POLITECNICO
MILANO 1863

Artificial Neural Network and Deep Learning

Notes by: Alessio Bray

Accademic Year 2020/2021

Contents

1 Introduction to Deep Learning

1.1 What is Machine Learning?

Machine Learning is a **category of research and algorithms focused on finding patterns in data and using those patterns to make predictions**. Machine learning falls within the artificial intelligence (AI) field but it also intersect with several disciplines like neurocomputing, deep learning, the broader field of knowledge discovery and data mining, pattern recognition and statistics.
A machine learning algorithm is an algorithm that is **able to learn from data**. But what do we mean by learning? Tom M. Mitchell (1997) provides a succinct definition:

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

This definition has 3 elements that we should take into account:

- **Task, T**: it is the **problem we are trying to solve**. So the process of learning itself is not the task: learning is our means of attaining the ability to perform the task. For example, if we want a robot to be able to walk, then walking is the task. We could program the robot to learn to walk, or we could attempt to directly write a program that specifies how to walk manually, so as said the learning is only a way to to be able to perform a task.
  Machine learning tasks are usually described in terms of how the machine learning system should process an **example**. An example is a collection of **features** that have been quantitatively measured from some object or event that we want the machine learning system to process. We typically represent an example as a vector $x \in \mathbf{R}^n$ where each entry x_i of the vector is another feature. For example,the features of an image are usually the values of the pixels in the image.

  We should always know what we are trying to face since depending on it there are different models and tools (e.g. unsupervised learning and supervised learning have different tools and set of metrics to evaluate the result).

  Many kinds of tasks can be solved with machine learning.

- **Performance measure, P**: To **evaluate the abilities of a machine learning algorithm**, we must design a **quantitative measure of its performance**. **Usually** this performance measure **P is specific to the task T** being carried out by the system. For tasks such as classification, classification with missing inputs, and transcription, we often measure the accuracy of the model. Accuracy is just the proportion of examples for which the model produces the correct output. We can also obtain equivalent information by measuring the **error rate**, the proportion of examples for which the model produces an incorrect output. We often refer to the error rate as the expected 0-1 loss. The 0-1 loss on a particular example is 0 if it is correctly classified and 1 if

it is not. For tasks such as density estimation, it does not make sense to measure **accuracy**, error rate, or any other kind of 0-1 loss. Instead, we must use a different performance metric that gives the model a continuous-valued score for each example. The most common approach is to report the average log-probability the model assigns to some examples.

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when deployed in the real world. We therefore evaluate these performance measures using a test set of data that is separate from the data used for training the machine learning system.

The choice of performance measure may seem straightforward and objective, but it is often difficult to choose a performance measure that corresponds well to the desired behavior of the system. In some cases, this is because it is difficult to decide what should be measured. In other cases, we know what quantity we would ideally like to measure, but measuring it is impractical. In these cases, one must design an alternative criterion that still corresponds to the design objectives, or design a good approximation to the desired criterion.

In conclusion we could **use different loss function for different tasks to improve the performance of the computer program**.

- **Experience (i.e. data), E**: Machine learning algorithms can be broadly categorized as **unsupervised** or **supervised** by what kind of experience they are allowed to have during the learning process. The algorithms we usually see are allowed to experience an entire **dataset** D. A **dataset is a collection of many examples**, as defined before. Sometimes we call examples **data points** x_i.

$$D = x_1, x_2, \ldots, x_N$$

Many **kinds of tasks** can be solved with machine learning. Some of the **most common** machine learning tasks include the following:

- **Classification**: In this type of task, the computer program is **asked to specify which of *k* categories some input belongs to**. To solve this task, **the learning algorithm is usually asked to produce a function** $f : \mathbf{R}^n \rightarrow 1, ..., k$. When $y = f(x)$, **the model assigns an input described by vector** x **to a category identified by numeric code** y. There are other variants of the classification task, for example, where f **outputs a probability distribution over classes**. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image.

- **Regression**: In this type of task, the computer program is asked to **predict a numerical value given some input**. To solve this task, the learning algorithm is asked to **output a function** $f : \mathbf{R}^n \rightarrow \mathbf{R}$. This type of task is similar to classification, except that the format of output is different. An example of a regression task is the prediction of the expected claim amount

that an insured person will make (used to set insurance premiums), or the prediction of future prices of stocks. These kinds of predictions are also used for algorithmic trading.

There are **3 main machine learning paradigms** that uses the experience E (i.e. data $D = x_1, \dots x_N$):

- **Unsupervised learning algorithms: experience a dataset** containing many features, then **learn useful properties of the structure of this dataset** i.e. **exploit regularities in to build a representation to be used for reasoning or prediction**. In the **context of deep learning**, we usually want to **learn the entire probability distribution that generated a dataset**, whether explicitly, as in density estimation, or implicitly, for tasks like synthesis or denoising. Some other unsupervised learning algorithms perform other roles, like **clustering**, which consists of dividing the dataset into clusters of similar examples.

- **Supervised learning algorithms**: **experience a dataset** containing features, **but each example is also associated with a label or target**. For example, the Iris dataset is annotated with the species of each iris plant. A supervised learning algorithm can study the Iris dataset and learn to classify iris plants into three different species based on their measurements i.e **given the desired outputs** $t_1, t_2, ...t_n$ **learn to produce the correct output given a new set of input**.

- **Reinforcement Learning producing actions** $a_1, a_2, \dots, a_N$ **which affect the environment and receiving rewards** $r_1, r_2, \dots r_N$ **learn to act in order to maximize the reward in the long term**. It is characterized by the interaction of an agent and the environment. The agent does some action on the environment receiving an observation and a reward. The objective of the agent is to maximize the return (i.e. the long term reward). The reward somehow encodes the performance P of the definition.

Roughly speaking, unsupervised learning involves observing several examples of a random vector x and attempting to implicitly or explicitly learn the probability distribution $p(x)$, or some interesting properties of that distribution; while supervised learning involves observing several examples of a random vector x and an associated value or vector y, then learning to predict y from x, usually by estimating $p(y|x)$. The term **supervised** learning originates from the view of **the target** y **being provided by an instructor or teacher** who shows the machine learning system what to do. **In unsupervised learning, there is no instructor or teacher, and the algorithm must learn to make sense of the data without this guide**.
Though unsupervised learning and supervised learning are not completely formal or distinct concepts, they do help roughly categorize some of the things we do with machine learning algorithms. Traditionally, people refer to regression, classification and structured output problems as supervised learning. Density estimation in support of other tasks is usually considered unsupervised learning. **Other variants**

of the learning paradigm are possible. For example, in **semi-supervised learning**, **some examples include a supervision target but others do not**. Some machine learning algorithms do not just experience a fixed dataset. For example, **reinforcement learning algorithms** interact with an environment, so there is a feedback loop between the learning system and its experiences: producing actions $a_1, a_2, ..., a_n$ which affect the environment, and receiving rewards $r_1, r_2, ..., r_n$ learn to act in order to maximize rewards in the long term.

NOTE Just as there is no formal definition of supervised and unsupervised learning, there is no rigid taxonomy of datasets or experiences.

NOTE **Datasets** are a set of inputs, samples collected. they are **vectors** not values (e.g exam-i: name-i + grade-i)

Example Assume we are a head hunter that want to understand for which position you should apply: in this case the Experience E is all the applicant in the past and the position now $\implies$ I know the desired output i.e the desired position $\implies$ model tell the best position for a person that is not in the data. This is Supervised learning since I have examples of both input and target values.

NOTE The core of machine learning is to build a model that can be used to predict the output at runtime/inference time $\implies$ **Generalization**: extract knowledge i.e. model that can be applied to more general context on new data. The output is not an algorithm but a model that you train, customize, fit given the data and then apply on new data.

NOTE In unsupervised learning you don't have the desired output but just the data $\implies$ by looking at the data build a representation to be used for reasoning $\implies$ **build higher level knowledge, exploiting regularities** (e.g. clustering the study plan in different categories and then by looking at similarities: I don't know if a category exist , I only look at similarities)

NOTE An example of reinforcement learning is dog training: for sure not unsupervised learning (i.e. let the dog learn by himself), not supervised (i.e. not showing examples), but allow him to do some actions and receive from the environment reward or punishment and based on this they **learn** what is the **best policy**: not supervised you don't see the desired output and there is a supervisor that gives a sort of weak feedback to the system and you have to learn (common in robot: system interacting with environment learning a policy e.g controllers (action, perception cycle with feedback from the environment coming from the perception)

The two main problems we solve through Supervised Learning 2 are:

- **Classification**: labeled images i.e. input x $\implies$ extract some features from the images (e.g presence of wheels) $\implies$ predict the label of the image $\implies$

learning is to learn the **classifier** which **receive the features of the input image and tell the class to which belongs to**: the classes are a set of labels, not a number, and may be a 0/1 decision (e.g. 0 car 1 motorcycle) $\implies$ during leaning phase you know if prediction is done well or not.

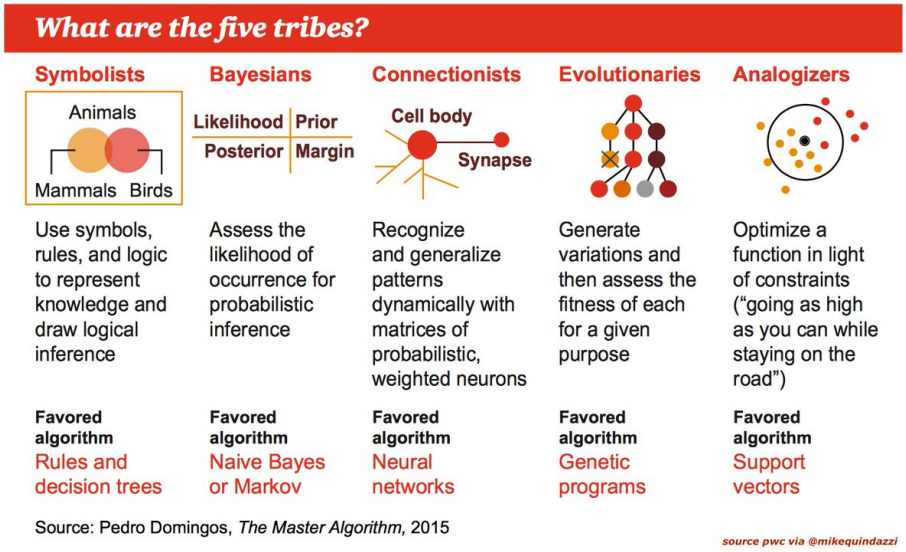
- **Regression**: is the continuous counterpart of classification $\implies$ regression provide a number $\implies$ the function goes from real to real domain $\implies$ during learning is possible to compute the error that tells you how much you were off from the target output $\implies$ you can use different loss functions and different models to predict this.

An example of Unsupervised learning task is **clustering**: based on some similarities it can distinguish the group to which the input belongs. initially there is no group distinction, then the **model find some correlation and can predict the belonging to one or other group it "detected" during learning**. We will mostly focus on Supervised Learning and Unsupervised Learning.

1.2 Deep Learning

Discuss what is Deep Learning in terms of differences with reference to the classical paradigms of Supervised Learning and Unsupervised ones.
The objective of research in machine learning may be seen as the discovery of a master algorithm that could learn any sort of Dataset $\implies$ **No-free-Lunch theorem**: **the best algorithm depends on the dataset, there is not a master algorithm**. No free-lunch theorem tell us that does not exist a single best algorithm but, depending on the problem, one algorithm can be better than another.

A classifier is a criterion that separate one class from the other $\implies$ learning the separating boundaries between different classes if we are lucky the decision is easy to take because our data is distributed in the input space in a way that the separating boundary is a linear function (can be found for e.g. via logistic regression) but the separating boundary to take the decision might be more complex. Considering the different classes of machine learning algorithm they differ in the kind of boundary they can draw and the procedure to find the boundary. In some cases they do not model the boundary but the probability distribution of the points and then they classify the points based on the maximum a posteriori probability (e.g Bayesian techniques: model the data distribution and the use the data distribution to decide for the class).



So basically the objective of machine learning is to find the algorithm that can draw the line every time the line exists. The problem is that ML focused on finding that line but feature have to be good enough to shape the space in such a way the line exist: **selecting features that are not correlated to the class you might end up in a situation where drawing the line is impossible** since **all the regularities in the data are hidden and is not possible anymore to classify the data**. **Machine learning** focused on the classifier and other techniques **assuming that the features are correctly selected**, otherwise if features are not correctly selected the dynamics/structure of the data is not correctly captured, ending up in a bad model. The **idea** to **improve** with this **approach** is to **work also or only on the feature side** i.e. **what if you don't know the relevant features such that the problem could be solved** with a simple logistic regression (this would be a good case: simple classifier, efficient and do not overfit) $\implies$ if data are well behaved the classification is well behaved. This idea isn't new since before Unsupervised and Supervised learning were combined getting semi-supervised learning, obtaining a similar result: having a set unlabeled data you try to extract features as generic as possible in principle adapted to any Task applying unsupervised learning (Clustering) to the unsupervised features, i.e. extracting relevant correlation implied by those unlabeled images. Now that the data is described by these features vectors we apply the supervised part of the paradigm using few labeled images from which you extract the relevant features selected by the Unsupervised learning using those for the learning process of a classifier: labeled image must be used to learn the classifier because it needs to know what are the classes to draw the line in the feature space (i.e. task relevant features space, in a sense learned by an unsupervised approach). Advantages of semi-supervised learning comes from the fact that only few images must be labeled to learn a classifier: because the features were optimized in order

to make the life easier for the classifier $\implies$ the classification process becomes the following: **extract general** (i.e. valid for each task) **features which are not task specific**, **feature projection to task relevant ones (learned trough unsupervised learning)** and then classify using the learned classifier (learned trough supervised learning). **First general features are called low level features and the one selected through unsupervised learning are called Mid/high level features**.

Before deep learning (2013) the classification task was resolved with semi-supervised learning.

Transfer Learning Instead of the task related features are used general features, features from other domains (e.g. random images from the web) from which are extracted some features then used to classify the data for my task. An example is Text classification. In a text the general feature is the dictionary of word considered, since it is not task specific for example we can take all wikipedia pages (i.e. unlabeled documents) and build the dictionary in a unsupervised way. To distinguish the type of documents you should then learn the classifier with labeled documents (scientific, legal ...). So transfer learning means this **use of knowledge that are unlabeled or unrelated to the task to improve the task**.

NOTE Labeled example are used to learn how to draw the line in the feature space. For example: Unlabeled data are cars and motorbikes, so those are used to get general features, but then if I ask to the model to classify if the object is pointing right or left we don't know how to answer if we don't **describe the task to the model through** the few **labeled** images the classifier cannot understand the task it has to solve.

The **difference** between **Machine Learning** and **Deep Learning** is **how much human is involved in feature engineering and design**: **in ML** the **low level features extracted from data are defined by a human**, so **what if the features from which we extract the task specific features (through unsupervised learning) are wrong?** Instead in **deep learning** the **whole feature extraction**, **from raw data to the classifier input, does not involve human i.e there isn't feature engineering anymore, the machine is asked to learn what is the best selection of features**. So deep learning is **not only learning the classifier** (as in ML) **but also the representation of the data**: Machine learned **features optimized for the task** and classifier is easier to learn i.e **deep learning is finding the best space to put the points so that they can be easily separated**. So deep learning **is about feature learning, data representation**, learning from data what is the **best set of features** that can be used to perform classification, regression and so on.

NOTE Difference between semi-supervised and transfer learning: if data used for classification include data different (i.e. not much correlated) from what you are

doing in classification you are using transfer learning (TL will be seen later on in the course).

Hence deep learning uses machine learned features optimized for the task that make the classification problem we are trying to solve easier $\implies$ the idea of deep learning is to learn features that will make our classification easier: **the way to learn data representation is usually done by building a sort of hierarchical representation optimized for the task**: so **various learned hierarchical layer of features** going **from lower level features to higher level one**. Deep learning, as said, is abut learning data representation from data.

NOTE In deep learning features are not only selected by the algorithm, i.e. computed from data as in ML, but are even extracted, i.e. learned.

NOTE DL will learn features that are the best for the classifier $\implies$ DL is also called **end-to-end learning**: you have **input** (image) **on one side** and then the **output** (label) **on the other**, all the things in the **middle are part of the learning process**: when can distinguish between features and classifier but we train them together as a single model i.e. classification of the image through all the layers of abstract features $\implies$ **DL PROBLEM IS THAT IS REALLY DATA EAGER**: researchers are trying to find techniques to improve DL performances also without labeled images (e.g. word embedding) $\implies$ exist DL algorithm for supervised learning tasks (classification, regression) but also for unsupervised tasks.

For what concerns **features** you can look at them and guess what they mean but know the **exact interpretation is not always possible**: features extracted by the first layer are very very basic and low level and cannot be straight forwardly interpreted, and then **layer by layer the level of the features increases and its interpretation may be easier since may be possible to start to see some patterns**.

Another characteristic of DL is that the **model is a black box**: you input the data and get the result but the process that is done inside the boxes is difficult to understand (to understand the underling process we should resort to much more classical and structured learning algorithm).

NOTE Explanation is done with classification since is the easiest one to describe

NOTE DL use labeled input: DL has both task learning classifier and learning the features and the information you use to learn the features are the target class.

Hence, **instead of using the knowledge of the designer** DL **use data to extract the features**, letting the algorithm learn the representation by itself $\implies$ **DL is about learning data hierarchical representation optimized for data from data, to do this a lot of data is needed** $\implies$ How we **collect those data?** If you are

a big company or a small company you have a lot of labeled data.

Especially with unstructured data as images, speech, graphs and text DL is very good: (learn best features for the task and then learn the task) since **is very difficult for humans to extract the best features for the task**, i.e. is very difficult to understand the structure of the data.

According to MIT the **core problem of DL**, that was making the difference from other approaches, was the **massive computational power**, but its all about **(Big) data**.

ImageNET Is a collection of images classified according to a taxonomy of terms (1.5 million images - 1000 labels) i.e. is **a huge classification problem**: predict the most likely 5 concept which correspond to the images. Before 2012 the improvement was very limited since they were using a ML approach (human designed feature set to apply unsupervised learning). In 2012 using just a simple neural network researchers managed to get an improve of 10-15%, which corresponded to the improvement of the past 4 years. This result made clear that DL was a huge leap forward.

NOTE **Very unbalanced dataset may create some problems: DL is still ML so it struggle with unbalanced dataset or overfitting**. It's not the Domingo's master algorithm but it has the potential to improve significantly performances.

Since DL is able to extract features by itself why we don't use it for everything? **Not always we are able to gather a lot of labeled data**: sometimes if you have very special tasks for which you **known the relevant features, you can beat DL with a small amount of data**. **The problem DL solve is to automatically extract the right feature, but if those are known isn't worth to use it since is way more data and computational eager**; this is what **happens** with **very structured** input domain: **DL will still require a lot of data and computing power without giving astonishing better result than standard ML algorithm**.

Finally DL should be used when:

- Don't kow how to represent data, i.e. we do not know the structure of the data.
- Have lots and lots of data.
- (Most important thing) Compare your results against baseline: standard tools may work better than more powerful tool.

instead standard approaches are:

- Less data eager
- More interpretable: easier the technique easier is to explain what it does. DL is complex so is difficult to tell what it does (e.g DL: test for a disease that

tells you if you are affected, tells you nothing about the diagnostic process; standard ML: analyze the symptoms and then do the diagnosis)

Other advanced task are:

- Semantic segmentation: classify pixel according to their class individually (we will see it).
- Instance segmentation
- Image Quality Upscaling

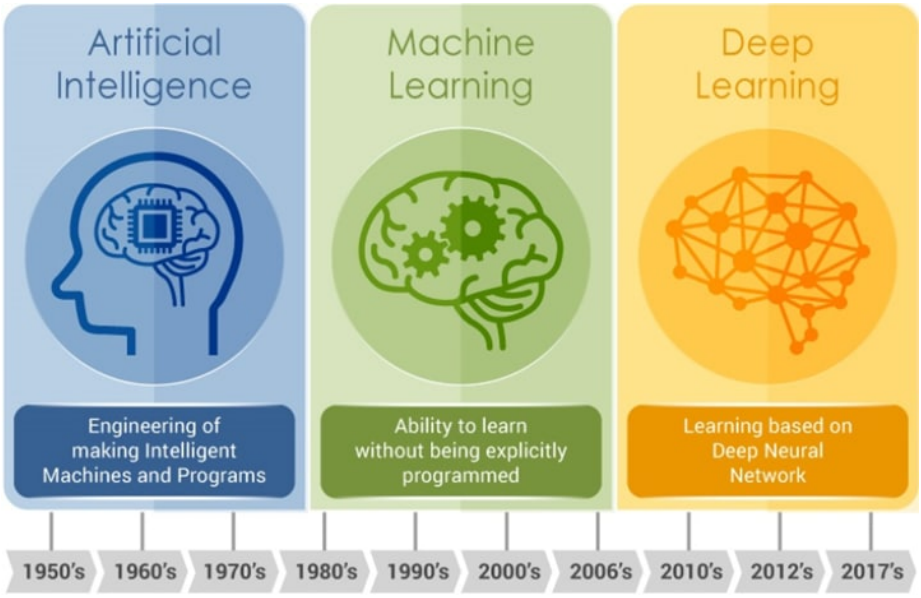
Domingo's master algorithm does not exist even with DL.
DL differs from standard ML since it extract the features used to perform the ML task: so at the same time learning and ML $\implies$ techniques to train jointly (supervised, unsupervised, reinforcement) $\implies$ **how I build models with a feature extractor and a classifier in such a way i can learn them end to end**: extract features + classification.
Nowadays to learn end to end models: able to extract non linear alternative representation and to perform classification the solution is to use huge Neural Networks; to make them perform as you want need to be structured and trained in the proper way.

2 Artificial Neural Networks

We know what is deep learning and we are going to study models used to implement DL: **Neural Networks**.

DEEP LEARNING IS NOT AI, NOR MACHINE LEARNING. AI is a discipline started in 1950's: goals was to create a model in such a way a machine by executing the code could reproduce some of the human intelligence aspects. Among several topics of AI (planning, multi agent systems, intelligent expects) there is machine learning. A **subfield of ML is DL**: approaches to **learn data representation jointly with classifier** $\implies$ effective when we don't have an **effective data representation i.e. unstructured data**. It is **totally wrong is associating DL with Neural Networks**: **DL is not about using deep (long) neural networks** $\implies$ **deep means** that **learning is pushed down** in the hierarchy between raw data and classification through all the layers of abstraction implemented by **features extractor**: then by chance they are implemented with neural networks.

NN are as old as AI: we will see the evolution of Neural Networks from perceptron (first ANN) and Feed Forward-NN and how they learn representation and so perform deep learning, to contemporary deep learning techniques.



2.1 Perceptron

Perceptron was the first ANN and it worked the way we saw: teach it by saying if each input was correctly predicted or not and it learns how to classify instances.

2.2 What is the Perceptron?

Was originally presented in the Dartmouth summer research project of Artificial Intelligence where was discussed about modeling and learning the functioning of human brain using NNs as a tool to solve problem computers could not resolve and study how programs can self improve; The concept of abstraction: how to layer the knowledge from sensor to concept by extracting categories or classes from the image. So **at the beginning of AI the goal was to model the processes of human intelligence** and **Perceptrons was the major tool to obtain that**.

Computers in the 40's were already good at:

- doing precisely what the programmer programs them to do
- doing arithmetic very fast

However they would have liked them to:

- **Interact with noisy data or directly with the environment**
- **Be massively parallel and fault tolerant**

- **Adapt to circumstances**

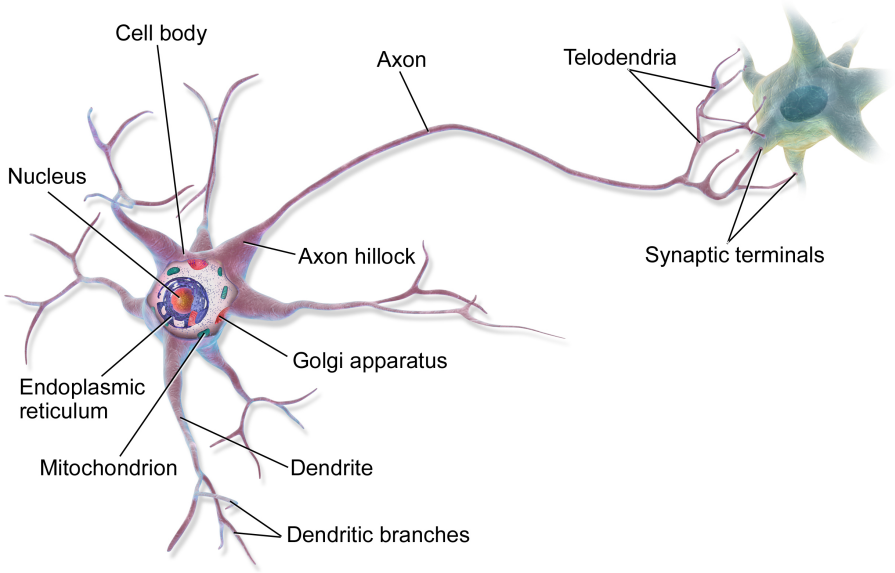
Since all these characteristics are very well managed by human brain they became interested in model the human brain: the idea of NN were in terms of hardware, computational model different from the standard Von Neumann Machine and could obtain better result similar to human brain. To do that they started to study and create how computation happens in the brain. Human brain has a huge number of computing units:

- 10^{11} (one hundred million) neurons
- 7000 synaptic connections to other neurons
- In total from 10^{14} to $5 * 10^{14}$ (from 100 to 500 trillion) in adults to 10^{15} synapses (1 quadrillion) in a three year old child.

The computational model of the brain is:

- Distributed among simple non-linear units that perform very easy computation
- Redundant and thus fault tolerant
- Intrinsically parallel since they are so many

So the idea to mimic the **brain computational model** was to start from the simplest basic unit called Perceptron modeling a neuron: combining many percepton to obtain an **hardware similar or at least comparable to the human brain**.

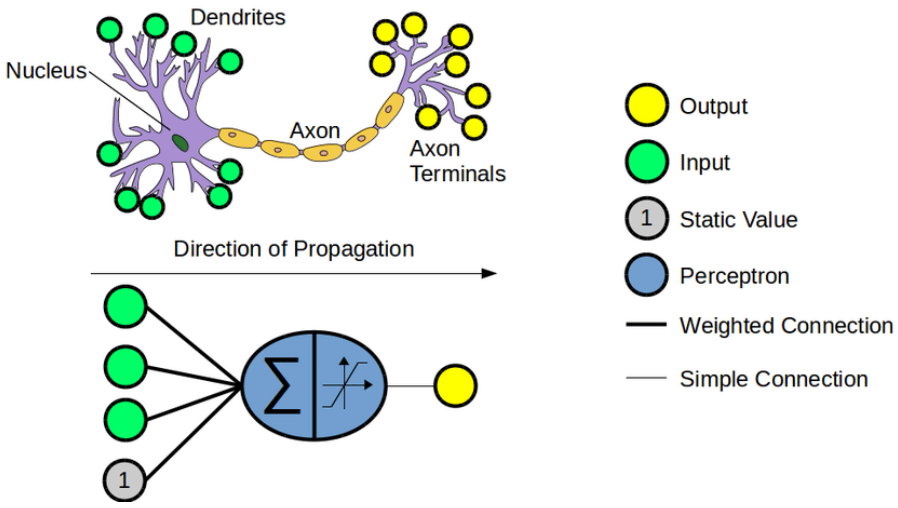


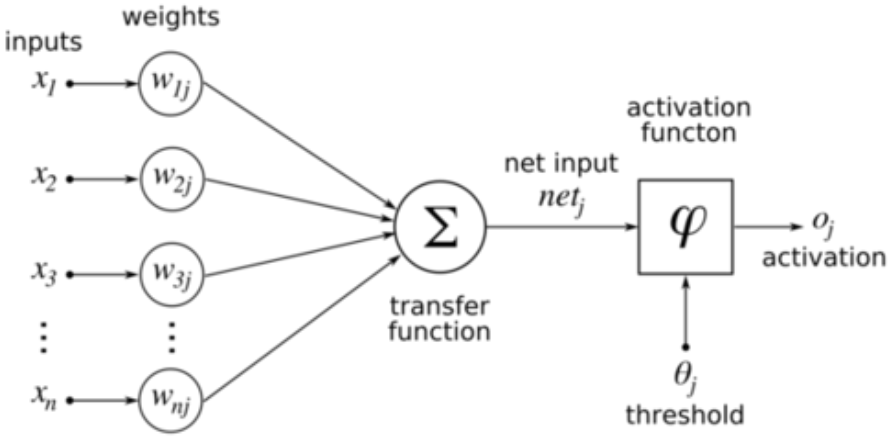
Synapses are the places where communication happens by an exchange of charge. A neuron cumulates charge received from adjacent neurons and deliver it to all

adjacent neurons through its synapses deciding to increase or reduce the charge it has cumulated.
Information is transmitted through chemical mechanisms:

- Dendrites collect charges from synapses, both inhibitory (negative) and excitatory (positive)
- Cumulated charge is released (neuron fires) once a threshold of charge is passed

These characteristics were simplified in the Perceptron model:





x_i are input signals sum ($\sum$) exemplify the cumulative effect of the cell. Once the charge is accumulated is compared with the threshold called bias: if the charge

accumulated $>$ bias $\implies$ (SUM - b > 0) else if is $<$ bias $\implies$ (SUM - b $<$ 0). Depending on the activation function the output value can change depending on the sign of (SUM - b). The output of this neuron as function of the input is a non-linear function h (the function computed by the neuron e.g. the step function): $h_j(x|w,b)$

$$h_j(x|w,b) = h_j(\sum_{i=1}^{I} w_i \cdot x_i - b)$$

or by considering the bias as a weight ($w_0 = -b$) on a input that is always 1 ($x_0 = 1$):

$$h_j(x|w,b) = h_j(\sum_{i=0}^{I} w_i \cdot x_i)$$

and from this:

$$h_j(x|w,b) = h_j(w^t * x)$$

Historically this has a great impact since you treat the bias as another weight and then you can learn it together with the weights. What is the relevance of the bias? The bias is so important because it plays the role of the constant term in the linear regression represented by $\sum_{i=1}^{I} w_i * x_i$ (the part before the activation function is a linear regression), so by removing it we remove the constant term of the linear regression i.e. your linear regression model goes trough the origin: **if you don't remove the bias you will be never be able to learn the model**. The bias **allow to move forward and backward the linear regression and vary the threshold among which the neuron is firing**. Sometimes the weight related to the bias is not shown but you have to remember that the number of input of a perceptron is the true number of inputs plus the fitticius input related to the bias and set to one and related to another weight w_0.
Furthermore , historically the activation function was the Heaviside function i.e. the step function from -1 to 1.

The first perceptron (1950's) was a piece of hardware that was learning only the weights while the biases were fixed either initialized random or to a fixed value, only after arrived the idea of representing the threshold value as a bias term (1960's).
What a perceptron can do?

- **OR**
- **AND**

Since the perceptron could implement these two operators by putting together **many perceptron I can get the same result of a logic network** i.e. I can compute the same computation a VN machine was doing at the time $\implies$ is interesting how **by tuning the weights i can obtain different behaviours**. Why don't we simply use the or or the and gate? Those two gate are different from the other, what we would like is a sort of programmable logic: by tuning the parameters we select the behaviour of the perceptron, but more importantly those weights can be learned by giving examples.

NOTE w_0 cannot be learnt alone, to learn more complex multi-input function we need to adapt more weights.

The great point of perceptron is that we are able to replace the classical boolean logic needed to execute programs with hundreds of perceptrons so we don't need to learn programs but we only need to learn weights. We want the perceptron to learn any boolean function: we showed with the simple example of *and* and *or* that they are as powerful as the boolean logic but we don't need them to learn boolean operators but truth tables i.e. functions by showing example. We will forget about instruction fetch, memory ... just by showing example to a bunch of perceptrons we are able to set our decision and learn the weights.
The **power of perceptrons** is the fact that the **weights are not set but learnt by showing only some example**: if the machine gets it right I say ok, otherwise I say wrong and adjust the weights. This procedure will converge to the set of weights able to classify the examples, and this is done only by saying yes or no to the class.

The idea to perform this learning comes from Donald Hebb, thats why is called **Hebbian Learning**. He found out that "The strenght of a synapse increases according to the simultaneous activation of the relative input and the desired output". This means that in a real neuron, if the input is high and the output is high next time it will be easier when the input is high to get an high output. Hebbian learning can be **summarized** by the following **rule**:

$$w_i^{k+1} = w_i^k + \Delta w_i^k \quad \forall i$$
$$\Delta w_i^k = \eta \cdot x_i^k \cdot t^k$$

so **whenever you make a mistake you modify** the synaptic **weight**, considering:

- i-th perceptron
- η: **learning rate** i.e. how much we modify the synaptic weight, so it is how fast is the train.
- x_i^k: the i^{th} perceptron input at time k
- t^k: desired output at time k, depending on the activation function (step: 1, 0)

So this very simple formula mimics the natural hebbian learning mechanism. The value of the **weight** start from a **random initialization** and the **weights are fixed one sample at a time** (**online**), and **only if the sample is not correctly predicted**.

NOTE We will see what happens if the learning rate is too high or too low.

Nowadays perceptrons are very rarely used, but with this example we start to understand concept like iteration, step size, batch learning, stability of training and Stochastic gradient descent. **Basically Hebbian Learning is implementing stochastic gradient descent**.

NOTE **If η is too high the learning can become unstable with standard learning**, but this is a special kind of model: **since it is an affine model you can use any η you want but in general can create problems**.

NOTE There is an **equivalence between perceptron and boolean algebra**, and because of this **anything that can be computed with bool algebra can be done with a perceptron**. At the very beginning the perceptron was an hardware, thought to replace Von Newmann architecture: the nice thing to this piece of hardware is that based on the weights it learned it behave differently; furthermore the weights are not programmed but are learned through the submission of a lot of examples, so programming is replaced by this procedure. The idea of Hebbian learning is to mimic in the learning procedure what was observed in the biological neurons: "**when input and output trigger simultaneously then is more likely than they will trigger simultaneously in the future**" and what is observed is that the synapsis i.e. the weight that say how much the output depends on the specific input is increased. Hebbian learning can be written in a simple update rule for which all weights are updated when the perceptron makes an error.

2.3 Example of Supervised Learning with Hebbian learning on a perceptron

Since perceptron model is really naive, the training algorithm is naive too and can be done by hand; as we see more complex non-linear models will be not possible to do an exercise by hand since the model will be to complex. What we do in this exercise will be still valid for complex NN, what will be different is the rule according to which we update the weights.
This exercise will help us to enter in the mindset of what does it mean to perform **online learning**, update the weights, an epoch of training and other concepts. It turns out the **learning rate η is not really critical for perceptrons** but we will consider anyway to show how does it work and what is the meaning of having a learning rate.

Lets learn the weights to implement the OR operator:

- Start from random initialization of weights, e.g.:

$$w^{(0)} = [w_0, w_1, w_2]^{(0)} = [1, 1, 1]$$

- Choose a learning rate e.g.:

$$\eta = 0.5$$

- Cycle through the records adjusting the weights for those that are not correct
- End the learning process (i.e. stop cycling) once **all** the records are correctly predicted.

By doing this we obtain a model that is able to predict correctly all the data, i.e. **the learning process converged to the solution**.

x_0	x_1	x_2	t_{OR}
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1

Since we want to learn the OR function we consider its truth table: we will use the truth table written in 1 and -1 since it is very handy for the computation and as we will see later on it's a matter of translating the point but the result is the same. where we consider the **bias as an input whose value is always 1**.
The output of the perceptron can e seen as:

$$h(w^T x) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a = 0 \\ -1 & \text{if } a < 0 \end{cases} = Sign(w^T x)$$

so if $Sign(w^T x) = 0$ it means that surely the record is not correctly predicted and I have to change the weights Let's start to iterate through the examples from the truth table:

(Row 1)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1 - 1 - 1) = -1$$

CORRECT PREDICTION

(Row 2)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1 - 1 + 1) = +1$$

CORRECT PREDICTION

(Row 3)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1 + 1 - 1) = +1$$

CORRECT PREDICTION

(Row 4)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1 + 1 + 1) = +1$$

CORRECT PREDICTION

The pass through all the data gave the correct prediction for each example, so it was a lucky initialization. Let's see what happens if the weights initialization is different:

$$w^{(0)} = [w_0, w_1, w_2]^{(0)} = [-1, -1, -1]$$

this time:

(Row 1)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(-1+1+1) = Sign(1) = 1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(1)} = w_0^{(0)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = -1 + 1/2 * 1 * (-1) = -3/2$$

$$w_1^{(1)} = w_1^{(0)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = -1 + 1/2 * (-1) * (-1) = -1/2$$

$$w_2^{(1)} = w_2^{(0)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = -1 + 1/2 * (-1) * (-1) = -1/2$$

i.e. for each synapses it looks at the sincronous activation of both input and output:

- if the input is $+1$ (or -1) and the output is $+1$ (or -1) the **product is positive and the synapses weight increase**
- if the input is $+1$ (or -1) and the output is -1 (or $+1$) the **product is negative and the synapses weight decrease**

Now the weights are updated to:

$$w^{(1)} = [w_0, w_1, w_2]^{(1)} = [-3/2, -1/2, -1/2]$$

This procedure where we use **immediately the weights updated** with the last input is called **online** update; the alternative is to go through all the data, collect all the errors and then **update the weights at the same time considering all the errors** and is called **batch** update. With Hebbian learning is not possible to use batch update but we will see other learning algorithm.

(Row 2)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(-3/2+1/2-1/2) = Sign(-3/2) = -1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(2)} = w_0^{(1)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = -3/2 + 1/2 * (1) * (1) = -1$$

$$w_1^{(2)} = w_1^{(1)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (-1) * (1) = -1$$

$$w_2^{(2)} = w_2^{(1)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (1) * (1) = 0$$

Now the weights are updated to:

$$w^{(2)} = [w_0, w_1, w_2]^{(2)} = [-1, -1, 0]$$

(Row 3)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(-1-1-0) = Sign(-2) = -1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(3)} = w_0^{(2)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = -1 + 1/2 * (1) * (1) = -1/2$$

$$w_1^{(3)} = w_1^{(2)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = -1 + 1/2 * (1) * (1) = -1/2$$

$$w_2^{(3)} = w_2^{(2)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = 0 + 1/2 * (-1) * (1) = -1/2$$

Now the weights are updated to:

$$w^{(3)} = [w_0, w_1, w_2]^{(3)} = [-1/2, -1/2, -1/2]$$

(Row 4)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(-1/2-1/2-1/2) = Sign(-3/2) = -1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(4)} = w_0^{(3)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (1) * (1) = 0$$

$$w_1^{(4)} = w_1^{(3)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (1) * (1) = 0$$

$$w_2^{(4)} = w_2^{(3)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (1) * (1) = 0$$

Now the weights are updated to:

$$w^{(4)} = [w_0, w_1, w_2]^{(4)} = [0, 0, 0]$$

We did what is called **epoch** i.e one pass through all the data; instead **iteration** is each update of the weights vector: until now in this last example we did 4 iterations in one epoch. Obviously in batch learning iterations and epochs are the same since each iteration is done after one epoch. There are even case where the iteration is done each n datapoints (we will see why we may want to do this). Now that we have done one epoch we don't have finished yet since the algorithm **converge to the solution only after a epoch is done without doing any iteration i.e the last update of the weights is correct for each example** (test with all the examples): when we update the weights to fix the output for a certain example the new weights may not work with a previously correctly predicted example, so we have to be sure the weights lead the perceptron to a correct prediction for each example. Under some condition Hebbian learning will converge and in general each epoch will decrease the errors until it reach the **convergence which may have either 0 error (optimal condition) or a certain amount of errors (sub-optimal condition) depending on the problem**. So let's try to continue with the next epoch:

(Row 1)

$$y = Sign(w^T x) = Sign(w_0x_0+w_1x_1+w_2x_2) = Sign(0+0+0) = Sign(0) = 0$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(5)} = w_0^{(4)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = 0 + 1/2 * (1) * (-1) = -1/2$$

$$w_1^{(5)} = w_1^{(4)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = 0 + 1/2 * (-1) * (-1) = 1/2$$

$$w_2^{(5)} = w_2^{(4)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = 0 + 1/2 * (-1) * (-1) = 1/2$$

Now the weights are updated to:

$$w^{(5)} = [w_0, w_1, w_2]^{(5)} = [-1/2, 1/2, 1/2]$$

(Row 2)

$$y = Sign(w^T x) = Sign(w_0x_0+w_1x_1+w_2x_2) = Sign(-1/2-1/2+1/2) = Sign(-1/2) = -1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(6)} = w_0^{(5)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = -1/2 + 1/2 * (1) * (1) = 0$$

$$w_1^{(6)} = w_1^{(5)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = 1/2 + 1/2 * (-1) * (1) = 0$$

$$w_2^{(6)} = w_2^{(5)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = 1/2 + 1/2 * (1) * (1) = 1$$

Now the weights are updated to:

$$w^{(6)} = [w_0, w_1, w_2]^{(6)} = [0, 0, 1]$$

(Row 3)

$$y = Sign(w^T x) = Sign(w_0x_0+w_1x_1+w_2x_2) = Sign(0+0-1) = Sign(-1) = -1$$

WRONG PREDICTION

so we change the weights with Hebbian learning:

$$w_0^{(7)} = w_0^{(6)} + \eta \cdot x_0 \cdot t_{OR}^{row1} = 0 + 1/2 * (1) * (1) = 1/2$$

$$w_1^{(7)} = w_1^{(6)} + \eta \cdot x_1 \cdot t_{OR}^{row1} = 0 + 1/2 * (1) * (1) = 1/2$$

$$w_2^{(7)} = w_2^{(6)} + \eta \cdot x_2 \cdot t_{OR}^{row1} = 1 + 1/2 * (-1) * (1) = 1/2$$

Now the weights are updated to:

$$w^{(7)} = [w_0, w_1, w_2]^{(7)} = [1/2, 1/2, 1/2]$$

(Row 4)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1/2 + 1/2 + 1/2) = Sign(3/2) = +1$$

CORRECT PREDICTION

As we said earlier even if the last prediction was correct we have to check if the set of weights work with all the other examples, so let's continue with the next epoch:

(Row 1)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1/2 - 1/2 - 1/2) = Sign(-1) = -1$$

CORRECT PREDICTION

(Row 2)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1/2 - 1/2 + 1/2) = Sign(1/2) = +1$$

CORRECT PREDICTION

(Row 3)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1/2 + 1/2 - 1/2) = Sign(1/2) = +1$$

CORRECT PREDICTION

(Row 4)

$$y = Sign(w^T x) = Sign(w_0 x_0 + w_1 x_1 + w_2 x_2) = Sign(1/2 + 1/2 + 1/2) = Sign(3/2) = +1$$

CORRECT PREDICTION

Now we can consider the algorithm to be converged into a set of weights that can correctly predict each example.

Two important things to know about the Hebbian learning algorithm is that:

- **Does the procedures always converge? Not necessarily**: in general with Hebbian learning is **possible to loop infinitly without reaching an end**; How do I deal with this cases? **set a maximum amount of epoch for which you stop the algorithm anyway**: this maximum amount of epochs **depends on the time we have and how many epochs we really need**.
- **Does it always converge to the same set of weights?** As we can notice from the two past examples the algorithm converge in two **different sets of weights** but both can correctly predict all the truth table.

To understand the reason of the two answers we should see more in depth the math behind the perceptron:

$$h_j(x|w) = h_j(\sum_{i=0}^{I} w_i \cdot x_i) = Sign(w_0 + w_1 \cdot x_1 + ... + w_I \cdot x_I)$$

i.e computes a weighted sum $w_T \cdot x$ and return the sign of the result. So the set of points for which our perceptron is in doubt for the sign (either $+1$ or -1) is:

$$\mathbf{w_0} + \mathbf{w_1} \cdot \mathbf{x_1} + ... + \mathbf{w_I} \cdot \mathbf{x_I} = \mathbf{0}$$

Basically the perceptron does a linear classification: linear **because** the **decision boundary** upon which you decide one class or the other is a line. It is a linear classifier for which the decision boundary is the **hyperplane**:

$$w_0 + w_1 \cdot x_1 + ... + w_I \cdot x_I = 0$$

In 2D, this turns into:

$$w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$

$$w_2 \cdot x_2 = -w_0 - w_1 \cdot x_1$$

$$x_2 = -w_0/w_2 - w_1/w_2 \cdot x_1$$

that is clearly a line with known term $-w_0/w_2$ and first grade term $-w_1/w_2 \cdot x_1$ In higher dimensional spaces (e.g. 10 dimension space) isn't anymore a line but **still a linear combination of the input and linear combination in a n dimension space** is called **hyperplane**. So the perceptron what is doing is **looking for the hyperplane that separates all the points** that are classified as $+1$ (positive class) above the plane and all the points that are classified as -1 (negative class) below the plane. From this we can see some interesting **properties** of the perceptron:

- **Multiplying by a constant term all the weights you get always the same hyperplane because the hyperplane will be 0 in the same point**: this is called **affine set**. This is why the two example converged to two **different set of weights that represent the same hyperplane**. For this reason the value of the learning rate for the perceptron is not important.
- There are several decision **hyperplane** which separate the point: the one we **find depend on the initialization** (e.g starting from $w = [0, -1, -1]$ I would ended up in a different hyperplane): we can notice that the set of weights obtained through the learning algorithm is different in the two cases: based of the weight initialization the solution of the algorithm may be different.
- **If exists an hyperplane that divides the point of the dataset in the two desired classes the Hebbian learning will converge, while if the dataset cannot be divided by an hyperlane the learning can remain stuck in an infinite loop of weights update**: at each update some points are still on the wrong side of the hyperplane and the update continue to happen.

- **The learning rate η is a measure of the changing of the hyperplane**: **changing** the learning rate **smaller and smaller could help in the case there isn't an hyperplane that divides the set**: the **oscillation of the hyperplane at a certain point will end since the learning rate will be infinitesimal and the algorithm will converge to an hyperplane that can make some classification errors** since the set cannot be divided precisely by an hyperplane. So **changing the learning rate might affect the solution but this is not true for the Hebbian learning and the perceptron** since it is so simple that changing the learning rate does not affect it at all, being an affine set.

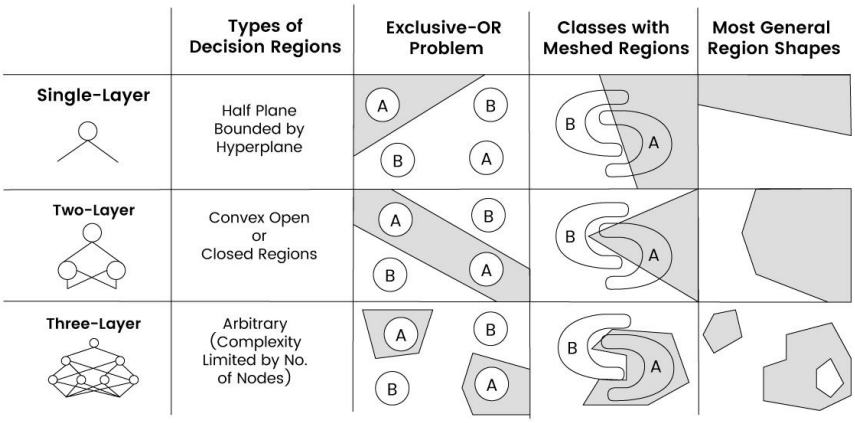
So the perceptron is nothing but a linear classifier that gets a set of inputs and decides if the item should go above or below the hyperplane, so classified as 1 or -1. If the input are 0/1 the **target has to remain +1 and -1** since the perceptron is a **classifier based on the sign function has as output +1 and -1**, but what happens is that the points in the plane change their coordinates, so it is just a linear transformation of the space and because of this the perceptron will still be able to learn the solution, and the algorithm will converge to a separating hyperplane. The target must be coded as $+1$ and -1 because of the algorithm: it considers the perceptrons a classifier of $+1$ and -1. When this algorithm was introduced they didn't know about this implications: they just **developed this non-linear model (step activation) taking inspiration from biology to derive the update equations**.

2.4 What are the limit of the Perceptron

Boolean operators AND and OR can be seen as linear boundaries: any hyperplane that divides the points of the sets correctly can be a solution of the learning. The **limitations of the perceptron** come from the fact that **not everything is linearly separable** and this can be easily highlighted by the **XOR function**: doesn't matter on how many dimensions you are working on, the separating boundary is an hyperplane and if the points cannot be separated with an hyperplane it's useless the perceptron cannot learn it. Since the perceptron does not work anymore we need **alternative solution**:

- Consider a **non-linear boundary**: **if data are not linearly separable** is possible to find a linear boundary
- **Change the input representation** in such a way the perceptron could take it apart, i.e. change the feature space in which the points become linearly separable.

in a sense **both ideas** were suggesting to use what were called **multi-layer perceptrons**: putting together several layers of perceptrons **could fix the issue**; such as was done to create complex boolean function, by connecting several perceptrons.



This figure summarizes the way the boundary changes by considering a single perceptron or multi layered perceptrons highlighting the type of data they are able to classify: **putting more layer of perceptrons creates more complex separating boundaries**. The idea is very straight forward but the **issue is that Hebbian learning does not work anymore on multi-layered perceptrons**.
They knew the solution to the XOR problem but the solution was only theoretical since a substitute for Hebbian learning had to be found. Let's see the difference of what happens if we applying hebbian learning to a single perceptron and on a simple multi-layer perceptron composed by two single perceptron queued:

- Single perceptron: for each input signal we know the target

$$w_j^{(i+1)} = w_j^{(i)} + \eta \cdot x_j \cdot t_j$$

- Two queued perceptrons: since we only know the desired target of the entire network and its input, for the first perceptron we do not know the desired target:

$$w_{1j}^{(i+1)} = w_{1j}^{(i)} + \eta \cdot x_j \cdot \mathbf{?}$$

  and consequently for the second perceptron in the queue we don't know the input:

$$w_{2j}^{(i+1)} = w_{2j}^{(i)} + \eta \cdot \mathbf{?} \cdot t_j$$

  so it's clear that the **update of the weights is not possible since we are missing important informations**.

So it's clear how the **problem does not lay in the model but in the learning algorithm** that does not works if there isn't a direct connection between input and target, so it does not work for multi-layer perceptrons.

NOTE Why **don't we solve directly in closed form?** It is a **highly non-linear problem** (in two unknowns for the queue of two perceptrons) so **cannot be solved directly**.

NOTE Why don't we consider the input of the first neuron in the two neuron queue to be $x \cdot w_1$? First of all we don't know if it's correct or not because we don't know if w_1 is correct or not, so we might end up to correct w_2 using the wrong w_1 and to correct w_1 we have to know the correct target of the first perceptron which is unknown.

2.5 What is the difference between Perceptron and today's NN

The **solution** proposed tried to **blame on each of the weights part of the error of the target**. With this the biological metaphora finishes, and the research stated to be approached in a more mathematical perspective. For example Widrow try to assume that all the neurons were linear: if this is the case the relation between input and output is linear so we can blame each of the weights accordingly, but unfortunately also means that the separating boundary is linear so the problem is solved on the learning side but not on the separating hyperplane side.
Several rules have been proposed to replace the Hebbian learning as training algorithm, but the most successful one that is still used is the **backpropagation**: this means to be able to **backpropagate the error of the output touching all the weights** i.e. **blaming all the weights accordingly starting from the nearest layer to the output to the furthest one**. **In order to apply the backpropagation algorithm the activation function must be differentiable**: the one used in the perceptron is not differentiable since is a step function, so the biggest problem of the perceptron was the fact that it had the step function (Heaviside function) as activation function; that's why we don't use anymore the term multi-layer perceptrons: we have to replace the classic activation function of the perceptron; the term we use for **multi-layered perceptrons with different activation functions** is **Feed Forward Neural Networks** (FFNN).

3 FFNN

Perceptron was the very first step in neural network era, but it had limits: it couldn't solve the XOR problem. Everybody knew that XOR problem could be solved with multi-layer perceptron but the problem was that there wasn't a learning algorithm to apply since Hebbian learning is not applicable to multi-layer perceptron. The most successful learning algorithm introduced back then is **backpropagation**. The latter algorithm need a **different structure of the neuron**, so we don't talk anymore of multi-layer perceptrons since they are made out of perceptrons and perceptrons have as activation function the Heaviside function, but we talk about **Feed Forward Neural Networks** and **artificial neurons**.

FFNN have a similar topology to multi-layer perceptron:

- **Input layer**: set of neurons connected to the input of the problem. Is **not computing any function, takes the input and propagates the input signal to the first hidden layer**.
- **Output layer**: represent the predicted output of the network, composed by **as many neurons as the output of the problem** that **receive the transformed input (through hidden layers) and then compute the output**.
- **Set of hidden layers**: **intermediate layer** each of which is a layer of artificial neurons **computing a non-linear function of their input**. Since they are not observed by input nor by outputs they are called hidden.

One possible way to **solve the linearity issue of the perceptron** is to compute a **non-linear boundary or change the input in such a way the linear output can be applied to solve the problem**. The example **FFNN is doing both** of this things: on one hand is changing the input into a **new representation** for 3 times **through the hidden layers** that transform the data **from the input space to a new hidden space** which are 3 non-linear transformation done on the input so on the other hand the **output layer produce a highly non-linear output**. The **result** is a **non-linear model** characterized by the number of layers, the number of neurons for each layer, the activation functions and the values of weights. So there are **several parameter** to build this model: **weights** are the true parameter of the model, while the other are called **hyperparameters** since **once they are decided the number of parameter to train, i.e. the number of weights is decided too**. Hyperparameters (**learning rate, activation functions, layer number, neuron number for each layer**) are all the things you have to decide to build the model. For now we consider the hyperparameters decided, we will see how to choose them.

Layers are connected through weights. A **single connection between two layers** can be **represented** through the **matrix of weights**, in general:

$$W^{(l)} = \{w_{ji}^{(l)}\}$$

where l is the layer, j is the index of the hidden neuron and i is the index of the input for a total of $J \cdot I$ weights. If some connection do not exist than the **weight can be set to zero to erase the connection**, but for ease of thinking we assume the matrix is complete and we have a fully connected **hidden layer**. Is **said** to be **fully connected if each input is connected to each neuron** i.e. to each output of the layer; nowadays the term used is dense opposed to sparse. In a dense layer all the weights exist but they could be in principle zero from learning. So a **FFNN** can be **represented through a set of matrices containing the weights**.
The number of neurons is a hyperparameter so they can change: usually this number is bigger than the input length.

NOTE Designing the neural network leave a lot of freedom. We will see how we could in principle choose the best number of hidden layer and neurons. We will figure that there is no secret recipe: part of the **design of a deep neural network is a trial and error process** where you **try to fine tune how many neurons you should have on this hidden layers**. For the moment we will assume we can select hyperparameters in a good way.

