Politecnico di Milano

Artificial Neural Networks And Deep Learning

Notes
a.a. 2019-2020

Authors
Alessio Russo Introito
Matteo Moreschini

March 28, 2020

Contents

1 From Perceptrons to Feed Forward Neural Networks

Guardati le slides per la prima parte.

1.1 A Note on Maximum Likelihood Estimation

The goal of **Maximum Likelihood** estimation is to make inferences about the population that is most likely to have generated the sample. Maximum likelihood estimation is a method of estimating the parameters of a probability distribution by maximizing a likelihood function, so that under the assumed statistical model the observed data is most probable. The main appeal of this method derives from the fact that it can be shown to be the best estimator asymptotically (as the number of examples $m \to \infty$) in terms of its rate of convergence as m increases. Under appropriate conditions, as the number of samples goes to infinity, the maximum likelihood estimate of a parameter converges to the true value of a parameter; this property is called *consistency.*

Let's observe independent identically distributed samples from a Gaussian distribution with known σ^2:

$$x_1, x_2, \ldots, x_N \sim N(\mu, \sigma^2) \quad p(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Let $\theta = (\theta_1, ..., \theta_p)^T$ a vector of parameters, we want to find the MLE for θ; we do the following:

- write the likelihood $L = P(Data|\theta)$ for the data;
- take the logarithm of the likelihood $l = logP(Data|\theta)$, which is called the **log-likelihood**; this has several advantages, for example for a numerical point of view the product is more prone to underflow;
- find the maximum solving $\frac{\partial l}{\partial \theta_i} = 0$.

The likelihood of the data can be expressed as follows:

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

We can transform the product in a sum by taking the logarithm, having the log-likelihood:

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

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

Find out the derivative, noting that the first part does not depend on μ:

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

Solve the set of simultaneous equations $\frac{\partial l}{\partial \theta_i} = 0$:

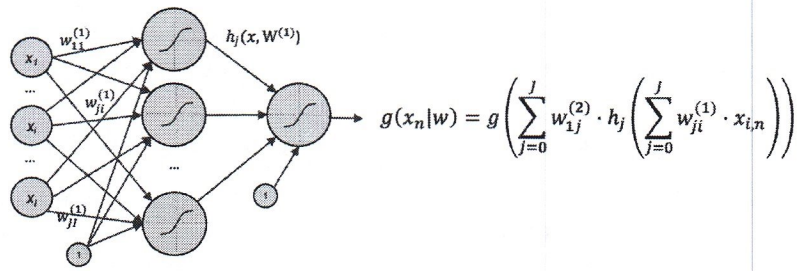
$$-\frac{1}{2\sigma^2}\sum_n^N 2\left(x_n-\mu\right)=0$$
$$\sum_n^N\left(x_n-\mu\right)=0$$
$$\sum_n^N x_n=\sum_n^N \mu$$

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

As we can see, the μ^{MLE} is the average of the x_n, so the sample mean is an estimate of the mean and this comes from the fact that you are doing a maximum likelihood estimation of the mean of a Gaussian distribution. This is an unbiased estimator, so it's a nice estimator and we'd like to do the same with neural networks.

1.2 Neural Networks for Regression

You want to find the weights so that this network approximates some target value.



Let's assume $\mathbf{t}$ as the function we want to approximate with N observations.

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

Let's assume that this network is so good for some weights that besides some noise it is exactly equal to the function ; so let's assume that this data really comes from this network but has been corrupted by noise. I assume that the difference between the prediction and the target is an error which comes from a Gaussian distribution with some noise; this is one possible assumption, we'll take this. We will approximate t as:

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

So we have i.i.d samples coming from a Gaussian distribution as follows:

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

Let's try to compute the maximum likelihood as a function of the weights, which are our parameters. The likelihood is the joint probability of all the observations [in case of a Gaussian distribution]:

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

Look for the weights which maximize the likelihood:

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

These are the canonic Gaussian but the mean is replaced with $g(x_n|w)$, which means a function parametrized in a given set of parameters w evaluated in x. You want to compute the maximum of the likelihood; instead of product we can take the logarithm:

$$\text{argmax}_w \sum_n^N \log\left(\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - g(x_n|w))^2}{2\sigma^2}}\right) = \text{argmax}_n \sum_n^N \log\frac{1}{\sqrt{2\pi}\sigma} - \frac{1}{2\sigma^2}(t_n - g(x_n|w))^2$$

We can see that $-\frac{1}{2\sigma^2}$ is a negative constant so we have to remove it by changing the sign of this function: instead of maximize we have to minimize. The max likelihood weights are obtained by minimizing this:

$$\text{argmin}_w \sum_n^N (t_n - g(x_n|w))^2$$

What if our error does not come from a Gaussian distribution? If the error is not a Gaussian distribution, you can still do this but it's not the best solution and it's not a maximum likelihood estimation so you might end up with a solution that is not unbiased (unbiased means an estimator with zero bias). But if you know that the errors are distributed differently you can follow the process and you will get the exact solution.

1.3 Neural Networks for Classification

Let's start with a very well known problem: *Classification.* Let's talk about binary classification: the output is either 0 or 1 Let's assume our model now estimates the probability of class 1, instead of the class 0, for example using a sigmoid function. We can assume that the target comes from a Bernoulli distribution and the network is basically estimating what is the probability of 1 (or 0, equally).

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

So when you are training this network you are basically training a classifier which predicts the a-posteriori probability of a class. We can write:

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

So the probability of getting a $t = 1$ is $g(x|w)^t$ and $(1 - g(x|w))^{1-t}$ if $t = 0$.
t acts as a *selector*. The likelihood is the following, assuming that the samples are i.i.d:

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

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

We want to maximize the likelihood and once again we take the sum of logarithms:

$$\text{argmax}_w L(w) = \text{argmax}_w \prod_{n=1}^N g(x_n|w)^{t_n} \cdot (1 - g(x_n|w))^{1-t_n}$$
$$= \text{argmin}_w - \sum_n^N t_n \log g(x_n|w) + (1 - t_n)\log(1 - g(x_n|w))$$

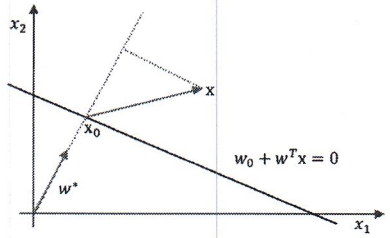
We minimize instead of maximizing and what we get is really different from the sum of squares, this is our error function:

$$E(w) = -\sum_n^N t_n \log g(x_n|w) + (1 - t_n)\log(1 - g(x_n|w))$$

This error is called **Binary Cross-Entropy**: minimizing this is very different from minimizing the sum of squares error because they solve two different problems, Regression vs Classification, additive Gaussian noise vs predictive Bernoulli distribution. The key aspect is that the error function that you are minimizing is describing the problem to the network. Basically nowadays many papers on "how to learn something" are describing the error functions.

We have discussed the binary classification problem but this can be written also for multi-class problems. The error is called **Cross-Entropy**.

If you consider perceptron, is there any implicit cost in the perceptron rule? To understand this we need a little bit of math.
Let's consider the hyperplane (affine set) $\boldsymbol{L} \in \mathbb{R}^2$ [the decision boundary of the perceptron]:



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

Let's think about our problem in a 2-dim space: the perceptron is deciding between a +1 or a -1. For any two points x_1 and x_1 on $\boldsymbol{L} \in \mathbb{R}^2$, it means that $w_0 + w^T \mathrm{x}_1 = 0$ and $w_0 + w^T \mathrm{x}_2 = 0$. We have that the *normal vector* of $\mathrm{x}_1 - \mathrm{x}_2$ (which specify the direction of the hyperplane $\boldsymbol{L}$) is w^T, in fact:

$$w^T (\mathrm{x}_1 - \mathrm{x}_2) = 0$$

The *versor normal* to $\boldsymbol{L} \in \mathbb{R}^2$ is then: $w^* = \frac{w}{||w||}$. For any point x_0 in $\boldsymbol{L} \in \mathbb{R}^2$ we have:

$$w^T x_0 + w_0 = 0 \rightarrow w^T x_0 = -w_0$$

w is something like $(\mathrm{w}_1, \mathrm{w}_2)$ so it's a vector as well.
Taking any point $\boldsymbol{x}$ in $\mathbb{R}^2$ and considering the difference vector of $\boldsymbol{x} - \mathrm{x}_0$, the projection on the direction of W^* of that vector is:

$$w^{*T}(\boldsymbol{x} - \mathrm{x}_0) = \frac{w^T}{||w||}(\boldsymbol{x} - \mathrm{x}_0) = \frac{1}{||w||} w^T (\boldsymbol{x} - \mathrm{x}_0) = \frac{1}{||w||}(w^T \boldsymbol{x} - w^T \mathrm{x}_0) = \frac{1}{||w||}(w^T \boldsymbol{x} + w_0)$$

where $w^{*T}(\boldsymbol{x} - \mathrm{x}_0)$ corresponds to the distance of x from the hyperplane $\boldsymbol{L}$ [the projection], so $(w^T \boldsymbol{x} + w_0)$ is proportional to the distance of $\boldsymbol{x}$ from the plane defined by $(w^T \mathrm{x} + w_0) = 0$ [$\boldsymbol{L}$].
To sum up, the previous procedure means that the distance of a point $\boldsymbol{x}$ from the hyperplane (i.e. the projection of $(\boldsymbol{x} - \mathrm{x}_0)$ on the direction of w^*, the normal versor of the hyperplane) is *equal* to $(w^T \boldsymbol{x} + w_0)$ multiplied by a scalar factor $\frac{1}{||w||}$.
If the vector is above the line, the distance is positive; if the vector is below the line then the distance is negative.

Perceptron Learning Algorithm The output of the perceptron can be +1/-1:

- If an output $t = +1$ is misclassified (your prediction is $t = -1$), then $(w^T x + w_0) < 0$
- On the other side if the output is $t = -1$ and your prediction is $t = +1$, then $(w^T x + w_0) > 0$

We can see that in errors the product of the two (classification label and $(w^T x + w_0)$) is always negative, so we can write this error which tries to minimize the product between the target and the distance of a misclassified point x_i from the hyperplane:

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

where M is the set of misclassified points.

Minimizing by stochastic gradient descent the error function $d(w, w_0)$, the gradients with respect to the model parameters are:

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

Stochastic gradient descent applies for each misclassified point:

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

This is exactly Hebbian Learning: model the error as the distance of points wrongly classified from the boundary. This is different because it does not say anything about probabilities. The perceptron is a classical *discriminative* approach, on the contrary feed forward neural network trained with cross-entropy is a classical *generative* approach (generative methods try to predict some a-posteriori probability, discriminative approaches instead don't care at all of probabilities, they just take a line).

Anyways this latter thing is not used anymore because you need to do this update for each misclassified point and it could be very long.

These error functions are nowadays called **Loss Functions**. The term loss actually comes from stochastic risk minimization, where you want to minimize the risk and integrate the loss over all possible outcomes etc., but for us it's something that you want to minimize. How do I design error functions? You can use the knowledge about the data distribution or you can have some background information (cross-entropy and perceptron are built in different ways in this sense) or finally, you can use your creativity.

2 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.
The connection between Computer Vision and Machine Learning is undergoing a dramatic change: once, most of techniques and algorithms were built upon a mathematical/statistical description of images. Nowadays, machine-learning methods are much more popular: you don't need anymore to go to all these mathematical/statistical models, but you just let a network to resolve a defined task.
In case of Classification, the input is an **image** which corresponds to a set of pixels each associated to a *set of colors*: in each pixel in a colored image you can get three different values (Red, Green, Blue), so we can consider three different images. Videos are sequence of images (frames), so if a frame is $I \in \mathbb{R}^{R \times C \times 3}$ where 3 represents the RGB values, a video of T frames is defined as

$$V \in \mathbb{R}^{R \times C \times 3 \times T}$$

The fourth dimensions is the frame number in the sequence.
Dimensions is terribly increasing: a pixel is 1 byte, so without compression a single frame in full HD is stored with $6MB$. Fortunately, visual data are very redundant, thus compressible. A picture is saved in jpeg, which is a lossless compression algorithm that transform the input in a different domain where it can be better compressed.

2.1 Local (Spatial) Transformations

The simplest operation on an image is a *local transformation* which can be written as

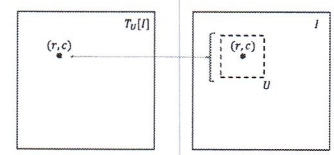
$$G(r, c) = T_U[I(r, c)]$$

where

- I is the input image to be transformed
- G is the output
- $T_U : \mathbb{R}^3 \to \mathbb{R}^3$ or $T_U : \mathbb{R}^3 \to \mathbb{R}$ is a function
- U is neighbourhood, identifies a region of the image that will concur in the output definition

T operates on I "around" U. T can be either linear or nonlinear.

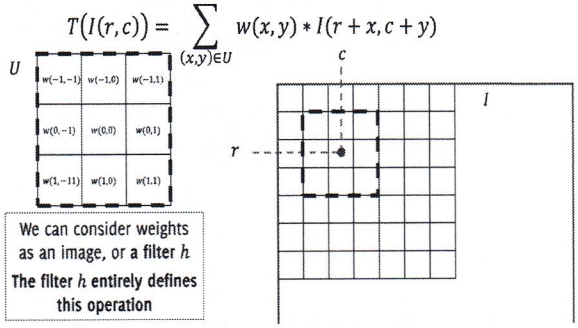
Filter Transformation takes around each pixel a suitable neighbourhood and defines the output of the transformation by looking only on those values here around. The output of a filter is a sort of function defined in a neighbour of a pixel.
The simplest one is the **Linear Transformation**: given a pixel, the output of the filter in that pixel is provided by a



linear combination of the pixel around that pixel. The output is the linear combination of the weights and the intensity of the values.

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

The parameters of that operation are the weights w_i: they can be written like an image, in a matrix. The weights now are arranged in a matrix are called **Filter** h: the filter h entirely defines this operation. This is the Linear Filtering:



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

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

where the filter h is of size $(2L+1) * (2L+1)$. The idea is that you take a filter and you overlap it to each location of the image. This operation is performed on each color of the image.
"UTK" correlated with "T" example - slide 48:
The weights of the filters are 0 or 1 (black or white), when I put the filter over the T in the image, I get the number of pixels equal to 1 in the T. The zeros next to the T is higher, as you move slightly about of the T the intensity of the output increasing since there will be some zeros that will overlap to the T. In this way we can compare images or I can use it to find where the T is in my image. It is enough to know where the T is by taking the local maxima in the correlation output. What you get is where the Ts are. Sometimes we need some sort of normalization to get the things works.

2.2 Problem Definitions

There are different problems to which we focus on:

- **Image Classification**: Assign to an input image I a label l from a fixed set of categories: the classifier gives you the probabilities over each class.

$$I \to l$$

- **Localization**: Assign to an input image I a label l from a fixed set of categories and the coordinates (x, y, h, w) of the bounding box enclosing that object.

$$I \to (x, y, h, w, l)$$

- **Object Detection**: Assign to an input image I **multiple** labels $\{l_i\}$ from a fixed set of categories, each corresponding to an **instance of that object**, and the coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing **each** object.

$$I \rightarrow \{(x, y, h, w, l)_1, \ldots, (x, y, h, w, l)_N\}$$

- **Image Segmentation**: Assign to **each pixel** of an input image I a label $\{l_i\}$ from a fixed set of categories Λ

$$I \rightarrow S(x, y)$$

where $S(x, y) \in \Lambda$. The output of segmentation is an image itself.

- **Instance Segmentation**: Assign to an input image I **multiple** labels $\{l_i\}$ from a fixed set of categories Λ, each corresponding to an **instance of that object**, the coordinates $\{(x, y, h, w)_i\}$ of the **bounding box enclosing each object**, the **set of pixels** in each bounding box corresponding to that label

$$I \rightarrow \{(x, y, h, w, l, S)_1, \ldots, (x, y, h, w, l, S)_N\}$$

It is like a mix between object detection and segmentation because for each input image you have to assign multiple labels, each one corresponding to instance of an object, to each object you have to provide a bounding box and there is a list of pixel inside the bounding box covered by the object. You can distinguish different persons, in segmentation you cannot.

Is this a *challenging problem*?

1. Images are very **high-dimensional data**.
2. **Label Ambiguity**: a label might not uniquely identify the image.
3. **Transformation**: you can perform an image transformation that does not change the content, the meaning of the image, but changes the intensity values of the image, so the pixel values.
4. **Inter-class Variability**: images belonging to the same class can be dramatically different.
5. **Perceptual Similarity**: it's not pixel-wise similarity, but it means the way you perceive two images to be similar.

2.3 Nearest Neighborhood Classifier

Assign to each test image, the label of the **closest image** in the training set:

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

Distances are typically measured as: $d(x_j, x_i) = \|x_j - x_i\|_2 = \sqrt{\sum_k \left([x_j]_k - [x_i]_k\right)^2}$

You are measured the distance between two vectors, so basically you are computing a *pixel-wise similarity.*

In the same way you can create a K-Nearest Neighborhood Classifier which assigns to

each test image, the most frequent label among the **K-closest** images in the training set. Both the methods will not work since the pixel-wise similarity does not *understand* the perceptual similarity.
Pros:

- Easy to understand and implement
- It takes no training time

Cons:

- Computationally demanding at test time
- Large training set have to be stored in memory
- Rarely practical on images: distances on high-dimensional objects are difficult to interpret

2.4 Linear Classifier

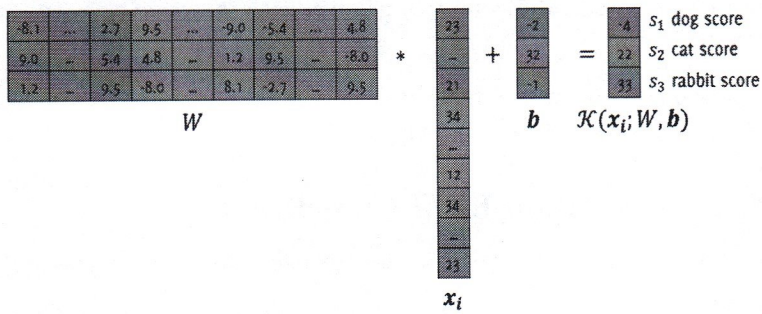
A classifier can be seen as a function that maps an image x to a confidence score for each of the L classes:

$$\kappa : \mathbb{R}^d \to \mathbb{R}^L$$

where $\kappa(x)$ is a L-dimensional vector and the i-th component $s_i = [\kappa(x)]_i$. A good classifier associates to the correct class a score that is larger than the scores associated to incorrect classes.
Dealing with images, the input $\mathbf{x}$ belongs to $\mathbb{R}^d$ where d is the dimension of the image. The Linear Classifier get an input image and extract a set of scores for each class, but need to be linear. In Linear Classification, κ is a **linear function**:

$$\kappa(x) = Wx + b$$

where $W \in \mathbb{R}^{L \times d}$ are the weights , $b \in \mathbb{R}^L$ is the bias, both are the parameters of the classifier.



So you need to perform a linear transformation of the input vector and what you obtain as output is a vector in which each component is the score assigned to each class. In the *weights matrix*, each row corresponds to the weights related to a class; a row has the same size d of the input image vector x_i. To create the input image vector, starting from an image (and from the red image of an RGB), you unroll it column-wise: from top-left corner, you read column wise and move from red to green to blue images. This

corresponds to your vector x_i, your input image i. So the linear classifier operation is row (a class) times column (your input), which output is a vector of L values, one for each class. Adding the bias we get the last output of the classifier as results. The classifier assign to an input image the class corresponding to the largest score

$$\hat{y}_j = \underset{i=1,\dots,L}{\operatorname{argmax}} [s_j]_i$$

The parameters of this classifier are the weights, so we need to train this classifier in order to learn the weights: weights indicate which are the most important pixels / colors. Also the value of bias is in your parameters. The number of parameters are $d \times L + L$, which are the exactly number of parameter in a layer of a neural network.
Given a training set and a loss function, define the parameters that minimize the loss function over the whole training set, so in case of linear classifier:

$$[W, b] = \operatorname{argmin}_{W \in \mathbb{R}^{L \times d}, b \in \mathbb{R}^L} \sum_{(x_i, y_i) \in TR} \mathcal{L}(\boldsymbol{x}, y_i) + \lambda \mathcal{R}(W, b)$$

The loss function has to be regularized to achieve a unique solution satisfying some desired property, where $\lambda > 0$ is a parameter balancing the two terms.

2.4.1 Geometric Interpretation of a Linear Classifier

Given a classifier already trained with W and b. The first intuition, if you take the matrix W and input x, you can add the bias and compute the scores. What contribute to the scores of the first class is just the first row of the matrix. So you can see the i-th row of W as the classifier corresponding to the i-th class. It just a matter of computing a inner product between w_i and an input x.
If we were in two-dimension, an image is like a point, each row of the matrix identifies that linear function which corresponds to a set of (hyper-)plane where if the input point is above or below that plane it means that the point belongs or not to a class. So what you are doing is looking at your image in a d-dimensional space and trying to separate classes by hyper-plane and dividing your input images by 'putting' them in one of these hyper-planes.

Geometric interpretation: you take an high-dimensional space, try to separate images through hyper-planes and you define the hyper-plane in order to do the best you can do for the given training set.

Another interpretation: taking W, a row of W contains d values which is the same size as image, so we can reshape it to generate an image and visualize it like an image $\rightarrow$ Template. So you multiply this template for your input and sum the bias for each template and this is exactly correlation that we have seen before. The Linear Classifier is just learning some template to which it performs correlation against and you provide as output the class of the template that provides you the largest response.

