Expressiveness & Training

Next argument

Expressiveness

Expressiveness

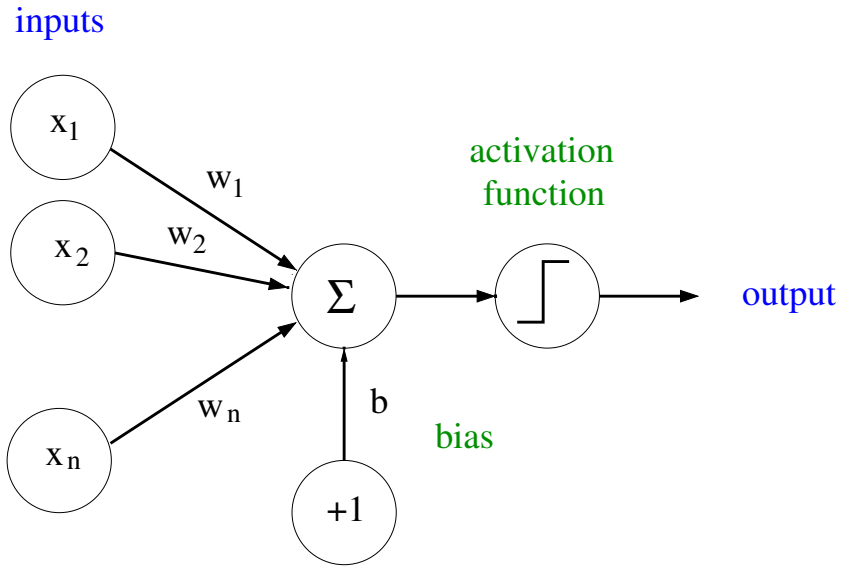
Can we compute any function by means of a Neural Network?

Do we really need **deep** networks?

Can we compute any function with a single neuron?

Single layer case: the perceptron

Binary threshold:



$$output = \begin{cases} 1 & if \sum_i w_i x_i + b \geq 0 \\ 0 & otherwise \end{cases} \qquad output = \begin{cases} 1 & if \sum_i w_i x_i \geq -b \\ 0 & otherwise \end{cases}$$

Remark: the bias set the position of threshold.

Hyperplanes

The set of points

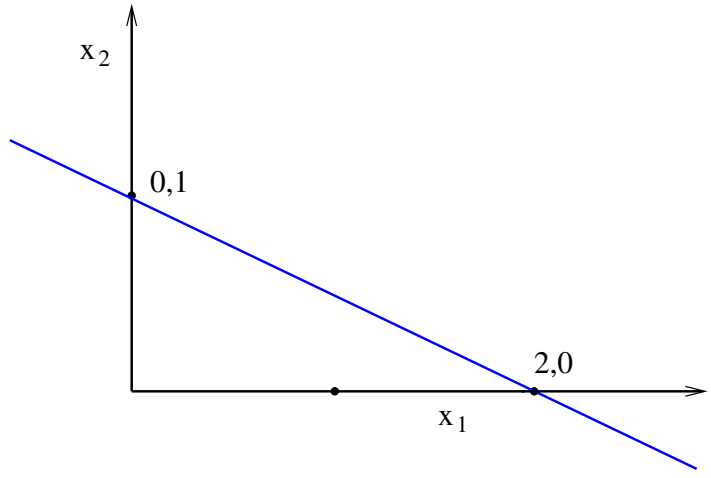
$$\sum_i w_i x_i + b = 0$$

defines a hyperplane in the space of the variables x_i

Example:

$$-\frac{1}{2}x_1 + x_2 + 1 = 0$$

is a line in the bidimensional space



Hyperplanes

The hyperplane

$$\sum_i w_i x_i + b = 0$$

divides the space in two parts: to one of them (above the line) the perceptron gives value 1, to the other (below the line) value 0.

