UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Fisica e Astronomia
"Galileo Galilei"

Corso di Laurea in Fisica

Network Architecture of Unsupervised Boltzmann Machines

Supervisor:
Samir Suweis
Co-Supervisor:
Alberto Testolin

Author:
Michele Piccolini

Anno Accademico: 2015/2016

Acknowledgements

Grazie Alba, per essermi stata sempre vicina nei momenti difficili.

Contents

Acknowledgements

1 Introduction **1**

2 Basics **2**
2.1 Bayesian Inference . . . 2
2.1.1 Bayesian probability and Conditional Dependence . 2
2.1.2 Likelihood, Prior, Posterior and Model Comparison . 2
2.1.3 Maximum Likelihood . . . 3
2.2 Monte Carlo methods . . . 3
2.2.1 Gibbs Sampling . . . 3
2.2.2 Monte Carlo methods as Markov Chains . . . 4
2.3 Ising Models . . . 5
2.3.1 Boltzmann Equilibrium Distribution of Binary Arrays 5
2.3.2 Monte Carlo simulations in Ising Models . . . 5
2.4 Latent Variable Models . . . 6

3 Neural Networks **7**
3.1 Basic Concepts . . . 7
3.1.1 The Single Neuron . . . 7
3.1.2 Supervised Learning - Example of a Binary Classifier 8
3.1.3 Generative Learning VS Discriminative Learning . . 9
3.2 Hopfield Networks . . . 9
3.2.1 Associative memories . . . 10
3.3 Boltzmann Machines . . . 10
3.3.1 Boltzmann Machines with Hidden Units . . . 11
3.3.2 Restricted Boltzmann Machine . . . 12
3.3.3 Improvements - Contrastive Divergence . . . 12
3.4 Multilayer Networks . . . 12
3.4.1 Deep Belief Network . . . 12
3.4.2 Training a Deep Belief Net . . . 13

4 Network Architecture **16**
4.1 Types of Networks . . . 16
4.2 Network Properties . . . 16
4.3 Network Analysis of a Trained Deep Belief Net . . . 18
Receptive Field . . . 19
Weights Distribution . . . 20
Strengths Distribution . . . 22
Visual Representation of Weighted Matrices . . . 23
Components, Average Path Length, and Resilience . 28
4.4 Conclusions . . . 30

A **31**

Chapter 1

Introduction

Artificial Neural Networks (ANN) are information processing archetypes that draw inspiration from the biological networks of neurons. ANN are systems that learn by example, and whose main purpose, among others, is solving pattern recognition and data classification tasks.
The seminal work of McCulloch (a neuro-physiologist) and Walter Pits (a logician), first laid the foundations for the artificial neurons and ANN. During the eighties, many scientists saw the real potential of neural networks, after having tried different models for the understanding of the human brain. [2] At that time, the available technology imposed a serious hindrance to the success and the usability of those models. [2] After a span of discredit, the ANN field has again started to attract attention, specially thanks to the landmark work done by the psychologist and computer scientist Geoffrey Hinton. [2] Currently, neural networks offer a valid alternative approach to classical algorithmic computing. They do not need to be programmed for specific tasks, and therein lies their strength. Indeed, neural networks work under a parallel processing framework in order to solve problems, as opposed to the conventional sequential approach of computer programs. They also show an exceptional ability to detect trends and recognize patterns from sophisticated and defected data. Recent develops in ANN have been achieved by exploiting tools and ideas from interacting particle models rooted in Statistical Physics (Ising Model, Boltzmann Machines, etc..). ANNs have been successfully applied to a vast spectrum of fields and operations. They are currently used in finance, in medical diagnosis, in industrial process control, in chemistry and physics, in biological systems analysis, in data mining and classification tasks, and also in some fringe applications such as auto-driving cars designing and sports betting. [11] [16]
Our thesis wants to illustrate recent developments in ANN, and study the topological properties of a specific type of ANN using tools from graph theory. The work is divided in two main parts. First, it presents useful concepts and models such as Bayesian probability, Ising models, Monte Carlo methods, and simpler neural networks such as the single neuron and Boltzmann Machines. We then focus on understanding the mode of operation of a Deep Belief Network (DBN), a multi-layer neural network that works under the unsupervised learning framework (i.e., the ANN learns features of the training data without the need of labeled data that would 'direct' the learning process).
The second part of this work analyzes a trained DBN (qualified on reading digits images from the popular MNIST database [10]) from a network perspective. We inspect the topological properties of the DBN, making use of graph theory. The goal of this unprecedented analysis is to seek a deeper knowledge of the topological modifications that the DBN experiences during the training.

Chapter 2

Basics

2.1 Bayesian Inference

2.1.1 Bayesian probability and Conditional Dependence

We will use probability in the Bayesian meaning: a probability is a degree of belief in propositions that do not involve random variables. It's the meaning used, for example, in the sentence 'The probability that Mr. S. was the murderer of Mrs. S., given the evidence'.

A 'good' (satisfying the Cox probability axioms) set of beliefs can be mapped onto probabilities satisfying the properties: $P(false) = 0, P(true) = 1$, $0 \leq P(x) \leq 1$, $P(x) = 1 - P(\bar{x})$, $P(x, y) = P(x|y)P(y)$. [13]

In probability theory, *conditional dependence* is a relationship between two or more events that are dependent when a third event occurs. For example, if A and B are two events that individually affect the happening of a third event C, and do not directly affect each other, then, when the event C has not occurred, A and B are independent. Eventually the event C occurs, and now if event A occurs, the probability of occurrence of the event B will decrease. Hence, now the two events A and B become conditionally dependent.

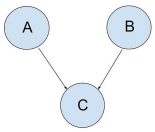


Figure 2.1: Let the event A be 'I have a new car'; event B be 'I have a new watch'; and event C be 'I am happy'. C occurs - 'I am happy'. Now if a third person sees my new watch, he/she will attribute this reason to my happiness. In his/her view the probability of the event A ('I have a new car') to have been the cause of the event C ('I am happy') will decrease as the event C has been explained away by the event B.

2.1.2 Likelihood, Prior, Posterior and Model Comparison

If θ denotes the unknown parameters, D denotes the data, and H denotes the overall hypothesis space, we can write the general equation from *Bayes theorem*:

$$P(\theta|D, H) = \frac{P(D|\theta, H)P(\theta|H)}{P(D|H)} \tag{2.1}$$

that can also be read as: posterior $= \frac{\text{likelihood} \times \text{prior}}{\text{evidence}}$.

Bayes theorem provides the correct language for describing the inference process (understanding a model from data, or improving a pre-existing model with the help of the data).

If we have two theories H_1 and H_2 whose credibility we intend to compare, we can use Bayes theorem. We can compute how probable the two theories are, given the data:

$$P(H_1|D) = \frac{P(D|H_1)P(H_1)}{P(D)} \qquad P(H_2|D) = \frac{P(D|H_2)P(H_2)}{P(D)}$$

We first need to assign prior probabilities to each of them (we could assign the same value if we have no prior knowledge at all). If we have a way of computing the terms $P(D|H_1), P(D|H_2)$, we will be able to compute the *posterior probability ratio* of model H_1 to model H_2. (A quantity that gives our degree of belief in one explanation for the data rather then the other.)

$$\frac{P(H_1|D)}{P(H_2|D)}$$

This idea can lead somehow to the hypothesis - hypothesis that is actually being studied in Neuroscience field - that the human cognitive system works in a similar way, by trying to attribute the most likely interpretation to the sensorial input data. [7] [6] [3]

2.1.3 Maximum Likelihood

When we need to find an explanation for something, we could perform exact inference by enumerating all possible hypothesis regarding our problem, and evaluate the corresponding probabilities. This approach is unfeasible most of the times as the space of all possible solutions is too big.
An approximate approach is homing in on one good hypothesis that fits the data well. This leads to the *maximum likelihood* method: finding the set of parameters θ of a theory that maximize the likelihood $P(D|\theta, H)$. We will see in the next chapter that this is the model that our neural networks will follow: 1) read input data; 2) build an internal generative model that could explain data; 3) try to maximize the likelihood (or "minimize the energy") of the data to improve the model.

2.2 Monte Carlo methods

Monte Carlo methods are computational techniques that make use of random numbers. The aims of Monte Carlo methods are to solve one or both of the following problems:

Sampling To generate samples $\{x_r\}_{r=1}^{R}$ from a given probability distribution $P(x)$.

Expectations To estimate expectations of functions under distribution, for example

$$\langle \Phi(x) \rangle_P := \int P(x)\Phi(x)\, \mathrm{d}^N x.$$

We call the probability distribution $P(x)$ *target density*. In our case it will be the posterior probability of a model's parameters given some observed data. Simple examples of Φ could be the first and second momentum of quantities that we wish to predict.

2.2.1 Gibbs Sampling

We will focus on sampling. Monte Carlo methods provide many sampling algorithms for different purposes. One possible algorithm to sample from multidimensional distributions is *Gibbs Sampling* (also known as *heat bath method*). We will make use of it.

Be $\boldsymbol{x}$ our K-dimensional random variable, distributed according to $P(\boldsymbol{x})$. We assume that $P(\boldsymbol{x})$ is too complex to draw samples from directly, but its 1 dimensional conditional distributions $P(x_i|\{x_j\}_{j\neq i})$ are most of the time tractable, i.e, random number can be easily be numerically sampled from it. Let's assume we have a data point $\boldsymbol{x}$, and we want to sample a new point $\boldsymbol{y}$ from $P(\boldsymbol{x})$. We denote by x_i the i-th component of the vector x, and by $x_{>j}$ the $(K-j)$-dimensional vector whose components are $\{x_i\}_{i=j,\dots,K}$. Similarly, we define the symbol $x_{<j}$. We also use $(u,\ v)$ to denote the vector whose components are the components of the vectors u and v, placed one after the other.

Gibbs algorithm

1 Using $\boldsymbol{x}$, we sample the first component of $\boldsymbol{y}$: y_1 from $P(y_1|\ x_{>1})$.

2 Using $\boldsymbol{x}$ AND the just calculated y_1, we sample y_2 from $P(y_2|\ (y_1,\ x_{>2}))$.

... Sample y_i from $P(y_i|\ (y_{<i},\ x_{>i}))$.

K Sample y_K from $P(y_K|\ y_{<K})$.

Essentially, what we are doing is 'upgrading' the data vector $\boldsymbol{x}$ to a new vector $\boldsymbol{y}$ step by step, sampling and 'updating' one component at a time, using, as prior knowledge to compute the i-th conditional distribution, the semi-updated vector $(y_{<i},\ x_{>i})$ instead of the old $\boldsymbol{x}$.
Gibbs is slow - because it explores the space state by a random walk -, unless a fortuitous parameterization has been chosen that makes the probability distribution $P(\boldsymbol{x})$ separable. Indeed, if some or all the components of our random variables are conditionally independent, the probability distribution from which it will be sampled will not depend on the previous values of the other components of the vector. This means that, if for example all the components of $\boldsymbol{x}$ are independent, we would be able to sample all the components of a new vector $\boldsymbol{y}$ in one very single step. [13]
Luckily, this will be our case.

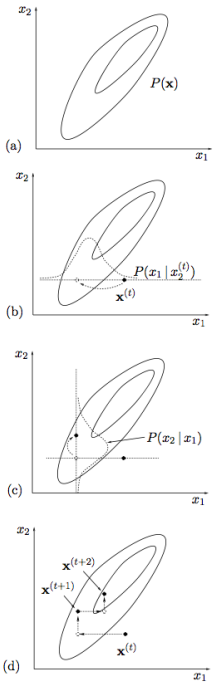


Figure 2.2:
Gibbs sampling for a 2D variable.
a) A point $x^{(t)}$ is used to start
b) The first component $x_1^{(t+1)}$ of a new point is sampled from $P(x_1|x_2^{(t)})$, given the *old* component 2 of $x^{(t)}$
c) The new second component $x_2^{(t+1)}$ is sampled from $P(x_2|x_1^{(t+1)})$, given the *new* component 1 of the point.

