Doctoral Dissertation
Doctoral Program in Physics (31stcycle)

From Statistical Physics to Algorithms in Deep Neural Systems

By

Enzo Tartaglione
******

Supervisor:
Prof. Riccardo Zecchina, Supervisor

Politecnico di Torino
2019

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Enzo Tartaglione
2019

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my parents

Acknowledgements

The first person I would like to thank is Riccardo Zecchina. I will never forget the first meeting we had in which he was warning me about the complexity and the deep problems I were about to face from the very beginning of my PhD: that speech gave me the motivation to go in depth in the analysis of ANN problems, motivation which was always fed by the extremely interesting talks we had as a group and typically led by your ideas.
I would like to thank Carlo Baldassi for the discussions on more "empirical" approaches and not strictly bound to the theory, Carlo Lucibello and Luca Saglietti for the hard work done together on the stochastic models. A very special thank to Federica Gerace, which suffered in silence while patiently listened to my daily crazy ideas on new learning algorithms.
I want to thank the research group at Joint Open Lab by Telecom Italia: Attilio Fiandrotti for giving me motivation and challenge, especially while writing our NIPS paper, Gianluca Francini for the complete support provided during my entire PhD, and a special thank goes to Skjalg Lepsøy, for the academic and especially human support provided.
A thank goes to my friends from outside the work: Sara D. for supporting me in my private life during critical moments, Stefano Z. for being a real friend even with some (many) defects, Carlo S. for the evenings at the pub and for the always interesting talks, Flavio R., Nicolo N, Luca D. and Stefano V. for hosting me and spending spare time together with me. A big thank goes also to all the wonderful people I met at Collegio Einaudi in the last 9 years and to all the fellow referees from A.I.A. Collegno.
Finally, the very final special thank goes to my parents: always present in my life, always supporting me despite my choices might be against their will, I have been able to go so far only thanks to you.

Abstract

Nowadays it is pretty common for people to talk about Artificial Intelligence. Contrarily from what it is imaginable, AI research is not focused on the realization of "self-thinking" machines: instead, it involves the study of any algorithm, machine or more general, artificial agent being able to perceive the environment and to react in a "smart" way, i.e. maximizing the chance of successfully achieving its own goals (like, for example, correctly classifying an object). In particular, great interest is shown towards Artificial Neural Networks (ANNs) modeling, which are biological brain-inspired. The research around ANNs started about 80 years ago, but they received huge consideration by the research community just in the last few decades. Empirically, their learning capability for non-trivial tasks has been acknowledged since 1990; however there was the lack of powerful simulation tools. In the last years, particularly thanks to the use of Graphical Processing Units (GPUs) to exploit most of the computation (and recently the use of TPUs), almost everyone owns a device able to simulate an ANN. For this reason, in the last few years artificial neural networks were challenged to solve more and more complex tasks, being able, for example, to correctly classify images in 1000 classes, to win over the world champion of Go (with the machine named AlphaGo), to understand the human speech etc. In order to accomplish all of these tasks, their size and complexity scaled-up: nowadays it is common to train ANNs having more than hundred millions of parameters. As long as the learning techniques are able to "somewhat" train the network, the community is in general not very interested in understanding all the learning dynamics inside the network.

I lived my PhD work like a journey, starting from the simplest possible model, the perceptron. Such an ANN architecture shows similarities to the Ising model; hence, analytical tools from the mechanical statistics can be borrowed to analyze the version (solution) space of the problem aimed to be solved. From the theory, we know that local research tools are typically destined to fail in the learning dynamics for the binary perceptron (a model with binary parameters). However, recently it has been shown the existence of dense sub-dominant clusters of solutions for the learning problem. Exploiting this property of the version space, a model to solve the binary perceptron problem has been designed, in which the synaptic couplings (the parameters to be learned) are considered being stochastic, according to a given distribution. It was observed that this learning dynamics, even though relying on a local-research algorithm, still finds solutions lying in the dense cluster. Such a model was extended to more complex ANN architectures. Sadly, without some extra heuristics borrowed from the common knowledge of deep learning, it seems not to achieve state-of-the-art performances and it is still matter of studies. The analysis of the version space for the perceptron model showed some geometrical characteristics, the same for more complex architectures is however not duable because of the complexity of the network itself. For this reason, an empirical exploration algorithm has been designed, aiming to investigate the properties of the version space for the Tree Committee Machine (TCM). Even though from the theory the version space should not be connected in the general case, it was empirically observed that the algorithmically-accessible version subspace is connected. Other works showed the same behavior for more complex architectures: this could be a hint for understanding why the current learning algorithms still work on larger networks. Finally, the problem of parameter reduction for deep networks (deep refers to the great number of layers) has been explored: do we really need huge architectures to solve the learning tasks? We have showed that a massive parameter reduction without performance loss is possible, designing a proper regularization term which allows us to prune parameters from the network which are not relevant to solve the learning problem. This is done by designing an additional regularization term, which is a biologically-inspired penalty for non-relevant synaptic couplings in our model.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

« What Are We? Where Do We Come From? Where Are We Going?»

Being able to provide an answer to each of these questions represents an important size of the research brought by the human kind. In particular, let us focus on the first question. Giving an answer to it is definitely not an easy task: certainly, one of the most fascinating objects of study is the brain.
Such an organ allows us to see, hear, *learn*, to elaborate thoughts... It certainly is a center which allows us to be whatever we are. Several studies have been conducted on the brain in order to understand which are the basic mechanisms allowing it to work and from these, some models have been deducted.
The essential building block of the brain is a specialized cell known as *neuron*. The neuron (referred, now on, as *biological neuron*) is a cell which receives electrical stimuli (typically) from neighborhood cells, processes them and, eventually, produces itself an electrical spike to be propagated to other neurons.
In order to investigate how the idea of the existence took place, we need to step-back in time, to about 200 years ago, when the *cell* was discovered in living creatures. Although the cell was discovered by Robert Hooke in 1665 by the observation of cork fragments, we need to wait until mid of the 19th century, when Matthias Jacob Schleiden [1] and Theodor Schwann [2] gave final and concrete proofs of the existence of cells for vegetables and animals, supported by higher-quality microscopes. At that age, it was broadly believed that the structure of the nervous system might have a different structure and organization.
However, in 1837, Jan Evangelista Purkyně, one of the fathers of anatomy, observed

the first biological neuron cells while studying the nervous tissue, and identified them in the cerebellum. Those are a particular class of neurons: they are responsible for coordination, and they take name from their discoverer (Purkyně neurons).
After some years, in 1860, a more accurate description of the neurons was provided by Otto Friedrich Karl Deiters: he was able to identify for the very first time *axons* and its *dendrites*, which are the basic elements to connect neurons [3]. The name provided to these components was different at the age: they were called respectively axis cylinders and protoplasmatic processes, as their function was merely speculated.
Along with this, there was a huge debate about the structure of the nervous system:

- **reticularists** believed that the nervous system (and, for instance, the brain) was essentially an ensemble of connections born by stretched neurons
- **neuronists**, on the contrary, were more statuary on the concept of neuron as clearly identifiable cell in the nervous system

As a result of a famous experiment, Joseph von Gerlach, in 1871, attempted to enforce reticularists' thesis, postulating that the entire nervous system was essentially a unique, continuous network, what he called "protoplasmatic network", or reticulum.
However, just two years later, the italian nobel prize winner Camillo Golgi, using the revolutionary technique named *la reazione nera* [4], was able to make a distinction between axons and dendrites, introducing a new distinction between neurons, based on the length of their connections.

« I spend long hours at the microscope. I am delighted that I have found a new reaction to demonstrate [...] the structure of the interstitial stroma of the cerebral cortex. I let the silver nitrate react with pieces of brain hardened in potassium dichromate. I have obtained magnificent results and hope to do even better in the future.»

Anyway, Golgi was a reticularist and, even though his new technique will led to the real structure of the neural system, he still believed that the real essence of the nervous system relied in the connection network.
The first signals regarding the wrongness of the reticularists thesis came very soon: in 1877 Edward Schäfer, while studying the jellyfishs, observed that they did not have connections between nerve cells and, in 1887, Fridtjof Nansen observed the same in more aquatic creatures [5]. These observations led to the study of the growth

of the nervous system: Wilhelm His, in the last decade of the 19th century, observed that the growth of the nervous system was essentially consistent to the cell theory, but the final, definitive proof was provided by the nobel-prize winner Santiago Ramón y Cajal. He used a modified version of la reazione nera to visualize the cells composing the nervous system:

« The special character of these cells is the striking arrangement of their nerve filament (axon), which arises from the cell body but also very often from any thick, protoplasmic expansion (dendrite).»

At this point, in 1891, the term *neuron* was used for the first time by Heinrich Wilhelm Gottfried von Waldeyer-Hartz, definitively asserting the correctness of the neuronists thesis.
Nowadays, thanks to the advent of the electronic microscopy, research still continues in the biological field: it was discovered that electrochemical signals were propagated along the nervous system, neurons were classified according to their specific function etc. Still, neurons in the brain are extremely complex to study and is currently an open task for biomedicine.
In the end, we can sum-up the most significant fondants of the *neuron doctrine*:

- The fundamental element of the nervous system is the neuron.
- Neurons are discrete entities.
- The parts a neuron is made-of are: cell body, dendrites and axons.
- Electric pulses travel from a neuron to another, from dendrites to the axon, stepping through the cell body.

1.1 From biological neurons to their models

Once the neuron has been identified as the basic element of the entire nervous system, a significant part of the community began proposing mathematical models of them. The so-called *biological neuron models*, also referred-to as *spiking neuron models*, attempted to comprehensively describe the properties of a part of the cells in the

nervous system. These, in particular, under certain conditions, emit electrical pulses, which are, then, propagated through the axons and received by neighbor neurons via dendrites.
Spiking neurons are known to be a major signaling unit of the nervous system: deeply understanding their behavior is the first step towards a more comprehensive understanding of the nervous system working. Hence, characterizing them is of great importance. It is worth noting that not all the cells of the nervous system produce the same type of signal characterizing the spiking neuron models: for instance, neural cells like cochlear hair cells, retinal receptor cells, and retinal bipolar cells do not spike and they have a different behavior. Furthermore, some cells lying in the nervous system are not even classified as neurons, but they are named as *glia*. Hence, describing how the nervous system works is far from being simple and still, a work in progress.

Talking about biological neuron models, their final goal is to accurate describe all the electrochemical, biological processes happening in a spiking neuron. Allowing scientists to simulate biologically-compatible models, it is possible to have a deeper understanding on what happens to the nervous system under most of the possible events. Through the presented work, we are not going to talk about these models: instead, we are going to talk about a more abstract model: the so-called *formal neuron*.

The formal neuron is a simplified version of the biological neuron. It does not emit spikes: on the contrary, an output variable describes its state. Its very first formulation appeared far in the 1943, under the so-called MCCulloch-Pitts neuron model [6]. That moment was historical: it represented the birth of a new research wing, which is still ongoing and represents one of the currently most-important fields in the human progress, known as research on *artificial neural networks* (ANN). In their model, McCullogh and Pitts considered neurons as bi-stable linear threshold elements for which just two states were acceptable: *active* or *passive*. Here, the neuron accumulates signals coming from the environment, which are weighted according to the *synaptic couplings*: according to the *post-synaptic potential*, the state of the neuron is decided.
Of course, such a model is hugely far form the biology behind the neuron and a lot of objections were raised against this model. However, those were the gold ages for the digital electronics (the Z3, the very first digital computer, was completed in

1941, the BJT was invented in 1947), and modeling a neuron as a bi-stable element appeared to follow the technological progresses of the age.
The first simulation of artificial neural networks were conducted few years later, in the 1950s, by a research team leaded by Nathaniel Rochester. He is nowadays acknowledged as one of the senators in informatics history, as he projected the IBM 701, the first commercial computer in history, and coded the very first symbolic assembly language. This and other concurrent works on artificial neural networks were presented in 1956 at the Dartomuth workshop, which is one of the oldest scientific events having the artificial intelligence as main theme.
Concurrently, along with theoretical modeling, also some immediate applications to these model were spotted by John von Neumann: he suggested that neuron functions could be replicated using electrical devices available at the ages, like vacuum tubes (the natural predecessors of transistors). Along this idea, Frank Rosenblatt, a psychologist, was the main contributor for the realization of *Mark 1*, the first electrical device which was able to learn. Such a device implemented the so-called *perceptron* algorithm, developed in 1957 at the Cornell Aeronautical Laboratory. It is among the earliest learning algorithms for neural networks. His mentor, Marvin Lee Minsky, already pioneered the design of electrical ANNs (for example, he designed SNARC, the first learning machine, in 1951), but the perceptron received huge resonance by the community:

« Dr. Rosenblatt defined the perceptron as the first non-biological object which will achieve an organization o its external environment in a meaningful way. It interacts with its environment, forming concepts that have not been made ready for it by a human agent. [...] It can tell the difference between a cat and a dog [...] Right now it is of no practical use, Dr. Rosenblatt conceded, but he said that one day it might be useful to send one into outer space to take in impressions for us.»

The New Yorker, 6 December 1958

However, beyond all this optimism, Minsky and Papert published a book in 1969 [7], in which all the strengths and, sadly, weaknesses, have been unveiled. One of the relevant weakness involves what the history then recalls it as *the XOR affair*. It was proved inside the book that a single artificial neuron, in spite of the huge potential learning capability many contemporary researchers praised on, is incapable of learning simple logic functions and, in particular, the XOR. Minsky and Papert speculated

that this was an intrinsic limitation for even larger ANN models, contributing to the so-called *first AI winter*. Nowadays, however, we know it was a wrong claim: some works, in fact, proved that XOR function is learnable by more complex artificial neural networks.
We need to jump to 1982, when John Hopfield designed a new type of neural network, named *recurrent neural network*, to turn back on the interest of the community towards ANNs. However, the new research boom happened in 1986, when Rumenhalt, Hinton and Williams showed that an automatic differentiation technique, known as *back-propagation*, could be effectively used to learn internal representations in ANNs [8]. Back-propagation is a technique which was designed in 1970 by Seppo Linnainmaa, even though at the age it was still no named. Such a self-differentiation technique, easy to use and potentially scalable, is nowadays the fundamental base for training ANNs.
From that moment on, thanks to the hugely-increasing computational capability of machines (Moore's law, GPUs [9]), and thanks to the effectiveness of back-propagation, it is nowadays possibile to simulate larger and larger ANNs. It is not weird, for example, to train a model made of $1 \cdot 10^8$ plastic connections.

1.2 The importance of artificial neural networks

Nowadays, *machine learning* (ML) and ANNs are commonly thought to be the same thing. This is not entirely true: while machine learning refers to learning some parameters for a given model in general (hence, not typically structured as an ANN), artificial neural networks have a particular structure, high-level identifiable as neurons connected together according to a given topology. ANNs are just a subset of ML: genetic algorithms, decition tree learning and sparse dictionary learning are just few example of what else ML includes. We are going to focus on ANN models and their wide applications to the real world.
In general, it is relatively easy to write a code identifying the image of an object, for example a dog. What comes tricky is to make the processing machine to *generalize* the concept of dog itself. Let us take, for example, the three possible inputs for an ANN in Fig. 1.1. Once a program becomes able to recognize the first dog (Fig. 1.1a), if you are going to ask the same code to recognize the second (Fig. 1.1b), in general there could be the following outcomes:

Fig. 1.1 These images represent a possible input to be fed to an ANN. While Fig. 1.1a and Fig. 1.1b belong to the same class (dog), Fig. 1.1c is in a different class (cat). A possible task for an ANN, for example, could be to detect whether an image represents a dog or not

- **This is not a dog**: when this happens, then the algorithm was not able to properly generalize the concept of "dog" as it will acknowledge as dog only the first image. Such a problem is one of the most known in ANNs and is named *overfit* [10].
- **This is a dog**: which is the correct answer to the problem.
  Hence, we proceed presenting as input the image of a non-dog subject, for example, a cat (Fig. 1.1c):
  - **This is not a dog**: the network is properly trained detecting the proper characteristics a dog has and is able to state whether an image contains a dog or not, this is the target scenario.
  - **This is a dog**: evidently something in the training has not properly worked as a non-dog object has been identified as dog.

According to the field of application, the topology and the training procedure, in general we acknowledge three macro-categories of learning:

- *supervised*: the task is to learn a function which maps an input to a desired output from already labeled data.
- *unsupervised*: here the ANN learns from unlabeled, unclassified and uncategorized data. The essence is here to find similarities in the data and then properly associate new data analyzing common features to already-learned data.
- *reinforcement*: here the ANN learns by taking a given action. According to a feedback reward signal, it is able to understand whether the action taken is correct or wrong, and learns by reinforcing correct actions.

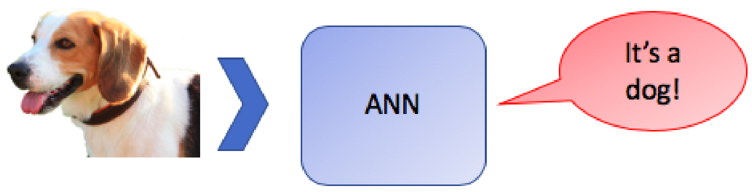


Fig. 1.2 High-level view of how an ANN works for a classification task. For a given input (here, a dog), it provides a response.

In the following, we will be focusing on the supervised learning case only.

1.3 Supervised learning

When we are performing a supervised learning task, our aim is in general to infer a function from some labeled data which provides a desired output. Typical supervised learning algorithms analyze data and attempt to infer a function whose final goal is to correctly reproduce data in the training set and accurately predict new ones. The ability of correctly predict unseen data, as we have already seen in Sec. 1.2, is called *generalization*. The dataset to be learned is here organized in pairs: each set of inputs $\xi^\mu = (\xi_1^\mu, \xi_2^\mu, ..., \xi_{N-1}^\mu, \xi_N^\mu)$ we call *pattern*, is associated to a desired output $\sigma^\mu = (\sigma_1, \sigma_2, ..., \sigma_{C-1}, \sigma_C)$ and the dataset Ξ consists in M (ξ^μ, σ^μ) pairs. A high-level picture is shown in Fig. 1.2. In that case, the input pattern ξ^μ is given by an RGB image in $\mathbb{R}^{N_1 \times N_2 \times 3}$ (where 3 refers to the color channels) and the output y^μ can be a binary classifier (dog/non-dog) in $\mathbb{R}^1$ or, as we will see, it can be a *multiclass classifier* with $y^\mu \in \mathbb{R}^C$.
In general, some steps have to be followed to solve a supervised learning task:

1. Understand the problem to be solved. A wide variety of supervised learning problems can be investigated, and each of them is subject to some restrictions (parameter constraint, model selection, required output).

2. Get the dataset. If a generic problem is asked to be solved, the proper dataset best-fitting the problem has to be chosen. For example, if the aim is to implement an OCR (optical character recognition), datasets involving written characters have to be chosen. Even here, if the main goal is to scan book prints,

datasets like EMNIST (extended-MNIST), which is a hand-written characters dataset, is not the right fit for the problem.

3. Understand the shape, the content and the distribution of the dataset. According to the problem to be solved, the input can be an image, a sound, a text, a sequence of numbers etc. and a proper learning model has to be chosen. Furthermore, there might be some issues with the dataset: it can be unbalanced (for example, if the dataset is on cancer screening, the positive cases might be lower than negative ones), data un-normalized or biased, input data might be a sparse representation, making the learning problem harder to be solved. In general, some data pre-processing might be required.

4. Choose the most appropriate model to train. There is a wide variety of supervised learning models: SVM (support vector machines), decision trees, k-nearest neighbors and artificial neural networks are just a few examples. In this work, we will be focused on ANNs which are the best for image classification.

5. Train the model. This phase is further divided in:

   (a) Divide the dataset in training set, validation set (optionally) and test set.

   (b) Choose the objective function to be minimized.

   (c) Choose a learning optimizer.

   (d) Update the parameters of the model until a given stop condition arises.

6. Evaluate the final performance.

Any ANN model here on to be explored will be structured in *layers*, as shown in Figure 1.3. In order to generate the desired output, the signal is propagated from the input to the output, in a process which is called *forward propagation*. We will see that such a structure is fundamental in order to train these models.

1.3.1 Dataset

Once the model and the dataset have been chosen, we need to train it. As we have observed in Sec. 1.2, a critical task we aim to address along with the learning process is to make our model "properly" learning the dataset, avoiding data overfit and

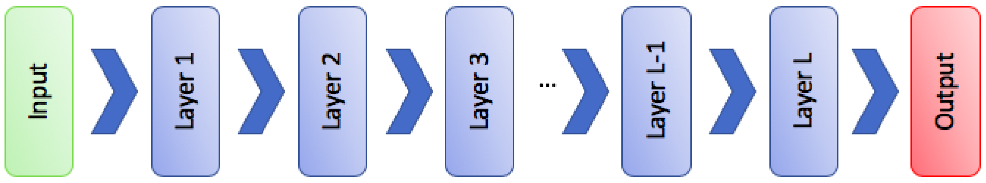


Fig. 1.3 Structure of feedforward neural networks. Here we have L layers, with L that might be a large value (in that case, we might talk about deep learning).

boosting as much as possible the generalization. What happens if we use the whole dataset for the training? Simply, we are not able to evaluate its generalization or, in other words, the performance of the model.
Typically, three datasets are used while training a model:

- *training set* Ξ_{train}: in the early stages of learning, the model is fitted on this partition of the dataset, using any parameter update strategy. Given a known set of inputs ξ^{μ}, the model produces an output y^{μ} which is compared to the desired output σ^{μ} and the internal parameters $\mathbf{W}$ of our model are (eventually) changed such that $y^{\mu} \rightarrow \sigma^{\mu}$.

- *validation set* Ξ_{val}: typical model learning strategies make use of hyperparameters, whose optimal value is generally a-priori unknown. In order to tune these, it can be useful to have some data, still following the same probability distribution of the training set. The use of the validation set, referred also as *development set* (or dev set), leads to the design of sophisticated learning strategies, like *early-stopping* and *parameter decay*.

- *test set* Ξ_{test}: the final performance of the model is evaluated on the test set, whose information has never been used for the training process: this is the reason for which this is also named *holdout set*. The main goal for supervised learning aiming to boost the generalization is to make the lowest possible error on this partition of the dataset.

The most used dataset already comes with a default slicing: for example, the MNIST dataset, consisting in 60k images, is pre-sliced in 50k images for the training set and 10k for the test set (hence, the ratio is 5:1), or the ImageNet dataset is divided in 1.2M images for the training set, 50k images for the validation set and 100k images for the test set, with ratio 20:1:2. Typically, both the validation and the test set

are smaller than the training set, but this may change according to the aim and the application for the given supervised learning problem.
Now on, for sake of simplicity, we will refer to the training set as Ξ, dropping the subscript train.

1.3.2 Learning the dataset

Once the learning problem has been properly investigated and the proper training set is chosen, here comes the model selection problem. In literature, there is a broad variety of supervised models which can be used for solving a particular problem, each with its strengths and weaknesses. However, in order to select a model, some potential issues can be observed and used to help the designer in the choice of the model.

The bias-variance dilemma

This is probably the most significant issue we should take into account when facing a supervised learning problem. If we do not take into severe consideration this possible issue, we will not be able to properly boost the generalization for our predictive model.
A first example providing a general overview of this problem was shown in Sec. 1.2: learning that a given image represents a dog does not mean that our predictor is able to properly understand which are the relevant features which represent the class "dog". The causes for a high error on the test set may be two:

- *bias*. Because of some wrong assumptions taken for the model, it is not able to learn some meaningful relations between input features, leading to high test error.
- *variance*. Our model becomes extremely sensitive to small fluctuations of the input, over-fitting the training set and worsening the performance on the test set.

A graphical example for the bias-variance trade-off is represented in Figure 1.4. Models with high variance are allowed to learn more complex predictive models but they are extremely prone to over-fitting the data; on the other hand, low variance

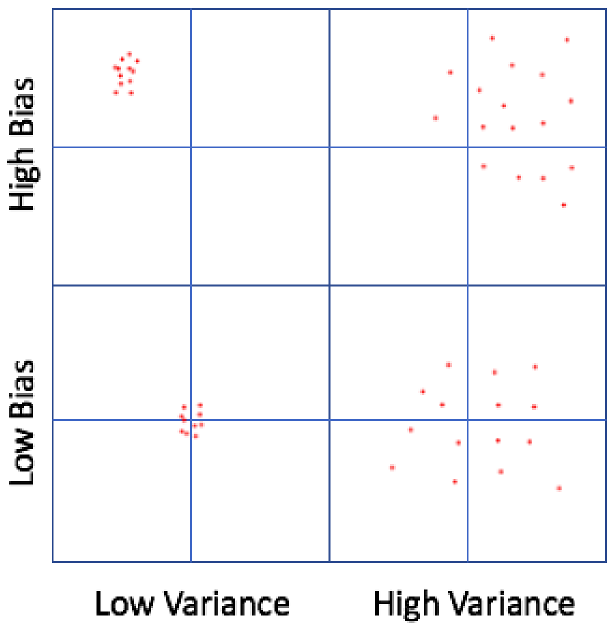


Fig. 1.4 Example of different bias-variance trade-offs in data.

models are simpler but are not capable of getting all the correlations between the input features, meaning in a poorer performance of the test set because of under-fit. Properly finding a good compromise between high and low variance models is currently a hot topic, and some approaches may be used to overcome this issue:

- a proper dimensionality choice for the internal data representation (or, in other words, the number of internal parameters) is an approach which can be used to tune bias and variance, through the use of a validation set.
- introduce some extra constraint for the learning rule, like regularization , helps in preventing data over-fit for high-variance models.
- use of *mixture models*, in which an ensemble of models is trained, each of which is specialized for a particular task, and is selected when needed. In this case, low variance models can be efficiently used for learning a simple task with no generalization loss. The trade-off, in this case, is the largeness of the final model.

Input space dimensionality

It might happen that for a given problem, each pattern is made of a huge collection of data (i.e. N very large). We might be used to the knowledge of “the more the better”, however this might make the learning process very hard to have a good generalization. Having more input features means making the model harder to recognize the most relevant ones. Let us assume we have an image of a dog and we surrond it with extra pixels, containing information from a very different problem, or even white noise. The algorithm must be guaranteed to be robust to it and to focus its attention on the slice of the image strictly containing the dog. We will see that, even if all the input features contain relevant information for the problem to be solved, we have to fight with the *curse of dimensionality* and some dimensionality-reduction strategies have to been adopted. Model complexity vs training data. If the function to be learned is expected to be simple, then we are able to force our model to be high-bias and low-variance, with a very small dataset. However, if we expect from the input model to have very complex interactions between the input features, we can not force high bias as we could easily incur in overfitting the data, and we have to relax the variance for the output, working with a larger dataset. An example of this can be found in the comparison between the MNIST dataset (60k images, divided in 10 classes of handwritten digits, each pattern of size 784) and the ImageNet dataset (1.4M RGB images, divided in 1k classes, each pattern of size 150k).

In general, it is hard to write a working program which specializes on the detection of *features* determining whether an image belongs to a certain class or not. The huge advantage of ANNs is that they are able to self-focus on the relevant features of an image to identify it to the target class, without low-level programming. Furthermore, a type of them have the capability of being *space-invariants*: if the image is taken from different angulations, rotated or tilted, the network is still able to recognize it as dog.
The training procedure of supervised learning is very simple and typically exploits a gradient-descent (GD) conputed for an objective function J which is made of a *loss* term L and an additional *regularization* term R.
L is a function of the desired output σ^{μ} and the output provided by the network y^{μ}.

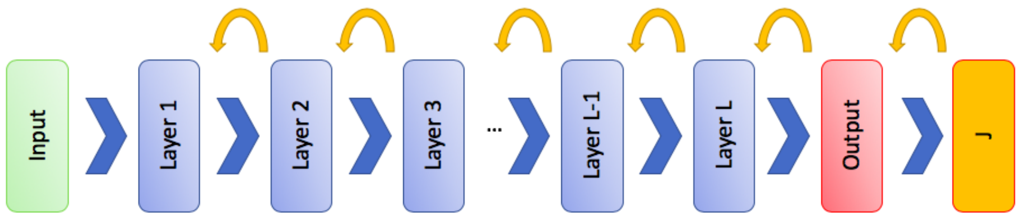


Fig. 1.5 Back propagation at work. After the forward propagation step (blue arrow), starting from J the signal is back-propagated according to the chain rule (orange arrows).

1.4 Back-propagation

Back-propagation is a technique used to train most of the currently-available ANNs to solve supervised learning problems. Thanks to it, we can theoretically train an arbitrarily deep neural network.
In order to apply back propagation, we need four elements:

- An input ξ^μ
- An ANN, made of a set of parameters **W** to be trained, and structured in layers
- A desired output σ^μ
- An objective function J, which is proportional to the error of the network's output y^μ when compared to the desired σ^μ.

Back-propagation follows some precise steps:

1. Perform the *forward propagation*: the input signal is fed to the first layer of the network and is propagated onward, until the network produces its output y^μ
2. Evaluate the loss
3. Compute the derivative $\frac{\partial J}{\partial w}$ for all the parameters in the network. This will be used in gradient-descent techniques to minimize the loss, meaning in a decrease of the network's error and, for instance, training it.

