Neural Networks Course

José María Lago Alonso

February 20, 2018

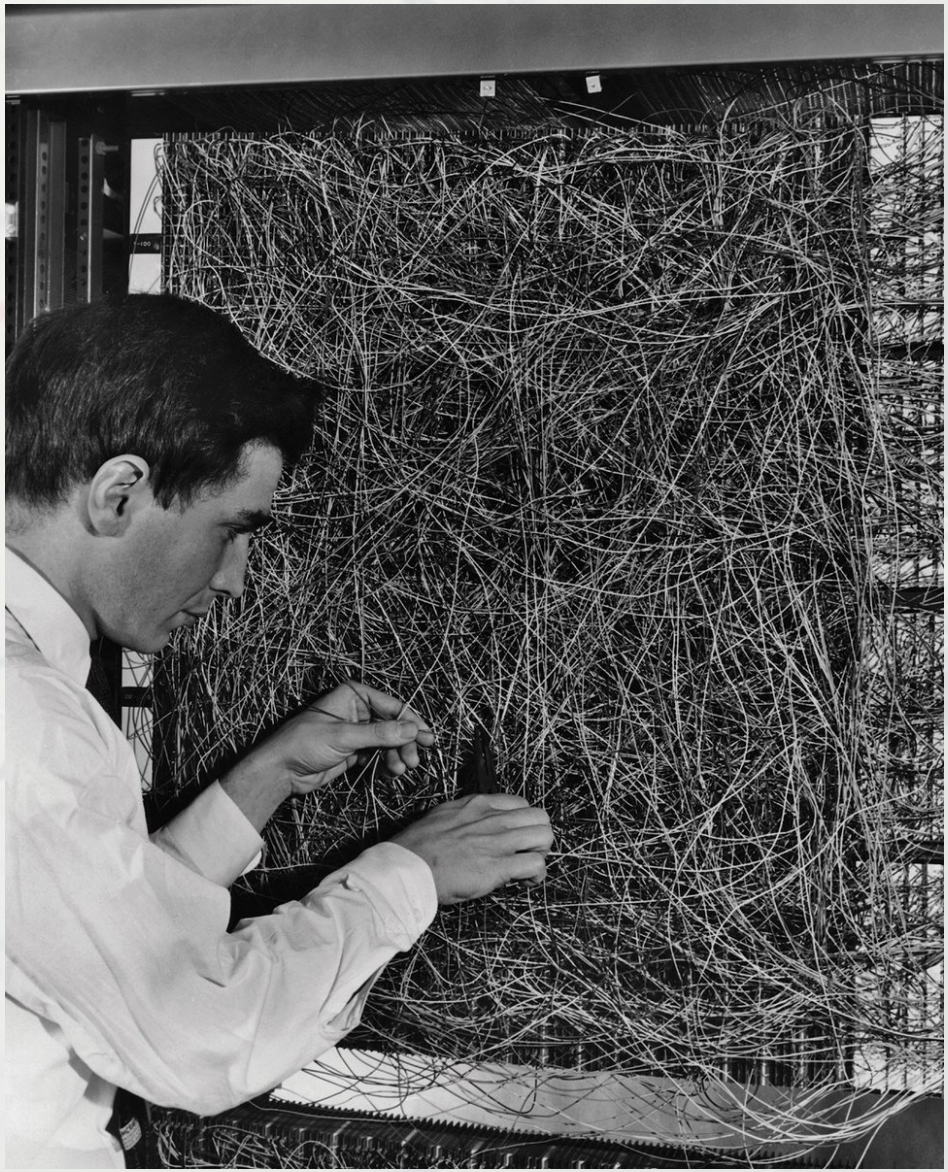
NN world

History

In 1957, Frank Rosenblatt developed the Perceptron Algorithm at the Cornell Aeronautical Laboratory. In 1986, David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams published a paper in Nature magazine where they described the back-propagation algorithm.

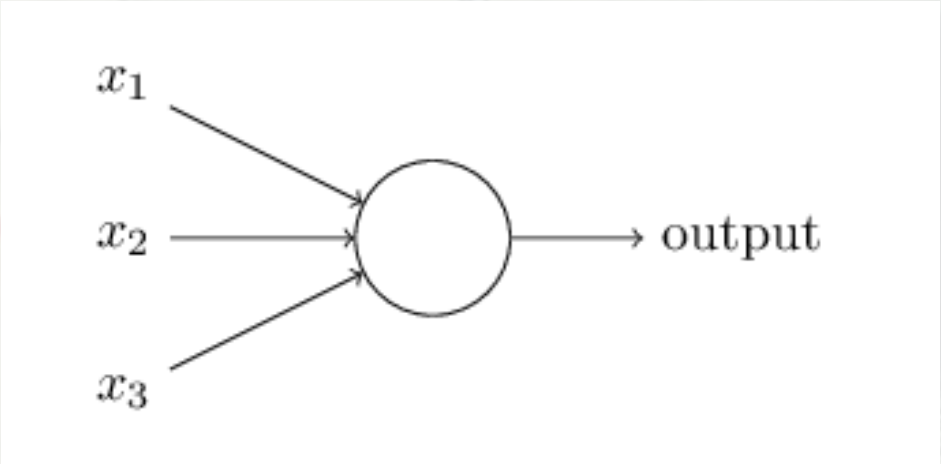
- DNN (Deep Neural Network) predecessor of all others, can be used in any dataset.
- CNN (Convolutional Neural Network) are widely used for image recognition and image segmentation.
- RNN (Recurrent Neural Network) are used in Natural Language Processing world for example in text auto-completing.
- GAN (Generative Adversarial Network) it is a network that **improves the results of other networks**
- Many many others...

First NNs



Neurons

How a neuron works?

