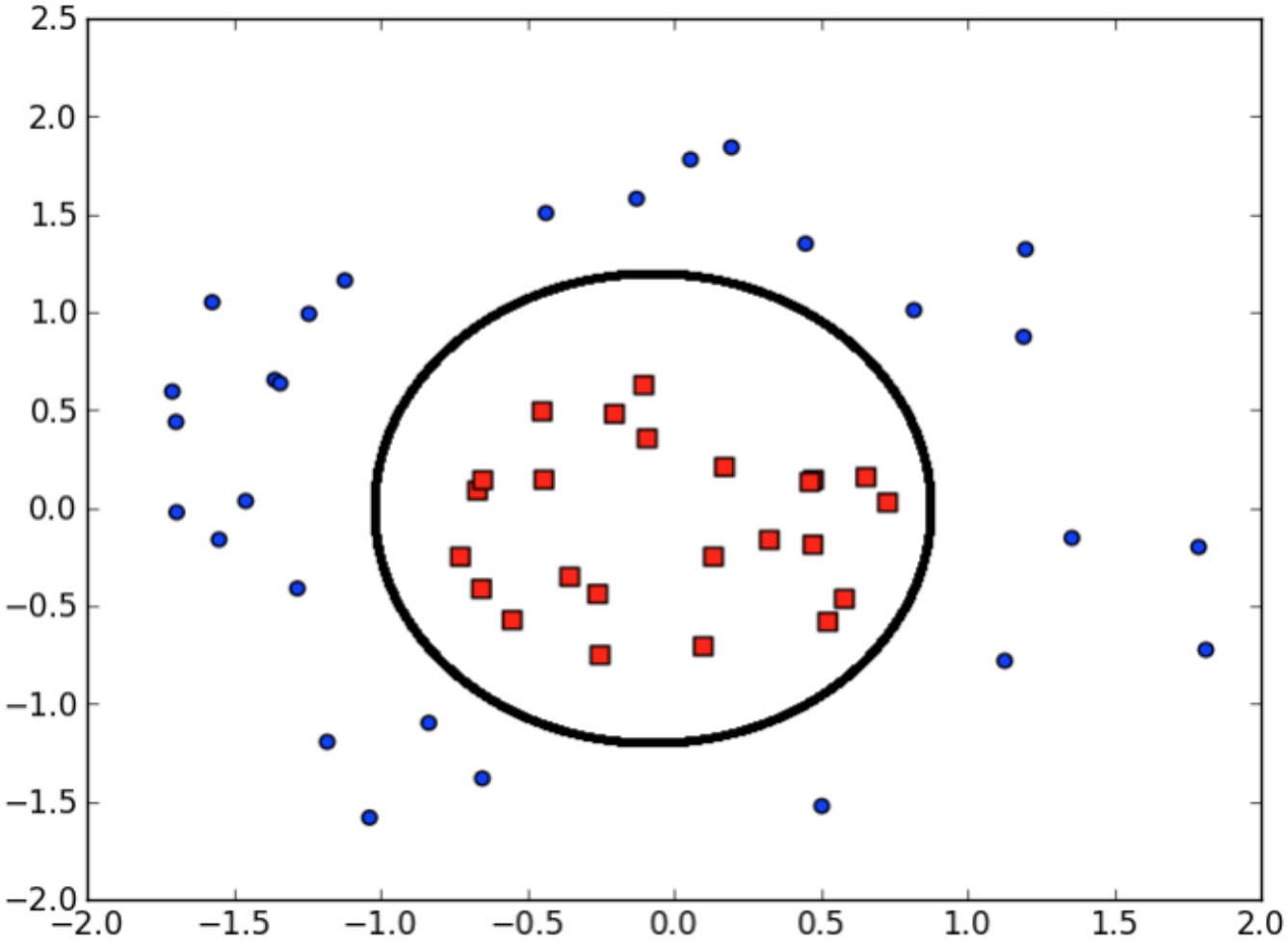
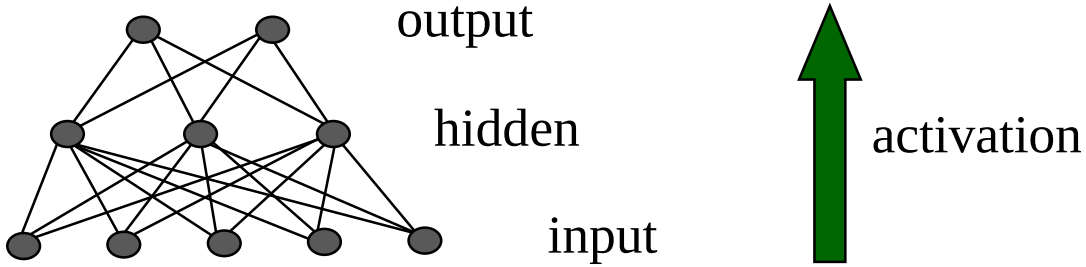
Neural Networks: multi-layer networks and the backpropagation model

Perceptrons are binary classifiers.
They learn a single linear boundary.
What if data not linearly separable?



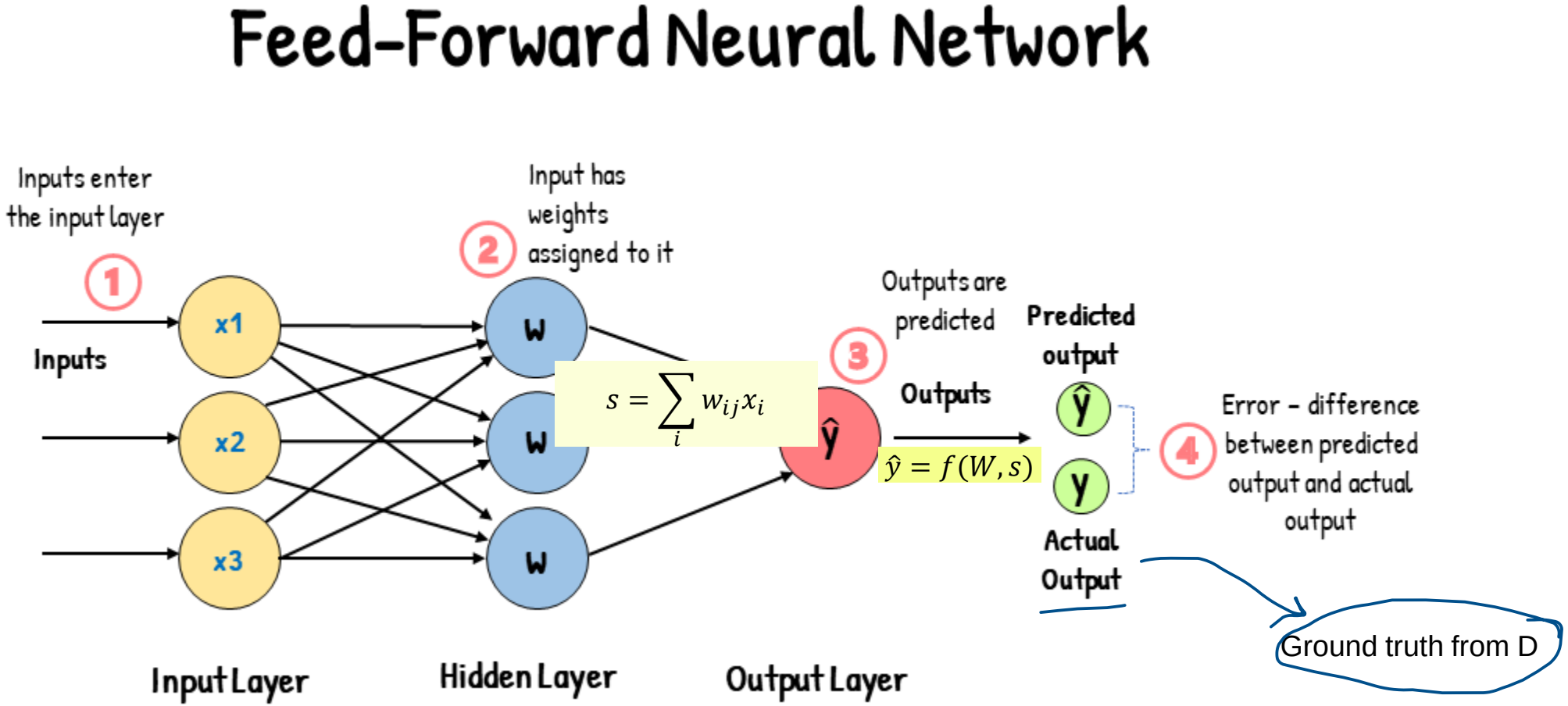
Multi-Layer Networks: architecture

Multi-layer networks can represent **arbitrary functions.** A typical multi-layer network consists of **input**, **hidden** and **output layers**, each layer fully connected to the next, with activation feeding **forward**.



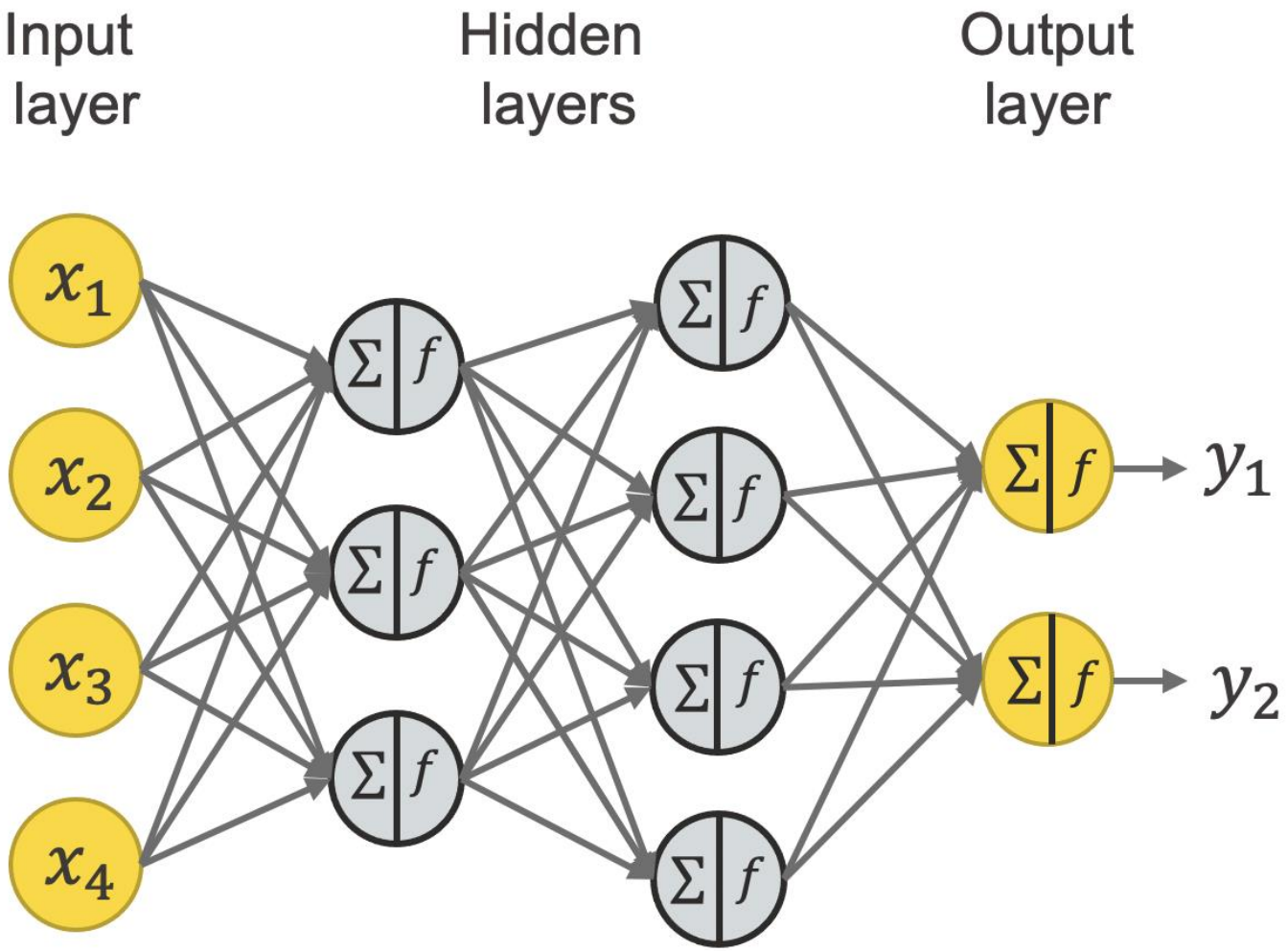
- **Input nodes do not have activation functions**. Thus, they are placeholders for the input feature values of instances **x:** $<x_1..x_m>$.
- Edges between nodes are weighted by w_{ij} (indexes i, j **now** identify **nodes** in adjacent **layers i→j**)
  - Now, an example instances **x** from the training set D, is a pair of vectors ($<x_1, x_2,..., x_m>$, $<y_1, y_2,..., y_K>$), **called the feature vector and ground-truth vector. Note that now we can have multiple outputs (k), and m in general is different from k!**
  - **Parameters to be learned :** the w_{ij}
  - **Hyperparameters to be selected:** the network architecture (layers, hidden nodes..) the neuron activation functions φ, and (especially for deep networks) many others

Multilayer networks: the basic computation step



There might be very many hidden layers

There might be multiple layers and multiple output nodes



Multilayer networks

- **Target is to learn the weights (including threshold/bias) of edges such as to minimize the error** between the output observed on output nodes $\hat{y}_j$ and the **true** output values y_j in training set D
- Differently from DT and Perceptron, the ground truth vector $<y_1, y_2, \ldots, y_K>$ is a complex output functions of real (regressor) or discrete (classifier) values
- Note that we would need 4 indexes now: i and j to identify connections w_{ij} between node pairs in connected layers (i➔j), a superscript (h) to identify an example in D (i.e $(<x^h_1, x^h_2, \ldots, x^h_m>, <y^h_1, y^h_2, \ldots, y^h_K>)$), and another superscript [L] to identify layers in the NN (e.g. $w_{ij}^{[L]}$.) We will omit h and [L] whenever possible, to avoid overloading the notation.

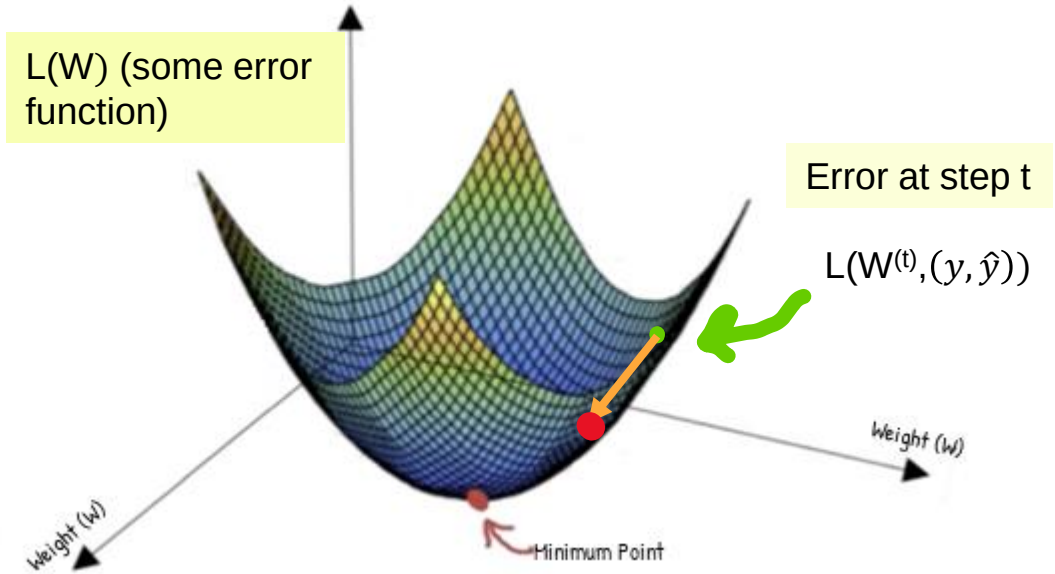
Gradient descent

- Weight updating rule is based on the Hill-Climbing heuristic, like for perceptron.
- In a **Hill-Climbing** heuristic:
  1. We start with an initial solution (a **random** set of weights ***W*** (***W*** denotes a matrix).
  2. Generate one or more "neighboring" solutions (weights which are "close" to previous values) with the objective of reducing the error.
  3. Pick the best neighboring solution according to some criterion, and continue until there are no better neighboring solutions
- In the case of the neural network, **gradient descent** is used to identify the "best" neighboring solutions.

