DRAFT
Deep Learning: A Statistical Perspective

Pietro Lesci

Latest version: 20 November 2018

Declaration

Blank

Acknowledgment

Blank

Blank

Abstract

Blank

Contents

1 Introduction **1**
1.1 The Two Cultures . . . 2
1.2 Types of learning . . . 5
1.3 The Anatomy of a Learner . . . 5
1.4 Shallow Learners and Neural Networks . . . 6
1.5 Deep Learners and Deep Neural Networks . . . 9

2 The Learning Process **12**
2.1 Optimization . . . 12
2.1.1 The learning rate problem . . . 15
2.1.2 Differentiation: Back-Propagation . . . 15
2.2 Regularization . . . 15
2.2.1 Parameter Norm Penalties . . . 16
2.2.2 Early Stopping . . . 17
2.2.3 Sparse Representations . . . 18
2.2.4 Bagging and Ensemble Methods . . . 18
2.2.5 Dropout . . . 19

3 Deep Probabilistic Learning **20**
3.1 Deep Learning: The Probabilistic model . . . 20
3.2 Bayesian Inference for Deep Learning . . . 20

4 **21**

List of Figures

1.1 A stylized example of a fully-connected feedforward neural network. The input layer has four units, corresponding to the components of the vector $x_i \in \mathbb{R}^Q,\ i = 1, \ldots, N$, with $Q = 4$. Each circle is a neuron calculates a weighted sum of an input vector plus bias and applies a nonlinear function to produce an output, which is this case is a scalar $y_i,\ i = 1, \ldots, N$, that is $D = 1$. 8
1.2 Most used activation functions. 9
1.3 The network graph of linear regression. For each observation, the predictor is a 3-D vector. The addition of the intercept has been made explicit. 9

Notation

The next list describes several symbols that will be later used within the body of the document

Numbers and Arrays

A	A matrix or a random quantity, clarified by the context
D	Output dimesionality
I	Identity matrix
L	Number of layers in the network or per-example loss function, clarified by the context
N	Dimensionality of the training set
$N_{(test)}$	Dimensionality of the test set
Q	Input dimesionality
W	The weight matrix
X	Training set (matrix with N rows, and D columns)
x	A deterministic quantity, vector or scalar

Set and Graphs

$\mathbb{R}$	The set of real numbers
$\{0, 1, \ldots, n\}$	The set of all integers between 0 and n

Indexing

a_i	Element i of deterministic or random vector a, with indexing starting at 1
$A_{:,i}$	Column i of matrix A
$A_{i,:}$	Row i of matrix A

A_{ij}	Element i, j of matrix A

Acronyms / Abbreviations

i.i.d.	Independent and identically distributed
DL	Deep Learning
DNN	Deep Neural Network

Chapter 1

Introduction Algorithms and Data Models: The Anatomy of Learners

The desire to create machines capable of thinking accompanies the human kind since the first programmable computer was invented. Nowadays, artificial intelligence (AI) is a thriving field of research that encompasses various disciplines such as computer science, engineering, statistics, neuroscience, biology, with applications that ranges from automating routine labour, interpreting images, understanding speech, to making diagnoses in medicine.

Problems that are intellectually difficult for humans to tackle, namely those problems that can be described by a list of formal, mathematical rules, are among the easiest for computers to solve. The real challenge to AI is the resolution of tasks relatively easy for people to perform, but hard to describe in formal terms – problems that we, as human beings, solve intuitively, instinctively - such as recognizing objects in images. This instinctive "intuitions" are drawn from experience: collections of small facts (data) and personal judgments of facts (information).

One solution to let machines mimic the process of deduction from experience is to hard-code the knowledge about the reality in formal language, allowing the computer to reason using logical inference rules. This approach, called **knowledge-based** [9], is brute-force in nature and not viable in many practical applications.

Therefore, AI systems need to have the ability to "build" their own knowledge by extracting information from raw data. This field of research is known as **machine learning**. However, the performance of this approach crucially depends on the *representation* of data used – the usual maxim "garbage in, garbage out". Each bit of information included in the representation is called **feature**. Selecting or extracting features, that form representations, from raw data is a form of art of its own. Often, as an alternative to hand-designed features, machine learning algorithms are used to learn such *representations* besides learning the *mapping* from representations to outputs. This approach is called **representation leaning** [9].

Regardless the methodology used to build such features, the aim is often to separate the **factors of variation** that explain the observed data. In many cases, these factors are not directly observed:

"[...] They may exist as either unobserved objects or unobserved forces [...] constructs in the human mind [...] concepts or abstractions that help us make sense of the rich variability in the data" – [9] pp. 4-5

Therefore, extracting representations can be as difficult as solving the original problem. In these cases representation learning comes unhandy. **Deep learning** (DL) provides a solution by taking a constructionist approach: creating autonomously complex, expressive, representations of the world in terms of simple, more basic, ones. It can be defined as

"[...] A form of machine learning that uses hierarchical abstract layers of latent variables to perform pattern matching and prediction" – [17] p. 1

In the statistical literature inputs are often called *predictors* or, more classically, *independent variables*. Outputs are usually referred to as *responses* or, more classically, *dependent variables*. Throughout the thesis these terms are used interchangeably. However, differences between statistics and machine learning are not limited to nomenclature. They truly can be defined as "two cultures" [3]. In the next section we will discover how a different philosophical approach has been the cause of two, sometimes competing and opposite, visions of the world, that with this work we would like to bring closer.

1.1 The Two Cultures

In one of its most contended contributions [3], Leo Breiman states that

"There are two cultures in the use of statistical modeling to reach conclusions from data. One assumes that the data are generated by a given *stochastic data model*. The other uses *algorithmic models* and treats the data mechanism as unknown. The statistical community has been committed to the almost exclusive use of data models. This commitment has led to irrelevant theory, questionable conclusions, and has kept statisticians from working on a large range of interesting current problems. Algorithmic modeling, both in theory and practice, has developed rapidly in fields outside statistics. It can be used both on large complex data sets and as a more accurate and informative alternative to data modeling on smaller data sets"

Philosophically, one can think of the world as a deterministic chain of causes and effects. Albeit complex, one can believe that there exist, unique, a cause-effect chain that leads to a specific outcome, manifests itself as a fact of our reality. If this is the case, then an all-mighty intellect could, in theory, be able to trace back each effect to its cause. Chance would not exist. Even if we admit that randomness pervades parts of out reality, we would still be bounded to reason about the complementary deterministic part. Our human nature, its finiteness, limits us twice: first, it limits the scope of investigation to the deterministic component of the real; second, it limits our possibilities to reason directly about it. We are bound to observe the deterministic part of our universe in an indirect way that, alone, creates an additional layer of randomness between us and the “truth”. Fortunately, statistics provides us with mathematical tools to deal with this latter layer.

