UNIVERSIDAD DEL ROSARIO

School of Engineering, Science and Technology
Applied Mathematics and Computer Science

UNDERGRADUATE THESIS

Automatic Determination of the Learning Rate for Multivariate and Multinomial Regression Models

---

Author: Manuela Acosta Fajardo

Supervisor: Alexander Caicedo Dorado

A thesis submitted in fulfillment of the requirements for the degree of Professional in Applied Mathematics and Computer Science

November 2022

Abstract

Throughout the years, artificial intelligence has developed into a widely researched and applied field, as a result of the significant advancements in technology and the expansion in computer resources. Artificial intelligence attempts not only to understand how the human mind works, but also to develop systems that can mimic human behaviour.

Machine learning is one of the main branches of artificial intelligence, and it aims to build and improve models that can learn from a set of data, and from experience, via computational methods, with no need to be explicitly programmed. Machine learning algorithms build models based on sample data, in order to make predictions or decisions, and are used in different applications, such as medicine, computer vision, image classification, among others.

A machine learning algorithm is a program that finds patterns or makes predictions from previously unseen data. Depending on the goals of the algorithm, as well as on the data used, there are different types of learning models: supervised learning, unsupervised learning and reinforcement learning. One of the most common learning algorithms is Gradient Descent, which is used to find a local minimum of a differentiable function. It works by taking repeated steps in the opposite direction of the gradient of the function.

The size of the steps taken by the gradient descent algorithm is determined by an hyper-parameter known as the Learning Rate. This parameter indicates how fast or slow is the movement towards the optimal parameters of the algorithm. Usually, it is set manually. However, in order to reach the function minima it is necessary to set an appropriate learning rate, i.e. neither too big, nor too small. In the first case, the steps taken are too big, and the algorithm can diverge as a consequence. On the contrary, if the learning rate is too small, it results in slow learning, and the algorithm could also never converge.

Most of the times a fast learning is desired, so high learning rates might be selected. Nevertheless, it is important to select the proper value for this parameter, so one can guarantee the convergence of the algorithm. A method to determine an upper-bound for the learning rate of models based on linear regression models was presented in [1], doing an analysis of the gradient descent algorithm as a discrete dynamical system.

This thesis work aims to extend these results to models based in classification and multinomial regression. We also seek to find an optimal value for the learning rate for these methods. Throughout this thesis an algorithm that automatically determines an optimal value for the learning rate of classification and regression models is developed. In the first place, the results obtained for the linear regression models are generalized to other activation functions. As a result, an upper-bound and an optimal value for the learning rate are found for models using regression and classification. Then, the results obtained are extended to a multinomial regression model.

We propose an analysis of the gradient descent as a discrete dynamical system, where the upper-bound arises as a criteria to determine the stability of this system. Besides, we present an optimal value for the learning rate, which minimizes the sum of the distance of the extreme poles of the dynamical system studied. This analysis is done by linearizing the gradient descent algorithm, and applying it to linear, logistic and multinomial regression. The upper-bound and the optimal value of the learning rate are approximations to the optimal value that guarantee the fastest convergence of the algorithm.

We present simulations and experiments to test the results obtained. We first test them with toy examples, by manually creating the data to study the behaviour of the algorithm for the linear and the logistic regression model. Then, we validate our approach in real datasets.

The results show that, although the maximum learning rate, which is given by the upper-bound, seems to make the algorithm converge faster than the optimal learning rate for the logistic and multinomial case, it is better to use this last value, as it guarantees a smooth and relatively fast convergence to the minimum in all cases.

Acknowledgements

I want to thank my tutor, Professor Alexander Caicedo Dorado for his unconditional support throughout the development of this thesis, for his good disposition, and for his will to help and to make this process enjoyable. I also want to thank Professor Edgar Andrade, for his readiness to help in anything I needed along this process. I extend my thanks to my closest friends for being an emotional support during this whole time. Finally, thanks to my family for always supporting me, and for believing in my capability to achieve my goals.

Contents

A mi familia, a los profesores que me apoyaron en este proceso,
y a mis amigos más cercanos.

Chapter 1

Introduction

1.1 Machine Learning

For many years, people have tried to understand how the human mind works, i.e. how we think. **Artificial Intelligence**, also known as AI, attempts not only to understand this, but also to build intelligent entities, and to develop systems that can mimic human behaviour [2]. AI is one of the newest fields in science and engineering, and it has different approaches, depending on the goals in mind. These approaches depend on whether one is concerned with thinking or behaviour, and if one wants to model either humans or work from an ideal standard [2]. The foundations of AI come from different disciplines that contributed with ideas, techniques and viewpoints, such as: philosophy, mathematics, economics, neuroscience, psychology, computer engineering, control theory and cybernetics and linguistics.

Machine Learning is one of the main branches of Artificial Intelligence. Its purpose is to develop techniques that allow computers to learn, as well as to create computer systems that automatically improve their performance through experience [3]. That is, methods that leverage data to improve performance on some set of tasks [4]. A different definition of machine learning is that it is the technique that improves system performance by learning from experience via computational methods [5]. In computer systems, this experience exists in the form of data. Machine learning algorithms build a model based on sample data, known as training data, in order to make predictions or decisions [6]. These algorithms are used in different applications, including medicine, computer vision, speech recognition, image classification and e-mail filtering. These fields have in common that it is difficult to develop a conventional algorithm to achieve the goals and perform the required tasks.

When we talk about machine learning, and machine learning algorithms, it is important to understand what *learning* means. Learning can be understood as the ability to change according to external stimuli and remembering most of all previous experiences. So machine learning gives maximum importance to the techniques that improves or increases the propensity for changing adaptively [7]. Therefore, the main goal of this field is to study and improve mathematical models, also known as algorithms, that can be trained with context-related data to make decisions without knowing all influencing elements, i.e. external factors.

1.2 Learning Algorithms

A machine learning model, or algorithm, is a program that can find patterns or make decisions from previously unseen dataset [8]. It is also defined as a mathematical representation of the output of the training process [9]. These models are represented as a mathematical function that makes predictions on the input data, and provides an output. The models are first trained over a set of data and then they are provided an algorithm to reason over data. Afterwards, they extract the pattern from fed data, and learn from it. Once the models are trained, they are used to predict unseen data [9].

There are different types of machine learning models, depending on the goals and datasets. The three main categories are: supervised learning, unsupervised learning and reinforcement learning. *Supervised learning* is the simplest model. Here, the algorithm is provided input data, with a known label, and is optimized to meet a set of specific outputs [8]. Supervised learning is divided into two categories: classification and regression. Classification is used in situations when there is only a discrete number of possible outcomes, which are called categories. On the other hand, regression is used if the problem is based on continuous output [7]. Some of the most common applications of this type of models are pattern detection, spam detection, automatic image classification, Natural Lenguage Processing, among others.

In contrast, when we talk about *unsupervised learning*, the algorithm is also provided an input dataset, but is not optimized to specific outputs. Instead, it is trained to group objects by common characteristics [8]. This approach is based on the absence of any supervisor, and its useful when it is necessary to learn how a set of elements can be grouped, or clustered, according to their similarity, or distance measure [7]. This category is also subdivided into: clustering, association rules, and dimensionality reduction; and some of its applications are object segmentation, automatic labeling and similarity detection.

Finally, in *reinforcement learning* the algorithm is made to train itself using many trial and error experiments. In this case, the algorithm interacts continually with the environment [8], thus, this approach is also based on feedback provided by the environment. This feedback is known as reward, and it is used to understand if a performed action is positive or not.

In this section we discuss different characteristics of one of the most common learning algorithms: Gradient Descent. We explain the algorithm itself, its different variations, as well as the influence of an specific hyper-parameter (the learning rate), and its initialization.

1.2.1 Gradient Descent

Gradient Descent is one of the most common algorithms used for training models in Machine Learning. It is and optimization algorithm used to find a local minimum of a differentiable function [10]. It works by taking repeated steps in the opposite direction of the gradient of the function, at the current point. In most of the cases, the function used is the cost function, also known as the prediction error function. It is called $J(\boldsymbol{\omega})$, and depends on the model parameters (or weights) ω. Gradient descent finds the values of ω that correspond to the minimum of the cost function. Choosing the cost function depends on the type of models that are being studied.

Once $J(\boldsymbol{\omega})$ is defined, the update rule on the coefficients $\boldsymbol{\omega}$ is applied according to the following equations:

$$\begin{aligned} \omega_i &:= \omega_i + \Delta\omega_i, \\ \Delta\omega_i &= -\eta \frac{\partial J(\boldsymbol{\omega})}{\partial \omega_i}, \end{aligned} \tag{1.1}$$

where η is a constant hyper-parameter, known as the *learning rate*, and determines the size of the steps taken. Then, we can summarize the way the algorithm works with the following steps:

1. Initialize the vector $\boldsymbol{\omega}$ of parameters randomly.
2. Calculate the gradient of the cost function $J(\boldsymbol{\omega})$ by calculating its partial derivatives $\partial J(\boldsymbol{\omega})/\partial \omega_i$.
3. Update the parameters following the rule defined in equation (1.1).
4. Repeat until the cost function $J(\boldsymbol{\omega})$ stops reducing, or until a pre-defined termination criteria is met.

There are different types and variations of gradient descent, which depend on the amount of data used. The most known are: Batch Gradient Descent, Stochastic Gradient Descent and Mini-Batch Gradient Descent.

