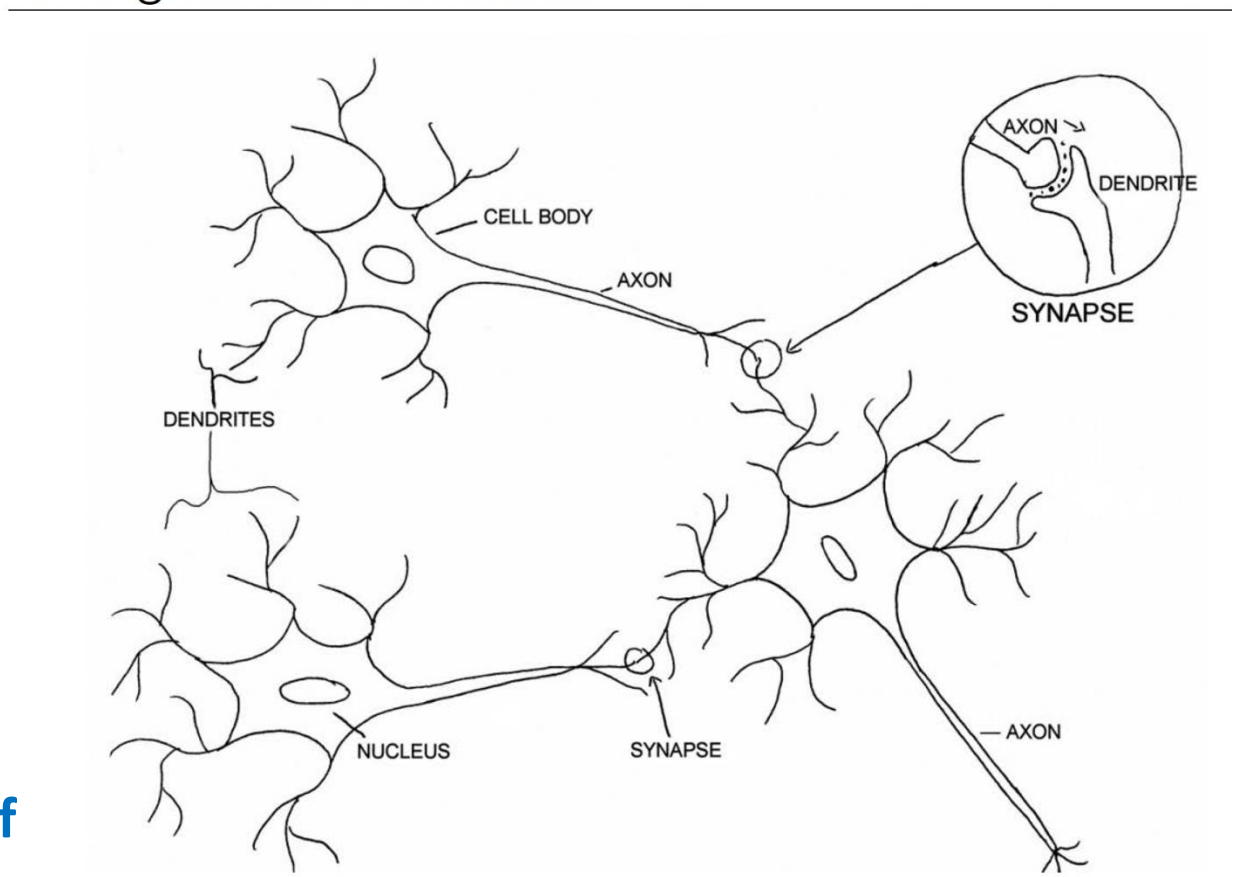
Neural Networks



UNIVERSITÀ DI PISA

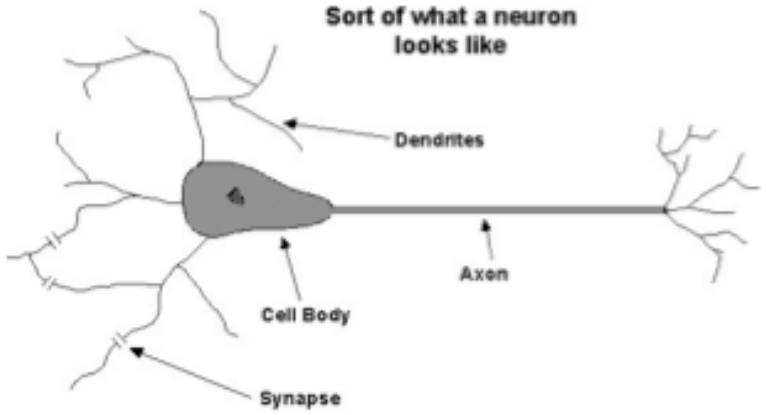
The Neuron Metaphor

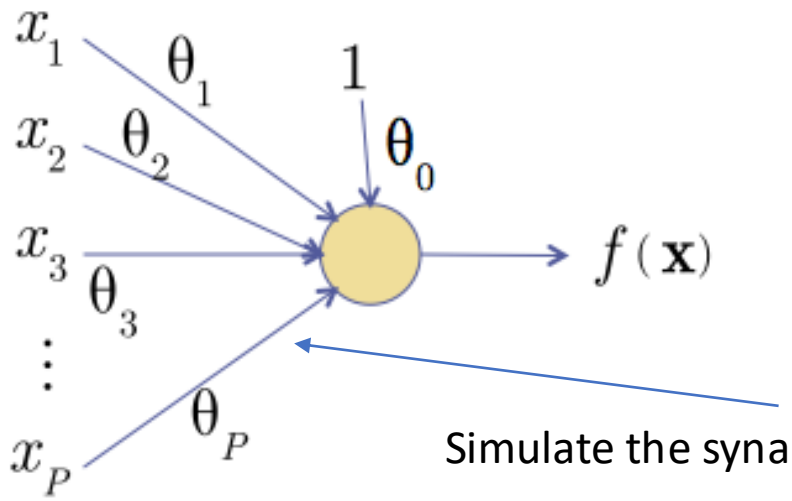
- **Artificial neural networks (ANNs)** were inspired by the structure and functioning of **biological neural systems**.
- **Neurons** receive signals mainly through **dendrites**, extensions of the cell body, and transmit nerve impulses along an **axon**.
- A **synapse** is a junction through which one neuron communicates with another, typically between an axon terminal and a dendrite.
- Learning in the brain involves changes in the **strength of synaptic connections** in response to experience and neural activity.
- Similarly, ANNs learn by adjusting the **weights of connections** between artificial neurons.



The Neuron Metaphor

- Inspired by attempts to simulate biological neural systems.
- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node

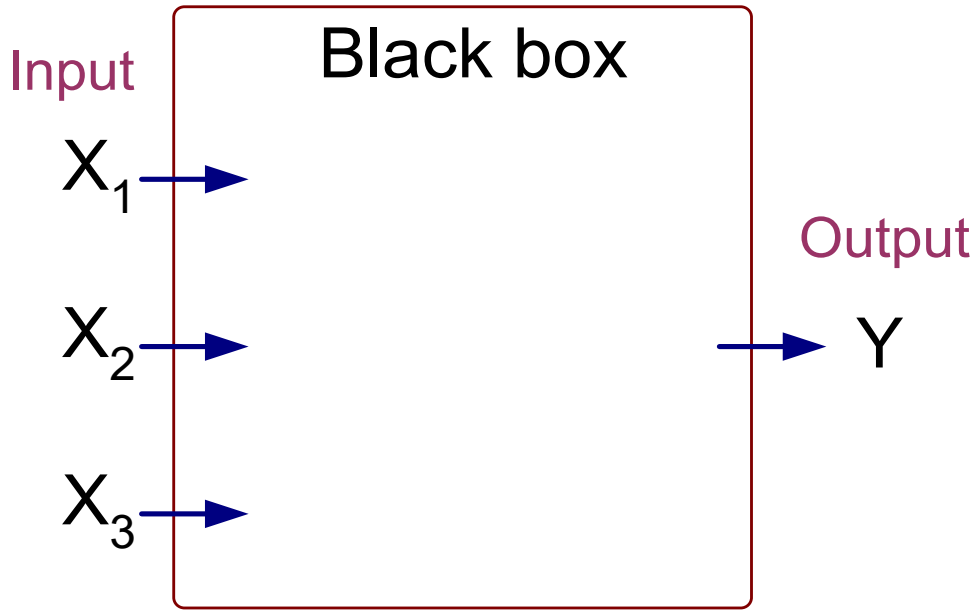




Simulate the synaptic connection

Artificial Neural Networks (ANN)

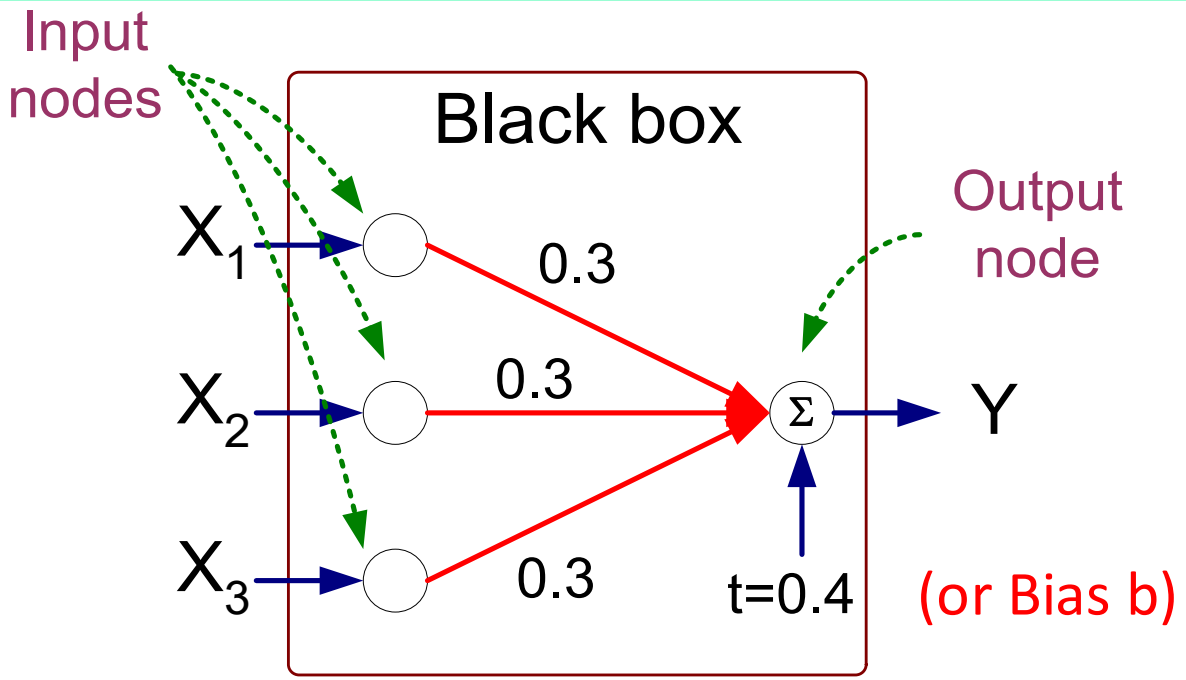
X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Output Y is 1 if at least two of the three inputs are equal to 1.