The role of statistics is to try to discover and understand the functioning of our reality starting from collections of facts, the data. These are regarded as being generated by a black box in which a set of possible causes, interpreted as a vector of input variables x, enter on one side and an effect, a response variables y, comes out on the other:

$$\begin{tikzcd} y & \boxed{\text{Nature}} \arrow[l] & x \arrow[l] \end{tikzcd}$$

Inside the black box, nature associates the predictor variables with the response variables. We get to collect y and x as data, and the goal in analyzing them is twofold:

> *Understanding.* To extract some information about how nature is associating the response variables to the input variables
>
> *Prediction.* To be able to predict what the responses are going to be when future inputs will be fed in

Similarly, there are two approaches towards these goals:

- *The Data Modeling Culture*: The analysis starts by assuming a *stochastic data model* for the inside of the black box, e.g. assuming that data are generated by independent draws from $y = f(x, \text{random noise}, \text{parameters})$, and then the values of the parameters are estimated from the data; model validation is performed based on goodness-of-fit tests and residuals analysis

- *The Algorithmic Modeling Culture*: The analysis considers the inside of the box complex and unknown and the purpose is to find a function $f(x)$ – an algorithm, a rule that operates on x to predict the responses y; model validation is performed measuring predictive accuracy

These approaches differ in the logic used. The former start by trying to impose on nature a set of models – often those the statisticians are comfortable with, well-known, whose properties are well studied – and verifies that at least one of them is compatible with the data observed. The latter, on the contrary, is model-agnostic. Its ultimate target is finding a a rule that *approximates* the mechanism associating inputs and outputs, even if this algorithm is convolute, inscrutable and unexpected, without imposing any a priori characteristics on the data-generating process.

Breiman [3] argues that statisticians in applied research consider data modeling as the main, if not the only, go-to for statistical analysis:

> “I started reading the *Annals of Statistics*, the flagship journal of theoretical statistics, and was bemused. Every article started with: *Assume that the data are generated by the following model: ...*”

The underlying assumption of the data modeling approach is that the parametric class of models imagined by the statistician for the complex mechanism devised by nature is indeed reasonable. Parameters are then estimated and conclusions are drawn.

However, the conclusions drawn are about the model's mechanism, and not about nature's. Obviously, if the model is a poor approximation of nature, the conclusions may be wrong. On the other hand, a plus of data modeling is that it produces a simple and understandable description of the relationship between inputs and responses. This simplicity comes at the cost of uniqueness: different models, although equally valid, may give different descriptions of this relation. The fact that, following this approach, models are evaluated mainly using goodness-of-fit tests and other methods for checking fit, that yield a yes-no answer, creates difficulties in ranking the various models by quality [3]. The alternative proposed by the algorithmic culture is to measure the accuracy of the model's predictions: estimating the parameters in the model using the data and then using the model to predict the data checking how good the prediction is. The extent to which the model emulates nature's mechanics is a measure of how well the model can reproduce the natural phenomenon producing the data.

Sometimes approaching problems by looking for a data model imposes an a priori restriction to the ability of statisticians to deal with a wide range of statistical problems when they cannot be dealt with using models that remain enough understandable. The algorithmic community rarely make use of data models. Their approach starts by looking at the mechanics of how nature produces data as a black box partly unknowable. They stop at the mere acceptance of the fact that what is observed is a set of x's that goes in and a set of y's that comes out. The problem is rephrased as finding an algorithm, a rule $f(x)$, such that for future observed x, $f(x)$ will be a good predictor of y. The focus is shifted from data models to the properties of algorithms. The implicit assumption made in the theory is that the data is drawn *i.i.d.* from an unknown multivariate distribution. The drawback of this approach is that models that best emulate nature in terms of predictive accuracy are also the most complex.

Recent developments in both fields are trying to tie together these two cultures: this thesis goes in this direction, starting from the algorithmic approach and trying to extend it with concepts borrowed from the data modeling one. The philosophical underpinning throughout this work can be summarized as:

> The point of a model is to get useful information about the relation between the response and predictor variables. Interpretability is a way of getting information. But a model does not have to be simple to provide reliable information about the relation between predictor and response variables; neither does it have to be a data model. The goal is not interpretability, but accurate information [...] the emphasis needs to be on the problem and on the data – [3]

with a focus on the uncertainty regarding the information the model provides, i.e. its reliability in real life applications, that are the final purpose of all our efforts.

How can we understand, concretely, the mechanics of nature? That depends on how nature manifests itself to us, namely what kind of data we are able to get. This shapes the way we can learn about nature from data and is the topic of the next section.

1.2 Types of learning

All models, methods, techniques, algorithms, implemented to discover and understand the mechanics of nature are informally considered to perform a kind of leaning. Based on what data we get to see, learning is specified in different ways broadly organized in two[1] categories: supervised and unsupervised.

Supervised learning problems entail learning an input-output mapping f by examples. The learner is provided with both the predictors and the outcome variables. The learner can thus be "trained" by letting it produce a prediction, given a particular input, that is then compared with the ground truth. Furthermore, based on the output type the prediction task takes two different names: *regression* for quantitative outputs, and *classification* for categorical outputs. These two tasks have a lot in common and both can be viewed as a function approximation task. This is the approach taken, especially, in applied mathematics and statistics [10].

In **unsupervised learning** problems, on the contrary, only the predictors are observed, no measurements of the outcome is provided. The learner's task is to describe how the data are organized or clustered, rather than make predictions [10]. In what follows we will only consider supervised learning problems. In next section the identity of the subject performing the learning will be revealed.