The derivative at point 3 is in general the core of back-propagation. We can compute it if we use the *chain rule*: for example, if we wish to compute the derivative for the parameter w_{ij} which is the j-th parameter in the i-th layer:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial y^{\mu}} \cdot \frac{\partial y^{\mu}}{\partial y^{\mu}_{L-1}} \cdots \frac{\partial y^{\mu}_{i+1}}{\partial y^{\mu}_{i}} \cdot \frac{\partial y^{\mu}_{i}}{\partial w_{ij}} \tag{1.1}$$

where y^{μ}_{i} is the output of the i-th layer for the μ-th pattern. Applying the chain rule is here possible because of the layered structure of ANNs; or better, the trained structure must not contain any loop.
Another significant constraint for back-propagation is that the activation function ϕ for the output of the neurons must be a differentiable function. Hyperbolic tangents and ReLU[1] are suitable for it and are the most-commonly used.
The derivatives can be computed layer-by-layer, in groups, thanks to the structure in layers, fact that is computationally-friendly and which significantly contributed in the use of this technique as base to train ANNs.

1.4.1 Objective function

The objective function J is in general the quantity we desire to minimize all along our training. It is proportional to the error of the network; for this reason, minimizing it results in an overall increment of the performance of the network on the training set. In general, the objective function to be minimized takes the form

$$J = \eta L + \lambda R \tag{1.2}$$

where L, the *loss* function, evaluates the error of the network, R, the *regularizer term*, represents an optional additive term, η and λ are two real positive scalars. Sadly, there are two intrinsic problems involving these networks: the *vanishing gradient* and the *curse of dimensionality*.

[1]ignoring the discontinuity in zero for this

Loss function

Here we can provide a wider view of the minimization of the loss function if we provide a probabilistic overview. Assume the target y^μ being distributed according to a Gaussian distribution, depending on the input ξ:

$$p(y|\xi,\mathbf{W}) = \mathscr{N}(y|\sigma,\beta^{-1}) \tag{1.3}$$

where $\mathscr{N}$ indicates a gaussian distribution and β is defined as *precision*. From eq. 1.3 we can write the log-likelihood

$$\mathscr{L} = \prod_{\mu=1}^{M} \mathscr{N}(\sigma^\mu|y^\mu,\beta^{-1}) \tag{1.4}$$

where M is the size of the training/test set. Our optimization aim is here to maximize eq. 1.4. We can write from this the negative log-likelihood (to be minimized, assuming the shape of an error function)

$$\mathscr{L} = \frac{\beta}{2}\sum_{\mu=1}^{M}(\sigma^\mu - y^\mu)^2 + \frac{M}{2}\left[\log(2\pi) - \log(\beta)\right] \tag{1.5}$$

Here the problem becomes equivalent to minimizing the term

$$L^{MSE}(\sigma^\mu, y^\mu) = \frac{1}{2}\sum_{\mu=1}^{M}(\sigma^\mu - y^\mu)^2 \tag{1.6}$$

which is the *mean squared error* function, one of the earliest choices for training ANNs. The first ANNs trained with back-propagation had $J = L^{MSE}$.
Now, let us assume we are working with a binary classification task. This implies that the target output $y^\mu \in \{0;1\}$. If we use a sigmoid activation function at the last layer of an ANN

$$\phi(x) = \frac{1}{1+e^{-x}} \tag{1.7}$$

with $\phi(x) \in (0;1)$, we can write the Bernoulli distribution

$$p(\sigma^\mu|\xi,\mathbf{W}) = (y^\mu)^{\sigma^\mu}(1-y^\mu)^{1-\sigma^\mu} \tag{1.8}$$

Taking the negative log-likelihood (to be minimized), we have the form

$$L^{CE}(y^\mu,\sigma^\mu) = -\frac{1}{M}\sum_{\mu=1}^{M}\sigma^\mu \log(y^\mu) + (1-\sigma^\mu)\log(1-y^\mu) \tag{1.9}$$

which is called *binary cross-entropy loss* function. Eq. 1.9 can be extended to multi-class classification problems (hence, problems for which the class to be distinguished are > 2). Assuming the class labels are independent, we can write

$$p(\sigma^\mu|\xi,\mathbf{W}) = \prod_{k=1}^{C}(y_k^\mu)^{\sigma_k^\mu}(1-y_k^\mu)^{1-\sigma_k^\mu} \tag{1.10}$$

where C is the number of classes in our model. Here we can take the negative log-likelihood as usual. If the target vector follows a one-hot encoding[2], we can write the so-called *multi-class cross-entropy loss*:

$$L^{M-CE}(y^\mu,\sigma^\mu) = -\frac{1}{M}\sum_{\mu=1}^{M}\sum_{c=1}^{C}\sigma_c^\mu \log(y_c^\mu) \tag{1.11}$$

where $y^\mu, \sigma^\mu \in \mathbb{R}^C$, $y_c^\mu \in \{0;1\}\forall c$ and $\sigma_c^\mu \in (0;1)\forall c$.
These are just two examples of loss functions: many more have been implemented and can be used to train an ANN; however, most of the times, multi-class cross-entropy is used.

1.5 The generalization problem

Now we have all the minimal ingredients to train an artificial neural network on classification tasks. When we are minimizing the loss function, we are training our model on the data set. Hopefully, when the gradient is zero, then the loss function will be zero as well, and the association between target output and the inferred output

[2]in this case, we have a one of K mutually-exclusive class map: if the correct class is the c-th, the target vector σ^μ entries are all 0 except for the c-th which will be 1

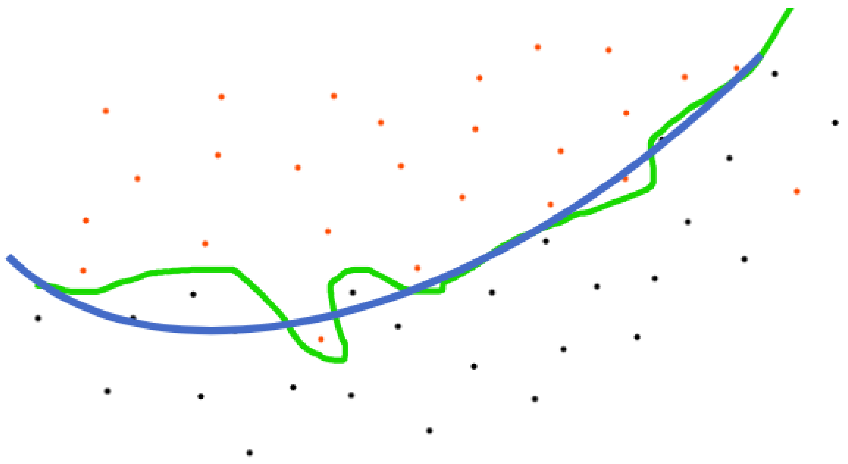


Fig. 1.6 Example of overfit (green line) vs optimal learning (blue line).

for the patterns in the training set overlaps.
However, things are not so easy. As we hinted in Sec. 1.2, we need to guarantee our model is able to correctly classify unseen data, or better, to *generalize* well. Towards this end, a number of techniques have been developed, and still lot of research around this topic is being done nowadays. In this section, we are going to analyze more in depth all the main techniques aiming to improve the generalization, defining what *regularization* is and evidencing its effects while training a model.

1.5.1 Proper data fitting

We can provide many definitions for the "overfit" problem, but essentially it means that the trained model learned too tightly the training set and it fails in classifying properly unseen data (for example, in the test set). A symmetric issue, less discussed in ANNs, is the *underfit* problem, where the training data are learned too loosely, making the performance on both training and test set poor.
As an example, let us say we wish to train our model the data on Fig. 1.6. Therefore, we have two classes, and we ask our ANN to learn the class boundary. If the data are trained too tightly (like in the green line) we are evidently over-fitting the training set. On the contrary, if we are training our model too loosely, we have underfit. A proper regularization strategy aims to make the ANN learn the black boundary. In that case, some patterns in the training set are not properly learned; however, this is beneficial for the generalization: as it is possible to observe, some noise is present

in the data. Such a noisy environment is a very typical scenario for real datasets and typically is not known a-priori, making the generalization problem even more challenging and relevant.

1.5.2 Data augmentation

One of the most used techniques aiming to improve the generalization of a deep model involves the generation of new patterns from those already in the training set. This generation process *augments* the training data, and this is why we refer to these in general as data augmentation techniques.
In the case of deterministic algorithms, this approach was introduced in a famous work of 1977 by Dempster et al. [11]. Here, data augmentation aim was mainly to substitute (or fix) incomplete (or noisy) data for maximum likelihood estimation.
A few years later, such an approach became popular as well in the statistics field: in a work by Tanner et al. [12] data augmentation was successfully used for the computation of posterior distribution. This work gave a kick-off to statistical mechanics applications: an algorithm for Montecarlo simulations of the Ising and the Potts models was proposed by Swendsen and Wang in in 1987 [13]. It introduces extra bond variables, which empirically improve the convergence of the montecarlo sampling. In such a frame, datat augmentation can be used both to improve the quality of the posterior distribution (and, for instance, the generalization) and the convergence of the training algorithms. How can we apply data augmentation to deep models?
Let us focus, for sake of simplicity, on datasets made of images, for classification tasks. For example, we ask our deep neural network to recognize that in the picture there is a dog. Of course, the main information contained in the image (i.e. the presence of the dog) does not change if we rotate the image by 90 degrees, or if we flip it, twist, tilt etc. These are just few examples of all the transformations available to augment an image dataset. Currently, all the deep learning libraries (like, for example, *torchvision*) include many data augmentation strategies.

1.5.3 Regularization term

Regularization is a process of introducing additional constraints to any problem (hence, not strictly related to machine learning or ANNs) to solve any ill-posed

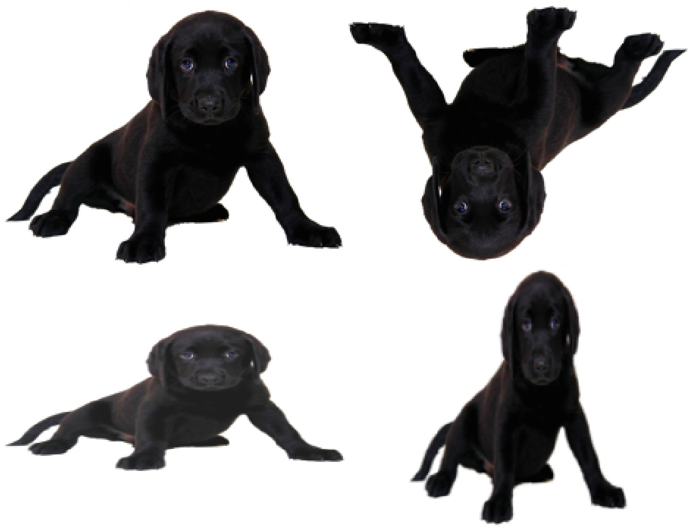


Fig. 1.7 Some examples of data augmentation: rotation, tilting, squeezing.

problem [14]. In general, the dimensionality N of the inputs is imposed by the problem (or, in our case, the dataset) while the architecture of the network (and, for instance, the number of trainable parameters) is a "free parameters" as it is decided by the trainer. As observed before, for each problem there exist an optimal architecture for a given problem, which provides the best generalization error. Sadly, having an optimal structure and size of our neural network is not a sufficient condition to have the best performance because of the existence of local minima and saddle points in the loss function [15]. Here comes the effect of the regularization function: it can be seen as a corrective factor of the loss function which helps avoiding to get stuck in these local minima.
Of course, other techniques may be used to prevent this: *stochastic gradient descent*, for example, introduces a noise which significantly helps in overcoming such a problems, as well as *dropout*. However, what the regularization function can do for us is to select a subset of "nice" minima in which the learning procedure, in the end, lands. According to different purposes, such a term may have different forms. One of the most-known is the so-called *L2 regularization*, also known as *weight-decay*

$$R(\mathbf{W}) = \frac{\lambda}{2}\sum_i w_i^2 \tag{1.12}$$

In the case of $L = 0$, this regularizer pushes all the parameters w_i to decay towards zero. The main and absolute advantage of such a regularizer is that the derivative still depends on the parameter w_i and for this reason an exact minimizer can be found in closed form.
Actually a general form of such a regularizer is

$$R(\mathbf{W}) = \frac{\lambda}{2}\sum_i |w_i|^p \tag{1.13}$$

and the case for which $p = 1$, used for obtaining sparse networks, is known as *lasso regularizer*. Indeed, lasso regularization favors solutions having less parameters in the system and is used for feature selection (this particular class of regularizers will be discussed more in details in Chapter 4).
Focusing back on weight-decay, from the Bayesian perspective, it corresponds to using a symmetric multivariate normal distribution as a weights prior:

$$p(\mathbf{W}) = N\left(\mathbf{W} \middle| 0, \frac{1}{\lambda} I\right) \tag{1.14}$$

having

$$-\log N\left(\mathbf{W} \middle| 0, \frac{1}{\lambda} I\right) \propto -\log e^{-\lambda^2 \|\mathbf{W}\|_2^2} = \lambda^2 \|\mathbf{W}\|_2^2 = R(\mathbf{W}) \tag{1.15}$$

. Currently, weight-decay is the most used regularizer for training deep architectures.

1.5.4 Optimizers

As we have previously stated, state-of-the-art training techniques in supervised learning involve gradient-descent techniques. Assuming Ξ our training set, the update step for a generic parameter of the network w_i is

$$w_i^{t+1} = w_i^t - \eta \left\langle \frac{\partial L}{\partial w_i^t} \right\rangle_{\Xi} \tag{1.16}$$

where we are assuming an optimization function in the form $J = \eta L$ for sake of simplicity, η is a real positive number known as *learning rate* and $\frac{\partial L}{\partial w_i^t}$ is computed through back-propagation (for further details see Sec. 1.4). Most of the training

sets are however very large: for example, MNIST, which is one of the smallest, contains $60k$ examples in the training set. Computing the average in eq. 1.16 can be extremely time-inefficient and we expect the proposed update, due to the averaging effect, to be very small, commonly making the entire learning process being very slow. Furthermore, automatic back-propagation engines (like the widely-used *Autograd* [16], which is the core of most of the deep learning frameworks used to perform research like pyTorch) saves all the intermediate computation results from forward-propagation, becoming also a memory-inefficient optimizer.
A widely-used variant of gradient descent is called *stochastic gradient descent* (SGD) [17] which proposes the following update step:

$$w_i^{t+1} = w_i^t - \eta \left\langle \frac{\partial L}{\partial w_i^t} \right\rangle_{\tilde{\Xi}^t} \tag{1.17}$$

where $\tilde{\Xi}^t \subset \Xi$ is a sampled portion of the entire training set. In a work by Bottou et al. [17], the size of $\tilde{\Xi}^t$ is 1; hence, one pattern at a time was used to update the entire model and this condition is called *online learning*. Each step of eq. 1.17 is called *iteration* and in each step the subset $\tilde{\Xi}$ changes in order to cover the entire training set Ξ. Once the training set has been fully covered, we say a training *epoch* has completed. Let us say we are in the middle of an epoch having performed I samplings of $\tilde{\Xi}$. We define their union being the patterns already sampled in the current epoch $\Xi_{sam} = \bigcup_{i=1}^{I} \tilde{\Xi}_i$. We now need to generate the new $\tilde{\Xi}_{i+1}$. For this reason, we need to perform a sampling from the set of non-sampled patterns $\Xi_{unsam} = \Xi \setminus \Xi_{sam}$ and for this reason such a technique is stochastic.
The overall training procedure can however take a significant amount of time to converge to a satisfactory solution: for such a reason, other more sophisticated optimizations based on eq. 1.17 have been implemented and currently widely used. The very first of these upgrades was designed by Rummelhart, Hinton and Williams in 1986 [18] and was called *momentum SGD*. Differently from plain SGD, which performs the update of the parameters just locally evaluating the state $\mathbf{W}$ of the network, momentum SGD includes a memory term v which keeps the history of the gradients computed for the previous steps. Hence, the update rule in this case becomes

$$w_i^{t+1} = w_i^t - \eta \left\langle \frac{\partial L}{\partial w_i^t} \right\rangle_{\tilde{\Xi}^t} + \mu v_i^t \tag{1.18}$$

where μ is a positive real hyperparameter and

$$v_i^t = a \cdot v_i^{t-1} + (1-a) \cdot \left\langle \frac{\partial L}{\partial w_i^{t-1}} \right\rangle_{\tilde{\Xi}^{t-1}} \tag{1.19}$$

having $a \in [0;1]$. Is was empirically observed that the use of momentum-based techniques on ANNs can significantly reduce the simulation time favoring the escape from saddle points and there are some hints suggesting it helps escaping from local minima, favoring the generalization improvement [19, 20].
A famous and widely-used variant of momentum SGD was proposed by Nesterov et al. [21] and is known as *Nesterov accelerated gradient*. It essentially uses the same information as momentum SGD, but it evaluates the gradient where the momentum v^t would push the parameters:

$$\begin{aligned}
\tilde{w}_i^t &= w_i^t + \mu v_i^{t-1} \\
v_i^t &= a \cdot v_i^{t-1} + (1-a) \cdot \left\langle \frac{\partial L}{\partial \tilde{w}_i^t} \right\rangle_{\tilde{\Xi}^t} \\
w_i^{t+1} &= w_i^t - \mu v_i^t
\end{aligned}$$

This technique is supposed to be much more accurate than momentum SGD: the gradient computed on $\mathbf{W}$ is a local measure, and evaluating it in two different (even if close) points may dramatically affect the dynamics of the optimization. State-of-the-art deep networks are typically trained using Nesterov accelerated gradient.
Finally, we would like to present another optimizer which gained huge popularity in the last few years: *Adam* [22]. Kingma et al. presented in their work Adam as the combination of two other recently-proposed optimizers:

- Adaptive Gradient Algorithm (AdaGrad), proposed by Duchi et al. in 2011 [23], suggests that each parameter should have a custom-designed learning rate (adaptive). Such an approach boosts the convergence to solution in settings where the gradient is typically a sparse quantity.

- Root Mean Square Propagation (RMSProp), proposed by Tielemans and Hinton in 2012 [24], in which the idea of a per-parameter learning rate is further exploited, but in a more general setting. In particular, the learning rate is tuned according to the magnitude of the recent gradients (hence, a quantity similar to the momentum).

Both these techniques tune a per-parameter learning rate according to some considerations to the gradients. What Adam does is not just focusing on the first moment (the mean) of the gradients as in RMSProp, but it goes further, taking into account as well the average of the second moments of the gradients (their un-centered variance):

$$
\begin{aligned}
m^{t+1} &= \beta_1 m^t + (1-\beta_1)\frac{\partial L}{\partial \tilde{w}_i^t} \\
v^{t+1} &= \beta_2 v^t + (1-\beta_2)\left(\frac{\partial L}{\partial \tilde{w}_i^t}\right)^2 \\
\hat{m} &= \frac{m^{t+1}}{1-\beta_1} \\
\hat{v} &= \frac{v^{t+1}}{1-\beta_2} \\
w^{t+1} &= w^t - \eta\frac{\hat{m}}{\sqrt{\hat{v}}+\varepsilon}
\end{aligned}
$$

where ε is a small positive quantity preventing divisions to zero and β_1, β_2 are some hyper-parameters to tune the effect of the two moments of the gradient. According to the authors and to some empirical evaluation, Adam outperforms other methods for a variety of models and datasets, while still scaling well on high-dimensional machine learning problems. Such an optimization technique is currently being used more and more often for training deep models and entered in the common knowledge for this field.

1.5.5 Dropout

ANNs (and, in particular, deep networks) are complex models, containing huge number of parameters aiming to learn complex relationships between input patterns and output. As we have already observed, however, input data are typically noisy (which can be caused by a number of factors) and one of the final goals of ANN training is to make a distinction between the real information identifying a class and noise. If such a distinction is not explored, the model is prone to over-fit the data. In order to improve the generalization, it is possible to regularize a model by averaging the predictions coming from all the possible combinations of parameters and weighting

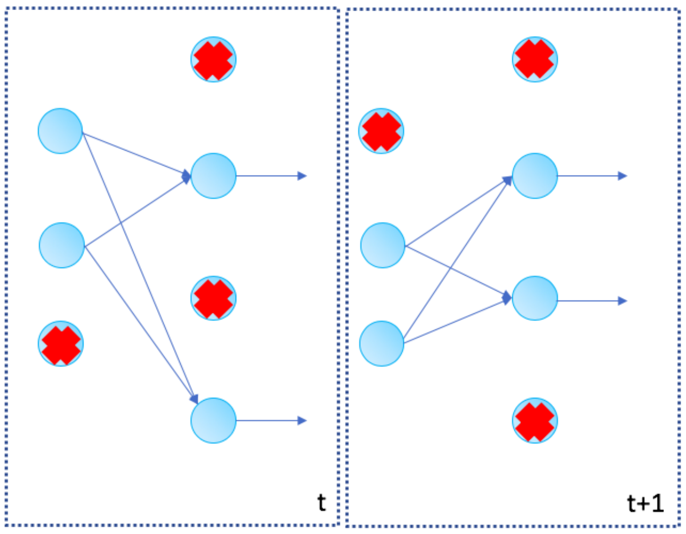


Fig. 1.8 Example of dropout. During training, stochastically some neurons are excluded from the network. At the successive training iteration, dropout probabilities are reset.

each configuration by its posterior probability for the given training set. Such an approach has been successfully explored for probabilistic matrix factorization [25] and even applied to RNA sequencing [26]; however it is computationally inefficient and scales badly with the model complexity. From this observation, Hinton et al. [27] and later Srivastava et al. [28] proposed an approximation to such a technique. *Dropout* fights over-fitting and provides a way of approximately combining an exponential number of configurations for the trained ANN in an efficient way.

"Dropout" literally refers to stochastically dropping-out entire units along the training in a neural network. Along a training iteration, when a unit is dropped, it is momentarily removed and it is completely insensitive to the inputs it receives. Furthermore, as it is temporarily removed from the network, its weights are insensitive to the update kick coming from back-propagation. The probability of dropping-out any unit is fully independent from that of other units, and resets after every training iteration. An example of how dropout works can be found in Fig. 1.8. The process of tuning such a dropout probability is a completely heuristic process and can be set using a validation set.

An interesting motivation given by the authors about dropout comes from the biology and, in particular, natural selection [29]. In particular, the ANN trained with dropout learns how to still correctly classify the training set minimizing the co-dependence of particular sub-groups of neurons, making the overall behavior of the deep model

more robust and ideally focusing the entire learning process towards the relevant features. Besides this, there could be plenty other romantic and philosophical explanations regarding what is the real effect of dropout or how such a technique is related to living beings interactions... In our context, which is the training of ANNs aiming to boost the generalization, such a work has been highly-considered and dropout is included as base training technique for a high number of deep models.
Besides the improvement in generalization, however, evidently such a technique requires longer training times, and sometimes a proper choice of the dropout probability significantly affects the effectiveness of the technique. A number of variants of the dropout technique have been proposed: one of the nowadays most used is *dropconnect* by Wan *et al.* [30], which focuses on dropping the connections between neurons in place of the neuron themselves. We are going to see in Sec. 4.1.2 that dropout can be used also to introduce sparsity in deep models with a custom technique called *variational dropout*.

1.5.6 Batch normalization

Another recently proposed technique aiming at first to boost the convergence to a solution for deep network is *batch normalization* [31]. Ioffe et al. introduced such a technique in 2015 aiming to un-bias and normalize the signal traveling between layers.
As we are working with multi-layer architectures, we can imagine that small changes (or perturbations) to the network parameters in the early layers may amplify changes in the forward-propagated signal, as the network becomes deeper and deeper. This potentially may represent a stability issue: in case each layer, for each minibatch, need to adapt to a new signal distribution, the back-propagated signal might result too strong. Of course, such an issue can be resolved by using lower learning rates, or a proper initialization of the parameters; however, this can be directly tackled using regularization techniques.
In general, at a given layer l, when the signal distribution at the input ξ of the network varies, we say it is subject to *covariate shift* [32]. Artificial neural network models typically suffer of covariate shift, especially for the signal propagated between its layers (not just between the input and the output of a layer, but also between output signals of different layers). In this case, we are going to refer to it as *internal covariate shift*. According to its authors, batch normalization aims to minimize the

internal covariate shift (hence, it regularizes the signal traveling between different layers) forcing a “homogeneity” in the forward propagated signal and, for instance, minimizing the risk of gradient explosions.
Let us say we are in an ANN model al the l-th layer, and we are forward-propagating a signal generated by the input minibatch $\tilde{\Xi}$ of size M (or, in other words, such a minibatch contains M different patterns). If we call $\mathbf{y}_l^\mu$ the vector containing all the outputs of the neurons at the l-th layer for the μ-th patter, we can say that we have, for $\tilde{\Xi}$, a mean $\mathbf{m}_{\tilde{\Xi}}$ and a variance $\mathbf{v}_{\tilde{\Xi}}$ for the signal

$$\mathbf{m}_{\tilde{\Xi}} = \langle \mathbf{y}_l^\mu \rangle_\mu$$
$$\mathbf{v}_{\tilde{\Xi}} = \left\langle \left(\mathbf{y}_l^\mu - \mathbf{m}_{\tilde{\Xi}}\right)^2 \right\rangle_\mu$$

Of course, during the forward propagation step, the signal is normalized according to

$$\hat{\mathbf{y}} = \frac{\mathbf{y} - \mathbf{m}}{\sqrt{\mathbf{v}} + \varepsilon} \tag{1.20}$$

In this way, however, we have signal with no average and unit variance. In order to allow the network to *learn* mean and variance in a more direct way, a *batch normalization* layer is here applied:

$$\mathbf{y}_{\mathbf{BN}} = \gamma \hat{\mathbf{y}} + \beta \tag{1.21}$$

where β and γ are parameters to be learned using back-propagation, and are parameters dependent on all the minibatches.
According to empirical results, inserting batch normalization allows to use higher learning rates, sometimes boosting generalization, especially in deep architectures. However, nowadays there is still a huge debate around the true effects of this technique. For example, Santurkar et al. claim that batch normalization’s main effect does not involve internal covariate shift, but rather smoothing the overall objective function, and that was the reason for which higher learning rates are allowable in batch-normalized networks [33]. Many other interpretations on the effect of batch normalization have been formulated in the last few years [34], however most of the data scientists agree on the effectiveness of such a technique and include it in many

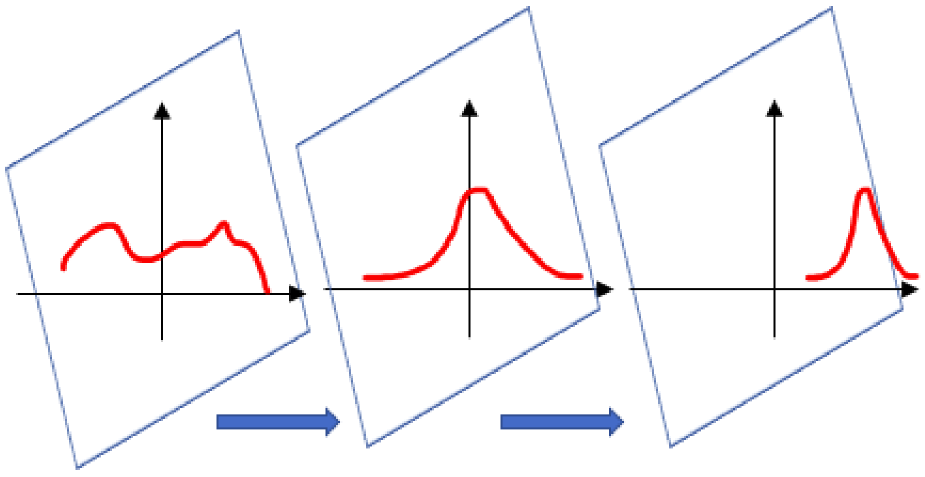


Fig. 1.9 Example of how batch-norm works. It takes first the output distribution for the layer, then it normalizes it and finally applies the learned mean and variance.

deep architectures, regardless the necessary increase in the computational complexity.

1.5.7 Vanishing gradient

The vanishing gradient problem is a difficulty found in training all the deep artificial neural networks whose learning is performed using gradient-based optimization, e.g. back-propagation. The problem is that in some cases the error signal, while it is being back-propagated, becomes extremely small, making hard and extremely inefficient the update of the weights.
Let us take a standard feed-forward model described in Sec. 1.3 trained with back-propagation (Sec. 1.4). The chain rule in eq. 1.1 allows us to minimize the error of the ANN by computing partial derivatives to all the parameters of the model, which are then updated according to a gradient descent-based optimizer (Sec. 1.5.4). However, typically the deepest the network is, the weakest the error signal back-propagated to the first layers is. We have a worst-case scenario when the gradient, due to this effect, become zero despite the loss term L is still positive. This happens because the traditionally-used activation function used is the hyperbolic tangent or the sigmoid function: they have the nice property of saturation; hence, there are fixed bounds for the signal. However, if the signal is outside the linear regime, the error signal is very

low, and it is back-propagated to the previous layers.
Let us take the case of $\phi(x) = \tanh(x)$. As we know, its derivative is $\phi' = \mathrm{sech}^2(x) \in (0;1]$ and has its maximum for $x = 0$. Now, let us assume $x \ll 0$ which is a bad guess from the network (hence, $L > 0$). The error signal, arriving from the neighboring levels given by back-propagation, according to the chain rule, is multiplied by $\mathrm{sech}^2(x) \approx 0$. For this reason, the learning process becomes extremely slow and, if x is sufficiently small, impracticable because of the numerical approximation of such a function.
In order to overcome this problem, a number of alternative activations has been proposed. In particular, the currently most-used is the ReLU activation:

$$\mathrm{ReLU}(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \tag{1.22}$$

The derivative of such a function is $\mathrm{ReLU}'(x) = \Theta(x)$ where $\Theta(x)$ is the step function. If we are using this activation function to all the network, of course we are requiring all the output of the network being positive as well as the output σ^μ to be evaluated by the loss function, and the back-propagated signal no longer suffers of the vanishing gradient problem.
There is nowadays a large number of proposed activation functions designed to solve the vanishing gradient problem. A trivial idea to solve the problem could have been to work with activations whose derivative is larger than one. However, this would lead to the opposite problem of *exploding gradient*.
Other approaches can be however used to solve this problem: we mention here, for example, *batch normalization* (see Sec. 1.5.6).

