MACHINE LEARNING FOR VISION AND MULTIMEDIA

Lecture notes

Carlo Migliaccio

AA 2024/2025

Contents

Chapter 1

Introduction

Among the definitions one gives of **Machine learning** we can say that it is a *"Field of study that gives computers the ability to learn without being explicitly programmed"*. Nowadays, the *Artificial intelligence* is in general that the electricity was in the 19^{th} century. Something of paramount importance!
There are several methodologies and subfields in Machine Learning and the distinction is based on *how much and how* the human collaborate and of the type of provided data. The most important classification is the one between:

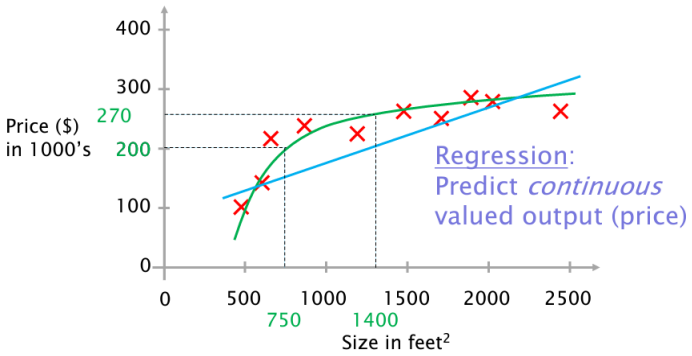
- *Supervised learning* (this course), is the approach which uses **a-priori knowledge** embedded in the data that are used for training algorithms and recognize patterns;
- *Unsupervided learning*, is the approach at the opposite whose main feature is not using *labeled data* for assess the tasks.
- *Other approaches*. Due to its vastness, in machine learning you can find for sure other subfields. For example the *Reinforcement learning, Semi-Supervised learning, trasnfer learning*. However they are all outside the purposes of this course.

1.1 Supervised learning

From Wikipedia (EN): Supervised learning (SL) is a paradigm in machine learning where input objects (for example, a vector of predictor variables) and a desired output value (also known as a human-labeled supervisory signal) train a model. In the following we are giving some simple introductory examples about two among the most used techniques in supervised learning.

1.1.1 Linear Regression

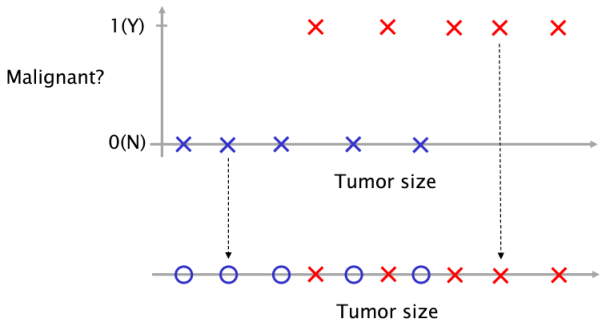
Let us imagine we are supposed to create a model that allows us to **predict the price of an house**. For sake of simplicity and clarity, suppose that our *dataset entries* have one feature (the house size in ft^2) and the *price* which represents the **correct answers**. Using this data we seek for a model which could predict, given the size of an unknown house, his price (in dollars, \$). Several choices can be made. At first, using either a linear or a nonlinear model and so on. It is remarkable that – even in such a simple example – we are facing a **supervised** problem since the right answers are given! This particular case is a problem of **regression** since we want to **predict a continuous valued output**, in our case the price.



In the figure above is shown the example in which two different models are used, clearly the predicted values for an unknown record is different according to the chosen model.

1.1.2 Classification

On the other hand, when we want to predict a discrete value (eg. YES/NO), we have a **classification** problem. Again, let us consider a trivial example: we want to predict whether a tumor is malignant or not according to its size. Even in this case we have one feature for the data (*tumor size*) and all the training data are labeled with the YES/NO answer.



The figure shows a graphical representation of the dataset. In this case since the answers are associated with different symbols, a more compact representation is given by a one-axis diagram: one feature is given, furthermore a different symbol is associated to different classes. Note that in this case we are in front of a a **binary classification problem**, in general the classes to predict are not necessarily in number of two.
This was just an example to understand and introduce the problem, but in real-world applications, one feature is not sufficient to build a good model! For sake of clarity, let us complicate

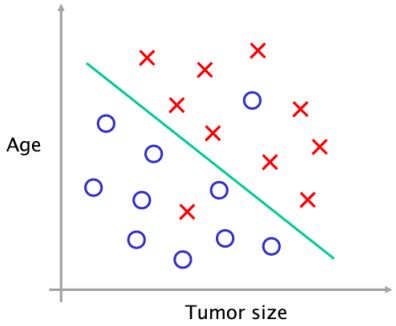


Figure 1.1: Bi-variate problem with linear decision boundary

a little bit the example we have just presented by adding a new feature associated with the *age of the patient*.

In this case the data set is represented in a 2D graph, one axis for each feature and a different symbol for each class. Now, given a record associated with a new patient, what is the class for its tumor? In this case can be useful to individuate a **decision boundary** according to which one can decide clearly what is the prediction (Positive/Negative). In the two parts there are some outliers, for this reason one can be tempted to build a more "accurate" decision boundary that perfectly split the two classes.

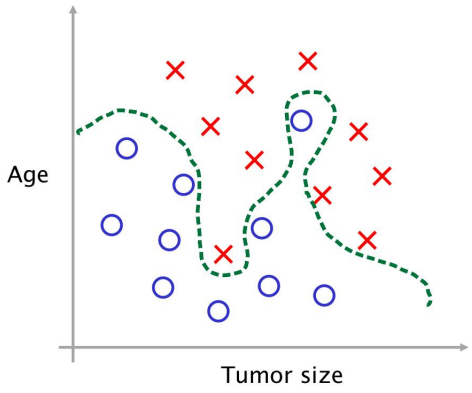


Figure 1.2: Example of overfitting

Is this a good model for the given problem? NO! This model will have very bad *performances of generalization* with new records to be classified, since it is too much related to the given dataset. In a colloquial way we say that: The model has learnt the by heart the dataset. A problem known as **overfitting**.
Finally, we can say that few features will result in a bad model, on the other hand also too much features will result in a bad model for another problem known as the **curse of dimensionality**.[1]

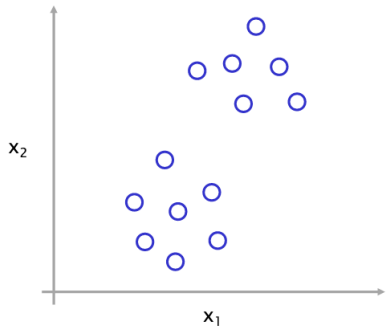
1.2 Unsupervided learning

At the opposite of the *supervised approach*, here patterns are learnt exclusively from unlabeled data. The most common example of such an approach is the **Clustering**.

[1] In these case techniques of dimensionality reduction has to be employed.

1.2.1 Clustering

In this case several algorithms are employed to discover groups called **clusters** associated with objects which are similar in some sense. In general, very often distance-based measures are used to individuate the groups. One of the most famous clustering algorithm is the *K-Mean*. The following figure is an example of bivariate clustered data.



Unsupervised techniques are used also in bioinformatics in manipulated *DNA microarrays*, for gropuing together similar web pages, for analysis of astronomical data and so on.

Chapter 2

Model, cost, parameter learning, Gradient Descent

Let us come back to the first example of *price prediction* and formalize some aspects we have only mentioned. The objective here is to exploit this example to introduce and better clarify several concepts.

2.1 Model representation

At first, the training set we are using is something similar to the following:

Size in feet^2(x)	**Price($) in 1000's($y$)**
2104	460
1416	232
1534	315
852	178
...	...

We will indicate with m the number of samples of the training set (number of rows), x is the input (possibly multivariate) variable, y is the output variable, (x, y) indicates generically a sample from the training set, while $(x^{(i)}, y^{(i)})$ indicates the i-th sample of the training set.

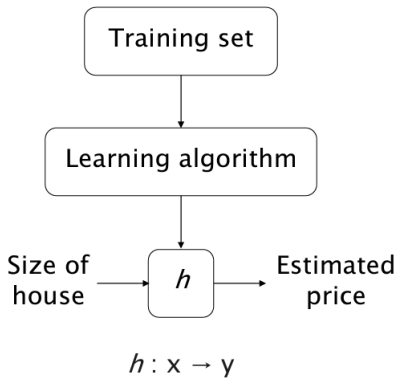


Figure 2.1: Scheme for model construction (price prediction)

The figure above shows schematically the steps in order to produce a certain model for the analysed case-study. Very briefly, a **training set** is used by a **learning algorithm** to obtain an *hypotesis* $h_\theta(x)$ which is later used for the prediction.
In the case we want to solve a **univariate linear regression problem** the hypotesis h has got the shape:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{2.1}$$

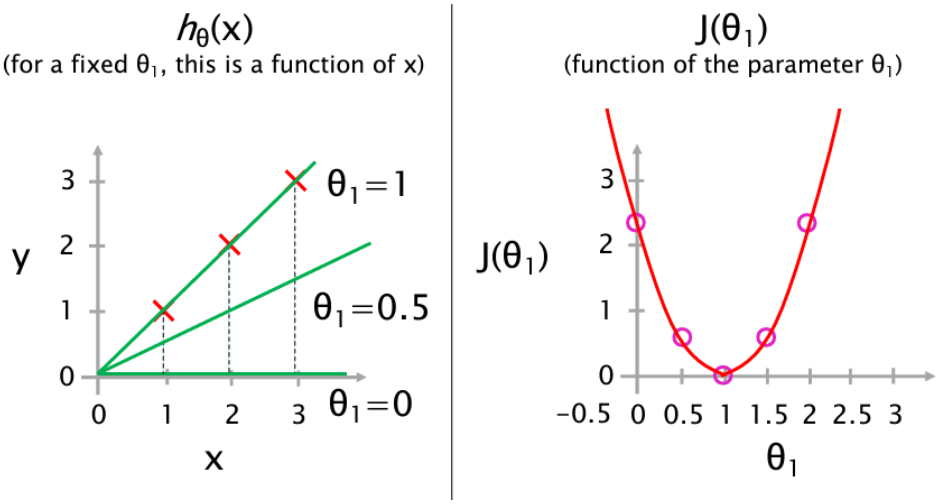
where θ_0 and θ_1 are the parameters of the line.[1] We call *univariate* the the problem since we have only one feature and it is a *linear regression* because we want to predict the price (output) according to a line.[2] **Question: How can we choose θ_0, θ_1?** Intuitively one can choose the parameters associated with the line $h_\theta(x)$ which is as closest as possible to the given y. Very often these parameters are the ones which solve the following problem:

$$\min_{\theta_0,\theta_1} \overbrace{\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\Big(\underbrace{h_\theta(x^{(i)})}_{\text{predicted value}} - \underbrace{y^{(i)}}_{\text{actual value}}\Big)^2}^{\text{SQUARE ERROR COST FUNCTION}} \tag{2.2}$$

The function $\frac{1}{2}(h_\theta(x^{(i)}) - y^{(i)})^2$ is the $\mathsf{Loss}(h_\theta(x), y)$ or $\mathsf{Cost}(h_\theta(x), y)$. If we call $J(\theta_0, \theta_1)$ the argument of the minimization problem the (2.2), the problem to solve can be expressed as

$$\min_{\theta_0,\theta_1} J(\theta_0, \theta_1) \tag{2.3}$$

Summarizing: we want to perform a prediction using the hypotesis h which is dependent on parameters θ_0, θ_1 which are issued by minimizing a certain functional $J(\theta_0, \theta_1)$. Let us investigate better on the role of J in this supervised learning task.
At first – for sake of simplicity – we can eliminate a degree of freedom fixing the parameter θ_0 to be (without loss of generality) $\theta_0 = 0$. For each choice of θ_1 we will obtain a $h_{\theta_1}(x)$. If we compute $J(\theta_1)$ (for each θ) will obtain a certain univariate function $J(\theta_1)$, the minimization of which will give us the *optimal* θ_1 parameter for our hypotesis. An example is shown in the following figure:



[1] We can imagine them as two handles to: move up/down the line (θ_0) and to rotate it (θ_1).