Artificial Neural Networks (ANN)

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



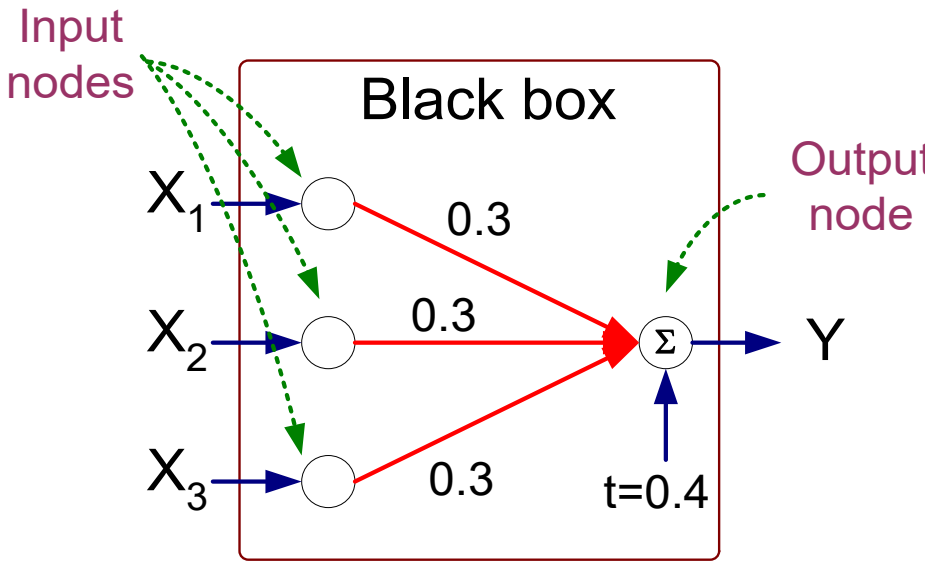
$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Artificial Neural Networks (ANN)

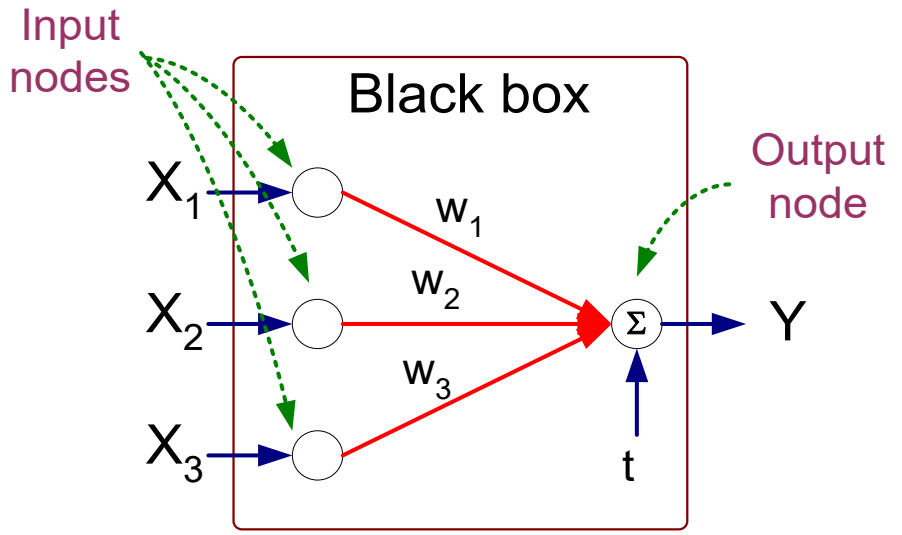
- A neural network consists of interconnected **nodes**, called **neurons** or **units**, joined by **weighted links**.
- In a **perceptron**, each input is connected to a single output neuron through a weighted link.
- The output neuron computes a **weighted sum** of the inputs and applies a **threshold function** to produce a binary prediction.
- The threshold can equivalently be represented by a **bias term** added to the weighted sum.
- Each weight controls the influence of an input, loosely resembling the **strength of a biological synaptic connection**.
- Training adjusts the **weights and bias** to learn the relationship between inputs and target outputs.

X_1	X_2	X_3	Y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1



Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links
- Output node sums up each of its input value according to the weights of its links
- Compare output node against some threshold t (also named bias b)
- Bias b => the **output** of the transformation is **biased toward being b** in the **absence of any input**.



$$Y = sign(\sum_{i=1}^{d} w_i X_i - t)$$

$$= sign(\sum_{i=0}^{d} w_i X_i)$$

$W_0 = -t$

$X_0 = 1$

Characterizing the Artificial Neuron

- Input/Output signal may be:
  - Real value
  - Unipolar {0, 1}
  - Bipolar [-1, +1]
- **Weight** (w or sigma): θ_{ij} – strength of connection from unit *j* to unit *i*
- Learning amounts to **adjusting the weights** θ_{ij} by means of an **optimization algorithm** aiming to minimize a cost function, i.e., as in biological systems training a perceptron model amounts to adapting the weights of the links until they **fit the input-output relationships of the underlying data.**

Characterizing the Artificial Neuron

- The bias b is a constant that can be written as $\theta_{ij}x_0$ with $x_0 = 1$ and $\theta_{i0} = \mathrm{b}$ such that

$$net_i = \sum_{j=0}^{n} \theta_{ij}x_j$$

- The function $f(net_i(x))$ is the unit's **activation function** for the output neuron.
- The simplest case, f is the **identity function**, and the unit's output is just its net input. This is called a ***linear unit*.**
- Otherwise we can have other functions that we see later ….

The Perceptron Classifier

Perceptron

- Single layer network
  - Contains only input and output nodes
- Activation function: f = sign(w•x)
- Applying model is straightforward

$$Y = sign(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4)$$

$$\text{where } sign(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

  - $X_1 = 1, X_2 = 0, X_3 = 1 \Rightarrow y = sign(0.2) = 1$

Learning Iterative Procedure

During the training phase the weight parameters are adjusted until the outputs of the perceptron become consistent with the true outputs of the training examples.

Initialize the weights (w_0, w_1, …, w_m) - randomly

Repeat

For each training example (x_i, y_i)

Compute $f(w^{(k)}, x_i)$

Update the weights:

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i$$

Iteration index

Learning rate

Until stopping condition is met

Perceptron Learning Rule

- Weight update formula:

$$w^{(k+1)} = w^{(k)} + \lambda\left[y_i - f(w^{(k)}, x_i)\right]x_i \;\; ; \;\; \lambda : \text{learning rate}$$

- Intuition:
  - Update weight based on error: $e = \left[y_i - f(w^{(k)}, x_i)\right]$
  - If y=f(x,w), e=0: no update needed
  - If y>f(x,w), e=2: weight must be increased so that f(x,w) will increase
  - If y<f(x,w), e=-2: weight must be decreased so that f(x,w) will decrease

The Learning Rate

Learning rate is a parameter with value between 0 and 1 used to control the amount of adjustment made in each iteration.

- If is close to 0 the new weight is mostly influenced by the value of the old weight.
- If it is close to 1, then the new weight is mostly influenced by the current adjustment.

The learning rate can be adaptive: initially moderately large and the gradually decreases in subsequent iterations.

The delta rule

- Whenever the perceptron makes a wrong classification
  ➔ change weights and threshold in ”**appropriate direction**”.

- If the **desired output is 1 and the perceptron’s output is 0** lower the threshold and adjust the weights depending on the sign and magnitude of the inputs.

- If the **desired output is 0 and the perceptron’s output is 1** increase the threshold and adjust the weights depending on the sign and magnitude of the inputs.

The delta rule

- The **delta rule** recommends to adjust the weight and the threshold values as:

$$w_i^{new} = w_i^{old} + \Delta w_i$$

$$b^{new} = b^{old} + \Delta b$$

- w_i: A weight of the perceptron
- b : The threshold value of the perceptron
- $(x_1, x_2, \dots, x_n)$: An input vector
- t: the desired output for input vector $(x_1, x_2, \dots, x_n)$
- y: the real output of the Perceptron for input vector $(x_1, x_2, \dots, x_n)$
- $\lambda > 0$: the Learning rate

The delta rule

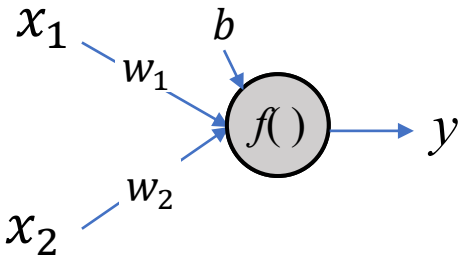
- The **delta rule** recommends to adjust the weight and the threshold values as:

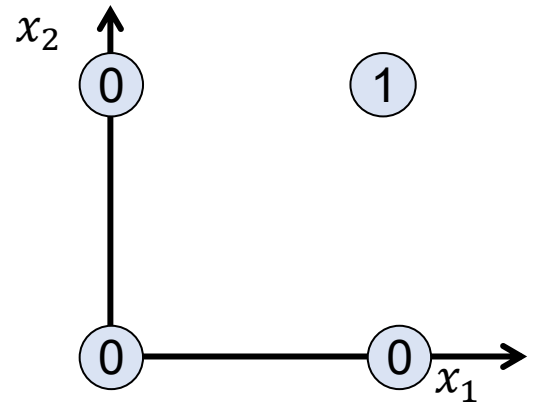
$$\Delta w_i = \begin{cases} 0 & if\ correct\ prediction \\ +\lambda x_i & if\ y = 0\ and\ t = 1 \\ -\lambda x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ correct\ prediction \\ -\lambda & if\ y = 0\ and\ t = 1 \\ +\lambda & if\ y = 1\ and\ t = 0 \end{cases}$$

