Deep Learning for PDEs

Report of the joint APSC-NAPDE courses project

Paolo Joseph Baioni*

May 1, 2020

*paolojoseph.baioni@mail.polimi.it

Contents

Introduction

The numerical solution of partial derivative equations (PDEs) plays a fundamental role in applied mathematics, science and engineering.
The recent advances in machine learning (ML) and the successes obtained by the application of these techniques in various areas suggest the possibility of using ML in solving PDEs. This approach has considerable potential compared to other numerical methods, for example it makes feasible to write mesh-free algorithms, however the theory is not yet developed and consequently important results of convergence and stability are missing, as well as general rules that would allow to identify the optimal parameters for the design of numerical codes. Finally, some intrinsic characteristics of the method make it considerable as a tool which is complementary to traditional numerical methods, finding application in the field of real-time control.
The aim of this project is to get into deep learning techniques and study their possible applications to PDEs, also reporting some useful theoretical results, as a complement to the methods illustrated during the NAPDE course, and to implement from scratch a Numerical Analysis relevant Neural Networks based C++ program, so to both gain and verify a low-level, detailed and complete understanding of the method and to experiment on some of the programming techniques that have been deepened during the APSC and *"Strumenti di sviluppo e distribuzione di software per la ricerca scientifica"* courses held at PoliMi.
The structure of the report is as follows.
In chapter 1 we present the general architecture of a Deep Neural Network (DNN) and the main idea of functioning of the algorithm that allows it to learn from the data (Deep Learning), consisting of two phases, called *forward propagation* and *backward propagation*. Therefore, some sector-specific issues are considered in detail, such as: distinction between train, development and test sets, problems related to overfitting, possible regularization techniques aimed at reducing it, different optimization algorithms and an overview of the main parameters that must be tuned adequately to get good results.
In chapter 2 two possible approaches to solving PDEs via DNNs are exposed,

also highlighting some choices that can be made in the formulation of the problem and in the treatment of the boundary conditions. We then deepen the comparison between DNNs and the piecewise continuous linear function which are used as bases of finite element spaces of order one, in order to provide a greater intuition of the reasons why the DNN-based method works and to identify, albeit in this particular case, some indications on the ideal number of nodes and layers of the DNN.
In chapter 3 we explain the programming architectural choices that have been taken during the development of the code and we illustrate its structure and some of the content of its main files, also providing some insights with respect to the actual implementation of the previously stated theory and methods. The chapter ends with proposing and discussing possible extensions which might be taken in consideration for future developments of the code.
Chapter 4 provides instructions on how to obtain,[1] compile, link and run the program, as well as a complete benchmark case, which may be used as test example to verify the correct functioning of the code. After that we dwell on some insights on the behavior of the neural network with respect to those changes in the settings which could be of greater interest, and we illustrate a general procedure for tuning by hand the learning algorithm.
In appendix we report the [FreeFem++] code written for the computations performed in section 2.2.

[1]The code is freely available on GitHub: *https://github.com/pjbaioni/neural-net*

Chapter 1

Neural Networks and Deep Learning

In this chapter we give an introduction to an emerging branch of artificial intelligence which is called *Deep Learning*, and we focus in particular to how to build a *Deep Neural Network* (DNN). Despite this introduction being general, not application-focused, it is not intended to be complete: only the tools relevant for the subsequent chapters are here developed.
In the section 1.1 we explain foundations of neural networks, with the aim of showing how to build a simple DNN and how to train it on data.
In the section 1.2 we talk about some, in a certain sense more specific, aspects of constructing a DNN, which in turn really make the difference in making the algorithm effective. Indeed it turns out that in order to build up a DNN which actually performs well it is necessary to consider with care: the tuning of *hyperparameters*, that are the parameters which are not being optimized by the neural network, the problems arising from over/under-fitting of the data and the techniques used to deal with them, which go under the name of *regularization*, as well as different optimization algorithms, which can become very relevant in order to not remain stuck in a local minima.

1.1 Architecture and operation of a Deep Neural Network

The very fundamental unit which compose every neural network is the *node* or *neuron*, that is a structure that takes some input features, performs a specific transformation on them, and gives the output:

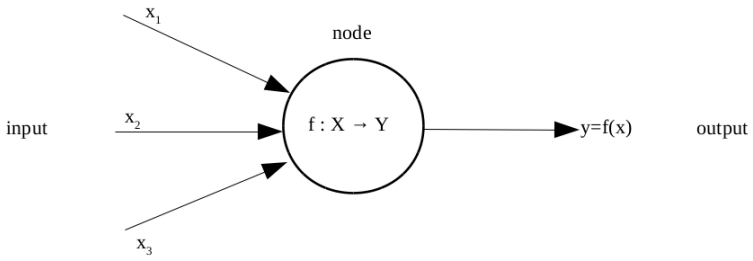

Figure 1: Node

To gain an intuition of what a node is really doing, it's useful to get a more precise idea of a possible problem we could want to face with our node and of an appropriate map $f : X \to Y$ we might want to use.

Example 1. Consider a given dataset $\{(x_i, y_i)\}$ like the one in the next picture, and suppose to have to find an appropriate relation in the form $y = f(x)$ in order to predict the value of the variable y, even for unknown x which we might encounter in the future.

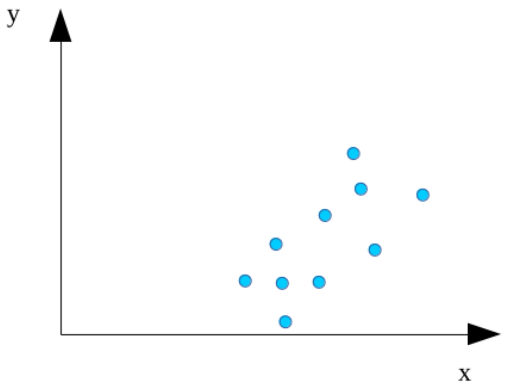

Figure 2

As widely known, linear regression can be a first good answer to the problem,

so, once performed the calculations, we are able to write:

$$y = wx + b$$

for some w, b; let's call this linear transformation L: $Lx = wx + b$. Now suppose that the output y has a constrain, e.g. that $y \geq 0$ must hold $\forall y$.[1] Then it would be reasonable to update the f function as in the figure below:

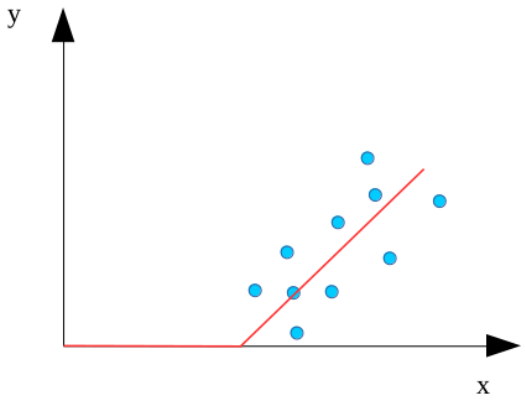

Figure 3

Which mathematically means:

$$y = max\{0, wx + b\}$$

So we have that the response function f is given by the composition of a linear function, L, and a non linear one, A *s.t.* $x \xrightarrow{A} max\{0, x\}$, where of course every function is defined from X to Y:

$$f : X \to Y \quad s.t. \quad f = A \circ L$$

In the deep learning literature the non linear function acting after the linear one is called *Activation function*, while the specific one used in this example, namely $x \to max\{0, x\}$, is known as *ReLU* function, which stands for *Rectified Linear Unit*. □

An approach like the one presented in example 1 unload all the complexity of a problem on the function f that maps the given data to the predicted output; it thus become early less feasible as complexity grows. It's here that the neural network paradigm come in.

[1] The reason why non linear function like the one arising from this constrain are needed in deep learning will be seen in the end of this section.

The main ideas behind it can be divided in two: a "divide et impera"-like approach and a statistical-like one, based on random functions and optimization of appropriate functionals.
For what concerns the first, the idea is to stuck together different nodes in order to be able to reconstruct complex behaviors as the composition of simpler ones. This assembly of nodes is performed in two fashions: from one side we can imagine to feed in the data to different neurons, which will calculate their own output, and then to put together the outputs, building up what in the literature is called a *layer* of nodes. From the other side we can put different layer in sequence, so that the first layer takes the input from the data, the second layer takes as input the output of the first one, and so on, until the last layer outputs the predicted y.[2]
Following the literature we call *Hidden layer* every intermediate layer, as in the following picture.

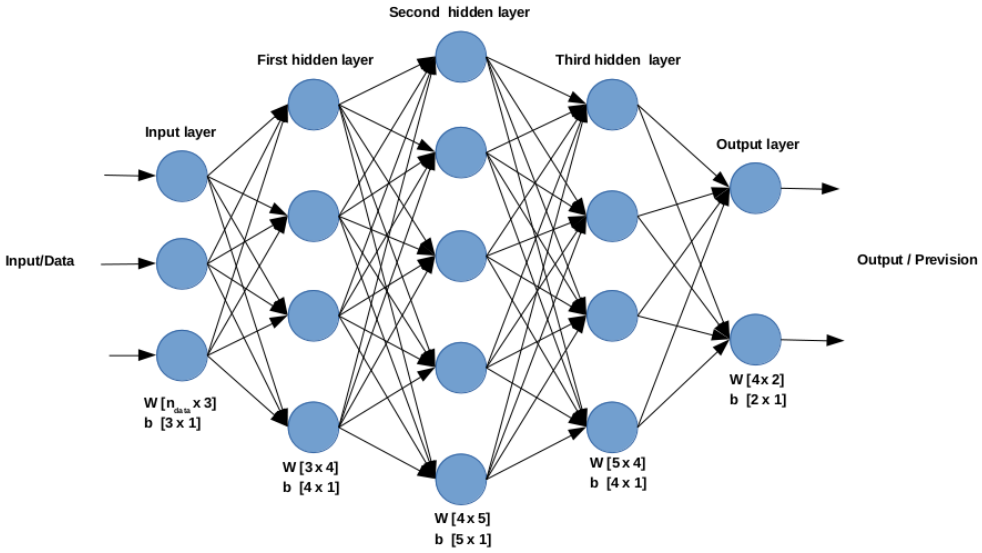

Figure 4: Neural network

In deep learning the above mentioned process that brings the input to the output, passing by all the intermediate layers, is called *forward propagation*, or, in short, forprop, and constitute the first part of the training algorithm of a neural network.

[2]Technically it's more correct to say "every node in the $n-th$ layer takes the input from every node in the $(n+1)-th$ layer", but it's preferred to use the short sentence above when it's clear enough.

One surprising aspect of neural networks is that they don't require their designer to decide which node of the first layer takes which part of the input, nor which nodes of the $n-th$ layer a generic node of the $(n+1)-th$ layer should consider, neither how much importance any node should give to each of its inputs. In fact, he gives all the inputs to every node, and lets the neural network find it out by itself. In other words, given enough training samples, i.e. enough (x_i, y_i) known couples, neural networks are remarkably good at figuring out how to map unknown x to the right y.
This capability is achieved by means of the second main idea of neural networks: what is called *back propagation*, or backprop.
Let's consider again the single node as in Figure 1, with $y = ReLU(wx + b)$. More precisely in this case we have:

$$L = \boldsymbol{w} \cdot \boldsymbol{x} + b \tag{1.1}$$

where $\boldsymbol{x} = (x_1, x_2, x_3)^T$, $\boldsymbol{w}$ is the vector of the *weights*, since the dot product in (1.1) can be seen as a weighted sum of the inputs, while b is called *bias*, since it affects this sum with a its contribution.
The basic idea is to perform a forward propagation with known data and randomly initialized parameters $\boldsymbol{w}, b$, then to calculate an appropriate distance between the predicted output, let's call it $\hat{y}$, and the known output$y$. This distance measure the *cost* C, or the loss, of our computation.
Let's consider for simplicity:

$$C = \frac{1}{2}(y - \hat{y})^2$$

Now we begin the backprop phase: we calculate the gradient of C w.r.t. our parameters, and then we update them using an optimization algorithm aiming at minimizing the cost C.
For example using gradient descent:

$$\begin{cases} \boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{dC}{d\boldsymbol{w}} \\ b \leftarrow b - \alpha \frac{dC}{db} \end{cases} \tag{1.2}$$

where α is a, typically small, positive real number, which controls how fast we update our weights, and is thus called *learning rate*.
The learning process is thus construct as a loop composed by forward propagation followed by backward propagation, with this sequence being repeated until the cost reaches a satisfying value.
Coming back to a full DNN, i.e. to a neural network with more than one hidden layer like the one in Figure 4, it's easy to generalize the above algorithm.

Let's suppose we have a large enough dataset of x_i with the corresponding y_j, and divided it in two disjoint subset, the *training set* and the *test set*.
We initialize all the weights randomly, and we start the loop, called *training loop*, over the training set.
First we have forward propagation:

- every node of the input layer takes all the data as input, performs an affinity like the one in (1.1) using its own weights and bias, non-linearizes the result through an activation function like the ReLU, and sends it's output to every node of the next layer;
- every node of the hidden layers takes as input all the outputs coming from the previous layer, performs a weighted sum of them and adds a bias, using its own parameters, and outputs the activated result to each node of the next layer;
- the nodes in the last layer usually calculate only the linear transformation and outputs the predicted result.[3]

At this point we calculate the value of the cost functional, and then we start backprop, in which every node, starting from the ones inside the last layer, calculate the gradients of the cost w.r.t. its own parameters w, b and updates them using an algorithm like gradient descent. Once we have updated all the parameters till the first layers we do another iteration of the algorithm.
When we are satisfied of the performance of our neural network, we do *only one* forward propagation step, keeping the optimal parameters, but using the test set, and we evaluate the accuracy of the results.

Before going on with the discussion of deeper aspects of deep learning, we show why some of the choices made in the design of the neural network and of the algorithm are appropriate.
First of all it is important to notice that non linear activation function are in general necessary. In fact, consider a general DNN, identify with the subscript l the quantities relative to the l-th layer, call n_d the number of the data in the set, n_l the number of nodes in the layer l and suppose that A is the identity for each node. We then have:

$$A_l = L_l = W_l A_{l-1} + b_l$$

where A_l denotes the $n_l \times n_d$ matrix of the outputs of the l-th layer, W_l is a $n_{l-1} \times n_l$ matrix, and b_l is a n_l vector. Then we have:

$$A_l = W_l(W_{l-1}A_{l-2} + b_{l-1}) + b_l = W_l W_{l-1} A_{l-2} + W_l b_{l-1} + b_l =: W_l' A_{l-2} + b_l'$$

[3]Unless a binary classification program is being run: in that case it's ok to activate the last layer output too. Further insights will be given later on.

By recursion is then easy to prove that in the end with a complete step of forward propagation we are computing only a linear combination of the initial data, as we would do in a much simpler way with just one node.
In this report we focus mainly on $tanh(\cdot)$ and ReLU activation functions, which are both very common choices, but others are being investigated as well.
For similar reasons it is very important to initialize the weights randomly: if we don't, and, for example, we initialize every parameter to a given value, then at the very first iteration all the nodes in the same layer are computing the very same output, and moreover are being updated at the same way during backward propagation. In the end this result in having a DNN that produces the same output of one which has just one node per layer.
Finally, it's worth noting that deep neural networks turn out to be more effective than bigger shallow neural networks in reconstructing and predicting complex behavior, especially when it is decomposable in a hierarchical grade of complexity, and that a minimum number of hidden layer is sometimes necessary. We don't prove it, but we will give a quantitative result of a comparison in section 2.2, despite in the specific case of reconstruction of continuous piecewise linear functions.

