Lectures on Machine Learning

Written and designed by: Riccardo Manetti
Gemma Martini

Freely inspired by: ML course (MD Computer Science)
Held by prof. Alessio Micheli (Unipi)

21st December 2020

Contents

1 Introduction **1**
- 1.1 What is machine learning? When to use it? . . . 1
- 1.2 Machine learning basics . . . 2
- 1.3 Generalization . . . 5
- 1.4 Evaluation . . . 6

2 Concept Learning **9**
- 2.1 Concept Learning Task . . . 9
- 2.2 Concept Learning as Search . . . 10
  - 2.2.1 General-to-Specific Ordering of Hypothesis . . . 11
- 2.3 Find-S Learning Algorithm . . . 12
- 2.4 Version Spaces and the CANDIDATE-ELIMINATION Algorithm . . . 13
  - 2.4.1 The LIST-THEN-ELIMINATE Algorithm . . . 13
  - 2.4.2 A More Compact Representation for Version Space . . . 13
  - 2.4.3 The CANDIDATE-ELIMINATION Learning Algorithm . . . 14
- 2.5 Inductive Bias . . . 15
  - 2.5.1 A Biased Hypothesis Space . . . 15
  - 2.5.2 Unbiased Learning . . . 15
  - 2.5.3 The Futility of Bias-Free Learning . . . 16

3 Linear models **19**
- 3.1 Regression . . . 19
- 3.2 Classification . . . 20
- 3.3 Models for regression and classification . . . 21
  - 3.3.1 Normal equations . . . 21
  - 3.3.2 Gradient descent . . . 22
- 3.4 Limitations . . . 23

4 *k*-nn **27**
- 4.1 Nearest-Neighbor Methods . . . 28
  - 4.1.1 K-nn variants . . . 29
  - 4.1.2 Linear vs. *k*-nn with various *k* values . . . 29
  - 4.1.3 The optimal Bayes . . . 29
  - 4.1.4 Limits of *k*-nn . . . 30

5 Neural Networks **31**
- 5.1 Introduction . . . 31
- 5.2 Perceptron learning algorithm . . . 33
- 5.3 Least Mean Squares vs. Perceptron Learning Algorithm . . . 36
- 5.4 Activation functions in Neural Networks . . . 37
  - 5.4.1 Derivatives of activation functions . . . 37
  - 5.4.2 LMS and activation functions . . . 38
- 5.5 More on neural networks . . . 39
- 5.6 Backpropagation . . . 41
- 5.7 Issues in training a neural network . . . 44
  - 5.7.1 Choosing the initial weights . . . 44
  - 5.7.2 Multiple minima . . . 44
  - 5.7.3 Online vs batch . . . 45

5.7.4 Learning rate . . . 45
5.7.5 Stopping criteria . . . 47
5.7.6 Overfitting and regularization . . . 47
5.7.7 Number of hidden units . . . 49
5.7.8 Input scaling and Output representation . . . 51
5.7.9 More on loss functions . . . 51
5.8 Pro and cons of Neural Networks . . . 52
5.9 Convolutional neural networks . . . 53

6 More on Neural Networks **55**
6.1 Deep Learning . . . 55
6.1.1 Hidden representation . . . 57
6.1.2 Localist & distributed representation . . . 58
6.2 Extreme Learning Machine . . . 61
6.3 Unsupervised Learning . . . 62
6.3.1 Vector quantization and clustering . . . 63
6.3.2 Self Organizing Map . . . 64
6.4 Recurrent Neural Networks . . . 66
Appendix 6.B Boltzmann Machines . . . 67

7 Validation and Statistical Learning Theory (SLT) **69**
7.1 Validation . . . 69
7.1.1 More on hyperparameters selection . . . 73
7.1.2 Variants . . . 73
7.2 Statistical Learning Theory . . . 74

8 Bias-Variance **77**
8.1 Bias-variance tradeoff . . . 77
8.2 Ensemble learning . . . 79

9 Support Vector Machines (SVM) **81**
9.1 Introduction . . . 81
9.2 Support Vector Machine for binary classification . . . 83
9.2.1 Hard margin . . . 83
9.2.2 Soft margin . . . 84
9.2.3 As kernel machine . . . 86
9.3 Support Vector Machine for non-linear regression . . . 88
Appendix 9.A Math of the SVM . . . 89
Appendix 9.B Dual form . . . 90
Appendix 9.C SVM for kernel transformation . . . 90

10 Structured Domain Learning **91**
10.1 Overview and motivation . . . 91
10.2 Recursive approaches for trees . . . 93
10.3 Analysis and moving to graphs . . . 94

11 Bayesian Networks **95**
11.1 Introduction . . . 95
11.2 Parameters learning with fully observed variables . . . 98
11.2.1 Continuous problems . . . 100
11.2.2 Learning Naive Bayes classifiers . . . 101
11.3 Parameters learning with hidden variables . . . 102
11.3.1 Learning Gaussian Mixture Models . . . 103
11.4 Recap on Bayesian Learning . . . 104

Bibliography **105**

1. Introduction

1.1 What is machine learning? When to use it?

Machine learning is a research field that uses adaptive and statistical techniques to build softwares able to learn from examples.

Machine learning friendly problems are those where it is not needed to provide rules that model the functioning, but indeed we have many examples available.

Machine learning techniques are applied to real world systems in new interdisciplinary areas (such as pattern recognition, computer vision, robotics, ...).

> **Example:** Recognition of handwritten digits
> The input is a collection of images of handwritten characters and the problem is to build a model (i.e. a function) that outputs the "class" to which a certain character belongs to.

> **Example:** Face recognition
> The input is a collection of photos (belonging to more than 4k identities) and the task it to be able to guess if two pictures show the same person. DeepFace answers correctly to 97.5% of the queries.

In *machine learning* we may have available a set of couples $(input, output)$, that are used to train the machine on the problem (**supervised models**); conversely, the output labels may not be present sometimes and for solving such tasks we make use of **unsupervised models**.

> **Mantra**
> Anything that is powerful may be dangerous (by misuses) but the ultimate aim of AI/ML is to bring benefits to people by solving big and small problems: accelerate human progress, to add intelligence for any other science field.
> "AI/ML aims to augments our abilities enhancing our humanness in unprecedent ways".

Modern Machine Learning moves from a collection of heuristic methods to the construction of general conceptual frameworks.
The aim of machine learning is three-fold:

- as AI methodology, its objective is to build adaptive/intelligent systems;
- as statistical learning, it aims to build powerful predictive systems for intelligent data analysis;
- as computer science method for innovative application areas, its objective is to use models as a tool for complex (interdisciplinary) problems.

In practice, we are interested in: 1) learning how to develop new machine learning models, 2) learning to apply current state-of-the-art methodologies for problems in other interdisciplinary fields.

As a wrap up, the aim of *machine learning* is to "guess" functions (univariate or multivariate), that map to discrete values (**classification**) or continuous values (**regression**). Changing the perspective, *machine learning* tasks can be divided into two groups: supervised and unsupervised. The **supervised** tasks are those that make use of a *training set* (a dataset of labelled inputs) which allows us to build a **model** (i.e. what defines the set of candidate functions) and specialize it. **Unsupervised** learning, conversely,deals with extracting features of the observed world, without using examples. In Figure 1.1 a pictorial representation of such division.

Machine learning approaches are useful whenever a theoretical study is missing or there are uncertain, noisy or incomplete data. For example, the task of face recognition is a process that is easy to perform for a

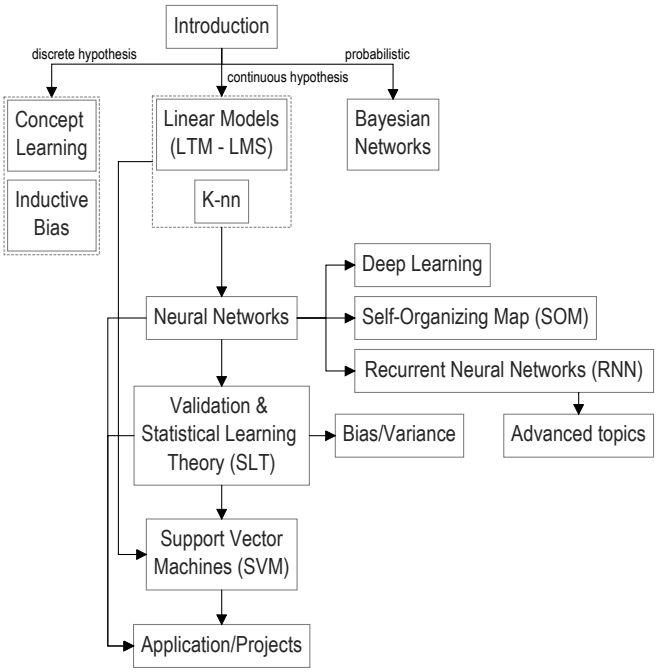


Figure 1.1: Schema of machine learning techniques.

human being, but it is not trivial to decode into an algorithm. Moreover, some tasks require a certain level of "personalization", e.g. spam filters recognize as spam different topics according to the user's tastes.

It is important to stress that the crucial task of humans in machine learning is to provide representative data and to allow some tolerance to the precision of results.

1.2 Machine learning basics

Mantra

In machine learning, the process of *learning* corresponds to finding a good approximation of an unknown function from examples.

Definition 1.2.1 (Machine learning)**.** *Machine learning* studies *and proposes methods to* build *(infer)* dependencies, functions, hypotheses *from examples of observed data. Such a model should satisfy the following:*

- *fitting the known examples;*
- *being able to generalize, with reasonable accuracy for new data, according to verifiable results and under statistical and computational conditions and criteria;*
- *considering the expressiveness and algorithmic complexity of the models and learning algorithms.*

A machine learning system is usually made of a set of data, a task, a model, a learning algorithm and finally by some validation techniques. All these components are described below.

Example:

In the following table, a list of examples of machine learning problems, showing the input x, the output $f(x)$ and the training set if present.

	x	$f(x)$	**TR** $\langle x, f(x)\rangle$
Handwriting Recognition	data from pen motion	letter of the alphabet	
Disease diagnosis	properties of patient (symptoms, lab tests)	disease (or maybe, recommended therapy)	database of past medical records
Face recognition	bitmap picture of person's face	Name of the person	
Spam Detection	email message	Spam or not spam	
Protein folding	sequence of amino acids (type: strings of variable length)	sequence of atoms' 3D coordinates (type: sequence of 3D vectors)	known proteins
Drug design	a molecule (type: a graph or a relational description of atoms or chemical bonds)	binding strength to HIV protease (type: a real number)	molecules already tested

Data

The data fed to the machine should represent the world, or equivalently the available facts, obtained through experience. Data can have different types:

- **Flat.** The data are represented via fixed-size vectors of features, where the different entries may happen to have different types. In such scenario, we will use the following notation:

  l for the number of examples;

  n for the number of features;

  x_j is the value of the j-th attribute of the vector x;

  $\mathbf{x}_p$ (**bold**) refers to the input vector of the p-th pattern.

- **Structured.** Data appears to have a more involved structure: for example sequences, lists, trees, graphs, multi-relational data, ...

Sometimes, it may be useful to perform a *feature selection* process, in order to select only a subset of informative features. The aim of such operation is to provide an optimal representation for a learning process;

Definition 1.2.2 (Noise)**.** *We term **noise** the addition of external factors to the stream of target information (signal); due to randomness in the measurements.*

Definition 1.2.3 (Outliers)**.** *We define **outliers** unusual data values that are not consistent with most observations (e.g. due to abnormal measurements errors).*

Tasks

A task defines the purpose of an application. A first distinction among tasks is if the learning process is **supervised** (provided with examples) or **unsupervised** (where we have a set of unlabeled data and we aim to find *natural groupings*). Supervised tasks can be divided into two different categories:

- **Predictive.** Where the aim is to find a good approximation of a function. Such tasks are split into *classification* (discrete) and *regression* (continuous). In classification, $f(x)$ returns the correct *class* for x; in other words, we need to provide a boundary that correctly separates points in the space. Conversely, in regression we are interested in estimating a real-valued function, on the basis of a finite set of noisy samples;

  Definition 1.2.4 (Classification task)**.** *Let $D = (\mathbf{x}, \mathbf{y}) \in \mathbb{R}^d \times \{-1, 1\}^Y$, the objective of the classification task is to build a function h that estimates $\mathbb{P}(y_i|\mathbf{x} = \underline{\mathbf{x}}), \forall i = 1, \ldots, Y$.*

  Definition 1.2.5 (Regression task)**.** *Let $D = (\mathbf{x}, \mathbf{y}) \in \mathbb{R}^d \times \mathbb{R}^Y$, the objective of the classification task is to build a function h that estimates $\mathbb{E}(y_i|\mathbf{x} = \underline{\mathbf{x}}), \forall i = 1, \ldots, Y$, where $\mathbb{E}$ indicates the expected value.*

- **Descriptive.** Where the aim is to find subsets or groups of unclassified data.

Formally, we define a task from a *given set of training examples* with the aim of *finding* a good approximation to f, i.e. building an **hypothesis function** denoted as h that can be used for prediction of unseen data x'. In such a scenario, we will use the following notation:

target d is for numerical data;

target t is a categorical datum;

About tasks, we can say more:

- **Dimensionality reduction** (unsup. learning)**.** Where we work with a subset of features, so this is known also as feature selection.
- **Reinforcement Learning** (sup. learning)**.** Instead of labeling the data with flags, we can imagine an agent that performs actions in the world and get some feedbacks that help it to learn some policies.
- **Semi-supervised learning.** Where labeled data and unlabeled data can coexist.

Models

The model is the function that should be able to capture the relationships among data, in a language that is related to the tools used to collect the knowledge. More formally, a model defines the **hypothesis space** (class of functions that the learning machine can implement). When we define a model, we need the following four ingredients: training examples, target function, hypothesis and hypothesis space (e.g. linear models, symbolic/logical rules, probabilistic models).

Sadly, we cannot design a universal "best model" and this fact is stated in the **no free lunch theorem**: "*if an algorithm achieves superior results on some problems, it must pay with inferiority on other problems. In this sense there is no free lunch*".

Learning algorithm

A learning algorithm has the aim of *searching through the hypothesis space H* for the best hypothesis. It goes without saying that defining such a set H implies to restrict to some families of functions, hence incurring in a **bias**. Moreover, we are interested in evaluating the goodness of an approximation and this is obtained introducing the concept of **loss**.

Definition 1.2.6 (Loss)**.** *We term **loss function** a function that measures the "distance" of the model from the objective function.*

$$L(h(x), d)$$

Notice that a high value for the loss function implies that the model is not a good approximation. What follows is a list of common loss functions for different tasks:

- Regression, *squared error* (i.e. the square of the difference between the target and the output)

$$L(h(x_i), d_i) = (d_i - h(x_i))^2 \qquad \text{where } d_i = f(x) + e, \forall i = 1, \ldots, n$$

- Classification, where the loss function measures the classification error (the mean over the dataset provides the number of misclassified patterns)

$$L(h(x_i), t_i) = \begin{cases} 0 \text{ if } h(x_i) = t_i \\ 1 \text{ otherwise} \end{cases} \qquad \text{where e.g. } t_i \in \{0, 1\}$$

- Clustering (unsupervised learning), where by the inner product ($\bullet$) we have

$$L(h(x_i)) = (x_i - h(x_i)) \bullet (x_i - h(x_i))$$

- Density estimation

$$L(h(x_i)) = -\ln(h(x_i))$$

Definition 1.2.7 (Error or risk)**.** *We define **error** the expected value of the loss function. Formally,*

$$R_{emp} = \frac{1}{l} \sum_{p=1}^{l} L(h(x_p), d_p)$$

Validation techniques

"Validation" is the word used to express the process which is carried on during the training and testing of a machine learning algorithm. This process has the aim of evaluating all the trained models and provide estimates about the generalization capabilities (for further details see Chapter 7).

1.3 Generalization

As soon as a model is built, we are interested in evaluating how well it performs on new data. Such performances on new data are called **generalization capabilities**.

Definition 1.3.1 (Generalization error)**.** *The so-called* ***generalization error*** *measures how well (or how badly) the model predicts over new samples of data.*

The issues of machine learning are the following:

- Inferring general functions from known data: with finite data we cannot expect to find the exact solution;
- Work with a restricted hypothesis space: there is a bias in choosing the hypothesis space;
- Expressive power tradeoff: searching a balance between the expressiveness of the model and the computational complexity of the learning algorithms.

Definition 1.3.2 (Overfitting)**.** *We say that a learner* ***overfits*** *the data if there is another hypothesis which is not as close to the data but allows a better generalization. Somehow, an overfitting model "fits the noise". Formally, the* ***overfitting*** *phenomenon takes place whenever we selected a hypothesis $h \in \mathcal{H}$ having true error ε and empirical error E, but $\exists h' \neq h \in \mathcal{H}$ having $E' > E$ and $\varepsilon' < \varepsilon$.*

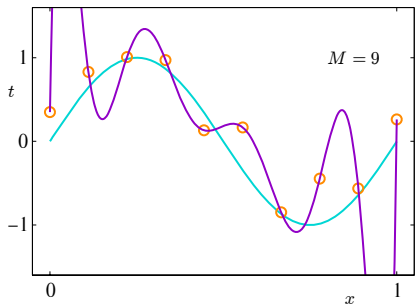


Figure 1.2: Here is an example of an overfitting model (violet) with respect to the original function (cyan), where the discrete training set is indicated with orange circles.

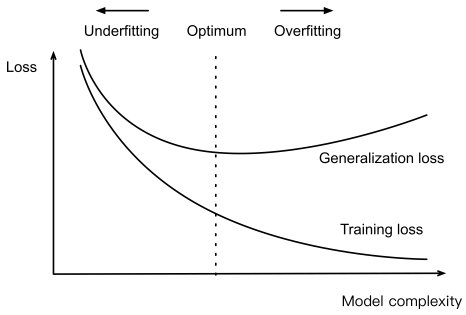


Figure 1.3: Here is a graphical representation of the areas where we have overfitting and underfitting.

Terminology

For polynomial functions, we use the parameter M to indicate the degree of the polynomial. E.g. $y(x, \mathbf{w})$ is a polynomial of degree M iff it can be written as $w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$.

Definition 1.3.3 (Underfitting)**.** *We term* ***underfitting*** *the phenomenon of choosing a model which is too simple for the problem under study.*

Definition 1.3.4 (Bias)**.** *A* ***bias*** *is a* ***preference****, that leads to considering only some solutions. We can find three different kinds of biases:*

- ***Language bias****: an assumption made on the model, which takes place when restricting the hypothesis space to some families of functions;*
- ***Research bias****: a restriction on the research algorithm, that allows to select only some hypotheses;*
- ***Inductive bias****: it also known as learning bias and represents the set of additional assumptions sufficient to justify the inductive inferences of the model as deductive inferences.*

Example: on biases . . .
A *language bias* is generated by the choice of approximating the objective function with a linear function.
A *research bias* is due to choosing an algorithm that finds a local maximum instead of another one (this is originated starting from different points).
A *inductive bias* takes place when we use a certain loss function with the belief that it represents correctly the distance from the learned hypothesis and the objective function.

1.4 Evaluation

As already discussed in previous sections, the aim of machine learning is to find the function $h \in \mathcal{H}$ that best models the training data, showing good generalization capabilities. These capabilities are assessed computing a risk function that represents the true error over all data and it is expressed formally as

$$R = \int L(d, h(x)) \, \mathrm{d}\mathbb{P}(x, d)$$

We are interested in minimizing such risk, but from a computational point of view, trying to maximize an integral is not feasible. It is for this purpose that we introduce the so-called *empirical risk* R_{emp} which is defined as

$$R_{\text{emp}} = \frac{1}{l} \sum_{p=1}^{l} (d_p - h(x_p))^2$$

It is subject of SLT(Section 7.2) providing the proof for the analytical upper-bound R_{emp} to the risk R.
How is it possible to select the best model and assess its goodness on a new set of examples when we are in presence of a small amount of data? Two basic techniques that serve such aim are the following:

- **Simple hold-out** is a technique that partitions (partition → disjoint sets) the data set D into a (i) *training set* (TR), used to run the training algorithm, a (ii) *validation* or *selection set* (VL), used to select the best model (hyperparameters tuning) and a (iii) *test set* (TS), only used for model assessment. Sets (i) and (ii) are used during the development, while the (iii) is used to evaluate the goodness of the approximation;
- **K-fold cross validation** comes into play to exploit better our data, and works as follows:
    1. Split the data set D into k mutually exclusive subsets $D_1, D_2, \ldots, D_k$;

2. Train the learning algorithm on $D \setminus D_i$ and test it on D_i;

It can be applied for both VL or TS splitting. This process may be used for splitting training from validation or for splitting training set and validation set (usually called "development set") from the test set. These techniques require some hyperparameters (which value of k should we use?) and tuning them may be computationally expensive.

More details about how the process of validation is carried on can be found in Section 7.1.
A glimpse of the design cycle of a machine learning model is displayed in Figure 1.4.

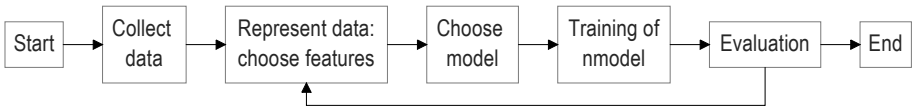


Figure 1.4: Flow chart for machine learning model construction.

Definition 1.4.1 (Confusion matrix)**.** *The **confusion matrix** is a pretty intuitive way of displaying true positives, false positive, true negatives and false negatives.*

		Predicted: Positive	Predicted: Negative	
Actual	Positive	True Positive (TP)	False Negative (FN)	**Sensitivity/Recall** $\frac{TP}{TP+FN}$
	Negative	Flase Positive (FP)	True Negative (TN)	**Specifity** $\frac{TN}{TN+FP}$
		Precision $\frac{TP}{TP+FP}$	**Negative Predictive Value** $\frac{TN}{TN+FN}$	**Accuracy** $\frac{TP+TN}{TP+TN+FP+FN}$

*The **sensitivity** (or recall) measures the proportion of actual positives that are correctly identified as such, while the **specificity** measures the proportion of actual negatives that are correctly identified as such, in other words it describes the capability to find the target.*

*The **positive** and **negative predictive values** (PPV or precision and NPV respectively) are the proportions of positive and negative results in statistics and diagnostic tests that are true positive and true negative results, respectively. The PPV and NPV describe the performance of a diagnostic test.*

***Accuracy** scores the percentage of correctly classified patterns.*

*For comparing different classifiers, a good visual technique is to draw the **area under the curve (AUC)**, as displayed in Figure 1.5.*

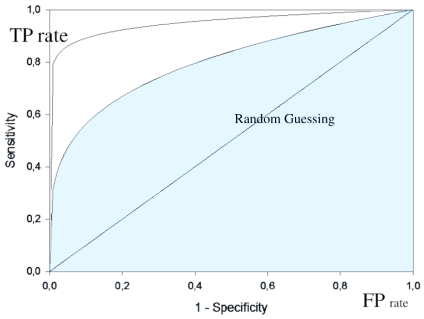


Figure 1.5: Example of AUC curve.

2. Concept Learning

The learning process involves acquiring general concepts from specific training examples (e.g.the general category of bird). Each concept can be thought of as a boolean-valued function defined over a larger set (e.g. a function defined over all animals, whose value is true for birds and false for other animals).

Definition 2.0.1 (Concept Learning)**.** *The aim of concept learning is to infer boolean function from training examples* (inputs and output)*.*

For example, let us assume to have n different binary attributes and consider each training example as a binary string of length n. This is an **ill posed** (inverse) problem: we may violate either existence, uniqueness and stability of the solution(s).
In the general case, given n binary attributes, there are 2^{2^n} distinct boolean functions (e.g. assuming to have only 4 binary attributes, there are $2^{16} = 65536$ possible boolean functions). For the large number of possible boolean functions we cannot figure out which one is correct until we have seen every possible input-output pair (*lookup table model*), so we need a learning algorithm that chooses the best function efficiently.

2.1 Concept Learning Task

Example:
Notice that this example is present as a use case throughout the chapter to explain and discuss concept learning. The objective of this learning task is to be able to guess the "days on which my friend Aldo enjoys his favorite water sport". The table Table 2.1 describes a set of example days, each represented by a set of attributes. The task is to learn to predict the value of the variable *EnjoySport* for an arbitrary day, based on the values of its attributes.

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Warm	Same	Yes
2	Sunny	Warm	High	Strong	Warm	Same	Yes
3	Rainy	Cold	High	Strong	Warm	Change	No
4	Sunny	Warm	High	Strong	Cold	Change	Yes

Table 2.1

We will use a simple representation in which each hypothesis consists of a conjunction of constraints on the instance attributes. For each attribute the constraints can be:

- a single record.value for the attribute,
- the symbol "?", which means that any value is acceptable,
- the symbol "$\emptyset$", which means that no value is acceptable.

If some instance $\mathbf{x}$ satisfies all the the constraints of hypothesis h, then h classifies $\mathbf{x}$ as a positive example ($h(\mathbf{x}) = 1$).
The **most general** hypothesis ("every day is a positive example") and the **most specific** hypothesis ("no day is a positive example") are represented as

$$\langle ?,?,?,?,?,? \rangle \qquad \text{and} \qquad \langle \emptyset,\emptyset,\emptyset,\emptyset,\emptyset,\emptyset \rangle.$$

The set of items over which the concept is defined is called the set of ***instance***, which we denote by X. The concept or function to be learned is called ***target concept***, which we denote c, and it can be any boolean-valued function defined over the instances X. Formally, $c : X \to \{0, 1\}$.

Concept learning starts from a set of ***training examples***, denoted as D, each of the ones is an instance x of X, along with its target concept value $c(x)$, and is usually denoted by $\langle x, c(x)\rangle$. We define an instance as *positive example* if $c(x) = 1$ or *negative example* if $c(x) = 0$.

We will use the symbol H to denote the set of all ***possible hypotheses*** that the learner may consider regarding the identity of the target concept. Each hypothesis $h \in H$ represents a boolean-valued function defined over X: $h : X \to \{0, 1\}$. The learner's goal is to find a hypothesis h such that $h(x) = c(x)$ for all $x \in X$ (we say that h is *consistent* with the example $\langle x, c(x)\rangle$).

Following this notation, we are ready to provide some definitions and divide the case in which an instance satisfies a hypothesis, and when a hypothesis is consistent with a set of training examples.

Definition 2.1.1 (Satisfy)**.** *For any given instance $x \in X$ and hypothesis $h \in H$, we say that x **satisfies** h if and only if $h(x) = 1$.*

Definition 2.1.2 (Consistent)**.** *An hypothesis h is **consistent** with a set of training examples D if and only if $h(x) = c(x)$ for each example $\langle x, c(x)\rangle$ in D.*

$$Consistent(h, D) \equiv \forall\langle x, c(x)\rangle \in D\ h(x) = c(x)$$

The *EnjoySport* concept learning task requires learning the set of days for which *EnjoySport* = *yes*, describing the set by a conjunction of constraints over the instance attributes. The concept learning task in the general form is given in Table 2.2.

- **Given**:
  - Instance X: Possible days, each described by the attributes
    - *Sky* (with possible values *Sunny*, *Cloudy* and *Rainy*)
    - *AirTemp* (with values *Warm* and *Cold*)
    - *Humidity* (with values *Normal* and *High*)
    - *Wind* (with values *Strong* and *Weak*)
    - *Water* (with values *Warm* and *Cold*)
    - *Forecast* (with values *Same* and *Change*)
  - Hypotheses H: Each hypothesis is described by a conjunction of constraints on the attributes *Sky*, *AirTemp*, *Humidity*, *Wind*, *Water* and *Forecast*. The constraints may be "?", "∅", or a specific value.
  - Target concept c: *EnjoySport* : $X \to \{0, 1\}$
  - Training examples D: Positive and negative examples of the target function (Table 2.1).
- **Determine**:
  - An hypothesis h in H such that $h(x) = c(x)$ for all x in H.

Table 2.2

The Inductive Learning Hypothesis

Notice that although the learning task is to determine an hypothesis h identical to the target concept c over the entire set of instances X, the only information available about c is its value over the training examples. Therefore, inductive learning algorithms can at best guarantee that the output hypothesis fits the target concept over the training data, hence we need the following to hold:

Definition 2.1.3 (Inductive learning hypothesis)**.** *Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.*

2.2 Concept Learning as Search

It is possible to view the concept leaning as a search through a large space of hypotheses, which goal is to find the hypothesis that best fits the training examples.

Considering that each example is described by n attributes, and each attribute a_i has δ_i different possible values, there are

$\prod_{i=0}^{n}(\delta_i)$	***distinct instances***
$\prod_{i=0}^{n}(\delta_i + 2)$	***syntactically*** *distinct hypotheses* (since we can have "?" or "∅")
$1 + \prod_{i=0}^{n}(\delta_i + 1)$	***semantically*** *distinct hypotheses* (from the fact that every hypothesis may contain one or more "∅" symbols — which classifies every instance as negative)

In the case of binary attributes, so we have $\delta_i = 2 \forall i = 1, \ldots, n$, hence there are 2^n distinct instances, 4^n syntactically distinct instances and $1 + 3^n$ semantically distinct instances.

Consider the instances X and the hypothesis space H in the *EnjoySport* learning task. Given that the attribute *Sky* has three possible values, and each of the other attributes has only two possible values, the instance space X contains exactly $3 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 96$ distinct instances. There are $5 \cdot 4 \cdot 4 \cdot 4 \cdot 4 \cdot 4 = 5120$ *syntactically distinct* hypotheses and $1 + (4 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3) = 973$ *semantically different* hypotheses within H. Finally there are $2^{96} \approx 10^{36}$ distinct concepts.

2.2.1 General-to-Specific Ordering of Hypothesis

Many algorithms for concept learning organize the search through the hypothesis space by relying on a structure that exists for any concept learning problem: a *general-to-specific ordering* of hypotheses.

Definition 2.2.1 (More general)**.** *Let h_j and h_k be boolean-valued functions defined over X. We say that h_j is **more-general-than-or-equal-to** h_k ($h_j \geq h_k$) if the set of inputs to which h_j answers positively is bigger than the one of h_k. Formally, if and only if*

$$\forall x \in X s.t. h_k(x) = 1 \text{ then } h_j(x) = 1$$

*Therefore, h_j is **strictly more-general-than** h_k ($h_j < h_k$) if and only if*

$$(h_j \geq h_k) \wedge (h_k \not\geq h_j)$$

Intuitively, the "generality" accounts for how often a certain function says 'yes' (1), hence quantifying how "tolerant" it is.

Now consider the following two hypotheses from the *EnjoySport* example

$$\langle h_1 = Sunny,?,?,Strong,?,? \rangle \qquad \text{and} \qquad \langle h_2 = Sunny,?,?,?,?,? \rangle$$

Consider also the sets of instances that are classified positive by h_1 and by h_2. Since h_2 imposes fewer constraints on the instance, it classifies more instances as positive (any instance classified positive by h_1 will also be classified positive by h_2). So we can say that h_2 is more general than h_1.

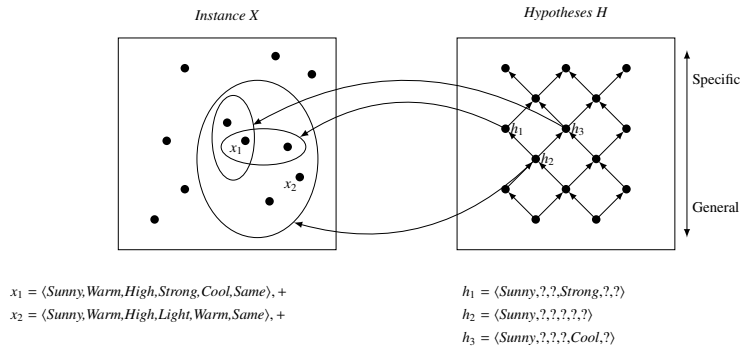


Figure 2.1: Each hypothesis corresponds to some subset of X.The arrows connecting hypotheses represent the *more-general-than* relation.The subset of instances characterized by h_2 subsumes the subset characterized by h_1 and h_3, hence h_2 is *more-general-than* both h_1 and h_3, while neither of these two hypotheses is more general than the other.

2.3 Find-S Learning Algorithm

The FIND-S algorithm illustrates one way in which the more-general-than partial ordering can be used to organize the search for an acceptable hypothesis. The search moves from hypothesis to hypothesis, searching from the most specific to progressively more general hypotheses along one chain of the partial ordering. To be more precise about how the partial ordering is used, consider Algorithm 2.1.

1: Initialize h to the most specific hypothesis in H
2: **for each** positive training instance x **do**
3: **for each** attribute constraint a_i in h **do**
4: **if** the constraint a_i is satisfied by x **then**
5: do nothing
6: **else**
7: replace a_i in h by the next more general constraint that is satisfied by x
8: Output the hypothesis h

Algorithm 2.1: The FIND-S algorithm.

Let us illustrate the FIND-S algorithm on the *EnjoySport* task. The first step of the algorithm is to initialize h with the most specific hypothesis in H

$$h \leftarrow \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

From the training examples from Table 2.1 we can see that none of the "$\emptyset$" constraints in h are satisfied by this example, so each of them is replaced by the next more general constraint that fits the example

$$h \leftarrow \langle Sunny, Warm, Normal, Strong, Warm, Same \rangle$$

This h is still very specific. The second training example forces the algorithm to further generalize h, this time substituting a "?" in place of any attribute value in h that is not satisfied by the new example.

$$h \leftarrow \langle Sunny, Warm, ?, Strong, Warm, Same \rangle$$

Upon encountering the third training example, the algorithm makes no change to h. To complete our trace of the algorithm, the fourth example leads to a further generalization of h

$$h \leftarrow \langle Sunny, Warm, ?, Strong, ?, ? \rangle$$

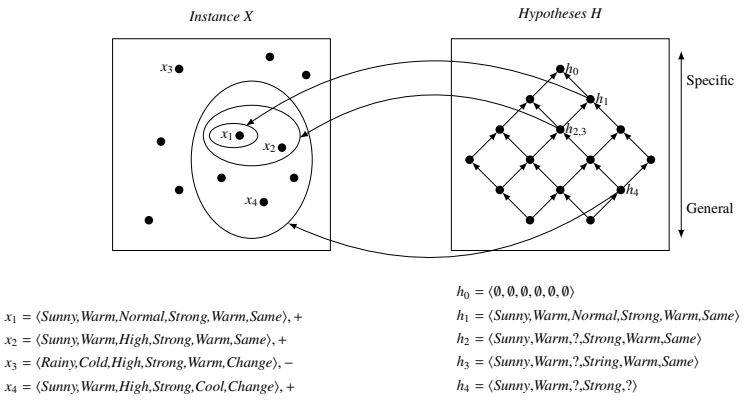


Figure 2.2: The hypothesis space search performed by FIND-S. The search begins with the most specific hypothesis in $H(h_0)$, then considers increasingly general hypotheses (from h_1 to h_4) as mandated by the training examples.

The key property of FIND-S algorithm is that for hypothesis spaces described by conjunctions of attribute constraints, FIND-S is *guaranteed to output the most specific hypothesis* within H that is consistent with the

positive training examples. Notice that FIND-S ignores every negative example. Its final hypothesis will also be consistent with the negative examples provided that the correct target concept is contained in H, and provided that the training examples are correct.

However, there are some issues with this algorithm:

- Although FIND-S will find a hypothesis consistent with the training data, it has no way to determine whether it has found the only hypothesis in H consistent with the data, or whether there are many other consistent hypotheses as well.
- In case there are multiple hypotheses consistent with the training examples, FIND-S will output only one hypothesis: the most specific.
- In most practical learning problems there is some chance that the training examples will contain at least some errors or noise, which impact the accuracy of the hypothesis.

2.4 Version Spaces and the CANDIDATE-ELIMINATION Algorithm

The CANDIDATE-ELIMINATION algorithm represents the set of *all* hypotheses consistent with the observed training examples. This subset of all hypotheses is called the ***version space*** with respect to the hypothesis space H and the training examples D, because it contains all plausible versions of the target concept.

Definition 2.4.1 (Version Space)**.** *The **version space**, denoted $VS_{H,D}$, with respect to hypothesis space H and training set D, is the subset of hypotheses from H consistent with the training examples in D.*

$$VS_{H,D} \equiv \{h \in H | Consistent(h, D)\}$$

We are now interested in finding the VERSION SPACE for a certain task in the most efficient way.

In the rest of the chapter, two algorithms will be proposed to the reader: the first one is trivial, but practically unfeasible, while the second one is pretty smart.

2.4.1 The LIST-THEN-ELIMINATE Algorithm

One obvious way to represent the version space is simply to list all of its members. This leads to a simple learning algorithm, which we might call the LIST-THEN-ELIMINATE algorithm, defined in Algorithm 2.2.

1: *VersionSpace* ← a list of every hypothesis in H
2: **for each** training sample $\langle x, c(x)\rangle$ **do**
3: remove from *VersionSpace* any hypothesis h for which $h(x) \neq c(x)$
4: Output the list of hypotheses in *VersionSpace*

Algorithm 2.2: The LIST-THEN-ELIMINATE algorithm.

The LIST-THEN-ELIMINATE algorithm first initializes the version space to contain all hypotheses in H, then eliminates any hypothesis found inconsistent with any training example. The version space of candidate hypotheses thus shrinks as more examples are observed, until ideally just one hypothesis remains that is consistent with all the observed examples.

In principle, the LIST-THEN-ELIMINATE algorithm can be applied whenever the hypothesis space H is finite. It has many advantages, including the fact that it is guaranteed to output all hypotheses consistent with the training data. Unfortunately, it requires exhaustively enumerating all hypotheses in H, an unrealistic requirement for all but the most trivial hypothesis spaces.

2.4.2 A More Compact Representation for Version Space

The CANDIDATE-ELIMINATION algorithm works on the same principles as the LIST-THEN-ELIMINATE algorithm: it exploits a much more compact representation of the version space, which is represented by its most general and least general members.

This algorithm represents the version space by storing only its most general members (G) and its most specific (S). Given only these two sets it is possible to enumerate all members of the version space.

Definition 2.4.2 (General boundary)**.** *The* ***general boundary*** *G, with respect to hypothesis space H and training data D, is the set of* maximally general *members of H consistent with D.*

$$G \equiv \{g \in H | Consistent(g, D) \wedge (\neg \exists g' \in H)[(g' > g) \wedge Consistent(g', D)]\}$$

Conversely,

Definition 2.4.3 (Specific boundary)**.** *The* ***specific boundary*** *S, with respect to hypothesis space H and training data D, is the set of* minimally general *members of H consistent with D.*

$$S \equiv \{s \in H | Consistent(s, D) \wedge (\neg \exists s' \in H)[(s > s') \wedge Consistent(s', D)]\}$$

In particular, we can show that the version space is precisely the set of hypotheses contained in G, plus those contained in S, plus those that lie between G and S in the partially ordered hypothesis space.