Example: Learning the logical operator AND

- Training Data $\big((x_1, x_2), t\big)$: {((0,0),**0**), ((0,1),**0**), ((1,0),**0**), ((1,1), **1**)}
- Learning rate $\lambda = 1$
- t = target
- Initialization: $w_1 = w_2 = b = 0$
- $z = w_1 x_1 + w_2 x_2 - b, \qquad y = \begin{cases} 1 & z > 0, \\ 0 & z \leq 0. \end{cases}$





Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Init.								0	0	0
1	0 0	0	0	0	0	0	0	0	0	0
	0 1	0	0	0	0	0	0	0	0	0
	1 0	0	0	0	0	0	0	0	0	0
	1 1	1	0	0	1	1	-1	1	1	-1

$$\Delta w_i = \lambda(t - y)x_i,$$
$$\Delta b = -\lambda(t - y)$$

$$\Delta w_i = \begin{cases} 0 & if\ correct\ prediction \\ +\lambda x_i & if\ y = 0\ and\ t = 1 \\ -\lambda x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ correct\ prediction \\ -\lambda & if\ y = 0\ and\ t = 1 \\ +\lambda & if\ y = 1\ and\ t = 0 \end{cases}$$

Example: Learning the logical operator AND

$$z = w_1 x_1 + w_2 x_2 - b, \qquad y = \begin{cases} 1 & z > 0, \\ 0 & z \leq 0. \end{cases}$$

$$\Delta w_i = \lambda(t - y)x_i,$$
$$\Delta b = -\lambda(t - y)$$

$$\Delta w_i = \begin{cases} 0 & if\ correct\ prediction \\ +\lambda x_i & if\ y = 0\ and\ t = 1 \\ -\lambda x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ correct\ prediction \\ -\lambda & if\ y = 0\ and\ t = 1 \\ +\lambda & if\ y = 1\ and\ t = 0 \end{cases}$$

$$w_i^{new} = w_i^{old} + \Delta w_i$$

$$b^{new} = b^{old} + \Delta b$$

Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
								1	1	-1
2	0 0	0	1	1	0	0	1	1	1	0
	0 1	0	1	1	0	-1	1	1	0	1
	1 0	0	0	0	0	0	0	1	0	1
	1 1	1	0	0	1	1	-1	2	1	0

Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
								2	1	0
3	0 0	0	0	0	0	0	0	2	1	0
	0 1	0	1	1	0	-1	1	2	0	1
	1 0	0	1	1	-1	0	1	1	0	2
	1 1	1	-1	0	1	1	-1	2	1	1

Example: Learning the logical operator AND

$$z = w_1 x_1 + w_2 x_2 - b, \qquad y = \begin{cases} 1 & z > 0, \\ 0 & z \leq 0. \end{cases}$$

$$\Delta w_i = \lambda(t - y)x_i,$$
$$\Delta b = -\lambda(t - y)$$

$$\Delta w_i = \begin{cases} 0 & if\ correct\ prediction \\ +\lambda x_i & if\ y = 0\ and\ t = 1 \\ -\lambda x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ correct\ prediction \\ -\lambda & if\ y = 0\ and\ t = 1 \\ +\lambda & if\ y = 1\ and\ t = 0 \end{cases}$$

$$w_i^{new} = w_i^{old} + \Delta w_i$$

$$b^{new} = b^{old} + \Delta b$$

Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
								2	1	1
4	0 0	0	-1	0	0	0	0	2	1	1
	0 1	0	0	0	0	0	0	2	1	1
	1 0	0	1	1	-1	0	1	1	1	2
	1 1	1	0	0	1	1	-1	2	2	1

Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
								2	2	1
5	0 0	0	-1	0	0	0	0	2	2	1
	0 1	0	1	1	0	-1	1	2	1	2
	1 0	0	0	0	0	0	0	2	1	2
	1 1	1	1	1	0	0	0	2	1	2

Example: Learning the logical operator AND

$$z = w_1 x_1 + w_2 x_2 - b, \qquad y = \begin{cases} 1 & z > 0, \\ 0 & z \leq 0. \end{cases}$$

$$\Delta w_i = \lambda(t - y)x_i,$$
$$\Delta b = -\lambda(t - y)$$

$$\Delta w_i = \begin{cases} 0 & if\ correct\ prediction \\ +\lambda x_i & if\ y = 0\ and\ t = 1 \\ -\lambda x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ correct\ prediction \\ -\lambda & if\ y = 0\ and\ t = 1 \\ +\lambda & if\ y = 1\ and\ t = 0 \end{cases}$$

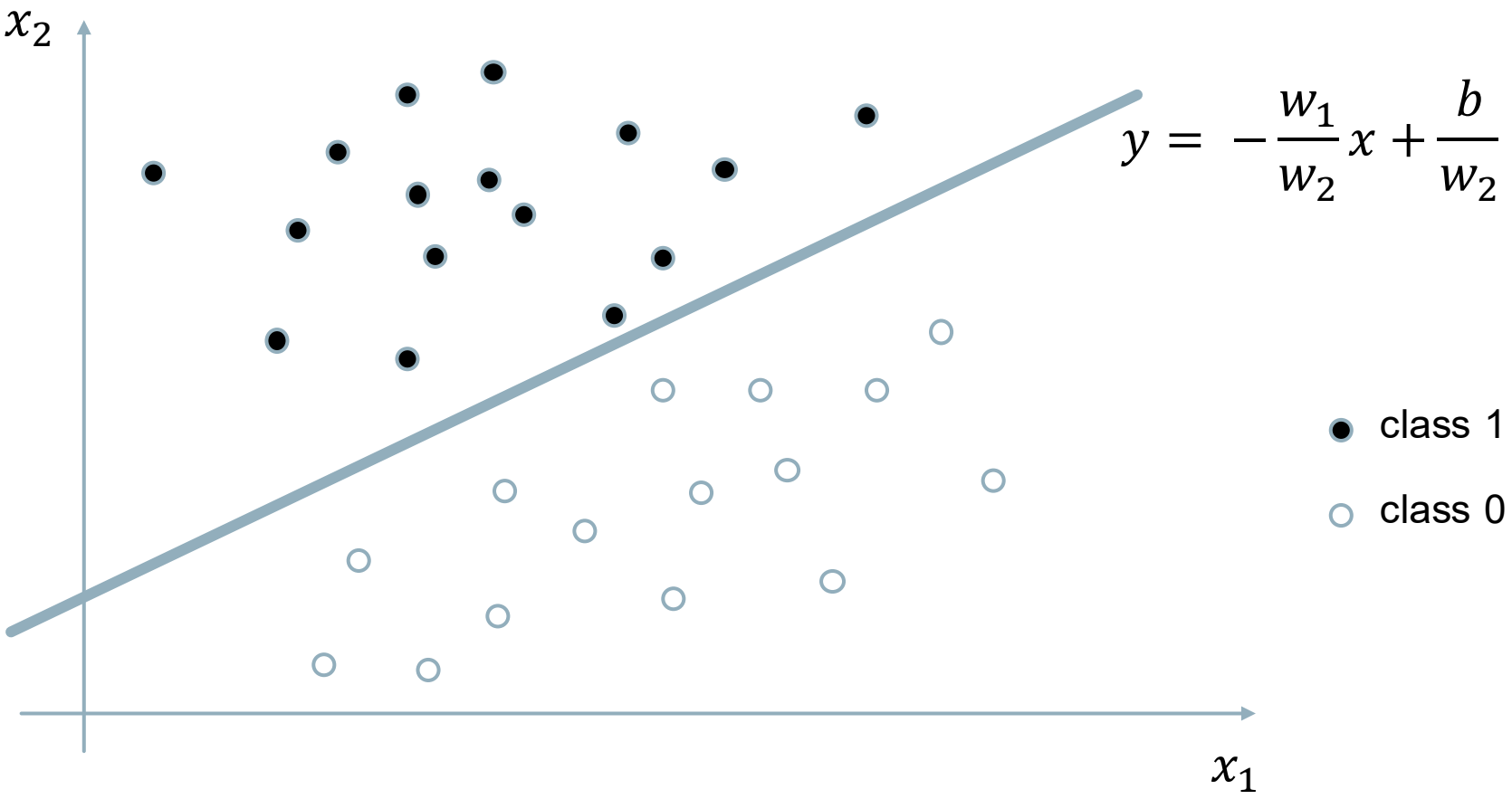
$$w_i^{new} = w_i^{old} + \Delta w_i$$

$$b^{new} = b^{old} + \Delta b$$

Epoch	x	t	z	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
								2	1	2
6	0 0	0	-2	0	0	0	0	2	1	2
	0 1	0	-1	0	0	0	0	2	1	2
	1 0	0	0	0	0	0	0	2	1	2
	1 1	1	1	1	0	0	0	2	1	2

What classification problems can a perceptron solve?

Linear Separability



The parameters w_1, w_2, define the line. All input patterns above this line are assigned to class 1, all input patterns below the line to class 0.

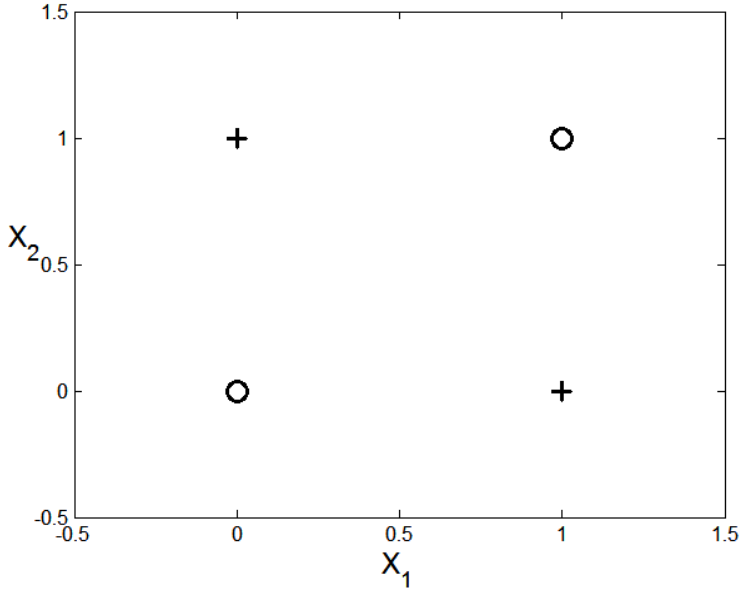
Non-linearly Separable Data

- Since *f(w,x)* is a linear combination of input variables, decision boundary is linear.
- For nonlinearly separable problems, **the perceptron fails** because no linear hyperplane can separate the data perfectly.
- An example of nonlinearly separable data is the XOR function.

