Deep Networks

From MLC to NN to Deep-NN

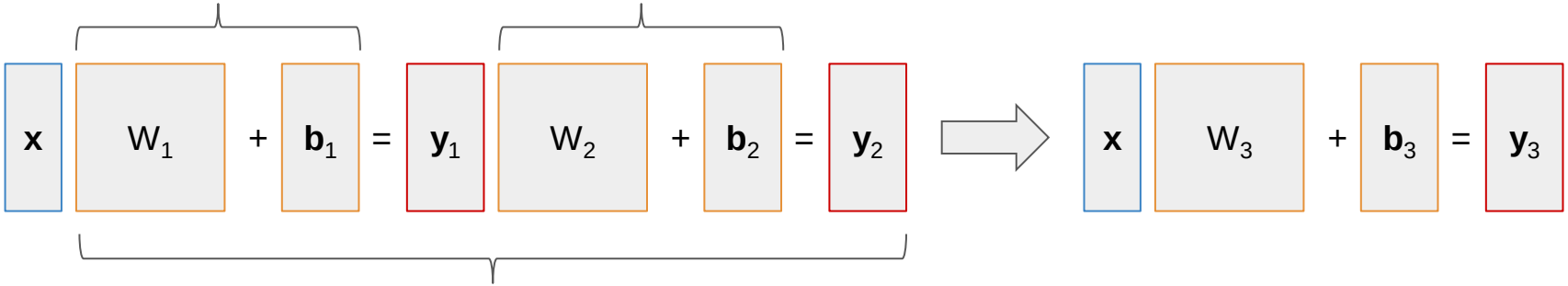
- **Multinomial Logistic Classification** (MLC)
  - *fast*: efficient computation due to the linear model
  - *stable*: small input variations generates small output variations and the derivative of the model is constant

  but...
  - trains only a “small” parameter set
  - can’t represent non linear relations among the inputs
- What if I concatenate multiple MLCs?

$$D(s(\mathbf{x}W+\mathbf{b}), \mathbf{L})$$

$x_1 + x_2$

$x_1 * x_2$



still a linear model…!

From MLC to NN to Deep-NN

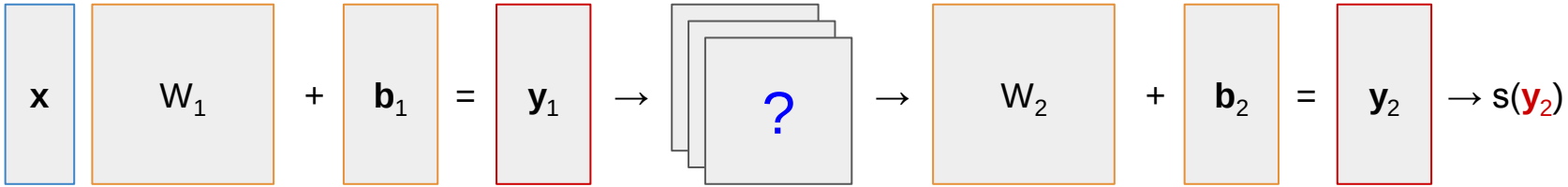
- <u>GOAL</u>: build a bigger and non linear model
- <u>IDEA</u>: concatenate many linear systems (MLC) and insert a ***non linear function*** between two consecutive MLCs, that is, the activation function

Which function?