Theorem 2.4.1 (Version space representation theorem)**.** *Let X be an arbitrary set of instances and let H be a set of boolean-valued hypotheses defined over X. Let* $c : X \rightarrow \{0, 1\}$ *be an arbitrary target concept defined over X, and let D be an arbitrary set of training examples* $\{\langle x, c(x)\rangle\}$*. For all X, H, c and D such that S and G are well defined,*

$$VS_{H,D} = \{h \in H | (\exists s \in S)(\exists g \in G)(g \geq h \geq s)\}$$

To illustrate this characterization of version spaces, consider again the *EnjoySport* concept learning problem described in Table 2.2. Recall that given the four training examples from Table 2.1, FIND-S outputs the hypothesis

$$h = \langle Sunny, Warm, ?, Strong, ?, ?\rangle$$

In fact, this is just one of six different hypotheses from H that are consistent with these training examples. All six hypotheses are shown in Figure 2.3.

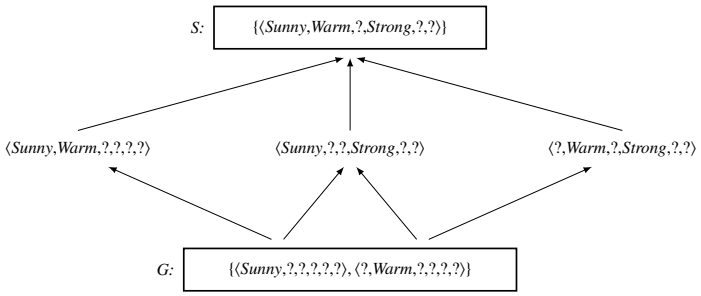


Figure 2.3: A graph-like representation of the vesion space for the *Enjoysport* concept learning problem and training examples described in Table 2.2.The version space includes all six hypotheses shown here, but can be represented more simply by S and G. Arrows indicate instances of the more-general-than relation.

2.4.3 The CANDIDATE-ELIMINATION Learning Algorithm

It begins by initializing the version space to the set of all hypotheses in H; that is, by initializing the G boundary set to contain the most general hypothesis in H

$$G_0 \leftarrow \{\langle ?, ?, ?, ?, ?, ?\rangle\}$$

and initializing the S boundary set to contain the most specific (least general) hypothesis

$$S_0 \leftarrow \{\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle\}$$

These two boundary sets delimit the entire hypothesis space, because every other hypothesis in H is both more general than S_0 and more specific than G_0. As each training example is considered, the S and G boundary sets are generalized and specialized, respectively, to eliminate from the version space an hypothesis found inconsistent with the new training example. This algorithm is summarized in Algorithm 2.3.

```
Initialize G to the set of maximally general hypotheses in H
Initialize S to the set of minimally general hypotheses in H
for each training example d do
    if d is a positive example then
        Remove from G any hypothesis inconsistent with d
        for each hypothesis s in S that is not consistent with d do
            Remove s from S
            Add to S all minimal generalization h of s such that h is consistent with d, and some member of G is more general than h
            Remove from S any hypothesis that is more general than another hypothesis in S
    else if d is a negative example then
        Remove from S any hypothesis inconsistent with d
        for each hypothesis g in G that is not consistent with d do
            Remove g from G
            Add to G all minimal specialization h of g such that h is consistent with d, and some member of S is more specific than h
            Remove from G any hypothesis that is less general than another hypothesis in G
```

Algorithm 2.3: The CANDIDATE-ELIMINATION algorithm using version spaces.

2.5 Inductive Bias

2.5.1 A Biased Hypothesis Space

Let us suppose that we want to assure that the hypothesis space contains the unknown target concept. The obvious solution is to enrich the hypothesis space to include every possible hypothesis.

> Consider again the *EnjoySport* example in which we restricted the hypothesis space to include only conjunctions of attribute values. Because of this restriction, the hypothesis space is unable to represent even simple disjunctive target concepts such as "*Sky* = *Sunny* or *Sky* = *Cloudy*".
> To see why there are no hypotheses consistent with the three examples given in Table 2.3, note that the most specific hypothesis consistent with the first two examples and representable in the given hypothesis space H is
>
> $$S_2 : \langle ?, \text{Warm}, \text{Normal}, \text{Strong}, \text{Cool}, \text{Change} \rangle$$

Example	*Sky*	*AirTemp*	*Humidity*	*Wind*	*Water*	*Forecast*	*EnjoySport*
1	Sunny	Warm	Normal	Strong	Cool	Change	Yes
2	Cloudy	Warm	Normal	Strong	Cool	Change	Yes
3	Rainy	Warm	Normal	Strong	Cool	Change	No

Table 2.3

2.5.2 Unbiased Learning

The obvious solution to the problem of assuring that the target concept is in the hypothesis space H is to provide an hypothesis space capable of representing *every* concept; that is, it is capable of representing every possible subset of the instances X.

Definition 2.5.1 (Power set)**.** *The set of all subsets of a set X is called the* ***power set*** *of X.*

The number of distinct subsets that can be defined over a set X containing $|X|$ elements is $2^{|X|}$.

> Now we reformulate the *EnjoySport* learning task in an unbiased way by defining a new hypothesis space H' that can represent every subset of instances: H' corresponds to the power set of X and one way to define it is to allow all arbitrary, conjunctions and negations of our earlier hypotheses. So the target "*Sky* = *Sunny* or *Sky* = *Cloudy*" could be described as

$$\langle Sunny,?,?,?,?,? \rangle \vee \langle Cloudy,?,?,?,?,? \rangle$$

Given the hypothesis space, we can use the Candidate-Elimination algorithm, but then the problem becomes that our learning algorithm is completely unable to generalize beyond the observed examples. To see why, suppose to have five examples: three positive (x_1,x_2,x_3) and two negative (x_4,x_5). The S boundary of the version space will contain the hypotheses that are the disjunction of the positive examples and the G boundary will consist of the hypothesis that rules out only the observed negative examples:

$$S : \{x_1 \vee x_2 \vee x_3\} \qquad G : \{\neg(x_4 \vee x_5)\}$$

The only examples that will be unambiguously classified by S and G are the observed training examples themselves. In order to converge to a single target concept, we will have to present every single instance in X as a training example.

Property 2.5.1. *Each unobserved instance will be classified positive by precisely half the hypotheses in the version space and will be classified negative by the other half.*

Proof. When H is the power set of X and x is some previously unobserved instance, then for any hypothesis h in the version space that covers x, there will be another hypothesis h' in the power set that is identical to h except for its classification of x. And of course if h is in the version space, then h' will be as well, because it agrees with h on all the observed training examples. ■

2.5.3 The Futility of Bias-Free Learning

Property 2.5.2. *A learner that makes no a priori assumptions regarding the identity of the target concept has no rational basis for classifying any unseen instances.*

If we take L to be the Candidate-Elimination algorithm, D_c to be the training data from Table 2.1, and x_i to be the instance

$$\langle Sunny,Warm,Normal,Strong,Cool,Change \rangle \textbf{ ?}$$

then the inductive inference performed in this case concludes that $L(x_i, D_c) = (EnjoySport = yes)$.

Definition 2.5.2 (Inductive bias)**.** *Consider a concept learning algorithm L for the set of instances X. Let c be an arbitrary concept defined over X, and let $D_c = \{\langle x, c(x)\rangle\}$ be an arbitrary set of training examples of c. Let $L(x_i, D_c)$ denote the classification assigned to the instance x_i by L after training on the data D_c. The **inductive bias** of L is any minimal set of assertions B such that for any target concept c and corresponding training examples D_c*

$$(\forall x_i \in X)[(B \wedge D_c \wedge x_i) \vdash L(x_i, D_c)]$$

Figure 2.4 schematize the fact that the inductive bias of Candidate-Elimination algorithm is "*The target concept c is contained in the hypothesis space H*".

One advantage of viewing inductive inferences systems in terms of their inductive bias is that it allows comparison of different learns according to the strength of the inductive bias they employ. Consider the following three learning algorithms and its corresponding bias.

1. Rote-Learner: learning corresponds simply to storing each observed training example in memory. Subsequent instances are classified by looking them up in memory. If the instance is found in memory, the stored classification is returned. Otherwise the system refuses to classify the new instance.This algorithm has no inductive bias. The classification is provided for new instances follow deductively from the observed training examples, with no additional assumptions required.
2. Candidate-Elimination algorithm: new instances are classified only in the case were all members of the current version space agree on the classification. Otherwise the system refuse to classify the new instance.This algorithm has a stronger inductive bias: that the target concept can be represented in its hypothesis space.

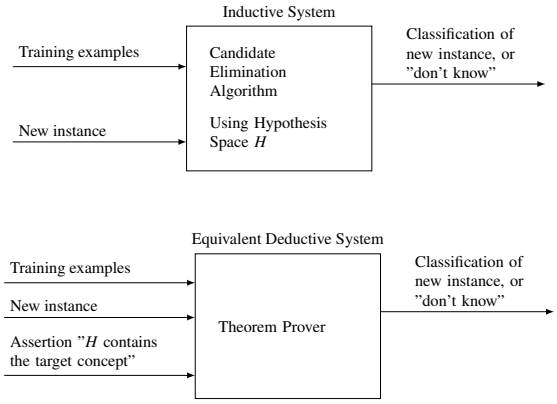


Figure 2.4: Modeling inductive systems by equivalent deductive systems. The input-output behavior of the CANDIDATE-ELIMINATION algorithm using a hypothesis space H is identical to that of a deductive theorem prover using the assertion "*H contains the target concept.*" This assertion is therefore called the *inductive bias* of the CANDIDATE-ELIMINATION algorithm. Characterizing inductive systems by their inductive bias allows modeling them by their equivalent deductive systems.

3. FIND-S algorithm: finds the most specific hypothesis consistent with the training examples. It then uses this hypothesis to classify all subsequent instances. This algorithm has an even stronger inductive bias. In addition to the assumption that the target concept can be described in its hypothesis space, it has an additional inductive bias assumption: that all instances are negative instances unless the opposite is entailed by its other knowledge.

3. Linear models

From this chapter onward, most of the models that will be described are able to deal with both classification and regression problems.

Terminology

Notice that we refer to the i-th component of the p-th pattern as $\mathbf{x}_{p,j}$ and the total number of patterns (examples) is denoted as l.

3.1 Regression

Do you recall?

As a reminder, we term **regression problem** the process of estimating a real-valued function on the basis of a finite set of noisy samples. We are given a set of known pairs $(x, f(x) + \text{random noise})$.

Univariate case

In this simple case, we are provided one input variable and one output variable and our model is supposed to have the following structure:

$$h_{\mathbf{w}}(x) = w_1 x + w_0 \tag{3.1}$$

where $x \in \mathbb{R}$ is the variable and $w_0, w_1 \in \mathbb{R}$ are the parameters we want to guess. In this case, we are interested in finding $\mathbf{w} \in \mathbb{R}^2$ that minimizes the error/empirical loss (i.e. allows for the best data fitting). In the general case, when $\mathbf{x} \in \mathbb{R}$, according to the least mean square error, we can estimate the loss of $h_{\mathbf{w}}$ (cfr. Definition 1.2.7) as

$$\text{Loss}(h_{\mathbf{w}}) = E(\mathbf{w}) = \frac{1}{l}\sum_{p=1}^{l}(y_p - h_{\mathbf{w}}(x_p))^2 = \frac{1}{l}\sum_{p=1}^{l}(y_p - (w_1 x_p + w_0))^2$$

It goes without saying that our objective is to find a local minimum of such function and for doing so we look for a stationary point, a point where the gradient has value 0. Formally,

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = 0 \qquad \forall i = 1, \ldots, n$$

where $n-1$ is the dimension of the space x belongs to (in the case of Equation (3.1) $n = 2$).In our example, we require both $\frac{\partial E(\mathbf{w})}{\partial w_0} = 0$ and $\frac{\partial E(\mathbf{w})}{\partial w_1} = 0$

Multivariate case

For the n-variate case we are looking for a vector $\mathbf{w} \in \mathbb{R}^n$ such that $\mathbf{w}^T\mathbf{x} + w_0 = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$ that represents weights assigned to the different features. Notice that the vector $\mathbf{w}$ is **orthogonal** to the line

induced by the model. In order to ease notation, we often refer to vectors belonging to $\mathbb{R}^{n+1}$, such that

$$\mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{pmatrix} \qquad \text{and} \qquad \mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{pmatrix}$$

3.2 Classification

Example:
In Figure 3.1 there are 200 points belonging to $\mathbb{R}^2$, generated from an unknown distribution: 100 points in each of the two classes. Our aim is to build a rule that can predict the color of the points, that are not yet in the plot, but will be submitted to the system.

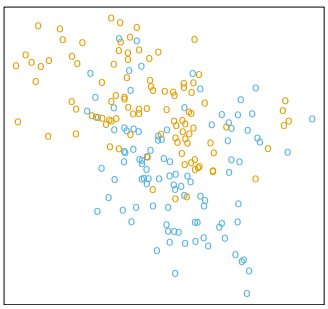


Figure 3.1: Example of a classification problem.

A simple strategy exploits linear models and use them to "draw a boundary" that delimits points belonging to a set or the other.

It goes without saying that in the multivariate case the boundary is an hyperplane that distinguishes two categories according to many features (dimensions). A pictorial example, may be observed in Figure 3.2.

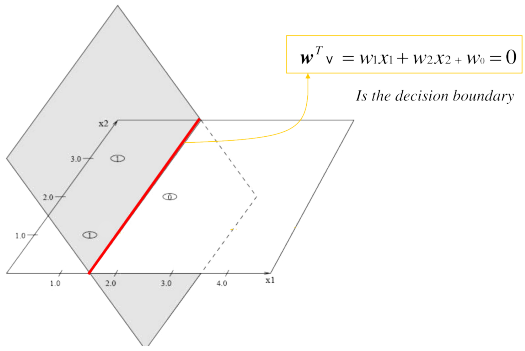


Figure 3.2: Here is an example of multivariate classification task, where the decision boundary (red line) can be used to classify correctly the points $(1, 0), (0.5, 3), (2, 2)$ as shown.

Notice that it is very often the case that there are more than one hyperplanes that correctly separate points. Conversely, it happens frequently that classification problems are not linearly separable.

Terminology

When we are studying a classification problem, we refer to the loss as $\mathbf{w}^T\mathbf{x}$ and we **do not use** $h(\mathbf{w})$, since in classification the following holds: $h(\mathbf{w}) = \text{sign}(\mathbf{w}^T\mathbf{x})$.

3.3 Models for regression and classification

We are going to present learning algorithms for both regression and classification tasks using a linear model. The first approach is based on *normal equations*, while the second one is based on the *gradient descent*.

Do you recall?

Given a set of l training examples $\{(\mathbf{x}_p, y_p)\}$ and a loss function measure L we want to find the weight vector $\mathbf{w}$ that minimizes the expected loss on the training data

$$\min_{\mathbf{w}} R_{\text{emp}} = \min_{\mathbf{w}} \frac{1}{l} \sum_{p=1}^{l} L(\mathbf{w}^T \mathbf{x}_p, y_p) = \min_{\mathbf{w}} \frac{1}{l} \sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 = \min_{\mathbf{w}} \frac{1}{l} \cdot \left\| \mathbf{y} - \mathbf{w}^T X \right\|^2$$

Our objective is to find a minimum point of the loss function and to do so we first need to find the points $\mathbf{w}$ in which the gradient of R_{emp} is 0.

Let us compute first the gradient of the empirical risk (notice that we prepone a product by the number of samples in the training set to ease notation):

$$\begin{aligned}
l \cdot \frac{\partial E(\mathbf{w})}{\partial w_j} &= l \cdot \frac{\partial \left(\frac{1}{l} \sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 \right)}{\partial w_j} \\
&= \cancel{l} \cdot \frac{1}{\cancel{l}} \frac{\partial \left(\frac{1}{l} \sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 \right)}{\partial w_j} \\
&= \frac{\partial \left(\sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 \right)}{\partial w_j} \\
&= \sum_{p=1}^{l} 2 \cdot \underbrace{(y_p - \mathbf{w}^T \mathbf{x}_p)}_{\delta_p} \cdot \frac{\partial \left(y_p - \mathbf{w}^T \mathbf{x}_p \right)}{\partial w_j} \\
&\overset{(1)}{=} -2 \cdot \sum_{p=1}^{l} \delta_p \cdot \frac{\partial \left(\mathbf{w}^T \mathbf{x}_p \right)}{\partial w_j} \\
&= -2 \cdot \sum_{p=1}^{l} \delta_p \cdot \frac{\partial \left(\sum_{k=1}^{n} w_k x_{pk} \right)}{\partial w_j} \\
&= -2 \cdot \sum_{p=1}^{l} \delta_p x_{pj}
\end{aligned}$$

where $\overset{(1)}{=}$ is obtained by substituting $\frac{\partial y_p}{\partial w_j} = 0$ and $\delta_p = y_p - \mathbf{w}^T \mathbf{x}_p$, which is the simplified notation for $\delta_p(\mathbf{w})$ indicating that δ_p is a function of $\mathbf{w}$.

3.3.1 Normal equations

This method is based on the idea of finding the solution of the equation $\frac{\partial E(\mathbf{w})}{\partial w_j} = 0$, using a direct process, rewriting the conditions.
Notice that we can forget from now on the term l, since $l \cdot \frac{\partial E(\mathbf{w})}{\partial w_j} = 0$ iff $\frac{\partial E(\mathbf{w})}{\partial w_j} = 0$.

Following the same reasoning, we can discard the -2 term as well, so the following holds:

$$\begin{aligned}\frac{\partial E(\mathbf{w})}{\partial w_j} &= \sum_{p=1}^{l} \delta_p \cdot x_{pj} = \sum_{p=1}^{l} x_{pj} \cdot \delta_p = \\ &= \sum_{p=1}^{l} x_{pj} \cdot (y_p - \mathbf{w}^T \mathbf{x}_p) = \\ &= \sum_{p=1}^{l} (x_{pj} y_j - x_{pj} \mathbf{w}^T \mathbf{x}_p) = \sum_{p=1}^{l} (x_{pj} y_j - x_{pj} {\mathbf{x}_p}^T \mathbf{w}) = \\ &= {\mathbf{x}_j}^T y - {\mathbf{x}_j}^T X \mathbf{w} = 0 \iff {\mathbf{x}_j}^T y = {\mathbf{x}_j}^T X \mathbf{w}\end{aligned}$$

On all training examples we get the so-called "normal equations":

$$X^T y = X^T X w$$

When $X^T X$ is not invertible, we may use the Moore-Penrose pseudoinverse X^+, that can be computed using the singular value decomposition.
This kind of methods are often not feasible, since computing the inverse of a large matrix is very expensive in terms of computational complexity.
In order to solve such issues, we introduce the gradient descent method.

3.3.2 Gradient descent

This class of methods allows an iterative procedure to reach a local minimum (i.e. linear function construction) from a certain point. Intuitively, we pick the direction for the next step as the opposite of the gradient in that point. Formally,

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \eta \cdot \Delta \mathbf{w}$$

where η accounts for the step size parameter (which is chosen to be between 0 and 1) and in machine learning is called **learning rate**.

Both the speed of convergence and the stability of the algorithm depend on the value η: a small value decreases convergence speed, while increases stability (speed/stability tradeoff). Notice that we need to start from an initial value for $\mathbf{w}$ and from such a point, we move locally towards a minimum. Formally, Algorithm 3.1 describes the procedure.

1: Initialize a small weight vector $\mathbf{w}$
2: Initialize $0 < \eta < 1$
3: **repeat**
4: **for each** $w_j \in \mathbf{w}$ **do**
5: $(\Delta \mathbf{w})_j = -\dfrac{\partial E(\mathbf{w})}{\partial w_j}$
6: $(\mathbf{w}_{\text{new}})_j = (\mathbf{w}_{\text{old}})_j + \eta \cdot (\Delta \mathbf{w})_j$
7: **until** convergence ($E(\mathbf{w})$ sufficiently small)

Algorithm 3.1: The Gradient Descent algorithm.

Definition 3.3.1 (Batch)**.** *We term **batch** the size of the training set (l in our notation), i.e. the number of samples to work through before updating the internal model parameters* $\mathbf{w}$.

Definition 3.3.2 (Mini-batch)**.** *We refer to **mini-batch** as set of training examples to work through before updating the internal model parameters. In our notation, the mini-batch size if denoted as mb.*

We can design an **online/stocastic version** where we update the weights for *each* new training example which is submitted to the algorithm ($mb = 1$). In this case, we are not interested in computing $\frac{\partial E(\mathbf{w})}{\partial w_j}$, but we compute

$$\frac{\partial E_p(\mathbf{w})}{\partial w_j} = -2 \cdot (y_p - {\mathbf{w}_p}^T \mathbf{w}) x_{p,j}$$

Definition 3.3.3 (Widrow-Hoff rule)**.** *We refer to the* ***Widrow-Hoff error correction rule*** *as the method of changing* $\mathbf{w}$ *proportionally to the error (distance from the target and the output). Intuitively, the bigger the error on the* j*-th output, the more the* j*-th component of the vector* $\mathbf{w}$ *gets changed. Formally,*

$$\Delta w_j = \sum_{p=1}^{l} x_{pj} \cdot (y_p - \mathbf{w}^T \mathbf{x}_p)$$

3.4 Limitations

For both for regression and classification tasks, we introduced two different kinds of biases: *language* (since H has be chosen as the family of linear functions) and *search* (we have been guided by minimizing the least squares error starting from a particular weights distribution).

For some unlucky cases, see Figure 3.3.

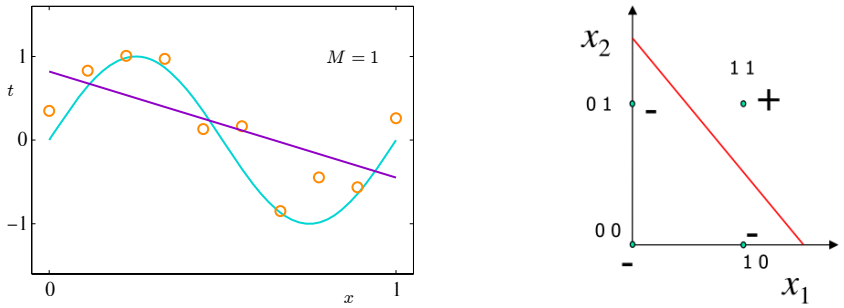


Figure 3.3: On the left-hand side, there is a regression task that is poorely solved using a linear model. On the right-hand side, a classification task that is poorely solved with a linear model.

Definition 3.4.1 (Linearly separable)**.** *In geometry, two set of points in a two-dimensional plot are* ***linearly separable*** *when they can be completely separated by a single line. In general,* $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ *are linearly separable if they can be separated by a* $(n-1)$*-dimensional hyperplane.*

In case of non-linearly separable problems, we can use a *conjunction* of many linear models.

In the regression task, we may perform a **linear basis expansion** (LBE) in order to increase the flexibility of the model:

$$h_{\mathbf{w}}(\mathbf{x}) = \sum_{k=1}^{K} w_k \Phi_k(\mathbf{x}) \tag{3.2}$$

Intuitively, the LBE augments the inputs features with some transformation and works using linear models in the new space.

We are interested in keeping the model linear in $\mathbf{w}$, but not in x, increasing the number of parameters (from n to K), so that we can model more complicated decision boundaries/regression relationships. Some examples of functions for Φ are:

- polynomial representation, $\Phi_j(\mathbf{x}) = {x_j}^2$ or $\Phi(\mathbf{x}) = x_j \mathbf{x}$
- non-linear transformation of single inputs, $\Phi_j(\mathbf{x}) \log x_j$ or $\Phi_j(\mathbf{x}) = \text{root}(x_j)$
- non-linear transformation of multiple inputs, $\Phi(\mathbf{x}) = \|\mathbf{x}\|$

Example:

An example of polynomial LBE is $\Phi : \mathbb{R}^2 \to \mathbb{R}^5$ such that $\Phi(\mathbf{x}) = \begin{pmatrix} x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{pmatrix}$.

As usual, introducing a linear basis expansion, we increase flexibility but at the same time we increase the risk of overfitting.
In order to keep the overfitting phenomenon under control, the following regularization comes in handy:

Definition 3.4.2 (Tikhonov regularization)**.** *The **Tikhonov regularization** works adding constraints (search bias) to the parameters (*$\mathbf{w}$*) in order to reduce dimensionality. Equivalently, some entries of* $\mathbf{w}$ *are forced to be small in absolute value.*

$$E(\mathbf{w}) = \sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 + \lambda \|\mathbf{w}\|_i^2$$

where λ *is the* regularization parameter *and the* $\|\mathbf{w}\|$ *is used as a regularization (penalty) term. If we use the* 1*-norm (*$\|\cdot\|_1$*) we talk about **lasso regression**, while the usage of the* 2*-norm is called **ridge regression**. It is possible to use a combination of the two, which is referred as **elastic nets**.*

Elastic nets

$$E(\mathbf{w}) = \sum_{p=1}^{l} (y_p - \mathbf{w}^T \mathbf{x}_p)^2 + \lambda_1 \|\mathbf{w}\|_1^2 + \lambda_2 \|\mathbf{w}\|_2^2$$

It goes without saying that Tikhonov regularization can be applied to the normal equations so we get $\mathbf{w} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y}$.

Lemma 3.4.1. *In terms of gradient, we are practically performing a* weight decay

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta \cdot \Delta\mathbf{w} - 2\lambda\mathbf{w}_{old}$$

Proof. As stated above, $\Delta\mathbf{w} = -\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$. Since the error is summed to a quantity we can apply the rule of the summation of derivatives and get

$$\Delta\mathbf{w}' = \Delta\mathbf{w} - \frac{\partial \lambda \|\mathbf{w}\|_2^2}{\partial \mathbf{w}} = \Delta\mathbf{w} - \lambda \cdot \frac{\partial \sum_{i=1}^{n} {w_i}^2}{\partial \mathbf{w}} = \Delta\mathbf{w} - \lambda \cdot 2\mathbf{w}$$

Plugging $\Delta\mathbf{w}$ into the weights update, we get

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \eta \cdot \Delta\mathbf{w}' = \mathbf{w}_{\text{old}} + \eta \cdot (\Delta\mathbf{w} - 2\lambda\mathbf{w}_{\text{old}}) \overset{*}{=} \mathbf{w}_{\text{old}} + \eta \cdot \Delta\mathbf{w} - 2\lambda\mathbf{w}_{\text{old}}$$

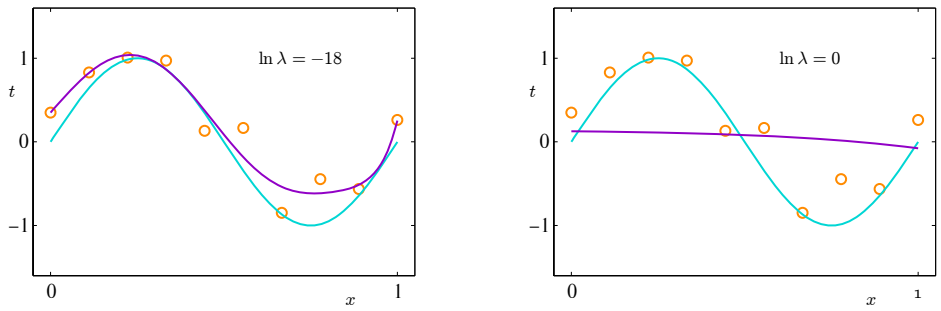
where $\overset{*}{=}$ holds because we prefer to discard the step size from the regularization term, since we prefer to keep η and λ independent.
Notice that in the case of $\|\cdot\|_1$ the proof is very similar, the only thing that changes is the derivative. ∎

One could ask himself: why should we choose to use the Tikhonov regularization instead of decreasing the degree of the polynomial? Because using the continuous parameter λ allows more flexibility, decreasing the search bias (allowing H to contain higher level function, possibly discarded by weight decay).

Notice that changing λ can rule the underfitting/overfitting cases (see Figure 3.4), because forcing some weights to ≈ 0 actually means that higher order terms are dropped and the degree of the polynomial decreases. Conversely, using a smaller value for λ, the degree of the polynomial keeps the same. According to the learning task, it may be the case that a higher/lower order polynomial is a good approximation or not.

As a fast forward, Lemma 7.2.1 ($R \leq R_{\text{emp}} + \varepsilon(1/l, VC - dim, 1/\delta)$) states that the actual risk is bounded from above by the empirical risk summed to a positive quantity, which is directly proportional to VC-dim. What is crucial to stress is that a higher regularization term forces some weights to ≈ 0 and if these terms are those of the higher degree components of a polynomial, the complexity of the model decreases and with it the VC-dim.

Such a bound becomes closer to the real value of the actual risk. As a conclusive observation, we can highlight that *linear basis expansion* is a technique used to implement more flexibility, while *regularization* controls complexity.



(a) $\lambda = 0.0000000152$ (b) $\lambda \approx 1$

Figure 3.4: In this picture we may observe that a too small value for λ leads to overfitting, while a too high value for λ brings to underfitting.

4. k-nn

Terminology

In this chapter, we denote $\mathbf{x}$ the p-th training example to ease notation (instead $\mathbf{x}_p$).

We are in the context of classification (see the example in the previous chapter represented in Figure 3.1) and our aim is to define a linear model to determine the class of each point.

Figure 4.1 shows a scatterplot of training data with the output class variable that has the values blue and orange. As mentioned before, there are 100 points in each of the two classes.

The linear regression model was fit to these data, with the response $h(\mathbf{x})$ coded as 0 for blue and 1 for orange. The fitted values from $h(\mathbf{x})$ are converted to a fitted class variable according to the rule

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T\mathbf{x} > 0.5 \\ 0 & \text{if } \mathbf{w}^T\mathbf{x} \leq 0.5 \end{cases}$$

The points in $\mathbb{R}^2$ classified as orange are those in $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} > 0.5\}$ and the two predicted classes are separated by the ***decision boundary*** $\{\mathbf{x} : \mathbf{w}^T\mathbf{x} = 0.5\}$, which is linear in this case.

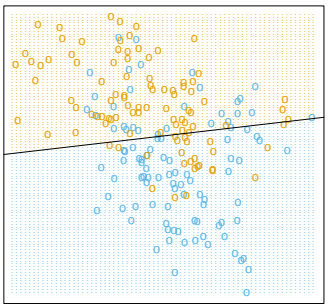


Figure 4.1: A classification example in two dimensions. The classes are coded as a binary variable (blue = 0, orange = 1), and then fit by linear regression. The line is the decision boundary defined by $\mathbf{w}^T\mathbf{x} = 0.5$. The orange shaded region denotes the part of input space that is classified as orange, while the blue region is classified as blue.

Consider the two possible scenarios:

1. The training data in each class are generated from bivariate Gaussian distribution with uncorrelated components and different means.
2. The training data in each class comes from a mixture of 10 low-variance Gaussian distributions.

In the first scenario, the linear regression line (computed through least squares algorithm) is almost optimal and the region of overlap is inevitable. While for the second scenario, a linear decision boundary is unlikely to be optimal, and in fact is not. The optimal decision boundary is nonlinear, and as such it is much more difficult to obtain.

A learning algorithm is based on timing, which can be of two types:

- **Eager**: analyze the training data and construct an explicit hypothesis.
- **Lazy**: store the training data and wait until a test data point is presented, then construct an ad hoc hypothesis to classify that one data point.

4.1 Nearest-Neighbor Methods

The k-nearest neighbour fit for $\underset{k}{\text{avg}}(\mathbf{x})$ is defined as follow:

$$\underset{k}{\text{avg}}(\mathbf{x}) = \frac{1}{k} \sum_{\mathbf{x}' \in N_k(\mathbf{x})} y_j \tag{4.1}$$

where $N_k(\mathbf{x})$ is the neighbourhood of $\mathbf{x}$ defined by the k closest points $\mathbf{x}'$ in the training sample and $y' = \mathbf{w}^T\mathbf{x}'$. Closeness implies the usage of a metric, which for the moment we assume to be Euclidean distance:

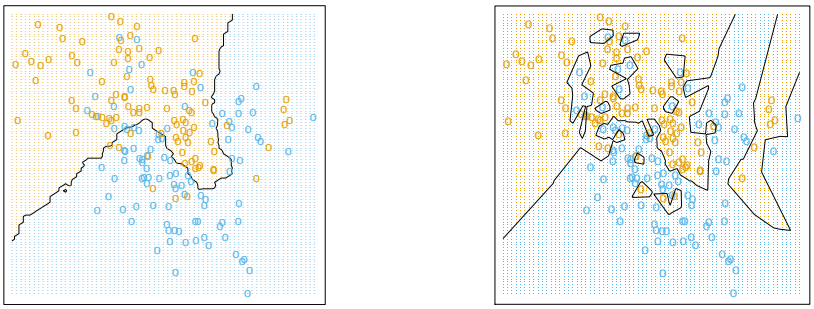
$$d(\mathbf{x}, \mathbf{x_j}) = \sqrt{\sum_{i=1}^{n} (x_i - x_i')^2} = \left\|\mathbf{x} - \mathbf{x}'\right\|$$

where x_{jt} corresponds to the t-th component of the j-th pattern.

If there is a clear dominance of one of the classes in the neighbourhood of an observation $\mathbf{x}$, then it is likely that the observation itself would belong to that class too. Thus, the classification rule is the **majority voting** among the members of $N_k(\mathbf{x})$. As before,

$$h(\mathbf{x}) = \begin{cases} 1 & \text{if } \underset{k}{\text{avg}}(\mathbf{x}) > 0.5 \\ 0 & \text{if } \underset{k}{\text{avg}}(\mathbf{x}) \leq 0.5 \end{cases}$$

for targets y that assume values 0 or 1.



(a) 15-nn (b) 1-nn

Figure 4.2: Here the same classification problem as in Figure 4.1. The classes are coded as a binary variable, then fit by k-nearest-neighbour averaging as in (4.1) with $k = 15$ in Figure 4.2(a) and $k = 1$ in Figure 4.2(b).

In Figure 4.2(a) we use the same training data as in Figure 4.1 and use 15-nearest-neighbour averaging of the binary coded responde as the method of fitting. We see that the decision boundaries that separate the blue from the orange regions are far more irregular (it is very flexible), and respond to local clusters where one class dominates.

Figure 4.2(b) shows the results for 1-nearest-neighbour classification: $h(\mathbf{x})$ is assigned the value y' of the closest point $\mathbf{x}'$ to $\mathbf{x}$ in the training data. The decision boundary is even more irregular than before. In this case the regions of classification can be computed relatively easily, and correspond to a ***Voronoi tessellation*** of the training data: each cell consisting of all points closer to $\mathbf{x}$ than to any other pattern; the segments of the Voronoi diagram are all the points in the plane that are equidistant to two patterns.

In Figure 4.2(a) we see that far fewer training observation are missclassified than in Figure 4.1. This should not give us too much comfort, though, since in Figure 4.2(b) none of the training data are missclassified.
It appears that k-nearest-neighbour fits have a single parameter, the number of neighbours k, we will see that the *effective* number of parameters of k-nearest-neighbour is N/k. To get an idea of why, note that if the neighbourhoods were non-overlapping there would be N/k neighbourhoods and we would fit one parameter in each neighbourhood.

4.1.1 K-nn variants

The k-nearest-neighbour algorithm allows some variations:

- **K-nn for multi-class**: return the most common class (denoted as v) amongst its k-nearest-neighbour

$$h(\mathbf{x}) = \arg\max_{v} \sum_{\mathbf{x}' \in N_k(\mathbf{x})} \varphi_{v,y'} \qquad \text{where} \qquad \varphi_{v,y'} = \begin{cases} 1 & \text{if } v = y' \\ 0 & \text{otherwise} \end{cases}$$

- **K-nn with weighted distance**: it can be useful to weight the contributions of the neighbours according to the proximity to the target pattern $\mathbf{x}$, so that the nearer neighbours contribute more to the average than the further ones. Notice that if $d(\mathbf{x}, \mathbf{x}')$ is 0, then y' is returned.

$$h(\mathbf{x}) = \arg\max_{v} \sum_{\mathbf{x}' \in N_k(\mathbf{x})} \varphi_{v,y'} \frac{1}{d(\mathbf{x}, \mathbf{x}')^2} \qquad \text{where} \qquad \varphi_{v,y_p} = \begin{cases} 1 & \text{if } v = y' \\ 0 & \text{otherwise} \end{cases}$$

4.1.2 Linear vs. k-nn with various k values

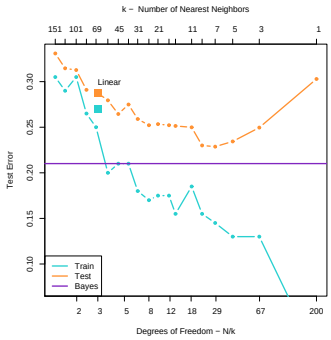


Figure 4.3: Misclassification curves for the simulation example used in Figures 4.1, 4.2(a) and 4.2(b), where $|TR| = 200$ and $|TS| = 10,000$. For k-nearest neighbour classifier the orange line represents the test arror and blue line the training error. The results for linear regression are the bigger orange and blue squares at three degrees of freedom. The purple line is the optimal Bayes error rate.

4.1.3 The optimal Bayes

The **Bayes classifier** says that we classify to the most probable class v, using the conditional (discrete) distribution $\mathbb{P}(v|\mathbf{x})$, for $\mathbf{x} \in \{C_1, \ldots, C_k\}$. Figure 4.4 shows the Bayes-optimal decision boundary for our simulation example. The error rate of the Bayes classifier is called the **Bayes rate**.

Notice that the k-nearest neighbour classifier directly approximates this solution. A majority vote in a nearest neighbourhood amounts to exactly this, except that conditional probability at a point is relaxed to conditional probability within a neighbourhood of a point, and probabilities are estimated by training-sample proportions.

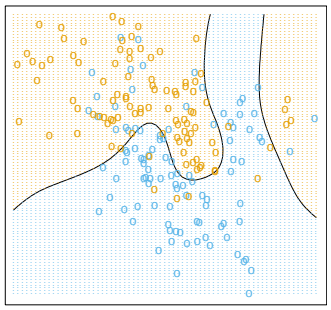


Figure 4.4: The optimal Bayes decision boundary for the simulation example of Figures 4.1, 4.2(a) and 4.2(b). Since the generating density is known for each class, this boundary can be calculated exactly.

4.1.4 Limits of k-nn

Computational cost

Note that k-nn makes the local approximation to the target for each new example to be predicted, so the computational cost is deferred to the prediction phase.

Such phase is costly both in time and space. For each new input, computing the distances from the test sample to all stored vectors, takes time is proportional to the number of stored patterns. It is possible to use a proximity search algorithm to optimize this computation, i.e. by indexing the patterns.