XOR Data

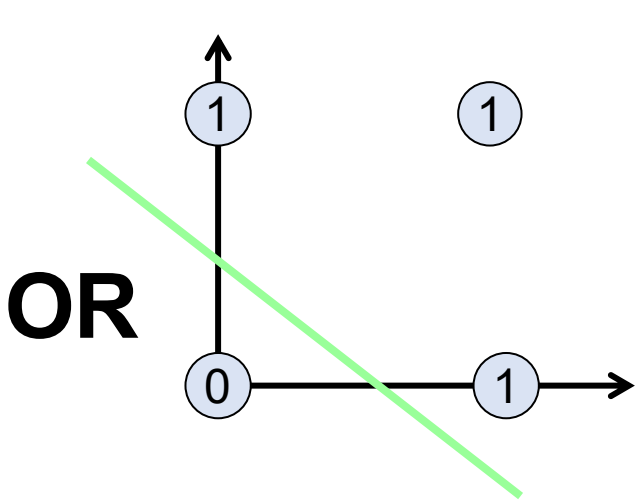
x_1	x_2	y
0	0	-1
1	0	1
0	1	1
1	1	-1

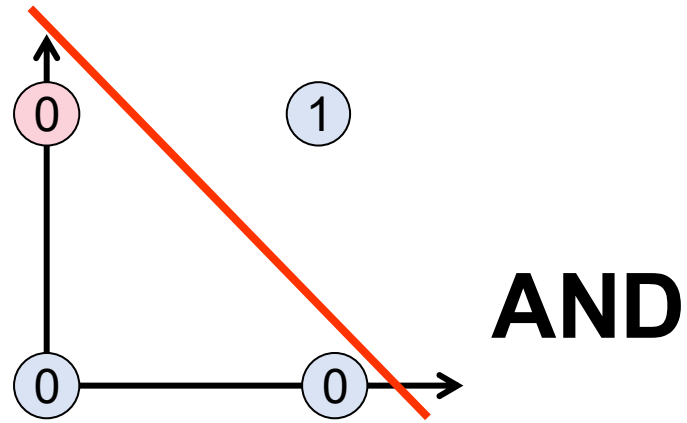
$$y = x_1 \oplus x_2$$

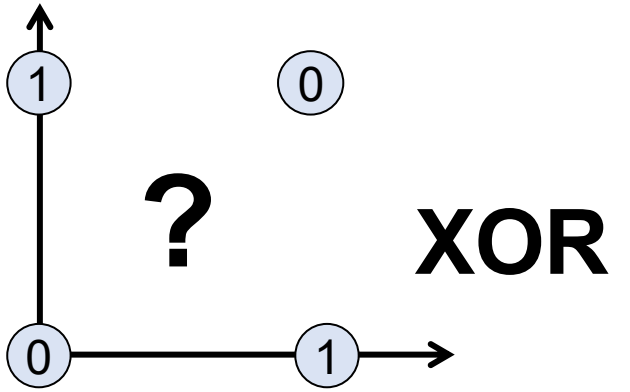


What can a single Perceptron do?

Example: The exclusive OR (XOR) defines a classification task which is not linearly separable.