1.2 Further insights

When designing a neural network, a great number of choices have to be taken: the number of layers, of nodes, the value of the learning rate, the kind of activation function... Moreover what can be a good setting in one field usually isn't that good in another, making DNN programming a very iterative process.
Before getting into it, it's useful to split the data in three sets: the training set, containing the most of the data, the development set and the test set. Then, the first thing to do is to evaluate the performance of the network on the training set: if the error on it is high it means that the network is underfitting the data[4], and increasing the size of the network or training it longer could be possible tries. Once the error on the training set is low enough we look at the performance on the dev set. If it is low we've done, otherwise it means that in the previous stage we have overfitted the data[5], that is we have tuned the DNN so much on the training data that it has difficulties to generalize to new ones. In this case we can try to get more data and/ore to use regularization techniques, which are the topic of the next paragraph.

[4]This situation is known in the literature as "high bias".
[5]Also described as a "high variance" situation.

It's important to notice that once this modification are done, we've to start again from the beginning and check if the error on the training test is still low enough.
Once we have reached a reasonably good result on bot sets, we can finally verify the robustness of our network on the test set, which will give us an unbiased estimation, not having been used yet.
This iterative process is resumed in the following flowchart:

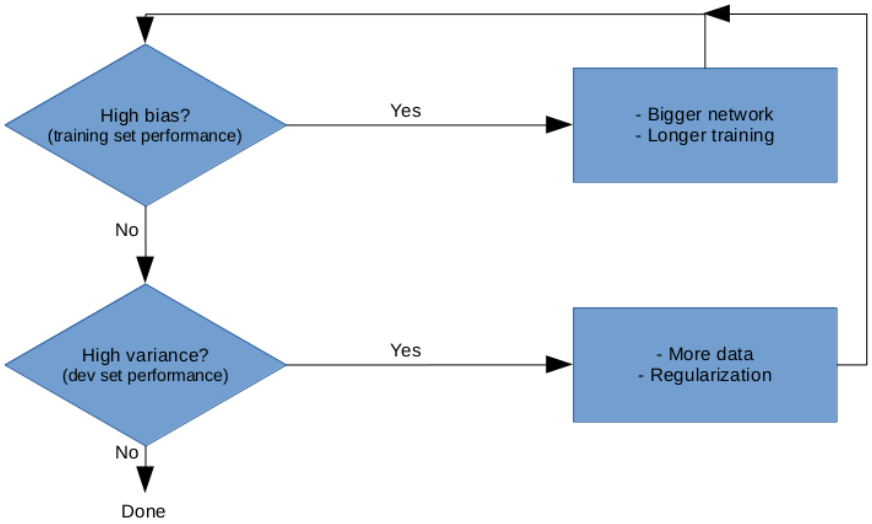

Figure 5

Another difficulty comes from the fact that in general the cost functional which has to be minimized is not convex, thus it may have local minima, and the space in which it lives, the one generated by the hyperparameters W, b becomes early high dimensional, enhancing the number of plateaus, and a simple optimization algorithm might stuck in both of them (and it usually do so). Some work can be done on the learning rate and on choice of the optimization algorithm, as it is shown in paragraph 1.2.2.

1.2.1 Regularization

There are many kind of regularization techniques, all aiming at reducing the overfitting on the training data. The most common is the L_2 *regularization*, which consist in adding a term containing the norms of the parameters to

the cost functional:

$$C \longrightarrow C + \frac{\lambda}{2n_L}\left(\sum_{l=1}^{L} ||W_l||_F + ||b_l||_2\right)^2$$

where $||\cdot||_F$ indicates the Frobenius norm, L the number of layers and λ the regularization parameter. L_2 regularization has the effect of keeping the parameters relatively small, and thus from one side it is somehow as if we had a smaller neural network, and from the other side, being:

$$A_l = W_l A_{l-1} + b_l$$

helps keeping the layers more linear, since the activation functions usually have a *tanh*-like form. In both cases L_2 regularization force the network to take simpler "decisions", reducing so the overfitting.
Another famous technique is *dropout*, which consist in selecting a different random fraction of the nodes at the beginning of every training iteration, and turning them off by removing the Figure 4 links. In this way if we look at a single iteration we are working exactly with a smaller network, and so we are reducing overfitting, but at the same time at every iteration we are using a different network, and so we are keeping the capability of the original bigger network to fit the complexity of the data.
Of course, while dropout may be useful to find weights able to generalize to data they've never seen, it is meaningful to use it only during the training - development phase of the process, while it hasn't to be used on the test set.

1.2.2 Optimization

A large number of famous algorithms in deep learning are based, among the others, on the generalization of the moving averages concept, that is here recalled.
Consider a set of data and, for simplicity, figure them as points in a 2D plane; suppose that their pattern follows somehow a certain curve, but in a noisy way. If we want to obtain that smooth curve we can't just take their average, otherwise we'll get a constant line, but we can imagine to cluster them and then to take averages. In order to recover a smooth function, consecutive clusters have to overlap. In practice we can proceed in this way[6]:

$$\begin{cases} m_0 = 0 \\ m_t = \beta m_{t-1} + (1-\beta)\theta_t, \qquad 0 \leq \beta \leq 1,\ t \geq 1 \end{cases} \tag{1.3}$$

[6]We use the notation that we will find when we'll introduce Adam as in [Kingma-Ba]. Here θ is the data.

Let's suppose we choose $\beta = 0.9$, then roughly speaking (1.3) computes at every step the average of the last 10 values, producing a quite smooth curve that fits the pattern of the data.

We now introduce four algorithms: gradient descent with momentum, RMSprop, Adam, which is based on the previous two and that from its presentation at International Conference on Learning Representations in 2015 has became increasingly popular, and its infinity-norm variant AdaMax.

Gradient descent with momentum is the direct application of (1.3) to standard gradient descent. For every iteration t:[7]

- compute dW, db
- update the momenta:

$$\begin{cases} m_{dW} \longleftarrow \beta m_{dW} + (1-\beta) dW \\ m_{db} \longleftarrow \beta m_{db} + (1-\beta) db \end{cases} \tag{1.4}$$

- update the parameters:

$$\begin{cases} W \longleftarrow W - \alpha m_{dW} \\ b \longleftarrow b - \alpha m_{db} \end{cases} \tag{1.5}$$

The effect of the moving averages on gradient descent is a reduction of the oscillations in the path towards the optimum value of W, b, which in turn results in a faster convergence: as oscillations go to zero, the whole step is taken in the right direction. In other words, imagine a 2D plane with some parameters that we are optimizing taken as orthogonal axis; then the situation before the addition of the momentum might be portrait as a triangle wave, instead performing averages the vertical oscillations cancels out themselves, while, if the algorithm converges to a point, the displacements to right sum up each other.

RMSprop pursue the same aim, namely fastening convergence of gradient descent when there is an oscillating like behavior, but in a slightly different way:

- $\forall t$ compute dW, db

[7] For ease of notation, in the following the t subscript is omitted and the gradient of the objective w.r.t. to a variable it's indicated as d(variable).

- update the momenta:

$$\begin{cases} s_{dW} \longleftarrow \beta s_{dW} + (1-\beta) dW^2 \\ s_{db} \longleftarrow \beta s_{db} + (1-\beta) db^2 \end{cases} \tag{1.6}$$

- update the parameters:

$$\begin{cases} W \longleftarrow W - \alpha \frac{dW}{\sqrt{s_{dW}}} \\ b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}} \end{cases} \tag{1.7}$$

where the superscript 2, indicating square operation, has to be intended element wise, and s stands for square.
To visualize the idea behind RMSprop let's suppose to have $dW >> db$; then in every gradient descent iteration it is as if we are taking a great step in the W direction, and a significantly smaller one in the b direction. But with RMSprop, if dW is big and db is small, being β usually small, we have in turn s_{dW} big and s_{db} small in (1.6), so in (1.7) we take comparable steps in both directions.

As anticipated, Adam basically puts together this two algorithms, and performs a *bias-correction* of the moving averages. The reference for this algorithm is the article by [Kingma-Ba], here we sketch it:

- gradient descent with momentum step:

$$\begin{cases} m_{dW} \longleftarrow \beta_1 m_{dW} + (1-\beta_1) dW \\ m_{db} \longleftarrow \beta_1 m_{db} + (1-\beta_1) db \end{cases}$$

- RMSprop step:

$$\begin{cases} v_{dW} \longleftarrow \beta_2 v_{dW} + (1-\beta_2) dW^2 \\ v_{db} \longleftarrow \beta_2 v_{db} + (1-\beta_2) db^2 \end{cases}$$

- momenta correction:

$$\begin{cases} m_{dW}^{corr} = \frac{m_{dW}}{1-\beta_1^t}, \ v_{dW}^{corr} = \frac{v_{dW}}{1-\beta_2^t} \\ m_{db}^{corr} = \frac{m_{db}}{1-\beta_1^t}, \ v_{db}^{corr} = \frac{v_{db}}{1-\beta_2^t} \end{cases}$$

- parameters update:

$$\begin{cases} W \longleftarrow W - \alpha \frac{m_{dW}}{\sqrt{v_{dW}} + \varepsilon} \\ b \longleftarrow b - \alpha \frac{m_{db}}{\sqrt{v_{db}} + \varepsilon} \end{cases}$$

where ε is a small positive number, used to prevent division by zero.

AdaMax is a variant of Adam, that can be obtained re-formulating the second step by means of the infinity norm. An interesting aspect of this algorithm is that it turns out that doing so we can avoid to correct for initialization bias, as needed in Adam[8]. The steps are:

- gradient descent with momentum step:

$$\begin{cases} m_{dW} \longleftarrow \beta_1 m_{dW} + (1-\beta_1) dW \\ m_{db} \longleftarrow \beta_1 m_{db} + (1-\beta_1) db \end{cases}$$

- infinity norm update:

$$\begin{cases} u_{dW} \longleftarrow max\{\beta_2 u_{dW}, |dW|\} \\ u_{db} \longleftarrow max\{\beta_2 u_{db}, |db|\} \end{cases}$$

- parameters update:

$$\begin{cases} W \longleftarrow W - \frac{\alpha}{1-\beta_1^t} \cdot \frac{m_{dW}}{\sqrt{u_{dW}}+\varepsilon} \\ b \longleftarrow b - \frac{\alpha}{1-\beta_1^t} \cdot \frac{m_{db}}{\sqrt{u_{db}}+\varepsilon} \end{cases}$$

Before leaving the optimization topic, it's worth mentioning the *learning rate decay*, a technique which involves the progressive reduction of α and that is often used along with the optimization algorithms.
Consider for simplicity standard gradient descent. What happens actually is that the steps are quite noisy in the taken direction, not exactly as the previous mentioned triangle wave. This doesn't give big problems at the beginning of the algorithms, but as we approach the minimum we might end up wandering for a technically not finite amount of time about it, but taking too big steps to really reach it. Reducing in an appropriate way the learning rate as the iterations increase in principle doesn't eliminate this problem (unless we exactly hit the minimum), but by sure lead us to wander in a tighter region around the minimum, which usually is a good enough result. Typical choices are:

- $\alpha = \frac{1}{1-d*t} \cdot \alpha_0$, d being a decay rate,
- $\alpha \propto \frac{1}{\sqrt{t}} \cdot \alpha_0$

[8]See [Kingma-Ba].

where in both cases t stands for the iteration number and α_0 is the initial learning rate, but also a simpler piecewise constant decay can make the difference.

In this chapter a significant number of hyperparameters has been explicitly introduced, each of which has to be properly tuned; moreover the list is not exhaustive, for example how to split the data into the three sets has to be decided too[9]. Fortunately not all the parameters plays the same role: a good idea might be to start from hyperparameters like the learning rate, the number of hidden units and of nodes per layer, while other parameters like for example the default Adam ones ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 1e-8$) are already good for most applications.

[9]This depends strongly on the total amount of data, e.g. it could range from $60\% - 20\% - 20\%$ respectively for train-dev-test sets when we have about 10^3 data, to $98\% - 1\% - 1\%$ when we have millions of data, basically because the dev and the test set just have to be big enough to evaluate the performance of the network, while the training set is the one on which the network really *learns* the optimal parameters. Moreover, when having such a big amount of data, it's better considering techniques based on mini-batches.

Chapter 2

Application to PDEs

The idea of applying machine learning to numerical analysis can be traced back up to [Lagaris et al.], who tried to solve ODEs and PDEs by means of neural networks. The authors have shown some of the potential of DNNs, such as their being excellent interpolators and the ease of parallelization of the deep learning algorithm; moreover they proposed a first comparison with finite element method and faced the problem one encounters when dealing with boundary value problems (BVPs) in a deep learning setting, which we will expose too. However they still used a mesh, which is a not necessary burden as we'll show, and they used a shallow neural network, with only one hidden layer, that, albeit theoretically able to recover any continuous function, is quite far from being optimal.
More recently the explosion of the Big Data field has given rise to stronger interests in deep learning, leading to the development of new tools which are being applied in the study of PDEs too with promising results; for example [Sirignano-Spiliopoulos] face the problem of dimensionality, solving PDEs in space up to 200 dimension using a mesh-free algorithm.
Anyway research in this field is still very young, most of the papers having 1-2 years old, and coherently there isn't a solid general theoretical framework as the one of FEM; nonetheless there are already some interesting results, like the one of [Jinchao Xu et al.].
The aim of the present chapter is to outline possible ways of proceeding for problems that one most often encounters in engineering and applied sciences, taking as example the classical Poisson-Dirichlet problem, highlighting pros and cons of the method too (section 2.1), and then to provide some useful theorems (section 2.2), which compare DNNs accuracy w.r.t. FEMs.

2.1 The Poisson-Dirichlet problem

There are different ways to employ deep learning to solve the Poisson-Dirichlet problem:

$$\begin{cases} -\Delta u = f & in\,\Omega \\ u_{|\partial\Omega} = g & on\,\partial\Omega \end{cases} \tag{2.1}$$

Recently [Kailai et al.] have obtained good results in few dimension with well-behaved manufactured solutions; their strategy is based on [Lagaris et al.] for the treatment of the boundary condition and on [Sirignano-Spiliopoulos] for the mesh; this method follows.
Having in mind what has been said in chapter 1, the first thing to do is to generate the datasets. Concerning this, there are at least two important things to notice:

- the points can be sampled randomly in $\overline{\Omega}$, so to wipe out the need of a mesh, guaranteeing in this way large computational savings, especially in high dimensions;
- the probability that points given by a random generator lie on $\partial\Omega$ is technically zero, since $|\partial\Omega|/|\Omega| = 0$, so it has to be ensured that the network is trained on the boundary too.

The authors splits the problem in two similar one, the first having a training set made only by boundary points, the second only by interior points;[1] in both cases they use equation (2.1) RHS to calculate f, g at the random generated points. Then they build up two neural networks, specifically with 3 layers and 64×3 nodes, $tanh(\cdot)$ activation function and Adam optimizer, to reconstruct the solution u as:

$$u(x, y; w_1, w_2) = A(x, y; w_1) + B(x, y) \cdot N(x, y; w_2) \tag{2.2}$$

where A is the output of the neural network approximating the boundary condition, N the one approximating the solution in the domain, and B a function that goes to zero on the boundary; consequently their cost functional has the form:

$$\sum_{i=1}^{m} ((g_D)_i - u(x_i, y_i))^2 + \sum_{i=1}^{n} (f(x_i, y_i) - Lu(x_i, y_i))^2 \tag{2.3}$$

[1] The fundamental aspect is to have "enough" boundary points. A conceptually simpler solution may be to force the random generator to output an appropriate fraction of boundary points over the total generated, and then to use only one network training on the whole set.

