MACHINE LEARNING FOR VISION AND MULTIMEDIA

Lecture notes

Carlo Migliaccio

AA 2024/2025

Contents

1 Introduction **5**
- 1.1 Supervised learning . . . 5
  - 1.1.1 Linear Regression . . . 6
  - 1.1.2 Classification . . . 6
- 1.2 Unsupervided learning . . . 7
  - 1.2.1 Clustering . . . 8

2 Model, cost, parameter learning, Gradient Descent **9**
- 2.1 Model representation . . . 9
- 2.2 Parameter learning: Gradient descent . . . 11
- 2.3 Multivariate linear regression . . . 13
- 2.4 Data mean normalization . . . 13
- 2.5 Debug of Gradient Descent algorithm . . . 13
- 2.6 Alternative to Gradient Descent . . . 13
- 2.7 Polynomial regression . . . 14

3 Logistic Regression **15**
- 3.1 Classification vs Regression . . . 15
- 3.2 Logistic regression . . . 16
- 3.3 Cost function for logistic regression . . . 18
- 3.4 Training logistic regression . . . 18
- 3.5 Multiclass classification . . . 19
- 3.6 Overfitting and Regularization . . . 19

4 Neural Networks: an introduction **21**
- 4.1 Ideas underlying neural networks . . . 21
  - 4.1.1 Notation . . . 22
  - 4.1.2 Different types of NN for different types of purposes . . . 23
- 4.2 Logistic regression as a NN . . . 23
- 4.3 Automatic differentiation and computation graph . . . 24
  - 4.3.1 Feedback and Backward propagation for Logistic Regression . . . 25
- 4.4 Training Neural Networks . . . 26
  - 4.4.1 Forward propagation . . . 26
  - 4.4.2 Backward propagation . . . 27
- 4.5 Activation functions . . . 28
- 4.6 Initialization of the parameters . . . 28
- 4.7 Training a neural network (Recipe) . . . 28
- 4.8 Hyperparameters . . . 29
- 4.9 Training, Development, Test sets . . . 29

5 Evaluating learning algorithm **31**
5.1 Underfitting and overfitting data . . . 31
5.2 Metrics for model evaluation . . . 32
5.2.1 Confusion matrix and Precision/Recall . . . 32
5.3 Human-level performance . . . 33
5.4 Facing bias and variance . . . 33

6 Large Datasets and Big models **35**
6.1 Why deep networks? . . . 35
6.2 Aspects related to large datasets and deep networks . . . 35
6.2.1 Regularizing neural networks . . . 36
6.2.2 Dropout . . . 36
6.2.3 Data augmentation . . . 37
6.2.4 Mini-batch gradient descent . . . 37
6.2.5 The problem of local minima . . . 38
6.2.6 Exploding/Vanishing gradients and initialization in DNN . . . 38
6.2.7 Batch normalization . . . 39
6.2.8 Softmax Layer . . . 40
6.2.9 Transfer learning . . . 41

7 Computer vision and CNN **42**
7.1 Convolutional Neural Networks: main ingredients . . . 42
7.1.1 Convolution . . . 42
7.1.2 Convolutions on RGB images . . . 44
7.1.3 Notation . . . 45
7.1.4 Pooling layer: Max-Pooling . . . 46
7.1.5 Fully connected layer . . . 47
7.1.6 Why Convolutions? . . . 47
7.2 Case studies and tasks . . . 47
7.2.1 AlexNet . . . 48
7.2.2 VGG-16 . . . 48
7.2.3 Residual Network(ResNet) . . . 49
7.3 1×1 convolutions . . . 50
7.3.1 Inception: another DNN architecture . . . 50
7.4 Other computer vision tasks . . . 51

8 Localization and Object Detection **52**
8.1 Classification with localization . . . 52
8.2 Object detection . . . 53
8.2.1 OverFeat: a fully Convolutional Architecture . . . 53
8.2.2 Region Proposal: R-CNN . . . 55
8.3 Fast R-CNN . . . 56
8.4 Faster CNN . . . 56
8.5 You Only Look Once (YOLO) . . . 58
8.5.1 Basics for YOLO . . . 58
8.5.2 Overlapping objects: introduction of anchors . . . 58
8.6 Non-max suppression algorithm . . . 60
8.7 Evaluating object localization and detection performance . . . 60
8.8 Final considerations . . . 61

9 Segmentation and Neural Style Transfer 62
9.1 Semantic segmentation . . . 62
9.1.1 In-Network upsampling: Unpooling . . . 63
9.1.2 Learnable upsampling: Deconvolution . . . 63
9.1.3 SegNet: Encoder-Decoder for Image Segmentation . . . 64
9.1.4 Other Architectures for segmentation . . . 64
9.2 Instance segmentation . . . 65
9.2.1 Segmentation mask . . . 66
9.2.2 RoI-Align . . . 66
9.2.3 Traing Mask R-CNN . . . 66
9.3 Face verification/recognition . . . 67
9.3.1 The need of a similarity function . . . 67
9.3.2 Triplet loss . . . 67
9.3.3 Siamese Network . . . 68
9.4 Neural style transfer . . . 68
9.4.1 $J_{\text{Content}}(C, G)$: content cost function . . . 69
9.4.2 $J_{\text{Style}}(S, G)$: style cost function . . . 69
9.4.3 Generating the output image . . . 70
9.4.4 Final comments . . . 70

10 Sequential models: RNN, LSTM, GRU, Transformers 71
10.1 Notation . . . 71
10.2 Representing words . . . 72
10.3 Recurrent Neural Networks (RNN) . . . 72
10.3.1 Motivations for introducing a novel architecture . . . 72
10.3.2 Recurrent neurons and layers . . . 73
10.3.3 RNN architectures . . . 75
10.3.4 Bidirectional RNN . . . 76
10.3.5 Deep RNN . . . 76
10.4 Language Modeling with RNN . . . 77
10.4.1 Training an RNN language model . . . 78
10.4.2 Use case: Sentence generation . . . 78
10.5 Issues with RNN training . . . 79
10.6 Long-Short Term memories (LSTM) . . . 79
10.7 Gated Recurrent Unit (GRU) . . . 80
10.8 Image captioning with RNN . . . 81
10.9 Attention mechanisms . . . 82
10.9.1 Image captioning with attention . . . 83
10.9.2 Visual question answering . . . 84
10.10 Attention is all you need: *Transformer* architecture . . . 85
10.10.1 Scaled Dot-product attention . . . 86
10.10.2 Multi-head Attention layer . . . 86
10.11 Transformers vs RNN . . . 88

11 Machine and Deep Learning for Audio 91

12 Generative Adversarial Networks (GAN) 92
12.1 Introduction . . . 92
12.2 Variational Auto-Encoders (VAE) . . . 92
12.2.1 Autoencoders . . . 93
12.2.2 Variational autoencoders . . . 93
12.3 Generative Adversarial Networks . . . 94
12.3.1 GAN anatomy . . . 95
12.3.2 Training GAN . . . 96
12.3.3 GAN Formulation . . . 96
12.3.4 When to stop training GAN? . . . 97
12.3.5 The challenges in Training GAN . . . 97
12.4 From GAN to DCGAN . . . 97

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 19th 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:

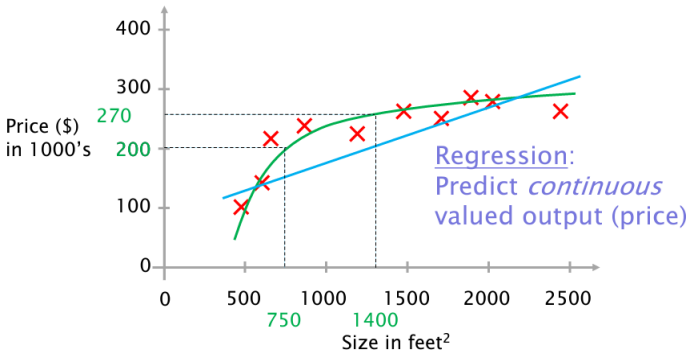
- *Supervised learning* (this course), is the approach which uses **a-priori knowledge** embedded in the data that are used for training algorithms and recognize patterns;
- *Unsupervided learning*, is the approach at the opposite whose main feature is not using *labeled data* for assess the tasks.
- *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

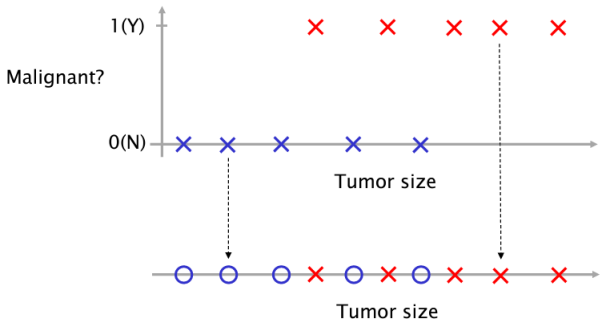
Let us imagine we are supposed to create a model that allows us to **predict the price of an house**. For sake of simplicity and clarity, suppose that our *dataset entries* have one feature (the house size in ft^2) and the *price* which represents the **correct answers**. Using this data we seek for 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

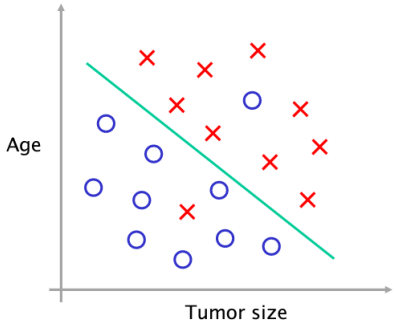


Figure 1.1: Bi-variate problem with linear decision boundary

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.

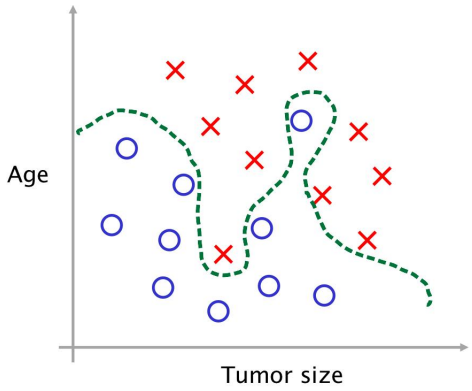


Figure 1.2: Example of overfitting

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]

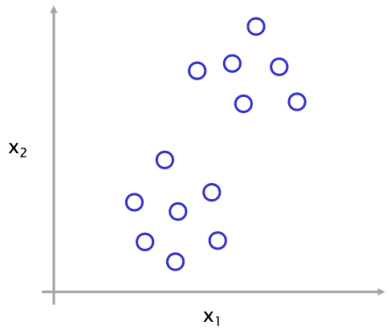
1.2 Unsupervided learning

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.

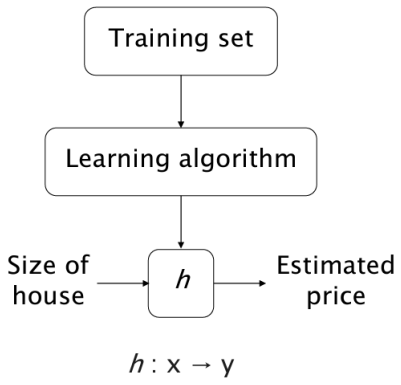


Figure 2.1: Scheme for model construction (price prediction)

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:

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

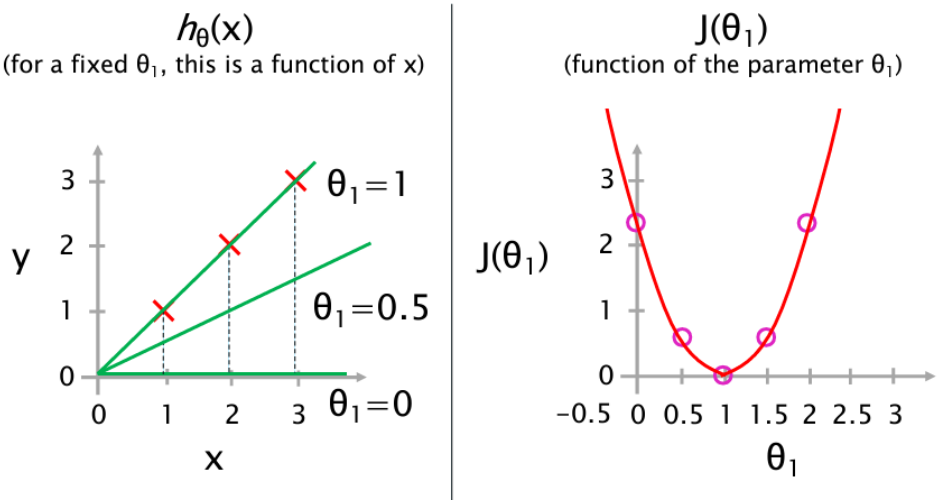
where θ_0 and θ_1 are the parameters of the line.[1] We call *univariate* the the problem since we have only one feature and it is a *linear regression* because we want to predict the price (output) according to a line.[2] **Question: How can we choose θ_0, θ_1?** Intuitively one can choose the parameters 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:

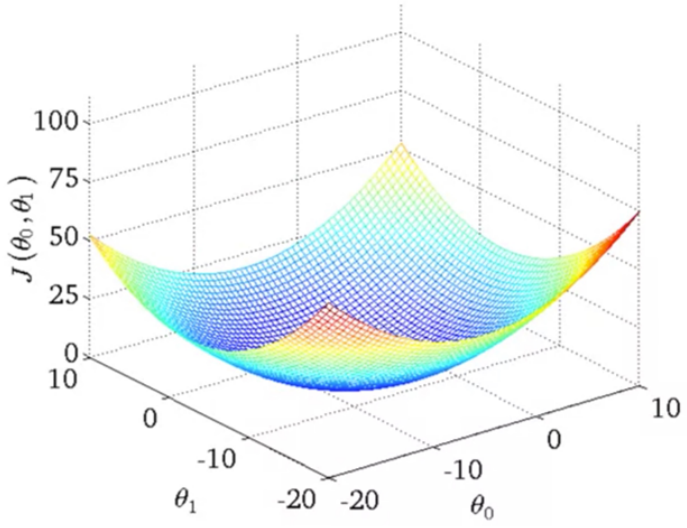
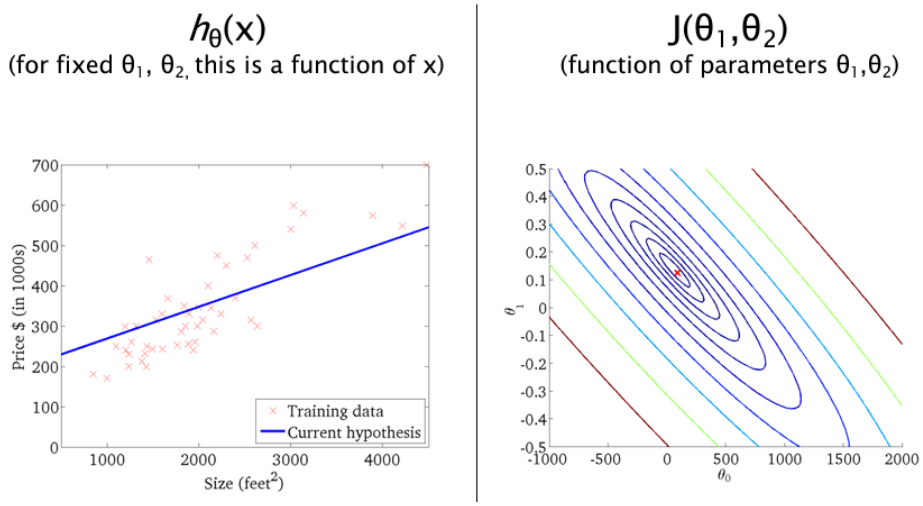


Figure 2.2: Example of $J(\theta_0, \theta_1)$

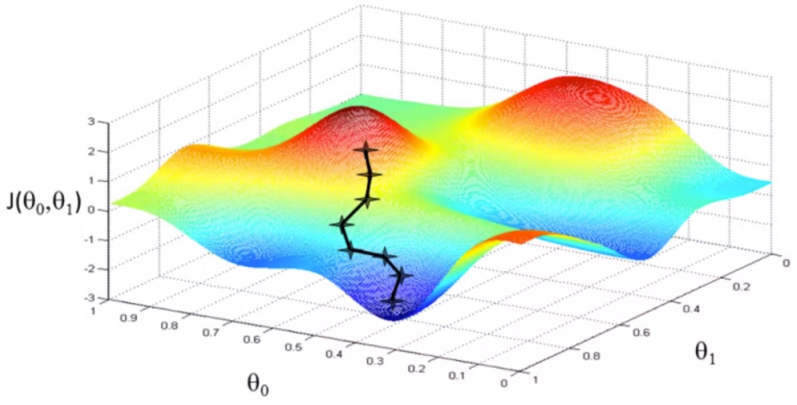
In the common case of bivariate minimization problem one can use *contour plot* which analyze the shape of the function at different heights. It is remarkable that points in the space (θ_0, θ_1) which are on the same *countour line* result in very different hypotesis. It is trivial to understand that, 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.

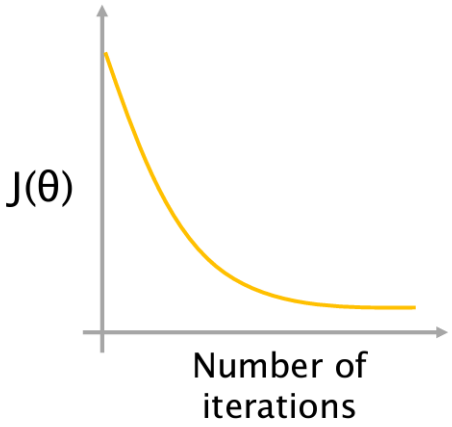


Figure 2.3: Desired behaviour for $J(\theta)$ vs # iterations

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

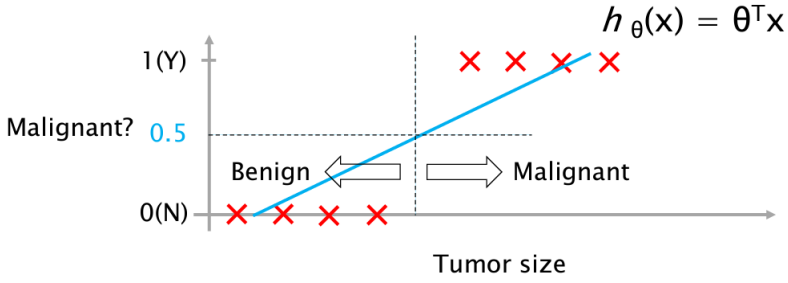


Figure 3.1: Classification with linear hypotesis

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.

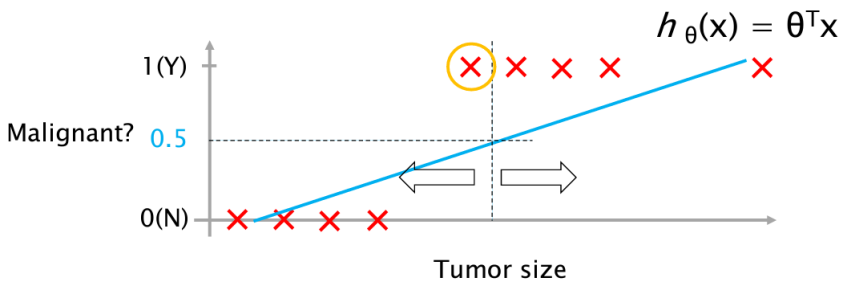


Figure 3.2: Effect of changing the dataset

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:

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

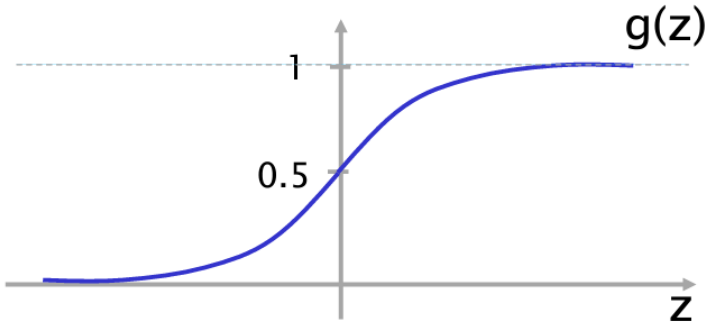


Figure 3.3: The logistic function $g(z)$

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

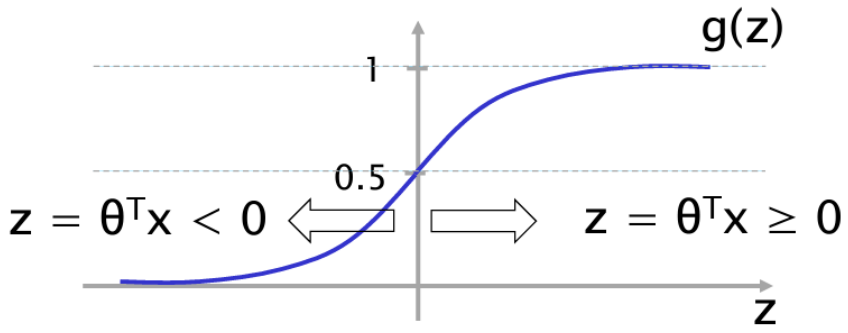
$$h_\theta(x) = P(y = 1|x;\ \theta) \tag{3.3}$$

[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:

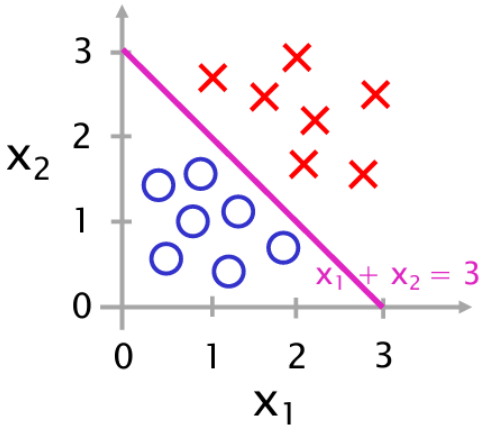


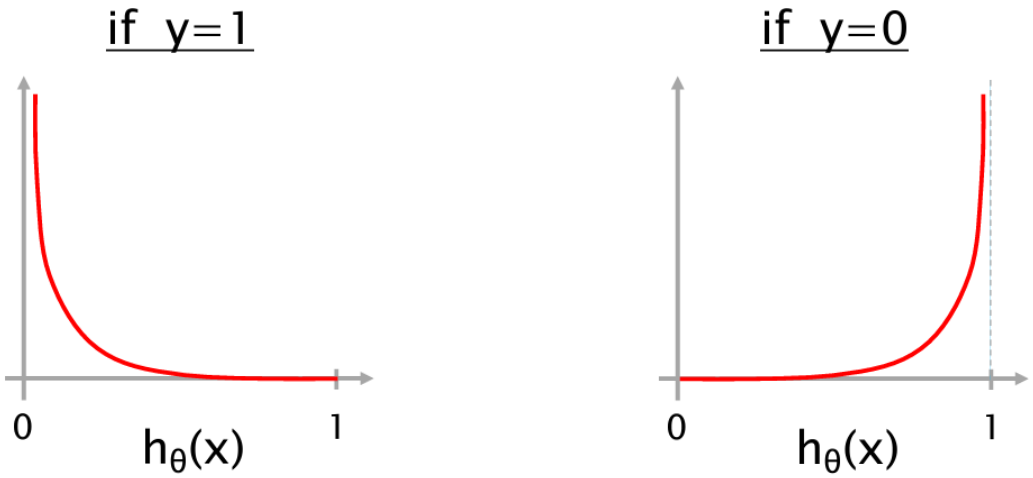
Figure 3.4: Decision boundary

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

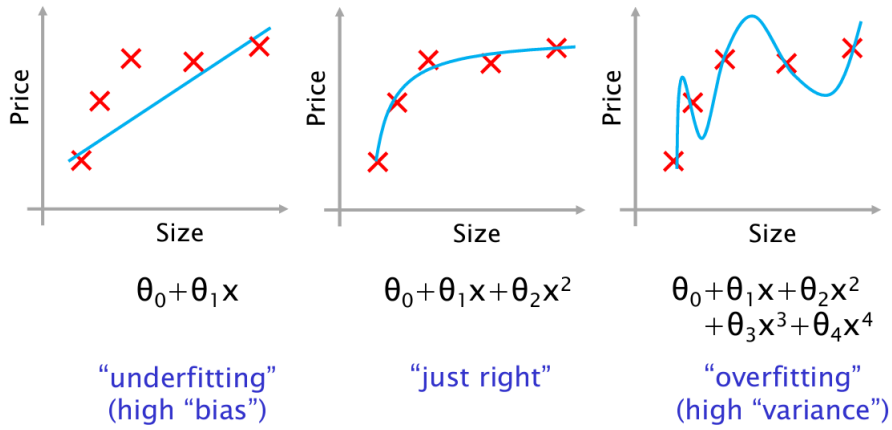
Suppose we want to do a multiclass classification, for example for tagging mail as **SPAM, WORK, FRIENDS...** Can we use logistic regression in order to carry out such a task? The answer is YES, but with some modifications. In the sense that we can reformulate the problem in *one class against the others*. 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

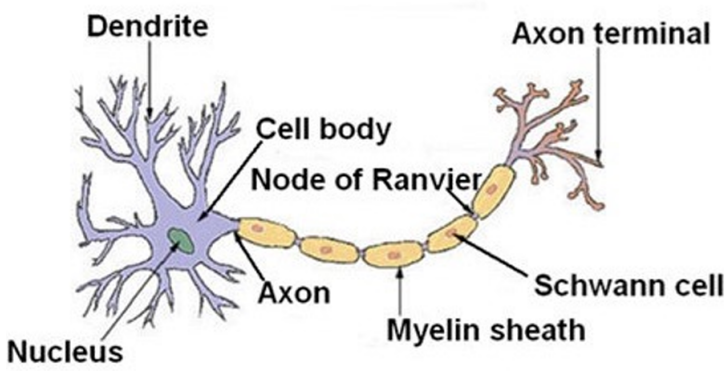
Chapter 4

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**.