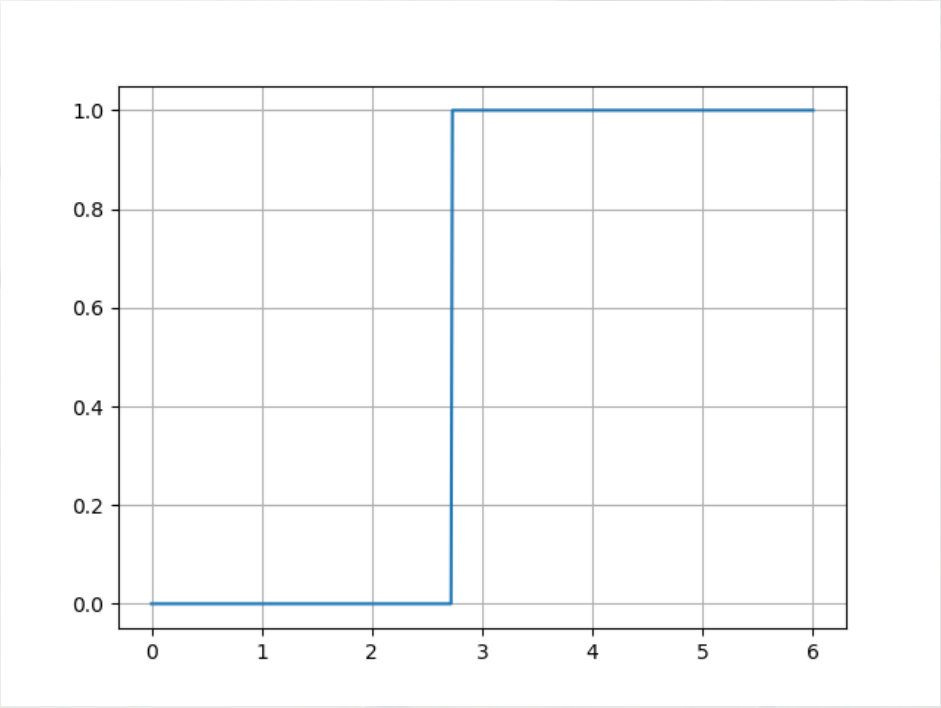


Kinds of neurons

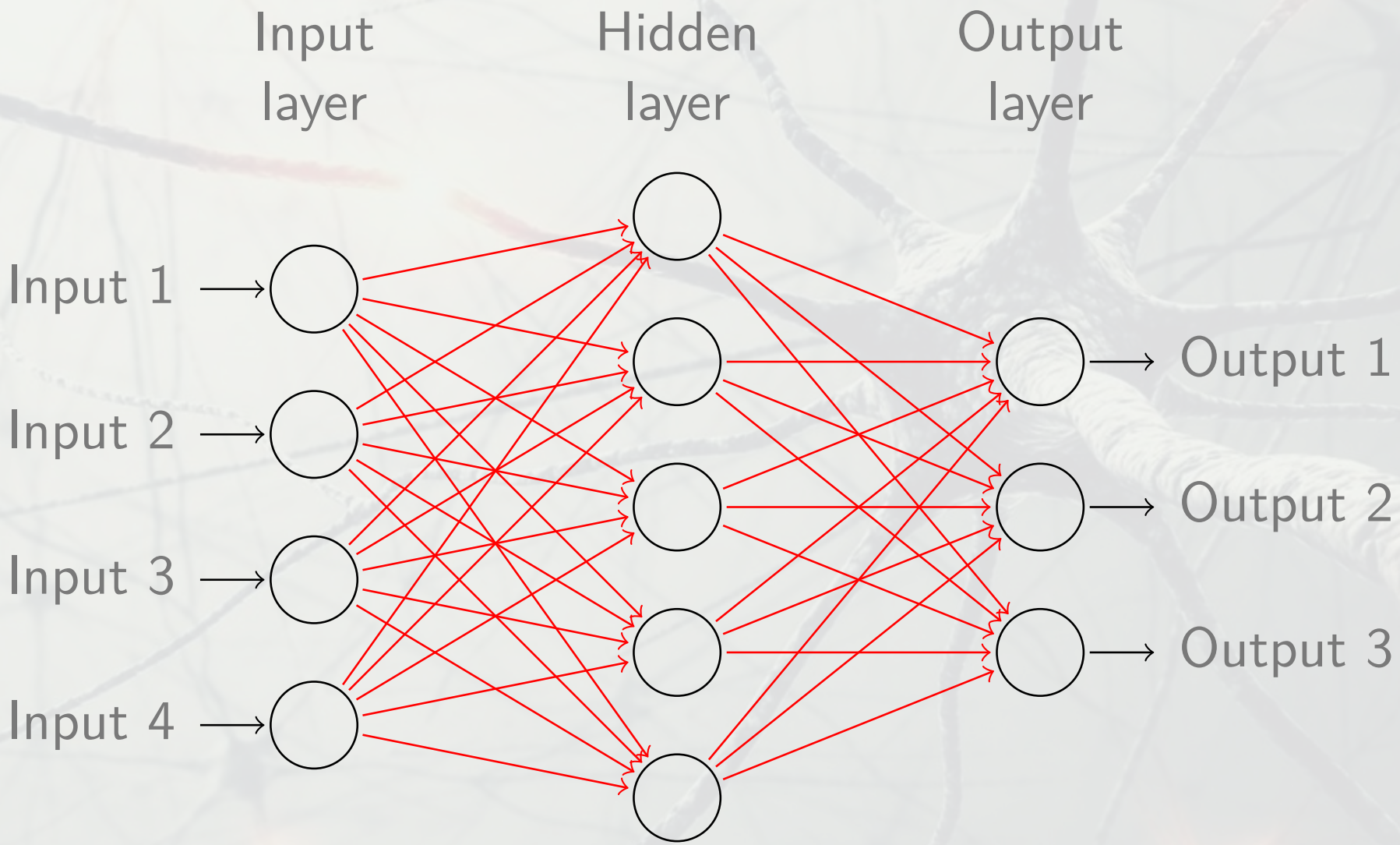
- **Perceptrons**
- Sigmoids
- Tanhs
- ReLUs
- ...

Perceptrons

```
def perceptron(x,w,threshold):
    if np.sum(x*w) < threshold:## star acts as Hadamard
     ↪  product
        return 0
    else:
        return 1
```



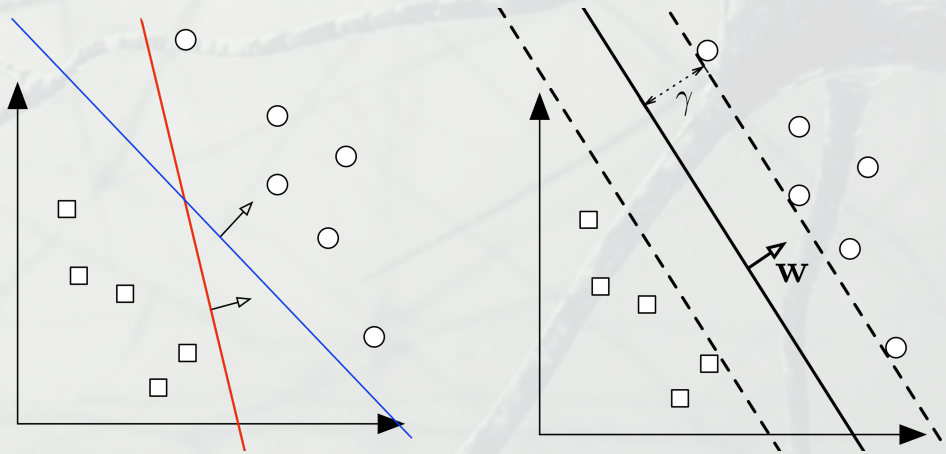
Neural Network Scheme



Basic classifiers statement

Suppose that $(X_1, Y_1), ..., (X_n, Y_n)$ are random variables such $X_i \in \mathbb{R}^d$, $Y_i \in \{-1, 1\}$ and linearly separable. Then we can find an $\omega \in \mathbb{R}^d$ such $\omega^T X_i Y_i > 0$, for all $i \in \{1, 2, 3, ..., n\}$.
We can define a binary classifier as:

$$f_\omega(x) = \begin{cases} -1 & \text{if } \omega^T x > 0 \\ 1 & \text{otherwise} \end{cases}$$



Perceptron Learning Algorithm

Here we explain a method to find a vector ω that follows the properties of the previous slide, in a finite number of steps.
We start with an initial $\omega_0 \neq 0$, and taking the data cyclically