An IMPORTANT thing to notice is that the **function computed by each neuron depends in terms of input only on the neurons of the previous layer**, but **going backward** we see that the **output depends on all neurons**. **For now we assume that the output of a layer depends only on the previous layer**:

$$h^{(l)} = \{h_j^{(l)}(h^{(l-1)}, W^{(l)}) \quad \forall j\}$$

There are some exceptions for example a connection that jumps one layer or more going to one of the next called shortcut connection, but the model we discuss works also in this case since we can consider a fitticius neuron that copies the input, and in such a way we see that shortcut connections can be represented as a special case neuron. We will see things in the context of a unique hidden layer but they generalize immediately to network with multiple layers and shortcut connection.
For now we **assume** we are **fully connected and the output of each hidden layer depends only on the previous one**. What is really important is that the **information flow goes only forward** and **from this characteristic comes the name**: the input signal feeding mechanism goes only forward, **there aren't loops** (**is not possible for the signal to go back**) and the **signal does not goes to neurons of the same layer** (not a big simplification since we can add a fictitious layer that separate the two). **If the signal goes back we have recurrent neural networks** (we will see them).
The **two requirement to train FFNN** is that **signal goes only forward** and **activation function are differentiable** (the fact we don't have loop prevent the FFNN to have a state as in finite state automaton: if we want to memorize a state or have a dynamic model implemented by the NN we need to add loop). **Differentiable function means that the function must have a derivative and the derivative must be continue**. In practice if all activation functions are differentiable it means that the **linear combination of differentiable function is still differentiable** and a differentiable function applied to a differentible function is again differentiable: this **means** that **the FFNN non-linear model no matter how complex it is can be derived, you can compute the derivative of the output w.r.t. all the parameters of the model**.

Now that we know what is a FFNN we miss only a piece to construct our first NN: **which activation function should we use?** It must be differentiable, but does any differentiable functions is ok? In principle yes, but **in practice** only 3 activation function have been used and somehow are related to the original model. Heaviside function is not differentiable in 0 since the derivative has a discontinuity in 0.

We want a differentiable function that allow us to compute the derivative of the entire model. **Commonly used** activation function are:

- **Linear activation function** i.e. the **output of the neuron** is the **weighted sum of the input**:

$$g(a) = a$$
$$g'(a) = 1$$

  but obviously **is not non-linear but** we **want non-linear one** since the **complexity of the model comes from the usage of non-linear function**.

- **Sigmoid activation function**: is a **sort of approximation** of the step function that **smooth the step function around zero**, i.e. is the **differentiable version of the step function**:

$$g(a) = \frac{1}{1 + exp(-a)}$$

$$g'(a) = g(a) \cdot (1 - g(a))$$

  the fact that the **value of the derivative is easily and efficiently computable is useful**.

- **Tanh** (hyperbolic tangent) activation function: is a **sort of approximation of the sign function** that **smooth the sign function around zero**, i.e. is he **differentiable version on the sign function**:

$$g(a) = \frac{exp(+a) - exp(-a)}{exp(+a) + exp(-a)}$$

$$g'(a) = 1 - g(a)^2$$

  as the sigmoid the fact that the **value of the derivative is easily and efficiently computable is useful**.

Which one to use? There are some **general rules** to pic one. **Remember that the linear combination of linear function is a linear function, so using for all the neuron the linear function would not help at all**. The **other** two functions are **basically interchangeable in the hidden layers**. For the **output really makes a difference** choosing the sigmoid or tanh. The only problem with FFNN and these activation functions comes when the **network is very very deep**, since a **phenomenon** of the **vanishing gradient prevents the network to learn** and the **ReLU (Rectified Linear Unit) activation function is the solution**.

To understand the criterion to choose from the three **activation functions** we should start from the **output**. The choice of the activation function of the output neurons **depends on the task**:

- **Regression** is a function for which the **output is a real** for each neuron, so the **output spans from** $(-\infty, +\infty)$ **i.e. the whole** $\mathbf{R}$ **domain**. If you choose sigmoid or tanh as activation function for an output neuron the output won't span in $(-\infty, +\infty)$ but respectively $(0, +1)$ and $(-1, 1)$. For example

using the **sigmoid** and considering the **output normalized between 0 and 1** the **problem** comes if in your data you **don't have seen the maximum value** (e.g regression of heights) the **model won't predict correctly the values since the model was told a wrong maximum value**, the one we had was not the real one (e.g. not able to predict 2.11 cm height if we considered 2 m as maximum height we had in our data). So in general you can pre-process the output centering into 0 and normalizing the variance, but in regression problems the domain should always be $\mathbf{R}$ **i.e. we should use a linear activation function for the output neurons**. **For all the other neurons can be either sigmoidal or tanh**, there is not a real significant difference (in some cases tanh is better but not a significant difference). As we said we **don't use linear activation function in the other neurons since with linear input and all linear neurons the output would be just a linear combination of the input i.e. the boundary would be linear** and not non-linear as we want to solve more complex problems. If the output layer is using linear activation function and **in the hidden layers tanh (or sigmoid)** since the **weights can have any real value** the tanh (or sigmoid) are **not really limiting the input of the last layer and so its output**.

- **Binary classification**: is a function for which the input is classified **choosing one of the two classes** that can be **encoded without any difference as**:
  - $\{0, 1\}$ i.e. **sigmoid as output activation**
  - $\{-1, +1\}$ i.e. **tanh as output activation**

  There is **no real difference in the two choices** but usually the second is more used since **being between 0 and 1 the output can be interpreted as a (posterior) probability, giving the probability of the input belonging to class 1 or class 0**.

- **Multiple classes classification**: is a function for which the **input is classified choosing one of the K classes**. **Usually** is done the **one-out-of-K encoding** also known as **one-hot encoding**: each class is represented by a vector in which the element of index i represents the $i-th$ class:

$$\Omega_i = \begin{cases} 1 & i-th \\ 0 & \text{others} \end{cases}$$

  so we must **use as many neurons as the number of classes since each neuron will output 1 if the input belongs to the class it represent**. There is **nothing that imposes that the sum of the output must be equal to one**, so if you want to **interpret it with probability it may be tricky**. In general we **would consider the class of the input the one associated with the neuron that outputs the maximum value**; but usually output neurons use a **softmax unit** (unit since can be represented as a small neural network that perform some computation but **since it does not contain trainable**

parameters inside it can be considered as an activation function):

$$y_k = \frac{exp(z_k)}{\sum_k exp(z_k)} = \frac{exp(\sum_j w_{kj} \cdot h_j(\sum_i^I w_{ji} \cdot x_i))}{\sum_{k=1}^K exp(\sum_j w_{kj} \cdot h_j(\sum_i^I w_{ji} \cdot x_i))}$$

this **normalization forces** the output of the neuron to have an **output vector that sums to one** and at that point you can **pic** the class with the **highest value** in terms of classification. If we want to have an output that sums to 1 we have to normalize: usually if normalization is not done inside the model with the softmax is done after on the output. Furthermore when you train the model you don't consider that the output is normalized , so the idea is that you already include the decision function in the output. So for a **binary classification we could use a single neuron with sigmoid activation function or two neurons with softmax activation function but the output would be different since the activation function is different**: in general **softmax is steeper than sigmoid but beside that the result is similar**. Softmax is used usually in the output layers and not in the hidden one.

There are **theorems** that say that **using a FFNN with one hidden layer and a S shaped functions (sigmoid, tanh...) you can approximate any non-linear function**, so using them would not limit the power of the model. To **increase precision we should add neurons**, and is the fact that we can add more that let us to approximate any function. Hence regardless the function we are learning, a single layer can represent it:

- It does **not mean that a learning algorithm can find the necessary weights**.
- In the **worst case, an exponential number of hidden units may be required**
- The layer **may have to be unfeasibly large and may fail to learn and generalize** (overfitting).

NOTE Mixing functions (sigmoid, tanh) nothing bad happens: the objective is to figure out the best mix set of function, but in practice there is no real advantage to mix so is not usually done.

NOTE We don't want linear models since if we need them we use linear regression, but if data are not linearly separable and we want to solve non-linear problems we cannot use linear models, but we have to resort to non-linear ones.

NOTE **More non-linear function we have more flexible is the model**: more hidden layer more flexible is the network as **polynomial increasing their degree**. Increasing the non-linearities we are projecting the input in a higher dimensional space, which means our model il more flexible and can learn more complex functions.

NOTE MATH interpretation of sigmoid: The **sigmoid** is **basically** a **logistic regression**, so the output is performing a logistic regression on the input: from ML we know that logistic regression is a linear classifier so we must transform the input with non-linear function so that in the output layer you discriminate between different classes by using logistic regression. The sigmoid does a logistic regression on the input that has been transformed into a non-linear space where the logistic regression is able to take apart the two classes.

NOTE In classification if we use only K neurons where k is the number of classes there is nothing in the model that prevents the k output to have more than 1 be 1 at the same time. This means the model cannot decide between more classes, so we normalize the output with the sum.

NOTE **Hidden layers** should be **composed by the sigmoid or tanh** to **generate a non-linear transformation of the input** for all three the categories; we will see that this choice in some cases can be improved using ReLU for all the hidden layers.

We saw the issue of the perceptron is the linearity of the separating boundary and the solution of this issue to deal with more complex functions is to implement non-linear separating boundary: to do this we can add multiple perceptrons (since for example one cut the hyperspace horizontally the other vertically and another one decide from the two classes). The problem of multiple perceptron works in theory but cannot be trained with hebbian learning. So we moved from non-linear boundaries implemented by several perceptrons to the ones implemented by FFNN. By using linear activation function for all the network we obtain linear transformations of the input i.e. a linear boundary, instead in multi-layer perceptrons since the activation function is the Heaviside function the transformations are non-linear i.e. the boundary is a non-linear one as we saw. **Changing our model to FFNN is done to allow training**.

- **one perceptron**: **function non-linear but linear boundary**
- **multi-layer perceptron**: **function non-linear and non-linear boundary**
- **FFNN** : non-linear boundary if the activation function are non-linear; if all the activation function are linear the boundary is linear.

3.1 FFNN training

Once we have designed the FFNN respecting the rule of: **fully connected**, **feed forward**, **non-linear activation functions**; **how** I **train** the **resulting parameters?** Once we designed the network we get what is called **FFNN** and **also known as universal approximator** from the **Universal Approximation Theorem** (Kurt Hornik, 1991)

Universal Approximation Theorem. *A single hidden layer feed-forward neural network with S (sigmoid, tanh) shaped activation functions can approximate* ***any measurable function*** *to* ***any desired degree of accuracy*** *on a compact set.*

i.e. **any non-linear continuous function can be approximated with any arbitrary accuracy considering enough neurons in the hidden layer**, so regardless the function we are learning **a single layer can represent it**. But the latter doesn't mean a learning algorithm can find the necessary weights i.e. **there is no guarantee we can find the proper set of weights to solve the problem**: in the worst case an exponential number of hidden units may be required to approximate with arbitrary accuracy. But if you have a such powerful model is true you can approximate any function but it might happen that the network **may have to be unfeasibly large and since has been trained with a limited amount of samples might fail to learn and generalize** (i.e. it might overfit)
The fact that the number of parameters may increase exponentially will not be touched since we don't want to put many neurons: **more neurons means more flexible model but it also increase the problem of overfitting**. We will see how to limit the number of neurons and to improve the model generalization.
The thesis of **the theorem speaks about measurable functions i.e. a single hidden layer is enough for regression**: if you want to do **classification exist another theorem that states that another layer is needed**. The **extra layer is to overcame the problem** of compact set and measurable function, since **classification is a discontinuous function**. **So to perform and succeed for regression is needed at least one layer and for classification at least two**. **In practice** have been observed that having **more layers with less neurons is more effective**: for example think about a new input that needs to be transformed into a new space, if this transformation is complex we would want to learn the sequence with simpler transformation whose composition give the complex one i.e. **every time a transformation is hierarchical is easier to learn a sequence of simple transformation than a single complex one**. For example text is a very hierarchical process: characters, words, sentence, paragraph, document; in practice splitting the task in a hierarchical way (learn to represent characters, then represent words ...) it is more simple to classify a document than classify the document all together.

NOTE The example we will see will contain a single hidden layer, so for the theorem they are capable of fit anything adding enough neurons, since is easier to do the calculation on.

NOTE For **hidden units** usually we indicate the **number of hidden neurons in a hidden layer** instead with **hidden layers** the **number of hidden layers**.

How do I train the weights? A **parametric model** is a **model described as a function of a set of parameters**, let's see how is trained for regression and classification (for ease of visualization let's assume we are dealing with regression, but everything works also for classification).

Given a **training dataset** of **pairs input-target**:

$$D =< x_1, t_1 > ... < x_N, t_N >$$

given a model function of x, we want to **find model parameters such that for new data**:

$$y(x_n|\theta) \sim t_n$$

i.e. in such a way the output is similar as much as possible to the target. In general y **is a parametric model** so it can be a linear function or polynomial function but in case of a NN can be rewritten as:

$$g(x_n|w) \sim t_n$$

i.e. **with** g **the non-linear function implemented by the neural network**, and to **obtain this we can minimize**:

$$E = \sum_n^N (t_n - g(x_n|w))^2$$

i.e. **minimize the square distance between the target and the output**, so finding the **least square fitting for the model**.

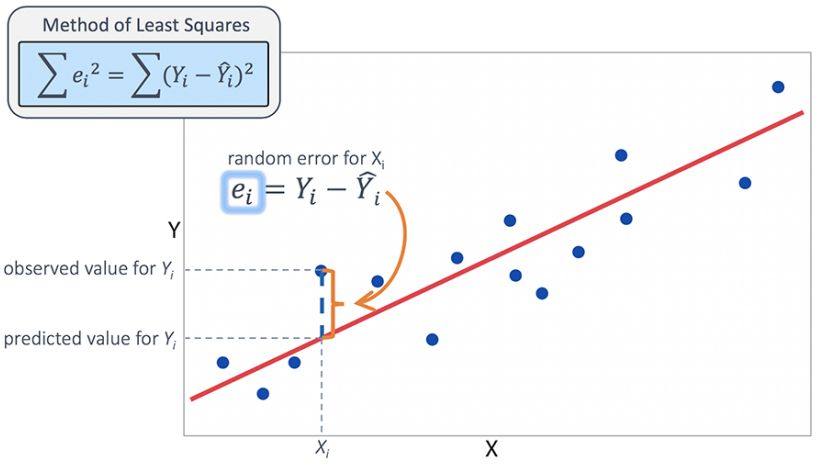


Figure 1: Caption

The image represent the result for a linear model that minimize $E = \sum_n^N (t_n - g(x_n|w))^2$, but **since we are analyzing a non-linear model the model can be much more complex**, but in every case it is minimizing $E = \sum_n^N (t_n - g(x_n|w))^2$ with the difference that E **will be lower as the model becomes more complex, more non-linear**. Using **too complex model could cause overfitting** a phenomenon that we should **avoid** and that we will analyze in future lectures.

NOTE We will do the examples with a unique output neuron and a single hidden layer, but the result can be generalized to many output and many hidden layers.

We saw how to build g but we didn't discuss **how to fit the model i.e. learn parameters**. With FFNN $g(x_n|w)$ is a **non-linear model**, so it is very flexible and we **cannot use the typical techniques valid for the linear models** (closed form solution, pseudo inverse solution, singular value decomposition,...) **to compute straight forwardly the parameters minimizing the error function**: if g is linear the minimum is easy to find, instead **if g is non-linear is not easy to find**. So **learning means** we want to **find the minimum of the error function**: the parameters w such that the error function is minimized. How we do this?

- **To find the minimum of a generic function**, we compute the **partial derivatives** of the function and set them **to zero**:

  $$\frac{\partial J(w)}{\partial w} = 0$$

  solving the system of equations, with weights as unknowns, the minimum is found. The problem is that finding the derivative and the **solution of the system can be never done in closed-form**: since in general is a solution of a **non-linear system** we **have to use iterative solutions**:

  - **Initialize the weights to a random value** (e.g. to zero)
  - **Iterate the weight update until it converges**:

    $$w^{k+1} = w^k - \eta \cdot \frac{\partial J(w)}{\partial w}\Big|_{w^k}$$

    This formula is called **gradient descent**: is very similar to the Hebbian learning but you **don't have the input and target directly but the error between the target and the output of the model** i.e the **idea is to modify the weights by changing them in the direction opposite to the one which is the derivative of the function with respect to the weight** i.e the gradient with respect to that parameter, so is the **direction of decrease of the function** (-gradient): if the gradient is positive the weight will decrease otherwise if it is negative the weight will increase.

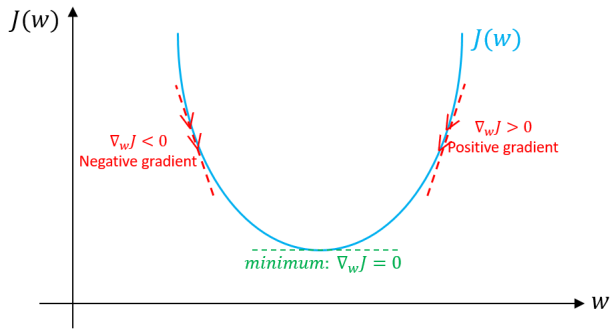


Figure 2: Gradient descent

In this way the parameters will be **updated until the minimum is reached**.

NOTE The gradient direction is the steepest direction, but considering the $-$ (minus) gradient we are considering the verse towards the minimum.

In the **latter example the problem is easy since the function is convex** that has a single minimum, but **in neural networks** things are much more complex since the **error function is a highly non-linear function of the parameters (weights)**. So **finding the weights of a neural network is a non-linear optimization**:

$$argmin_w E(w) = \sum (t_n - g(x_n, w))^2$$

NOTE We want to represent our neural network, a non linear function approximator: non-linear function computed once the number of layers, neurons and activation functions are fixed depends only on the parameters. Thinking about regression the easiest thing to do is to find the weights for which the output of the network is as close as possible to the target values.

The problem with this is that given that the network is a non-linear function the **error function will not be a linear function in its parameters**: it is **quadratic in the error but non-linear in the parameters**. The example saw with a convex function happens if the model is linear, but is even useless to use gradient descent in those situations since the minimum can be found with closed-form solutions. Applying the gradient descent approach to the non-linear error function:

- Initialize the weights to a random value w^0 (e.g. to zero)

- We iterate starting from the initial random configuration:

$$w^{(k+1)} = w^{(k)} - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w^{(k)}}$$

i.e. the **movement** is always opposite to the direction of the gradient since it says the **direction of the maximum increase of the function so to minimize we have to go in the opposite direction**. The $-\eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w^{(k)}}$ says **how much and in which direction I have to move the weights**: the length of the movement is decided by both the learning rate η and the magnitude of the gradient. Iterating the procedure may pass the minimum but eventually it will converge to it: **having a too big η may cause an oscillation around the minimum avoiding the algorithm to converge to the minimum**; **but if η is small enough you will converge to the minimum**. So if the valley is very steep and you see the error oscillating the solution is to reduce η so that the algorithm can converge to the minimum. One thing done is to change η during the process: **usually once you reach the area around the minimum is possible to reduce η in order to improve the precision**.

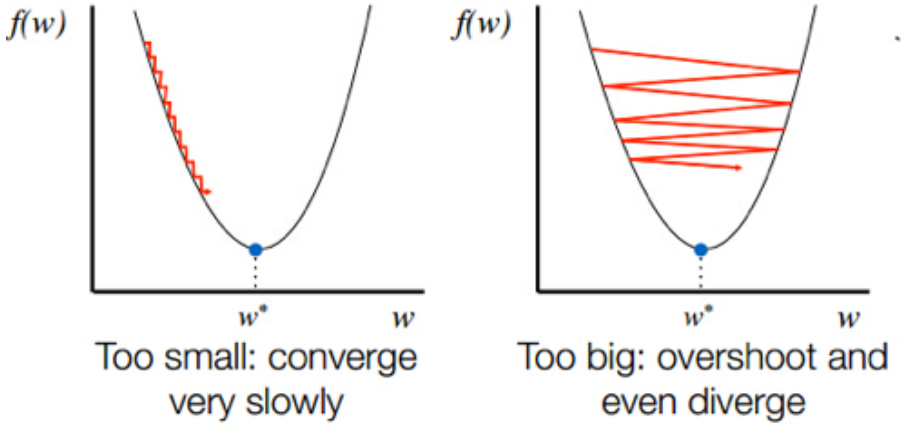


Figure 3: learning rate: too small vs too big

A small η will slow down the gradient descent, this must be taken into account but that is not the main issue of gradient descent indeed the **bigger problem** is that the **result depends on the starting point: depending on the starting point** we can obtain a really worse solution i.e. the gradient descent does **not find always the absolute minimum but the nearest minimum (local or absolute) in the opposite direction of the gradient**. To **avoid** the possibility to **converge** into **some types of local minima** we

can use gradient descent **with momentum**:

$$w^{(k+1)} = w^{(k)} - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w^{(k)}} - \alpha \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w^{(k-1)}}$$

Figure 4: gd with momentum

> **It is one of the possible improvement on gradient descent**, that **let you to pass over small bumps in the error function** and is **less prone to the noise in the error function letting you to find better local mimima**. But is **not still guaranteed to converge in the global minimum** since the solution **still depends on the initial values assigned to the parameters**: gradient descent with momentum **let only to find possibly the best local minima but it may not be the global one**. Since we have a greedy algorithm which has a solution depending on the starting point, we **might use multiple restarts to seek for a proper global minimum instead of a local minimum**, but since is not guaranteed to find what we **usually look for is a good local minimum**. Obviously the number of restarts is measured on the length of the training and the time available.

Furthermore, **thinking about overfitting the global minimum is the solution passing from all the points, so in principle the very minimum of the error function with a very flexible model is a very dangerous solution**.

NOTE Exploitation is gradient descent instead exploration are random movements: doing gradient descent not using all the data, gradient descent becomes very noisy so it adds some exploration by itself.

NOTE The reason why we don't use closed form solution is why they don't exists so we have to resort to gradient descent. With linear models is not a good idea to use gradient descent because: is way slower than finding closed form solution, and

the solution is unique so each method will converge to the same solution. Having multiple solutions without a closed form solution the only way to proceed is through an iterative algorithm and in this case gradient descent is the more effective solution in terms of speed.

3.2 Example of gradient descent and Backpropagation

Assuming the error function is the quadratic error and we want to compute the least square approximation.
Backpropagation is nothing but gradient descent: the formulas we will obtain by **applying gradient descent to all the weights of the network are backpropagation formulas**. So the **problems of backpropagation are the problems of gradient descent**:

- **may be slow to converge**
- it **might find local minima**
- you **should start from different initial points**
- the gradient should be defined, i.e. **activation function must be differentiable**.

We have the analytical expression of the output g, that we derive **to do backpropagation**, but it **doesn't mean you understand what it is doing** and what is the meaning, the semantic of what is doing.
By doing this example we will learn how backpropagation works in practice and how is implemented in the libraries, why works so well on gpu and highly parallel architectures.

Let's compute the update formula on the weight w_{ji}^1 of a network with I inputs, J hidden neurons and only 1 output. Obviously the procedure we will see works on multiple outputs and on multiple layers, things becomes only more complex.

ATTENTION Considering the bias as a weight over an input fixed to 1, the input layer has $I+1$ inputs so the summation goes from 0 to I ($\sum_{i=0}^{I}$) and the same for the other layers weights.

The number of weights to update in the connection between input layer and hidden layer are $(I+1)\cdot J$ since we must consider the weight related to the bias; similarly the number of weights to update in the connection between hidden layer and output layer are $(J+1)\cdot 1$ since we must consider the weight related to the bias; finally the total number of weights of the network is $[(I+1)\cdot J]+[(J+1)\cdot 1]$. So it will be very easy to obtain a network with millions of parameters if you don't restrict yourself to small networks. The **weights between two layers can be represented in a matrix** W where each **column** i **represent the weights connected to the** $i-th$ **input** and each **row represent the weights connected to the** $j-th$ **neuron of**

the next layer (W_{ji}). We will consider $j = 3$ and $i = 5$ so w_{35} (remembering that j is the destination and i is the source; this is only a notation and can be switched, if confusing, to w_{ij} still with i source and j destination).
To represent the fact that the **hidden layer's neurons activation function can be different from the one of the output neurons** let's call the activation function of the hidden neurons h and the one of the output neurons g. The notation we will use is $w_{ds}^{(l)}$ where: **d is the destination neuron, s is the source neuron and l is the index of the interface between layer of neurons being $l = 1$ for the connections between input and hidden layer and $l = 2$ for the connections between hidden and output layer**: the one we see is the most difficult but hides more properties. So the output of the network can be expressed as:

$$y = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_i\right)\right)$$

and the error can be expressed as:

$$E = \sum_n (t_n - y_n)^2$$

Now that we have the problem let's start doing the math:

$$w_{ds}^{(l)} = w_{ds}^{(l)} - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w_{ds}^{(l)}}$$

so I need to compute:

$$\frac{\partial E}{\partial w_{35}^{(1)}}$$

the formula we are computing works for all the connection for $l = 1$ and there will be another formula to apply for $l = 2$, where n is the index of the data sample used during the training:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = \frac{\partial \sum_n (t_n - y_n)^2}{\partial w_{35}^{(1)}} = \sum_n \frac{\partial (t_n - y_n)^2}{\partial w_{35}^{(1)}} =$$

$$= \sum_n 2 \cdot (t_n - y_n) \frac{\partial (t_n - y_n)}{\partial w_{35}^{(1)}} = 2 \cdot \sum_n (t_n - y_n) \frac{\partial (t_n - y_n)}{\partial w_{35}^{(1)}} =$$

NOTE The sum over n imply that we are using all the data points (batch) for training.

but since t_n **is a constant value**:

$$= -2 \cdot \sum_n (t_n - y_n) \frac{\partial y_n}{\partial w_{35}^{(1)}}$$

To compute this derivative we need to compute:

$$\frac{\partial y_n}{\partial w_{35}^{(1)}} = \frac{\partial g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in}\right)\right)}{\partial w_{35}^{(1)}}$$

so:

$$\frac{\partial y_n}{\partial w_{35}^{(1)}} = g'\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in}\right)\right) \cdot \frac{\partial\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in}\right)\right)}{\partial w_{35}^{(1)}}$$

from wich we need to compute:

$$\frac{\partial\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in}\right)\right)}{\partial w_{35}^{(1)}} =$$

$$= \frac{\partial\left(w_{10}^{(2)} \cdot 1 + w_{11}^{(2)} \cdot h_1\left(\sum_i w_{1i}^{(1)} \cdot x_{in}\right) + w_{12}^{(2)} \cdot h_2\left(\sum_i w_{2i}^{(1)} \cdot x_{in}\right) + ... + w_{J1}^{(2)} \cdot h_J\left(\sum_i w_{Ji}^{(1)} \cdot x_{in}\right)\right)}{\partial w_{35}^{(1)}}$$

since the first term $w_{10}^{(2)} \cdot 1$ does not contain the variable $w_{35}^{(1)}$ the derivative of the first term term is 0; **since all the $w_{1i}^{(1)}$ in the second term summatory have as destination the neuron 1 they will not contain $w_{35}^{(1)}$, so the derivative of the second term is 0**; **again for all the other terms except the fourth one $w_{13}^{(2)} \cdot h_1\left(\sum_i w_{3i}^{(1)} \cdot x_{in}\right)$ contain the derivation variable** $w_{35}^{(1)}$ so we obtain:

$$= w_{13}^{(2)} \cdot \frac{\partial h_3\left(\sum w_{3i}^{(1)} \cdot x_{in}\right)}{\partial w_{35}^{(1)}}$$

from which we need to compute:

$$\frac{\partial h_3\left(\sum w_{3i}^{(1)} \cdot x_{in}\right)}{\partial w_{35}^{(1)}} = h_3'\left(\sum w_{3i}^{(1)} \cdot x_{in}\right) \cdot \frac{\partial\left(\sum w_{3i}^{(1)} \cdot x_{in}\right)}{\partial w_{35}^{(1)}}$$

from which we need to compute:

$$\frac{\partial\left(\sum w_{3i}^{(1)} \cdot x_{in}\right)}{\partial w_{35}^{(1)}} =$$

$$= \frac{\partial\left(w_{30}^{(1)} \cdot 1 + w_{31}^{(1)} \cdot x_{1n} + w_{32}^{(1)} \cdot x_{2n} + ... + w_{3I}^{(1)} \cdot x_{In}\right)}{\partial w_{35}^{(1)}} = x_{5n}$$

since as before the terms where the derivation variable does not appear have a derivative equal to zero.
So the final formula is:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = -2 \cdot \sum_n (t_n - y_n) \cdot g' \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right) \cdot w_{13}^{(1)} \cdot h_3' \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right) \cdot x_{5n}$$

This is the **backpropagation formula utilized to propagate the error on the output back to the weights of the network, more precisely to $w_{35}^{(1)}$ in this case**. From this formula we can do several observations:

- The step of the gradient takes into account all the data points since it contains a $\sum_n$ where n is the index of the sample data used in the training: what it does it gathers all the gradients with respect to all the data points data points, it sums (or averages) them all, and it takes one step in the average direction. It makes sense since if one data point pulls the model in one direction and another data point pulls in the opposite direction, doing it one sample at time it will probably oscillate; instead by **doing this batch update considering all the data we get a smoother optimization**.

- The **problem is that to compute this average move we need to process all the data**. So sometimes this update is performed on graphical process unit; the formula:

  $$w_{ds}^{(l+1)} = w_{ds}^{(l)} - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w_{ds}^{(l)}}$$

  it computes the update of any weights of the network based on the previous weights so you can update simultaneously all the weights of the network since at each iteration you have all the data to do it and you don't have to wait for other computation. The possibility of **parallelizing the computations allow it to be done on very efficient machines as GPU**: the problem depend on the fact that since the formula comprehend $x_i \quad 0 < i < I$ it means that the **data must be available in the machine (e.g. in the GPU memory)**. **So sometimes is not feasible to load all the data in the memory of the GPU to perform this update in a batch way**, but the **data point used to compute the error derivative must be reduced**.

Adding many layers will cause this computation to be repeated for each added layer, so probably there is some more better and smart way to do this.

3.3 Backpropagation Variations

There are several **variation of gradient descent algorithm** that **improve its performances**:

- **Batch gradient descent**: update the weights by considering the derivative of all the data:

$$\frac{\partial E(w)}{\partial w} = \frac{1}{N}\sum_{n}^{N}\frac{\partial E(x_n, w)}{\partial w}$$

- **Stochastic gradient descent (SGD)**: **use** a **single sample** so it is **unbiased, but with high variance**:

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{SGD}(w)}{\partial w} = \frac{\partial E(x_n, w)}{\partial w}$$

  So **like Hebbian learning** updates the weights one sample at a time and is **called stochastic since updating using the derivative of one data point each time will do a step in one direction causing to be very erratic**. But on average you get the same result: so the **estimate of the gradient is unbiased**, but the problem is that **has a very high variance**: on average you end up in the same point but it is **very noisy**.

- **Mini-batch gradient descent**: **use** a **subset of samples**, **good trade-off variance-computation** i.e. we **mix the two latter approaches**:

$$\frac{\partial E(w)}{\partial w} \approx \frac{\partial E_{MB}(w)}{\partial w} = \frac{1}{M}\cdot\sum_{n\in Mini-batch}^{M<N}\frac{\partial E(x_n, w)}{\partial w}$$

  Data are grouped in **mini-batches that contain as many samples as the GPU is capable of contain in its memory**, so the **batch size is a parameter that must be fine tuned on your computational capabilities**: the objective is to **find** the **best i.e the bigger batch size the GPU is capable of containing since bigger is the batch smoother will be the convergence and less noisy will be the error function during the gradient descent**. Mini-batch gradient descent is **still unbiased**, **because you still do one full epoch by going through the dataset in mini-batches**, and has **less variance than the SGD due to the fact that data is grouped into batches so the oscillation that may come from different data samples is reduced**.

NOTE $\frac{1}{N}$ in batch GD and $\frac{1}{M}$ in mini-batch GD is multiplied **to have the same magnitude in each gradient variation**; this is **not mandatory**: if not considered in the error gradient the scaling factor $\frac{1}{N}$ or $\frac{1}{M}$ **must be considered in the learning rate to obtain the same step in the weight update**.

It turns out that in some experiments SGD gets better results if the learning rate is fixed for all the gradient descent updates, instead if the learning rate is re-scaled so that more or less the size of the step is the same on average you get the same results. **So to have a less noisy process is suggested to use mini-batch** but we can try different solutions.

As seen in lab **one of the parameter to set is the batch size**. **Tensors** are data-structures similar to matrices with multiple dimension e.g. dataset of images [height x width x color channels x batch size] i.e. a matrix with four dimensions. We have to start to **think to matrices with multiples dimensions**, and **one of this is typically the batch size**: several algorithm is **parallelized on the batch size**:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = -2 \cdot \sum_{n \in Mini-batch}^{M<N} (t_n - y_n) \cdot g' \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right) \cdot w_{13}^{(1)} \cdot h_3' \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right) \cdot x_{5n}$$

the operation done on each data sample in the batch are the same and then they are summed up to compute the error derivative.

The **backpropagation be automatized** in a very simple way that **let us** to **compute it with respect to any weight in the network**. Analyzing the error derivative formula we obtained earlier:

- Since $E = \sum_n^N (t_n - y_n)^2$ the fist term by considering y_n the variable can be seen as:
$$-2 \cdot \sum_n^N (t_n - y_n) = \frac{\partial E}{\partial y_n}$$
- By considering the argument of g as a single variable y_n, the second term can be seen as:
$$g' \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right) = \frac{\partial y_n}{\partial \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right)}$$
since y_n is the outupt of the network i.e. $y_n = g(x_n, w)$.
- The third term can be seen as:
$$w_{13}^{(1)} = \frac{\partial \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right)}{\partial h_3 \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right)}$$
- The fourth term can be seen as:
$$h_3' \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right) = \frac{\partial h_3 \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)}{\partial \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)}$$
- The last term can be seen as:
$$x_{5n} = \frac{\partial \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)}{\partial w_{35}^{(1)}}$$

From all the latter the observation we obtain:

$$\frac{\partial E}{\partial w_{35}^{(1)}} = \frac{\partial E}{\partial y_n} \cdot \frac{\partial y_n}{\partial \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right)} \cdot \frac{\partial \left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right) \right)}{\partial h_3 \left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{in} \right)} \cdot \frac{\partial h_3 \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)}{\partial \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)} \cdot$$

so we can **build our derivative by multiplying partial derivatives**, in which is possible to **see a pattern**: **each term as at the denominator the numerator of the next term** i.e. **is the derivative of composed functions**.
For the latter pattern **weights update can be done in parallel, locally**, and **requires just two passes**:

- Let x be a real number and two functions $f : \Re \to \Re$ and $g : \Re \to \Re$
- Consider the composed function $z = f(g(x)) = f(y)$ where $y = g(x)$
- The derivative of f w.r.t. x can be computed applying the **chain rule**:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y) \cdot g'(x) = f'(g(x)) \cdot g'(x)$$

This is formalizing what we have seen as an empirical result after the latter substitutions in the backpropagation formula, so the **same holds in backpropagation**. But **this does not solve the automation problem** since **we only know** that the **derivative** of the backpropagation can be **composed by splitting it in multiple pieces**, but **does not solve the problem of how do I extract this pieces**. To extract this pieces we **look how the FFNN is built**: by doing a **forward pass** into the network **you can compute all the pieces you need to compute the error derivative starting from the last one** ($\frac{\partial \left(\sum_i w_{3i}^{(1)} \cdot x_{in} \right)}{\partial w_{35}^{(1)}}$) to the first one ($\frac{\partial E}{\partial y_n}$). Each time the derivative con be computed, during the forward pass, is **computed and stored**.

NOTE We can notice that the formula require $w_{ji}^{(1)}$ and $w_{1j}^{(2)}$ respectively for the last and the third term: they are available from the previous iteration, since **the update process is iterative that only need the input and the previous weights**.

Once all the pieces to compute the error derivative are computed and stored, a **backward pass** is **done to update all the weights** using:

$$w_{ds}^{(l+1)} = w_{ds}^{(l)} - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w=w_{ds}^{(l)}}$$

using one of the backpropagation variants seen (batch, SGD, MB).
During the forward pass we store, in order:

- x_i
- h_j, h'_j

- g, g'
- $error$

Obviously the weights must be stored since are the parameters of the network, and at each update iteration are used the weights of the previous iteration. These **quantities we store can be associated to a part of the neural network**:

- x_i: is **associated to the connection between the input and the hidden layer**.
- h_j: is **associated to the neuron of the hidden layer**.
- h'_j: is **associated to the connection between the hidden layer and the output layer**.
- g, g': is a**ssociated to the neuron of the output layer**.
- $error$: is **associated to the output**.

so by considering this associations becomes easy to find out which terms must be considered to compute $\frac{\partial E(w)}{\partial w}\Big|_{w=w_{ds}^{(l)}}$: the **only things to do is to do a backward pass starting from the output considering the terms encountered during the way, stopping at the connection of the weight we are considering in the update**. So the **backward pass** is **useful since you do not require to do all the computation done at the beginning**, but it has also **another advantage**: **changes in the model does not require more computations** (i.e. does not add any difficulty) but **only little changes in what we store**: replacing, and changing the activation function of the hidden layer require only to change h, easily allowing having mixed activation function since the only change to do is in the h and h' stored; replacing the error function only need to change the error computation at the output; and so on. **Also adding more layers does not add any difficulty, but** in that case things are slightly different since **a weight can be reached form multiple path and this can be easily take into account to sum them all**. Backward pass help also with shortcut connections since the only things to consider when we want to update a weight is the fact that it can be reached from multiple connection.

Hence, **backward pass works well with all sort of complex network**: provided that they are feed forward neural networks then computing weights through backpropagation is straight forward. So everything we have seen with the simple network can be easily done on complex network by simply using backpropagation.

NOTE In ML frameworks they give you layers and for each layer they implement the forward pass giving the output and the backward pass giving the related stored quantities.

Is clear how **backpropagation** is as said **highly parallelizable**: **GPU really speed up the process**.

3.4 Error function design

We have seen many expects of **gradient descent** and how is **simple and automatable** to **compute the error gradient**, based on the network topology using backpropagation.
Until now we have computed the network's output error as sum of squared errors $\sum_n (t_n - g(x_n, w))^2$ (remember that the use of $\frac{1}{N}$ does not change the meaning of the error function since the scaling will be considered in the learning rate). **Why should we use it? Is this the right error function to use?**

3.4.1 Maximum likelihood estimation

Maximum likelihood estimation is the e**stimation of some unknown parameters of the distribution based on the observation on some data**, i.e. a way to **find the right distribution that fit the sample data**.
Let's observe independent and identical distributed (i.i.d.) samples x_n from a Gaussian distribution (i.e. distributed identically inside a gaussian distribution) with known σ^2 (e.g. voltage measured by a sensor: it can be negative):

$$x_1, x_2, ..., x_n \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

One criterion to decide if one distribution fit the data is to look at the data distribution and concentration and ask ourselves how likely is one distribution to be the one that fit the data. So the **likelihood principle is to select the parameter of the distribution by determine how likely is the distribution in line with the observations** i.e. **maximum likelihood choose parameters which maximize data probability, makes the most of the points likely to be observed**.

NOTE Maximum Likelihood select the parameters that maximize the data probability.

So is very easy to write a recipe to find any parameters for any parameter based model:

- Let $\theta = (\theta_1, \theta_2, ..., \theta_p)^T$ a vector of parameters, find the MLE for θ
- Write the likelihood $L = P(Data, \theta)$ for the data
- OPTIONAL: Take the logarithm of the likelihood $l = log(P(Data|\theta))$. Sometimes is useful to compute the logarithm of the likelihood function since the likelihood function is the product of many terms, instead applying the logarithm we obtain a sum over the logarithm of those terms: this helps a lot when we want to find the maximum of the likelihood function since **without using the logarithm we would have to compute the derivative of a lot** (the same number as the number of data samples) **of multiplied terms**; furthermore **the multiplication of a lot of numbers smaller than one** (they

are probabilities) **may cause the underflow of the precision of our computer**. This can be done since the **logarithm is a monotonic function, so the maximum of the logarithm of the function is in the same place of the maximum of the function**.

- Work out $\frac{\partial L}{\partial \theta}$ or $\frac{\partial l}{\partial \theta}$
- To **find the stationary point in which we find the maximum** solve the set of simultaneous equations $\frac{\partial L}{\partial \theta} = 0$ or $\frac{\partial l}{\partial \theta} = 0$
- **Check** that θ^{MLE} **is a maximum** (usually is not a problem)

To maximize/minimize the (log-)likelihood you can use:

- **Analytical Techniques** (i.e. solve the equations)
- **Optimization Techniques** (e.g. Lagrange multipliers)
- **Numerical Techniques** (e.g. gradient descent)

We know already about gradient descent, let's try to use some analytical solutions. Let's observe independent and identical distributed samples x_n from a Gaussian distribution with known σ^2 (e.g. voltage measured by a sensor: it can be negative):

$$x_1, x_2, ..., x_n \sim N(\mu, \sigma^2) \qquad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

and **find the maximum likelihood estimator for** μ (only μ because we **already know** σ^2):

- Let μ **the parameter we look for**, find the MLE for μ.
- Write the likelihood $L = P(Data, \theta)$ for the data:

$$L(\mu) = p(x_1, x_2, ..., x_N|\mu, \sigma^2) = \prod_{n=1}^{N} p(x_n|\mu, \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}$$

  since the x_n **are independent and identical distributed (i.i.d.) we can say that the total probability is the product of the probability on the single sample** and since the distribution we are analyzing is the Gaussian one we know how the probability is distributed.

- Take the logarithm of the likelihood $l = log(P(Data|\theta))$ of the likelihood.

$$l(\mu) = log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) = \sum_{n=1}^{N} log\left(\frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) =$$

$$N \cdot log\left(\frac{1}{\sqrt{2\pi} \cdot \sigma}\right) - \frac{1}{2 \cdot \sigma^2} \sum_{n}^{N} (x_n - \mu)^2$$

- Work out $\frac{\partial l(\mu)}{\partial \mu}$:

$$\frac{\partial l(\mu)}{\partial \mu} = \frac{\partial \left(N \cdot log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma}\right) - \frac{1}{2\cdot\sigma^2}\sum_n^N (x_n-\mu)^2 \right)}{\partial \mu}$$

$$-\frac{1}{2\cdot\sigma^2} \cdot \frac{\partial \sum_n^N (x_n-\mu)^2}{\partial \mu} = \frac{1}{2\cdot\sigma^2} \cdot \sum_n^N 2\cdot(x_n-\mu)$$

- To find the stationary point in which we find the maximum solve the set of simultaneous equations $\frac{\partial l}{\partial \mu} = 0$:

$$\frac{1}{2\cdot\sigma^2} \cdot \sum_n^N 2\cdot(x_n-\mu) = 0$$

$$\Rightarrow \sum_n^N 2\cdot(x_n-\mu) = 0$$

$$\Rightarrow \sum_n^N 2x_n = \sum_n^N 2\mu$$

$$\Rightarrow \mu^{MLE} = \frac{1}{N}\sum_n^N x_n$$

that is the maximum likelihood extimation of the mean of the gaussian distribution on our sample data, knowing σ^2

3.5 Error function for Regression

In neural networks the goal is to approximate a target function t having N observation (target values associated to the N input values):

$$t_n = g(x_n|w) + \epsilon_n \qquad \epsilon_n \sim N(0,\sigma^2)$$

i.e. we assume that **on top of the observation coming from our model $g(x_n|w)$, considering we have set the right parameters (deterministic part) we must add a noise ϵ_n (stochastic part)**, assuming that the noise comes from a gaussian distribution, that **represent the error between the target and the output of the network**. Is a reasonably assumption since the FFNN is a universal approximator so it can representing any possible non linear function, even the true one. By this **assumption**:

$$\Rightarrow t_n \sim N(g(x_n|w),\sigma^2)$$

i.e. **the observation are basically distributed as a normal function with mean value equal to the output of the network and variance equal to the variance of the error**. This can be easily seen by the fact that $g(x_n|w)$ is a value and ϵ_n

is a distribution, so by adding $g(x_n|w)$ to ϵ_n the mean value of the distribution represented by ϵ_n is shifted. **ATTENTION THE INPUT IS NOT GAUSSIAN BUT CONSTANT VALUES i.e. deterministic**: in the TV-sales graph they are on the x axis and the $g(x_n|w)$ is on the y axis. This is a **typical statistical learning framework: your data comes from a deterministic function g plus some noise ϵ_n.**

The **estimate of the mean** of this distribution **is the output of the neural network** that depends on the set of weights of the network, **so the mean is parametrized by the weights of the network (since each other thing is fixed, even the function g)**:

$$t_n \sim N(g(x_n|w), \sigma^2) \quad p(t|g(x_n|w), \sigma^2) = \frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(t-g(x_n|w))^2}{2\sigma^2}}$$

Hence **solving the maximum likelihood estimation on this distribution means to find the value of the weights such that the observation probability is maximized**, since they are the only parameter we have:

- **Let w the parameter we look for, find the MLE for w.**
- Write the likelihood $L = P(Data, \theta)$ for the data:

  $$L(w) = p(t_1, t_2, ..., t_N|g(x_n|w), \sigma^2) = \prod_{n=1}^{N} p(t_n|g(x_n|w), \sigma^2) = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}}$$

  since the distribution we are analyzing is the gaussian one.

- Take the logarithm of the likelihood $l = log(P(Data|\theta))$ of the likelihood.

  $$l(\mu) = log\left(\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) = \sum_{n=1}^{N} log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(x-\mu)^2}{2\sigma^2}}\right) =$$

  $$N \cdot log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma}\right) - \frac{1}{2\cdot\sigma^2}\sum_{n}^{N}(x_n - \mu)^2$$

- Look for the weights which maximize the likelihood:

  $$argmax_w L(w) = argmax_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}} =$$

  $$= argmax_w \sum_{n}^{N} log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{(t_n-g(x_n|w))^2}{2\sigma^2}}\right) =$$

  $$= argmax_w \sum_{n}^{N} log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma}\right) - \frac{1}{2\cdot\sigma^2}\cdot(t_n - g(x_n|w))^2$$

$$= argmin_w \sum_n^N (t_n - g(x_n|w))^2$$

where $log\left(\frac{1}{\sqrt{2\pi}\cdot\sigma}\right)$ does not depend on w so it can be eliminated since we are looking to the mixiumum over w (so it is a constant term that does not affect the position of the maximum) and $argmax$ **becomes** $argmin$ **due to the multiplicative minus**.

So we found out that **if we assume that the observation are the observed data** $g(x_n|w)$ **coming from our network plus some noise** $\sim N(0, \sigma^2)$ **and a known** σ^2 then the **maximum likelihood estimation of the network weights is obtained by**

$$argmin_w \sum_n^N (t_n - g(x_n|w))^2$$

i.e. the least square error. Hence every time the assumptions are respected we can compute the weights in this way. But what we found out has a **deeper meaning**, indeed we **demonstrated that when this assumptions are verified the least square error can be used as the error function to learn the weights**.

NOTE the **assumption of knowing** σ^2 **is not really needed since the value of the weights in least square error does not depend on** σ, so the **only thing that must be verified on** σ **is that it is constant and does not depend on the input** x **and on the weights, but we do not have to necessarily know its value** (i.e. i.i.d samples assumption).

NOTE is not a naive bayes approach because we assume the observation to be independent and identically distributed not the features to be independent: there aren't any assumption on the input but only on the output; in the naive bayes approach the assumption is that all the input (features) are independent given the output.

NOTE t_n and x_n are iterating over the same index n since the index is the number of the observation: a certain input comes from a certain output. Don't get misleaded, x_n **is a vector of l component**, not a component.

NOTE the assumption the variance is constant is not the correct assumption under which the solution is optimal: when we have a variance that changes with the mean g as in the TV-sales picture (error is smaller for small g and higher for high g) we should find a way to remove this correlation. One way to do this is to transform the output variable in order to make as much as possible the variance indipendent from the variable: for example instead of learning to predict t you learn to predict $log(t)$ or t^2:

$$argmin_w \sum_n^N (log(t_n) - g(x_n|w))^2$$

indeed if the error is linearly depending on the mean this help fixing the things. In general there are some transformation that pre-process the output helping into make the assumption of constant σ to be true, but to do this we have to know the data t we are dealing with.

So **using the least square error** as we have done so far with the FFNN **is the right thing if**:

- We have **a regression problem** i.e. our output is a continuous function
- We can **assume the error to be gaussian**
- The **variance is constant**

but **not all the problems are regressions and not all the problems have a gaussian noise**. Is possible to set other error functions which try to make the maximum likelihood estimation less sensitive to the violation of the gaussian error assumption.

NOTE To change error function if you can write the distribution of the error you can derive the error function.

3.5.1 Error function for classification

When we shouldn't use least square error? For classification for example. There is another case where the **gaussian assumption doens't work**: classification. You **cannot think classification in terms of real value output plus some gaussian** noise so you have to model the distribution represented by the neural network in a different way.

Let's assume we are working on classification (not more on regression). The goal is to **approximate a posterior probability** t **having N observations** $t_1, t_2, ..., t_N$ where:

$$g(x_n|w) = p(t_n|x_n) \quad t_n \in \{0, 1\}$$

Thinking the output of the network as a random variable with 2 possible outcomes the first thing that comes to mind is that the **output of this network comes from a Bernoulli distribution with a certain parameter** p **which will tell you how likely is to get the one and how likely is to get the zero**:

$$\Rightarrow t_n \sim Be(g(x_n|w)) = Be(p(t_n|x_n))$$

so if you think about how to **represent a result of your model you don't think anymore in terms of deterministic function plus some gaussian noise but in terms of a variable that is either 0 or 1 and the probability of getting 1 depends on the class of the object** so the output can be represented as the probability of the output being one given the input x_n i.e. **the output of the network can be interpreted in term of probability**.

Since we have re-coded the **output** of the network in terms of a **random variable** we can **estimate the parameter using the maximum likelihood estimation**:

- We have some independent and identically distributed samples t_n coming from a **Bernoulli distribution**:

  $$t_n \sim Be(g(x_n|w)) \qquad p(t|g(x|w)) = g(x|w)^t \cdot (1 - g(x|w))^{1-t}$$

  where the probability is computed considering the probability that the output is one if the target is one $g(x|w)^t$ or it is the probability that the output is zero if you observe zero $(1 - g(x|w))^{1-t}$

- The parameter describing the Bernoulli is what we want to learn, so let w the parameter we look for, **find the MLE for** w.

- Write the likelihood $L = P(Data, \theta)$ for the data:

  $$L(w) = p(t_1, t_2, ..., t_N|g(x_n|w)) = \prod_{n=1}^{N} p(t_n|g(x_n|w)) = \prod_{n=1}^{N} g(x_n|w)^{t_n} \cdot 1 - (1 - g(x|w))^{1-t_n}$$

- Look for the weights which maximize the likelihood:

  $$argmax_w L(w) = argmax_w \prod_{n=1}^{N} = \prod_{n=1}^{N} g(x_n|w)^{t_n} \cdot (1 - g(x_n|w))^{1-t_n} =$$

  $$= argmax_w log \left(\prod_{n=1}^{N} g(x_n|w)^{t_n} \cdot (1 - g(x_n|w))^{1-t_n} \right) =$$

  $$= argmax_w \sum_{n}^{N} t_n \cdot log(g(x_n|w)) + (1 - t_n) \cdot log(1 - g(x_n|w)) =$$

  $$= argmin_w - \sum_{n}^{N} t_n \cdot log(g(x_n|w)) + (1 - t_n) \cdot log(1 - g(x_n|w))$$

  so if the observation t_n is one we consider the first term, otherwise if the observation is zero we consider the second term.

So the **error function in case of classification** is not anymore the sum of squared errors but is the so called **crossentropy**, usually written as:

$$- \sum_{n}^{N} t_n^T log(g(x_n|w))$$

The formula we found is the binary crossentropy that works on networks with only one output neuron; the **latter instead works even with a multi neuron output layer** (i.e. a classification over k classes) **with the target one-hot encoded** and for this reason is called **multi-class crossentropy** where: t_n is a one-hot vector and $g(x_n|w)$ is the output vector (i.e. the difference is that we go from two to more classes).

NOTE Notice the fact that in the binary cross entropy we have a single target which is either 0 or 1 so we have to represent both the cases for $t = 0$ and $t = 1$; instead in the multi class crossentropy the target is a one-hot vector so each target has a component different from 0 that produce the value $log(g(x_n|w))$ being $t_n^{(i)} = 1$.

For example let's assume we have a network output $[g_1, g_2, g_3]^T$ and the target output $t = [0, 0, 1]^T, [0, 1, 0]^T, [1, 0, 0]^T$ and the crossentropy:

$$-\sum_n t_n^T log(g) = -\sum_n t_n^T log([g_1, g_2, g_3]) = -[0, 0, 1]log([g_1, g_2, g_3]) - [0, 1, 0]log([g_1, g_2, g_3]) - [1, 0, 0]log([g_1, g_2, g_3$$

But what if g_i is equal to 0? What happens to the logarithm? This is a very mathematical question but from a practical point of view does not happen because the output of a classification network comes from a sigmoid so the only case of it being 0 is for $-\infty$ which means never, because is bounded by the minimum flow of the computer memory (may happen if one of the input of the output layer goes to $-\infty$, but then the problem is not from cross entropy but the fact that a variable goes to $-\infty$ and that should never happen). g is already normalized with softmax, thats why if the input is $-\infty$ the output goes to 0. g being 0 can be a problem with tanh as output activation function, but tanh is not used for multi-class classification since we use the softmax: **using tanh you cannot interpret the output as probability so you cannot apply the reasoning we did and crossentropy cannot be used anymore**. So **if we want to use crossentropy error we should't use tanh, but sigmoid for binary classification and softmax for multi-class classification**.

NOTE the cross entropy is multiplied for -1 since we want to write as a minimization problem since it is the typical setting.

NOTE we don't need to remember all this formulas but we have to know: how to derive them, the hypothesis behind regression that lead to square error and the hypothesis behind classification that lead to crossentropy.

So the **different error functions for classification and Regression come from the fact that the assumption we do on the distribution of the observations t_n are different but in both cases we apply the maximum likelihood estimation**.

NOTE **using logarithmic max likelihood estimation obviously the maximum has a different value but we don't care about it, since we are only interested in the value of the weights for which the function has a maximum value and these values do not change passing to the logarithm**.

NOTE $p(t|g(x|w)))$ is the **a posteriori probability of the class given your input**: the classifier can be learnt by learning the a posteriori probability of class given your input.

The error function are those functions related to finding the weights which minimize them, so gradient descent is used to minimize these error functions. **Depending on the problem we are facing you should use different error functions**. Hence **MLE is one approach to design error function and gradient descent is the approach to minimize them**.

We saw where the squared error function we used comes from, that is the regression problem with gaussian error. In case of classification according to the same principle i.e. estimating parameters for which data are more likely we should use crossentropy.

3.5.2 Hebbian learning and perceptron special case

The perceptron and hebbian learning can be seen as a special case of backpropagation on a special error function. Let's see under which assumption the hebbian learning can be interpret in terms of gradient descent and MLE (or at least error function minimization)

NOTE Perceptron and Hebbian learning was invented before the use of gradient descent and the use of global error fucntion in the neural network: this is just a way to look at Hebbian learning in a different perspective.

To understand how the perceptron rule relates to this we need to look at different error functions. So far we have **observed different error functions**:

- **Sum of squared errors** that should be used with **Regression**
- **Binary crossentropy** that should be used for **binary classification**.

