Deep Neural Networks with Flexible Activation Function

Faculty of Information Engineering, Informatics, and Statistics
Master in Artificial Intelligence and Robotics

Candidate
Alberto Marinelli
1571560

Supervisor
Aurelio Uncini

Co-supervisor
Simone Scardapane

Academic year: 2015/2016
Abstract

Deep Neural Networks with Flexible Activation Functions

Artificial Neural Networks include a wide variety of architectures, training techniques and activation functions. While many advances have been made through architectures and training techniques methods, activation functions, that reach better results through adaptation compared to fixed ones, are not widely used yet. In this work, an Autoencoder with Flexible Spline Activation Functions (SAFs) is presented, where each layer neuron has an independent adaptable activation function. This is motivated by the fact that the local adaptation properties of SAFs allow to not modify the entire shape of the activation function globally but locally. In this way, each neuron implements a possibly different activation function with different statistics that has been learned from data. Moreover, how to build and train an Autoencoder implementing (SAFs) from scratch will be shown. Some tests will prove the effectiveness of the model against a standard one that uses Hyperbolic activation function on different datasets. Finally, a Deep Neural Network made of stacked Autoencoders is presented in order to compare the learning capabilities of Flexible Spline Activation Functions against a traditional Activation Function.
## Contents

Abstract i

List of Figures iv

List of Tables vi

1 Introduction 1
   1.1 Deep Neural Networks 1
   1.2 Supervised Learning and Unsupervised Learning 4
   1.3 Related Work 5

2 Artificial Neural Networks 7
   2.1 Definition 7
   2.2 Activation Functions 11
      2.2.1 Step Function 12
      2.2.2 Logistic Function 12
      2.2.3 Rectifier (ReLU) 13
   2.3 Backpropagation 14
   2.4 Autoencoder 16

3 Spline Activation Function 19
   3.1 Spline interpolation scheme 19
   3.2 Spline Generalized Sigmoidal Neuron 22

4 Spline Autoencoder 25
   4.1 Definition 25
   4.2 Training 27

5 Implementation and Results 29
   5.1 Implementation 29
   5.2 Results 31
   5.3 Deep Neural Network Discussion 36

6 Conclusion 40
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Schematic representation of how a Convolutional Neural Network learns features by itself. Image taken from Geoff Hinton, Yoshua Bengio &amp; Yann LeCun presentation at NIPS’2015.[12]</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>Feedforward NN dependency graph with 3 input neurons, 2 hidden neurons and one output neuron.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Single neuron based on the McCulloch &amp; Pitts model [21]</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Plot of 2.9 for an interval between -6 and 6.</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Schematic representation of an Autoencoder. The circles are the neurons, which are in the same number in the input and in the output layer, as the goal of an Autoencoder is to reconstruct the input data.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of $q_{y,i}$ control points interpolation using a CR-spline function with a fixed step for the x-axes control points $\Delta x = q_{x,i} - q_{x,i-1}$[27]</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison of the test errors in a run of Adagrad compared to Stochastic Gradient Descent using 0.1 and 0.01 as learning rates for both weights and 0.05 for control points. Adagrad for both weights and control points updates has the parameter $\alpha$ set to 0.01.</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison of the test errors in a training run of a Standard Autoencoder, represented as “Standard AE”, against the Spline Autoencoder, represented as “Spline AE”. The parameters used are the ones reported in table 5.1 while the dataset is the Chemical Dataset.</td>
<td>33</td>
</tr>
<tr>
<td>5.3</td>
<td>Comparison of the objective function values for the Spline Autoencoder and the Standard autoencoders implementations</td>
<td>34</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of the test errors, training errors and $\Delta x$ values over 9 different values. Dataset used for the run is Chemical Dataset.</td>
<td>35</td>
</tr>
<tr>
<td>5.5</td>
<td>Plots of samples of the Flexible Spline Activation Function taken from neurons used for the digits dataset test.</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>Plots of samples of the Flexible Spline Activation Function taken from neurons used for the Calhousing dataset test. It is intuitive that the solution are too dependent on data, preventing the network to reach a good generalization</td>
<td>37</td>
</tr>
</tbody>
</table>
5.7 How to train a Deep Neural Network based on greedy layer-wise pre training of Autoencoders. First the Autoencoders are trained singularly, then their encoding weights are used from $X$ to $H_1$ and from $H_1$ to $H_2$ to be used in classification or regression tasks that take part in the last layer, which in this case is a neuron that can perform binary classification or regression. . . . . . . . . . . . . . . . 38
List of Tables

5.1 Results for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Chemical Dataset. . . . . . . . . . . . . 33
5.2 Values for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Digits Dataset. . . . . . . . . . . . . 34
5.3 Values for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Calhousing Dataset. . . . . . . . . . . . . 35
5.4 Best classification scores (in terms of accuracy) obtained over 15 runs for training and testing of a Standard Stacked Deep Neural Network and a Spline Deep Neural Network on the Digits dataset. . 38
To my family, who have always supported me throughout my career...
Chapter 1

Introduction

1.1 Deep Neural Networks

Deep Neural Networks are Neural Network architectures that exploit the combination of multiple layers to model feature representation. This is possible using learning algorithms as Backpropagation, that is useful for gradient computation of a cost function minimized by a Neural Network. Many layers allow to process the input data in a hierarchical way, where first layers have a lower representation of features while higher layers have more high level representation.

Deep Learning architectures have reached a breakthrough in many tasks such as image recognition, speech and audio processing. Before very effective Deep Learning techniques, which have made a big improvement over traditional Neural Networks, extracting features from raw data was hard, and it required manual engineering of features. Features used as a representations or feature vectors, could be used to train a classifier to recognize patterns. Deep Learning methods are very good in representation learning, which is the extraction of useful information from raw data used for classification or detection of patterns. These techniques do not need a manual engineering of features. Instead they require the tuning of internal parameters that will automatically learn useful features. Deep Learning exploits the generalization and high level representation capability of Neural Networks with many layers (Deep Neural Networks) that transform, from one layer to the other, input data from low level to high level features.
For example, let’s consider an input image that is given as input data to a Deep Neural Network. Every layer processes the data from the layer before. This way in the first layer edges present in the image are usually detected. The second layer takes, as input, the output of the first layer and will learn to detect small patterns based on the learned edges. The third layer and the higher level layers will learn to combine the small patterns into part of objects present in the image and so on, with last layers that can learn entire objects present in the image. By presenting many examples of several objects, the Neural Network learns to generalize the learned features, and it is able to classify new unseen data based on the general information it has stored.

![Figure 1.1: Schematic representation of how a Convolutional Neural Network learns features by itself. Image taken from Geoff Hinton, Yoshua Bengio & Yann LeCun presentation at NIPS’2015.][12]

Although the universal approximation theorem [11] proves that Feed Forward Neural Networks with one hidden layer can approximate continuous and multivariate functions, by setting the proper hidden neurons and learning rate, Deep Neural Network architectures that are trained with Deep Learning techniques are much better in handling more complex input-output relationships that arise from real world data. The latest breakthrough in Deep Neural Networks was in 2006 when three researchers (Hinton et al., 2006; Bengio et al., 2007; Ranzato et al., 2007) managed to train Networks with many layers. In particular, Hinton showed that a Deep Belief Network could be efficiently trained using greedy layer-wise pre training and Bengio and Ranzato adapted the same strategy to train many different Deep Network models. This allowed to train deeper architectures and thus getting much better results in test examples. In that way, neural networks are getting
closer to AI, replacing other methodologies based on other machine learning algorithms or hand designed feature representation. Many state of the art result have been achieved for image recognition problems [13] and on tasks that can get a lot of attention from the media such as games [14] and videogames [15].