[2] Note that in case of a **neural network** the parameters and the hypotesis assume a different notation. In particular the hypotesis becomes the *predicted value* indicated with $\hat{y}$, the parameters are split in a **bias**, indicated with b whose role is the one played by θ_0, while the θ_i, $i = 1, ..., n$ are the weights w_i

Analyzing the complete model, we have two degrees of freedom (DOF) since θ_0, θ_1 can vary. In this case the functional to be minimized has to be represented in a 3D space, then we obtain a surface similar to one presented in the following figure:

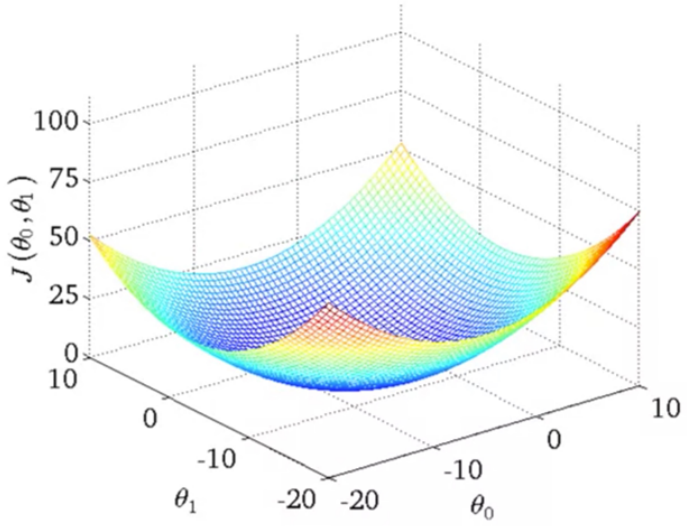
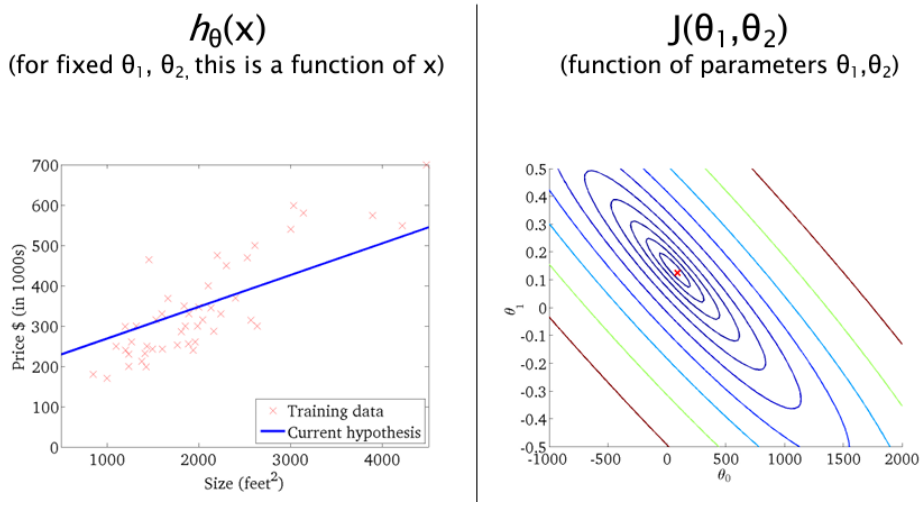


Figure 2.2: Example of $J(\theta_0, \theta_1)$

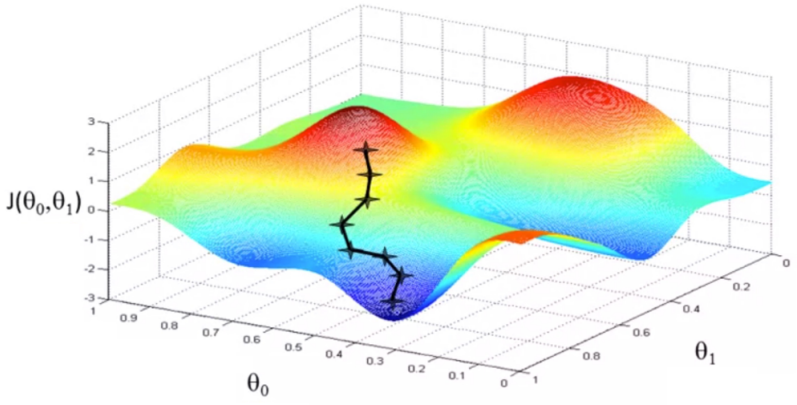
In the common case of bivariate minimization problem one can use *contour plot* which analyze the shape of the function at different heights. It is remarkable that points in the space (θ_0, θ_1) which are on the same *countour line* result in very different hypotesis. It is trivial to understand that, in this case the minimum $J(\theta_0, \theta_1)$ is attained on the bottom of such a *bowl-shaped* surface.



2.2 Parameter learning: Gradient descent

The objective here is to find a way to minimize a certain multivariate functional $J(\theta_1, ..., \theta_n)$, the idea is using some methods that iteratively bring us to the minimum according to a certain criteria. In this paragraph we analyse the **Gradient Descent** algorithm, the main idea here

is to start with some θ_0, θ_1[3], and keep changing them until J evaluated at those parameters could reach (hopefully) the minimum, in the gradient descent this change is made up on the basis of the direction dictated by the **gradient of the functional** computed at the current parameters value (from which the name). The algorithm for GD is simply as follows:



Algorithm 1 Gradient Descent

while !convergence **do**
$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ ▷ for j=0, j=1,...
end while

The := symbol is associated with a *simultaneous update*, note that if you put together for each j the partial derivatives of J you will obtain the gradient. The parameter α is called the **learning rate** and it must be properly chosen because:

- If α is **too small**, then the convergence to the minimum (within a certain tolerance) could be very slow;
- If α is **too large** the algorithm can overshoot the minimum either failing to converge, or diverging.

Even when the learning rate is fixed the GD can converge to a (local) minimum since we are moving toward *steep* directions which decrease the functional over time. If we apply the algorithm to the functional of the problem in (2.2) we obtain:

$$\begin{aligned} \theta_0 &= \theta_0 - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)} - y^{(i)})) \\ \theta_1 &= \theta_1 - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)} - y^{(i)}))x^{(i)} \end{aligned} \tag{2.4}$$

this is known as **batch gradient descent** since for each step we use all the training samples. There are cases in which the minimization is particularly 'simple'. This happens when the functional is convex in θ. Besides, for the class of convex functions a local minima is also a **global and only min**.

[3]They are chosen either randomly or $\theta_i = 0, \ \forall i$.

2.3 Multivariate linear regression

It is quite immediate to understand that the linear regression can be used also for a *multivariate context* in which the samples are characterized by many features. In this context the hypotesis becomes:

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + ... + \theta_n x_n = \theta^T x \tag{2.5}$$

In this case we have n parameters and associated features x_i, so that the functional J is function of n parameters, in this case the partial derivatives to compute, obviously, will increase. Note that a *fictitious* feature $x_0 = 1$ has been added with the purpose to employ a vector notation.[4]

2.4 Data mean normalization

Sometimes, before starting with the model construction, some preliminary operations are needed. For example, often it is better for the features being on a **similar scale**. In this case we replace in each sample for each feature $x_i = x_i/s_i$ where s_i can be either the range (max-min) for that feature or some index similar to variance/standard deviation.
Other times, one is supposed to normalize the data so that they can have a *zero mean*. The trick here is replacing $x_i = x_i - \mu_i$, where μ_i is the mean for the i-th feature. Not rarely, you can find the two transformation combined such thet

$$x_i = \frac{x_i - \mu_i}{s_i} \tag{2.6}$$

2.5 Debug of Gradient Descent algorithm

The *gradient algorithm* is clearly a descent method in the sense that – being k the k-th iteration – it holds that $J(\theta_{k+1}) < J(\theta_k)$, this is the same to state that the $J(\theta)$ function is required to be strictly decreasing. An **automatic convergence test** can be performed: for example the objective function J, had had a decreasing less than a certain threshold $\varepsilon = 10^{-3}$ (for example).
Whether the algorithm is not working well the value for the **hyperparameter** α must be changed (for example decreasing it). One way to choose *manually* α is by *trial-error*[5], choosing the α in a range and then plotting $J(\theta)$ as a function of the number of iterations.

2.6 Alternative to Gradient Descent

There are alternative methods to gradient descent, for example the normal equation method which is derived by the analytical solution of the well-known **least-squares** problem. In this case θ is found by solving the system (normal equations):

$$(X^T X)\theta = X^T y \tag{2.7}$$

[4]The great majority of tools and softwares which are used for machine learning exploit vector and matrices calculus to carry out their work.

[5]A more accurate method is the **backtracking line-search** which repeat some calculations until the so-called *Armijo condition* is not met; however it requires that additive hypotesis are made on the regularity of the objective and its gradient.

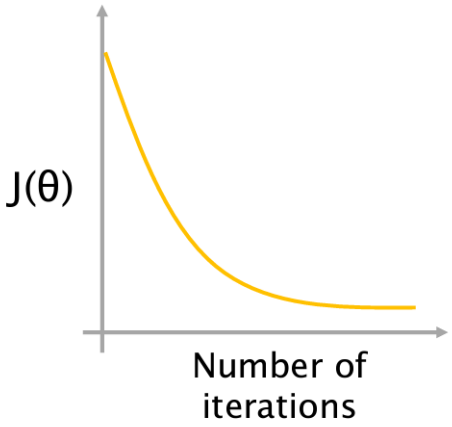


Figure 2.3: Desired behaviour for $J(\theta)$ vs # iterations

where the X matrix contains the dataset features and y is the vector with the "right answers". The solution of such a problem gives *one-shot* the solution without proceeding by step as in the case of gradient descent. The main limitation of such a method is the inversion of the matrix $X^T X$, which could be significantly slow if n (number of features and parameters) is very large.[6]

2.7 Polynomial regression

Not rarely can happen that a linear hypotesis is not satisfactory for our task of regression and so could be useful to introduce some other features by doing the so-called *handcrafting*. The derived features can be nonlinear, and specifically polynomial, combination of the available features. In the case of the price predition the handcrafted features could be for example the square of the size and the cube of the size.

[6]It is sufficient to think about the number of parameters involved in a problem of image classification. They are in a number of 3 billion for an RGB 1000×1000 image.

Chapter 3

Logistic Regression

The objective here is discussing the **Logistic regression** model which is used for **binary**, and also, **multiclass classification**. We will start from the hypotesis $h_\theta(x)$ used in the case of regression, we will analyze the aspects which will be maintained of it, and also the cons.

3.1 Classification vs Regression

Coming back for a while to the problem of *tumor classification*, suppose that a linear hypotesis can be used in order to separate the data. The comprehension of this concept is aided by the following figure: The hypotesis $h_\theta(x)$ used in order to separate the two class is the line showed

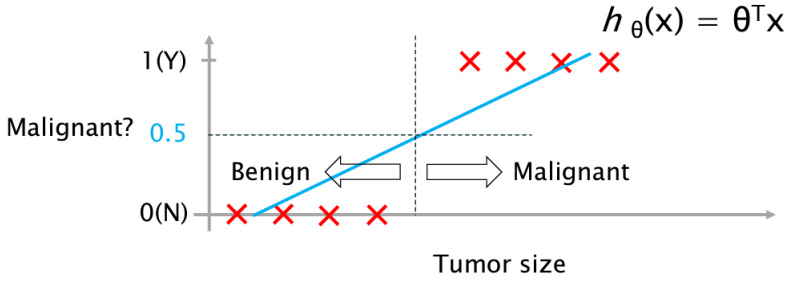


Figure 3.1: Classification with linear hypotesis

in blue. Since here we are not in the case of prediction of a continuous value, we have to find a way for shrink all possible output from the hypotesis in two values (Positive/Negative). At this point an idea could be using a threshold according which you can separate data from the two classes, that is:

$$y = \begin{cases} 1 & \text{if } h_\theta(x) > 0.5 \\ 0 & \text{if } h_\theta(x) \leq 0.5 \end{cases} \tag{3.1}$$

This appproach seems to work, until we do not change the data used for building the model. Let us consider, for example, the following scenario: It appears quite evident that one of the data for which we know that belongs to the positive class, is classified as negative.
This is not the only problem: we want that the predicted output[1], that is the hypotesis is between 0 and 1, despite the fact of using a threshold this is not satisfied, since a linear function is *unbounded*. At this point our reasoning leads to the formulation of the following two issues:

[1]Later we will call it $\hat{y}$, for comparing it to the true output y.

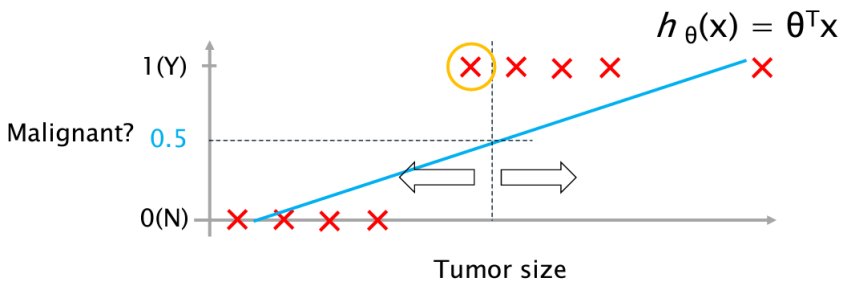


Figure 3.2: Effect of changing the dataset

1. A *linear hypotesis* is not suitable for a classification problem, the performances would be awful also on the training set;
2. It is required that
$$0 \leq h_\theta(x) \leq 1$$
this is not happening in the cases a linear hypotesis is employed.