1.5.8 Curse of dimensionality

While the vanishing gradient problem involves the depth of a neural network, the *curse of dimensionality* involves the size of a single layer [35]. The highest is the size N of a layer, the highest is the dimensionality of the data we need to deal with. It is, indeed, not unusual in a neural network to have layers receiving data having hundreds or thousands of dimensions. The problem is here that increasing the dimensionality of a problem the volume of the space exponentially increases such that all the available data for solving a problem become sparse and, hence,

insufficient to cover the space.
Such a sparsity is a concrete problem because it commonly drives to *data overfit* problems, which are critical in deep learning. For this reason, some approaches are used to reduce the dimensionality of data and to properly correlate data to be classified. Probably the most known approach for image classification is the use of *convolutional layers* which process data emphasizing the desired features and eventually reducing the dimensionality of the problem.
Other effective strategies involves the use of unsupervised techniques, ranging from *principal component analysis* (PCA) to the *nearest neighbor search* to sophisticated techniques involving the use of *autoencoders*.
Taking high dimensionality data, properly find correlations between them and finally reducing the dimensionality provides high-quality training set than roughly reducing the dimensionality of data and is extremely advantageous. This process and effect goes under the name of *blessing of dimensionality* [36].

1.6 From the simple to the complex

In the very beginning, the research involving neural networks was essentially theoretical and methodological, due to the limited simulation capabilities of the machines. With the current massive presence of powerful computers, deep learning has become something almost everyone can experiment.
Furthermore, thanks to the possibility of accomplishing very complex tasks with an automatic learning procedure, this field is attracting more and more interest. Every day more complex architectures of neural networks are designed and simulated.
However, in all this great excitement, the initial approach of starting from simple models and incrementally increasing the complexity, has been forgotten.
During my PhD I aimed to have a methodological approach. In order to do this, first the simplest model, the perceptron, has been explored in Chapter 2. Its similarities to the models of statistical mechanics (the Ising model) allow a formal and grounded analysis. A step forward towards multi-layer ANNs is done in Chapter 3. There, some inspirations coming from the perceptron problem are applied to more complex models, observing their behavior and empirically analyzing the version space. Then, aiming to solve complex problems with small architecture, a sparsification technique for deep models has been designed and introduced in Chapter 4, still having in mind

some key stones learned for simpler models. Finally, in Chapter 5 the conclusions are drawn.

Chapter 2

The perceptron problem

In this chapter we are going to talk about the simplest architecture of neural network is the *perceptron*. It was proposed for the first time by Rosenblatt in 1957 while working at the Cornell Aeronautical Laboratory [37] [38]. It was first naturally thought to be a machine (hardware) rather than software and it was actually realized in the "Mark 1 perceptron" [39]. In the early 60s the whole AI community was shocked by Rosenblatt model and huge debates took place. Indeed, Rosenblatt claimed that the perceptron was *the embryo of an electronic computer that [...] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence* [40]. Now we know Rosenblatt was right, but for the community those were heavy statements and this gave the kick-off to the research around neural networks. We are going to introduce first the model in Sec. 2.1. Such a model can be investigated using tools borrowed by the statistical mechanics: in Sec. 2.2 and Sec. 2.3 two models, the perceptron with real-value parameters (continuous perceptron) and the perceptron having parameters with binary values (binary perceptron) are presented, along with state-of-the-art analysis and commonly-used learning rules. Particular emphasis is devoted to the analysis of the binary perceptron: in order to train this model, in fact, standard GD-based learning rules are in general unfeasible. In Sec. 2.4 we introduce our original stochastic perceptron. The stochastic perceptron succeeds in learning binary parameters using gradient descent. Furthermore, it is able to reach solutions lying in rare, exponentially dense solutions clusters, consequently guaranteeing robustness for the found solution.

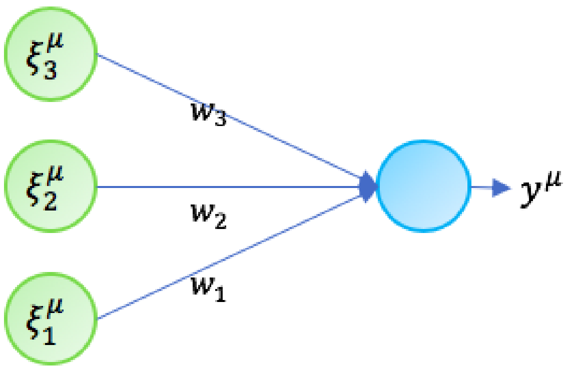


Fig. 2.1 The perceptron model. Once presented a pattern ξ^μ as input, each of its N components are re-weighted by N synaptic couplings. Then, the obtained pre-activation potential goes through a threshold function, producing the output y^μ.

2.1 The model

The model proposed by Rosenblatt was very simple. Its structure is made of two layers:

- *input layer*: of size N, it is the layer which receives the input pattern ξ.
- *output layer*: in the perceptron model is always of size 1, it collects the inputs through N *couplings* or *weights* w and generates the output y through an activation function ϕ.

Hence, the perceptron can be described by

$$y = \phi\left(b + \sum_{i=1}^{N} \xi_i w_i\right) \tag{2.1}$$

where b is named *spiking threshold* or *bias*. Of course, eq. 2.1 can be extended to a multiple set of patterns. Let us say we have M different patterns and $\mu \in [1;M]$. The μ-th set of input will ptodUce an output according to

$$y^\mu = \phi\left(b + \sum_{i=1}^{N} \xi_i^\mu w_i\right) \tag{2.2}$$

The activation function for the perceptron problem is commonly a limited (saturating) odd function. For example, binary perceptrons use the sign activation function

$$\mathrm{sign}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ +1 & x > 0 \end{cases} \tag{2.3}$$

while continuous perceptrons make use of the hyperbolic tangent or the sigmoid function. It comes natural from this that the perceptron is a binary classifier in which we aim to learn the correct class σ^μ (known as *label*) associated to ξ^μ. This is more evident if we try to identify the *decision boundary*. Imposing the condition

$$\phi\left(b + \sum_{i=1}^{N} \xi_i^\mu w_i\right) = 0 \tag{2.4}$$

and observing that ϕ is an odd function, eq. 2.4 becomes

$$b + \sum_{i=1}^{N} \xi_i^\mu w_i = 0 \tag{2.5}$$

This represents an $N-1$-dimensional hyperplane dividing the phase space in two halves. Such a model has limits. The most relevant is that it is a binary classifier, so it can discriminate between two classes only. Furthermore, it is able to learn linearly-separable classes only. As pointed out by Minsky and Papert, perceptrons are not able, for example, to learn the XOR function [7]. These limitations, however, can be overcome using more complex models of neural networks, the simplest is the *multilayer perceptron*. We will discuss this in Chapter 3.

2.1.1 Performance evaluation

As we have previously observed in Sec. 1.3, the performance of a neural network is in general evaluated through a loss function. For instance we can “count” how many of the M patterns are properly evaluated by the network by the loss function

$$\varepsilon = \frac{1}{M} \sum_{\mu=1}^{M} \Theta(-y^{\mu} \sigma^{\mu}) \tag{2.6}$$

where $\Theta(x)$ is the heaviside function

$$\Theta(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases} \tag{2.7}$$

The lower ε is, the better we are performing and when we reach $\varepsilon = 0$ we have learned all the training set. However, there is no warranty regarding the generalization of the network: depending on the learning technique, we might need more or less examples in our training set to have a good generalization. However, in general, the larger the training set is, the better the generalization as well.
Our analysis will be here divided in two different learning settings:

- the teacher-student (or generalization) scenario: here we have a neural network (student) which has to reproduce in the most accurate way a target rule $\sigma^{\mu} = \varphi(\xi^{\mu})$ which can be generated, for example, by another neural network, named *teacher*. The most relevant performance indices is here the generalization error, evaluated on a test set.

- the storage scenario: here we try to give an answer to the question: "How much a neural network is able to learn?". Such an estimation is performed feeding the network with non-correlated random input patterns and learning as many as possible.

In the next sections we are going to present the continuous and the binary perceptron problems. We will show that these models can be analyzed using techniques from the statistical physics, using which we are able to better understand some of the properties of the solution space. This knowledge motivates the design of our stochastic perceptron, in which we solve the hard binary perceptron learning problem taking all the advantages of training continuous quantities.

2.2 Continuous perceptron

In the so-called *continuous perceptron* $w_i \in \mathbb{R} \forall i$. Eq. 2.2 still holds; however, learning these parameters is easier as simple techniques like gradient-descent are applicable and it is empirically proven they are extremely effective. For these reasons, this model was object of many studies in the past.
A known constraint applied to the continuous perceptron is

$$\sum_{i=1}^{N} w_i^2 = N \tag{2.8}$$

through which the vector of weights is normalized to a sphere having radius $\sqrt{N}$. For this reason, this is known as *spherical perceptron*. We will see that such a constraint, even though it is not affecting the quality of the final result, is necessary in order to use some results from statistical mechanics and, in particular, the *Gardner analysis*.
We know that each classification problem has a set of some input patterns ξ^μ and their associated label y^μ. In such a framework, we can work assuming a sampling from the probability distributions $P(\xi^\mu)$ and $P(y^\mu)$. From here, the link to statistical physics appears. Let us think to the ws as microscopic variables which live in the phase space. Assuming the learning dynamics converges to a final equilibrium point, we can say the weights are distributed in the phase space according to the Gibbs measure

$$P(w) = \frac{1}{Z(\xi)} e^{-\beta \varepsilon(w,\xi)} \tag{2.9}$$

where $\beta = \frac{1}{T}$ and Z is the partition function which is the sum over all the possible configurations of w

$$Z(\xi) = \int d\mu(w) e^{-\beta \varepsilon(w,\xi)} \tag{2.10}$$

In order to compute eq. 2.10 with the spherical constraint, we can define $d\mu(w)$ as

$$d\mu(w) = \frac{dw\delta(w^2 - N)}{\int dw\delta(w^2 - N)} \tag{2.11}$$

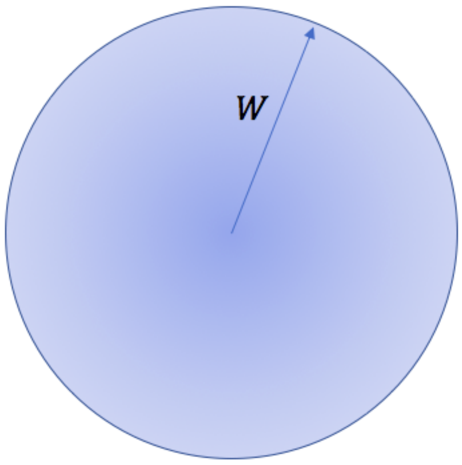


Fig. 2.2 Projection of the phase space for the spherical perceptron. The vector indicating the configuration is **W** and just the solutions lying on the hypersphere of radius $\sqrt{N}$ are here acceptable.

which gives the fractional phase space in the hypersphere having radius $\sqrt{N}$. A schematic representation can be seen in Fig. 2.2
When we let $\beta \to \infty$ in eq. 2.9 we are investigating the minima of the energy function, i.e. the solutions of our learning problem (which are occupying a subset of this space also known as *version space*). In this limit, the computation of the partition function reduces to the computation of the volume Ω of the version space

$$\Omega = \int d\mu(w) \prod_{\mu=1}^{M} \Theta(y^\mu \sigma^\mu) \tag{2.12}$$

Ω provides the information whether a solution exists (SAT, having $\Omega > 0$) or no solution is available (UNSAT).
In order to study analytically the properties of Ω the case of i.i.d. random patterns, the key obstacle is to perform the average over the random patterns while computing the partition function Z.
Z (or Ω) is an exponentially fluctuating random variable and in order to find its most probable values we need to average its logarithm, a complicated task which we perform by the replica method. Once this is done, the typical value of Z can be recovered by

$$Z_{\text{typ}} \simeq e^{N\langle \log Z \rangle_\xi} \tag{2.13}$$

where $\langle\cdot\rangle_\xi$ stands for the average over the random patterns.
We refer to [41] for a thorough review of the replica method. Here we just remind the reader that the replica method is an analytic continuation technique which allows in some cases (*mean-field models*) to compute the expectation of the logarithm of the partition function from the knowledge of its integer moments. The starting point is the following small n expansion

$$Z^n = 1 + n \log Z + o\left(n^2\right) \tag{2.14}$$

This identity may be averaged over the random patterns and gives the average of the log from the averaged n-th power of the partition function

$$\langle \log Z \rangle_\xi = \min_{n \to 0} \frac{\langle Z^n \rangle_\xi - 1}{n} \tag{2.15}$$

The idea of the replica method is to restrict to integer n and to take the analytic continuation $n \to 0$:

$$\langle Z^n \rangle_\xi = \prod_{a=1}^{n} \langle Z_a \rangle_\xi = \sum_{\{\mathbf{W}^1,\ldots,\mathbf{W}^n\}} \left\langle e^{-\beta \sum_{a=1}^{n} E(\mathbf{W}^a)} \right\rangle_\xi \tag{2.16}$$

We have n replicas of the initial model. The random patterns in the expression of the energy disappear once the average has been carried out. Eventually one computes the partition function of an effective system of $n \cdot N$ variables with a non random energy function resulting from the average. The result may be written formally as

$$\langle Z^n \rangle_\xi = e^{N \cdot F(n)} \tag{2.17}$$

where F is the expression resulting from the sum over all configurations. Once the small n limit is taken, the final expression can be estimated analytically by means of the saddle-point method given that N is assumed to be large.

2.2.1 Teacher-student

In this scenario, a teacher $\mathbf{T}$ assigns some labels σ^μ to input patterns ξ^μ. The student $\mathbf{W}$, which is in our case the perceptron to be trained, attempts to match its y^μ to the desired σ^μ. Notice that the teacher-student scenario is general and applies to any

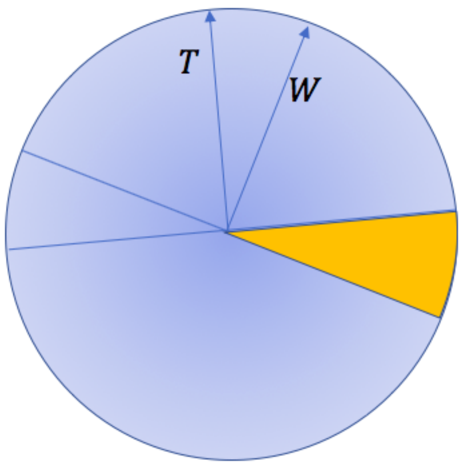


Fig. 2.3 Projection of the phase space for the spherical perceptron in the teacher-student scenario. Here **W** represents the student while **T** is the teacher. When there is complete overlap between teacher and student the orange area completely disappears, and it represents an inverse measure than the overlap.

neural network architecture. The final aim of the student (whose synaptic weights are w) is to match as much as possible those of the teacher. In such a way, a measure can be defined, namely the *teacher-student overlap*

$$R = \frac{\mathbf{T}' \cdot \mathbf{W}}{||\mathbf{T}||_2 ||\mathbf{W}||_2} = \frac{\mathbf{T}' \cdot \mathbf{W}}{N} \tag{2.18}$$

where $\cdot$ indicates the scalar product and $|| \cdot ||_2$ is the euclidean norm. R is nothing but the cosine distance between the two synaptic coupling arrays **T** and **W**. A graphical representation of the problem can be found in Fig. 2.3.
Recalling eq. 2.12, the volume of the version space becomes

$$Z_{ts} = \int d\mu(\mathbf{W}) \prod_{\mu=1}^{M} \Theta \left[\left(\frac{\mathbf{T} \cdot \xi^\mu}{\sqrt{N}} \right) \left(\frac{\mathbf{W} \cdot \xi^\mu}{\sqrt{N}} \right) \right] \tag{2.19}$$

Using the replica trick, we can write

$$\begin{aligned}\langle Z_{ts}\rangle &= \lim_{n\to 0}\tfrac{1}{n}\log\left\langle\left\{\int d\mu(\mathbf{W})\prod_{\mu=1}^{M}\Theta\left[\left(\tfrac{\mathbf{T}\cdot\xi^{\mu}}{\sqrt{N}}\right)\left(\tfrac{\mathbf{W}\cdot\xi^{\mu}}{\sqrt{N}}\right)\right]\right\}^{n}\right\rangle_{\xi}\\ &= \lim_{n\to 0}\tfrac{1}{n}\log\left\langle\int\prod_{a=1}^{n}d\mu(\mathbf{W}^{a})\prod_{a=1}^{n}\prod_{\mu=1}^{M}\Theta\left[\left(\tfrac{\mathbf{T}\cdot\xi^{\mu}}{\sqrt{N}}\right)\left(\tfrac{\mathbf{W}^{a}\cdot\xi^{\mu}}{\sqrt{N}}\right)\right]\right\rangle_{\xi}\end{aligned} \tag{2.20}$$

Here we are sampling the couplings according to

$$P(\xi)=\prod_{i=1}^{N}\left(\frac{\xi_i-1}{2}+\frac{\xi_i+1}{2}\right) \tag{2.21}$$

where we assume uncorrelated input patterns and

$$P(\mathbf{T})=(2\pi e)^{-\frac{N}{2}}\delta(\mathbf{T}^2-N) \tag{2.22}$$

in which the synaptic couplings of the teacher are uniformly distributed over the hypersphere.
At this point we need to introduce two auxiliary variables, q^{ab} and R^{a}:

$$q^{ab}=\frac{\mathbf{W}'_a\cdot\mathbf{W}_b}{N} \tag{2.23}$$

which is the overlap between two given replicas a and b and

$$R^{a}=\frac{\mathbf{T}'\cdot\mathbf{W}_a}{N} \tag{2.24}$$

is the overlap between the teacher and the replica a. The typical value for these quantities is independent on the replica index as all the replicas play a symmetric role: we can substitute $q^{ab}=q$ and $R^{a}=R$. Now, it is possible to solve the saddle-point equations under the assumption of replica-symmetry ansatz (RS solution).
In the end, the value of the overlap can be expressed as

$$R=\frac{\alpha}{\pi}\sqrt{1-R}\int Dt\,\frac{e^{-\frac{Rt^2}{2}}}{\mathrm{H}(t\sqrt{R})} \tag{2.25}$$

where $\mathrm{H}(\cdot)=\int_x^{\infty}Dt$ is a gaussian integral and $\alpha=\frac{M}{N}$. Eq. 2.25 is fundamental for the description of the behavior of R as a function of α. When $\alpha=0$, we have almost no overlap ($R = 0$) while, for $\alpha \to \infty$, $R \to 1$. This results is also intuitive: whenever a teacher exists, the student will always be able to find a configuration solving the problem and, in the limit of large alpha, it will recover exactly the teacher configuration. This means that the teacher-student problem is always satisfiable (SAT).

2.2.2 Storage problem

In the case of the storage problem, we have no longer a teacher we need to learn from. Instead, we have a random distribution for the labels y^μ. In order to ensure that the patterns are correctly stored, we need to impose

$$y^\mu \mathbf{W}' \cdot \xi^\mu \geq k \, \forall \mu \tag{2.26}$$

where k is said *stability*. In such a context, we are looking for the so-called *critical alpha* α_c, which is the highest $\frac{M}{N}$ ratio for which we are able to learn all the patterns. As we have already done in Sec. 2.2.1, we can use the Gardner analysis for the computation of the typical entropy (or volume)

$$S^{typ} = \frac{1}{N} \langle \log Z \rangle_\xi \tag{2.27}$$

where

$$Z = \int d\mu \prod_{\mu=1}^{M} \Theta\left(\frac{y^\mu}{\sqrt{N}} \mathbf{W}' \cdot \xi^\mu - k \right) \tag{2.28}$$

As done in Sec. 2.2.1, we can use the replica trick and, in the RS ansatz, we can write the saddle-point equation

$$\frac{q}{1-q} = \frac{\alpha}{\pi} \int Dt e^{-\frac{(k - t\sqrt{q})^2}{1-q}} \left[\mathrm{H}\left(\frac{k - t\sqrt{q}}{\sqrt{1-q}} \right) \right]^{-2} \tag{2.29}$$

Here, when $\alpha = 0$, also $q = 0$, which means that all the configurations are admissible solutions. Of course, having no patterns to learn, any solution is here acceptable. We are going to look for the α_c such that $q \to 1$. In such a limit, all the solutions

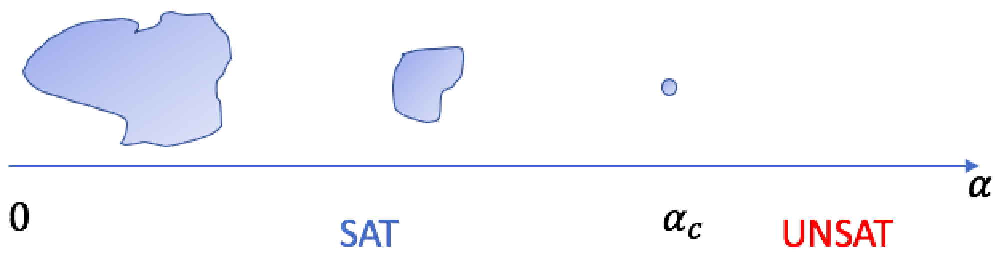


Fig. 2.4 A schematic representation on how the version space Ω modifies as α increases. It does exist a critical value α_c above which the version space no longer exists and our problem is unsatisfiable.

collapse in the same configuration and the volume reduces to a single point.
Under the limit $q \to 1$, eq. 2.29 becomes

$$\frac{1}{\alpha_c} = \int_{-\infty}^{k} Dt(k-t)^2 \tag{2.30}$$

Here, for $k = 0$, we find the maximum quantity of storable information in the spherical perceptron, $\alpha_c = 2$. As k increases, α_c decreases, as expected. α_c constitutes a phase transition point between a SAT region and an UNSAT region where no solution exists. An idea on what happens to the version space as α increases is given in Fig. 2.5.

2.2.3 Learning algorithms

Despite architectural simplicity, we have seen that the analysis of the solutions for the spherical perceptron is not trivial. However, a number of learning algorithms have been designed, some of them more effective than others. In what follows, we are just describing two among the several possible training strategies for the perceptron problem.

The perceptron learning rule

The most ancient learning rule we discuss is the so-called *Hebb rule*, introduced by Donald Hebb itself in the 1940s which in turn is based on the much older work on *classical conditioning* by Ivan Pavlov from the end of the 19th century.

The Hebb rule follows a simple strategy: in particular, all the connections in the perceptron which fire together are strengthened. The Hebb rule is general and typically applies to systems with many-neurons: its equivalent for the perceptron problem is known as *perceptron learning rule*.
Before the training begins, an initial value for the synaptic couplings **W** sampled from a gaussian distribution having mean 0 and some standard deviation.
As a first step, we present to the network a pattern ξ^{μ}. Here, the network, according to eq. 2.1, generates an output y^{μ} which is compared to the desired output y^{μ}:

- if $y^{\mu} = \sigma^{\mu}$ then the network has correctly classified the pattern and no modifications to **W** is required for the μ-th pattern.
- if $y^{\mu} \neq \sigma^{\mu}$ then the network is not correctly classifying ξ^{μ} and some modifications of **W** is necessary. In this case, we push the configuration of the network to properly classify ξ^{μ} with the required sign provided by σ^{μ}.

The update of **W** using the perceptron learning rule in the spherical perceptron can be written as

$$w_i^{t+1} = w_i^t + \Theta(-y^{\mu}\sigma^{\mu})\frac{\eta}{\sqrt{N}}y^{\mu}\xi_i^{\mu} \tag{2.31}$$

where η is a positive real number named *learning rate*. Using this update rule we are minimizing the loss function

$$L = \sum_{\mu=1}^{M} \Theta(-y^{\mu}\sigma^{\mu})\left(-y^{\mu}\frac{\mathbf{W}'\cdot\xi^{\mu}}{\sqrt{N}}\right) \tag{2.32}$$

The Δ rule

The Delta rule is a gradient descent-based learning rule for single-layered artificial neural networks. In particular, it is a special case of the more general backpropagation algorithm.
This technique, differently from the perceptron rule, is not applicable to networks

having non-differentiable activation functions (like the sign activation). Let us assume we wish to work with the mean squared error as our loss function:

$$L = \frac{1}{2M}\sum_{\mu=1}^{M}(\sigma^\mu - y^\mu)^2 = \frac{1}{2M}\sum_{\mu=1}^{M}\left[y^\mu - \phi\left(\frac{\mathbf{W}'\cdot\xi^\mu}{\sqrt{N}}\right)\right]^2 \tag{2.33}$$

Here we could compute the derivative of the loss function with respect to the parameter to be updated w_i for a single presented pattern ξ^μ:

$$\frac{\partial L^\mu}{\partial w_i} = \left[\sigma^\mu - \phi\left(\frac{\mathbf{W}'\cdot\xi^\mu}{\sqrt{N}}\right)\right]\phi'\left(\frac{\mathbf{W}'\cdot\xi^\mu}{\sqrt{N}}\right)\xi_i^\mu \tag{2.34}$$

and a GD-based optimizer can be used for training the network. This technique is easily applicable to multi-layer ANNs thanks to the chain rule, with the assumption of using a differentiable $\phi(\cdot)$ function.
If we wish to compare the delta rule with the perceptron rule, we can see that while with the perceptron rule we have no clue on how far we are from the correct configuration having just a binary signal stating whether a pattern is correctly or incorrectly classified, in the delta rule we have an error signal which decreases coming closer to the solution of the problem, which is a wise re-weighting of the error signal.

2.3 Binary perceptron

The so-called *binary perceptron* is the usual model

$$y^\mu = \text{sign}\left(\sum_{i=1}^{N} w_i\xi_i^\mu\right) \tag{2.35}$$

but with binary weights $w_i \in \{-1;+1\}$. Such a constraint introduces a direct similarity with the Ising model, which can be recovered assuming the parameters being magnetic couplings. This is the reason for which the binary perceptron is also known as *Ising perceptron* [42].

2.3.1 Ising model vs Ising perceptron

The Ising model is one of the fundamental models in statistical mechanics. Its initial purpose was to describe magnetism in the matter and is particularly useful to describe the phase transition between paramagnetism and ferromagnetism. It is named after Ernst Ising, who worked on this as a student of Wilhelm Lenz, in 1920. The model consists in some coupling variables (spins) which represent magnetic dipole moments of atomic spins. Typically, their allowable values are $\pm$ 1. Each spin is arranged in a graph structure (typically, lattice) and is able to interact with its neighbors through a pairwise *interaction term I*.
Let us consider a d-dimensional lattice, made of N different sites. For each of these we have a spin $s_i \in \{-1;+1\}$. We define a *configuration* of the system an assignment $\mathbf{s}$. For any i, j adjacent sites there is an interaction term $I_{i,j}$. In general, at the i-th site an external magnetic field h_i is applied. The energy of the systems can be written as

$$H(\mathbf{s}) = -\sum_{(i,j)} I_{i,j} s_i s_j - \sum_i h_i s_i \tag{2.36}$$

The probability distribution for the configurations $\mathbf{s}$ is given by the Boltzmann distribution

$$P(\mathbf{s}) = \frac{e^{\beta H(\mathbf{s})}}{Z} \tag{2.37}$$

where Z is the partition function computed at the inverse temperature β.
The Ising model can be directly mapped onto the perceptron by considering the spin variables as the input of our system and the couplings as the parameters we aim to learn.

2.3.2 Storage problem

Using the replica trick and the Gardner analysis it is possible to compute the critical alpha also for the binary perceptron. The modification to be introduce is the replacement of the integral over the configurations of $\mathbf{W}$ with a sum over all the discrete allowable configurations. Using the RS ansatz, the quenched entropy is given by

$$S^{typ}(\alpha) = extr_{q,\hat{q}}\left[-\frac{\hat{q}}{2}(1-q) + \int Dz \log 2\cosh\left(z\sqrt{\hat{q}}\right) + \alpha \int Dt \log \mathrm{H}\left(\frac{k-t\sqrt{q}}{\sqrt{1-q}}\right)\right] \tag{2.38}$$

where the overlap parameter q and the auxiliary parameter $\hat{q}$ are computed from the saddle-point equations

$$q = \int Dz \tanh^2\left(z\sqrt{\hat{q}}\right) \tag{2.39}$$

$$\hat{q} = \frac{\alpha}{2\pi(1-q)} \int Dt e^{-\frac{(k-t\sqrt{q})^2}{1-q}} \left[\mathrm{H}\left(\frac{k-t\sqrt{q}}{\sqrt{1-q}}\right)\right]^{-2} \tag{2.40}$$

All the details about the presented results are extensively shown in [43]. At this point, for $k = 0$, in the case $q \to 1$, we have that $\hat{q} \sim \frac{\alpha}{2\pi(1-q)^2}$ and we would derive a critical value of alpha $\alpha_c = \frac{4}{\pi} = 1.27$. This value is lower than the critical value found for the spherical perceptron (which was 2), however it is incorrect. $\alpha_c = 1.27$ means that we are theoretically able to map $1.27N$ bits (as we are working with a binary network) in the structure of the binary perceptron which only has N bits. So, where is the problem?
In order to find the correct result one needs to either go beyond the replica symmetry ansatz or identify the critical capacity by looking at the value of alpha at which the RS entropy vanishes. These two results are in fact equivalent, as discussed by Krauth and Mézard in 1989 [44]. A careful analysis of the version space shows that in case of the binary perceptron it is never connected.
This scenario is known in statistical physics of disordered systems as *replica symmetry breaking* (RSB) scenario, where the minima of the energy function have a smaller symmetry that the function itself. Geometrically, this corresponds to a shattering of the version space. Using the zero-entropy solution, the critical alpha value is $\alpha_c = 0.83$, which is consistent with the information theoretic bound.