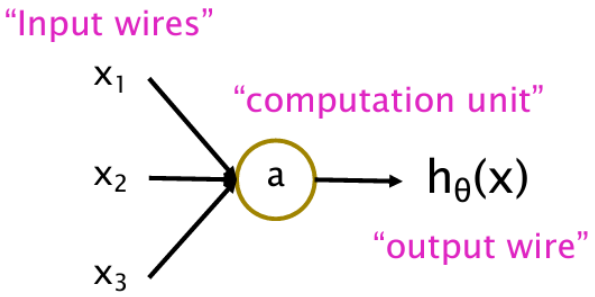


Figure 4.1: Structure of an artificial neuron

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:

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

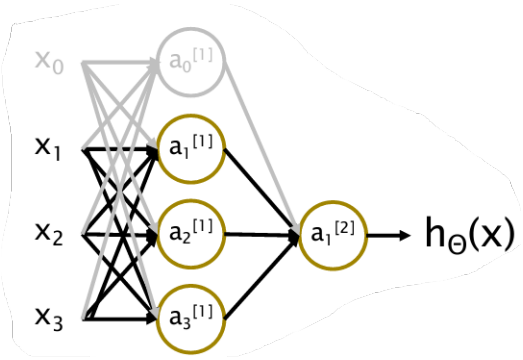
In this case the weighting matrices are for the first layer $\Theta^{[1]} \in \mathbb{R}^{3,4}$, for the second layer is $\Theta^{[2]} \in \mathbb{R}^{1,4}$.

[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:

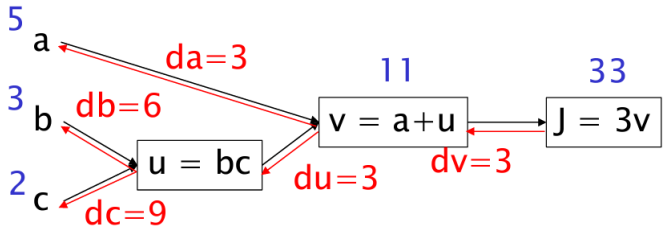
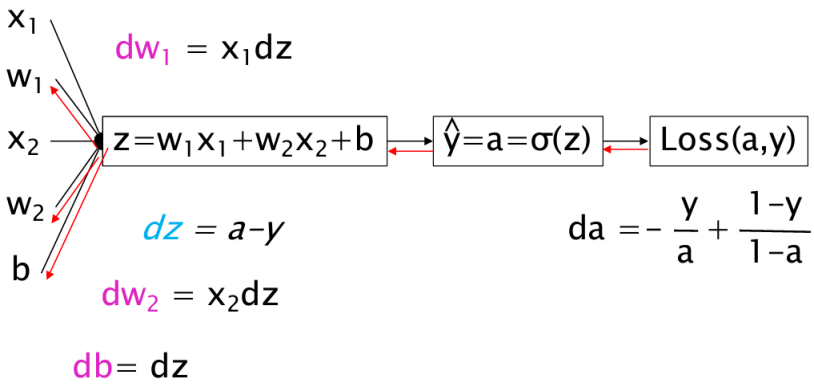


Figure 4.2: Computation graph for $J = 3(a + bc)$

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).

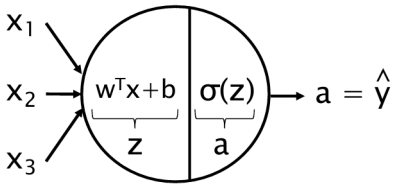
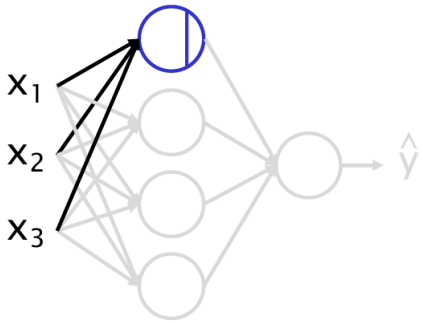


Figure 4.3: Single neuron in form of logistic regression

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

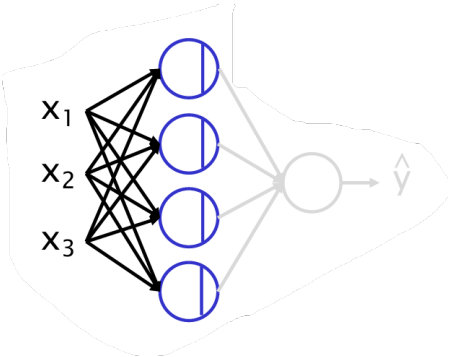


A single neuron of the layer is considered, which clearly has his own weights and bias.

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

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:

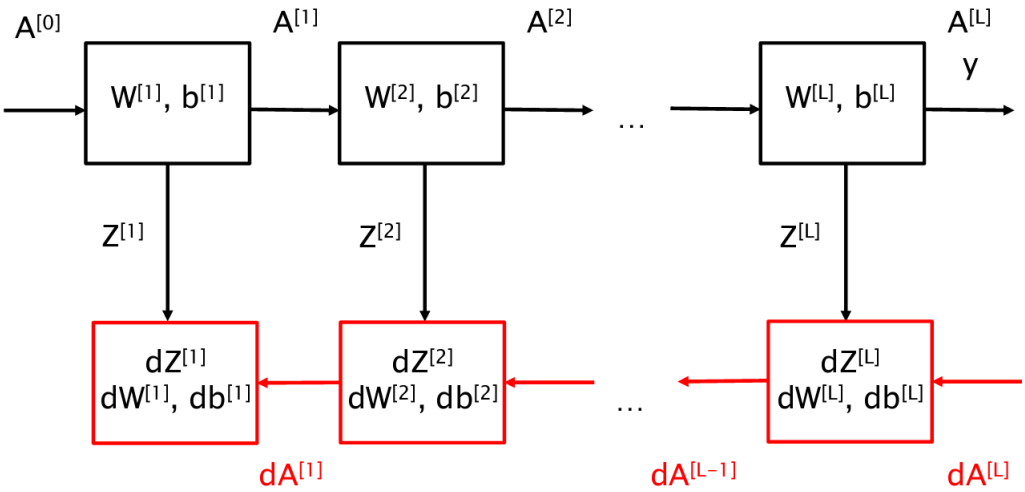
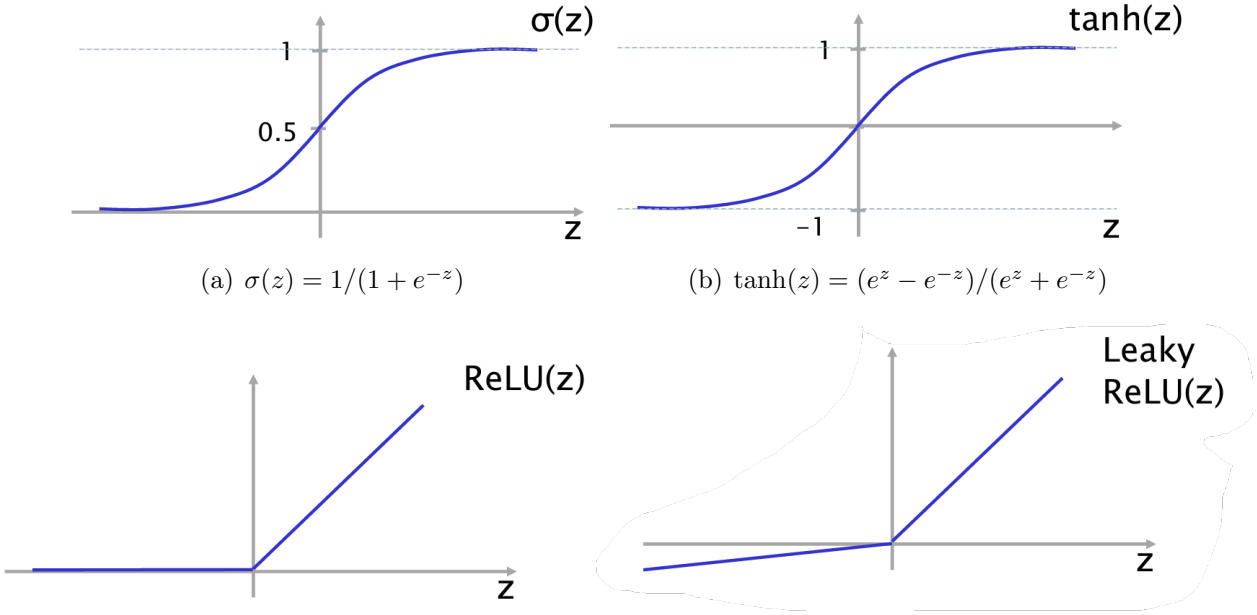


Figure 4.4: Block diagram for the GD

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:

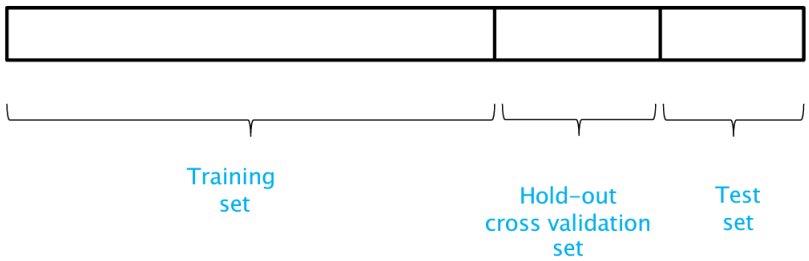


Figure 4.6: Traditional dataset partitioning

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:

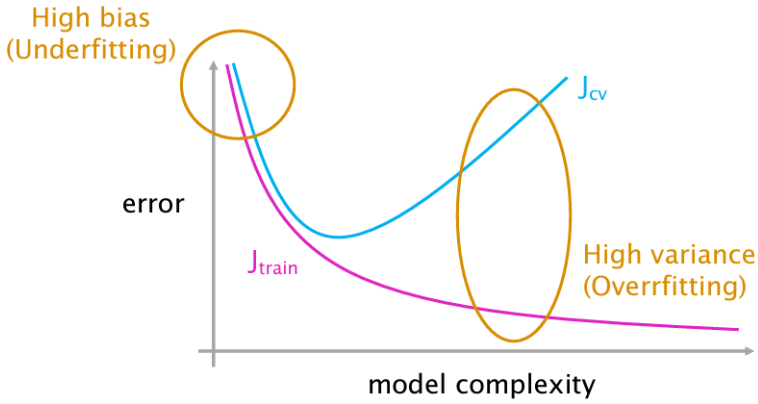
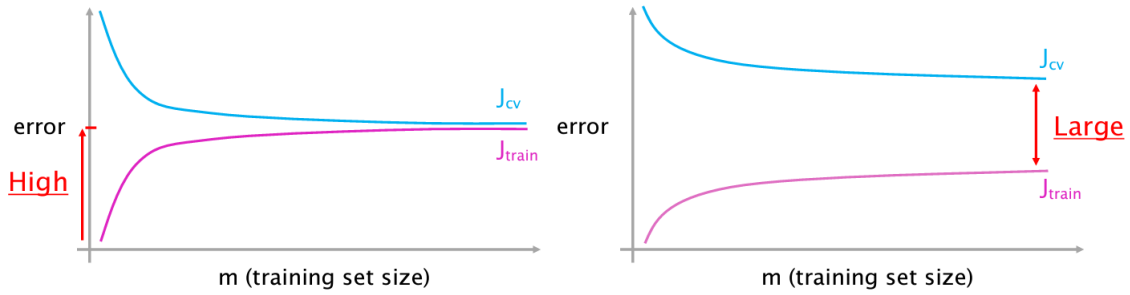


Figure 5.1: High-bias vs High-variance

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:

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

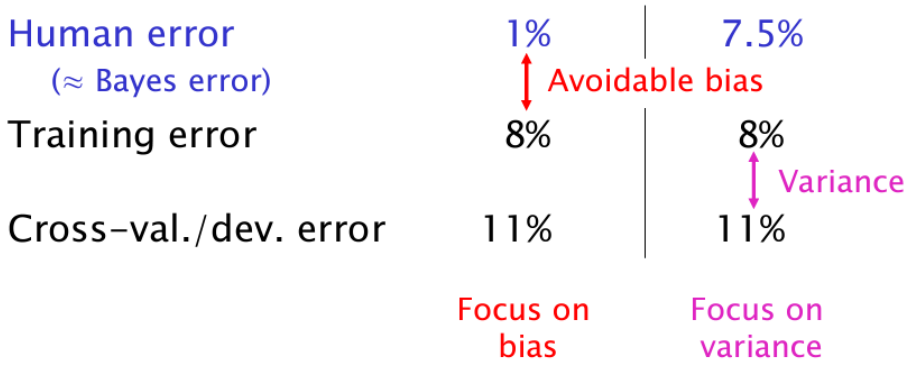
In order to compare such metrics, another auxiliary index is introduced, the F Score which is the armonic mean between recall and precision:

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

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

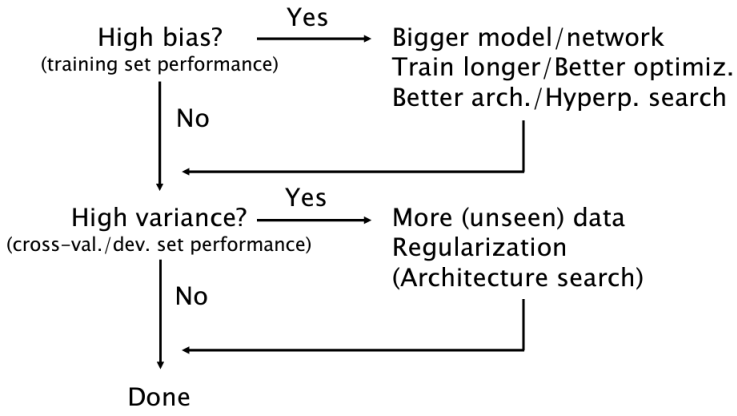
From a study of 2001 it appears evident that:

> "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

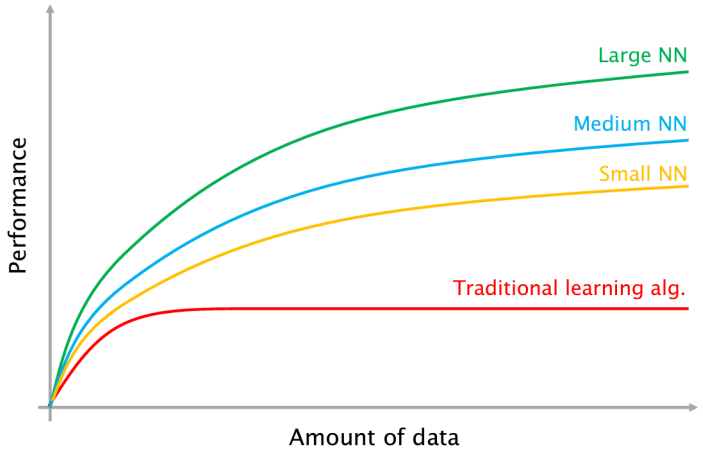


Figure 6.1: Amount of data vs Performances varying the model

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.

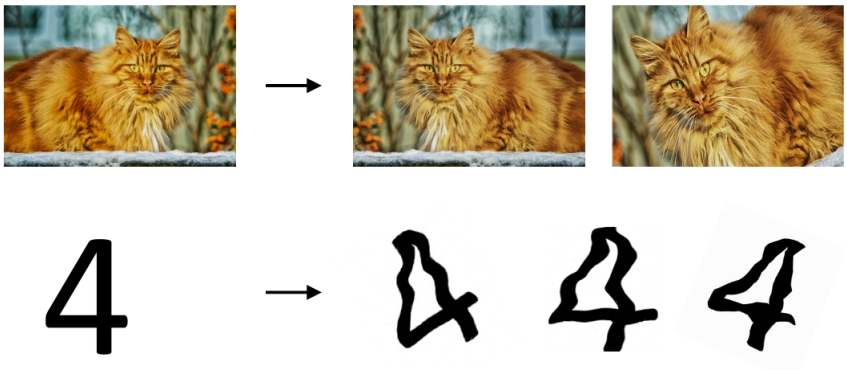


Figure 6.2: Data augmentation

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 =

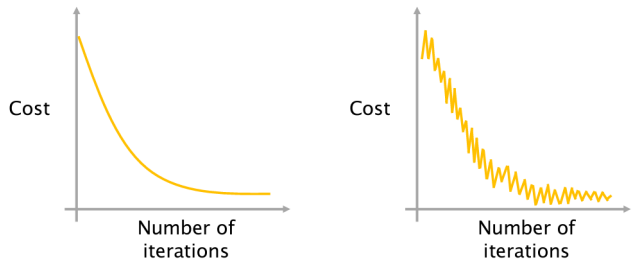
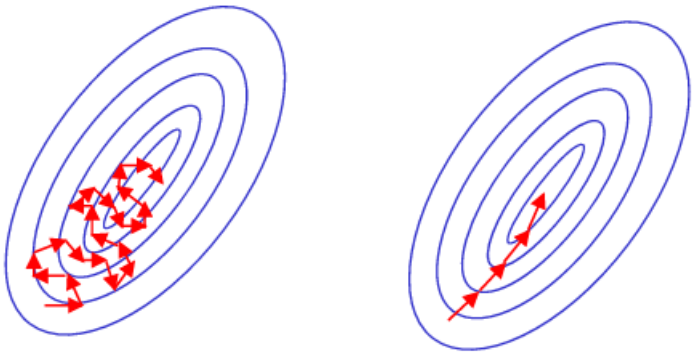


Figure 6.3: # of iterations vs Cost (Batch vs Mini-Batch)

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*

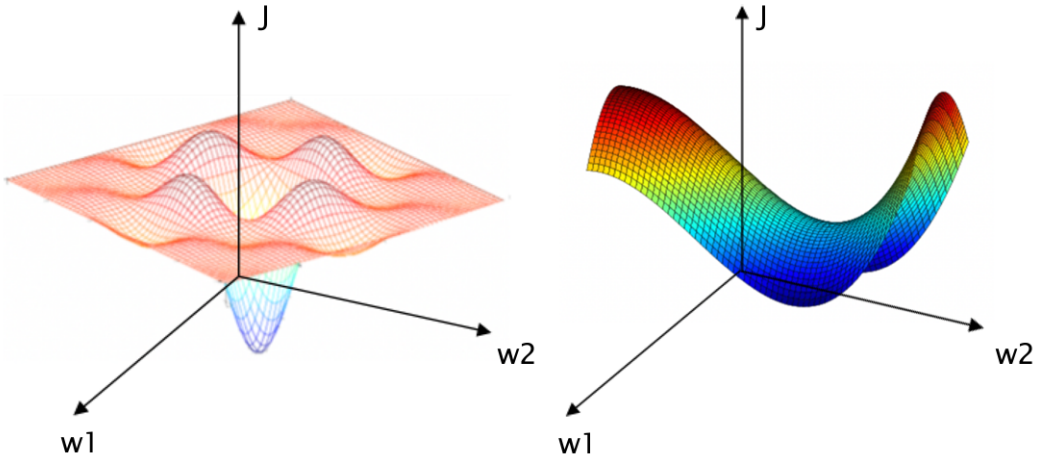


Figure 6.5: Cost function for a NN with two parameters

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

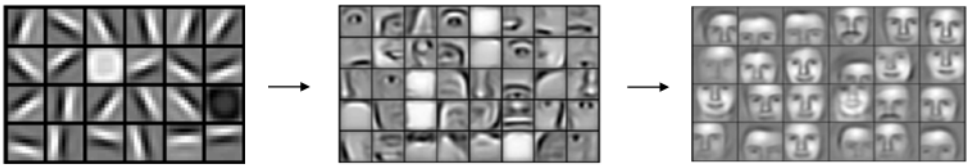
Computer Vision is a field of Artificial Intelligence which aims to implement models that performs visual tasks. Some of the tasks in this field are for example: *image classification*, *object detection* within an image, the so-called *neural style transfer* and so on. All of this tasks, for the nature 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.

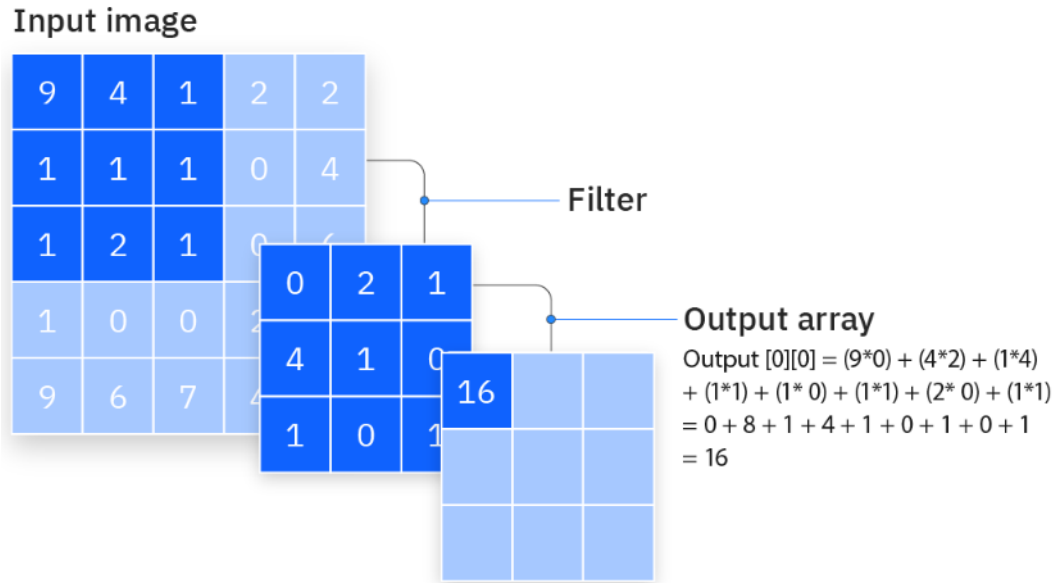


Figure 7.1: Convolution: main steps

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.

