100\}$: number of neurons per hidden layers. The number of neurons per hidden layer is the same for each hidden layer.
- $LR \in \{$1E-05, 5E-4, 1E-4, 5E-3, 1E-3$\}$: learning rate of the ADAM optimizer.

We use a grid-search method in order to find the best combination of parameters. In other words, we compute a model for each combination of HL, M and LR (training), then compute how well the models perform on new data (testing), to finally compare the models and pick the one with the lowest test loss. Table 2 summarizes the results. The best model obtained a test loss of $3.81 \cdot 10^{-9}$, with a learning rate of $5 \cdot 10^{-4}$, 2 hidden layers and 10 neurons. This represents $N_{DNN} = 599$ trained parameters.

Overall, the test loss is lower for shallow neural networks, i.e. with a small number of hidden layers. This is suprising as deep neural networks have shown greater performance [2]. Moreover, not a lot of neurons are necessary to obtain a good result. Only ten are necessary to get the best model, even though the output has dimension $N_h = 39$. Observing the impact of the learning rate, we notice that if it is too small, like for $LR = 10^{-5}$, then the optimizer is too slow and the loss does not go down enough.

A solution of the wave equation is plotted in figure 10. The best model was used and we indeed see its great performance. Its points coincide with the solution given by the Newmark scheme and the exact solutions.

From now on, we will train the neural network models with the best parameters we have found here, i.e. $HL = 2$, $M = 10$ and $LR = 5 \cdot 10^{-4}$.

HL	NN	LR				
		1E-05	5E-4	1E-4	5E-3	1E-3
1	5	6.79E-02	8.00E-09	3.04E-04	1.00E-06	9.02E-08
	10	3.09E-02	5.41E-08	5.64E-08	3.55E-06	2.80E-06
	25	2.41E-03	2.54E-08	7.30E-09	2.03E-08	1.05E-06
	50	5.47E-04	9.40E-09	4.06E-07	1.14E-07	4.78E-07
	75	3.64E-04	8.10E-07	9.44E-08	1.09E-07	7.63E-06
	100	2.08E-04	2.31E-08	4.86E-07	5.30E-06	2.33E-07
2	5	6.19E-02	2.77E-04	1.26E-03	1.53E-05	2.28E-07
	10	1.25E-02	3.81E-09	2.28E-07	9.62E-07	2.48E-07
	25	1.35E-03	4.76E-08	1.37E-07	1.23E-06	1.21E-06
	50	2.67E-04	1.95E-07	2.32E-07	6.41E-05	2.22E-06
	75	1.53E-04	2.04E-06	9.22E-07	2.29E-06	1.56E-07
	100	9.97E-05	3.95E-06	5.39E-06	1.49E-06	4.58E-06
3	5	1.09E-01	2.78E-04	1.01E-03	1.07E-03	3.06E-04
	10	3.20E-02	1.03E-07	1.23E-07	3.15E-07	1.87E-07
	25	1.97E-03	2.31E-07	2.53E-07	8.83E-06	2.48E-05
	50	2.60E-04	4.84E-07	1.21E-06	5.87E-06	7.85E-06
	75	1.19E-04	4.20E-06	4.98E-06	1.37E-04	8.80E-06
	100	6.73E-05	5.10E-06	9.87E-06	3.42E-06	1.25E-05
10	5	1.86E-01	1.37E-02	1.38E-02	1.83E-01	3.11E-03
	10	4.99E-02	3.08E-03	3.03E-03	1.38E-02	3.10E-03
	25	1.40E-02	1.12E-03	9.89E-04	1.38E-02	3.11E-03
	50	3.30E-03	1.01E-03	3.26E-04	1.36E-02	1.04E-03
	75	3.17E-03	1.00E-03	2.90E-04	1.24E-02	1.05E-03
	100	3.07E-03	9.88E-04	3.00E-05	3.17E-03	1.03E-03
20	5	1.86E-01	1.81E-01	1.83E-01	1.83E-01	1.87E-01
	10	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.38E-02
	25	1.85E-01	1.39E-02	1.38E-02	1.83E-01	1.86E-01
	50	1.39E-02	3.14E-03	1.38E-02	1.83E-01	1.87E-01
	75	1.39E-02	1.22E-02	4.31E-03	1.84E-01	1.40E-02
	100	6.37E-03	1.81E-01	3.12E-03	1.83E-01	1.86E-01
50	5	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.87E-01
	10	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.87E-01
	25	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.86E-01
	50	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.86E-01
	75	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.87E-01
	100	1.85E-01	1.81E-01	1.83E-01	1.83E-01	1.86E-01