L being the differential operator in strong form.
More precisely their algorithm has two nested loops: for every iteration of the interior network, they perform a full training of the boundary network. Using a $sin(\cdot)sin(\cdot)$ manufactured solution on the unit square with their code, freely available on Github, is possible to recover a very good solution in few hundreds of iterations[2]:

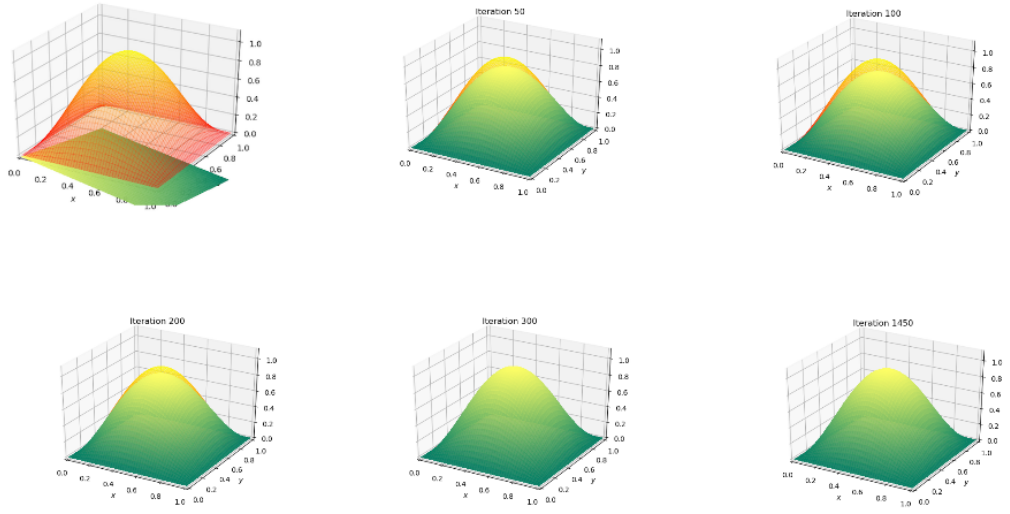

Figure 6

However their algorithm does not perform equally good nor in higher dimension neither with ill-behaved solutions, and further developments are needed.[3]

A very different approach, but again based on deep learning, has been proposed by [Weinan-Bing]. Here it is reported in the case regarding the problem under analysis, in order to show both the largeness of the range of possible DNN based methods for solving PDEs and their common traits, and to point out a very interesting link between this approach, weak formulations and optimal control theory.
In their article the homogeneous Poisson-Dirichlet problem is re-casted analytically in the corresponding optimal control problem, leading to a cost functional of the form:

$$\int_{\Omega} \left(\frac{1}{2} |\nabla u|^2 - uf \right) dx + \beta \int_{\partial\Omega} u^2 d\sigma \tag{2.4}$$

[2]Code and report are here: [Kailai et al.]; notice that it requires [Python] 2.7 and [TensorFlow]. Figure 6 can be found at the same site.
[3]Further insights and possible reasons are proposed in the next section.

which, besides the positive constant β and considering homogeneous Dirichlet conditions, turns out to be the exactly the weak form of the [Kailai et al.] cost functional.
The method then proceeds minimizing the cost functional (2.4) over the set of admissible functions U_{ad}, and is here that deep learning comes in.
In fact functions in U_{ad} are re-constructed through neural networks, so moving from an additive-like construction proper of standard approximation theory to a compositional-based one.
At this point the authors apply a quadrature formula to the integral in (2.4) and then solve the final optimization problem with a proper optimization algorithm.

The algorithm produces good results but, as other approaches based on DNN, has a drawback side: even when the initial problem is convex, the variational problem which is then obtained isn't so, inheriting thus the problem of local minima, plateau and saddle points proper of deep learning.

2.2 Some more in-depth theoretical results

In this paragraph some results, obtained mainly by [Jinchao Xu et al.], are presented, with the aim of giving estimates of the optimal size for a DNN employed to solve PDEs; in particular the results found are useful to ensure that the DNN recovers the same accuracy of the finite element method based on linear base functions.
We recall some FEM notation we use in the following: suppose to have a bounded domain[4] $\Omega \in \mathbb{R}^d$ ad a conforming mesh or grid $\mathcal{T}_h \subset \Omega$, define moreover k_h as the maximum number of mesh elements neighboring a grid point; then the FE space of grade one is:

$$V_h = \{v \in C_\Omega \text{ s.t. } v \text{ is linear on every } \tau_k \in \mathcal{T}_h\}$$

In other words, the base functions of the grade one FEM are continuous piecewise linear functions, in short CPWL; it is thus natural to proceed into the study using ReLU-DNNs, since the ReLU activation function is a CPWL function and DNNs construct their output as a composition of linear functions and activation functions, giving so a CPWL function as output.

[4] In the sense of open and connected.

The two main results are[5]:

Theorem 1. *Given a locally convex finite element grid $\mathcal{T}_h$, any linear finite element function in $\mathbb{R}^d$ with N degrees of freedom can be written as a ReLU-DNN with at most $\lceil log_2(k_h) \rceil + 1$ hidden layers and at most $\mathcal{O}(k_h N)$ neurons.*

Theorem 2. *Given a locally convex finite element grid $\mathcal{T}_h$, any linear finite element function in $\mathbb{R}^d$ with N degrees of freedom can be written as a ReLU-DNN with at most $\lceil log_2(d+1) \rceil$ hidden layers and at most $\mathcal{O}(d 2^{(d+1)k_h} N)$ neurons.*

A direct comparison of the two theorems shows that, although it is possible to achieve linear FEM like accuracy with a relatively shallow network, the deeper one has a considerably smaller size.
To fix the ideas it has been considered $d = 2$ and Ω being the unit circle. Then a standard triangulation of the domain is:

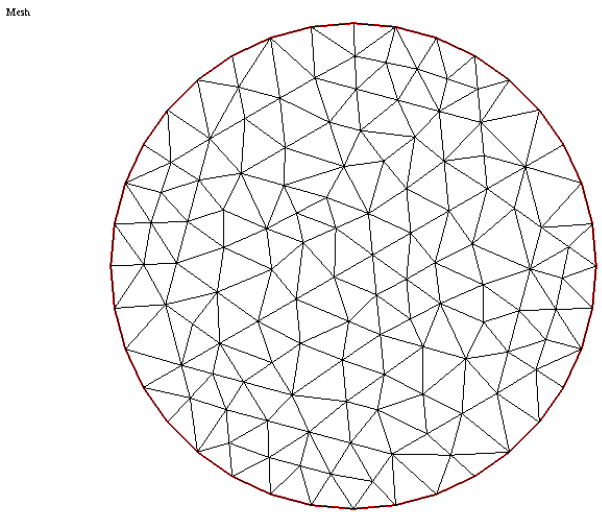

Figure 7: Mesh

When doing an estimate it's thus possible to take $k_h = 6$.
To find a proper value for N we consider the very well-behaved solution:

$$u_{exact} = e^{-(x^2+y^2)}(x^2 + y^2 - 1)$$

of the problem (2.1) obtained computing f as Δu_{exact}; we then solve the problem different times using the FOSS software [FreeFem++] refining the

[5]These are taken respectively from Theorem 3.1 and Corollary 5.1 of [Jinchao Xu et al.].

mesh until a good result is reached.[6].
We obtain a relative L_2 error, defined as $err_{L_2} = \frac{||u_{exact} - u_h||_2}{\frac{1}{2}(||u_{exact}||_2 + ||u_h||_2)}$, of 1.65% when using 36 triangles on the border of the domain, as in Figure 7, which lead to a V_h space with 137 degrees of freedom:

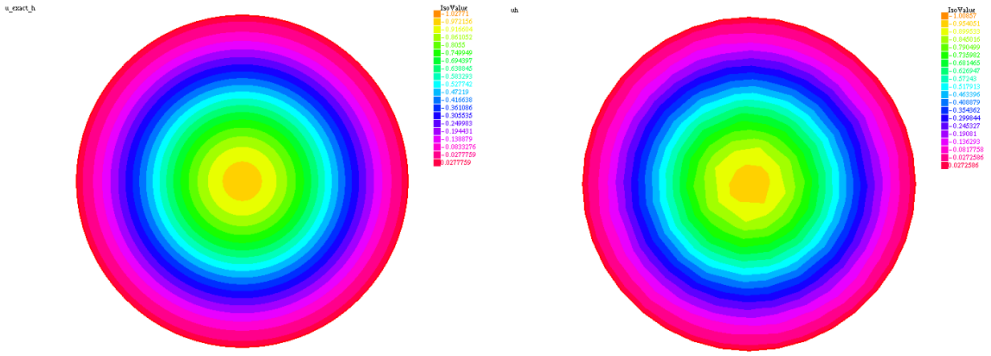

Figure 8

In the end it is possible to conclude that to recover the same result using a ReLU-DNN *at most*:

- $\lceil log_2(k_h) \rceil + 1 = 4$ layers and $\mathcal{O}(k_h N) \simeq 10^3$ total nodes, or
- $\lceil log_2(d+1) \rceil = 2$ layers and $\mathcal{O}(d 2^{(d+1)k_h} N) \simeq 10^8$ total nodes

are needed, confirming that a deep neural network is computational more convenient that a shallow one[7] and, perhaps more importantly, giving a quantitative estimate of this convenience.

[6]Code in appendix.

[7]This is coherent with the fact that the power of the neural networks approach stands in their remarkable capability of approximate non linear functions, which is achieved by means of composition between affinities and non linear activation functions, and the step of activation occurs in the forward propagation as many times as the number of layers.

Chapter 3

Implementation of neural-net

In this chapter an in-depth view of a Deep Learning C++ program, called *neural-net*, is given. As just shown in chapter 2, what different DNN based PDE solvers have in common is their relying on what turns out to be DNN's numerical analysis strongest point: their remarkably capability of reconstructing function, by means of compositions, which is used to construct the basis of the approximation spaces. For this reason it has been decided to start writing an interpolator based on DNNs; different possible expansions are being investigated as well.
The program has been built using the [Git] distributed version control system, with GitHub as hosting service, and can be freely cloned or downloaded from the web page: *https://github.com/pjbaioni/neural-net*.[1]
In order to build and run the program, some not included dependencies are needed, in particular a C++ compiler, [Make], [gnuplot], [Eigen] and [Boost] libraries, while to build the documentation a TeX compiler has to be used; more details are given in the README.md file and in chapter 4, where it is shown how to obtain, build and run the program.

[1] At the moment, already merged branches have not been deleted yet for personal ease in history navigation, e.g. when typing `$ git log --pretty=format:"%h %s" --graph`, but the master branch is the only one which is really active and updated.

3.1 Directory tree

Downloading the repository "neural-net" one finds:

- the *data* folder, which contains the input and output data in .dat and .pot (for [GetPot]) format, and [gnuplot] scripts in .gnu format to generate datasets graphs;
- the *doc* folder, containing this documentation in .tex and .pdf format, plus the *img* sub-folder where the images included by the TeX file are stored;
- the *include* folder, containing the headers *GetPot.hpp*, an utility used for parsing parameters files and command line options, *gnuplot-iostream .hpp*, an utility used to output a graphical representation of the results in an interactive way, *NeuralNetwork.hpp*, which holds the NeuralNetwork class, and *Optimizers.hpp*, where all the optimizers employable from NeuralNetwork are defined as class templates;
- the *src* folder, which contain the source files *main.cpp*, *NeuralNetwork.cpp*, where the NeuralNetwork class member functions are defined, the *Makefile* which can be used to automatic build the main program, and the *write_set* sub-folder, where the *write_set.cpp* file used to generate datasets is placed;
- the COPYING file, a plain text file containing copying informations and licenses;
- the README.md file, a markdown file containing basic informations;
- the (hidden) *.gitignore* file, that traces the file extensions which are not being pushed to the remote, mainly compilation files, executables and comments file.

Despite their different extensions, every non-png nor non-pdf file can be opened by any text editor.

3.2 The Neural Network class

The class NeuralNetwork implements a computational efficient design of a deep neural network (DNN).
As seen in chapter 1, to which we refer for the names of the principal quantities as for the justification of the main algorithms, a DNN is composed by

different layers, each of which is composed by a (possibly) different number of nodes. At the beginning of the present project, this hierarchical structure naturally inherent to DNNs led to the idea of developing three classes, NeuralNetwork, Layer and Node. For reasons that will appear clearer later on[2], Layer would have been an abstract class, from witch the child classes InputLayer, HiddenLayer and OutputLayer would have inherited. That approach doubtless had the pro of adhering to the abstract model of a DNN, from the more general structure to the really low-level computations, but initial tests have shown that it wasn't so competitive from a computational efficiency point of view, besides being more complex than strictly necessary.
The final choice has been to "vectorize" the computations as much as possible, grouping together all the operations performed by nodes in each layer by means of matrices. Of course this passage, which can be considered computationally mandatory in machine learning typical languages such as Python, hasn't brought efficiency by itself, being C++ a compiled language: the difference lies in the fact that it has allowed to rely on strongly optimized well known linear algebra libraries, which have provided considerable computational time savings. In particular the choice is fallen on the [Eigen] template library, for its immediate compatibility with C++ and its ease of use and linking with respect to other classical libraries such as Lapack, while maintaining a very good performance. Having reached this point, and having lost a somehow eye-apparent sight on each specific node operations, it has appeared senseless to keep the original design of the neural network class, and thus it has been decided to simplify the code, by developing just one class which holds vectors of the quantities that used to characterize every layer in the original design.
The choices made have thus led to a structure as the following one:

```
1  class NeuralNetwork{
2
3  private:
4
5    //Architecture hyperparameters:
6    size_t nlayers;
7    VectorXs nnodes;
8
9    //Parameters to be optimized:
10   vector<MatrixXd> W;
11   vector<VectorXd> b;
12
13   //Forward propagation outputs:
14   vector<MatrixXd> L;
```

[2]Basically, because first, internal and last layer performs slightly different works.

```
15    vector<MatrixXd> A;
16
17    //Back propagation gradients:
18    vector<MatrixXd> B;
19    vector<MatrixXd> dW;
20    vector<VectorXd> db;
21
22    //Optimizers:
23    vector<shared_ptr<GradientDescent<MatrixXd>>> W_optimizer;
24    vector<shared_ptr<GradientDescent<VectorXd>>> b_optimizer;
25
26  public:
27
28    //Constructor:
29    NeuralNetwork(const VectorXs &);
30
31    //Training function:
32    void train(const Eigen::MatrixXd & Data, double alpha, std
          ::size_t niter, const std::size_t W_opt, const std::
          size_t b_opt, double tolerance=-1., const std::size_t
          nrefinements=1, const bool verbose=false);
33
34    //Test function:
35    pair<VectorXd,double> test(const MatrixXd & Data);
36
37  };
```

Note that here and in most of the following codes all preprocessor's directives as includes and header guards, as well as scope operators, typedefs and comments have been omitted for brevity.[3]
So the class NeuralNetwork holds basically four kind of data members:

(i) architecture (hyper)parameters, like `nlayers` and `nnodes`, which specify respectively the total number of layers and the number of nodes for each layer, `VectorXs` being a `typedef` for an Eigen dynamic vector holding `std::size_t`s, miming the one of `Eigen::VectorXd`;

(ii) the parameters to be optimized, namely `W` and `b`; notice that the latter has to be a vector of vectors, and not a matrix, since the number of nodes varies for each layer;

(iii) the *propagation members* `L`, `A`, `B`, `dW` and `db`, that hold the data computed at every cycle of forward and backward propagation;

[3]See NeuralNetwork.hpp for the complete version.