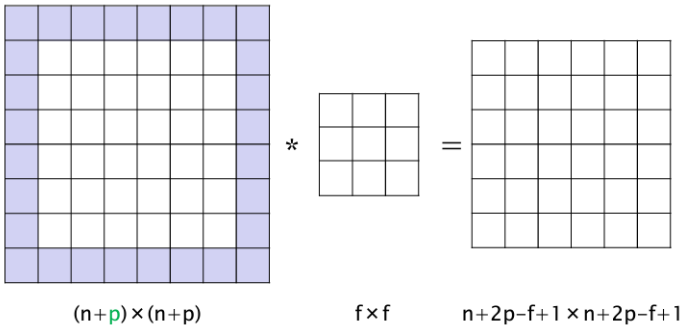


Figure 7.2: Image padding for avoiding dimension reduction

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 (number of channels of the output) 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*

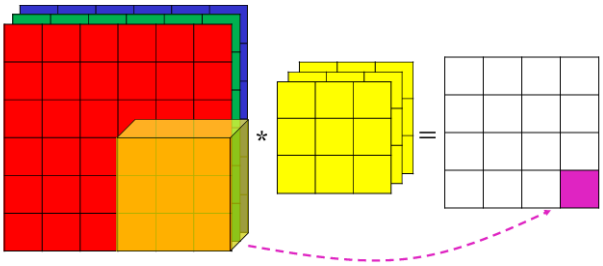


Figure 7.3: Convolution on RGB images producing 2D-output

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:

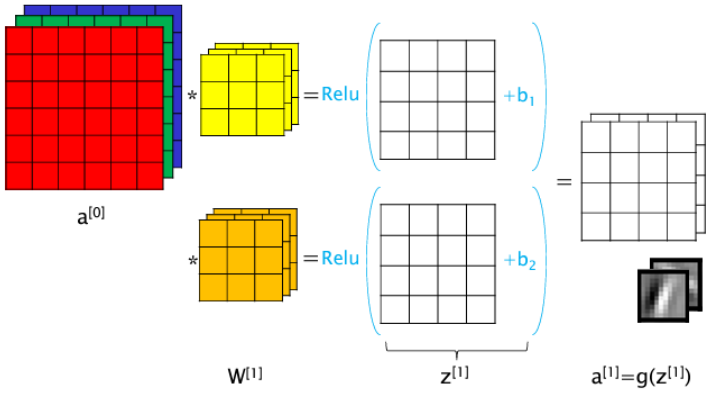


Figure 7.4: ReLU on feature maps

Example: Number of parameters in a CNN

How many parameters we have in a layer of a convolutional neural network which use *10 filters* $3 \times 3 \times 3$? For a single filter we have: 3×3 parameter for each 'sheet', there are 3 sheets for a filter, then for a single filter we have 27 parameters. Furthermore, there is another parameter for each filter which is related to the bias which is added before passing for the ReLU. Since we have 10 filters, the total number of parameters is

$$(3 \times 3 + 1) \times 10 = 280$$

7.1.3 Notation

Here we introduce some notation which will be useful in the comprehension of the examples of CNNs. Suppose we have the ℓ-th convolutional layer, for such a layer we can have some filters of dimension $f^{[l]}$ (they are square), we could apply some padding and/or stride, respectively $p^{[l]}$ and $s^{[l]}$. Then, the **input** will have dimension $n_H^{[l-1]} \times n_W^{[l-1]} \times n_C^{[l-1]}$, while **the output** will have dimension $n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$, where $n_C^{[l]}$ is the number of filters for the ℓ-th layer, while $n_{H/W}^{[l]}$ is equal to:

$$\left\lfloor \frac{n^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor \tag{7.1}$$

Each filter has dimension $f^{[l]} \times f^{[l]} \times n_c^{[l]}$, the dimension for an activation for a certain layer is equal to the output, since we have m examples we have m times the dimension of the output. All of the activations are indicated with $A^{[l-1]}$. The **number of parameters** for a layer is:

$$f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]} + n_C^{[l]} \tag{7.2}$$

Note here that a filter has a dimension that is equal to the dimension of the output of the previous layer. In the following an example is showed in which there are 4 convolutional layers:

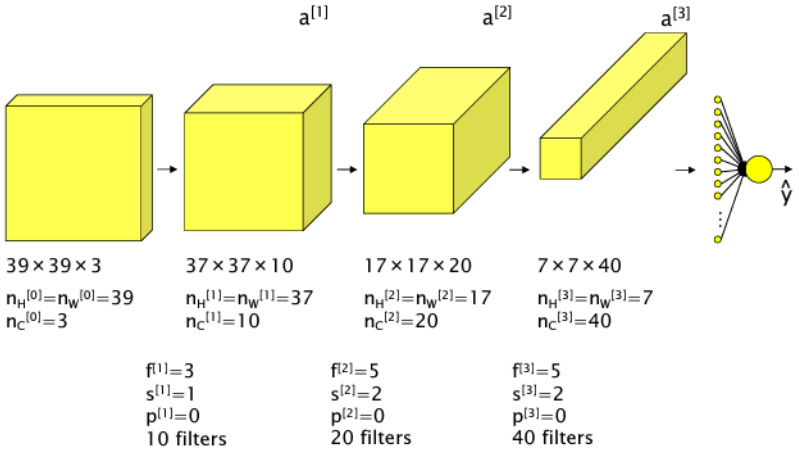
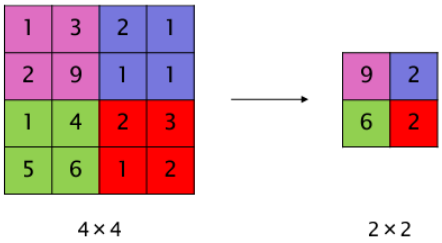


Figure 7.5: Example of a CNN (with all the dimensions)

7.1.4 Pooling layer: Max-Pooling

Once also the ReLU has been computed, the output is further modified. A **pooling function** replaces the output of the net for a certain location with a *summary statistic* of the nearby outputs. The most commons are the **max-pooling** and the **average-pooling**. The type of statistic to be used is a user-defined choice. The introduction of pooling bring with itself other two hyperparameters, the dimension of the *neighbourhood* on which the pooling is applied, the stride by which this occurs.



In this case the added hyperparameters are $f = 2$ (dimension of the sub-blocks) and $s = 2$ since at each pooling stage a cell is skipped.
Clearly the pooling reduces the dimension of the activations, but it is useful in order to summarize the information obtained at a certain stage. It is remarkable that different than the convolution stage, the pooling stage is performed separately for each channel of given activations.

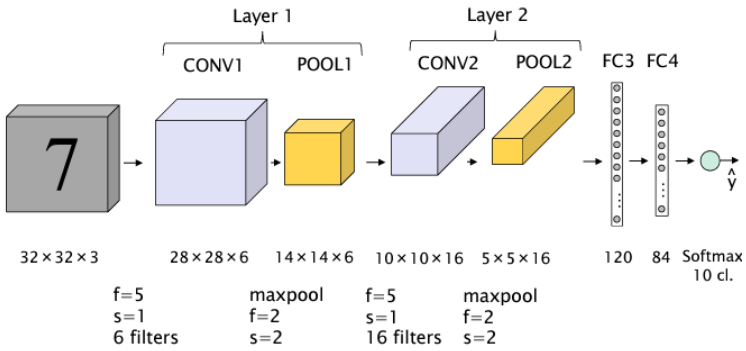


Figure 7.6: **LeNet-5**: CNN for digit recognition

7.1.5 Fully connected layer

The convolutional layers – made up of convolutional and pooling stage – works as feature extractors, finally at the end of the architecture some *fully connected layers* are present, they carries out the work of classifying a given example, then the last layer is a *softmax* one, which gives for each class a probability for the sample to be part of a certain class. Look at the Figure (7.1.4), for the last volume if we isolate a single cell, this is nothing but a linear combination of the parameters plus a bias, this is a neuron! Then the first fully connected (Dense) layer is nothing but the unrolling of all of the neurons contained in the last volume.

As first example, the CNN for the *digit classification is given* (see [4] for further information):

Note how the pooling stage does not change the third dimension (the depth of the volume) but only the height and width. As usual the last volume from POOL2 is unrolled in some computational units which made up the first layer for the fully connected part of the network. In the pooling stage there are no learned parameters since only a statistic summary is done on subregion of the activations.

7.1.6 Why Convolutions?

The convolutional stage is very used, mainly for two aspects:

- **Parameter sharing**: when a filter for a part of the image is used (eg. edge detection) probably it will be useful also for another part of the image; the same parameters used for different part of an image and (why not) for other images in which the same low/high level features, could be detected.
- **Sparsity of connections**: at each layer the outputs depend on a small number of inputs.

It could appear strange, but in a CNN the great majority of the parameters are concentrated in the fully connected layers! This is one of the reason why shallow fully connected networks performs very bad in terms of image classification.

7.2 Case studies and tasks

In the following some examples of CNN architectures are reported, for sake of completeness also the paper are cited.

7.2.1 AlexNet

This type of architecture was introduced in [3], and the objective was the image classification, differently from *LeNet-5* having 2 convolutional layers, this architecture contains 6 convolutional layers. The depth of the network was relevant for the obtained results which opened the road to a lot of studies on computer vision. The AlexNet paper is one of the most cited ones especially for the obtained results. Just for give an idea, the training set had 1.2 million images. It was trained for 90 epochs, which took five to six days on two NVIDIA GTX 580 3GB GPUs which had been working in parallel.

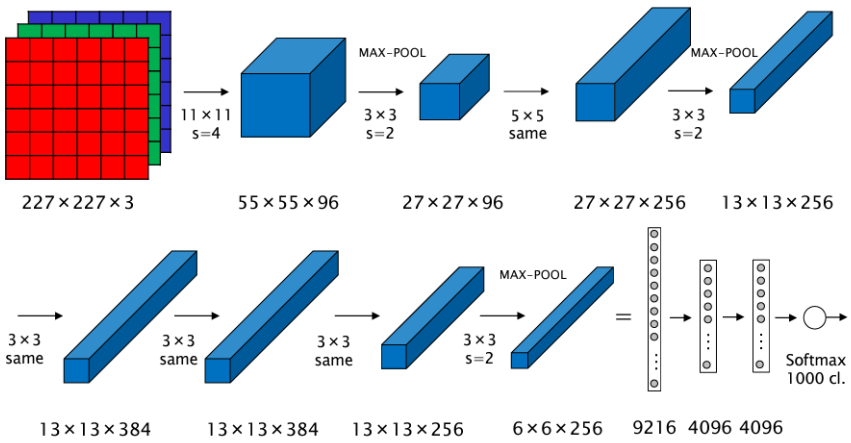


Figure 7.7: **AlexNet** architecture

7.2.2 VGG-16

They are named after the *Visual Geometry Group (VGG)* of the Oxford University. The full description of the net can be found in [25]. 16 is the number of its layers (13 convolutional, 3 deep), there are other VGG networks with a different number of layers.

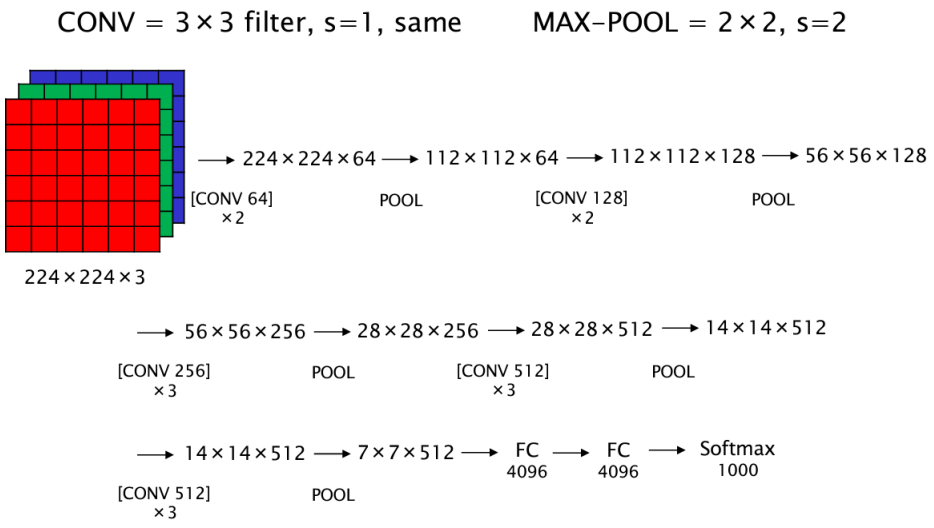


Figure 7.8: **VGG-16** architecture

7.2.3 Residual Network(ResNet)

Till now we have mentioned the structures of the DNN *LeNet, AlexNet, VGG-16*. In a similar way we could build up our deep neural network, ma in the practice they are hard to train. Residual Networks allow us to perform a more efficient training of them.
We have seen in the previous paragraph that DNNs suffer the problem of the *vanishing gradient*, we can say that data is disappearing through the network. Some reaserchers from Microsoft found that the split of a deep network into chuncks help eliminate much of this disappearing signal problem. In other words *ResNets* breaks down a very deep plain neural network into **small chuncks of network** connected each other by using *skip* or *shortcut connections*.

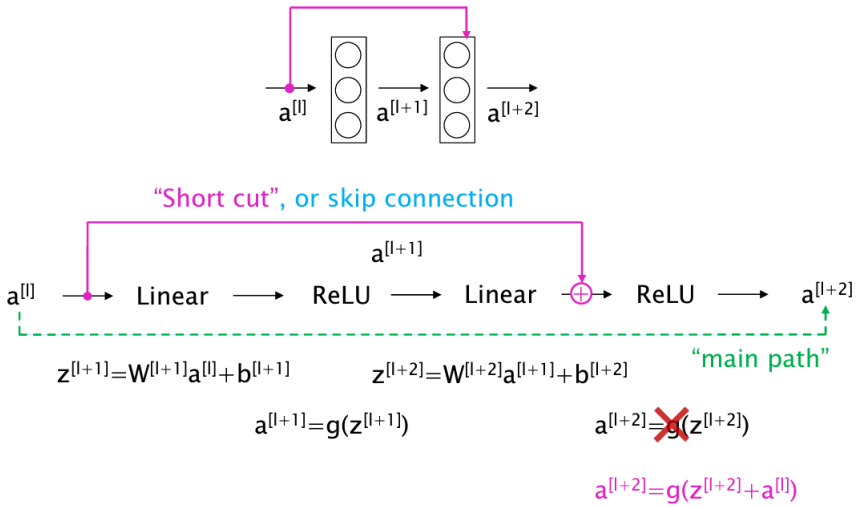


Figure 7.9: **ResNet** skip connections

In the figure above we can see the core constituting ResNets: **skip connections**. Roughly speaking the activation of the layer $l+2$ are computed using also the activation from the layer l. In this sense there is a skip through the layers. On such a type of architecture we have that the training error curve is how we can expect from the theory, differently from the *plain neural networks*. In the following we show an example of ResNet with 34 layers in which *skip connections* are used. The number of layers skipped is two, but "very surprisingly" the number of skipped layers can become another hyperparameter. In the original case presented in [2] the skip connection is from the level $l \to l+2$, that is the activation of the level l influences the ones in the level $l+2$.

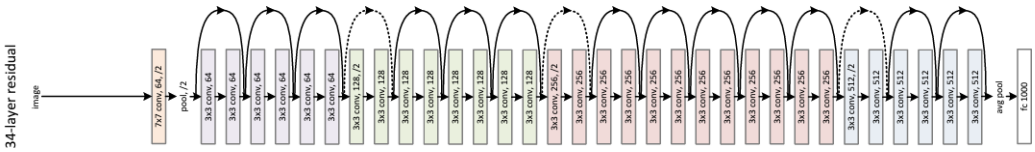


Figure 7.10: Example of 34-layers ResNet

7.3 1×1 convolutions

We have seen that in the convolutional stage some filters are applied in order to produce a linear combination of the input data. Could be strange, but there are some cases in which doing a 1×1 convolution is useful in order to reduce the number of computations, since the number of multiplications is directly proportional to the dimension of the filter!
Now, suppose we have an initial volume as the one represented in blue and we apply a filter as the one in yellow, we are doing nothing but the computation that occurs in a neuron (a part from the ReLU which is performed at the end): the $1 \times 1 \times 32$ filter applied to the volume gives us a linear combination of the input at the same Height and Width, but on different channels through the parameters contained into the filter. This is the reason why such a type of procedure is called *Network-In-Network* (see [5] for further details), this clearly reduces the number of multiplications that are performed.

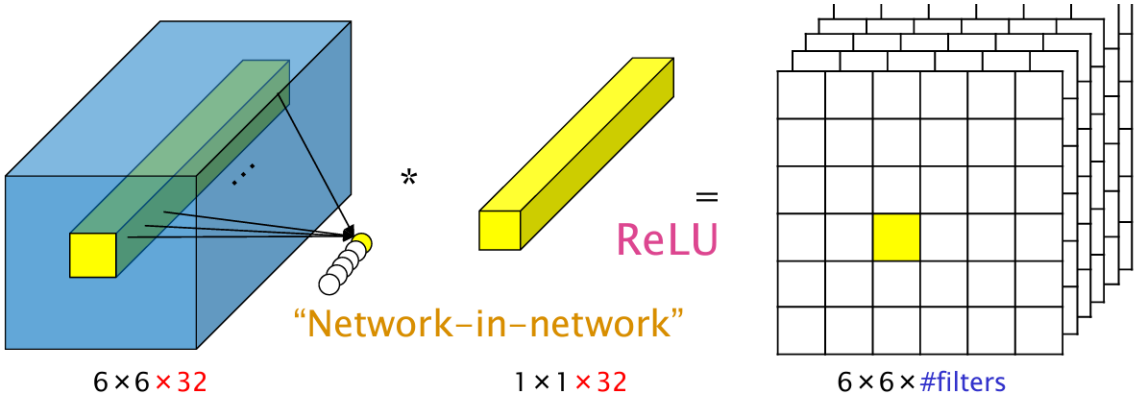


Figure 7.11: Network-In-Network and 1×1 convolutions

7.3.1 Inception: another DNN architecture

From the ideas presented in [5] and with the increasing in computation capabilities, in Google was created a new CNN architecture who has been called **Inception** (see [1]) (from a famous film from which they was inspired, in particular by an iconic phrase appeared in a meme *"We need to go deeper"*). Such a network has 22 layers, the great majority of them are inception modules, which are a culmination of the results presented in [5]. Such an CNN is one the most used architecture in computer vision.

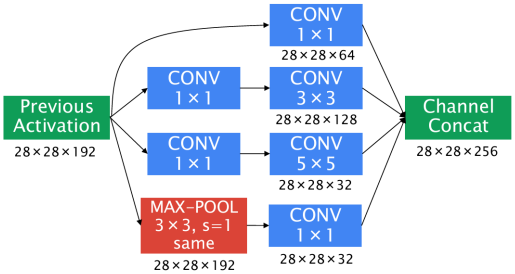


Figure 7.12: An Inception Module

In an *inception module* some 1×1 convolutions are used in order to reduce the number of computations of a 10 factor. The main motivation behind the Inception modules is that DNN

performs better if the number of layers is increased (that is the dimension of the network is increased).

At the end of this discussion about DNN a graph in which the perfromances of the presented architectures is compared to the *human/Bayes error*.

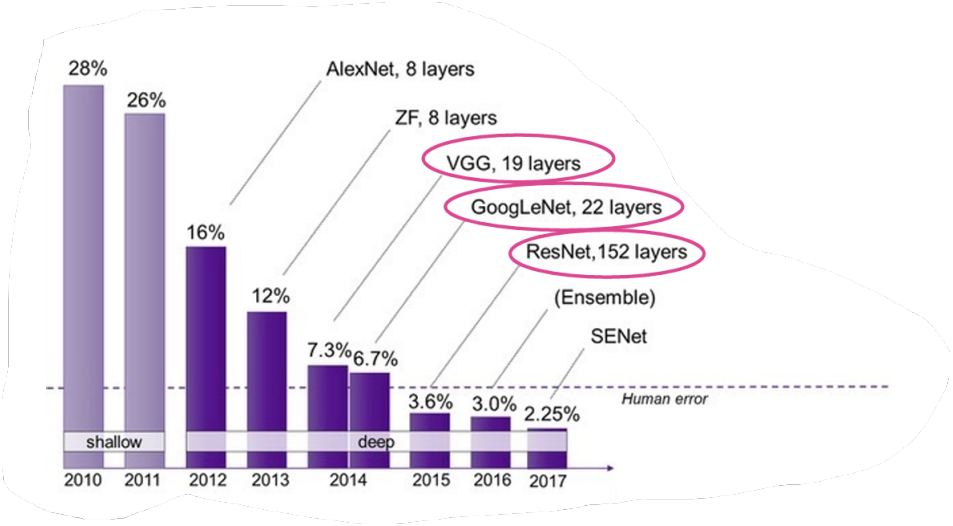


Figure 7.13: DNN performances vs human error

7.4 Other computer vision tasks

Till now we have introduced the main architectures of CNN, we have discussed the basic operations behind them as convolution and pooling. Beyond the classification task there are other more complicate tasks such as object detection, segmentation and semantic segmentation, human-pose recognition and so forth. There are other tasks that combine computer vision to language processing (see for example the *Image captioning*).

At the very beginning of this notes we have seen that the idea of *artificial intelligence* is not new at all! But today we are having an increase in the amount of data and computational capability that is favorable for the Machine Learning growth. Just to give an idea the architecture *LeNet-5* was trained on a computer with 10^6 transistors and no GPUs; *AlexNet* was trained on GPUs with an availability of 10^9 transistors (three order of magnitude bigger!) As far as the number of pixels is concerned we are talking about 10^7 in the first case and 10^{14} talking about AlexNet.

Chapter 8

Beyond image classification: Localization and Object Detection

In this chapter we will see other Computer Vision tasks which goes beyond classification: we are talking about *localization* (the task of localizing a given object by using a bounding box into an image) and *object detection* (the task of localizing multiple object within an image with their bounding boxes).

8.1 Classification with localization

Here we are talking about of an extension of the concepts we have presented for classification in order to give as output of the prediction for an object of a given class a **bounding box** for the object itself in term of box coordinates. This is of big practical interest: Autonomous driving, Industry 4.0, Pedestrian detection and so on.
Let us say that our final objective is, given an image, detecting multiple object inside it with both a **confidence score** and a **bounding box**. However in order to better and slightly understand, here we focus at first on a simpler task: **there is a single object to detect** within the image, the difference now is that we want to find the bounding box. How can we do it? It is quite "simple"!
Let us suppose that we have our CNN for the classification of five class of objects, say, *pedestrians, car, motorcycle, background*. In the last part of the network, we have some fully connected layers, that at the end will say by using a softmax some confidence scores. It is sufficient to introduce some new **numbers** for the network to learn which are related to the object bounding box (these can be the coordinate of the top-left most point and width/height of the box itself).

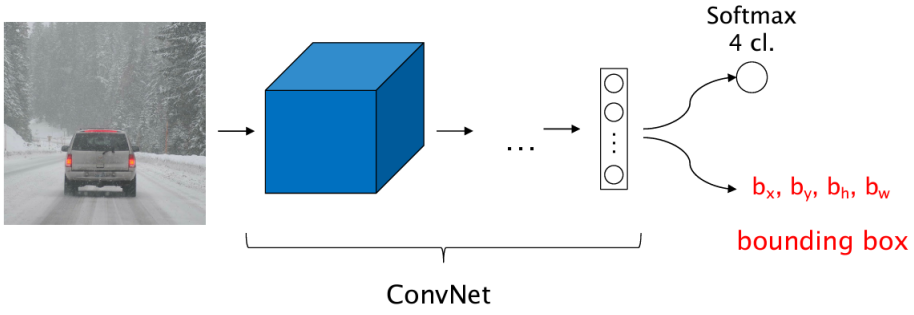


Figure 8.1: Classification + localization

The predicted output $\hat{y}$ of the ConvNet in this case is made up of $C + 5$ numbers where C is the number of classes. For example if we have 3 classes y (and then $\hat{y}$) is like:

$$y = \begin{bmatrix} p_c \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 0.5 \\ 0.9 \\ 0.3 \\ 0.5 \\ 0 \\ 1 \\ 0 \end{bmatrix} \tag{8.1}$$

The number p_c is *confidence level* used to indicate whether in the image is present or not an object of any class; this is followed by four numbers the normalized coordinates of the top left corner (or the center) of the bounding box (b_x, b_y) and its normalized width and height (b_h, b_w); the last three numbers tells us (using a one-hot encoding fashion) what is the class to which the localized object is associated.

