Universidad Autónoma de Madrid
Escuela Politécnica Superior - Departamento de Ingeniería Informática
Facultad de Ciencias - Departamento de Matemáticas

Deep Neural Networks

Master's thesis presented to apply for the
Master in Computer Science and Telecommunication Engineering
and the
Master in Mathematics and Applications

By
David Díaz Vico
under the direction of
José Ramón Dorronsoro Ibero

Madrid, September 19, 2012

Contents

Contents ii

1 Shallow neural networks **1**
1.1 The Ramón y Cajal's neuron and the McCulloch-Pitts neuron 1
1.2 The single layer perceptron 2
1.2.1 Model 2
1.2.2 Delta rule and Novikov's Theorem 3
1.3 The multilayer perceptron 5
1.3.1 Model 5
1.3.2 Error minimization 7
1.3.3 ADALINE 8
1.3.4 Backpropagation 8
1.3.5 Benefits and problems of the deep networks 10
1.4 Logistic and softmax perceptron 11
1.4.1 Logistic perceptron 11
1.4.2 Softmax perceptron 13

2 Deep network components **15**
2.1 Principal Component Analysis 15
2.1.1 Variance maximization 16
2.1.2 Squared error minimization 17
2.1.3 Applications 20
2.1.4 Probabilistic PCA 22
2.2 Independent Component Analysis 26
2.2.1 Mutual information minimization 27
2.2.2 Non Gaussianity maximization 28
2.2.3 Maximum likelihood ICA 28
2.3 Autoencoders 29
2.3.1 Single hidden layer autoencoders 30
2.3.2 Multiple hidden layer autoencoders 33
2.3.3 Infomax principle 33
2.3.4 Sparse autoencoders 35
2.3.5 Denoising autoencoders 37
2.3.6 Maximal activation pattern 38
2.4 Boltzmann machines 39
2.4.1 Definition 39
2.4.2 Learning 41

2.4.3 Difficulties . . . 43
2.4.4 Restricted Boltzmann machines . . . 43

3 Deep networks **47**
3.1 Deep multilayer perceptrons . . . 48
3.1.1 Cost function . . . 48
3.1.2 Activation function . . . 48
3.1.3 Initialization . . . 50
3.2 Deep belief networks and stacked autoencoders . . . 54
3.3 Other models . . . 55
3.3.1 Convolutional networks . . . 55
3.3.2 Receptive fields . . . 57

4 Experiments **59**
4.1 Datasets . . . 59
4.2 Software implementation . . . 60
4.3 Building blocks and overall architectures . . . 62
4.4 Shallow networks experiments . . . 62
4.5 Deep networks experiments . . . 67

5 Conclusion **83**
5.1 Further work . . . 83

A Information and entropy **85**
A.1 Mutual information . . . 87

Bibliography **88**

List of Figures

1.1.1 The biological neuron 2
1.1.2 The artificial neuron 3
1.3.1 XOR problem 6
1.3.2 XOR perceptron 6
1.3.3 Triangle problem 6
1.3.4 Triangle perceptron 7
1.4.1 Softmax perceptron 14

2.3.1 Autoencoder 30
2.4.1 Boltzmann machine 40
2.4.2 Restricted Boltzmann machine 44

3.1.1 Cost function 49
3.1.2 Activation function 1 50
3.1.3 Activation function 2 50
3.1.4 Initialization 1 53
3.1.5 Initialization 2 54
3.2.1 Deep belief network or stacked autoencoder 55
3.3.1 Convolutional network 56

4.4.1 Reconstruction with 100 units AE 64
4.4.2 Reconstruction with 100 units DAE 65
4.4.3 Reconstruction with 100 units SAE 66
4.4.4 Reconstruction with 100 units SDAE 67
4.4.5 Reconstruction with 100 units RBM 68
4.4.6 Reconstruction with 256 units AE 69
4.4.7 Reconstruction with 256 units DAE 70
4.4.8 Reconstruction with 256 units SAE 71
4.4.9 Reconstruction with 256 units SDAE 72
4.4.10 Reconstruction with 256 units RBM 73
4.4.11 Reconstruction errors 74
4.5.1 Classification with 3 hidden layers of 49 units MLP 75
4.5.2 Classification with 3 hidden layers of 49 units AMLP 76
4.5.3 Classification with 2 layers of 49 units AE 77
4.5.4 Classification with 2 layers of 49 units AE 78
4.5.5 Classification with 2 layers of 49 units AE 79
4.5.6 Classification with 2 layers of 49 units AE 80

4.5.7 Classification with 2 layers of 49 units DBN 81
4.5.8 Classification errors for deep networks with 49 units per layer 82

Abstract

Composed of neuron channels that progressively transform a sensory perception into a high level cognitive representation, deep neural networks are one of the most recent computational paradigms at our disposal. These models have not attracted much attention until recent times, mainly because they could not be successfully trained with the standard backpropagation algorithms usually employed in traditional shallow feed-forward networks. Only in the few last years have some effective greedy unsupervised algorithms been developed. That kind of algorithms are important for two reasons: first, they are unsupervised, which makes them closer to the natural human learning process than supervised algorithms, and second, they are computationally feasible and put the deep neural network in a state where it is possible to use the backpropagation training efficiently. However, even now, deep networks are not fully understood and many research lines remain open.

In this memoir, we will provide a brief summary of the models and techniques considered classical neural networks and, from that point, we will present the current state of the new deep networks, including different architectures and training techniques, as well as some related statistical methods. Finally, we will present some standard experiments and discuss the results obtained.

Overview

The main objective of this master thesis is to present the state of the art in deep neural networks design. Composed of neuron channels that progressively transform a sensory perception into a high level cognitive representation, deep neural networks are one of the most recent computational paradigms at our disposal. These models have not attracted much attention until recent times, mainly because they could not be successfully trained with the standard backpropagation algorithms usually employed in traditional shallow feed-forward networks. Only in the few last years have some effective greedy unsupervised algorithms been developed. That kind of algorithms are important for two reasons: first, they are unsupervised, which makes them closer to the natural human learning process than supervised algorithms, and second, they are computationally feasible and put the deep neural network in a state where it is possible to use the backpropagation training efficiently. We present the new advancements in the field following this general structure:

The first chapter will provide a brief summary of the models and techniques considered in classical neural networks. Using the basic components, the McCulloch-Pitts artificial neurons, as building blocks, several networks will be developed: from the simplest Rosenblatt's perceptron to the more sophisticated multilayer perceptrons, used to solve complicated classification problems. Concepts as forward propagation, gradient descent or error backpropagation learning will be presented. We will see the limitations of shallow networks and the necessity of developing deep architectures.

The second chapter presents the basic building blocks of a new deep layer-wise architecture, partially inspired by the mammals visual cortex. These building blocks, the autoencoders and the Boltzmann machines, are closely related to the principal components analysis and independent component analysis techniques, that are also discussed.

In the third chapter several approaches to deep networks are presented. First, deep multilayer perceptrons can be efficiently trained if some precautions are taken. Second, the stacking of layers of autoencoders or Boltzmann machines to create a deep network with an unsupervised learning algorithm is explained. Third, other deep networks models are presented: the convolutional networks and the receptive fields.

Finally, the fourth chapter details the experiments performed. The experiments compare several of the architectures described in the previous chapters, with special attention to the stacked autoencoders and the deep belief networks, considered by some the state of the art in artificial neural networks. There is no available software that performs all the experiments found in the literature, so a simple deep neural network library has been developed, also with the aim of having a building block for further studies and experiments.

Chapter 1

Shallow neural networks

The human nervous system is the most powerful information processing system known. It can parallelly perform many highly complex functions in a quite different way from any present computer. We have already been able to express mathematically some of these functions, like classification and prediction, which enables us to study neural systems from a mathematical point of view.

Although simple shallow neural networks can perform many pattern recognition or regression tasks, we know that highly complex deep neural networks are needed to achieve acceptable results in more advanced jobs.

The basic constituents of a neural network are the neurons. These cells connect and communicate to each other and form the different parts of a nervous system. Let's start this study with an overview of the basic biological neuron and it's artificial counterpart.

1.1 The Ramón y Cajal's neuron and the McCulloch-Pitts neuron

The biological neuron is an electrically excitable cell that, typically, possesses a cell body or soma, dendrites and an axon. The cell body frequently gives rise to multiple dendrites, but only one axon. Usually, electric signals are sent from the axon of a neuron to a dendrite of another.

It is estimated that the human brain has between 80 and 120 billion neurons, and several million only in the primary visual cortex. The primary visual cortex in mammals is a portion of the brain highly specialized for the processing of images, static or moving, and pattern recognition. The number of neurons that compose the primary visual cortex is estimated at around 140 million, and it is more powerful and complex than any artificial neural network ever built.

The mathematical model of the biological neuron we will use is the McCullock-Pitts electronic neuron ([3]). The artificial neuron receives weighted input signals

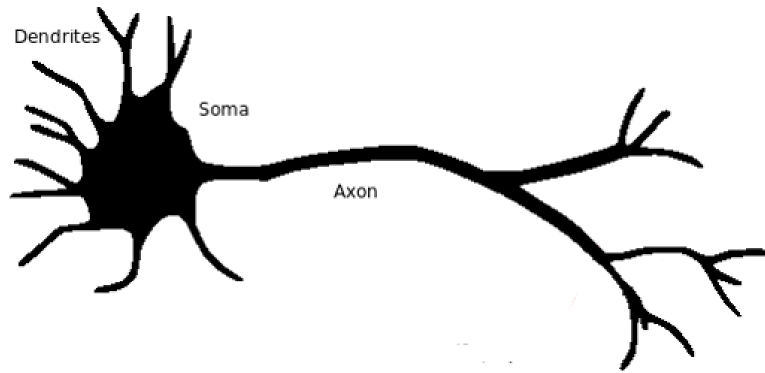

Figure 1.1.1: The biological neuron

(that represent the dendrite inputs), sums them, and transforms the sum with an activation function to produce an output signal (representing the axon output signal).

In standard notation, we use x_d, $d = 0, \dots, D$ for the input signals, w_d, $d = 0, \dots, D$ for the weights, F or φ for the activation function, usually non-linear, and y for the output signal. x_0 is a special input typically assigned the value $+1$, and the associated weight or bias, w_0, is also denoted b. The bias term represent the effect of an external current source I. So we have

$$y = F\left(\sum_{d=0}^{D} w_d x_d\right) = F\left(\sum_{d=1}^{D} w_d x_d + b\right)$$

that, in a more compact matrix notation with $X = (x_0, \dots, x_D)^T$ and $W = (w_0, \dots, w_D)^T$, is expressed as

$$y = F\left(W \cdot X\right),$$

or, if we prefer to use $X = (x_1, \dots, x_D)^T$, $W = (w_1, \dots, w_D)^T$ and b,

$$y = F\left(W \cdot X + b\right).$$

We will now take a look to the simplest feedforward neural networks, the perceptrons.

1.2 The single layer perceptron

1.2.1 Model

The Rosenblatt's perceptron ([4]) is the simplest single layer perceptron, and in fact the simplest neural network. It can be trained to perform pattern classification with two linearly separable classes. This perceptron consists of a single neuron with

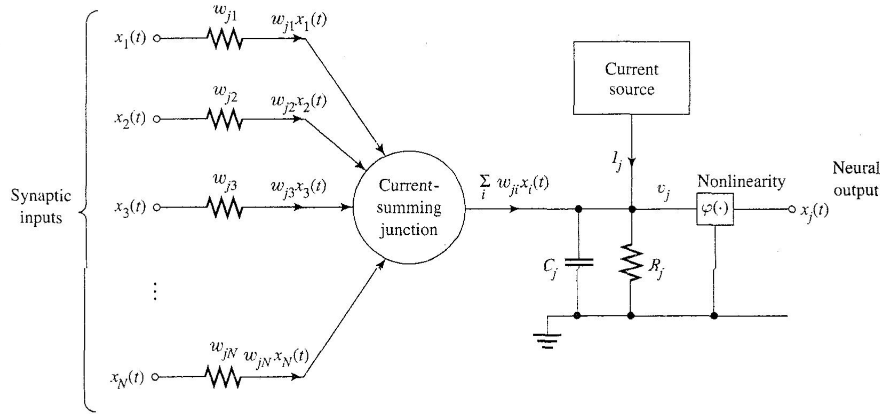

Figure 1.1.2: The artificial neuron

a sign activation function, that is, if the perceptron receives pattern X^n as input, it assigns it a target value y^n representing a class ($+1$ or -1)

$$\sum_{d=0}^{D} W_d X_d^n \begin{cases} > 0 & y^n \Leftarrow +1 \\ < 0 & y^n \Leftarrow -1 \end{cases}$$

What we would like is to obtain the weights and bias that give

$$\sum_{d=1}^{D} W_d X_d^n \begin{cases} > -b & \text{if } y^n = +1 \\ < b & \text{if } y^n = -1 \end{cases}$$

or, expressed in compact notation

$$y^n \sum_{d=1}^{D} W_d X_d^n \equiv y^n W \cdot X^n > -b,$$

so the bias is chosen to model the noise, as the classes are only assigned out of the interval $[b, -b]$.

1.2.2 Delta rule and Novikov's Theorem

The Rosenblatt's perceptron is trained in a supervised way with the algorithm of the delta rule:

The delta rule works as intended: let $b = 0$ for simplicity, then, if the perceptron incorrectly classifies pattern n, we have

$$\begin{aligned} y^n W^n \cdot X^n &= y^n \left(W^{n-1} + y^n X^n\right) \cdot X^n \\ &= y^n W^{n-1} \cdot X^n + y^n \left(y^n X^n\right) \cdot X^n \\ &= y^n W^{n-1} \cdot X^n + \|X^n\|^2 \\ &> y^n W^{n-1} \cdot X^n, \end{aligned}$$

Algorithm 1 Delta rule

```
Initialize W^0
for n = 1...N do
  if y^n W^{n-1} · X^n > 0 then
    W^n ← W^{n-1}
  else
    W^n ← W^{n-1} + y^n X^n
  end if
end for
```

so it improves classification for pattern n.

The Novikov's Theorem ([5]) proves that the Rosenblatt's perceptron learns if the problem is solvable.

Theorem 1 (Novikov)**.** *If the classes are linearly separable, then in a finite number of steps, the Rosenblatt's perceptron will find a solution, given by vector W^*.*

To prove it, we need two inequalities. In what folows, let X^n represent only the incorrectly classified patterns, and W^n the weights after processing pattern X^n.

We have a first inequality

$$\begin{aligned}\|W^n\|^2 &= W^n \cdot W^n \\ &= \left(W^{n-1} + y^n X^n\right) \cdot \left(W^{n-1} + y^n X^n\right) \\ &= W^{n-1} \cdot W^{n-1} + X^n \cdot X^n + 2y^n W^{n-1} \cdot X^n \\ &= \left\|W^{n-1}\right\|^2 + \|X^n\|^2 + 2y^n W^{n-1} \cdot X^n \\ &\leq \left\|W^{n-1}\right\|^2 + \|X^n\|^2 \\ &\leq \left\|W^{n-2}\right\|^2 + \|X^n\|^2 + \left\|X^{n-1}\right\|^2 \\ &\dots \\ &\leq \left\|W^0\right\|^2 + \|X^n\|^2 + \cdots + \left\|X^1\right\|^2,\end{aligned}$$

where we are free to chose $W^0 = 0$, and, if we call

$$R^2 = max\left\{\|X^n\|^2\right\},$$

we have that

$$\|W^n\|^2 \leq nR^2.$$

And, if the problem has a solution W^* with $\|W^*\| = 1$, and

$$\gamma = min\left\{y^n W^* \cdot X^n\right\}$$

that, being W^* a solution, will always be greater than 0, then we have a second inequality

$$\begin{aligned} W^n \cdot W^* &= \left(W^{n-1} + y^n X^n\right) \cdot W^* \\ &= W^{n-1} \cdot W^* + y^n X^n \cdot W^* \\ &\geq W^{n-1} \cdot W^* + \gamma \\ &\geq W^{n-2} \cdot W^* + 2\gamma \\ &\dots \\ &\geq W^0 \cdot W^* + n\gamma, \end{aligned}$$

so

$$W^n \cdot W^* \geq n\gamma$$

and so

$$\|W^n\| \geq n\gamma.$$

Combining both inequalities we have

$$n^2\gamma^2 \leq \|W^n\|^2 \leq nR^2$$

so

$$n \leq \frac{R^2}{\gamma^2}$$

which means that, in as much as $\frac{R^2}{\gamma^2}$ weight changes, we will reach a solution, and the theorem is proved. Also, we deduce that, as R is fixed, if we have a big γ, also called the *margin* of the problem, that is the minimum distance between the classes of the sample, then the problem will be easy and we will need few weight changes.

1.3 The multilayer perceptron

1.3.1 Model

As we have proved, linearly separable problems can be solved by a single layer perceptron. Examples include the OR and the AND problems. But to solve harder problems, like the XOR problem, we need a more powerful model.

By combining several simple perceptrons we get what we know as multilayer perceptron. Multilayer perceptrons have an input layer, one or more hidden layers, and an output layer. They only have feed-forward connections, that is, there are no connections between units of the same layer, between not consecutive layers, or feedback to a previous layer.

It is easy to see that a multilayer perceptron will solve the XOR problem by feeding the outputs of two simple perceptrons into another simple perceptron in a second layer.

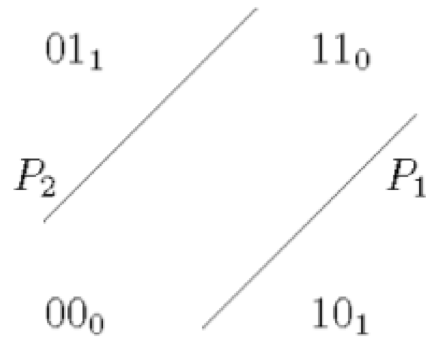

Figure 1.3.1: XOR problem

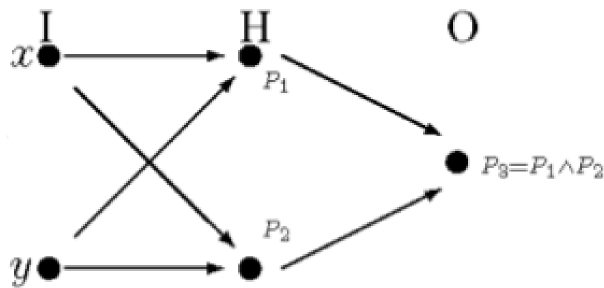

Figure 1.3.2: XOR perceptron

The multilayer perceptron can solve even more complicated problems ([2] chapter 6): by using three units, forming a first layer, and feeding their outputs to another unit, forming the second layer, we can solve any triangular region. And, as any polygon can be built by combining triangles, we can use several of these two-layer perceptrons to detect triangles, and combine them with another unit, being the third layer, which enables us to solve any polygonal region.

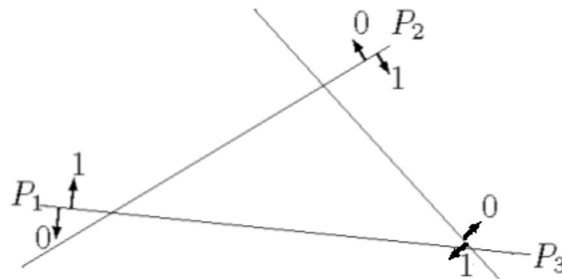

Figure 1.3.3: Triangle problem

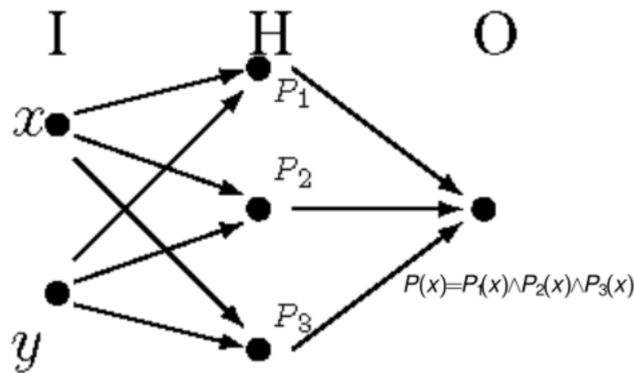

Figure 1.3.4: Triangle perceptron

1.3.2 Error minimization

We have come to the conclusion that multiple layers of units are needed to solve complex problems, but the delta rule is designed to train only single layer perceptrons. To solve this we start by changing our objectives: from predicting a target, to minimizing an error function (also called cost function) of the target and the output of our perceptron with no hidden layers. That is, if t^n is the target (the class) of pattern n, where we wanted

$$y^n = W \cdot X^n = t^n \ \forall n,$$

we now want

$$y^n = W \cdot X^n \approx t^n \ \forall n$$

and we do this by minimizing an error function, usually the mean squared error function

$$E(W) = \frac{1}{2}\frac{1}{N}\sum_{n=1}^{N}(W \cdot X^n - t^n)^2,$$

where the $\frac{1}{2}$ factor appears to simplify calculations. And if we consider that the patterns follow a distribution $p(X,t)$ then

$$E(W) = \frac{1}{2}\int (W \cdot X - t)^2 p(X,t)\, dX dt = \frac{1}{2}\mathbb{E}_{p(X,t)}\left[(W \cdot X - t)^2\right].$$

To minimize the function we perform a gradient descent, so we need the error function to be differentiable, as is the case for the mean squared error. We have

$$E(W) = \frac{1}{2}\mathbb{E}_{p(X,t)}\left[\left(\sum_{d=1}^{D} W_d X_d - t\right)^2\right]$$

and so

$$\frac{\partial E(W)}{\partial W_d} = \frac{1}{2}\mathbb{E}_{p(X,t)}\left[\frac{\partial}{\partial W_d}\left(\sum_{d=1}^{D} W_d X_d - t\right)^2\right]$$
$$= \mathbb{E}_{p(X,t)}\left[\left(W_d X_d - t\right) X_d\right]$$

and

$$\nabla_W E(W) = \mathbb{E}_{p(X,t)}\left[\left(W \cdot X - t\right) X\right]$$

that has a closed solution, but we are not interested in a solution that involves knowing all the patterns of the sample. We want an algorithm that makes our perceptron learn progressively as it receives the patterns.

1.3.3 ADALINE

The solution to our problem is an adaptative learning algorithm that uses local gradient descent. The adaptive linear element or adaptive linear neuron, ADALINE ([6]), is another single unit perceptron. The difference between ADALINE and the Rosenblatt's perceptron is the learning algorithm. In this case, a linear output function like the identity is used, so

$$y^n = W^n \cdot X^n + b^n$$

and, given each pair of input pattern and target, (X^n, t^n), a local error is computed to determine the goodness of the output for this particular pattern. We have

$$E_L\left(W^n\right) = \frac{1}{2}\left(y^n - t^n\right)^2.$$

The local error function is used to adjust the weights through a local gradient descent method. That is, weights are changed in the direction where the local error function decreases fastest, given by the local error function gradient for the selected pattern

$$\nabla_W E_L\left(W^n\right) = \left(y^n - t^n\right) X^n$$

and a learning rate constant, η, determines how much the weights change in each step, so we have the learning rule

$$W^{n+1} = W^n - \eta \nabla_W E_L\left(W^n\right).$$

So, ADALINE means adjusting the weights in a linear and adaptive way through a local gradient descent. Widrow and Hoff (Widrow and Hoff, 1962) proved that, if some conditions (stationarity, ergodicity, correct learning rate η and others) are met, then the weights converge to its optimal value,

1.3.4 Backpropagation

With ADALINE we can properly train a unit when we know the local error, but we need an algorithm to propagate backwards the error from the units in the output layer to the units in preceding layers.

Given a training pattern and target, the perceptron will compute all the activations from the input layer to the output layer, what we can call a *forward pass*. Once we have the output values, we can calculate an error with the selected error function, but that error can only be used to compute the gradient and the new weight values for the output layer units. Backpropagation defines a kind of error associated to a hidden unit as the weighted average of the errors of the units in the following layer.

Let i, j, k be the indices of units in three consecutive layers, w_{ji} the weight of the connection from unit i to unit j, z_i the output of unit i, $s_j = \sum_i w_{ji} z_i$ the sum of the weighted inputs of unit j, and δ_k the generalized error at unit k. Then we have that

$$\frac{\partial E_L}{\partial w_{ji}} = \frac{\partial E_L}{\partial s_j}\frac{\partial s_j}{\partial w_{ji}} = \delta_j \frac{\partial s_j}{\partial w_{ji}},$$

where

$$\begin{aligned}\delta_j &= \sum_k \frac{\partial E_L}{\partial s_k}\frac{\partial s_k}{\partial s_j} \\ &= \sum_k \frac{\partial E_L}{\partial s_k}\frac{\partial s_k}{\partial z_j}\frac{\partial z_j}{\partial s_j} \\ &= \left(\sum_k \frac{\partial E_L}{\partial s_k} w_{kj}\right) F'(s_j)\end{aligned}$$

and

$$\frac{\partial s_j}{\partial w_{ji}} = z_i,$$

so we have that, for the output layer, where we can calculate the error directly,

$$\frac{\partial E_L}{\partial w_{ji}} = \delta_j z_i$$

with

$$\delta_j = y_j - t_j$$

and for the hidden layers

$$\frac{\partial E_L}{\partial w_{ji}} = \left(\sum_k \frac{\partial E_L}{\partial s_k} w_{kj}\right) F'(s_j) z_i = \left(\sum_k \delta_k w_{kj}\right) F'(s_j) z_i.$$

So, with backpropagation we compute the error of the output units, then the general error of the units of the last hidden layer, and successively all the previous hidden layers. This is the *backward pass* of the learning algorithm.

If we suppose that our multilayer perceptron has L layers, all off them with M units, and use N training patterns, then the computational cost of backpropagation is $\mathcal{O}\left(L \times N^3 \times N\right)$.

As we have seen, it is necessary to calculate the derivative of the activation function, so we need it to be differentiable. The most common activation functions are

- Linear (identity) activation:
$$F(x) = x.$$

- Logistic (sigmoid) activation:
$$F(x) = \frac{1}{1 + e^{-\beta x}}.$$

- Hyperbolic tangent activation:
$$F(x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}}.$$

If, we want to solve regression instead of classification problems, we can use linear activation functions instead of the sign function in the last layer ([2] chapter 6). Note, however, that if we build a multilayer perceptron we shouldn't use linear functions in consecutive layers, as the composition of linear functions is also a linear function, and a multilayer perceptron with linear activations in all its layers is equivalent to a single layer perceptron. So, in regression problems, it is typical to use linear activation functions in the output layer but not in the previous layers.