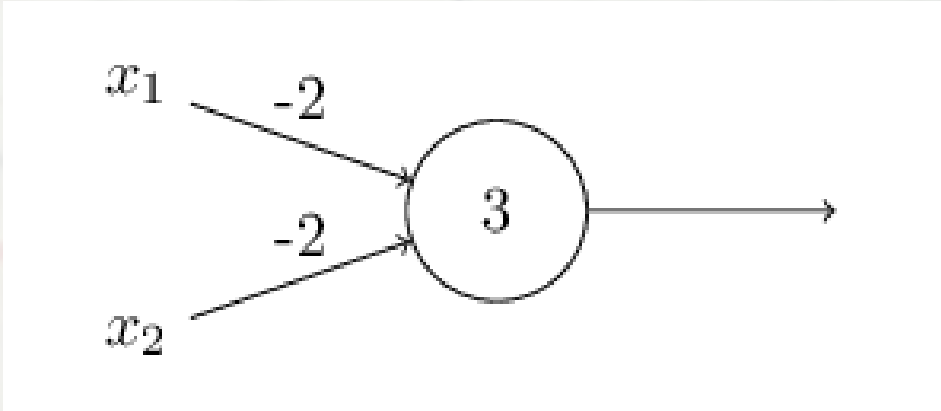
$$(X_1, ..., X_n) \to (X_1, ..., X_n) \to (X_1, ..., X_n) \to ...$$

At each time iteration $t \in \{1, 2, 3, ...\}$

$$\begin{cases} \omega_t = \omega_{t-1} + X_t Y_t & \text{if } \omega_{t-1}^T X_t Y_t < 0 \\ \omega_t = \omega_{t-1} & \text{otherwise} \end{cases}$$

We can think the ω_t as the threshold of our perceptron neuron.
After many t iterations, we can compute some error expression like MSE to check if our network is learning.

How to build a NAND gate with a perceptron



The input $(0, 0)$ produces $0 * (-2) + 0 * (-2) + 3 = 3 > 0$ an output positive, so class 1. The input $(0, 1)$ produces
$0 * (-2) + 1 * (-2) + 3 = 1 > 0$ that is also class 1. Doing similar computations with $(1, 0)$. And $(1, 1)$ produces
$1 * (-2) + 1 * (-2) + 3 = -1 < 0$ that is class 0. So this is in fact a NAND which is a universal gate. Thus perceptrons are universal for computation also. That means we can build all logic gates just using perceptrons.
https://en.wikipedia.org/wiki/NAND_logic

NNs vs Von Neumann

NNs

- Parallel structure
- Learn by example

Von Neumann

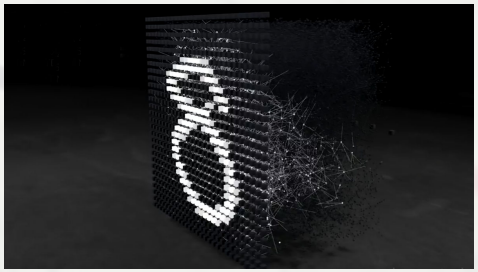
- Serial behavior
- Learn by rules

Von Neumann machines requires either big processors or the idea of parallel processors, while neural networks requires the use of multiple chips customly built for the application.

Why NNs?

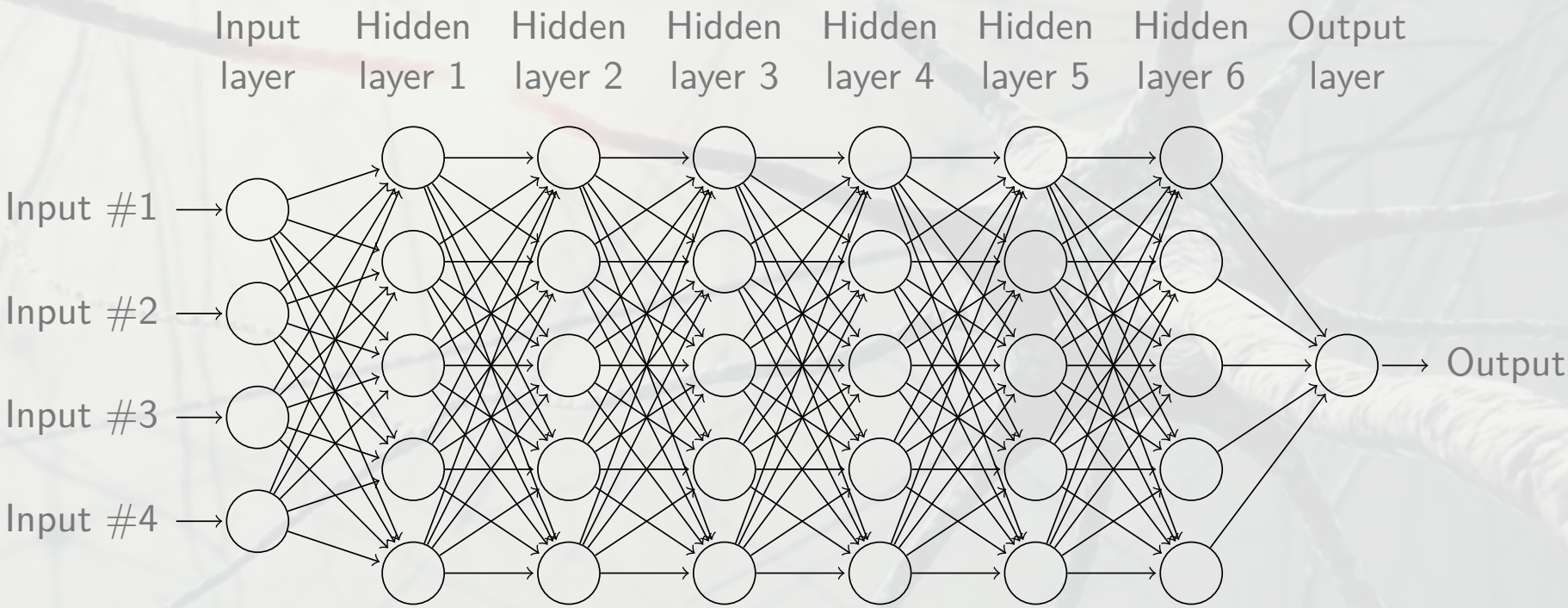
Usually NNs are used to perform human-like tasks in an automatic way (e.g. image recognition, text analysis, speech recognition, ...)

`http://deeplearning.net/datasets/`



IMAGENET

Deep Neural Network Scheme



How does a neuron process the information?