One possible choice for the *loss function* is the following:

$$\text{Loss}(\hat{y}, y) = \begin{cases} \sum_{i=0}^{8} (\hat{y}_i - y_i)^2 & \text{if } y_1 = 1 \\ (y_1 - \hat{y_1})^2 & \text{if } y_1 = 0 \end{cases} \tag{8.2}$$

In order to give some *take away* information, we can say that the localization task can be performed using the architectures of CNN we have already seen with the only difference we are asking to the network to learn some other additive information related to the bounding box of the (potentially) classified object. To tell the truth we can do also other things: for example we can train the network so that it can detect *joint positions* and *landmark*.This is the right moment to highlight the fact that here we are combining two different but integrated tasks: **classification of images** and **regression of bounding boxes**. Now, what is the drawback? Often the labeling of the samples must be done "by hand" using some specific tools. As you can imagine, since a DNN needs a large amount of data this task is not so easy.

8.2 Object detection

Are we able to perform classification using DNN and the added information we have just introduced? Practically speaking, yes, but in what sense? Using one of the most famous computer science paradigms: *divide-et-impera*.
Let us suppose our (fine-tuned) ConvNet is trained for *localizing cars* and we want to detect within an image **multiple cars**. We can split the input image in a set of **crops** using a **sliding window** with certain dimensions. At the end of the day we have that some crops are containing the localized cars and we are done! The task of detecting multiple object within an image has been carried out by blindly applying the well-known techniques. Several problem occurs: what is the ideal dimension for the sliding window? What if I use a stride? Is it better, worse? More than this, this solution **does not scale** and cannot be use for large scale and real time object detection: identical computations are repeated over and over without reusing them!

8.2.1 OverFeat: a fully Convolutional Architecture

We know from the previous chapter that a generic ConvNet in its final part has a **classificator** which is constituted by several fully connected layers, we have seen also that different architectures have different *dense* layers with several units. By using 1x1 convolutions we can turn replace such layers with convolutional ones, clearly there is not an equivalence, but the architecture use convolutions *end-to-end* from the input to the output.

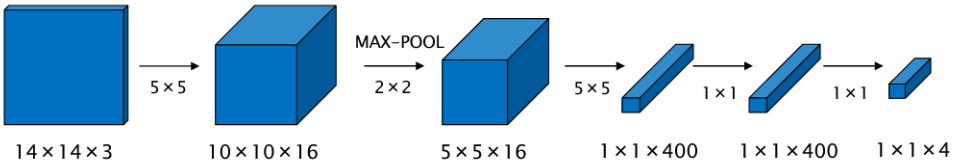


Figure 8.2: Fully Convolutional network

This idea is used in [22] as fundamental idea to implement a network that in an integrated way perform three tasks of increasing difficulty: *recognition/classification*, *localization* and *detection*. The main idea is **using the convolutional network** in a *sliding window fashion*. When convolutions are applied from the input to the output make the network producing a **map of class predictions** with one **spatial location** for each *window* (field of view) of the input.

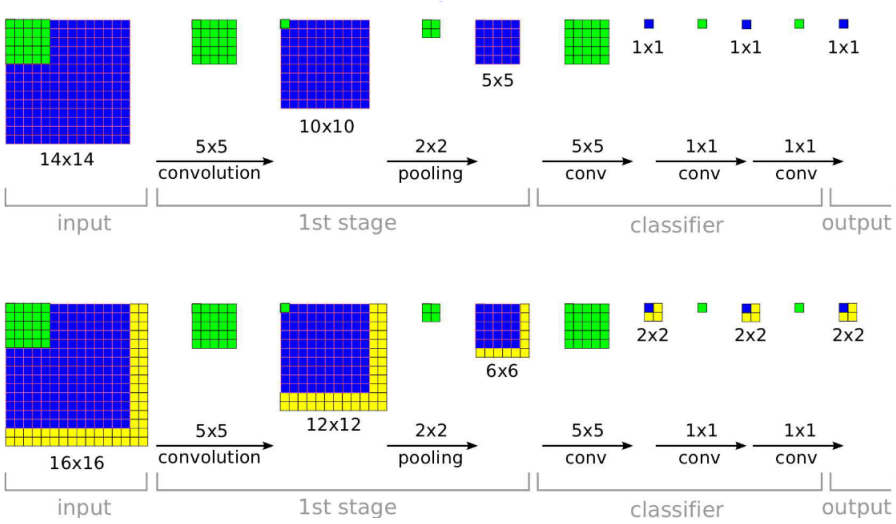


Figure 8.3: Convolutions and produced output maps

In the context of DNN the concept of *field of view* or **receptive field** is central. In particular it is defined as **the size of the region of the input that produces a feature** at a certain convolutional layer, since it is well known that by construction each one of the feature in the intermediate level depends only from a part of the input.

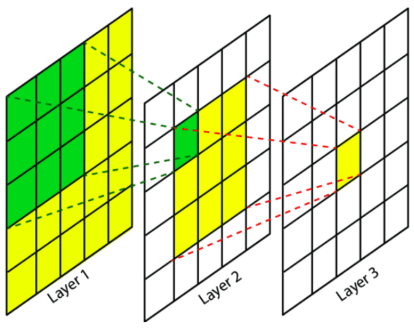


Figure 8.4: Receptive field (field of view)

In the case of *OverFeat* [22] the dimension of the receptive field, that play effectively the role of sliding window, is strongly connected to to the dimensions of convolutional and pooling filters. In the specific analysed case the dimension of the sliding window (in turn *receptive field*) is 14×14. The output is a map $2 \times 2 \times 4$ that are nothing but the information in (8.1) for each one of the four subregions of our image[1]. Clearly if we change the parameters of the filters (at different stages) also the receptive field is different. The figure 8.2.1 shows the results after the convolutions starting from a $14 \times 14 \times 3$ image and for a $16 \times 16 \times 3$, the yellow region is associated to the added computation with the increased input size, this is for underlining again the fact that using convolutions there are a lot of shared computations. We have cited [22] in order to do a first step toward the more sophisticate techniques for object detection. The paper explain very well what is the manner by which we are getting rid of all the useless bounding box, you can refer to it for further details. However, briefly speaking, differently from the naive approach that compute an entire pipeline for each one of the analyzed crop, here there is a sharing of computations among overlapped regions.

8.2.2 Region Proposal: R-CNN

In the previous chapter we have seen that a first more cleaver approach to object detection is using a fully convolutional network which improves a lot the original sliding window approach, however we can do better.
In 2014, in [11] was proposed a new architectures that completely eliminates the sliding window concept since it is inefficient and can produce a large amount of useless information. Such an architecture is made up of **three modules**:

- The first generates **region proposals** by using some *proposal methods* such as *Selective Search*[2]; a region proposal is a region of the input image which is likely to contain an object of a given class. Before passing to the second stage, each of one of the proposed region is **warped** in order to make it of suitable dimension for the following stage;
- The **second module** is a CNN, and in particular AlexNet or VGG-16 fine tuned on the ImageNet dataset. The output of this stage is a **feature vector** which representing the content of the region.
- The **third stage** is the one in which each feature vector is fed into a machine learning classifier trained on the class of interest (SVM is used).

In addition to the classification, of the RoI (Region of Interest) there is a part of the architectures which is devoted to the *regression of the bounding boxes*. In particular for each class there is a trained regressor that refines the initial boxes retrieved by using proposal methods.

After having predicted the bounding boxes for each region proposal, **non-max suppression** (see later) is applied in order to eliminate the 'non-optimal' boxes. In this way the **final object detection are obtained**.

[1] Note that we have potentially: identified multiple objects while classifying them and providing a bounding box. We assessed all of the three tasks: classification, localization and detection.

[2] **Selective Search**, just for give an example, operates by merging or splitting segments of the image based on various image cues like color, texture, and shape to create a diverse set of region proposals.

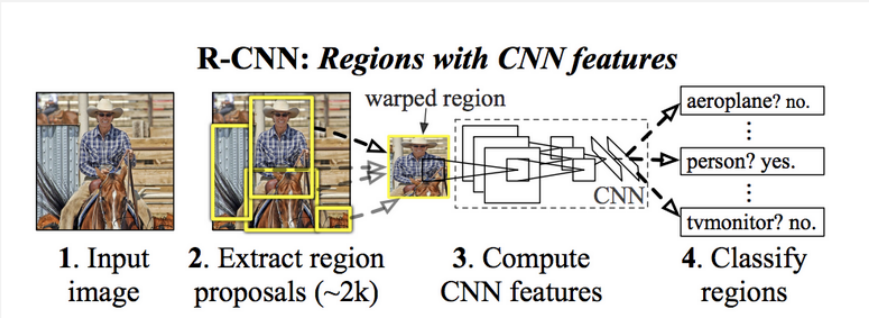


Figure 8.5: R-CNN structure

R-CNN features

Despite R-CNN are a step ahead in the field of Computer Vision, there are some non trivial drawbacks to be taken into account:

- Training is very slow (84h), requires a lot of space on the disk;
- Making a prediction is very slow, 47s for image (using as backbone[3] architecture VGG-16)

8.3 Fast R-CNN

We have seen that in R-CNN the main problem is that the **training is a multi-stage pipeline**, moreover it is expensive and it requires long time in order to correctly perform detection. In order to solve such problems, in [12] Girshick proposes a new method which has several advantages:

- Higher performances with respect to R-CNN;
- The training is done in a **single stage** while using a **multi-task** loss.

A *Fast R-CNN* network takes as input an **entire image** from which is extracted a common feature map and a set of **object proposal**. Then, for each object proposal a *Region of Interest (RoI)* pooling layer extracts a fixed length feature vector from the feature map. Each feature vector is fed up into a sequence of fully connected layers that finally branch into two output layers: one producing a softmax probability for the K classes plus one *catch-all* "background" class; the other is producing four real-valued numbers related to the **refined bounding-box** for that RoI. Note that the region proposals are obtained via non-neural methods, moreover the architecture related to the neural part is trained e2e (end-to-end). The following shows and summarizes the main features of *Fast R-CNN*:

How it is showed in the following histograms the bottleneck of Fast R-CNN is the RoI generation, without it we achieve a *near real-time* object detection.

Ren et al. (see [19]) solved this problem by introducing the so-called *region proposal network* that is completely based on ConvNets.

8.4 Faster CNN

Such an object detection system, called **Faster R-CNN** is composed of two modules related two networks which are trained jointly. The first is a *deep fully convolutional network* that

[3]In the field of Computer Vision and DNN, **backbone architecture** is generally, the term backbone refers to the feature-extracting network that processes input data into a certain feature representation.

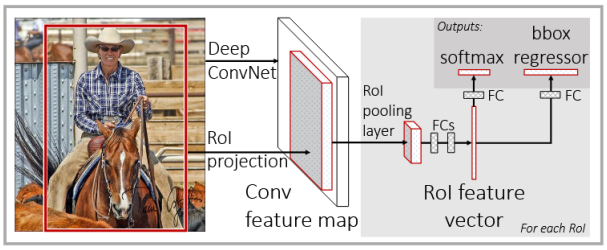


Figure 8.6: **Fast R-CNN architecture**. An input image and multiple region of interest (RoIs) are input into a Fully Convolutional Newtork. Each RoI is pooled into a fixed-size feature map and then mapped to a *feature vector* by fully connected layers

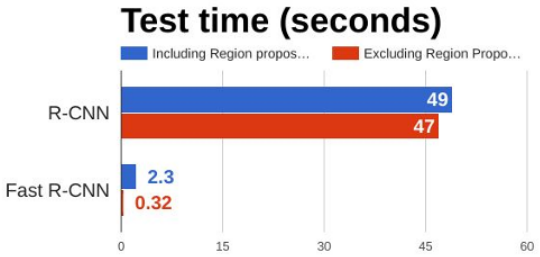
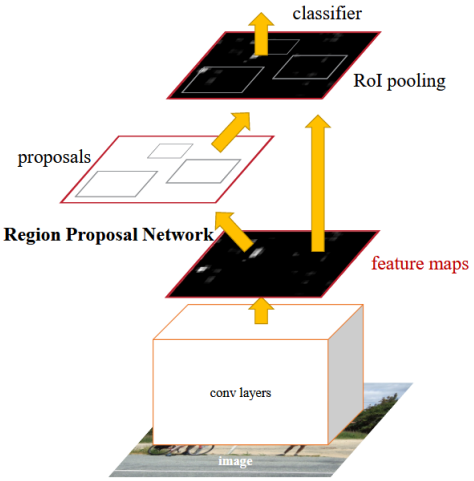


Figure 8.7: R-CNN and Fast R-CNN compared wrt Test time

proposes regions and works in a sliding-window fashion, the second module is the **Fast R-CNN** detector that substancially uses the proposed regions. In practice, the RPN (Region Proposal Netwoek) tells the Fast R-CNN where to look. The architecture of the network is presented in the figure that follows:



How it is written in the paper:

> *"Faster R-CNN is a single, unified network for object detection. The RPN module serves as the 'attention' of this unified network."*

Keep in mind that our goal is to share computation with a Fast R-CNN object detection network, both RPN and Fast R-CNN share common convolutional layers constituting the **backbone architecture**. Another detail is that for each region proposal several bounding boxes are predicted simultaneously. In conclusion the loss function is something like:

$$L = \alpha L_{cls} + \beta L_{reg} \tag{8.3}$$

where L_{reg} is the contribution for bounding boxes prediction, while L_{cls} is the contribution for the classification.

The Table 8.1 shows main differences between *Fast R-CNN* and *Faster R-CNN*.

Aspect	Fast R-CNN	Faster R-CNN
Region Proposal	External (e.g., Selective Search)	Integrated RPN
Speed	Slower	Faster
Architecture	Two-stage, with separate proposals	Unified, shared layers with RPN
Training	Partially end-to-end	Fully end-to-end
Performance	Good	Better (accuracy and speed)

Table 8.1: Comparison of Fast R-CNN and Faster R-CNN

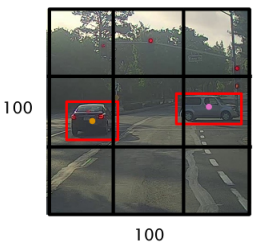
8.5 You Only Look Once (YOLO)

In this section we introduce YOLO (see "You only look once: Unified, real-time object detection" [18]) which is a **single shot method** that do not use region proposals, since it performs localization and classification at the same time. YOLO has the following features:

1. In order to handle complexity decomposes the image in a regular grid $S \times S$;
2. **If the center of an object falls into a grid cell, it is responsible for detecting that object**
3. Each grid cell predicts B bounding boxes and confidence scores for those boxes;
4. Each grid cell also predicts C conditional class probabilities, in particular there is **one set** of class probabilities, regardless of the number of boxes $B = S \times S$

8.5.1 Basics for YOLO

The output label y for the trainng of the model has a shape very similar than the one we have seen in Equation (8.1), clearly it depends also on the number of classes we have.[4] In order to better understand the main concept behind such a method, we use an example:



For sake of simplicity we use a simplified version in which we take a tensor input of $100 \times 100 \times 3$. Here $S = 3$, $C = 3$. Among all the cells of the image what happens is that almost all of them will have $p_c = 0$ since no object of the given classes is found. The coordinates of the center are normalized with respect to a certain cell, while width and height can be for sure greater than 1. Here we assume that a single bounding box per cell is predicted.

8.5.2 Overlapping objects: introduction of anchors

What happens if there is more than one object in a grid cell? You are supposed to be able to make **more than one prediction** per cell. The novelty of the anchors have been introduced from YOLOv2. Clearly the dimension of the output and the label used for training grows of a factor equal to the number of anchors. In particular it will be

$$S \times S \times (A \cdot (1 + 4 + C)) \tag{8.4}$$

[4]The most common example when YOLO is cited, is the one with *Pascal VOC* dataset where the output tensor is: $7 \times 7 \times 30$ since whe have $S = 7$ and $C = 20$ with C the number of classes.

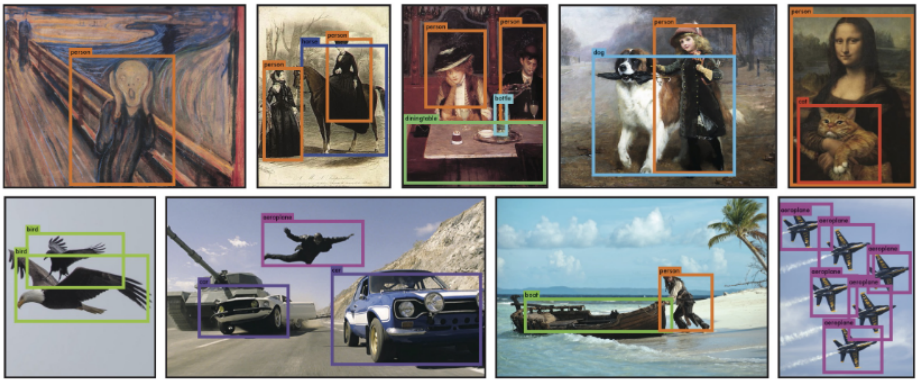


Figure 8.8: YOLO running on sample artwork and natural images from the internet

where A is the number of anchors. With the introduction of the anchors *each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU*. It is remarkable that **anchor boxes** are human-encoded priors on the size and aspect ratio of the objects. These are tipically defined as a list of pairs (height, width)[5]. The predictions are improved since the bounding boxes are not retrieved from scratch but using prior information.
Once the image is passed through the architecture a **post-processing stage** is required:

- For each cell grid, we have some bounding boxes with associated probability. The first step is getting rid of the low probability predictions (these will be associated to cell in which there are no object to detect), then **for each class** non-max suppression is used.
- The selected bounding boxes and associated labels are drawn on the image by using suitable tools/libraries[6].

Scaling Bounding Box Coordinates (Example by ChatGPT)

Let's assume that the bounding box coordinates are given in the original image (e.g., YOLO's default grid size of 416×416) and we need to scale them to a new image size (e.g., 1000×667). The original coordinates are:

$$x_{\min} = 116, \quad y_{\min} = 132, \quad x_{\max} = 241, \quad y_{\max} = 340$$

The new image dimensions are new_width $= 1000$ and new_height $= 667$.
We can scale the coordinates as follows:

$$x'_{\min} = \frac{x_{\min}}{\text{old\_width}} \times \text{new\_width} \quad y'_{\min} = \frac{y_{\min}}{\text{old\_height}} \times \text{new\_height}$$
$$x'_{\max} = \frac{x_{\max}}{\text{old\_width}} \times \text{new\_width} \quad y'_{\max} = \frac{y_{\max}}{\text{old\_height}} \times \text{new\_height}$$

Substituting the values:

[5]The anchor boxes are usually determined based on the statistics of the dataset-that is, by analyzing the ground truth bounding boxes and clustering their widths and heights using methods like K-means clustering.

[6]In particular, **bounding boxes** are drawn as rectangles using scaled coordinates, **labels** and **certainty** scores are shown with a background for ease of reading.

$$x'_{\text{min}} = \frac{116}{416} \times 1000 = 278.85 \approx 279 \qquad y'_{\text{min}} = \frac{132}{416} \times 667 = 211.97 \approx 212$$
$$x'_{\text{max}} = \frac{241}{416} \times 1000 = 579.33 \approx 579 \qquad y'_{\text{max}} = \frac{340}{416} \times 667 = 544.47 \approx 544$$

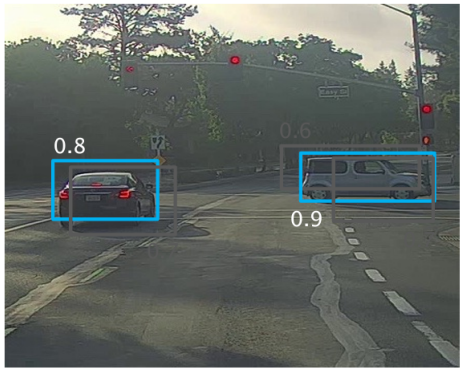
So, the scaled bounding box coordinates for the new image size 1000×667 are:

$$x'_{\text{min}} = 279, \quad y'_{\text{min}} = 212, \quad x'_{\text{max}} = 579, \quad y'_{\text{max}} = 544$$

These are the new coordinates that you can use to draw the bounding box on the new image.

8.6 Non-max suppression algorithm

Near to the end of this chapter, now we are going to better clarify the **non-max suppression** technique.
It is common that the object detection model gave more bounding boxes for a given identified object.



How can we choose the one that I will show onto the image? The procedure is the following, **for each class**:

1. Discard all boxes with low probability (eg. $p_c \leq 0.6$);
2. Pick the box with the largest p_c and discarding any remaining box with $IoU \geq 0.5$ having the same box output in the previous step.

8.7 Evaluating object localization and detection performance

At this point we need a way in order to evaluate how well a prediction has been done for a certain bounding box. A metric which is used is the **Intersection over Union (IoU)**, it is a measure of the overlap between two bounding boxes: the predicted one and the *ground truth* taken from the dataset. It is defined as the ratio between the intersection of the bounding boxes and the union of them.

Figure 8.9: **Intersection over Union (IoU)** definition

Usually we can say that a good prediction has a $IoU \geq 0.5$.

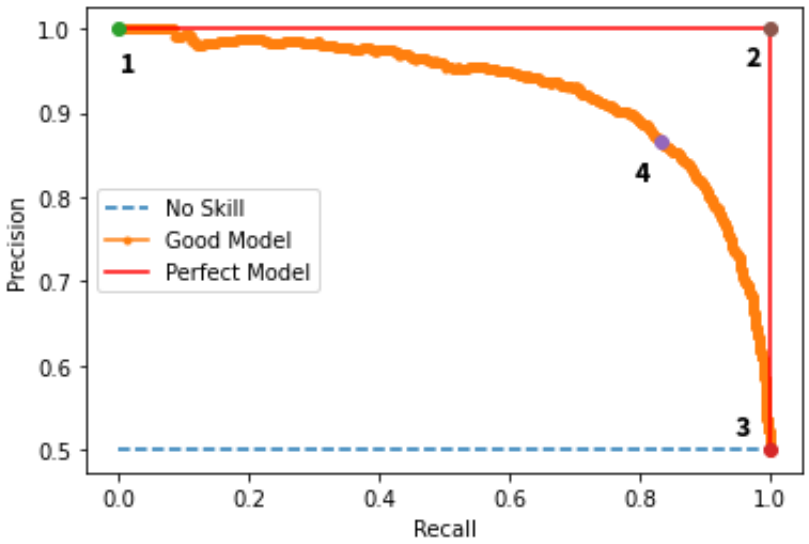


Figure 8.10: **Precision-Recall Curve** for a bad, good and perfect model

Since object detection includes a classification task we can build a **confusion matrix** like the one we introduced in the previuous chapters, in which also/only the IoU is taken into account. In particular:

- A **True positive** is counted in the case that there is a correct class prediction and IoU metric greater than 0.5;
- A **False positive** is counted if there is a wrong class prediction or $IoU < 0.5$
- A **False negative** is considered in the case a certain object is not detected.

According to such assumptions we are able to compute *precision* and *recall* for each class and then the F-measure can be computed. The *precision-recall curve* can be also drawn in order to select the best value for threshold depending on the user requirements.

A curve for each class can be drawn, and we can associate each one with another important metric: the *Average precision* that is nothing but the area under the curve. The object detectors are usually ranked using the **mean Average Precision (mAP)** which is the average AP over all classes:

$$\text{mAP} = \frac{\sum_{c \in C} AP_c}{|C|} \tag{8.5}$$

In some benchmarks mAP is computed at different IoU and then averaged again, this is the reason why sometimes it is denoted with mAP^{IoU}.

8.8 Final considerations

Object detection architectures are not *end-to-end* ones, like the models used for image classification: there are few base architectures but a lot of variations, some post-processing is required. Moreover when there are strong a-priori information are available on the target shapes the use of anchors is strongly recommended. In any case bounding boxes representation is not optimal since there can be irregular shapes, packed objects or rotated objects.

Chapter 9

Beyond classification and detection: Segmentation, Instance Segmentation, Neural Style Transfer

In this chapter we will consider other, even more complex, computer vision tasks including *semantic segmentation* (the task of classifying each pixel of an image) here encoder-decoder architectures are used, *instance segmentation* (combining object detection and semantic segmentation), *face recognition* using Siamese Network and finally the first step toward the world of generative AI concerning *neural style transfer*.