1.3.5 Benefits and problems of the deep networks

Let $\mathcal{D} \subset \mathbb{R}^D$, a family of functions $\mathcal{F}$ is said to be a *Universal Approximation Family* (UAF) if for every continuous function $\phi : \mathcal{D} \to \mathbb{R}$ on compact subsets of $\mathbb{R}^n$ and $\forall\ \epsilon$, then it exists $F \in \mathcal{F}$ that makes

$$\int_{\mathcal{D}} (F(x) - \phi(x))^2 \, p(x) \, dx \leq \epsilon$$

with $p(x)$ a density in $\mathcal{D}$.

For instance, for $\mathcal{D} = [a, b]$, if F is continuous in $[a, b]$, the Weierstrass Theorem says that $\forall\ \epsilon$ there is a polynomial P so that $|F(x) - P(x)| \leq \epsilon$, so the polynomials in $[a, b]$ are a FAU.

For $\mathcal{D} \subset \mathbb{R}^n$, multilayer perceptrons with one hidden layer and linear activations in the output units also form a UAF (Cybenko, 1989, for sigmoid activations. Hornik, 1991), and, for our problems, perceptrons are more useful than polynomials, as we want them to be of use not only for the approximation of the known cases, but also for the unknown, and high order polynomials are not desirable.

If perceptrons with one hidden layer are a UAF, why could we want deeper architectures? It is known that functions that require very large number of units to be computed in a shallow architecture, can be computed with a very small number of units in a deep architecture. The next theorem applies to monotone weighted threshold circuits (see [8]), like perceptrons. If we have a function f_k of N^{2k-2} variables, that is defined by a k-level tree, then

Theorem 2 (Hastad and Goldmann). *A monotone weighted threshold circuit computing f_k that is of depth $k-1$ has size at least 2^{cN} for some constant $c > 0$ and $N > N_0$.*

That is, any function needs a minimum number of units to be computed, and if we use a $k-1$-level architecture instead of a k-level architecture, then the number of units grows exponentially. For more details, see [8] and [9].

However, training a deep neural network has proved to be a very difficult task. The traditional backpropagation algorithm fails to train networks with more than a couple of layers. As can be seen in [10], the gradient vanishes when it is backpropagated through several hidden layers if we don't take the proper precautions. We will discuss this in detail later.

1.4 Logistic and softmax perceptron

Let's see now two of the most common perceptrons used in classification problems.

1.4.1 Logistic perceptron

When solving a two class problem with a perceptron, we can use a sigmoid activation function instead of the sign or linear function we had before. As we have seen, we need to calculate a gradient, so it is important to have a differentiable activation function, which makes the sigmoid function an interesting choice. Also, this perceptron is equivalent to a well known model, the logistic regression.

The logistic regression is useful to obtain the probability of an event as a function of some observed variables, called *explanatory variables*. The model takes a set of $0-1$ observations $\left\{y^1, \ldots, y^N\right\}$ of the event, each one of them following a Bernoulli distribution, and a set of observations

$$\left\{\left(x_1^1, \ldots, x_D^1\right)^T, \ldots, \left(x_1^N, \ldots, x_D^N\right)^T\right\} \equiv \left\{X^1, \ldots, X^N\right\}$$

of the explanatory variables, and estimates the probability of the event based on the result of each test and the information in the explanatory variables, so

$$p^n = \mathbb{E}\left[y^n = 1 \middle| X^n\right].$$

We assume that the *logit* of this probability is a linear function of the explanatory variables

$$logit\,(p^n) = \log \frac{p^n}{1-p^n} = w_0 + w_1 x_1^n + \cdots + w_D x_D^n$$

where the *regression coefficients*, w_j, $j = 1, \ldots, D$, represent the influence of each of the variables in the outcome of the event.

With some simple operations we have

$$\frac{p^n}{1-p^n} = e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}$$

so

$$p^n = e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n} - p^n e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}$$

and so

$$p^n(1 + e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}) = e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}$$

and, finally,

$$p^n = \frac{e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}}{1 + e^{w_0 + w_1 x_1^n + \cdots + w_D x_D^n}},$$

and we get the usual logistic or sigmoid transformation

$$p^n = \frac{1}{1 + e^{-\left(w_0 + w_1 x_1^n + \cdots + w_D x_D^n\right)}},$$

that takes values between 0 and 1 and is monotonically increasing and differentiable in every point. It is clear now that the logistic regression is the function computed by a single unit perceptron with sigmoid activation function where the connection weights are the regression coefficients (w_0 being the bias) and that receives as input X the vector with the values of the explanatory variables.

The regression coefficients are usually estimated with the maximum likelihood method: to estimate a parameter θ and given that the observations $X^1, \ldots, X^N$ are i.i.d. with density function $p(X^n)$, the likelihood function is

$$\begin{aligned}\mathcal{L}\left(\theta | X^1, \ldots, X^N\right) &= p(X^1, \ldots, X^N | \theta) \\ &= \prod_{i=1}^{N} p(X^n | \theta)\end{aligned}$$

To avoid deriving products, and knowing that the logarithm of a function will have its maximum at the same point as the function will, we can use the log-likelihood function

$$\log \mathcal{L}\left(\theta | X^1, \ldots, X^N\right) = \sum_{n=1}^{N} \log p(X^n | \theta)$$

In our problem

$$\begin{aligned}\mathcal{L}\left(w_0,\ldots,w_D\middle|\left(X^1,y^1\right),\ldots,\left(X^N,y^N\right)\right) &= p\left(\left(X^1,y^1\right),\ldots,\left(X^N,y^N\right)\middle| w_0,\ldots,w_D\right)\\ &= \prod_{n=1}^{N} p\left(\left(X^n,y^n\right)\middle| w_0,\ldots,w_D\right)\end{aligned}$$

so we have the log-likelihood function

$$\begin{aligned}\log\mathcal{L}\left(w_0,\ldots,w_D\middle|\left(X^1,y^1\right),\ldots,\left(X^N,y^N\right)\right) &= \sum_{n=1}^{N}\log p\left(\left(X^n,y^n\right)\middle| w_0,\ldots,w_D\right)\\ &= \sum_{n=1}^{N} y^n\log p^n + (1-y^n)\log(1-p^n)\end{aligned}$$

Knowing this, another way to obtain the regression coefficients, is using the labeled observations-patterns $\{(X^1,y^1),\ldots,(X^N,y^N)\}$ to train the perceptron minimizing the error function

$$E = -\frac{1}{N}\sum_{n=1}^{N} y^n\log p^n + (1-y^n)\log(1-p^n).$$

It is clear that backpropagation can also be applied to this network, so it is possible to build a logistic multilayer perceptron.

1.4.2 Softmax perceptron

The single unit perceptron or logistic regression can be generalized to solve classification problems with more than two classes. Instead of estimating the probability of an event, we can estimate a vector with the probabilities of each of the multiple possible outcomes. That is, if the event has K possible outcomes, $\{1,\ldots,K\}$, the softmax regression gives $p\left(y=k|X\right)\ \forall k=1,\ldots,K$ so that $\sum_{k=1}^{K} p\left(y=k|X\right)=1$, so

$$\begin{pmatrix} p\left(y^n=1|X^n,w_{1,\cdot}\right)\\ p\left(y^n=2|X^n,w_{2,\cdot}\right)\\ \vdots\\ p\left(y^n=K|X^n,w_{K,\cdot}\right)\end{pmatrix} = \frac{1}{\sum_{k=1}^{K} e^{w_{k,0}+w_{k,1}x_1^n+\cdots+w_{k,D}x_D^n}}\begin{pmatrix} e^{w_{1,0}+w_{1,1}x_1^n+\cdots+w_{1,D}x_D^n}\\ e^{w_{2,0}+w_{2,1}x_1^n+\cdots+w_{2,D}x_D^n}\\ \vdots\\ e^{w_{K,0}+w_{K,1}x_1^n+\cdots+w_{K,D}x_D^n}\end{pmatrix}$$

the corresponding softmax perceptron will have an output layer of K units, all of them connected to all the inputs.

For the implementation, we can use the cost function

$$E = -\frac{1}{N}\left(\sum_{n=1}^{N}\sum_{k=1}^{K} 1_{\{y^n=k\}}\log\frac{e^{w_{k,0}+w_{k,1}x_1^n+\cdots+w_{k,D}x_D^n}}{\sum_{l=1}^{K} e^{w_{l,0}+w_{l,1}x_1^n+\cdots+w_{l,D}x_D^n}}\right),$$

that is a generalization of the cost function used with the logistic regression.

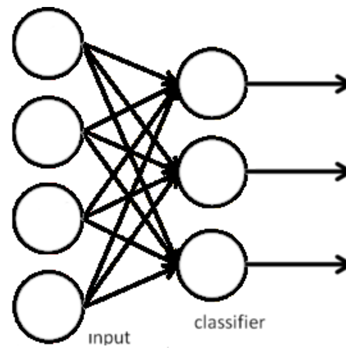

Figure 1.4.1: Softmax perceptron

We don't have a solution for the minimum of this cost function, but using an iterative method we obtain this gradient

$$\nabla_k E = -\frac{1}{N}\sum_{n=1}^{N}\left(X^n\left(\mathbf{1}_{\{y^n=k\}} - p\left(y^n = k|X^n; w_{k,\cdot}\right)\right)\right);$$

for more details, see [7] and [19].

In this case, the extension to multilayer perceptrons is also quite obvious.

Chapter 2

Deep network components

Some studies indicate that the primary visual cortex of mammals could be organized in layers of neurons that progressively transform a sensory perception into a high level cognitive representation. Also, it appears that the functioning of this part of the visual cortex is similar to that of a combination of Gabor filters, widely used for edge detection.

It is also necessary to develop unsupervised or semi-unsupervised learning algorithms for deep neural networks for mainly two reasons: the first comes from our desire to, if possible, imitate human learning, that is in part unsupervised; the second arises when we realise that it is practically impossible to obtain the large quantities of labelled patterns necessary to properly train a deep neural network in a supervised way.

All these observations inspire the development of deep artificial neural networks formed by layers of autoencoders or Boltzmann machines, that are able to learn, with an unsupervised learning algorithm, to operate like Gabor filters, and are appropriate for stacking. In this section we will study these basic components of our proposed deep neural network model and why they are similar to statistical methods like principal component analysis or independent component analysis.

2.1 Principal Component Analysis

Principal component analysis (PCA) is a technique widely used to reduce the dimension of a problem's input data. There are two equivalent definitions of PCA:

- Orthogonal projection of the data to a lower-dimensional space, known as the principal subspace, maximizing the projected data variance.

- Linear projection that minimizes the mean projection error, defined as the mean squared distance between the data and the projections.

In what follows we will discuss these two approaches.

2.1.1 Variance maximization

Let $\{X^n\}$, $n = 1, \ldots, N$ be a set of observations of a D-dimensional variable. We want to project this data to a M-dimensional space, $M < D$, with M fixed, maximizing the variance of the projected data.

We begin considering the projection to a 1-dimensional space ($M = 1$). We define the direction of this space with the D-dimensional vector U^1, that, for simplicity and without loss of generality, will be an unitary vector so that $U^1 \cdot U^1 = 1$. Each point X^n is projected to this space as the scalar value $U^1 \cdot X^n$. The projected data mean is $U^1 \cdot \overline{X}$, where $\overline{X}$ is the sample mean of the data

$$\overline{X} = \frac{1}{N} \sum_{n=1}^{N} X^n$$

and the sample variance of the projected data is

$$\frac{1}{N} \sum_{n=1}^{N} \left(U^1 \cdot X^n - U^1 \cdot \overline{X} \right)^2 .$$

If S is the data covariance matrix, defined as

$$S = \frac{1}{N} \sum_{n=1}^{N} (X^n - \overline{X})(X^n - \overline{X})^T$$

we have that

$$\begin{aligned} \frac{1}{N} \sum_{n=1}^{N} \left(U^1 \cdot X^n - U^1 \cdot \overline{X} \right)^2 &= \frac{1}{N} \sum_{n=1}^{N} \left(U^1 \cdot \left(X^n - \overline{X} \right) \right)^2 \\ &= \frac{1}{N} \sum_{n=1}^{N} U^1 \cdot \left(X^n - \overline{X} \right) \left(X^n - \overline{X} \right)^T U^1 \\ &= \left(U^1 \right)^T S U^1 \end{aligned} \tag{2.1.1}$$

Now we maximize the variance $\left(U^1 \right)^T S U^1$ with respect to U^1. This maximization must be bounded to avoid $\|U^1\| \to \infty$, which is given by the normalization condition $U^1 \cdot U^1 = 1$. To achieve this restricted maximization we use the Lagrange multiplier λ_1:

$$\left(U^1 \right)^T S U^1 - \lambda_1 \left(U^1 \cdot U^1 - 1 \right) ,$$

whose derivative with respect to U^1 is zero when

$$S U^1 = \lambda_1 U^1,$$

that is, U^1 is an eigenvector of S, with associated eigenvalue λ_1 that we obtain using that $U^1 \cdot U^1 = 1$ and gives

$$\lambda_1 = \left(U^1 \right)^T S U^1.$$

So, the variance will be maximized when U^1 is the eigenvector with the largest associated eigenvalue, λ_1. This eigenvector is called the first principal component.

We can define additional principal components in an incremental way taking each new direction as that which maximizes the global projected variance. In the case of projecting to a M-dimensional space, the optimal linear projection that maximizes the projected data variance is the one given by the M eigenvectors of the covariance matrix S associated to the largest M eigenvalues. We prove next this fact.

Formally, if we project to a space of dimension M, the sample variance is

$$\frac{1}{N}\sum_{n=1}^{N}\left(\left(U^1 \cdot X^n - U^1 \cdot \overline{X}\right)^2 + \left(U^2 \cdot X^n - U^2 \cdot \overline{X}\right)^2 + \cdots + \left(U^M \cdot X^n - U^M \cdot \overline{X}\right)^2\right)$$

that, using what we saw for the case $M = 1$ in 2.1.1, is

$$\left(U^1\right)^T SU^1 + \left(U^2\right)^T SU^2 + \cdots + \left(U^M\right)^T SU^M.$$

To maximize this quantity it is necessary to impose bounds as we did before, so we use the Lagrange multipliers $\lambda_1, \ldots, \lambda_M$:

$$\sum_{i=1}^{M}\left(\left(U^i\right)^T SU^i - \lambda_i\left(U^i \cdot U^i - 1\right)\right);$$

computing the partial derivatives with respect to the U^i and making them zero, we get

$$SU^i = \lambda_i U^i$$

that is, each U^i is eigenvector of S with associated eigenvalue λ_i, and

$$\lambda_i = \left(U^i\right)^T SU^i$$

for $i = 1, \ldots, M$, and so, the variance will be maximal when we project X using M eigenvectors of S with the largest eigenvalues.

2.1.2 Squared error minimization

Let's see now a definition of principal components based on the minimization of the projection error. Let $\{U^i\}$, $i = 1, \ldots, D$ be a D-dimensional orthonormal basis, that is,

$$U^i \cdot U^j = \delta_{ij} = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases};$$

being a basis, therefore complete, each point can be represented in a unique way by a linear combination of vectors from the basis

$$X^n = \sum_{i=1}^{D}\alpha_i^n U^i;$$

this corresponds to a rotation of the reference system to a new system defined by the $\{U^i\}$, with the original D components replaced by new ones. Projecting each X^n over the new basis, we get $\alpha_i^n = X^n \cdot U^i$, and we can write

$$X^n = \sum_{i=1}^{D}\left(X^n \cdot U^i\right)U^i. \tag{2.1.2}$$

Our objective is, however, to approximate the points using a representation with less variables, $M < D$, corresponding to a projection to a lower-dimensional subspace, with the smallest possible error. This linear M-dimensional subspace can be represented without loss of generality with the first M vectors of the basis, so we approximate each X^n with

$$\tilde{X}^n = \sum_{i=1}^{M} z_i^n U^i + \sum_{i=M+1}^{D} b_i U^i. \tag{2.1.3}$$

We are free to select the $\{U^i\}$, $\{z_i^n\}$ and $\{b_i\}$ so that the error introduced by the dimension reduction is minimized. We choose the b_i as constants that are the same for all projected points, so that they lie in an M-dimensional subspace. As error measure we will use the square distance between each original point X^n and its approximation $\tilde{X}^n$. So, the objective is to minimize

$$J = \frac{1}{N}\sum_{n=1}^{N} \left\| X^n - \tilde{X}^n \right\|^2 = \frac{1}{N}\sum_{n=1}^{N} \left\| X^n - \sum_{i=1}^{M} z_i^n U^i - \sum_{i=M+1}^{D} b_i U^i \right\|^2.$$

Let's consider first the minimization with respect to $\{z_i^n\}$. Deriving and using orthonormality of the basis, for $j = 1, \ldots, M$, we have

$$\begin{aligned}
\frac{\partial J}{\partial z_j^n} &= \frac{\partial}{\partial z_j^n}\frac{1}{N}\sum_{k=1}^{N}\left\| X^k - \sum_{i=1}^{M} z_i^k U^i - \sum_{i=M+1}^{D} b_i U^i \right\|^2 \\
&= \frac{\partial}{\partial z_j^n}\frac{1}{N}\sum_{k=1}^{N}\sum_{d=1}^{D}\left(X_d^k - \sum_{i=1}^{M} z_i^k U_d^i - \sum_{i=M+1}^{D} b_i U_d^i \right)^2 \\
&= \frac{\partial}{\partial z_j^n}\frac{1}{N}\sum_{k=1}^{N}\sum_{d=1}^{D}\Bigg((X_d^k)^2 - 2X_d^k \cdot \sum_{i=1}^{M} z_i^k U_d^i - 2X_d^k \cdot \sum_{i=M+1}^{D} b_i U_d^i \\
&\quad + 2\left(\sum_{i=1}^{M} z_i^k U_d^i\right)\left(\sum_{i=M+1}^{D} b_i U_d^i\right) + \left(\sum_{i=1}^{M} z_i^k U_d^i\right)^2 + \left(\sum_{i=M+1}^{D} b_i U_d^i\right)^2\Bigg) \\
&= \frac{1}{N}\sum_{d=1}^{D}\left(-2X_d^n \cdot U_d^j + 2U_d^j \cdot \sum_{i=M+1}^{D} b_i U_d^i + 2\left(\sum_{i=1}^{M} z_i^n U_d^i\right)\cdot U_d^j \right) \\
&= \frac{2}{N}\sum_{d=1}^{D}\left(-X_d^n \cdot U_d^j + z_j^n U_d^j \cdot U_d^j \right) \\
&= \frac{2}{N}\left(-X^n \cdot U^j + z_j^n \right)
\end{aligned}$$

which is zero when

$$z_j^n = X^n \cdot U^j. \tag{2.1.4}$$

When minimizing with respect to b_j, for $j = M+1, \ldots, D$, we get

$$\frac{\partial J}{\partial b_j} = \frac{1}{N}\sum_{n=1}^{N}\sum_{d=1}^{D}\left(-2X_d^n \cdot U_d^j + 2\sum_{i=1}^{M} z_i^n U_d^i \cdot U_d^j + 2\left(\sum_{i=M+1}^{D} b_i U_d^i\right)\cdot U_d^j\right)$$
$$= \frac{2}{N}\sum_{n=1}^{N}\left(-X^n \cdot U^j + b_j\right),$$

that is zero when

$$b_j = \overline{X}\cdot U^j. \tag{2.1.5}$$

Using equations (2.1.2), (2.1.3), (2.1.4) and (2.1.5), we have

$$X^n - \tilde{X}^n = \sum_{i=1}^{D}\left(X^n \cdot U^i\right)U^i - \sum_{i=1}^{M} z_i^n U^i - \sum_{i=M+1}^{D} b_i U^i$$
$$= \sum_{i=1}^{D}\left(X^n \cdot U^i\right)U^i - \sum_{i=1}^{M}\left(X^n \cdot U^i\right)U^i - \sum_{i=M+1}^{D}\left(\overline{X}\cdot U^i\right)U^i$$
$$= \sum_{i=M+1}^{D}\left(\left(X^n - \overline{X}\right)\cdot U^i\right)\cdot U^i,$$

where it can be seen that the projection from X^n to $\tilde{X}^n$ is in the subspace orthogonal to the principal subspace, as this projection is a linear combination of the $\{U^i\}$ for $i = M+1, \ldots, D$, which is logical, as the minimal error is obtained by performing an orthogonal projection.

We can express the error J as a function of $\{U^i\}$

$$J = \frac{1}{N}\sum_{n=1}^{N}\left\|\sum_{i=M+1}^{D}\left(\left(X^n - \overline{X}\right)\cdot U^i\right)\cdot U^i\right\|^2$$
$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{i=M+1}^{D}\left(X^n \cdot U^i - \overline{X}\cdot U^i\right)^2$$
$$= \sum_{i=M+1}^{D}\left(\frac{1}{N}\sum_{n=1}^{N}\left(X^n \cdot U^i - \overline{X}\cdot U^i\right)^2\right)$$
$$= \sum_{i=M+1}^{D}\left(U^i\right)^T S U^i.$$

We still have to minimize J with respect to $\{U^i\}$, where we will use the orthonormality conditions to avoid $U^i = 0$. Let's suppose we have to choose the direction U^i that minimizes J, with the condition $U^i \cdot U^i = 1$. We will use the Lagrange multiplier λ_i to satisfy the restriction, and so we have to minimize

$$\tilde{J} = \sum_{i=M+1}^{D}\left(\left(U^i\right)^T S U^i - \lambda_i\left(U^i \cdot U^i - 1\right)\right)$$

then, deriving with respect to U^i and making it zero, we have that $SU^i = \lambda_i U^i$, that is, U^i is an eigenvector of S with eigenvalue λ_i, for $i = 1, \ldots, D$. $\tilde{J}$ will be minimal when we take the eigenvectors associated to the smallest eigenvalues. and

$$J = \sum_{i=M+1}^{D} \lambda_i.$$

That is, to find the smallest value of J, we must choose the eigenvectors that correspond to the largest eigenvalues, and the principal subspace, complement of the subspace generated by the eigenvectors with smallest eigenvalues, will be that generated by the eigenvectors of largest eigenvalues. All this agrees with what we saw in the previous section, that consisted in projecting to the subspace that maximizes the variance of the projected data.

2.1.3 Applications

Maybe the most immediate application of PCA is data compression. When we do the transformation, the smaller the value of M, the greater the compression we will achieve. Another application is data pre-processing. In this case we want to standardize certain properties of the data set, for example when we have attributes with different orders of magnitude. Also, PCA can decorrelate the variables, an effect known as *whitening* or *sphereing*. For this purpose, we write the eigenvector equation

$$SU = UL$$

or equivalently

$$U^T SU = L,$$

where L is a $D \times D$ diagonal matrix with elements λ_i and U is a $D \times D$ orthonormal matrix with the U^i as columns. Then, for each point X^n we define its transformed value as

$$Y^n = L^{-1/2} U^T \left(X^n - \overline{X} \right)$$

with $\overline{X}$ the sample mean. It is easy to see that the $\{Y^n\}$ have zero mean and ther covariance matrix is the identity

$$\begin{aligned}
\frac{1}{N} \sum_{n=1}^{N} Y^n (Y^n)^T &= \frac{1}{N} \sum_{n=1}^{N} L^{-1/2} U^T \left(X^n - \overline{X} \right) \left(X^n - \overline{X} \right)^T U L^{-1/2} \\
&= L^{-1/2} U^T \frac{1}{N} \sum_{n=1}^{N} \left(X^n - \overline{X} \right) \left(X^n - \overline{X} \right)^T U L^{-1/2} \\
&= L^{-1/2} U^T S U L^{-1/2} \\
&= L^{-1/2} L L^{-1/2} = I,
\end{aligned}$$

where we have used that $U^T SU = L$ and L is diagonal.

A practical problem we may face when computing PCA is the computational cost of calculating all the eigenvectors of a $D \times D$ matrix, that is $\mathcal{O}(D^3)$ (see [11] chapter 12). If we only need the first M principal components, and so only need the first M eigenvectors and eigenvalues, we can use some more efficient techniques that have cost only $\mathcal{O}(MD^2)$ (see [11] chapter 12).

In some cases (see [11] for more detailed examples), the number of points is much smaller than the dimension of the space. We have this situation, for example, when we use PCA over a set of hundreds of images where each one of them is a vector of several millions of dimensions (three values of color per pixel, for each pixel in the image). If in a D-dimensional space we have a set of N points, where $N < D$, then the subspace these points form has, as much, dimension $N - 1$, and so there is no point in using PCA for values of M larger than $N - 1$. In fact, if we perform PCA, we will find that at least $D - N + 1$ eigenvalues are zero, as they are related to eigenvectors for which the data set has variance zero.

To solve this problem, we define X as the $(N \times D)$-dimensional matrix whose n-th row is $\left(X^n - \overline{X}\right)^T$. The covariance matrix is $S = \frac{1}{N}X^T X$, and its eigenvector equation is

$$\frac{1}{N}X^T X U^i = \lambda_i U^i.$$

If we now multiply each side by X, we have

$$\frac{1}{N}XX^T(XU^i) = \lambda_i(XU^i),$$

and, if $V^i = XU^i$, we get

$$\frac{1}{N}XX^T V^i = \lambda_i V^i,$$

that is the eigenvector equation of the $N \times N$ matrix $\frac{1}{N}XX^T$. We can see it has the same $N-1$ eigenvalues λ_i as the original covariance matrix, but it does not have the $D - N + 1$ zero eigenvalues. So, we can calculate the eigenvectors with cost $\mathcal{O}(N^3)$ instead of $\mathcal{O}(D^3)$.

To find the eigenvectors $\{U^i\}$ we multiply both sides by X^T to obtain

$$\left(\frac{1}{N}X^T X\right)\left(X^T V^i\right) = \lambda_i \left(X^T V^i\right),$$

where we see that $\left(X^T V^i\right)$ is an eigenvector of S with eigenvalue λ_i. Note that these eigenvectors are not normalized, and if we normalize V^i, then

$$X^T V^i = X^T X U^i = N\lambda_i U^i$$

with the vector $X^T V^i$ having norm $\sqrt{N\lambda_i}$, and $U^i = \frac{1}{\sqrt{N\lambda_i}}X^T V^i$.

2.1.4 Probabilistic PCA

The definition of PCA seen before is based in a linear projection of the data to a subspace of lower dimension than the original data space. Lets see now how PCA can also be expressed as the solution of a problem of maximum likelihood in a probabilistic model with latent variables.

We start introducing an explicit latent variable, Z, corresponding to the principal component subspace. Then, we define a gaussian prior distribution over this latent variable, $p(Z)$, that has zero mean and unit covariance

$$p(Z) = \mathcal{N}(0, I).$$

We also assume a Gaussian distribution, conditioned by Z, over the observed variable X, $p(X|Z)$. X has as mean a linear function of Z controlled by the $D \times M$ matrix W and the D-dimensional vector μ.

$$p(X|Z) = \mathcal{N}\left(WZ + \mu, \sigma^2 I\right)$$

The columns of W generate a linear subspace in the space of the data that corresponds to the principal subspace. The parameter σ^2 determines the variance of the conditional distribution. It is clear that there is no loss of generality if we assume that $p(Z)$ has zero mean and unit covariance.