Curse of dimensionality

The k-nn method provides a good approximation if we can find a significant set of data close to any $\mathbf{x}$, with dense sampling, but there exist cases in which it fails. A typical case is when we have a lot of input variables: the **curse of dimensionality**. There are many manifestations of this problem and it is hard to find nearby points in highly dimensional spaces. We have low sampling density for high dimensionality data.

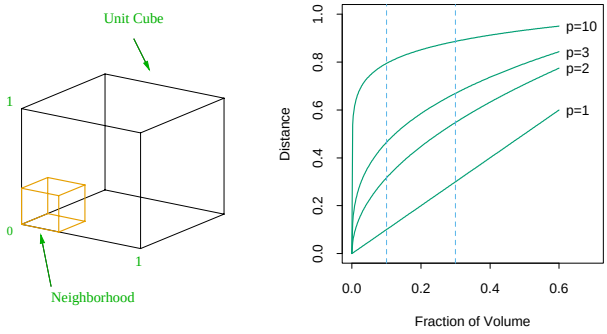


Figure 4.5: The curse of dimensionality is well illustrated by a subcubical neighbourhood for uniform data in a unit cube. The figure on the right shows the side-length of the subcube needed to capture a fraction r of the volume of the data, for different dimensions d. In ten dimensions we need to cover 80% of the range of each coordinate to capture 10% of the data.

Consider the nearest-neighbour procedure for inputs uniformly distributed in a p-dimensional unit hypercube, as in Figure 4.5. Suppose we send out a hypercubical neighbourhood about a target point to capture a fraction r of the observations. Since this corresponds to a fraction r of the unit volume, the expected edge (side) length will be $e_d(r) = r^{1/d}$ in a space with d dimensions (reverse: $r = (e_d)^d$). In ten dimensions $e_{10}(0.01) = 0.63$ and $e_{10}(0.1) = 0.80$, while the entire range for each input is only 1.0. So to capture 1% or 10% of the data to form a local average, we must cover 63% or 80% of the range of each input variable.

Another manifestation of this curse is that the sampling density is proportional to $l_{1/d}$, where d is the dimension of the input space and l is the sample size. Thus, if $l_1 = 100$ represents a dense sample for a single input problem, then $l_{10} = 10010$ is the sample size required for the same sampling density with 10 inputs.

Finally if the target depends on only few of many features in $\mathbf{x}$, we could retrieve a "similar pattern" with the similarity dominated by the large number of irrelevant features. Moreover, this fact grows with the dimensionality. A possible improvement is to weight features according to their relevance, i.e. stretching the axes along some dimensions. Wights can be searched by a (expensive) cross-validation approach or other approaches.

5. Neural Networks

5.1 Introduction

Far funzionare la bibliografia e sceglierne lo stile

Working on artificial neural networks has been motivated right from its inception by the observation that the human brain makes computations in an entirely different way from the conventional digital computer.
When we talk about neural networks we may refer to either

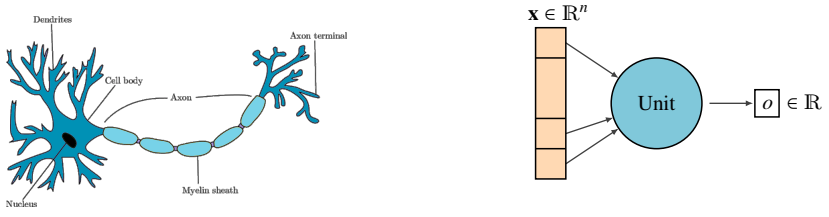
- study and model biological systems and learning processes, where the biological nature is essential, or
- effective ML systems and algorithms that take inspiration from the brain cells to make computations and predictions on real data (often loosing a strict biological realism).

In this chapter, we will deal with the second class, the so-called *artificial neural networks*.
The main features of neural networks are the following:

- they can learn from examples;
- they are universal approximators (see Theorem of Cybenko);
- they can deal with noise and incomplete data;
- they can handle continuous real and discrete data for both regression and classification tasks.

Definition 5.1.1 (Connectionism)**.** *We call **connectionism** the complex behaviour emerging from the interaction (interconnected networks) of simple computational units.*

Artificial neural networks imitate somehow the mechanisms of the human brain. In biology, excitatory or inhibitory inputs come from the dendrites and cause changes to the *resting potential* of the neuron (see Figure 5.1(a)). The graded potential coming from the various extremes of the dendrites will reach the axon following a decay due to the distance between the source of the signal and the tank. Such decay will impact the signal received by the trigger zone, based at the initial segment of the axon. The size of this potentials needs to be large enough to push the membrane of the trigger zone up over the threshold potential in order to lead to the generation of a new electric signal that is propagated along the axon (firing neuron).



(a) Mammal's neuron (b) Artificial neuron

Figure 5.1: How neurons look like

Following this behaviour, an artificial neural network is a union of **neurons** (or *nodes* or *units*), each of the ones takes several inputs (a vector in our case), which impact the neuron with a certain weight w, and outputs a value, see Figure 5.1(b).
In order to properly simulate the behaviour of neurons, artificial units compute a weighted sum of the inputs and apply a function, called **activation function** to map such value in another interval (threshold simulation).

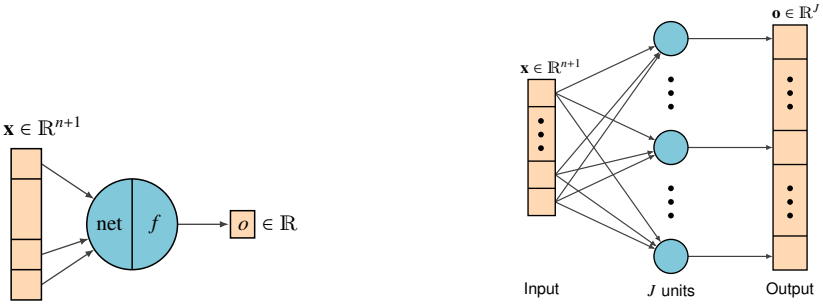
Definition 5.1.2 (Net input)**.** *We call **net input** and denote net the weighted sum of all the inputs of a certain*

unit. Formally,

$$net = \mathbf{w}^T\mathbf{x} = \sum_{j=1}^{n} w_j x_j$$

where $\mathbf{w}$ *is called* ***weights vector*** *or* ***vector of free parameters*** *and has the same dimension of* $\mathbf{x}$.

It is important to remark that the input vector of a unit is such that $x_1 = 1$, in order to allow the linear function $\mathbf{w}^T\mathbf{x}$ to have a bias, hence $\mathbf{x}, \mathbf{w} \in \mathbb{R}^{n+1}$.
The output of a single neuron is computed as $o = f_\sigma(\text{net}(\mathbf{x})) \in \mathbb{R}$, where the function σ is the unit's **activation function** (for further details see Section 5.4). A more detailed representation that takes into account what just presented can be found in Figure 5.2.



(a) A neuron as the composition of weighted sum and activation function

(b) One layer of neurons

Figure 5.2: Some more details about artificial neural networks

Definition 5.1.3 (Hidden unit)**.** *We term* ***hidden units*** *the functions* h_i *which are an intermediate product of a neural network (between input and output).*

Definition 5.1.4 (Perceptron)**.** *We term* ***perceptron*** *a single layer neural network, just like the one displayed in Figure 5.2(b).*

Example:

Let us focus on how to model logical functions using neural networks. In Table 5.1 the reader can find the truth tables of AND, OR and XOR logical operators.

AND		
x_1	x_2	$f(\mathbf{x})$
0	0	0
0	1	0
1	0	0
1	1	1

OR		
x_1	x_2	$f(\mathbf{x})$
0	0	0
0	1	1
1	0	1
1	1	1

XOR		
x_1	x_2	$f(\mathbf{x})$
0	0	0
0	1	1
1	0	1
1	1	0

Table 5.1: AND, OR and XOR truth tables.

Let us consider a generic neuron and its input net: $net = \mathbf{w}^T\mathbf{x} = \sum_{j=0}^{n} w_j x_j$.The target function f is approximated as

$$sign(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

Two nets (drawn as neural networks in Figure 5.3) that model exactly the logical AND and OR are:

- AND: $f(\mathbf{x}) \approx h(\mathbf{x}) = sign(1 \cdot w_0 + x_1 w_1 + x_2 w_2) = sign(x_1 + x_2 - 1.5)$
- OR: $f(\mathbf{x}) \approx h(\mathbf{x}) = sign(1 \cdot w_0 + x_1 w_1 + x_2 w_2) = sign(x_1 + x_2 - 0.5)$

(a) Network modelling logical AND

(b) Network modelling logical OR

Figure 5.3: Neural network structure of logical AND and OR.

Exclusive OR (Table 5.1) does not allow a linear separation, hence a perceptron cannot model XOR. To model such a logical operator, we may use a two layer network: let $h_1 = x_1 \cdot x_2$ and let $h_2 = x_1 + x_2$[a]. Our target function $x_1 \oplus x_2 = x_i \cdot \overline{x_2} + \overline{x_1} \cdot x_2$ can be written using h_1 and h_2 as $x_1 \oplus x_2 = \overline{h_1} \cdot h_2$. Informally, our problem becomes linearly separable in the new space, where the components of vectors refer to h_1 and h_2.
A pictorial representation of such a network is presented in Figure 5.4, but in table Table 5.2 the reader can find a more intuitive rewriting of such network.

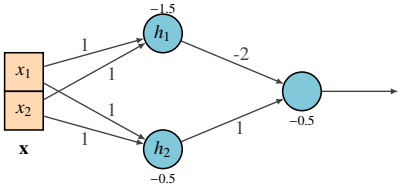


Figure 5.4: Neural network that models the XOR logical operator with function $f(x) \approx h(x) = sign(-2 \cdot h_1(x) + 1 \cdot h_2(x) - 0.5)$, where h_1 and h_2 are respectively the AND and the OR functions.

XOR					
x_1	x_2	$f(\mathbf{x})$	$h_1(\mathbf{x})$	$h_2(\mathbf{x})$	$h(\mathbf{x})$
0	0	0	$sign(0+0-1.5)$	$sign(0+0-0.5)$	$sign(-2\cdot 0+1\cdot 0-0.5)$
0	1	1	$sign(0+1-1.5)$	$sign(0+1-0.5)$	$sign(-2\cdot 0+1\cdot 1-0.5)$
1	0	1	$sign(1+0-1.5)$	$sign(1+0-0.5)$	$sign(-2\cdot 0+1\cdot 1-0.5)$
1	1	0	$sign(1+1-1.5)$	$sign(1+1-0.5)$	$sign(-2\cdot 1+1\cdot 1-0.5)$

Table 5.2: Explanation of XOR neural network.

[a] Notice that from now on we will refer to the AND operator using $\cdot$, while we will denote with $+$ the OR operator.

Theorem 5.1.1 (McCulloch and Pitts theorem)**.** *The following holds:*

- *Perceptrons can represent* AND, OR, NOT (NAND, NOR)*;*
- *Two levels of neural networks are enough to represent* each *boolean function;*
- *Single-layer neural networks cannot model problems which are not linearly separable.*

We are left with providing a formal recipe to modelling a function using the perceptron model.

5.2 Perceptron learning algorithm

The *perceptron learning algorithm* works as follows: first, we set an initial $\mathbf{w}_0$ * and a step size η. If a misclassified example $< \mathbf{x}_{\text{wrong}}, d_{\text{wrong}} >$ is found, then the weight vector $\mathbf{w}$ is updated summing to the old

*Where the subscript 0 indicates the step and not the component of the vector

value $d_{\text{wrong}} \cdot \eta \cdot \mathbf{x}_{\text{wrong}}$. This process continues until no point is misclassified.

Intuitively, this algorithm moves in the direction of minimizing the number of misclassified patterns. Before introducing the formal algorithm (Algorithm 5.1), let us make an example.

Example:

Let us take the linearly separable classification problem displayed in Figure 5.5 and set $\mathbf{w}_0 = \mathbf{0}$ and $\eta = 1$. The first separating line is $y = 0$. In order to check the misclassified points we proceed from $\mathbf{x}_1$ to $\mathbf{x}_{13}$. In particular we notice that $\mathbf{x}_2$ is misclassified by the line $y = 0$, hence we get

$$\mathbf{w}_1 = \mathbf{w}_0 + \eta \cdot d_2 \cdot \mathbf{x}_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} + (-1) \cdot \begin{pmatrix} 1 \\ 8 \\ 2 \end{pmatrix} = \begin{pmatrix} -1 \\ -8 \\ -2 \end{pmatrix}$$

The new line is $\mathbf{w}_1{}^T\mathbf{x} = \begin{pmatrix} -1 & -8 & -2 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} = -1 - 8x - 2y = 0.$

Notice that $\mathbf{w}_1{}^T\mathbf{x}$ classifies correctly $\mathbf{x}_1$ and $\mathbf{x}_2$, but misclassifies $\mathbf{x}_3$. Table 5.3 contains all the steps done by the alorithm, up to obtain $y = x - \frac{1}{13}$ that is the optimal line, since it classifies correctly all the points.

				Steps	
Step	i	d_i	$\mathbf{x}_i{}^T$	$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta \cdot d_i \cdot \mathbf{x}_i$	$y = -w_1/w_2 \cdot x - w_3/w_2$
1	2	−1	(1, 8, 2)	$\mathbf{w}_1{}^T = (-1, -8, -2)$	$y = -4 \cdot x - 1/2$
2	3	1	(1, 6, 7)	$\mathbf{w}_2{}^T = (-2, 5, 0)$	$y = 2/5 \cdot x$
3	4	−1	(1, 7, 4)	$\mathbf{w}_3{}^T = (-1, -9, 1)$	$y = 9x + 1$
4	3	1	(1, 6, 7)	$\mathbf{w}_4{}^T = (0, -3, 8)$	$y = 3/8 \cdot x$
5	4	−1	(1, 7, 4)	$\mathbf{w}_5{}^T = (-1, -10, 4)$	$y = 5/2 \cdot x + 1/4$
6	3	1	(1, 6, 7)	$\mathbf{w}_6{}^T = (0, -4, 11)$	$y = 4/11 \cdot x$
7	4	−1	(1, 7, 4)	$\mathbf{w}_7{}^T = (-1, -11, 7)$	$y = 11/7 \cdot x + 1/7$
8	3	1	(1, 6, 7)	$\mathbf{w}_8{}^T = (0, -5, 14)$	$y = 5/14 \cdot x$
9	4	−1	(1, 7, 4)	$\mathbf{w}_9{}^T = (-1, -12, 10)$	$y = 6/5 \cdot x + 1/10$
10	3	1	(1, 6, 7)	$\mathbf{w}_{10}{}^T = (0, -6, 17)$	$y = 6/17 \cdot x$
11	4	−1	(1, 7, 4)	$\mathbf{w}_{11}{}^T = (-1, -13, 13)$	$y = x - 1/13$

Table 5.3: All the steps before convergence.

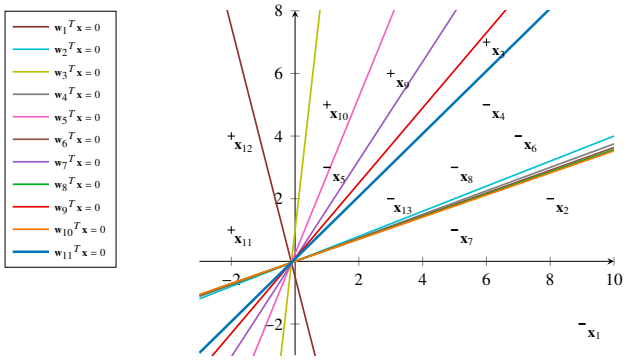


Figure 5.5: Representation of the points and the separating hyperplanes.

Theorem 5.2.1 (Perceptron convergence theorem)**.** *The perceptron is guaranteed to converge (classifying correctly all the input patterns) in a finite number of steps if the problem is linearly separable.*

Notice that in the case of not linearly separable problems, the algorithm may be *unstable*.

Proof. Assuming that the problem is linearly separable then $\exists\, \underline{\mathbf{w}} \in \mathbb{R}^n$ that correctly separates all the training

1: Initialize the weights (either to zero or to a small random value)
2: Pick a learning rate $\eta \in [0, 1]$
3: **repeat**
4: **for each** training pattern $(\mathbf{x}_p, d_p)$, where $d_p = \pm 1$ **do**
5: Compute output activation $out = sign(\mathbf{w}^T \mathbf{x}_p)$
6: **if** $out = d_p$ **then**
7: Continue
8: **else**
9: **if** $\mathbf{w}^T \mathbf{x}_p \leq 0$ and $d_p = 1$ **then** ▷ False negative: d_p is 1 but is predicted −1
10: $\mathbf{w}_{t+1} = \mathbf{w}_t + \eta \cdot d_p \cdot \mathbf{x}_p$
11: **else** $\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \cdot d_p \cdot \mathbf{x}_p$ ▷ False positive: d_p is −1 but is predicted 1
12: **until** All pattern examples are correctly classified

Algorithm 5.1: The Perceptron learning algorithm.

examples. Formally,

$$\exists\, \underline{\mathbf{w}} \in \mathbb{R}^n \text{ such that } \forall p = 1, \ldots, l \; d_p \cdot (\underline{\mathbf{w}}^T \mathbf{x}_p) \geq \alpha, \qquad \text{where } \alpha = \min_p (d_p \cdot (\underline{\mathbf{w}}^T \mathbf{x}_p)) > 0$$

and α accounts for the minimum distance between a training point and the boundary.

Let us focus on the positive patterns, so if the model produces a positive solution it is correct, otherwise it is not. Formally, we will consider only those *input-output* couples $(\mathbf{x}_p, d_p)$, where $d_p = +1$. We leave to the reader to prove the theorem when $d_p = -1$.

The proof is structured in two main parts, finding a *lower-bound* and an *upper-bound* to the norm of the vector $\mathbf{w}_q$[†]. As a final step, we will find an upper-bound to the number of steps q needed to reach convergence.

- **Lower-bound** Let us assume that $\mathbf{w}_0 = \mathbf{0}$ (step 1) and $\eta = 1$, where $\|\cdot\|$ corresponds to the Euclidean norm ($\|\cdot\|_2$). After q iterations we are in the situation in which exactly q $\mathbf{x}_i$ are misclassified by one of the q hyperplanes. In our "positive output only" setting this means that we encountered q false negatives, so we have

$$\mathbf{w}_q = \sum_{j \in X_q^-} \mathbf{x}_j \tag{5.1}$$

Where we indicate with X_q^- the *multiset* of the q points $\mathbf{w}_i$, which where misclassified by one of the q candidate hyperplanes (hence $\left| X_q^- \right| = q$).

The formula above holds since the update rule (line 10 of Algorithm 5.1) $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}_j$ is applied any time that ${\mathbf{w}_t}^T \mathbf{x}_j$ does not correctly classify at least one positive example.

We are now interested in obtaining a lower-bound for $\left\|\mathbf{w}_q\right\|$. Assuming that $\underline{\mathbf{w}}^T$ is one of the possible solutions, by multiplying Equation (5.1) by $\underline{\mathbf{w}}^T$ we get:

$$\underline{\mathbf{w}}^T \mathbf{w}_q = \underline{\mathbf{w}}^T \sum_{j \in X_q^-} \mathbf{x}_j \geq q\alpha \tag{5.2}$$

where $\alpha = \min_p(\underline{\mathbf{w}}^T \mathbf{x}_p)$. From the Cauchy-Schwartz inequality $\left((\mathbf{z}^T \mathbf{v})\right)^2 \leq \|\mathbf{z}\|^2 \cdot \|\mathbf{v}\|^2)$ we have

$$\left\|\underline{\mathbf{w}}\right\|^2 \left\|\mathbf{w}_q\right\|^2 \geq \left(\underline{\mathbf{w}}^T \mathbf{w}_q\right)^2 \geq (q\alpha)^2$$
$$\left\|\mathbf{w}_q\right\|^2 \geq (q\alpha)^2 / \left\|\underline{\mathbf{w}}\right\|^2$$

- **Upper-bound** Recall the following relation for two arbitrary vectors $\mathbf{a}, \mathbf{b}$

$$\|\mathbf{a} + \mathbf{b}\|^2 = \sum_i (a_i + b_i)^2 = \sum_i {a_i}^2 + \sum_i 2a_i b_i + \sum_i {b_i}^2 = \|\mathbf{a}\|^2 + 2\mathbf{ab} + \|\mathbf{b}\|^2$$

Plugging $\mathbf{w}_q = \mathbf{w}_{q-1} + \mathbf{x}_j$ (where $\mathbf{x}_j$ is the first misclassified vector at step $q-1$) into the above relation, we have

$$\left\|\mathbf{w}_q\right\|^2 = \left\|\mathbf{w}_{q-1} + \mathbf{x}_j\right\|^2 = \left\|\mathbf{w}_{q-1}\right\|^2 + 2\mathbf{w}_{q-1}\mathbf{x}_j + \|\mathbf{x}_i\|^2$$

[†]Notice that in this proof we will use subscripts to the weight vector $\mathbf{w}$ to indicate the iteration of the algorithm (e.g. $\mathbf{w}_q$ is the weight vector after q steps of the algorithm).

where the term $2\mathbf{w}_{q-1}\mathbf{x}_j$ is negative because at the $(q-1)$-th iteration x_j was miscalssified, so

$$\left\|\mathbf{w}_q\right\|^2 \leq \left\|\mathbf{w}_{q-1}\right\|^2 + \left\|\mathbf{x}_j\right\|^2$$

Recursively substituting $\left\|\mathbf{w}_{q-1}\right\|^2 \leq \left\|\mathbf{w}_{q-2}\right\|^2 + \|\mathbf{x}_{i'}\|^2$ until iteration 1 and for $\beta = \max_{j \in X_q^-} \left\|\mathbf{x}_j\right\|^2$ we obtain

$$\left\|\mathbf{w}_q\right\|^2 \leq \sum_{j \in X_q^-} \left\|\mathbf{x}_j\right\|^2 \leq q\beta$$

Finally, we can write our function between the lower and upper bound

$$q\beta \geq \left\|\mathbf{w}_q\right\|^2 \geq q^2 \underbrace{\alpha^2 / \left\|\underline{\mathbf{w}}^T\right\|^2}_{\alpha'}$$

$$q\beta \geq q^2\alpha'$$

$$q \leq \beta/\alpha'$$

■

5.3 Least Mean Squares vs. Perceptron Learning Algorithm

Notice that the update rule in the algorithm is stated in the form of *Hebbian learning*, while it can be formulated as *error-correction learning* (learning equation), if written as

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta(d - out) \cdot \mathbf{x} \tag{5.3}$$

The differences between the Perceptron Learning Algorithm (PL Algorithm) and the Least Mean Squares Algorithm (LMS Algorithm) are listed below.

- LMS rule derived without threshold activation functions, minimize the error of the linear unit (using directly $\mathbf{w}^\mathbf{T}\mathbf{x}$)
- For the learning we have $\delta_{PLA} = (d - \mathbf{w}^\mathbf{T}\mathbf{x})$ and $\delta_{LMS} = d - sign(\mathbf{w}^\mathbf{T}\mathbf{x})$
- The model trained with LMS can still be used for classification, by applying the threshold function $h(x) = sign(\mathbf{w}^\mathbf{T}\mathbf{x})$ (LTU).

Observation 5.3.1. *Perceptron algorithm moves* only *if there are some misclassifications on the training set, while the LMS moves according to the weights of the points. As an example, see Figure 5.6, where the point in the right-top corner moves the LMS model away from the best classifier.*

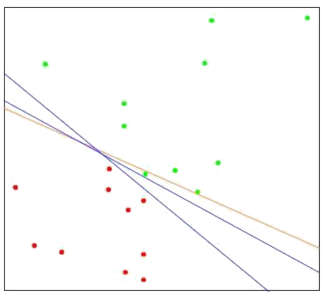


Figure 5.6: The orange line represents the function obtained using the least squares model, while the blue one is obtained through the perceptron learning algorithm

Notice that the perceptron learning algorithm *always* converges to a perfect classifier, if training examples are linearly separable, otherwise it does not.

Conversely, LMS gradient-descent converges asymptotically, regardless of whether the training data are linearly separable.

5.4 Activation functions in Neural Networks

Some of the most significant activation functions f_σ that can be applied on $net = \sum_{i=0}^{n} w_i x_i$ to obtain the output o of the neural network are listed below:

1. **Linear function**, in which case any function f_σ is applied, so $o = net(\mathbf{x})$.
2. **Threshold function** (called also **perceptron** or LTU), where $f_\sigma = sign$, so we have $o = f_\sigma(net) = sign(net)$.
3. A **non-linear squashing function** like the **sigmoidal logistic function**, that assumes a continuous range of values in the bounded interval $[0, 1]$

   $$f_\sigma(x) = \frac{1}{1 + e^{(-ax)}} \tag{5.4}$$

   where a is the *slope* parameter of the function. This function has the property to be the smooth and differentiable version of the threshold function.

   Then, in this case we have $o = f_\sigma(net)$ with $a = 1$.

   The function expressed in Equation (5.4) is not symmetric with respect to the origin point $(0, 0)$, but there exists a symmetric version:

   $$f_{\sigma_{\text{tanh}}} = 2f_\sigma(x) - 1 = \tanh(ax/2) \tag{5.5}$$

   Examples of these two versions of the third type of activation function are shown in figures 5.7(a) and 5.7(b). Notice that, in both cases, if $a \to 0$, then we obtain a linear function, while if $a \to \infty$, then we obtain a function like a LTU.
4. **Radial basis** functions, $f_\sigma = e^{-ax^2}$, which plot is shown in Figure 5.7(c). This type is used in the RBF networks.
5. **Stochastic neurons**, in this case we have

   $$f_\sigma = \begin{cases} +1 & \text{with probability } \mathbb{P}(net) \\ -1 & \text{with probability } 1 - \mathbb{P}(net) \end{cases}$$

   this type is used in the Boltzmann machines (see Section Appendix 6.B) and other models rooted in statistical machines.
6. **Rectifier** or **Rectified Linear Unit** (ReLU), which function type is

   $$f_\sigma = \max(0, x) \quad \text{or (equal)} \quad f_\sigma = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$$

   This type becomes a default choice for Deep models, a plot is shown in Figure 5.7(d).
7. **Softplus**, in which case we have $f_\sigma = \ln(1 + e^x)$, plotted as a light blue line in Figure 5.7(d).

5.4.1 Derivatives of activation functions

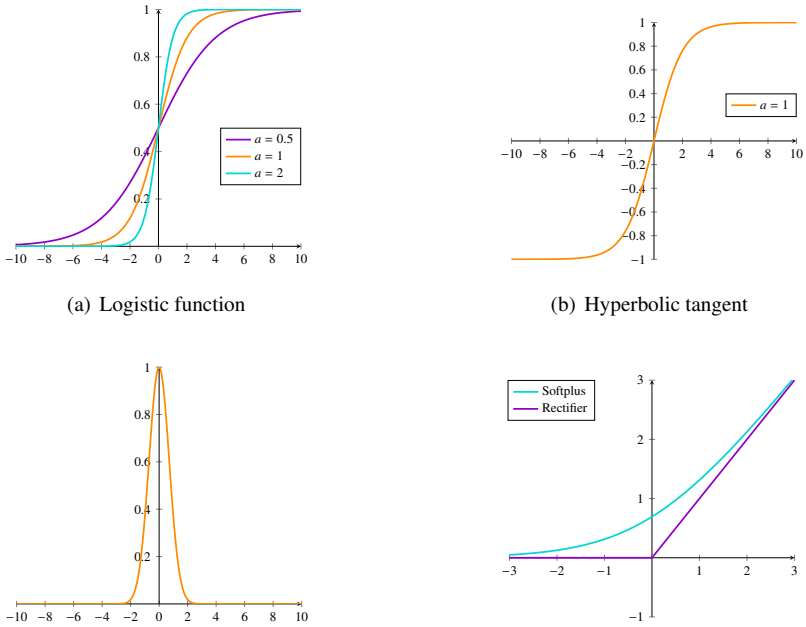
Its important for us to evaluate the derivatives of the activation functions, the reason of this will be clear later on. Such derivatives follow:

- for the identity function it is 1,
- for the step function it is not defined which is exactly why it isn't used,
- for the sigmoid, for asymmetric and symmetric case (for $a = 1$) we have:

  $$\frac{\partial f_{\sigma_{\text{logistic}}}(x)}{\partial x} = f_{\sigma_{\text{logistic}}}(1 - f_{\sigma_{\text{logistic}}}) = \frac{e^{-x}}{(1 + e^{-x})} \qquad \frac{\partial f_{\sigma_{\text{tanh}}}(x)}{\partial x} = 1 - f_{\sigma_{\text{tanh}}}(x)^2 = 1 - (\tanh(x/2))^2$$

- ReLU is not differentiable in $(0, 0)$, but softplus is, so we get in the case of $a = 1$

  $$\frac{\partial f_{\sigma_{\text{softplus}}}}{\partial x} = \frac{e^x}{1 + e^x} = \frac{1}{1 + e^{-x}}$$



(a) Logistic function

(b) Hyperbolic tangent

(c) Radial basis

(d) ReLU and Softplus

Figure 5.7: Plots of the activation functions. Figures (a) and (b) shown respectively the asymmetric and symmetric version of the sigmoidal function. Figure (c) shows the radial basis function, while (d) shown respectively the softplus and rectifier functions.

5.4.2 LMS and activation functions

As already stated, a LMS algorithm is obtained by computing the gradient of the loss function with respect to the weights, where the update rule is the one in Equation (5.3).

Provided that, for an activation function f_σ, the loss function is computed as follows

$$E(\mathbf{w}) = \sum_{p=1}^{l} \left(d_p - o(\mathbf{x}_p)\right)^2 = \sum_{p=1}^{l} \left(d_p - f_\sigma(\mathbf{w}^T \mathbf{x}_p)\right)^2$$

we are interested in finding a new delta to plug to tweak the weights as follows:

$$\begin{aligned}
\frac{\partial E(\mathbf{w})}{\partial w_j} &= \frac{\partial \left(\sum_{p=1}^{l} \left(d_p - f_\sigma(\mathbf{w}_p^T \mathbf{x})\right)^2\right)}{\partial w_j} = \\
&= \sum_{p=1}^{l} \frac{\partial \left((d_p - f_\sigma({\mathbf{w}_p}^T \mathbf{x})\right)^2)}{\partial w_j} = \\
&= \sum_{p=1}^{l} 2 \cdot \underbrace{\left(d_p - f_\sigma({\mathbf{x}_p}^T \mathbf{x})\right)}_{\delta_p} \cdot \frac{\partial \left(d_p - f_\sigma(\mathbf{w}^T \mathbf{x}_p)\right)}{\partial w_j} = \\
&= -2 \cdot \sum_{p=1}^{l} \delta_p \cdot \frac{\partial (\mathbf{w}^T \mathbf{x}_p)}{\partial w_j} \cdot f'_\sigma(\mathbf{w}^T \mathbf{x}_p) = \\
&= -2 \cdot \sum_{p=1}^{l} \delta_p \cdot {\mathbf{x}_p}^T \cdot f'_\sigma(\mathbf{w}^T \mathbf{x}_p) = \\
&= -2 \cdot \sum_{p=1}^{l} \mathbf{x}_p \cdot \delta_p \cdot f'_\sigma(\mathbf{w}^T \mathbf{x}_p)
\end{aligned}$$

Hence, for each component of $\mathbf{w}$ we get $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = -2 \cdot \sum_{p=1}^{l} \mathbf{x}_p \delta_p f'_\sigma(\text{net}_p)$. Plugging this tweak into the Equation (5.3) we get the new delta rule, where $\delta'_p := \delta_p \cdot f'_\sigma(\text{net}_p)$

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \eta \delta'_p \mathbf{x}_p = \mathbf{w}_{\text{old}} + \eta (d_p - f_\sigma(net_p)) f'_\sigma(net_p) \mathbf{x}_p$$

Considering the sigmoidal logistic function as the activation function, the slope parameter a can affect the step of the gradient descent also in case of its derivatives. Maximum of $f_\sigma{}'$ is used for those inputs which are close to 0, hence linear unit, while the minimum of $f_\sigma{}'$ is used for saturated cases. For the last case it is better to start with small weights to avoid saturation [1], and hence very slow changes of w, at the beginning.

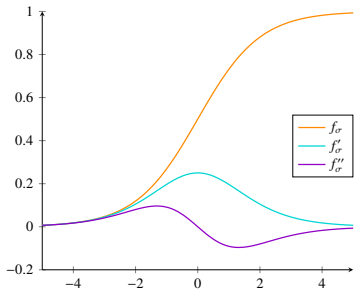


Figure 5.8: Plot of the sigmoidal activation function and its first and second derivative. Notice that whenever the derivative of the activation function goes to 0 we get that $\delta_p = (d_p - f_\sigma(net_p)) \cdot f_\sigma{}'(net_p) = 0$ although the error may be large.

5.5 More on neural networks

Let us consider a 2-layers architecture, as shown in Figure 5.9 as a *flexible function*

$$h(\mathbf{x}) = f_k\Big(\sum_{j=1}^{J} w_{kj} f_j(\mathbf{w}_j{}^T \mathbf{x})\Big) = f_k\Big(\sum_{j=1}^{J} w_{kj} f_j\Big(\sum_{i=1}^{n} w_{ji} x_i\Big)\Big) \tag{5.6}$$

where $\mathbf{w}$ is the matrix of the free parameters, f_k, f_j are non-linear activation functions of the output layer and hidden layer respectively and $\mathbf{x}$ is the vector of input variables.

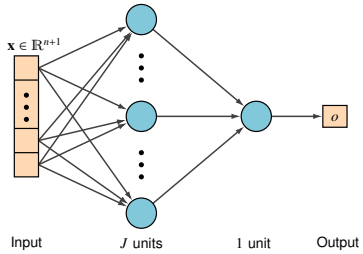


Figure 5.9: Graphical representation of neural network.

A neural network model is characterized by the type of:

- **unit**, as the network and the activation functions;
- **architecture**, as the number of units or the topology (e.g. number of layers);
- **learning algorithm**.

The architecture of a neural network defines the topology of the connections among the units. The two-layer **feed-forward** (the direction of all arrows is from input to output) neural networks described in Equation (5.6) corresponds to the well-know MLP (Multi Layer Perceptron) architecture:

- the units are connected by weighted links and they are organized in the form of layers;
- the input layer is simply takes the input $\mathbf{x}$ and projects it onto the hidden layer of units;
- the hidden layer projects onto the output layer.

Recurrent neural networks (Section 5.5) have a different architecture, based on the addition of *feedback loops* connections in the network topology:

- the presence of *self-loop* connections provides the network with dynamical properties, letting a memory of the past computations in the model;
- this allows us to extend the representation capability of the model to the processing of the sequences (and structured data).

Now we will describe the neural networks answering to some technical questions.

Why is a neural network a flexible model? Why does it work well?

- **Hypothesis space**: continuous space of all the functions that can be represented by assigning the weight values of the given architecture.
- Depending on the class of values produced by the network output units, discrete or continuous, the model can deal, respectively, with **classification** (sigmoidal output function) or **regression** (linear output function) task. The output function corresponds to f_k in Equation (5.6).
- Multi-regression or multi-classes classifier can be obtained by using **multiple output units**.
- The Equation (5.6) corresponds to the function computed by a two-layer feedforward neural network. Units and architecture are just a graphical representation. Each $f(\sum wx)$ can be seen as computed by an independent processing element (unit), thus a neural network is a **non-linear function** in the parameters w.

As for the linear basis expansions (discussed in Section 3.4), which is is linear with respect to the parameters w and Φ, we can see the neural network expressed in Equation (5.6) as the Equation (3.2) with Φ equal to the "internal unit" $f(\sum wx)$, more clearly

$$h(\mathbf{x}) = \sum_j w_j \Phi_j(\mathbf{x}) \qquad \text{where } \Phi_j(\mathbf{x}, \mathbf{w}) = f_j\Big(\sum_j w_{ji} x_i\Big) \tag{5.7}$$

where f_j represents the activation function of the j-th layer.

In this case the basis functions themselves are adapted to data (by fitting of w in Φ). Note also that we fix the same type of basis functions for all the terms in the basis expansions, given by the activation function.

Hence $h(\mathbf{x})$ (non-linear function in w) results in the summation of non-linearly transformed linear models, to enhance adaptivity.

Each basis function (hidden unit) computes a new non-linear derived feature, adaptively by learning according to the training data (e.g. the parameters of the basis function w are learned from data by learning Φ in Equation (5.7)).

In other words, the representational capacity of the model is related to the presence of a hidden layer of units, with the use of non-linear activation function, that transforms the input pattern into the internal representation of the network. The learning process can define a suitable internal representation, also visible as new hidden features of data, allowing the model to extract from data the higher-order statistics that are relevant to approximate the target function.

Non-linear units are essential because an MLP with linear units is equal to a one-layer neural network.

Advantages of neural networks

- Learn and model non linear and complex relationships
- Allow good generalization performances

- Do not impose constraints to the distribution of data
- Can make parallel computations

Is the flexibility theoretically grounded?

A single hidden-layer network, with logistic activation functions, can approximate (arbitrarily well) every continuous (on hypercubes) function, if provided with enough units in the hidden layer. A MLP network can approximate (arbitrarily well) every input-output mapping, again, if provided with enough units in the hidden layer. The fact that MLP is able to represent any function, is formalized in the following

Theorem 5.5.1 (Universal approximation)**.** *Given* ε

$$\exists h(x) \text{ such that } |f(x) - h(x)| < \varepsilon \quad \forall x \text{ in the hypercube}$$

Observation 5.5.1. *The theorem proves* only *the existence of a MLP network, without providing the algorithm to build it or the number of units it is made of.*

At this point we are left with: (1) how to learn by neural network and (2) how to decide a neural network architecture.

The **expressive power** of neural network is strongly influenced by two aspects: the number of units and their configuration (architecture). The number of units can be related to the discussion of the **VC-dimension** of the model. Specifically, the network capabilities are influenced by the number of parameters, that is proportional to the number of units, and further studies report also the dependencies on their value sizes.

The universal approximation theorem is fundamental to answer the question "how many layers should the network have?". One-hidden-layer is sufficient in general, but we have no guarantess that a "small number" of units could be sufficient (it does not provide a limit on such number). It is possible to find boundaries on such number, but also find "no flattening" results. The last scenario includes cases for the ones the implementation by a single hidden layer would require an exponential number of units, of non-zero weight, while more layers can help (w.r.t. the number of units/weights and/or for learning such approximation).

How to learn the weights of a neural network?

The **learning algorithm** allows adapting the free-parameters of the model, i.e. the values of the connection-weight, in order to obtain the best approximation of the target function. In the ML framework this is often realized in terms of minimization of an error (or loss) function on the training data set.

