Universidade de Aveiro Departamento de Matemática
2015

Sofia da Silva Fernandes

Classificação Automática do Estado de Células Microglia Usando Stacked Denoising Auto-encoders

Automatic Classification of Microglial Cells' State Using Stacked Denoising Auto-encoders

Sofia da Silva Fernandes

Classificação Automática do Estado de Células Microglia Usando Stacked Denoising Auto-encoders

Automatic Classification of Microglial Cells' State Using Stacked Denoising Auto-encoders

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Matemática e Aplicações, realizada sob a orientação científica do Doutor Luís Silva (orientador), Professor auxiliar convidado do Departamento de Matemática da Universidade de Aveiro, e do Doutor Ricardo Sousa (coorientador), Investigador de Pós-Doutoramento do Instituto Nacional de Engenharia Biomédica.

Esta dissertação foi financiada pelo Fundo Europeu de Desenvolvimento Regional – FEDER – através do Programa Operacional Fatores de Competitividade – COMPETE – e por fundos nacionais através da Fundação para a Ciência e Tecnologia – FCT – no âmbito do projeto PTDC/EIA-EIA/119004/2010.

o júri / the jury

presidente / president

Doutor Pedro Filipe Pessoa Macedo
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Doutor Jorge Manuel Fernandes dos Santos
Professor Adjunto do Instituto Superior de Engenharia do Porto, do Instituto Politécnico do Porto

Doutor Luís Miguel Almeida da Silva
Professor Auxiliar Convidado da Universidade de Aveiro

agradecimentos / acknowledgements

Em primeiro lugar agradeço ao meu orientador, Professor Luís Silva, tanto pela orientação como pela disponibilidade e apoio que me facultou ao longo da realização desta dissertação.

Da mesma forma, agradeço ao meu coorientador, Doutor Ricardo Sousa, pelo suporte que me prestou, pelas sugestões e pela constante disponibilidade.

Agradeço ainda ao Doutor Renato Socodato pela ajuda prestada no âmbito da contextualização da componente biológica do problema que tratei.

Por último, agradeço aos meus pais, irmão e amigos pelo apoio, encorajamento e paciência que demonstraram ao longo desta etapa.

Resumo

Enquanto classe de células constituinte do Sistema Nervoso Central, a microglia é responsável pela sua manutenção e defesa imunológica. Uma célula desta classe pode ser encontrada em três estados distintos (repouso, transição e ativo) sendo que o estado reflete o que está a ocorrer no Sistema Nervoso Central; em particular, pode indiciar o início do desenvolvimento de uma doença neurodegenerativa.

Nesta dissertação, apresentamos o primeiro estudo para o reconhecimento automático do estado de células microglia utilizando *stacked denoising auto-encoders*. Para obter o modelo de reconhecimento mais adequado, recorremos a diferentes estratégias, nomeadamente, ao pré-processamento de imagem, ao aumento artificial do conjunto de dados (usando rotações das imagens) e à resolução de sub problemas do problema original. Aplicamos também transferência de aprendizagem considerando cinco problemas fonte.

Os resultados obtidos mostram que o estado de transição é o mais difícil de reconhecer. Em termos de taxa de acertos, um desempenho de aproximadamente 64% é obtido.

Abstract

As a class of cells composing the Central Nervous System, microglia is responsible for its maintenance and immunological defense. A cell of such class may be found in three distinct states (resting, transition and active) and the state reflects what is occurring on the Central Nervous System; particularly, it may indicate the beginning of the development of a neurodegenerative disease.

In this dissertation, we present the first study for the automatic recognition of microglial cells' state using stacked denoising auto-encoders. In order to obtain the most appropriate recognition model, we resort to different strategies, namely, to image pre-processing, to artificial increase of the dataset (using image rotations) and to solving sub problems of the original problem. We also apply transfer learning considering five source problems.

The obtained results show that the transition state is the most hard to recognize. In terms of accuracy, a performance of approximately 64% is achieved.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Microglia

Introduced by Rio-Hortega [45], the concept of *microglia* refers to a class of cells of the Central Nervous System (CNS) responsible for its maintenance and immunological defense. A *microgliocyte* or a *microglial cell*, which is a specimen of microglia, can be found in three different states: resting, transition and active. In a healthy CNS, the microgliocytes are in a resting state. Unlike the term "resting" may suggest, a resting microgliocyte is responsible for scanning its surrounding environment in order to detect threats to the CNS. After the detection of a threat, such as an infection, the microglial cells undergo a phase of transition leading to its activation. When active, the microgliocytes migrate to the site of the injury, being in charge of removing its causing agents.

The transformation process from rest to active is associated not only to functional changes but also to morphological modifications: while in a resting state, microgliocytes exhibit a ramified shape and, as the cells transit to an active stage, the ramifications are retracted and the cellular body becomes larger, acquiring a round appearance that favors its migration to the local of the injury (see Figure 1.1).

Thus, the identification of the microglial cells' state by an expert may be driven by the observation of its morphology. The classification of a microglial cell's state via morphological inspection has been empirically assessed in a two-step procedure. In the first phase, the classification problem is reduced to a two-class problem: if a ramified morphology is predominant, the expert assumes that the cell is either in the resting state or in transition; on the other hand, a round shape indicates that the cell is either active or in transition. Finally, if the cell exhibits an intermediate morphology (featuring some ramifications and a large cellular body), the microgliocyte is classified as in transition; otherwise, it is assumed that the cell is resting/active. It should be noted that the existence of a clear boundary between the resting and the transition states, as well as between the transition and the active states, is yet unknown, whereby the classification by an expert using the study of the morphology may be misleading.

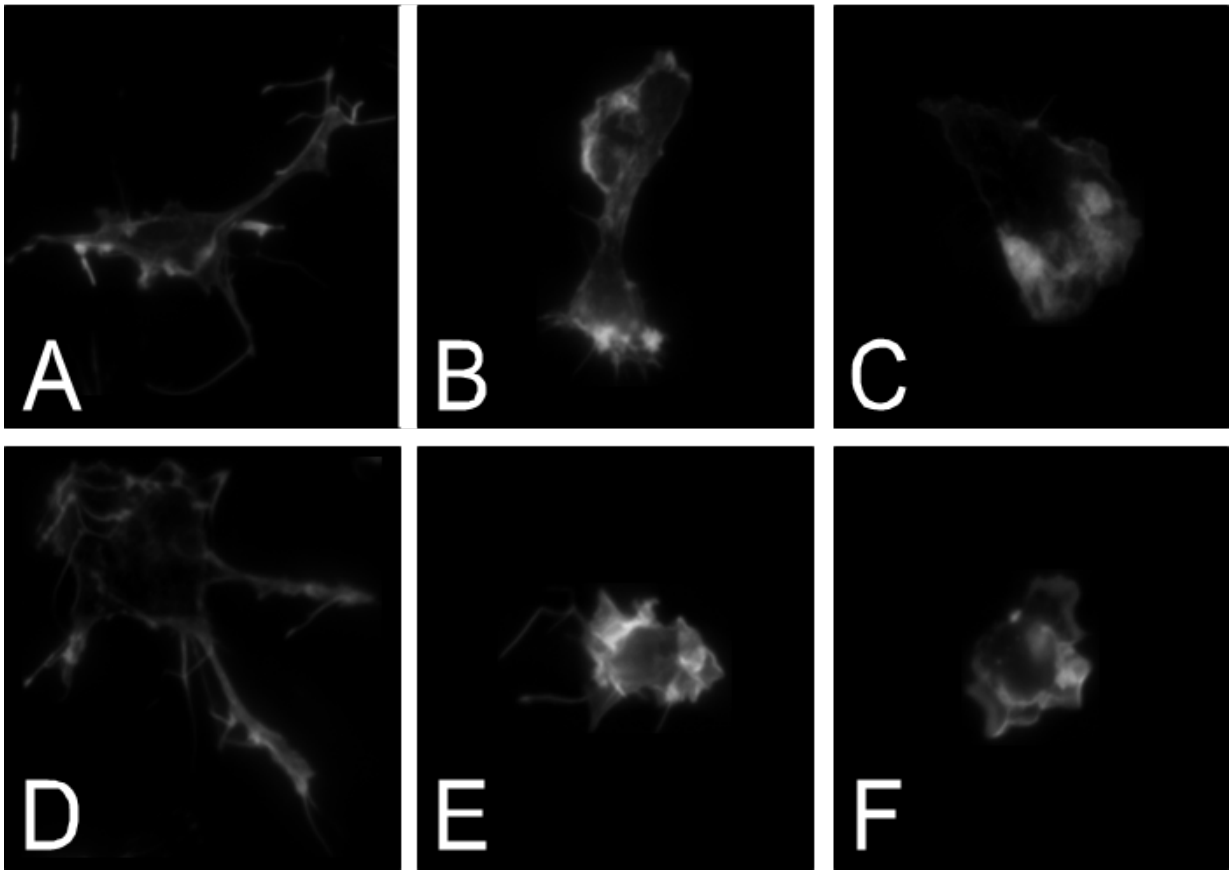

Figure 1.1: Microglial cells in distinct states. A, D: Resting microglial cell; B,E: Transition microglial cell; C, F: Active microglial cell.

1.2 Pattern Recognition

Creating a machine capable of thinking like a human being is one of the main goals of artificial intelligence. The behavior and structure of the human brain have been studied over the years and guided some of the research on this field.

As a subarea of artificial intelligence, machine learning focus on building algorithms able to learn and extract information from data. The purpose of these algorithms is to generalize the underlying behaviors of the available data. In particular, machine learning covers the branch of *pattern recognition*. Pattern recognition focus on classification tasks such as handwritten recognition [33], computer-aided diagnosis (often applied to reduce false negative results [39, 29]) and computer vision (commonly employed on the measurement of products' quality in the industry [17] and on biometrics [54]).

Depending on whether the label of the samples is used or not to build the model, the classification algorithms can be grouped in two categories: if the algorithm uses the labels of the data, it is said to be a case of supervised learning; otherwise, it is said to be a case of unsupervised learning. An example of unsupervised learning is clustering whose

goal is to group identical samples using some similarity measure. Through this work, only supervised learning will be considered.

An example of a classification problem is the diagnosis of a disease. Let $X = \{\mathbf{x}_i\}_{i\in\{1,\dots,n\}}$ be a set of collected data where each $\mathbf{x}_i$ is a vector of features evaluated to determine whether the patient i is a sufferer or a non-sufferer of a certain disease, $diseaseW$, and let $T = \{t_i\}_{i\in\{1,\dots,n\}}$ be a set of label codes for which each t_i is 1 or 0 depending on whether the patient i is a sufferer or a non-sufferer of $diseaseW$, respectively. The model, y, able to predict with some error if a new patient is a sufferer or a non-sufferer of $diseaseW$, is accomplished by a phase of training: a subset of the samples in X, X_{train}, is used as input for y and the difference between the obtained output, $y(\mathbf{x}_i)$, $\mathbf{x}_i \in X_{train}$, and the expected output, t_i, is somehow measured by a cost function; the trainable parameters of the model, which depend on the algorithm used, are so modified in order to minimize the cost function. The performance of the created model is measured by applying y to the remaining samples (not contained in X_{train}) and comparing the obtained output with the respective label.

The set $X_{train} \subset X$, used for training, is called *training set* and the set of remaining samples, used for testing, is called *test set* and is given by $X_{test} = X \backslash X_{train}$. Each sample $\mathbf{x} \in X$ is described as a tuple $\mathbf{x} = (x_1, x_2, \dots, x_d)$ of d *attributes* (which may be continuous or not); the set $S \supset X$ will denote the attribute space. The label t_i for each sample $\mathbf{x}_i$ represents the class to which the sample belongs and it is referred as *target*; it should be noted that the $diseaseW$ example above is a two-class problem (where 0 is the codification associated to a non-sufferer patient and 1 to a sufferer patient). When the problem has cl classes ($cl > 2$), each class is usually encoded in a cl-vector, called target vector, where the c^{th} component of the vector represents the c^{th} class. For example, for a 10-class problem, the target vector for class 3 is a 10-vector with a 1 in the 3^{rd} component and zeros otherwise. This type of target codification will be denoted as $1\text{-}of\text{-}cl$ codification and the set of all possible encoded classes will be denoted by C.

Let $\boldsymbol{\theta}$ represent a vector of all the trainable parameters of the model. The model y, which works as a classifier (a class predictor), may be described as a function $y : S \to C$ that maps a sample $\mathbf{x} \in S$ into a class. The construction of the model y goes through two stages: the training phase, where the model learns the classification task using the training samples, and the test phase, where the performance of the classifier y, previously trained, is assessed. In the case that the model has more than one output component, as it occurs when considering a classification problem of more than 2 classes and the $1\text{-}of\text{-}cl$ target codification, the model is denoted by $\mathbf{y}$ (instead of y).

A possible approach for classification is artificial neural networks. As the name suggests, they simulate, in a way, the behavior of the brain; they are formed by layers of artificial neurons which interact with each other. Some of the first attempts in this area were the perceptron, proposed by Rosenblatt in 1958 [47], and the ADALINE, proposed by Widrow and Hoff in 1960 [53].

In the 1980's, the backpropagation algorithm was developed to train networks with multiple layers. However, when applied to neural networks with a large number of layers, referred as deep neural networks, this algorithm showed some limitations, leading

researchers to use other approaches such as support vector machines (SVMs) [15].

In 2006, deep learning showed a significant advance: the idea explored was to locally train the intermediate layers of neural networks using an unsupervised learning greedy approach such as the Restricted Boltzmann Machines [24] and the auto-encoders [9]. Following the use of auto-encoders, stacked denoising auto-encoders [51], which refer to a neural network which is trained layer-wise applying denoising auto-encoders, emerged. The purpose of the greedy layer-wise training is to obtain a better initialization of the trainable parameters. Such initialization is then considered when fully training the deep neural network.