In the first place, *Batch Gradient Descent*, also known as vanilla gradient descent, uses all the training data to take a single step. It takes the average of all the training examples, and then uses the mean gradient resulting to update the parameters. Therefore, this method uses just one step of gradient descent in one epoch [11]. It is computationally efficient, as it produces a stable error gradient and a stable convergence [10]. The batch gradient descent is defined as follows [12]:

Algorithm 1 Batch Gradient Descent

```
Require: Training set: $T$; Learning Rate: $\eta$.
Ensure: Model Parameters $\boldsymbol{\omega}$.
  Randomly initialize the parameters $\boldsymbol{\omega}$
  Initialize convergence $tag = False$
  while $tag == False$ do
    Compute gradient $\nabla_\omega(\boldsymbol{\omega}; T)$ on the training set $T$
    Update variable $\boldsymbol{\omega} = \boldsymbol{\omega} - \eta \cdot \nabla_\omega(\boldsymbol{\omega}; T)$
    if convergence condition holds then
      $tag = True$
    end if
  end while
  return Model variables $\boldsymbol{\omega}$
```

On the other hand, *Stochastic Gradient Descent* updates the parameters for each training example, one by one [10]. It is a good option for huge datasets, as it is faster than batch gradient descent, and it does not need the whole dataset to do the update. Stochastic gradient descent is defined as follows [12]:

Algorithm 2 Stochastic Gradient Descent

```
Require: Training set: $T$; Learning Rate: $\eta$.
Ensure: Model Parameters $\omega$.
  Randomly initialize the parameters $\omega$
  Initialize convergence $tag = False$
  while $tag == False$ do
    Shuffle the training set $T$
    for each data instance $(x_i, y_i) \in T$ do
      Compute gradient $\nabla_\omega(\omega; (x_i, y_i))$ on the training instance $(x_i, y_i)$
      Update variable $\omega = \omega - \eta \cdot \nabla_\omega(\omega; (x_i, y_i))$
    end for
    if convergence condition holds then
      $tag = True$
    end if
  end while
  return Model variables $\omega$
```

Finally, *Mini-Batch Gradient Descent* is a combination of the concepts used in both methods explained above. Basically, mini-batch gradient descent splits the training data into small batches, and updates the parameters for each of those batches. Usually is the chosen method, as it creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent [10]. Mini-batch gradient descent is defined as follows [12]:

Algorithm 3 Mini-batch Gradient Descent

```
Require: Training set: $T$; Learning Rate: $\eta$; Mini-batch size: $b$.
Ensure: Model Parameters $\omega$.
  Randomly initialize the parameters $\omega$
  Initialize convergence $tag = False$
  while $tag == False$ do
    Shuffle the training set $T$
    for each mini-batch $\beta \in T$ do
      Compute gradient $\nabla_\omega(\omega; \beta)$ on the mini-batch $\beta$
      Update variable $\omega = \omega - \eta \cdot \nabla_\omega(\omega; \beta)$
    end for
    if convergence condition holds then
      $tag = True$
    end if
  end while
  return Model variables $\omega$
```

1.2.2 Learning Rate

As mentioned in the section 1.2.1, the learning rate is a hyper-parameter of the Gradient Descent algorithm that determines the size of the steps taken on each epoch. In other words, it dictates how big are the steps taken into the direction of the local minimum. Thus, it indicates how fast or slow the movement is towards the optimal weights [10].

In most of the cases, the initial value of this parameter is set manually. In order to reach the local minimum, it is necessary to set the learning rate to an appropriate value, which is neither too low nor too high [10]. If the learning rate is too high, the steps taken are too big. Thus, a high learning rate can cause the system to diverge in terms of the objective function [13]. On the other hand, if the steps taken are too small, it results in slow learning [13], and the algorithm could also never converge. Therefore, determining a good learning rate becomes an important task.

1.3 State of the art

The determination of the learning rate is a topic that has been addressed previously in the literature. Some of the studies performed on this field are found in the following academic papers: the first one is titled *ADADELTA: An Adaptive Learning Rate Method*, written by Matthew D. Zailer. This article presents a method for the estimation of the learning rate for each of the dimensions of the input variables. The method dynamically adapts over time using only first order information, and requires no manual tuning of the learning rate [13].

There are also other optimizers that automatically determine the learning rate. For example, Adagrad is an optimizer that modifies the learning rate adapting to the direction of the descent towards the optimal value. As explained in the academic paper entitled *Adagrad - An Optimizer for Stochastic Gradient Descent*, Adagrad maintains low learning rates for frequently occurring features and high learning rates for less frequently occurring features [14].

Besides, Adam and RMSProp are two of the most influential adaptive stochastic algorithms for training deep neural networks. On the article entitled *A Sufficient Condition for Convergences of Adam and RMSProp* an alternative sufficient condition is proposed, which merely depends on the parameters of the base learning rate and combinations of historical second-order moments, to guarantee the global convergence of generic Adam/RMSProp for solving large-scale non-convex stochastic optimization [15].

In addition, Pierre Baldi presents a paper entitled *Gradient Descent Learning Algorithm Overview: A General Dynamical Systems Perspective*, where he gives a unified treatment of gradient descent learning algorithms for neural networks using a general framework of dynamical systems.This general approach organizes and simplifies all the known algorithms and results, and can also be applied to derive new algorithms. The author then briefly examines some of the complexity issues and limitations intrinsic to gradient descent learning [16].

Finally, in the thesis work by Juan Camilo Ruiz, graduate student from MACC, entitled "*Una propuesta de neurona artificial: la Unidad Neuro Vascular Artificial (UNVA)*", on chapter 4, he explains the conditions the learning rate from an artificial neuron needs to meet to guarantee the convergence of the gradient descent algorithm when using a linear regression model. This is done by modeling the evolution of the coefficients of the architectures as a dynamical system. In other words, the equations from the gradient descent algorithm are presented as a dynamical system, and two upper-bounds are found for the learning rate [1].

1.4 Problem Statement

In machine learning, and statistics, the learning rate is a tuning parameter in an optimization algorithm that determines the step size at each iteration, while moving toward a minimum of a loss function [17]. As mentioned before, it determines how big are the steps taken into the direction of the local minimum. Thus, it represents how fast a machine learning model learns. Most of the times a fast learning is desired, therefore high learning rates are selected. However, as explained in 1.2.2, it is necessary to determine a correct learning rate to avoid the divergence of the model.

Previously, a method to determine the upper-bound for the learning rate of models based on linear regression problems was presented in the thesis work done by Juan Camilo Ruiz [1]. There, he analyzed the gradient descent algorithm as a discrete dynamical system, and the upper-bound arises as criteria to determine the stability of this system. Thus, this thesis work aims to extend these results to models based in classification and multinomial regression. In addition, we seek to find an optimal value for the learning rate of these models.

1.5 Objectives

The main objective of this thesis is to develop an algorithm that automatically determines the optimal value for the Learning Rate of classification and multinomial regression models.

To reach this main objective, the following specific objectives for the thesis are proposed:

1. To generalize the results obtained by Juan Camilo Ruiz [1], where he found and upper-bound for the learning rate of models based on linear regression, to other activation functions.
2. To find an upper-bound and the optimal value for the learning rate, using regression and classification models.
3. To extend the results from the logistic regression model to a multinomial regression model.

Chapter 2

Dynamical Systems

In this chapter we present a theoretical explanation of Dynamical Systems. Once this concept is explained, we illustrate how the gradient descent algorithm can be modeled as a dynamical system. Some of the derivations and the theoretical fundamentals on this chapter are based on the Chapter 4 of the thesis work by Juan Camilo Ruiz [1].

2.1 Dynamical Systems

A Dynamical System describes the state of some variables, or phenomena, at any instant. These variables are called *state variables*, which evolve with time, and can be described in discrete time, i.e. fixed time instants separated by an specific quantity, or continuous time. Dynamical systems based in variables described in discrete time are modeled by difference equations, whereas dynamical systems based in variables described in continuous time are modeled by differential equations.

We are interested in studying discrete dynamical systems, as the variables we want to model are the coefficients ω of the gradient descent algorithm, which evolve in discrete time, described by each epoch of the algorithm.

A discrete dynamical system with m linear related variables, with constant coefficients, can be described in a general way as follows:

$$\begin{aligned} x_1[n+1] &= a_{11}x_1[n] + \cdots + a_{1m}x_m[n] + b_1 \\ x_2[n+1] &= a_{21}x_1[n] + \cdots + a_{2m}x_m[n] + b_2 \\ &\vdots \\ x_m[n+1] &= a_{m1}x_1[n] + ... + a_{mm}x_m[n] + b_m. \end{aligned}$$

It can also be written in matrix way as:

$$\boldsymbol{x}[n+1] = \boldsymbol{A}\boldsymbol{x}[n] + \boldsymbol{b}, \tag{2.1}$$

where:

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix}, \quad \boldsymbol{x}[n] = \begin{bmatrix} x_1[n] \\ \vdots \\ x_m[n] \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}.$$

The *equilibrium* of a discrete dynamical system is a simple but fundamental concept that helps us to analyze complicated behaviours of the variables. It is a solution where the state variable does not change over time, i.e. the variable is a constant. In other words, the vector $\boldsymbol{x}^*$ is an equilibrium if, once the vector of state variables of the system takes the value of $\boldsymbol{x}^*$, it remains constant for the future instants. Thus, if we have an equilibrium for the system defined in equation (2.1), then, we get that:

$$\boldsymbol{x}^* = \boldsymbol{A}\boldsymbol{x}^* + \boldsymbol{b}. \tag{2.2}$$

Moreover, an equilibrium of a dynamical system can be *stable* or *unstable*. This depends on whether the trajectories that start near the equilibrium keep near this point, or move away. Formally, an equilibrium point $\boldsymbol{x}^*$ is called *asymptotically stable* if, for every initial condition, the vector of state variables tends to $\boldsymbol{x}^*$ over time [1].