Table 2: MSE loss with respect to the number of hidden layers HL, the number of neurons NN and the learning rate LR. In red: the lowest test result for a given learning rate.

4.4 Convergence of $\frac{1}{N_M}\left\|\mathbf{u}(\boldsymbol{\mu}) - \mathbf{u}_{DNN}(\boldsymbol{\mu};\tilde{\boldsymbol{\Theta}})\right\|^2$

4.4.1 $N_\mu = 5$

For $N_\mu = 5$, a model is trained with the parameters mentioned before, but with double the number of training samples, i.e. 20'000 training samples instead of 10'000. That way, the model takes more time for training, but its precision increases. Then, we repeat this procedure for $N_h \in \{19,\ 29,\ 49,\ 59\}$ with the same architecture, so that we can study how the error (17) changes with respect to the number of space steps N_h. To have a better approximation of the error for each combination of h and N_M, the final error is a mean over 10 trials. The first 3 columns of table 3 presents the final errors with respect to h and N_M. We notice that all errors are reasonable (under 10^{-6}) and that the lowest error is for $h = 0.025$, i.e. $N_h = 39$. This makes sense as we have trained our model for this exact parameter.

In addition to the table, figure 11 presents the results as a plot with respect to N_h. We notice that with the number of Monte-Carlo replicas $N_M = 10$, the error is about 10 times smaller than with $N_M = 100$ or 1000. This difference is surely due to the fact that 10 replicas is not big enough so that the Monte-Carlo method can perform well. We notice that the errors with $N_M = 100$ and $N_M = 1000$ are close. Thus 100 replicas are enough for the Monte-Carlo method to provide a good approximation of the original integral over the parameters $\boldsymbol{\mu}$.

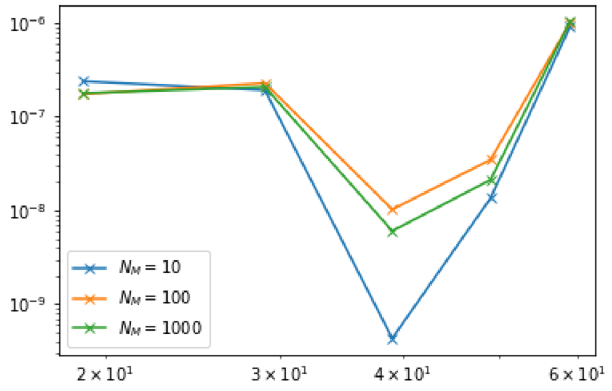


Figure 11: For $N_\mu = 5$, average error (17) over 10 trials, with respect to N_h (horizontal axis).

4.4.2 Varying N_μ

Furthermore, we train new models with the same parameters and procedure, but also for different $N_\mu = 5, 10, 15, 20, 25$. Note that we take bigger N_μ as we have seen that the Monte-Carlo method is only useful for space dimension bigger than 4. Figure 12 presents the average error over 10 trials with respect to N_h, for different N_M and N_μ. We decided to plot the error of models with the same N_μ of the same color. For example, for $N_\mu = 5$, the graphs are in blue in the figure. For more precision, the average errors (17) for $N_\mu = 5, 10, 15, 20, 25$ are also presented in table 3.