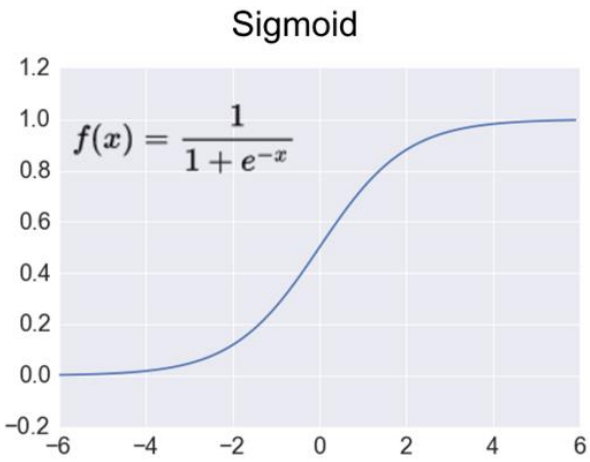
ReLU

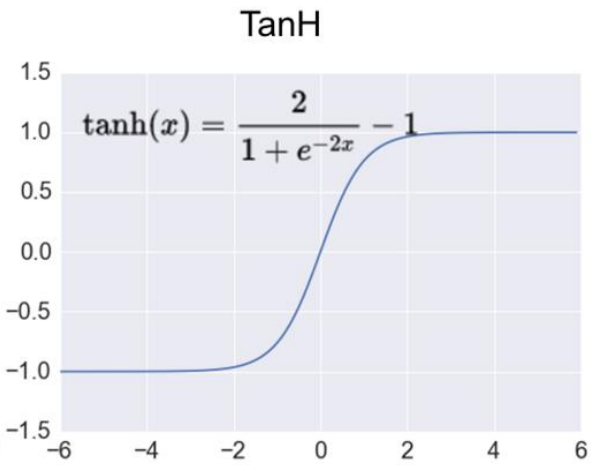
sigmoid

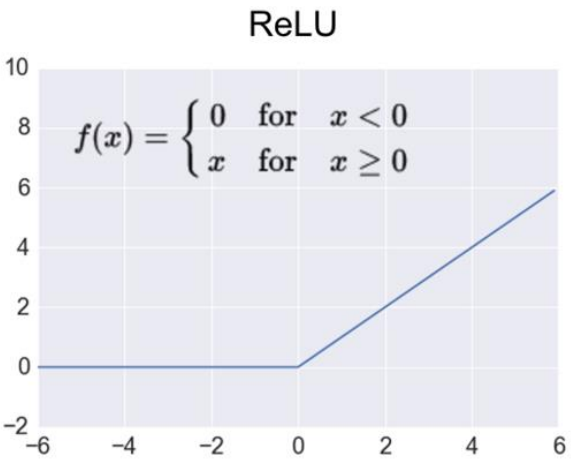
tanh



Choose the (non linear) activation function







- A perceptron classic

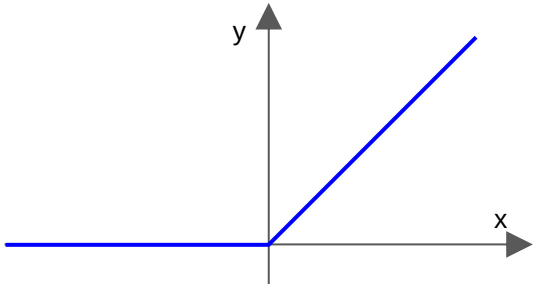
- similar to sigmoid

- simplest function
- constant derivative

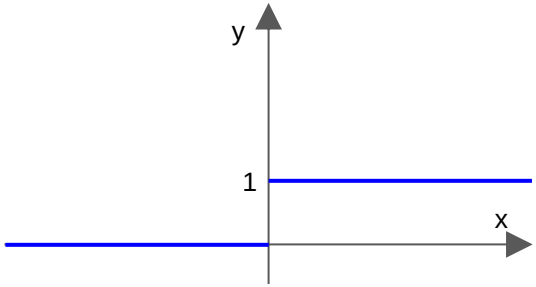
http://adilmoujahid.com/posts/2016/06/introduction-deep-learning-python-caffe/