Another approach, similar to the greedy layer-wise training in the sense that the random initialization is replaced by a more suitable initialization, consists in transfer knowledge from an already trained deep neural network, associated with a (related) problem, to a new deep neural network, associated to the original problem. The idea is to reuse the hidden layers of the already trained deep neural network as initial hidden layers of the original deep neural network in the process of training. This procedure is a type of *transfer learning* [41].

1.3 Problem Statement and Methodology

1.3.1 Problem Motivation

By identifying the state of microglial cells, it is possible to understand what is occurring in the CNS. Particularly, as sustained activation and overactivation may have a detrimental effect by contributing to the loss of neuronal cells [49], the identification of the transition states allows the diagnosis of such activation states: subtle shape changes may anticipate microglia activation. Thus, the identification of such states allows an early diagnosis of neurodegenerative diseases, enabling an early (and, consequently, more efficient) medical intervention.

To the best of our knowledge, no automatic model was built for the identification of a microglial cell's state via its morphology. Thereby, this process always required the intervention of one or more experts. This limitation makes the process slow and highly dependent on the experts.

1.3.2 Proposed Solution

To build a model for the identification of microglial cells' state, we propose the application of stacked denoising auto-encoders with the expectation that the automation of this process may work as an ancillary tool for the experts.

The power of deep neural networks in well known problems such as handwritten recognition has already been shown [9, 33], namely when using stacked denoising auto-encoders [51]. Thus, with the application of a deep neural network to this classification problem, we expect to obtain a model with a small (test) error.

To achieve the best model, several models are defined by varying the training parameters along with the training sets and the network parameters. Moreover, with the aim of improving the performance of the models, we also consider the application of transfer learning, using distinct datasets to build the deep neural networks which are used as sources of knowledge.

The experimental simulations are performed using a Python library called Theano and executed on a Graphics Processing Unity.

1.4 Further Organization of this Thesis

In Chapter 2, we describe deep feedforward neural networks and present its particularities. We also briefly expose the process of transfer learning in the framework of deep feedforward neural networks. In Chapter 3, the experimental procedures are described and the respective results are presented and discussed. The experiments associated only to stacked denoising auto-encoders are treated first and separately from the ones involving transfer learning. Finally, in Chapter 4, we present the conclusions related to all the work, exposing its major limitations.

Chapter 2

Deep Feedforward Neural Networks

2.1 The Perceptron

To understand how deep neural networks work, let us start by understanding the mathematical formulation of an artificial neuron, in particular of a perceptron.

The Rosenblatt perceptron [47] is characterized by a real-valued vector of inputs, $\mathbf{x} = (x_1, \ldots, x_d)$, a real-valued weight vector $\mathbf{w} = (w_1, \ldots, w_d)$, a bias, b, an adder, $\sum$, an activation function, φ, and an output, y (see Figure 2.1).

Given an input $\mathbf{x}$, the adder $\sum$ implements the weighted sum of the inputs along with the bias, $\sum_{i=1}^{d} w_i x_i + b$; the activation function φ takes as input the previous sum and the output is $y = \varphi(\sum_{i=1}^{d} w_i x_i + b)$.

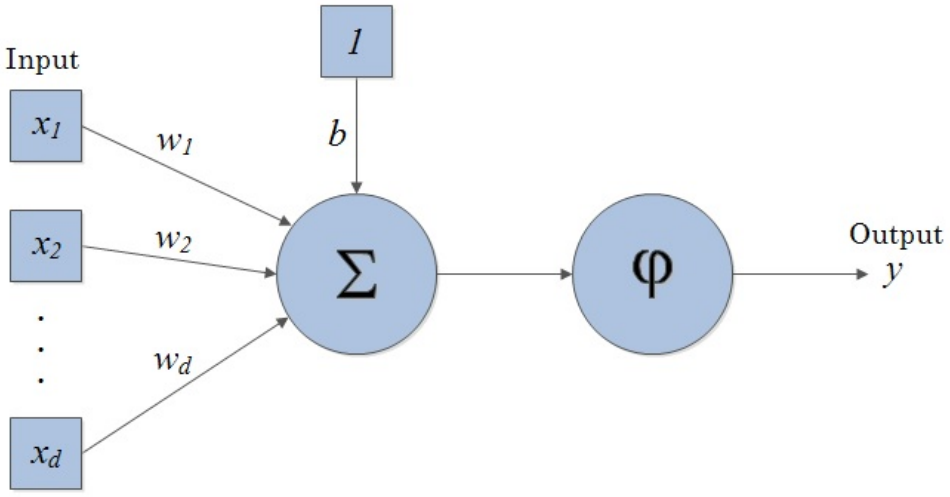

Figure 2.1: Graph representation of a perceptron.

The output values depend on the activation function used; in a perceptron the activation

function is the following step function:

$$\varphi(a) = \begin{cases} 1, \text{if } a \geq 0 \\ -1, \text{if } a < 0 \end{cases}, \tag{2.1}$$

so the output can be -1 in case $\sum_{j=1}^{d} w_j x_j + b < 0$, or 1 otherwise. The symmetric of the bias, $-b$, works as a threshold: the weighted sum, $\sum_{j=1}^{d} w_j x_j$, must surpass $-b$ so that the output is 1. Each weight w_j determines the influence of x_j on the output.

Given that the activation function divides the plane $\mathbb{R}^d$ in two regions: $R_1 = \{\mathbf{x} \in \mathbb{R}^d : \varphi(\sum_{j=1}^{d} w_j x_j + b) = 1\}$ and $R_{-1} = \{\mathbf{x} \in \mathbb{R}^d : \varphi(\sum_{j=1}^{d} w_j x_j + b) = -1\}$ with the decision border being the hyperplane $\sum_{j=1}^{d} w_j x_j + b = 0$, the perceptron should only be applied on linearly separable problems, that is, on problems for which it is possible to find $\mathbf{w}$ and b such that $\sum_{j=1}^{d} w_j x_j > b, \forall \mathbf{x} \in C_1$ and $\sum_{j=1}^{d} w_j x_j < b, \forall \mathbf{x} \in C_2$, where C_1, C_2 represent the classes of the problem; otherwise the convergence is not guaranteed (the convergence proof considering linearly separable classes is commonly found in literature, namely, in [6]).

Furthermore, since the activation function Eq. (2.1) is not continuous, a small change in the weights or bias can cause a large impact on the output; note that the output possible values are -1 and 1 therefore such perturbations on $\mathbf{w}$ or b can change the output from 1 to -1 and vice versa. Instead of the step function, a differentiable function with similar behavior may be considered as activation function; the most commonly used are presented in the following table (Table 2.1).

Function	Definition	Range
Linear (Identity)	$\varphi(a) = a$	$]-\infty, +\infty[$
Logistic Sigmoid [1]	$\varphi(a) = \frac{1}{1+e^{-\alpha a}}, \alpha > 0$	$]0, 1[$
Hyperbolic Tangent	$\varphi(a) = \beta \tanh(\alpha a), \alpha, \beta > 0$	$]-\beta, \beta[$

Table 2.1: Examples of differentiable activation functions.

The generalization of the perceptron in which a differentiable activation function is used will be denoted by unit, artificial neuron or simply neuron. The weight vector $\hat{\mathbf{w}}$ will denote $(w_0, w_1, \ldots, w_d)$, where w_0 is the bias b, and the extended input, $\hat{\mathbf{x}}$, will include an extra entry, $x_0 = 1$, which will be associated to the bias, $\hat{\mathbf{x}} = (x_0, x_1, \ldots, x_d)$, so that $\sum_{j=1}^{d} w_j x_j + b = \sum_{j=1}^{d} w_j x_j + w_0 = \sum_{j=0}^{d} w_j x_j = \hat{\mathbf{w}}^T \hat{\mathbf{x}}$.

Using differentiable activation functions instead of the step function Eq. (2.1) makes the neuron more stable, that is, a small change in the weights or bias has a smaller impact on the output [40].

[1] In this work we will consider $\alpha = 1$.

2.1.1 Perceptron Training

Suppose we want to create a perceptron to solve a problem of distinguishing two linearly separable classes using just a small set of labeled samples, $X \subset \mathbb{R}^d$. How do we find the weights that define a good decision hyperplane? What characterizes a good decision hyperplane?

For the latter question, since the available data is just a small subset of all the possibilities, the quality of a decision hyperplane cannot be exactly measured (except if the behavior of each class is known); however, the samples of X can give an idea if the decision hyperplane defines a good separation of the classes.

The weights and bias adjusted to the problem we are trying to solve are thus obtained with a phase of training. In 1958, Rosenblatt proposed the following algorithm (Algorithm 1):

$it = 0$;
Randomly initialize the weights (with values close to 0), $\hat{\mathbf{w}}^{(it)}$;
while *not all training samples are correctly classified* **do**
  for *each* $\mathbf{x} \in X_{train}$ **do**
    Compute $y = \varphi(\hat{\mathbf{w}}^T \hat{\mathbf{x}})$;
    Update the weights: $\hat{\mathbf{w}}^{(it+1)} = \hat{\mathbf{w}}^{(it)} + \eta(t - y)\hat{\mathbf{x}}$;
    it++;
  end
end

Algorithm 1: The perceptron learning rule.

The update rule:

$$\overbrace{\begin{bmatrix} w_0^{(it+1)} \\ w_1^{(it+1)} \\ \vdots \\ w_d^{(it+1)} \end{bmatrix}}^{\hat{\mathbf{w}}^{(it+1)}} = \overbrace{\begin{bmatrix} w_0^{(it)} + \eta(t-y)x_0 \\ w_1^{(it)} + \eta(t-y)x_1 \\ \vdots \\ w_d^{(it)} + \eta(t-y)x_d \end{bmatrix}}^{\hat{\mathbf{w}}^{(it)} + \eta(t-y)\hat{\mathbf{x}}} = \begin{bmatrix} b^{(it)} + \eta(t-y) \\ w_1^{(it)} + \eta(t-y)x_1 \\ \vdots \\ w_d^{(it)} + \eta(t-y)x_d \end{bmatrix} \tag{2.2}$$

is called *perceptron learning rule* (η, called *learning rate*, is used to control the magnitude of each update and its value may be adjusted in each iteration). Note that the update is only done when a sample is misclassified (otherwise $t - y = 0$). It should be remembered that if the classes are not linearly separable, a different stopping criteria, such as a training error tolerance, should be applied.

Another approach, used when the activation function is differentiable, is to apply the gradient descent method to minimize a *cost function* $E(\boldsymbol{\theta}) = E(\hat{\mathbf{w}})$ which is a function that measures the training error (difference between the output and the target; not to be confused, in this context, with classification error); the update rule is, thus, defined by the *delta rule*: $\hat{\mathbf{w}}^{(it+1)} = \hat{\mathbf{w}}^{(it)} + \Delta\hat{\mathbf{w}}^{(it)}$ where $\Delta\hat{\mathbf{w}}^{(it)} = -\eta\nabla E(\hat{\mathbf{w}}^{(it)})$.

Furthermore, when training neurons, the most common modes to consider are the batch mode and the online mode. In batch mode the update is done using all the samples of the training set - the cost function is a sum over all the training samples. In online mode, the update is done using just the information of a sample - the cost function is defined for a pair (y, t) where $y = \varphi(\hat{\mathbf{w}}^T \hat{\mathbf{x}})$ is the output obtained for $\hat{\mathbf{x}}$ and t is the target for $\mathbf{x}$.

2.2 Feedforward Neural Networks

Informally, a neural network consists in a set of neurons connected to each other. Visually, it can be seen as a directed graph with the nodes corresponding to the network's inputs and neurons and the edges representing the connections between them: the directed edge from node α to node β indicates that the output of neuron α will be part of the input of neuron β.

A neural network is classified depending on the type of connections and on the way neurons are organized. A neural network which is represented by a directed acyclic graph is known as a feedforward neural network (FFNN). In this type of neural networks, considering all the paths from an input to node α, there is no node β in such paths having an edge emerging from α, that is, there is no previous neuron in the paths, receiving the output of neuron α in its input. The same is valid for neuron α itself: its output cannot be used later as part of its own input.

This type of neural network can be organized by layers in such a way that each neuron within a layer can only receive as part of its input the output of neurons in lower layers, receiving at least an input component from the previous layer. The output neurons define the output layer, the other layers are called hidden layers.

Throughout this work, only FFNNs whose connections are restricted to neurons in consecutive layers will be considered: all of the input components of a neuron in a certain layer are associated to the neurons of the previous layer; in particular, the input components of a neuron will include the output of all the neurons in the previous layer (a FFNN with this structure is said to be *fully-connected*, see Figure 2.2 for an example).

Let us mathematically formulate a FFNN with the characteristics just described, assuming it has L layers: let $\mathbf{h}_k$ denote the output vector of layer k, $0 \leq k \leq L$, where $\mathbf{h}_0 = \mathbf{x}$ (which corresponds to the network's input). If γ_k denotes the number of neurons composing layer k, we have $\mathbf{h}_k = (h_{k,1}, \ldots, h_{k,\gamma_k})$ where $h_{k,l}$ denotes the output of the l^{th} neuron belonging to layer k. Let $\mathbf{W}_k$ be the matrix of weight vectors for which the l^{th} column, $\mathbf{w}_{k,l}$, is the weight vector associated to the input of neuron l of layer k; $w_{k,l,m}$ will denote the m^{th} component of the weight vector $\mathbf{w}_{k,l}$ which corresponds to the weight associated to the output of the m^{th} neuron belonging to layer $k-1$ used as part of the input of neuron l of layer k. Let $\mathbf{b}_k = (b_{k,1}, \ldots, b_{k,\gamma_k})$ be the vector of bias used as part of the input considered for neurons of layer k, where $b_{k,l}$ denotes the bias considered for the input of the l^{th} neuron of layer k.