(iv) the optimizers for `W` and `b`, which hold a (smart) pointer to the base class of the Optimizers class hierarchy.[4]

For what concerns the (i) group, `nnodes` is the *true* parameter, indeed it is the only argument that the constructor takes, while `nlayers` is stored for convenience, since it is used very often in for loops in the member functions, but could be removed. In general, it has been chosen to use `std::size_t` every time an unsigned was needed, to adopt a standard that would fit every necessity.
As explained above, `W` and `b` are the vectorized version of chapter 1 parameters: if we call `l` the generic layer, then `W[l]` is an `nnodes(l)`×`nnodes(l+1)` matrix, while `b[l]` is a `nnodes(l+1)` column vector, as in Figure 4.
A similar comment holds for the propagation members (iii), with two observations:

- forward members, `L` and `A`, have been chosen to be the most general possible, even if for the problems for which the program has been designed the first one and the last one will be always column vectors, and not true matrices;
- backward members, `B` (which stands for "backward"), `dW` and `db`, aren't strictly needed, like the previous `nlayers`, but simplify significantly the subsequent code.

The (iv) group has been introduced later, indeed simple algorithms like Gradient Descent can be applied directly during the training phase, i.e. can be written directly by hand in the `training()` function. However it turns out that most finer optimizers need to store estimators of the gradients, and this estimators have to be updated at every training iteration, thus the idea of storing them in the neural net.
Since, as it will be seen in the following section, optimizers have been implemented applying inheritance, here we store smart pointers to the base class, instead of the proper optimizer itself, in order to be able to use polymorphism later and to let the user choose the desired optimizer runtime, when calling the training function.

[4]Optimizers.hpp is illustrated in section 3.3

Having introduced the data members, we consider now the member functions. The constructor initialize every member, in order of declaration:

```
NeuralNetwork(const VectorXs & nn): nlayers(nn.rows()),nnodes
    (nn){

  W.reserve(nlayers-1);
  b.reserve(nlayers-1);
  for(size_t l=0; l<nlayers-1; ++l){
    W.emplace_back(MatrixXd::Random(nnodes(l),nnodes(l+1)));
    b.emplace_back(VectorXd(nnodes(l+1)));
  }

  L.reserve(nlayers);
  A.reserve(nlayers-1);

  B.reserve(nlayers-1);
  dW=W;
  db=b;

  W_optimizer.reserve(nlayers-1);
  b_optimizer.reserve(nlayers-1);
}
```

In all methods whenever possible efficiency has been considered: for example here `nlayers` and `nnodes` are initialized in the member initializer list, and `std::vector<T>::reserve()` followed by `std::vector<T>::emplace_back()` is preferred over `std::vector<T>::push_back()`.
As already explained `W` has to be initialized randomly; for this purpose it has been chosen to use the Eigen `Random()` function, which returns a double uniformly distributed between zero and one. Of course different choices are possible, for example the ones based on the `<random>` header of the standard library. In particular, to be strictly random, as requested by the chapter 1 argumentation, we should use the latter and guarantee effective randomness including "real entropy" in the seed, for example in this way:

```
size_t sd = chrono::system_clock::now().time_since_epoch().
    count();
default_random_engine gen(sd);
uniform_distribution<double> random_value(0,1);
for(size_t i=0; i<W.rows(); ++i)
  for(size_t j=0; j<W.cols(); ++j)
    W(i,j) = random_value(gen);
```

However, the Eigen default pseudo-random function has shown very good performances, so it has been preferred due to its simplicity.

It's worth noting that only `W`, `b` and their gradients can be fully initialized, since it has been chosen to take the input data (whatever it is, training or test data, chosen optimizer...) only when it is really used. Consequently variables like the number of data, which is one of the dimension of the `A[l]`, `L[l]` matrices, is not known at this moment.
In every case the needed space in the vectors is known from `nlayers`, and thus is pre-allocated; only the strictly needed space is allocated, often only `nlayers-1`, an action that is possible due to the choices made in the following `train()` member function (here decomposed in different frames for clarity):

```
1  void NeuralNetwork::train(const MatrixXd & Data, double alpha
       , size_t niter, const size_t W_opt, const size_t b_opt,
       double tolerance, const size_t nrefinements, const bool
       verbose){
2    ////////////////////
3    //       Init     //
4    ////////////////////
5
6    size_t ndata=Data.rows();
7
8    //L has nlayers components:
9    for(size_t l=0; l<nlayers;++l)
10     L.emplace_back(ndata,nnodes(l));
11   //A hasn't the last one:
12   for(size_t l=0; l<nlayers-1;++l)
13     A.emplace_back(ndata,nnodes(l));
14   //B hasn't the first one:
15   for(size_t l=1; l<nlayers;++l)
16     B.emplace_back(ndata,nnodes(l));
17
18   //First layer only reads the input:
19   L[0]=Data.col(0);
20   A[0]=L[0];
```

Since the input datasets have to be read somehow, and since the "layer structure" was already ready, the first layer has been reserved for reading only. This is of course a personal choice, motivated basically from the fact that it's easy and works good. A more elaborate one could have been to let the first layer weight the (x, y) couples received during training from the dataset, and thus having e.g. `W[0]` an `ndata x nnodes[0]` matrix, but, being `ndata` quite large, this introduces by sure some overhead, which would be unjustified in this case.
Since in the present version the first layer only reads the input, it don't need weights, biases and gradients at all, and for this reason their vectors have one component less.

A different discussion must be made with regards to the activated outputs variable `A`, which again has just `nlayers-1` elements, but for another reason, less subjective: since we want our output to be a function, in the sense of mathematical analysis, with no particular requirements, we cannot in general activate the last output, because if we would do so for example with a $tanh(\cdot)$, we would be able to reconstruct only $tanh(\cdot)$s.
The "Init" phase of the `train()` function finishes with the initialization of the optimizers, which is made through a switch-case construct:

```
//Optimizers:
string W_opt_name, b_opt_name;
switch(W_opt){
  case 0:
    for(size_t l=0; l<nlayers-1; ++l)
      W_optimizer.emplace_back(make_shared<GradientDescent<
          MatrixXd>>(dW[l].rows(),dW[l].cols()));
    W_opt_name = "GradientDescent";
    break;
  case 1:
    for(size_t l=0; l<nlayers-1; ++l)
      W_optimizer.emplace_back(make_shared<GDwithMomentum<
          MatrixXd>>(dW[l].rows(),dW[l].cols()));
    W_opt_name = "GDwithMomentum";
    break;
  case 2:
    for(size_t l=0; l<nlayers-1; ++l)
      W_optimizer.emplace_back(make_shared<RMSprop<MatrixXd
          >>(dW[l].rows(),dW[l].cols()));
    W_opt_name = "RMSprop";
    break;
  case 3:
    for(size_t l=0; l<nlayers-1; ++l)
      W_optimizer.emplace_back(make_shared<Adam<MatrixXd>>(dW
          [l].rows(),dW[l].cols()));
    W_opt_name = "Adam";
    break;
  default:
    for(size_t l=0; l<nlayers-1; ++l)
      W_optimizer.emplace_back(make_shared<AdaMax<MatrixXd>>(
          dW[l].rows(),dW[l].cols()));
    W_opt_name = "AdaMax";
    break;
}

switch(b_opt){
  case 0:
    for(size_t l=0; l<nlayers-1; ++l)
```

```
34          b_optimizer.emplace_back(make_shared<GradientDescent<
                VectorXd>>(db[l].rows(),db[l].cols()));
35        b_opt_name = "GradientDescent";
36        break;
37      case 1:
38        for(size_t l=0; l<nlayers-1; ++l)
39          b_optimizer.emplace_back(make_shared<GDwithMomentum<
                VectorXd>>(db[l].rows(),db[l].cols()));
40        b_opt_name = "GDwithMomentum";
41        break;
42      case 2:
43        for(size_t l=0; l<nlayers-1; ++l)
44          b_optimizer.emplace_back(make_shared<RMSprop<VectorXd
                >>(db[l].rows(),db[l].cols()));
45        b_opt_name = "RMSprop";
46        break;
47      default:
48        for(size_t l=0; l<nlayers-1; ++l)
49          b_optimizer.emplace_back(make_shared<Adam<VectorXd>>(db
                [l].rows(),db[l].cols()));
50        b_opt_name = "Adam";
51      break;
52      case 4:
53        for(size_t l=0; l<nlayers-1; ++l)
54          b_optimizer.emplace_back(make_shared<AdaMax<VectorXd>>(
                db[l].rows(),db[l].cols()));
55        b_opt_name = "AdaMax";
56        break;
57  }
```

Notwithstanding the possibility to choose different optimizer for `W` and `b` was foreseen only to investigate the relative importance of the two parameters, preliminary tests done trying to predict a wavepacket-like function has shown that the best compromise between accuracy and number of iteration is reached when using AdaMax for `W` and Adam for `b`, so these have become the default.
The `train()` function implements a piecewise constant learning rate decay, obtained in this way:

```
1  ////////////////////
2  //Training loops //
3  ////////////////////
4  double old_cost{numeric_limits<double>::infinity()};
5  double cost{-1.}, err{old_cost};
6  niter=niter/nrefinements;
7  vector<size_t> backup_t(nrefinements);
8
```

```
9  for(size_t ref=1; ref<=nrefinements; ++ref){
10   for(size_t t=1; t<=niter; ++t){
11     //do one forward propagation
12     //eventually output the cost
13     //do one backprop
14   }
15 //update refinement parameters for the next training loop:
16 alpha=alpha/10;
17 tolerance=tolerance/((10-2*ref)*10);
18 }
```

Learning rate decay has been implemented because it turned out to be very important to obtain acceptable accuracy. This implementation has the pros of being automatic, very simple and effective enough, but, since everything in DNN programming is very problem dependent and has to be tuned properly case by case, better results can be obtained miming this procedure directly in the main, when training the net. Indeed it is enough to don't pass the number of refinements as argument to turn off the automatic learning rate decay, and then implement it manually with subsequent calls to the training function. Guidelines to do it are presented in chapter 4.
The forward propagation step that takes place in the above inner loop follows the general theory:

```
1  ////////////////////////
2  // Forward propagation //
3  ////////////////////////
4  for(size_t l=1; l<nlayers-1; ++l){
5    //Summing the column vector b (nnodes(l+1)x1) to each
6    //column of A*W (ndataxnnodes(l)*nnodes(l)xnnodes(l+1)):
7    L[l] = ( A[l-1]*W[l-1] ).rowwise() + b[l-1].transpose();
8    A[l] = tanh( L[l].array() );
9  }
10 //The final output shouldn't be activated, otherwise it will
     be necessarly a tanh:
11 L[nlayers-1] = ( A[nlayers-2]*W[nlayers-2] ).rowwise() + b[
     nlayers-2].transpose();
12
13 //Computing cost as the L2 distance: (divided by 2, for ease
     in later differentiation)
14 cost = .5 * (L[nlayers-1] - Data.col(1)).array().square().
     matrix().sum();
```

As evident, the code strongly relies on the Eigen built-in operations and function; in particular at line 7 and 11 of the above code a Python-like *broadcasting* is implemented. Another great advantage of using Eigen is the ease of switching between matrix like operations, like the products in the

same lines of code, to element wise operations, which are the ones invoked with a `.array()`. Moreover, when optimization is activated, e.g. with the `g++ -O3` option, conversions from matrices to array, as well as transpositions and so on, are implemented in such a way that they are costless from a computational point of view.
Then a convergence check and, eventually, an output to the standard output is performed:

```
//////////////////////////
//  Convergence check  //
//////////////////////////
//Output the current cost
//and check if convergence is reached:
if( t%100==0 ){
  if(verbose)
    cout<<"t="<<t<<" cost="<<cost<<" W_opt="<<W_opt_name<<"
       b_opt="<<b_opt_name<<" alpha="<<alpha<<"\n";
  err = abs(old_cost-cost) / ( (cost+old_cost)/2 );
  if(err<tolerance){
    backup_t[nref-1]=t;
    break;
  }
  else
    old_cost = cost;
}
```

Just after that the backward propagation phase begins. In the initial stages of the project, the backprop algorithm was designed following strictly the standard theory, with vectors of gradients starting from the right of the net and going to the left, e.g. `dW[0]` held the gradient of the cost functional w.r.t. the weights of the last layer.[5] That designed showed soon some inconveniences: in the update phase we had e.g. `dW[l]` and `W[l]` referring to different layers, and an ad hoc construction for the gradients was needed too, instead of the simpler `dW=W`; for this reason it has been decided soon to "overturn" the backprop section while keeping its natural sequence, simply reverting its for loop condition. That's the final code:

```
  //////////////////////////
  // Backward propagation //
  //////////////////////////

  //Compute B as d(cost)/d(output):
  //(no tanh now because it's the last layer)
  B[nlayers-2] = L[nlayers-1] - Data.col(1);
```

[5] See commit 6432f1d517205f3920ab1b77bf224735fbeec819

```
8
9   for(size_t l=nlayers-2; l>0; --l){
10    //Compute gradient of cost wrt W:
11    dW[l] = ( L[l].transpose() )*B[l];
12    //Update W:
13    //W[l] = W[l] - alpha*dW[l];
14    (*W_optimizer[l])(W[l],dW[l],alpha,t);
15    //Compute gradient of cost wrt b:
16    db[l] = B[l].transpose().rowwise().sum();
17    //Update b:
18    //b[l] = b[l] - alpha*db[l];
19    (*b_optimizer[l])(b[l],db[l],alpha,t);
20    //Compute previous B:
21    //(now there is tanh, and dx[tanh(x)]=1-x^2)
22    B[l-1] = (1. - (A[l].array().square())) * ( (B[l]* (W[l].
         transpose()) ).array() );
23  }
24  //Updating parameters of the first hidden layer:
25  dW[0] = ( L[0].transpose() )*B[0];
26  W[0] = W[0] - alpha*dW[0];
27  db[0] = B[0].transpose().rowwise().sum();
28  b[0] = b[0] - alpha*db[0];
29 }//End of the training loop
```

As can bee seen, `B[l]` is a support variable, that holds the derivative of the cost w.r.t to the output of the layer `l`. This is useful since the gradients of the cost w.r.t the weights and the biases are computed by means of the chain rule:

$$\begin{cases} B[l] := \frac{dC}{dY[l]} \\ \frac{dC}{dW[l]} = \frac{dC}{dY[l]} \cdot \frac{dY[l]}{dW[l]} \\ \frac{dC}{db[l]} = \frac{dC}{dY[l]} \cdot \frac{dY[l]}{db[l]} \end{cases}$$

with *Y[l]* being equal to $L[l] = A[l-1] \times W[l] + b[l]$ for the last layer and to $A[l] = tanh(L[l])$ for the others.
The above code implements exactly this chain rule computations, taking into account that the activation function is a $tanh(\cdot)$, and performs also the subsequent update of the parameters, based on their gradients. At the beginning only gradient descent was implemented, directly into the code; that lines have been kept commented for documentation purposes, since they show what the call to the optimizer roughly speaking does.[6]

[6]Calling the overloaded call operator isn't very elegant anymore, but it was designed when polymorphism wasn't implemented yet, so one had something like `W_optimizer[l](W[l],dW[l],alpha,t)`, but then polymorphism required to store pointers in the optimizers vectors. Of course one can always do `W_optimizer[l]->()(W[l],dW[l],alpha,t)`.