These are the main points that leads to the *correct formulation* of **logistic regression**[2].

3.2 Logistic regression

The main concept behind *logistic regression* is using a nonlinear function that saturates the hypotesis between 0 and 1. Basically we have to apply such a function which we will call $g(z)$ to the linear $\theta^T x$, such a function is called **sigmoid** or **logistic function**. It is depicted in the following and its expression is:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

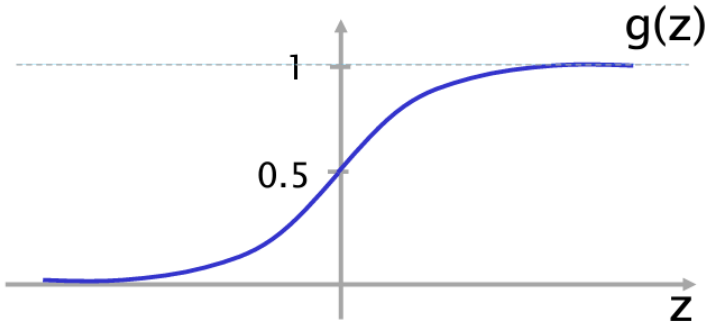


Figure 3.3: The logistic function $g(z)$

The output of such an hypotesis can be interpreted as *the probability that the output y=1 on a given input x*, for example in the case of tumor classification a value of 0.7 of the hypotesis results in a prediction that for 70% the given tumor (with its feature is malignant). In order to be mathematically formal we can say that

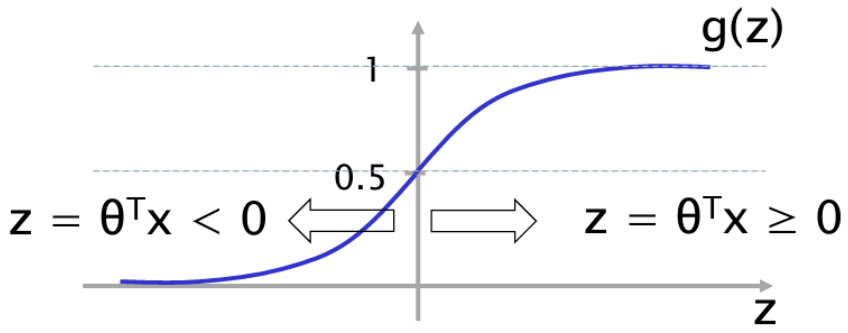
$$h_\theta(x) = P(y = 1|x;\ \theta) \tag{3.3}$$

[2]The name *logistic* is related to the fact that we are solving a dicotomic/binary classification problem.

which is read "the probability for the output y to be 1, given x, parametrized by θ. Clearly, at the opposite we can compute

$$P(y = 0|x;\ \theta) = 1 - P(y = 1|x;\ \theta) \tag{3.4}$$

Here we can stick to the fact of having a threshold. More specifically, we use as an hypotesis $g(\theta^T x)$ and we can use the criterion used in (3.1). Moreover for the particular function $g(z)$ it is used we can say that, given the features x then it is classified as positive if $\theta^T x \geq 0$ or negative if $\theta^T x < 0$, since the counterimage of 0.5 through $g(z)$ is equal to 0, as showed in the following figure.



It appears clear that the linear combination $\theta^T x$ is a (linear) **decision boundary** since it provides us with the information of having a positive or negative record. It is useful to give an example of this fact:

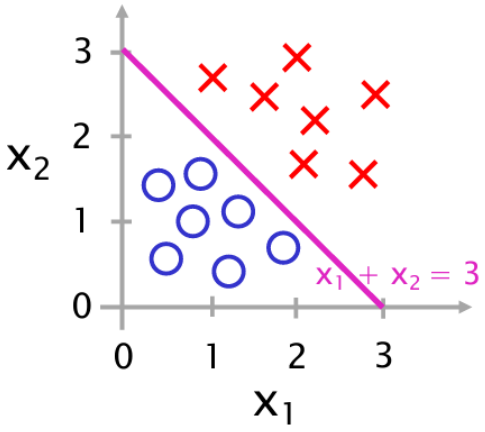


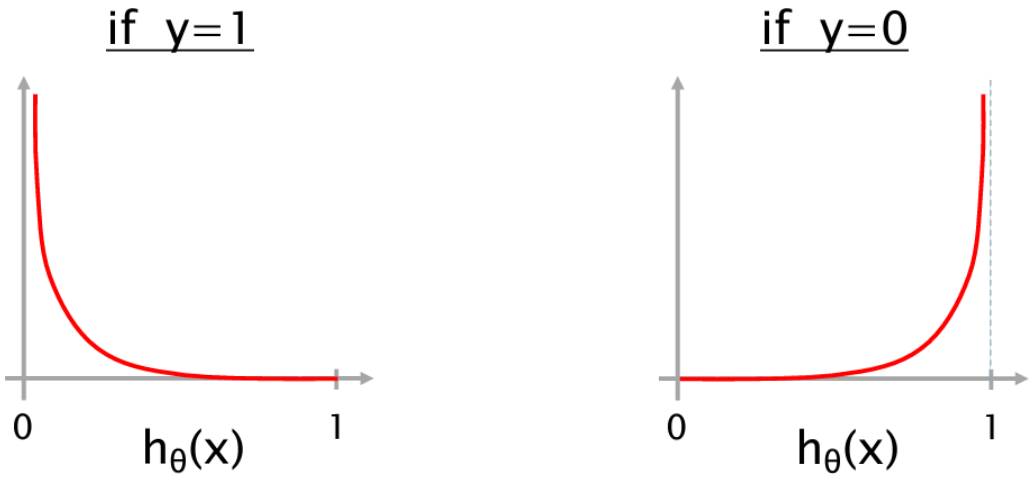
Figure 3.4: Decision boundary

Suppose we trained the model and the parameter theta resulted in being $\theta = [-3\ 1\ 1]^T$, this is the same to say that we assign *positive class*to records with feature x_1, x_2 such that $\theta^T x$ is greater or equal than 0, negative class otherwise. Then, I can completely remove the record from the dataset and using such a decision boundary for doing classification. There are some cases in which due to the distribution of the data of each class, it is not possible to separate them with a linear decision boundary. In that case higher order nonlinear functions (eg. polynomials) must be used.

3.3 Cost function for logistic regression

We have seen in the former chapter about linear regression that a cost function is introduced to be minimized in order to find the parameter θ which are the best for our model.
In case we had a regression problem to be solved the functional $J(\theta)$ (Square Error) was a convex one, if we stick to the use of a sigmoidal function (and it is a proper choice for classification) the *Square Error functional* becomes a non-convex one, so that the Gradient Descent Algorithm is not converging to a global minimum. What is changed for logistic regression is the **Loss function** associated to a single training sample. A particularly clever choice is the following:

$$\text{Loss}(h_\theta(x), y) = \begin{cases} -\log(h_\theta) & \text{if } y = 1 \\ -\log(1 - h(\theta)) & \text{if } y = 0 \end{cases} \tag{3.5}$$



Since the Loss function must penalize the objective to be effective in case the effective output is $y = 1$ and the hypotesis (formulated with those parameters) would give 0, then the cost is very high. On the contrary a positive hypotesis with an actual output of $y = 0$ will give to the functional a very high contribution, resulting in an high penalization (keep in mind that the functional must be minimized). This concept has a quite intuitive explanation as we have seen. It would be useful having an *overall functional* and with the aim of obtaining it, we badly exploit the fact that the output is logistic. In particular:

$$\text{Loss}(h(\theta), y) = -y \log h_\theta(x) - (1 - y) \log (1 - h_\theta(x)) \tag{3.6}$$

The cost function coming from such a loss function is the following and it is denoted as **Binary cross-entropy cost function**:

$$J(\theta) = \underbrace{-\frac{1}{m} \sum_{i=1}^{m} \left[- y^{(i)} \log h_\theta(x^{(i)}) - (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)})) \right]}_{\text{BINARY CROSS-ENTROPY COST FUNCTION}} \tag{3.7}$$

3.4 Training logistic regression

Given the cost function (3.7), we minimize it by using gradient descent algorithm, given an unknown x the output is provided by using the hypotesis

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^\intercal x}} \tag{3.8}$$

The algorithm of gradient descent follows the same step as in *linear regression*, what is changed is the shape of the hypotesis $h_\theta(x)$.

3.5 Multiclass classification

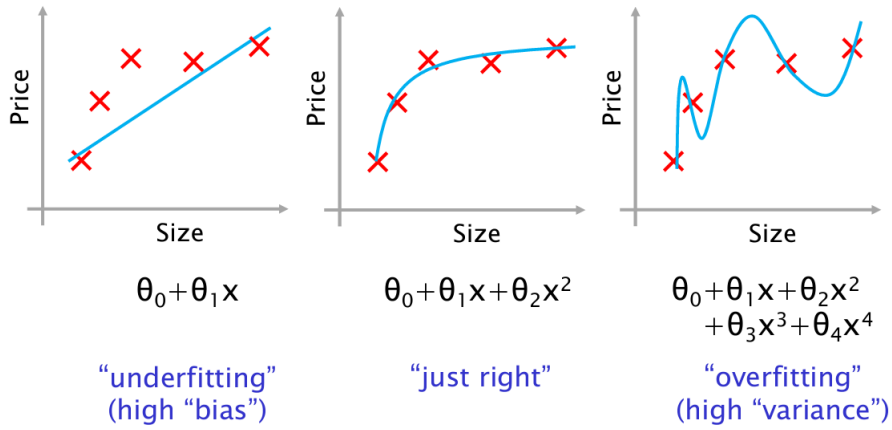
Suppose we want to do a multiclass classification, for example for tagging mail as **SPAM, WORK, FRIENDS...** Can we use logistic regression in order to carry out such a task? The answer is YES, but with some modifications. In the sense that we can reformulate the problem in *one class against the others*. The steps are the following: (A) We train a logistic regression classifier $h_\theta^{(i)}(x)$ in order to predict $P(y = i|x;\ \theta)$ for each class i. On a new input the prediction is done as follows:

$$i = \arg\max_i h_\theta^{(i)}(x) \tag{3.9}$$

where i is the i-th class. Is this a good idea for a multiclass classification? Not so much! In the sense that the computational load grows of a factor n, with n the number of classes.

3.6 Overfitting and Regularization

In building a predictive model, there are usually two problems which we want to avoid: **underfitting and overfitting**. In the former case, provided that there are a small number of features, the learned hypotesis will not fit properly the training set. We can also say that there is an **high bias** in the data. In the latter case (overfitting) the learnt model has got excellent performances on the training set, in the limit case $J(\theta) = 0$, but *fails to generalize new examples*, in this case there are *too many features*, so there is an *high variance* in the data.
The configuration which is in the middle is the so-called *just right*, and it is the one we want to reach aiming to have a good model.



What are the solutions for avoiding overfitting? The first way is to reduce theb number of features the algorithm use for building the model (some techniques as PCA[3] and LDA[4] can be used). Another way is introducing in the cost function a **regularization term**, its main purpose is to reduce the magnitude of the θ_i while keeping all the features. The regularization term acts directly on the parameters and it is proportional to the number of features. Using the regularization, simpler models can be obtained reducing the problem of overfitting the

[3]PCA → Principal Component Analysis
[4]LDA→ Linear Discriminant Analysis

data. Let us consider for example the *Square-error cost function*, when regularization is used it becomes:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2 \tag{3.10}$$

It is remarkable that the square-error cost function works on the m training examples, while the regularization term[5]interests directly the parameter of the model. The parameter λ becomes another hyperparameter and we refer to it as the *regularization parameter*. Clearly the optimization algorithm (Gradient Descent) must be updated accordingly. We also call the regularization in (3.10) the ℓ_2-regularization since it involves the ℓ_2-norm definition. In the field of optimization models also the ℓ_1-norm is considered, but in this case alternative techniques must be employed since the ℓ_1-regularization makes the functional a non-differentiable one. Different algorithm, like ISTA and FISTA, have been developed in order to deal with such a type of optimization problem.