2.2.2 Monte Carlo methods as Markov Chains

Gibbs and other sampling methods are based on the theory of Markov Chains.
A Markov Chain is a process that can be specified by an *initial* probability distribution $p^0(x)$ and a *transition probability density* $T(x';x)$ (the probability of going from the state x to the state x'). [13]
The probability distribution of the state at the $(t+1)$-th iteration of the Markov chain, $p^{t+1}(x)$, is given by

$$p^{t+1}(x') = \int T(x';x)p^t(x)\mathrm{d}^N x \tag{2.2}$$

Required properties
For our purposes, we require the following properties:

Equilibrium The desired distribution $P(x)$ is *invariant* under the chain. That is, $P(x)$ is such that

$$P(x') = \int T(x';x)P(x)\mathrm{d}^N x. \tag{2.3}$$

Ergodicity $p^t(x) \to P(x)$ as $t \to \infty$, for any 'starting distribution' $p^0(x)$. Also known as "there aren't unreachable states".

2.3 Ising Models

2.3.1 Boltzmann Equilibrium Distribution of Binary Arrays

Figure 2.3: Example of array of spins with binary states (Up or Down).

An Ising model is an array of binary variables (e.g., atomic spins, or artificial neurons, that can take states ± 1, or $\{0, 1\}$) that are 'magnetically' coupled to each other. If one atom is, say, in the +1 state (or equivalently a neuron is in the 'ON' state) then it is energetically favorable for its immediate neighbors to be in the same state.
This can be mathematically represented by assigning an *energy* function to the state of the whole array x ($\in \mathbb{R}^K$), given the coupling constants $J_{mn} = J$ if m and n are neighbors and $J_{mn} = 0$, and the 'external magnetic field' H:

$$E(x; J, H) = -\left(\frac{1}{2}\sum_{m,n} J_{mn}x_m x_n + \sum_n H x_n\right) \tag{2.4}$$

This energy is useful to define the Boltzmann probability distribution of the system. We state that, at equilibrium the stationary state distribution for the spin system at the temperature T is ($\beta = \frac{1}{k_B T}$):

$$P(x|\beta, J, H) = \frac{1}{Z(\beta, J, H)} e^{-\beta E(x;J,H)} \tag{2.5}$$

Where $Z(\beta, J, H) := \sum_x e^{-\beta E(x;J,H)}$ is the normalization known as *partition function*.
By studying Ising models we can find out phase transitions in many systems that can be described within this model.
If we generalize the energy function to:

$$E(x; J, h) = -\left(\frac{1}{2}\sum_{m,n} J_{mn}x_m x_n + \sum_n h_n x_n\right) \tag{2.6}$$

we obtain a family of models known as 'spin glasses', or as 'Hopfield networks' or 'Boltzmann Machines' to the neural network community. Those models are useful for describing how the associative memory could work.

2.3.2 Monte Carlo simulations in Ising Models

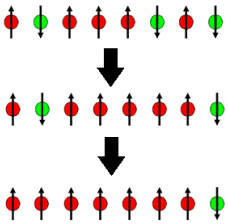


Figure 2.4: Example of the dynamic of the array of spins under a Monte Carlo simulation. Each step may flip a spin state.

Monte Carlo sampling methods (like Gibbs sampling) can be used to simulate an 'artificial dynamic' of the Ising array. Basically, in each step of the Markov chain, we flip, according to a transition probability that satisfies the detailed balance, the state of one unit at random. The transition probability is chosen in such a way that the system tends to change state so that it can lower its energy. This method brings, after several steps, the system state distribution to the equilibrium distribution, from which we can for example sample values or compute expectations of functions of the state.
This is the same mechanism implemented in a Hopfield neural network. [13] Simply

replace 'spin state' with 'neuron state', 'energy' with 'likelihood of the parameters of the internal model that explains input data', ' minimization of energy' with 'maximization of likelihood', and 'local magnetic field' with 'bias of a neuron' (its tendency to remain on or off).

2.4 Latent Variable Models

In statistics, *latent variables*, are variables that are not directly observed but are rather inferred, through a mathematical model, from other variables that are directly observed. Mathematical models that aim to explain observed variables in terms of latent variables are called *latent variable models*.
Sometimes latent variables correspond to aspects of physical reality, which could in principle be measured, but may not be for practical reasons. In this situation, the term *hidden variables* is commonly used (the variables are "really there", but hidden).
One advantage of using latent variables is that this reduces the dimensionality of data. A large number of observable variables can be aggregated in a model to represent an underlying concept, making it easier to understand the data.
Examples of latent variable models include *mixture models*, probabilistic models in which the observables are assumed to come from a superposed mixture of simple probability distributions. [5]

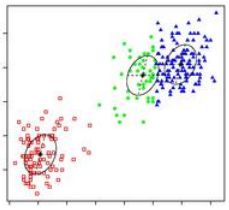


Figure 2.5:
A visual example of a Mixture of Gaussians. We initially have unlabeled data (all the points are the same color), then, we make a model and compute extimations of the parameters of this model (in this case, the label of each data point, and the shape of the underlying Gaussians we suppose the data come from). This image shows the outcome of the model for a particular data set, assuming the data can be assumed as generated from 3 Gaussians. Each data was colored with a color from among the 3 possible colors (1 for each Gaussian).

An example - Mixture of Gaussians
We now briefly discuss the example of a *mixture of Gaussians* [8] problem, to show an example of usage of *latent variables*.
Suppose that we are given a data set $D = \{x_n\}_{n=1}^{N}$, each of them coming with a hidden label t_n (each of them being a integer in $\{1, ..., k\}$). Our model posits that each x_n was generated by randomly choosing t_n from $\{1, ..., k\}$, and then x_n was drawn from one of k Gaussians depending on t_n. The random variables t_n indicate which of the k Gaussians each x_n had come from. This is called the *mixture of Gaussians* model. Also, note that the t_n's are *latent random variables*, meaning that they're hidden/unobserved.
We wish to model the data by specifying a joint distribution $p(x_n, t_n) = p(x_n|t_n)p(t_n)$. Here, $z \sim \text{Multinomial}(\phi)$, where $\phi_j \geq 0$, $\sum_{j=1}^{k} \phi_j = 1$, and the parameter ϕ_j gives $p(t_n = j)$, and $x_n|t_n = j \sim \mathcal{N}(\mu_j, \sigma_j)$.
The parameters of our model are thus ϕ (probabilities of the hidden labels), μ and σ (means and standard deviations of the Gaussians). The aim in this problem is to estimate them. We can succeed by writing down the likelihood of our data, and then trying to maximize it to find the best fitting parameters (usually with the use of an approximate iterative algorithm, since it is not possible to find the maximum likelihood estimates of the parameters in closed form). [4]

Chapter 3

Neural Networks

3.1 Basic Concepts

3.1.1 The Single Neuron

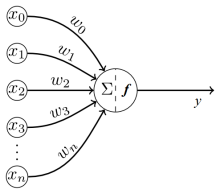


Figure 3.1: Structure of an artificial neuron.

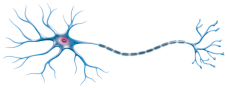


Figure 3.2: Structure of a biological neuron, from which the artificial one derives. The dendrites (the thin branches on the left) are the inputs, the cell body (the blob on the left) is the 'computation center', the axon (the long slender projection that goes from the body to the right) is the output.

Many neural network models are built out of single neurons.The single neuron (also called *Perceptron*) is a *feedforward* device - it is a network with a specific *architecture* that can perform the activity of taking input values and using them to compute and then 'fire' an output value, based on some inner rule (= the *Activity rule*). A single neuron is itself capable of 'learning' - indeed it can provide the simplest model for *supervised learning*. Therefore it is good to understand them in detail. [13]

Definition of a single neuron

Architecture A single artificial neuron is a unit that has I *inputs* x_i and one *output* y. Associated with each input is a *weight* w_i (see 1^{st} Figure). There may be an additional parameter w_0 called the *bias*, which may be seen as the weight associated with an input x_0 that is permanently set to 1 and that reaches all neurons. The *bias* is not necessary, but it can be useful in certain situations in which we want to impose an 'default tendency' in the activation of the neuron.

Activity rule The activity rule has two steps.

1 First, in response to the imposed input $\boldsymbol{x} \in \mathbb{R}^I$, the *activation* of the neuron is computed,

$$a = \sum_{i=0}^{I} w_i x_i = \boldsymbol{w} \cdot \boldsymbol{x} \tag{3.1}$$

2 The *output* is set as a function $f(a)$ of the activation. The output is also called the *activity* of the neuron ($\neq$ activation!).
There are several possible *activation functions* that can be used. The most popular are:

a) Deterministic Sigmoids

$$y(a) = \frac{1}{1+e^{-a}} \qquad \text{or} \qquad y(a) = \text{th}(a) \tag{3.2}$$

b) Stochastic activation functions:
y is stochastically selected from ± 1 for example with

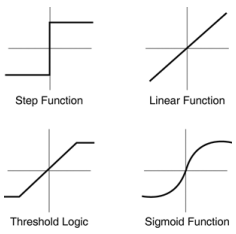


Figure 3.3: Several different activation functions are available. In the figure: threshold, linear, threshold logic, and sigmoid.

$$y(a) = \begin{cases} 1 & \text{with probability } (1+e^{-a})^{-1} \\ -1 & \text{otherwise} \end{cases} \tag{3.3}$$

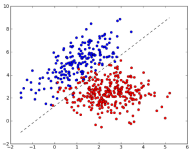


Figure 3.4: The purpose of the single neuron is to become able to discern between different kinds of data (for example to distinguish between blue and red data points in the figure).

3.1.2 Supervised Learning - Example of a Binary Classifier

We are given this problem: assume we have I-dimensional data that could be binary classified (each data point could belong to the '0' class or the '1' class). We would like to use an artificial neuron to perform the classification task. But the neuron doesn't know how to distinguish between data types, we must first train it!
What does exactly mean to 'learn a task'? In this context, it means becoming progressively better at doing something, proceeding by trial and error, with the help of a teacher. This way of learning is called *supervised learning*.
How do we translate all of this in a simple algorithm that makes use of our neuron? *We* can be the teachers for our neuron, and provide a *training data set* $\{\boldsymbol{x}^{(n)}\}_{n=1}^{N}$ ($\boldsymbol{x}^{(n)} \in \mathbb{R}^I \quad \forall\, n$) with a corresponding set of binary labels $\{t^{(n)}\}_{n=1}^{N}$. The labels are values $\in \{0,1\}$ specifying the correct class of each input $\boldsymbol{x}^{(n)}$.
Activity rule - The neuron could take an input $\boldsymbol{x}^{(n)}$ and perform its activity rule to guess an answer. It could use the sigmoid function $y(\boldsymbol{x},\boldsymbol{w}) = (1+e^{-\boldsymbol{x}\cdot\boldsymbol{w}})^{-1}$ to compute an output y (bonded between 0 and 1). This output might be viewed as stating the probability, according to the neuron, that the given input is in class 1 rather than class 0.
But if we don't 'teach' the neuron about the correctness of its response, it will never learn and improve. Its attempts will always be random.
We then write down an *error function*:

$$G(\boldsymbol{w}) = -\sum_n \left(t^{(n)} \ln(y(\boldsymbol{x}^{(n)};\boldsymbol{w})) + (1-t^{(n)}) \ln(1-y(\boldsymbol{x}^{(n)};\boldsymbol{w})) \right) \tag{3.4}$$

that can be viewed as the relative Shannon entropy between the empirical probability distribution constructed from the vector $(t^{(n)}, 1-t^{(n)})$ and the probability distribution obtained by the output of the neuron $(y, 1-y)$. This function quantifies the error in the answer given by the neuron.
'Learning', for the neuron, will mean to modify, after having seen and tried to classify an input, its weights $\boldsymbol{w}$, in such a way that the error (function) between the real label and the guessed one is decreased in value.
We can now define a learning rule for the neuron:
Learning rule - The *error signal* is computed as the difference from the supplied target value t and the output y

$$e = t - y$$

Then the weights are adjusted in a direction that would reduce the magnitude of the error function

$$w_i^{(\text{new})} := w_i^{(\text{old})} + \Delta w_i = w_i^{(\text{old})} + \eta\, e\, x_i$$

where η is the *learning rate*.The training consists then in repeating the activity rule and the learning rule for each input/target pair $(\boldsymbol{x}, t)$ that is presented.
This algorithm is called *backpropagation algorithm*, and it basically performs a gradient descent on the function G to approach its minimums, step after step. The fact

that it works it's easy to prove, since $\frac{\partial G}{\partial w_j} = \sum_{n=1}^{N} -(t^{(n)} - y^{(n)})x_j^{(n)}$.
At the end of the training, the neuron will have the information about its inner classification model stored in the values of its input weights. [13]