The D-dimensional variable X is defined as a linear transformation of the M-dimensional latent variable Z with Gaussian noise added, so

$$X = WZ + \mu + \epsilon$$

with Z a Gaussian M-dimensional variable and ϵ a Gaussian D-dimensional variable with zero mean and $\sigma^2 I$ variance.

Now we want to determine the values of W, μ and σ^2 using the maximum likelihood method. We need an expression for the marginal distribution of the observed variable, $p(X)$, to build the likelihood function

$$p(X) = \int p(X|Z)p(Z)\, dZ.$$

The marginal distribution is also Gaussian (see [11] chapter 12 and [12] for more details), with D-dimensional mean vector

$$\mathbb{E}[X] = \mathbb{E}[WZ + \mu + \epsilon] = \mu,$$

where we have used that $Z \sim \mathcal{N}(0, I)$ and $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$. And the $D \times D$ covariance matrix can be derived as follows

$$\begin{aligned} C = Cov[X] &= \mathbb{E}\left[(WZ+\epsilon)(WZ+\epsilon)^T\right] \\ &= \mathbb{E}\left[(WZ+\epsilon)(Z^TW^T+\epsilon^T)\right] \\ &= \mathbb{E}\left[WZZ^TW^T + WZ\epsilon^T + \epsilon Z^TW^T + \epsilon\epsilon^T\right] \\ &= \mathbb{E}\left[WZZ^TW^T\right] + \mathbb{E}\left[\epsilon\epsilon^T\right] \\ &= WW^T + \sigma^2 I, \end{aligned}$$

where we have used $Z \sim \mathcal{N}(0, I)$, $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$ and that Z and ϵ are independent. Finally, we have that

$$p(X) = \mathcal{N}(\mu, C).$$

Apart from the distribution $p(X)$, we need the posterior distribution $p(Z|X)$

$$p(Z|X) = \mathcal{N}\left(M^{-1}W^T(X-\mu), \sigma^{-2}M\right),$$

where $M = W^TW + \sigma^2 I$. For more detailed information on how to compute the posterior, see [11] chapter 12 and [12].

Now we will find the model parameters by the method of maximum likelihood. In our problem, $X^n \sim \mathcal{N}(\mu, C)$, and so the density function is

$$p(X^n) = \frac{1}{\sqrt{(2\pi)^D}}\frac{1}{\sqrt{|C|}}e^{-\frac{1}{2}(X^n-\mu)^TC^{-1}(X^n-\mu)}$$

and so the log-likelihood is

$$\begin{aligned} \log \mathcal{L}\left(W, \mu, \sigma^2 | X^1, \ldots, X^N\right) &= \sum_{n=1}^{N} \log p(X^n | W, \mu, \sigma^2) \\ &= -\frac{ND}{2}\log 2\pi - \frac{N}{2}\log|C| - \frac{1}{2}\sum_{n=1}^{N}(X^n-\mu)^TC^{-1}(X^n-\mu). \end{aligned}$$

To maximize with respect to μ, we derive the function with respect to μ and make it zero

$$\frac{\partial}{\partial\mu}\log \mathcal{L}\left(W, \mu, \sigma^2 | X^1, \ldots, X^N\right) = \sum_{n=1}^{N} C^{-1}(X^n-\mu) = 0$$

and so

$$\sum_{n=1}^{N} C^{-1}X^n = NC^{-1}\mu,$$

and we have that

$$\mu_{ML} = \frac{1}{N}\sum_{n=1}^{N} X^n = \overline{X}$$

as we expected, and this maximum is unique because the log-likelihood is quadratic with respect to μ.

Maximization with respect to W is more complex, but also has a closed solution:

$$W_{ML} = U_M(L_M - \sigma^2 I)^{1/2} R$$

where U_M is a $D \times M$ matrix with any subset of size M of the eigenvectors of S, the data covariance matrix, as columns, L_M is the diagonal matrix with the corresponding eigenvalues, and R is any $M \times M$ orthogonal matrix. Again, for more details, refer to [11] and [12].

We can see that

$$\begin{aligned} W_{ML} W_{ML}^T &= U_M(L_M - \sigma^2 I)^{1/2} R R^T (L_M - \sigma^2 I)^{1/2} U_M^T \\ &= U_M(L_M - \sigma^2 I) U_M^T \\ &= U_M L_M U_M^T - \sigma^2 I \\ &= \tilde{S} - \sigma^2 I, \end{aligned}$$

with $\tilde{S}$ an estimation of S.

We can prove that the maximum likelihood is obtained choosing the M eigenvectors with the larger eigenvalues. If we order the eigenvectors U^i from the smallest to the largest eigenvalue, we have that the ones that generate the principal subspace (the columns of W) are $U^1, \dots, U^M$. The solution of σ^2 that maximizes the likelihood is

$$\sigma_{ML}^2 = \frac{1}{D - M} \sum_{i=M+1}^{D} \lambda_i,$$

so σ_{ML}^2 is the mean variance associated to the discarded dimensions. The idea is to capture the variance of the data along the principal directions, and approximating the variance in the other directions with a mean value.

Once we have an exact and closed solution for the parameters with the method of maximum likelihood, we may want an alternative, iterative method to find the solution in high-dimensional spaces, where $M << D$, to reduce the computational costs. The EM algorithm can be helpful, as it avoids the calculations to get the covariance matrix and its eigenvectors and eigenvalues.

In the EM algorithm, in the M step we must first write the log-likelihood of the data and calculate its expected value with respect to the posterior distribution of the latent variable using the previous values of the parameters. Then, in the next step, M, we will maximize this expected log-likelihood value to get then new values of the parameters

Assuming the data are independent, and, as we already know

$$Z \sim \mathcal{N}(0, I),$$

$$p(Z) = \frac{1}{\sqrt{(2\pi)^D}} e^{-\frac{1}{2}Z^T Z}$$

and

$$X|Z \sim N(WZ + \mu, \sigma^2 I),$$

$$p(X|Z) = \frac{1}{\sqrt{(2\pi\sigma^2)^D}} e^{-\frac{1}{2\sigma^2}(X-WZ-\mu)^T(X-WZ-\mu)};$$

so, the log-likelihood is

$$\log p(X, Z|\mu, W, \sigma^2) = \sum_{n=1}^{N} \big(\log p(X^n|Z^n) + \log p(Z^n)\big),$$

where X is the matrix that has X^n as its n-th row and Z is the matrix that has Z^n as its n-th row. It is convenient to substitute μ with $\overline{X}$ at this point.

In the M step, we calculate the expected value of the log-likelihood

$$\begin{aligned}\phi = \mathbb{E}\left[\log p(X, Z|\mu, W, \sigma^2)\right] &= -\sum_{n=1}^{N} \Bigg(\frac{D}{2}\log 2\pi\sigma^2 + \frac{1}{2}Tr(\mathbb{E}\left[Z^n(Z^n)^T\right]) \\ &+\frac{1}{2\sigma^2}\|X^n - \mu\|^2 - \frac{1}{\sigma^2}\mathbb{E}\left[Z^n\right]^T W^T(X^n - \mu) \\ &+ \frac{1}{2\sigma^2}Tr(\mathbb{E}\left[Z^n(Z^n)^T\right] W^T W)\Bigg),\end{aligned}$$

and we use that $\mathbb{E}\left[Z^n(Z^n)^T\right] = Cov[Z^n] + \mathbb{E}\left[Z^n\right]\mathbb{E}\left[Z^n\right]^T$ to evaluate

$$\mathbb{E}\left[Z^n\right] = M^{-1}W^T(X^n - \overline{X}),$$

$$\mathbb{E}\left[Z^n(Z^n)^T\right] = \sigma^2 M^{-1} + \mathbb{E}\left[Z^n\right]\mathbb{E}\left[Z^n\right]^T;$$

that is, we use first W_t and σ_t and then $\mathbb{E}\left[Z^n(Z^n)^T\right]_t$ and $\mathbb{E}\left[Z^n\right]_t$ and we obtain the expected value of the log-likelihood as a component explicitly dependent of Z and a component implicitly dependent of Z through X (see [11] chapter 12 for more details).

In the M step we maximize with respect to W and σ^2

$$W_{new} = \left(\sum_{n=1}^{N}(X^n - \overline{X})\mathbb{E}\left[Z^n\right]^T\right)\left(\sum_{n=1}^{N}\mathbb{E}\left[Z^n(Z^n)^T\right]\right)^{-1},$$

$$\sigma^2_{new} = \frac{1}{ND}\sum_{n=1}^{N}\left(\|X^n - \overline{X}\|^2 - 2\mathbb{E}\left[Z^n\right]^T W^T_{new}(X^n - \overline{X}) + Tr(\mathbb{E}\left[Z^n(Z^n)^T\right] W^T_{new}W_{new})\right);$$

that is, in this step we maximize the function $\phi(W, \sigma, W_t, \sigma_t)$, obtained in the previous step, to calculate the new values of the parameters, that we will use in the next iteration of the algorithm.

2.2 Independent Component Analysis

We will now discuss a technique related to principal component analysis and that share some common ideas with the restricted Boltzmann machines, that we will study later.

Imagine a situation where two people are talking at the same time, and we have two microphones that record their voices. The amplitude of one of the voices is not related to the amplitude of the other, so we can say they are independent. If we ignore the delays or echoes, then the signals received by the microphones in an instant will be linear combinations of the amplitudes of the two voices. The coefficients of this combination will be constants, and if we can infer their values from the sampled data, we will be able to revert the mixing process and obtain the two original signals. This kind of problems are called *blind source separation*, as we only have the mixed signals, not the sources or the mixing coefficients.

Consider now a model where the observed variables, X_d^n, $d = 1, \dots, D$, are linearly related to the latent variables, Z_m^n, $m = 1, \dots, M$, and the latent variables do not follow a Gaussian distribution. Independent component analysis tries to find latent variables that are independent, so their distribution factorizes as

$$p(Z^n) = \prod_{m=1}^{M} p(Z_m^n).$$

We represent the set of the latent variables as

$$Z = \begin{pmatrix} Z^1 & \dots & Z^N \end{pmatrix} = \begin{pmatrix} Z_1^1 & \dots & Z_1^N \\ \vdots & \ddots & \vdots \\ Z_M^1 & \dots & Z_M^N \end{pmatrix};$$

we represent the observed variables as

$$X = \begin{pmatrix} X^1 & \dots & X^N \end{pmatrix} = \begin{pmatrix} X_1^1 & \dots & X_1^N \\ \vdots & \ddots & \vdots \\ X_D^1 & \dots & X_D^N \end{pmatrix}.$$

Each of the latent variables will contribute with a different coefficient to each of the observed variables, that is, $X_d^n = \sum_{m=1}^{M} a_{d,m} Z_m^n$, so we can represent the mixing matrix as

$$A = \begin{pmatrix} a_{1,1} & \dots & a_{1,M} \\ \vdots & \ddots & \vdots \\ a_{D,1} & \dots & a_{D,M} \end{pmatrix}$$

and so $X = AZ$. Also, we can define the inverse of A,

$$A^{-1} = \begin{pmatrix} \alpha_{1,1} & \dots & \alpha_{1,D} \\ \vdots & \ddots & \vdots \\ \alpha_{M,1} & \dots & \alpha_{M,D} \end{pmatrix},$$

that, naturally, reverts the mixing and gives the latent variables from the observed ones

$$Z = A^{-1}X.$$

As we usually don't know A, we cannot calculate the exact A^{-1}, but just an approximation, W, that will give us an approximation of the latent variables, $Y = WX$

$$Y = \begin{pmatrix} Y^1 & \dots & Y^N \end{pmatrix} = \begin{pmatrix} Y_1^1 & \dots & Y_1^N \\ \vdots & \ddots & \vdots \\ Y_M^1 & \dots & Y_M^N \end{pmatrix}.$$

It is important to note that ICA needs $D \geq M$, and, usually, to simplify calculations by operating with square matrices, $D = M$.

The non-Gaussianity condition is necessary to have a unique solution to the ICA problem. If we have Gaussian independent latent variables, then any orthogonal transformation of them will be a feasible solution. That is, if R is an orthogonal matrix ($RR^T = I$), then $Z' = RZ$ and $A' = AR^T$ give another solution to the problem, with $X = A'Z'$. The new latent variables are also uncorrelated

$$\mathbb{E}\left[Z'(Z')^T\right] = \mathbb{E}\left[RZZ^TR^T\right] = R\mathbb{E}\left[ZZ^T\right]R^T = RIR^T = I$$

and, in our case, having both a joint Gaussian distribution and being uncorrelated, independent.

2.2.1 Mutual information minimization

Now we know what we are looking for, but working with statistical independence is complex, as we would have to verify independence for every subset of the latent variables. We can, however, use the mutual information to measure in some way the independence between the variables. It can be proved (see Appendix A on information and entropy) that the mutual information is non-negative, being zero when the variables are independent. Also, we can generalize the mutual information (see the Appendix)

$$\mathbb{I}(X, Y) = \mathbb{H}(X) - \mathbb{H}(X|Y)$$

from two variables to M variables

$$\mathbb{I}(Z_1, \dots, Z_M) = \sum_{m=1}^{M} \mathbb{H}(Z_m) - \mathbb{H}(Z_1, \dots, Z_M)$$

and minimize the mutual information as a way of maximizing the independence between the variables.

That said, we can try to find a transformation $Y = WX$ that minimizes the mutual information of the latent variables Y.

2.2.2 Non Gaussianity maximization

Another approach consists in maximizing the non-Gaussianity of the Y variables. Mixing at least two independent latent variables we get observed variables that are not independent, as $Y = WX = WAZ$, and so

$$Y_m = \sum_{d=1}^{D} \sum_{i=1}^{M} W_{m,d} A_{d,i} Z_i$$

By the Central Limit Theorem, we know that this variables, being a sum of the hidden variables, will be "more Gaussian" than, at least, one of the hidden variables, and if we maximize the non-Gaussianity, we will find a linear combination that has all but one of the coefficients as zero, so, we get an approximation of the latent variables Z^n.

What measure of Gaussianity can we use? One option is kurtosis. The kurtosis is 0 for Gaussian distributions, and maximizing its absolute value will somehow minimize the distribution Gaussianity.

Another option comes from the observation that the Normal distribution has the highest entropy of all the distributions with the same variance. If we maximize the negentropy

$$\mathbb{J}(Y^n) = \mathbb{H}(Y_G^n) - \mathbb{H}(Y^n),$$

that is defined as the difference between the entropy of our variable, $\mathbb{H}(Y^n)$, and the entropy of a variable that follows the Gaussian distribution with the same variance as our distribution, $\mathbb{H}(Y_G^n)$, then we will be minimizing the Gaussianity. In practice, only approximations of the entropy are used, as it is difficult to calculate the precise value of the entropy.

2.2.3 Maximum likelihood ICA

We can perform ICA with the method of maximum likelihood. We build a likelihood function as a function of the linear combination coefficients, and then use it to estimate the optimal matrix A^{-1} that maximizes the probability of obtaining the mixtures from the sources. That is, we will search the matrix W that makes the distribution of $Y = WX$ as close as possible to the distribution of Z.

Let p_X be the joint distribution function of the observed variables, and p_Z that of the latent variables. In our example, the number of latent variables and observed variables is the same, so A^{-1} is a square matrix. Then

$$p_X(X) = p_Z(Z)|A^{-1}|$$

where $|A^{-1}| = |\delta Z/\delta X|$ is the jacobian of Z with respect to X. This function gives us the probability of getting X given A^{-1}.

We use this function to define our likelihood function

$$\mathcal{L}\,(W|X) = p_X(X|W) = p_Z(WX)|W|$$

and, to simplify calculations when maximizing with respect to W, we use the log-likelihood function

$$\log \mathcal{L}\,(W|X) = \sum_{n=1}^{N} \log p_Z(W^T X^n) + N \log |W|$$

and the matrix W that maximizes this function will be the desired matrix A^{-1}.

We have not assumed any distribution for the latent variables. In practice, some common choices are

$$p_Z(Y) = 1 - tanh(Y)^2$$

and

$$p_Z(Y) = \frac{1}{\pi cosh(Y)} = \frac{1}{\pi(e^Y + e^{-Y})};$$

see [11] chapter 12 and [14] for more details.

2.3 Autoencoders

It is possible to train neural networks in an unsupervised way, for example to reduce the dimension of a problem. This can be achieved with a special type of perceptron that has the same number of input and output units, and that optimizes the weights to minimize the error between the input and its reconstruction at the output.

Lets consider a single hidden layer perceptron, with an input layer of D units, an output layer of D units, and M, $M < D$, hidden layer units. The targets used to train it are the same input patterns, so the network will try to reconstruct each input vector at the output. This kind of autoassociative perceptron is called autoencoder.

If we have less hidden units than input/output units then, in general, a perfect reconstruction will be impossible, so the hidden layer will become some kind of narrow channel that reduces the data dimension. The weights of the network will be determined in the process of minimizing the error function that measures the difference between the input vectors and their reconstructions at the output, and we expect to have an optimal representation (of the given dimension) of these vectors at the hidden layer.

We can also use autoencoders with broad hidden layers, even broader than the input/output layers, but, as we will see, additional precautions are needed to avoid training a useless network that simply copies the input to the hidden layer and then to the output.

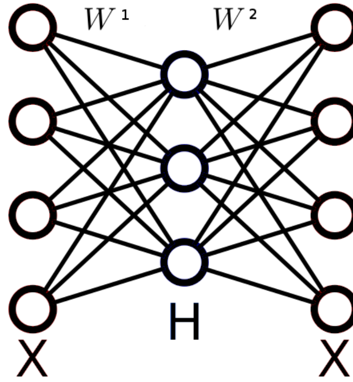

Figure 2.3.1: Autoencoder

2.3.1 Single hidden layer autoencoders

If the input is the D-dimensional real vector X^n, $n = 1, 2, \ldots, N$, the hidden layer output will be the M-dimensional vector

$$H^n = F\left(W^1 X^n + B^1\right)$$

where W^1 is the $M \times D$ weight matrix from the input to the hidden units, and B^1 is the M-vector of the hidden units biases. The function F can be non-linear, like a sigmoid function. The corresponding output will be

$$Y^n = W^2 H^n + B^2$$

where W^2 is the $D \times M$ weight matrix from the hidden to the output units, and B^2 is the D-vector of the output units biases. Our objective is to find the matrices W^1, W^2, and the vectors B^1, B^2 that minimize the mean squared error function

$$E = \sum_{n=1}^{N} \|X^n - Y^n\|^2$$

Let $X = \left[X^1, \ldots, X^N\right]$ be the $D \times N$ matrix formed with the N input vectors of the training set, $H = \left[H^1, \ldots, H^N\right]$ the $M \times N$ matrix with the corresponding hidden layer vectors, and $Y = \left[Y^1, \ldots, Y^N\right]$ the $D \times N$ matrix of the corresponding vectors at the output layer. The process to get Y from X is

$$H = F\left(W^1 X + B^1 U^T\right)$$

$$Y = W^2 H + B^2 U^T$$

where U is a ones vector of the appropriate dimension. In this notation, the error function is written

$$E = \|X - Y\|^2 = \left\|X - W^2 H - B^2 U^T\right\|^2$$

with $\|\cdot\|$ the Frobenius norm $\|A\| = \sqrt{\sum_i \sum_j |a_{ij}|^2}$, and using that $\|A\|^2 = tr\left(AA^T\right)$ we have

$$
\begin{aligned}
E &= tr\left[\left(X - W^2H - B^2U^T\right)\left(X^T - H^T\left(W^2\right)^T - U\left(B^2\right)^T\right)\right] \\
&= tr\left[XX^T - W^2HX^T - B^2U^TX^T - XH^T\left(W^2\right)^T + W^2HH^T\left(W^2\right)^T\right. \\
&\quad \left. + B^2U^T H^T\left(W^2\right)^T - XU\left(B^2\right)^T + W^2HU\left(B^2\right)^T + B^2U^TU\left(B^2\right)^T\right] \\
&\equiv tr(\xi).
\end{aligned}
$$

We will calculate the optimal B^2 by minimizing E with respect to B^2, using that

$$\frac{\partial}{\partial B^2} tr(\xi) = tr\left(\frac{\partial}{\partial B^2}\xi\right)$$

and

$$
\begin{aligned}
\frac{\partial}{\partial B^2}\xi &= -XU + W^2HU - XU + W^2HU + 2B^2U^TU \\
&= -2XU + 2W^2HU + 2B^2U^TU \\
&= -2XU + 2W^2HU + 2B^2N,
\end{aligned}
$$

wich is zero when

$$B^2 = \frac{1}{N}\left(X - W^2H\right)U, \tag{2.3.1}$$

to get

$$
\begin{aligned}
E &= \left\|X - W^2H - B^2U^T\right\|^2 \\
&= \left\|X - W^2H - \frac{1}{N}\left(X - W^2H\right)UU^T\right\|^2 \\
&= \left\|X' - W^2H'\right\|^2
\end{aligned}
$$

where $X' = X\left(I - UU^T/N\right)$ y $H' = H\left(I - UU^T/N\right)$. And, as W^2 has M rank or smaller, we get that the W^2H' that minimizes E is the best M rank approximation for X' with the Frobenius norm (see [15] for more details).

Lets consider the Singular Value Decomposition (SVD) of X', which is, precisely, the best approximation of the matrix with respect to the Frobenius norm for a given rank (see [15])

$$X' = U_D\Sigma_D V_D^T,$$

where U_D is an unitary $D \times D$ matrix with the $X'X'^T$ eigenvectors as columns and V_D a $N \times D$ matrix with orthonormal column vectors that are the eigenvectors of X'^TX', with each eigenvector associated with the eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_D$, and $\Sigma_D = diag[\sigma_1, \sigma_2, \ldots, \sigma_n]$ the diagonal matrix with $\sigma_i = \sqrt{\lambda_i}$.

Now, if we take Σ_M as the submatrix of Σ_D that has in its diagonal the M greater σ, and U_M and V_M as the matrices formed with the first M columns of U_D and V_D, respectively, we can express the best approximation of X' with rank M as

$$\hat{W}^2\hat{H}' = U_M\Sigma_M V_M^T$$

and as a consequence, we can choose

$$\hat{W}^2 = U_M T^{-1}, \tag{2.3.2}$$

$$\hat{H}' = T\Sigma_M V_M^T,$$

with T an arbitrary non-singular $M \times M$ matrix. And we have obtained the expressions of the optimal B^2 and W^2 in (2.3.1) and (2.3.2).

We will now try to demonstrate the equivalence between PCA and the transformations performed in the hidden layer of such autoencoder. To this purpose, we must prove that the covariance matrix of the hidden layer activations is the best M-rank approximation of the original input data covariance matrix (see [15]).

Let $\overline{\mu}_X$ be the mean of the training input vectors, $X^1, \ldots, X^N$, that is, $\overline{\mu}_X = \frac{1}{N}XU$ and $\overline{\mu}_H = \frac{1}{N}HU$ and $\overline{\mu}_Y = \frac{1}{N}YU$ the mean of the respective hidden and output layer vectors. Replacing the optimal biases vector, $\hat{B}^2$, in Y we have

$$\begin{aligned}
\overline{\mu}_Y &= \frac{1}{N}YU = \frac{1}{N}\left(W^2H + B^2U^T\right)U \\
&= \frac{1}{N}\left(W^2H + \frac{1}{N}\left(X - W^2H\right)UU^T\right)U \\
&= \frac{1}{N}\left(W^2H + X - W^2H\right)U \\
&= \frac{1}{N}XU = \overline{\mu}_X;
\end{aligned}$$

that is, the mean input and output vectors are equal. Also we can rewrite $X' = X - \overline{\mu}_X U^T$ and $H' = H - \overline{\mu}_H U^T$, so that X' and H' represent the input and hidden layer vectors after subtracting the means.

Finally, it is possible to prove that the output vector covariance matrix is the best M rank approximation of the input vectors covariance matrix, and so, the autoassociative multilayer perceptron is equivalent to PCA. We will use that V_D is unitary, so $V_D V_D^T = V_D^T V_D = I$, then we have

$$C_X = X'X'^T = U_D\Sigma_D V_D^T V_D \Sigma_D U_D^T = U_D\Sigma_D^2 U_D^T$$

and therefore

$$
\begin{aligned}
C_Y &= \left(Y - \overline{\mu}_Y U^T\right)\left(Y - \overline{\mu}_Y U^T\right)^T = \left(Y - \overline{\mu}_X U^T\right)\left(Y - \overline{\mu}_X U^T\right)^T \\
&= \left(Y - \frac{1}{N} X U U^T\right)\left(Y - \frac{1}{N} X U U^T\right)^T = \left(Y - X + X'\right)\left(Y - X + X'\right)^T \\
&= \left(W^2 H + B^2 U^T - X + X'\right)\left(W^2 H + B^2 U^T - X + X'\right)^T \\
&= \left(W^2 H + \frac{1}{N}\left(X - W^2 H\right) U U^T - X + X'\right)\left(W^2 H + \frac{1}{N}\left(X - W^2 H\right) U U^T - X + X'\right)^T \\
&= \left(W^2 H' - X' + X'\right)\left(W^2 H' - X' + X'\right)^T = \left(W^2 H'\right)\left(W^2 H'\right)^T \\
&= U_M \Sigma_M V_M^T V_M \Sigma_M^T U_M^T = U_M \Sigma_M^2 U_M^T.
\end{aligned}
$$

See [15] for more details.

2.3.2 Multiple hidden layer autoencoders

All the previously seen properties are independent of the hidden units activation function F. If the hidden units have linear activation functions, it can be proved that the error function has a unique global minimum. In the two hidden layers perceptrons the situation is similar. However, this changes for three hidden layers perceptrons (see [11]).

Consider a perceptron with a D linear unit input layer, a first hidden layer with sigmoid units, a second hidden layer with M linear units, a third hidden layer with sigmoid units, and an output layer with D linear units. The network is trained the same way as before. This net can be seen as two successive correspondences F_1 and F_2. F_1 projects the original D dimensional vectors in a M-dimensional subspace defined by the activations of the second hidden layer units. This correspondence is non-linear because of the sigmoid activations of the first hidden layer units. In the same way, F_2 gives a correspondence between the M-dimensional space and the D-dimensional original space.

The described network performs, in fact, a non-linear PCA, that contains linear PCA as a particular case, but is not limited to linear transformations. The problem is that now the error function is not a quadratic function of the network parameters, which produces a more complex optimization problem possibly with local minima.

2.3.3 Infomax principle

Instead of minimizing the mean squared error, we can choose other reconstruction criteria. One of them is the *infomax* principle, that consists in preserving at the output as much information from the input as possible, or, in Information Theory terms, maximize the mutual information $\mathbb{I}\left(X, Y\right)$ between the input random variable X and its reconstruction Y.