1.3 The Anatomy of a Learner

An informative high-level definition of learner, or learning algorithm, is provided in [16]:

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E"

Of course, the variety of experiences, tasks, and performance measures is wide. Below a high-level overview of the possible instances of this concepts is provided:

- Experiences E: supervised or unsupervised learning
- Tasks T: classification, classification with missing inputs, regression, anomaly detection, imputation of missing values, denoising, density estimation
- Performance measures P: accuracy, error rate, goodness-of-fit tests

The experience the algorithm is allowed to make shapes its tasks and performance measures. The starting point is the definition of a model. The statistician chooses an appropriate loss function according to the problem at hand. Once the loss function is chosen, an optimization criterion is defined, e.g. maximum likelihood, and an optimization routine must be set up. Typically the optimization problem is not solved analytically deriving a closed-form formula to provide a symbolic expression for the

[1]For sake of simplicity, I am consciously not mentioning: reinforcement learning, meta-learning, semi-supervised learning, transfer learning.

correct solution. On the contrary, updated estimates of the solutions are found via iterative processes[2].

Therefore, all learning algorithms share a common "anatomy":

1. A specification of a dataset
2. A model that defines a parametrised input-output mapping
3. A loss function that depends on the parameters of the model
4. An optimization procedure that finds the parameters values minimizing the loss function

These components are modular: replacing them independently we can obtain a wide range of algorithms [9]. The cost function typically includes at least one term that causes the learning process to perform statistical estimation. It may also include additional terms used for regularization.

1.4 Shallow Learners and Neural Networks

The simplest example of a supervised learning is linear regression. The task is to predict a quantitative value Y given some inputs X. We are given a set of N input-output pairs $\{(x_i, y_i)\}_{i=1}^{N}$. We assume that natural mechanics associating inputs and outputs can be described by a linear function mapping each $x_i \in \mathbb{R}^Q$ to $y_i \in \mathbb{R}^D$. For each input-output pair, the model is a linear transformation of the inputs:

$$f(x) = Wx$$

where W is a D-by-Q matrix of parameters, known as **weights**. Usually, a modified version of the model is used: an intercept term, $b \in \mathbb{R}^D$, is added in the equation

$$f(x) = Wx + b$$

This still preserves the linear mapping from parameters to predictions, but the mapping from features to predictions is now an affine function[3]. The intercept, in the machine learning literature, is often called the **bias**[4] [9] parameter (of the affine transformation) or **offset vector** [17]. This nomenclature derives from the fact that, in the absence of any input, the transformation is biased towards b.

Different parameters W, b define different linear transformations. The goal is to find those values that minimize the errors in prediction, a procedure often referred to as "fitting" the model to data. There are many different methods to fit the model to data. The most popular, especially for linear regression, is the method of least squares:

[2]Optimization refers to the task of either minimizing or maximizing some function $J(\theta)$ by altering θ. Since any maximization problem can be turned into a minimization problem, we will only consider minimization problems.

[3]Albeit trivial as a remark, it will be useful in what follows, especially when defining deep learning formally.

[4]This term is not related whatsoever to the idea of *statistical bias*.

the values in W that minimize the residual sum of squares between the true outcomes and the predicted ones are chosen:

$$J(\theta) = \sum_{i-1}^{N} \|y_i - (Wx_i + b)\|_2^2 , \quad \theta = (W, b)$$

where $\| \cdot \|$ is the L^2-norm (euclidean norm). The function J will represent the loss function henceforth, also called cost or objective function. Minimizing the loss function is the core of any machine learning algorithm: there exists a tight relationship between learning and optimization. In the specific case of this loss function a close-form solution exists. However, usually, such an analytic solution does not exist and parameters estimation must be performed via numerical computation.

One way to measure the performance of the model is to compute the *mean squared error* of the model on the new, unseen data. For this purpose, often in practical applications, the dataset is divided into **training** set, used to perform the learning process, and **test** set, used to measure out-of-sample performances. Sometimes, especially for DL, a third partition is created called **validation** set and is used to learn the hyper-parameters of the model, if not calibrated ex-ante. If not explicitly declared, N will refer to the number of observations in the training set.

To improve models performances means to reduce the mean squared error on the test set, MSE_{test}. However, only the training data are used to train the model. One can intuitively minimize the mean squared error on the training set, MSE_{train}, simply solving for where its gradient is $\nabla_w \text{MSE}_{train} = 0$. What we actually care about is the test error or generalization error, computed on the unobserved data, i.e. test set. How can we affect the performance on the test set when we can observe only the training set? Statistical theory comes handy in justifying this. We are able to do so thanks to the implicit assumption (§1.1) that the training and test set are produced by the same data-generating process and each observation is independent of any other.

Generally, the relation between X and Y need not be linear. Indeed, in many interesting cases characterized by big and complex data (e.g. speech recognition, image processing) the mapping is better described by a nonlinear function. Nevertheless, one can resort a kind of linear mapping, called *linear basis function regression* [1], where each observed input x, in a preliminary step, is passed through K fixed scalar-valued nonlinear transformations φ_k to compose a K-dimensional[5] feature vector $\varphi(x) = [\varphi_1(x), \dots, \varphi_K(x)]$. Linear regression is then performed using the transformed vector, $\varphi(x)$, instead of x itself. The transformations $\{\varphi_k\}_{k=1}^{K}$ are called *basis functions*, often assumed to be fixed and orthogonal to each other. To achieve a higher degree of flexibility, one can relax these constraints allowing the basis functions to be parametrised [1].

Neural networks are a particular case in which the basis functions are defined as $\varphi_k^{w_k,b_k}(x)$ where the scalar-valued nonlinear function is applied to the affine transformation of the inputs, $w_k x + b_k$, with w_k a K-dimensional vector and b_k a scalar. The nonlinearity is defined to be identical for all k, i.e. $\varphi_k = \varphi$. Again, linear regression is then performed using the transformed vectors as features. The model output can be written as

$$f(x) = W_{reg}\, \varphi^{W,b}(x) + b_{reg}$$

[5]When $K < Q$ we are indeed performing dimensionality reduction.