One of the biggest problems when training networks with many hidden layers using gradient descent techniques, is the vanishing gradient. For the case of Feedforward Neural Networks, a parameter change in the output layer, will result in a small change in the first layers, which yields to very slow learning capabilities of first layers with respect to the last layers. This problem is present with Sigmoid and Tanh Activation functions, that are widely used activation functions for Feed Forward Neural Networks. Autoencoders are semi-supervised learning algorithms widely used in Deep Neural Networks. They allow to extract complex information from data in an semi-supervised way. Stacking pre-trained Autoencoders composing many layers, solves the vanishing gradient problem and it is one of the most important technique for Deep Learning. In this work, a first approach on greedy layer-wise pre training approach is presented as preliminary work, pointing out some directions that can be followed in the future.

Spline Activation Functions (Spline AFs) [33] [31], are an alternative approach where the function shape is adapted locally, using a lookup table interpolated by a local cubic polynomial curve. In the case of adaptive filters, the table entries are adapted using least mean square approach. The main advantage is that a local adaptation of the AF shape is done, allowing to prevent an update from changing the overall shape of the AF. Spline AFs use a small number of parameters and only a fixed number of them is used to compute the output of the function [19]. Most of research on Spline AFs was made in online settings with updates made one sample at a time and only recently investigations on batch and mini-batch training are made [1].
1.2 Supervised Learning and Unsupervised Learning

In Machine Learning field, Supervised Learning is a very common method to train a model in order to make predictions or classify data given target values. A typical Supervised Learning problem is when there is some labeled data, or data pairs, where each data sample has a target value. For example, in the famous problem of Digits recognition, the data pairs are composed of an image (pixel values) and a target label, which indicates which number corresponds to the image. This is a typical example of Classification task. A Regression example would be having some data pairs, that describe house pricing. For instance, the data would be composed of numerical indexes that rates the house and the neighborhood quality, while the target value is the house price. Supervised Learning for Regression and Classification tasks, tries to learn a function \( f(x) = y \) where \( x \) is the data and \( y \) is the target value. Once the function \( f(x) \) has been learned, it should be able to predict new unseen values. The main difference between Classification and Regression is that in Classification, the target values \( y \) take discrete values, which can be 1 or 0 for binary problems, or in the case of multiclass, from 0 to 9 like for digits classification. A problem is called a Regression problem, if the target values are continuous, like the price of a house based on its characteristics, which of course can not be divided into a finite number of classes. In the case of Neural Networks, the learning approach is to find a set of suitable weights, or parameters, in order to minimize a cost function that express the distance, or error, between the output and the target values. A mathematical approach to Neural Networks will be given in section 2. Other Supervised Learning techniques that are not discussed here are Support Vector Machines, Naive Bayes Classifier and many other. For an introduction on these topics see [17].

Unsupervised Learning is a learning approach where there is no labeled data to use as a target for the function. In this way, the function has to extract data features without knowing how these features are. For example, in the case of clustering, the objective is to find hidden relations between data and group it according to the rules discovered. In the case of Neural Networks, there are some network architecture that are used for Unsupervised Learning, which are Adaptive Resonance Theory and Self Organizing Maps, that takes a winner-takes-all approach for neurons to learn from unlabeled data. These techniques are inspired by the
capacity of the mammal brain to respond to visual, auditory and tactile stimulus [18].

Autoencoders, which are presented in this work, can be seen as a way between Supervised and Unsupervised Learning. In fact, since the method tries to learn hidden data structure, reconstructing the input without a separate target signal, it can be seen as Unsupervised technique. However, Autoencoder uses a target signal that is the input data itself, which is different on how usually Supervised Learning works, because the target data is different from the input. For this reason an Autoencoder can be seen as a semi-supervised Learning technique.

1.3 Related Work

An important element for performance and results of a Neural Network is the choice of the Activation Function. While the results achieved by Neural Networks, as said before, have reached the state of the art in many fields, they currently use Activation Functions that are considered good after selecting by hand between some of them. According to this approach, it seems there is no reason to think that a certain nonlinearity is optimal for a given problem. Moreover, sharing the same nonlinearity all over the layers and neurons might be nonoptimal. As shown by Agostinelli et al. [2] where they used an adaptable piecewise linear function with different parameters, the standard one-activation-function-fits-all approach may be suboptimal. There are many researches on adaptable activation functions but there are not many applications in the field. A simple approach on adaptation can be found in [3] and [4]. Each Sigmoid function was adapted using some parameters that changed the shape of the AF. Polynomial AFs were also investigated. The polynomial coefficients were adapted by gradient descent approach [5]. More work on this topic can be found in [6][7][8][9][10]. One of the biggest drawbacks of these approaches, is that the change of shape in the activation function is done globally. This could cancel previous adaptations and affect other regions of the AF.

Following the results of [1], this work is focused on how to train a Neural Network with flexible activation functions[1]. In particular, here will be presented how to design and train Autoencoders with flexible activation functions and stack them to form a Deep Neural Network using greedy layer-wise pre-training. The Thesis work is divided as follows. Chapter 2 is a recap on Neural Networks, Activation
functions and the Backpropagation algorithm. Chapter 3 introduces the Spline
AF and Chapter 4 introduces the proposed Autoencoder and the Deep Network
architecture. Chapter 5 discuss about conclusions.
Chapter 2

Artificial Neural Networks

2.1 Definition

In the field of Machine Learning what is called Artificial Neural Network (ANN) or Neural Network (NN), is a family of models or architectures vaguely inspired on how we know the biological brain works. The main components of a Neural Network are the Neurons, or processing units (PU). PUs are linked together through connections and they perform a form of processing that is local, in a way that the result depends on the values that arrives from the inputs and its local memory (if present). NNs functionalities are similar to biological brain ones. Knowledge to represent is acquired from the outside through some learning mechanism and the connection between Neurons is defined by Synaptic Weights that store the acquired knowledge. The learning algorithm is used to modify the Synaptic Weights such that they are able to minimize a certain cost function. There are many different kind of NN and a way to identify and classify them is by

- Network Architecture, or topology
- Neuron response characteristics
- Learning Rule

The Network Architecture is how the Neurons are connected between them. Usually, in the Feed Forward Neural Network model, Neurons are disposed in layers,
where neurons of the same layers are connected only with neurons of adjacent layers.

The network needs a Learning Algorithm based on some learning rules to approximate an usually unknown function based on the input data. Learning depends on the Neuron response value, weights or connections values and, in the case of supervised learning, the target values. A very popular learning algorithm is the Backpropagation algorithm that will be discussed in Section 2.3. In the case of supervised learning, the NN is fed with many training examples that are divided into an input signal and a reference output. The aim is to change the weights values in order to compute an output that is very close to the output used as reference. Weights are changed iteratively after presenting many samples present in the training set, and the learning process stops when no significant changes are made or when the network is not able to learn anymore.

In order to be clear and consistent in the whole document a basic notation is presented that will be integrated in the future chapters when new concepts will be introduced. This model is based on the Multi Layer Perceptron (MLP) [20]. Let’s consider a general network that, similarly to figure 2.1 has an input layer, an hidden layer and an output layer where the hidden and output neurons are MLP adaptive neurons. We define then

- \( L \): number of layers
• $l$: layer index
• $N_l$: number of neurons in the $l^{th}$ layer.
• $w_{lk}^l$: weight referring to the $k^{th}$ neuron of the $l^{th}$ layer which takes as input the output of the $i^{th}$ neuron of the $l-1^{th}$ layer.
• $w_{0k}^l$: bias relative to the $k^{th}$ neuron of the $l^{th}$ layer.
• $s_k^l$: linear combiner output for the $k^{th}$ neuron of the $l^{th}$ layer.
• $x_k^l$: output of the $k^{th}$ neuron of the $l^{th}$ layer
• $\phi(s_k^l)$: Activation Function.