We want now to illustrate these concepts. The following considerations are based in the calculations presented in the thesis work by Juan Camilo Ruiz [1]. For this purpose, consider a linear non homogeneous system as the one defined in equation (2.1), and suppose the vector $\boldsymbol{x}^*$ is an equilibrium of that system. If we subtract equation (2.2) from equation (2.1), we have that:

$$\begin{aligned} \boldsymbol{x}[n+1] - \boldsymbol{x}^* &= \boldsymbol{A}\boldsymbol{x}[n] + \boldsymbol{b} - (\boldsymbol{A}\boldsymbol{x}^* + \boldsymbol{b}) \\ &= \boldsymbol{A}(\boldsymbol{x}[n] - \boldsymbol{x}^*). \end{aligned}$$

Lets call $\boldsymbol{z}[n]$ the state variables vector, which is equal to:

$$\boldsymbol{z}[n] = \boldsymbol{x}[n] - \boldsymbol{x}^*.$$

Note that $\boldsymbol{x}[n]$ tends to the equilibrium $\boldsymbol{x}^*$ if and only if $\boldsymbol{z}[n]$ tends to the $\boldsymbol{0}$ vector as n increases. Lets analyze, then, the new system:

$$\boldsymbol{z}[n+1] = \boldsymbol{A}\boldsymbol{z}[n].$$

One can observe that the behaviour of the system defined above is determined by the behaviour of the matrix $\boldsymbol{A}$. Therefore, the stability of the matrix $\boldsymbol{A}$ determines the stability of the system. Let $\lambda_1, \lambda_2, \cdots, \lambda_m$ be the eigenvalues of the matrix $\boldsymbol{A}$. According to the results shown by Juan Camilo Ruiz [1], the system is asymptotically stable if all the eigenvalues of the matrix $\boldsymbol{A}$ are inside the unit circle in the complex plane. Besides, if at least one eigenvalue λ is outside the unit circle, the system is unstable, i.e. it diverges. Finally, if one eigenvalue λ has magnitude equal to 1, i.e. the eigenvalue is a point over the unit circle, the system has an oscillatory behaviour around an equilibrium point.

2.2 Gradient Descent and Dynamical Systems

In this section we explain how the gradient descent algorithm can be modeled as a discrete dynamical system, and what are the effects the learning rate has on the convergence of this system.

2.2.1 Gradient Descent as a Dynamical System

According to section 1.2.1, the vector $\boldsymbol{\omega}$ of coefficients is updated in each iteration of the gradient descent algorithm. Therefore, we can understand this process as a discrete dynamical system, where the state variables are each of the coefficients ω. We follow the same analysis as before; however, the update formula is now defined as follows:

$$\boldsymbol{\omega}[n+1] = \boldsymbol{\omega}[n] - \eta \frac{\partial J}{\partial \boldsymbol{\omega}}, \tag{2.3}$$

where

$$\frac{\partial J}{\partial \boldsymbol{\omega}} = \frac{\partial J}{\partial h_i} \frac{\partial h_i}{\partial \boldsymbol{\omega}}.$$

2.2.2 Effects of the Learning Rate on the Convergence of Gradient Descent Algorithm

As mentioned in section 1.2.2, the learning rate influences the convergence of the gradient descent algorithm. Having a high value for this parameter can lead to the divergence of the model, as the minimum is never found, whereas having a low value can cause a slow learning process, and the algorithm may never find the minimum, i.e. never converge.

In order to understand more deeply the influence this parameter has on the convergence of the algorithm, consider the gradient descent algorithm modeled as a discrete dynamical system, where the update formula is given by equation (2.3). The evolution of the cost function on each epoch, for different values of the learning rate η is shown in figure 2.1.

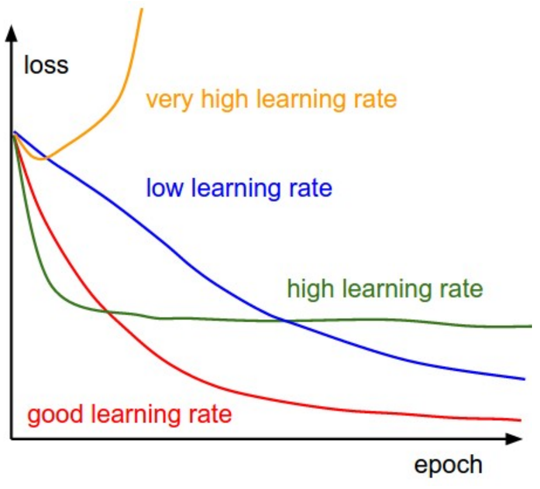

FIGURE 2.1: Evolution of the cost function depending on the learning rate η. *Credit: cs231n*

In the figure 2.1 shown above one can understand how the cost function evolve in time (epochs) for each choice of the learning rate. The loss is the same at the beginning, as the vector of weights $\boldsymbol{\omega}$ is initialized with the same values in all cases.

However, one can note that the learning rate has a significant influence in the evolution of the training error.

For the yellow curve, a very high learning rate was chosen, and we can observe that the algorithm diverges, as the cost function diverges. On the other hand, the green curve represents a high choice of the learning rate, smaller than the one represented with the yellow curve. In this case, even though the error was reduced, the algorithm does not find the minimum, and thus converges abruptly to a value that is not the minimum of the cost function. The red curve represents a good choice for the learning rate. We can observe that the error is reduced, and the cost function converges to the minimum of this function. Finally, for the blue curve a low learning rate was chosen, and the figure shows that despite the cost function converges to the minimum value, the number of epochs needed is considerably higher than the previous case.

The dynamic of the gradient descent is modeled in the figure 2.2 for the different learning rates mentioned before. The color code corresponds to the cases indicated in 2.1.

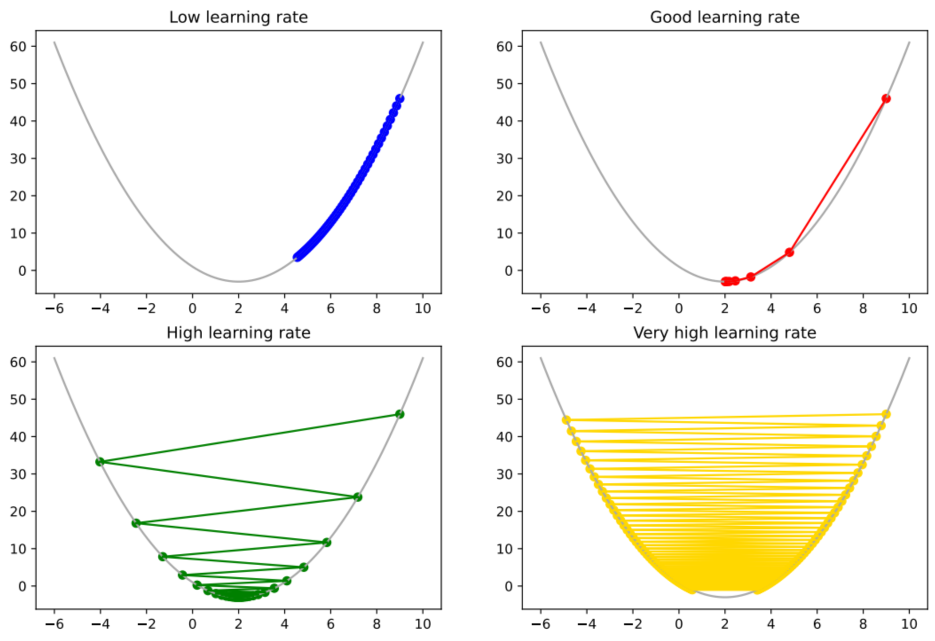

FIGURE 2.2: Dynamic of gradient descent for different learning rates. Image by author.

From the figure 2.2 one can observe that, for the low learning rate, the algorithm does not converges, because the number of iterations needed to find the minimum of the function is really high. However, the good learning rate allows the algorithm to find the minimum of the function in a short number of iterations. On the other hand, when the learning rate is high, the algorithm takes big steps, making more difficult to find the minimum. Finally, the very high learning rate takes even bigger steps than before, and does not find the minimum.

Therefore, while using gradient descent, it is important to evaluate the evolution of the error, in order to determine if the value of the hyper-parameters chosen guarantee the convergence of the algorithm [1].

Chapter 3

Automatic estimation of the Learning Rate

In this chapter we explain the algorithm to find the upper-bound for the Learning Rate when using gradient descent for linear and logistic regression. This upper-limit is determined by analyzing gradient descent as a dynamical system. The derivation is also generalized to other activation functions. In addition, in this chapter we propose an optimal value for the Learning Rate, and we demonstrate that this value is always lower than the upper-limit established before. Finally, we generalize these results to a multinomial logistic regression, which can be interpreted as a layer of a neural network, i.e. a multilayer perceptron neural network.

3.1 Regression

This section provides a brief explanation of what Linear and Logistic Regression problems are, their differences, and how they are addressed in Machine Learning.

Regression is a technique used in supervised machine learning, where the objective is to determine the relationship between independent variables (or features) and dependent variables (or outcomes) [18]. It is used as a method for predictive modelling, where an algorithm predicts continuous outcomes. Thus, regression problems are one of the most common applications in machine learning models. In order to solve them, the algorithms are trained with the purpose of understanding the relationship between the independent and dependent variables.

Most of the regression models are described according to how the outcome variables are modeled, e.g. in *linear regression* the outcome is continuous, whereas in *logistic regression* the outcome is dichotomous, i.e. there are only two possible values [18].

Regression can also be divided into two different types of methods: methods with multiple input variables, which are known as *multivariable*, and methods with multiple outputs.

3.1.1 Linear Regression

Linear Regression is probably one of the most common algorithms in statistics and machine learning. This type of regression is a linear model, that means that is a model that assumes a linear relationship between the model parameters, and the outcome, or dependent variables. In other words, the outcome can be calculated from a linear combination of the input variables [19].

The representation of a linear regression problem is a linear equation, which combines the input values to provide an estimated output [18]. In addition, another coefficient, known as the *intercept*, or the *bias coefficient* is also found.

There are lots techniques to solve a linear regression problem. In machine learning, the most used is Gradient Descent. This technique optimizes the values of the coefficients by minimizing the error of the model on the training data, iteratively. It starts with a random value for each coefficient, and calculates the sum of the squared errors for each pair of input and output values [19].

In this technique, the *learning rate* is used to determine the size of the step to take in each iteration, seeking the parameters that minimize the root mean squared error (RMSE).