3 Training and Overfitting

"A single hidden layer feedforward neural network with S shaped activation functions can approximate any measurable function to any desired degree of accuracy on a compact set"
Universal approximation theorem (Kurt Hornik, 1991)

The universal approximation is good if you want a non-linear function approximator, you can use feed-forward neural network and you can learn anything but there are some drawbacks:

- It doesn't mean we can find the necessary weights: it may be difficult to learn everything, since it is not easy to find out the starting point and be sure to converge on the right place. There is a place in parameter space where your network approximate your function perfectly, but it is up to you to find this place.
- An exponential number of hidden units may be required: it is true that this object can approximate everything but the point is that it may require an infinite/exponential number of hidden neurons, so it may be impractical to do it.
- It might be useless in practice if it does not generalize: it can perfectly fit the data but it might not follow the original function → **Overfitting**: a model has learned to provide good performance at training time but it is not able to perform (to generalize) properly at testing time.

If you think to the problem of learning a continue function, let's assume you have a set of samples and you have a network that is over complex, so you can incur in a situation like the one represented in the right part of Fig.1. The problem is when you get new data, those data may be close to the function, but it does not mean that the function that I have learned is closed to those data [since the predicted function is over complex].

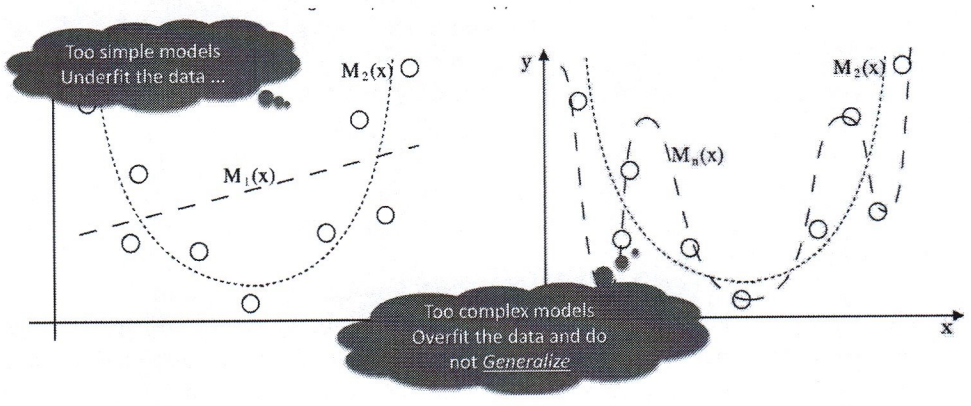


Figure 1: Under-fitting & Over-fitting

In fact, you tried to minimize the error on the train, but you lost the ability to generalize over new data: the model is not able to Generalize. It is an old story called Ockham's Razor: *between a simple model and a very complex one you have to chose always the simple one.*

In the other hand we have **Underfitting** when the model is too simple w.r.t. the real model. Too complex model tends to overfitting, so very good performance in training but very low performance in testing. You need to prevent overfitting.
We need to find a way to measure the **Generalization**: training error/loss is not a good indicator of performance on future data:

- The model has been learned from the very same training data, any estimate based on that data will be optimistic
- New data will probably not be exactly the same as training data
- You can find patterns even in random data

The idea is that you cannot use the same data of the training; you require a new independent set of data. Usually you take the data and split them in two sets: we train on one set and we test on the other. We *never ever* use the test set at training time. Sometimes you can *sample* the data *with replacing*: it means that there is some probability for the same data to be in test and in train. This random sampling with replacing is called **Bootstrap** and it is used when the dataset is very small. What is important is that you want the training data to have the same distribution of your test data; what is done is **Stratified Sampling**: by doing it, essentially you are doing Cross Validation. The latter can have different form: hold-out set (but you have to be lucky to get a set that reflect the test set. With a small dataset, it could be problematic because the performance on test-set will depend on the separation taken into account), leave-one-out (not feasible for a huge amount of data), k-fold cross validation (averaging the performance of each fold).

3.1 Dealing with Overfitting

Some of the strategies to deal with overfitting in neural network are presented in the following section:

3.1.1 Early Stopping: Limiting Overfitting by Cross-Validation

This is the most effective one if you have enough data. The point is that if you monitor the error on unseen data (validation-set) during the training process, you will reach a point in which your model has the minimum error on those new data. Once found this point, you can stop the training and consider it as the best model. To do this you need another set of data besides train and test: splitting the train-set, you get another train-set and a validation-set. The test set is used as the last thing to estimate the model on new data. The validation set is used during the training. You never use the validation to learn but only to stop the learning procedure.
To recap, *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 we need to:

- Hold out some data
- Train on the training set
- Perform cross-validation on the hold out set

- Stop train when validation error increases

One tipical procedure that is used to set up the neurons is to evaluate early stopping (the generalization error) to decide on other parameters, since the validation set error is an estimate of the error that you will get on test set. This idea of deciding on the parameters of the network is called **Hyper-parameter tuning**. Hyper-parameter describe the network structure and, once defined it, you know how many parameter you have to optimize.

3.1.2 Weights Regularization

Sometimes you don't want to remove some data because you would like to exploit all of them. So there are technique to prevent overfitting without stopping the model, but instead somehow limiting the capability of the complex model: we have a powerful model but we impose some constraint on the model *freedom*, based on a-priori assumption.
In case of neural network, we need to introduce a concept based on Bayesian Statistics: so far we have seen the problem of learning a model as a maximum learning estimation problem. We have been estimated the parameters of model in order to maximize the *likelihood* of the data. What we show now is the difference between *Frequentist* approach and *Bayesian* approach. If we take the coin toss, the Frequentist approach bases the probabilities on the number of times I hit head/cross over the total number of tosses. The Bayesian approach has an opinion on the probabilities of head based on a bias (initially bias means that we assume that the coin is balanced, so we have 50% head/cross); I toss the coin and I see head: the frequentist said that the prob. of head is 1. The bayesian instead said 0.5, one toss it is not enough to change the opinion. Second toss, second head: frequentiest said prob. of head is 1; bayesian said is not enough. After 10000 toss and 10000 heads the bayesian changes opinion.

According to the frequentist approach, you have no opinion: you look to the data and you select among all possible model the one that fits better the data → Unfortunately, this is basic overfitting, because you cannot bias the model.

On the other hand, the bayesian approach starts from an opinion that have a great bias, and this prevent bayesianist to change opinion because he have seen only few data.

In statistics, opinions are named *priors*. Maximum likelihood estimation finds the weights which maximize the data probability. A bayesianist looks for the most probable weights: in this way we are looking for **maximum a-posteriori**, we look for the most probable weights having observed data. The main difference between this two is that one looks for the probability of the data given the weights and the other for probability of the weights given the data. Substantially it is the **Bayes theorem**:

$$\begin{aligned} w_{MAP} &= \text{argmax}_w \, P(w|D) \\ &= \text{argmax}_w \, P(D|w) \cdot P(w) \end{aligned}$$

This formula give you the possibility to bias the weights: i.e. if for you any weight is the same, this probability is uniform and you get maximum likelihood estimation because there is no difference between one and the other; if you want a model which is smooth, you have to put a prior probability which act as smoothy factor.

The idea of maximum a-posteriori is to force the model to select some weights and not other. It's sort of forcing, not a constraint.

How we define the prior in a way that our network does not overfit? It is observed in practice that *small weights* makes network less prone to overfit; the bigger the weights the more likely the network overfit. Basically what we would like to do is limiting the capability of changing w.r.t. the variation on the input.
But what is that a small variation in the input become big in the output? The weights: they are the parameters, so they control the magnitude of the model. That's why in general networks with small weights overfit less. We need to force our algorithm to prefer small weights: to do it, one possible way is to say that if you take all the weights, on average they have to be as small as possible, so near to zero on average. Then the bigger they become, the less likely they become.

One possible way to enforce bias on the weight distribution is to ask for the weight distribution as a Gaussian with zero-mean and some variance σ_w^2.
Instead of maximize likelihood we want to maximize a-posteriori probability, so we maximize the probability of the weights given the data. Let's assume we are in regression setting and the probabilities of weights are:

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

$$P(w) = \frac{1}{\sqrt{2\pi}\sigma_w} e^{-\frac{(w_q)^2}{2\sigma_w^2}}$$

Assume that Q is the number of weights on the network (includes all the parameters in the network):

$$\begin{aligned}
\widehat{w} &= \text{argmax}_w \, P(w|D) = \text{argmax}_w \, P(D|w)P(w) \\
&= \text{argmax}_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - g(x_n|w))^2}{2\sigma^2}} \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w} e^{-\frac{(w_q)^2}{2\sigma_w^2}} \\
&= \text{argmin}_w \sum_{n=1}^{N} \frac{\left(t_n - g\left(x_n|w\right)\right)^2}{2\sigma^2} + \sum_{q=1}^{Q} \frac{\left(w_q\right)^2}{2\sigma_w^2} \\
&= \text{argmin}_w \sum_{n=1}^{N} \left(t_n - g\left(x_n|w\right)\right)^2 + \gamma \sum_{q=1}^{Q} \left(w_q\right)^2
\end{aligned}$$

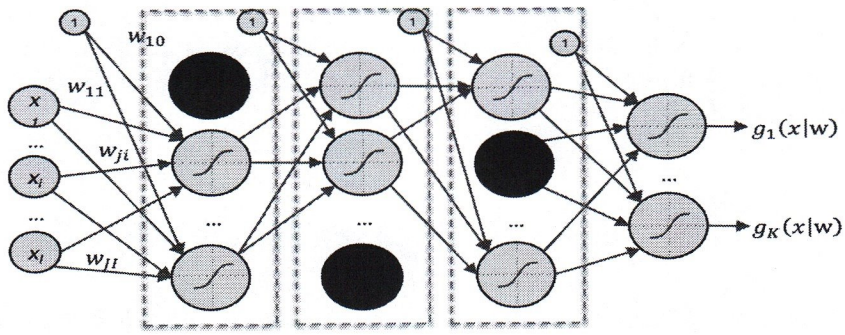
Where we define $\gamma = \frac{\sigma^2}{\sigma_w^2}$
The regularization term is positive. Adding a regularization term you are less prone to overfit because it makes your model smooth.
To force the network to have small weights we have to minimize the fit of the data plus some regularization term that is the sum of the squared of the weights → Ridge regression in Regression model does exactly this.
This is a classical form of loss function, another one is to add the sum of the absolute values of the weights → Lasso: it brings some weight to zero.

The best γ is the one that minimize the generalization error: we can use Cross-Validation to select the proper γ.

In neural network literature, this approach that uses quadratic regularization term in error function is called **Weight Decay**.

3.1.3 Dropout

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 is set to zero with $p_j^{(l)}$ probability, e.g. $p_j^{(l)} = 0.3$.
Dropout trains weaker classifiers, on different mini-batches and then at test time we implicitly average the responses of all ensemble members. At testing time we remove masks and average output (by weight scaling).
This technique modifies the network itself, not its loss function. It removes a small percentage of neurons on each hidden layer, chosen randomly.
It behaves as an ensemble method: it's like we have an ensemble of a lot of different networks.



3.2 Tips and Tricks

Activation functions such as Sigmoid or Tanh saturate, which leads to:

- Gradient is close to zero
- Backpropagation requires gradient multiplication
- Gradient faraway from the output vanishes
- Learning in deep networks does not happen

The problem of **Vanishing Gradient** raises from the fact that when you have very complex long neural network, to compute the gradient, you have to compute all the chain of derivatives:

$$\frac{\partial E\left(w_{ji}^{(1)}\right)}{\partial w_{ji}^{(1)}} = -2\sum_{n}^{N}\left(t_n - g_1\left(x_n, w\right)\right) \cdot g_1'\left(x_n, w\right) \cdot w_{1j}^{(2)} \cdot h_j'\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right) \cdot x_i$$

In deep networks, the gradient results in a very small number.
The issue is in Sigmoidal and Hyperbolic Tangent functions since the maximum of the derivative is in the origin and if we define they as a function $g()$, we know that

- for Sigmoid: $g'(a) = g(a)(1 - g(a)) \rightarrow$ the maximum of the derivative is where the values is 0.5, so the maximum is equal to $0.5(1 - 0.5) = 0.25$
- for Tanh: $g'(a) = 1 - g(a)^2$

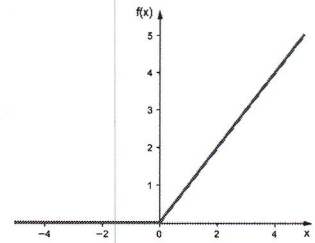
In general the derivative is less than 1. The main issue with gradient descent is that the only way to fix it is by changing the operator function, so we need to use activation functions that do not suffer of this issue of having the derivative less than 1 $\Rightarrow$ **ReLU**

3.2.1 ReLU

The ReLU (Rectified Linear Unit) activation function is defined as:

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

$$g'^{(a)} = 1_{a>0}$$



and it has some interesting properties:

- The derivative is either 1 or 0. Compute the derivative is essentially an "if"
- Faster SGD Convergence: in the backward chaining, once you get the zero, you can forget about the past
- Sparse activation (only part of hidden units are activated): if on average half of the activation functions will be positive and the other half negative, then half of these neuron are switched off, so we don't have to compute it. Increase the efficiency and increase generalization since function is more simple, so it is more likely to not overfit.
- Efficient gradient propagation (no vanishing or exploding gradient problems): you multiply by 1 or 0.
- Efficient computation (just thresholding at zero)
- Scale-invariant: $max(0, a = amax(0, x)$. To some extent it is more robust to conditions, it is not affected by the values of current activation functions.
- It is not a linear function.

But also potential disadvantages:

- Non-differentiable at zero: however it is differentiable (so not a significant issue)
- Non-zero centered output: if you want the model outputs a number that is not strictly positive, you can use it. If you want a strictly positive output, you have to use a linear activation function.
- Unbounded: Could potentially blow up

- *Dying Neurons*: ReLU neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. No gradients flow backward through the neuron, and so the neuron becomes stuck and "dies", no way of switching it on. It decrease the model capacity and usually happens with high learning rates

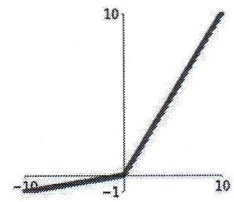
Dying Neurons is an issue, but in the other hand switching off some neuron it's good since we create a simpler model. The problem is when it happens too early, i.e. at initialization time. Another thing that could happen: sometimes having a big learning rate could generate issues with these neurons because they become inactive too early w.r.t. training time.

The main solutions are:

- Leaky RELU: the negative part decreases slowly, so there is a small gradient and when you go near or less than 0, you have a little chance to come back/to recover instead of switching itself off. It fix the "dying ReLU"

Leaky ReLU: fix for the "dying ReLU" problem

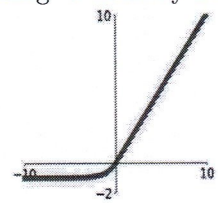
$$f(x) = \begin{cases} x & if\ x \geq 0 \\ 0.01x & otherwise \end{cases}$$



- ELU: you have an exponential and a lineal part and you can tune the α for the exponential part to define how much you would like to go under zero. It tries to make the mean activations closer to zero which speeds up learning. α hand by hand

ELU: try to make the mean activations closer to zero which speeds up learning. Alpha is tuned by hand*by hand*

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



Modern architectures use ReLU, since initialization is a problem per-se.

3.2.2 Weights Inizialization

The final result of gradient descent is affected by weight initialization:

- *Zero*: it does not work. All gradient would be zero, no learning will happen.
- *Big Numbers*: bad idea. Assume to use sigmoid function and very big weights , you are in an area which gradient is close to zero → Long-time to converge
- *Classic strategy*: small weights sampled by Gaussian distribution with zero mean and low variance. It works decently. The bigger is the variance the more it takes to training, since the weights are big and they are big when there are on the wrong side, so they take more time. Small number are fine: the weights starts more or less with the maximum of the gradient → at the very beginning the learning is fast. It is good for small networks, but it might be a problem for deeper neural networks.

However, in deep networks:

- In a big network, if the weights are too small, you get the shrink gradient, since the gradient is the product of derivatives but also of the weights along a path. The gradient will not be zero but will be very small
- If you start with large weights, the gradient grows as it passes through the layers and at certain point will be too big. Assume you have a fixed learning rate, since in a way you tune the weights as (weight - learning rate * gradient), if the gradient grows, the product grows, so you have huge change in the weights.

Some proposal to solve the problem is **Xavier Initialization**:
Suppose we have an input x with I components and a linear neuron with random weights w. The output of this neuron is

$$h_j = w_{j1}x_1 + \cdots + w_{ji}x_I + \cdots + w_{jI}x_I$$

We can derive that $w_{ji}x_i$ is going to have variance:

$$\text{Var}(w_{ji}x_i) = E[x_i]^2 \text{Var}(w_{ji}) + E[w_{ji}]^2 \text{Var}(x_i) + \text{Var}(w_{ji}) \text{Var}(x_i)$$

Let's assume that both input and weights have zero-mean:

$$\text{Var}(w_{ji}x_i) = \text{Var}(w_{ji}) \text{Var}(x_i)$$

For the weights it is reasonable; for the input, it is not mandatory, but it has been observe that if you condition the input to have zero-mean and unitary variance, the optimization algorithm is better conditioned and it is more stable. If we assume all w_i and x_i are $i.i.d$ we obtain:

$$\text{Var}(h_j) = \text{Var}(w_{j1}x_1 + \cdots + w_{ji}x_I + \cdots + w_{jI}x_I) = n \text{Var}(w_i) \text{Var}(x_i)$$

So each neuron amplifies the variance of the input by a factor $nVar(w_i)$.
Assume that we have unitary variance input, what we don't want to do is to increase the variance of the input too much, so we try to force it to be consistently zero-mean and unitary along all the layers, we should enforce that the gain of the weights on each neuron should be 1:

$$nVar(w_j) = 1$$

For this reason Xavier proposes to initialize

$$w \sim N\left(0, \frac{1}{n_{in}}\right)$$

More fast in converging and more stable.

Glorot and Bengio found a similar result: $n_{out}Var(w_j) = 1$, so they propose:

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

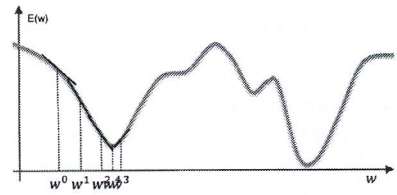
More recently, He proposed, for rectified linear units:

$$w \sim N\left(0, \frac{2}{n_{in}}\right)$$

3.2.3 Momentum

The idea is to accelerate training and giving to it some inertia.

Recalling Backpropagation, finding weights of a Neural Network is a non-linear minimization process:



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

We iterate from an initial configuration:

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

To avoid local minima we can use **Momentum**:

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

Nesterov Accelerated Gradient makes a jump as momentum, then adjust

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

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

The Nesterov idea is improving the estimate by dividing in two steps: the first step uses the momentum and it gives you a direction; in the second step re-estimate/update the gradient here, where the momentum leads you.
It should be a better estimate since the gradient will be closer to the minimum. It turns out to be more efficient, converges fast than SGD.

There are other algorithms that try to **adapt learning rates** while learning, to deal with:

- Gradient magnitudes vary across layers
- Early layers get "vanishing gradients": the gradient will be smaller and smaller the more you go back. It means that in the very first layer it should be boosted by an higher learning rate.
- Should ideally use separate adaptive learning rates

Several algorithm are proposed: Rprop, Adagrad, RMSprop, AdaDelta and many others.
How they works?
Basically they study the error functions finding a lot of *saddle points*: they are points in where you have two directional derivatives equal to 0, but in one direction it is a minimum, in the other is a maximum. Neural Networks have a lot of these points and having some stochasticity, which means not necessarily going directly to the minimum but having some noise, it will help a lot.

3.2.4 Batch Normalization

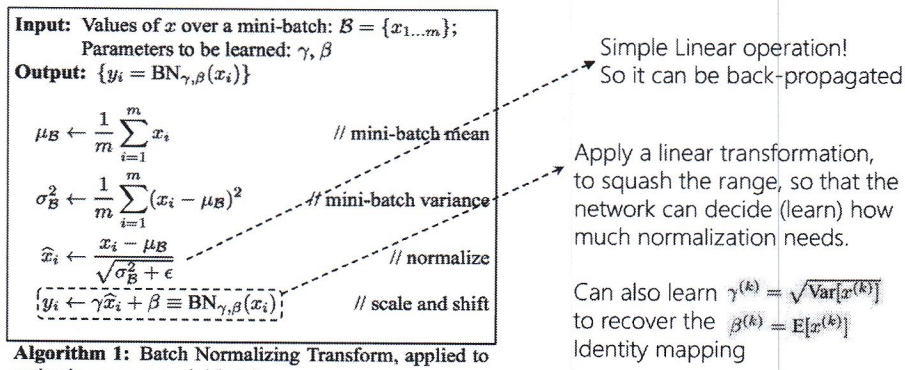
Networks converge faster if the inputs have been whitened (zero mean and unit variances) and are uncorrelated to account for *covariate shift*. With Neural Networks we can have internal covariate shift so normalization could be useful also at the level of hidden layers.

Batch Normalization is a technique to cope with this:

- Leverages the fact that normalization is a differentiable
- Forces activations throughout the network to take on a unit Gaussian at the beginning of the training
- Adds a BatchNorm layer after fully connected layers (or convolutional layers), and before nonlinearities.

- Can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable way.

So the idea is to take the original data and normalize them centering the data with zero-mean and unitary variance for each feature. In that way you are sure that your data are in a square of size 1. This can be done for the input too. It is a specific layer network and you can have back-propagation even with this kind of layer (which is removing the constant value and dividing by the constant value) because it is differentiable. You can learn or adapt the values of the mean and the variance.
In mini-batches, another relevant thing you would like to have is that all the batches have the same distribution, same mean and variance. The idea is to add as part of the training a normalization procedure: normalize the data as they pass through the layers. Obviously you can do it with Batch Normalization.



Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

As you have noticed, during: $\widehat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma^2_B + \epsilon}}$ we add some random noise to avoid division by zero.

In practice:

1. Each unit's pre-activation is normalized (mean subtraction, stddev division)
2. During training, mean and stddev is computed for each minibatch
3. Backpropagation takes into account the normalization

4. At test time, the global mean/stddev is used (global statistics are estimated using running averages during the training)

It has shown to:

- Improve gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization slightly reduces the need for dropout

4 Convolutional Neural Networks

Convolutional neural networks are the tools for performing most of the visual tasks, like classification and detection, and they are probably the most widely known deep learning model.