The output $y_k^l$ of a MLP network written in scalar form is defined as:

$$
\begin{cases}
  s_k^l = \sum_{j=0}^{N_{l-1}} w_{kj}^l x_{j}^{l-1} & x_0^{l-1} = 1 \\
  y_k^l = \phi_k^l(s_k^l)
\end{cases}
$$

(2.1)

while for a more compact notation, using vector and matrix form for the operations, the vectors are:

$$
\begin{align*}
  s^l &= [s_1^l, s_2^l, ..., s_{N_l}^l]^T \\
  w_k^l &= [w_{0k}^l, w_{1k}^l, ..., w_{N_l k}^l]^T \\
  \Phi^l &= [\phi_1^l, \phi_2^l, ..., \phi_{N_l}^l]^T \\
  x^{l-1} &= [1, x_1^{l-1}, ..., x_{N_{l-1}}^{l-1}]^T \\
  y^l &= [y_1^l, ..., y_{N_l}^l]^T
\end{align*}
$$

(2.2)

this way, the linear combiner result for the $l^{th}$ layer is:

$$
\begin{align*}
  s^l &= [w_1^{(l)T} x^{l-1}, ..., w_{N_l}^{(l)T} x^{l-1}]^T \\
  &= W^l x^{l-1}
\end{align*}
$$

(2.3)
The weight matrix $W$ is defined as

$$W \in \mathbb{R}^{(N_l \times N_{l-1})} = [w^{(l)}_1 \ldots w^{(l)}_{N_l}]^T = \begin{bmatrix}
w_{00} & w_{01} & \ldots & w_{0N_l-1} \\
w_{10} & w_{11} & \ldots & w_{1N_l-1} \\
\vdots & \vdots & \ddots & \vdots \\
w_{0N_l} & w_{1N_l} & \ldots & w_{N_lN_l-1}
\end{bmatrix} \quad (2.4)$$

Using this notation, the output vector of the $l^{th}$ layer can be written as

$$x^l = [\phi^l_1(w^{(l)}_1 x^{l-1}) \ldots \phi^l_{N_l}(w^{(l)}_{N_l} x^{l-1})]^T = \Phi^l(W^l x^{l-1}). \quad (2.5)$$

and the output of the MLP after $L$ layers is

$$y = x^L = \Phi^L(W^L \Phi^{L-1}(\ldots \Phi^{(2)}(W^{(2)} \Phi^{(1)}(W^{(1)} x^{(0)})))) \quad (2.6)$$
2.2 Activation Functions

![Neuron Diagram](image)

Figure 2.2: Single neuron based on the McCulloch & Pitts model [21]

Taking for reference picture 2.2 that describes the neuron model, an Activation function (AF) is a mapping between an input $s$ and an output $y$. An Activation Function is suitable for Neural Networks if the mapping between the linear combiner output $s$ and the output $y$ is be a non-linearity. There are other desirable properties, but they are not necessary as long as they can be adjusted for gradient computation, like being bounded between a maximum and a minimum output and have a continuous first derivative for gradient computation. An exception in this sense is ReLu [22]. ReLu doesn’t have a maximum and its first derivative is not continuous on 0 However, by taking a subgradient the ReLu is suitable for being used in optimization algorithms.

There are many factors to consider when choosing an activation function, which are based on several aspects of the problem to solve. The difference in performance between AFs, depends on the kind of problem that has to be solved (Classification, Regression...). It depends on how data is normalized, for example between 0 and 1 or between -1 and 1. It can depend on the design of the Neural Network Architecture, like using only ReLU [22] to solve the vanishing gradient problem in a network with many layers, or combining layers with different activation functions.

Some Activation functions used in Neural Networks are given in the following:
2.2.1 Step Function

The Step Function was originally used in the original perceptron model [21]. It is defined as

$$y = \phi(s) = \begin{cases} 
0 & s < 0, \\
1 & s \geq 0. 
\end{cases} \quad (2.7)$$

With this function the possible states are 0 or 1, that represent off or on states. A combination of neurons using such activation functions allows to state if a certain feature is present or not in the data.

2.2.2 Logistic Function

The Logistic Function is a curve defined as

$$y = \phi(s) = \frac{G}{1 + e^{-\alpha s}} \quad (2.8)$$

where the parameter $G$ control the output range, which is from 0 to $G$ and $\alpha$ controls the steepness of the function. A particular form of the Logistic Function is the Sigmoid, where $G = \alpha = 1$

$$y = \phi(s) = \frac{1}{1 + e^{-s}} \quad (2.9)$$

2.9 is widely used in Neural Networks and it is relatively simple to compute and to differentiate. The derivative

$$\phi'(s) = \phi(s)(1 - \phi(s)) \quad (2.10)$$
Another widely used activation function is the Continuous Tan-Sigmoid Function, defined as

\[ \phi(s) = \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} \]  

and its derivative is

\[ \frac{\partial \phi(s)}{\partial s} = 1 - \tanh^2(s) = \text{sech}^2(s) = 1 - \frac{(e^s - e^{-s})^2}{(e^s + e^{-s})^2} \]

(2.12)

2.9 and 2.11 are continuous and suitable for algorithms such as Backpropagation, which is widely used in learning the optimal parameters of a Neural Network. A Neural Network with many hidden neurons where each one of them implements a non-linearity, allows to approximate any function [11]. The level of activation of each neuron is used to compute decisions for classifications tasks or to predict values as in the case of regression tasks. A Neural Network without nonlinearities, is capable of separate only linear separable data, because combining linear combinations results in another linear combination. For example it couldn’t be able to separate points on a plane that are divided by a curve.

### 2.2.3 Rectifier (ReLU)

A unit that implements a Rectifier is called a Rectified Linear Unit (ReLU)[22]. Such units are widely researched now because they carry out some important improvements over Activation functions like Sigmoid or Tanh. ReLU activation is
positive, which is similar to biological neurons. moreover, not every unit will be active and so there is sparse activation. It is easier to design multi layer networks because there is not the vanishing gradient problem and computation is efficient compared to Logistic Activation functions.

\[ \phi(s) = \max(0, s) \quad (2.13) \]

However, this function is not differentiable at 0. It also suffers from the dying ReLU problem, that is, at some point during the training process, the activation of many neurons could become always zero due to large weights. A solution to these problems is the use of Leaky ReLU [23].

2.3 Backpropagation

In order for a Neural Network to learn, several learning methodologies can be used. One of these is the Backpropagation algorithm [24] which is based on gradient descent. The Backpropagation algorithm aims to iteratively find the gradients of an error based cost function and then modify the weights, or parameters, to gradually reduce the function value. Let’s take a training data set composed of \( \mathbf{x}_i \in \mathbb{R}^{N_T \times 1}, \mathbf{d}_i \in \mathbb{R}^{N_T \times 1}, i = 1, 2, ..., N_T \) where \( [\mathbf{x}_i, \mathbf{d}_i]^{N_T} \) are data pairs. The desired output of the network is \( \mathbf{d}_i \) and it is the target for each training input \( \mathbf{x}_i \) while the network output of the \( i^{th} \) training pattern is \( \mathbf{y}_i \). The algorithm minimizes a cost function \( J(\mathbf{w}) \). Various choices of \( J(\mathbf{w}) \) can be selected. Here the mean squared difference between desired and actual network output is used.

\[
J(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N_T} (\mathbf{d}_i - \mathbf{y}_i)^2 = \frac{1}{2} \sum_{i=1}^{N_T} \mathbf{e}_i^2 \quad (2.14)
\]

The algorithm is mainly divided in two parts: the propagation part and the update part. In propagation, output values are computed and the error is back propagated.
through the network. In the Backpropagation step the gradients are calculated. Gradients are then used in the update part, where weights are updated according to the Delta Rule. Considering a weight vector \( \mathbf{w} \), the Delta Rule for the weights vector at iteration \( n \) is

\[
\mathbf{w}_{n+1} = \mathbf{w}_n - \mu \frac{1}{2} \nabla J(\mathbf{w}_n)
\]  

(2.15)