As is detailed in the Appendix, mutual information is defined as $\mathbb{I}(X,Y) = \mathbb{H}(X) - \mathbb{H}(X|Y)$. Given that X follows a distribution $q(X)$ that is unknown to us and is not influenced by the autoencoder parameters, the infomax principle is reduced to maximizing $-\mathbb{H}(X|Y)$:

$$\begin{aligned} argmax_{W^1,B^1,W^2,B^2}\left[\mathbb{I}(X,Y)\right] &= argmax_{W^1,B^1,W^2,B^2}\left[-\mathbb{H}(X|Y)\right] \\ &= argmax_{W^1,B^1,W^2,B^2}\left[\mathbb{E}_{q(X,Y)}\left[\log q(X|Y)\right]\right]. \end{aligned}$$

The Kullback-Leibler divergence,

$$\mathbb{D}_{KL}(p||q) = \sum_k p_k \log\left(\frac{p_k}{q_k}\right),$$

also detailed in the Appendix, for two distributions p and q, has the property

$$\mathbb{D}_{KL}(q||p) \geq 0$$

and, in particular,

$$\mathbb{D}_{KL}(q(X,Y)||p(X|Y)) \geq 0.$$

Using it we have that

$$\begin{aligned} 0 &\leq \mathbb{D}_{KL}(q(X,Y)||p(X|Y)) \\ &= \mathbb{E}_{q(X,Y)}\left[\log\frac{q(X,Y)}{p(X|Y)}\right] \\ &= \mathbb{E}_{q(X,Y)}\left[\log q(X,Y)\right] - \mathbb{E}_{q(X,Y)}\left[\log p(X|Y)\right], \end{aligned}$$

where $\mathbb{E}_{q(X,Y)}[\cdot]$ is the expected value over the distribution $q(X,Y)$, and so

$$\begin{aligned} \mathbb{E}_{q(X,Y)}\left[\log p(X|Y)\right] &\leq \mathbb{E}_{q(X,Y)}\left[\log q(X|Y)\right] \\ &= -\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} q(X,Y)\log q(X|Y)\,dx\,dy \\ &= -\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} q(X|Y)q(Y)\log q(X|Y)\,dx\,dy \\ &= -\int_{-\infty}^{\infty} q(Y)\mathbb{H}(X|Y)\;dy \\ &= -\mathbb{H}(X|Y) \end{aligned}$$

that is, maximizing $\mathbb{E}_{q(X,Y)}[\log p(X|Y)]$ we are somehow maximizing a lower bound on $-\mathbb{H}(X|Y)$ and so we maximize the mutual information. If there exist some parameters $\hat{W}^1$, $\hat{B}^1$, $\hat{W}^2$, $\hat{B}^2$ that give $p(X|Y,\hat{W}^1,\hat{W}^2,\hat{B}^1,\hat{B}^2) = q(X|Y)$, then we can maximize the exact mutual information, writing the criterion as

$$max_{\hat{W}^1,\hat{B}^1,\hat{W}^2,\hat{B}^2}\left[\mathbb{E}_{q(X,Y,\hat{W}^1,\hat{B}^1,\hat{W}^2,\hat{B}^2)}\left[\log p\left(X|Y,\hat{W}^1,\hat{B}^1,\hat{W}^2,\hat{B}^2\right)\right]\right]$$

or, as Y is determined by X, W^1, B^1, W^2 and B^2, we can write

$$max_{\hat{W}^1,\hat{B}^1,\hat{W}^2,\hat{B}^2}\left[\mathbb{E}_{q(X)}\left[\log p\left(X|Y,\hat{W}^1,\hat{B}^1,\hat{W}^2,\hat{B}^2\right)\right]\right]$$

Since we don't know $q(X)$, but have a sample, it is possible to use the sample expected value as an estimator of the expected value. For instance, if input values X are in $\mathbb{R}^D$, we suppose that $X|Y \sim \mathcal{N}(Y, \sigma^2 I)$, which leads us to minimize the function $\mathbb{E}\left[\|X - Y\|^2\right]$, the mean squared error function traditionally used in autoencoders (see [21] for more information).

2.3.4 Sparse autoencoders

Before, we assumed that M, the number of hidden units of the autoencoder, was smaller than D, the number of units in the input and output layers. This is necessary to force the autoencoder to find the underlying structure in the data, but we can impose a sparsity condition ([18]) to force it to find this structure even when $M > D$.

If the hidden layer units use sigmoid activation functions, we will say that a unit is active if its output is close to 1, and that it is inactive if its output is close to 0. If we use an hyperbolic tangent function, the unit is inactive if its output is close to -1. The sparsity condition is met when the units are inactive for most of the patterns. If the activation vector of the hidden units for the pattern n is H^n then, the mean activation for the hidden unit m over the training set is

$$\hat{\rho}_m = \frac{1}{N} \sum_{n=1}^{N} H_m^n.$$

We would like that, for all the hidden units

$$\hat{\rho}_m \simeq \rho,$$

with ρ the sparsity parameter with a value close to zero, for example 0.05. To achieve this we can add a penalty term to the global error function over all the training set so it penalizes the $\hat{\rho}_m$ that are far from ρ. There are many valid options for this penalty term; one of them is

$$\sum_{m=1}^{M} \mathbb{D}_{KL}\left(\rho || \hat{\rho}_m\right) = \sum_{m=1}^{M} \left(\rho \log \frac{\rho}{\hat{\rho}_m} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_m}\right)$$

where the Kullback-Leibler divergence, that indicates how different two distributions are, operates in this case over a ρ mean Bernoulli and a $\hat{\rho}_m$ mean Bernoulli.

This penalty has the property that $\mathbb{D}_{KL}\left(\rho || \hat{\rho}_m\right) = 0$ if $\hat{\rho}_m = \rho$, and increases monotonically as $\hat{\rho}_m$ and ρ move away, being ∞ when $\hat{\rho}_m$ is 0 or 1. The penalty term is usually applied with a weight factor.

The local error function for pattern n is

$$\begin{aligned} E_{local}\left(W^1, W^2, B^1, B^2, X^n\right) &= \frac{1}{2}\left\|X^n - Y^n\right\|^2 \\ &= \frac{1}{2}\left\|X^n - \left(W^2 H^n + W^2\right)\right\|^2 \\ &= \frac{1}{2}\left\|X^n - \left(W^2 F\left(W^1 X^n + B^1\right) + B^2\right)\right\|^2, \end{aligned}$$

from where we get the global error function

$$E_{global}\left(W^1, W^2, B^1, B^2\right) = \frac{1}{N}\sum_{n=1}^{N} E_{local}\left(W^1, W^2, B^1, B^2, X^n\right),$$

to which we add a weight decay penalty term, so we get

$$E_{decay}\left(W^1, W^2, B^1, B^2, \lambda\right) = E_{global}\left(W^1, W^2, B^1, B^2\right) + \frac{\lambda}{2}\sum_{m=1}^{M}\left(\|W_m^1\|^2 + \|W_m^2\|^2\right),$$

that is the usual error function employed in a single hidden layer perceptron. We can now add the new sparsity penalty term to get

$$E_{sparse}\left(W^1, W^2, B^1, B^2, \lambda, \rho, \beta\right) = E_{decay}\left(W^1, W^2, B^1, B^2, \lambda\right) + \beta\sum_{m=1}^{M}\mathbb{D}_{KL}\left(\rho||\hat{\rho}_m\right).$$

Training the network with this error function requires previous knowledge of the mean activations $\hat{\rho}_m$, so it is necessary to perform a previous processing of all the training patterns to calculate these activations. The gradient descent method is well known for non-sparse perceptrons, and in our case the only components of the gradient affected by the sparsity term are the ones differentiated with respect to W^1 and B^1, because the term does not depend on W^2 or B^2. We get the component differentiated with respect to W^1 as

$$\begin{aligned}
\frac{\partial E_{sparse}}{\partial W^1} &= \frac{\partial E_{decay}}{\partial W^1} + \beta\sum_{m=1}^{M}\frac{\partial \mathbb{D}_{KL}\left(\rho||\hat{\rho}_m\right)}{\partial W^1} \\
&= \frac{\partial E_{decay}}{\partial W^1} + \beta\sum_{m=1}^{M}\frac{\partial}{\partial W^1}\left(\rho\log\frac{\rho}{\hat{\rho}_m} + (1-\rho)\log\frac{1-\rho}{1-\hat{\rho}_m}\right) \\
&= \frac{\partial E_{decay}}{\partial W^1} + \beta\sum_{m=1}^{M}\frac{\partial \hat{\rho}_m}{\partial W^1}\left(\frac{-\rho}{\hat{\rho}_m} + \frac{1-\rho}{1-\hat{\rho}_m}\right),
\end{aligned}$$

the component differentiated with respect to B^1 as

$$\begin{aligned}
\frac{\partial E_{sparse}}{\partial B^1} &= \frac{\partial E_{decay}}{\partial B^1} + \beta\sum_{m=1}^{M}\frac{\partial \mathbb{D}_{KL}\left(\rho||\hat{\rho}_m\right)}{\partial B^1} \\
&= \dots \\
&= \frac{\partial E_{decay}}{\partial B^1} + \beta\sum_{m=1}^{M}\frac{\partial \hat{\rho}_m}{\partial B^1}\left(\frac{-\rho}{\hat{\rho}_m} + \frac{1-\rho}{1-\hat{\rho}_m}\right),
\end{aligned}$$

and the components not affected by the sparsity as

$$\frac{\partial E_{sparse}}{\partial W^2} = \frac{\partial E_{decay}}{\partial W^2},$$

$$\frac{\partial E_{sparse}}{\partial B^2} = \frac{\partial E_{decay}}{\partial B^2},$$

finally

$$\frac{\partial \hat{\rho}_m}{\partial W^1} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial H_m^n}{\partial W^1} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial W^1}F\left(W_m^1 X^n + B_m^1\right) = \frac{1}{N}\sum_{n=1}^{N} X^n F'\left(W_m^1 X^n + B_m^1\right),$$

$$\frac{\partial \hat{\rho}_m}{\partial B^1} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial H_m^n}{\partial B^1} = \frac{1}{N}\sum_{n=1}^{N}\frac{\partial}{\partial B^1}F\left(W_m^1 X^n + B_m^1\right) = \frac{1}{N}\sum_{n=1}^{N} F'\left(W_m^1 X^n + B_m^1\right).$$

2.3.5 Denoising autoencoders

We have other options to force a useful feature extraction from the data. One of them is trying to reconstruct the original data from a corrupted version of them, what, we hope, will force the autoencoder to learn a robust and stable representation of the data, preserving the underlying structure and ignoring the noise.

To this purpose, we must corrupt each of the input data vectors X^n to get a vector $\tilde{X}^n$ through a stochastic mapping, while the targets remain untouched. Then, we will proceed like with the normal autoencoder, but having

$$H^n = F(W^1 \tilde{X}^n + B^1)$$

instead of

$$H^n = F(W^1 X^n + B^1)$$

as we had before.

Examples of different data corruption processes are

- Gaussian noise: $\tilde{X^n} \sim \mathcal{N}\left(X^n, \sigma^2 I\right)$.
- Masking noise: change a fraction of vector X^n elements, randomly selected, to 0.
- Salt-pepper noise: change a fraction of vector X^n elements, randomly selected, to the maximum or minimum possible values following a distribution $\mathcal{B}\left(\frac{1}{2}\right)$.

The masking noise is specially interesting, as it can be understood as representing data with the missing values and forcing the autoencoder to reconstruct the information. Masking and salt-pepper noise can be useful to emphasize (or attenuate) the missing values reconstruction function of the autoencoder. If the error function used is the mean squared error, we can add some parameters α, for the corrupted components, and β, for the intact ones, and modify the function so we get

$$E_{local} = \frac{\alpha}{2}\sum_{d \in J\left(\tilde{X^n}\right)}(X_d^n - Y_d^n)^2 + \frac{\beta}{2}\sum_{d \notin J\left(\tilde{X}\right)}(X_d^n - Y_d^n)^2,$$

where $J\left(\tilde{X^n}\right)$ denotes the indices of the corrupted components of the transformed pattern $\tilde{X}^n$.

The geometric interpretation of this process is the following: assume the data in a high-dimensional space tend to be close to a non-linear low-dimensional subspace. By corrupting the data, they will move away from this subspace, and the autoencoder will learn to move these uncommon corrupted data back to the vicinity of the subspace, where the probability of finding data is higher. Thus, the denoising autoencoder tries to define and learn a manifold and to give a representation of the data that captures the main variations along the manifold. The result is that we obtain a more robust representation of the data, which captures the useful structure of the input distribution, given that the obtained representation allows a good reconstruction even when high levels of noise are present.

2.3.6 Maximal activation pattern

Once we have a trained autoencoder by any of the previous methods, we may want to identify which pattern would produce a highest hidden unit activation. For pattern n, hidden unit m calculates

$$H_m^n = F\left(W_m^1 X^n + B_m^1\right).$$

We will obtain the function computed by the hidden unit, H_m, a function of W_m^1, as a two-dimensional image, and will find the input $\tilde{X}$ that produces a maximum activation in unit H_m. To have a non-trivial solution, we must impose a condition to input $\tilde{X}$:

$$\|\tilde{X}\|^2 = \sum_{d=1}^{D} \tilde{X}_d^2 \leq 1$$

and maximizing with this condition, using the Lagrange multipliers method, we have to find the critical point of

$$\Lambda_m = F\left(W_m^1 \tilde{X} + B_m^1\right) + \lambda_m \left(1 - \|\tilde{X}\|^2\right)$$

If we use a sigmoid activation function, that is close to linear around 0, we have

$$\frac{\partial}{\partial \tilde{X}} \Lambda_m \simeq W_m^1 - 2\lambda_m \tilde{X},$$

that is 0 iff

$$\tilde{X} = \frac{W_m^1}{2\lambda_m}.$$

Also

$$\frac{\partial}{\partial \lambda_m} \Lambda_m = 1 - \|\tilde{X}\|^2,$$

that is 0 iff

$$\|\tilde{X}\|^2 = 1.$$

And, using both conditions, we get

$$\tilde{X} = \frac{W_m^1}{\|W_m^1\|}$$

so the pattern that maximizes the hidden unit activation is that composed by the unit normalized weights.

2.4 Boltzmann machines

An alternative to the autoencoders are the Boltzmann machines. Boltzmann machines ([25]) are stochastic recurrent neural networks that are capable to detect complex patterns in the data and to solve complex combinatorial problems. These machines are not much used in practice because of the high cost of their training, but there exists a simplification, the restricted Boltzmann machines ([23], [26]), that give good results in practice while making training more feasible computationally.

Depending on how we operate with the Boltzmann machine, it will solve a search problem or a learning problem: if the weights between its units are fixed representing a cost function, the machine will find those input patterns that have a low cost according to this function; if the weights are left unclamped and change during training, the machine will learn to generate patterns following the input pattern distribution ([23]).

2.4.1 Definition

As was said before, the Boltzmann machine is a recurrent neural network, so all the units are connected to each other. The weight of the link between unit i and unit j is W_{ij}. Two additional conditions of the Boltzmann machines are $W_{ii} = 0\ \forall i$ (a unit cannot be connected to itself) and $W_{ij} = W_{ji}\ \forall i, j$ (connections are symmetric). The weight matrix for the connections is denoted W. The bias for unit i is B_i and the bias vector is B.

The Boltzmann machine units are divided in two groups, the visible units, that we will denote with the vector $V = \{V_1, \dots, V_D\}$, and the hidden units, that we will denote with the vector $H = \{H_1, \dots, H_M\}$. To simplify notation in following calculations, we will use $U = \{V, H\} = \{V_1, \dots, V_D, H_1, \dots, H_M\} = \{U_1, \dots, U_{D+M}\}$ for the set of all the network units. The visible units are both the input and the output of the Boltzmann machine, so, if we feed the input pattern $X = \{X_1, \dots, X_D\}$ to our network, we will have that, initially, $V = X$. Each unit can only take value 0 or 1 in a given instant.

We define now a function that maps each possible state of the network to an energy

$$E(U|W, B) = -\sum_i \sum_{j \neq i} W_{ij} U_i U_j - \sum_i B_i U_i;$$

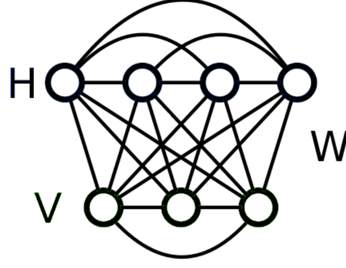

Figure 2.4.1: Boltzmann machine

if we suppose that the states follow a Boltzmann distribution with $\beta = \frac{1}{K_B T}$, then we have that the probability of a state U is given by

$$P(U|W,B) = \frac{e^{-\beta E(U|W,B)}}{\sum_{\check{U}} e^{-\beta E(\check{U}|W,B)}}$$

notice that we are dividing by a sum over all the possible states, 2^{M+D}.

From this, we can calculate that the probability for a unit to take value 1 given the values of the other units, U_i, is

$$\begin{aligned} P\left(U_i = 1|\overline{U}_i, W, B\right) &= \frac{P(U_i = 1, \overline{U}_i, W, B)}{P(\overline{U}_i, W, B)} \\ &= \frac{P(U_i = 1, \overline{U}_i|W, B)P(W,B)}{P(\overline{U}_i|W,B)P(W,B)} \\ &= \frac{P(U_i = 1, \overline{U}_i|W,B)}{P(U_i = 1, \overline{U}_i|W,B) + P(U_i = 0, \overline{U}_i|W,B)} \\ &= \frac{e^{\beta\Xi}}{e^{\beta\Xi} + e^{\beta\Xi}e^{-\beta\left(\sum_{j\neq i} W_{ij}U_j + B_i\right)}} \\ &= \frac{1}{1 + e^{-\beta\left(\sum_{j\neq i} W_{ij}U_j + B_i\right)}}, \end{aligned}$$

where

$$\Xi = \sum_{k\neq i}\sum_{j\neq i,k} W_{kj}U_kU_j + \sum_{k\neq i} B_kU_k + \sum_{j\neq i} W_{ij}U_j + B_i$$

and $\overline{U}_i$ denotes the states of all units but U_i.

It can be proved that, when $T = 0$, and so $\beta = \infty$, the activation of the unit becomes deterministic and the Boltzmann machine is reduced to its non-stochastic version, a Hopfield network

$$T = 0 \Rightarrow P(U_i = 1|\overline{U}_i, W, B) = \begin{cases} 1 & \text{if } \sum_{j\neq i} W_{ij}U_j + B_i > 0 \\ 0 & \text{if } \sum_{j\neq i} W_{ij}U_j + B_i < 0 \end{cases};$$

see [23], [26] and [28] for more details.

2.4.2 Learning

Given a set of independent training patterns, learning consists on finding the weights and biases that make the patterns correspond to low energy states, or, in other words, finding the weights and biases that define a distribution that make the states induced by the training patterns the most probable ones.

We can learn the Boltzmann machine parameters from the training patterns using the maximum likelihood method. Given a set of N training vectors, we have the following likelihood function

$$\mathcal{L}\left(W,B\right)=\prod_{n=1}^{N}P\left(V^{n}|W,B\right)=\prod_{n=1}^{N}\left(\sum_{H}P\left(V^{n},H|W,B\right)\right),$$

where we marginalize over the hidden units states, and the log-likelihood function

$$\begin{aligned}\log\mathcal{L}\left(W,B\right)&=\sum_{n=1}^{N}\log P(V^{n}|W,B)\\&=\sum_{n=1}^{N}\log\sum_{H}P(V^{n},H|W,B)\\&=\sum_{n=1}^{N}\log\sum_{H}\frac{e^{-\beta E(V^{n},H|W,B)}}{\sum_{\check{V}}e^{-\beta E(\check{V},H|W,B)}}\\&=\sum_{n=1}^{N}\left(\log\sum_{H}e^{-\beta E(V^{n},H|W,B)}-\log\sum_{H}\sum_{\check{V}}e^{-\beta E(\check{V},H|W,B)}\right).\end{aligned}$$

We get the gradient by differentiation of the log-likelihood function. The calculations for the derivative with respect to W_{ij} are shown next, the process to get the

derivative with respect to B_i is similar. We have

$$\begin{aligned}
\frac{\partial}{\partial W_{ij}} \log \mathcal{L}(W,B) &= \sum_{n=1}^{N}\left(\frac{\partial}{\partial W_{ij}} \log \sum_{H} e^{-\beta E(V^n,H|W,B)} - \frac{\partial}{\partial W_{ij}} \log \sum_{H}\sum_{\check{V}} e^{-\beta E(\check{V},H|W,B)}\right) \\
&= \sum_{n=1}^{N}\left(\frac{\sum_H \frac{\partial}{\partial W_{ij}}\left(-\beta E(V^n,H|W,B)\right) e^{-\beta E(V^n,H|W,B)}}{\sum_H e^{-\beta E(V^n,H|W,B)}}\right. \\
&\quad \left. - \frac{\sum_H \sum_{\check{V}} \frac{\partial}{\partial W_{ij}}\left(-\beta E(\check{V},H|W,B)\right) e^{-\beta E(\check{V},H|W,B)}}{\sum_H \sum_{\check{V}} e^{-\beta E(\check{V},H|W,B)}}\right) \\
&= \sum_{n=1}^{N}\left(\sum_{H}\left[\frac{\partial}{\partial W_{ij}}\left(-\beta E(V^n,H|W,B)\right)\right]\frac{e^{-\beta E(V^n,H|W,B)}}{\sum_H e^{-\beta E(V^n,H|W,B)}}\right. \\
&\quad \left. - \sum_{H}\sum_{\check{V}}\left[\frac{\partial}{\partial W_{ij}}\left(-\beta E(\check{V},H|W,B)\right)\right]\frac{e^{-\beta E(\check{V},H|W,B)}}{\sum_H \sum_{\check{V}} e^{-\beta E(\check{V},H|W,B)}}\right) \\
&= \sum_{n=1}^{N}\left(\sum_{H}\left[\frac{\partial}{\partial W_{ij}}\left(-\beta E(V^n,H|W,B)\right)\right] P(H|\{V^n\},W,B)\right. \\
&\quad \left. - \sum_{H}\sum_{\check{V}}\left[\frac{\partial}{\partial W_{ij}}\left(-\beta E(\check{V},H|W,B)\right)\right] P(H,\check{V}|W,B)\right) \\
&= \sum_{n=1}^{N}\left(\left\langle\frac{\partial}{\partial W_{ij}}\left(-\beta E(V^n,H|W,B)\right)\right\rangle_d - \left\langle\frac{\partial}{\partial W_{ij}}\left(-\beta E(\check{V},H|W,B)\right)\right\rangle_m\right)
\end{aligned}$$

and so

$$\frac{\partial \log \mathcal{L}(W,B)}{\partial W_{ij}} = \sum_{n=1}^{N} -\beta\left(\langle U_i U_j\rangle_d - \langle U_i U_j\rangle_m\right);$$

similarly, we have for the bias partials

$$\frac{\partial \log \mathcal{L}(W,B)}{\partial B_i} = \sum_{n=1}^{N} -\beta\left(\langle U_i\rangle_d - \langle U_i\rangle_m\right)$$

where $\langle\cdot\rangle_P$ is the expected value over distribution P, and $d \equiv P(H|\{V^n\},W,B)$, $m \equiv P(U|W,B)$, that represent the probability of the hidden units when the visible units are clamped to the input pattern (we say then that the network is in positive phase, using the data, hence the d), and the probability of all the units when none of them is clamped (we say then that the network is in negative phase, using the model, hence the m). See [23] for more details.

With the training, the machine is expected to reach an equilibrium as the state distribution converges. The stochastic part of the Boltzmann machines, given by the activation function of the units, is a way of escaping from local minima during the gradient descent.

It is possible to improve the learning algorithm using the temperature T in the same way as in the simulated annealing method ([25]). Making the temperature decrease progressively until the Boltzmann machine becomes a Hopfield network we make big aleatory perturbations in the beginning, when we are far from the global minimum, and small perturbations near the end of the process, when we expect to be close to the global minimum.

Boltzmann machines can work only with visible units, but are more powerful when they have hidden units that work as latent variables and allow the network to model vector distributions that cannot be modelled only with interactions between pairs of visible units.

2.4.3 Difficulties

We have a closed formula for the learning rule, but, in practice, it is impossible to calculate the gradient because of the normalization coefficient that appears in $P(U|W,B)$, that requires calculating a sum over all the possible configurations of the machine, whose number grows exponentially with the number of units. As an alternative to the exact calculations, Gibbs sampling is proposed to obtain an estimation of the gradient.

The procedure consists in estimating the gradient using only a few samples selected randomly, called the *negative particles*, instead of all the possible values. Usually, the method works adequately even if only one sample is used per iteration.

Another observation is that learning with many hidden units is very slow, because it is difficult to reach the equilibrium distribution, specially when it is multimodal, which is usually the case when the visible units are unclamped.

A simplification of the model that reduces in some way all the problems, is the restricted Boltzmann machine.

2.4.4 Restricted Boltzmann machines

The restricted Boltzmann machines are Boltzmann machines with one hidden layer and one visible layer that have no connections between units of the same layer, that is, they have no connections between two hidden units, or between two visible units. With this conditions, hidden units are independent from each other when an input pattern is clamped to the visible units.

The energy function is now

$$E(V,H|W,B) = -\sum_{i=1}^{D}\sum_{j=1}^{M} W_{i,j}V_iH_j - \sum_{i=1}^{D} B_iV_i - \sum_{j=1}^{M} B_{j+D}H_j$$

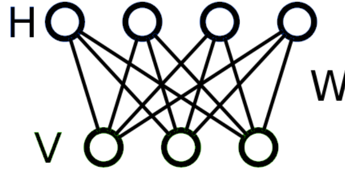

Figure 2.4.2: Restricted Boltzmann machine

and the probability for each state is

$$P(V,H|W,B)=\frac{e^{-\beta E(V,H|W,B)}}{\sum_{\check{V},\check{H}}e^{-\beta E(\check{V},\check{H}|W,B)}}.$$

When the visible units are clamped to the input pattern, each hidden unit is independent from the rest, and we can sum the probabilities in the hidden units to get

$$\begin{aligned}P(V|W,B)&=\prod_{j=1}^{M}\big(P(V,H_j=1|W,B)+P(V,H_j=0|W,B)\big)\\&=\frac{e^{\beta\sum_{i=1}^{D}B_iV_i}\prod_{j=1}^{M}\left(1+e^{\beta(B_{j+D}+\sum_{i=1}^{D}W_{i,j}V_i)}\right)}{\sum_{\check{V}}e^{-\beta E(\check{V}|W,B)}}.\end{aligned}$$

Arguing as done before, the gradient for the restricted Boltzmann machine is

$$\begin{aligned}\frac{\partial\log\mathcal{L}\left(W,B\right)}{\partial W_{ij}}&=\langle V_iH_j\rangle_d-\langle V_iH_j\rangle_m\\\frac{\partial\log\mathcal{L}\left(W,B\right)}{\partial B_i}&=\langle U_i\rangle_d-\langle U_i\rangle_m\end{aligned}$$

The training algorithm for the restricted Boltzmann machines is called *contrastive divergence gradient* ([26], [23]) and it consists in alternating phases where the hidden units states are recalculated from the visible units states, and phases where the visible units states are recalculated from the hidden units states. This method is biased, but it produces acceptable results in practice, even when only one Gibbs sample is used per iteration.