9.1 Semantic segmentation

Semantic Segmentation is the task of labeling each pixel within an image with a *category label* without differentiate instances/object of a certain class. Roughly speaking: I know that a certain pixel of a given image is associated to a cow, but I don't know that there is one or more cow in the image itself.
Let us introduce this topic by doing an important observation: since I want to classify each pixel of a given image, the size of the output (width, height) is supposed to be the same as the input. **What approach can we use?** Let us analyze the problem step by step, following the intuition and then introducing more complex reasonings in order to make the architecture scale up.

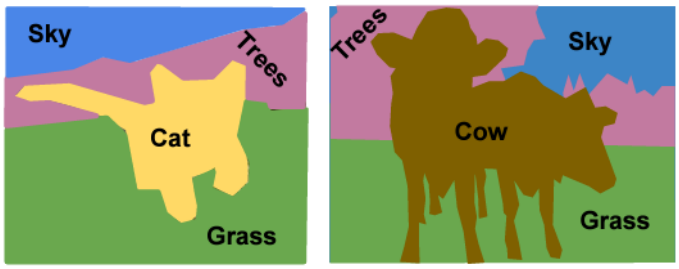


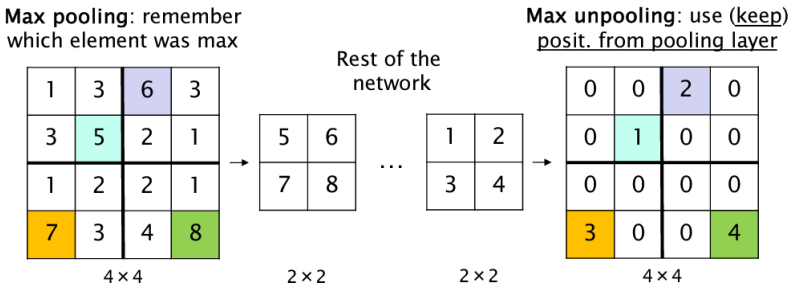
Figure 9.1: **Semantic segmentation** Since here I am not able to separate different instances of a given class, I cannot distinguish two cows within the image on the right

In the case we want to perform a *semantic segmentation* we can try to use the sliding window approach classifying the center pixel with a ConvNet. How you can image, this approach is extremely inefficient since the image must pass through an entire pipeline until all of its pixels

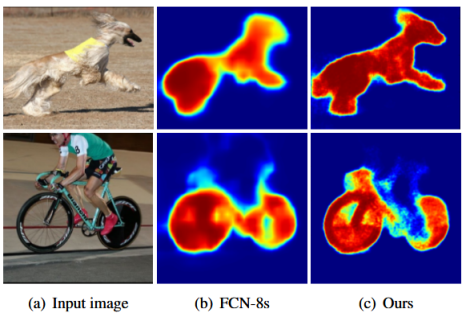
have been classified, without reusing the shared area among the several patches/window. Now, trying to follow the same path as the object detection, we can go toward a **fully convolutional approach**, since we have to fulfill the requirement on the shape of the output, one could propose to fix 2/3 of the dimension in a way that *height and width* can have the same shape across the convolutional layers. This solution does not scale on the input size, for this reason new models are used called **encoder-decoder** that downsample and then **upsample** in the second part of the network in order to match the input size by using: (i) *in-network upsampling* (**unpooling**), (ii) *learnable upsampling* (**deconvolution or transpose convolutions**) (both these aspects are explained in [15]).

9.1.1 In-Network upsampling: Unpooling

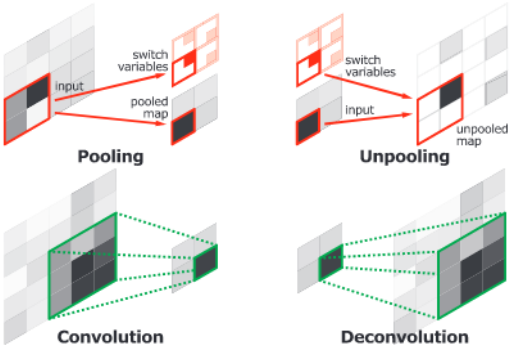
The *pooling* operation in convolutional networks helps us to filter **non-robust activation** by keeping a single (max/average...) representative value. However all the spatial information about such a value is lost. In order to solve such an issue the unpooling layer is employed into the deconvolutional part of the network (decoder) is used. The unpooling perform the **reverse operation of pooling** and reconstruct the original size of activations. Some **switch variables** are used in order to store the location of the maximum activation as showed in the following:



The first part and second part of such an architecture are simmetric, so that the pooled and unpooled layers are specular.



Activation maps from FCN and Encoder-Decoder architecture



Pooling/Unpooling and Convolution/Deconvolution

9.1.2 Learnable upsampling: Deconvolution

When the unpooling operation is applied we retrieve an **enlarged** but **sparse** map. The main role of the *deconvolutional part* of an encoder is to **densify the sparse activations** obtained by unpooling. How the **??** shows, the deconvolution operation maps a single input into multiple outputs. Several *learnable deconvolutional filters* are used which have a similar function with

respect to the convolutional one. Lower layers are likely to capture the shape of the overall object while the deeper one will the capture other *fine details*. In this way the decoder **directly takes class-specific shape information into account**.
The **transpose convolution** operation takes the input feature map and multiply a certain value for all the value contained in the filter so that the important information from the encoder are spread back; in the overlapped region takes as activation the sum of the numbers. In conclusion as in the case of "normal" convolution there are some hyperparameters (filter dimensions and stride).[1]

9.1.3 SegNet: Encoder-Decoder for Image Segmentation

SegNet (Badrinarayanan, Kendall, and Cipolla [7]) introduces a more robust way to segment a given image without the necessity to deconvolve it, on the contrary it uses 'normal' convolutional filters. Those that in Noh, Hong, and Han ([15]) are called *switch variables*, here are called *max-pooling indexes*. The underlying concept is the same: keep unchanged the position of the most important information.

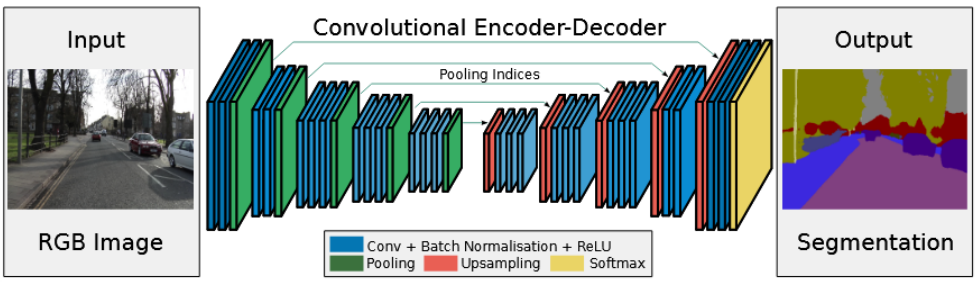


Figure 9.2: SegNet architecture

The performance of such a network is even better with the respect to the one presented before, due to the simpler method employed for upsampling. At the end of the decoder convolutional layer, logits are passed through a softmax layer in order to obtain **probabilities**. After that we compute the class by doing the max. **Each class is associated with a specific color**, this by doing post-processing is transformed into a **color-coded segmentation map**. Table 9.1 shows a comparison between the Deconvolution Network and Seg-Net.

9.1.4 Other Architectures for segmentation

U-Net: a framework for semantic segmentation in fine-grained domains

U-Net (Ronneberger, Fischer, and Brox Ronneberger, Fischer, and Brox) is a ConvNEt designed for *biomedical image segmentation*, but this is not a restriction since can be used also in other fields. Its name is due the **U-shaped structure** with two parts: encoder (pooling and

[1]Going more deeply, we can say that *"...unpooling and deconvolution play different roles for the construction ot segmentation masks. Unpooling captures example-specific by tracing the original locations (with strong activations) back to the image space. As a result, it effectively reconstructs the detailed structure of an object in finer resolutions. On the other hand, learned filters in deconvolutional layers tend to capture class-specific shapes. Through deconvolutions, the activations closely related to the target classes are amplified while noisy activations from other regions are suppressed effectively. By the combination of unpooling and deconvolution, our network generates accurate segmentation maps."*(from Noh, Hong, and Han [15]).

Feature	DECONVOLUTION NETWORK	SEGNET
Guiding Spatial Information	Switch variables (max-pooling indices)	Max-pooling indices
Upsampling Method	Unpooling + Deconvolution (transpose convolutions)	Unpooling
Convolution Type in Decoder	Transpose convolutions (learnable filters)	Normal convolutions (learnable filters)
Output of Unpooling	Sparse feature map	Sparse feature map
Densification Process	Deconvolution spreads activations and learns filters	Normal convolutions refine feature maps
Complexity	Higher (learnable upsampling)	Lower (fixed unpooling, separate convolutions)

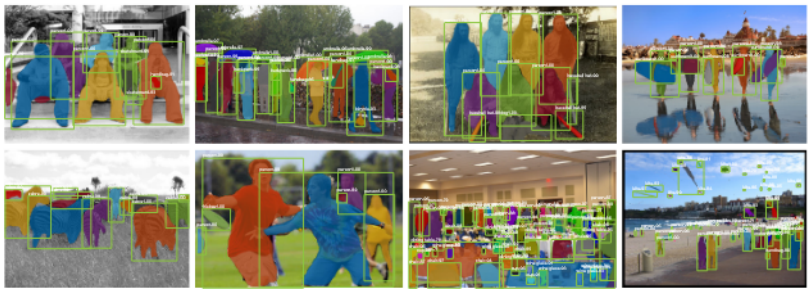
Table 9.1: Comparison between Noh et al. (Deconvolution Network) and SegNet.

convolution) and decoder (unpooling and transposed convolutions). Also here **skip connections** are used in order to preserve spatial information with the guide the upsampling process. Such a network works well also with *small datasets* and is effective for segmentation tasks with *fine structures and boundaries*.

E-Net: real-time semantic segmentation

E-Net (Paszke et al. [17]) is a neural network taylored for **real-time semantic segmentation** especially on **mobile** and **embedded devices**. Due to the context in which is used and the devices on which it has to run, the architecture is *highly optimized* introducing novelties (asymmetric encoder-decoder, dilated convolutions, pyramid pooling).

9.2 Instance segmentation

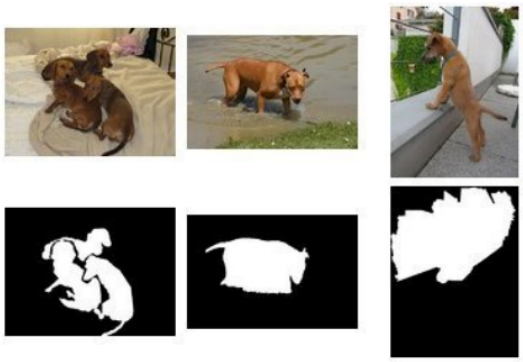


Instance Segmentation is an advanced deep learning task which combines *object detection* and *semantic segmentation*: here we want to label in the image with a category label, moreover we want differentiate instances of a given class.
The work in which such a technique was presented is called MASK R-CNN (He et al., 2017, [13]). This extends the FASTER R-CNN architecture by adding a branch for *predicting sementation*

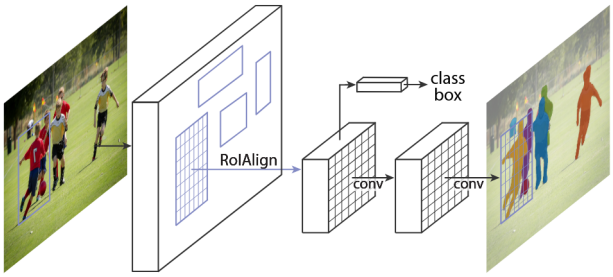
masks for each RoI. The main stages are kept, in particular the same Region Proposal Network is used and Fast R-CNN in order to classify each RoI.

9.2.1 Segmentation mask

The difference here is that in the second stage, in parallel, to bounding boxes and class prediction there is also a **binary segmentation map** for each proposal extracted by RPN.
A mask contains information about the spatial layout of an object, for this the mask prediction is done by using a fully convolutional network that preserve pixel-per-pixel features.
Keep in mind that the output mask is not a direct representation of pixels, it is a low-resolution mask of the object for the given RoI, then it is upsampled in order to meet the original dimensions (for example if the RoI spans an area of 56×56 and the extracted mask is 14×14, then it will be upsampled to 56×56). The upsampled mask contains continuous values ranging from 0 to 1, a certain threshold is used in order to keep/discard the values.



Segmentation mask examples



Mask R-CNN architecture

9.2.2 RoI-Align

Since a segmentation mask need to preserve spatial information, it is needed that the RoI features are faithfully coherent with the layout of a certain image.
In Fast R-CNN and Faster R-CNN, RoI pooling is used in order to extract the main features from the region proposals. An harsh quantization effect is introduced by RoI pooling which surely results in lack of important information that here are crucial! *RoI Align* method introduced in [13] avoid such effects by aligning the RoI feature maps.[2]

9.2.3 Traing Mask R-CNN

Similarly than the Faster R-CNN, here a *multi-task loss* is employed on each sampled RoI which has the following structure:

$$L = L_{cls} + L_{box} + L_{mask} \tag{9.1}$$

where L_{mask} is the cost function part devoted to the mask prediction, in particular a binary cross entropy loss is used and clearly a per-pixel prediction is done.

In conclusion, we can add a further detail. Following the same road we have done till now, we could ask to the network to learn other information like **joint positions**, clearly complexity is

[2]By using *bilinear interpolation methods*. See the article for more detailed information

added to the network structure. More complex and complete labeled datasets are used in order to train such even more complex architectures.

9.3 Face verification/recognition

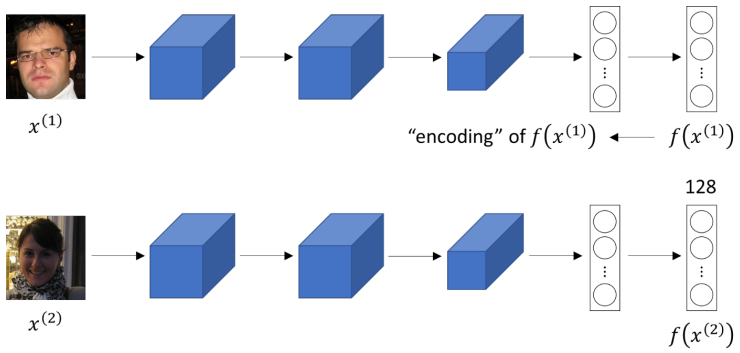


Figure 9.3: Face recognition

Here we introduce two similar computer vision task: (i) *face verification*, that deals with saying wheter a given image owns to a given person; (ii) *face recognition* where given a database of K people and given an input image, the task is saying given a new sample if that image is any of the K people in the database. The related work is *"FaceNet: A Unified Embedding for Face Recognition and Clustering"* ([21]). In this field is common to introduce the concept of **single shot learning** which is the task of learning from one example to recognize a given person again.

9.3.1 The need of a similarity function

Here we need a **similarity function** between two images, a sort of distance d(img1, img2) so that we can use it for both verification and recognition. In the first case we can use a threshold τ since the output is YES/NO, in the second part we can output the person identity whose image distance with the input is minimized.
In the context of ConvNets we know that passing through a generic image sample $x^{(1)}$ in the last layer before softmax is a vector of features (**embedding vector**), that we can call $f(x^{(1)})$. Now, given two images $x^{(1)}$ and $x^{(2)}$, we can compute a distance as:

$$d(x^{(1)}, x^{(2)}) = \|f(x^{(1)}) - f(x^{(2)})\|_2^2 \tag{9.2}$$

The network is supposed to learn parameters so that such a distance is small if the two images are related to the same people, otherwise it is larger.

9.3.2 Triplet loss

The so-called **triplet loss function** is more suitable for face recognition, the main motivation is that other loss functions try to project a given sample on a single point, the triplet loss – instead – tries to enforce a margin of difference.
In this context we want to ensure that an image x_i^a anchor of a specific person is closer to all other images x_i^p of the same people than it is is with respect to the other images x_i^n of other people. More specifically we want that:

$$\|f(x_i^p) - f(x_i^a)\|_2^2 + \alpha < \|f(x_i^a) - f(x_i^n)\|_2^2 \quad \forall \left(x_i^p, x_i^a, x_i^n\right) \in \mathcal{T} \tag{9.3}$$

Figure 9.4: Triplet loss objective

where $\mathcal{T}$ is the set of all the triplets in the training set and has cardinality N. The loss function to be minimized in this context is:

$$\mathcal{L} = \sum_{i}^{N} \left[\|f(x_i^p) - f(x_i^a)\|_2^2 - \|f(x_i^a) - f(x_i^n)\|_2^2 + \alpha \right] \tag{9.4}$$

The triplets must be chosen in a way that the inequality is not satisfied in the great majority of the cases. In the cited paper [21] there is an entire section devoted to how properly select the triplets of the set $\mathcal{T}$.

9.3.3 Siamese Network

A **Siamese Network** is a class of neural network architectures that **contain two or more *identical* subnetworks**, in the sense they share the same configuration, but also the same set of parameters. Siamese networks learn a similarity function, and they can be used together with binary classification to learn similarities. Here we have an example:

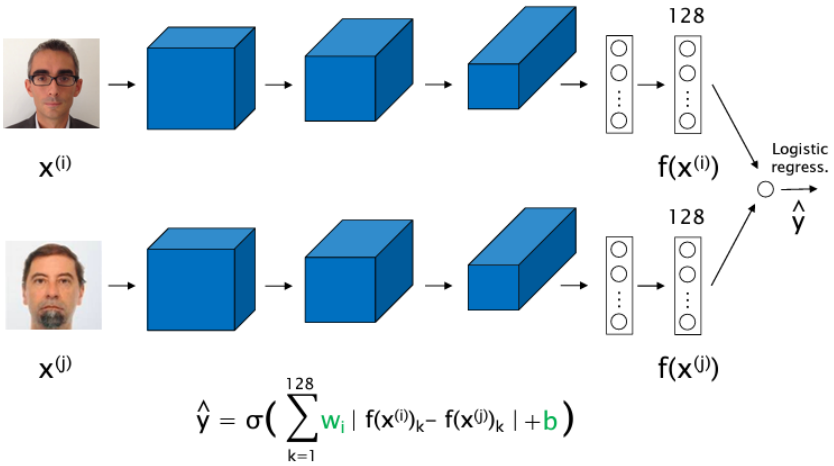


Figure 9.5: Siamese network

The parameters w_i and b are learnt for the pair of networks. The single network is trained using the triplet loss that – summarizing – maps each image of the input in a space of embeddings where similar faces are closer than different faces.

9.4 Neural style transfer

Now we present a technique which is the first step toward the generative AI techniques. The main reference for this part is *"A Neural Algorithm of Artistic Style" [10].* Essentially the objective here is to generate a new image that could have:

- The **content (C)**, that is the structure, of a certain image;
- The **style (S)**, from another image.

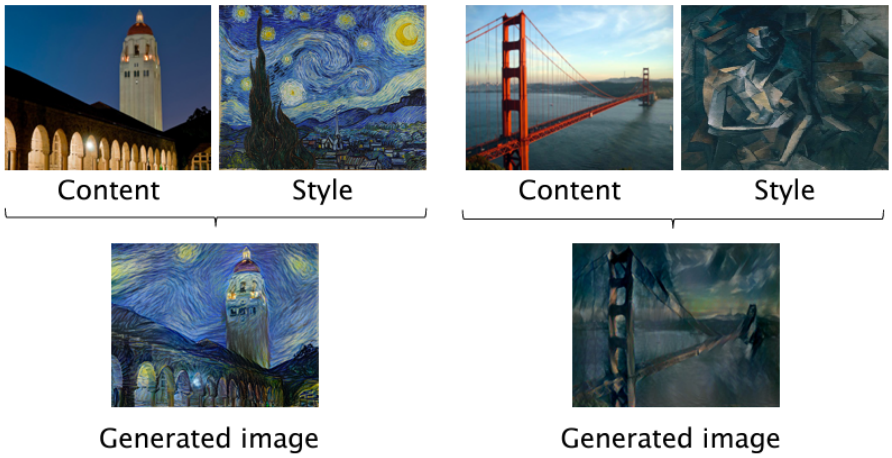


Figure 9.6: Neural Transfer Style

Before entering into more details, it is crucial dedicate few words on the type of information a convolutional network is able to learn at different layers. We can say that high level details about the structure (**content**) is learnt at deeper layers while *low level details* like textures and patterns related to the **style**, are mapped into the initial layers.

The cost function to be used in such a context is a *multi-task* one like:

$$J(G) = \alpha J_{\text{Content}}(C, G) + \beta J_{\text{Style}}(S, G) \tag{9.5}$$

where G is referred to the **generated image**. Let us see more deeper what is the structure of the functional part related to the content and to the style.

9.4.1 $J_{\text{Content}}(C, G)$: content cost function

In order to compute the cost function we sample the activations of the network at a certain layer l, let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of a certain network structure computed on the content image and on the generated image. Such activations are similar if both images have similar content. Then, the functional related to the content is:

$$J_{\text{Content}}(C, G) = \frac{1}{2}\|a^{[l](C)} - a^{[l](G)}\|_2^2 \tag{9.6}$$

9.4.2 $J_{\text{Style}}(S, G)$: style cost function

As it us stated in [10] the style of an image can be computed as the existing correlation among different channels of a certain layer l. These can be expressed in term of the *Gramian matrix* of the layer l itself. For the style image, I take the activations of the layer l and I compute the matrix $G^{[l](S)}$ where the entry G_{ij} is:

$$G_{ij} = \sum_k F_{i,k} \cdot F_{j,k} \tag{9.7}$$

Practically speaking given the activation tensor of a certain layer $G^{[l](S)}$ can be computed in the following way:

- Given the tensor $n_H \times n_W \times n_C$, we have to perform the reshape $n_C \times (n_H \times n_W)$ by unrolling in a row vector the matrix associated in a channel and then composing them by row.
- The matrix $G^{[l](S)}$ can be computed by multiplying this intermediate matrix by its transpose.

At this point, we are to give (approximately) the structure for the style cost function:

$$J^{[l]}_{\text{Style}}(S, G) = \|G^{[l](S)} - G^{[l](G)}\|^2_F \tag{9.8}$$

Since for the style several layers are considered the final shape is:

$$J_{\text{Style}}(S, G) = \sum_l \lambda^{[l]} J^{[l]}_{\text{Style}}(S, G) \tag{9.9}$$

where $\lambda^{[l]}$ are the weights for the different layers.

9.4.3 Generating the output image

The output image is generated in the following way:

1. Given C and S, initialize G randomly or starting by the content image;
2. Gradient Descent is used in order to minimize the cost function $J(G)$

An example is shown in the following figure:

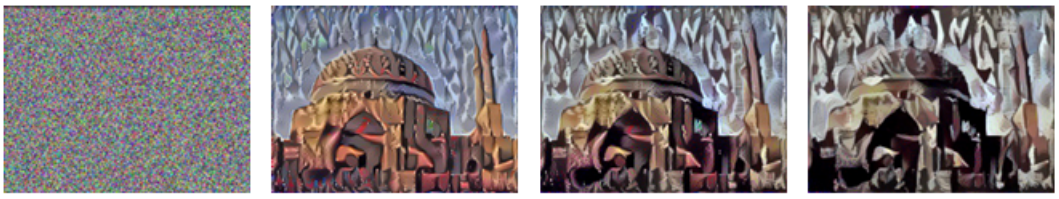


Figure 9.7: Generating the output image

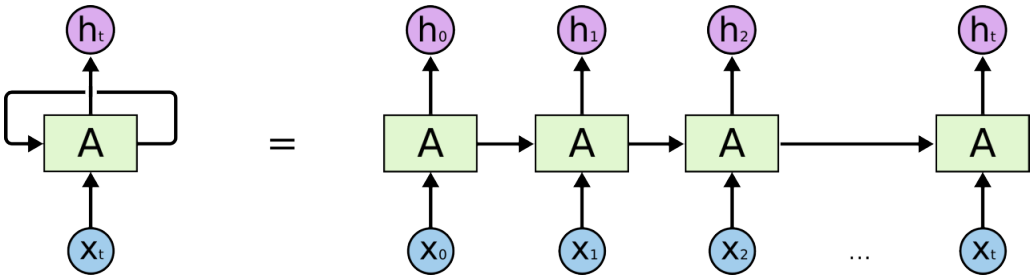
9.4.4 Final comments

Note that an architecture like VGG-16 can be used in order to perform the task, we do not absolutely care about the classification that such a network provides us, since we use it only as a **feature extractor**. In the practice, given the content and style images, we pass them through the network in order to sample the activations we need in order to compute the loss. After having suitably initialized the generated output, we pass it many times through the network updating it according to the partial derivatives of $J(G)$.