Activation functions

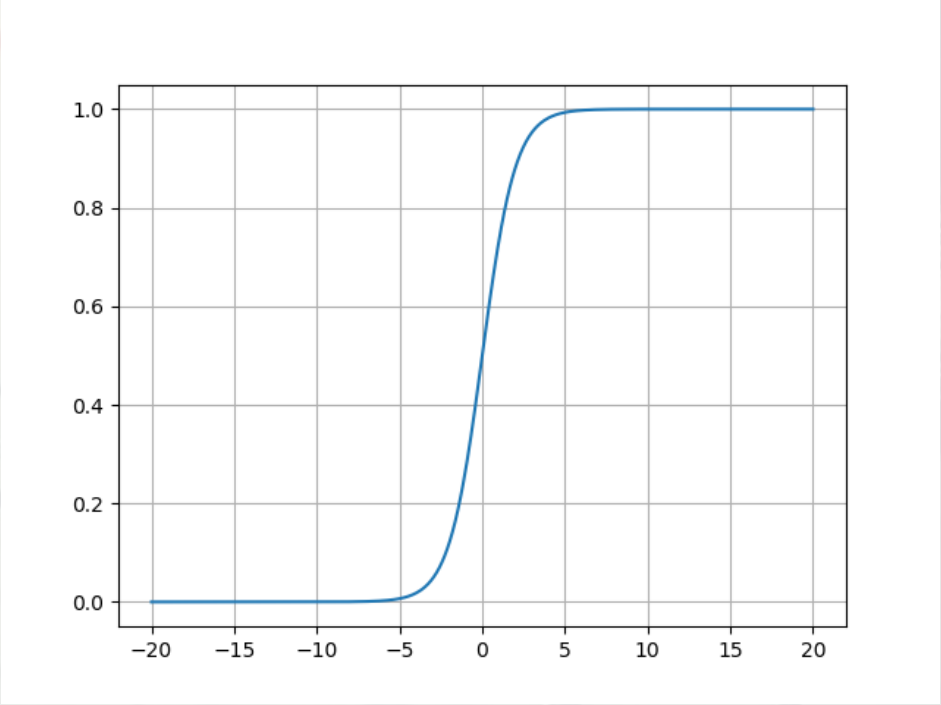
The only process done by the neuron. Receives an input, and applies some function σ and sends the same output a to every neuron in the following layer.

Problems of the Perceptron

By varying the weights and the threshold, we can get different decision-making models.
If it were true that a small change in a weight (or threshold) causes only a small change in output, then we could use this fact to modify the weights and threshold to get our network to behave more in the manner we want.
A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip.

Sigmoid functions

```
def sigmoid(x):
    return 1/(1+exp(-x))
```



Is equivalent for `tanh` and other smooth, monotone and bounded functions.

How a NN learn by example?

Cost functions

Cost functions are the functions used to measure the error that we have in the output layer of our neural network.
All the cost functions follow these two properties:

- The function is non-negative
- The values of the function are close to zero if the output returned by the NN is close to the desired output.

$$MSE = C(w, t) \equiv \frac{1}{2n} \sum_{x} \|y(x) - a\|^2.$$

The feed forward

Feed forward

Is the process that propagates the input across the network to obtain the output.

$$a_j^l = \sigma\Big(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\Big)$$

$$z^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$$

Back-Propagation algorithm

Backprop algorithm

The back-propagation algorithm is the process that is used to repair the weights and biases of the neural network (which are randomly initialized). This process makes the neural-network gain accuracy.

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{1}$$

$$\delta^l = ((\omega^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{3}$$

$$\frac{\partial C}{\partial \omega^l_{jk}} = a^{l-1}_k \delta^l_j \tag{4}$$

Vanishing and exploding problems

Because of this formula:

$$\delta^l = ((\omega^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

If the term $\sigma'(z^l)$ goes to zero, our error in the initial layers goes to zero exponentially even if we have real errors at our initial layers. Also if we are using sigmoids (or similar activation functions), together with our neuron is not identifying the data as relevant due to some random initialized weight, $\sigma'(-3) \approx 0$; thus our input is relevant, and the network has errors, but during the back-propagation the error is vanished. Also if $\sigma'(z^l) > 1$, we have an exploding problem. **These problems make the parameter tuning task very difficult.**

Optimization methods for 3 and 4

Equations 3 and 4 represent the gradient of the cost function respect to b^l_j and w^l_{jk}. So, if we want to minimize the value of C, we can do it using the gradient descent method in the following way:

$$w^l \longrightarrow w^l - \frac{\eta}{m} \sum_{x=1}^{m} \delta^{x,l} (a^{x,l-1})^T$$

$$b^l \longrightarrow b^l - \frac{\eta}{m} \sum_{x=1}^{m} \delta^{x,l}$$

Where η is a hyper-parameter of the model called learning rate, and m is the total number of examples in our training set.

SGD and alternative Op. methods

The Stochastic gradient descent method operates like the regular gradient descent method, but splitting the input examples of the whole training set (here in after called **epoch**) into many **batches**. If each batch has length m, we can keep applying these formulas:

$$w^l \longrightarrow w^l - \frac{\eta}{m} \sum_{x=1}^{m} \delta^{x,l} (a^{x,l-1})^T$$

$$b^l \longrightarrow b^l - \frac{\eta}{m} \sum_{x=1}^{m} \delta^{x,l}$$

The advantage of using this method is the reduced time of its computations. If we are working on a **distributed environment**, each node computes an SGD and then we compute the **average of all nodes** to obtain the result of the gradient descent method and propagate the error backwards with this result.

We can also apply other optimization methods like genetic algorithms to minimize the cost function respect to the parameters of the network.

Why NNs work?

Sigmoidal function

A continuous sigmoidal function $\sigma(t)$ is a continuous function that goes to zero as t goes to minus infinity and goes to one as t goes to infinity.

Universal Approximation Theorem

This theorem says that every $f \in C(I_n)$ can be approximated by $G(x)$ in the form

$$|G(x) - f(x)| < \varepsilon$$

for all $\varepsilon > 0$ and for all $x \in I_n$, where

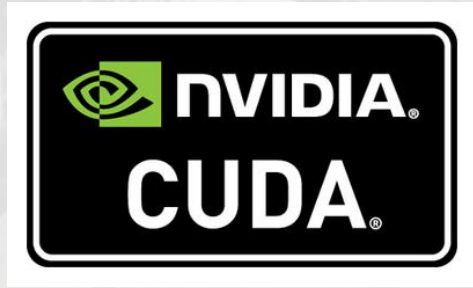
$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

and σ is a continuous sigmoidal function.

http://arxiv.org/abs/1505.03654

Implementation

theano



PyTorch

We are going to use `pytorch` during all the course. Is a python library, only supported on linux systems (has GPU accelerators for NVIDIA cards).

`http://pytorch.org/`

Open exercises

Tricks section

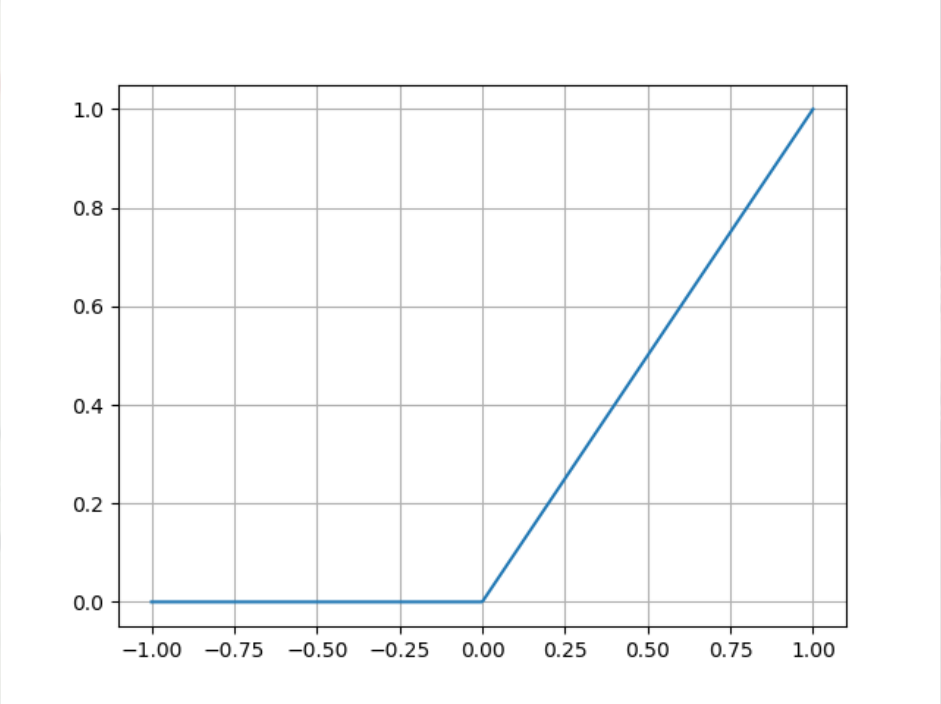
Usually, in real datasets, all neural networks that have state-of-the-art results have very complicated architectures and loads of techniques (tricks) that make them perform substantially better.

