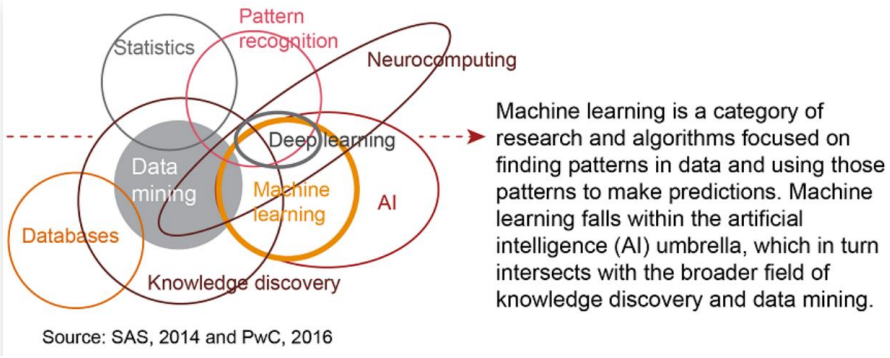
MACHINE LEARNING vs DEEP LEARNING



NN and deep learning are a subset of machine learning. Deep learning is an intersection between AI, pattern recognition and data mining. Deep learning is made to make inference predictions done on the basis of pattern of the data.

MACHINE LEARNING PARADIGMS

Imagine you have a certain experience E, i.e., data, and let's name it $D = x_1, x_2, x_3, \dots, x_N$

• **Supervised learning**: given a training set of pairs (input, desired output) $\{(x_1, t_1), \dots, (x_N, t_N)\}$, learn to produce the correct output of new inputs

For ex:

- CLASSIFICATION → image/data -> hand crafted features -> learned classifier -> classification
- REGRESSION → image/data -> hand crafted features -> learned regressor -> prediction

• **Unsupervised learning**: exploit regularities in D to build a meaningful/compact representation of these, which can help regression/prediction. For ex: CLUSTERING

• **Reinforcement learning**: producing actions which affect the environment, and receiving rewards learn to act in order to maximize rewards in the long term

Example with parcel classification: slide 42 -

In Supervised Learning Classification it is possible to use Rule Based Classifiers or Data Driven Models.

- **Rule Based Classifiers** → It is difficult to grasp what are meaningful dependencies over multiple variables (it is also impossible to visualize these)

- **Data Driven Models**

  They are defined from a training set of supervised pairs

  $$TR = \{(x, t)_i, i = 1, \dots, N\}$$

  The model parameters (e.g. Neural Network weights) are set to minimize a loss function (e.g., the classification error in case of discrete output or the reconstruction error in case of continuous output)

  They can definitively boost the image classification performance

  This is how, during training, the computer learns.

  • Annotated training set is always needed

  • Classification performance depends on the training set

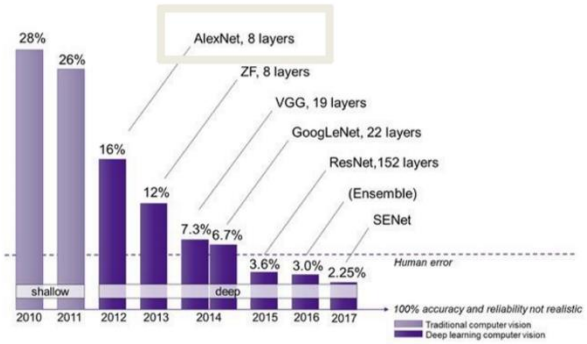
  • Generalization is not guaranteed

Hand Crafted Features

PROS	CONS
• Exploit a priori / expert information • Features are interpretable (you might understand why they are not working) • You can adjust features to improve your performance • Limited amount of training data needed • You can give more relevance to some features	• Requires a lot of design/programming efforts • Not viable in many visual recognition tasks that are easily performed by humans (e.g. when dealing with natural images) • Risk of overfitting the training set used in the feature design • Not very general and "portable"

DEEP LEARNING → DATA DRIVE FEATURES

The feature extraction part is done by several layers of non linear units, which extract characteristics from data. So the features are obtained automatically by the algorithm and are not hand-crafted.



The graph shows the classification error -> in 2012 we have a drop in the errors, this is thanks to the introduction of deep learning and machine learning data drive features

Deep Learning became possible thanks to:

- Large collections of annotated data - IMAGENET
  The ImageNet project is a large visual database designed for use in visual object recognition software research. More than 14 million images have been hand-annotated by the project to indicate what objects are pictured and in at least one million of the images, bounding boxes are also provided. ImageNet contains more than 20,000 categories

- Parallel Computing Architectures

- Software libraries (Tensorflow and Pytorch)

- New Network architectures

Advanced Visual Recognition Problems with DL

- Objects recognition,
- enhance images resolution,
- stile transfer,
- image captioning,
- artificial faces

FROM PERCEPTRONS TO FEED FORWARD NEURAL NETWORKS

1940s

Computers were already good at

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

However we would have liked them to:

- Interact with noisy data or directly with the environment
- Be massively parallel and fault tolerant
- Adapt to circumstances

→ Researchers were seeking a computational model beyond the Von Neumann Machine!

BRAIN COMPUTATIONAL MODEL

The human brain has a huge number of computing units:

- 10^{11} (one hundred billion) neurons
- 7,000 synaptic connections to other neurons
- In total from 10^{14} to 5 x 10^{14} (100 to 500 trillion) in adults to 10^{15} synapses (1 quadrillion) in a three year old child

The computational model of the brain is:

- Distributed among simple nonlinear units
- Redundant and thus fault tolerant
- Intrinsically parallel

PERCEPTRON

Perceptron: a computational model based on the brain

Excitatory and inhibitory synapses → the charge in the neuron increase or decreases

The first artificial neuron (the perceptron) used to mimic this behaviour → circuitry with charge that could decrease or decrease

Computation in Artificial Neurons

Information is transmitted through chemical mechanisms:

• Dendrites collect charges from synapses, both Inhibitory and Excitatory

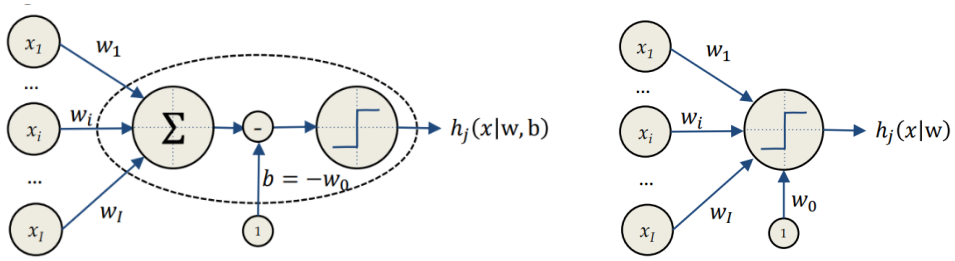
Positive coeff -> exitatory synapses

Negative coeff -> inhibitory synapse

• Cumulates charge is released (neuron fires) once a Threshold is passed

If the difference between the synapses (weighted sum of the inputs) and the bias is positive → we get a firing action (in binary representation 1)

If it is negative → no firiging (in binary representation 0 or -1)

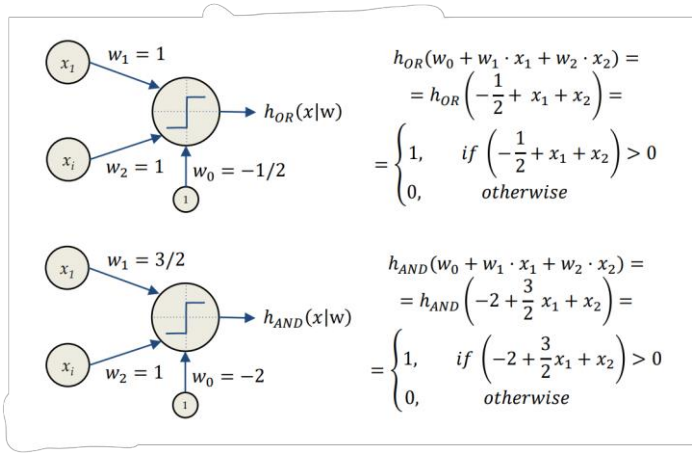


$$h_j(x|\mathrm{w},\mathrm{b}) = \mathrm{h_j}\left(\Sigma_{i=1}^{I} w_i \cdot x_i - b\right) = \mathrm{h_j}\left(\Sigma_{i=0}^{I} w_i \cdot x_i\right) = h_j(w^T x)$$

the output of the neuron is given by a nonlinear function that takes in input the scalar product between vector of dimension I+1

Several researchers were investigating models for the brain
• In 1943, Warren McCullog and Walter Harry Pitts proposed the Treshold Logic Unit or Linear Unit, the activation function was a threshold unit equivalent to the Heaviside step function
• In 1957, Frank Rosemblatt developed the first Perceptron. Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors. The perceptron was a piece of hardware.
• In 1960, Bernard Widrow introduced the idea of representing the threshold value as a bias term in the ADALINE (Adaptive Linear Neuron or later Adaptive Linear Element)

We can get programmable HW with logic units by just changing the weights: we get AND and OR



HEBBIAN LEARNING
"The strength of a synapse increases according to the simultaneous activation of the relative input and the desired target"
Hebbian learning can be summarized by the following:

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

- Start from a random initialization
- Fix the weights one sample at the time (online) and only if the samples is not correctly predicted

- η is the learning rate (in the perceptron is not so essential, it might be just 1)
- x_i^k is the ith perceptron input at time k
- t^k is the desired output at time k

NB. X can be positive or negative, t can be positive or negative

If X is positive and t is positive → Δw will be positive -> w increases
If X is positive and t is negative→ Δw will become negative -> w decreases
This is the procedure of setting the weights → iterative process (k is the iteration)

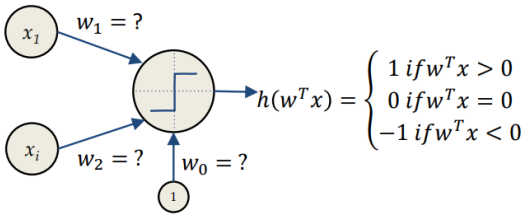
NB:
That fact that we fix the weights one sample at the time is called ONLINE procedure
This DOES converge (if the problem can be learned, it does always converge)

Epoch: a complete cycle with all the data
Once we have done an epoch we restart again until all the weight are correct

Example:

Learn the weights to implement the OR operator

- Start from random weights, e.g., w=[1, 1, 1]
- Chose a learning rate, e.g. η =0.5
- Cycle through the records by fixing those which are not correct
- End once all the records are correctly predicted



x_0	x_1	x_2	OR
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1

$w^0 = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \begin{matrix} w_0 \\ w_1 \\ w_2 \end{matrix} \quad \eta = 0,5$

(1) $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [1\ -1\ -1] = 1-1-1 = -1 < 0$ ✓

(2) $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [1\ -1\ 1] = 1-1+1 = 1 > 0$ ✓

(3) $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [1\ 1\ -1] \quad 1 > 0$ ✓

(4) $\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} [1\ 1\ 1] = 3 > 0$ ✓

→ the first initialization (w^0) was already the best one.

Different initialisation:

$w_0^0 = [0,0,0]$ → EPOCH 1

(1) $w_0^{0T} x_1 = 0 \Rightarrow w^1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_0 t \\ x_1 t \\ x_2 t \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1\ (-1) \\ -1\ (-1) \\ -1\ (-1) \end{bmatrix} = \begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \end{bmatrix}$

(2) $w^{1T} x_2 = 0 \quad \Rightarrow w^2 = \begin{bmatrix} -1/2 \\ -1/2 \\ -1/2 \end{bmatrix} + 1/2 \begin{bmatrix} 1 \cdot 1 \\ -1 \cdot 1 \\ 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$

$-1/2$

(3) $w^{2T} x_3 = -1 \Rightarrow w^3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} + 1/2 \begin{bmatrix} 1 \cdot 1 \\ 1 \cdot 1 \\ -1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1/2 \\ -1/2 \\ 1/2 \end{bmatrix}$

④ $w^3 x_4 = 1$ ok !

→ EPOCH 2

$w = \begin{bmatrix} 1/2 \\ 1/2 \\ 1/2 \end{bmatrix}$ [1 -1 -1] [1 -1 1] [1 1 -1] [1 1 1] ⇒ NO correction needed ⇒ w ok !

Does the procedure always converge? YES
Does it always converge to the same sets of weights? NO

Perceptron Maths

A perceptron computes a weighted sum, returns its Sign (Thresholding)

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

It is a linear classifier for which the decision boundary is the hyperplane

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

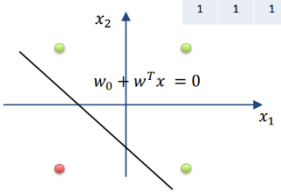
In 2D, this turns into

$$w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 = 0$$
$$w_2 \cdot x_2 = -w_0 - w_1 \cdot x_1$$
$$x_2 = -\frac{w_0}{w_2} - \frac{w_1}{w_2} \cdot x_1$$

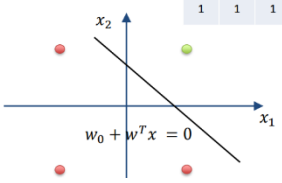
Linear boundary explains how Perceptron implements Boolean operators

Yes! But this ia a single trainable HW!

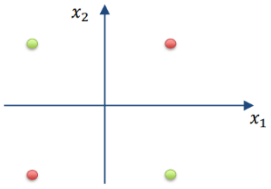
x_0	x_1	x_2	OR
1	-1	-1	-1
1	-1	1	1
1	1	-1	1
1	1	1	1



x_0	x_1	x_2	AND
1	-1	-1	-1
1	-1	1	-1
1	1	-1	-1
1	1	1	1



What if the dataset we want to learn does not have a linear separation boundary?

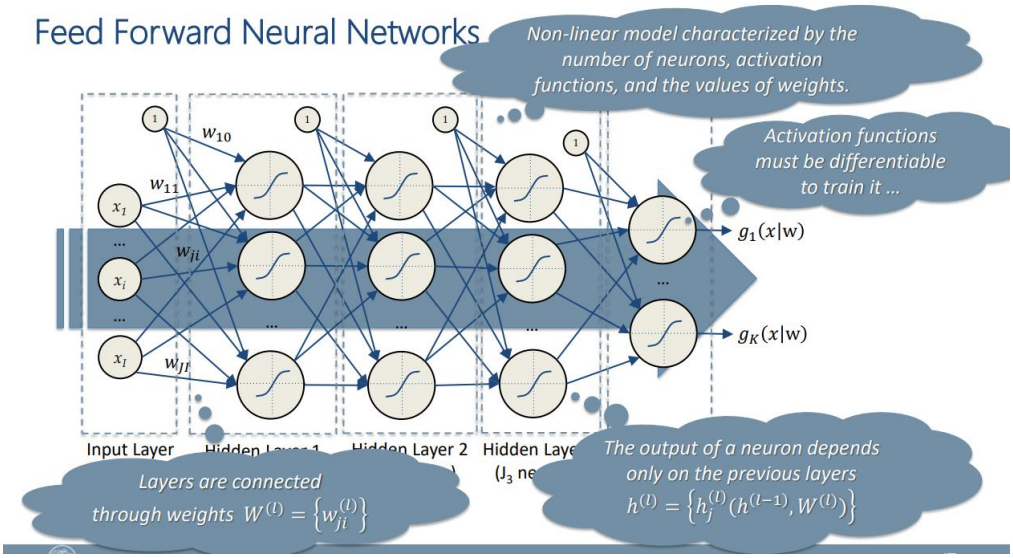


The Perceptron does not work any more and we need alternative solutions

- Non linear boundary
- Alternative input representations

FEEDFORWARD NEURAL NETWORKS

→ Perceptrons organized in layers in order to get a nonlinear boundary
If we want to stack multiple layers of perceptrons the hebbian learning does not work anymore
The idea was good but there was a problem realated the procedure for training (Hebbian).
→ The solution came with the backpropagation algorithm to learn parameters.



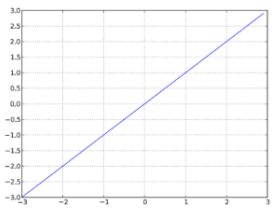
This model can be thanked as a nonlinear function. The output depends on the number of inputs, the number of outputs, hidden layers and the type of activation function inside each neuron, the weights and the bias. This is a very complex non linear funcion.

The weights of every layer can be resumed in a matrix (we are considering a fully connected network)

$$\begin{bmatrix} w_{10} & \cdots & w_{1I} \\ \vdots & \ddots & \vdots \\ w_{J_1 0} & \cdots & w_{J_1 I} \end{bmatrix}$$

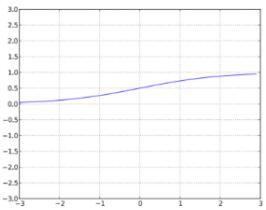
The matrix is $(I+1)$ x $J1$
J1 is the number of neuros in hidden layer 1
→ we have a matrix for every layer
For layer 2 the matrix is $(J1+1)$ x $J2$
For layer 3 the matrix is $(J2+1)$ x $J3$
Etc...

ACTIVATION FUNCTION



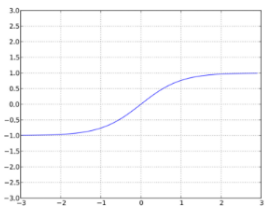
Linear activation function

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



Sigmoid activation function

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



Tanh activation function

$$g(a) = \frac{\exp(a)-\exp(-a)}{\exp(a)+\exp(-a)}$$
$$g'(a) = 1-g(a)^2$$

The sigmoid is kind of a continuous approximation of the step function
The sigmoid has output [0,1]
The tanh has output [-1, 1]

These are the classical activation function. They have some problems related to the norm of the gradient.

NB:
we don't need to have the same activation function for every layer
In principle we could have different functions for every neuron (but this leads to very complex model, we would never come up with a right solution)

Output Layer in Regression and Classification
In Regression the output spans the whole $\Re$ domain:
• Use a Linear activation function for the output neuron

In Classification with two classes, chose according to their coding:
• Two classes $\Omega_0 = -1$, $\Omega_1 = +1$ then use Tanh output activation
• Two classes $\Omega_0 = 0$, $\Omega_1 = 1$ then use Sigmoid output activation (it can be interpreted as class posterior probability)

When dealing with multiple classes (K) use as many neuron as classes ("one hot" encoding)
Classes are coded as {Ω_0 = [0 0 1], Ω_1 = [0 1 0], Ω_2 = [1 0 0]}
Output neurons use a SoftMax unit

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

Neural Networks are Universal Approximators
"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)

Regardless the function we are learning, a single layer can represent it:
• Doesn't mean a learning algorithm can find the necessary weights!
• In the worst case, an exponential number of hidden units may be required
• The layer may have to be unfeasibly large and may fail to learn and generalize

Classification requires just one extra layer (with respect to regression)…

BUT the more neuros we have, the more parameters we have -> the more flexible the network is

OPTIMIZATION AND LEARNING

Recall about learning a parametric model $y(x_n|\theta)$ in regression and classification
• Given a training set

$$D = < x_1, t_1 > \cdots < x_N, t_N >$$

• We want to find model parameters such that for new data

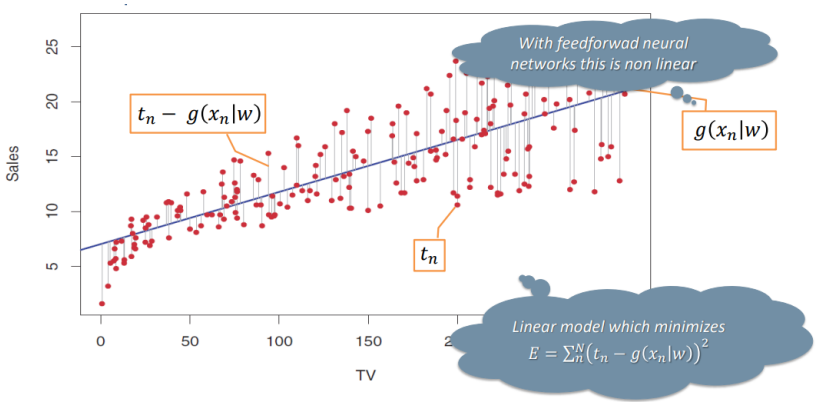
$$y(x_n|\theta) \sim t_n$$

Let's call h the activation function of the hidden neurons and let's call g of the output node (this is because hidden neuron and output neuron have different functions)
The output of the network is given by: $g(\sum_0^J w_J \, h_J \, (\sum_0^I w_{Ji} x_i))$
The main goal is to make g(…) approximate the desired output by changing the values of the weights:

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

For this you can minimize $E = \sum_n^N (t_n - g(x_n|w))^2$

SUM OF SQUARED ERRORS



NON LINEAR OPTIMIZATION

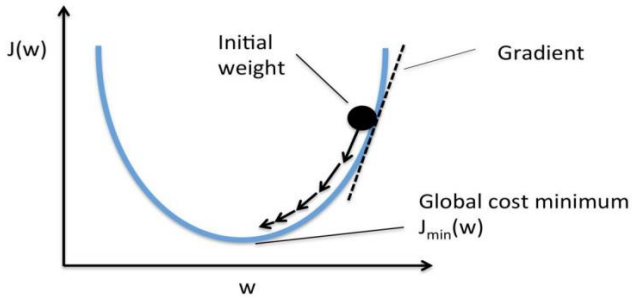
To find the minimum of a generic function, we compute the partial derivatives of the function and set them to zero

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

Closed-form solutions are practically never available so we can use iterative solutions:

- Initialize the weights to a random value
- Iterate until convergence

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



NB: gradient descent = backpropagation

GRADIENT DESCENT – BACKPROPAGATION

Finding the weighs of a Neural Network is a non linear optimization

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

We iterate starting from an initial random configuration

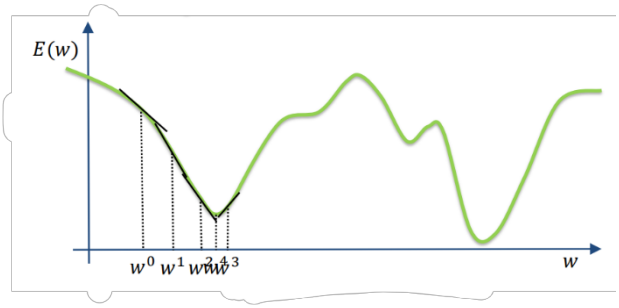
$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^k}$$

We compute a step on the opposite direction of the gradient (which is the direction of increase of the weights).
This is why we have a - in the formula for updating the weights
If the gradient is positive -> reduce the weights
If the gradient is negative -> increase the weights
The learning rate is the size of the step we make at each iteration. This has to be set in an appropriate way, in order not to go right and left in a zig zag way.

To avoid local minima can use momentum

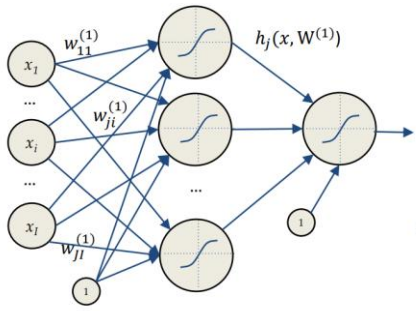
$$w^{k+1} = w^k - \eta \frac{\partial E(w)}{\partial w}\bigg|_{w^k} - \alpha \frac{\partial E(w)}{\partial w}\bigg|_{w^{k-1}}$$

Use multiple restarts to seek for a proper global minimum.
It depends on where we start from



The solution depends on where we start with out initialization (we could converge to a local minimum)
To avoid local minima we use the momentum (this is one of the variations of the standard backpropagation)
NB: Usually they suggest to make different trials and keep the one that provides the lower minimum

EXAMPLE – Gradient descent



$$g_1(x_n|w) = g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{i=0}^{I} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Compute the wji (1) weight update formula by gradient descent.
NB: $w_{j,i}$ is the weight of the j-th neuron that multiplies the i-th input

Use j=3 and i=5

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

$$\frac{\partial E(w)}{\partial w_{3,5}^{(1)}} = \frac{\partial \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2}{\partial w_{3,5}^{(1)}} = \sum_{n=1}^{N} \frac{\partial (t_n - g_1(x_n, w))^2}{\partial w_{3,5}^{(1)}} = -2 \sum_{n}^{N} (t_n - g_1(x_n, w)) \frac{\partial g_1(x_n, w)}{\partial w_{3,5}^{(1)}}$$

$$\frac{\partial g_1(x_n, w)}{\partial w_{3,5}^{(1)}} = \frac{\partial g_1\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j(.)\right)}{\partial w_{3,5}^{(1)}} = g_1'(x_n, w) \cdot \frac{\partial \sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j(.)}{\partial w_{3,5}^{(1)}} = g_1'(x_n, w) \cdot w_{1,3}^{(2)} \cdot \frac{\partial h_3\left(\sum_{i=0}^{I} w_{3i}^{(1)} \cdot x_{i,n}\right)}{\partial w_{3,5}^{(1)}}$$

$$\frac{\partial h_3\left(\sum_{i=0}^{I} w_{3i}^{(1)} \cdot x_{i,n}\right)}{\partial w_{3,5}^{(1)}} = h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) \frac{\partial \sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}}{\partial w_{3,5}^{(1)}} = h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) x_{5,n}$$

$$\frac{\partial E(w)}{\partial w_{3,5}^{(1)}} = -2 \sum_{n}^{N} (t_n - g_1(x_n, w)) g_1'(x_n, w) w_{1,3}^{(2)} h_3'\left(\sum_{i=0}^{I} w_{3,i}^{(1)} \cdot x_{i,n}\right) x_{5,n}$$

- $x_{i,n}$ means that it is the i-th element in the vector x_n, where n is the number of sample
- g_1' depends on the nature of g1.

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

Before updating a weight we sum the derivatives over all samples → using all the samples is called **BATCH**
Modern libraries of ANN put in parallel all the computations in the GPU -> this batch approach works if we can load all the data on our GPU, otherwise we have to iterate on the memory of the processor and so all the advantage is lost. HOWEVER when working with complex data and models this does not work -> the GPUs are not large enough

GRADIENT DESCENT VARIATIONS

- Batch gradient descent

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

If we don't fit all the data in the memory → **ONLINE gradient descent**

- Stochastic gradient descent (SGD)
  Use a single sample, unbiased, but with high variance

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

- Mini-batch gradient descent
  Use a subset of samples, good trade off variance-computation

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

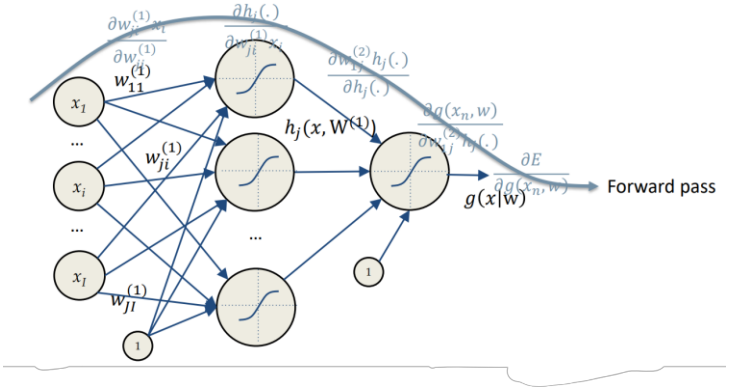
BACKPROPAGATION AND CHAIN RULE

Weights update can be done in parallel, locally, and requires just 2 passes

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

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

The same holds for backpropagation



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

Forward pass: same operation as when we compute the output of the network, with the difference that we store the value of the derivatives (while normally we just compute the output of the network)

We can perform backward pass → that's the opposite direction of forward pass (this is feasible because the flow of data in the forward NN goes only in one direction).
The backward pass relies on the fact that we store derivatives during forward pass

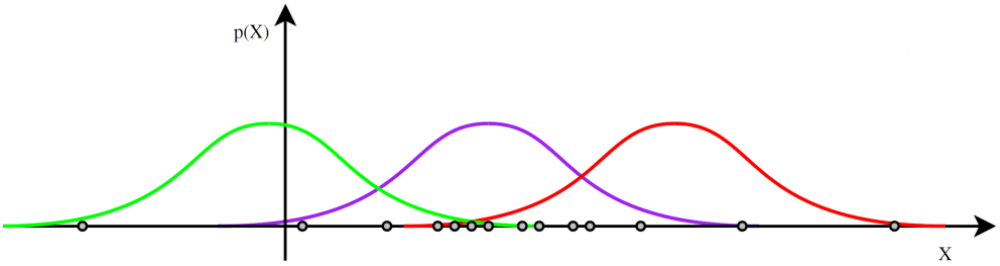
NB the computation for each weight can be parallelized in the gpu!

MAXIMUM LIKELIHOOD ESTIMATION

In statistics, maximum likelihood estimation (MLE) is a method of estimating the parameters of an assumed probability distribution, given some observed data. This is achieved by maximizing a likelihood function so that, under the assumed statistical model, the observed data is most probable.

Let's observe i.i.d. samples from a Gaussian distribution with known σ 2

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



The violet hypothesis makes the most of the points likely to be observed

Maximum Likelihood: Chose parameters which maximize data probability

Let θ =(θ1, θ2, … , $\theta p)^T$ a vector of parameters, find the MLE for θ:

1. Write the likelihood $L = P(Data|\theta)$ for the data
   OR Take the logarithm of likelihood l = log $P(Data|\theta)$, 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
2. Work out $\partial L/\partial\theta$ or $\partial l/\partial\theta$ using high-school calculus
3. Solve the set of simultaneous equations $\partial L/\partial\theta i$ = 0 or $\partial l/\partial\theta i$ = 0
4. Check that θ_{MLE} is a maximum

To maximize/minimize the (log)likelihood you can use:
• Analytical Techniques (i.e., solve the equations)
• Optimization Techniques (e.g., Lagrange multipliers)
• Numerical Techniques (e.g., gradient descend)

Write the likelihood $L = P(Data|\theta)$ for the data

$$L(\mu) = p(x_1, x_2, \dots, x_N|\mu, \sigma^2) = \prod_{n=1}^{N} p(x_n|\mu, \sigma^2) =$$

$$= \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x_n-\mu)^2}{2\sigma^2}}$$

Take the logarithm $l = \log P\ Data|\theta$ of the likelihood

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

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

Work out $\partial l/\partial\theta$ using high-school calculus

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

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

Solve the set of simultaneous equations $\partial l\ \partial\theta i = 0$

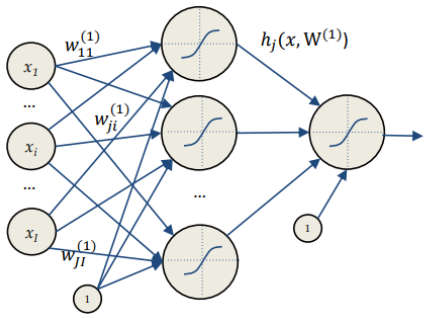
$$\frac{1}{2\sigma^2}\sum_{n}^{N}2(x_n-\mu) = 0$$

$$\sum_{n}^{N}(x_n-\mu) = 0$$

$$\sum_{n}^{N}x_n = \sum_{n}^{N}\mu \quad \Rightarrow \quad \mu^{MLE} = \frac{1}{N}\sum_{n}^{N}x_n$$

Let's apply this all to Neural Networks!

NEURAL NETWORKS FOR REGRESSION

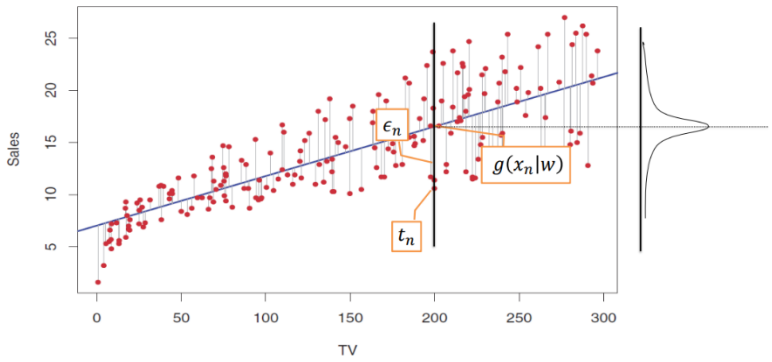


$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)}\cdot h_j\left(\sum_{j=0}^{J} w_{ji}^{(1)}\cdot x_{i,n}\right)\right)$$

Goal: approximate a target function t having N observations

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

→ our target data is distributed as a gaussian with mean given by the output of the network and a certain sigma
We can use maximum likelihood to compute the parameters of this distribution



We have a gaussian function that MOVES together with our regression function

We have i.i.d. samples coming from a Gaussian with known σ^2

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

Write the likelihood $L = P(Data|\theta)$ for the data

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

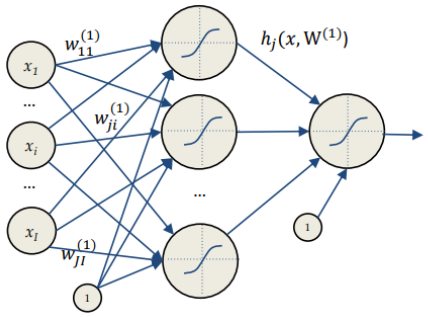
Write the loglikelihood l = log $P\ Data|\theta$ for the data

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

Look for the weights which maximixe the loglikelihood

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

NEURAL NETWORKS FOR CLASSIFICATION



$$g(x_n|w) = g\left(\sum_{j=0}^{J} w_{1j}^{(2)} \cdot h_j\left(\sum_{j=0}^{J} w_{ji}^{(1)} \cdot x_{i,n}\right)\right)$$

Goal: approximate a posterior probability t having N observations

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

We have some i.i.d. samples coming from a Bernulli distribution.
We can't considered an additive gaussian noise, even in the binomial case: the output will be always 1 or 0
→ it would be a wrong statistical assumption

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

Write the likelihood $L = P(Data|\theta)$ for the data

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

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

Compute the log likelihood $l = logP(Data|\theta)$ for the data

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

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

Look for the weights which maximize the loglikelihood → crossentropy

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

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

HOW TO CHOOSE ERROR FUNCTION

We have observed different error functions so far.

Sum of Squared Errors

$$E(w) = \sum_{n=1}^{N} (t_n - g_1(x_n, w))^2$$

Binary Crossentropy

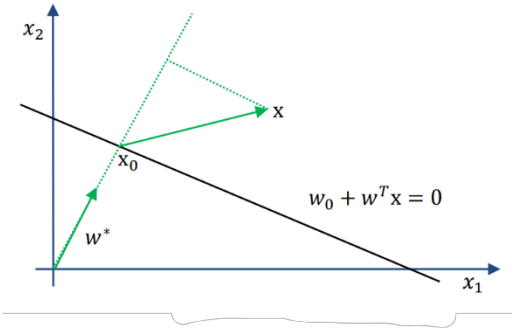
$$E(w) = -\sum_{n}^{N} t_n \, log \, g(x_n|w) + (1 - t_n) \, log(1 - g(x_n|w))$$

what is doing the network? task of the network written in the loss function! → SSE for regression and crossentropy for classification.

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

- Use all your knowledge/assumptions about the data distribution
- Exploit background knowledge on the task and the model (as for the perceptron)
- Use your creativity! = this requires lots of trials and errors

HYPERPLANES LINEAR ALGEBRA



Let consider the hyperplane (affine set) $L \in \Re 2$

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

Any two points x1 and x2 on $L \in \Re^2$ have

$w^T (x_1 - x_2) = 0$

The versor normal to $L \in \Re^2$ is then

$$w^* = \frac{w}{\|w\|}$$

For any point x0 in $L \in \Re^2$ we have

$w^T x_0 = -w_0$

The signed distance of any point x from $L \in \Re^2$

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

→ (wT x + $w0$) is proportional to the distance of x from the plane defined by wT x + $w0$ = 0

PERCEPTRON LEARNING ALGORITHM

It can be shown, the error function the Hebbian rule is minimizing is the distance of misclassified points from the decision boundary.

Let's code the perceptron output as +1/-1

- If an output which should be +1 is misclassified then wT x + w0 < 0
- For an output with -1 we have the opposite

The goal becomes minimizing

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

Set of points misclassified

This is non negative and proportional to the distance of the misclassified points from wT x + w0 = 0

Let's minimize by stochastic gradient descend the error function

$$D(w, w_0) = -\sum_{i \in \mathrm{M}} t_i(\mathrm{w}^{\mathrm{T}}\mathrm{x_i} + \mathrm{w}_0)$$

The gradients with respect to the model parameters are

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

Stochastic gradient descent applies for each misclassified point

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

Hebbian learning implements Stocastic Gradient DescenT

NEURAL NETWORKS

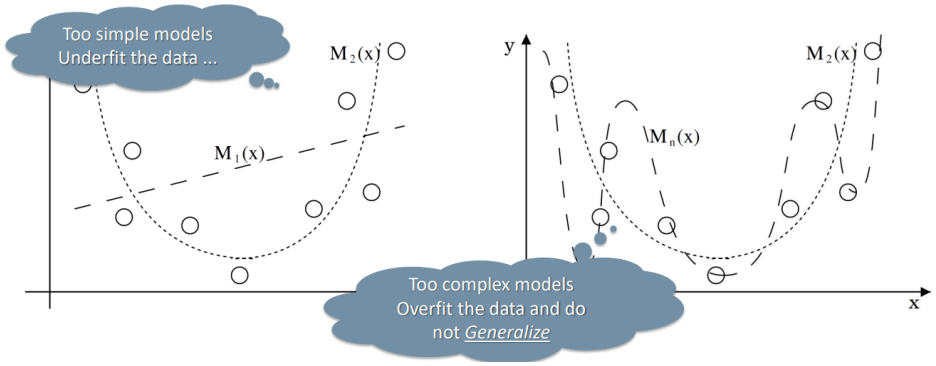
TRAINING AND OVERFITTING

"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)

Neural Networks are Universal Approximators
Regardless of what function we are learning; a single layer can do it …
- … but it doesn't mean we can find the necessary weights!
- … but an exponential number of hidden units may be required
- … but it might be useless in practice if it does not generalize!

Ockham razor -> if we have two models with the same performance, we should go with the simplest one (the complex one is more likely to overfit)

MODEL COMPLEXITY
Inductive Hypothesis: A solution approximating the target function over a sufficiently large set of training examples will also approximate it over unobserved examples

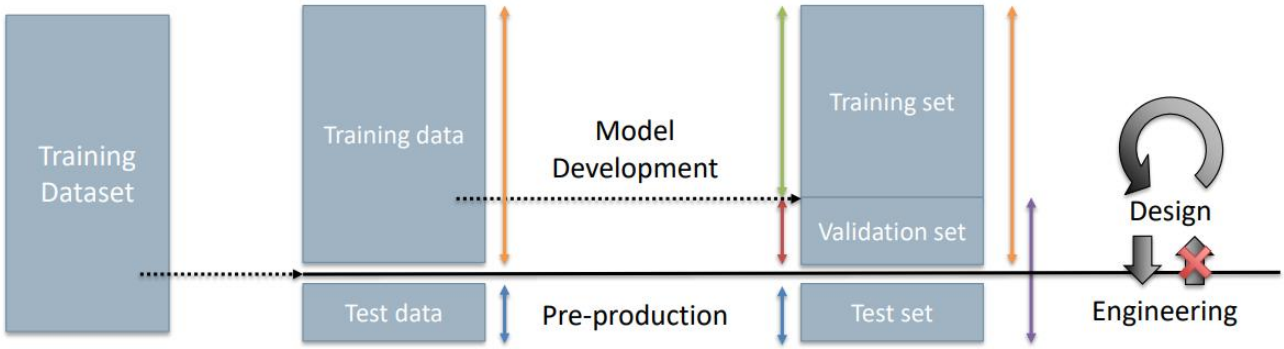


GENERALIZATION
Training error/loss is not a good indicator of performance on future data:
- The classifier has been learned from the very same training data, any estimate based on that data will be optimistic → very high bias
- New data will probably not be exactly the same as training data
- You can find patterns even in random data

We need to test on an independent new test set
- Someone provides you a new dataset
- Split the data and hide some of them for later evaluation
- Perform random subsampling (with replacement) of the dataset

The test is the last thing we do, if we go back we cheat. We need to test the model on a fully independent test and NOT adjust the model on the test results.