Error functions define the task to be solved, but **how to design them?**

- **Use all your knowledge/assumptions about the data distribution**: making some assumption for modeling the error of this models and **derive the error function by using Maximum Likelihood Estimation**.
- **Exploit background knowledge on the task and the model** (e.g. in images segmentation i.e. distinguish pixel belonging to different objects: write an error function that takes into account that the segmentation wasn't done correctly)
- Use **creativity**: design of errors functions requires lots of trial and. errors

For perceptron we will exploit background knowledge on the task and the model.
Remembering how a perceptron is built: computes the class as 1 or -1 according to the sign (Heaviside function) of the weighted sum input. **Since the outputs are 1 or -1 (as for tanh) we should not use crossentropy, because +1 and -1 do**

not represent a probability distribution so we have to look to how to design an error function for the perceptron. To do that we have to look inside the algebra behind the perceptron. The perceptron to distinguish between the classes **creates the hyperplane (affine set)** $L \in \Re^2$ i.e. the set of points in the plane for which:

$$L : w_0 + w^T \cdot x = 0$$

For any two points x_1 and x_2 on $L \in \Re$ we have that:

$$w_0 + w^T \cdot x_1 = 0 \qquad w_0 + w^T \cdot x_2 = 0$$

$$\Rightarrow w^T \cdot (x_1 - x_2) = 0$$

with the vector $(x_1 - x_2)$ on the hyperplane since both x_1 and x_2 belongs to the vector and since the latter product is equal to zero the vector w is orthogonal to $(x_1 - x_2)$ i.e. **orthogonal to the line**. So the **versor normal to** $L \in \Re^2$ is then:

$$\hat{w} = \frac{w}{\|w\|}$$

Hence this is **another way to interpret the parameters of the perceptron i.e. a versor orthogonal to the line**. For any poin x_0 in $L \in \Re^2$ we have:

$$w^t \cdot x_0 = -w_0$$

Hence the **signed distance from the hyperplane** $L \in \Re^2$ **of any point** $x \in \Re^2$ is **defined** by:

$$\hat{w}^T \cdot (x - x_0)$$

i.e. the component of the vector $x - x_0$ **along the orthogonal to the hyperplane passing by** x_0, that from the upper equations can be computed as:

$$\hat{w}^T \cdot (x - x_0) = \frac{1}{\|w\|} \cdot (w^T \cdot x + w_0)$$

i.e. $(w^T \cdot x + w_0)$ **is proportional to the distance of** x **from the plane defined by** $w^T \cdot x_0 + w_0 = 0$. Now the question is: **how can we find the best separating hyperplane such that the points classified as +1 are above the hyperplane and the one classified as -1 below the hyperplane?** We can use the property of the hyperplane we found, i.e. the distance from the plane can be computed as:

$$\frac{1}{\|w\|} \cdot (w^T \cdot x + w_0)$$

that gets closer to 0 as it is closer to the hyperplane that is indeed $(w^T \cdot x + w_0) = 0$. But **we do not only have the distance, we have the signed distance indeed the sign is given by the verse of the versor** $\hat{w}$: so if a point is above the hyperplane the result is positive, viceversa if the point is below the hyperplane the result is negative.

Now **let's exploit this property to define the error function** for our perceptron whose output is $+1$ or -1 respectively if the input point is above or below the hyperplane described by the weights: hence **what the perceptron does is computing the sign of the distance of a point from the separating boundary**. It can be shown that the **error function** the Hebbian learning **minimize is the distance of miss-classified points from the decision boundary**. Let's code the perceptron output as $+1$ and -1, so we have an error when a point is on the other side of the line i.e when there is a miss-classification:

- If an output which should be $+1$ is miss-classified is an error and is classified as -1 i.e. $w^T \cdot x + w_0 < 0$
- For an output with -1 we have the opposite i.e is classified as $+1$ i.e. $w^T \cdot x + w_0 > 0$

The **goal becomes minimizing the sum of this errors**, so we define the following error function:

$$D(w, w_0) = -\sum_{i \in M} t_i \cdot (w^T \cdot x_i + w_0)$$

where M **is the set of points miss-classified** i.e the **error function is the total distance of the miss-classified points**. Each term of the summation is negative:

- If t_i is $+1$ and the point is miss-classified then $w^T \cdot x_i + w_0 < 0$ so their multiplication is negative.
- If t_i is -1 and the point is miss-classified then $w^T \cdot x_i + w_0 > 0$ so their multiplication is negative.

but the **multiplication for -1 of the whole summation turns all the terms into non negative**. Furthermore as we have said $w^T \cdot x_i + w_0$ are proportional to the distance of the miss-classified points from $w^T \cdot x + w_0 = 0$ so **the summation of all these non negative terms is proportional to the total distance from the hyperplane of the miss-classified points**. Hence smaller the total distance of wrong points less are the wrong points, thats why we want to minimize. This new error function is different from the ones we have seen before indeed it is **not based on the assumption of the probability distribution of the data but this cost function is built on geometric reasoning**. To minimize the error function just defined we can apply SGD (or batch gradient descent). So the first thing to do is to compute the gradients with respect to the model parameters:

$$\frac{\partial D(w, w_0)}{\partial w} = -\sum_{i \in M} t_i \cdot x_i$$

$$\frac{\partial D(w, w_0)}{\partial w_0} = -\sum_{i \in M} t_i$$

Then SGD applies the update function for each miss-classified point (indeed Hebbian Learning applies SGD):

$$\begin{pmatrix} w^{(k+1)} \\ w_0^{(k+1)} \end{pmatrix} = \begin{pmatrix} w^{(k)} \\ w_0^{(k)} \end{pmatrix} + \eta \begin{pmatrix} \frac{\partial D(w,w_0)}{\partial w} \\ \frac{\partial D(w,w_0)}{\partial w_0} \end{pmatrix} =$$

$$= \begin{pmatrix} w^{(k)} \\ w_0^{(k)} \end{pmatrix} + \eta \begin{pmatrix} t_i \cdot x_i \\ t_i \end{pmatrix} =$$

that, remembering that w_0 is the bias weight and that the input x_0 is equal to 1, can be written as:

$$\begin{pmatrix} w^{(k+1)} \\ w_0^{(k+1)} \end{pmatrix} = \begin{pmatrix} w^{(k)} \\ w_0^{(k)} \end{pmatrix} + \eta \begin{pmatrix} t_i \cdot x_i \\ t_i \cdot x_0 \end{pmatrix}$$

What we can **observe in this function is that it is the same formula as Hebbian learning, so Hebbian learning implements Stocastic Gradient Descent on the error function of the total distance of miss-classified points** we defined from the geometric interpretation of the perceptron model. At the beginning they didn't know this, they took inspiration from the real neurons and the synaptic weights strenghtening but we saw it is equivalent of SGD on the error function of the total distance of miss-classified points.
This reasonment is very specific to the perceptron since is derived from its model's geometrical characteristics. But again we can **think about learning in FFNN as the fact of using gradient descent on a error function**: the **difficult** part is not to apply the gradient descent but to **design the error function** and this **can be done, as said previously, using some assumption on the distribution of data and then apply MLE or** as was done in the case of the perceptron **using the knowledge on the model itself**.

Some tasks might not be easily described by an error function and sometimes an error function might not be differentiable which is a problem for the backpropagation algorithm (e.g. segmentation).

!! FINISCI ULTIMI 25 MINUTI VIDEO 6 !!

3.6 Neural Networks training and overfitting

Until now we have understood what are neural networks and the theory behind them. Now we will see some **practical issue that comes with training**.

The first and one of the **most important issues** of the neural networks and **comes from the universal approximation theorem**:

Universal Approximation Theorem. *A single hidden layer feed-forward neural network with S (sigmoid, tanh) shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set*

So regardless of what function we are learning, a single layer can do it, but:

- We **need a learning procedure to find the weights** (e.g. backpropagation), but it **doesn't mean we can find the necessary weights i.e. the algorithm not always converge to a solution**.
- Is true that is a universal approximator but it might happen that an **exponential number of hidden units may be required** since the function we want to approximate is very highly non-linear, so the **model** becomes **unnecessarily complex**.
- An **unnecessarily complex model** might be **useless in practice if it does not generalize** i.e. we have the so called **overfitting** problem.

So a model is said to be **overfitted** if the training of the model imply **very high performance (low error) in the training set** and **very bad error on new data** i.e. the model is **not able to generalize i.e. there is a big difference between training error and test error**. Deep learning is a machine learning framework and as all the machine learning framework is aiming at developing **learning models which generalize**, hence it means it provides good results on new data. As soon as the model becomes overly complex you end up overfitting that is learning perfectly the training set, so when you go over new data the model is not able to provide a robust solution, failing on new data. **Overfitting is a problem** of machine learning and its **solution** comes from the problem-solving principle **"Occam's razor" or law of parsimony** that says that "**entities should not be multiplied without necessity**" i.e. **transposed into machine learning: refer to the simplest model you can**. So since neural networks are universal approximators you may be tempted to go as complex as possible fitting any function or learning any possible classifier but beware since once you see new data your model won't work as expected due to overfitting.

Overfitting problem is highly coupled with the model complexity:

- **Too simple model underfit data**. In machine learning having a **too simple model** is also called having a **high bias**.
- **Too complex model overfit data** and do **not generalize**. The neural network is able to fit all the data passing through all the points but a point very

close to the optimal model may have an very high error: we have **very low error (perfectly fitted) on the data of the training set but very high error (not properly fitted) on new data**.

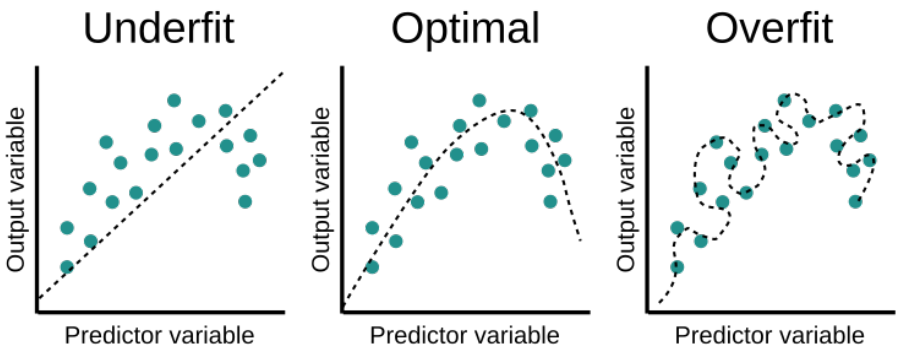


Figure 5: Overfitting and underfitting in regression

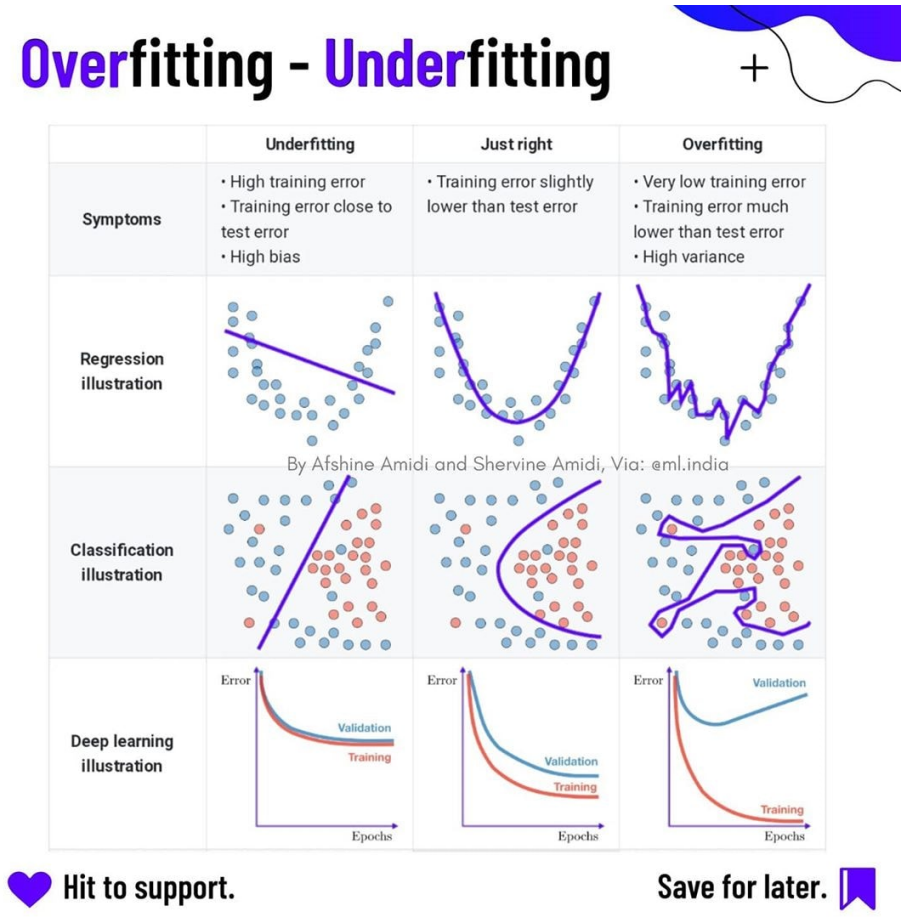


Figure 6: Overfitting and underfitting in different situations

Hence **selecting the right complexity for the model is one of the typical step** in the process of **machine learning**: you start from simple models and then increase the complexity. **In neural networks the problem of being able to doing anything (theorem) may lead to overfitting the model**.

How do we measure if our model is working properly i.e. how do we measure generalization? Training error/loss is not a good indicator of performance of future data, because the fit of the model on training data is always biased:

- The classifier has been **learned from the very same training data**: any **estimate based on that data will be optimistic and biased** i.e. optimistically biased.
- **New data will probably not be exactly the same as training data**

- You can find patterns even in random data, if the model is complex enough

Usually to test a machine learning algorithm we **need an indipendent** new **test set**; this can be done in different ways:

- **Someone provides** you **a new (test) dataset**.
- There is no one that can find the data and give it to you so you **split the data into training and test**: the **training set** is used **for** the **parameter tuning**, so for training the model and the **test set** is **used** to **evaluate the goodness of out model**. This is called **hold-out set** validation. This means that we hide some of them for later evaluation.
- **If the dataset you have is small and you don't want to split it** (since this would lead to even smaller training set and test set) you can perform **random subsampling with replacement (i.e. copies the samples and put it in the test set) of the dataset**. The **replacement creates a correlation between training data and test data**, so we have to **take into account** that the **estimate** of the performances **will be biased**. The random **subsampling can be done several times and then take the average errors**. This method is called **bootstrap method** but since is a **biased method** we **should avoid it and should be done only on very small datasets**.

  NOTE subsampling means to take a random subset of the whole set of samples (i.e. dataset) by coping them (e.g 50% of random subsampling over a 20 samples dataset means take randomly 10 samples coping them: training data will be the set of 20 samples and the test set will be the 10 random subsampled samples).

NOTE **training error** cannot be ignored since it is **used by the backpropagation during learning**: what you cannot do is believing that the error on new data will be the same as training error, how far is from it is a matter of how much you are overfitting.

NOTE the test set should come from the same distribution of the training set otherwise, if it comes from a different distribution it does not make a lot of sense. E.g. problem of recognizing car and motorbikes from a pictures dataset and then you use as test set pictures of boats: this are different distributions and the answer of the model will be basically random since there is not a correlation between the output label and the input. Sometimes happens that the test set comes from a slightly different distribution, this phenomenon is called covariate shift (we don't see it in this course).

NOTE the bootstrap method in practice is usually considered considered for dataset that contain less than 100 samples; if the number of samples are more there are other approaches.

NOTE **Splitting data will result in a model trained on less data**, so this **introduces variance** but variance **exist anyway from the fact that you train the model on different data on the one you test**: there is nothing we can do, variance is an **intrinsic property of the model and of the size of the dataset**; the real problem of variance is in the number of parameters not in the number of records.

You should always have the same distribution, especially with **classification**: if you want to do a proper evaluation of the model you **should preserve the distribution between training data and test data**. In classification this means that you should have the **same distribution of classes in training and test date especially if they are unbalanced** (e.g. if in the training set there are a lot of cars and only few motorbikes, should be the same for the test set), so is **very important** to **maintain the proportion of the data**. The fact of preserving the distribution is called **stratified sampling**: you **first split the data in classes and then get the same percentage of data from each class such that the proportion between the classes in the test set would be the same of the one in the training set**.

Until now we have seen the idea of testing on one set: assuming we have a set of records we **shuffle** them and then we split the set in training (e.g. 85%) and test (e.g. 15%). The problem of this approach is that the **performances will depend on ho lucky the split is** (e.g. examples very very simple ending in the training set and the one very very difficult ending in the test set): the evaluation could be unfair, so usually we **want to have a fair evaluation**. What can we do **to have a more solid decision?** If even if taking the 10% of the dataset to create the test set is too much to get a fair evaluation we have to find a different procedure to validate your data: one of this technique is called **leave-one-out cross-validation**. With LOO cross-validation we **hide one sample, train on all the others and then compute the error on the hidden samples**: this is **done for all the samples in the dataset**. This approach has some issues but is **more correct as an estimate of error on unseen data** since it:

- **Train with all the data** which would be the ideal case.
- **Test on all the data**.

The result is that the estimation is ubiased, differently from the one obtained using a hold-out set. In the **hold-out set validation** you train on some and test on the other and **depending on how lucky the split is and how much uniform is the data you might get a good or bad estimate on the test error**, instead **in this case you train and test on all the data**. The **problem of LOO** is that you **can do it only with few data samples because you have to repeat the procedure for each sample**, so **if the samples are too many this procedure is impossible and you have to use the hold out set validation**. Furthermore exists an intermediate procedure which is a **trade-off between hold out set validation and LOO cross-validation** called **k-fold cross-validation**: the **dataset** is **divided in k groups** and the **training and test procedure is repeated k times each time**

using one of the k group as test set and the other groups as training set:

$$\hat{e}_k = \frac{1}{|N_k|} \sum_{n_k \in N_k} E(x_{n_k}|w)$$

where $\hat{e}_k$ is the error obtained at each iteration and **then we average all the k errors $\hat{e}_k$ to get the average error on new data**:

$$\Rightarrow \hat{E} = \frac{1}{K} \sum_{k}^{K} \hat{e}_k$$

in this way the **model** is **trained for k times on all the data and tested at one time on each samples**. **Usually this estimate is more accurate than the hold out set and the LOO cross-validation**.

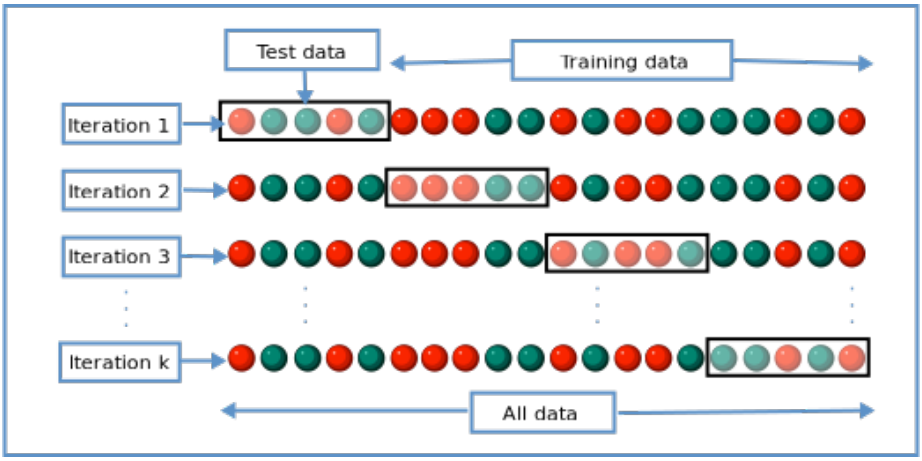


Figure 7: K-fold cross-validation

So **the best option to test and select a model is k-fold cross-validation**, the **second best option is to perform hold-out** set validation then **for small dataset we can use LOO cross-validation**.

NOTE The problem of hold out cross validation comes from the fact that the performance estimation are biased by the split we choose, leading to a not precise estimation. It can be used if a lot of data is available since the split reduces the amount of data used for training.

NOTE cross-validation is a variation of hold out set i.e. it split the data and hide some of them for later evaluation, so if someone provides you a new dataset as test set we don't need to apply it.

NOTE the word cross means that you validate your model using the original dataset and not an external validation set: but since you use the original data for test you cannot use them for training (that would cause a biased validation) and you split them.

NOTE **LOO cross-validation with small dataset is the best thing to do**, instead with **big dataset is impossible to do**.

NOTE in k-fold cross validation each iteration i.e. **each time we have a different test set it is a different model that we train** and this doesn't have to do for how many epochs we train it: e.g. model to predict rice cost of tomorrow, I have the dataset and train it for 1000 epochs; if I don't test the model I have no clue of how good the model is so i have to apply k-fold cross-validation to find out and to do that I train and test the model for 1000 epochs for each of the k fold.

NOTE **To understand the best number of folds we should do on our dataset we can understand the variance of the estimator** $\hat{E}$: there is not a theoretical bound since the **problem is more practical** because **bigger the folds more are the training iteration** (i.e. longer training), so **usually the folds are not much** (e.g 5-10).

NOTE **k-fold is not a biased estimation since the test sets are independent**: **for each fold the evaluation is unbiased and** $\hat{E}$ **is the average of unbiased estimations**.

NOTE in k-fold cross-validation each fold is a different training procedure over different training sets, so it does not amplify overfitting: if it was using 5 times the same data it is only training more.

NOTE k-fold cross-validation is better than LOO since sometimes datasets has double instances, so LOO might be tricked by the fact that you have double instances: exact copies of the same data point in the dataset would lead to not having a total independence between the data. **In theory LOO averages more samples so as theoretical convergence is less unbiased than k-fold cross validation** but: first of all is unpractical with big datasets and second having some doubled data points the result may be tricky.

NOTE **k-fold cross validation can also be applied with stratified sampling in classification problems**.

ATTENTION until now we didn't spoke about validation set: we spoke about training set, test set and validation i.e. the **procedure assessing in an unbiased way the quality of the model**.

NOTE the term validation refers to the evaluation of the quality of a model and you do it by considering a training set and a test set. **Using k-fold cross-validation it means that the test sets are k**: the idea is to validate i.e. estimate the performances k times for k models on different training and test set: training set is used to fit the parameters and the test set is to asses how good is the model.

NOTE not doing this things properly is the worst error we can do: validation is a critical step.

NOTE there is not an issue with **cross validation** in general. The **only issue** is that the **training has to be performed more than once**: this let you to **obtain a better estimation of the model performances but extends the training time**; another issue can be considered the fact that you have different models each one trained on a different set and you have to choose what to do with them.

ATTENTION never ever use the result from training set as validation measurement of performance of the model: **validation must be done on new data since it has to measure the capability of the model to generalize**.

The strategies for validation are:

1. **Ask for new data**: this makes life easy.
2. If you cannot obtain a new set **perform a hold out validation**.
3. **If the hold out set creates biased validation depending on how the split is done perform k-fold cross-validation validating using the average of the k models**.

This procedure is very important and distinguish someone that knows what is doing from someone that does not: **NEVER TRUST THE TRAINING SET ERROR due to overfitting**.

From k-fold cross validation I obtain k different models from the k different trainings: **what do I do with these models? Which one I pic?** In practical applications there are two options:

- You **keep them all and average them** (e.g. **classification**: each model tell a class and then you **take the majority order**; **regression**: each model gives an output and you **take the average of the output**). This **build an ensemble method** called **bagging** that **let us to obtain in general a better generalization performance from the model**.
- Why did you do all these evaluations? Sometimes these evaluations are done to understand the number of neurons and epochs to train, hence **cross-validation** is **done to evaluate different models and at the end of this procedure you obtain the best model you want to train**, i.e. you performed hyper-parameter tuning. At that point you pic the model you found

and train it again from scratch. Is **suggested not to use this method for neural networks, if done it must be done under some hypothesis**.

NOTE in what we will see we have no clue on the distribution and variability of the data. To say something about how much data is needed to train and test we should take into account also how data is distributed, but in general for neural networks we can consider a dataset composed by hundreds of samples small but starting to be a set of thousands becomes ok.

How to apply validation to neural network in practice? i.e. how I estimate the performance of my model? In practice I train the data with the training set and the use the test set for the validation: if the **two error evaluations differ considerably the model is overfitting**. **Unfortunately this does not help in reducing or solving overfitting i.e. does not tell you what you should do to the model to reduce the difference**: if I do something to the model and the gap reduces it means that the something was useful but **the fact that I measured the gap is not a technique to improve the model**. **So validation is only measuring the amount of overfitting but is very important**.

Based on the validation of the model there is an important technique to reduce overfitting.

NOTE Cross-validation is the use of the training dataset (that is different from the training set) to both train the model (parameter fitting + model selection) and estimate its error on new data (i.e. generalization performances).

Remember that **cross-validation** can be **applied both** for **model assessment, i.e. to evaluate the performances of the model on new data**, and **for model selection, i.e. hyper-parameter tuning**.

3.7 Preventing Neural Networks Overfitting

Introduced the problem of overfitting, defined as a **discrepancy between the performances of the model on training data and new data** (i.e. **low capability of generalization**), and how this discrepancy can be measured, training on some data and test on new data, we can now look at some **techniques used to prevent overfitting**.

The first technique we see is the simplest and is called **early stopping**. Let's suppose we train on some data and what we expect, especially for very big and complex models, is that with very big datasets the error will go to 0: the theorem of universal approximator say that this error if the model is big enough goes to 0. So the **problem** is that **if the training error at some point goes to zero the model overfit**. Hence I **need a way to stop the training before the model overfit losing its generalization capabilities**.

So overfitting networks show a **monotone decreasing** training error trend (on average with SGD) as the number of gradient descent iterations k, but they lose generalization at some point. So the **process to limit the overfitting** of the network **consist** in:

- **Hold out some data as validation set** (if we decide to use **k-fold cross validation this step will create k different model and related training and validation set**)
- **Train on the training set for an epoch**
- Perform cross-validation on the hold out set (the validation set) i.e. **compute the validation error**.
- **Stop train** when the **validation error increases** w.r.t. the training error, **obtaining the final model**.

i.e. **after each epoch validation is performed** applying early stopping and the **test of the model is outside the procedure of building the model**. So early stopping **is an online estimate of the generalization error**. Since the test procedure is outside the building procedure of the model, if you start again the training after the test the model will be biased because in this way it is not anymore an independent test, you have conditioned your training on the test result and this is a thing that should never be done.

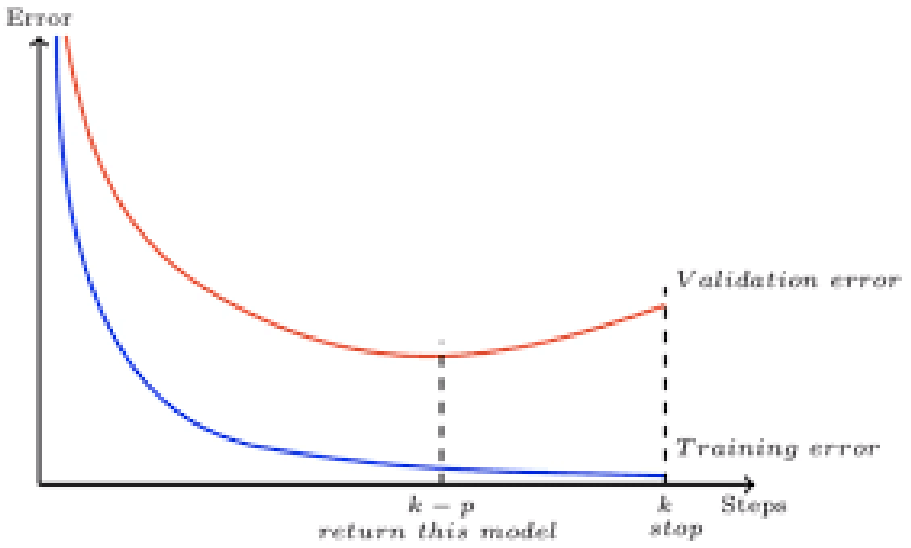


Figure 8: Early stopping

Usually test data have a error that trend more similar to the validation data than to the training data. But this **does not means that the model we choose with**

early stopping using the validation set will get the best possible results (less error) on the test set, maybe we needed to continue the training for a bit more or stop it before (as in the following picture) to obtain the minimum error over the test set: this **should be expected since the test set is composed by new data never seen and we will never know during training where is this best point to stop to obtain the minimum test error**.

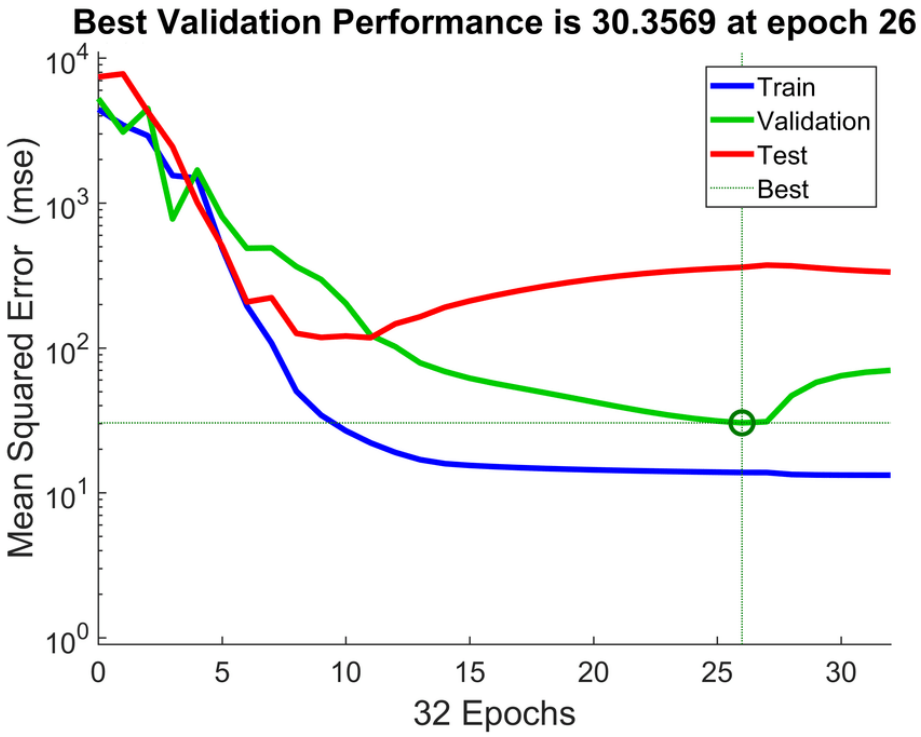


Figure 9: Example-accuracy-evolution-of-the-training-validation-and-test-datasets

Example e.g. production of a model for a certain task for a company: we train the model (i.e. minimize the error) using the training dataset and then the company does the final evaluation of the model using a test dataset to decide if the model is good enough to pay us. So we should want to understand how the training is going and for this reason we use the validation set: the training set is used by us to minimize the error, **the validation set is used by us while training to understand if the network is overfitting and used to make decision** and the test set is the final evaluation of the final delivered model done by someone else.

We **introduce the validation set**, a set we use during the training procedure to evaluate the training of the model. This means that the **validation error, since is computed during the training, can be used to take decisions**.

NOTE the **validation error** can be computed at each iteration of the training (usually done) or at the end of each epoch. **For early stopping is better to do that at the end of the epoch to be more resilient to the noise on the mini batches**.

NOTE epochs are mandatory because you are doing **gradient descent**, you will be never be able to perform training error minimization in one step. **You have to perform may epochs to reduce the training set error**.

NOTE **when you compute the test error the model has finished its training**.

NOTE since the evaluation must be done on new data the validation set must be composed by new data: to do that some data is hold out from the dataset. Furthermore **validation set will be different from the test set because you are allowed to take decision on the validation set**, so in a sense you will customize your model to stop in a certain point that may be not optimal for the validation set since we don't know the testing set. Hence **ideally training, validation and test set should be indipendent (i.e. disjuncted) set**.

NOTE training and validation set are involved in the preparation of the model, instead the the test set is involved in a practical context when the model must be used (sometimes you don't have the final use so you simulate it with the test set).

NOTE **the validation error when the model start to lose its capability to generalize start to rumping up**: the model overfit (starts to memorize the training data losing generalization) and the error on new data (i.e. validation set) increases.

NOTE to sum up we **use validation set to find when/where stop our training**

The **k-fold cross-validation comes in at all levels**, so you can adopt a k-fold approach at all level: you can **even apply it at the very beginning training, using a different validation set for k model**; this gives you the average performance of your model so **this can take a lot of time**, **that's why usually only one validation set is hold out but ideally k-fold cross-validation can be applied also in this case**. Since is applicable to all levels it can also be used by the tester for the test set: consider k-folds of the dataset holding out each time only one fold, then the remaining data each time is the training dataset from which can be holded out a single validation set (simple hold out set) or can be applied k fold again.
Hence early stopping is one approach that uses the validation set to stop the training before overfitting, and at that point your model is done. What we discussed earlier about the test set must be applied regardless of the usage of the early stopping technique or any other technique. To asses the quality of the model people invented the validation set but it is not the test set, is another set.
In practice, to know when the validation set error minimum is reached you **monitor**

the validation error for a longer time with respect to the minimum: you will see that the training error continues to go down and the validation error increases. So by putting some rules as "**if validation error is not decreasing after 10/20 epochs stop the training**" we will obtain the minimum of the validation set that was saved together with the fitted parameters.

NOTE when you have a model that can fit everything (theorem of universal approximator) by definition the **global minimum of the training error is reached with overfitting**. So this is the **issue with neural networks: very powerful but it can be too much (overfit)**. **Hence we don't want to reach the global minimum but the local minimum on the training set for which the validation error reaches the minimum, i.e. where the model is still able to generalize on new data**. Is not at all guaranteed that the validation set has a unique global minimum: staring from a different set of random parameters the plot validation and training error plot will change every time and the plot is usually very noisy; that's why **with early stopping we don't look at a global minimum of the validation error but we apply some heuristic to stop (e.g. error not decreasing for k epochs)**.

NOTE **the overfitting of the model depends on both the complexity of the model and the number of epochs of the training: that's why the early stopping technique** (which is cross-validation) is used to prevent overfitting by stopping the training earlier not letting the model to overfit the data.

NOTE supposing that the test date is not a new dataset but is holded out from the dataset we proceed in this way:

1. the **dataset should be firstly splitted in training and test sets (can be done also with k-fold cross validation)**: test set is used for the **model evaluation/assessment** (or validation: not to confuse with validation set) by the **(pre)production team**.

2. the **training dataset** obtained from the previous split **is then splitted again in the real training set and the validation set (can be done also with k-fold cross validation)**: the training set is used to **train the model** and the validation set is used for **model selection**, both done by the **development team**. Model training means backpropagation and model selection means, for example, early stopping.

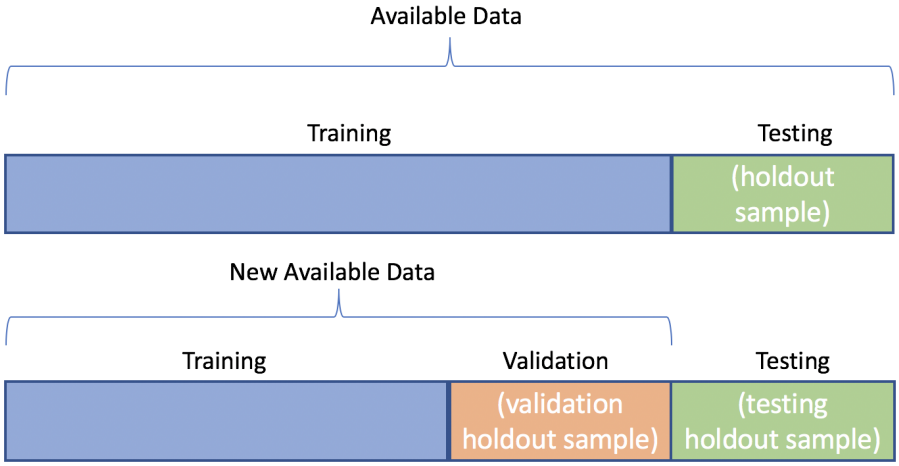


Figure 10: Splitting of a dataset

Usually data are **splitted** in this way, but it **really depends on the dimension of the dataset**: 10-15% for test set, 10-20% for validation set and 60-80% for training set. The test is done on the on the final model and must be done using a test set disjointed from the validation set since you could take some selection which basically might be too optimistic because when you select you have seen the data and when you select in a sense you have overfitted the validation data.

NOTE we do not early stop considering a change in the derivative sign of the validation error since the gradient descent is very noisy.

Validation set is used to do online estimate of the generalization error. If you train on a training set and then using a **validation set you obtain a bad result in the error we can deduce that the model is not well built and must be changed**, indeed the validation set represent a scenario where the model is exposed to new data i.e. it try to see if it is able to generalize. So **if the validation error is bad it means that the model probably is overfitted since is not able to generalize on new data i.e. the validation set**.

In early stopping the training is stopped when the model starts to overfit, but in general there is the **idea of training validation and testing also for** other aspects of the training like the **hyper-parameters**: when we choose to stop training at a certain epoch that is an hyper-parameter; the number of hidden neurons; the number of layers in a neural network. In general **model selection (through the validation set) and evaluation (through the test set) happens at different levels**:

- **Parameters level** i.e. when we **learn the weights** w for a neural network

- **Hyper-parameters level**, i.e. when we choose the number of layers L or the number of hidden neurons $j(l)$ of a given layer
- **Meta-learning** i.e. we **learn from data a mode to chose hyper-parameters**

The idea of training and validation is that you have two sets: one for training the neural network (training set) and the other for validating the neural network (validation set). The **difference between the the validation set error and the training set error gives to the developer an idea weather the training is going bad or well i.e. how good is your model**. Supposing we have a network with only one hidden layer with J neurons, by training and validating using early stopping, the final error on the validation set is: (ES stands for early stopping)

$$E_{ES}^{(J)}(x|w) \quad \text{at iteration } k_{ES}$$

that is the generalization error (i.e. the error on new data) with J neurons in 1 hidden layer. We can **repeat the process starting from $J = 1$ and stopping at a certain point H (always considering a single hidden layer)**:

$$E_{ES}^{(i)}(x|w) \quad i = 1, 2, 3..., H$$

In this way we **obtain a series of generalization errors that decrease as the number of neurons in the layers increase until a certain point where the generalization error increase since every neurons you add will try to overfit**: early stopping could prevent the model to overfit the data. **In this way early stopping can be used not only to prevent overfitting but also to perform a model selection**, selecting **how many neurons should the neural network have in the hidden layer**.

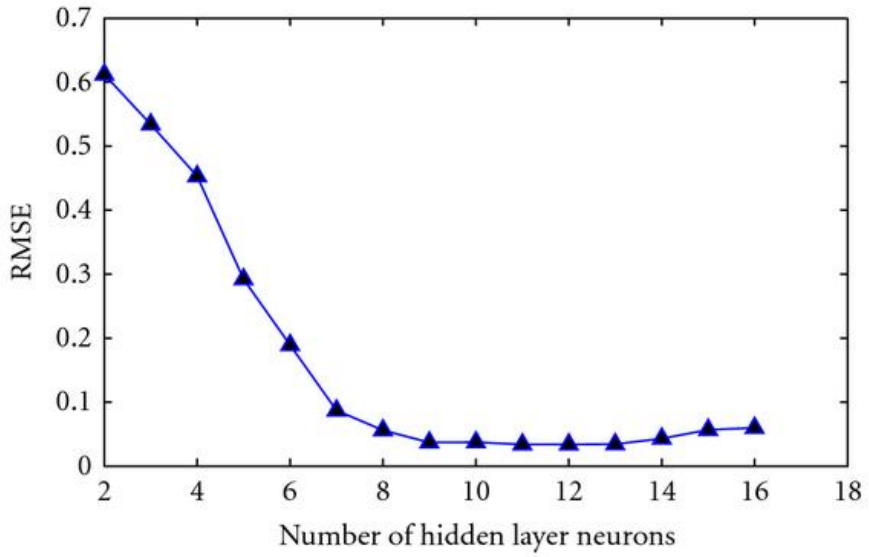


Figure 11: Hidden neurons number selection.

On average the more neurons you add the lower is the early stopping validation error, up to a certain point after which the validation error start to show the overfitting behaviour of the network. For example in the last image before 10 neurons the model was **underfitting since it was too simple adding more neurons helped to reach a better result**, but **after** 10 the model is **not able to reduce the validation error** (can be seen as the more complex problem would overfit); furthermore after 13 the error increases: this happens because **each time we train and validate with the validation set the weights are initialized with different values so the validation error could result lower, equal or higher depending on the initial weights, but on average it would remain equal to the error in 10** (the average curve goes to E_{10} to the infinity: since is an average we can have also lower or grater error). Obviously **in general the epoch at which you stop adding new neurons may stop at a different** k_{ES}: for significantly different models we should expect significantly different k_{ES}.

NOTE **changing** the number of neurons in the **model** may cause the early stopping to **stop the training at a different epoch than before**. **But also by training the same model the stopping epoch may be different due to different random initialization**: depending on the initialization the model could **converge on a different minimum ending in a slightly different number of epochs**. (usually we are not interest in the number of epoch needed but only on the validation error).

NOTE if we are adding new data to our dataset and this new data add only variance i.e. they don't provide more information probably they don't reduce the validation error on which we stop, but adding data we may need a more complex model i.e. the procedure of hyper-parameter selection would have the minimum error for a bigger number of hidden neurons.

NOTE deciding the dimension that my dataset should have to fit a certain network is a **very complex task**; we reverse the question saying **how big the network can be before overfitting a certain dataset**. So **the dataset decide: if the dataset is not big you can either reduce the size of the model or keep it big and preventing overfitting with early stopping or use other techniques to improve generalization**.

NOTE **the generalization errors curve may be not monotone due to the fact that each validation may be start from different weights set and the (batch) gradient descent is noisy so the whole error curve will be noisy**.

ATTENTION by **using the validation set over and over we are kind of overfitting it**, **that's why** at the very end the real quality of the model will be given by a new and never seen set, the **test set**. The validation set is a sort of proxy of the validation error and since we continue to **use it several times we would become too optimistic in considering our model** and because of this the final statement of the accuracy of the model would be given by the test set. In this sense the **validation error is an estimate of the generalization error** i.e. an estimation of the difference between what you expect from your training and what you get from a new different dataset: the **point where training of the network stop gives an estimation of the generalization error**, of the performances of the model on data used for training. The fact that **using the validation set to select the model makes, in a sense, the validation set used for training the model**, hence the **the model selection is biased/fitted on the considered validation set** and it is for this **reason** that the final performance evaluation is evaluated on a new set (**test set**) **and the test error may not be the optimal (minimum) one**. The continuous usage of the validation set cause overfitting even if we are not learning on it because **overfitting** is defined as **having an optimistic estimate on the error with respect what you will get on the real practice**: overfitting over the validation set means that the **procedure is optimized over the validation set**, and it might happen that by going on production you **may obtain less good result** since the choice was based on the validation set. The fact that even validation overfit the model does not mean that is the same of using the training error as estimate of the generalization error of the neural network, since as said the training error is an awful estimate for the network generalization capability because you can always send it to zero just increasing the number of parameters.

NOTE **if during the training the validation error is bad we have a problem in the model since the validation set cannot be changed because the**

validation set is the data you set apart from the training ones: changing the validation imply changing the training, so we cannot easily change the validation set during the training.

NOTE the ideal case would be to have an infinite dataset: if training set, validation set and test set are huge there is no problem, when they are limited you have to be cautious on what you do because you can start to have problems (e.g. overfitting).

Setting up one network that works with the dataset is easy, the real difficult part in training a neural network comes when you want to find the right network or improve your network since you have to setup a metodology: the methodology we saw require to setup a validation set, to use it on different models changing the number of neurons of the hidden layer and finally to select the configuration (the hyperparameter set) of the model that gives you the best validation error.

NOTE **during the model selection (using the validation set) and the model validation (using the test set)** the network is feeded in the input but the target output is considered but not for backpropagation and setting up the weights: **the only thing done in this phases is the computation of the error considering the target output and the output of the network**. **Backpropagation indeed is only performed at training time** (*) when the weights of the network need to be set correctly. Hence in the plot backpropagation is used to reduce the training error but then **the validation error (y-axis) is computed using the weights learned in the corresponding epoch (x-axis)**.

The **configurations of the neural networks are a lot, so to find the best model we should try them all**: one layer with one neuron, two layers with one neurons, three layers with one neuron, one layer with two neurons and so on. By doing this we **obtain a validation set error for each configuration i.e. a grid of validation set errors on different model configurations** indeed the **search of the best model** over the one we tested is called **grid search**. The **hyperparameters research won't go too far** in trying all the different network configurations since for the Occam razor principle we would **select the simplest model with the best validation set error**. Is **not feasible to search in all the space of hyperparameters** (i.e. all the possibilities) so we **take only samples of it**: for example having two layers of hidden neurons we would test the network validation error for 5 neurons for each layer, 10, 15, 30; furthermore **the sample of the hyperparameter space can be done with an intelligent automatic way (e.g. genetic algorithm)**.

NOTE early stopping is treating the number of epochs as an hyperparameter by choosing the epoch to stop the parameter learning. Then without as we have seen with the number of hidden neurons you select the other hyperparameters without the early stopping by considering the validation errors obtained.

Training can be done without early stopping but we have to do something that won't leave us with a huge gap between validation set error and training error at the end of the the training i.e. to **prevent overfitting** in the sense of bad capability of the model to generalize the results. **Early stopping** is only one of the techniques to prevent overfitting but is the **easiest one, the one that is more effective and reduces the training time the most**.

NOTE the training error reduction is connected to the number of gradient descent iterations but is also to the complexity of the model (number of layers and their number of neurons). **In some conditions also the choice of a proper learning rate may lead to a lower error or a bigger one if the choice is not done well, and for this reason it is one of the hyperparameters**. Then can be added other elements to reduce the gap between validation set error and training error.

NOTE **hyperparameters selection basically means the model selection** but to choose the right one we have to train different models and look at their performances through the validation set: the problem is that if you **train the models without thinking you may overfit them, so we use techniques like early stopping to prevent this**.

NOTE **validation set is used for both early stopping and hyperparameter tuning**.

NOTE It exist another level of model selection called **meta-learning** (we don't see it): we only discuss about learning weights (parameters selection) and model selection (hyperparameters tuning) which are the number of layers, the number of neurons in each layer, the activation function and some other.

As said early stopping is not the only way to limit overfitting, we will discuss about:

- **Weight decay**
- **Drop out**

These **three techniques** are the **most used to train neural networks preventing overfit**.

LESSON INTRODUCTION START
The problem of overfitting: neural networks (or models in general) may be too powerful and fit the training data which is finding the global minimum for the error on the training data that causes overfitting i.e. lack of generalization: if the training error is performing too well with respect to the test set error is called overfitting, instead not going as well as you like you call it lack of generalization. This is mostly

due to the fact that neural networks are universal approximator and you might end up to build neural networks with million of parameters, so this means that they are so good in approximating your data that they are able to drive your error to zero. For instance having an overly complex neural network will imply that more epochs you train more likely you will overfit the dataset. Drawing the error to zero is good in theory but not in practice since what happens is that the network has a zero error on the training data but when you test it over an hold out (for example) set called validation set you obtain much worse results: performance on validation set and on the training set will diverge since all the deterministic information were extracted from the data by the universal approximator and after that what you get is just fitting the noise of the data. Before the divergence occurs we can stop the training of the model and report that the model we found is the best generalizing model we could train with those training data and validation set.

The validation error we obtain when we stop the process is a proxy error of what will be the generalization error on the test set i.e. a set of data that the model has never seen: is only a proxy error since the test set is composed by new data and since the validation set is used to choose when to stop the training in a sense it overfits the model because the error it computes will be an optimistic estimation of the one of the test set.

Since the stopping validation error is an estimation of the generalization error, it can be used to make many choices on my network: the network can be optimized under different perspectives: during training you optimize the weights but during optimizing procedures you can optimize how many neuron you have in the layer by making a lot of tests. The decision can be taken using the early stopping error of the many test we have done. [LESSON INTRODUCTION END]

With early stopping we trained a big model and used a validation set to prevent the model to overfit by stopping it earlier.

Early stopping is the simplest way to prevent overfitting and basically what it does is compensating for every overestimation of the required parameters that you might have put in your network. But **early stopping as one important drawback**: **a validation set must be set apart**, that means that this part of data won't be used for the training. At a certain point we may want to try to see what happens to the model if we add more data, and we are wasting data for training into the validation set: we can't think about rotating the data between the validation set and the training set because it means using the data of the validation set for training which will cause an overestimation in the model performance i.e. overfitting. **So how can we use all the data and at the same time prevent overfitting?**

NOTE validation set data, if used, should never be used for training. In a certain sense since it is used for model selection it is still used for training.

3.7.1 Weight Decay: limiting overfitting by weights regularization

Another way to **improve** the **generalization** capability of a model, but **without wasting data into the validation set**, is to perform what is called **Regularization**.

The presence of parameters makes the model very flexible, so this means that the model is free to bend and fit every single data point. We should **limit the capability of the model to go trough (i.e. memorizing) all the data point to limit its probability to overfit** by memo. Hence regularization is about **constraining the model "freedom"/flexibility, based on a-priori assumptions on the model, to reduce overfitting** i.e. improving generalization. [So it is a trade-off between variance and bias: you limit the power/flexibility of the model i.e. **increase the bias of the model but at the same time you reduce the variance** because the freedom is the variability of the model; by doing that you expect better performances of the model since the generalization error is given by bias^2 + variance.](Machine learning course explanation)
There are **different way to do regularization**:

- The obvious and extreme one is to **put less parameters**. This can be easily done by removing some of the links that means zeroing the corresponding weights.
- Try to **shrink the weights down by making some assumption on the fact that they should be small**.

So far we have maximized the data likelihood:

$$w_{MLE} = \underset{w}{\operatorname{argmax}}\ P(D|w)$$

and we have seen how the error function we have used so far are the solution of the maximum likelihood estimation problem: they maximize the data probability to be explained by the weights, hence they basically look only to the data fitting because the **optimization is done by only looking to the data probability**. So the best thing you could do for maximizing data probability is to fit the data perfectly and because of this we have to stop the optimization procedure with early stopping.
Writing a different loss function, one **that generalizes by construction without the need of early stopping**. You can decide you don't want to find the best fitting of the data leaving the weights free but you would like to **limit the freedom of the optimization algorithm looking for the weights**: with early stopping you can imagine this freedom of weight choice limited by the fact that at a certain point the training is stopped.
The **regularization can be interpreted under the maximum a-posteriori approach**. We can **reduce the model "freedom"** by **using a Bayesian approach**: it seems more or less the same thing but according to the Bayesian approach you can **find the most likely weights given the data**, which **is very different to the latter weights that maximize the likelihood of the data**:

$$w_{MAP} = \underset{w}{\operatorname{argmax}}\ P(D|w)$$

that is, using the bayes theorem:

$$w_{MAP} = \underset{w}{\operatorname{argmax}}\ P(w|D) = \underset{w}{\operatorname{argmax}}\ [P(w|D) \cdot P(w)]$$

so the **probability of the weights given the data** is proportional to the probability of data given the weights (is exactly what we have been doing for MLE) times the **a-priori probability of the weights**. Hence **the a-priori probability is shaping the weights search space for the algorithm**: taking into account that the distribution of the parameters (weights) is $P(w)$ you can optimize the probability of the data given the weights. So we are **making an assumption on the parameters (a-priori) distribution**. There are **different ways of designing priors, but most of the ways to design priors are based on two things**: some **knowledge you would put into the algorithm**, i.e. it makes sense and describes some heuristic ideas, **and some opportunistic way of writing this probability distribution, i.e. it makes the computation easy**.
So the **maximum likelihood (maximizing the probability of the data given the weights)** and the **maximum a-posteriori (maximizing the a-posteriori probability of the parameters given the data)** are **two possible way to train/fitting a model** by finding its parameters. The real **difference** between the two is the fact that **in MLE data is the only thing that matters** and in **MA you can say something about the weights that the algorithm has to take into account**.

From **empirical observations** we observed that **small weights improve the generalization** of neural networks: so **regularization limit the flexibility of the model limiting the space that your algorithm can traverse seeking for the weights**. The fact that usually networks that are capable of a better generalization than other have smaller weights means that the **probability distribution of the weights is closer to zero**. To express a distribution close to zero we could, for example, say that:

$$P(w) \sim N(0, \sigma_w^2)$$

i.e. is a normal distribution with mean 0 and a standard deviation of σ_w, that represent in general a weight close to zero plus or minus something: depending on the value of this something the network can be very rigid (weights very close to 0) or very flexible (weights can be far from 0). Hence **changing the variance σ_w^2 of the distribution we can make the weights very regularized (with a small σ_w) or very flexible (with a very big σ_w)**.

Let's see what does it mean in practice considering the regression case, for which $P(D|w) \sim N(g(x|w), \sigma^2)$, and the probability distribution of the weights as $P(w) \sim N(0, \sigma_w^2)$

$$\hat{w} = \underset{w}{\operatorname{argmax}}\, P(D|w) = \underset{w}{\operatorname{argmax}}\, [P(D|w) \cdot P(w)] =$$

$$\underset{w}{\operatorname{argmax}} \left[\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi} \cdot \sigma} \cdot e^{-\frac{t_n - g(x_n|w))^2}{2 \cdot \sigma^2}} \cdot \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi} \cdot \sigma_w} \cdot e^{-\frac{w_q^2}{2 \cdot \sigma_w^2}} \right]$$

where we **considered each weight is independent from each other to write the probability $P(w)$ as the product of the probability of each weight**. Unfortunately **the noise σ in the data is not known and σ_w is the limiting factor of the regularization, that tell how close will be the distribution of the weights to**

the mean 0, that is also unknown. Taking the natural logarithm of the previous expression we obtain:

$$\hat{w} = \underset{w}{\operatorname{argmax}}\, ln(\left[\prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\cdot\sigma} \cdot e^{-\frac{t_n - g(x_n|w))^2}{2\cdot\sigma^2}} \cdot \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\cdot\sigma_w} \cdot e^{-\frac{w_q^2}{2\cdot\sigma_w^2}}\right]) =$$

since the logarithm of the product is the sum of the logarithms we obtain:

$$= \underset{w}{\operatorname{argmax}}\, \left[\sum_{n=1}^{N} ln(\frac{1}{\sqrt{2\pi}\cdot\sigma}) + \sum_{n=1}^{N} ln(e^{-\frac{t_n - g(x_n|w))^2}{2\cdot\sigma^2}}) + \sum_{q=1}^{Q} ln(\frac{1}{\sqrt{2\pi}\cdot\sigma_w}) + \sum_{q=1}^{Q} ln(e^{-\frac{w_q^2}{2\cdot\sigma_w^2}})\right]) =$$

since $ln(\frac{1}{\sqrt{2\pi}\cdot\sigma})$ and $ln(\frac{1}{\sqrt{2\pi}\cdot\sigma_w})$ are constant for the $\underset{w}{\operatorname{argmax}}$ because they do not contain w we obtain:

$$= \underset{w}{\operatorname{argmin}}\, \left[\sum_{n=1}^{N} \frac{(t_n - g(x_n|w))^2}{2\cdot\sigma^2} + \sum_{q=1}^{Q} \frac{w_q^2}{2\cdot\sigma_w^2}\right]$$

where the $\underset{w}{\operatorname{argmin}}$ comes from the fact that we had a multiplicative -1. Differently from what we got from the maximum likelihood estimation the $\frac{1}{2\cdot\sigma^2}$ does not cancel out. To do that we have to multiply everything for $2\cdot\sigma^2$ obtaining:

$$\hat{w} = \underset{w}{\operatorname{argmin}}\, \left[\sum_{n=1}^{N} (t_n - g(x_n|w))^2 + \sum_{q=1}^{Q} \gamma \cdot w_q^2\right]$$

with $\gamma = \frac{\sigma^2}{\sigma_{\mathbf{w}}^2}$, where σ_w is the assumption for the weights model flexibility and σ is the variance of the noise (error) of the data (unknown). In the latter expression we can distinguish **two terms**:

- The **fitting term**:
$$\sum_{n=1}^{N} (t_n - g(x_n|w))^2$$
is the same term **obtained with maximum likelihood estimation** (always speaking about regularization) and so **tells you how much you should fit the data**.