Additionally, let us define the vector of inputs for all the neurons of layer k:

$$\mathbf{a}_k \equiv \mathbf{W}_k^T \mathbf{h}_{k-1} + \mathbf{b}_k \tag{2.3}$$

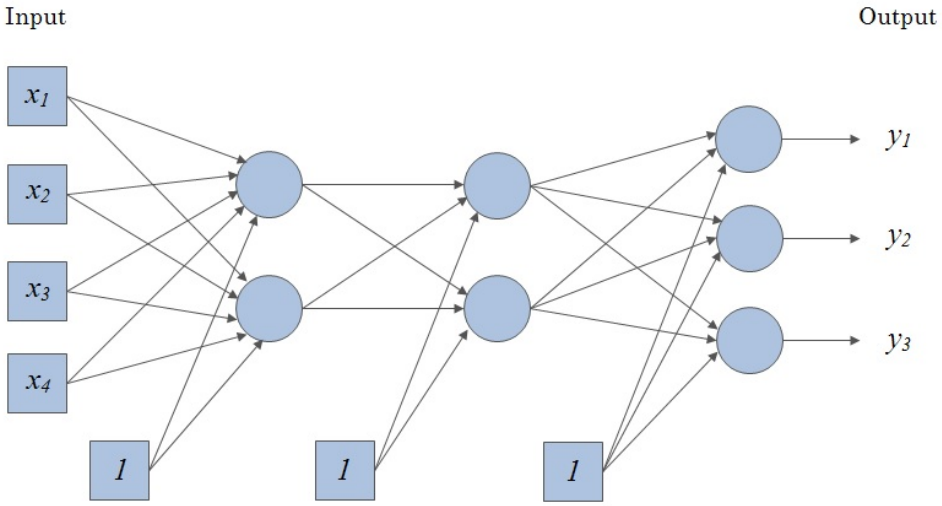

Figure 2.2: Example of a fully-connected FFNN: arrows are used to represent the flow in which the neural network works, each neuron is represented by a circle, the inputs (including the bias) are represented via a square.

---

so that

$$a_{k,l} = \mathbf{w}_{k,l}^T \mathbf{h}_{k-1} + b_{k,l} \tag{2.4}$$

will denote the input of the l^{th} neuron in layer k.

Finally, let φ_k denote the activation function applied to neurons of the k^{th} layer (it should be noted that the activation function used in each neuron may differ, nevertheless, we will only consider FFNNs for which the activation function varies, eventually, between neurons of distinct layers). Besides the activation functions shown in Table 2.1, the *softmax function* is also usually applied at the output layer. For the l^{th} neuron in layer L, the softmax function is given by:

$$\frac{e^{a_{L,l}}}{\sum_{s=1}^{\gamma_L} e^{a_{L,s}}} \ , \tag{2.5}$$

where the function $f(a) = e^a$ denotes the (natural) exponential function and $\gamma_L = cl$ (in case that $\gamma_L > 1$, by assuming the 1-of-cl target codification).

With this activation function the sum of all the outputs is 1 and the output components may have a probabilistic interpretation, in which the value of the l^{th} output component represents a probability estimate that an input belongs to class l [11].

The output of layer k, $1 \leq k \leq L$, is obtained by:

$$\begin{aligned} \mathbf{h}_k &= \varphi_k(\mathbf{W}_k^T \mathbf{h}_{k-1} + \mathbf{b}_k) \\ &= \left(\varphi_k(\mathbf{w}_{k,1}^T \mathbf{h}_{k-1} + b_{k,1}), \ldots, \varphi_k(\mathbf{w}_{k,\gamma_k}^T \mathbf{h}_{k-1} + b_{k,\gamma_k})\right) \end{aligned} \ . \tag{2.6}$$

The final output, $\mathbf{y} = (y_1, \ldots, y_{\gamma_L})$, is thus obtained by the composition of functions of the type of Eq. (2.6) resultant of the propagation of the signal through the associated

neurons within the network: $\mathbf{y} = \mathbf{h}_L$. In the case that the FFNN is being addressed to a classification problem, then its outputs should be mapped into classes. By assuming we are using the 1-*of*-*cl* target codification and a non-linear output activation function, the class is assigned according to the index $l \in \{1, \ldots, \gamma_L\}$ for which the component y_l assumes the largest value if $\gamma_L > 1$ or to the nearest value of the obtained output, otherwise.

2.2.1 Backpropagation Algorithm

An approach used to train neural networks, similar to the one presented for neurons, is to consider an optimization problem. The idea is to maximize/minimize a function, $E(\boldsymbol{\theta})$, that evaluates the performance of the network. Since a maximization problem may be converted in a minimization problem, we will consider the minimization problem approach.

In 1986, Rumelhart *et al.* [48] suggested the application of the gradient descent method using backpropagation of the error to efficiently compute the partial derivatives of the cost function; the algorithm was, though, originally conceived by Werbos in 1974 [52].

Given the vector of parameters $\boldsymbol{\theta}$, which includes all the weights and biases, the update rule used in the gradient descent method is defined by:

$$\boldsymbol{\theta}^{(it+1)} = \boldsymbol{\theta}^{(it)} - \eta \nabla E(\boldsymbol{\theta}^{(it)}) \ . \tag{2.7}$$

This means that the core of the training phase lies on the gradient computation. Hereupon, how should the components of the gradient for each weight vector be computed?

As previously stated, depending on the training mode, the cost function may comprise information from just one training sample or all the training samples. If the batch mode is used and supposing that the cost function may be decomposed in a sum of the errors for each sample, that is, $E(\boldsymbol{\theta}) = \sum_{i=1}^{n} E_i(\boldsymbol{\theta})$ and, consequently, $\nabla E(\boldsymbol{\theta}) = \sum_{i=1}^{n} \nabla E_i(\boldsymbol{\theta})$, the problem of evaluating the gradient will lie on the evaluation of $\nabla E_i(\boldsymbol{\theta})$ for each $i \in \{1, \ldots, n\}$; for this reason let us simplify notation by denoting $E_i(\boldsymbol{\theta})$ as $E(\boldsymbol{\theta})$.

By applying the chain rule to a component of the gradient we have:

$$\begin{cases} \dfrac{\partial E}{\partial w_{k,l,m}} = \dfrac{\partial E}{\partial a_{k,l}} \dfrac{\partial a_{k,l}}{\partial w_{k,l,m}} \\ \dfrac{\partial E}{\partial b_{k,l}} = \dfrac{\partial E}{\partial a_{k,l}} \dfrac{\partial a_{k,l}}{\partial b_{k,l}} \end{cases} . \tag{2.8}$$

Note that the cost function is a function of the output and therefore is a composition of functions in such way that it can be described as a function of $a_{k,l}$ which is by itself a function of $w_{k,l,m}$ and $b_{k,l}$.

Moreover, by Eq. (2.4),

$$\begin{cases} \dfrac{\partial a_{k,l}}{\partial w_{k,l,m}} = \dfrac{\partial \left(\mathbf{w}_{k,l}^T \mathbf{h}_{k-1} + b_{k,l} \right)}{\partial w_{k,l,m}} = h_{k-1,m} \\ \dfrac{\partial a_{k,l}}{\partial b_{k,l}} = \dfrac{\partial \left(\mathbf{w}_{k,l}^T \mathbf{h}_{k-1} + b_{k,l} \right)}{\partial b_{k,l}} = 1 \end{cases} . \tag{2.9}$$

In addition, let us define:

$$\delta_{k,l} \equiv \frac{\partial E}{\partial a_{k,l}} \ . \tag{2.10}$$

Thus, from Eq. (2.9) and Eq. (2.10), in Eq. (2.8), we have:

$$\begin{cases} \dfrac{\partial E}{\partial w_{k,l,m}} = \delta_{k,l} h_{k-1,m} \\ \dfrac{\partial E}{\partial b_{k,l}} = \delta_{k,l} \end{cases} , \tag{2.11}$$

where

$$\delta_{L,l} = \frac{\partial E}{\partial a_{L,l}} \tag{2.12}$$

is obtained directly. As an example, let us consider the sigmoid activation function on the output layer and the cost function given by:

$$\frac{1}{2}||\mathbf{y}-\mathbf{t}||_2^2 = \frac{1}{2}||(y_1,\ldots,y_{\gamma_L})-(t_1,\ldots,t_{\gamma_L})||_2^2 = \frac{1}{2}\sum_{s=1}^{\gamma_L}(y_s-t_s)^2 \tag{2.13}$$

where $\mathbf{t}$ is the γ_L-vector corresponding to the expected output. For output neurons, we have that

$$\begin{aligned} \delta_{L,l} &= \frac{\partial\left(\frac{1}{2}\sum_{s=1}^{\gamma_L}(y_s-t_s)^2\right)}{\partial a_{L,l}} = \frac{\partial\left(\frac{1}{2}\sum_{s=1}^{\gamma_L}(\varphi_L(a_{L,s})-t_s)^2\right)}{\partial a_{L,l}} \\ &= \frac{\partial\left(\frac{1}{2}(\varphi_L(a_{L,l})-t_l)^2\right)}{\partial a_{L,l}} = (\varphi_L(a_{L,l})-t_l)\varphi_L'(a_{L,l}) \end{aligned} \ ; \tag{2.14}$$

note that the contribution of $a_{L,l}$ to the output is given through $\varphi_L(a_{L,l})$ which is y_l.

Back to the general case, by applying the chain rule, we have

$$\delta_{k,l} = \sum_{s=1}^{\gamma_{k+1}} \frac{\partial E}{\partial a_{k+1,s}} \frac{\partial a_{k+1,s}}{\partial a_{k,l}} \overset{\text{Eq. (2.10)}}{=} \sum_{s=1}^{\gamma_{k+1}} \delta_{k+1,s} \frac{\partial a_{k+1,s}}{\partial a_{k,l}} \ , \tag{2.15}$$

for $k < L$.

According to Eq. (2.4),

$$\begin{aligned} \frac{\partial a_{k+1,s}}{\partial a_{k,l}} &= \frac{\partial\left(\mathbf{w}_{k+1,s}^T \mathbf{h}_k + b_{k+1,s}\right)}{\partial a_{k,l}} = \frac{\partial\left(\sum_{u=1}^{\gamma_k} w_{k+1,s,u}\varphi_k(a_{k,u}) + b_{k+1,s}\right)}{\partial a_{k,l}} \\ &= w_{k+1,s,l}\varphi_k'(a_{k,l}) \end{aligned} \tag{2.16}$$

so that in Eq. (2.15), we obtain the backpropagation formula

$$\delta_{k,l} = \sum_{s=1}^{\gamma_{k+1}} \delta_{k+1,s} w_{k+1,s,l} \varphi_k'(a_{k,l}) = \varphi_k'(a_{k,l}) \sum_{s=1}^{\gamma_{k+1}} \delta_{k+1,s} w_{k+1,s,l} \ . \tag{2.17}$$

(In the case that the softmax function is used, the backpropagation formula assumes a slightly different form: it is necessary to consider $\frac{\partial \varphi_{k,u}}{\partial a_{k,l}}$ in Eq. (2.16)).

Consequently, it is possible, using Eq. (2.11), to evaluate the gradients for each weight vector starting from the highest positioned neurons, in this case, starting from the output layer of the network, and propagating through the lower positioned neurons: after the processing of the input by all the layers, the values $\delta_{L,l}$ can be computed and, by Eq. (2.17), the other δ's may be recursively obtained.

Having stated the backpropagation of the error, the backprogagation algorithm, in its online form, is presented in Algorithm 2.

Initialize the extended weight vectors with small values;
while *the stop criterion is not reached* **do**
  for *each* $\mathbf{x} \in X_{train}$ **do**
    Forward propagation phase:
    Input the sample $\mathbf{x}$ to the network;
    for *each layer* k *from 1 to* L **do**
      Compute $\mathbf{a}_k$ using Eq. (2.3) ;
      Compute $\mathbf{h}_k$ using Eq. (2.6) ;
    end
    Backward propagation phase:
    for *each output neuron* l **do**
      Compute $\delta_{L,l}$ using Eq. (2.12);
      Compute the gradients $\frac{\partial E}{\partial w_{L,l,m}}$ and $\frac{\partial E}{\partial b_{L,l}}$ using Eq. (2.11);
    end
    for *each hidden layer* k *from* $L-1$ *to* 1 **do**
      for *each neuron* l *of layer* k **do**
        Compute $\delta_{k,l}$ using Eq. (2.17);
        Compute the gradients $\frac{\partial E}{\partial w_{k,l,m}}$ and $\frac{\partial E}{\partial b_{k,l}}$ using Eq. (2.11);
      end
    end
    Update all the weights, $w_{k,l,m}$, and biases, $b_{k,l}$, using the delta rule:

$$\begin{cases} w_{k,l,m} = w_{k,l,m} - \eta \dfrac{\partial E}{\partial w_{k,l,m}} \\ b_{k,l} = b_{k,l} - \eta \dfrac{\partial E}{\partial b_{k,l}} \end{cases}$$

  end
end

Algorithm 2: Online Backpropagation Algorithm.

2.2.2 Cost Functions

Different cost functions may be chosen to guide the training phase in a FFNN. The suitability of each cost function depends on the nature of the problem. Since in this work we are considering a classification problem, let us assume the 1-*of*-*cl* target codification if $\gamma_L > 1$ and the binary target codification (0 or 1), otherwise (see Section 1.2).

By considering the online mode, the traditionally used function is the *squared error* function (SE) which is given by

$$\text{SE} = E_{\text{SE}}(\boldsymbol{\theta}) = ||\mathbf{y} - \mathbf{t}||_2^2 = \sum_{s=1}^{\gamma_L}(y_s - t_s)^2 \ . \tag{2.18}$$

As alternative to the SE, the *cross-entropy* function (CE), given by:

$$\text{CE} = E_{\text{CE}}(\boldsymbol{\theta}) = \begin{cases} -\left(t\ln(y) + (1-t)\ln(1-y)\right), \text{ if } \gamma_L = 1 \\ -\sum_{s=1}^{\gamma_L} t_s \ln(y_s), \text{ if } \gamma_L > 1 \end{cases} , \tag{2.19}$$

may be applied by assuming that the output components, y_s, may be interpreted as probabilities estimates, that is, the y_s components are in $(0, 1)$ (and sum to 1 in the case that $\gamma_L > 1$). These cost functions are easily adapted to a batch mode by considering the sum or the mean over the samples.