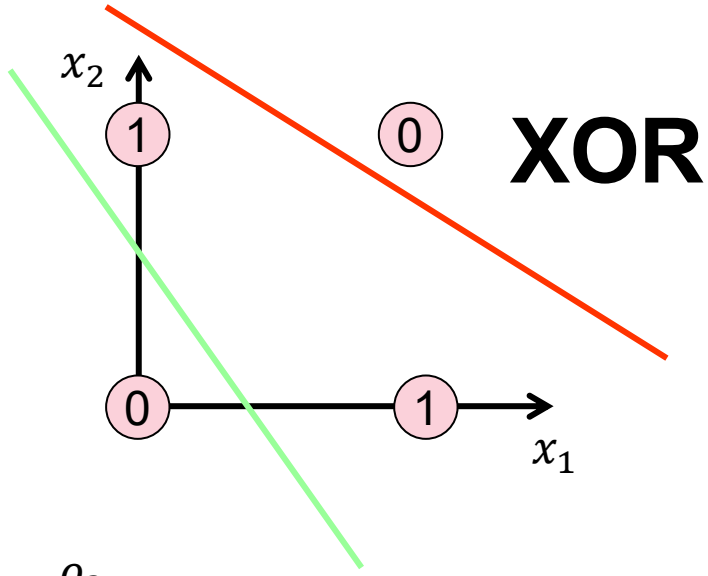


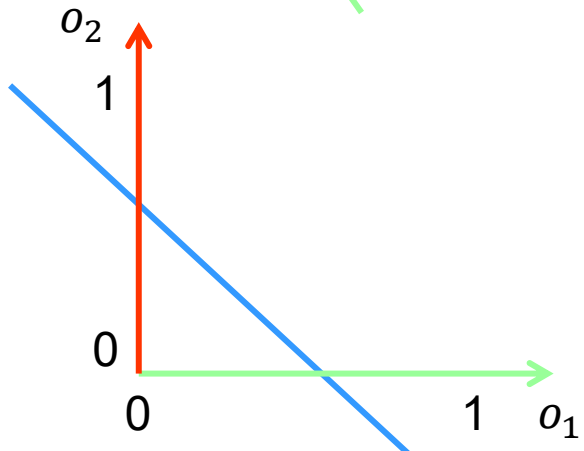


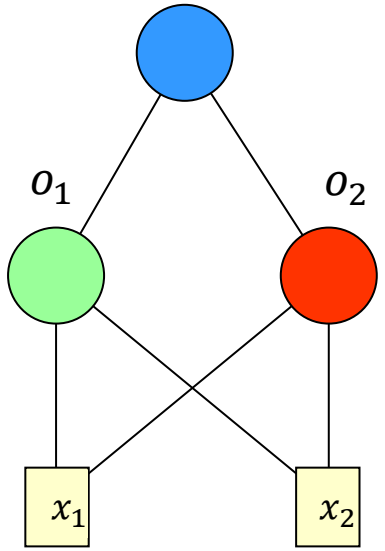


Adding one more layer

The exclusive OR (XOR) can be solved adding one more layer



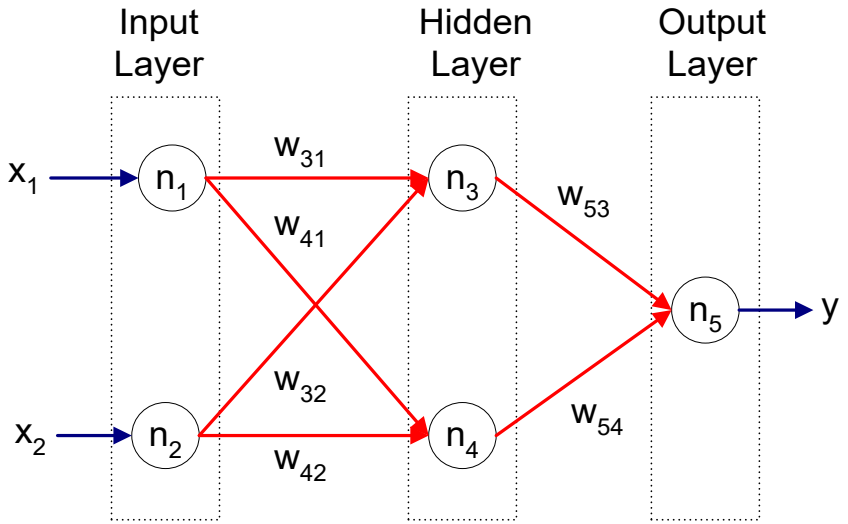


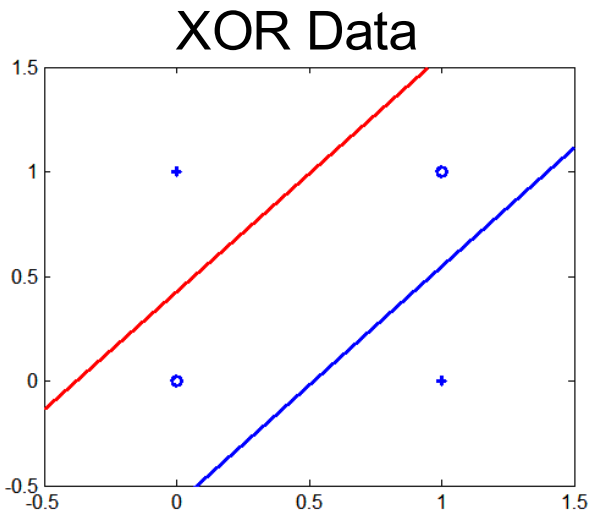


Multilayer Neural Network

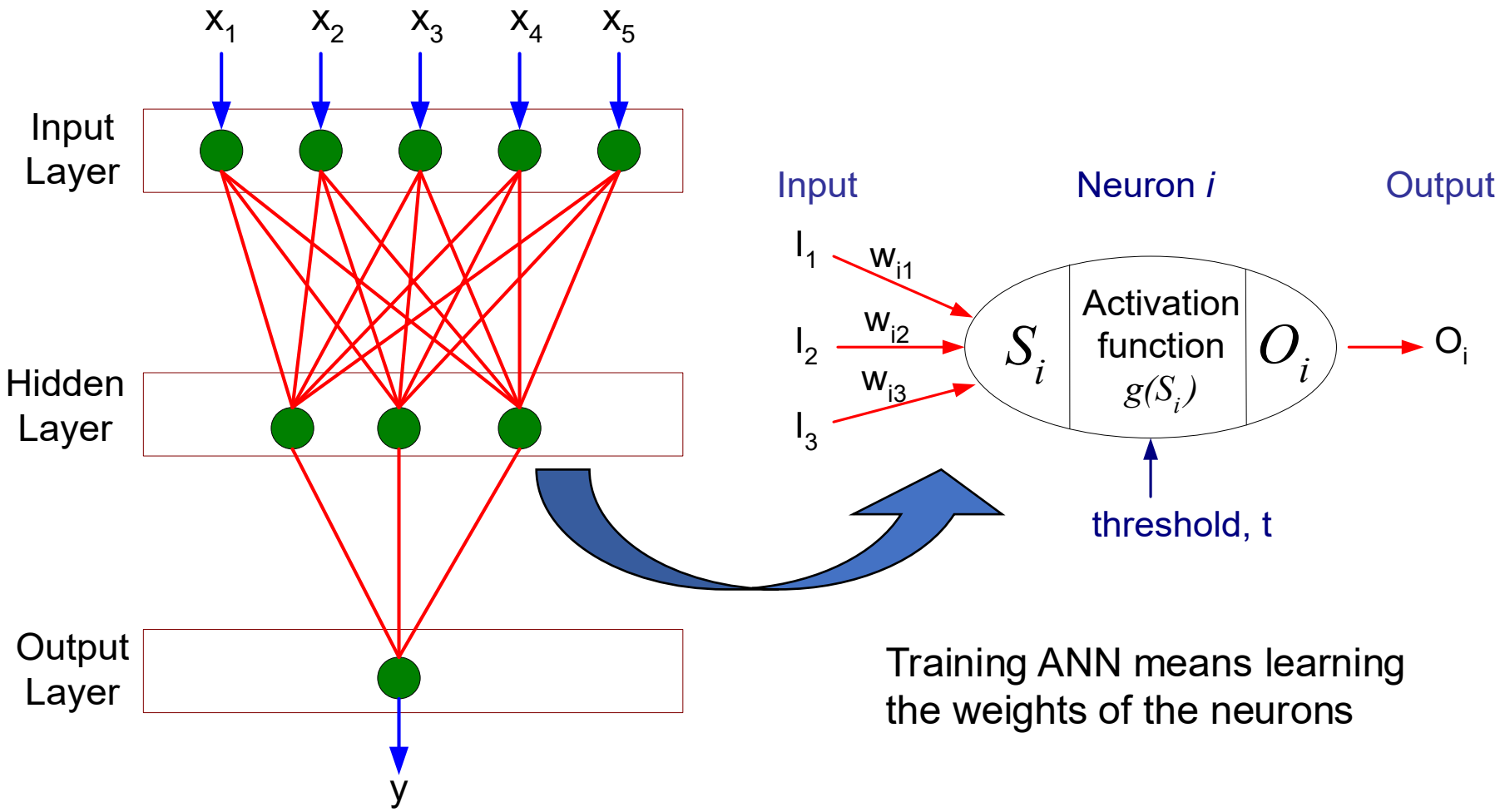
Multilayer Neural Network

- **Hidden Layers**: intermediary layers between input and output layers.
- Different type of **activation functions** (sigmoid, linear, hyperbolic tangent, etc.).
- Multi-layer neural network can solve any type of classification task involving nonlinear decision surfaces.
- Perceptron is single layer.
- We can think to each hidden node as a perceptron that tries to construct one hyperplane, while the output node combines the results to return the decision boundary.





General Structure of ANN

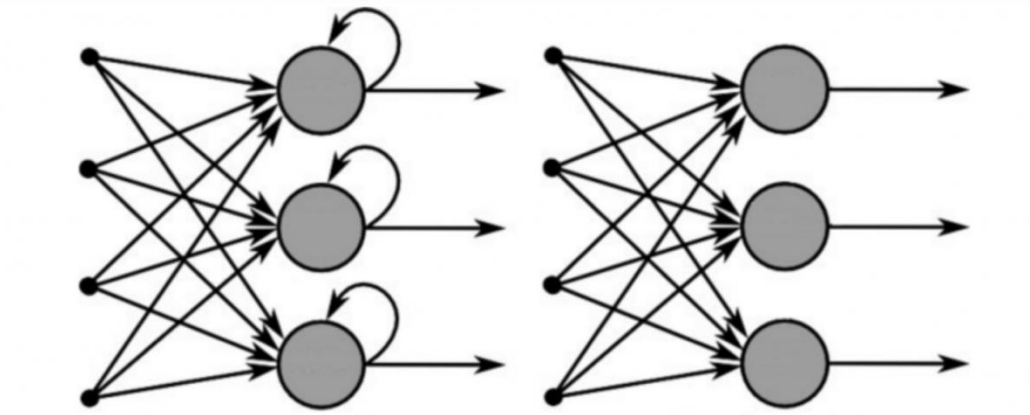


Training ANN means learning the weights of the neurons

- The neurons perform a linear transformation on this input using the weights and biases.
- An activation function is applied to it
- The output moves to the next hidden layer

Artificial Neural Networks (ANN)

- Various types of neural network topology
  - single-layered network (perceptron) versus multi-layered network
  - **Feed-forward**: connections only between nodes of level L_i and the next one L_{i+1}
  - **Recurrent network:** feedback connections in which **outputs** of the model **are fed back into itself.**



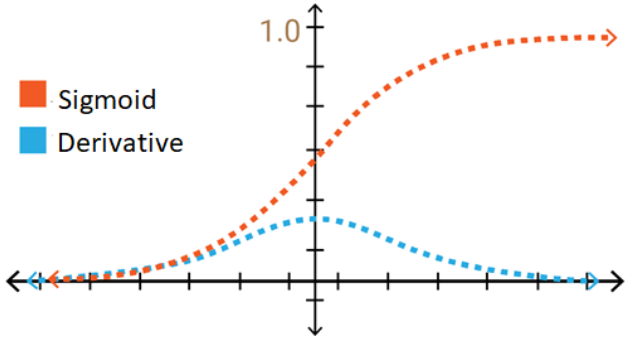
- Various types of activation functions (f)

$$Y = f(\sum_i w_i X_i)$$

Activation function: Sigmoid

- Logistic function (Sigmoid): values in [0,1]

$$f(net) = \sigma(net) = \frac{1}{1 + e^{-net}}$$



- Derivative:

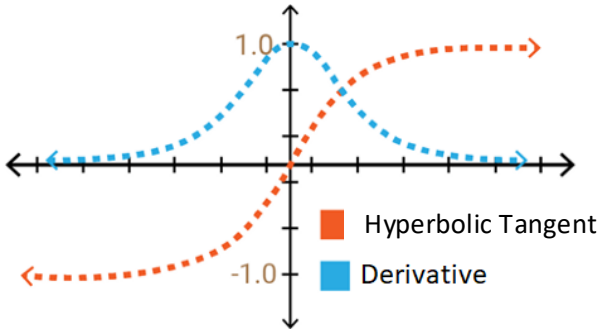
$$\sigma'(net) = \frac{\partial}{\partial net}\left(\frac{1}{1 + e^{-net}}\right) = \frac{e^{-net}}{(1 + e^{-net})^2} = \sigma(net)\big(1 - \sigma(net)\big)$$

Activation function: Hyperbolic Tangent

- Hyperbolic Tangent has values in [-1,1]
- More complex wrt Sigmoid
- Better than sigmoid because symmetric wrt 0 and leads to a faster convergence
- We can obtain it by Sigmoid

$$f(net) = \tau(net) = 2\sigma(2 \cdot net) - 1$$

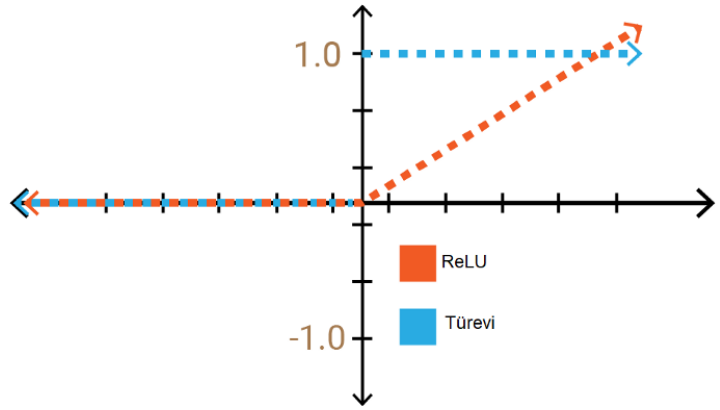
- Derivative: $\tau'(net) = 1 - \tau(net)^2$



Activation function: RELU

- Rectified Linear Unit: values in [0,inf]

$$f(net) = \begin{cases} 0\ (or\ \epsilon)\ for\ net < 0 \\ net \qquad\qquad for\ net \geq 0 \end{cases}$$



- It will output the input directly if it is positive, otherwise, it will output zero

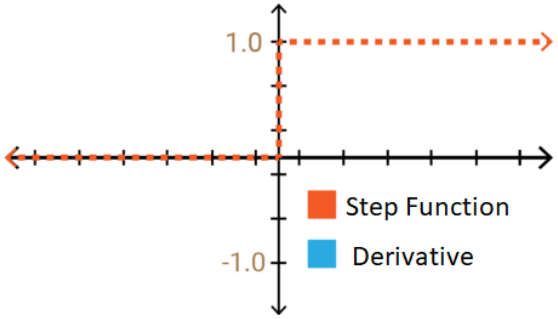
- Overcomes the **vanishing gradient problem**, allowing models to **learn faster** and perform better.

Activation function: Softmax

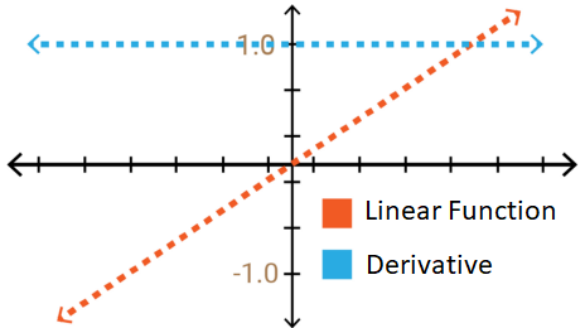
- Each value in the output of the softmax function is interpreted as the probability of membership for each class.
- Generalization of the logistic function to multiple dimensions
- Useful for addressing a multiclass problem

$$z_k = f(net_k) = \frac{e^{net_k}}{\sum_{c=1\ldots s} e^{net_c}}$$

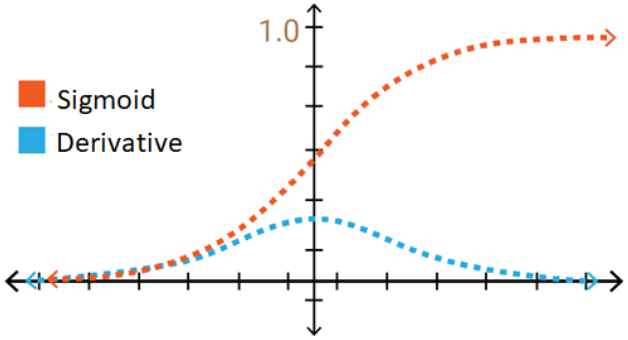
Activation Functions Summary



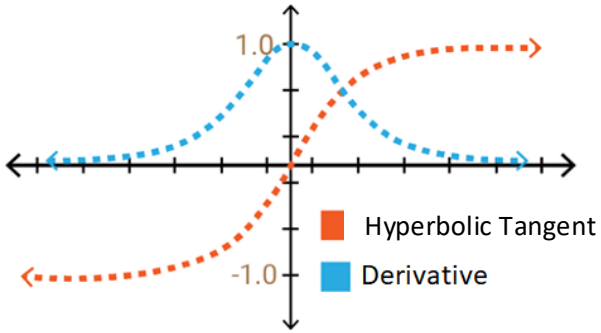
$$f(x) = \begin{cases} 0 \; for \; x < 0 \\ 1 \; for \; x \geq 0 \end{cases}$$