4.1 The Feature Extraction Perspective

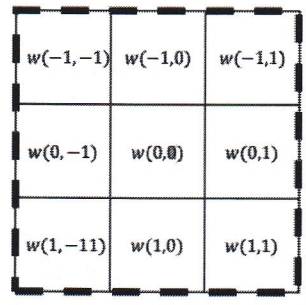
Instead of just feeding directly our image to a classifier, we perform some preliminary step to extract features from the images, we will perform some preliminary algorithm that maps our image into a vector which is informative for defining the label where the image belongs to. You are not trying to directly handle the image has a vector but you extract some meaningful information out of the image in order to feed this additional information to a standard classifier. Feature extraction is a winning strategy, for example it lets you embed some a-priori additional information you have on your data. One way is to do exploratory data -analysis and to draw some rule from what comes out, identifying the various classes by some rule based feature; the problem with this kind of features is that is difficult to design complex relations which involve many variables in which sometimes it's not easy to separate the variables. Neural networks and things like SVM on the other side are able to learn more complex relationships among variables that include many variables. An option is to add hand written features that come from our expertise to the neural network. For natural images, the hand crafted approach doesn't work and you have to go to the data driven approach. This means using Convolutional Neural Networks. Before using CNN, we need to understand how convolution works.

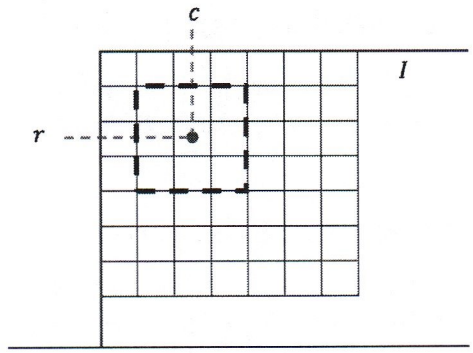
4.2 Convolution

Convolution it's, up to some change of sign, exactly as *correlation*. Convolution is a linear transformation applied to an image.

$$T[I](r,c) = \sum_{(x,y)\in U} w(x,y) * I(r-x, c-y)$$

If you have an image and a filter (which is nothing more but a small image); we can consider the weights as a filter H and the filter entirely defines the convolution, which operates the same in each pixel:





The convolution of the input image against this filter is still an image. The size of the output image in principal should be the same of the input one. The same operation is performed in each pixel of the input image and we get a value for each pixel, which is gonna be just the linear combination of the pixel values in a neighbor of the pixel (which has the same size of the pixel).
So you take your filter, you place it over the image right in the position where you want to compute the output (in this case you place the filter so that the center goes in (r, c)) and the output is the linear combination of the pixel values using these coefficients as weights for the linear combination.
Convolution is the process of adding each element of the image to its local neighbors, weighted by the kernel. What happens on boundaries?
There are different options; the simplest thing is to use 0 padding, so you set 0s where the filter goes out of the image (so its black), otherwise you can compute the output of the convolution only at those pixels where the filter can be entirely included in the image.
Note that a minus sign means that you have to flip your image, it's exactly the same if you flip your filter, it's important only if you define the filters but in general we will learn the w. Convolution was already used in signal processing, it's used to compute the Fourier transformed, it wasn't invented for neural networks.
Correlation is the same but in correlation you have the plus sign whether in convolution you have the minus sign.

4.3 CNN Layers

CNN are networks that take as input an image and provide as output as any neural networks, for example for classification, a set of probabilities associated to each class. For example, for the CIFAR dataset we take as input 32x32x3 images and we provide as output 10 values which are the posterior probabilities over the 10 classes. In a CNN you see some structure of the image which is preserved, you see the feature maps. As you move deeper you see that they are increasing in number an decreasing in size, each of this images are layers of the volume; feature maps become smaller but deeper so the height and width of the volume decrease whether the dimension increases. In order to perform this operation that reduces the size and increases the dimension, you use 3 different layers:

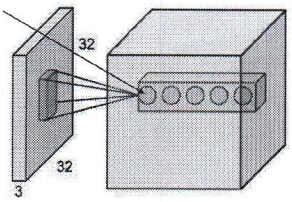
- convolutional layers
- activation functions

- pooling layers, in particular max pooling

4.3.1 Convolutional Layers

In a CNN you have a set of filters given for each layer and these filters represent the network parameters. Let's see what is the output of the convolution of this image against a filter which is, for example, 3x3x3:

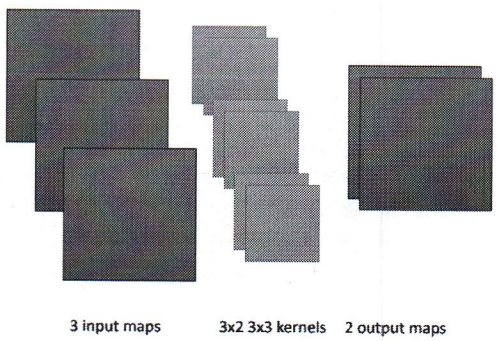
$$y = \sum_i w_{i,j}^1 x_i + b^1$$



Remember that each filter has to have the same depth as the input volume, in this case you have 3 channels so the filter will be 3x3x3. Each pixel in the output image is gonna be given by the filter and the pixel values of your image in a neighbor of the same pixel, as we saw before. What we have in that point are the weights w multiplied by the input x_i plus some bias, it's a linear combination. You compute different filters and the output images becomes deeper. Increasing the depth is done simply by adding multiple filters; the parameters of a convolutional layer are all the weights that are stored in filters plus one bias associated to each filter.

As you can imagine, one input image with one filter gives one output map, two filters give two output maps etc. The important thing is that the filter has to have the same depth as the input map, and the filter is applied to the whole spatial extent of the input, and adding more filters increases the number of output maps.

If you feed a network with 3 input maps, what happens is that all your filter will be 3 different layers and the filters corresponding to the moving output map are those that are also moving. How many parameters does this layer have?



Each layer has 9 parameters (3x3), but each filter has 3 layers so its 9x3=27 for each filter, x2=54; moreover you have to consider the bias, one for each filters. So: 54 for the filters + 2 biases = 56 trainable parameters. What's really important is that convolution mixes all the input channels. Typically you have small filters in these networks, they have a negligible size w.r.t. the image size.

4.3.2 Activation Functions

Convolutional layers perform linear combinations, so what you get at the end is again a linear combination. Like in the multilayer perceptron, you have to introduce some non-linearity and you do this with the activation function, otherwise the CNN would be equal to a linear classifier. The most popular is the Rectified Linear Unit (RELU):

$$T(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

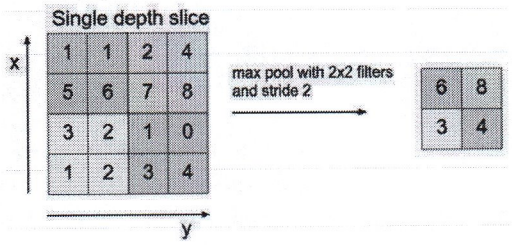
Another option is a variant of the RELU called leaky RELU:

$$T(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0.01 * x & \text{if } x < 0 \end{cases}$$

These functions activates the neurons performing some thresholding on the feature maps. Other options are:

4.3.3 Pooling Layers

These layers reduce the spatial extent of the input volume. The most popular is the MaxPooling layer which operates independently on each slice.



4.3.4 Fully Connected Layer

When you get to the end reducing the spatial extend with the pooling layers, what you get is that your volume will have a special extent equal to 1, so it's just a large vector. This vector is the feature vector and with it you train a multilayer perceptron, which is called a fully connected layer.

The first CNN goes back to 1998, the LeNet-5. After that NN were forgotten for a while until a CNN model won the ImageNet competition of 2012, AlexNet, which was the first deep CNN. There are two main reasons for which they had difficulties in spreading after 1998: heavy computation and lack of data. Moving to these days, a lot of things have changed: on one side new mosfet technologies, fast CPU, graphic computation, on the other side amazingly increasing data of all kinds are created each second. These two factors, parallel fast computing and a lot of training data, really played a crucial role for the escalation of neural networks.

5 Image Segmentation

The goal of Semantic Segmentation is:
Given an image I, associate to each pixel (r, c) a label from Λ.
The result of segmentation is a map of labels containing in each pixel the estimated class.

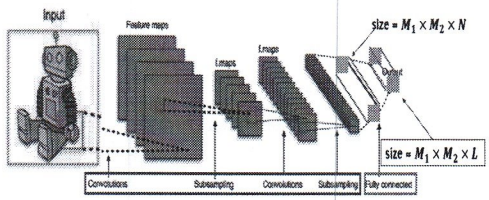
Remark: segmentation does not separate different instances belonging to the same class. That would be instance segmentation.

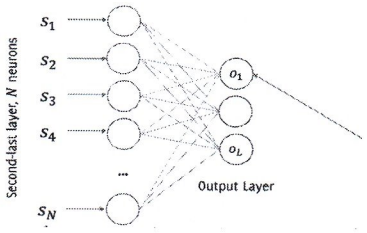
The training set is made of pairs (I, GT), where the GT is a pixel-wise annotated imgae over the categories in Λ.

5.1 Fully-Convolutional Networks

CNNs are meant to process input of a fixed size. The *convolutional* and *subsampling* layers operate in a sliding manner over image having arbitrary size. The *fully-connected* layer constrains the input to a fixed size.

If I fed the network with larger images, convolutional filters can be applied to volumes of any size, yielding larger volumes in the network until the FC layer. What breaks is in the FC layer because instead of having $1 * 1 * N$, it processes a volume of size $M_1 * M_2 * N$. Thus, CNN cannot compute class scores, yet can extract features!





$$o_i = \sum_{j=1:N} w_j^i s_j + b_i$$

$$o_i = (\boldsymbol{w^i})' \boldsymbol{s} + b_i$$

$$o_i = (\boldsymbol{w_i} \circledcirc \boldsymbol{s})(0,0) + b_i$$

However, since the **FC** is **linear**, it can be represented as convolution! Weights associated to output neuron i:

$$\boldsymbol{w_i} = \{w_{i,j}\}_{i=j:N}$$

A FC layer of L outputs is a 2DConv Layer against L filters having size $1 * 1 * N$. Each of these convolutional filters contains the weights of the FC for the corresponding output neuron. This transformation can be applied to each hidden layer of a FC network placed at the CNN top.
For each output class we obtain an image, having:

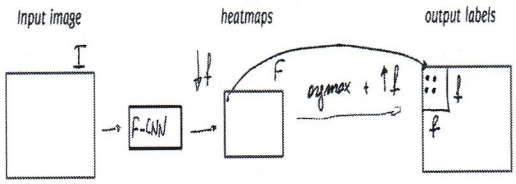
- Lower resolution than the input image
- Class probabilities for the receptive field of each pixel

So how many filter should I use to replace the fully-connected layer into a convolutional layer?
L filters: one filter per each output neuron and each filter has size $1 * 1 * N$. We can move each fully-connected layer that takes N input and provides L output as a convolutional

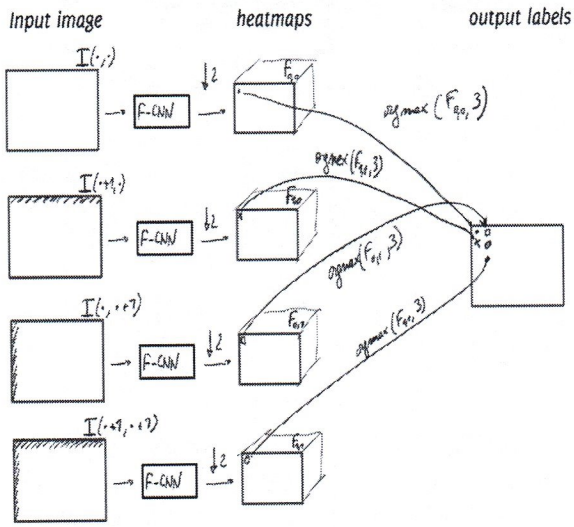
layer having L filter each one having size $1 * 1 * N$. This transformation can be applied if you have a multi-layer perceptron too. What you get is, given an image, you have multiple small images (*heatmaps*), one for each classes, and these small image contains a score which tell you how much likely there is something that looks like something, for instance a wheel.

5.1.1 Simple Solutions

The first way to perform segmetation is: take the heatmaps, for each pixel take the most likely class. So given an input image to fed a fully-convolutional network, you get the heatmaps: with these multiple heatmaps, for each pixel we take the argmax among those heatmaps for that pixel, which corresponds to the most probable class. However that would be a very **coarse estimate**



Another option would be the *Shift and Stitch*:

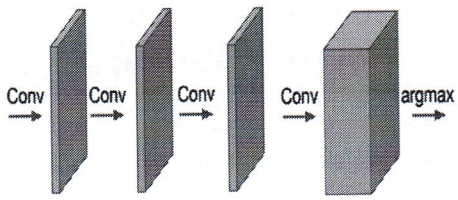


Assume there is a ration f between the size of the input and of the output heatmap. You have to:

- Compute the heatmaps for all f^2 possible *shifts* of the input $(0 \leq r, c < f)$
- 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

This exploit the whole depth of the network, however the upsampling method is very rigid.

The last possible approach is to perform *Only Convolutions*: you can design a network without any downsampling, without any pooling. In principle, you can get in output an image with the same size, but the major drawback is that you do not increase enough the receptive



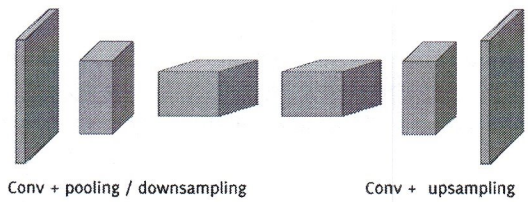
field for having meaningful information ⇒ Very inefficient.

5.1.2 Main Solution

On the one hand we need to "go deep" to extract high level information on the image. On the other hand we want to stay local not to loose spatial resolution in the predictions. Semantic segmentation faces an inherent tension between semantics and location:

- global information resolves what, while
- local information resolves where

Combining fine layers and coarse layers lets the model make local predictions that respect global structure.
An architecture like the following would probably be more suitable for semantic segmentation:

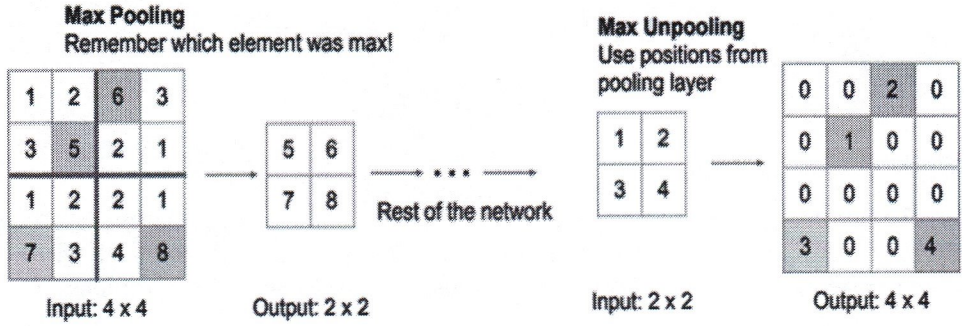


In the first part you go deep encoding semantic information; in the second part, perform upsampling and get high resolution output to recover local information: this part is meant to upsample the predictions to cover each pixel in the image. Increasing the image size is necessary to obtain sharp contours and spatially detailed class predictions.

5.2 Upsampling

Linear upsampling of a factor f can be implemented as a convolution against a filter with a fractional stride $1/f$.
Upsampling filters can thus be learned during network training.
Upsampling can be performed in several ways:

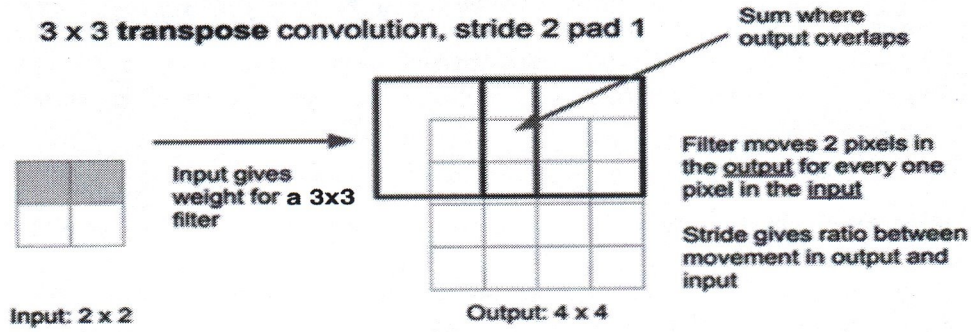
- Nearest-Neighbor: the output is of a larger size and the pixel value correspond to the value of the nearest pixel.
- Bed of Nails: the output pixels correspond to the input ones, located in the top left corner of the output sub-squares, and the remaining pixels are filled with zeros.
- **Max Unpooling**: you have to keep track of the locations of the max during max-pooling. Remember which element was the max and, during max unpooling phase, use those positions to upsample the input and locate the max valued pixel in the same position as original.



How to perform upsampling?

Transpose Convolution can be seen as a traditional convolution after having upsampled the input image. In this way our network can learn how to upsample optimally.
A classical convolution operation forms a many-to-one relationship: in general, the convolution operation calculates the sum of the element-wise multiplication between the input matrix and the kernel matrix; from a matrix you get only one value.

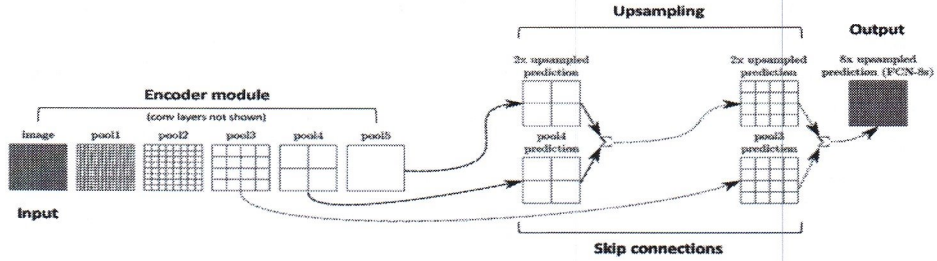
The idea of Transpose Convolution is to going backward, it's a one-to-many relationship: we want to associate one value to a matrix of values. If your input is a $2*2$ matrix, you want a $4*4$ output and you have a filter of size $3*3$ [stride 2, pad 1], get the first pixel value of the input and multiply it by the kernel values: this operation returns a filter that has been rescaled by the value of the image, so the input gives weight for the $3*3$ filter. Compute the same operation for the other pixels and sum the values where output overlaps.



These predictions however are very coarse. Upsampling filters are learned with initialization equal to the bilinear interpolation. The solution is to introduce **Skip Connections**.

5.3 Skip Connections

There was some information that was captured in the initial layers and was required for reconstruction during the up-sampling. If we would not have used the skip architecture that information would have been lost. So the information that we had in the primary layers can be fed explicitly to the later layers using the skip architecture.



Supplement a traditional "contracting" network by successive layers where convolution is replaced by transpose convolution.
Upsampling is ubiquitous and upsampling filters are learned during training.
Upsampling filters are initialized using bilinear interpolation.
This approach yields 3 networks:

- Train first the lowest resolution network (FCN-32s)
- Then, the weights of the next network (FCN-16s) are initialized with (FCN-32s)
- Tha same for FCN8s
- All the FC layers are converted to convolutional layers $1 * 1$

Retaining intermediate information is beneficial, the deeper layers contribute to provide a better refined estimate of segments.

How to train this network?
No need to train patchwise and then transform weights in filters.
It is also possible to directly train the whole network on the entire image:

- Define a classification loss in each pixel
- Average this loss over the receptive field of each pixel

Derivative of the sum of the losses can be easily computed and the whole network can be trained through backpropagation.
The *patch-based* way:

1. Prepare a training set for a classification network
2. Crop as many patches $\boldsymbol{x_i}$ from annotated images and assign to each patch label corresponding to the path center
3. Train a CNN for classification from scratches, or fine tune a pre-trained model over the segmentation classes
4. Once trained the network, move the FC layers to $1 * 1$ convolutions
5. Train the upsampling filters

The classification network is trained to minimize the classification loss l over a mini-batch:

$$\hat{\theta} = \min_{\theta} \sum_{x_j} \ell\left(x_j, \theta\right)$$

where $\boldsymbol{x_j}$ belongs to a mini-batch. Batches of patches are randomly assembled during training. It is possible to resample patches for solving class imbalance.
It is very inefficient, since convolutions on overlapping patches are repeated multiple times.

The *full-image* way: since the network provides dense predictions, it is possible to directly train a FCNN that includes upsampling layers as well.
Learning becomes:

$$min \sum_{x_j} \ell\left(x_j, \theta\right)$$

Where $\boldsymbol{x_j}$ are all the pixels in a region of the input image and the loss is evaluated over the corresponding labels. Therefore, each path provides already a mini-batch estimate for computing gradient.
The full-image way works as follows:

1. FCNN are trained in an end-to-end manner
2. No need to pass through a classification
3. Takes advantage of FCNN efficiency, does not have to re-compute convolutional features in overlapping regions

Drawbacks and solutions:

- Mini-batches are assembled randomly. Image regions are not. To make the estimated loss a bit stochastic, adopt random mask

$$min \sum_{x_j} M\left(x_j\right) \ell\left(x_j, \theta\right)$$

  where $M(\boldsymbol{x_j})$ is a binary random variable

- It is not possible to perform patch resampling to compensate for class imbalance:

$$min \sum_{x_j} w\left(x_j\right) \ell\left(x_j, \theta\right)$$

  where $w(\boldsymbol{x_j})$ is a weight that takes into account the true label $\boldsymbol{x_j}$

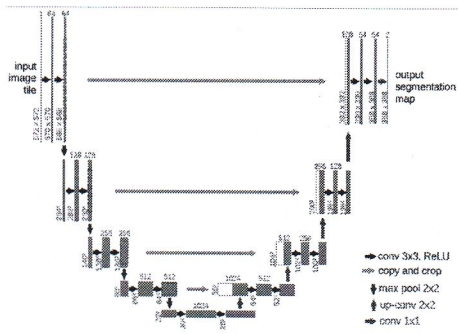
5.3.1 Comments

Both learning and inference can be performed on the whole-image at-a-time. Both in full-image or batch-training it is possible to perform transfer learning/fine-tuning of pre-trained classification models.
Accurate pixel-wise prediction is achieved by upsampling layers.
End-to-end training is more efficient than patch-wise training.
Being fully-convolutional, this network handles arbitrarily sized input.