List of Tricks

- ReLU functions
- Cross entropy cost function
- Batch normalization and renormalization
- Dropout
- L1 and L2 regularization
- ResNet blocks

ReLU functions

```
def ReLU(x):
    return x * (x > 0)
```



This function avoids the vanishing problem when the neuron is active, and we know that this kind of functions can approximate any function because of the **UAT** for unbounded functions.

Cross entropy cost function

$$C = -\frac{1}{n}\sum_{x}(y ln(a) + (1-y) ln(1-a))$$

This function was explicitly crafted to vanish the derivate term in the error corresponding to the last layer.

Is CE really a cost function?

proof

- First of all, in order to see whether **this function is positive** $C > 0$, we know that all numbers inside the logarithm functions are between 0 and 1, so the logarithm is negative. **As the** y **numbers are classification numbers**, they are also between 0 and 1 so **the entire sum is negative**, then because of the minus sign at the beginning, we can see that $C > 0$.
- In order to see whether this function is also accomplishing **the property that goes to zero when the neural network is classifying correctly,** let us suppose that $y = 0$ and $a \approx 0$ for some input x. This is a case when the neuron is excelling on that input. **We see that the first term in the expression for the cost vanishes, since** $y = 0$**, while the second term is just** $-ln(1 - a) \approx 0$**.** A similar analysis holds when $y = 1$ and $a \approx 1$. So, the contribution to the cost will be low provided the actual output is close to the desired output.

Batch normalization

We define **internal covariate shift** as the change in the distribution of network activations due to the change in network parameters during training. Having:

$$\mu_\beta = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_\beta^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\beta)^2$$

1. Compute μ_β
2. Compute σ_β^2
3. Normalize the data as:

$$\hat{x}_i = \frac{x_i - \mu_\beta}{\sqrt{\sigma_\beta^2}}$$

4. The final output to be processed by the activation function is $y_i = \gamma\hat{x}_i + \alpha$

Dropout

Imagine that we randomly **drop out** half of the hidden neurons on each layer of our neural network and we train the other "half of the network" alone. After doing this, we do the opposite movement, that means to turn on the dropped out neurons and drop out the previously trained ones. After that, we train again our "half neural network". Consequently we turn on all the hidden neurons and divide all the weights and all the biases by two.

Example

Imagine a number k, larger than two, for example $k = 10$. Heuristically, when we dropout different sets of neurons, it is rather like we are training different neural networks (in this case 10 neural networks). The different networks will overfit in different ways and, hopefully, the net effect of the dropout will be to reduce overfitting.
For example if 8 out of 10 networks assert that the classified number is a 3, it is reasonable to think that this would be the correct solution.

L1 and L2 regularization

Being C_0 the cross-entropy cost function, we define:

L2 regularization

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2$$

L1 regularization

$$C = C_0 + \frac{\lambda}{2n} \sum_w |w|$$

These two methods share the following property:
The network prefers learning on small weights. Large weights are only learned when they truly improve the efficiency of the network.

L1 and L2 differences

- Imagine that $w = K >> 1$, then L1 contributes to the cost function much less than L2.
- Imagine now that $w \approx 0$ then L1 contributes much more to the cost function than L2.

Deep Learning for Image processing

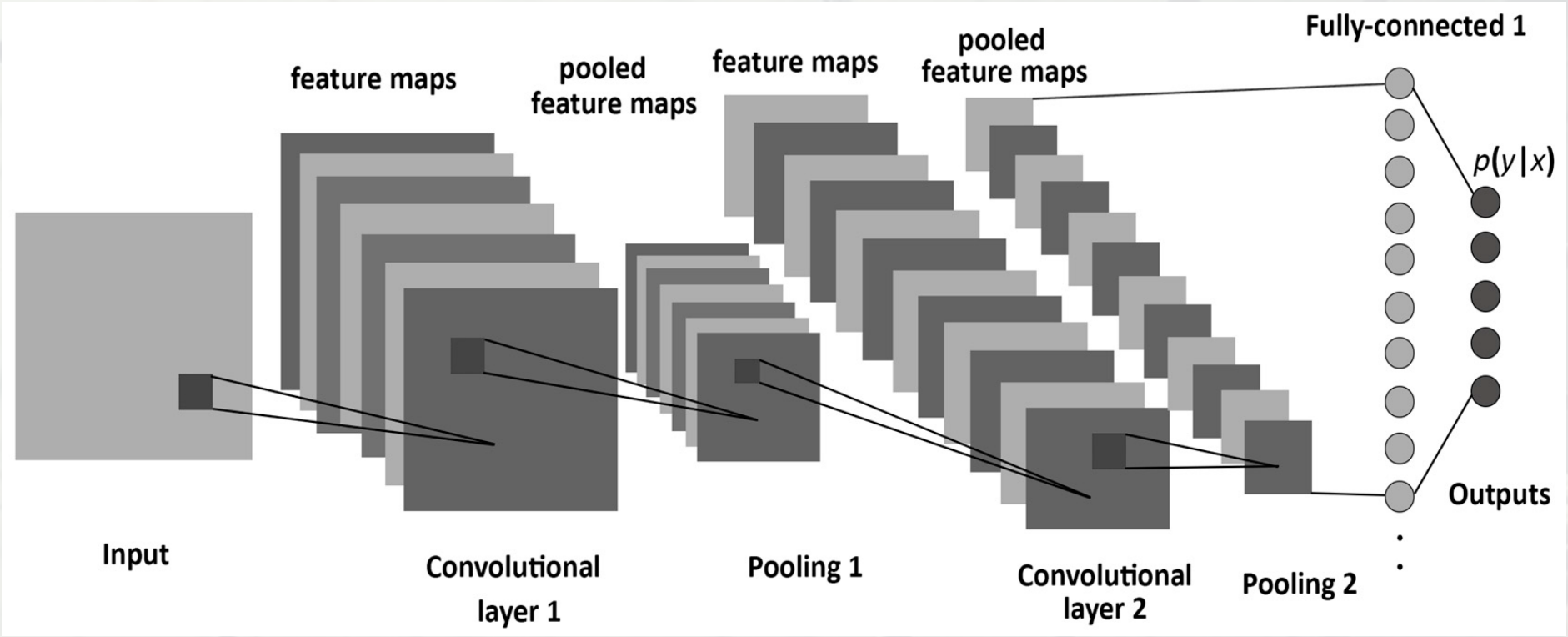
Convolutional neural networks (**CNNs**) have been applied to visual tasks **since the late 1980s**. However, despite a few scattered applications, they were dormant until the mid-2000s when **developments in computing** power (more concretely in GPU's computing power) and the advent of large amounts of labeled data, supplemented by improved algorithms, contributed to their advancement and **brought them to the forefront of a neural network renaissance that has seen rapid progression since 2012.**

During this part of the course, **we will focus on the application of CNNs to image classification** tasks, we cover their parts and development. **Finally we will study a list of tricks** that will power our models to achieve good accuracy levels.