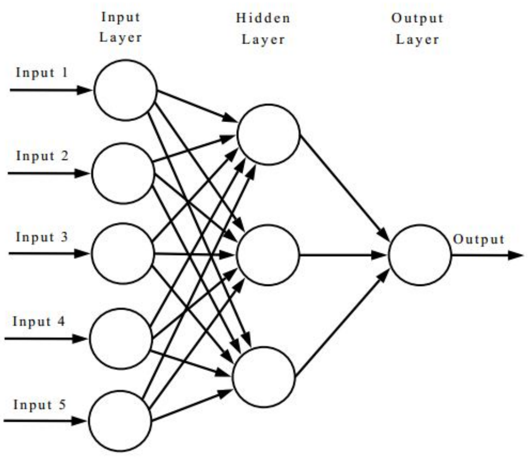

Figure 1.1: *A stylized example of a fully-connected feedforward neural network. The input layer has four units, corresponding to the components of the vector $x_i \in \mathbb{R}^Q,\ i = 1, \ldots, N$, with $Q = 4$. Each circle is a neuron calculates a weighted sum of an input vector plus bias and applies a nonlinear function to produce an output, which is this case is a scalar $y_i,\ i = 1, \ldots, N$, that is $D = 1$.*

where the subscripts are used to distinguish between the parameters of the original linear regression, (reg), and the parameters of the transformation. More precisely, $\varphi^{W,b}(x) = \left[\varphi^{w_1,b_1}(x), \ldots, \varphi^{w_K,b_K}(x)\right]$, W_{reg} is a D-by-Q matrix, b_{reg} is a D-dimensional vector, W is a K-by-Q matrix, and b is a K-dimensional vector. To fit the model to data one has to find W, b, W_{reg}, b_{reg} that minimize the average $\|y - f(x)\|_2^2$.

This is the canonical example of neural network, known also as fully-connected feedforward neural network [19, 9][6]. It can be interpreted as a two-stage regression model: in the first stage the features are derived via a composition of an affine transformation of the inputs and a component-wise nonlinear function – referred to as *activation function* – ; these extracted features, in the second stage, are mapped to the output via a second affine transformation [10]. In fact, also linear regression can be interpreted as a very simple example of neural network where φ is an identity mapping. These models are associated with directed acyclic graphs, known as *network diagram*. The neural network described above can be represented as in figure 1.1.

The most common activation function used are presented in figure 1.2. The intermediate layer, highlighted in grey, is called **hidden layers**. The dimesionality, K, of the hidden layers defines the **width** of the model, while the overall number of hidden layers determines the **depth** of the model – being a plain neural network the depth is obviously one. Usually, layers are not interpreted as vector-to-vector functions. Rather, they are thought of as consisting of computational units or neurons, that act in parallel, each representing a vector-to-scalar function. This view highlights the inspiration of neural networks to the human brain.

The neural network described above is an example of a shallow learner: informally, every model that cannot be categorized as deep is a shallow learner. Almost all shallow

[6]For a historical review of neural networks and deep learning consult [20].

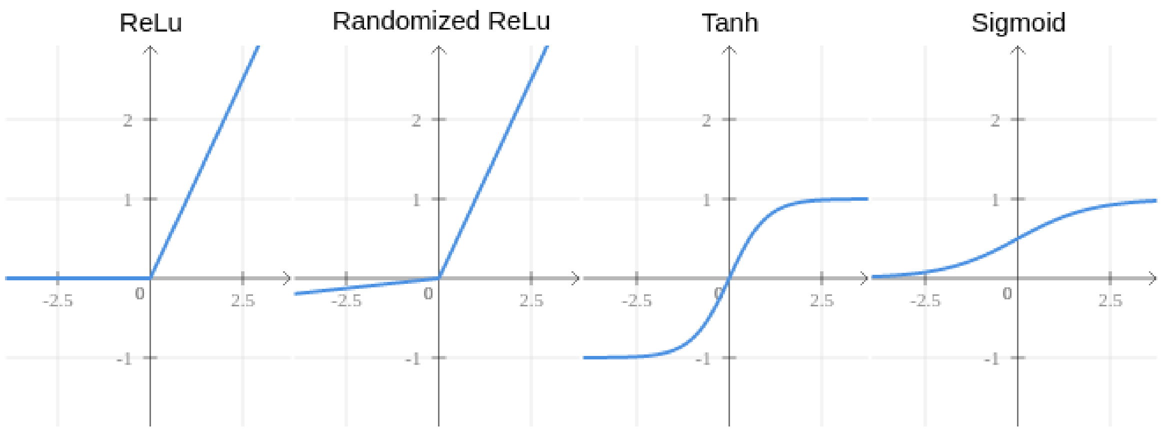

Figure 1.2: *Most used activation functions.*

learners are data reduction techniques that consist of a low dimensional auxiliary variable Z and a prediction rule specified by a composition of functions

$$f(x) = f_1\big(f_2(x)\big) = f_1(z), \qquad f_2(x) := z$$

Linear regression, Logistic regression, Principal component analysis, Partial least

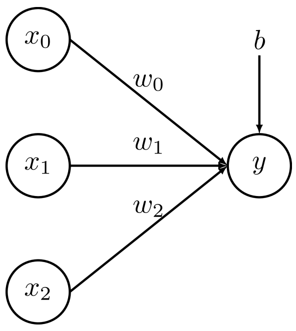

Figure 1.3: *The network graph of linear regression. For each observation, the predictor is a 3-D vector. The addition of the intercept has been made explicit.*

squares, Reduced rank regression, Linear discriminant analysis and Project pursuit regression are all examples of shallow learners. Furthermore, they all can be represented as a neural network, for example linear regression network diagram is shown in figure 1.3.

1.5 Deep Learners and Deep Neural Networks

Deep learning refers to a wide class of machine learning techniques and architectures, with the common trait of "[...] using many layers of nonlinear information processing" [6]. In general, a deep learner, is a representation learning methods with multiple levels of representation, obtained by composing simple but nonlinear transformations of the raw data, with each of these transforming the representation at one level (starting with the raw input) into a representation at a higher level of abstraction. Composing enough of such transformations, very complex functions can be learned [14]. A deep architecture can be thought as a multilayer stack of simple models.

