University of Salamanca

Master's Degree Physics and Mathematics
End of Master's Project

High performance optimization algorithms for neural networks

Author:
Carlos Barranquero Díez

Tutors:
Vidal Moreno Rodilla
Francisco Javier Villarroel Rodríguez

September 3, 2019

Vidal Moreno Rodilla and Javier Villarroel Rodríguez, professors of the "Dept. de Informática y Automática " and, respectively, "Estadística", of the "Universidad de Salamanca", certify: That they have acted as tutors of Mr. Carlos Barranquero Diez during the studies for the Master's degree in Physics and Mathematics at the University of Salamanca, and that the memory entitled "High performance optimization algorithms for neural networks" that is presented here as a final Project has been satisfactorily carried out by Mr. Carlos Barranquero Diez.

Salamanca, September 3, 2019.

Acknowledgement
I would like to convey my warm acknowledgement towards Mr. Roberto López González, chief executive officer in Artificial Intelligence Techniques S.L.(Artelnics), for all his support and help during the undertaking of this project and the consideration shown during my master's degree studies.

Contents

1 Introduction **2**
- 1.1 State of the art . . . 2
- 1.2 Motivation . . . 4

2 Machine learning techniques **6**
- 2.1 Foundations of statistics and probability . . . 7
  - 2.1.1 Random variables and distributions . . . 7
  - 2.1.2 Expected value and moments . . . 9
  - 2.1.3 Common distributions . . . 10
  - 2.1.4 Estimators . . . 11
  - 2.1.5 Conditional probability distributions . . . 12
  - 2.1.6 Machine learning approach . . . 13
- 2.2 Optimization methods . . . 15
  - 2.2.1 Ordinary least squares . . . 18
  - 2.2.2 Gradient decent . . . 18

3 Neural networks **20**
- 3.1 Data set . . . 20
  - 3.1.1 Variables . . . 21
  - 3.1.2 Samples . . . 21
  - 3.1.3 Batches . . . 21
- 3.2 Neural network . . . 22
  - 3.2.1 Neuron . . . 22
  - 3.2.2 Neuron layer . . . 26
  - 3.2.3 Feed-forward architecture . . . 32
- 3.3 Loss index . . . 33
  - 3.3.1 Error term . . . 33
  - 3.3.2 Regularization term . . . 34
  - 3.3.3 Batch approximation . . . 35
- 3.4 Optimization algorithms . . . 35
  - 3.4.1 Traditional algorithms . . . 36

4 Batch optimization algorithms **39**
4.1 Stochastic gradient descent 41
4.2 Adagrad 43
4.3 RMSprop 44
4.4 Adadelta 44
4.5 Adam 45

5 Performance comparison **47**
5.1 Benchmark description 47
5.2 OpenNN vs TensorFlow 49
5.3 10 variables 10000 samples 50
5.3.1 Tensorflow performance 50
5.3.2 OpenNN performance 50
5.3.3 Results 51
5.4 100 variables 100000 samples 52
5.4.1 Tensorflow performance 52
5.4.2 OpenNN performance 53
5.4.3 Results 53
5.5 1000 variables 1000000 samples 54
5.5.1 Tensorflow performance 54
5.5.2 OpenNN performance 54
5.5.3 Results 54
5.6 Bath vs traditional algorithms 56

6 Conclusions **58**

List of Figures

3.1 Plots for different subsets depending on the size of the batch . 22
3.2 Perceptron neuron model. 23
3.3 Hyperbolic tangent. 25
3.4 Rectified linear. 26
3.5 Layer. 27
3.6 Layer example. 31
3.7 Feed forward architecture 32
3.8 Training process. 37

4.1 Batch optimization algorithms structure. 40

5.1 Rosenbrock data 47
5.2 Accuracy vs Epoch 50
5.3 Accuracy vs Epoch 51
5.4 Time vs Batch 52
5.5 Accuracy vs Epoch 53
5.6 Accuracy vs Epoch 53
5.7 Initial configuration TensorFlow 55
5.8 Initial configuration OpenNN 55

List of Tables

3.1 Split data set example 30

5.1 Computer features 49
5.2 Results 10 variables 10000 samples 51
5.3 Results 100 variables 100000 samples 54
5.4 Results 100 variables 100000 samples 55
5.5 Results 10 variables 10000 QuasiNewton vs Adam 56
5.6 Results 100 variables 10000 QuasiNewton vs Adam 57
5.7 Results 1000 variables 100000 QuasiNewton vs Adam 57

Abstract

The computational paradigm of neural networks is described, to a large extent, by a set of parameters called weights. For the supervised case, the combination of these weights with input data allow to predict output values. In order to quantify the accuracy of these predictions, it is usual to consider a function that measure the error, between the predictions and the actual output data (sometimes called experimental data).

From algorithmic approach, the objective of supervised neural networks consist to reach the minimum of the error function. Optimization or training algorithms bring about this process. One of the most widely used algorithmic techniques to train neural networks are those based on "Gradient Descent" methods. This technique changes the weights in small increments, considering the gradient of the error function. This information allow to find the optimal direction of the space of the weights (of very high dimension) to "descend" towards the global minimum of the function. This is not achievable in a single process, but a multitude of iterations are needed to find that absolute minimum. It should be noted that it is the dimensionality of the weight space (hundreds, thousands or more) that determines the viability of the solution that is proposed.

In the present work, new techniques for training algorithms derived from Gradient Descent will be developed; SGD, ADAM, RMSprop, Adagrad and Adadelta. These algorithms incorporate notable improvements over other more traditional ones, such as the optimization of one of the most important parameters of the procedure: the "learning rate" or the learning speed, as well as the batch architecture, that enable to train the neural network with a sets of data.

Chapter 1

Introduction

1.1 State of the art

In recent times, artificial neural networks (ANN) have become popular and useful in a wide variability of disciplines such as physics, engineering, or statistics. Scientists use neural networks as tools for finding solutions in highly complex problems. For instance, within the field of physics, ANN has been applied in astronomy for searching gravitational lenses [1], in optics for controlling the spectral phase of an ultrashort laser pulse [2] or even in particle physics to discover Higgs Boson at LHC [3].

Neural networks together with other techniques such as support vector machines, decision trees or Bayesian networks, make up a larger group called "machine learning" [4]. Machine learning techniques extract complicated relations within data, making use of mathematical techniques like statistics, optimization, probability and numerical analysis, to create models, that reproduce the behavior of the study [5].

In order to create these models, three paradigms of learning can be applied; supervised, unsupervised and reinforced learning. Supervised learning consist in feeding the model with the correct data beforehand, in this way the model is able to "learn", to later predict accurate results in situations that it had not seen before, but are quite similar to what it had learned[6]. Unsupervised learning, consist in just using input data to feed the model, without any reinforcement about what the model should predict; as a result it extracts patterns from the input data, joining those that have similar features.

Reinforced learning consist in making decisions as a consequence of how the model acts, that is, correct the model when it behaves incorrectly, or otherwise reinforce it [6].

Neural networks stand out in machine learning techniques, since they offer a good balance between solutions and suitable implementation. Supervised learning is the learning paradigm used by ANN, thus, they are capable of learning from experience, storing knowledge and then applying this knowledge to make predictions.

Basically a neural network is a functional algorithm able to model a part of reality from empirical observations of it. As it is often the case with most advanced behaviors and structures, the complexity of neural networks emerges from the interaction of many simpler parts working together. This elemental parts are called neurons, and are quite similar to biological neurons [7]. Artificial neuron has input connections through which they receive external stimulus (input values), and a set of internal parameters called synaptic weights and bias which interact with the input signals to provide an output value. The neural network's parameters values are chosen in terms of providing an accurate response giving an specific input values.

The process for finding the optimal neural networks parameters is called training or learning process, and is carried out through mathematical optimization techniques. One of the most famous first-order optimization algorithm is 'Gradient descent' [8], also know as steepest descent. This algorithm calculates the minimum of a function by means of the derivatives. In the case of neural networks, the function to be minimized is called Loss Index or Cost function. This function marks the error between the model and the real data, so the goal is to minimize this function using the neural network's parameters derivatives.

Some open source libraries of machine learning are OpenNN [9], TensorFlow [10] or Scikit-learn [11], all of them implement a wide variety of mathematical tools such as statistical and optimization methods, which allow developers to create very robust machine learning models that meet the needs of the moment. The most known and complete library open source is TensorFlow. This library provide cutting edge technology and is available for any developer. TensorFlow was developed by the Google Brain team for internal Google use, and was released under the Apache License 2.0 on November 9, 2015.

1.2 Motivation

Machine learning has become a widely used tool in physics and technology, and in particular, in particle physics and astronomy or any other field where large amounts of data need to be combed and analyzed to extract and dissect useful information from "noise".

Within particle physics, applying machine learning to shift through the data for finding a particle, such as "the Higgs boson"(one of the most important discoveries in recent times), allows us focus on interesting results within a stack of trillions data, ushering in faster discoveries and less wasted time.

Another remarkable breakthrough in physics, specifically in astronomy field, was to take the first image of a black hole. To get this achievement, it was necessary to set up a network of eight linked telescopes around the world, (no single telescope is powerful enough to image the black hole), together, they formed the Event Horizon Telescope.

The information gathered by these telescopes was too much to be sent across the internet. Instead, the data was stored on hundreds of hard drives that were flown to central processing centers in Boston, US, and Bonn, Germany, to assemble the information.

The main algorithms used to obtain the image, were carried out with the latest machine learning techniques. For this particular case convolution neural networks (CNN) were used, quite efficient in recognizing real world objects.

Nonetheless, the most important application that machine learning can offer, is to solve the biggest challenge facing by humanity, the climate change [12]. Some solutions that these techniques can provide to tackle this issue are; Improve electricity predictions, create better estimates of energy consumption, improve climate predictions, measure $C0_2$ emissions, etc.

Solutions offered by machine learning can improve the quality life of people, as well as our understanding of reality. However, a very large amount of data are needed to be able to rely on machine learning models.

The optimization techniques used to analyze Big data, stand out for their simplicity, since they perform simple operations over different sets of the total data. However, it is crucial that the implementation of these algorithms be as optimal as possible, as the time to reach a solution can vary between

hours to days or even to such long times that it is not profitable use these techniques.

In the present work we will develop and implement in OpenNN, high performance optimization algorithms for neural networks, with the aim of being able to provide solution with in massive amounts of data, in the shortest possible time.

Chapter 2

Machine learning techniques

The basic premise of machine learning is to build algorithms that can receive input data and use statistical analysis to predict an accurate output. There are a wide range of applications of machine learning, for instance, to self-driving cars, speech recognition, or even to improve the understanding of the human genome.

This techniques, albeit usually hidden, has become one of the mainstays of the current technology. Many a chairperson like Google CEO Sundar Pichai, also thinks that this is the best way to make progress towards human-level AI.

> *Machine learning is a core, transformative way by which we are rethinking everything we are doing.*

This so called learning procedure is nothing more than a standard problem of optimization carried out through mathematical techniques. This may seem a priori a simple task; however, picking the correct model for each situation is usually a non-trivial problem.

There are some mathematical techniques that provide useful insights to determine what model is more appropriate to our problem. Probability, statistics and estimation theory, are the main techniques applied to data in order to obtain this useful information.

2.1 Foundations of statistics and probability

Probability is the mathematical theory that studies randomness. As such it is a language naturally adapted to quantify uncertainty, and is the perfect tool to summarize our uncertainty about a issue, typically due to lack of data. Probability allows to compress much of the variability of our reality, being easier to manage the information that we receive from the environment and making possible to infer from these information the patterns of future behavior.