In classification preserve class distribution, i.e., stratified sampling!



• Training dataset: the available data

• Training set: the data used to learn model parameters
• Test set: the data used to perform final model assessment
• Validation set: the data used to perform model selection

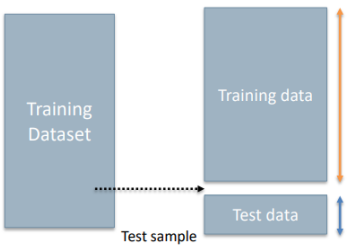
• Training data: used to train the model (fitting + selection)
• Validation data: used to assess the model quality (selection + assessment) = (validation set + test set)

Model development: deciding the structure of the model and train the weights.

We train with training set two neural networks with different number of neurons, we test the two networks with the validation. We train a third network with more neurons and a forth with even more neurons, the performance increases but at some point we incur in overfitting, so we stop and look at the best model on the validation set. At this point we have the best model on the validation set and finally we use the test set.
NB: After the test if we don't like the outcome we should restart all the process from scratch.

If we have done the things in the right way (stratified sampling, no mistake in splitting data etc), the validation error should be very close to the test set error. If they are different, we have a symptom of overfitting.

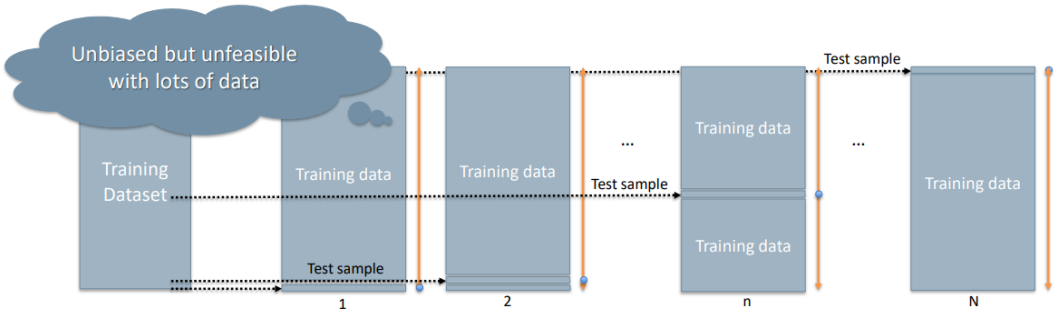
CROSS VALIDATION
Cross-validation is the use of the training dataset to both train the model (parameter fitting + model selection) and estimate its error on new data
• When lots of data are available use a **Hold Out** set and perform validation



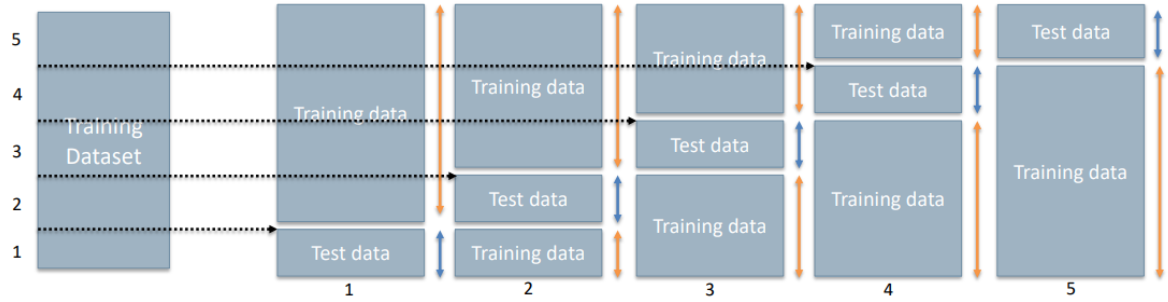
Hold out error might be biased by the specific split!
Hold out cross validation problem: depends on how luck we are on spitting the data (there could be more easy and more difficult samples). We need a better mix.

• When having few data available use **Leave-One-Out** Cross-Validation (LOOCV)
We don't trust our luck, we hide one sample, train all the other and test on that sample and so on by changing the hidden sample.
We get an unbiased estimate on the dataset of the performance of our network. We count how many often we have classified correctly / or we make an average over the regression error.

LOOCV problem: we can do it if we have few data, but cannot expect to train 200 models. This is not really used in deep learning, there are some variations that can be used.



• **K-fold Cross-Validation** is a good trade-off (sometime better than LOOCV)



K folds cross validation is the most used in ML, not often in deep learning, but for now we assume that we can use it. We split the training data in k equal groups, and we perform train and test k times and then we average over the k in order to get the overall performance.
The k fold error is the average.

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

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

We can use k fold for *MODEL SELECTION*. We can obtain how good is a model with x number of neurons over k splits, so it's more accurate. We can decide on the average error what is the best network (with n neurons, and all specific parameters like learning rate etc). At this point we re-train the network with all data (training data) At that point we have the model testing.

All the procedure we are talking about work for the model development part.
We use the k fold for model development, train the model on all training data and then we test. We can use k fold for model assessment, but usually we do k fold for model selection (at validation).

Most of the time we will use hold out method because k-fold needs to train too many models and it is very time consuming.

PREVENTING OVERFITTING

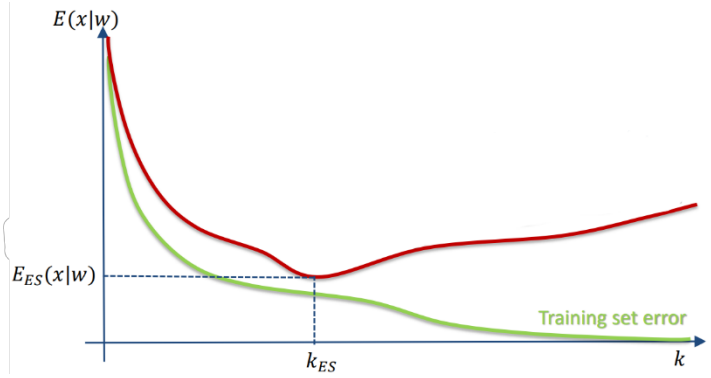
Early Stopping: Limiting Overfitting by Cross-validation

Overfitting networks show a monotone training error trend (on average with SGD) as the number of gradient descent iterations k, but they lose generalization at some point ...

• Hold out some data

• Train on the training set

• Perform cross-validation on the hold out set
While we perform gradient descent on training set we check the performance on the validation set (we just look at the performance, not trying to make it better).



• Stop train when validation error increases
At some point the error on the training set will decrease, while the error in the validation will stop decreasing or it will even increase. If our model is powerful enough it will learn the noise on the training set, which is different from the noise of the validation set. We stop the learning before the model starts to learn the nose of the training set → We stop training at some epoch where the validation error stops decreasing.

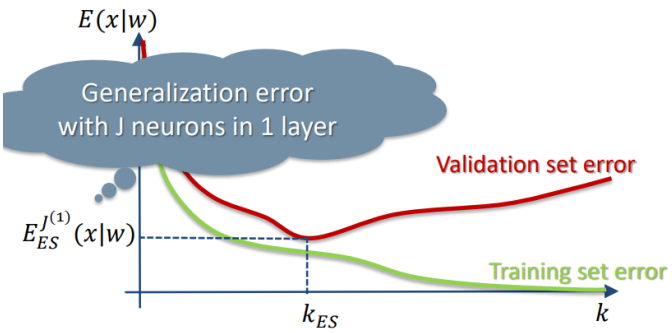
We set a x number of epochs of patients, if in x epochs the validation error does not decrease, we stop the learning.

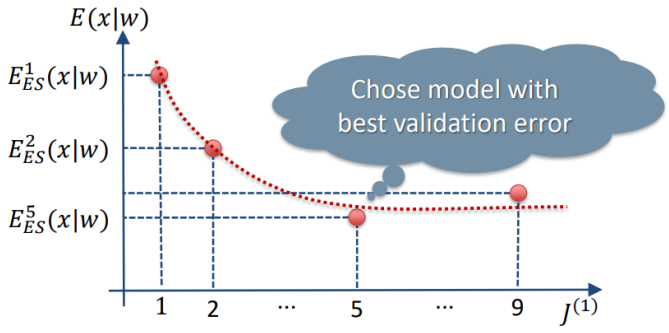
Cross-validation and Hyperparameters Tuning

= grid search

Model selection and evaluation happens at different levels:
• Parameters level, i.e, when we learn the weights w for a neural network
• Hyperparameters level, i.e., when we chose the number of layers L or the number of hidden neurons J (l) for a given layer

In order to set the hyper parameters, we need to train several networks with different hyper parameters and of each we get the early stopping error. By increasing the power of the network, the validation error will decrease at first but then it will stop decreasing and eventually increase. We select the model with the best validation error.





Weights decay

→ Limiting Overfitting by Weights Regularization

Performing early stopping means that we will not use 20% on average in the training data (we need it for the validation data). We are not exploiting all the data available. This is a method that does not take away data.

One of the reasons why the neural network overfits is because it is set free on its behaviour. Regularisation sets request and limits the freedom of the NN. → A priori constraints.
One way is to put a priori distribution on the weights.

Regularization is about constraining the model «freedom», based on a-priori assumption on the model, to reduce overfitting

So far, we have maximized the data likelihood (max likelihood method):

$$w_{MLE} = argmax_w\, P(D|w)$$

We can reduce model «freedom» by using a Bayesian approach:

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

Small weights observed to improve generalization of neural networks:

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

We try to push our weights to be small, so have a null average with a certain standard deviation

Hp: all the weights are together and come from the same distribution (Q refers to all weights)

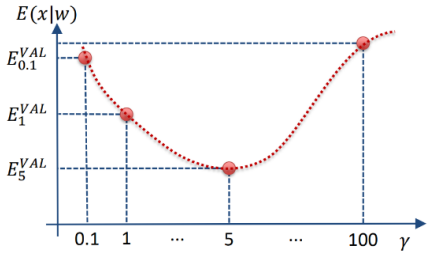
$$\begin{aligned} \hat{w} &= argmax_w\, P(w|D) = argmax_w\, P(D|w)\, P(w) \\ &= argmax_w \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t_n - g(x_n|w))^2}{2\sigma^2}} \prod_{q=1}^{Q} \frac{1}{\sqrt{2\pi}\sigma_w} e^{-\frac{(w_q)^2}{2\sigma_w^2}} \\ &= argmin_w \sum_{n=1}^{N} \frac{(t_n - g(x_n|w))^2}{2\sigma^2} + \sum_{q=1}^{Q} \frac{(w_q)^2}{2\sigma_w^2} \\ &= argmin_w \underbrace{\sum_{n=1}^{N} (t_n - g(x_n|w))^2}_{\text{Fitting}} + \gamma \underbrace{\sum_{q=1}^{Q} (w_q)^2}_{\text{Regularization}} \end{aligned}$$

With this regularization we penalize a lot weights that are > 1, while we give more space to weigth in [0,1]

Recall Cross-validation and Hyperparameters Tuning

You can use cross-validaton to select the proper γ:

- Split data in training and validation sets
- Minimize for different values of γ
- Evaluate the mode
- Chose the $\gamma*$ with the best validation error
- Put back all data together and minimize



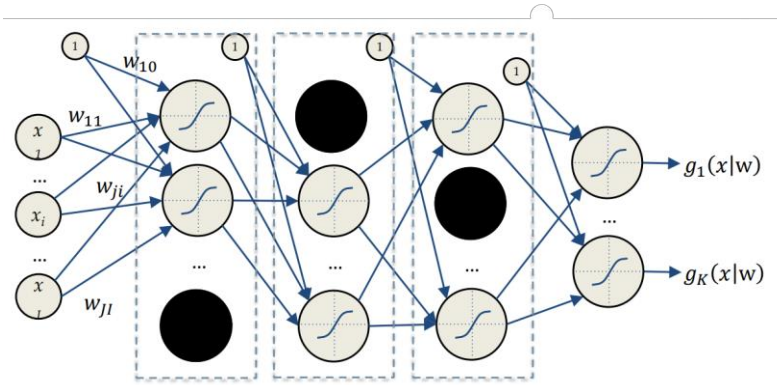
$\gamma \rightarrow$ infinite = we don't care about the data and we want everything to be zero (nothing woks)

γ = 0 we get standard overfitting

Dropout

By turning off randomly some neurons we force to learn an independent feature preventing hidden units to rely on other units (co-adaptation, which can lead to overfitting):

Each hidden unit is set to zero with pj (l) probability, e.g., pj (l) = 0.3



$$m^{(l)} = [m_1^{(l)}, \dots m_{J^{(l)}}^{(l)}]$$

$$m_j^{(l)} \sim Be\left(p_j^{(l)}\right)$$

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

We change the mask for the neurons at each iterations

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)
<-> At the end we have trained some submodels and we merge them together
→ Behaves as an ensemble method

NB: this has proved to improve the generalisation capability (reduces the difference from the training and the validation error).

NB: we can mix the 3 methods, but this implies that we need to tune all the parameters

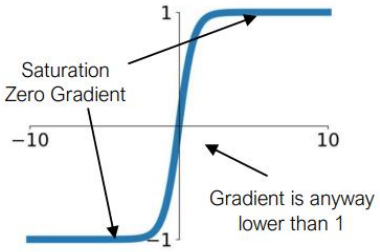
TIPS AND TRICKS IN NEURAL NETWORKS TRAINING

Better Activation functions

Activation functions such as Sigmoid or Tanh saturate

- Gradient is close to zero
- Backprop. requires gradient multiplications
- Gradient faraway from the output vanishes

→ Learning in deep networks does not happen



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

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

The gradient is the force that drives to the optimal, zero gradient means no movement. So the S shape function might slow down the training. When we are near the edges, we incur into saturation. If somehow we start near the edges the procedures will be very slow.

The way we compute the gradient of the error wrt one weight: it's a multiplication of derivatives and weights. The problem is that: if we look at the gradient of the sigmoid, we get a function witch is btw 0 and a number that is at most 1.
The maximum of the derivative of a sigmoid is when the function crosses 0.5.
We multiply multiple times a number that is below 1. The product goes exponentially fast to 0. This a problem:

→ vanishing gradient.

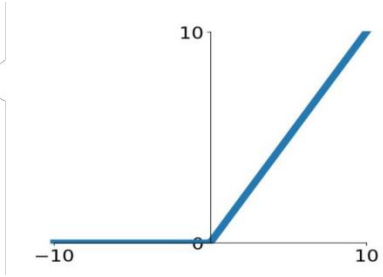
If we have multiple layers we start from a random initialisation, we compute the gradient and there is a high chance that the gradient of the first layer will be already close to 0. So the learning speed will be very low.

The same problem occurs with the tanh, as its gradient is small too.

Furthermore, as we increase the number of layers, the gradient at the input layer is very close to 0. So, the problem with back-propagation is present especially with recurrent NN (if we use sigmoid function). This problem was studied when using recurrent NN because the training was very jeopardised by the type of the activation functions.

How to solve the problem? The activation function needs to have a gradient greater than 1. But if it's greater than 1 we have the opposite problem. The multiplication will go towards infinite.
The solution is related to use a function with a gradient equal to 1
If we have gradient <1 —> vanishing gradient
If we have gradient >1 —> exploding gradient

RELU - Rectified Linear Unit



The ReLU activation function has been introduced.

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

It has several advantages:

• Faster SGD Convergence (6x w.r.t sigmoid/tanh)

• Sparse activation (only part of hidden units are activated)

• Efficient gradient propagation (no vanishing or exploding gradient problems), and Efficient computation (just thresholding at zero)

• Scale-invariant: max (0, ax) = a max (0, x)

The linear function has gradient equal to 1. But we cannot have only a linear function.
If the input is < 0 it gives 0 output
If the input if > 0 it gives the value of the input (so gradient = 1).

In general, it has been shown that there are not vanishing gradient (we multiply lots of 1) and no exploding gradient (we multiply lots of 0)

It has potential disadvantages:

• non-differentiable at zero: however it is differentiable anywhere else

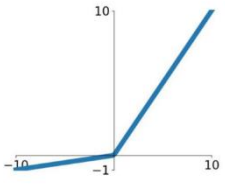
• Non-zero cantered output

• 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".

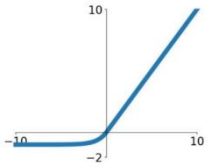
* this happens with high learning rated (= decreased model capacity)

Rectified Linear Unit (Variants)



Leaky ReLU: fix for the “dying ReLU” problem

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



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

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

Weights initialization

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

• Zeros: it does not work! All gradients would be zero, no learning will happen

• Big Numbers: bad idea, if unlucky might take very long to converge

• Gaussian distribution with zero mean and low variance, $w \sim N$ (0, σ2 = 0.01) : good for small networks, but it might be a problem for deeper neural networks

- Sigmoid or tanh: we speed up training very fast (we start from where the steep is high)
- Relu: it depends whether the activation function will be greater than 0 or lower than 0.
- Leaky relu: good anyway

In deep networks:

• If weights start too small, then gradient shrinks as it passes through each layer

• If the weights in a network start too large, then gradient grows as it passes through each layer until it's too massive to be useful (exploding gradient)

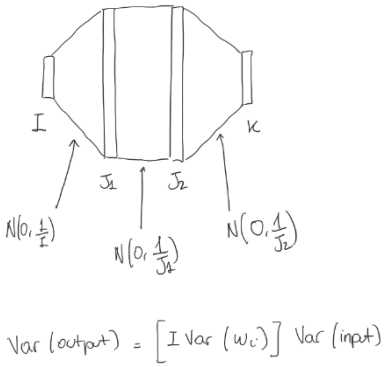
NB: this is not a theoretical issue, but numerical

Some proposal to solve this Xavier initialization or He initialization …

XAVIER INITIALIZATION

Suppose we have an input x with I components and a linear neuron with random weights w. Its output 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

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

Now if our inputs and weights both have mean 0, that simplifies to

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

If we assume all w_i and x_i are i.i.d*. we obtain

$$Var(h_j) = Var(w_{j1}x_1 + \cdots + w_{ji}x_I + \cdots + w_{jI}\,x_I) = I * Var(w_i)Var(x_i)$$

*independent and identically distributed random variables

Variance of output is the variance of the input but scaled by $\mathbf{I} * \boldsymbol{Var(w_i)}$.

If we want the variance of the input and the output to be the same

$$I * Var(w_j) = 1$$

For this reason Xavier proposes to initialize (Linear assumption seem too much, but in practice it works!)

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

Glorot & Bengio suggested a different approach: Performing similar reasoning for the they found

$$n_{out}Var(w_j) = 1$$

To accommodate the Xavier idea, they decided to work half-way. So, the variance of weights presents both Nin and Nout

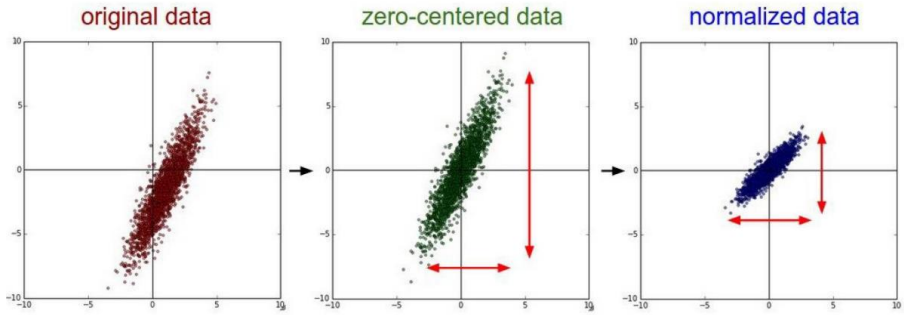
$$w \sim N\left(0, \frac{2}{n_{in}+n_{out}}\right)$$

More recently **He** proposed, for rectified linear units

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

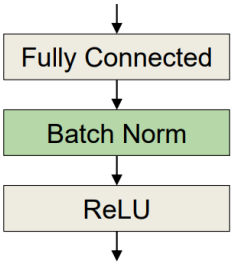
Batch Normalization

Networks converge faster if inputs have been whitened (zero mean, unit variances) and are uncorrelated to account for covariate shift



Covariate shift: phenomenon that occurs when training and test set comes from different distribution. For ex for images, the test can have more light than the training (or we can have more noise).
If we normalize the image before training (we remove the differences btw the training and the test, which are not related to task, but related to the fact that input variables have been shifted a little bit, therefore it's called covariate shift).

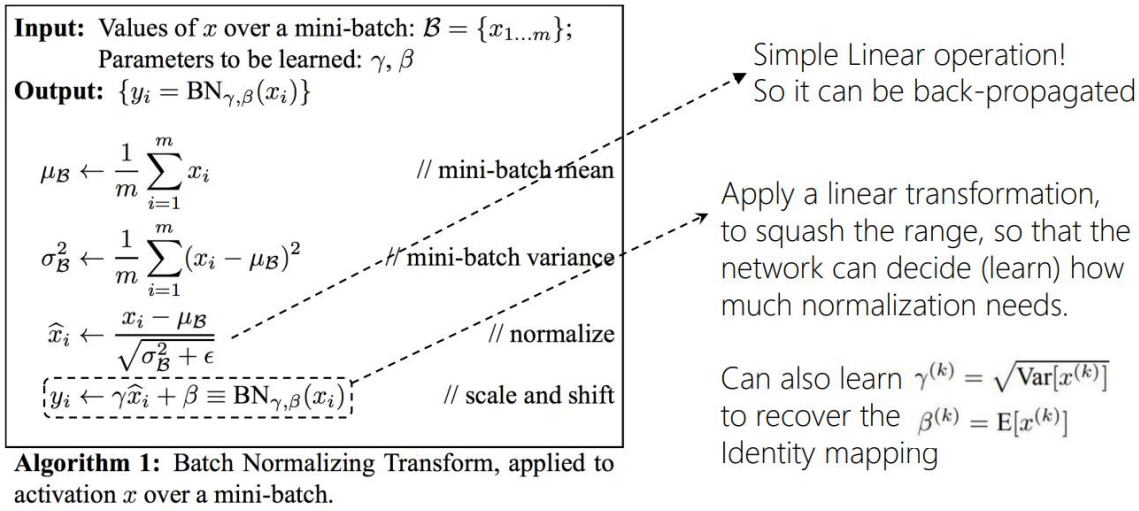
We can have internal covariate shift; normalization could be useful also at the level of hidden layers.
Batch normalization is a technique to cope with this:



- Forces activations to take values 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 pre-processing at every layer of the network, but integrated into the network itself in a differentiable way.

In practice

- Each unit's pre-activation is normalized (mean subtraction, stddev division)
- During training, mean and std are computed for each minibatch
- Backpropagation takes into account normalization
- At test time, the global mean / std are used (global statistics are estimated using training running averages)



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

We must update the mean and variance at each batch

NB!!!: batch normalisation HAS parameters (typical error in the exam) that are gamma and beta.

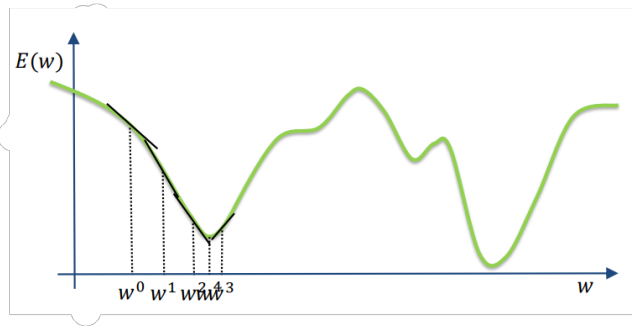
Batch Normalization has shown to

- Improve gradient flow through the network
- Allow using higher learning rates (faster learning)
- Reduce the strong dependence on weights initialization
- Act as a form of regularization slightly reducing the need for dropout

Recall about Backpropagation

Finding weighs of a Neural Network is a nonlinear minimization process

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



We iterate from an initial configuration

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

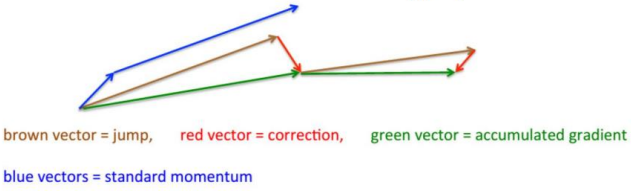
To avoid local minima can use *momentum*

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

Nesterov Accelerated gradient: make a jump as momentum, then adjust

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

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



It first applies the momentum, then it computes the gradient (where we have arrived with the momentum) then it makes a step in gradient descent

Adaptive Learning Rate

Adaptive Learning Rates Neurons in each layer learn differently

- Gradient magnitudes vary across layers
- Early layers get "vanishing gradients"
- Should ideally use separate adaptive learning rates

Several algorithm proposed:

- Resilient Propagation (Rprop – Riedmiller and Braun 1993)
- Adaptive Gradient (AdaGrad – Duchi et al. 2010)
- RMSprop (SGD + Rprop – Teileman and Hinton 2012)
- AdaDelta (Zeiler et at. 2012)
- Adam (Kingma and Ba, 2012)

IMAGE CLASSIFICATION – INTRO

COMPUTER VISION: An interdisciplinary scientific field that deals with how computers can be made to gain high-level understanding from digital images or videos.

Lately there has been a dramatic change in CV:
→ once most of techniques and algorithms build upon a mathematical/statistical description of images
→ Nowadays, ML methods are much more popular.

IMAGES

The image is actually a matrix that contains some content
Coloured images are composed by 3matrixes, each one decodes one colour (red, green, blue)
If we zoom a lot we see the pixels, each pixel has a number that is encoded in 8 bits, so in range [0, 255]

We have 3 bytes per pixel, and an image taken by our phone is 10 megapixel, so we store 30 megabytes
JPEG format is a compressed format that does 2 two types of encoding (zip + spoils some image property). This does a good job, so the images are quite redundant.
When we open an image in order to be given into a nn, we want the entire image, so it becomes huge in memory (it becomes larger than jpeg format). Therefore we work with much smaller images

LOCAL (SPATIAL) TRANSFORMATION

In general, these can be written as:

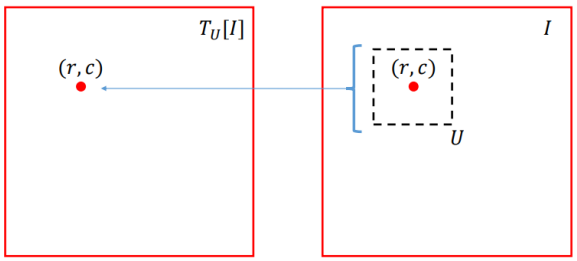
$$G(r,c) = T_U[I](r,c)$$

Where
- I input image to be transformed
- G is the output
- T_U is a function transforming the image
- U is a neighbourhood, identifies a region of the image that will concur in the output definition

T operates on I "around" U. This means that the output at pixel (r, c) i.e., T_U [I](r, c) is defined by all pixel in U in the input image

$$\{I(r+u, c+v), \qquad (u,v) \in U\}$$

Such that U is described by a set of «displacements»

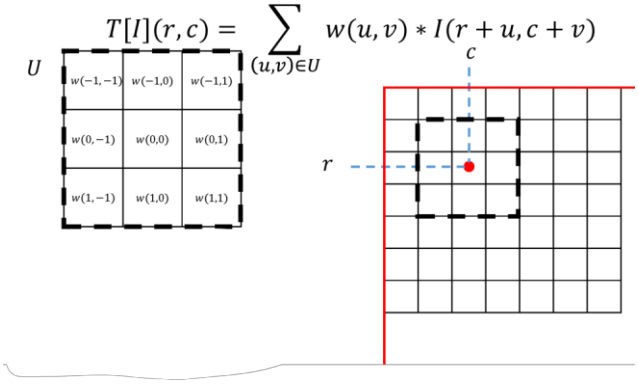


The location of the output does not change
The operation is repeated for each pixel
T can be either linear or nonlinear

Linear Spatial Filters

Linear Transformation: Linearity implies that

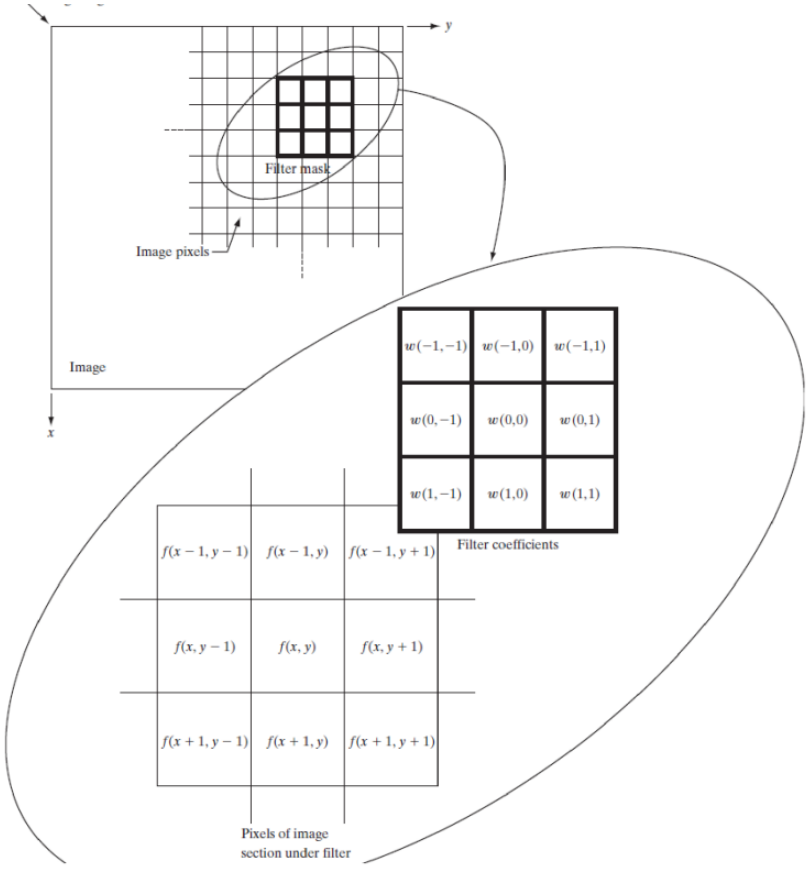
$$T[I](r,c) = \sum_{(u,v)\in U} w(u,v) * I(r+u, c+v)$$



We can consider weights as an image, or a filter w. The filter w entirely defines this operation.

This operation is repeated for each and every pixel in the input image

CONVOLUTION



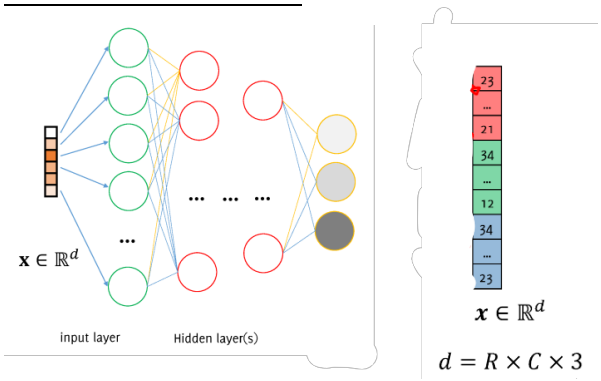
Correlation

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

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

where the filter w is of size (2ZL+1)x(2L+1)

LINEAR CLASSIFIERS

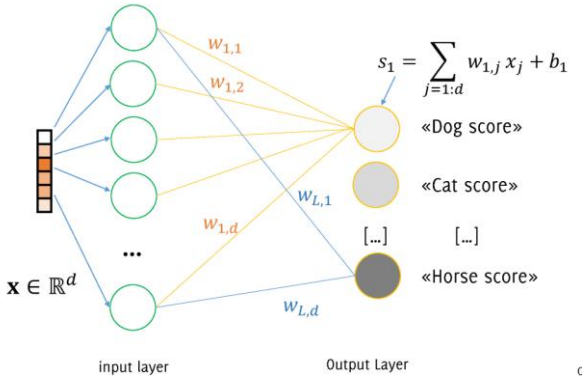


We can unfold the image into a vector (flattering).
We unfold in column by column → COLUMN-WISE UNFOLDING

$w_{i,j}$ is the weight associated to the i-th neuron of the input when computing the value at the j-th output neuron

EXAMPLE: - CIFAR10 images

1-Layer NN to Classify images



model.summary();

Layer (type)	Output Shape	Param #
Input (InputLayer)	[(None, 32, 32, 3)]	0
Flatten (Flatten)	(None, 3072)	0
Output (Dense)	(None, 10)	30730

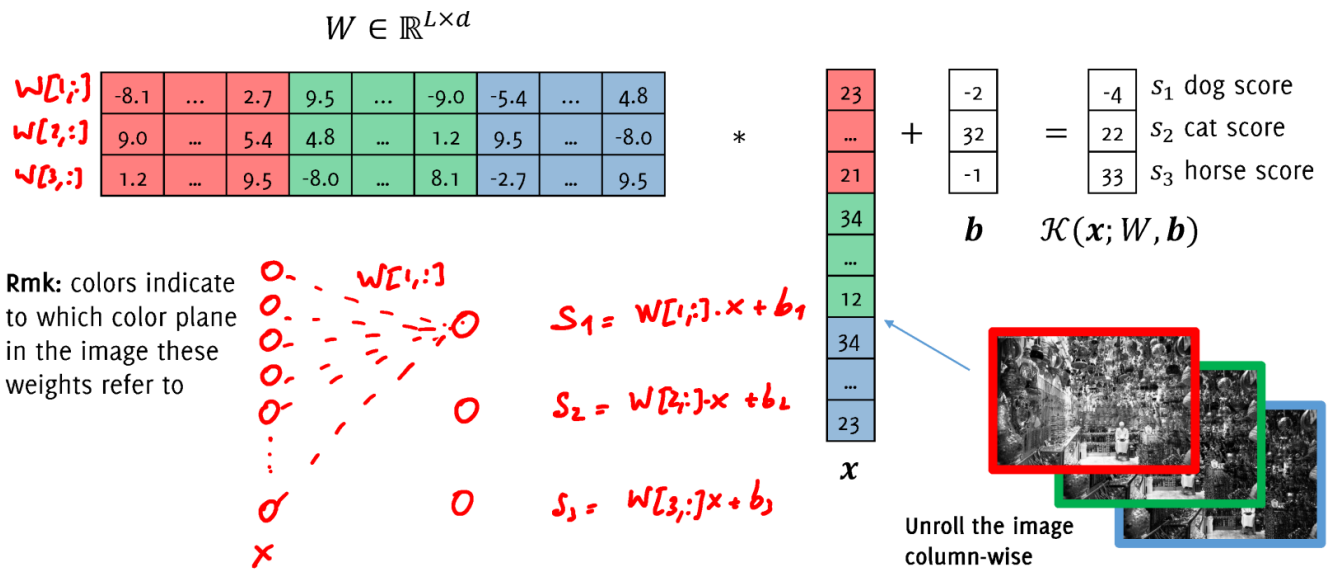
Total params: 30,730
Trainable params: 30,730
Non-trainable params: 0

Why don't we take a larger network?
Dimensionality prevents us from using in a straightforward manner deep NN as those seen so far.
Let's take a network with a hidden layer having half of the neurons of the input layer.
On CIFAR10 images, the-number of neurons would be:
3072 first layer, 1536 second layer, 10 output layer
→ 1536 * 3072 + 1536 = 4,720,128 and 10* 1536 + 10 = 15370 paramters

Considering 1-layer network:
We can arrange weights in a matrix $W \in \mathbb{R}^{Lxd}$, then the scores of the i-th class is given by inner product between the matrix rows W|i,:] and x. Scores then becomes:

$$s_i = W[i,:] * \boldsymbol{x} + b_i$$

NB: Non linearity is not needed here since there are no layers following.
We can also ignore the softmax in the output since this would not change the order of the scores (would just normalize them)



Training the nn is equivalent to train a linear classifier

Why nonlinear layers?
Each layer in a NN can be seen as matrix multiplication (+ bias).

$$\mathbf{s} = W\boldsymbol{x} + \boldsymbol{b}$$

If we stack 3 layers without activations:

$$\mathbf{s} = \big((W_1\boldsymbol{x} + \boldsymbol{b}_1)\mathrm{W}_2 + \boldsymbol{b}_2\big)W_3 + b_3$$

This becomes equivalent to

$$\mathbf{s} = W\boldsymbol{x} + \boldsymbol{b}$$

This is a further confirmation why it becomes pointless to stack many layers without including a nonlinear activations...

TRAINING A CLASSIFIER

Given a training set TR and a loss function, define the parameters that minimize the loss function over the whole TR. In case of linear classifier

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

Solving this minimization problem provides the weights of our classifier
Loss function: a function L that measures our unhappiness with the score assigned to training images.
The loss L will be high on a training image that is not correctly classifier, low otherwise.

Loss function can be minimized by gradient descent and all its variants.

The loss function has to be typically regularized to achieve a unique solution satisfying some desired property

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

being $\lambda > 0$ a parameter balancing the two terms

The training data is used to learn the parameters W, b. The classifier is expected to assign to the correct class a score that is larger than that assigned to the incorrect classes. Once the training is completed, it is possible to discard the whole training set and keep only the learned parameters.

Geometric interpretation of a Linear Classifier

W[i,:] is a d —dimensional vector containing the weights of the score function for the i-th class.
Computing the score function for the i-th class corresponds to computing the inner product (and summing the bias): W[i, :] *x+ bi;
Thus, the NN computes the inner products against L different weights vectors, and selects the one yielding the largest score (up to bias correction).
Rmk: these "inner product classifiers"" operate independently, and the output of the j-th row is not influenced weights at a different row
Rmk: this would not be the case if the network had hidden layer that would mix the outputs of intermediate layers

Interpret each image as a point in $\mathbb{R}^d$. Each classifier is a weighted sum of pixels, which corresponds to a linear function in $\mathbb{R}^d$. In $\mathbb{R}^2$ these would be

$$f([x1,x2]) = w1x1 + w2x2 + b$$

Then, points [x1, x2] yelding

$$f([x1, x2]) = 0$$

would be lines.
→ Thus, in $\mathbb{R}^2$ the region that separates positive from negative scores for each class is a line. This region becomes an hyperplane in $\mathbb{R}^d$

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
→ **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.

PROBLEM DEFINITION

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.

NEAREST NEIGHBOURHOOD CLASSIFIER

Nearest Neighbourhood

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

$$\hat{y}_j = y_{j^*}, \qquad \text{being} \quad j^* = \underset{i=1\dots N}{\operatorname{argmin}}\, d(\boldsymbol{x}_j, \boldsymbol{x}_i)$$

Distances are typically measured as

$$d(\boldsymbol{x}_j, \boldsymbol{x}_i) = \|\boldsymbol{x}_j - \boldsymbol{x}_i\|_2 = \sqrt{\sum_k \left([\boldsymbol{x}_j]_k - [\boldsymbol{x}_i]_k\right)^2}$$

$$d(\boldsymbol{x}_j, \boldsymbol{x}_i) = |\boldsymbol{x}_j - \boldsymbol{x}_i| = \sum_k \left|[\boldsymbol{x}_j]_k - [\boldsymbol{x}_i]_k\right|$$

The distance is between two vectors, so basically you are computing a pixel-wise similarity.