3.1.2 Logistic Regression

Logistic Regression is also a very well known algorithm in machine learning, and is used to predict a categorical variable, or set of variables, in case of multinomial regression, given a set of observations. The output of this algorithm is a categorical or discrete value, i.e. it can take only two possible values. It is very similar to linear regression. Though, while linear regression is used for solving regression problems, logistic regression is used for solving classification problems [20].

In this case, coefficients are also assigned to each independent variable. However, there is no regression line to be fitted, but a curve with the shape of a logistic function, which predicts values between (0 or 1). This function is called sigmoid function, and is used to map the predicted values to probabilities. It maps any real value within a range of 0 and 1 [20]. Figure 3.1 shows the logistic function:

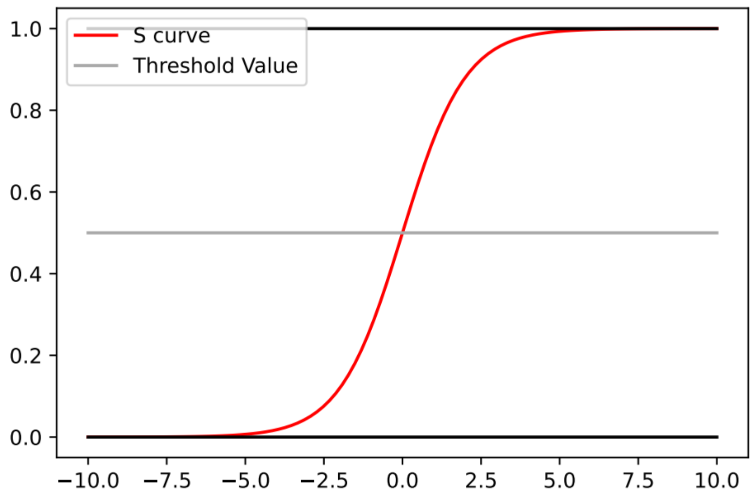

FIGURE 3.1: Logistic Function (Sigmoid Function). Image by author.

3.2 Determination of the upper-bound for the Learning Rate

In this section we provide a detailed explanation on how the upper-bound for the learning rate is found for a logistic regression problem, analyzing the gradient descent algorithm as a discrete dynamical system. We also extend these results to other activation functions.

3.2.1 Logistic Regression Problem

In the thesis work by Juan Camilo Ruiz [1], he proposed an upper-bound for the learning rate when using gradient descent in a linear regression problem. Following this procedure, we extend these results for logistic regression problems.

Logistic regression is characterized by the activation function:

$$h_\omega(\mathbf{x}_i) = \frac{1}{1+e^{-\omega^T \mathbf{x}_i}}, \tag{3.1}$$

where $\mathbf{x}_i \in \mathbb{R}^{d+1}$ is a $d+1$-dimensional vector that represents the i-th observation, $i = \{1, \ldots, N\}$ is an scalar indicating an observation, and $\boldsymbol{\omega} \in \mathbb{R}^d$ is the vector of parameters. To solve the regression problem, we look for the values of $\boldsymbol{\omega}$ that minimize the cross-entropy function for all the observations. The cross-entropy function, for the binary case, is given by:

$$J(\omega) = -\frac{1}{N}\sum_i y_i \log(h_\omega(\mathbf{x}_i)) + (1-y_i)\log(1-h_\omega(\mathbf{x}_i)), \tag{3.2}$$

where y_i is the output associated to the input $\mathbf{x}_i$. Equation (3.2) is known as the cost function. To minimize this function we use gradient descent. Interpreting gradient descent as a discrete dynamical system, as explained in the Chapter 2, the update formula looks as follows:

$$\boldsymbol{\omega}[n+1] = \boldsymbol{\omega}[n] - \eta\frac{\partial J}{\partial \boldsymbol{\omega}}, \tag{3.3}$$

where:

$$\frac{\partial J}{\partial \omega} = \frac{\partial J}{\partial h_\omega}\frac{\partial h_\omega}{\partial \omega}.$$

The derivative of the cost function given by equation (3.2) can be found by calculating both derivatives expressed before. In the first place, we have that:

$$\begin{aligned}
\frac{\partial J}{\partial h_\omega} &= -\frac{1}{N}\sum_i y_i\left(\frac{1}{h_\omega(\mathbf{x}_i)}\right) + (1-y_i)\left(\frac{-1}{1-h_\omega(\mathbf{x}_i)}\right)\\
&= -\frac{1}{N}\sum_i \left(\frac{y_i}{h_\omega(\mathbf{x}_i)}\right) - \left(\frac{1-y_i}{1-h_\omega(\mathbf{x}_i)}\right)\\
&= \frac{1}{N}\sum_i \left(\frac{1-y_i}{1-h_\omega(\mathbf{x}_i)}\right) - \left(\frac{y_i}{h_\omega(\mathbf{x}_i)}\right).
\end{aligned}$$

On the other hand, we have that:

$$\begin{aligned}
\frac{\partial h_\omega}{\partial \omega_j} &= \frac{x_i^{(j)} e^{-\omega^T x_i}}{(1+e^{-\omega^T x_i})^2} \\
&= x_i^{(j)} \left[\frac{e^{-\omega^T x_i}}{(1+e^{-\omega^T x_i})^2} \right] \\
&= x_i^{(j)} \left[\frac{e^{-\omega^T x_i}+1-1}{(1+e^{-\omega^T x_i})^2} \right] \\
&= x_i^{(j)} \left[\frac{1}{1+e^{-\omega^T x_i}} - \frac{1}{(1+e^{-\omega^T x_i})^2} \right] \\
&= x_i^{(j)} (h_\omega - h_\omega^2) \\
&= x_i^{(j)} h_\omega (1-h_\omega).
\end{aligned}$$

Thus, taking into account these results, we have that:

$$\begin{aligned}
\frac{\partial J}{\partial \omega} &= \frac{\partial J}{\partial h_\omega} \frac{\partial h_\omega}{\partial \omega} \\
&= \left[\frac{1}{N} \sum_i \left(\frac{1-y_i}{1-h_\omega(\mathbf{x}_i)} \right) - \left(\frac{y_i}{h_\omega(\mathbf{x}_i)} \right) \right] x_i^{(j)} h_\omega(\mathbf{x}_i)(1-h_\omega(\mathbf{x}_i)) \\
&= \frac{1}{N} \sum_i (1-y_i) x_i^{(j)} h_\omega(\mathbf{x}_i) - y_i x_i^{(j)} (1-h_\omega(\mathbf{x}_i)) \\
&= \frac{1}{N} \sum_i x_i^{(j)} h_\omega(\mathbf{x}_i) - x_i^{(j)} y_i h_\omega(\mathbf{x}_i) - y_i x_i^{(j)} + x_i^{(j)} y_i h_\omega(\mathbf{x}_i) \\
&= \frac{1}{N} \sum_i x_i^{(j)} h_\omega(\mathbf{x}_i) - y_i x_i^{(j)} \\
&= \frac{1}{N} \sum_i (h_\omega(\mathbf{x}_i) - y_i) x_i^{(j)} \\
&= \frac{1}{N} (\boldsymbol{h} - \boldsymbol{y})^T \boldsymbol{x}^{(j)}.
\end{aligned}$$

Which in vectorial form looks like:

$$\frac{\partial \mathbf{J}}{\partial \boldsymbol{\omega}} = \frac{1}{N} \mathbf{X}^T (\mathbf{h} - \mathbf{y}),$$

where:

$$\mathbf{X} = \begin{bmatrix} \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(d)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \end{bmatrix},$$

with $\mathbf{X} \in \mathbb{R}^{N \times (d+1)}$, $\mathbf{h} \in \mathbb{R}^{N \times 1}$, $\boldsymbol{\omega} \in \mathbb{R}^{(d+1) \times 1}$ and $\mathbf{y} \in \mathbb{R}^{N \times 1}$. That means that there are N observations and $d+1$ variables. Then, equation (3.3) can be written as:

$$\boldsymbol{\omega}[n+1] = \boldsymbol{\omega}[n] - \frac{\eta}{N} \mathbf{X}^T (\mathbf{h} - \mathbf{y}).$$

But, since h depends on $\boldsymbol{\omega}[n]$, we have that:

$$\boldsymbol{\omega}[n+1] = \boldsymbol{\omega}[n] - \frac{\eta}{N}\mathbf{X}^T\mathbf{h}_\omega - \frac{\eta}{N}\mathbf{X}^T\mathbf{y}, \tag{3.4}$$

where:

$$\mathbf{h}_\omega = \frac{1}{1+e^{-\mathbf{X}\omega}}. \tag{3.5}$$

To linearize the system presented in equation (3.4), we will use $\boldsymbol{\omega}[n+1] = \mathbf{A}\boldsymbol{\omega}[n]$, where $\mathbf{A}$ is the Jacobian of the following function:

$$f_\omega = \boldsymbol{\omega}[n] - \frac{\eta}{N}\mathbf{X}^T h_\omega - \frac{\eta}{N}\mathbf{X}^T\mathbf{y},$$

where:

$$\boldsymbol{\omega}[n] = \begin{bmatrix} \omega_0[n] \\ \omega_1[n] \\ \vdots \\ \omega_d[n] \end{bmatrix}$$

Then, we have that the Jacobian of the function is:

$$\nabla_\omega f = \left[\frac{\partial f}{\partial \omega_1}\ \frac{\partial f}{\partial \omega_2} \cdots \frac{\partial f}{\partial \omega_d}\right]^T;$$

which can be written as:

$$\nabla_\omega f = \mathbf{I} - \frac{\eta}{N}\mathbf{X}^T\frac{\partial \mathbf{h}}{\partial \boldsymbol{\omega}}. \tag{3.6}$$

On the other hand, the derivative of the function h_ω shown in equation (3.5), for each observation, is given by:

$$\begin{aligned}\frac{\partial h_i}{\partial \omega_j} &= \frac{x_i^{(j)} e^{-\omega^T x_i}}{(1+e^{-\omega^T x_i})^2} \\ &= x_i^{(j)}\left(\frac{e^{-\omega^T x_i}}{(1+e^{-\omega^T x_i})^2}\right) \\ &= x_i^{(j)} h_i(1-h_i).\end{aligned}$$