Our brain applies similar systems to build models and thanks to that, we have capabilities such as conceptualizing, predicting, generalizing or learn. Discovering what these models are, is one of the basic objectives of the field of machine learning.

Data are the main source of any machine learning mode. Usually they are collected within variables, and each of this variables contain part of the global information. However, as it is in the majority of real the cases this data are a sample of a greater population. So it is possible to resort to probabilistic techniques to achieve data insight before feeding it into the model.

2.1.1 Random variables and distributions

A random variable X is a function that assigns a numerical value to the result of a randomized experiment, or outcome. In other words, is a mapping from a set of outcomes of an experiment to a set of real numbers.

For the case of machine learning models, the random variables correspond to the variables contained in our dataset, which represent a characteristic of reality. Example of random variables include the temperature recorded by a thermometer or the color of a pixel recorded in an image.

Definition. Given the triple (Ω, A, P) associated with a random experiment, where Ω is the sample space and A is collection of all subsets of Ω ($A \subset \Omega$). An application $X : \Omega \to \mathbb{R}$ is a random variable, if and only if, for all interval I $\subset \mathbb{R}$, the anti-image of I for X, $X^{-1}(I) \in A$, belongs to A.

$$X : \Omega \to \mathbb{R}$$
$$\omega \to X(\omega)$$

Random variables are characterized by describing all probabilities that the value taken by X belongs to I: X$\in I$, for any interval I.

Definition. For all x $\in \mathbb{R}$, the function $F(x)$ called distribution function of the random variable X is defined as:

$$F(x) = P(\left\{X^{-1}(\text{-}\infty, \text{x}]) = P\left\{\omega \in \Omega | X(\omega) \leq x\right\}.$$

There are five properties that characterize the distribution functions that all of them verify:

- $0 \leq F(x) \leq 1$
- F is a monotonically increasing function $x_1 \leq x_2 \Rightarrow F(x_1) \leq F(x_2)$
- F is continuous for the right: $\lim_{h \to 0^+} F(x + h) = F(x)$
- $F(-\infty) = \lim_{h \to -\infty} F(x) = 0$
- $F(+\infty) = \lim_{h \to \infty} F(x) = 1$

Random variables are classified into different types depending on the characteristics of their distribution function: a random variable is said to be continuous if has a continuous distribution function, or discrete if has a discrete distribution function.

A random variable is discrete when it takes a finite or demunerable infinite number of values. The values $(x_1, ..., x_i)$ with their probabilities $(p_1, ..., p_i)$, determines the probability distribution of a discrete random variable X which analytically is represented by the mass function, defined as:

$$P(X = x) = \begin{cases} p_i & if x = x_i, i = 1, 2, 3... \\ 0 & otherwise \end{cases}$$

If the mass function P is known, the distribution function can be obtained from:

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} P(X = x_i) = \sum_{x_i \leq x} p_i \tag{2.1}$$

On the other hand a random variable is continue if the distribution function associated with $F(x)$ is continuous. Here the probability associated to a continuous random variable is distributed along the line but it does not concentrate at any mass point. It can be represented in terms of a density $f(x)$.

$$F(x) = \int_{-\infty}^{x} f(x)dx$$

Where f is positive and the integral over the line equals 1.

2.1.2 Expected value and moments

Moments α_k allow to characterize and obtain useful information from random variables.

Definition. The moments of a random variable X are defined by:

$$\alpha_k = E[X^k] = \begin{cases} \sum_{i=1}^{n} x_i^k p_i = \sum_{i=1}^{n} x_i^k f(x_i) & if\ X\ is\ discrete \\ \\ \int_{-\infty}^{\infty} x^k f(x)dx & if\ X\ is\ continuous \end{cases}$$

The expected value μ corresponds to the first moment k = 1, also know as average or mean, it is a number that provides the average value of a random variable.

The variance σ^2 is associated with the second moment k = 2. It measures the amount of variation or dispersion of a set of data from the mean. The square of the variance is call standard deviation an is represented as σ.

The skewness is related to the third moment, k = 3, and it provides a measure of the lopsidedness of the distribution. A distribution that is skewed to the

left (the tail of the distribution is longer on the left) will have a negative skewness. On the other hand a distribution that is skewed to the right, the tail of the distribution is longer on the right, will have a positive skewness.

The kurtosis K corresponds to the fourth central moment k = 4, and provides a measure of the heaviness of the tail of the distribution, compared to the normal distribution of the same variance.

High-order moments are moments beyond 4th-order. As with variance, skewness, and kurtosis, these are higher-order statistics, involving non-linear combinations of the data, and can be used for description or estimation of further shape parameters.

2.1.3 Common distributions

There are multitude of discrete and continuous distribution functions. For instance, Bernuilli or Binomial are cases of discrete distribution, while gamma or beta are cases of continuous distribution functions. To analyze all of them is outside the scope of this work, and for this reason we will focus on the most important continuous distribution function, the normal distribution.

The normal distribution (also know as gaussian) is often used for statistical inferences since many statistical observables are found to be approximately normally distributed. Indeed, a variety of natural phenomena follow a normal distribution or can be transformed trough 'Central limit theorem' to follow a normal distribution.

The normal density function for a continuous variable is defined as:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.2}$$

Parameter μ is the mean (also know as expected value or first moment) and σ^2 is the variance.

The central limit theorem is one of the most powerful theorems of probability. It proves that under mild assumptions the describing law of many statistical observables under a long number of trials is normally distributed. The theorem emphasis in the fact that in some situations, when independent random variables are added, their properly normalized sum tends toward a

normal distribution, even if the original variables themselves are not normally distributed.

Theorem 2.1.1 (TLC)**.**
Denote by X_i independent random variables with means μ_i and standard deviation σ_i. Then as m tends to infinity:

$$Z_m := \left[\sum_{i=1}^{m} \sigma_i^2\right]^{-1/2} \left[\sum_{i=1}^{m} X_i - \mu_i\right]$$

converges to a normal distribution with zero mean and unit variance.

2.1.4 Estimators

An estimator($\hat{\theta}$) is a rule for calculating a parameter of interest based on observed data. It can be understood as a function that maps the sample space to a set of sample estimates. Commonly is symbolized as a function of a random variable with the expression $\theta(X)$.

Some estimators properties are:

Bias

The difference between the expected value (or mean) of the estimator and the true value of the parameter to be estimated is called bias of an estimator. It is desirable that an estimator be unbiased, that is, his bias be null.

The arithmetic mean of the sample is an unbiased estimator of population mean since, its expectation (expected value) is equal to the average of the population.

Suppose $X_1, ..., X_n$ are independent and identically distributed random variables with expectation μ and variance σ^2. If the mean are defined as:

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Then:

$$E[\bar{X}] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}E\left[\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} E[X_i] = \frac{1}{n}\sum_{i=1}^{n} \mu = \frac{1}{n}n\mu = \mu$$

Efficiency

An estimator is more efficient or more accurate than another estimator, if the variance of the first is smaller than the second.

$$Var(\hat{\theta}_1) < Var(\hat{\theta}_2)$$

The efficiency of the estimators is limited by the characteristics of the probability distribution of the sample from which they come. The Cramér-Rao theorem determines that the variance of an unbiased estimator of a parameter is, bounded below by:

$$var(\hat{\theta}) \geq \frac{1}{E\left[\left[\frac{\partial}{\partial\theta} log f(x;\theta)\right]^2\right]}$$

Here $f(x;\theta)$ is the probability density function of the underlying random variable X which depends on the parameter θ.

Consistency

Consistency is a estimator property that means that as the sample size n grows, the value of the estimator tends to the value of the parameter.

$$E[\hat{\theta}] \to \theta \quad when \quad n \to \infty \tag{2.3}$$
$$Var[\hat{\theta}] \to 0 \quad when \quad n \to \infty \tag{2.4}$$

2.1.5 Conditional probability distributions

It is often interesting to know the probability of an event when another event has already been observed. In probability theory and statistics, Bayes'

theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event.

Mathematically Bayes' theorem follow:

$$P(Y \mid X = x) = \frac{P(X \mid Y = y).P(Y)}{P(X = x)}$$

- $P(Y \mid X = x)$: is a conditional probability, the likelihood of event Y occurring given that x has occurred.
- $P(X \mid Y = y)$: is also a conditional probability, the likelihood of event X occurring given that y has occurred.
- P(X) and P(Y): are the probabilities of observing X and Y independently of each other; this is known as the marginal probability.

2.1.6 Machine learning approach

These statistical and probabilistic concepts are the basis of any machine learning model. As we discussed earlier different models can be created depending on the learning paradigm chosen.

Supervised, unsupervised, and reinforced learning, infer what is the behavior of object of study, with a risk of measurable error, in terms of probability.

Supervised learning

The goal in supervised machine learning is to learn from labeled data. This means that for some inputs X, the desired outputs Y are known.

Probability allows to create mapping from X to Y. This implies that you could know the probability distribution over possible values of Y for a given observed sample; $P(Y|X = x)$.

Machine learning algorithms that find this distribution are called discriminative. For example, imagine that some relevant information about a particle, such as its mass or charge, is known. Can you discriminate between possible

particles and guess what it is?. Discriminative models do. Some examples of these models are; logistic regression, neural networks or random forests.

On the other hand, it also possible to obtain $P(X|Y = y)$, this means, the probability distribution over inputs given a target. Algorithms that perform this task are called generative. For instance, given an specific particle, can you describe the possible features of it?. Generative models accomplish this task, and generally they are used in unsupervised learning. However, within supervised learning they are useful as well.

Generative models are able to determinate $P(X|Y = y)$. If this probability is combined with P(Y) and P(X), in our particle example, the probability of any specific particle and the probability of any specific configuration, respectively, it is possible to obtain $P(Y|X = x)$ using Bayes' Rule.

Notice that supervised learning stands out in machine learning for its strong connection between probability regarding classification problems.

Also it is possible to learn a mapping from X to Y which is not in the form of a probability distribution. We could fit a deterministic function f to our training data such that $f(X) \approx Y$. However, the main advantage of having a probabilistic model is that it quantifies uncertainty and the regular function does not.

Unsupervised learning

Unsupervised learning is a broad set of techniques for learning from unlabeled data, where just X are known. The general objective is to infer a function to describe hidden structures from unlabeled data.

Characterizing the distribution of unlabeled data is useful for many tasks. One example is anomaly detection. If we learn P(X), where X represents normal bank transactions, then we can use P(X) to measure the likelihood of future transactions. If we observe a transaction with low probability, we can flag it as suspicious and possibly fraudulent.

Dimension reduction, (Principal component analysis), is the other main area of unsupervised learning. High dimensional data takes up memory, slows down computations, and is hard to visualize and interpret. One can think of this problem as finding a distribution in a lower dimensional space with

similar characteristics to the distribution of the original data.

Reinforcement Learning

Reinforced learning consist in making decisions as a consequence of how the model acts, that is, correct the model when it behaves incorrectly, or otherwise reinforce it. The goal is training artificial agents, that will be able to perform well specific tasks. Probability is used in reinforcement learning for several aspects of the learning process. The agent's learning process often revolves around quantifying the uncertainty of the utility of taking one specific action over another.

2.2 Optimization methods

Linear regression

One of the most relevant aspects of statistics is the relationship analysis or dependence between variables. It is often interesting to know the effect that one or several variables can cause on another, and even predict to a greater or lesser degree values in one variable from another. Regression methods study the construction of models to explain or represent the dependency between a independent variable (Y) and the dependent variable/s, X. In this subsection we will address the the linear regression models.

