Deep Learning for PDEs

Report of the joint APSC-NAPDE courses project

Paolo Joseph Baioni*

March 25, 2020

*paolojoseph.baioni@mail.polimi.it

Contents

Introduction **2**

1 Neural Networks and Deep Learning **4**
1.1 Architecture and operation of a Deep Neural Network 4
1.2 Further insights 10
1.2.1 Regularization 11
1.2.2 Optimization 12

2 Application to PDEs **17**
2.1 The Poisson-Dirichlet problem 17
2.2 Some more in-depth theoretical results 17

3 Implementation of neural-net **18**
3.1 Directory tree 18
3.2 The Neural Network class 19
3.3 The Optimizers class templates 19
3.4 Write_set and data 19
3.5 The main 19

4 Examples **20**
4.1 Building instructions 20
4.2 Reconstruction of a wave packet 20

Conclusions and further developments **21**

Appendix **22**

Bibliography **23**

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: from the possibility of writing mesh-free algorithms to that of solving data-learned equations, whose explicit functional form is unknown; 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 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

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:
To gain an intuition of what a node is really doing, it's useful to get a more

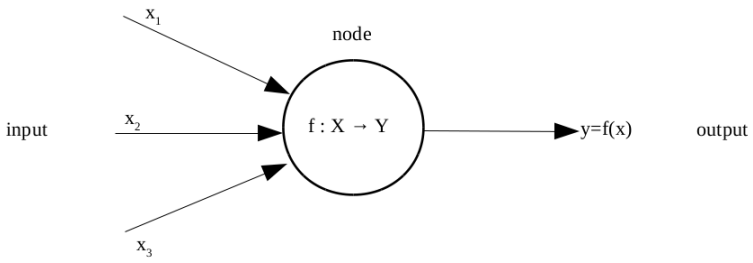

Figure 1

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.

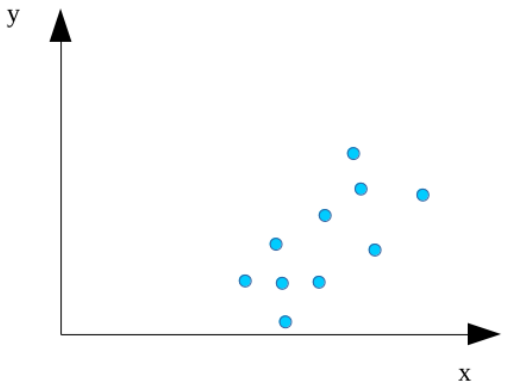

Figure 2

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.

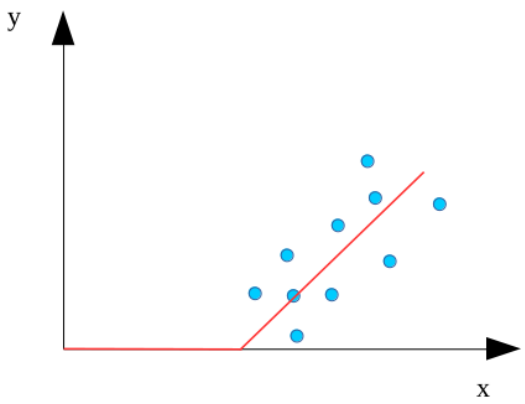

Figure 3

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.

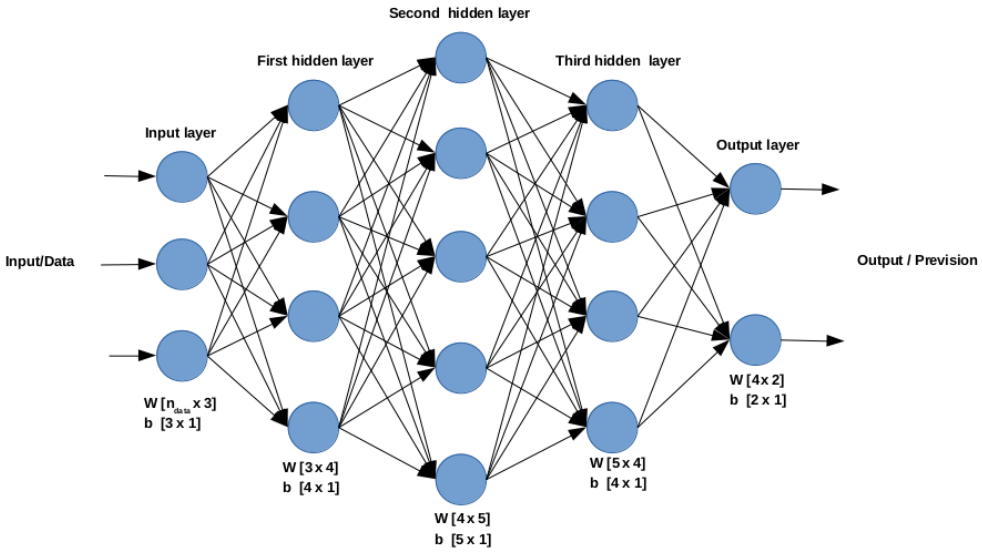

Figure 4

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 net-

[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.

works: 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 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.

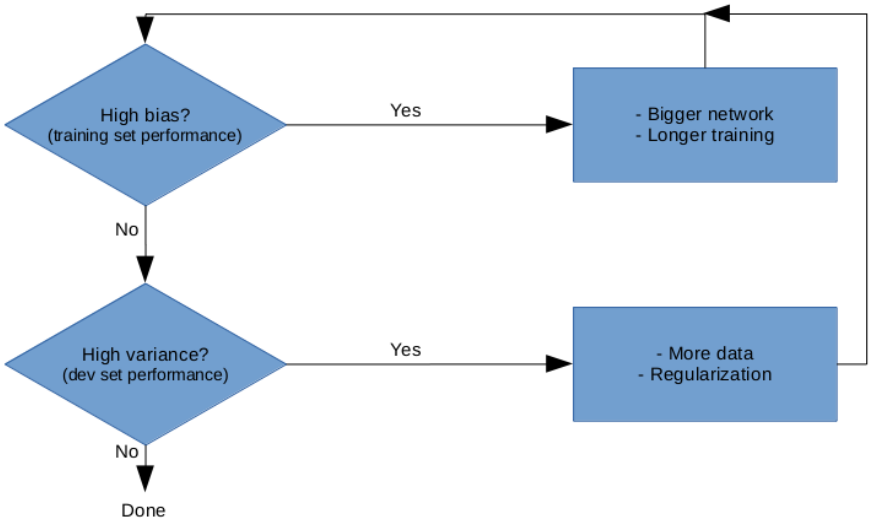

Figure 5

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

2.1 The Poisson-Dirichlet problem

2.2 Some more in-depth theoretical results

Chapter 3

Implementation of neural-net

In this chapter an in-depth view of a Deep Learning C++ program, called *neural-net*, is given.
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 author's 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's find:

- 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 there are the images included by the TeX file;
- the *include* folder, containing the headers *GetPot.hpp*, an utility used for parsing parameters file and command line option, *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 trace the file extensions which are not being pushed to the remote, mainly compilation files, executables and comments file.

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

3.2 The Neural Network class

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

Bibliography

[APSC] Advanced Programming for Scientific Computing course lectures, 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*

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

[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*