According to Bishop [11], there is a natural paring between the cost function and the output activation function. He suggests the application of a linear activation function on the output layer combined with the SE cost function, the association of the sigmoid to the $\text{CE}_{\gamma_L=1}$ cost function (which represents the CE function when $\gamma_L = 1$) and the combination of the the softmax function with the $\text{CE}_{\gamma_L>1}$ cost function (which corresponds to the CE function when $\gamma_L > 1$).

In order to understand the motivation inherent to the use of those combinations, let us compute the partial derivatives of the cost functions with respect to the input of the neurons in the output layer, $\delta_{L,l}$.

By considering the SE cost function, we obtain:

$$\begin{aligned} \delta_{L,l} = \frac{\partial \text{ SE}}{\partial a_{L,l}} \overset{\text{Eq. (2.18)}}{=} \frac{\partial\left(\sum_{s=1}^{\gamma_L}(y_s - t_s)^2\right)}{\partial a_{L,l}} = \frac{\partial\left((y_l - t_l)^2\right)}{\partial a_{L,l}} \overset{(*)}{=} 2(y_l - t_l)\frac{\partial y_l}{\partial a_{L,l}} \\ \overset{(**)}{=} 2(\varphi_L(a_{L,l}) - t_l)\frac{\partial \varphi_L(a_{L,l})}{\partial a_{L,l}} = 2(\varphi_L(a_{L,l}) - t_l)\varphi'_L(a_{L,l}) \end{aligned} , \tag{2.20}$$

where $(*)$ denotes the application of the chain rule and $(**)$ consists in the application of the definition of y_l (this notation will be considered for the rest of this section).

Thus, if $\varphi_L(a_{L,l}) = a_{L,l}$, that is, if the output activation function is linear then $\varphi'_L(a_{L,l}) = 1$ and the previous expression, Eq.(2.20), simplifies to

$$\delta_{L,l} = 2(\varphi_L(a_{L,l}) - t_l) \ . \tag{2.21}$$

In the case of the $\text{CE}_{\gamma_L=1}$ cost function, we have that:

$$\begin{aligned}\delta_{L,l} = \frac{\partial\ \text{CE}_{\gamma_L=1}}{\partial a_{L,l}} &\overset{\text{Eq. (2.19)}}{=} -\frac{\partial\left(t\ln(y)+(1-t)\ln(1-y)\right)}{\partial a_{L,l}} \\ &\overset{l\in\{1\}}{=} -\frac{\partial\left(t_l\ln(y_l)+(1-t_l)\ln(1-y_l)\right)}{\partial a_{L,l}} \\ &\overset{(*)}{=} -\frac{t_l}{y_l}\frac{\partial y_l}{\partial a_{L,l}} - \frac{t_l-1}{1-y_l}\frac{\partial y_l}{\partial a_{L,l}} \\ &\overset{(*,**)}{=} \varphi'_L(a_{L,l})\left(\frac{-t_l}{\varphi_L(a_{L,l})} - \frac{t_l-1}{1-\varphi_L(a_{L,l})}\right) \\ &= -\varphi'_L(a_{L,l})\left(\frac{t_l-\varphi_L(a_{L,l})}{\varphi_L(a_{L,l})(1-\varphi_L(a_{L,l}))}\right)\end{aligned} \quad . \tag{2.22}$$

Given that, when considering the sigmoid function, we have:

$$\begin{aligned}\varphi'_L(a_{L,l}) = \frac{e^{-a_{L,l}}}{(1+e^{-a_{L,l}})^2} &= \frac{e^{-a_{L,l}}}{1+e^{-a_{L,l}}}\varphi_L(a_{L,l}) = \left(1-\frac{1}{1+e^{-a_{L,l}}}\right)\varphi_L(a_{L,l}) \\ &= (1-\varphi_L(a_{L,l}))\varphi_L(a_{L,l})\end{aligned} \quad , \tag{2.23}$$

by combining the sigmoid function with the $\text{CE}_{\gamma_L=1}$ cost function, the expression Eq. (2.22) simplifies to:

$$\delta_{L,l} = \varphi_L(a_{L,l}) - t_l \quad . \tag{2.24}$$

Finally let us consider the $\text{CE}_{\gamma_L>1}$ cost function. By denoting the output activation function associated to the l^{th} neuron by $\varphi_{L,l}$, we verify that:

$$\begin{aligned}\frac{\partial\ \text{CE}_{\gamma_L>1}}{\partial a_{L,l}} &\overset{\text{Eq. (2.19)}}{=} -\frac{\partial\left(\sum_{s=1}^{\gamma_L} t_s\ln(y_s)\right)}{\partial a_{L,l}} = -\sum_{s=1}^{\gamma_L}\frac{\partial\left(t_s\ln(y_s)\right)}{\partial a_{L,l}} \\ &\overset{(*)}{=} -\sum_{s=1}^{\gamma_L} t_s\frac{1}{y_s}\frac{\partial y_s}{\partial a_{L,l}} \overset{(**)}{=} \sum_{s=1}^{\gamma_L}\frac{-t_s}{\varphi_{L,s}}\frac{\partial\varphi_{L,s}}{\partial a_{L,l}}\end{aligned} \quad , \tag{2.25}$$

In this case, by considering the softmax as the output activation function, we have that:

$$\frac{\partial\varphi_{L,s}}{\partial a_{L,l}} = \frac{\partial\left(\frac{e^{a_{L,s}}}{\sum_{p=1}^{\gamma_L}e^{a_{L,p}}}\right)}{\partial a_{L,l}} \quad , \tag{2.26}$$

whereby if $s=l$:

$$\begin{aligned}\frac{\partial\varphi_{L,l}}{\partial a_{L,l}} = \frac{e^{a_{L,l}}\sum_{p=1}^{\gamma_L}e^{a_{L,p}} - (e^{a_{L,l}})^2}{\left(\sum_{p=1}^{\gamma_L}e^{a_{L,p}}\right)^2} &= \varphi_{L,l}\left(\frac{\sum_{p=1}^{\gamma_L}e^{a_{L,p}} - e^{a_{L,l}}}{\sum_{p=1}^{\gamma_L}e^{a_{L,p}}}\right) \\ &= \varphi_{L,l}\left(1-\varphi_{L,l}\right)\end{aligned} \quad , \tag{2.27}$$

otherwise (if $s \neq l$),

$$\frac{\partial \varphi_{L,s}}{\partial a_{L,l}} = \frac{-e^{a_{L,s}} e^{a_{L,l}}}{\left(\sum_{p=1}^{\gamma_L} e^{a_{L,p}}\right)^2} = -\varphi_{L,s}\varphi_{L,l} \tag{2.28}$$

and, by substituting $\frac{\partial \varphi_{L,s}}{\partial a_{L,l}}$ in Eq. (2.25), we obtain:

$$\begin{aligned}
\delta_{L,l} &= \frac{-t_l}{\varphi_{L,l}} \varphi_{L,l} \left(1 - \varphi_{L,l}\right) + \sum_{s \in \{1,\ldots,\gamma_L\} \setminus \{l\}} \frac{-t_s}{\varphi_{L,s}} (-\varphi_{L,s}\varphi_{L,l}) \\
&= -t_l \left(1 - \varphi_{L,l}\right) + \sum_{s \in \{1,\ldots,\gamma_L\} \setminus \{l\}} t_s \varphi_{L,l} \\
&= -t_l + \varphi_{L,l} \left(\sum_{s=1}^{\gamma_L} t_s \right) \\
&\overset{(***)}{=} \varphi_{L,l} - t_l
\end{aligned} \tag{2.29}$$

where the step (***) is executed since we are using the 1-of-*cl* target codification.

It is thus possible to verify a common behavior of the partial derivatives when considering the respective activation function: the term $\varphi'_L(a_{L,l})$ (or $\frac{\partial \varphi_{L,p}}{\partial a_{L,l}}$ in the case of the $\text{CE}_{\gamma_L>1}$ cost function with softmax function) is canceled; simplifying the expressions of $\delta_{L,l}$. In particular, the greater the difference $\varphi_L(a_{L,l}) - t_l$, the greater the term $\delta_{L,l}$.

2.2.3 Training Aspects

Training Mode

As mentioned above, if the batch mode is used, the final derivatives will be the sum of the derivatives obtained for each sample. As a consequence, the online mode may be faster, namely if the set of samples is redundant (that is, if there are several identical samples). This is due to the similar impact of the redundant samples on the gradients' computation using batch mode. Furthermore, the variation associated to the online mode due to the use of one sample in each update may cause the algorithm to escape from local minima [11]. Based also, but not only, on this aspects, Wilson and Martinez [55] suggest that the online mode is usually a more appropriate/efficient choice. Alternatively to these modes, a *mini-batch* mode is sometimes used whereupon the samples' set is divided into smaller subsets, mini-batches, so that a mini-batch is considered for each update.

Stopping Criteria and Validation

Some of the stopping criteria usually applied are a tolerance for the training error and the maximum number of *epochs*. By using the training error tolerance, ϵ, the network training is stopped when $E(\boldsymbol{\theta}) < \epsilon$. In the context of neural networks, an epoch occurs

after a total presentation of the samples in the training set; thus, by setting a maximum number of epochs, we limit the training process by establishing the maximum number of times the training samples are presented to the model.

An important aspect of training, common to other models in machine learning, is the generalization ability, by which we mean the capacity to correctly predict the target of an unseen sample (a sample which is not in the training set). If the model does not capture enough significant information about the general behavior of the training samples and, because of that, incorrectly predicts the output of new samples, we say that an *under-fitting* phenomena has occurred. This may be due to the use of a poor training set. On the other hand, it may occur that the trained model is too adjusted to the training samples, capturing eventual existing noise. In this case, it is said that *over-fitting* occurs. The over-fitting problem may be avoided by using an appropriate stopping criteria involving, for example, a validation set; the idea is to divide the training set in two disjoint subsets, a new smaller training set and a validation set. The training set is used to train the model and, in each epoch, the validation set is used to evaluate the error in unseen samples. Theoretically, the network training should be stopped when the error on the validation set starts increasing. In the *K-fold cross-validation*, which is a variation of the *validation set* approach just described, the training set is divided in K disjoint and equally sized subsets (folds); the network is trained K times using, each time, a different fold as a validation set and the other $K-1$ folds as a training set; the final errors (training and validation) are obtained by averaging the K errors previously obtained.

Improved Backpropagation Versions

In what concerns to the performance of the algorithm, although Hinton [22] refers to the slow convergence of the algorithm as the main issue, experimental results [9, 18] suggest that randomly initialized backpropagation gets stuck in poor local minima; particularly, it is suggested that the probability of getting stuck in poor minima increases as the depth (number of layers) increases. Moreover, it should also be noted that the performance of the algorithm may be influenced by the use of a fixed learning rate which can lead the algorithm to get stuck in a local minima: on one hand, a small η increases the time needed to train; a large η, on the other hand, may cause the algorithm to skip the optimal solution [35, 43].

For these reasons improvements were suggested to the backpropagation algorithm [35, 37, 46, 48], namely, the use of a learning rate for each weight/bias, the introduction of momentum and the norm-regularization. In the backpropagation with momentum, the z^{th} update rule for each weight (or bias) θ is modified to:

$$\theta(z) = \theta(z-1) + \overbrace{\left(-\eta\frac{\partial E}{\partial \theta}(z) + \xi\Delta\theta(z-1)\right)}^{\Delta\theta(z)} \tag{2.30}$$

where ξ, known as *momentum rate*, varies between 0 and 1. The introduction of the momentum term $\xi\Delta\theta(z-1)$ allows that, in each update z, the previous changes in θ

influence the current update z; ξ determines the influence of the past updates. Considering this term in the update rule, the algorithm tends to converge more rapidly as it reduces the oscillations.

Another strategy used to improve this training algorithm is the L_p regularization method. This type of regularization introduces a term in the cost function in order to penalize large weights, which are associated to the over-fitting phenomena [11]. The new cost function is then given by

$$E_R(\boldsymbol{\theta}) = E(\boldsymbol{\theta}) + \lambda R(\boldsymbol{\theta}) \ , \tag{2.31}$$

where λ works as a *regularization coefficent* that determines the influence of the penalization term with respect to the original cost function $E(\boldsymbol{\theta})$. When considering the L_p norm regularization, the term $R(\boldsymbol{\theta})$ is defined as $||\boldsymbol{\theta}||_p^p = \sum_{q=1}^{|\boldsymbol{\theta}|} |\theta_q|^p$. In the case that $p = 2$, the regularization is known as *weight decay*.

2.3 Deep Feedforward Neural Networks

According to Cybenko [16], Hornik *et al.* [25] and Castro *et al.* [13], a FFNN with a single hidden layer is, under some assumptions, able to approximate any continuous function provided that a sufficient number of hidden neurons is used. Thereby, why should we use FFNNs with several hidden layers given the power of a FFNN with a single hidden layer?

First, the appropriate number of hidden neurons needed to approximate such functions using a FFNN with a hidden layer was not discussed. Particularly, in the context of boolean functions, Hastad [21] proved the existence of functions whose representation (via a logic circuit) exhibited polynomial-size when considering a logic circuit of depth k, but exhibited exponential size when restricted to a logic circuit of depth $k-1$. Considering the mapping of a k-depth logic-gate circuit as a FFNN of k layers, with the neurons corresponding to logic gates, then every boolean function may be represented by a FFNN. Thus, the result presented by Hastad may indicate that the number of neurons in a FFNN with k layers, needed to reproduce a FFNN originally with $k+1$ layers, is very large, compared with the number of neurons used in the $(k+1)$-layer network.