- The **regularization term**:
$$\sum_{q=1}^{Q} \gamma \cdot w_q^2$$
that tell you **how much you should minimize the weight**.

Hence **the more you try to fit data bigger will be the first term and more you try to regularize (make a more rigid model) bigger will be the second term**. The **trade-off between fitting and regularization** is **given by** the $\gamma = \frac{\sigma^2}{\sigma_{\mathbf{w}}^2}$.

NOTE **between the weights there is a clear co-adaptation that makes their probability not independent**. We **assumed them to be independent** to solve the weight regularization problem in an easy way, but by doing that we **obtain an approximation of the joint probability of the weights, but it is good enough**.

NOTE from two unknowns σ and σ_w we express the expression to minimize in term of a single unknown γ. This let us to **interpret the formula in an easier way depending on the value of γ: if γ is very small ($\sigma^2 << \sigma_w^2$) we are basically just fitting going back to the maximum likelihood estimation; if γ is very high you are highly regularizing so basically you are ignoring the data pushing the weights to zero**.

NOTE you don't want exactly the a-posteriori distribution $P(w|D)$ but you want to find its maximum, so the probability of the data σ coming from $P(D|w)$ that comes from the bayes theorem basically is useless because is treated as a constant term, and that is the most difficult part to estimate in bayesian statistics (you should compute the integral of the probability distribution).

NOTE **the distribution of the weight is chosen by us: if we don't like the gaussian we should find another distribution which make the a-posteriori maximum calculation simpler**. We must **remember that our objective is to concentrate the weights close to zero**, so any distribution that may have most of the mass around zero works (and the gaussian is one of this).

NOTE usually speaking of model **regularization** means to write explicitly an **added cost** (a term that should be minimized) term to the cost function on the parameters of the models (e.g. $\gamma \cdot \sum_{q=1}^{Q} w_q^2$).

NOTE this procedure limits overfitting since has been proven empirically that small weights improve generalization of neural networks: you can think to the weights representing how much does the output change based on a small input change, so basically having small weights means having a small change in the output. Consequently **the smoothness of the model is related to the domain of the weights**: very smoothed models with weights very close to zero basically have an output that is very very close to zero; models not smoothed at all that can have any value of weights they can change very much the output. From this comes the sense of the **term regularization: regular means regular in shape i.e. smoothing the model**.

NOTE the type of **regularization we saw $\sum_q w_q^2$ is called L_2 regularization** or **ridge regression**. Instead the L_1 **regularization** use **lasso** i.e. we have $sum_q|w_q|$: **in this way you are pushing the weights very very close to zero and that's why you get feature selection**.

Basically what we did **finding the maximum a-posteriori probability** is **another loss/error function**, **indeed the task of a neural network is described by the loss function**: we are asking to **fit the data (fitting term) and keep the model simple (regularization term) for a regression task**(***); similarly can be done **for the classification task using as fitting the crossentropy plus a regularization term**. Playing with the loss function is very important, but the problem is that we have to **find a way to set properly the parameter γ, i.e. the trade-off between fitting and regularization**, that is an hyperparameter of the model.

How do we set the value of γ? We have to find the trade-off for regularization. For this reason γ is basically an **hyperparameter** because is a parameter **I have to set before starting training in order to find the right balance between fitting the model and performing regularization**. So **recalling the hyperparameter tuning we can use cross-validation to select the proper** γ:

- **Split the data** in training and validation sets.
- Using the training set, **minimize the weight decay error for different values** of γ:

  $$E_\gamma^{TRAIN} = \sum_{n=1}^{N_{TRAIN}} (t_n - g(x_n|w))^2 + \gamma \sum_{q=1}^{Q} w_q^2$$

  All these training are done on the regularized model **without early stopping because we want to find the right amount of regularization such that without early stopping we get a good result**. (Q are the number of weights of the model)
- **Using the validation set evaluate** the model looking at the prediction error:

  $$E_\gamma^{VAL} = \sum_{n=1}^{N_{VAL}} (t_n - g(x_n|w))^2$$

- **Choose** the γ^* with the **best validation error**
- **Put back <u>all data together</u> and minimize**:

  $$E_{\gamma^*} = \sum_{n=1}^{N} (t_n - g(x_n|w))^2 + \gamma^* \sum_{q=1}^{Q} w_q^2$$

  figuring out which is the best set of weights given γ^* and the set of all the data.

What you expect by training on different γ and evaluating the model prediction error is that, **differently from the effect on the early stopping**, **more you increase the regularization at a certain point for big γ you reach a level where all the weights are zero** (since big $\gamma \Rightarrow \sigma >> \sigma_w$ i.e because the standard deviation is very close to 0 all the weights goes to 0) and **basically you are predicting** $\sum_n t_n$

because you have regularized so much that all the weights are 0 and so g is always equal to 0. For this reason increasing γ at some point the performance of the model start to decrease due to underfitting, i.e. the model is to simple to describe data since it is too much regularized. **Once we select the γ^* we minimize the loss function over all the dataset, indeed the idea of using regularization is to use all the data for training without wasting any in the validation set**. The weight decay error function computed using γ^* has the right amount of generalization so **it can be trained without early stopping and it won't overfit**, due to the selection we did with cross-validation.

NOTE for γ we should test values form 0 to possibly ∞ (possibly since having an infinite regularization doesn't make sense). But as we will see the γ **is usually very small and usually not bigger than 0.2/0.3**. The **search in the γ space can be done in an intelligent way as it was for the hyperparameter tuning**.

NOTE **the splitting of the dataset in training and validation set is done only once and then each value of γ tested uses the same training and validation set**.

NOTE the test set is not considered: is used by a third person to test the performance of generalization of our model.

NOTE in the challenge, for example, putting back all the data makes the model to perform better: the dataset was composed by thousand of data so removing 10/20% of it for the validation set and using early stopping resulted in worse performances.

NOTE another thing that can be done for the challenge is use early stopping: we use early stopping to find which is the epoch to stop in, then use that k_{ES} as regularization factor in the sense in that epoch I'm overfitting, even if that k is not the optimal one but i can try it for the model. (not suggested in real life but in some cases it gives better performances)

NOTE regularization techniques like **weight decay** is that it **penalize the models that are more complex than what you need**.

The regularization through weight decay can be applied on all the weights (as we have done) or on each single layer deciding to regularize only certain layers; only the regularized weight will be considered in the summation. Training and designing (i.e. hyperparameter tuning of) a neural network is very expensive, and a correct subsampling of the possible parameters may reduce significantly the training time.

3.7.2 Dropout: limiting overfitting by stochastic regularization

Another way to overcome overfitting without wasting any data for a validation set like in early stopping is the so called **dropout**, which doesn't have a classical statistical counterpart as was for early stopping (ridge regression).
In reality weights are not independent and sometimes they tend to co-adapt: a neuron will set a certain output because another neuron was set to another. **This fact may prevent a proper learning since one weight may be optimized based on what another neuron has done**. To **reduce this co-adaptive behaviour** dropout was invented: by **turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation)**.

Each hidden unit (neuron) is set is set to zero with $p_j^{(l)}$ probability (e.g. $p_j^{(l)} = 0.3$): **higher the probability higher is the effect of regularization**. So, **for each layer, you have a mask**:

$$m^{(l)} = [m_1^{(l)}, m_2^{(l)}, ..., m_{j^{(l)}}^{(l)}]$$

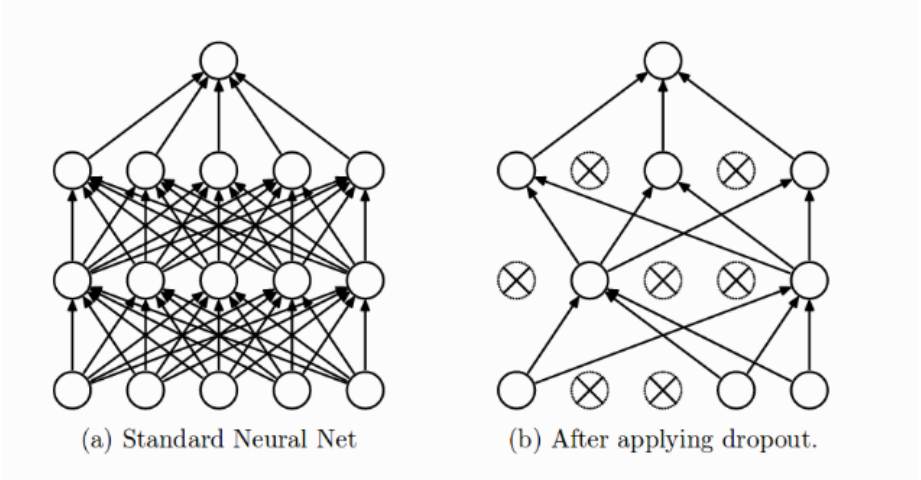
given by a set of bits sampled by a binary Bernoulli distribution:

$$m_i^{(l)} \sim Be(p_i^{(l)}$$

then applied to the corresponding layer masking off the neurons for which the sampled value was 0 (or 1 depending on the interpretation):

$$h^{(l)}((W^{(l)} \cdot h^{(l-1))} \cdot m^{(l)}$$

with $h^{(l)}$ the output of the l-th layer, $W^{(l)}$ the weight matrix of the l-th layer and $m^{(l)}$ the dropout mask of the l-th layer.



(a) Standard Neural Net (b) After applying dropout.

Switching off a neuron means that all the output exiting from the neuron are zero implying that the neurons whose input comes also from one of the masked

neurons will have less input values and is then **obliged to learn the model even with less parameters**. This procedure is done and repeated at each epoch i.e. **at each epoch the mask for each layer is computed again**. This means that at **each epoch the model changes shape since the switched off neurons are treated as they are disconnected from the other neurons**: basically is like they are not existing. At **each epoch is trained a different simplified model with less connection, so a less powerful one**. At the end instead of having a huge monolithic model in which any weight is trained somehow based on the value of all the others, using a different mask for each layer for each epoch **we obtain a model that can be interpret as the sum (or the average) of many other simplified models trained independently one from the other**. Hence dropout **behaves as an ensemble method**: **trains smaller and weaker classifiers**, **on different mini-batches (or at each epoch, depending on the implementation) and then at test time we implicitly average the responses of all ensemble members**: at **testing time the masks are removed** and in this way the output is averaged (by weight scaling) since is like we are considering all the smaller trained networks together being all the neurons active.

NOTE **on average** using dropout **all the neurons are trained and each neuron on average is trained with all the others**; that's why dropout works.

NOTE applying dropout to a certain layer would mean to inserting a dropout layer working as the mask, before the desired layer. The mask can be applied only to a subset of the layers keeping the other "fully connected" each time, or to all the layers. The mask is just a mathematical trick to select randomly ($m_i^{(l)} \sim Be(p_i^{(l)})$ the neurons to switch off each time.

NOTE the practical **reason of applying dropout per mini-batch (i.e. changing the mask at each mini-batch) is that mini-batch training can be done in parallel, so it is an optimization reason** (*).

NOTE the mask element are sampled from a Bernoulli distribution since their value is binary: remember that the **Bernoulli distribution return 0 for p probability and 1 for 1-p probability**.

So in the design phase of the neural network **we have to choose in which layer we should apply the dropout** layers that improve the generalization capability of the network. The **generalization** capability of this technique comes **from the fact that we are obtaining the model as the average of many simpler model**: we are basically **performing (implicit) bagging on the model**. The **decision of where to put a dropout layer is done through trial and error**: where to put a dropout layer is an **hyperparameter** so, **again**, we should **explore the hyperparameter space, validate the models we decide to test and use cross-validation to select the best model**. **Another hyperparameter is the probability of switching out the neurons**.

May happen that adding dropout won't help since it might end up that the model is not capable to learn data with the dropout.

Which method to prevent overfitting should be used? The best is the one that gives you better results, so **you have to do model search**: basically **test many different models and have a methodology to select a model with respect to the other**. The methodology is called **cross-validation** which means to set aside some data not used for training but to decide which the model is the best in this context.

CHALLENGE start with early stopping, select the complexity of the model always using early stopping, try with regularization, see if using dropout helps or not and then you put together the data and train the model with all the parameter set.

There are other tricks to improve the model generalization performances like data augmentation that we will see later on.

3.8 Tips and tricks in neural networks training

3.8.1 Better activation functions

So far we have seen **sigmoid** and **tanh** activation function, supported by the universal approximation theorem. But **these functions have some problems**: activation functions such as Sigmoid or Tanh **saturate**.
For value not close to zero the gradient of the function is very close to zero. The issue lay in the fact that this means that **starting from a wrong initialization, i.e. with a neuron working in that saturated region, will take a lot of time to move along the gradient to move on the other side and correct the weights**. Furthermore **backpropagation error gradient** has the derivative of all the activation function through all the layer that you are traversing g' and h': So as seen using the chain rule, **backpropagation requires gradient multiplications that imply another problem**: **multiplying number smaller than 1 the result may become very very small**. In particular looking at the derivative of the hyperbolic tangent and the one of the sigmoid the maximum of the derivative is in the origin and is less than 1. **More layers we add more multiplications of number smaller than 1 backpropagation has to compute, so it means that faraway from the output the gradient becomes zero**. This is called **vanishing gradient**: is the effect of **multiplying many times number less than one obtaining a gradient that is basically zero**. **Learning in deep networks does not happen since due to the vanishing gradient** for weights faraway from output the $\frac{\partial E}{\partial w}$ is basically zero so **the weights are not updated**. Hence **since you start from a random initialization the earlier input are not corrected**.

NOTE to get an intuitive estimation of how the order of magnitude of the gradient of the error decrease for each layer we can consider a multiplication of 0.1 (e.g

for 10 layers is 10^{-10} which means 0). So we can notice how even with more than 5 layers we cannot train a neural network that uses sigmoid or tanh.

NOTE even by playing with the learning rate, i.e. changing it for each layer to compensate the very small error gradient, does not solve the problem also because the learning rate should be adapted online with the error gradient.

This is a well known problem in Recurrent Neural Networks (we will see it), but it affects also deep networks, and it has always hindered neural network training.

The **solution to the problem** is to **change activation function**.
The **rectified linear unit** (**ReLU**) has been **introduced to solve the problem of the vanishing gradient**. It is a **non-linear function** that can be expressed as:

$$g(a) = ReLU(a) = max(0, a)$$

$$g'(a) = \begin{cases} 1 & a \leq 0 \\ 0 & a < 0 \end{cases}$$

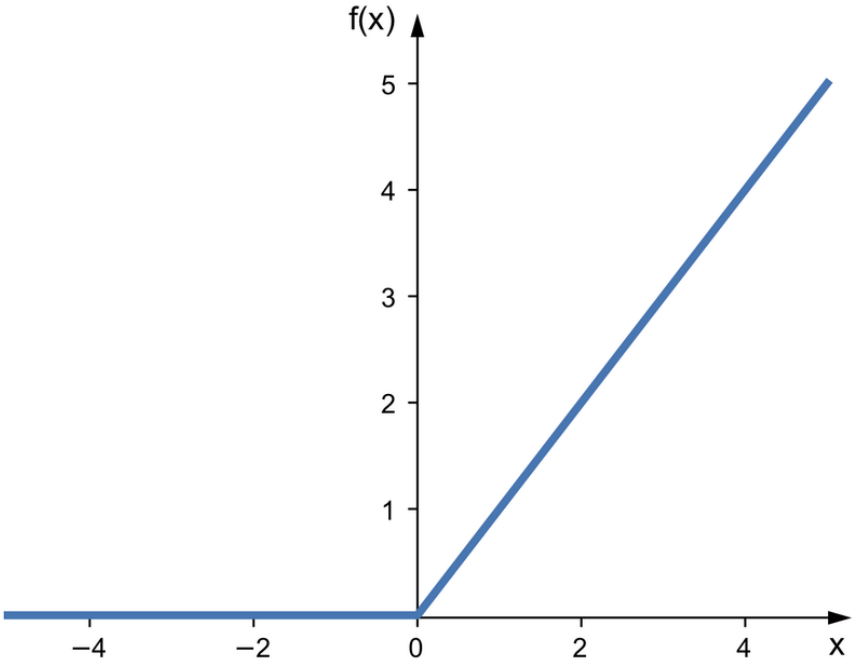


Figure 12: Rectified linear unit (ReLU)

The gradient equal to 1 is very nice since multiplying for 1 in the back-propagation multiplication chain does not change the value, so the learning

can go through the beginning of the network and basically we can have the gradient pass through even larger networks. So nowadays we use ReLU in hidden layers since: **it is easy to train and it pass thorough many layers**.

The **universal approximation theorem does not work anymore with the ReLU activation function since it is not an s-function but is a piece-wise linear function**, but there are **other theorems that proof that even with ReLU function the neural network can approximate anything**.

ReLU **solve the problem of vanishing gradient** and **several advantages**:

- **Efficient gradient propagation** (**no vanishing or exploding problems**) and efficient computation (just thresholding at zero to distinguish the function piece). This means respectively that **training works better and faster**. So using ReLU, as said, we are **able to train** bigger and **bigger networks** (since it does not suffer of vanishing gradient as sigmoid and tanh).
- **Faster SGD convergence** (6 times faster than sigmoid/tanh) because **computing the derivative is very easy** and can be done with an if.
- **Sparse activation** function in the sense that **only a part of the hidden units will be activated** i.e. have a positive output because **some of them will have a zero activation function that means that the neuron are switched off**: the **computation will be faster (due to turned off neurons) and the network will have less parameters** i.e. will be less flexible **helping also in generalization**.
- **Scale-invariant**: $max(0, ax) = a \cdot max(0, x)$. So you **can do computational tricks** using this numerical property.

but it has also **potential disadvantages**:

- **Non-differentiable at zero**: however it is differentiable anywhere else. So this is a problem for a mathematician but not for an engineer **since in a continuous function the probability of having exactly one point is equal to zero: if it happens I set the gradient equal to 1 or 0 with equal probability**.
- **Non-zero centered output** (differently from sigmoid and tanh), which means that ReLU **cannot be used as output if the network has to compute positive or negative values** (as it was for tanh). Again, this is not a real problem because if you want your output to be between $-\infty$ and $+\infty$ you put the linear activation function in the output layers.
- **Unbounded and can potentially blow up**. indeed **for positive numbers it is not saturating** to a certain value so the **output can be very very big especially during training with not well behaved data**.
- The **real problem** of ReLU is the **dying neurons**. **For negative input** values ReLU set the output to zero but **also the gradient is set to zero**. This means

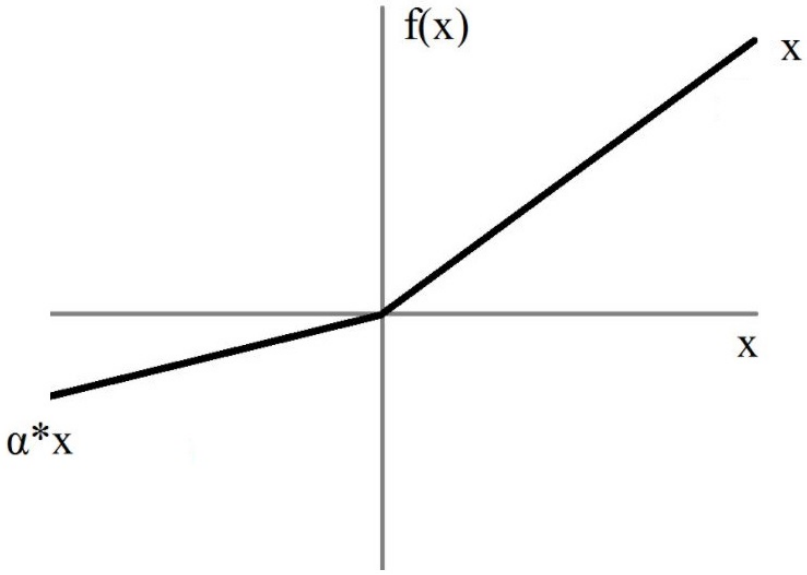
that **if by mistake is working in the negative area and you want to move the neuron somewhere else it would be impossible**. So ReLU neurons can sometimes be pushed into states in which they **become inactive** for essentially all inputs. Since **the gradient is zero in this cases, no gradient flow backward through the neuron and so the neuron becomes stuck and "dies" because they cannot be switched on**. This phenomenon **can be caused** both by networks mistakes or by a **wrong initialization** and **causes the lost of some computational power of the network i.e. dying neurons decreases model capacity**. It **happens especially with high learning rates because from the positive side of the ReLU you may end up in the negative side switching off the neuron making it die**.

NOTE there may be some problems with the gradient getting bigger and bigger since the weights are updated with very huge number, so the input of the neurons may become so high that the ReLU in a sense dies, indeed the input would be really distant from the characteristic point of the ReLU in zero. There are some techniques to avoid that, like gradient clipping due to which the gradient cannot be higher than a certain value at each step.

A **possible solution** for the **dying neuron problem** is to **ignore it**. But there are **alternative activation function defined to solve the ReLU problems**:

- **Leaky ReLU**:

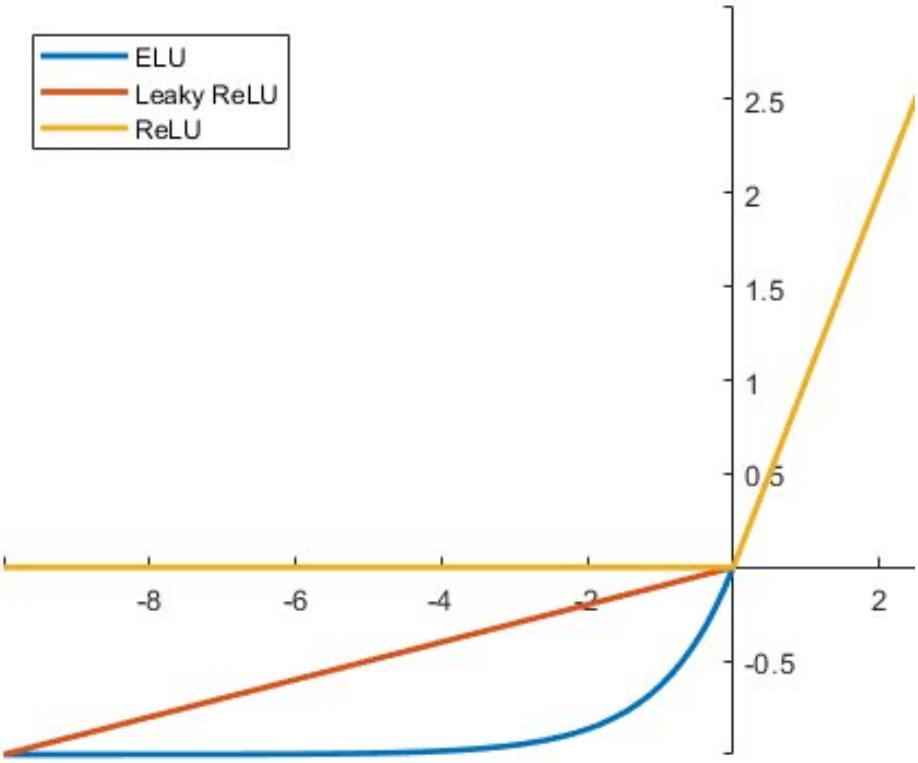
$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$



Since with an input smaller than zero the output is not zero, it **fixes the dying neuron problem because even if the input is less than zero it will need some time but it will be able get unstuck from that condition not letting the neuron die**.

- **Exponential linear unit ELU** try to **make the mean activation closer to zero which speeds up the learning**:

$$f(x) = \begin{cases} x & \text{if } x \leq 0 \\ \alpha \cdot (e^x - 1) & \text{otherwise} \end{cases}$$



where alpha is tuned hand-by-hand. In this case you are **both able to escape the dying neuron problem** and you **have also the differentiability in zero**.

NOTE **sometimes Leaky ReLU helps in performance but is only a slightly improvement with reference to standard ReLU**. But using leaky ReLU you will have **less dying neurons because you were able to rescue some neurons unnecessarily killed**.

3.8.2 Weight initialization

One of the thing that affects gradient descent is weight initialization because if you **initialize the neuron in the wrong place it will take time to change them because of the saturating derivative of sigmoid/tanh or the dying neurons for ReLU**. So the **final result of gradient descent is affected by weight initialization**(*):

- **Initializing to zero the weights is not going to work**. indeed the formula of backpropagation the weights are multiplied so all gradient would be zero and **no learning will happen**.
- **Big numbers are a bad idea because you would have big values** that are not good and **with sigmoid/tanh ending** in the wrong side **might take very long to converge since they would be in the saturated part i.e. gradient basically equal to zero**.
- Using **a Gaussian distribution with mean 0 and a very small variance** i.e. $w \sim N(0, \sigma^2 = 0.01)$ is the **classical easiest way used**. It is **good especially for small networks but might be a problem for deeper neural networks** because again in the backpropagation expression (which is the multiplication of: the weights of the links traversed by the backward pass and the derivative of the activation functions of the neurons traversed by the backward pass) we **managed to fix the issue of the vanishing gradient due to the activation function but if we have weights close to zero we would end up in obtaining the vanishing gradient due to the weights**.

Working with **deep networks** we should **have different ways of initializing the weights**:

- **If weights start too small, then gradient shrinks** as it passes through each other.
- If the **weights in a network start too large, then gradient grows as it passes through each layer until it's too massive to be useful**. indeed in the backward pass every time you traverse a link you multiply for the weight of that link.

Hence **we should have weights that preserve the magnitude of the gradient**, not to have very big discontinuities between each layer i.e. **the signal that enters and exists from each layer should be more or less in the same range otherwise if you shrink or multiply every time it passes through a layer you end up destroying the gradient**(***). This is the idea of **Xavier initialization**.

3.8.3 Xavier initialization

We are trying to find the value of the weights not to end up in vanishing or exploding gradient.
How can I relate the weights initialization and the effect that this initialization has on the network? The **output of the neuron depends on the input that is amplified by the weights**. So if you don't want to make it exploding or vanishing at the beginning ruining the learning process since it would be stuck you need to **initialize the weight to a reasonable state where the amplification given by them is close to 1** i.e. the amplification of the output starting from the input is 1.

Suppose we have an input x with I components and a linear neuron (non-linear case might be more complex) with random weights w. Its output is:

$$h_j = w_{j1} \cdot x_1 + ... + w_{ji} \cdot x_i + ... + w_{jI} \cdot x_I$$

we can assume what is the range i.e. the variance (which is the variability) of the output of this neuron and what we **would like is to obtain an output not too big or not too small**. We can derive that $w_{ji} \cdot x_i$ is going to have **variance**:

$$Var(w_{ji} \cdot x_i) = E[x]^2 \cdot Var(w_{ji} \cdot x_i) + E[w_{ji}]^2 \cdot Var(x_i) + Var(w_j i) \cdot Var(x_i)$$

where Var() is the variance, E[] is the expected value and $Var(w_j i) \cdot Var(x_i)$ is the co-variance between $w_j i$ and x_i. So if we assume that the input x and the weights w can have different values **if we want to keep the output of the neuron in a fixed range we have to control the variability of** $w_{ji} \cdot x_i$. Now if our inputs and weights both have mean 0, the latter simplifies to:

$$Var(w_{ij} \cdot x_i) = Var(w_{ji}) \cdot Var(x_i)$$

If we assume all w_i and x_i are independently and identically distributed **(i.i.d.) the variability of the output of the neuron can be expressed as**:

$$Var(h_j) = Var(w_{j1} \cdot x_1 + ... + w_{ji} \cdot x_i + ... + w_{jI} \cdot x_I) = I \cdot Var(w_i) \cdot Var(x_i)$$

Hence the **variance of the output is the variance of the input but scaled by** $I \cdot Var(w_i)$. So **we don't want the weights to amplify too much the input (exploding gradient) nor to shrink the input too much (vanishing gradient)** and to do that we would like to **initialize the weights in such a way that the scaling term** $I \cdot Var(w_i)$ **is more or less 1**. If we manage to do that the gradient won't vanish nor explode and it would go through all the layers.
So **if we want the variance of the input and the output to be the same** we should have:

$$I \cdot Var(w_j) = 1$$

For this reason **Xavier proposes to initialize the weights as**:

$$w \sim N(0, \frac{1}{n_{input}})$$

i.e. sampling from a Gaussian distribution with zero mean and a variance inversely proportional to the number of inputs $n_{input} = I$ (indeed inverting the latter expression $Var(w_i) = \frac{1}{I}$).
Performing **similar reasoning** for the gradient **Glorot & Bengio** found a similar formula from the output perspective that says that **if you don't want to amplify too much the gradient especially at the beginning of the training you should initialize the weight to obtain**:

$$n_{output} \cdot Var(w_j) = 1$$

To accommodate for this they **proposed**:

$$w \sim N(0, \frac{2}{n_{in} + n_{output}})$$

These two ways are **very similar** even in numbers since the **order of magnitude of the wewight variance is the same**.

More recently **He** proposed, **for rectified linear units (ReLU)**, the following initialization:

$$w \sim N(0, \frac{2}{n_{input}})$$

that **corrects the Xavier formula multiplying the variance** σ^2 **by 2 since on average only half of the neurons are active, so consequently only half of the input would be useful** ($\frac{n_{input}}{2}$). The nice thing is that now according to the He proposal the original hypothesis that the output of a neuron is a linear combination works ($h_j = w_{j1} \cdot x_1 + ... + w_{ji} \cdot x_i + ... + w_{jI} \cdot x_I$) because for the not dead neurons this is exactly the output.

So for a small neural networks:

$$w \sim N(0, \sigma^2)$$

where the variance should be small; and for deep networks with ReLU:

$$w \sim N(0, \frac{2}{n_{input}})$$

So far we have seen how to stop overfitting, select the activation function, fine tune the hyperparameters, initialize the weights. What we didn't talk about is an **alternative training algorithm**. We have seen **Backpropagation**. **Finding weights** of a neural network is a **non-linear minimization process**:

$$\underset{w}{\operatorname{argmin}}\ E(w) = \sum_{n=1}^{N} (t_n - g(x_n, w))^2$$

We iterate from the initial configuration:

$$w^{k+1} = w^k - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w^k}$$

Then **to avoid local minimum can be used the momentum**:

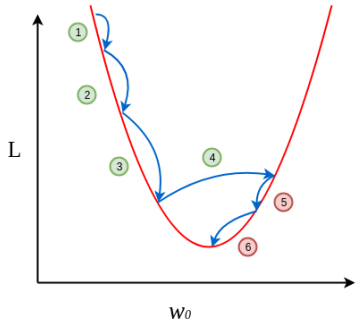
$$w^{k+1} = w^k - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w^k} - \alpha \cdot \frac{\partial E(w)}{\partial w}\Big|_{w^{k-1}}$$

There is huge literature of non-linear optimization algorithms. One of them is the **Nesterov Accelerated Gradient**: it is **similar to gradient descent with momentum but it does it in two steps**:

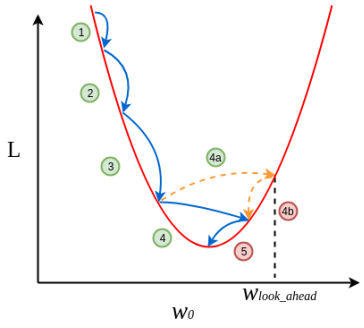
$$w^{k+\frac{1}{2}} = w^k - \alpha \cdot \frac{\partial E(w)}{\partial w}\Big|_{w^{k-1}}$$

$$w^{k+1} = w^k - \eta \cdot \frac{\partial E(w)}{\partial w}\Big|_{w^{k+\frac{1}{2}}}$$

i.e. in the **first step applies the momentum** $(w^k \to w^{k+\frac{1}{2}})$ then in the **second step it applies the gradient descent** $(w^{k+\frac{1}{2}} \to w^{k+1})$.



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

$\Rightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Positive(+)}$ $\Rightarrow \frac{\partial L}{\partial w_0} = \frac{Negative(-)}{Negative(-)}$

Usually what happens is that this algorithm **tends to converge faster**.

There are **also algorithms with adaptive learning rate**: algorithm like **Adam** (the more recent one) **keep in memory the gradient and starting from an initial guess try to adapt at each iteration the learning rate to see if the gradient is decreasing, increasing and oscillating**.
Resilient propagation algorithm doesn't care how big is the gradient and only decide in which direction to move.

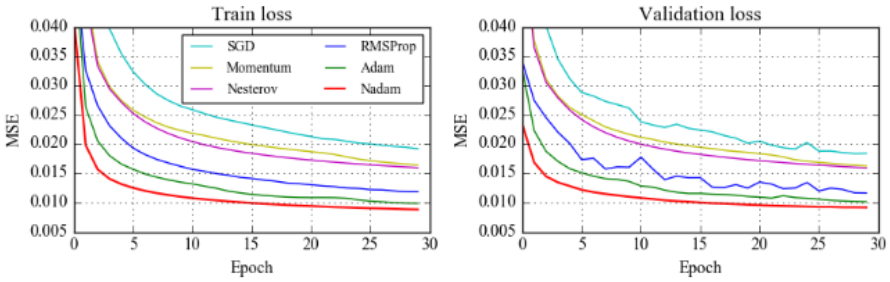


Figure 13: Comparison between different learning algorithms

As we can see Adam and NAdam are very fast (**i.e. steep loss function**) because **they adapt the step of the algorithms based on the shape of the error function**. Instead SGD, gradient descent with momentum and Nesterov are much slower.

Learning rate adjustment depends on the algorithm **some do it other not**. But there is also a **technique** called **learning rate scheduling** that consist in **scheduling a reduction of the learning rate**: if the learning rate is too high to reach the minimum causing oscillation around it a reduction help to get closer to the minimum.

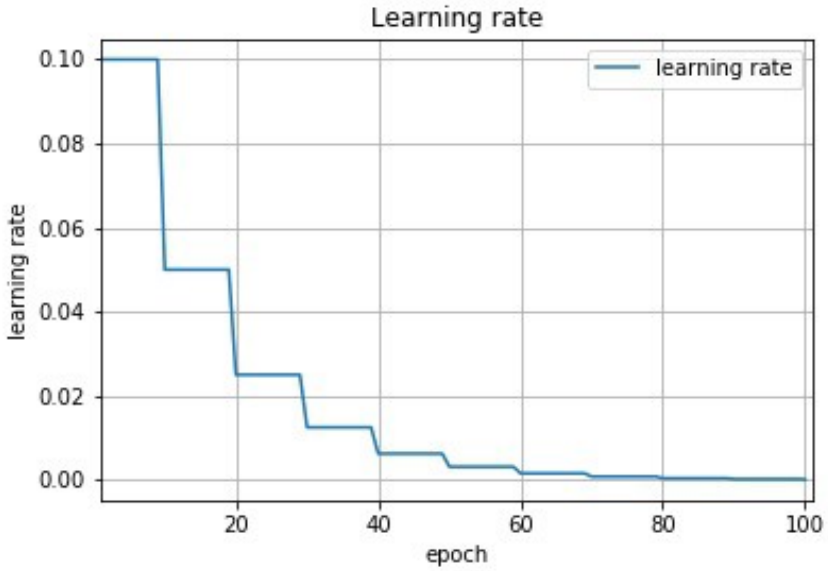


Figure 14: Learning rate scheduling

NOTE **a big learning rate with ReLU may cause an increasing in the dying neuron problem wasting more computational power**.

NOTE vanishing gradient problem is present if the gradient is less than 1 like in sigmoid/tanh. Several layers means several multiplication for the gradient that if is less than one may make the gradient vanishing to zero.

4 Convolutional Neural Networks

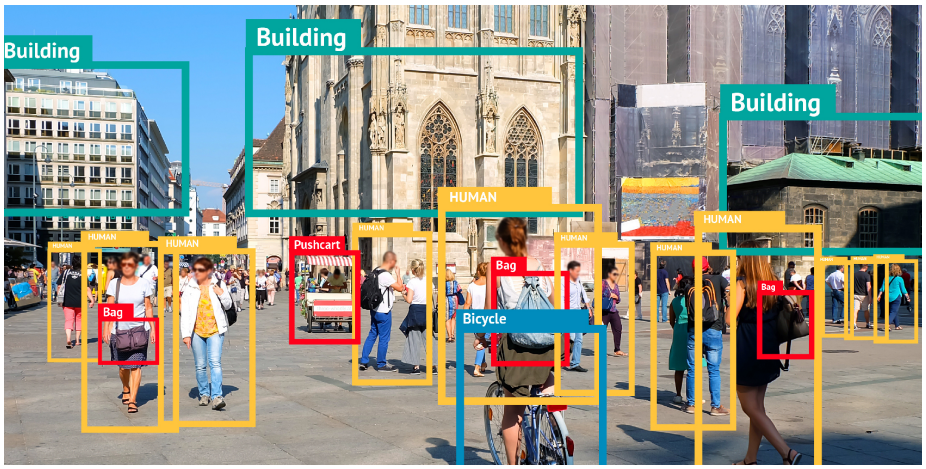
4.1 Image Classification problem

We will see basics notions for images representation and operations over images

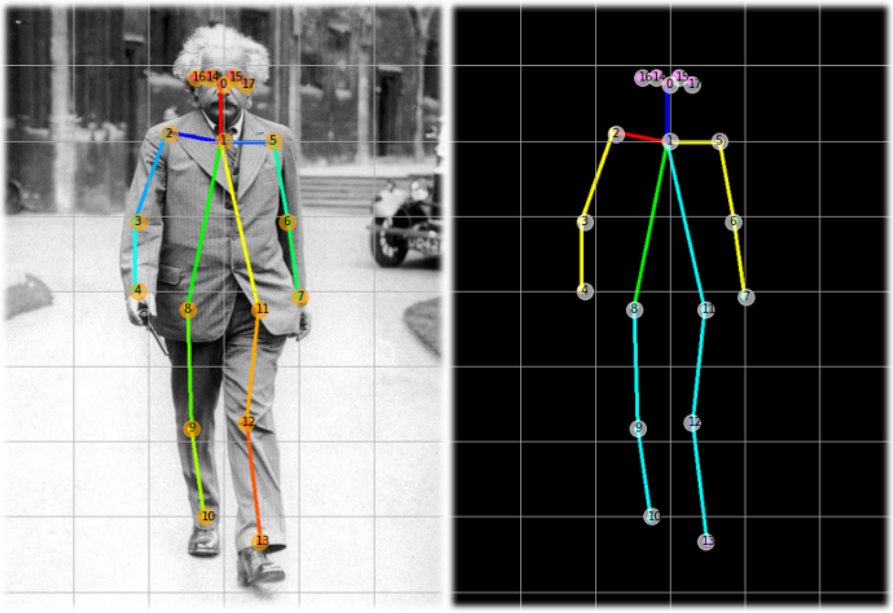
4.1.1 Image classification

Computer vision is an interdisciplinary scientific field that deals with how computers can be made to gain high level understanding from digital images or videos. There has been an increasing connection between computer vision problems and neural networks. **High level image understanding** means being able to perform tasks such as:

- Object detection: identification of multiple objects in a video frame or an image, labelling the different object it can recognize. It works in real time.



- Pose estimation: identification of the joints of the human body in a video frame or an image. It works in real time.



There are dozen of applications for this kind of task as in autonomous driving, automatic shelf analysis, quality inspections, automatic image captioning.
There has been a dramatic change in Computer Vision. Once, most of techniques and algorithms were built upon a mathematical/statistical description of images but nowadays machine-learning methods are much more popular, indeed deep learning models solve these computer vision tasks in a very successful way: differently from before, experts don't have to engineer an algorithm that knows how to solve a problem and computes the solution but an algorithm that can learn how to solve the task for you improving not only the difficulty but also the accuracy.

4.1.2 Images

Now we will see how images are represented from a computer science perspective, how is **described as data** since it will be the **input of how model**.
An RGB image is stored in a file as a 3 dimensional matrix where each cell is a pixel:

$$\mathcal{I} \in \mathcal{R}^{RxCx3}$$

where **R is the number of rows, C is the number of columns and 3 is the number of colour channels**. So we have a matrix for each of the three colour channels:

$$\mathcal{R} \in \mathcal{R}^{RxC}$$
$$\mathcal{G} \in \mathcal{R}^{RxC}$$
$$\mathcal{B} \in \mathcal{R}^{RxC}$$

Each cell of the matrix relative to a colour channel stores the correspondent **colour information** in **8 bits** (**i.e. 1 byte**). So images are rescaled and casted in [0, 255] where **0 is black and 255 is white**. By **considering a colour channel** the **value of the cell tells** the **intensity of that colour** is that pixel: from 0 if the colour is not present at all to 255 if is the maximum intensity. To create the RGB images we have to combine the three RGB channels: by combining the three colorwise matrices the RGB image matrix cell contains a triplet of bytes for the R, G and B intensity values of the pixel. For example: for white all three channels are close to 255; for black all three channels are close to 0; for gray all three channels have the same intensity.
Gray-scale images have only one value, that tells the intensity of the colour from black to white.

In some cases also multi spectral images are used. In these cases each channel contain a different part of the light spectrum and can be used for deeper analysis.

4.1.3 Videos

Videos are **sequences of images** (frames). So if a frame is $\mathcal{I} \in \mathcal{R}^{RxCx3}$ a **video of T frames** is:

$$\mathcal{V} \in \mathcal{R}^{RxCx3xT}$$

so **we have to add the temporal dimension T**.
Dimensions increases very quickly. Without compression 1 byte per color per pixel means that:

- 1 frame in full HD has $R = 1080$, $C = 1920$ which means $R * C * 3 \approx 6MB$
- 1 second in full HD at 24fps $\approx 150MB$

Fortunately, visual data are very redundant, thus compressible so for example when we save a video on our phones we don't save the raw data of each frame but a compressed file that has a much smaller dimension, encoded for example in mpeg. This has to be taken into account when you design a Machine learning algorithm to be used on images: **the compressed image files must be opened and fed to the network in the matrix form** we have seen. So to do this is **required a huge amount of memory for processing and training a network** and for that **reason** most of deep learning process have been **developed over small images** and not the one we are used to: we don't feed to a neural network one second of video in full HD, it is too much. Furthermore, bearing in mind that an **image dimension is not equal to the file size due to compression**, images dimension would even **increase during processing requiring a lot of memory**.

4.1.4 Local (Spatial) Transformations

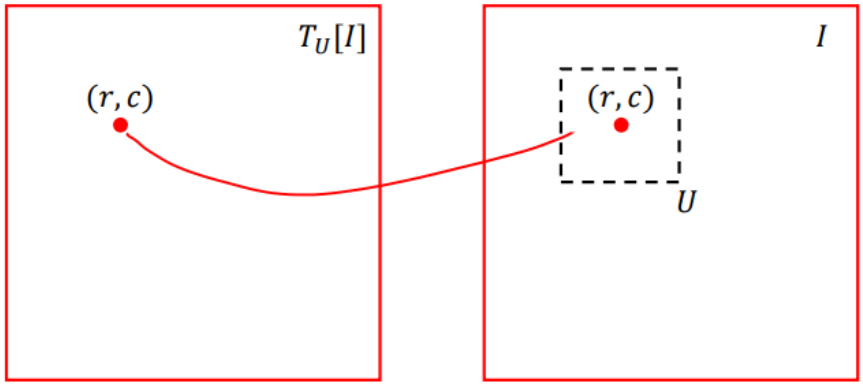
The **most important image processing operations for classification problems**. In general, local (spatial) transformation can be written as the **transformation T**

that take an image I and transform it in an image G, so **for each pixel (r,c) of I**:

$$G(r,c) = T_U[I(r,c)]$$

Where:

- **I is the input image transformed**
- **G is the output image**
- $T_U : \mathcal{R}^3 \longrightarrow \mathcal{R}^3$ or $T_U : \mathcal{R}^3 \longrightarrow \mathcal{R}$ is the **transformation function**.
- **U is a neighborhood** that identifies a **region on the image** i.e. **a set of pixels I(x,y) around the considered pixel I(r,c) that will concur in the output definition i.e. the transformation T won't consider only the pixel in row r and column c, I(r,c), but T operates on I inside U**: the output at pixel (r,c) i.e. $T_U[I(r,c)]$ is defined by all pixels $\{I(r+x, r+y)|(x,y) \in U\}$ (or $\{I(x,y)|(x-r, y-c) \in U\}$) where for example $U = \{(-1,-1),(0,-1),(+1,-1),(-1,0),(0,0),(+1,0),(-1,+1),(0,+1),(+1,+1)\}$.



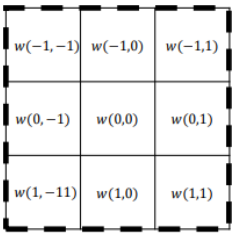
The **transformation T can be either linear or non-linear** and is **repeated for each and every pixel in the input image**. Applying a linear transformation, **linearity implies that**:

$$T(I(r,c)) = \sum_{(x,y)\in U} w_i \cdot I(r+x, c+y)$$

considering some weights $\{w_i\}$. In general the **set of weights can be considered as an image or a filter h that entirely defines this operation** i.e. the **weights** can be reorganized **as the filter matrix which is as big as the neighborhood U**:

$$T(I(r,c)) = \sum_{(x,y)\in U} w_{(x,y)} \cdot I(r+x, c+y)$$

The filter weights, in general, **may be different for each pixel (i.e. changing depending on r and c) and also for each cell (x, y) of the filter**. We **usually fix** the weights **of the neighborhood U** and apply them to each every pixel in the input image (**i.e. they only depends on (x, y) and not on (r, c)**):



that applied according to the latter formula is:

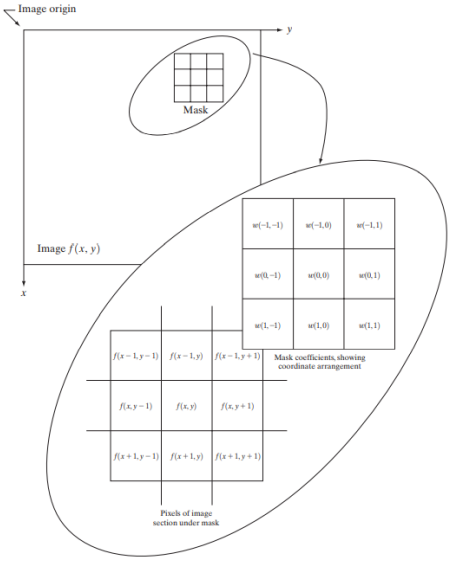


Figure 15: Image filtering through a 3x3 filter

We will see **neural networks that learn the filters**.

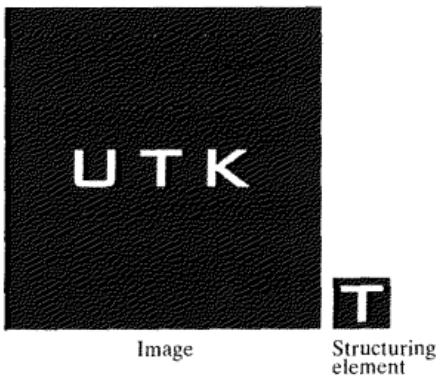
4.1.5 Correlation

The **correlation among a filter h and an image is defined as**:

$$(I(r,c) \otimes h) = \sum_{u=-L}^{L} \sum_{v=-L}^{L} w(u,v) \cdot I(r+u, c+v)$$

where the filter w is of size $(2L+1)x(2L+1)$. Hence the correlation for a pixel I(r,c) is a linear combination of all the pixels around it covered by the filter w. Must

be noted that **the complete correlation between the entire image I and the filter has the same size of the image I**. This operation is easy to understand with binary images i.e. black and white images where the pixel value is 0 if it is black and 1 if it is white.



The **resulting image will be of the same size of I** and the value of each pixel will be the number of overlapping white pixels between the image and the filter: since the black pixel are zero they do not give any contribution to the sum, only the correspondence between two white pixels add 1 to the sum over all the filter. **The pixel where the resulting images will have the maximum value would be the one that matches the most the white part of the filter** (in the latter example the middle of the T pattern since the filter contains the same T pattern) so it is a **kind of pattern detection**: **as the value of the pixel increase higher the correlation with the filters is** i.e. more similar is the neighborhood U of the pixel to the filter, and will be maximum if the pattern matches 1:1. We must notice that **inverting the images (0→1 and 1→0) the result changes** since we will have white areas in places where the pattern is not matching due to the white background, so **normalization is needed to prevent this problems**(*) (we will discuss on how to perform template matching). **Higher correlation means higher value in the pixel**.

4.1.6 Problems in image understanding

There are **several task** that we will face concerning the image processing:

- **Image Classification**: feeding an image to the network the output is a label i.e. categorical value that typically correspond to the content of the image. Can also be given together with the label the probability of correctness of the latter. **Assign to an input image $I \in \mathcal{R}^{RxCx3}$ a label l from a fixed set of categories**:

$$\Lambda = \{"wheel", "car", ..., "castle", "baboon"\}$$

$$I \Rightarrow l \in \Lambda$$

- **Localization**: a network doing localization **in the simplest form provide the label with 4 real numbers (x, y, w, h) which correspond to the coordinates of the bounding box containing the labeled object**: **x and y are the coordinates of the top left corner of the box, w is the width and h is the height**. **Assign** to an input image $I \in \mathcal{R}^{RxCx3}$ **a label** l **from a fixed set of categories**:

$$\Lambda = \{"wheel", "car", ..., "castle", "baboon"\}$$

and the coordinates (x, y, w, h) of the bounding box enclosing the object:

$$I \Rightarrow (\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \mathbf{l})$$

- **Object detection**: similarly to localization **returns both the bounding box and the label of the objects but is different from localization from a mathematical point of view since it does not return a single tuple but a variety of tuple depending on the image content**. Assign to an input image $I \in \mathcal{R}^{RxCx3}$ **multiple labels** $\{l_i\}$ from a fixed set of categories:

$$\Lambda = \{"wheel", "car", ..., "castle", "baboon"\}$$

each corresponding to an instance of that object and the **coordinates (x, y, w, h) of the bounding box** enclosing the **each object**:

$$I \Rightarrow \{(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \mathbf{l})_\mathbf{1}, ..., (\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \mathbf{l})_\mathbf{N}\}$$

with N the number of objects detected.

- **Segmentation**: each of the pixel is classified depending to the class the object it belongs to, so the output is a kind of mask . **Assign to each pixel of an input image** $I \in \mathcal{R}^{RxCx3}$ **a label** $\{l_i\}$ **from a fixed set of categories**:

$$\Lambda = \{"wheel", "car", ..., "castle", "baboon"\}$$

$$I \Rightarrow S \in \Lambda^{RxC}$$

where $S(x, y) \in \Lambda$ denotes the class associated to the pixel (x, y) and pixel **without a label are associated to the background**. Hence the **output of segmentation is another image whose pixel values are labels instead of intensity values**.

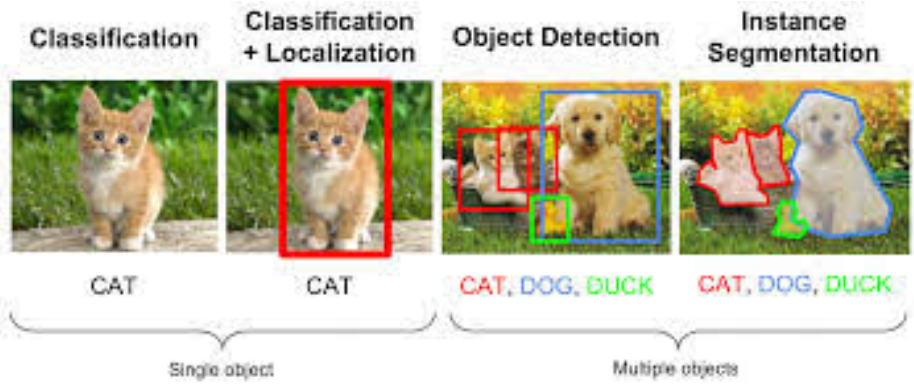


Figure 16: Image processing tasks.

The latter techniques can be mixed in **Instance segmentation**: separates like in object detection the objects detected, providing the bounding box and the label and on top of that a mask inside each bounding box that tell which are the pixels belonging to that class. Assign to an input image $I \in \mathcal{R}^{RxCx3}$ multiple labels $\{l_i\}$ from a fixed set of categories:

$$\Lambda = \{"wheel", "car", ..., "castle", "baboon"\}$$

each corresponding to an instance of that object and the coordinates $\{(x, y, w, h)_i\}$ of the bounding box enclosing the each object **and the set of pixels S in each bounding box corresponding to that label**:

$$I \Rightarrow \{(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \mathbf{l}, \mathbf{S})_\mathbf{1}, ..., (\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \mathbf{l}, \mathbf{S})_\mathbf{N}\}$$

Figure 17: Instance segmentation.

4.1.7 Is image classification a challenging problem?

Image classification is a challenging problem.

(i) The **first challenge is dimensionality**: **images are very high dimensional data**. For example the CIFAR-10 dataset is composed by 32x32 images i.e. extremely small images but high-demensional since $dim = 32 \cdot 32 \cdot 3 = 3072$. Looking to them as a vector i.e. as a possible input to a neural network we have seen so far, then the input layer should have 3072 neurons and if we even add some hidden layer the number of parameters start to increase terribly. **Bear in mind how large an image is (in terms of Bytes)** when you'll be implementing your CNN: **the whole batch and the corresponding activation have to be stored in memory**.