The simple lineal regression model can be defined mathematically in two-dimensional space as the line that best fits the bidimensional data:

$$y = w_0 + w_1 x$$

On the one hand we have the independent term w_0, it shows how high the straight line cuts to the Y axis, on the other hand we have the slope w_1 that graphically defines the inclination of the line and conceptually defines what is the relationship between the input and target variables. This simple model would work in the case of having a single input variable. However, this model is very limited due to its lack of generalization. A single variable cannot reproduce the behavior of a reality.

Multiple linear regression are models with higher complexity since more than one input variable are involved, for instance with two input variables the model can be defined mathematically in tree-dimensional space as a plane that best fits the tridimensional data:

$$y = w_0 + w_1 x_1 + w_1 x_2$$

If more variables are added to the model, hyperplanes we will used to fit data in multidimensional spaces. Notice, that each of this variables represent a feature of the reality, in fact, the greater the number of features we have, the greater the size of the space has to be modeled.

This linear combination can be represented as:

$$\begin{aligned}
y_1 &= w_0 + w_1 x_{11} + w_2 x_{12} + ... + w_n x_{1n} \\
y_2 &= w_0 + w_1 x_{21} + w_2 x_{22} + ... + w_n x_{2n} \\
y_3 &= w_0 + w_1 x_{31} + w_2 x_{32} + ... + w_n x_{3n} \\
y_4 &= w_0 + w_1 x_{41} + w_2 x_{42} + ... + w_n x_{4n} \\
&\vdots \\
y_m &= w_0 + w_1 x_{m1} + w_2 x_{m2} + ... + w_n x_{mn}
\end{aligned}$$

This way of representing all possible linear combinations is not really optimal. The most comfortable way to represent all combination of variables for each of our data is in vector notation. Let's denote by X the inputs data matrix.

$$\boldsymbol{X} = \begin{bmatrix}
1 & x_{11} & x_{12} & x_{13} & ... & x_{1n} \\
1 & x_{21} & x_{22} & x_{23} & ... & x_{2n} \\
1 & x_{31} & x_{32} & x_{33} & ... & x_{3n} \\
1 & x_{41} & x_{42} & x_{43} & ... & x_{4n} \\
\vdots & \vdots & \vdots & \vdots & & \vdots \\
1 & x_{m1} & x_{m2} & x_{m3} & ... & x_{mn}
\end{bmatrix}$$

Each column of X represents a characteristic of the input data (denoted with n subindex), and each row represent the measurements of these input variables(denoted with m subindex).

Likewise, we can redefine the independent variables $(y_1, ..., y_m)$ using a vector notation.

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_m \end{bmatrix}$$

The parameter $(w_1, ..., w_n)$ can be redefined in vector notation as well as:

$$\boldsymbol{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{bmatrix}$$

Notice that with this new redefinitions we can transform all lineal combinations to:

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} \tag{2.5}$$

Equation 2.5, brings us significant improvements, since now all operation can be programed in the GPU as a matrix and vector products.

Optimization methods are techniques that find the combination of parameters $\boldsymbol{w}$ that best fits the model's predictions to the real data. In order to do that is necessary to introduce the concept of error. The error provides a measurement of the model accuracy. There are multiple error functions, for instance, one of the most famous is the Mean Square Error (MSE), it calculates the average square error between the model outputs and the target an is defined as:

$$MSE = \frac{1}{q}\sum_{m=1}^{q}(y_m - t_m)^2$$

q the number of samples, t_m the actual value of the independent variable, and y_m the predictions of the model ($y_1 = w_0 + w_1 x_{11} + w_2 x_{12} + ... + w_n x_{1n}$).

There are two main methods, to get optimum combination of parameters $\boldsymbol{w}$ the first one is called least squares and the second one gradient descent.

2.2.1 Ordinary least squares

Ordinary least squares find the minimum of the error function by matching the derivative of the function to 0, the mean square error in vectorial form can be expressed as:

$$(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w})$$

If we multiply the parentheses, we reach at the following expression:

$$\boldsymbol{y}^T\boldsymbol{y} - \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{y} - \boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w} + \boldsymbol{w}^T\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}$$

Deriving the expression and matching to zero, we have:

$$-2\boldsymbol{X}^T\boldsymbol{y} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} = 0 \tag{2.6}$$

Finally we obtain:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y} \tag{2.7}$$

Using the method of ordinary squares provides a formula that directly calculates the value of the parameters $\boldsymbol{w}$, however this does not always happen since if we work with other models or other error functions, we will not always be able to find the minimum of the error function analytically. Thus if we apply the equation 2.7, we have to perform $(\boldsymbol{X}^T\boldsymbol{X})^{-1}$ and this operation can be very inefficient for any computer.

2.2.2 Gradient decent

Gradient descent is an iterative method that gradually brings us closer to the minimum of the error function, this means that in every iteration step

we use information about the gradient $\boldsymbol{g}$ at the current point. As a result we obtain the optimum parameters associated with the smallest error.

Initially the vector of parameters ($\boldsymbol{w}$) are initialized randomly using a probability distribution, then the method calculates the parameter's derivatives with respect the error function to obtain the gradient vector.

This vector provides the direction of the maximum variation of the error, therefore the algorithm takes the opposite direction and advances an step. The amount to be advanced is called the learning or training rate, and plays a crucial role in the convergence of the algorithm. Choosing the learning rate too low will result in a slow convergence to the minimum, and choosing the learning rate too high will overstep the minimum, causing oscillation around it.

The recurrence rule is as follows:

$$\boxed{\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \boldsymbol{g}_i.\eta} \tag{2.8}$$

$\boldsymbol{w}$ is the parameter's vector and the subindex i marks the iteration number, g is the gradient vector and η is the learning step.

Chapter 3

Neural networks

In the previous chapter we talked about some of the most important techniques of machine leraning. In this chapter we will focus on neural networks, and will explain the fundamental concepts to understand them. Also the batch architecture implemented in this work, will be explaned.

Intuitively we can think neural networks as a machine learning models designed to imitate the way that the human brain works [13]. Creating models with neural networks provides advantages over other techniques. Perhaps the greatest of which is the property of universal approximation [14]; which says there will always be an ideal neural network able of generate a function that approximates any set of data with an error as small as desired.

3.1 Data set

Data is the source from which models are created, that is, the domain space of our problem.

A data set is a collection of data used to fit the parameters of the neural network [15]. The most common way to visualize the data set is through a matrix, where the columns represent variables (inputs or targets), and rows represent the current measurements of these variables.

3.1.1 Variables

Variables, also called attributes or features, might represent physical measurements (temperature, velocity...), personal characteristics (gender, age...), marketing dimensions (frequency, monetary...), etc. Variables correspond to dataset's columns and can be separated as inputs and targets.

- Inputs are independent variables which feed the model.
- Target are dependent variables that provide the correct measure that the model has to predict for given a sample.

Within supervised learning, target variables play a crucial role because these will be the values from which the model will learn.

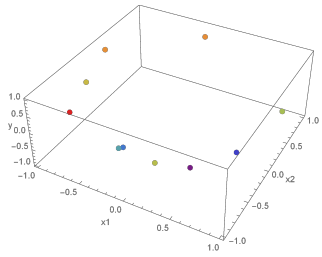
3.1.2 Samples

Many times, it is not useful to create a model that just memorize a set of data. The objective consist in design a neural network able to generalize and predict a correct value with data that has never seen. To achieve that, we divide the data set into three different subsets:

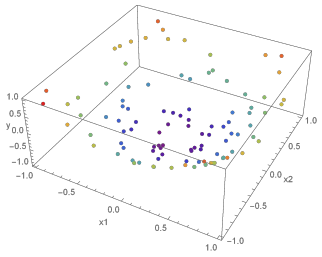
- Training instances are used to build all models. Each of these models will have different architectures but the data used for build them are the same.
- Selection instances are used for choosing the model with best generalization properties.
- Testing instances are used to validate the functioning of the model.

3.1.3 Batches

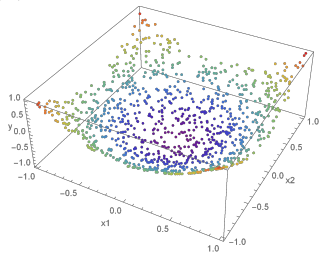
When the amount of data is considerably large, it is convenient to divide the data set into subset of data call batches. This lots stores part of the general information; indeed probabilistic techniques can be applied within batches if we treat them as samples of a greater population.



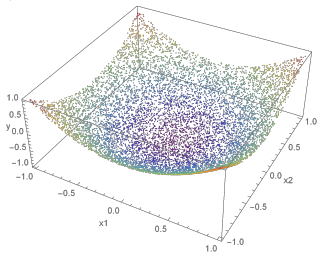
(a) Batch size : 10 samples



(b) Batch size : 100 samples



(c) Batch size : 1000 samples



(d) Batch size : 10000 samples

Figure 3.1: Plots for different subsets depending on the size of the batch

In the following sections we will explain how this techniques allow batch optimization algorithms to converge faster to a given value.

For now, it will be enough to have an intuitive idea of what this concept represents. In the example 1, it can be seen that as the size of the data set increases, more information is available on the current batch.

Example 1. *Consider a data set of 10000 samples, figure 3.1 shows information contained in each batch according to its size.*

3.2 Neural network

3.2.1 Neuron

Neural networks are composed of basic units called neurons. It is possible to create models with just a single neuron. However they are very limited.

Neuron's models are able to solve problems that do not require high complexity. Indeed they are quite similar to regression models, since under certain

conditions the result of having one input with one neuron is the simple model regression, and the result of having multiple inputs with one neuron is the multiple linear regression model.

Figure 3.2 shows the internal structure of a neuron. Basically, a neuron transforms a set of inputs, into an output value, making use of internal parameters. These parameters are a collection of synaptic weight $W = \{w_1, ..., w_n\}$ and an independent variable called bias b.

n index indicates the number of variables. For this model the number of parameters to be modeled will be the number of variables plus the bias.

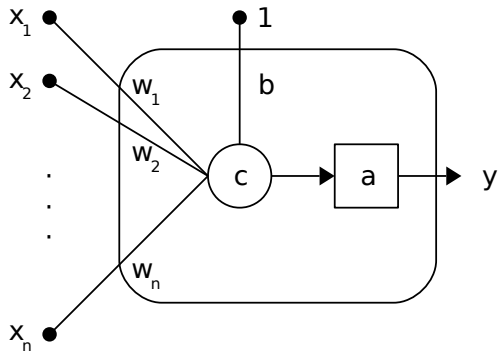


Figure 3.2: Perceptron neuron model.

The basic operation of a neuron is called 'Calculate outputs'. This function is represented in terms of composition of two other more elementary functions that are; combination function and activation function.

$$y = a \circ c. \tag{3.1}$$

Combination function

The function that combines the inputs entries with the neuron parameters to obtain a value is called the combination function. There are two ways to calculate the combination in a neuron:

- Using a vector: the input to the neuron is a vector, which corresponds to a unique sample (a row within the data set).

- Using a matrix: the input to the neuron is a matrix, which corresponds to a batch (a set of rows within the data set).

Combination function (vector form)

Given $X_1, ..., X_n$ random variables with their respective collection values $S_1, .., S_n$, being S = Support(X), and $S = \{x_1, ..., x_j\}$

The combination function in a neuron transforms, a single measure of the n random variables into a new value:

$$c = b + \sum_{i=1}^{n} w_i x_{ij}. \tag{3.2}$$

Notice, that the combination function in vector form is applied just for a single measure (a row in our dataset) of a set of variables.

Index i denotes the number of random variables, and j index denotes the values that these variables could take. Therefore for this particular operation j must take a single value, making c an scalar.