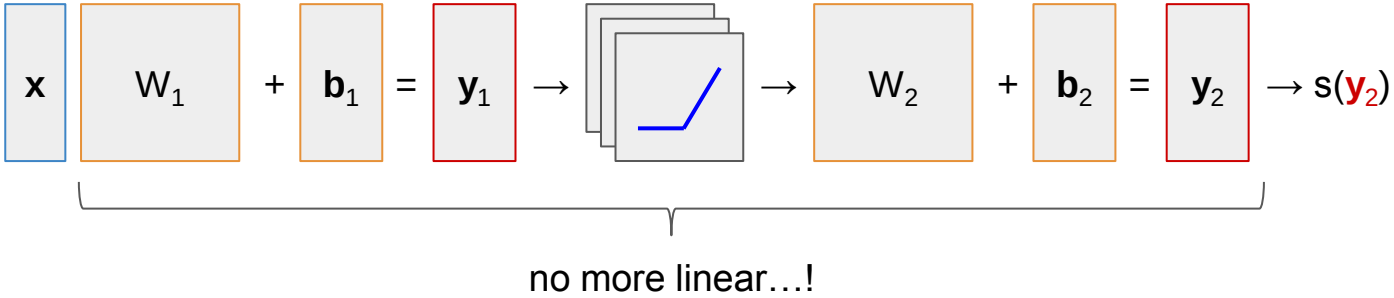
ReLU: Rectified Linear Unit

$$ReLU(x) = max(0, x)$$



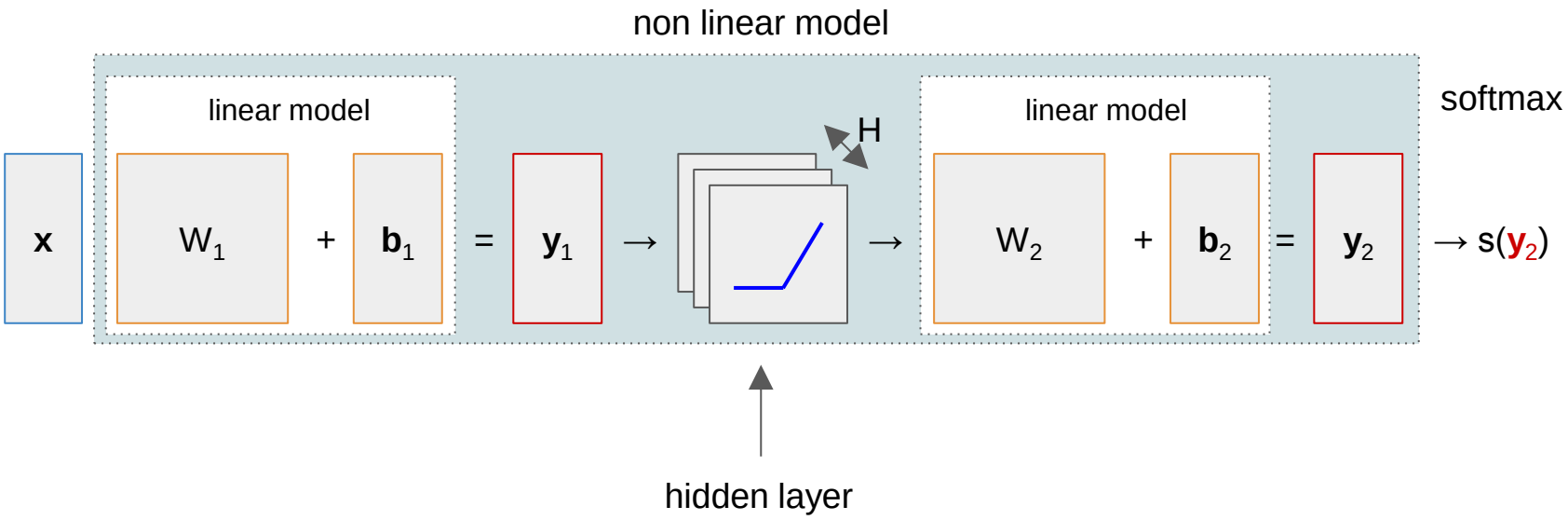
derivative





http://www.kdnuggets.com/2016/03/must-know-tips-deep-learning-part-2.html

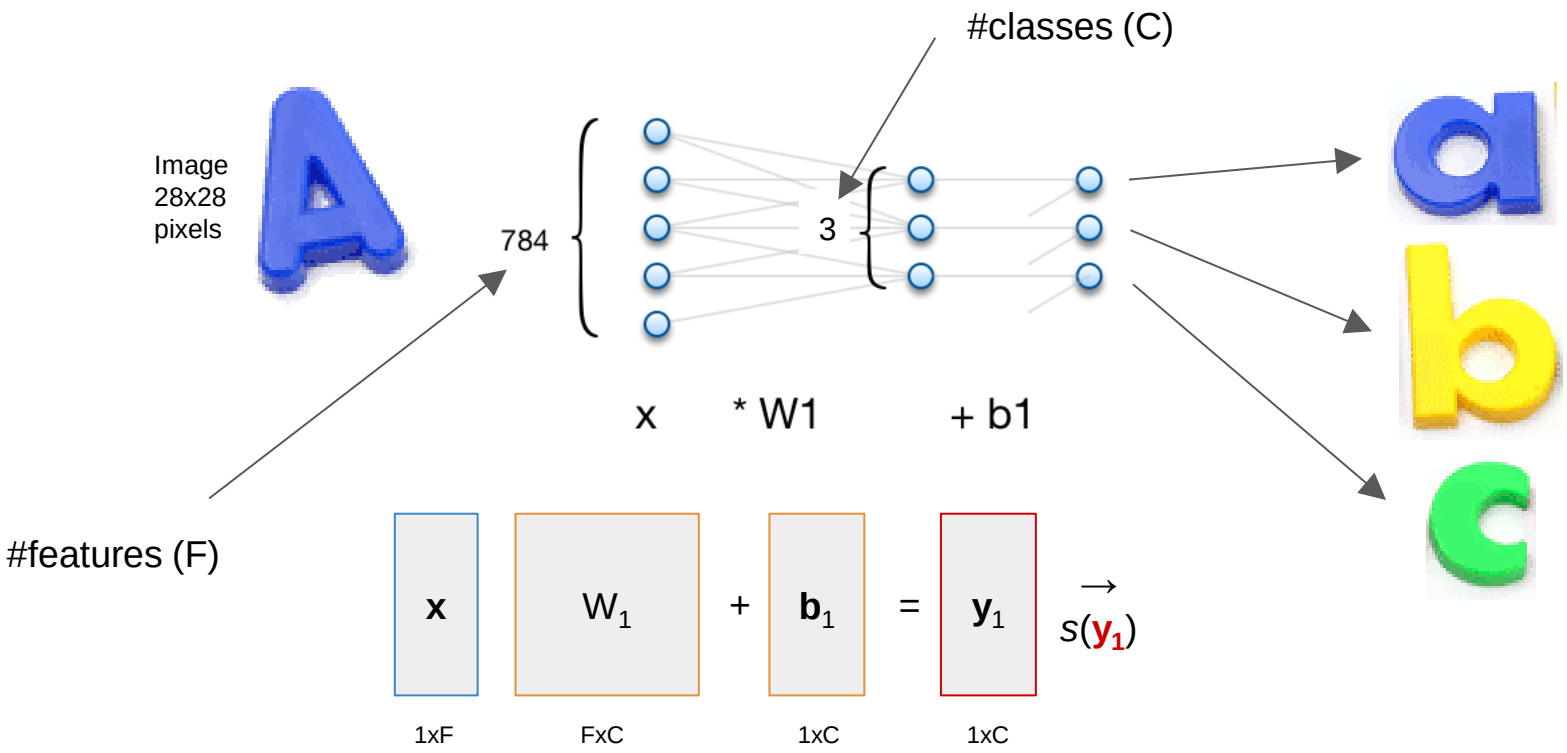
Résumé



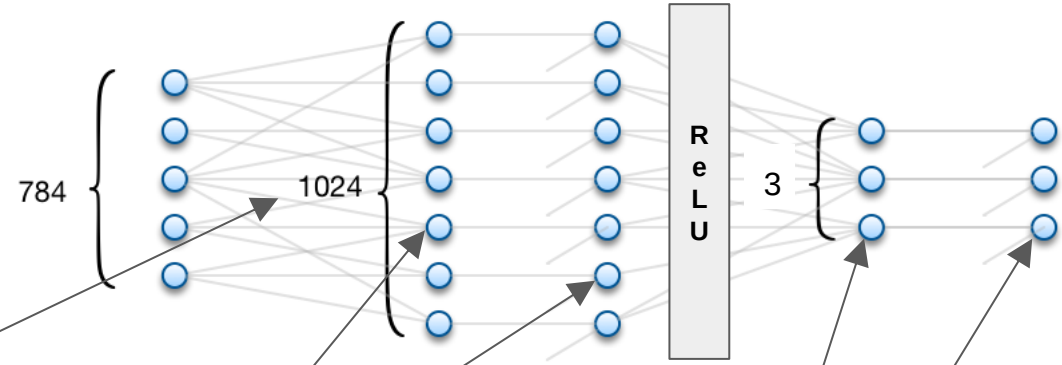
not very deep yet...