Gradient and Gradient Descent

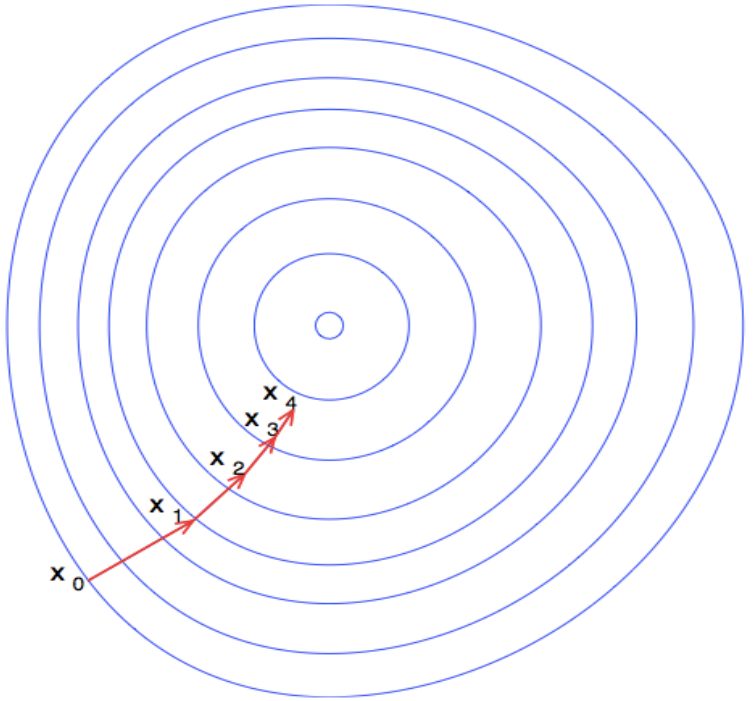
- The **gradient** of a **scalar field x** is a **vector field** $\overrightarrow{x'}$ that points in the direction of the *greatest rate of increase (or decrease)* of the scalar field, and whose **magnitude** is that rate.
- In simple terms, the variation of any quantity – e.g. an error function - can be represented (e.g. graphically) by a **slope**. The gradient represents the **steepness** and **direction** of that slope.
- Note: Finding the gradient is essentially finding the **derivative** of a function.

- Example of a simple error function with 2 parameters



Gradient and Gradient Descent

To find a **local minimum** of a function (e.g., **error**(x)) using the gradient descent, one takes (small) steps proportional to the *negative* of the **gradient** (or the approximate gradient) of the function at the current point (gradient=derivative)



Gradient and Gradient Descent

- **The objective of the learning algorithm, then, is to find the parameters *W (all the coefficients w_{ij} for all layers)* which give the «minimum possible cost» of some optimization problem – called *Loss(W).***
- For the Gradient Descent of "basic" NNs, the Loss is computed by the **Mean Squared Error function (MSE)**. The MSE for the multilayer NN:

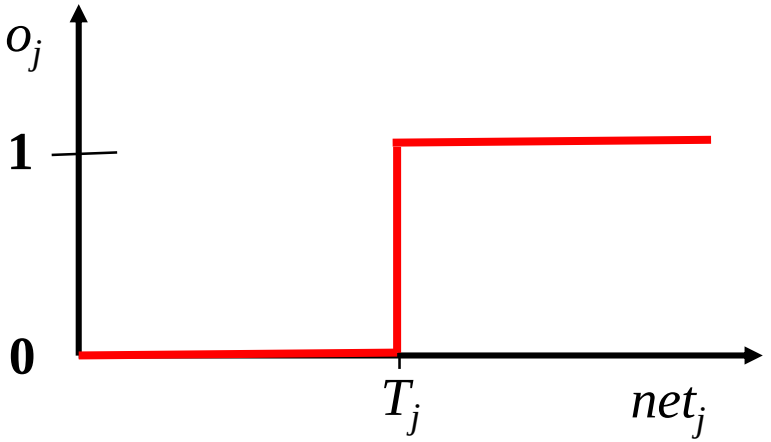
$$Loss(W) = \frac{1}{|D|}\sum_{h\epsilon D}\sum_{j=1..k}\left((\boldsymbol{y}_j^h - \hat{\boldsymbol{y}}_j^h)^2\right)$$

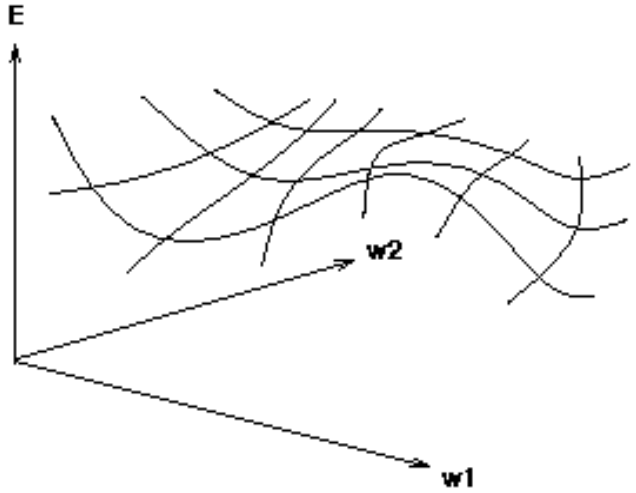
  - ➢ *D* is the set of training examples $\mathbf{x}^h$
  - ➢ n = |D|; the number of training instances (h: 1..n)
  - ➢ k is the number of **output** nodes of the NN (we have output vectors)
  - ➢ **W** is the set of weights and to simplify, it **includes the thresholds/bias** (that can be represented as the weight w_o of a dummy node with constant value 1).
  - ➢ $\boldsymbol{y}_j^h$ is the "ground truth" output for an example $\mathbf{x}^h$ in D and $\hat{\boldsymbol{y}}_j^h$ is the observed output for $\mathbf{x}^h$ with the current weights

- The **Loss function** is also called **Cost function or Error function**. There are many loss functions (will see some). See link, link

Gradient descent needs a differentiable activation function

- As we said, the gradient is a derivative
- To do gradient descent of a Loss function, which depends on the y_j, we need the activation ($y_j = f(net_j)$) function of neurons to be a **differentiable function of its input and weights.**
- The **binary step function** of the Perceptron is **not differentiable**.





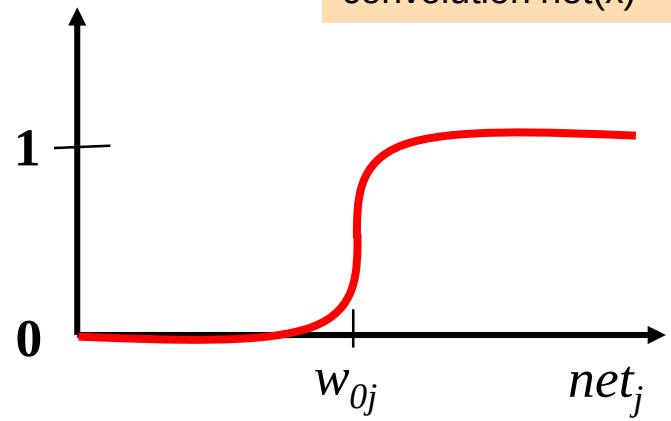
Differentiable activation function

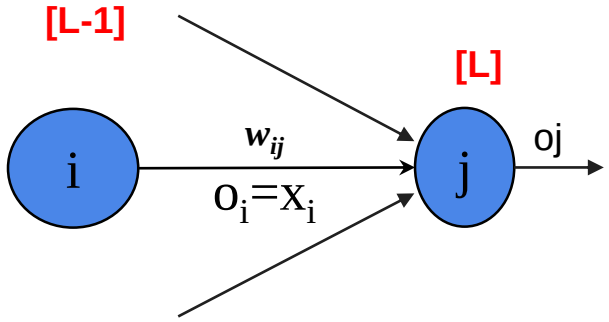
A common activation function is the non-linear,
differentiable sigmoid function (aka. logistic function):

$$o_j = \sigma(net_j) = \frac{1}{1+e^{-net_j}}$$

$$net_j = \sum_i w_{ij} x_i + w_0$$

Note: the threshold or bias
Is commonly denoted with symbols T or or w_0 or θ **In any case**, we are simply adding a constant to the convolution net(x)

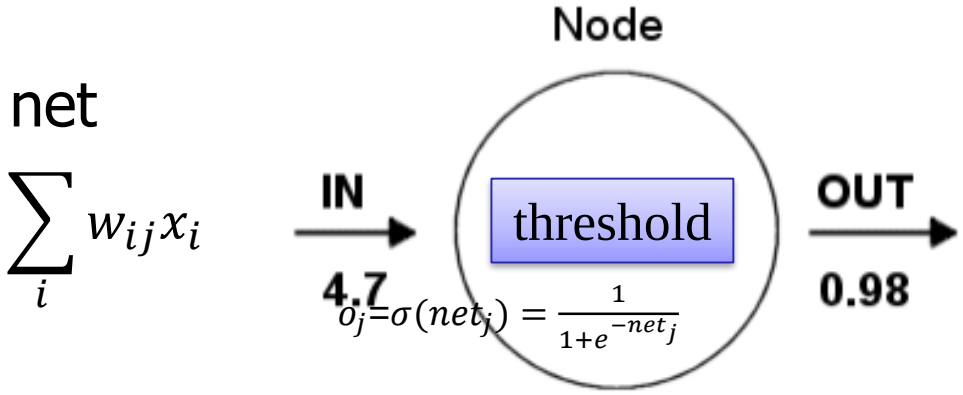




Note: for multiple layers, **the input values x_i to node n_j in layer L is the output o_i of previous nodes in layer L-1**, and w_{ij} is the synaptic weight of the connection i➔j.

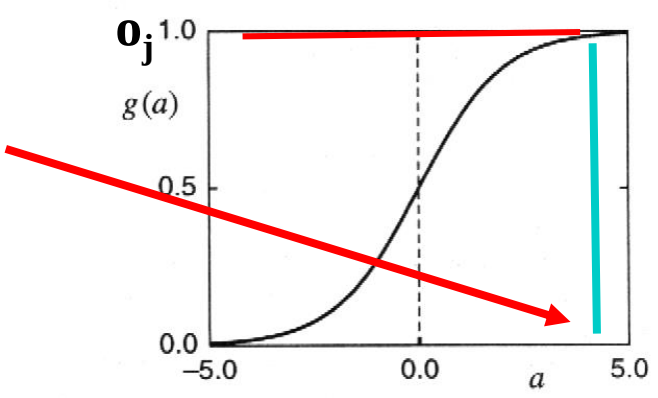
Example:

Feeding data through the NN with the Sigmoid Activation function)



The activation function **limits** the node output values in [0 1]
(threshold is 0 in the figure):

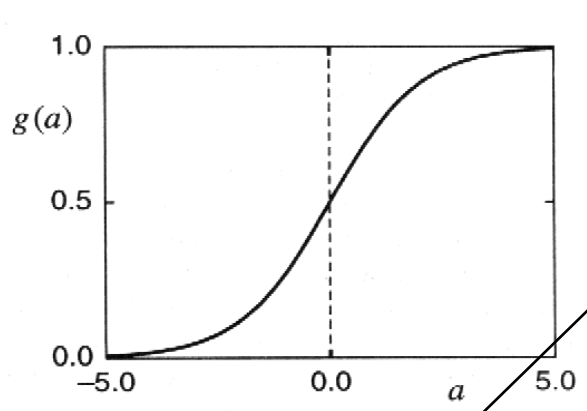
$$o_j=\sigma(net_j)=\frac{1}{1+e^{-net_j}}$$

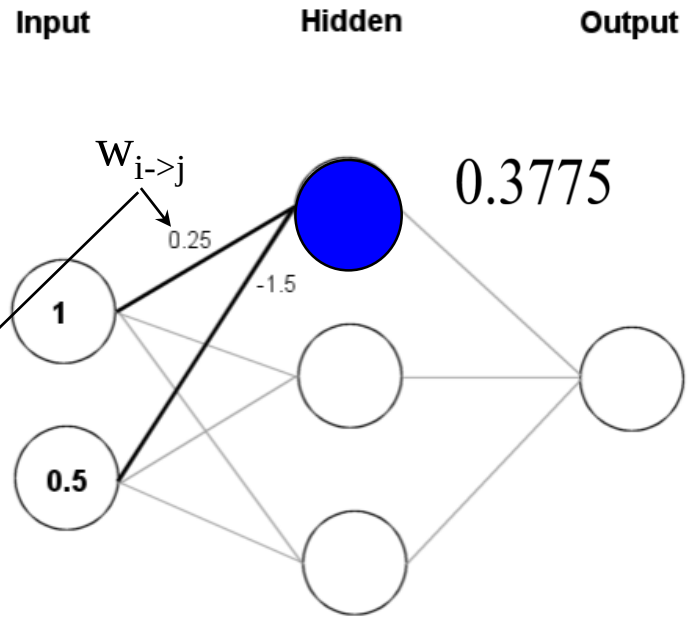


Example (2):

Feeding data through the NN with the Sigmoid Activation function (with threshold 0)

x:<x_1,x_2>=<1,0.5>





net_j= (**1** × 0.25) + (**0.5** × (-1.5)) = 0.25 + (-0.75) = **- 0.5**

Thresholding with σ:

$$\frac{1}{1+e^{0.5}} = 0.3775$$

Back to Gradient Descent

- The «classic» NN model uses the MSE (mean square error) as a **loss function** to compute the error to be minimized ($\hat{y}$ is the output estimate by the current network)

$$Loss(W) = \frac{1}{n} \sum_{h=1..n} \sum_{j=1..k} \left((y_j^h - \widehat{y_j^h})^2 \right)$$ n=|D|

- A popular modification of this formula is to multiplying it by ½ so that when we compute the derivative, the 2s cancel out for a cleaner formula:

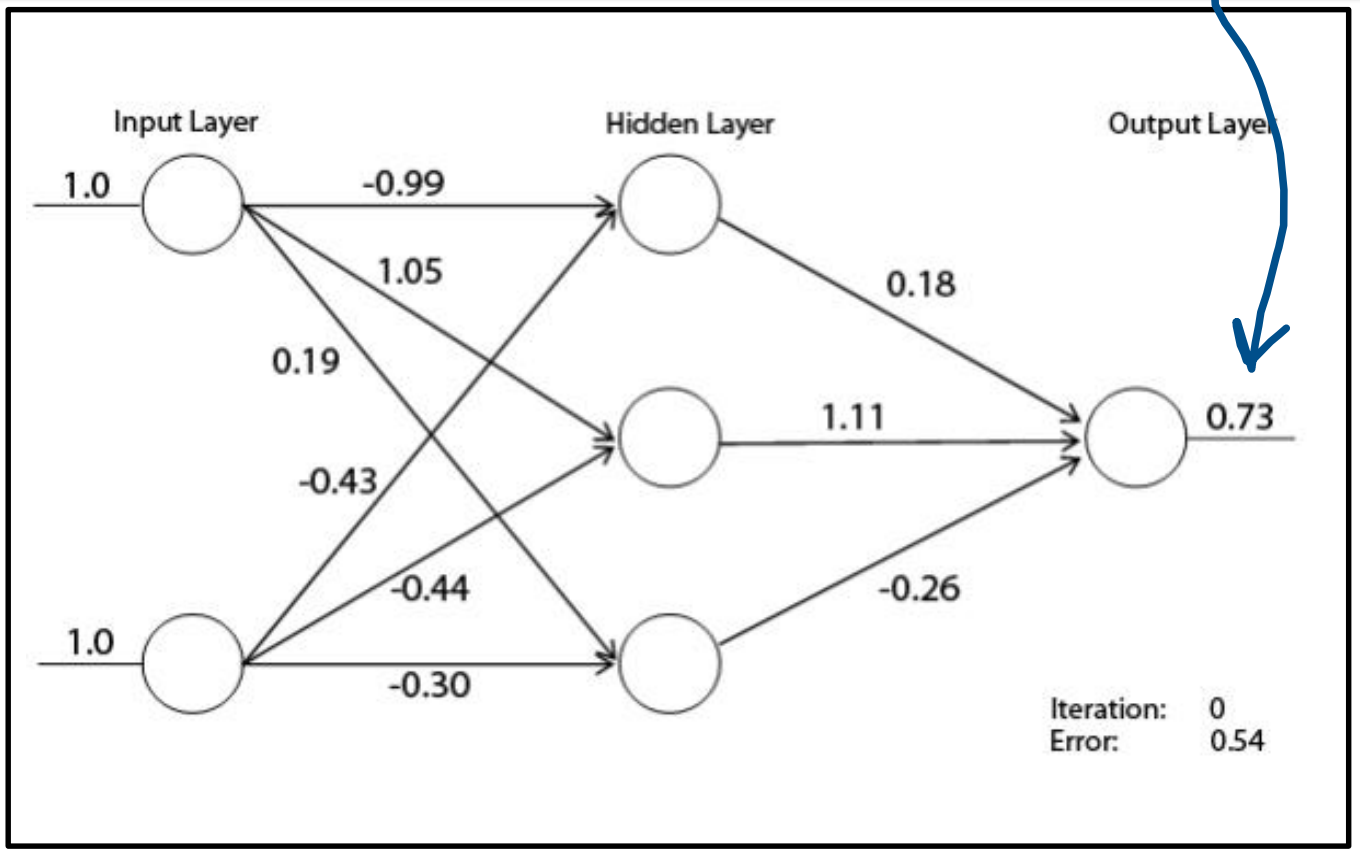
$$Loss(W) = \frac{1}{2n} \sum_{h=1..n} \sum_{j=1..k} \left((y_j^h - \widehat{y_j^h})^2 \right)$$

- For derivation, the gradient descent algorithm will consider only **one input-output pair** (the inner sum of the total Loss) **at each iteration**. Once this is derived, the general form <u>for all input-output pairs</u> can be generated by summing the individual gradients. The error function is therefore simplified as: $Loss(W) = (1/2 \sum_j (y_j^h - \widehat{y_j^h})^2)$

Example of loss computation:

1 output node

Input instance **x**<(1.0,1.0),0.50>; Loss(W,**x**)=1/2(0.50-0.73)^2=0.0529/2



How can we minimize the error ?

$$Loss(W) = \left(\frac{\sum_{h=1..n} \quad \sum_{j=1..K} (y_j^h - \widehat{y_j^h})^2}{2n}\right)$$

- Gradient descent implies taking the derivative of the Loss function. To compute a derivative, we must re-write the Loss function in a way that «highlights» the relevant variables
- First of all, what are our relevant «variables» (what can be changed in the network)?
- **The network weights w_{ij}**! In fact the output $o_j^{h,[L]}$ in each layer L are computed by feeding the network with data INSTANCES $\mathbf{x}^h$ (i.e., the **values** of the features of the training instances). These values propagate through the networks and are weighted, summed, and thresholded on each neuron, from input up to the output nodes.
- Therefore the final output vector $\hat{\mathbf{y}}^h$ depends on the weights (parameters) and on the type of activation function that generates the output of each neuron (note that the activation function is an hyperparameter, not adjustable during training).
- **So, to minimize the error, we can only adjust the weights!** and change the activation function (hyperparameter), but the latter is to be chosen a priori

Gradient Descent updating rule:

Hill Climbing in the Gradient Descent

- **How can we apply gradient descent to learn weights in a neural network?** The idea is to use gradient descent to adjust the weights to minimize the error (**loss function**) at each iteration (similarly to Perceptron):

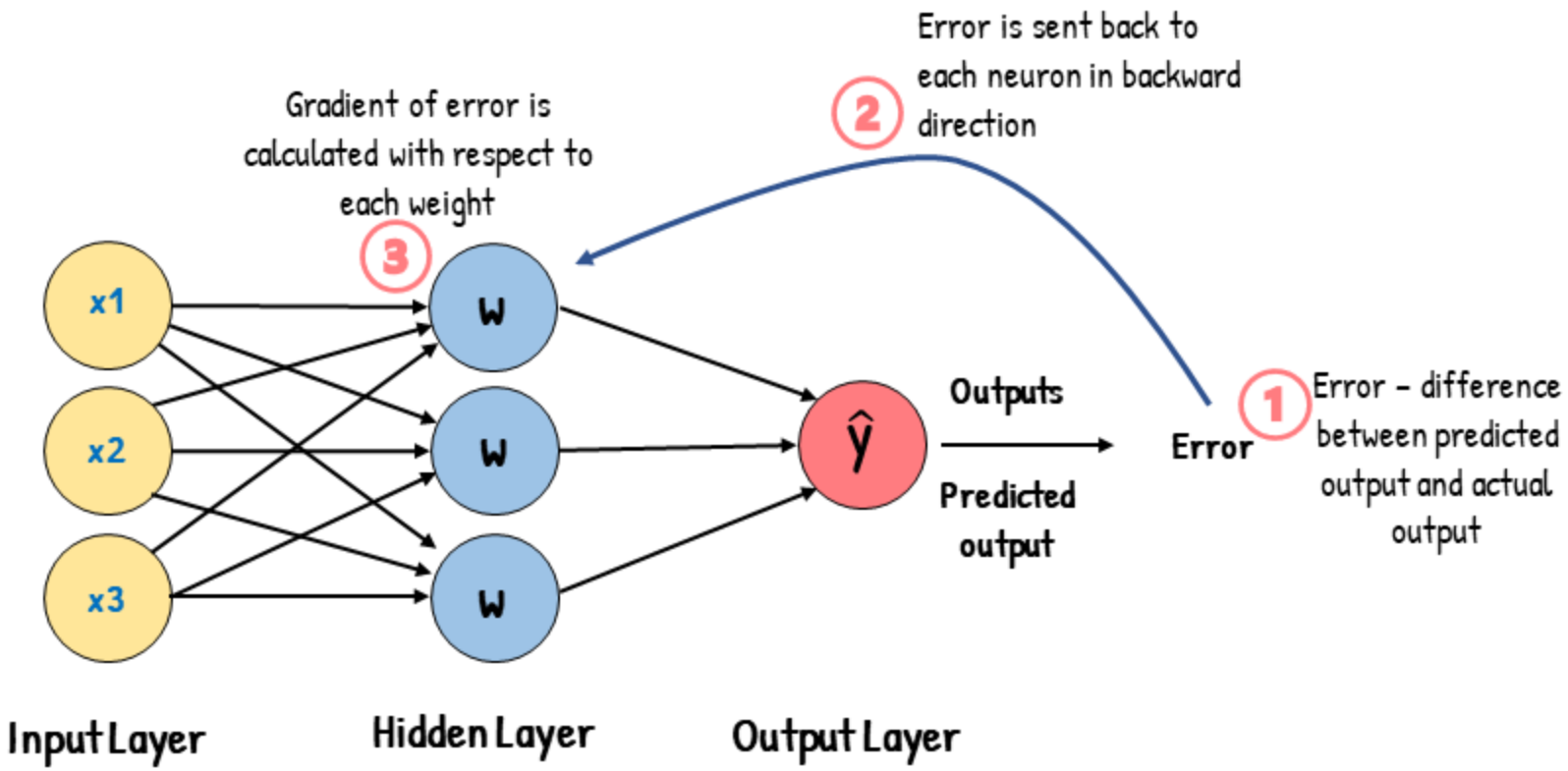
$$w_{ij} := w_{ij} + \Delta w_{ij} = w_{ij} - \eta \frac{\partial Loss(w_{ij})}{\partial w_{ij}}$$

- Each weight w_{ij} (note that we omit the superscript L indicating the layer and the h superscript whenever possible) is updated proportionally to (η) the *partial derivative* (∂(Loss)/ ∂w_{ij}), e.g. the fraction of the gradient of the total **loss that can be «imputed»** to the weight w_{ij}
- η is a hyperparameter (like the activation function and the Loss function) and is usually <<1
- **Hill Climbing**: by repeatedly applying this updating rule to **all weights** and all instances of the training set D, we can "roll down the hill", and hopefully find a minimum of the cost function.
- BUT: **the problem is that in the formula of the loss, we don't have the weights!**

How do we go about?

- We define some Loss function, e.g. $Loss(W) = (\frac{\sum_{h \in D} \quad \sum_{j \in K} (y_j^h - \widehat{y_j^h})^2}{2n})$
- But, in order to update the weights *W* of the NN, we need to compute $\frac{\partial Loss()}{\partial w_{ij}^{[L]}}$ for **all** weights and layers.
- Note: $\frac{\partial Loss()}{\partial w_{ij}^{[L]}}$ tells us «how fast» the observed loss changes when a specific parameter $w_{ij}^{[L]}$ of the NN changes, in other words WHAT IS THE CONTRIBUTION of that parameter to the observed error, and next, we use this information to adjust the parameter
- The problem is: we can only observe the output error $(y_j^h - \widehat{y_j^h})$, how do we compute these derivatives?? The relation between the numerator and the parameters $w_{ij}^{[L]}$ is not explicit
- SOLUTION: we model the network with COMPUTATIONAL GRAPHS, were all nodes represent differentiable operators, and we apply an algorithm called BACKPROPAGATION to derive the contribution of each computational step to the total loss

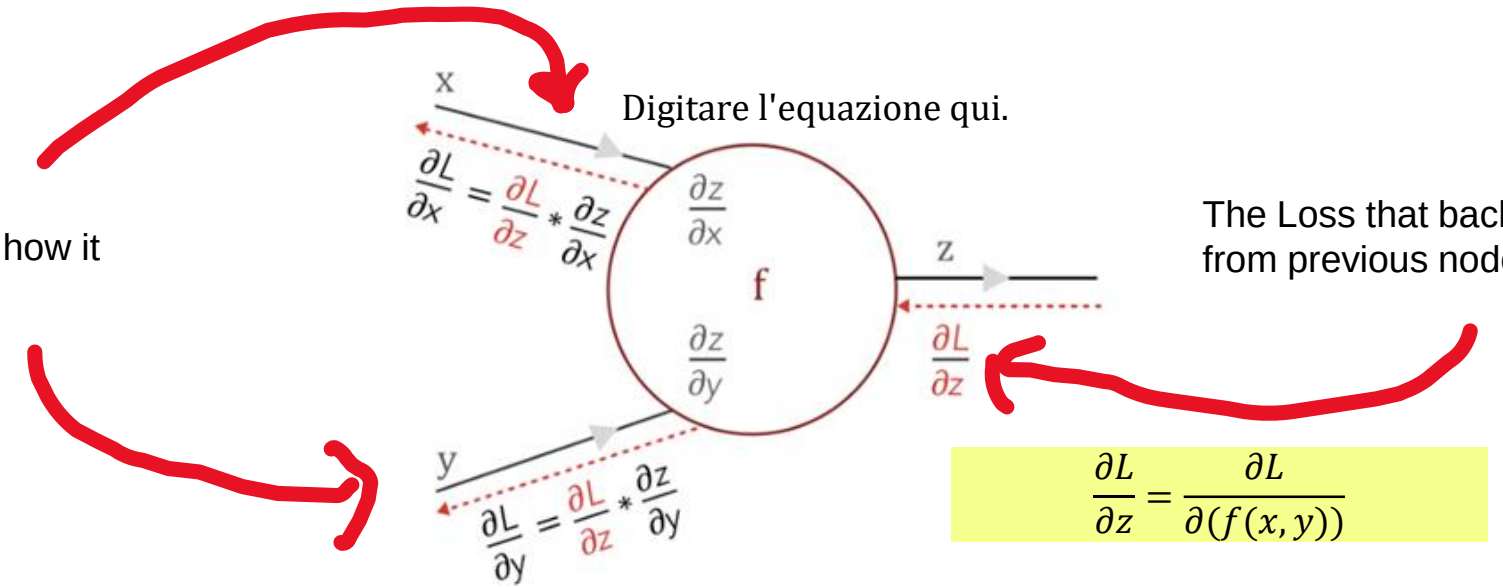
Backpropagation



Backword pass: the basic step

- General backpropagation step in each node of the network (*f* here denotes a generic function computed on a generic node, x, y and z are the input and output of the node $f(x, y) = z$):

The contribution of a generic node $f(x, y)$ to the Loss, and how it propagates backwords



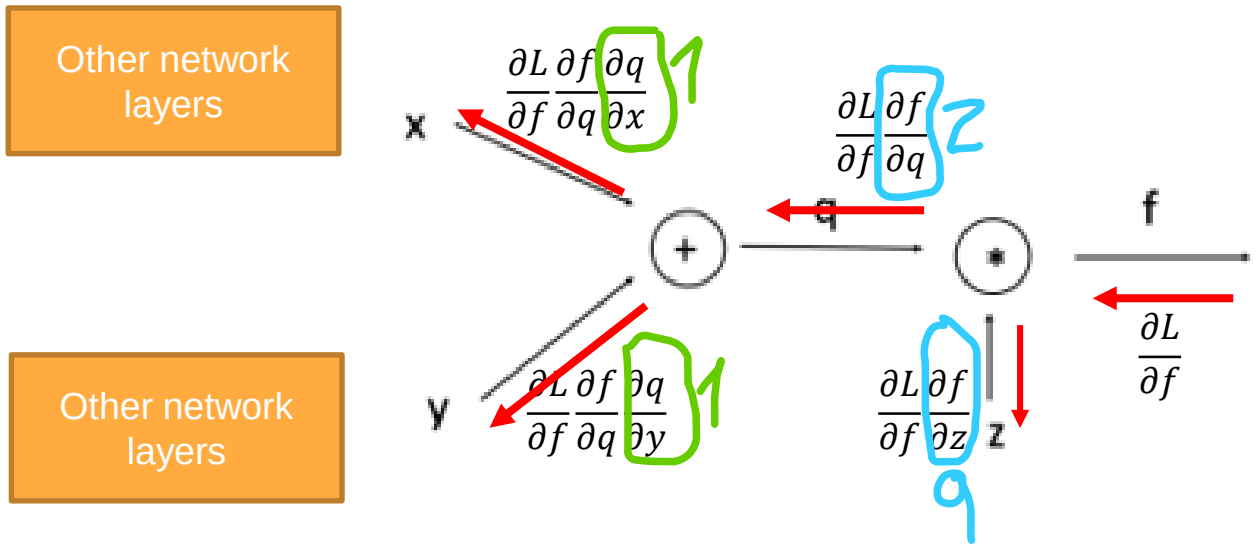
The Loss that backpropagates from previous nodes

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial (f(x, y))}$$

We apply the **derivative chain rule**. Of course, it is important that f is a **DERIVABLE** function

We can compute these derivatives step by step in an easier way using the computational graph representation of NN

- A computational graph is a directed graph where the nodes correspond to operations or variables. By applying the **derivative chain rule**, we can "backpropagate" the gradients
- Example: consider the function $f = (x + y) \times z = q \times z$ within a



The key is to understand how a change in one variable brings change on the variable that depends on it. If **q** directly affects **f**, then we want to know **how it affects f**.