The training function ends with the refinement and the final output:

```
  //update refinement parameters for the next training loop:
  alpha=alpha/10;
  tolerance=tolerance/((10-2*ref)*10);
  }//End of the refinements loop

  ////////////////////////
  //     Final output     //
  ////////////////////////
  size_t total_iter=0;
  for(const auto & r: backup_t)
    r == 0 ? total_iter += niter : total_iter += r;
  cout<<"Total iterations = "<<total_iter<<"\n"<<"Cost
     functional on the training set = "<<cost<<endl;

}//End of the train function
```

where a range-based for loop is used to compute the total number of iterations, since the counter is reset at every refinement. Of course one could have distinguished the cases with refinements from the ones which haven't, for example so to not defining the vector `backup_t` when not required, but it has been preferred to not do so, in order to avoid a further decision tree in the code. Differently the last check, based on the conditional operator, is needed: it distinguishes between the case in which the inner training loop ends for having reached convergence from the case in which the maximum number of iterations is hit.

The `test()` function performs just one forward propagation, with two main differences w.r.t. `train()`:

- it doesn't store nor data neither intermediate results, since it wouldn't re-use them, to save memory;
- it is a non-void function, in part due to the above fact, but mainly because one expects to have an output from the test.

```
pair<VectorXd,double> test(const MatrixXd & Data){
  //One forprop as before during training,
  //w/o storing anything but the final result:
  MatrixXd Ltest{Data.col(0)};
  MatrixXd Atest{Ltest};
  for(size_t l=1; l<nlayers-1; ++l){
    Ltest = ( Atest*W[l-1] ).rowwise() + b[l-1].transpose();
    Atest = tanh( Ltest.array() );
  }
```

```
10    Ltest = ( Atest*W[nlayers-2] ).rowwise() + b[nlayers-2].
          transpose();
11
12    //Computation of the L2 relative error:
13    double numerator=(Data.col(1)-Ltest).norm();
14    double denominator=(Data.col(1).norm() + Ltest.norm())/2.;
15
16    return make_pair(Ltest, (numerator/denominator) );
17  }
```

One could ask why don't use the members `A, L` anyway, so to don't even have to allocate the two matrices. In principle it could be done actually, but it would require more coding: `A[l], L[l]` have to be resized for every `l`, since in general the numerosity of the training set is different from the one of the tests set; instead the above approach has been preferred in order to keep the code as simple as possible.

Before moving to the optimizers classes, we consider some final remarks.
The code can be easily extended to a n-D interpolator with no conceptual effort. The very simplest way to do so would be to equip also the first layer with a `W`[7], basically to do a (weighted) average of every input point $\boldsymbol{x} = (x_1, x_2, x_3 ... x_n)$. Indeed, doing so the second layer would take as input just 1D x points as usual, x being the (weighted) averages of the components. This method includes the already implemented one as a special case with n=1 and `W = Ones(ndata,1)`. Moreover also in this case it would be possible to give different importance even to the single tuples of coordinates too, as it has been proposed before for the 1D case. That would by sure introduce some overhead, so it has to be done only if the results won't be satisfactory enough otherwise. Finally that "squeezing" from n-D to 1D could be performed later on, in another layer. This action might have two concurring effects: from one side the computational effort of every training iteration is being increased, but from the other one less iterations might be needed to reach convergence, since the richness of the data is being kept "alive" longer. These improvements haven't been introduced yet, since they go over the aims of the present project as they are outlined in the introduction. Notwithstanding this, it is worth noting that, especially in DNN programming, the more the code is enriched and exponentially the more tuning is necessary; in fact, not only the new choices must be tested, but also all their combinations with the previous hyper-parameters; so the above modifications, albeit quite simple conceptually, could require a slightly significant time to be implemented in an effective way.

[7] and a `b`, and so even backprop variables too.

3.3 The Optimizers class templates

In the `namespace Optimizers{...}`, defined in *Optimizers.hpp*, six classes are defined. Since they have to work either with `W` and `b`, they are defined as class templates. Moreover, in order to be able to use polymorphism, so to let the user choose the desired optimizer runtime, they are all related by inheritance, following this hierarchy:

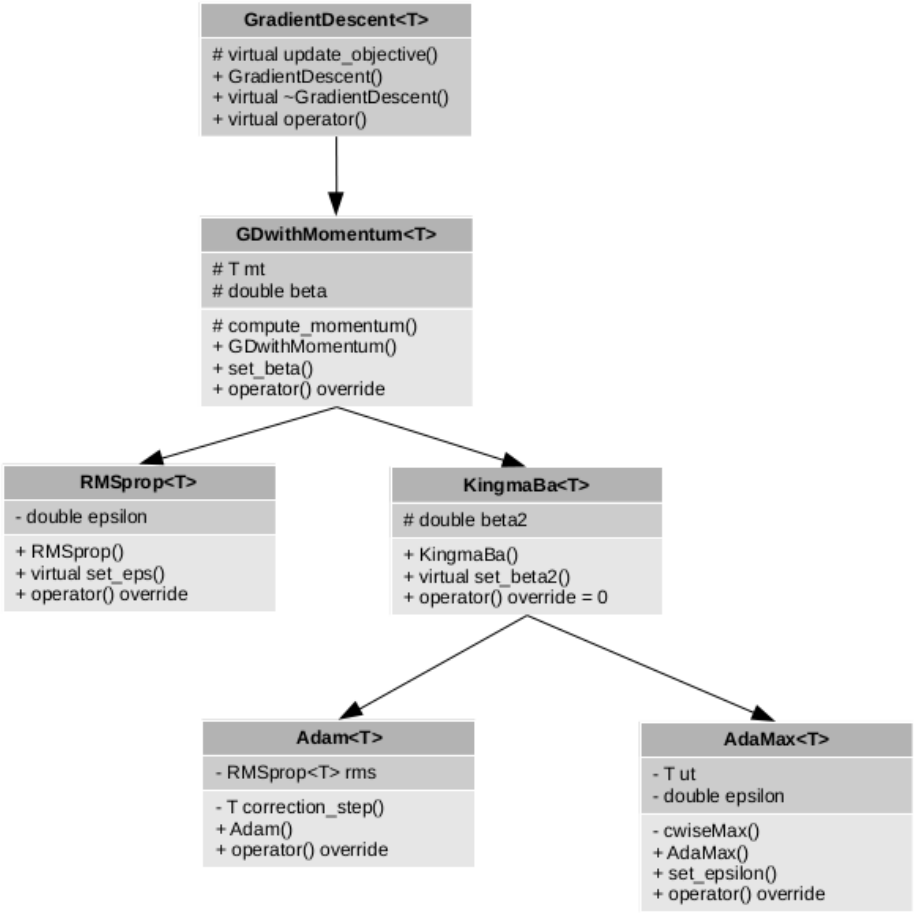

Figure 9: single inheritance pattern

As it was done during early stages of the development, it might seem reasonable to imagine a logical structure like the one sketched here instead:[8]

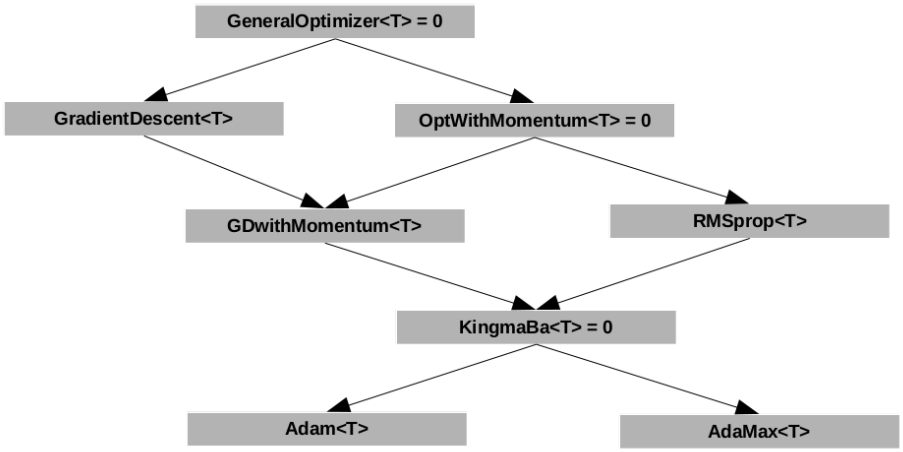

Figure 10: multiple inheritance pattern

but in this last case the multiple inheritance parts would have involved deadlock diamonds, requiring virtual inheritance.
Despite the last one being a possible choice, it has been decided to use only public single inheritance, employing only "is a" and "has a" relationships, so to keep the code as simple as possible, and also to avoid some overhead.
Finally, in order to keep the correct encapsulation even in the single inheritance - Fig. 9 design while guaranteeing `Adam<T>` class had the necessary access to the members of its `RMSprop<T>` member `rms`, class `RMSprop<T>` privately declares class `Adam<T>` as friend.

In the following we go a little deeper in some implementation details, class by class.
The gradient descent class is straightforward, it's here reported for completeness since it's the base class:

```
1 template<typename T>
2 class GradientDescent{
3 protected:
4   virtual void update_objective(T& theta, const T&
       delta_theta, const double& alpha){
5     theta = theta - alpha*delta_theta;
6   }
```

[8] "class name = 0" used as shorthand for "abstract class"

```
public:
  GradientDescent()=default;
  GradientDescent(const size_t & m, const size_t & n):
     GradientDescent(){}
  virtual ~GradientDescent()=default;
  virtual void operator()(T& theta, const T& gt, const double
     & alpha, const std::size_t & t){
    update_objective(theta,gt,alpha);
  }
};
```

Of course it defines a virtual destructor, even if it doens't do any work more than the default one does, to allow objects in the inheritance hierarchy to be dynamically allocated. Two decisions have been taken, mostly for ease: the template has been kept general, while it could have been specified a little more, using for example `Eigen::MatrixBase<T>`; and unused arguments have been passed to the overloaded call operator, so to have all the arguments that a general optimizer could need, in order to allow overriding in future derived classes.

The gradient descent with momentum class is the first one that implements gradients estimation, for this reason it has a member which is of the same type of the gradient to be estimated:

```
template<typename T>
class GDwithMomentum : public GradientDescent<T>{
protected:
  T mt;
  double beta{0.9};
  virtual void compute_momentum(const T & to_be_averaged){
    mt = beta*mt + (1. - beta)*to_be_averaged;
  }

public:
  GDwithMomentum(const size_t & m, const size_t &n):
     GradientDescent<T>::GradientDescent(m,n),mt(m,n){}

  virtual void set_beta(double b){beta=b;}

  void operator()(T& theta, const T& gt, const double& alpha,
      const std::size_t & t) override {
    compute_momentum(gt);
    this->update_objective(theta,mt,alpha);
  }
};
```

The RMSprop optimizer, besides updating in its own way the objective, it's the first to introduce a member, named epsilon, which is not foreseen in the theory, and that it's here used to prevent division by zero. It's also the first class in the hierarchy to be declared as final, and consequently to use `private` instead of `protected` in its definition:

```
template<typename T>
class RMSprop final : public GDwithMomentum<T>{
private:
  double epsilon{1e-8};
  friend class Adam<T>;

public:
  RMSprop(const size_t & m, const size_t & n):GDwithMomentum<
     T>::GDwithMomentum(m,n){}
  virtual void set_eps(double e){epsilon=e;}

  void operator()(T& theta, const T& gt, const double& alpha,
      const std::size_t & t) override {
    this->compute_momentum(gt.cwiseProduct(gt));
    this->update_objective(theta, (gt.cwiseProduct((1./(
       epsilon + sqrt(this->mt.array())))).matrix())), alpha);
  }
};
```

Of course, to allow declaring the Adam optimizer as friend class, it has been pre-declared before the definition of `RMSprop<T>`.

The `KingmaBa<T>` class is the only abstract class of the hierarchy. It has been implemented to provide the common interface to two great optimizers, Adam and AdaMax, which have been recently introduced in the same article by [Kingma-Ba]:

```
template<typename T>
class KingmaBa : public GDwithMomentum<T>{
protected:
  double beta2{0.999};
public:
  KingmaBa(const size_t & m, const size_t & n):GDwithMomentum
     <T>::GDwithMomentum(m,n){}

  virtual void set_beta2(double b2){ beta2=b2;}

  void operator()(T& theta, const T& gt, const double& alpha,
      const std::size_t & t)override = 0;
};
```

The Adam optimizer includes an operation of elevation to the iteration number-th power, which is a non-negative integer. Since `std::pow()` is known to be a "performance killer", because it solves a non linear problem which is completely unnecessary if the exponent is an integer, at first the `Adam<T>` class employed a function like this one:

```
double mypow(const double b, const std::size_t e){
  double ret{1.};
  for(size_t i=0; i<e; ++i)
    ret*=b;
  return ret;
}
```

But later on, compiling with optimization options, it turned out that, at least on the performed test, the standard library function was actually faster, with a total execution time of 4-5 seconds instead of 21-22 seconds on the used machine, and guaranteed a better result too, namely 2% instead of 3.9% relative L_2 error on the test set.[9]
So the Adam optimizer has been implemented in this way:

```
template<typename T>
class Adam final : public KingmaBa<T>{
private:
  RMSprop<T> rms;
  T correction_step(const double beta, const std::size_t t,
     const T & mt){
    return ((1.0/(1.0-std::pow(beta,t)))*mt);
  }
public:
  Adam(const size_t & m, const size_t & n): KingmaBa<T>::
     KingmaBa(m,n), rms(m,n){}
  void operator()(T& theta, const T& gt, const double& alpha,
      const std::size_t & t)override{
    //compute the momenta:
    this->compute_momentum(gt); //GDwithMomentum step
    rms.set_beta(this->beta2);
    rms.compute_momentum(gt.cwiseProduct(gt));//RMSprop step
    //update:
    this->update_objective(theta,( (correction_step(this->
       beta,t,this->mt)).cwiseProduct( ( 1./(rms.epsilon +
       sqrt((correction_step(this->beta2,t,rms.mt)).array()))
        ).matrix() )), alpha);
  }
};
```

[9]This seems to be due to the fact that the iteration number, that is the exponent, becomes early a large number, typical values' order of magnitude ranging from 10^4 to 10^5.

The above implementation doesn't strictly follow the one proposed in the literature: the correction step required by the algorithm has been composed with the update step, so to enhance performance. Anyway, also the standard algorithm steps are present in the code, albeit commented out (not reported here to save space).