3.1.3 Generative Learning VS Discriminative Learning

Consider a classification problem in which we want to learn to distinguish between elephants (t = 1) and dogs (t = 0), based on some features of an animal. Given a training set, an algorithm like the perceptron algorithm tries to find a straight line (that is, a decision boundary) that separates the elephants and dogs. Then, to classify a new animal as either an elephant or a dog, it checks on which side of the decision boundary it falls, and makes its prediction accordingly.
Here's a different approach. First, looking at elephants, we can build a model of what elephants look like. Then, looking at dogs, we can build a separate model of what dogs look like. Finally, to classify a new animal, we can match the new animal against the elephant model, and match it against the dog model, to see whether the new animal looks more like the elephants or more like the dogs we had seen in the training set.
Algorithms that try to learn $p(t|\boldsymbol{x})$ directly, or algorithms that try to learn mappings directly from the space of inputs X to the labels 0, 1, (such as the Perceptron algorithm) are called *discriminative learning* algorithms.
A different approach is the one of the *generative learning* [9] algorithms, that instead try to model $p(\boldsymbol{x}|t)$ (and $p(t)$). For instance, if t indicates whether an example is a dog (0) or an elephant (1), then $p(\boldsymbol{x}|t = 0)$ models the distribution of dogs' features, and $p(\boldsymbol{x}|t = 1)$ models the distribution of elephants' features. [15]
The distribution $p(t|\boldsymbol{x})$ is the natural distribution for classifying a given example x into a class t, which is why algorithms that model this directly are called discriminative algorithms. Generative algorithms model $p(\boldsymbol{x}, t)$, which can be tranformed into $p(t|\boldsymbol{x})$ by applying Bayes rule and then used for classification. However, the distribution $p(\boldsymbol{x}, t)$ can also be used for other purposes. For example you could use $p(\boldsymbol{x}, t)$ to generate likely (x,t) pairs. [15]
Unsupervised learning is the machine learning task of inferring a function to describe hidden structure from unlabeled data. Usually, the term *unsupervised learning* is used to describe most generative models, since in this case we don't need conditional probabilities (supervision), but it suffices to observe variables (full joint distribution). Hereafter, we begin to see a new meaning of 'learning' as opposed to the supervised case. 'Learning' can be, for example, the process of building memories of the data and links between them (as we will see in the next section), or the process of trying to reproduce a seen input. All of this can be achieved *without* the need of a teacher that repeatedly corrects us. As you can imagine, this can be a much more efficient way of learning, because you don't always have the availability of large sets of labeled data.

3.2 Hopfield Networks

What we are going to do after having defined a single neuron, is attempting to connect multiple neurons together, making the output of one neuron be the input to

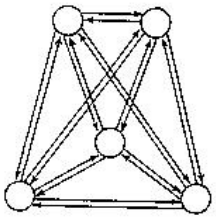


Figure 3.5: Example of a 5 neurons-fully connected Hopfield network. Each neuron's output serves as input for the others.

another, so as to make neural networks. A neural network will form a *graph* structure (or a *network*, in physical language).
A Hopfield network is a fully connected graph (each neuron takes the output of each other neuron as input), with the constraint that all the weights are *symmetric* (w_{ji} from i to j unit is equal to w_{ij} from j to i). Hopfield networks have two applications. First, they can act as *associative memories*. Second, they can be used to solve *optimization problems*. [13] Here we focus on associative memories.

3.2.1 Associative memories

Hopfield networks as associative memories are fully connected networks of units (single neurons) with:

Activity Rule Each neuron updates its state with a threshold activation function. Since each neuron has a feedback, we must choose if the updates are all synchronous or asynchronous.

Learning rule The weights are set using the so called *Hebb rule*: $w_{ij} = \eta \sum_{n=1}^{N} x_i^{(n)} x_j^{(n)}$, where η is the learning rate, and $x_i^{(n)}$ is the i-th component of the $\boldsymbol{x}^{(n)}$ input data.

The learning rule captures the idea of associative memory, because it modifies the weights between units according to the correlation between those units. Let's imagine that when stimulus i is present (for example, the smell of a banana), the activity of neuron i increases. Be the neuron j associated with another stimulus (the sight of a yellow object). If these two stimuli co-occur in the same environment, then the learning rule will increase the weights w_{ij} and w_{ji}. This means that when, on later occasion, stimulus i occurs in isolation, the positive weights w_{ij} will cause the neuron j also to be activated.
By training a Hopfield network with a data set $\{\boldsymbol{x}^{(n)}\}$, we make those patterns *stable states* of the Hopfield network's activity rule (i.e. they become favorable states in which the network could settle due to the dynamic of a training). [13]
The purpose of associative memory models is to be able to take a partial or corrupted memory (i.e. a grainy image, an incomplete word) and perform pattern completion or error correction to restore the original memory.

3.3 Boltzmann Machines

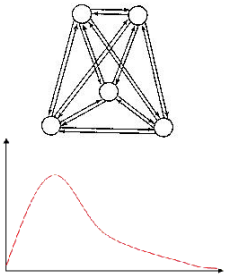


Figure 3.6: A Boltzmann machine is a Hopfield network that updates its states stochastically. The network is made to have its state follow a Boltzmann distribution.

Boltzmann machines are stochastic Hopfield networks. Each global state of the network can be assigned a single number called the *energy* of that state. The individual units can be made to act so as to *minimize the global energy*. The energy can be interpreted as the extent to which that combination of hypothesis violates the constraints implicit in the problem domain. The system then evolves towards 'interpretations' of that input that increasingly satisfy the constraints. The energy of a Hopfield network/Boltzmann machine for a configuration x is

$$E(x) = -\frac{1}{2}\boldsymbol{x}^T W \boldsymbol{x} \tag{3.5}$$

where W is the weight matrix: $(W)_{ij} = w_{ij}$.
A Boltzmann machine implements activity and learning rules that try to bring the

system configuration to an absolute minimum.

Activity Rule After computing the activation a_i, set $x_i = \begin{cases} 1 & \text{with probability } (1+e^{-2a_i})^{-1} \\ -1 & \text{otherwise} \end{cases}$
The noise introduce with the stochastic process helps the system to avoid getting stuck in local minima. [1] A system obeying this rule for changing its state will eventually reach a Boltzmann equilibrium distribution [18]

$$P(\boldsymbol{x}|W) = \frac{1}{Z(W)} e^{-E(\boldsymbol{x})} \tag{3.6}$$

Learning Rule Given a set of examples $\{\boldsymbol{x}^{(n)}\}_1^N$ we would like to adjust the weights W such that the model of the Boltzmann distribution is well matched by those examples. We can derive a learning algorithm by using maximum likelihood: we should maximize $\mathcal{L}(W) = \prod_{n=1}^{N} P(\boldsymbol{x}^{(n)}|W)$. This process brings to the rule: [13]

$$\Delta w_{ij} = \eta \sum_{n=1}^{N} \left(x_i^{(n)} x_j^{(n)} - \langle x_i x_j \rangle_{P(x|W)} \right) = \eta \, N \left(\langle x_i x_j \rangle_{Data} - \langle x_i x_j \rangle_{P(x|W)} \right)$$

This models thus learns by comparing the empirical correlation between units (computed from data) and the correlation between units under the current theoretical generative model. To estimate this second correlation we could use Monte Carlo methods. This is the step where the algorithm really slows down in computational time.
The two steps in which the calculation of the two correlations is made are picturesquely called *wake* and *sleep* phase. While the network is 'awake' it measures the correlation between the real x_i and x_j. While the network is 'asleep', it 'dreams' about the world using the generative model, and measures the correlations according to it. Finally, the weights are increased and decreased respectively in proportion to the 'awake' and the 'asleep' correlations. [1]

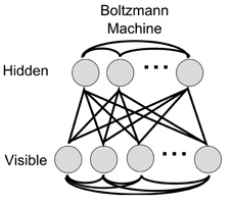


Figure 3.7: A Boltzmann machine with visible and hidden units is a classic Boltzmann machine, in which only one group of neurons takes the role of 'input neurons'. We start to see the presence of two different functional groups of neurons in a network.

3.3.1 Boltzmann Machines with Hidden Units

We now add *hidden neurons* to our model. These are neurons that do not correspond to observed variables. They usually take on interpretable roles, performing 'feature extraction'. We can identify two functional groups of neurons in this new network architecture: one group of *visible units* (input neurons), and one group of *hidden units* (feature extractors).The activity rule is identical to that of the original Boltzmann machine.
The learning rule can again be derived by maximum likelihood by taking into account the fact that the states of the hidden units are unknown. Be the state of the hidden units $\boldsymbol{h}$ ($\boldsymbol{h} \in \{-1,1\}^J$ if the network has J hidden neurons), and the total state $\boldsymbol{y}^{(n)} = (\boldsymbol{x}^{(n)}, \boldsymbol{h}) \in \{-1,1\}^I \times \{-1,1\}^J \quad \forall\, n$ if the network has I visible units and J hidden units. [13]
We can get to the *learning rule*, (analogous to the previous one):

$$\Delta w_{ij} = \sum_{n} \left(\langle y_i y_j \rangle_{P(h|x^{(n)},W)} - \langle y_i y_j \rangle_{P(x,h|W)} \right) \tag{3.7}$$

Here the first term in the sum is the correlation between y_i and y_j when the Boltzmann machine is simulated with the visible variables clamped to $\boldsymbol{x}^{(n)}$ and the hidden variables freely sampling from their conditional distribution. The second term is the correlation between y_i and y_j when the Boltzmann machine generates samples from its model distribution.
The learning rule is still time-consuming to simulate because it depends on taking the difference of two gradients both found by Monte Carlo methods.

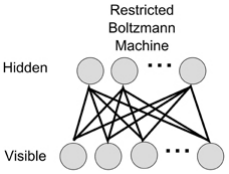


Figure 3.8: A Restricted Boltzmann Machine is a Boltzmann Machine with visible and hidden units, in which there are no connections between the neurons of a same group. (No hidden-hidden links, nor visible-visible links.)

3.3.2 Restricted Boltzmann Machine

A Restricted Boltzmann Machine (or RBM) is a variant of the Boltzmann machine with visible and hidden units, obtained by removing within-layer lateral connections to form a *bipartite graph*. This little change allows to perform efficient inference and learning. Block Gibbs sampling can be used to efficiently sample the probabilities and to update the neurons of an entire layer in a single step, instead of updating one neuron at a time, thanks to the absence of within-layer connections that would otherwise make the neurons in a layer conditionally dependent. [18]

3.3.3 Improvements - Contrastive Divergence

The use of Boltzmann machines was strongly discouraged by the very high computational demand of the learning algorithm, until the recent development of *contrastive divergence learning* (developed by Hinton in 2002). The original algorithm (with a data-driven *positive phase* and a model-driven *negative phase*) is slow because it implies running a Markov chain until convergence. The breakthrough that led to contrastive divergence is the finding that the negative phase does not need to be run until full equilibrium. If sampling starts from the hidden unit state computed in the positive phase (data-driven phase), correlations computed after a fixed number of steps in the Markov chain are sufficient to drive the weights toward a state in which the input data will be accurately reconstructed. Contrastive divergence gives good results even with a single step. [18]

3.4 Multilayer Networks

We can stack multiple layers of neurons to obtain systems capable of doing more complicated tasks (extracting high-order correlations between data or more complex features and patterns, improving binary and multi-class classification tasks, etc.). [13]

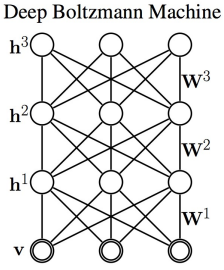


Figure 3.9: Structure of a Deep Belief Network (or Deep Boltzmann Machine - DBM). It is made of several RBNs stacked. Only links from one layer to another exist.