5.4 U-Net

It is the standard for segmentation. The network is composed by a *contracting* part and an *expansive* part, like an encoder-decoder architecture.
It takes an image as input, which goes through convolutional layers until the deepest part. Then the upsampling part of the network starts, based on skip connections to get informations of the contracting part.
There are no fully-connected layers: there are L convolutions against filters $1 * 1 * N$, to yield predictions out of the convolutional feature maps.
The output image is smaller than the input image by a constant boarder. Major differences w.r.t. (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
- Use excessive data-augmentation by applying elastic deformations to the training images



The *contracting path*:
Repeats blocks of

- $3 * 3$ convolution + ReLU
- $3 * 3$ convolution + ReLU
- Max-pooling $2 * 2$

At each downsampling the number of feature maps is doubled
The *expanding path*: Repeats blocks of:

- $2 * 2$ transpose convolution, halving the number of feature maps
- Concatenation of corresponding cropped features
- $3 * 3$ convolution + ReLU
- $3 * 3$ convolution + ReLU

The main part consists in "aggregation through concatenation + convolution to mix different feature maps in a learnable manner".

Training:
Full-image training by a weighted loss function:

$$\hat{\theta} = \min_{\theta} \sum_{x_j} w\left(x_j\right) \ell\left(x_j, \theta\right)$$

where the weight:

$$w(\boldsymbol{x}) = w_c(\boldsymbol{x}) + w_0 e^{-\frac{(d_1(\boldsymbol{x}) + d_2(\boldsymbol{x}))^2}{2\sigma^2}}$$

- w_c is used to balance class proportions. $w_c(x)$ is a large number for class c that are not very representative in the training set, and it is small for class c that are very representative.
- d_1 is the distance to the boarder of the closest cell
- d_2 is the distance to the boarder of the second closest cell
- The first part of the summation takes into account class unbalance in the training set; the second part enhances classification performance at boarders of different objects: in fact, this term is large at pixels close to boarders delimiting object of different classes
- For the second term, you have an high weight if the distance d1 and d2 are large, where d1 and d2 are the distances between the label containing the object and the closest label that is different. Instead when d1 and d2 are small, the weights are small.

5.5 Global Averaging Pooling

5.5.1 Network in Network

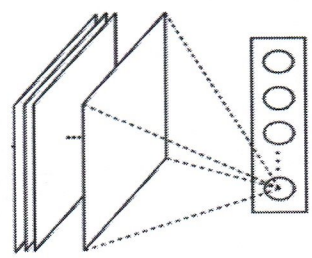
Introduce two different contributions. When you take a convolutional layer, the output is a linear combination of the inputs. So a linear combination could be too poor approximation to learn. We can use a **Mlpconv layers**: (Multi-layer-perceptron convolutional layer) so a fully-connected network over a small portion of the input; 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 networks used in a sliding manner on the whole image.
The same multi-layer perception MLP is used through all the image, this is why MLP-conv layer.
Each layer features a more powerful functional approximation than a convolutional layer which is just linear + ReLU.

The other contribution is due to the introduction of **Global Averaging Pooling** layer: instead of a FC layer at the end of the network, compute the average of each feature map.

- The transformation corresponding to GAP is a block diagonal, constant matrix
- The transformation of each layer in MLP corresponds to a dense matrix

The reasons behind the introduction of GAP are that *FC layers* are *prone to overfitting*: they have many parameters; the dropout was proposed as a regularized that randomly sets to zero a percentage of activation in the FC layers during training.

Global Averaging Pooling Layer



The GAP strategy instead is **remove the FC**

layer at the end of the network and **predict by softmax** after the GAP.
The advantages of Global Averaging Pooling Layers:

1. No parameters to optimize, lighter network less prone to overfitting
2. More interpretability, creates a direct connection between layers and classes output
3. This makes GAP a structural regularizer
4. More robustness to spatial transformation of the input images
5. The network can be used to images of different sizes

Classification is performed by a softmax layer at the end of the GAP.

6 Localization and Object Detection

6.1 Localization

The input image contains a single relevant object to be classified in a fixed set of categories. The task is:

- Assign the object class to the image
- Locate the object in the image by its bounding box

The task is a prediction task which involve a discrete output and 4 different outputs which are the bounding box limits.

6.1.1 Simplest Solution

The network have to be both a *classification* network and a *regression* network to identify bounding boxes, so you need to combine two different losses. The training loss has to be a single scalar since we compute gradient of a scalar function w.r.t. network parameters. The idea is to minimize a multitask loss to merge two losses:

$$\mathcal{L}(x) = \alpha\delta(x) + (1 - \alpha)\mathcal{R}(x)$$

where the hyper-parameter α is used to balance the losses to make the two losses in a similar range. Watch out that α directly influences the loss definition, so tuning might be difficult, better to do cross-validation looking at some other loss.
So you will have *two different ends* of your network, one part with L (number of classes) neurons, and the other will predict the 4 bounding boxes.

Human-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 approach is very simple to design and train. 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, possibly normalized w.r.t. the bounding box enclosing the human.

6.1.2 Weakly-Supervised Localization

Perform localization over an image without any annotating bounding box. So you train your network as a classification network to assign to each image a label, but you want also your network to perform localization. The only form of supervision is the classification label, not the bounding box: only the classification labels are provided.

This is a sort of Global Averaging Pooling *revisited*: the advantages of GAP layer extend beyond simply acting as a structural regularizer that prevents overfitting; in fact, the network can retain a remarkable localization ability until the final layer. By a simple tweak it is possible to easily identify the discriminative image regions leading to a prediction.

> *A CNN trained on* ***object categorization*** *is successfully able to localize the discriminative regions for action classification as the object that the humans are interacting with rather than the humans themselves*

Your network takes in input an image and provides a big volume. The larger of image, the larger the volume is. In case of classification, you have to squeeze down this volume to a fixed size, which is the input of the fully connected layer.
An option is to *resize* the image, the other option is to introduce another *GAP layer*: average the values of each slice of the volume. So you have a vector with a value for each one of the n slices and, at the end, what you get is a vector of length n.
GAP is shown to perform better in some cases than Dropout in order to perform Regularization. This is why it is useful, it's reduce overfitting and can be used with images of any shape.

Figure 2: Heatmaps

In the paper, GAP it is also presented as a tool to modify slightly your network and get instead a classification network, a network that perform a sort of Localization: it is able to localize discriminative regions for the classification task at the object that human are interactive with. In fact, the heatmaps show which is the image region which has mostly influenced the network decision. This kind of heatmap is perfect to identify bounding boxes.

The nice part of the story is that if you take any classification network, you introduce a GAP layer, you get for free a network that is able to provide that sort of heatmap, so you are able to perform Localization.

This approach of identifying exactly which regions of an image are being used for discrimination is called **Class Activation Map**: it is very easy to compute since it just requires an FC layer after the GAP and a minor tweak.

Considering Fig.3, a very simple architecture made only of convolutions and activation functions leads to a final layer having:

- n feature maps $f_k(:,:)$ having resolution "similar" to the input image
- a vector after GAP made of n averages F_k

The FC layer after the GAP it's the output layer which computes S_c for each class c by the weighted sum of F_k, where weights w_k^c are defined during training. Then, we can extract the class probability P_c via softmax, for each class c. The weights w_k^c encodes the importance of F_k for the class c

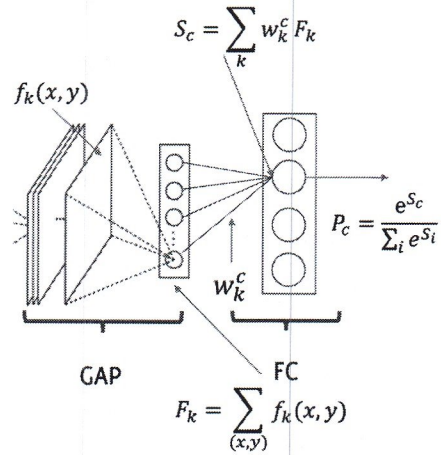


Figure 3: GAP layer

$$S_c = \sum_{k=1}^{N} w_k^c F_k = \sum_{k=1}^{N} w_k^c \sum_{x,y} f_k(x,y) = \sum_{x,y} \sum_{k} w_k^c f_k(x,y)$$

So you can construct an image where in each pixel there is the linear combination of all the weights corresponding to a slices rescaled by the corresponding weighth w_k^c. The image in the output is exactly the heatmap which highlights which part of the image contributes more on the decision of the network. The results is called Class Activation Map which is defined as:

$$M_c(x, y) = \sum_k w_k^c f_k(x, y)$$

where $M_c(x, y)$ directly indicates the importanc of the acrivations at (x, y) for predicting the class c. Thanks to the softmax, the depth of the last convolutional volume can differ from the number of classes.

When we feed the CNN with an input image, we get a convolutional block (the volume) which becomes smaller as the input pass through the network. We need to perform an upsampling to obtain the an heatmap of the same size of the input image. Another option would be to perform a sort of U-net as the segmentation problem, obtaining more resolution but less semantic.
As shown in Fig.4, the weights represents the importance of each feature map to yield the final prediction.

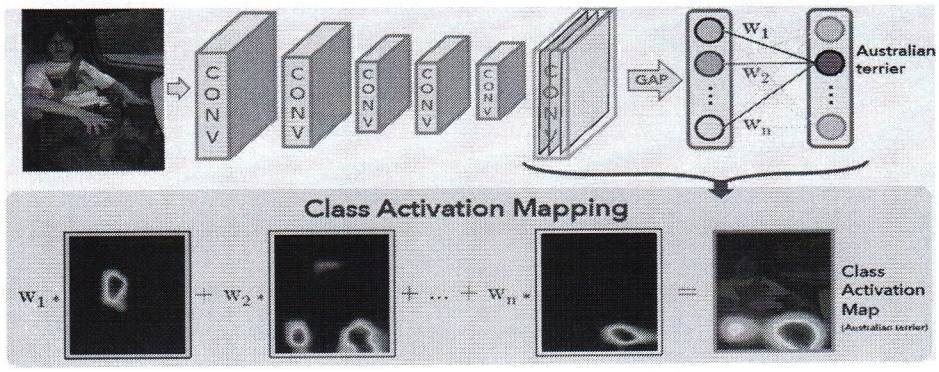


Figure 4: Class Activation Map

Remarks:

- CAM can be included in any pre-trained network, as long as the FC layer is removed
- The FC used for CAM is simple, few neurons and nohidden layers
- Classification performance might drop (in VGG removing FC means loosing 90% of parameters)
- CAM resolution (localization accuracy) can improve by "anticipating" GAP to larger convolutional feature maps
- GAP: encourages the identification of the whole object, as all the part of the values in the activation map concurs to the classification
- **GMP (Global Max Pooling)**: it is enough to have a high maximum, thus promotes specific discriminative features

The fully-connected part of the network is very simple, but it requires changing the network and loosing performance (adding a fully-connected layer drops the performance).

Grad-CAM computes that weights to the gradient of back propagation, you don't even have to change the network (a fully-connected network at the end is not required) and it is good because if you get a pre-trained network for doing other stuff (captioning, object detection) you don't have to change it. You can use these heatmaps even attaching them to an existing network without the need to retrain them.

6.2 Object Detection

Given a fixed set of categories and an input image which contains an unknow and varying number of instances, draw a bounding box on each object instance. A training set of annotated images with label and bounding boxes for each object is required. Each image might require a different number of outputs, depending on the number of object detected.

In fact, the tricky part is that the number of output is not pre-defined, it depends on the input image.
Different solutions are proposed:

6.2.1 Sliding Windows

Define a certain patch (a sort of box that captures a portion of the image) size and slide this patch on the image, classify the content in that patch. If you get some output with high probability then you might take that as output of your object detection algorithm. A pre-trained model is meant to process a fixed input size, so you have to slide on the image a window of that size and classify each region, assigning the predicted label to the central pixel. There is a special class that is 'background' which means that there is no relevant object inside that patch.
It is very inefficient since for identify an object, e.g. a car, entirely you need a bigger bounding box. So theoretically you have to test different patch size. It does not re-use features that are "shared" among overlapping crops. Inefficient also if you consider only a single patch size: take a patch and slide means compute convolution multiple times for each pixel because each pixel belongs to multiple patches. Moreover, each patch feeds the network independently from the others and all the operation are repeated multiple times. The only advantage is that there is no need of retraining the CNN.

6.2.2 Region Proposal

Region Proposal algorithms (and networks) are meant to identify bounding boxes that corresponds to a candidate object in the image.

The first network performing object detection were actually based over these techniques. These algorithms have an high recall (if there is an object, those algorithms provide a bounding box), but very low precision because they provide a lots of region proposal out of a image.

The idea is: apply a region proposal algorithm → Classify by a CNN the image inside each proposal regions

6.2.3 R-CNN: Region-CNN

Object detection by means of region proposal. So given an input image, run the Region Proposal algorithm and extract a huge amount of proposals, e.g.. 2K of proposals. Modify the content of your region proposal to become a square since the network is a standard CNN which expects a fixed input size.

Taking the cowboy image, once extracted the regions, you squeeze the cowboy to be a square. The major drawback is that in this way you lost the proportion, relevant to understand the content.

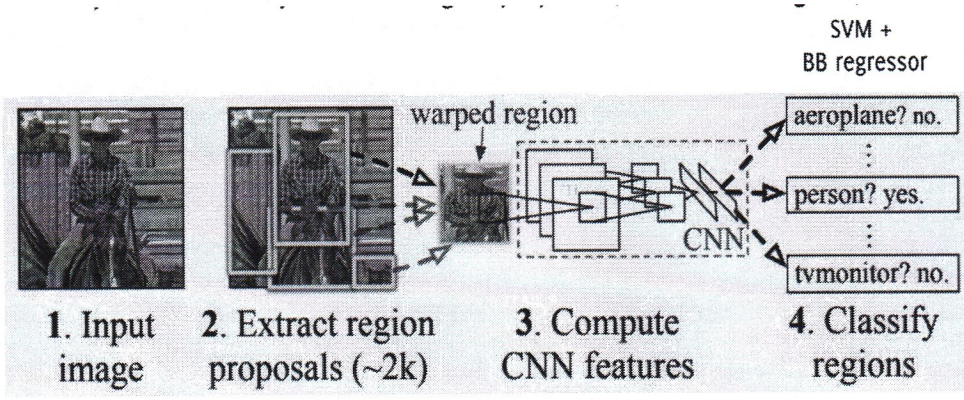


Figure 5: R-CNN

So Region Proposal detects the input, CNN extracts the features. At the end it is used a SVM in order to classify and find out what will be the label of the region proposal. On top of that, a bounding box regression in order to improve the localization, so there will be different losses.
There was no learning algorithm in the Region Proposal, so a very high recall. Instead, the CNN is fine-tuned during training.
However, this architecture has some limitations:

- Ad-hoc training objectives and not an end-to-end training: fine-tune network with a softmax classifier, train post-hoc linear SVMs, train post-hoc bounding-box regressions
- **Region proposal are from a different algorithm** (Region Proposal) and that part has not been optimized for the detection by CNN
- Training is **slow**, you get a lot of proposal and for each of them you have to extract the features by the CNN → takes a **lot of disk space** to store features.
- **Inference is slow** since the CNN has to be executed on each region proposal (**no feature re-use**)

6.2.4 Fast R-CNN

To save a lot of time, especially in inference time, instead of cropping the regions on the image and passing the image, cropped and resized, to another network, you can just **project the region proposals** from the input image over the last fully-convolutional layer: you have to perform all the CNN just once.

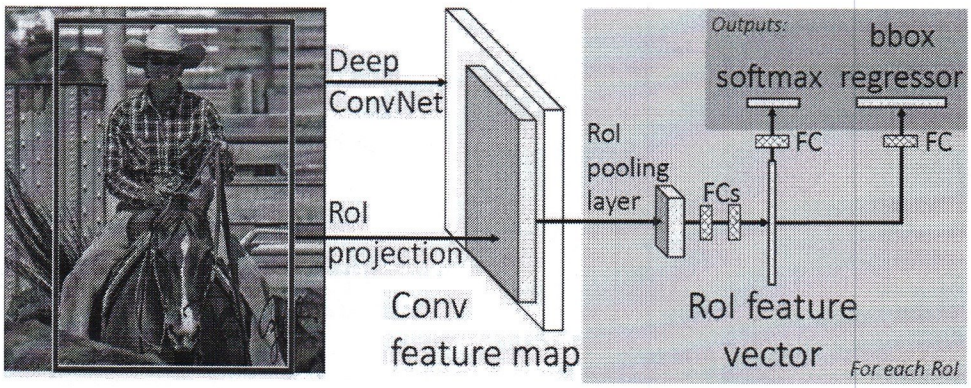


Figure 6: Fast R-CNN

1. The wholw image is fed to a CNN that extracts feature maps
2. Region proposal are identified from the image and progected into the feature maps. Regions are directly cropped from the feature maps, instead from the image → 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 HxW from each region proposal. Each ROI in the feature maps is divided in a HxW grid and then max-pooling over this provides the feature vector
4. The FC layers estimate both classes and BB location (BB regressor). A convex Combination of the two is used as a multi-task loss to be optimized (as in R-CNN, but no SVM here)
5. Training in an end-to-end manner

Some issues: here you are using this network only to extract features; once you have these feature and you have to feed an architecture with them for classification, you have to fix the input size. So the problem has been postponed: what Ross (the creator of Fast R-CNN) does is *ROI Pooling layer*: whatever partial size the projected region proposal has, you divide it in HxW blocks and you perform max pooling on these regions.
What you get is a volume with a fixed special extent: it's like max pooling which works on larger neighbours if the regions are larger, and smaller vice-versa.
Still the cowboy is squeezed but it is *squeezed on the features* and with the max pooling. Then you can predict the class and the bounding box location (softmax and BBox regressor).

By doing this you can *backpropagate this loss through the whole network*, which becomes incredibly faster than R-CNN during testing. Now that convolutionss are not repeated on overlapping areas, the vast majority of test is spent on ROI extraction: most of inference time is spending on extracting Region Proposal.

6.2.5 Faster R-CNN

Faster R-CNN gets close to real-time object detection. You move Region Proposal inside the network: you train a network in order to extract the convolutional features, from them you extract a region proposal. You do not look at the image anymore, you look at the feature map directly.

Instead of the ROI extraction algorithm, a region proposal network (**RPN**) is used in order to extract a region proposal from the convolutional features. So RPN operates on feature maps of the last convolutional layers. The remaining operation are very similar to a Fast R-CNN

You feed your CNN with an input image, you get a volume (the convolutional features) and there you run RPN.
The Region Proposal network works as follows: it takes a filter of $3x3$ and slide the feature maps. It maps (by 1x1 convolution) the region to a lower dimension vector. In each point it consider k anchor boxes, i.e. different ROI size/proportions (k-possible templates for region), thus there $HxWxk$ candidate anchors.
The classification network (*cls* network) is trained to predict the object probability, i.e. that each anchor contains an object *[obj, no-obj]* $\rightarrow 2k$ probability estimates. The region network (*reg* network) is trained to adjust the anchor to the object ground truth $\rightarrow 4k$ estimates.
There is no need to design different RPN to account for the k different anchors. **Anchors just influence the way labels of RPN are comuted**.
Training now involves 4 losses: RPN classify object/non-object, RPN regression coordinates, final classification score, final BB coordinates. During the training, object/non-object ground truth is defined by measuring the overlap with annotated BB.
The network becomes much faster also because you don't have to design a different network for each shape of region proposal, you just train the network to predict for each of these k different outputs, which correspond to an object in the image.
At test time:

- Take the top 300 anchors according to their object scores
- Consider the refined boundig box location of these 300 anchors
- These are the ROI to be fed to a Fast R-CNN (ROI pooling to perform the downsampling)
- Classify ROI

Therefore, Faster R-CNN provides to each image a set of BB with their objectness score.
It's still a **two stage detector**, where in the first stage:

- run a backbone network (e.g. VGG16) to extract features
- run the Region Proposal network to estimate circa 300 ROI

In the second stage (the same as in Fast R-CNN):

- Crop features through ROI pooling (with alignment)
- Predict object class using FC + softmax
- Predict bounding box offset to improve localization using FC + softmax

6.2.6 You Only Look Once (YOLO)/ Single Shot Detectors (SSD)

R-CNN methods are based on Region proposal, but there are also region-free methods, like YOLO or SSD. Detection networks are indeed a pipeline of multiple steps. In particular, region-based methods make 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

> *We reframe the object detectino as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities*

And solve these regression problems **all at once**, with a large CNN.
These algorithms don't have to go to this two stage phases, they cast all the object detection as bounding box regression problem, starting from the image.

In YOLO:

1. Divide the image in a coarse grid
2. each grid cell contains B *base-bounding boxes* associated
3. For each cell and *base-bounding box* we want to 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)

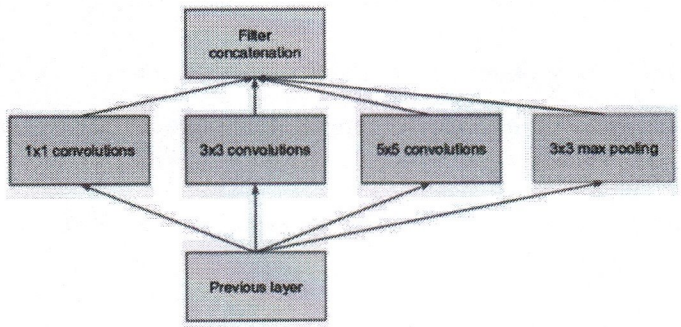
The whole prediction is performed in a single forward pass over the image, by a single convolutional network. Training this network is sort of tricky to assess the loss (matched/not-matched).
YOLO/SSD shres a similar ground of the RPN used in Faster R-CNN. Typically networks based on region-proposal are more accurate, single shot detectors are faster but less accurate.