Finally, a FFNN with a large number of layers, known as deep FFNN (DNN), when compared to their shallow counterparts (which have one or two hidden layers), exhibit the ability to describe an hierarchy of features (data representations): higher level features are built through lower level features allowing an increasing level of abstraction from the lowest level features to the highest within the hierarchy; each level of representation is so associated to a layer [8].

Inspired by the model of the nervous visual system proposed by Hubel and Wiesel [27, 28], Fukushima [19] introduced convolutional neural networks (CNN) in 1980. Afterwards, Le Cun *et al.*[32, 34] presented an improved model which became the first DNN successfully trained applying the backpropagation algorithm. Originally modeled for visual pattern recognition systems, CNNs were developed according to the following key concepts: local

receptive fields, weight sharing and sub-sampling. The processing in this type of neural network switches between feature detection and sub-sampling stages in order to decrease the sensitivity of the network to distortions and shifts.

Except for convolutional neural networks, the backpropagation algorithm was not appropriate for training general DNNs (see Section 2.2.3) and, due to this limitation, the usage of DNNs was not appellative despite its potential.

In 2006 a new approach to train DNNs was proposed, involving a two-phase training process: a *pre-training* phase, in which each hidden layer is trained separately (layer-wise); and a *fine-tuning* phase, in which the full DNN is trained using backprogation. Originally, the layer wise training was performed via Restricted Boltzmann Machines (RBM)[23, 24]. In 2007, Bengio *et al.* [9] suggested the use of auto-encoders and, shortly after, Vincent *et al.*[51] proposed denoising auto-encoders.

Since its inception, DNNs have been applied to several problems, namely, to handwritten recognition [24, 33], face recognition [31], text retrieval (categorization) [44], robotic autonomous offroad navigation [20] and computer aided medical diagnosis [36, 38].

2.3.1 Stacked Denoising Auto-Encoders

Also known as auto-associator [12], an auto-encoder (AE) is a FFNN usually with one hidden layer that reconstructs its own input; for this reason, in this type of FFNN, the output layer must have the same dimension as the input and the target is the input itself. Let us consider an AE with a single hidden layer (see Figure 2.3 for an example). The process of reconstruction is divided in two phases: an encoding phase followed by a decoding phase so that the result of processing the AE hidden layer, $c(\mathbf{x})$, is the encoding of the input $\mathbf{x}$ and the final output, which is the representation $\tilde{\mathbf{x}}$ of $\mathbf{x}$, is obtained by decoding $c(\mathbf{x})$, $\tilde{\mathbf{x}} = d(c(\mathbf{x}))$. Considering the notation for FFNNs, an AE may be described as:

$$\tilde{\mathbf{x}} = \underbrace{\varphi_2(\mathbf{W}_2^T \overbrace{\varphi_1\left(\mathbf{W}_1^T\mathbf{x} + \mathbf{b}_1\right)}^{c(\mathbf{x})} + \mathbf{b}_2)}_{d(c(\mathbf{x}))} \tag{2.32}$$

where $\{\mathbf{W}_1, \mathbf{b}_1\}$ are the trainable parameters associated to the hidden layer, $\{\mathbf{W}_2, \mathbf{b}_2\}$ are the trainable parameters associated to the output layer and $\tilde{\mathbf{x}}$ and $\mathbf{x}$ have the same dimension. In order to simplify the notation in the context of AEs, $\mathbf{W}$ and $\mathbf{b}$ will denote $\mathbf{W}_1$ and $\mathbf{b}_1$, respectively; likewise, $\mathbf{W}'$ and $\mathbf{b}'$ will denote (respectively) $\mathbf{W}_2$ and $\mathbf{b}_2$. In the case that $\mathbf{W}' = \mathbf{W}^T$ it is said that tied weights are being used.

Depending on the type of input, the activation function applied to the neurons of the output layer may be linear or nonlinear. For real valued inputs, the linear activation function should be chosen in order to not restrict the output values, however, if the input components domain is in [0,1], the sigmoid function would be more appropriate. Note that the idea of an AE is to reproduce the input, therefore the output layer range of values should cover the input domain. Similarly, the cost function should be chosen according to the attribute space, S. In the case that $S = \mathbb{R}^d$, the SE cost function may be more

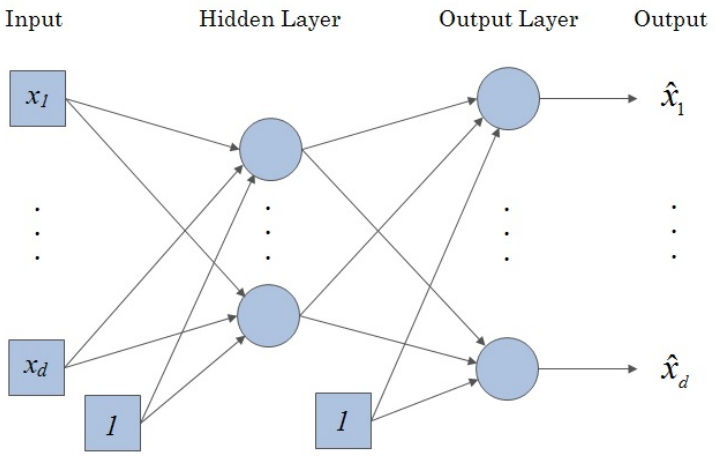

Figure 2.3: Model of an auto-encoder with a hidden layer.

appropriate. If, on the other hand, $S = [0,1]^d$ the common applied cost function is the *element wise* CE [9, 51], which is given by:

$$-\sum_{s=1}^{\gamma_L}(t_s\log(y_s)+(1-t_s)\log(1-y_s)) = -\sum_{s=1}^{d}(x_s\log(y_s)+(1-x_s)\log(1-y_s)) \quad . \tag{2.33}$$

Furthermore, since the usual training method used is the backpropagation algorithm (Section 2.2.1), it is important to prevent the AE from learning the identity function which presents minimal error. A possibility to overcome this issue is to constrain the hidden layer size (number of neurons) to be smaller than the size (number of components) of the input so that the encoding phase corresponds to a dimensionality reduction.

Introduced by Bengio *et al.* [9], the concept of stacked auto-encoders (SAE) is associated to the pre-training of each hidden layer of a deep neural network separately regarding them as AEs; the aim is to obtain an initialization of the weights more appropriate than the random one, which is usually considered in algorithms like backpropagation. In the pre-training phase, each hidden layer of the DNN, starting from the lowest-level layer to the highest-level layer, is considered as the hidden layer of an AE. The input of each AE is obtained by processing the original (network) input through the previous already trained layers of the DNN. Thus, after the first AE is trained, its output layer is discarded and the resultant trained layer is used to obtain the inputs for the next AE, which has, as hidden layer, the second hidden layer of the DNN. The process is successively repeated for the other hidden layers until all the hidden layers of the DNN are trained. Finally, in the fine-tuning phase, the whole DNN is trained using a gradient descent method like back-propagation where the initial weights of the output layer are randomly initialized while the weights of the hidden layers are the ones obtained in the pre-training phase.

Although, as mentioned above, the hidden layer in a AE should have smaller size than the input, according to Bengio *et al.*[9] when applying this model of training, the obtained

SAE performs well even if the size of its hidden layers is not strictly decreasing from the lowest to the highest-level layers.

As an illustrative example of the pre-training process, let us consider the DNN presented in Figure 2.4. In order to simplify the notation, let us denote the matrix of weight vectors associated to the output layer of the k^{th} AE by $\mathbf{W}'_k$ remembering that $\mathbf{W}_k$ is the weight matrix associated to the k^{th} AE hidden layer and, consequently, to the k^{th} layer of the DNN.

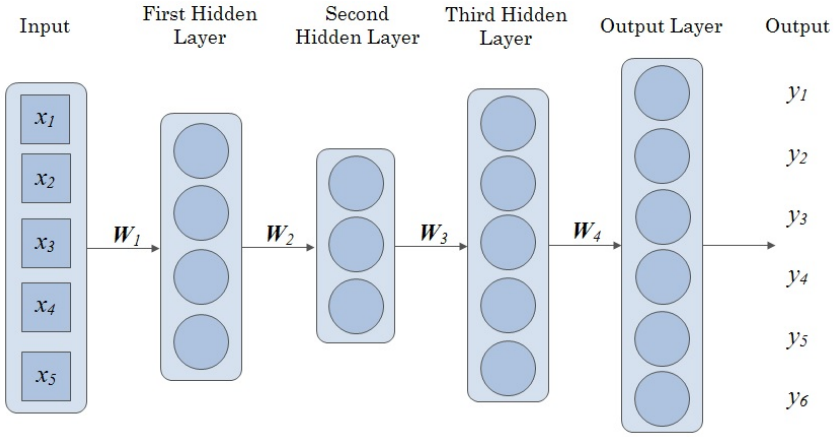

Figure 2.4: Untrained DNN (with the bias omitted).

In the first built AE, the input and the hidden layer are the input and the first hidden layer of the DNN (see Figure 2.4). The AE (as shown in Figure 2.5) is then trained with backpropagation to obtain $\mathbf{W}_1$ and $\mathbf{W}'_1$. Then, the output layer (and consequently $\mathbf{W}'_1$) is discarded and the first hidden layer of the network is used to obtain the input to a new AE, for which the hidden layer is the second hidden layer of the network (see Figure 2.6: the inputs of the AE are represented using dashed squares to distinguish them from the inputs of the network, $x_1, \ldots, x_5$). Analogously, the new AE is trained using the backpropagation algorithm to obtain $\mathbf{W}_2$.

Once the second AE is trained, its output layer is discarded. The last AE has, as hidden layer, the third hidden layer of the DNN and its inputs are the processing result of the original inputs through the first two hidden layers of the DNN (Figure 2.7). In the same way, this AE is trained to obtain $\mathbf{W}_3$.

In the fine-tuning phase the trained weights $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$ are used as initial weights for each hidden layer of the DNN, while the $\mathbf{W}_4$ weights are randomly initialized. Backpropagation is now used in a supervised way to drive the DNN output as closest as possible to the target response.

Alternatively, the use of AEs may be replaced by the use of denoising auto-encoders (dAEs) as proposed by Vincent *et al.* [51]. The training of the dAEs is guided in an effort to reconstruct the original (uncorrupted) input from a noisy version of it; the noise is introduced with the purpose of improving the robustness of the algorithm. According to [51], having defined the portion of input's components to be corrupted, ζ, the process

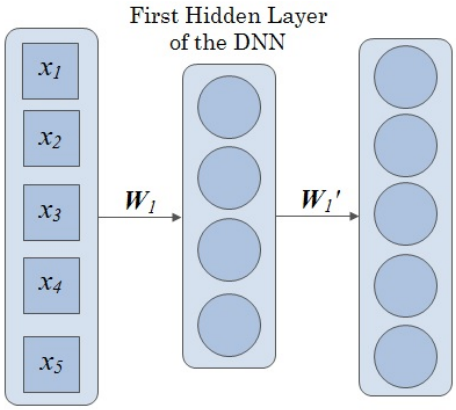

Figure 2.5: AE built to obtain the initialization of the weights of the first hidden layer.

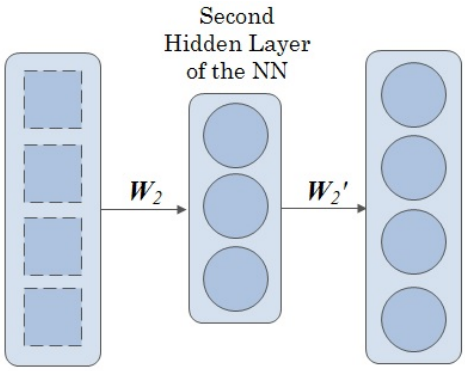

Figure 2.6: AE built to obtain the initialization of the weights of the second hidden layer.

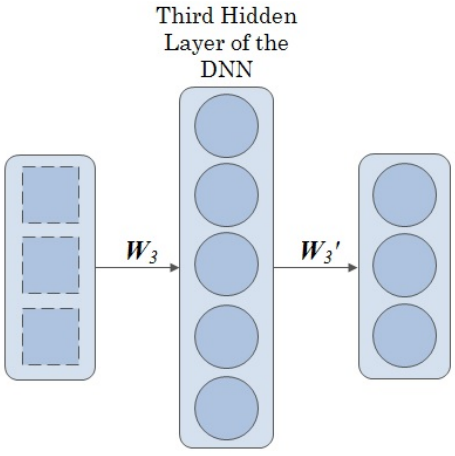

Figure 2.7: AE built to obtain the initialization of the weights of the third hidden layer.

of disturbance may be achieved by randomly choosing, for each training sample $\mathbf{x}$, $\zeta\%$ of the components of $\mathbf{x}$ to be set to 0. ζ is referred as the corruption rate.

The application of the greedy layer-wise training to DNNs considering dAEs (instead of AEs) is similar to the one described above. Such approach is known as stacked denoising auto-encoders (SdAs). It should be noted that the input used to train the k^{th} dAE, associated to the k^{th} layer of the DNN, is obtained by processing the original input $\mathbf{x}$ through the first $k-1$ trained layers of the DNN; the result is then subject to the corruption mechanism (see the illustrative example in Figure 2.8 - before being processed by the current dAE, in this case, the second dAE, the input $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)$ is processed by the (already) trained precedent layers of the DNN; the result, $\mathbf{y} = (y_1, y_2, y_3, y_4)$ is then subject to the corruption mechanism which consists, in this case, in setting 25% of the dAE inputs components to 0; finally, the corrupted input, $\mathbf{y} = (y_1, y_2, 0, y_4)$, is processed by the dAE).

