Deep Learning for PDEs

Report of the joint APSC-NAPDE courses project

Paolo Joseph Baioni*

March 29, 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 treating 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 general indications on the ideal number of nodes and layers of the DNN.
In chapter 3 we explain the programming architectural choices and the structure of the developed code, freely available on GitHub[1], as well as possible extensions.
In chapter 4, after providing instructions on how to compile, link and run the program, we show some examples by means of benchmark cases.
In appendix we report the [FreeFem++] code written for the computations performed in section 2.2.

[1] *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, 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 practical, 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 being 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:

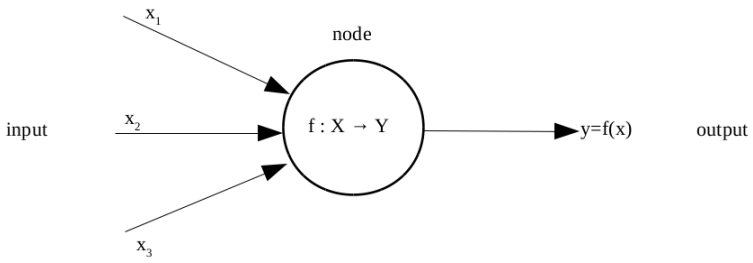

Figure 1

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.

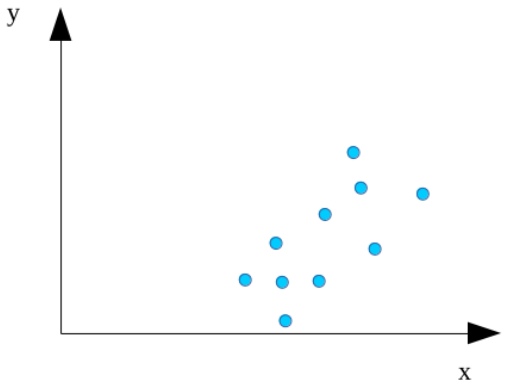

Figure 2

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[1] $\forall y$. Then it would be reasonable to update the f function as in the figure below:

[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.

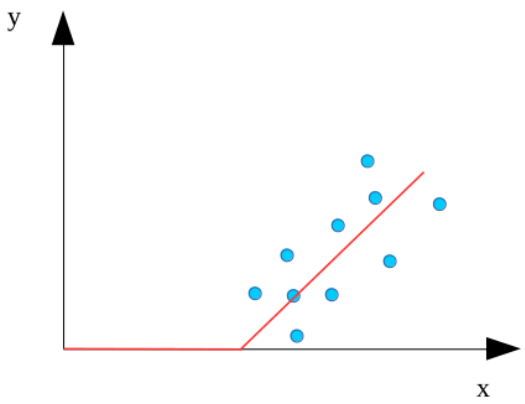

Figure 3

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.
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.

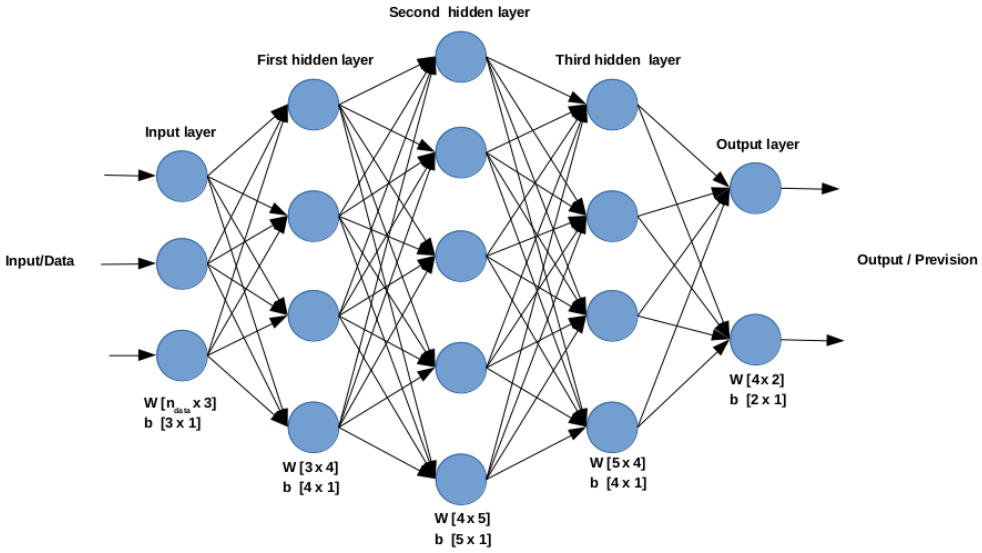

Figure 4

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.