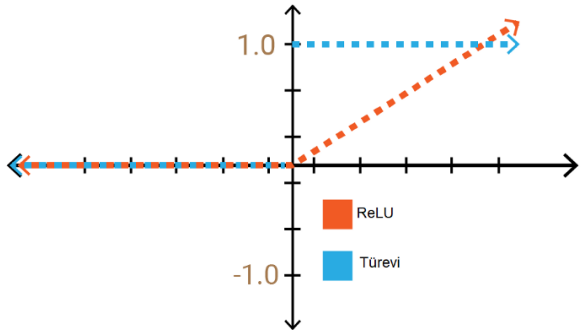
$$f(x) = x$$



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



$$f(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

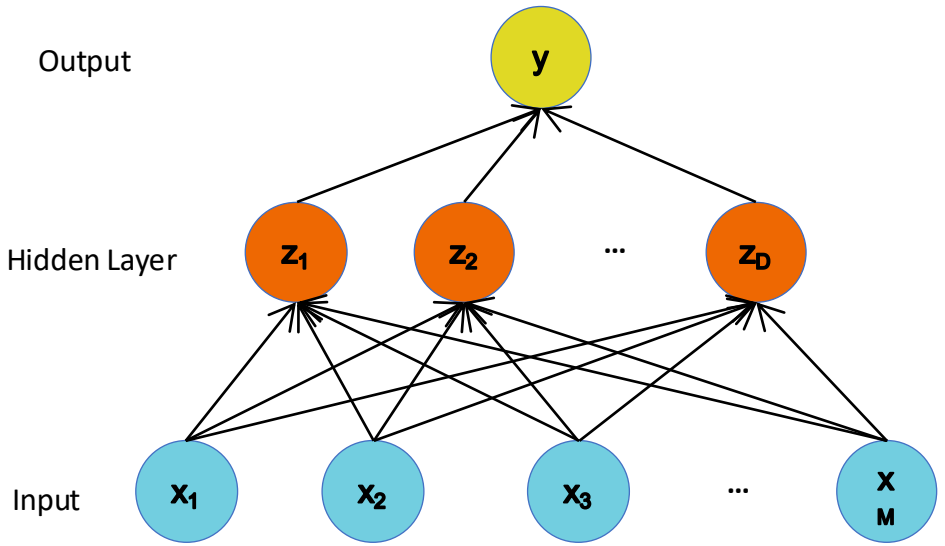


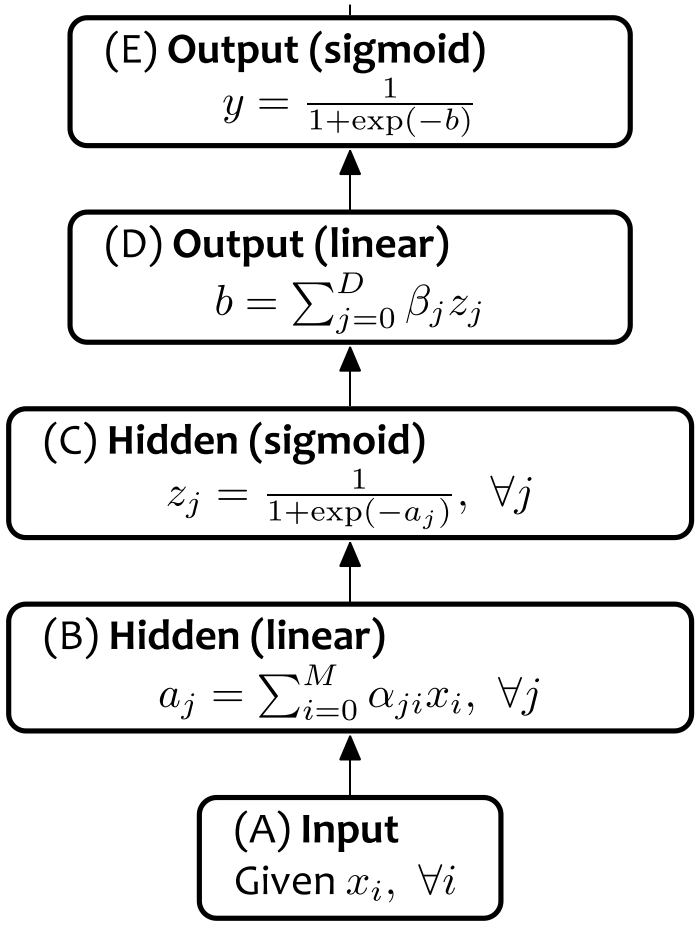
$$f(x) = \begin{cases} 0 \; (or \; \epsilon) \; for \; x < 0 \\ x \qquad\qquad for \; x \geq 0 \end{cases}$$