H *= number of "neurons" in the hidden layer*

Practical example (1/2)

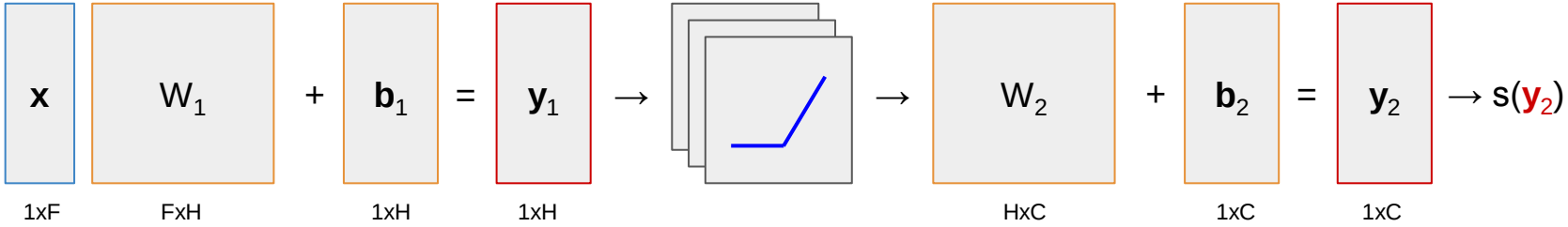


Practical example (2/2)



#hidden_neurons (H)

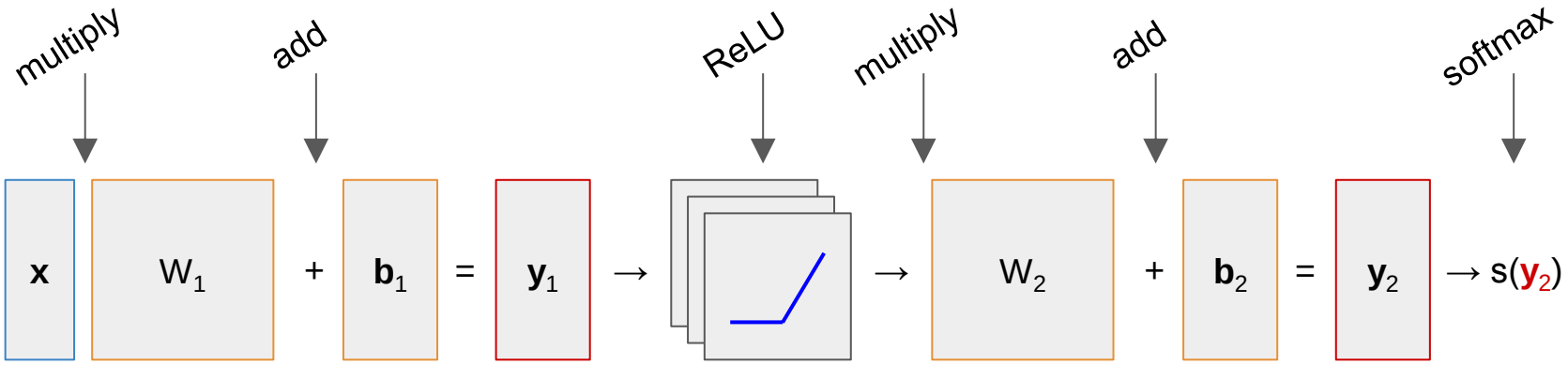
$$\mathbf{x} * W_1 + \mathbf{b}_1 = \mathbf{y}_1 \rightarrow \mathrm{ReLU}(\mathbf{y}_1) * W_2 + \mathbf{b}_2 = \mathbf{y}_2$$