K-Nearest Neighbourhood

Assign to a each test image, the most frequent label among the K —closest images in the training set
where Uk(x;) contains the K closest training images to x;

$$\hat{y}_j = y_{j^*}, \qquad \text{being} \quad j^* \text{ the mode of } \mathcal{U}_K(\boldsymbol{x}_j)$$

NB: setting the parameter K and the distance measure is an issue

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:

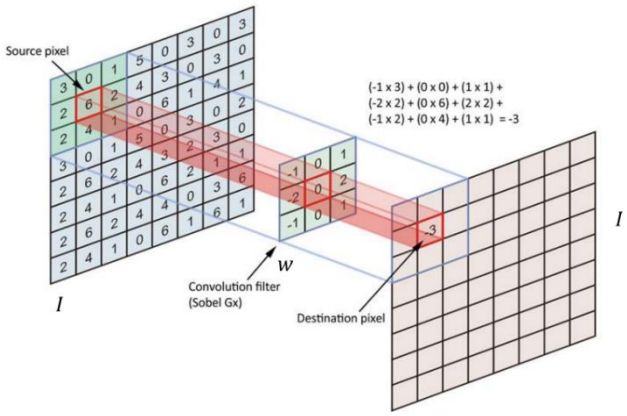
- Computationally demanding at test time
- Large training set have to be stored in memory
- Rarely practical on images: distances on high-dimensional objects are difficult to interpret

CONVOLUTIONAL NEURAL NETWORKS

CONVOLUTION

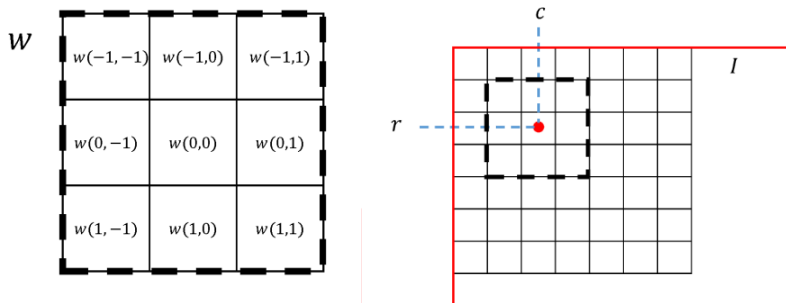
Convolution is a linear transformation. Linearity implies that:

$$(I \otimes w)(r,c) = \sum_{(u,v)\in U} w(u,v)\, I(r+u, c+v)$$



We can consider weights as a filter h The filter h entirely defines convolution.

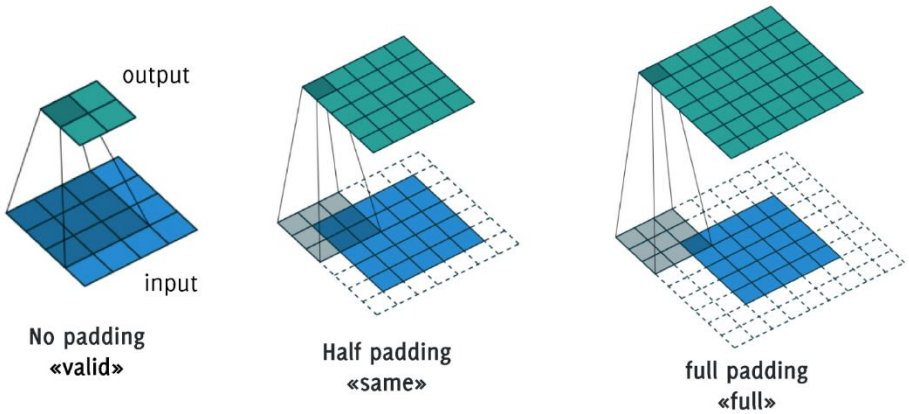
Convolution operates the same in each pixel



The same operation is being performed in each pixel of the input image. This is equivalent to 2D Correlation up to a "flip" in the filter w. Convolution is defined up to the "filter flip" for the Fourier Theorem to apply.
Filter flip has to be taken into account when computing convolution in Fourier domain and when designing filters. However, in CNN, convolutional filters are being learned from data, thus it is only important to use these in a consistent way. → In practice, in CNN arithmetic there is no flip!

PADDING - How to define convolution output close to image boundaries?
Padding with zero is the most frequent option, as this does not change the output size. However, no padding or symmetric padding are also viable options

Rmk: Blue maps are inputs, and cyan maps the outputs. the filter here is 3 x 3



→ Padding can reduce or increase the dimension of the output wrt the input.

THE FEATURE EXTRACTION PERSPECTIVE

Images cannot be directly fed to a classifier. We need some intermediate step to:

- Extract meaningful information (to our understanding)
- Reduce data-dimension

We need to extract features: → The better our features, the better the classifier

Hand-Crafted Features

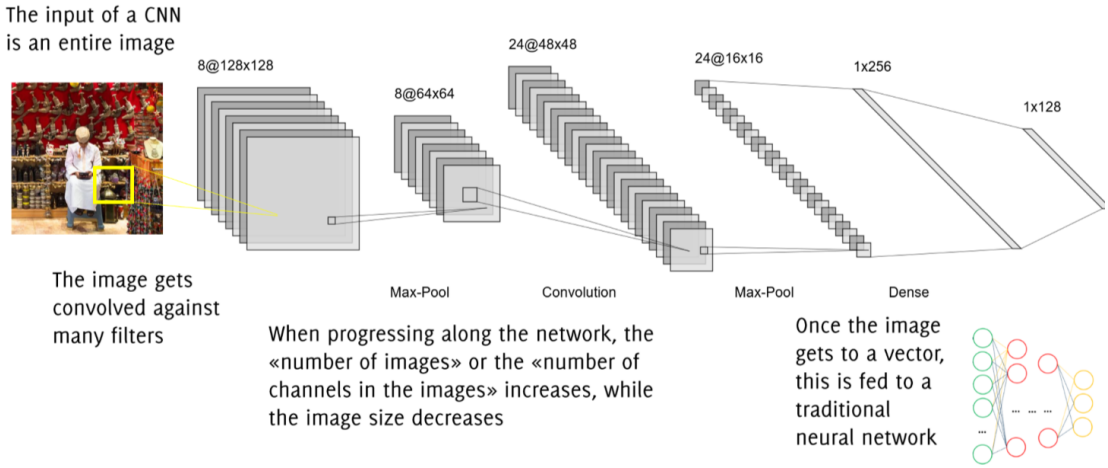
PROS	CONS
• Exploit a priori / expert information • Features are interpretable (you might understand why they are not working) • You can adjust features to improve your performance • Limited amount of training data needed • You can give more relevance to some features	• Requires a lot of design/programming efforts • Not viable in many visual recognition tasks (e.g. on natural images) which are easily performed by humans • Risk of overfitting the training set used in the design • Not very general and "portable"

DATA DRIVEN FEATURES – CONVOLUTIONAL NEURAL NETWORKS

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.

CNN - STRUCTURE

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



Channel= like RGB for colors

CNN are typically made of blocks that include:

- Convolutional layers
- Nonlinearities (activation functions)
- Pooling Layers (Subsampling / maxpooling)

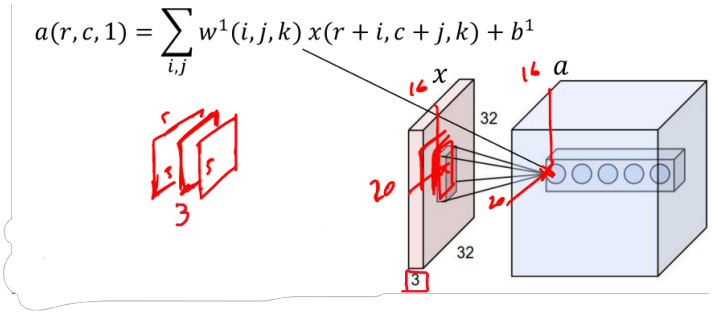
An image passing through a CNN is transformed in a sequence of volumes.
As the depth increases, the height and width of the volume decreases
Each layer takes as input and returns a volume

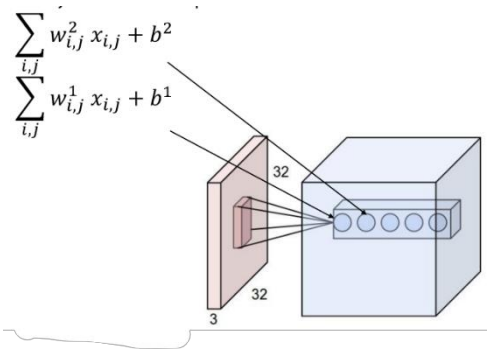
CONVOLUTIONAL LAYERS

Convolutional layers "mix" all the input components
The output is a linear combination of all the values in a region of the input, considering all the channels

The parameters of this layer are called filters. The same filter is used through the whole spatial extent of the input.
NB: Filters need to have the same number of channels as the input, to process all the values from the input layer

The contribution of the 3 filters will produce the ouput in the first channel of the following layer.
Different filters yield different layers in the output.





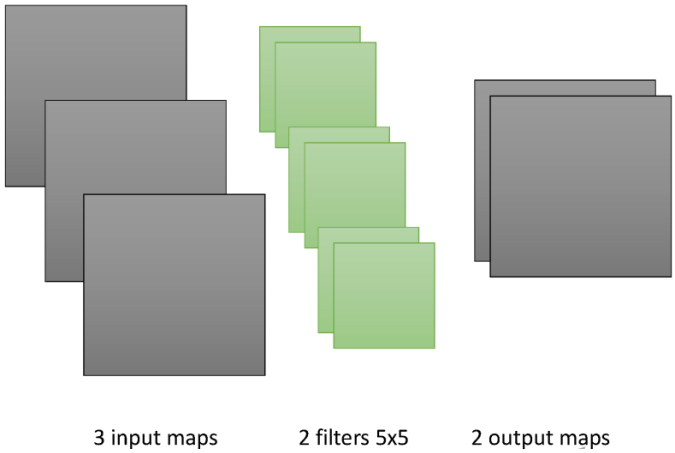
RECAP:

- Convolutional Layers are described by a set of filters
- Filters represent the weights of these linear combination.
- Filters have very small spatial extent and large depth extent,
- The filter depth is typically not specified, as it corresponds to the number of layers of the input volume
- The output of the convolution against a filter becomes a slice in the volume feed to the next layer
- Convolutional layers "mix" all the input components
    - The output is also called volume or activation maps
    - Each filter yields a different slice of the output volume
    - Each filter has depth equal to the depth of the input volume

CNN ARITHMETIC

Example

Hp: half-padding



Each filter has depth = 3.

The output is related to the number of filters, in this case we have 2 outputs because we have 2 filters.

How many parameters?

NB: THERE IS ONE BIAS FOR EACH FILTER
So here there are 2 biases.

2 filters 5x5x3 → 5x5x3 + 2 = 152 trainable parameters (weights)

ACTIVATION LAYERS

Introduce nonlinearities in the network, otherwise the CNN might be equivalent to a linear classifier...
Activation functions are scalar functions, namely they operate on each single value of the volume (they don't change volume size).

RELU (Rectifier Linear Units): it's a thresholding on the feature maps, i.e., a max(0,-) operator.

- The most popular activation function in deep NN (since when it has been used in AlexNet)
- Dying neuron problem: a few neurons become insensitive to the input (vanishing grandient problem)

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

NB: Images are normalised, so the image have negative values too (it's better to have all the values around 0)

LEAKY RELU: like the relu but include a small slope for negative values

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

TANH (hyperbolic Tangent): has a range (-1,1), continuous and differentiable

$$T(x) = \frac{2}{1 + e^{-2x}} - 1$$

SIGMOID: has a range (0,1), continuous and differentiable

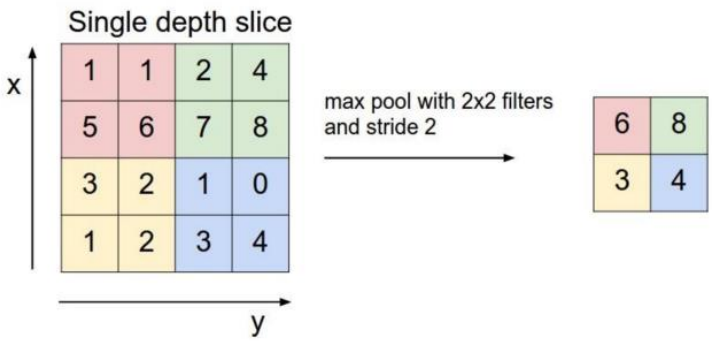
$$S(x) = \frac{1}{1 + e^{-2x}}$$

These last two activation functions are mostly popular in MLP architectures

NB: AFTER EACH CONVOLUTION THERE IS A NON-LINEAR ACTIVATION FUNCTION!!!

POOLING LAYERS

Pooling Layers reduce the spatial size of the volume.
The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, often using the MAX operation.

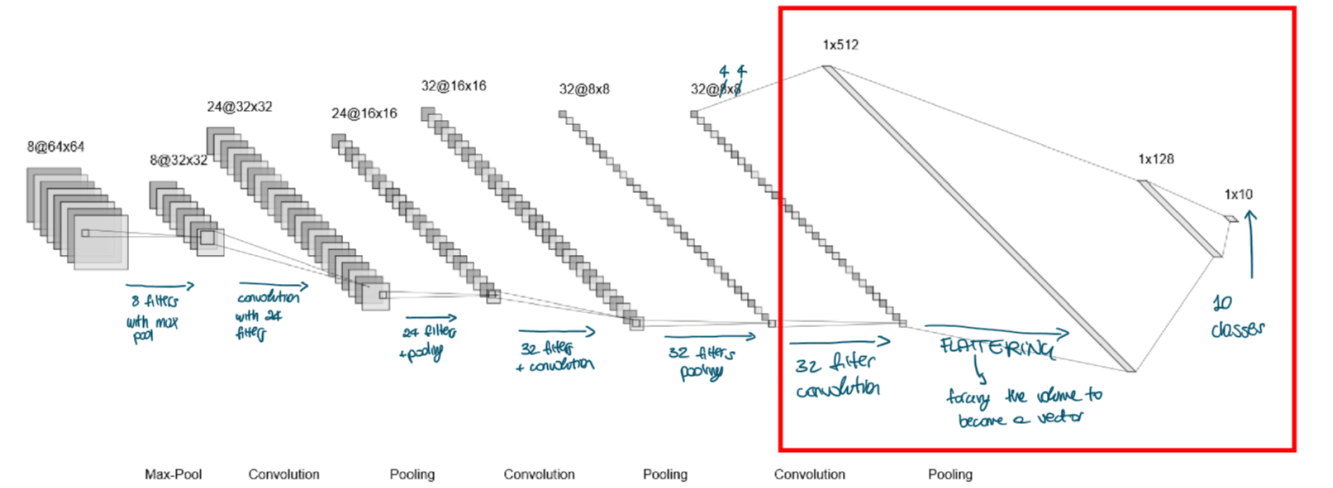


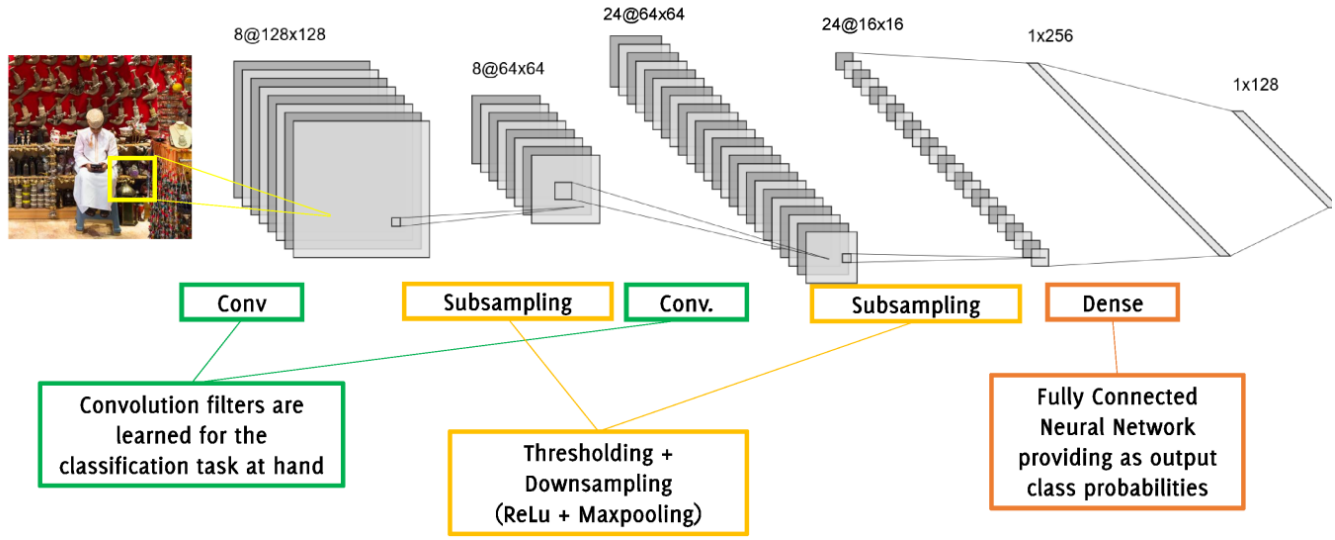
The stride is the size of the step we make for applying filters
→ Like if we compute the filter for every pixel of the image or if we skip some. Like in the example we consider a pixel every two)

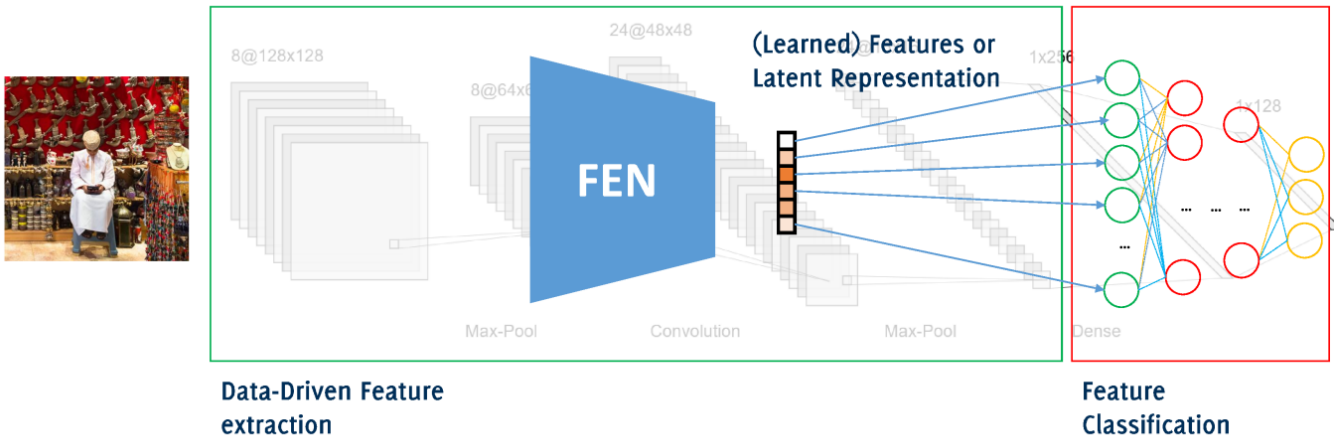
DENSE LAYERS

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.
Here the spatial dimension is lost, the CNN stacks hidden layers from a MLP NN. It is called Dense as each output neuron is connected to each input neuron.

The output of the fully connected (FC) layer has the same size as the number of classes, and provides a score for the input image to belong to each class.





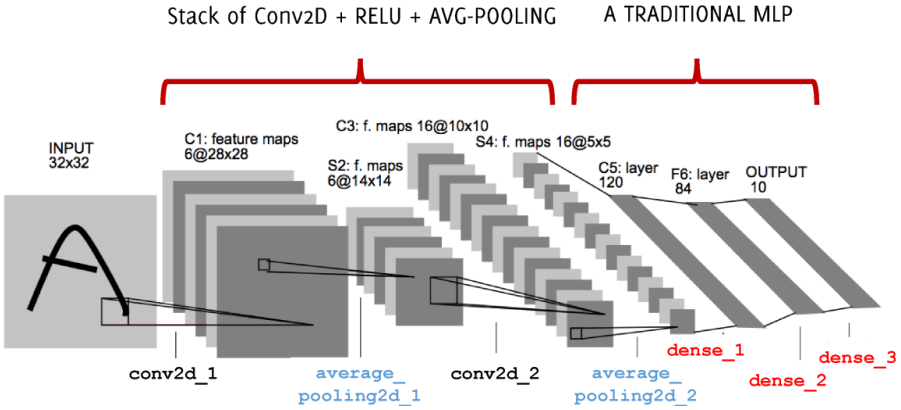


THE FIRST CNN

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.

LeNet-5 (1998)



The First CNN

Do not use each pixel as a separate input of a large MLP, because: images are highly spatially correlated, using individual pixel of the image as separate input features would not take advantage of these correlations.

The first convolutional layer: 6 filters 5x5

The second convolutional layer: 16 filters 5x5

Layer (type)	Output Shape	Param #	
conv2d_1 (Conv2D)	(None, 28, 28, 6)	156 (6 x 5 x 5 + 6)	Input is a grayscale image
average_pooling2d_1 (Average	(None, 14, 14, 6)	0	
conv2d_2 (Conv2D)	(None, 10, 10, 16)	2416 (16 x 5 x 5 x 6 + 16)	
average_pooling2d_2 (Average	(None, 5, 5, 16)	0	The input is a volume having depth = 6
flatten_1 (Flatten)	(None, 400)	0	
dense_1 (Dense)	(None, 120)	48120	Most parameters are still in the MLP
dense_2 (Dense)	(None, 84)	10164	
dense_3 (Dense)	(None, 10)	850	

Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0

Most of parameters are in MLP

What about a MLP taking as input the whole image?

Input 32 x 32 = 1024 pixels, fed to a 84 neurons (the last FC layers of the network)

→ 86950 parameters: 1024 * 84 + 84 + 84 * 10 + 10

But… If you take an RGB input: 32 x 32 x 3,

CNN: only the nr. of parameters in the filters at the first layer increases

156 + 61550 → 456+ 61550

(6 X 5 X 5)→(6 X 5 X 5 X 3)

MLP: only the first layer increases the # of parameters by a factor 3

1024 x 84 → 1024 x 84 x 3

NON SO SE IL RESTO DELLA LEZ 2 E' STATO FATTO!

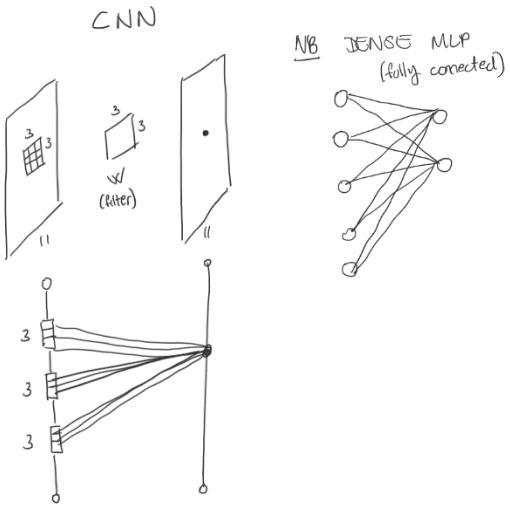
CNN PARAMTERS

Convolution is a linear operation! Therefore, if you unroll the input image to a vector, you can consider convolution weights as the weights of a Multilayer Perceptron Network!

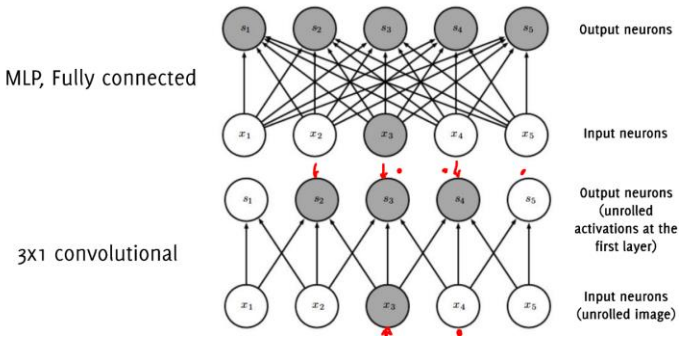
$$a(x,y,k) = \sum_{u,v,z} w_k(u,v,z)\, I(x-u, y-v, z) + b_k$$

inputs

parameters



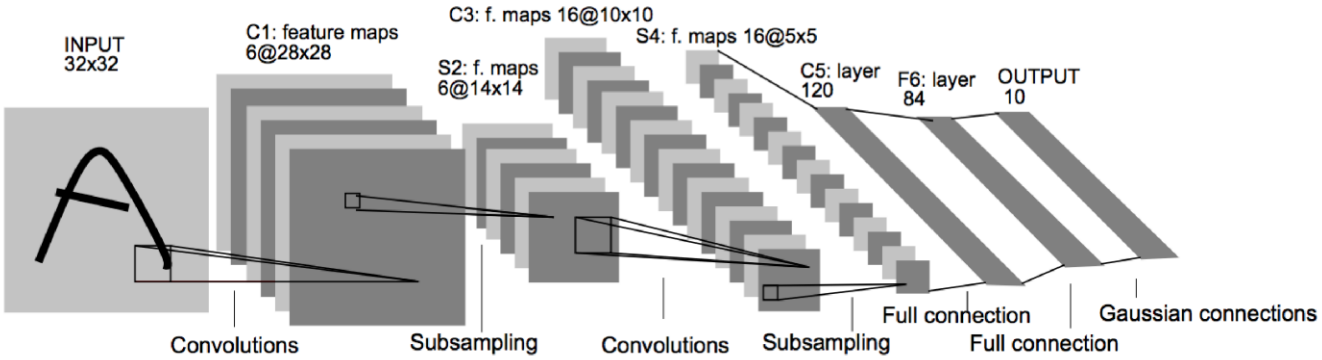
BUT the main difference between CNN and MLP is that CNN has sparse connectivity and MLP is fully connected



WEIGHT SHARING – SPATIAL INVARIANCE

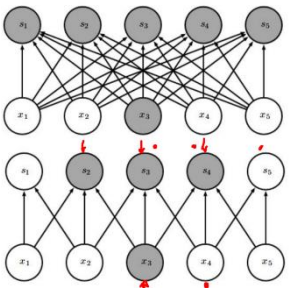
In a CNN, all the neurons in the same slice of a feature map use the same weights and bias: this reduces the nr. of parameters in the CNN.

Underlying assumption: if one feature is useful to compute at some spatial position (x, y), then it should also be useful to compute at a different position (x2, y2)



If the first layer was a MLP:

- MLP layer: this should have had 28 x 28 x 6 neurons in the output
- MLP layer with sparse connectivity: only 5x5 nonzero weights each neuron
- MLP layer: 28 x 28 x 6 x 25 weights + 28 x 28 x 6 biases (122 304)
- Conv layer: 25 weights + 6 biases shared among neurons of the same layer

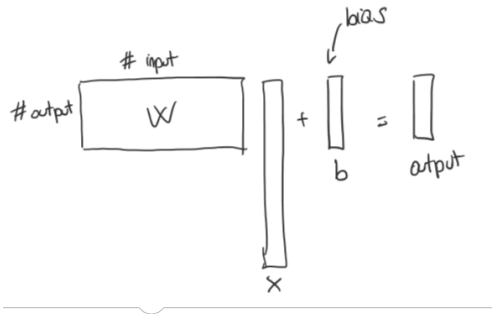


For ex:
In fully connected network: 5x5 weights + 5 biases
In convolutional network: 3x1 + bias

To Summarize

Any CONV layer can be implemented by a FC layer performing exactly the same computations.

The weight matrix W of the FC layer would be
- a large matrix (#rows equal to the number of output neurons, #cols equal to the nr of input neurons).
- That is mostly zero except for at certain blocks where the local connectivity takes place (sparse matrix).
- The weights in many of the blocks are equal due to parameter sharing.



NB: we will see that the converse interpretation (FC as conv) is also viable and very useful!

RECEPTIVE FIELD

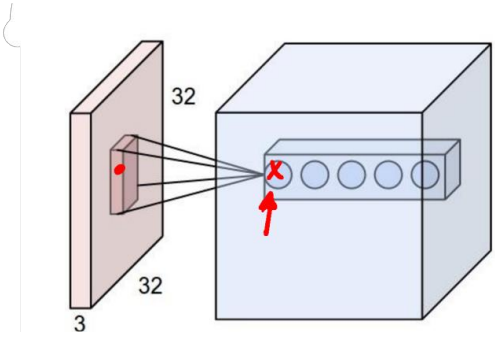
One of the basic concepts in deep CNNs.

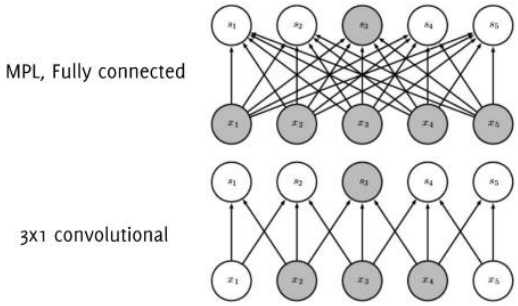
Due to sparse connectivity, unlike in FC networks where the value of each output depends on the entire input, in CNN each output only depends on a specific region in the input.

This region in the input is the **receptive field** for that output

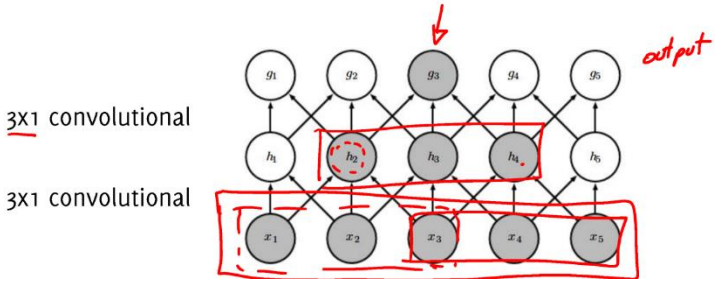
The deeper you go, the wider the receptive field is: maxpooling, convolutions and stride > 1 increase the receptive field

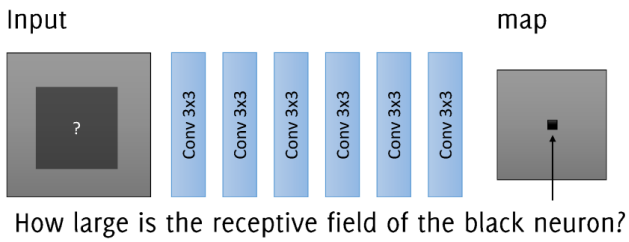
Usually, the receptive field refers to the final output unit of the network in relation to the network input, but the same definition holds for intermediate volumes



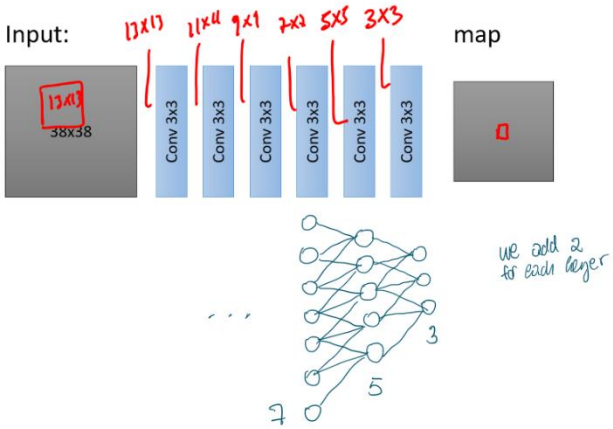


Deeper neurons depend on wider patches of the input (convolution is enough to increase receptive field, no need of maxpooling).





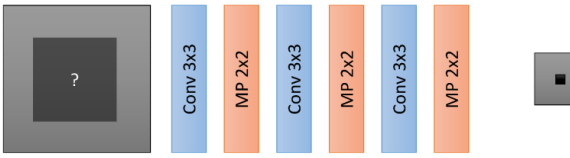
How large is the receptive field of the black neuron?

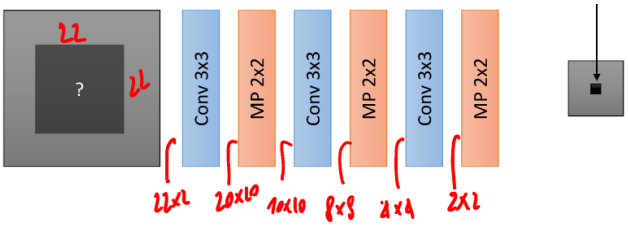


The deeper we go in the network the higher the receptive field → we add 2 for every layer.

Another example:

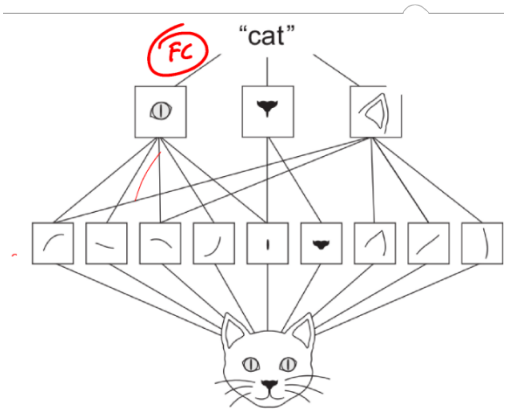
How large is the receptive field of the black neuron?





As we move to deeper layers:
spatial resolution is reduced
the number of maps increases

We search for higher-level patterns, and don't care too much about their exact location.
There are more high-level patterns than low-level details!

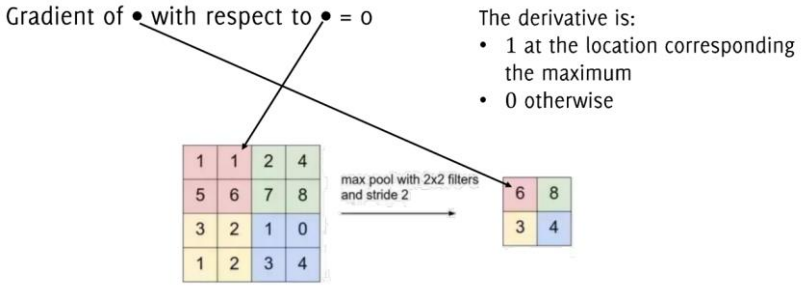


CNN TRAINING

- Each CNN can be seen as a MLP, with sparse and shared connectivities
- CNN can be in principle trained by gradient descent to minimize a loss function over a batch (e.g. binary cross-entropy, RMSE, Hinge loss.)
- Gradient can be computed by backpropagation (chain rule) as long as we can derive each layer of the CNN
- Weight sharing needs to be taken into account (fewer parameters to be used in the derivatives) while computing derivatives
- There are just a few details missing...

Backprop with max pooling

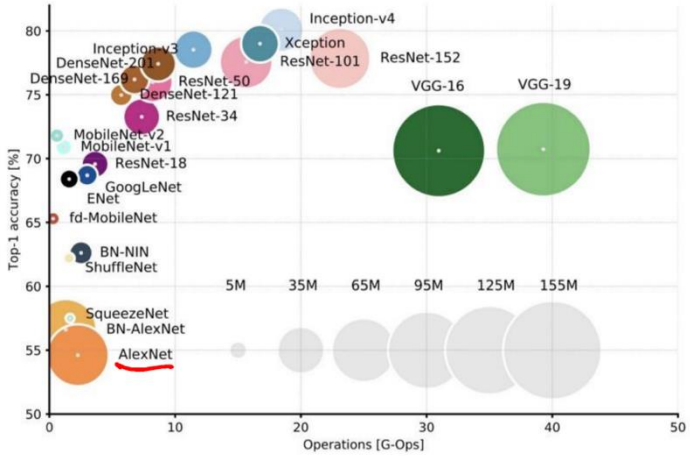
The gradient is only routed through the input pixel that contributes to the output value; e.g.:



DATA SCARCITY

- Training a CNN with Limited Amount of Data.

Deep learning models are very data hungry. Networks such as AlexNet have been trained on ImageNet datasets containing tens of thousands of images over hundreds of classes. This is necessary to define millions of parameters characterizing these networks.



Deep learning models are very data hungry. .. watch out: each image in the training set have to be annotated! How to train a deep learning model with a few training images?

→ Data augmentation

→ Transfer Learning

DATA AUGMENTATION

Data augmentation is typically performed by means of

Geometric Transformations:
- Shifts /Rotation/Affine/perspective distortions
- Shear
- Scaling
- Flip

Photometric Transformations:
- Adding noise
- Modifying average intensity
- Superimposing other images
- Modifying image contrast

Augmented versions should preserve the input label

e.g. if size/orientation is a key information to determine the output target (either the class or the value in case of regression), wisely consider scaling/rotation as transformation

NB: Augmentation is meant to promote network invariance w.r.t. transformation used for augmentation

Given an annotated image (I, y) and a set of augmentation transformations $\{\mathrm{A_l}\}_\mathrm{l}$ we train the network using these pairs.

$$\{(A_l(I), y)_l\}_l$$

- Training including augmentation reduces the risk of overfitting, as it significantly increase the training set size.
- Data augmentation can be used to compensate for class imbalance in the training set, by creating more realistic examples from the minority class
- Transformations used in data-augmentation $\{\mathrm{A_l}\}_\mathrm{l}$ can be also class-specific, in order to preserve the image label

Through data augmentation we train the network to «become invariant» to selected transformations. Since the same label is associated to I and $A_l(I)$.
Unfortunately, invariance might not be always achieved in practice.

Test Time Augmentation (TTA) or Self-ensembling

Even if the CNN is trained using augmentation, it won't achieve perfect invariance w.r.t. considered transformations.
Test time augmentation (TTA): augmentation can be also performed at test time to improve prediction accuracy.

- Perform random augmentation of each test image
  $\{A_l(I)\}_l$
- Average the predictions of each augmented image
  $\boldsymbol{p_l} = CNN(A_l(I))$
- Take the average vector of posterior for defining the final guess
  $\text{e.g.}\, \boldsymbol{p} = Avg(\{\boldsymbol{p}_l\}_l)$

NB: Test Time Augmentation is particularly useful for test images where the model is pretty unsure.

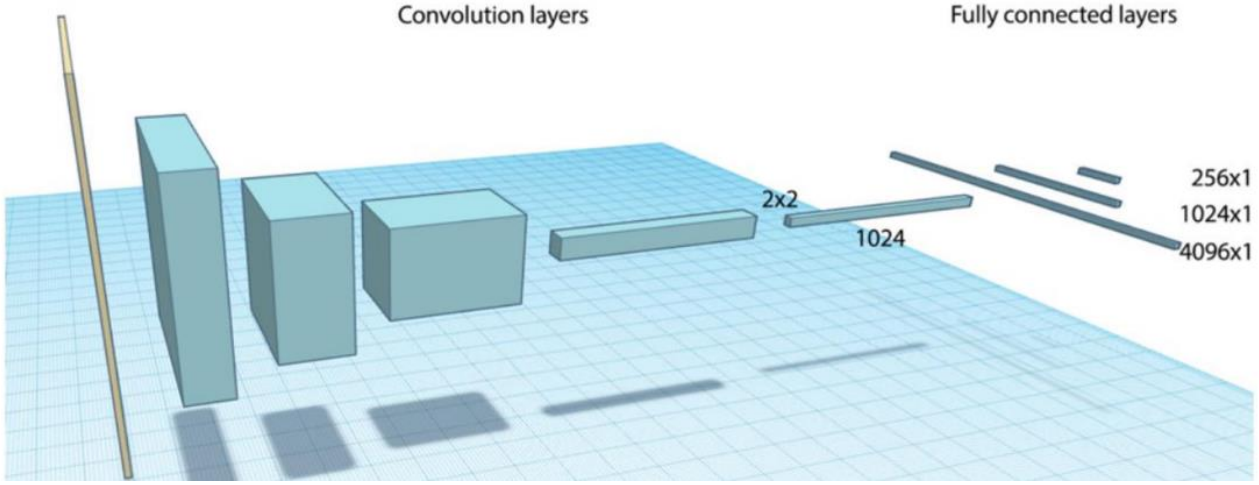
TRANSFER LEARNING

If we consider the CNN as a feature extractor, that takes the image and returns a vector of features, this vector is a very good representation of the image.
Distances in the latent representation of a CNN are meaningful for image semantics, in contrast with the Euclidean Distance (between images, which is not meaningful).

The latent representation, the input of the fully connected layer can be seen as a data-driven descriptor of the input image, i.e. a feature vector.
These features are: - Defined to maximize classification performance
- Trained via backpropagation as the NN they are fed

As we move to deeper layers:

- The spatial resolution decreases: these layers search for higher-level patterns, and don't care too much about their exact location.
- The number of maps increases → there are more high-level patterns than low-level details!