2.3.3 Learning algorithms

Training a binary perceptron for a classification task is not as easy for a continuous perceptron. As we have seen, for the spherical perceptron problem the version space

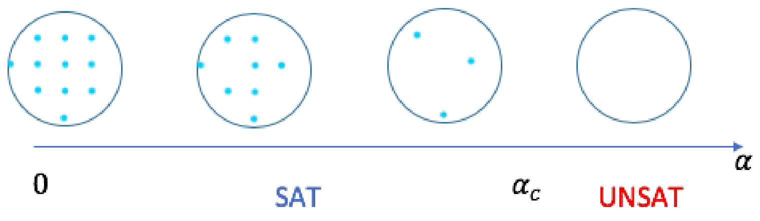


Fig. 2.5 A schematic representation on how the version space Ω modifies as α increases for the binary perceptron. Differently as seen for Fig. 2.5, the version space is fractured in non-connected domains, which makes the RS assumption failing.

is all connected, until it reaches the critical alpha. However, for the binary perceptron, we need to find solutions for the 1-RSB ansatz, meaning in a non-connected solution space. In particular, if we analyze the phase diagram for the binary perceptron case, we will see that we have a *spin glass phase*. According to the theory for disordered systems, this phase corresponds the existence of a large number of meta-stable states separated by energy barriers which slow down the learning dynamics.
For this reason, algorithms performing local search are not particularly effective to solve the Ising perceptron model.
Typical solvers for the binary perceptron model involve the implementation of Metropolis-Hastings sampling combined with simulated annealing strategy. This approach allows to find a solution to the learning problem; however, it results in a very long training time and is typically algorithmically inefficient.
Another possible approach is to use the *clipped Hebb rule*. It consists in applying the rule as described in sec. 2.2.3, but clipping the contribution to the learned parameters to ± 1. Similarly, it is possible as well to consider a properly-weighted perceptron rule.
Recently, some new innovative and effective learning algorithms have been proposed. In particular, they are based on the *Belief Propagation* (BP) algorithm [45]. This powerful tool is able to compute marginals of some variables of interest (in our case, the weights of the neural network), and are able to reach algorithmic critical capacities around 0.75, very close to the theoretical limit.

2.3.4 Dense clusters

As we have discussed in Sec. 2.3.2, for the binary perceptron there is a massive presence of isolated solutions, which makes typical local-search heuristics ineffective.

However, we have also seen in Sec. 2.3.3 that there exist at least one a class of algorithms, based on belief propagation, that are rather effective in solving the problem. A recent work [46] suggested that such a behavior could be explained by the existence of rare, dense clusters of solutions (hence, solutions are no longer all isolated) which are the weight configurations accessible by these algorithms.
To identify analytically regions having an exponential number of solutions we need to define a new measure which *counts* all the configurations $\mathbf{W}$ solving the classification problem around a reference solution $\tilde{\mathbf{W}}$. We hereby introduce the concept of *local entropy* [47]

$$S^{loc}(\tilde{\mathbf{W}},l) = \frac{1}{N}\log\sum_{\mathbf{W}}\prod_{\mu}\Theta(y^{\mu}\sigma^{\mu})\delta(d(\mathbf{W},\tilde{\mathbf{W}})-l) \tag{2.41}$$

where $d(\cdot)$ is a function determining a distance between two synaptic coupling configuration (it could be a Euclidean norm, or just the Hamming distance for the binary perceptron) and l is the distance which identifies the regions inside which we want to compute the local entropy.
A nice property of the local entropy is that it is a self-averaging quantity: if we average it on the training set we are able to know how many typical solutions exist surrounding our reference configuration $\tilde{\mathbf{W}}$:

$$S^{FP}(l) = \left\langle \sum_{\tilde{\mathbf{W}}}\prod_{\mu}\Theta(y^{\mu}\sigma^{\mu})S^{loc}(\tilde{\mathbf{W}},l)\right\rangle_{\Xi} \tag{2.42}$$

where S^{FP} stands for *Franz-Parisi entropy* [48]. Investigating this measure, it was observed that typical solutions for the binary perceptron are isolated (and this confirms the failure of the RS assumption) but some solutions, in particular those found with the abovementioned algorithms, are not isolated. This analysis shows the existence of *dense clusters* of solutions which are however sub-dominant (and, for instance, rare). The existence of dense clusters allows learning algorithms to solve the binary perceptron problem. The analytical results have been corroborated by the implementation of a Montecarlo strategy which proved to be effective in the search of these dense clusters [49], or even in optimizers for deep networks [50].

2.4 Stochastic perceptron

In the previous sections we have seen the analysis of the perceptron problem, starting from the continuous case and evolving to the Ising perceptron. It would be nice to design a method which is computationally less expensive than the top state-of-the-art algorithms which still lands in the dense solution regions (so, it must intrinsically use the concept of local entropy).
Towards this end, we studied a model of stochastic perceptron. In particular, we aim to solve the learning problem

$$\min_{\mathbf{W}} \mathrm{H}(\mathbf{W}) = \sum_{\mu} \Theta(-\sigma^{\mu} \sum_{i=1}^{N} w_i \xi_i^{\mu}) \tag{2.43}$$

where now the $w_i \in \{-1;+1\}$ are stochastic variables. We can re-frame this as a log-likelihood maximization problem

$$\max_{\mathbf{W}} \mathscr{L}(\mathbf{W}) = \sum_{\mu} \log P(\sigma^{\mu}|\xi^{\mu}, \mathbf{W}) \tag{2.44}$$

which can be solved using gradient descent-based techniques minimizing the loss function $-\mathscr{L}(\mathbf{W})$. In our model we associate a family of distributions $Q_m(\mathbf{W})$ to the synaptic configurations $\mathbf{W}$ parameterized by some variables m. Our learning problem becomes in this way

$$\max_{\mathbf{m}} \mathscr{L}(\mathbf{m}) = \sum_{\mu} \log \mathbb{E}_{\mathbf{W}\sim Q_m} P(y^{\mu}|\xi^{\mu}, \mathbf{W}) \tag{2.45}$$

From eq. 2.45 we are able to derive two different classes of predictors:

- $\sigma_1 = \underset{\mathbf{W}}{\mathrm{argmax}} Q_m(\mathbf{W})$
- $\sigma_2 = \underset{\mathbf{W}}{\mathrm{argmax}} \int d\mathbf{W} P(y^{\mu}|\xi^{\mu}, \mathbf{W}) Q_m(\mathbf{W})$

In our work [51] we decided to focus our attention on σ_1.
Maximizing the log-likelihood which depends on $\mathbf{m}$ gives us some advantages in the computation. In particular, many optimizing techniques commonly used in deep learning (some examples have been discussed in Sec. 1.5.4) are applicable to

continuous problems, but not to the discrete ones. The stochastic perceptron model allows learning binary synapses, working with a log-likelihood, whose parameter are real values. The known gradient descent techniques already incorporate a sort of knowledge of local entropy (see Sec. 1.5.4); hence, thanks to our model, we are able to make use of the nice properties of the solution space in the binary perceptron.

2.4.1 Formulation

In the optimization problem introduced in eq. 2.45 we randomly extract the configuration for the weights **W** according to

$$Q_{\mathbf{m}} = \prod_{i=1}^{N} \left[\frac{1+m_i}{2} \delta w_i, +1 + \frac{1-m_i}{2} \delta_{w_i,-1} \right] \tag{2.46}$$

where $\delta_{a,b}$ is the Kronecker delta. Of course the model is valid for $m_i \in [-1;+1] \forall i$ and m_i, named *magnetizations*, are our control parameters. Extracting the configuration from eq. 2.46, according to the hamiltonian formulation in eq. 2.43, we choose to use the probability

$$P(y^\mu | \xi^\mu, \mathbf{W}) = \Theta \left(y^\mu \sum_{i=1}^{N} w_i \xi_i^\mu \right) \tag{2.47}$$

Hence, the log-likelihood we aim to maximize reads

$$\mathscr{L}(\mathbf{m}) = \sum_\mu \log \Theta \left(y^\mu \sum_{i=1}^{N} w_i \xi_i^\mu \right) Q_{\mathbf{m}} \tag{2.48}$$

Here we can make the assumption of having large N. From the central limit theorem, for $N \to \infty$, the weighted sum over the N synaptic couplings becomes gaussian-distributed, reading

$$\mathscr{L}(\mathbf{m}) = \sum_\mu \log \mathrm{H} \left(- \frac{y^\mu \sum_{i=1}^{N} w_i \xi_i^\mu}{\sqrt{\sum_{i=1}^{N} (1-m_i^2)(\xi_i^\mu)^2}} \right) \tag{2.49}$$

where

$$\mathrm{H}(z) := \int_x^{\infty} dz \frac{e^{-\frac{z^2}{2}}}{\sqrt{2\pi}}$$

If the input is binary having $\xi_i^\mu \in \{-1;+1\} \forall i, \mu$, eq. 2.49 becomes

$$\mathscr{L}(\mathbf{m}) = \sum_\mu \log \mathrm{H}\left(-\frac{y^\mu \sum_{i=1}^N w_i \xi_i^\mu}{\sqrt{\sum_{i=1}^N (1-m_i^2)}}\right) \tag{2.50}$$

At this point, we are able to use any of the GD-based techniques to maximize eq. 2.50. In particular, we will minimize $-\mathscr{L}(\mathbf{m})$. As previously observed, the validity limit for our model is $m_i \in [-1;+1] \forall i$; hence, for the update step, we need to bound all the m_i to their validity region:

$$m_i^{t+1} = \mathrm{clip}\left(m_i^t + \eta \frac{\partial \mathscr{L}(\mathbf{m}^t)}{\partial m_i^t}\right) \tag{2.51}$$

where t indicates the update time step, η is the *learning rate* and clip is the non-linear operator

$$\mathrm{clip}(z) = \begin{cases} -1 & z < -1 \\ z & -1 \leq z \leq +1 \\ +1 & z > 1 \end{cases}$$

2.4.2 Dynamics of learning

During the simulation time, it was observed that the training error was progressively approaching zero when the squared norm of the magnetization was approaching 1. Let us say we have a control parameter

$$q^* = \frac{1}{N} \sum_{i=1}^N m_i^2 \tag{2.52}$$

q^* is a global indicator of the degree of certainty of the network: when $q^* \to 0$ the sampling is almost random, having almost all $m_i \approx 0$ while, when $q^* \to 1$, almost all $m_i \approx \pm 1$ and $Q_{\mathbf{m}}$ concentrates around some particular configuration.

2.4.3 Theoretical analysis

It is possible to perform an analysis regarding the equilibrium properties of our stochastic perceptron model. We can write the partition function

$$Z = \int_{\Omega} \prod_{i=1}^{N} dm_i \delta \left(\sum_{i=1}^{N} m_i^2 - q^* N \right) e^{\beta \mathscr{L}(\mathbf{m})} \tag{2.53}$$

where δ is the delta function selecting solutions for the given q^* value, β is an inverse temperature, $\mathscr{L}(\mathbf{m})$ is the log-likelihood in our model (eq. 2.49) and Ω is the version space which, in our case, is $\Omega \in [-1;+1]$. As already observed, for the binary perceptron the RS-ansatz is unstable for large values of β; for this reason, as already seen in Sec. 2.2.1, we are supposed to break the replica symmetry. However, we are mainly interested in studying real cases in which the temperature of our system is not zero. Towards this end, we can study the stability of the saddle-point equations as a function of the control parameter q^*.

Energy of the binarized configuration

The aim of our model is to obtain a configuration for the $\mathbf{W}$, which is a binary configuration. In particular, we would like to keep the error of the so-obtained network as low as possible. The error of the network is a kind of final energy and we would like to find a solution for the zero-temperature configuration (which in our case is the binary one) with the lowest possible energy. Here we can write

$$E = \lim_{N \to \infty} \frac{1}{\alpha N} \mathbb{E} \left[\sum_{\mu} \left\langle \Theta \left(-\sigma^{\mu} \sum_{i=1}^{N} \text{sign}(m_i) \xi_i^{\mu} \right) \right\rangle \right] \tag{2.54}$$

where the thermal average is computed over $\mathbf{m}$. Notice that eq. 2.54 depends also on q^* and β, which appears in the thermal average formula. Using the usual replica technique it is here possible to compute the average energy as a function of q^*. As we see in Fig. 2.6, we have evidence that the energy approaches 0 as $q^* \to 1$. The theoretical curve is here compared with two simulations performed using SGD. What we can observe here is that the speed (for instance, the learning rate η) plays an important role in the convergence to lower energy configurations. The difference between the curves can be explained by two factors:

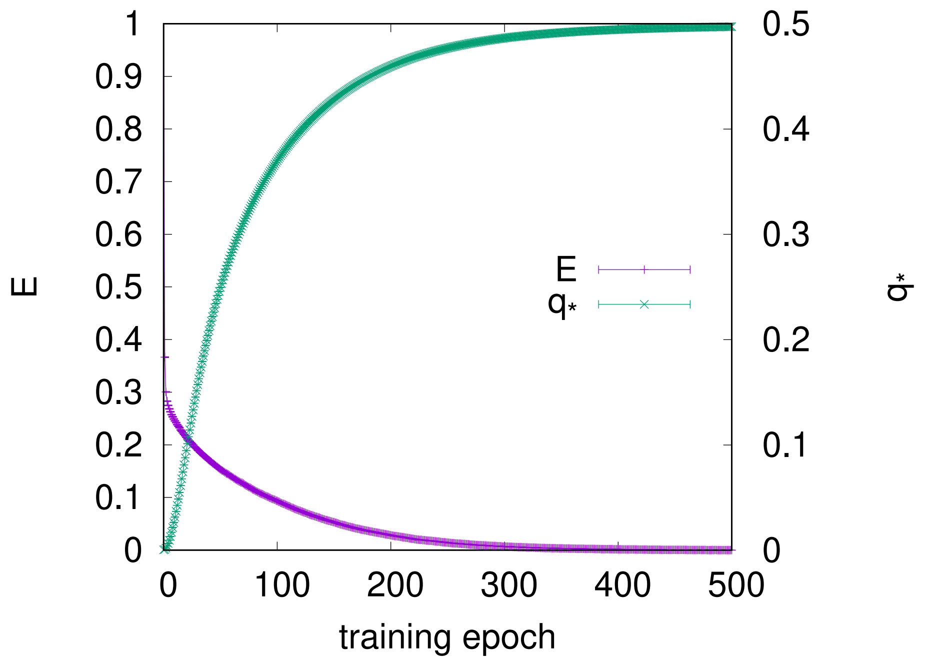


Fig. 2.6 Energy vs q^* parameter for a typical learning problem in the stochastic perceptron. Here $\alpha = 0.55$.

- *Non-equilibrium dynamics*: the theoretical computation assumes a dynamics moving from an equilibrium state to another, while SGD does not have this as constraint. However, reducing the learning rate, or even forcing the system to find a thermal equilibrium for a fixed q^* value makes the empirical energy curve getting closer to the theoretical one.
- *Finite size effect*: our theoretical formulation assumes $N \to \infty$; however, empirical results are obtained for a finite N, even though large.

Geometry of the solution space

As we have already stated in Sec. 2.3.4, in the binary perceptron the greatest number of solutions are isolated. However, a subdominant number of them concentrates in a dense, connected region [46]. It was observed that just this kind of solutions are accessible by the state-of-the-art algorithms. What we aim to show here is that the binary solutions found with the stochastic perceptron are typically inside these dense regions. In order to do this, what we can do is literally to count all the solutions to the learning problem at a fixed distance (in the binary case it is a Hamming distance

d). We can here use the approach introduced by Franz and Parisi in their work [48]. Let us define a constrained partition function

$$Z_{FP}(d,\mathbf{m}) = \sum_{\mathbf{W}} \prod_{\mu} \Theta\left(y^{\mu} \sum_{i=1}^{N} w_i \xi_i^{\mu}\right) \delta\left(N(1-2d) - \sum_{i=1}^{N} \mathrm{sign}(m_i) w_i\right) \tag{2.55}$$

in which the sum os over all the possible binary configurations. The Franz-Parisi entropy is

$$S_{FP}(d) = \lim_{N\to\infty} \frac{1}{N} \mathbb{E}\left\langle \log Z_{FP}(d,\mathbf{m})\right\rangle \tag{2.56}$$

We have compared a typical solution found with the stochastic perceptron model to typical solutions found with the binary model and the spherical perceptron model. As it is possible to see in Fig. 2.7, the solution found by the stochastic perceptron model is in a dense solution region contrarily to the typical solutions found by the other models.

2.4.4 Experimental results

The stochastic perceptron was designed such that it could be trained using simple, general optimization techniques like stochastic gradient descent, which has as strength the hidden knowledge of performing a sort of local entropy optimization. For this reason, the cost function to be minimized contains a loss term

$$L = -\frac{1}{M_b} \sum_{\mu=1}^{M_b} \log \mathrm{H}\left(-\frac{\sigma^{\mu} \sum_{i=1}^{N} m_i \xi_i^{\mu}}{\sqrt{\sum_{i=1}^{N} 1 - m_i^2}}\right) \tag{2.57}$$

where M_b is the size of the minibatch. We still have to remember the clip in the update of the magnetizations as stated in eq. 2.51. The performance of the network has to be evaluated through the binarized configuration of the network

$$\hat{\mathbf{W}} = \mathrm{sign}(\mathbf{m})$$

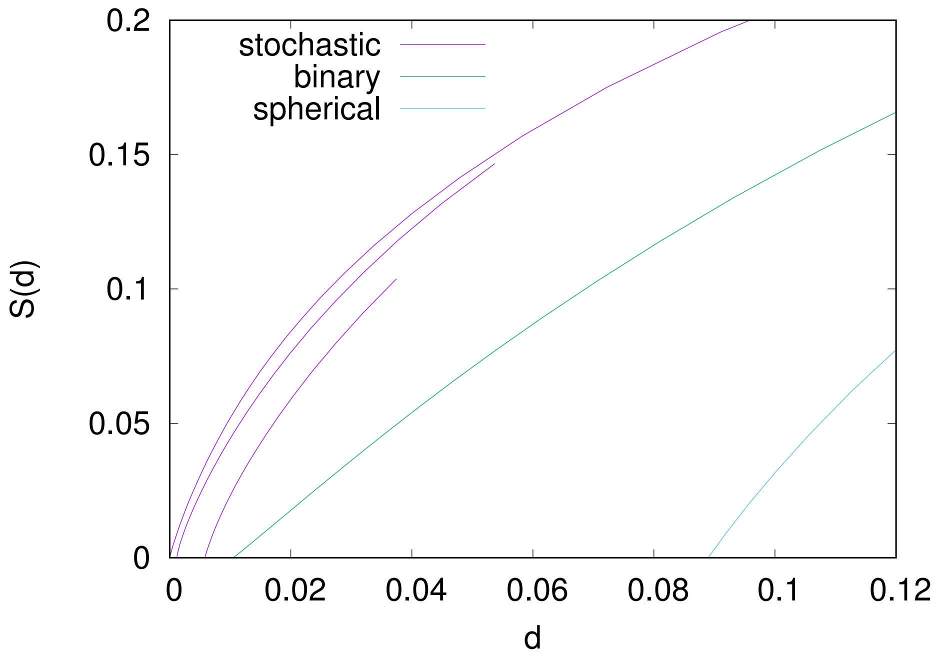


Fig. 2.7 Franz-Parisi potential for typical solutions found in the spherical, stochastic and binary perceptron at thermo-dynamical equilibrium. The curves for the stochastic perceptron are computed for different q^* values (0.7, 0.8 and 0.9): the higher it is, the more dense the region the final solution is. Here $\alpha = 0.55$, $\beta = 20$ for the stochastic perceptron while $\beta \to \infty$ for the binary and spherical perceptron.

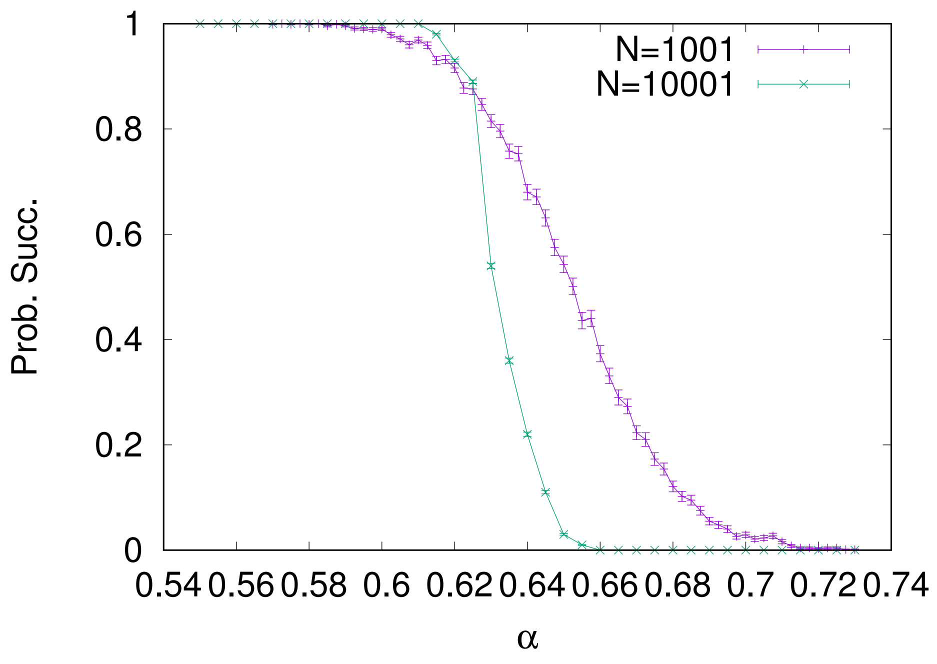


Fig. 2.8 Empirical analysis for finding-out the algorithmic critical alpha for the stochastic perceptron model. Here all the points are averaged over 100 samples and results are obtained for $N = 1001$ and $N = 10001$.

Hence, for us, the effective training error is evaluated through $\hat{\mathbf{W}}$, being

$$\hat{E} = \frac{1}{M}\sum_{\mu=1}^{N} \Theta\left(-\sigma^{\mu}\sum_{i=1}^{N}\hat{\mathbf{W}}_i \xi_i^{\mu}\right) \tag{2.58}$$

Empirically it was observed that the algorithmic critical alpha $\alpha_c^* \approx 0.63$ (see Fig. 2.8). As found from the theoretical computation, this value is below the theoretical critical capacity $\alpha_c = 0.83$, as for higher values no typical solutions can be found. Many state-of-the-art algorithms have a critical alpha value between 0.6 and 0.74; so, we are in this range. However, most of the algorithms reaching these capacities are computationally expensive and extremely complex, while our stochastic perceptron relies on a very simple and complexity-friendly training strategy.
Here we can further analyze the behavior of the average energy (error) as function of q^*. From the first simulations we observed that the behavior was different from the one we expected from eq. 2.54 (as we can see in Fig. 2.9). However, when we allow the empirical learning to find the best energy for a fixed q^* annealing the parameter more slowly, we get closer to the theoretical curve.

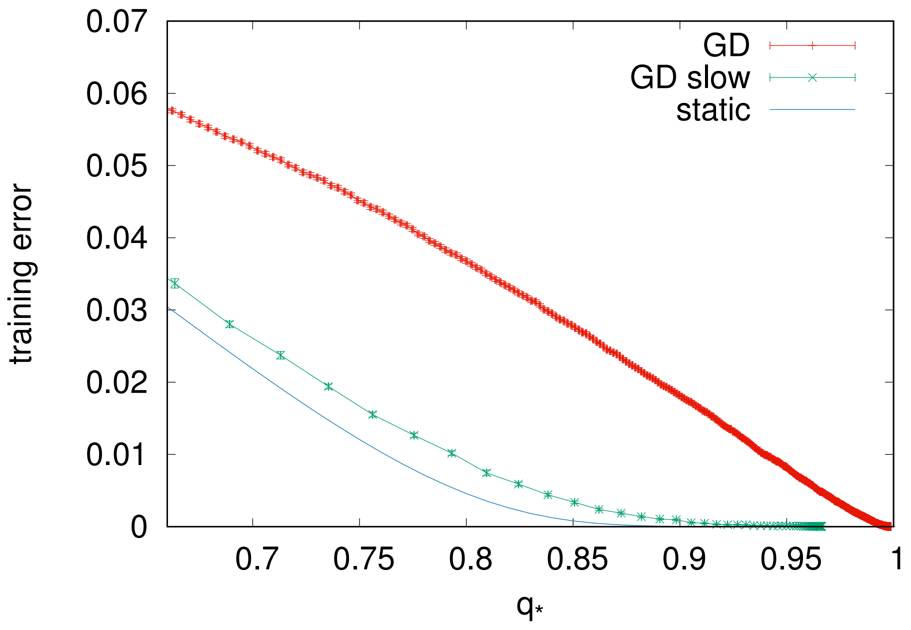


Fig. 2.9 Average energy as a function of q^* for the empirical simulations (GD) and expectations from the theory (eq. 2.54). If we do not make GD slowly thermalize the assumption of thermal equilibrium is not valid.

Some variants to our approach were also attempted: for example, instead of updating the magnetizations directly, an attempt in optimizing the fields h_i defined as

$$h_i := \tanh^{-1}(m_i)$$

was performed, as well as the use of different optimizers, with no significant change. An extension to deep network has been also done, which will be discussed in Sec. 3.5.

2.4.5 Variant: a game of scales

At this point we could be satisfied with our results. However, as already pointed-out, there are some other techniques of learning for the binary perceptron which are performing better. Of course they are more complex and the lack of performance derives from the computational efficiency and being a trade-off... What if, instead, we just need to do some adjustment to our rule?
As previously described in eq. 2.57, we are minimizing a loss function which

describes the behavior of our stochastic perceptron. The log-likelihood was obtained applying the central limit theorem for the case $N \to \infty$. Assume now to introduce a factor γ in eq. 2.57 such that

$$L = -\frac{1}{M_b} \sum_{\mu=1}^{M_b} \log \mathrm{H} \left(-\frac{\sigma^\mu \sum_{i=1}^N m_i \xi_i^\mu}{\sqrt{\gamma \sum_{i=1}^N 1 - m_i^2}} \right) \tag{2.59}$$

with $\gamma \in (0; \infty)$. This factor scales the variance in the denominator: with $\gamma \to 0$ the loss in eq. 2.59 becomes

$$L_{\gamma \to 0} = -\frac{1}{M_b} \sum_{\mu=1}^{M_b} \log \Theta \left(\sigma^\mu \sum_{i=1}^N m_i \xi_i^\mu \right) \tag{2.60}$$

Here, for all the correct predictions, the argument of the Θ function is positive and the contribution to the loss is zero; however, for the incorrect prediction, the argument of the theta function is negative, and $L \to \infty$. In such a condition, we are not allowing any error of the network and we are amplifying the loss penalty. We are in a deterministic case: the sampling distribution is reduced to a dirac delta centered to the value of m_i. On the other hand, for the case $\gamma \to \infty$,

$$L_{\gamma \to \infty} = -\log(0.5) \tag{2.61}$$

in this case, regardless the value of the magnetizations, it is like we are spreading the sample probability in the range $(-\infty; +\infty)$ and the uncertainty is at its peak. Of course, having $\gamma = 1$, we recover the standard model in eq. 2.59.
Introducing this parameter allows us to have a further control to the global scale of the variances of our algorithm and, according to the simulations in Fig. 2.10, we have a meaningful boost in the performances.
What is this telling us? We think that globally we should take care of this annealing, as also intuitively in the early stages of training the uncertainty is greater than in the last ones.