Therefore, the previous derivative can be written in matrix form as follows:

$$\frac{\partial \mathbf{h}}{\partial \boldsymbol{\omega}} = \begin{bmatrix} x_1^{(1)}h_1(1-h_1) & \cdots & x_1^{(d)}h_1(1-h_1) \\ x_2^{(1)}h_2(1-h_2) & \cdots & x_2^{(d)}h_2(1-h_2) \\ \vdots & \ddots & \vdots \\ x_N^{(1)}h_N(1-h_N) & \cdots & x_N^{(d)}h_N(1-h_N) \end{bmatrix}.$$

Note that:

$$\frac{\partial \mathbf{h}}{\partial \boldsymbol{\omega}} = \begin{bmatrix} h_1(1-h_1) & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & h_N(1-h_N) \end{bmatrix} \begin{bmatrix} 1 & x_1^{(1)} & \cdots & x_1^{(d)} \\ 1 & x_2^{(1)} & \cdots & x_2^{(d)} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N^{(1)} & \cdots & x_N^{(d)} \end{bmatrix}.$$

Hence, the derivative can be written as:

$$\frac{\partial \mathbf{h}}{\partial \boldsymbol{\omega}} = \mathbf{D}_f \mathbf{X},$$

where:

$$\mathbf{D}_f = diag[\mathbf{h} \odot (\mathbf{1} - \mathbf{h})], \tag{3.7}$$

where $\odot$ represents the element-wise product. Thus, equation (3.6) becomes:

$$\nabla_\omega f = \mathbf{I} - \frac{\eta}{N} \mathbf{X}^T \mathbf{D}_f \mathbf{X}.$$

Therefore, the gradient descent equation given by equation (3.4) can be written in a linearized way as:

$$\boldsymbol{\omega}[n+1] \approx \left[\mathbf{I} - \frac{\eta}{N} \mathbf{X}^T \mathbf{D}_f \mathbf{X}\right] \boldsymbol{\omega}[n]. \tag{3.8}$$

Then, to ensure that the linear system is stable, the eigenvalues of the matrix $\mathbf{A}$, defined as:

$$\mathbf{A} = \mathbf{I} - \frac{\eta}{N} \mathbf{X}^T \mathbf{D}_f \mathbf{X}$$

must be inside the unit circle.

Note that $\mathbf{D}_f$ is a diagonal matrix with positive values. Then:

$$\mathbf{X}^T \mathbf{D}_f \mathbf{X} = \mathbf{X}^T \mathbf{D}_f^{1/2} \mathbf{D}_f^{1/2} \mathbf{X} = (\mathbf{X} \mathbf{D}_f^{1/2})^T (\mathbf{X} \mathbf{D}_f^{1/2}).$$

Hence, $\mathbf{X}^T \mathbf{D}_f \mathbf{X}$ is a semidefinite positive matrix. On the other hand, we have that:

$$\begin{aligned} eig(\mathbf{A}) &= eig\left(\mathbf{I} - \frac{\eta}{N} \mathbf{X}^T \mathbf{D}_f \mathbf{X}\right) \\ &= \mathbf{1} - \frac{\eta}{N} eig(\mathbf{X}^T \mathbf{D}_f \mathbf{X}), \end{aligned}$$

where *eig* represents the eigenvalues. Then, to ensure that the system is stable, it must be satisfied that:

$$\left|1 - \frac{\eta}{N} \lambda\right| < 1.$$

For all the eigenvalues λ of the matrix given by $\mathbf{X}^T \mathbf{D}_f \mathbf{X}$. Solving the inequality, we have that:

$$\begin{aligned} -1 < 1 - \frac{\eta}{N} \lambda &< 1 \\ -2 < -\frac{\eta}{N} \lambda &< 0 \\ -2 < -\frac{\eta}{N} \lambda & \\ 2 > \frac{\eta}{N} \lambda & \\ \eta < \frac{2N}{\lambda}. & \end{aligned} \tag{3.9}$$

As the matrix is a semidefinite positive matrix, the smallest value this upper-bound takes is given when we have the greatest eigenvalue, λ_{max}. Therefore, the learning rate, given by η, must satisfy the inequality resulting in equation (3.9). This equation gives a condition for the system to be stable, as well as an upper limit for the learning rate when using gradient descent. Thus, the maximum value the learning rate can take is given by:

$$\eta_{\text{max}} = \frac{2N}{\lambda_{max}} \tag{3.10}$$

3.2.2 Extension to other Activation Functions

The derivation explained before can be generalized to any activation function. This is possible because, if we recall how the matrix $\mathbf{D}_f$ is defined in equation (3.7), we can observe that it depends only on the activation function, h. Hence, the gradient descent algorithm, defined by equation (3.8) depends also only on the activation function, and the matrix of observations $\mathbf{X}$. Therefore, based on this derivation, the same upper bound presented in equation (3.9) can be used for the learning rate of problems with different activation functions.

3.3 Determination of the optimal Learning Rate

Now, as the upper-limit for the Learning Rate is defined, we propose an optimal value for this hyperparameter. For this purpose, let's consider the function:

$$J(\eta) = \sum_i \left(1 - \frac{\eta}{N}\lambda_i\right),$$

which can be expressed in a vectorial form as:

$$J(\eta) = \left(1 - \frac{\eta}{N}\boldsymbol{\lambda}\right)^T \left(1 - \frac{\eta}{N}\boldsymbol{\lambda}\right), \tag{3.11}$$

where $\boldsymbol{\lambda} = \begin{bmatrix} \lambda_{min} \\ \lambda_{max} \end{bmatrix}$.

In order to minimize the distance of these eigenvalues to the origin in the unit circle, we need to minimize the function given by equation (3.11). The derivative of this function can be calculated as follows:

$$
\begin{aligned}
\frac{\partial J}{\partial \eta} &= \frac{\partial J}{\partial \eta}\left(\sum_i \left(1-\frac{\eta}{N}\lambda_i\right)^2\right)\\
&= \left(\sum_i \frac{\partial J}{\partial \eta}\left(1-\frac{\eta}{N}\lambda_i\right)^2\right)\\
&= \left(\sum_i 2\left(1-\frac{\eta}{N}\lambda_i\right)\left(\frac{\partial J}{\partial \eta}1-\frac{\partial J}{\partial \eta}\frac{\eta}{N}\lambda_i\right)\right)\\
&= 2\sum_i \left(1-\frac{\eta}{N}\lambda_i\right)\left(\frac{-\lambda_i}{N}\right)\\
&= 2\sum_i \left(\frac{-\lambda_i}{N}+\eta\frac{h_i^2}{N^2}\right)\\
&= 2\sum_i \left(\frac{-N\lambda_i+\eta\lambda_i^2}{N^2}\right)\\
&= \frac{2}{N^2}\sum_i(\lambda_i(\lambda_i\eta-N)).
\end{aligned}
$$

This derivative can be written in a vectorial form as:

$$
\frac{\partial J}{\partial \eta} = \frac{2}{N^2}\boldsymbol{\lambda}^T(\boldsymbol{\lambda}\eta - N\cdot\mathbf{1}). \tag{3.12}
$$

We seek the optimal value of equation (3.12) by minimizing the function given by this equation. To this effect, we set this equation equal to zero, and we solve:

$$
\begin{aligned}
0 &= \frac{2}{N^2}\boldsymbol{\lambda}^T(\boldsymbol{\lambda}\eta - N\cdot\mathbf{1})\\
0 &= \boldsymbol{\lambda}^T(\boldsymbol{\lambda}\eta - N\cdot\mathbf{1})\\
0 &= \boldsymbol{\lambda}^T\boldsymbol{\lambda}\eta - \mathbf{1}^T\boldsymbol{\lambda}N\\
\boldsymbol{\lambda}^T\boldsymbol{\lambda}\eta &= \mathbf{1}^T\boldsymbol{\lambda}N
\end{aligned}
$$

Then, solving for η, we get that the optimal value for the learning rate is:

$$
\eta_{\text{opt}} = \frac{\mathbf{1}^T\boldsymbol{\lambda}N}{\boldsymbol{\lambda}^T\boldsymbol{\lambda}} = \frac{N(\lambda_{min}+\lambda_{max})}{(\lambda_{min}^2+\lambda_{max}^2)}. \tag{3.13}
$$

To guarantee the convergence for this optimal η, it is necessary to prove that its value is smaller than the upper bound found. i.e. we need to prove that:

$$
\eta_{\text{opt}} < \frac{2N}{\lambda_{max}}
$$

Since $\mathbf{X}^T\mathbf{D}_f\mathbf{X}$ is semipositive definite, let's consider $\lambda_{min} = c\lambda_{max}$, with $c \in [0,1]$. Then, replacing in equation (3.13), we get:

$$
\eta_{\text{opt}} = \frac{(c+1)N}{(c^2+1)\lambda_{max}}.
$$

As we want to prove that the value of η_{opt} is smaller than the upper bound, we need to find the maximum value of the following function:

$$f(c) = \frac{c+1}{c^2+1},$$

which corresponds to the worst case for the eigenvalues of the matrix. Then, maximizing this function, we get that the value of c where the function $f(c)$ has its maximum is $c = -1 + \sqrt{2}$. Hence, evaluating the function in this value we get that the maximum of the function is:

$$f(c) = \frac{\sqrt{2}}{2} + \frac{1}{2} < 2.$$

Therefore, we have that, in the worst case scenario,

$$\eta_{\text{opt}} < \frac{2N}{\lambda_{max}}$$

Thus, for every value of the eigenvalues, the optimal value for η is smaller than the upper bound. Ergo, we have proven the convergence of the optimal value for the Learning Rate, established in equation (3.13).