Chapter 10

Sequential and Attention models: RNN, LSTM, GRU, Transformers

In this chapter we will discuss about some tasks that according to their features requires special neural network architectures that are something different with respect to the model we have already seen talking about MLP and ConvNet for computer vision activities. We are talking about *Recurrent Neural Network (RNN)*, such models are used in order to perform computation on *time series data*. **Time** is the new component to handle. The related tasks include: speech recognition, music generation, DNA sequence analysis and so on. After some prerequisites, we are introducing RNN and more robust architectures (*LSTM, GRU*). Finally the state-of-art architecture for NLP is analyzed (*Transformer*).



10.1 Notation

In order to introduce some **notation**, we use a practical example. In the field of information extraction in the context of *Natural Language processing (NLP)*, there is a sub-task which is called **Named entity recognition**, this deals with the classification of the name appearing in a sentence according to predetermined categories. Suppose you want to recognize the person name in the sentence:

Harry Potter and Hermione Granger invented a new spell

This is a sequence of words constituting a phrase. We need a notation in order to handle the concept of *sequentiality*. One possibility is to use for the words, which will be the input of our models (anyhow they are made), the notation

$$x = \{x_{(1)} \quad x_{(2)} \quad \dots \quad x_{(T_x)}\}$$

Indicating with an index the *time at which the word appear in the sentence*. The size of the input sample x is T_x. The same reasoning holds for the output y, which is suggesting us, the words related to name of person. In the specific case we will have

$$y = \{y_{(1)}, y_{(2)}, \ldots, y_{(T_y)}\} = \{1, 1, 0, 1, 1, 0, 0, 0, 0\}$$

Since *Harry Potter* and *Hermion Granger* are the names in the given sequence. It is not said that T_x and T_y are of the same length. The concept of **time** is a novelty with respect to the other tasks we have seen. Sequentiality makes necessary the introduction of new aspects we have not considered till now. Let us start!

10.2 Representing words

It is not a novelty if we say that Neural networks manage effectively numbers by doing several forms of computation in order to perform their task. Well, even in the presence of sequential data, we have to map them in some "numeric space", first of all the **words**.

When we are dealing with NLP, mostly, there is a **dictionary** with a great number of words which can be used in the analysis. Then, if we have examples made up of phrases, each word is mapped into a *sparse vector* in which **only the number at the position where the word itself is located in the vocabulary is one**, all the other numbers are 0. For this reason such an encoding is called **one-hot encoding**. Such a work is carried out after that the sentences have been **tokenized**[1].
To tell the truth the architectures we are going to see, do not take into account this sparse representation, at least directly. Differently, often the words during the training of *language models* are mapped into an *hyperdimensional dense space* in the so-called **word embeddings** which are succint synthesis of the general meaning for a certain word. Such encoding can be pretrained or fine tuned or made by scratch in some cases, being included into the *trainable parameters*. Anyway, *one-hot encodings* are important since they are fed into an *embedding lookup module* which will provide the inputs x to the NN. To conclude this part, let us provide an example. We want the one-hot encodings for the word *cat*, while the voabulary is:

$$\text{Vocabulary} = \{a,\ aaron,\ ...,\ aerospace,\ cat,\ ...,\ zulu\}$$

The one-hot encoding is:

$$[0,\ 0,\ ...\ 0,\ 1,\ ..., 0]$$

10.3 Recurrent Neural Networks (RNN)

10.3.1 Motivations for introducing a novel architecture

Everytime we have to introduce a new architecture, it is quite natural asking ourself: *Can we use, instead, the architectures we already have?* There are several reasons for which the answer is NO.

1. In models dealing with sequential data, the **size** of input/output **can be different** in different samples.

[1]Sometimes, for certain types of computation, the **stemming** is preferred; this is the process by which each word is recasted to its *root form*.

2. Standard NNs do not share the features learned across the different position of the sequence;
3. Standard NNs, first of all, do not include *memory mechanisms* which are fundamental here, due to the presence of *sequentiality*

10.3.2 Recurrent neurons and layers

Up to now we have seen models where the outputs flowed only in one direction: forward. A **recurrent neural network** has more or less the same structure of a Feed-forward Neural Network, except the fact it have **backward connections**.
The simplest possible RNN is the one made up of *one neuron* that receive the input, produce the output and send its output, or in more complex situations, its **hidden state**, back to itself. When the first input is received this output/hidden state is usually initialized to 0 since the network has not already produced any output. As showed in Figure 10.1 the recurrent neuron can be represented *against the time axis* in the so-called **unrolled representation** (it's the same recurrent neuron once per time step).

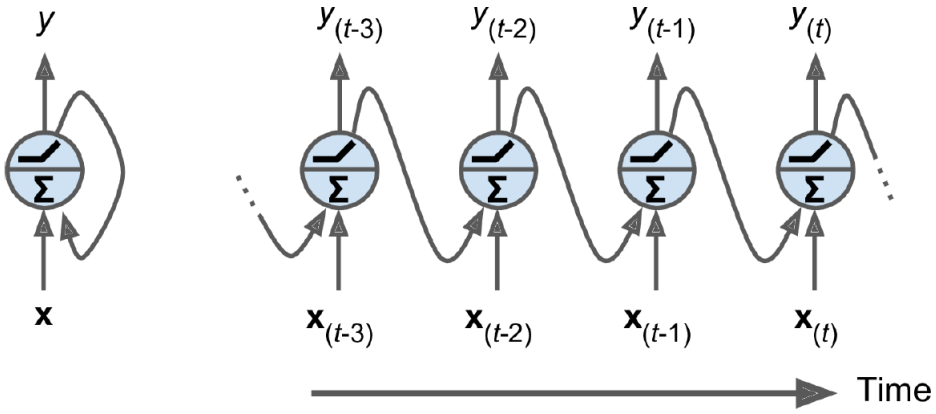


Figure 10.1: Recurrent neuron

Here, it is even more clear that for each time instant the neuron, receives not only the input x but also another information coming from past computations. The output of such a *tiny network* is simply a scalar.
It is quite simple to extend the reasoning we have done for a **layer recurrent neurons** where each neuron receives the input and the **backward information**. Here the output is a vector, since there many neurons. A *recurrent layer* together with its unrolled representation is showed in the Figure 10.2.

Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, this is the reason why we refer to recurrent architecture by using the term **memory cell**. In the examples we have analyzed of recurrent neuron and layer, we have assumed that the backward information was the output, in more complex architectures this is not the case, but we call it **hidden state** (see section 10.3.2), we are indicating it with the notation $h^{<t>}$.
Each neuron has **three sets of weights**:

1. $\mathbf{w}_{hx}$ from the input to the hidden state;
2. $\mathbf{w}_{hh}$ from one hidden state and the following;
3. $\mathbf{w}_{yh}$ from the hidden state to the output

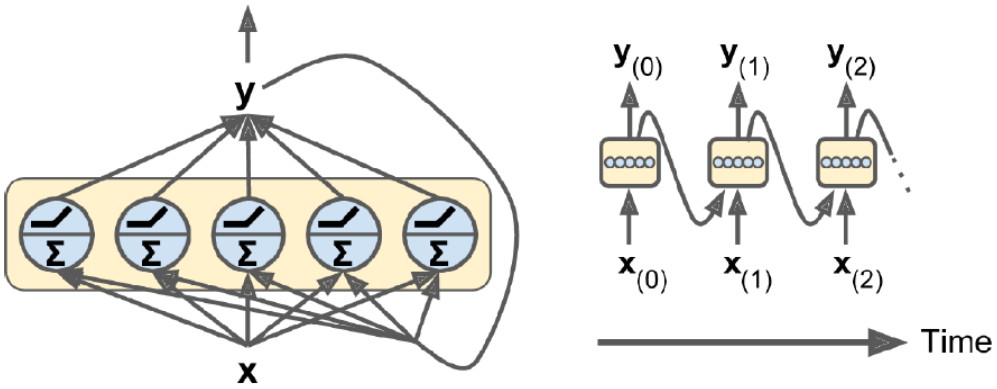


Figure 10.2: Recurrent layer

Since we have multiple neurons we can group such weights in matrices, like we did for feedforward architectures. Then, we have $\mathbf{W}_{hx}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{ya}$.

What about the training of an RNN?

We have to use a trick which we have already seen in some sense, that is, we have to unroll the network through the time and then apply, after the forward pass, the backward propagation. Let us clarify in details these aspects.

Forward propagation

The following are the steps to perform in order to carry out the **forward propagation**, the most general case is considered when the hidden state h is different with respect to the output y. The equations for the hidden state and for the output are respectively:

$$\mathbf{h}_{(t)} = g_h(\mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{W}_{hx}\mathbf{x}_{(t)} + \mathbf{b}_h) \tag{10.1}$$

$$\hat{\mathbf{y}}_{(t)} = g_o(\mathbf{W}_{yh}\mathbf{h}_{(t)} + \mathbf{b}_y) \tag{10.2}$$

where g_h, g_o are the activation functions related to, respectively, the hidden state and the output, while $\mathbf{b}_h$, $\mathbf{b}_o$ are the bias vectors. A simplified notation can be used if the two matrices of the hidden state are collapsed into

$$\mathbf{W} = [\mathbf{W}_{hh} \quad \mathbf{W}_{hx}]$$

so that the Equation (10.1) becomes:

$$h_{(t)} = g_h(\mathbf{W} \cdot [\mathbf{h}_{(t-1)} \quad \mathbf{x}_{(t)}]^T + \mathbf{b}_h)$$

If we had considered all of the examples of the mini-batch on which we compute the forward pass the all of the involved quantities would have been matrices, in particular $\mathbf{H}_{(t)}$, $\mathbf{X}_{(t)}$ and $\mathbf{Y}_{(t)}$.

Backward propagation

The strategy according to we can update the weights in an RNN, is called **Backpropagation through time (BPTT)**. Like in a regular BP, there is first a forward step through the unrolled network, then a loss function is computed according to the outputs. In particular we have that the loss $\mathrm{J}(\hat{y}, y)$ will be:

$$\mathrm{J}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathrm{Loss}_{(t)}(\hat{y}_{(t)}, y_{(t)}) \tag{10.3}$$

The derivatives (gradients) of such a cost function are computed and backpropagated through the unrolled network. In some cases, the cost function could depend only on a subset of outputs. It should be clear that, since the unrolled network is nothing but the same architecture repeated over time, the weights are the same for each *time step* or *frame*. The Section 10.3.2 shows the BPTT process, the dashed arrows represent the forward pass, the solid ones the backward pass. Note that here is considered the case where the cost function accounts only for three out of five of the outputs, moreover the cost function is indicated with C while considering together the whole mini-batch.

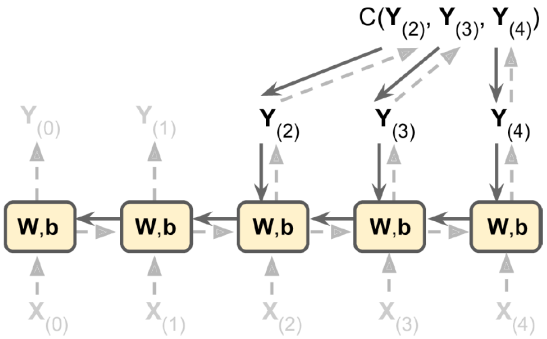


Figure 10.3: *Backpropagation through time*

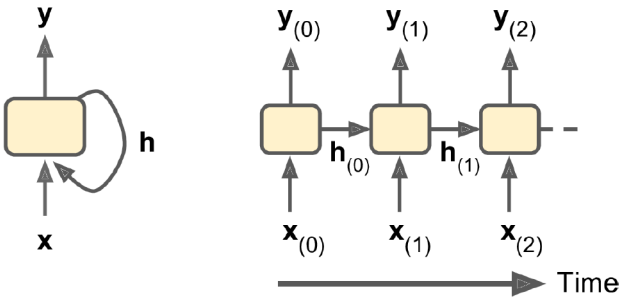


Figure 10.4: *Hidden state ≠ Output*

10.3.3 RNN architectures

In the introduction we have mentioned the fact that T_x could be different than T_y, in the great majority this is the case. Several combinations are possible. The Section 10.3.3 shows the different cases, including the encoder-decoder architecture.

Sequence-to-sequence

In this case there is a sequence as input and a sequence for output. Such a network is useful for example for *predicting time series* such as stock prices.

Sequence-to-vector

In this case a sequence of data is fed into the network, but the output are all ignored except the last one. This is common when an RNN is used for *sentiment analysis*. For example the output is a sequence of words constituting a film review the output is a score between -1 [hate] and 1[love].

Vector-to-sequence

When you feed the network with the *same vector* over and over and the output is a sequence, you build a *vector-to-sequence* RNN. An example is the **image captioning** (on which we dedicate a section) where the input is, for example, the feature vector coming from a ConvNet, the output is a sentence (sequence of words) containing a description for that image.

Encoder-Decoder

An **Encoder-Decoder RNN** is a quite particular architecture made up of: (i) a sequence-to-vector structure followed by a (ii) vector-to-sequence structure. This is one of the first architectures used in the field of *Neural Machine Translation (NMT)*. Here, the encoder (sequence2vector) is converting the input sequence into a **single vector representation**, this

is the input of the decoder (vector-to-sequence) part which is decoding into a sentence in another language. This works much better than translating a sentence on fly by using a single sequence-to-sequence architectures, since the meaning of the first word could depend from the following. Usually the architecture is a little bit more complex with respect to the one showed in Section 10.3.3 how we will see.

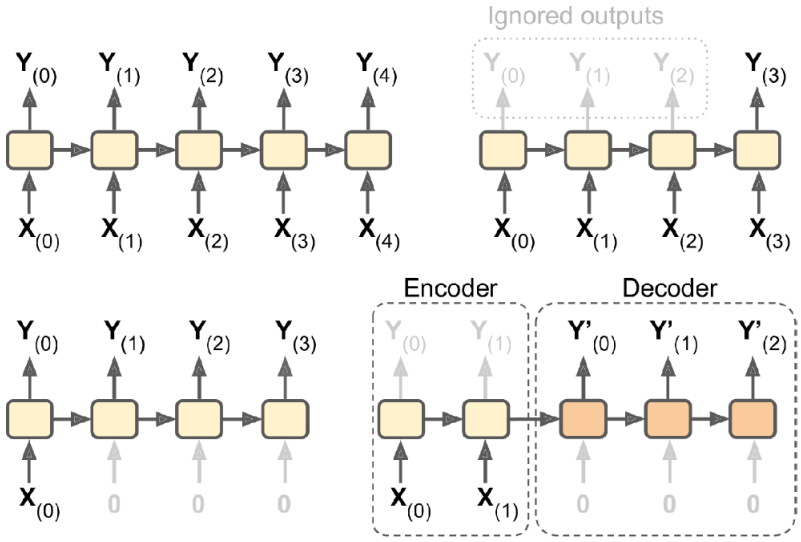


Figure 10.5: RNN architectures

10.3.4 Bidirectional RNN

At each time step a usual recurrent layer only looks at past and present inputs before generating its outputs. In some applications it is preferable that to look ahead the next words before giving the output. For example in order to well encode the word "queen" in the sentences "The Queen of the United Kingdom" and "The queen of the hearts", we have to look ahead the other words since the meaning of the word is completely different in the two situations. In order to solve this problem we have to run in parallel **two recurrent layers** fed with the same input sequence, the difference is that one is reading from the beginning to the end, the other from the end to the beginning. Then, "simply" the output is *combined* at each time step. The resulting architecture is the so-called **bidirectional RNN**.

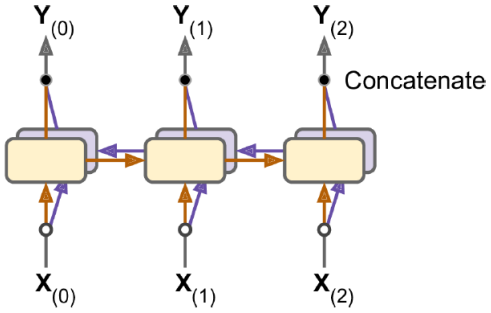


Figure 10.6: Bidirectional recurrent layer

10.3.5 Deep RNN

It is a common pratctice to stack several RNN cells, what you obtain is a *Deep RNN* architecture. The figure shows a deep recurrent network together with its unrolled version.

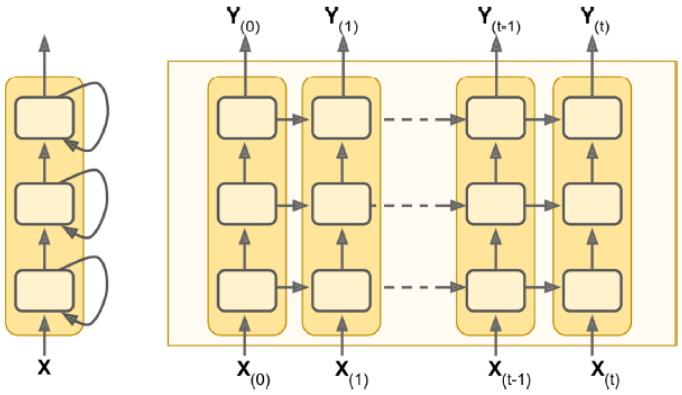


Figure 10.7: A *Deep RNN*

What changes in the notation is that all the showed equations for feedforward and backward propagation have another index indicating the layer. Clearly the matrices **W** are different according to the layer of the deep RNN.

10.4 Language Modeling with RNN

Before entering in the discussion on *"How RNN can be used for NLP*[2] *"*. It is important to understand, in general what is the idea behind having a **model for the language**. The **language modeling** can be seen in a general framework where it is required to *predict the probability of a certain sequence of words*. For example, between these two sentences:

The apple and pair salad
The apple and pear salad

what is the **most probable**? Clearly the second! The task is formulated in general as **predicting the next word given a set of previous words**. In formula:

$$P(w_1, w_2, ..., w_T) = P(w_1) \cdot P(w_2|w_1) \cdot \cdots \cdot P(w_T|w_1, ..., w_{T-1}) \tag{10.4}$$

So the probability of a certain sequence is given by the **conditional probabilities** of its own words. For such models the **training set** are *large corpora*, entire repository with books, paper and so on. While the **objective of the training** is *minimizing the prediction error on the next word*.
This is the right place to highlight that for each sequence a special token of <EOS> (End of sequence is used), in order to split a sequence from another.
In this context the **role of an RNN architecture** is to predict, step-by-step, the probability for the vocabulary words to be the next word, given a sequence of previous words. Now, there are two common situations:

1. The **Word Embeddings** are pretrained, and eventually only fine-tuned, on the current sequences; embedding like WORD2VEC can be used in this case;

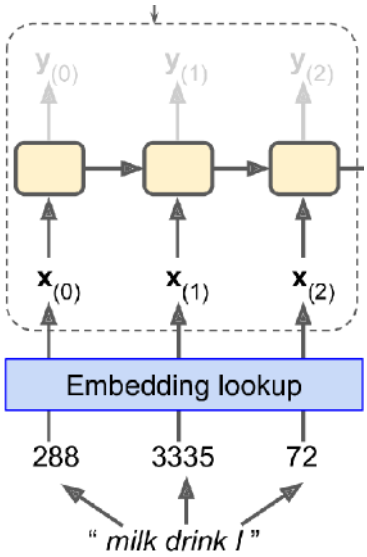
2. The **Word Embeddings** are part of the trainable parameters.

In the former case only the weights and biases are update by backward propagation, in the latter case, also the vocabulary embeddings are updated in order to obtain a general representation for the meaning of a certain word.

[2]Natural Language Processing (NLP tasks)

10.4.1 Training an RNN language model

The sentences of the training set are passed through the network, at each step an output probability is computed, using a softmax on the output vector $\hat{\mathbf{y}}_{(t)}$. The process is going on until the token <EOS> is reached. Such a situation is showed in the figure below:



Note that the hidden state are updated using the word embeddings obtained by passing the words one-hot encodings through a lookup module. After this forward propagation step a *cross-entropy loss* is computed comparing the generated and real next word of the sentences, then weights are updated. More specifically the required steps are:

❶ Update the hidden state $\mathbf{h}_{(t)}$ using $\mathbf{h}_{(t-1)}$ and $\mathbf{e}_{(t)}$ (embedding);

❷ An output $\hat{\mathbf{y}}_{(t)}$ is computed using the hidden state at time t linearly combined and passed through an activation function, this is mapped into a probability vector $\mathbf{p}_{(t+1)}$ using the softmax;

❸ The loss is computed for each time step using the real word $w_{(t+1)}$ and the computed probability distribution using the cross-entropy[3]:

$$\text{Loss}_t = -\log(\mathbf{p}_{t+1}[w_{t+1}]) \qquad (10.5)$$

the final loss is obtained summing up these contributions for all time steps.

❸ BPTT is performed in order to update the weight matrices $\mathbf{W}_h$ and $\mathbf{W}_y$ and biases vector $\mathbf{b}_h$ and $\mathbf{b}_y$.

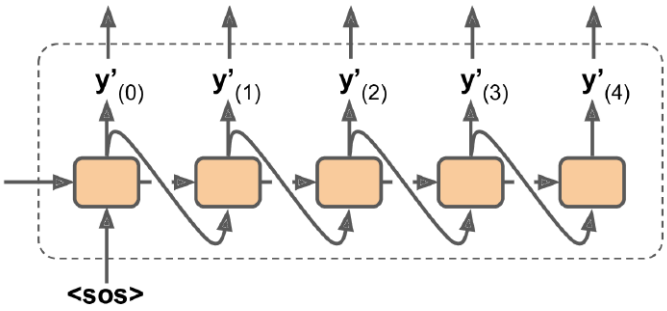
10.4.2 Use case: Sentence generation

After the training of such a model, how can we use it? **Sentence generation** is one of the tasks for which such a model can be used. The detailed procedure is:

1. We **initialize the state** $\mathbf{h}_0$ which may be zero or a learned value, while as first input of the sentence we pass the embedding of a <SOS>[4] token;
2. In order to **obtain the first word**, the hidden state corresponding to $\mathbf{h}_0$ is computed, and then the softmax-probability on the logit outputs.
3. Now, we have to **select a word**, this can be done using several strategies: (i) We take the argmax, (ii) we sample the probability distribution. (There is another more complicated approach (*Beam Search*) which is not treated here).
4. The selected w_1 is used as an input (we mean $\mathbf{E}[w_1]$ where E is the **embedding matrix**) for the next step.
5. Such steps are repeated till the token <EOS> is not predicted, this token (the same holds for <SOS> (or <BOS>)) in included in the embedding matrix.

The following figure depicts effectively the process we have just explained:

[3]What is the sense behind the notation $\mathbf{p}_{t+1}[w_{t+1}]$? $\mathbf{p}_{t+1}$ is the probability distribution for the next word, while w_{t+1} is the true next word. If such an index is very low, this result in a bad prediction for the network,



10.5 Issues with RNN training

The main problem in the training of basic RNN architectures is the **short-term memory**: due to the way the information are passing through the network, *some information is lost at each time step*, after a while the RNN hidden state has actually no trace of the first input. Image you are reading a sentence, and at the end you have not understood due to the fact you forgot the first part. In order to tackle this problem different types of **cells** have been introduced. Such cells have some **long-term memory** that over the time have made unused the basic cell.

In the figure a basic cell is showed. This is nothing but the graphical representation of the formulas (10.1)-(10.2).

In the article Pascanu, *"On the difficulty of training recurrent neural networks"*, 2013, [16] such issues are better explained.