Algorithm 2 Contrastive divergence gradient

for $n = 1 \ldots N$ **do**
  Clamp the visible units to the n-th input pattern value, $pvstates$.
  Calculate the activations of the hidden units, $phstates$.
  Calculate the activations of the visible units, $nvstates$.
  Calculate the activations of the hidden units, $nhstates$.
  Update weights, the gradient is $pvstates * phstates - nvstates * nhstates$.
end for

Chapter 3

Deep networks

As we saw previously, it is known ([8], [9]) that a deep neural network with many hidden layers can perform, with a relatively small number of units per layer, functions for which a shallow network would need a very high number of units per layer.

However, although deep networks are more powerful and expressive than shallow networks, they have been almost totally ignored until only a few years ago. The reason for that is, as we mentioned, apart from convolutional networks, that there is no known efficient algorithm to train such networks, because the classical backpropagation algorithm with gradient descent losses all its effectiveness when the networks has more than a couple of layers, and trains only the last layers, with the rest stuck in a state close to the initial random initialization.

Apart from convolutional nets, that are like *pruned* multilayer perceptrons, all the deep networks we will discuss have two components. The first one is a series of hidden layers that are trained independently with a greedy unsupervised algorithm, and whose function is to obtain a highly abstract and robust representation of the input data, that gets better with each added layer. The second component is a shallow, but possibly multilayer, perceptron that performs the classification or regression task we want to solve, like, for example, a softmax perceptron, and that is connected to the last of the hidden layers mentioned before.

It is also possible to use only the first component, the stack of hidden layers, with another classifier, not necessarily based on neural networks, like a support vector machine, and also get a significant improvement of the classifier results thanks to the pre-processing of the stack of layers. Many experimental results show that classifiers get better results when working with high level representations of the data like the one generated by such stacks of hidden layers.

The training algorithm of such deep networks has two steps. The first is a greedy unsupervised layer by layer pre-training that initializes the connection weights better than the usual random initialization that, most probably, will put the network in a state where it is easy get stuck in local minima of the error function. Apart

from getting a better initialization, it is extremely interesting the fact that this pre-training is totally unsupervised. The second step is performed after connecting the final perceptron to the last layer of the stack, and consists in a traditional supervised backpropagation training, called sometimes *fine-tuning* in this case, that configures the perceptron to perform the desired task and also makes the final minor adjustments to the hidden layers.

Although the combination of traditional deep multilayer perceptrons and backpropagation may seem to be a bad choice, we will fist of all see in detail the reasons for this bad performance and consider some alternatives that could make possible the use of relatively deep perceptrons with a supervised backpropagation training.

3.1 Deep multilayer perceptrons

Before the development of new greedy pre-training algorithms (see [27]), it appeared that deep neural networks could not be successfully trained. Now we know that a greedy pre-training process after the random initialization will put the network in a state where it will perform well, but, why a random initialization and backpropagation are not just enough to properly train a deep network?

[10] studies the impact of the cost function, the activation functions of the hidden layers, and the weights initialization in the performance of the backpropagation algorithm. Let's take a closer look.

3.1.1 Cost function

As cost functions, we usually consider for multilayer perceptrons the traditional quadratic cost function $\|Y - X\|^2$, where X is the input pattern and Y the output target. As is mentioned in [10], the logistic regression or conditional log-likelihood cost function $-\log p\,(Y|X)$ usually works better for classification tasks, as it seems that the quadratic cost function is more prone to have plateaus that would make the learning process much more difficult.

3.1.2 Activation function

On the activation function, we consider the sigmoid function $\frac{1}{1+e^{-x}}$, the hyperbolic tangent $\tanh x$, and the softsign function $\frac{x}{1+|x|}$. The softsign is similar to the hyperbolic tangent but the tails converge quadratically (instead of exponentially) to the asymptotes 1 and -1. The softsign is also differentiable, as

$$\frac{\partial}{\partial x}\frac{x}{1+|x|} = \begin{cases} \frac{1}{(1-x)^2} \to_{x\to 0^-} 1 & \text{if } x < 0 \\ 1 & \text{if } x = 0 \\ \frac{1}{(1+x)^2} \to_{x\to 0^+} 1 & \text{if } x > 0 \end{cases},$$

and so it is suitable as activation function.

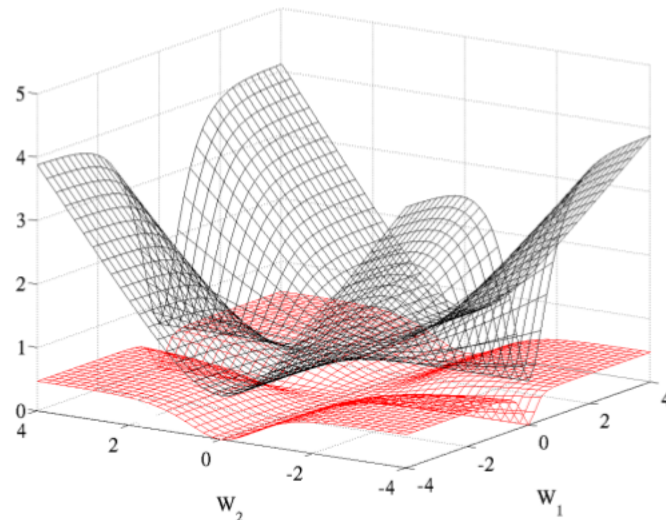

Figure 3.1.1: Log-likelihood (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer. Source: [10]

As noted in [10], when training a deep network with sigmoid activation functions, the activation values of the last hidden layer quickly saturate to 0, while other layers take a mean activation value greater than 0.5 that increases as we go from the input layer to the output layer. It seems that pushing the outputs to 0 puts them in a saturation regime which prevents gradients to flow backward and cuts off the lower layers learning. If the number of layers is small (around 2 or 3 layers), however, the network may eventually escape from the saturation regime.

As we can see in figure 3.1.2, top row, being symmetric around 0, the hyperbolic tangent does not have the saturation problems observed with the sigmoid function. In this case the layers saturate sequentially starting from the first hidden layer and ending with the last one. The behavior of the network is, apparently, better than with the sigmoid function.

In figure 3.1.2, bottom row, we note that the softsign has smoother asymptotes than the hyperbolic tangent, and the saturation does not occur sequentially, but all layers move slowly together towards larger weights. The final activation values obtained with this function are totally different from the ones obtained with the hyperbolic tangent.

In figure 3.1.3, top row, we notice that, for hyperbolic tangent activation, the lower layers have important saturation, while in the bottom row, with softsign activation, their activation values are concentrated around $(-0.6, -0.8)$ and $(0.6, 0.8)$ and the units do not saturate but are non-linear.

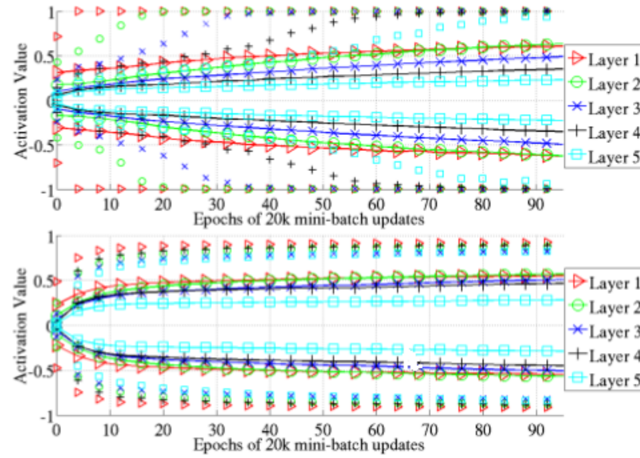

Figure 3.1.2: Top: 98 percentiles (markers alone) and standard deviation (solid lines with markers) of the distribution of the activation values for the hyperbolic tangent networks in the course of learning. We see the first hidden layer saturating first, then the second, etc. Bottom: 98 percentiles (markers alone) and standard deviation (solid lines with markers) of the distribution of activation values for the softsing during learning. Here the different layers saturate less and do so together. Source: [10]

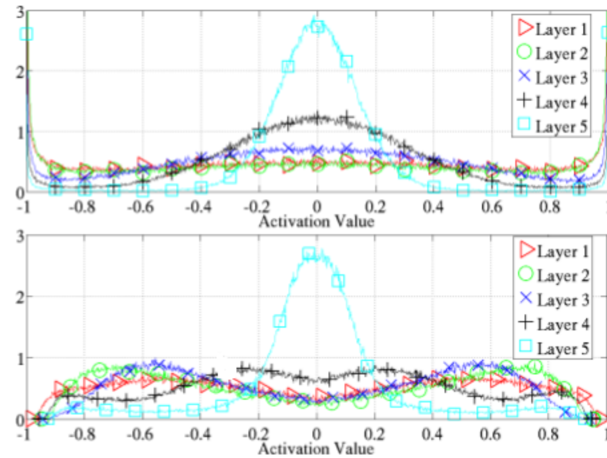

Figure 3.1.3: Activation values normalized histogram at the end of learning, averaged across units of the same layer and across 300 test examples. Top: activation function is hyperbolic tangent, we see important saturation of the lower layers. Bottom: activation function is softsing, we see many activation values around $(-0.6, -0.8)$ and $(0.6, 0.8)$ where the units do not saturate but are non-linear. Source: [10]

3.1.3 Initialization

We usually initialize the biases to 0 and the weights W_{ji} at each layer with the widely used heuristic

$$W_{ji} \sim \mathcal{U}\left(-\frac{1}{\sqrt{M}}, \frac{1}{\sqrt{M}}\right),$$

where M is the size of the previous layer. However, as we will see, this initialization creates some problems.

Given a multilayer perceptron with a symmetric activation function F with $F'(0) = 1$ (e.g. hyperbolic tangent or softsign) and cost function E, we write Z^l for the activation vector and S^l for the argument vector of the activation function of layer l, so that $S^l = Z^l W^l + B^l$ and $Z^{l+1} = F(S^l)$. Then we have

$$\frac{\partial E}{\partial W^l_{ji}} = \frac{\partial E}{\partial S^l_j} Z^l_i,$$

with

$$\begin{aligned}\frac{\partial E}{\partial S^l_j} &= \sum_k \frac{\partial E}{\partial S^{l+1}_k} \frac{\partial S^{l+1}_k}{\partial S^l_j} \\ &= \sum_k \frac{\partial E}{\partial S^{l+1}_k} \frac{\partial \left(Z^{l+1}_k W^{l+1}_{kj} + B^{l+1}_k\right)}{\partial S^l_j} \\ &= \sum_k \frac{\partial E}{\partial S^{l+1}_k} \frac{\partial \left(F(S^l_j) W^{l+1}_{kj} + B^{l+1}_k\right)}{\partial S^l_j} \\ &= \sum_k \frac{\partial F(S^l_j)}{\partial S^l_j} W^{l+1}_{kj} \frac{\partial E}{\partial S^{l+1}_k}.\end{aligned}$$

Lets ignore the bias to simplify the following calculations. On the other side, if we express the variances with respect to the input, output and weight initialization, we have

$$\begin{aligned}Var[Z^l] &= Var[F(Z^{l-1}W^{l-1})] \\ &= Var[F(F(Z^{l-2}W^{l-2})W^{l-1}] \\ &= \dots\end{aligned}$$

The hypothesis that we are in a linear regime, so $F(X) \simeq X$ at initialization gives us

$$Var[Z^l] \simeq Var[Z^0 W^0 W^1 W^2 \dots]$$

and then, considering that the weights are initialized independently, the input data variances are all $Var[X] \equiv Var[Z^0]$ and the means are zero, we have (see [10])

$$Var[Z^l] \simeq Var[X] \prod_{l'=0}^{l-1} M_{l'} Var[W^{l'}],$$

where $Var[W^{l'}]$ is the shared scalar variance of all weights at layer l', M_l is the width of layer l and X is the network input.

If the perceptron has L layers, then for the same reasons as before (see [10]), moving from the output to the input layer, we have

$$Var\left[\frac{\partial E}{\partial S^l}\right] \simeq Var\left[\frac{\partial E}{\partial S^L}\right]\prod_{l'=l}^{L} M_{l'+1}Var[W^{l'}]$$

and also, noting that $\mathbb{E}\left[\frac{\partial F}{\partial S^l}\right] \simeq \mathbb{E}\left[Z^l\right] \simeq 0$,

$$\begin{aligned} Var\left[\frac{\partial E}{\partial W^l}\right] &= Var\left[\frac{\partial E}{\partial S^l}Z^l\right] \\ &= Var\left[\frac{\partial E}{\partial S^l}\right]Var[Z^l] \\ &\simeq Var\left[\frac{\partial E}{\partial S^L}\right]\prod_{l'=l}^{L-1} M_{l'+1}Var[W^{l'}]Var[X]\prod_{l'=0}^{l-1} M_{l'}Var[W^{l'}]. \end{aligned}$$

If we assume, for simplicity, that all layers have the same width, M, and we have the same initialization for the weights then

$$Var\left[\frac{\partial E}{\partial S^l}\right] = (MVar[W])^{L-l}Var\left[\frac{\partial E}{\partial S^L}\right] \forall l,$$

$$Var\left[\frac{\partial E}{\partial W^l}\right] = (MVar[W])^{L}Var[X]Var\left[\frac{\partial E}{\partial S^L}\right] \forall l,$$

where we can see that the L and $L-l$ exponents may drive the variance of the back-propagated gradient to vanish or explode if we consider deep enough networks, which is what happens with the usual random initialization of the weights.

The variance of a variable $X \sim U[A,B]$ is $\frac{1}{12}(B-A)^2$, so the initialization described before gives a variance with the property

$$MVar[W] = \frac{1}{3},$$

that, being smaller than 1, this will cause the variance of the back-propagated gradient to vanish quickly. If we want information to keep flowing forward through the network, we would like that

$$Var[Z^l] = Var[Z^{l'}] \ \forall l, l',$$

which, in our case, is equivalent to

$$M_l Var[W^l] = 1 \ \forall l.$$

Similarly, if we want the error to back-propagate properly, we would like that

$$Var\left[\frac{\partial E}{\partial S^l}\right] = Var\left[\frac{\partial E}{\partial S^{l'}}\right] \ \forall l, l',$$

which is, in this case, equivalent to

$$M_{l+1}Var[W^l] = 1 \ \forall l.$$

Thus, we might want

$$Var[W^l] = \frac{2}{M_l + M_{l+1}} \ \forall l$$

as a compromise between the two constraints.

It is important, then, to use an appropriate initialization procedure that maintains stable back-propagated gradient variances. [10] suggests the following:

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{M_l + M_{l+1}}}, \frac{\sqrt{6}}{\sqrt{M_l + M_{l+1}}}\right]$$

In this case we have that

$$Var[W] = \frac{1}{12}\left(\frac{2\sqrt{6}}{\sqrt{M_l + M_{l+1}}}\right)^2 = \frac{2}{M_l + M_{l+1}}$$

which is the compromise constraint seen before, and that induces

$$M_l Var[W] = \frac{2}{1 + \frac{M_{l+1}}{M_l}}$$

wich is approximately 1 when $M_{l+1} \approx M_l$.

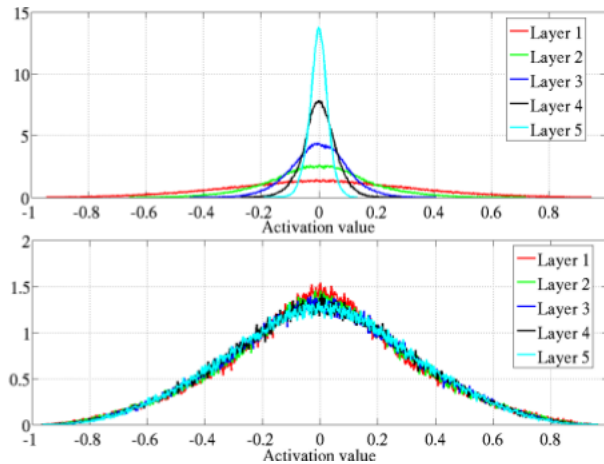

Figure 3.1.4: Activation values normalized histograms with hyperbolic tangent activations, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases of higher layers. Source: [10]

Figures 3.1.4 and 3.1.5 illustrate how the activations and gradients vanish with backpropagation if we use the standard weight initialization, but remain stable if the normalized initialization is used.

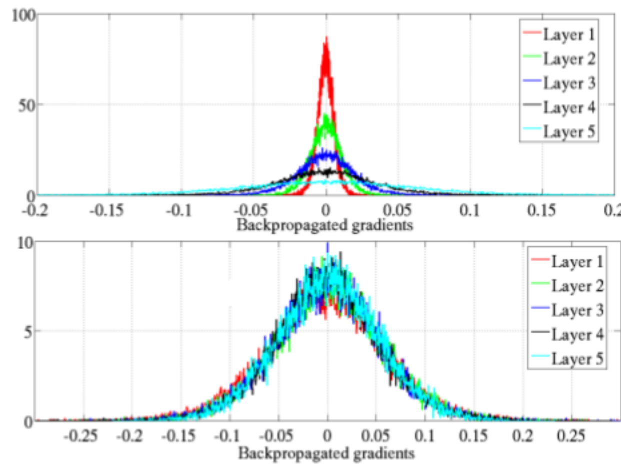

Figure 3.1.5: Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers. Source: [10]

3.2 Deep belief networks and stacked autoencoders

Previously, we mentioned that an unsupervised learning algorithm would be interesting, first, because one of our aims is to emulate human learning, and second, because working with huge data sets makes supervised algorithms impractical. The original idea of Hinton, Osindero and Teh (see [27]) consists in training the network layer by layer trying to preserve as much useful information as possible at each step. The final result is a deep network that can be easily trained with backpropagation to perform a classification or regression task.

Many authors (see [21], [20]) have built deep networks dedicated to solve classification problems by stacking layers of restricted Boltzmann machines. The way those layers are stacked is the following

Algorithm 3 Deep belief network

Train a RBM using the original data.
Use the hidden units of the RBM as the first hidden layer.
for $l = 1 \dots L$ **do**
  Train a new RBM clamping its visible units to the values $P(H_i = 1|V, W, B)$ of the H_i units of the last available hidden layer of the network.
  Use the hidden units of the RBM as the next hidden layer.
end for
Connect a perceptron to the last hidden layer, train all the network with a supervised backpropagation algorithm.

That is, the greedy unsupervised pre-training phase builds the network by stacking layers trained to reconstruct the data, then, in the final training phase, a perceptron

is attached to the last layer and the complete network is trained, now with a classical supervised algorithm, to perform the desired task.

Recently, a new model of deep neural network has been developed by stacking autoencoders instead of restricted Botlzmann machines. This networks have a comparable performance to the deep belief networks. If the autoencoders used are sparse or denoising ones, the results might be even better than in deep belief networks. A stacked autoencoder is built with the following procedure

Algorithm 4 Stacked autoencoder

Train an autoencoder using the original data.
Use the hidden units of the autoencoder as the first hidden layer.
for $l = 1 \ldots L$ **do**
  Train a new autoencoder using as input the output values of the last available hidden layer of the network.
  Use the hidden units of the autoencoder as the next hidden layer.
end for
Connect a perceptron to the last hidden layer, train all the network with a supervised backpropagation algorithm.

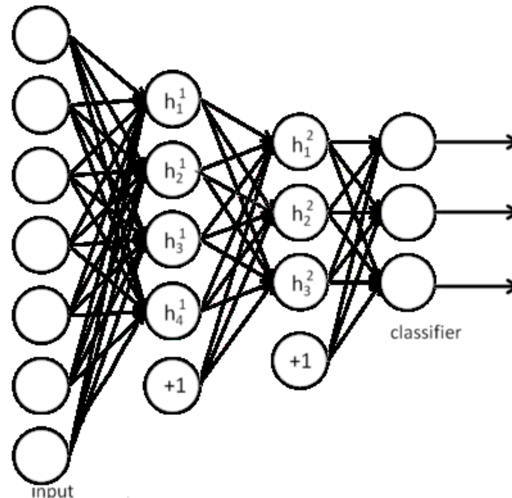

Figure 3.2.1: Deep belief network or stacked autoencoder

3.3 Other models

3.3.1 Convolutional networks

The first working model of a deep neural network, specifically designed for image processing, is the convolutional network (see [31]). Convolutional networks are multi-

layer perceptrons where each unit is only connected to a fixed number of consecutive units in the previous layer.

The reason this networks can be trained efficiently with backpropagation is that all the units in each layer have the same number of connections with the previous layer and the connections also have the same weight, that is, the weight of the i-th connection of the m-th unit is the same that the weight of the i-th connection of the m'-th unit of the same layer.

If the activation function is the hyperbolic tangent, the activations of each of this groups of units of a layer that share weights are

$$H = \tanh\left(W * X + b\right)$$

where $*$ is the convolution, that is usually defined as

$$f(n) * g(n) = \sum_{u=-\infty}^{\infty} f(u)g(n-u) = \sum_{u=-\infty}^{\infty} f(n-u)g(u)$$

and that is also the operation that gives name to this model.

The reason for this structure is that, being a model specially designed for image processing, it uses the stationary structure of natural images, where the characteristics of a region of the image are statistically similar to those of other regions in the same image. So, each region of the image is assigned a group of units, called *feature map*, and it is possible to train all the different groups together, adding up the gradients.

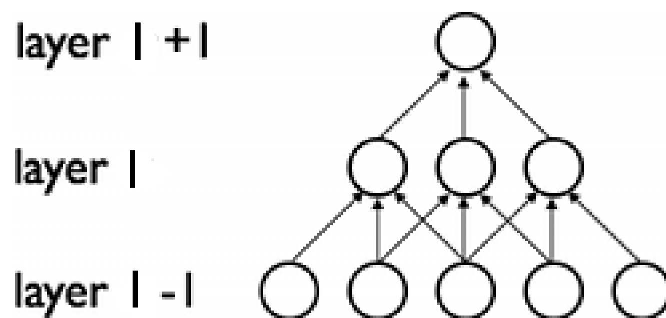

Figure 3.3.1: Convolutional network

As we will see now, this model is a particular case of the more general receptive fields.

3.3.2 Receptive fields

An important concern when designing a deep neural network is to specify how the units in a layer are connected to the units in the layer beneath. A fully connected neural network will be difficult to represent, train and even to implement, considering that some algorithms are difficult to generalize to large dimension domains. This problem can be solved by limiting the number of connections between a high level unit and lower level units. Ideally, a method that selects which connections are made will be unsupervised and able to operate without prior knowledge of the problem and the input data, so we can use it as part of our unsupervised training algorithm for stacking restricted Boltzmann machines or autoencoders.

Usually, when designing networks for visual tasks as the previously mentioned convolutional network, we connect a high level unit only to a small number of lower level units called the *feature map* or *receptive field*. For each of the second layer units, the selected lower level units will form a rectangular area within the complete input image. Though there are still spatial relationships between the higher pairs of layers, it is not clear how connections should be done, so typically every unit in a higher level will be connected to a random subset of the previous level units.

A possible solution to this problem, proposed by Coates and Ng (see [32]), consists in choosing receptive fields that group together lower-level features according to a similarity metric. We will assume that we have a feature dataset X of vectors X^n, $n \in \{1, \dots, N\}$ with elements X^n_m, $m \in \{1, \dots, M\}$. These vectors can be the input data (for the first layer) or features generated by lower layers of the deep network.

We must now define a similarity metric between features. We should group together closely related features to allow their relationship to be modelled more finely and because it makes sense to model seemingly independent features separately. Various metrics can be used to quantify similarity between features. A common choice is the square correlation of feature responses. If features X are linearly uncorrelated (whitening has been applied previously, so $\mathbb{E}\left[X\right] = 0$ and $\mathbb{E}\left[XX^T\right] = I$) then we can define the similarity between features X_j and X_k as the correlation between the

squared responses:

$$\begin{aligned}
S[X_j, X_k] &= Corr\left(X_j^2, X_k^2\right) \\
&= \frac{Cov\left(X_j^2, X_k^2\right)}{\sigma_{X_j^2}\sigma_{X_k^2}} \\
&= \frac{\mathbb{E}\left[\left(X_j^2 - \mu_{X_j^2}\right)\left(X_k^2 - \mu_{X_k^2}\right)\right]}{\sigma_{X_j^2}\sigma_{X_k^2}} \\
&= \frac{\mathbb{E}\left[\left(X_j^2 - \mu_{X_j^2}\right)\left(X_k^2 - \mu_{X_k^2}\right)\right]}{\sqrt{\mathbb{E}\left[\left(X_j^2 - \mu_{X_j^2}\right)^2\right]\mathbb{E}\left[\left(X_k^2 - \mu_{X_k^2}\right)^2\right]}} \\
&= \frac{\mathbb{E}\left[X_j^2 X_k^2 - X_j^2\mu_{X_k^2} - \mu_{X_j^2}X_k^2 + \mu_{X_j^2}\mu_{X_k^2}\right]}{\sqrt{\mathbb{E}\left[X_j^2 + \mu_{X_j^2}^2 - 2X_j^2\mu_{X_j^2}\right]\mathbb{E}\left[X_k^2 + \mu_{X_k^2}^2 - 2X_k^2\mu_{X_j^2}\right]}} \\
&= \frac{\mathbb{E}\left[X_j^2 X_k^2 - 1\right]}{\sqrt{\mathbb{E}\left[X_j^4 - 1\right]\mathbb{E}\left[X_k^4 - 1\right]}}
\end{aligned}$$

Which is easy to compute if we first apply whitening to the data and then calculate the pairwise similarities between all the features from a sample:

$$S_{j,k} \equiv S_X[X_j, X_k] \equiv \frac{\sum_n \left(X_j^n\right)^2 \left(X_k^n\right)^2 - 1}{\sqrt{\sum_n \left(\left(X_j^n\right)^4 - 1\right)\left(\left(X_k^n\right)^4 - 1\right)}}$$

Now that we have the matrix $S_{j,k}$ we can construct the receptive fields R_m, $m = 1, \ldots, M$ that will define the connections between the two selected layers. We want each R_m to contain pairs of features with large values of $S_{j,k}$. We might achieve this using an agglomerative or spectral clustering method, like k-means or similar, but usually a simple greedy procedure works well:

1. We select M rows $j_1, \ldots, j_M$ of the matrix S randomly (the seed of each group).
2. We construct a receptive field R_m that contains the features X_k that have the T top values of $S_{j_m,k}$.