Feature Extraction Network: take the convolutional layers of a network trained to solve a large classification problem (e.g., 1000 classes from Imagenet) This should be a very effective and general feature extractor!
The FC layers are instead very custom and task-specific, as they are meant to solve the specific classification task at hand.

TRANSFER LEARNING:

- Take a successful pre-trained model
- Remove and the FC layers. Design new FC layers to match the new problem.
- «Freeze» the weights in the convolutional layers.
- Train the whole network on the new training data

NB: often the features extracted by the CNN can be useful even if the CNN is fed with images that it has never seen before
But what about the input size? The input size (of the image) can be different from the one fixed for the CNN, simply the network will provide a larger vector of feature. We change the FC part, so it can still work fine. BUT this depends on the difference in the size of the input. If the difference is very big, this might not work.

TRANSFER LEARNING VS FINE TUNING DIFFERENT OPTIONS:
Transfer Learning: only the FC layers are being trained. A good option when little training data are provided and the pre-trained model is expected to match the problem at hand
Fine tuning: the whole CNN is retrained, but the convolutional layers are initialized to the pre-trained model. A good option when enough training data are provided or when the pre-trained model is not expected to match the problem at hand. Typically, for the same optimizer, lower learning rates are used when performing fine tuning than when training from scratches

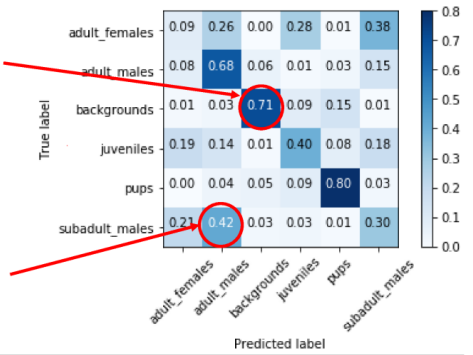
PERFORMANCE MEASURES

CONFUSION MATRIX

The element $C(i,j)$ i.e. at the i-th row and j-th column corresponds to the percentage of elements belonging to class i classified as elements of class j

71% of background patches have been correctly classified as background

42% of sub-adult males patches have been wrongly classified as adult-males



BINARY CLASSIFICATION – ROC

Background:

In a two-class classification problem (binary classification), the **CNN output is equivalent to a scalar,** since

$$CNN(I) = [p, 1-p]$$

being p the probability of I to belong to the first class.

Thus we can write

$$CNN(I) = p$$

Then, we can decide that I belongs to the first class when

$$CNN(I) > \Gamma$$

and use Γ different from 0.5, which is the standard.

We require stronger evidence before claiming I **belongs to class 1.**

Changing Γ **establishes a trade off between FPR and TPR.**

We can use a threshold to convert the probability into the class 0 or 1. When changing the threshold (gamma) we change the confusion matrix (we change the confidence that a sample belongs to a class or another).

Classification performance in case of **binary classifiers** can be also measured in terms of the **ROC** (receiver operating characteristic) **curve,** which does not depend on the threshold you set for each class

This is useful in case you plan to modify this and not use 0.5

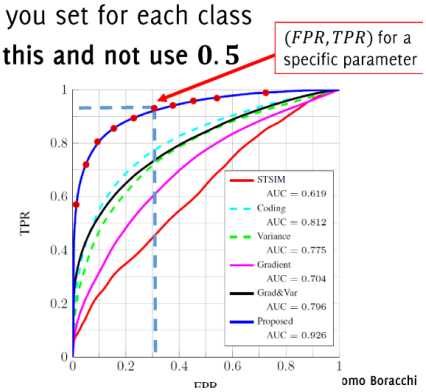
The ideal detector would achieve:

- $FPR = 0\%$,
- $TPR = 100\%$

Thus, the closer to (0,1) the better

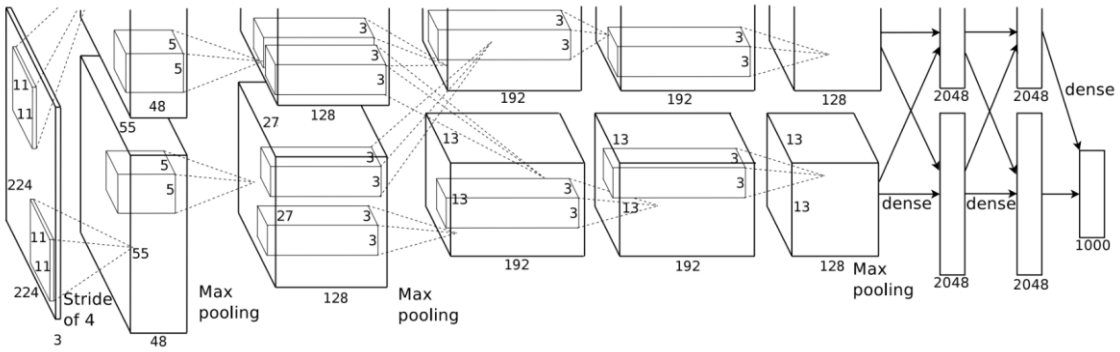
The largest the **Area Under the Curve** (AUC), the better

The optimal parameter is the one yielding the point closest to (0,1)



CNN POPULAR ARCHITECTURES

AlexNet (2012)



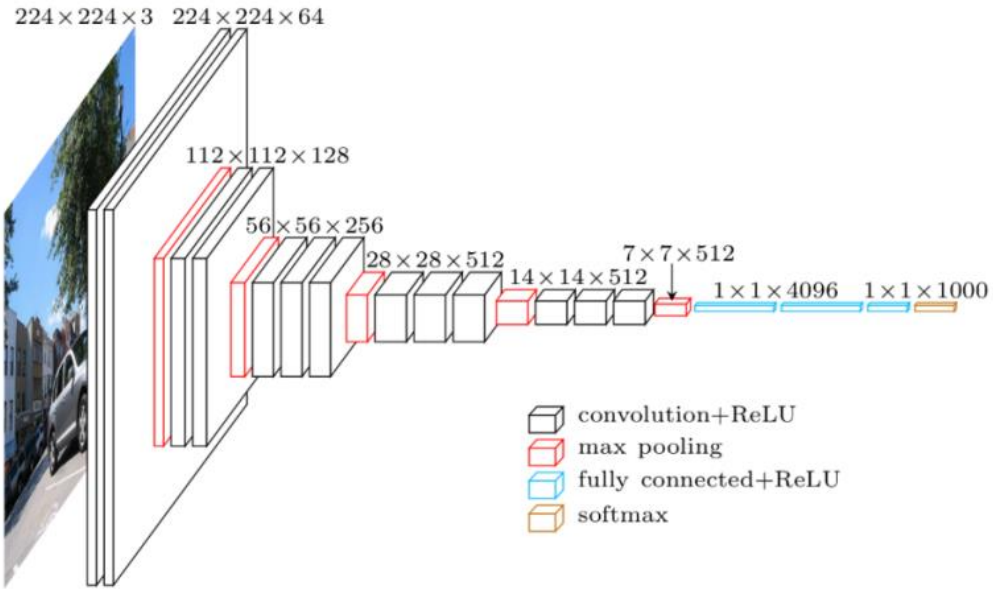
5 convolutional layer, 3 MLP, Input size 224x224x3
Parameters: 60 million [Conv: 6%, FC: 94%]

To counteract overfitting, they introduced:
- RELU
- Dropout (0.5), weight decay and norm layers (not used anymore)
- Maxpooling

The first conv layer has 96 11x 11 filters, stride 4.
The output are two volumes of 55 x 55 x 48 separated over two GTX 580 GPUs (1.5GB each GPU, 90 epochs, 5/6 days to train). NB: It is split in two because one GPU was not enough (only a memory issue)

Most connections are among feature maps of the same GPU, which will be mixed at the last layer.
At the end they also trained an ensemble of 7 models to drop error: 18.2%->15.4%

VGG16 (2014)



The VGG16, introduced in 2014, is a deeper Variant of the AlexNet convolutional structure. Smaller filters are used, and network is deeper.
Parameters: 138 million [Conv:11%, FC:89%]

The idea is to use multiple 3x3 convolutions in sequence to obtain wider receptive fields with fewer parameters and more non-linearity rather than using wider ones in a single layer.

	3 layers 3x3	1 layer 7x7
Receptive field	7x7	7x7
Nr of parameters	3 x 3 x 3 = 27	49
Nr of nonlinearities	3	1

High memory request, about 100MB per image (224 x 224 x 3) to be stored in all the activation maps, only for the forward pass. During training, with the backward pass it's about twice.

Network in Network

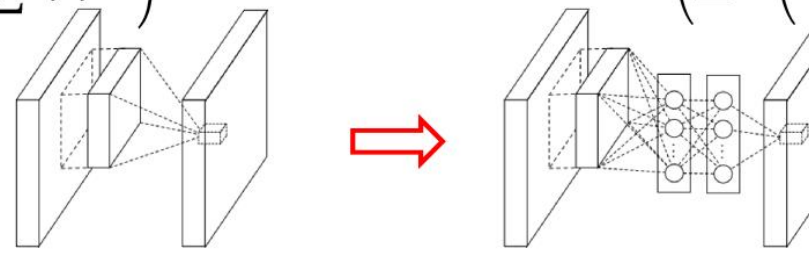
Network in Network is a paper that proposed two ideas: Mlpconv (didn't work very well) and GAP

MIpconv layers: instead of conv layers, use a sequence of FC + RELU

- Uses a stack of FC layers followed by RELU in a sliding manner on the entire image. This corresponds to MLP networks used convolutionally →nested multiple combinations
- Each layer features a more powerful functional approximation than a convolutional layer which is just linear + RELU

$$f = ReLU\left(\sum w_i x_i + b\right)$$

$$f = ReLU\left(\sum w_i^1\left(ReLU\left(\sum w_i^2 x_i + b^2\right)\right) + b^1\right)$$

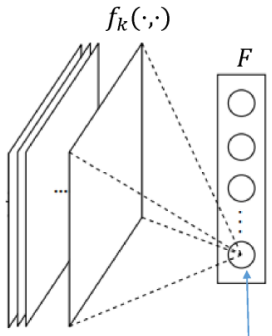


(a) Linear convolution layer (b) Mlpconv layer

GAP

Global Averaging Pooling (GAP) layers 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 (consider the input unrolled layer-wise in a vector).
- The transformation of each layer in MLP corresponds to a dense matrix.



$$F_k = \frac{1}{N}\sum_{(x,y)} f_k(x,y)$$

Each element of the vector is the average of a single channel. This allow us to use images with different sizes.

Rationale behind GAP

Fully connected layers are prone to overfitting

- They have many parameters
- Dropout was proposed as a regularized that randomly sets to zero a percentage of activations in the FC layers during training

The GAP was used as follows:
1. Remove the fully connected layer at the end of the network!
2. Introduce a GAP layer.
3. Predict by a simple soft-max after the GAP.
Watch out: the number of feature maps has to correspond to the number of output classes! In general, GAP can be used with more/fewer classes than channels provided an hidden layer to adjust feature dimension

The Advantages of GAP Layers:

- No parameters to optimize, lighter networks less prone to overfitting
- Classification is performed by a SoftMax layer at the end of the GAP
- More interpretability, creates a direct connection between layers and classes output (we'll see in localization) This makes GAP a structural regularizer
- Increases robustness to spatial transformation of the input images
- The network can be used to classify images of different sizes

GAP Increases Invariance to Shifts
Features extracted by the convolutional part of the network are invariant to shift of the input image
The MLP after the flattening is not invariant to shifts (different input neurons are connected by different weights)→ A CNN-flattening networks will fail classifying shifted images!

Therefore, a CNN trained on centered images. might not be able to correctly classify shifted ones
The GAP solves this problem, as the two images lead to the same or very similar features which are averaged.

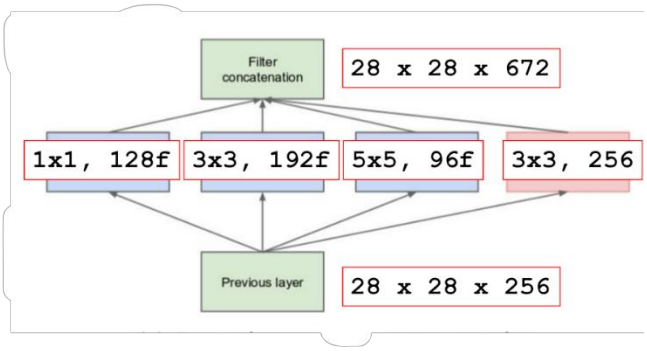
Inception Module

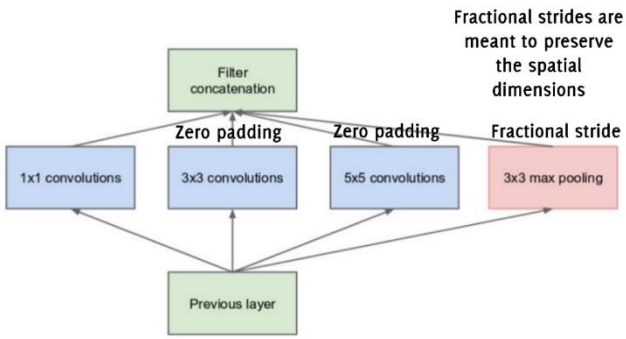
The most straightforward way of improving the performance of deep neural networks is by increasing their size (either in depth or width).
Bigger size typically means a larger number of parameters, which makes the enlarged network more prone to overfitting dramatically increase in the use of computational resources.
Moreover, image features might appear at different scale, and it is difficult to define the right filter size.

Inception modules are sort of «networks inside the network» or «local modules»
NB: Concatenation preserves spatial resolution

The solution is to exploit multiple filter size at the same level (1x1, 3x3, 5X5) and then merge by concatenation the output activation maps together.
AII the blocks preserve the dimensions spatial dimension by zero padding (filters) or by fractional stride (for Maxpooling)

Thus, outputs can be concatenated depth-wise

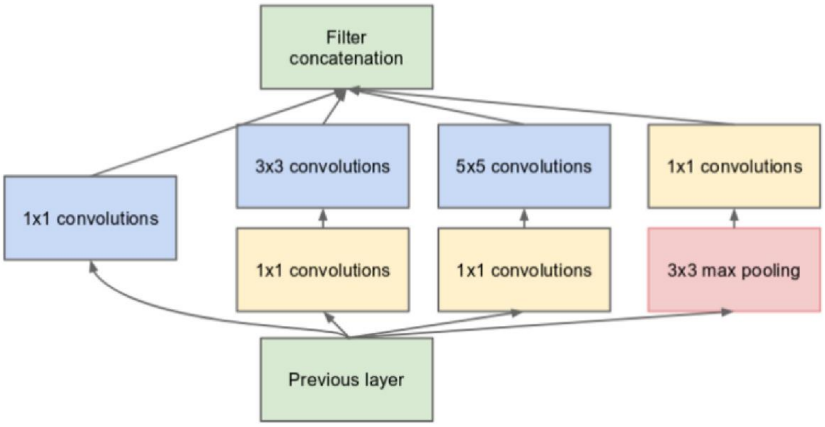


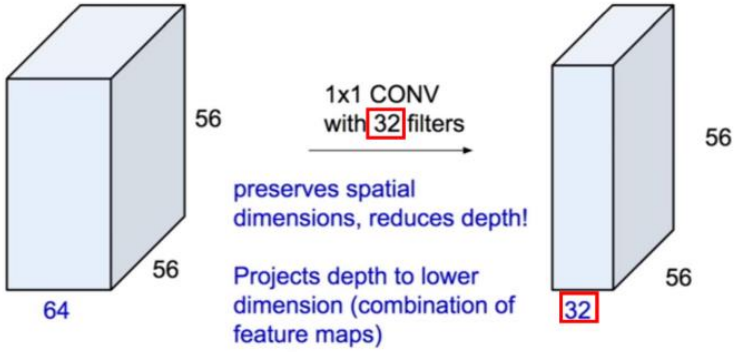


NB:

- Zero padding to preserve spatial size
- The activation map grows much in depth → expensive computation
- A large number of operations to be performed due to the large depth of the input of each convolutional block: 854M operations in this example → computational problems will get significantly wrost when stacking multiple layers

Idea: To reduce the computational load of the network, the number of input channels of each conv layer is reduced thanks to 1x1 convolution layers (*bottleneck layer*) before the 3x3 and 5x5 convolutions





The output volume has similar size but the number of operation required is significantly reduced due to the 1x1 conv: 358M operations now

Adding 1x1 convolution layers increases the number of nonlinearities

GoogLeNet (2014)

Deep network, with high computational efficiency
Only 5 million parameters, 22 layers of Inception modules

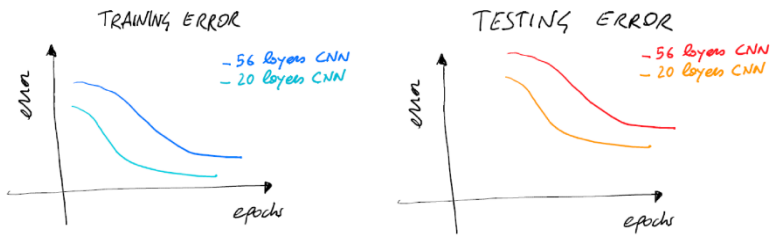
GoogleNet stacks 27 layers considering pooling ones.
At the beginning there are two blocks of conv + pool layers
Then, there are a stack of 9 of inception modules
No Fully connected layer at the end, simple global averaging pooling (GAP) + linear classifier + softmax.
Overall, it contains only 5 M parameters.

It also suffers of the dying neuron problem, → therefore the authors add two extra auxiliary classifiers on the intermediate representation to compute an intermediate loss that is used during training.
You expect intermediate layers to provide meaningful features for classification as well.
NB: Classification heads are then ignored / removed at inference time

ResNet (2015)

Very Deep network: 152 layers for a deep network trained on Imagenet!
1202 layers on CIFAR!
- Showed a performance better than the human one

INTUITION
Main investigation: is it possible to continuously improve accuracy by stacking more and more layers?
→ Increasing the network depth, by stacking an increasingly number of layers, does not always improve performance.
But this is not due to overfitting, since the same trend is shown in the training error (while for overfitting we have that training and test error diverge)

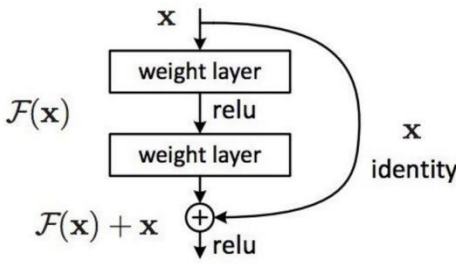


Similar trend, different # of layers

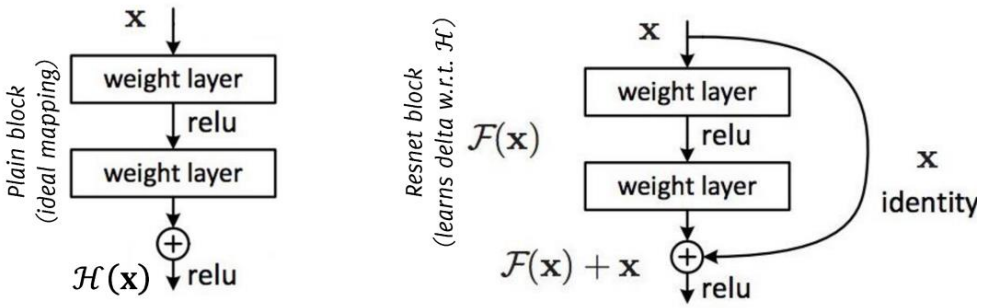
Deeper model are harder to optimize than shallower models.
→ However, we might in principle copy the parameters of the shallow network in the deeper one and then in the remaining part, set the weights to yield an identity mapping.
Therefore, deeper networks should be in principle as good as the shallow ones.
Since the experimental evidence, is different the identity function is not easy to learn!

Adding an "identity shortcut connection":

- helps in mitigating the vanishing gradient problem and enables deeper architectures
- Does not add parameters
- In case the previous network was optimal, the weights to be learned goes to zero and information is propagated by the identity
- The network can still be trained through back-propagation



→ RESIDUAL LEARNING
Intuition: force the network to learn a different task in each block. If H(x) is the ideal mapping to be learned from a plain network, by skip connections we force the network to learn F(x) = H(x) — x.



F(x) is called the *residual* (something to add on top of identity), which turns to be easier to train in deep networks.
Weights in between the skip connection can be used to learn a «delta», a residual i.e., F(x) to improve over the solution that can be achieved by a shallow network

The weights (convolutional layers) are such that to preserve dimension depth-wise or are re-arranged by 1x1 convolutions.

The rationale behind adding this identity mapping is that:
* It is easier for the following layers to learn features on top of the input value
* In practice the layers between an identity mapping would otherwise fail at learning the identity function to transfer the input to the output
* The performance achieved by resNet suggests that probably most of the deep layers have to be close to the identity!

The ResNet is a stack of *152 layers of this module*.
The network alternates:

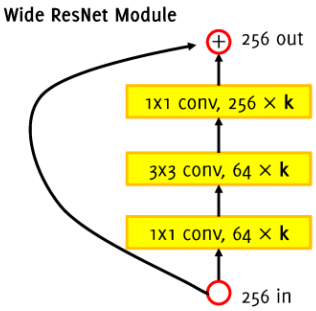
- Some spatial pooling by convolution with stride 2
- Doubling the number of filters

At the beginning there is a convolutional layer
At the end, no FC but just a GAP to be fed in the final soft max

Deeper networks are able to achieve lower errors as expected

Wide ResNet

Use wider residual blocks (F x k filters instead of F filters in each layer)
* 50-layer wide ResNet outperforms 152-layer original ResNet
* Increasing width instead of depth more computationally efficient (parallelizable)

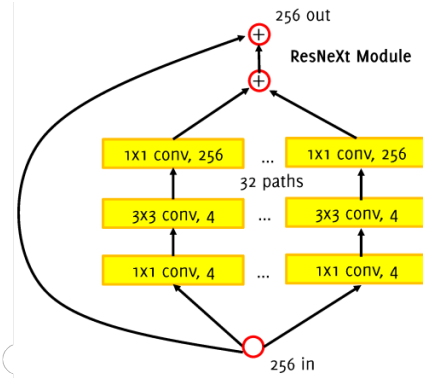


ResNeXt

Widen the ResNet module by adding multiple pathways in parallel (previous wide Resnet was just increasing the number of filters and showing it achieves similar perf. with fewer blocks)

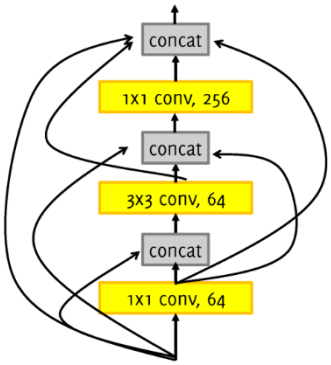
Similar to inception module where the activation maps are being processed in parallel

Different from inception module, all the paths share the same topology



DenseNet (2017)

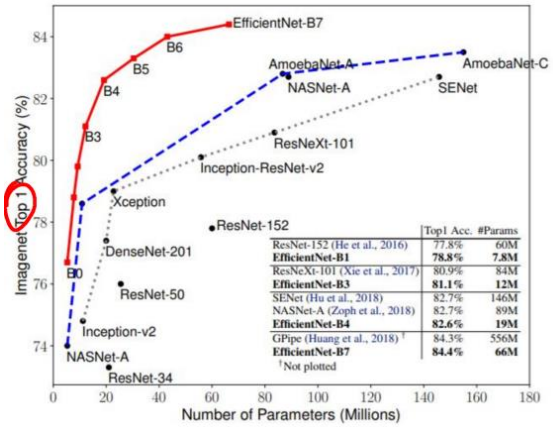
In each block of a DenseNet, each convolutional layer takes as input the output of the previous layers.
Short connections between convolutional layers of the network.
Each layer is connected to every other layer in a feed-forward fashion



In each block a DenseNet, each convolutional layer takes as input the output of the previous layers
Each layer is connected to every other layer in a feed-forward fashion
This alleviates vanishing gradient problem, promotes feature re-use since each feature is spread through the network

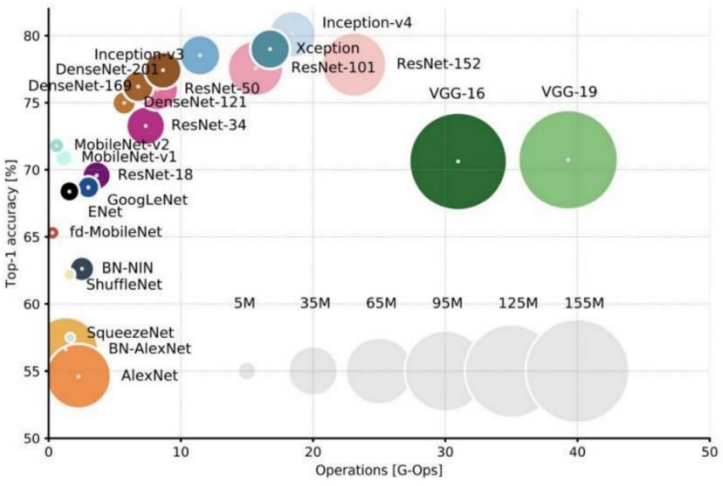
EfficientNet (2019)

A new scaling method that uniformly scales all dimensions of depth/width/resolution using a simple vet highly effective compound coefficient



COMPARISON

- Made in 2016



- VGGs are the least efficient: very large computational and memory usage

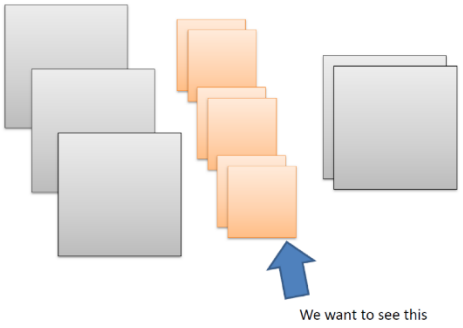
- Inception models are the most efficient and best performing. Inception v4 combines ResNet and Inception

- AlexNet are the least performing and not particularly efficient

- Mobile net and squeezenet are instead designed for improving efficiency

CNN VISUALIZATION

Visualizing CNN filters



Remember: Relation between convolution and template matching → the first layer seems to match low level features such as edges and simple patterns that are discriminative to describe data

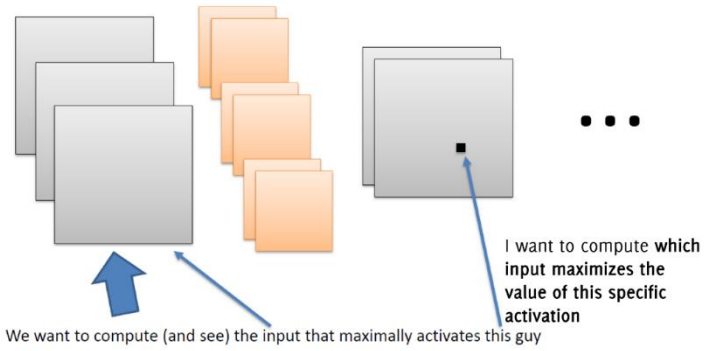
First layer's filters are not easily interpretable
→ another way to determine "what the deepest layer see" is required! → we look at activations

Visualizing Maximally Activating Patches

1. Select a neuron in a deep layer of a pre-trained CNN on ImageNet
2. Perform inference and store the activations for each input image.
3. Select the images yielding the maximum activation.
4. Show the region (patches) corresponding to the receptive field of the neuron.
5. Iterate for many neurons.

See slide 90

Computing Input maximally activating a neuron



1. Compute the gradient of the output with respect to the input
We can always compute gradient between values in a CNN, not only for the minimizing the loss. All the operations are known

2. Nudge the input accordingly: our output will increase its activation
Now we perform gradient ascent, we modify the input in the direction of the gradient to increase our function that is the value of the selected pixel in the activation map

Iterate the procedure to modify the input.
Some form of regularization can be added to the selected pixel value to steer the input to look more like a natural image.

Shallow layers respond to fine, low level patterns.
Intermediate layers -> middle level patterns
Deep layers respond to complex high level patterns

Computing images maximally activating softmax input

Adopt gradient descent to maximally activate a neuron before the SoftMax (thus the network «score», which indicates the prediction)

$$\hat{I} = \underset{I}{\operatorname{argmax}} S_c(I) + \lambda \|I\|_2^2$$

Being λ > 0 regularization parameter, c is a given output class.

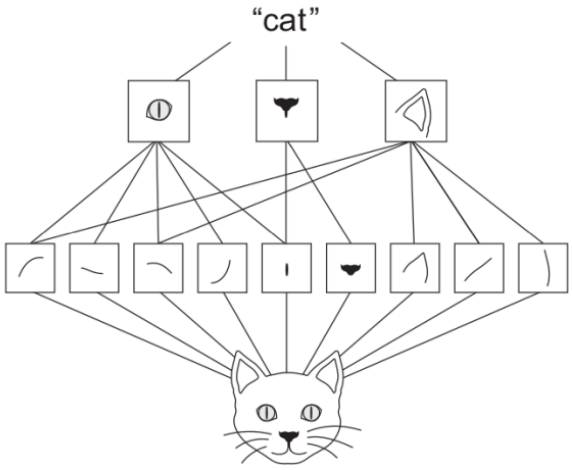
We add the regularization term to obtain smoother images

Optimize this through gradient ascent from the network for different classes c

See slide 100

Why CNNs work?

Convnets learn a hierarchy of translation-invariant spatial pattern detectors

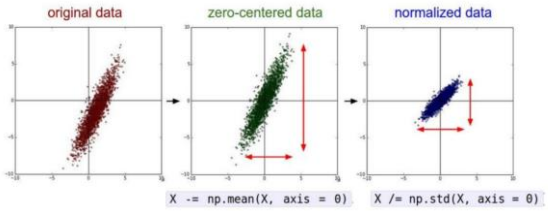


CNN FOR SEMANTIC SEGMENTATION

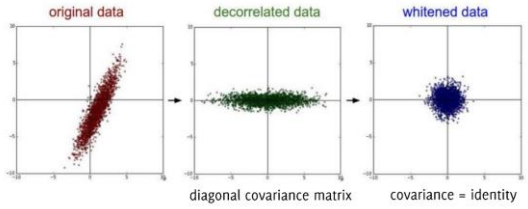
Data Pre-processing and Batch Normalization

In general, normalization is useful in gradient-based optimizers.
Normalization is meant to bring training data "around the origin" and possibly further rescale the data
In practice, optimization on pre-processed data is made easier and results are less sensitive to perturbations in the parameters

Standardization



PCA - based preprocessing
This is performed after having «zero-centered» the data



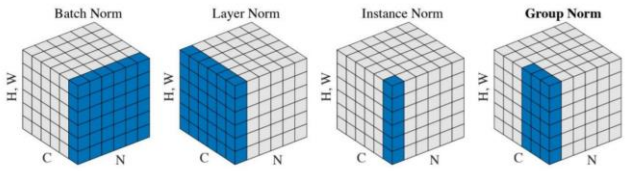
PCA/Whitening preprocessing are not commonly used with Convolutional Networks.
The most frequent option is to zero-center the data, and it is common to normalize every pixel as well.

Preprocessing and Training:
- Any preprocessing statistics (e.g. the data mean) must be computed on training data, and applied to the validation / test data.
- Do not normalize first and then split in training, validation, test
- Normalization statistics are parameters of your ML model

BATCH NORMALIZATION
Consider a batch of activations {x;}, the following transformation bring these to unit variance and zero mean

$$x_i' = \frac{x_i - E[x_i]}{\sqrt{\text{var}[x_i]}}$$

Where E[xi] and var[xi] are computed from each batch and separately for each channel!



Batch normalization adds after standard normalization a further a parametric transformation. Where parameters y and £ are learnable scale and shift parameters.

$$y_{i,j} = \gamma_j x_i' + \beta_j$$

Rmk: estimates E[x;] and var[x;] are computed on each minibatch, need to be fixed after training. After training, these are replaced by (running) averages of values seen during training.

During testing batch normalization becomes a linear operator! Can be fused with the previous fully-connected or conv layer. In practice networks that use Batch Normalization are significantly more robust to bad initialization. Typically Batch Normalization is used in between FC layers of deep CNN, but sometimes also between Conv Layers.

Batch Normalization

Pros: Makes deep networks much easier to train!
Improves gradient flow
Allows higher learning rates, faster convergence
Networks become more robust to initialization
Acts as regularization during training
Zero overhead at test-time: can be fused with conv!

Watch out: Behaves differently during training and testing: this is a very common source of bugs!

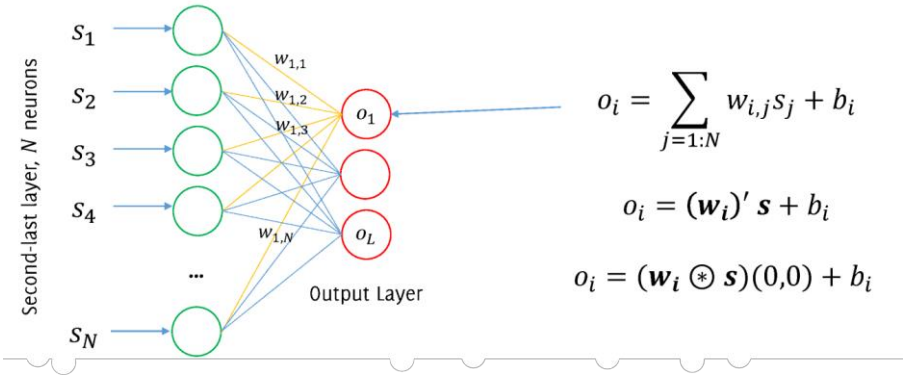
FULLY CONVOLUTIONAL NETWORKS

CNNSs are meant to process input of a fixed size (e.g. 200 x 200). 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.

Once I have my filters learned, I can use them open any image dimension (NB: but we need the right amount of channels). At a certain point when we reach the FC network we will have a different dimension of vector
→ CNN cannot compute class scores but can extract features

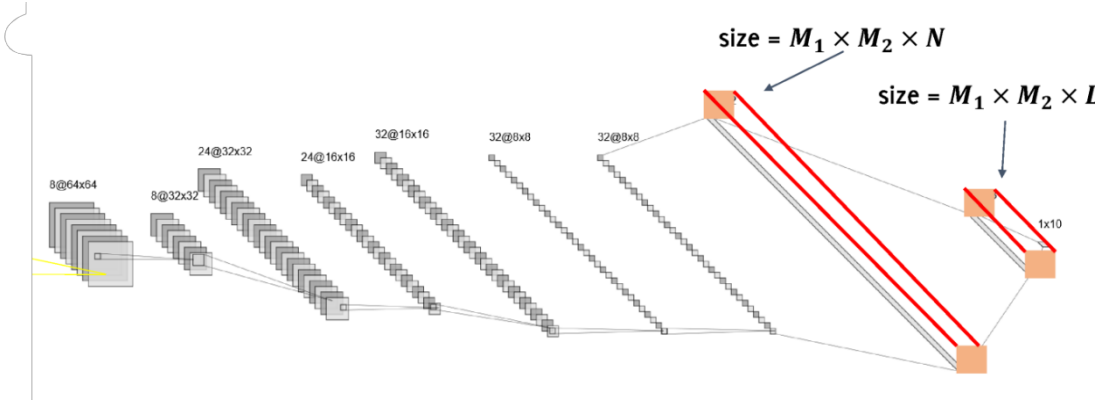
However, since the FC is linear, it can be represented as convolution!

Weights associated to output neuron i: $\boldsymbol{w_i} = \left[w_{i,j}\right]_{j=1:N}$

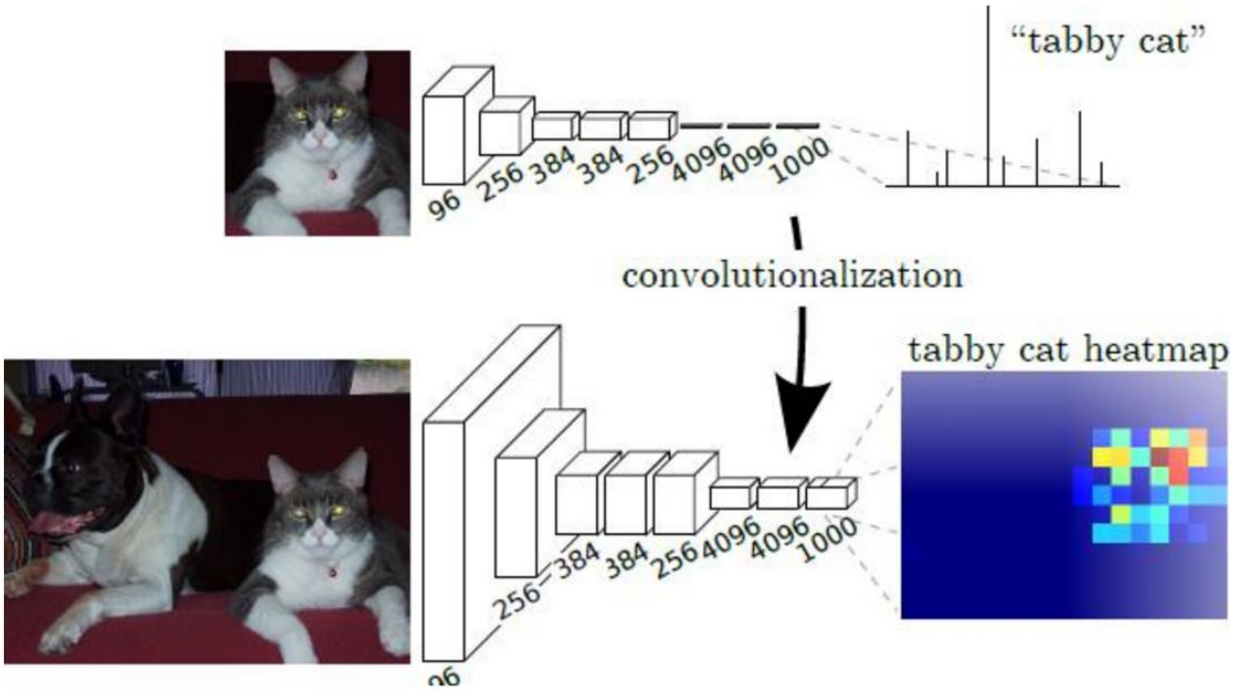


A FC layer of L outputs NN corresponds to a 2DConv layer having *L filters with size 1x1xN*

This transformation can be applied to each hidden layer of a FC network placed at the CNN top. This transformation consists in reading weights from the neuron in the dense layer and recycling together with bias in the new convolutional layer.



MITIGATION TO FCNN OF A PRETRAINED MODEL



This stack of convolutions operates on the whole image as a filter. Significantly more efficient than patch extraction and classification: this avoids multiple repeated computations within overlapping patches

Fully convolutional networks in keras

lt is necessary to get and set the weights of networks by means of the methods get _weights and set weights

* get the weights of the trained CNN

w7, b7 = model.layers[7].get weights ()

* reshape these weights to become ten 1x1 convolutional filters having depth equal to 256 (these sizes are dictated by the sizes of the dense layer in the "traditional" CNN)

w7.reshape (1, 1, 256, 10)

* assign these weights to the FCNN architecture

model2.layers[7].set_weights(w7, Db7)

IMAGE SEGMENTATION

THE PROBLEM

Assign to each pixel of an image $I \in \mathbb{R}^{RxCx3}$,
a label $\{l_i\}$ from a fixed set of categories $\Lambda = \{\text{"wheel", "cars", "castle", "baboon"}\}$,

$$I \rightarrow S \in \Lambda^{\mathrm{R\,x\,C}}$$

where $S(x, y) \in \Lambda$ denotes the class associated to the pixel (x,y)

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.