The problem is the same we have seen for the other models: given a set of l training example (x_p, d_p) and a loss function L, we aim to find the weight vector $\mathbf{w}$ that minimizes the expected loss on the training data

$$\widehat{\mathbf{w}} = \arg\min_{\mathbf{w}}(E(\mathbf{w}) = R_{emp}) = \arg\min_{\mathbf{w}}\left(\frac{1}{l}\sum_{p=1}^{l} L(h(\mathbf{x}_p), d_p)\right)$$

5.6 Backpropagation

A network of perceptrons can represent every Boolean function, but the problem is defininig a learning algorithm for the network (MLP), since we have a *credit assignment problem* (aka we do not know how to find the node responsible for some errors).

At this point, we would like to apply a gradient descent approach to this problem.

Definition 5.6.1 (Loading problem)**.** *Given a network and a set of examples (where the answer is yes/no) the task of checking if there is a set of weights so that the network will be consistent with the examples is called* ***loading problem*** *and it is a NP-complete problem.*

We are interested in finding a $\mathbf{w}$ by computing the gradient of the loss function

$$E(\mathbf{w}) = \sum_{p=1}^{l} (d_p - h(\mathbf{x}_p))^2$$

Do you recall?

The reader should keep in mind that we need the loss and the activation functions to be differentiable. Moreover, we recall that an *epoch* is an entire cycle of training pattern presentation.

Back-propagation based learning algorithm

The problem that we want solve is to estimate the contribution of hidden units on the error at the output level and this implies that we need a generalized version of the delta rule. Since we are in the case of *supervised learning*, we are in presence of a *training set*: $\{(\mathbf{x}_1, \mathbf{d}_1), \dots (\mathbf{x}_l, \mathbf{d}_l)\}$, where $\mathbf{d}_i \in \mathbb{R}^q$.

Our aim is to find the values of the weights $\mathbf{w}$, such that we minimize the total error

$$E_{\text{tot}} = \sum_{p=1}^{l} E_p \qquad \text{where} \quad E_p = \frac{1}{q} \sum_{i=1}^{q} ((\mathbf{d}_p)_i - o_i)^2 \quad \text{(aka Least Mean Square)}$$

1: Initialize the weights (either to zero or to a small random value)
2: Initialize η
3: Compute *out* and E_{tot}
4: **repeat**
5: **for each** $w \in \mathbf{w}$ **do**
6: $\Delta w = -\frac{\partial E_{\text{tot}}}{\partial w}$ Gradient computation
7: $w_{\text{new}} = w_{\text{old}} + \eta \Delta w + \dots$ which depends on the updating rule
8: Compute *out* and E_{tot}
9: **until** The total error is below a certain desired threshold or other criteria

Algorithm 5.2: The ITERATIVE GRADIENT DESCENT TRAINING algorithm

Let us concentrate on the gradient computation:

$$\Delta w = -\frac{\partial E_{\text{tot}}}{\partial w} = -\sum_{p=1}^{l} \frac{\partial E_p}{\partial w} = \sum_{p=1}^{l} \Delta_p w$$

where p is the index of the p-th pattern fed as input the neural network and w is a component of a generic weight vector $\mathbf{w}$.

For a generic layer t, let us consider a generic unit net_{t_τ}, that will be from now on referred to as net_t to ease notation, as showed in Figure 5.10.

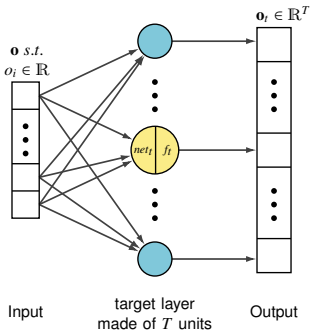


Figure 5.10: Generic layer of a feedforward fully connected neural network.

We denote $\mathbf{o}$ an input of such unit and f_t the activation function applied to net_t, so we get:

$$net_t = \sum_{i=1}^{n} w_{ti} o_i \qquad \text{and } o_t = f_t(net_t)$$

We are interested in computing, for each component i of the vector $\mathbf{w}$, $\Delta_p w_{ti}$:

$$\Delta_p w_{ti} = -\frac{\partial E_p}{\partial w_{ti}} = \underbrace{-\frac{\partial E_p}{\partial net_t}}_{\delta_t} \cdot \frac{\partial net_t}{\partial w_{ti}} \tag{5.8}$$

We denote the first factor as δ_t and evaluate the second factor:

$$\frac{\partial net_t}{\partial w_{ti}} = \frac{\partial \sum_h w_{th} o_h}{\partial w_{ti}} \overset{*}{=} o_i$$

where $\overset{*}{=}$ is justified observing that the quantity $w_{th} o_h$ is constant with respect to w_{ti}. Thus, the Equation (5.8) can be written as

$$\Delta_p w_{ti} = \delta_t \cdot o_i$$

To compute δ_t we apply the chain rule to write this partial derivative as the product of two factors: (1) reflects the change in error as a function of the output of the unit and (2) reflecting the change in the output as a function of changes in the input. Thus, we have

$$\delta_t = -\frac{\partial E_p}{\partial net_t} = -\frac{\partial E_p}{\partial o_t} \cdot \frac{\partial o_t}{\partial net_t} \tag{5.9}$$

Computing the second factor of Equation (5.9), we obtain

$$\frac{\partial o_t}{\partial net_t} = \frac{\partial f_t(net_t)}{\partial net_t} = f_t'(net_t)$$

In order to compute the first factor of Equation (5.9) $\left(-\frac{\partial E_p}{\partial o_t}\right)$, we consider two cases (see Figure 5.11):

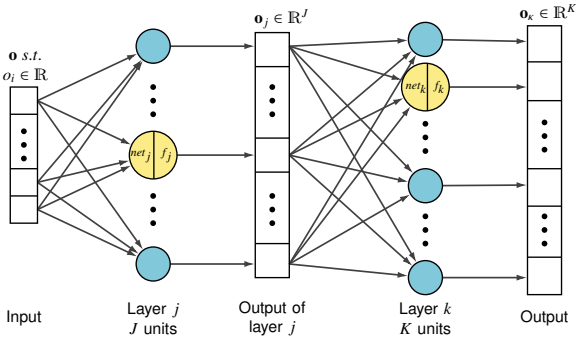


Figure 5.11: Generic architecture of a feedforward fully connected neural network.

Output layer: let us assume $t = k$, hence we are considering an output unit of the network. In this case, it follows from the definition of E_p that for an internal node r

$$-\frac{\partial E_p}{\partial o_k} = -\frac{\partial\left(\frac{1}{K}\sum_{\kappa=1}^{K}(d_\kappa - o_\kappa)^2\right)}{\partial o_k} = \frac{2}{K}(d_k - o_k)$$

Substituting the two factors in Equation (5.9), we get

$$\delta_k = \frac{2}{K}(d_k - o_k) \cdot f_k'(net_k)$$

Hidden layer: let us assume that unit $t = j$, hence we are considering a hidden unit of the network. In this case we can use the chain rule to write

$$-\frac{\partial E_p}{\partial o_j} \overset{*}{=} \sum_{\kappa=1}^{K} -\frac{\partial E_p}{\partial net_\kappa} \cdot \frac{\partial net_\kappa}{\partial o_j} = \sum_{\kappa=1}^{K} \delta_\kappa \cdot \frac{\partial \sum_i w_{ki} o_i}{\partial o_j} = \sum_{\kappa=1}^{K} \delta_\kappa \cdot w_{\kappa j}$$

Substituting for the two factors in Equation (5.9), we get

$$\delta_j = \left(\sum_{\kappa=1}^{K} \delta_\kappa \cdot w_{\kappa j} \right) \cdot f_j'(net_j)$$

Non è che mi torni tantissimo la $\overset{*}{=}$...

The algorithm for the gradient descent is the same as before, but in this case we compute $\Delta\mathbf{w}$ through back-propagation algorithm for any weight in the network.

5.7 Issues in training a neural network

Training a neural network is a process which requires a lot of adjustments due to the nature of the problem. In this section we are going to address some issues in training such a model.

5.7.1 Choosing the initial weights

The following values for $\mathbf{w}$ should be avoided (since they can hamper the training):

- all zeros
- high values
- all equal values (symmetry)

Conversely, some good guesses for $\mathbf{w}$ are:

- some values in a certain range (e.g $[-0.7, 0.7]$);
- some values in a range of size $2 \cdot$ fan-in, where the fan-in is the number of inputs of a hidden layer. Notice that this heuristic should not be used if the fan-in is too large or in the case of output units;
- other heuristics (e.g. Glorot, Bengio, ...).

5.7.2 Multiple minima

It goes without saying that if the loss function is not convex it has a lot of local minima and, depending on the starting point, we may incur on one minimum or another. In order to be reasonably sure that we picked the best minimum, we need to run multiple trials (starting from different points), in order to:

- pick the solution that achieves best results in terms of error or the mean of such values;
- take advantage of different end points by an average response: mean of the outputs/voting.

An attentive reader may notice that talking about the global minimum of a function which is only an approximation of the empirical risk may not be particularly important.
Moreover, we would like to achieve a good generalization performance and to do so we should not reduce the empirical risk if this means overfitting the data (recall that training a neural network increases the VC-dim). Notice that in the case of neural networks we talk about **overtraining**, which is the process which leads to a too high VC-dim, due to a training process which decreases too much the empirical risk.

Mantra

The real problem of neural networks is not to find a local (or global) minimum, but to *stop the beast* before reaching it.

5.7.3 Online vs batch

We need to choose between the two. In particular, in the *online* version we recompute $\Delta_p \mathbf{w}$ and $\mathbf{w}_{t+1}$ *for each pattern* p. It goes without saying that the online version can be faster, but needs a smaller η (learning rate).

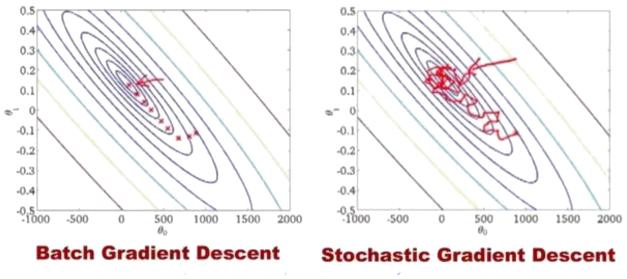


Figure 5.12: A graphical hint on how the convergence changes in presence of batch and online

A trade-off between batch and online versions for computing the gradient descent is to design **mini-batch** (also called Stochastic Gradient Descent). The size of mini-batches (denoted as **mb**) belongs to the interval $[1, l]$, where the leftmost extreme corresponds to the stochastic/online version, while the rightmost extreme is the batch version.
An attentive reader may notice that in both the online and the minibatch case we can end up in a bias due to the order of training examples.
To overcome this issue, we choose to perform some *shuffling* of the training examples. Another good hint about the learning rate is to compute the mean (average) of the gradients over the epochs.

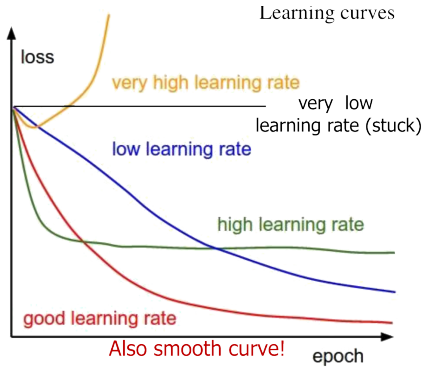


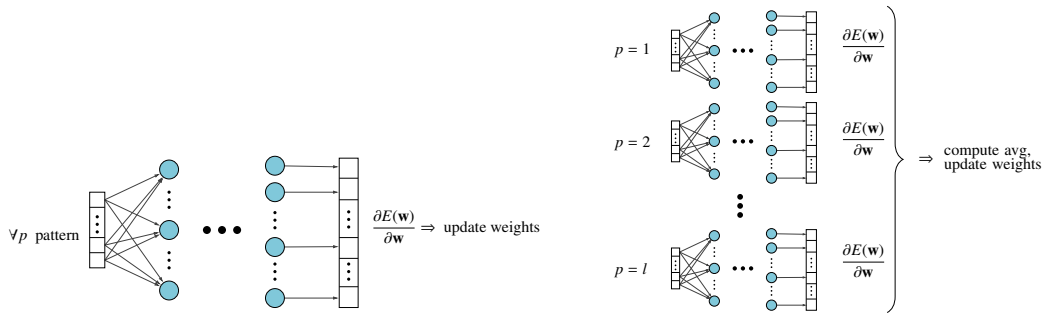
Figure 5.13: In this picture the plots with different values for learning rates are displayed: it goes without saying that a very high learning rate is not good (yellow) and a too low learning rate is bad either (blue), but it is not trivial to find the best learning rate (red) without getting trapped in a high learning rate which has good performaces at first but then gets stuck in a local minimum (green).

Notice that when using the average of gradient for the epochs (batch), on-line training will require *much smaller* η to be comparable, since it is multiplied by l (or mb in the case of mini batches). To convince yourself about this statement have a look at Figure 5.14.

5.7.4 Learning rate

Momentum

Intuitively, when moving towards the minimum, we would like to use a quite big step size, without incurring in the problem of diverging from the solution. It is in this scenario that we define the **momentum**, which is a strategy to "selectively increase the speed": when a gradient from the previous step "points" in the same direction as our current time step, we increase "the speed" since we are moving in the decreasing direction.



(a) Online (b) Batch

Figure 5.14: Comparison between weights updates in online and batch mode.

Definition 5.7.1 (Momentum)**.** *We call* ***(Polyak) momentum*** *the strategy used to add a sort of memory to the step direction, in order not to follow a wide zig zag path when we are in presence of a canyon. Formally, we term* ***momentum*** *the quantity* $\alpha\Delta\mathbf{w}_t$ *(where* $\alpha \in \mathbb{R}$ *is called* momentum parameter *and it is constant across iterations) that should be added to the new delta, resulting in*

$$\Delta w_{t_{new}} = -\eta \cdot \frac{\partial E(\mathbf{w}_t)}{\partial w_t} + \alpha\Delta w_t \in \mathbb{R} \tag{5.10}$$

that is equivalent to the following (multidimensional version)

$$\Delta \mathbf{w}_{t_{new}} = -\eta \cdot \frac{\partial E(\mathbf{w}_t)}{\partial \mathbf{w}_t} + \alpha\Delta \mathbf{w}_t \in \mathbb{R}^{len(\mathbf{w}_t)} \tag{5.11}$$

where $\mathbf{w}_t$ *refers to the weight vector at the t-th iteration.*

Mantra

An attentive reader may have noticed that using the momentum on online version of backpropagation has a completely different meaning, since it smooths the gradient over different training examples.

Definition 5.7.2 (Nesterov momentum)**.** *We call* ***Nesterov momentum*** *the momentum that computes the gradient at the point already "shifted", formally*

$$\begin{cases} \mathbf{w}_{t+1} = \mathbf{v}_t - \eta \cdot \frac{\partial E(\mathbf{v}_t)}{\partial \mathbf{w}_t} \\ \mathbf{v}_{t+1} = \mathbf{w}_{t+1} + \beta\Delta\mathbf{w}_{t+1} \end{cases}$$

It has been proved formally that the Nesterov momentum improves the convergence rate for the *batch mode*.

Adaptive Learning Rates

It is possible to use separate η for each parameter to reduce the fine tuning phase via hyperparameters selection.

Convergence speed

It is possibile to use other heuristic approaches, such as R-prop, which makes use of the sign of the gradient instead of its value.

5.7.5 Stopping criteria

The basic stopping criterion is to use loss, but the problem is that we do not always have the tolerance of data. We will stop in the case in which the loss is lower than a threshold.
As an alternative we can use:

- max (instead of mean) tolerance of the data
- classification, we will use the number of missclassified examples (a.k.a error rate);
- no more relevant weight changes, near zero gradient, e.g. the euclidean norm of the gradient il less than a threshold, or there is no more significant error decreasing for a epoch. Note that in this case it may be premature.

In any case we will have to stop after an excessive number of epochs. Is not necessary to stop in presence of very low training error.

5.7.6 Overfitting and regularization

Typically we do not want to reach the global minimizer of $R_{\text{emp}}(w)$, as this is likely to be an overfitting solution. This is an example of diversity of aim in machine learning with respect to optimization methods.

The control of complexity it is our main concern to achieve the best generalization capability. For instance, we need to add some *regularization*: this is achieved directly through a penalty term, or indirectly by early stopping, or again a model selection (e.g. cross-validation) on empirical data in order to find the best trade-off.

A first approach is to start learning with small weights.

In the case where the mapping of input to output is nearly linear, the number of effective free parameters (and VC-dim) are nearly as in perceptron. As optimization proceeds hidden units tend to saturate, increasing the effective number of free parameters and hence increase VC-dim.

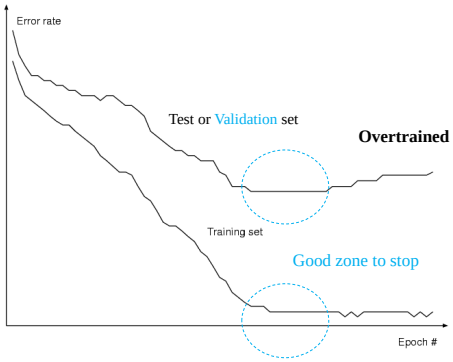


Figure 5.15: Plot of the learning curve with the best zone to stop for training and validation/test set.

From Figure 5.15 we can see that the region where it is best to stop can be different from validation and training set. In order to avoid the problem of the early stop it is common practice to use a validation set for determining when to stop moving. Moreover, it is better to use more than one epoch before estimating the correct stop or backtracking approaches. Note that since the effective number of parameters grows during the course of training, halting training correspond to limit the effective complexity.

We are interested in penalizing complexity and this can be achieved adding all weights to the loss function (that won't quite work because some parameters are positive and some are negative, so we'll use the norm). Notice that without the wise use of an hyperparameter, it might result in our loss getting so huge that the best model would be to set all the parameters to 0.

Definition 5.7.3 (Weight decay)**.** *We term* ***weight decay*** *the regularization technique related to Tikhonov theory, that consists in computing the loss as the summation of a penalty term and the error function*

$$Loss(\mathbf{w}) = E(\mathbf{w}) + \underbrace{\lambda||\mathbf{w}||_i^2}_{\text{penalty term}} = \sum_{p=1}^{l} (d_p - o(\mathbf{x}_p))^2 + \lambda||\mathbf{w}||_i^2$$

where in the case of $i = 1$ *we talk about* Lasso regression *(or* L_1*), while when* $i = 2$ *we are in presence of* Ridge regression *(or* L_2*).*

Lemma 5.7.1. *In the case of* weight decay*, the delta rule becomes*

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \Delta\mathbf{w}_{t+1} - 2\eta\lambda\mathbf{w}_{old}$$

Proof. We need to compute the new value for $\frac{\partial \text{Loss}(\mathbf{w})}{\partial w}$

$$\frac{\partial \text{Loss}(\mathbf{w})}{\partial w} = \frac{\partial E(\mathbf{w})}{\partial w} + \frac{\partial(\lambda\|\mathbf{w}\|_i^2)}{\partial w} = \frac{\partial E(\mathbf{w})}{\partial w} + \frac{\partial\left(\lambda \cdot \sum_{i=1}^{\text{len}(\mathbf{w})} {w_i}^2\right)}{\partial w} = \frac{\partial E(\mathbf{w})}{\partial w} + 2\lambda w$$

Provided that $\Delta\mathbf{w}_{t+1} = -\eta \cdot \frac{\partial \text{Loss}(\mathbf{w})}{\partial w}$, we get the thesis

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \Delta\mathbf{w}_{\text{new(no reg)}} - 2\eta\lambda\mathbf{w}_t$$

■

Notice that the regularization parameter λ is generally chosen as a very small value (e.g. 0.01) and is selected in the model selection phase.

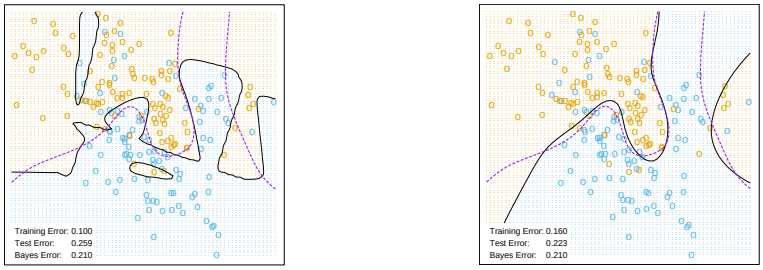
Terminology

We often simplify the terminology calling equivalently Loss, Error or Risk of the objective function, but when we re in presence of regularization techniques it is better to clarify:

Loss for the objective function, and

Error (or Risk) for the "data term" $\sum_p (d_p - o(\mathbf{x}_p))^2$

to evaluate model error, because this is the measure which is useful to the user.



(a) (b)

Figure 5.16: A neural network on the mixture example of Chapter 4. The left panel uses no weight decay, and overfits the training data. The right panel uses weight decay, and achieves close to the Bayes error rate (broken purple boundary). Both use the softmax activation function and cross-entropy error.

Note that often the bias w_0 is omitted from the regularizer (because its inclusion causes the results to be not independent from target shift/scaling) or it may be included but with its own regularization coefficient.

Finally, for on-line/mini-batch take care of possible effects over many steps (patterns/examples). In order to compare those with the batch version in a fair way, do not force the same value for λ, use $\lambda \cdot mb/l$ instead. Of course if you choose λ by model selection they will automatically select different λ for on-line and batch (or any mini-batch).

Lemma 5.7.2 (Weight decay and Momentum)**.** *It is possible to use both regularization techniques and optimizers (momentum), formally*

$$\begin{cases} \Delta\mathbf{w}_{new_i} = \eta\delta_t\mathbf{x}_i + \alpha\Delta\mathbf{w}_{old_i} \\ \mathbf{w} = \mathbf{w} + \Delta\mathbf{w}_{t+1} - \lambda\mathbf{w} \end{cases}$$

Proof. Recall

Momentum: $\Delta \mathbf{w}_{t+1} = -\eta \cdot \frac{\partial \text{Loss}(\mathbf{w})}{\partial w} + \alpha \Delta \mathbf{w}_t$

Weight decay: $\Delta w = -\eta \cdot \frac{\partial Loss(w)}{\partial w} = \Delta w - 2\eta\lambda w \overset{*}{=} \eta\delta_t x - 2\eta\lambda w$ where $\overset{*}{=}$ follows from the backpropagation chain of equations.

Let us delete the scalar η from the second term in the subtraction above, in order to make η and λ independent. We are now ready to plug the formula for the weight decay into the delta rule with momentum and get

$$\Delta \mathbf{w}_{t+1} = \eta\delta\mathbf{x} - 2\lambda\mathbf{w}_t + \alpha\Delta\mathbf{w}_t$$

We do not like the term λw, since this implies that the exponential decay of δ is reflected also on λ and this leads to indirect effects. To solve this issue, we split as follows

$$\begin{cases} \Delta \mathbf{w}_{\text{new}_i} = \eta\delta_t \mathbf{x}_i + \alpha\Delta\mathbf{w}_{\text{old}_i} \\ \mathbf{w} = \mathbf{w} + \Delta\mathbf{w}_{t+1} - \lambda\mathbf{w} \end{cases}$$

■

5.7.7 Number of hidden units

The number of hidden units is related to the complexity of the model, in particular it can be high if appropriate regularization is used. It is also related to the input dimension and to the size of the training set.

There are two kinds of *model selection issues*: if there are too few hidden units we incur in underfitting. Conversely, if there are too many hidden units, we are in presence of overfitting.

Such issues can be avoided exploiting one of the following methods:

- *constructive approaches* (which we will discuss in detail in the section below), in which the learning algorithm determines the number of units the network should be initialized with and then adds new units during training;
- *pruning methods*, that start with large networks and progressively eliminate weights or units.

A Constructive approach: Cascade Correlation learning algorithm

The *Cascade Correlation* (CC) learning algorithm is used for both regression and classification tasks. Since it is one of the *constructive approaches*, it is initialized with a minimum network and units are added when needed, until the error decreases to a certain threshold.

For each new hidden unit, we attempt to maximize the magnitude of the correlation (or more precisely the *covariance*) between the new unit's output and the residual error signal we are trying to eliminate. Notice that trying to maximize the correlation corresponds to design steps which *add* the gradient of the loss function (gradient *ascent*).

The beauty of the CC algorithm is that it can learn both the network weights and network topology, automatically determining the dimension and the topology of the network, training a single unit for each step. The pseudocode of the CC algorithm is reported in Algorithm 5.3.

1: Initialize N as a network without hidden units, and compute its error
2: **repeat**
3: Add to N a hidden unit such that the correlation between the output of the unit and the residual error of the previous network is maximized
4: Train the new network
5: Fix the weights of the new unit (they cannot be retrained in the next step) and train again the remaining weights
6: **until** the residual error on the output layer satisfies a specified stopping criterion

Algorithm 5.3: The Cascade Correlation algorithm.

In Figure 5.17 the evolution of the CC algorithm for a network with up to 3 hidden units. The method dynamically builds up a neural network and terminates once a sufficient number of hidden units to solve the given problem has been found.

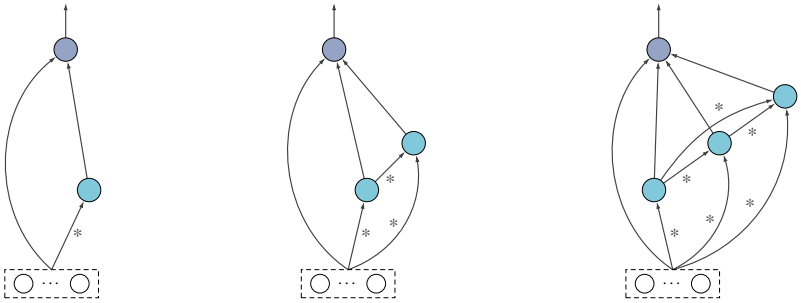


Figure 5.17: The evolution of a CC network with up to 3 hidden units. The symbol "*" represents a frozen weight after candidate training.

Specifically, the algorithm works interleaving the minimization of the total error function (LMS) and the maximization of the (non-normalized) correlation, of the new inserted hidden unit with the residual error S:

$$S = \sum_{k=1}^{K} \left| \sum_{p=1}^{l} (o_p - \text{avg}_p(o)) \cdot (E_{k,p} - \text{avg}_p(E_k)) \right|$$

where o_k is the network output unit at which the error is measured, p is the index of the training pattern under consideration and $E_{k,p} = (o_{k,p} - d_{k,p})$ is the residual error.

In order to maximize S we must compute $\partial S / \partial w_j$, the partial derivative of S with respect to each of the candidate unit's incoming weights w_j.

Fact 5.7.3. *The partial derivative of S with respect to each of the candidate unit's incoming weights is*

$$\frac{\partial S}{\partial w_j} = \sum_{k=1}^{K} \left[sign(S_k) \cdot \sum_{p=1}^{l} \left[\left(E_{k,p} - \text{avg}_p(E_k) \right) \cdot f'(net_p) \cdot I_{j,p} \right] \right]$$

Rivedere notazione. Forse $o_p \to o_k$ e $\text{net}_p \to \text{net}_k$

Proof. It is only a matter of algebra:

$$\frac{\partial S}{\partial w_j} = \frac{\partial \left(\sum_{k=1}^{K} \left| \sum_{p=1}^{l} \left(o_p - \text{avg}_p(o) \right) \cdot \left(E_{k,p} - \text{avg}_p(E_k) \right) \right| \right)}{\partial w_j} =$$

$$\overset{(1)}{=} \sum_{k=1}^{K} \left[sign(S_k) \cdot \sum_{p=1}^{l} \frac{\partial \left(o_p - \text{avg}_p(o) \right) \cdot \left(E_{k,p} - \text{avg}_p(E_k) \right)}{\partial w_j} \right] =$$

$$\overset{(2)}{=} \sum_{k=1}^{K} \left[sign(S_k) \cdot \sum_{p=1}^{l} \left[\left(E_{k,p} - \text{avg}_p(E_k) \right) \cdot \frac{\partial \left(o_p - \text{avg}_p(o) \right)}{\partial net_p} \cdot \frac{\partial net_p}{\partial w_j} \right] \right] =$$

$$\overset{(3)}{=} \sum_{k=1}^{K} \left[sign(S_k) \cdot \sum_{p=1}^{l} \left[\left(E_{k,p} - \text{avg}_p(E_k) \right) \cdot \underbrace{\frac{\partial \left(o_p - \text{avg}_p(o) \right)}{\partial o_p}}_{1} \cdot \underbrace{\frac{\partial o_p}{\partial net_p}}_{f'(net_p)} \cdot \underbrace{\frac{\partial net_p}{\partial w_j}}_{I_{j,p}} \right] \right] =$$

$$\overset{(4)}{=} \sum_{k=1}^{K} \left[sign(S_k) \cdot \sum_{p=1}^{l} \left[\left(E_{k,p} - \text{avg}_p(E_k) \right) \cdot f'(net_p) \cdot I_{j,p} \right] \right]$$

where $\overset{(1)}{=}$ follows from the equality $\frac{\partial |f(x)|}{\partial x} = sign(f(x)) \cdot f'(x)$, $\overset{(2)}{=}$ follows from the application of the chain rule and the fact that $\left(E_{k,p} - \text{avg}_p(E_k) \right)$ is constant with respect to w_j because of how the algorithm works. While $\overset{(3)}{=}$ is explained by the fact that $\text{avg}_p(o)$ does not depend on o_p, since we have many training examples and we may consider the average constant on o_p and $\overset{(4)}{=}$ follows by noticing that $I_{j,p} = \frac{\partial net_p}{\partial w_j}$ is the input that the candidate unit receives from unit j for pattern p. ∎

The role of hidden units is to reduce the residual output error, solving a specific sub-problem or becoming a permanent "feature-detector". Typically, since the maximization of the correlation is obtained using a gradient ascent technique on a surface with several maxima, a pool of hidden units is trained and the best one selected. This behaviour favors avoiding local maxima.

5.7.8 Input scaling and Output representation

For the **input** we can take into account different points of preprocessing, that can have large effect:

- for different scale values, normalization via: (1) *standardization*, where for each feature v we obtain mean 0 and standard deviation 1 by $v - \text{avg}_v / std_v$, or (2) *rescaling* to the interval $[0, 1]$ by performing the transformation $v - \min_v / (\max_v - \min_v)$.
- for categorical inputs it is possible to map each value to a string of bits (one hot representation).
- in case of missing data we must pay attention on the fact that 0 does not mean "no input" if it is in the input range.

Considering the **output** we can operate on it in two different ways:

1. **regression**, the output is linear with the units, we have one output for a single unit or multi output for multiple quantitative response.
2. **classification**, here we have multiple outputs and many possibilities to classify them:
   - *1-of-k* encoding, where each output is coded as 0/1 or ±1;
   - sigmoidal activation, where we select some threshold to assign the target to the class;
   - Rejection zone;
   - softmax[‡] function for 0/1 targets, that can be interpreted as probability of the class, $\mathbb{P}(\text{class}|x)$. SOftmax works also on multi-class target values;
   - alternatives to softmax that minimize the "cross entropy".

5.7.9 More on loss functions

For a specific learning task it is crucial to choose an activation function that properly highlights the delta between the prediction and the target value. In a binary classification task the *cross entropy* loss is widely used. Let us warm up with some useful notation:

- $\forall i \in \{1, 2, \dots, l\}$ indicating a training example, we denote $y_i \in \{0, 1\}$ the target value;
- $\forall i \in \{1, 2, \dots, l\}$ indicating a training example, we denote $o_i \in [0, 1] \subseteq \mathbb{R}$ the predicted value;
- $\forall i \in \{1, 2, \dots, l\}$ indicating a training example, we denote $p_i(1) = \mathbb{P}[y_i = 1]$ and $p_i(0) = \mathbb{P}[y_i = 0]$. It goes without saying that $p_i(0) = 1 - p_i(1)$;
- $\forall i \in \{1, 2, \dots, l\}$ indicating a training example, we denote $q_i(1) = \mathbb{P}[\text{the model outputs 1}]$ and $p_i(0) = \mathbb{P}[\text{the model outputs 0}]$. Just like before, $q_i(0) = 1 - q_i(1)$.

Definition 5.7.4 (Cross-entropy)**.** *The **cross-entropy** or **log loss** or **logistic loss** of a binary classifier $h : \mathbb{R}^n \to \{0, 1\}$ is defined as follows:*

$$H(h) = -\frac{1}{l} \cdot \sum_{i=1}^{l} p_i(1) \cdot \log(q_i(1)) + (1 - p_i(0)) \cdot \log(1 - q_i(0))$$

Or, equivalently

$$H(h) = -\frac{1}{l} \cdot \sum_{i=1}^{l} y_i \cdot \log(o_i) + (1 - y_i) \cdot \log(1 - o_i)$$

[‡] $\sigma(\mathbf{v})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \forall i = 1, \dots, K$

Lemma 5.7.4. *For a given hypothesis $h : \mathbb{R}^n \to \{0, 1\}$, the partial derivative of $H(h)$ with respect to the j-th training example is computed as*

$$\frac{\partial H(h)}{\partial o_j} = -\frac{y_j}{o_j} + \frac{1 - y_j}{1 - o_j}$$

Proof.

$$\begin{aligned}
\frac{\partial H(h)}{\partial o_j} &= \frac{\partial \left(-\sum_{i=1}^{l} y_i \cdot \log(o_i) + (1 - y_i) \cdot \log(1 - o_i) \right)}{\partial o_j} \\
&\overset{(1)}{=} -\sum_{i=1}^{l} \left[\frac{\partial \left(y_i \cdot \log(o_i) \right)}{\partial o_j} + \frac{\partial \left((1 - y_i) \cdot \log(1 - o_i) \right)}{\partial o_j} \right] \\
&\overset{(2)}{=} -\left[\frac{\partial \left(y_j \cdot \log(o_j) \right)}{\partial o_j} + \frac{\partial \left((1 - y_j) \cdot \log(1 - o_j) \right)}{\partial o_j} \right] \\
&= -y_j \cdot \underbrace{\frac{\partial \left(\log(o_j) \right)}{\partial o_j}}_{\frac{1}{o_j}} - (1 - y_j) \cdot \underbrace{\frac{\partial \left(\log(1 - o_j) \right)}{\partial o_j}}_{\frac{1}{1-o_j} \cdot \frac{\partial(-o_j)}{\partial o_j} = -\frac{1}{1-o_j}} = -\frac{y_j}{o_j} + \frac{1 - y_j}{1 - o_j}
\end{aligned}$$

Where $\overset{(1)}{=}$ is justified by the fact that the derivative of the sum is the sum of the derivatives, while $\overset{(2)}{=}$ is explained by considering that both the quantities $y_i \cdot \log(o_i)$ and $(1 - y_i) \cdot \log(1 - o_i)$ are costant with respect to o_j for each $i \neq j$. ■

5.8 Pro and cons of Neural Networks

A neural network provides flexible methods for machine learning, extending linear models to arbitrary function estimation and constructing an explicit hypothesis. The expressive power is given by the number of units, the architecture and the training.

If the hypothesis space is a rich continuous space of functions and the loss function is differentiable, then the training works moving towards minima, making use of the gradient descent technique.

What follows is a list of the pros of using neural networks for learning tasks. Neural networks

- ✔ can learn from examples;
- ✔ are universal approximators, that provide flexible approaches for arbitrary functions;
- ✔ can deal with noise and incomplete data;
- ✔ can handle continuous, real and discrete data for both regression and classification tasks;
- ✔ are successful model in machine learning due to the flexibility in applications;
- ✔ encompass a wide set of models, it is a paradigm;
- ✔ are distributed representation of knowledge, which is easily stored as weight matrices;
- ✔ have fault tolerance (robustness);
- ✔ adapt to changes (nonstationary environment);

Neural networks are also afflicted by critical points. One of this is the so called *black-box problem*, which is the inability to interpret or explain the acquired knowledge. An example of this problem is the extraction and representation of knowledge for human experts. Moreover this problem is related to the current trend in the research for interpretability of machine learning models and the "explainable artificial intelligence" issue.

Other critical points are the architectural design and the training process dependency, where the training realization have effect on the final solution.

5.9 Convolutional neural networks

CNNs are regularized versions of multilayer perceptrons, that overcome the issue of overfitting due to the "fully-connectedness" of these networks. Typical ways of regularization of MLP include adding some form of magnitude measurement of weights to the loss function. However, CNNs make use of a different approach towards regularization: they take advantage of the hierarchical pattern in data and assemble more complex patterns using smaller and simpler patterns.

Convolutional networks were inspired by biological processes: the connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a *restricted region* of the visual field known as the *receptive field*. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

Intuitively, a convolutional neural network works taking only a part of the inputs (**sliding window**) using every time the same weights (**weight sharing**).

Example:

Let $o_i = \sum_{i=1}^{3} w_i x_{t+i-2}$. A convolutional neural network that implements such a function can be seen in Figure 5.18. In such CNN $o_2 = w_1 x_1 + w_2 x_2 + w_3 x_3$ and $o_3 = w_1 x_2 + w_2 x_3 + w_3 x_4$.

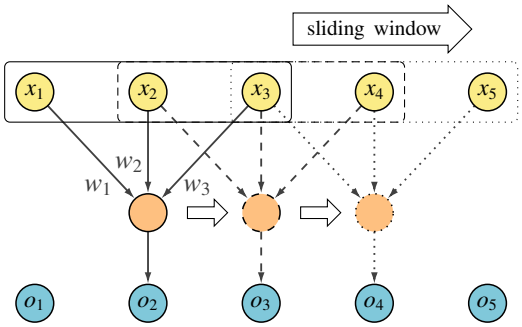


Figure 5.18: In this example, we can see that x_3 is taken 3 times to give three different outputs. The weights w_1, w_2 and w_3 are used for each triplet of inputs (**weights sharing**).

The key point here is to introduce architectural constraints that produce the *strongest response* to spatially local input patterns.

Convolutions can be seen also in 2 dimensions as displayed in Figure 5.19, where a bidimensional group of inputs is taken to give place to one output.

Definition 5.9.1 (CNN kernel)**.** *We call **kernel** or **filter** or **feature detector** the (possibly multi-dimensional) function that is applied to any portion of the inputs at each step.*

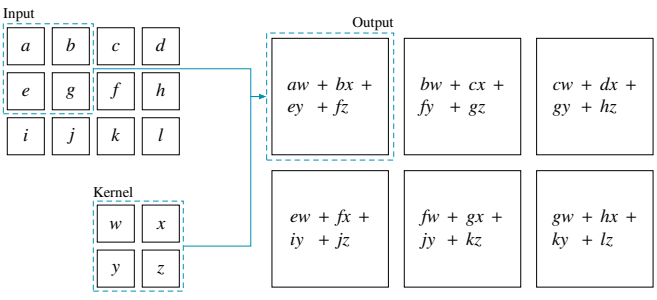


Figure 5.19: An example of a 2-D convolutional neural network.