"above" and "below" can be inverted by just inverting parameters:

$$\sum_i w_i x_i + b \leq 0 \iff \sum_i -w_i x_i - b \geq 0$$

Computing logical connectives: NAND

Can we implement this function (NAND) with a perceptron?

x_1	x_2	*output*
0	0	1
0	1	1
1	0	1
1	1	0

Can we find two weights w_1 and w_2 and a bias b such that

$$nand(x_1, x_2) = \begin{cases} 1 & if \sum_i w_i x_i + b' \geq 0 \\ 0 & otherwise \end{cases}$$

Computing logical connectives: NAND

Can we implement this function (NAND) with a perceptron?

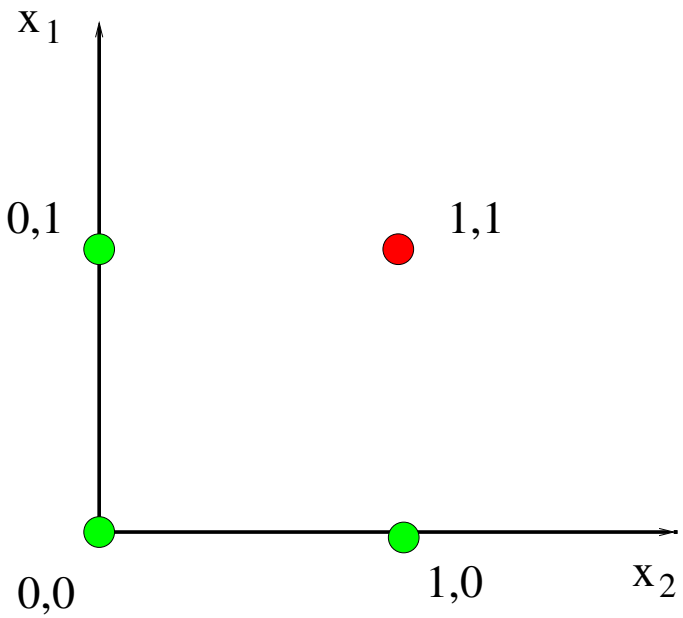
x_1	x_2	*output*
0	0	1
0	1	1
1	0	1
1	1	0

Can we find two weights w_1 and w_2 and a bias b such that

$$nand(x_1, x_2) = \begin{cases} 1 & if \sum_i w_i x_i + b' \geq 0 \\ 0 & otherwise \end{cases}$$

Graphical representation

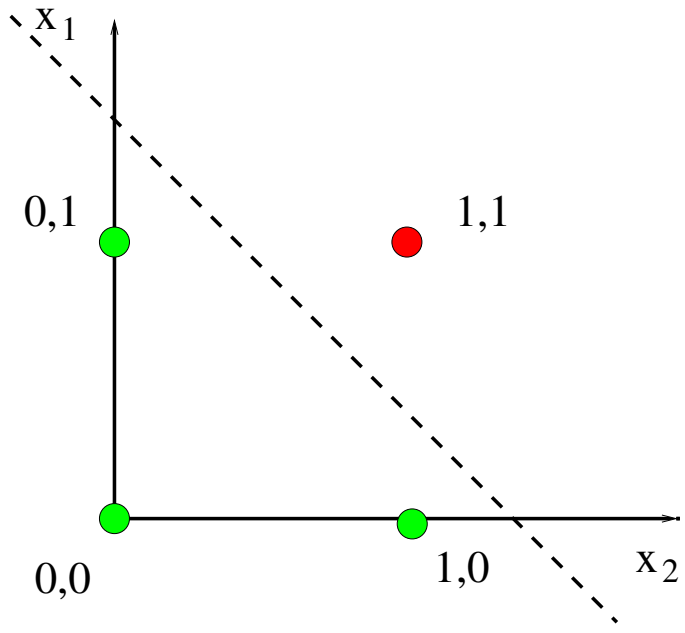
Same as asking:
can we draw a straight line to separate green and red points?