Deep neural networks (DNN) belong of course to this class of models. Given an input, $X \in \mathbb{R}^Q$, and an output, $Y \in \mathbb{R}^D$, usually both high-dimensional, the mapping $f : R^Q \to R^D$ is modeled via the superposition of univariate semi-affine functions where univariate activation functions are used to decompose the high-dimensional X, as a plain neural network. The particular characteristic of DNN is the depth of the networks, usually more than 2 hidden layers. Then a deep prediction rule can be expressed as:

$$\begin{aligned}
h^{(1)} &= \varphi_1 \left(W^{(0)} X + b_0\right) \\
h^{(2)} &= \varphi_2 \left(W^{(1)} h^{(1)} + b_1\right) \\
&\dots \\
h^{(L)} &= \varphi_L \left(W^{(L-1)} h^{(L-1)} + b_{L-1}\right) \\
f(X) &= W^{(L)} h^{(L)} + b_L
\end{aligned}$$

DNN are, perhaps, the most important inhabitant of this class of models, however other models characterized by deep architecture exists: deep gaussian processes and neural processes that are treated in the following chapters.

How a series of simple operation can represent complicated mappings? In other words, why this approach works? The formal roots of DL can be traced back in Kolmogorov's representation of a multivariate response surface as a superposition of univariate activation functions applied to an affine transformation of the input variables [13]. In 1957 the Russian mathematician Kolmogorov showed that *any* continuous function f of many variables can be represented as a composition of addition and some functions of one variable. The original version of this theorem can be expressed as follows[7]:

Theorem 1 (Kolmogorov–Arnold superposition theorem) *Let $f : [0,1]^d \to \mathbb{R}$ be an arbitrary multivariate continuous function defined on the identity hypercube. Then it has the representation*

$$f(\mathbf{x}) = f(x_1, \dots, x_d) = \sum_{q=0}^{2d} \phi_q \left(\sum_{p=1}^{d} \psi_{q,p}(x_p) \right)$$

with continuous one–dimensional inner and outer functions ϕ_q and $\psi_{q,p}$, defined on the real line. The inner functions $\psi_{q,p}$ are independent of the function f.

Starting from the 2000s, Kolmogorov's superposition theorem found the attention of the machine learning community interest in providing a theoretical justification of neural networks. Hecht–Nielsen's [15] tries to apply the theorem to a feed-forward network with an input layer, one hidden layer and an output layer [11]. However, the inner functions, ψ, in this application of the theorem are highly nonsmooth and thus

[7]We propose a version of the theorem deducted from [2, 15].

not useful in optimization. Sprecher [21] contributed to this research topic providing an explicit method to construct the univariate functions ψ, corresponding to the activation functions of the network. Bryant [4] implements Sprecher's [22] algorithm to estimate the inner link function. In his application of the theorem, deep layers allow for smooth activation functions to provide "learned" hyperplanes which find the underlying complex interactions and regions without having to see an exponentially large number of training samples.

To summarize, deep learning architectures rather than manually engineering the transformations to be applied to create more abstract concepts, learn them: the behaviour of the hidden layers is not directly specified in the data, it is learned.

Chapter 2

The Learning Process

The goal of the learning process of any learning algorithm is to minimize the expected generalization error, known as **risk**, i.e. to maximize its accuracy when new inputs are provided. This is achieved via two paths: **optimization**, which is the process of finding parameters values that minimize the loss function, and **regularization** which is the process of reducing model capacity avoiding overfitting on the training set and improve generalization performances. To use [16] codification, we care about some performance measure P, that is defined with respect to the test set. However, it can be an intractable problem. Therefore, optimization of P is only performed indirectly by defining a different cost function $J(\theta)$, minimizing it, and hoping that in doing this we are improving P. In the next sections we will analyze how this "hope" can be made more concrete.

2.1 Optimization

The aim of optimization is to find values of parameter θ in the parameter space Θ that minimize the loss function $J(\theta)$, which usually include a term specifying a performance measure evaluated on the training set and a regularization term to improve the generalization error and avoid overfitting which will be discussed in the next section.

Note that for the purpose of this thesis, we will treat optimization in a supervised learning context, i.e. we are provided with the outcome variable Y as well as the inputs X. Furthermore, throughout this section, we will refer to the unregularized optimization case, i.e. the loss function will not contain any regularization term. The development of the regularized cases will be addressed in the next section.

It is common in machine learning applications to use loss functions that decompose as a sum over some per-example loss functions. Therefore, up to a multiplicative constant, the cost function can be seen as an expectation [9]. In principle, we would like to minimize the objective function where the expectation is taken over the data-generating distribution, p_{data}:

$$J^{\star}(\theta) = \mathrm{E}_{(X,Y)\sim p_{data}} L\big(f(x;\theta);y\big) \tag{2.1}$$

that defines the risk, i.e. the expected generalization error, while $f(x;\theta)$ defines the predicted output when the input is x.

However, in practise we only have a finite training set. Therefore, we are bound to use a reduced form of $J^{\star}(\theta)$, namely the **empirical risk** defined as

$$J(\theta) = \mathrm{E}_{(X,Y)\sim\hat{p}_{data}} L\big(f(x;\theta);y\big) = \frac{1}{N}\sum_{i=1}^{N} L\big(f(x_i;\theta);y_i\big) \tag{2.2}$$

where $\hat{p}_{data}$ is the empirical distribution on the training set. Therefore, rather than optimizing the risk directly, the empirical risk is minimized with the hope that the risk decreases as well – a procedure called empirical risk minimization.

The most important mathematical tool used for optimization is the derivative operator, denoted as $J'(\theta)$ or $\frac{dJ}{d\theta}$. The concept of derivative is crucial for optimization algorithms because it specifies how to scale a small change in the parameters θ to obtain the corresponding change in the loss function valuation

$$J(\theta+\eta) \approx J(\theta) + \eta J'(\theta)$$

It tells us that $J(\theta)$ can be reduced by moving θ in small steps with the opposite sign of the derivative: $J\big(\theta - \eta\ sign(J'(\theta))\big)$. The type of optimization techniques employing the derivative concept as tool is called **gradient-based** optimization. In machine learning this is the most used class of methods and will be the object of this section.