(ii) The **second challenge is the label ambiguity**: there might not be a label that uniquely identify the image. In an image may be a lot of ambiguity:

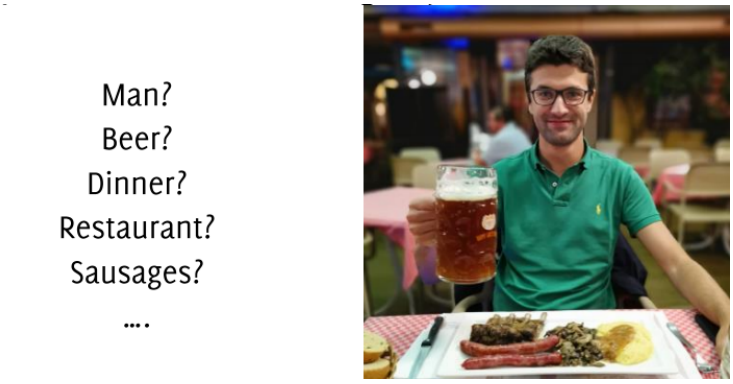


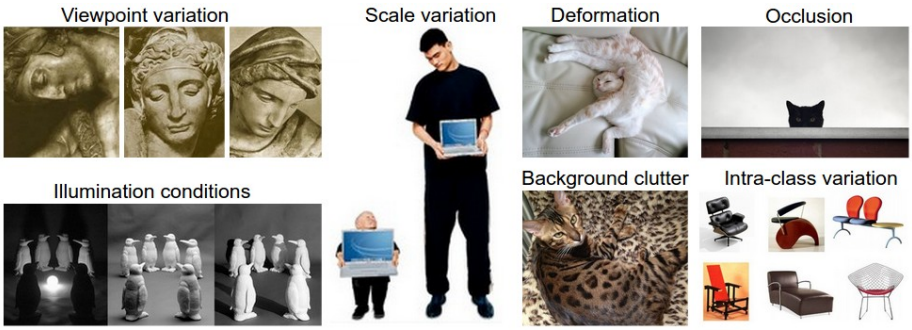
Figure 18: Label ambiguity in image classification.

and maybe the classifier consider various of this label but at the end it has to choose a single one. Indeed most of classification algorithm output the most likely 5 (for example) outputs and the image is considered correctly classified if the true label is among this 5.

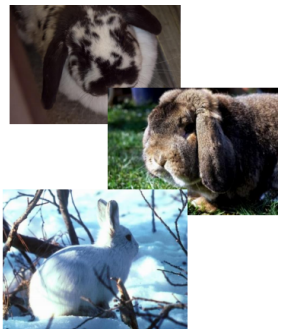
(iii) The **third challenge are the transformations**: there are many transformations **that change the image dramatically, while not its label**. For example:

- Changing the **illumination conditions** we want the label to be always the same even considering the fact that the **input of the classifier would be drastically different**.

- **Deformations**: the content can appear in different position and shape.
- **Changing the view point**
- **Occlusion**
- Background clutter
- **Scale variation**



(iv) The **fourth challenge is the intra-class variability**: images in the **same class might be dramatically different since they don't share many features** (e.g. chairs: some have four legs other don't)



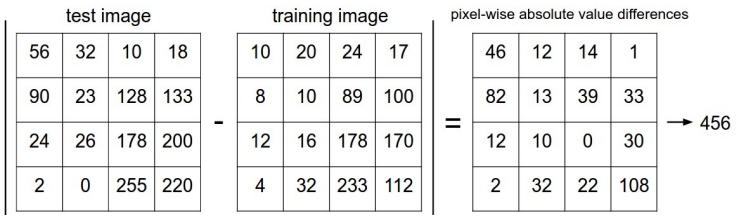
(v) The **fifth problem is the perceptual similarity**: perceptual similarity in images is not related to pixel-similarity. The simplest classification you can think off assign to each test image the label of the closest image in the training set (nearest neighbour):

$$\hat{y}_j = y_{j^*} \qquad j^* = \underset{i=1,\ldots,N}{\operatorname{argmin}}\, d(x_j, x_i)$$

One of the simplest possibilities is to compare the images pixel by pixel and add up all the differences. In other words, given two images and representing them as vectors I_1,I_2 , a reasonable choice for comparing them might be the L1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

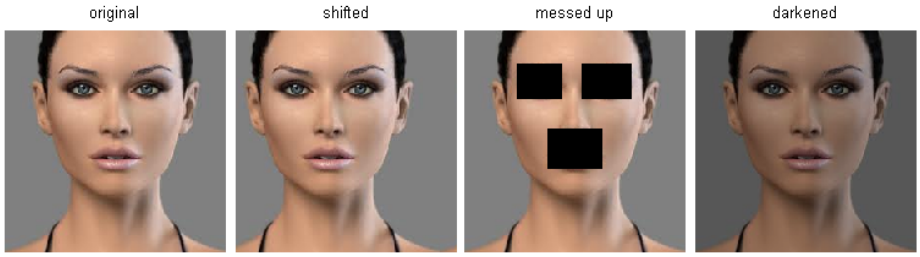
Where the sum is taken over all pixels. Here is the procedure visualized:



There are many other ways of computing distances between vectors. Another common choice could be to instead use the L2 distance, which has the geometric interpretation of computing the euclidean distance between two vectors. The distance takes the form:

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

In other words we would be computing the pixelwise difference as before, but this time we square all of them, add them up and finally take the square root. **Using L1 norm or L2 norm, we fail the task since we get a distance measure that is pointless in images**. Indeed looking at the following image where an original image (left) and three other images next to it that are all equally far away from it based on L2 pixel distance, **clearly, the pixel-wise distance does not correspond at all to perceptual or semantic similarity** since we can clearly distinguish the fact that three different transformation have been applied.



Using a data visualization technique called t-SNE, that places the data of a dataset nearby if they are closer according to a given distance, using L1 norm or L2 norm to measure the distances we can easily spot the fact that comparing images with the pixel differences i.e. **handling images vector as you would with traditional vectors is not appropriate** (for example in the cifrar set we obtain that a frog is more similar to a truck than another frog) **because what we are interested in is the perceptual content of the image but what we are measuring is a distance in two vectors that does not correlate with that**. Hence **perceptual similarity is not related to pixel similarity.**

4.1.8 Linear Classifier

Let's analyze for example a simple linear classifier for images that has only one layer of neurons, without any hidden layer, and that classify over 3 categories. A classifier can be seen as a function that maps an image x to a confidence scores for each of the L class:

$$\mathcal{K} : \mathcal{R}^d \longrightarrow \mathcal{R}^L$$

where d is the number of pixels of the image, $\mathcal{K}(x)$ is a L-dimensional vector and its $i-th$ component contains a score of how likely x belongs to class i. Intuitively, a good classifier associates to the correct class a score that is larger than associated to incorrect classes. In linear classification $\mathcal{K}$ is a linear function:

$$\mathcal{K}(x) = W \cdot x + b$$

where $W \in \mathcal{R}^{Lxd}$ are the weights, $b \in \mathcal{R}^L$ is the bias i.e. are the parameters of the classifier $\mathcal{K}$. Hence the classifier can be represented as a dense neural network that has L linear (i.e. linear activation function, it does not give a probability since it is not the softmax) neurons i.e. a number of neurons equal to the number of classes it has to recognize and d-dimensional input.

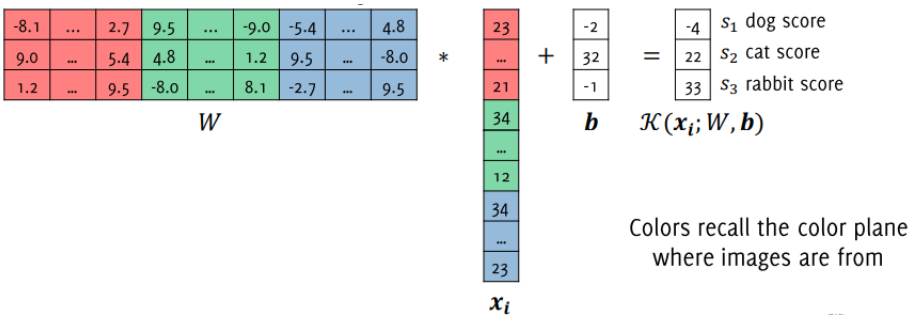


Figure 19: Caption

The input of the network is a image unrolled column-wise (column 0, column 1,...) for each color channel as in the latter picture. The classifier assign to an input image the class corresponding to the largest score:

$$\hat{y}_j = \underset{i=1,\ldots,L}{\operatorname{argmax}} [s_j]_i$$

being $[s_j]_i$ the $i-th$ component of the vector:

$$\mathcal{K}(x_j) = W \cdot x_j + b$$

The score of a class is the weighted sum of all the image pixels. Weights are actually the classifier parameters. Weights indicate which are the most important pixels/colors i.e. the one that count the most in the weighted sum.

Given a training set TR and a loss function, define the parameters that minimize the loss function over the whole TR. In case of a linear classifier:

$$[W, b] = \underset{W \in \mathcal{R}^{Lxd}, b \in \mathcal{R}^L}{\operatorname{argmin}} \sum_{(x_i, y_i) \in TR} \mathcal{L}(x_i, y_i)$$

Solving this minimization problem provides the wights of our classifier. The loss function $\mathcal{L}$ measures our unhappiness with the score assigned to training images: the loss will be high on a training image that is not correctly classified, low otherwise. The loss function can be minimized by specific algorithms (e.g. gradient descent). The loss function has to be typically regularized to achieve a unique solution satisfying some desired property:

$$[W, b] = \underset{W \in \mathcal{R}^{Lxd}, b \in \mathcal{R}^L}{\operatorname{argmin}} \sum_{(x_i, y_i) \in TR} \mathcal{L}(x, y_i) + \lambda \cdot \mathcal{R}(W, b)$$

being $\lambda > 0$ a parameter balancing the regularization term and the fitting term. Once trained the classifier is expected to provide to the correct class a score that is larger than that assigned to the incorrect classes. The training data is used to learn the parameters W, b and once the training is completed it is possible to discard the training set and only keep the learned parameters. $W(i, :)$ is a d-dimensional vector

containing the weights of the score function for the $i-th$ class, so the computation of the score function for the $i-th$ class corresponds to compute the inner product:

$$W(i,:) \cdot x$$

thus, linear classifiers identify an hyperplane in a d-dimensional space i.e. a line in a 2-dimensional space $w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$. Each image can be interpreted as a point in $\mathcal{R}^d$ whose coordinates are represented by the vector we obtain by unrolling it colour-wise and column-wise. Each classifier is a weighted sum of pixels which correspond to a linear function in $\mathcal{R}^d$.

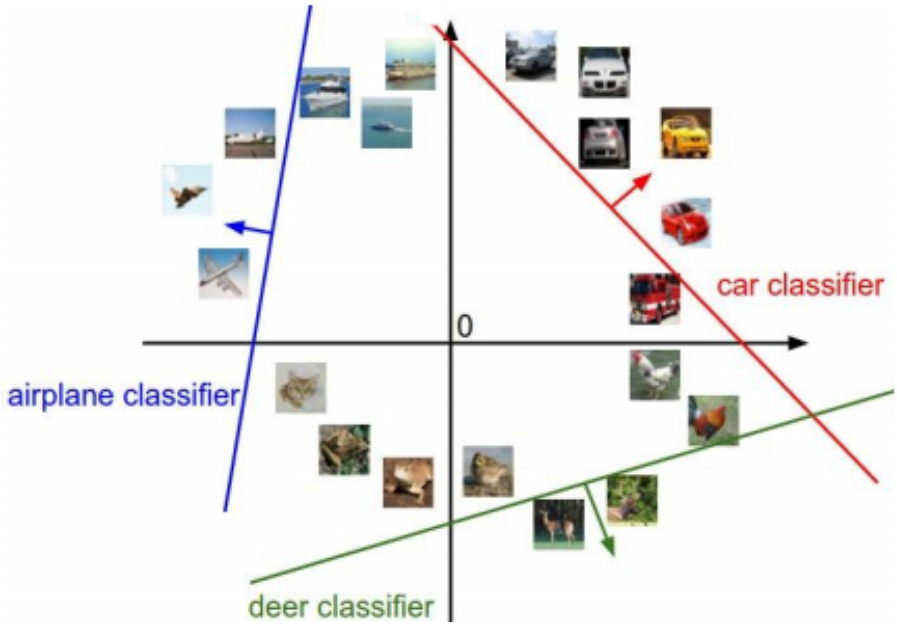
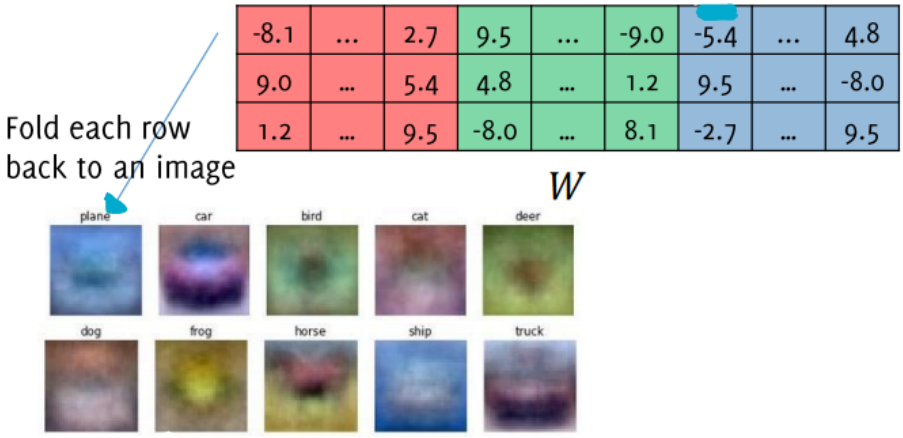


Figure 20: Caption

So each hyperplane separates the samples belonging to a certain class from all the others: if the image is on the hyperplane the score is 0, and the further from the hyperplane on the right side higher is the score. So from the output, since we take the class for which the network produced the highest score, we take the class for which the hyperplane is the furthest. **From the picture above we can see how $W(i,:)$, i.e. the i-th row of W, is multiplied for the input to generate (adding the bias) the i-th output. But since the input is an unrolled (column-wise and channel-wise) image $W(i,:)$ can be reshaped (column-wise and channel-wise) as an image as well and can be seen (after some normalization due to its value outside [0, 255] range) as an image template used in matching, learned to match at best images belonging to the $i-th$ class**.



Furthermore **since $W(i,:)$ can be seen as a template used in matching, the operation done by the linear classifier $W(i,:) \cdot x$ is equal to the correlation between the input image and the weights image that is the filter for the center pixel of the image** (since W and the image have the same dimension and the **output is a scalar**) **for each colour channel then summed up**:

$$W(i,:) \cdot x \equiv \sum_{(x,y) \in ImageCoordinates} W(x,y) \cdot I(x,y)$$

We have seen the correlation for a grayscale image, but for **RGB** pictures is very similar: you should **apply the correlation colour channel-wise** i.e. correlation channel per channel and then sum them up:

$$R_W \otimes R_I + G_W \otimes G_I + B_W \otimes B_I =$$

$$= \sum R_W(x,y) \cdot R_I(x,y) + \sum G_W(x,y) \cdot G_I(x,y) + \sum B_W(x,y) \cdot B_I(x,y)$$

being R, G, B respectively the red, green and blue the colour channels. As we have seen for the binary image T pattern filter the maximum correlation is given when the image and the filter are identical. In the following images for example the correlation is higher between the images on the right since the image to match (the on the left in both pairs) is more similar to the weights image (the on the right in both pairs) that has high values in the blue channel because most of the pixel have a colour similar to blue:

What has the classifier learned? Due to the fact that what **is performing is correlation** it learned that:

- That the background of a bird and a frog is green i.e. the weights are much higher in the green section giving more importance to that colour; that the background of a plane and a boat is blue i.e. the weights are much higher in the blue section giving more importance to that colour. **Indeed the correlation is very high when the filter (W) and the image are very similar**.
- Cars are typically red, because probably in the training set the image labeled as cars contained a red car.
- Horses have two heads because in the dataset the horses in the images were looking right and left.

Hence **what a simple classifier, i.e. a simple neural network, would do is template matching** and the **learning would be to estimate the template that provides the highest score to the considered class** and the lowest score to the other wrong class. This can be **interpreted as** that a simple neural network would perform **a visual matching** and as we have seen there are many **problems for image classification that won't be solved with this simple approach even considering a network with many hidden layers, indeed the number of parameters increase quickly impeding the training**.

The **model was definitively too simple** and data were not enough for achieving higher performance and better templates. However:

- Linear classifiers are among the most important layer of NN.
- **Such a simple model can be interpreted** (with more sophisticated models you typically can't)

But there should be a **better way for handling images**.

4.2 Convolutional Neural Network

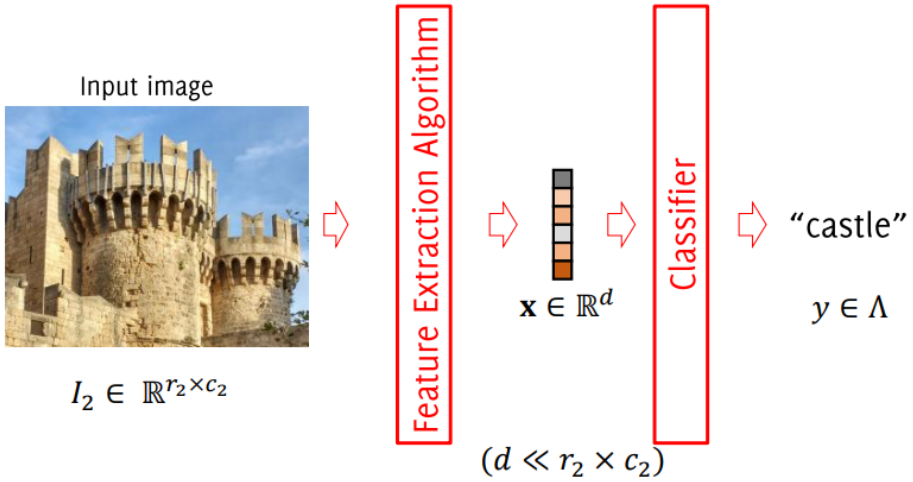
We have seen that a classifier can learn to match an image i.e. **learn major features like the background colour but** it **won't detect the most relevant pattern inside the image**. This is very useful to understand because it will be what a **Convolutional Neural Network** will do at the beginning: by designing a **different architecture** by **stacking multiple layers capable of correlation** we can design neural networks capable of highlight small or increasingly large pattern inside an image that can make a real distinction between what the objects really are, hence **not only macroscopic features from the background but** other **important pattern discriminating for the objects**.

4.2.1 Feature Extraction

The **approach** to **classify** the images mainly pushed by the literature was to **perform feature extraction**. **Images can not be directly fed to a classifier and we need some intermediate step to**:

- **Extract meaningful information** (to our understanding)
- **Reduce data-dimension**

Hence we need to **extract features: the better the features the better the classifier**.

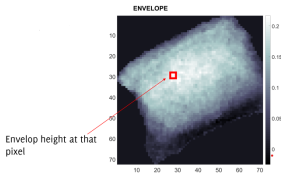


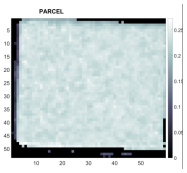
That is a **pipeline** of algorithms in which **the feature extraction algorithm get as input the image, process it and then outputs a vector containing the meaningful information for the classification task**. The objective of the feature extraction algorithm is to transform an image, difficult to handle for the various challenges we have seen, into a **standard vector whose dimension is much smaller than the number of pixel of the image** ($d << r_2 x c_2$). Then the obtained vector that **synthesize all the features of the image** can be used as an input for a classifier that can be for example a neural network. Changing the image the feature extraction algorithm output would be a different vector.
Features can be, and were in the past, hand crafted, indeed:

- **Engineers know what's meaningful in an image** (e.g. a specific color/shape, the area, the size)
- **Engineers can implement algorithms to map these information in a feature vector**.

For example in heartbeat morphology doctors know which patterns are meaningful for classifying each beat i.e. for a diagnosis. Let's consider a parcel classifier that

works with RGB-D images in which there are not colour information but only depth measures. It has to distinguish 3 classes of input: parcels, envelope or double i.e. multiple objects in the pictures that must be separated and analyzed separately. In this situation examples of features are:





- Average height
- Area (coverage with non-zero measurements)
- Distribution of heights
- Perimeter
- Diagonals

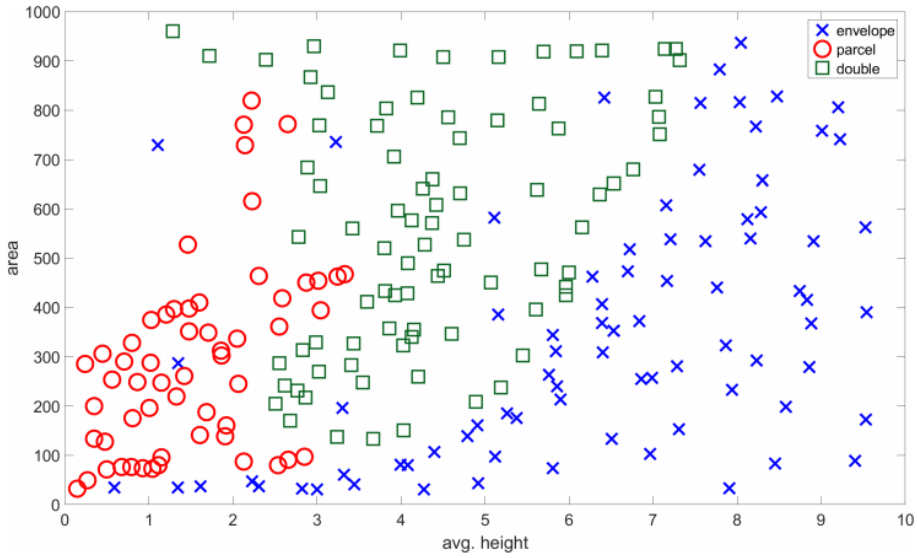
So the feature extraction algorithm would compute all of this features, put them in a vector (of dimension 5, equal to the number of extracted features), then the classifier will figure out the class of the features and so of the starting image. The training set is a set of annotated examples:

$$TR = \{(I, l)_i, i = 1, ..., N\}$$

Each couple $(I, l)_i$ corresponds to:

- an image I_i
- the corresponding label l_i

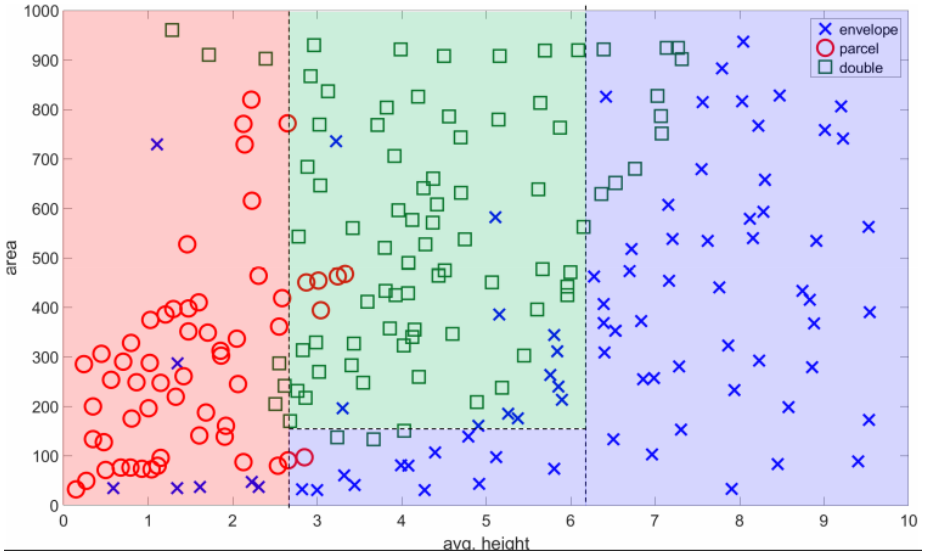
From the image of the training set can be extracted features and then plotted to understand how they are distributed. For example plotting the area and the average height we obtain:



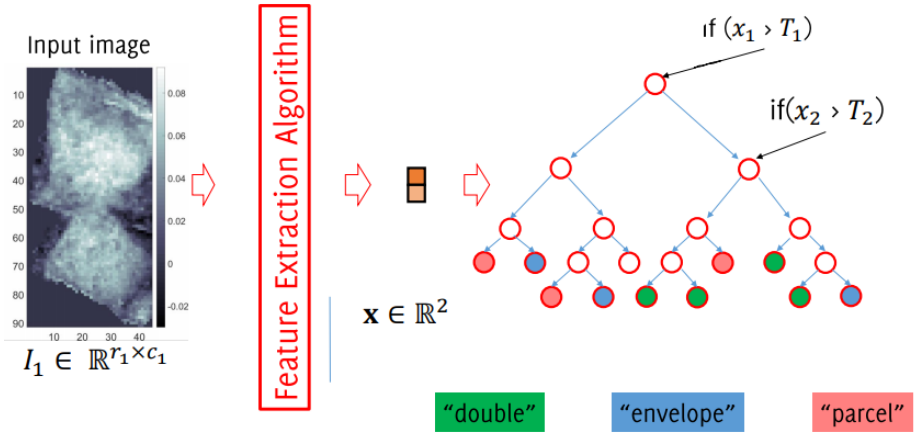
From this plot can be manually defined some rules that, after the features extraction can tell to which class the image belongs to concerning the two plotted features:

- If average height $<$ 2.5 then l $=$ "parcel"
- If average height $>$ 6.2 then l $=$ "envelope"
- If 3.5 $<$ average height ¡ 6.2 && area $>$ 200 then l $=$ "double"
- If 3.5 $<$ average height $<$ 6.2 && area $<$ 200 then l $=$ "envelope"

In this way the classifier output would be:



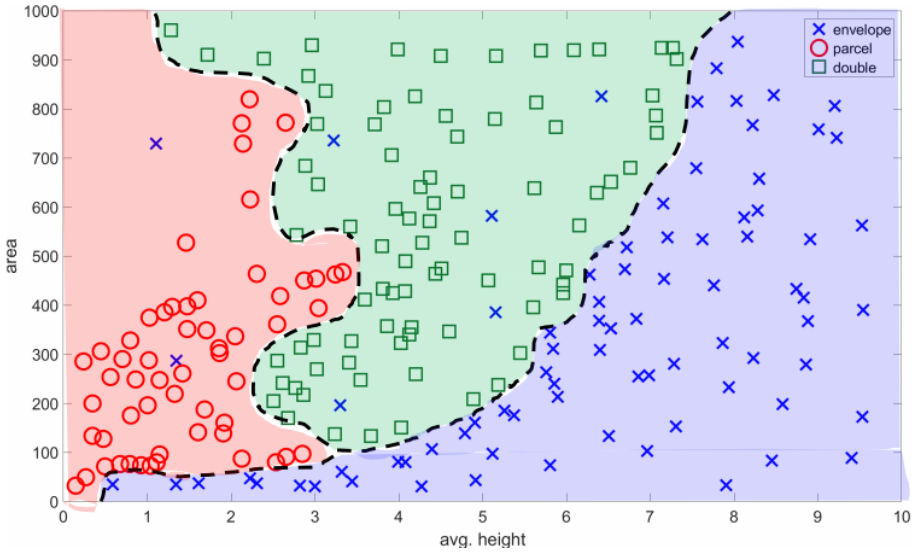
so the classifier divided the feature space in 3 regions: depending on where the input will fall the image will be classified. This type of classification of the image, from its features, correspond to a tree classifying image features:



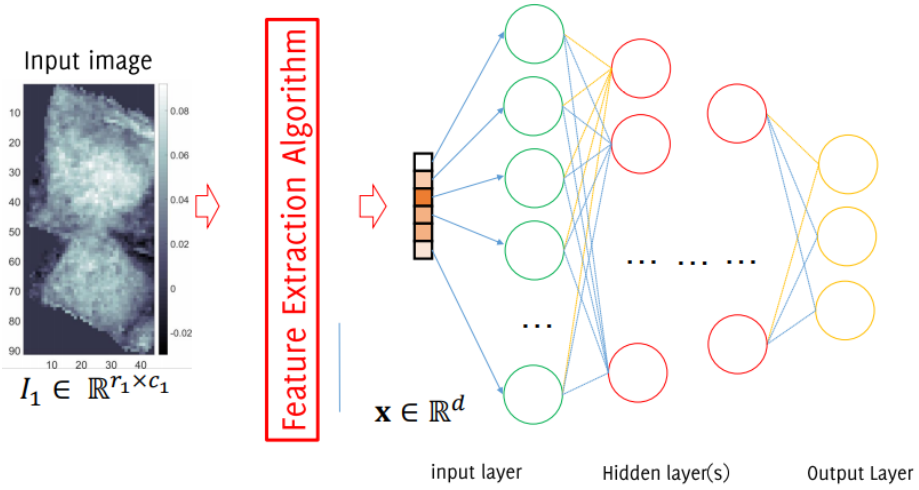
This classification tree can be manually designed or obtained from a neural network. Of course the separation line are difficult to design since the in way we have defined it we do not obtain the best result.

Furthermore, is very complex to design more complicated non-linear relation and increasing the number of variables (i.e. features) becomes more complex to classify the input. **So it is very difficult to grasp what are meaningful dependencies over multiple variables and is also impossible to visualize these**. For this

reason we resort to a data-driven model (e.g. neural network) for the only task of separating feature vectors in different classes. A neural network **provide a non-linear separation of the boundaries among classes**.



Hence the best method for image classification is to combine the feature extraction algorithm with a neural network that classify the image given its features. The **input layer will have a number of neuron equal to the size of the feature vector** and the **output layer the same size of the as the number of classes** such that, **using softmax as the output neurons activation function, each output neurons output the probability of the input image to belong to the correspondent class**. Hidden layers instead can be an arbitrary number.



Hand crafted features have several **pros**:

- **Exploit a priori/expert information** and you don't have to analyze all the images raw data.
- **Features are interpretable**, so you **might understand why they are not working**. You do not interpret the classifier if is not a very simple one.
- You can **adjust features to improve your performance, for example if you discover something new**
- **Limited amount of training data needed, because the features are directly coded by an expert**
- **You can give more relevance to some features**

instead the **cons** are:

- **Requires a lot of design/programming efforts**
- **Not viable in many visual recognition task** (e.g. on natural images) which are **easily performed by humans**.
- **Risk** of **overfitting** the **(small) training set used in the design** since the features may consider only a **partial view over the characteristics defining the images** belonging to a given class.
- **Not very general and "portable"** (i.e. one that solve the task for the wheel does not solve the task for a car).

With the advent of **Deep Learning** the approach is to **resort a data driven model also for feature extraction**. The idea is that as for the classifier resorting to a data driven model improved the performance of classification, maybe feature extraction can be improved resorting to a data driven model. The **optimization of the two model can be done together**, indeed in this way the **feature extraction part will be optimized to minimize the classification error**.

4.2.2 Convolution

2D Correlation is a linear transformation. Linearity implies that:

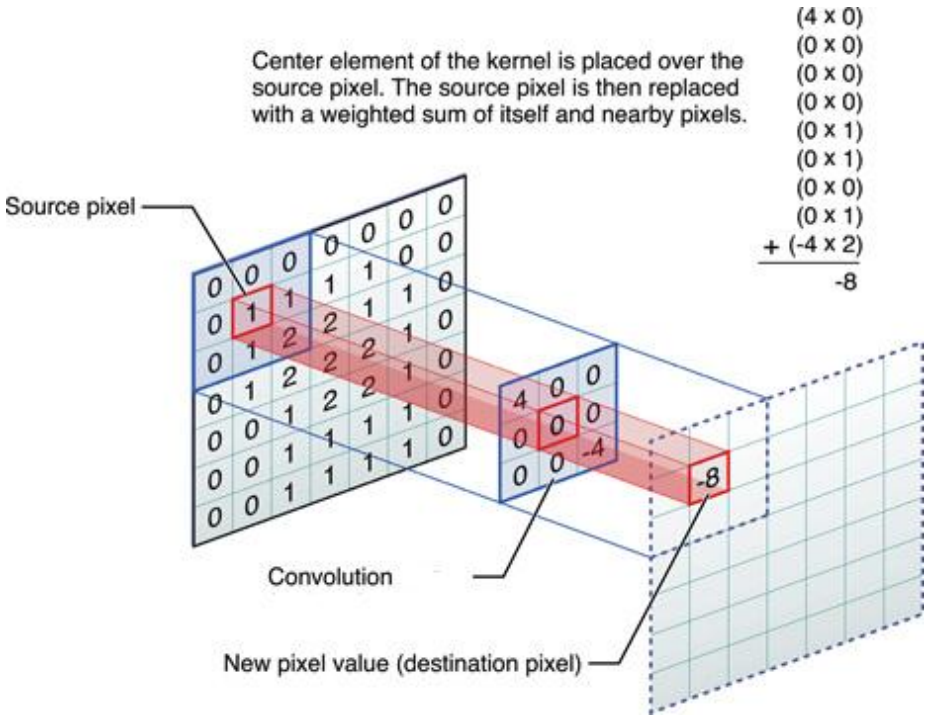
$$T[I](r,c) = \sum_{(x,y)\in U} w(x,y) \cdot I(r+x, c+y)$$

2D Convolution is a linear transformation. Linearity implies that:

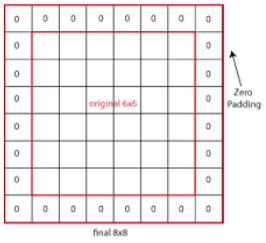
$$T[I](r,c) = \sum_{(x,y)\in U} w(x,y) \cdot I(r-x, c-y)$$

that is similar to the correlation except for the minus sign, indeed in correlation is $I(r+x, c+y)$. Hence, **as for the correlation, the weights can be considered as**

a filter h. Furthermore the **filter h entirely defines convolution since it operates the same on each pixel**. So since the same operation is being performed in each pixel of the input image it is equivalent to the correlation up to a "flip" (both vertically and horizontally) in the filter w. The fact that the filter is flipped is not relevant for us since the filters coefficients are not defined a priory but will be learnt, so they will be already flipped for the convolution.

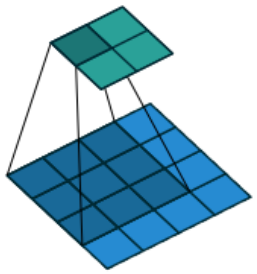


When the filter is applied to the image boundaries pixels a part of the filter does not overlap with any pixel of the image so: how to define convolution output close to image boundaries? Padding with zero is the most frequent option, as this does not change the output size. However, no padding or symmetric padding (i.e. the opposite side pixel of the image are the padding) are also viable options.

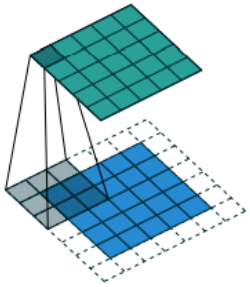


Other than the padding values there are different type of padding:

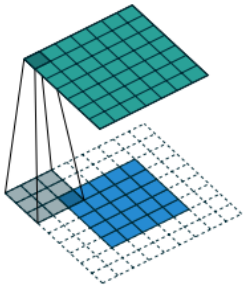
- **No padding**: the **size** of the input is **not preserved since the output will be smaller**



- **Half padding**: preserve the size of the input i.e. the **output has the same dimension as the input**



- **Full padding**: it **increase the size of the image**, indeed the output dimensions are bigger than the input one. **Not usually done** since the padding defines partially the output so the **added pixel to the output won't be exactly related to the image**

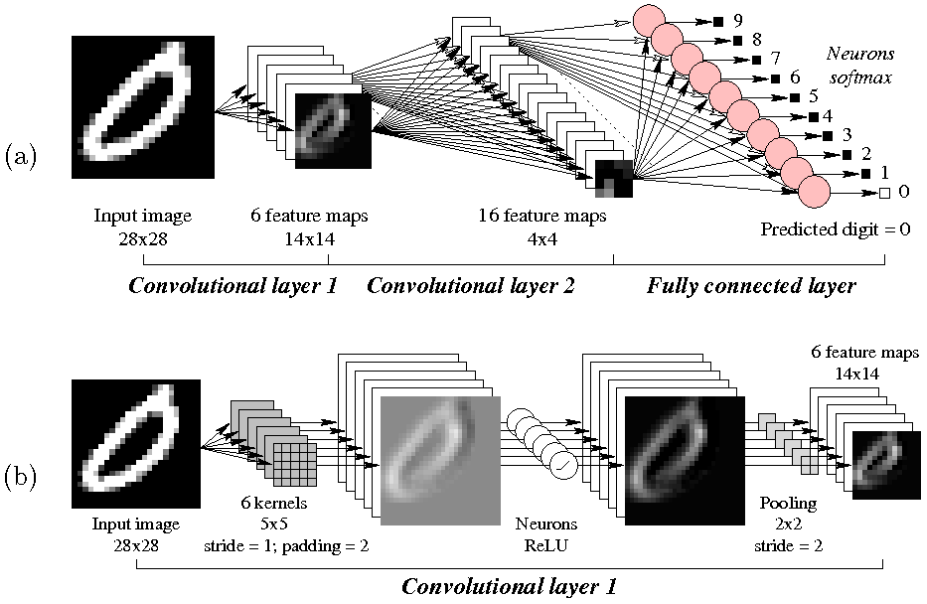


NOTE Using a padding the dimension of the input image is increased by 2 in both height and width for each "line" of padding added.

NOTE usually filters are not very big (e.g. 3x3, 5x5), **square and with odd dimension so that is easy to identify the center**. **Larger filters were used in the past but it was shown that is better to have smaller ones**.

4.2.3 CNN

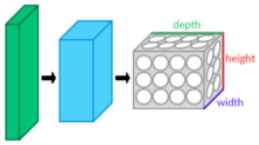
A typical convolutional neural network is the following:



Usually in this networks weights are not drawn. **The convolution preserve the dimension of the image** (i.e. spatial extent, spatial resolution) **and multiply its depth** (from 1, binary image, to 6; if it would be a RGB the depth would have been 3) creating the activations. Then **with Max pooling the spacial extension of the image is reduced** (in the image by half) **preserving the depth** (still 6). **These two operations can be repeated making the image losing a lot of spacial resolution and gaining depth (i.e increase the number of channels of the image) until the image becomes a vector that can be fed to a fully connected neural network**.

As we have seen **CNN are typically made of blocks that include**:

- **Convolutional layers**
- **Non-linearities** (activation function)
- **Maxpooling**



The **image must be considered as a volume** (that in the upper image was stratified) that has **width, height and depth**, and **all the activations are organized over the volume**. An **image passing through a CNN is transformed in a sequence of volumes**:

- **As the depth increases, the height and width of the volume decreases**.
- **Each layer takes an input and returns a volume**.

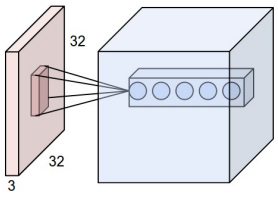
4.2.4 Convolutional Layers

Convolutional layers "mix" all the input components. the output is a linear combination of all the values in a region of the input:

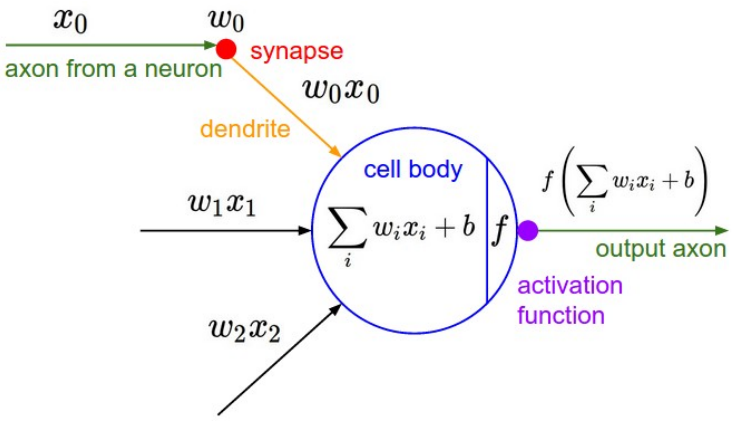
$$y = \left(\sum_{i \in RGBchannels} w_{ij}^{1} \cdot x_i \right) + b^1 =$$

$$= \left[\left(\sum_{i \in Rchannel} w_{ij}^{1(R)} \cdot x_i^{(R)} \right) + \left(\sum_{i \in Gchannel} w_{ij}^{1(G)} \cdot x_i^{(G)} \right) + \left(\sum_{i \in Bchannel} w_{ij}^{1(B)} \cdot x_i^{(B)} \right) \right] + b^1$$

with the index i spanning over the three channels of the input image, i.e. **the filter has also a depth over the colour channels**. The convolution of a RGB filter over a RGB image produces one layer of the output volume.



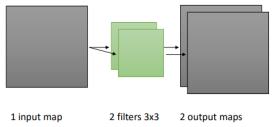
whose operation can be seen as computed by the following neuron where f is the activation function (see later):

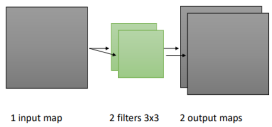


The **parameters (learned) of this layer are called filters**. The same filter is used through the whole spatial extent of the input and **each filter has a single bias, for all its depth, associated to it**. **Different filters yield different layers in the output**:

$$\left(\sum_{i \in RGBchannels} w_{ij}^1 \cdot x_i\right) + b^1$$

$$\left(\sum_{i \in RGBchannels} w_{ij}^2 \cdot x_i\right) + b^2$$





In this way, **using different filters, the depth of the output volume is increased**. So the **depth of the output volume is equal to the number of filters used** (e.g. in the first RGB image of depth 3 in the first convolution are used 8 filters of depth 3 to pass from the bazar photo to the volume of depth 8) Hence, to recap, **convolutional layers "mix" all the input components**:

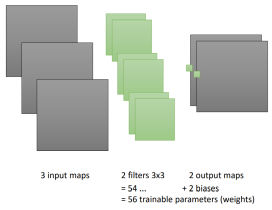
- The **output is also called volume or activation maps**.
- **Each filter yields a different slice of the output volume**.

- **Each filter has depth equal to the depth of the input volume**(***).

Some observation about convolutional layers:

- **Convolutional layers are described by a set of filters**.
- Filters represent the weights of these linear combination.
- **Filters have very small spatial extent (each time mix only a small quantity of pixel) and large depth extent**.
- The **filter depth is typically not specified, because it corresponds to the number of layers of the input volume** since the output of the convolution against a filter becomes a slice in the volume feed to the next layer.

So the convolution layer increase the image depth: the values in a single point of the activation maps are the mixture of the input values covered by the centered filter. **Since filters are parameters they are learned**: the only things to **define is the width and the height of the filters and their number, so they are hyperparameters**(***). Considering an **input volume with 3 depth layers** and 2 **filters (that will also have 3 depth layers)**:



where each output layer is given by the sum of the convolution of the input layers over the corresponding filter (***) (i.e. each output layer correspond to a filter). The **parameters can be count considering the number of filters**, **the depth of the filters (i.e. the depth of the input)**, the **size of the filter** (usually square) and the **biases that are one for each filter**:

$$N_{param} = N_{filters} \cdot FilterDepth \cdot FilterSize^2 + N_{bias} =$$

$$= 2N_{filters} \cdot FilterDepth \cdot FilterSize^2$$

since each layer of each filter change (each input channel has a learnable filter) and the filter are considered squared and to each filter correspond a bias.

REMEMBER **the filter depth is equal to the input depth** and **the number of filters will determine the depth of the output** i.e. how many layer has the output volume. (***)

NOTE CNN Arithmetic is commonly asked at the exams.

The **idea of increasing the volume** of the image is that each time we do it the value of **activation contain information that refers to a rather large region of the input image, so it is an high level information from the semantic point of view i.e. from the pattern point of view**: as you **extend the depth smaller patterns are extracted**.

4.2.5 Activation

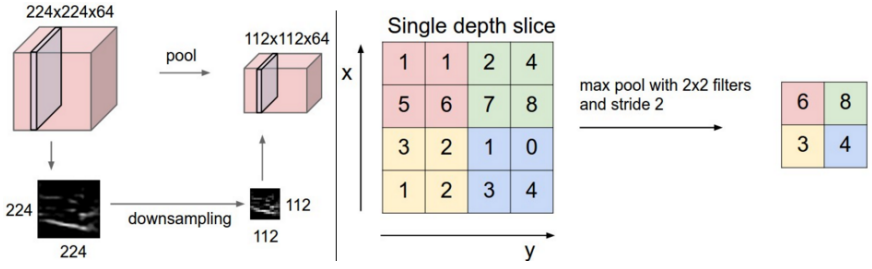
Activation layers introduce non-linearities in the network, otherwise the CNN might be equivalent to a linear classifier: as it was for the neural network there is not point to stack many layers **if you don't insert a non-linearity** because you **would still obtain a linear transformation in the end**. **Hence after any convolutional layer we should place a non-linear activation function**:

- The **ReLU** it's a thresholding on the feature maps, i.e. a max(0,·) operator, and by far is the **most popular** activation function in deep NN.
- **Leaky ReLU**: is like ReLU but include a small slope for negative values, **reducing the problem of dying neurons**.
- **Tanh**: has **range (-1,1)**.
- **Sigmoid** has a **range (0,1)**.

The **most used** in CNN are **ReLU and leaky ReLU**. Remember that the activation function **are applied to the output of the convolutional layer** that is **a single layer of the output volume**, so **it works layer-wise** as showed in one of the images above.

4.2.6 Pooling

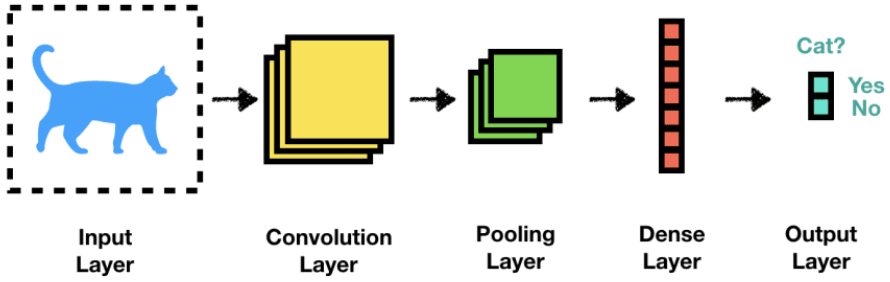
Pooling layers reduce the spatial size of the volume. The pooling layer **operates** independently **on every depth slice of the input** and resizes it spatially, **often using the MAX operation i.e. max pooling**.



A 2x2 filter with stride of 2, discards 75% of samples in a volume (regardless the dimension of the slice) **halving the spatial extension of the volume**.

4.2.7 Dense layers

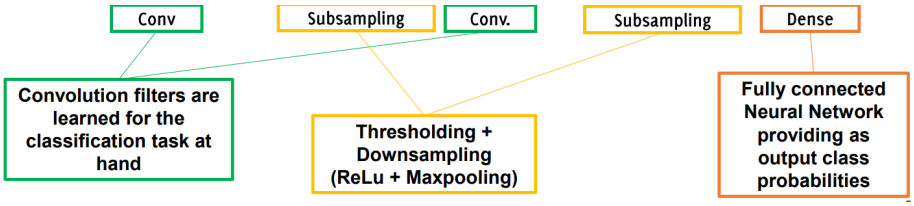
The typical architecture of a convolutional neural network is the following:



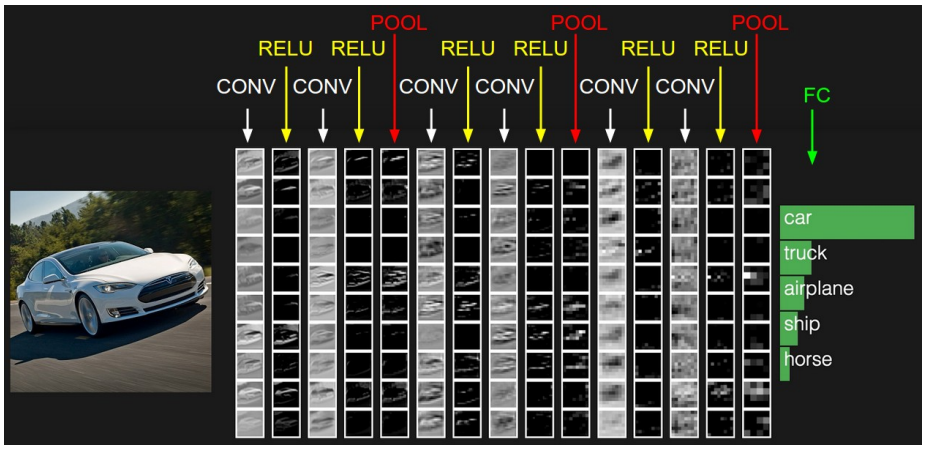
where the **dense layer is nothing that a fully connected multi-layer neural network**. The output of a fully connected (**FC**) layer has the same size as the number of classes, and provides a score for the input image to belong to each class in the form of probabilities (through softmax).

4.2.8 CNN in action

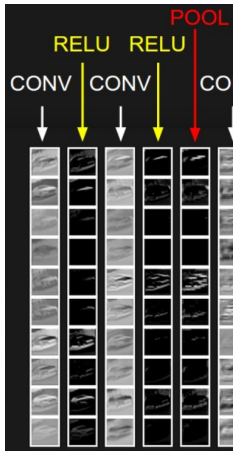
As we have seen the important operation inside a CNN are the following:



where the **convolution extend the depth of the input volume** (maybe also reduce the spatial extension, i.e. width and height, if it uses no padding), the **activation function add the non-linearities**, the **pooling reduces the spatial extension** (i.e. width and height) of the volume and finally the **dense layer does the classification** among the considered classes. Remember that **convolution and pooling are repeated until the volume becomes a vector that can be fed to the dense layer**. Let's see how a CNN works in a visual way:



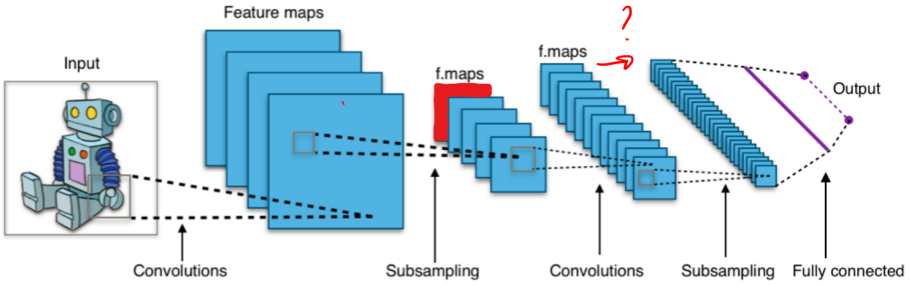
where each layer is represented in the volume as an image (after a pooling layer, is used the same size but different resolution for visualization sake). Looking at the first operations we can see that after the first convolution the car is still distinguishable:



because the convolution is done with a small filter so mixing together the pixels in the neighborhood covered by the filter for each pixel, you won't obtain something much different from the input. After, the ReLU set to zero the negative pixels keeping untouched the positive pixels i.e. it applies a threshold to negative values. In the second convolution to maintain the depth of 10 the convolution layer has 10 filters: **each filter is applied to the whole depth of the output of the ReLU layer (10 layers, i.e. the depth of each filter is 10 and we have 10 filters) , then summing their result (***)**. Hence considering a layer, **some shape eliminated by the previus ReLU**, in the same layer, can pop up again because they were preserved in one of the other layers. After another thresholding done by the ReLU there is the pooling layer that decreases the spatial resolution (after a pooling

layer, is used the same size but different resolution for visualization sake). **Stacking** this **Convolution-ReLU-Pooling** layers at the end we **obtain 10 (depth) really small images (e.g. 4x4 as width x height) for a total of 160 (= 4x4x10) pixel values**. These values are **flatten into a vector and then fed to a fully connected** (i.e. dense) neural network used for classification.
The reason why we repeat convolution-ReLU-pooling operation for several times is that it reduces the dimension of the input **condensing in each pixel of the final output, that is fed to the neural network** after it is flatten, the **information referring to a larger region of the input of the network** and are the result of many linear combination, non-linerarity and thresholding, so they can **represent and highlight** (in terms of pixel intensity) **a feature of the input image** (e.g. the shape of the car lights). **Since the convolution layer plus the activation function can be represented through a neuron** the parameters i.e. the **filters can be trained together with the weights of the fully connected neural network**.

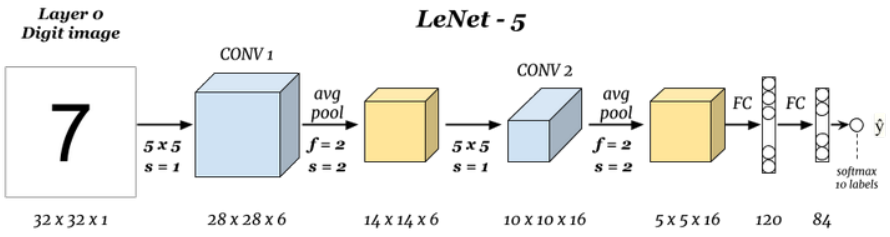
The following CNN representation has an error:



Indeed the **pooling is a subsampling and does not increase the depth of the volume**.