NAND

Lines, planes, hyperplanes

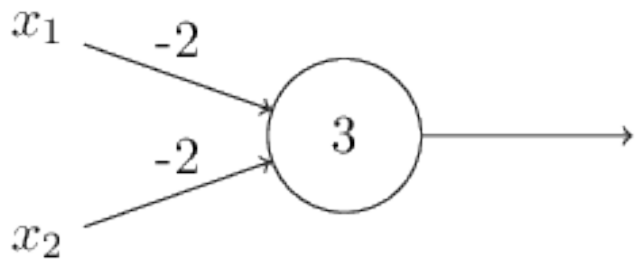
Yes!



NAND

line equation: $1.5 - x_1 - x_2 = 0$ or $3 - 2x_1 - 2x_2 = 0$

The NAND-perpceptron

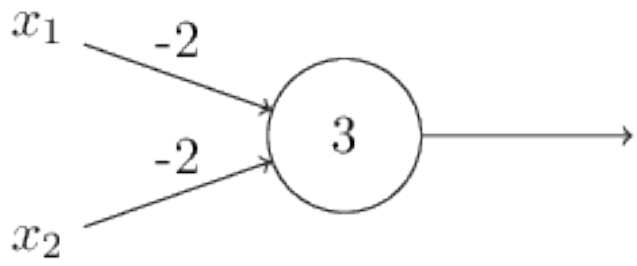


$$output = \begin{cases} 1 & if\ -2x_1 - 2x_2 + 3 \geq 0 \\ 0 & otherwise \end{cases}$$

x_1	x_2	*output*
0	0	1
0	1	1
1	0	1
1	1	0

Can we compute any logical circuit with a perceptron?

The NAND-perpceptron



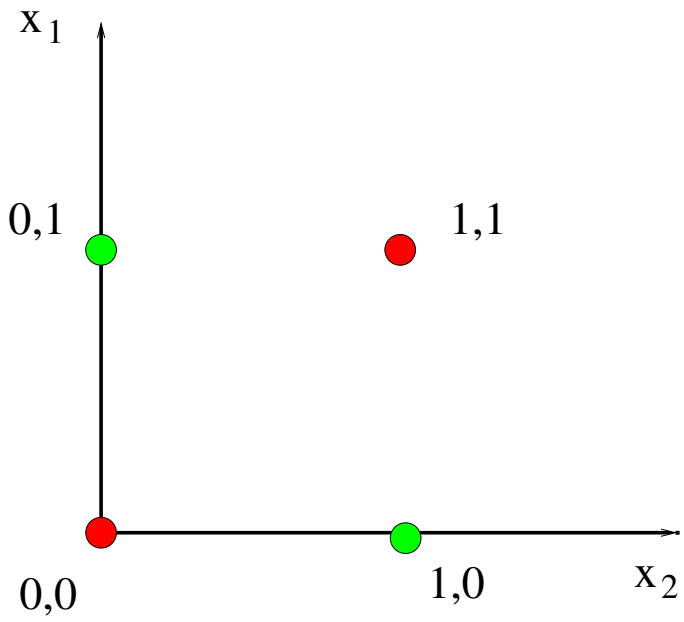
$$output = \begin{cases} 1 & if\ -2x_1 - 2x_2 + 3 \geq 0 \\ 0 & otherwise \end{cases}$$

x_1	x_2	*output*
0	0	1
0	1	1
1	0	1
1	1	0

Can we compute any logical circuit with a perceptron?

The XOR case

Can we draw a straight line separating red and green points?