Let’s do math

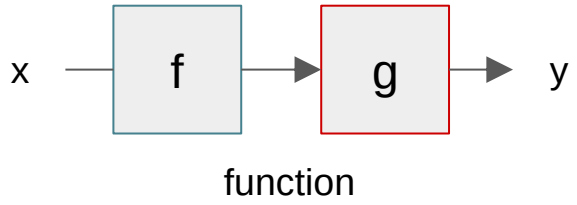
Stacking up simple operations makes the math easier to understand...

the computation of the derivative of the function (used for training the network) becomes very easy



The chain rule

$$[g(f(x))]' = g'(f(x))f'(x)$$



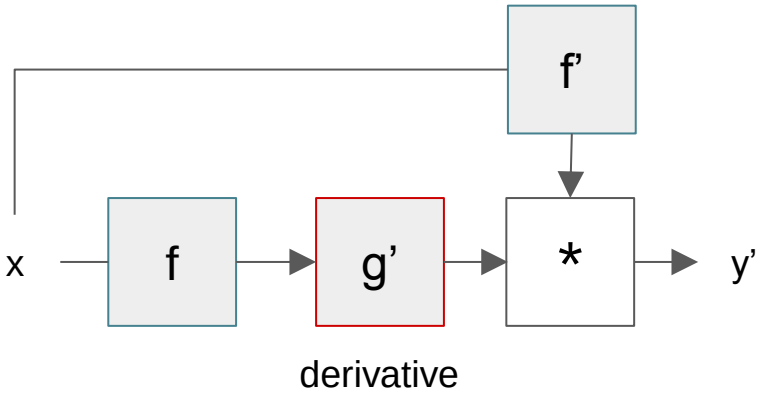
You can compute the derivative of a function by taking the product of the derivatives of the components.

Pros

- efficient data pipeline
- lots of data reuse

Cons

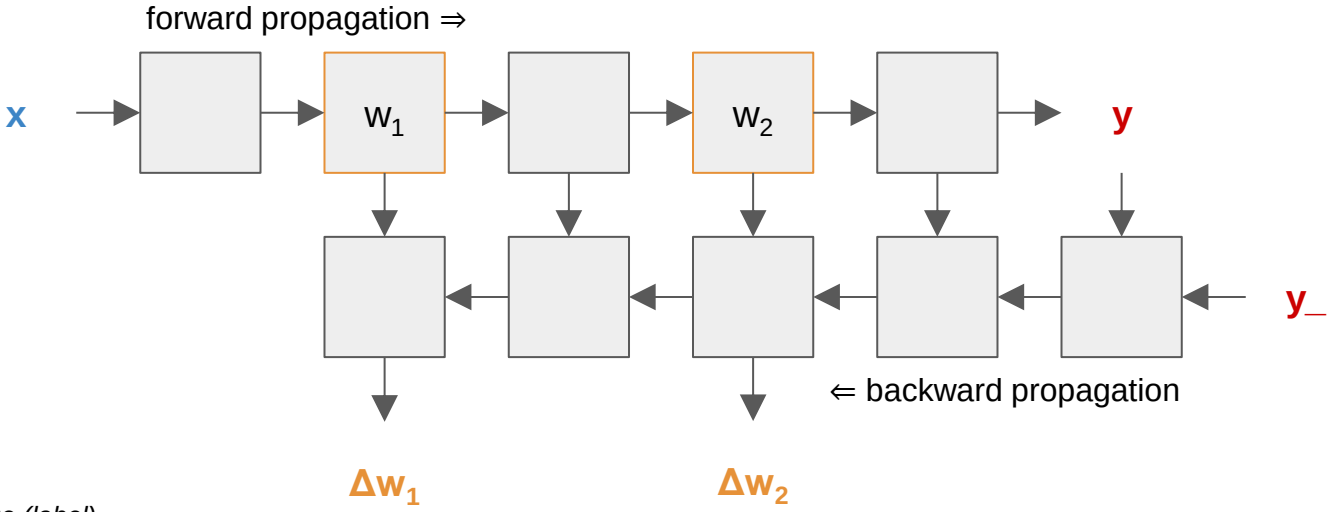
- Memory usage



Back propagation

Algorithm that computes the derivatives of complex functions in a efficient way, used in the Stochastic Gradient Descent algorithm.