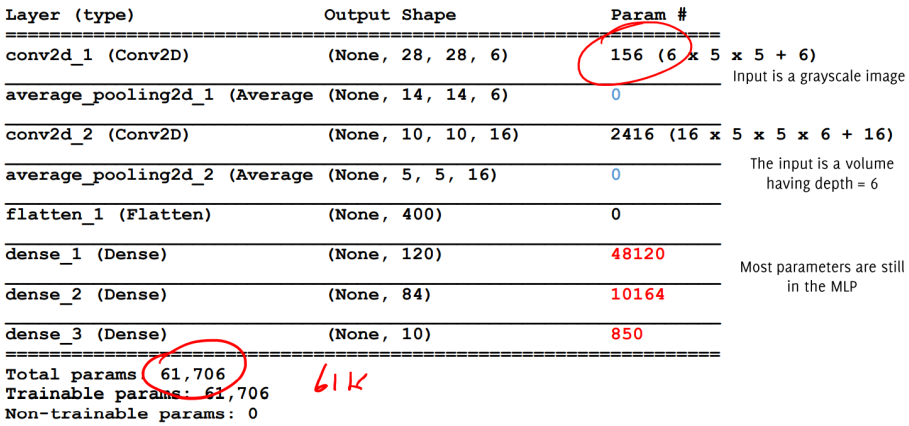
4.2.9 The first CNN

The first CNN that was built was LeNet-5 in 1998. I was meant to classify 32x32 greyscale (i.e. input has depth 1: 32x32x1) digits (i.e. numbers):



The first convolutional layer used a 6 filters 5x5, instead the second convolutional layer 16 filters 5x5. Indeed the input size in both convolution layers reduces by 4

since they were using the **"valid" convolution** that does **not use any padding on the input**. They used the **average pooling because wasn't yet discovered that the max pooling performs better since it introduce an additional non-linearity**. The average pooling was done with a 2x2 coverage (I **don't call it filter since they are not parameters to be learned, it just compute an average over the covered pixels**) and stride 2 since the **spatial extension of the volume is halved**. Then the dense neural network was composed by an input layer with 120 neurons, an hidden layer with 84 neurons and an output layer with 10 neurons i.e. the number of classes (0, 1, ..., 9).
Let's see how many parameters does this network have:



Is important to **remember that the depth of each (*) filter is equal to the one of the input volume** and **for each filters is associated one bias**:

- The first convolutional layer's 6 filters are of depth 1 since the input of the layer is a grayscale image 32x32x1.

$$N_{params}^{(Conv1)} = 6x(5x5x1) + 6$$

- The second 16 convolutional layer's 16 **filters are of depth 6 since the input of the layer is a volume 14x14x6** output of the average pooling layer.

$$N_{params}^{(Conv2)} = 16x(5x5x6) + 16$$

Furthermore average pooling does not have any parameters since it only averages the values in a 2x2 neighborhood using a stride of 2. After the convolutions and the poolings the result is fed to a neural network after it is flattened: **since the resulting volume is 5 x 5 x 16 the flattening creates an input vector of length 400**. Then the input vector is fed to a dense (i.e. fully connected) neural network whose input layer has 120 layers:

$$N_{params}^{(Dense1)} = 400x120 + 120 = 48120$$

where the **+120 is due to the biases, one for each neuron** (***). Then after the input layer there is an hidden layer of 84 neurons:

$$N_{params}^{(Dense2)} = 120x84 + 84 = 10164$$

And finally there is the output layer of 10 neurons:

$$N_{params}^{(Dense3)} = 84x10 + 10 = 850$$

So, as we can see, **most of the parameters comes from the connection inside the multi-layer neural network**, the convolution were there to reduce the size of the input image a bit. Indeed if the neural network would take as input the whole image, **considering only the last two layers of the neural network (the hidden layer and the output layer), the number of parameters to learn would be**:

$$32 \times 32 \times 1 = 1024 pixels \Rightarrow (1024 \times 84 + 84) + (84 \times 10 + 10) = 86950$$

In this case the difference is not very big, but the number is still higher than before. Furthermore adding also the input layer with 120 neurons, to consider the same neural network as before, would drastically increase the number of parameters (134014). To **highlight even more the reduction of parameters that can be obtained through the convolution and the pooling layers is that if the input is RGB** i.e. $32 \times 32 \times 3$:

- CNN: **only the number of parameters in the filters at the first convolutional layer increases since the filter depth is equal to the input volume depth** (grayscale $5 \times 5 \times \underline{1}$ and RGB $5 \times 5 \times \underline{3}$):

  $$(6 \times (5 \times 5 \times 1)) \Rightarrow (6 \times (5 \times 5 \times \mathbf{3}))$$

  so:

  $$156 + 61550 \Rightarrow 456 + 61550$$

- **Multi-layer NN: everything increases by a factor 3**:

  $$32 \times 32 \times 3 = 1024 \times 3 pixels \Rightarrow (1024 \times 3 \times 84 + 84) + (84 \times 10 + 10) = 86950 + (1024 \times 2 \times 84) \sim 86950 \times 2.98$$

  so:

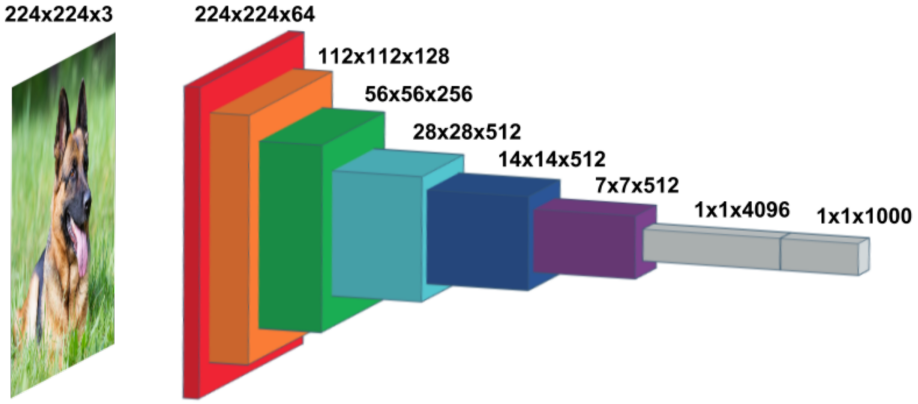
  $$86950 \Rightarrow \sim \mathbf{86950} \times \mathbf{3}$$

To summarize **it does not use each pixel as a separate input of a large multi-layer perceptron** for the following reasons:

- **there would be much more parameters** as the **complexity** of either the input (spatial extention increases and/or depth increases) or the **network** (bigger and/or more layers) **increases**.
- **images are highly spatially correlated and using individual pixel of the image as separate input features would not take advantage of these correlations**.

4.2.10 Latent representation in CNNs

The first layers of the CNN, the ones before the dense neural network **operate a feature extraction**: **starting from low-level features** present in the image increasing the number of layer of convolution-ReLU-pooling increases the level of the feature extracted from the input image. A 3D representation of the feature extraction part of a CNN can be the following:



Theoretically the feature extracted by the feature extraction layers can be 1×1 as spatial extension but with a high depth. The **output of the feature extraction layers of the CNN is a vector**: the **last volume will be then rearranged as a vector**, also called **latent representation of the input**, that **synthesize the extracted features in a form that can be fed to the neural network**.

Repeating the t-SNE representation **using the latent representation of the input to evaluate the difference of the input images we obtain a good results since similar objects are next to each other**:

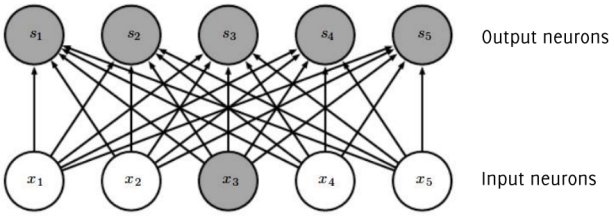
Hence distances in the latent representation of a CNN are much more meaningful than data itself, i.e. than pixel-wise distance.

4.3 CNN parameters and Learning

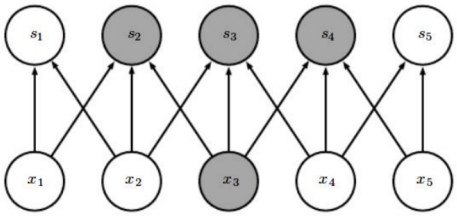
Convolution is a linear operation (*), therefore, if you unroll the input image to a vector as well as the output image, you can **consider convolution weights (i.e. the filter) as the weights of a multi-layer neural network that has two layers connected by the weights**. How a CNN can provides features that are meaningful to understand the content of the image? i.e. **what are the differences between multi-layer neural network and CNN then?**

- **CNN feature a sparse connectivity** (i.e. **only few inputs contribute to a different output**) **and not fully connectivity (i.e. each input contribute to each output) as in dense neural networks**. For example a convolutional neural network on a single layer (of 5 pixel for simplicity) with a filter with spatial extension of 3×1 can be compared to a fully connected neural network with the same neurons in this way:

Fully connected



3x1 convolutional



Having a connection between the input (pixel) x_i and the output (pixel) s_j means that the input contribute to define the output. Hence the sparse connectivity in a CNN is due to the fact that each pixel of the **input** (i.e. each x_i) is **considered**, by the transformation, in a pixel of the **output** (i.e. each s_j) **only if it is covered by the filter** (i.e. with a 3×1 filter if the filter is centered either on it or on its the neighbours). The **fact that the connection is sparse can be noted even considering the number of parameters in the two configurations**:

- Dense layer: the parameters between two fully connected layer are:

$$N_{params}^{(FC)} = I \times O + B$$

  where I is the number of input neurons, O in the number of output neurons and B the number of biases (with B = O). So in the latter pictures are:

$$5 \times 5 + 5 = 30$$

- Convolutional layer: the parameters in a convolution layer are:

$$N_{params}^{(CNN)} = W_w \times W_h \times N_W + B$$

  where W_w is the filter width, W_h is the filter height, N_W is the number of filters and B is the the number of biases (with B = N_W). So in the latter pictures are:

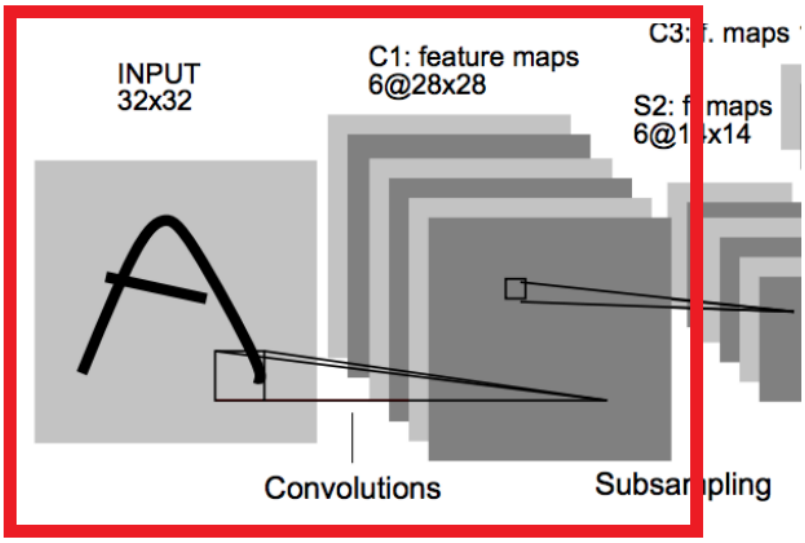
$$3 \times 1 \times 1 + 1 = 4$$

  where $N_W = 1$ and $B = 1$ since we are considering a single layer.

indeed, **the number of weights, i.e. connection, is much lower for the convolution layer than a dense layer**. Still considering a single output layer and considering the number of input neurons I equal to the number of

neurons O, as typically is in a convolution layer (i.e. typically is used half padding) **the number of parameters for a dense layer is** $\propto I^2$, **instead the number of parameters for a convolution layer is** $\propto W_w^2$ ($\times N_w$ but we consider a single output layer) if the filters are considered square ($W_w = W_h$) as usually is. Hence, **as the input increases the difference in the number of parameters between the two architecture increases even more**, indeed the spatial extension of the filters is always kept very small since they works better. Considering more filters, i.e. increasing the number of parameters of the convolutional layer, the difference in parameters does not change since each filters produce another output layer that in a dense architecture can be represented again as above so the number of parameters we counted should be multiplied in both cases for the number of output layers.

- In a **convolution** layer weights are shared and equal for each part of the input, so all the neurons in the same layer of a feature map use the same weights and bias: this why the number of parameters in a convolution layer with respect to a dense layer are reduced of a factor I. The **underlying assumption** is the **spatial invariance**: **if one feature is useful to compute at some spatial position (x, y) then it should also be useful to compute at a different position** (x_2, y_2).



If the first layer were a multi-layer neural network, since is a convolution it has:

- 32×32 input.
- $28 \times 28 \times 6$ neurons in the output.
- 5×5 non-zero weights shared by each input.
- it would be a $(5 \times 5 \times 6) weights + (6) biases$ for a total of 156 parameters, instead of $(32 \times 32) \times (28 \times 28 \times 6) weights + (28 \times 28 \times 6) biases$ for a total of a lot (??????check?????)

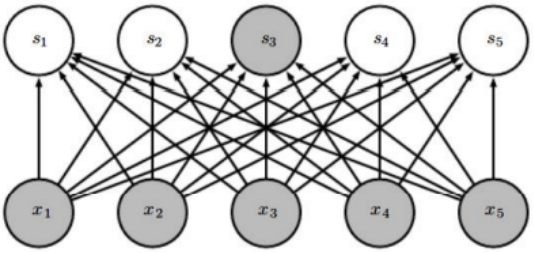
4.3.1 The receptive field (***)

Why does CNN have sparse connectivity and weight sharing? Normally in a FFNN it **does not make sense in a regression task** for example, to design some feature and take into consideration only a part of it (sparse connectivity, so not all inputs contribute to a certain output). **But CNNs handle images**, and images **are very redundant** and due to locality the value in a certain pixel might be not influenced by a pixel too far: **to understand what a part of the image represent is not important to take all the image but only a small portion around the considered region i.e. the patterns are local**. Furthermore using full connectivity does not make sense using shared weights since each of the output would have the same value (different of a multiplicative factor w_j due to the fact that each input is involved in the definition of the output and each time the weight related to the output j is the same for each input), but **with sparse connectivity using shared weights gains a sense** since, as we have seen, the **filters are matching a pattern that must be searched locally (i.e. what the filter cover, i.e. spatial connectivity)** in all the image (i.e. in each pixel/input) i.e. in all the input, because they are **local but can be found in every region of the image so we must look for it**.

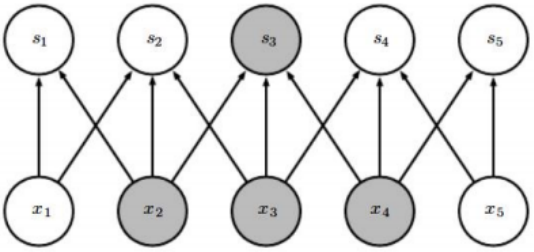
The **receptive field** is one of the basic concept and a very important aspect in CNNs. **Due to sparse connectivity**, unlike in dense networks where the value of each output depends on the entire input, in a **CNN an output only depends on a region of the input**. **This region in the input is the receptive field for that output**. The **deeper you go the wider the receptive field is**: **maxpooling, convolutions and stride $>$ 1 increase the perceptive field**:

- **Maxpooling condense information under the pooling filter in one value**, the maximum one i.e. **the one with higher intensity**.
- The **next convolution**, considering the spatial extent of the convolution filter still the same, **will cover a bigger area than before, due to the effects of maxpooling**.

Fully connected



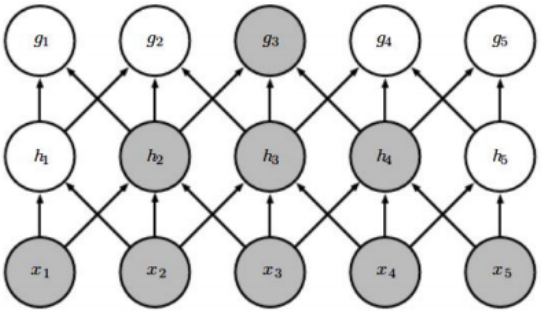
3x1 convolutional



In a fully connected neural network the receptive field is the whole input because each output neurons depends from all the inputs (i.e. takes as input the whole input layer). Instead **in a CNN** the receptive field **from one layer to another is limited to few inputs but at the end** of the operation chain, i.e. o**f the feature extraction network, the receptive field may be quite large, so one element of the vector fed to the NN may refer to a rather relevant portion of the input image**. Deeper the neurons depend on wider patches of the input (**convolution is enough to increase receptive field, no need of maxpooling**):

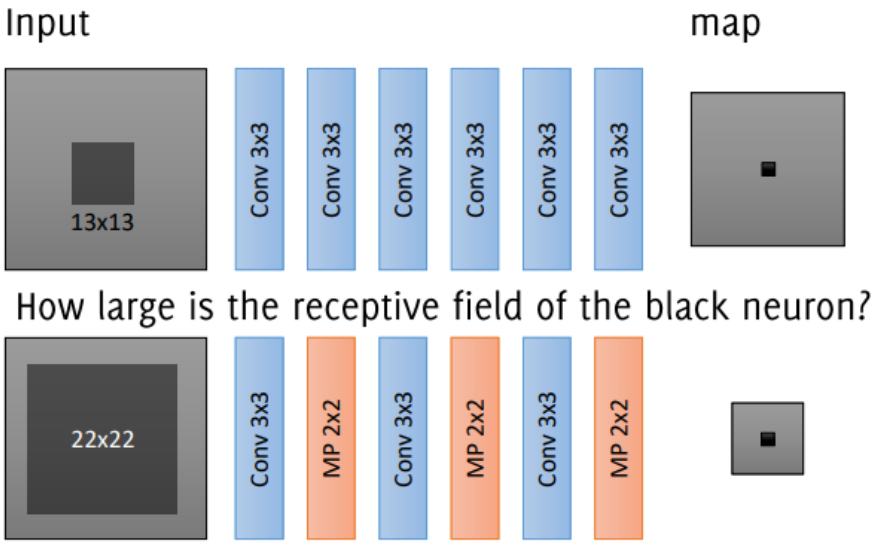
3x1 convolutional

3x1 convolutional



Usually, the receptive fields refers to the final output unit of the network in relation to the network input, but the same definition holds for intermediate volumes.

Let's consider the following example:

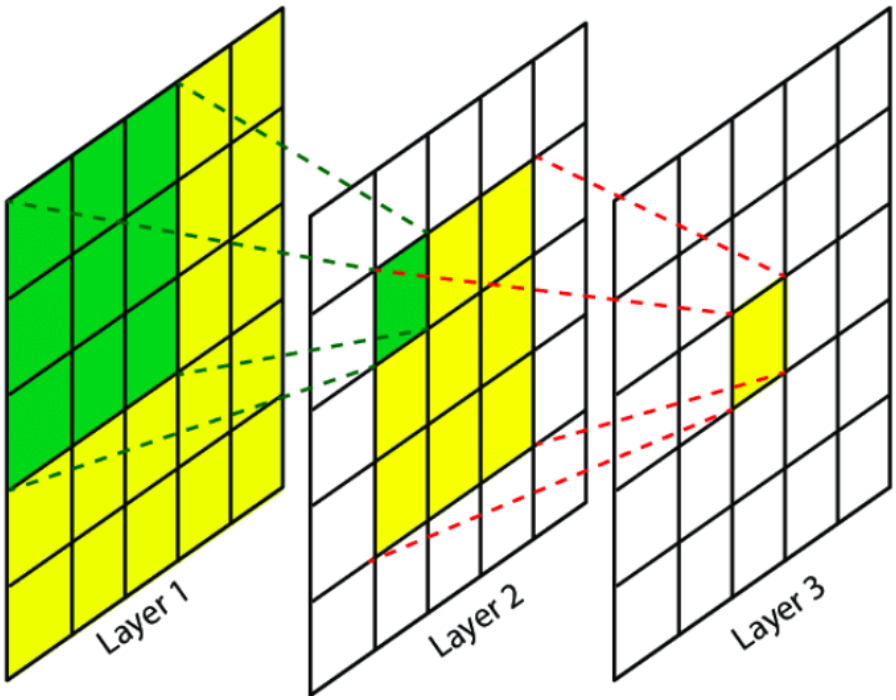


The input image as well as the output after an operation can be imagined as a layer of spheres i.e. the neurons, one for each pixel. The application of the filter creates the sparse connections between the two layer that, each time connects the neuron covered by the filter to one neuron of the output centered w.r.t. the filter. Starting to one neuron of the final output going backwards we can consider only the connection that each time are used to compute the output of the layer we are considering until we arrive to the initial input.

- **Only convolutional layers**: each of the convolution layer increase the receptive fields such that:

  $$\mathbf{D_i = (2i + 1) \times (2i + 1)}$$

  where D_i **is the dimension after the** $i-th$ **convolution layer and the** $+1$ **are due to the fact that the starting pixel has dimension** 1×1.



- **Convolutional layer + maxpooling**: the **convolutional layer behave as before, adding each time 2 to each dimension**, instead the **maxpooling layers double each dimension since the filter is** 2×2. So each of the convolution + maxpooling layer increase the receptive fields such that:

$$\mathbf{D_i} = (\mathbf{2} + \mathbf{2} \cdot \mathbf{D_{i-1}^{width}}) \times (\mathbf{2} + \mathbf{2} \cdot \mathbf{D_{i-1}^{height}})$$

$$D_0 = 1$$

  where D_i **is the dimension after the** $i-th$ **convolution + maxpooling layer, the** D_{i-1}^{width} **and** D_{i-1}^{height} **are the width and the height of after the** $(i-1)-th$ **convolution + maxpooling layer** $D_0 = 1$ **due to the fact that the starting pixel has dimension** 1×1. Hence each time starting from the (i-1)-th step we multiply by two due to the 2x2 maxpooling and then apply the convolution that increase the receptive field by 2. This time the index i is considered in the maxpooling, not in the convolution as before.

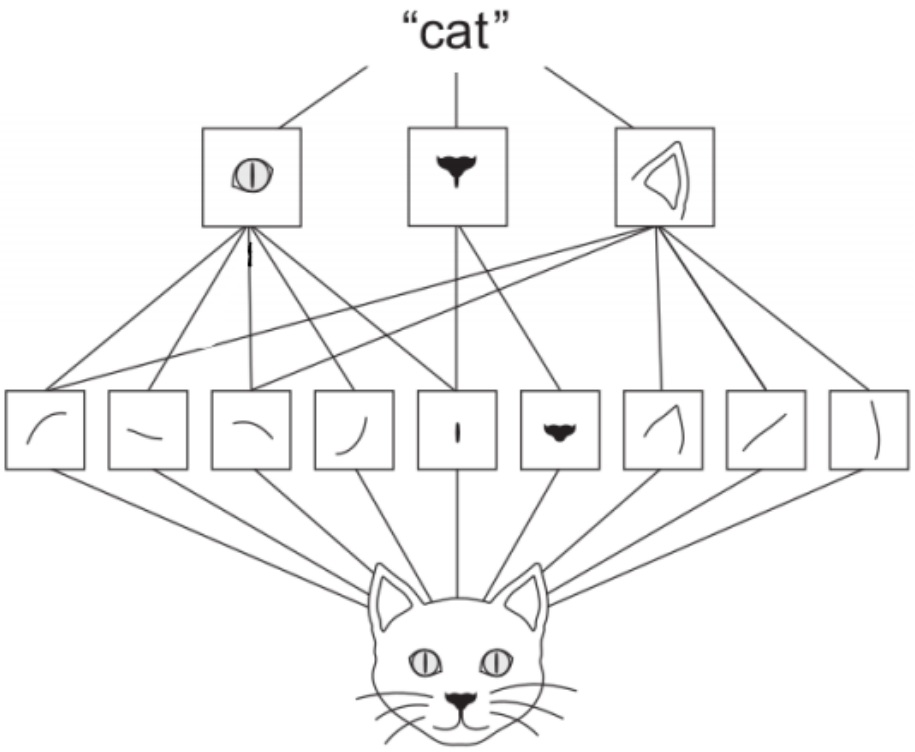
Looking at the resulting **receptive field** is clear that **deeper we go bigger they become**. This result can be obtained with only convolutions but **coupling it with maxpooling it increases the effectiveness of the procedure since with the same number of layer but with less parameters to learn**, since maxpooling has no parameters, **the receptive field increases in a significant way**. Hence the **pros** are two:

- **Halved the parameters to be learned**.
- **Increased the perceptive field, with the same depth**. (13x13 vs 22x22).

As we move **deeper layers**:

- **spatial resolution is reduced (height x width)**
- **the number of maps increases (depth)**

The **first convolution** layer will **extract very localized small features**, since the output **perceptive field is very small**. As the perceptive field increases the exact location in the image loses relevance for the network since the **receptive field is greater and the feature can be anywhere inside it**, and this **aligns with our interest since we are interest in its existence inside the input image, not about its exact location**. Hence the feature extraction goes **from low level features very localized to high level features whose exact location is not known** (***). We search for **high-level patterns, and don't care too much about their exact location**, since they are **more meaningful to separate the dataset into the target classes**(*). **Low level details like lines or edges are not interesting since they are not characteristics present in only some classes but obviously are present in all the classes**, so they **are not useful for classification**. Furthermore, **there are more high-level patterns than low-level details because the depth of the activation maps (volumes) i.e. the number of detected features, increases with the depth of convolution layer chain**.



So is **not only to reduce image dimensionality** (*) they came up with the structure of CNN we have seen and is not a case this architecture works, indeed due to the nature of the input image **connectivity must be sparse due to locality of patterns** (*) and **weights must be shared since patterns can be found in any place of the image** (*). So **CNN can be applied to every data different from image** that **also** has the said **properties**:

- **pattern locality i.e. spatial coherence**
- **redundancy of the input**
- **classes distinguished through patterns**

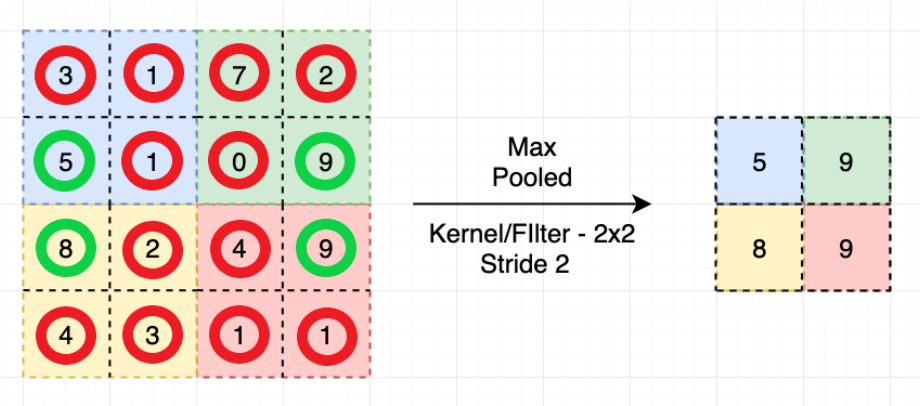
and so where is **meaningful to detect patterns in all the data domain** like **ECG signals, audio signals, video signals**.

4.3.2 CNN training

The **CNN can be seen as a multi-layer with sparse connectivities and shared weights**. Hence **CNN can be in principle trained by gradient descent to minimize a loss function over a batch** (e.g. binary cross-entropy, RMSE, Hinge loss...). Gradient can be computed by **backpropagation (chain rule) as long as**

we can derive each layer of the CNN. **Sparse connectivity is not a problem**, you just take fewer weights when you compute the backpropagation. **Weight sharing needs to be taken into account while computing derivatives for the loss**: fewer parameters to be used in the derivatives since they are equal for all the layer, so **is just a matter of organization of the derivatives for backpropagation**. There are just **few details missing**:

- **backpropagation with max pooling**: **the gradient is only routed through the input pixel that contributes to the output value i.e. the input pixel with the maximum value for each filter coverage**. The **derivative** is:
  - **1 at the location corresponding the maximum** (green circle)
  - **0 otherwise** (red circle)



- The **derivative of ReLU is straightforward**:

$$g'(x) = \begin{cases} 0 & x \leq 0 \\ 1 & x > 0 \end{cases}$$

  To be fair **in zero it is not differentiable but we could consider it equal to 0 without having any effects in practice**.

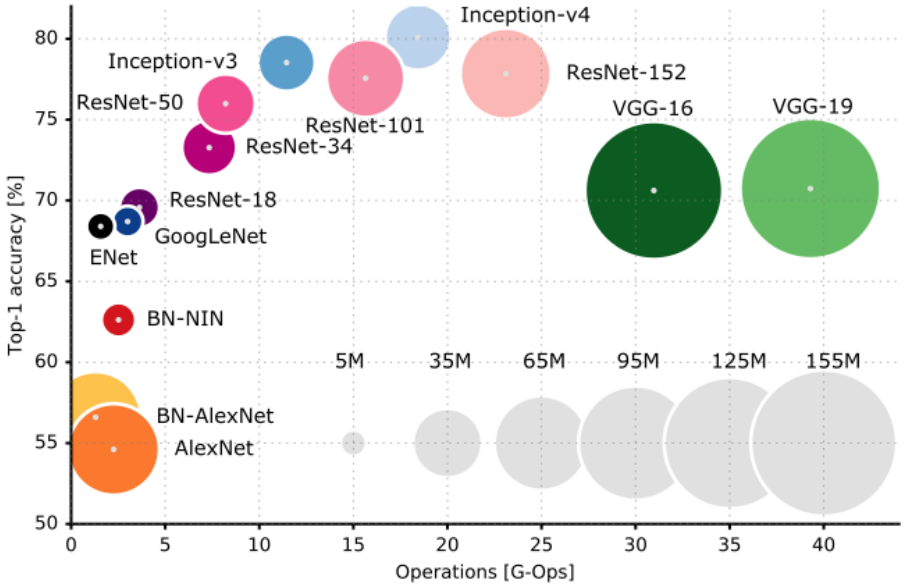
So all gradients are definable and backpropagation can be performed without problems.

Remember The gradient descent is the optimization procedure, that needs the computation of the gradient of the loss function to be applied. The gradient of the loss function is then computed through backpropagation.

4.3.3 Data scarcity: training a CNN with limited amount of data.

Deep learning models are very data hungry, they **need a lot of data to set properly all their parameters**. Networks such as Alexnet have trained on ImageNet

datasets containing tens of thousands of images over hundreds of classes. This is **necessary to define millions of parameters characterizing these networks**:



Indeed in the latter graph the diameter of the circle is proportional to the number of parameters of the networks: **even if the feature extraction part has not a lot of parameters the neural network may be very deep, so the CNN may have a lot of parameters**. To train millions of parameters we expect a lot of data.

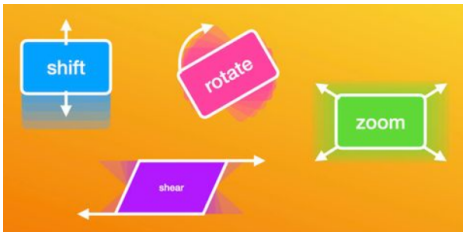
Not in any application we are given such a large training set an may be that each image in the training set has to be annotated. **How to train a deep learning model with a few training images (few hundred or thousand)?** We can resort to:

- **Data Augmentation**
- **Transfer Learning**

4.3.4 Limited amount of data: data augmentation

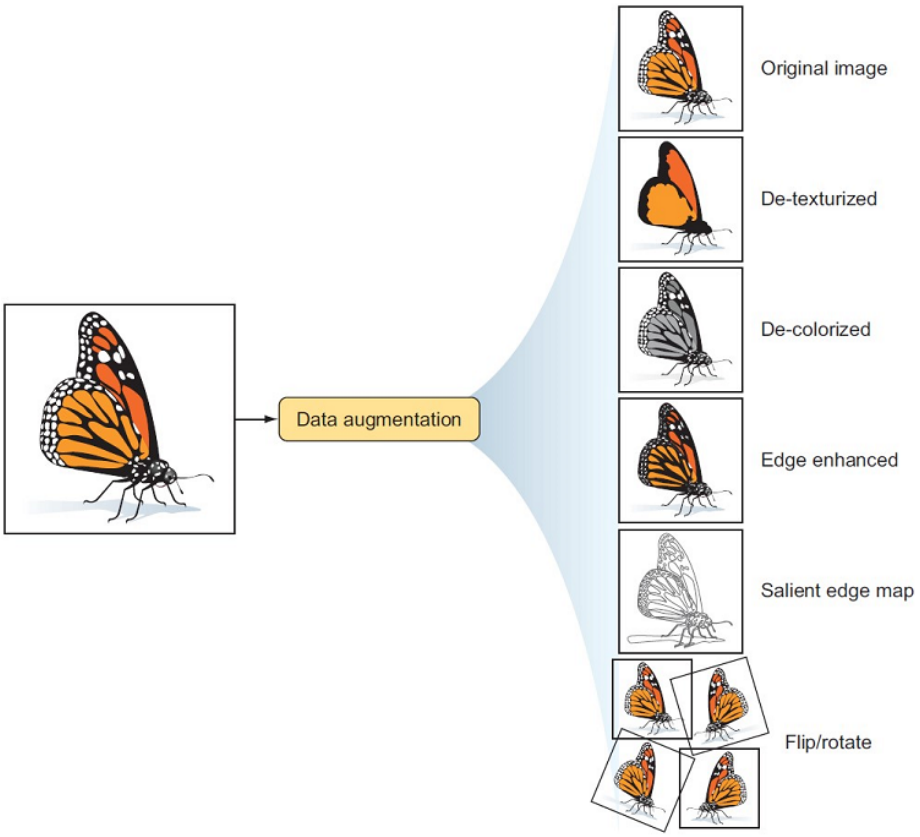
Often, each annotated image represents a class of images that are all likely to belong to the same class. In aerial photographs, for instance, it is normal to have rotated, shifted or scaled images without changing the label, since the orientation does not change the object class. Hence they **are different vector/input that must be classified equally**. **Data augmentation help to teach to the network the invariant properties of the classes** (***). Data augmentation is **typically performed by means of**:

- **Geometric transformations**:
  - Shifts/Rotation/Affine/perspective distortions help to render your image **as it would have seen from a different view**.
  - **Shear**
  - **Scaling**: **teach to the network that the feature dimension is irrelevant**.
  - **Flip**



This transformation can be done on the dataset image, keeping in mind that they do not have to change its classification label (as for example may be if we scale of a 20 factor), indeed in that case it would confuse the network. We should point out that **for some transformation there is the need of adding a padding that can be for example black, or can repeat a value of the border of the image**.

- **Photometric trasformations**:
  - **Adding noise**
  - **Modifying average intensity**
  - **Superimposing other images**
  - **Modifying image contrast**



Maybe **using data augmentation you are not introducing the same data variability as using different images**, but we are **increasing the number of samples** that can be used to compute the loss, and they are all different. Furthermore, it may increase the capability of the network to generalize.

Augmentation **can greatly improve classification performance** and **successfully handle class imbalance such that the class proportion can be made almost the same** (*). This avoid the classification towards the majority of classes and never to the minority classes.

Augmentation can be used to provide new training sample at each epoch: **with data augmentation plugged in you never take the same image twice in different epochs**. **Inserting data augmentation layer before the CNN at each iteration a image is picked form the dataset, a random subset of transformations are applied and then it is fed to the CNN**. This allow the network to **learn not only the peculiar patterns of a certain class but also what are the irrelevant variation in the images for defining the class output** (*). **Feeding images with different orientation, for example, may help the network to understand that the orientation of some of the detected features are not relevant to define the images belonging to a certain class**.

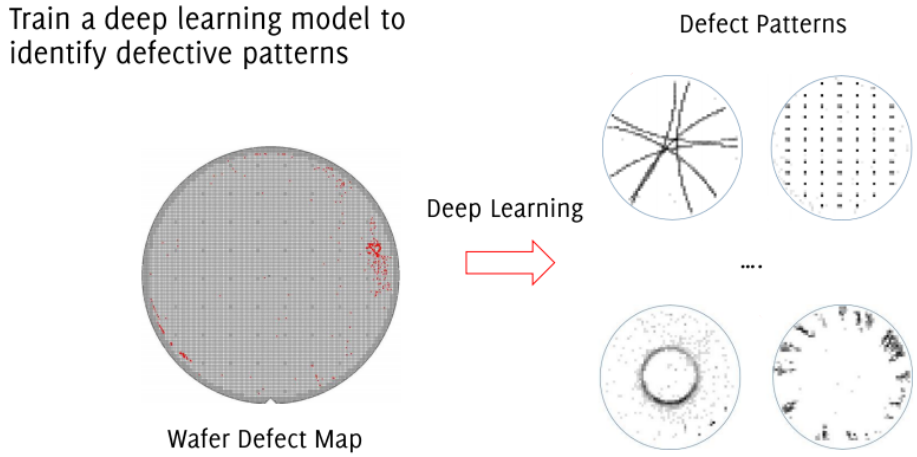
The way you design data augmentation is up to us. **Data augmentation is a way to control the kind of examples we are giving to the network during training, letting us to define what the network is learning i.e. what kind of invariant we would expect for the network to be considered**.

This sort of data augmentation **does not include all the possible transformation you might get**, so **might not be enough to capture the intra-class variability of images**, due to:

- superimposition of targets i.e. **multiple object in the same image**
- **background variations**
- **out of focus, bad exposure**

To some extent this type of transformation can be included in the dataset inserting manually the transformation of the image (e.g. saturation). In case you **need extra flexibility for data augmentation** the software let you configure it on your own.

For example in the following problem:



In some cases certain kind of patterns might appear only in certain locations of the wafer, so depending on the label you can define specific relevant augmentation that do not change the class of the pattern (e.g. rotation transformation on the round one, or flipping on the matrix one).

Even if the CNN is trained using augmentation, it won't achieve perfect invariance with reference to considered transformations. With **test time augmentation** (TTA), augmentation can be also performed at test time **improving prediction accuracy**:

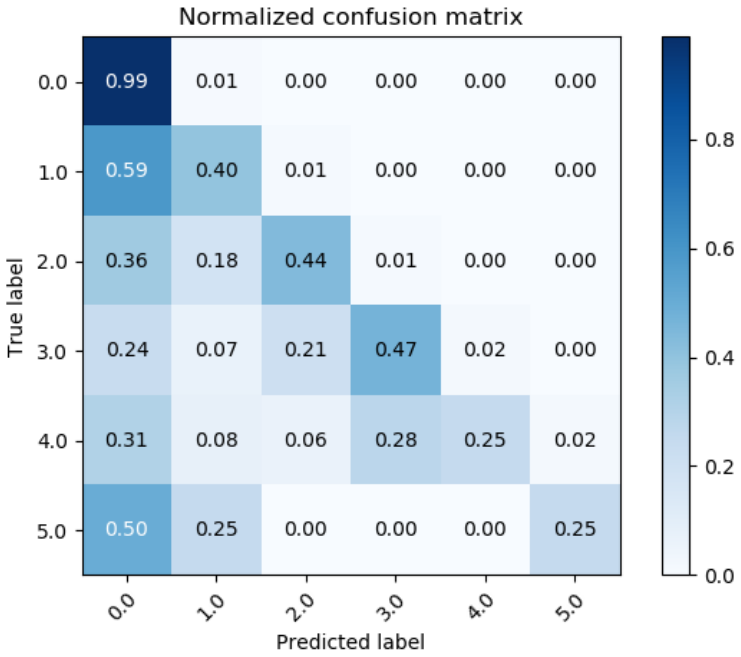
- Perform random augmentation of each test image I creating a batch of augmented images of I. **The augmentation chosen must be the ones that do not change the class but since we don't know yet the label of the sample we should exclude all the transformation that change or might change the class of at least one sample** (e.g. rotation in the wafer defect detection since the matrix defect pattern must be horizontal or vertical and cannot be rotated). In practice this is done by including an augmentation layer.
- **Feed** one at a time **all the augmented sample of the batch**.
- **Average the predictions over all the class of each augmented image**
- **Take the average vector of predictions over the entire batch for defining the final guess**.

This is **like having an ensemble of classifiers where each classifier is the same but you only change the input**. This process can be **parallelized** feeding the same neural network with different augmented images. **Test time augmentation is particularly useful for test images where the model is pretty unsure**.

There are recent study to improve performance of a CNN like framework based on **reinforcement learning to learn what is the subset of transformations that allow to improve the best the classification**.
So **data augmentation can be considered as an hyperparameter of the model that must be tuned correctly** (***).

4.3.5 Performance measures

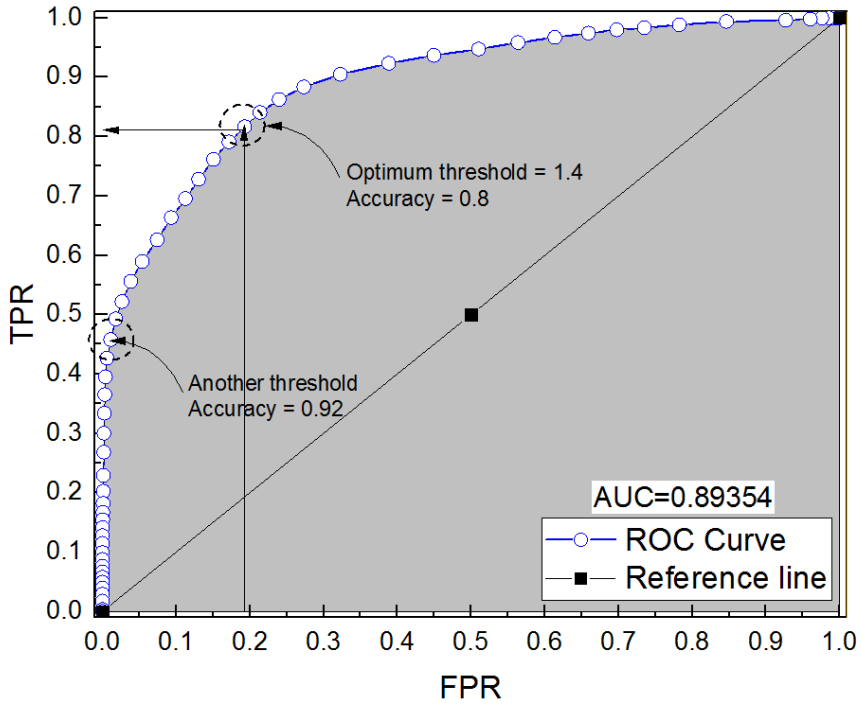
Working on a **classification problem**, to **evaluate its performance** is useful to use **confusion matrix**, a $L \times L$ matrix, with L **is the number of classes** of the problem, where the element $C(i,j)$ **i.e. at the** $i-th$ **row and** $j-th$ **column corresponds to the percentage of elements belonging to class** i **classified as elements of class** j.



The **elements outside the diagonal represent the classification errors**, so the **ideal** confusion matrix is the **identity matrix** since each input of class i is classified as i (never happens for complex classification, except for very trivial problems). The identity matrix is **never reached in practice** due to some issues like noise labels i.e. label wrongly assigned maybe due to ambiguity. In particular confusion matrices are **very useful in a multi-class classification problem to inspect what is happening**: may happen that we are predicting correctly a few amount of classes and wrong for the other classes, **reaching a very high accuracy since the dataset is very unbalanced with most of the images belonging to the correctly predicted classes**.

Considering a two-class classification problem the network will tell you the probability to belong to one class (softmax output). Let's consider a binary classification example for fraud detection: if we obtain a result of 48% to be a fraud and 52% to be a regular transaction we should pay attention to say that it is a regular one; **to be cautious we should consider a greater threshold for the classification as a regular transaction** e.g. 90% in a way there is more evidence to be the correct classification. **In many cases in binary classification problem can happen we want to tune the classification by setting an higher classification threshold**. So in case you plan to **modify the threshold of a binary classifier** and not use 0.5 (50%), classification **performance can be measured in terms** of the **ROC (Receiver Operating Characteristic) curve**, which **does not depend on the threshold you set for each class**, looking at the **area under the curve (AUC)**:

the AUC **is a good indicator of how powerful the model is, the higher AUC the better (disregarding the threshold)**.



For each and every threshold the false positive (FPR) and the true positive (TPR) change giving a different point of the curve (each point of the curve is given by considering a different threshold). So the **ideal detector** would achieve:

- **(False Positives Rate) FPR** $= 0\%$
- **(True Positives Rate) TPR** $= 100\%$

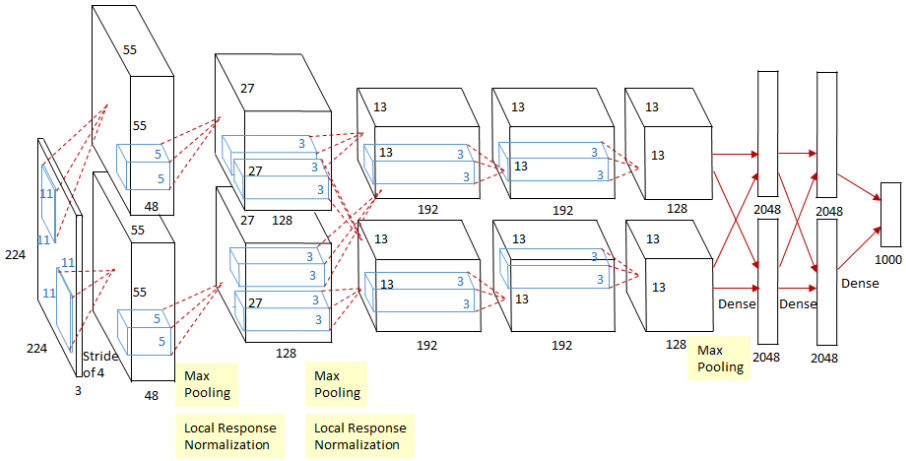
Thus, **as the point of the curve is closer to (0, 1) the better the related threshold is for that classification problem**. The optimal parameter is the one yielding the **curve with the closest point to (0,1)**.

4.3.6 Popular architectures

Yann Lecun's LeNet-5 model was developed in 1998 to identify handwritten digits for zip code recognition in the postal service. This pioneering model largely introduced the convolutional neural network as we know it today. It has 60000 parameters. The idea of using a sequence of 3 layers: convolution, pooling, non-linearity.

- **Pixels in an image are highly correlated**: there is no point to separately feed each pixel value through a fully connected classification network: **add learnable convolutional layers**.
- **Convolution as a way to extract spatial features and to have sparse connections (thus reducing the number of parameters)**.
- **Subsample using spatial average of maps (average pooling)**
- **Non-linearities** such as tanh and sigmoids
- Multi-layer classifier for the final layer.

AlexNet was developed by Alex Krizhevky et al. in 2012 and won the Imagenet competiotion. The architecture is quite similar to LeNet-5:



The subdivision in two part of the network is not part of the architecture, the network need to be **divided for training purposes to be trained on 2 different GPU** (not so powerful at that time). It is composed by:

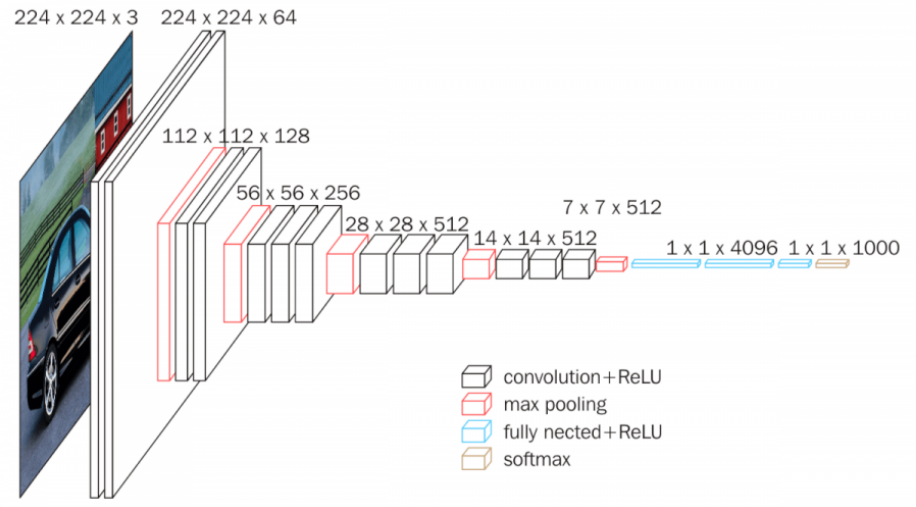
- 5 convolutional layers (rather large filters, 11×11, 5×5)
- 3 multi-layer fully connected neural networks

The input size is $224 \times 224 \times 3$ and the **number of parameters 60 million** (**convolution**: 3.7 millions i.e. **6%** ; **FCNN**: 58.6 million i.e. **94%**). To **counteract overfitting**, they introduce:

- **ReLU** (also faster than tanh for gradient computations)
- **Dropout** (0.5), **weight decay** and norm layers (not used anymore)
- Maxpooling

The first convolution layer has 11×11 filters, stride 4, and that's why the after the first convolution the spatial extention of the image is reduced from 224×224 to 55×55. The output are two volumes of $55 \times 55 \times 48$ separated over two GTX 580 GPUs (3GB, 90 epochs, 5/6 days to train). Most connections are among feature maps of the same GPU, which will be mixed at the last layer. It won the Imganet challenge in 2012. At the end they also trained an ensemble of 7 models to drop error from 18.2% to 15.4% (the classification may be done feeding the 7 trained network with the input and using the average result).

The VGG16, introduced in 2014 is a **deeper variant of the AlexNet** convolutional structure. **Smaller filters are used** ad the network is deeper. The input size is $224 \times 224 \times 3$ and it has **138 million parameters** with **most of them belonging to the dense neural network** (Convolution: **11%** ; FC: **89%**). This architecture won the first place (localization) and the second place (classification) tracks in ImageNet challenge 2014.



The network is much deeper than AlexNet since it **has multiple convolution not separated by a pooling layer that does subsampling**. The paper actually present a thorough **study on the role of network depth**: "**increase the depth** of the network by adding more convolutional layers, which is **feasible due to the use of very small** 3×3 **convolution filters in all layers**". The **idea** is that multiple 3×3 convolution in a sequence **achieve large receptive fields with**:

- **Less parameters**
- **More non-linearities**

than larger filters in a single layer. Due to this fact this is still done nowadays.

	3 layers 3×3	1 layer 7×7
Receptive field	7×7	7×7
N of parameters	$3 \times 3 \times 3 = 27$	49
N of non-linearities	3	1

These two architectures are **not equivalent in term of output since the computations done change due to the presence of non linearity**. But as we can see the **number of parameters is much lower** with multiple small filters and after each convolution layer is placed a **non-linearity** (e.g. ReLU). So this **is very convenient from the learning perspective and from the efficiency perspective**.

When we use this kind of **deep model** we must **keep in mind that they have an high memory request** (***). **Indeed each activation map must be in memory during training**, so about 100MB per image to be stored in all the activation maps, **only for the forward pass (for the backward it's about twice)**.

4.4 Limited Amount of Data: Transfer Learning

You might not be interested in the network used to solve the ImageNet challenge per se, since they where trained on the classification of different categories, but **this network can be used to solve your classification problem using much less training data than you would need training your network from scratch**. So **pre-trained models can be used to solve different problems** (*).

The CNNs architecture is composed by a **data-driven feature extraction** and by a feature classifier. In the **typical architecture** of a convolutional neural network:

- The output of the fully connected layer has the same size as the number of classes L ($1 \times 1 \times L$), and each component provide a score for the input image to belong to a specific class.
- **The input of the fully connected layer can be seen as a data-driven descriptor of the input image**, i.e. **a feature vector (**$1 \times 1 \times N$**)**. These features are:
  - **defined to maximize classification performance** i.e. the best one.
  - **Trained via backpropagation** as the NN they are fed.

  and it is the most important part of the network.

Deep networks are great at extracting (learned) features from images. As we **move to deeper layers**:

- **spatial resolution is reduced (downsampling)**
- the **number of activation maps increases** so the level of the features increases (**low-level** $\longrightarrow$ **mid-level** $\longrightarrow$ **high-level**)
- the **perceptive field increases**.

We **look for higher-level patterns, and don't care too much about their exact location**. There are **more high-level patterns than low-level details since the number of activation maps increases**. **So hopefully the information contained in the feature vector are not too specific for the classification problem that the network was trained for**: **of course the features where optimized for the classification problem presented during training but you expect the features to be general enough in a way they can be portable to solve different classification problem**. Hence in VGG (or AlexNet) network that was **trained to classify 1000 classes from ImageNet, hence the convolutional part should be a very general feature extractor** not to specific for the classification task it was trained for. The **FC layers are instead very custom, as they are meant to solve the specific classification task at hand**. So, **Transfer Learning/Fine Tuning**:

- Take a successful **pre-trained model** such as VGG
- **Remove and modify the fully connected layers for the problem at hand** (reducing the parameter and the hidden layers, if the classification problem is very simple)
- **For training there are two options**:
  - **Train the whole network on the new training data**, having as **starting** point for the convolution layers **weights the value of the pre-trained model** (**Fine Tuning**)
  - **Train only the FC layers "freezing" the weights in the convolutional layers** (**Transfer Learning**)

The technique to choose depends on the amount of data and how close the classification problem that you want to address now is with the classification problem addressed during the training of the pre-trained classification problem (***). Hence:

- **Transfer Learning**: **only the FC layers are being trained**. A good option when **little training data** (e.g. hundreds of samples) are provided and the **pre-trained model is expected to match the problem at hand**.
- **Fine Tuning**: the **whole CNN is retrained**, but the convolutional layers are initialized to the pre-trained model i.e. you have to train an entire model but the **weight initialization is better than random indeed at least you are starting in a region of the parameter space close to the local minimum for the new classification problem** (**changing the dataset change the shape of the error function to minimize but since the pre-trained network was built to solve a more difficult problem the local minimum might be close**). A **good option when enough training data** are provided or when the **pre-trained model is not expected to match the problem at hand**.

Of curse **in both cases the solution is biased** (**more for the transfer learning since the weights of the convolution are kept**) **by the problem of classification**

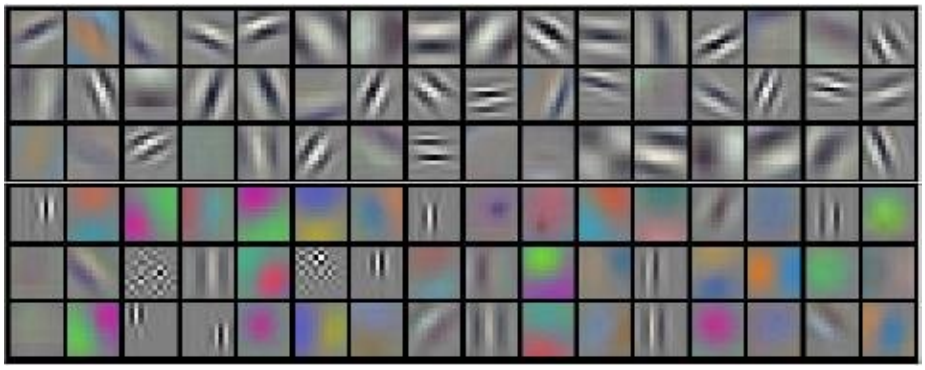
of the pre-trained model but since **we are taking only the feature extractor part, whose number of parameters is very small w.r.t. the number of parameter of the fully connected layers, the bias is acceptable and having enough data the training can be successful to solve the classification problem**.
Another **technique** is to mix Transfer Learning and Fine Tuning **"freezing" the weights of only some of the convolutional layer at the beginning of the pre-trained network**. Indeed the convolution layer, closer is to the FC layers **more high-level and classification specific are the feature extracted, so it make sense to change those even if more parameters are involved** (***). Instead **low-level features are not related to the classification problem since represent very local features as line, edges** (*). This approach is **useful if the condition of training written above are not fully coinciding with the one of fine tuning or transfer learning**, for example if the classification problem is slightly different from the one of the pre-trained network.

The point of transfer learning and fine tuning is the fact that we are using a pre-trained model to solve much easier classification problem, such that the number of parameter of the new FC layers may be much less than the number of parameter of the convolutional part. So **this can be particularly meaningful with data scarcity**: **less parameters can be trained with less data and a small number of parameters is less prone to overfitting with a small amount of data**.

4.4.1 CNN visualization

Now let's see how to **infer what a CNN network has learnt**.

We can visualize the CNN filters. For example the $11 \times 11 \times 3$ filters (visualized in RGB) extracted from the first convolutional layer are the follwoing:



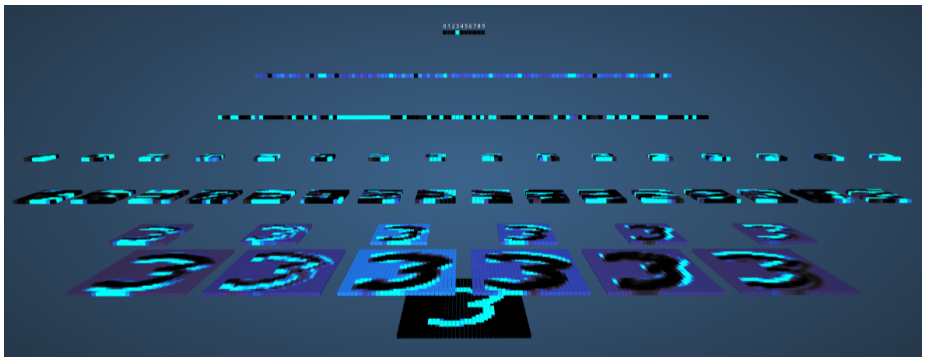
All of them are **detecting basic low level pattern (as edges)** and should be notice that **most of them are grayscale**: being grayscale **means they treat all the color the same, hence the pattern being detected from the first layers are manly related to the image intensity rather than the colour**, so the **response**

will be the same whether the lines are red, green or blue, i.e. is just a matter of intensity. On the other hand there are **some filters that detect colour pattern that detect boundaries between the colour showed in the filter**: the boundary between the two colour is positioned in the grey zone of the filter so it can match various type of boundaries like straight lines with different inclinations. Hence either you get very specific details in image intensity or very low resolution variation in the colour i.e. the first layer seems to match **low-level features** such as edges and simple patterns that are discriminative to describe the data. Furthermore and important thing is that **there aren't many differences between the first layer filters of different CNN with different filter sizes, they are learning the same low level patterns e.g. edge detection and colour detection**.
Repeating the experiment we did for the first layer filters we find out that **more deeper we go more difficult becomes to interpret the pattern the filter are matching**:

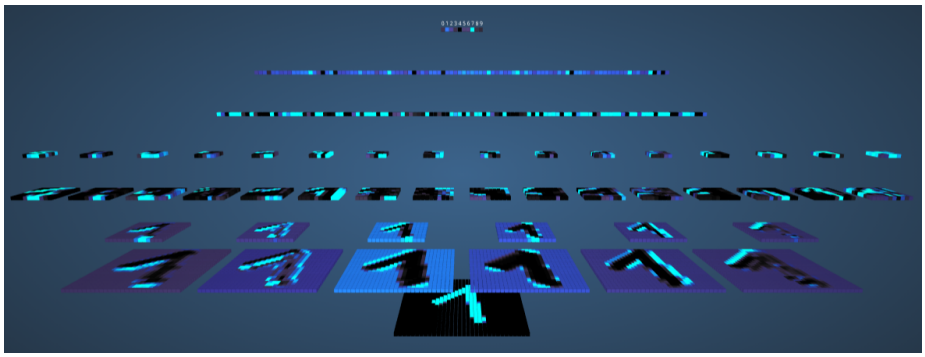
- **filters does not have just 3 channels but they have a lot more so they cannot be interpreted as coloured images**.
- they are **no more detecting patterns we are used to, since they are not operating over the original image but they are operating over the activation maps that are the output of the convolution over the previous filters**.