[5]Do not confuse yourself with the *normalization* which is done on the data

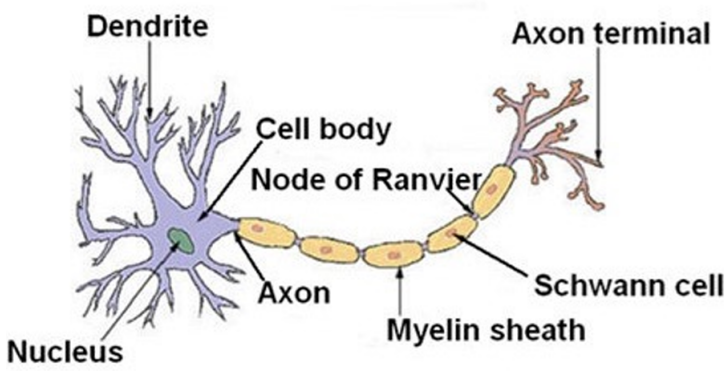
Chapter 4

Neural Networks: an introduction

The idea of **neureal network (NN)** was introduced in the late 50s, in order to implement algorithm which could try to mimic the brain functionalities. They were very used in 80s, but their popularity decreased in the late 90s. Two determinant factors have aided the resurgence of such technologies: the increasing in the quantity of data, and the increasing in computation capacity. For example, Neural Networks are used, among the others tasks, for performing classification. We have understood that a linear hypotesis is not suitable for such a task, then nonlinearity is needed.
Now, let us imagine we want to build a model for classifying in a binary way, some images in two classes: CAR, NO CAR. An image is in general composed of pixels. Can we use *logistic regression* for doing image classification? Let us consider a training set made up of 64×64 images, then 4096 pixels which is the same number for the parameter if we have a gray-scale image. If we have RGB images, the number of parameters grows by a 3 factor, that is a number of parameters equal to $n = 12288$, a huge number that makes highly unsuitable the logistic regression models. In this situation a neural network of some type is used.

4.1 Ideas underlying neural networks

Before going on through the discussion of (Artificial) Neural Networks, we have to just mention how a biological neuron is made.



Three are the main components of a neuron:

1. Some inputs wires (**Dendrites**)
2. A **Nucleus** which is the *computational unit*
3. An output wire (**Axon**)

Clearly such a type of cells are connected each other by mean of *synapses* realized by neutrotransmitters.
The **artificial neural newtork** has exactly the same structure of a biological one whose building blocks are some *artificial neurons* made up of the same three basic elements, since it has got: (i) some inputs which are the features, (ii) a computation unit that performs a weighted sum of the features, (iii) an output which is the result of an **activation function** which often can be a sigmoid. [From this point we can see the strong relationship with the logistic regression.] Other activation functions can be used, besides another example is the **ReLU**.

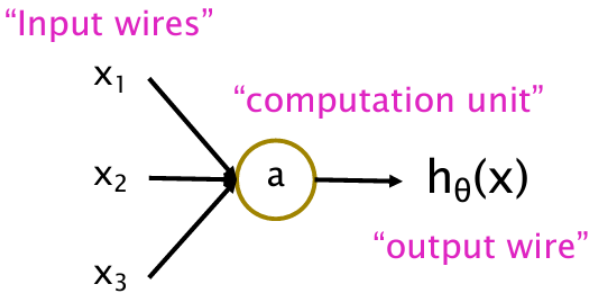


Figure 4.1: Structure of an artificial neuron

4.1.1 Notation

In the following some notation is introduced that will be used in the remaining part of the course dealing with neural networks. When we put together some artificial neurons, what we obtained in general is a **multilayer perceptron** or *classical NN*. The *layer 0* is the input layer, the following are numbered as first, second layer and so on. In each layer we can find one or more computational units. For example the first layer of the showed NN has 3 computational units (without considering the unit 0 which is associated to the bias). The notation $a_i^{[j]}$ indicates the **activation** in the **i-th unit** in the **j-th level** of the network, while $\Theta^{[j]}$ is the weighting matrix for the j-th layer of the network.[1] Usually also the input are renamed as *zero-layer activation*, then are indicated with $a_i^{[0]}$. For the presented MLP[2] let us try to write all of the activation of each layer:

$$
\begin{aligned}
a_1^{[1]} &= g(\Theta_{10}^{[1]}a_0^{[0]} + \Theta_{11}^{[1]}a_1^{[0]} + \Theta_{12}^{[1]}a_2^{[0]} + \Theta_{13}^{[1]}a_3^{[0]}) = g(z_1^{[1]}) \\
a_2^{[1]} &= g(\Theta_{20}^{[1]}a_0^{[0]} + \Theta_{21}^{[1]}a_1^{[0]} + \Theta_{22}^{[1]}a_2^{[1]} + \Theta_{23}^{[1]}a_3^{[0]}) = g(z_2^{[1]}) \\
a_3^{[1]} &= g(\Theta_{30}^{[1]}a_0^{[0]} + \Theta_{31}^{[1]}a_1^{[0]} + \Theta_{32}^{[1]}a_2^{[0]} + \Theta_{33}^{[1]}a_3^{[0]}) = g(z_3^{[1]}) \\
a_1^{[2]} &= g(\Theta_{10}^{[2]}a_0^{[1]} + \Theta_{11}^{[2]}a_1^{[1]} + \Theta_{12}^{[2]}a_2^{[1]} + \Theta_{13}^{[2]}a_3^{[1]}) = g(z_1^{[2]})
\end{aligned}
$$

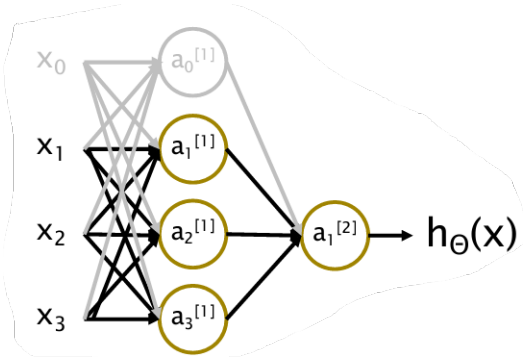
In this case the weighting matrices are for the first layer $\Theta^{[1]} \in \mathbb{R}^{3,4}$, for the second layer is $\Theta^{[2]} \in \mathbb{R}^{1,4}$.

[1] The intermediate layers of a neural network are called **hidden layers** since the output produced is hidden and are generated by linear and non linear combination of the features.

[2] multilayer perceptron

4.1.2 Different types of NN for different types of purposes

A neural network can be used either for binary or multiclass classification. In the former case the last layer has got one neuron whose activation is 0 or 1, in the latter case we have a neuron for each class in a level that is (how we will see) the *softmax layer*. Moreover a neural network is said to be **shallow** (typically) when it is made up of a number of layers which is less than seven, otherwise we have a **deep neural network**.



4.2 Logistic regression as a NN

If we better analyse the structure of a neuron and the path going from the input to the output of it, we will discover that it is something very similar to what we have seen in the case of logistic regression, where we have combined a linear part with a non linear one in order to correctly perform the (binary) classification task.
Now, let us suppose we want to perform a binary classification a set of images, in particular we want distinguish when they are cars or not. Can we use logistic regression in order to perform such a task? NO! It could be very slow and inefficient: a logistic regression (that is a single neuron) cannot perform in a good way such a task, then a neural network is used in this case. The next step is: how can we represent an image as a vector of features? We know that a gray-scale image can be represented as a matrix of numbers in the range [0-255], if the image is RGB we have a different matrix for each one of the channel R, G and B. We can turn the matrix into a vector by simply unrolling it row by row, in a way that each single pixel of each one of the channel is a feature for our classification problem. This example was just to present the problem of **vectorization**, that in the field of neural network is a very common operation which is done on the data in order to make them suitable for network itself.

We have seen in the logistic regression that our **predicted value** $\hat{y}$ (which was the hypotesis), is nothing but the result of a sigmoidal function applied to the linear combination $w^T x + b$, where w are the weights, b is the bias while x is the feature vector. Then we have that:

$$\hat{y} = a = \sigma(w^T x + b) \tag{4.1}$$

How we mentioned, this is exactly the work performed by a neuron from the inputs to the output. Furthermore, we have seen that for the logistic regression a different cost function must be considered in order not to dealing with *non-convex optimization problem*. For the description of the logistic regression as a single-neuron NN, nothing change a part from few differences in the notation. In fact we indicate the hypotesis $h_\theta(x)$, with the *predicted value* $\hat{y}$, while the θ_i parameters are split in weights w and a single bias b. Another difference we can

find in the field of NN, is that the partial derivatives of the functional J to be minimized with respect to the weights/bias, that is

$$\frac{\partial J}{\partial w}, \quad \frac{\partial J}{\partial b} \tag{4.2}$$

are denoted simply with dw and db, in order to make lighter the notation. The *gradient descent step* in order to decrease the functional becomes:

$$w = w - \alpha\ dw$$
$$b = b - \alpha\ db$$

Now, **How can we compute the partial derivatives?** For sure, we can say that no explicit analytic calculation are performed, instead a very powerful tool that is the **automatic differentiation** leveraging on the so-called **computation graph** is used. The main concept behind it is to express a function by using *intermediate auxiliary variables* and computing the derivatives by using the **Leibnitz's Chain Rule**.

4.3 Automatic differentiation and computation graph

Suppose we have a function

$$J(a, b, c) = 3(a + bc) \tag{4.3}$$

we want to compute the ppartial derivatives of J with respect to the three variables a, b, c. We can introduce some intermediate variables which can be called

$$u = bc \quad v = a + u$$

Our function J becomes: $J = 3v$. In this way we have split the original function in trhee different simpler functions. This procedure can be graphically represented as shown in the figure:

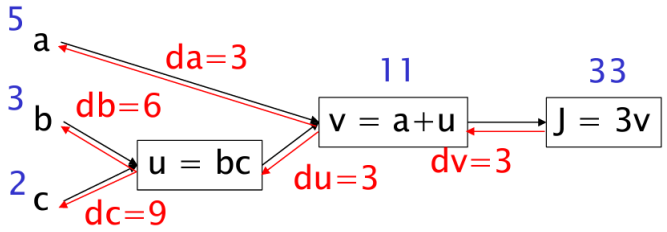
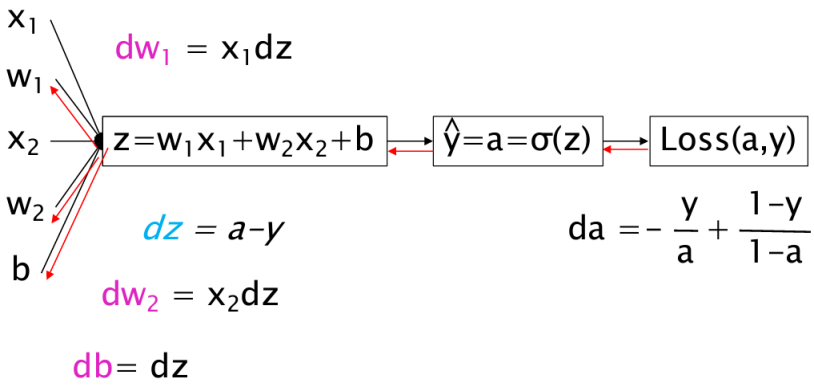


Figure 4.2: Computation graph for $J = 3(a + bc)$

In particular from the input a, b, c, we can compute the value of the intermediate variable u, which can be used for obtaining v, finally we can compute J. This path from $(a, b, c) \to J$ is called **Forward Propagation**. Now, what about the partial derivatives? We can proceed step by step, doing an inverse path from $J \to (a, b, c)$, intuitively such a path is the **Backward propagation**, here the *Chain rule* is used in order to carry out 'gradually' the computation of the partial derivatives. The following steps are done:

$$\frac{\partial J}{\partial v} \doteq \mathbf{dv} = \mathbf{3} \quad \frac{\partial J}{\partial u} \doteq \mathbf{du} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u} = 3 \cdot 1 = \mathbf{3}$$
$$\frac{\partial J}{\partial a} \doteq \mathbf{da} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial a} = \mathbf{3} \quad \frac{\partial J}{\partial b} \doteq \mathbf{db} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial b} = 3 \cdot 1 \cdot c = 3c = \mathbf{6}$$
$$\frac{\partial J}{\partial c} \doteq \mathbf{dc} = \frac{\partial J}{\partial v}\frac{\partial v}{\partial u}\frac{\partial u}{\partial c} = 3 \cdot b = \mathbf{9}$$

The procedure we have just shown is THE way in which are computed the derivatives in the field of Neural Network. Clearly, the same reasoning we have done for the functional (4.3) can be repeated for the *loss function* used for the case of logistic regression. The figure below shows the final result of forward and backward propagation applied for the **cross-entropy loss function**.