7 Standard Architectures for Image Classification

We are talking about networks for image classification.

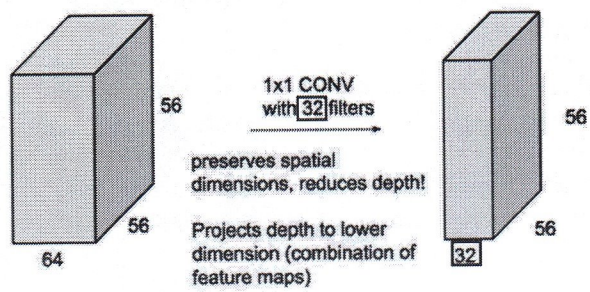
7.1 GooglLeNet and Inception v1

This is the network that won the competition in 2014 achieving a classification error of 6.7%. It is composed of the so called inception modules. The basic idea of the model is this:



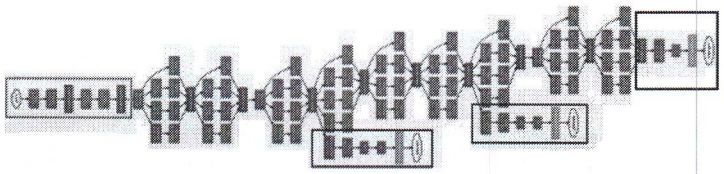
A volume enters not a convolutional layer but in parallel multiple convolutional layers and a max pooling layer. Then all these outputs are combined together by concatenation to produce the layer output. There are multiple filter sizes in this way. The idea is that this was a sort of solution basing on the fact that it's hard to choose the filter size for feature extraction.

Assuming you have a very small volume of 28x28x256, you perform some convolution (depending of the number of filters you will obviously have different numbers) with 128 filters 1x1, 192 3x3, 128 5x5 and 256 3x3 max pooling. To concatenate the output you perform some zero padding to preserve the spatial size, but the depth is increased. There are a large number of operations to be performed due to the large depth of the input of each convolutional layer (858 millions in this example). This is too expensive to compute, this network is expanding its depth preserving the spatial extent. To reduce the computational demand we can use the 1x1 convolution: we can stack 1x1 convolutional blocks right before 3x3 and 5x5 and after the max pooling. These 1x1 blocks are used as *bottleneck layers*, in order to compress the number of features and create another non linearity.



If you take 32 of these 1x1 filters, the output will have size 32 because each 1x1 filter will pass through the image providing one slice of the output. In this way the number of

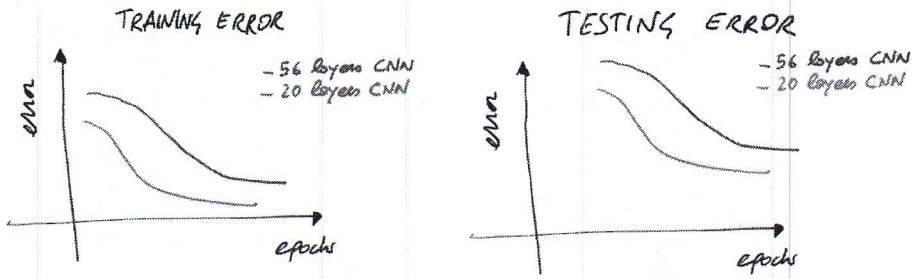
operations is significantly lower. There are overall 27 layers made of 9 inception modules; at the end you don't need a fully connected layer, there's only a global averaging pooling, a linear classifier and a softmax.



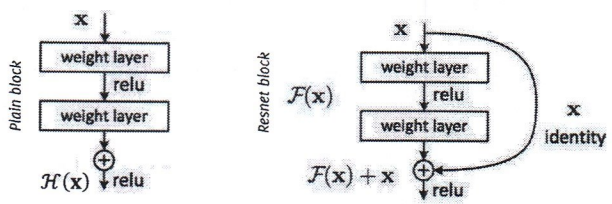
They introduced 2 extra classifiers. By introducing those classifiers, the loss to be minimized is no more the loss of the final network output but it's the sum of those losses and the final result is taken from that. This is just a trick for the vanishing gradient problem.

7.2 ResNet

This model achieved 3% performance the following year. There is a completely change of perspective in this network (which today has different versions). For the Inception the purpose was to handle feature patterns of multiple resolutions combining and replicating different sized filters; here the purpose was to design a network that was incredibly deep stacking more and more layers. They find out this:



The testing error for a network with 20 layers and 56 layers had the same trend, but the one with 56 layers was larger. One can think that this is due to overfitting, but this is not the case because training and testing error should diverge but in this case this does not happen. Probably the reason is that you cannot explain the result as an overfitting problem, is just a problem of minimizing the loss function: training deeper networks is harder with respect to shallow networks. They added a skip connection, which corresponds to the identity, copying the parameters of the shallow network into the deeper network; if you are not good at learning deeper networks at least skip connections provide the identity. What happens is that you take the input of the two convolutional layers, which are the weights and non linearities, and what you do is summing the input to the output:



This is something easier to train and the substantial increase in performance confirms that skip connections were a good idea. This is called ***residual network*** because if $H(x)$ is the function that you would like to learn than instead of learning $H(x)$ what you're learning is something like $\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$. Note that this does not add parameters to the network, it's just a copy and a sum, you are just letting the network solve a different learning problem. The weights in between the skip connections can be used to learn a "delta", a residual, $F(x)$ in order to improve the solution achieved by the shallow network. There are a few consideration, in particular you have to be sure that the spatial extent is the same to perform the sum, so you have to do some padding to adjust the spatial extent.

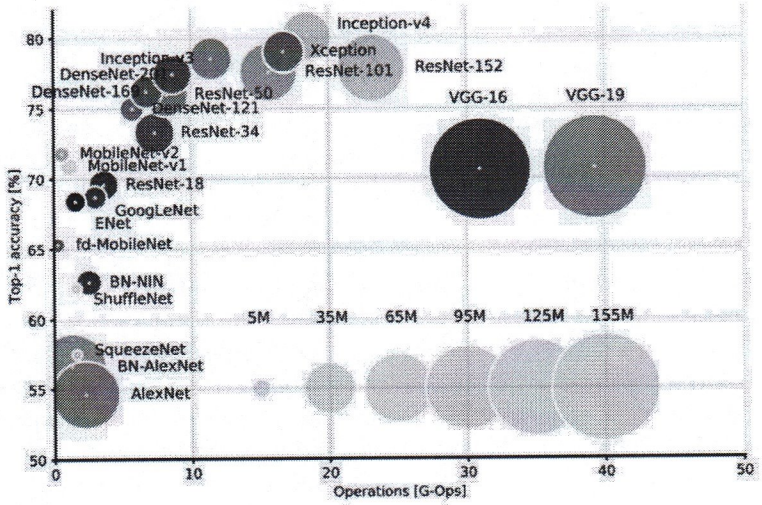
The substantial improvement achieved by the ResNet from the Inception (almost 50% better in the classification error) probably can be justified that deeper networks are able to achieve lower errors, as expected.

7.3 DenseNet

In this architecture there are skip connections everywhere. There are multiple blocks and inside each block the output is connected to each of the previous input for skip connections.

7.4 Comparison

You can see a model comparison using as variables the size of the parameters and the performance achieved. The AlexNet was the first actually and it was below 55%, VGG had a lot of parameters and better performance but not incredibly good.



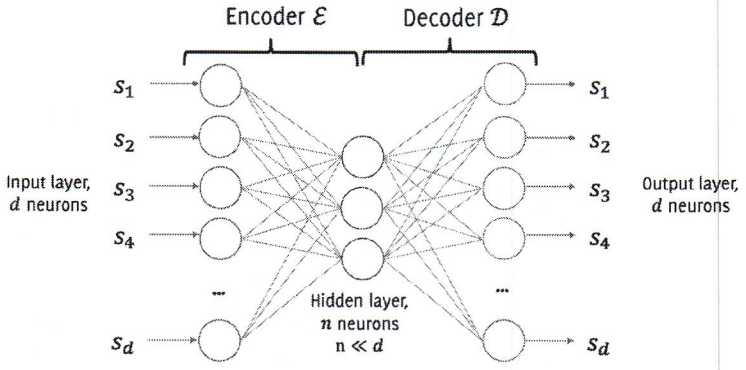
DenseNet and Inception are quite reduced but achieve very good performance. Inception models are the better performing, in particular the InceptionV4 that combines Inception and ResNet. Another important architecture is MobileNet which has very few parameters and it's very efficient but achieves decent performance; this uses separable convolutions and it's been designed for mobile purposes.

8 Generative Adversarial Networks

This is an hyped field, but first we need some background regarding autoencoders.

8.1 Autoencoder

An **autoencoder** is a network which is meant to learn a sort of identity mapping; you are given an input of input neurons and you train a network which has as output some output neurons and the output is meant to be exactly the same input. The first part that goes from the input to the bottleneck is called **encoder**, while in output you have the **decoder**.



This seems some stupid task because you are already given the input and you are trying to learn a neural network to transform the input into itself. The autoencoders are used, for example, for data reconstruction. An autoencoder is a type of artificial neural network used to learn efficient data codings in an unsupervised manner. The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for dimensionality reduction, by training the network to ignore signal "noise". Indeed, you can design your network to have a bottleneck of much fewer neurons than the input, so you learn a function that compresses our input signal to a lower dimension representation, it's a more compact representation of the input signal.
So Autoencoders are non-parametric models that can be trained to reconstruct all the data in a training set. The reconstruction loss is:

$$\sum_{s \in S} \|s - \mathcal{D}(\mathcal{E}(s))\|_2$$

The training of $\mathcal{D}(\mathcal{E}(\cdot))$ is done via standard backpropagation algorithms. This is important in the context of having a latent representation of object; with autoencoders you want to learn a *latent representation* which is meant to provide a compact representation of your input which allows to recover the signal (it's like setting a 1-to-1 correspondence).

You can use these autoencoders also in CNN., for example segmentation networks are similar to the architecture of autoencoders. You can train convolutional autoencoders to find some compressed representation removing the noise.

Remarks:

- Features z are typically referred to as **latent representation**

- Autoencoders typically do not provide exact reconstruction since $n << d$, by doing so we expect the latent representation to be a meaningful and compact representation of the input
- Additional *regularization terms* might be included in the loss function
- More powerful and non-linear representation can be learned by *stacking multiple hidden layers* (deep architectures)

These are also useful to perform in practice to provide a **good initialization** to a network. As soon as you are given a huge amount of train images which are sort of unlabeled (imagine a semi-supervise setting), a way to learn also from not annotated images is the following:

1. first you train, using not annotated images, an autoencoder which tries to reconstruct your input, in a fully unsupervised way;
2. once you have trained the autoencoder, you get rid of the decoder and you keep the compressed representation;
3. than if you are given some training samples, you plug in the rest of your network for classification purposes and you fine-tune the full network on the training samples.

In this case you are also able to exploit an unsupervised problem to provide a good initialization and prevent the risk of overfitting. The feature that you learn for reconstruction may not be too good for classification but in principle the first they should be good for the first convolutional features.

Autoencoders provide a *good initialization* (and reduce the risk of overfitting) because their latent vector is actually a good representation of the inputs used for training.

8.2 Generative Models: Variational Autoencoders

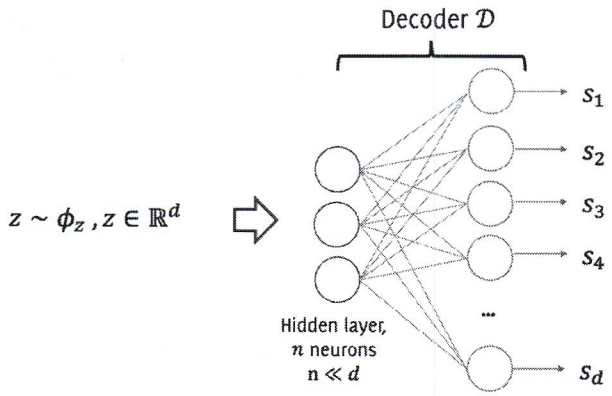
Goal Generative models generate, given a training set of images (data) S, other images (data) that are similar to those in S.

Generative models can be used for *data augmentation*, simulation and planning. Training generative models can also enable inference of latent representation that can be useful as general features.

What about using Autoencoders as Generative Models?
The important thing for us in this case is the latent representation that we get from an autoencoder. This latent representation is good at describing images but wouldn't be good at generating images. Given an image I can compress and decompress it, so one option could be to take the decoder network $\mathcal{D}$ and discard the encoder. Variational autoencoders leverage a prior distribution learnt by the autoencoders over z to generate images with an architecture similar to this.
The procedure:

1. Train an autoencoder on S
2. Discard the encoder

3. Draw random vectors to replace the latent representation and feed this to the decoder input



Unfortunately this does not work, you cannot randomly generate an image by just throwing some random numbers inside the latent representation. The problem is that we don't know the distribution of proper latent representation (or it's very difficult to estimate it). The goal is to find out a way to randomly generate images that actually look like a real image, an image with a meaning. This task was solved by **Generative Adversarial Networks**.

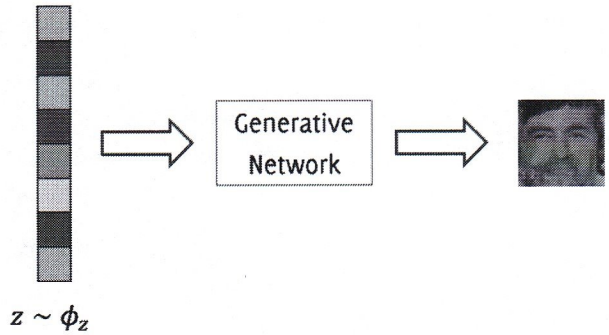
8.3 Generative Models: GANs

We are not looking for an explicit density model ϕ_s describing the "secret" manifold of natural images, we just want to find out a model able to generate samples that look like training samples.

Instead of sampling from ϕ_s, just:

- Sample a seed from a known distribution ϕ_z
- Feed this seed to a learned transformation that generates realistic samples, as if they were drawn from ϕ_s

We will be given some sort of random distribution ϕ_z which is a uniform distribution over a d-dimensional space. From the distribution we sample a seed z, which is a random number with d components and ϕ_z is the distribution used to generate this number.
Now what I would like to do is to train a network $\mathcal{G}$ that takes as input $z \sim \phi_z, z \in \mathbb{R}^d$ and provides me $\mathcal{S}$ which is a "nice" image and of course belongs to $\mathbb{R}^n$, in particular it belongs to the secret manifold of images. In general what I want is taking a random noise, feed it to a generative network and obtain an image like this that's randomly generated:



If you change the noise, you get a different image.

Why is this problem challenging?

What misses here is an *appropriate loss function.* The loss function should tell how "happy" we are with the output image, and being happy means that the image looks like a natural image. Obviously this is hard to measure because you need a number, so the trick of GANs was to replace this loss with another neural network, called **discriminator** $\mathcal{D}$.

The discriminator network $\mathcal{D}$ is trained to determine if the images generated by the generator network $\mathcal{G}$ are real, to classify between real and fake images.

Thus, The GAN solution is to train a pair of neural networks with different tasks that compete in a sort of two player game.

The models are:

- **Generator** $\mathcal{G}$ that produces realistic samples e.g. taking as input some random noise. $\mathcal{G}$ tries to fool the discriminator
- **Discriminator** $\mathcal{D}$ that takes as input an image and assess whether it is real or generated by $\mathcal{G}$

Train the two model and at the end, keep only $\mathcal{G}$

The key idea is that instead of having a measure of how much our image looks like a natural image, you train another network simultaneously which in this context operates like a loss function. After a successful GAN training, the discriminator $\mathcal{D}$ is completely useless because it is no longer able to distinguish between real and fake images and the generator is able to cheat the discriminator, so we throw it away and we take only the generator $\mathcal{G}$.

Let's write this formally. Let's call $\mathcal{G}$ the generator and $\mathcal{D}$ the discriminator.

$$\mathcal{D} = \mathcal{D}(\boldsymbol{s}, \theta_d)$$
$$\mathcal{G} = \mathcal{G}(\mathbf{z}, \theta_g)$$

(**Note** that the role of n and d are reversed: from now on, n is the size of an image and d is the size of the input noise)

These networks will have some parameters θ_d and θ_g, $\boldsymbol{s} \in \mathbb{R}^n$ is an input image (either real or generated by $\mathcal{G}$) and $\boldsymbol{z} \in \mathbb{R}^d$ is some random noise to be fed to the generator network. Our network gives as output:

- The posteriori for an input to be a true image: $\mathcal{D}(\cdot, \theta_d) : \mathbb{R}^n \to [0, 1]$

- The generated image: $\mathcal{G}(\cdot, \theta_g) : \mathbb{R}^d \to \mathbb{R}^n$

A good discriminator is such:

- $\mathcal{D}(\boldsymbol{s}, \theta_d)$ is maximum when $s \in S$
- $1 - \mathcal{D}(\boldsymbol{s}, \theta_d)$ is maximum when s was generated from $\mathcal{G}$
- $1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \theta_g), \theta_d)$ is maximum when $\boldsymbol{z} \sim \phi_z$

When training $\mathcal{D}$ we try to maximize the binary cross-entropy, the classification accuracy, so training $\mathcal{D}$ means solving this classification problem:

$$\max_{\theta_d} \left(\mathrm{E}_{s\sim\phi_S}\left[\log \mathcal{D}(\boldsymbol{s}, \theta_d)\right] + \mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}(G(\mathbf{z}, \theta_g), \theta_d)\right)\right]\right)$$

$\mathrm{E}_{s\sim\phi_S}$ is the expectation, which is replaced in practice by an average over mini-batch, of S drawn from ϕ_S (the distribution of natural images), so this is the expectation over the distribution of natural images. When images are drawn from the manifold you want the first probability part to be large, while you want the probability to be small when images are drawn from the fake distribution (when they are generated by $\mathcal{G}$). The first term is the probability that the discriminator gives to an image that is true, so you want this to as close as possible to 1:

- $\mathrm{E}_{s\sim\phi_S}[\log \mathcal{D}(\boldsymbol{s}, \theta_d)]$ has to be 1 since $s \sim \phi_S$, thus images are real

$\mathrm{E}_{z\sim\phi_Z}$ is the expectation over the fake distribution generated by the generator network. The second term is the probability that you want to be zero because that image is not a natural image.

- $\mathrm{E}_{z\sim\phi_Z}[\log(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}, \phi_g), \theta_d)]$ has to be 0 since $\mathcal{G}(\boldsymbol{z}, \phi_g)$ is a generated (fake) image

You train the neural network to maximize this sum which is the sum of the probabilities for real images and one minus the probability of fake images.

Now let's move to $\mathcal{G}$: a good generator $\mathcal{G}$ is the one which makes $\mathcal{D}$ **to fail.**
A good loss for $\mathcal{G}$ would be something that tells that the discriminator fails because we are trying to trick the discriminator providing fake images that seem real, it's a two player game. We'd like to find the parameter θ_g in order to minimize this:

$$\min_{\theta_g} \max_{\theta_d} \left(\mathrm{E}_{S\sim\phi_S}\left[\log \mathcal{D}(\mathbf{s}, \theta_d)\right] + \mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}(G(\mathbf{z}, \theta_g), \theta_d)\right)\right]\right)$$

The generator should minimize the effectiveness of the discriminator. It's a min-max optimization, you have to perform the optimization simultaneously but typically it is difficult to solve directly, so what you do in practice is alternate:

1. First you do k-steps of SGA (in this case you perform the ascent gradient obviously) maximizing the discriminator function taking $\mathcal{G}$ fixed and optimize for $\mathcal{D}$

$$\max_{\theta_d} \left(\mathrm{E}_{S\sim\phi_S}\left[\log \mathcal{D}(\mathbf{s}, \theta_d)\right] + \mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}(G(\mathbf{z}, \theta_g), \theta_d)\right)\right]\right)$$

2. Then you take 1-step of SGD maximizing the generator function taking $\mathcal{D}$ fixed and optimize for $\mathcal{G}$

$$\min_{\theta_g} \left(\mathrm{E}_{S\sim\phi_S}\left[\log \mathcal{D}\left(\mathbf{s}, \theta_d\right)\right] + \mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}\left(G\left(\mathbf{z}, \theta_g\right), \theta_d\right)\right)\right]\right)$$

we can remove the first term because it does not depend on θ_g:

$$\min_{\theta_g} \left(\mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}\left(G\left(\mathbf{z}, \theta_g\right), \theta_d\right)\right)\right]\right)$$

Nothing changes if we use Gradient Ascent instead of Gradient Descent unless for the change from *min* to *max*:

$$\min_{\theta_g} \left(\mathrm{E}_{z\sim\phi_Z}\left[\log\left(1 - \mathcal{D}\left(G\left(\mathbf{z}, \theta_g\right), \theta_d\right)\right)\right]\right) \longrightarrow \max_{\theta_g} \left(\mathrm{E}_{z\sim\phi_Z}\left[\log\left(\mathcal{D}\left(G\left(\mathbf{z}, \theta_g\right), \theta_d\right)\right)\right]\right)$$

This change does not modify the global solution of the min-max problem, but provides a stronger gradient during the early learning stages.

Moreover, in order to achieve convergence, instead of $\min \nabla_{\theta_g}\left[\sum_i \log\left(1 - \mathcal{D}\left(\mathcal{G}\left(\mathbf{z}_i, \theta_g\right), \theta_d\right)\right)\right]$, it's better to $\max \nabla_{\theta_g}\left[\sum_i \log\left(\mathcal{D}\left(\mathcal{G}\left(\mathbf{z}_i, \theta_g\right), \theta_d\right)\right)\right]$.