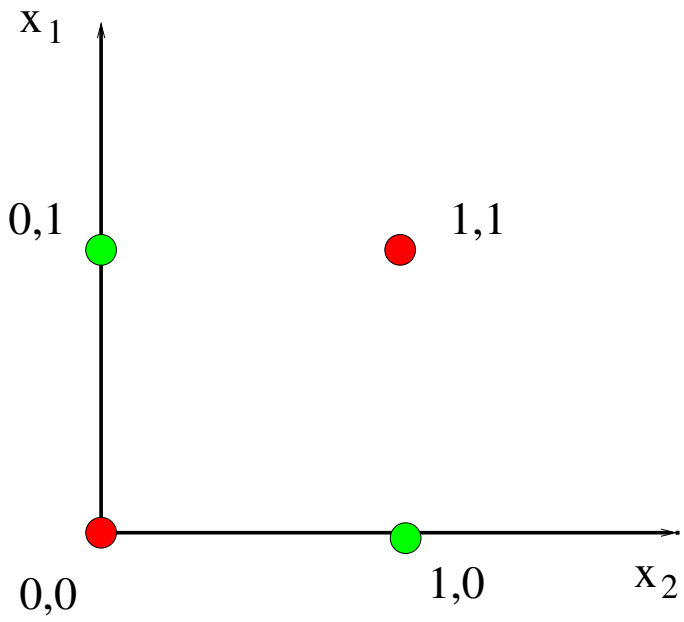


No way!

Single layer perceptrons are not complete!

The XOR case

Can we draw a straight line separating red and green points?



No way!

Single layer perceptrons are not complete!

Multi-layer perceptrons

Question:

- we know we can compute nand with a perceptron
- we know that nand is logically complete
  (i.e. we can compute any connective with nands)

so:

why perceptrons are not complete?

answer:

because we need to compose them and consider
Multi-layer perceptrons

Multi-layer perceptrons

Question:

- we know we can compute nand with a perceptron
- we know that nand is logically complete
  (i.e. we can compute any connective with nands)

so:

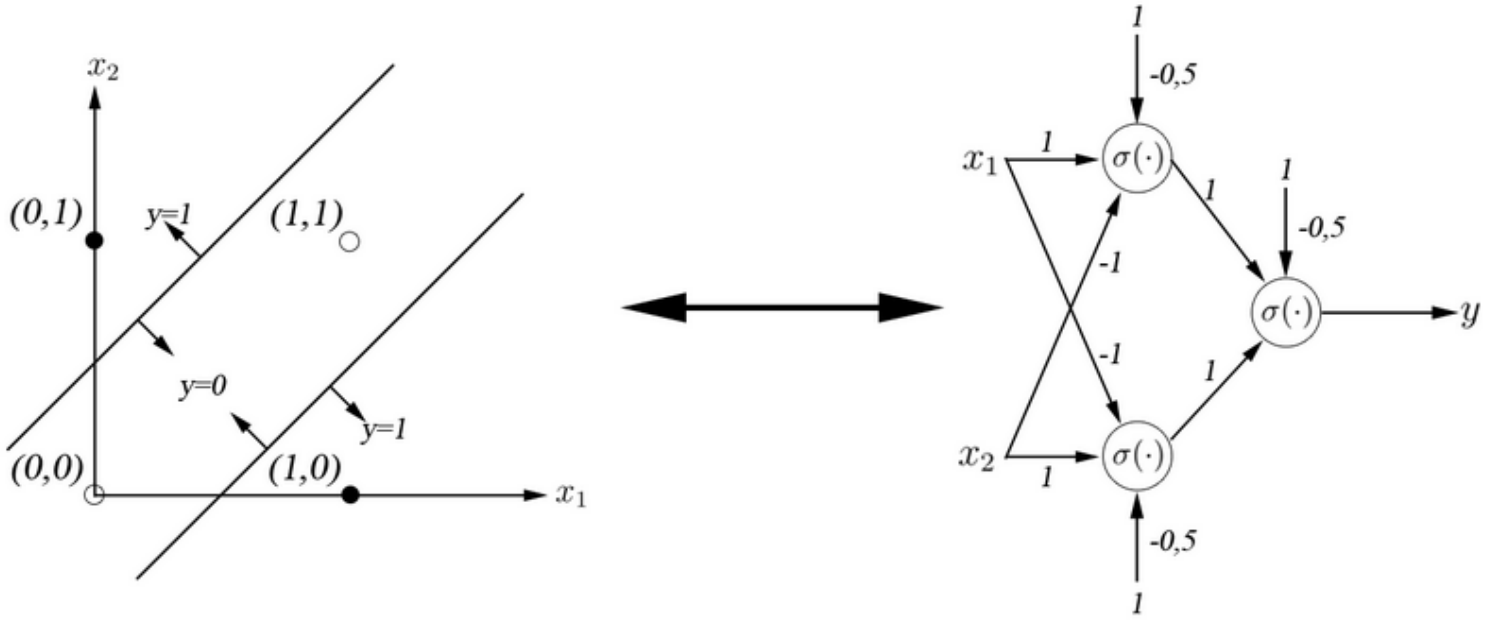
why perceptrons are not complete?