Let’s consider \( w_{lk} \) as the weight of the \( k^{th} \) neuron of the \( l^{th} \) layer. Then, \( \nabla J(w_{lk}) = \frac{\partial J}{\partial w_{lk}} \). This derivative can be expanded in order to retrieve its value by using the derivative chain rule as

\[
\frac{\partial J}{\partial w_{lk}} = \frac{\partial J}{\partial x_k} \frac{\partial x_k}{\partial s_k} \frac{\partial s_k}{\partial w_{lk}}
\]  

(2.16)

Then, considering that \( s_k \) is the result of the weighted sum of the inputs \( \mathbf{x}^l \) as in 2.1, the derivative becomes simply

\[
\frac{\partial s_k}{\partial w_{lk}} = x_k
\]  

(2.17)

Defining the term \( \delta^l_k \), which is the error term for the \( k^{th} \) neuron of the \( l^{th} \) layer and indicating as

\[
\delta_k = \frac{\partial J}{\partial x_k} \frac{\partial x_k}{\partial s_k}
\]  

(2.18)

the derivative of the activation function with respect to the linear combiner result is

\[
\frac{\partial x_k}{\partial s_k} = \phi'(s_k)
\]  

(2.19)

where \( \phi'(.) \) is the derivative of the activation function chosen. For example it can be the derivative of the Sigmoid 2.10. The last term to calculate for 2.18 is the
derivative of the error cost function $J$ with respect to the input $x_k$:

$$\frac{\partial J}{\partial x_k} = (d_{ik} - y_{ik})$$  \hspace{1cm} (2.20)

where the index $i$ stands for the $i^{th}$ training sample. For further details on the Algorithm and its properties see [24].

Using the previous results, a generalized and compact Vector and Matrix notation for the Backpropagation Algorithm is presented here. Using the notation introduced before, rewriting 2.5 and 2.6 in a more algorithmic way, the forward phase of a Neural Network can be written as

**Algorithm 1** Forward Pass

for Every input pattern do  
\[ x^1 \equiv y_0 \]  
for \( l = 1 \) to \( L \) do  
\[ \begin{cases} s^l = W^{(l)} x^l & x \equiv y^{l-1} \\ y^l = \phi^l(s^l) \end{cases} \]

And the recursive backward pass using Backpropagation is

**Algorithm 2** Backward Pass

\[ e = d - y^L \]  
for \( l = L \) to \( 1 \) do  
\[ \begin{array}{c} \delta^l = e \odot \phi'(s^l) \\ e = W^l \delta^{l+1} \\ W^l = W^l + \mu \delta^l x^{l-1} \end{array} \]

Where $\odot$ is the Hadamard product, $x$ is the input data in the case of $l = 0$, or the $l^{th}$ layer output. $x^l[0] = 1$, which stands for the bias term and $y$ is the network output vector, or $x^L$.

### 2.4 Autoencoder

An Autoencoder is a particular form of Multilayer Perceptron that is useful to encode, or learn, features from input data. The Autoencoder aims to learn to reconstruct the input data, by first encoding it in a smaller (or bigger) dimensionality and then attempt to reconstruct it in order to learn suitable encodings. The
difference from a traditional Neural Network is that the number of output is the same as the input, and the objective function tries to minimize the input data. Consider a $N - L - N$ Autoencoder with $N$ input and output neurons and $L$ hidden nodes, $W$ as the weight matrix, in particular, $W^{(1)} \in \mathbb{R}^{N \times L+1}$ the input-hidden weights and $W^{(2)} \in \mathbb{R}^{L+1 \times N}$ the hidden output weights. The input data is $[x_k]_{1}^{N_T}$ where $x_k \in \mathbb{R}^{N \times 1}$. The function to minimize is the distance between the input as reference data and the network output. For example a quadratic cost function of the form

$$J(W) = \frac{1}{N_T} \sum_{k=1}^{N_T} ||x_k - f(W, x_k)||_2^2$$

(2.21)

can be used, where $f(W, x_k)$ is the function which represent the Neural Network. To find suitable weights it is possible to train the network using the Backpropagation algorithm as in traditional ones.

**Figure 2.4:** Schematic representation of an Autoencoder. The circles are the neurons, which are in the same number in the input and in the output layer, as the goal of an Autoencoder is to reconstruct the input data.

In order to discover more general structures in input data and reduce overfitting, several methods can be implemented. One of the techniques used is to impose sparsity on the hidden nodes, for example by using a bigger number of hidden nodes than the input nodes, or adding a sparsity constraint in the objective function such as the Kullback-Leibler (KL) distance [25]. Another technique is to use Denoising Autoencoders, which try to reconstruct the input data starting from corrupted
ones. This allows to learn more generalized representations, especially in higher level representation when the Denoising Autoencoders are stacked [26]. In this work, an Autoencoder implementing a Spline Activation function is presented. It is then trained in a “Deep” fashion, using greedy layer-wise pre-training stacking the layers to form a Deep Neural Network.
Chapter 3

Spline Activation Function

3.1 Spline interpolation scheme

The goal of this section, is to describe an activation function that is suitable for a neural network. This material is taken from [27] which is a good starting point to read about Nonlinear Spline Filtering. The Activation Function must have the features of Nonlinear Activation Functions (as seen in section 2.2). It should also allow to locally modify its shape adapting a few number of parameters. Most of the functions used in the literature, lack of local adaptability property. Functions like Polynomial Activation functions or adaptable Sigmoid, have the property that adapting a point in a part of the curve can change the whole shape of the function. This can lead to the cancellation of the learned features because of the global adaptation property. Moreover, this causes instability in the learning process of the parameters. To adapt parameters such that changes in the shape of the AF are made, a LUT table is used. In this table, the control points are stored as table entries. In order to reduce the amount of free parameters, typical of LUT activation functions, a suitable interpolation scheme has to be chosen. The scheme should retain the continuity of the derivative otherwise it would cause instability in the learning algorithm.

Splines comes from elastic rulers that are used for technical drawing. They cross a certain number of points thus allowing to form different shapes. Splines are polynomial curves which are differentiable up to a certain order and they allow to interpolate $Q + 1$ knots inside a convex-hull. Knots are taken constrained in the
Spline Activation Function

order $q_{x,0} < q_{x,1} < \ldots < q_{x,Q}$ and they represent a tabulated function on the x-y plane.

Splines considered here are the univariate basis or blending functions developed by Schoenberg, which come from Bézier splines [28]. Given $u \in [q_{x,i}, q_{x,i+1}] \in R^+$ which is the abscissa between two consecutive knots, the Spline function is calculated as an affine combination of a $P$ degree spline basis function $N_i^P(u)$ [29] and some knots.

$$\phi(u) = \sum_{i=0}^{Q-P-1} Q_i N_i^P(u), u \in [q_{x,n}, q_{x,Q-P}]. \quad (3.1)$$

$N_i^P(u)$ is given by the following Cox-de Boor recursion[30]

$$N_i^0 = \begin{cases} 1, & q_{x,i} \leq u < q_{x,i+1}, \\ 0, & \text{otherwise,} \end{cases} \quad (3.2)$$

$$N_i^P(u) = \frac{u - q_{x,i}}{q_{x,i+1} - q_{x,i}} N_i^{P-1}(u) + \frac{q_{x,i+P+1} - u}{q_{x,i+P+1} - q_{x,i+1}} N_{i+1}^{P-1}(u).$$