Last, the AdaMax optimizer:

```
template<typename T>
class AdaMax final : public KingmaBa<T>{
private:
  T ut;
  T cwiseMax(const T& a, const T& b){
    T ret(a.rows(),a.cols());
    if(a.rows()!=b.rows() || a.cols()!=b.cols()){
      std::cerr<<"In AdaMax dimensions must match!\n";
      return ret;
    }
    for(size_t i=0; i<a.rows(); ++i)
      for(size_t j=0; j<a.cols();++j)
        ret(i,j)=std::max(a(i,j),b(i,j));
    return ret;
  }
  double epsilon{1e-8};

public:
  AdaMax(const size_t & m, const size_t & n): KingmaBa<T>::
     KingmaBa(m,n), ut(m,n){}

  void set_epsilon(double e){epsilon=e;}

  void operator()(T& theta, const T& gt, const double& alpha,
      const std::size_t & t)override{
    //compute the momenta:
    this->compute_momentum(gt);       //GDwithMomentum step
    ut=cwiseMax(this->beta2*ut,gt.cwiseAbs());//AdaMax step

    //update:
    this->update_objective(theta, (( 1. / (1.-std::pow(this->
       beta,t)) ) * ( this->mt.cwiseProduct( (1./(epsilon+ut.
       array())).matrix() ) )), alpha);
  }

};
```

Notice that the definition epsilon member could have been avoided if multiple inheritance were used.

3.4 Write_set and datasets

The src/write_set folder contains the source file write_set.cpp, which is used to generate the training and test datasets.
This is the first source file to rely to GetPot for the input phase:

```
int main(int argc, char** argv){
  GetPot commandline(argc,argv);
  if(commandline.search(2,"-h","--help")){
    help()<<endl;
    return 0;
  }
  string param_filename = commandline.follow("./../../data/
     parameters.pot",2,"-p","--parameters");
  GetPot datafile(param_filename.c_str());
  const size_t ntrain = datafile("ntraindata", 70);
  const size_t ntest = datafile("ntestdata", 30);
  VectorXd X;
  X.setLinSpaced(ntrain,-1.5,1.5);
  string file1{"./../../data/TrainingSet"+to_string(ntrain)},
      file2{"./../../data/TestSet"+to_string(ntest)};
  write_set(file1+".dat",X);
  X.setLinSpaced(ntest,-1.5,1.5);
  write_set(file2+".dat",X);
  return 0;
}
```

The first thing write_set `main` does is to check the optional parameters passed runtime, if it finds `-h` or `--help` it calls the help() function, that prints this:

```
$ ./a.out -h
Run with ./a.out [options]
Options:
-h, --help:  print this help
-p, --parameters <filename>:  reads parameters from <filename>,
default filename = "./../../data/parameters.pot"
```

and returns; otherwise it reads the number of training and test samples from the parameters file, which can thus be changed without recompiling the program.

The two datasets are then generated calling the following function, which currently generate a wave-packet-like function:

```
constexpr double pi=4.*std::atan(1.);
void write_set(string filename, const VectorXd & X, const
    char & separator = ' '){
  ofstream ofs(filename);
  auto u = [] (const double & x) {return exp(-x*x*5)*sin(5*pi
     *x);};
  for(size_t i=0; i<X.rows(); ++i)
    ofs<<X(i)<<separator<<u(X(i))<<"\n";
  ofs.close();
}
```

and are saved in the data folder.
To visualize them is possible to use the [gnuplot] script "PlotDataset.gnu", located in data/ too, eventually modifying the name of the desired dataset. For example for the standard 70 samples training set its output is:

```
$ gnuplot PlotDataset.gnu
Click any mouse button on selected data point to close
```

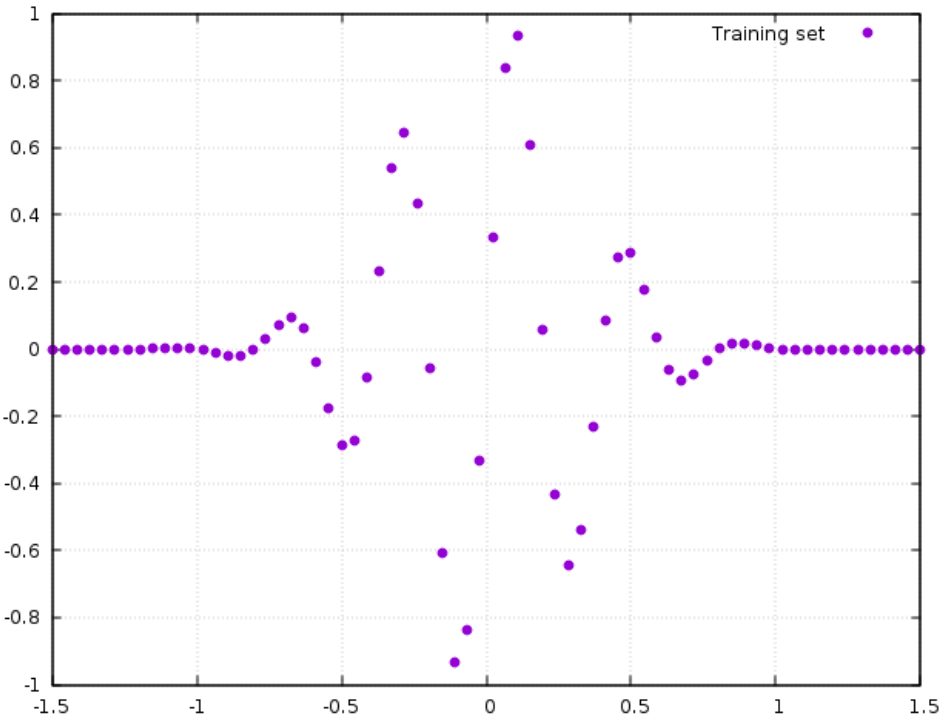

Figure 11: 70 samples training set

3.5 The main

The `main.cpp` source file takes into account for both execution of the main program and input/output operations.
The input stage is managed analogously to `write_set.cpp`, thus it's not reported here, for example we have:

```
$ ./main.out -h
Run with ./main.out [options]
Options:
-h, --help:  print this help
-v, --verbose:  activate verbose mode
-p, --parameters <filename>:  reads parameters from <filename>,
default filename = "./../data/parameters.pot"
```

So after a first GetPot based input phase, the training dataset and the architecture of the neural network are loaded:

```
1  //Load the training data:
2  MatrixXd TrainData(ntraindata,2);
3  bool read=read_set(train_filename+".dat",TrainData);
4  if(!read) return -1;
5
6  //Load the net architecture:
7  VectorXs architecture(nlayers);
8  read=read_set(architecture_filename+".dat",architecture);
9  if(!read) return -1;
```

Since the reading has to work both with matrices and vectors, the read_set function has been implemented as a function template, so to work with any matrix-like type which implements the overload of the call operator for accessing the elements:

```
1  template <typename T>
2  bool read_set(const string& setname, T & M, const char &
       separator=' '){
3
4    ifstream in(setname);
5    if(!in){
6    cerr<<"Error in opening: "<<setname<<std::endl;
7    return false;
8    }
9
10   string riga,numero;
11   for(size_t i=0; getline(in,riga); ++i){
```

```
    istringstream rigastream(riga);
    for(size_t j=0;getline(rigastream,numero,separator);++j)
      M(i,j)=stod(numero);
  }
  return true;
}
```

After that, the network is constructed following the "architecture.dat" file instructions, calling the constructor of the NeuralNetwork class, and then trained and tested:

```
//Construct the net:
NeuralNetwork nn(architecture);

//Train the net:
nn.train(TrainData,alpha,niter,W_opt,b_opt,tol,nref,verbose);

//Load the test data:
MatrixXd TestData(ntestdata,2);
read=read_set(test_filename+".dat",TestData);
if(!read) return -1;

//Test the net:
VectorXd yhat;
double errL2;
tie(yhat,errL2)=nn.test(TestData);
```

The training results are printed to the standard output by the `train()` function itself, by default only in a synthetic way if the `--verbose` option is not passed to main.
Differently, the test results are printed later, directly in the main, both in a written and graphical way.
First the numerical results are output on the terminal window and the predicted solution is saved in the prevision file, which by default is data/yhat300.dat:

```
//Print the results:
cout<<"Relative L2 error on test set = "<<errL2<<endl;
write_vector(prevision_filename+".dat",yhat)<<"\n"<<endl;
```

by means of the `write_vector()` function:

```
ostream & write_vector(const string & ofsname, const VectorXd
    & X){
  ofstream out_file(ofsname);
  streambuf* stream_buffer_cout = cout.rdbuf();
```

```
4    streambuf* stream_buffer_file = out_file.rdbuf();
5    cout.rdbuf(stream_buffer_file);
6    cout<<X;
7    out_file.close();
8    cout.rdbuf(stream_buffer_cout);
9    cout<<"Output written on "<<ofsname;
10   return cout;
11 }
```

which redirects the `std::cout` stream buffer to the output file one (and then restore it) to store the y-coordinate only of the predicted function in the file. In the end, before returning the control to the terminal user, the predicted function is plotted at screen together with the correct solution, given by the test dataset, using the gnuplot-iostream utility:

```
1  Gnuplot gp;
2  //first way:
3  vector<double> xtest(TestData.rows()),ytest(TestData.rows());
4  VectorXd::Map(&xtest[0], ntestdata) = TestData.col(0);
5  VectorXd::Map(&ytest[0], ntestdata) = TestData.col(1);
6  //second way
7  vector<double> prevision(yhat.data(), yhat.data() + yhat.rows
       ()*yhat.cols());
8  //plot:
9  gp<<"plot"<<gp.file1d(std::tie(xtest,ytest))<<
10 "w lp lw 4 title 'Test Data',"<< gp.file1d(std::tie(xtest,
       prevision))<<
11 "w lp lw 1.5 title 'Prevision'"<<endl;
```

It's also possible to plot in a second time this last graph without executing again the main program. Indeed, the data folder contains a "PlotPrevision.gnu" gnuplot script too, which makes use of the `paste`[10] GNU coreutil to avoid duplicates of the common x-coordinate:

```
1  plot "TestSet300.dat" u 1:2 w lp lw 4 title "Test Data", \
2  "< paste TestSet300.dat yhat300.dat" u 1:3 w lp lw 1.5 title
       "Prevision"
3  pause mouse "Click any mouse button on selected data point to
       close\n"
```

At the moment of writing, some graphical functionalities, like adding the grid interactively from the plot window, are possible only when the "PlotPrevision.gnu" script is used, due to current version gnuplot-iostream limitations.

[10] see e.g. `$ man paste`

Chapter 4

Examples

In this chapter we provide complete examples of building and running the program; some info are already reported in README.md.
First of all the program can be obtained at GitHub. It's suggested to clone the repository, for example if using https executing in a terminal window:

```
$ git clone https://github.com/pjbaioni/neural-net
```

that will create the folder neural-net in the current directory; otherwise, if downloading is preferred, it enough to go here.
In the following it is assumed that the user is in the directory neural-net, that employs the [APSC] course modules, year 2018-2019, and that [Eigen] and [Boost] modules have been already loaded, e.g. like that:

```
$ source /u/sw/etc/bash.bashrc
$ module load gcc-glibc/7
$ module load eigen
$ module load boost
```

The last versions of the code have been tested only with modules, which make uses of gcc 7.1, eigen 3.3.3 and boost 1.63.0, but there should be no problems if a local standard installation of the required libraries is used. In this case the make options containing environmental variables that start with `mk`, which are present in the two children Makefiles, have to be corrected in such a way that they take the correct path, as explained in the README.md file.

4.1 Building

The program can be automatically built through [Make]: it's enough to type `make`, `make all` or `make build` while being in the root directory. For example typing `make` will produce the following output:

```
neural-net $ make
for dir in ./src/write_set ./src ; do \
make build -C $dir ; \
done
make[1]:  Entering directory '/home/pjb/neural-net/src/write_set'
g++ -std=c++14 -O3 -DNDEBUG -I$mkEigenInc -I./../../include/
write_set.cpp -o write_set.out
make[1]:  Leaving directory '/home/pjb/neural-net/src/write_set'
make[1]:  Entering directory '/home/pjb/neural-net/src'
g++ -std=c++14 -Ofast -DNDEBUG -I$mkEigenInc -I$mkBoostInc
-I./../include/ -c -o NeuralNetwork.o NeuralNetwork.cpp
g++ -std=c++14 -Ofast -DNDEBUG -I$mkEigenInc -I$mkBoostInc
-I./../include/ -c -o main.o main.cpp
g++ -std=c++14 -Ofast -L$mkBoostLib -lboost_filesystem -lboost_system
-lboost_iostreams NeuralNetwork.o main.o -o main.out
make[1]:  Leaving directory '/home/pjb/neural-net/src'
```

Indeed, the base Makefile placed in the neural-net folder recursively calls the sub-directories Makefiles. Of course the program can be built even without make: it's enough to copy and paste the above lines starting with `g++` in the terminal emulator.
The father Makefile foresees different commands too, which can be inspected typing `make help`:

```
neural-net $ make help
Type:
- make, make all or make build to build
- make clean to remove object files
- make distclean to clean and to remove executables and generated
datafiles
- make run to run the main program with default parameters and
options, as specified in data/Parameters.pot
- make test<number> to execute the number-th test case
- make verbose_run to do as make run but in verbose mode
See doc/report.pdf for more informations
```

Most of them will be used in the next sections.
Mind that the children Makefiles have some more specific options that might be useful when running the program, and that further personalization can be achieved by direct running the executables, as shown typing e.g. `main.out --help` while being in the `src` directory.

4.2 Test cases

In the following some examples of execution are reported, both to provide guided instructions on how to run the program and to show its effectiveness. The main program, which is contained in the `src` directory, needs input datafiles to be run; some of them, namely the "TrainingSet*.dat" and "TestSet*.dat" files, can be generated by means of the write_set utility, the others, such as the "*.pot" and "architecture*.dat" files, can be easily created by the user relying on the already provided ones.
All the reported examples use a 70 samples training set and a 300 samples test set, albeit different choices are possibles. It might be noticed that, as anticipated in chapter 1, actually the test set has to be just big enough to check the performance of the neural network. For this purpose a 30 samples test set would have been perfect, and it wouldn't have produced appreciable differences in the numerical result w.r.t. the employed ones. Here it has been chosen to use a 10× greater test set only to obtain smoother graphical results, e.g. going from:

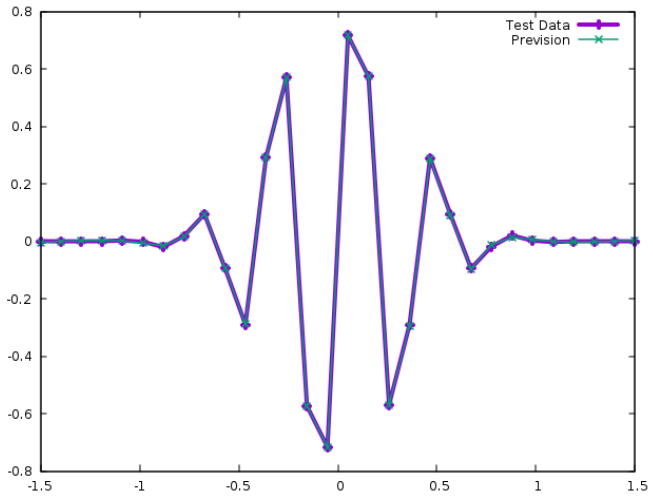

Figure 12: Test3 - 30 samples

to Figure 15.