How does gradient-based optimization work? Since, as said, often loss functions decompose as a sum over some per-example loss function, the computation of the gradient is performed using the linearity of the derivative operator

$$\begin{aligned} g = \nabla_\theta J(\theta) &= \nabla_\theta \mathrm{E}_{(X,Y)\sim\hat{p}_{data}} L\big(f(x;\theta);y\big) \\ &= \nabla_\theta \frac{1}{N}\sum_{i=1}^{N} L\big(f(x_i;\theta);y_i\big) \\ &= \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta L\big(f(x_i;\theta);y_i\big) \end{aligned} \tag{2.3}$$

The most basic example of gradient-based optimization algorithm is **gradient descent** [5, 1]. It proposes a new parameter value according to the updating rule

$$\theta^{new} \leftarrow \theta - \lambda g \tag{2.4}$$

where $\lambda \in \mathbb{R}$ is an hyperparameter called the **learning rate**, a positive scalar determining how the gradient affects the proposed value[1].

One issue in applying gradient descent is that, when the data is big, computing the expectation in eq. (2.3) is computationally very expensive since before an update is proposed, the model must be evaluated on the entire dataset in order to compute the per-example loss. Optimization methods that use the entire training set to compute the gradient are called **batch** gradient methods. One way to solve the issue is to resort

[1]Its determination has posed serious challenges to both practitioners and researchers. In subsection (§2.1.1) we will review ways to set the parameter. Usually for gradient descent it is kept fixed during the entire optimization.

to statistical estimation. Instead of computing the exact gradient, it is approximated by randomly sampling a small number of examples from the training set, computing the per-example loss, and taking the average over them only. This type of optimization methods is called **minibatch** or **stochastic** methods. Of course, to compute an unbiased estimate of the expected gradient, minibatches must be selected truly randomly. It can be shown[2] that when the stochastic gradient is computed on different observations – i.e. no observation is resampled – the approximate expected gradient follows the gradient of the true generalization error (2.1). Each time the training set is fully sampled, the procedure restarts again. Each of this iterations is usually called **epoch**.

The basic example of a stochastic optimization method is **stochastic gradient descent**. The update rule for the parameters is the same as in eq. (2.4) but instead of an exact g, an approximate $\hat{g}$ is used. The learning rate usually is not held fixed, but it is gradually decreased over time. This is necessary since the estimation of g introduces noise, via the random sampling, that does not vanish even when a minimum is reached. Adaptive learning rate methods are discussed below. On the contrary, batch optimization methods, such as gradient descent, can hold λ fixed over time[3],

Another example of stochastic optimization is the **momentum** method. This algorithm adds a variable v. called *velocity*, that stores value of an exponentially decaying moving average of the past gradient and suggests the algorithm to move in the same direction. A memory parameter $\alpha \in [0, 1)$ determines how much past information to store. The update rule becomes

$$\begin{aligned} v^{new} &\leftarrow \alpha v - \lambda \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\left(f(x_i;\theta); y_i\right) \right), \quad m < N \\ \theta^{new} &\leftarrow \theta + v^{new} \end{aligned}$$

Given λ – that should be modeled to vary in some ways – the size of the step now depends on how large and aligned the sequence of gradients is.

A slightly modified version of the momentum algorithm is the **Nesterov momentum**. The gradient is evaluated after the velocity is applied. The update rule in this case is

$$\begin{aligned} v^{new} &\leftarrow \alpha v - \lambda \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} L\big(f(x_i;\theta + \alpha v); y_i\big) \right) \\ \theta^{new} &\leftarrow \theta + v^{new} \end{aligned}$$

It is important to note that given the common non-convexity of loss functions in deep learning, an important factor impacting the convergence of optimization algorithms is parameter initialization, i.e. the starting point of the path towards the minimum.

[2][9] pp. 273-274.

[3]For more details on how to tackle the issue, the reader is redirected to [9] ch. 8.

2.1.1 The learning rate problem

How does the learning parameter need to be chosen? This question has multiple answers, each corresponding to a different algorithm. Here we briefly review the most important that will be useful in actual applications.

Delta-bar-delta. The delta-bar-delta [18] algorithm was one of the first heuristic method to adapt the learning rate of batch optimization methods. The rule is simple: if the partial derivative of the loss with respect to a given model parameter remains the same sign, then the learning rate should increase, otherwise the learning rate should decrease.

AdaGrad. A similar heuristic approach is AdaGrad [7] that scales the individual model parameters inversely proportional to the square root of the sum of all the historical squared values of the gradient.

RMSProp. A modification of AdaGrad is provided by the RMSProp algorithm [8]. It changes the gradient accumulation into an exponentially weighted moving average commanded by an hyperparameter ρ determining the algorithm's memory.

Adam. The Adam algorithm combines RMSProp to learn the learning rate with the momentum method [4].

Supervised pretraining. Sometimes optimization is brought about using a completely different approach [9]: training a simpler model and use the trained parameters as initial values of the more complex model.

2.1.2 Differentiation: Back-Propagation

How does gradient-based optimization is performed? In other words, how the derivatives are computed? The flow of information from the inputs X to the estimated output $f(X)$ and finally to the cost function is called **forward propagation**. The inverse flow is called **back-propagation** and allows the information to flow from the loss function backward in order to compute the gradients. The error made in the estimation measured by the loss function is "propagated" back into the network to allow each parameter to receive its share of the error. Back-propagation is a method to compute the derivative via the chain rule, $dz/dx = dz/dy \times dy/dx$, with a specific order of operation that is highly efficient. It contributed to the recent fame of deep learning since it allowed the training of deeper network architectures[5].

2.2 Regularization

As said, the aim of a learning algorithm is twofold: minimizing the training error and the the gap between training and test error. In other words, avoiding

[4]For further details [9] ch. 8

[5]For additional details consult [9] ch. 6.5.

- **Underfitting**: high error rate on the training set
- **Overfitting**: large gap between training error and generalization error

We can control whether a model is more likely to overfit or underfit by altering its **capacity**, that is its ability to fit certain variety of functions – e.g. linear regression as a lower capacity than polynomial regression – and one way to control the capacity of a learning algorithm is by choosing its **hypothesis space**: the set of functions that the learning algorithms is able to approximate [9]. In the next subsections we go into the details of such procedures.