Combination function (matrix form)

If the operation is performed with a set of measures, that is to say instead of one single measure of several variables (a row within the dataset), with a set of measures of several variables (a set of rows within a dataset), the combination function must be redefined.

$$\mathbf{C} = b + \sum_{i=1}^{n} w_i x_{ij}. \tag{3.3}$$

Notice, that now C is capital and bold, this means that the $\mathbf{C} = \{c_1, ..., c_j\}$ is a set composed of j elements, and each of them represent the combination for a given set of measures that variables can take.

Activation function

For either artificial and biological neurons, the model does not simply output the bare input it receives. Instead, there is one more step, called activation function (sometimes called transfer function).

The activation function was a fundamental part of the development of neural networks, since it allowed introduce nonlinearities which solved highly complex problems.

Modern neural network use non-linear activation functions. The reason is because these functions allow the model to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality.

Almost any process imaginable can be represented as a functional computation in a neural network, provided that the activation function is non-linear.

Non linearity is what differentiates the perceptron (elementary neural network), and the linear regression model. Without an activation function both models would be the same. A simple polynomial of one degree, unable to solve simple no linear problems, such as classify the output of a XOR logic door.

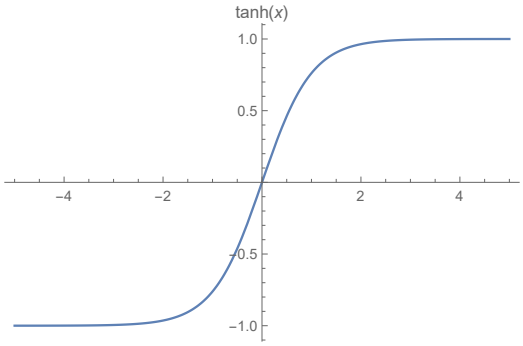


Figure 3.3: Hyperbolic tangent.

The tanh function is the most popular and oldest activation function. It is defined as:

$$f(x) = \frac{2}{1 + e^{-2x}} - 1 \tag{3.4}$$

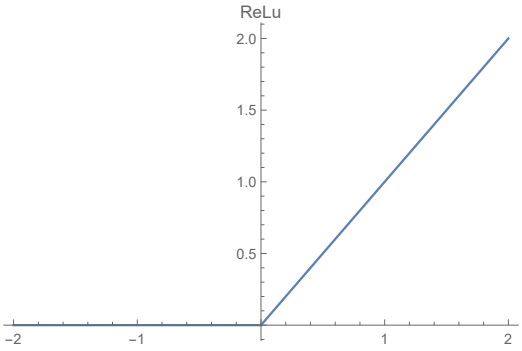


Figure 3.4: Rectified linear.

And rectified linear is defined as:

$$f(x) \begin{cases} 0 & if x < 0 \\ x & if x \geq 0 \end{cases}$$

Choosing the activation function depends on the problem to be solved, if one knows the function to be approximate has certain characteristics, you can choose an activation function which will approximate the function faster leading to faster training process.

When one does not the nature of the function to be approximated then ReLu is the best option.

Once the activation function is applied, this value will be the output of the neuron.

3.2.2 Neuron layer

Most neural networks, even biological neural networks, exhibit a layered structure [16] [17]. In this work layers are the basis to determine the architecture of a neural network. Figure 3.5 shows the internal structure of a layer:

The procedure to obtain the output of the layer is reduced to do the same as for the neuron. The main difference is that now the elements in W, are not just the synaptic weights of a single neuron, but W is made up of the

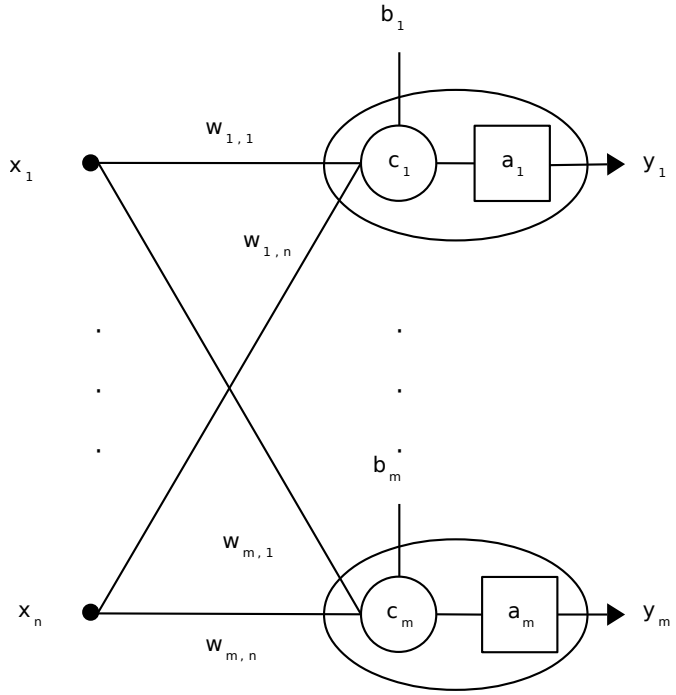


Figure 3.5: Layer.

weights of all the neurons that compose the layer. For a better visualization, the elements of the set W are shown in matrix form:

$$\mathbf{W} = \begin{pmatrix} w_{1,1} & \cdots & w_{1,n} \\ \vdots & \ddots & \vdots \\ w_{m,1} & \cdots & w_{m,n} \end{pmatrix}. \tag{3.5}$$

n index represents the number of random variables involved, (features of our dataset), m subindex represent the number of neurons that make up the layer.

Layer combination

As in a neuron, in a layer it is also possible to calculate the combination in two ways:

- Using a vector: the input to the neuron is a vector, which corresponds to a unique sample (a row with in the data set).
- Using a matrix: the input to the neuron is a matrix, which corresponds to a set of samples(a batch).

Layer combination (vector form)

The layer combination function transforms, a single measure of the n random variables into a set of outputs:

$$\mathbf{C} = b + \sum_{i=1}^{n} w_{mi} x_{ij}. \tag{3.6}$$

In equation 3.6, only one measure is computed at once, thus j take a single value

Notice, that the layer combination function in vector form is applied just for a single measure (a row in our dataset) of a set of variables.

Index i denotes the number of random variables, index m denotes the number of neurons that compose the layer, and j index denotes the values that the these variables could take. Therefore in this particular operation j must take a single value, this means that the $\mathbf{C} = \{c_1, ..., c_m\}$, is a collection composed of m elements of which are the combination of each neuron for a given measure.

Layer Combination (matrix form)

If the operation is performed with a set of measures, this means instead of one single measure of several variables (a row within the dataset), with a set of measures of several variables (a set of rows within a dataset), the combination function must be redefined.

$$\mathbf{C} = b + \sum_{i=1}^{n} w_{mi} x_{ij}. \tag{3.7}$$

In equation 3.7, j subindex indicates the number of measures that are computed (length of the batch)

Notice, that C, is now a larger collection. The elements of this group are the combinations of each measure j with each neuron m. If C is represented in matrix form in order to visualize its elements:

$$\mathbf{C} = \begin{pmatrix} c_{1,1} & \cdots & c_{1,j} \\ \vdots & \ddots & \vdots \\ c_{m,1} & \cdots & c_{m,j} \end{pmatrix}. \tag{3.8}$$

The first row of this matrix is the combination for neuron 1 of all the samples that make up the batch, likewise, the m file is the combination for the neuron m of all the samples that make up the batch.

To perform this operation the bias parameter must be an matrix of dimensions m x k.

Layer activation

The combination function will be exactly the same for the layer, in fact, all the neurons of a layer have the same activation function. Once the activation function is applied, the values will be the output of the layer.

Batch improvement

Calculate outputs is the main operation in neural network, therefore it is extremely important to optimize the entire process as much as possible, since this function will be called many times throughout the training process.

For this reason the matrix form has been introduced into OpenNN library to improve the performance, indeed, is the main method to carry out this operation.

Calculating the layer output using batches provides a higher speed in training. However, the general architecture is more complex to understand, so an example is included:

Example 2. *Consider the layer of neurons L. Let the input to that layer be the first batch of 3.1:*

x1	*x2*	*x3*	*x4*	*y*
1.2	11.2	0.12	3.2	1.2
0.2	12.2	0.12	3.2	2.4
2.3	13.2	0.13	3.2	2.1
7.6	14.2	0.15	3.2	4.3
23	15.2	0.16	3.2	3.1
14.2	16.2	0.17	3.2	4.2
12.3	17.2	0.18	3.2	5.6
18.2	17.2	0.19	3.2	3.9
9.5	17.2	0.20	3.2	7.6
7.5	16.2	0.21	3.2	6.4
9.2	15.2	0.20	3.2	7.8
10.6	14.2	0.19	3.2	3.9
5.2	13.2	0.18	3.2	3.4
17.2	12.2	0.17	3.2	3.2
20.9	10.2	0.16	3.2	2.1
14.2	9.2	0.15	3.2	2.5

Table 3.1: Split data set example

The combination of L will be:

$$C(X) = \underbrace{\begin{pmatrix} 0.41 & 0.41 & 0.41 & 0.41 \\ -0.70 & -0.70 & -0.70 & -0.70 \end{pmatrix}}_{\mathbf{B}} + \underbrace{\begin{pmatrix} -0.68 & 0.14 & -0.50 & 0.52 \\ 0.85 & -0.18 & -0.65 & 0.05 \end{pmatrix}}_{\mathbf{W}} \cdot$$

$$\underbrace{\begin{pmatrix} 1.2 & 11.2 & 0.12 & 3.2 \\ 0.2 & 12.2 & 0.12 & 3.2 \\ 2.3 & 13.2 & 0.13 & 3.2 \\ 7.6 & 14.2 & 0.15 & 3.2 \end{pmatrix}}_{\mathbf{X}} = \begin{pmatrix} 2.4 & -4.71 & -0.4 & 1.3 \\ 0.8 & -1.3 & 0.7 & -0.5 \end{pmatrix}$$

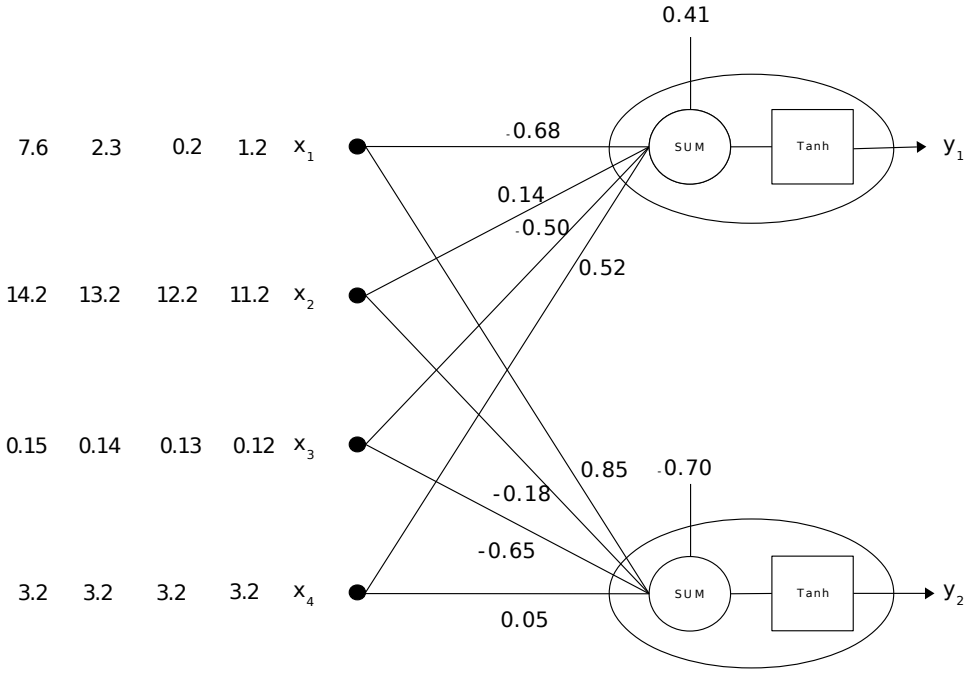


Figure 3.6: Layer example.