4.2.1 Test 1: simple Gaussian function

As first example of execution it's possible to run:

```
neural-net $ make test1
```

which will produce the following output:

```
cd ./src/write_set && ./write_set.out -p "./../../data/Test1.pot"
Writing ./../../data/TrainingSetTest1.dat
Writing ./../../data/TestSetTest1.dat
Done
cd ./src && ./main.out -p "./../data/Test1.pot"
Total iterations = 37500
Cost functional on the training set = 1.72527e-06
Relative L2 error on test set = 0.000497972
Output written on data/YhatTest1.dat
```

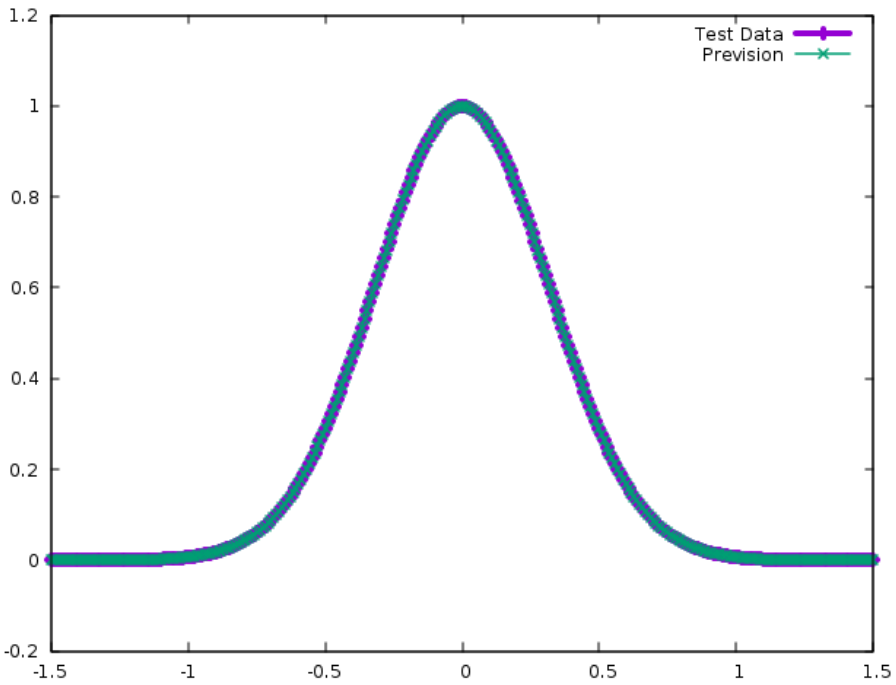

Figure 13: Test1

Both the executables read parameters from the file “Test1.pot”:

```
################################################################
############ Runtime data for Neural Network ###############
################################################################

# Number of training samples:
ntraindata = 70

# Number of test samples:
ntestdata = 300

# Number of layers:
nlayers = 9

# (Initial) learning rate:
alpha = 0.01

# Maximum number of training iterations:
niter = 37500

# (Initial) Tolerance (for convergence check):
tol = 0.001

# Weights optimizer:
# 0 = Gradient descent
# 1 = Gradient descent with momentum
# 2 = RMS prop
# 3 = Adam
# 4 = AdaMax
W_opt = 4

# biases optimizer (values as above):
b_opt = 3

# Number of refinements for alpha decay:
nref = 3

# frequence of the wave:
omega = 0.

# phase of the wave:
phi = 0.5

# Kind of dataset:
# 0 Linspaced
# 1 Uniform random
# 2 Normal random
xspacing = 0
```

All the used .pot files follows this structure, so it's possible to solve different problems editing a pre-existent .pot file or creating a new one which specifies the values of the above quantities; order doesn't matter.
All the non .pot files needed by the main program are automatically generated by the write_set subroutine, but one, which specifies the architecture of the network. In this examples it has been used the file provided "architecture9.dat" file, in the next paragraph it is shown how to use another one or to modify it.

4.2.2 Test 2: changing architecture

In the above example the error on the test set was just 0.05%, so solving this simple problem with a 9 layer neural network might have be an overshoot. When this is the case it's easy to change the architecture: here a 5 layer one is employed, as specified in the file "architecture5.dat", which is simply a one column file that contains the number of nodes per layer:

```
1
3
5
3
1
```

In principle any file could be used, with two restriction:

- the first and the last layer must have only one node;
- the filename must be "architectureNL.dat", where NL is the same number of layers specified in the .pot file used.

With the above file, keeping fixed the other settings, it is obtained:

```
neural-net $ make test2
cd ./src/write_set && ./write_set.out -p "./../../data/Test2.pot"
Writing ./../../data/TrainingSetTest2.dat
Writing ./../../data/TestSetTest2.dat
Done
cd ./src && ./main.out -p "./../data/Test2.pot"
Total iterations = 37500
Cost functional on the training set = 4.89304e-05
Relative L2 error on test set = 0.00273389
Output written on data/YhatTest2.dat
```

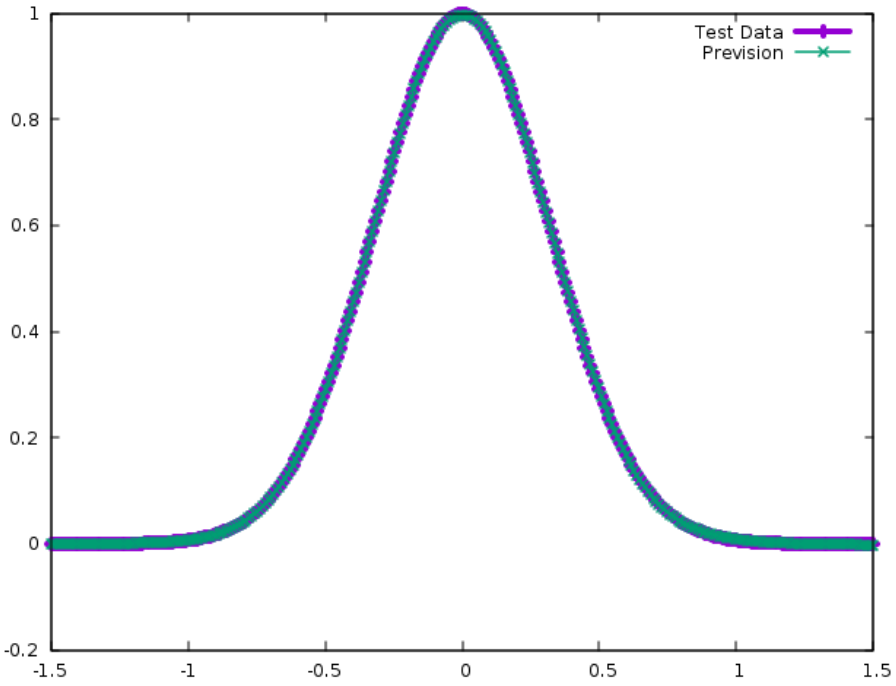

Figure 14: Test2

which is still a satisfactory result.

4.2.3 Test 3: reconstruction of a wave-packet

The third test shows a more interesting problem, where a wave-packet like functions is reconstructed. Both the pulsation and the phase of the modulated wave can be changed, editing the .pot file. With a zero phase displacement and a 5π pulsation it is possible to obtain:

```
neural-net $ make test3
cd ./src/write_set && ./write_set.out -p "./../../data/Test3.pot"
Writing ./../../data/TrainingSetTest3.dat
Writing ./../../data/TestSetTest3.dat
Done
cd ./src && ./main.out -p "./../data/Test3.pot"
Total iterations = 75000
Cost functional on the training set = 0.000116357
Relative L2 error on test set = 0.00612406
Output written on data/YhatTest3.dat
```

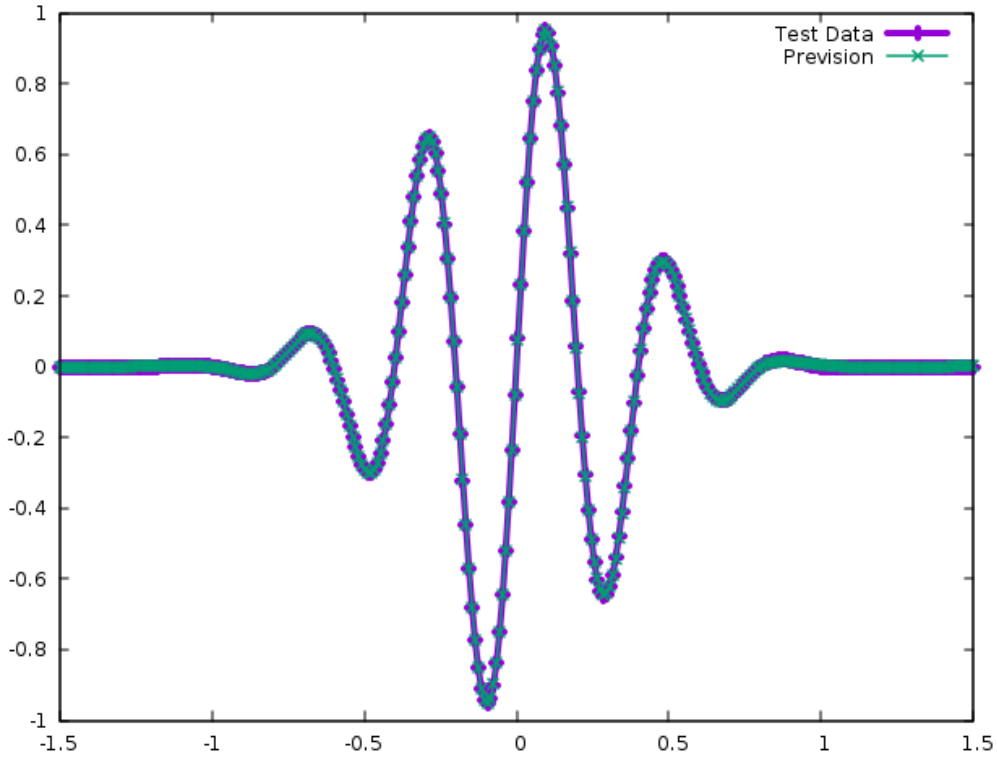

Figure 15: Test3

where the already used 9 layer network as been employed.

4.2.4 Test 4: random sampling

Until now the dataset where made up of equispaced x points and their relative y coordinate. The program is able to handle random generated samples too, choosing a different flag value in the .pot file.
It has to be noticed that in this case the datasets won't be sorted in ascending order in the x coordinate anymore; this of course doesn't create any problem to the solver, but would result in a very messy graph if "with line points" option is used in gnuplot-iostream as above. For this reason it has been chosen to distinguish the two cases, random or not random, and to plot only points in the random case. For example, using a gaussian distribution for the x coordinates, changing the maximum number of iterations to 150000 and keeping fixed other parameters we get:

```
neural-net $ make test4
cd ./src/write_set && ./write_set.out -p "./../../data/Test4.pot"
```

```
Writing ./../../data/TrainingSetTest4.dat
Writing ./../../data/TestSetTest4.dat
Done
cd ./src && ./main.out -p "./../data/Test4.pot"
Total iterations = 98300
Cost functional on the training set = 0.000283243
Relative L2 error on test set = 0.0538738
Output written on data/YhatTest4.dat
```

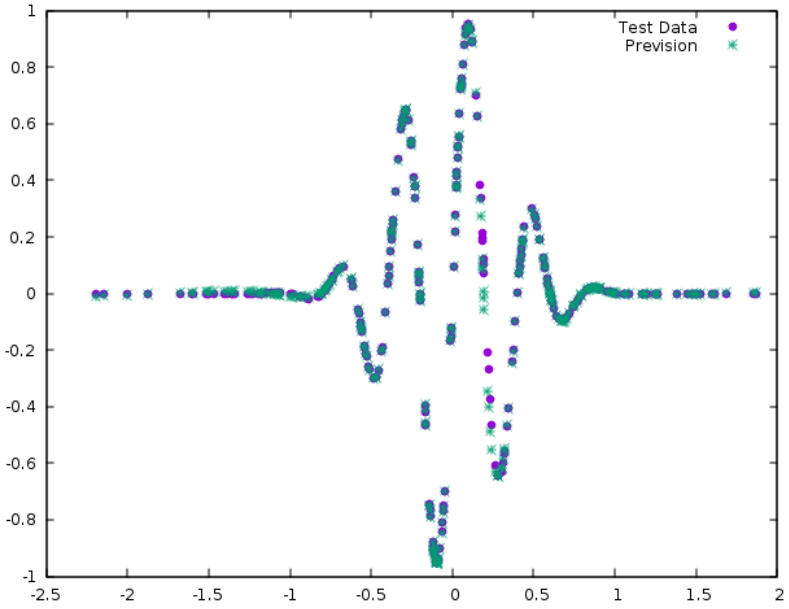

Figure 16: Test4

The result is still acceptable, but it starts getting worse, since the test set error is about at 5%.
When this happens it's very important to check the change in the training set too: one could be tented to simply train the net longer, or more or less equivalently to train a bigger network, but in this case it might be "dangerous". Indeed, going from example 3 to example 4, the train set error has doubled, and this might be reasonable since the data is now random, but the test error has increased tenfold, starting to show clues of over-fitting. In such cases it's better to generate larger train set instead; a counter example is provided in the following paragraph.

4.2.5 Test 5: over-fitting

This test serves as counterexample to show what can happen if a too-strong network, relatively to the faced problem, is used. Same conclusion applies to a normal-size network trained too much.
It's worth nothing that this test, differently from the above ones, might take a bit longer to execute, about 1 minute on the tested machine, exactly because essentially it is doing too much training. Typing `ctrl+C` in any moment would stop execution, resulting in a `make: *** [Makefile:44: test5] Interrupt`.
The expected result is:

```
neural-net $ make test5
cd ./src/write_set && ./write_set.out -p "./../../data/Test5.pot"
Writing ./../../data/TrainingSetTest5.dat
Writing ./../../data/TestSetTest5.dat
Done
cd ./src && ./main.out -p "./../data/Test5.pot"
Total iterations = 150000
Cost functional on the training set = 4.17662e-06
Relative L2 error on test set = 0.142108
Output written on data/YhatTest5.dat
```

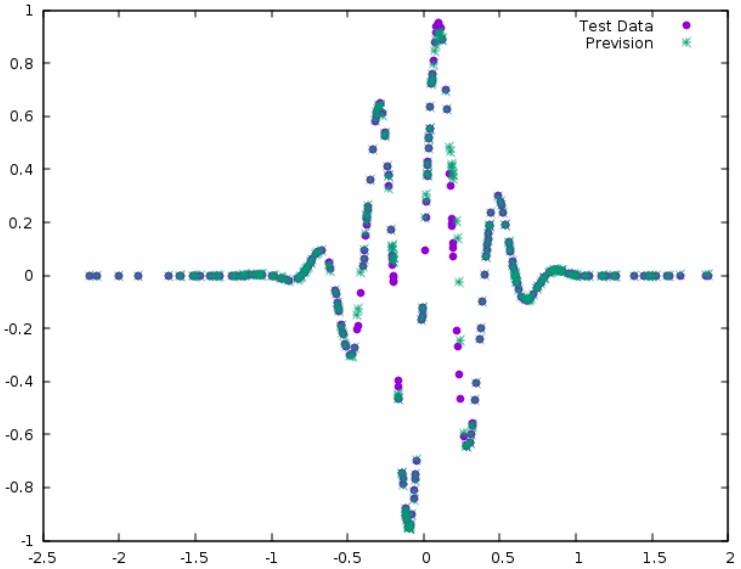

Figure 17: Test5

So, although the train set error has decreased 100 times, the test set error has become 3 times greater: the employed 15 layer network has specialized too much on the train set and struggles to generalize to the unseen test set. Unfortunately this situation is very common in DNN programming. Some general methods to face it are provided in chapter 1, while some practical advices related to the present code are given in the next 4.3 section.