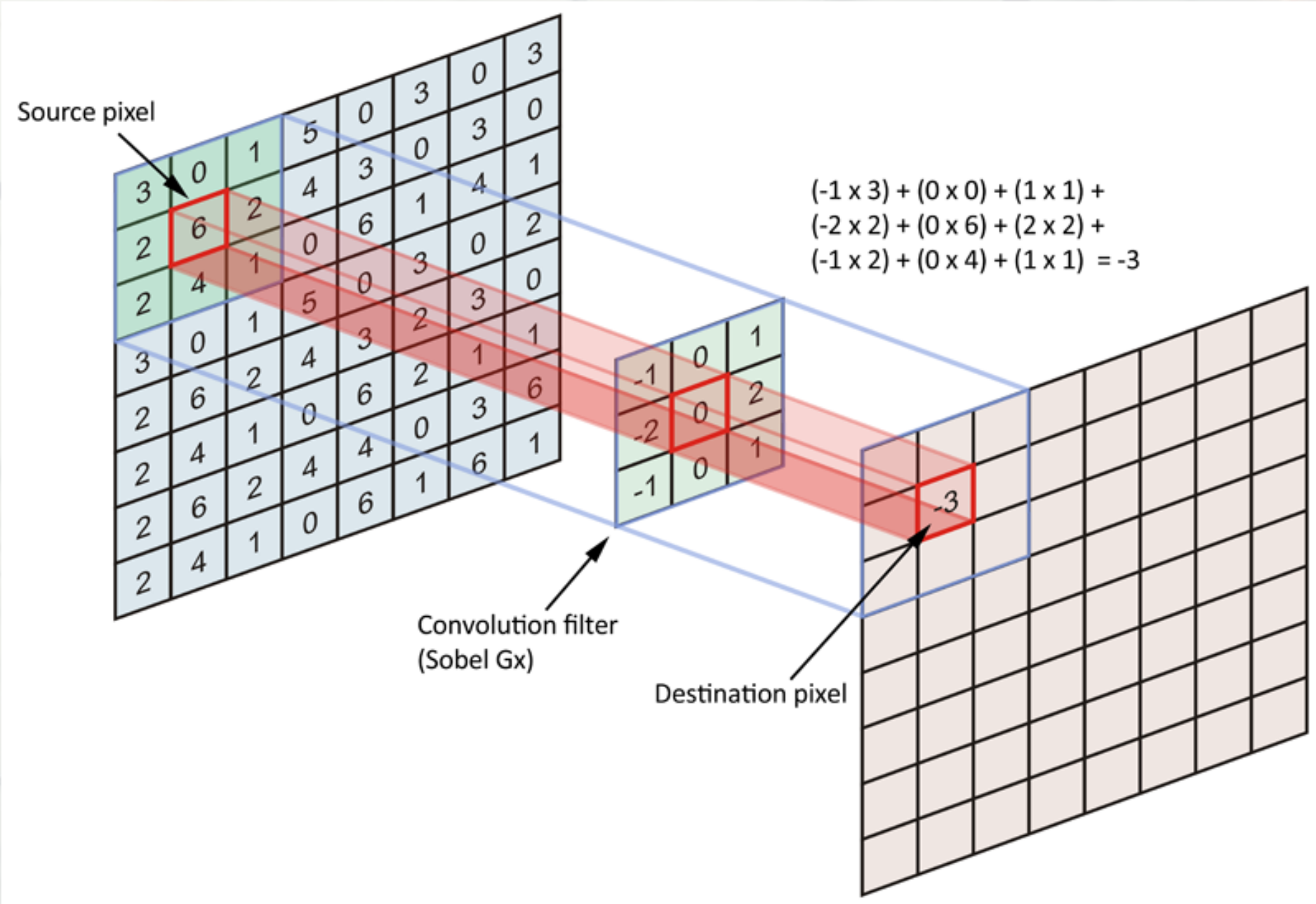
CNN history

- Traditional approach for classifying images:
  Handcrafted features were first extracted from images using feature descriptors, and these served as an input to a trainable classifier.
- In 1980 Kunihiko Fukushima established the fundamentals of CNNs.
- In 1998 Yann LeCun et al. improved the model **by adding the back-propagation algorithm.** (not trivial)
- In 2012 Dan Ciresan et al. **implemented the algorithm in GPU.**
- In 2012 the ImageNet Large Scale Visual Recognition Challenge (ILSVRC 2012) Krizhevsky et al. used a **CNN** to classify approximately 1.2 million images into 1000 classes, **with record-breaking results.**

Convolutional Neural Networks



What is a convolutional filter?



Why these filters?