3.4 Generalization to a Multinomial Regression

When using multinomial regression, the otuput of the model is given by:

$$\begin{aligned}\mathbf{Z} &= \mathbf{XW},\\ \mathbf{P} &= e^{Z} \odot \left(e^{Z}\mathbf{1}_h\right)^{-1};\end{aligned}$$

where $\mathbf{W} \in \mathbb{R}^{(d+1)\times h}$ is a matrix containing the parameters for the h different classifiers, $\mathbf{P} \in \mathbb{R}^{\mathrm{N}\times h}$ is the matrix of soft-max outputs, representing the probability of an input observation to belong to one of the h output classes; the rows of this matrix sum to one, the $\odot$ symbol represents the element-wise product of matrices, i.e. the *Hardaman* product; and the right side of the equations represent a matrix containing in its rows the normalization terms for each observation, with $\mathbf{1}_h \in \mathbb{R}^h$ is a vector of ones with dimension equal to the number of output classes.

When using gradient descent the update rule in matrix form is given by:

$$\mathbf{W}[n+1] = \mathbf{W}[n] - \frac{\eta}{\mathrm{N}}\mathbf{X}^{\mathrm{T}}\left(\mathbf{P} - \mathbf{Y}\right),$$

where $\mathbf{Y} \in \mathbb{R}^{\mathrm{N}\times h}$ is the matrix containing the output for each observation. If we rewritte this equation independently for each output, we get:

$$\boldsymbol{\omega}_j[n+1] = \boldsymbol{\omega}_j[n] - \frac{\eta_j}{\mathrm{N}}\mathbf{X}^{\mathrm{T}}\left(\mathbf{p}_j - \mathbf{y}_j\right),$$

where $\boldsymbol{\omega}_j$ is the j-th column of $\mathbf{W}$, $\mathbf{p}_j$ is the j-th column of $\mathbf{P}$, $\mathbf{y}_j$ is the j-th column of $\mathbf{Y}$, and η_j is the learning rate associated to this output. Linearizing this equation we get $\mathbf{A} = \mathbf{I} - \frac{\eta_j}{\mathrm{N}}\mathbf{X}^{\mathrm{T}}\left(\mathbf{D}_f\right)_j\mathbf{X}$ just as in previous sections; with $\left(\mathbf{D}_f\right)_j$ a diagonal matrix containing the derivatives of the logistic function $\frac{\partial \mathbf{p}_j}{\partial \boldsymbol{\omega}_j} = \mathbf{p}_j \odot \left(\mathbf{1}_{\mathrm{N}} - \mathbf{p}_j\right)$. Thereby, the maximum and optimal values for the learning rate are given by:

$$\eta_{\max}^{(j)} = \frac{2\mathrm{N}}{\left(\lambda_{\max}\right)_j}, \text{ and } \quad \eta_{\text{opt}}^{(j)} = \mathrm{N}\frac{\mathbf{1}_2 \left(\boldsymbol{\lambda}_e\right)_j}{\left(\boldsymbol{\lambda}_e\right)_j^{\mathrm{T}} \left(\boldsymbol{\lambda}_e\right)_j};$$

where $\left(\lambda_{\max}\right)_j$ and $\left(\lambda_{\min}\right)_j$ are obtained from the eigen-values of $\mathbf{X}^{\mathrm{T}} \left(\mathbf{D}_f\right)_j \mathbf{X}$.

Chapter 4

Simulations and Results

In this chapter we present experimental results for the simulations performed to test the results obtained before. First, we present the data used in each simulation. We also explain how the simulations were designed, and we discuss the obtained results.

4.1 Data discussion

We first use some toy examples with manually created data to study the behaviour of the algorithm for the linear regression and the logistic regression model. Then, we use the *MNIST* and *Fisher Iris* datasets to validate our approach in real datasets.

In the first place, for the linear regression we randomly created 200 samples for a $1-$dimensional vector, as the input data. Then, we inserted a column of ones at the end, as the bias, in order to build the regressor matrix. We also generated 2 random numbers, as the true coefficients for the model.

For the logistic regression we generated samples for a binary classification problem, i.e. we manually created two classes. For each class, we draw 1000 samples for a normally distributed random variable. The first class with data around $(1, 1)$, i.e. with a mean value of 1, and the second class with data around $(-1, -1)$, i.e. with a mean value of -1. We created a vector of ones as the labels for class 1, and a vector of zeros as the labels for class 2. After these vectors were created, we concatenated the data for both classes and added a vector of ones for the bias, and we also created the output vector, concatenating both vectors of labels.

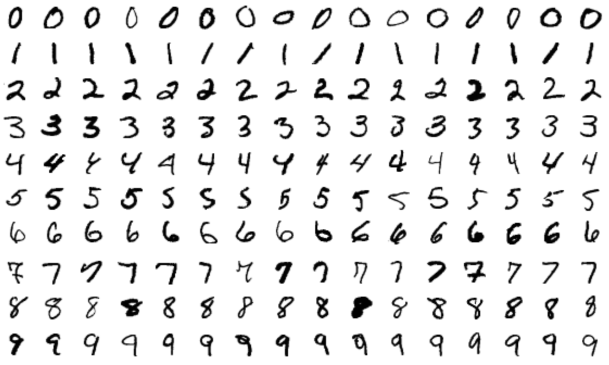

FIGURE 4.1: Sample images from MNIST test dataset. *Credit: MNIST database*.

MNIST and Fisher Iris were used for a multinomial logistic regression case. The first one is a database of hand written digits, commonly used for training various image processing systems, [21], as well as training and testing in the field of machine learning [22]. It contains 60.000 training images and 10.000 testing images. Figure 4.1 shows some sample images from this database. For the simulation, we transferred the data from the database to a matrix, by reshaping it, and we added the bias vector. Then, we converted the labels to one-hot encoding, which is a way of label encoding.

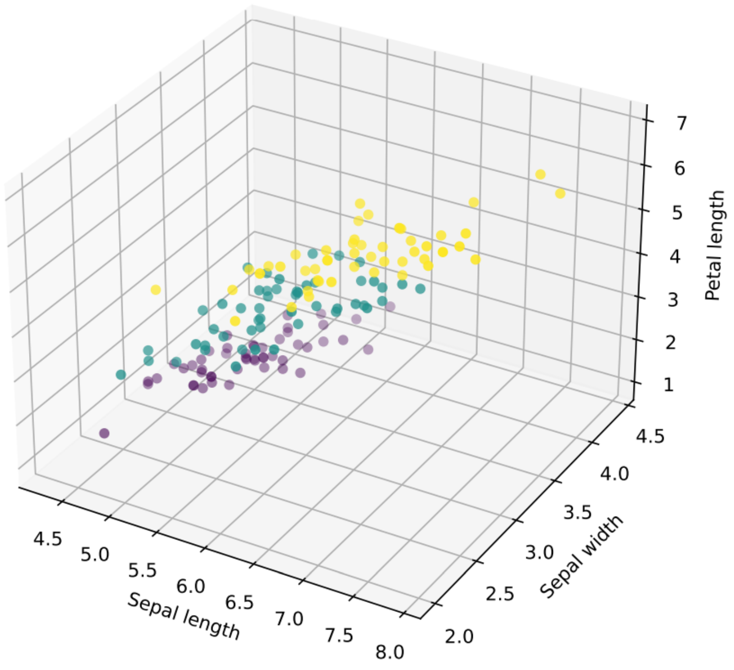

FIGURE 4.2: First 3 variables of Iris dataset. Image by author.

On the other hand, Fisher Iris is perhaps the best known database to be found in the pattern recognition literature [23]. The dataset contains 3 classes of 50 instances each, where each class refers to a type of iris plant. Figure 4.2 shows the first three columns of the dataset: *Sepal length*, *Sepal width* and *Petal length*, and how the 3 types of iris plants are scattered. For the simulation, we first extracted the measurements, and subtracted the mean from the data. Then, we added the bias vector. In this case, we also converted the labels into one-hot encoding.

4.2 Simulations design

For the linear regression problem simulation, as mentioned above, we chose a problem with 2 parameters. This with the purpose of nicely displaying the convergence of the algorithm. The problem was solved using batch gradient descent. We performed three simulations: the first one using the upper-bound for the learning rate, computed as mentioned in equation (3.10); the second one with the optimal value of the learning rate, computed as shown in equation (3.13); and the third one with a sub-optimal value of the learning rate, using the trace of the matrix.

Likewise, for the logistic regression we also chose a problem with two parameters. In this case, the algorithm was trained for 100 epochs. We repeated the experiment 100 times, creating new data from the same distribution each time, in order to observe the variability due to the input data. The algorithm was tested in each repetition using 200 new samples from the same distribution as in training, 100 per class.

On the other hand, with the MNIST and Iris datasets, we run simulations using a multinomial logistic regression. In the previous cases, we used batch gradient descent; however, in this example we used mini-batches, with the purpose of validating that our approach is still valid in this context. After each epoch of the algorithm the training data was randomly shuffled, and performed 10 repetitions each time initializing in the same point, in order to generate confidence intervals for the simulations.

4.3 Results

In this section we present the results obtained from the simulations and experiments described above. All of these simulations where performed in MATLAB.

On the first place, figure 4.3 shows the data created for the linear regression, and the results of the line corresponding to the linear regression obtained.

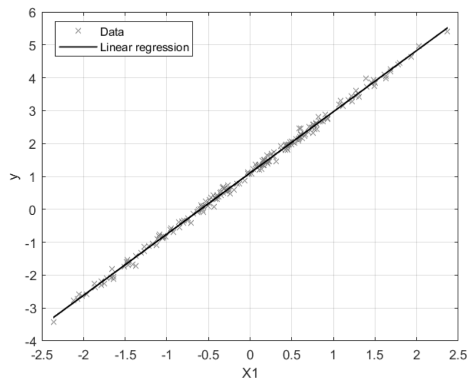

FIGURE 4.3: Results for the linear regression.

Figure 4.4 shows as well the results for the linear regression. One can observe that, when using the maximum value for the learning rate, given by the upper bound found, η_{max}, the algorithm oscillates between two points in the cost function. This is reflected in a constant value for the cost function which does not reach the minimum, and in values for the parameters $\boldsymbol{\omega}$ that oscillate. Meanwhile, when using $\eta_{\text{sub-opt}}$, we can observe that parameters converge slowly to the true values, and the cost function also converges slowly to the minimum. However, when using the optimal value for the learning rate, η_{opt}, the cost function converges to the minimum in a faster way, as well as the parameters, which converge fast to the true values.