answer:

because we need to compose them and consider
Multi-layer perceptrons

Example: Multi-layer perceptron for XOR

Can we compute XOR by **stacking** perceptrons?



Multilayer perceptrons are logically complete!

Important Points

- shallow nets are already **complete**

  Why going for deep networks?

  With deep nets, the same function may be computed with **less neural units** (Cohen, et al.)

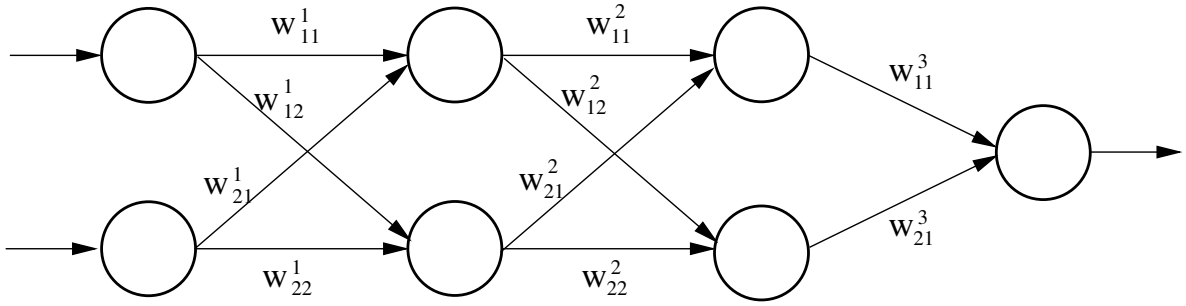
- Activation functions play an **essential role**, since they are the only source of nonlinearity, and hence of the expressiveness of NNs.

  Composing linear layers not separated by nonlinear activations **makes no sense**!

Training

Current loss

Suppose to have a neural network with some configurations of the parameters θ.

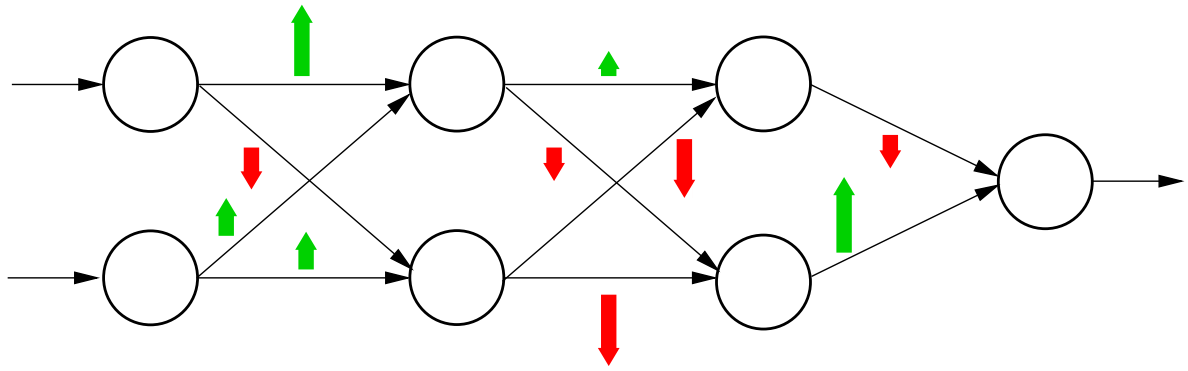


We can take a **batch** of data, pass them (in parallel) through the network, compute the output, and evaluate the current loss relative to θ.