Each column of the matrix stores the information of the combination for a sample, therefore the number of columns will be equal to the size of the batch.

This matrix once applied the combination function would be the input matrix for the next layer.

Notice that if layer is composed only for one neuron the combination function would return a matrix of dimensions 2 x 1, which is the way how to calculate the outputs using a vector.

3.2.3 Feed-forward architecture

Feed forward architecture consists of multiple layers of basic units, usually interconnected in a feed-forward way. This means that the information only travels forward in the neural network, through the input nodes then through the hidden layers (single or many layers) and finally through the output nodes. Each neuron in one layer has directed connections to the neurons of the subsequent layer.

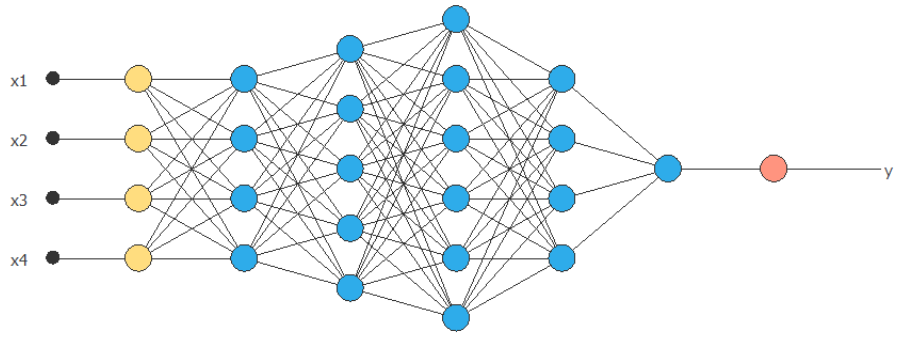


Figure 3.7: Feed forward architecture

These networks are mostly used for supervised machine learning, and the difference between a single layer or just a neuron, is the property of universal approximation, that claim that theoretically there will always be an ideal architecture able of generate a function that approximates any set of data with an error as small as desired [14].

Let φ be a non constant, bounded, and monotonically-increasing continuous function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$ there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

as an approximate realization of the function f, independent of φ; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

3.3 Loss index

The objective when a data set is analyzed is that the neural network response matches with the correct predictions. The loss Index defines the task to carry out by the neural network and provides a quantitative measure of the model accuracy[13].

In general, it is composed of two terms, error term and regularization. The difference between the outputs of the neural network and the correct predictions is measured by the error term. The regularization term is applied in order to obtain a good generalization,

The loss index can be defined as:

$$\mathbf{L} = \mathbf{E} + \mathbf{\Omega} \tag{3.9}$$

As we will see in the next section, training algorithms make use of loss gradient to find the set of optimal parameters $\boldsymbol{\theta}$ of the neural network, therefore we will also define the gradient of the loss as:

$$\mathbf{g} = \begin{pmatrix} \dfrac{\partial \mathbf{L}(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \dfrac{\partial \mathbf{L}(\boldsymbol{\theta})}{\partial \theta_d} \end{pmatrix}. \tag{3.10}$$

3.3.1 Error term

The error between the neural network outputs and the data set targets is the most important term in the loss index.

There is a great variety of error functions some of the most relevant are:

- Mean squared error (MSE): It calculates the average square error between the neural networks outputs and the target in the dataset.

$$\boxed{MSE = \frac{1}{q}\sum_{i=1}^{q}(y_i - t_i)^2} \tag{3.11}$$

- Normalized squared error (NSE): It calculates the square error between neural networks outputs and the target in the dataset, and then normalizes the result with respect to the mean.

$$\boxed{NSE = \frac{1}{N}\sum_{i=1}^{q}(y_i - t_i)^2} \tag{3.12}$$

We will define as q the number of instances, y_i the predictions of the neural network and t_i the actual value of the dependent variable.

3.3.2 Regularization term

Regularization is a value that deals with the excess of complexity in the model (over fitting), generally it is produced by an excessive number of neurons.

The regularization terms add additional information to the loos index in order to obtain smoother response.

There are two commons regularization terms:

- L1 regularization: sum of neural network's parameters

$$l1 = w\sum_{i=1}^{q}|\theta| \tag{3.13}$$

- L2 regularization: sum of square neural network's parameters

$$l2 = w\sum_{i=1}^{q}\left|\theta^2\right| \tag{3.14}$$

3.3.3 Batch approximation

We will denote as loss index approximation, the function resulting from using a set of samples (batch) to generate it.

$$\mathcal{L} = \mathcal{E} + \Omega \tag{3.15}$$

This approach is used in the batch algorithms, the error of this approach is slightly different from the classic loss index error. It is calculated with the average error of all batches that make up the data set.

Using the mean square error without regularization, calculate loss is defined as:

$$\boxed{MSE = \frac{1}{k}\sum_{j=1}^{k}\frac{1}{q}\sum_{i=1}^{q}(y_i^j - t_i^j)^2} \tag{3.16}$$

Here q represent the number of instances that compose a batch and k represent batches number, y_i^j the prediction i for the current batch j and t_i^j the actual value i of the dependent variable in batch j.

We will also define the gradient of the function approximation error as:

$$\mathfrak{g} = \begin{pmatrix} \dfrac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_1} \\ \vdots \\ \dfrac{\partial \mathcal{L}(\boldsymbol{\theta})}{\partial \theta_d} \end{pmatrix}. \tag{3.17}$$

3.4 Optimization algorithms

The learning problem for neural networks consist in searching for a set of parameters at which the loss index takes a minimum value, then the gradient over this point will be zero.

The loss index is, in general, a non-linear function. As a consequence, it is not possible to find closed optimization algorithms for the minima.

Instead, we consider a search through the parameter space consisting of a succession of steps.

Optimization algorithm carry out this task, and it stops when a specified condition is satisfied. Some stopping criteria commonly used are:

- The parameters increment norm is less than a minimum value.
- The loss improvement in one epoch is less than a set value.
- Loss has been minimized to a goal value.
- The norm of the loss index gradient falls below a goal.
- A maximum number of epochs is reached.
- A maximum amount of computing time has been exceeded.
- The error on the selection subset increases during a number of epochs.

3.4.1 Traditional algorithms

We call traditional algorithms those that use all instances to create and operate over loss index. All traditional algorithms follow the same flow:

Quasi-newton method

Quasi Newton method build up an approximation to the inverse Hessian at each iteration of the algorithm. This approximation is computed using only information on the first derivatives of the loss function.

The Hessian matrix is composed of the second partial derivatives of the loss function. The main idea behind the quasi-Newton method is to approximate the inverse Hessian by another matrix

The recurrence rule is as follows:

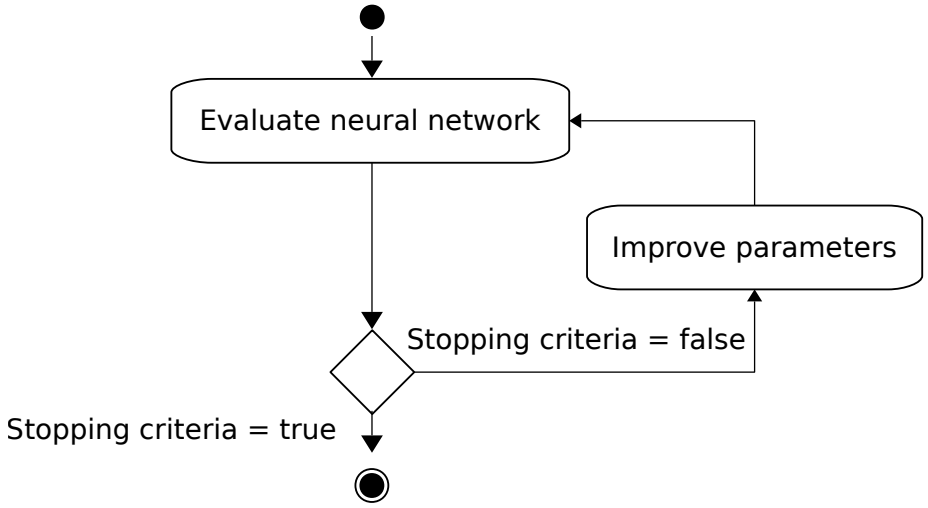


Figure 3.8: Training process.

$$\boxed{\theta_{i+1} = \theta_i - H_i^{-1} g_i.\eta} \tag{3.18}$$

Conjugate gradient

The conjugate gradient method can be regarded as something intermediate between gradient descent and Newton's method.

It is motivated by the desire to accelerate the typically slow convergence associated with gradient descent.

In the conjugate gradient training algorithm, the search is performed along conjugate directions which produces generally faster convergence than gradient descent directions.

These training directions are conjugated with respect to the Hessian matrix.

The recurrence rule is as follows:

$$\boxed{\theta_{i+1} = \theta_i - \gamma_i g_i.\eta} \tag{3.19}$$

Here γ is called the conjugate parameter, and there are different ways to calculate it.

For all conjugate gradient algorithms, the training direction is periodically reset to the negative of the gradient.

Chapter 4

Batch optimization algorithms

Batch optimization algorithms updates the neural network's parameters after every subset of data is analyzed, therefore, they use an approximation of the Loss Index.

To perform this task, these algorithms calculate the gradient over this approximation, instead of the true gradient computed from the entire data set.

Batches are used to approximate the error function, so it is important that the batch keep in enough information for a suitable approximation.

Figure 3.1 shows different batch sizes for a given data set, as data decrease (batch size become smaller) it is harder to find a function able to generalize correctly, in other words the problem is worse conditioned.

When an very small batch size is used to approximate the Loss Index the variance of the gradient is usually large, and these algorithms spend much time bouncing around, leading to slower convergence, indeed, the time to reach the same degree of convergence increases exponentially.

There are some experimental techniques to reduce the variance of the gradient [18], however they are out of the theoretical framework of this work.

All algorithms explained below follow the same rules for updating the parameters and we can summarize these rules as:

- Split data set into data subsets called Batches.