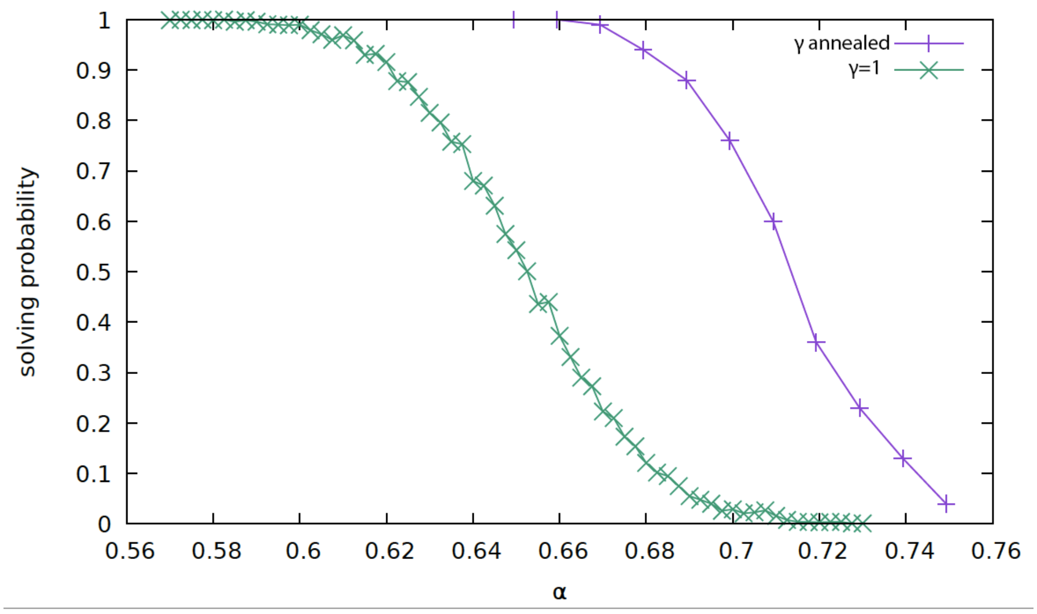


Fig. 2.10 Empirical results for $N = 1001$. There is a significant improvement in the algorithmic performance annealing γ.

Chapter 3

Feedforward networks

In Chapter 2 we have presented the perceptron from a theoretical point of view and from an algorithmic one. In particular, we have seen its closeness to the Ising model and, thanks to results and tools borrowed from the statistical physics, we have been able to perform exact computations on the maximum performances such a model can reach and how the state-of-the-art performs in terms of optimization and final performance. A fundamental result obtained from such an analysis was involving the geometry of the solution space: in the case of the continuous perceptron the solution space resulted all connected and, for this reason, any decent learning algorithm is able to find optimal solutions. However, the Ising perceptron problem is much harder as the solution space is in general not connected. In particular, it was shown the existence of some dense, sub-dominant clusters of solutions, in which there are the only algorithmically-accessible solutions.
Tn this chapter we are going to talk about the natural extension of the perceptron problem to multi-neurons architectures. These, in particular, are named *feed-forward neural network* because of their hierarchical structure: neurons are grouped in *layers* and the output signal from a lower layer is forwarded to the successive one [52]. We will see some of the most-used architectures, some theoretical computations performed and empirical investigations on the solution space inspired from the theory. In Sec. 3.1 we provide an overview on the structure of feed-forward neural networks: tree-committee machines, fully-connected networks and convolutional networks (which are the state-of-the-art architectures used for classification tasks) are presented. Then, in Sec. 3.2, we show our attempt to boost the generalization of simple feed-forward neural networks using a heuristic algorithm inspired by the replica trick. In

Sec. 3.3 we pioneer in exploring the version space for high-dimensionality learning problems, using a heuristic approach. Interestingly, the solution sub-space for the solutions found using SGD seems to be a convex connected volume. Finally, in Sec. 3.5 we extend the stochastic perceptron model to deep networks after reviewing state-of-the-art approaches in Sec 3.4.

3.1 Structure of feed-forward neural networks

We have already seen that the main components for training a neural network attempting to learn a classification task are

- some parameters $\mathbf{W}$ to be learnt
- a *training set* Ξ, made of a set of M input patterns ξ having dimensionality N, associated to a desired output y
- a topology of the network, involving how the connections are set in the network and the desired activation function ϕ appearing as the output of the network

Regarding the topology of a neural network, we can define three different types of layers depending on their positioning in the network:

- *input layer*: it is the very first layer and it is a layer with no parameters to be learned, it simply forwards the input pattern ξ^μ to the successive layer
- *output layer*: it is the very last layer of the network and provides the output y^μ.
- *hidden layer(s)*: it is any layer between the input and the output layer, its name comes from the fact that it not directly visible from the output [53]

According to this hierarchy of layers, the perceptron model has no hidden layers, the input layer has size N and the output layer's size is 1. The size of hidden layers and output layers depends on the classification task we are deciding to learn: for example, if we decide to learn a distinction between 3 different classes we could set the size of the output layer to 3 and y can be represented by a one-hot coding. We will see that in the case of deep networks the activation of the output layer is a soft-max function which normalizes the C possible outputs to 1, making y^μ a probability.

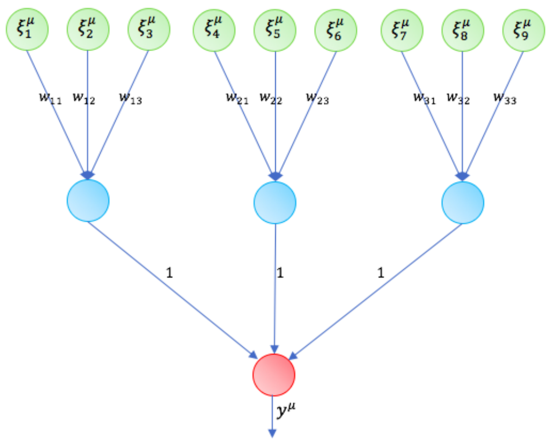


Fig. 3.1 The structure fo a TCM with $N = 9$ and $K = 3$. It should be possible also to learn the weights of the output layer.

3.1.1 Tree committee machines

The tree-committee machine is certainly one of the easiest feed-forward neural networks. It is made of 1 hidden layer of size K and input layer of size N and an output layer of size 1. Its name comes from the fact that each of the K *hidden units* belonging to the hidden layer receives as input an exclusive subset of input of size $\frac{N}{K}$ and appears to have a tree structure as it is possible to see from Fig. 3.1. The output layer takes the K outputs of the hidden layer and commonly acts like a majority voting. Hence, the output y^μ is generated:

$$y^\mu = \phi_2 \left[\sum_{j=1}^{K} w_{j,2} \phi_1 \left(\sum_{i=1}^{N/K} w_{i,1} \xi_{i,j}^\mu \right) \right] \tag{3.1}$$

where $\xi_{i,j}$ indicates the i-th input destined to the j-th neuron.

The storage problem

Let us assume for our problem we have $\phi_1(\cdot) = \phi_2(\cdot) = \text{sign}(\cdot)$ and $w_{j,2} = 1 \forall i$. Eq. 3.1 becomes

$$y^\mu = \text{sign} \left[\sum_{j=1}^{K} \text{sign} \left(\sum_{i=1}^{N/K} w_{i,1} \xi_{i,j}^\mu \right) \right] \tag{3.2}$$

assuming to have an odd value for K. From the model in eq. 3.2 we can say that a certain configuration of the network $\mathbf{W}$ is allowing the network to successfully perform the classification in the training set if

$$\prod_{\mu=1}^{M} \Theta\left(y^{\mu}\sigma^{\mu}\right) = 1$$

Hence, we are able to write the expression for the volume of the version space

$$\Omega(\xi, y) = \int \prod_{j=1}^{K} d\mu(\mathbf{w}_j) \prod_{\mu=1}^{M} \left\{ y^{\mu} \operatorname{sign}\left[\sum_{j=1}^{K} \operatorname{sign}\left(\sum_{i=1}^{N/K} w_{i,1} \xi_{i,j}^{\mu} \right) \right] \right\} \tag{3.3}$$

where $\mathbf{w}_j$ represents the subset of parameters belonging to the j-th hidden neuron and

$$d\mu(\mathbf{w}_j) = \prod_{i=1}^{N/K} \frac{dw_{j,i}}{\sqrt{2\pi e}} \tag{3.4}$$

From here the standard replica trick can be used and a calculation of the typical $S^{typ} = \log \Omega(\xi, y)$ in the RS ansatz can be performed (the same procedure we have done in Sec. 2.2.1). In the end we find the expression

$$\frac{1}{\alpha_c} = K \int_0^{\infty} Dt\, t^2 \sum_{j=0}^{\frac{K-1}{2}} \binom{K-1}{j} [\mathrm{H}(t)]^{K-1-j} [1 - \mathrm{H}(t)]^{k} \tag{3.5}$$

Hence the result depends on our choice of K. The found values of the critical alpha for the tree committee machines are 4.02, 5.78 and 7.31 for K values 3, 5, 7. Such a result is extremely relevant: even though the number of parameters in our architecture is constant (N), depending on the number of neurons (hence the topology) the learning capability changes.
However, if we perform a stability analysis on the saddle-point equations, we find that we have instabilities for $\alpha < \alpha_c$; so, the results found working with the RS ansatz are out of their validity region and we should proceed, like we did for the binary perceptron, with the replica symmetry breaking. Replica symmetry breaking can be implemented in an iterative fashion, following the Parisi's scheme [54]. The

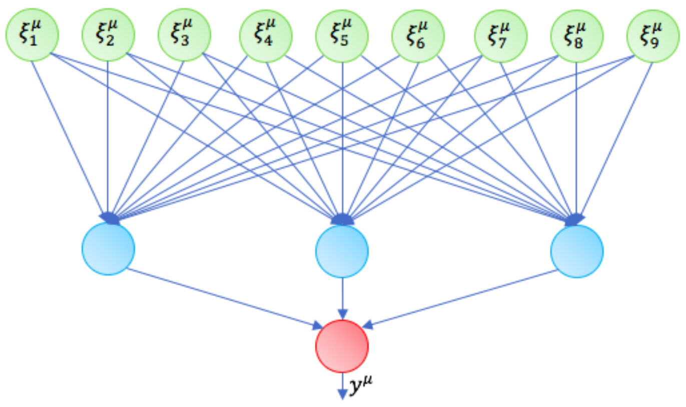


Fig. 3.2 The structure fo a one hidden layer fully-connected ANN with $N = 9$ and $K = 3$. As it is possible to see, all the hidden neurons are able to see all the inputs.

simplest level of RSB is the so-called one-step RSB (1-RSB), in which the space of minima is divided into many similar clusters. The critical alpha this way found for $K = 3$ is approximately 3.0, and still having a increasing trend for $K \to \infty$. Notice that in this limit K still grows to infinity slower than N.

3.1.2 Fully connected networks

We have seen in Sec. 3.1.1 the tree committee machine architecture. For such an architecture, every neuron in the *hidden layer* has its own exclusive subset of the inputs. If we define a matrix of weights $\mathbf{w}_1^{TCM} \in \mathbb{R}^{K\times N}$ we might say that, for the tree committee machine, we have

$$\sum_{i=1}^{N/K} w_{i,1}\xi_{i,j}^{\mu} \forall j \leftrightarrow \mathbf{w}_1^{TCM} \cdot \xi^{\mu} = \mathbf{a}_1$$

where

$$\mathbf{w}_1^{TCM} = \begin{bmatrix} w_{1,1} & \dots & w_{1,\frac{N}{K}} & 0 & \dots & 0 \\ 0 & \dots & 0 & w_{2,\frac{N}{K}+1} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & w_{K,N} \end{bmatrix}$$

and $\mathbf{a}_1$ is called *activation*. For the tree committee machine, $\mathbf{a}_1 \in \mathbb{R}^{K,M}$, where M is the number of patterns simultaneously fed to the network. As we see, $\mathbf{w}_1^{TCM}$ is a sparse matrix having just N non-null entries of the matrix. If we decide to allow all of the entries having non-null value we will have

$$\mathbf{w}_1^{FC} = \begin{bmatrix} w_{1,1} & \dots & w_{1,2} & \dots & w_{1,N-1} & w_{1,N} \\ w_{2,1} & \dots & w_{2,2} & \dots & w_{2,N-1} & w_{2,N} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{K,1} & \dots & w_{K,2} & \dots & w_{K,N-1} & w_{K,N} \end{bmatrix}$$

Hence, any of the K hidden neurons will be able to receive all the N input data $\forall \mu$. As all the possible connections between layer l and layer $l+1$ are taken into account, this kind of layer is named *fully-connected*. A sketch of the architecture is in Fig. 3.2.
This structure can be easily extended to have multiple layers, of any size: for example, if we have a fully-connected neural network having L learnable layers:

- the layer $l = 0$ is the *input layer* and has no learnable parameters.
- if the input layer has N neurons and the first layer size is K_1 the layer $l = 1$ is the first hidden layer having $N \times K_1$ learnable parameters $+K_1$ learnable *biases*.
- the layer $l = 2$ takes as input the output of layer $l = 1$; hence, the total number of learnable parameters is in this case $K_1 \times K_2 + K_2$ being K_2 the size of the second hidden layer.
- coming to the end of the network, the layer $l = L$ is said *output layer* bot its structure is always fully-connected. Contrarily to the TCM case, here the output layer has learnable parameters which are, as usual, $K_{l-1} \times K_l + K_l$.

Another name fully connected networks are also known is *multi-layer perceptron* (or MLP). Indeed, if we take any neuron not belonging to the input layer, the basic structure is the one of the neuron. This observation made most of the research in the early years focusing on the perceptron model, as it is the basic brick used to build complex artificial neural networks.
Towards this end, a theoretical analysis, similar to the one done for the perceptron

model and as well done for the TCM, has been done.
Let us take a fully connected network having $L = 2$ and $\mathbf{w_2} = \mathbf{1}$ (hence, the output layer has no learning parameters and is just a majority vote, like for the TCM case). In the TCM case we know each of the K hidden neurons would be *specialized*: each will learn something different, because they would have an exclusive subset of the inputs. In the case of fully-connected network; however, all the hidden neurons take the same inputs. This might be a problem, as for low α regimes the network can behave as a perceptron, and all the K units might learn exactly the same solution space partitioning. We expect that, at a certain alpha $\alpha^* < \alpha_c$, the neurons begin to specialize and start having different behaviors. Furthermore, we might have the same solution for $K!$ permutations of te hidden neurons, which introduces an extra order parameter in the theoretical computation. This behavior introduces algorithmic difficulties which we will investigate.

3.1.3 Convolutional networks

Nowadays, one of the most widely used architectures of neural networks is the so-called *convolutional neural network* (CNN). It is also known as the *shift invariant* or *space invariant* neural network because it relies on the concepts of weight-sharing architecture and translation invariance.
This class on neural networks was inspired by the biology: in fact, they are an attempt of modeling the topology of the animal visual cortex. Hubel and Wiesel, in the 1960s, published a work in which they studied the cat and the monkey visual cortex [55] [56]. They found that the neurons in such a region were stimulated by very small regions of the visual field. Let us say that the region which provides the stimulus to a given neuron in the visual cortex is its *receptive field*. This behavior was observed in all the visual cortex's neurons; furthermore, it has been observed that many of these had overlapping receptive fields. In the end, all the receptive fields were placed such that they covered the entire *visual space*.
These observations have been modeled in the so-called CNNs, inspiring first the well-known LeNet-5 [57].

Differences with fully-connected architectures

It was empirically observed that multi-layer perceptron architectures are extremely useful for image recognition (and, for instance, *classification*). However, because of their extremely high connectivity between nodes (if we have the number of neurons N in the same order of magnitude of the neurons in the network K, we have $O(N^2)$ parameters per layer), they suffer from the so-called *curse of dimensionality* (please refer to Sec. 1.5.8), which represents an obstacle to the learning process. In order to give an estimate of how relevant this problem is, let us take for example images from the ImageNet dataset. A standard image is large 224x224x3 (it is large 224x224 pixels and then it has 3 color channels), hence each neuron in the first layer would have approximately $150k$ parameters.
Most importantly, in fully-connected architectures, all the in-coming information is treated exactly in the same way: no spatial correlation is taken into account which, we know, is fundamental for image processing problems. An evolution of fully-connected architecture for image classification problems is here necessary.
Convolutional neural networks address the problems of spatial correlation and curse of dimensionality with the following characteristics:

- *principle of locality*: from the knowledge from the biology, each neuron has its weights (receptive field) which focuses on a small portion of the input (visual space). Then, several layers of "convolutional neurons" are stuck such that, like in a tree structure, all the inputs are combined together. Of course, the size of the receptive field is relevant: the smallest it is, the most we are focusing on details of the image. Furthermore, the hidden neurons are spatially organized as well, and they are placed in as many dimensions as the input is (hence for the images we will have a 3D organization[1], for movies it will be 4D and so on...), highly exploiting the locality principle of the information. The parameters belonging to the convolutional neuron is named *filter*.

- *weight sharing*: each neurons has its own receptive field whose size is smaller than the visual space. In order to entirely cover it, each filter is replicated. All of these units share the same parameters and in the end they form the *feature map*. The weight sharing and, hence, replicating the units, has two main

[1]recalling that the number of colors in an image (namely, the *channels*) is a dimension as well

effects: significantly reduce the number of required parameters and allowing the detection of meaningful features regardless their spatial positioning (hence, having *translation invariance*).

It was empirically observed that this topology of network, having way less parameters of fully-connected architectures, achieves better generalization performances for image classification tasks. Thanks to their scaling capabilities, nowadays they represent the basic structure of the most complex and most used artificial neural networks.
Convolutional neural networks are mainly divided in two parts, depending on the task they have to accomplish:

- *feature extraction*: here the features are extracted and then fed to the next part of the network, and this happens with proper distinction between two different kind of layers:
  - *convolutional layers*: here the true convolution operation happens
  - *pooling layers*: here a proper sub-sampling happens
- *classification*: here the classification problem is solved, not anymore starting from the raw input, but from the features extracted. Here, fully-connected layers are the most used.

Convolutional layer

Certainly, the convolutional layer is the most important part of a CNN. It is a learnable layer which extracts the most relevant features from the input, exploiting the spatial correlation between data and making the classification task easier to be solved for the successive fully-connected layers.
Every convolutional neuron has its own *filter* or *kernel*, having its receptive field which progressively is shifted through all the input. The output of this layer has a smaller dimensionality of the input and each entry is computed as the dot product between the corresponding receptive field and the kernel. Let us make a toy example. Assume our input is $\xi \in \mathbb{R}^{4\times 4}$

$$\xi = \begin{bmatrix} \xi_{1,1} & \xi_{1,2} & \xi_{1,3} & \xi_{1,4} \\ \xi_{2,1} & \xi_{2,2} & \xi_{2,3} & \xi_{2,4} \\ \xi_{3,1} & \xi_{3,2} & \xi_{3,3} & \xi_{3,4} \\ \xi_{4,1} & \xi_{4,2} & \xi_{4,3} & \xi_{4,4} \end{bmatrix}$$

and the kernel of the first neuron is $\mathbf{w}_1 \in \mathbb{R}^{2\times 2}$

$$\mathbf{w}_1 = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

We expect the output to be

$$y = \phi(\mathbf{a}_1) \tag{3.6}$$

where $\phi(\cdot)$ is a generic activation function and $\mathbf{a}_1$ is the activation

$$\mathbf{a}_1 = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

How exactly each activation is computed? Let us take, for example, the entry $a_{1,1}$:

$$a_{1,1} = \sum \tilde{\xi}_{1,1}\mathbf{w}_1$$

where $\tilde{\xi}_{1,1}\mathbf{w}_1$ is a dot product, the sum is among all the elements and $\tilde{\xi}_{1,1}$ is the receptive field which, in our case, is

$$\tilde{\xi}_{1,1} = \begin{bmatrix} \xi_{1,1} & \xi_{1,2} \\ \xi_{2,1} & \xi_{2,2} \end{bmatrix}$$

Hence, we can explicitly write the expression:

$$a_{1,1} = \xi_{1,1}w_{1,1} + \xi_{1,2}w_{1,2} + \xi_{2,1}w_{2,1} + \xi_{2,2}w_{2,2}$$

For all the other activations, the receptive field is just shifted by a quantity which typically is 1, having

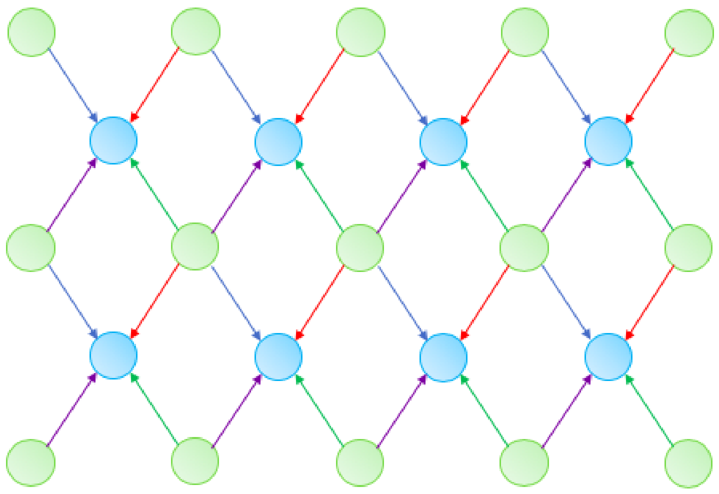


Fig. 3.3 Example of how a convolutional layer works. Here the filter is 2x2 and the input is 5x3. The input is indicated by green neurons while the output is in light blue. In this case the parameters are 4 only and the same parameter is indicated with the same color of arrow (blue, red purple or green). It is assumed here stride 1.

$$a_{1,2} = \xi_{1,2}w_{1,1} + \xi_{1,3}w_{1,2} + \xi_{2,2}w_{2,1} + \xi_{2,3}w_{2,2}$$

and so on. The shift S of the receptive field is called *stride* and in our case $S = 1$
An example of how a convolutional filter works can be found in Fig. 3.3.
Typically, a frame of zeros is added around the entire input, transforming, for example $\xi \in \mathbb{R}^{(4+P)\times(4+P)}$ into

$$\hat{\xi} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \xi_{1,1} & \xi_{1,2} & \xi_{1,3} & \xi_{1,4} & 0 \\ 0 & \xi_{2,1} & \xi_{2,2} & \xi_{2,3} & \xi_{2,4} & 0 \\ 0 & \xi_{3,1} & \xi_{3,2} & \xi_{3,3} & \xi_{3,4} & 0 \\ 0 & \xi_{4,1} & \xi_{4,2} & \xi_{4,3} & \xi_{4,4} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

where P (in our case 1) is called *padding*. There are some reasons to use padding: for example, to have the output of the same size of the original input.
In general, we can say that the final size of the activations coming from a convolution having an input ξ of size $N \times N$, a kernel $K \times K$, stride S and padding P, is

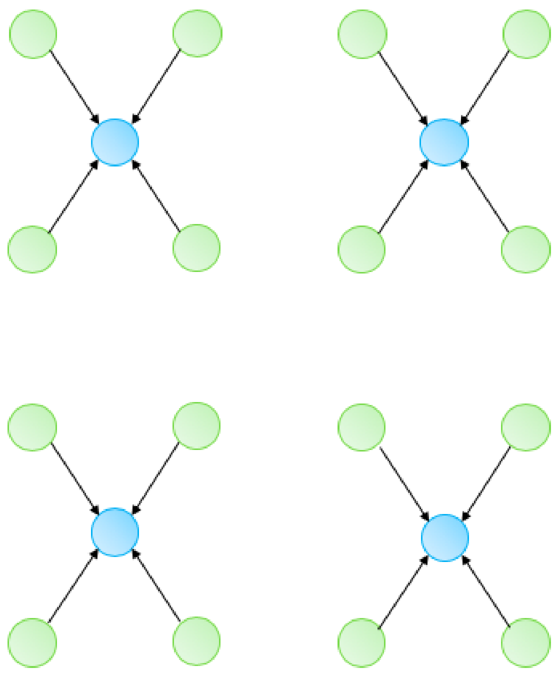


Fig. 3.4 Example of how a pooling layer works. Differently from a convolutional layer, there are no overlapping fields and there are no parameters to be learnt.

$$\text{size}(\mathbf{a}) = \left(\frac{N-K+2P}{S}+1\right) \times \left(\frac{N-K+2P}{S}+1\right) \tag{3.7}$$

Pooling layer

The so-called *pooling layer* can be inserted after a convolutional layer and is a way to down-sample an image. In practice, what is done in pooling is:

1. Take the input ξ arriving from the convolutional layer
2. Divide ξ in non-overlapping regions. Let us say for sake of simplicity, these are squared regions $\tilde{\xi}$ of size $K \times K$ (a typical value of K is 2)
3. Apply a sub-sampling function $\varphi(\tilde{\xi})$. Such a function produces 1 output per $\tilde{\xi}$.

The graphical sketch of how pooling layer works can be found in Fig. 3.4. Several choices for $\varphi(\cdot)$ are available. For example, one of the most popular in the early 2000 was the *average pool*

$$\varphi^{AVG}(\tilde{\xi}) = \left\langle \tilde{\xi} \right\rangle \tag{3.8}$$

However, the widely, by measure most-used, is *max-pool*, which outputs only the maximum of the input:

$$\varphi^{MAX}(\tilde{\xi}) = \max(\tilde{\xi}) \tag{3.9}$$

Max-pool is nowadays the most used pooling function because it empirically provides the best results.
Of course, pooling is a sub-sampling transformation which reduces the dimensionality of the problem from $\mathbb{R}^{N_1 \times N_2}$ to $\mathbb{R}^{\frac{N_1}{K_1} \times \frac{N_2}{K_2}}$ and for this appears to be very useful for the feature extraction problem.

3.1.4 The universal approximation theorem

In all the excitement coming from the possible effectiveness of multi-layer artificial neural networks, a question rises: is it possible, for this kind of networks, to properly approximate any required behavior? Of course we know this to be true for the teacher-student scenario (whenever a teacher exists, at least one student should exist), but what happens when a teacher is not guaranteed to exist?
The *universal approximation theorem* states that a feed-forward network, having one hidden layer with a finite number of neurons K, is able to approximate continuous functions under some assumptions on the activation function [58].
Assume $\phi(\cdot)$ being a continuous, bounded and strictly-increasing function. Let I_n be an n-dimensional hypercube $[-1;1]^n$, the space of continuous functions of I_n is $C(I_n)$, given any $\varepsilon > 0$ and any $y \in C(I_n)$ there exist N, $w_{2,i}, b_i \in \mathbb{R}$ and $w_{1,i} \in \mathbb{R}^n$ such that we can define

$$F(\xi) = \sum_{i=1}^{K} w_{2,i} \phi \left(b_i + \sum_{j=1}^{N} w_{1,i,j} \xi_j \right) \tag{3.10}$$

which is an approximation of any f function which is in general independent of ϕ, in other words,

$$|F(\xi) - f(\xi)| < \varepsilon \tag{3.11}$$

Such a structure recalls a TCM (see Sec. 3.1.1) with the difference that here we are actually learning the output layer and that we have overlapping input fields. This theorem provides a more analytical understanding why deep learning is nowadays successful and gives us more interest in understanding the basic mechanisms allowing the learning to proceed smooth.