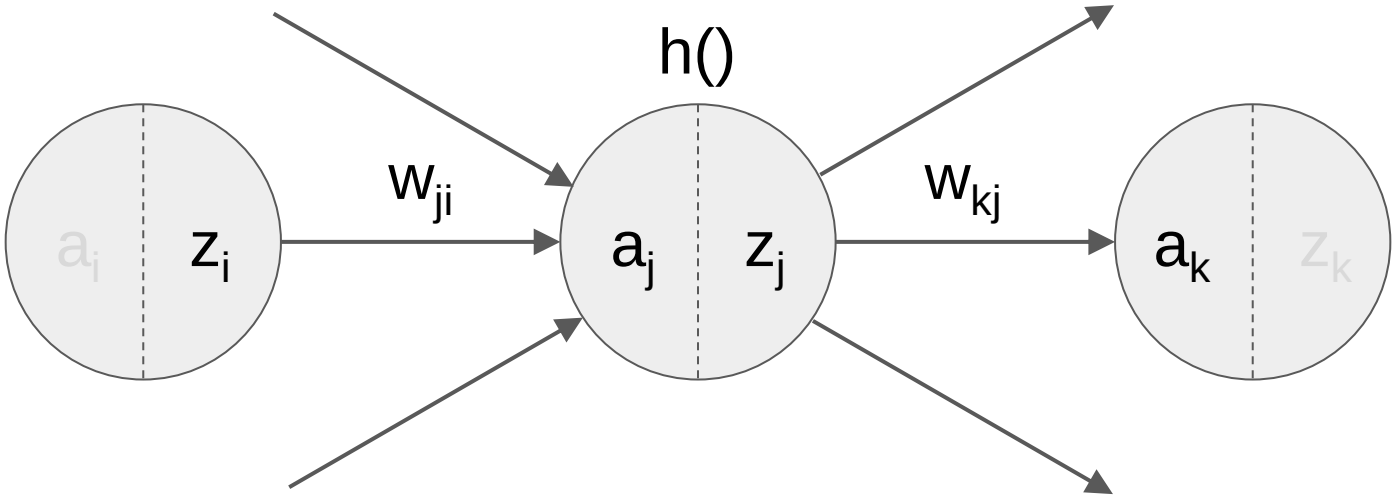
IDEA: thanks to the chain-rule, use the prediction error to refine the weights W and biases **b** of the framework



y_ *= ground truth y-value (label)*

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Back propagation



$$a_j = \sum_i w_{ji} z_i \qquad z_j = h(a_j)$$

h *= activation function (i.e. sigmoid, tanh, ReLU)*

Back propagation

Due to simplicity, we define the loss function as a variation of the MSE:

$$\mathcal{E}_n = \frac{1}{2}(y_n - t_n)^2$$

so that its derivative is

$$\frac{\delta \mathcal{E}_n}{\delta y_n} = (y_n - t_n) \xrightarrow{\text{drop } n \text{ index}} \frac{\delta \mathcal{E}}{\delta y} = (y - t)$$

Note that this is the customizable part of the backpropagation algorithm: according to the task, one could define a different type of loss function.

y *= prediction value;* ***t*** *= target value*

Back propagation

Define the math of the neurons:

$$a_j = \sum_i w_{ji} z_i \qquad\qquad z_j = h(a_j)$$

where the derivatives are

$$\frac{\delta a_j}{\delta w_{ji}} = z_i = h(a_i) \qquad\qquad \frac{\delta z_j}{\delta a_j} = h'(a_j)$$

a *= linear combination of neurons of the previous layer;* ***z*** *= activation value;* ***h*** *= activation function*

Back propagation

Exploiting the chain-rule, we can write:

$$\frac{\delta \mathcal{E}}{\delta w_{ji}} = \frac{\delta \mathcal{E}}{\delta a_j} \frac{\delta a_j}{\delta w_{ji}}$$

and then define the formulation of the backprop in two steps:

- base case
- inductive case

Back propagation: base case

depends on the loss function

In the last layer there is no activation function, so $\mathbf{a_j = y_j}$ and then:

$$\delta_j = \frac{\delta \mathcal{E}}{\delta a_j} = \frac{\delta \mathcal{E}}{\delta y_j} = (y_j - t_j)$$

Now, we must propagate the error through the network until we reach the input layer.