3.4.1 Deep Belief Network

A case of interest is the Deep Belief Net (DBN), a multilayer network built by stacking RBMs. The network is therefore composed of a visible layer of input neurons, and multiple hidden layers. Each layer takes the layer below as input.
Each RBM can be trained in an unsupervised 'greedy' way (they are trained one at a time, starting from the lowest, and going up). [18]
Using such a network architecture we can perform *deep learning*: the machine learning framework that exploits multiple layers of hidden units to build hierarchical representations of the input data. Indeed, by training a generative model at level l, using

as input the hidden causes discovered at level $l-1$, the network will progressively build more structured and abstract representations of the input data. The discovered representations are not tied to a particular discriminative task, because the aim of learning is only to model the hidden causes of the data. However, once the system has developed expressive abstract representations, possible supervised tasks can be carried out by introducing additional modules to perform a variety of supervised-fashion tasks. [18] For example, we could use a DBN, trained on just 'reading' and reconstructing images of digits, to recognize digits and classify them.
For the following thesis work we used DBNs that were trained on reading and reproducing handwritten digits.

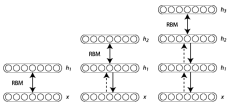


Figure 3.10: Training scheme of a DBN. The first RBM is first trained (left). Then a second RBM is trained by using as inputs the states of the first hidden layer that has just been trained (center). Then a third RBM is trained onto the second, and so on (right).

3.4.2 Training a Deep Belief Net

For our work, we trained DBNs composed of various layers. The main architecture we focused on was a network with a visible layer of 784 units (the input layer is this big because each neuron is responsible for the state of a pixel of an input image 28×28 pixels big), surmounted by 3 hidden layers (500-500-2000 hidden units respectively, for a total of about 1.6 million connections), trained on digits image reading task, using one-step contrastive divergence. The task of learning word perception can be seen as a stochastic inference problem where the goal is to estimate the posterior distribution over latent variables given the image of a word as input. [18]

Training Set

The training data set was the popular MNIST dataset [10] , that contains handwritten digits encoded as 28×28-pixel graylevel images. The dataset contains 60,000 training images.

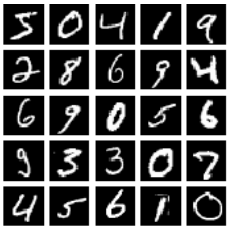


Figure 3.11: Some images from the MNIST digits data set.

Training Implementation

For the implementation of the algorithm we used MATLAB. The algorithm can work on the graphic processor unit (GPU) of a normal laptop, speeding up the process a lot (the training can take minutes instead of hours!), thanks to CUDA, a massive parallel computing framework for GPUs presented by NVIDIA. For this purpose, we exploited the high-level wrappers of CUDA provided by MATLAB via the Parallel-Computing Toolbox. [17] The use of such high-level functions greatly simplifies the parallelization, only requiring to work on *gpu array* data types.
The entire data set was divided into subsets called *mini-batches*. Rather than updating the network weights with the gradient computed on each training data pattern, the gradient in mini-batch learning is averaged over the patterns of the mini-batch. This improves convergence and learning speed.
Each RBM of the network is trained separately. First we train the RBM composed of visible layer + 1^{st} hidden layer, then 1^{st} hidden + 2^{nd} hidden, and so on. Every time using the states of the higher layer obtained in the previous training as base input for the training of the next, upper RBM.

Training Algorithm

Now we will explain the skeleton of the training algorithm. (The actual code we refer to can be found in the appendix A).
The variable `data_GPU` is a 'parallepiped' (3D matrix) containing the data of the

entire training set (60,000 images). We can think of it as a parallepiped made of rectangular 'slices' (each 'slice' is a mini-batch). There are 480 slices (number of batches), and each of them is a matrix of which each row contains a 'vectorized' input describing the state of the lower layer of the RBM. For example, in training the first RMB, each row of each batch matrix has 784 elements, containing the information on each pixel of one training image. Each matrix has 125 rows (dimension of the batch). Each batch then contains information about 125 possible inputs. (480 batches $\times$ 125 images-per-batch $=$ 60,000 images).
First, the biases (`visbiases_GPU` and `hidbiases_GPU`) are initialized to zero. The weights of the RBM that is going to be trained are instead initialized with random numbers sampled from a Gaussian with mean 0 and standard deviation 0.1:

$$W^i_j \quad \text{sampled from} \quad G(w;\ 0,\ 0.1) = \frac{1}{0.1\sqrt{2\pi}} e^{-\frac{1}{2}\frac{(w-0)^2}{0.1^2}} \tag{3.8}$$

The algorithm loops through the batches, and compute the *activations* for the neurons of the upper layer (`poshidprobs_GPU`) with a sigmoid, including biases. Be W^i_j the weight matrix for the RBM that is being trained (i for the lower units and j for the upper units), be D^k_i the mini-batch matrix (k is the indice for the k-th input image, and i is the indice for the i-th unit of the lower layer), and b^k_j the biases of the upper layer units (each b^k_j is equal to the others, when j is fixed), then the activations are (using Einstein summation notation):

$$a^k_j = (1 + \exp(-D^k_i\, W^i_j - b^k_j))^{-1} \tag{3.9}$$

Then `posprods_GPU` (P^i_j = data-driven correlation term between the i-th unit from the lower layer and the j-th unit from the upper layer) are computed. They are the *positive-phase* ('P' stands for 'positive') correlation terms.

$$P^i_j = (D^T A)^i{}_j = D^i{}_k\, a^k_j \tag{3.10}$$

The states, or activities, of the upper level units (`poshidstates_GPU` - that we will denote as h^k_j - for the unit j of the upper layer and the input image k) are stochastically set to 1 with probability given from the activities

$$h^k_j = \begin{cases} 1 & \text{with probability } a^k_j \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

In the *negative phase*, the neural network tries to recreate the data: `negdata_GPU`, that we will denote as $\hat{D}^k_i$ (D-hat), as opposed to D^k_i (D-without-hat), the real data batch matrix. A sigmoid is used to choose the 'dreamed' states for the lower layer units

$$\hat{D}^k_i = (1 + \exp(-H^k_j\, W^j_i - \hat{b}^k_i))^{-1} \tag{3.12}$$

where W^j_i is the transpose of W^i_j and $\hat{b}^k_i$ are the biases for the i-th units of the lower layer.
Then, model-driven activities (`neghidprobs_GPU` alias $\hat{a}^k_j$) are again computed, starting from the generated input

$$\hat{a}^k_j = (1 + \exp(-\hat{D}^k_i\, W^i_j - b^k_j))^{-1} \tag{3.13}$$

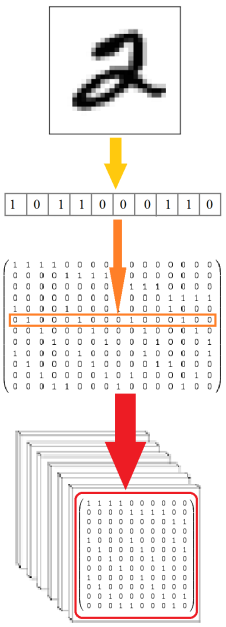


Figure 3.12: For the first RBM's training, each MNIST image is reduced to a vector of values in [0,1], then each of these vectors is put into a matrix (the mini-batch) as a row). 480 of these matrices form the whole data set: `data_GPU`.

The correlations (`negprods_GPU` alias N^i_j - 'N' is for 'negative' as opposed to P^i_j) for the model-driven (negative) phase are computed

$$N^i_j = (\hat{D}^T \hat{A})^i_j = \hat{D}^i_k \, \hat{a}^k_j \tag{3.14}$$

The error of the reconstruction is gathered as the sum of all the L^2 distances between the real data and the reproduced data of each batch. This will come in handy to monitor the quality of the ongoing reconstruction.
Weights and biases are finally updated. The update for the weights is

$$\Delta W^i_j = \frac{\eta}{125} (P^i_j - N^i_j) + \mu \, W^i_j - \lambda W^i_j \tag{3.15}$$

This is almost the same update rule of a classical Boltzmann machine, with two slight differences:

1) The *momentum* μ: a parameter whose value is empirically set, and whose purpose is to help the process avoiding getting stuck in local minima of the energy;

2) The *decay rate* λ, which is there to avoid overfitting, that is: to curb an excessive growth of the weight values.

This algorithm is repeated many times. Each of these 'big training loops' is called an *epoch*.

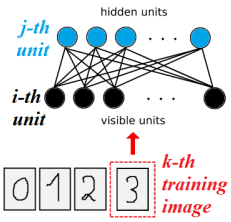


Figure 3.13:
i = 'below'
j = 'above'
k = 'training loop stage'
Indices notation: In this paragraph we use the indices i, j, k with precise, different meanings: **(i)** The letter i is used to denote the i-th unit of a *lower* layer of an RBM. [Black in figure] (e.g. if we are considering the first, lowest trained RBM, i will refer to a unit of the visible layer. If we consider the second RBM, the lower layer will instead be the first hidden layer of the whole network.) **(j)** j is used to refer to the j-th unit of the *upper* layer of the RBM. [Blue in figure] **(k)** The k indice refers to the *training image* that is currently been used for the training. [Red in figure] We can say for example: 'The k-th initial state of the lower RBM's layer', or 'The k-th row of a mini-batch matrix'. A 'training image' is an actual image (MNIST digit) only when training the lowest RBM of the network. From the second RBM on, a 'training image' is just the final state of the last trained RBM's upper layer.

Chapter 4

Network Architecture

4.1 Types of Networks

A *network* (or a *graph*, in mathematical language), is a set of items, which we will call *nodes*, with connections between them, called *edges* or *links*. A link is said to be *directed* if it runs only in one direction, and *undirected* if it runs in both directions. Directed links can be thought as arrows indicating their orientation.
Systems taking the form of networks abound in the world. Examples include the Internet, social networks, networks of business relations between companies or people, food webs, postal delivery routes, and last but not least, neural networks. [14]
In this part of the thesis, we introduce tools to analyze the structure and function of networked systems. These tools and properties that we are going to introduce will come in handy to make a network analysis of the DBNs we have trained and talked about in the previous chapter.
There are different types of *networks* or *graphs*, depending on their topological structure and their links properties. Here are some of them:

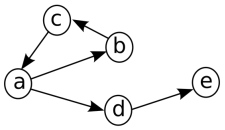


Figure 4.1: Directed graph (links are arrows)

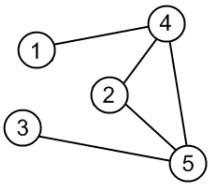


Figure 4.2: Undirected graph

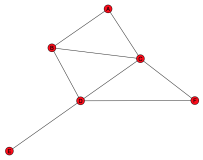


Figure 4.3: Simple graph (all links are equals)

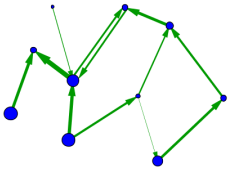


Figure 4.4: Weighted graph (links can be small or big)

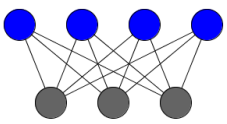


Figure 4.5: Bipartite graph

Directed A graph is *directed* (*digraph*) if all of its links are directed.

Undirected A graph is *undirected* if all of its links are undirected.

Simple/Weighted A graph can have *simple* links (either the link exist or doesn't exist), or can be a *weighted graph*, in which each link has a weight value that states the strength of the connection between two nodes. The latter is the case of our DBNs.

Bipartite *Bipartite graphs* are graphs containing nodes of two distinct types, with links running only between nodes of different types. This is the case of an RBM.

Acyclic An *acyclic graph* is a directed graph that shows no loops.

Random A *random graph* is a graph which is built by connecting nodes (originally not connected) with a random process (e.g. each link is built with a probability p).

Hypergraph A hyperedge is an edge joining more than two nodes together. Hypergraphs are graphs containing such edges.

4.2 Network Properties

Given a graph, one can define mathematically several network properties:

Degree In a simple graph, the *degree* of a node is the number of links connected to it. For a directed graph we must define both *in-degree* and *out-degree* for each node, which are the numbers of incoming and outgoing links respectively.

Strength In a weighted graph, we use the *strength* of a node in place of the *degree*. The *strength* is the total of the weights of the links connected to a node. *In-strength* and *out-strength* are defined as well.

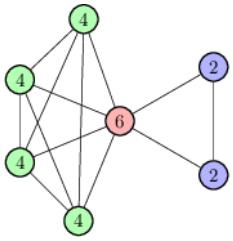


Figure 4.6: Illustration of degree 'k'. Green nodes have $k = 4$, blue nodes have $k = 2$, and red node has $k = 6$.