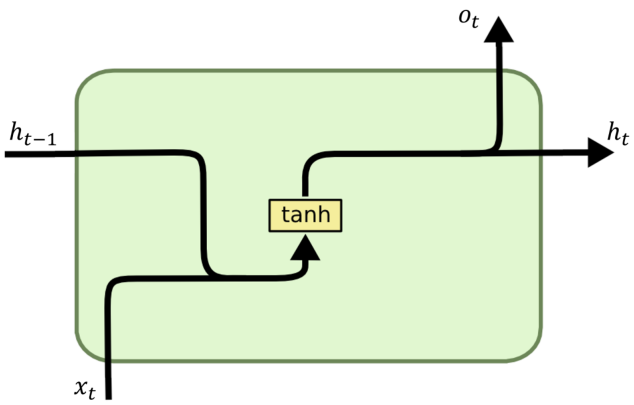


Figure 10.8: A basic RNN cell

10.6 Long-Short Term memories (LSTM)

The *Long-Short Term memories (LSTM)* ([14]) memory cell was introduced in 1997 by Hochreiter. If you consider an LSTM cell as a black-box, it can be used in practice as a basic cell, with the only difference that it performs much better. Another difference is that its state is split into **two vectors**: $\mathbf{h}_{(t)}$ that is the *short-term state* and $\mathbf{c}_{(t)}$ (c $\rightarrow$ cell) which is the *long-term state*. Now it is interesting to understand, **What is inside the box?** The key idea behind this novel type of cell is that the network can learn what to store in the long-term state, what to forget and what to read from it in order to give the short-term state. The input $\mathbf{x}_{(t)}$ and the hidden state $\mathbf{h}_{(t)}$ are fed into four different *fully connected layers*. They have different purposes:

❶ The main layer is the one which do have as output $\mathbf{g}_(t)$. This is nothing but the role charachterizing a basic cell, here the difference is that only most important part are retained in the long-term state.

❷ The other three layers have the role of **gate controllers**, since the have *logistic activation* (see the figure below). Their outputs are convolved into element-wise multiplication

taking $-log(\mathbf{p}_{t+1}[w_{t+1}])$ increase the loss contribution of a number which is bigger when the probability is low.

[4] *Start of Sequence* or *Start of Sentence*

operator such that if the output is 0s they close the gate, if they are 1s the gate are opened. More specifically:

- The **forget gate** (controlled by $\mathbf{f}_{(t)}$) controls *which part of the long-term state should be deleted*;
- The **input gate** (controlled by $\mathbf{i}_{(t)}$) controls which part of $\mathbf{g}_{(t)}$ should be added to the long-term state;
- the **output gate** (controlled by $\mathbf{o}_{(t)}$) controls which part of the long-state should be read and output in the term $\mathbf{h}_{(t)}$. In this case the output $\hat{\mathbf{y}}_{(t)}$ and $\mathbf{h}_{(t)}$ are coincident.

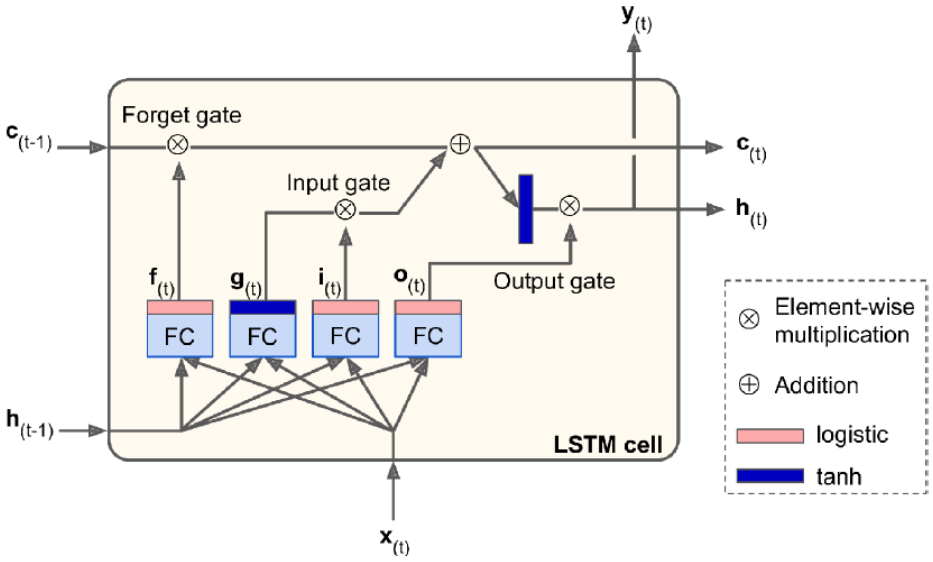


Figure 10.9: LSTM cell

An LSTM is able to recognize an important input (input gate) and store it (in the long-term state), until is needed (role of forget gate), and extract it whenever it is needed (role of the output gate). The equations (for a single instance) describing the LSTM cell are reported here:

$$\mathbf{i}_{(t)} = \sigma\left(\mathbf{W}_{xi}^T\mathbf{x}_{(t)} + \mathbf{W}_{hi}^T\mathbf{h}_{(t-1)} + \mathbf{b}_i\right) \quad \textbf{(input gate} \text{ controller)} \tag{10.6}$$

$$\mathbf{f}_{(t)} = \sigma\left(\mathbf{W}_{xf}^T\mathbf{x}_{(t)} + \mathbf{W}_{hf}^T\mathbf{h}_{(t-1)} + \mathbf{b}_f\right) \quad \textbf{(forget gate} \text{ controller)} \tag{10.7}$$

$$\mathbf{o}_{(t)} = \sigma\left(\mathbf{W}_{xo}^T\mathbf{x}_{(t)} + \mathbf{W}_{ho}^T\mathbf{h}_{(t-1)} + \mathbf{b}_o\right) \quad \textbf{(output gate} \text{ controller)} \tag{10.8}$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^T\mathbf{x}_{(t)} + \mathbf{W}_{hg}^T\mathbf{h}_{(t-1)} + \mathbf{b}_g\right) \quad \text{(basic cell output)} \tag{10.9}$$

$$\mathbf{c}_{(t)} = \mathbf{f}_{(t)} \otimes \mathbf{c}_{(t-1)} + \mathbf{i}_{(t)} \otimes \mathbf{g}_{(t)} \quad \text{(long-term state)} \tag{10.10}$$

$$\mathbf{h}_{(t)} = \mathbf{y}_{(t)} = \mathbf{o}_{(t)} \otimes \tanh(\mathbf{c}_{(t)}) \quad \text{(short-term state and output)} \tag{10.11}$$

10.7 Gated Recurrent Unit (GRU)

The *Gated Recurrent Unit (GRU)* cell is a **simplified version** of the LSTM cell. Different studies has demonstrated that the performances are the same, this explains their growing popularity. Such a type of cell has been introduced in 2014 in the article *"Learning phrase representations using RNN encoder-decoder for statistical machine translation"*, that also introduced the encoder-decoder network we have introduced so far. In the following we are going to mention the main simplifications:

1. Long-term and Short-term states are merged into a single hidden state denoted with $\mathbf{h}_{(t)}$;

2. There is a single **gate controller** called $\mathbf{z}_{(t)}$ for the *forget* and *input* gates. If $\mathbf{z}_{(t)}$ outputs a 1, the forget gate is open, and the input gate is closed (1-1=0) and viceversa.

3. There is **no output gate**, the full state vector is output at every step. However there is a new controller gate $\mathbf{r}_{(t)}$ that decides which part of the state must be shown into the main layer $\mathbf{g}_{(t)}$.

Here the equations are the following:

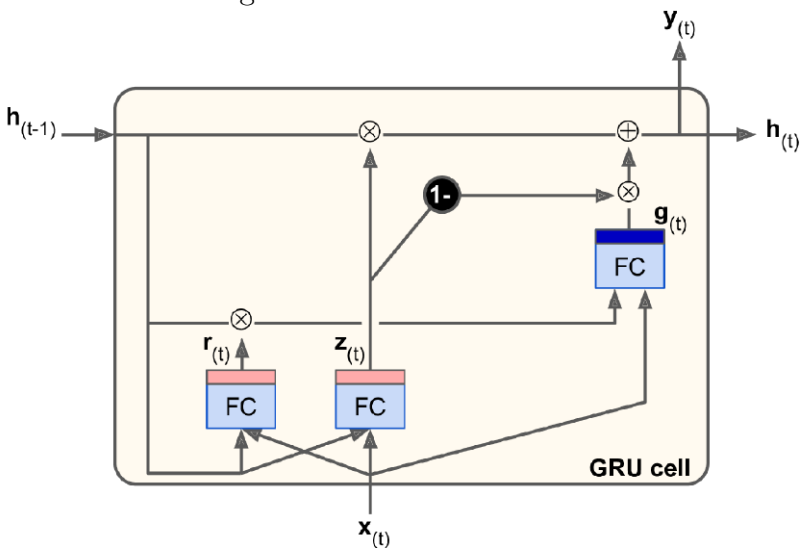


Figure 10.10: A *GRU cell*

$$\mathbf{z}_{(t)} = \sigma\left(\mathbf{W}_{xz}^T \mathbf{x}_{(t)} + \mathbf{W}_{hz}^T \mathbf{h}_{(t-1)} + \mathbf{b}_z\right) \tag{10.12}$$

$$\mathbf{r}_{(t)} = \sigma\left(\mathbf{W}_{xr}^T \mathbf{x}_{(t)} + \mathbf{W}_{hr}^T \mathbf{h}_{(t-1)} + \mathbf{b}_r\right) \tag{10.13}$$

$$\mathbf{g}_{(t)} = \tanh\left(\mathbf{W}_{xg}^T \mathbf{x}_{(t)} + \mathbf{W}_{hg}^T \left(\mathbf{r}_{(t)} \otimes \mathbf{h}_{(t-1)}\right) + \mathbf{b}_g\right) \tag{10.14}$$

$$\mathbf{h}_{(t)} = \mathbf{z}_{(t)} \otimes \mathbf{h}_{(t-1)} + \left(1 - \mathbf{z}_{(t)}\right) \otimes \mathbf{g}_{(t)} \tag{10.15}$$

LSTM and GRU are the main reason behind the success of RNNs, however still there are problem with the short-term memory, moreover there is an hard time learningfor long patterns longer than 100 steps.

10.8 Image captioning with RNN

The **Image captioning** is a task that deals with **extracting a brief description (caption) given an input image**. This is a not so simple task which sees the colaboration between two different deep learning models: a convolutional network (ConvNet) for image classification (Inception, ResNet, VGG16...) with an RNN that is employed for generating the textual description. The general steps are:

1. Image Preprocessing The image is resized to a fixed dimension and normalized, a CNN is used to extract *visual features* from the image;
2. Encoding the image The visual features are encoded into a compact representation. This often involves using the *last layer output as a feature vector*, which mainly captures the high level information about the image;

Figure 10.11: Example of image captioning

3. Caption generation Using the encoded image features as input, a sequence generation model (RNN, LSTM, GRU) is used to *generate the caption*. The model generates the caption word by word, starting from a special <SOS> token and ending with another special token <EOS>. The model is **trained** on large datasets with corresponding image-caption pairs, so it learns to predict the most likely words or phrases based on the encoded features vector.

Let v be the feature vector extracted from the CNN output, the hidden state equation is modified as follows in order to take into account the image features in the caption generation:

$$\mathbf{h}_{(t)} = g\big(\mathbf{W}_{hh}\mathbf{h}_{(t-1)} + \mathbf{W}_{hx}\mathbf{x}_{(t)} + \mathbf{W}_{hi}\mathbf{v} + \mathbf{b}_h\big)$$

where $\mathbf{W}_{hi}$ is a new weight matrix accounting for the relationship between the hidden state and image features vector.

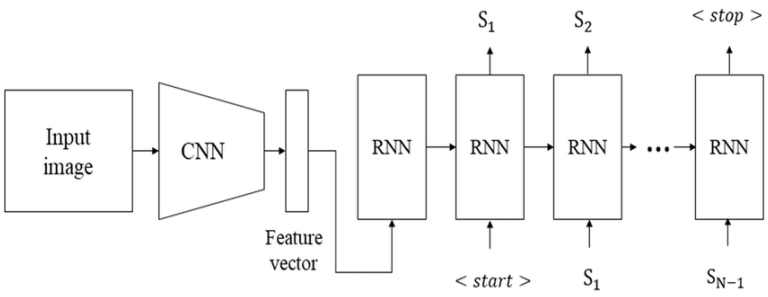


Figure 10.12: Architecture for image captioning

10.9 Attention mechanisms

Let us focus for a while on the neural machine translation task. We have seen that an encoder-decoder mechanism is used, moreover we have seen that using a bidirectional RNN is better for such a task. If we better analyze this process, we will understand that the path between the word to translate and the translated one is quite long.
Bahdanau in the paper [8], introduced a technique by which allows the decoder to focus on the

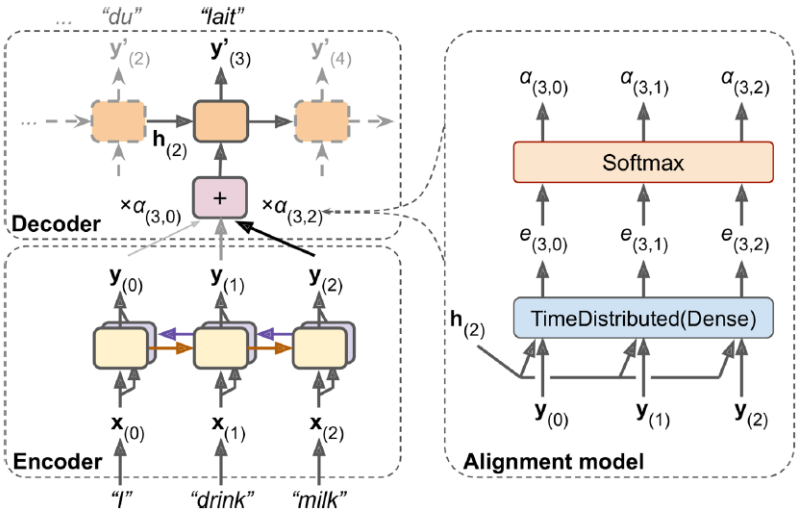


Figure 10.13: Encoder-Decoder with attention

appropriate word to be translated. By using such a modified architecture, the encoder instead of sending only the final hidden state, it sends **all of its outputs to the decoder**. The decoder computes a weighted sum of such outputs by using some weights $\alpha_{(t,i)}$ for the t-th time step and for the i-th encoder output. For example if

$$\alpha_{(3,2)} > \alpha_{(3,1)} > \alpha_{(3,0)}$$

this means that at the third time step the decode will pay much more **attention** on the term 2 which in this case is *milk*. The weights $\alpha_{(t,i)}$ are produced by a small neural network which is called the **alignment model**[5] (or **attention layer**). How it can be seen this is made up of a **Time Distributed dense layer**[6] whose inputs are all the outputs from the decoder and the hidden state from the previous step-time. The `Dense Layer` outputs an *energy score* for each encoder output which measures how *well aligned* is that output with respect to the previous hidden state. The α-weights are obtained using a softmax layer which is not time distributed. This attention mechanism is called *Bahdanau attention* or *concatenative attention*.

Another mechanism which is called *Luong attention* or *multiplicative attention* is based on the **dot-product** between the the encoder's output and the **decoder previous hidden state**, this is a quite fair *similarity measure*, since the dot product is related to a $\cos\theta$. The result is passed through a softmax which will compute the attention weigths.

10.9.1 Image captioning with attention

This is in general the same task we have seen before with the only difference that the attention mechanism enhances the performances of the model.
Here some **feature vectors** are generated corresponding to different region of the image. These serve as the **values** and **keys** in the attention mechanism.
The *Caption generation* also here occurs one word at a time, and according to an RNN with attention or a transformer. Here the hidden state plays the role of **query** in the attention mechanism. At each time step **attention weights** are computed for each image region highlighting the regions most relevant to the current word.

[5] **Alignment model** comes from the fact that we are seeking a form of coherence (alignment) between the encoder's outputs and the decoder hidden state. The closer the encoding of the hidden state to a certain output, the higher the associated α-weight.

[6] The layer is applied independently at each time step

Figure 10.14: Examples of image captioning with attention

Attention weight calculation

The model calculates attention scores using dot product between the query (current RNN hidden state) and the keys (image features). The scores are passed through a softmax in order to obtain the attention **attention weights**.

Context vector

The weighted sum of image features, using attention weights, creates a **context vector**, this combined with the hidden state of the RNN, is used to predict the next word.

10.9.2 Visual question answering

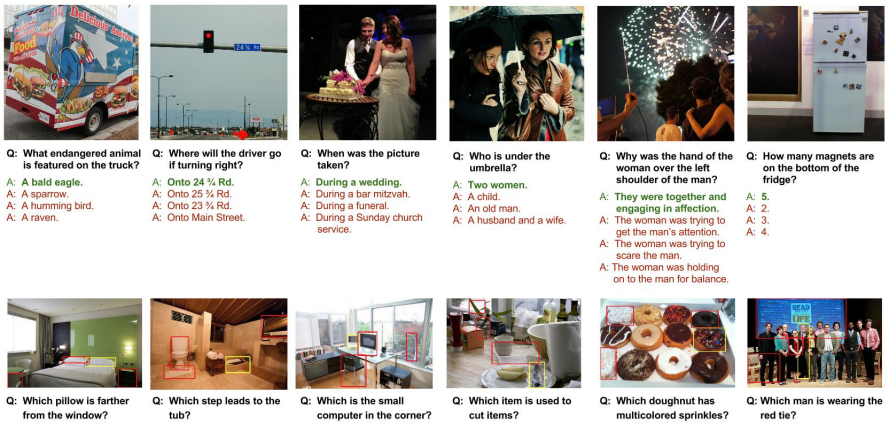


Figure 10.15: *Visual question answering example*

The task of **Visual Question Answering (VQA)** is a complex deep-learning task, that sees the fusion of visual task with textual tasks. The input of the model is an image with a related question. For example an image of a car, and a question *what color is the car?* The output is in textual form (for example *red*). There are three main phases:

1. Feature Extraction the visual features are extracted using a ConvNet, the textual ones using a model like RNN with attention or a transformer.

2. Multimodal fusion The features are combined by using some form of element-wise multiplication, dot product...
3. Answer generation Here there is a classifier that predicts the answer by using a predefined set of answers for VQA.

Attention in VQA

Attention mechanisms are crucial in VQA to focus on relevant parts of the image or question:

- **Visual Attention**: identifies specific regions in the image relevant to the question. For example, when asked, "What is the person holding?", the model attends to the hands and objects in the image;
- **Question Attention**: highlights important words or phrases in the question to guide the visual attention.
- **Multimodal Attention**: dynamically attends to both the image and the question simultaneously, ensuring the model aligns the two modalities effectively.

Famous papers on VQA are:

- Antol et al. "Vqa: Visual question answering", **vqa**, [6]
- Zhu et al. "Visual7w: Grounded question answering in images", 2016, [24]

10.10 Attention is all you need: *Transformer* architecture

In a 2017 paper ([23], "Attention is all you need"), some Google researchers proposed a mechanism which significantly improved the NMT field in which no recurrent or convolutional layers was used, only **attention mechanism** together with fully connected, embedding, normalization layers and few other pieces of data. The transformer architecture is presented in the Figure 10.16.
The left part of the figure is the **encoder** which receives the input as *word IDs*, these are passed through an embedding lookup, the top part of the encoder is stacked N times.

The right part of the figure represents the **decoder** which during the training is fed with the target sequence and with the output from the encoder, similarly than the other encder-decoder models, the output is a probability distribution over all of the words of a given vocabulary.

Looking more closely there are 2 embedding layers, $5 \times N$ skip connections, $2 \times N$ feed-forward layers composed of two dense layers which are not-recurrent and so **time-distributed** so that each word is treated independently from the others.
Now, how can be avoided the presence of recurrent layers? That is how can a word be processed indepently from the others? At this point two novel elements comes into play:

- Multi-Head attention module such a module encode in a compact way all the relationship between a word in the sentence and all the others. For example in the sentence *They welcomed the Queen of the United Kingdom.* The output of this layer for the word "Queen" will have higher weights for the words "United" and "Kingdom" than for other words. This mechanism is called **self-attention** (since the sentence is paying attention to itself); the Multi-head attention module is based on the **scale dot-product attention**.

- Positional embedding module The multi-head attention module does not keep track of the position of a certain word in a given sentence from the mini-batch, this is an important information for the Transformer to be processed. For this reason a *positional embedding vector* is added to the word embedding. Such added vector allows to the model to keep track about absolute/relative positions of the words within a sentence.

10.10.1 Scaled Dot-product attention

A fundamental brick for Transformer is the **scaled-dot product attention**, this allows us to focus on the most relevant part when meaking decision.

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{keys}}}\right)\mathbf{V} \tag{10.16}$$

where the components are:

- **Query (Q)** A vector for each word embedding that represents what we are looking for.
- **Key (K)** Is a reference point, these determine how relevant different parts are to the query, they represents all the "things" we can focus on
- **Value (V)** is the actual content that we want to extract after deciding where to focus (eg. the meaning of the most relevant word).

Q, K, V are obtained by applying three separate learned linear transformations to the input embeddings. In particular they are obtained through the multiplication by using learnable weight matrices W^Q, W^K, W^V. Then they are obtained as:

$$\mathbf{Q} = \mathbf{X}W^Q \quad \mathbf{K} = \mathbf{X}W^K \quad \mathbf{V} = \mathbf{X}W^V \tag{10.17}$$

10.10.2 Multi-head Attention layer

When we want to improve the attention mechanism in Transformers, **Multi-head attention** is used, here the model computes multiple attention in parallel. In particular **each head** learns to focus on different parts of the input. This leads to an enhancement in the general performances. Here there are the steps behind the multi-head attention mechanism:

1. **Splitting the input** Here the input embeddings X are transformed into Query, Key and Value matrices by projecting X on a certain subspace defined by the learnable matrices W^Q, W^K, W^V.

2. **Parallel Attention heads** In this context different weight matrices are used for each head. Moreover if we have h heads and the dimension of the embeddings is d, the i-th head works on vectors of dimension d/h, that is:

$$\text{head}_i = \text{Attention}(\mathbf{Q}W_i^Q, \mathbf{K}W_i^K, \mathbf{V}W_i^V) \tag{10.18}$$

3. **Concatenation of heads** the output of the h heads is concatenated as follows:

$$\text{Concat} = [\text{head}_1, \text{head}_2, ..., \text{head}_h] \tag{10.19}$$

   so that the resulting matrix has dimensione $N \times d$ where N is the batch size.

4. **Final linear transformation** The final attention result is obtained as:

$$\text{Multihead}(Q, K, V) = \text{Concat} = [\text{head}_1, \text{head}_2, ..., \text{head}_h]W^O \tag{10.20}$$

   with W^O learnable matrix.