$$f(x_j) = \frac{e^{x_j}}{\sum_k e^{x_k}}$$

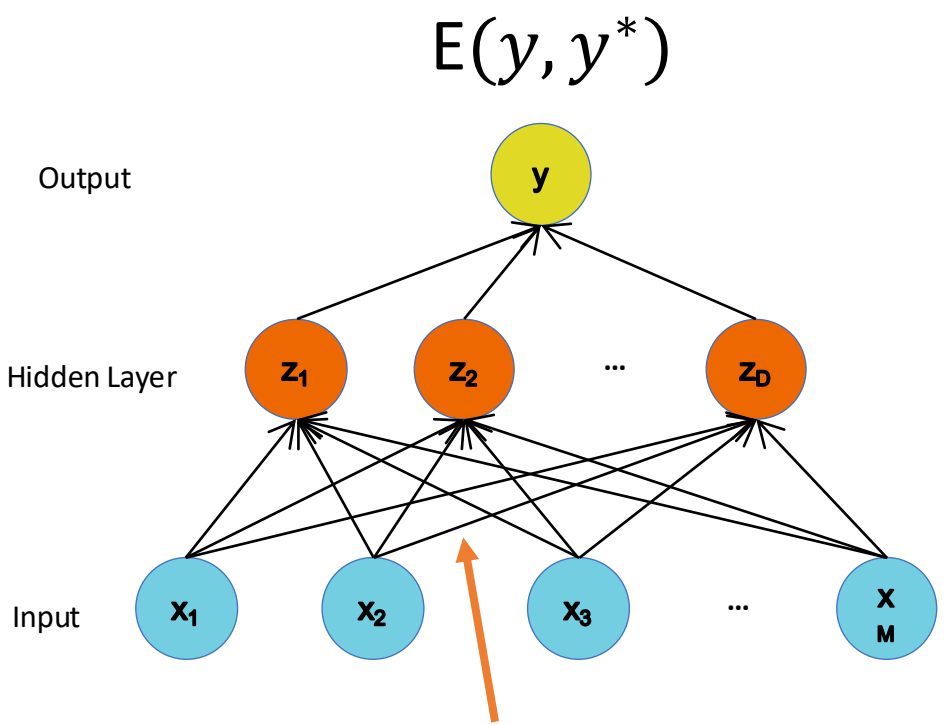
Softmax Function

Training Multilayer NN

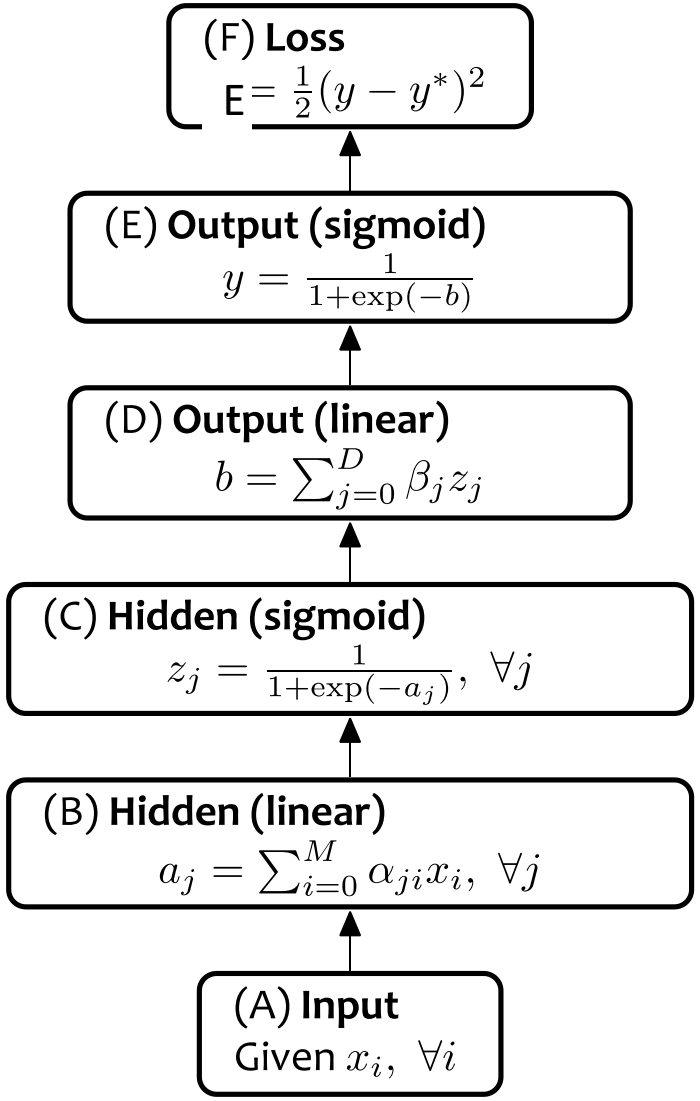




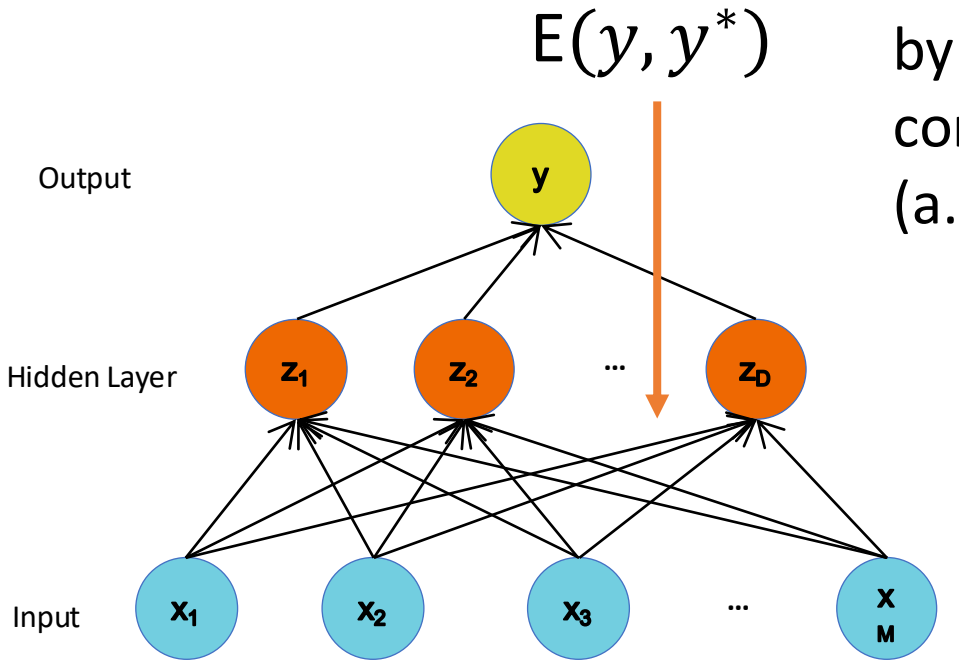
Training Multilayer NN



How do we update these weights given the loss is available only at the output unit?



Error Backpropagation

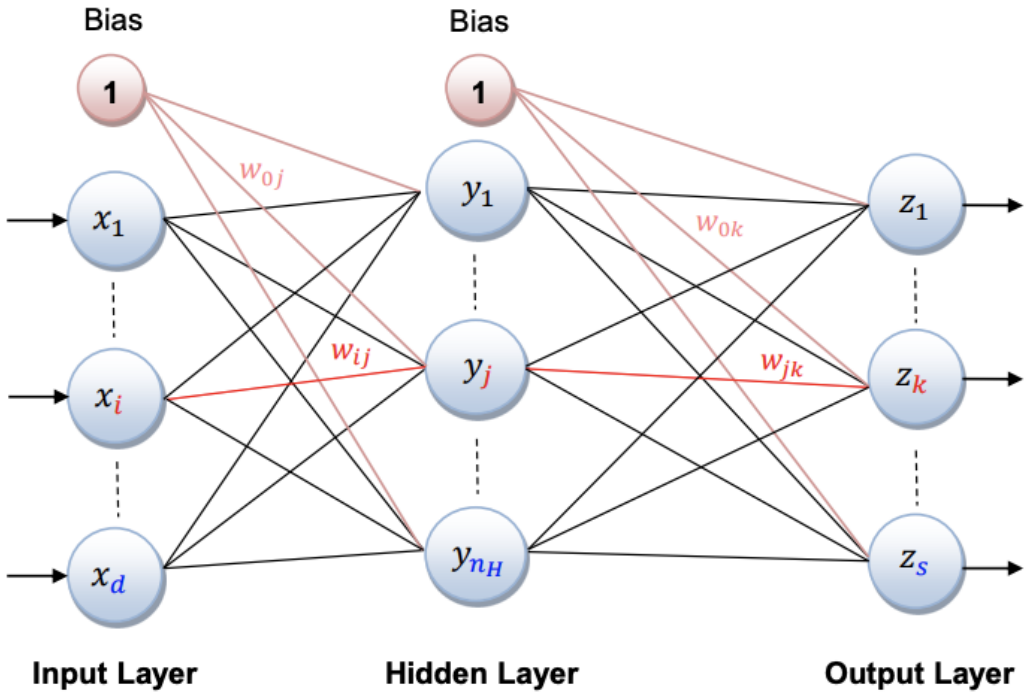


Error is computed at the output and propagated back to the input by chain rule to compute the contribution of each weight (a.k.a. derivative) to the loss

A 2-step process

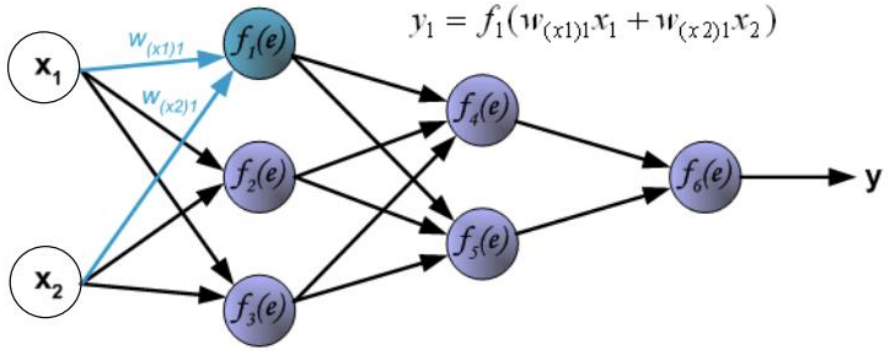
1. Forward pass - Compute the network output
2. Backward pass – Compute the loss function gradients and update

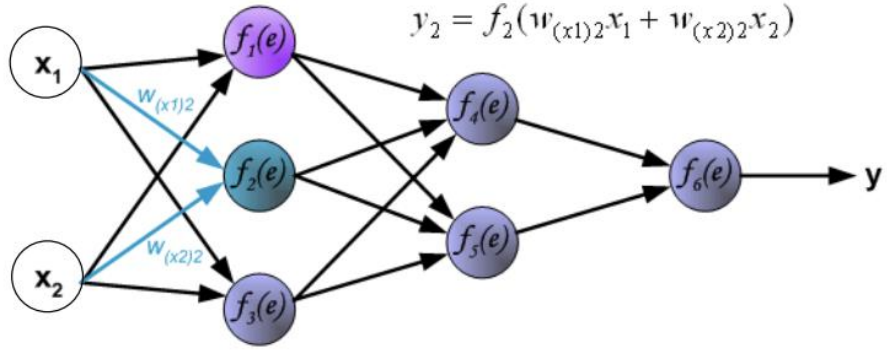
Forward propagation

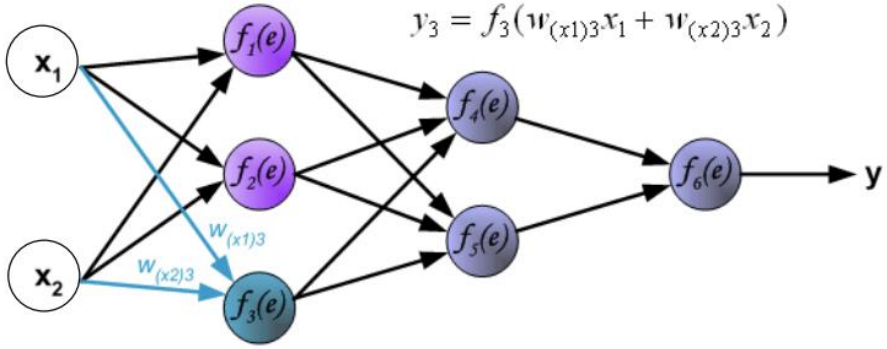


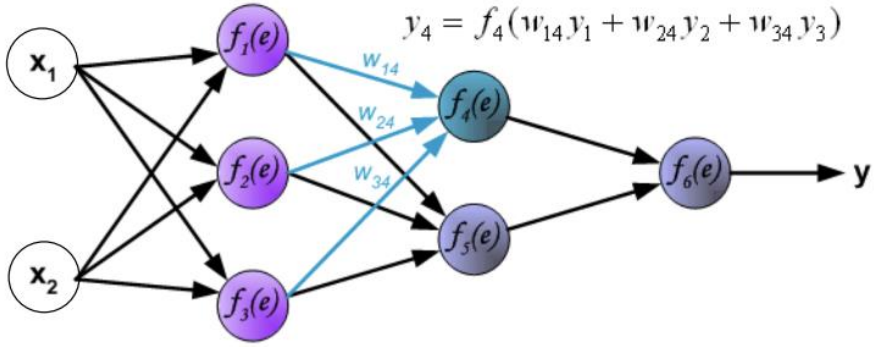
$$z_k = f\left(\sum_{j=1\ldots n_H} w_{jk} \cdot y_j + w_{0k}\right) = f\left(\sum_{j=1\ldots n_H} w_{jk} \cdot f\left(\sum_{i=1\ldots d} w_{ij} \cdot x_i + w_{0j}\right) + w_{0k}\right)$$

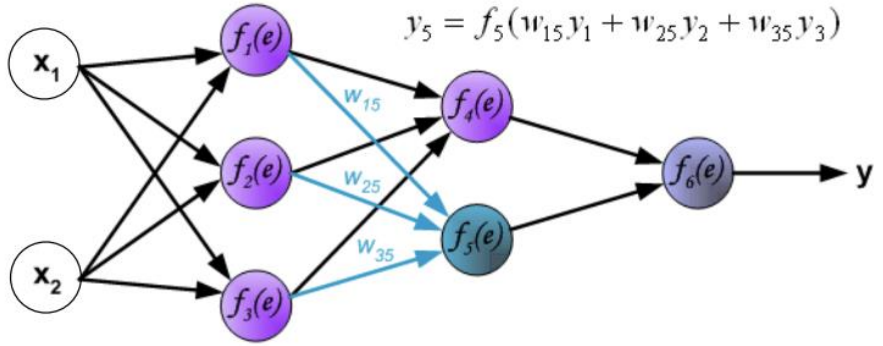
Forward propagation

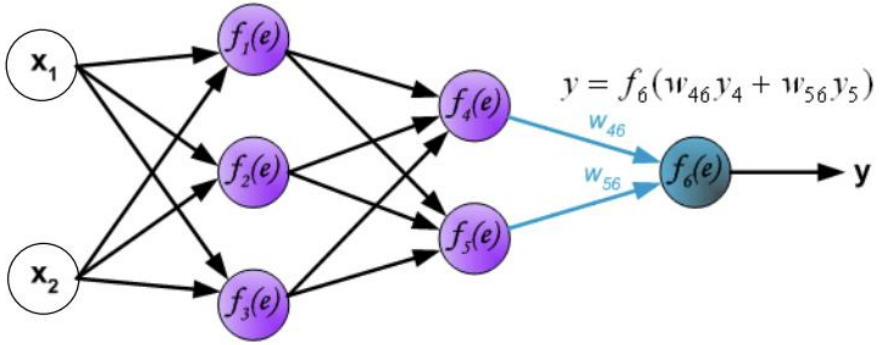












Learning Multi-layer Neural Network

- Can we apply perceptron learning to each node, including hidden nodes?
- Perceptron computes error $e = y - f(w,x)$ and updates weights accordingly
- **Problem: how to determine the true value of *y* for hidden nodes?**
- Approximate error in hidden nodes by error in the output nodes
- Problems:
  - Not clear how adjustment in the hidden nodes affect overall error
  - No guarantee of convergence to optimal solution