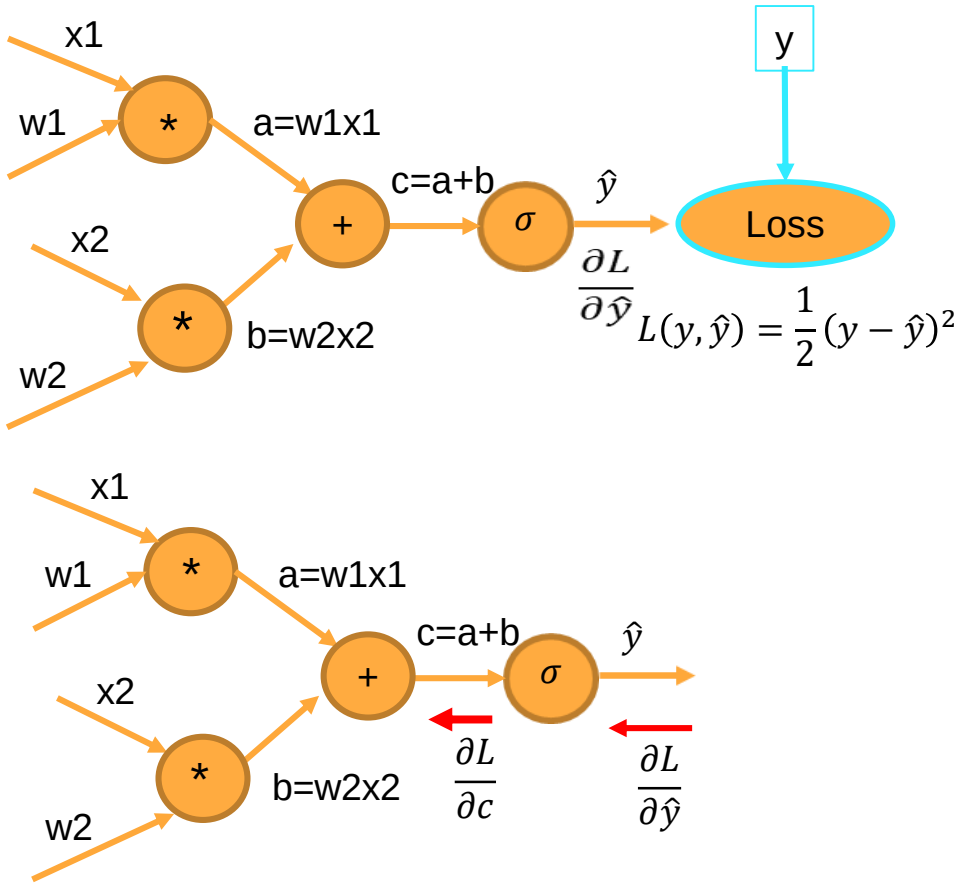
If we make a slight change in the value of **q** how does **f** change? We can term this as the **partial derivative of f with respect to q**.

Note: $\frac{\partial f}{\partial q}$=z and $\frac{\partial f}{\partial z}$=q=(x+y) $\frac{\partial q}{\partial x}$= $\frac{\partial q}{\partial y}$=1

If, e.g., the gradient backpropagating from subsequent computations $\frac{\partial L}{\partial f}$ =2, and x=1, y=2, and z=2

then we have $\frac{\partial L}{\partial f}\frac{\partial f}{\partial q}$=$2 * 2$; $\frac{\partial L}{\partial f}\frac{\partial f}{\partial z}$=$2 *$(1+2); $\frac{\partial L}{\partial f}\frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$=$2 * 2 * 1$; $\frac{\partial L}{\partial f}\frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = 2 * 2 * 1$

Let's see how it works for a simple neuron with sigmoid activation



- **Step 1 compute the partial derivative of the loss**

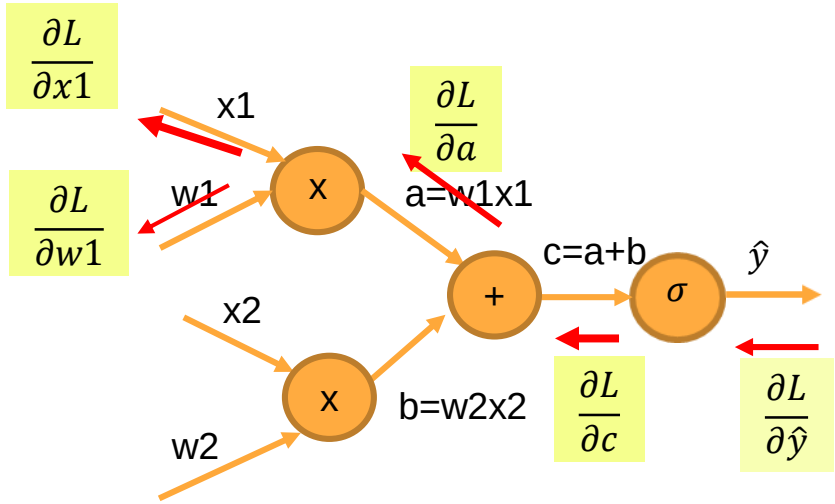
$$\frac{\partial L}{\partial \hat{y}} = (y - \hat{y})$$

- **Step 2** start backpropagating the loss, using the chain rule:

$$\frac{\partial L}{\partial c} = \left(\frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial c}\right) = \frac{\partial L}{\partial \hat{y}}\ \sigma(c)(1-\sigma(c)) = (y-\hat{y})\frac{e^{-c}}{(1+e^{-c})^2} =$$

$(y - \hat{y})\ \hat{y}(1- \hat{y})$

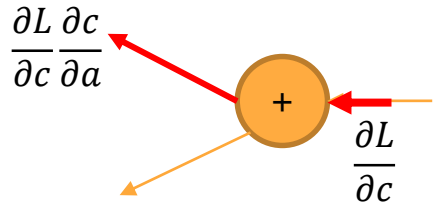
Simple NN backpropagation with CG



Note: during each backpropagation step, x1,x2, w1,w2, y and $\hat{y}$ are NUMBERS!! η is an hyperparameter and is established apriori (e.g., 0,1)

Also note that here we have written $y - \hat{y}$ for simplicity, but since the process is repeated for every example h in the dataset, we should write $\left(y^h - \hat{y}^h\right)$ where h is the superscript that identifies one example in D

- **Step 3: backpropagation form summation node**



$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial c}\frac{\partial c}{\partial a} = \frac{\partial L}{\partial c}\left(\frac{\partial (a+b)}{a}\right) = \frac{\partial L}{\partial c}1 = (y - \hat{y})\frac{e^{-c}}{(1+e^{-c})^2}$$

- **Step 4:** multiplication nodes

$$\frac{\partial L}{\partial w1} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w1} = \frac{\partial L}{\partial a}\left(\frac{\partial (w_1 x_1)}{\partial w_1}\right) = \frac{\partial L}{\partial a}x_1 = (y - \hat{y})\frac{e^{-c}}{(1+e^{-c})^2}x_1$$

Where $c = w_1 x_1 + w_2 x_2$

So, we can update w1!!

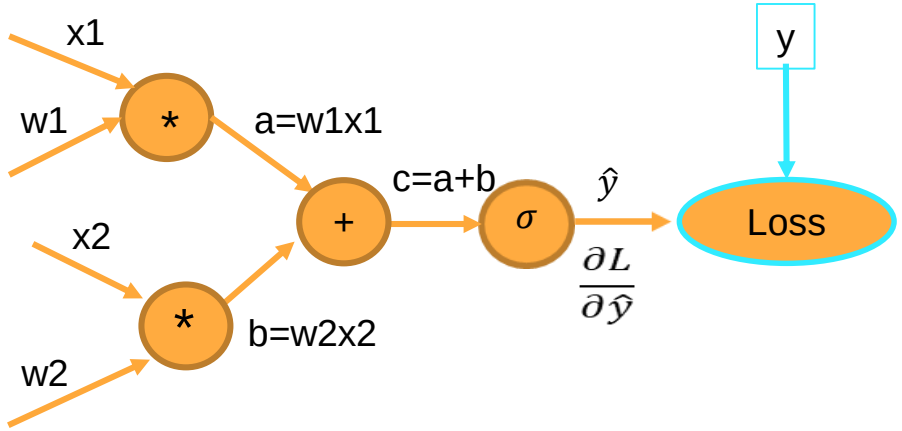
$$w_1 = \left(w_1 + \eta \frac{\partial L}{\partial w_1}\right)$$

Finally, $\frac{\partial L}{\partial x1} = w_1 \frac{\partial L}{\partial a}$

In class exercise

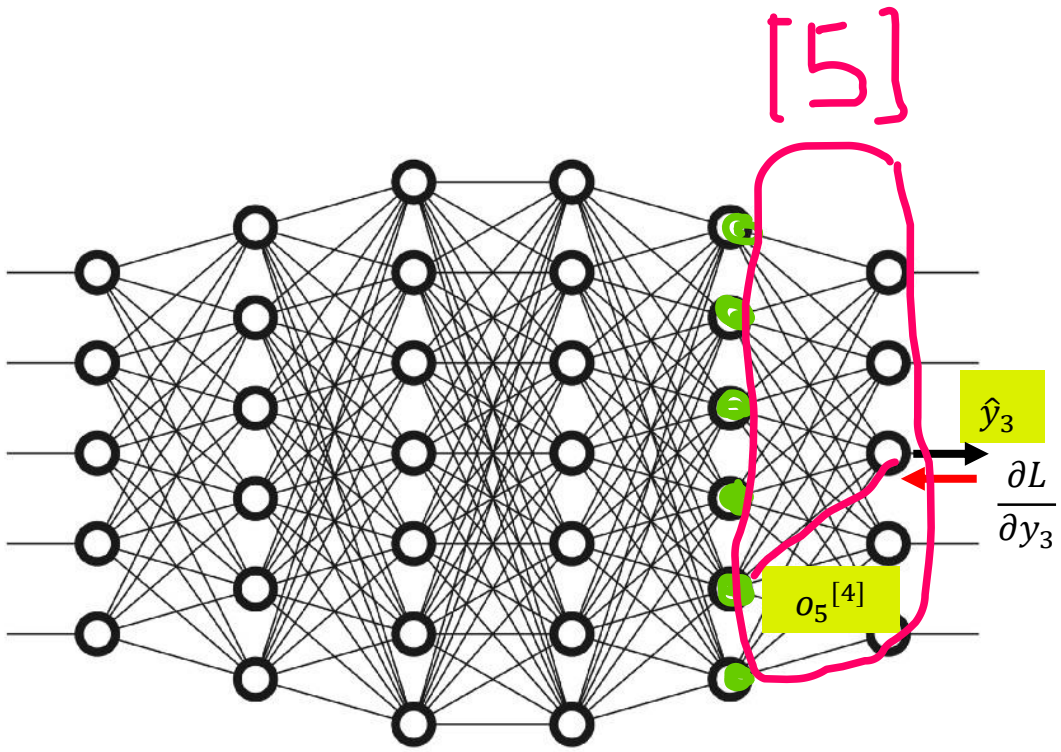
- Rework all the formulas of backpropagation up to x1 and w1 on a simple neuron, as before, when the loss function is the binary cross entropy (here, y is either 1 or 0, while $\hat{y}$ is the output of the sigmoid)
- Consider the two cases: when y=1 and y=0)

$$Loss(y, \hat{y}) = -(y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$



What happens for a more complex NN

- We have computed step by step the weight updating formula for a simple neuron
- This formula also applies to the FIRST set of weights (those closest to the output nodes in figure)
- The only difference is that rather than having x_i (the input) in the formula, we have $o_i^{[L-1]}$, i.e., the output of previous layer (the green perceptrons)
- So, the formula for updating the weights of the «closest» layer L-1 is
- $\frac{\partial L}{\partial w^{[L]}{}_i} = \boldsymbol{o_i^{[L-1]}}(y - \hat{y})\ \hat{y}(1- \hat{y})$
- And since we might have multiple output nodes and multiple examples, the complete formula is:
- $\frac{\partial L}{\partial w^{[L]}{}_{ij}} = o_i^{[L-1]}\left(y_j^h - \hat{y}_j{}^h\right)\left(1 - \hat{y}_j{}^h\right)\hat{y}_j{}^h$ where j is the j[th] output node

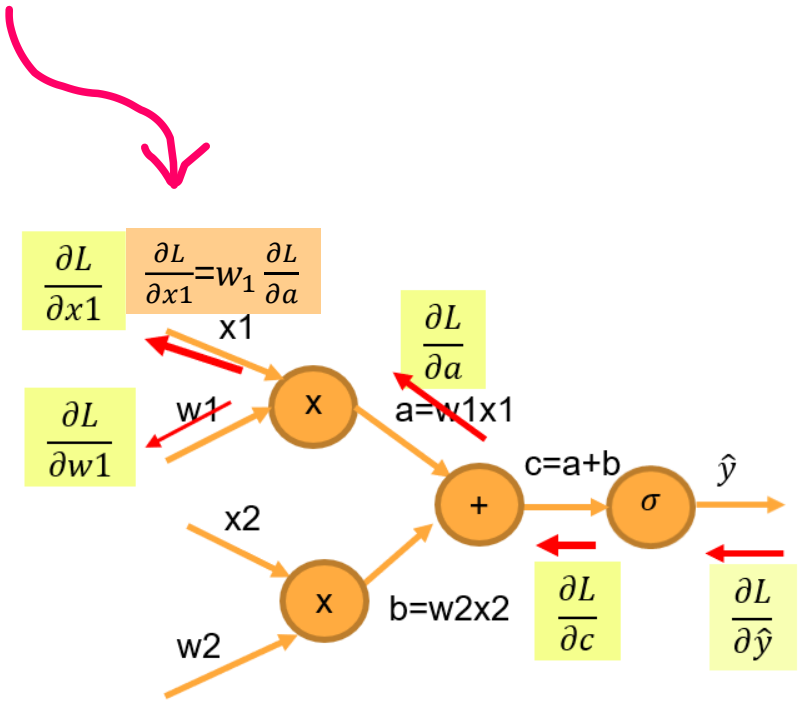


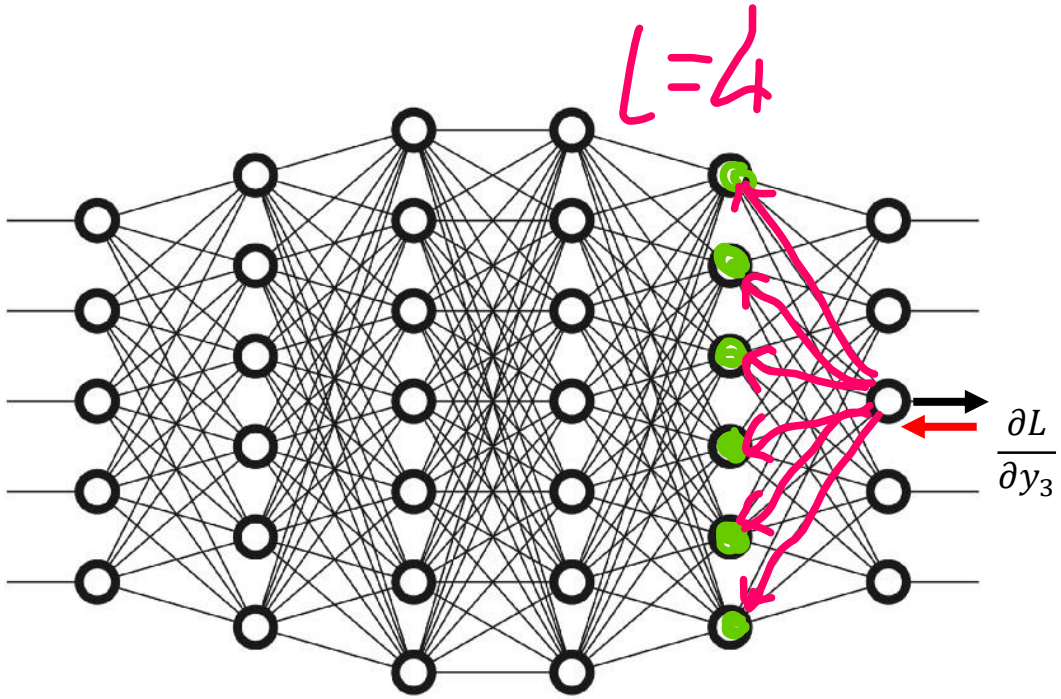
NOTE: COMMONLY, the quantity $\left(y_j^h - \hat{y}_j{}^h\right)\left(1 - \hat{y}_j{}^h\right)\hat{y}_j{}^h$ is denoted with $\delta_j{}^{[L]}$, the contribution of output node $n_j^{[L]}$ j to the total loss at step h. So we have $\boldsymbol{\frac{\partial L}{\partial w^{[L]}{}_{ij}} = o_i^{[L-1]}\delta_j^{[L]}}$