3.2 is a linear interpolation scheme and, being $N_i^0(u)$ a rectangular function, $N_i^P(u)$ is a sequential convolution of $P$ rectangular pulse functions $N_i^P = N_i^0 \ast \ldots \ast N_i^0$. The curve 3.1 is then described as

$$\phi(u) = \phi_i(u), \quad \forall i = 0, \ldots, Q \text{ and } u \in [0,1). \quad (3.3)$$

where $\phi_i(u)$ is the i-th local polynomial curve span defined over the $P+1$ knots, or control points. The useful property of the vector containing all the control points is that scaling or modifying its dimension does not change the basis function, which can be defined before. Taking the approach in [31], a uniform sampling step of the abscissa $\Delta x = q_{x,i} - q_{x,i+1}$ allows to pre compute the recursion 3.2 up to the $P$-th level of the polynomial. The local polynomial $\phi_i(u)$ can be defined as

$$\phi_i(u) = u^T C_i. \quad (3.4)$$
Spline Activation Function

Where \( C \in R^{(P+1) \times (P+1)} \) is pre-computed and is called spline basis matrix. \( u \) is a vector defined as \( u \in R^{(P+1) \times 1} = [u^P \ u^{P-1} \ ... \ u \ 1]^T \), and \( u \), as before, is the normalized abscissa value between two knots, or control points \( \in Q \). The vector \( q \) is defined as \( q \in R^{(P+1) \times 1} = [q_i \ q_{i+1} \ ... \ q_{i+P}]^T \). A result that is useful for designing LMS-like algorithm is the derivative of the polynomial \( \phi_i(u) \). The derivative of 3.4 is

\[
\frac{\partial \phi_i(u)}{\partial u} = \phi'_i(u) = u^T C q_i
\]

with \( u \in R^{(P+1) \times 1} = [P u^{P-1} \ (P-1) u^{P-2} \ ... \ 1 \ 0]^T \). In order to derive the algorithm for computing the mapping of \( u \), an example for \( P=2 \) and \( P=3 \) is presented. For \( P = 2 \) the recursion 3.2 becomes

\[
N^2_i(u) = \begin{cases} 
\frac{1}{2} (u - q_i), & q_{x,i} \leq u < q_{x,i+1}, \\
\frac{1}{2} - (u - q_{i+1}) - (u - q_{i+1})^2, & q_{x,i+1} \leq u < q_{x,i+2}, \\
\frac{1}{2} [1 - (u - q_{i+2})]^2, & q_{x,i+2} \leq u < q_{x,i+3},
\end{cases}
\]

the system reformulated in matrix form 3.4 becomes

\[
\phi_i(u) = [u^2 \ u^1 \ 1] \frac{1}{2} \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} q_i \\ q_{i+1} \\ q_{i+2} \end{bmatrix},
\]

with \( i = 0, 1, ..., Q \) and \( u \in [0, 1) \). With \( P=3 \) the matrix form of 3.2 is

\[
\phi_i(u) = [u^3 \ u^2 \ u^1] C \begin{bmatrix} q_i \\ q_{i+1} \\ q_{i+2} \\ q_{i+3} \end{bmatrix},
\]

and the basis matrix \( C \) is

\[
C = \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}.
\]

By using different interpolation constraints, other spline basis that have different properties can be used. A different basis which has a more local approximation
than B-Spline is the Catmull-Rom spline [34]. The interpolation scheme is the same as exposed before, but the spline basis matrix $C$ is

$$
C = \frac{1}{2} \begin{bmatrix}
-1 & 3 & -3 & 1 \\
2 & -5 & 4 & -1 \\
-1 & 0 & 1 & 0 \\
0 & 2 & 0 & 0
\end{bmatrix}.
$$

CR-Splines constraint the curve to pass through all the control points, while B-Spline does not necessarily do it. This result in more local approximation. The first derivative is continuous for B-Splines and CR-Splines while the second derivative is continuous only for B-Splines. However, there is a regularization property which is common in splines (CR- and B-Splines included), called variational diminishing property [30] that ensures no unwanted oscillations between knots and exact linear segments.

### 3.2 Spline Generalized Sigmoidal Neuron

The property of the Spline interpolation scheme described in 3.4 are suitable for the definition of a neuron with a flexible activation function. The Spline Generalized
Sigmoidal Neuron was first introduced in [32] and it can be described as a LUT followed by a spline interpolator. Figure 3.1 shows the spline approximation and for further details see [27].

Defining the output of the neuron’s linear combiner as

\[ s = w^T x \]
\[ y = \phi(s; q) \tag{3.6} \]

where \( x \) is an input vector \( x \in \mathbb{R}^D \) and \( w \in \mathbb{R}^D \) the weight vector. The Bias term is added as an additional input feature.

Using an equidistant spacing between control points, centered around the x-axis origin and not adapting them, an efficient computation of the span index \( i \) and the local \( u \) needed to compute 3.4 can be derived. Given the iteration index \( n \), the output of the neuron \( y \) and the result of the linear combiner \( n \), \( s = w^T x \)

\[ u = \frac{s}{\Delta x} - \left\lfloor \frac{s}{\Delta x} \right\rfloor, \tag{3.7} \]
\[ i = \left\lfloor \frac{s}{\Delta x} \right\rfloor + \frac{Q - 1}{2}, \]

where \( \Delta x \) is the spacing between control points and \( Q \) is the size of the control points vector or the number of knots. \( \lfloor . \rfloor \) is the floor operator. To compute 3.4 the vector \( u \) is constructed as \( u = [u^3 \ u^2 \ u \ 1] \) and the index \( i \) is used to extract the span \( q_i = [q_i \ q_{i+1} \ldots \ q_{i+p}] \) where \( q_i \) is the \( i \)-th control point. The computation of 3.4 is done efficiently, in fact, it involves the computation of small vector-matrix multiplications. The derivative of \( \phi(s; q) \) with respect to \( s \) can be easily computed and is

\[ \frac{\partial \phi(s)}{\partial s} = \phi'(s) = \frac{\partial \phi(s)}{\partial u} \cdot \frac{\partial u}{\partial s} = \left( \frac{1}{\Delta x} \right) \dot{u} B q_i, \tag{3.8} \]

and the vector \( \dot{u} \) is \( \dot{u} = \frac{\partial u}{\partial u} = [Pu^{p-1} \ (P-1)u^{p-2} \ldots \ 1 \ 0]^T \). Then the derivative of the output \( y \) with respect to \( w \) is

\[ \frac{\partial \phi(s)}{\partial w} = \phi'(s) \cdot \frac{\partial s}{\partial w} = \phi'(s)x \tag{3.9} \]
while the derivative of $y$ with respect to the control points $q$ is

$$\frac{\partial \phi(s)}{\partial q} = B^T v.$$

and the derivative of $\phi(s)$ with respect to any $q_k$ that is outside the current span $q_i$ is 0.
Chapter 4

Spline Autoencoder

4.1 Definition

The proposed solution is an Autoencoder that using a Spline Activation function aims to learn better data representation than Autoencoders that implement Sigmoidal Activation Functions. This is motivated by the fact that Activation Functions in Neural Networks are not chosen using effective criteria, but they are chosen based on trial and error. Learning activation functions that are dependent on how data is shaped and distributed and that are independent for each neuron, is the goal of Spline Activation functions. As investigated in [1] lets define a single hidden layer \( N - H - N \) Autoencoder with \( N \) input and output nodes and \( H \) hidden neurons. Hidden and output layers implement a Spline Activation Function while the input layer is linear.

Referring to the Autoencoder at chapter 2.4 the Autoencoder can be written as a function of the input \( x \), where the weight matrix is \( W \), and the control points vector is \( q \):

\[
\begin{align*}
  h_i &= \phi(w^h_{hi} x; q_{hi}) \\
  f_i(x) &= y_i = \phi(w^y_{yi} h; q_{yi})
\end{align*}
\]

(4.1)

where the index \( i \) stands for the \( i^{th} \) neuron and the index \( h \) refers to the connections between the input and hidden layer, while the index \( y \) refers to the connections between the hidden layer and the output layer. Concatenating the intermediate
result, \( h_i \), in a vector \( \mathbf{h} = [h_1, ..., h_H, 1]^T \) the \( i \)th output of the network with \( i = 1, ..., N \) outputs is given by \( f_i(\mathbf{x}) \). The function \( \phi(\cdot) \) is given by the Spline Activation function 3.6. The derivatives with respect to the weights are similar to the ones used in Backpropagation algorithm except for the derivative of the Activation Function which can be calculated as 3.8. Referring to the delta rule of the Backpropagation algorithm in 2.3 the \( \delta \) component for the \( iy \)th neuron with respect to the \( j \)th connection is

\[
\delta_{y_j} = e\phi'(s_{y_j}) \tag{4.2}
\]

for the deltas of the output layer and

\[
\delta_h = \delta_{y_j} \phi'(s_{h_j}) \tag{4.3}
\]

for the deltas of the hidden layer. This way, the update rule for the weights is similar to the Backpropagation algorithm 2.3 where the deltas used here are substituted in the standard one.