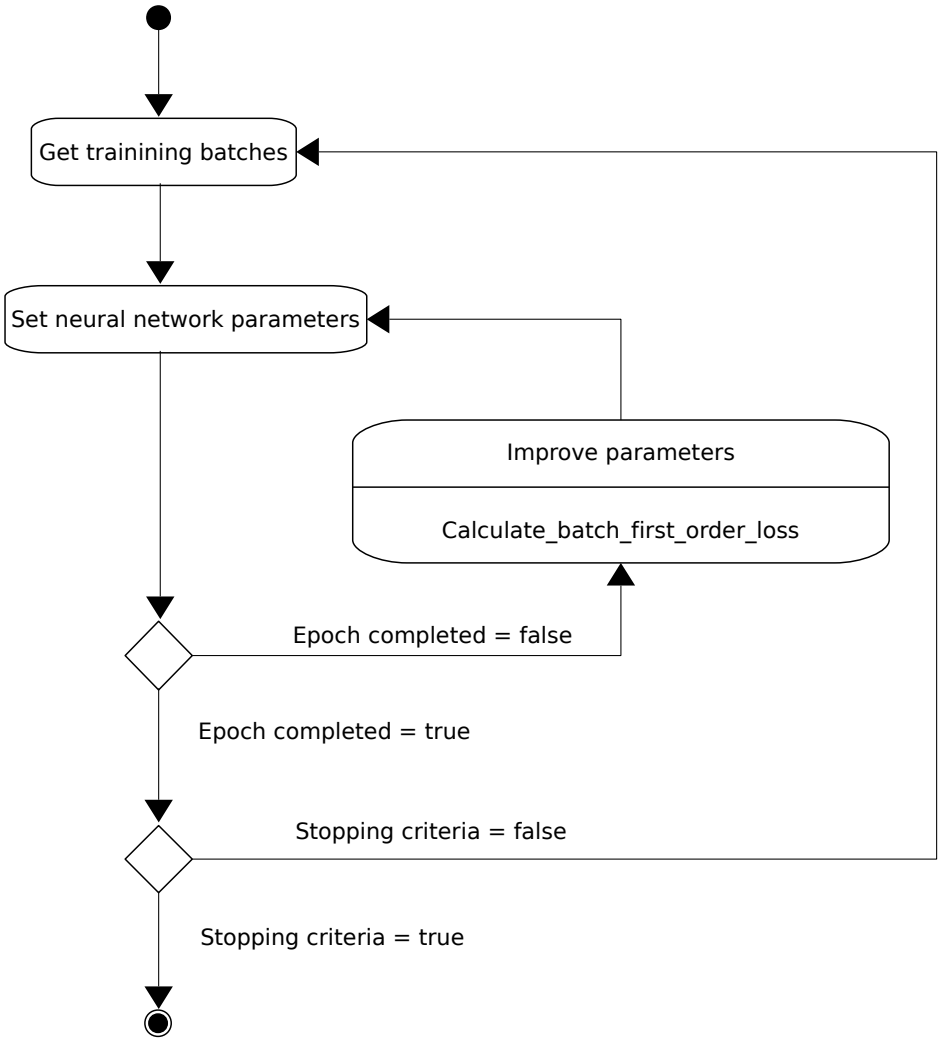


Figure 4.1: Batch optimization algorithms structure.

- Calculate the gradient for each batch.
- Calculate the loss for each batch.
- Once all bathes are analyzed check stopping criteria.
- If stopping criteria is false repeat process.

Maybe you have notice that two loops are involved. The behavior of these loops is as follows: once the general data set has been divided into batches,

the internal loop denoted by letter i (iteration number) begins. The performance of the loop is based on calculating the gradient over the error function approximation for each of the batches, when all of them have been computed, it comes out of the inner loop.

The external loop denoted by the letter e (epoch number) basically marks the amount of times that the process will be repeated. For a better visualization figure 4.1, shows the internal structure.

4.1 Stochastic gradient descent

Stochastic gradient descent, also known as SGD, was the first algorithm used together with the batch structure to deal with large amount of data.

This algorithm works equal to gradient descent. It uses the gradient to look for the direction of the local minimum and the learning rate to move thought $\mathbb{R}^d$ space to reach it, the only difference between both is how data is computed.

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_{i}^{(e)} - \mathfrak{g}_{i}^{(e)}.\eta} \tag{4.1}$$

Momentum

Momentum is a technique that deal with SGD oscillations accelerating in the relevant directions, adding a fraction μ of the update vector of the past step to the current update vector.

This momentum is the same as the momentum in classical physics, as we throw a ball down a hill it gathers momentum and its velocity keeps on increasing, this is the very simple idea behind Momentum optimization, proposed by Boris Polyak in 1964 [19].

We will denote by v the velocity vector that contains the information of the current gradient plus information of the past gradients with less relevance.

$$\boldsymbol{\theta}_{i+1}^{(e)} - \boldsymbol{\theta}_{i}^{(e)} = -\mathfrak{g}_{i}^{(e)}.\eta \tag{4.2}$$

$$-\mathfrak{g}_i^{(e)}.\eta = \Delta\boldsymbol{\theta}_i^{(e)} \tag{4.3}$$

$$\boxed{\boldsymbol{v}_i^{(e)} = \Delta\boldsymbol{\theta}_i^{(e)} + \mu.\boldsymbol{v}_{i-1}^{(e)}} \tag{4.4}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_i^{(e)} + \boldsymbol{v}_i^{(e)}} \tag{4.5}$$

The velocity term decreases for dimensions whose gradients change directions and increases for dimensions whose gradients point in the same directions.

In other words the momentum method relies on the exponentially weighted moving average (EWMA) to make the direction of the independent variable more consistent, thus reducing the possibility of divergence.

Nesterov

Nesterov is a variant of momentum proposed by Yurii Nesterov in 1983 [20], the idea behind this algorithm consist in evaluate the gradient slightly ahead in the direction of the momentum not at the local position.

Momentum generally provides more accuracy direction about where the minima is located, for this reason evaluate the gradient over this position will be more accurate, instead of calculate the gradient at the origin position.

To avoid performance problems, it will compute the "adjusted gradient" by assuming that the new gradient (evaluated at momentum direction) will be the current average gradient plus the product of momentum and the change in the average gradient.

$$\boldsymbol{\theta}_{i+1}^{(e)} - \boldsymbol{\theta}_i^{(e)} = -\mathfrak{g}_i^{(e)}.\eta \tag{4.6}$$

$$-\mathfrak{g}_i^{(e)}.\eta = \Delta\boldsymbol{\theta}_i^{(e)} \tag{4.7}$$

$$\boxed{\boldsymbol{v}_i^{(e)} = \Delta\boldsymbol{\theta}_i^{(e)} + \mu.\boldsymbol{v}_{i-1}^{(e)}} \tag{4.8}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_{i}^{(e)} + \boldsymbol{v}_{i}^{(e)}} \tag{4.9}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_{i}^{(e)} + \mu.\boldsymbol{v}_{i}^{(e)} + \Delta\boldsymbol{\theta}_{i}^{(e)}} \tag{4.10}$$

4.2 Adagrad

Adaptive gradient algorithm is an algorithm that, as the name suggests, adapts the learning rate for each parameter by using the gradient square [21].

We will denote by a (accumulator) the vector that keeps the sum of the square gradients, ϵ a smoothing term that avoids division by zero and η the initial learning rate.

$$\boxed{\boldsymbol{a}_{i}^{(e)} \;=\; \boldsymbol{a}_{i-1}^{(e)} + \mathfrak{g}_{i}^{(e)}.\mathfrak{g}_{i}^{(e)}} \tag{4.11}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_{i}^{(e)} - \frac{\eta\mathfrak{g}_{i}^{(e)}}{\sqrt{\boldsymbol{a}_{i}^{(e)}} + \epsilon}} \tag{4.12}$$

Initially Adagrad, perform large updates for parameters associated with larger derivatives, however as time goes by, the algorithm performs smaller updates, due to the increase in the denominator.

The philosophy of this algorithm is summarized in making large updates at first interactions, to reach the surroundings of the loss function minimum, and then make small updates that allow you to arrive more accurately towards the local minimum.

The less complexity the error function has, the better this algorithm will work.

4.3 RMSprop

Root mean square propagation algorithm is a gradient-based optimization algorithm proposed by Geoffrey Hinton at his Neural Networks course. [22].

This algorithm arises to provide a solution for the vanishing gradient problem that occurs in Adagrad. Instead of keeping the total sum of the square gradients it uses a exponential moving average of squared gradients to normalize the gradient itself.

We will denote by m (exponential square moving average) the vector that keeps information of exponential average of square of gradients, ϵ a smoothing term that avoids division by zero, η the initial learning rate and ρ a parameter that indicates the length of the exponential moving average.

$$\boxed{\boldsymbol{m}_i^{(e)} = \boldsymbol{m}_{i-1}^{(e)}\rho + \mathfrak{g}_i^{(e)}.\mathfrak{g}_i^{(e)}(1-\rho).} \tag{4.13}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_i^{(e)} - \frac{\eta.\mathfrak{g}_i^{(e)}}{\sqrt{\boldsymbol{m}_i^{(e)}} + \epsilon}} \tag{4.14}$$

$\boldsymbol{m}$ vector help to balance the updates, allowing decrease the step for gradients too large and increasing it for very small gradients avoiding fading.

4.4 Adadelta

Adaptive learning rate method was developed at the same time as RMSprop, to solve the continual decay of learning rates throughout training process, and for automatically select the global learning rate [23].

As well as RMSprop, Adadelta calculates the exponential moving average of squared gradients, to prevent fading and reduce the aggressiveness caused by Adagrad.

On the other hand AdaDelta also calculate an exponential moving average of parameters.

Both combined strategies result in a well balanced algorithm.

$$\boxed{\boldsymbol{m}_i^{(e)} = \boldsymbol{m}_{i-1}^{(e)}\rho + \mathfrak{g}_i^{(e)}.\mathfrak{g}_i^{(e)}(1-\rho).} \tag{4.15}$$

$$\boxed{\boldsymbol{s}_i^{(e)} = \boldsymbol{s}_{i-1}^{(e)}\rho + \boldsymbol{\theta}_i^{(e)}.\boldsymbol{\theta}_i^{(e)}(1-\rho).} \tag{4.16}$$

$$\boxed{\boldsymbol{\theta}_{i+1}^{(e)} = \boldsymbol{\theta}_i^{(e)} - \eta.\frac{\sqrt{\boldsymbol{s}_{i-1}^{(e)}} + \boldsymbol{\epsilon}}{\sqrt{\boldsymbol{m}_i^{(e)}} + \boldsymbol{\epsilon}}} \tag{4.17}$$

We will denote by m the vector that keeps information of exponential average of square of gradients, s the vector keeps information of exponential average of parameters history (as momentum), ϵ a smoothing term that avoids division by zero, η the initial learning rate and ρ a parameter that indicates the length of the exponential moving average.

4.5 Adam

Adaptive moment estimator was presented by Diederik Kingma and Jimmy Ba from the University of Toronto in 2015 [24].

This algorithm combine the advantages of AdaGrad and RMSProp the two main extensions of stochastic gradient descent.

Instead of adapting the parameter learning rates based on exponential square moving average of the gradients as in RMSProp, this algorithm also makes use of exponential moving average of gradients.

Adam's papers provide an intuitive idea of what this concepts represent. On one hand exponential moving average of gradients are the first estimator of the gradient also know as the mean m, on the other hand exponential moving average of the square gradients are the second moment of the gradients also know as the variance v.

Both estimators combined together provide useful information to reach fast and accurate the local minimum of the loss function.

$$\boxed{\boldsymbol{m}_i^{(e)} = \beta_1 \boldsymbol{m}_{i-1}^{(e)} + (1-\beta_1)\mathfrak{g}_i^{(e)}} \tag{4.18}$$

$$\boxed{\boldsymbol{v}_i^{(e)} = \beta_2 \boldsymbol{v}_{i-1}^{(e)} + (1-\beta_2)\mathfrak{g}_i^{(e)}.\mathfrak{g}_i^{(e)}} \tag{4.19}$$

In order to avoid vanish gradient, especially during the initial time steps, and especially when the decay rates are small, Adam's author made a slight adjustment of the estimators

$$\boxed{\widetilde{\boldsymbol{m}}_i^{(e)} = \frac{\boldsymbol{m}_i^{(e)}}{1-\beta_1^{l+1}}} \tag{4.20}$$

$$\boxed{\widetilde{\boldsymbol{v}}_i^{(e)} = \frac{\boldsymbol{v}_i^{(e)}}{1-\beta_2^{l+1}}} \tag{4.21}$$

$$\boxed{\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \widetilde{\boldsymbol{m}}_i^{(e)}.\frac{\eta}{\sqrt{\widetilde{\boldsymbol{v}}_i^{(e)}} + \epsilon}} \tag{4.22}$$

We will denote by m the vector that keeps information of exponential average of gradients, (the mean), v the vector that keeps information of exponential average of the square gradients, (the variance), ϵ a smoothing term that avoids division by zero, β_1 and β_2 parameters that indicates the length of the exponential moving average and η the initial learning rate.