4.3.1 Feedback and Backward propagation for Logistic Regression

For a single training sample we have that the feedback and backward propagation mathematical steps are the following:

FORWARD PROPAGATION

$$z_i = w^T x^{(i)} + b \quad \text{(linear part)}$$
$$\hat{y}_i = a_i = \sigma(z_i) \quad \text{(activation)}$$
$$J_i = -[y_i \log a_i + (1 - y_i) \log(1 - a_i)]$$
(cost function)

BACKWARD PROPAGATION

$$dz_i = a_i - y_i$$
$$dw_i = x^{(i)} dz_i$$
$$db_i = dz_i$$

Whether we extend such computations to the whole dataset, we have matrices and vectors instead of vectors and scalars. In particular:

FORWARD PROPAGATION

$$z = w^T X + b \quad \text{(linear part)}$$
$$\hat{y} = a = \sigma(z) \quad \text{(activation)}$$
$$J_i = -1/m \sum [y \log a + (1 - y) \log(1 - a)]$$
(cost function)

BACKWARD PROPAGATION

$$dz = a - y$$
$$dw = \frac{1}{m}(X^T dz^T)\mathbf{1}$$
$$db = \frac{1}{m}(\mathbf{1}^T dz)$$

Where $\mathbf{1}$ is a column vector with all ones. After having performed the forward and backward steps, both weights and bias must be updated as follows:

$$w := w - \alpha \cdot dw \quad b := b - \alpha \cdot db \tag{4.4}$$

Such steps must be repeated until convergence (in some sense). Keep always in mind that dw and db are the partial derivatives of the cost function with respect to the weights and bias.

4.4 Training Neural Networks

Till now we have seen the optimization procedure of the *logistic regression* as a single neuron. However, we know that a neural network is made up of several layers which in turn are composed of several neurons (computational units). The objective here is to understand how we can generalize the **optimization procedure** in the case when the whole neural network must be trained (in particular the parameters for each neuron, for each layer must be computed).

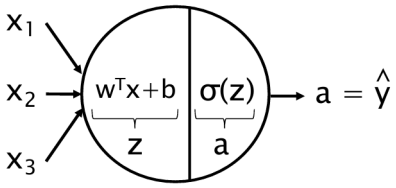
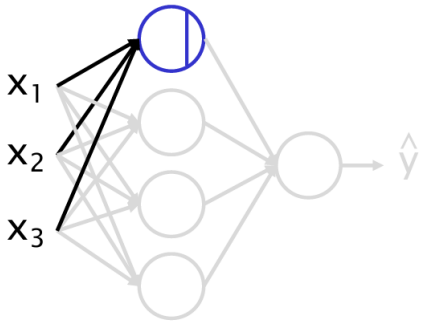


Figure 4.3: Single neuron in form of logistic regression

We are going to proceed step by step starting from a single neuron, going to the whole layer analyzing both the forward and backward propagations aimed to generalize the gradient descent algorithm to the whole network. For sake of simplicity but without loss of generality, the analysis has been conducted on a 2-layer NN. In the following the activation function will be indicated with g in order to generalize to the use of different functions which can be used.

4.4.1 Forward propagation

Single neuron, single sample

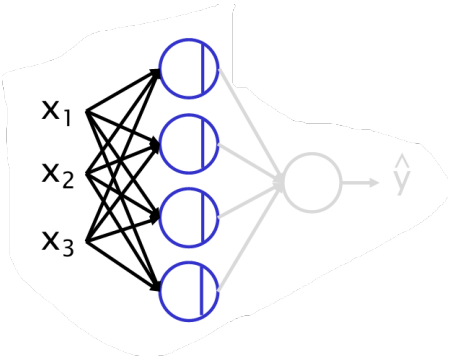


A single neuron of the layer is considered, which clearly has his own weights and bias.

$$\begin{aligned} z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]} \\ a_1^{[1]} &= g(z_1^{[1]}) \end{aligned} \tag{4.5}$$

The function g can be something different with respect to a sigmoid. Only one sample x of the dataset is considered.

Single Layer, single sample



Considering all the neurons in the first layer, we have the (4.5) repeated four times, that is:

$$\begin{aligned} z_i^{[1]} &= w_i^{[1]T} a^{[0]} + b_i^{[1]} \\ a_i^{[1]} &= g(z_i^{[1]}) \end{aligned} \quad , \quad i = 1, ..., 4$$

Here the only difference is that we have indicated in terms of activations the input. In a more compact form we can say that:

$$\begin{aligned} z^{[l]} &= W^{[l]} a^{[l-1]} + b^{[l]} \\ a^{[l]} &= g(z^{[l]}) \end{aligned} \tag{4.6}$$

where l indicates the l-th layer of the network.

Single layer, whole dataset

Till now we have seen the *forward step* for a single neuron and for a layer of the network. What if considering the whole dataset instead of a single sample? The vector z of the linear combination becomes a matrix in which the i**-th row** contains the linear combination for the j**-th** sample of the activation of the past layer. We have:

$$\begin{aligned} Z^{[l]} &= W^{[l]} A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}(Z^{[l]}) \end{aligned} \tag{4.7}$$

Here is noticeable that the activation function can be different for each level of the network. This is the reason why we put the l superposed also on the g.

4.4.2 Backward propagation

For the case of backward propagation we give directly the result in the most general case in which the loss function is even different than the *cross-entropy*. Then, we have:

$$\begin{aligned} dZ^{[l]} &= dA^{[l]} .* g^{[l]'}\ Z^{[l]} \\ dW^{[l]} &= \frac{1}{m}(dZ^{[l]} A^{[l-1]T}) \\ db^{[l]} &= \frac{1}{m}\text{sum}(dZ^{[l]}) = \frac{1}{m} dZ^{[l]} \mathbf{1} \\ dA^{[l-1]} &= W^{[l]T} dZ^{[l]} \end{aligned} \tag{4.8}$$

For the output layer $dA^{[L]} = A^{[L]}$. The gradient step must be performed for each layer once all the derivatives have been computed:

$$W^{[l]} := W^{[l]} - \alpha \cdot dW^{[l]} \quad b^{[l]} := b^{[l]} - \alpha \cdot db^{[l-1]} \tag{4.9}$$

The whole procedure of forward and backward propagation can be schematically represented by using a block diagram:

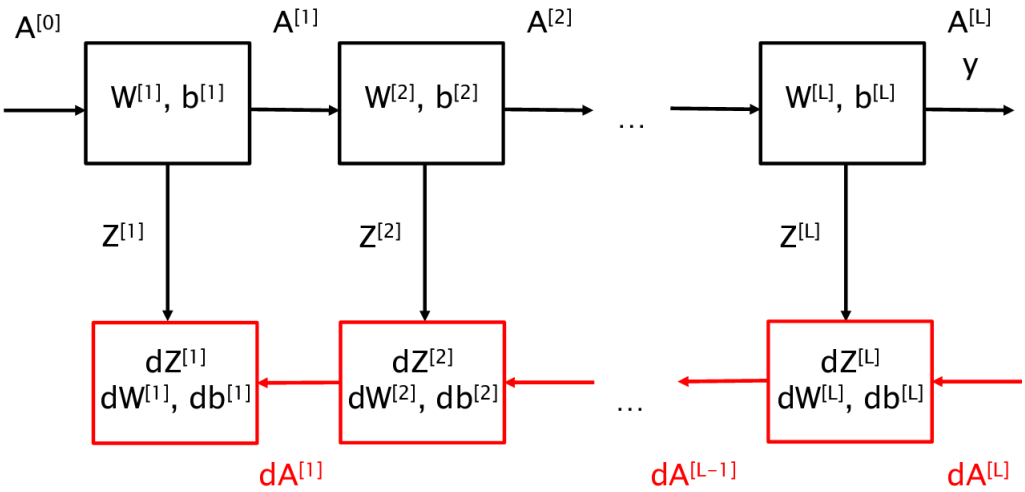
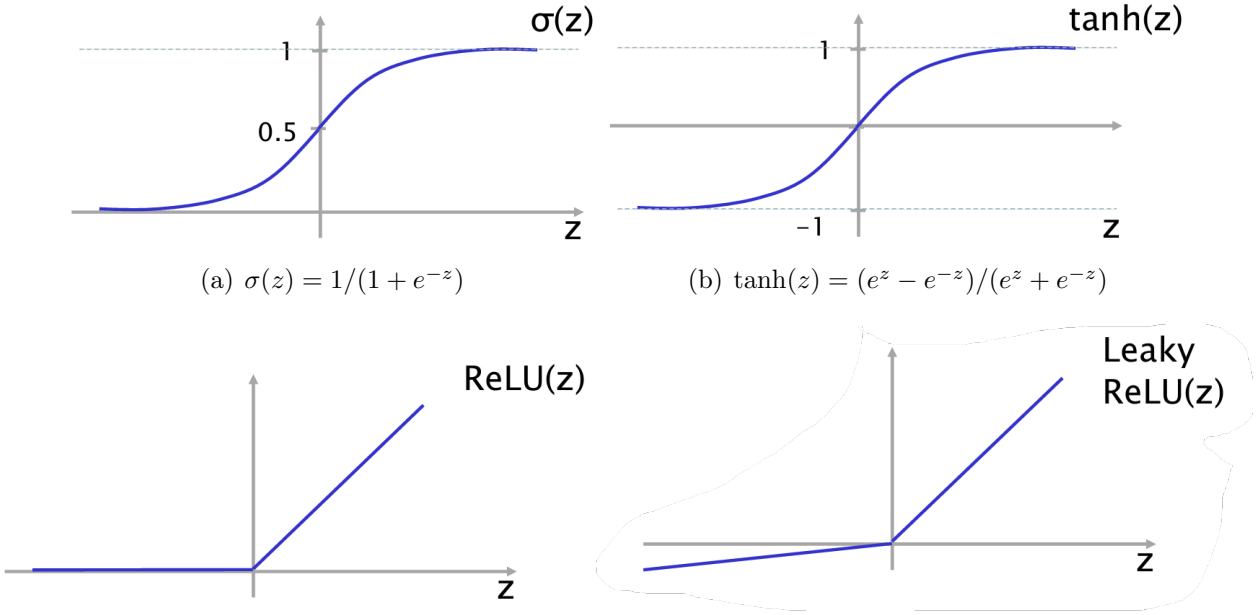


Figure 4.4: Block diagram for the GD

4.5 Activation functions

At different layers of a neural network, different activation functions can be used. Till now we have seen the sigmoid, since it is the one arising in the case of *logistic regression*. Other choices can be done according to the needed convergence property and the task to be performed. The most common used activation functions are reported in the figure below (in the description there is also the definition).



(a) $\sigma(z) = 1/(1+e^{-z})$

(b) $\tanh(z) = (e^{z} - e^{-z})/(e^{z} + e^{-z})$

(c) $\mathrm{ReLU}(z) = \max(0, z)$

(d) $LeakyReLU(z) = max(\varphi z, z)$

Figure 4.5: Some activation functions

Note that in the $LeakyReLU(z)$ there is a parameter φ, this is in order to deal with the fact that the derivatives of $ReLU(z)$ for $z < 0$ is equal to zero. Such an ε becomes an hyperparameter.

4.6 Initialization of the parameters

When we have seen the *linear regression* and the *logistic regression*, we have said that the parameters θ_i would have been initialized for example to zero. This does not work in the case of neural networks. It has been demonstrate that the **zero-Initialization** of the weights leads to a problem of **Symmetric weights**, that is after each update, parameters associated to the inputs going to the next hidden layer unit are *identical*. One possible solution, at least for simple NN, is to iniatialize randomly the weights $W^{[l]}$ and biases $b^{[l]}$ with numbers in the interval $[-\epsilon, \epsilon]$. Indeed, more sophisticated approaches are needed in the case of deep neural networks.

4.7 Training a neural network (Recipe)

Once we have presented the main issues related to neural networks and their training, we are ready to give a list of steps aimed to the training:

0. Pick a network structure, fix the numer of input and output unit with respect to number of inputs and number of classes respectively; the *number of hidden layers* can be decided

at this stage and also the number of neurons for each one of such layers. This are all **hyperparameters**.

1. Randomly initialize the weights
2. Implement *forward propagation* in order to get the estimate $\hat{y}^{(i)}$ for any $x^{(i)}$;
3. Implement code to compute the cost function $J(w, b)$ (this is another choice that we have done according to the task to be performed);
4. Implement **backward propagation** to compute partial derivatives (of the functional wrt the parameters);
5. Use **gradient descent** or other optimization methods together with backward propagation to try to minimize $J(w, b)$.

4.8 Hyperparameters

In the field of neural networks is fundamental the distinction between **parameters** and **hyperparameters**. The former ones are the ouput of the training phase, they are those that characterize a model from another. The latter ones are related to the choices that the *machine learning engineer* makes in order to improve the performances of the predictive model. Examples of hyperparameters are:

- Learning rate α of the gradient descent;
- Number of iterations of the optimization algorithm;
- Number of hidden layers and for each one the number of computational units;
- Activation functions for each layer and related hyperparameters (eg. in the Leaky ReLU there is φ to choose)

The **optimal (in some sense) configuration** must be found.