Summing up, for k times you draw a minibatch of noise realizations z, you sample a minibatch of images and then you update $\mathcal{D}$ by SDA; then you draw a minibatch only of noise realizations and you just update $\mathcal{G}$ by SDG, which in practice is SGA for what we just said about convergence.

GAN Training

for $i = 1 \ldots$

for k –times

- Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realizations
- Sample a minibatch of images $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m\}$
- Update $\mathcal{D}$ by stochastic gradient ascend:

$$\nabla_{\theta_d}\left[\sum_i \log \mathcal{D}(\boldsymbol{s}_i, \theta_d) + \log\left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d)\right)\right]$$

Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realization

Update $\mathcal{G}$ by stochastic gradient **ascent:**

$$\nabla_{\theta_g}\left[\sum_i \log\left(\mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d)\right)\right]$$

GAN Training

for $i = 1 \ldots$

for k –times

- Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realization
- Sample a minibatch of images $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m\}$
- Update $\mathcal{D}$ by stochastic gradient ascend:

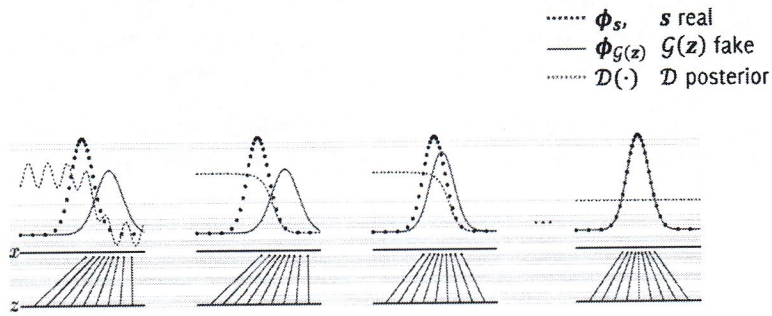
$$\nabla_{\theta_d}\left[\sum_i \log \mathcal{D}(\boldsymbol{s}_i, \theta_d) + \log\left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d)\right)\right]$$

Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realization

Update $\mathcal{G}$ by stochastic gradient descent:

$$\nabla_{\theta_g}\left[\sum_i \log\left(1 - \mathcal{D}(\mathcal{G}(\boldsymbol{z}_i, \theta_g), \theta_d)\right)\right]$$

This illustration shows how the training proceeds:



Again ϕ_s is the real manifold where natural images live, it's what you want to be able to sample, and ϕ_z is the distribution of noise which are feed to the generator to get an image. At the very beginning, the images that you generate are very far from the natural images. When training better $\mathcal{D}$ you are defining better the separation between the two classes, the one generated by the network and the one of real images; but now when you are training $\mathcal{G}$, what you do is steering your generator network to provide images that are more realistic and closer to the distribution of real images. As the training proceeds, the two distribution finally overlap and the discriminator gives 0.5 probability (random probability) because it can no longer distinguish between real and fake. This is **unstable to train** but it's trained by standard tools: backpropagation and dropout. What is important is that the generator does not use directly $\mathcal{S}$ for training, it just takes as input random noise and the real images are just used in order to define some sort of loss through the game with the discriminator.
Generator performance is difficult to assess quantitatively. There is **no explicit** expression for the generator, it is provided in an **implicit** form $\rightarrow$ you cannot compute the likelihood of a sample w.r.t. the learned GAN

9 Recurrent Neural Networks

So far we have considered only *static* datasets: input and output don't have any time, any ordering or dependencies between each other.
(Aggiungere figura sequencernn)
The main idea is to think the input as a vector X_i and consider a sequence of inputs in order to model that sequence of samples.

Some relevant sequence are, for instance, time-series or text that, in general, are some sort of *dynamical* data. To model this sequence you have different approaches:

- **Memoryless** models: you don't explictly store a representation of the past, but you just use the previous k-samples to predict the next one
  - *Autoregressive models*: predict the next input from previous ones using "delay taps"
  - *Feed Forward Neural Networks*: generalize autoregressive models using non-linear hidden layers. You feed your network with the k past samples and you try to predict the next one. This approach is known as *Sliding Window*
- **Memory-based** models: they are generative models with a real-valued hidden state which cannot be observed directly. The hidden state has some dynamics possibly affected by noise and produces the output. To compute the output the model has to infer hidden state. Input as treated as driving inputs.
  - *Linear Dynamical System*: keep the state of the system which models the evolution of your data and define an output function which models the output given the current state of the system. In Linear Dynamical Systems the system becomes state continuous with Gaussian uncertaintly, transformations are assumed to be linear and the state can be estimated using *Kalman Filtering*. One observation is that if you model your data in this way, you get a **stochastic** system.
  - *Hidden Markov models*: you have a random variable which models the state of the system and another one that models the output of the system. The state is assumed to be discrete, state transitions are *stochastic* (transition matrix). The output is a *stochastic* function of hidden states. The states can be estimated via *Viterbi algorithm*. In this model you do not have any input, but only a hidden state. This is a **stochastic** system
  - *Recurrent Neural Network*: they are **deterministic** systems and they will be deepened more in the following chapters.

9.1 Recurrent Neural Networks

Let's start modeling a standard system with a standard Feedforward Neural Network such as the one seen so far: the output is a non linear function of the weighted sum of the hidden neurons and these hidden neurons are a non linear function of the weighted sum of the input.
The classical way to add the memory is through *Recurrent Hidden Neurons*, the blue ones in the Fig.7: a set of hidden neurons is added and their outputs are recurred. These

hidden neurons have a feedback loop which makes their output depending on the previous value.
Memory via recurrent connections has some benefits:

- Distributed hidden state allows to store a information efficiently
- Non-linear dynamics allows complex hidden state updates

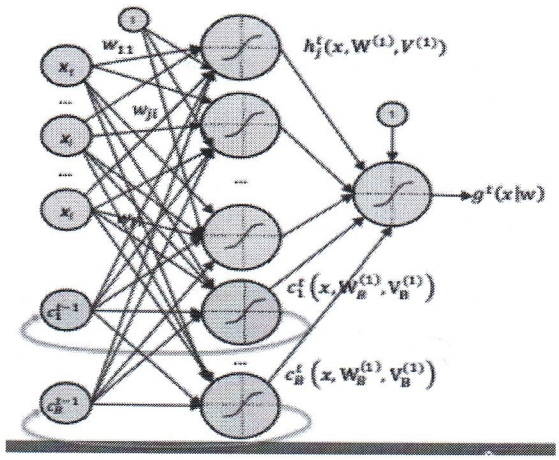


Figure 7: Recurrent Neural Network

These approach is due to Helmann and it is different from linear dynamical systems since one part of the hidden layer [the brown part] is *time-less* and the other part is in charge of bringing forward the information. The blue part is a sort of summary of all the input that the network have seen so far. This summary is brought forward to the next step and it becomes non-linear combination of the previous state and the current input.
The blue nodes are also called *c-nodes* because they provide the context to create the output.

The context part of the network depends on the previous value of the context network and the current input which update the state: in fact, the *context-neuron* is :

$$c_{\mathrm{B}}^{\mathrm{t}}\left(x, \mathrm{W}_{\mathrm{B}}^{(1)}, \mathrm{V}_{\mathrm{B}}^{(1)}\right)$$

where x is the input, W_B are the weights from the input to the context network and V_B are the weights from the context network to itself, so from the previous step to the current step. In the figure there are other weights from c_i^{t-1} to the brown neurons (from the context to the hidden) that can be called V.
In general, the main formula of the RNN are shown below

$$g^t(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot)\right)$$

$$h_j^t(\cdot) = h_j^t\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n} + \sum_{b=0}^{B} v_{jb}^{(1)} \cdot c_b^{t-1}\right)$$

$$c_b^t(\cdot) = c_b^t\left(\sum_{i=0}^{I} v_{bi}^{(1)} \cdot x_{i,n} + \sum_{b'=0}^{B} v_{bb'}^{(1)} \cdot c_{b'}^{t-1}\right)$$

"With enough neurons and time, RNNs can compute anything that can be computed by a computer

In 1990 there was a theorem that proves that if you have enough neurons and time, you can prove that this model is Turing-complete, as powerful as Turing machine.
This recurrent loop needs to be estimate: back-propagation needs that all the function to be differentiable and the network to be feed-forward, but this architecture is not feed-forward anymore. So we have to modify back-propagation or find different algorithm to discover this loop.

If you want to consider the output at time t:

$$g^t(x_n|w) = g\left(\sum_{j=0}^{l} w_{1j}^{(2)} \cdot h_j^t(\cdot) + \sum_{b=0}^{B} v_{1b}^{(2)} \cdot c_b^t(\cdot)\right)$$

it is divided in two part, the forward and the recurrent respectively. We assume B hidden neuron in the context part, J are the neuron in the hidden state.
The context network is combination of previous values of the context network and the weighted sum of the input.

Instead, the **context update** is a non-linear function c_b^t of the weighted sum of the previous context plus the linear combination of the current input.
We have more weights but not necessary in form of numbers but as a sort of weights, in the sense that the weights from the previous values of the memory state to the current value of the memory state are tricky to be defined because they represent the memory: these weights are the feedback loop.

But how to build the context part of the network?
By definition, the previous layer of the hidden state has the same number of memory states because they are the same thing: basically, you have B hidden neurons here in the context network [we are referring on the blue neurons on the right in Fig.7] and B inputs which are the previous values of those hidden neurons, so the number of weights is $B * B$ and this is why you see here

$$c_b^t(\cdot) = c_b^t\left(\sum_{i=0}^{I} v_{bi}^{(1)} \cdot x_{i,n} + \sum_{b'=0}^{B} v_{bb'}^{(1)} \cdot c_{b'}^{t-1}\right)$$

b' which goes from 0 to B of the weights from b' to B.
This is a relevant part because we want to learn these weights via back-propagation, but while it's a reasonable standard back-propagation in the brown part, it's more tricky to have back-propagation in the blue part.

The trick is called **Back-propagation Through Time** and it is based on the following idea:
if the input V_B^t is the output of the previous neuron, I can replicate this structure back in time and I can use the input V_B^{t-1} to the previous neuron. So if the output at time t is $c_B^t\left(x, W_B^{(t)}, V_B^{(t)}\right)$, i.e. a combination of its previous value and the current input, I can replicate this node back. So you can back-propagate through these weights. Obviously to get the output you have to back-propagate again and again and again.

The idea is to make multiple copies of the context network and associate weights to them, so you can back-propagate as a standard Feedforward Neural Network: it's standard because your dataset is finite so you do not need an infinitely long training sequence;

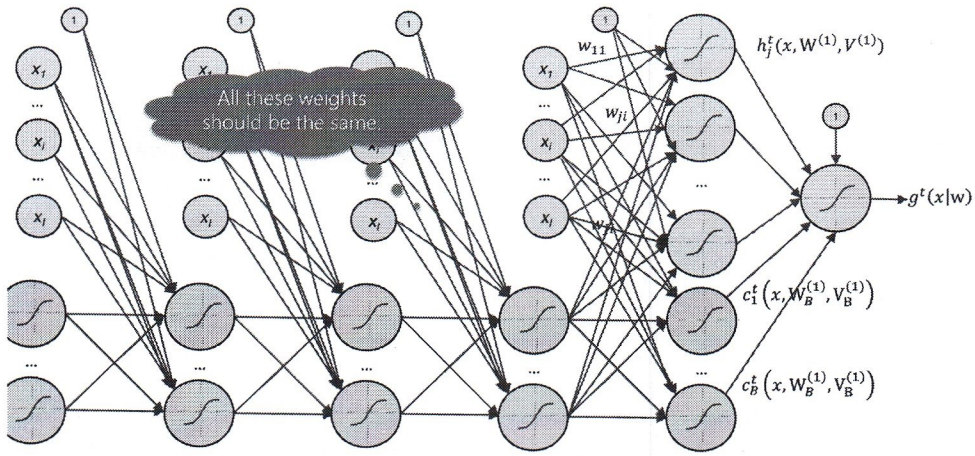


Figure 8: Backpropagation Through Time

if your training sequence has an hundred samples, you unroll for an hundred steps and you can back-propagate on that.
The problem is that this approach is not really sustainable for very long sequences, so you have to unroll until a certain point: this approach is called **Truncated Back-propagation**. This may look like limiting because you do not show, to the network, sequences longer than the limit imposed, but in practice it is not a limitation.
One approach instead of doing full back-propagation can be to slice your dataset in chunks and unroll for the length of the chunks. So the idea is that you start from your loop and you unroll, then you get in principle a feedforward neural network.

Another important observation is that all the weights of the unrolled part must be the same through all the layers because if you run back-propagation in one neuron (such as the most right blue one), the gradient will be "the error x the derivatives x these weights x derivatives x the output of this neuron"; instead the gradient on the most left neuron is "derivative of the error x derivative of this x weights x derivative x weights x ... and so on (starting from the right to the left) x the output of this neuron". The two gradients are different because are computed via different formula.
The point is that if you unroll and you treat this "fake" feedforward neural network as classic feedforward neural network, you will end up with an inconsistency because all the weights should be equals $\rightarrow$ you have to implement back-propagation by forcing all these replicas to have the same value.
In practice, instead, works as follows:

- Perform network unroll for U steps
- Initialize V, V_B matrices (all the replicas) to be the same. The trick is here, to keep them equal, initialize them equal.
- Compute gradients, but you update using the average gradient of all the copies, so you update replicas with the average of their gradients

$$V = V - \eta \cdot \frac{1}{U} \sum_{0}^{U-1} \frac{\partial E}{\partial V^{t-u}} \quad V_B = V_B - \eta \cdot \frac{1}{U} \sum_{0}^{U-1} \frac{\partial E}{\partial V_B^{t-u}}$$

There are two ways to perform this pipeline:

1. Unroll → Compute Gradient → Update all the weights independently → Compute the Average Gradient
2. Unroll → Compute Gradient → Update all the weights with the Average Gradient

With this trick we are forcing all the replicas to have the same weights. It's a sort of weights sharing.

How do you initialize this?

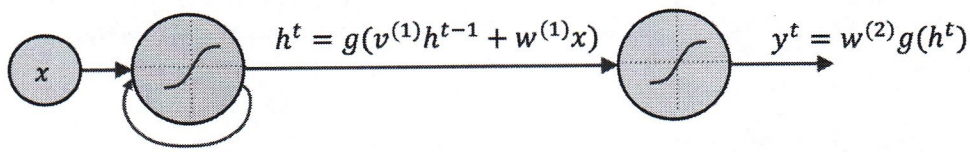
- First approach: initialize to 0
- Second approach: learn how to initialize. Take the most back-in-time [blue] node as the first value of the state, so you can learn what is its value: instead of differentiating with respect to this weight, you differentiate w.r.t. that node. By doing this you train the sequence to learn what should be the initial state that works best.

How long we can go back in time?
Sometimes output might be related to some input happened quite long before. However, back-propagation through time was not able to train recurrent neural networks significantly back in time because it is not able to back-propagate through many layers.

It is a problem mainly due to text analysis. They find out that there is no way to store more than 10-20 steps or in other term there is no way to learn with network unrolling a window longer than few dozen of terms.

The reason is due to **Vanishing Gradient**: if you think about network unrolling for a very long series of time, what happens is that the gradient will be the product of the derivatives of all the intermediate functions and will get to zero.
Usually the initialization of the network is random and there is no way to move the memory of the initial steps (the blue top left neurons) anywhere with respect to the initial initialization. So the memory of this model is random since it is not possible to use inputs from the past.

To better understand why it is not working, consider a simplified case:



The model represented is a simplified model. Considering $v^{(1)}$, it is how much the output of this neuron is weighted to compute the next one. The output of the hidden neuron is a non-linear function g() depending on the weighted sum of the previous loop $v * h^{t-1}$ plus a weighted input. Focusing on the first part of the network, assume you want to derive the error w.r.t. input weight and you want to minimize the sum over all the sequences of the derivative of the error at time t w.r.t. to the weight of the input.

This weight appears in the input but also in the previous inputs. So this derivative is not so easy, but we can use the trick of the chain rule of derivatives:

$$\frac{\partial E}{\partial w} = \sum_{t=1}^{S} \left(\frac{\partial E^t}{\partial w} \right) \rightarrow \frac{\partial E^t}{\partial w} = \sum_{t=1}^{t} \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial h^k}{\partial w}$$

So basically, depending on how much is k, you derive w.r.t. to one of the past loops. If you look at the second formula, you notice that the derivative of the hidden neuron output w.r.t. any of the output in the past, is

$$\frac{\partial h^t}{\partial h^k} = \prod_{t=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} v^{(1)} g' \left(h^{i-1} \right)$$

i.e. the product of the derivatives from k to t of the derivative of time t over $t-1$: Note that the derivative of $g(v() * h)$ is $g'() * v^{(1)}()$. Considering the norm

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| = \left\| v^{(1)} \right\| \left\| g' \left(h^{i-1} \right) \right\| \Rightarrow \left\| \frac{\partial h^t}{\partial h^k} \right\| \leq \left(\gamma_v \cdot \gamma_{g'} \right)^{t-k}$$

It means that the the norm is less or equal than the product of two terms: γ_v is given by the weights, the other $\gamma_{g'}$ is given by the derivatives, and the exponent is how long is the past.
So if you go one step back, you see that this is the v*derivative, if you go 2 step back you have the product v derivatives * v*derivative: depending of the value of this term, you have a sequence of products which might go either to infinite or zerO.
In particular, if

$$\rho = \left\| \frac{\partial h^t}{\partial h^k} \right\| < 1$$

it converges to zero. If the derivative is less than 1, the gradient of the error function w.r.t. to an input $t-k$ steps back in the past goes to 0.
The point is that if you try to update the weight of an input k epochs ago, your gradient is null. If you use for this weight either a *tanh* or a *sigmoid*, by definition $\gamma_{g'}$ is less than 1. We have seen that the derivative of sigmoid and tanh, are respectively, 0.5 or 0.25: it means that in few steps in the past, the derivative of the output w.r.t. to the input will be $0 \rightarrow$ With Sigmoids and Tanh we have Vanishing Gradient.
You may think that the other number γ_v is going to compensate, but it means the more you go in the past, the more it should be bigger, but it make no sense: this value γ_v is a constant. If it is a big constant such that the product is bigger than 1, it worst because you have that $\rho > 1$ so the gradient will explode numerically.

The only solution is to force all gradient to be either 0 or 1, otherwise vanishing gradient or gradient explosion occur. The only way to perform is using **ReLu**:

$$g(a) = ReLu(a) = max(0, a) \Rightarrow g'^{(a)} = 1_{a>0}$$

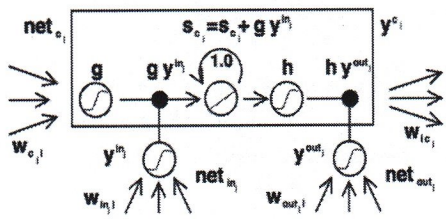
This solve part of the story, no more vanishing gradient, but you have still the risk of exploding gradient.
If we force this loop to have weight = 1, we guarantee nor exploding or vanishing gradient because the derivative w.r.t. to the previous step will be always 1.

It is constraining the model by solving the vanishing gradient and the exploding gradient. By doing this you make the output dependent on whatever time you want. It is like a shortcut connection with all timestamps in the past.

If you look closer, if the weight $v^{(1)}$ is $= 1$, you do not need to train the model; now this is a pure Feedforward Neural Network with all previous steps. This weight has not to be trained, for all those replicas you know already the weights, they are all 1, so no training is require for this loop.
This kind of object is called **Constant Error Carousel**.
By doing this the output depends on all the history of the input. The problem of the network is that it can *only accumulate the input*, it just computes the integral of the input.
This topology is useless because it is only integrating the input.

9.1.1 Long-Short Term Memories

Hochreiter & Schmidhuber (1997) solved the problem of vanishing gradient designing a memory cell using logistic and linear units with multiplicative interactions.



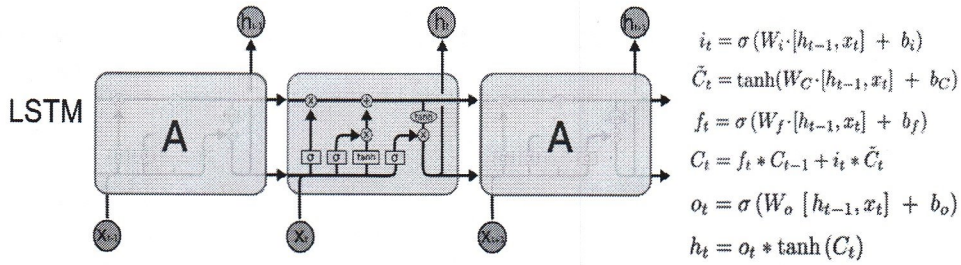
The creators started from the idea of Constant Error Carousel (CEC) and they thinks that now that we have an accumulator of the input, the only things to do is to build a network to learn when it is worth to add an input to the memory and when it is worth to read an input from the memory.

The idea is to have some of *gate* mechanism where:

- Information gets into the cell whenever its *"write"* gate is on.
- The information stays in the cell so long as its *"keep"* gate is on.
- Information is read from the cell by turning on its *"read"* gate.

The gates are based on Constant Error Carousel. You can back-propagate through this since the loop has fixed weight.

Decision (e.g. as write/not-write) are based on the output of a Sigmoid; instead, the loop is ruled by a ReLu activation function. This kind of cell is the first **Long-Short Term Memories** idea.
The current architecture of the cell is shown below:



LSTM

Different gates are highlighted:

- Input gate: the input is computed by the *tanh* path and it is multiplied by this *sigmoidal* part. It states if it is worth or not to write in the memory
- Forget gate: in charge of re-writing or erasing the content of the memory. The memory is multiplied by a *sigmoid*, which indicates to keep the input if the sigmoid is equal to 1, otherwise erase the memory before adding the input.
- Memory gate: decide on adding the input to the previous state of your network. The memory is the sum of the forget gate and the input gate.
- Output gate: it is the combination of tanh and the reading mechanism controlled by a sigmoid.