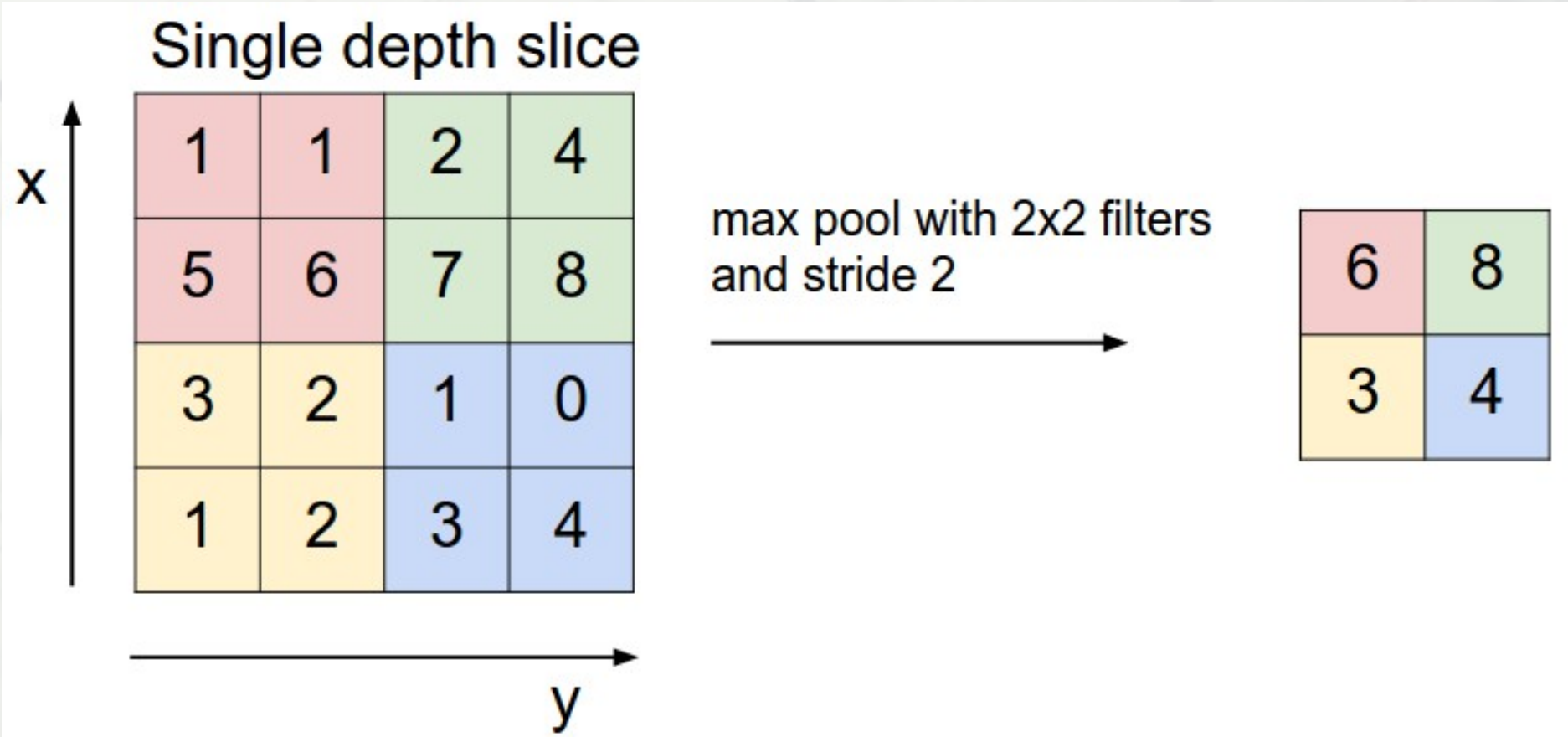
2D property

These filters encode 2D information. Using regular neurons of a DNN, we can lay the pixels of an image in any order, but when we use a filter, we create the new features, taking care of their position in a 2D plain. Therefore as images are 2D, this property helps.

Memory efficiency

Fully connected layers, have $n \times m$ coefficients stored in memory, if our layer goes from n neurons to m. When using filters, if the size of our filter is 3×3 we only need to store 9 coefficients.

What is a pooling?



Kinds of poolings

- **Max pooling** makes the most active neuron of the previous layer, stay in the following layer, as a new feature.
- **Average pooling** prevents the network from learning the image structures, such as edges and textures. It measures the mean value of existence of a pattern in a given region.
- **Sum pooling** the same as average pooling.

CNNs explained 1

Architecture

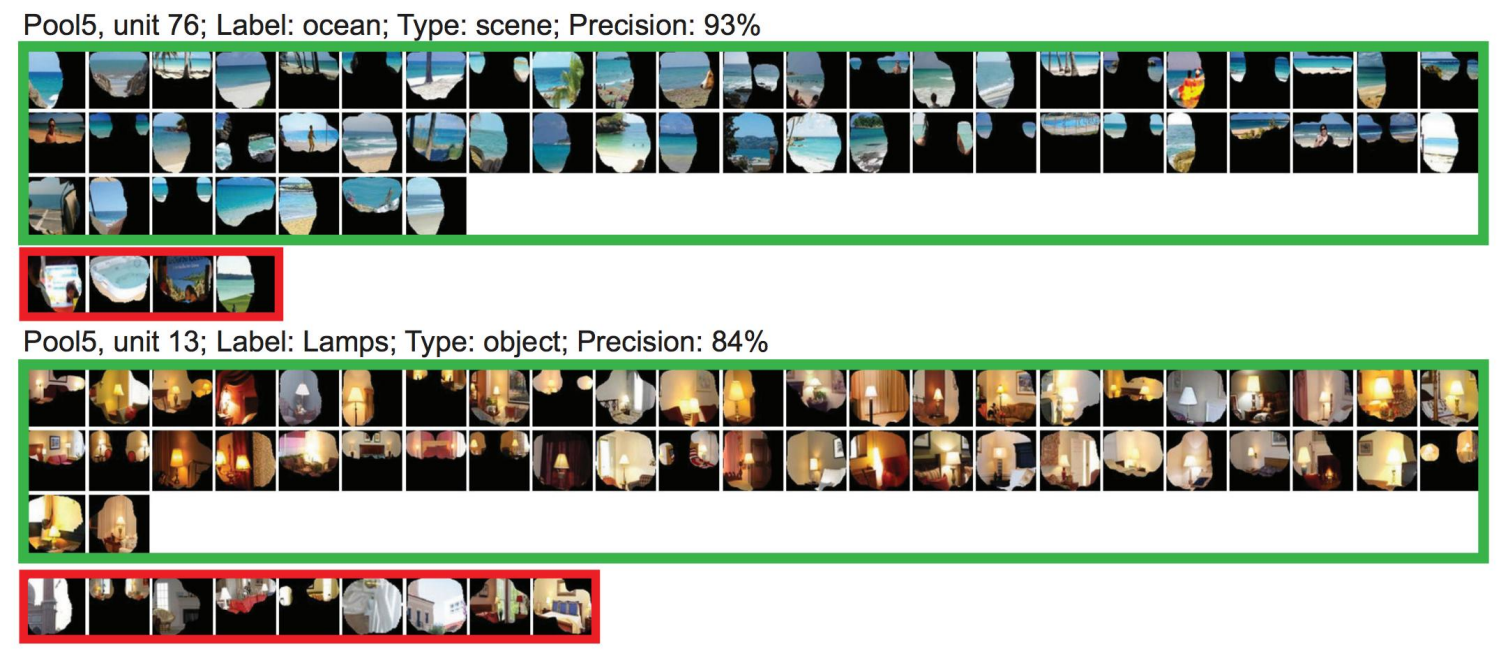
- Pre-process the images
- Build the network
  - Convolutional layers
  - Pooling layers

  till some depth
- Stablish the features of the FC layer
- Fully Connect the layers
- Build the DNN after the FC layer
- Train the network with data examples
- Change hyper-parameters of the architecture (i.e. how many kernels, size of these kernels, etc) till some accuracy on validation set

CNNs explained 2

Local receptive field

For a neuron n in some layer of the CNN, we call the receptive field of this neuron, all pixels of the original image that contribute to n neuron.



This receptive field changes with every example, therefore looking to the receptive field of a neuron in many examples we can deduce properties of the architecture (i.e. if we need more layers, if the neuron has enough receptive field to contain the important characteristics of the objects)

CNNs explained 3

Back-prop across new operations

Convolutional filters

Show the `.gif` that computes the convolution, and deduce back-prop of the error.

Max poolings

There is no gradient with respect to non maximum values, since changing them slightly does not affect the output. Furthermore the max is locally linear with slope 1, with respect to the input that actually achieves the max. Thus, the gradient from the next layer is passed back to only that neuron which achieved the max. All other neurons get zero gradient.

$$\delta_i^l = 0 \ \ \forall i \in l \ \ \textit{if} \ \ i \neq \textit{argmax}(z_i^l) \ \ \textit{else} \ \ \delta_i^l = \delta_j^{l+1}$$

`http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/`

When to use a CNN in terms of data

Which kinds of data can we process with CNNs with state-of-the-art performance?

- **2D and 3D images**
- Point-clouds on many dimensions (with spatial meaning)
- Text (NLP-word embeddings)
- **Audio (Speech)**

We will focus on image processing with CNNs till the end of the course.

Real image data

When we work with real images (not toy examples nor challenges), usually images are very different in terms of color, size, resolution, etc.
How can CNN handle all these differences?
We need pre-processing?

Example

Usually with color images, we split our data in at least 3 channels, and we can apply the same operations in a CNN, but in a vectorial way to compute these 3 simpler CNNs (Usually all deep learning frameworks work with the [N,C,H,W,D] format).