The control points of the last layer are adapted according to the following rule, that is derived from the Delta Rule and is used in the Backpropagation Algorithm.

\[
\mathbf{q}_y = \mathbf{q}_y + e\mathbf{B}^T u_{y_i} \tag{4.4}
\]

Where the term \( e \) is the derivative of the error cost function given the inputs and the subscript \( y \), again, is referring to the control points of the neurons the output layer. The control points for the hidden layer are adapted following the Backpropagation rule as

\[
\mathbf{q}_h = \mathbf{q}_h + \phi'(s_{y_i}) \mathbf{w}[j]_{h} \mathbf{B}^T u_{h_i} \tag{4.5}
\]

The subscript \( h \), again, is referring to the control points of the neurons the hidden layer and the index \( j \) on the weight vector \( \mathbf{w} \) stands for the \( j \)th element of the vector.
4.2 Training

In order to achieve an efficient training, a suitable cost function has to be defined. Following the work on [1] a cost function of the form

\[ J(w, q) = \frac{1}{B \times N} \sum_{i=1}^{B} L(d_i, f(x)) + \lambda_w R_w(w) + \lambda_q R_q(q) \]  

(4.6)

where \( W \) is the weights matrix, \( q \) the control points vector and \( q_0 \) the initialized control points vector. \( d \in R^{B \times N} \) is the desired output vector where \( B \) is the batch dimension and \( N \) is the number of input neurons, used for better balance of the three components. For the error term \( L \), similarly to [1], a squared error term \( L(d_i, f(x)) = \|f(x_i) - d_i\|_2^2 \). This cost function is a measure of the reconstruction error, which means how different are the outputs from the input data that are reconstructed.

For the regularization terms, an \( L2 \) norm on the weight vector is used, which is

\[ R_w(w) = \frac{1}{2N_W} \|W\|_2^2. \]  

(4.7)

The term \( N_W \) is the dimension of the weights array, which is used to get a better balance in the three components of the cost function. For the regularization of the control points, traditional approaches used in Neural Networks, (such as the one used for the weights) are not suitable for the control points. The idea is that, given the fact that the Spline is sampled from values that are already optimum, like a Sigmoid or Tanh, instead of penalize unwanted deviations from very small initial values in the case of weights, a different approach should be used. As investigated in [1], let’s denote \( q_0 \) as the control points vector, initialized sampling the Spline from an activation function in a given range \((r, R)\). Denoting also \( q \) as the points vector that are adapted in the algorithm, a regularization term that is a damping parameters can be defined. The term has the form:

\[ R_q(q) = \frac{1}{2N_q} \|q - q_0\|_2^2. \]  

(4.8)
in this way, using the damping factor, unwanted oscillations in the Spline shape can be prevented. In fact, high values for $\lambda_q$ have the effect on limiting or essentially disabling the adaptation of the control points, while small values lead to high overfitting or bad results. The term $N_q$ is the dimension of the control points array of all the neurons, which helps to scale the magnitude of the term with respect to the others. This choice has been made in order to make the error cost function more sensible to the variations of the hyperparameters. As concluded in [1] this damping criterion yields to good results, although new form of regularization can be designed. Another way to regularize the adaptation of the control points is to define a suitable $\Delta x$, which is the sampling step of the points on the Spline. Generally, but it may depend on the Dataset, different $\Delta x$ values lead to more overfitting in training data in the case of values less than 0.2 and reduced overfitting for values bigger than 0.2. Note however that using very big sampling steps, can be seen as against the Spline Adaptation itself.
Chapter 5

Implementation and Results

5.1 Implementation

The Autoencoder is implemented using Python Language [39]. It is a very popular language used in many scientific projects and with many available libraries and datasets that are free to use. In this implementation, a combination of NumPy [40] and Scikit [41] libraries is used in order to avoid rewriting very basic mathematical operations, such as dot product of arrays, downloading and scaling datasets and computing scores on classification and errors. Moreover, the libraries implement optimized and fast functions which are a must when writing Neural Network scripts. Due to the novelty of Spline Activation Functions, a gpu implementation for Spline Neural Network is not available in well known libraries as Theano or Google Tensor Flow. For this reason, some classes implementing a Neural Network with a Spline Activation Function have been implemented from scratch. However, a modern quad core laptop gives very long training times, in term of days, for finding the correct parameters for medium sized datasets. The datasets tested in this work are to be considered rather small, and much more improvement of the results can be obtained with a bigger number of examples. Still, the results obtained are a good indicator of the capabilities of the proposed solution. To better understand the results, a comparison with a Standard Autoencoder trained with similar condition is proposed. The code can be found at https://bitbucket.org/alberto_marinelli/asnn/src/a583736e69189ebae3e9f18a022b46817d10d8e5?at=master The properties of the Spline Interpolation scheme, allow to modify only a subset of the spline control
Implementation and Results

points, for instance, using a cubic Spline, only 4 control points, multiplied by the batch size, are updated at each epoch. Moreover, by reusing already computed $I$ and $U$ values (see 3.2) from the forward step, the backward pass of the update step involves only small array operations. Additionally, using vectorization for all the operations, it is possible to get suitable computational times with small datasets. In fact, the Spline neurons do not depend on each other and they allow to be vectorized, or stacked in matrix form, to perform computation efficiently. This implementation allows to get training times that are around 2 or 3 times the training times of a standard neural network with Sigmoid activation function.

The algorithm used to update the weights and the control points of the Autoencoder is Adagrad [35]. It is chosen over a fixed learning rate for its better learning speed over other solutions such as momentum [38]. Adagrad, also called Adaptive Gradient Algorithm, adapts the learning rate of the parameters following a criterion that allows to perform big updates for infrequent components and small updates for frequent ones. As result, due to its adaptation properties, it allows to reach good solutions without cumbersome tuning of hyperparameters. Setting $\nabla J(w_i) = g_{n,i}$ as the gradient of the objective function with respect to the parameter $w_i$ at a time step $n$, where $i$ is the parameter index, the Stochastic Gradient Descent rule is $w_{(n+1),i} = w_{n,i} - \mu g_{n,i}$. Using Adagrad, the learning rate $\mu$ is adapted for each parameter $w_i$ depending on the sum of the squares of the past gradients.

$$w_{(n+1),i} = w_{n,i} - \frac{\alpha}{\sqrt{G_{n,ii} + \epsilon}} \cdot g_{n,i}$$  \hspace{1cm} (5.1)

$G_{n,ii}$ is the sum of the squares of the past gradients $G_{n,ii} = G_{n,ii} + g_{n,i}^2$ accumulated up to time step $n$ and $\epsilon$ is a very small value in the order of $10^{-8}$ used to prevent division by 0. Efficiency of Adagrad can be seen in the following result, where a Spline Autoencoder is trained on the Chemical Dataset for 1000 epochs using a fixed learning rate of 0.1 and 0.01 and Adagrad algorithm with $\alpha$ parameter of 0.01.