First, we notice that the models perform worse for other N_μ. The error stay in between 10^{-4} and 10^{-5}, which is superior compare to the errors for $N_\mu = 5$ that lie under 10^{-6}. The machine learning algorithm has a slightly better performance for $N_\mu = 10$. We can speculate that it is because 10 is the closest to 5 among our samples, but this needs to be confirmed by further studies. Also, the errors are not smaller for $N_h = 39$ and $N_\mu > 5$. Thus the best combination of parameters found before really depend on both N_h and N_μ. Nevertheless, the models do not perform that badly. The errors are still around 10^{-4}, even for small N_h.

If we would like to have better models for different N_h and N_μ, we could use grid-search again for each couple N_h and N_μ. However, this needs to be avoided, as grid search costs a lot computationally. For $M_1, \ldots, M_P$ the number of samples of P parameters, doing grid search on all those parameters cost

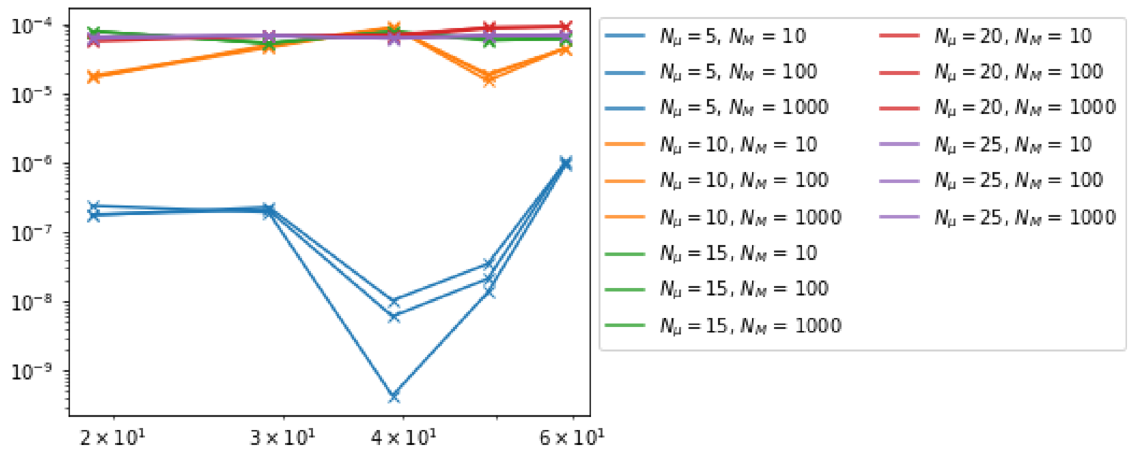


Figure 12: Average error (17) over 10 trials with respect to N_h (horizontal axis), N_μ and N_M (see legend).

$O(\prod_{i=1}^{P} M_i)$, which is a lot. Second, there is no way to predict how close the samples need to be to each other, in other words, how thin the grid should be. In our case, the problem was simple enough so that we use the grid search technique and obtain a reasonable result, but, in general, applying it to several dimensions has to be avoided.

h	N_M	N_μ				
		5	10	15	20	25
0.050	10	2.38E-07	1.68E-05	7.77E-05	5.56E-05	6.14E-05
	100	1.74E-07	1.69E-05	7.82E-05	6.17E-05	6.56E-05
	1000	1.77E-07	1.79E-05	7.88E-05	6.10E-05	6.44E-05
0.033	10	1.91E-07	5.01E-05	5.24E-05	6.72E-05	6.87E-05
	100	2.27E-07	4.57E-05	5.31E-05	6.66E-05	6.75E-05
	1000	2.07E-07	4.70E-05	5.20E-05	6.67E-05	6.62E-05
0.025	10	4.33E-10	9.01E-05	7.71E-05	6.40E-05	6.16E-05
	100	1.02E-08	8.86E-05	7.81E-05	6.97E-05	6.23E-05
	1000	6.03E-09	8.81E-05	7.76E-05	6.69E-05	6.30E-05
0.020	10	1.35E-08	1.51E-05	5.70E-05	8.55E-05	6.68E-05
	100	3.46E-08	1.79E-05	5.96E-05	8.79E-05	6.74E-05
	1000	2.11E-08	1.89E-05	5.93E-05	8.99E-05	6.84E-05
0.017	10	9.19E-07	4.45E-05	6.24E-05	9.02E-05	6.97E-05
	100	1.01E-06	4.45E-05	6.00E-05	9.21E-05	6.64E-05
	1000	1.04E-06	4.44E-05	6.03E-05	9.19E-05	6.75E-05