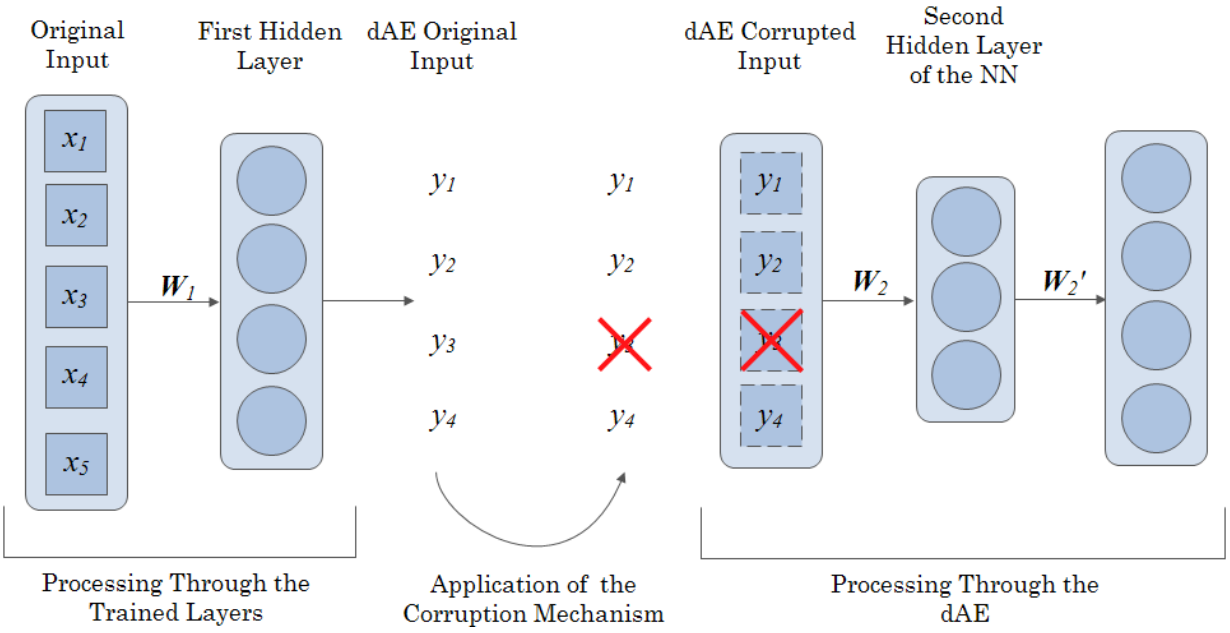

Figure 2.8: Illustrative example of the application of the corruption mechanism in the second dAE of the SdA considering the DNN in Figure 2.4.

2.4 Transfer Learning for Deep Feedforward Neural Networks

The learning process that each person goes through is influenced by his/her previous experience. When facing a new challenge, we, human beings, are capable of applying our knowledge to overcome it. It is expected that, if our previous acquired knowledge is directly

or indirectly related to the challenge, we are able to solve it more efficiently (in terms of time and solution quality).

As a result of the attempt to reproduce such process in machine learning, *transfer learning* was introduced. This term refers to the reuse of knowledge, associated to one or more *source problems*, on a *target problem* [41] (the attribute space, the data distribution and the number of classes may differ between the source and the target problems). In other words, given the sources of knowledge associated to the respective (source) problems, transfer learning consists in applying such knowledge to the original (target) problem in order to improve its knowledge. The knowledge improvement may translate in a higher initial training performance, a faster learning and/or a higher final training performance [50]. Transfer learning is thus particularly useful when the available training set is reduced [4] or rapidly becomes outdated as it occurs in the problem of locating a device based on its WiFi signal [42].

In spite of its improvement purpose, transfer learning may have the opposite effect. In that case, it is said that negative transfer has occurred; otherwise, the transfer is dubbed as positive. In order to avoid negative transfer learning, when transferring the knowledge, there are three main issues that need to be addressed: what to transfer, how to transfer and when to transfer. In what concerns to the "what to transfer" issue, transfer learning methods are categorized into instance-based transfer, feature-based transfer, parameter-based transfer and relational-transfer learning. In the case that the source and target attribute spaces are similar (but not the same) or the same but the respective data distributions differ, some of the source samples may be reused in the target problem by applying a re-weighting procedure; this type of transfer learning is referred as instance-based transfer. When the similarity between the attribute spaces is low, a possible approach, known as feature-based transfer, is to find a mapping such that, after its application on both source and target samples, the difference between the obtained source and target domains (by which we mean the attribute space and samples distribution), is reduced. The goal is to reuse the source samples on the target problem considering the new representation of the data. In parameter-based transfer, by assuming that the source and the target problems are related, then some of the structure/parameters may be transferred from the source problem to the target problem. Finally, the relational-transfer learning deals with data that can be described by relations between samples. Given related source and target problems, the idea is to map the relationships in the source domain to the target domain.

When restricted to the context of DNNs, transfer learning may be associated to the reuse of (already trained) DNN weights and biases on a new (untrained) DNN related to a different task [4, 14, 26]. The source of knowledge is the trained DNN, DNN_{source}, and the transfer consists in consider as initial weights and biases of the hidden layers in the target DNN, DNN_{target}, the respective weights and biases of the hidden layers of DNN_{source}. Starting from such trainable parameters, the DNN_{target} is then fine-tuned. Several variations may be considered. For example, we may consider to train only the output layer or to retrain also the last k layers. This type of transfer learning may be classified as parameter-based since the structure, namely, the hidden layers, are being shared.

Chapter 3

Computational Simulations

3.1 Equipment and Software

The software used to construct the model was Theano [7, 10], a Python library directed to the optimization and evaluation of mathematical expressions. The main data structures, such as the SdA class and the classes on which the SdA class is built, were based on the ones provided in [2].

When combined with the use of a Graphics Processing Unit (GPU), Theano allows to increase the processing speed. Since the number of parameters as well as the volume of data used in the training of a DNN is usually high, the network training is clearly a computationally heavy process. For this reason, the simulations executed to achieve the best model were performed on a GTX 770 GPU.

3.2 Stacked Denoising Auto-encoders Experiments

3.2.1 Datasets

The two considered datasets consisted of a set of images from microglial cells which were acquired in a DMI6000B inverted microscope using the ORCA-Flash4.0 V2 (Hamamatsu Photonics) CMOS camera. Images were exported as raw 16-bit TIFF using the LAS AF software with the original metadata preserved. TIFFs had their background subtracted in FIJI using the roller-ball ramp in between $35 - 50\%$ pixel radius. Images were segmented in FIJI using a modification of the triangle threshold algorithm for epifluorescence images. Each thresholded microglial cell was delineated using the particle analyze tool in calibrated images and exported to FIJI ROI manager (each cell was isolated in an image and the images vary in size and shape).

The datasets were disjoint but followed a similar class distribution. The main difference between them was that, in the second dataset (Dataset II) the labeling of the cells' state was discussed by a panel of four experts while in the first dataset (Dataset I) only two experts were considered. In Dataset II, the images of cells whose state did not reached to

a consensus on the experts panel were discarded.

The first dataset was composed by 227 images. The most common state in the dataset was the transition state while the less represented state was the resting state. The class distribution is shown in Table 3.1.

State	Number of Images	Percentage (%)
Resting	34	14.98
Transition	118	51.98
Active	75	33.04

Table 3.1: Distribution of the classes/states in Dataset I.

Dataset II was composed by 45 images (six samples of the dataset are presented in Figure 1.1). The class distribution was similar to Dataset I (the detailed dataset composition is described in Table 3.2).

State	Number of Images	Percentage (%)
Resting	7	15.56
Transition	22	48.89
Active	16	35.55

Table 3.2: Distribution of the classes/states in Dataset II.

In order to simplify the notation, the resting, transition and active states are coded as C1, C2 and C3, respectively.

3.2.2 Simulations

The experimental component was driven through simulations following a common structure. In each simulation, all images were resized to an image with dimension 30×30 and flatten to a vector of 900 entries which was used as input of the network. The dataset was then divided into a training set, a validation set and a test set with, respectively, 40%, 20% and 40% of the samples. The samples were normalized according to the maximum and minimum values of the samples in the training set (after the normalization each component/entry of the input assumed a value in $[0, 1]$).

A SdA of a fixed architecture of 5 hidden layers with 500 neurons each was then trained using the training set until the validation error started to increase significantly (executing a maximum of 1000 fine-tuning epochs). The training parameters used for the network were the following: 200 epochs for pre-training, 0.01 for the pre-training learning rate, 0.2 for the fine-tuning learning rate, 10 for the batch size and 0.1 for the corruption rate. The

latter implies that 10% of the input components were randomly chosen to be set to 0 (the input components set to 0 varied within samples).

Finally, the model was applied to the samples of the test set and its performance was evaluated using the Balanced Error Rate (BER). The BER is given by the average of the class errors; in this case, it is given by:

$$\text{BER} = \frac{E_{C1} + E_{C2} + E_{C3}}{3} \ , \tag{3.1}$$

where E_{C1}, E_{C2} and E_{C3} are the rates of wrong predictions within the classes C1, C2 and C3, respectively. The term *accuracy* will denote the value $(1 - \text{BER}) \times 100\%$.

The above procedure was repeated 20 times (by randomly shuffling the data) in order to capture the general performance of the model regardless of the samples used to train. The final result of the simulation is the average (and standard deviation) of the accuracy obtained in the 20 models.

Some of the algorithm's parameters were already defined in the acquired code and were not modified, namely, the training algorithm, the training stopping criterion and the cost and activation functions.

As training algorithm, the standard backpropagation (see Section 2.2.1) was applied. In the pre-training phase of each SdA, the dAEs were trained until the maximum number of epochs was reached. In the fine-tuning phase, as previously mentioned, a validation set was used to prevent overfitting.

With respect to the activation functions, the sigmoid function was applied in the dAEs neurons and in the hidden layers' neurons of the DNN. The activation function applied in the output layer of the DNN was the softmax function. The cost function used in the pre-training phase corresponded to the mean of the element wise CE across the mini-batches. In the fine-tuning phase, the considered cost function was the mean of the CE cost function across the mini-batches.

3.2.3 Results

Experiments I

In the first set of experiments we considered Dataset I. Initially, we used a stratified training set, that is, a training set which respected the class distribution of the dataset. Since the classes were not equally represented, the use of a stratified training set compromised the learning of the less represented classes (in this case, C1 and C3). The average accuracy was 32.51% with a standard deviation of 3.84 and we observed an error rate in class C1 of almost 100% in some of the repetitions.

In order to improve the results, we decided to square the images so that their ratio was kept on the resizing to 30×30 (a black background was added to the images so that the they become square before downsizing). This approach had no impact on the performance: only an average accuracy of 29.15% (with 1.77 of standard deviation) was achieved.

Alternatively, we considered a balanced training set, that is, a training set with the same number of samples for each class, and we repeated the simulations (first using the

original images and then using square images). Globally, the results remained the same although the class errors have been balanced. By using the original images, we obtained an average accuracy of 30.62% (with a standard deviation of 8.26) and by applying the squaring process, we reached an average accuracy of 29.54% (with a standard deviation of 5.66).

Given these results, we decided to use the original images and balanced training sets in the next simulations. The purpose of using the original images, instead of squared versions of them, was to avoid unnecessary pre-processing (the squaring did not improve the results). Moreover, the choice of using a balanced training set, instead of a stratified one, was driven by the need of a more homogeneous class learning.

Inspired by the approach presented in [5], we repeated the simulations by including (equally spaced) rotations of the images on the training sets. It was expected that the artificial increase of the training set using rotations would lead to better results. However, the results remained unchanged: $32.37 \pm 3.20\%$ of average accuracy.

Additionally, in an effort to reinforce the image contrast and, therefore, enhance the contours of each cell, we tried to define a mechanism which intensified the lighter tones of the images. This procedure, however, produced an undesirable effect: the resultant images were noisier than their originals (as we can observe in Figure 3.1 - in spite of the enhancement of the cell body after the normalization, it is observed a greater diffusion around the cell) whereby this approach was not useful to our problem (the accuracy remained similar: $31.56 \pm 7.51\%$).

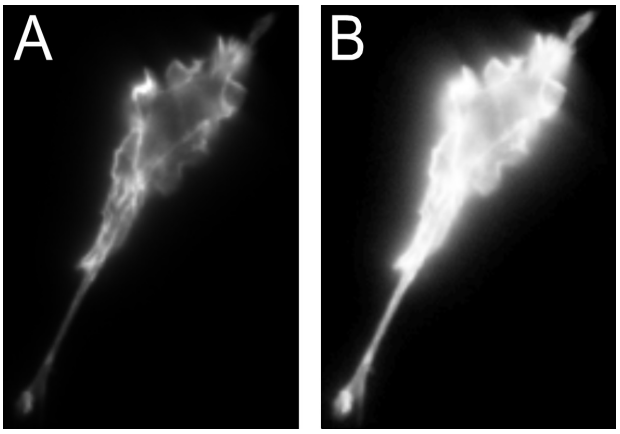

Figure 3.1: Dataset image before (A) and after (B) the normalization process to enhance the cell contour.

Given the morphological differences between cells of different states (see Sect. 1.1), both in thickness and in extension, we tried a different approach. Considering the mapping of each image into a matrix of pixels, the idea was to capture the intensities of each image by line and column. Thus, each image was compressed into a vector obtained by the concatenation of two vectors: a vector with the same size as the number of lines of the image, whose τ^{th} component was the sum of intensities of the pixels in line τ, and a vector

with the same size as the number of columns of the image, whose ι^{th} component was the sum of intensities of the pixels in column ι. The resultant vector was then resized to a vector of 60 entries and its entries were normalized to a value in $[0, 1]$ by dividing each entry by the maximum value of the vector (this vector was used as the network input). The obtained accuracy was, on average, $32.86 \pm 7.35\%$. This accuracy translate, once more, into no improvement.