With LSTM networks you can build a computation graph with continuous transformation: the idea is that you can build an internal representation of the history of the input. This internal representation is updated at each input.
We can build different configurations of LSTM:

- One-to-One: translate one to one, one input and you want to translate it in another. For each input you predict one output
- One-to-many: one input and you want to predict many output. If x is the input, the idea is that the input modifies the state of the network and what happens is that you output one value, then another one, then another one. This kind of approach is used in Captioning: you enter an input (an image), you start with the first word, the you feed the model with this first word and you output the second word and so on; you write textual description for a single input.
- Many-to-one: sequence of input and you want one output: example is Text Classification.
- Many-to-many: it is in Video Classification or Video Captioning.
- Sequence-to-Sequence: you enter a sequence, then you have a state and the machine is started with this state. Then you generate a sequence. You think this kind of network as a sequence encoder. Since this model can be used also for generation, you have an input sequence and you use the state of this machine to generate a new sequence. This is the way most of text prediction and question answer are trained: you have a sequence encoder; once encoded this hidden representation for the sentence, you use a decoder (which is the same or a different LSTM) which is trained to output sequence conditioned on the value of this state.

9.1.2 Gated Recurrent Unit

An evolution of this memory is the **Gated Recurrent Unit**. It is more or less the same of LSTM: it combines the forget and input gates into a single "update gate". It also merges the cell state and hidden state, and makes some other changes.

9.2 Tips and Tricks

Sometimes you can use **bidirectional LSTM**: if you have for instance many-to-one approach, you can read the sequence from left to right and vice-versa, put the two encoding together and classify the output with it. This is used often if you have to predict the output only having seen the entire sequence: you can process the sequence in the two direction. You cannot do it always, if you predict online, you cannot use it.

Another trick is in weight initialization:

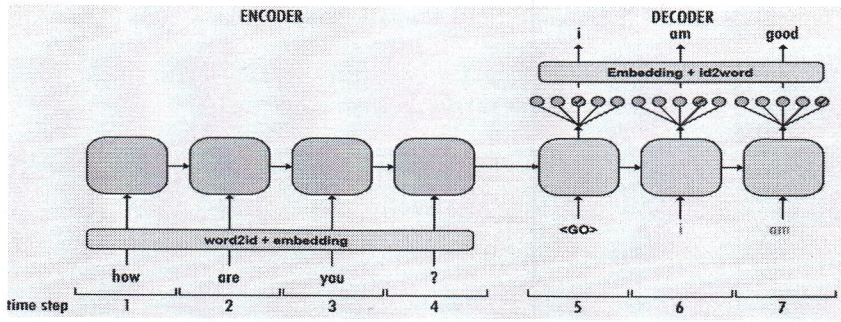
- Could initialize them to a fixed value
- Better to treat the initial state as learned parameters:
  1. Start off with random guesses of the initial state values
  2. Backpropagate the prediction error through time all the way to the initial state values and compute the gradient of the error with respect to these
  3. Update these parameters by gradient descent

10 Sequence2Sequence

Some examples based on the input/output sequence:

- One-to-many - *Image Captioning*: input a single image and get a series or sequence of words as output which describe it. The image has a fixed size, but the output has variable length
- Many-to-one - *Sentiment Classification/Analysis*: input a sequence of characters or words, e.g., a tweet, and classify the sequence into positive or negative sentiment. Input has variable lengths, output is of a fixed type and size. You accumulate the inputs until the last LSTM, which states an output.
- Many-to-many - *Language Translation*: having some text in a particular language, e.g., English, we wish to translate it in another, e.g., French. Each language has it's own semantics and it has variable lengths for the same sentence. Can be done in two ways: accumulating the inputs through hidden states or answering directly. The second looks like more natural, but it has some drawbacks: you need that translation can be treat as an aligned process, it works if the input and the outputs are synchronized. Most of Sequence2Sequence model works like the first one.

The Seq2Seq model follows the classical encoder decoder architecture: at training time the decoder **does not** feed the output of each time step to the next; the input to the decoder time steps are the target from the training. At inference time the decoder feeds the output of each time step as an input to the next one.



At training time, you write the input with no target [the green part]. Only when you input the $< GO >$, you start with the target. Then in order to predict the second target, you feed your network with the previous target.
Once the model is trained, you give the input, which ends with $< GO >$ and the model starts to predict the most likely output. As before, once the output is computed, in order to predict the next output, the previous one is given as input. When the network see $< eos >$ is terminates.
Each sequence can have different lenght, but you can enforce a maximum length for the output.
Word Embedding means represent the same word in less sparse representation. The yellow part (the one in the bottom) transforms words into vector , the other yellow part (the one in the top) reverse the process. Because of the embedding, the output is the selection among the possible words that it should say as output: usually it is a softmax output among all possible output words.

Special characters:

- $< PAD >$: During training, examples are fed to the network in batches. The inputs in these batches need to be the same width. This is used to pad shorter inputs to the same width of the batch
- $< EOS >$: Needed for batching on the decoder side. It tells the decoder where a sentence ends, and it allows the decoder to indicate the same thing in its outputs as well.
- $< UNK >$: On real data, it can vastly improve the resource efficiency to ignore words that do not show up often enough in your vocabulary by replace those with this character.
- $< SOS >/< GO >$: This is the input to the first time step of the decoder to let the decoder know when to start generating output.

Sometimes the first best results is not the real best. Usually if you take the second best, the sequence goes better. So you can do what is called *Beam Search* (?): take the first 3 results, then you run three parallel executions. Considering the results of these execution, you choose the best one according to some criteria.

Dataset Batch Preparation:

1. Sample *batch_size* pairs of (*source_sequence*, *target_sequence*).
2. Append $< EOS >$ to the *source_sequence*
3. Prepend $< SOS >$ to the *target_sequence* to obtain the *target_input_sequence* and append $< EOS >$ to obtain *target_output_sequence*.
4. Pad up to the *max_input_length* (*max_target_length*) within the batch using the $< PAD >$ token.
5. Encode tokens based of vocabulary (or embedding)
6. Replace out of vocabulary (OOV) tokens with $< UNK >$. Compute the length of each input and target sequence in the batch.

In order to compute the error, you have to compute the probability of one character given the state v and the outputs. Given $< S, T >$ pairs, read S, and output T' that matches T:

$$p\left(y_1, \ldots, y_{T'} | x_1, \ldots, x_T\right) = \prod_{t=1}^{T'} p\left(y_t | v, y_1, \ldots, y_{t-1}\right)$$

$$1/|\mathcal{S}| \sum_{(T,S)\in\mathcal{S}} \log p(T|S)$$

So you can train this model with Cross-Entropy. You can image that each of these networks are softmax over a specific word; you sum the cross-entropy over the sequences and you minimize by the network estimates. So backpropagate on the cross entropy, trying to get the exact sequence.

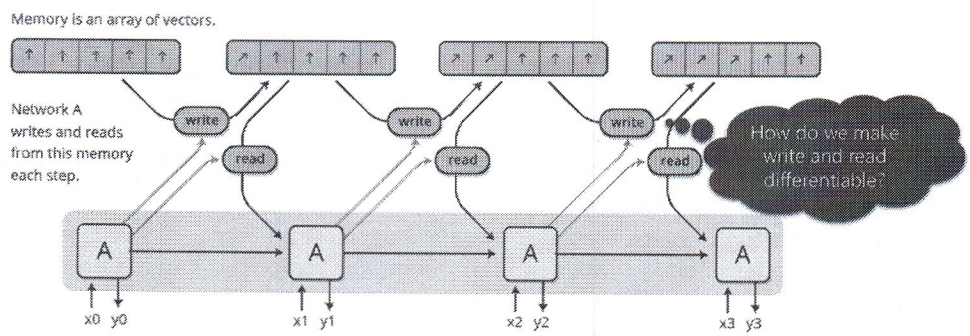
In general you can stack multiple layer of LSTM: you might have a sort of hierarchical representation. You can perform this embedding hierarchy but through time and then you can have some non-linear transformation (ReLU) at the top. You can also have shortcut connections.

10.0.0.1 Bidirectional LSTM You can have two level, one goes forward, the other backward through the LSTMs and then you concatenate the LSTMs and you predict the output. The point is: let's assume you want to predict for each word in the sentence if that word is used in positive or negative way in the sentence. If we do it in one direction only, the output depends only on the previous ones. But if the mean of that word change through the sentence, for each word in might be useful what will be after.

The Sequence2Sequence idea has possible drawbacks: you are compressing long sequences into a vector. It works, backpropagation does not suffer the vanishing gradient, but you can have problem in processing long sequences. The way Seq2Seq is done, the way embedding of sequence into a state and decoding this embedding in another sequence is done, has still some limitation into how much information and how much knowledge or context you can embed into this vector.
After a few tests on LSTM people tried to add memory to recurrent neural Networks: it can be achieved in different ways, the most interesting one is the so called **Neural Turing machine**.

10.1 Neural Turing machine

If you remember, in Turing machine you have a infinite string of symbols and you can perform any computation by a series of operations that reads and writes in a sequence from this string. We can do it with the neural network in the sense that what this recurrent model does is basically storing information in the memory, reading from it and writing in it. It uses the memory explicitly instead of an encoding of it.
So **Neural Turing Machines** combine a RNN with an external memory bank.



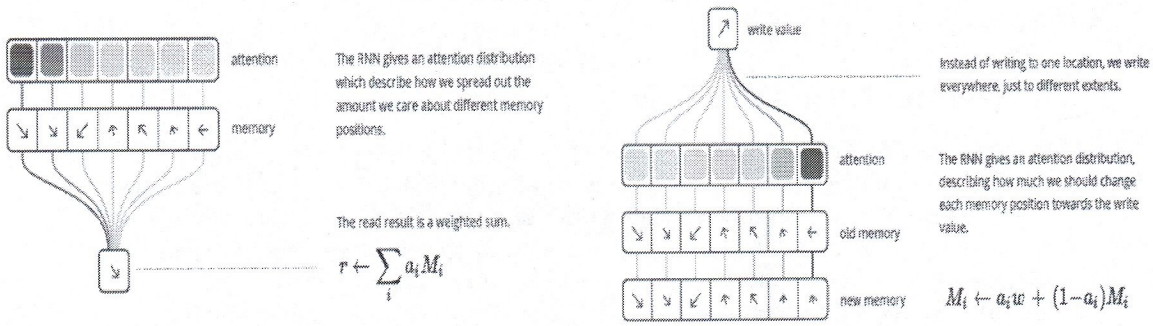
I'm using an NTM mostly because the trick used in NTM to read and write for the memory is probably the most interesting part and it's the same trick upon which we base the attention mechanism.
In the figure, you have a network which can write or rewrite the memory given some input; you decide to modify the first cell to store this input and then you read the next

step; you get another input and you decide to write the second based on the state of the memory; the same for the third and fourth and so on and so forth.
In general, especially when you have long sequences, you would like to partition your memory in states and you would like to use this memory to do things.
The problem is that *read* and *write* are *not differentiable* operations.
Think about read: reading means you select one address and extract the cell and only the cell at a specific address; in the same way, writing means you select one address and you change the content of that address.
So while writing and reading per se are easy to differentiate, *addressing* is the problem. We would like to learn where to write. Memory addresses are discrete by definition so you should be able to derive the write and read operation with respect to the address. This seems a very complex issue which has a very simple solution: *write always everywhere.*

Neural Turing Machine challenge:

- We want to learn what to write/read but also where to write it
- Memory addresses are be fundamentally discrete
- Write/read differentiable w.r.t the location we read from or write to

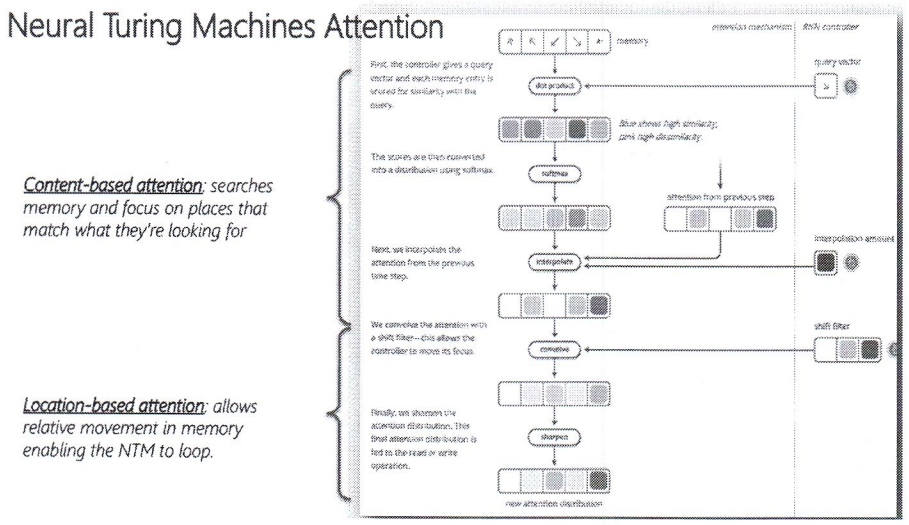
The solution is: *every step, read and write everywhere, just to different extents* [Attention mechanism].



How much you decide to write depends on the input. In each step, I write everything but mostly in some cells, so it means that based on my input I will write in a cell and not in another one. It's a sort of *associative writing* but in principle at each step I could write everywhere.
Thinking about LSTM gates, the idea is the same: in each cell you decide where and what to write; here the gate is over the entire memory and you decide on all the cells to which cell to write or not to write.
So the idea is to have an **Attention mechanism**: basically your data will tell you where to write, in order to put your attention on specific cells.
In the left part of the Figure, you have your memory and a 'read': you want to read from a particular cell, thus you put attention there.
Two types of attention:

- **Content-based attention**: searches memory and focus on places that match what they're looking for

- **Location-based attention**: allows relative movement in memory enabling the NTM to loop



Think about having a softmax output:

Basically, given a certain input you can select with this softmax which cell you're gonna read; what happens is that the reading would be the weighted average where the weights are the softmax weights of your entire reading.

Because the softmax is very selective (it's a normalised exponential function - it's almost everywhere equal to 0) you would read mostly from one or two or three cells and the interesting part is that you can derive this because it is nothing but the product between the memory, which is nothing more than a set of vectors, and the attention, which is a softmax and it is differentiable.

Each item is thus weighted with the query response to produce a score.

Based on the input, you have a softmax on the memory and the output gets changed based on this memory. The softmax is driven by the input, so it's a content-based attention, and once you have retrieved the most similar area in the memory, you have a sort of location-based mechanism and this location mechanism allows you to move from one area to another.

What is interesting and relevant of NTM is the idea that you can **focus on a PART** of your memory: this is **Attention**!

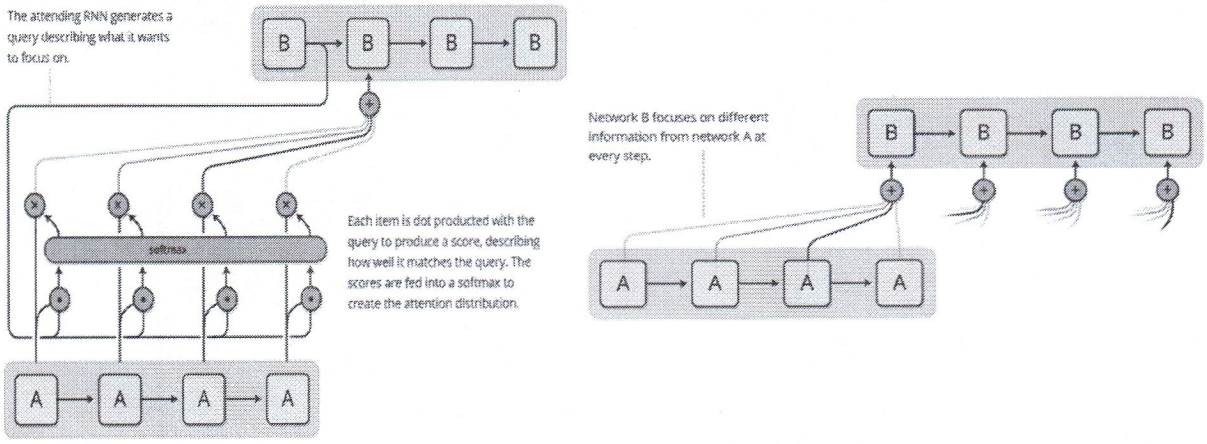
10.1.1 Attention Mechanism in Seq2Seq Models

So people realized that the attention mechanism could have been an interesting add-on for the Sequence to sequence model I was talking about before.

Considering the sequential dataset $\{((x_1, \ldots, x_n), (y_1, \ldots, y_m))\}_{i=1}^{N}$, the decoder role is to model the generative probability: $P(y_1, .., y_m|x)$.

In "vanilla" seq2seq models, the decoder is conditioned initializing the initial state with

last state of the encoder. That works well for short and medium-length sentences; however, for long sentences, becomes a bottleneck.
Let's use the same idea of Neural Turing Machines to get a differentiable attention and learn where to focus attention.



Attention distribution is usually generated with content-based attention.
Each item is thus weighted with the query response to produce a score. Scores are fed into a softmax to create the attention distribution

Attention function maps query and set of key-value pairs to an output.
Output computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function:

1. Compare current target hidden state $\boldsymbol{h_t}$ with source states $\boldsymbol{h_s}$ to derive attention

$$\text{score}\left(\boldsymbol{h}_t, \overline{\boldsymbol{h}}_s\right) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \overline{\boldsymbol{h}}_s \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_1 \boldsymbol{h}_t + \boldsymbol{W}_2 \overline{\boldsymbol{h}}_s\right) \end{cases}$$

2. Apply the softmax function on the attention scores and compute the attention weights, one for each encoder token

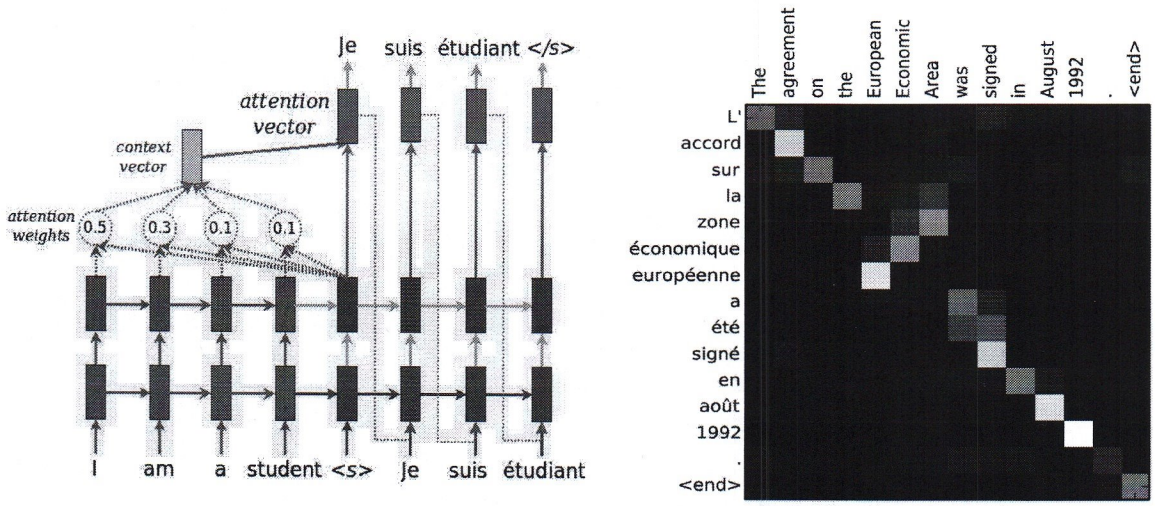
$$\alpha_{ts} = \frac{\exp\left(\text{score}\left(\boldsymbol{h}_t, \overline{\boldsymbol{h}}_s\right)\right)}{\sum_{s'=1}^{S} \exp\left(\text{score}\left(\boldsymbol{h}_t, \overline{\boldsymbol{h}}_{s'}\right)\right)}$$

3. Compute the context vector as the weighted average of the source states

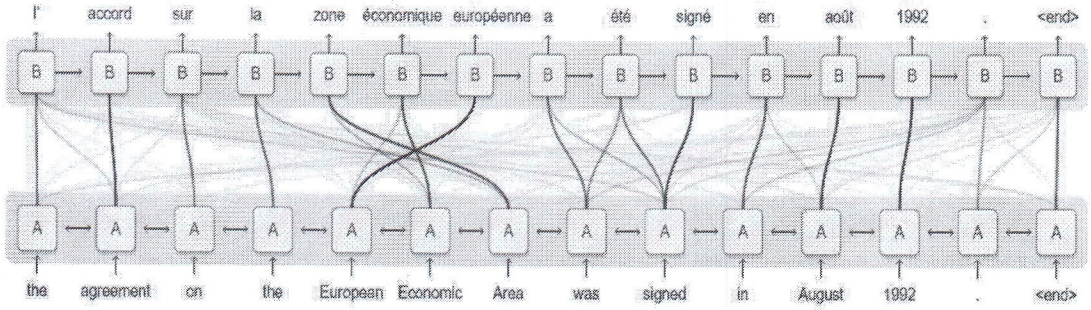
$$\boldsymbol{c_t} = \sum_S \alpha_{ts} \overline{\boldsymbol{h}}_{\boldsymbol{s}}$$

4. Combine the context vector with current target hidden state to yield the final attention vector

$$\boldsymbol{a}_t = f\left(\boldsymbol{c}_t, \boldsymbol{h}_t\right) = \tanh\left(\boldsymbol{W}_c \left[\boldsymbol{c}_t; \boldsymbol{h}_t\right]\right)$$



The Figure on the right represents the **Alignment matrix**, used to visualize attention weights between *source* and *target* sentences.
For each decoding step, i.e., each generated target token, describes which are the source tokens that are more present in the weighted sum that conditioned the decoding.
We can see attention as a tool in the network's bag that, while decoding, allows it to pay attention on different parts of the source sentence.