Back propagation: inductive case

Thanks to the chain rule, we have:

$$\delta_j = \frac{\delta \mathcal{E}}{\delta a_j} = \sum_k \frac{\delta \mathcal{E}}{\delta a_k} \frac{\delta a_k}{\delta a_j} = \sum_k \delta_k \frac{\delta a_k}{\delta a_j}$$

We substitute the equation of $\mathbf{a_k}$ and $\mathbf{z_j}$ to expand the previous formula:

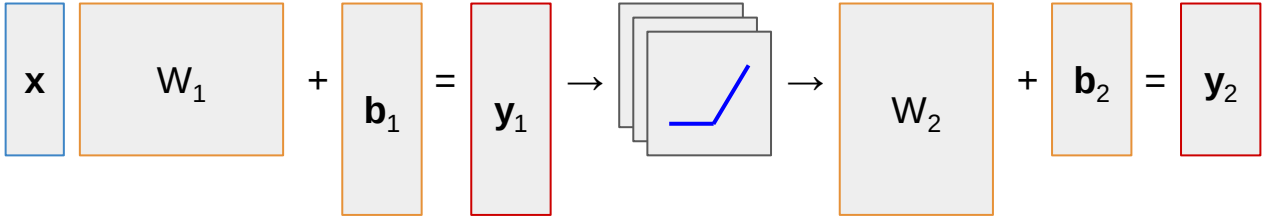
$$\delta_j = \sum_k \delta_k \frac{\delta \sum_{j'} w_{kj'} z_{j'}}{\delta a_j} = \sum_k \delta_k \frac{\delta \sum_{j'} w_{kj'} h(a_{j'})}{\delta a_j} = \sum_k \delta_k w_{kj} h'(a_j)$$

The inductive pass of the algorithm is now:

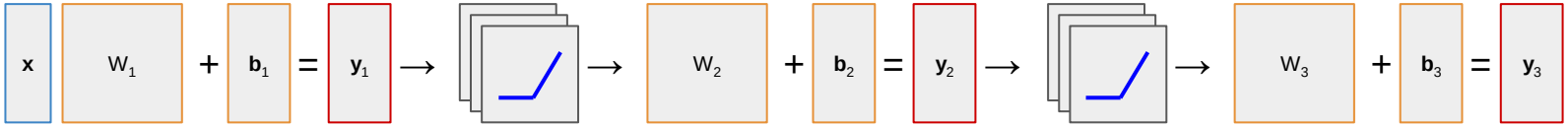
$$\delta_j = h'(a_j) \sum_k \delta_k w_{kj}$$

The iteration ends when $\mathbf{z_j} = \mathbf{x_j}$.

Deep Neural Networks (DNN)



wider or deeper?



Deep Neural Networks (DNN)

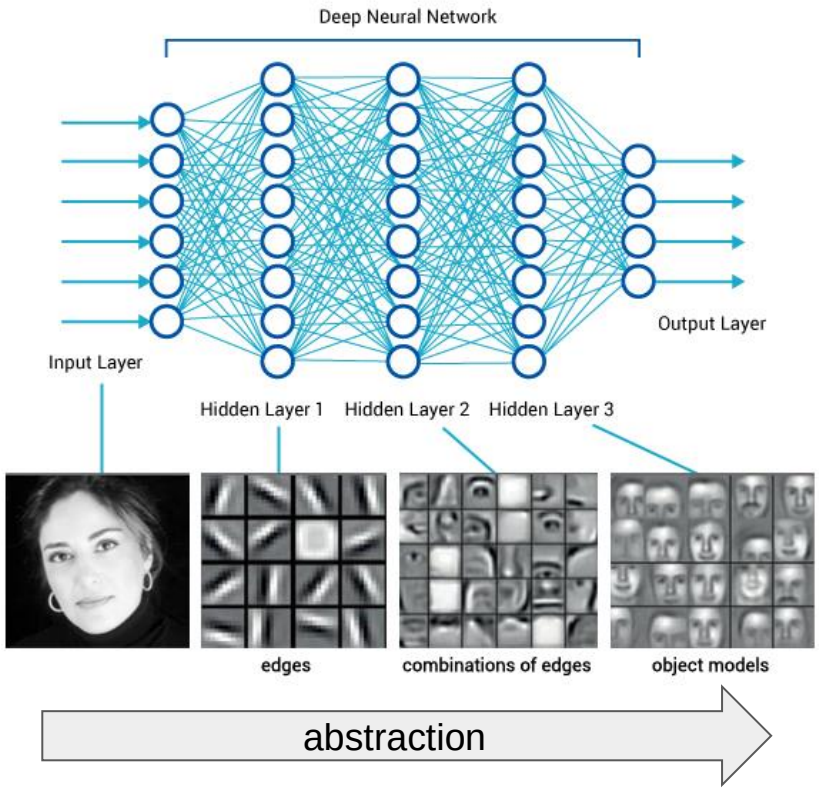
The *deep* choice…

Pros:

- **parameter efficiency**: much more performance
- **hierarchical structure**: a lot of natural phenomena could be represented since each layer describes a specific level of abstraction of the input

Cons:

- needs a **huge amount of data**
- needs **regularization**
- **In the case of fully connected architectures, tons of params to tune**



Regularization

Prevent overfitting

DNNs are very hard to optimize

so...

Train a bigger model and do
your best to prevent overfitting

How?