Table 3: Average error (17) over 10 trials with respect to N_μ, N_M and h.

4.4.3 Time study

To go further in our study, we compare the time needed by the machine learning algorithm and by the Newmark scheme in order to compute an approximation of the solution. Note that the finite difference scheme is implemented via steps (6) - (7). Table 4 displays the mean time over 1000 trials for the neural network model and the Newmark scheme, with respect to N_h and N_μ. For the machine learning model, the time taken to output a solution is constant with respect to N_μ and N_h. Its value is 0.0002 second. The constancy of time can be explained as the model simply computes ReLU evaluations and matrix operations. The latest, which takes the most computation time, is optimized well in python. As N_μ and N_h do not vary hugely, the matrix operations take approximately the same time.

On the other hand, the Newmark scheme depends on N_h, but not on N_μ. One of the reasons is that N_h is the number of space steps. The bigger it is, the more iterations the Newmark scheme needs to do, thus the more time is necessary to compute the solution. Also, python takes a lot of time to deal with loops. The finite difference scheme does not depend on N_μ because that parameter defines the initial condition. It is a linear combination of sinusoidals and stay continuous for $N_\mu \leq 25$. Because of that, the Newmark scheme does not have any difficulty to approximate the solution.

Overall, the neural network outputs a solution faster than the Newmark scheme. The bigger N_h, the bigger the difference in between the time (as the time stays constant for the machine learning algorithm, but increases for the finite difference scheme).

N_μ	Model	N_h				
		19	29	39	49	59
5	NN	0.0002	0.0002	0.0002	0.0002	0.0002
	Newmark	0.0010	0.0019	0.0032	0.0049	0.0072
10	NN	0.0002	0.0002	0.0002	0.0002	0.0002
	Newmark	0.0010	0.0018	0.0030	0.0046	0.0066
15	NN	0.0002	0.0002	0.0002	0.0002	0.0002
	Newmark	0.0009	0.0018	0.0030	0.0045	0.0064
20	NN	0.0002	0.0002	0.0002	0.0002	0.0002
	Newmark	0.0009	0.0018	0.0030	0.0045	0.0064
25	NN	0.0002	0.0002	0.0002	0.0002	0.0002
	Newmark	0.0009	0.0018	0.0030	0.0046	0.0065

Table 4: Average time (in seconds) needed to compute a solution by the neural network (NN) model and the Newmark scheme, with respect to N_μ and N_h.

5 Conclusion

In this project, our goal was to approximate the solution of the 1D wave equation with a deep neural network, given some parameters $\boldsymbol{\mu}$, instances of $\mathcal{P}$, which controlled the behaviour of the initial condition. An iterative finite difference scheme was implemented; the *Newmark scheme*. It showed great performances, to say the least, as the approximated solutions were in fact exact when the time step was equal to the space step. This was due to the fact that the initial condition was a linear combination of sinus functions. Therefore, the Newmark scheme will always be equal or better than any numerical scheme, precision-wise.

Then, to tackle the issue of approximating multi-dimensional integrals, which were used to measure the difference in between two solutions, the Monte-Carlo method was implemented. This popular method is a good way of avoiding quadrature formula which depends of the dimension of the integral.

Furthermore, a short introduction to machine learning and neural network was given. It offered an overview of the techniques used in our problem. When implementing the neural network, which was in fact a *multi-layer perceptron*, the *grid-search method* was used in order to find the parameters which would fit our data the best. The varying parameters were the number of hidden layers, the number of neurons per layer, and the learning rate of the ADAM optimizer.