4.9 Training, Development, Test sets

When we want to build a machine learning model, we must have a **dataset**. Clearly, since the optimal configuration of the network has to be found, only a portion of this data is used for the training phase. In general the dataset is divided into three portions:

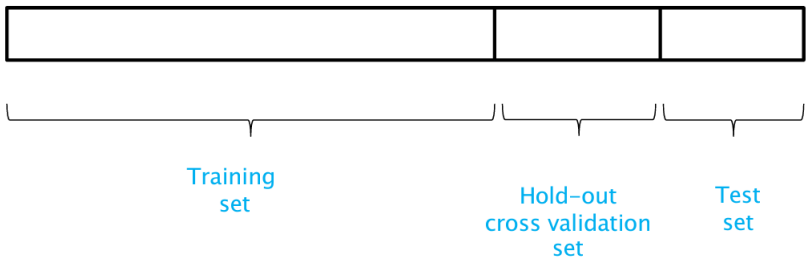


Figure 4.6: Traditional dataset partitioning

In the past, when not too much data was available the ratios were 60%, 20%, 20%; however with the increasing of the data availability the trend is using as much data as possible for the training set (98%) and the remaining part for development and test (1% for each one).

Training set Is the biggest part of the dataset which is used for building the model (obtaining parameters). On this set could be necessary preliminary operations of *data preparation* (eg. normalization, data augmentation and so on).

Cross-Validation/Development set Is the set used for evaluating different models (not necessarily different architectures, but also different hyperparameters). The data distribution should match with the one in the training set. The validation set which is used, clearly is the same for all the selected models.

Test set Once the model has been chosen a test on data that the network has never seen should be done. Such data can be extracted from the dataset, or coming from other sources making optional the presence of this type of set.

Once having defined how it is split the dataset, the **model selection** takes place, performing the following procedure. Given different models:

- I minimize the cost function on the training set finding the parameters for each of them;
- Compute the cross-validation error (on the validation set) using the output parameters for that model;
- Chosse the model with the lowest error, then evaluate the **generalization performances** (using the *test set*).

Chapter 5

Evaluating learning algorithm

5.1 Underfitting and overfitting data

Once the model has been built, we are interested in detecting wheter out model is affected by either *high bias* or *high variance*. In order to better formalize such a concept is interesting to analyze the graph below:

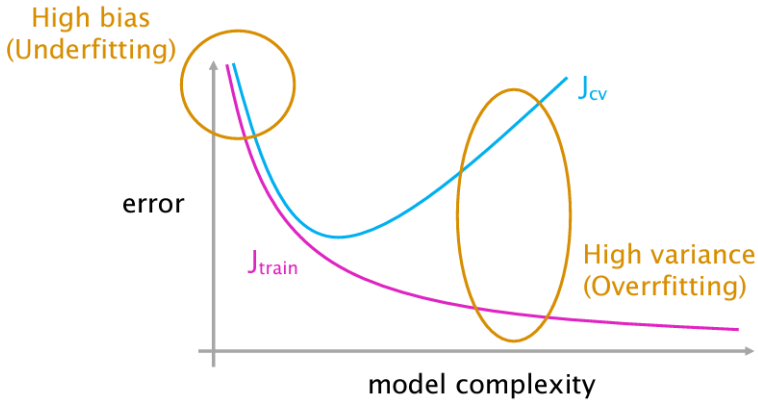
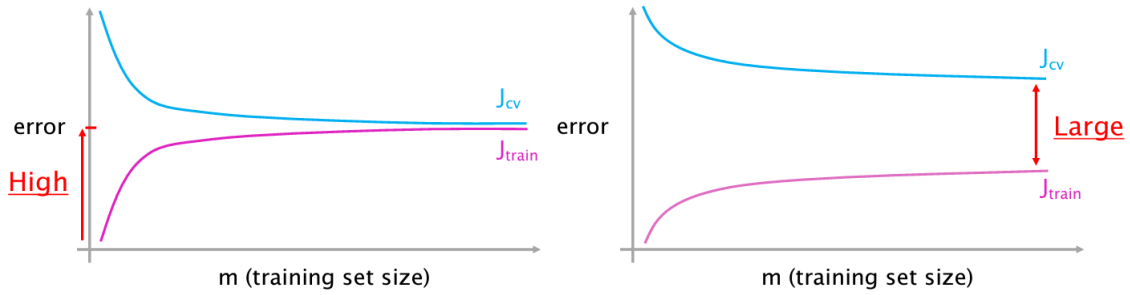


Figure 5.1: High-bias vs High-variance

Only for diagnostic purposes, we collect the data related to the error that the model does on the training data. Clearly the cost function J_{train} is very high with a very simple model because the network has not seen a sufficient amount of data. The error on the training set becomes smaller and smaller with the increasing complexity (number of features and so number of parameters of the model). What is very interesting to observe is what happen to the cost function when the validation data are used and the model complexity is growing. At start with a very simple model we have similar cost function, in fact J_{cv} is very similar to J_{train}. This situation arises when the model is too simple and the model is **underfitting the data**. At the opposite when the model complexity grows there is a big difference between the two cost functions. This is related to the fact that the model has very bad performances with never seen data. In this situation we are in front of a problem of **overfitting the data**: the model has learnt by heart the data, but it is not able to generalize.
Both situations must be avoided, as they make a model unusable! The same reasoning can be done by a *different perspective*, that is analyzing what happen to the J_* in function of the dataset size. I have an *high-bias* if at the end of the training phase I have a big error (*with respect to the human error*). On the other hand, I have an *high-variance* if the model has bad

performances on new data, or differently said, the two errors are very different. Note that, both high-variance and bias can be present in some situation. In particular in all cases when the error on training data is high and this is at the same time distant from the cross-validation error.



(a) High-bias (b) High-variance

5.2 Metrics for model evaluation

Motivation

Given a model which makes a *cancer classification*, suppose we want to evaluate its performance by using the so-called **accuracy**, we get a 1% error on the validation/test set. From the labeled data, furthermore, can be analyzed for example that among all the patients only the 0.5% has cancer. In this case if we take a *Naive classifier* that ignoring the output predicts always $y = 0$ (no cancer), such a classifier has better performances than the one we have properly built. The accuracy is not a good metric for evaluating the performance of a machine learning model. In this case the problem appear very evident since the data distribution is **skewed**. Conclusion: the introduction of other metrics is needed.

5.2.1 Confusion matrix and Precision/Recall

Especially for classification tasks is useful building a matrix which compare *actual and predicted* values, defining the true/false positive/negative. The one reported below is the so-called **confusion matrix**:

		Actual value 1	Actual value 0
Predicted value	1	True positive	False positive
	0	False negative	True negative

Based on the data collected in such a matrix, we can compute two different metrics: **precision** and **recall**. The former answers to the question: "Of all patient we predicted $y = 1$, what

portion has actually cancer?", the latter "Of all patients where we predicted $y = 1$, what portion we did we correctly estimate?". In formulas:

$$\text{Precision}(p) = \frac{TP}{TP + FP} \quad \text{Recall}(r) = \frac{TP}{TP + FN} \tag{5.1}$$

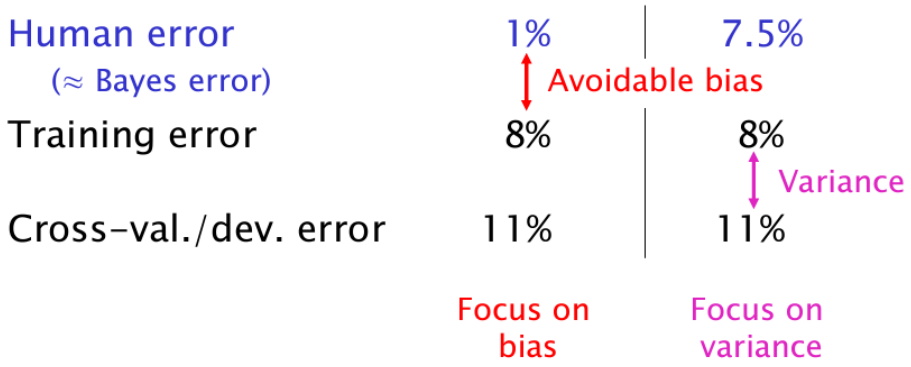
In order to compare such metrics, another auxiliary index is introduced, the F Score which is the armonic mean between recall and precision:

$$F\text{-score} = \frac{pr}{p + r} \tag{5.2}$$

Other metrics can be used, for example the **average of the diffenent accuracy indexes** in some situation can make sense, in other different situations also *handcrafted* metrics can be used. It is remarkable that whether we want use eterogeneous metrics it is adviceable to maximize/minimize a single index while having the others as constraints (eg. *maximize Accuracy subject to Running time*≤ 100 *ns*)

5.3 Human-level performance

Sometimes can be useful what is the error that a human do in a classification task in order to understand on what to put the focus (ie. High-bias or high-variance or both), moreover other statistical error-rate can be computed as the **Bayes Error** which is the **lowest possible error-rate** for a given classifier. The *Bayes Error* in some situation can be higher with respect to the *Human-error*, this becuse by properly training a neural network the model can have the experience of several humans. Let us give an example:



In the first case we can note that there is a bigger difference between the Human and Training error (here is assumed to be very similar to the Bayes error) than the one between training and validation error → we have to focus ourselves on the bias and use some strategies in order to reduce it. (This is an avoidable bias since it is mostly sufficient making the model grow to eliminate it).
In the second case we have similar human and training error, while there is an higher difference between the training and validation error. The most desireable thing is orthogonalize such properties which implies **having small bias while keeping low variance**, so we want them not to influencing each other (in this sense *orthogonal*).

5.4 Facing bias and variance

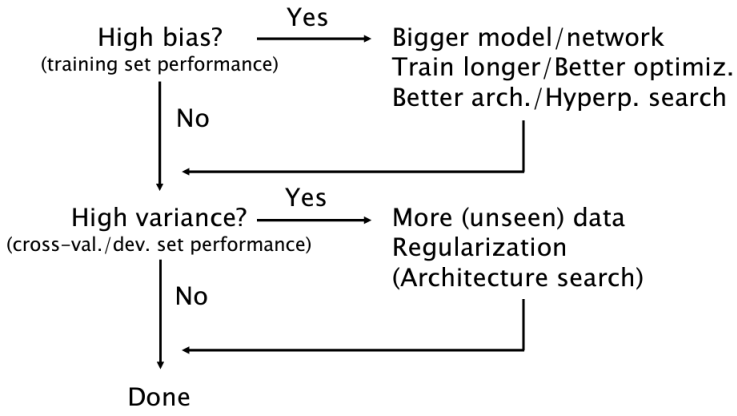
From a study of 2001 it appears evident that:

> "It's not who has the best algorithm that wins, it's who has the most data" (Banko and Brill, 2001)

In principle:

- In order to **reduce the bias** could be sufficient to have a **bigger model**;
- In order to **reduce the variance** could be sufficent to **use more data** in the training phase of the model.

From a conceptual point of view it is sufficient to use the following flow-chart:



The real-world examples demonstrates that variance and bias cannot be orthogonalized, then there is a **trade-off** to manage.

Chapter 6

Large Datasets and Big Models

6.1 Why deep networks?

Several studies have demonstrated that for certain task the performance of certain machine learning models are better with the increasing complexity of the model working on the same set of data. That is when you have a huge amount of data more complex models have better performances. The graph showed below explains such a feature: The classical example can

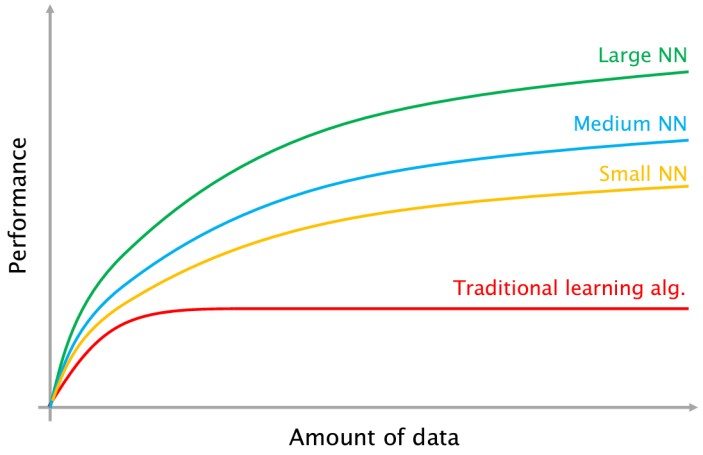


Figure 6.1: Amount of data vs Performances varying the model

be done is the following: imagine that a classification task on images have to be performed, especially whether the figures are colored, there is an exploding number of features. A classical fully connected neural network has very bad performance with respect to a deep network, for example a *Convolutional Neural Network*.