Definition 5.9.2 (Stride)**.** *We call **stride** a couple of integers that repsresent the x-shift and the y-shift of the window at each step.*

Formally, the generic element in position (i, j) of the output matrix S of a convolutional neural network is obtained as

$$S_{ij} = K_{mn} \times I[i : i + m - 1; j : j + n - 1]$$

where we denote with K_{mn} the kernel 2-D matrix of dimension $m \times n$ and we assume the stride to be 0, otherwise we have $\frac{i}{\text{stride}_i}$ and $\frac{j}{\text{stride}_j}$.
Notice that it is possible to design learnable filters, which are inferred from the problem and are not hand-engineered.

Definition 5.9.3 (Pooling)**.** *We call **pooling** a possible approach for down sampling feature maps by summarizing the presence of features in patches of the feature map.*

Two common pooling methods are average pooling and max pooling (Figure 5.20) that summarize the average presence of a feature and the most activated presence of a feature respectively.

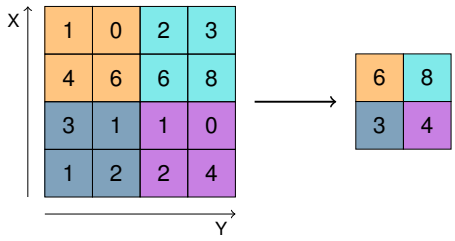


Figure 5.20: Max pooling with a 2×2 filter and stride = 2 (no overlapping).

We can observe that the max pool operation further helps to make the representation become approximately invariant to small translations of the input, hence *noise-tolerant* and also reduce the dimensionality of the problem.
As a whole, a convolutional neural network performs multiple alternating steps of convolution and pooling, until a feed-forward neural network is built, see Figure 5.21.

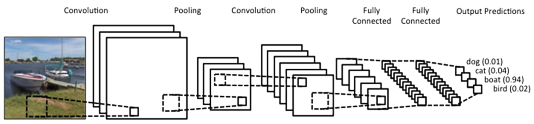


Figure 5.21: Example of CNN, made of multiple layers performing both pooling and convolutional operations.

The training of convolutional neural networks is performed making use of the back-propagation technique, already presented previously in this chapter.
Convolutional networks are very well-performing on image recognition. As an example, ImageNet[§] is a neural network (made of 60 millions of parameters and 500′000 neurons) that consists of five convolutional layers, some of which are followed by max-pooling layers, and two globally connected layers with a final 1000-way softmax.
In neural networks, matrix-matrix products are performed very often and therefore some performance improvements can be achieved making parallel computations on GPU.

Definition 5.9.4 (Tensor)**.** *We term **tensor** the generalization of a matrix in a n-dimensional space.*

The tensor notation is very useful when we are trying to parallelize operations inside a neural network.

[§]Krizhevsky, Sutskever, Hinton, ImageNet Classification with Deep Convolutional Neural Networks, NIPS 2012

6. More on Neural Networks

6.1 Deep Learning

Mantra

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These methods have dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains.

The main characteristic of deep learnig is that they exploit a big number of layers to focus on some relevant concepts, such as *representation learning* and *distributed representation*.

In general, layers of MLPs form a hierarchy from low-level to high-level features, leading to different levels of abstraction.

Example:

Let us take the example of image recognition: we want to move from pixels to object identity.

Deep learning resolves this difficulty by breaking the desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. A series of Hidden layers extracts increasingly abstract features from the image, as can be observed in Figure 6.1.

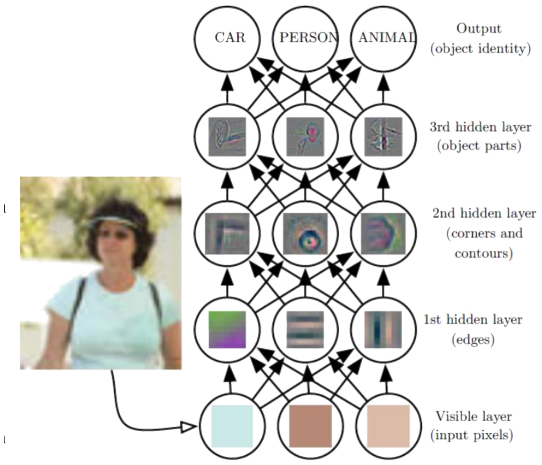


Figure 6.1: Max pooling with a 2×2 filter and stride $= 2$ (no overlapping).

The abstract features that have been identified can also be combined to *generalize* to *unseen* examples, as shown in Figure 6.2.

Our claim in talking about deep learning is that deep neural networks come in handy whenever a problem may be easily expressed using a multilayer neural network, while it is hardly representable with a much shallower one.

Example:

As already seen in Section 5.1, the binary XOR function cannot be expressed using a single layer perceptron.

Let us consider the more general example of multiple input XOR. In general, the XOR function returns

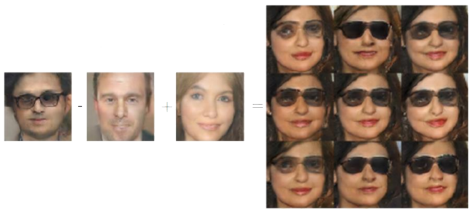


Figure 6.2: Combining higher level features to generalize to "woman with glasses".

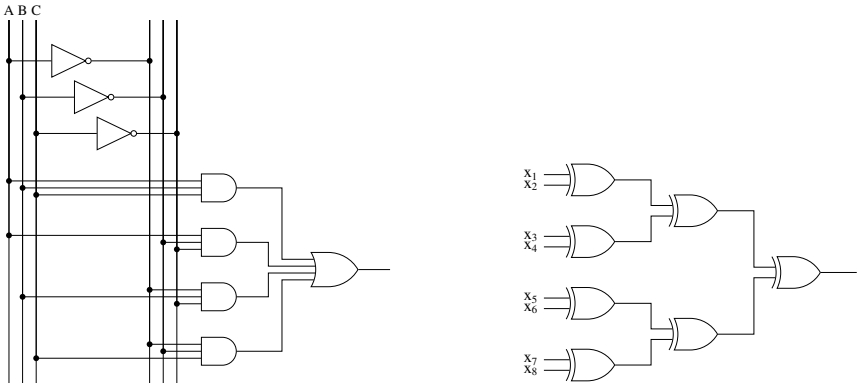
true if and only if the number of inputs is *odd*. Thanks to this more general definition we can write the following

$$\text{XOR}(x_1, x_2, x_3) = x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3}$$

where $\cdot$ stands for logical AND, $+$ represents the OR and $\overline{*}$ is for negation.
An attentive reader may notice that the number of disjunctions is exaclty equal to the number of possible choices of configurations of the inputs with an odd number of non-negated literals. Thanks to this observation we may deduce the general rule for counting the number of disjunctions, denoting with d_i the number of *odd* integers in the range $[1, n]$, $\sum_{d_i} \binom{n}{d_i} = O(2^{n-1})$.

This simple implementation of AND and OR gates leads to a tree with depth 2, but ariety exponential in the number of inputs: for N inputs there are $2^{N-1} + 1$ gates.

The schema of the example with 3 inputs is represented in Figure 6.3(a). While a cleverer implementation may build a deeper *binary* tree can be observed in Figure 6.3(b).



(a) Example of naive XOR implementation with only 3 inputs

(b) Binary tree for XOR with 8 inputs

Figure 6.3: Two examples of implementations of the logical XOR

This example shows also that a deep neural network is not necessarily a huge model, it is indeed a way to write a more compact model with respect to potentially larger shallow alternatives. Deep neural networks also allow learning using a smaller number of examples, by extracting salient features (hidden layers).

Some very lucky tasks for deep learning are those that have a hierarchical structure, for example images (composition of subgraphical parts) and language (complex union of parts of the speech). In these cases, it happens often that better performances in terms of generalization are achieved using a deep neural network if compared with a model that has less layers and the same number of units.

How to choose the best number of layers and units? It is true in general that deeper networks are able to work exploiting less units per each level, hence requiring a smaller number of parameters and less training data. It important to highlight that for a given number of parameters deeper networks impose more smoothness than shallow ones. In particular, each layer works on the already smooth surface which is the output by the previous layer.

6.1.1 Hidden representation

In machine learning, representing the inputs in a proper manner is a crucial task that makes the learning task easier.

Deep learning searches a representation on multiple hierarchical levels.

Such information about data is obtained through the so-called *pretraining* technique, which should be followed by a *fine-tuning* phase (proper training of the model).

Definition 6.1.1 (Autoencoder)**.** *We call **autoencoder** a neural network trained to learn efficient data codings in an unsupervised manner. Internally, it has a hidden layer h, which is a code used to represent the input. A function g(h) should return the original input (see Figure 6.4).*

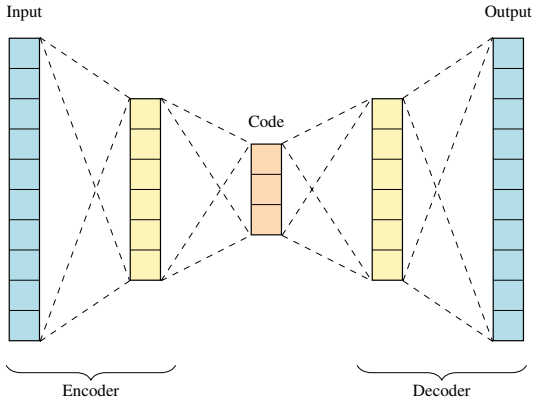


Figure 6.4: An example of an autoencoder.

Autoencoders can be divided into two big families:

- **undercomplete:** where the hidden layer is smaller than the input. This implementation forces to capture the most salient features of the training data by architectural constraints;
- **overcomplete:** in this case, the hidden layer is grater than the input. This approach leads to sparse results, which can be regularized, adding robusteness to noise and/or other properties of interest beside the trivial (for the over complete case) input-output "copy" capability.

Following we will discuss how obtain the hidden representation, *pretraining*, and how exploit it, *transfer learning*.

Pretraining

Pretraining is introduced to overcome the issue of weights initialization of a neural network. The initial values of the weights have nothing to do with the task that we are trying to solve. Why should a set of values be any better than another set? If we knew how to initialize them properly for the task, we might as well set them to the optimal values (slightly exaggerated). Pretraining gives the network a head start, as if it has seen the data before.

1: Train the first layer as an autoassociator to minimize the reconstruction error of the raw input ▷ Unsupervised
2: **repeat**
3: The hidden units' outputs in the autoassociator are now used as input for another layer ▷ Unsupervised
4: **until** the desired number of layers is obtained
5: Take the output of the last hidden layer as input to a supervised layer and initialize its parameters (either randomly or by supervised training, keep the rest of the network fixed)
6: Fine-tune all the parameters of this deep architecture with respect to the supervised criterion.

Algorithm 6.1: The PRETRAINING algorithm by Y. Bengio.

In this setting, the predictive task takes as inputs the new representation of the domain to produce the objective function.

Transfer Learning

We refer to **transfer learning** when we use the representation discovered in a model to improve another model. While we say that we perform **multi-task learning** when we use a trained model for another task (same inputs, different target). Conversely, we can have a **domain adaptation** changing the input domain.

6.1.2 Localist & distributed representation

With deep learning we allow **distrubuted representation** of inputs, meaning that we shift from classifying to *representing* inputs using normalized values that can represent how much an input "belongs" to a certain symbol/concept. The distributed representation is opposite to the so-called **localist-representation** (aka one hot representation). An example of this distinction can be seen in Figure 6.5.

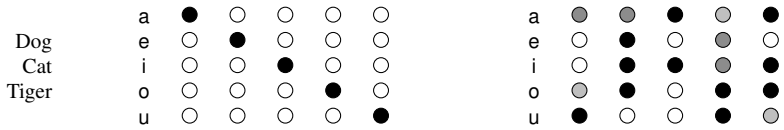


Figure 6.5: On the lefthand side a *localist representation*, while on the right a *distributed representation*.

Disentangling concepts

Distributed representation can represent n features with k values to describe k^n different concepts and this allows to **disentangle** concepts, meaning that some attributes may be "applied" to elements belonging to different classes.
As an example, we can represent the four concepts: red car, red bike, blue car, blue bike using two neurons, one representing the color and the other the vehicle.
The fact of sharing some features between different entities allows the model to treat them similarly (when needed by the context) and to automatically learn such similarity.

> **Example:**
> The distributed representation process applied to words gives place to the so-called **word embeddings**, which are representations of words in a vector space, which allows to grasp similarity between words.
> In a 2-dimensional space and example of word embeddings can be observed in Figure 6.6

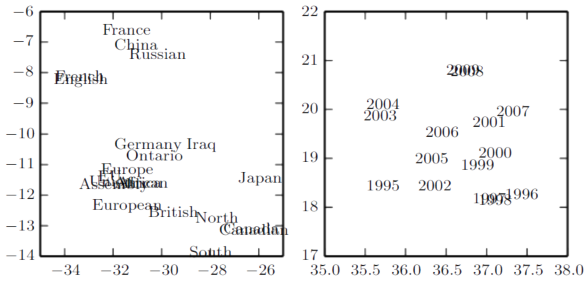


Figure 6.6: Both words and numbers are clustered in the vector space.

Notice that sometimes understanding the features extracted by a DL model can be quite challenging (explainability issues).

Mantra

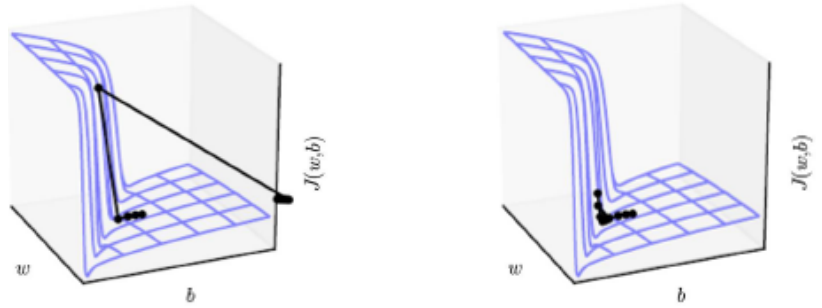
A main puzzle of deep networks revolves around the absence of overfitting despite large overparametrization and despite the large capacity demonstrated by zero training error on randomly labeled data.

Gradient issues

What happens to the magnitude of the gradients as we backpropagate through many layers?

- If the weights are small, the gradients shrink exponentially (especially down-sized by Jacobian of sigmoidal function)
- If the weights are big the gradients grow exponentially

For the second element, the difficulty that arises is that when the parameter gradient is very large, a gradient descent parameter update could throw the parameters very far (Figure 6.7(a)), into a region where the objective function is larger, undoing much of the work that had been done to reach the current solution.



(a) High values of derivatives bring W very far

(b) Moderate reaction to cliff by clipping

Figure 6.7: Comparison of the learning with and without clipping techniques.

A common solution to the problem of exploding gradients is to change the error derivative before propagating it backward through the network and using it to update the weights. By rescaling the error derivative, the updates to the weights will also be rescaled, dramatically decreasing the likelihood of an overflow or underflow.
There are two main methods for updating the error derivative; they are:

- **Gradient scaling** involves normalizing the error gradient vector such that vector norm (magnitude) equals a defined value, such as 1.0
- **Gradient clipping** involves forcing the gradient values (element-wise) to a specific minimum or maximum value if the gradient exceeded an expected range (see Figure 6.7(b)). Formally, if the gradient's norm exceeds a certain threshold t ($\|g\| > t$) we "scale" it as $g := t \cdot \frac{g}{\|g\|}$.

In machine learning, the **vanishing gradient** problem is a difficulty found in training artificial neural networks, in particular deep ones, since each of the weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of this behaviour take a traditional activation function, such as the *hyperbolic tangent**. Such function has gradients in the range $(-1, 1)$, with very gentle slope that leads to the so-called "neuron saturation". This has the effect of multiplying N of these small numbers to compute gradients of the "front" layers in an N-layer

*The **hyperbolic tangent** is defined as $\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x}-1}{e^{2x}+1}$

network, meaning that the gradient (error signal) decreases exponentially with N while the front layers train very slowly.

Such an issue can be overcome using ReLU (Rectified Linear Unit - Item 6 and Figure 5.7(d)) as activation function, since it is easy to compute and does not have any vanishing effect.
It goes without saying that the ReLU activation function does not have any gradient in 0, but it is assumed simply to be 0 (left derivative) or 1 (right derivative). This approximation is acceptable and safe because it is very unlikely to be in a situation in which the input's numerical approximation is exactly 0, so

$$\frac{\partial f_\sigma(\mathbf{x})}{\partial \mathbf{x}} = \begin{cases} 0 & \text{if } \mathbf{x} < 0 \\ 1 & \text{if } \mathbf{x} > 0 \end{cases}$$

and arbitrarily 0 or 1 in zero.

Tips for the project

Definition 6.1.2 (Dropout)**.** *We call **dropout** the technique of ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By "ignoring", it is meant that these units are not considered during a particular forward or backward pass.*
More technically, at each training stage, individual nodes are either dropped out of the net with probability $1 - p$ *or kept with probability* p*, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.*

This technique is very useful when we are trying to avoid overfitting. The dropout technique can be seen from the ensembling point of view since it corresponds to training the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network.
For a pictorial representation of this technique see Figure 6.8.

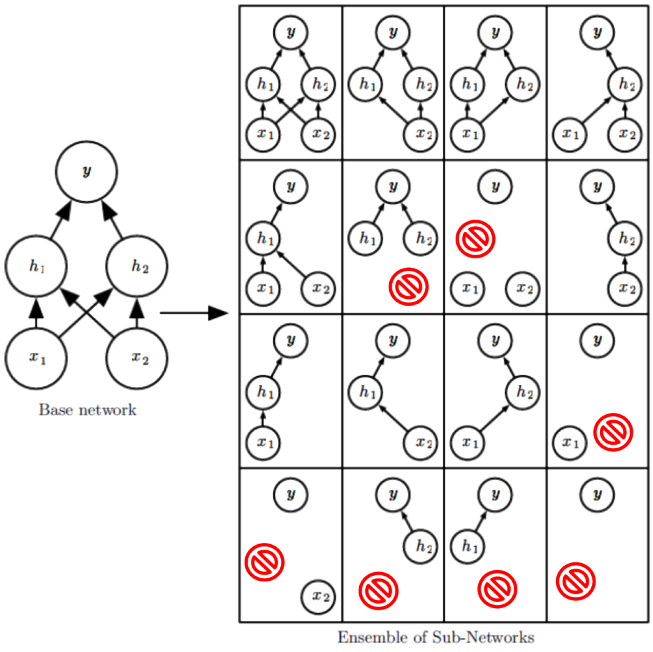


Figure 6.8: The networks with the red sign do not make any sense (no input), but whith larger networks it is very unlikely to have no path from the input to the output

The training when using dropout is carried on using *minibatch* as follows: first, a probability p is chosen (e.g. 0.2), then for each input example in the minibatch the 20% of the nodes in the network are dropped randomly and the model is trained (feed-forward + backpropagation).
Once a minibatch has been used, the nodes are reintroduced with the same weights they had before.

- Start with positive nets (initializing the bias to a small, positive value, such as 0.1). This makes it very likely that the rectified linear units will be initially active for most inputs in the training set and allow the derivatives to pass through.
- Some pretraining techniques can be used to obtain a smart initialization of the weights.
- *Batch normalization*: to increase the stability of a neural network, batch normalization normalizes the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

  We can use higher learning rates because batch normalization makes sure that there's no activation that's gone really high or really low. And by that, things that previously couldn't get to train, will start to train. It reduces overfitting because it has a slight regularization effect. Similar to dropout, it adds some noise to each hidden layer's activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information.

6.2 Extreme Learning Machine

The so-called Extreme Learning Machine (ELM) is one of the recent neural network paradigms and it includes *Randomized Machine Learning* and *Randomized Weights Neural Networks*.

Randomized Machine Learning

A randomized approach employs a degree of randomness as part of its constructive strategy. The randomness can be viewed as an effective part of machine learning approaches and it can enhance several aspects of machine learning methodologies.

Randomness can be present at different levels for enhancing predictive performances or to alleviate difficulties of classical machine learning methodologies. Moreover, randomness can be found in specific aspects of machine learning, like data processing or learning and hyperparameter selection. An example is the case of Cross Validation, where the data sections assigned to training, validation and test sets are selected using a random function.

Randomness can also be a key factor for building efficient machine learning models, some examples are the random neurons of Restricted Boltzmann Machines, the regularization or the Random Forests. The latter is an algorithm used to solve classification problems and it is based on random points: each decision tree is build from samples randomly drawn from the training set, moreover, the selection of the input variables for splitting the nodes is randomized.

Randomized Weights Neural Networks

Extreme Learning Machine aims at answering to the following question: *Is it possible to exploit the MLP/RNN architectures even without long training cycles in the hidden layers?* The basic idea is to have a network containing a randomly connected hidden layer(s) where weights are fixed after random initialization, and then we can train only the output weights. In this way we overcome the problems associated with the complex computationally demanding training algorithms traditionally used to train neural network.

In the neural network area randomness gave rise to a rich set of models that are rooted in the pioneering work on the Perceptropn. Other works pointed out to relevance of learning in the output labels with respect to the hidden layers. Recently, randomized neural networks delineate a research line of growing interest, particularly suitable when efficiency is of primary relevance.

Figure 6.9 represents a general architectural structure of a randomized neural network, composed by an untrained hidden layer based on randomness and a trained readout layer. The untrained hidden embes in a non linear way the input into a high-dimensional feature space where the problem is more likely to be solved linearly. This work is based on the Cover theorem.

Theorem 6.2.1 (Cover)**.** *A complex pattern-classification problem, cast in a high-dimensional space nonlinearly, is more likely to be linearly separable than in a low-dimensional space, provided that the space is not densely populated.*

Proof. In particular, using a deterministic mapping: Assuming n examples, lift them onto the vertices of the $n-1$ dim triangle (simplex). Now, every binary partition of the samples is linearly separable. ■

In a randomized neural network, the trained readout layer combines the features in the hidden space for output computation. This process is typically implemented by linear models.

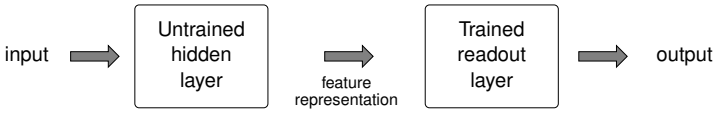
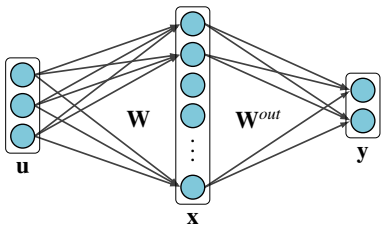


Figure 6.9: General architectural structure.

The input-output relation is so implemented through an untrained hidden layer that implements randomized basis expansion. Imagining to have a real neural network like the following one



where $\mathbf{u}$ and $\mathbf{y}$ are respectively the input and output vector, $\mathbf{W}$ is the weight matrix for the hidden layer and $\mathbf{W}^{out}$ is the readout weight matrix. Naming f the activation function, we can write the network as:

$$\mathbf{y} = \begin{bmatrix} \sum_{j=1}^{N_X} w_{1,j}^{out} \cdot f(\mathbf{w}_j \mathbf{u}) \\ \cdots \\ \sum_{j=1}^{N_X} w_{N_Y,j}^{out} \cdot f(\mathbf{w}_j \mathbf{u}) \end{bmatrix} = \mathbf{W}^{out} f(\mathbf{W}\,\mathbf{u})$$

where $\mathbf{W}^{out}$ can be rewritten compactly as a LMS problem often implemented with the L2 regularization to reduce the useless features:

$$\mathbf{W}^{out} = \arg\min_{\mathbf{W}^{ro}} \left\{ \left\| \mathbf{W}^{ro}\mathbf{X} - \mathbf{Y}_{target} \right\|_2^2 \right\}$$

The large set of hidden units play an interesting role: such a projection expands non-linearly the dimension of an input vector so as to make the data more separable. Moreover, a large number of units can provide a sufficient basis expansion to solve the task at hand. Finally, this kind of networks are extremely efficient, many variants and improvements exist today.

6.3 Unsupervised Learning

This section focuses on *clustering* and *vector quantization*. The clustering problem has been addressed in many contexts and used by researches in many different disciplines. There exist thousands of different algorithms, but not all of these are important for our study so we prefer to concentrate on a simple approach of a historical relevance for neural network models: the Self Organization Maps (SOM).

A key point that is present in all these models is the metric, which tells us which are the similar examples. A clustering algorithm decides the grouping based on that metric. There are many possible metrics, Euclidean metric is popular and it is a reasonable choice (like for the Nearest Neighbor in Chapter 4).

6.3.1 Vector quantization and clustering

The clustering methods provide a partition of data into clusters (subsets of "similar" data) where patterns within a valid cluster are more similar to each other than they are to a pattern belonging to a different cluster.

Each cluster has a *centroid* (or *prototype*) that corresponds to the center, mathematically the average of all the point in the cluster.

Vector quantization techniques encode a data manifold, e.g. a sub-manifold $V \subseteq \mathbb{R}^n$ using only a finite set $W = (\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_N)$ of reference vectors $\mathbf{w}_i \in \mathbb{R}^n$, $i = 1, 2, \ldots, N$. A data vector $\mathbf{x} \in V$ is described by the best-matching or winning reference vector $\mathbf{w}_{i^*(\mathbf{x})}$ of $\mathbf{w}$, for which the distortion error $d(\mathbf{x}, \mathbf{w}_{i^*(\mathbf{x})})$ is minimal. This procedure divides the manifold V into a number of subregions V_i defined as

$$V_i = \left\{ \mathbf{x} \in V \text{ s.t. } \|\mathbf{x} - \mathbf{w}_i\| \leq \left\|\mathbf{x} - \mathbf{w}_j\right\| \ \forall j \right\}$$

V_i is called Voronoi polyhedron and out of it each data vector $\mathbf{x}$ is described by the corresponding reference vector $\mathbf{w}_{i^*(\mathbf{x})}$.

Figure 6.10: Example of a Voronoi diagram.

In Figure 6.10 there is an example of a Voronoi diagram. Each cell consists of all points closer to $\mathbf{x}$ than to any other pattern. The segments of the diagram are all the points in the plane that are equidistant from two patterns. A formalization of this problem is: *given the dataset, find centers such that minimize loss E*. In general, the Voronoi diagram is hard to compute. An easy example is the quantization in (one dimension) digital signal processing. Here quantization is the process of approximating a continuous range of values by a relatively small set of discrete symbols or integer values.

Clustering has a more general objective, namely the one of finding interesting (or useful) groupings of data. The interesting concept is often implicitly defined via the computational procedure by itself and not necessarily measured by the minimum quantization error.

As explained before, the goal is to find an optimal partitioning of an unknown distribution in the input space into regions (clusters) approximated by a cluster center or prototype. We can see the problem as a function that maps a continuous space of $\mathbf{x}$ into a discrete space of the number of centers. The average value over the distribution of inputs is the average (expected) distortion or quantization error.

Definition 6.3.1 (Quantization error function)**.** *The* ***quantization error function*** *accounts for the difference between the true vector and its quantized version. Formally, it can be written in two forms:*

- squared error criterion *(eq. (6.1))*

$$E = \int f(d(\mathbf{x}, \mathbf{w}_{i^*(\mathbf{x})}))p(\mathbf{x})\, d\mathbf{x} = \int \left\|\mathbf{x} - \mathbf{w}_{i^*(\mathbf{x})}\right\|^2 p(\mathbf{x})\, d\mathbf{x} \tag{6.1}$$

  where $p(\mathbf{x})$ is the probability distribution of $\mathbf{x}$

- discrete version *(eq. (6.2)):*

$$E = \sum_i \sum_j \left\|\mathbf{x}_i - \mathbf{w}_j\right\|^2 \delta_{winner}(i, j) \tag{6.2}$$

  where δ_{winner} in eq. (6.2) is the characteristic function of the receptive field of $\mathbf{w}_j$ and has value 1 if $\mathbf{w}_j$ is the winner for $\mathbf{x}_i$ and 0 otherwise.

For Equation (6.1), the set of reference vectors $\mathbf{w}$ that minimizes E is the solution of the vector quantization problem. Note that in this equation, the integrand of E is not continuously differentiable. Indeed $\mathbf{w}_{i^*(\mathbf{x})}$ it has discrete changes, however we compute derivatives locally, fixing $\mathbf{x}$ on a Voronoi cell. Taking now the Equation (6.2) and the derivatives of E with respect to the $\mathbf{w}_j$ yields the learning rule of vector quantization.

Lloyd and McQueen's provide the well-known K-means clustering algorithm in the on-line version, that, for the learning rate η, is based on

$$\Delta \mathbf{w}_{i^*} = \eta \delta_{winner}(i, i^*)(\mathbf{x}_i - \mathbf{w}_{i^*})$$

Notice that the only thing that changes is the set of winner prototypes, adapting to $\mathbf{x}$. Moreover, the method depends on the initialization and needs to be run using as hyperparameter the number of clusters. The generalized Lloyd algorithm is reported in Algorithm 6.2 and a demonstration of its steps is showed in Figure 6.11.

1: Chose k cluster centers $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k$ to coincide with k randomly-chosen patterns or k randomly defined points inside the hypervolumne containing the pattern set
2: **repeat**
3: Assign each pattern to the closest cluster center, for each $\mathbf{x}$ the next prototype is

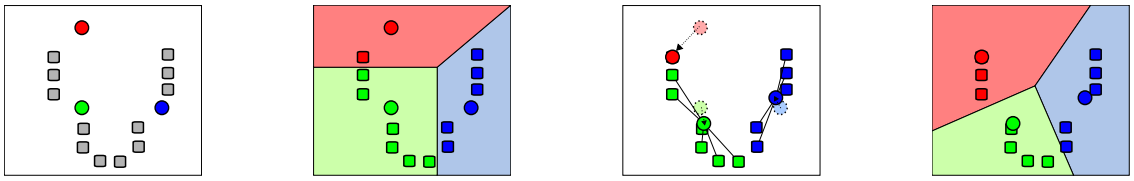
$$i^*(\mathbf{x}) = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2^2 = \arg\min_i \sum_{j=1}^{n} (\mathbf{x}_j - \mathbf{w}_{ij})^2$$

4: Recompute the cluster centers by the current cluster membership, for each cluster C_i the new centroid is

$$\mathbf{w}_i = \frac{1}{|C_i|} \sum_{j:\mathbf{x}_j \in C_i} \mathbf{x}_j$$

5: **until** a convergence criterion is met, e.g. no reassignment of patterns to new cluster centers or miniminal decrease is squared error

Algorithm 6.2: The generalized Lloyd K-means algorithm.



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Figure 6.11: Demonstration of the Algorithm 6.2 steps.

k-means is the simplest and most commonly used clustering algorithm. It employs a squared error criterion and it is also popular because it is easy to implement and efficient.
The negative sides are that the number k of clusters must be provided and local minima of E make the method dependent on initialization leading to sub-optimal choice of the reference vectors. Luckily, it works very well for compact and hyperspherical cluster.

In order to avoid confinement to local minima, a common approach is to introduce "soft-max" adaption rule. This method modifies not only the winning reference, but the adaptation to $\mathbf{x}$ affects all cluster centers depending on their proximity to $\mathbf{x}$, typically with a step size that decreases with the distance $d(\mathbf{x} - \mathbf{w}_i)$.

An instance of this strategy in the field of neural networks is the *Kohonen self organization maps*, where the proximity among reference vectors are arranged.

6.3.2 Self Organizing Map

A Self Organization Map (SOM) is a type of neural network, trained using *unsupervised learning* that has the aim of creating a *low-dimensional discretized* representation of the input (used for dimensionality reduction). The idea behind SOM is to cause different parts of the network to respond similarly to certain input patterns (just like what happens in the human brain).

A SOM consists of N neurons located on a regular low-dimensional (usually 2D) grid (lattice) where each unit (i) can be identified by the coordinate on the map, (ii) receives the same input $\mathbf{x}$ and (iii) has a weight $\mathbf{w}$ ($\dim(\mathbf{x}) = \dim(\mathbf{w})$). Here we have a topology-preserving map from the fact that neighboring units respond to similar input pattern and the data points close in the input space are mapped onto the same nearby map units.

The SOM can be used in various cases like *clustering*[†], where clusters can be identified on the map, *pattern recognition* and *vector quantization*, in which the input is transformed into the closest neuron weights. Others use cases can be the *data compression*, where the inputs is transformed into the indexes, or digital codes, of the winner unit, finally the *projection* and *exploration*, for which the distribution of the multidimensional data is visualized in a lower dimensional space.

Basically, at each step, the input vector is compared to the wiehgts. The weight closer to $\mathbf{x}$ is taken and the units in its neighbourhood are moved closer to $\mathbf{x}$. Algorithm 6.3 reports the passes of the SOM procedure and Figure 6.12 shows how the map changes during the training of the algorithm

1: Random initialization of the map (weight values)
2: **repeat**
3: Drawn a sample input $\mathbf{x}$
4: **Competitive stage**, the winner is the unit (with $\mathbf{w}$) most similar to $\mathbf{x}$. The current input vector $\mathbf{x}$ is compared with all the unit weights using an Euclidean distance criteria, adopting a "winner-take-all" strategy, the winner, for an index i on the map, is

$$i^*(\mathbf{x}) = \arg\min_i \|\mathbf{x} - \mathbf{w}_i\|_2$$

5: **Cooperative stage**, upgrade the weights of the units that have topological relationships with the winner unit. The weights of the winning unit and the weight of the units in its neighborhood are moved closer to the input vector $\mathbf{x}$. At iteration t we have

$$\mathbf{w}_i(t+1) = \mathbf{w}_i(t) + \eta(t) h_{i,i^*}(t)[\mathbf{x} - \mathbf{w}_i(t)]$$

where $h(t)$ is the neighborhood function, i.e. $h_{i,j}(t)$ is a function that decreases monotonically for increasing $\|i - j\|$:

$$h_{i,i^*}(t) = \exp\left(-\frac{\|\mathbf{r}_i - \mathbf{r}_{i^*}\|^2}{2\sigma_{nh}^2(t)}\right)$$

where $\mathbf{r}_i$ are the coordinates of the unit i and n is the weight.
6: **until** a convergence criterion is not met, e.g. no changes

Algorithm 6.3: The Self Organizing Map (SOM) algorithm.

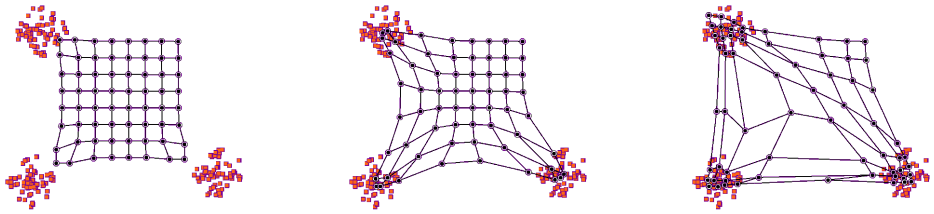


Figure 6.12: Changing of the map during the training phase of the SOM.

The learning rate and the neighborhood radius values are decreased as function of iteration t. Neighborhoods are wide at the beginning and then width and height slowly decrease during learning. In practice, the shape of the neighborhood is chosen from standard geometrical form to include a finite set of the map units.

It is important to stress that the update rule of the cooperative stage is fundamental to the formation of topographically ordered maps In fact, the weights are not modified independently from each other but as topologically related subsets. For each subset similar kinds of weight updates are performed. For each step a subset is selected on the basis of the neighborhood of the current winner unit. Hence, topological information is supplied to the map because both the winning unit and its lattice neighbors receive similar weight updates that allow them, after learning, to respond to similar inputs. Although this seems quite natural, a formal proof of ordering is difficult.

[†]Compared to K-means algorithm, SOM is more costly in terms of computational complexity, but overcomes some of the issues of K-means (it does not depend on initialization, it is less sensitive to noise and it does not need to know the number of clusters in advance)

The map can be used to visualize different features of the SOM and of the data represented on the map. For example, the density of the reference vectors or the distance between prototype vectors of neighboring units.

6.4 Recurrent Neural Networks

Until now we have analyzed *feedforward* neural networks, which take some data as inputs, process them, and produce an output. In this kind of networks the input of each node is the output of previous nodes, so the data moves forward. But this in not the only type of neural network: there are also networks, called **Recurrent Neural Network** (RNN), in which the output of a node can be an input for a previous node, creating some some loops of the computed data inside the network.

This kind of neural networks are very good for modelling situations where the output of the model depends on the history of the inputs, or the domain is composed by varying-size sequences. Some examples of these sequential data are the dynamical processes, temporal series and genome or protein sequences. Nowadays the recurrent neural networks are an approach for sequence processing, especially for speech recognition, text processing or, even more, music and text composition.

So far, the data fed to neural networks was expressed in a *flat* form, as vectors. There also exist *structured* versions of data, namely sequences, trees, graphs or multi-relational (further references in Chapter 10). Let us assume to have the data as a discrete sequence of vectors, serially ordered (e.g. time), the task is to perform a *transduction*: mapping the input sequence into a value or a new sequence.

The aim is to produce an output that depends on the previous inputs. We need to take into account two facts: first the input delay of the neural networks and, second, that memory is finite. We can imagine to have a register with finite size and a sliding window over the input sequence. At each iteration, the oldest information is deleted to have sufficient space for the result of the new iteration.

Figure 6.13 shows the architecture of a recurrent neural unit with feedback. The neural unit takes as input $\mathbf{l}_t$, as the input label at time t, the weight $\mathbf{W}$ and the recurrent weight $\hat{\mathbf{W}}$; and produces $\mathbf{x}_t$:

$$\mathbf{x}_t = \tau(\mathbf{l}_t, \mathbf{x}_{t-1}) = f(\mathbf{W}\mathbf{l}_t + \hat{\mathbf{W}}\mathbf{x}_{t-1} + \theta)$$

where τ is the state transition function realized by a neural network, f is the activation function and θ is the bias. Then the output $\mathbf{y}_t$ corresponds to the application of the activation function g on the evaluation of the inputs $\mathbf{l}_t$ and $\mathbf{x}_t$: $\mathbf{y}_t = g(\mathbf{l}_t, \mathbf{x}_t)$.

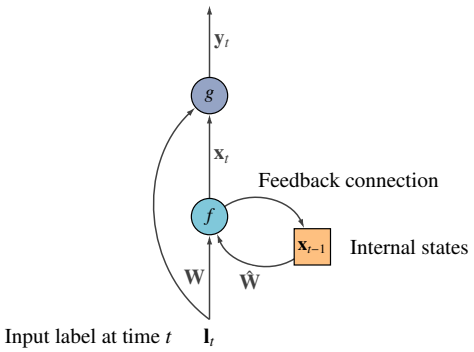


Figure 6.13: Illustration of a recurrent model with one recurrent unit.