4.3 Running the program

The easiest way to perform general runs of the program is to use the `make run` command in the root directory, or the final-output-equivalent `make verbose_run`, and to modify the provided "Parameters.pot" file as wanted. For example, performing a verbose run with the current setting results in:

```
neural-net $ make verbose_run
make run -C ./src/write_set
make[1]:  Entering directory '/home/pjb/neural-net/src/write_set'
./write_set.out
Writing ./../../data/TrainingSetParameters.dat
Writing ./../../data/TestSetParameters.dat
Done
make[1]:  Leaving directory '/home/pjb/neural-net/src/write_set'
cd ./src && ./main.out --verbose
t=100 cost=3.20761 W_opt=AdaMax b_opt=Adam alpha=0.01
t=200 cost=2.34495 W_opt=AdaMax b_opt=Adam alpha=0.01
t=300 cost=1.53293 W_opt=AdaMax b_opt=Adam alpha=0.01
[...]
t=24800 cost=0.000997826 W_opt=AdaMax b_opt=Adam alpha=0.0001
t=24900 cost=0.000996447 W_opt=AdaMax b_opt=Adam alpha=0.0001
t=25000 cost=0.000996266 W_opt=AdaMax b_opt=Adam alpha=0.0001
Total iterations = 60400
Cost functional on the training set = 0.000996266
Relative L2 error on test set = 0.0173802
Output written on data/YhatParameters.dat
```

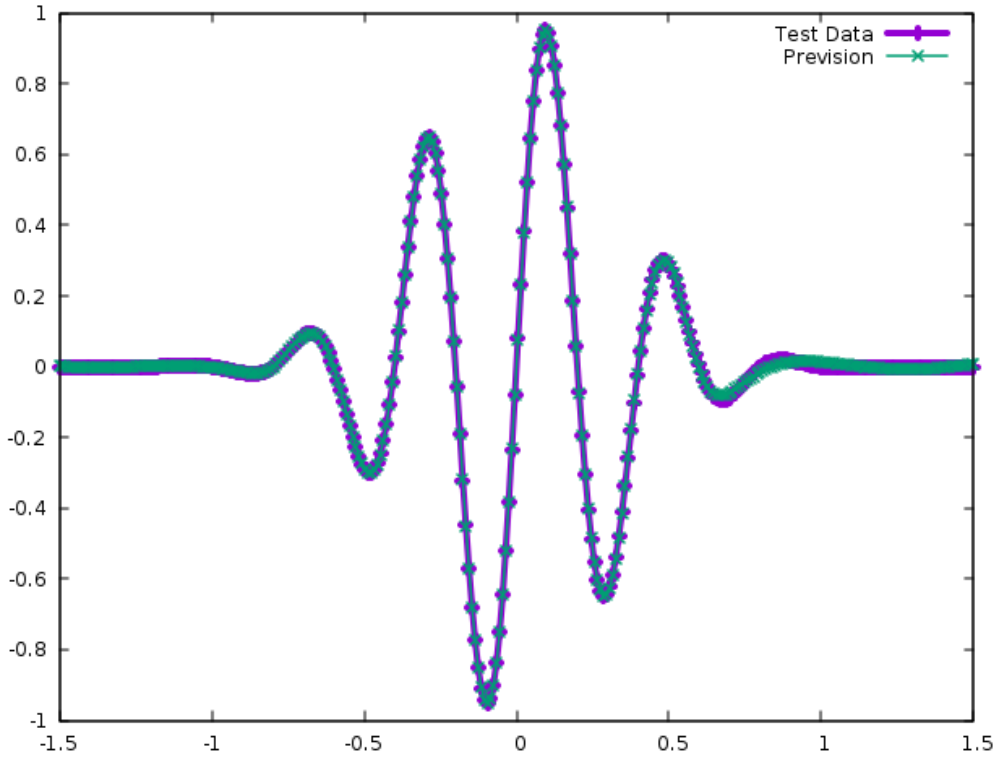

Figure 18: verbose run

commenti: runtime command line options, non rewriting same datasets (separate use of main and write_set). Poi refinements, automatic alpha decay, niter infatti è minore di 75k. Passare quindi a spiegare come tunare.

Old section

It's also possible to pass options to the main by make, redefining the variable `OPTIONS` runtime, e.g.:

```
src $ make run OPTIONS=-p="parameters.pot" OPTIONS+="--verbose"
./main.out -p=parameters.pot --verbose
t=100 cost=2.14956 W_opt=AdaMax b_opt=Adam alpha=0.01
t=200 cost=1.43787 W_opt=AdaMax b_opt=Adam alpha=0.01
t=300 cost=0.996626 W_opt=AdaMax b_opt=Adam alpha=0.01
...
t=24900 cost=0.000127942 W_opt=AdaMax b_opt=Adam alpha=0.0001
```

```
t=25000 cost=0.000127854 W_opt=AdaMax b_opt=Adam alpha=0.0001
Total iterations = 70300
Cost functional on the training set = 0.000127854
Relative L2 error on test set = 0.00665947
Output written on ./../data/yhat300.dat
```

Some parameters can be changed editing the "parameters.pot" file (or another one, and passing it's name to main):

```
1 ############################################################
2 ######### Runtime data for Neural Network - main ##########
3 ############################################################
```

It's worth noting that increasing the net size not necessarily results in a better performance. For example, if it is used as architecture {`1, 14, 12, 11, 15, 11, 14, 13, 15, 12, 1`}, that corresponds to a neural net composed by one reading and ten working layers, the hidden ones with a random number of nodes between 11 and 15, the output on the same dataset is:

```
Total iterations = 75000
Cost functional on the training set = 6.82184e-08
Relative L2 error on test set = 0.0543924
Output written on ./../data/yhat300.dat
```

meaning that the net is starting to overfit the training data, and thus to have difficulties to generalize to the unseen test data; on the contrary, if we take a more complex function, e.g. one with more oscillations, a deeper and bigger network will be needed to obtain good results.
In very extreme underfitting and overfitting case, even NaNs and Infs can be appear, so in general a good procedure is to start with a smaller network, since in this case the result is produced faster, and then to add layers and nodes until the results starts to get worse.

As previously mentioned, it might happen that better results are achieved with an hand made piecewise constant learning rate decay. To do that it's enough to:

1. change the `train(...)` line in main.cpp so that it doesn't take `tol`, `nref` and `verbose`, and fix a low `alpha`, say 0.01, and a large `niter`, say 25000;

2. execute the code, and notice at what iteration the cost starts to become constant or oscillating;

3. change the `niter` to that number;

4. add another training line with a lower `alpha`, for example 0.001, and a large number of iterations, as at point 1;

5. repeat until you get a good result.

Of course, instead of doing as at point 1, parameters can be changed also in parameters.pot; for example point 1 it's equivalent to put `nref`=1, `tol`=−1.0, `alpha`=0.01, `niter`=25000 and to run the code with the "`-v`" (or "`--verbose`") option.
As last remark, it's worth underlining that changing the architecture will invalidate the optimality of the `alpha` and `niter` couples of hyper-parameters found following the above procedure.

Conclusions

In the first chapter of this report we have illustrated in detail the mechanism behind deep learning, going also over the surface for what concerns typical problems one encounters when designing a deep neural network, such as hyperparameter tuning and overfitting, proposing solutions like L_2 regularization and dropout and giving an idea of the scheme one can follow when tuning the network. Moreover we've showed how gradient descent can be extended to tackle deep learning problems, till arriving to Adam and AdaMax, which have been proposed the first time in 2015 and which are now employed in many different situations.
In chapter 2 we have showed two main techniques which can be used to face PDEs problems by means of deep neural networks, as some interesting and useful theoretical results, following the most recent articles and reporting pros, like the ease of extension to more dimensions, aspects that require some care, like the treatment of BCs, and problems that are still open, in general related to the lack of a theoretical framework, implying among the others uncertainty about theoretical error bounds and appropriate values of hyperparameters.
In chapters 3 and 4 we have focused on explaining how a DNN based efficient and numerical analysis relevant application has been developed from scratch, exposing both the technicalities of the adopted algorithms and the design choices that have been taken, both in the details of the implementation and in the general architecture of the program; some insights into expansions of possible interest have been discussed too. After this a complete benchmark case has been presented, accompanied by comprehensive instructions on how to build and run the program.
Before concluding, we notice that it isn't uncommon to find comparisons between results achievable with DNNs methods and classical ones, for example [Jinchao Xu et al.] prove that in a 1D specific example a ReLU DNN can recover a better result, in terms of convergence accuracy, than adaptive finite element method, while [Weinan-Bing] show that with fewer parameters, their DNN based method gives more accurate solution than the finite

difference method . In such comparisons it's worth considering that the time required by a DNN-like approach is very different from the one of standard methods, indeed from one side tuning a DNN program properly requires a large amount of time, because even hyperparameters which have given great results in one field of application are often completely not suitable to the others, so a considerably high number of combinations have to be tried; from the other side, once tuned and trained, the DNN it's always ready to perform different tests, and so to make all the predictions needed, just at the price of one forward propagation, while a standard solver has to work every time from scratch.
Considering all these aspect, we can certainly conclude that deep learning is a very promising and interesting field even for numerical analysis, both from the theoretical and from the applications point of view.
Moreover, its completely specific time scales makes it particularly adapt to find its natural space in the standard procedure of computational science and engineering: if we have an engineering machine or a scientific experiment on which we have to able to operate in very short times, for example to perform rapid slight changes to the configuration in response to sudden perturbations so to guarantee the stability of the system, we start proceeding with the mathematical modeling of the physical problem, we analyze it theoretically and then we move to numerical analysis and finally to programming for example a FEM code. At this point we can design a DNN based solver which we train on the solutions we have found with the previous methods. In this way we end up with a software which is able to find in very short times, the time of just one forward propagation, solutions to the model when the data, i.e. the physical situation, has changed a little, providing in such a way a powerful tool for the real-time control of the state of the machine or of the experiment.

Appendix

The code used in section 2.2 follows:

```
// Code written to extimate the number of degrees
// of freedom (dof) needed to reach a good extimate
// in section 2.2 of the report.

////////////////////////////////////
////////////Instructions//////////
////////////////////////////////////

/*
To run the code, save it as and edp file: <filename>.edp
(Note that edp files can be opened by any text editor, saving
as .txt works as well)
Then in a terminal emulator do:
$ FreeFem++ <filename>.edp

The code has been tested on Debian 9,
equipped with FreeFem++ version  3.47

pjbaioni 2020
*/

////////////////////////////////////
///////////////Code///////////////
////////////////////////////////////

// Mesh
int n=36;
//n->dofs: 9->13, 12->20, 16->30, 18->41, 20->45, 36->137
border c(t=0, 2*pi){x=cos(t); y=sin(t); label=1;}
mesh Th=buildmesh(c(n));
plot(Th,cmm="Mesh",wait=1);

// Space
fespace Vh(Th,P1);
cout<<"Degrees of freedom N = "<<Vh.ndof<<endl;

```

```
// Manufactured solution:
real a=1.;
mesh T=buildmesh(c(5*n));
fespace V(T,P2);
V uexact = exp(-a*(x^2+y^2))*(x^2+y^2-1);
plot(uexact,fill=1,value=1,cmm="u_exact_h",wait=1);

// Data
func f = -4*exp(-a*(x^2 + y^2))*(a^2*x^4 + 2*a^2*x^2*y^2 - a
   ^2*x^2 + a^2*y^4 - a^2*y^2 - 3*a*x^2 - 3*a*y^2 + a + 1);
func g = 0;

// Solution
Vh uh,vh;
macro grad(u) [dx(u), dy(u)] //
solve poisson2d(uh,vh,solver=CG)=
      int2d(Th)(grad(uh)'*grad(vh))
      -int2d(Th)(f*vh)
      +on(1,uh=g);
plot(uh,fill=1,value=1,cmm="uh",wait=1);

//L2 error (normalized)
real num=int2d(Th)((uexact-uh)*(uexact-uh)); num=sqrt(num);
real den1=int2d(Th)(uexact*uexact); den1=sqrt(den1);
real den2=int2d(Th)(uh*uh); den2=sqrt(den2);
real den=(den1+den2)/2;
cout<<"Relative error in  L2 norm = "<<num/den<<endl;

// Expected output:
/*
  --  mesh:  Nb of Triangles =    236, Nb of Vertices 137
Degrees of freedom N = 137
--  mesh:  Nb of Triangles =   5706, Nb of Vertices 2944
-- Solve :
min -0.981309  max -1.42832e-31
Relative error in  L2 norm = 0.0165159
times: compile 0.016902s, execution 0.093685s,  mpirank:0
CodeAlloc : nb ptr  2902,  size :366776 mpirank: 0
Ok: Normal End
*/
```

Bibliography

[APSC] Advanced Programming for Scientific Computing course lectures and material, Politecnico di Milano, 2019.

[Andrew NG] Professor Andrew NG's Deep Learning MOOC at Coursera, *https://www.coursera.org/specializations/deep-learning*, consulted in 2019.

[Boost] Boost C++ libraries, *https://www.boost.org/*

[cppreference.com] An online reference of the C++ language, *https://en.cppreference.com/w/cpp*

[Eigen] Eigen C++ template library for linear algebra, *https://eigen.tuxfamily.org/*

[FreeFem++] FreeFem++ PDE solver, *https://freefem.org/*

[GetPot] GetPot command line parser, *http://getpot.sourceforge.net/*

[Git] Git distributed version control system, *https://git-scm.com/*

[gnuplot] A GNU graphing utility, *http://www.gnuplot.info/*

[gnuplot-iostream] A C++ interface to gnuplot, *https://github.com/dstahlke/gnuplot-iostream*

[Jinchao Xu et al.] Juncai He, Lin li, Jinchao Xu, Chunyue Zheng, *ReLU Deep Neural Networks and Linear Finite Elements*, 2018, found at *https://arxiv.org/abs/1807.03973*

[Kailai et al.] Kailai Xu, Bella Shi, Shuyi Yin, code and technical report of the project for the course CS230 *Deep Learning, Winter 2018, Stanford University*, found at *https://github.com/kailaix/nnpde*

[Kingma-Ba] Diederik P. Kingma, Jimmy Lei Ba,*Adam: a method for stochastic optimization*, 2015, found at *https://arxiv.org/abs/1412.6980*

[Lagaris et al.] I.E. Lagaris, A. Likas and D.I. Fotiadis: *Artificial Neural Networks for Solving Ordinary and Partial Differential Equations*, 1997, found at *https://arxiv.org/abs/physics/9705023*

[Lippman et al.] Stanley B. Lippman, Josee Lajoie, Barbara E. Mojo, C++ primer - Fifth Edition, Addison-Weasly/Pearson Education, 2012.

[Make] GNU Make doc, *https://www.gnu.org/software/make/manual*

[Python] Python programming language, https://www.python.org/

[Sirignano-Spiliopoulos] Justin Sirignano, Konstantinos Spiliopoulos: *DGM: A deep learning algorithm for solving partial differential equations*, 2018, found at *https://arxiv.org/abs/1708.07469*

[TensorFlow] TensorFlow ML library, https://www.tensorflow.org/

[Weinan-Bing] Weinan E, Bing Yu *The Deep Ritz method: A deep learning-based numerical algorithm for solving variational problems*, 2017, found at *https://arxiv.org/abs/1710.00211*