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 let the neural network find it out by itself. In other words, given enough training example, 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.

[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.

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 it's own weights and bias, nonlinearize 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 it's 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.

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. it's 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:

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

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 a specific case.

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[3], 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[4], 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. 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 still.
This iterative process is resumed in the following flowchart:

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

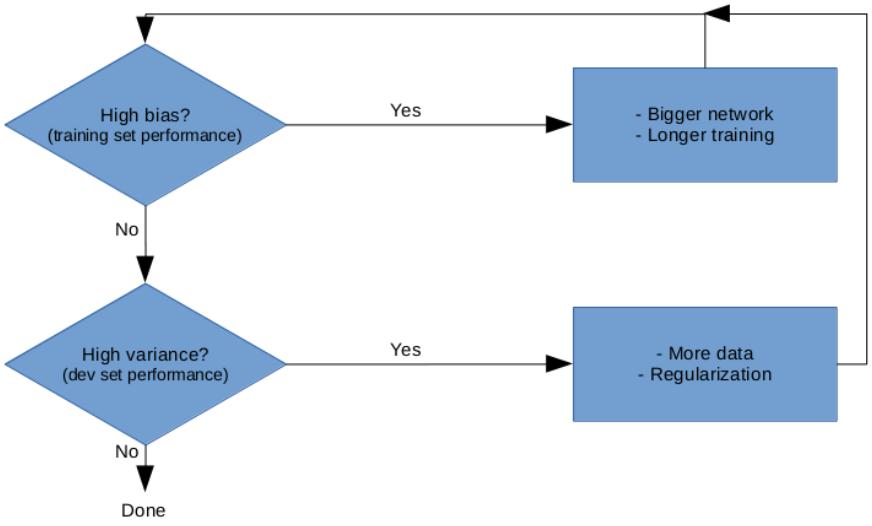

Figure 5

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 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[5]:

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

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.

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

Gradient descent with momentum is the direct application of (1.3) to standard gradient descent. For every iteration t:[6]

- calculate 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 on 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 cancel 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:

- $\forall t$ calculate dW, db
- 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:

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

[6]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).

where the superscript 2 has to be intended element wise, and s stands for square.
To visualize the idea behind RMSprop let's suppose we 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-LeiBa], 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[7]. The steps are:

[7] See [Kingma-LeiBa].

- 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 to big step 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,
- $\alpha \propto \frac{1}{\sqrt{t}} \cdot \alpha_0$

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[8]. 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.

[8]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 has to be just big enough to evaluate the performance of the network, while the training set is the one on which the network really *learn* 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 BVPs in a deep learning setting, which we will expose too. However they still use 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 already are 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 theoretical result (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 point 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 dimension;
- 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 point. 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]:

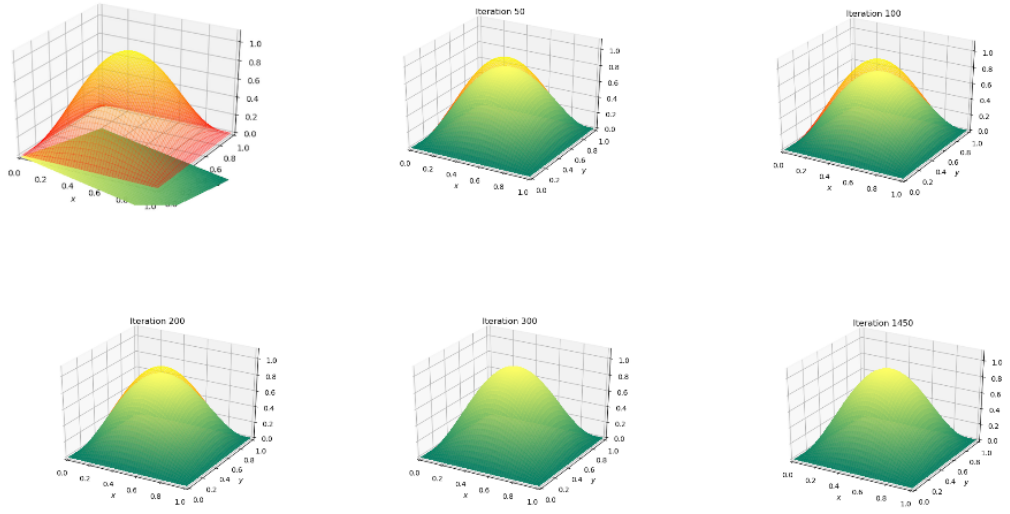

Figure 6

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 Python2 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 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.

The two main results are[5]:

[4] In the sense of open and connected.

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

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 with a standard triangulation of the domain is:

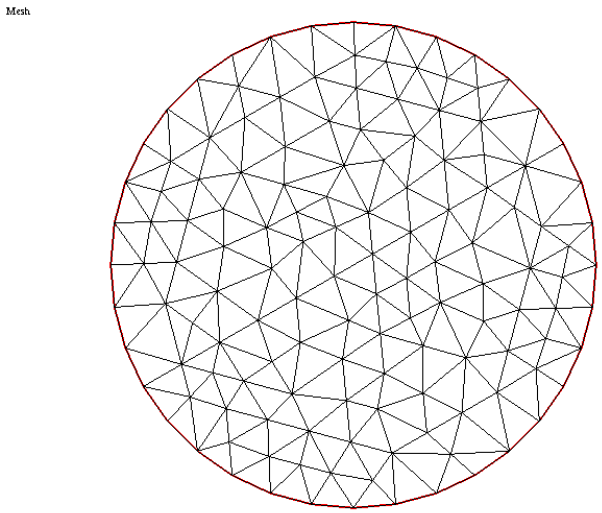

Figure 7

To do an estimate it's thus possible to assume $k_h = 6$.
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 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%

[6]Code in 4.2

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:

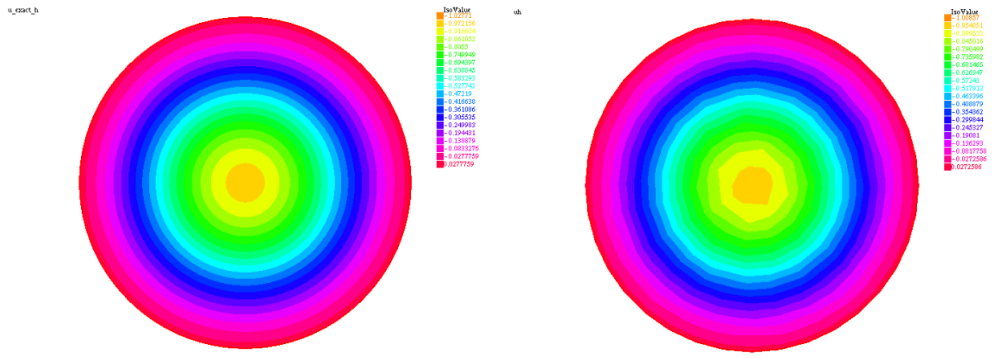

Figure 8

In the end 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 $\mathcal{O}(d2^{(d+1)k_h} N) \simeq 10^8$ 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.

[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 whats turn out to be DNN's numerical analysis strongest point: their remarkably capability of reconstructing function, 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*.
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 build the program and run an example.

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, [gnuplot] scripts in .gnu format, and their graphical output in .png format;
- 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 file 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 reason that will appear clearer later on[1], 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