Example:
Realize a 1-unit RNN (find the values for the weights $\mathbf{W}$ and $\hat{\mathbf{W}}$) which outputs the sum of 1 received in input so far. Drawn the flow of input ($\mathbf{l}_t$), state ($\hat{\mathbf{W}}\mathbf{x}_{t-1}$) and output ($\mathbf{x}_t$) values time-by-time ($t$) for example 101101.This task can be solved for an activation function with the following simple computation, for $\mathbf{W} = 1$ and $\hat{\mathbf{W}} = 1$:

$$\mathbf{x}_t = f(\mathbf{W}\mathbf{l}_t) + \hat{\mathbf{W}}\mathbf{x}_{t-1}$$

and the result for input sequence 101101 is computed as shown in Table 6.1.

Time	$t =$	1	2	3	4	5	6
Input	$\mathbf{l}_t =$	1	0	1	1	0	1
State	$\hat{\mathbf{W}}\mathbf{x}_{t-1} =$	0	1	1	2	3	3
Output	$\mathbf{x}_t =$	1	1	2	3	3	4

Table 6.1: Flow of input, state and output, for sequence 101101 and $\mathbf{W} = \hat{\mathbf{W}} = 1$.

Of course many architectures are possible, however the simple RNN seen before is already powerful. Recurrent neural networks are based on the following assumptions:

- *causality*, a system is causal if the output at time t_0 (or vertex v) only depends on inputs at time $t < t_0$ (respectively depends only on v and its descendants);
- *stationarity*, time invariance, state transition function τ is independent on node v;
- *adaptivity*, transition functions are realized by neural network (with free weight parameters), hence they are learnt from data.

The supervised learning algorithms BPTT (back-propagation through time) and the RTRL (real time recurrent learning) are designed for recurrent neural networks. They compute in different ways (the same) gradient values of the output errors across an unfolded over time network that is equivalent to the recurrent one (encoding networks).

An **Echo State Network** (ESN) consists in (i) a large reservoir of sparsely connected recurrent units, untrained after random initialization, and (ii) a simple feed-forward readout of linear units, trained by efficient linear methods.

Appendix 6.B Boltzmann Machines

Although this is not part of the exam, we would like to make a digression on such machines.

A Boltzmann machine is a network of symmetrically connected, neuron-like units that make stochastic decisions about whether to be on or off. Boltzmann machines have a simple learning algorithm (Hinton & Sejnowski, 1983) that allows them to discover interesting features that represent complex regularities in the training data. The learning algorithm is very slow in networks with many layers of feature detectors, but it is fast in "restricted Boltzmann machines" that have a single layer of feature detectors.

In formulas, each unit is given the opportunity of update its binary state, according to the input it receives from neighbouring units:

$$\text{net}_t = \sum_i x_i w_{ti}$$

where $\mathbf{x}$ is a *binary* vector that has a 0 in its i-th component if the i-th input is not firing, 1 otherwise. We term $\mathbf{w}_t$ the weight vector relative to unit t.

The t-th unit turns on with the following probability:

$$\mathbb{P}(x_t = 1) = \frac{1}{1 + e^{-\text{net}_t}}$$

If the units are updated sequentially in any order that does not depend on their total inputs, the network will eventually reach a Boltzmann distribution (also called its equilibrium or stationary distribution) in which the probability of a state vector $\mathbf{x}$ is determined solely by the "energy" of that state vector relative to the energies of all possible binary state vectors:

$$\mathbb{P}(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\sum u e^{-E(u)}}$$

where E denotes the energy that is defined as

$$E(\mathbf{x}) = \sum_t x_t b_t + \sum_{t<t'} v_t v_{t'} w_{tt'}$$

just like in Hopfield networks.

7. Validation and Statistical Learning Theory (SLT)

Do you recall?

As already introduced before, it is crucial to find the best trade-off between fitting capabilities (bias) and model flexibility (variance), see Figure 7.1.

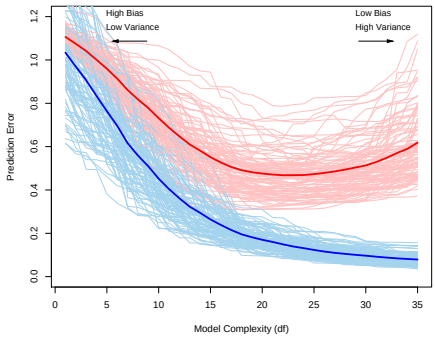


Figure 7.1: Behaviour of test error and training error as the model complexity varies. The light blue curves show the training error $\overline{err}$, while the light red curves show the conditional test error $\mathrm{Err}_{\mathcal{T}}$ for 100 training sets of size 50 each, as the model complexity is increased. The solid curves show the expected test error $\mathrm{Err}_{\mathcal{T}}$ and the expected training error $\mathrm{E}[\overline{err}]$.

In this chapter we approach both the problem of selecting the best model and evaluating its performances and the problem of providing an analytical bound to the risk.

7.1 Validation

The aim of the **validation process** is twofold: one may want to choose the best model for a learning task (**model selection**) or to evaluate the goodness of a certain model or family of models (**model assessment**). Nowadays, a lot of techniques are available for accomplishing validation purposes; in this chapter we examine: (1) *simple hold-out*, (2) *cross-validation* and (3) *double cross-validation* in both selection and assessment mode.

Definition 7.1.1 (Model selection)**.** *We define **model selection** the process of estimating the performance (generalization error) of different learning models in order to choose the best one (to generalize).*

Definition 7.1.2 (Model assessment)**.** *We define **model assessment** the process of estimating/evaluating prediction error/ risk (generalization error) of the selected model on new test data (measure of the quality/performance of the ultimately chosen model).*

Mantra

Keep separation between goals and use separate data sets.

For the sake of completeness, we cite the following involved strategies to perform validation: Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), Minimum Description Length (MDL) and Structural Risk Minimization (SRM) & VC-dimension (VC).

Definition 7.1.3 (Hold-out technique)**.** *We call* ***hold-out technique*** *the process of splitting the available data set into* training set *(TR),* validation set *(VL) and* test set *(TS).*

If the test set is used in a repeated design cycle we are actually overfitting the data we have, incurring in a overoptimistic estimation of the generalization error. If we have enough data, we can assign 50% of the samples to the training set, another 25% to the validation set and the remaining 25% to the test set.

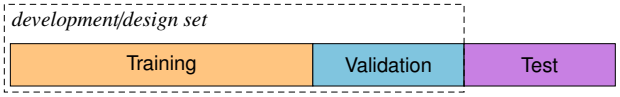


Figure 7.2: An example of splitting the initial dataset into training, validation and test set. Notice that the union of training and validation set is often called *development/design set*

Definition 7.1.4 (Subset selection bias)**.** *The error of using the entire dataset for feature/model selection that prejudices the estimation is called* ***biased estimation*** *or* ***subset selection bias****.*

Definition 7.1.5 (k-fold cross-validation)**.** *We term* ***k-fold cross-validation*** *the process of splitting the dataset D into k mutually exclusive subsets $D_1, D_2, \ldots, D_k$ and training the available models on $\overline{D}_i = D \setminus D_i$, $\forall i = 1, \ldots, k$. Once this operation has been performed, either the best model is selected or it is evaluated (tested) (see Figure 7.3).*

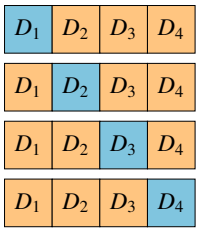


Figure 7.3: An example of the k-fold cross-validation for $k = 4$, where for each part D_i the training set is painted in orange and the validation set is painted in light blue.

This method has the following drawbacks: (1) the choice of k, (2) the computational complexity and (3) the fact that it should be performed on both test set (*external*) and validation set (*internal*).

Definition 7.1.6 (Double-cross validation)**.** *The* ***double cross-validation*** *method is used to partition the dataset into training, validation and test, performing a nested cross-validation. A working example is displayed in Figure 7.4.*

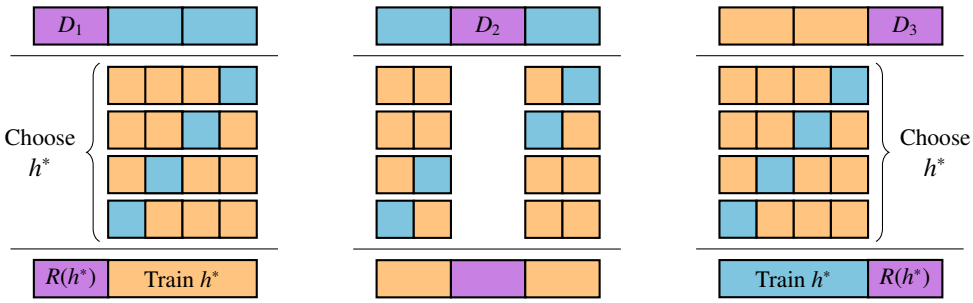


Figure 7.4: An example of the *double* k-fold cross-validation for $k_1 = 3, k_2 = 4$, where initially the whole dataset is partitioned into 3 chuncks. A 4-fold cross validation is then performed in each training set $\overline{D}_i$.

What follows is a detailed explanation of the algorithms used to perform *model selection* and *model assessment*.

Model selection

The aim of model selection is to pick the best model for solving a predictive task, given a certain dataset of labelled examples. In this scenario, we present the formalization of the *simple hold-out* algorithm and *k-fold cross validation* (Algorithm 7.1 and Algorithm 7.2 respectively).

1: select a family of functions H with hyperparameter Θ
2: divide the dataset D into two parts: TR and VL
3: **for each** value θ of the hyperparameter Θ **do**
4: select $h^*_\theta(TR) = \arg\min_{h \in H_\theta} R_{emp}(h, TR)$
5: estimate $R(h^*_\theta)$ with $R_{emp}(h^*_\theta, VL) = \frac{1}{|VL|} \sum_{d \in VL} L(h^*_\theta(TR), d)$
6: select $\theta^* = \arg\min_\theta R(h^*_\theta)$
7: **return** $h^*(D) = \arg\min_{h \in H_{\theta^*}} R_{emp}(h, D)$ ▷ retraining

Algorithm 7.1: Model selection: simple hold-out.

1: select a family of functions H with hyperparameter Θ
2: divide the dataset D into k distinct and equal parts $D_1, \ldots, D_i, \ldots, D_k$.
3: **for each** value θ of the hyperparameter Θ **do**
4: **for each** part D_i **do**
5: let $\overline{D}_i$ be the set of examples that are in D but not in D_i
6: select $h^*_\theta(\overline{D}_i) = \arg\min_{h \in H_\theta} R_{emp}(h, \overline{D}_i)$
7: estimate $R(h^*_\theta(\overline{D}_i))$ with $R_{emp}(h^*_\theta(\overline{D}_i), D_i) = \frac{1}{|D_i|} \sum_{d \in D_i} L(h^*_\theta(\overline{D}_i), d)$
8: select $\theta^* = \arg\min_\theta R(h^*_\theta(D))$
9: **return** $h^*(D) = \arg\min_{h \in H_{\theta^*}} R_{emp}(h, D)$ ▷ retraining

Algorithm 7.2: Model selection: k-fold cross validation.

Model assessment

The aim of model assessment is to provide an estimation of the performances of a certain selected model, approximating its empirical risk (R_{emp}) on a certain dataset. Algorithm 7.3, Algorithm 7.4 and Algorithm 7.5 formalize respectively *simple hold-out*, *k-fold cross validation* and *double cross validation* algorithms in the case of model assessment.

1: divide the dataset D into two parts: TR and TS
2: select $h^*(TR) = \arg\min_{h \in H} R_{emp}(h, TR)$ ▷ this optimization process could include model selection
3: estimate $R(h^*(TR))$ with $R_{emp}(h^*(TR), TS) = \frac{1}{|TS|} \sum_{d \in TS} L(h^*(TR), d)$

Algorithm 7.3: Model assessment: simple hold-out.

Model selection & assessment

In the following lines we present simple hold-out (Algorithm 7.6) and k-fold cross validation (Algorithm 7.7) algorithms used in order to both select the best model and assess its goodness.

divide the dataset D into k distinct and equal parts $D_1, \ldots, D_i, \ldots, D_k$.
for each part D_i **do**
    let $\overline{D}_i$ be the set of examples that are in D but not in D_i
    select $h^*(\overline{D}_i) = \arg\min_{h \in H} R_{emp}(h, \overline{D}_i)$ ▹ this optimization process could include model selection
    estimate $R(h^*(\overline{D}_i))$ with $R_{emp}(h^*(\overline{D}_i), D_i) = \frac{1}{|D_i|} \sum_{d \in D_i} L(h^*_\theta(\overline{D}_i), d)$
estimate $R(h^*(D))$ with $\frac{1}{K} \sum_{i=1}^{k} R(h^*(\overline{D}_i))$

Algorithm 7.4: Model assessment: k-fold cross validation.

select a family of functions H with hyperparameter Θ
divide the dataset D into k distinct and equal parts $D_1, \ldots, D_i, \ldots, D_k$
for each part D_i **do**
    let $\overline{D}_i$ be the set of examples that are in D but not in D_i
    select the best model $h^*(\overline{D}_i)$ by cross-validation
    estimate $R(h^*(\overline{D}_i))$ with $R_{emp}(h^*(\overline{D}_i), D_i) = \frac{1}{|D_i|} \sum_{d \in D_i} L(h^*(\overline{D}_i), d)$
estimate $R(h^*(D))$ with $\frac{1}{K} \sum_{k} R(h^*(\overline{D}_i))$

Algorithm 7.5: Model assessment: double cross validation.

select a family of functions H with hyperparameter Θ
divide the dataset D into three parts: TR, VL and TS
for each value θ of the hyperparameter Θ **do**
    select $h^*_\theta(TR) = \arg\min_{h \in H_\theta} R_{emp}(h, TR)$
    let $R_{emp}(h^*_\theta(TR), VL) = \frac{1}{|VL|} \sum_{d \in VL} L(h^*_\theta(TR), d)$
select $\theta^* = \arg\min_{\theta} R_{emp}(h^*_\theta(TR), VL)$
select $h^*(TR \cup VL) = \arg\min_{h \in H_{\theta^*}} R_{emp}(h, TR \cup VL)$ ▹ Train the best model
estimate $R(h^*(TR \cup VL))$ with $\frac{1}{|TS|} \sum_{d \in TS} L(h^*(TR \cup VL), d)$ ▹ Evaluate it on test set

Algorithm 7.6: Model Selection and Assessment: simple hold-out.

select a family of functions H with hyperparameter Θ
divide the dataset D into two parts: TR and TS
for each value θ of the hyperparameter Θ **do**
    estimate $R(h^*_\theta(TR))$ using cross-validation
select $\theta^* = \arg\min_{\theta} R(h^*_\theta(TR))$
retrain $h^*(TR) = \arg\min_{h \in H_\theta} R_{emp}(h, TR)$
estimate $R(h^*(TR))$ with $\frac{1}{|TS|} \sum_{d \in TS} L(h^*(TR), d)$

Algorithm 7.7: Model selection and assessment: k-fold cross validation.

7.1.1 More on hyperparameters selection

At this point we are left with the choice of those hyperparameters which are not selected directly through the usage of estimators. There are two techniques to search the *hyperparameters' space*, that follow:

- **Exhaustive Grid Search:** the whole search space is explored (it is a discrete set). Such a technique allows parallelization over the different combinations of values. It goes without saying that the computational cost is very high (Cartesian product between the sets of values for each parameter). In order to avoid combinations that show little performances, a coarse grid search is performed to obtain some good intervals and then a finer search is run.
- **Randomized Parameter Optimization**

Let us go back to the sampling for cross-validation: we want to avoid to introduce a bias due to the examples chosen. Three techniques achieve that result:

- **Stratification:** grouping member of the population into relatively homogeneous subgroups before sampling;
- **Classification:** in each partition the distribution of data of the whole dataset is preserved;
- **Folder composition:** if the data presents some order-related patterns, use such information.

It goes without saying that if the data available is few it is impossible to build statistically significant samples.

Do you recall?

During evaluation we can use one of the following error functions.

- *classification*: Accuracy, specificity, sensitivity, ROC curve;
- *regression*: after having computed the *residual*: $r_i = (y_i - o_i)$, for the target y_i
  - Mean Square Error (MSE), as $\text{avg}_i\, r_i^2$
  - Mean Absolute Error (MAE): $\text{avg}_i\, |r_i|$
  - Maximum Absolute Error (MaxAE): $\max_i |r_i|$
  - R correlation coefficient: $R = \sqrt{1 - \frac{S^2}{S_y^2}}$, where $S_y^2 = \text{avg}_i\, (y_i\, \text{avg}_i\, y^2)^2$
  - Output vs target plot
  - Various statistical tests

7.1.2 Variants

A major challenge in training neural networks is how long to train them, or equivalently the number of epochs.

A too short training brings the model to underfit the train and the test sets. Conversely, a training that continues for a lot of time leads to overfitting of the training dataset, incurring in poor performances on the test set.

A compromise is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called *early stopping*[*].

For what concerns the initialization of the weight vector $\mathbf{w}$ we need to take into account the fact that different initial choices for such vector may lead to a different model, hence we can take all of these models and perform some ensemble technique.

[*]For further readings see here

7.2 Statistical Learning Theory

In this section, we provide an analytical bound to the complexity of the hypothesis space H. The most famous measure of the complexity (capacity or flexibility) of a class of hypotheses is given by the Vapnik-Chervonenkis dimension (VC-dim). The VC dimension measures the complexity of the hypothesis space H, not by the number of distinct hypotheses $|H|$, but instead by the number of distinct instances from X that can be completely discriminated using H.

Given the instance space X, the hypothesis space H and a binary classification, then there are 2^N possible dichotomies (partitions or labelings that assign a binary target value to each of the N inputs). A particular dichotomy is "recognized" by H if there exists a hypothesis $h \in H$ that realizes (correctly classifies) the dichotomy.

Definition 7.2.1. *A set of instances X is **shattered** by hypothesis space H if and only if for every dichotomy of X there exists some hypotheses in H consistent with this dichotomy.*

Definition 7.2.2. *The **Vapnik-Chervonenkis dimension**, $VC(H)$, of an hypothesis space H defined over the instance space X is the size of the largest finite subset of X shattered by H.*

Note that for any finite hypothesis space H, $VC(H) \leq \log_2 |H|$. To see this, suppose that $VC(H) = p$. Then H will require 2^p distinct hypotheses to shatter p instances. Moreover H shatters at least one set of p points, but cannot shatter any set of $p + 1$ points. If arbitrarily large finite sets of X can be shattered by H, then $VC(H) = \infty$.
In general, the $VC(H)$ dimension of a class of linear (separator/decision) hyperplanes in a n-dimensional space is $n + 1$.

Example:

Let us consider a couple of examples:

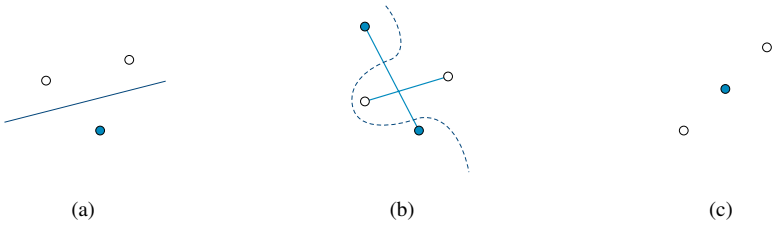
1. Let $f(x) = k$, where k is fixed, be a *constant classifier* (with no parameters). The VC-dim of such f is 0, since it cannot shatter even a single point (when it changes its class the line does not have any parameter to reassign).
2. Let $f(x) = \theta$ with $\theta \in \mathbb{R}$ be a single-parametric threshold classifier on real numbers (for a certain threshold θ the classifier f_θ returns 1 if the input number is larger than θ and 0 otherwise). The VC-dim of f is 1 because one single point that changes its class is correclty classified changing the value of the parameter θ. Moreover, it cannot shatter *any* set of two points (intuitively, it cannot classify vertically).
3. Let $f(x) = \theta_1 x + \theta_2$ be a straight line. There exists at least one set of three points that are not collinear that can be shattered. However, no set of 4 points can be shattered, so the VC-dim is 3.

Let us consider more in detail the last example: let us assume that X contains exaclty three points in $\mathbb{R}^2$ and H is a set of lines $h(\mathbf{x}) = sign(w\mathbf{x} + w_0)$ for $\mathbf{x} \in \mathbb{R}^2$. A dichotomy is a particular labeling, in $\{-1, +1\}$, of the points. Furthermore, we assume that the points in X are placed as shown in Figure 7.5(a). This specific dichotomy can be represented in H because there exists a line that correctly separates the points.

We are interested in checking if $VC - dim(H) = 3$ and this can be done following these steps:

1. Given the three poingts displayed in Figure 7.5(a), does H shatter X? Yes, since for each dichotomy there exists a line that is consistent with it. Then $VC - dim(H) \geq 3$.
2. Could $VC - dim(H)$ be 4? The answer is no, because for each possibl configuration of 4 points in a bidimensional space there is no line that is consistent with some dichotomies (see Figure 7.5(b)).

Notice that $VC - dim(H)$ is exactly 3 only if the 3 points are not colinear (see Figure 7.5(c)), hence it is strictly dependent on the instance space X.



(a) (b) (c)

Figure 7.5: The blue points are labeled with +1, while the white ones with −1.

Intuitively, we an say that the VC-dimension is related to the number of parameters, but this is not always correct: for example, we may add redundant free parameters keeping the same VC-dim or, conversely, there exist models with one parameter and infinite VC dimension (the VC-dim for the nearest neighbor algorithm is infinite).

Structural Risk Minimization

Lemma 7.2.1 (Vapnik-Chervonenkis bound)**.** *With probability* $1-\delta$ *the risk is bounded from above by a quantity which is infinitesimal and inversely proposional to the number of input data and* δ *and proportional to the Vapnik-Chervonenkis confidence. Formally,*

$$R \leq \underbrace{R_{emp} + \overbrace{\varepsilon\left(\frac{1}{l}, VC-dim, \frac{1}{\delta}\right)}^{VC\text{-}confidence}}_{guaranteed\ risk}$$

where l *accounts for the dimension of the data and the function* ε *is called* VC-confidence.

The ***VC*-confidence**, called also capacity term, can be of different types, one example for the 0/1 loss is

$$\varepsilon(VC, l, \delta) = \sqrt{\frac{VC\left(\ln\frac{2l}{VC}+1\right) - \ln\frac{\delta}{4}}{l}}$$

with probability at least $(1-\delta)$ for every $VC < l$.

For many reasonable hypothesis classes (e.g., linear approximators) the VC dimension is linear in the number of "parameters" of the hypothesis. This shows that to learn "well", we need a number of examples that is linear in the VC dimension (so linear in the number of parameters, in this case).

For a fixed number of training examples l, the training error decreases monotonically as the VC dimension is increased, whereas the confidence interval increases monotonically. Both the guaranteed risk and the true risk goes through a minimum.

These trends are illustrated in Figure 7.6. Before the minimum point is reached, the learning problem is *overdetermined* in the sense that the machine capacity (VC dimension) is too small for the amount of training detail. Beyond the minimum point, the learning problem is *underdetermined* because the machine capacity is too large for the amount of training data.
The method of *Structural Risk Minimization* (SRM) provides to find the best generalization performance by matching the machine capacity to the available amount of training data for the problem at hand.

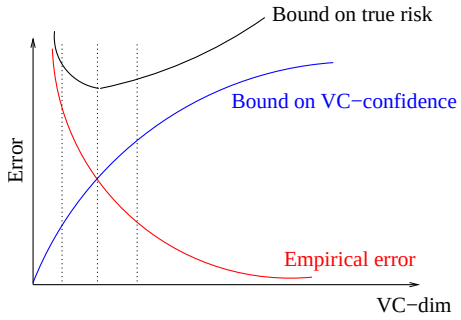


Figure 7.6: Illustration of the relationship between true risk, empirical error and the bound on VC-confidence.

The SRM uses VC dimension as a controlling parameter for minimizing the generalization bound on R. Assuming finite VC dimension we can define a nested structure of models-hypothesis spaces according to the VC dimension follows

$$H_1 \subseteq H_2 \subseteq \cdots \subseteq H_n \qquad\qquad VC(H_1) \leq VC(H_2) \leq \cdots \leq VC(H_n)$$

The use of the bound given by the SRM provides a fundamental theoretical ground for principled machine learning, independently from specific models details or learning algorithms, and the optimal choice of the model complexity (on the structure) provides the minimum of the expected risk, **the inductive principle of SRM**. Moreover this bound can be an estimation of predictive errors as a probabilistic upper bound for all the functions $h(x, \theta)$.

In order to minimize the structural risk there exist two practical approaches:

1. Choose appropriate structure/complexity, fix the model (and hence the VC confidence) and then minimize the training error.
2. Fix the training error then automatically minimize the VC confidence.

8. Bias-Variance

Do you recall?

Definition 8.0.1 (Expectation)**.** *Let Z be a* discrete *random variable with possible values z_i, with $i = 1, \dots, n$ and a probability distribution $\mathbb{P}(Z)$. We term* ***expected value*** *of Z the following*

$$\mathbb{E}_{\mathbb{P}}[Z] = \sum_{i=1}^{n} z_i \mathbb{P}(z_i)$$

Notice that if Z is continuous the sum is replaced by an integral and the distribution by a density function.

Definition 8.0.2 (Variance)**.** *Let Z be a random variable with possible values z_i, with $i = 1, \dots, n$ and a probability distribution $\mathbb{P}(Z)$. We term* ***variance*** *of Z the following*

$$Var[Z] = \mathbb{E}[(Z - \mathbb{E}[Z])^2]$$

Fact 8.0.1 (Variance lemma)**.** *The following holds:*

$$Var[Z] = \mathbb{E}[Z^2] - \mathbb{E}[Z]^2$$

Proof.

$$\begin{aligned}
\mathrm{Var}[Z] &= \mathbb{E}[(Z - \mathbb{E}[Z]^2)] \\
&= \sum_{i=1}^{n} (z_i - \mathbb{E}[Z])^2 \cdot \mathbb{P}(z_i) \\
&= \sum_{i=1}^{n} ({z_i}^2 - 2 z_i \mathbb{E}[Z] + \mathbb{E}[Z]^2) \cdot \mathbb{P}(z_i) \\
&= \sum_{i=1}^{n} {z_i}^2 \mathbb{P}(z_i) - 2\mathbb{E}[Z] \sum_{i=1}^{n} z_i \mathbb{P}(z_i) + \mathbb{E}[Z]^2 \sum_{i=1}^{n} \mathbb{P}(z_i) \\
&= \mathbb{E}[Z^2] - 2\mathbb{E}[Z] \cdot \mathbb{E}[Z] + \mathbb{E}[Z]^2 \\
&= \mathbb{E}[Z^2] - \mathbb{E}[Z]^2
\end{aligned} \tag{8.1}$$

■

Corollary 8.0.2.

$$\mathbb{E}[Z^2] = \mathbb{E}[Z]^2 + Var[Z]$$

8.1 Bias-variance tradeoff

In statistics and machine learning we talk about bias–variance tradeoff when we would like to minimize the flexibility of the model and its bias simoultaneously, since they move orthogonally one another.

- The *bias* error is an error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).

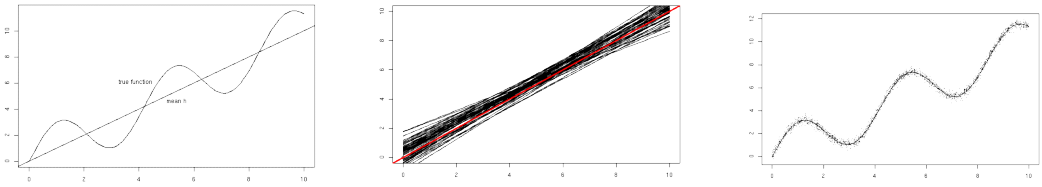
- The *variance* is an error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).

We are interested in computing the *expected value* of the prediction error on a *new* training example.

Definition 8.1.1 (Expected prediction error)**.** *Let us assume that the training data points are drawn from the "world" using a probability distribution* $\mathbb{P}$*. We term* ***expected prediction error*** *the following:*

$$\mathbb{E}_{\mathbb{P}}[(y - h(\mathbf{x}))^2]$$

where $y \in \mathbb{R}$ *is the target variable, while* $h(\mathbf{x})$ *is the predicted value.*



(a) Bias (b) Variance (c) Noise

Figure 8.1: Graphical hints about bias, variance and noise.

The expectation on prediction error can be split into 3 components:

- **Bias:** which is the disrepancy between the true function values ($f(\mathbf{x})$) and the hypothesis values $h(\mathbf{x})$
- **Variance:** which accounts for the variability of the response of model h for different realizations of the training data
- **Noise:** that is reflected into a quantity called the *irreducible error*.

For a graphical hint see Figure 8.1, while it is formalized in the following

Theorem 8.1.1 (Bias-Variance theorem)**.**

$$\mathbb{E}_{\mathbb{P}}[(y - h(\mathbf{x}))^2] = Var[h(\mathbf{x})] + Bias[h(\mathbf{x})]^2 + \sigma^2$$

Proof.

$$\begin{aligned}
\mathbb{E}_{\mathbb{P}}[(y - h(\mathbf{x}))^2] &= \mathbb{E}_{\mathbb{P}}[h(\mathbf{x})^2 - 2yh(\mathbf{x}) + y^2] \\
&= \mathbb{E}_{\mathbb{P}}[h(\mathbf{x})^2] + \mathbb{E}_{\mathbb{P}}[y^2] - 2\mathbb{E}_{\mathbb{P}}[y \cdot h(\mathbf{x})] \\
&\overset{(0)}{=} \mathbb{E}_{\mathbb{P}}[h(\mathbf{x})^2] + \mathbb{E}_{\mathbb{P}}[y^2] - 2\mathbb{E}_{\mathbb{P}}[y]\mathbb{E}_{\mathbb{P}}[h(\mathbf{x})] \\
&\overset{(1)}{=} \mathbb{E}_{P}[(h(\mathbf{x}) - \overline{h}(\mathbf{x}))^2] + \overline{h}(\mathbf{x})^2 + \mathbb{E}_{\mathbb{P}}[y^2] - 2\mathbb{E}_{\mathbb{P}}[y]\mathbb{E}_{\mathbb{P}}[h(\mathbf{x})] \\
&\overset{(2)}{=} \mathbb{E}_{P}[(h(\mathbf{x}) - \overline{h}(\mathbf{x}))^2] + \overline{h}(\mathbf{x})^2 + \mathbb{E}_{\mathbb{P}}[(y - f(\mathbf{x}))^2] + f(\mathbf{x})^2 - 2\mathbb{E}_{\mathbb{P}}[y]\mathbb{E}_{\mathbb{P}}[h(\mathbf{x})] \\
&\overset{(3)}{=} \mathbb{E}_{\mathbb{P}}[(h(\mathbf{x}) - \overline{h}(\mathbf{x})^2)] + \overline{h}(\mathbf{x})^2 - 2f(\mathbf{x})\overline{h}(\mathbf{x}) + f(\mathbf{x})^2 + \mathbb{E}_{\mathbb{P}}[(y - f(\mathbf{x}))^2] \\
&= \underbrace{\mathbb{E}_{\mathbb{P}}[(h(\mathbf{x}) - \overline{h}(\mathbf{x})^2)]}_{\text{variance}} + \underbrace{(\overline{h}(\mathbf{x}) - f(\mathbf{x}))^2}_{(\text{bias})^2} + \underbrace{\mathbb{E}_{\mathbb{P}}[(y - f(\mathbf{x}))^2]}_{\text{noise}} \\
&= \text{Var}[h(\mathbf{x})] + \text{Bias}[h(\mathbf{x})]^2 + \mathbb{E}_{\mathbb{P}}[\varepsilon^2] \\
&= \text{Var}[h(\mathbf{x})] + \text{Bias}[h(\mathbf{x})]^2 + \sigma^2
\end{aligned} \tag{8.2}$$

where we denote $\overline{h}(\mathbf{x}) = \mathbb{E}_{\mathbb{P}}[h(\mathbf{x})]$ the *mean prediction* of the hypothesis at $\mathbf{x}$, when h is trained with data drawn from $\mathbb{P}$. The "numbered equalities" are motivated as follows:

- $\overset{(0)}{=}$ follows from the independence of y and $h(\mathbf{x})$;
- $\overset{(1)}{=}$ comes from the equality $\mathbb{E}_{\mathbb{P}}[h(\mathbf{x})^2] = \mathbb{E}_{P}[(h(\mathbf{x}) - \overline{h}(\mathbf{x}))^2] + \overline{h}(\mathbf{x})^2$ (Corollary 8.0.2);

- $\overset{(2)}{=}$ comes from the following observation (motivated by the fact that $\mathbb{E}[\varepsilon] = 0$ and for a deterministic function f, $\mathbb{E}[f] = f$):

$$\mathbb{E}[y] = \mathbb{E}[f(\mathbf{x}) + \varepsilon] = \mathbb{E}[f(\mathbf{x})] = f(\mathbf{x}) \;\Rightarrow\; \mathbb{E}[y] - \mathbb{E}[f(x)] = 0 \tag{8.3}$$

  Hence, $\mathbb{E}[y^2] = \mathbb{E}[y] \cdot \mathbb{E}[y] + 0 = f(\mathbf{x}) \cdot f(\mathbf{x}) + 0 = f(\mathbf{x})^2 + \mathbb{E}[(y - f(\mathbf{x}))^2]$

- $\overset{(3)}{=}$ comes from $-2\mathbb{E}_{\mathbb{P}}[y]\mathbb{E}_{\mathbb{P}}[h(\mathbf{x})] = -2f(\mathbf{x})\bar{h}(\mathbf{x})$.

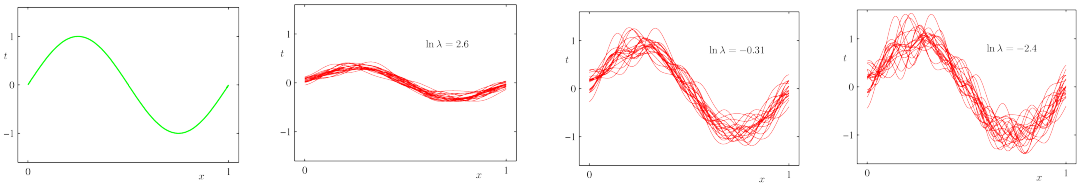
■

Example:
Let us consider the regularization term in the loss function

$$E(\mathbf{w}) = \sum_p (d_p - o(\mathbf{x}_p))^2 + \lambda \|\mathbf{w}\|^2$$

When λ is small we get complex solutions (low bias), while when λ is big, we get simple solutions (high bias).

More graphical examples in Figure 8.2.



(a) Objective function (b) High bias, low variance (c) Lower bias, higher variance (d) Low bias, high variance

Figure 8.2: How the hypothesis changes with respect to the magnitude of the regularization parameter λ.

8.2 Ensemble learning

Ensemble learning is the process by which multiple models, such as classifiers or experts, are strategically generated and combined to solve a particular computational intelligence problem. Ensemble learning is primarily used to improve the (classification, prediction, function approximation, etc.) performance of a model, or reduce the likelihood of an unfortunate selection of a poor one. Other applications of ensemble learning include assigning a confidence to the decision made by the model, selecting optimal (or near optimal) features, data fusion, incremental learning, nonstationary learning and error-correcting.

A simple way of performing ensemble is to compute the outcome as the average of the outputs returned by different models.For example, if 10 neural networks are trained using different initializations the ensemble of such models may be computed performing the average among the answers provided by each of the models. More involved thecniques in ensemble learning are: **bagging** and **boosting**.

Definition 8.2.1 (Bagging)**.** *We term **bagging** the strategy that trains n different models on* different *subsets of the training set (bootstrap resampling) and then computes the hypothesis as the average of such models.*

Definition 8.2.2 (Boosting)**.** ***Boosting** works creating an ensemble of n different models trained by resampling the misclassified data at each step. Such models are then combined by majority voting.*

The aim of boosting is to improve the accuracy of each model incrementally. In particular, resampling is strategically geared to provide the most informative training data for each consecutive classifier. In essence, each iteration of boosting creates k weak classifiers: the first model h_1 is trained with a random subset of the available training data. The training data subset for the second model h_2 is chosen as the most informative subset, given h_1. Specifically, h_2 is trained on a training set which is made for a half by those patterns that were misclassified by h_1. The third model h_3 is trained with instances on which h_1 and h_2 disagree. This precedure goes on k times and then the models are combined through a majority vote.

9. Support Vector Machines (SVM)

Terminology

In this chapter we will talk about $\mathbf{w} \in \mathbb{R}^n$ instead of $\mathbb{R}^{n+1}$ and denote w_0 as b.

9.1 Introduction

In this chapter we are going to discuss the problem of separating points in the space. The points will be separated by a **discriminant function** $g(\mathbf{x}) = \mathbf{w}_o{}^T\mathbf{x} + b_o$, referring so to the hypothesis as $h(\mathbf{x}) = sign(g(\mathbf{x}))$. Moreover we are in the context of *unsupervised learning* for classification.

Definition 9.1.1 (Separating hyperplane)**.** *Given the training set* $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$ *we term* ***separating hyperplane*** *the hyperplane of equation* $\mathbf{w}^T\mathbf{x} + b = 0$*, which separates the training examples as follows:*

$$\begin{cases} \mathbf{w}^T\mathbf{x}_p + b \geq 0 \text{ if } d_p = +1 \\ \mathbf{w}^T\mathbf{x}_p + b < 0 \text{ if } d_p = -1 \end{cases}$$

Notice that the vector $\mathbf{w}$ is *orthogonal* to the separating hyperplane. Every hyperplane is characterized by its direction determined by vector $\mathbf{w}$ and its exact position determined by scalar b. Our aim is to find the best *discriminant function* $g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$, such that $h(\mathbf{x}) = sign(g(\mathbf{x}))$.

It is important to stress that we want *the best* hyperplane, given the fact that there are many options and there is a preferred answer, namely the hyperplane which is "more distant from any point in the space".