TRAINING SET

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

I *GT*

From Classification to Segmentation

Setup:

- You are given a pre-trained CNN for classification
- You possibly performed transfer learning to the problem at hand
- You have «convolutionalized» it to extract heatmaps

Limitation:

- Heatmaps are very low-resolution

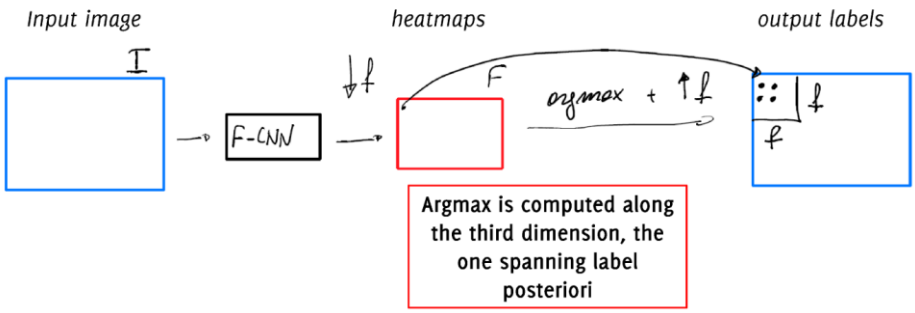
Goal:

- Obtain a Semantic Segmentation network for images of arbitrary size

Simple Solution(1) → DIRECT HEATMAP PREDICTIONS

We can assign the predicted label in the heatmap to the whole receptive field, however that would be a **very coarse estimate.**

We need to upsample the estimated map!



Simple Solution (2) → THE SHIFT AND STICH

Shift and Stich: Assume there is a downsampling ratio f between the size of input and of the output heatmap

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

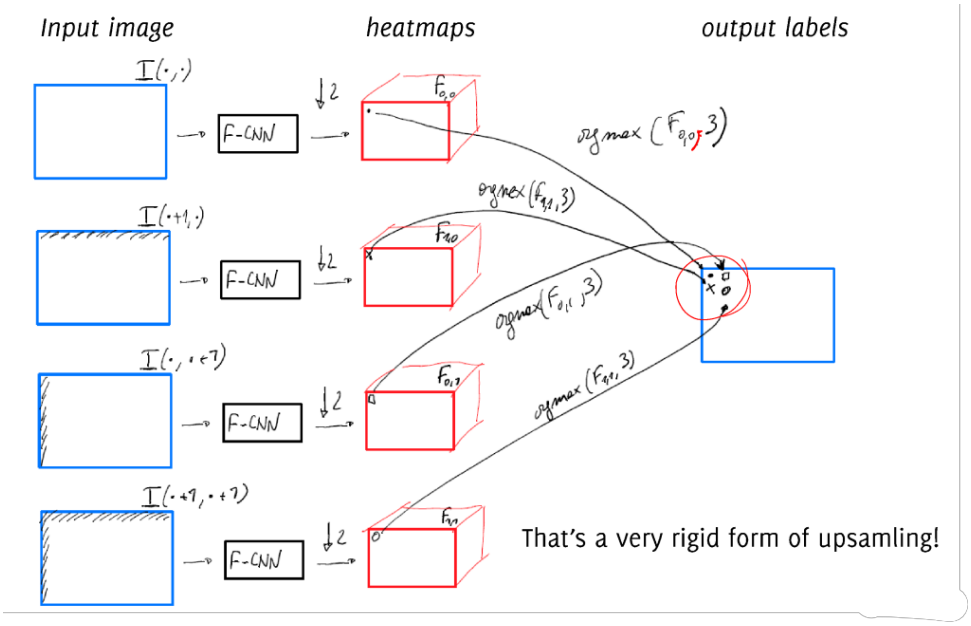
Map predictions from the f^2 heatmaps to the image: each pixel in the heatmap provides prediction of the central pixel of the receptive field

Interleave the heatmaps to form an image as large as the input

This exploits the whole depth of the network

Efficient implementation through the à trous algorithm in wavelet

However, the upsampling method is very rigid

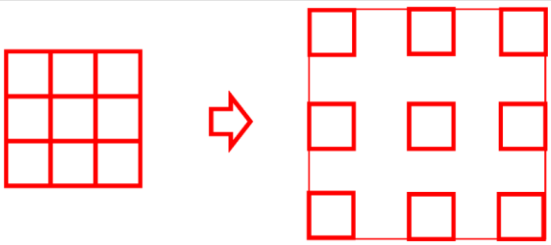


NB: This is an option of we have a classification network and we do not have annotations for the segmentation

The same effect as shift and stich can be obtained by

- Remove strides in pooling layer (conv or mp). This is equivalent as computing the output of all the shifted versions at once
- Rarefy the filters of the following convolution layer by upsamping and zero padding

$$f'_{ij} = \begin{cases} f_{i/s,j/s} & \text{if } s \text{ divides both } i \text{ and } j; \\ 0 & \text{otherwise,} \end{cases}$$



- Repeat the same operation along each channel dimension
- Repeat for each subsampling layer

Simple Solution (3) → ONLY CONVOLUTIONS

What if we avoid any pooling (just conv2d and activation layers)?

- Very small receptive field
- Very inefficient

Drawbacks of convolutions only

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*

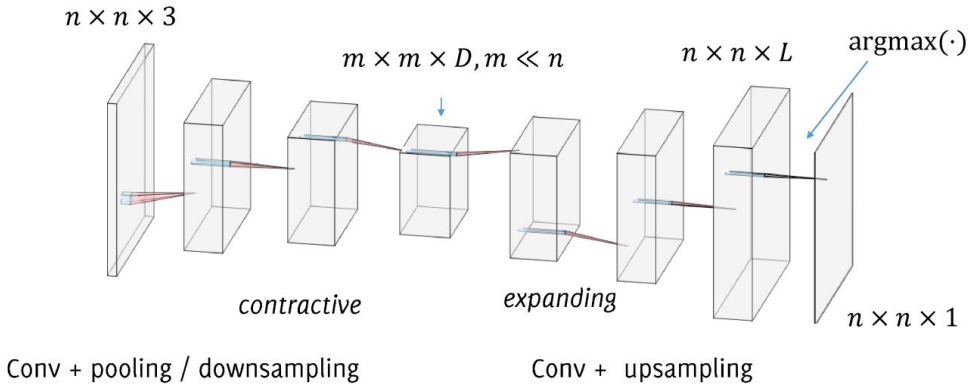
* *local* information resolves *where*

Combining fine layers and coarse layers lets the model make local predictions that respect global structure.

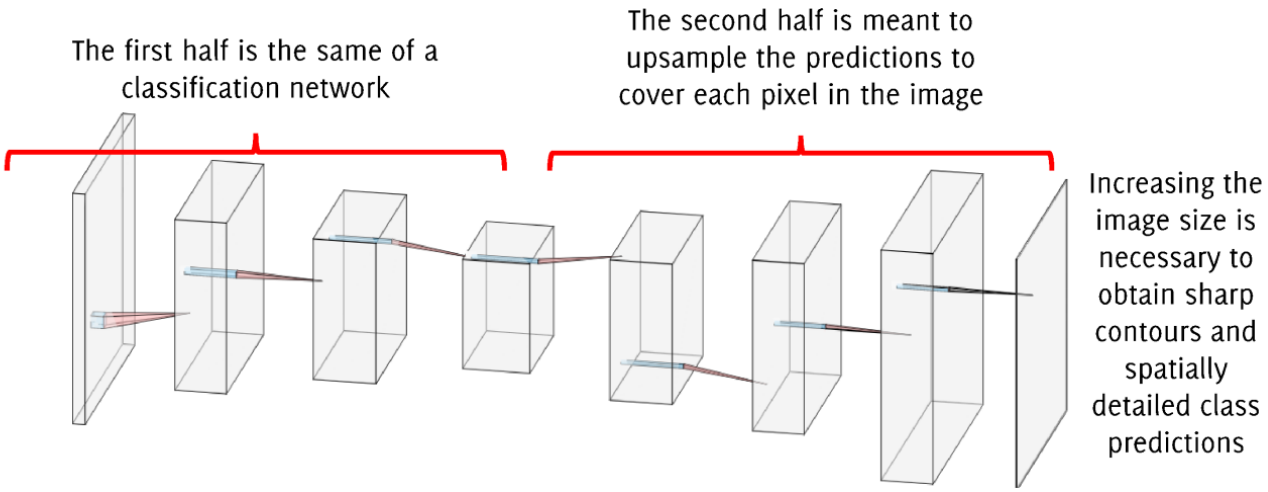
Low-dimensional representation and upsampling

An architecture like the following would probably be more suitable for semantic segmentation

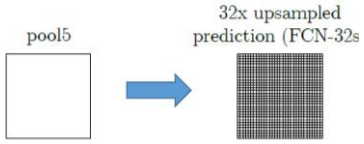
There is a substantial reduction and then a substantial increase



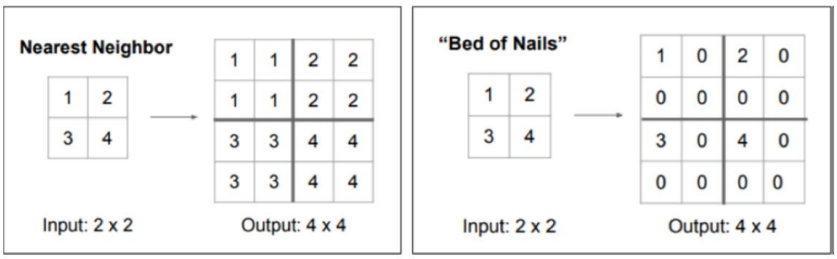
- Deep features encode semantic information
- High resolution output to recover local information



HOW TO PERFORM UPSAMPLING?

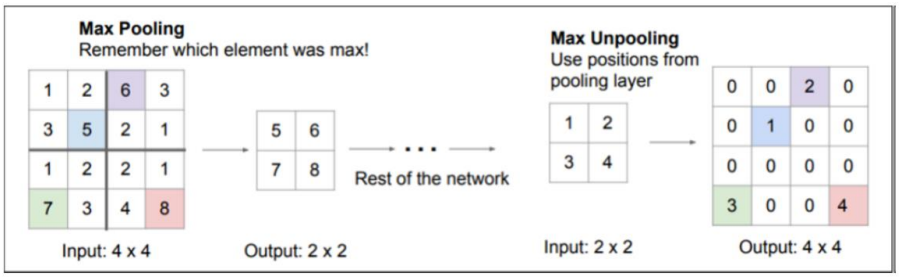


- NEAREST NEIGHBOUR & BED OF NAILS

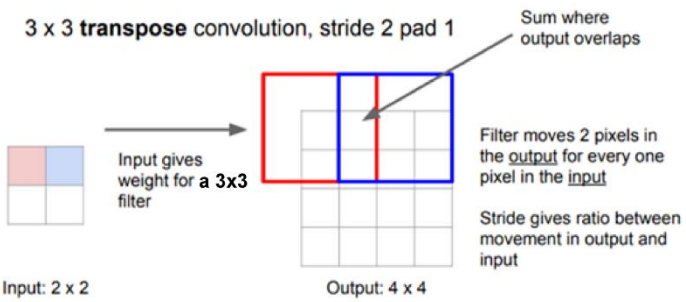


- MAX UNPOOLING

You have to keep track of the locations of the max during maxpooling



- TRANSPOSE CONVOLUTION

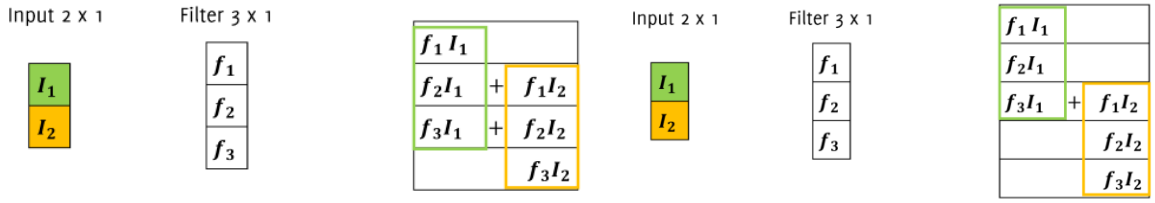


Transpose Convolution can be seen as a traditional convolution after having upsampled the input image.

Many names for transpose convolution: fractional strided convolution, backward strided convolution, deconvolution (very misleading!!!)

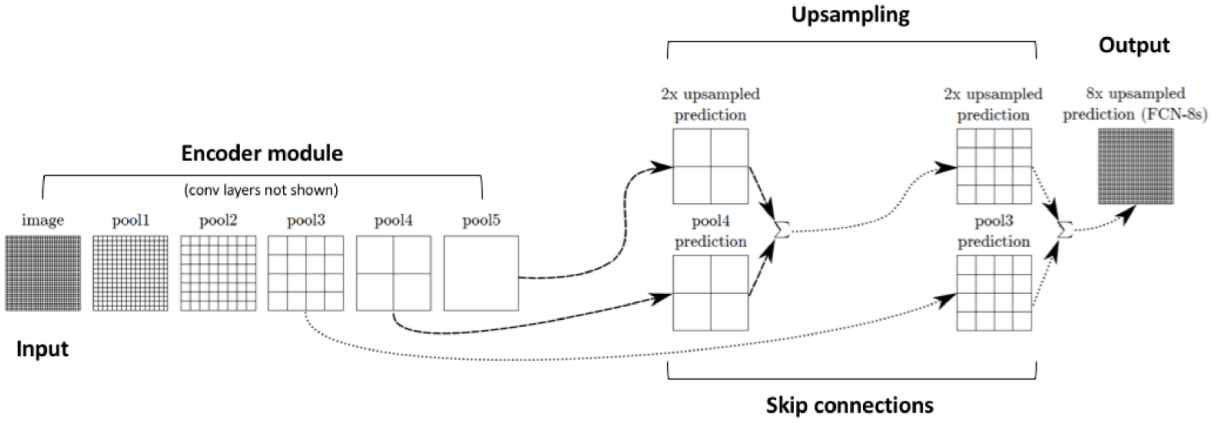
The idea of Transpose Convolution is to going backward, it's a one-to-many relationship: we want to associate one value to a matrix of values. If your input is a $2 * 2$ matrix, you want a $4 * 4$ output and you have a filter of size $3 * 3$ [stride 2, pad 1], get the first pixel value of the input and 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.

With stride = 1 and stride = 2:



Upsamping based on convolution gives more degrees of freedom, since the filters can be learned!
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 and are initialized equal to the bilinear interpolation.
These predictions however are very coarse. Upsampling filters are learned with initialization equal to the bilinear interpolation. The solution is to introduce Skip Connections

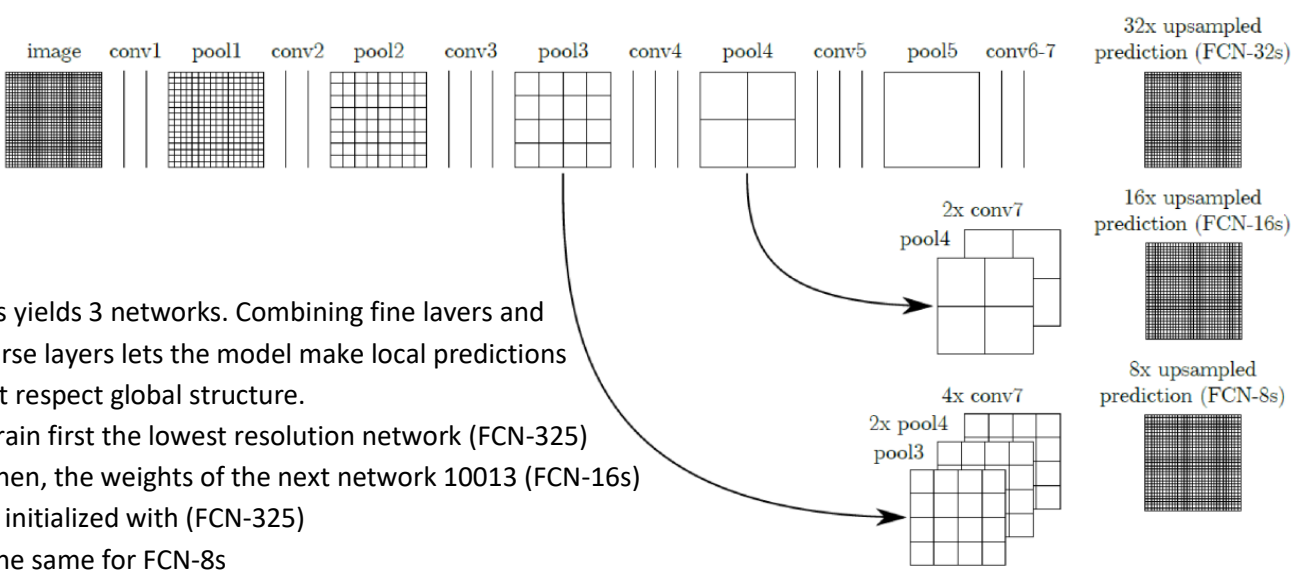
Skip connections



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.

Skip Connections:

- Supplement a traditional «contracting» network by successive layers where convolution is replaced by transpose convolution
- Upsampling is ubiquitous
- Upsampling filters are learned during training
- Upsampling filters are initialized using bilinear interpolation



This yields 3 networks. Combining fine lavers and coarse layers lets the model make local predictions that respect global structure.
* Train first the lowest resolution network (FCN-325)
* Then, the weights of the next network 10013 (FCN-16s) are initialized with (FCN-325)
* The same for FCN-8s
* All the FC layers are converted to convolutional layers 1x1

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

Comments on skip connections
* Both learning and inference can be performed on the whole-image at-a-time
* Itis possible to perform transfer learning/fine tuning of pre-trained classification models (segmentation typically requires fewer labels than classification)
* Accurate pixel-wise prediction is achieved by upsampling layers
* Outperforms state-of the art in 2015
* Being fully convolutional, this network handles arbitrarily sized input

Training a Fully-Convolutional CNN

The «patch-based» way

1. Preparea training set for a classification network
2. Crop as many patches x; from annotated images and assign to each patch, the label corresponding to the patch center
3. Train a CNN for classification from scratches, or fine tune a pre-trained model over the segmentation classes
4. Convolutionalization: once trained the network, move the FC layers to 1x1 convolutions
5. Design and train the upsampling side of the network

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

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \sum_{\boldsymbol{x_j} \in \boldsymbol{B}} \ell\left(\boldsymbol{x_j}, \theta\right)$$

where xj belongs to a mini-batch B
* Batches of patches are randomly assembled during training
* It is possible to resample patches for solving class imbalance
* lt 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:

$$\hat{\theta} = \underset{\boldsymbol{x_j} \in I}{\operatorname{argmin}} \sum_{\boldsymbol{x_j} \in \boldsymbol{R}} \ell\left(\boldsymbol{x_j}, \theta\right)$$

Where xj are all the pixels in a region R of the input image and the loss is evaluated over the corresponding labels in the annotation for semantic segmentation.
Therefore, each region provides already a mini-batch estimate for computing gradient.

Whole image fully convolutional training is identical to patchwise training where each batch consists of all the receptive fields of the units below the loss for an image.
While this is more efficient than uniform sampling of patches, it reduces the number of possible batches

* FCNN are trained in an end-to-end manner to predict the segmented output S(-,-).
* This loss is the sum of losses over different pixels. Derivatives can be easily computed through the whole network, and this can be trained through backpropagation.
* No need to pass through a classification network first.
* Takes advantage of FC-CNN efficiency, does not have to re-compute convolutional features in overlapping regions.
* End-to-end training is more efficient than patch-wise training

Observations:
Loss function is computed all at once using categorical crossentropy and comparing the labels of the map with the ground truth.
This method is better than the previous one if we consider the fact that we avoid convolutions on overlapping regions which increase the computational effort but at the same time we are cutting out the randomicity with which we assembled batches cause ow we have just an image. Moreover we cannot resample under-represented batches to adjust the loss.

Limitations of full-image training and solutions:

- Minibatches in patch-wise training are assambled randomly. Image regions in full-image training are not. To make the estimated loss a bit stochastic, adopt random mask
  being M(x;) a binary random variable

$$\text{minimize} \sum_{x_j} M(\boldsymbol{x_j}) \ell(\boldsymbol{x_j}, \theta)$$

  Instead of stochastic assembling of batches which is not possible in this case, we can compute the loss function on the resulting map after applying at each epoch a mask composed of 1s and 0s in order to evaluate at each epoch a different set of pixels.

- It is not possible to perform patch resampling to compensate for class imbalance. We can introduce a weighted loss to face unbalanced dataset: higher weights will be attributed to under-represented classes.

$$\text{minimize} \sum_{x_j} w(\boldsymbol{x_j}) \ell(\boldsymbol{x_j}, \theta)$$

U-Net: Convolutional Networks for Biomedical Image Segmentation

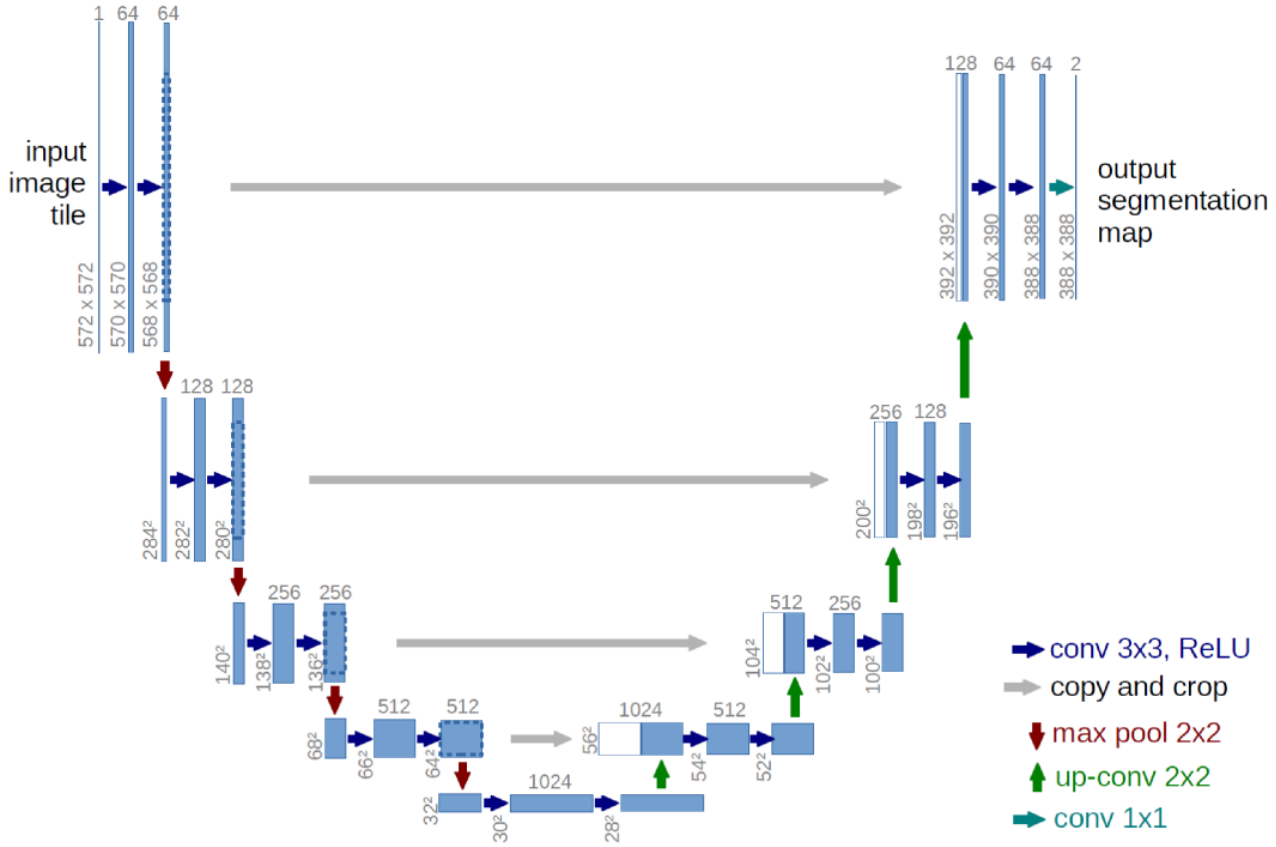
Network formed by:

- A contracting path
- An expansive path

No fully connected layers

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



U-Net: Contracting path

Repeats blocks of:

- 3x3 convolution + ReLU ('valid' option, no padding)
- 3x3 convolution + ReLU ('valid' option, no padding)
- Maxpooling 2 x 2
- At each downsampling the number of feature maps is doubled

U-Net: Expanding path

Repeats blocks of:

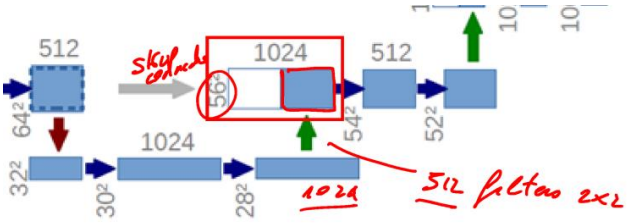
- 2x2 transpose convolution, halving the number of feature maps (but doubling the spatial resolution)
- Concatenation of corresponding cropped features
- 3x3 convolution + ReLU
- 3x3 convolution + ReLU

  NB: these two 3x3 conv are aggregated during upsampling

U-Net: Network Top

No fully connected layers: there are L convolutions against filters 1X1XN, to yield predictions out of the convolutional feature maps Output image is smaller than the input image by a constant border

At the very bottom of the network we have 1024 activations of dimensions 30x30. At the beginning they were almost 600x600 so the down-sampling factor was overall almost 20.



After computing a traditional convolution (blue arrow) we perform up-sampling of factor 2 with transpose convolution (green arrow); the result is aggregated through concatenation to the activation map having same spatial extent (after cropping only the central part) in the contractive part and then through another traditional convolution the 1024 feature maps are mixed up obtaining 512 maps as output. Same procedure is then repeated till the end of the expanding part.

U-Net Training

Loss function: categorical crossentropy with the use of weight composed by class weight + an additional weight used to take more into account classification at boundaries of the image to have sharp divisions among different adjacent objects to be segmented

Full-image training by the weighted loss function

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

where the weight

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

- wc is used to balance class proportions (remember no patch resampling in full-image training)
- w0e^() enhances classification performance at borders of different objects. This term is large at pixels close to borders delimiting objects of different cells.
  - d1 is the distance to the border of the closest cell
  - d2 is the distance to the border of the second closest cell

NB: weights are large when the distance to the first two closest cell is small

GAP, LOCALIZATION & CNN EXPLANATIONS

LOCALIZATION

The input image contains a single relevant object to be classified in a fixed set of categories. The task is to:
1. assign the object class to the image
2. locate the object in the image by its bounding box

A training set of annotated images with label and a bounding box around each object is required.
Extended localization problems involve regression over more complicated geometries (eg human skeleton)

THE PROBLEM:

Assign to each pixel of an image $I \in \mathbb{R}^{RxCx3}$,
a label $\{l\}$ from a fixed set of categories $\Lambda = \{\text{"wheel"}, \text{"cars"}, \text{"castle"}, \text{"baboon"}\}$,

$$I \rightarrow (x, y, h, w, l$$

where (x, y, h, w) are the coordinates of the bounding box enclosing the object.

NB: We could use categorical cross entropy as loss, but for now it wouldn't work because categorical cross entropy if for the prob of matching a label. We need both the categorical cross entropy and a loss for a linear output to predict x y z

THE SIMPLEST SOLUTION
Train a network to predict both the class label and the bounding box.
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:

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

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.

It is also possible to adopt a pre-trained model and then train the two FC separately... however it is always better to perform at least some fine tuning to train the two jointly.

NB: we need to tune correctly the α as it determines whether the net is more focused on classification or regression.

EXAMPLE:

- Extension to human pose estimation
  Pose estimation is formulated as a CNN-regression problem towards body joints. The output is fixed, so the network predicts position of joints even if they are not visible in the picture.

  * 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.
Train a CNN to predict a 2k vector as output by using an Alexnet-like architecture.
Adopt a l^2 regression loss of the estimated pose parameters over the annotations.
This can be also defined when a few joints are not visible.
Reduce overfitting by augmentation (translation and flips). Multiple networks have been trained to improve localization by refining joint position in a crop around the initial detection.

WEAKLY-SUPERVISED LOCALIZATION

Global Averaging Pooling Revisited → visualizing what matters most for CNN predicitons

Perform localization over an image without images with annotated bounding box
* Training set provided as for classification with image-label pairs $\{(I, l)\}$ where no localization information is provided

THE GAP REVISITED

The advantages of GAP layer extend beyond simply acting as a structural regularizer that prevents overfitting

In fact, CNNs 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.

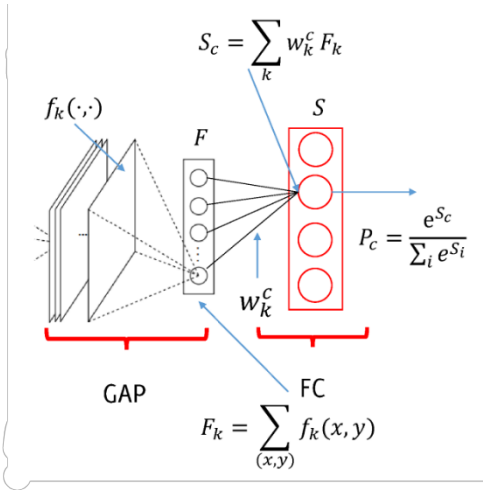
A CNN trained on object categorization is successfully able to localize the discriminative regions for action classification as the objects that the humans are interacting with rather than the humans themselves.

CLASS ACTIVATION MAPPING (CAM)

Identifying exactly which regions of an image are being used for discrimination.
CAM are very easy to compute.
It just requires: * FC layer after the GAP
* a minor tweak

The Global Averaging Pooling (GAP) Layer

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



Add (and train) a single FC layer after the GAP. The FC computes Sc, for each class c as the weighted sum of {Fk}, where weights are defined during training.
Then, the class probability Pc via soft-max (class c).

Remark: when computing

$$S_c = \sum_k w_k^c F_k$$

w_k^c encodes the importance of Fk for the class c, $\{w_k^c\}_{k,c}$ are the parameters of the last FC layer.

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

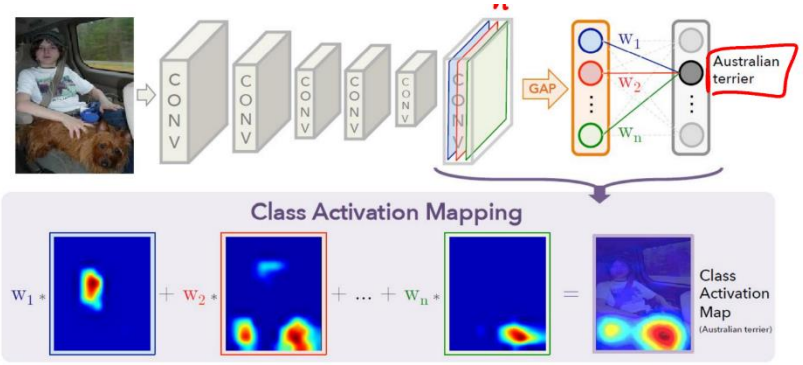
So you can construct an image where in each pixel there is the linear combination of all the weights corresponding to a slices rescaled by the corresponding weighth w c k . The image in the output is exactly the heatmap which highlights which part of the image contributes more on the decision of the 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.(x,y) directly indicates the importance of the activations at (x,y) for predicting the class c

Rmk: unlike GAP thanks to the softmax, the depth of the last convolutional activations can differ from the number of classes

Now, the weights represents the importance of each feature map to yield the final prediction. Upsampling might be necessary to match the input image



In the example the CAM has been computed for the second class which was “Australian Terrier” but it can be computed for all the classes considering the associated weights (indeed the n feature maps remain the same for all the classes).

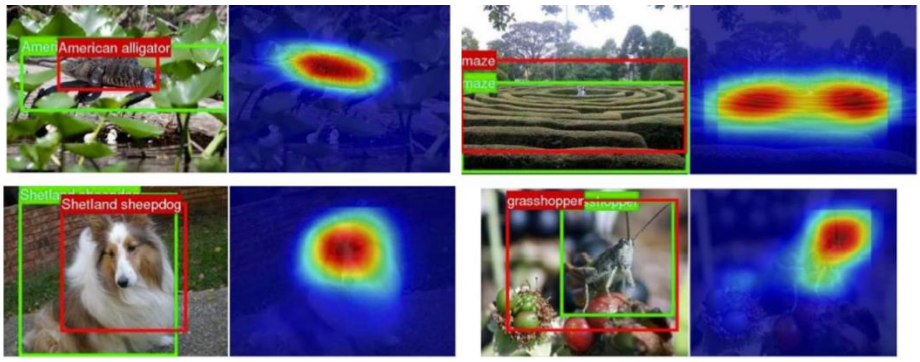
Remarks

- CAMcan be included in any pre-trained network, as long as all the FC layers at the end are removed
- The FC used for CAM is simple, few neurons and no hidden 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 (but this reduces the semantic information within these layers)
- GAP: encourages the identification of the whole object, as all the parts 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

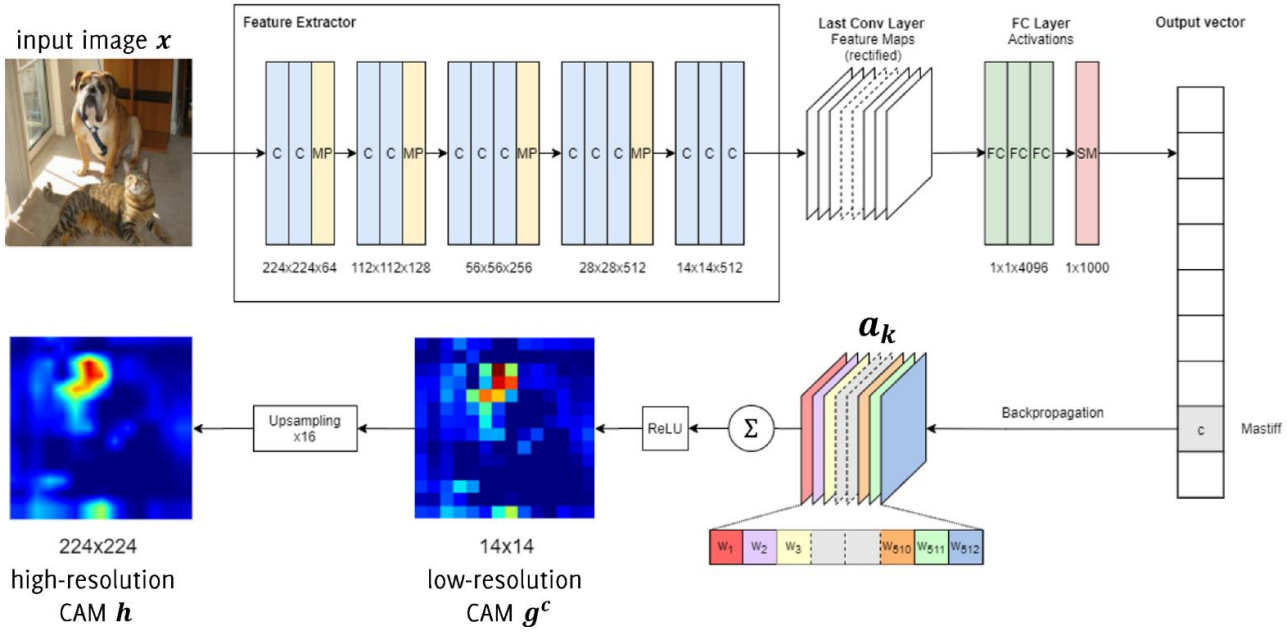
How to compute the bounding boxes as a result of the localization process?

Even though we have not provided annotated bounding boxes to perform the training of the network, it is providing us them as output! They are obtained according to the intensity of the CAMs, specifically a box (green) is located in the region of the map where we have a peak of intensity minus up to 20% of intensity which is actually coherent with the ground truth box (red one).

→ Weakly supervised localization



GRAD-CAM AND CAM-BASED TECHNIQUES



→ More flexibility and superior performance, no need to modify the network top.

The desired heatmaps should:
- be class discriminative
- capture fine-gradient details (high-resolution)
  This is critical in many applications (eg. Medical/industrial imaging)

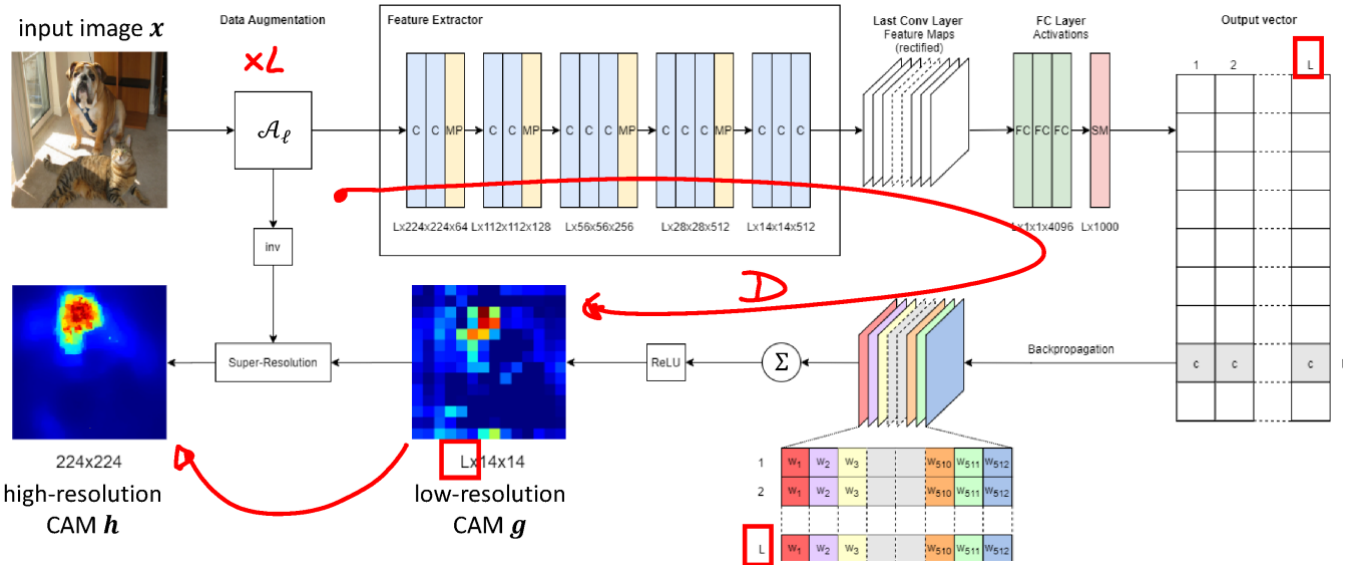
AUGMENTED GRAD-CAM

We consider the augmentation operator A_l: $R^{NxM} \rightarrow R^{NxM}$, including random rotations and translations of the input image x.

Augmented Grad-CAM: increase heat-maps resolution through image augmentation

All the responses that the CNN generates to the multiple augmented versions of the same input image are very informative for reconstructing the high-resolution heatmap h

NB: By using augmentation of the image, we get different heat maps



SUPER RESOLUTION

We perform heat-map **Super-Resolution** (SR) by taking advantage of the information shared in multiple low-resolution heat-maps computed from the same input under different - but known – transformations.

CNNs are in general invariant to roto-translations, in terms of predictions, but each map actually contains different information.

General approach, our SR framework can be combined with any visualization tool (not only Grad-CAM)

We model heat-maps computed by Grad-CAM as the result of an unknown downsampling operator D : RIXYM + Rnm The high-resolution heat-map h is recovered by solving an inverse problem

$$\underset{h}{\operatorname{argmin}} \frac{1}{2} \sum_{l=1}^{L} \|\mathcal{D}\mathcal{A}_\ell h - g_\ell\|_2^2 + \lambda TV_{\ell_1}(h) + \frac{\mu}{2}\|h\|_2^2$$

TV_{l1}: Anistropic Total Variation regularization is used to preserve the edges in the target heat-map (high-resolution)

$$TV_{\ell_1}(\boldsymbol{h}) = \Sigma_{i,j} \|\partial_x \boldsymbol{h}(i,j)\| + \|\partial_y \boldsymbol{h}(i,j)\|$$

This is solved through Subgradient Descent since the function is convex and non-smooth

OTHER GRADIENT-BASED SALIENCY MAPS

Grad-CAM++ : Same formulation of CAM as Grad-CAM, but weights are computed by higher-order derivatives of the class score with respect to the feature maps. Increases the localization accuracy of the heat-maps in presence of multiple occurrences of the same object in the image.