3.2 Exploring solutions in TCM

The theoretical results shown for the binary perceptron model ignited the possibility of boosting the generalization in more complex artificial neural network models. In particular, the great discovery of the existence of dense solution clusters [46] gave some hints on the possibility of having a direct translation of those promising results also for deep architectures. Towards this end, some heuristic approach for training the tree committee machine attempting to maximize the entropy needed to be attempted.
Recently, a work by Zhang et al [59] indirectly went down such a path. The aim of its *elastic averaging SGD*, however, was different: boosting the convergence speed of learning.
Let us assume we have P different ANN models, trained on the same problem Ξ. Assume, as well, that stochastic gradient descent is the optimizer here used. The proposed update step for the i-th neuron belonging to the p-th ANN, is

$$w_{i,p} := w_{i,p} - \eta \frac{\partial L}{\partial w_{i,p}} - \gamma(w_{i,p} - \hat{w}_i) \tag{3.12}$$

where η and γ are two positive hyper-parameters and $\hat{w}_i$ is the average point of the same i-th parameter computed for all the P models (we call here *replica*):

$$\hat{w}_i = \left\langle w_{i,p} \right\rangle_p \tag{3.13}$$

The main result shown by the work involved the faster convergence to the result. However, there are some interesting similarities to the *replica trick* which is used for the Gardner analysis: the same system is split in replicas, and in the end of the

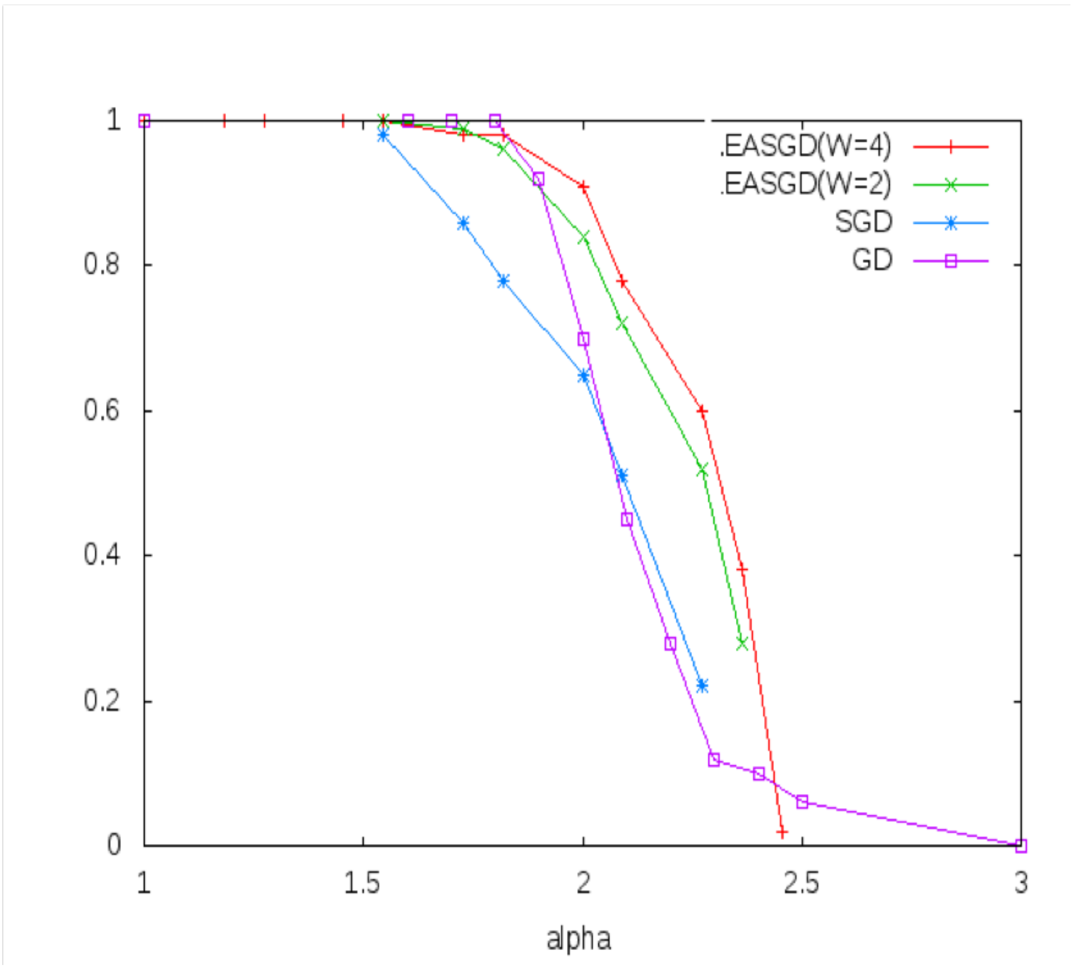


Fig. 3.5 Probability of successfully learning a classification task varying alpha. EASGD with 2 and 4 workers (W) compared to GD and SGD.

process they collapse to one, single configuration. This is supposed to allow a much more satisfactory exploration for the energy landscape (or in other words, the loss function). Could we use it to boost the performances for artificial neural networks? Some attempts have been performed towards this end. In particular, a tree committee machine has been tested with the Elastic-SGD approach, aiming to increase the algorithmically-accessible storage capacity. Figure 3.5 compares EASGD to standard SGD and to the solution reached with full gradient descent. Even though some improvement in the performances is observed, it is still not a completely satisfactory one. From another point of view, there have been recently in the literature some hints regarding the nature of local minima: a work by Lin et al [60] suggests that solutions in flat, wide minima generalize better than solutions in narrow ones. Figure 3.6 shows

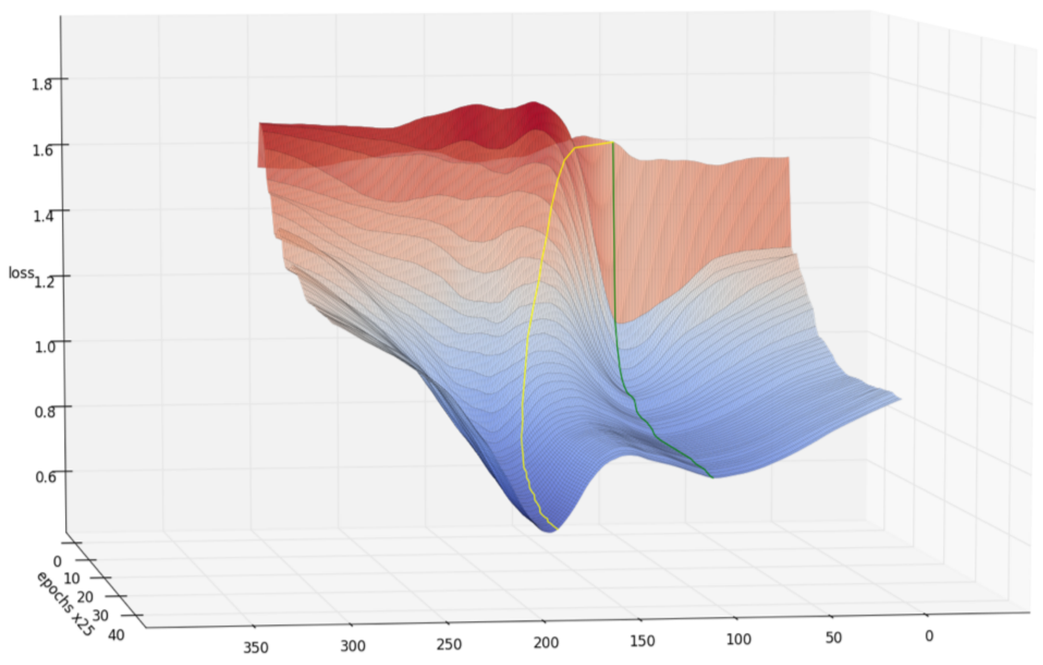


Fig. 3.6 Evolution of the energy landscape around SGD solution (yellow) and Elastic-SGD (green)

the dynamic evolution of the energy landscape surrounding a solution obtained for the same training problem by SGD and elastic-SGD. It is evident that elastic-SGD chooses a wider, flat minima than the one chose by the standard SGD approach. This was a hint suggesting that EASGD favors this class of solutions, and most likely boosts generalization.

An equivalent formulation to EASGD has been formulated by Chaudhari et al. with his Entropy-SGD [50]. He showed, indeed, that Entropy-SGD improves the generalization on datasets like MNIST and CIFAR-10 (or, at least, approaches state-of-the-art performances obtained using bunch of heuristics).

From this exploration we have learned that, probably, focusing on the optimization technique before understanding the version space we deal with (which is typically different from the perceptron problem one) might not be the best approach, even if we are driven by inspirations from the theory. In the next section we are going to deal with a more grounded levels: we are exploring the version space, aiming to find some hints on basic properties which could inspire some deeper understanding of deep models.

3.3 Exploring the version space

We would like here to explore more in depth how the solution space to a given training problem Ξ is partitioned. Of course, if we want to perform an empirical analysis, then our analysis will be restricted just to the algorithmically-accessible volumes of solution. For the continuous perceptron problem, it is theoretically demonstrated such a space being connected: for that, if we have a configuration $\mathbf{W}_a$ and a configuration $\mathbf{W}_b$, both solving the same classification problem Ξ, then we know there exists a continuous path $\Gamma_{a,b} \in \Omega$ (where Ω is the version space) which connects $\mathbf{W}_a$ to $\mathbf{W}_b$.
As we have observed in Sec. 3.1.1, the solution space for more complex architectures like TCMs appears to be in general not connected. What we are attempting to see here is whether the algorithmically accessible subspace (which we will indicate as $\Omega_{acc} \subset \Omega$) is or is not connected.
An article by Goodfellow et al. in 2014 [61] attempted to investigate such a point. In particular, given two solutions $\mathbf{W}_a$ and $\mathbf{W}_b$ it was drawn a straight line

$$\nu(t) = (\mathbf{W}_b - \mathbf{W}_a)t + \mathbf{W}_a \tag{3.14}$$

having $t \in [0,1]$ and was plotting the number of errors found along $\nu(t)$.
Goodfellow et al. observed that, in general, there is always a potential barrier separating the two solutions and in general $\nu(t)$ lies in outer regions than Ω (see Fig. 3.7). These experiments were performed on deep architectures and, as already evidenced in the theoretical analysis (Sec. 3.1.2), if there are overlapping receptive fields then there are symmetric equivalent configurations of the network we need to take care of. Hence, the result reached by such a work is not completely satisfactory and, for this reason, we are going to work on this problem with tree committee machines, which do not have the problem of taking into account symmetries.
According to our findings, the scenario described by Goodfellow et al still applies to TCMs. However, the design of $\nu(t)$ is somehow very strict and is not a sufficient condition to state whether a solution exists or not: we are going to implement a heuristic approach to explore the solution space in a more satisfactory way.

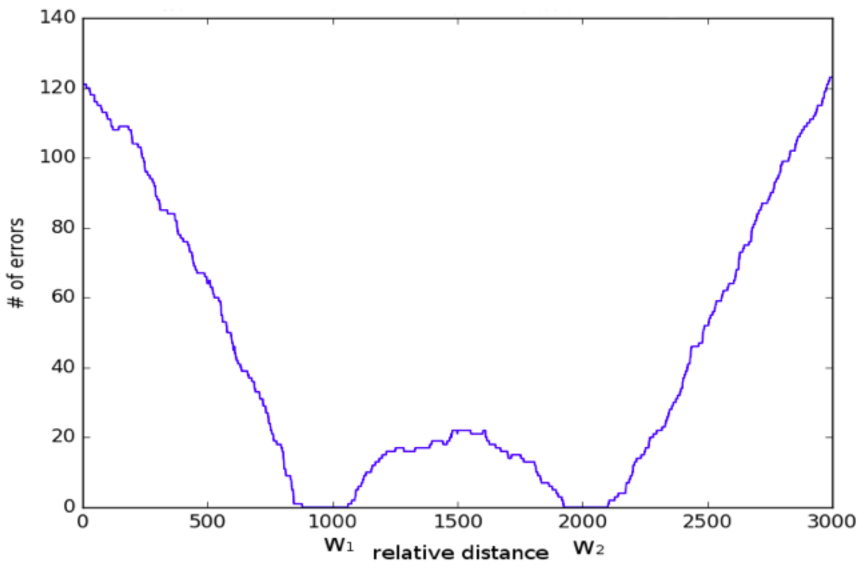


Fig. 3.7 Typical scenario observed for two generic solutions joined by $\nu(t)$. An energy barrier separates the two configurations.

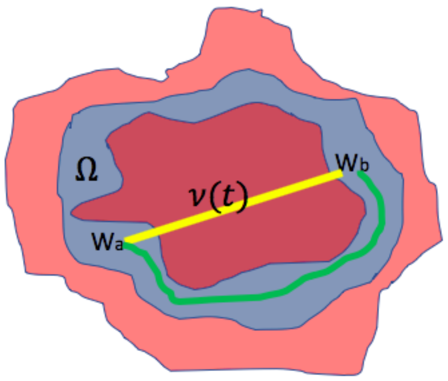


Fig. 3.8 Example of shape for the version space (Ω, in blue). The path tracked by $\nu(t)$ (yellow) typically gets out Ω, but maybe a connection path between $\mathbf{W}_a$ and $\mathbf{W}_b$ exists (green line).

3.3.1 Finding the path

The main idea is to *relax* the trajectory $v(t)$ in such a way it, in the end, will completely lie in Ω (typical scenario pictured in Fig. 3.8). However, in order to do this, we need some geometrical information about Ω we do not have a-priori as it depends on the choice of the training set Ξ. We are, however, able to compute the gradient $\forall t$, which might help us to remain in Ω. From this observation, we take a different approach.
Assume we are initializing a given neural network $\mathbf{W}_*$ to $\mathbf{W}_a$. We require $\mathbf{W}_*$ to cross some meta-states which still belong to Ω in order to reach the final state $\mathbf{W}_b$. In order to accomplish this, we still need to locally minimize the energy of our system (in our case, the value of the loss function) but we are subjected to an *elastic force* which is proportional to the distance from the target configuration $\mathbf{W}_b$

$$\Delta\mathbf{W}_* = \gamma(\mathbf{W}_b - \mathbf{W}_*) \tag{3.15}$$

with $\gamma \ll 1$. If we just applying eq. 3.15, by construction, $\mathbf{W}_*$ will reconstruct $v(t)$ just with a different parametrization. Of course, we might apply GD steps combined to these elastic coupling. The introduction of the energy minimization step, however, can potentially obstacle us. When the elastic force becomes comparable to the gradient (it may happen, especially when we are beyond the half of our path), it is possible to get stuck in a local minima. Also, what can happen is that, because of the particular shape of the zero errors region, we may be blocked in some hypercorners (we remember that, by our choice, the input patterns $\xi_i^\mu \in \{-1;+1\}\forall i$), driven by the loss function minimization. In order to overcome this problem, as the boundary of this region is shaped by hyperplans, it makes sense to impose a norm constraint on out trajectory:

$$n(\mathbf{W}_*) = ||\mathbf{W}_b||_2 - \frac{||\mathbf{W}_b||_2 - ||\mathbf{W}_a||_2}{||\mathbf{W}_b - \mathbf{W}_a||_2}||\mathbf{W}_b - \mathbf{W}_*||_2 \tag{3.16}$$

Here we are imposing a linear constraint to the norm of $\mathbf{W}_*$, which is function of the distance from $\mathbf{W}_b$. The trajectory this way obtained is no longer a straight trivial line $v(t)$ but is a more complex path $\Gamma(t)$. If the learning rate and the elastic coupling parameters are properly tuned, we observed that, in general, solutions for Ω_{acc} are connected ones. Recently a similar work was published on more complex architectures suggesting that for real problems (hence, the patterns are correlated) the algorithmically-accessible subspace of the version space is connected [62].

3.3.2 Properties of the path

As seen, we do not expect $\Gamma(t)$ being a straight-forward path. If such a path for "far" solutions is found, then we certainly are interested in finding properties for it. All the properties we are able to inspect are necessarily related to the loss function. In order to do this, we can inspect the shape of the binary cross entropy along $\Gamma(t)$. However, that would be a 1-dimensional information and, for this, not completely satisfactory. What can be more informative here is the Hessian, computed for all the points along the trajectory.
If we wanted to compute it in an efficient way, we could proceed as

$$\frac{\partial^2 L}{\partial W^2} = \frac{\partial}{\partial W}\frac{\partial L}{\partial W} = \frac{\partial}{\partial W}\nabla L\,. \tag{3.17}$$

Hence, as we already have the gradient for a given point, by locally perturbating the weights we can obtain a good approximation for the Hessian. However, thanks to the simple structure of our TCM, it is possible to compute the exact Hessian for all the points. Assuming our loss is a *binary cross-entropy*

$$L = -\frac{1}{M}\sum_{\mu=1}^{M}\left[\sigma^\mu \log y^\mu + (1-\sigma^\mu)\log(1-y^\mu)\right]\ .$$

The computation and analysis of the eigenvalues of the Hessian, from which we can deduct the local shape of the landscape (in this case, of the solution space) is a straightforward step. According to empirical simulations, a typical scenario is sketched out in Fig. 3.9: along the entire zero-energy path the eigenvalues of the Hessian are all positive and are higher in values. This means that we are climbing the loss function and that we are in non-typical solutions for the computed loss function, even though we still are in the version space. An idea for further work here could be to boost the learning by focusing on the search of these particular regions.
Experiments on LeNet-5 solutions trained on a reduced partition for the MNIST dataset (100 and 1000 examples) and on the full MNIST dataset have also been conducted. Here $\eta = 0.1$, $\gamma = 0.001$ and $N_{epochs} = 5$. The software here used is PyTorch 1.1 with CUDA 10. In spite of the higher dimensionality and complexity

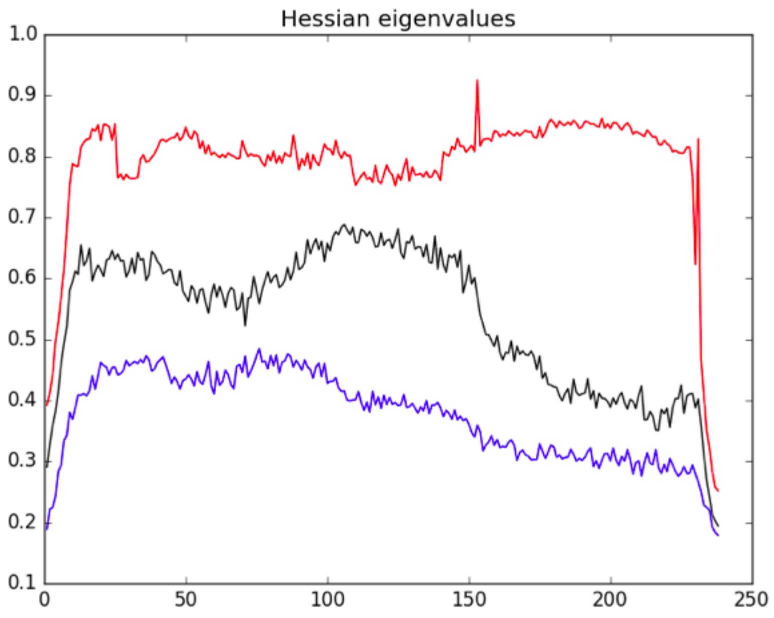
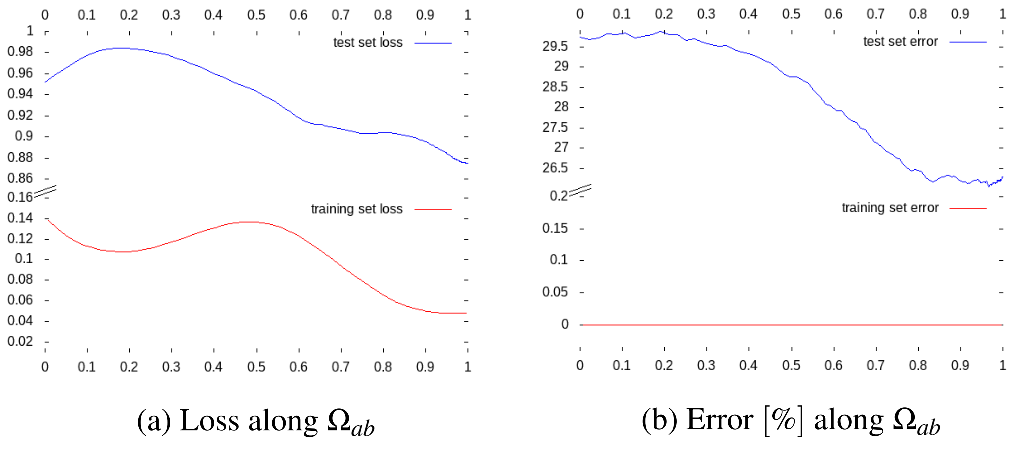


Fig. 3.9 Typical Hessian eigenvalues scenario along $\Gamma(t)$. While the error still remains clamped to zero, the non-zero eigenvalues are all positive and just a very small percentage of them are not zero. In this case, for $N = 300$, $K = 3$, just three of them are non-zero values.

of the model, also in this case it has always been possible to find a Ω_{ab} path in S. The solutions for the training problem were obtained using SGD with learning rate 0.1 with mini-batches of size 100 and 1000 with different initialization seeds (using the Xavier initializer). A typical observed behavior is shown in Fig. 3.10. Working



(a) Loss along Ω_{ab} (b) Error $[\%]$ along Ω_{ab}

Fig. 3.10 Example of Ω_{ab} for LeNet-5 with a training set of 100 images. The x axis is a normalized distance between W_a and W_b.

on LeNet-5 with the MNIST dataset, it is possible to evaluate the behavior of the generalization error (or, in other words, the error on the test set) along Ω_{ab}. It is here interesting to observe that in general, moving through Ω, both the training and test loss have not a monotonic or bitonic behavior but a more complex behavior has typically been observed (an example is in Fig. 3.10(a)). Furthermore, observing the test set error, it shows a similar behavior to the test set loss, but not locally exactly the same(Fig. 3.10(b)).

3.4 Deep binary models

In the community, it is known that state-of-the-art deep neural networks suffer of over-parametrization, and one common guess is that, along the training, signals suffer from high redundancy. We have observed that all the typical learning strategies, because of this, are as well prone to over-fit data, and a possible solution to this problem is to introduce some extra learning constraints (regularization). In order to prevent this, a possible strategy is to *prune* all the redundant parameters in the network: in such a way, the architecture of the deep model is expected to become much simpler ideally at the same performance. Such a possibility will be explored in Chapter 4, which is entirely devolved to sparse networks.
Another similar approach is to reduce the quantity of *bits* necessary to represent a parameter of the network. Since now, whenever he have talked about real numerical parameters, we assumed to have a 32 bits floating point representation for each of the parameters. Do we really need such a precision?
Recent works suggests that few bits are necessary to successfully learn an artificial neural network model [49] and this opens the doors to bit-precision minimizing techniques (or, in other words, *quantization*). Many examples can be provided around this topic, but the most fascinating and interesting goal is to have full binary deep neural networks. Nowadays, a typical ANN model, in order to be simulated, typically requires a CPU (or, in any case, a powerful processing unit) because of the operations between non-binary variables. However, all the electronics is still based on binary logic: having deep models which run with binary variables only would allow the design of extremely efficient electronics devices with huge advantages from many sides. However, solving problems with binary variables, as we have observed in Chapter 2, is far from being trivial.

Nowadays, the state-of-the-art top-performance approach to train deep binary model is *BinaryNet* by Courbariaux et al. [63]. It certainly shows relevance for the performance achieved; however it uses an ensemble of techniques like:

- train real value parameters and then binarize them: the entire work trains real-value parameters, whose sign is extracted during forward propagation. This allows the entire framework to ideally run with back-propagation algorithms.
- different sign function derivative: as it is known, the derivative of the sign function (which is the neuron activation for BinaryNet) is a dirac delta; hence, back-propagation should not fit sign activations and typically some relaxation should be required. In BinaryNet, it is assumed an *hard hyperbolic tangent* as derivative of the sign activation.
- Adam optimizer is used.
- batch normalization layers are used.
- dropout is used.

Sadly, it is really difficult to try explaining a model which makes use of so many heuristics (even if it is the state-of-the-art). In the next section an extension of the stochastic perceptron model for deep architectures is shown. Differently from BinaryNet, the use of heuristics is limited and the model is grounded.

3.5 Stochastic deep networks

As we have seen in Sec. 2.4, a new learning rule for training the binary perceptron was proposed. According to the empirical observations, it resulted being effective to solve the problem exploiting the concept of having stochastic synapses.
Those encouraging results suggested an extension to more complex architectures. Let us consider a binary multi-layer perceptron model

$$\tau^l = \text{sign}\left(\mathbf{W}^l \cdot \tau^l\right) \tag{3.18}$$

where $\mathbf{W}^l$ is a binary weights configuration, $l \in [1;L]$ and L is the number of learnable layers. Of course here $\tau^0 = \xi^\mu$ and $\tau^L = y^\mu$. Optionally, it is possible to include the presence of biases in the model

$$\tau^l = \text{sign}\left(\mathbf{W}^l \cdot \tau^l + \mathbf{b}^l\right) \tag{3.19}$$

where $\mathbf{b}^l$ is a vector of continuous variables. According to the plant presented for the stochastic perceptron, $\mathbf{W}^l$ are made of independent random variables w_i^l, having mean m_i^l and variance $(1-m_i^l)$ (such an approach has successfully already been applied [64]). In such a framework, the output of the l-th layer τ^l represents a sampled *trajectory* for our system, which is a random variable itself. Let us keep it fixed for a moment (or better, let us take into account one sample for it only).
We could here make the same factorized approximation as made for the stochastic perceptron in eq. 2.49, but analyzing it layer-by layer as done in eq. 3.19. Element-by-element it becomes:

$$a_i^l = 2\text{H}\left(-\frac{\sum_j m_{ij}^{l-1} a_j^{l-1} + b_i^{l-1}}{\sum_j 1-(m_{ij}^l)^2 (a_j^l)^2}\right) - 1 \tag{3.20}$$

where a_i^{l-1} is the mean of the pre-activation of the $l-1$-th layer and which is $a_i^{l-1} \in [-1;+1] \forall i,l$. With this framework we are ready to use forward propagation steps and to train MLP on a real-case problem. As loss function we could here use a cross-entropy function with the insertion of a soft-max layer. An example of simulation results is shown in Tab. 3.1: a number of heuristics have been simulated for a 3 hidden fully-connected stochastic architecture solving the MNIST dataset. We have attempted to use the dropout heuristics and the Adam optimizer. We see that the best performance is obtained using Adam + applying dropout to the input (which can be considered as input pre-processing). Beyond this point, we noticed that using a standard ReLU-activated network in place of our sign activation and real value synaptic couplings, the final performance using the same heuristics is around 1.3% of error in the best case. This evidences the benefits of our technique, which is marginally deteriorating the performance of the network, even if we are using binary synaptic couplings and activation functions.

Table 3.1 Confrontation of the results obtained for a 801x801x801x10 fully-connected stochastic architecture on MNIST. The learning rate here is fixed to 0.01 and the batch size is 1000. All the results are averaged on 10 seeds.

Method	input dropout	dropout	binary error[%]
SGD	0	0	1.82 ± 0.0002
SGD	0.2	0	$1.6 \pm 5 \cdot 10^{-5}$
SGD	0.2	0.3	$1.5 \pm 1 \cdot 10^{-4}$
Adam	0.2	0	$1.47 \pm 1 \cdot 10^{-4}$

3.5.1 Extension to convolutional layers

The above derivation was computed for multi-layer perceptron models. However, state-of-the-art models typically include convolutional layers. How our model translates to convolutional layers? Let us take a toy model in which we have an input $\tau^{\mathbf{0}} \in \mathbb{R}^{3\times 3}$ and a filter $\mathbf{W} \in \mathbb{R}^{2\times 2}$ having associated the averages $\mathbf{m} \in \mathbb{R}^{2\times 2}$. Remembering that we are working with random variables having mean m_i and variance $1 - m_i^2$ we will have as mean

$$E_{1,1} = m_{1,1}\tau_{1,1} + m_{1,2}\tau_{1,2} + m_{2,1}\tau_{2,1} + m_{2,2}\tau_{2,2} \tag{3.21}$$

and under the assumption of stochastically-independent variables, variance

$$V_{1,1} = (1 - m_{1,1}^2)\tau_{1,1} + (1 - m_{1,2}^2)\tau_{1,2} + (1 - m_{2,1}^2)\tau_{2,1} + (1 - m_{2,2}^2)\tau_{2,2} \tag{3.22}$$

and so on for the other terms from the convolution. As we have observed, the operations described in eq. 3.21 and eq. 3.22 are essentially convolutions in which we have as variable the means in eq. 3.21 and the variances in eq. 3.22.
Here we might attempt to apply the usual gaussian approximation having in the end

$$\tau_{1,1} = 2\mathrm{H}\left(\frac{E_{1,1}}{V_{1,1}}\right) - 1 \tag{3.23}$$

However, in convolutional layers there always is the insertion of a pooling layer. Of course, for our model the use of max-pool does not make much sense as we are propagating means and variances, while the average pool is definitely applicable. In such a frame, the sequence of forward propagation steps to be performed are:

- convolutional layer
- pooling layer
- activation

Our results are not very satisfactory: while our technique is able to reach about 0.78% of accuracy for the binary configuration, other proposed heuristics, like BinaryNet [63], are able to achieve performances in the order of 0.65%, and such a difference is more evidence scaling up to more complex architectures and more difficult training sets. We think the problem with convolutional layers is that the gaussian approximation no longer holds (each filter has a very limited number of synapses in these models, for example, in LeNet-5 each neuron in the first layer has 25 synaptic couplings only), making the empirical results sub-optimal.

3.5.2 Sampling the trajectories

Another possibility is here to sample the trajectories. Here, we are no longer able to perform a usual full back-propagation because, as output for each layer, we will sample a given number of trajectories according to the probability distributions derived from the pre-activation values *a* as computed in eq. 3.20. Hence, for the forward step, the forward steps to be followed in each layer will be:

1. Perform the usual forward-propagation step till the *a* value is computed
2. According to the value of *a* acting as probability of having a positive sign, extract the trajectories

Once arrived at the output layer, we will have a given number of sampled trajectories. Here we are supposed to weight each of the sampled trajectories according to the exactness of the final obtained results. Towards this end, we compute

$$\rho^{\mu,\zeta} = \frac{\prod_i \mathrm{H}(y_i^\mu \wp_{L,i}^{\mu,\zeta})}{\sum_s \prod_i \mathrm{H}(y^\mu \wp_{L,i}^{\mu,s})} \tag{3.24}$$

Table 3.2 Confrontation of the results obtained for a 201x201x10 fully-connected stochastic architecture on a reduced MNIST (10000 samples)

Method	learning rate	Batch size	MC iters	dropout	binary error[%]
SGD	50	100	N/A	0	6.02 ± 0.21
SGD	10	100	N/A	0.5	5.04 ± 0.21
Adam	0.01	50	N/A	0	4.98 ± 0.28
MC	0.1	1000	80	0	4.9 ± 0.09

where

$$\wp^{\mu,s}_{L,i} = \frac{\sum_j m^L_{ij}\tau^{L-1,s}_j + b^L_i}{\sum_j 1-(m^L_{ij})^2(\tau^{L-1,s}_j)^2} \tag{3.25}$$

which is the product of all the output error components (where the energy used is the H function).

The results here obtained with small architectures are quite encouraging and promising (Tab. 3.2). We are here comparing SGD-based approach with or without dropout, Adam-based optimization and Montecarlo sampling (MC) on a reduced training set from MNIST. We see that MC performance, on the average of 10 different training seeds, is the best and is also the more robust having the least variance of all the other techniques.