This is a **forward pass** through the network.

Parameter updating

Next, we would like to adjust the parameters in such a way to decrease the current loss.

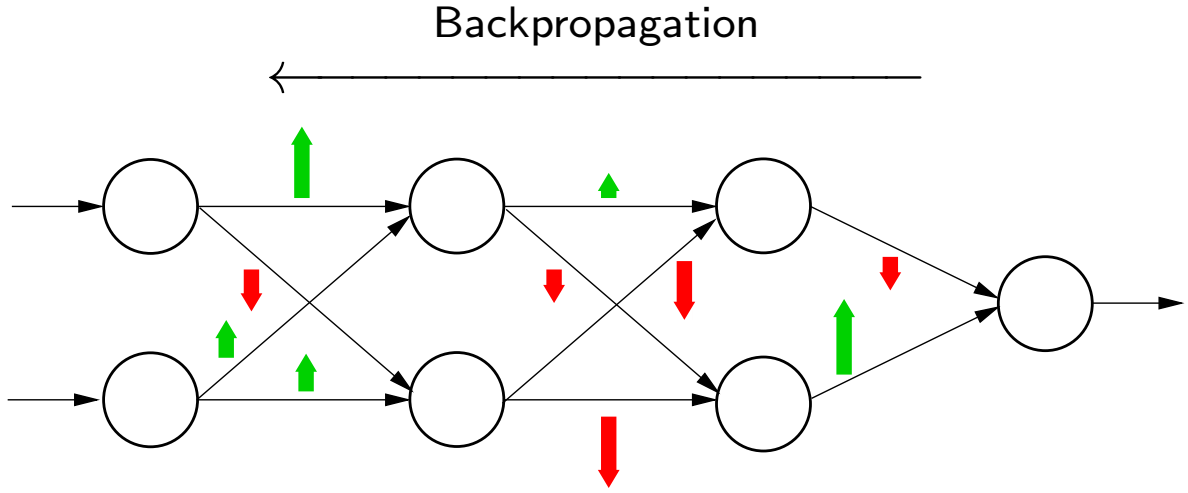


Each parameter should deserve a different adjustment, some of them positive, other negative.

The mathematical tool that allows us to establish in which way parameters should be updated is the **gradient**: a vector of **partial derivatives**.

Backpropagation

The gradient is computed **backward**, **backpropagating** the loss to all neurons inside a networks, and their connections.



This is a **backward pass** through the network.

The algorithm for computing parameters updates is known as **backpropagation algorithm**.

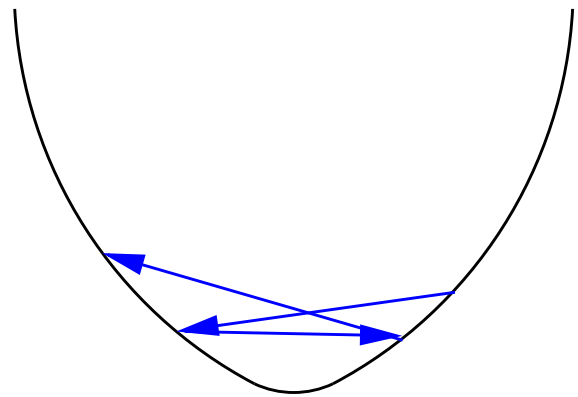
Learning rate

The backpropagation algorithms only gives a **direction** in which gradients should be updated.

The actual amount of the update is obtained by multiplication with a scalar hyperparameter (fixed externally, not learned) called **learning rate**.

Increasing the learning rate can make training faster, but it reduces the accuracy of the result.

If we make large steps nearby the optimum, we can miss it.



Optimizer

Many techniques can be used to **tune** the learning rate during training.