Next, we tried to create a mask of the cells. The goal was to reinforce the cells contours by capturing the cells morphology. Each mask was obtained by defining a boolean image of the form $img < threshold$, where img was the original image and $threshold$ was a threshold whose value was in $[0, 255]$. The masks were then resized to 30×30 and flatten to vectors which were used as network inputs. Several constant values were tested for the threshold and, since the thresholds chosen were not appropriate for all the images (see Figure 3.2), we tried to define a function which varied according to the image ratio (width/length) and to the image minimum and maximum intensities but we did not reach more appropriate masks. Considering a fixed threshold, we achieved an accuracy of $36.19 \pm 9.29\%$. Figure 3.2 reflects the difficulty of creating a masking function which was appropriate for all the cells. We can observe that the built mask for the first cell captures its shape, however, if we observe the masks of the cells in images C and E, we notice two opposite effects. In the first case, the threshold is too small and the mask covers non-cell areas. On the other hand, the threshold is too high for the cell in image E and the mask does not incorporate the cell body completely.

Globally, the attempts of improving the model accuracy were not successful. Particularly, we found hard to define an appropriate pre-processing procedure for the images.

Given that the results were invariant despite all the considered approaches, we decided to observe the training and validation errors with the purpose of understanding whether the models were learning or not. We verified that, in the majority of the repetitions within the simulations, the error curves shared the same behavior. The training error was initially high and started decreasing as the number of epochs increased while the validation error remained nearly constant. The training error curve was having the expected behavior. The validation error, on the contrary, exhibited an irregular evolution: it was expected an decreasing of the error followed by an increasing. This may indicate that something is compromising the performance of the models.

Experiments II

Due to the low accuracy obtained using the first dataset, we felt the need of either request the relabeling of the samples on Dataset I by a panel with more experts or acquire a new dataset in which the labels were assigned by more than two experts. As discussed in Section 1.1, the cells' morphology is highly variant even within the same class which makes their classification difficult. Thus, the effort of several experts instead of just two may result in a more accurate labeling. In this sense, we obtained a new dataset, Dataset II, described in Table 3.2 which we considered for the new set of experiments.

Given the results obtained in the previous set of experiments, we only considered bal-

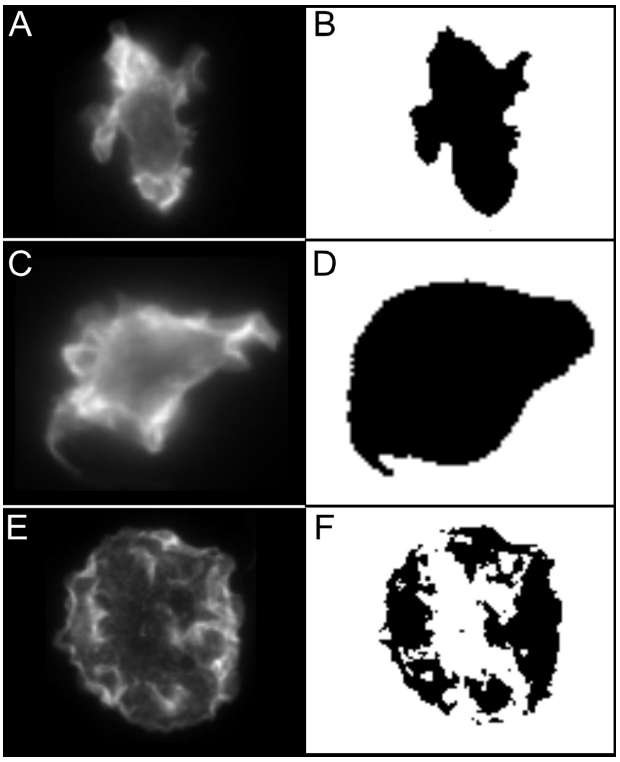

Figure 3.2: Cell images (left) and respective masks (right) generated with a threshold of 25.

anced training sets for the new set of experiments since it allowed an "uniform" learning of the 3 classes.

In the presence of such a small dataset as ours, we started by considering a training set with 3 samples per class. Given that a training set of 9 elements is poor, we artificially increased the training set by adding rotated versions (equally spaced) of the original images as in Experiments I (see Section 3.2.3). Thus, the training sets were obtained by randomly selecting images following a stratified division; the rotations (of the selected images) were then added. The number of rotations considered for each image within a class was nearly the same and was computed so that each class had the same number of elements for training. As an attempt to achieve higher accuracy, we varied the number of elements per class in the training set, in particular, we considered the values of 30, 50, 70 and 100 elements per class.

Analogously to the first set of experiments, we repeated the simulations considering squared images. This approach, in which all the images have a squared shape, will be called *squared* while the initial approach will be denoted as *non-squared*. Results are shown in Table 3.3.

We start by observing that artificially increasing the training set size is beneficial for the SdA learning. Still, it is clear that it is sufficient to use 50 (non-squared) or 30 (squared) images per class (the accuracy obtained with 70 and 100 images is not significantly higher).

	Non-squared	Squared
3 Images/class	54.81(7.51)	56.99(7.57)
30 Images/class	55.83(8.82)	63.75(7.46)
50 Images/class	60.05(6.46)	62.74(7.36)
70 Images/class	60.73(7.67)	65.16(7.03)
100 Images/class	60.91(6.42)	65.26(7.25)

Table 3.3: Average accuracy (in % over 20 repetitions, standard deviations in parenthesis) for the original 3-class problem varying the number of elements per class. Results for squared and non-squared images are shown.

It is also clear that the use of squared images is important. In fact, by using this strategy the downsizing step maintains the microglial cell aspect ratio which is fundamental as the recognition is based on the morphology of such cells. Overall, the results show that this is a hard problem with the highest accuracy around 64%, with squared images. Standard deviations are also high due to the small size of the original dataset.

To further investigate where does the difficulty lies, we repeated the study by considering some 2-class sub problems, namely C1 vs C2, C1 vs C3 and C2 vs C3. The results are shown in Table 3.4.

	C1 vs C2		C1 vs C3		C2 vs C3	
	Non-squared	Squared	Non-squared	Squared	Non-squared	Squared
3 Images/class	52.64(10.88)	56.39(12.24)	82.92(10.53)	82.29(9.58)	63.61(12.45)	66.39(11.76)
30 Images/class	58.03(12.67)	64.57(8.68)	83.63(8.27)	85.5(5.84)	67.42(9.37)	69.96(11.76)
50 Images/class	61.49(9.41)	65.43(8.97)	84.88(7.81)	85.75(7.03)	68.73(10.30)	70.92(11.55)
70 Images/class	60.87(10.68)	64.29(10.69)	84.86(8.53)	86.00(7.43)	67.31(8.97)	70.92(11.55)

Table 3.4: Average accuracy (in % over 20 repetitions, standard deviations in parenthesis) for the 2-class sub problems varying the number of elements per class. Results for squared and non-squared images are shown.

The lowest accuracy was obtained in the C1 vs C2 problem, while the highest accuracy was obtained in the sub problem C1 vs C3. This was expected due to the high morphological difference between cells in the states C1 and C3 (see Section 1.1). It is particularly evident the difficulty in distinguishing the transition state (class C2) from the others, specifically from class C1. In fact, the resting and transition states share some morphological similarities (such as elongated body and sharper boundaries) that hinder the SdA learning. We also observe that the performance of the models obtained in each repetition is highly variant. This is essentially due to the small size of the dataset and consequent heterogeneity within classes, making the models too dependent on the particular training

set.

Analogously to what was observed in the original problem, a slight improvement is obtained when squared images are used. Also, the use of rotations is still beneficial although with a smaller impact for the C1 vs C3 problem.

Considering the results obtained for the original problem and respective sub problems, we decided to approach the original problem with a hierarchical classification methodology [30]. Given the classes A and B, where $B = \{B_1, B_2\}$, the idea consists in building a classifier for the problem A vs B_1 vs B_2 from two binary classifiers: the first is trained to solve the problem A vs B while the second is trained to solve the problem B_1 vs B_2. Thus, in the final classifier, a sample is first presented to the classifier A vs B and if its predicted class is B, the sample is then presented to the second classifier in order to be classified either in B_1 or B_2. With this approach, we try to simplify the original problem by dividing it in two sub problems.

By observing the results depicted in Table 3.4, we verify a greater similarity between samples of classes C1 and C2 whereby the assignment $A \rightarrow$ C3, $B \rightarrow \{$C1, C2$\}$ may be the most appropriate among the available choices. In the experimental procedure executed following such assignment, we obtained an average accuracy of $52.46 \pm 7.09\%$ which is lower than the one obtained considering a single classifier to solve the original problem. Given that the samples of classes C2 and C3 also share some similarities, we decided to repeat the simulation using, this time, the $A \rightarrow$ C1, $B \rightarrow \{$C2, C3$\}$ assignment. Despite not significantly, the accuracy decreased: $50.38 \pm 7.26\%$. The last possible assignment, $A \rightarrow$ C2, $B \rightarrow \{$C1, C3$\}$, was expected to be, in the context of the problem, the less suitable since the samples of C1 and C3 classes have a different morphology while samples of C2 share similarities to both C1 and C3 classes' samples. This was verified experimentally: the average accuracy, when considering the later assignment, was $45.37 \pm 9.88\%$, the worst of the 3 possibilities.

Back to the original approach for the C1 vs C2 vs C3 problem, in an effort to understand if the architecture of the network was compromising its performance, we decided to vary the number of hidden layers of the network and repeated the simulations considering 3, 7, 10 and 15 hidden layers. Due to the results presented in Table 3.3, we considered, for these simulations, 30 squared images per class. The results are shown in Table 3.5. It should be noted that the appropriate number of elements per class may vary between different architectures, however, such study will not be performed.

The increasing of the network complexity did not introduce performance improvements. On the contrary, we verify a decline in the average accuracy of the models whereby the initial architecture is appropriate to the problem. It should be noted that the training set may not be sufficiently large to successfully train more complex networks (with more than 5 hidden layers), which may justify such performance decline.

Since, according to the discussion in Section 2.2.3, the online mode may lead to better results, we decided to change the batch size to 1 thus simulating the online mode. We repeated the simulation (considering the initial architecture and 30 squared images per class) and obtained an average accuracy of $63.21 \pm 8.89\%$. Such result indicates that the online mode is not beneficial for our problem, at least when using a small dataset as Dataset

Number of Hidden Layers	Accuracy
3	62.27(7.58)
5	63.75(7.46)
7	60.45(7.67)
10	59.57(7.86)
15	56.32(10.97)

Table 3.5: Average accuracy (in % over 20 repetitions, standard deviations in parenthesis) for the 3-class problem varying the number of hidden layers. The colored line relates to the initial architecture.

II.

3.3 Transfer Learning Experiments

3.3.1 Datasets

For the source network, five image datasets (respectively called babyAIshapes 1, modified babyAIshapes 1, babyAIshapes 2, shapes and small MNIST), presented in Table 3.6, were considered. The (original and modified) babyAIshapes 1 datasets consisted in datasets of colored images of 3 basic shapes: circle, square and triangle (the images of both datasets were modified to a gray tone). In the latter dataset, the images whose shape color had a lower value than the background were modified in order to fill the shape with a higher value than the background. The idea was to render the dataset images more identical to the images of microglial cells (in which the background is black - have a lower value - and the cell has a lighter tone which is associated to a higher tone value). Similarly, the babyAIshapes 2 dataset was composed by colored images of 3 basic shapes: ellipse, rectangle and triangle (and the images were also modified to a gray tone). The shapes dataset was composed by images (in gray tones) of circles, ellipses, squares and rectangles. The shapes in the four datasets just described varied in size, orientation and position. The small MNIST dataset was a subset of the MNIST dataset which is composed by gray images of handwritten digits.

The dataset considered for the target network was Dataset II (described in 3.2.1).

3.3.2 Simulations

The implementation of transfer learning was divided in two phases. In the first phase, a DNN for the source problem was created. The training of the source DNN, along with its performance assessment, followed the procedure described in Section 3.2.2. The training parameters and the architecture - except eventually for the the number of inputs and

Dataset Name	Classification Problem	Number of Classes	Image Dimensions	Number of Training Samples
babyAIshapes 1 [1]	Basic shapes	3	32×32	10000
modified babyAIshapes 1	Basic shapes	3	32×32	10000
babyAIshapes 2 [1]	Basic shapes	3	32×32	10000
shapes	Shapes	4	20×20	100
small MNIST [3]	Handwritten digits	10	28×28	10000

Table 3.6: Brief description of the source datasets.

output neurons - were the same as previously considered, however the process was not repeated 20 times, it was executed just once.

The second phase consisted in initializing the hidden layers weights and biases of the target DNN, with the respective weights and biases of the source DNN. The target DNN was then (fully) trained using backpropagation. This procedure is similar to the greedy-layer wise approach, the difference is that the pre-training is replaced by the initialization using the source DNN. For this reason, the code created for the simulations described in Section 3.2.2 was reused to create the target DNNs by slightly changing it to perform such initialization instead of the unsupervised pre-training.

As target DNN training set, a subset of 30 squared images per class was considered (this choice was based on results presented in Section 3.2.3). The input dimensions of the target DNNs were adjusted to assume the same value as the input dimensions of the respective source DNN.

3.3.3 Results

Following the procedure described in the previous section, we obtained source DNNs whose accuracy is shown in Table 3.7. The source DNN which exhibited the highest accuracy was the one associated to the small MNIST dataset, while the DNN with the lowest accuracy was associated to the shapes dataset, which had the smallest training set, about 100 times smaller than the others.

Given that the attribute space dimensions of the target DNNs differed from the one considered in Section 3.2, we repeated the SdA simulations for the original problem (microglia) by considering images of $20 \times 20, 28 \times 28$ and 32×32 dimensions and a training set of 30 squared images per class. The goal was to compare the performances of the DNNs applying SdAs and transfer learning. The accuracy of the target DNNs and respective SdAs (baseline) is shown in Table 3.8.

Regardless of the source dataset, we observe that transfer learning did not have a significant impact on the accuracy of the target DNN, when comparing to the accuracy achieved using only SdAs.

The lack of improvement may be justified by several factors. First, the source problems

Dataset	Source DNN Accuracy
babyAIshapes 1	91.89
modified babyAIshapes 1	70.43
babyAIshapes 2	74.17
shapes	65.50
small MNIST	96.60

Table 3.7: Accuracy (in %) for the source DNNs varying the source dataset.