Chapter 4

Sparse networks

Nowadays, the whole deep learning community is hunger of solving more and more complex tasks. From easier classification tasks like MNIST [65], the community toggled to more complex tasks like CIFAR-10, arriving to the challenging ImageNet: a database of 1.4M of images divided in 1000 classes. In order to solve these ever complexity-increasing problems, deep learning researchers decided to dramatically increase the size of the networks, from the few thousands parameters of the earlier models to hundreds of millions or even billions of parameters for the most recent architectures.
We have several ways to measure the *complexity* of an artificial neural network: for example, we can measure it as the number of parameters (weights and biases) the network can learn. We have already seen in Sec. 3.1.3 the difference between a fully-connected and a convolutional layer: the first typically has more parameters than the second (because convolutional neural networks have shared weights); however, talking in terms of computational complexity, convolutional layers are more expensive than fully-connected (because of the convolution). Furthermore, the theory for smaller ANNs suggested that an optimal architecture for solving a particular problem Ξ exists, and might boost the final performance in terms of generalization [66] [67]. Because of this, many regularization techniques are required to prevent over-fitting. For these reasons, recently there has been a huge race for introducing *sparsity* in artificial neural network models to reduce the total number of parameters required, still holding a good generalization performance. In this chapter we first introduce state-of-the-art sparsification techniques in Sec. 4.1. Then, starting from Sec. 4.2, we present our sensitivity-based regularization, a novel approach to introduce sparsity

without affecting the performance in deep neural networks. It has been inspired by the biological principle of pruning all the less-used neural connections. Empirical results show that such a technique performs better than state-of-the-art heuristics for fully-connected architectures, and shows promising results also on convolutional ones [68].

4.1 Techniques to sparsify

As we previously introduced in Sec. 1.4.1, all the continuous networks problems are solved using gradient descent-based techniques evaluating a given objective function J, typically shaped as the sum of two contributions:

- loss function: this term is responsible for learning the training set, and is proportional to the number of errors the neural network is performing.
- regularization function: it is responsible for introducing some extra constraints to the learning problem, typically aiming to prevent over-fitting.

J can be successfully, headless minimized assuming:

- all the parameters are real values.
- all the activation functions used in our deep model are differentiable.
- both loss function and regularizer term are differentiable functions in the work domain.
- the hyper-parameters scaling loss function and regularizer term are properly set.
- no loops are introduced in the deep model (in order to successfully use back-propagation).
- the input is a bounded quantity.

All of these conditions are typically satisfied in a standard deep learning problem, apart from properly setting the hyper-parameters. In particular, if we write the objective function as

$$J = \eta L + \lambda R \tag{4.1}$$

even if we assume to use the vanilla stochastic gradient descent (SGD, see eq. 1.17), giving an exact, a-priori estimation for the optimal η and λ is impossible, as it depends on factors like

- training set.
- artificial neural network model (connectivity, size, type of layers).
- initialization for the parameters.

For this, typically the use of a validation set besides the training set to tune the hyper-parameters is required. Thanks to this empirical approach, it was observed that in the early stages of training, the dominant term in the objective function should be the loss. According to this observation, typically $\eta \gg \lambda$. It was further observed that, for deep models, after some initial epochs, for better generalization performance, the *learning rate* η should be decayed. This is known as *learning rate scheduling* and typically acts independently from the standard regularization techniques (like dropout and weight decay). Some of the regularization techniques we are going to show here as a side effect slightly deteriorates the generalization performance and require the ANN model to be pre-trained using a different objective function.

4.1.1 Lasso regularization

Also known as least absolute shrinkage and selection operator (or L1 regularization), this is probably the most ancient techniques used to introduce sparsity in a trained model. It is a special case of eq. 1.13, reading

$$R(\mathbf{w}) = \sum_i |w_i| \tag{4.2}$$

Its use is extremely broad: from the optimization of any regression model to the geophysics. Unlike weight decay, it is not necessarily improving the generalization for a training model, as it can definitely worsen it. However, it performs extremely well in problems for which there are very few co-variate parameters. Sparsification via L1 regularization has been used also to attempt explaining the relevant structure of a deep model, unsuccessfully.
However, most of the main deep learning frameworks include it as a standard

regularizer. The reason for it is very simple: computational efficiency. Given the parameter w_i, the update for just the regularization term reduces to

$$w_i := w_i - \lambda \text{sign}(w_i) \tag{4.3}$$

Hence, R does not even need to be back-propagated. We quote here that the optimal best R for sparsification should be the chimeric $L0$ regularizer

$$R(\mathbf{w}) = \sum_i |w_i|^0 \tag{4.4}$$

Essentially, eq. 4.4 counts how many parameters are left in the architecture: minimizing it results in introducing sparsity. Sadly, it is a non-differentiable function, and can not be used for back-propagation. Many attempts to relax such a regularizer have been done, some in a general way, and others focusing on convolutional layers; however, it is in general acknowledged lasso regularization to be a fair approximation.

4.1.2 Variational dropout

We have already introduced the dropout technique [27] aiming to boost generalization in artificial neural networks in Sec. 1.5.5. Also named as Bernoulli or Binary dropout, it consists in stochastically pruning some neurons during the forward propagation phase during the training phase. In a successive work, it was further observed that having a gaussian dropout for the parameters works as well [28]. Such a result is extremely important: multiplying some inputs by a gaussian noise is equivalent to have a gaussian noise on the parameters themselves. From this observation, it comes possible to use the so-called *re-parametrization trick* [69] to have a suitable model for forward propagation. In particular, the observed, sampled parameter θ_i of the network is

$$\theta_i = w_i(1 + \sqrt{\alpha}\varepsilon) \tag{4.5}$$

where ε is a gaussian distribution with mean 0 and unitary standard deviation and α is in general the hyper-parameter tuning the dropout probability. Here, α was earlier introduced as an hyper-parameter, modeling the dropout probability; however, in their approach, they consider α a per-parameter variable, tuned using a variational approach (hence, at that point, it indicates whether a parameter is included or not in the network). It was observed that variational dropout boosts sparsification while still holding good generalization performances.

4.2 A biological inspiration

Ideally, it would be nice being able to combine both the nice sparsity property of the lasso regularizer (and, maybe, boost it) and all the generalization benefits coming from L2 regularization. Several works attempted to tackle such a problem from a mathematical point of view but no one gave an exhaustive reply to such a problem. Let us think for a moment biologically, borrowing some useful knowledge from the world of unsupervised learning theory.

« Let us assume that the persistence or repetition of a reverberatory activity (or "trace") tends to induce lasting cellular changes that add to its stability.[. . .] When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased» .

This is a famous quote, being the base of the so-called *Hebb theory*, by Donald Hebb, who wrote in 1949 a book named *The Organization of Behavior*, a milestone in the neural network learning, theory and formulation [70]. Such an idea is used for associative memories and, for this reason, out of our scope. However, it is not necessarily true that a valid concept can not be used successfully inspire solutions to similar problems, even out of the original scope: we have also seen that it was an inspiration for the perceptron learning rule (see Sec. 2.2.3).
Assume we have a given neural network and that we associate a performance estimator A_i we name *activity* to each of the parameters of the neural network, defined as

$$A_i = \left\langle |\xi_i^\mu| \right\rangle_\mu |w_i| \tag{4.6}$$

Such measure indicates how much a given connection is "used" during the forward propagation step: if the average of the input $\left\langle |\xi_i^\mu| \right\rangle_\Xi$ is low, then the value of the weight w_i can be penalized as the activity of the connection is very low (Fig. 4.1). The same happens if the w_i itself is low: we are implicitly assuming a loss function L is pushing the neural network towards configurations which solve the given classification task and if a given weight remains or is low then it might be a good "guess"

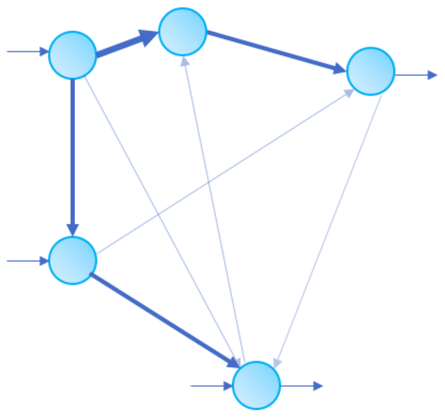


Fig. 4.1 A very intuitive diagram to give a high-level idea of Hebb's claim. Neighbor neurons typically influences each other, and their connection is typically stronger than far ones. As in ANNs there is no spatial distance concept between neurons, we can still evaluate the "usage" of a connection.

it will not affect the final outcome of the network. However, such a statement is not in general correct as we have no global knowledge of how a parameter is influencing the final outcome of the network...
From such an observation, it comes natural to think on the design of a penalty term for any weight w_i of the network depending on how a parameter effectively influences the output.

4.3 Parameter sensitivity

In the previous section we have explored the possibility of having a local estimator for the activity of each parameter in a deep model. However, we were not entirely convinced on the effective use of it in order to regularize the network: in order to have a better estimator on the effectiveness of the possible pruning of a parameter, we should have at first an estimate on how much a perturbation for a given parameter would affect the outcome for the trained ANN.
Hence, what we are asking here is to check the perturbation of the k-th neural network's output Δy_k for a corresponding perturbation of the tested parameter w_i

$$\Delta y_k \approx \Delta w_i \frac{\partial y_k}{\partial w_i} \tag{4.7}$$

Here, recalling that an ANN model has C different outputs, we can sum over all the contributions

$$\sum_{k=1}^{C} \alpha_k \left|\Delta y_k\right| = \left|\Delta w_i\right| \sum_{k=1}^{C} \alpha_k \left|\frac{\partial y_k}{\partial w_i}\right| \tag{4.8}$$

where $\alpha_k > 0$ is a scale factor. From eq. 4.8, we define *sensitivity* of the artificial neural network output with respect to the i-th parameter as

$$S(\mathbf{y}, w_i) = \sum_{k=1}^{C} \alpha_k \left|\frac{\partial y_k}{\partial w_i}\right|, \tag{4.9}$$

The sensitivity plays a key role in the evaluation of the *importance* of the examined parameter in the generation of the current outcome. Of course, it is a *local* measure: it both assumes all the other parameters being constant and (eventually) it is evaluated on a particular minibatch (hence, it is an averaged quantity).
However, it still provides some useful information aiming to sparsify the network: if the sensitivity is small, then a small change of such a parameter, to be driven towards zero, results in a very small perturbation for the output of the network. In such a sense, it can be seen like a confidence estimator: the higher it is, the less we are supposed to modify it. As it is possible to imagine, sensitivity can be computed using the standard back-propagation approach; hence, it is a proper suit for deep models. However, we would like to have an estimator of how confident we are in *modifying* the parameter without introducing perturbations in the system: towards this end, we define the *insensitivity* function $\bar{S}$

$$\bar{S}(\mathbf{y}, w_i) = 1 - S(\mathbf{y}, w_i) \tag{4.10}$$

whose range is $(-\infty; 1]$. Having $\bar{S} < 0 \Leftrightarrow S > 1$ results in an atypical condition: the derivative for a particular parameter is higher than one. In such a case we like to call output *super-sensitive* to such a parameter.
In our context we are definitely not interested in sparsifying those parameters, in particular because they result being extremely meaningful to the generation of the output. We aim to focus on the parameters whose output is *sub-sensitive* to, or in other words, for which $\sum_k \alpha_k |\Delta y_k| < \Delta w$.
So, here we define a bounded insensitivity

$$\bar{S}_b(\mathbf{y}, w_i) = \max\left[0, \bar{S}(\mathbf{y}, w_i)\right] \tag{4.11}$$

having $\bar{S}_b \in [0,1]$.
A parameter showing small sensitivity (or high bounded insensitivity) may be confidently pushed towards zero. We can accomplish this subtracting to it the product between the same parameter and its insensitivity, properly scaled by some hyperparameter λ.
We can decide to perform such an operation together with the optimization for the loss function (in our case, we decided to use vanilla SGD). At the t-th update iteration, the i-th weight update rule will be

$$w_i^t := w_i^{t-1} - \eta \frac{\partial L}{\partial w_i^{t-1}} - \lambda w_i^{t-1} \bar{S}_b(\mathbf{y}, w_i^{t-1}) \tag{4.12}$$

According to the chain rule, we can factorize the derivative of the loss function as

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w_i} \tag{4.13}$$

which is a scalar product between the derivative of the loss L to the output elements $\mathbf{y}$ and the derivative of $\mathbf{y}$ to the parameter.
By the Hölder inequality, we have that

$$\left| \frac{\partial L}{\partial w_i} \right| \leq \max_k \left| \frac{\partial L}{\partial y_k} \right| \left\| \frac{\partial \mathbf{y}}{\partial w_i} \right\|_1 . \tag{4.14}$$

If we assume the loss function L being the multi-class cross-entropy + the softmax function, we know that

$$\max_k \left| \frac{\partial L}{\partial y_k} \right| \leq 1 \tag{4.15}$$

having in the end

$$\left| \frac{\partial L}{\partial w_i} \right| \leq \left\| \frac{\partial \mathbf{y}}{\partial w_i} \right\|_1 . \tag{4.16}$$

We emphasize that the term $\left\| \frac{\partial \mathbf{y}}{\partial w_i} \right\|_1$ is equivalent to the sensitivity assuming $\alpha_k = 1 \forall k$.
If the derivative with respect to the loss is large, then by eq. 4.16 also the sensitivity must be large. A large sensitivity results in a small (or even null) bounded insensitivity. Hence, a large gradient from the loss function implies a small regularization correction term. This typically happens in the early stages of training, when the loss contribution should be more relevant than the regularization term, or in other words,

whenever $\frac{\partial L}{\partial w_i}$ is large. On the other hand, whenever the loss function is close to a minimum, then the contribution coming from the loss is minimal, but the sensitivity might be large or small, according to the ANN and the training set.

4.3.1 Cost function formulation

The update rule showed in eq. 4.23 provides some extra constraints to the learning problem, typical of regularization methods.
Since eq. 4.23 specifies how a parameter is updated through the *derivative* of R, we can integrate the update term to get what is the regularization term we aim to minimize.
Towards this end, we define the overall regularization term as a sum over all parameters

$$R(\theta) = \sum_i R_i(w_i) \tag{4.17}$$

and integrate each term over w_i

$$R_i(w_i) = \int w_i \bar{S}_b(\mathbf{y}, w_i) dw_i. \tag{4.18}$$

In this derivation we are dropping the index i for convenience. First, we make explicit $\bar{S}_b$

$$\begin{aligned} R &= \int w \Theta\left[\bar{S}(\mathbf{y}, w)\right] \left[1 - \sum_{k=1}^{C} \alpha_k \frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right)\right] dw \\ &= \Theta\left[\bar{S}(\mathbf{y}, w)\right] \int \left[w - w \sum_{k=1}^{C} \alpha_k \frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right)\right] dw \\ &= \Theta\left[\bar{S}(\mathbf{y}, w)\right] \left[\frac{w^2}{2} - \int w \sum_{k=1}^{C} \alpha_k \frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) dw\right] \end{aligned}$$

As we are here working with positive quantities ($\bar{S}_b$), we can use Fubini-Tonelli's theorem

$$R = \Theta\left[\bar{S}(\mathbf{y}, w)\right] \left[\frac{w^2}{2} - \sum_{k=1}^{C} \alpha_k \int w \frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) dw\right] \tag{4.19}$$

Now the part of eq. 4.19 to be solved is

$$R_1 = \int w \frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) dw$$

Let us integrate by parts:

$$R_1 = \frac{w^2}{2}\frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) - \int \frac{w^2}{2}\frac{\partial^2 y_k}{\partial w^2} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) dw$$

If we apply another step:

$$R_1 = \frac{w^2}{2}\frac{\partial y_k}{\partial w} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) - \frac{w^3}{6}\frac{\partial^2 y_k}{\partial w^2} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) + \int \frac{w^3}{6}\frac{\partial^3 y_k}{\partial w^3} \text{sign}\left(\frac{\partial y_k}{\partial w}\right) dw$$

Applying infinite steps of integration by parts we have in the end

$$R_1 = \text{sign}\left(\frac{\partial y_k}{\partial w}\right) \sum_{m=1}^{\infty} -1^{m+1} \frac{w^{m+1}}{(m+1)!}\frac{\partial^m y_k}{\partial w^m} \tag{4.20}$$

Merging eq. 4.20 to eq. 4.19 we get the final R formulation

$$R_i(w_{n,i}) = \Theta\left[\bar{S}(\mathbf{y}, w_i)\right] \frac{w_i^2}{2} \cdot \left[1 - \sum_{k=1}^{C} \alpha_k \text{sign}\left(\frac{\partial y_k}{\partial w_i}\right) \sum_{m=1}^{\infty} -1^{m+1} \frac{w^{m-1}}{(m+1)!}\frac{\partial^m y_k}{\partial w_i^m}\right] \tag{4.21}$$

Notice that eq. 4.21 is absolutely general and is valid for any kind of network and activation.
Now, let us suppose that all activation functions are rectified linear units (ReLU). Their derivative is the step function; the higher order derivatives are therefore zero. This results in dropping all the $m > 1$ terms in eq. 4.21. Thus, the regularization term for ReLU-activated networks reduces to

$$R_i(w_i) = \frac{w_i^2}{2}\bar{S}(\mathbf{y}, w_i) \tag{4.22}$$

The first factor in this expression is the square of the weight, showing the relation to L2 regularization (see eq. 1.12).
The other factor is a selection and damping mechanism. Only the sub-sensitive weights are influenced by the regularization in proportion to their insensitivity.

4.4 Sensitivity driven regularization effect to the learning dynamics

In this section we are going to discuss the impact of the update term to the learning dynamics of a generic neural network. We recall that, in our work, the relative impact of the correction term in the update rule (dropping index *i* for convenience)

$$w^t := w^{t-1} - \eta \frac{\partial L}{\partial w^{t-1}} - \lambda w^{t-1} \bar{S_b}(\mathbf{y}, w^{t-1}) \tag{4.23}$$

is determined by the magnitude of the insensitivity.

4.4.1 Loss term vs Regularization term

From eq. 4.23 we would like here to analyze the case for which

$$\eta \left| \frac{\partial L}{\partial w} \right| > \lambda \left| w \bar{S_b}(\mathbf{y}, w) \right|$$

If we assume we have no *super-sensitive* parameters ($S(\mathbf{y}, w) \leq 1 \Leftrightarrow \bar{S}(\mathbf{y}, w) \geq 0$), $\bar{S_b} = \bar{S}$, we can write

$$\eta \left| \frac{\partial L}{\partial w} \right| > \lambda \left| w \right| \bar{S}(\mathbf{y}, w) \tag{4.24}$$

Making $\bar{S}$ explicit, we have

$$\eta \left| \frac{\partial L}{\partial w} \right| > \lambda |w| \left[1 - \sum_{k=1}^{C} \alpha_k \left| \frac{\partial y_k}{\partial w} \right| \right] \tag{4.25}$$

If we assume $\alpha_k \leq 1 \forall k$ we can write

$$\sum_{k=1}^{C} \alpha_k \left| \frac{\partial y_k}{\partial w} \right| \leq \sum_{k=1}^{C} \left| \frac{\partial y_k}{\partial w} \right| = \left\| \frac{\partial \mathbf{y}}{\partial w} \right\|_1 \Leftrightarrow \left[1 - \sum_{k=1}^{C} \alpha_k \left| \frac{\partial y_k}{\partial w} \right| \right] \geq 1 - \left\| \frac{\partial \mathbf{y}}{\partial w} \right\|_1$$

Hence, we can write eq. 4.25 as

$$\eta \left| \frac{\partial L}{\partial w} \right| > \lambda |w| \left[1 - \sum_{k=1}^{C} \alpha_k \left| \frac{\partial y_k}{\partial w} \right| \right] \geq \lambda |w| \left(1 - \left\| \frac{\partial \mathbf{y}}{\partial w} \right\|_1 \right) \tag{4.26}$$

According to Hölder's inequality, we can write

$$\left|\frac{\partial L}{\partial w}\right| \leq \max_k \left|\frac{\partial L}{\partial y_k}\right| \left\|\frac{\partial \mathbf{y}}{\partial w}\right\|_1 . \tag{4.27}$$

So, we can substitute eq. 4.27 in eq. 4.26, having

$$\max_k \left|\frac{\partial L}{\partial y_k}\right| \left\|\frac{\partial \mathbf{y}}{\partial w}\right\|_1 > \frac{\lambda}{\eta}|w| \left(1 - \left\|\frac{\partial \mathbf{y}}{\partial w}\right\|_1\right) \tag{4.28}$$

which is equivalent, under the assumption $\max_k \left|\frac{\partial L}{\partial y_k}\right| > 0$, to

$$\left\|\frac{\partial \mathbf{y}}{\partial w}\right\|_1 > \frac{\frac{\lambda}{\eta}|w|}{\max_k \left|\frac{\partial L}{\partial y_k}\right| + \frac{\lambda}{\eta}|w|} \tag{4.29}$$

From eq. 4.29 we see that, in the early stages of training, as commonly $\lambda \ll \eta$, it is easy to have $\frac{\frac{\lambda}{\eta}|w|}{\max_k \left|\frac{\partial L}{\partial y_k}\right| + \frac{\lambda}{\eta}|w|} \approx 0$ as the error of the network is high; hence, the regularization term allows the network to learn.

4.4.2 Sensitivity in depth

Now, we are going to discuss our choice of S and what we are expecting from such a parameter. In particular, we are going to investigate which is the expected sensitivity range at each training step.
According to

$$\bar{S_b}(\mathbf{y}, w_i) = \Theta\left[\bar{S}(\mathbf{y}, w_i)\right] \bar{S}\left(\mathbf{y}, w_i\right) \tag{4.30}$$

$\bar{S} \in [0;1]$. However, we are lower-bounding it to be positive. Assuming we are working with the unbound insensitivity

$$\bar{S}(\mathbf{y}, w_i) = 1 - S(\mathbf{y}, w_i) \tag{4.31}$$

we are here interested in investigating the case for which

$$\sum_{k=1}^{C} \alpha_k \left|\frac{\partial y_k}{\partial w_i}\right| > 1$$

Table 4.1 Most commonly used activation functions in deep learning

Activation ($f(x)$)	$\frac{d}{dx}f$	Range ($\frac{d}{dx}f$)
$\mathrm{ReLU}(x)$	$\Theta(x)$	$\{0;1\}$
$\tanh(x)$	$\mathrm{sech}^2(x)$	$(0;1]$
x	1	$\{1\}$
$\frac{e^x}{k+e^x}$	$\frac{ke^x}{(k+e^x)^2}$	$(0;\frac{1}{4}]$

Let us inspect the single k-th output

$$S_k(\mathbf{y}, w_i) = \left|\frac{\partial y_k}{\partial w_i}\right|$$

Its value is computed using the chain rule

$$\frac{\partial y_k}{\partial w_i} = \frac{\partial x_{N,k}}{\partial w_i} = \frac{\partial x_{N,k}}{\partial \mathbf{x}_{N-1}} \prod_{j=n+1}^{N-1} \left(\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}_{j-1}}\right) \frac{\partial \mathbf{x}_n}{\partial w_i} \tag{4.32}$$

We know that

$$\frac{\partial \mathbf{x}_j}{\partial \mathbf{x}_{j-1}} = f'(\mathbf{x}_j)B(\mathbf{w}_j) \tag{4.33}$$

where $B(\mathbf{w}_j)$ is the linear part of the function g_j.[1] For convenience, we report some commonly used activation functions in deep learning in Table 4.1. In such a framework, $0 \leq f'(\mathbf{x}_j) \leq 1 \forall j$. This is a very relevant observation: $\bar{S}$ value for parameters in layer n are not amplified by activation functions, but they strongly depend on the values of the parameters in layers $j > n$.
In deep learning it is very common to have weight distributions with almost zero-mean and standard deviation being very small (this depends on the network: the larger it is, the smaller the deviation is). According to eq. 4.32 and to eq. 4.33, not only it is extremely unlikely that $\bar{S} < 0$ (and in such a case, $\mathbf{y}$ is *super-sensitive* to w), but most of the parameters of the network have very low sensitivity. For example, if we take a LeNet-300 trained on MNIST we can see that the sensitivity (computed without boundaries) for the network trained, with and without our regularizer, is enclosed in $[0,1]$ (Fig. 4.2). Notice that for the network trained without regularizer (no pruning), having low sensitivity doesn't necessarily mean we can prune all those parameters. In order to show this more evidently, we refer to Fig. 4.3 in which

[1] $\mathbf{x}_n = f_n\left[g_n\left(\mathbf{x}_{n-1}, \mathbf{w}_n\right)\right]$

we plot $\max\{|w|, S(w)\}$ which is a good estimator of how many parameters we are confident to prune.

4.5 Thresholding

In this section we are going to analyze the cases for which a parameter will be pruned from our network. In particular, We will make a distinction between applying or not applying a threshold T.

4.5.1 T = 0

According to eq. 4.23, we will have a parameter exactly set to zero if

$$w_i - \eta \frac{\partial L}{\partial w_i} - \lambda w_i \bar{S_b}(\mathbf{y}, w_i) = 0 \tag{4.34}$$

Let us investigate some cases.

S = 0

Having $S = 0$ is equivalent to say $\bar{S_b} = 1$. For this reason, eq. 4.34 becomes

$$w_i - \eta \frac{\partial L}{\partial w_i} - \lambda w_i = 0$$

However, from Hölder's inequality (eq. 4.27), we also know that $\frac{\partial L}{\partial w_i} = 0$, reading

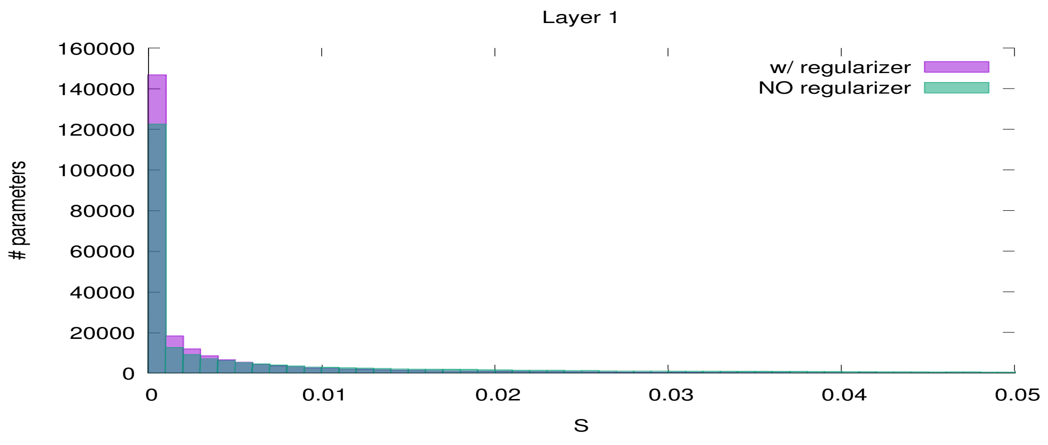
$$w_i - \lambda w_i = 0$$

The only solution is here to have $\lambda = 1$, which however is not a common choice.