Sharpen Focus: highlights only the pixels where the gradients are positive.

$$w_k^c = \frac{1}{nm} \sum_i \sum_j \text{RELU}\left(\frac{\partial y_c}{\partial \boldsymbol{a_k}(\boldsymbol{i},\boldsymbol{j})}\right)$$

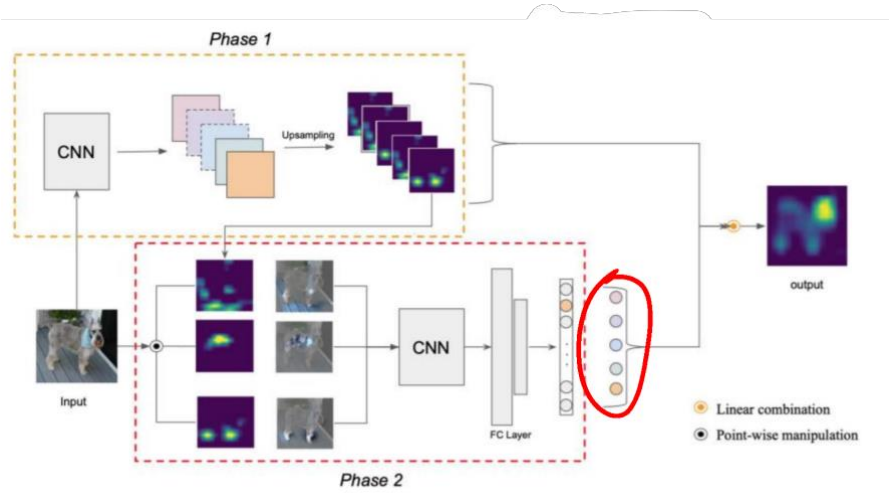
Smooth Grad-CAM++: it averages multiple heat-maps corresponding to noisy versions of the same input image.

OTHER PERTURBATION-BASED SALIENCY MAPS

Idea: Perturb the input image and assess how the class score changes.

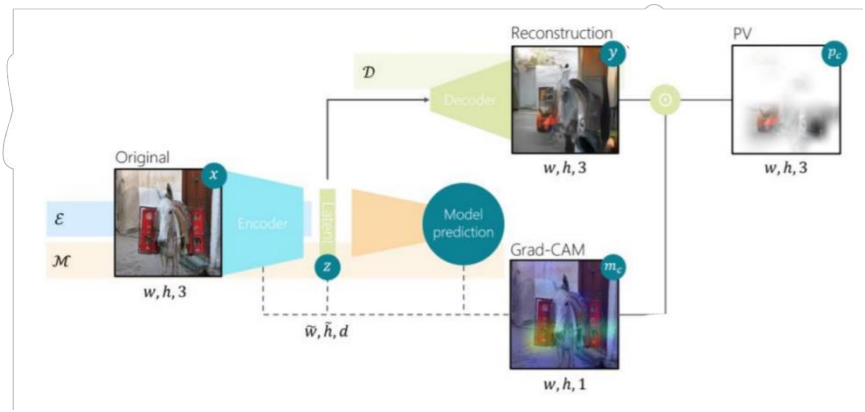
Score-CAM: Each feature map (unsampled and normalized in [0,1]) operates as a mask on the original image, which is then forward-passed to obtain the score on the target class.

$$w_k^c = CNN(\boldsymbol{h_k^c} \circ \boldsymbol{x}) - CNN(\boldsymbol{x})$$



PERCEPTION VISUALIZATION

Perception Visualization: provides explanations by exploiting a neural network to invert latent representations



Encoder E is a truncation of the model M which we want to explain, decoder D is trained to reconstruct the encoder's latent representations. From these, we compute Grad-CAM saliency maps and reconstructions, which are then combined to obtain PV.

Give better insight on the model's functioning than what was previously achievable using only saliency maps. A study on circa 100 — subjects shows that PV is able to help respondents better determine the predicted class in cases where the model had made an error

CNN FOR OBJECT DETECTION

OBJECT DETECTION - THE PROBLEM:

Assign to each pixel of an image $I \in \mathbb{R}^{RxCx3}$,

- Multiple labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{\text{"wheel"}, \text{"cars"}, \text{"castle"}, \text{"baboon"}\}$, each corresponding to an instance of that object.
- The coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object.

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

Object Detection Task

Given a fixed set of categories and an input image which contains an unknown and varying number of instances, draw a bounding box on each object instance.

A training set of annotated images with labels and bounding boxes for each object is required Each image requires a varying number of outputs

INSTANCE SEGMENTATION - THE PROBLEM

Assign to each pixel of an image $I \in \mathbb{R}^{RxCx3}$,

- Multiple labels $\{l_i\}$ from a fixed set of categories $\Lambda = \{\text{"wheel"}, \text{"cars"}, \text{"castle"}, \text{"baboon"}\}$, each corresponding to an instance of that object.
- The coordinates $\{(x, y, h, w)_i\}$ of the bounding box enclosing each object.
- thesetof pixels S in each bounding box corresponding to that label

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

THE STRAIGHTFORWARD SOLUTION: SLIDING WINDOW

* Similar to the sliding window for semantic segmentation
* A pretrained model is meant to process a fixed input size (e.g. 224 x 224 X 3)
* Slide on the image a window of that size and classify each region.

Adopt the whole machinery seen so far to each crop of the image

NB: The background class has to be included!

Cons:

* Very inefficient! Does not re-use features that are «shared» among overlapping crops
* How to choose the crop size?
* Difficult to detect objects at different scales!
* A huge number of crops of different sizes should be considered...

Plus:

* No need of retraining the CNN

REGION PROPOSAL

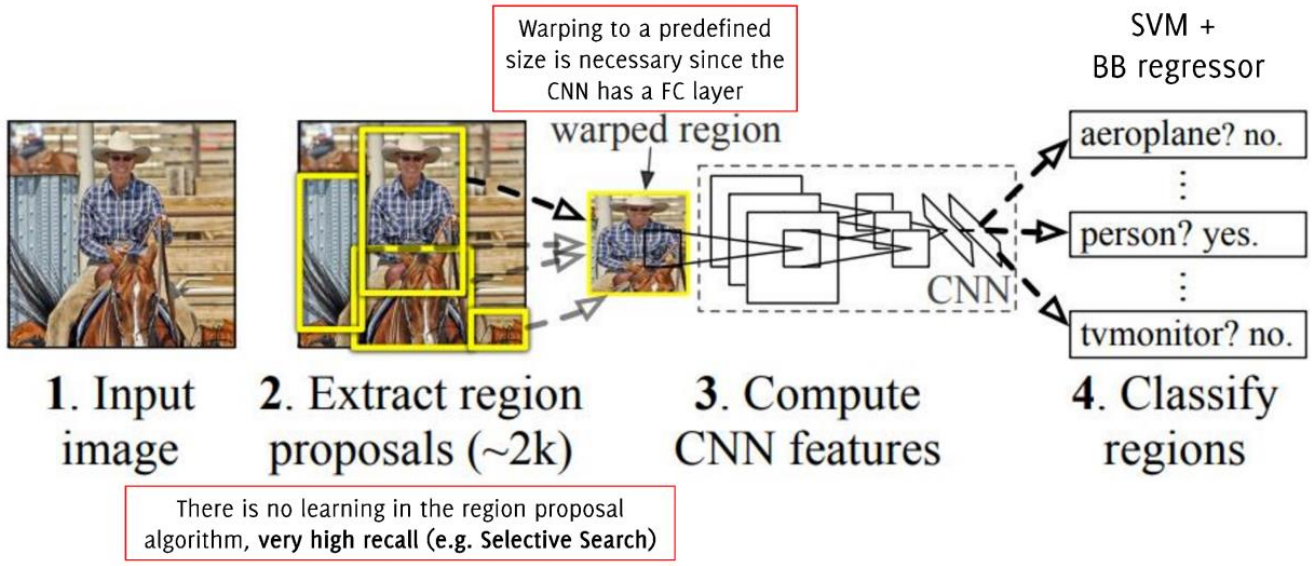
Region proposal algorithms (and networks) are meant to identify bounding boxes that correspond to a candidate object in the image. Algorithms with very high recall (but low precision) were there before the deep learning advent

The idea is to:

* Apply a region proposal algorithm
* Classify by a CNN the image inside each proposal regions

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.



AlexNet is used as a pre-trained network to compute features
SVM is trained to minimize the classification error over the extracted ROI

The regions are refined by a regression network to correct the bounding box estimate from ROI algorithm .
Include a background class to get rid of those regions not corresponding to an object

The pretrained CNN is fine-tuned over the classes to be detected (21 vs 1000 of Alextnet) by placing a FC layer after feature extraction. No end-to-end training of the SVM.

R-CNN Limitations

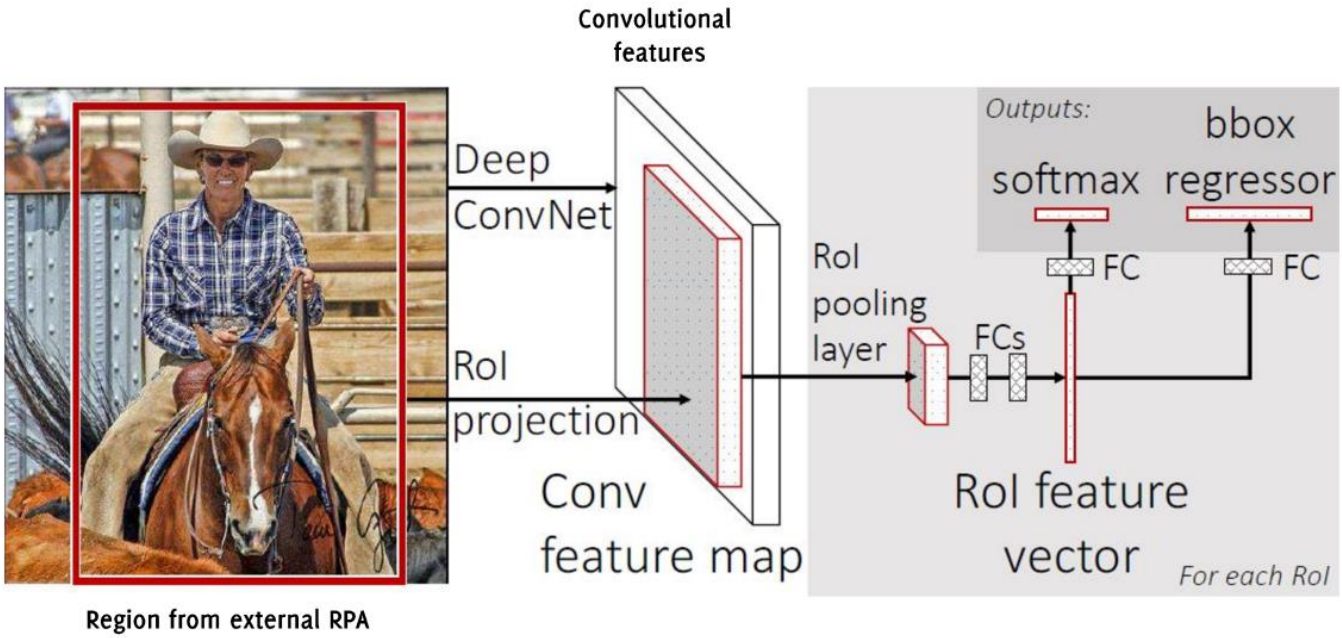
- Ad-hoc training objectives and not an end-to-end training
- Fine-tuning network with softmax classifier (log loss) before training SVM
- Train post-hoc linear SVMs (hinge loss)
- Train post-hoc bounding-box regressions (least squares)
- Region proposals are from a different algorithm and that part has not been optimized for the detection by CNN
- Training is slow (84h), takes a lot of disk space to store features
- Inference (detection) is slow since the CNN has to be executed on each region proposal (no feature re-use)
  47s / image with VGG16

Rmk: efficiency in object detection network is key! Otherwise, you might want to train a segmentation network instead!

Fast R-CNN

Instead of resizing the proposal and give it in input to the network, the proposal is mapped into the latent space which has been created previously feeding the network with the entire image; in this way we can just run the inference once.
Then the ROI pooling layer, which is a sort of maxpooling layer, make the input size fixed.
Then the map is given to FC layers (which are NN so there's no need to store latent representations to train at different times) and the output is given to the 2 heads: classification (softmax) and regression (BB).

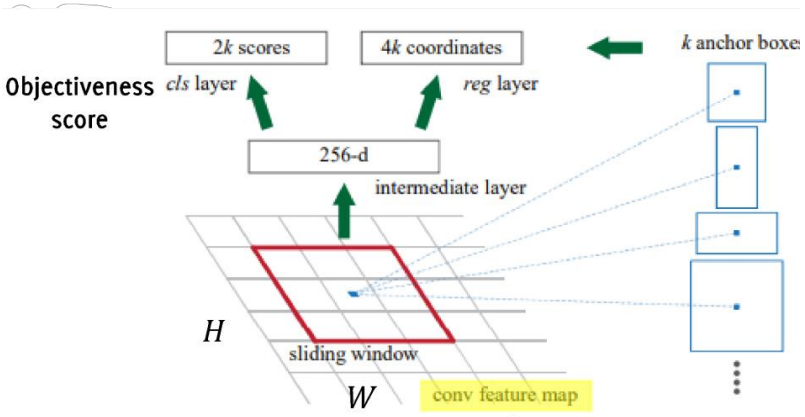
1. The whole image is fed to a CNN that extracts feature maps.
2. Region proposals are identified from the image and projected into the feature maps. Regions are directly cropped form the feature maps, instead from the image: → re-use convolutional computation.
3. Fixed size is still required to feed data to a fully connected layer. ROI pooling layers extract a feature vector of fixed size H Xx W from each region proposal. Each ROI in the feature maps is divided in a H x W grid and then maxpooling 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 multitask loss to be optimized (as in RCNN, but no SVM here).
5. Training performed in an end-to-end manner; convolutional part executed only once



In this new architecture it is possible to back-propagate through the whole network, thus train the whole network in an end-to-end manner
It becomes incredibly faster than R-CNN during testing.
Now that convolutions are not repeated on overlapping areas, the vast majority of test time is spent on ROI extraction (e.g. selective search).

FASTER R-CNN

- Instead of the ROI extraction algorithm, train a **region proposal network (RPN),** which is a F-CNN (3x3 filter size)
- RPN operates on the same feature maps used for classification, thus at the last conv layers
- RPN can be seen as an additional module that improves efficiency and focus Fast R-CNN over the most promising regions for object detection



Region Proposal Network

Goal: Associate to each spatial location k anchor boxes, i.e. ROI having different scales and ratios (e.g. K = 3 x 3, 3 sizes of the anchor side, 3 height/width ratios).

The network outputs H x W x k candidates anchor and estimate objectiveness scores for each anchor.

Intermediate layer

It is a standard CNN layer that takes as input the last layer of the feature-extraction network and uses 256 filters of size 3 x 3.

It reduces the dimensionality of the feature map and maps each region to a lower dimensional vector of size (output size H x W x 256)

Estimating k Anchors

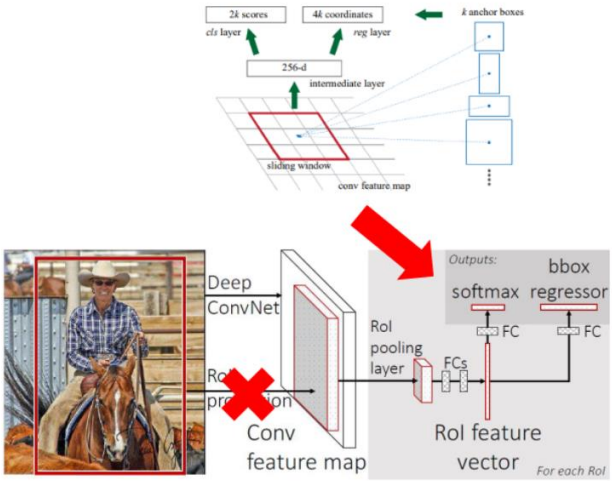
- The cls (classification) network is trained to predict the object probability, i.e. the probability for an anchor to contain an object [contains / does not contain] → 2k probability estimates
- Made of a stack of convolutional layers having size 1 x 1
- Each of these K probability pairs corresponds to a specific anchor (having a specific dimension) and expresses the probability for that anchor in that spatial location to contain any object
- The reg (regression) network is trained to adjust each of the k predicted anchor to better match object ground truth → 4k estimates for the 4 bounding box coordinates
- Each of these k 4 — tuples expresses the refinements for a specific anchor

If you want to train the network to predict different anchors there is no need to design different RPN, but just to define different labels when training the RPN, associated to different anchors.

RPN return HxWxk region proposals, thus replaces the region proposal algorithms (selective search) -> to have boxes spread all over the image

After RPN there is a non-maximum suppression based on the objectiveness score

Remaining proposals are then fed to the ROI pooling and then classified by the standard Fast-RCNN architecture.



Training

Training now involves 4 losses:

- RPN classify object/non object
- RPN regression coordinates
- Final classification score
- Final BB coordinates

- During training, object/non object ground truth is defined by measuring the overlap with annotated BB
- The loss includes a term of final BB coordinates, as these are defined over the image, RPN in the latent domain.

Training procedure:

1. Train RPN keeping backbone network frozen and training only RPN layers. This ignores object classes by just bounding box locations (Multi-task loss cls + reg)
2. Train Fast-RCNN using proposals from RPN including the backbone
3. Fine tune the RPN in cascade of the new backbone
4. Freeze backbone and RPN and fine tune the last layers of the Faster R-CNN

At test time,
* Take the top — 300 anchors according to their object scores
* Consider the refined bounding box location of these 300 anchors
* These are the ROI to be fed to a Fast R-CNN
* Classify each ROI and provide the non-background ones as output

Faster R-CNN provides as output to each image a set of BB with their classifier posterior
The network becomes much faster (0.25 test time per image)

Faster R-CNN → It's still a **two stage** detector:
First stage:
* run a backbone network (e.g. VGG16) to extract featues
* run the Region Proposal Network to estimate — 300 ROI

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

Example An industrial application slide 40 – 46

R-CNN methods are based on region proposals
There are also region-free methods, like: YOLO: You Only Look Once
SSD: Single Shot Detectors

YOLO

You Only Look Once → region-free method

The rationale

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 detection 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

Pipeline
1. divide the image in a coarse grid (e.g. 7x7)
2. each grid cell contains B anchors (base bounding box) associated
3. For each cell and anchor we predict:
    * The offset of the base bounding box, to better match the object: (dx, dy, dh, dw, objectness_score)
    * The classification score of the base-bounding box over the C considered categories (including background)

So, the output of the network has dimension 7x7xBx(5+C)

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 shares a similar ground of the RPN used in Faster R-CCN

Typically networks based on region-proposals are more accurate, single shot detectors are faster but less accurate

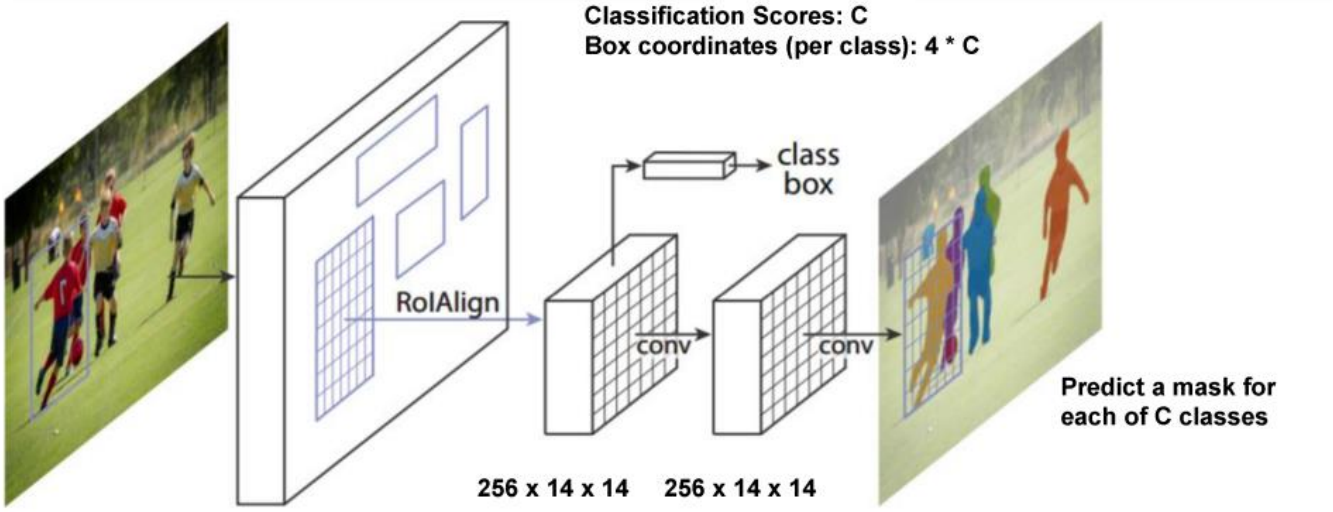
MASK R-CNN: INSTANCE SEGMENTATION

Instance Segmentation It combines the challenges of:

- Object detection (multiple instances present in the image)
- Semantic segmentation (associate a label to each pixel) separating each object instance

MASK R-CNN

As in Faster R-CNN, each ROI is classified and the bounding boxes are estimated
Semantic segmentation inside each ROI



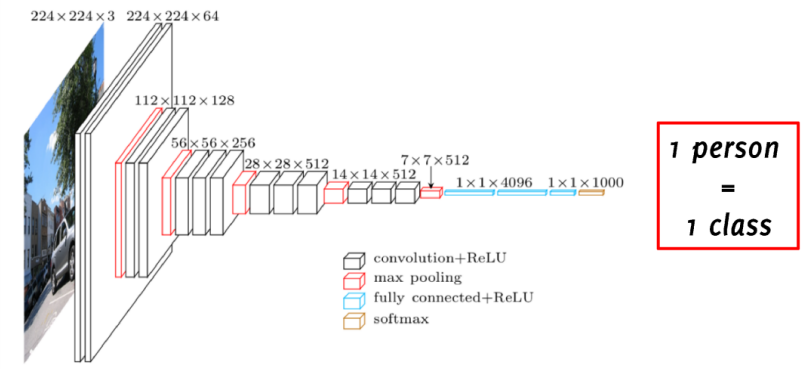
Extends Faster R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition
Mask is estimated for each ROI and each class

Example instance segmentation – slide 64-68

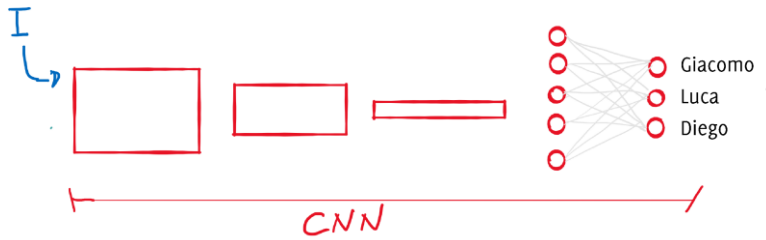
METRIC LEARNING

Say you are asked to implement a face identification system to open Politecnico di Milano door.
Classification? Detection? Segmentation?
No detection, no segmentation because there would be just one face in front of the camera.
No regression cause there's no possibility to "order" the faces. It can only be a classification problem.

Let's start fresh from a Convolutional Neural Network for classification



FACE ID IDENTIFICATION BY CLASSIFICATION



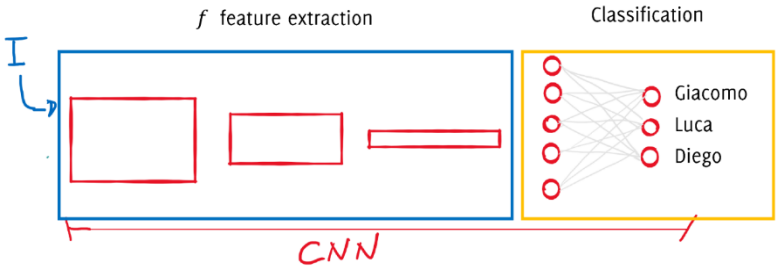
A training set
* A few images per class / person
* Images in different conditions (position, light, expression, clothes ...)
* A few Py snippets and a GPU...

What happens when you need to enroll a new employee? The whole network has to be retrained for each new person to be identified
→ we are not happy with this solution → a new approach is needed

IMAGE COMPARISON
Why don't we store a picture for each employee (template T1, T2, …), an then perform identification by pairing the input to its closest template?
... but how to perform identification? → $\hat{\imath} = \underset{j=1,..,4}{\operatorname{argmin}} \|I - T_j\|_2$
Where I is the image we want to classify and T is the template

NUT there is no Euclidian norm (distance from the center) to compare new input images with the templates!
We need a better distance measure for face identification:
distance from the latent representation!!!

LATENT REPRESENTATION



The feature extraction part is typically more general-purpose than the classification part

f(I): latent representation of I provides a meaningful description of the image in terms of patterns that are useful for solving the classification problem

DISTANCE among latent representations

$$\hat{\imath} = \underset{j=1,..,4}{\operatorname{argmin}} \left\| f(I) - f(T_j) \right\|_2$$

In practice:

1. Extract features from the first image f(I)
2. Perform identification as

$$\hat{\imath} = \underset{j=1,..,4}{\operatorname{argmin}} \left\| f(I) - f(T_j) \right\|_2$$

We would like that the distance among the latent representations of different individuals is bigger than that of the same individuals!

NB: No need to compute f(Tj) for every j, these can be directly stored

Can we do any better? .... After all, the network was not trained for this purpose...

METRIC LEARNING

We are still comparing latent representations trained for classification (over a few persons?), while not for comparing images

A more appealing perspective would be to train the network to measure distances between images

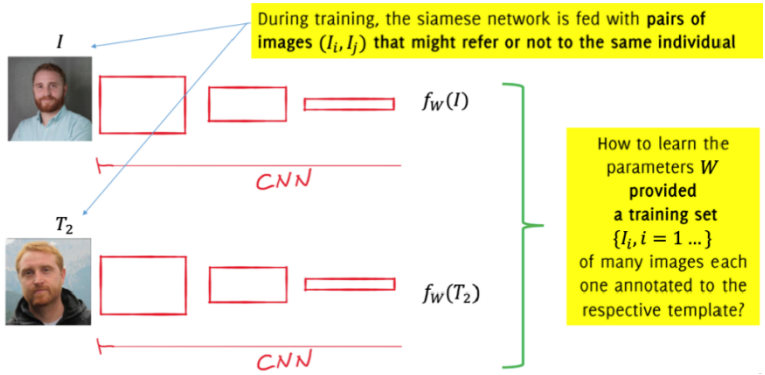
We would like to train the weights W of our CNN such that

$$\left\| f_W(I) - f_W(T_i) \right\|_2 < \left\| f_W(I) - f_W(T_j) \right\|_2 \quad \forall j \neq i$$

When I belongs to class i

SIAMESE NETWORKS

The "two" networks have to perform the same operations using the same weights W → "Siamese"



Positive pair = images belonging to the same person

Negative pair = images belonging to different people

Each network is fed with an image and has to be trained in such a way allowing to obtain the same weights of the siamese network at the end. How is it possible? Ad hoc loss functions should be minimized:

The **contrastive loss function** is defined as follow:

$$W = \underset{\omega}{\operatorname{argmin}} \sum_{i,j} \mathcal{L}_\omega(I_i, I_j, y_{i,j})$$

Where:

$$\mathcal{L}_\omega(I_i, I_j, y_{i,j}) = \frac{(1 - y_{i,j})}{2} \|f_\omega(I_i) - f_\omega(I_j)\|_2 + \frac{y_{i,j}}{2} \max\left(0, m - \|f_\omega(I_i) - f_\omega(I_j)\|_2\right)$$

- $y_{i,j} \in \{0,1\}$ is the label associate with the input pair (I_i, I_j):
  - 0 when (I_i, I_j) refers to the same person
  - 1 otherwise
- m is a hyperparameter indicating the margin we want (like in Hinge Loss)
- $\|f_\omega(I_i) - f_\omega(I_j)\|_2$ is the distance in the latent space.

M = safe margin, minimum distance between 2 images related to 2 different subjects.

This loss is assessed among a pair of images.

Looking at the expression of the loss we can derive that when the images are from the same individual (y=0) second term is zeroed thus we want to minimize the distance between the 2 images latent representations; in contrast when the images are from different individuals the first term is zeroed and we want to minimize the maximum between 0 and the difference between m and the distance between the latent representations whose objective is basically maximize the distance between the 2 latent representation.

Triplet Loss

A loss function such that a training sample I is compared against

* P a positive input, referring to the same person

* N a negative input, referring to a different person

We train the network to minimize the distance from the positive samples and maximize the distance from the negative ones.

$$\mathcal{L}_\omega(I, P, N) = \max(0, m + (\|f_\omega(I) - f_\omega(N)\|_2 - \|f_\omega(I) - f_\omega(P)\|_2))$$

Triplet loss forces that a pair of samples from the same individual are smaller in distance than those with different ones.

m always play the role of the margin.

The selection of triplets for training is a matter of study

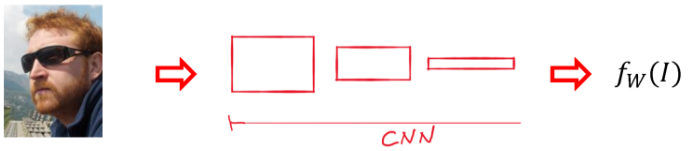
NB: this is an alternative to contrastive loss where 3 images are considered: I, P, N.

In this case we want the distance between I and N and that between I and P as large as possible.

To RECAP:

When a new image has to be verified

1. We feed the image I to the trained network, thus compute fw(I)



2. Identify the person having average minimum distance from templates (in case there are many associated to the same individual)

$$\hat{\imath} = \underset{u}{\operatorname{argmin}} \frac{\sum_{T_{u,j}} \left\| f_W(I) - f_W(T_{u,j}) \right\|_2}{\#\{T_u\}}$$

3. Assert whether

$$\frac{\sum_{T_{i,j}} \left\| f_W(I) - f_W(T_{i,j}) \right\|_2}{\#\{T_{i,j}\}} < \gamma$$

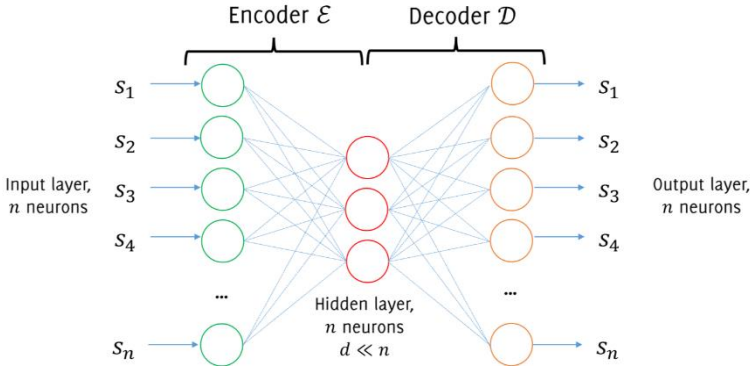
is sufficiently small, otherwise no identification

Other decision rules can be adopted (e.g. searching for the person giving the template with a minimum distance)

AUTOENCODERS AND GENERATIVE ADVERSARIAL NETWORKS

AUTOENCODERS

Autoencoders are neural networks used for data reconstruction (UNSUPERVISED LEARNING).
The typical structure of an autoencoder is:



The input needs to be compressed into 3 numbers and then expanded into 5 numbers.

We can train the network to make predictions of compressed input data.

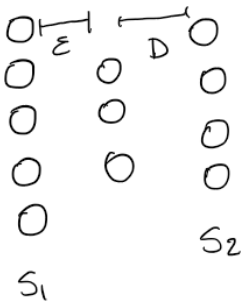
It's a regression problem: → loss:

$\|s - \hat{s}\|_2 \longrightarrow \|s - \mathcal{D}(\mathcal{E}(s))\|_2$

$\mathbb{R}^n \longrightarrow \mathbb{R}^d \quad d \ll n$

$\mathcal{E} : s \longmapsto \mathcal{E}(s)$

$\mathcal{D} : \hat{s} \longleftarrow \mathcal{E}(s)$



Autoencoders can be trained to reconstruct all the data in a training set.
The reconstruction loss over a batch S is:

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

and training of $D(\varepsilon())$ is performed through standard backpropagation algorithms (e.g. SGD). The autoencoder thus learns the identity mapping.

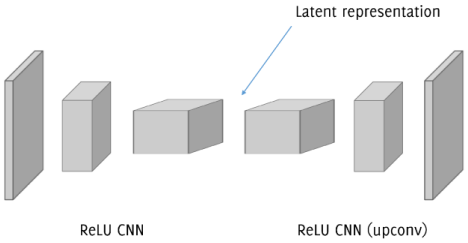
NB: there are no external labels involved in training the autoencoder, as it performs reconstruction of the input

Remark:

- Features z = ε(s) are typically referred to as latent representation
- AE 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
- Itis possible to add a regularization term +λR(s) to steer latent representation ε(s) to satisfy desired properties (e.g. sparsity) or the reconstruction $D(\varepsilon(s))$ (e.g. smoothness, sharp edges in case of images)
- More powerful and nonlinear representations can be learned by stacking multiple hidden layers (deep autoencoders)

Convolutional AutoEncoders

Of course it is possible to use convolutional layers and transpose convolution to implement a deep convolutional autoencoder



USING AUTOENCODERS FOR CLASSIFIER INITIALIZATION

If the objective of the classification is to distinguish people with and without glasses but we have very few labeled samples with which we could train a classification model, we can exploit autoencoders.

Indeed, we could use as input for the classifier the latent representation learned by the autoencoder (that at the center of the network) which hopefully contains meaningful features cause the autoencoder can be fed with unlabeled images (unsupervised learning), generally available.

Autoencoders can be used to initialize the classifier when the training set includes
* few annotated data (e.g. a large set S = {si} of unlabelled human faces)
* many unlabelled ones (e.g. a small set L = {(si, yi)} of faces labelled as Male, Female, Kids)

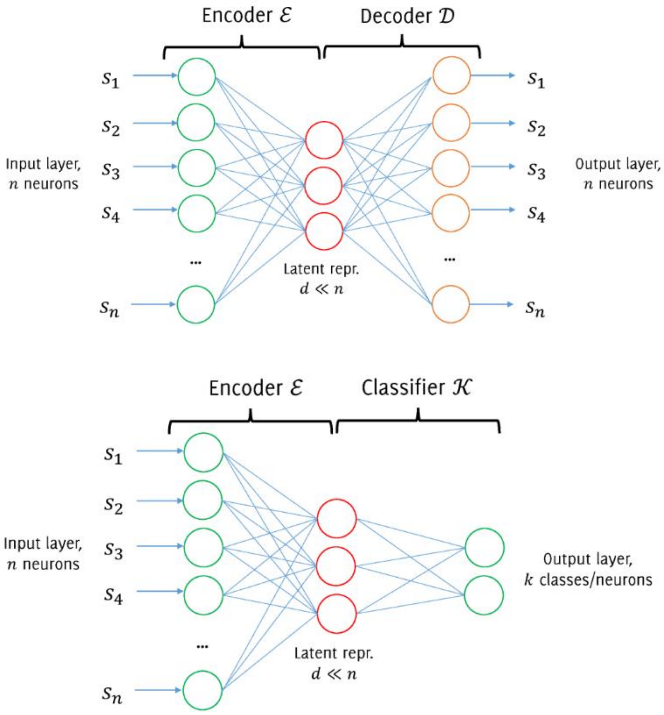
PROCEDURE:

1) Train the autoencoder in a fully unsupervised way, using the unlabelled data S

2) Get rid of the decoder and keep the encoder weights

3) Plug in a FC layer for classifying samples from the latent representation

4) Fine tune the autoencoder using the few supervised samples provided L. This is perfectly in line with «Transfer Learning» and holds for whatever model. If L is large enough, the encoder weights ε can also be fine-tuned



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

Applications of autoencoders:
Features extraction, Dimensionality Reduction, Image Denoising, Image Compression, Image Search, Anomaly Detection, Missing Value Imputation

GENERATIVE MODELS

GOAL: generate, given a training set of images TR = {xi}, generate other images that are similar to those in TR.

We can assume that images are located in a huge dimensional space according to the order of magnitude of the number of pixels present in an image: 10^6.
We can imagine that a sort of manifold in this space exists where images are located so if we move along this manifold, we can find other combination of pixels leading to have other images.

Characteristic of AI generated faces: the eyes are always in the same position and the back is very blurred

- Generative models can be used for data augmentation, simulation and planning
- Training generative models can also enable inference of latent representations that can be useful as general features
- Realistic samples for artwork, super-resolution, colorization, etc.
- You are getting close to the "holy grail" of modelling the distribution of natural images
- This can be a very useful regularization prior in other problems or to perform anomaly detection
- On top of specific application of image generation, the fist effective generative model (i.e. GANS) give rise to new training paradigm and practices (adversarial training)

Why don't we use autoencoders?

Intuition: if the autoencoder provides a good latent representation vector we could sample the vector and use it to realize/generate images too (in this case we would get rid of the encoder and use the decoder to generate images)!
However it does not work cause the latent representation is very high dimensional and we don't know the distribution of proper latent representation (or it is very difficult to estimate), there are lots of correlated features so it is quite difficult to model it correctly in order to obtain meaningful samplings to generate meaningful images…

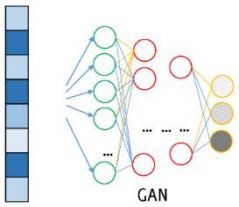
Other possibilities exist to generate images: Generative Adversarial Networks (GAN)!

Generative Adversarial Networks (GAN)

The GAN approach:
* Do not look for an explicit density model ϕ_s describing the manifold of natural images.
* Just find out a model able to generate samples that «looks like» training samples S c R^n

Instead of sampling from ϕ_s, just:
* Sample a seed from a known distribution ϕ_z . This is defined a priori and also referred to as noise.
* Feed this seed to a learned transformation that generates realistic samples, as if they were drawn from ϕ_s

Use a neural network to learn this transformation.
The neural network is going to be trained in an *unsupervised manner*, no label needed



$z \sim \phi_z$
Draw a sample from the noise distribution

A GAN is trained choosing a sample from a known distribution called "noise" (not from the manifold of the natural images which is too difficult to derive) and return an image in a higher dimensional space hopefully belonging to natural images manifold.
This without having ever seen an image!

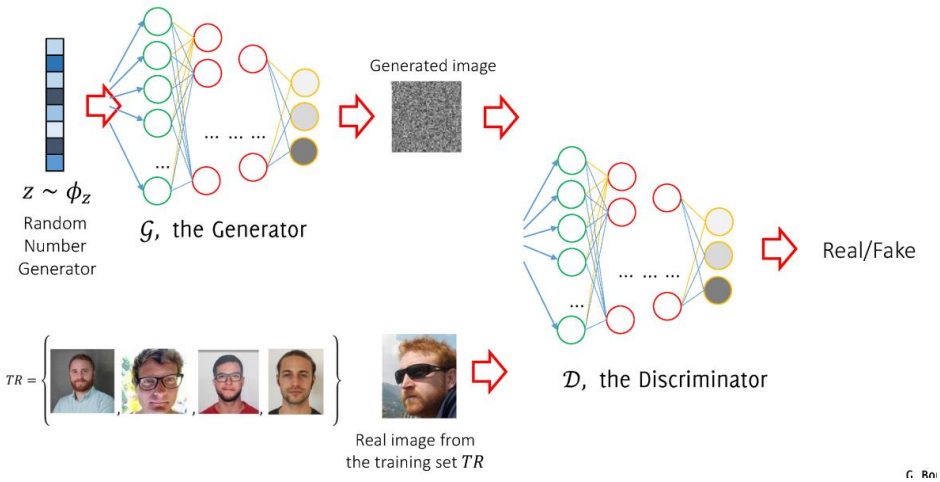
The biggest challenge is to define a suitable **loss** for assessing whether the output is a realistic image or not, whether it belongs to the manifold or not.
GAN solution: resort to a neural network to define the loss!