What happens for a more complex NN

- Also note that the loss that backpropagates from each node *j* of the last layer L=5 back to the nodes *i* of layer 4 is

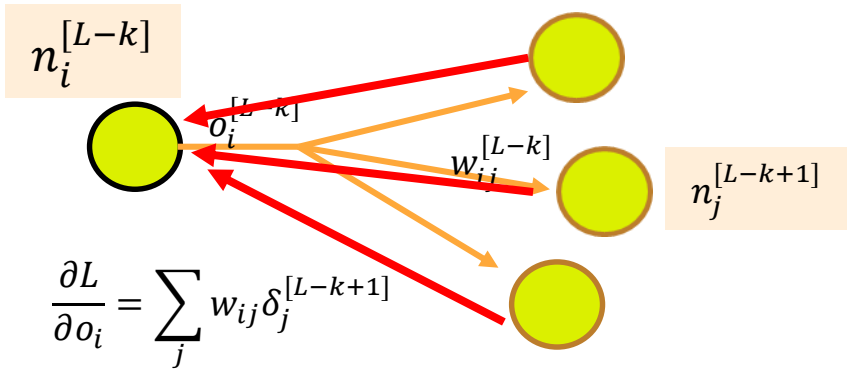
$$\frac{\partial L}{\partial o_i^{[4]}} = w_{ij}^{[5]} \delta_j^{[5]} = w_{ij}^{[5]} \left(y_j^h - \hat{y}_j{}^h\right)\left(1 - \hat{y}_j{}^h\right) \hat{y}_j{}^h$$

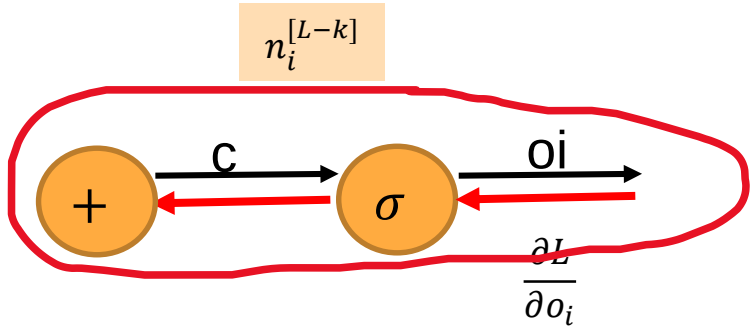




But now we need to update also the weights of the other layers

- The gradients coming from each node of layer [L-k+1] backpropagate to each node $n_i^{[L-k]}$ of the preceding layer [L-k] and **gets summed**
- Remember: $\delta_j^{[L-k+1]}$ denotes the gradient of node $n_j^{[L-k+1]}$ and $w_{ij}^{[L-k]}\delta_j^{[L-k+1]}$ is the gradient travelling back along the synaptic connection $w_{ij}^{[L-k]}$ towards node $n_i^{[L-k]}$
- The node itself further contributes to the gradient with the multiplier $(o_i)(1- o_i)$ (the derivative of the sigmoid) since the derivative of the sum is 1.

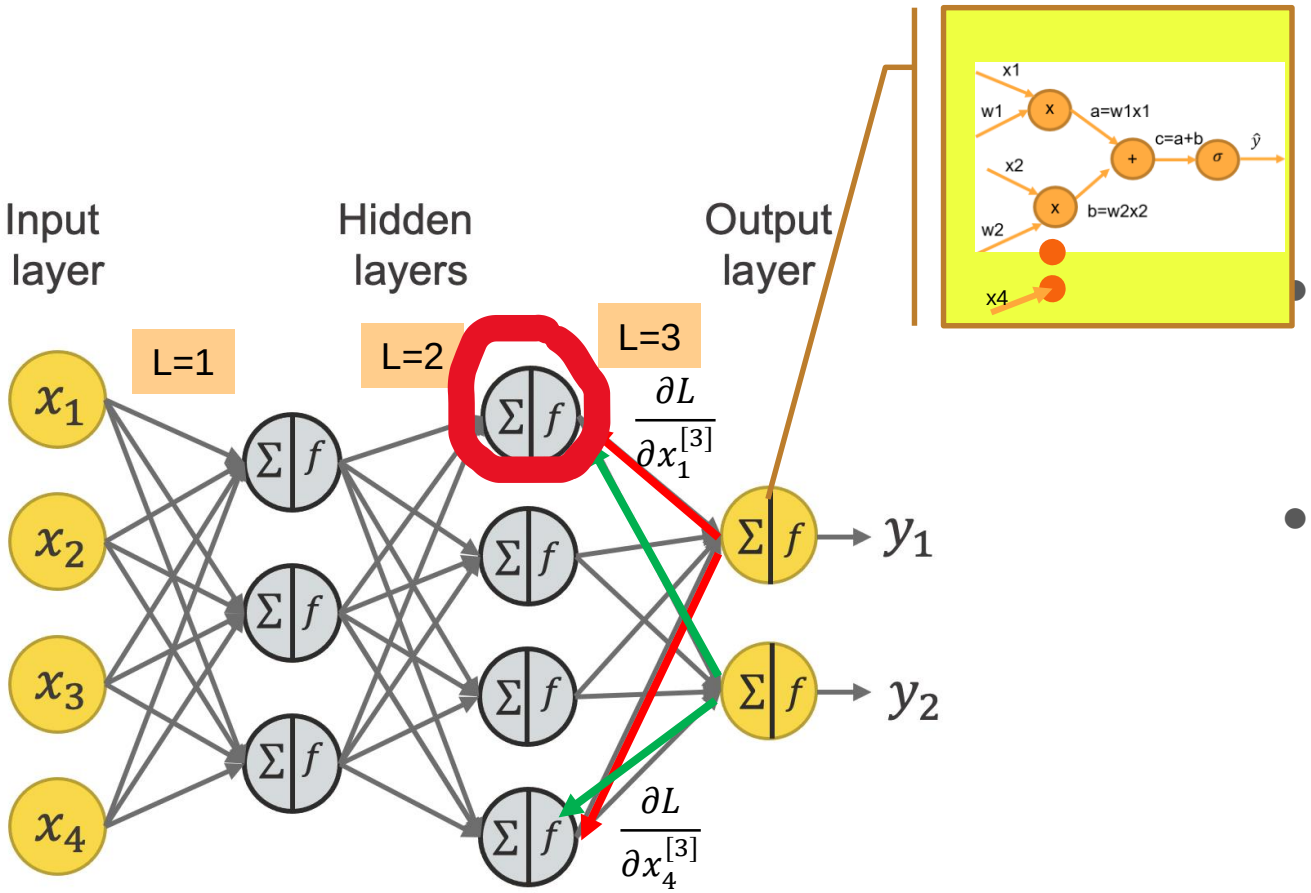




$$\delta_i^{[L-k]} = \frac{\partial L}{\partial o_i}\frac{\partial oi}{\partial c} = \sum_j w_{ij}\delta_j^{[L-K+1]} \quad (o_i)(1- o_i)$$

Pls note that here I omitted the layer superscript on weights and output..)

Example: from layer L=3 to L=2



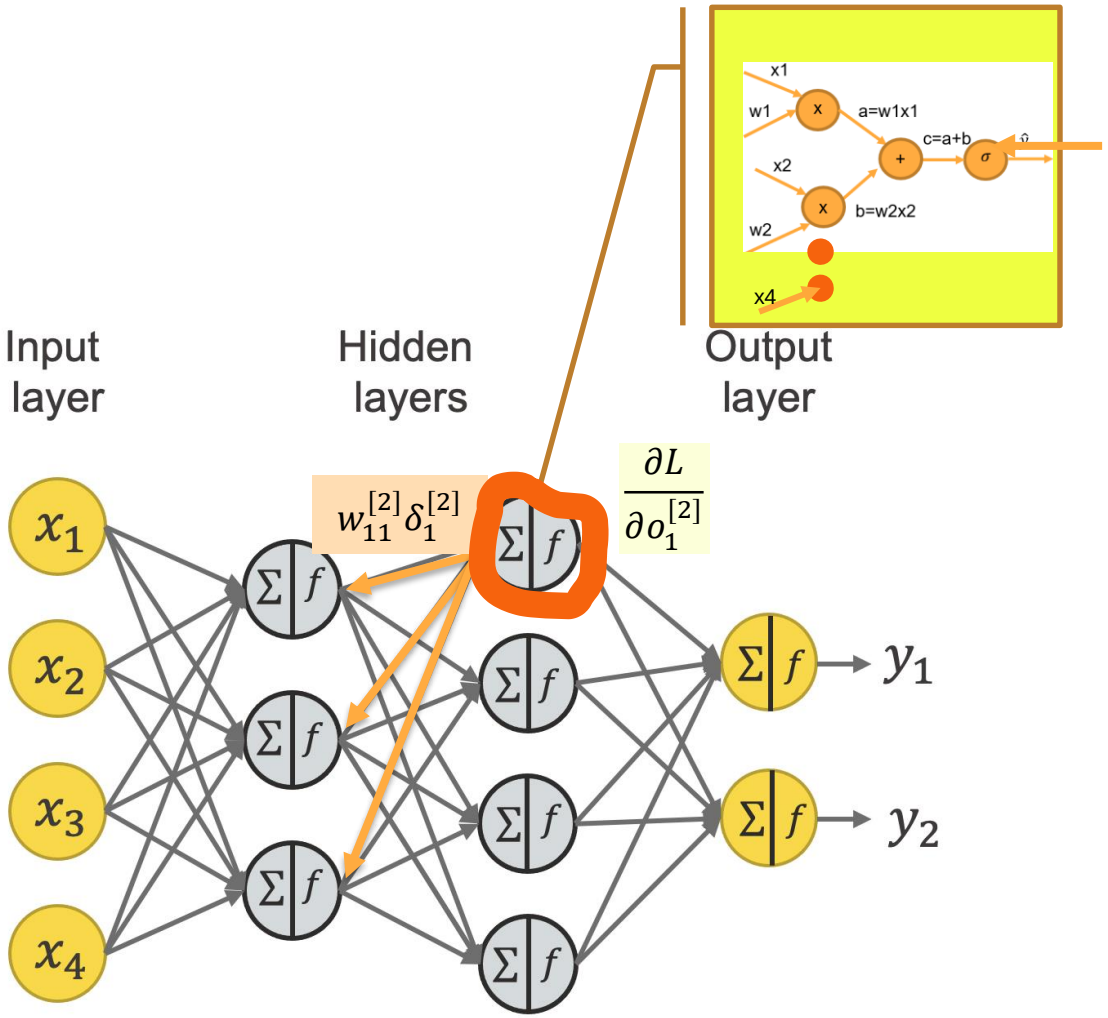
- The gradient on node $n_1^{[2]}$ backpropagates from $n_1^{[3]}$ and $n_2^{[3]}$

$$\frac{\partial L}{\partial o_1^{[2]}} = \sum_{j=1,2} \frac{\partial L}{\partial x_{1j}^{[3]}} = w_{11}^{[3]}\,\delta_1^{[3]} + w_{12}^{[3]}\delta_2^{[3]}$$

- Where $\delta_1^{[3]} = \left(y_1^h - \hat{y}_1{}^h\right)\left(1 - \hat{y}_1{}^h\right)\hat{y}_1{}^h$ and a similar formula for $\delta_2^{[3]}$

- **The gradient backpropagating on node $n_1^{[2]}$ is the sum of the gradients caused by $n_1^{[3]}$ and $n_2^{[3]}$**

And from L=2 to L=1



- The gradient on node $n_1^{[2]}$ is

$\delta_1^{[2]} = \frac{\partial L}{\partial o_1^{[2]}} o_1^{[2]}(1- o_1^{[2]}) \cdot 1 =$

$(w_{11}^{[3]}\delta_1^{[3]} + w_{12}^{[3]}\delta_2^{[3]})\ o_1^{[2]}(1- o_1^{[2]})$

(remember the derivative of the $\sum x_i$ is 1 and the derivative of the sigmoid is $\sigma(x)(1-\sigma(x))$)

- The gradient $\delta_1^{[2]}$of node $n_1^{[2]}$ further backpropagates towards $n_1^{[1]}$, $n_2^{[1]}$ and $n_3^{[1]}$
- Note that as we backpropagate towards the first layers, we keep multiplying coefficients and derivatives.. Will se later that this is a problem!

So the general formulation is

- $\delta_i^{[K]} = \sum_j w_{ij}^{[K+1]} \delta_j^{[K+1]} o_i(1 - o_i)$

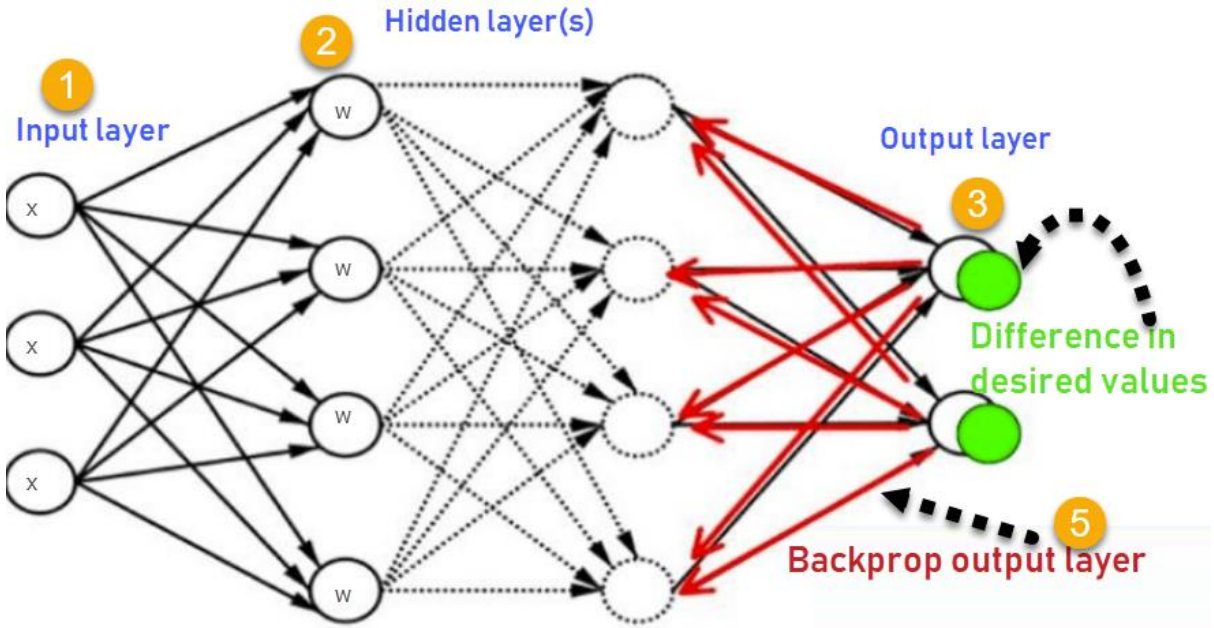
For K=1..L-1, and:

- $\delta_i^{[L]} = \left(y_i^h - \hat{y}_i{}^h\right)\left(1 - \hat{y}_i{}^h\right) \hat{y}_i{}^h$ for output nodes in last layer L

The weights updating rule is:

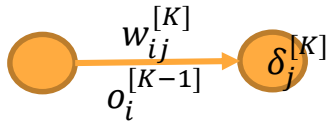
- $w_{ij}^{[K]} = w_{ij}^{[K]} - \eta \frac{\delta L}{\delta w_{ij}^{[K]}}$

Where $\frac{\delta L}{\delta w_{ij}^{[K]}} = o_i^{[k-1]} \delta_j^{[K]}$ for K=1..[L]