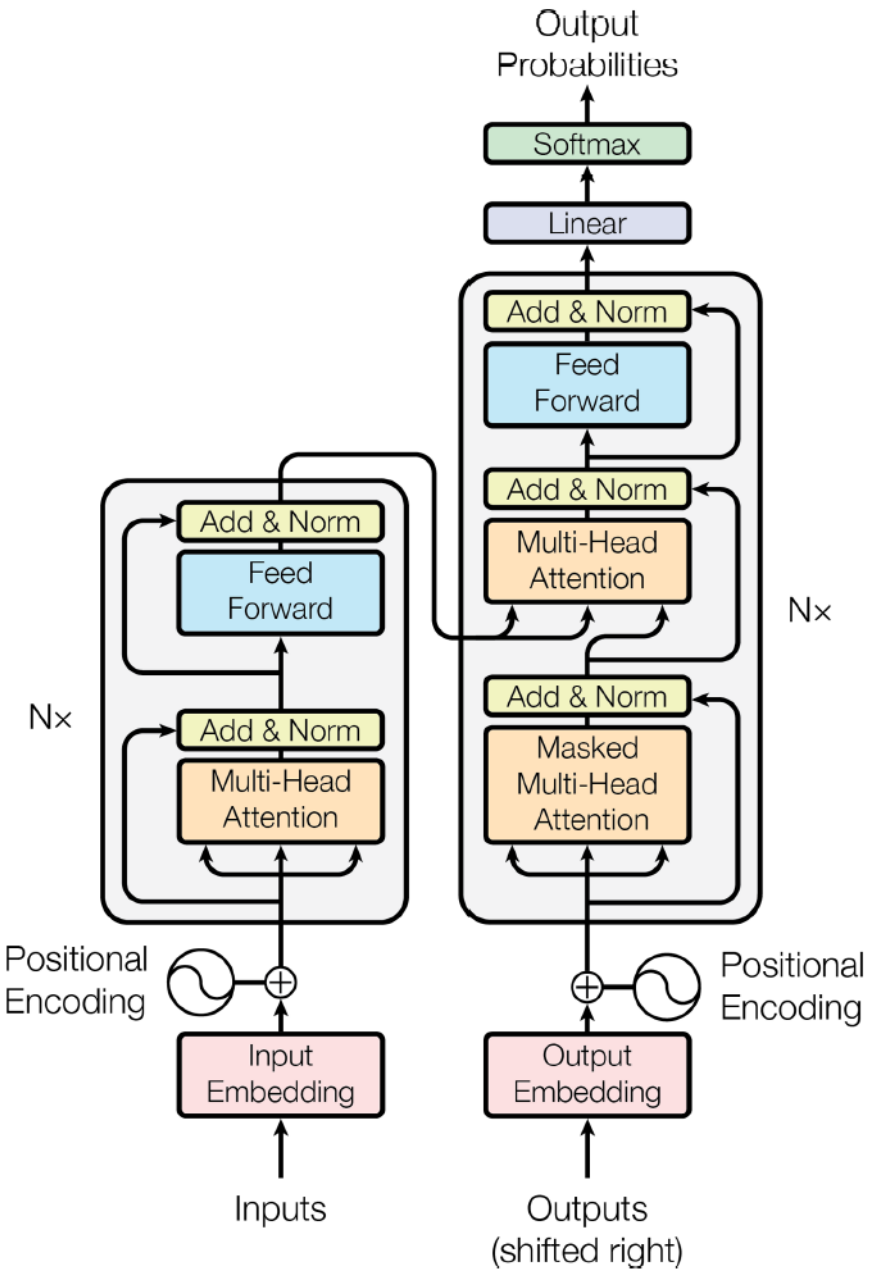


Figure 10.16: The **Transformer** architecture

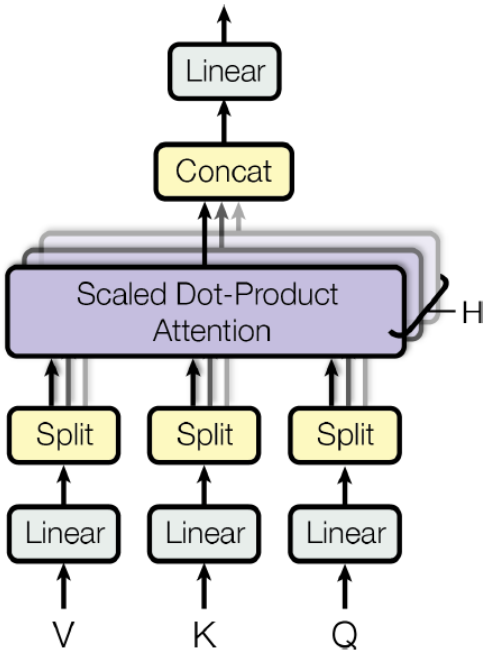


Figure 10.17: Architecture of the *Multi-Head layer*

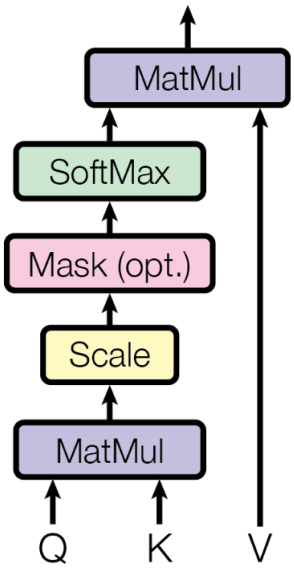


Figure 10.18: Scaled-dot product attention

10.11 Transformers vs RNN

In order to conclude this chapter we provide a comparison between Transformers and RNN in term of mechanisms, performance and so on.

Aspect	Transformers	RNNs
Architecture	Uses self-attention to process input sequences in parallel.	Processes sequences step-by-step using recurrence, one token at a time.
Sequence Processing	Processes the entire sequence simultaneously (parallel processing).	Sequentially processes tokens, one after another.
Memory Handling	Handles long-range dependencies effectively using self-attention.	Struggles with long-term dependencies due to vanishing/exploding gradients.
Parallelism	Fully parallelizable; faster training and inference.	Sequential nature prevents parallelism; slower training and inference.
Positional Information	Requires explicit positional encoding (e.g., sinusoidal or learned embeddings).	Naturally handles positional information through sequence order.

Aspect	Transformers	RNNs
Variable-Length Sequences	Easily handles variable-length sequences with masking.	Handles variable-length sequences natively but requires padding for training.
Long-Term Dependencies	Excellent; attends to all tokens regardless of their distance.	Poor; performance degrades with longer dependencies (though improved with LSTMs/GRUs).
Expressiveness	Highly expressive due to self-attention and multiple heads.	Limited by step-by-step processing and simpler architectures.
Convergence	Faster convergence due to parallelism and better gradient flow.	Slower convergence due to sequential processing and gradient challenges.
Scalability	Scales well to large datasets and models (e.g., GPT, BERT).	Struggles to scale efficiently to large datasets.
Sequence Length	Handles long sequences effectively with global attention.	Performance degrades significantly for long sequences.
Applications	State-of-the-art in NLP tasks (e.g., translation, summarization). Emerging use in vision (Vision Transformers).	Previously dominant in NLP. Common in time-series tasks but rare in vision applications.
Strengths	• Parallel processing speeds up training and inference. • Captures long-term dependencies well. • Scales effectively to large datasets. • Adapts well to different modalities (text, images, audio).	• Naturally processes sequences step-by-step. • Compact models are resource-efficient. • Implicitly handles positional information.

Aspect	Transformers	RNNs
Weaknesses	• Quadratic complexity ($O(N^2)$) in self-attention can be costly for long sequences. • Requires explicit positional encoding. • High memory and computational requirements.	• Slow training due to sequential bottlenecks. • Gradient issues (vanishing/exploding) hinder long-term dependency modeling. • Limited expressiveness compared to Transformers.
Best Use Cases	• Long sequences and large datasets. • NLP tasks (e.g., machine translation, summarization). • Tasks requiring long-term dependency modeling.	• Small-scale, sequential tasks (e.g., time-series forecasting). • Applications where resources are limited.

Chapter 11

Machine and Deep Learning for Audio

Chapter 12

Generative Adversarial Networks (GAN)

12.1 Introduction

Till now we have focused our attention on *discriminative models* that roughly speaking, given some data in a given space x, gives by a certain *probability distribution* the information on which class y they are part. There are plenty of machine learning techniques that solve effectively this type of task. These models are seeking in existing data some interesting patterns and use them in order to predict a class (classification) or a continuous number (regression).
At the opposite **generative models** given a noise input z and a certain class y, generate a new, never seen sample x for that class. In this chapter, in particular we are going to talk about *GANs (Generative Adversarial Networks)* which were invented by a PhD student (Ian Goodfellow), this enabled computers in the generation of new data using not one, but **two different neural networks**.

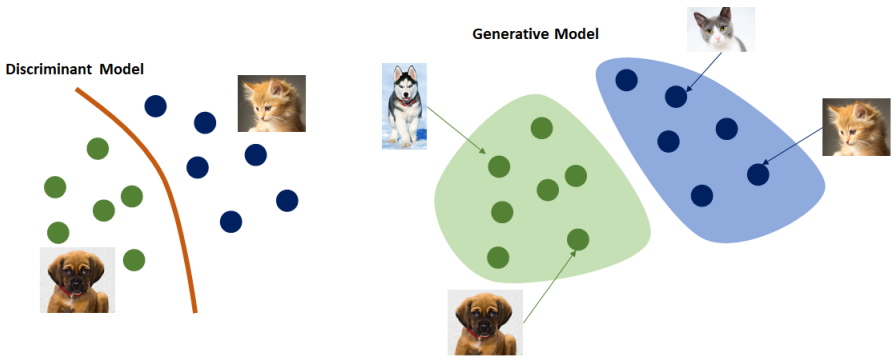


Figure 12.1: Discriminative vs Generative Models

12.2 Variational Auto-Encoders (VAE)

We consider here as a first approach to generative models the **autoencoders** and **variational autoencoders**, for several reasons: (i) it is an easier setting for generative AI; (ii) since generative models are challenging to be understood, autoencoders are closer to the models we have already seen, (iii) the autoencoders are directly or implicitly used in some variants of GAN architectures.

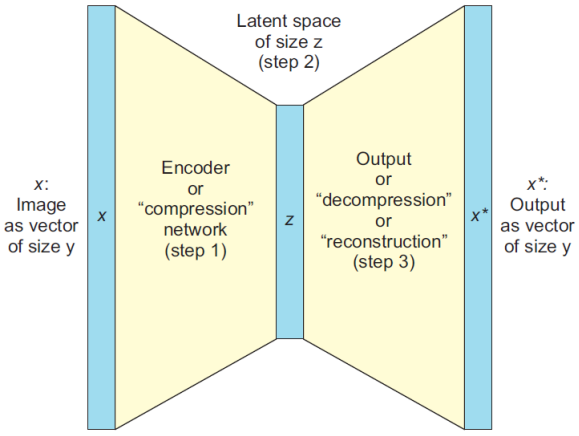


Figure 12.2: Autoencoder architecture

12.2.1 Autoencoders

The structure of an autoencoder is quite intuitive and follows these few steps:

1. ENCODER NETWORK this is the stage where we take a (full) representation x (of an image for example) and reduce its dimension into a space z by using a learned encoder (a classical ConvNet for example);
2. LATENT SPACE this is an intermediate stage in which the autoencoder architecture tidies its *thoughts*;
3. DECODER NETWORK we reconstruct the original dimension of the input x, starting from the latent space into a new generated image we call x^*.

The *training of an autoencoder* occurs as follows:

1. We take the images x through the autoencoder;
2. We collect the generated images x^* as reconstruction of the given images;
3. We measure a form of *reconstruction loss* by mean (for example) of a mean squate error between the pixels of x and x^*
4. We obtain an explicit objective function to be minimized by mean of a gradient descent approach:

$$\mathcal{L} = \|x - x^*\|_2^2 \tag{12.1}$$

The autoencoders can work by mean of an unsupervised machine learning model where we learn only from the training data without the labels. Note that we have a single loss function to be optimized with the common goal of *minimizing the differences between the input and output images*. We can use an autoencoder for different purposes, for example: image denoising, image colorization...

12.2.2 Variational autoencoders

The traditional autoencoder maps the features of the input space into a latent space where the representation z is nothing but a set of numbers. The main difference between autoencoders and Variational autoencoders (VAE) is just on the "magic" latent space. In fact here the choice is to represent the latent space as a *probability distribution* with a certain mean (μ) and a standard deviation (σ). Note that you have to learn such a distribution! Once you

have the distribution, you sample some numbers from it, add some noise and feed them to the decoder that will generate something that looks like the images of the training set, with the only difference they are **newly generated**.

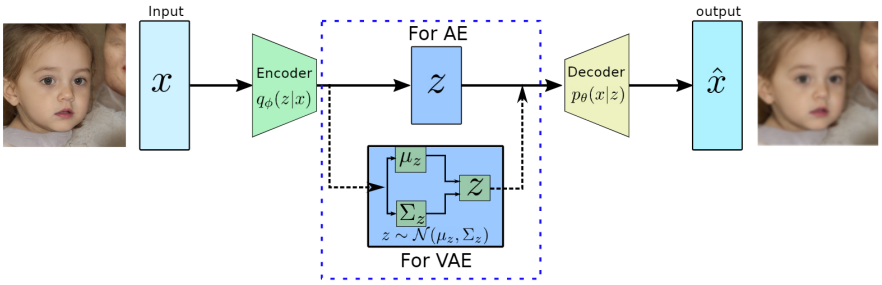


Figure 12.3: (Variational) Auto-Encoder

12.3 Generative Adversarial Networks

Generative Adversarial Networks **(GANs)** constitute a class of machine learning techniques which consist of two jointly trained models: the first (the *Generator*) trained to create (generate) fake data, and the other (the *Discriminator*) trained to binary distinguish fake data from real data. Let us go more deeply in the description of the GAN term:

GENERATIVE is the overall purpose of the machine learning model: generate new data. Clearly the generated data (images for example) depends on the features of the training set. If we want to generated new pictures of Leonardo Da Vinci, we must have a training set with Leonardo Da Vinci's portrait.

ADVERSARIAL term refers to the game-like, competitive dynamic between the two models that constitute such a framework: the Generator and the Discriminator. The form is like an *art forgery* while the discriminator is like an *art expert* whose role is to say wheter a painting is fake or real;

NETWORK the models used for representing the two counterpart of the architecturesare neural networks. According to the complexity of the model, these can be simply Fully Connected Network, or ConvNet or in even more complex cases architectures like U-Net.

This vanilla architecture is not suitable when:

1. We have large variations and different classes, while it performs well with small variations and small datasets;

2. In such a type of architecture there is no way to control what the network have to generate.

Both these issues are addressed introducing some novel aspects in the basic architecture, how we will see in the next sections.
Nowadays, GANs are used in order to synthetize artificially some images of some class or for other purposes like text to image or image to image translation. It is remarkable that for GAN there is not a latent space that will be decoded into a generated sample, due to the presence of a discriminator that allows the generator to be improved.

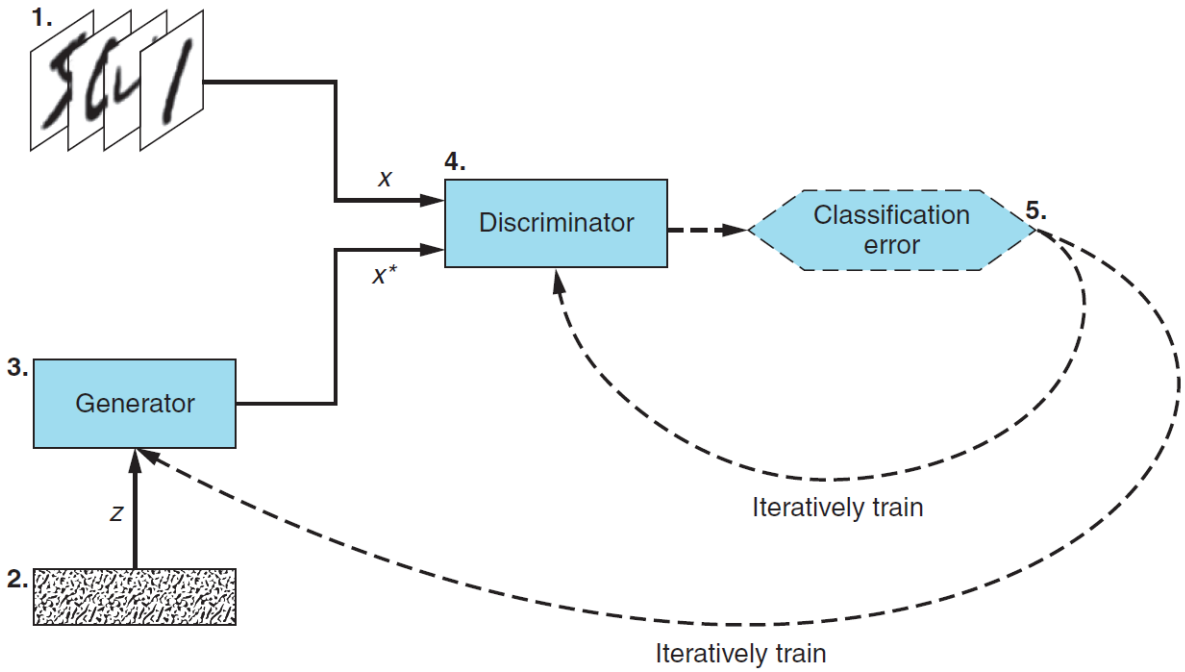


Figure 12.4: GAN architecture

12.3.1 GAN anatomy

The Figure 12.4 shows the architecture of a GAN in its original form. Let us analyze the details of such diagram:

1. *Training dataset* these are the real examples from which we want to learn the features, this is the input x to the discriminator network
2. *Random noise vector* Is the input z of the Generator network, is nothing but a vector of random number that the generating network uses as a starting point;
3. *Generator network* Takes as input the noise vector z and outputs fake samples x^*, its goal is to produce samples that are as close as possible to the real data in order to fool the Discriminator;
4. *Discriminator network* takes as input either a real example x or a fake one x^*, for each generated example the Discriminator determines and outputs the probability of wheter an example is real/fake.
5. *Iterative training tuning* where the Discriminator's weights and biases are update in order to maximize the classification accuracy (maximizing the probability that $x \rightarrow$real and $x^* \rightarrow$ fake), while the Generator's weights and biases are updated in order to maximize the probability that the discriminator classify x^* is real. This is the reason why the two networks are as in a competition.

Now we can say that: (i) at the end of the training the discriminator models $p(y|x)$ where y can be real or fake; (ii) the generator learns a certain probability $p(x|y)$ that depends on the characteristics of the training set, that is the most common examples in the training set will be the most likely to be generated.

Note that the discriminator is used only in training phase: once you have obtained the learned generator, it is sufficient to provide it some noise vectors that will be mapped into generated images.

12.3.2 Training GAN

Till now we have done a snapshot of the engine analyzing the main features and the components it is made up of. An explanation of the training process is given here in order to better understand the mechanisms under the hood. The training of a GAN sees the alternate tuning of the Discriminator and Generator. In particular:

1. Train the Discriminator
    a Take a random real example x from the training dataset;
    b Get a random noise vector in order to generate a fake example x^*;
    c Use the Discriminator to classify x and x^*
    d Compute the classification error (loss) and backpropagate the total error in order to update the Discriminator trainable parameters, seeking to minimize the classification errors;
2. Train the Generator
    a Get a random vector z and using the Generator Network we synthetize a new fake example x^*;
    b Use the discriminator to clasify x^*
    c After having computed the classification (loss function), backpropagate the error in order to update the Generator's learnable parameters in order to maximize the classification error.

These two steps are repeated for each iteration. The alternate training of both models, these are supposed to improve together! Indeed, a perfect discriminator will not permit the generator to improve, a perfect generator model will always fool the discriminator without allowing it to improve. In the following we will indicate the elaboration of an image through the discriminator as $D(x)$ or $D(G(z))$ (in the first case it takes a real example in the second case a generated one), the elaboration carried out by the generator are indicated with $G(z)$ (where z is a noise vector). Next, we are going to enter moore deeply into the GAN formulation including the cost function and possible stopping criteria to learning.

12.3.3 GAN Formulation

A Generative Adversarial Networks model can be formulated as a **min-max game** where the discriminator is trying to maximize its reward, while the generator is trying to minimize such reward. That is

$$\min_G \max_D V(D, G) \tag{12.2}$$

where

$$V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log(D(x))] + \mathbb{E}_{z \sim p_g(z)}[\log(1 - D(G(z)))] \tag{12.3}$$

we take the *expected values* since both x and z are random vectors. From such a $V(D, G)$ we are able to extract the loss functions J^D and J^G for the discriminator and generator according to the objective we have stated about them.

Discriminator loss function J^D and gradient ascent

It is required that the Discriminator predicts 1 for real image, 0 for fake ones, it is sufficient to impose

$$J^D = V(G, D) \tag{12.4}$$

since the first term is maximum when $D(x) = 1$ (D applied to the real example x gives 1 (real)) and the second term is maximum when $D(G(z)) = 0$ (that is D applied to the generated example $G(z)$ gives $0 \rightarrow$ fake). The **gradient ascent** algorithm can be used so that the parameters θ_d are updated as:

$$\theta_d \leftarrow \theta_d + \mu \nabla J^D(\theta_d) \tag{12.5}$$

with μ being the learning rate.

Generator loss function J^G and gradient descent

Since the generator needs to fool the discriminator it must update its own parameters so that the discriminator could predict 1 when $G(z)$ is provided. The second term of $V(G, D)$ is taken as J^G since it is the one in which the $G(z)$ appears. Then:

$$\mathbb{E}_{z \sim p_g(z)}[\log(1 - D(G(z)))] \tag{12.6}$$

such a functional is minimum when $D(G(z)) = 1$. Since we want to minimize J^G, **gradient descent** must be applied:

$$\theta_g \leftarrow \theta_g - \mu \nabla J^G(\theta_g) \tag{12.7}$$

with μ being the learning rate.

An alternative formulation, called the Non-Saturating GAN, sees the generator to maximize the log-probability that the discriminator could fail.

12.3.4 When to stop training GAN?

Those familiar with *game theory* will recognize that the GANs can be seen as a *zero-sum game*, where there is at a certain point a **Nash equilibrium point**. This can be reached either when the generator has the same distribution of data with respect to the discriminator, or when the discriminator always outputs a probability of $1/2$ for each given sample. In this situation neither player can improve its own position.

12.3.5 The challenges in Training GAN

We have understood that training a GAN is not so simple, since the two networks are in competition the one with another. Moreover the training is even more **challenging** due to two main reasons: (i) Non convergence; (ii) Mode collapse.

Non-convergence

The deep-learning model we have seen before introducing GANs involved a single player that has been trying to maximize its reward, we used Stochastic Gradient Descent that guaranteed convergence under certain conditions. In the case of GANs there is a player which is trying to minimize the reward of the other. There is no collaboration, moreover the gradient based method are not converging to the Nash Equilibrium.

Mode collapse

12.4 From GAN to DCGAN

Bibliography

[1] Szegedy et al. "Going deeper with convolutions". In: (2014).

[2] He et al. "Deep residual networks for image recognition". In: (2015).

[3] Krizhevsky et al. "ImageNet classification with deep convolutional neural networks". In: (2012).

[4] LeCun et al. "Gradient-based learning applied to document recognition". In: (1998).

[5] Lin et al. "Network in network". In: (2013).

[6] Stanislaw Antol et al. "Vqa: Visual question answering". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 2425–2433.

[7] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. "Segnet: A deep convolutional encoder-decoder architecture for image segmentation". In: *IEEE transactions on pattern analysis and machine intelligence* 39.12 (2017), pp. 2481–2495.

[8] Dzmitry Bahdanau. "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (2014).

[9] Kyunghyun Cho. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[10] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 2414–2423.

[11] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[12] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). arXiv: 1504.08083. URL: http://arxiv.org/abs/1504.08083.

[13] Kaiming He et al. "Mask R-CNN". In: *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.322. URL: https://doi.org/10.1109/ICCV.2017.322.

[14] S Hochreiter. "Long Short-term Memory". In: *Neural Computation MIT-Press* (1997).

[15] Hyeonwoo Noh, Seunghoon Hong, and Bohyung Han. "Learning deconvolution network for semantic segmentation". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1520–1528.

[16] R Pascanu. "On the difficulty of training recurrent neural networks". In: *arXiv preprint arXiv:1211.5063* (2013).

[17] Adam Paszke et al. "ENet: A Deep Neural Network Architecture for Real-Time Semantic Segmentation". In: *arXiv preprint arXiv:1606.02147* (2016). URL: https://arxiv.org/abs/1606.02147.

[18] J Redmon. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[19] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497. URL: http://arxiv.org/abs/1506.01497.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer. 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[21] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A Unified Embedding for Face Recognition and Clustering". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 815–823.

[22] Pierre Sermanet et al. *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*. arXiv:1312.6229. Feb. 2014. DOI: 10.48550/arXiv.1312.6229. URL: http://arxiv.org/abs/1312.6229 (visited on 11/27/2024).

[23] A Vaswani. "Attention is all you need". In: *Advances in Neural Information Processing Systems* (2017).

[24] Yuke Zhu et al. "Visual7w: Grounded question answering in images". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4995–5004.

[25] Simonyan & Zisserman. "Very deep convolutional networks for large-scale image recognition". In: (2015).