Upon completion, we have M possibly overlapping receptive fields R_m.

Computing the matrix $S_{j,k}$ is practical for large numbers of features, but when working with deep networks that use thousands of units per layer we will be unable to compute the matrix directly. However, our greedy algorithm requires only pairs of rows of the matrix, so, provided we can perform both pair-wise whitening and pair-wise correlation between the selected pair of features, it is clear that $S_{j,k}$ can be computed for a pair of features, and we don't need to store the entire matrix.

Chapter 4

Experiments

The experiments performed consist on reconstructions and classification of handwritten digits from the MNIST database (http://yann.lecun.com/exdb/mnist/). The high requirements of deep networks in terms of computational power, and the limited resources at our disposal for the development of this master thesis, force us to reduce the dimension of the MNIST database patterns, and excludes completely any experiment with higher-dimensional image sets like CIFAR (http://www.cs.toronto.edu/~kriz/cifar.html). Our objective is to compare which architectures perform better at each combination of number of units per hidden layer in the case of reconstruction, and number of hidden layers, units per hidden layer, and initialization algorithm in the case of classification.

We have performed two different kinds of experiments, the first, consists in reconstructing patterns with autoencoders and restricted Boltzmann machines, to illustrate their reconstruction properties and to validate the models programmed. The second consist on investigating the capacity of deep networks in a particular problem of MNIST digit classification.

4.1 Datasets

The dataset used is a sample from MNIST, that is again a sample from NIST. As it can be read in http://yann.lecun.com/exdb/mnist/, the original black and white images from NIST were size normalized to fit in a 20×20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. the images were centered in a 28×28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28×28 field.

Our sample has 5000 training patterns and 1000 test patterns, and, for the purpose of testing networks that have hidden layers both narrower and wider than the input layer, we have reduced the dimension of the images from 28×28 to 14×14, and so, we can study the effect of the width of the hidden layers with less computational capacity. The transformation consists in dividing the original image in 2×2 pixel sets, picking the mean value of each set and normalizing to $[0, 1]$.

4.2 Software implementation

We have developed a basic *Octave* library for the construction and testing of autoencoders and its variants, and deep neural networks like deep perceptrons, stacked autoencoders, and deep belief networks. The code has been tested on *GNU Octave* 3.2.4 *for x86-64-pc-linux-gnu*. The RBM code is based on the implementation of *RBMLib* by Andrej Karpathy, available at `http://code.google.com/p/matrbm/`.

Functions have been separated into six categories: activation, image, neural, noise, utilities and main.

- Activation:
  - didentity: derivative of the identity function.
  - dlogistic: derivative of the logistic function.
  - dsoftsign: derivative of the softsign function.
  - dtahyp: derivative of the hyperbolic tangent function.
  - identity: identity function.
  - logistic: logistic function.
  - softsign: softsign function.
  - tanhyp: hyperbolic tangent function.
- Image:
  - reduceDataset: reduces the dimension of the images of a MNIST dataset.
  - reduceDim: reduces the dimension of an image.
- Neural:
  - advancedRandomInitialization: initializes the weigths and biases of a perceptron with the advanced algorithm described in section 3.1.3.
  - ensembleDeepNetwork: assembles a deep network by connecting the stacked layers to the perceptron classifier.
  - greedyPretraining: pre-trains the stack of autoencoders or rbms. It receives the data and the number of layers and hidden units and creates the stack of hidden layers (autoencoders or restricted Boltzmann machines) by the greedy pre-training algorithm. It uses mlpTrain if the layers are autoencoders or rbmTrain if the layers are restricted Boltzmann machines. Also, it uses the function propagate to pass the data from the input to the layer that is being trained.
  - labels2targets: gets the pattern targets from the pattern labels.
  - mlpTrain: trains a multilayer perceptron, that can add noise to the inputs, a sparsity condition, weight decay, etc. Autoencoders are MLPs trained to reproduce the input data at the output. It uses the function propagate during the forward pass.

  - propagate: propagates through a perceptron.
  - randomInitialization: initializes the weights and biases of a perceptron randomly.
  - rbmTrain: trains a restricted Boltzmann machine. It is a modification of the function rbmFit of RBMLib by Andrej Karpathy.
  - targets2labels: gets the pattern labels from the pattern targets.
  - visualize: visualizes a vector as an image. Taken from RBMLib.
  - visualizePropagation: visualizes the input data, internal representations and learned weights of a perceptron during propagation.
  - visualizeReconstruction: visualizes the input data, internal representations, learned weights and reconstructions of an autoencoder or rbm during reconstruction.

- Noise:
  - gaussian: adds gaussian noise to the data received in matrix form.
  - masking: adds masking noise to the data received in matrix form.
  - none: adds no noise to the data received in matrix form.
  - saltPepper: adds salt and pepper noise to the data received in matrix form.
- Utilities: several useful functions taken from RBMLib.
- Main: experimentsDeep and experimentsShallow, Octave scripts with the experiments performed in this master thesis.

With the described functions, the following steps must be performed in order to create a stacked autoencoder or a deep belief network classifier and to use it to classify the test patterns:

1. Read the data.
2. Create the targets to train the classifier from the data labels.
3. Create the hidden layers stack with the greedy pre-training algorithm. This can be done using the four autoencoder variants or restricted Boltzmann machines, depending on the arguments passed to the greedy unsupervised pre-training function.
4. Initialize the weights of a one hidden layer perceptron that will be used as classifier. Both the random initialization and the advanced random initialization can be used.
5. Assemble the stack of hidden layers and the perceptron classifier to form the deep network classifier.

6. Train the whole network with the supervised algorithm of function mlpTrain. This function receives the training data and targets and the weights and biases of the deep network classifier.

7. Classify the test patterns by propagating the test data through the deep network classifier. The function propagate receives the test data and the weights and biases of the classifier, and returns the predicted targets.

4.3 Building blocks and overall architectures

The building blocks tested are the four variants of autoencoders (normal, denoising, sparse and sparse-denoising) and the restricted Boltzmann machines. The experiments on these blocks consist on training the network with the training set and then reconstructing the test set. The results presented consist on a sample of the original data and its reconstruction, and the visualization of the learned weights and the sample's internal representation. The reconstruction error computed for the restricted Boltzmann machine cannot be compared against the reconstruction errors of the different autencoders, as the first reconstructs a black and white image, while the former give a grayscale image. However, we also present the restricted Boltzmann machine reconstructions for visual inspection.

The autoencoders have logistic hidden units and linear output units, and the restricted Boltzmann machines have logistic units both hidden and visible. The error function used is the standard quadratic error function. The noise selected for the denoising and sparse-denoising autoencoders is gaussian with mean 0 and standard deviation 0.25, values which are selected based on the experiments performed in [21]. For the sparse and sparse-denoising autoencoders, the sparsity parameter and the sparsity factor in the error function are both 0.2.

The overall architecture of the deep networks is formed by 1 to 4 layers of stacked autoencoders or restricted Boltzmann machines, with a one hidden layer perceptron on top. The perceptron is initialized using the advanced random initialization described previously, to ensure the efficient training of the whole network as a classifier, and the hidden layer has the same width as the stacked layers preceding it. We test the deep networks both when the whole assembled network is fine-tuned with backpropagation, and when only the classifier is trained with backpropagation, feeding it with the outputs of the stacked layers. Also we compare these architectures with traditional deep multilayer perceptrons, initialized both the usual way or with the advanced random initialization.

4.4 Shallow networks experiments

The first set of experiments involves

- autoencoders (AE),

- denoising autoencoders (DAE),
- sparse autoencoders (SAE),
- sparse denoising autoencoders (SDAE),
- restricted Boltzmann machines (RBM).

The objective is to illustrate the reconstruction properties of these models, through visual inspection and checking of the numeric error, and to validate them before building a deep network of stacked autoencoders or restricted Boltzmann machines. It is expected that the internal representation of the patterns will retain as much information as possible from the input patterns. The learned weights are expected to show some kind of structure, and they also represent the patterns that produce the maximal activation of the hidden units, as we saw in the autoencoders visualization section. The first 16 patterns of the test set will be used for visual inspection of the input, internal, and output values.

The tests have been performed with $100(10^2)$, $144(12^2)$, $196(14^2)$ and $256(16^2)$ hidden units. The reason to pick those numbers is that the Octave visualization function is designed for squared images. We will observe two different regimes: one, quite good, for networks that are narrow in their hidden layer, and other, not so good, for networks that are wide in their hidden layer, with width similar or greater than that of the input and output layers. Only the cases of 100 and 256 hidden units are depicted, as the 144 units case gives similar results to the first, and the 196 units case to the second. For the 100 unit nets, we can summarize our results as follows

- The autoencoder with 100 hidden units (figure 4.4.1) does a very good reconstruction (bottom right) from the original data (bottom left). However, the learned weights (top left) don't show much structure and the internal representation of the patterns (top right) is difficult to interpret, although some similarities can be found between patterns of the same class, like the two 9s in the bottom row.

- The denoising autoencoder with 100 hidden units (figure 4.4.2) also performs a very good reconstruction (bottom right) of the original data (bottom left). The learned weights (top left) have more complex structures than those of the standard autoencoder, and the internal representations (top right) are also quite different.

- The sparse autoencoder with 100 hidden units (figure 4.4.3) has similar learned weights (top left) and internal representations, maybe more blurred (top right) to those of the standard autoencoder. The reconstruction (bottom right) is also good, but a bit fuzzier.

- The sparse denoising autoencoder with 100 units (figure 4.4.4) performs similarly to the denoising autoencoder, with good reconstructions (bottom right) and complex learned weights (top left). The internal representations are darker, showing few light spots, due to the effect of the sparsity condition.

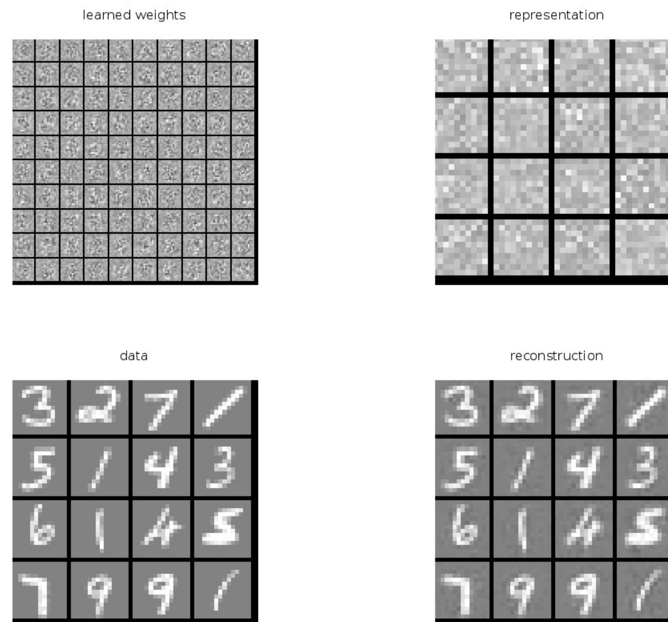

Figure 4.4.1: Reconstruction with 100 units AE: the original patterns (bottom left) have a good reconstruction (bottom right), but the learned weights (top left) and internal representations (top right) show little structure.

- The restricted Boltzmann machine with 100 hidden units (figure 4.4.5) does a good black and white reconstruction (bottom right) of the data (bottom left). The learned weights (top left) show much more structures than those of the autoencoders, and the internal representations (top right) are similar to those of the denoising autoencoder.

For the 256 unit networks, the summary of our results is as follows

- The 256 units autoencoder (figure 4.4.6) doesn't seem to work very well. It could be expected that the reconstruction (bottom right) would be almost perfect, as the wider hidden layer can pass all the information from the input to the output, but in this case the reconstruction is much worse than in the 100 units case, and the learned weights (top left) and internal representations (top right) have a totally different appearance. Probably, an overfitting problem is present.

- The denoising autoencoder with 256 units (figure 4.4.7) performs like the standard autoencoder. The reconstructions (bottom right) are a little better, but the learned weights (top left) and internal representations (top right) are similar. As with the standard autoencoder, we may have an overfitting problem.

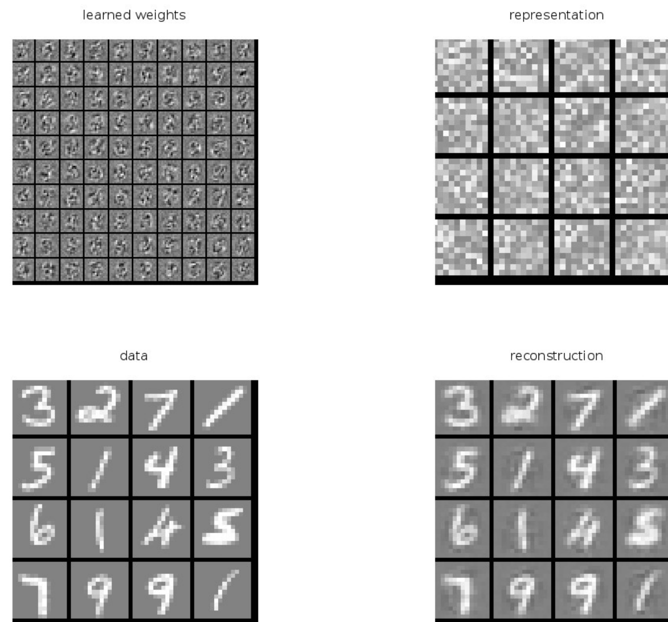

Figure 4.4.2: Reconstruction with 100 units DAE: the original patterns (bottom left) and their reconstructions (bottom right). Learned weights (top left) are complex, and the internal representations (top right) show greater contrast than those of the standard autoencoder.

- The sparse autoencoder with 256 units (figure 4.4.7) performs well, with reconstructions (bottom right) almost as good as those of the 100 units case. The learned weights (top left) show some structure, unlike those of the autoencoder and denoising autoencoder of 256 units, and the internal representations (top right) are similar to those of the 100 units case. The sparsity condition could be controlling the possible overfitting problem seen in other variants.

- The 256 units sparse denosing autoencoder (figure 4.4.9) is also doing a good reconstruction (bottom right). The learned weights (top left) and internal representations (top right) are quite similar to those of the 100 units sparse denoising autoencoder. As with the sparse autoencoder, it seems that the possible overfitting issue is controlled in this case.

- The restricted Boltzmann machine with 256 hidden units (figure 4.4.10) seems to be the only model that gets better results with high number of hidden units, as the reconstructions (bottom right) are sharper and clearer. The learned weights (top left) and internal representations (top right) are like those of the 100 hidden units case. The restricted Boltzmann machine is not affected by the overfitting problem seen in the autoencoder and denoising autoencoder.

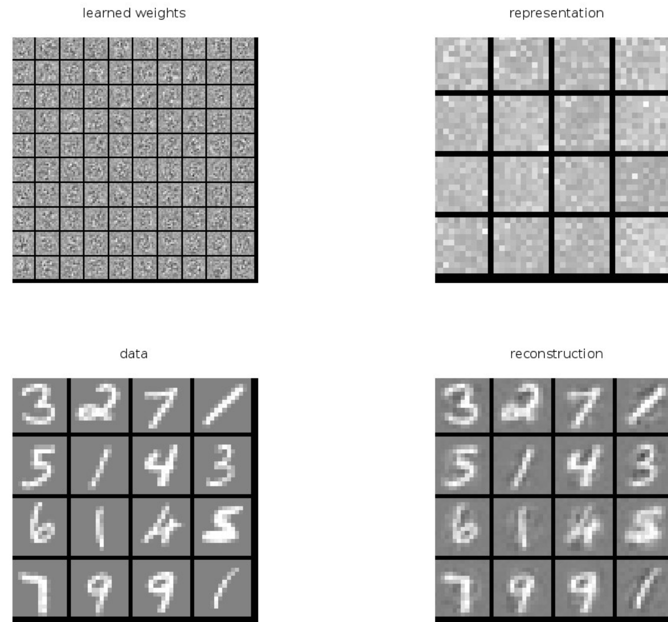

Figure 4.4.3: Reconstruction with 100 units SAE: original patterns (bottom left), their reconstructions (bottom right), learned weights (top left) that are similar to those of the standard autoencoder, and internal representations (top right) more blurred, which seems reasonable because of the sparsity condition.

As a conclusion, with this first experiment, we observe that with 100 hidden units all the models give fair reconstructions, although the internal representations and learned weights differ notoriously between them. When using 144 hidden units, the results are similar. However, when we try 196 and 256 units, the reconstructions get worse for the standard and denoising autoencoders. We could expect them to perform a nearly perfect reconstruction, given that having a hidden layer wider than the input and output layers it is possible to transfer all the information through the autoencoder, but in the experiments only the sparse and sparse-denoising autoencoder and the restricted Boltzmann machine give a good reconstruction of the data. The small number of training patterns, combined with a high number of parameters to determine when the hidden layer is wide, may be causing overfitting. The graphic (figure 4.4.11) showing the reconstruction errors (euclidean distance between data and reconstruction) for all the autoencoders, illustrates this observation.

We have not been able to find a similar test published to contrast our results. It seems that, when using a wide (similar or wider than the input) hidden layer, the autoencoder needs a sparsity condition to, not only improve, but also to maintain the performance obtained with a narrow hidden layer. It is possible that, when using a high number of hidden units, as the number of parameters is much larger than the number of training patterns, overfitting appears and the results with the test set get

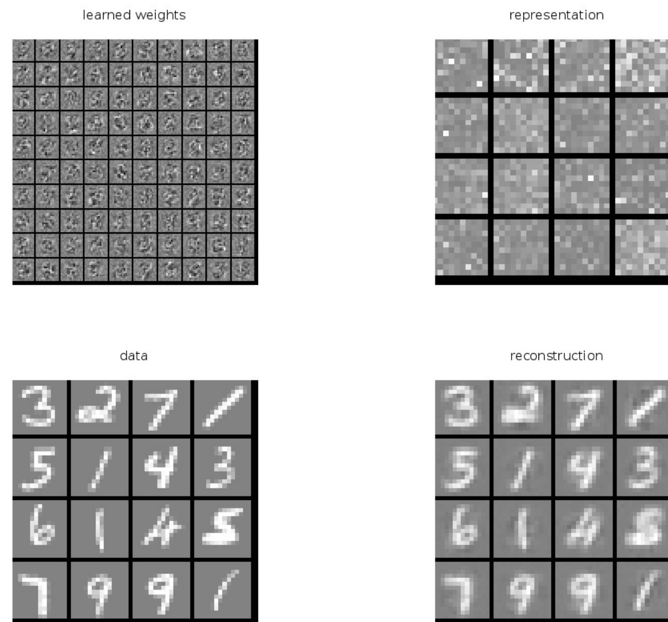

Figure 4.4.4: Reconstruction with 100 units SDAE: original patterns (bottom left), reconstructions (bottom right) and learned weights (top left) are similar to those of the denoising autoencoder. The internal representations are darker, because of the sparsity condition.

worse.

4.5 Deep networks experiments

The second experiment compares different types of deep networks: the standard multilayer perceptron with random initialization (MLP) and advanced random initialization (AMLP), and stacked autoencoders (AE), denosing autoencoders (DAE), sparse autoencoders (SAE), sparse-denoising autoencoders (SDAE) and restricted Boltzmann machines (RBM), these five with both full fine-tuning (backpropagation with the full network assembled) or just classifier fine-tuning (backpropagation only through the classifer, keeping the stacked layers untouched). The objective is to demonstrate the capabilities of the different types of deep network to solve classification problems.

The number of training patterns, 5000, and the dimensions of the images, $14 \times 14 = 196$, can induce overfitting if we use hidden layers with a high number of units. Also, we have observed in our first set of experiments that the reconstructions are better when the number of hidden units is not very high. Therefore, we will use $7 \times 7 = 49$ hidden units per hidden layer in our tests.

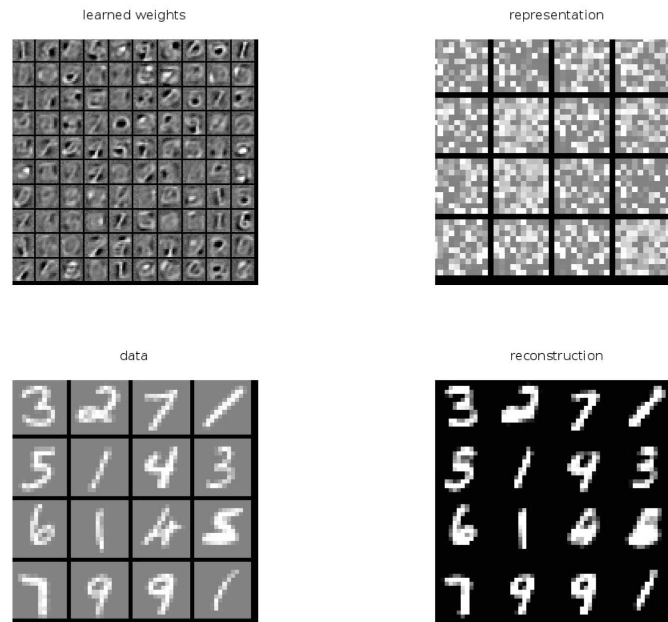

Figure 4.4.5: Reconstruction with 100 units RBM: the reconstruction (bottom right) is now black and white. The learned weights (top left) show complex structures and the internal representations (top right) are similar to those of the denoising autoencoder.

The number of stacked layers are 1 to 4, which means that, adding the two-layer classifier, the networks have a depth of 3 to 7. We expect to see how the greedy layer-wise pre-training performs versus the standard and advanced random initialization and study the effect of the final fine-tuning. The images show a sample of 16 test patterns, their internal representations through the different layers of the network, and the weights learned at each layer. Only the images for 2 stacked layers (4 layers in total) networks are shown, as the internal representations and weights are similar for the other cases.

- The multilayer perceptron with 3 hidden layers of 49 units (figure 4.5.1) shows some structure in the second layer weights and, specially, in the first layer weights (middle row). The classification error is below 0.1, proving that back-propagation works well for 3 hidden layers. We will see that this is not the case for 4 or more hidden layers. The internal representations (bottom row) look more faded in the last layers.

- The multilayer perceptron with 3 hidden layers of 49 units and advanced initialization (figure 4.5.2) also shows some structure in the first and second layer weights (middle row). The classification error is lower than in the multilayer perceptron with standard initialization. Also, the internal representations (bottom row) look more faded in the last layers.

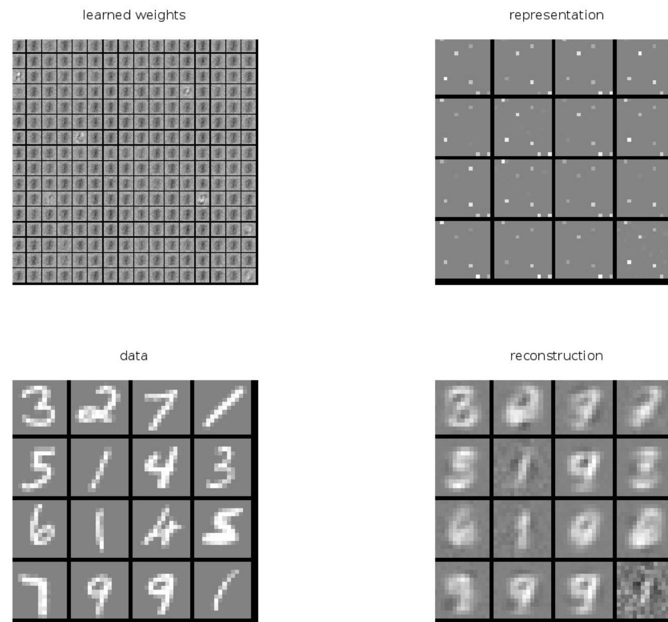

Figure 4.4.6: Reconstruction with 256 units AE: not very good reconstructions (bottom right), disorganized learned weights (top left) and internal representations (top right).

- The stacked autoencoder with 2 stacked layers (3 hidden layers in total) and 49 hidden units (figure 4.5.3) has learned weights similar to those of the standard multilayer perceptrons. The activations also look more faded in the last layers.

- The stacked denoising autoencoder with 2 stacked layers (3 hidden layers in total) and 49 hidden units (figure 4.5.4) has learned weights and internal representations visually very similar to those of the multilayer perceptrons.
- The stacked sparse autoencoder with 2 stacked layers (3 hidden layers in total) and 49 hidden units (figure 4.5.5) has learned weights and internal representations visually very similar to those of the multilayer perceptrons.
- The stacked sparse denoising autoencoder with 2 stacked layers (3 hidden layers in total) and 49 hidden units (figure 4.5.6) has learned weights and internal representations visually very similar to those of the multilayer perceptrons.
- The deep belief network with 2 stacked layers (3 hidden layers in total) and 49 hidden units (figure 4.5.7) has learned weights and internal representations visually very similar to those of the multilayer perceptrons.

The tests (figure 4.5.8) show similar results both for networks with fine-tuning and without it. As expected, the standard multilayer perceptron with random ini-

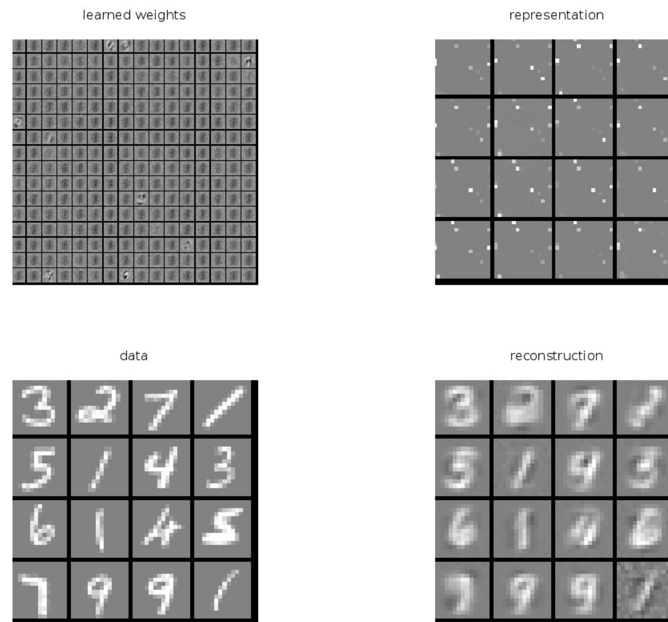

Figure 4.4.7: Reconstruction with 256 units DAE: the reconstructions (bottom right) are a bit better than in the standard autoencoder, but the learned weights (top left) and internal representations (top right) are similar.

tialization has a poor performance when the number of layers is greater than 3, but the advanced initialization gives good results. All the stacked autoencoders and the deep belief network have a good performance, confirming what was shown in [21] and [10], but only the perceptron initialized with the advanced method and the stacked standard autoencoder perform better with 4 hidden layers than with 3 hidden layers. The stacked autoencoder achieves an error rate close to 0.05.