So while we can recognize patterns over the filters of the first layer since we know how the images looks like, then we cannot do the same thing on **deeper layer's filters because they are used to match the pattern on activations that we don't know how they look like**.

Hence another way to determine what the deepest layer are learning is required. What if we look at the activations? **Can the activation tell us something about what the network is learning?**



The image show the layer-wise representation of the output volumes i.e. activation maps. We can see as some of the layer activates (blue) and some others don't (black). Obviously changing input change also the activation maps:



The **different activation would lead to a different feature vector fed to the FC layers**. So if we want to analyze the features matched at deeper layer (than the first one) we should **visualize the input that maximally activates an activation map neuron, that is equivalent of telling what the correspondent filter is detecting**. This can be done by looking for the input that maximally activates some (maybe to get a better result) neurons, and this means those neuron are very sensitive to that kind of input i.e. the filter applied is maximally matching (i.e. matching very well):

- (Step 1) **Compute the gradient of the considered neuron value with respect to the input**. We can always compute gradient between values in a CNN since all the operations are known: indeed the function to derive is a concatenation of multiple convolution-ReLU-MaxPooling. The gradient can be computed not only for training loss.
- (Step 2) **Change the input accordingly, such that the activation of the considered neuron will increase**. Hence we perform **gradient ascent**: we **modify the input in the direction of the gradient, to increase our function that is the value of the selected pixel in the activation map**.
- **Iterate the procedure to modify the input**. Since the input may not look as an image, **some form of regularization can be added to the selected pixel value to steer the input to look more like a natural image**.

This operation can be done for activation maps at each depth. This is not a visual intuition like analyzing the filters but **can show us to which kind of input the activations maps are recognising**. ATTENTION, that **are not the filter themselves but the input image that at that point the filter is recognising from the input and that maximize the activation**, indeed those are the image recognized after some convolution so are not the value of the filter itself, **is the concatenation of filters that recognize those patterns**. So is the image that mostly exites a specific pixel of a specific activation map.
For example, these are the filters i.e. the pattern matched in the image that maximize the activation of all the neurons of the activation map, at different depth level, from the VGG16 network:

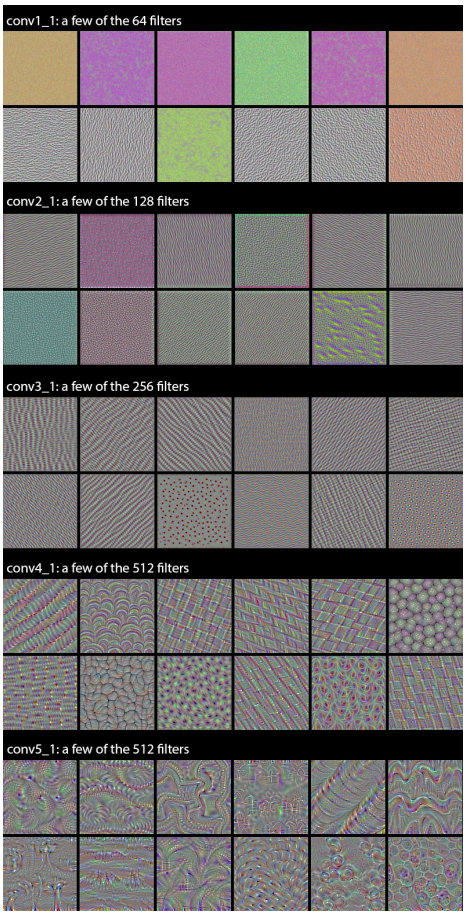


Figure 21: Visualization of Convolution Filters. With the increase of depth in the convolutional neural network using convolution layers and pooling layers, the pattern searched by convolution filters becomes larger in scale and tend to be more sophisticated

Looking at the image is clear how:

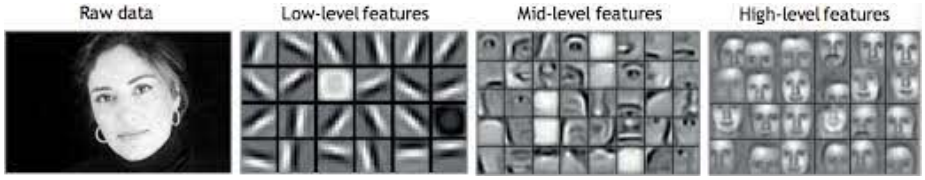
- **Shallow layers** respond to **simple, fine, low-level patterns**. The first layers basically **just encode direction and color**.
- In **intermediate layers** the direction and color filters then get combined into basic grid and spot textures. These textures gradually get **combined into increasingly complex patterns**.
- **Deep layers respond to complex, high-level patterns**.

Going to shallow layers to deep layer filters we can notice how **the level of the features become higher**: is **like they are zooming out, matching less local**

pattern i.e. the receptive field is higher.
You can **think of the filters in each layer as a basis of vectors**, typically over-complete, **that can be used to encode the layer's input in a compact way**. The filters become more intricate as they start incorporating information from an increasingly larger spatial extent.

A remarkable **observation**: a lot of **these filters are identical, but rotated by some non-random factor (typically 90 degrees)**. This **means** that we could **potentially compress the number of filters used in a convnet by a large factor by finding a way to make the convolution filters rotation-invariant**. Shockingly, the rotation observation holds true even for relatively high-level filters, such as those in block4_conv1. In the highest layers (block5_conv2, block5_conv3) we start to recognize textures similar to that found in the objects that network was trained to classify, such as feathers, eyes, etc.

Why does CNNs work? Convolutional neural networks **learn a hierarchy (low level patterns $\rightarrow$ high-level patterns) of translation-invariant spatial pattern detectors**.



4.4.2 Fully Convolutional Networks

What happens when we feed the network with images of different size?
CNNs are meant to process input of a fixed size (e.g. 200×200). Indeed the **convolutional and subsampling layers operate in a sliding manner over image having arbitrary size but the fully connected layer constrains the input to a fixed size**. But specifically, what happens when we **feed a larger image to the network?** Convolutional filters can be applied to volumes of any size, **yielding larger volumes in the network until the FC layer**. The **FC network however does require a fixed input size**, thus CNN cannot compute class scores, yet can extract features. Indeed since the volume **at the end of the feature extraction is bigger than it suppose to be** (e.g. $M_1 \times M_2 \times N$ **instead of** $1 \times 1 \times N$**, with** $M_1, M_2 > 1$), which means that the **flattening will produce a vector of greater size than the one is designed for** (the input size of a FC neural network, as said, is fixed).

However, since the **FC is linear** (**if we consider the linear activation function**), it **can be represented as convolution**: the weights associated to output neuron o_i are:

$$w_i = \{w_{ij}\}_{i=j:N}$$

so the value of the processed by the output neuron o_i is:

$$o_i = \sum_{j=1}^{N} w_j^i \cdot s_j + b_i$$

but this is **a linear combination as the convolution**, so I can **apply that as a filter on the input of whatever input size**, i.e. **even if the input is not flat** $(1 \times 1 \times N)$ **but has some spatial extension** $(M_1 \times M_2 \times N)$.

$$o_i = (w^i)' \cdot s + b_i$$

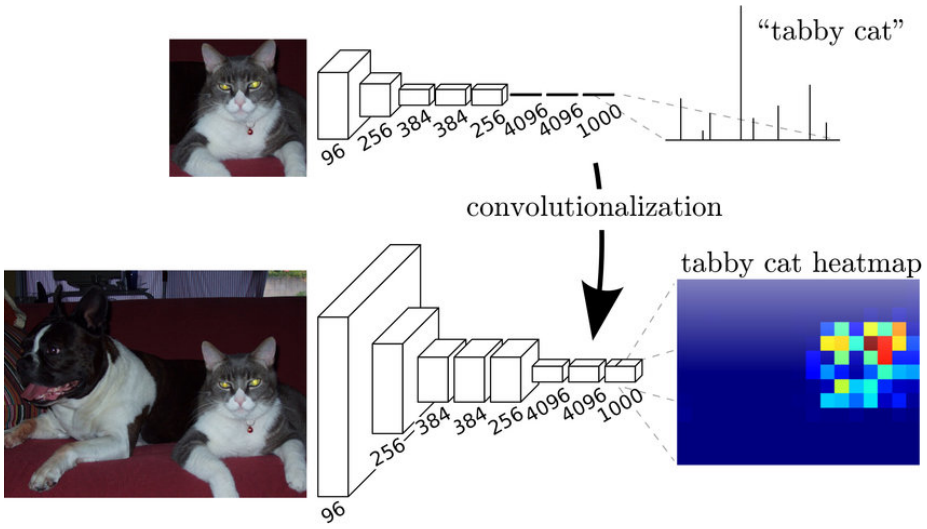
$$o_i = (w_i \otimes s)(0,0) + b_i$$

where the $\otimes$ **is a convolution that is done with a filter whose dimension is equal to the one of the processed input** $s = \{s_i | i = 1, ..., N\}$ having size $1 \times 1 \times N$, **i.e. a filter of** 1×1 **for each input**; since they **have the same dimension the filter must be centered in** $(0,0)$**, i.e. returns a single value** (instead **FC-CNN return an heatmap i.e. an image with spatial extent**). Hence a FC layer of **L outputs** o_i is a **2D Convolution layer against L filters** w_i **having size** $1 \times 1 \times N$ (xN since we have to remember that a filter is applied to the entire volume and then summed up), whose value are the one of the weights that connect each input to the output neuron o_i. **This transformation from FC layer to 2D convolution layer can be applied to each hidden layer of a FC network placed at the CNN top**. However, since the FC is linear, it can be represented as convolution against L filters (i.e. L outputs) of size $1 \times 1 \times N$. Each of these convolutional filters contains the weights of the FC for the corresponding output neuron. So by convutionalize each dense layer of the FC network **considering the same activation function after the convolution**, the entire network **can** be applied **also to process images of different size**: the output you will get is the same you would obtain with the dense layer since is a 1:1 mapping that allow to manage different size. The 1×1 convolution let us to adjust the depth size of the volume (features) that is resized according to the size of the next neuron layer adding non-linearities until the output layer where the features would be as the same number of the classes. This kind of networks where **the FC layers are transformed into a set of** 1×1 **convolutional filters i.e. into a convolution with a** 1×1 **filter** (the $\times N$ is not specified since it depends on the input size of the dense layer, so it can be higher or lower) are called **fully convolutional neural networks** (FC-CNN). **Performing a** 1×1 **convolution does not mix the input pixels but it just compute a score, the classification output, out of the depth dimension**. **For each output class we obtain an image, having**:

- **Lower resolution than the input image**
- **Class probabilities for the receptive field of each pixel**

The output of a FC-CNN instead of a vector of values as for a normal CNN can be seen as a vector of images, but more specifically **heatmaps**. **Each** heatmaps $(M_1 \times M_2 \times L)$ **correspond to a class of the output** (i.e. to a output

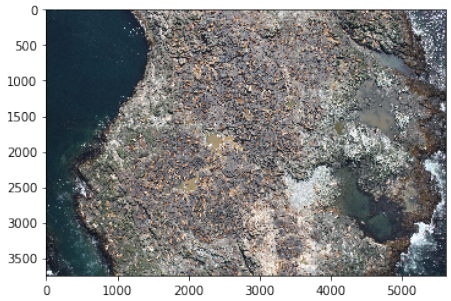
neuron) and **each pixel of the heatmap corresponds to a receptive field in the input image**. **The value of the heatmaps pixel can be seen as the probability of having an object of the correspondent class in the receptive field of the pixel**. The heatmap's pixel value can be considered as a **kind of seed for image segmentation or object detection**.

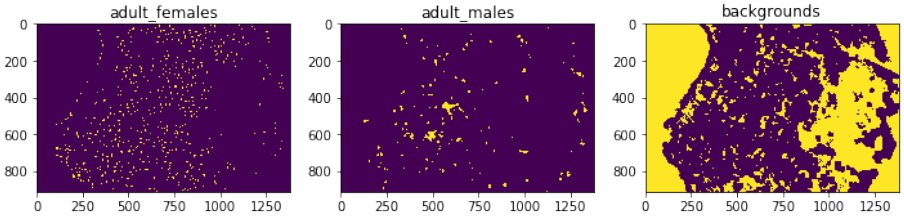


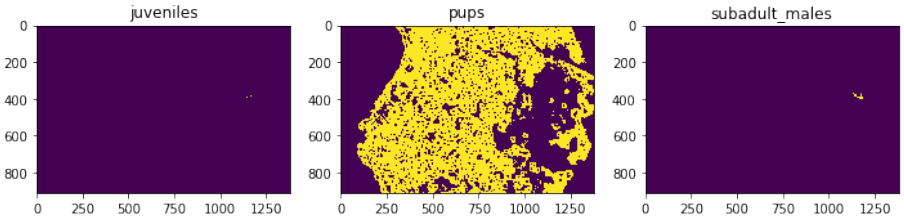
Note that the receptive field of neighbour heatmap's pixel overlap substantially giving close value in the region where the object is identified by the network as it is for the cat above.

NOTE changing the scaling of the object to detect in the image would cause some scaling problem since the network may not cope with the scale change. Some improvements may be achieved using some data augmentation. This is true in general for all the things that the network is not trained for.

To make a fully convolutional network what is done is to **migrate from a pretrained model to a FC-CNN, getting the weights of the pretrained model and setting them to the FC-CNN architecture (where FC layers are transformed into convolutions)** after they are reshaped to become a convolution filter: **the number of convolutional filters depends on the number of neurons of the dense layer**, so $(1 \times 1 \times N) \times L$ convolutions where N is the number of neuron that has the input layer and L is is the number of neuron that has the output layer). **The stack of convolutions operates on the whole image as a filter**. **Significantly more efficient (i.e. huge computational saving) than patch extraction and classification (avoids multiple repeated computations within overlapping patches, that make the computation much more demanding since a lot of values are computed twice)** like we would have done on a 4k image in which the network should be able to detect and classify objects like sea lions types.







Remember that there is a heatmap for each of the classification output and each of the pixel is the probability of having an object of that class in its receptive field.

To recap **FC-CNN make possible to feed the network with image of different sizes by transforming the FC layers of the network into convolutions, that do not have the problem of needing an input of fixed dimensions, but perform the same operation**. **Instead of having a prediction for each class the FC-CNN produce a heatmap related to a certain class in which each pixel value is proportional to the probability of finding a object of that class in the receptive field** (since by keeping the same activation function the value of each pixel is exactly the softmax class probability). The **heatmaps are of course of low resolution because through the network we perform a lot of downsampling** and present similar intensity in the location around the object of the class since the receptive fields of the adjacent pixels overlap. This **heatmaps can be the starting point for object detection and object segmentation algorithms**.

5 Convolutional Neural networks for Semantic segmentation

5.1 Semantic segmentation task

The **goal of the semantic segmentation** is:

- **Given an image** I, **associate to each pixel** (r, c) **a label from** Λ **i.e. the class/categories set**
- The **result** of segmentation **is a map of label containing in each pixel the estimated class**.

Remark in the image **there is no distinction among objects of the same class**, for example lamp 1 and lamp 2. **Segmentation does not separate different instances belonging to the same class. That would be instance segmentation**.

This task is typically performed **training** the network **in a supervised manner**: the training set is made of pairs (I, GT), where **the ground truth (GT) is a pixel-wise annotated image over the categories in** Λ. Training set are very difficult to annotate (e.g. Microsoft COCO dataset provides 200000 annotated images over 80 categories, and human joints are annoated as well)

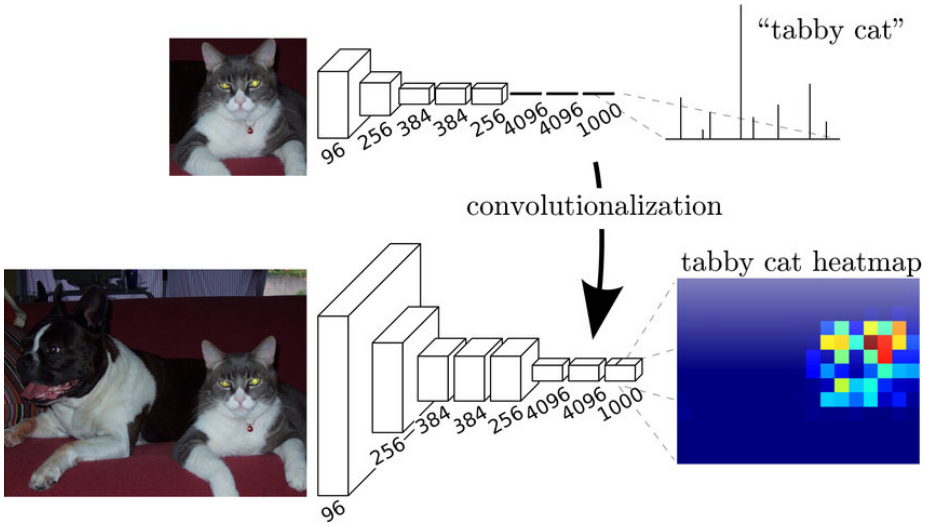
I *GT*

Before the use of neural network the image segmentation was done in an unsupervised in a sort of clustering looking at pixels with similar characteristic/intensity.

5.2 Fully Convolutional Neural Networks for Semantic Segmentation (J. Long)

There are different solutions to **predict the semantic segmentation for arbitrary-sized inputs**:

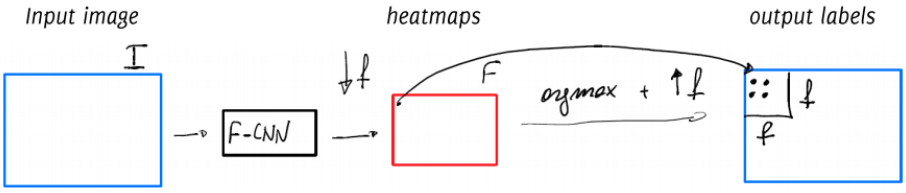
(i) The **simplest** solution is to have **direct heatmaps predictions** assigning the predicted label in the heatmap to the whole receptive field, however that would be a **very coarse estimate**. This, as we have seen, can be done transforming the network in a fully convolutional one.



So this method uses a F-CNN to obtain the segmentation and design an effective **scheme in order to go back to the original full resolution through**

upsampling.

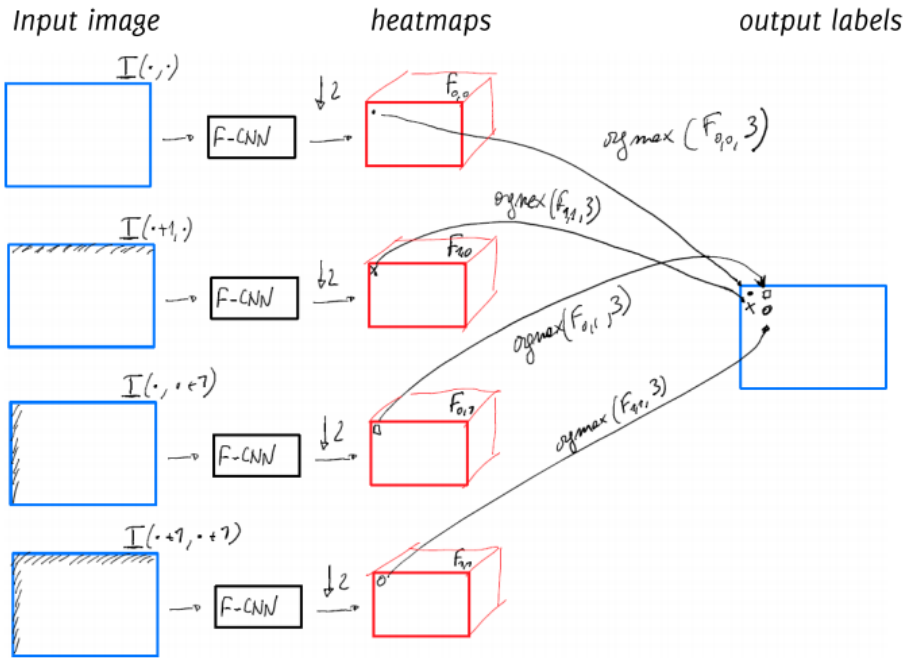
To obtain the segmentation we should compute the pixel-wise argmax (i.e. for each pixel $1 \times 1 \times L$, we compute the argmax taking the class corresponding to the heatmap to which belongs the pixel with the maximum value) throughout the whole volume of heatmaps (output of the F-CNN) to associate to each pixel the most probable class **obtaining a segmented image $m \times m$ (with $m << n$ i.e. at a very low scale, where I is $n \times n$, due to the downsampling $\downarrow f$) in which each pixel value is the index of the most probable class**. Over the latter output is done an upsampling ($\uparrow f$), so that the label set in the low scale output of the F-CNN is associated to a $f \times f$ region and not to a single pixel. **The upsampling factor ($\uparrow f$) is the exact opposite of the downsampling factor ($\downarrow f$) such that at the end the dimension of the input image are reconstructed**.



This type of upsampling from 1×1 to $f \times f$ region lead to a coarse (blocky) segmentation estimate, indeed it just repeat the class prediction over the $f \times f$ area.

(ii) Another **simple solution**, a little more advance, is the **shift and stich**. It consists in **assuming there is a ratio f between the size of the input and of the output of the heatmap** (after the argmax operation):

- Compute heatmaps for all f^2 **possible shifts of the input** $(0 \leq (r, c) < f)$ i.e **compute f^2 heatmaps instead of a single one** (the one after the argmax).
- Map predictions from the f^2 heatmaps to the image: each pixel in the heatmap provides prediction of the central pixel of the receptive field.
- **Interleave the heatmaps to form an image as large as the input**.



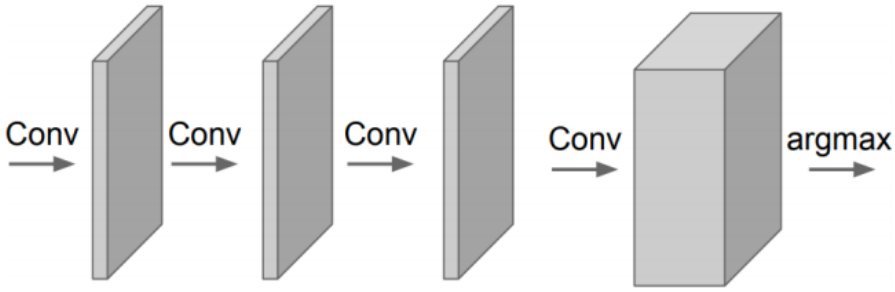
In the last image the input is shifted respectively of: (0, 0), (0, 1), (1, 0), (1, 1) **since the downsampling/upsampling factor is 2 i.e.** $2^2 = 4$ **heatmaps i.e. shifts**. By shifting the image the content of the receptive field of the same pixel is different for each of the f^2 heatmaps. Furthermore, in argmax(F, 3) the 3 indicate that the argmax is done in the third dimension i.e. depth (pixel per pixel depth-wise).

This exploits the whole depth of the network. Can be efficiently implemented through the à trous algorithm in wavelet. However, the **major drawback** is that the **upsampling method is very rigid**: just aggregating output in a copy paste way is not optimal since **there is no learning in the way heatmaps are aggregated**.

(iii) Another simple solution is to **use only convolutions**. Indeed **avoiding any pooling (downsampling)**, i.e. just 2D convolutions and, after a certain number of convolution, activation layers, we obtain:

- **Very small receptive field** since we eliminated the downsampling, and by doing so even the **upsampling is not required since the output will be exactly of the size** of the input. **To obtain a larger receptive field are needed plenty of convolutions but then the problem becomes the huge number of operation to perform**, i.e. the number of parameters increase. **Very small receptive field**.
- **Everything is learnable**, and this is a good thing **but the things to learn do not have to be too much i.e. very deep**.

- **Very inefficient because you never reduce the spatial extent and by increasing the depth you increase the number of covolution needed at each stage as the volume increase in size without reducing the spatial extent**.



where the **argmax is still applied pixel-wise in the depth dimension**. So the drawbacks of using convolutions only, are that:

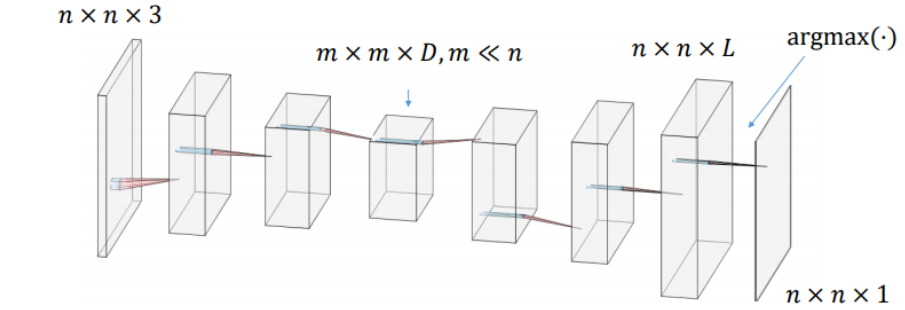
- **on the one hand we need to "go deep" stacking convolution and non-linearities to extract high level information** (i.e. complex, articulated features) on the image, **i.e. information on a large input region of the image**.
- **on the other hand we want to stay local not to lose spatial resolution in the predictions, to obtain more precise segmentation predictions**.

Semantic segmentation faces an inherent **tension between semantics and location**:

- **Global information (depth) resolves what i.e. classification task**
- **Local information (locality, shallowness) resolves where i.e. exact pixel location**

Hence **combining fine layers and coarse layers** lets the model make local predictions that respect global structure.

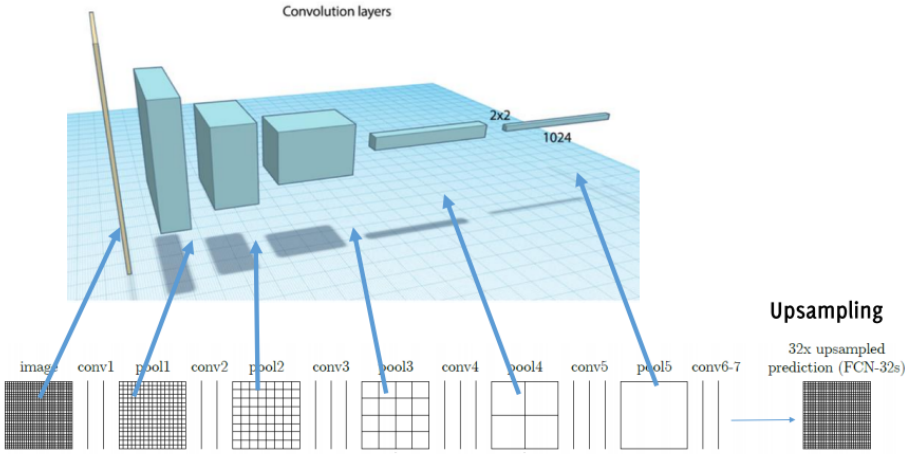
The **real solution** is to **reduce the latent representation dimension**. An architecture like the following would probably be more probably be more **suitable for semantic segmentation**:



The first half is the same of a CNN for classification involving convolution and downsampling (i.e. pooling, reducing spatial extension) to obtain a large receptive field, instead the second half is meant to upsample the predictions to recover the input image spatial extension (possibly reducing the depth), in order to cover each pixel in the image. Increasing the image size is **necessary to obtain sharp countours and spatially detailed class predictions**.

- **Deep features** (after the first half) of the network **encode semantic information**.
- **High resolution output** (after upsampling) **recovers local information**.

With this architecture is possible to to take a pre-trained neural network for the segmentation part and then add a second part or train both part together.

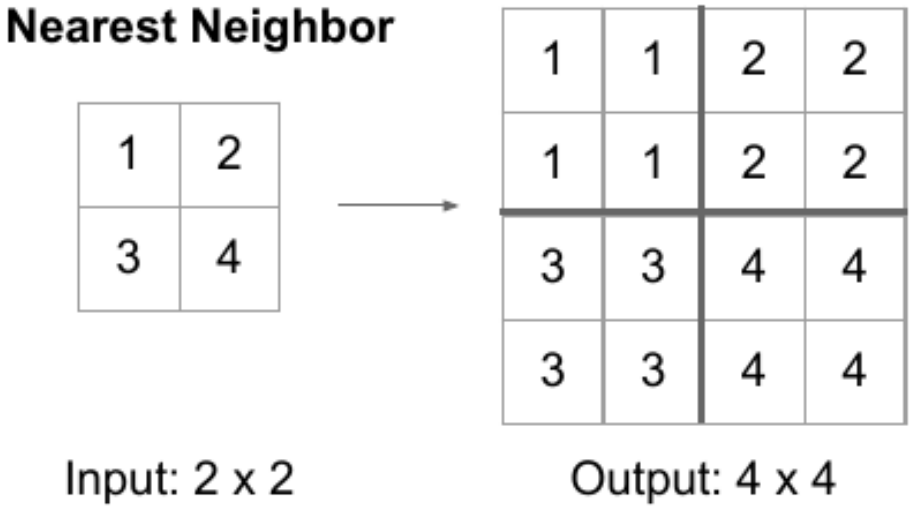


In the last image the dimension of the grid becomes bigger and bigger **as the depth increases** to indicate that the **receptive field increases its spatial dimensions** i.e. the spatial resolution of the volume decreases. In the paper from which

the image is taken the downsampling factor is of 32. Arrived at the conv6-7 i.e. the bottleneck of the architecture (as in the picture above), we want to restore the spatial extension of the input image by performing upsampling.

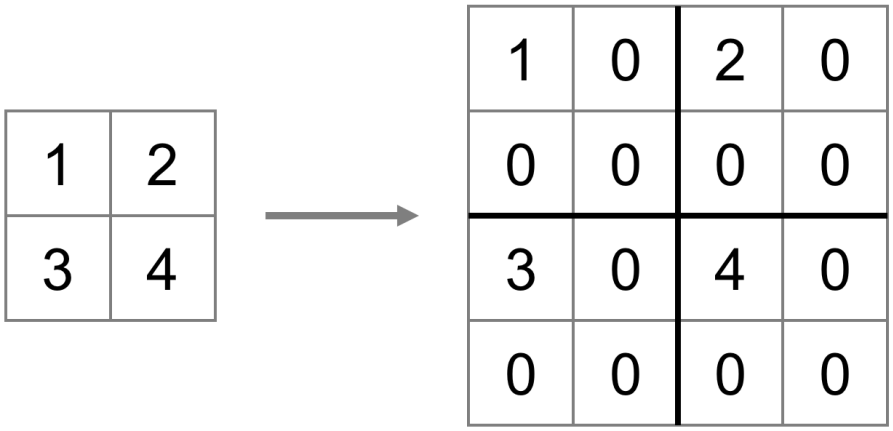
To perform **upsampling** there are **different techniques**:

- **Nearest neighbour** perform upsampling **without any learning process**. This is the same approach used in the direct heatmap prediction technique seen before.



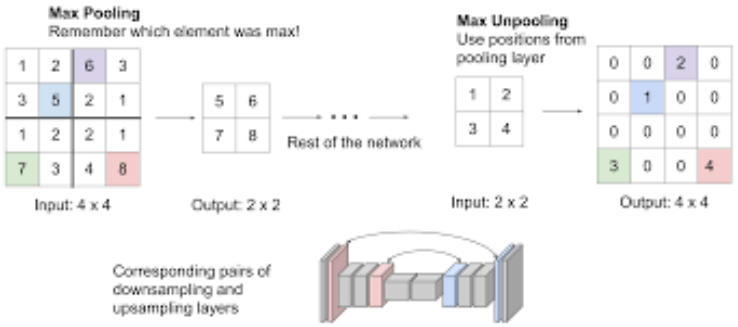
  This is **not a very successful way to perform upsampling since as we have said it gives a coarse estimate**.

- **Bed of nails**



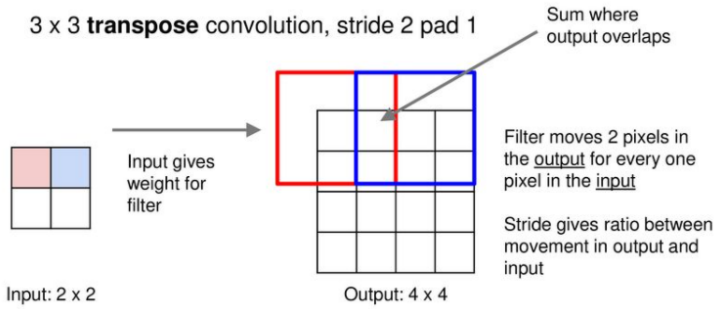
But as we could expect **does not help much as the previous method** to obtain a good estimate.

- **Max unpooling**. This method **require to keep track (i.e. store) of the locations of the max during maxpooling**.

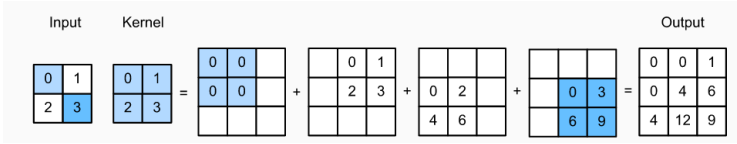


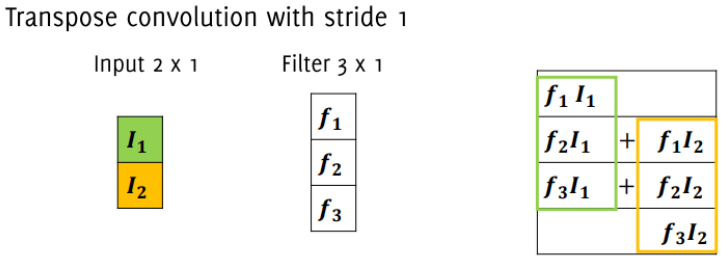
Hence **must be created a (symmetric) correspondance between pooling and unpooling so that during the unpooling the location of the maximum is the correct one and the rest values to recover spatial extent are set with bed of nails technique**. Note that during the upsampling the values are not the same we had during the downsampling indeed the latter are features extracted by the contracting path (downsampling network).

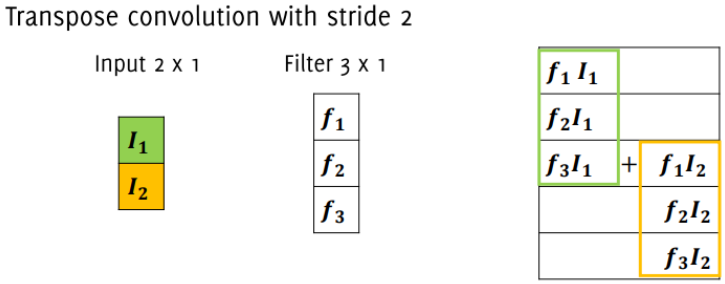
- **Transpose convolution**. This method can be seen as **a traditional convolution after having upsampled the input image**.



In the latter picture **the red pixel of the input is multiplied for the** 3×3 **weight** (i.e. the input pixels are coefficient to scale the filter values), **and then placed where the red square is**, and the same is done with the blue pixel. The **overlapping output values are summed**. The input image in this way is **upsampled of a factor given by the stride of the filter**:



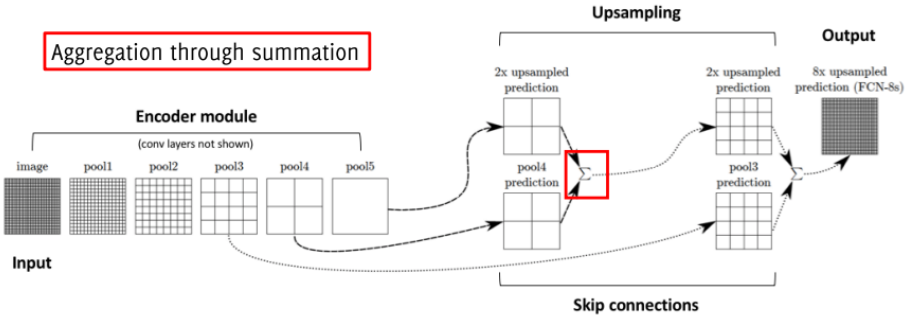




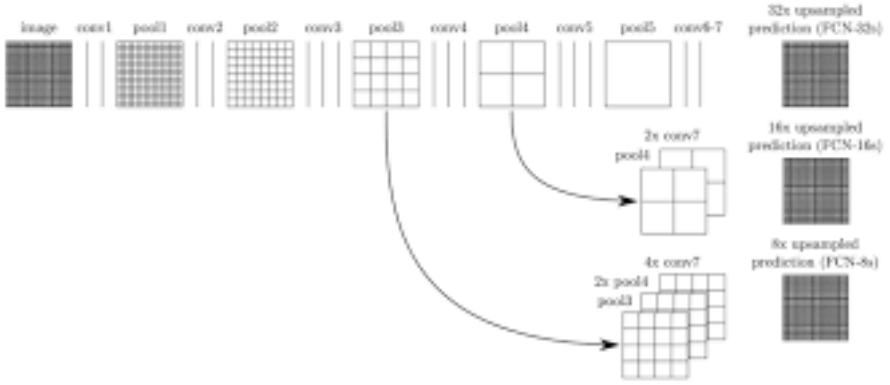
There are many names for transpose convolution: fractional strided convolution, backward strided convolution, deconvolution (very misleading). **Upsampling based on convolution gives more degrees of freedom, since the filters can be learned**.

- **Prediction upsampling**. This is **a linear upsampling of a factor f that can be implemented as a convolution against a filter with a fractional stride $\frac{1}{f}$**. Upsampling filters can thus be **learned during network training** and are learned with initialization equal to the bilinear interpolation. These **predictions however are very coarse, not as blocky as nearest neighbour but do not have fine details**.

The **solution the the coarsity of the predictions**, including some high resolution details, is to **skip connections aggregating the result through summation**.



We **skip connection to use** higher resolution volumes (i.e. **volume with higher spatial extent close to the input**) **useful for locality together with** upsampled lower resolution volumes (i.e. **volumes with lower spatial extent coming from deeper areas in the network**) **useful for high level features** (global information since some details are lost). Observe that **there is no argmax**. Hence the skip connection let us to aggragate (sum) upsampled prediction with prediction coming from a skip connection that has the same spatial extent of the upsampled prediction; **this can be done with informations at any level of spatial extent in the encoder module**.

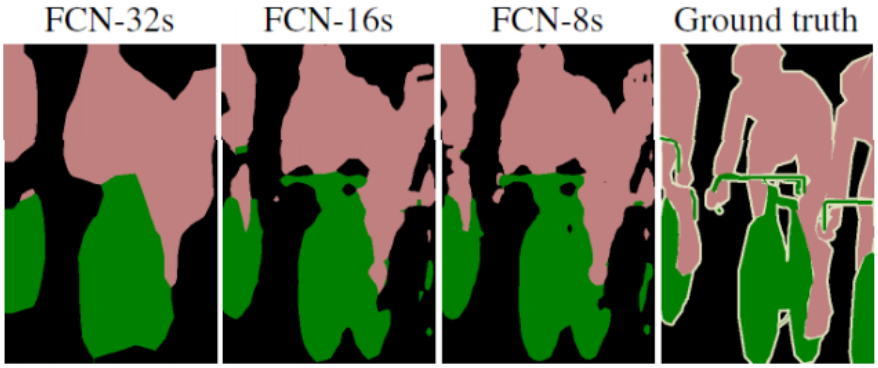


- Supplement a traditional "contracting" (downsampling) network by successive layers where convolution is replaced by **transpose convolution**.
- Upsampling is ubiquitous, are very present in the network.
- **Upsampling filters are learned during training**.
- **Upsampling filters are initialized using bilinear interpolation, since is the best initialization for that task**.
- This yields 3 networks. **Combining i.e. aggregating fine layers (shallow) and coarse layers (deep) lets the model make local predictions that**

respect global structure. I.e. global iformation are combined with local information.

- **Train first the lowest resolution network (FCN-32s) without any skip connection.**
- **Adding the first skip connection** at pool4 and initializing the weights of the next network (FCN-16s) are with the one of the previous one (FCN-32s) **and train again**.
- The **same** for FCN-8s
- **All the FC layers are converted to convolutional layers** 1×1.

So through the **skip connection** that preserve the initial spatial extent and the **upsampling layers that we are learning the network** is capable of a **finer segmentation i.e. the segmentation error reduces**. Retaining intermediate information is beneficial, the deeper layers contribute to provide a better refined estimate of segments.



There are **several F-CNN training options**:

- The **patch based** way (**classical way for a segmentation problem**):
  - Prepare a training set for a classification network.
  - **Crop as many patches x_i from annotated images and assign to each patch, the label corresponding to the patch center**.
  - Train a CNN for classification from scratches, or fine tune a pre-trained model over the segmentation classes.
  - Convolutionalization: once trained the network, move the FC layers to 1×1 convolutions.
  - Design the upsampling side of the network and train these filters.
  - The classification network is trained to minimize the classification loss l over a mini-batch:

$$\hat{\theta} = min_\theta \sum_{x_j} l(X_j, \theta)$$

where x_j belongs to a mini-batch.

- Batches of patches are randomly assembled during training. So can be applied also data augmentation.
- It is possible to resample patches to solve class imbalance.
- It is **very inefficient, since convolutions one overlapping patches are repeated multiple times. We are performing the same operation over same pixels (overlapping ones)**.

- The **full image way**: Since the network provide dense predictions (i.e. for each output), it is possible to **directly train a FCNN that includes upsampling layers as well, giving as a target the segmentation map**. Learning becomes:

$$min_{x_j \in I} \sum_{x_j} l(x_j, \theta)$$

where x_j are all the pixels in a region of the input image and not over a mini-batch. The **loss is evaluated over the corresponding labels in the annotation pixel by pixel**. **Therefore, each patch provides already a mini-batch estimate for computing gradient**, that is **composed by the pixels and not a mini-batch composed by some patches as we have seen before**.

  - FCNN are **trained in an end-to-end manner**: it takes the input image and predict directly the segmented output S(·,·) where each pixel is labeled. You can **still embed a pre-trained neural network but then you have to train all the neural network at once since the training should be end-to-end to learn the segmented output correctly**.
  - This **loss is the sum of losses over different pixels**. **Derivatives can be easily computed through the whole network, and this can be trained through backpropagation**.
  - No need to pass through a classification network first. You can still embed a pre-trained neural network but then you have to train all the neural network at once since the training should be end-to-end to learn the segmented output correctly.
  - **Takes advantage of F-CNN efficiency, does not have to re-compute convolutional features in overlapping regions**.

  Limitations of the full-image taining **and solutions**:

  - **Mini-batches in patch-wise training are assambled randomly i.e. there is some stochasticity in the way the gradient is computed. Image regions in full-image training are not changing randomly, so at each epoch we would take always the same mini-batch (image pixels) without any randomicity in the mini-batch selection**. To make the estimated loss **a bit stochastic, adopt random mask**:

$$minimize \sum_{x_j} M(x_j) \cdot l(x_j, \theta)$$

being $M(x_j)$ a binary random variable. This approach is **similar to a dropout layer**: the binary mask sets randomly its pixels to 0 and 1 **presenting each time a different mini-batch of pixels**, introducing some randomicity when we are assessing the loss. This **has a regularizing effect similar to the one of data augmentation**.

- **It is not possible to perform patch resampling to compensate for class imbalance**. One should go for the **weighting the loss over different labels**:

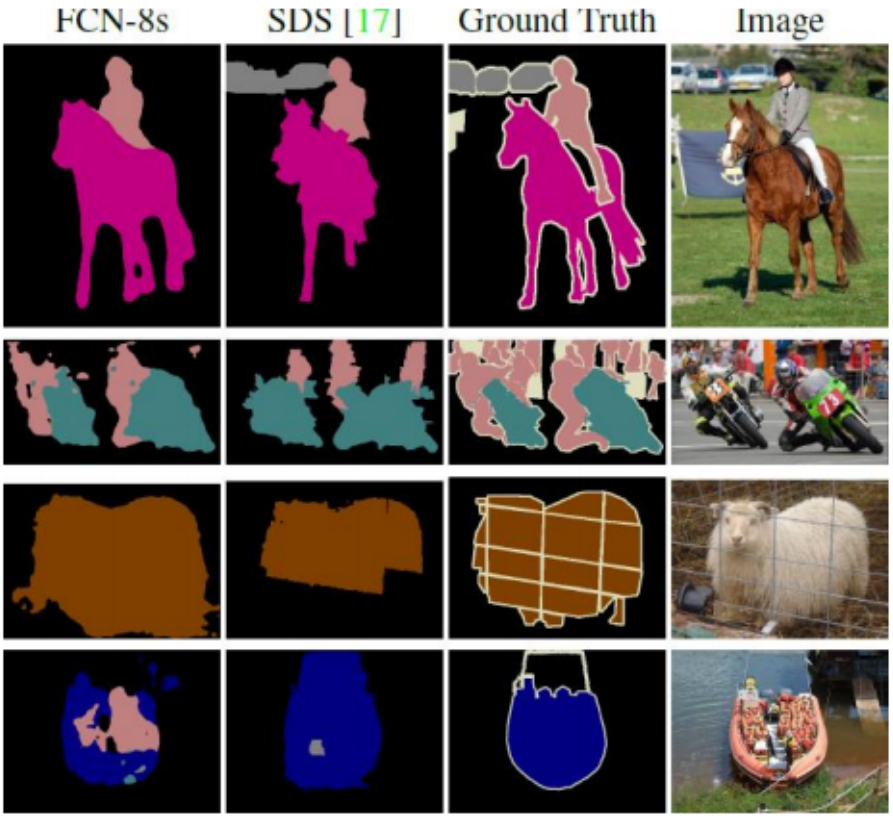
$$minimize \sum_{x_j} w(x_j) \cdot l(x_j, \theta)$$

being $w(x_j)$ a weight that takes into account the true label of x_j.

Some observations and comment:

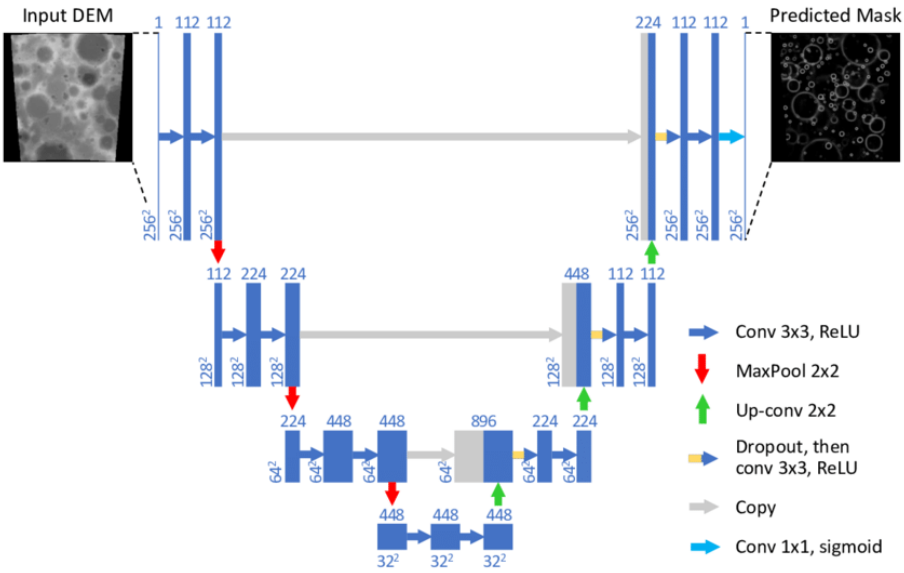
- **Both learning and inference can be performed on the whole-image-at-a-time**.
- Both in full-image or patch-training it is **possible to perform transfer learning/fine tuning of pre-trained classification models** (segmentation typically requires fewer labels than classification)
- **Accurate pixel-wise prediction is achieved by upsampling layers**.
- **End-to-end training is more efficient than patch-wise training**.
- Outperforms state-of the art in 2015
- **Being fully convolutional, this network handles arbitrarily sized input**.

In the following image the result of a semantic segmentation:



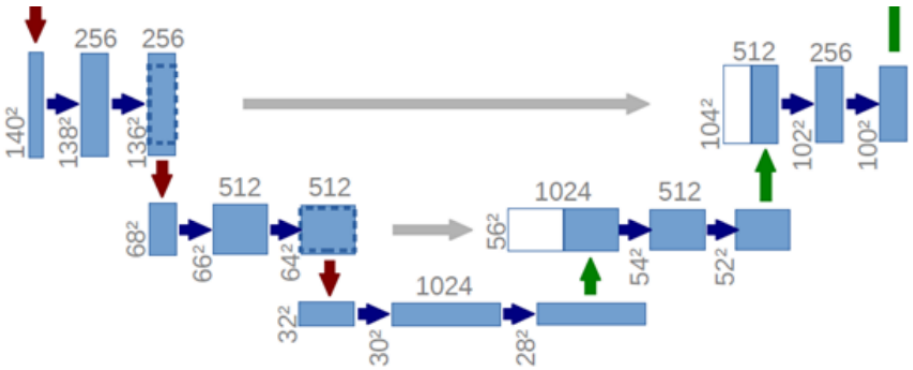
5.3 U-Net: convolutional networks for biomedical image segmentation (O. Ronneberger)

Its name comes from its shape:

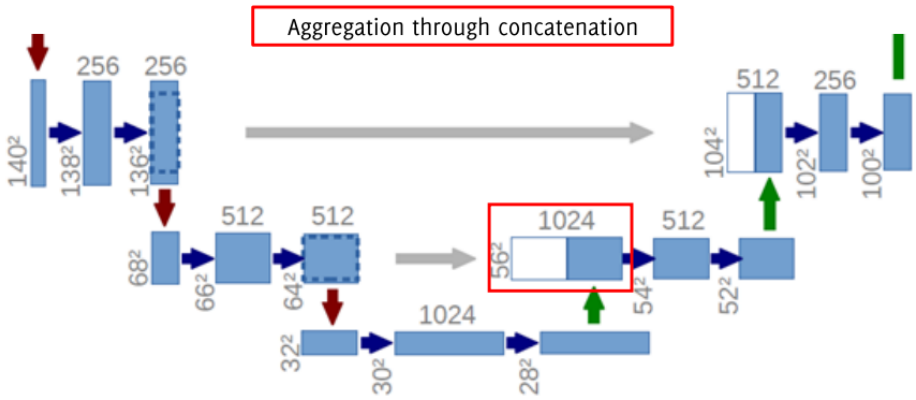


The network formed by:

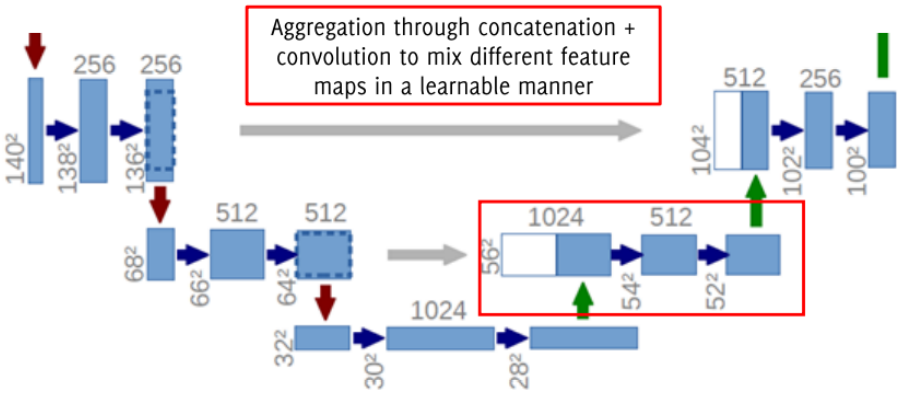
- A **contracting path** (**downsampling**) i.e. the left part of the U. The **spatial extent decreases and the depth increases**.
- An **expansive path** (**upsampling**) i.e. the right part of the U. The **spatial extent increases and the depth decreases**.
- **Skip connections** (horizontal arrows, copy and crop) that connect high level information from the lower level of the network to the corresponding level of the expanding path.
- **There isn't any fully-connected layer**.
- Arrived to the **bottleneck** the network **switch from contracting path to expansive path**. The **upsampling filters** are the **same number** as the one of the corresponding **downsampling block**.



But the **convolution block in the expansive part reduce the depth of the input volumes**, indeed it processes not only the upsampled volume from the previous expansive convolution block but the **aggregation** of it and the output volume of the corresponding convolution block in the contracting path:



From the latter image we can see how **aggregation is not done through summation** as we have seen before **but through volume concatenation** (this happens often), since **this enrich the operation**. Furthermore, we can observe that the volume that skip the connection, and then aggregated is not taken in its full spatial extent but only the central part, indeed the expanding path do not restore the initial input spatial extent, so the upsampled volumes in the expanding path have a smaller spatial extent than the downsampled volume in the corresponding block in the contractive part.



So the volume coming from the skip connection is simply **concatenated** depth-wise to the output of the upsample layer, that is different from the concatenation used by the F-CNN et al Long. In this way the **information from both paths are conbined**.

- The **output of the network comes from the very last convolution layer** with 1×1 filters ($(1 \times 1 \times 64) \times 2$), i.e. a **convolutionalized FC**, and are two since it was a binary classification.

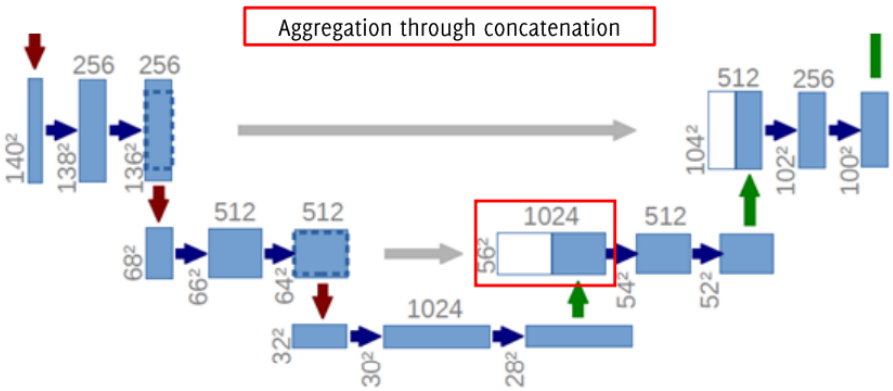
Major **differences w.r.t. F-CNN** for semantic segmentation (Long et al. 2015):

- **use a large number of feature channels in the upsampling part**, while in (Long et al. 2015) there were a few upsampling. **The network become symmetric** (both parts are deep).
- Use excessive **data-augmentation** by appling elastic deformations to the training images.