The GAN solution: Train a pair of neural networks addressing two different tasks that compete in a sort of two player (adversarial) game.
These models are: - Generator G that produces realistic samples e.g. taking as input some random noise. G tries to fool the discriminator
- Discriminator D that takes as input an image and assess whether it is real or generated by G
Train the two and at the end, keep only G.

The goal of D is to recognize all the images generated by G and classify them as fake.
G is trained to generate images that can fool D, namely can be classified as «real» by D. The loss of G is therefore given by D.
D and G play opposite games: they are Adversarial Networks!

At the end of training, we hope G to succeed in fooling D consistently. We discard D and keep G as generator.
D is expected effective to distinguish real and fake images... if G can fool G, this means that G is a generator.

Discriminator D is completely useless and as such dropped: after a successful GAN training, D is not able to distinguish the real/fake.



GAN - Setting up the stage
Both D and G are conveniently chosen as MLP or CNN.

NB: this notation is meant to visualize what are the NN parameters. The network is taking a single input.
Our networks take as input:

- $D = D(s, \theta_d)$
- $G = G(z, \theta_g)$

θ_g and θ_d are network parameters, s ∈ R^n is an input image (either real or generated by G) and z ∈ R^d is some random noise to be fed to the generator.

Our networks give as output:
$D(., \theta_d)$: $\mathbb{R}^n \rightarrow [0,1]$ gives as output the posterior for the input to be a true image
$G(., \theta_d)$: $\mathbb{R}^d \rightarrow \mathbb{R}^n$ gives as output the generated image

GAN - Training

A good discriminator is such:

- $D(s, \theta_d)$ is maximum when s ∈ S (true image from the training set)
- $1 - D(s, \theta_d)$ is maximum when s was generated from G
- $1 - D\big(G(z, \theta_g), \theta_d\big)$ is maximum when $z \sim \phi_z$

Training D consists in maximizing the binary cross-entropy:

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

This has to be 1 since $s \sim \phi_S$, thus images are real

This has to be 0 since $\mathcal{G}(\boldsymbol{z}, \theta_g)$ is a generated (fake) image

A good generator G makes D to fail, thus minimizes the above

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

Solve by an iterative numerical approach:

Alternate:

k-steps of Stochastic Gradient Ascent w.r.t. θ_d, keep θ_g fixed and solve

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

1-step of Stochastic Grandient Descent w.r.t. θ_g being θ_d fixed

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

and since the first term does not depend on θ_g, this consists in minimizing

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

Training procedure:

for $i = 1 \ldots$ #number of epochs

for k −times # gradient ascent steps for θ_d

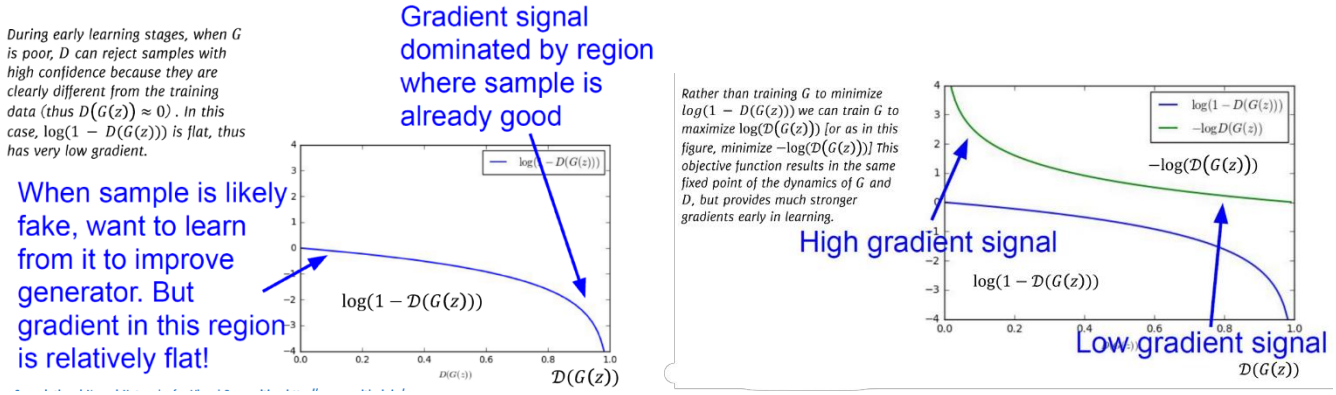
- Draw a minibatch $\{\boldsymbol{z_1}, \ldots, \boldsymbol{z_m}\}$ of noise realization
- Sample a minibatch of images $\{\boldsymbol{s_1}, \ldots, \boldsymbol{s_m}\}$
- Update θ_d by stochastic gradient ascend:

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

Draw a minibatch $\{\boldsymbol{z_1}, \ldots, \boldsymbol{z_m}\}$ of noise realizations # gradient descent steps for θ_g

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

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



One of the many GAN Training «trick» When optimizing for θ_g, instead of minimizing the following

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

we maximize this

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

Which is equivalent in therms of loss function.. provides a stronger gradient during the early learning stages

GAN Training procedure 2.0

for $i = 1 \ldots$ #number of epochs

for k −times # gradient ascent steps for θ_d

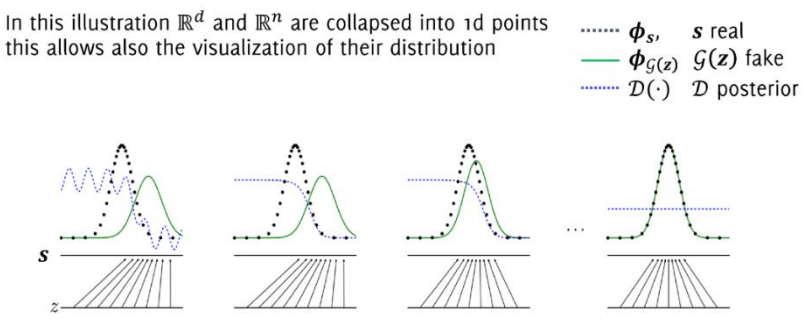
- Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realization
- Sample a minibatch of images $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m\}$
- Update θ_d by stochastic gradient ascend:

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

Draw a minibatch $\{\boldsymbol{z}_1, \ldots, \boldsymbol{z}_m\}$ of noise realizations # gradient descent steps for θ_g

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

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



We have to imagine an image distribution in 1 dimension as reported in the illustration.

G learns where the real images are and creates realistic images → distributions of fake and real images overimposed.

Initially D is high (1) for real images and low (0) for fake ones, then after the training it becomes flat cause it is no more able to distinguish between the 2 categories.

At the end of the day...
The discriminator D is discarded The generator G and ϕ_z are preserved as generative model

Remarks:

- The training is rather unstable, need to carefully synchronize the two steps (many later works in this direction, e.g. Wasserstein GAN)
- Training by standard tools: backpropagation and dropout
- Theoretical analysis provided in the paper
- Generator does not use S directly during training
- Generator performance is difficult to assess quantitatively
- There is no explicit expression for the generator, it is provided in an implicit form -> you cannot compute the likelihood of a sample w.r.t. the learned GAN

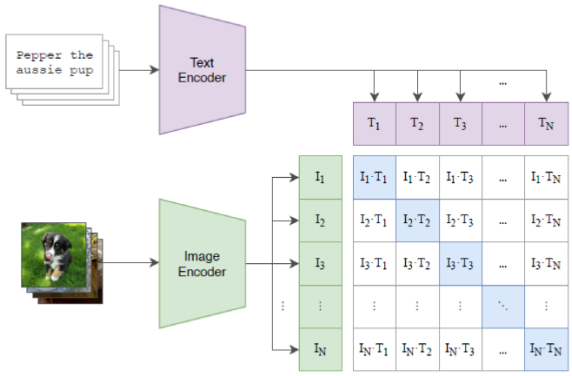
See slides for: MNIST example, Toronto Face Database, CIFAR-10, Vector Arithmetic, GAN for Anomaly Detection, bidirectional GAN

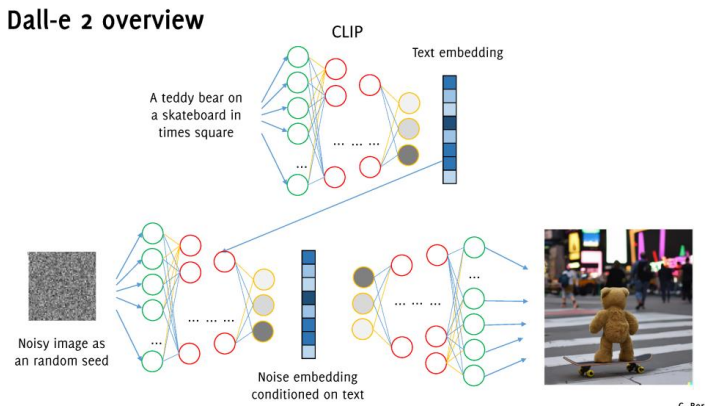
Example:

DALL-E2

DALL-E 2 is a powerful machine learning models from OpenaAl that generates digital images from natural language descriptions It is not based on GANSs but on diffusion models, a more recent generative model. .. but how to feed text to a generative model? We need some help from an external model: CLIP

CLIP Clip is a huge model, trained from image captions to generate consistent representations of both images and text. CLIP is trained by contrastive loss from a training set of images and their caption





DYNAMICAL SERIES - RECURRENT NEURAL NETWORKS

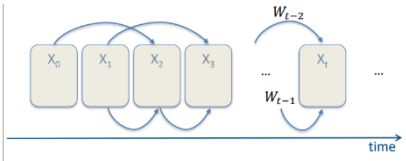
The networks seen so far do not have memory, they can't analyse the order of inputs (= static datasets).
While
Sequential data → the order of inputs is important. The most trivial example of sequential data is time series, but that's not ne only one. Text is a sequential data. The meaning of a text is given by the order of words

SEQUENCE MODELLING

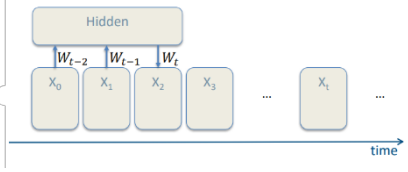
Different ways to deal with «dynamic» data:

Memoryless models:

If we want to predict Xi from previous inputs we can build networks with ficticious inputs by concatenations multiple inputs in the past. This is like a time window approach. We isolate the past with a small window.



• Autoregressive models → Predict the next input from previous ones using «delay taps» = weighted sum of the past.



• Feedforward neural networks → Generalize autoregressive models using non-linear hidden layers,
NB: This is not a real memory

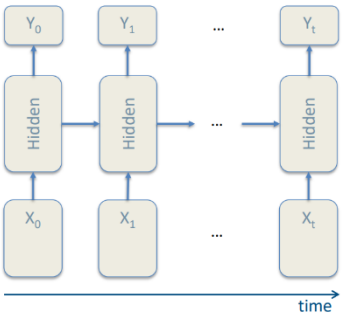
Models with memory:

- Linear dynamical systems
- Hidden Markov models
- Recurrent Neural Networks

DYNAMICAL SYSTEMS

Generative models with a hidden state which cannot be observed directly (stochastic systems)

• The hidden state has some dynamics possibly affected by noise and produces the output
• To compute the output, need to infer hidden state
• Inputs are treated as driving inputs



In linear dynamical systems this becomes:

- State continuous with Gaussian uncertainty
- Transformations are assumed to be linear
- State can be estimated using Kalman filtering
- Linear because we assume that the next state is the linear combination of the previous states.

Memory is often represented as a form of state. The state is the summary of the sequence of input. The state is the summary of way the input has modified the initial state.

$$\begin{cases} X_{t+1} = AX_t + BU_t + \sigma \\ Y_t = CX_t + \xi \end{cases} \qquad \begin{cases} X_{t+1} = f(X,U) + \sigma \\ Y_t = g(X) + \xi \end{cases}$$

Linear version (left), nonlinear version with f and g functions (right)
x = state variable, U = inputs, sigma and epsilon = noises, f = nonlinear function

In *hidden Markov models* this becomes:

- State assumed to be discrete, state transitions are stochastic (transition matrix)
- Output is a stochastic function of hidden states
- State can be estimated via Viterbi algorithm
- Just the state evolving, no input.
- The state is not directly observable, while the output is. The state can be estimated

RECURRENT NEURAL NETWORKS

Linear state space models and Hidden Markow models are stochastic but with RNN the system is DETERMINISTIC.

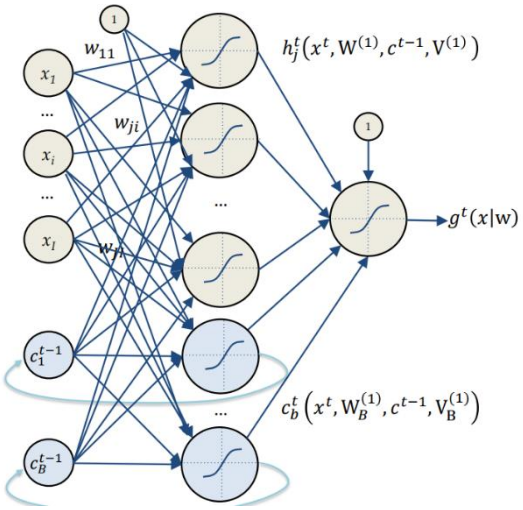
Memory via recurrent connections:

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

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

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

$$c_b^t(\cdot) = c_b\left(\sum_{j=0}^{J} w_{bi}^{(1)} \cdot x_{i,n}^t + \sum_{b\prime=0}^{B} v_{bb\prime}^{(1)} \cdot c_{b\prime}^{t-1}\right)$$



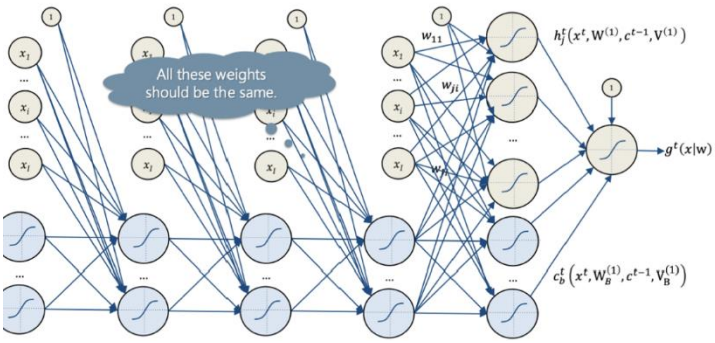
The recurrent part is storing the memory of the input.
The input and the previous value of the state will change the state itself.
Recurrent neurons are function of the current input and the previous output of the recurrent neuron.

For ex the flip flop: basic element of memory in computer and the memory is given by feedback.
A piece of the networks stores the past like a flip flop.

The image represents the most complex version of recurrent networks, it can be easier.

Output at time t= weighed sum over the hidden neuron + weighted sum of the context neuron
The output of the hidden layer = weighted sum of the input + weighted sum of the past context (=recurrent) neuron (this last term is not mandatory)

With this last model there's <u>one big issue related to training</u>:



since backpropagation tries to modify the loss function of the output based on the values of the current input; with this network we should backpropagate also to all the previous inputs cause they all have modified the memory and thus the output!
To backpropagate through time we build a network copying the actual one and repeating it as many times as the steps needed, i.e. all from the starting time instant.

Anyway, we should guarantee that multiple copies of the network related to past values have the same weights since they are copies.
How to do that? There are different ways, the first one is initializing at the same values, perform backpropagation so update the weights and then use the average of the weight matrices. A more efficient way consists in initializing at the same values, computing the gradient and substituting it with the average gradient. Generally, we choose a finite number of steps U through which backpropagate → "truncated" backpropagation over time.

- Perform network unroll for U steps
- Initialize WB, VB replicas to be the same
- Compute gradients and update replicas with the average of their gradients

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

W_B → how you change the state from the input
V_B → how you update the state based on the previous state

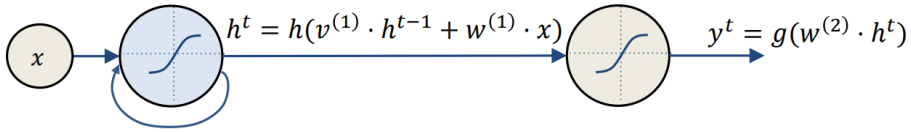
We unroll the network, we decided for a number U of steps and we unroll only for U steps.
U can be decided by us, the number that we consider to be more than enough considering our time series.

The truncation is useful for writing the algorithm, in order to update the weights with the average of gradients. It's not mandatory to truncate, but it's very convenient when the compute everything in parallel with modern GPU. This is convenient when we have inputs of different sizes

How much should we go back in time?
Sometime output might be related to some input happened quite long before:
Jane walked into the room. John walked in too. It was late in the day. Jane said hi to <???>
Text classification: predict if positive or negative sentence. We can have sentences with different lengths.

However, backpropagation through time was not able to train recurrent neural networks significantly back in time (Was due to not being able to backprop through many layers ...).
We could use very efficient activation function (relu) but at that the time they were using sigmoid, and right while performing backpropagation they "discovered" vanishing gradient. (Late 80s).

To better understand why it was not working consider a simplified case:
Let's take the simplest recurrent neural network.
Let's suppose to have a very long sequence.



Backpropagation over an entire sequence S is computed as

$$\frac{\partial E}{\partial w} = \sum_{t=1}^{S} \frac{\partial E^t}{\partial w} \quad \frac{\partial E^t}{\partial w} = \sum_{t=1}^{t} \frac{\partial E^t}{\partial y^t} \frac{\partial y^t}{\partial h^t} \frac{\partial h^t}{\partial h^k} \frac{\partial h^k}{\partial w} \quad \frac{\partial h^t}{\partial h^k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{i=k+1}^{t} v^{(1)} h'(v^{(1)} \cdot h^{i-1} + w^{(1)} \cdot x)$$

If we consider the norm of these terms

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

If $\gamma v \cdot \gamma h' < 1$this converges to 0 ...
With Sigmoids and Tanh we have vanishing gradients

How much can we backtrack in time?
γv = norm of the weights
$\gamma h'$= norm of the derivative of the activation function

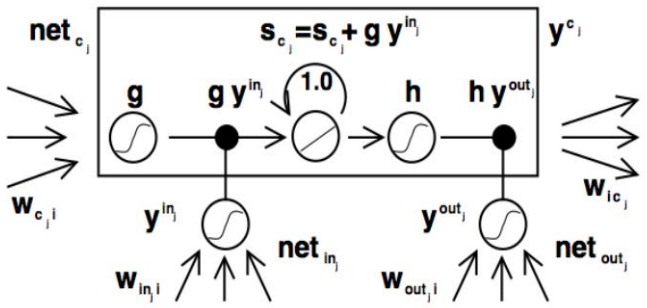
If the product between the 2 gamma is < 1 the derivative will go to zero with standard small weights (they are initialized around zero with a certain variance, the classical initialization)!
Vanishing gradient has been studied for recurrent network and not for feed forward neural network!
Big weights have another problem though, so we cannot use them either cause if the product will be >> 1 the gradient would simply explode and the model would not train…

Solution: change the activation function → ReLU (this is a partial solution)
With relu we have an improvement, we can learn in principle up to 100 of steps.
Even if we use a linear activation function is not enough because it can lead to exploding gradient.
The only way to remove the exploding and the vanishing gradient effects is to guarantee that the product of the recurrent weight and the derivative is =1.

Nb: the basic recurrent neural network works fine if we use short sequences and take some precautions on weights and on activation function.

Long Short-Term Memories (LSTM)

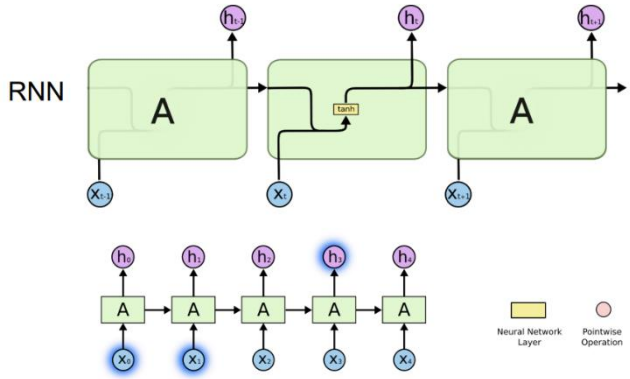
Hochreiter & Schmidhuber (1997) solved the problem of vanishing gradient designing a memory cell using logistic and linear units with multiplicative interactions:
• Information gets into the cell whenever its “write” gate is on.
• The information stays in the cell so long as its “keep” gate is on.
• Information is read from the cell by turning on its “read” gate.



We can backpropagate through this since the loop has fixed weight.

LSTM adds the concept of gates which are represented as black dots on the scheme.
First of all they introduced a linear neuron with unitary recurrent connection (dealy of 1 time step) and called the associated loop “constant error carousel” CEC to say that it does not have any parameters and it is always unitary as well as its derivative. The gate open and close access to CEC.
Since it was simply summing inputs, they added also the “gates” which were feed forward neurons used to decide if we want to add an element or not.

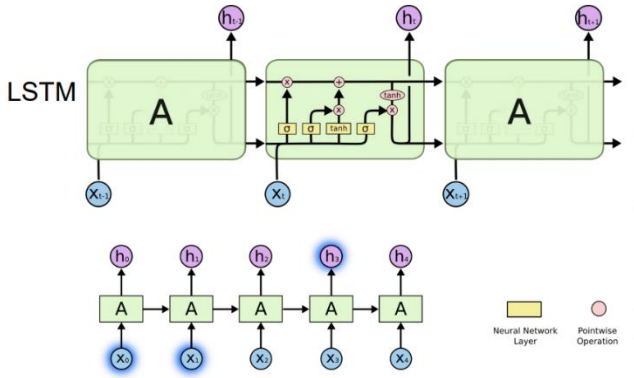
RNN vs LSTM



$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1}$$

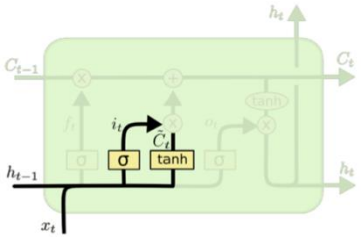
$$b_h^t = \theta_h(a_h^t)$$

The activation of the recurrent cell is equal to a nonlinear function (tanh in the example) of the weighted sum of the input + the weighted sum of the previous state.



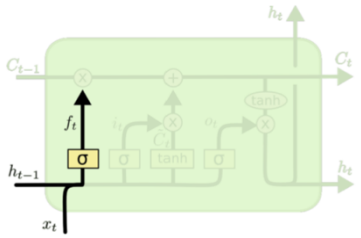
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$
$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

LSTM is slightly more complex. there are 3 gates given by sigmoidal functions (sigma): when the output of sigma is 0 the gate is closed (no signal move forward) while when it is 1 the gate is open (signal can pass).



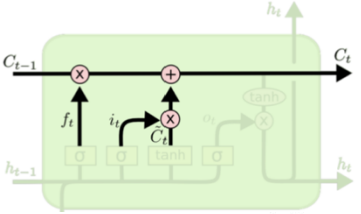
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Input gate:
What to write into memory is called C tilde and it is some nonlinear function of the current input and of the previous output.
The decision on whether to write or not C tilde is taken trough the sigma.



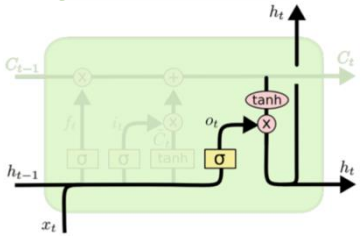
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Forget gate:
At a certain point it can decide to delete from the memory to not accumulate too many inputs. This is also done through a gate and a sigma function.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Memory gate:
Trough the input gate and the forget gate the cell performs writing and erasing of the memory.

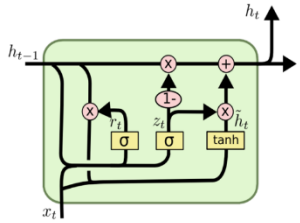


$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Output gate:
Output is some nonlinear function (e.g. tanh) of the content of the network but we might decide to get or not an output through another gate.

Gated Recurrent Unit (GRU)

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.

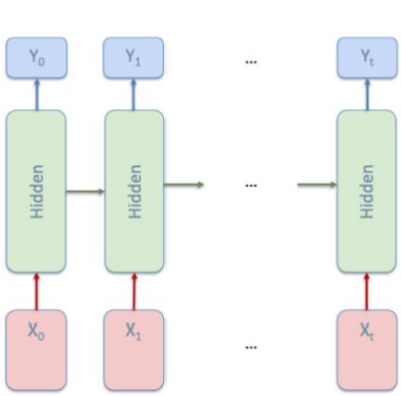


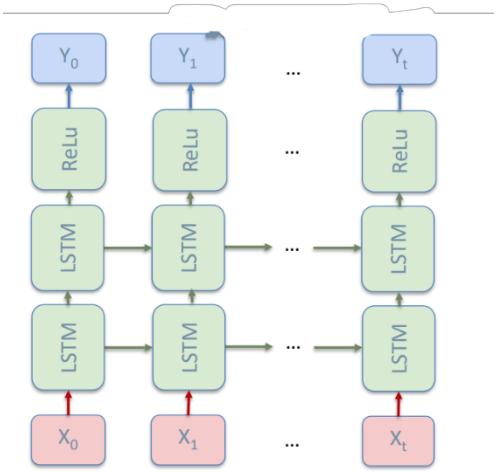
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

They are said to be more effective than LSTM, especially when there is a limited amount of data

LSTM Networks and Bidirectional LSTM Networks

You can build a computation graph with continuous transformations.





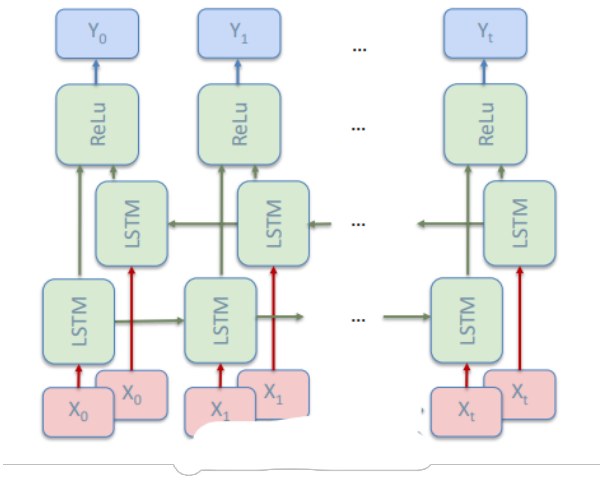
In the simplest case we only have a sequence of input (red) and through a unique hidden layer we obtain a sequence of output.

In more complex cases we could have more than a unique hidden layer. The first LSTM layer summarizes the signal at the level of the high frequencies while the second one summarizes the summary made from the previous layer at the level of the low frequencies → hierarchical representation.

Different architectures can be possible with skip connections till the nonlinear activation (ReLU).

Bidirectional

If we have directly the entire sequence and not one sample at the time we could read the sequence from the left to the right but also form the right to the left to have a clearer interpretation using 2 LSTM.



When conditioning on full input sequence Bidirectional RNNs exploit it:

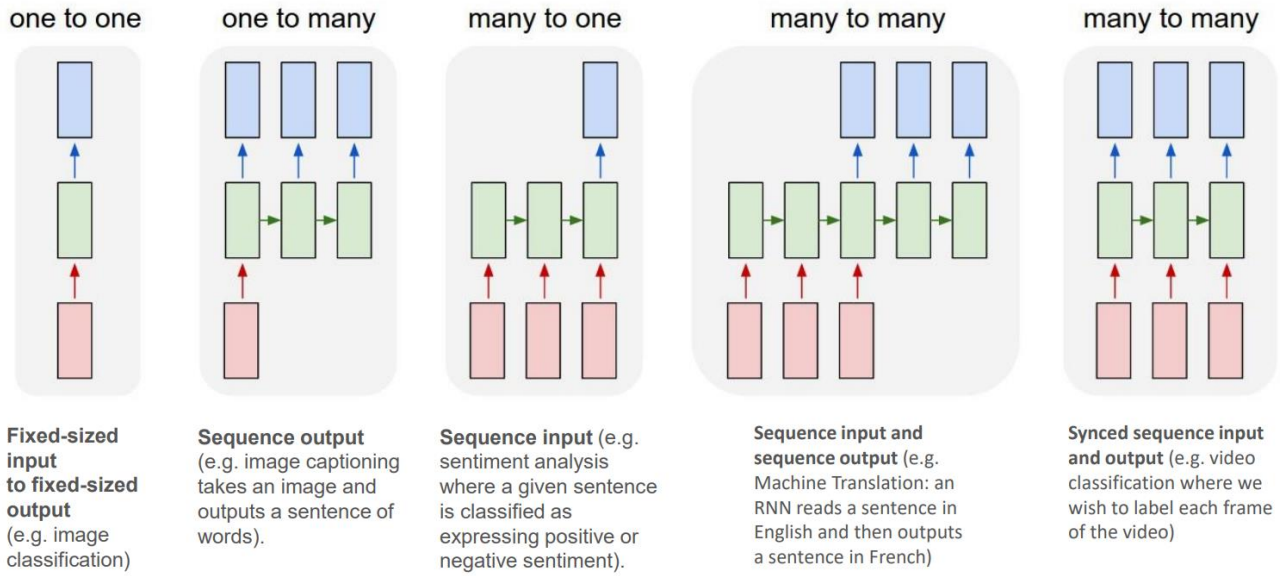
- Have one RNNs traverse the sequence left-to-right
- Have another RNN traverse the sequence right-to-left
- Use concatenation of hidden layers as feature representation

Tips and Tricks

When initializing RNN we need to specify the initial state

- Could initialize them to a fixed value (such as 0)
- Better to treat the initial state as learned parameters
    - Start off with random guesses of the initial state values
    - 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
    - Update these parameters by gradient descent

SEQUENCE TO SEQUENCE LEARNING

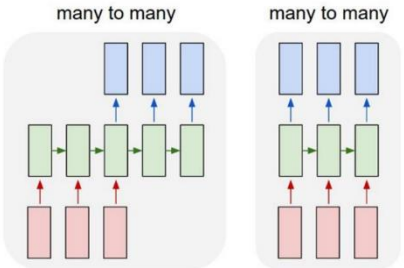


In classification of sequences such as text, we do not have an output for each word (input) but we may want to classify the text after reading a certain amount of words so we would have an output not after each single input word. In other cases, such as in image caption we could also have one input corresponding to many output (second case) or many input corresponding to many output like in Sentiment Classification (third case).

Examples:

- *Image captioning* (one to many): 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 varying length
- *Sentiment Classification/Analysis* (many to one): input a sequence of characters or words, e.g., a tweet, and classify the sequence into positive or negative sentiment. Input has varying lengths; output is of a fixed type and size

  This is the classical application of bidirectional LSTM cause the sequence is read from both left and right.
- *Language Translation* (many to may): 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 varying lengths for the same sentence.

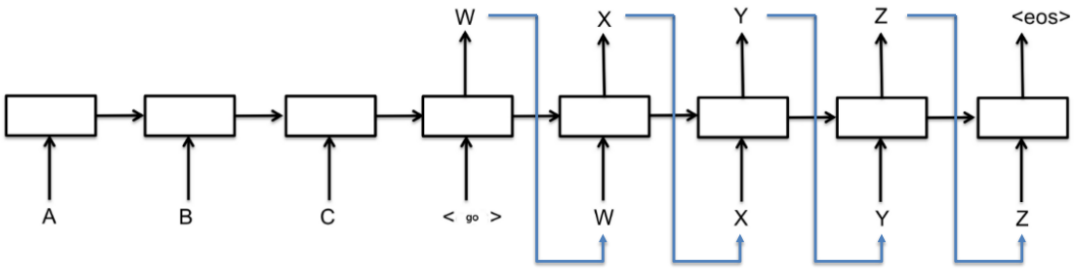


Language translation is the classical case where a sequence of inputs corresponds to a sequence of outputs; we could have one output at the time (sample by sample – left) or all the sequence together (sequence 2 sequence problem - right).

In some cases, the second approach could be more effective cause it summarize the text provided in input in a sort of "state" before

providing the outputs (translation of the summarized version of the text and not the translation word by word that the second network would have made).

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 (in the image – blue arrows)

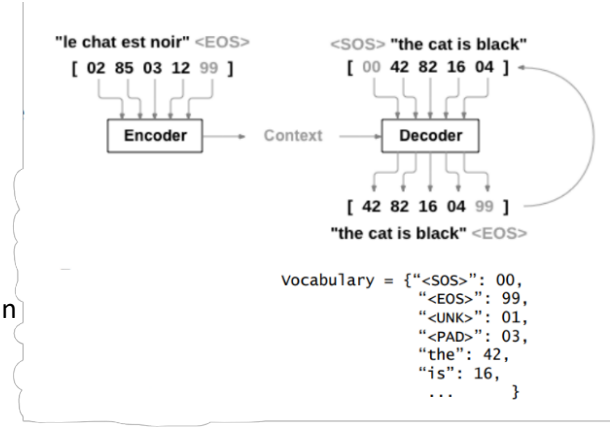


Each output of the translation is given n input to the following cell of the network (blue arrows) but the recurrent part lies on the horizontal connection among all the cells of the network. Translation goes on till all the words are transduced and at the end the output "eos" = end of sequence is provided. In this case bidirectional configuration would be too complex, thus in general bidirectional is not applied to seq2seq model. If we encode each possible word in a sort of dictionary we are able to perform the training using crossentropy as loss function (many possible outputs which are considered like numbers).

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; we should pad also the output if we want it to be always of the same length.
<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. (sos = start of sequence)

DATASET BATCH PREPARATION

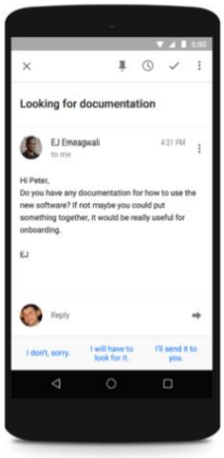
1. Sample batch_size pairs of (source_sequence, target_sequence).
2. Append <EOS> to the source_sequence
3. Prepend <SOS> to the target_sequence to obtain the target_input_sequence and append <EOS> to obtain target_output_sequence.
4. Pad up to the max_input_length (max_target_length) within the batch 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.

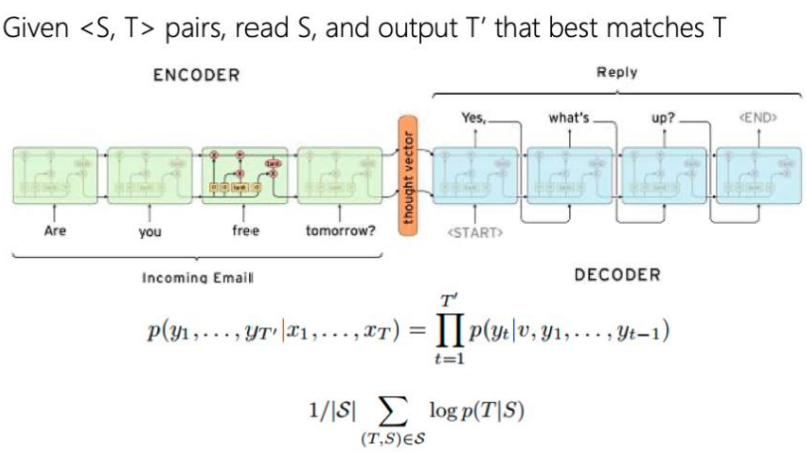


NB: these types of represent text are not effective → we will see better alternatives

SEQUENCE TO SEQUENCE MODELING

Let's assume we want to implement a question-answering problem e.g. chatbot for a bank. In this case the objective is to maximize the probability to have the correct output sequence given the specific input sequence encoded in a vector (context) and the previous predicted output.





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

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

We want to maximize each probability of the sequence of output given the context and the sequence of words received in input.

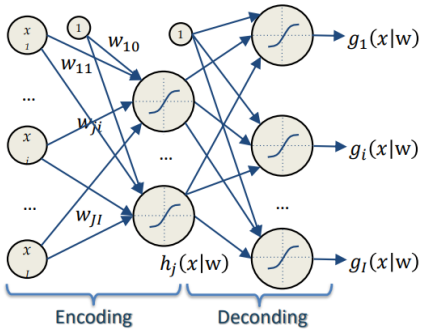
WORD EMBEDDING

NEURAL AUTOENCODER RECALL

Network trained to output the input (i.e., to learn the identity function)

• Limited number of units in hidden layers (compressed representation)

• Constrain the representation to be sparse (sparse representation)



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

$$E = \underbrace{\|g_i(x_i|w) - x_i\|^2}_{\text{Reconstruction error}} + \underbrace{\lambda \sum_j \left| h_j\left(\sum_i w_{ji}^{(1)} x_i\right)\right|}_{\text{Sparsity term}}$$

$g_i(x_i|w) \sim x_i$ $\quad$ $h_j(x_i|w) \sim 0$

Autoencoders: network which tries to reconstruct the input minimizing the difference between the input and the reconstructed output generating in the process a smaller and hidden representation of the input vector. Sometimes we may want to constraint the hidden representation to be sparse so that the model is forced to use as few hidden neurons as possible.

WORD EMBEDDING MOTIVATION

Natural language processing treats words as discrete atomic symbols

• 'cat' is encoded as Id537 (items in a dictionary)

• 'dog' is encoded as Id143

Images are dense data → you move slowly from one image to another changing one pixel at the time cause modifying just one pixel does not change the overall perception of the image.

Sequence of words is a sparse data → if we change one bit of the encoding only we would change completely the semantic and the meaning of the word and probably of the sequence (there are lots of zeroes too)

When each word is encoded with a code/number, all words are distant 1 bit so distance is not informative. Another possible way to encode a word is using a vector as long as the dictionary containing ones at each word included in the text and 0s in all the remaining positions → very sparse representation; this type of representation could be useful to derive if 2 text talk about the same topics cause they could have more less the same words used so similar arrays!

Anyway having a representation with a lot of 0s leads to some problems: we have mostly orthogonal vectors for different texts so that their distance doesn't make any sense; alternatively we should fill a high dimensional space with less data. So how to efficiently represent words? → **ENCODING IS A SERIOUS THING**

Performance of real-world applications (e.g., chatbot, document classifiers, information retrieval systems) depends on input encoding:

Local representations

• N-grams

• Bag-of-words

• 1-of-N coding

They use some features related to the words

Continuous representations

• Latent Semantic Analysis

• Latent Dirichlet Allocation

• Distributed Representations

N-GRAM

try to model the probability to have a certain sequence (e.g. text) in a certain domain (e.g. semantic domain like cooking, school…).

Then we can use this info to make classification.

We cannot consider all the probabilities of the words but those of the last N words.