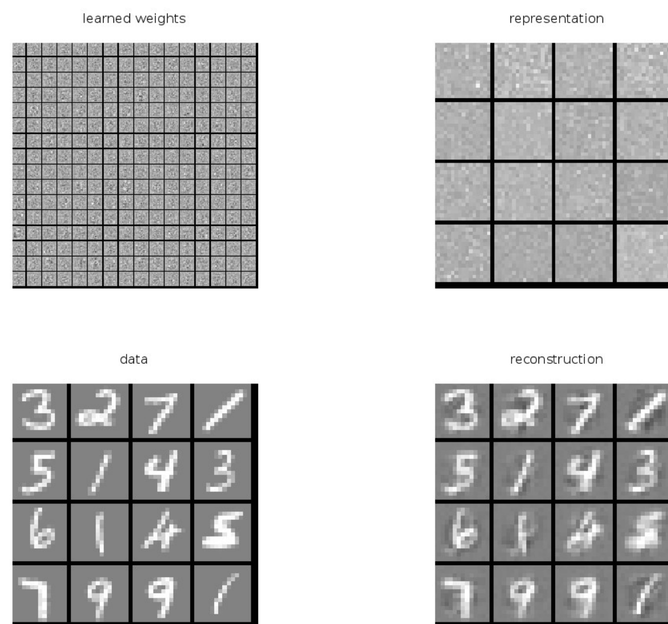

Figure 4.4.8: Reconstruction with 256 units SAE: the sparse autoencoder doesn't seem to have worse performance when the number of hidden units increases, as the reconstructions (bottom right) are quite good. The learned weights (top left) and internal representations (top right) are also similar to the ones obtained with 100 units.

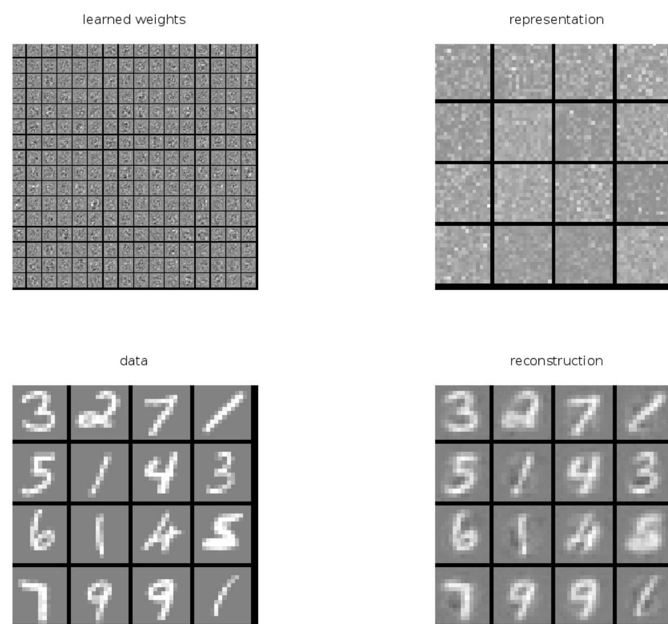

Figure 4.4.9: Reconstruction with 256 units SDAE: good reconstructions (bottom right) and complex learned weights (top left) indicate that the sparse denoising autoencoder performs well when the number of hidden units is high.

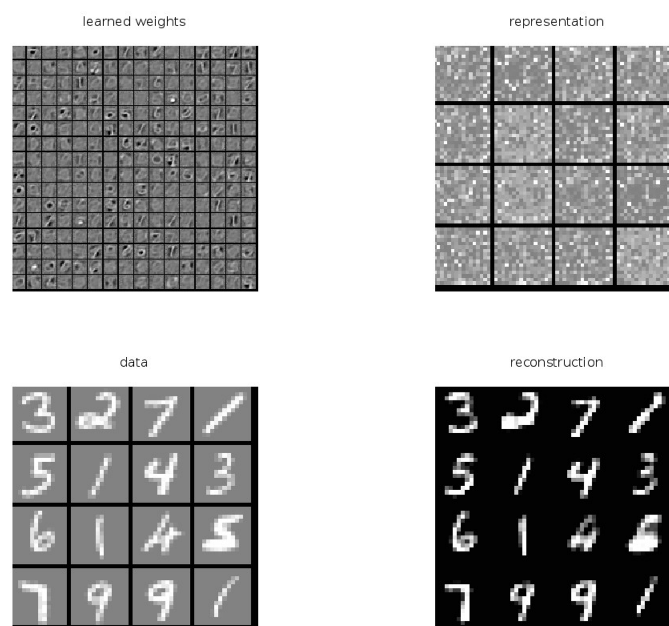

Figure 4.4.10: Reconstruction with 256 units RBM: the reconstructions (bottom right) are even better than with 100 units, and the learned weights (top left) and internal representations (top right) are similar.

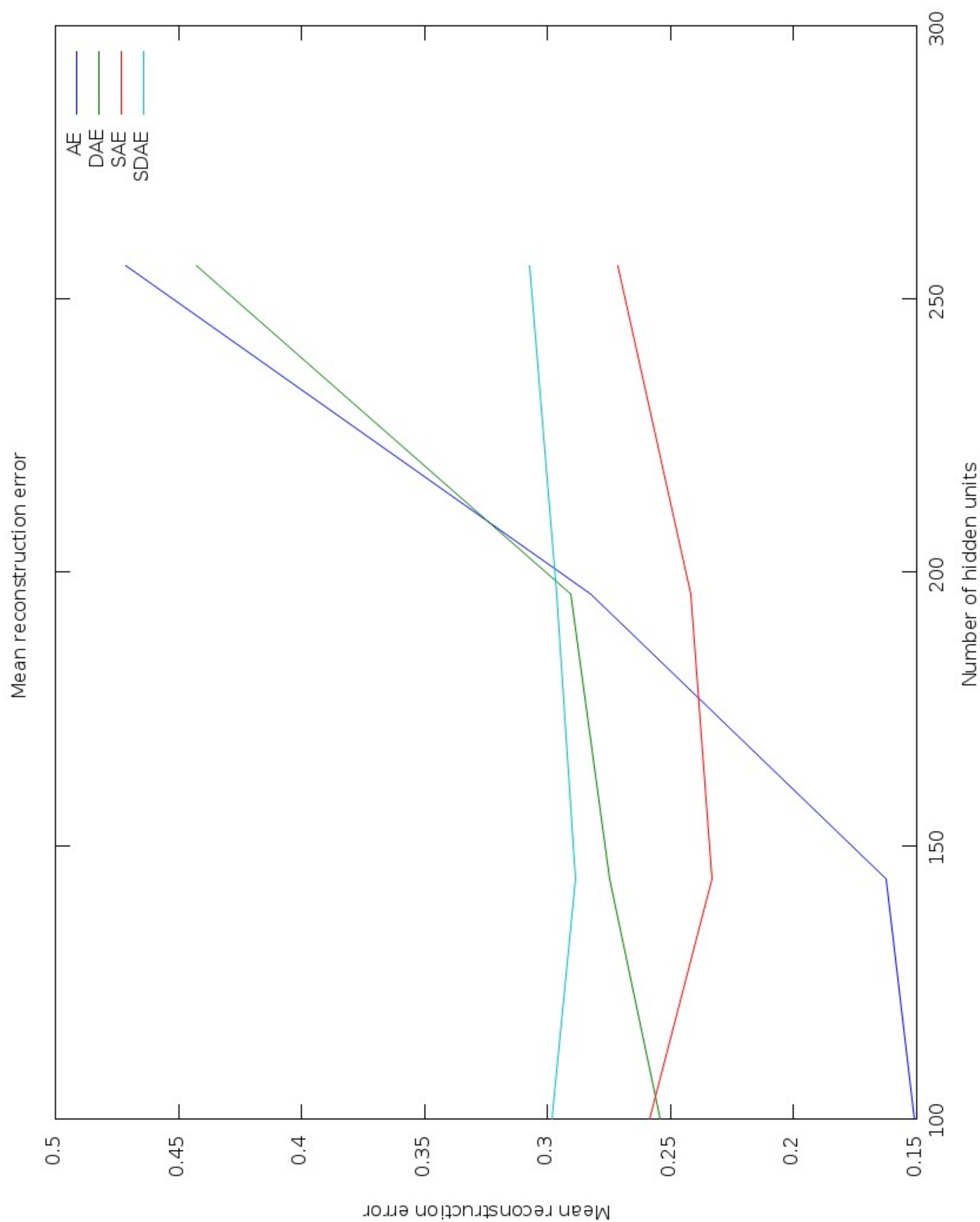

Figure 4.4.11: Reconstruction errors: the standard and denoising autoencoders behave worse when using a broad hidden layer. The autoencoders with sparsity have good performace in all cases. The restricted Boltzmann machine cannot be compared to them, as the reconstruction it performs is in black and white.

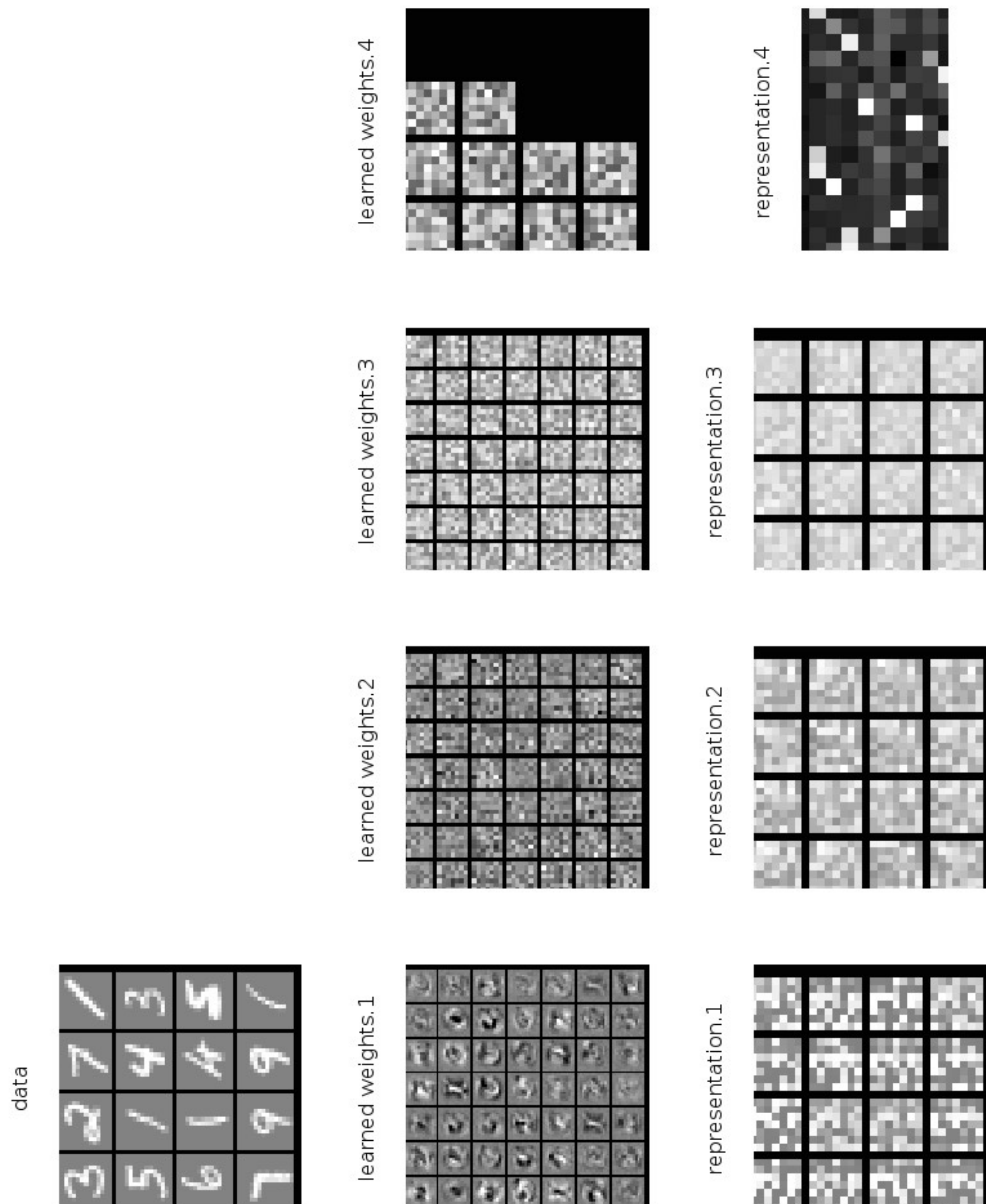

Figure 4.5.1: Classification with 3 hidden layers of 49 units MLP: the weights show some structure for the first and second layers (middle row), the internal representations (bottom row) look more faded in the last layers.

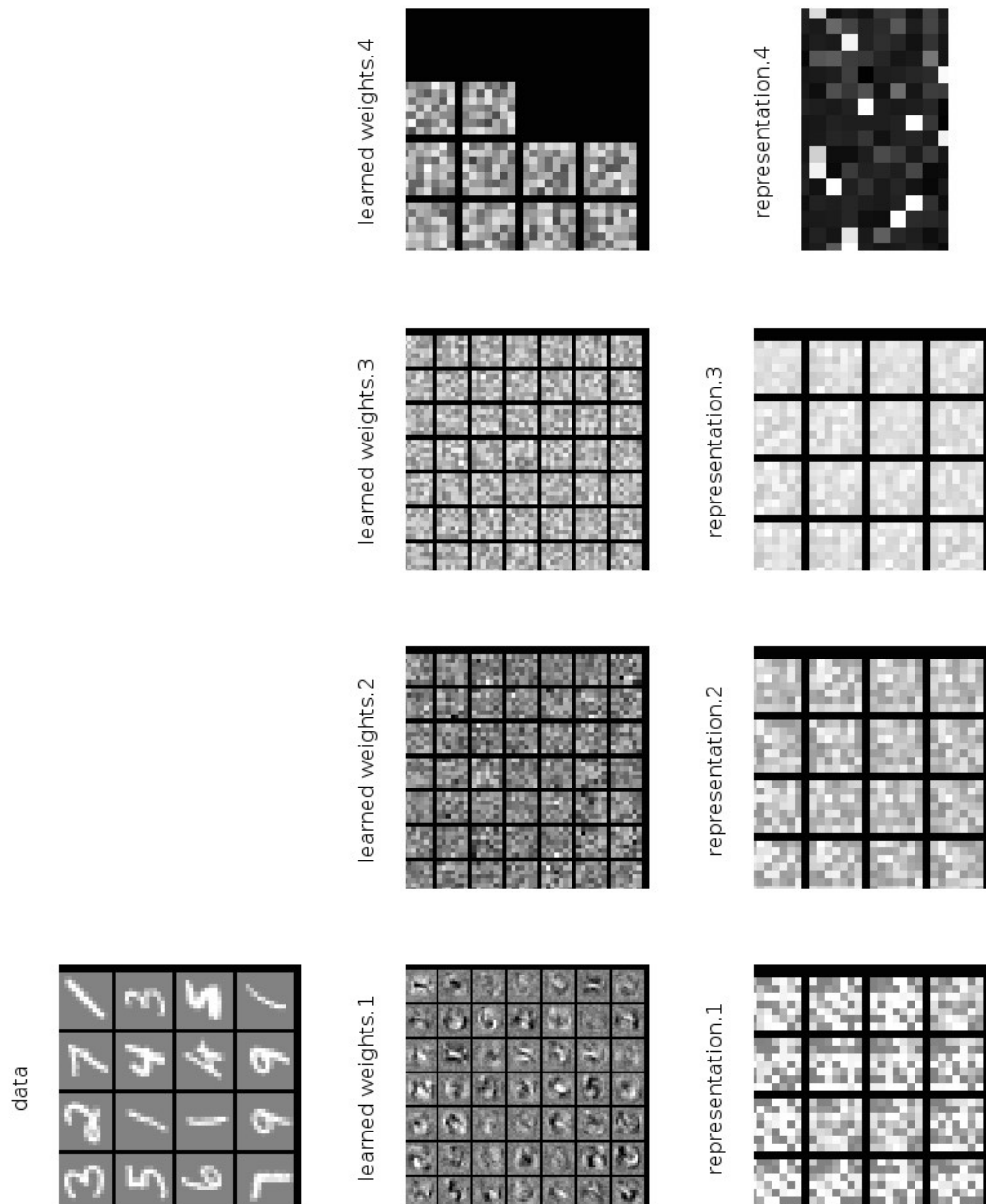

Figure 4.5.2: Classification with 3 hidden layers of 49 units AMLP: the weights show some structure for the first and second layers (middle row), the internal representations (bottom row) look more faded in the last layers.

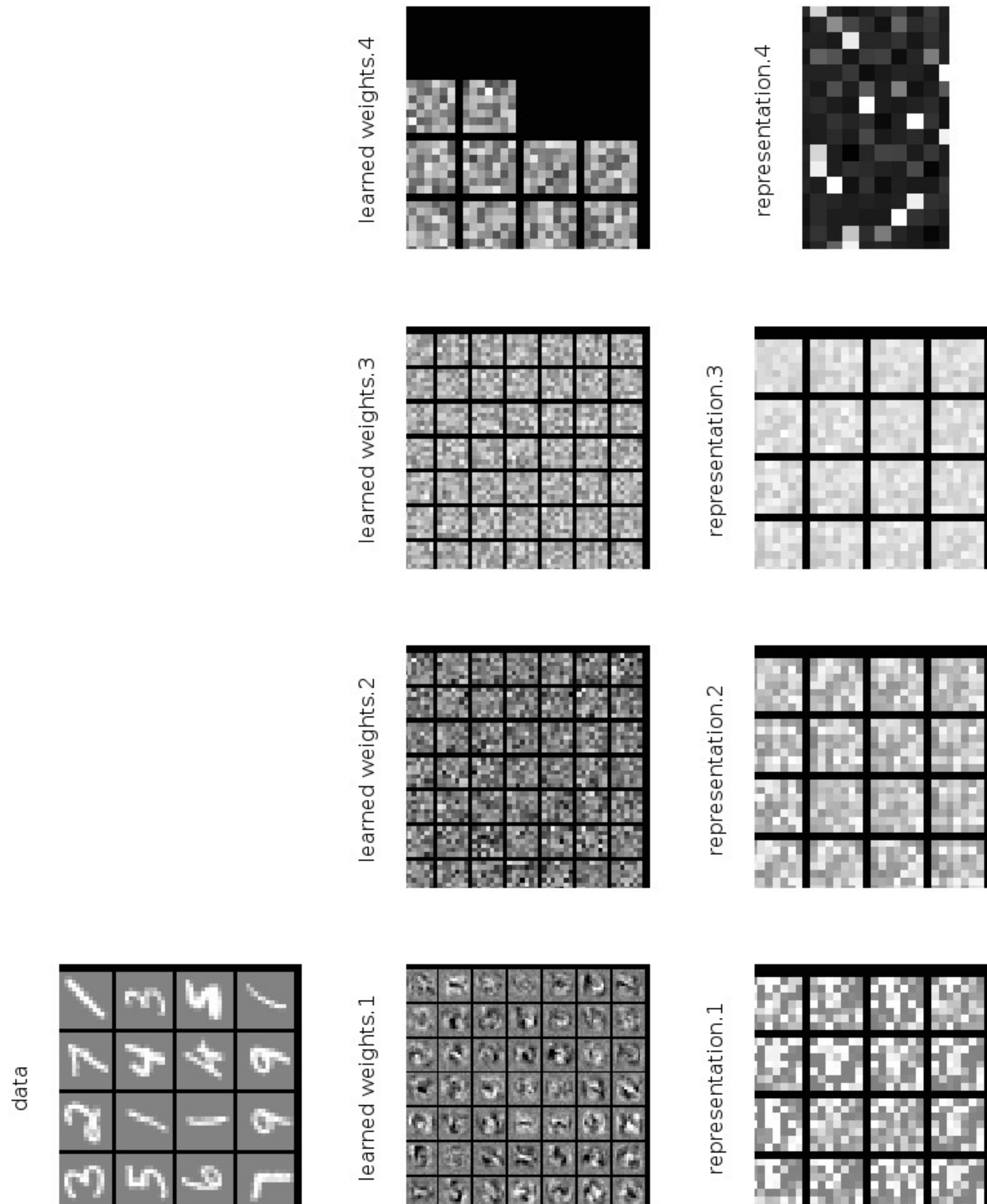

Figure 4.5.3: Classification with 2 layers of 49 units AE: only the first two layers show visible structures in the learned weights.

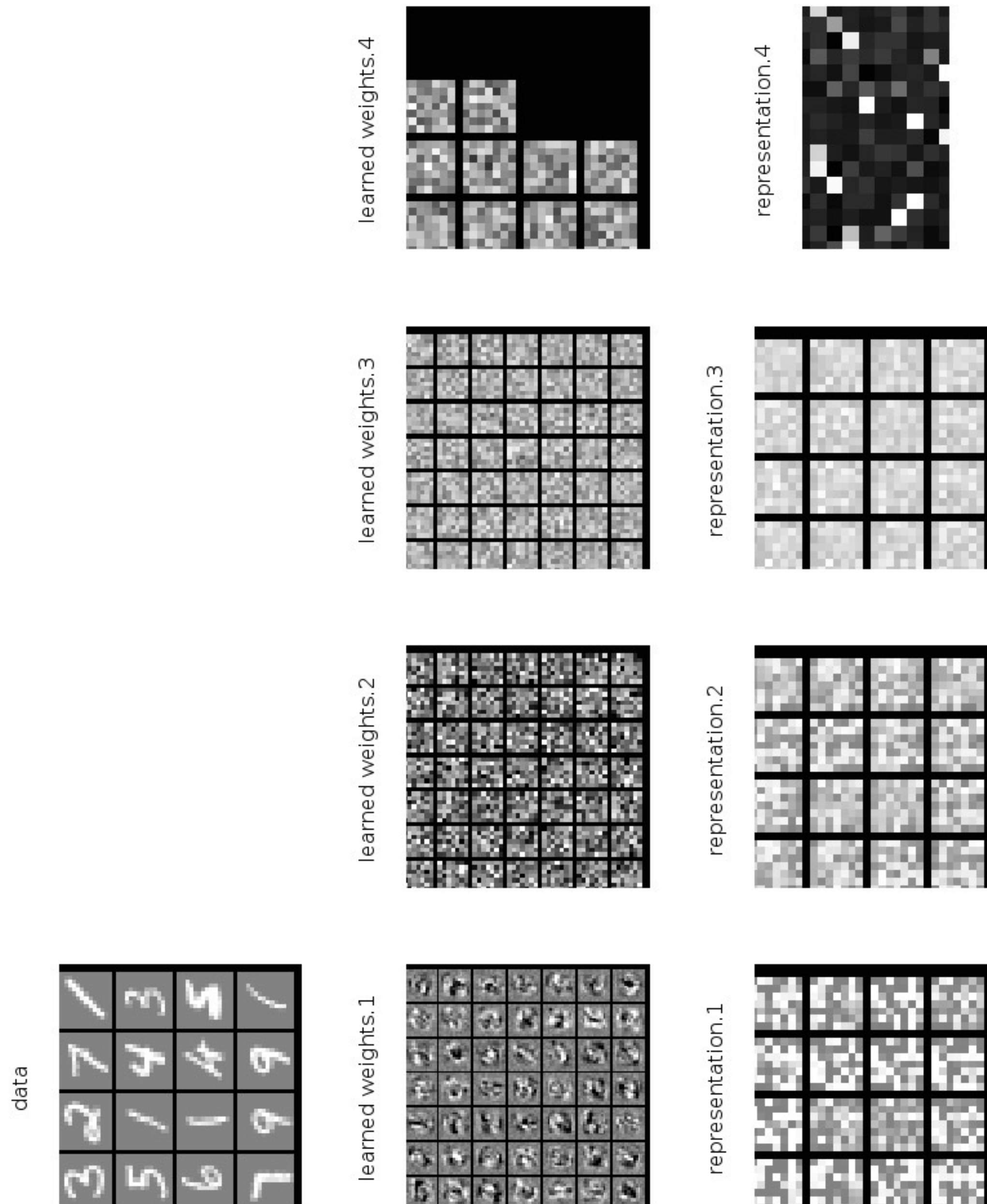

Figure 4.5.4: Classification with 2 layers of 49 units DAE: visually similar results to those of the multilayer perceptrons.

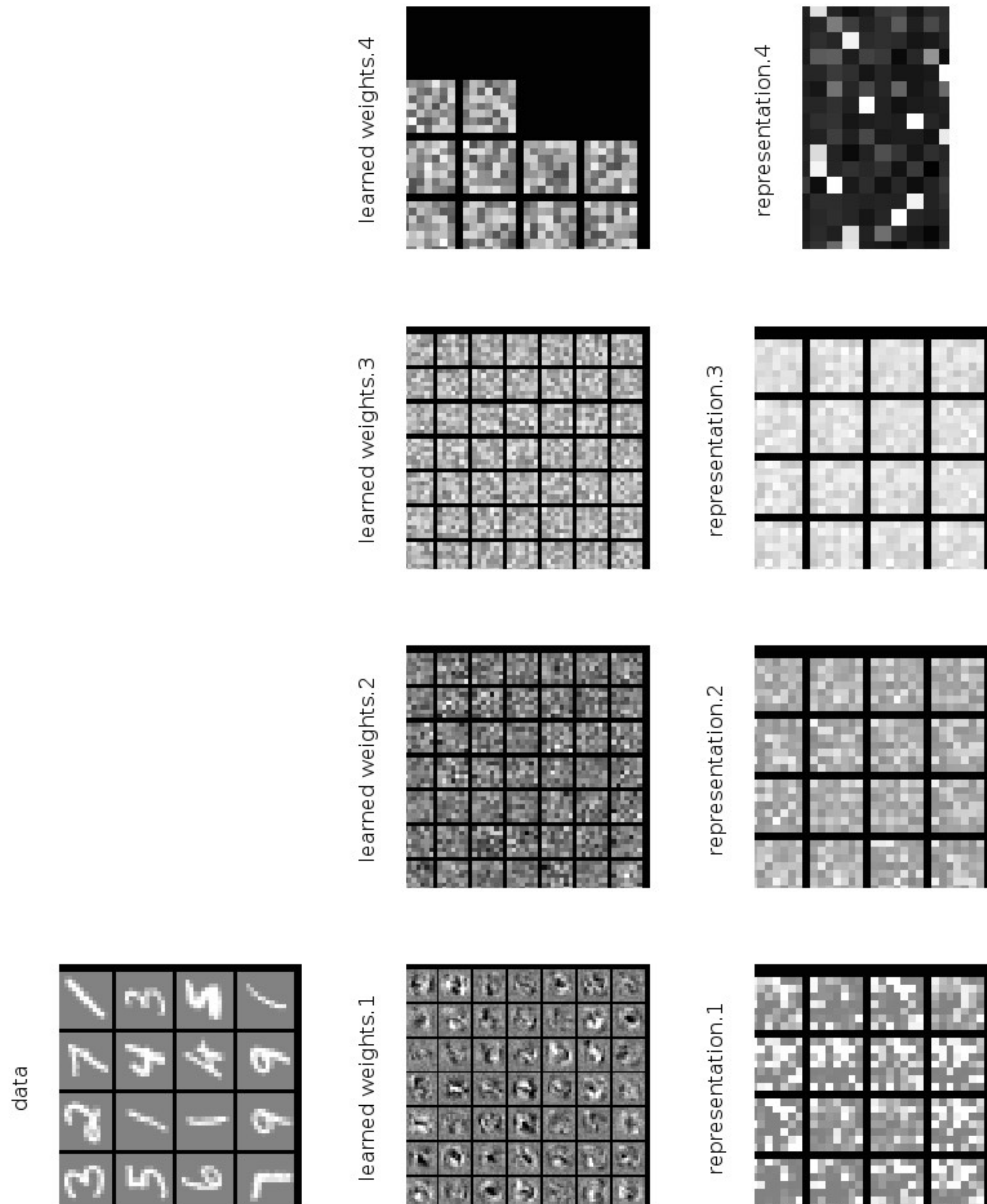

Figure 4.5.5: Classification with 2 layers of 49 units SAE: visually similar results to those of the multilayer perceptrons.