Attention allows processing the input to pass along information about each word it sees, and then for generating the output to focus on words. Attention can be used in different scopes:

- Translation
- Voice Recognition: attention allows one RNN to process the audio and then have another RNN skim over it, focusing on relevant parts as it generates a transcript.
- Image Captioning: a CNN processes the image, extracting high-level features. Then an RNN runs, generating a description of the image based on the features. As it generates each word in the description, the RNN focuses on the CNN interpretation of the relevant parts of the image.

10.1.2 Attention in Respose Generation - Chatbots

Chatbots can be defined along at least two dimensions:

- **Core Algorithm**:
  - *Generative*: encode the question into a context vector and generate the answer word by word using conditioned probability distribution over answer's vocabulary. E.g., an encoder-decoder model.
  - *Retrieval*: rely on knowledge base of question-answer pairs. When a new question comes in, inference phase encodes it in a context vector and by using similarity measure retrieves the top-k neighbor knowledge base items.
- **Context Handling**:
  - *Single-turn*: build the input vector by considering the incoming question. They may lose important information about the history of the conversation and generate irrelevant responses
    $$(q_i, a_i)$$
  - *Multi-turn*: the input vector is built by considering a multi-turn conversational context, containing also incoming question
    $$\{([q_{i-2}; a_{i-2}; q_{i-1}; a_{i-1}; q_i], a_i)\}$$

Vinyals and Le, 2015 and Shang et al., 2015 proposed to directly apply sequence to sequence models to the conversation between two agents:

1. The first person utters "ABC"
2. The second person replies "WXYZ"

Generative chatbots use an RNN and train it to map "ABC" to "WXYZ": we can borrow the model from machine translation; a flat model simple and general; Attention mechanisms apply as usual.

Generative Hierarchical Chatbots The idea could be concatenating multiple turns into a single long input sequence, but this probably results in poor performances.
LSTM cells often fail to catch the long term dependencies within input sequences that are longer than 100 tokens.
No explicit representation of turns can be exploited by the attention mechanism.

Xing et al., in 2017, extended attention mechanism from single-turn response generation to a hierarchical attention mechanism:

- Hierarchical attention networks (e.g., characters → words → sentences)
- Generate hidden representation of a sequence from contextualized words

(Hierarchical Generative Multi-turn Chatbots - Hierarchical Document Classification guardare le slide)

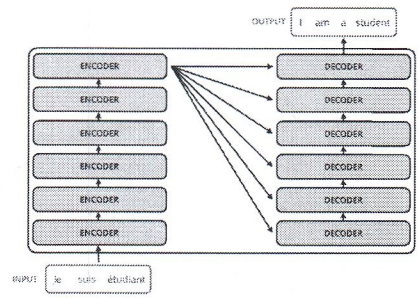
Attention is all you need Having seen attention is what makes things working you start wondering:

- Sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.
- Attention mechanisms have become an integral part of compelling sequence modeling and transduction models in various tasks. Can we base solely on attention mechanisms, dispensing with recurrence and convolutions entirely?
- Without recurrence, nor convolution, in order for the model to make use of the order of the sequence, we must **inject** some information about the relative or absolute position of the tokens in the sequence.

There has been a running joke in the NLP community that an **LSTM with attention** will yield *state-of-the-art* performance on any task. Attention is built upon RNN, .. The **Transformer** breaks this assumption!

10.1.3 Transformer

In a Transformer model we can distinguish an encoding component, a decoding component, and connections between them.
The Encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:



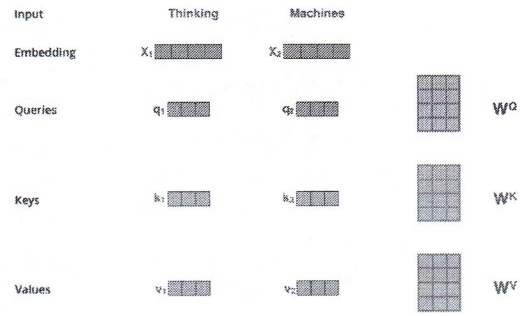
1. *Self-Attention*: a layer that helps the encoder look at other words in the input sentence as it encodes a specific word
2. *Feed Forward Neural Network*: The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position [**Position-wise Feed-Forward NN**]

The Decoder has the following layer:

1. *Self-Attention*: as before
2. *Encoder-Decoder Attention*: helps the decoder focus on relevant parts of the input sentence
3. *Feed Forward Neural Network*: as before

We begin by turning each input word into a vector using an embedding algorithm. The embedding only happens in the bottom-most encoder.
After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder. Here we begin to see one key property of the Transformer, which is that the word in each position flows through its own path in the encoder. There are dependencies between these paths in the self-attention layer. The feed-forward layer does not have those dependencies. So the self-attention layer, in some way, stores the dependencies of inputs, while the Feed Forward is treated as 'independent' for each input.

Scaled Dot-Product Attention



The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors: a *Query vector*, a *Key vector*, and a *Value vector* [they are abstraction that are useful for the calculus]. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.
Multiplying x_1 by the W^Q weight matrix produces q_1, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

The **second step** in calculating self-attention is to calculate a score: we need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position. The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring: so if we're processing the self-attention for the word in position #1, the first score would be the dot product of q_1 and k_1. The second score would be the dot product of q_1 and k_2.

The **third** and **forth steps** are to divide the scores by the square root of the dimension of the key vectors $\sqrt{d_k}$: this leads to having more stable gradients.
Then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

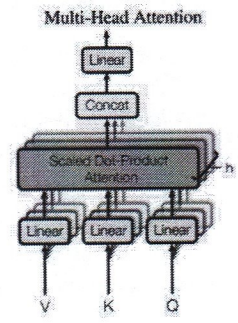
The **fifth step** is to multiply each value vector by the softmax score: the intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example)

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

At the end the *Attention* can be expressed as

$$Z = \text{ Attention } (Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Multi-head Attention If we do the same self-attention calculation we outlined above, just h different times with different weight matrices, we end up with h different Z matrices.
The feed-forward layer is not expecting h matrices – it's expecting a single matrix (a vector for each word). So we concat the matrices then multiple them by an additional weights matrix W^0 (which was trained jointly with the model).
The result would be the Z matrix that captures information from all the attention heads: this matrix is sent to the



FNN.

Positional Encoding One thing that's missing from the model as we have described it so far is a way to account for the order of the words in the input sequence. To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine **the position** of each word, or the distance between different words in the sequence. The intuition here is that adding these values to the embeddings provides meaningful distances between the embedding vectors once they're projected into Q/K/V vectors and during dot-product attention.

To sum up, a Transformer model is made out of:

- Scaled Dot-Product Attention
- Multi-head Attention
- Position-wise Feed-Forward Networks
- Embeddings and Softmax
- Positional Encoding

Self-Attention has $O(1)$ maximum path length; when $n < d$, Self-Attention has lower complexity per layer.

11 Word Embedding

Considering a word, you have to represent it as a vector, it means that exists a multi-dimensional space [the dimension is due to the cardinality of the vector], where you have to represent the word. This space is called **Embedding** of the word.
Word Embedding means finding a space where you can represent words. An embedding is a representation.

Imagine you have a certain experience E, and let's name it $D = x_1, x_2, ..., x_N$, you can describe:

- Supervised Learning: given the desired outputs $t_1, t_2, ..., t_N$ learn to produce the correct output given a new set of input
- Unsupervised learning: exploit regualrities in D to build a representation to be used for reasoning or prediction
- Reinforcement learning: [not relevant]

Word Embedding is Unsupervised learning since it is exactly: exploiting regularities in the text to find a space (an embedding) where you can represent the text in a way it makes you useful for reasoning or prediction.

Briefly recall on Neural Autoencoder: get an input, project it in a "latent space" (a space where you have a number of dimension to which the input will be compressed) and then you decompress the input into the original space in such a way that you can reconstruct all the original information.
Doing this, you are:

- Limiting the number of units in hidden layers (compressed representation)
- Constraining the representation to be sparse (sparse representation)

Neurons in the hidden layer represent categories, so the goal is to push the inner representation to have as many zeros as possible (you want that your input belongs to a class or few classes).
This model has an additional constrain that allow to *sparsify* this representation.
This kind of model start from a space which

$$x \in \Re^I \xrightarrow{\text{enc}} h \in \Re^J \xrightarrow{\text{dec}} g \in \Re^I$$
$$I >> J$$

After the decompression, the goal is to have x and g as similar as possible. This is allowed by the minimization of an error function which is

$$E = \underbrace{\|g_i(x_i|w) - x_i\|^2}_{ReconstructionError} + \underbrace{\lambda \sum_j \left| h_j \left(\sum_i w_{ji}^{(1)} x_i \right) \right|}_{SparsityTerm}$$

The first term is the squared error of the reconstructed signal $g(h(x)) - x$; the second is the term that enhance the sparsity.

Natural Language processing treats words as discrete atomic symbols: among the list of all possible objects, the term 'cat' is an id. So words are basically id symbols for objects, and a language is a set of symbols and rules to combine them.
In this way, a sentence is a set of words with very high sparse representation.
One of the typical representation of text is the *Bag-of-Words* representation: take text, represent it with a vector with 1 in the first position if first word in the bag is in your text, 0 otherwise.
If the possible words are N, the space is 2^N since for each word you can have 1 or 0 (there is or nor). The problem is that when you represent text, you will end up with an huge representation which is impossible to explore with data.
Most of the time you divide your vector by the frequency of which the corresponding word appears. So you perform something like TF-IDF: discounts the occurrences by how frequent a term is.

The results is a sparse and high-dimensional vector $\rightarrow$ Curse of Dimensionality: exponential complexity in terms of data that you need to estimate a multi-dimensional distribution. The more is the space and the more sparse it is, the more becomes impossible to learn a distribution.

11.1 Encoding Text

Performance of real-world applications depends on input encoding. There are two main approach to encode text: *Local representation* and *Continuous representation.*
In Local representation, you look to each word by looking around it.

- *Bag-of-Words*: represent a document like a N-dimensional vector, where N is equal to the number of words in your dictionary (bag). The vector contains 1 in the first position if first word in the bag is in your text, 0 otherwise. The same for each position.
- *N-grams*: it represents a word probability by a probability distribution which is based on the previous N-1 terms.
- *1-of-N coding*: it not represent the text but a word. It becomes unfeasible to perform some relevant representation of the text

In Continuous representation, you use some probability distribution to encode the text:

- *Latent Semantic Analysis*
- *Latent Dirichlet Allocation*
- *Distributed Representations*

11.1.1 N-grams

The idea is that you describe the way you write a text by a probability distribution in such a way that if you want to understand what is the probability of a given document, you have to compute the join probability of observing word 1 in position 1, word 2 in position 2 and so on.

So you take a word and, basing on it, you take another bag-of-words where the probabilities are different and they depend on the word extracted. It is complex because you need as many bags as the possible word to be extracted.
For Bag-of-word you need only one bag. Too simple
For N-grams, when N=1, you only need the previous word, so a bag for each word you might extract. In N-grans, in general, you need N^N bags each time.

To be more precise, define a N-grams in a Language Model means:
Determine $P(s = w_1, .., w_k)$ in some domain of interest

$$P(s_k) = \prod_i^k P(w_i|w_1, \ldots, w_{i-1})$$

means that in order to predict the probability of a word in a sentence, you can multiply the probability of each word given the previous ones. In traditional N-grams language models "the probability of a word depends only on the context of $n-1$ previous words"

$$\widehat{P}(s_k) = \prod_i^k P(w_i|w_{i-n+1}, \ldots, w_{i-1})$$

The space is very sparse and the amount of text you have, do not cover it entirely, so you might have a lot of zeros. It might happen that you have never seen a word after another, in particular in the text you are considering. Because of this sometimes you have a smoothing procedure which basically adds one occurrences to every possible combination, so no zero anymore.
Typical ML-smoothing learning process (e.g. Katz 1987), where first compute
$\widehat{P}(w_i|w_{i-n+1}, \ldots, w_{i-1}) = \frac{\#w_{i-n+1},\ldots,w_{i-1},w_i}{\#w_{i-n+1},\ldots,w_{i-1}}$ and then smooth to avoid zero probabilities.

The problem with this approach is that N cannot be too big, the curse of dimensionality depends on N.
Let's assume a 10-gram Language Model on a corpus of 100.00 unique words.

- The model lives in a 10D hyper-cube where each dimension has 100.000 slots
- Model training $\rightarrow$ assigning a probability to each of the 100.000^{10} slots: for each possible combination of 10-words before, compute the probabilities of my 100.000 words.
- *Probability mass vanishes* $\rightarrow$ more data is needed to fill the huge space:
- A solution can be to add more data, but it increases number of unique words! $\rightarrow$ Is not going to work!

In practice, corpuses can have 10^6 unique words, but the contexts are typicaaly limited to size 2 (so 3-gram model). It means that the correlations between words represented in this language model are very short and a lot of information is not captured.

Suppose we observe the following similar sentences:

- *Obama speaks to the media in Illinois*

- *The President addresses the press in Chicago*

With classical one-hot-encoding vector vector space representation, we have:

- speaks = [0 0 1 0 .. 0 0 0 0]
- addresses = [0 0 0 0 .. 0 0 1 0]
- obama = [0 0 0 0 .. 0 1 0 0]
- president = [0 0 0 1 .. 0 0 0 0]
- illinois = [1 0 0 0.. 0 0 0 0]
- chicago = [0 1 0 0 .. 0 0 0 0]

This approach is already unfeasible at computation, furthermore in this way you cannot state when two sentences means the same thing: 'speaks' and 'addresses' are referenced as two different words and the corresponding vectors are orthogonal. The same for each vector. So this model ignores synonyms or other form of symmetries ⇒ **Word Similarity Ignorance**: word pairs share no similarity (don't take into account semantics) and we need word similarity to generalize.

11.1.2 Embedding

Word Embedding is a technique to map a word (or phrase) from it's original high-dimensional input space (the body of all words) to a lower-dimensional numerical vector space - so one embeds the word in a different space.

You start from a long sentence and you want to project it in a lower dimensional space in such a way that this sentence, when you try to reconstruct it, has the same meaning. The idea is that terms with a close semantic are mapped close each other in the lower-dimensional space: closer point are closer in the meaning and they form clusters.
By forcing term with similar meaning to stay close, you have a compression which maintains also the meaning.

How to do it?
Each unique word $\boldsymbol{w}$ in a vocabulary V (typically $|V| > 10^6$) is mapped to a continuous m-dimensional space (typically $100 < m < 500$):

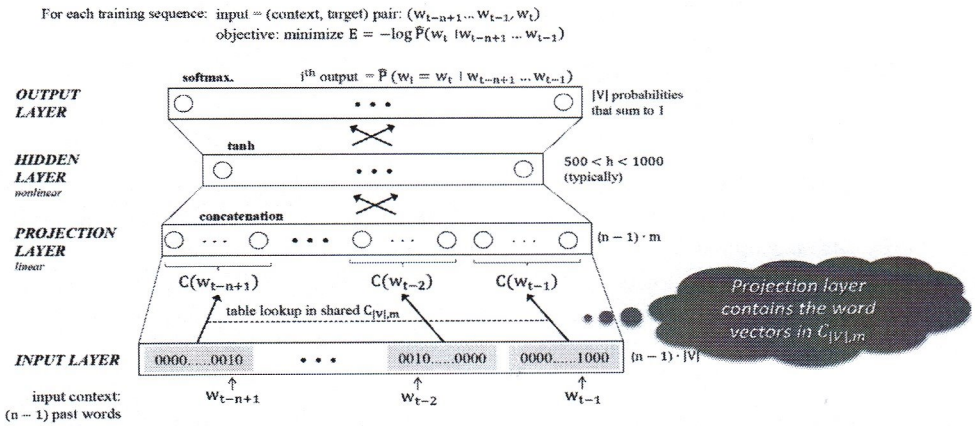
$$w \in V \quad \overset{\text{mapping } C}{\longrightarrow} \quad \mathfrak{R}^m$$

The trick is that you move from a very sparse boolean space in a continuous real-valued space: in the latter you have a lot of space w.r.t. binary one since instead of having only 1 or 0, you can specify a real value.
The goal is to finding a mapping from the original space into a real-valued space.
In this way we are fighting the Curse of Dimensionality with:

- Compression (*dimensionality reduction*): similar words should end up to be close to each other in the feature space.

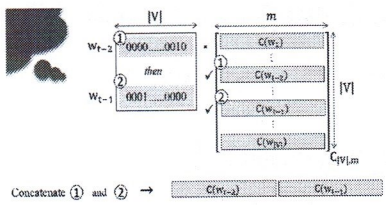
- Smoothing (*discrete to continuous*): if you don't have a word in a particular point, you just need to go close that point to find words with same meaning
- Densification (*sparse to dense*)

One idea to go beyond the Katz model is proposed by Bengio: he used a Neural Autoencoder to build a non-linear continuous language model.
He tried to predict what would be the next word, representing the language model (the prob. of the next word given the previous one) as a classification problem.



The inputs are a $(n-1)$ vectors of size $|V|$ in one-hot-encoding representing past words. You project each vector in a subspace, concatenate these projections and get the predictions by a softmax, trying to predict which word will be the next one. It's not exactly an autoencoder but the idea is to start from an high dimensional space, compress and decompress. It's not symmetric.
It tries to minimize the cross-entropy between the output of the network and the observed class:

$$min\mathrm{E} = -\log \widehat{\mathrm{P}}\left(\mathrm{w_t} | \mathrm{w_{t-n+1}} \dots \mathrm{w_{t-1}}\right)$$



The interesting part is to take the word and project to a smaller representation of size m: to do this, you learn a matrix that takes the word and it compresses it.
This matrix has $m*|V|$ elements and these are the parameters of your model: this matrix compress the one-hot-encoding representation into an m-dimensional representation. [This matrix is the 'trick']

If you have two words (in ohe) of time $t-1$ and $t-2$, you compress them one by one using the matrix, and then concatenate the results.

The brilliant idea of this model is this sharing of weights (the matrix $m*|V|$), because doing the FC explodes in terms of weights.
The training of this network by stochastic gradient descend has a *complexity* of $m*V$ (the matrix) $+ n*m*h$ (it is the FC) $+ h*|V|$ (the softmax).

Once you have trained by SGD, they (Bengio et al.) tested the model which improves the performance of 24% the state of the art.
The problem is that it is unfeasible to use it on large corpus because of the complexity in training, at least in time. So it is not work perfectly overall. The interest part is that Bengio and colleagues were aiming at beating this benchmark and they didn't recognize that they invented Word2Vec.

11.1.3 Word2Vec

The idea is: achieve better performance allowing a simpler (shallower) model to be trained on much larger amounts of data

- No hidden layer (leads to 1000X speed up)
- Projection layer is shared (not just the weights matrix)
- Context contains words both from history and future

The goal of Word2Vec is to overcome the issue in complexity by removing as much parameters as possible and keeps only the relevant ones: the relevant ones are those of the projection from a sparse space in a dense space.
So Word2Vec removed the hidden layer, shared the projection layer and add both word from the history and from the future to predict the missing one.

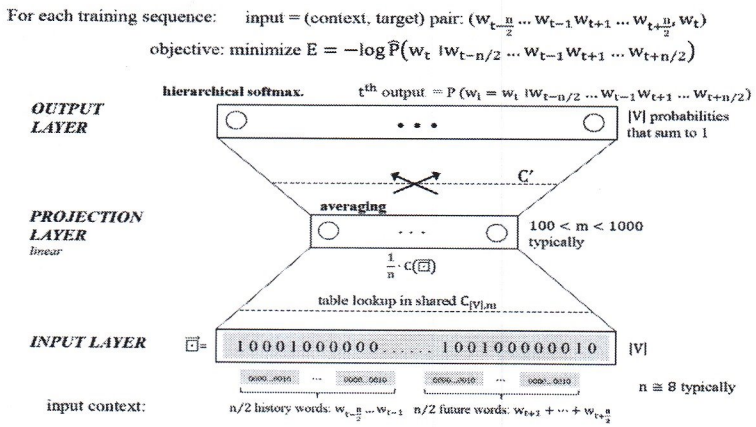
The main idea comes from the following quote:

> *"You shall know a word by the company it keeps"* , John R. Firth, 1957:11.

which means that the model tries to understand a word by the the words next to it.
The meaning of a word is given by the distribution of the word around it. If one word has same words next to it as another word, probably the two words are the same thing. You just need to project the neighbouring word into the same place and you are able to identify similar words.

There are two architecture to simplify the Bengio model

1. *Skip-grams*
2. *Continuous Bag-of-Words*



Procedure: binary encoding, table lookup shared among n words, instead of concatenation you will use averaging because it is smaller in a way that the result is m, not $m*n$ (? not sure about that). From this average you predict one of the words.
The complexity is $n*m+log|V|$: you do not need all the vocabulary but only $log|V|$.
This model somehow is able to learn language model, but you are much more interested in how to learn an embedding of each word in such a way it represents the structure in the language model in this real valued space.

There is some structure in the embedding space where you are projecting the terms and this structure comes from regularities in the text that you want to preserve when you learn a language model.
This structure is somehow a sort of algebraic space where you can do operation. Since you have a vector and you assume that vectors have semantic meanings, the idea is that you can do vector operations.
What is surprising is that this language tries to structure the space according to regularities and tries to preserve them.

11.1.4 GloVe

The meaning of offset vector in embedding space was emerging from the structure of the language. So the idea of GloVe was to try to predict the ratio between co-occurence probabilities: you do not try to predict the word, you try to predict the ratio between the prob. of k following a word wrt k following another word. You are trying to normalize the vector in the embedding space. At the end, you have to find a meaningful matrix to learn the language.

To Word2vec was word given the context, here they try to model directly ratio between co-occurence, so the ratio between one word appearing when there is one or when there is the other. They minimize a more complex term, a weigthed least squares.
GloVe turns out to be more robust and effective because it tries to enforce what word2vec tries to obtain by chance.
GloVe makes explicit what word2vec does implicitly:

- Encodes meaning as vector offsets in an embedding space
- Meaning is encoded by ratios of co-occurrece probabilities