6.2 Aspects related to large datasets and deep networks

We have seen in the past chapter, when a neural network has to be trained there is always a thread-off between bias and variance to manage. There are several aspects related to deep models which ought to be taken into account. In the following the most important aspects are presented.

6.2.1 Regularizing neural networks

How we mentioned in the past paragraphs, the *regularization* is a technique which is used to face the problem of overfitted models. Such a technique consists of introducing into the *cost function* $J(w, b)$ a term which depends on the parameters. For a single neuron the loss function is modified as follows:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \text{Loss}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2 \tag{6.1}$$

The term in red is the **regularization term**, its effect is to keep the complete set of features without eliminate nothing, the difference is that for certain parameters (associated to certain features) the magnitude is very small or equal to zero in order to reduce in some way the complexity of the model which was causing the overfitting phenomenon. The one showed in the (6.1) is the ℓ_2-norm regularization, since the ℓ_2-norm of the vector w of the weights multiplied by a parameter λ (regularization parameter) is added to the original functional. Other types of regualarizations can be used for example the ℓ_1-norm. The regularization term, then has the hyperparameter λ which is crucial. In particular:

- λ *very small* is associated with an **almost full model**;
- λ *very high* is associated with a model whose parameters are very small, and so to a very simplified model.

For a neural network of L layers the J functional becomes:

$$L(W^{[1]}, b^{[1]}, W^{[1]}, b^{[2]}, ...) = \frac{1}{m}\sum_{i=1}^{m} \text{Loss}(\hat{y}^{(i)}, y^{(i)}) + \sum_{l=1}^{L} \|W^{[l]}\|_F^2 \tag{6.2}$$

Where $\|W\|_F^2$ is the *Frobenius Norm* which is the generalization of the ℓ_2-norm in the case of matrices. Intuitively the goal is obtaining $\|W\|_F^2$ close to zero, since near the origin the $g(z)$ behaves in a linear way, this avoids the data to be overfitted.
Clearly, since J is modified, also the gradient descent is modified. More specifically a term

$$\frac{\lambda}{m} W^{[l]}$$

is added. This results in the update step, in multiplying the weights by a quantity equal to

$$1 - \alpha\frac{\lambda}{m}$$

the the higher λ the lower such a contribution which shrinks the parameters more and more near to the origin. This is the reason why the ℓ_2-norm regularization is also called *weight decay*.

6.2.2 Dropout

Dropout is another regularization technique in which for each layer of the network a certain threshold is fixed and this is associated to the probability of keeping or removing one or more of its neurons. The reason why such an apparently strange technique works very well is that removing some units from each layer according to the fixed probability the structure of the network is simpler resulting in a reducing in the overfitting entity. The dropout has to be disabled in the test phase, because is something of helpful only in the phase of construction of the model.

6.2.3 Data augmentation

Among the techniques to reduce overfitting **data augmentation** is used when the dataset is not so rich. This helps us in obtaining new data starting from the ones in the original dataset. Some distortion are introduced in a way that the model perceives that information as different ones. In the field of image classification this is a very used technique. More specifically when Convolutional Neural Networks grew larger in the 90s, there was a lack of data to use, especially considering that a portion of the dataset was devoted to the testing phase. It was proposed to *perturb existing data* with *affine transformation*, in order to create new examples with the same labels. The most common transformation are: geometric, color space transformation and a sort of noise injection. In the following two examples are showed with a cat image and with a number.

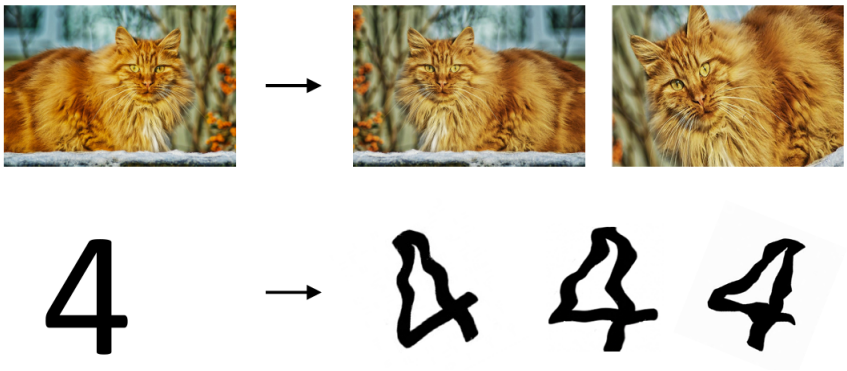


Figure 6.2: Data augmentation

6.2.4 Mini-batch gradient descent

Especially for bigger models than a classical multi-layer perceptron using the "classical" gradient descent could be very slow, since for each one the iterations, which hopefully, bring to the convergence, the **entire dataset** is scanned. This would have made the procedure very slow! That we called "classical Gradient Descent" is also known as **Batch Gradient Descent**.
The alternative here is to split the entire batch of data constituting the dataset into **mini-batch**. Doing in this way, one step of Gradient Descent passes through a subset of the data making the computations faster. When all of the batches of the training set have been used an **epoch** has been completed. In the classical approach one epoch is associated to one step of gradient descent, on the other hand if we split the dataset into M batches, M gradient steps are done in one epoch.

Loss function and mini-batch gradient descent

It is not supposed to be a surprise if we state that the shape of cost function through the different iterations is not so smooth as in the *batch version*, since at each step different data are used.

Mini-batch size

In the case the size of a mini-batch is 1, we talk about the **stochastic gradient descent** ot the opposite if the batch size coincides with the cardinality of the dataset, then batch GD =

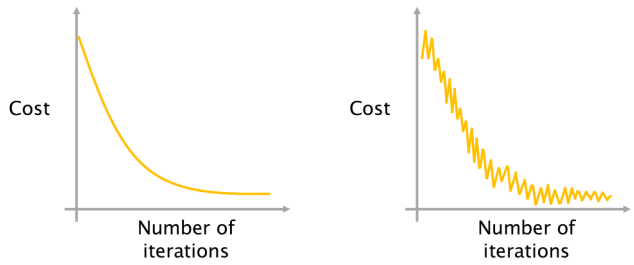
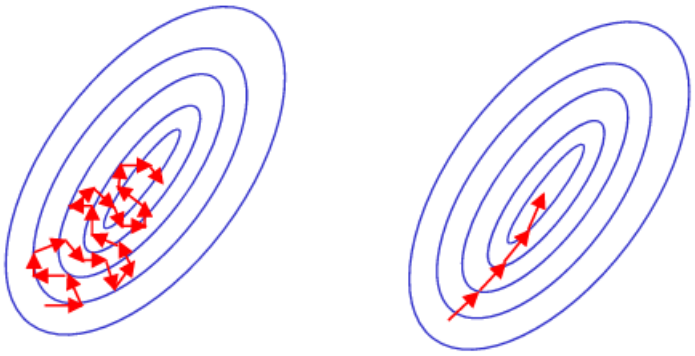


Figure 6.3: # of iterations vs Cost (Batch vs Mini-Batch)

mini-batch GD. If we go deeper into this aspect by analysing the level curves that from the initial conditions bring us to the minimum, the case of batch gradient descent is the ideal one since the path from the initial value to the minimum is straight. The same does not occur in the case SGD is used. In the practical case a value for the mini-batch size between 1 and m must be chosen, this choice results in another *hyperparameter*.



(a) size=m (b) size=1

Figure 6.4: Contour plots varying batch size

The suggestion in this field is to use the original version of gradient descent with a small dataset (eg. 2000 examples), otherwise typical sizes for mini-batches are 64, 128, 256 and so on. In order to avoid problems, you are supposed to be sure that it fits in the used CPU/GPU.

6.2.5 The problem of local minima

Let us make another objection on the cost function, we have seen which is a fundamental building block of our machine learning task. Now, after having combined the several layers of the network (each one of the layers use different activation functions) is the $J(w, b)$ convex? The answer is NO. The functional we obtain loses its convexity with the increasing complexity of the network. However, thanks to the structure of the final cost function, the probability to be trapped into a local optima is very low. `part to be rewiewed`

In the case that in the functional there are **plateaus** can be a problem, since being the derivatives constant the learning is very slow.

6.2.6 Exploding/Vanishing gradients and initialization in DNN

The **exploding** and **vanishing gradient** are both problems related to very deep networks were the weights are too high or too low, in the former case the computed gradients *grow*

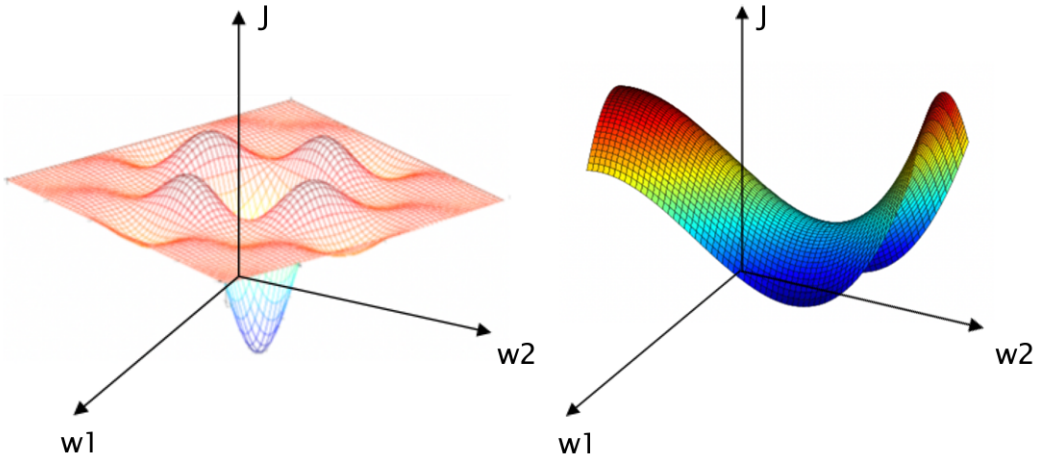


Figure 6.5: Cost function for a NN with two parameters

exponentially, in the latter case they decrease their own value till they become null.
For sake of simplicity suppose that the activation function is a line $g(z) = z$, moreover let $b = 0$, than the predicted output $\hat{y}$ has the shape:

$$\hat{y} = W^{[L]}W^{[L-1]}...W^{[1]}x$$

Whether for example the weights were all equal to 1.5 the $\hat{y}$ would be very big, on the other hand, whether all of the weights were 0.5 there would be an *exponential decreasing* of the activations, a similar situation occur for the derivatives. In deep neural networks not rarely such problems appear, and differently from the **shallow neural networks** a more accurate technique aimed to cope with them is needed. In particular, has been empirically demonstrated that such a problem is reduced when the weights $w^{[l]}$ of a certain layer l are randomly initialized with a value in the range $[0, 1]$ multiplied by the standard deviation

$$\sigma^{[l]} = \sqrt{\frac{1}{n^{[l-1]}}}$$

where $n^{[l-1]}$ is the number of unit of the $(l-1)$-th layer (previous layer).

6.2.7 Batch normalization

We have seen in the introduction in order to speedup the training phase a proper choice when data are on completely different scales is the **normalization**. To better clarify such an aspect, we can say that 2 different steps are performed[1]:

- Subtract the mean μ computed on that feature on the whole dataset (to be computed a-priori);
- Divide by standard deviation σ computed always over the whole training set for that specific feature.

In the past years, scientists working on deep learning has showed that if such a normalization is applied also for the activations (more specifically to the linear part z), then the learning

[1]The same μ, σ are supposed to be used in order to normalize also the remaing parts of the dataset: *test* and *development* set.

of the parameters for a certain level is **faster**. This is the main feature behind the **batch normalization**. We know that for a certain layer l, we can compute the activation $A^{[l]}$ as:

$$A^{[l]} = g(Z^{[l]})$$

then the *batch normalization* procedure is carried out as follows:

- For each layer l, for each feature i, the mean μ and variance σ^2 are computed.
- The normalized data $Z^{[l](i)}_{\text{norm}}$ are obtained as follows:

$$Z^{[l](i)}_{\text{norm}} = \frac{Z^{[l](i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

- For the training phase the following data are used:

$$\tilde{Z}^{[l](i)} = \gamma Z^{[l](i)}_{\text{norm}} + \eta \tag{6.3}$$

This approach, fortunately or not, leads with it other hyperparameters which are $\varepsilon, \gamma, \eta$. Moreover, just to further complicate the situation, for each layer different $\gamma^{[l]}$ and $\eta^{[l]}$ could be used. It is interesting now, after this formal description to better understand what are the guidelines leading to batch normalization and what is its effect.
In principle when I perform the normalization on the input data (remind they are also called 0-layer activations) after the first forward step, I completely lose the effect since the first-layer activations are something very different than the *normal* range. Moreover batch normalization also has a *slight regularization effect*: the fact that mean and variance are computed with respect to that mini-batch adds similarly than *dropout* adds some noise to each hidden layer activations.