Simplification *To simplify* a weighted graph means to transform it to a somehow 'equivalent' simple graph, using some algorithm.
For example, one way of *simplifying* a graph would be using the rule: "a weight whose absolute value is greater than a certain threshold, becomes a link, whilst a weight whose absolute value is too little, implies that the link is removed".

Length of a path In simple graphs, the *length of a path* is the number of links which that path is made of. In a weighted graph instead we have a plethora of ways to define 'how long is a path'.
For example, one way would be to *simplify* the graph by making it a simple graph as described above. In this way, we obtain a 'discrete' version of the weighted graph, and we then can compute the lengths in the original simple way.

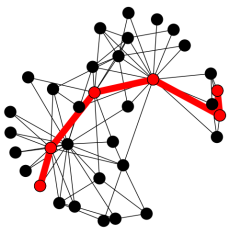


Figure 4.7: Red path is a geodesic linking two faraway nodes.

Geodesic A *geodesic path* from a node to another is the shortest path through the network between them.

APL *Average Path Length* is the mean of the geodesic paths in all the network.

Component The *component* to which a node belong is that set of nodes that can be reached from it by paths running along links of the graph. In a *digraph* a node has both an *in-component* and a *out-component*.

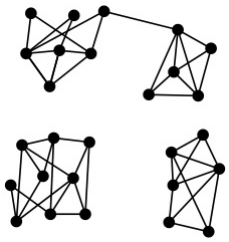


Figure 4.8: Example of graph with 3 separate connected components.

Weight matrix For a bipartite graph, the *weight matrix* (sometimes called *biadjacency matrix*) is the one we have seen before when talking about RBMs. It is the matrix W whose elements W_{ij} are the weights between the unit i of the 'lower' group of nodes (the lower layer, in a RBM) and the unit j of the 'upper' group of the graph (the upper layer of neurons). If the two parts of the bipartite graph have I and J nodes respectively, W is a $I \times J$ matrix.

Adjacency matrix For a simple graph with node set V, the *adjacency matrix* (sometimes called *connection/connectivity* matrix) is a square $|V| \times |V|$ matrix A such that its element A_{ij} is 1 when there is an edge from node i to node j, and 0 when there is no edge. The diagonal elements of the matrix are all zero, since edges from a node to itself are not allowed in simple graphs.
In a weighted graph, the *weighted adjacency matrix* has elements A_{ij} that are taken to be the weights between node i and j.
A is symmetric in an undirected graph.
The adjacency matrix A of a bipartite graph whose two parts have I and J nodes can be written in the form

$$\begin{pmatrix} 0_{I\times I} & W \\ W^T & 0_{J\times J} \end{pmatrix} \tag{4.1}$$

Where W is the weight matrix of the bipartite graph.
An adjacency matrix A of a graph which shows instead strong communities (a strong community is a subgraph in which each node is only linked with other nodes of the community it belongs), would be the exact opposite.

4.3 Network Analysis of a Trained Deep Belief Net

Now we have all the necessary tools in order to analyze and describe a trained neural network. Here, we examine the DBN whose training algorithm we have described in detail before. This is a multilayer neural network trained on digits images, and that is composed of 3 stacked RBMs (see architecture in Figure). The analysis of the network properties can be performed both on the DBN seen as a whole graph, and on the separate 3 RBMs that make the network. We in particular focused on analyzing the properties at different times in the training process - nominally, the beginning of the training (that we labeled as 't0', or 'time=0'), and the end of the training. Some specification must be made on what is 'the end of the training': we trained the network for 50 epochs (50 entire loops of training for all the 3 RBMs), and for 100 epochs. We could see the '50 epochs' (labeled as 't50' or 'time=50') situation as an intermediate step in the training that leads to '100 epochs' ('t100' or 'time=100'), but, as we will see, 50 epochs are more than enough to obtain a full satisfactory trained network. Hence, we took as the 'ending' situation of the training the 't50' state of the DBN in most cases. ('t100' and 't50' situations only differ slightly in network properties, therefore we can easily refer only to the 't50' case as the 'end of the training' state.).
An 'alternative' ending case is instead what we will call 't50 sparse', that is, we also trained the network with a *sparsity* constraint [18] , for 50 epochs. The *sparsity* imposition is a further step in the learning algorithm that can lead to a tweaked (often higher quality) trained network. This tweaked algorithm only affects the third hidden layer of the network, and calls for the following steps:

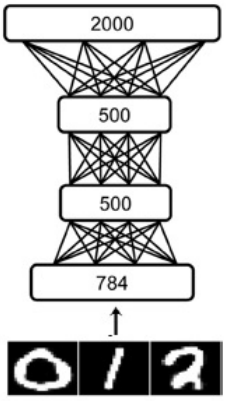


Figure 4.9: Structure of the analyzed DBN. The input layer gets one 28×28 (=784) pixels image at a time. Then, three hidden layers extracts abstract features on top of it.

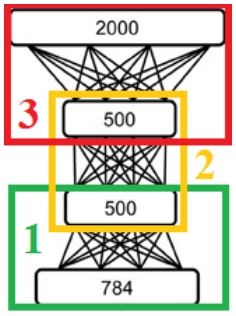


Figure 4.10: The DBN is made of 3 RBMs. Each 'upper' layer to an RBM serves as 'lower' (input) layer to the RBM above. Those 3 RBMs can be analyzed separately as bipartited graphs, each with its own properties.

(1) Inside each batch loop (i.e. once per batch), we compute the 'total' *activations* for each neuron in the last, third hidden layer of the network, that is we sum the activations $a^k\ j$ over the 'training image' index k:

$$A_j = \sum_k a_j^k \tag{4.2}$$

(2) Then we 'normalize' with respect to the batch size, and then compute the mean of this 'normalized total activities':

$$Q_j = A_j/\text{batchsize} = A_j/125 \quad , \qquad q = \sum_j Q_j \tag{4.3}$$

(3) Lastly, if q exceeds a certain threshold r, then we update the biases b_j of the neurons of the 3rd hidden layer so that they are lowered.

$$b_j = b_j - \eta(q - r) \tag{4.4}$$

This will make the neurons less prone to become activated in the following contrastive divergence training steps, and as a overall process should bring to a more sparse representation in the 3rd hidden layer of the hidden features in the data.

We will see later on the effects on the network properties that the *sparsity* brings.

Receptive Field

It is useful to first introduce the concept of *receptive field* [18] a neuron in order to have a powerful, straightforward visual tool for explaining our analysis results.
A neuron's *receptive field* is a visual representation of 'what neurons in the visual layer that specific neuron is responsible for'. Of course, since neurons in the visual layer are not linked, we can define the concept of *receptive field* only for neurons that live in upper, hidden layers.
The *receptive field* for a neuron of the first hidden layer is just the visual representation of the weights of its links toward the neurons in the visual, below layer. To plot the receptive field of the j-th neuron in the hidden layer 1 we just take the weight matrix of the 1st RBM: $W_{ij}^{(1)}$ ('1' stands for RBM 1), and extract the weights of the links that start from j and arrive to the below layer, i.e. the vector $W_{i\underline{j}}^{(1)}$ (j is fixed). We then reshape this vector to its 'original' square matrix form (we transform the vector $W_{i\underline{j}}^{(1)}$, whose dimension in 784, back to a matrix of 28×28 pixels, with the exact opposite process we used to 'vectorize' the input digits images during the training). Each i-th pixel will be wither or blacker in proportion to the value of the i-th weight value (white = positive, black = negative). By plotting this matrix we can obtain images like these: Clearly, these are impactful images that can give us a quick idea about

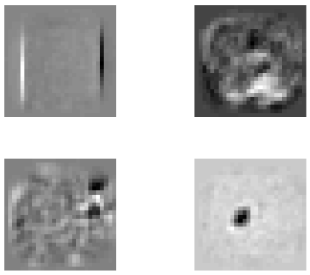


Figure 4.11: Some receptive fields

what parts and features of the training images a neuron has tied more. For example, a neuron could have been specialized in recognizing a straight vertical line to the left of the digits images, while completely having negative-valued links with an analogous straight line at the right side of the image (e.g. the first of the 4 images). Or it could have been able to strongly link with a circular-ish ring shape, while being anti-correlated with the space inside this ellipse (see 4th receptive field). Or further it could have assumed a distribution of weights that makes its receptive field a fuzzy blurred blob (as in the cases of the 2nd and 3rd images above). (A brief remarkable note on the first and second kind of receptive fields described: they show an astonishing similarity with the Gabor and Gaussian filters, which are filters vastly used in

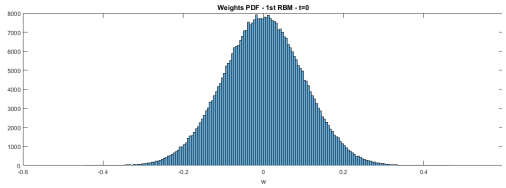
the field of image processing to enhance images or detect edges and similar features). How can we define a *receptive field* for a neuron that lives in the second or third hidden layer, since they are not directly connected to any of the neurons in the visible layer? In this case We define the receptive field in an indirect way (since there is no direct linkage or responsibility of higher neurons over the lowest ones). For a hidden layer 2 neuron (say, l), we first make a weighted average of the receptive fields of the neurons in the hidden layer 1, using the weights $W_{jl}^{(2)}$ that leave from the l-th neuron and link it with the below neurons. Then, reshape as a matrix and then plot the vector that we have obtained. Shortly, we plot $\sum_j W_{ij}^{(1)} W_{j\underline{l}}^{(2)}$ (l is fixed), obtaining an image of what pixels our l-th neuron in the second hidden layer is responsible for. For a neuron in the third layer we plot in a like manner: $\sum_{j,l} W_{ij}^{(1)} W_{jl}^{(2)} W_{l\underline{m}}^{(3)}$. We obtain images similar to a hidden layer 1 receptive field, but they are often more 'structured' images (see side Figures).

Figure 4.12: Receptive field of a hidden layer 2 neuron.

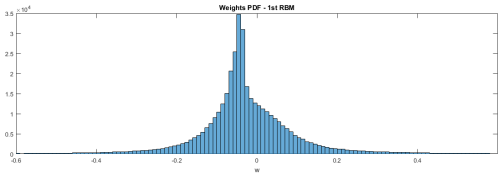
Figure 4.13: Receptive of a hidden layer 3 neuron.

Weights Distribution

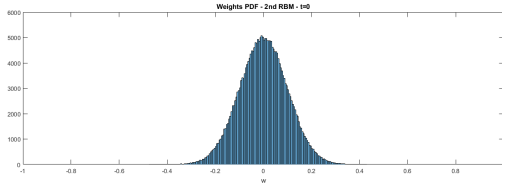
We took the weight values of each of the 3 RBMs of the network, and plotted their distributions at the beginning (t0) and at the end (t50) of the training. The next plots show these distributions. (The horizontal scales are equal for the plots in the same row, while they could be slightly different for plots in column).