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

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, but has allowed to rely on strongly optimized well known linear algebra libraries, which have made the difference. 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 loose somehow an eye-apparent evident sight on each node operations, keeping the original design of a neural network class composed by three different specifications of the layer class has appeared senseless, and thus it has been decided to simplify the code making the neural network class hold just vectors of the quantities that characterized every layer.
The choices made have thus led to a structure as the following:

```
class NeuralNetwork{

private:

  //Architecture hyperparameters:
  size_t nlayers;
  VectorXs nnodes;

  //Parameters to be optimized:
  vector<MatrixXd> W;
  vector<VectorXd> b;

  //Forward propagation outputs:
  vector<MatrixXd> L;
  vector<MatrixXd> A;

  //Back propagation gradients:
  vector<MatrixXd> B;
  vector<MatrixXd> dW;
  vector<VectorXd> db;

  //Optimizers:
  vector<shared_ptr<GradientDescent<MatrixXd>>> W_optimizer;
  vector<shared_ptr<GradientDescent<VectorXd>>> b_optimizer;

public:

  //Constructor:
```

```
29    NeuralNetwork(const VectorXs &);
30
31    //Training function:
32    void train(const MatrixXd & Data, double alpha, size_t
          niter, double tolerance, const size_t W_opt, const
          size_t b_opt, const 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.[2]
So the class NeuralNetwork hold 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`, that has to be somehow a vector of vector, i.e. not a matrix, since the number of nodes varies for each layer;

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

(iv) the optimizers for `W` and `b`, which hold a (smart) pointer to the base class of the Optimizers class hierarchy.[3]

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 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.

[2] See NeuralNetwork.hpp for the complete version.
[3] Optimizers.hpp is illustrated in section 3.3

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 a `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 need for 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.

Having introduced the data members, consider the member functions:

```
1  NeuralNetwork(const VectorXs & nn): nlayers(nn.rows()),nnodes
       (nn){
2
3    W.reserve(nlayers-1);
4    b.reserve(nlayers-1);
5    for(size_t l=0; l<nlayers-1; ++l){
6      W.emplace_back(MatrixXd::Random(nnodes(l),nnodes(l+1)));
7      b.emplace_back(VectorXd(nnodes(l+1)));
8    }
9
10   L.reserve(nlayers);
11   A.reserve(nlayers-1);
12
13   B.reserve(nlayers-1);
14   dW=W;
15   db=b;
16
17   W_optimizer.reserve(nlayers-1);
18   b_optimizer.reserve(nlayers-1);
19 }
```

The constructor initialize every member, in order of declaration. 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 them 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 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 (scomposed in different frames for clarity):

```
void NeuralNetwork::train(const MatrixXd & Data, double alpha
    , size_t niter, double tolerance, const size_t W_opt,
    const size_t b_opt, const size_t nrefinements, const bool
    verbose){
  ////////////////////
  //       Init     //
  ////////////////////

  size_t ndata=Data.rows();

  //L has nlayers components:
```

```
  for(size_t l=0; l<nlayers;++l)
    L.emplace_back(ndata,nnodes(l));
  //A hasn't the last one:
  for(size_t l=0; l<nlayers-1;++l)
    A.emplace_back(ndata,nnodes(l));
  //B hasn't the first one:
  for(size_t l=1; l<nlayers;++l)
    B.emplace_back(ndata,nnodes(l));

  //First layer only reads the input:
  L[0]=Data.col(0);
  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.
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()));
```

```
12      W_opt_name = "GDwithMomentum";
13      break;
14    case 2:
15      for(size_t l=0; l<nlayers-1; ++l)
16        W_optimizer.emplace_back(make_shared<RMSprop<MatrixXd
              >>(dW[l].rows(),dW[l].cols()));
17      W_opt_name = "RMSprop";
18      break;
19    case 3:
20      for(size_t l=0; l<nlayers-1; ++l)
21        W_optimizer.emplace_back(make_shared<Adam<MatrixXd>>(dW
              [l].rows(),dW[l].cols()));
22      W_opt_name = "Adam";
23      break;
24    default:
25      for(size_t l=0; l<nlayers-1; ++l)
26        W_optimizer.emplace_back(make_shared<AdaMax<MatrixXd>>(
              dW[l].rows(),dW[l].cols()));
27      W_opt_name = "AdaMax";
28      break;
29  }
30
31  switch(b_opt){
32    case 0:
33      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 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 take 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. An 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, invoked with a `.array()`; moreover, when optimization is activated, such conversions, as well as transpositions and so on, are implemented so that they are costless from a computational point of view.
Then a convergence check and, eventually, an output to std::output is performed:

```
1  ////////////////////////
2  //  Convergence check  //
3  ////////////////////////
4  //Output the current cost
5  //and check if convergence is reached:
6  if( verbose && (t%25==0) ){
7    cout<<"t="<<t<<" cost="<<cost<<" W_opt="<<W_opt_name<<"
         b_opt="<<b_opt_name<<" alpha="<<alpha<<"\n";
8    err = abs(old_cost-cost) / ( (cost+old_cost)/2 );
9    if(err<tolerance){
10     backup_t.push_back(t);
11     break;
12   }
13   else
14     old_cost = cost;
15 }
```