Regularization refers to strategies designed to reduce the generalization error, often paying the price of higher training error. They can all be seen as a way to reduce the capacity of the model. In general such strategies can be categorized into two classes:

1. Strategies that put extra constraints on a learning model (e.g. adding restrictions on parameters' values or on the activation functions)
2. Strategies that add extra terms in the objective function

These constraints and penalties can be generally interpreted as encoding some specific kind of prior knowledge or expressing preferences for simpler model classes in order to improve generalization reducing the threat of overfitting.

In the context of deep learning, most regularization strategies are based on regularizing estimators that have the effect of increasing bias while reducing variance. As usual, the bias of an estimator is defined as $\mathrm{Bias}(\hat{\theta}) = \mathrm{E}_{\hat{p}_{data}}(\hat{\theta} - \theta)$ which defines the expected error of the estimator, while the variance $\mathrm{V}(\hat{\theta})$ defines the error in the estimation stemming from the sensitivity to small fluctuations in the sample observed. High bias can cause an algorithm to mistake the mapping between features and target outputs (underfitting), while high variance can cause an algorithm to model the random noise in the training data rather than the intended outputs (overfitting). Therefore, an effective regularizer is one that is able to find the right trade-off, reducing variance significantly while not overly increasing the bias.

Via regularization we want to restrict at minimum the capacity of the model to include the data-generating process ruling out other possible candidate data-generating processes in the scope of the model capacity (reducing the variance). This conditions of the model capacity should not be too restrictive: the risk is to rule out of the scope of the model capacity the true data-generating process (bias). Unfortunately, we almost never have access to the true data-generating process and thus we can never be completely sure that the class of models estimated includes the true generating process.

2.2.1 Parameter Norm Penalties

A commonly used example of regularization approach belonging to the second class of strategies are the **parameter norm penalties**. These approach entails limiting the capacity of the model by adding a penalty in the loss function $J(\theta)$ proportional to the *parameter norm* $\Omega(\theta)$. Note that usually for neural networks the norm is computed only on the weights W, while the biases b are left unregularized. The regularized cost function can be written as

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(\theta) \tag{2.5}$$

where $\alpha \in [0, \infty)$ weights how the norm penalty affects the regularized objective function relative to the standard loss function. In neural networks, usually, the penalties are separated per layer and a different α is used for each layer. In practical applications, due to computational costs, the definition of Ω used is often the same for each layer. Different definitions of Ω result in different solutions being preferred. The two widely known definitions used are:

L^2-norm. In this case we have $\Omega(\theta) = \frac{1}{2}\|\theta\|_2^2$, commonly known as **weight decay**. Therefore the regularized loss becomes

$$\begin{aligned}\tilde{J}(\theta) &= \frac{\alpha}{2}\theta^\top\theta + J(\theta)\\ \nabla_\theta\tilde{J}(\theta) &= \alpha\theta + \nabla_\theta J(\theta)\end{aligned}$$

defining the update rule

$$\theta^{new} \leftarrow \theta - \lambda\big(\alpha\theta + \nabla_\theta J(\theta)\big)$$

The effect of the regularization term is to shrink θ by a constant factor on each step, just before performing the usual gradient update.

L^1-norm. In this case we have $\Omega(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$, obtaining

$$\begin{aligned}\tilde{J}(\theta) &= \alpha\|\theta\|_1 + J(\theta)\\ \nabla_\theta\tilde{J}(\theta) &= \alpha sign(\theta) + \nabla_\theta j(\theta)\end{aligned}$$

In this case, the gradient does not scale linearly as with the L^2-norm definition, but it is a constant factor with a sign equal to the sign of θ. The gradient so defined has no longer an algebraic solution and must be approximated. It can be shown [9] that L^1 regularization results in a solution that is more sparse, i.e. parameters will tend to have optimal value of zero. This can be interpreted as a form of variable selection mechanism. The LASSO is an example of this mechanism employed in linear models training.

The parameter norm penalties regularization can also be interpreted as MAP Bayesian inference: e.g. L^2 regularization is equivalent to MAP Bayesian inference with a Gaussian prior on θ.

2.2.2 Early Stopping

Often when training neural network with sufficient representational capacity, a signal that the model is overfitting is that training error decreases steadily over time, but validation set error begins to rise again. The validation set is used to evaluate a given model, like the test set, frequently during training. It corresponds to a small portion of the data that are mainly used to fine-tune the model hyperparameters. Hence the model “occasionally” sees the validation data, but never learns from them.

In principle, we can expect that minimizing the validation set error would also lead to better test error. Therefore, rather than stopping the optimization algorithm when the training error is minimum, it is possible to stop it earlier, when the validation error is the least, i.e. we use the parameters values in correspondence of the least validation error, rather than the latest values returned by the optimization algorithm.

This strategy is known as early stopping. In practise, the optimization algorithm is instructed to stop when no parameters have improved over the best recorded validation error for some pre-specified number of iterations. In deep learning, this is the most commonly used regularization strategy. In this way, we control how many times the optimization algorithm can run to fit the training set, limiting the threat of overfitting. This strategy does not require any change in the underlying procedure, loss function, or parameter set.

2.2.3 Sparse Representations

Another approach to regularize neural networks is to impose a penalty on the activation functions of the units by introducing sparseness. Essentially, the model is brought to prefer configurations in which more hidden units are set to zero; this reduces the capacity of the model. Indirectly, it is like imposing a complex penalization on the model parameters, As mentioned in subsection (§2.2.1), also L^1-penalization introduces sparseness in the parameters values.

Sparseness is induced in the representations, i.e. components of the hidden layer h_l are set to zero, not in the components of the parameters W_l or b_l. In practice, this regularization is applied to the model via adding a parameter norm penalty over the dimension of the hidden units in the loss function

$$\tilde{J}(\theta) = J(\theta) + \alpha\Omega(h), \quad \alpha \in [0, \infty)$$

Note how the argument of Ω is now the h vector and not θ.

2.2.4 Bagging and Ensemble Methods

The underlying idea is to reduce generalization error combining different models. In practise it is performed by training different models separately and for each input, X_i, each of these models collect the proposal $f_i(X_i)$. Techniques using this strategy, known as *model averaging*, go under the label of **ensemble methods**.

On average the ensemble performs at least as well as any of its members. In case the errors made by the individual models are independent, the ensemble performs significantly better than them.