Chapter 5

Performance comparison

5.1 Benchmark description

In this chapter we will compare the OpenNN [9] and Tensorflow [10] libraries for 3 different problems using batch optimization algorithms. Also, traditional algorithms with batch optimization algorithms will be compared, in order to know the advantages and disadvantages of them.

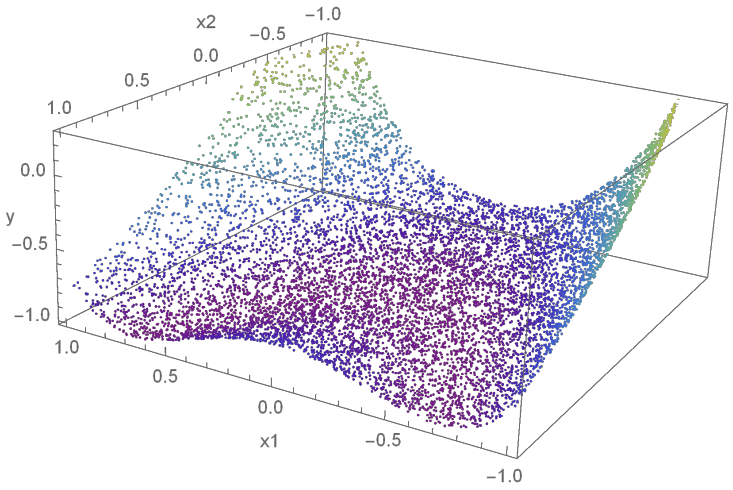


Figure 5.1: Rosenbrock data

The chosen problems have been Rosenbrock's data set of:

- 11 variables and 10.000 samples, total 110.000 data.
- 101 variables and 100.000 samples, total 10.100.000 data.
- 1001 variables and 1.000.000 samples, total 1.001.000.000 data.

The Rosenbrock function, is a non-convex function, introduced by Howard H. Rosenbrock in 1960, it is widely used as a performance test problem [25]. However for this particular case, it will be the error function associated with the Rosenbrecok data that the algorithm will have to minimize.

In other words, the goal is that batch optimization algorithms be able to find a neural network capable of reproducing the Rosenbrock function in 10, 100, and 1000 variables.

To generate the datasets we will use the Rosenbrock formula:

$$f(\boldsymbol{x}) = \sum_{i=1}^{N-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2 \; where \; \boldsymbol{x} = [x_1, ..., x_n] \in \mathbb{R}^N \quad (5.1)$$

The neural networks architecture will be the same for all problems, a single hidden layer with 1000 neurons, and an output layer with 1 neuron, however, the neural network parameters will increase as inputs number increase:

- 10 inputs: 10.010 parameters have to be modeled.
- 100 inputs: 100.100 parameters have to be modeled .
- 1000 inputs: 1.001.000 parameters have to be modeled.

The activation functions are the hyperbolic tangent for the first layer and the linear function for the second.

Mean square error has been used as loss index and Adaptive moment estimator as the training algorithm.

In order to provide an intuitive idea about Rosenbrock function, the data obtained from the 3-dimensional Rosenbrock function is attached figure 5.1.

The computer features to perform the comparisons are attached as well table 5.1.

System	Windows
Release	10
Version	10.0.17134
Machine	AMD64
Processor	Intel 64 Family 6 model 158
Version	3.6.8
Compiler	MSV v.1916 64 bit (AMD64)
Device	GeForce GTX 1950
Physical RAM	32 GB

Table 5.1: Computer features

5.2 OpenNN vs TensorFlow

OpenNN (Open Neural Networks Library) [9] is a software library written in the C++ programming language which implements neural networks, a main area of deep learning research.

The library is open-source, licensed under the GNU Lesser General Public License. OpenNN was released in 2005 as a comprehensive implementation of the multilayer perceptron neural network in the C++ programming language. This library implements a wide variety of mathematical tools such as statistical and optimization methods, which allow solve regression, classification, forecasting, and association problems.

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. It is a symbolic math library, and is also used for machine learning applications such as neural networks.[5] It is used for both research and production at Google [10].

Tensorflow has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications. This library has a multitude of functionalities implemented such as convolutional networks or LTSM networks ...

The objective in this section consists in comparing the performance of the main algorithm for massive data processing (Adam) in both libraries.

5.3 10 variables 10000 samples

In order to establish a criterion about which library performs better in minimizing the error function associated with the current problem, the time to perform 10000 epoch will be measured and the precision reached by both algorithms will be recorded.

5.3.1 Tensorflow performance

After 10000 epoch ADAM algorithm from TensorFlow library is able to converge to a value of: 0.0033.

Notice that in figure 5.6 the error remains constant over a period of time, we can consider this value as a potential well where the neural network is able to predict the same as the average.

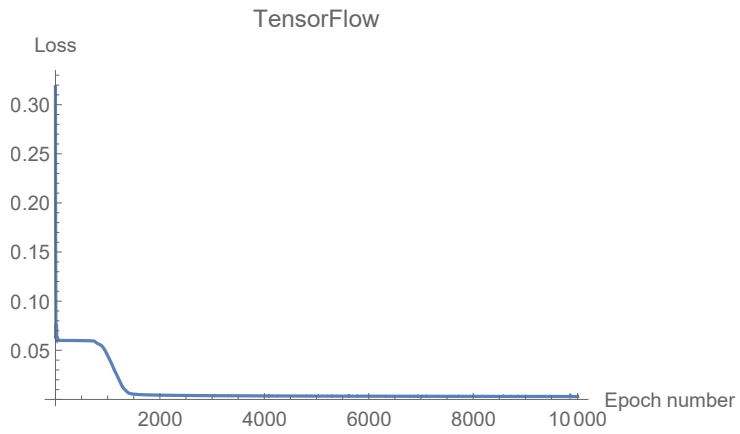


Figure 5.2: Accuracy vs Epoch

5.3.2 OpenNN performance

Adam implementation in OpenNN has a similar behavior, after 10000 epoch Adaptive moment estimator is able to converge to a value of: 0.0034

Notice that the degree of convergence is not exactly the same because different distributions have been used to initialize neural network parameters, however, the error in both cases is not far away.

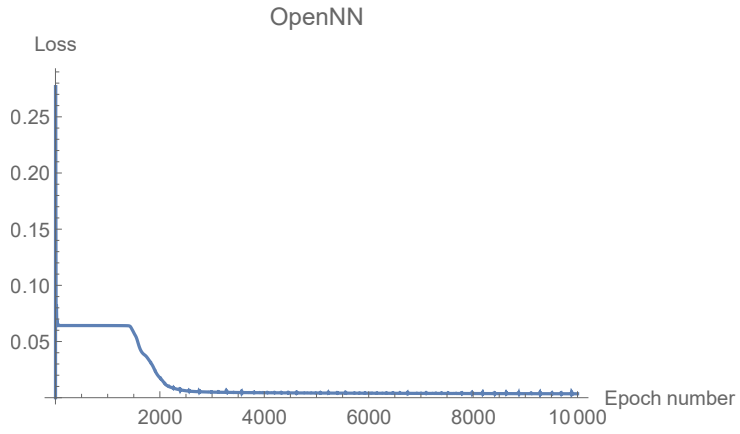


Figure 5.3: Accuracy vs Epoch

If we observe the convergence time against batch size we see an identical behavior to the case of Tensorflow 5.4:

5.3.3 Results

The following table 5.2 shows the results for this problem:

.	Epochs	Batch size	Final loss	Time (hh:mm:ss)
Tensorflow	10000	10000	0.0033	00:02:49
OpenNN	10000	10000	0.0034	00:01:24

Table 5.2: Results 10 variables 10000 samples

To finish this problem we will analyze how batch size affects to the convergence of the algorithm.

To do that we set the stopping criteria for a training error of 0.0033 in both implementations and then we vary the size of the batch to see how it affects the convergence time.

Figure 5.4 shows that as the size of the batch decreases, the time to reach the same degree of convergence increases exponentially.

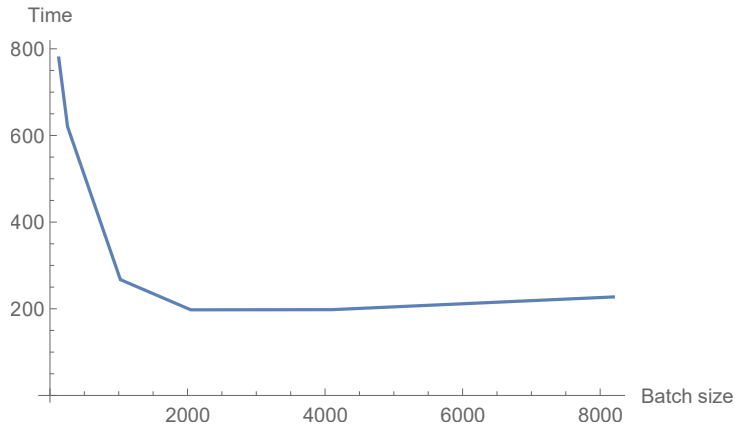


Figure 5.4: Time vs Batch

This is due to small batches do not have enough information, so they can not make a good approximation of the loss index and the problem becomes worse conditioned.

5.4 100 variables 100000 samples

The same test will be carried out as in the previous example, both algorithms will perform 10000 epoch and we will see the loss and the time it takes to reach it.

For this problem we will use a batch size of 10000, so, at each epoch 10 iterations will be done, the main reason to use this strategy is because Nvidia GTX 1050 graphic card does not have enough memory to perform operation with batches equal to the size of the data set.

5.4.1 Tensorflow performance

After 10000 epoch ADAM algorithm from TensorFlow library is able to converge to a value of: 0.00019

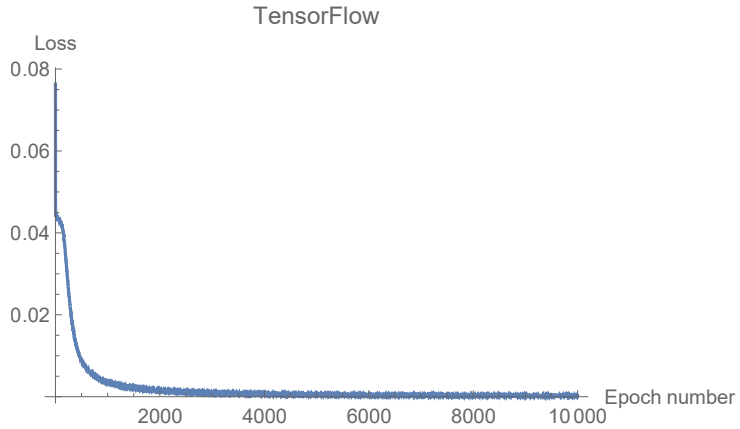


Figure 5.5: Accuracy vs Epoch

5.4.2 OpenNN performance

After 10000 epoch ADAM algorithm from OpenNN library is able to converge to a value of: 0.00019

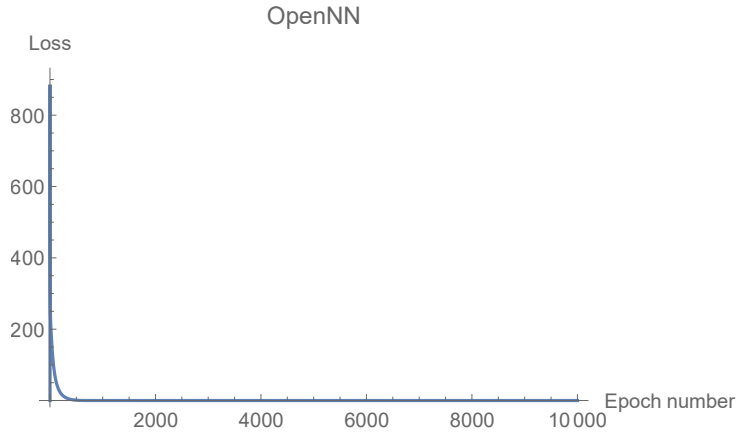


Figure 5.6: Accuracy vs Epoch

5.4.3 Results

Table shows the results for the current problem:

.	Epochs	Batch size	Final loss	Time (hh:mm:ss)
Tensorflow	10000	10000	0.00017	00:42:42
OpenNN	10000	10000	0.000097	00:19:08

Table 5.3: Results 100 variables 100000 samples

The error calculated for the previous problem is equivalent to the traditional algorithms error explained in section 4.1, however now we have used a batch size that corresponds to one tenth of the total data. We have made an approximation of the loss index, and this means that the error obtained is an average of the errors of all batches.