6.2.8 Softmax Layer

At the beginning we have seen that the hypotesis (later called *activation*) can be interpreted as the probability of belonging to a certain class given the records X, but how can be interpreted as a result? We would like to have on the last layer L a some probability that, differently from the other sum up to one. For this reason, very often in the neurons of the last layer a particular activation function is used:

$$a^{[L]}_i = \frac{t_i}{\sum_{j=1}^{n^{[L]}} t_j} \tag{6.4}$$

where $a^{[L]}_i$ is the activations for the $i-th$ unit in layer L, and $t_i \doteq e^{z^{[L]}_i}$. When the softmax activation function is used, the loss function to be used is the following (*for a single training sample*):

$$\text{Loss}^{(i)}(\hat{y}, y) = \sum_{j=1}^{n^{[L]}} -y_j \log(\hat{y}_j) \tag{6.5}$$

Note that $n^{[L]}$ is the number of classes, in this case for the m training samples the **one-hot encoding** is used for representing the labels, in particular:

$$Y = \begin{bmatrix} y^{(1)} & y^{(2)} & \dots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0 & 1 & & 0 \\ 1 & 0 & & 1 \\ 0 & 0 & \dots & 0 \\ 0 & 0 & & 0 \end{bmatrix} \in \mathbb{R}^{n^{[L]}, m} \tag{6.6}$$

For each example a vector in which only i-th element is one with correspondence to the true class for that record.

6.2.9 Transfer learning

Especially in the field of deep learning, rarely one starts from scratches to develop the neural network. There are several motivations for which this is not happening, a common one is that there is lack of resources. To the aim to train certain models several dozens of GPU could be required... The **Transfer learning** aids us to cope with this problem. It consists in the use of a part of a pre-trained model in order to use its parameters as input features for a task of us. Then, by doing the so called, **surgery of the network** several layers are freezed,in the sense that only the forward step is done on these, while the parameters are updated only for the few last layers. The difference is that if we had started from scratches a lot of time and hardware/software resources would have been needed. This is one of the best common practice of the deep models.

Can we always use transfer learning? The answer is clearly No! You can pass by this procedure: either when two tasks have common inputs, or you have more data for a task then for another, or also low level features for a task could be useful for the other.

Chapter 7

Computer vision and Convolutional Neural Networks

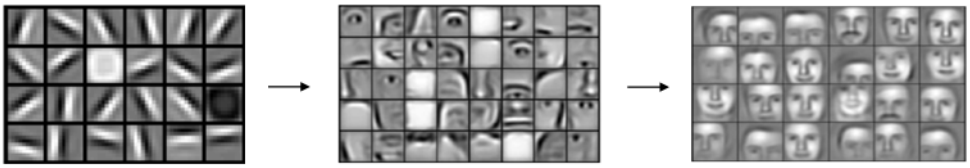
Computer Vision is a field of Artificial Intelligence which aims to implement models that performs visual tasks. Some of the tasks in this field are for example: *image classification*, *object detection* within an image, the so-called *neural style transfer* and so on. All of this tasks, for the nature of input data, involves an enormous number of features which corresponds to a huge number of parameters to learn. How we have said several times, a *classical neural network* (shallow) for example, cannot perform properly and in acceptable time such a task. Just for give an idea, the number of parameters to lean can be also around a billion! For this reason **convolutional architectures**, and then **convolutional neural networks** are introduced.

7.1 Convolutional Neural Networks: main ingredients

A **Convolutional Neural Network** (**CNN**) is made up of several parts: (i) a *convolutional layer*, (ii) a *pooling layer*, (iii) a *fully connected layer*.

7.1.1 Convolution

This is the core building block of a CNN, here the great majority of the computation occurs; there are several levels of this type. Besides, the convolutional layers are the ones in which the network learns the main feature from the input data. In the case of images, passing through the convolutional part of the NN low level to high level features are learned. The following figure shows an example of the extracted details at different levels of the architecture:



You can see in the first stage only some edges are detected, passing through more complicate details arriving to the detection of entire faces.
This procedure takes inspiration on how our brain solve the problems; biological studies have confirmed that in order to perform a certain task, our brain solves, step by step, simpler problems in order to reach more complicate ones.
From now on, we are going to focus our attention on **images**, and in particular we are going

deeper in some details on *how convolution process works*.

The *convolution* requires few components: (a) input data, (b) a **filter**, (c) a **feature map**. Considering that an image can be seen as a matrix of pixels, roughly speaking the filter moves across the image checking if the feature for which that filter itself has been built, is present. This process is known as **convolution**.

In the following there is a figure that shows, mathematically speaking, what are the main steps behind such a procedure.

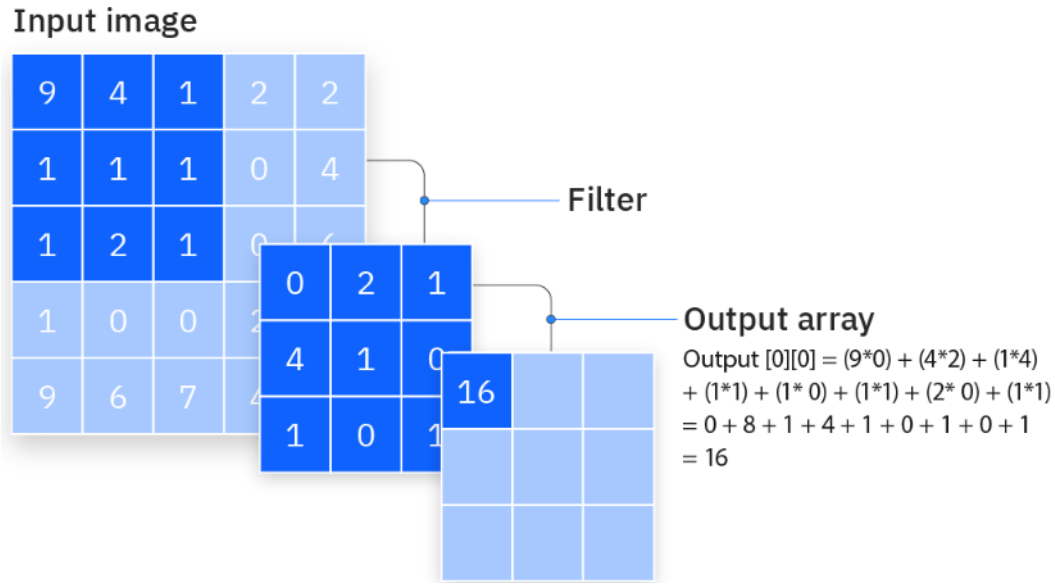


Figure 7.1: Convolution: main steps

In practical terms also a filter is a square matrix (with *odd dimension*) in a way that it has a center. Such a filter is *convolved* to the image in the sense it slides across subregions of it which have the same dimension of the feature detector; the output array (another matrix called the **feature map**) is done by scalars corresponding to the sum of the product element by element of the involved matrices.

How we will see, part of the parameters that the the network has to learn are those constituting such filters, then no one tells to the network how to find edges (also at different inclination) or other types of details!

It should be clear that, going across the process of convolution the dimension of the matrices is reduced. In particular: starting from an image (square for simplicity) $n \times n$, by applying on it a filter $f \times f$ the dimension of the ouput will be shrinked by a quantity $f - 1$ (for each dimension), with a resulting size of $(n - f + 1) \times (n - f + 1)$.

Padding

We can add a frame of padding to the image (usually by adding zeros) in order to avoid the phenomenon of *shrinking dimensions*. Then, if a padding of p is added to the input image (so that it results in being $(n + p) \times (n + p)$) and an $f \times f$ filter is applied the resulting image will have shape $(n + 2p - f + 1) \times (n + 2p - f + 1)$.

Only for a matter of nomenclature, we have to say that a convolution which does not use the padding is called *valid convolution*, otherwise we have a *same convolution*. The **amount of padding** to be added is such that the input and ouput images have exactly the same shape.

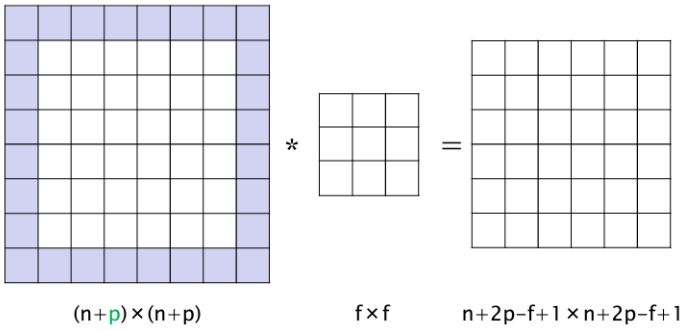


Figure 7.2: Image padding for avoiding dimension reduction

By doing simple calculations we can find that this quantity is equal to $p = (f-1)/2$, it gives always a non-fractional value since f is known to be odd.

Strided convolutions

The last step is needed to complete the overview on the first part of CNN: **strided convolutions**. Whether in the procedure of applying the kernel some pixels (cells of the matrix) are skipped, then the procedure is known to be **strided**. A number of skipped cells greater than one is rare, however it is remarkable that also in this case the dimension of the feature map is shrinked. More clearly, for a stride s the dimensions for the output are:

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor$$

How you will imagine, after some chapter of discussion on NN, such an s is another hyperparameter (usually $s = 1$, if a *same convolution*) is used.

7.1.2 Convolutions on RGB images

In the previous paragraphs, for sake of clarity about the main aspects of convolution, we have implied that the image for which we were training the CNN was a gray-scale one. A part from few tasks, nowadays colored images are used. Let us suppose, without loss of generality, on the contrary they are RGB ones. This implies that now the input images are not 2D-arrays anymore, they are 3D since there is an $n \times n$ matrix for each one of the three channels R, G, B. A *3D-kernel* is needed as the number of channels. Despite the shape of the inputs is changed, what is not changing is the simple computational procedure, since all of the values are summed up! Then the feature map is always a 2D-array and the network is clearly allowed to use different or same filters.

Multiple Filters

In the case that at this stage *multiple filters* are used, also the output has a three-dimensional shape. Now, whether on a RGB image whose shape is $n \times n \times n_C$, is applied n'_C filters whose shape is $f \times f \times n_C$ (with n_C being the number of channels) $\rightarrow$ the output shape will be (no padding, no striding) $(n-f+1) \times (n-f+1) \times n'_C$.

We can see that the convolution operation is nothing but a (just more complicated) linear combination. This is the counterpart of z in the linear regression, then – also here – a *nonlinear*

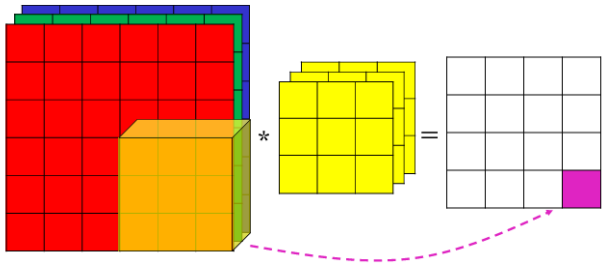


Figure 7.3: Convolution on RGB images producing 2D-output

part is missing! In fact, before passing the output to the next layer, even in this case an activation function is employed, in particular the ReLU. This prepares the activations for the next layer. Such a situation is well depicted in the following:

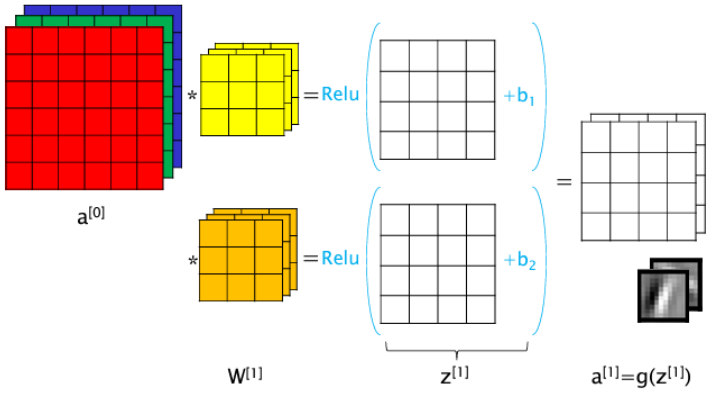


Figure 7.4: ReLU on feature maps