Besides, figure 4.5 shows the evolution of the different learning rates in epochs for the linear regression. We can observe that each value of the learning rate remains constant through time, as the matrix never changes, thus, the eigen-values do not change either.

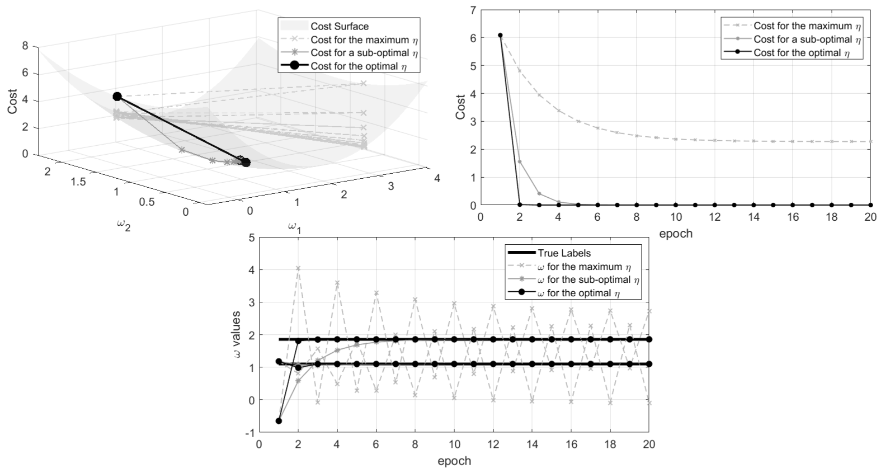

FIGURE 4.4: Results from gradient descent in a linear regression problem, for three different learning rates. In the upper-left figure the cost function and the trajectory followed by the parameters update. In the upper-right figure the evolution of the cost function in epochs. In the bottom figure the model parameters and the trajectory followed by the algorithm with the different learning rates.

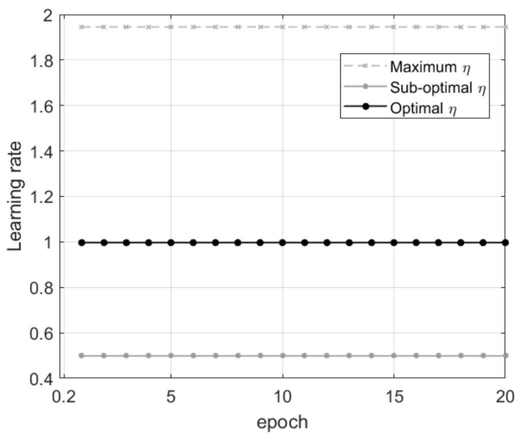

FIGURE 4.5: Evolution of the learning rate in epochs for the linear regression.

Figure 4.6, on the other hand, shows the results for the logistic regression. It is possible to observe that, in a similar way as the linear regression problem, when we use η_{max}, the algorithm oscillates between 2 points on the cost surface. Besides, the algorithm reaches the minimum for the cost function after 100 epochs. We can also observe that, for this learning rate, one of the parameters converges faster than for the other values of the learning rate; however, for the other parameter it oscillates, slowing down its global convergence. On the other hand, when using $\eta_{\text{sub-opt}}$, the algorithm converges in a slow way to the minimum, reaching this optimal value in approximately 33 epochs for the 100 repetitions. Finally, when using the optimal learning rate η_{opt}, the algorithm reaches the minimum of the cost function in 22 epochs. In contrast with the case for linear regression, here $\eta_{\max}$ and η_{opt} are approximations of the linear system in each iteration.

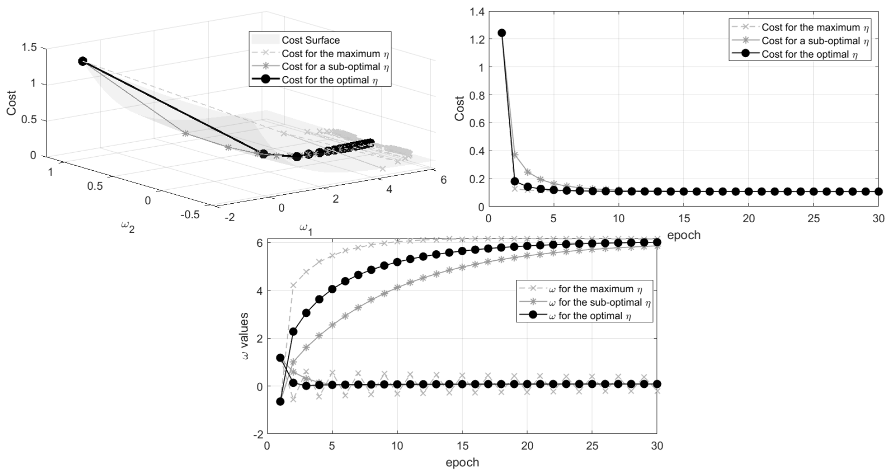

FIGURE 4.6: Results from gradient descent in a logistic regression problem, for three different learning rates. In the upper-left figure the cost function and the trajectory followed by the parameters update. In the upper-right figure the evolution of the cost function in epochs. In the bottom figure the model parameters and the trajectory followed by the algorithm with the different learning rates.

Moreover, figure 4.7 shows the evolution of the three values of the learning rate in epochs for the logistic regression. It shows that each value of the learning rate changes over time. This happens because the derivatives in the activation function decrease as reaching the minima, therefore, the matrix changes, making the learning rate to increase in each epoch.

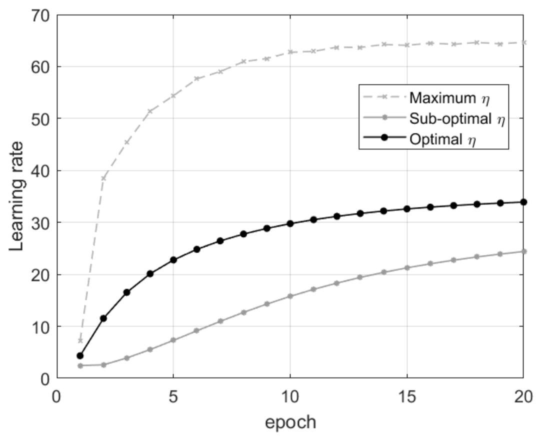

FIGURE 4.7: Evolution of the learning rate in epochs for the logistic regression.

In addition, figure 4.8 shows the results for the multinomial logistic regression problem, simulated using the Iris dataset. One can observe that even though the cost function did not converge for any of the values of η, the cost was considerably reduced.

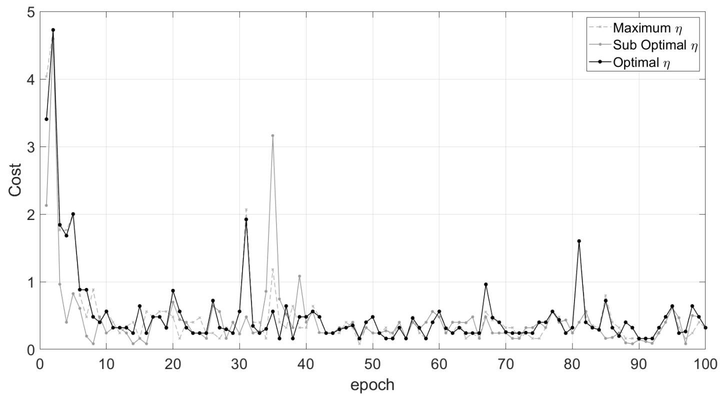

FIGURE 4.8: Evolution of the cost function in epochs for the multinomial logistic regression with Iris dataset.

Finally, figure 4.9 shows the results for the multinomial logistic regression problem, simulated with the MNIST dataset. For this simulation we could not perform the algorithm exactly as presented, since each mini-batch took approximately 2.6 ± 0.094 seconds to process, running the simulation for 100 epochs represents around 33 hours of processing time. The main bottleneck here is the eigen value decomposition. Therefore, in order to accelerate the algorithm, we used the lower-bound for the eigenvalues, $\lambda_{\min} = 0$, and an upper bound as presented in [24], given by:

$$\lambda_{\max} \leq \frac{\operatorname{tr}\left(\mathbf{X}^{\mathrm{T}}\mathbf{D}_f\mathbf{X}\right)}{d} + \left[\frac{d-1}{d}\left(\left|\left|\mathbf{X}^{\mathrm{T}}\mathbf{D}_f\mathbf{X}\right|\right|_F^2 - \frac{\operatorname{tr}\left(\mathbf{X}^{\mathrm{T}}\mathbf{D}_f\mathbf{X}\right)^2}{d}\right)\right]^{\frac{1}{2}},$$

where $\operatorname{tr}(\cdot)$ is the trace of the matrix, and $||\cdot||_F^2$ is the Frobenius norm. With this approximation, the processing time for each mini-batch was 92.8 ± 11 ms, completing 100 epochs in 1 hour and 12 minutes.

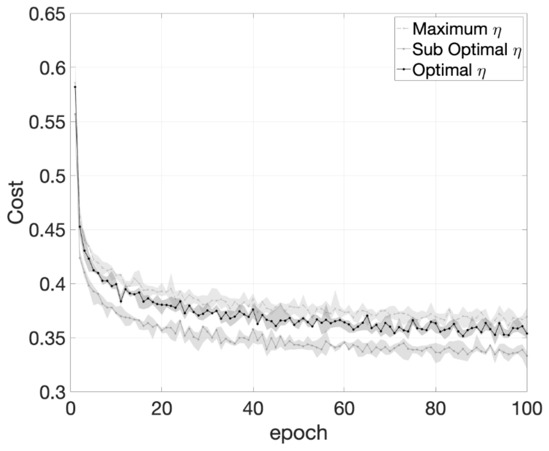

FIGURE 4.9: Evolution of the cost function in epochs for the multinomial logistic regression with MNIST dataset.