A particular example is the **bagging** method, short for "bootstrap aggregating". This approach allows the very same model to be reused, instead of fitting different models. In practise k dataset with the same number of observations of the training set are constructed by sampling with replacement from the training set. Model i is trained on dataset $i \in \{1, \ldots, k\}$. Given that the same model is used, the difference among the models in the ensemble is due to the different k "artificial" dataset on which the model is trained.

2.2.5 Dropout

Dropout is a type of ensemble method, but for the importance it will have in this thesis – and in practise – it deserves a dedicated subsection.

Bagging entails training the same model multiple times on different training sets. For large neural networks the computational cost soon skyrockets. Dropout provides a computationally inexpensive approximation to train and evaluate a *bagged* ensemble. In particular, dropout trains the ensemble consisting of all subnetworks that can be formed by removing non-output units from the underlying base network. The number of submodels, thus, grows exponentially in the number of units. Switching off a unit – in the input layer or hidden layers – is performed by multiplying it by zero. Dropout applied to input units serves as a form of variable selection [12, 23], interpretable as Bayesian ridge regression [17]. When applied to hidden layers it regularizes the choice of the number of hidden units in a layer. Once a variable from a layer is dropped, all terms above it in the network also disappear.

In practise, dropout removes units in the input layer and/or in the hidden layers randomly with a given probability p, different for each layer. Training with dropout is performed using stochastic optimization methods like SGD. Each time a minibatch is created, for each layer, a binary vector ζ_l is randomly sampled, where each component is distributed as a $bernoulli(p_l)$. The number of components in the binary vector matches the number of units in the layer. This vector is usually called **mask**. The components of ζ are sampled independently of each other. The probabilities p_l are hyperparameters fixed before training begins. Therefore the original input layer becomes $\tilde{x} = x \odot \zeta_0$, where $\odot$ represents the Hadamard (or element-wise) product. Similarly, each hidden layer becomes $\tilde{h}_l = \zeta_l \odot h_l,\ l = 1, \ldots, L$. The same values of the binary vector are used during optimization, namely when performing back-propagation.

Let $J(\theta, \zeta)$ define the cost of the model defined by parameters θ and the mask ζ. Marginalizing over the randomness, the objective becomes $\mathrm{E}_{\zeta \sim \mathrm{ber}(p)} J(\theta, \zeta)$. The expectation contains exponentially many terms, but an unbiased estimate of its gradient can be obtained by sampling and using only certain values of $\{\zeta_0, \ldots, \zeta_L\}$. This differentiate dropout from bagging: in the latter case, each model is trained until convergence on its respective training set; in the former case, most models are not explicitly trained at all, but only a tiny fraction of the possible subnetworks are trained for a single step, and the parameter sharing, across models with the same active units, causes the remaining subnetworks to arrive at good settings of the parameters.

Chapter 3

Deep Probabilistic Learning

3.1 Deep Learning: The Probabilistic model

3.2 Bayesian Inference for Deep Learning

Chapter 4

Bibliography

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag.

Braun, Jurgen and Michael Griebel (2009). “On a Constructive Proof of Kolmogorov’s Superposition Theorem”. In: *Constructive Approximation* 30.3, p. 653.

Breiman, Leo (2001). “Statistical modeling: The two cultures”. In: *Statistical Science*.

Bryant, Donald W. (2008). “Analysis of Kolmogorov’s superposition theorem and its implementation in applications with low and high dimensional data”. In:

Cauchy, A. (1847). “Methode genénérale pour la résolution des systèmes d’equations simultanées”. In: *Comptes Rendus Hebd. Seances Acad. Sci.* 10.383, pp. 399–402.

Deng, L. and D. Yu (2014). “Deep Learning: Methods and Applications”. In: *Found. Trends Signal Process.* 7, pp. 197–387. ISSN: 1932-8346.

Duchi, J., E. Hazan, and Y. Singer (2011). “Adaptive subgradient methods for online learning and stochastic optimization”. In: *Journal of Machine Learning Research* 12, pp. 2121–2159.

G., Hinton (2012). “Neural networks for machine learning”. In: *Coursera video lecture*, pp. 2121–2159.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning*. 2nd ed. Springer Series in Statistics. New York, NY, USA: Springer New York Inc.

Hecht-Nielsen, R. (1987). “Kolmogorov’s mapping neural network existence theorem”. In: *Proceedings of the International Conference on Neural Networks III*, pp. 11–14.

Hinton, G. E. and R. R. Salakhutdinov (2006). “Reducing the dimensionality of data with neural networks”. In: *Science* 313.5786, pp. 504–507.

Kolmogorov, A. N. (1957). “On the representation of continuous functions of many variables by superpositions of continuous functions of one variable and addition”. In: *Doklay Akademii Nauk USSR* 14.5, pp. 953–956.

LeCun, Y., Y. Bengio, and G. E. Hinton (2015). “Deep learning”. In: *Nature* 521.7553, pp. 436–444.

Leni, P. E., Y. Fougerolle, and F. Truchetet (2011). “Kolmogorov Superposition Theorem and its application to multivariate function decompositions and image representation”. In: *IEEE*. Ed. by IEEE Computer Society.

Mitchell, Thomas M. (1997). *Machine Learning*. 1st ed. New York, NY, USA: McGraw-Hill, Inc.

Polson, N. and V. Sokolov (2017). “Deep Learning: A Bayesian Perspective”. In: *ArXiv e-prints*. `http://adsabs.harvard.edu/abs/2017arXiv170600473P`.

R.A., Jacobs (2008). “Increased rates of convergence through learning rate adaptation”. In: *Neural Networks* 1, pp. 295–307.

Rosenblatt, F. (1958). “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review*, pp. 65–386.

Schmidhuber, Jurgen (2015). “Deep learning in neural networks: An overview”. In: *Neural Networks* 61, pp. 85–117.

Sprecher, D. A. (1965). “On the structure of continuous functions of several variables”. In: 115.3. Ed. by Amer. Math. Soc, pp. 340–355.

— (1972). “A survey of solved and unsolved problems on superpositions of functions”. In: *Journal of Approximation Theory* 6.2, pp. 123–134.

Srivastava, N. et al. (2014). “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1, pp. 1929–1958.