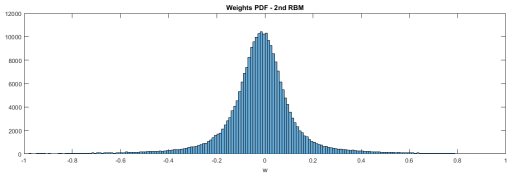
(a) W distribution, RBM 1, t0



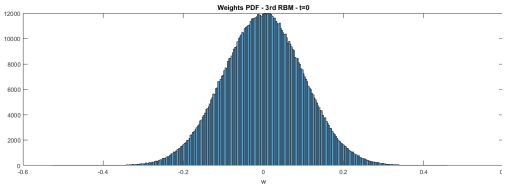
(b) W distribution, RBM 1, t50



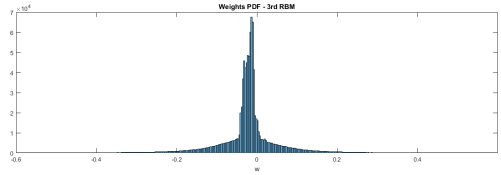
(c) W distribution, RBM 2, t0



(d) W distribution, RBM 2, t50



(e) W distribution, RBM 3, t0



(f) W distribution, RBM 3, t50

Figure 4.14: Weights distribution, separate RBMs

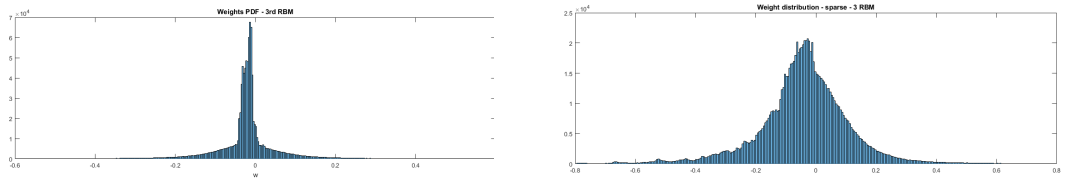
It is clear from the plots that the weights at 't0' (left column) follow a perfect Gaussian distribution (as we impose at the initialization step in the training algorithm). In the end (right column), the weights distributions are quite different. In all

the cases (RBM 1, 2, and 3), we can see a spreading of the tails of the distributions, in both directions (there appear some very positive-valued and some negative-valued values as well). In all the cases, the distributions tend to move leftward (this effect is stronger in the RBM 1 and RBM 3). What do we observe is therefore that many weights settle their values to a negative one. This could be explained by looking at the receptive fields of the neurons. Indeed, we observe that the neurons tend to specialize in 'describing' very narrow zones of the input layer. They usually becomes heavily positively linked with only a small part of the outstanding neurons, while becoming negatively linked with all the rest. This can be seen by the preponderance of darker pixels in the receptive field images. The network 'adopts' the strategy of training neurons of a layer so that they become able to describe small, localized features. They are taught to 'activate' in response to a very particular stimulus (stimuli of this kind could be, if we look for example at the receptive fields of the hidden layer 1 neurons, spots or straight bars in the input image), while keeping themselves anti-correlated with the rest of the inputs they can receive. An overall negative weight distribution is therefore plausible, at all the DBN's levels.
A second relevant information that can be drawn from the distributions is the highly-peaked shape of the RBM 3 distribution. To what can we attribute this outcome? We shall give a look to a sample of receptive fields of the neurons picked at random from the third layer.
There is no need to stare at the details of those pictures. Indeed, there is one thing

Figure 4.15: Some receptive fields of the 3rd hidden layer

that can be easily observed and that stands out: many receptive fields are *the same*. We notice a *redundancy* in the highest level features representations. Maybe a third hidden layer of 2000 units is too big for learning the series of features in the second hidden layer. Thus, we detected a lack of proficiency in the usage of the highest hidden layer.
The weight distributions reached after 100 epochs are almost the same of the ones reached at 50 epochs of training, then we will omit those additional superfluous

plots.
Actually, an interesting result is instead the one of the distributions reached with the sparsity constraint. Here we show the comparison between the 't50' distribution and the 't50 sparse' distribution for the third hidden layer (the one affected by the sparsity imposition). It can be seen that the direct effect of the sparsity constraint is the



(a) W distribution, RBM 3, t50

(b) W distribution, RBM 3, t50 sparse

Figure 4.16: Weights distribution 3rd hidden with sparsity

lowering of the high peak in the weight distribution. The whole final distribution has still a negative overall offset, but it is much more regular. It kind of resemble a Gaussian in its right part, while its left part is more irregular and it is higher on average than its right counterpart. As we can see by plotting some of the receptive fields, we have obtained a reduction of redundancy, and a more diverse set of images. Moreover, the structural complexity of these receptive fields suggests that this higher-layer of the network combines simpler features from the layers below in order to produce more useful, abstract representations of the input digits. [12]

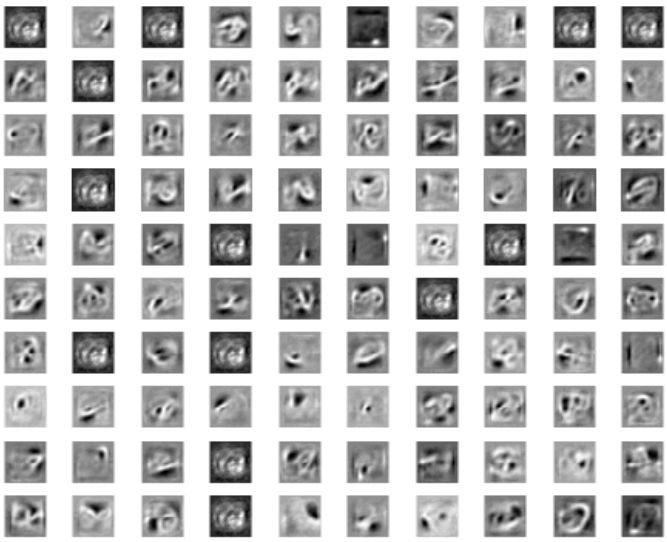


Figure 4.17: Some receptive fields of the 3rd hidden layer - with sparsity

Strengths Distribution

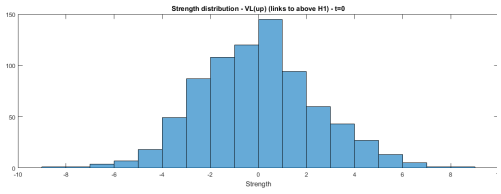
In this part of the analysis we focused on plotting the distributions of the undirected strengths, for each layer of each RBM of the DBN, taken separately. That is, we

considered one RBM at a time, and plotted the distributions of the strengths for each of the two layers of the DBN, only considering the links between those two layers. We end up then with 6 different distributions: one for the visible layer (lower layer of the RBM 1), one for the first hidden layer considered as the upper layer of the RBM 1, one for the first hidden layer this time considered as the lower layer of the RBM 2, two distributions for the hidden layer 2 considered as part of the RBM 2 and as part of RBM 3, and a final distribution of the neurons of the hidden layer 3 (that can be only considered as the upper layer of RBM 3). If we call the visible layer for a moment: 'layer 0', we could the identification numbers of each layer (0, 1, 2, 3) to label those 6 distributions as '0 to 1', '1 to 0', '1 to 2', '2 to 1', '2 to 3, and '3 to 2'.

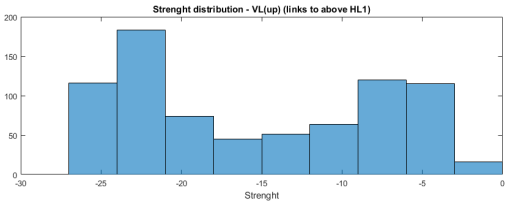
From the strengths distributions we can see that the initial distributions are still Gaussian. We expected this, since the strength is just the sum of the random weights of the links that leave or arrive from or to a particular neuron. Therefore, for the central limit theorem, the strength values follow a Gaussian distribution. As we can see, the final distributions are quite different. After the training, the neurons have an overall negative strength toward their closest layers. This reflects what we have already seen in the weight distribution analysis, since most negative weights give a negative strength when summed up. At the side of negative distributions' 'bulks' we can also see small, elongated positive tails, specially in the distributions '1 to 0', '1 to 2', '2 to 1'. Those distributions rapidly fall down near the 0, and then continue in the positive range of strengths with a long tail, reaching also very high values (up to $S \approx 30$ or $S \approx 40$). The first distribution ('0 to 1', or the distribution of the visible layer) is all negative. This tells us that the visible layer neurons have an 'average influence' on the first hidden layer's neurons that is always negative. Neurons in the visible layer are more strongly anti-correlated with the upper layer's neurons than they are correlated, and this could be due to the same reasons we addressed before when analyzing weights: the type of images used in training is very particular - black digits on a solid white background are very high contrast images, and they present 'something' (the digits lines) in only a small fraction of the whole picture. In the last distribution we can see the high peak of alike values of strengths among the neurons of the hidden layer 3. This is even a stronger hint than the ones we had before about the strong redundancy present in the last layer's neurons. Many units tend to fall, with the training, into an almost identical set of properties. This effects does not disappear with a longer training (read: 100 epochs). The 't100' strength distributions are almost identical to the 't50' ones, so we do not report the plots here. If we introduce the sparsity in the training, the third hidden layer strength distribution becomes more flattened, as we were expecting. Here the comparison between the 't50' and 't50 sparse' scenarios. The sparse case (at right) shows a more regular distribution, as we already saw in the weight distributions case. The new distribution resembles a Gaussian with a strong negative effect (the center of the bell is at about $S = -10$. Also, a small group of strongly negative values appears ($S \approx -80$), while in the normal 't50' case the negative tail of the distribution ends at $S = -30$.

Visual Representation of Weighted Matrices

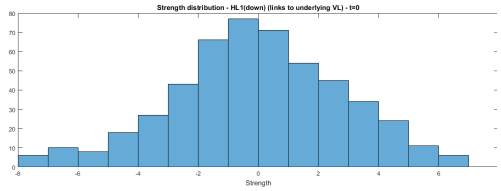
In this section we shall show some visual representations of the weighted matrices of the 3 RBMs of the trained DBN. What we do here is first simplifying the whole



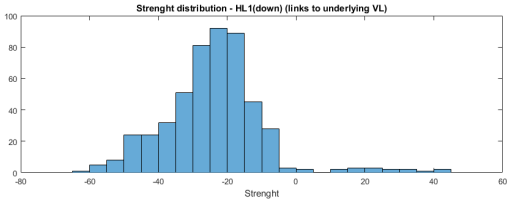
(a) 0 to 1 - S distribution, RBM 1, layer 0, t0



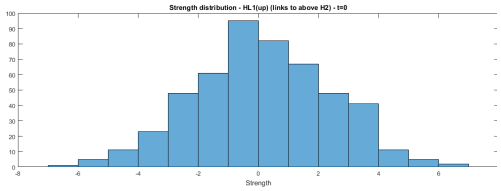
(b) 0 to 1 - S distribution, RBM 1, layer 0, t50



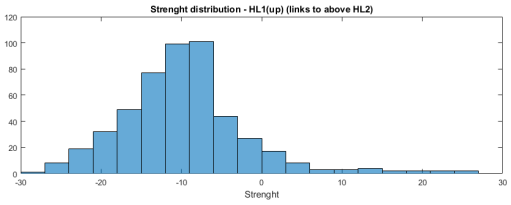
(c) 1 to 0 - S distribution, RBM 1, layer 1, t0



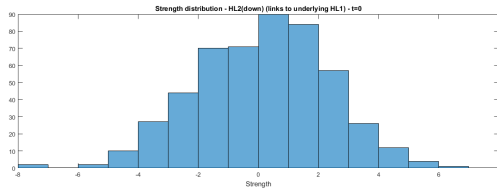
(d) 1 to 0 - S distribution, RBM 1, layer 1, t50



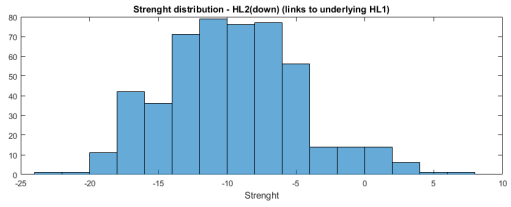
(e) 1 to 2 - S distribution, RBM 2, layer 1, t0



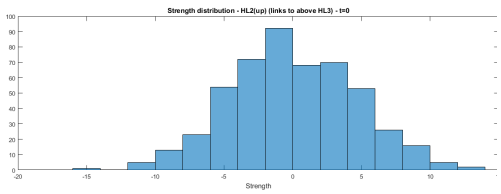
(f) 1 to 2 - S distribution, RBM 2, layer 1, t50



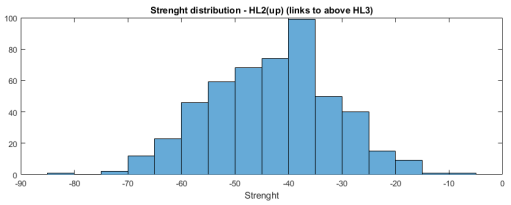
(g) 2 to 1 - S distribution, RBM 2, layer 2, t0



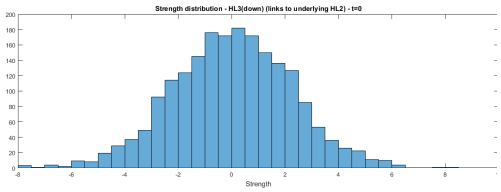
(h) 2 to 1 - S distribution, RBM 2, layer 2, t50



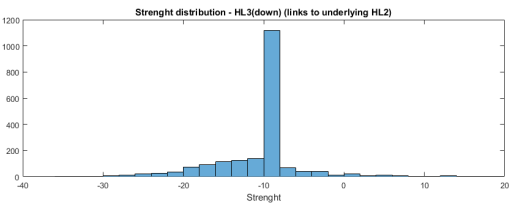
(i) 2 to 3 - S distribution, RBM 3, layer 2, t0