Just after it 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 to the left, e.g. `dW[0]` held the gradient of the cost functional w.r.t. the weights of the last layer.[4] 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 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);
  for(size_t l=nlayers-2; l>0; --l){
    //Compute gradient of cost wrt W:
    dW[l] = ( L[l].transpose() )*B[l];
    //Update W:
    //W[l] = W[l] - alpha*dW[l];
    (*W_optimizer[l])(W[l],dW[l],alpha,t);
    //Compute gradient of cost wrt b:
    db[l] = B[l].transpose().rowwise().sum();
    //Update b:
    //b[l] = b[l] - alpha*db[l];
    (*b_optimizer[l])(b[l],db[l],alpha,t);
    //Compute previous B:
    //(now there is tanh, and dx[tanh(x)]=1-x^2)
    B[l-1] = (1. - (A[l].array().square())) * ( (B[l]* (W[l].
        transpose()) ).array() );
  }
  //Updating parameters of the first hidden layer:
  dW[0] = ( L[0].transpose() )*B[0];
  W[0] = W[0] - alpha*dW[0];
  db[0] = B[0].transpose().rowwise().sum();
  b[0] = b[0] - alpha*db[0];

}//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

[4] See commit 6432f1d517205f3920ab1b77bf224735fbeec819

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 implement exactly this, taking into account that the activation function is a $tanh(\cdot)$, and 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 is supposed to do.[5]
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     //
  ////////////////////////
  backup_t[nrefinements-1] == 0 ?
  cout<<"Total iterations = "<<accumulate(backup_t.begin(),
     backup_t.end(), 0)+niter<<"\n"<<"Cost functional on the
     training set = "<<cost<<endl :
  cout<<"Total iterations = "<<accumulate(backup_t.begin(),
     backup_t.end(), 0)<<"\n"<<"Cost functional on the
     training set = "<<cost<<endl;

}//End of the train function
```

where `std::accumulate()` is used to output 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, so e.g. 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. The last check, based on the conditional operator, is instead needed: it distinguishes

[5]Optimization done 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)`.

between the case in which the training phase ends for having reached convergence or the maximum number of iterations.

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() );
  }
  Ltest = ( Atest*W[nlayers-2] ).rowwise() + b[nlayers-2].
     transpose();

  //Computation of the L2 relative error:
  double numerator=(Data.col(1)-Ltest).norm();
  double denominator=(Data.col(1).norm() + Ltest.norm())/2.;

  return make_pair(Ltest, (numerator/denominator) );
}
```

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.

Final remarks: as can be seen, the code, despite being 1D, can be extended to a n-d interpolator with no conceptual effort (of course, this potentially introduces a great number of choices for the coder, and so of a lot of time-consuming new-parameters tuning). The very simplest way to do so would be to make the first layer too to have a `W` (and a `b`, and so even backprop variables too), basically to do a (weighted) average of every input point

$\boldsymbol{x} = (x_1, x_2, x_3 ... x_n)$, in such a way that the second layer takes 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)`.
Of course even 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 wouldn't be satisfactory enough otherwise.
Moreover 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 we are increasing the computational effort, but from the other one less iterations might be needed to reach convergence, since we are keeping the richness of the data "alive" for more time.
It worth considering that, especially in DNN programming, the more one enrich the code and exponentially more tuning is needed, because not only new choices have to be tested, but even their combinations with the already existing hyper-parameters. It's thus clear that at the moment that would go over the aims of the present project as outlined in the introduction.

3.3 The Optimizers class templates

3.4 Write_set and data

3.5 The main

Chapter 4

Examples

4.1 Building instructions

4.2 Reconstruction of a wave packet

Conclusions and further developments

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 ...
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-LeiBa] 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*

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

[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*

[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*