The machine learning algorithm computed a solution with an error of 10^{-9}. Its performance was impressive, knowing that the neural network was only composed of 2 hidden layers with 10 neurons each. Nevertheless, it was not so adaptive to other parameters. As soon as we change N_M and/or N_μ, the machine learning algorithm's performance decreases up to 10^{-4}, in the limited number of cases we considered. We would need to perform grid search again with other parameters N_h and N_μ to find the best number of hidden layers, the best number of neurons and the best learning rate. Computing this would be very tedious and would require a lot of computational time.

In addition, considering the time study, the neural network performs way better than the Newmark scheme, as it is constant with respect to N_h and N_μ. Nevertheless, this has to be taken with a grain of salt, as the multi-layer perceptron needed a long training time before being able to work.

With more time, several points of the study could have been improved and implemented. First, when approximating the integral of the difference in between the exact solution and the approximated one, the chosen norm has the disadvantage of not considering the last point in space x_{N_h}. To counteract this problem, we could have used the trapezoidal rule instead of approximating the integral "from the left". That way, the sum over the points in space would have considered all points in space (even though, in that case, it would have put a weight of on half on the extremities).

Moreover, for a better final result, the loss of the neural network could have been the approximation of the integral of the difference between the two solutions. That way, the right norm would have been minimized, and not the MSE loss.

Acknowledgements

First, I would like to thank my supervisor, Prof. Marco Picasso, for his support and guidance through this CSE semester project. Secondly, thanks to Maude Girardin for helping me deepen my understanding of the Monte Carlo method and proofreading my paper.

Finally, I would like to thank my friends and family, who are always present for me and provide me great support. Special thanks to Alex.

A Appendix

We show equation (10) by recurrence, under the assumption that $\tau = h$. We have $\alpha_0 = 1 = \cos(0)$ and $\alpha_1 = 1 - (1 - \cos(j\pi\tau)) = \cos(j\pi\tau)$. Thus for $n = 0, 1$, we have $\alpha_n = \cos(j\pi n\tau)$. Now, for $n + 1 > 1$:

$$\alpha_{n+1} = 2\alpha_1\alpha_n - \alpha_{n-1} = 2\cos(j\pi\tau)\cos(j\pi n\tau) - \cos(j\pi(n-1)\tau)$$

by equation (9) and by recurrence hypothesis. With the trigonometric identities

$$\begin{aligned}
\cos(j\pi(k-1)\tau) &= \cos(j\pi k\tau)\cos(j\pi\tau) + \sin(j\pi k\tau)\sin(j\pi\tau)\\
\cos(j\pi(k+1)\tau) &= \cos(j\pi k\tau)\cos(j\pi\tau) - \sin(j\pi k\tau)\sin(j\pi\tau)
\end{aligned}$$

we get

$$\begin{aligned}
\alpha_{n+1} &= 2\cos(j\pi\tau)\cos(j\pi n\tau) - [\cos(j\pi n\tau)\cos(j\pi\tau) + \sin(j\pi n\tau)\sin(j\pi\tau)]\\
&= \cos(j\pi n\tau)\cos(j\pi\tau) - \sin(j\pi n\tau)\sin(j\pi\tau)\\
&= \cos(j\pi(n+1)\tau),
\end{aligned}$$

which is the expected result.

References

[1] François Fleuret. Lecture notes in deep learning, May 2022.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[3] Fabio Nobile. Lecture notes in stochastic simulation, December 2021.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[5] Jacques Rappaz and Marco Picasso. *Chapter 13: approximation de problèmes hyperboliques. Equation de transport et équation des ondes*, page 213–218. Presses polytechniques et universitaires romandes, 2017.

[6] Jacques Rappaz and Marco Picasso. *Introduction a l'analyse numerique*. Presses polytechniques et universitaires romandes, 2017.