Recap of backpropagation algorithm

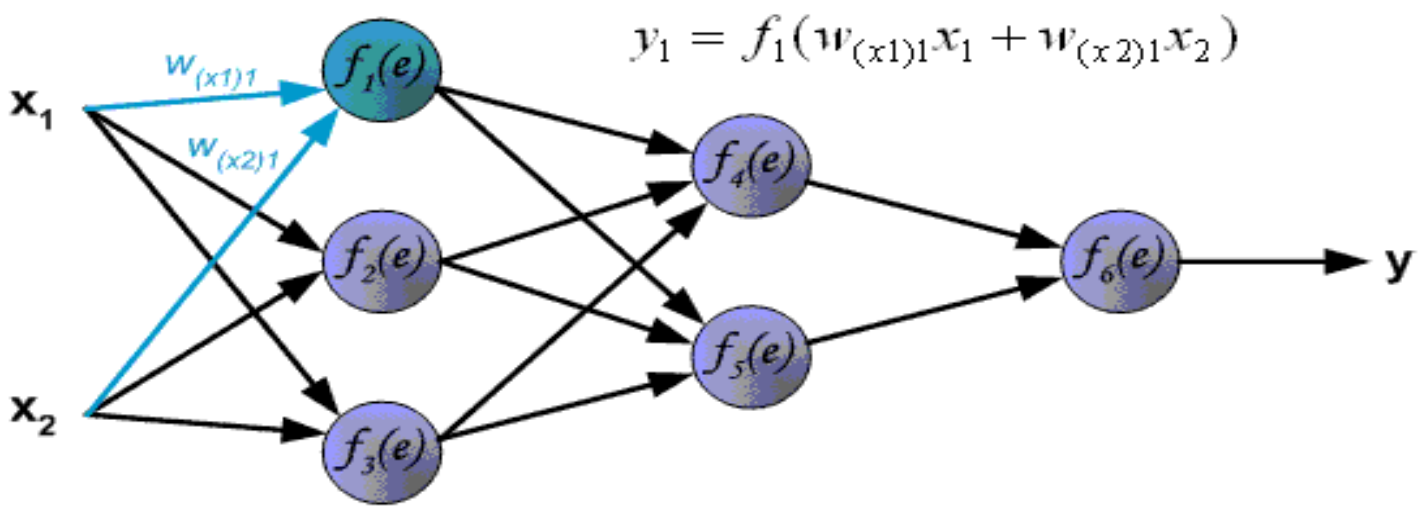
- To learn weights (and threshold), a hill-climbing approach (i.e. gradient descent) is used: $w_{ij}^{[K]} = w_{ij}^{[K]} - \eta \delta_j^{[K]} o_i^{[K-1]}$



- **Forward step**: **at each iteration and for each input x^h in D**, we compute the output vector $\boldsymbol{y}^{[L]}$ on last layer and the Loss on output nodes (this applies to any loss function, that must be derivable)
- **Backward step**: We compute the derivative of the Loss entering each ouput node, and then **update weights** with backpropagation, starting from output nodes back to input nodes using the gradient descent rule.

Weight updates w_{ij} on each layer K are proportional (η) to the contribution of the destination node $n_j^{[K]}$ to the total observed loss (which is $\delta_j^{[K]}$) and to the intensity of the signal travelling on the connection i➔j, $o_i^{[K-1]}$

FP

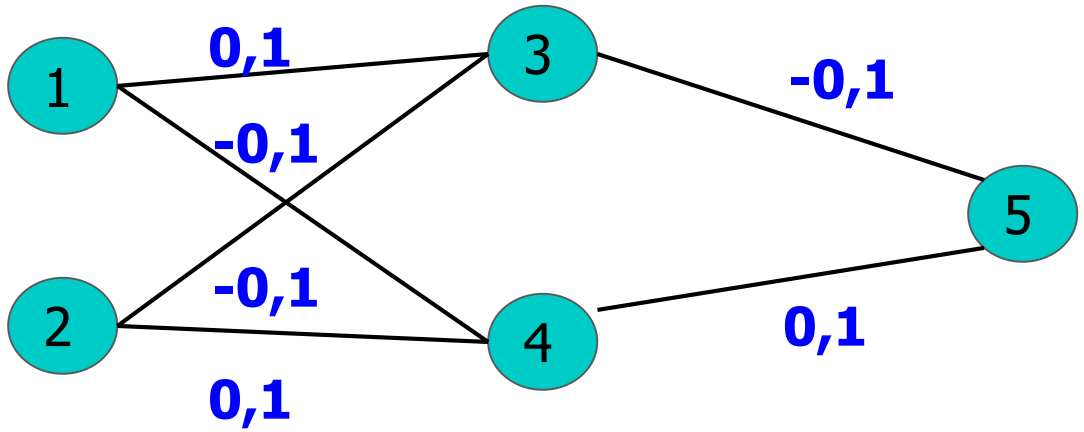


Example: 1

Training set:

1. <0.1, 0.1>, 0.1
2. <0.1, 0.9>, 0.9
3. <0.9, 0.1>, 0.9
4. <0.9, 0.9>, 0.1

- All thresholds are equal to zero



- **Error**
- **Weight**
- **Output**

Example:
Forwarding pass

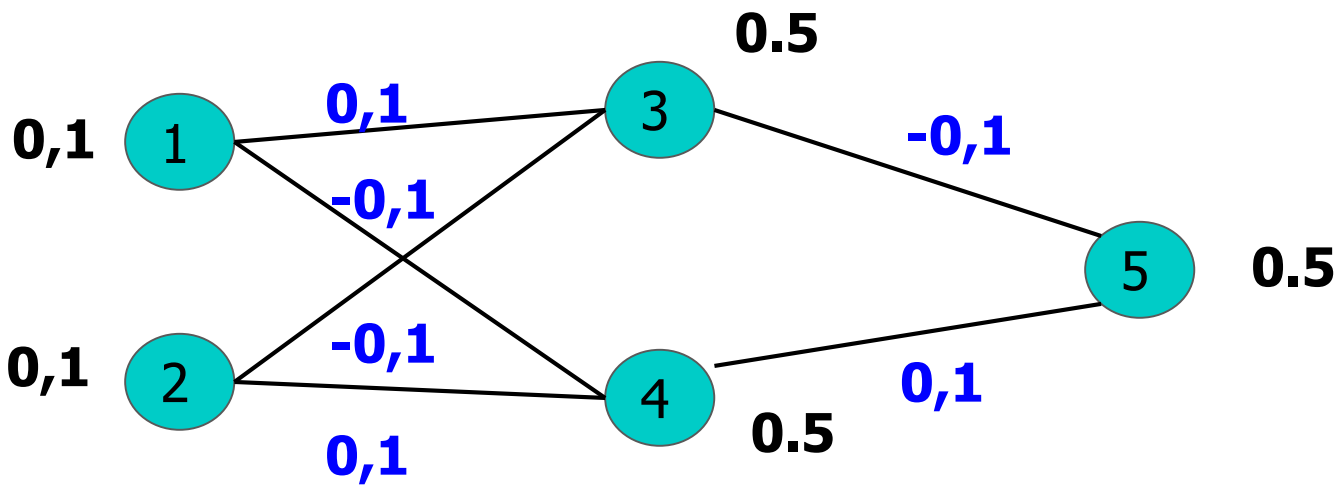
Example with the first instance:

1. <0.1, 0.1>, 0.1

$$o_3 = \varphi(o_1 w_{1,3} + o_2 w_{2,3}) = \varphi(x_1 w_{1,3} + x_2 w_{2,3}) =$$

$$\varphi(0.1 \times 0.1 + 0.1 \times -0.1) = \varphi(0) = \frac{1}{1+e^0} = 0.5$$

- **Error**
- **Weight**
- **Output**



- Let's suppose thresholds w_0 are equal to zero

Example:
Back-propagate the error

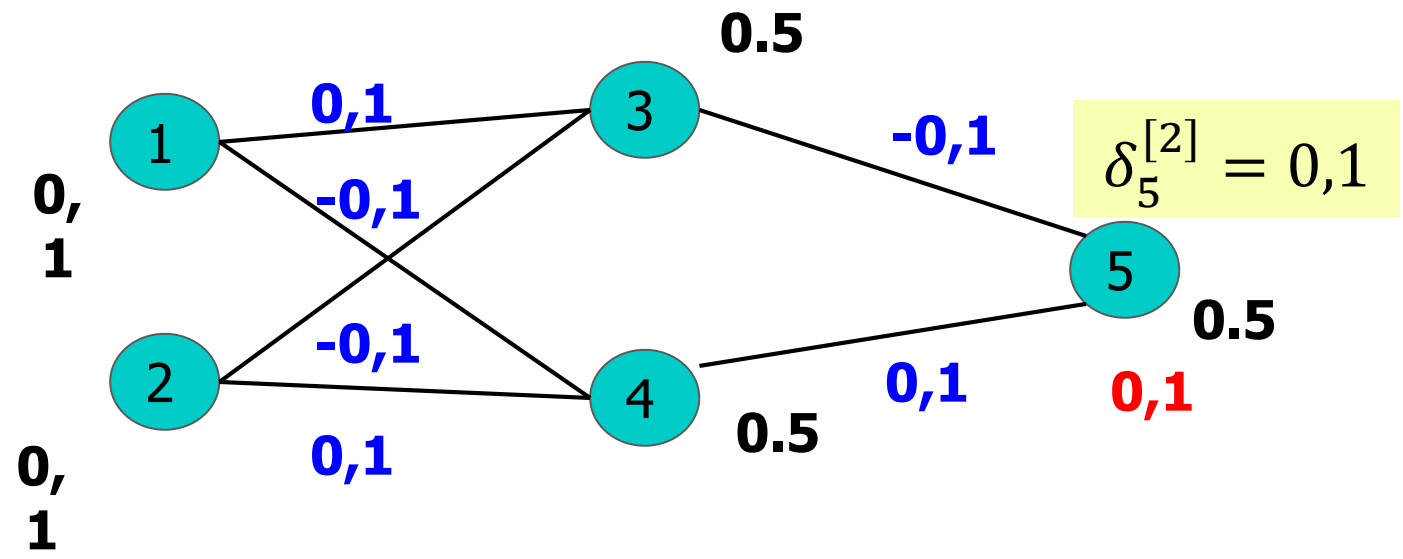
Example with the first instance:

1. <0.1, 0.1>, 0.1

$$\delta_5 = (o_5 - y_5^1)o_5(1 - o_5) = (0.5 - 0.1)0.5(1 - 0.5)$$

$$= 0.4 \times 0.5 \times 0.5 = 0.1$$

- **Error**
- **Weight**
- **Output**



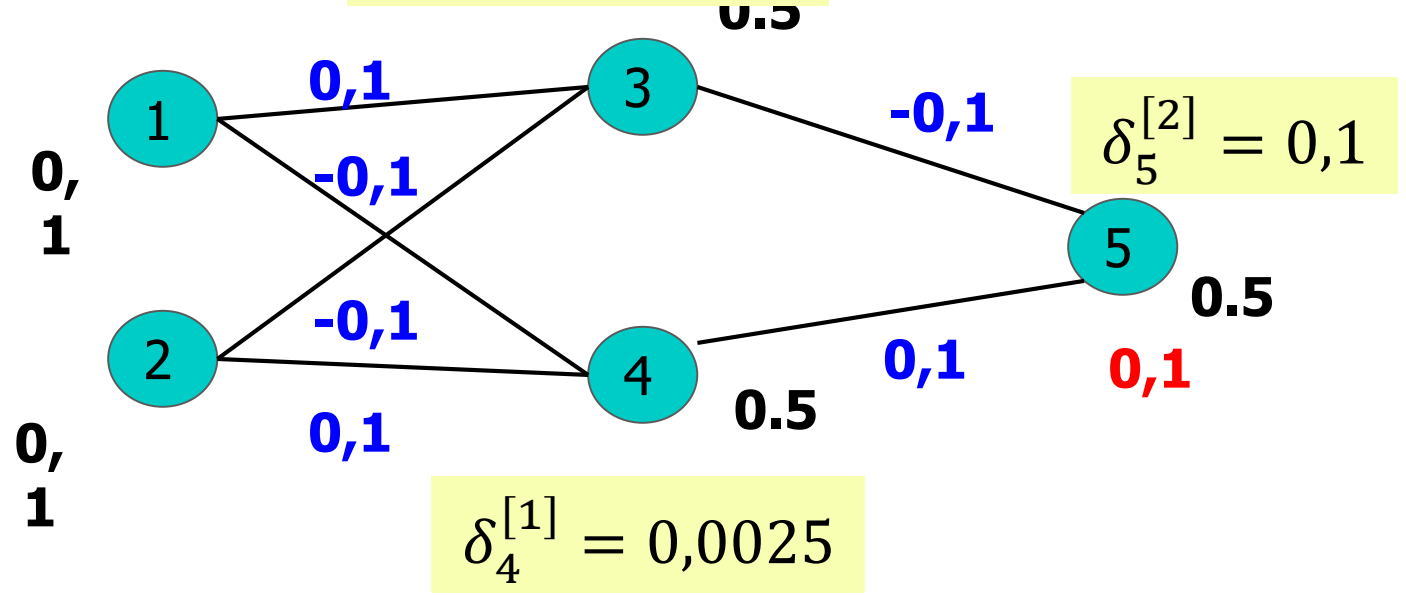
Example:
Back-propagate the error

$$\delta_4 = (0.1 \times 0.1)0.5(1 - 0.5) = 0.0025$$

$$\delta_3 = (0.1 \times -0.1)0.5(1 - 0.5) = -0.0025$$

$$\delta_3^{[1]} = -0{,}0025$$

- **Error**
- **Weight**
- **Output**



$$\delta_5^{[2]} = 0{,}1$$

$$\delta_4^{[1]} = 0{,}0025$$

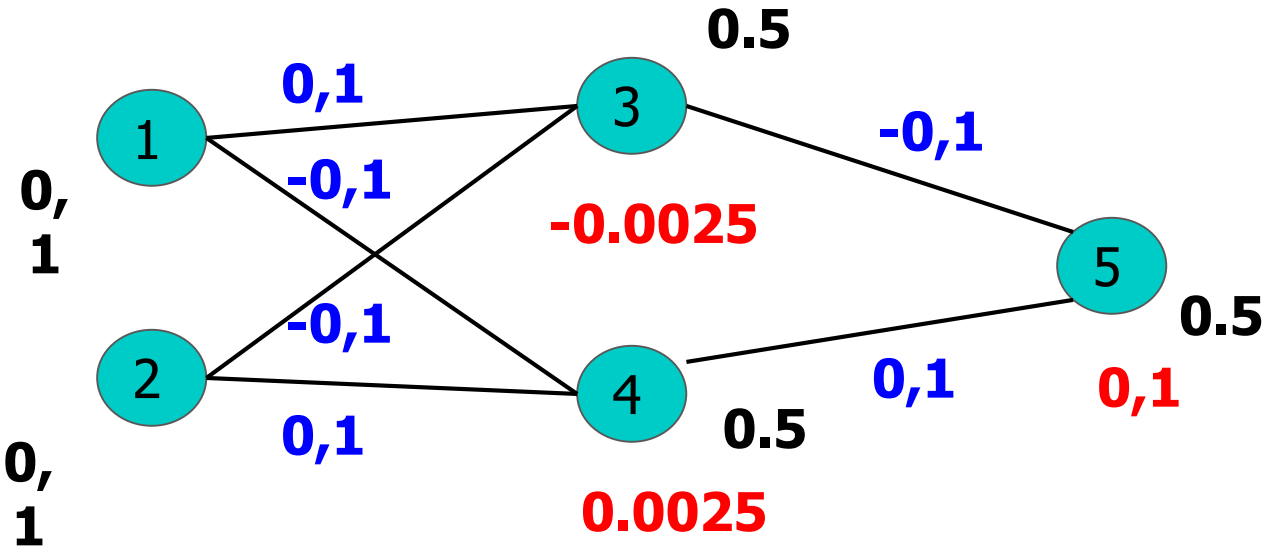
Example:
Update all weights

$$w_{ij} := w_{ij} + \Delta w_{ij} = w_{ij} - \eta \delta_j o_i$$

- $\eta = 0.2$

$$w_{45} = 0.1 - (0.2 \times 0.1 \times 0.5) = 0.09$$
$$w_{35} = -0.1 - (0.2 \times 0.1 \times 0.5) = -0.11$$
$$w_{24} = 0.1 - (0.2 \times 0.0025 \times 0.1) = 0.09995$$
$$w_{23} = -0.1 - (0.2 \times -0.0025 \times 0.1) = -0.09995$$
$$w_{14} = -0.1 - (0.2 \times 0.0025 \times 0.1) = -0.10005$$
$$w_{13} = 0.1 - (0.2 \times -0.0025 \times 0.1) = 0.10005$$

- **Error**
- **Weight**
- **Output**



Read the second example and continue until all examples have been consumed;
If Loss>0 then start new epoch

Issues with «basic» backpropagation Algorithm

- Not guaranteed that training error converges to zero, it may converge to **local minima** or oscillate indefinitely.
- Convergence speed may also depends on **initial choice** of random weights **W**
- Convergence and accuracy of results also depend on network architecture (how many hidden units? How many nodes per unit?) and the other hyperparameters
- When adding many hidden layers, everything gets worst and we experience another undesired phenomenon (vanishing gradient, see Deep learning slides)

Best weights W to begin?