(j) 2 to 3 - S distribution, RBM 3, layer 2, t50



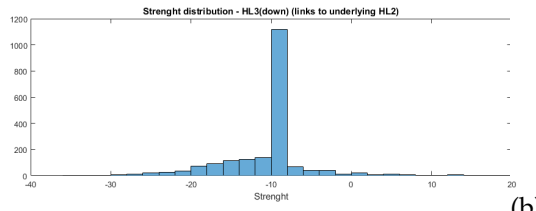
(k) 3 to 2 - S distribution, RBM 3, layer 3, t0



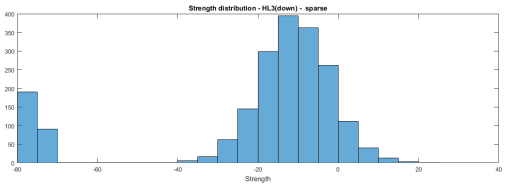
(l) 3 to 2 - S distribution, RBM 3, layer 3, t50

Figure 4.18: Strengths distribution, separate RBMs

network transforming it in a simple graph. The purpose for this is that, since we are going to show the weighted matrices as a whole, we want to keep the information



(a) 3 to 2 - S distribution, RBM 3, layer 3, t50



(b) 3 to 2 - S distribution, RBM 3, layer 3, t50 sparse

Figure 4.19: Strengths distribution 3rd hidden with sparsity

contained in these plots as simple and direct as possible. By simplifying the graph we make the weighted matrices binarized (they will contain, from now on, just zeros and ones). What we use here is the method described in the section Network Properties when we wrote about the Simplification process. We choose a threshold (after having tried empirically, the choice fell on a threshold of 0.2), and we transform weight values W_{ij} into 1 (if $|W_{ij}| > \text{threshold}$) or 0 (if the absolute value of the weight was lesser than the threshold). We choose this particular value because we have seen that the network architecture is simplified, but not enough to lose too much information about what neuron was linked to whom. We select only the 'strongest' weights, and we just look at their absolute value for this part of the analysis (the sign of the weight values is more of a functional property than a topological -i.e. about the structure of the linkage- one). See side figure for a picture of the simplified network. For this part of analysis we divide the DBN in the same way as we did in the previous one: we are going to look at the properties for *each* layer of *each* RBM. What we need is therefore 6 weight matrices, 1 for each layer, seen as the lower or the upper layer of an RBM. Each of those matrices has its rows (each row stands for a neuron in the layer we are considering) reordered with the strength value as a criterion. That is, we compute the total sums of values of each row, and then we put the highest valued rows on top, and the lowest at the bottom. Then, we plot the matrices as images, with a white pixel where the element's value was 1, black otherwise. We also make the surface plot of this image, adding a height to the white pixels (black pixels remain on the floor). Here are the results. As usual, we plot the matrix and its surface plot for the 't0' case (left) and the 't50' case (right), for each layer in each RBM, considering only the weights of the links in that RBM.

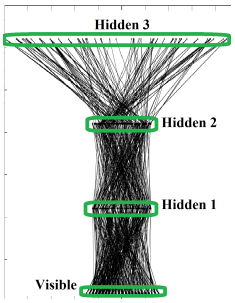
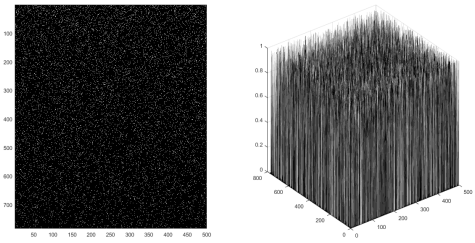
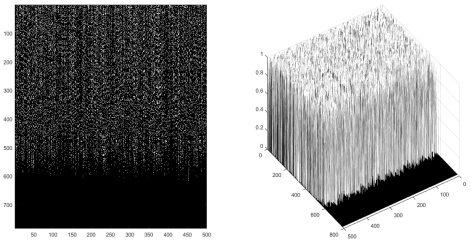


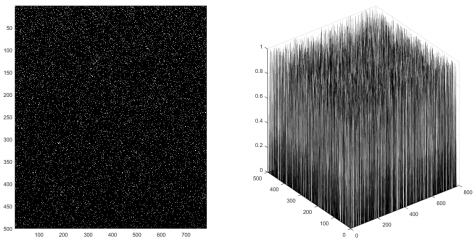
Figure 4.20: Representation of the simplified DBN. Threshold: 1.5



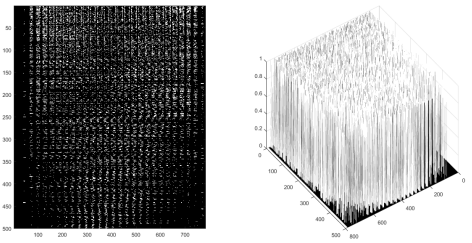
(a) 0 to 1 - Weight matrix and surface plot, RBM 1, layer 0, t0



(b) 0 to 1 - Weight matrix and surface plot, RBM 1, layer 0, t50

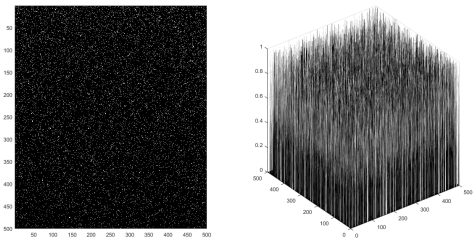


(c) 1 to 0 - Weight matrix and surface plot, RBM 1, layer 1, t0

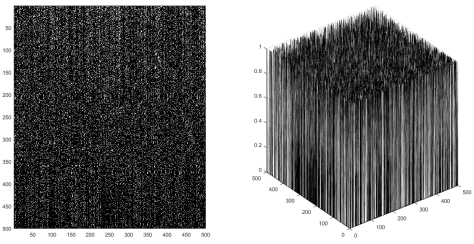


(d) 1 to 0 - Weight matrix and surface plot, RBM 1, layer 1, t50

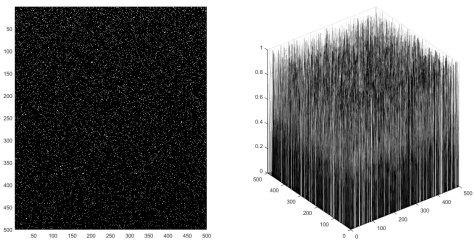
Figure 4.21: Visual reordered weight matrices, RBM 1



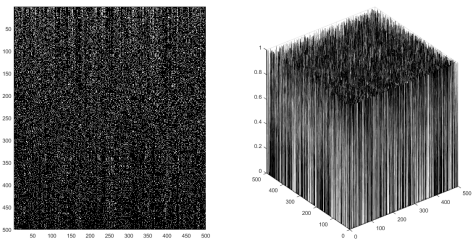
(a) 1 to 2 - Weight matrix and surface plot, RBM 2, layer 1, t0



(b) 1 to 2 - Weight matrix and surface plot, RBM 2, layer 1, t50

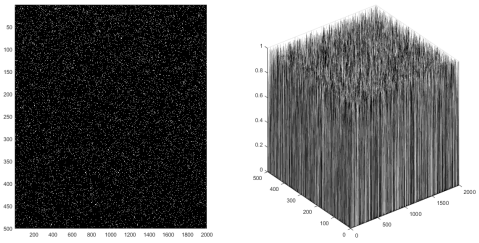


(c) 2 to 1 - Weight matrix and surface plot, RBM 2, layer 2, t0

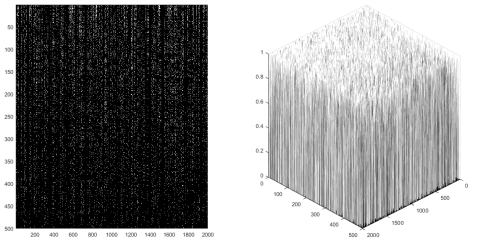


(d) 2 to 1 - Weight matrix and surface plot, RBM 2, layer 2, t50

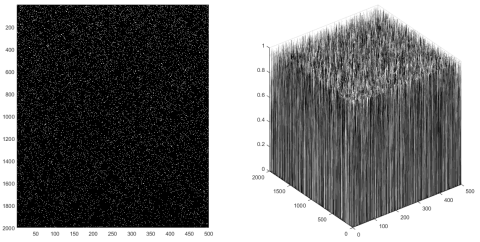
Figure 4.22: Visual reordered weight matrices, RBM 2



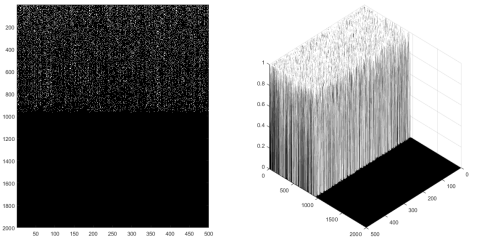
(a) 2 to 3 - Weight matrix and surface plot, RBM 3, layer 2, t0



(b) 2 to 3 - Weight matrix and surface plot, RBM 3, layer 2, t50



(c) 3 to 2 - Weight matrix and surface plot, RBM 3, layer 3, t0



(d) 3 to 2 - Weight matrix and surface plot, RBM 3, layer 3, t50

Figure 4.23: Visual reordered weight matrices, RBM 3

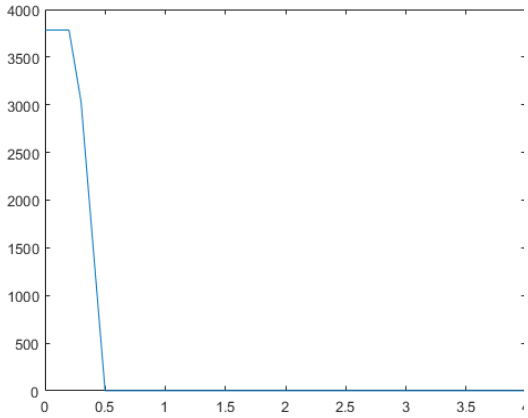
It can be seen that the initial states are quite randomized. Even after having reordered the rows based on strength, the pixels in the matrices do not appear to have an interesting structure. What we see in the trained weighted matrices is instead different. While the matrices relative to the RBM 2 do not have a particular structure, in the matrices of the RBM 1 and 3 we see something else. If we look at plot '0 to 1 - Weighted matrix and surface plot, RBM 1, layer 0, t50', it appears that a fair amount of neurons do not present strong enough connections with *all* the neurons in the hidden layer 1. Indeed, there are many solid black lines at the bottom of the represented matrix (as we can better see in the surface plot, that shows a sharp step). Those were neurons that have too little weights in its links toward the neurons of the other layer. What could be the reason for this? We think that this is another consequence of the type of input images that were used. The digits images have a huge amount of pixels that are actually never used, like the pixels in the borders of the images. Those pixels are always part of the background, for all digits from 0 to 9. Therefore, the neurons that represent those pixels in the input are almost disconnected from the above hidden layer 1. They never play a major role in the learning of features for any of the input images. The plot '1 to 0 - Weighted matrix and surface plot, RBM 1, layer 1, t50' kind of confirms this thesis, since it shows a particular pattern of vertical black stripes (the white pixels are not uniformly mixed as in the other cases). Since in this matrix the *columns* represent the neurons of the visible layer (while the *rows* represent the neurons of the first hidden layer), we might conclude that the neurons to whom the first hidden layer's neurons are little or no linked at all are always the same (a vertical black i-th column gives us the information that all the neurons in the hidden layer 1 -i.e. all the rows- are not connected with the i-th neuron that that column represent in the visible layer). We can notice that the presence of black vertical stripes is more consistent in the first and in the last column positions -they could

be the highest and the lowest picture in the training images, that are always part of the background, and are therefore unused. The last weighted matrix (the one of the third hidden layer neurons) shows an analogous sharp step as in the first one. This is attributable to the fact that there are a lot of neurons in the highest hidden layer that 'do nothing' (they are poorly linked with the entire below layer. This reinforces the thesis of the redundancy in the representations of the data features by the neurons in the last (too?) big layer. Indeed, in the sparse case, this effect (the sharp step) disappears (we do not show the plot for brevity).