Regularization

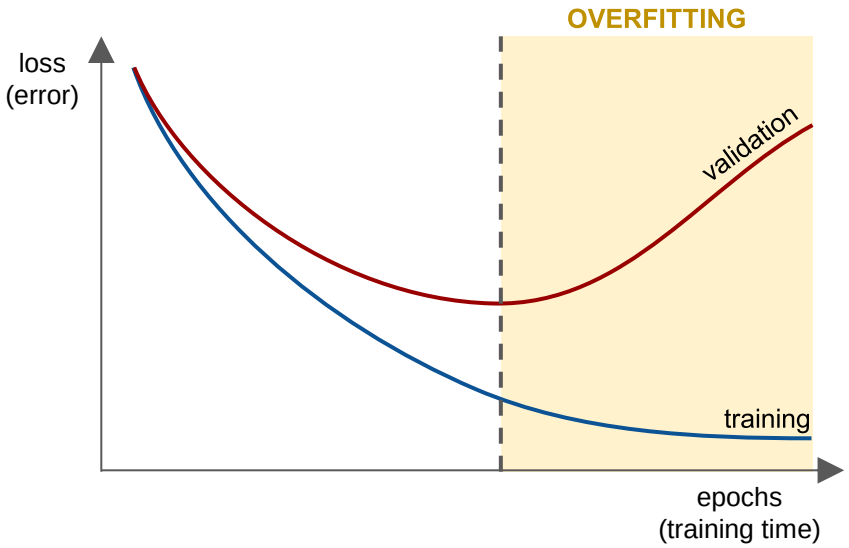
early
termination

ridge
regression

dropout

Regularization: early termination

- IDEA: stop to train as soon as the network stops improving the performance of the validation set



Regularization: ridge regression

- <u>IDEA</u>: apply artificial constraints on the network to reduce the number of free parameters

In other words, we force the network to prevent big disparity among the (absolute) value of the weights

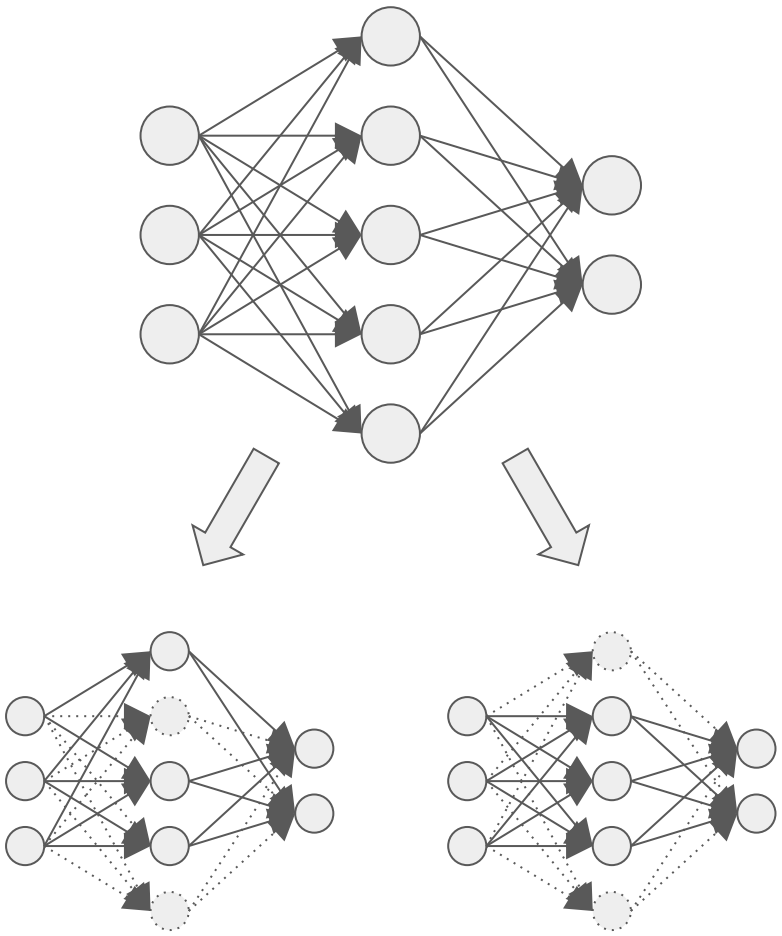
$$\mathcal{L}' = \mathcal{L} + \boxed{\beta \frac{1}{2} \|W\|_2^2} = \mathcal{L} + \beta \frac{1}{2} \sum_i w_i^2$$

new loss

add **regularization term** (L_2 norm)
to the loss function
which penalizes large weights

β *= penalty factor (real number)*

Regularization: dropout

- <u>GOAL</u>: prevent the neurons to co-adapting
- <u>IDEA</u>: randomly switch off the neurons (set the weights to zero) for each learning step on one training input.

Dropout simulates **different** networks that are trained to represent the same input data in many ways (**redundant**).