The U-Net contracting path repeats blocks of:

- 3×3 convolution + ReLU ('valid' option i.e. no padding)
- 3×3 convolution + ReLU ('valid' option i.e. no padding)
- Max-pooling 2×2

And at each downsampling the number of feature maps is doubled



The U-Net expanding path repeats blocks of:

- 2×2 transpose convolution, halving the number of features maps (but doubling the spatial resolution)
- Concatenation of corresponding cropped features
- Aggragation during upsampling:
  - 3×3 convolution + ReLU
  - 3×3 convolution + ReLU

The U-Net top is NOT fully connected: **there are L convolutions against filters $1 \times 1 \times N$, to yield predictions out of the convolutional feature maps**. The **output image is smaller than the input image by a constant border**.

The U-Net **training** is done through **full-image training by a weighted loss function to compensate for class imbalance** (since as we have seen is not possible to perform patch resampling):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{x_j} w(x_j) \cdot l(x_j, \theta)$$

where the weight:

$$w(x) = w_c(x) + w_0 \cdot e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$$

- w_c is used to balance class proportions (**remember that is not possible to do patch resampling in full-image training**) and is higher when the class are underrepresented. Hence **the term $w_c(x)$ takes into account unbalance in the training set**.
- d_1 is the distance to the border of the closest cell.
- d_2 is the distance to the border of the second closest cell.

- Hence the weights are large when the distance to the first two closest cells (d_1 and d_2)is small. Indeed, the term $w_0 \cdot e^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}}$ **enhances classification performance at boarders of different objects**. Hence **takes into account how critical the classification is taking into account the boarders between different objects**:

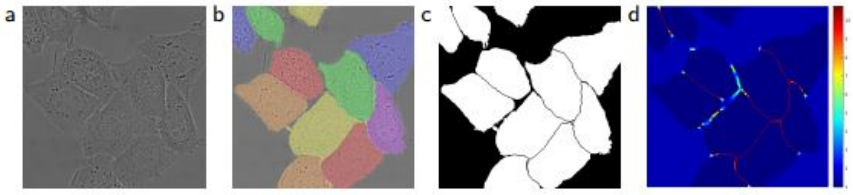


Fig. 3. HeLa cells on glass recorded with DIC (differential interference contrast) microscopy. (**a**) raw image. (**b**) overlay with ground truth segmentation. Different colors indicate different instances of the HeLa cells. (**c**) generated segmentation mask (white: foreground, black: background). (**d**) map with a pixel-wise loss weight to force the network to learn the border pixels.

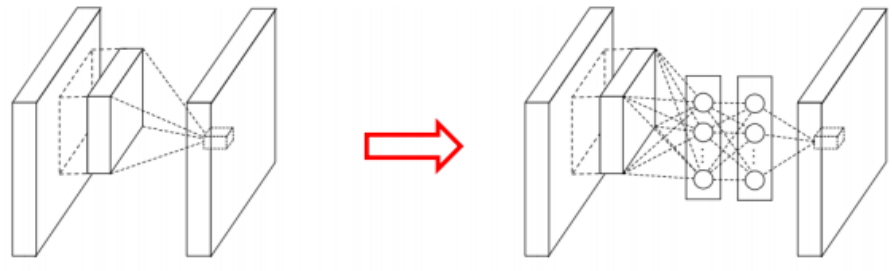
The term of the weight is large at pixels close to borders delimiting objects of different cells (look at the colormap on the right), i.e. the objective of this term of the weight is to increase the contribution in the classification error of pixels that are close to the boarders of different cells, **such that the segmentation data is more accurate and sharp**.

5.4 Global Average Pooling

Network in network paper introduced a **very important layer** that can be **found in many CNN**. They were **proposing two new layers**:

- **Mlpconv layers**: instead of traditional convolutions, a stack of 1×1 convolutions + ReLU
  - 1×1 convolutions used in a stack followed by ReLU corresponds to a MLP network used in a sliding manner on the whole image.

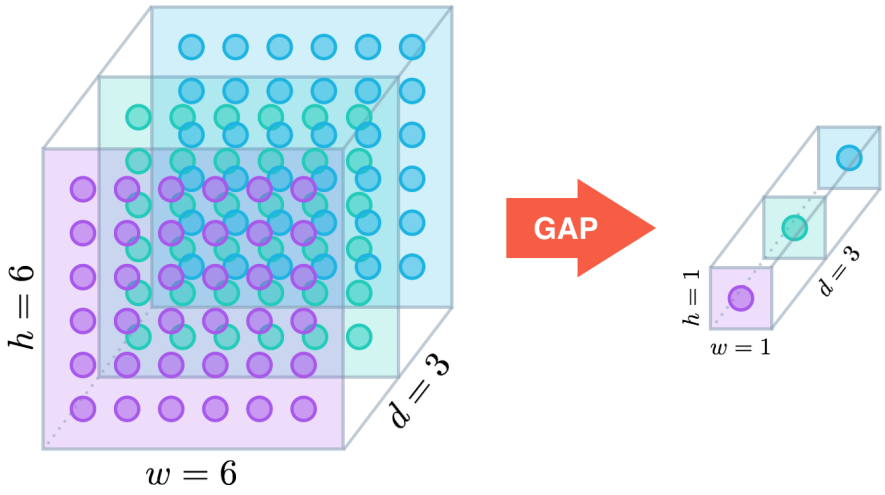
  Each layer features a more powerful functional approximation than a convolutional layer which is just linear + ReLU



(a) Linear convolution layer (b) Mlpconv layer

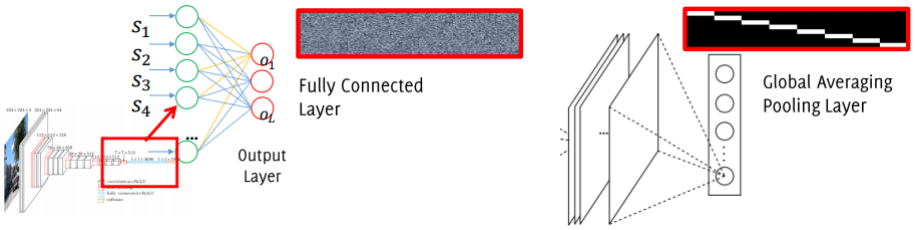
The idea is that convolution is performing just a linear combination + a non linearity is not general enough, so they implemented a small convolutionalized MLP, operating in a sliding manner sharing the filter for all the input extent, to provide more non-linear features as output. This layer was not very successful but gave the name to the paper since they are placing networks inside the networks.

- **Global averaraging pooling** layers



One of the major drawbacks of powerful architectures, such as VGG, is the large amount of coefficients in the FC layers in the network top ($\sim$ 90%) that make them very prone to overfitting. To simplify this they proposed to substitute all the connected layers at the end of a convolutional neural network with a global average pooling layer. Hence, the global average pooling (GAP) layers instead of a FC layer at the end of the network, compute the average of each feature map converting a volume into a vector i.e. each feature map in an element of the vector.

- The transformation corresponding to GAP is a block diagonal, constant matrix (consider the input unrolled layer-wise in a vector) i.e. a diagonal non-learnable matrix.
- The tranformation of each layer in MLP correspond to a dense matrix (weights are the values) i.e. a dense learnable matrix.



If the last volume of the convolutional network has depth L the GAP can directly convert the output in a vector of length L that can be passed to a softmax activation function to normalize the output between 0 and 1 for the classification task. Instead if the last volume has not depth L, we can obtain that depth through some comnvolutionalized FC layers i.e. 1×1 convolutions. The main point of the idea is to use GAP as a layer without learnable parameters in a way that the classification can be learned by the convolutional layers only, getting rid of the huge number of parameters in the fully connected layers preventing the network to perform overfitting.

The rationale behind GAP the fully connected layers are prone to overfitting:

- They have many parameters.
- Dropout was proposed as a regularized that randomly sets to zero a percentage of activations in the FC layers during training.

The GAP strategy is:

- Remove the fully connected layer at the end of the network! Making the network lighter and less prone to overfitting.
- Predict by a simple soft-max after the GAP.
- Watch out: the number of feature maps has to be equal to the number of output classes!

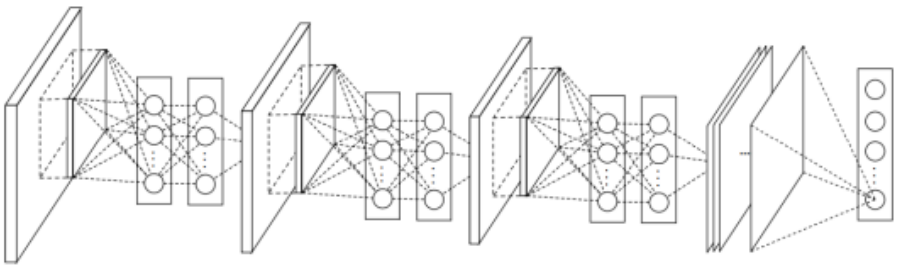
The advantages of GAP layers:

- No parameters to optimize, lighter networks less prone to overfitting.
- M̆ore interpretability, creates a direct connection between layers and classes output (we'll see soon). What the network is learning and which part of the image made the network to take the decision becomes more interpretable.

- This makes GAP a structural regularizer, to prevent overfitting.
- More robustness (i.e. invariability) to spatial transformation of the input images (such as flip, rotation) since the feature map are averaged by the GAP and this imply the invariability.
- The network can be used to classify images of different sizes
- Classification is performed by a softmax layer at the end of the GAP

The whole Network in Network stacks:

- A few layers of:
  - mlpconv layers (ReLU) + dropout
  - Max-pooling
- At the end of the network:
  - Global average pooling (GAP) layer
  - Softmax



Simple NiNs achieve state-of-the-art performance on "small" datasets (CIFAR10, CIFAR100, SVHN, MNIST) and that GAP effectively reduces overfitting w.r.t. FC.

We indeed see that GAP is acting as a (structural) regularizer:

Method	Testing Error
mlpconv + Fully Connected	11.59%
mlpconv + Fully Connected + Dropout	10.88%
mlpconv + Global Average Pooling	10.41%

better countering overfitting than FC + dropout.

GAP can be found in many new network architectures since they do not contain many large FC layers, but huge convolutional blocks with at the end a GAP layer to gain invariance to image size.

So GAP is used when we are sure to overfit, we have few data and we want the network to be invariant to image size. Typically, GAP is used at the end of the network and then followed by a little FC layer that do not have a lot of parameters since the GAP reduce greatly the input size of the layer i.e. from a volume that should be flatten to a vector of length equal to the depth of the volume.

6 CNN for localization

6.1 Localization task

In the localization task the input image contains a single relevant object to be classified in a fixed set of categories. The task is to:

- assign the object class to the image
- locate the object in the image by its bounding box

Hence for each image the output would be a vector with five values:

$$(x, y, h, w, label)$$

that identifies the bounding box and the class of the object contained into the box, where (x,y) is the coordinate of the top left corner of the box, h is the height and w is the width. A training set of annotated images with label and bounding box around each object is required. Extended localization problems involve regression over more complicated geometries (e.g. human skeleton).

The simplest solution of this problem if you are given an annotated training set which means that you are given an image, a label and the bounding box associated to that label is to train the network to predict both the class label and the bounding box i.e. to solve both a classification problem and a regression problem since we need to find the label (classification) and the bounding box coordinates (regression):

- Target class: found with softmax loss S i.e. classification loss.
- Bounding box coordinates (x, y, h, w): found with regression loss R e.g. the l^2 norm ($||predicted - groud_truth||_2$), l^1 norm ($||predicted - groud_truth||_1$), ...

The problem is that the losses to optimize are two, since the localization network would be accurate in both:

- Estimating the label
- Estimaing the bounding box

Performing multi-task optimization is way more difficult than optimize for a single task since the training loss has to be a single scalar (i.e. single loss) since we compute gradient of scalar function with respect to network parameters. So one

tend to minimize a **multitask loss** i.e. a combination of the two losses to merge two losses:

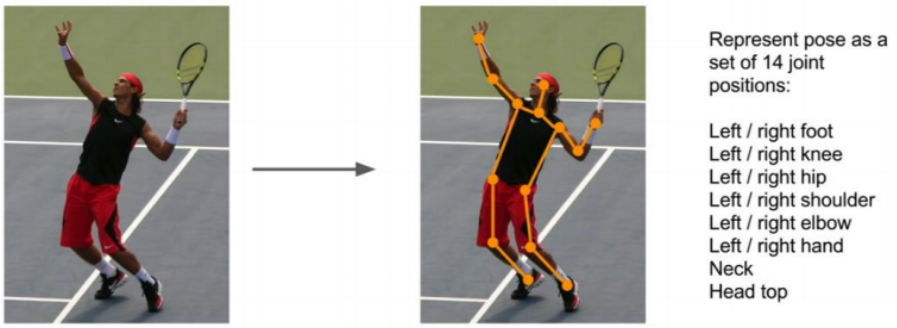
$$\mathcal{L}(x) = \alpha \cdot S(x) + (1 - \alpha) \cdot \mathcal{R}(x)$$

and α is an hyper parameter of the network. Watch out that α directly influences the loss definition, indeed changing α would mean to solve a different task i.e. a different mixture of classification and regression, so tuning might be difficult, better to do cross-validation looking at some other loss. It is also possible to adopt a pre-trained model and the train the two FC separately, however it is always better to perform at least some fine tuning to train the two jointly.

Of course there is no need to predict only the object bounding box, for example the network can be extended to human pose estimation i.e. predict the position of a selected number of joints. Pose estimation is formulated as a CNN-regression problem towards, body joints.

- The network receives as input the whole image, capturing the full-context of each body joints.
- The network doesn't change with respect to a image classification task, i.e. is still a CNN network, just the final layer instead of performing classification performs regression and the annotation i.e. the ground truth fed to the network are the joint coordinates. The approach is very simple to design and train: the ground truth is the picture with the joint manually set. Hence by simply changing the target, loss function and top layers the same network can be trained for a different task. For this reason training problems can be alleviated by transfer learning of existing classification networks.

Pose is defined as a vector of k joints location for the human body, randomly normalized w.r.t. the bounding box enclosing the human. Train a CNN to predict a 2k vector as output by using an AlexNet-like architecture.



In the latter image for example the human pose estimation is a 28-dimension ((x, y) coordinates for each of the 14 joints) pose estimation regression problem, so it does not have to minimize the multitask loss. Adopting a l^2 regression loss of the estimated pose parameters over the annotations. Some improvement can be obtained:

- This can be also defined when a few joints are not visible.
- Since the framework is very similar to the one saw for classification is possible to reduce overfitting by augmentations (translations and flips)
- Multiple networks have been trained to improve localization by refining joint locations in a crop around the previous detection.

The very important thing is that a very similar framework with very dew changes can be used for a different task.

Open pose is an example of realtime (frame by frame) multi-person 2D pose estimation using Part Affinity Fields (et al. Cao 2017). So this is not done by tracking but is a frame by frame estimation. It is also capable of multi-person tracking in a single frame with different scale. Of course the architecture isn't a simple VGG.

Bear in mind the concepts of:

- Multi task loss
- Localization is just a simple regression problem: can be solved with the same framework of image classification changing only the top network, the loss and of course the annotation given to compute the loss.
- For task that only require only localization without classification, like human joint position estimation, there is no need to use multi task loss.

6.2 Weakly-Supervised Localization: GAP revisited and visualization of what matters for CNN predictions

Weakly supervised localization means that we perform localization over an image without images with annotated bounding box:

- Training set provided as for classification with image-label pairs (I, l) where no localization information (bounding box) is provided, i.e. the classical training set used for classification.

In "Learning Deep features for Discriminative Localization" paper is proposed a revisited GAP.

The advantages of GAP layer extend beyond simply acting as a structural regularizer that prevents overfitting. Indeed, the network can retain a remarkable localization ability until the final layer, before the GAP. Hence, by a simple tweak it is possible to easily identify the discriminative image regions leading to a prediction i.e. the regions that the network use to take its classification decision.

A CNN trained on object categorization is successfully able to localize the discriminative regions for the classification. For example in the following image, in an action (e.g. brushing teeth, cutting trees, ...) classification problem, the network is able to localize the objects that the humans are interacting with rather than the humans themselves.

The class activation mapping (CAM) identifies exactly the regions of an image are being used for discrimination: the pixel intensity is higher (red) for pixel that contributed the most to define that classification output, the pecularities of the image that let the network decide for a class other than another.

CAM are very easy to compute. It just requires:

- FC layer after GAP
- A minor tweak

A very simple architecture made only of convolutions and activation functions leads to a final layer, before the GAP, having:

- n feature maps f_k having resolution "similar" to the input image.
- A vector after GAP made of n (i.e. the same number of the input volume depth, number of activation maps) averages F_k:

$$F_k = \frac{1}{D} \sum_{(x,y)} f_k(x, y)$$

  where $f_k(x, y)$ is a single point of the $k-th$ feature map and D is the number of pixels (x, y) in the activation map.

Add **(and train)** a single FC layer after GAP. The FC computes score S_c for each class c as the weighted sum of $\{F_k\}$:

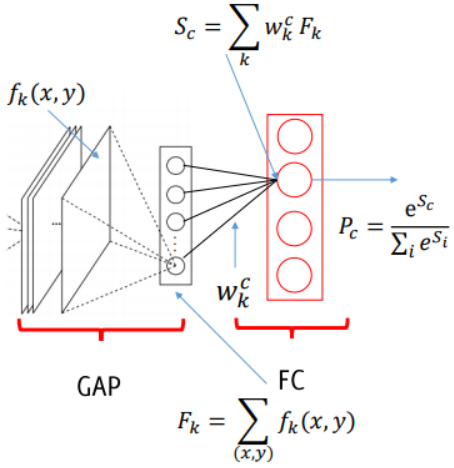
$$S_c = \sum_k w_k^c \cdot F_k$$

where k is the index of the element of the GAP vector output (so it refers to the $k-th$ activation map) and the weights w_k^c are defined during training and encode the

importance of F_k for the class c. The FC layer is followed by an possibly non-linear activation function, not showed since it does not change the general behaviour. Then, the class probability P_c is computed via soft-max for each class c: higher the S_c value higher the computed probability P_c:

$$P_c = \frac{e^{S_c}}{\sum_i e^{S_i}}$$

.



However:

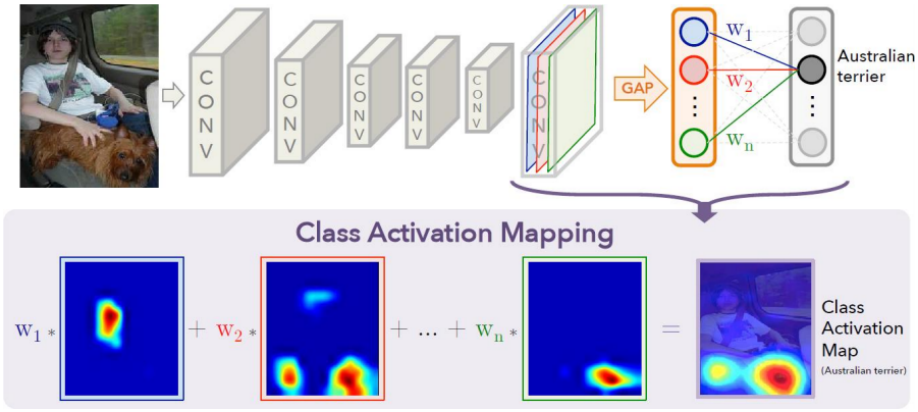
$$S_c = \sum_k w_k^c \cdot F_k = \sum_k w_k^c \sum_{x,y} f_k(x,y) = \sum_{x,y} \sum_k w_k^c f_k(x,y)$$

Form this arithmetic trick that let us to invert the two summation CAM is defined as:

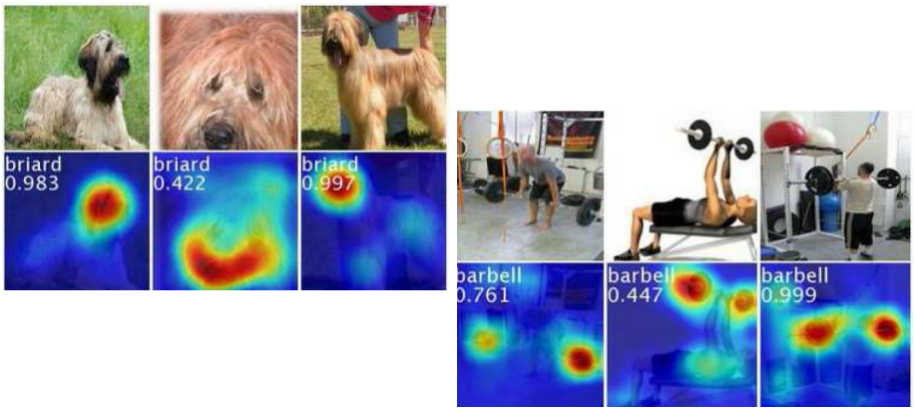
$$M_c(x,y) = \sum_k w_k^c \cdot f_k(x,y)$$

where the CAM $M_c(x,y)$ is defined for each pixel and directly indicates the importance of the activations at (x, y) for predicting the class c i.e. the importance of each pixel (x,y) in the class choice. Thanks to the softmax, the depth of the last convolutional volume can be different fro the number of classes.

Now, the weights represent the importance of each feature map to yield the final prediction i.e. to associate to that input image the predicted class. For example assuming that the FC layer added has the number of neurons equal to the number of the classes:



Hence the CAM can be seen as the weighted sum of the final activation maps before the GAP layer. The final activation maps contain some high level pattern recognition that are used with different weights to recognise each class (once the network is trained) i.e. a linear combination (non-linear if an activation function is added to the layer) of the activation maps, where the weights are the coefficient of the linear combination, hence different weights lead to different class prediction as the network give more importance to different features/region of the image/activation maps. So by changing the weights different CAM are obtained.



As we can see when the CAM of different input change, and the probability of the right class changes according to how many the computed CAM match the one that the network learnt during training, as it is in a classification task. But CAM let us see in which part the network focused its attention to make the prediction, and the probability related to the class, tells in which amount the network is sure about its prediction so how many the CAM is similar to the one it learnt from training. Upsampling might be necessary to match the input image.

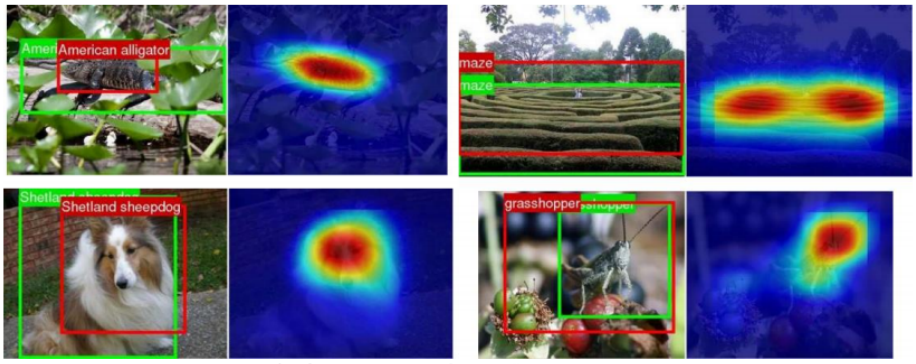
Hence after a classification network is trained, performing this simple trick of

replacing the top FC layers with a GAP layer, a single FC layer and a softmax layer, after having retrained the network with the replaced layers freezing all the pre-trained convolutional layers, we obtain a class activation map that tell us how much each region of the input image is contributing to the classification of the input image. So the network is trained for classification and with the simple trick we can extract the CAM.

Remarks:

- CAM can be included in any pre-trained network, as long as all the FC layers at the end are removed.
- The FC used in CAM is simple, few neurons and no hidden layers. For this reason and for the usage of GAP before the FC layer the parameters added to the one of the convolution block are few.
- Classification performance might drop (in VGG removing FC means losing 90% of parameters).
- CAM resolution (localization accuracy) can improve by "anticipating" GAP to larger convolutional feature maps, but this reduces the semantic information within these layers, indeed the features the activation map represent at shallow layers are of a lower level. This anticipation gives over-smoothed CAM: as you get close to the input you get higher resolution (spatial extent) but the ability to interpret the image content is reduced.
- GAP: encourages the identification of the whole object, as all the parts of the values in the activation map concurs to the classification, i.e. each portion is considered peculiar to recognise the class. Indeed, since the GAP condense the values of the whole activation map into a single average value, the value of each pixel of the activation map becomes more important and is pushed by the training to be more precise to characterize a class: in other words the whole object is important to distinguish the class.
- Global Max-Pooling (GMP), instead promotes specific discriminative features, i.e. the network is pushed to learn better only the most important part, the maximum values, that distinguish the class: in other words only the more distinctive part of the object (not the whole object as for GAP).

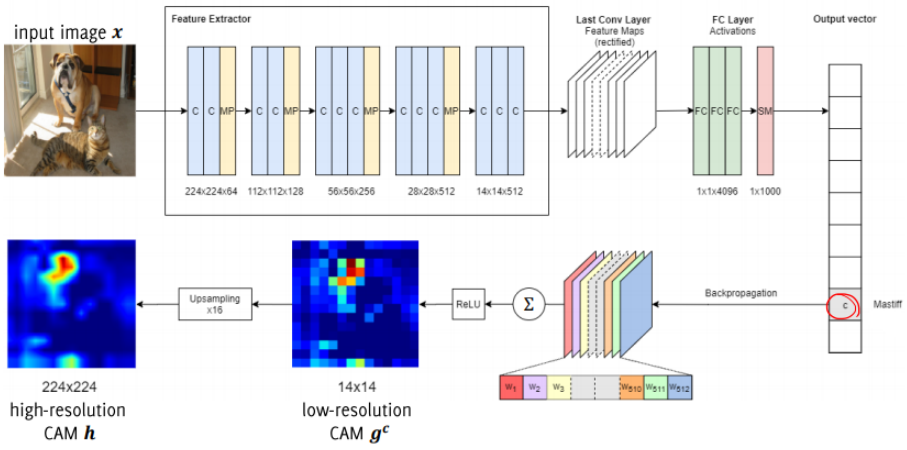
CAM can be used for weakly supervised localization. Indeed, weakly supervised localization use thresholding CAM values e.g. ¿ 20% max (CAM), then take the largest component of the threshold map:



where the green is the ground truth and in red the estimated location. The result is impressive since the network is trained for classification calculating the loss over the annotated label, but is able to solve in a pretty good way the localization task

6.3 Grad-CAM and CAM-based techniques

The objective of CAM is very important: reach a good level of interpretability of what the network does and how it took its decisions. The CAM has the drawback that you need to change the architecture of the network to obtain them, making the network less powerful since the number of parameters decreases significantly. For this reason there are alternatives to simple CAM, like Grad-CAM that do not need to modify the classification network degrading its performances. Grad stands for gradient since you can get a low resolution heat-map from the standard output of a classification CNN:



where:

$$g^c = ReLU\left(\sum_k w_k^c \cdot a_k\right)$$

where the weights are given by the value of some gradient as:

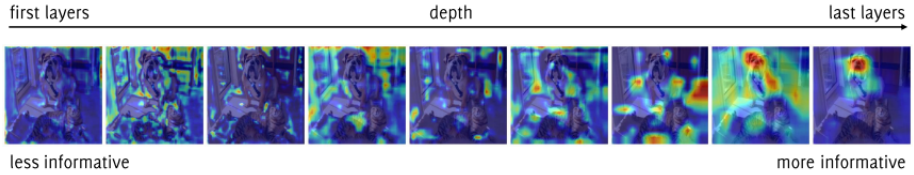
$$w_k^c = \frac{1}{n \cdot m} \sum_i \sum_j \frac{\partial y_c}{\partial a_k(i,j)}$$

with a_k the feature maps after the feature extraction network. The upsampling then let us to obtain a smooth heat-map.

The important thing about this approach is that there is no need at all to modify the network top, eliminating the possibility to decrease the performance of the network.

The desired heat-maps should be:

- Class discriminative
- Capture fine-grained details (high-resolution). This is crucial in many applications (e.g. medical imaging, industrial process).



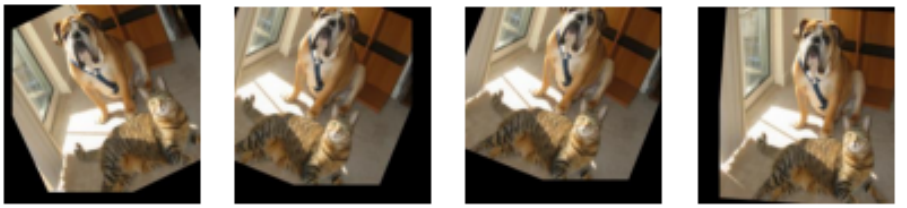
As e said before as the depth of the activation maps used to find CAM increase the result contains less semantic i.e. is less informative about the discriminative information used for classification.

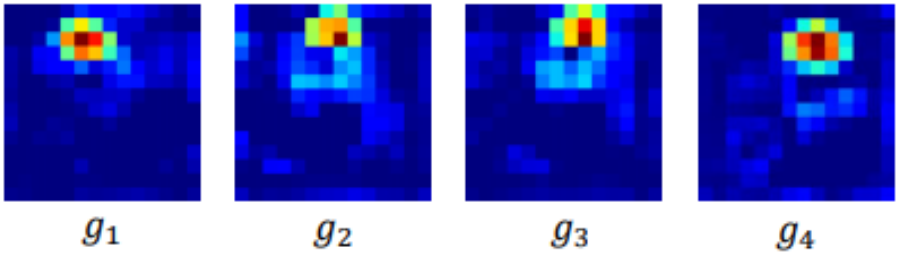
Furthermore, **Augmented Grad-CAM**, include data augmentation to improve the resolution of the heat-maps. We consider the augmentation operator:

$$\mathcal{A}_l : \mathcal{R}^{N \times M} \mathcal{R}^{N \times M}$$

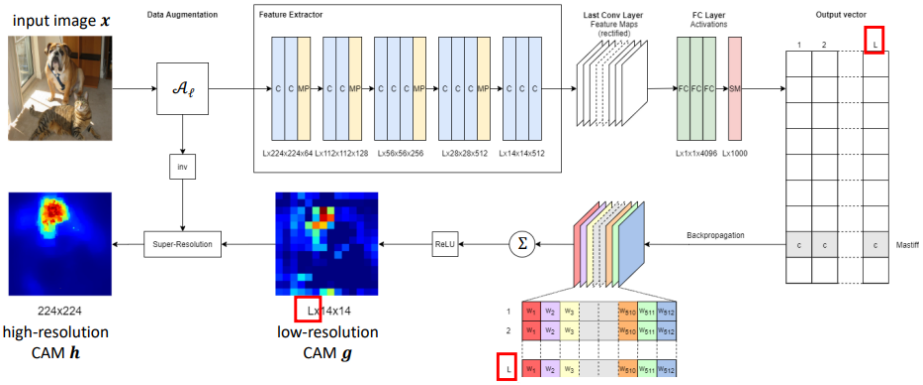
including random rotations and translations of the input image x.

All the responses that the CNN generates to the multiple augmented versions of the same input image provide multiple estimates, very informative for reconstructing the high-resolution heat-map h



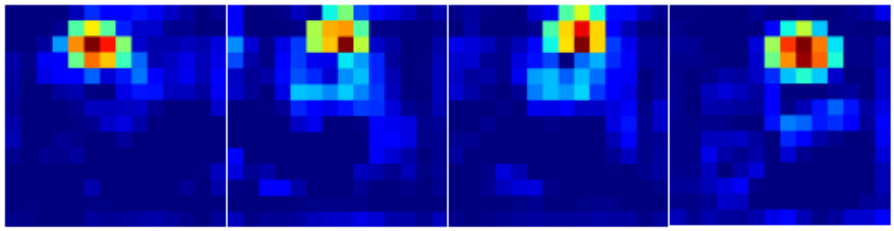


The idea is that instead of feeding a single image to the network, is performed augmentation at test time obtaining L images to feed to network, obtaining the classification from the networks and knowing for each image the kind of augmentation done.



Notice that instead of upsampling is then used a super resolution that uses the multiple low resolution CAM from the L data augmented inputs to obtain a high resolution CAM. Hence we perform heat-map Super-Resolution (SR) by taking advantage of the information shared in multiple low-resolution heat-maps computed from the same input under different - but known - transformations.

CNNs are in general invariant to roto-translations, in terms of predictions, but each g_l actually contains different information:



The general approach is that our SR framework can be combined with any visualization tool (not only Grad-CAM).

The Super-resolution framework has the goal to substitute the upsampling giving higher resolution CAM, and is formulated in the following way. We model heat-maps computed by Grad-CAM as the result of an unknown downsampling operator:

$$\mathcal{D} : \mathcal{R}^{N \times M} \rightarrow \mathcal{R}^{n \times m}$$

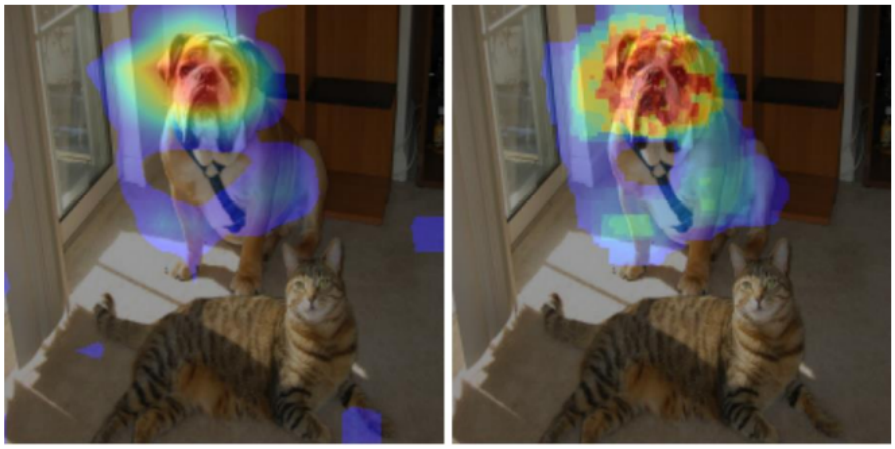
and the high-resolution heat-map h is recovered by solving an inverse problem:

$$\underset{h}{\operatorname{argmin}} \frac{1}{2} \sum_{l=1}^{L} ||\mathcal{D}\mathcal{A}_l h - g_l||_2^2 + \lambda TV_{l_1}(h) + \frac{\mu}{2}||h||_2^2$$

where TV_{l_1} is the anistopic total variation regularization is used to preserve the edges in the target heat-map (high-resolution):

$$TV_{l_1}(h) = \sum_{i,j} ||\partial_x h(i,j)|| + ||\partial_y h(i,j)||$$

and it is solved to through subgradient descent since the function is convex and non-smooth.



(a) Grad-CAM. (b) Augmented Grad-CAM.

As we can see Augmented Grad-CAM is covering better the shape of the dog and the number of augmented images used L is a design parameter.

NOTE: CNNs are, in general, not invariant to rototranslation: using data augmentation for training does not imply necessarily that the network would be invariant to rototranslation, you train them to be invariant but is never guaranteed.

7 Object detection

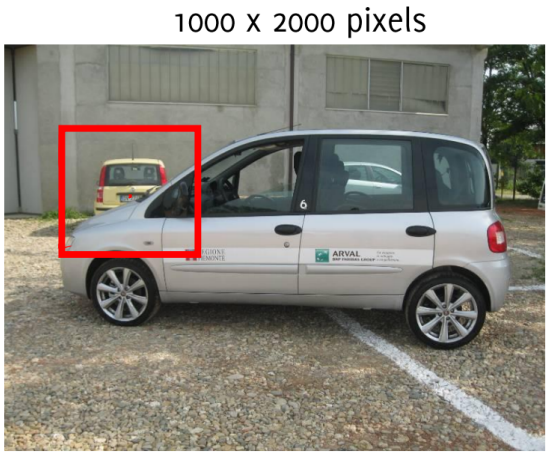
In the object detection task given a fixed set of categories and an input image which contains an unknown and varying number of instances, the network should find a bounding box for each object instance.

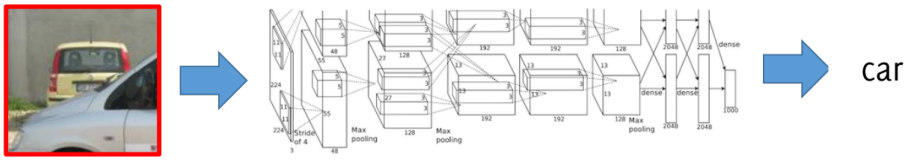
A training set of annotated images with labels and bounding boxes for each object is required.

Each image requires a varying number of outputs, one for each object.

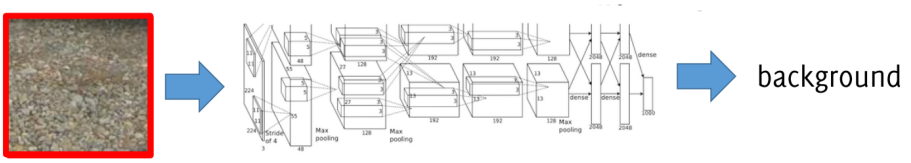
During the years were proposed many solutions to solve this task:

- The straightforward solution is to use a sliding window.
  - Similar to the sliding window for semantic segmentation, i.e. analyze the image patch-wise, parts cropped from the image.
  - A pre-trained model is meant to process a fixed input size (e.g. $224 \times 224 \times 3$)
  - Slide on the image a window of that size and classify each region.
  - Assign the predicated label to the central pixel of the region analized.





We should observe that the background class has to be included since the images crop may contain only the background.



This approach has many drawbacks:

- Is very inefficient since it does not re-use features that are "shared" among overlapping crops. The overlapping crops are a lot depending on the stride of the sliding window, so it can be a computational nightmare, since all the convolution should be recomputed even if for the vast majority they are the same.
- How to choose the crop size? You choose the image size accepted by the pre-trained classification network but you may want for example detect the gray car in the latter image, so you would need, to do it precisely, a bigger crop that contain all the car.
- Difficult to detect objects at different scales! As it is for the two car in the latter image: one can be contained in a patch and recognised by the classification network, and the other is too big to be in a single patch, so the bounding box may be not precise.

Hence from this observation we could understand how would be better to consider a huge number of crops of different sizes. Furthermore, the only plus is that:

  - There is no need of retraining the CNN.

- Region proposal algorithms (and networks) are meant to identify bounding boxes that correspond to a candidate object in the image. Algorithms were there before the deep learning advent, hence they are not trained but their functioning is based on some heuristics. They return an high number of bounding boxes and have a very high recall (but low precision), this means that they have an high probability of identifying all the objects but the precision is low since they may return bounding boxes where there is no object.

  The idea is to:

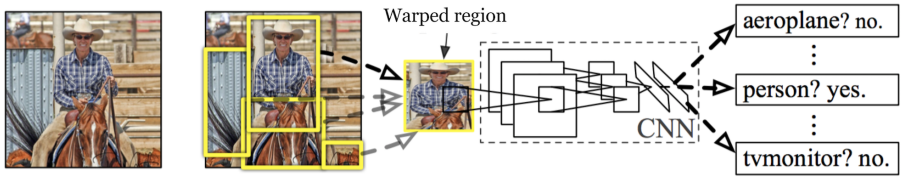
  - Apply a region proposal algorithm.
  - Classify by a CNN the image inside each proposal regions

  The network that use a region proposal algorithm instead of a sliding window are called **R-CNN**.

- Fast R-CNN
- YOLO

7.1 Rich feature hierarchies for accurate object detection and semantic segmentation

R-CNN consist in object detection by means of region proposal (R stands for regions): there is no learning in the region proposal algorithm, very high recall (e.g. selective search):



Notice that between step (2) and step (3) the region proposed should be resized to match the input dimension of the CNN. This operation is not good since the patch of the image is highly probable to be warped, changing the proportion (e.g. smashed cowboy) losing the information about the size and the proportion of the region. Warping to a predefined size is necessary since the CNN has a FC layer.

The pre-trained CNN used in the paper was AlexNet. The pre-trained CNN is fine-tuned over the classes to be detected (21 vs 1000 of AlexNet) by placing a FC layer after the feature extraction. Then they used a Support Vector Machine (SVM), i.e. a machine learning classifier, trained to minimize the classification error over the extracted features from the region of interest (ROI) extracted by the region proposal algorithm. Furthermore, they were separately training a bounding box regressor correcting (i.e. refining) the estimate of the bounding box (BB) estimated by the algorithm. Both SVM and the BB regressor where placed after the Alexnet CNN and trained separately, so no end-to-end training of the SVM and BB regressor.

1. Fine tune the pre-trained CNN for classification over the new set of classes.
2. Extract and store the latent representation, i.e. the feature, of the extracted regions for the all training set, using the fine tuned CNN.
3. Train the SVM using the stored feature extracted (as said in the previous point) from all the region for all the training set .
4. Train the BB regressor (a neural network)

Is clear how this approach require a lot of time to train and also a lot of coputational resources in terms of memory (store features for all the region extracted from all the image of the training set $\sim 2k \cdot Trining_set_dimesion$ features)

Include a background class to get rid of those regions not corresponding to an object i.e. all the non annotated bounding boxes are considered background for each input image. In this way is possible to ignore all the regions that return as predicted class the background class.

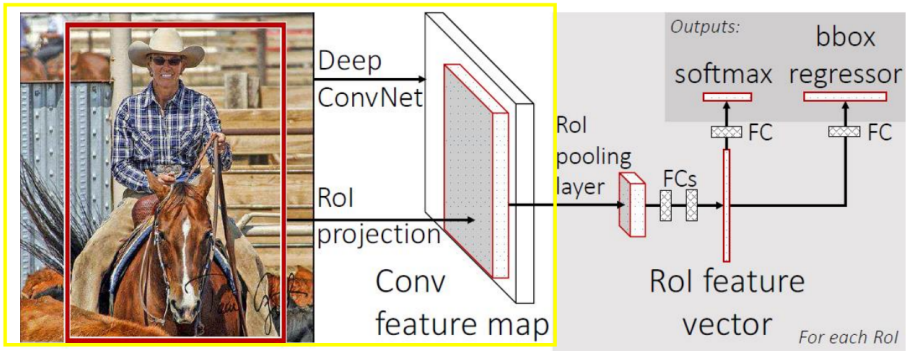
R-CNN though has some limitations:

- Ad-hoc training objectives (i.e. complicated training procedure) and not an end-to-end training
  - Fine-tuning network with softmax classifier (log loss) before training SVM
  - Train post-hoc linear SVM (hinge loss)
  - Train post-hoc bounding-box regressions (least squares)
- Region proposals are done by a different algorithm that has not been optimized (i.e. learned) for the detection by CNN.
- Training is slow (84h), takes a lot of disk space to store features.
- Inference (detection) is slow since the CNN has to be executed on each region proposal (no feature re-use) e.g. 47 [s/image] with VGG16.
- Still as the sliding window approach there is not a feature re-use (i.e. reuse of the CNN comutations) but at least overlapping is limited.

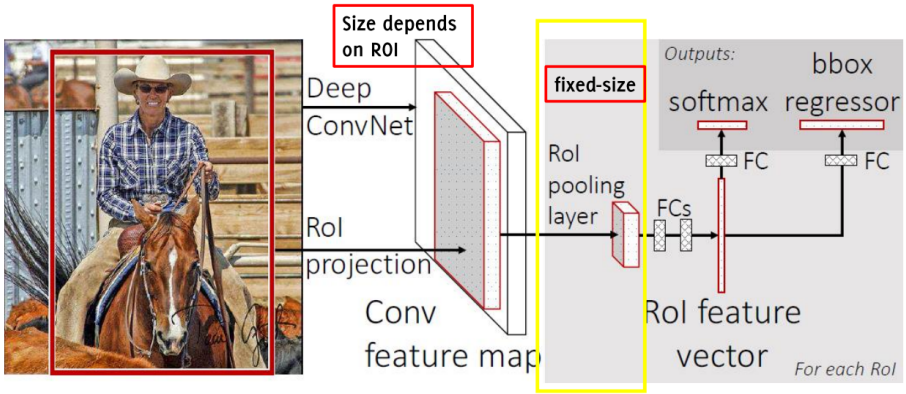
7.2 Fast R-CNN

In Fast R-CNN the idea is to use a different architecture that does not need to compute twice the same convolution and re-use the features.

1. The whole image is fed to a CNN that extracts feature maps.
2. Region proposal are identified from the image and projected into the feature maps obtained from the last convolutional layer (conv5 i.e. the last AlexNet layer). Regions are directly cropped from the feature maps, instead from the image. This let to overcome one of the biggest drawback of R-CNN, the **re-use convolutional computation**.



3. Fixed size is still required to feed data to a fully connected layer. ROI pooling layers extract a feature vector of fixed size $H \times W$ from each region proposal by operating a max-pooling. So each ROI in the feature maps is divided in a $H \times W$ grid and then max-pooling over each cell of the grid provides the ROI feature vector that is then processed by FC layers.



Hence in the latter way is extracted a fixed size (needed by FC leyers) output from a varying size region

4. The FC layers estimate both classes and BB location (bb regressor). A convex combination of the two is used as a multitask loss to be optimized. Hence is done as in R-CNN, but instead of SVM here is used a neural network.

5. Training performed in an end-to-end manner. There is no more need to store the latent representation to train, separately, the SVM and the bb regressor. Everything is done in end-to-end feeding the entire image and getting the estimates for all the objects in the input. Hence training can be done in one single step without training each peace separately.

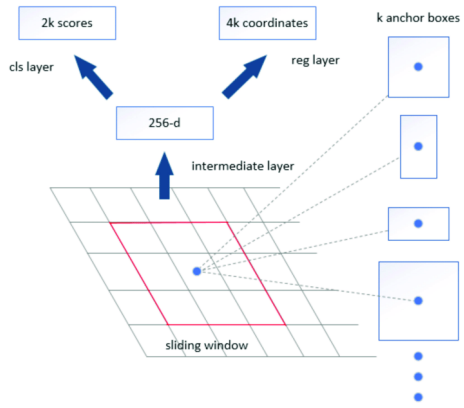
In this new architecture it is possible to backpropagate (ROI pooling is only performing a max-pooling) through the whole network, thus train the whole network in an end-to-end manner, without storing latent representation and training separately different parts.

It becomes incredibly faster than R-CNN during testing. Now that convolutions are computed only once and so are not repeated on overlapping areas, the vast majority of test time is spent on ROI extraction (e.g. selective search)

7.3 Faster R-CNN: towards real-time object detection with region proposal networks

The Faster R-CNN has uses a different way for ROI extraction:

- Instead of ROI extraction algorithm train a region proposal network (RPN) which is a F-CNN (3×3 filter size)



  Hence the region proposal can be trained and optimized for the specific dataset.

- RPN operates on the same feature maps used for classification, thus at the last convolutional layers.

- RPN can be seen as an additional module that improves efficiency and focus Fast R-CNN over the most promising regions for objects detection.

RPN goal is to associate to each spatial location (i.e. to each pixel) k different estimates of bounding boxe called anchor boxes, i.e. ROI having different scales and ratios (e.g. k $= 3 \times 3$, 3 sizes of the anchor side, 3 heigth/width ratios). The network outputs $H \times W \times k$ candidate anchor and estimate scores for each anchor.

The **RPN intermediate layer** is a standard CNN layer that takes as input the last layer of the feature-extraction network and uses 256 filters of size 3×3. It is used to reduce the dimensionality of the feature map, mapping the region to a lower dimensional vector size (output size $H \times W \times 256$) i.e. decreasing the depth of the feature map to 256.

The output of the intermediate layer is used as inpput of two networks, the classification network and the regression network. After the intermediate layer each pixel volume would be $1 \times 1 \times 256$ and this would be, each time, the input of the two networks as showed in the latter image (256-d).

- The classification network (cls) is trained to predict the object probability, i.e. the probability for an anchor (remember that the anchors refer to a certain pixel) to contain and object: so it gives 2k probability estimates since the output contains both the probability to contain and not contain the object for each of the k anchor estimated.
  - The cls network is made of stack of convolutional layers having size 1×1 i.e. convolutionalized FC network.
  - Each of these k probability pairs (a k pairs vector output) corresponds to a specific anchor (having a specific dimension) and **expresses the probability for that anchor in that spatial location (i.e. pixel) to contain any object, disregarding the class** (i.e. is only interested in the presence or not of any type of object).
- The regression network (reg) is trained to adjust each of the k predicted anchor to better match object ground truth, having as output 4k estimates for the 4 bounding box coordinates.
  - Each of these k 4-tuples expresses the refinements for the specific anchor.

If you want to train the network to predict different anchors there is no need to design different RPN, but just to define different labels when training the RPN, associated to different anchors i.e. computing the loss for the right labels. There is not an encoding of the anchors inside the network, is just a way to Of course if we want more than k anchors we have to change the cls layers and the reg layers to obtain the right outputs.

RPN returns $H \times W \times k$ region proposals, thus replaces the region proposal algorithms (selective search)

After RPN there is a non-maximum suppression based on the objectiveness score.

Remaining proposals are then fed to the ROI pooling and then classifier by the standard Fast-RCNN architecture.

Faster R-CNN training involves 4 losses:

- RPN classify object/non object

- RPN regression coordinates
- Final classification score
- Final BB coorfinates

During training, object/non-object ground truth is defined by measuring the overlap with annotated BB. The training procedure is the following:

1. Train RPN keeping backbone network frozen and training only RPN layers. this ignores object classes but just bounding box locations (Multi-task loss cls + reg)
2. Train Fast-RCNN using proposals from RPN trained before. Fine tune the whole Fast-RCNN, including the backbone.
3. Fine tune the RPN in cascade of the new backbone.
4. Freeze backbone and RPN and fine tune only the last layers of the Faster R-CNN.

At test time:

- Take the top ~ 300 anchors according to their scores.
- Consider the refined bounding box location of these 300 anchors
- These are the ROI to be fed to a Fast R-CNN
- Classify each ROI and provide the non-background ones as output.

Faster R-CNN provides as output to each iage a set of BB with their classifier posterior. The network becomes much faster (0.2 s test time per image)

Faster R-CNN is still a two stage detector:

1. First stage:
    - Run a backbone network (e.g. VGG16) to ectract features
    - Run the Region Proposal Network to estimate ~ 300 ROI.
2. Second stage:
    - Crop features through ROI pooling (with alignment)
    - Predict object class using FC + softmax
    - Predict bounding box offset to improve localization using FC + softmax

7.4 You Only Look Once: Unified, Real-Time Object Detection

R-CNN methods are based on region proposal but there are also region-free methods, like:

- YOLO: You Only Look Once
- SSD: Single Shot Detectors

Detection networks are indeed a pipeline of multiple steps. In particular, region-based methods male it necessary to have two steps during inference. This can be slow to run and hard to optimize, because each individual component must be trained separately. In YOLO the object detection is reframed as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. And solve this regression problems all at once with a large CNN.

1. Divide the image in a coarse grid (e.g. 7×7)
2. Each grid cell contains B anchors (base bounding box) associated.
3. For each cell and archor we predict:
    - The offset of the base bounding box, to better match the object:

      $$(dx, dy, dh, dw, objectness_score)$$

    - the classification score of the base-bounding box over the C considered categories (including background)

So, the output of the network has dimension:

$$7 \times 7 \times B \times (5 + C)$$

The whole prediction is performed is performed in a single forward pass over the image, by a single convolutional network. Training this network is sort of tricky to assess the loss (matched/not matched)

YOLO/SSD shares a similar ground of the RPN used in Faster R-CNN

Typically networks based on region-proposal are more accurate, single shot detectors are faster but less accurate.

8 Instance Segmentation

The image segmentation task combines the challenges of:

- Object detection (multiple instances present in the image)
- Semantic segmentation (asociate a label to each pixel) separating each object instance.

8.1 Mask R-CNN

As in Fast R-CNN classify the whole ROI and regress the bounding box (possibly estimate pose).

Does semantic segmentation inside each ROI.

Mask is estimated for each ROI and each class

Mask R-CNN can be trained end-to-end:

and include pose estimation:

and can be trained from segmentation dataset (like COCO dataset), from where you can infer bounding boxes Microsoft COCO dataset contains 200000 images segmented over 80 categories. Persons are provided with joints annotated. Since there are many instances per image, this provides a lot of training data.

9 COORECTION-TO ADD

- Skip connection are the solution to coarse upsampling: many information are lost during downsampling, and the way to recover a part of them is to skip connections (as UNET). This let the CAM to be more precise and finer, sharper indentifing better the object.
- UNET uses the upsample aproach of transpose convollution + convolution

DA AGGIUNGERE:

- Nella cross validation (hyperparameter tuning) del weight decay NON c'è early stopping: viene allenato e validato e l'errore nel grafico e quello che si ottiene nell'ultima validazione fatta. Sarà infatti la regolarizzazione a fermare proma dell'overfitting: la giusta regolazione è il minimo nel grfico degli errori di validazione e come si può vedere per un alto indice di regolazione il modello va in overfitting? da capire.
- ultimi 25 minuti lezione NON RICORDO IL NUMERO (era su webex $\implies$ ¿6??)
- DROPOUT: is based on the idea that an esemble of many models may obtain better performances i.e. taking the average of many models may improve the result of a single model. How many dropout layer we put in the network is an hyperparameter.
- WEIGHT DECAY: one of the solution to overfit is to limit the complexity of the model, but if I don't want to do that since the number of neurons let the model to learn some complex relation between the input and the output, a method is to limit the weights to acquire high values. !!!!! [Without γ the model would learn the easiest solution putting all the weigths to zero: int this way $\sum w^2$ is minimized] reason on this
- Data augmentation: increase the variance of the data. very useful for small dataset because having a small dataset increase the probability of overfitting even with a simple model.
- The layer of the volume output of a convolutional layer is called feature map
- in a convolution layer Parameter are shared, this optimize the memory consuption and faster operation. using a fcnn we would have many more parameter, so the convolutional layers also reduce the number of parameters
- maxpooling does subsampling since it reduces the spatial dimension of the volume
- Data augmentation done on each input (randomly) help to increase the variance of the dataset during the different epochs, especially with small datasets. Theoretically this creates an infinite number of images, but even if in reality they are very similar to the original one this help with a small dataset that otherwise may be overfitted.
- Deep learning change the game since the features are learned and no more set by a human: is a end to end learning that learns both the features and the model that solve the task.
- With CNN computation with only CPU becomes much slower since the input dimension are much larger

!!!!!!!!!!!!!!!!!!!!!!!!!!!! QUESTIONS !!!!!!!!!!!!!!!!!

- Slide 13 overfitting solutions: cross validation refers to the fact that we validate through the validation set different model to find out the best set of hyperparameters?
-