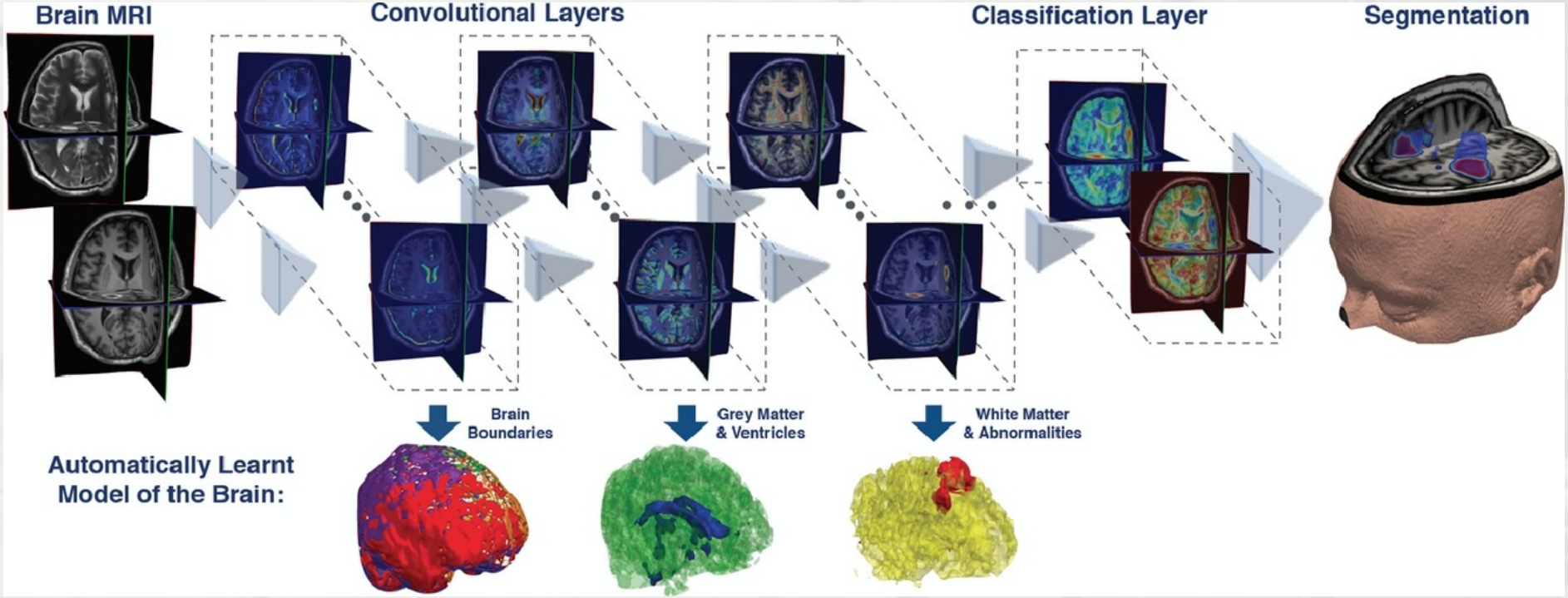
Padding and strides

Show the `.gif` files to see how padding and strides work.
The arithmetic of the layers of the CNN using padding and strides is the following:

$$o = floor\left(\frac{i + 2p - k}{s}\right) + 1$$

where o is the output size of the image, i is the input size, p is the zero padding, k is the kernel size and s is the stride size (all sizes are the same width and height). **With a stride of 1 we move our kernel without holes trough the image.** You can find more information about this on: `https://arxiv.org/pdf/1603.07285.pdf`

3D and ND Conv filters



Data augmentation techniques

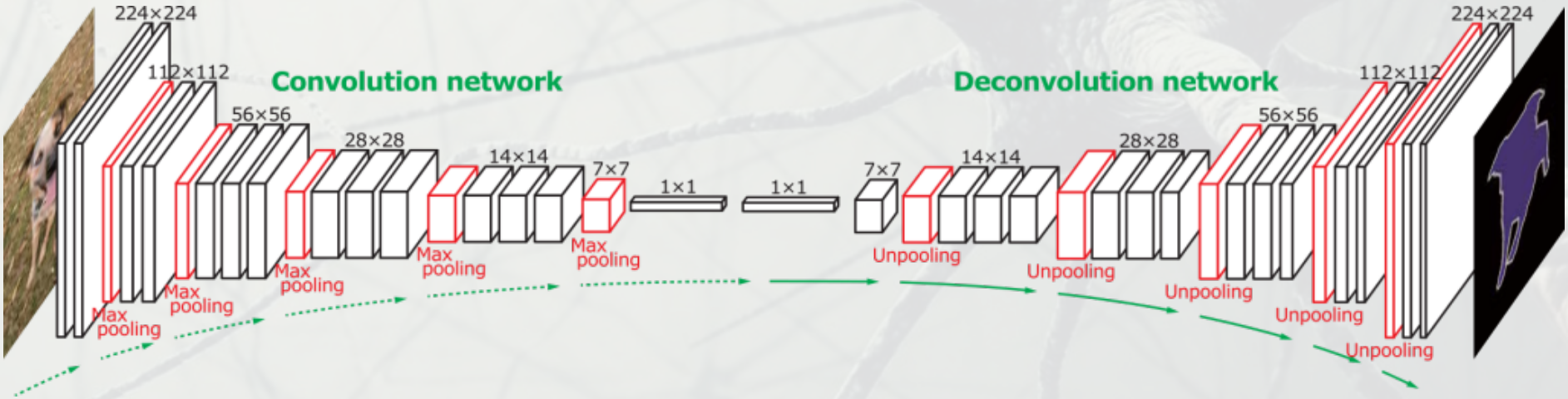
Usually when we are working with images, and we want to improve the accuracy of our network, we apply data augmentation techniques to augment our dataset, and have more training examples. These are some methods:

- Use rotations with little angles.
- Flipping the images.
- Scaling.
- Adding little white noise.
- Using NNs (autoencoders).

Can we achieve good results starting with very little amount of data?
`http://cognitivemedium.com/rmnist`

FB example

- Open your browser (Chrome better if possible)
- Open facebook
- Open an image and right click it
- Click inspect element and read the description



Face recognition challenge

the Messi Neymar Iniesta challenge

Hello! this is a small exercise that will help you to see the power of deep learning for image recognition.

The dataset consists of 1.588 images of either Lionel Messi (534), Neymar (444) or Iniesta (610). The images have been zoomed into the face and cropped to 60x60 pixels. They are all jpg format but some may not actually be Messi or Neymar or even a face as they have been downloaded from the internet with a Google search, most of them however, should be right.

The challenge consists of creating a deep neural network classifier that will discriminate between them. You should get at least 85% accuracy, note that a random classifier would already have a 33% accuracy. The neural network **MUST BE IMPLEMENTED WITH TENSORFLOW**, please do not use other frameworks.

The submission package include the following:

- The **python code** used for the network, please make it short and easy to review
- The **saved model (.cpkt)** and if needed, instructions on how to run it

Please also include in your email the following information:

- A brief description of the network and the work you have done
- The amount of working time you have spent in this challenge
- The amount of training time your model takes and the hardware used
- The accuracy of your model describing how you calculate it