Definition 9.1.2 (Separation margin)**.** *Given a hyperplane* $\mathbf{w}^T\mathbf{x} + b = 0$ *we term* ***separation margin*** $\rho \in \mathbb{R}$ *twice the distance between the hyperplane and the closest point (either below or above).*

Definition 9.1.3 (Support vector)**.** *A* ***support vector*** *is denoted as* $\mathbf{x}_s$ *and it is a vector which lies exaclty on the boundaries of the margin. Formally, a support vector satisfies exactly the equation*

$$d_s \cdot (\mathbf{w}^T\mathbf{x}_s + b) = 1 \tag{9.1}$$

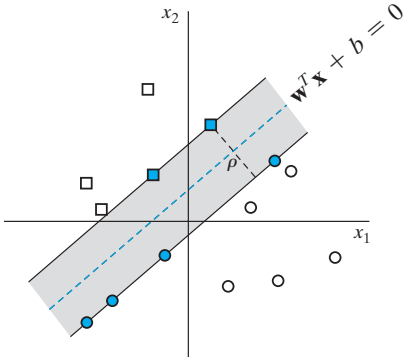


Figure 9.1: Illustration of a hyperplane with margin ρ. The support vectors are the points (squares and circles) filled in blue, which are the closest to the hyperplane.

We already stressed that separating hyperplanes are not equivalent one with the other, in particular their slope impacts the separation margin, as shown in Figure 9.2.

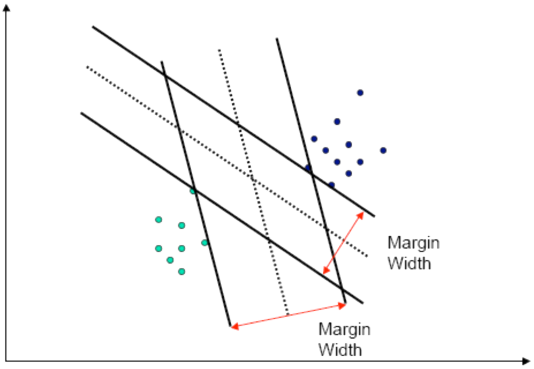


Figure 9.2: Two different margin for the same classification task.

Which hyperplane in Figure 9.2 would be a better choice? No doubt the steepest one is better, because it leaves more ”room” on either side, so that data in both classes can move a bit more freely, with less risk of causing an error. Such a hyperplane can be trusted more, when it faced with unknown test data. This touches the important issue of classifier design: the generalization performance of the classifier. This refers to the capability of the classifier, designed using the training set, to operate satisfactorily with data outside this set.

Definition 9.1.4 (Optimal hyperplane)**.** *We refer to the **optimal hyperplane** as the hyperplane which maximizes the margin ρ and it is denoted as*

$$\mathbf{w}_o{}^T\mathbf{x} + b_o = 0$$

Let us introduce the so-called **canonical representation** of the hyperplane: we are going to scale $\mathbf{w}$ and b such that the hyperplane keeps being the same (notice that the hyperplane equation is set equal to 0, hence we are allowed to multiply for a certain quantity $q \neq 0$ both sides), but we achieve the following property: $\forall \mathbf{x}_s$ support vector $|g(\mathbf{x}_s)| = \left|\mathbf{w}^T\mathbf{x}_s + b\right| = 1$ (-1 for the points below the hyperplane and $+1$ for the points above it).

Since the points which are closest to the boundary are not closer than 1, we get that $d_p \cdot (\mathbf{w}^T\mathbf{x}_p + b) \geq 1$ $\forall p = 1, \ldots, l$, which is the compact form for:

$$\begin{cases} \mathbf{w}^T\mathbf{x}_p + b \geq 1 \text{ if } d_p = +1 \\ \mathbf{w}^T\mathbf{x}_p + b < 1 \text{ if } d_p = -1 \end{cases}$$

Observation 9.1.1. *The support vectors have such a name, because the optimal solution to this classification problem only depends on them.*

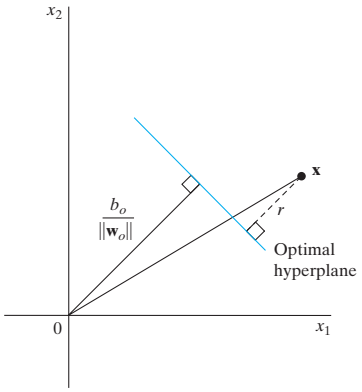


Figure 9.3: Geometric interpretation of algebraic distances of points to the optimal hyperplane for a two-dimensional case.

Fact 9.1.1. *The discriminant function $g(\mathbf{x})$ gives an algebraic measure ($r \in \mathbb{R}$) of the distance from $\mathbf{x}$ to the optimal hyperplane. Such value is computed as*

$$r = \frac{g(\mathbf{x})}{\|\mathbf{w}_o\|}$$

Proof. Denoting with r that algebraic distance between $\mathbf{x}$ and the otpimal hyperplane, we get that $\mathbf{x}$ can be rewritten as

$$\mathbf{x} = \hat{\mathbf{x}} + r \cdot \frac{\mathbf{w}_o}{\|\mathbf{w}_o\|}$$

r is positive if $\mathbf{x}$ is on the positive side of the optimal hyperplane and negative if $\mathbf{x}$ is on the negative side. Substituting $\mathbf{x}$ into the equation of the optimal hyperplane, we get

$$\begin{aligned} g(\mathbf{x}) &= g\left(\hat{\mathbf{x}} + r \cdot \frac{\mathbf{w}_o}{\|\mathbf{w}_o\|}\right) = \\ &= \mathbf{w}_o{}^T \hat{\mathbf{x}} + b_o + \mathbf{w}_o{}^T r \cdot \frac{\mathbf{w}_o}{\|\mathbf{w}_o\|} = \\ &= g(\hat{\mathbf{x}}) + r\mathbf{w}_o{}^T \frac{\mathbf{w}_o}{\|\mathbf{w_o}\|} = \\ &\overset{(1)}{=} r \cdot \frac{\|\mathbf{w}_o\|^2}{\|\mathbf{w}_o\|} = r \cdot \|\mathbf{w}_o\| \end{aligned} \tag{9.2}$$

where $\overset{(1)}{=}$ is due to the fact that $\hat{\mathbf{x}}$ lies on the hyperplane, so $g(\hat{\mathbf{x}}) = 0$. Equivalently we can write

$$r = \frac{g(\mathbf{x})}{\|\mathbf{w}_o\|}$$

■

Fact 9.1.2. *The margin size ρ is equal to $\dfrac{2}{\|\mathbf{w}_o\|}$.*

Proof. Consider a support vector $\mathbf{x}_s$ for which $d_s = +1$, we get $g(\mathbf{x}_s) = 1$ and

$$r = \frac{g(\mathbf{x}_s)}{\|\mathbf{w}_o\|} = \frac{1}{\|\mathbf{w}_o\|} \qquad \Rightarrow \qquad \rho = 2r = \frac{2}{\|\mathbf{w}_o\|}$$

■

Observation 9.1.2. *The optimum hyperplane maximizes ρ, while it minimizes $\|\mathbf{w}\|$.*

9.2 Support Vector Machine for binary classification

At this point we are interested in finding the optimum values $\mathbf{w}$ and b in order to maximize the margin and this is a **quadratic optimization problem**.

9.2.1 Hard margin

Definition 9.2.1 (Hard margin optimization problem — primal form)**.** *Given the training samples $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$ we define the following **optimization problem**: find the maximum margin hyperplane, given that such hyperplane should have 0 as training error (aka classifies correctly the points). Formally, the optimum values for $\mathbf{w}$ and b which minimize*

$$\Psi(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

while they also satisfy the following constraints

$$d_p \cdot (\mathbf{w}^T\mathbf{x}_p + b) \geq 1, \ \forall p = 1, \dots l$$

The optimization problem presented above can be solved using the *Lagrangian multipliers*.

Fact 9.2.1. *The optimal hyperplane is expressed as*

$$\mathbf{w}_o{}^T\mathbf{x} + b_o = 0 \iff \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p{}^T\mathbf{x} + b_o = 0$$

Proof. The optimal $\mathbf{w}_o$ has the following value:

$$\mathbf{w}_o = \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p$$

For further readings on the theoretical explanation for such result see Appendix 9.A ■

Corollary 9.2.2. *Recall the Equation (9.1). For each positive support vector $\mathbf{x}_s$ ($d_s = +1$) we get that the **optimal bias** has the following value*

$$b_o = 1 - \mathbf{w}_o^T\mathbf{x}_s = 1 - \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p{}^T\mathbf{x}_s$$

Conversely, for each negative support vector $\mathbf{x}_s$ ($d_s = -1$) we have

$$b_o = -1 - \mathbf{w}_o^T\mathbf{x}_s = -1 - \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p{}^T\mathbf{x}_s$$

Fact 9.2.3. *In a quadratic optimization problem, naming α_p the dual variables, the following holds:*

1. *if $\alpha_p > 0$ then $d_p(\mathbf{w}^T\mathbf{x}_p + b) = 1$ and $\mathbf{x}_p$ is a support vector;*
2. *if $\mathbf{x}_p$ is not a support then $\alpha_p = 0$.*

Proof. The Karush-Khun-Tucker conditions on such a problem are formalized as

$$\alpha_p(d_p(\mathbf{w}^T\mathbf{x}_p + b) - 1) = 0 \; \forall p = 1, \ldots, l$$

in the saddle point. The thesis follows from trivial reasoning on such equation. ■

Notice that, provided that the $\{\alpha_p\}$ are obtained solving the quadratic dual (for further readings see Appendix 9.B), we have the tools for finding the optimal hyperplane. We don't need to explicitly know $\mathbf{w}_o$, we only need to know Lagrangian multipliers, then we compute the optimal bias b_o.
Notice that here we are *fixing to* 0 the empirical risk on the training set, as anticipated in the Chapter 7:

$$R[h] \leq R_{\text{emp}}[h] + \varepsilon(VC, l, \delta)$$

Moreover, the following holds

Theorem 9.2.4 (Vapnik theorem)**.** *Let $\overline{d}$ be the diameter of the smallest ball around the data points $\{\mathbf{x}_1, \ldots, \mathbf{x}_l\}$. For the class of separating hyperplanes described by the equation $\mathbf{w}^T\mathbf{x} + b = 0$ the upper-bound for the VC-dimension is*

$$VC \leq \min\left(\left\lfloor \frac{\overline{d}}{\rho^2} \right\rfloor, m_o\right) + 1$$

Notice that at this point we have available an algorithm that provides a unique solution, obtained using an automatized approach, which does not require any hyper-parameter tuning.

9.2.2 Soft margin

We have already stated that the learning algorithm bases its output on the support vectors only, hence it has the good quality of not depending directly on the points far from the decision boundary.
We are interested in admitting some uncertainty: *what if some data points lie inside the margin* (as displayed in Figure 9.4)?

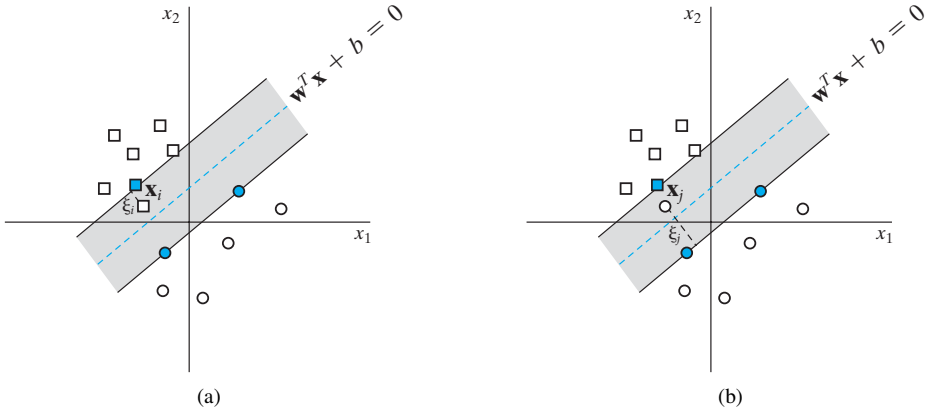


Figure 9.4: Soft margin hyperplane. In (a) a data point $\mathbf{x}_p$ falls inside the region of separation, but on the correct side of the decision surface. In (b) a data point $\mathbf{x}_j$ falls on the wrong side of the decision surface.

To overcome this issue, we introduce the **soft margin conditions**, obtained through *slack variables* $\xi_p \geq 0 \in \mathbb{R},\ \forall p = 1, \ldots, l$ such that

$$d_p \cdot (\mathbf{w}^T \mathbf{x}_p + b) \geq 1 - \xi_p \ \forall p = 1, \ldots l \tag{9.3}$$

In this new environment, we get that a *support vector* satisfies the equation exactly ($d_s \cdot (\mathbf{w}i^T \mathbf{x}_s + b) = 1 - \xi_p$), see Figure 9.4.

Observation 9.2.1. *Note that the Vapnik's theorem does not hold anymore, since a hard margin does not admit data points inside the margin.*

Definition 9.2.2 (Soft margin optimization problem — primal form)**.** *Given the training samples* $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$ *we define the following* ***optimization problem****: find the maximum margin hyperplane, given that such hyperplane should have a training error as small as possible. Formally, the optimum values for* $\mathbf{w}$ *and* b *which minimize*

$$\Psi(\mathbf{w}, \xi) = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C \cdot \sum_{p=1}^{l} \xi_p \tag{9.4}$$

while they also satisfy the following constraints $\forall p = 1, \ldots l$

- $d_p(\mathbf{w}^T \mathbf{x}_p + b) \geq 1 - \xi_p$
- $\xi_p \geq 0$

Observation 9.2.2. *The real constant* C *is a* ***regularization hyperparameter*** *and needs to be tuned in order to find a trade-off between empirical risk minimization and model complexity (estimated through the capacity term, also called VC-confidence). Moreover, a too low value for* C *leads to underfitting, while a too high value for it may bring to an overfitting problem.*

Exercise 9.2.1. *Take the primal slackened optimization problem (Equation (9.3)) and remove the term* $C \cdot \sum_{p=1}^{l} \xi_p$ *from Equation (9.4). Do you think that the problem collapses to the strict margin one? Motivate your answer.*

Solution: It does not, indeed the two constraints on the slackened variables are still there and this does not give any guarantee to how big the ξ_p will be, actually incurring in underfitting.

Fact 9.2.5. *In a quadratic optimization problem, naming* α_p *the dual variables, the following holds:*

1. *if* $0 < \alpha_p < C$ *then* $\xi_p = 0$ *hence* $\mathbf{x}_p$ *is a support vector;*
2. *if* $\alpha_p = C$ *then* $\xi_p \geq 0$ *then* $\mathbf{x}_p$ *is* inside *the margin;*
3. *if* $\alpha_p = 0$ *we can say nothing.*

Proof. The Karush-Khun-Tucker conditions on the slackened problem are formalized as

$$\alpha_p(d_p(\mathbf{w}^T\mathbf{x}_p + b) - \xi_p - 1) = 0 \; \forall p = 1, \dots, l$$
$$\mu_p\xi_p = 0 \; \forall p = 1, \dots, l$$

where μ_p are the Lagrange multipliers introduced to enforce the non-negativity of the slack variables of the dual problem. The thesis follows from trivial reasoning on such equation. ∎

Notice that, provided that the $\{\alpha_p\}$ are obtained solving the quadratic dual (for further readings see Appendix 9.B), we have the tools for finding the optimal hyperplane.
It goes without saying that the optimal hyperplane and the optimal bias are computed as above (Proposition 9.2.1 and Corollary 9.2.2) with $0 < \alpha_p < C$.
As we did before, we only need to plug such values into the formula for function g.

Observation 9.2.3. *In g there is the standard scalar product between $\mathbf{x}_p$ and itself ($\mathbf{x}_p{}^T\mathbf{x}_p$), but we do not need to use the standard scalar product, while we could use a feature space transformation to allow the classification of* non linearly separable *problems, just like what we did for the* linear basis expansion.

9.2.3 As kernel machine

Let us take a non linear function $\Phi : \mathbb{R}^n \to \mathbb{R}^{\bar{n}}$, the problem changes as follows:

- $D = \{(\Phi(\mathbf{x}_p), d_p)\}_{p=1}^l$
- $g(\mathbf{x}) = \mathbf{w}^T\Phi(\mathbf{x}) + b$
- The decision boundary is expressed as $\mathbf{w}^T\Phi(\mathbf{x}) + b = 0$

Let us perform the operation of "incorporating" the real number b in $\mathbf{w}$ as first component, so we get:

- $\mathbf{w} \in \mathbb{R}^{n+1}$
- $\Phi(\mathbf{x}) = (\Phi_0(\mathbf{x}) = 1, \Phi_1(\mathbf{x}), \dots, \Phi_{\bar{n}}(\mathbf{x}))^T$
- $\mathbf{w}^T\Phi(\mathbf{x}) = 0$ is called the **new** decision boundary.

In this new scenario, the optimal weight vector is computed as $\mathbf{w}_o = \sum_{p=1}^l \alpha_p d_p \Phi(\mathbf{x}_p)$, so the new hyperplane equation follows:

$$\mathbf{w}_o{}^T\Phi(\mathbf{x}) = \sum_{p=1}^{l} \alpha_p d_p \Phi^T(\mathbf{x}_p)\Phi(\mathbf{x}) = 0$$

where the dot product is now in the feature space.
At this point we are left with the problem of computing $\Phi(\mathbf{x})$. We observe that this may be unfeasible, thus we present a method that allows to compute **directly** the dot product.

Definition 9.2.3 (Kernel function)**.** *We term $k : \mathbb{R}^{m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}$ such that $\forall \mathbf{a}, \mathbf{b} \in \mathbb{R}^{m_0}$ $k(\mathbf{a}, \mathbf{b}) = \Phi^T(\mathbf{a})\Phi(\mathbf{b})$ an **inner product kernel function**.*

The idea underpinning the use of a kernel function is to focus on the representation and processing of pairwise comparisons rather than on the representation and processing of a single object.

Property 9.2.6. *The kernel function is symmetric. Formally,*

$$k(\mathbf{a}, \mathbf{b}) = k(\mathbf{b}, \mathbf{a})$$

Definition 9.2.4 (Kernel matrix)**.** *We term **kernel matrix** the $l \times l$ symmetric matrix that contains the values of the inner product kernel function on all the possible couples of training examples. Formally,*

$$K = \begin{pmatrix} k(\mathbf{x}_1, \mathbf{x}_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}_l) \\ k(\mathbf{x}_2, \mathbf{x}_1) & \cdots & k(\mathbf{x}_2, \mathbf{x}_l) \\ \vdots & & \vdots \\ k(\mathbf{x}_l, \mathbf{x}_1) & \cdots & k(\mathbf{x}_l, \mathbf{x}_l) \end{pmatrix} = \{k(\mathbf{x}_p, \mathbf{x}_j)\}_{p,j=1}^l$$

Example:
Let us take $\mathbf{x} = (x_1, x_2)^T \in \mathbb{R}^2$ and let $\Phi : \mathbb{R}^2 \to \mathbb{R}^3$ such that

$$\Phi(\mathbf{x}) = \Phi\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} x_1^2 \\ \sqrt{2}x_1x_2 \\ x_2^2 \end{pmatrix}$$

For each $\mathbf{x}, \mathbf{y} \in R^2$, we compute $\Phi^T(\mathbf{x})\Phi(\mathbf{y})$ as follows

$$\begin{aligned}(x_1^2,\ \sqrt{2}x_1x_2, x_2^2) \cdot \begin{pmatrix} y_1^2 \\ \sqrt{2}y_1y_2 \\ y_2^2 \end{pmatrix} &= x_1^2y_1^2 + 2x_1x_2y_1y_2 + x_2^2y_2^2 \\ &= (x_1y_1 + x_2y_2)^2 \\ &= \left((x_1, x_2) \cdot \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\right)^2 = (\mathbf{x}^T\mathbf{y})^2 = (k(\mathbf{x}, \mathbf{y}))^2\end{aligned} \tag{9.5}$$

Theorem 9.2.7 (Mercer's theorem)**.** *Let $k : \mathbb{R}^{m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}$ a continuous symmetric non-negative definite kernel. Then k can compute the inner product in a feature space.*

Property 9.2.8. *The set of kernel functions is closed for addition, scalar product and inter-kernel function product. Formally, $\forall k_1, k_2 : \mathbb{R}^{m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}$ kernel functions, the following are kernel function:*

- $k_p(\mathbf{x}, \mathbf{y}) + k_2(\mathbf{x}, \mathbf{y})$
- $\alpha k_1(\mathbf{x}, \mathbf{y}),\ \forall \alpha \in \mathbb{R}_+$
- $k_1(\mathbf{x}, \mathbf{y}) \cdot k_2(\mathbf{x}, \mathbf{y})$

For further details on how the primal and dual formulation of the SVM change when the kernel function is introduced see Appendix 9.C.

Notice that we can design a neural network that implements a support vector machine, as shown in Figure 9.5.

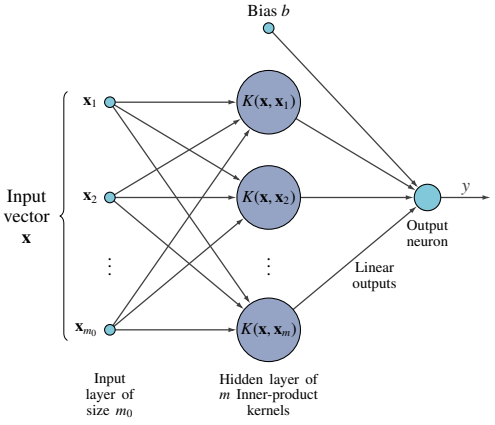


Figure 9.5: Architecture of Support Vector Machine, where the number of hidden units is exactly the number of support vectors.

Some functions that can be used as kernels are:

- **Polynomial learning machine:** $k(\mathbf{x}, \mathbf{x}_p) = (\mathbf{x}^T\mathbf{x}_p + 1)^p$, where p is a user specified parameter
- **Radial basis function net:** $k(\mathbf{x}, \mathbf{x}_p) = e^{-\frac{1}{2\sigma^2} \cdot ||\mathbf{x}-\mathbf{x}_p||^2}$, where σ^2 is a user specified parameter
- **Two-layer perceptron**: $k(\mathbf{x}, \mathbf{x}_p) = \tanh(\beta_0\mathbf{x}^T\mathbf{x}_p + \beta_1)$, where $\beta_0 (> 0)$ and $\beta_1 (< 0)$ are user specified parameters.

Notice that we need to choose a proper kernel function for the specific domain and it can be extended to generic input objects. It may be good to design new kernel functions based on edit-distance for bioinformatics of language duffix-trees for string representation.

9.3 Support Vector Machine for non-linear regression

Until now, we dealt with linear classification problems, but SVM may be used to find a good approximation to a non linear regression problem as well.

Let us assume we are interested in "guessing" a function f, which is not linear, given a set of points $\mathbf{x}$ in the vector space $\mathbb{R}^n$.

First, we need to take into account the issue of noise: it is crucial to design a soft margin loss function.

Definition 9.3.1 (ε-insensitive loss function)**.** *We term* ***ε-insensitive loss function*** L_ε *a loss function which does not take into account the error on those points which are closer than a certain constant ε to the hypothesis value. Formally, given ah hypothesis $h(\mathbf{x}) = \mathbf{w}^T\Phi(\mathbf{x}_p)$ and the i-th training example d_p*

$$L_\varepsilon(d_p, h(\mathbf{x}_p)) = \begin{cases} \left|d_p - h(\mathbf{x}_p)\right| - \varepsilon & \text{if } \left|d_p - (\mathbf{x}_p)\right| \geq \varepsilon \\ 0 & \text{otherwise} \end{cases}$$

A pictorial representation of the semantics of such loss function can be found in Figure 9.6.

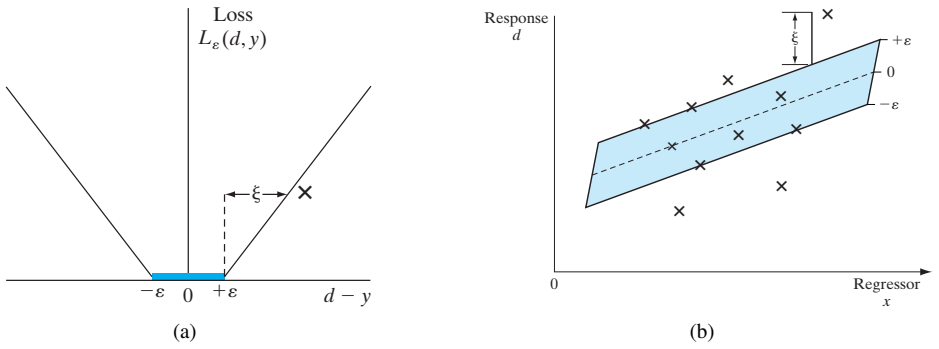


Figure 9.6: (a) Plot of the ε-intensive loss function for $y = h(\mathbf{x})$, where the cost function ignores any training data close to the model prediction. (b) Illustrating an ε-intensive tube of radius ε.

Let us introduce the *non-negative* slack variables ξ'_p and ξ_p for each $i = 1, \ldots, l$

$$\xi'_p - \varepsilon \leq d_p \mathbf{w}^T\Phi(\mathbf{x}_p) \leq \varepsilon + \xi_p \; \forall p = 1, \ldots, l$$

that leads to the following constraints $\forall p = 1, \ldots, l$

- $d_p - \mathbf{w}^T\Phi(\mathbf{x}_p) \leq \varepsilon + \xi_p$
- $\mathbf{w}^T\Phi(\mathbf{x}_p) - d_p \leq \varepsilon + \xi'_p$
- $\xi_p, \xi'_p \geq 0$

Definition 9.3.2 (Primal problem)**.** *Given the training samples $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^l$ we define the following* ***optimization problem****: find the maximum margin hyperplane, given that such hyperplane should have* 0 *as training error (aka classifies correctly the points). Formally, the optimum values for $\mathbf{w} \in \mathbb{R}^{n+1}$ which minimizes*

$$\Psi(\mathbf{w}, \xi, \xi') = \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{p=1}^{l}(\xi_p + \xi'_p)$$

under the constraints $\forall p = 1, \ldots, l$

- $d_p(\mathbf{w}^T\Phi(\mathbf{x}_p)) \leq \varepsilon + \xi_p$
- $\mathbf{w}^T\Phi(\mathbf{x}_p) - d_p \leq \varepsilon + \xi'_p$

- $\xi_p, \xi'_p \geq 0$

Definition 9.3.3 (Dual problem)**.** *Given the training samples* $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$*, we call* ***dual optimization problem*** *the task of finding the optimum values of the Lagrangian multipliers* $\{\alpha_p\}_{p=1}^{l}$ *and* $\{\alpha'_p\}_{p=1}^{l}$*, which maximize:*

$$Q(\alpha, \alpha') = \sum_{p=1}^{l} d_p(\alpha_p - \alpha'_p) - \varepsilon \sum_{p=1}^{l}(\alpha_p + \alpha'_p) - \frac{1}{2}\sum_{p=1}^{l}\sum_{j=1}^{l}(\alpha_p - \alpha'_p)(\alpha_j - \alpha'_j)k(\mathbf{x}_p, \mathbf{x}_j)$$

satisfying the constraints

- $0 \leq \alpha_p, \alpha'_p \leq C \; \forall p = 1, \ldots, l$
- $\sum_{p=1}^{l}(\alpha_p - \alpha'_p) = 0$

By solving the dual problem, we get the optimal values of the Lagrangina multipliers $\{\alpha_p\}_{p=1}^{l}$ and $\{\alpha'_p\}_{p=1}^{l}$, so that we can compute

$$\mathbf{w} = \sum_{p=1}^{l} \underbrace{(\alpha_p - \alpha'_p)}_{\gamma_p} \Phi(\mathbf{x}_p)$$

In this case support vectors correspond to non-zero values of γ_p. At this point we are ready to write explicitly

$$h(\mathbf{x}) = \mathbf{w}^T\Phi(\mathbf{x}) = \sum_{p=1}^{l}\gamma_p\Phi^T(\mathbf{x}_p)\Phi(\mathbf{x}) = \sum_{p=1}^{l}\gamma_p k(\mathbf{x}_p, \mathbf{x})$$

Do you recall?

The procedure for learning the model h is the following:

1. Select values for the user specied parameters C and ε
2. Choose an inner product kernel function k
3. Compute the kernel matrix K
4. Solve the optimization problem in the dual form and get the optimal values of the Lagrangian multipliers (we get the values of$\{\gamma_p\}_{p-1}^{l}$)
5. Sompute the optimal value for the bias
6. Obtain the estimate function as a linear combiner of dot products in a feature space

Notice that it is important to select hyperparamters extimating the VC-dim.

Appendix 9.A Math of the SVM

The Lagrangian function for our quadratic optimization problem is the following

$$J(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^T\mathbf{w} - \sum_{p=1}^{l}\alpha_p \cdot \Big(d_p \cdot (\mathbf{w}^T\mathbf{x}_p + b) - 1\Big)$$

where $\alpha_p \; \forall p = 1, \ldots, l$ are called **Lagrangian multipliers**. Each term in the sum corresponds to a constraint of the primal problem and J must be

- minimized with respect to $\mathbf{w}$, $\frac{\partial J}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_{p=1}^{l}\alpha_p d_p \mathbf{x}_p$
- minimized with respect to b, $\frac{\partial J}{\partial b} = 0 \Rightarrow \sum_{p=1}^{l}\alpha_p d_p = 0$

The solution corresponds to a saddle point of J.
By solving the Lagrangian dual, we get $\mathbf{w}_o = \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p$, hence the optimal hyperplane is expressed as

$$\mathbf{w}_o{}^T + b_o = 0 \iff \sum_{p=1}^{l} \alpha_{op} d_p \mathbf{x}_p{}^T \mathbf{x} + b_o = 0$$

Appendix 9.B Dual form

Definition 9.B.1 (Optimization problem — dual form)**.** *Given the training samples $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$, we call **dual optimization problem** the task of finding the optimum values of the Lagrangian multipliers $\{\alpha_p\}_{p=1}^{l}$, which maximize:*

$$Q(\alpha) = \sum_{p=1}^{l} \alpha_p - \frac{1}{2} \sum_{p=1}^{l} \sum_{j=1}^{l} \alpha_p \alpha_j d_p d_j \mathbf{x}_p{}^T \mathbf{x}_j$$

satisfying the constraints

- $\alpha_p \geq 0 \; \forall p = 1, \ldots, l$
- $\sum_{p=1}^{l} \alpha_p d_p = 0$

Appendix 9.C SVM for kernel transformation

Definition 9.C.1 (Primal problem)**.** *Given the training samples $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$ we define the following **optimization problem**: find the maximum margin hyperplane, given that such hyperplane should have 0 as training error (aka classifies correctly the points). Formally, the optimum values for $\mathbf{w} \in \mathbb{R}^{n+1}$ which minimizes*

$$\Psi(\mathbf{w}, \xi) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{p=1}^{l} \xi_p$$

under the constraints

- $d_p(\mathbf{w}^T \Phi(\mathbf{x}_p)) \geq 1 - \xi_p \; \forall p = 1, \ldots, l$
- $\xi_p \geq 0 \forall p = 1, \ldots, l$

Definition 9.C.2 (Dual problem)**.** *Given the training samples $D = \{(\mathbf{x}_p, d_p)\}_{p=1}^{l}$, we call **dual optimization problem** the task of finding the optimum values of the Lagrangian multipliers $\{\alpha_p\}_{p=1}^{l}$, which maximize:*

$$Q(\alpha) = \sum_{p=1}^{l} \alpha_p - \frac{1}{2} \sum_{p=1}^{l} \sum_{j=1}^{l} \alpha_p \alpha_j d_p d_j k(\mathbf{x}_p, \mathbf{x}_j)$$

satisfying the constraints

- $0 \leq \alpha_p \leq C \; \forall p = 1, \ldots, l$
- $\sum_{p=1}^{l} \alpha_p d_p = 0$

10. Structured Domain Learning

Mantra

"The ability to treat the proper inherent nature of the input data is the key feature for a successful application of the machine learning methodologies"

10.1 Overview and motivation

There are three different data types: (1) *static* data includes attribute-value or real vectors structures, (2) *sequential* data are characterized by a serially ordered entries and finally (3) *structural* data have relations among domain variables. In this chapert we will concentrate on the last kind of data studying the Structured Domain Learning (SDL).

The data usually used in neural network are expressed in a *flat* form as vectors. There also exist *structured* versions of the data that can be sequences, trees, graphs or multi-relational.

There are many different examples that shows how important and useful are the structured data, on the point that there exists a relationship between them. The first example is the recognition of a logo, where each component of the image can be associated as a node of a tree and, for the nature of this structure, some nodes may be grouped under a single one. This schema is showed in Figure 10.1. Other examples can be the trees build for the language parsing, terms in 1^{st} order logic or social network.

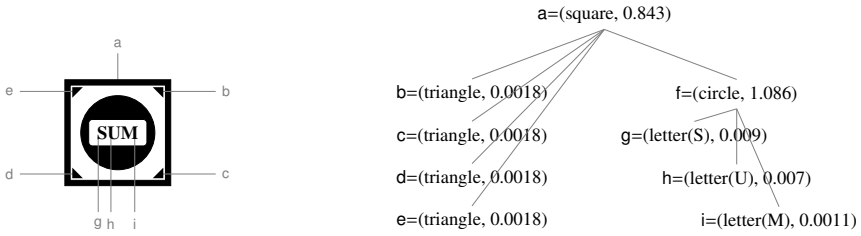


Figure 10.1: An example of data relationship in structured data.

Moreover the tree structure, the data can be represented in graph. An example of this kind of representation is the biological network, in which a node corresponds to a protein and a link between two of them is an interaction or a similarity value. A graph notation is given in Figure 10.2 where for a graph g we can define some nodes (or vertex) v labeled with some values l_v and connected with some edge (or link) that may be from a cycle inside g.

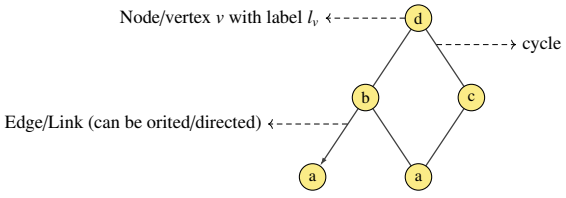


Figure 10.2: General schema of a graph.

The problem of graph representation is that there has been no systematic way (of general validity for any task) to extract features or metrics relations between examples for structured data. Features-based representations are incomplete (or strongly task-dependent, e.g. topological indexes), while adjacent/incident matrix representations (or other fixed-sizes representations) have some issues: topological order, alignment among different graphs and are incomplete.

Taking the example of the biological notwork, the goal is to learn a mapping between a structured information domain (SD) and a discrete or continuous space (transduction). Given a structure of a molecule apply a transduction to learn the property value (regression) or if it is toxic or not (classification). Molecules are not vectors and they can be more naturally represented by varying size structures. Is possible to start with this problem: classify variable size. For instance, classify different graphs starting from a training set of known couples $(graph_i, target_i)$. I this way we can learn hypothesis applying a transduction and mapping $T(graph)$.

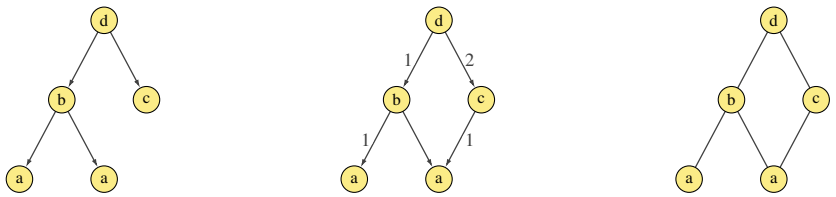
Summing the notion of SD and machine learning we otain the Structured Domain Learning (SDL) subject, that instead of moving data to models (e.g. graphs into vectors or trees into sequences, with alignment problems, loose of information, etc.) we move models to data. Table 10.1 summarize the learning algorithms for a specific model and data type.

Data type	**Model**		
	Symbolic	Connectionist	Probabilistic
Static	Rule induction, Decision Tree	Neural Network, SVM	Mixture models, Naïve Bayes
Sequential	Learning finite state automata	RNN	Hidden Markov Models
Structural	inductive logic programming	RNN (kernel for SD)	Recursive Markov models

Table 10.1

There are different classes of structured data:

- Single labeled vertex, where all the information are represent by a single node;
- Sequence, composed by some single vertex and ordered under a criteria;
- K-ary tree, a rooted tree bounded with *out-degree(k)* for each node (fig. 10.3(a));
- DPAG, labeled *direct positional acyclic graph* with supersource and bounded with *in-degree(k)* and *out-degree(k)* (fig. 10.3(b));
- Undirected graph (fig. 10.3(c)).



(a) K-ary tree (b) DPAG (c) Undirected graph

Figure 10.3: Structured data classes.

A more general transduction is represent in Figure 10.4, where the input graph g is transformed in a possible internal representation or encoding $x(g)$ and then is manipulated to obtain the output $y(g)$ that can be a structure, in case of input-output isomorphic, or a scalar/element, in case of a regression/classification.

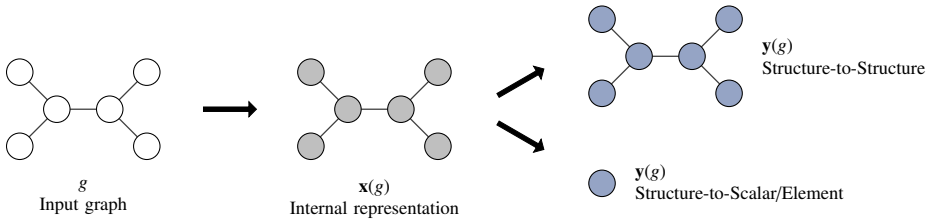


Figure 10.4: Schema of a general transduction.

10.2 Recursive approaches for trees

We have analyzed the Recurrent Neural Network (RNN) in Section 6.4. This technique boot-up encoding of input trees, extending states for children of vertex in the tree and gives a universal approximation over the tree domain. The RNN can be executed over different structures like unfolding and encoding network.

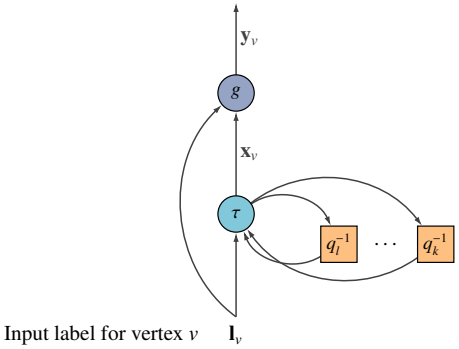


Figure 10.5: Graphical model of RNN for trees, where $\mathbf{x}_v = \tau(\mathbf{x}_{ch[v]}, \mathbf{l}_v)$ and $\mathbf{y}_v = g(\mathbf{x}_v, \mathbf{l}_v)$.

RNN transduction admit a recursive state representation with the following properties:

- *Causality*, the transduction computed in correspondence of a vertex v depends only on v and its descendants.
- *Stationary*, the transduction computed in correspondence of a vertex v is independent on node v.
- *Adaptivity*, the transduction is learnt from observed data.