Figure 5.1 clearly shows that Adagrad is a much more suitable solution for training Networks, with the advantage of a faster convergence rate, and a less cumbersome tuning of parameters. In fact, changes in the $\alpha$ parameter value, does not sensibly hurt the convergence, and it is enough to train the network for two or three values of $\alpha$, pick the one that converges better and leave it as it is [35]. While
for stochastic gradient descent many runs on different values are needed to tune the parameter. Other learning variants are Momentum and Adam, but they are not used here since they require the tuning of more parameters and can be useful to research state of the art results and very well optimized solutions which are heavily dependent on the dataset used. For a comparison purpose of two Network Architectures, Adagrad is a good trade off between quality of learning and complexity.

5.2 Results

Here are presented some results using different datasets. The comparison is made in term of Root Mean Squared Error between an Autoencoder that uses Spline Activation Functions and a 'Standard' Autoencoder that uses a Tanh Activation
Function. The cost function for the 'Standard' Autoencoder is similar, for comparison purpose, to the one used for the Spline Autoencoder. Indicating again with $N$ as the number of input neurons and $N_w$ the number of weights,

$$J(w) = \frac{1}{N \times B} \sum_{i=1}^{B} L(d_i, f(x)) + \frac{\lambda_w}{N_w} R_w(w). \quad (5.2)$$

While the root mean squared error formula, where $f(x_i)$ is the predicted value, $d_i$ is the target value and $B$ is the mini batch size, or the number of samples is:

$$RMSE = \sqrt{\frac{1}{B} \sum_{i=1}^{B} (f(x_i) - d_i)^2} \quad (5.3)$$

Again, $R_w(w) = ||w||^2$. The sum of the error term $L(.)$ is divided again by $N \times B$ and the weights regularization term is divided by $N_w$ for a comparison of the error with the other solution. Training is done again using mini batches approach. A first important result is the comparison of the two Autoencoder networks on the "Chemical Dataset", which is a dataset available in MATLAB dataset collection for function fitting. Its dimensionality is $498 \times 8$ and an Autoencoder and a Spline Autoencoder with $H = 15$ hidden neurons and 8 input and outputs has been trained. The search for parameters is done using 3-fold cross validation and a grid search for the lambdas and the $\Delta x$ (the spacing between knots of the Spline Activation function). The lambda parameters for the weights are chosen between a selection of powers of $10^{-j}, j \in 1, ... 5$ and the $\Delta x$ over some values between $[0.05, ... 0.8]$ with a stepsize of 0.1. After the correct choice of parameters the model was trained again on the whole dataset and tested on a dataset portion it has never seen. The results are the result of an average of 25 runs for each Autoencoder. A table with the choice of parameters and the error result is presented, where "Standard AE" refers to the Autoencoder with Tanh Activation function that minimizes 5.2 and Spline AE refers to the Spline Autoencoder that minimizes 4.6. The mini batch dimension varies for every dataset used. In this case a dimension of 80 was used. All the experiments were launched with the random seed of the numpy library at value 1.
TABLE 5.1: Results for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Chemical Dataset.

<table>
<thead>
<tr>
<th></th>
<th>Train RMSE</th>
<th>Test RMSE</th>
<th>(\lambda_w)</th>
<th>(\lambda_q)</th>
<th>(\Delta x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard AE</td>
<td>0.025 ±0.002</td>
<td>0.02700 ±0.002</td>
<td>0.001</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spline AE</td>
<td>0.0055 ±0.003</td>
<td>0.0094 ±0.003</td>
<td>0.0001</td>
<td>0.0001</td>
<td>0.3</td>
</tr>
</tbody>
</table>

Figure 5.2 shows the plot of the evolution of the test RMSE sampled in 5000 epochs, where the mini batch training dimension was 80 for a dataset of dimension 400 with 98 test examples. Backpropagation is used for both the network weights and control points. The update is done using Adagrad [35] algorithm with a value for the \(\alpha\) parameter of 0.01 for both the weight and control point updates.

Figure 5.3 shows a run on the Chemical dataset, for both the Autoencoder and the Spline Autoencoder values over 12500 weights updates of the objective function. Note how the noise in the function values are different while the same batch size and same scaling of cost function was used.

From figures 5.2 and 5.3 it appears that the Spline approach takes less iterations to reach a good and better minimum. Moreover, as table 5.1 shows, there is some room for improvement in order to reduce overfitting, as found for single hidden layer neural networks in [1], by changing the regularization term for the control points and combining it with a different \(\Delta x\).
Implementation and Results

Figure 5.3: Comparison of the objective function values for the Spline Autoencoder and the Standard autoencoders implementations

Table 5.2: Values for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Digits Dataset.

<table>
<thead>
<tr>
<th></th>
<th>Train RMSE</th>
<th>Test RMSE</th>
<th>$\lambda_w$</th>
<th>$\lambda_q$</th>
<th>$\Delta x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard AE</td>
<td>0.0430 ± 0.001</td>
<td>0.0496 ± 0.0001</td>
<td>0.001</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spline AE</td>
<td>0.0113 ± 0.003</td>
<td>0.0175 ± 0.003</td>
<td>0.01</td>
<td>0.01</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Another tested dataset is the “Digits” dataset available in Scikit-learn. A train set size of 1500 is used while a set of 297 samples is tested. Its dimensionality is 64. Here are presented some results of the reconstruction error as always in term of RMSE. It is run using 100 hidden neurons for 2500 epochs with a batch dimension of 100 for both Autoencoders. The idea is to let it run for many hidden neurons in order to obtain sparse reconstructions that can be useful for feature extraction. Table 5.2 is the comparison between the novel solution and a standard one.

Finally, a last test is presented, using the Calhousing Dataset, again freely available from the MATLAB regression datasets. This dataset contains 20640 samples with 8 features that are used for regression task. However, here only the reconstruction capabilities of Spline Autoencoder are taken into account.
Table 5.3: Values for training and testing of a Standard Autoencoder and a Spline Autoencoder on the Calhousing Dataset.

<table>
<thead>
<tr>
<th></th>
<th>Train RMSE</th>
<th>Test RMSE</th>
<th>$\lambda_w$</th>
<th>$\lambda_q$</th>
<th>$\Delta x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard AE</td>
<td>0.0346 ± 0.002</td>
<td>0.0349 ± 0.002</td>
<td>0.00001</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Spline AE</td>
<td>0.0058 ± 0.003</td>
<td>0.0065 ± 0.003</td>
<td>0.001</td>
<td>0.0001</td>
<td>0.3</td>
</tr>
</tbody>
</table>

The dataset was split using 15000 training examples and the remaining as testing examples with a batch size of 100 elements. The number of epochs was 1000 and 15 hidden neurons were used for both Autoencoders.

In addition, the sampling step $\Delta x$ of the Spline Neurons, act as a regularizer on the Network results, with a higher value that is able to reduce overfitting. As figure 5.4 shows, the choice for the $\Delta x$ parameter in the Chemical dataset was chosen according to the result in figure 5.4. Which was a sampling over some values, while bigger values, even though accomplish some good results, would result against the Spline Interpolation theory [1].

![Figure 5.4: Comparison of the test errors, training errors and $\Delta x$ values over 9 different values. Dataset used for the run is Chemical Dataset.](image)

The Spline plots of figure 5.5 indicate the regularized shape that is used to obtain the results on the Digits dataset. The plots, taken from some sample neurons,
show how different the obtained adaptation on different neurons is. It can be noted that sometimes only a region of the activation function is adapted, while, for the example C, a region of the spline becomes even linear.

\begin{figure}[h]
\centering
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{plot_a}
\end{subfigure}
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{plot_b}
\end{subfigure}
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{plot_c}
\end{subfigure}
\begin{subfigure}{0.4\textwidth}
\includegraphics[width=\textwidth]{plot_d}
\end{subfigure}
\caption{Plots of samples of the Flexible Spline Activation Function taken from neurons used for the digits dataset test.}
\end{figure}

A last plot, in figure 5.6 of a sampling of 2 Splines, using a very low regularization value for the control points (0.000001) and a low spacing $\Delta X$, show a trained network prone to overfitting and with very noisy activation values. It may reach some good reconstruction RMSE but it is an overfitted solution, that prevents to reach good generalization capabilities.