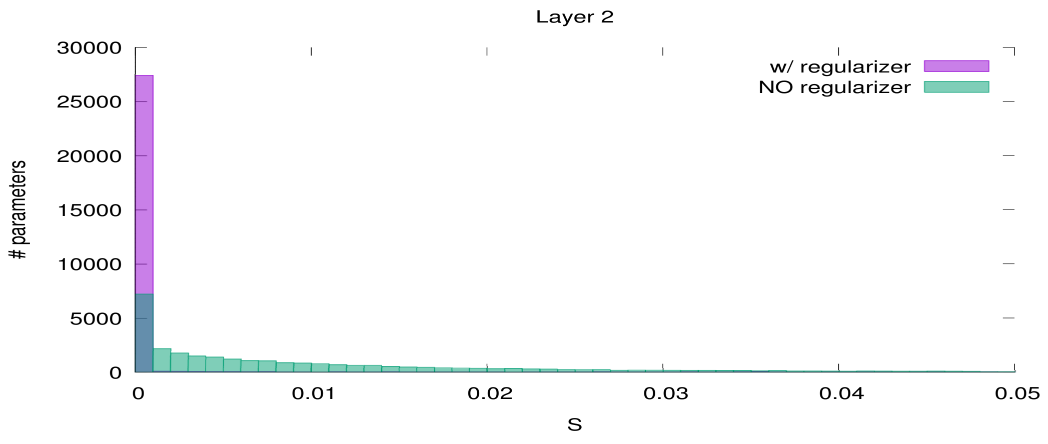
S ≥ 1

In this case we have $\bar{S_b} = 0$. Hence, the condition is

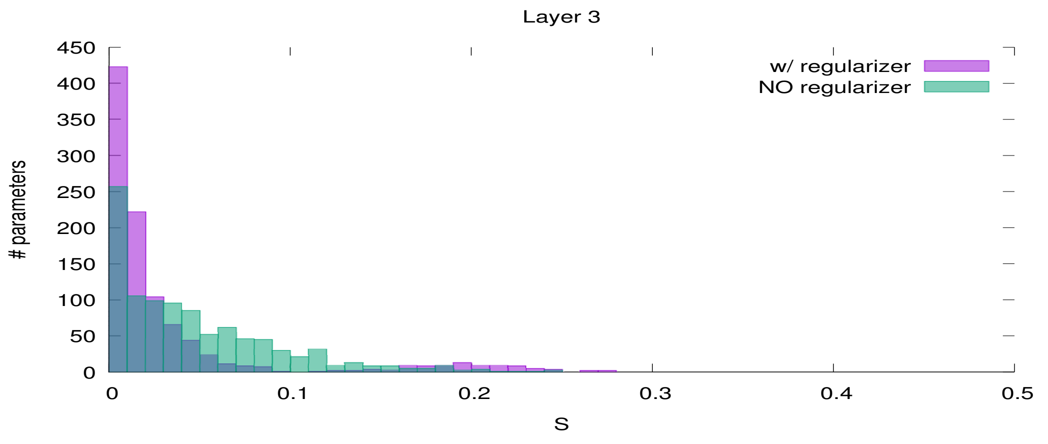
$$w_i - \eta \frac{\partial L}{\partial w_i} = 0$$



(a)

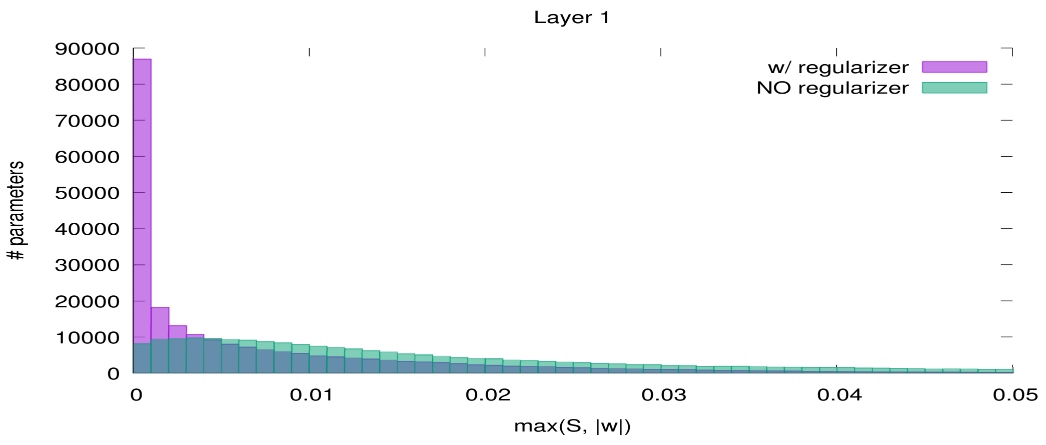


(b)

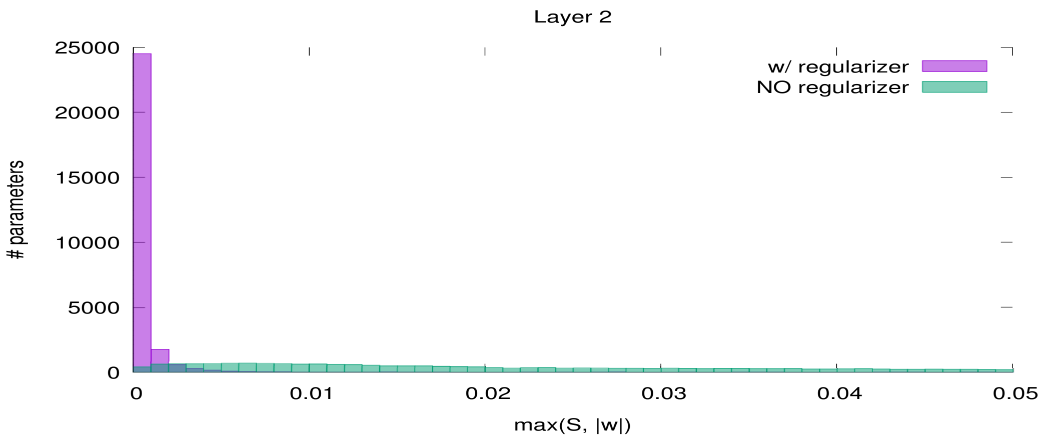


(c)

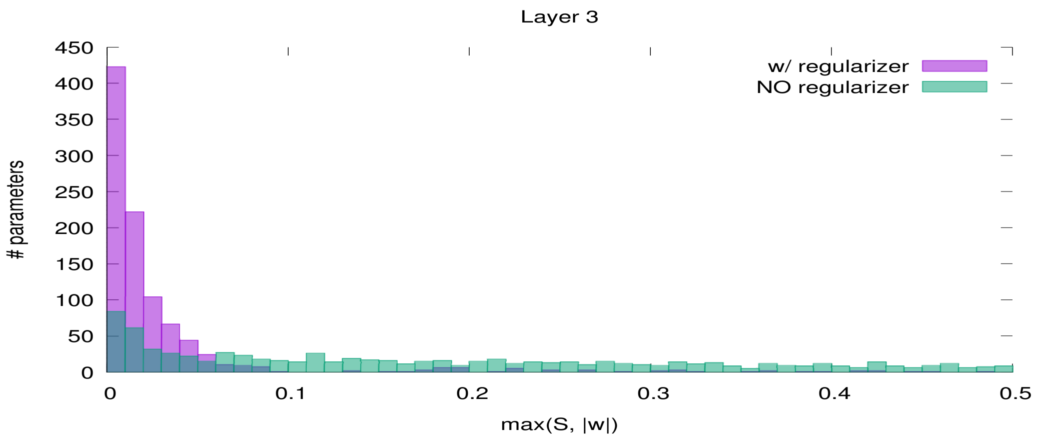
Fig. 4.2 Sensitivity computed in LeNet-300 trained on MNIST with and without our regularizer. The layer size is 784x300 (Fig. 4.2a), 300x100 (Fig. 4.2b) and 100x10 (Fig. 4.2c). The effect of the regularizer is to increase the number of parameters having low sensitivity. However, training without regularizer also results in having a significant slice of low-sensitivity parameters. As we will see in Fig. 4.3, most of the low-sensitivity parameters trained without regularization are not close to zero.



(a)



(b)



(c)

Fig. 4.3 Maximum between S and $|w|$ in LeNet-300 trained on MNIST with and without our regularizer. Here we are representing the same parameters and the same setting as Fig. 4.2. The lowest this value is, the most we are confident to prune these parameters, It is evident that using our regularizer the behavior with Fig. 4.2 is very close, meaning that we are effectively pushing towards zero the less relevant parameters. On the contrary, without regularizer we are far from being able to perform some efficient pruning.

or, in other words, it collapses to standard SGD and the parameter will be pruned if the loss term pushes it towards zero.

0 < S < 1

This is the most general case. Here, eq. 4.34 reads

$$w_i = \frac{\eta}{1-\lambda \bar{S_b}(\mathbf{y}, w_i)} \frac{\partial L}{\partial w_i}$$

which states we need an exact ratio between gradient loss and sensitivity to make w_i exactly zero (and this ratio depends also on the learning rate η and on λ). Hence, it appears natural to fix a threshold below which w_i is considered pruned

4.5.2 T > 0

Here we are going to introduce our threshold. In a general way, we say we are pruning a given parameter $w_{n,i}$ if

$$\left| w_i - \eta \frac{\partial L}{\partial w_i} - \lambda w_i \bar{S_b}(\mathbf{y}, w_i) \right| < T \tag{4.35}$$

Hence, we will prune all those parameters satisfying

$$\frac{\eta \frac{\partial L}{\partial w_i} - T}{1-\lambda \bar{S_b}(\mathbf{y}, w_i)} < w_i < \frac{\eta \frac{\partial L}{\partial w_i} + T}{1-\lambda \bar{S_b}(\mathbf{y}, w_i)} \tag{4.36}$$

A summary of the conditions to be satisfied to prune a parameter (depending on the value of its sensitivity) can be found in Table 4.2.

4.5.3 Types of sensitivity

According to eq. 4.9, we have the degree of freedom in choosing α_k values. In our framework, we have proposed two choices for such a parameter.

Table 4.2 Pruning a parameter: conditions to be satisfied

Sensitivity	Condition on w_i	Additional conditions
$S < 0$	$\nexists$	
$S = 0$	$\lvert w_i \rvert < \frac{T}{(1-\lambda)}$	$\lambda \in [0;1)$
	$\forall w_i$	$\lambda = 1$
	$\lvert w_i \rvert < \frac{T}{(\lambda-1)}$	$\lambda \in (1;+\infty)$
$0 < S < 1$	Eq. 4.36	$\lambda \in [0;1]$; $\eta \in [0;+\infty)$
$S \geq 1$	$\eta \frac{\partial L}{\partial w_i} - T < w_i < \eta \frac{\partial L}{\partial w_i} + T$	$\eta \in [0;+\infty)$

If we assume all of the k outputs having the same "relevance" (all $\alpha_k = \frac{1}{C}$) or better, the same weight for the computation of the sensitivity, we say we are using an *unspecific* formulation for the sensitivity:

$$S^{unspec}(\mathbf{y}, w_i) = \frac{1}{C} \sum_{k=1}^{C} \left| \frac{\partial y_k}{\partial w_i} \right| \tag{4.37}$$

Another choice we might have involves the use of the desired output vector $\mathbf{y}$. For classification tasks, it corresponds to the one-hot vector in which "1" refers to the correct class while "0" to all the others. In this case, we have what we name *specific* sensitivity:

$$S^{spec}(\mathbf{y}, \mathbf{y}, w_i) = \sum_{k=1}^{C} y_k \left| \frac{\partial y_k}{\partial w_i} \right| \tag{4.38}$$

An high-level representation of how sensitivity works is shown in Fig. 4.4.

4.6 Results

In this section we provide some experimental results using the sensitivity-based regularization method on different supervised image classification tasks. For each of the trained networks we also provide a sparsity measure as long as the corresponding memory footprint, assuming for all the conduced experiments, single-precision floating point parameter representation.

The first tests have been entirely conduced on some basic playground: MNIST [71] (60k training images and 10k test images) on two toy architectures: LeNet-300 and

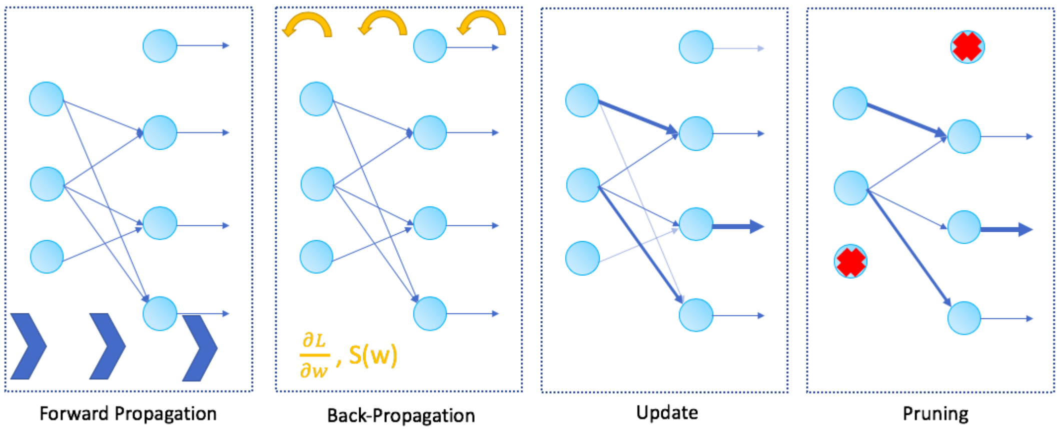


Fig. 4.4 Example of how sensitivity-based regularization works, inside a single iteration. Given a minibatch $\tilde{\Xi}$, first the forward propagation step is performed. Then, through back-propagation, gradient from the loss function and sensitivity are computed. With these, the update step modifies the value of the parameters: some are increased, others are pushed towards zero. Finally, the parameters are thresholded. At some point, entire neurons are no longer connected to the ANN model, and can be entirely pruned away.

LeNet-5.

Intentionally, no particular heuristics has been used for our experiments, in order to isolate the real benefits of the sensitivity-based approach towards other heuristics. For such a reason, vanilla SGD is the optimizer we decided to use. We further emphasize that no other sparsity-promoting method (like dropout or even batch normalization) has been used to produce the presented results.

In table 4.3 we show the results of the experiments over the LeNet-300 network (fully-connected architecture made of 300-100-10 neurons) in two different learning stages: in order to compare our results with some other concurrent results, we allowed some performance degradation. $\frac{|\theta|}{|\theta_{\neq 0}|}$ is the *compression ratio*, namely the ratio between the number of parameters in the original architecture and those remained after the sparsification process (the higher, the better).

The most similar work we compare with belongs to Han et al. [72]: they sparsify using L2 regularization together with other heuristics aiming to introduce sparsity. We consider their results a good reference as our formulation can be interpreted as a reshape of L2 regularization for ReLU-activated networks.

Our method proudly achieves in this setting twice the sparsity of [72] (27.8x vs. 12.2x compression ratio) still showing a comparable error.

In the bottom-half of the same table we still refer to the same network, further trained and allowing errors compared with other state-of-the-art results [73–75]. Even here,

Table 4.3 LeNet-300 network trained over the MNIST dataset

	Remaining parameters				$\frac{\lvert\theta\rvert}{\lvert\theta_{\neq 0}\rvert}$	Top-1
	FC1	FC2	FC3	Total		error
Han *et al.* [72]	8%	**9%**	**26%**	21.76k	12.2x	1.6%
Proposed (S^{unspec})	**2.25%**	11.93%	69.3%	**9.55k**	**27.87x**	1.65%
Proposed (S^{spec})	4.78%	24.75%	73.8%	19.39k	13.73x	**1.56%**
Louizos *et al.* [76]	9.95%	9.68%	33%	26.64k	12.2x	1.8%
SWS[73]	N/A	N/A	N/A	11.19k	23x	1.94%
Sparse VD[74]	1.1%	2.7%	38%	3.71k	68x	1.92%
DNS[75]	1.8%	1.8%	**5.5%**	4.72k	56x	1.99%
Proposed (S^{unspec})	**0.93%**	**1.12%**	5.9%	**2.53k**	**103x**	1.95%
Proposed (S^{spec})	1.12%	1.88%	13.4%	3.26k	80x	1.96%

Table 4.4 LeNet-5 network trained over the MNIST dataset

	Remaining parameters					$\frac{\lvert\theta\rvert}{\lvert\theta_{\neq 0}\rvert}$	Top-1
	Conv1	Conv2	FC1	FC2	Total		error
Han *et al.* [72]	**66%**	12%	8%	**19%**	36.28k	11.9x	0.77%
Prop. (S^{unspec})	67.6%	**11.8%**	**0.9%**	31.0%	**8.43k**	**51.1x**	0.78%
Prop. (S^{spec})	72.6%	12.0%	1.7%	37.4%	10.28k	41.9x	0.8%
Louizos *et al.* [76]	45%	36%	0.4%	5%	6.15k	70x	1.0%
SWS [73]	N/A	N/A	N/A	N/A	2.15k	200x	0.97%
Sparse VD [74]	33%	**2%**	**0.2%**	5%	**1.54k**	**280x**	**0.75%**
DNS [75]	**14%**	3%	0.7%	**4%**	3.88k	111x	0.91%

our method performs the best; so, apparently sensitivity-based approach works well on fully-connected architectures.

In table 4.4 we show performances obtained on LeNet-5.
Here, when we compare our method to Han et al, we still achieve far better sparsity (51.07x vs. 11.87x compression ratio) for comparable error. An interesting observation is regarding the sparsity levels between convolutional and fully-connected: the layer initially having the most parameters (fc1) is the most sparsified. However, here Sparse VD obtains the top performances. We speculate that a possible explanation to this is that the currently-presented formulation for the sensitivity does not take into account the parameter sharing effect on the convolutional layers: in this way, particularly for conv1, the sensitivity is highly over-estimated and sparsity can not be properly achieved.

Table 4.5 VGG16 network trained over the ImageNet dataset

Layer	Layer size	Han *et al.*	S^{unspec}	S^{spec}
conv1_1	2K	58%	97.80%	96.35%
conv1_2	37K	22%	90.47%	80.87%
conv2_1	74K	34%	87.81%	81.49%
conv2_2	148K	36%	84.96%	81.41%
conv3_1	295K	53%	83.44%	77.68%
conv3_2	590K	24%	81.92%	71.81%
conv3_3	590K	42%	80.85%	69.25%
conv4_1	1M	32%	71.07%	62.03%
conv4_2	2M	27%	62.96%	51.2%
conv4_3	2M	34%	62.34%	51.91%
conv5_1	2M	35%	60.47%	57.09%
conv5_2	2M	29%	59.66%	57.08%
conv5_3	2M	36%	59.63%	47.75%
fc6	103M	4%	1.08%	1.13%
fc7	17M	4%	6.27%	8.35%
fc8	4M	23%	35.43%	14.81%
Total size	138M	10.35M	11.34M	9.77M
Top1 error	31.50%	31.34%	29.29%	30.92%
Top5 error	11.32%	10.88%	9.8%	10.06%

Finally, we run an experiment on the deep VGG-16 [77] over the ImageNet [78] dataset. VGG-16 is a 13 convolutional, 3 fully connected layers deep network having more than 100M parameters, while the ImageNet dataset consists of 224x224 24-bit colour images, divided in 1k different classes.
In this case we decided to skip the initial training step: the open-source keras pretrained model [77] has been used as a starting configuration.
According to the empirical results we show in table 4.5, our method achieves a 1.08% minimization in Top-5 error (9.80% vs 10.88%) under a comparable sparsification tha Han et al's. This results is extremely important: we are evidently able to improve the generalization of a deep model introducing sparsity!

4.6.1 Benefits for generalization

Last, we investigate how our sensitivity-based regularization term affects the network generalization ability, which is the ultimate goal of regularization. As we focus on the effects of the regularization term, no thresholding or pruning is applied and we

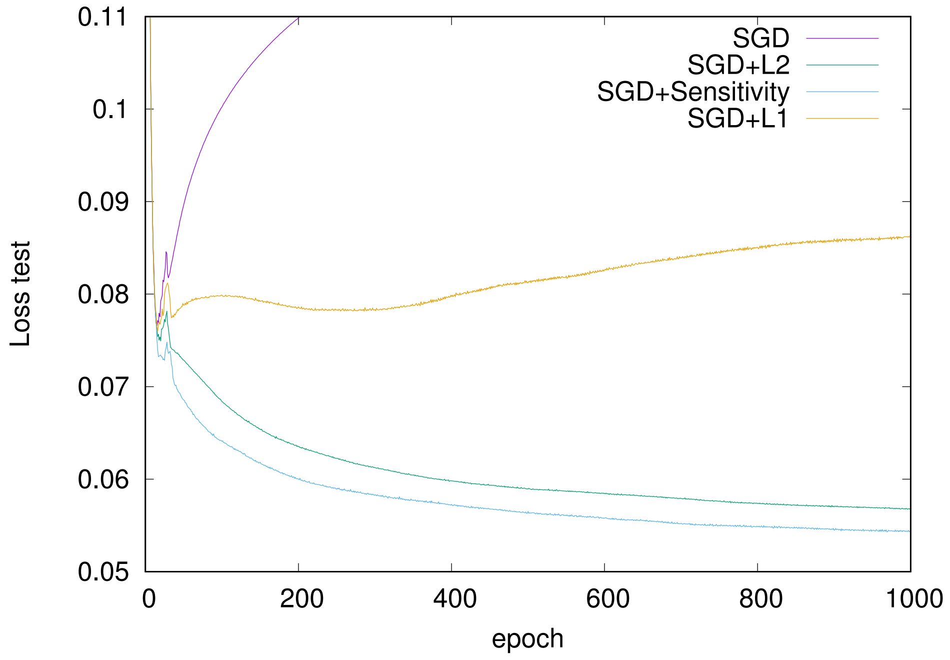


Fig. 4.5 Loss on test set across epochs for LeNet-300 trained on MNIST with different regularizers (without thresholding): our method enables improved generalization over L2-regularization.

consider the *unspecific* sensitivity formulation in eq. 4.37. We experiment over four formulations of the regularization term $R(\theta)$: no regularizer ($\lambda = 0$), weight decay (L2 regularization), L1 regularization, and our sensitivity-based regularizer. Fig. 4.5 shows the value of the loss function L (cross-entropy) over training. Without regularization, the loss increases after some epochs, indicating sharp over-fitting. With the L1-regularization, some over-fitting cannot be avoided, whereas L2-regularization prevents over-fit. However, our sensitivity-based regularizer is even more effective than weight decay regularization, achieving lower error. As seen from eq. 4.22, our regularization factor be interpreted as an *improved* L2 term with an additional factor promoting sparsity proportionally to each parameter insensitivity.

Chapter 5

Conclusion

In this work, the classification problem for artificial neural networks has been widely investigated. In order to do this, the very first analysis has been conducted on the perceptron, the simplest, *golden* model, reviewing state-of-the-art theoretical results and limits of algorithmic approaches.
Regarding the perceptron model, starting by observations from the theoretical approach and from some empirical results, a new way of training a binary perceptron, namely the *stochastic perceptron model*, was studied. It has been shown that such a model provides new and innovative insights in the world of optimization for binary neural networks.
Successively, the world of one hidden-layer neural networks has been explored. The theoretical analysis of the version space is here not straightforward because of the increasing complexity of the examined models. For such a reason, an heuristic approach for version-space exploration, inspired by the theory, has been designed: interesting properties regarding the accessible solution subspace were observed. Furthermore, an extension of the stochastic perceptron model for deep architectures has also been designed. However, it was empirically observed that such an approach stays below the state-of-the-art heuristic approaches for convolutional neural network topologies.
Finally, the problem of the parameters minimization in deep networks has been tackled. In particular, a regularization term from an update rule, inspired by the Hebb principle, has been proposed. It was observed that deep networks are able to save their performance using just a minimal part of the original parameters, which

recovers a theoretical concept observed for smaller networks: it is supposed to exist an optimal ANN architecture for a given learning problem.

References

[1] Matthias Jakob Schleiden. *Beiträge zur Phytogenesis*. 1838.

[2] Theodor Schwann. *Mikroskopische Untersuchungen über die Übereinstimmung in der Struktur und dem Wachstum der Tiere und Pflanzen*. BoD–Books on Demand, 2013.

[3] Otto Deiters. *Untersuchungen über die Lamina spiralis membranacea: ein Beitrag zur Kenntniss des inneren Gehörorgans*. Henry et Cohen, 1860.

[4] Paolo Mazzarello. *Il Nobel dimenticato: la vita e la scienza di Camillo Golgi*. Bollati Boringhieri Torino, 2006.

[5] Ortwin Bock. Cajal, golgi, nansen, schäfer and the neuron doctrine. *Endeavour*, 37(4):228–234, 2013.

[6] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[7] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry*. MIT press, 2017.

[8] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[9] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[10] Igor V Tetko, David J Livingstone, and Alexander I Luik. Neural network studies. 1. comparison of overfitting and overtraining. *Journal of chemical information and computer sciences*, 35(5):826–833, 1995.

[11] Arthur P Dempster, Nan M Laird, and Donald B Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977.

[12] Martin A Tanner and Wing Hung Wong. The calculation of posterior distributions by data augmentation. *Journal of the American statistical Association*, 82(398):528–540, 1987.

[13] Robert H Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in monte carlo simulations. *Physical review letters*, 58(2):86, 1987.

[14] Peter Bühlmann and Sara Van De Geer. *Statistics for high-dimensional data: methods, theory and applications*. Springer Science & Business Media, 2011.

[15] Donald F Specht. A general regression neural network. *IEEE transactions on neural networks*, 2(6):568–576, 1991.

[16] Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[17] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[18] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[19] Robert Kleinberg, Yuanzhi Li, and Yang Yuan. An alternative view: When does sgd escape local minima? *arXiv preprint arXiv:1802.06175*, 2018.

[20] Léon Bottou. Stochastic gradient learning in neural networks. *Proceedings of Neuro-Nımes*, 91(8):12, 1991.

[21] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[24] Tijmen Tieleman and Geoffery Hinton. Rmsprop gradient optimization. *URL http://www. cs. toronto. edu/tijmen/csc321/slides/lecture_slides_lec6. pdf*, 2014.

[25] Ruslan Salakhutdinov and Andriy Mnih. Bayesian probabilistic matrix factorization using markov chain monte carlo. In *Proceedings of the 25th international conference on Machine learning*, pages 880–887. ACM, 2008.

[26] Hui Yuan Xiong, Yoseph Barash, and Brendan J Frey. Bayesian prediction of tissue-regulated splicing using rna sequence and cellular context. *Bioinformatics*, 27(18):2554–2562, 2011.

[27] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[29] Adi Livnat, Christos Papadimitriou, Nicholas Pippenger, and Marcus W Feldman. Sex, mixability, and modularity. *Proceedings of the National Academy of Sciences*, 107(4):1452–1457, 2010.

[30] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[32] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of statistical planning and inference*, 90(2):227–244, 2000.

[33] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *Advances in Neural Information Processing Systems*, pages 2483–2493, 2018.

[34] Jonas Kohler, Hadi Daneshmand, Aurelien Lucchi, Ming Zhou, Klaus Neymeyr, and Thomas Hofmann. Towards a theoretical understanding of batch normalization. *arXiv preprint arXiv:1805.10694*, 2018.

[35] Richard E Bellman. *Adaptive control processes: a guided tour*, volume 2045. Princeton university press, 2015.

[36] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.

[37] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[38] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[39] John C Hay, FC Martin, and CW Wightman. The mark-1 perceptron-design and performance. In *Proceedings of the institute of radio engineers*, volume 48, pages 398–399, 1960.

[40] Mikel Olazaran. A sociological study of the official history of the perceptrons controversy. *Social Studies of Science*, 26(3):611–659, 1996.

[41] A Engel and Christian Van den Broeck. Systems that can learn from examples: Replica calculation of uniform convergence bounds for perceptrons. *Physical review letters*, 71(11):1772, 1993.

[42] JF Fontanari and R Meir. The statistical mechanics of the ising perceptron. *Journal of Physics A: Mathematical and General*, 26(5):1077, 1993.

[43] Andreas Engel and Christian Van den Broeck. *Statistical mechanics of learning*. Cambridge University Press, 2001.

[44] Werner Krauth and Marc Mézard. Storage capacity of memory networks with binary couplings. *Journal de Physique*, 50(20):3057–3066, 1989.

[45] Judea Pearl. *Reverend Bayes on inference engines: A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science ..., 1982.

[46] Carlo Baldassi, Alessandro Ingrosso, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Subdominant dense clusters allow for simple learning and high computational performance in neural networks with discrete synapses. *Physical review letters*, 115(12):128101, 2015.

[47] Carlo Baldassi, Alessandro Ingrosso, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Local entropy as a measure for sampling solutions in constraint satisfaction problems. *Journal of Statistical Mechanics: Theory and Experiment*, 2016(2):023301, 2016.

[48] Silvio Franz and Giorgio Parisi. Effective potential in glassy systems: theory and simulations. *Physica A: Statistical Mechanics and its Applications*, 261(3-4):317–339, 1998.

[49] Carlo Baldassi, Federica Gerace, Carlo Lucibello, Luca Saglietti, and Riccardo Zecchina. Learning may need only a few bits of synaptic precision. *Physical Review E*, 93(5):052313, 2016.

[50] Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. *arXiv preprint arXiv:1611.01838*, 2016.

[51] Carlo Baldassi, Federica Gerace, Hilbert J Kappen, Carlo Lucibello, Luca Saglietti, Enzo Tartaglione, and Riccardo Zecchina. Role of synaptic stochasticity in training low-precision neural networks. *Physical review letters*, 120(26):268103, 2018.

[52] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[53] Terrence J Sejnowski, Paul K Kienker, and Geoffrey E Hinton. Learning symmetry groups with hidden units: Beyond the perceptron. *Physica D*, 22(1-3):260–275, 1986.

[54] Marc Mézard, Giorgio Parisi, Nicolas Sourlas, Gérard Toulouse, and Miguel Virasoro. Replica symmetry breaking and the nature of the spin glass phase. *Journal de Physique*, 45(5):843–854, 1984.

[55] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[56] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1):215–243, 1968.

[57] Yann LeCun, Yoshua Bengio, et al. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[58] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[59] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[60] Henry W Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

[61] Ian J Goodfellow, Oriol Vinyals, and Andrew M Saxe. Qualitatively characterizing neural network optimization problems. *arXiv preprint arXiv:1412.6544*, 2014.

[62] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred A Hamprecht. Essentially no barriers in neural network energy landscape. *arXiv preprint arXiv:1803.00885*, 2018.

[63] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[64] José Miguel Hernández-Lobato and Ryan Adams. Probabilistic backpropagation for scalable learning of bayesian neural networks. In *International Conference on Machine Learning*, pages 1861–1869, 2015.

[65] Y. LECUN. The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*.

[66] Marwan Jabri and Barry Flower. Weight perturbation: An optimal architecture and learning technique for analog vlsi feedforward and recurrent multilayer networks. *IEEE Transactions on Neural Networks*, 3(1):154–157, 1992.

[67] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[68] Enzo Tartaglione, Skjalg Lepsøy, Attilio Fiandrotti, and Gianluca Francini. Learning sparse neural networks via sensitivity-driven regularization. In *Advances in Neural Information Processing Systems*, pages 3878–3888, 2018.

[69] Durk P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

[70] Donald O Hebb. The organization of behavior: A neurophysiological approach, 1949.

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

[72] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, pages 1135–1143, 2015.

[73] Karen Ullrich, Edward Meeds, and Max Welling. Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*, 2017.

[74] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks. *arXiv preprint arXiv:1701.05369*, 2017.

[75] Yiwen Guo, Anbang Yao, and Yurong Chen. Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387, 2016.

[76] Christos Louizos, Max Welling, and Diederik P Kingma. Learning sparse neural networks through l_0 regularization. *arXiv preprint arXiv:1712.01312*, 2017.

[77] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[78] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, December 2015.