Source Dataset	Target DNN Accuracy	SdA Accuracy
babyAIshapes 1	63.89(6.95)	62.59(7.38)
modified babyAIshapes 1	65.85(6.99)	62.59(7.38)
babyAIshapes 2	60.21(7.84)	62.59(7.38)
shapes	63.35(7.82)	61.16(8.37)
small MNIST	58.46(8.45)	60.70(7.93)

Table 3.8: Average accuracy (in % over 20 repetitions, standard deviations in parenthesis) for the target problem (varying the source dataset) and respective SdA, built considering the same input dimensions as the target DNN.

may not be sufficiently related to the target problem. If we consider the four datasets associated to the classification of shapes, it is possible to find similarities between the source and the target problems, for example, an active microglial cell usually has a circular shape (the resting and transitive cells, on the other hand, have an irregular and variant shape and do not assume any specific geometric shape). In what concerns to the problem associated to the small MNIST dataset, there is no clear relation between it and the target problem. The poor suitability of the source problems becomes more evident when we compare the performances of the source DNNs (Table 3.7) to the performances of the respective target DNNs (Table 3.8). We observe that the source DNN exhibiting the highest accuracy, associated to the small MNIST dataset, gave rise to the target DNN with the lowest accuracy. On the other hand, the source DNN associated to the modified babyAIshapes 1, which exhibits one of the lowest accuracy, resulted in the target DNN with the highest accuracy. These results reinforce that the source problem associated to the modified babyAIshapes 1 dataset is more related to the target problem than the source problem associated to the small MNIST dataset.

Once more, and due to the reasons discussed in Section 3.2.3, the size of the dataset may had also compromised the performance of the resultant target DNN.

Chapter 4

Conclusions

In this work, we applied stacked denoising auto-encoders to the classification problem of identifying a microglial cell's state. We started by considering a dataset with 227 images. It should be noted that such dataset is small when compared to the datasets usually used to train DNNs (for example, the MNIST dataset [3], which is a dataset of handwritten digits, has 60000 training samples). We initially used a stratified training set with no image pre-processing and obtained a global result of approximately 33%. In an effort to improve the performance, we tried multiple attempts, namely, by using balanced training sets, artificially increasing the dataset or implementing some image pre-processing procedures such as the image squaring, the low intensities reinforcement and the use of masks.

Since the accuracy using the first dataset was invariably low, less than 37% in all the attempts, we acquired a new dataset in which the labels were discussed by more than two experts. In spite of the small size of the second dataset, it was possible to observe a significant performance improvement when comparing to the performances obtained using the initial dataset. This improvement indicates the existence of incorrect labels on the first dataset, justifying the low accuracy obtained using it. Particularly, these results reinforce the difficulty of distinguishing the states and, therefore, reveal the need of several experts (when classifying in a non automatic way). The non automatic process becomes inefficient in the sense that requires the participation of several experts and is slow (mainly when leading with large amounts of cells). Such issues exalt the importance of an automatic model construction.

Despite the poor results using Dataset I, it was possible to observe that the use of a balanced training set is beneficial. It should be noted that, although class C2 is the most important to detect and the most represented in the dataset, a small error rate in class C2 is not necessarily advantageous, namely when the less represented classes (C1 and C3) exhibit a high class error as it was occurring when the training set followed a stratified division (the majority of the samples were being classified in class C2; the classes C1 and C3 were not properly learned). Furthermore, the attempts performed using the first dataset, namely, the several pre-processing approaches, revealed the difficulty in defining a suitable procedure, that is, a procedure which was adequate for all the images and allowed the distinction of the states.

In the second set of SdA experiments, in which we used the second dataset, by considering the original 3-class problem as well as some binary sub problems of it, we conclude that the transition state is the most difficult to recognize, mainly from the resting state. This is due to the morphological similarities between those classes that cause some disagreement even between the experts panel. We also found important to square the images in order to maintain their aspect ratio preventing, in that way, morphological distortions with the downsizing step. Rotated versions of the original images were also used to diminish the effect of having a (very) small size dataset.

Our last approach, which consisted in the application of transfer learning, was not successful in terms of performance improvement. The results were similar to the ones obtained considering SdAs, that is, similar to the ones obtained without resorting to transfer learning and may indicate the need of considering a more related source problem.

Globally, the generalization ability of the created models may also have been compromised because the dataset is not enough representative of the microglia population's morphology, even within the same class (recall that Dataset II has only 45 samples). In general, the experiments show that it is essential to collect more data in order to improve the performances of the models.

Besides the small number of samples in the dataset, the reduced available time was also a major limitation. It would be interesting to approach the problem using another methodology, namely, CNNs. Furthermore, a deeper search in what concerns the network architecture and the training parameters may be also beneficial.

Bibliography

[1] BabyAIshapes datasets. http://www.iro.umontreal.ca/~lisa/twiki/bin/view.cgi/Public/BabyAIShapesDatasets, accessed on 2015-06-01

[2] Deep learning structures. http://deeplearning.net/tutorial/code/, accessed on 2015-03-01

[3] MNIST dataset. http://yann.lecun.com/exdb/mnist/, accessed on 2015-06-01

[4] Amaral, T., Kandaswamy, C., Silva, L., Alexandre, L., Marques De Sá, J., Santos, J.: Improving performance on problems with few labelled data by reusing stacked auto-encoders. In: 13th International Conference on Machine Learning and Applications (ICMLA'14). pp. 367–372 (2014)

[5] Amaral, T., Silva, L.M., Alexandre, L.A., Kandaswamy, C., de Sá, J.M., Santos, J.: Transfer learning using rotated image data to improve deep neural network performance. In: Image Analysis and Recognition, Lecture Notes in Computer Science, vol. 8814, pp. 290–300. Springer International Publishing (2014)

[6] Arbib, M.A.: Brains, Machines, and Mathematics. Springer-Verlag New York, Inc. (1987)

[7] Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I.J., Bergeron, A., Bouchard, N., Bengio, Y.: Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop (2012)

[8] Bengio, Y.: Learning deep architectures for AI. Foundations and trends in Machine Learning 2(1), 1–127 (2009)

[9] Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Advances in Neural Information Processing Systems 19. pp. 153–160. MIT Press (2007)

[10] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., Bengio, Y.: Theano: a CPU and GPU math expression compiler. In: Proceedings of the Python for Scientific Computing Conference (SciPy) (2010)

[11] Bishop, C.M.: Neural networks for pattern recognition. Oxford university press (1995)

[12] Bourlard, H., Kamp, Y.: Auto-association by multilayer perceptrons and singular value decomposition. Biological Cybernetics 59(4), 291–294 (1988)

[13] Castro, J.L., Mantas, C.J., Benítez, J.M.: Neural networks with a continuous squashing function in the output are universal approximators. Neural Networks 13(6), 561–563 (2000)

[14] Ciresan, D., Meier, U., Schmidhuber, J.: Transfer learning for latin and chinese characters with deep neural networks. In: The 2012 International Joint Conference on Neural Networks (IJCNN'12). pp. 1–6 (2012)

[15] Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20(3), 273–297 (1995)

[16] Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals and Systems 2(4), 303–314 (1989)

[17] Dowlati, M., de la Guardia, M., Dowlati, M., Mohtasebi, S.S.: Application of machine-vision techniques to fish-quality assessment. TrAC Trends in Analytical Chemistry 40, 168 – 179 (2012)

[18] Erhan, D., Manzagol, P.A., Bengio, Y., Bengio, S., Vincent, P.: The difficulty of training deep architectures and the effect of unsupervised pre-training. In: The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS'09). pp. 153–160 (2009)

[19] Fukushima, K.: Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. Biological Cybernetics 36(4), 193–202 (1980)

[20] Hadsell, R., Erkan, A., Sermanet, P., Ben, J., Kavukcuoglu, K., Muller, U., LeCun, Y.: A multi-range vision strategy for autonomous offroad navigation. In: The 13th IASTED International Conference on Robotics and Applications (RA'07). pp. 457–463. ACTA Press (2007)

[21] Hastad, J.: Almost optimal lower bounds for small depth circuits. In: The Eighteenth Annual ACM Symposium on Theory of Computing (STOC'86). pp. 6–20. ACM (1986)

[22] Hinton, G.E.: Connectionist learning procedures. Artificial Intelligence 40(1-3), 185–234 (1989)

[23] Hinton, G.E., Salakhutdinov, R.R.: Reducing the dimensionality of data with neural networks. Science 313(5786), 504–507 (2006)

[24] Hinton, G.E., Osindero, S., Teh, Y.W.: A fast learning algorithm for deep belief nets. Neural Computation 18(7), 1527–1554 (2006)

[25] Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural Networks 2(5), 359–366 (1989)

[26] Huang, J.T., Li, J., Yu, D., Deng, L., Gong, Y.: Cross-language knowledge transfer using multilingual deep neural network with shared hidden layers. In: The 2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 7304–7308 (2013)

[27] Hubel, D.H., Wiesel, T.N.: Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. The Journal of Physiology 160(1), 106–154 (1962)

[28] Hubel, D.H., Wiesel, T.N.: Receptive fields and functional architecture of monkey striate cortex. Journal of Physiology (London) 195(1), 215–243 (1968)

[29] Joo, S., Moon, W.K., Kim, H.C.: Computer-aided diagnosis of solid breast nodules on ultrasound with digital image processing and artificial neural network. In: The 26th Annual International Conference of the IEEE Engeneering in Medicine and Biology Society. vol. 1, pp. 1397–1400 (2004)

[30] Kumar, S., Ghosh, J., Crawford, M.M.: Hierarchical Fusion of Multiple Classifiers for Hyperspectral Data Analysis. Pattern Analysis & Applications 5(2), 210–220 (2002)

[31] Lawrence, S., Giles, C., Tsoi, A.C., Back, A.: Face recognition: a convolutional neural-network approach. IEEE Transactions on Neural Networks 8(1), 98–113 (1997)

[32] LeCun, Y.: Generalization and network design strategies. In: Connectionism in Perspective. Elsevier (1989)

[33] LeCun, Y., Boser, B., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W., Jackel, L.D.: Backpropagation applied to handwritten zip code recognition. Neural Computation 1(4), 541–551 (1989)

[34] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)

[35] LeCun, Y., Bottou, L., Orr, G., Muller, K.: Efficient backprop. In: Neural Networks: Tricks of the trade. pp. 9–50. Springer (1998)

[36] Liu, S., Liu, S., Cai, W., Pujol, S., Kikinis, R., Feng, D.: Early diagnosis of alzheimer's disease with deep learning. In: The IEEE 11th International Symposium on Biomedical Imaging (ISBI'14). pp. 1015–1018 (2014)

[37] Magoulas, G.D., Vrahatis, M.N., Androulakis, G.S.: Improving the convergence of the backpropagation algorithm using learning rate adaptation methods. Neural Computation 11(7), 1769–1796 (1999)

[38] Maleki, M., Teshnehlab, M., Nabavi, M.: Diagnosis of multiple sclerosis (MS) using convolutional neural network (CNN) from MRIs. Global Journal of Medicinal Plant Research 1(1), 50–54 (2012)

[39] Nagaraj, S., rao, G.N., Koteswararao, K.: The role of pattern recognition in computer-aided diagnosis and computer-aided detection in medical imaging: A clinical validation. International Journal of Computer Applications 8(5), 18–22 (2010)

[40] Nielsen, M.A.: Neural Networks and Deep Learning. Determination Press (2015), `neuralnetworksanddeeplearning.com`

[41] Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Transactions on Knowledge and Data Engeneering 22(10), 1345–1359 (2010)

[42] Pan, S.J., Zheng, V.W., Yang, Q., Hu, D.H.: Transfer learning for wifi-based indoor localization. In: Workshop on Transfer Learning for Complex Task of the 23rd AAAI Conference on Artificial Intelligence (2008)

[43] Plagianakos, V.P., Sotiropoulos, D.G., Vrahatis, M.N.: Automatic adaptation of learning rate for backpropagation neural networks. Recent Advances in Circuits and Systems 337 (1998)

[44] Ranzato, M.A., Szummer, M.: Semi-supervised learning of compact document representations with deep networks. In: The 25th International Conference on Machine Learning (ICML'08). pp. 792–799 (2008)

[45] del Rio-Hortega, P.: Microglia. In: Cytology and Cellular Pathology of the Nervous System. pp. 481–534 (1937)

[46] Rojas, R.: Neural Networks: A Systematic Introduction. Springer-Verlag New York, Inc. (1996)

[47] Rosenblatt, F.: The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review 65(6), 386–408 (1958)

[48] Rumelhart, D., Hintont, G., Williams, R.: Learning representations by back-propagating errors. Nature 323, 533–536 (1986)

[49] Saijo, K., Glass, C.K.: Microglial cell origin and phenotypes in health and disease. Nature Reviews Immunology 11, 775–787 (2011)

[50] Torrey, L., Shavlik, J.: Transfer learning. In: Handbook of Research on Machine Learning Applications. IGI Global (2009)

[51] Vincent, P., Larochelle, H., Bengio, Y., Manzagol, P.A.: Extracting and composing robust features with denoising autoencoders. In: The 25th International Conference on Machine Learning (ICML'08). pp. 1096–1103 (2008)

[52] Werbos, P.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. Ph.D. thesis, Harvard University (1974)

[53] Widrow, B., Hoff, M.E.: Adaptive switching circuits. In: 1960 IRE WESCON Convention Record, Part 4. pp. 96–104. IRE (1960)

[54] Wildes, R., Asmuth, J., Green, G., Hsu, S., Kolczynski, R., Matey, J., McBride, S.: A machine-vision system for iris recognition. Machine Vision and Applications 9(1), 1–8 (1996)

[55] Wilson, D.R., Martinez, T.R.: The general inefficiency of batch training for gradient descent learning. Neural Networks 16(10), 1429–1451 (2003)