\section{5.3 Deep Neural Network Discussion}

Here a preliminary work on training Deep Neural Networks using Spline Activation Functions is presented. Training Deep Network is hard, because it is an architecture suitable for big datasets, while with small ones results can be very similar to single hidden layer networks. In fact, they began to become popular when bigger datasets and more computational power was available. The results presented here are the result of a greedy layer-wise pre training of Spline Autoencoders. Each
Autoencoder is trained singularly and the hidden layer encoded values are used as input to train the second Autoencoder. The Hidden Layer values of the second Autoencoder, which should represent higher level and more general data representation, are then fed as input to a Softmax\cite{37} layer, using multinomial logistic regression, through the scikit-learn library Logistic Regression. Figure 5.7 shows a schematic representation of the technique.

For instance, defining as \(X\) the input data, \(H_1\) the hidden layer and \(\tilde{X}\) the reconstructed input, one should learn suitable encodings \(H_1\) by finding weights \(W_1\) that transform the input data \(X\) into \(H_1\). The encodings are then used as input to a second Autoencoder, that this time tries to minimize \(\tilde{H}_1\) from \(H_1\) through the weights \(W_2\) that produce the encodings \(H_2\). For this architecture, only 2 layers are used, otherwise one can continue producing more hidden layers \(H_n\) that can be used for a n-layer Deep Neural Network. The parameters used are as follow: for the first layer of the Deep Spline Neural Network 100 hidden neurons are used, while \(\lambda_w = 0.05\), \(\lambda_q = 0.1\) and \(\Delta x = 0.6\). For the second layer 64 hidden neurons are used, while \(\lambda_w = 0.01\), \(\lambda_q = 0.01\) and \(\Delta x = 0.6\). Similarly for the Stacked Autoencoders, 100 and 64 hidden neurons are used, while a \(\lambda = 0.0001\) for the first and \(\lambda = 0.001\) for the second layer are used. The training process is repeated for 15 runs and the best runs are reported in table 5.4 as classifier scores.

As it can be seen from the data, there is no apparent improvement on the Network performance on classification task. However, it should be noted that getting the same results of a Standard solution in such preliminary results is a good sign.
How to train a Deep Neural Network based on greedy layer-wise pre training of Autoencoders. First the Autoencoders are trained singularly, then their encoding weights are used from $X$ to $H_1$ and from $H_1$ to $H_2$ to be used in classification or regression tasks that take part in the last layer, which in this case is a neuron that can perform binary classification or regression.

**Table 5.4:** Best classification scores (in terms of accuracy) obtained over 15 runs for training and testing of a Standard Stacked Deep Neural Network and a Spline Deep Neural Network on the Digits dataset.

<table>
<thead>
<tr>
<th></th>
<th>Train Accuracy</th>
<th>Test Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard AE</td>
<td>0.991</td>
<td>0.915</td>
</tr>
<tr>
<td>Spline AE</td>
<td>0.99</td>
<td>0.915</td>
</tr>
</tbody>
</table>

on the learning capabilities of Spline Activation Functions applied to Deep Networks. In fact, as shown in previous results, the single Autoencoder is able to reach much better testing results than a Standard one, while still getting some overfitting on the data. Trying to reduce overfitting and optimizing the Neural Network architecture can be a future work. Moreover, it is well known [42] that Deep Architectures can be much more difficult to train than some non-Deep ones. In fact, as discussed in [42] stacking pre-trained Autoencoders can be a good initialization method, but in the same way it can get to local minimum that are different from good minimum. This effect has been hypothesized as a result of using much more nonlinearities than a single hidden layer neural network that can
lead to more and different minimums, therefore making gradient descent approach more difficult to apply.

Some of the most famous techniques used to pre train Deep Architectures, that apply to relatively (to the one used here) big datasets, are stacked Denoising Autoencoders, Dropout, data augmentation, fine tuning and also a combination of these methods.

Denoising Autoencoders [26] are similar to Autoencoders, but instead of taking as input the raw data, some noise is applied to the input data and the original data is then reconstructed. Defining as $X$ the input data and $\tilde{X}$ the noisy data, the Autoencoder will try to reconstruct $X$ from $\tilde{X}$ applying the function $\Phi(\tilde{X}) = X$. This methods acts in a similar way to a regularizer, but it was proven to generate more meaningful representation of the inputs that gave better results in classification and regression tasks. Dropout [43] is a form of regularization that tries to reduce overfitting and get better learning outcomes by reducing the co-adaptation of neurons in big networks. It uses a different random mask for each iteration that randomly disables connections between neurons in subsequent layers, making information from other neurons unreliable and preventing their co-adaptation and overfitting. Moreover, since usually the number of dropped connections is in the order of 50%-70% it can result in a faster network. Data augmentation techniques are simply artificial ways to increase the available dataset, adding random noise to the data or applying random projections. Since Deep architectures learn much better when there is a big amount of data, data augmentation is a suitable solution to train Deep Networks. Last method discussed, and also one of the most important that now are used in every state of the art implementation is Fine Tuning of parameters. Stacking pre-trained Deep Autoencoders can lead to better results for classification and regression tasks. However, it is easy to conclude that fine tuning the whole stack of learned features using Backpropagation, can lead to even better results.

In conclusion, Deep Learning as it is today, can be seen more as an art than a science, where to obtain better results one cannot simply rely on mathematical theories but rather on intuition and a large number of tests, often helped by powerful libraries that allow to easily get more insights on the learning process and on the learned features.
Chapter 6

Conclusion

An Autoencoder that uses Flexible Spline Activation function has been presented. With this method, each neuron is able to learn activation functions from data. This way the Autoencoder has been shown to be able to reconstruct the input data better than an Autoencoder implementing a Tanh Activation function. Problems with the Flexible Spline Activation functions, are that without a proper regularization for the adaptation of the control points, the solution tend to overfit. With the implemented regularization term, overfitting is prevented, although the shape of the Activation function result constrained. In this way, new methodologies for regularization can prevent this behaviour. The Autoencoder developed in this work was then used as building block for a Deep Neural Network with two hidden layers and compared to a similar Deep Architecture that implements Tanh activation functions. Results obtained here are similar, but the expressive power of Flexible Spline Activation functions applied to Deep Networks need to be explored more in detail, using common regularization techniques often found in state of the art Deep Neural Networks implementations and enforcing sparsity constraints as Dropout, Sparse Activations or Denoising training. Finally, the work presented here has shown how to train an Autoencoder with Flexible Spline Activation functions, get better reconstruction errors comparing different techniques and how it can be used to lead to better Deep Neural Networks architectures.
Acknowledgements

First of all I would like to thank my Advisor, Aurelio Uncini and Co-Advisor, Simone Scardapane, who have been supportive and patient over these months of work. Their advice was crucial for the development of this thesis.

Thanks to my good friends from University Andriy, Daniele, Ennio, Gabriele, Lorenzo and Riccardo. Without you and the countless hours spent studying and working together I would have never been where I am now.

I would also like to offer my special thanks to my long time friends Daniele, Gianmarco, Giorgio, Federico, Marco and Riccardo who I know are always there in case of need.

Thanks to all the friends and travel companions I made in the student association BEST Rome. I cannot thank you all here, but I will in person. I will write here the first people I met that convinced me to invest time in this association. Thanks to Gabriele, Gianluca, Giuseppe, Francesca L, Francesca M, Fabrizio, Ludovico, Francesca Z and Federico. Thanks to the people I worked with for many projects in countless teams. You are now my friends and you inspired me to find my path.

Finally, thanks to my family who has always been supportive throughout these years of studies.
Bibliography


[22] Vinod Nair and Geoffrey Hinton Rectified linear units improve restricted Boltzmann machines 2010, ICML.

[23] Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng, Rectifier Nonlinearities Improve Neural Network Acoustic Models 2014