- Note that if we start with all-equal weights, all neurons will learn the same thing, and NN **will not converge**
- Simple rule is to select weights uniformly at random in a range $[-\alpha + \alpha]$
- See this paper for more insight on weight initialization techniques

Avoiding local minima

- **To avoid local-minima problems*:**
  - **run several trials** starting with different random weights (*random restarts*)
  - or using **perturbations** (need to introduce another hyperparameter, the momentum).

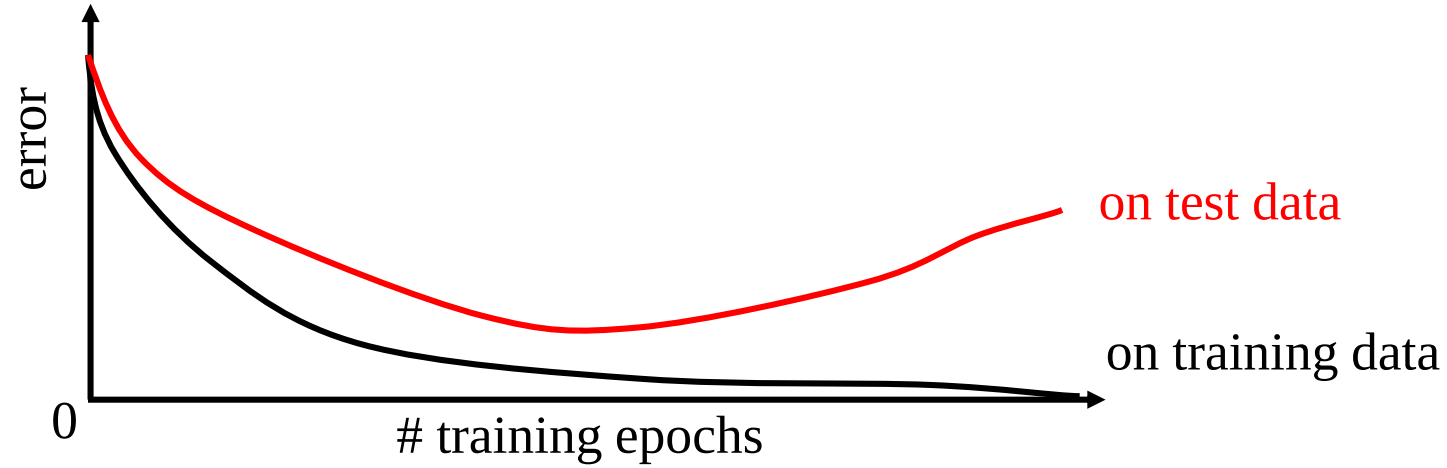
Avoiding local minima: momentum

Momentum:

- $\Delta w_{ij}^{t} = -\eta \partial Loss / \partial w_{ij} + \mu \Delta w_{ij}^{t-1}$
- Where *t* is the *t*-th update for weight w_{ij}
- Keeps memory of previous updates (t-1)
- Introduction of the **momentum** rate μ allows the **attenuation of oscillations** in the gradient descent (but yet another hyperparameter!)
- Here (for those interested) a recent paper (2021 JMLR) on the problem of local minima

NN Overfitting Prevention

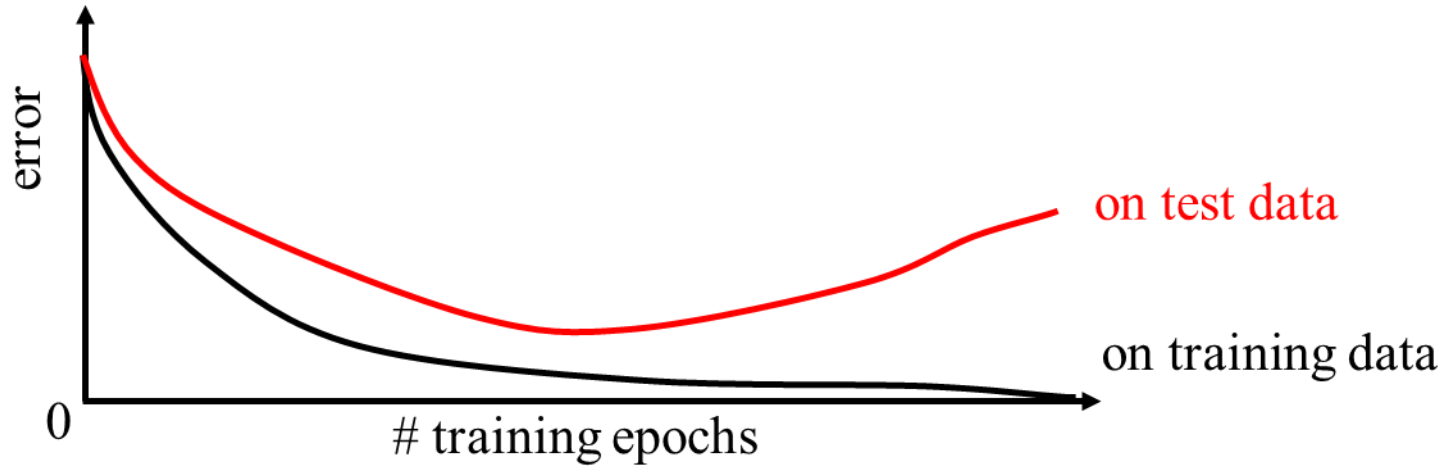
- Running too many epochs can result in overfitting



- Overfitting is a common problem in deep non-deep NN, will see some standard technique (regularization, drop out..). in the last part of the course, dedicated to the entire ML project pipeline

Determining the Best Number of Hidden Units

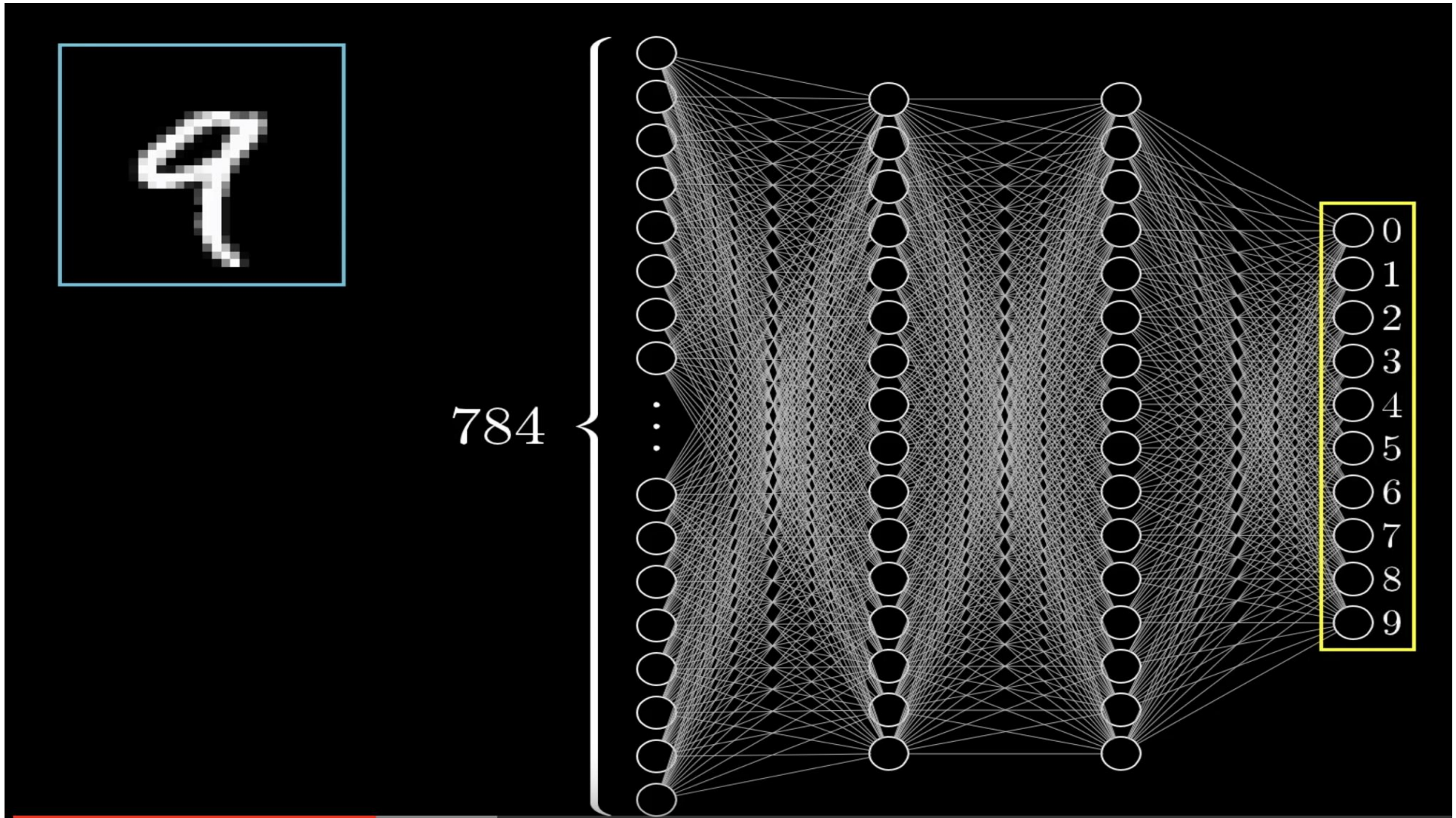
- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.



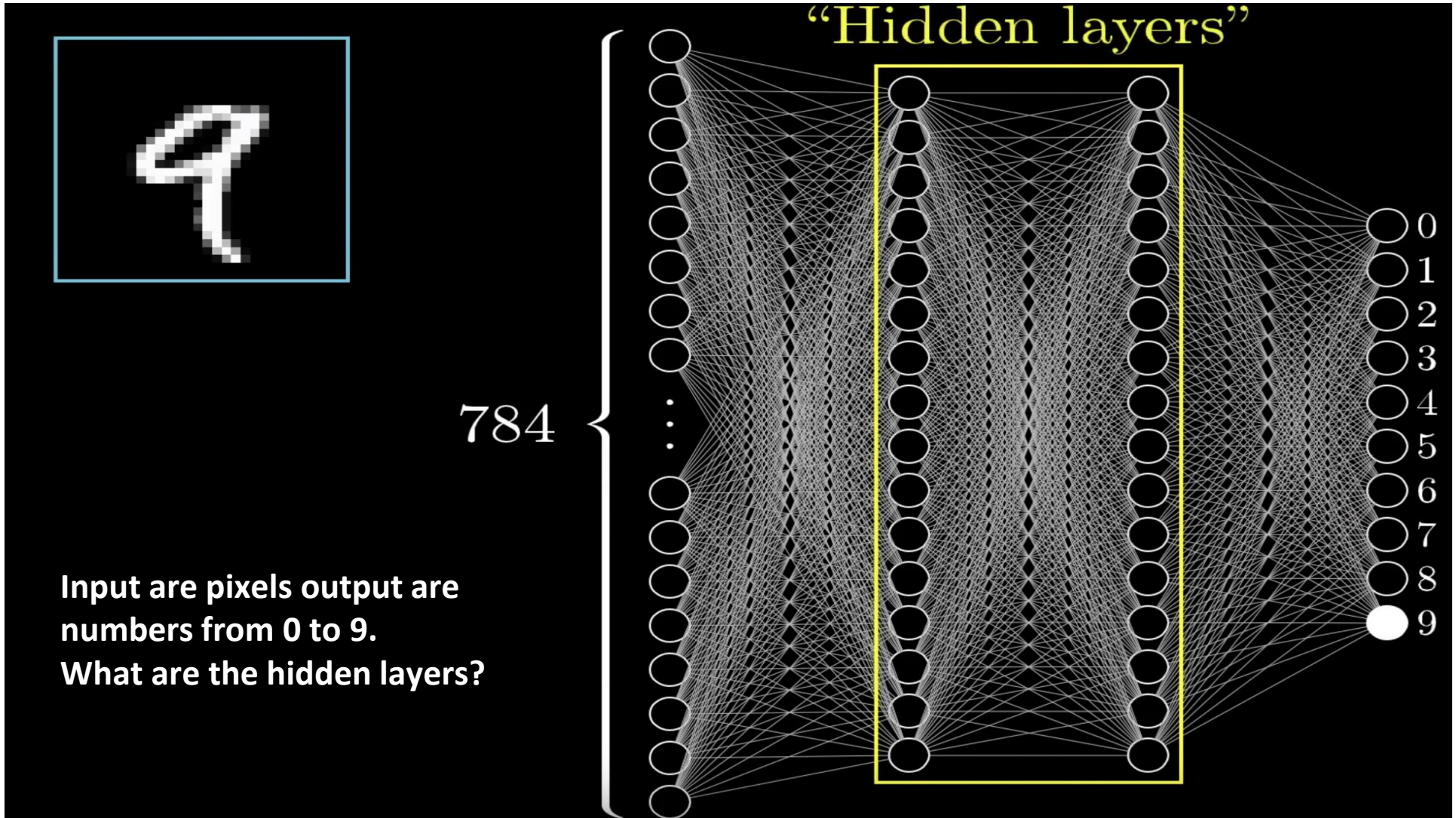
- Use internal cross-validation to empirically determine an optimal number of hidden units (or use ML algorithms to learn the best architecture, here is an example).