Chapter 5

Discussion

In this thesis we used an analysis of the gradient descent algorithm, studied as a dynamical system, to determine the upper-bound for the learning rate, in a way that the algorithm converges. Furthermore, we proposed an optimal value for the learning rate, which minimizes the sum of the distance of the extreme poles of this dynamical system. This analysis was performed by linearizing the gradient descent algorithm, and applying it to linear, logistic and multinomial regression. The upper-bound and the optimal value for the learning rate, given by η_{max} and η_{opt}, respectively, are approximations to the optimal value that guarantee the fastest convergence of the algorithm.

If we check again the results for the linear regression shown in figure 4.4, we can observe that the value for the learning rate that presents the fastest convergence is the optimal one, given by η_{opt}. For this value, the minimum of the cost function is reached in only 2 epochs. In addition, the model parameters have a similar behaviour, converging to their true values in about 3 epochs of the algorithm. In contrast, when we use $\eta_{\text{sub-opt}}$, the algorithm needs about 5 epochs to reach the minima of the cost function, and about 7 epochs to reach the true values of the model parameters. Thus, the algorithm converges in both cases, but it converges faster when using η_{opt} than when using $\eta_{\text{sub-opt}}$. However, when using the maximum value for the learning rate η_{max}, the algorithm gets stuck between two points on the cost function surface. Then, the algorithm converges to a sub-optimal value in the cost function and the model parameters do not converge, and oscillate between 2 values. This means that, in this case, at least one pole of the dynamical system representing the gradient descent is located over the unit circle, making the solution to oscillate.

In the case of the logistic regression, we can observe from figure 4.6 that when we use the upper-bound for the learning rate, given by η_{max}, the algorithm converges faster than the other two values of the learning rate. The reason why this happens is because, in this case, η_{max} is just an approximation from below to the real upper-bound of this parameter, i.e. there are values of the learning rate $\eta > \eta_{\text{max}}$ for which the algorithm still converges. Nevertheless, we can observe that, for this value of the learning rate, one of the model parameters oscillate between 2 points. This might indicate that at least one of the poles of the dynamical system is close to the unit circle. On the other hand, we can notice that the algorithm converges faster for η_{opt} than for $\eta_{\text{sub-opt}}$, as in the linear regression. Although η_{max} seems to make the algorithm converge faster than η_{opt}, it is better to use this last value of the learning rate, as it guarantees a smooth and relatively fast convergence to the minimum.

It is important to notice from figure 4.5 and figure 4.7 that for the linear regression the values of the learning rates used remained constant. This happens because the matrix used to define each value of η never changes, and, as a consequence, the eigen-values of this matrix never change either. On the contrary, the values of the learning rate for the logistic regression do change over time, increasing on each epoch. The reason why this happens is that, when reaching the minima, the derivatives of the activation function tend to decrease, making the matrix to change, and increasing the value of the learning rate, i.e. the learning rate has the capability of taking bigger steps, as the derivative is smaller.

For the simulations using multinomial logistic regression, we can observe, from figure 4.9 that the fastest convergence rate is presented for the value $\eta_{\text{sub-opt}}$. However, it is important to notice that we used an upper-bound for $\lambda_{\max}$ in this case. Because of this, η_{opt} might be far from the optimal value as computed in (3.13). Also, we expect the algorithm to converge for all the values of η under these conditions. It is important to notice that the simulations presented here are performed using the normal mini-batch gradient descent, and no momentum or other strategies used for classic optimizers have been used. The purpose of this simulation is to demonstrate that the algorithm presented has a reasonably good performance using mini-batch gradient descent in a real dataset. In the figure 4.9 we can see confidence intervals for 10 repetitions, and we can observe that these intervals are small, indicating that our algorithm is robust.

In this and the previous chapter we have presented the results for different activation functions: linear, logistic and softmax. However, for other activation functions such as ReLu, Leaky ReLu, hyperbolic tangent, and elu the performance is comparable to the ones presented here. In order to replace the activation function only the derivatives and the output for the functions is needed.

Despite the fact that the algorithm presented showed good results, it has some limitations. In the first place, we have that for non linear activation functions, the proposed valued of η_{opt} and $\eta_{\max}$ are just approximations to the real optimal value and the real upper-bound for the learning rate. This happens because the results were obtained by doing a linear approximation of the algorithm. However, if we always use η_{opt}, the algorithm is guaranteed to converge, and it does it relatively fast in the simulations presented. Secondly, in order to compute the values for η_{opt} and $\eta_{\max}$, it is necessary to compute the largest and the smallest eigenvalues of the matrix $\mathbf{X}^{\mathrm{T}}\mathbf{D}_f\mathbf{X}$, which is a $d \times d$ eigenvalue problem. Therefore, for large dimensions, it might represent a considerably computer load, making difficult to apply the algorithm proposed.

Finally, the simulations and results presented in this thesis are just for linear, logistic and multinomial regression, we have yet to extend this algorithm for the training of neural networks, and combine it with other optimization strategies.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we presented an algorithm that automatically determines an upper-bound and an optimal value for the learning rate of classification and multinomial regression models, such that it guarantees that the gradient descent algorithms converges smoothly and fast to the optimal minima. These values were obtained by analyzing the gradient descent algorithm as a discrete dynamical system. First, we found these values for a logistic regression problem, extending the results found in [1] for a linear regression problem. After that, the upper-bound an the optimal value for the learning rate were generalized to other activation functions, using models of regression and classification. The upper-bound found for this parameter is shown in equation (3.10), and the optimal value in equation (3.13). These results were also extended from the logistic regression model to a multinomial regression model.

The simulations and experiments done in the thesis show that the proposed algorithm produces good results. It also shows that, contrary to what is done in practice, the learning rate should be increased when reaching the minima. The algorithm presented works well for linear, logistic and multinomial regression problems.

6.2 Future Work

As future work, we plan first to extend this algorithm for the training of neural networks, combining it with other strategies such as ADAM, SGD with momentum, among others. We want also to develop a new optimizer, based on the algorithm presented. In this way, the results presented in this thesis and the extension to the optimizer can reduce the training time for deep learning models. It is also necessary to compare of the algorithm proposed with other optimizers, in order to check if the results obtained are functional and correct.

Bibliography

[1] J. C. Ruiz, *Una propuesta de neurona artificial: la Unidad Neuro Vascular Artificial (UNVA)*. 2021.

[2] J Stuart *et al.*, *Artificial intelligence a modern approach third edition*, 2010.

[3] T Mitchell, B Buchanan, G DeJong, T Dietterich, P Rosenbloom, and A Waibel, "Machine learning," *Annual Review of Computer Science*, vol. 4, no. 1, pp. 417–433, 1990. DOI: 10.1146/annurev.cs.04.060190.002221. eprint: https://doi.org/10.1146/annurev.cs.04.060190.002221. [Online]. Available: https://doi.org/10.1146/annurev.cs.04.060190.002221.

[4] T. M. Mitchell and T. M. Mitchell, *Machine learning*. McGraw-hill New York, 1997, vol. 1.

[5] Z.-H. Zhou, *Machine learning*. Springer Nature, 2021.

[6] *Machine learning*, 2022. [Online]. Available: https://en.wikipedia.org/wiki/Machine_learning#cite_note-2.

[7] G. Bonaccorso, *Machine learning algorithms*. Packt Publishing Ltd, 2017.

[8] *What are machine learning models?* 2022. [Online]. Available: https://www.databricks.com/glossary/machine-learning-models.

[9] J. Point, *Machine learning models - javatpoint*, 2022. [Online]. Available: https://www.javatpoint.com/machine-learning-models.

[10] N. Donges, *Gradient descent in machine learning: A basic introduction*, 2022. [Online]. Available: https://builtin.com/data-science/gradient-descent.

[11] S. Patrikar, *Batch, mini batch amp; stochastic gradient descent*, 2019. [Online]. Available: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a.

[12] J. Zhang, "Gradient descent based optimization algorithms for deep learning models training," *arXiv preprint arXiv:1903.03614*, 2019.

[13] M. D. Zeiler, "Adadelta: An adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[14] A. Lydia and S. Francis, "Adagrad—an optimizer for stochastic gradient descent," *Int. J. Inf. Comput. Sci*, vol. 6, no. 5, pp. 566–568,

[15] F. Zou, L. Shen, Z. Jie, W. Zhang, and W. Liu, "A sufficient condition for convergences of adam and rmsprop," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[16] P. Baldi, "Gradient descent learning algorithm overview: A general dynamical systems perspective," *IEEE Transactions on Neural Networks*, vol. 6, no. 1, pp. 182–195, 1995. DOI: 10.1109/72.363438.

[17] K. P. Murphy, *Machine learning: a probabilistic perspective*. MIT press, 2012.

[18] Seldon, *Machine learning regression explained*, 2021. [Online]. Available: https://www.seldon.io/machine-learning-regression-explained#:~:text=Regression%20is%20a%20technique%20for,used%20to%20predict%20continuous%20outcomes..

[19] J. Brownlee, *Linear regression for machine learning*, 2020. [Online]. Available: https://machinelearningmastery.com/linear-regression-for-machine-learning/.

[20] J. Point, *Logistic regression in machine learning - javatpoint*, 2022. [Online]. Available: https://www.javatpoint.com/logistic-regression-in-machine-learning.

[21] *Support vector machines speed pattern recognition - vision systems design*, 2004. [Online]. Available: https://www.vision-systems.com/home/article/16737424/support-vector-machines-speed-pattern-recognition.

[22] J. Platt, "Using analytic qp and sparseness to speed training of support vector machines," *Advances in neural information processing systems*, vol. 11, 1998.

[23] *Iris data set*. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/iris.

[24] P. Tarazaga, "Eigenvalue estimates for symmetric matrices," *Linear algebra and its applications*, vol. 135, pp. 171–179, 1990.