During years different models of recurrent approaches for tree are studied:

- *Recursive Cascade Correlation* (RCC, 1997-2000), Where the recursive network is build by the Cascade Correlation for a constructive approach that adds a new layer for each training step and a number of layer that is automatically computed by the training algorithm.
- *Unsupervised recursive models* (2003-2005), transfer recursive idea to unsupervised learning with no prior metric or pre-processing. It is an evolution of the similarity measure trough recursive comparision of sub-structures. It's possible to have recursive nodes embedding on a Self Organization Map.
- Tree ESN (2010-2013), which combine the Reservoir Computing and the recursive modeling to extend the applicability of the RC/ESN approach to the tree structured data, obtaining an extremely efficient way of modeling RNN.
- Deep Tree ESN (2018), that corresponds to the hierarchical abstraction both through the input structure and architectural layers.
- Hidden Tree Markov Models (HTMM, 2012-2018), an example is the Bottom-up HTMM that extends HMM to trees exploiting the recursive approach; it is a generative process from the leaves to the root that take into account the Markov assumpiton $Qch_1(u), \ldots, Qch_k(u) \rightarrow Qu$ where Q are hidden

states variables with discrete values and *ch* represent the "child". The children to parent hidden state transition corresponds to $\mathbb{P}(Q_u|Qch_1(u), \ldots, Qch_k(u))$. The issue is how we can decompose the joint state transition represented in Figure 10.5.

10.3 Analysis and moving to graphs

Inherited from time sequence processing, the RNN, allow adaptive representation of SD, handling of variability by stationary, with efficacy solution to parsimony without reducing expressive power, and causality, that affects the computational power.

The causality assumpion in RNN introduce issue in processing cycles (due to the mutual dependencies among state values), two main approaches to solve this fact are:

1. RNN, that explicity treat the cycles consisting state dynamics to be contractive (GrpahESN, GNN)
2. Layering, a contextual non-recursive approach (originated with NN4Gs and then as CNN for graphs), where the mutual dependencies are managed (architecturally) through different layers. Instead of iterating at the same layer, each vertex can take the context of the other vertices computed in the previous layers, accessing progressively to the entire graph/network, and each vertex take information from all the other, including the mutual influences (without the convergence issues).

11. Bayesian Networks

11.1 Introduction

We refer to Bayesian learning as a probabilistic learning and the term *Bayesian* is used since it is very common to use Bayes theorem in this field.
Why should we introduce Bayesian networks since we have neural networks?
Because many modern machine learning techniques use probabilistic models.

Definition 11.1.1 (Random variable)**.** *A **random variable (RV)** is a function describing the outcome of a random process by assigning unique values to all possible outcomes of the experiment.*

> **Example:**
> An example of *discrete* random variable is the output of a coin tossing.

From now on we will refer to discrete random variables, except for those cases where we will explicitly say the contrary.

Definition 11.1.2 (Sample space)**.** *The **sample space** S of a random process is the set of all possible outcomes.*

> In the coin tossing scenario, the *sample space* is $S = \{\text{head}, \text{tail}\}$.

Definition 11.1.3 (Event)**.** *An **event** e is a set of outcomes, that may occur or not as a result of the experiment. Equivalently, e is a subset of the sample space: $e \in S$.*

Definition 11.1.4 (Probability function)**.** *A **probability function** $\mathbb{P}(X = x) \in [0, 1]$ ($\mathbb{P}(x)$ in short) measures the probability of event x occurring. Formally, the probability of a random variable X attaining the value x.*

Definition 11.1.5 (Joint probability)**.** *If the random process is described by a set of RVs $X_1, \ldots, X_N$, then the joint conditional probability writes*

$$\mathbb{P}(X_1 = x_1, \ldots, X_N = x_n) = \mathbb{P}(x_1 \wedge \cdots \wedge x_n)$$

Definition 11.1.6 (Sum rule)**.** *The probabilities of all the events belonging to the* sample space *must sum to* 1*:*

$$\sum_{x \in S} \mathbb{P}(X = x) = 1$$

Definition 11.1.7 (Conditional probability)**.** *We term **conditional probability** a measure of the probability of an event A occurring given that another event B has (by assumption, presumption, assertion or evidence) occurred. Such a probability is denoted as $\mathbb{P}(A|B)$.*

> **Example:**
> For example, the probability that any given person has a cough on any given day may be only 5%. But if we know or assume that the person has a cold, then they are much more likely to be coughing. The conditional probability that someone coughing is unwell might be 75%, then: $\mathbb{P}(\text{Cough}) = 5\%$ and $\mathbb{P}(\text{Sick}|\text{Cough}) = 75\%$.

Definition 11.1.8 (Product rule)**.** *We term **product rule** or **chain rule** the following*

- ***Two random variables.** Let X, Y be two random variables. Their joint distribution can be computed as*

$$\mathbb{P}(X, Y) = \mathbb{P}(X|Y) \cdot \mathbb{P}(Y) = \mathbb{P}(Y|X) \cdot \mathbb{P}(X)$$

- ***More than two RVs.*** *Let $X_1, \dots, X_N$ be random variables. Their joint distribution can be computed as conditional probability to obtain:*

$$\begin{aligned}\mathbb{P}(X_N, \dots, X_1) &= \mathbb{P}(X_N | X_{N-1}, \dots, X_1) \cdot \mathbb{P}(X_{N-1}, \dots, X_1) \\ &= \mathbb{P}(X_N | X_{N-1}, \dots, X_1) \cdot \mathbb{P}(X_{N-1} | X_{N-2}, \dots, X_1) \cdot \mathbb{P}(X_{N-2}, \dots, X_1) \\ &= \prod_{i=1}^{N} \mathbb{P}(X_i | X_{i-1}, \dots, X_1)\end{aligned}$$

- ***More than two RVs and conditional probability.*** *Let $X_1, \dots, X_N, Y$ be random variables. The posterior probability of $X_1, \dots, X_N$ given Y is*

$$\mathbb{P}(X_1, \dots, X_N | Y) = \prod_{i=1}^{N} \mathbb{P}(X_i | X_1, \dots, X_{i-1}, Y) = \mathbb{P}(X_1 | Y) \cdot \mathbb{P}(X_2 | X_1, Y) \cdot \dots \cdot \mathbb{P}(X_N | X_1, \dots, X_{N-1}, Y)$$

This definition reflects the fact that the realization of an event Y may affect the occurrence of X.

Definition 11.1.9 (Marginalization)**.** ***Marginalisation*** *in probability refers to "summing out" the probability of a random variable X given the joint probability distribution of X with other variable(s).*
Formally,

$$\mathbb{P}(X = x) = \sum_{y \in S_y} \mathbb{P}(X = x, Y = y) = \sum_{y \in S_y} \mathbb{P}(X = x | Y = y) \cdot \mathbb{P}(Y = y)$$

where S_Y denotes the sample space of the random variable Y.

Observation 11.1.1. *The marginalization comes in handy if we want to compute $\mathbb{P}(X = x)$, but we are not given a direct probability distribution over X. We are provided with a joint probability distribution over X and some other random variable(s) Y.*
Informally, to find $\mathbb{P}(X = x)$, we sum all the probability values where $X = x$ occurs with all possible values of Y (e.g. $y_1, y_2, \dots y_n$).
We can find how often $X = x$ occurs if we consider how often $X = x$ occurs with each individual value of Y, and sum up all such values to get the total value of the "often-ness" of $X = x$.

> **Example:**
> Let us consider the two binary random variables S ="Riccardo slips while walking" and R ="The weather is rainy", that can assume Boolean values. If we want to estimate how likely it is for Riccardo to slip, we can use marginalization technique as follows
>
> $$\mathbb{P}(S) = \mathbb{P}(S, R) + P(S, \overline{R}) = \mathbb{P}(S, R) \cdot \mathbb{P}(R) + \mathbb{P}(S, \overline{R}) \mathbb{P}(\overline{R})$$
>
> where $\overline{R}$ indicates the event of "not raining".

Theorem 11.1.1 (Bayes rule)**.** *Given an hypothesis $h_i \in H$ and the whole training set $\mathbf{d}$ we can compute the "consistency" of h_1 with the training set as*

$$\mathbb{P}(h_i | \mathbf{d}) = \frac{\mathbb{P}(\mathbf{d} | h_i) \cdot \mathbb{P}(h_i)}{\mathbb{P}(\mathbf{d})} \overset{\text{(marg)}}{=} \frac{\mathbb{P}(\mathbf{d} | h_i) \cdot \mathbb{P}(h_i)}{\sum_j \mathbb{P}(\mathbf{d} | h_j) \cdot \mathbb{P}(h_j)}$$

where

- $\mathbb{P}(h_i)$ *is the* ***prior*** *probability of h_i;*
- $\mathbb{P}(\mathbf{d} | h_i)$ *is the* ***conditional probability*** *of observing $\mathbf{d}$ given that hypothesis h_i is true (*likelihood*);*
- $\mathbb{P}(\mathbf{d})$ *is the* ***marginal probability*** *of $\mathbf{d}$;*
- $\mathbb{P}(h_i | \mathbf{d})$ *is the* ***posterior probability*** *that hypothesis is true given the data and the previous belief about the hypothesis.*

The second form of the Bayes rule says that since we do not know the probability distribution of our data $\mathbb{P}(\mathbf{d})$ we use marginalization to have some "proxy".

Definition 11.1.10 (Independence)**.** *Let X and Y be two random variables. We say that X and Y are* ***independent*** *if knowledge about X does not change the uncertainty about Y and vice versa.*
Formally,

$$I(X,Y) \iff \mathbb{P}(X,Y) = \mathbb{P}(X|Y)\cdot\mathbb{P}(Y) = \mathbb{P}(Y|X)\cdot\mathbb{P}(X) = \mathbb{P}(X)\cdot\mathbb{P}(Y)$$

Definition 11.1.11 (Conditional independence)**.** *Let X and Y be two random variables. X and Y are* ***conditionally independent*** *given Z if the realization of X and Y is an independent event of their conditional probability distribution given Z. Formally,*

$$I(X,Y|Z) \iff \mathbb{P}(X,Y|Z) = \mathbb{P}(X|Y,Z)\cdot\mathbb{P}(Y|Z) = \mathbb{P}(Y|X,Z)\cdot\mathbb{P}(X|Z) = \mathbb{P}(X|Z)\cdot\mathbb{P}(Y|Z)$$

Let us consider once again the two random variables S and R, adding G ="Gemma is starving". It goes without saying that S and G are independent, hence we can write

$$\mathbb{P}(S,G) = \mathbb{P}(S|G)\cdot\mathbb{P}(G) = \mathbb{P}(G|S)\cdot\mathbb{P}(S) = \mathbb{P}(S)\cdot\mathbb{P}(G)$$

Conversely, one could assume that S and R are two dependent random variables, but if we know that Riccardo stays at home (H) then the weather has no influence of his fall out

$$\mathbb{P}(S,R|H) = \mathbb{P}(S|R,H)\cdot\mathbb{P}(R|H) = \mathbb{P}(R|S,H)\cdot\mathbb{P}(S|H) = \mathbb{P}(R|H)\cdot\mathbb{P}(S|H)$$

Notice that once we have the joint probability distribution of n random variables, we can perform a marginalization and reconstruct the probability of any of the random variables.
The issue here is that storing all possible configurations is exponential in space.

$\mathbf{X_1}$	…	$\mathbf{X_i}$	…	$\mathbf{X_n}$	$\mathbb{P}(\mathbf{X_1},\dots,\mathbf{X_n})$
x_1'	…	x_i'	…	$x_n{}'$	$\mathbb{P}(x_1',\dots,x_n')$
⋮		⋮		⋮	⋮
x_1^l	…	x_i^l	…	$x_n{}^l$	$\mathbb{P}(x_1^l,\dots,x_n^l)$

Table 11.1: Joint probability distribution table

Let us examine three ways of conducting probability learning.

Bayesian learning

Bayesian learning aims at providing the probability distribution of values of unseen data, using what has been observed so far

$$\mathbb{P}(X|\mathbf{d}) = \sum_i \mathbb{P}(X|\mathbf{d},h_i)\cdot\mathbb{P}(h_i|\mathbf{d}) = \sum_i \underbrace{\mathbb{P}(X|h_i)}_{\text{hp prediciton}}\cdot\underbrace{\mathbb{P}(h_i|\mathbf{d})}_{\text{posterior}} \tag{11.1}$$

where $\mathbf{d}$ is a vector representing all the training data and X is the random variable that describes the new example. It is crucial to observe that Bayesian learning makes use of *all* the hypotheses and their probabilities.
The issue here is that in a continuous scenario the summation becomes an integral, but also that in the simple discrete world the problem may become unfeasible, due to a huge number of possibilities for the functions (whole H).

Maximum A Posteriori (MAP)

This approach overcomes the issue of Bayesian learning, namely the fact that we would have to sum over too many different hypotheses.
MAP approximates the probability of encountering a new example with values ruled by the random variable X, through the most likely hypothesis h_{MAP} ($\mathbb{P}(X|h_{\text{MAP}})$), that is computed as

$$h_{\text{MAP}} = \arg\max_{h\in H}\mathbb{P}(h|d) \overset{\text{(Bayes)}}{=} \arg\max_{h\in H}\frac{\mathbb{P}(d|h)\cdot\mathbb{P}(h)}{\underbrace{\mathbb{P}(d)}_{\text{constant}}} = \arg\max_{h\in H}\mathbb{P}(d|h)\mathbb{P}(h)$$

Maximum likelihood

In the case of an hypothesis space such that all its elements are equally likely ($\mathbb{P}(h) = \mathbb{P}(h'), \forall h, h' \in H$) the term $\mathbb{P}(h)$ is constant and can be removed. This leads to a different computation of the most likely hypothesis (maximum a posteriori estimation), namely

$$h_{ML} = \arg\max_{h \in H} \mathbb{P}(d|h)$$

Aggiungere esempio con spiegazione ed immagini degli esempi cinema e caramelle

Definition 11.1.12 (Bayesian network)**.** *It is a directed acyclic graph (DAG), where the nodes represent random variables and conditional dependence between variables is represented through edges.*
Formally, if there is an edge (X, Y), connecting two random variables X and Y it means that $\mathbb{P}(X, Y)$ is a factor in the joint probability distibution.

Fact 11.1.2. *Bayesian networks satisfy the local Markov property: every variable is conditionally independent of its non-descendants, given its parents.*

Example:
Let us consider Figure 11.1. The Markov property implies that

$$\mathbb{P}(Z|X, Y, S) = \mathbb{P}(Z|X, Y)$$

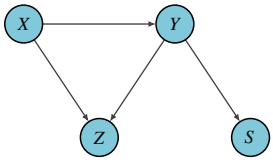


Figure 11.1: An example of Bayesian network.

Fact 11.1.3. *Conditional probability tables (CPT) local to each node describe the probability distribution given its parents*

$$\mathbb{P}(Y_1, \ldots, Y_n) = \prod_{i=1}^{n} \mathbb{P}(Y_i|Y_1, Y_2, \ldots, Y_n) = \prod_{i=1}^{n} \mathbb{P}(Y_i|parents(Y_i))$$

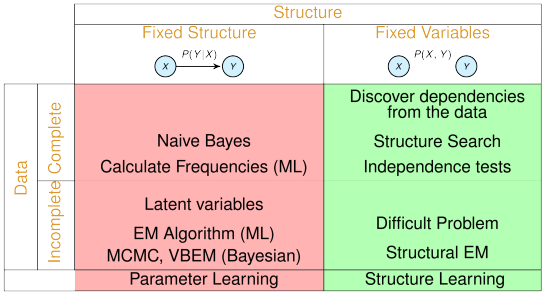


Figure 11.2: This table splits the situation into 4 main problems. We are going to address the problems where the structure is fixed. For complete data we refer to "fully observed variables", while the other row contains hidden variables.

11.2 Parameters learning with fully observed variables

Our aim here is to find numerical parameters for a Bayesian network with a *known structure*.
Formally, we want to determine the best hypothesis h_θ regulated by a (set of) parameter θ.

Example:
h_θ is the function that models the situation in which the expected proportion of coin tosses returning heads is θ.

The usual Bayesian probabilities are:

- *Prior* $\mathbb{P}(h_\theta) = \mathbb{P}(\theta)$;
- *Likelihood* $\mathbb{P}(\mathbf{d}|h_\theta) = \mathbb{P}(\mathbf{d}|\theta)$
- *Posterior* $\mathbb{P}(h_\theta|\mathbf{d}) = \mathbb{P}(\theta|\mathbf{d})$ (we are getting a function that says how likely the various values of the parameters are).

Fact 11.2.1. *If hypotheses are equiprobable it is reasonable to use the* Maximum Likelihood Estimation *and select the best parameter θ such that*

$$\theta_{ML} = \frac{H}{H+T}$$

where H accounts for the number of heads and T for the number of tails.

Proof. The maximum likelihood estimation is equivalent to writing that we want the parameters configuration θ such that the model is most likely to have generated the data $\mathbf{d}$

$$\theta_{ML} = \arg\max_{\theta\in\Theta} \mathbb{P}(\mathbf{d}|\theta)$$

where $\mathbf{d} = (d_1, \dots, d_l)$, such that $d_i = \begin{cases} 1 \text{ if head}(i) \\ 0 \text{ otherwise} \end{cases}$, from a family of parameterized distributions $\mathbb{P}(x|\theta)$.
This is an optimization problem that considers the likelihood function $L(\theta|x) = \mathbb{P}(x|\theta)$ to be a function of θ.
What we want to infer is $\mathbb{P}(\mathbf{d}|\theta)$.
We could approximate it using the Bernoulli (binomial) distibution, so we get

$$\mathbb{P}(\mathbf{d}|\theta) = \prod_{i=1}^{l} \mathbb{P}(d_i|\theta) = \theta^H \cdot (1-\theta)^T$$

One could think that a possible strategy would be to use the gradient descent algorithm, since we want to minimize a function, but we do not want to differentiate an exponential function.
The nifty trick here is to plug the logarithm and get that the estimate of the probability of a coin tossing returning head is

$$\max \log \mathbb{P}(H|\theta) = \max \log(\theta^H \cdot (1-\theta)^T) = \max(H \log\theta + T \cdot \log(1-\theta))$$

Le us differentiate $\log \mathbb{P}(H|\theta)$:

$$\frac{\partial \log(\mathbb{P}(H|\theta))}{\partial\theta} = \frac{\partial(H \cdot \log\theta + T \cdot \log(1-\theta))}{\partial\theta} = H \cdot \frac{\partial \log\theta}{\partial\theta} + T \cdot \frac{\partial \log(1-\theta)}{\partial\theta} = \frac{H}{\theta} - \frac{T}{1-\theta} \tag{11.2}$$

Since we are interested in finding the maximum of the logarithm of $\mathbb{P}(H|\theta)$ we impose the derivative to 0 and we get

$$H - H\theta - T\theta = 0 \iff H = \theta \cdot (H+T) \iff \theta_{ML} = \frac{H}{H+T}$$

■

What about the a posteriori reasoning?
We want to plug a probability distribution in order to take into account further facts, but the issue here is how to pick $\mathbb{P}(\theta)$ such that the result follows the same distribution?
Some mathematical theory says that the *conjugate* distribution of the binomial is the *beta* distribution:

$$\mathbb{P}(\theta|\mathbf{d}) = \mathbb{P}(\mathbf{d}|\theta) \cdot \mathbb{P}(\theta) \approx \mathbb{P}(\mathbf{d}|\theta) \cdot Beta(\alpha_H + \#heads, \alpha_T + \#tails)$$

11.2.1 Continuous problems

Just like we did in the discrete case, we are interested in computing the distribution parameters of our data (assuming *Gaussian distribution* and *independent and identically distributed data* $\mathbf{d} = (x_1, \dots, x_j, \dots, x_l \in \mathbb{R}^l)$).

Fact 11.2.2. *We can select the best parameters σ and μ as follows:*

$$\mu = \frac{\sum_{j=1}^{l} x_j}{N} \qquad \sigma = \sqrt{\frac{\sum_{j=1}^{l} (x_j - \mu)^2}{l}}$$

Proof.

$$\mathbb{P}(X|\mu, \sigma) = \frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(- \frac{(X-\mu)^2}{2\sigma^2} \right)$$

Just like the discrete case

$$< \mu_{ML}, \sigma_{ML} >= \arg\max_{\mu,\sigma} \mathcal{L}(\mu, \sigma|\mathbf{d}) = \arg\max_{\mu,\sigma} \prod_{j=1}^{l} \mathbb{P}(x_j|\mu, \sigma)$$

In general, it is better to work with logarithms (*log-likelihood*) by maximizing

$$\log(\mathcal{L}(\mu, \sigma|\mathbf{d})) = \log \prod_{j=1}^{l} \mathbb{P}(x_j|\mu, \sigma) = \sum_{j=1}^{l} \log \mathbb{P}(x_j|\mu, \sigma) =$$
$$= \sum_{n=1}^{l} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(- \frac{(X_j - \mu)^2}{2\sigma^2} \right) \right) = \sum_{j=1}^{l} \left(- \log(\sigma) - \frac{(x_j - \mu)^2}{2\sigma^2} \right)$$

Let us differentiate with respect to μ and σ separately and force them equal to 0 for finding the maximum of the log likelihood:

$$\begin{cases} \frac{\partial\left(\sum_{j=1}^{l} - \log(\sigma) - \frac{(x_j-\mu)^2}{2\sigma^2} \right)}{\partial\sigma} = \sum_{j=1}^{l} - \frac{\partial \log(\sigma)}{\partial\sigma} - \frac{\partial\left(\frac{(x_j-\mu)^2}{2\sigma^2} \right)}{\partial\sigma} = \sum_{j=1}^{l} \frac{1}{\sigma} - \frac{(x_j-\mu)^2}{2} \cdot (-2) \cdot \frac{1}{\sigma^3} = \sum_{j=1}^{l} \frac{1}{\sigma} - \frac{(x_j-\mu)^2}{\sigma^3} \\ \frac{\partial\left(\sum_{j=1}^{l} - \log(\sigma) - \frac{(x_j-\mu)^2}{2\sigma^2} \right)}{\partial\mu} = \sum_{j=1}^{l} - \frac{\partial \log(\sigma)}{\partial\mu} - \frac{\partial\left(\frac{(x_j-\mu)^2}{2\sigma^2} \right)}{\partial\mu} = \sum_{j=1}^{l} - \frac{2\cdot(x_j-\mu)}{2\sigma^2} \cdot (-1) = \sum_{j=1}^{l} \frac{(x_j-\mu)^2}{\sigma^2} \end{cases} \tag{11.3}$$

Hence,

$$\begin{cases} 0 = \sum_{j=1}^{l} \frac{1}{\sigma} - \frac{(x_j-\mu)^2}{\sigma^3} \iff 0 = -\frac{l}{\sigma} + \frac{1}{\delta^3} \sum_{j=1}^{l} (x_j - \mu)^2 \iff \sigma^2 l - \sum_{j=1}^{l} (x_j - \mu)^2 = 0 \iff \sigma^2 = \frac{\sum_j (x_j-\mu)^2}{l} \\ 0 = \sum_{j=1}^{l} \frac{x_j-\mu}{\sigma^2} \iff 0 = \frac{1}{\sigma^2} \cdot (-l\mu + \sum_{j=1}^{l} x_j) \iff \mu = \frac{\sum_j x_j}{l} \end{cases} \tag{11.4}$$

$$\mu = \frac{\sum_{j=1}^{l} x_j}{N} \qquad \sigma = \sqrt{\frac{\sum_{j=1}^{l} (x_j - \mu)^2}{l}}$$

Let us consider a supervised problem, that we want to solve using a linear model:

$$y_j = \theta_1 x_j + \theta_2 + \varepsilon_j \text{ s.t. } \varepsilon_j \approx \mathcal{N}(0, \sigma^2), \ \forall j = 1, \dots, l$$

where $\mathcal{N}$ stands for noise with center 0 and σ as variance. Notice that σ is not a parameter. Since $y_j \approx \mathcal{N}(\theta_1 x_j + \theta_2, \sigma^2)$ we can learn the conditional distribution $\mathbb{P}(y|x)$ by maximizing

$$\log(\mathcal{L}(\theta_1, \theta_2|\mathbf{d})) = \sum_{j=1}^{N} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \cdot \exp\left(- \frac{(y_j - (\theta_1 x_j + \theta_2))^2}{2\sigma^2} \right) \right)$$

The solution of the maximum likelihood that we obtain is

$$< \theta_{1\ ML}, \theta_{2\ ML} >= \arg\min_{\theta_1,\theta_2} \sum_{n=1}^{N} (y_j - (\theta_1 x_j + \theta_2))^2$$

■

11.2.2 Learning Naive Bayes classifiers

Naive Bayes classification for $x =< a_1, \ldots, a_L >$

$$c_{NB} = \arg\max_{c \in C} \mathbb{P}(c|x) = \arg\max_{c \in C} \mathbb{P}(c) \prod_{l=1}^{L} \mathbb{P}(a_l|c)$$

Maximum likelihood estimation of the distributions

- $\mathbb{P}(a_l = s|c = k)$ such that $1 \leq s \leq S$ and $1 \leq k \leq K$
- $\mathbb{P}(c = k)$ such that $1 \leq k \leq K$

Given N observed training pairs (x_j, c_j) such that $x_j =< a_1, \ldots, a_l >$

$$\begin{aligned}\mathcal{L}(\theta|\mathbf{d}) = \mathbb{P}(\mathbf{d}|\theta) &= \prod_{j=1}^{D} \mathbb{P}(c_j) \prod_{l=1}^{L} \mathbb{P}(a_{lj}|c_j) \\ &= \prod_{j=1}^{D} \prod_{k=1}^{K} \mathbb{P}(c = k)^{z_{jk}} \left(\prod_{l=1}^{L} \prod_{s=1}^{S} \left(\mathbb{P}(a_l = s|c = k) \right)^{t_j^{ls}} \right)^{z_{jk}} \quad \text{for} \quad z_{jk} = \begin{cases} 1 \text{ if } x_j \text{ is classified } k \\ 0 \text{ otherwise} \end{cases} \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad t_j^{ls} = \begin{cases} 1 \text{ if } a_{jl} \text{ is classified } 0 \\ 0 \text{ otherwise} \end{cases}\end{aligned}$$

Maximize with respect to parameters $\theta =< \mathbb{P}(c = k), \mathbb{P}(a_l = s|c = k) >$ subject to $\sum_k \mathbb{P}(c = k) = 1$ and $\sum_s \mathbb{P}(a_l = s|c = k) = 1$

$$\log(\mathcal{L}(\theta|\mathbf{d})) = \sum_{j=1}^{D} \sum_{k=1}^{K} z_{jk} \log \mathbb{P}(c = k) + \sum_{j=1}^{D} \sum_{k=1}^{K} z_{jk} \left(\sum_{l=1}^{L} \sum_{s=1}^{S} t_j^{ls} \log \mathbb{P}(a_l = s|c = k) \right)$$

yields

$$\begin{aligned}\mathbb{P}(c = k) &= \frac{\sum_{j=1}^{D} z_{jk}}{D} = \frac{N(k)}{D} \\ \mathbb{P}(a_l = s|c = k) &= \frac{\sum_{j=1}^{D} z_{jk}\, t_j^{ls}}{\sum_{j=1}^{D} \sum_{s=1}^{S} z_{jk}\, t_j^{ls}} = \frac{N_{ls}(k)}{\sum_{s=1}^{S} N_{ls}(k)}\end{aligned}$$

Learning essentially amounts to counting frequencies. Naive Bayes classification works surprisingly well when attributes are close to be independent, noisy data, high dimensional problems (scalability), and large datasets (point estimates). Need to be careful when applying Naive Bayes to sparse data.
However, what happens if a class c_k has no occurrences of an attribute $a_l = s$

$$\mathbb{P}(a_l|c = k) = \frac{N_{ls}(k)}{\sum_{s=1}^{S} N_{ls}(k)} = 0 \Rightarrow \mathbb{P}(c = k|x) = 0 \ \forall x$$

Smoothing involves to dealing with the zero events problem. Add a constant term α in both the numerator and the denominator to smooth the estimation

$$\mathbb{P}(a_l = s|c = k) = \frac{N_{ls}(k) + \alpha}{\sum_{s=1}^{S} N_{ls}(k + \alpha)}$$

If $\alpha = 1$ we have the Laplace smoothing. That can improve the Naive Bayes performance up to 20%, but it can also cause interference in learning, giving too much probability to unfrequented events. The parameter α is a priori estimate of the attribute-class probability. Consider $\beta = \mathbb{P}(a_l = s|c = k)$ as random variable we can have the prior distribution $\mathbb{P}(\beta|\alpha)$ with hyperparameter α.

Application of Naive Bayes to text classification

That loads of useful application, as learn to classify web-pages by topic or determine id an incoming email contains spam. The problem is characterized by a large sample size (i.e. large document collections) and a high dimensional data (i.e. the vocabulary).

We define *Bag of Words Assumption* the word order is not relevant for determining document semantics. To count the occurrences of each dictionary word in the document we may represent a document d ad a vector x_d of word counts. Its easy to compute frequencies from word counts.

Algorithm 11.1 gives the pseudocode of the text classifier.

1: Given a set of N training documents represented as vector of words counts $x_j = [w_1, \dots, w_l, \dots, w_L]$, where L is the vocabulary size.
2: **for each** document classification $k = 1$ to K **do**
3: $doc(k)$ = set of training documents in class k
4: $\mathbb{P}(c = k) = |doc(k)|/N$
5: $text(k)$ = concatenation of all docs in $doc(k)$
6: $N_j = |text(k)|$ including duplicates
7: **for each** work w_l in $text(k)$ **do**
8: n_l = no occurrences of w_l in $text(k)$
9: $\mathbb{P}(w_l|c = k) = \frac{n_l+1}{N_j+L}$

Algorithm 11.1: Text classification algorithm.

11.3 Parameters learning with hidden variables

Here we have incomplete data that may can be missing observation (imputation) or unobserved random variables. The key idea is to complete the data making hypothesis on the unobserved variables.

An algorithm developed for this situation is the Expectation-Maximization (EM) Algorithm, where with Expectation we mean to compute the current estimate of the hidden variables (expected counts), while with maximization we refer too use the estimate to update the model parameters θ.
Likelihood maximization procedure

$$\theta_{ML} = \arg\max_{\theta} \mathcal{L}(\theta|X) = \arg\max_{\theta \in \Theta} \theta \mathbb{P}(X|\theta)$$

but $\mathcal{L}(\theta|X)$ is incomplete likelihood and X is the incomplete data containing only observed random variables. Assume complete data $d = (X, Z)$ exists where $z \in Z$ is an hidden or latent variable compute the complete likelihood

$$\mathcal{L}_c(\theta|d) = \mathbb{P}(d|\theta) = \mathbb{P}(X, Z|\theta) = \mathbb{P}(Z|X, \theta)\mathbb{P}(X|\theta)$$

The pseudocode of the algorithm is expressed in Algorithm 11.2.

1: Initialize $k = 1$
2: Initialize parameters $\theta^{(1)}$
3: **repeat**
4: $Q(\theta|\theta^{(k)}) = E_{Z|X,\theta^{(k)}}[\log \mathcal{L}(\theta|X, Z)]$ ▷ Exprectation step
5: $\theta^{(k+1)} = \arg\max_\theta Q(\theta|\theta^{(k)})$ ▷ Maximization step
6: $k = k + 1$
7: **until** $\log \mathcal{L}_c$ stops increasing

Algorithm 11.2: Expectation-Maximization algorithm.

Theorem 11.3.1. *An EM process ensures* $\mathcal{L}(\theta^{(k+1)}|X) \geq \mathcal{L}(\theta^{(k)}|X)$.

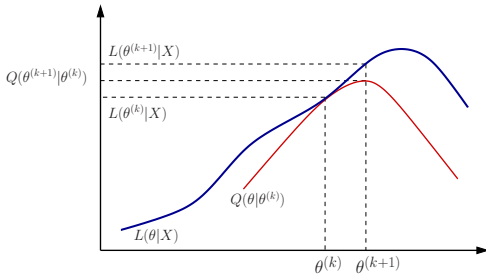


Figure 11.3: Illustration related to Theorem 11.3.1.

11.3.1 Learning Gaussian Mixture Models

Previously the maximum likelihood provides the estimations for a single univariate Gaussian. Now, with EM estimation we can estimate M univariate Gaussians, and this process is know as Gaussian Mixture Model (GMM). In this model z_{jm} is the hidden mixture selection variable with prior $\mathbb{P}(z_{jm}) = \pi_m$. Each observation x_j is generated by a single Gaussian $\mathbb{P}(x_j|z_{jm}) \approx \mathcal{N}(\mu_m, \sigma_m)$. Now the model parameters are $\theta = (\pi, \mu, \sigma)$.

A set of independently and identically distributed observations $X = \{x_1, \ldots, x_n\}$ has incomplete likelihood $\mathcal{L}(\theta|X)$, using marginalization to introduce the hidden variable z_{jm} we obtain

$$\mathcal{L}(\theta|X) = \mathbb{P}(X|\theta) = \prod_{j=1}^{N} \mathbb{P}(x_j|\theta) \quad \Rightarrow \quad \mathcal{L}(\theta|X) = \prod_{j=1}^{N} \sum_{m=1}^{M} \mathbb{P}(x_j, z_{jm}|\theta)$$

Following the independence relationships in the graphical model this rewrites

$$\mathcal{L}(\theta|X) = \prod_{j=1}^{N} \sum_{m=1}^{M} \mathbb{P}(z_{jm}|\pi)\mathbb{P}(x_j|z_{jm}, \mu, \sigma)$$

Assume we know the hidden assignment $z_{jm} = 1$ for each sample j, we have the following complete likelihood, that better handled by taking the log results as

$$\log \mathcal{L}_c(\theta|X, Z) = \sum_{j=1}^{N} \sum_{m=1}^{M} z_{jm} \log \left(\pi_m \, \mathbb{P}(x_j|\mu_m, \sigma_m)\right)$$

Now we can compute the auxiliary function

$$\begin{aligned} Q(\theta|\theta^{(k)}) &= E_{Z|X,\theta^{(k)}}[\log \mathcal{L}_c(\theta|X, Z)] = \\ &= \sum_{j=1}^{N} \sum_{m=1}^{M} \mathbb{P}(z_{jm}|x_j, \theta^{(k)}) \log \left(\pi_m^{(k)} \, \mathbb{P}(x_j|\mu_m^{(k)}, \sigma_m^{(k)})\right) \end{aligned}$$

that can be solve (expectation step of EM algorithm) estimating the posterior $\mathbb{P}(z_{jm}|x_j, \theta^{(k)})$ based on the current $\theta^{(k)}$ values

$$\mathbb{P}(z_{jm}|x_j, \theta^{(k)}) = \frac{\pi_m^{(k)} \mathbb{P}(x_j|\mu_m^{(k)}, \sigma_m^{(k)})}{\sum_m' \pi_m'^{(k)} \mathbb{P}(x_j|\mu_m'^{(k)}, \sigma_m'^{(k)})}$$

Then, evaluating the maximization of the auxiliary function (maximization step of EM algorithm) with respect to the parameters $\theta = (\pi, \mu, \sigma)$

$$Q(\theta|\theta^{(k)}) = \sum_{j=1}^{N} \sum_{m=1}^{M} \mathbb{P}(z_{jm}|x_j, \theta^{(k)}) \log \pi_m^{(k)} + \sum_{j=1}^{N} \sum_{m=1}^{M} \mathbb{P}(z_{jm}|x_j, \theta^{(k)}) \log \left(\mathbb{P}(x_j|\mu_m^{(k)}, \sigma_m^{(k)})\right)$$

As usual this is performed by solving the following equation taking into account the sum-to-one constraint $\sum_{m=1}^{M} \pi_m = 1$

$$\frac{\partial Q(\theta|\theta^{(k)})}{\partial \theta} = 0$$

Solving the independent maximization problems with respect to π_m, μ_m and σ_m yields

$$\pi_m^{(k+1)} = \frac{\sum_{j=1}^N \mathbb{P}(z_{jm}|x_j, \theta^{(k)})}{N}$$

$$\mu_m^{(k+1)} = \frac{\sum_{j=1}^N x_j \, \mathbb{P}(z_{jm}|x_j, \theta^{(k)})}{\sum_{j=1}^N \mathbb{P}(z_{jm}|x_j, \theta^{(k)})}$$

$$\sigma_m^{(k+1)} = \sqrt{\frac{\sum_{j=1}^N \mathbb{P}(z_{jm}|x_j, \theta^{(k)})(x_j - \mu^{(k)})^2}{\sum_{j=1}^N \mathbb{P}(z_{jm}|x_j, \theta^{(k)})}}$$

The EM algorithm can be summarized by the optimization problem

$$\theta^{(k+1)} = \arg\max_\theta \sum_z \mathbb{P}(Z = z|X, \theta^{(k)}) \log \mathcal{L}_c(\theta|X, Z = z)$$

The posterior $\mathbb{P}(Z = z|X, \theta^{(k)})$ provides an expected count for the vent z that is the EM counterpart of basic ML counting. The expectation-maximization algorithm is an example of Unsupervised Learning.

11.4 Recap on Bayesian Learning

Bayesian learning...

- ...is a powerful all-in-one model for modeling your knowledge about the world, interface and learning;
- ...uses conditional independence to break-down the joint probability into smaller pieces and describe the world;
- ...is generative because describes how data is generated and learns the joint probability;
- ...is parametric because need to make hypothesis on the data-generating distribution and nedd to choose suitable priors.

Two paradigms for learning statistical classifiers

1. Generative classifiers learn a model of the joint probability $\mathbb{P}(c, x)$ of the features and the corresponding class label. They provide a model of how data is generated, use Bayes rule to predict using class posterior $\mathbb{P}(c|x)$ and Bayesian Learning.
2. Discriminative classifiers directly model the class posterior $\mathbb{P}(c|x)$ or use a discriminant function. They directly target at optimizing classification accuracy. Moreover are best to solve a specific estimation problem rather than trying to estimate the general joint density.

Some pro and cons can be the following

- ✖ Discriminative approaches often lead to better classification/regression accuracies;
- ✖ Inference may require a significant computational cost
- ✖ Need to correctly guess the data generating and prior distributions;
- ✔ Elegant and solid mathematical basis: lots of tools for inference, probabilistic reasoning, learning, assessing the reliability of the acquired knowledge
- ✔ Straightforward to incorporate prior knowledge: data distributions and structure
- ✔ Powerful and sound techniques for dealing with complex unsupervised learning tasks: vision, natural language processing, ...

Bibliography

[1] Alan Lockett. Why would a saturated neuron be a problem? *Quora*, 07 2017. URL https://www.quora.com/Why-would-a-saturated-neuron-be-a-problem.