An intuitive explanation of hidden units in NN

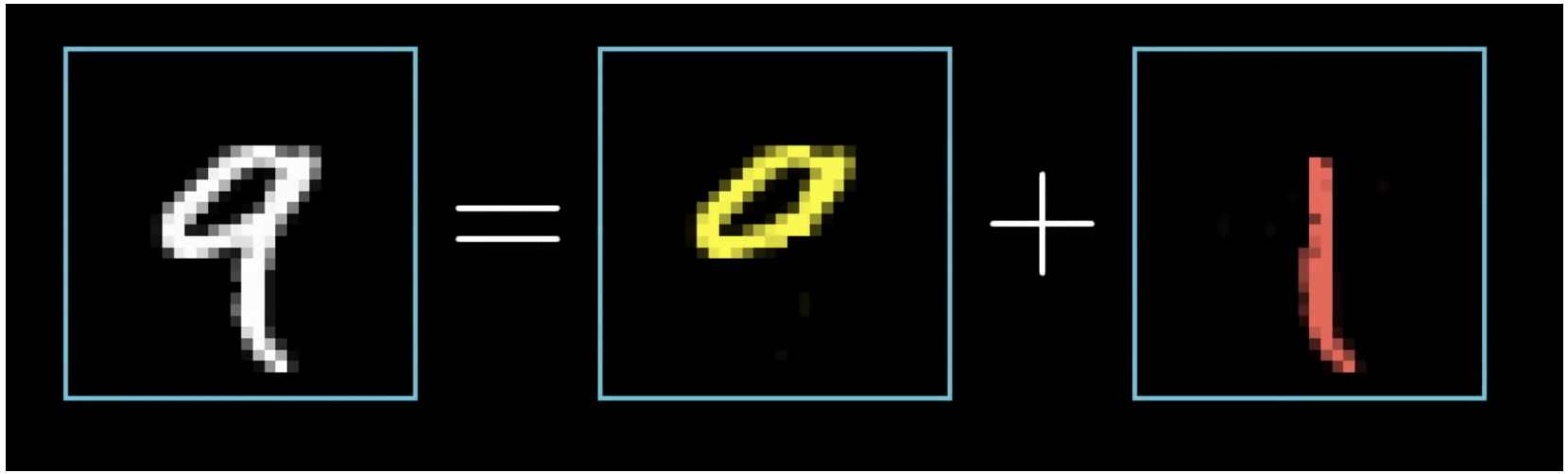
A visual intuition of how neural networks work (character recognition)



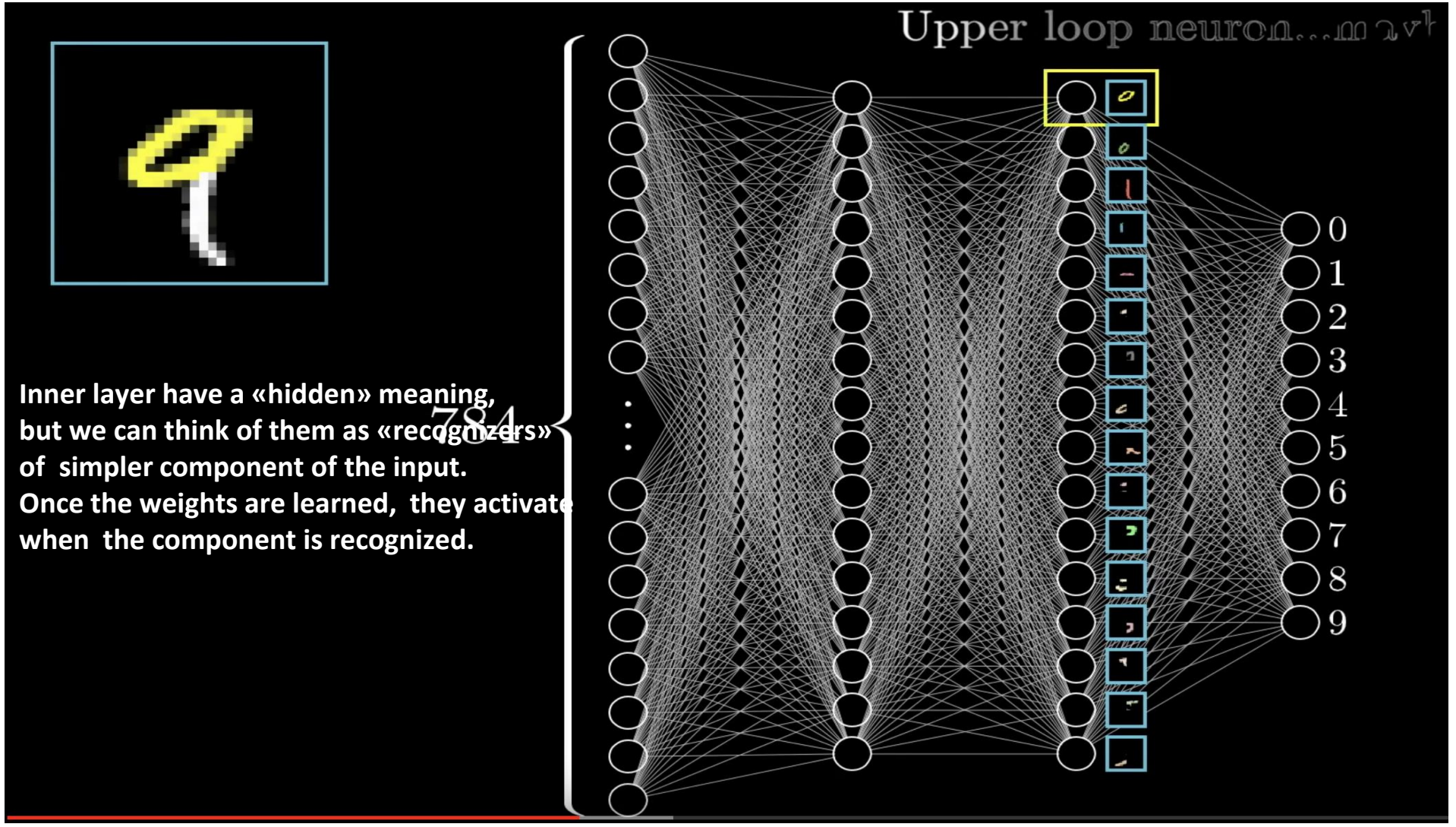
A visual intuition of neural networks (2)



A visual intuition of neural networks (3)

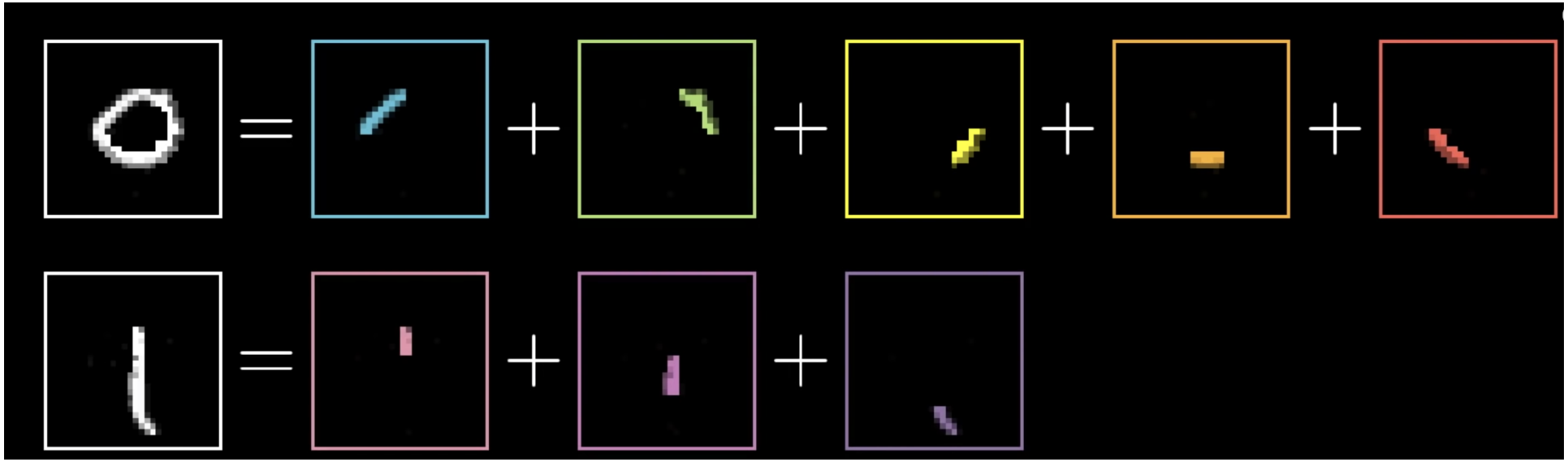


An intuition of neural networks (4)



Inner layer have a «hidden» meaning, but we can think of them as «recognizers» of simpler component of the input. Once the weights are learned, they activate when the component is recognized.

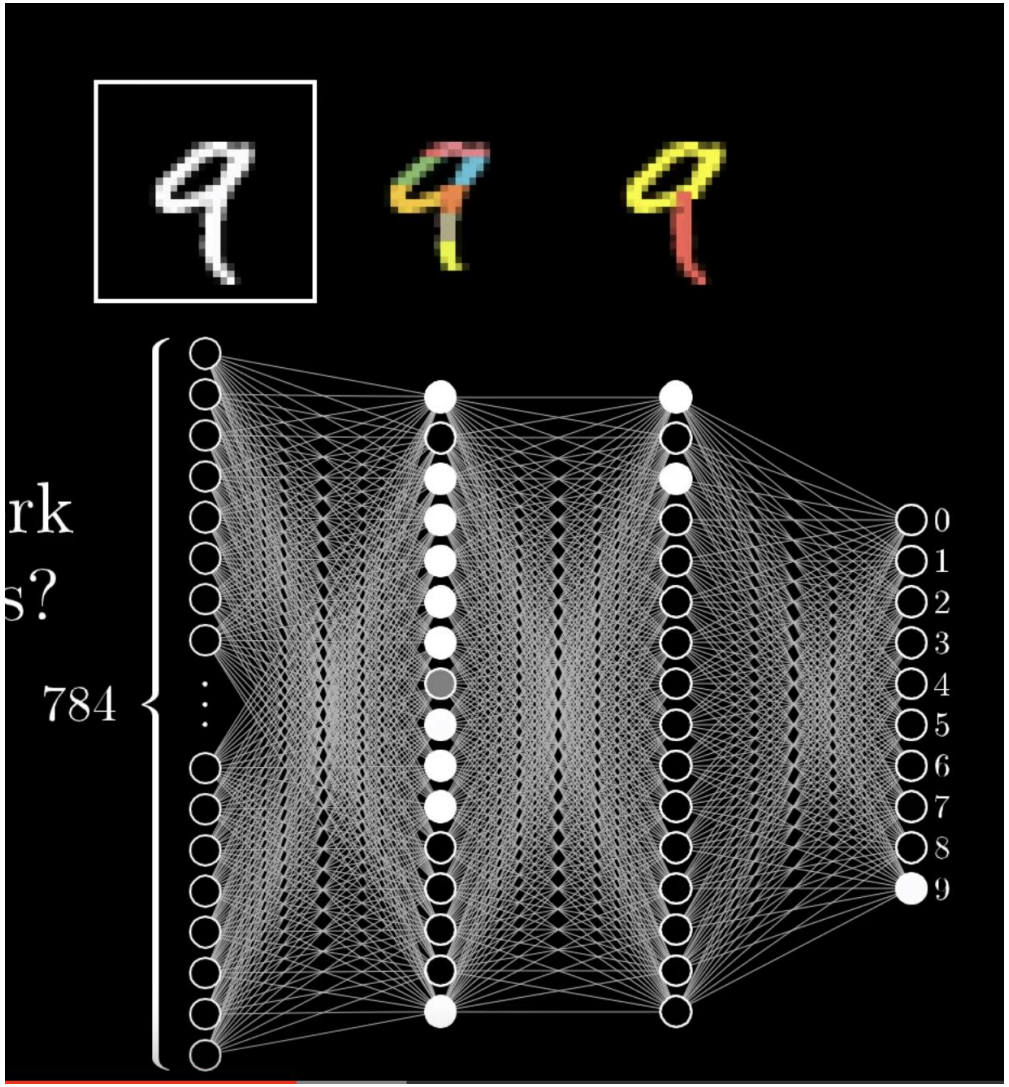
An intuition of neural networks (5)



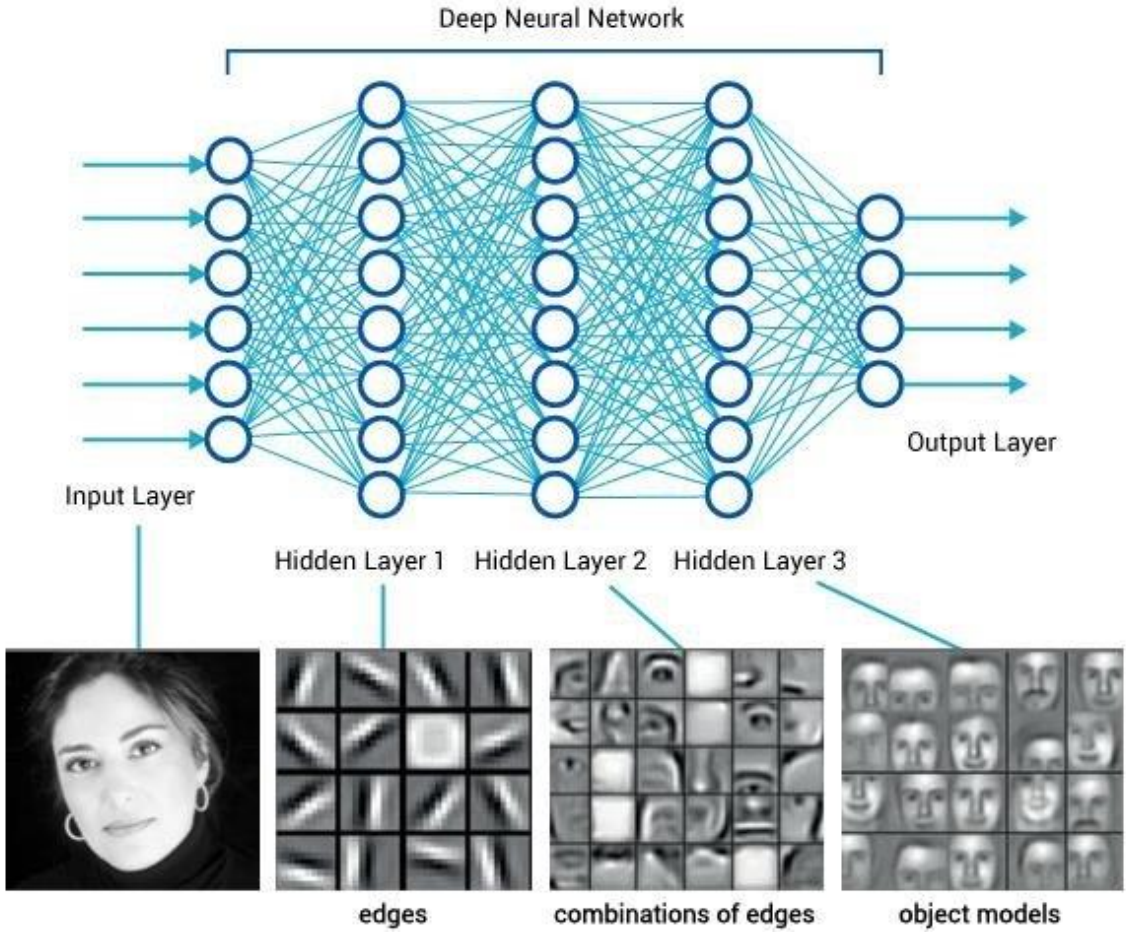
A visual intuition of neural networks (6)

Layer 2 recognizes smallest component, layer 3 more complex component.

HOWEVER this is an intuitive explanation. **We can’t actually tell the network does this..**

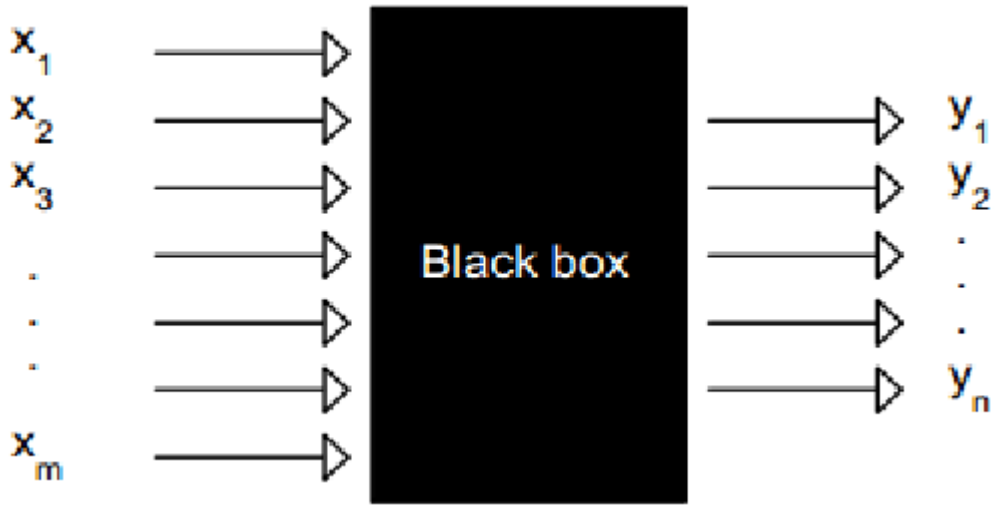


Another visual intuition: face recognition



HOWEVER this is an intuitive explanation. We can’t actually tellthe network does this..

So we have another issue with NN: Interpretability



Next lessons:
Deep Learning algorithms