5.5 1000 variables 1000000 samples

This problem is the most complex of all, since 1.001.000 parameters should be modeled.

Due to the size of the problem we will leave the algorithm running 1000 epoch and the batch size will be 1000 as well.

5.5.1 Tensorflow performance

Figure 5.7 shows the initial configuration of the problem, after 1000 epoch Tensorflow implementation reach a loss of 0.00062.

5.5.2 OpenNN performance

Figure 5.8 shows the initial configuration of the problem, after 1000 epoch Tensorflow implementation reach a loss of 0.00062.

5.5.3 Results

The degree of convergence is not exactly the same because different distributions have been used to initialize neural network parameters, however, the

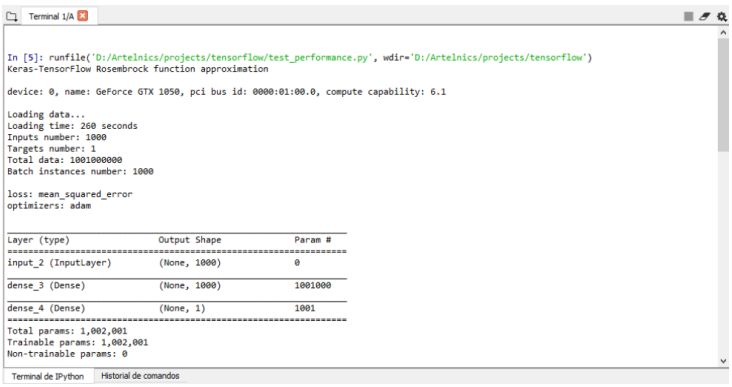


Figure 5.7: Initial configuration TensorFlow

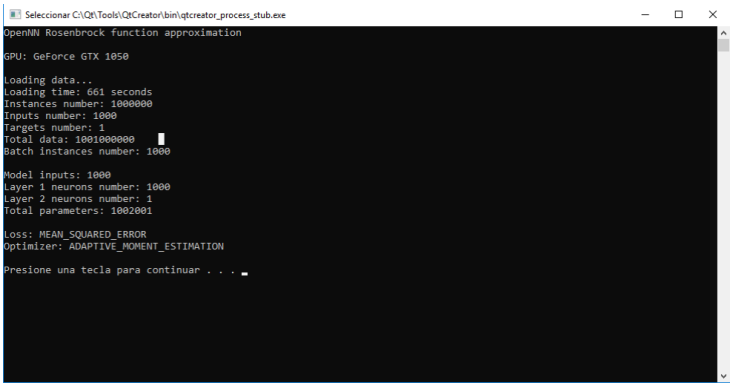


Figure 5.8: Initial configuration OpenNN

error in both cases is not far away.

Table 5.4 show the results for the given problem.

.	Epochs	Batch size	Final loss	Time (hh:mm:ss)
Tensorflow	1000	1000	0.00062	12:15:02
OpenNN	1000	1000	0.00062	2:07:21

Table 5.4: Results 100 variables 100000 samples

The reasons why OpenNN performance is higher than TensorFlow is mainly because OpennNN is a software library written in the C++ so it uses a compiled language, conversely, the most extensive use of TensorFlow is with Python that is an interpreter language.

5.6 Bath vs traditional algorithms

In this section we will compare the QuasiNewton algorithm with adaptive moment estimator(Adam), for the 3 previous problems:

- 11 variables and 10.000 samples, total 110.000 data.
- 101 variables and 100.000 samples, total 10.100.000 data.
- 1001 variables and 1.000.000 samples, total 1.001.000.000 data.

The neural networks architecture will be the same for both problems, a single hidden layer with 1000 neurons, and an output layer with 1 neuron. The activation functions are the hyperbolic tangent for the first layer and the linear function for the second.

The computer features to perform the comparisons are the same as in the previous chapter 5.1.

The results for the first problem 10 variables and 10000 instances are shown in table 5.5

.	Epochs	Final loss	Time (hh:mm:ss)
QuasiNewton	1000	0.00019	00:59:29
Adam	10000	0.0034	00:01:39

Table 5.5: Results 10 variables 10000 QuasiNewton vs Adam

For the first problem 10 variables and 10000 samples, we can appreciate that, QuasiNewton is much more precise than Adam, however the time to reach that degree of convergence is quite high.

The mathematics of both algorithms are very different, while Adam, calculates the derivatives of the function, QuasiNewton calculates an approximation of the hessian matrix. This implies a better search address of the minimum of the function, but a greater number of calculations are needed.

The results for the 100 variables and 1000000 samples problem are shown in table 5.6

Here again the same thing happens QuasiNewton provides us with a more precise solution, while Adam gives us a faster solution but less accurate.

.	Epochs	Final loss	Time (hh:mm:ss)
QuasiNewton	1000	2.10^{-8}	13:20:10
Adam	10000	0.000097	00:19:08

Table 5.6: Results 100 variables 10000 QuasiNewton vs Adam

Finally for the biggest problem, the results are show in table 5.7

.	Epochs	Final loss	Time (hh:mm:ss)
QuasiNewton	X	X	X
Adam	1000	0.00062	02:07:20

Table 5.7: Results 1000 variables 100000 QuasiNewton vs Adam

The QuasiNewton algorithm is not able to solve the current problem with this computer 5.1, since it needs a lot of memory to save the particular Hessian matrix (1.001.000 x 1.001.000).

Chapter 6

Conclusions

According to the results obtained, the main advantage of batch optimization algorithms is the capability for finding solutions to large problems, in a relative short time. On the other hand, the main disadvantage is that universal approximation property [14] is not preserved. Since they must meet certain conditions, such as the batch size should be big enough, in order to obtain convergence.

Usually, traditional algorithms offer greater degree of convergence, however the time that they spend to reach it, in some cases, is simply not worth it. Specially when a very high dimensional function should be modeled. For instance, in image processing and voice recognition, these algorithms are not profitable. Nonetheless in regression and classification problems these algorithms usually give pretty good results.

Non-frealance theorem [26] claims that there is no better algorithm than another, having said that, we can conclude that batch optimization algorithms allow us to cover more problems, than before were not possible with traditional.

Bibliography

[1] C Schaefer, M Geiger, T Kuntzer, and J-P Kneib. Deep convolutional neural networks as strong gravitational lens detectors. *Astronomy & Astrophysics*, 611:A2, 2018.

[2] John Peurifoy, Yichen Shen, Li Jing, Yi Yang, Fidel Cano-Renteria, Brendan G DeLacy, John D Joannopoulos, Max Tegmark, and Marin Soljačić. Nanophotonic particle simulation and inverse design using artificial neural networks. *Science advances*, 4(6):eaar4206, 2018.

[3] T Maggipinto, G Nardulli, S Dusini, F Ferrari, I Lazzizzera, A Sidoti, A Sartori, and Gian Pietro Tecchiolli. Role of neural networks in the search of the higgs boson at lhc. *Physics Letters B*, 409(1-4):517–522, 1997.

[4] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[5] Gianina Alina Negoita, Glenn R Luecke, James P Vary, Pieter Maris, Andrey M Shirokov, Ik Jae Shin, Youngman Kim, Esmond G Ng, and Chao Yang. Deep learning: a tool for computational nuclear physics. *arXiv preprint arXiv:1803.03215*, 2018.

[6] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In *New advances in machine learning*. IntechOpen, 2010.

[7] Imad A Basheer and Maha Hajmeer. Artificial neural networks: fundamentals, computing, design, and application. *Journal of microbiological methods*, 43(1):3–31, 2000.

[8] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Fre-

itas. Learning to learn by gradient descent. In *Advances in neural information processing systems*, pages 3981–3989, 2016.

[9] Artelnics. *OpenNN library*, 2019. `http://www.opennn.net`.

[10] Google. *TensorFlow library*, 2019. `https://www.tensorflow.org`.

[11] Scikit-learn. *Scikit-learn library*, 2019. `https://scikit-learn.org/`.

[12] David Rolnick, Priya L. Donti, Lynn H. Kaack, Kelly Kochanski, Alexandre Lacoste, Kris Sankaran, Andrew Slavin Ross, Nikola Milojevic-Dupont, Natasha Jaques, Anna Waldman-Brown, Alexandra Luccioni, Tegan Maharaj, Evan D. Sherwin, S. Karthik Mukkavilli, Konrad P. Körding, Carla Gomes, Andrew Y. Ng, Demis Hassabis, John C. Platt, Felix Creutzig, Jennifer Chayes, and Yoshua Bengio. Tackling climate change with machine learning. *CoRR*, abs/1906.05433, 2019.

[13] Roberto Lopez and Eugenio Oñate. A variational formulation for the multilayer perceptron. In *International Conference on Artificial Neural Networks*, pages 159–168. Springer, 2006.

[14] Jooyoung Park and Irwin W Sand berg. Universal approximation using radial-basis-function networks. *Neural computation*, 3(2):246–257, 1991.

[15] Alice J O'Toole, Fang Jiang, Hervé Abdi, Nils Pénard, Joseph P Dunlop, and Marc A Parent. Theoretical, statistical, and practical perspectives on pattern-based classification approaches to the analysis of functional neuroimaging data. *Journal of cognitive neuroscience*, 19(11):1735–1752, 2007.

[16] J. Šíma and P. Orponen. General-purpose computation with neural networks: A survey of complexity theoretic results. *Neural Computation*, 15:2727–2778, 2003.

[17] H. Demuth, M. Beale, and H. Martin. *Neural network design*. Singapore : Thomson Learning, 2009.

[18] Sashank J Reddi, Ahmed Hefny, Suvrit Sra, Barnabas Poczos, and Alex Smola. Stochastic variance reduction for nonconvex optimization. In *International conference on machine learning*, pages 314–323, 2016.

[19] Boris Polyak. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17, 12 1964.

[20] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[21] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[22] Geoffrey Hinton. Neural networks for machine learning coursera video lectures - geoffrey hinton. 2012.

[23] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[24] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[25] H. H. Rosenbrock. An Automatic Method for Finding the Greatest or Least Value of a Function. *The Computer Journal*, 3(3):175–184, 01 1960.

[26] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.