The tool in charge of governing the gradient descent technique - possibly dynamically adapting the learning rate - is the so called **optimizer** (e.g. Adam, in our example).

Many possibilites:
- Use a fixed learning rate
- adapt the global learning rate during time
- adapt the learning rate on each connection separately
- use the so called *momentum*
- ...

suggested lecture: Geoffrey Hinton's lecture

Varying the batchsize

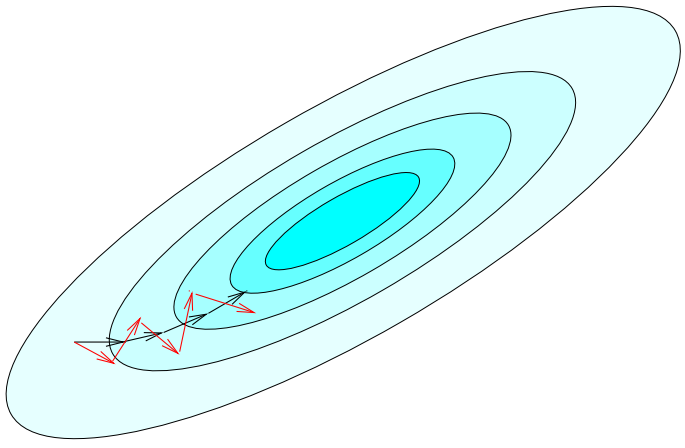
Parameters are updated **every time** a batch is processed.

During an epoch, we make a total number of updates equal to the size of the training set divided by the batchsize.

If we decrease the batchsize we update more frequently (that is good), but updates are less accurate, since we are backpropagating form a loss relative to very specific data.

Conversely, if we increase the batchsize, updates grow in accuracy (the ideal would be to compute them on the whole training set - **fullbatch**) but training can be slow, since parameters are too rarely updated.

Fullbatch, Online and MiniBatch



Fullbatch (all training samples): the gradient points to the direction of steepest descent on the error surface (perpendicular to contour lines of the error surface)

Online (one sample at a time) gradient zig-zags around the direction of the steepest descent.

Minibatch (random subset of training samples): a good compromise.

The Backpropagation algorithm (and its problems)

Computing the gradient

A neural network computes a complex function resulting from the composition of many neural layers. How can we compute the gradient w.r.t. a specific parameter (weight) of the net?

We need a mathematical rule know as the chain rule (for derivatives).

The chain rule

Given two derivable functions f, g with derivatives f' and g', the derivative of the composite function $h(x) = f(g(x))$ is

$$h'(x) = f'(g(x)) * g'(x))$$

Equivalently, letting $y = g(x)$,

$$h'(x) = f'(g(x)) * g'(x)) = f'(y) * g'(x)$$

The derivative of a **composition** of a sequence of functions is the **product** of the derivatives of the individual functions.

QUESTION: why binary thresholding is not a good activation function for backpropagation?

Backpropagation rules in vectorial notation

Given some error function E (e.g. euclidean distance) let us define the error derivative at l as the following vector of partial derivatives:

$$\delta^l = \frac{\partial E}{\partial z^l}$$

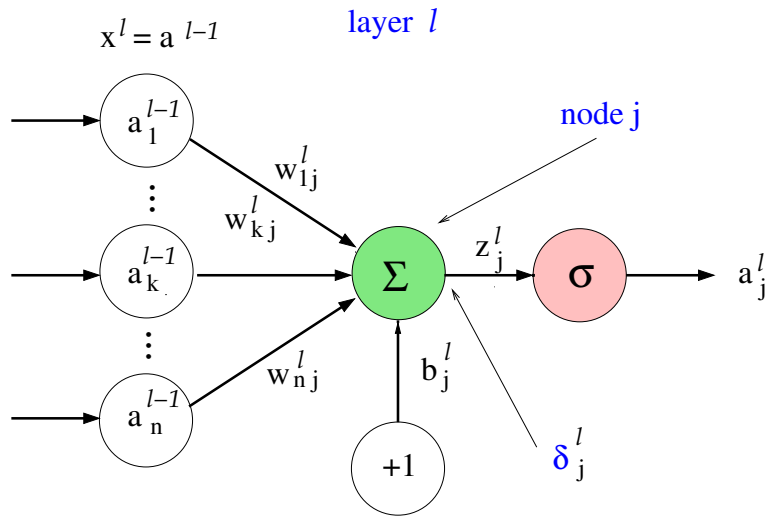
We have the following equations

$$(BP1) \quad \delta^L = \nabla_{a^L} E \odot \sigma'(z^L)$$

$$(BP2) \quad \delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

$$(BP3) \quad \frac{\partial E}{\partial b^l_j} = \delta^l_j$$

$$(BP4) \quad \frac{\partial E}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$



where $\odot$ is the Hadamard product (component-wise)

The vanishing gradient problem

$$(BP2) \quad \delta^l = (w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

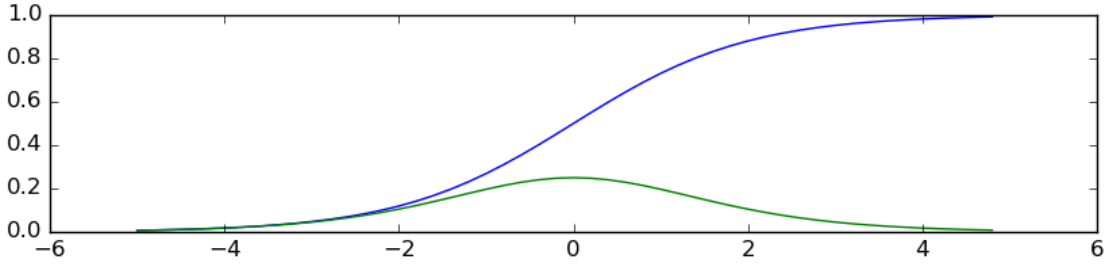
By the chain rule, the derivative is a long sequence of factors, where these factors are, alternately

- derivatives of activation functions
- derivative of linear functions, that are constants (in fact, the transposed matrix of the linear coefficients)

Let's have a look at the derivatives of a couple of activation functions.

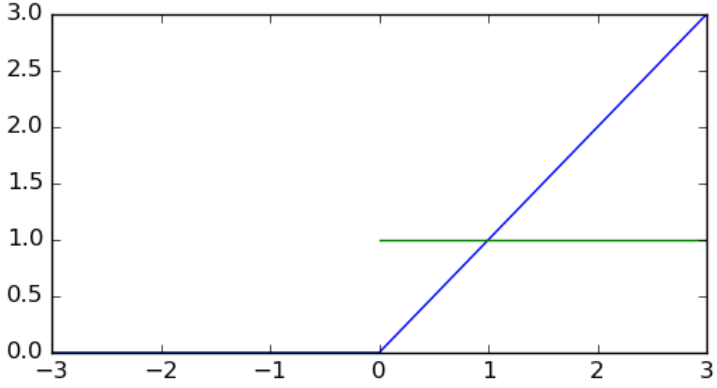
Derivatives of common activation functions

Sigmoid



Observe the flat shape of $\sigma'(x)$, always below 0.25

Relu



The vanishing gradient problem

If you systematically use the sigmoid as activation function in all layers of a deep network, the gradient will contain a lot of factors below 0.25, resulting in a very small value.

If the gradient is close to zero, learning is impossible.

This is known as the **vanishing gradient problem**.

A bit of history

The vanishing gradient problem blocked the progress on neural netwoks for almost 15 years (1990-2005).

It was first bypassed by network pre-training (e.g. with Boltzmann Machines), and later by the introduction on new activation functions, such as Rectified Linear Units (RELU), making pre-training obsolete.

Still, fine-tuning starting from good network weights (e.g. VGG) is a viable approach for many problems (transfer learning).