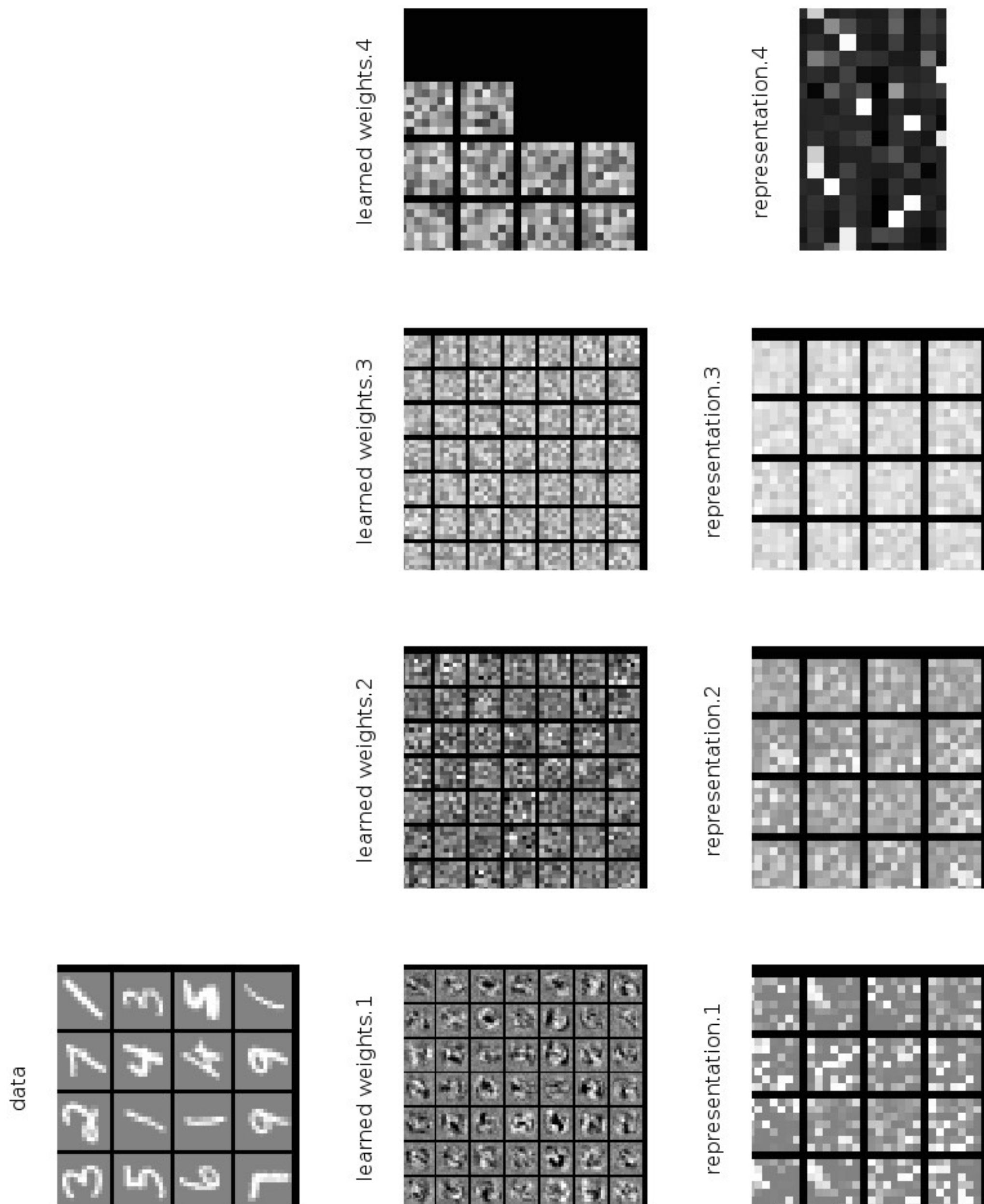

Figure 4.5.6: Classification with 2 layers of 49 units SDAE: visually similar results to those of the multilayer perceptrons.

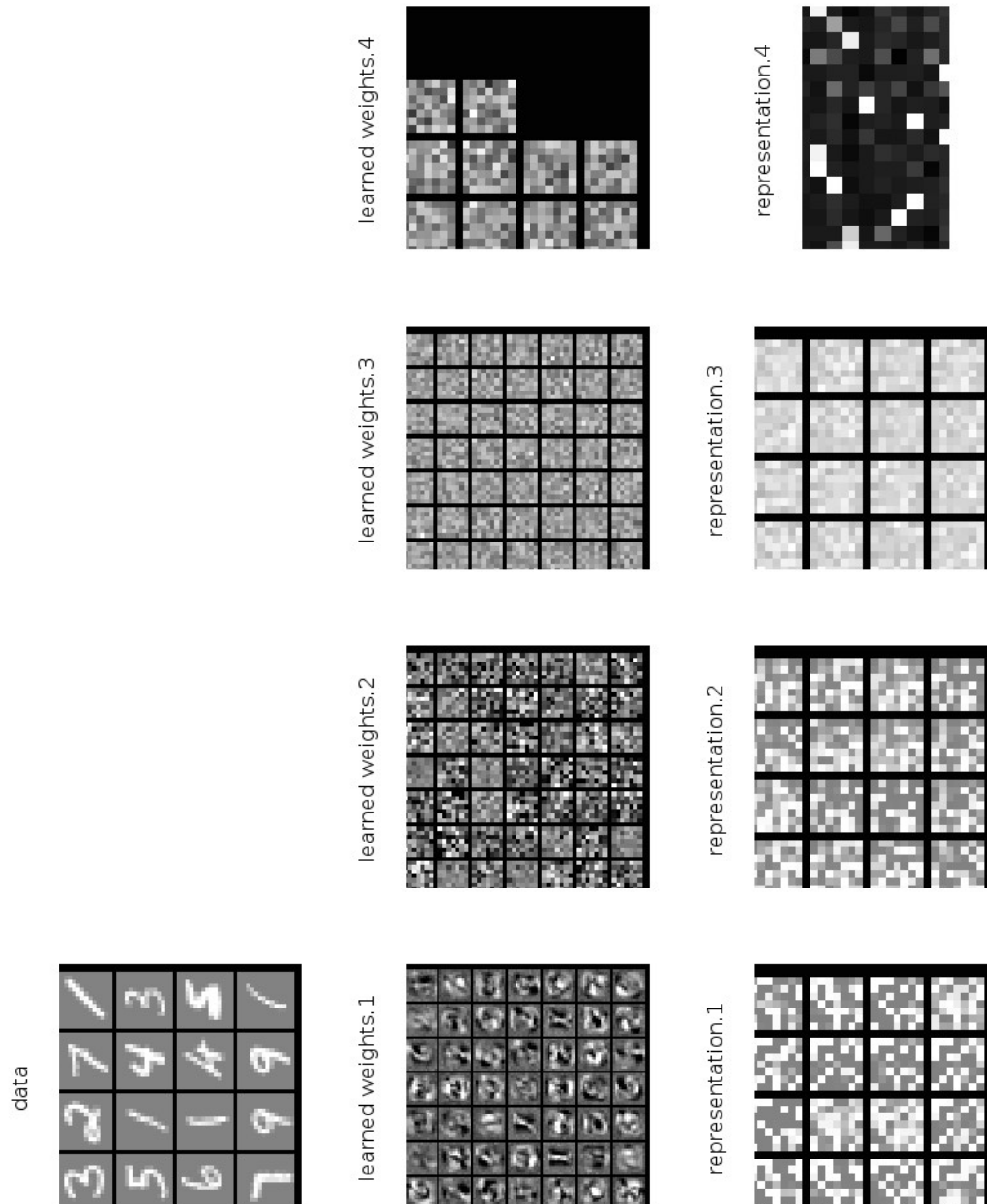

Figure 4.5.7: Classification with 2 layers of 49 units DBN: visually similar results to those of the multilayer perceptrons.

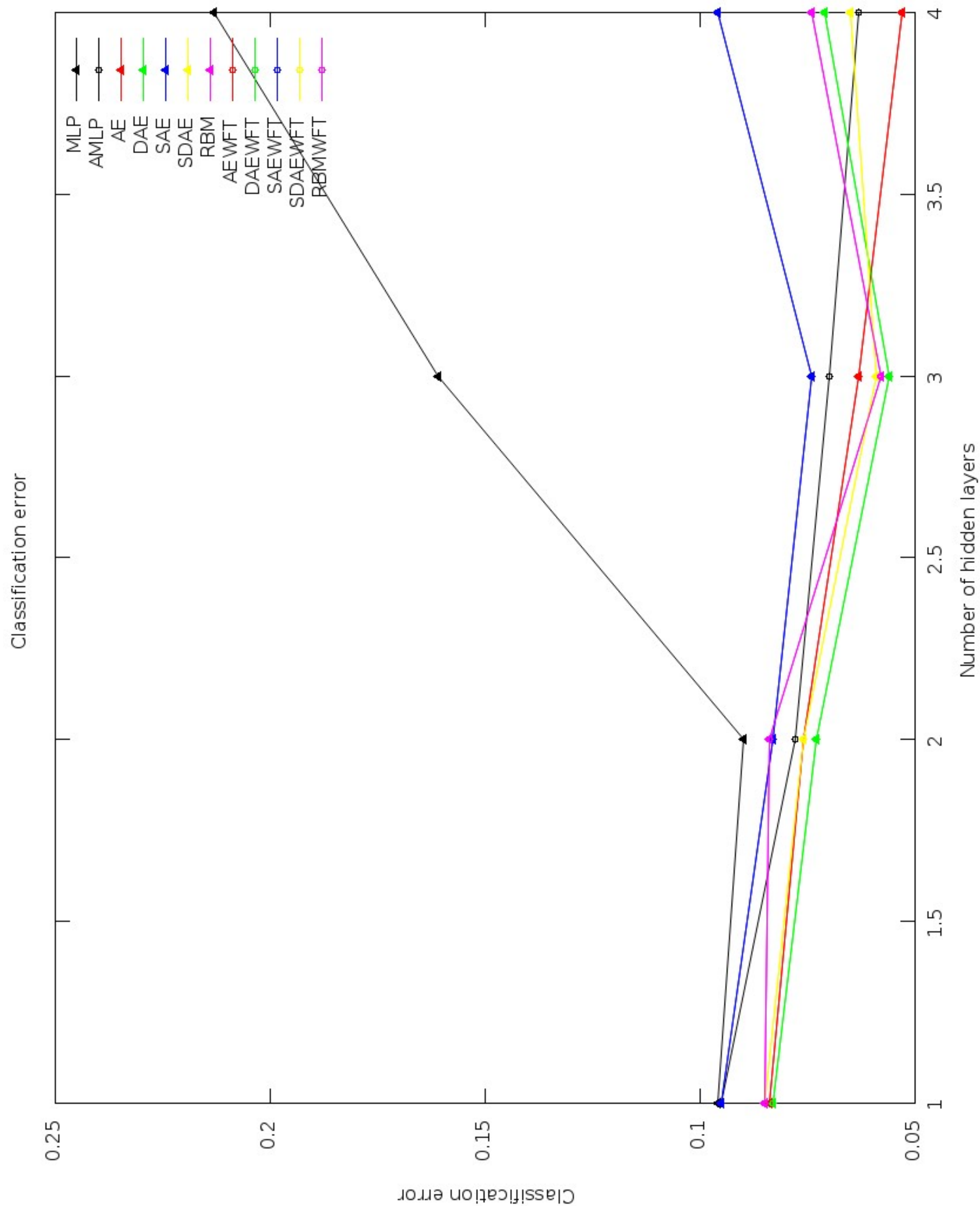

Figure 4.5.8: Classification errors for deep networks with 49 units per layer: The multilayer perceptron with standard initialization has bad performance when the number of hidden layers is high. The other networks perform well, but only the AMLP and SAE get better with more than 3 hidden layers.

Chapter 5

Conclusion

The objective of this master thesis was to present the state of the art in deep neural networks design: understanding the need of deep architectures, discussing related topics, and studying the most prevalent types of deep neural networks.

Also, we wanted to compare experimentally which architectures perform better at each combination of number of units per hidden layer in the case of reconstruction, and number of hidden layers, units per hidden layer, and initialization algorithm in the case of classification. As we have seen, the advanced random initialization for the multilayer perceptron gives good results, and is an interesting improvement that has trivial computational cost and is easy to implement. The greedy layer-wise pre-training algorithm, both using autoencoders and restricted Boltzmann machines, seems a more complex alternative. However, it also gives good results and is interesting from the theoretical point of view of being totally unsupervised and probably related to the mechanisms employed in human learning.

The experiments performed in this master thesis are limited due to the lack of hardware powerful enough to run a big network. The effect of a longer learning, with more epochs and training patterns, and with higher dimensional data and wider hidden layers remains unchecked, and possibly the results would be dramatically affected by changes to these parameters.

5.1 Further work

As future work, the first objective is to develop an efficient implementation to deal with large problems, like complete MNIST or CIFAR. We suggest an implementation of the algorithms in a map-reduce form, which will give us the opportunity to run experiments in computer clusters, with the consequent improvement in computational power. Some advances have been made in the last months in terms of theory, scale and complexity of the deep networks, as can be seen in [33] and [34].

Also, we could study the effect of combining the algorithm for the selection of the connections in a receptive field with the greedy layer-wise pre-training used in stacked autoencoders or deep belief networks, which could simplify the training complexity

and is also interesting from the biological point of view as it evokes the columns found in the visual cortex.

Another proposal is to use some evolutionary strategies like CMAES that could be interesting to improve training, specially when the error function has many sub-optimal solutions, or when the training algorithm is stochastic, like in Boltzmann machines.

Apart from classification problems, the use of deep networks for other applications like regression and modeling time series should be studied.

Appendix A

Information and entropy

Let X be a discrete random variable, that can take $2K+1$ values $\{X_k | k = 0, \pm 1, \pm 2, \ldots, \pm K\}$. The probability of the event $X = X_k$ is $p_k = p(X = X_k)$ and $0 \leq p_k \leq 1$, $\sum_{k=-K}^{K} p_k = 1$.

If we want to measure the information that we receive with each outcome of a random variable realization, we must take some details into account: if the event $X = x_k$ occurs with probability $p_k = 1$, and so $p_i = 0\ \forall i \neq k$, then we don't obtain information with each realization, because we already know the result. On the other side, if the probability p_k is small, it is more surprising that X takes the value x_k and, if it does, we gain more information than when it takes a more probable value. It is clear that the amount of information is related to the inverse of the probability of the outcome.

Having this into account, we can define the quantity of information we obtain observing the event $X = x_k$, that has probability p_k, as the function

$$\mathbb{I}(x_k) = \log \frac{1}{p_k} = -\log p_k$$

where, if the logarithm is natural, the information units are called *nats*, and, if the logarithm is binary, *bits*.

The function has the following properties:

1. $\mathbb{I}(x_k) = 0$ when $p_k = 1$.
2. $\mathbb{I}(x_k) \geq 0$ when $0 \leq p_k \leq 1$.
3. $\mathbb{I}(x_k) > \mathbb{I}(x_{k'})$ when $p_k < p_{k'}$.

The expected value of $\mathbb{I}(x_k)$ over the rank of $2K+1$ possible values is known as *entropy* because of the analogy with the thermodynamic entropy. From now on, we

will take $0 \log 0 = 0$

$$\begin{aligned}\mathbb{H}(X) &= \mathbb{E}\left[\mathbb{I}(x_k)\right] \\ &= \sum_{k=-K}^{K} p_k \mathbb{I}(x_k) \\ &= -\sum_{k=-K}^{K} p_k \log p_k\end{aligned}$$

The entropy meets that $0 \leq \mathbb{H}(X) \leq \log(2K+1)$. It is 0 if and only if $p_k = 1$ for a k and 0 for the rest, case that corresponds with the absence of uncertainty, and is $\log(2K+1)$ iff $p_k = \frac{1}{2K+1}$ $\forall k$, case that corresponds with the maximal uncertainty. The following definition and inequality are necessary to prove the upper bound and other results.

Given two probability distributions p_k and q_k for a discrete random variable X, then we define the *Kullback-Leibler divergence* or *relative entropy*

$$\mathbb{D}_{KL}(p||q) = \sum_k p_k \log\left(\frac{p_k}{q_k}\right)$$

that meets the following inequality

Theorem 3 (Fundamental information inequality)**.**

$$\mathbb{D}_{KL}(p||q) \geq 0$$

with the equality iff $q_k = p_k$ $\forall k$.

To prove this property we use that $\ln x \leq x - 1$ and $\log_a x = \frac{\log_b x}{\log_b a}$:

$$\begin{aligned}\mathbb{D}_{KL}(p||q) &= \sum_k p_k \log\left(\frac{p_k}{q_k}\right) \\ &= -\frac{1}{\ln 2}\sum_k p_k \ln\left(\frac{q_k}{p_k}\right) \\ &\geq \frac{1}{\ln 2}\sum_k p_k\left(1 - \frac{q_k}{p_k}\right) \\ &= \frac{1}{\ln 2}\left(\sum_k p_k - \sum_k q_k\right) \\ &= 0\end{aligned}$$

To prove the upper bound of the entropy we can use Lagrange multipliers, but with the previous inequality it is possible to do it directly, using the uniform density $\frac{1}{2K+1}$:

$$\begin{aligned} 0 &\leq \mathbb{D}_{KL}\left(p||1/(2K+1)\right) \\ &= \sum_k p_k \log \frac{p_k}{1/(2K+1)} \\ &= \sum_k p_k \log p_k + \sum_k p_k \log\left(2K+1\right) \\ &= -\mathbb{H}\left(X\right) + \log\left(2K+1\right) \end{aligned}$$

A.1 Mutual information

If we want to measure the relationship between, for example, the input and the output of a system, in terms of information, we can use the *mutual information*. Suppose that the input of the system is X, and the output Y, and that they only take discrete values. $\mathbb{H}(X)$ measures the prior uncertainty of X. What will be the posterior uncertainty of X, once we have observed Y? To answer this question we define the *conditional entropy*

$$\mathbb{H}\left(X|Y\right) = \mathbb{H}\left(X,Y\right) - \mathbb{H}\left(Y\right)$$

where

$$\mathbb{H}\left(X,Y\right) = -\sum_x \sum_y p(x,y) \log p(x,y)$$

is the *joint entropy* of X and Y, with $p(x,y)$ the joint distribution function.

This quantities meet that $0 \leq \mathbb{H}(X|Y) \leq \mathbb{H}(X) \leq \mathbb{H}(X,Y)$. The first inequality is clear. For the third, we just need to see that $\mathbb{H}(X,Y) = \mathbb{H}(X) + \mathbb{H}(Y|X)$, and for the second we have that $\mathbb{H}(X|Y) = \mathbb{H}(X,Y) - \mathbb{H}(Y) \leq \mathbb{H}(X) + \mathbb{H}(Y) - \mathbb{H}(Y) = \mathbb{H}(X)$, that, for the positiveness of the relative entropy

$$\begin{aligned} \mathbb{H}\left(X\right) + \mathbb{H}\left(Y\right) - \mathbb{H}\left(X,Y\right) &= -\sum_x \sum_y p(x,y) \log p(x) - \sum_x \sum_y p(x,y) \log p(y) \\ &\quad + \sum_x \sum_y p(x,y) \log p(x,y) \\ &= -\sum_x \sum_y p(x,y) \left(\log p(x) + \log p(y) - \log p(x,y)\right) \\ &= -\sum_x \sum_y p(x,y) \log\left(\frac{p(x)p(y)}{p(x,y)}\right) \\ &= \mathbb{D}_{KL}\left(p(x,y)||p(x)p(y)\right) \\ &\geq 0 \end{aligned}$$

that is a logical result, because the uncertainty over X cannot grow observing Y, and in general will decrease. It is easy to see that the equality is met only when the variables are independent.

The amount of uncertainty we resolve by watching Y is known as the *mutual information*

$$\begin{aligned}\mathbb{I}(X,Y) &= \mathbb{H}(X) - \mathbb{H}(X|Y) \\ &= \sum_x \sum_y p(x,y) \log\left(\frac{p(x,y)}{p(x)p(y)}\right) \\ &= \mathbb{D}_{KL}\left(p(x,y)||p(x)p(y)\right)\end{aligned}$$

that is a generalization of the entropy, because $\mathbb{H}(X) = \mathbb{I}(X,X)$.

The mutual information is symmetric, as $\mathbb{I}(Y,X) = \mathbb{I}(X,Y)$, and, as was said before, non negative.

Bibliography

[1] Simon Haykin, “Neural Networks: A Comprehensive Foundation”, Pearson Education.

[2] Richard O. Duda, Peter E. Hart, David G. Stork, “Pattern Classification”, John Wiley & Sons, Inc.

[3] Warren S. McCulloch, Walter H. Pitts, ”A logical calculus of the ideas immanent in nervous activity”, Bulletin of Mathematical Biophysics, 7:115-133, 1943.

[4] Frank Rosenblatt, “The Perceptron: A probabilistic model for information storage and organization in the brain”, Psychological Review, 1958.

[5] A. B. Novikoff, “On convergence proofs on perceptrons”, Symposium on the Mathematical Theory of Automata, 12, 615-622. Polytechnic Institute of Brooklyn, 1962.

[6] Bernard Widrow, Marcian E. Hoff, “Adaptive Switching Circuits”, Stanford University, 1960.

[7] Scott A. Czepiel, “Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation”.

[8] Johan Hastad, Mikael Goldmann, “On the power of small-depth threshold circuits”, 1991.

[9] Yoshua Bengio, “Learning Deep Architectures for AI”, Technical Report 1312, Université de Montréal, 2011.

[10] Xavier Glorot, Yoshua Bengio, “Understanding the difficulty of training deep feedforward neural networks”, Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, 2010.

[11] Christopher M. Bishop. “Pattern Recognition and Machine Learning”, Springer, 2006.

[12] Michael E. Tipping, Christopher M. Bishop. “Probabilistic Principal Component Analysis”, Journal of the Royal Statistical Society, Series B, 61, Part 3, pp. 611-622.

[13] James V. Stone, “Independent Components Analysis”, Sheffield University.

[14] Aapo Hyvärinen, "Survey on Independent Component Analysis", Helsinki University of Technology.

[15] H. Bourlard, Y. Kamp, "Auto-Association by Multilayer Perceptrons and Singular Value Decomposition", Biological Cybernetics, 59, 291-294, 1988.

[16] H. Bourlard, Y. Kamp, "Auto-Association by Multilayer Perceptrons and Singular Value Decomposition", IDIAP Research, 2000.

[17] Pierre Baldi, Kurt Hornik, "Neural Networks and Principal Component Analysis: Learning from Examples Without Local Minima", University of California, San Diego, Neural Networks, Vol. 2, 53-58, 1989.

[18] Andrew Ng, "Sparse autoencoder", Stanford University, CS294A Lecture notes.

[19] Andrew Ng, Jiquan Ngiam, Chuan Yu Foo, Yifan Mai, Caroline Suen, "Unsupervised Feature Learning and Deep Learning Tutorial", Stanford University, 2011.

[20] Theano Development Team, "Deep Learning Tutorial", LISA lab.

[21] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", Journal of Machine Learning Research 1 I, 2010, 3371-3408.

[22] Mark M. Wilde, "From Classical to Quantum Shannon Theory", McGill University, 2011.

[23] Kyung Hyun Cho, "Improved Learning Algorithms for Restricted Boltzmann Machines", Master Thesis, Aalto University, 2011.

[24] Hugo Larochelle, Yoshua Bengio, "Classification using Discriminative Restricted Boltzmann Machines", Proceedings of the 25th International Conference on Machine Learning, 2008.

[25] David H. Ackley, Geoffrey E. Hinton, Terrence J. Sejnowski, "A Learning Algorithm for Boltzmann Machines", Cognitive Science 9, 147-169, 1985.

[26] Geoffrey E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines", University of Toronto, UTML TR 2010-003, 2010.

[27] Geoffrey E. Hinton, Simon Osindero, Yee-Whye Teh, "A Fast Learning Algorithm for Deep Belief Nets", Neural Computation 18, 1527-1554, 2006.

[28] John J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", PNAS 79 2554, 1982.

[29] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, "Greedy Layer-Wise Training of Deep Networks", Université de Montréal.

[30] Athina Spiliopoulou, “Investigation of Deep CRBM Networks in modeling Sequential Data”, Master Thesis, University of Edinburgh, 2008.

[31] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, “Gradient-Based Learning Applied to Document Recognition”, Proc. of the IEEE, November 1998.

[32] Adam Coates, Andrew Y. Ng, “Selecting Receptive Fields in Deep Networks”.

[33] Salah Rifai, Grégoire Mesnil, Pascal Vincent, Xavier Muller, Yoshua Bengio, Yann Dauphin, Xavier Glorot, “Higher Order Contractive Auto-Encoder”.

[34] Quoc V. Le, Marc’Aurelio Ranzato, Rajat Monga, Matthieu Devin, Kai Chen, Greg S. Corrado, Jeff Dean, Andrew Y. Ng, ”Building high-level features using large scale unsupervised learning”, arXiv:1112.6209, 2012.