Determine $P(s = w_1, \dots, w_k)$ in some domain of interest

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

In traditional n-gram language models "the probability of a word depends only on the context of n−1 previous words"

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

Typical ML-smoothing learning process (e.g., Katz 1987):

- compute $\hat{P}(w_i | w_{i-n+1}, \dots, w_{i-1}) = \frac{\#w_{i-n+1}, \dots, w_{i-1}, w_i}{\#w_{i-n+1}, \dots, w_{i-1}}$
- smooth to avoid zero probabilities

Thus, with N-gram model we can model whatever text probability computing for each word in the dictionary what is its probability + that of the previous N-1 words. However, in order to train the model we should consider each single word of the dictionary and compute the probability for that word + the probability of having each possible combination of N-1 words choosing among all the words contained in the dictionary! This could turn to be too computationally expensive…

This was the winning approach in the 90s but it has some problems:

Let's assume a 10-gram LM on a corpus of 100.000 unique words
• The model lives in a 10D hypercube where each dimension has 100.000 slots
• Model training ↔ assigning a probability to each of the 100.00010 slots
• *Probability mass vanishes* → more data is needed to fill the huge space
• The more data, the more unique words! → Is not going to work …

In practice:
• Corpuses can have 10^6 unique words
• Contexts are typically limited to size 2 (trigram model), e.g., famous Katz (1987) smoothed trigram model
• With short context length a lot of information is not captured

N-gram Language Model: Word Similarity Ignorance

Let assume we observe the following similar sentences
• Obama speaks to the media in Illinois
• The President addresses the press in Chicago

With classic one-hot vector space representations

- speaks = [0 0 1 0 … 0 0 0 0]
- addresses = [0 0 0 0 … 0 0 1 0]

speaks ⊥ addresses

- obama = [0 0 0 0 … 0 1 0 0]
- president = [0 0 0 1 … 0 0 0 0]

obama ⊥ president

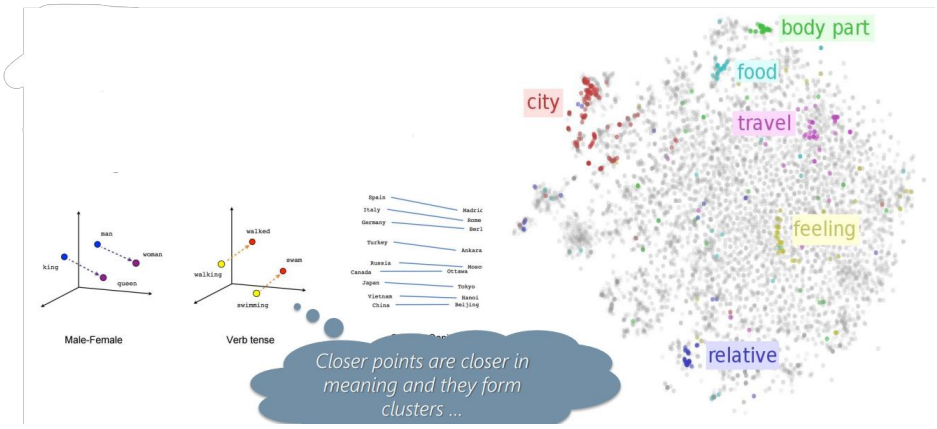
- illinois = [1 0 0 0 … 0 0 0 0]
- chicago = [0 1 0 0 … 0 0 0 0]

illinois ⊥ chicago

Word pairs share no similarity, and we need word similarity to generalize

Embedding

Any technique mapping 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.



Similar words -> close to each other, Different words -> very far apart

We could apply the process related to autoencoders to a sequence of word encoded in a one-hot encoding way in order to have its embedded representation; we are not actually interested in the reconstruction process. What we want is the embedded representation of sentences referred to the same topics to be similar.

WORD EMBEDDING: DISTRIBUTED REPRESENTATION

Autoencoders are trained generally in an unsupervised way; as they are they do not enforce words e.g. with similar meaning to be near each other in the feature mapping → language model + autoencoder. With the language model we set up the problem i.e. we try to predict the probability distribution of a word given the preceding ones. From the autoencoder we can embed the given word in the feature space and then build the language model from this embedding version of the word.

Compression → from one hot encoding to feature vector

Smoothing → moving from one word to another the vector does not change completely as happened with one-hot encoding

Each unique word w in a vocabulary V (typically $||V||$ > 10^6) is mapped to a continuous m-dimensional space (typically 100 < m < 500).

$$w \in V \xrightarrow{mapping\ C} \Re^m$$

obama = [0 0 ... 0 1 0 ... 0 0] (w_1 obama w_V) → obama = [0.12 ... -0.25] (f_1 f_m)

«one-hot» encoding — feature vector

Fighting the curse of dimensionality with:

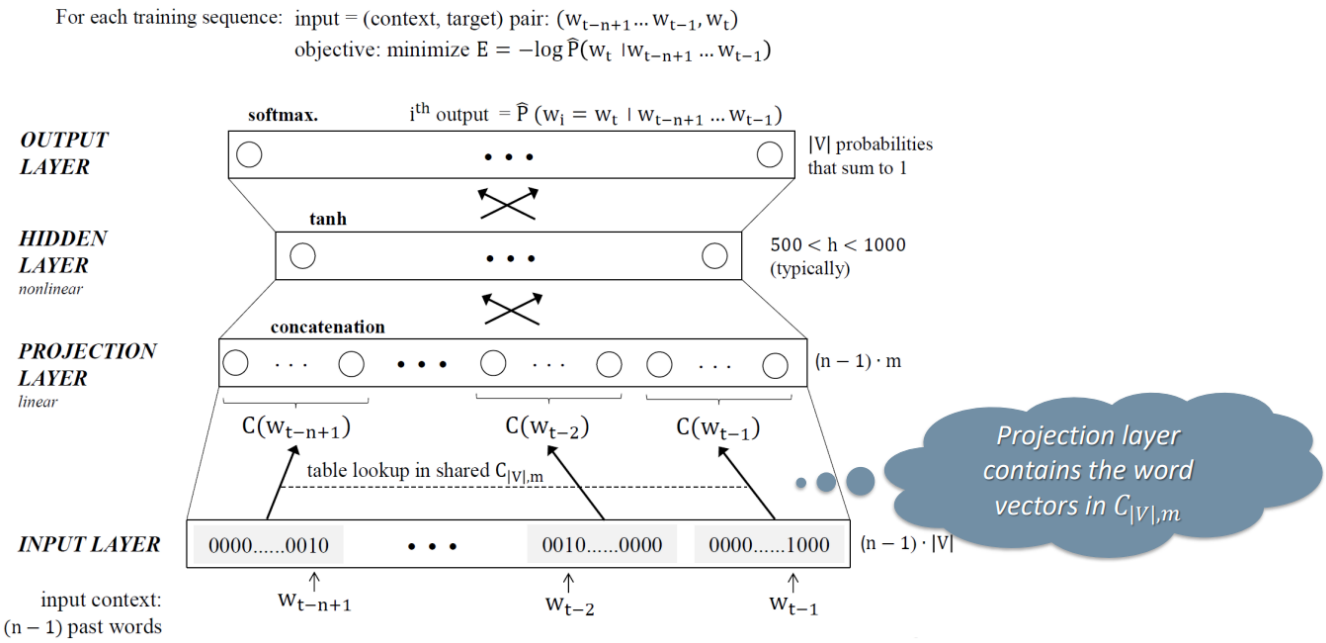
• Compression (dimensionality reduction) -> Similar words should end up to be close to each other in the feature space …

• Smoothing (discrete to continuous)

• Densification (sparse to dense)

Neural Net Language Model (Bengio et al. 2003)

Instead of learning for every possible combination a probability distribution, let's embed the words in a lower dimensional space and learn the probability distribution.



From the bottom we see that the n-1 preceding words expressed in the one-hot encoding way are given in input to the network and are embedded in a lower dimensional space (hidden layer) and then it learns the probability distribution of the next word through a SoftMax function. It is doing the same thing done by the N-grams with a neural network; however, since the probability distribution is learned from a reduced space it is much less computationally expensive!

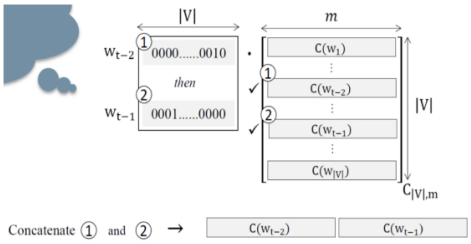
NB: this is an unsupervised training

There are some **tricks** to train the model:

- **Projection**

  → from the concatenation of the one-hot encoding vectors they are projected into an m-dimensional projection layer. This means that each vector is represented by m numbers.

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']



- **Non-linear dense layer**

then we have also a SoftMax to learn the probability of the next word which is based on the output of the previous layer which is nonlinear (tanh function).

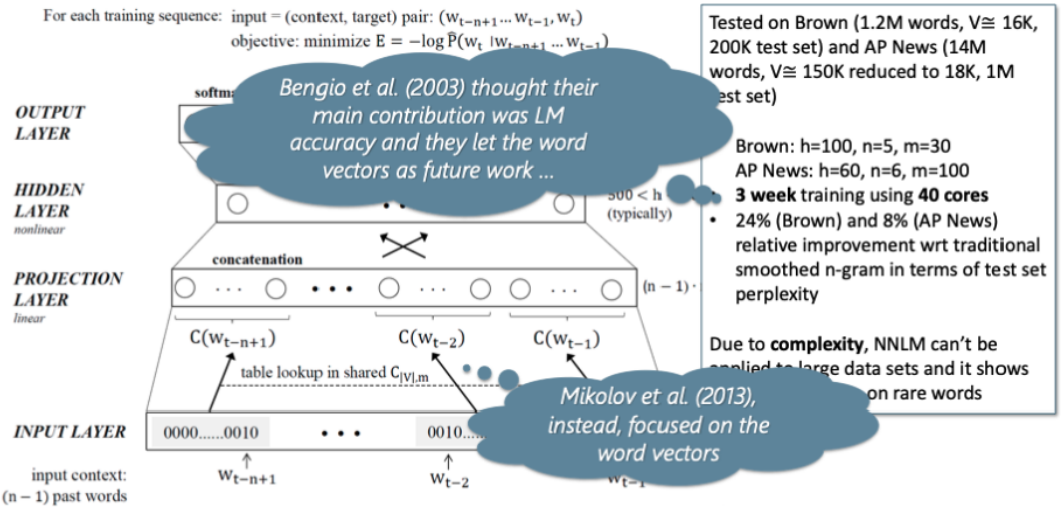
Softmax is used to output a multinomial distribution

$$\hat{P}(w_i = w_t \mid w_{t-n+1}, \dots, w_{t-1}) = \frac{e^{y_{w_i}}}{\sum_{i'}^{|V|} e^{y_{w_{i'}}}}$$

- $y = b + U \cdot \tanh(d + H \cdot x)$
- x is the concatenation $C(w)$ of the context weight vectors
- d and b are biases (respectively h and $|V|$ elements)
- U is the $|V| \times h$ matrix with hidden-to-output weights
- H is the $(h \times (n-1) \cdot m)$ projection-to-hidden weights matrix

The complexity of this model is dominated by the embedding size m and by the size of the vocabulary |V|.

Bengio applied this model to 2 standard datasets at that time. It turns out that with respect to traditional N-grams they improved 24% on the first dataset (Brown, smaller dataset → less complexity) and 8% on the other one (AP News, bigger dataset → more complexity).



Due to complexity NNLM can't be applied to large data sets and it shows poor performance on rare words.

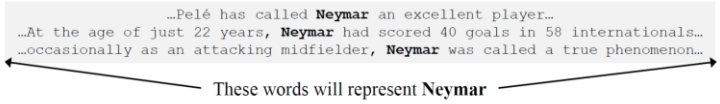
Google's word2vec (Mikolov et al. 2013a)

Idea: 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 weight matrix)
- Context contain words both from history and future

In this case they were not interested in predicting the following word in a text but in embedding. What they did consisted in creating a "context" encoding words preceding and following the word to be predicted and then use this vector to predict the word. If it works that context is the best representation of the word.

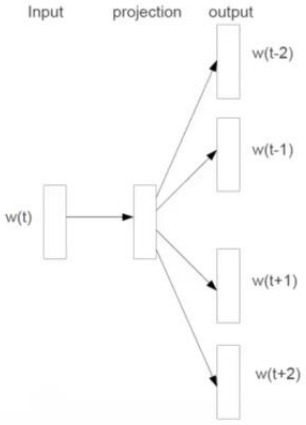
Given the n words surrounding one term -> predict the term.



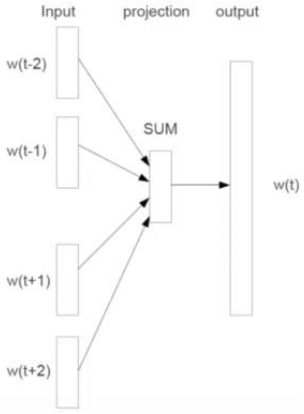
If the contexts of 2 words are very similar this means that probably also those 2 words are very similar.
Conclusion: we can represent a word with the embedding of the surrounding ones!

The approach can be seen in these 2 different ways:

- we give in input a word not with the one-hot encoding but with the continuous vector w(t), if through this vector we are able to predict the context of the word that is a good representation of that word.
- We could also give in input the representation of the preceding and following words and derive the representation of the considered one.



Skip-gram architecture



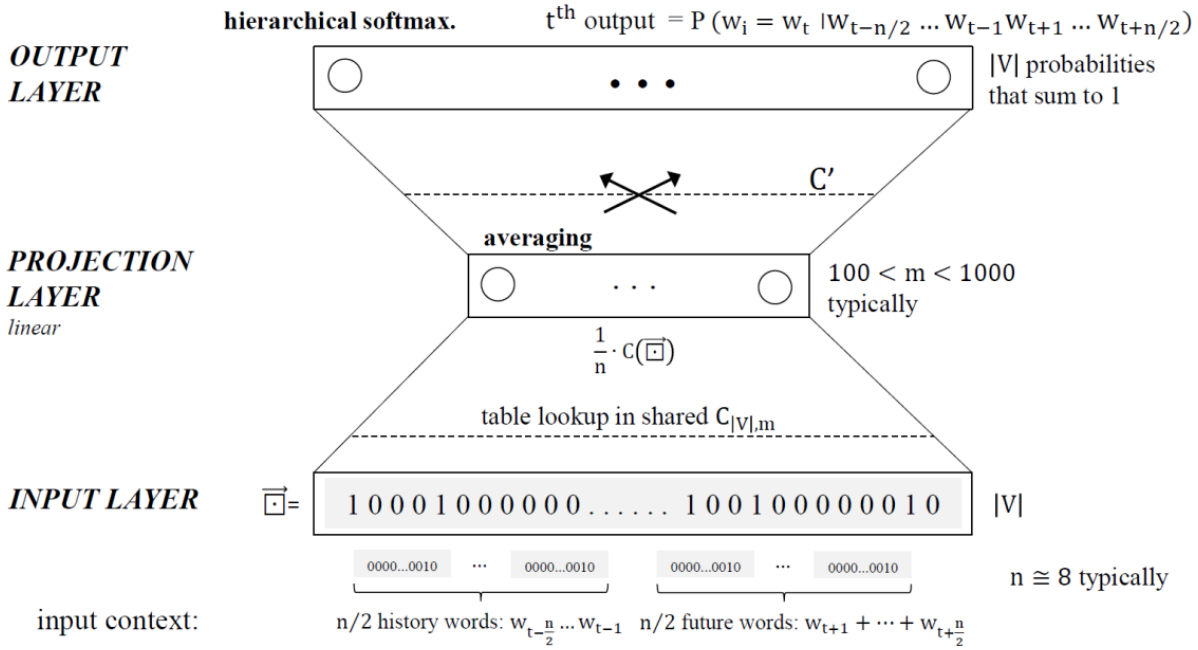
Continuous Bag-of-Words

Word2vec's Continuous Bag-of-Words (CBOW)

Encoder (embedding model) for words that represent each word with a vector of continuous space

For each training sequence: input = (context, target) pair: $(w_{t-\frac{n}{2}} \dots w_{t-1} w_{t+1} \dots w_{t+\frac{n}{2}}, w_t)$

objective: minimize $E = -\log \hat{P}(w_t \mid w_{t-n/2} \dots w_{t-1} w_{t+1} \dots w_{t+n/2})$



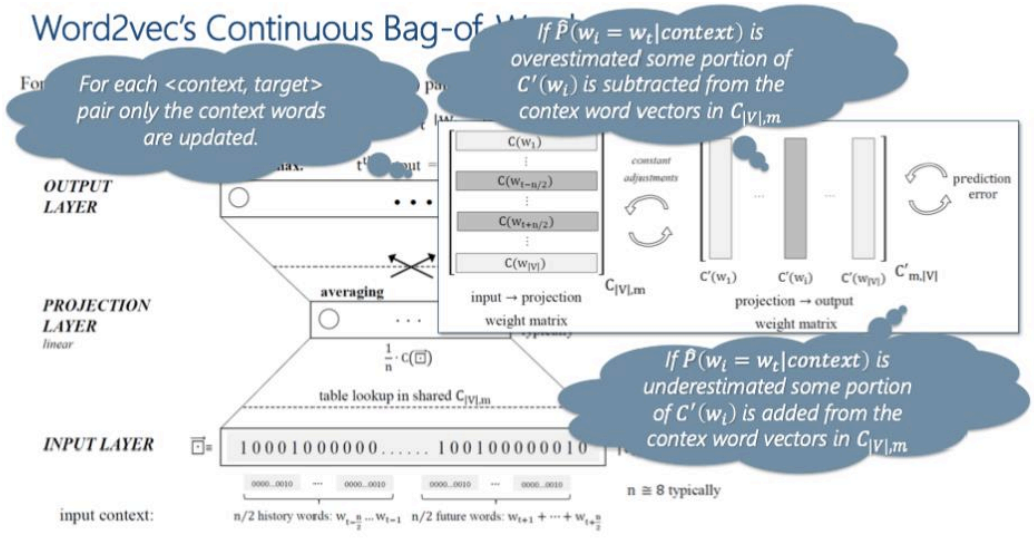
Procedure: binary encoding, table lookup shared among n words, instead of concate nation you will use averaging because it is smaller in a way that the result is m, not m∗n (? not sure about that). From this average you predict one of the words.

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.

The average encoding of the surrounding context represents the semantic context where usually the word is used → semantic of the term. Two words that are used in the same context are close together wrt other terms that does not belong to the same semantic context.

Tricks for training:



Word2vec shows significant improvements w.r.t. the NNML

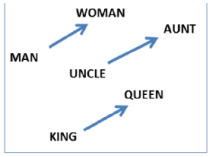
- Complexity is $n \times m + m \times log|V|$ (Mikolov et al. 2013a)
- On Google news 6B words training corpus, with $|V| \sim 106$
    - CBOW with m=1000 took 2 days to train on 140 cores
    - Skip-gram with m=1000 took 2.5 days on 125 cores
    - NNLM (Bengio et al. 2003) took 14 days on 180 cores, for m=100 only!
- word2vec training speed ≅ 100K-5M words/s
- Best NNLM: 12.3% overall accuracy vs. Word2vec (with Skip-gram): 53.3%

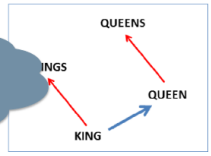
NB: it is independent from the size of the given context, cause we make the average

Regularities in embedding

- If we project the points we see that all the capitals are close between themselves and the counties are close between themselves and there is a constant distance between every capital and its country.
- Constant female-male difference vector
- Vector operations are supported make «intuitive sense»:

- $w_{king} - w_{man} + w_{woman} \cong w_{queen}$
- $w_{paris} - w_{france} + w_{italy} \cong w_{rome}$
- $w_{windows} - w_{microsoft} + w_{google} \cong w_{android}$
- $w_{einstein} - w_{scientist} + w_{painter} \cong w_{picasso}$
- $w_{his} - w_{he} + w_{she} \cong w_{her}$
- $w_{cu} - w_{copper} + w_{gold} \cong w_{au}$
- …





«You shall know a word by the company it keeps» John R. Firth, 1957:11.

from: scribd.com/document/285890694/NIPS-DeepLearningWorkshop-NikhilGa...

APPLICATIONS OF Word2vec
Information Retrieval
Document Classification/Similarity
Sentiment Analysis → We can add e.g. "sad" and "happy" to the text and we can simply count how many words are near to sad and how many to happy in terms of cosine distance as seen before to understand the "sentiment" of the text.

GloVe: Global Vectors for Word Representation (Pennington et al. 2014)

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

Trained by weighted least squares

NB: the training is still unsupervised

No mathematics needed to know here.

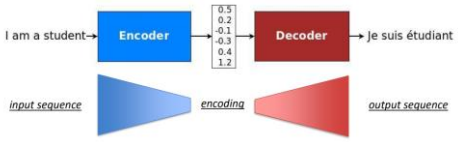
GloVe has been outmoded by better enders in the last years.

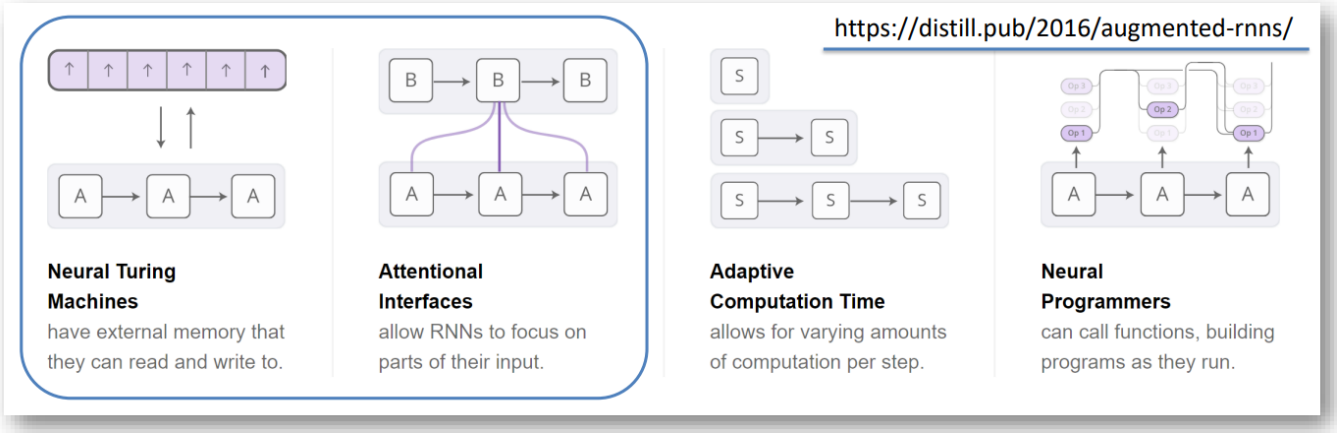
BEYOND SEQ2SEQ ARCHITECTURES

EXTENDING RECURRENT NEURAL NETWORKS

Recurrent Neural Networks have been extended with memory to cope with very long sequences and the encoding bottleneck…

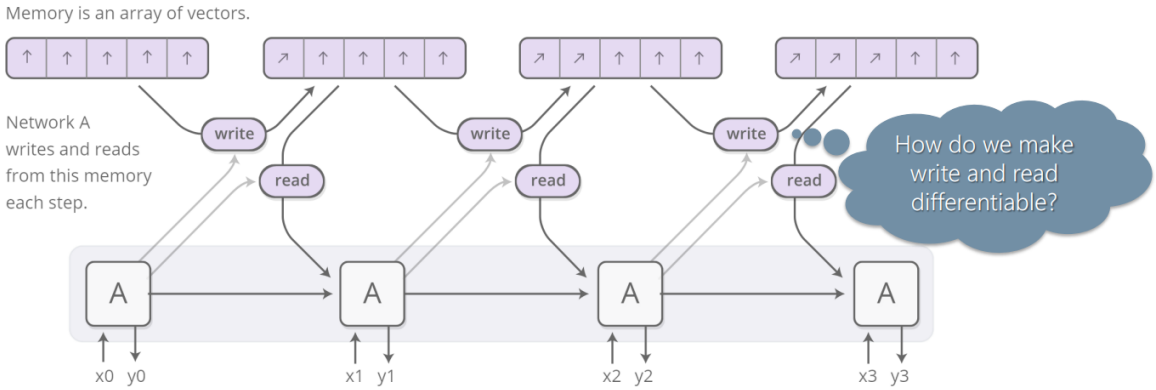
There are different ways to implement memories; we'll see Turing Machines and Attentional interfaces.





NEURAL TURING MACHINES

Neural Turing Machines combine a RNN with an external memory bank



What is the limit of a LSTM? Memory lies in the gates but it is not explicitly available like for external memories.

→ Neural Turing Machines

An external memory is a matrix of vectors where each element corresponds to a cell of the network.

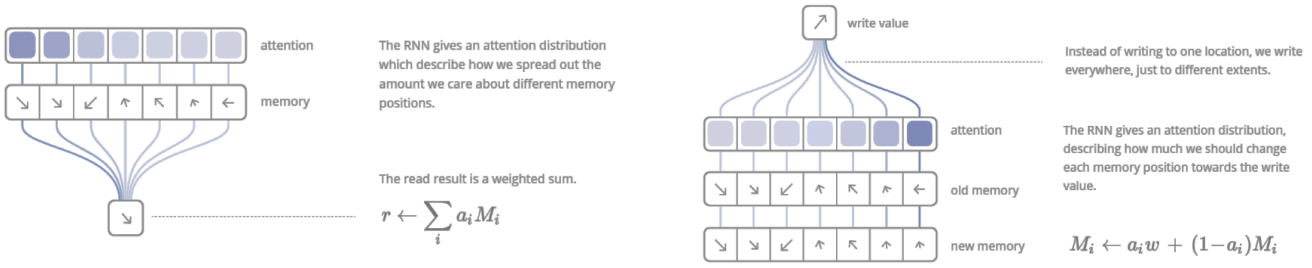
In Turing machines, it is possible to either write on the memory or to read form the memory.

We could have an infinite size, each cell has its own CEC, BUT we have difficult implementation. Why?

Neural Turing Machines challenge:

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

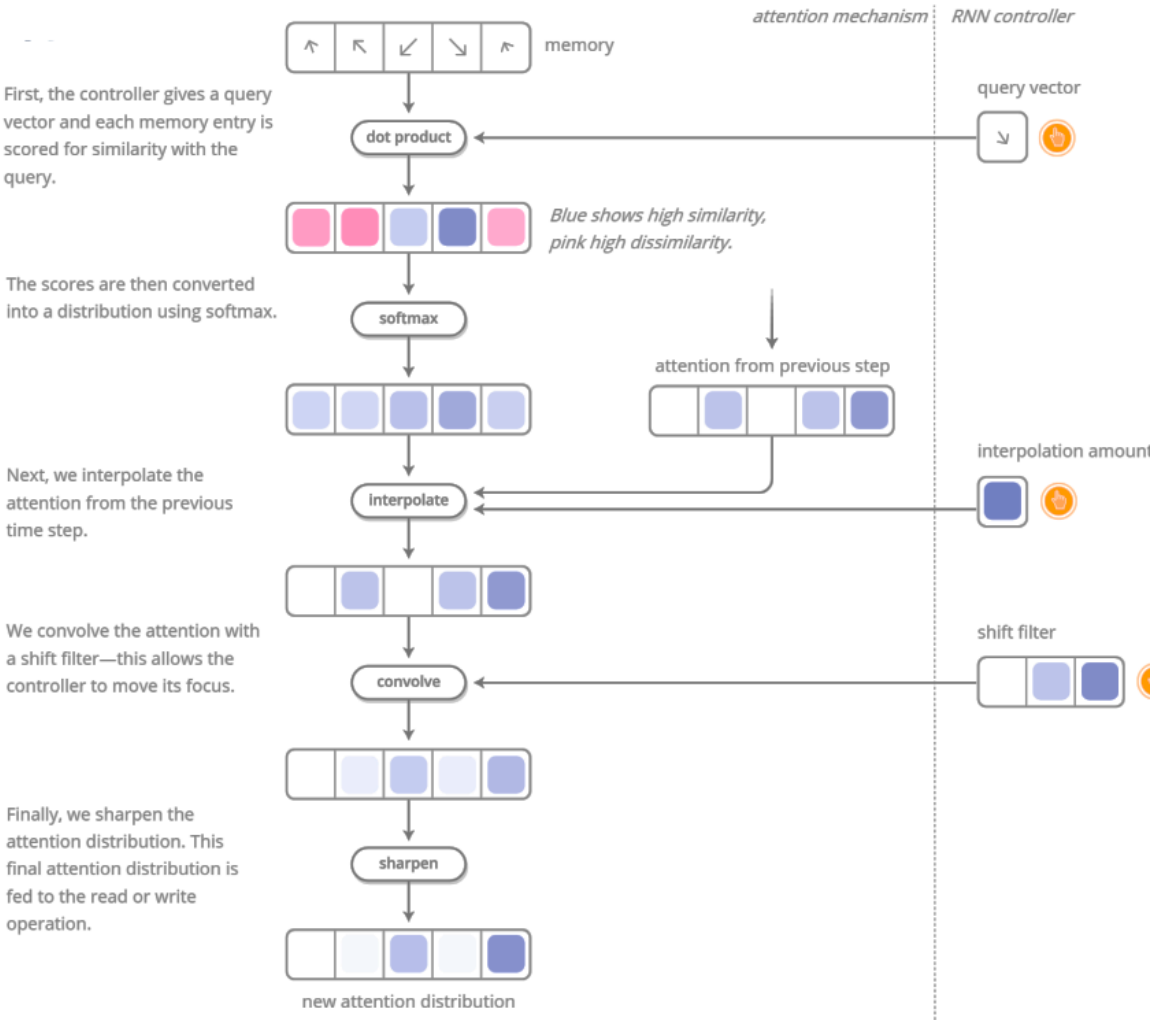
Solution: Go around the problem of differentiating wrt the memory. Every step, read and write everywhere, just to different extents.



The RNN gives an attention distribution which describe how we spread out the amount we are about different memory positions.
Instead of writing in one location we write everywhere, just to different extents. The RNN gives an attention distribution describing how much we should change each memory position towards the write value.
NB: every vector changes a little bit by the writing, but the one that is changed the most is the focus of the attention.

Attention mechanism:
we read/write every cell but put the main attention to one single cell which is that to which we would refer only. In this way we are not forced to deal with a single cell.

Content-based mechanism: searches on the memory for cells with a content similar to that of the input, this generates a focus of attention based on the content.
Location-based mechanism: it is used to move around the area where we focused.



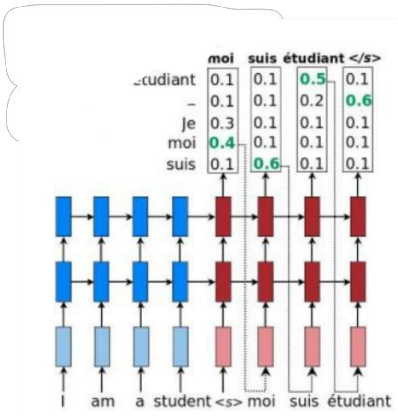
Content-based attention (first part): searches memory and focus on places that match what they're looking for
Location-based attention (second part): allows relative movement in memory enabling the NTM to loop.

NTM perform algorithms, previously beyond neural networks:
• Learn to store a long sequence in memory
• Learn to loop and repeat sequences back repeatedly
• Learn to mimic a lookup table
• Learn to sort numbers …

Some extensions have been proposed to go beyond this (but the most interesting is the attention mechanism):
• Neural GPU overcomes the NTM's inability to add and multiply numbers
• Zaremba & Sutskever train NTMs using reinforcement learning instead of the differentiable read/writes used by the original
• Neural Random Access Machines work based on pointers
• Others have explored differentiable data structures, like stacks and queues

ATTENTION MECHANISM IN SEQ2SEQ MODELS

In Seq2Seq models, given a sequence of input, we should predict the sequence of output. In the model we have seen so far the way to do that starts encoding the first sequence. The problem is that it is ok for short-medium sequences but it does not work for long sequence. This is because we want to memorize the input through its encoded version. Having also a memory of the original sequence would be much more effective.
Thus, we would like to embed the text into a vector which somehow stores the general meaning of the text and then on the base of that we should focus through the attention mechanism on the part to be translated.



Considering the sequential dataset:

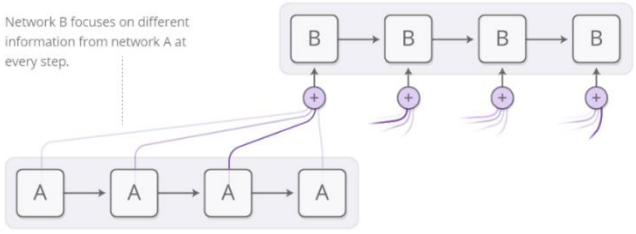
$$\{((x_1,\ldots,x_n),(y_1,\ldots,y_m))\}_{i=1}^{N}$$

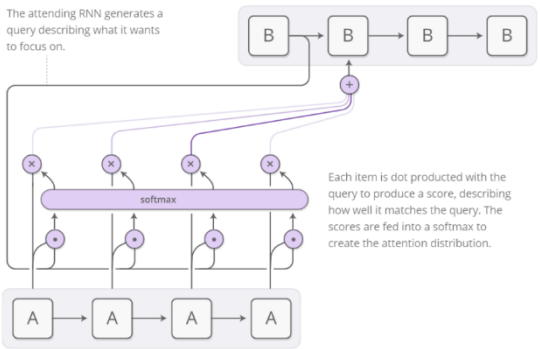
The decoder role is to model the generative probability

$$P(y_1,\ldots,y_m|x)$$

In "vanilla" seq2seq models, the decoder is conditioned initializing the initial state with last state of the encoder.
Works well for short and medium-length sentences; however, for long sentences, becomes a bottleneck.
Like for translation: it is better to read a whole paragraph and then translate rather than doing it word by word → we need space for long sentences.

We use the input as a memory, the state as a sort of trigger of focus and we have an attention mechanism.
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





Network B focuses on different information from network A at every step. The attending RNN generated a query describing what it wants to focus on. Each item is dot producted with the query to produce a score, describing how well it matched the query. The scores are fed into a softmax to create the attention distribution.

We have a text, we encode the text in some softmax, with these softmax we decide which element of the memory has the focus of our attention.

Implementation mechanism

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 ht , with source states hs to derive attention.

$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \boldsymbol{W} \bar{\boldsymbol{h}}_s & \text{[Luong's multiplicative style]} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W_1} \boldsymbol{h}_t + \boldsymbol{W_2} \bar{\boldsymbol{h}}_s\right) & \text{[Bahdanau's additive style]} \end{cases}$$

What are the most relevant vectors in the past attributing them a score

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

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

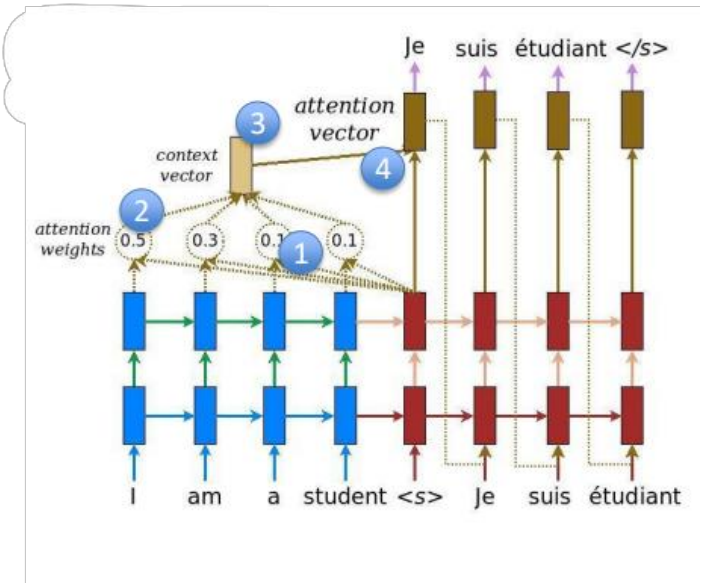
(⇔ definition of the weights of our reading)

3. Compute the context vector as the weighted average of the source states (which is the summary of the memory)

$$\boldsymbol{c}_t = \sum_s \alpha_{ts} \bar{\boldsymbol{h}}_s$$

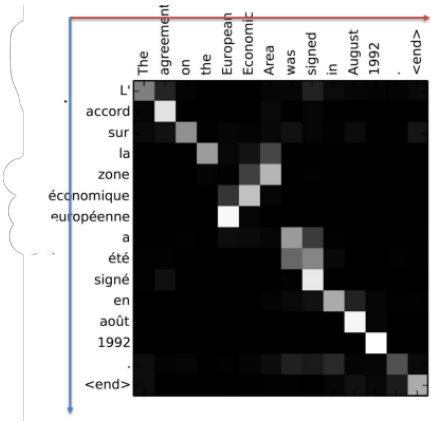
4. Combine the context vector with current target hidden state to yield the final attention vector

$$\boldsymbol{a}_t = f(\boldsymbol{c}_t, \boldsymbol{h}_t) = \tanh(\boldsymbol{W_c}[\boldsymbol{c}_t; \boldsymbol{h}_t])$$



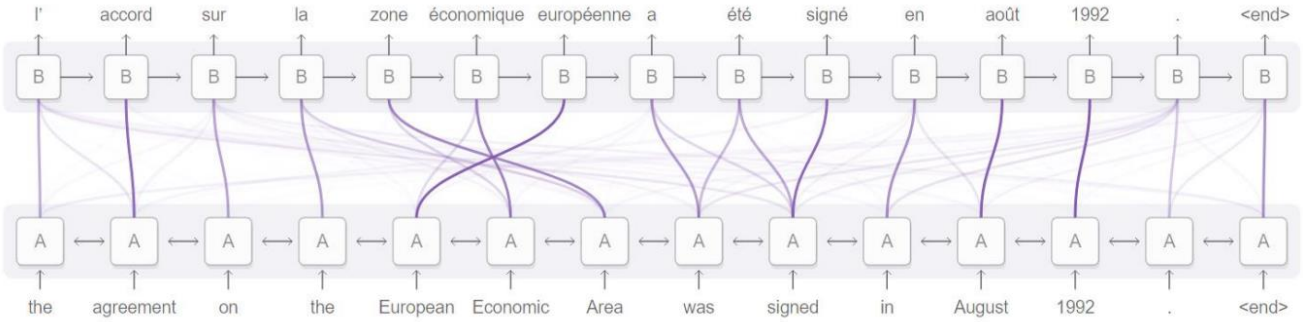
ATTENTION VISUALIZATION

Alignment matrix is use to visualize attention weights between source (horizontal) and target (vertical) 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 Mechanism in Translation

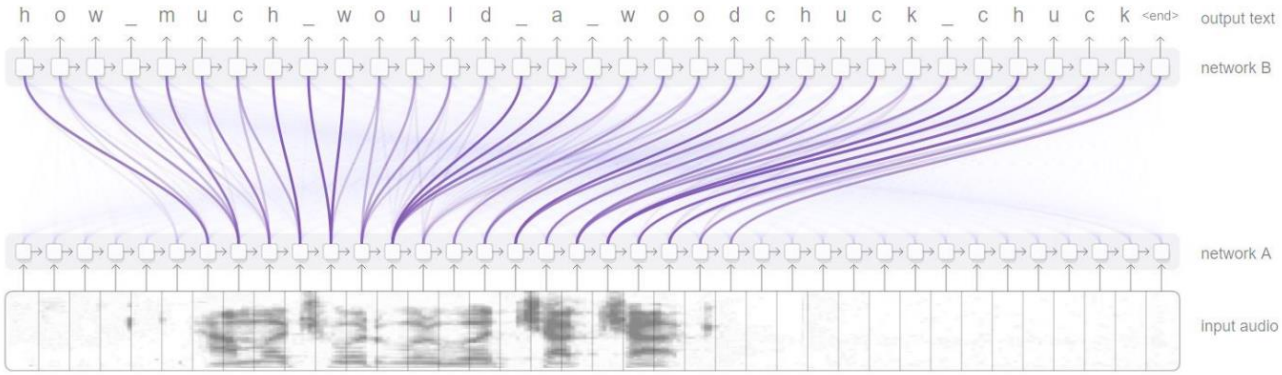
Attention allows processing the input to pass along information about each word it sees, and then for generating the output to focus on words



The group of words European Economic Area is translated as a group as noticeable from the weights associated to them.

Attention Mechanism in 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



Audio input

→ network embedding the audio

→ network that makes the representation of the embedding of the audio

→ this representation is then used to focus on pieces of the audio.

Attention Mechanism in 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.