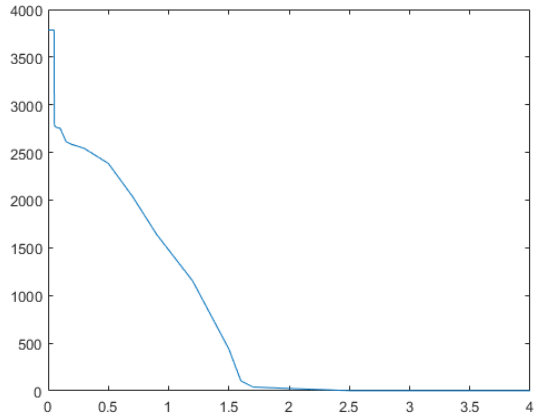
Components, Average Path Length, and Resilience

We lastly analyzed the size of the simplified network's components, and the average path length of the network, for different choices of simplification threshold. Raising the threshold means cutting more links off, while lowering it (to a minimum of 0) means keeping almost or all the links among all the neurons in close layers. Thus we used the threshold as a parameter for the 'destruction' of the network's structure. Even if this method doesn't provide a random removal of the network's links, it still gives us a good tool to study the *resilience* of the network. If we think of the links between neurons as something that tells us that the information can travel from one to another, then the removal of links could mean a damage to the network, since it could imply that the information that starts from a certain neuron in the visible layer is not able any more to reach neurons in the highest layer (and therefore the capacity of learning the features in the data could be ruined). We then analyzed the distributions of the sizes of the components in the simplified network, at different thresholds of simplification. What we discovered is that, at low thresholds, the distribution is always composed of a unique bin (all the neurons are part of a unique *giant component*), while at higher thresholds the distribution becomes always uniform (all the neurons start becoming disconnected, and many little isolated components arise) - this is the case of the completely spoiled network. The situation in between is usually made of a *giant component*, and a very low uniform tail of components of other sizes. The majority of the neurons still are part of a unique enormous connected component, and we can conclude that in this case the network is still 'healthy', since the information could travel virtually from any unit to any unit. The average path length is intimately related to the presence of the giant component. At threshold 0 the average path length tends to about 2 (since, for example, a neuron in the visible layer will be 1 link apart from the hidden layer 1's neurons, 2 links from the hidden layer 2's neurons, and 3 from the third, for a mean distance of approximately 2). At higher thresholds, we break almost all the links in the networks, and therefore the average path length goes to infinite (or it goes to 0, if we ignore in the computation the neurons who are not in the same component -each neuron will make its own component when they are all disconnected). Here we show the size of the giant component and the average path length as functions of the threshold. As usually, we show the 't0' randomly initialized case, and the 't50' trained case, for comparison.

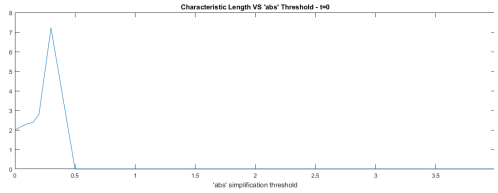
The two cases are different. In the 't0' case (we could see this as a random graph), the whole network forms a unique giant component when all the links are present, while it sharply breaks down to a multitude of paltry sized separate components. The presence of the giant component (that we associate with a healthy quality of the information travels through the network) suddenly disappears at the breaking of a small part of random links. This is also reflected in the average path length of the



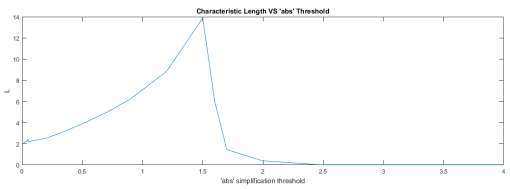
(a) Giant Component's size VS threshold, t0



(b) Giant Component's size VS th., t50



(c) Average path length VS th., t0



(d) Average path length VS th., t50

Figure 4.24: Giant component's size and APL as functions of threshold

network (image below), that rapidly increases and then it falls to 0 (or infinite, as we said before), meaning that the network was dismantled into small pieces. This transitions happen at about the same threshold, meaning that the two properties are inherent the same property that evolves in the network. The 't50' case shows that the network has trained itself to become more resilient. Indeed, while the 'low threshold' situation is the same as the random case -the network is a unique component-, this giant component doesn't dissolves immediately, but it gently decreases its size, until it fades altogether, leaving the network in the 'lots of isolated components' state. This transition happens at a highest threshold (and this is reflected by the average path length plot, in this case too) than in the random case. The trained network is therefore much more resistant to disruptions of its structure. The redundancy in the features representations that we notice in the previous sections could be a key factor in the high resilience of the network. Indeed, if a neuron is cut off from the network, the trained DBN could still perform well, because of the other neurons that, having the same combination of links of the cut off neuron, can still play the role of the departed one, without interrupting the information flux.
The 't100' and 't50 sparse' cases show analogous properties, with the only difference that the 'breaking transition' happens a bit later, at a threshold of 2 instead of 1.5 as in the 't50' normal case. The remarkable factor here is that the sparsity constraint helps the network reaching a highest resilience (comparable to a network trained in double the normal time).

4.4 Conclusions

This work is a first step to achieve the comprehension of the relation between an artificial neural network's topological (structural) properties and its functionality. Notably, from this first study many questions have risen: *does the weight distributions depend on the type of input data?* We suspect that the overall negative distribution for the weights in the first RBM could become less negative if the DBN were to be trained with images that don't show such a high contrast and such definite lines as in the digits case. We are talking about, for example, natural landscapes images, in which the texture is more mixed. Another interesting question is: *what if we change the initial distribution of the weights?* For example, a further step in the analysis of these networks could be analyzing a DBN trained whose weights are initialized at $t = 0$ with values sampled from a weight distribution taken from an already trained case. *Would this help the training or would it hinder its quality and its efficiency?* Suspicion falls on the second hypothesis, since a 'less neutral' initialization might prove hard to smooth and reshape during the training. Some questions arise from the strength distribution analysis, most of them in common with the open questions we asked in the former analysis section. *Would the 'zero' layer (visible layer) distribution be much more different if the DBN were trained with images of different kind? Would the distribution differ if we initialized the weights in a different manner? What could be the purpose of those highly negative strength valued neurons that arise in the sparse distribution?* And a more general purpose question we have already touched on: *how do training results depend on the type of the training input?*

Appendix A

```
%%%%%%%%% INITIALIZE WEIGHTS AND BIASES %%%%%%%%%
numhid   = DN.layersize(layer);
[numcases numdims numbatches] = size(data_GPU);
numcases_GPU      = gpuArray(numcases);
vishid_GPU        = gpuArray(0.1*randn(numdims, numhid, 'single'));
hidbiases_GPU     = gpuArray(zeros(1,numhid, 'single'));
visbiases_GPU     = gpuArray(zeros(1,numdims, 'single'));
%%%%%%%%% GET ALL DATA %%%%%%%%%
if layer == 1     data_GPU = gpuArray(single(batchdata));
else    data_GPU  = batchposhidprobs;
%%%%%%%%% FOR EACH BATCH... %%%%%%%%%
for mb = 1:numbatches
    data_mb = data_GPU(:, :, mb);
    %%%%%%%%% TRAIN AN RBM WITH CD-1 %%%%%%%%%
    initialmomentum = 0.5;
    finalmomentum   = 0.9;
    momentum_GPU    = gpuArray(initialmomentum);
    %%%%%%%%% POSITIVE PHASE %%%%%%%%%
    poshidprobs_GPU = 1./(1 + exp(-data_mb * vishid_GPU ...
              - repmat(hidbiases_GPU, numcases, 1)));
    posprods_GPU    = data_mb' * poshidprobs_GPU;
    poshidact_GPU   = sum(poshidprobs_GPU);
    posvisact_GPU   = sum(data_mb);
    poshidstates_GPU = poshidprobs_GPU > ...
              rand(numcases, numhid);
    %%%%%%%%% NEGATIVE PHASE %%%%%%%%%
    negdata_GPU = 1./(1 + exp(-poshidstates_GPU*vishid_GPU'...
              - repmat(visbiases_GPU, numcases, 1)));
    neghidprobs_GPU = 1./(1 + exp(-negdata_GPU*vishid_GPU...
              - repmat(hidbiases_GPU, numcases, 1)));
    negprods_GPU    = negdata_GPU' * neghidprobs_GPU;
    neghidact_GPU   = sum(neghidprobs_GPU);
    negvisact_GPU   = sum(negdata_GPU);
    %%%%%%%%% GET ERROR %%%%%%%%%
    err = gather(sqrt(sum(sum((data_mb - negdata_GPU).^2))));
    if epoch > 5,
        momentum_GPU = gpuArray(finalmomentum);
    end
    %%%%%%%%% UPDATE WEIGHTS, BIASES, AND ERROR %%%%%%%%%
    vishidinc_GPU  = momentum_GPU * vishidinc_GPU  + epsilonw_GPU *...
              ( (posprods_GPU-negprods_GPU)/numcases_GPU -...
              weightcost_GPU * vishid_GPU);
    visbiasinc_GPU = momentum_GPU * visbiasinc_GPU +...
        (epsilonvb_GPU/numcases_GPU)*(posvisact_GPU-negvisact_GPU);
    hidbiasinc_GPU = momentum_GPU * hidbiasinc_GPU +...
        (epsilonhb_GPU/numcases_GPU)*(poshidact_GPU-neghidact_GPU);
    vishid_GPU     = vishid_GPU + vishidinc_GPU;
    visbiases_GPU  = visbiases_GPU + visbiasinc_GPU;
    hidbiases_GPU  = hidbiases_GPU + hidbiasinc_GPU;
    errsum = errsum + err;
```

Bibliography

[1] David H. Ackley and Terrence J. Sejnowski. "A Learning Algorithm for Boltzmann Machines". In: *Cognitive Science* 9 (1985), pp. 147–169.

[2] I. Aleksander and H. Morton. *An introduction to neural computing*. 2nd edition.

[3] Bill J. Nessler B. Buesing L. and W. Maass. "Neural dynamics as sampling: A model for stochastic computation in recurrent networks of spiking neurons." In: *PLoS Comput. Biol.* 7 ().

[4] John Duchi. *CS 229 Machine Learning Course Materials*.

[5] BS Everitt. *An Introduction to Latent Variables Models*. Ed. by Chapman & Hall. 1984.

[6] Berkes P. Orbán G. Fiser J. and M. Lengyel. "Statistically optimal perception and learning: from behavior to neural representations." In: *Trends Cogn. Sci.* 14 (2010), 119–30.

[7] K. J. Friston. "The free-energy principle: a unified brain theory?" In: *Nat. Rev. Neurosci.* 11 (2010), pp. 127–38.

[8] Hinton and Ghahramani. "Generative models for discovering sparse distributed representations". In: *Phil. Trans. R. Soc. Lond.* 352 (1997), pp. 1177–1190.

[9] Geoffrey E. Hinton. "Learning multiple layers of representation". In: *Trends in Cognitive Science* 11.10 (2007).

[10] Burges LeCun Cortes. *THE MNIST DATABASE of handwritten digits*.

[11] Yoshua Bengio LeCun Yann and Geoffrey Hinton. "Deep learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[12] Lee and Ng Ekanadham. "Sparse deep belief net models for visual area V2". In: *Adv. Neural Inform. Process. Syst.* (2008).

[13] David J.C. MacKay. *Information Theory, Inference, and Learning Algorithms*. Ed. by Cambridge University Press. 4th edition.

[14] M.E.J. Newman. "The Structure and Function of Complex Networks". In: *SIAM review* 45.2 (May 2003), pp. 167–256.

[15] Andrew Y. Ng and Michael I. Jordan. "On Discriminative vs. Generative classifiers: A comparison of logis- tic regression and naive Bayes". In: ().

[16] Lambert Spaanenburg Stefan Neuber Jos Nijhuis. "Developments in autonomous vehicle navigation". In: ().

[17] Testolin and Zorzi Stoianov De Filippo De Grazia. "Deep Unsupervised learning on a desktop PC: a primer for cognitive scientists". In: *frontiers in Psychology* 4.515 (May 2013). Ed. by Swansea Christoph T. Weidemann.

[18] Zorzi and Stoianov Testolin. "Modeling Language and Cognition with deep unsupervised learning: a tutorial overview". In: *frontiers in Psychology* 4.515 (2013). Ed. by Julien Mayor.