Gradient Descent for Multilayer NN

- Error function to minimize: $$E = \frac{1}{2}\sum_{i=1}^{N}\left(y_i - f(\sum_j w_j x_{ij}) \right)^2$$

Quadratic function from which we can find a global minimum solution

- Weight update: $$w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$$

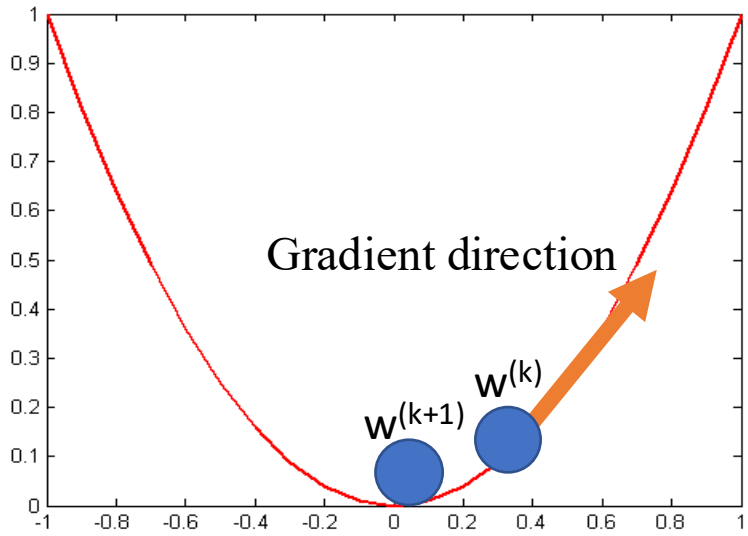
- **The second term states:**
  - adjust the weights based on the gradient descent technique, i.e. **proportionally to the gradient of the error function**
  - that weight should be increased in a direction reducing the overall error term
- The gradient descent learning rule moves a small (λ) step in the negative gradient direction
- Gradient indicates the direction of growing of the function
- Activation function *f* must be differentiable
- Error function is nonlinear GD method can get trapped in a local minimum

Gradient Descent for Multilayer NN

- Weights are updated in the opposite direction of the gradient of the loss function

$$w_j^{(k+1)} = w_j^{(k)} - \lambda \frac{\partial E}{\partial w_j}$$

- Gradient direction is the direction of uphill of the error function
- By taking the negative we are going downhill
- Hopefully to a minimum of the error

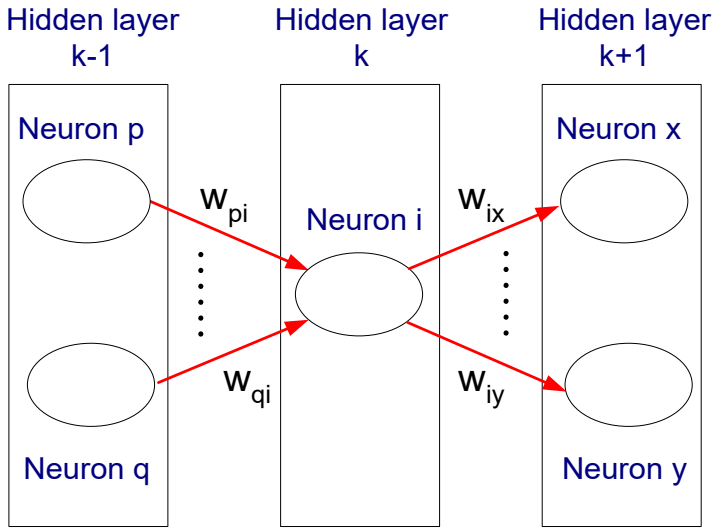


Gradient Descent for Multilayer NN

For **sigmoid neurons trained with squared-error loss**, backpropagation computes an error signal δ_i for each neuron.

Sigmoid activation and derivative

$$o_i = \sigma(net_i) = \frac{1}{1 + e^{-net_i}}, \qquad \sigma'(net_i) = o_i(1 - o_i)$$



Output neuron i**:** compute the error signal directly from its target (y_i:

$$\boxed{\delta_i = (y_i - o_i) \cdot o_i(1 - o_i)}$$

Hidden neuron i**:** compute the error signal from the neurons in the next layer:

$$\boxed{\delta_i = o_i(1 - o_i) \sum_{j \in \Phi_i} w_{ij}\, \delta_j}$$

Here, Φ_i contains the neurons receiving connections from neuron i.

Weight update — same rule for both output and hidden neurons

$$\boxed{\Delta w_{pi} = \lambda \cdot o_p \cdot \delta_i \qquad w_{pi}^{(k+1)} = w_{pi}^{(k)} + \Delta w_{pi}}$$

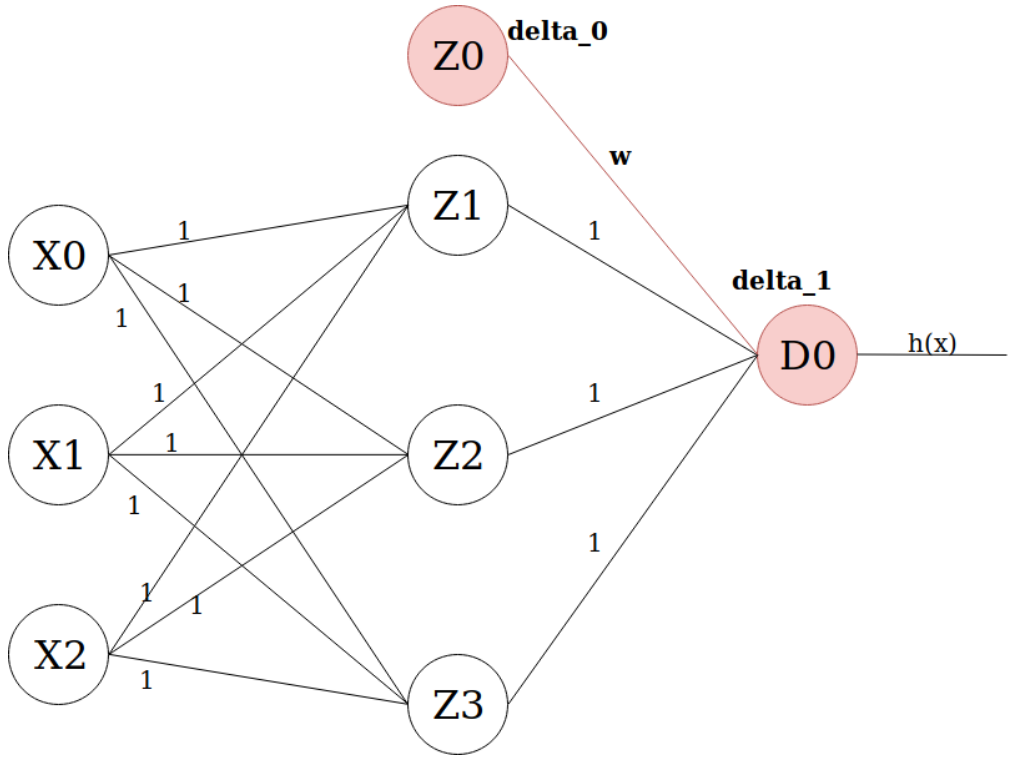
Here, o_p is the source neuron's output, w_{pi} connects p to i, and λ is the learning rate.

Compute error signals **from the output layer backwards**, using the current weights, then update the weights.

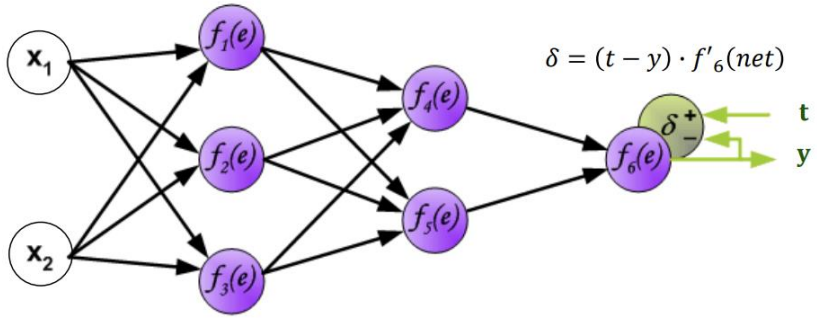
Backpropagation in other words

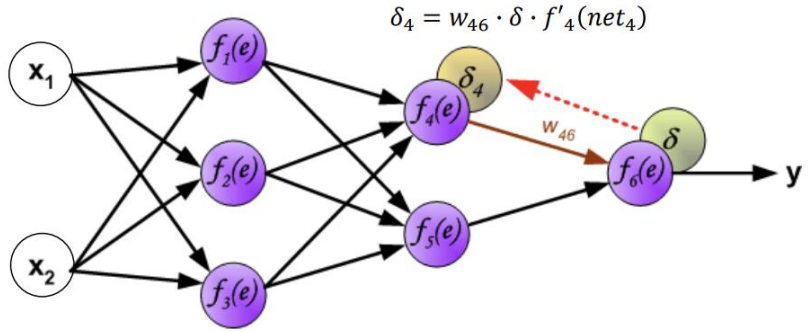
- In order to get the loss of a node we multiply the value of its corresponding f’(z) by the loss of the node it is connected to in the next layer (δ_1), by the weight of the link connecting both nodes

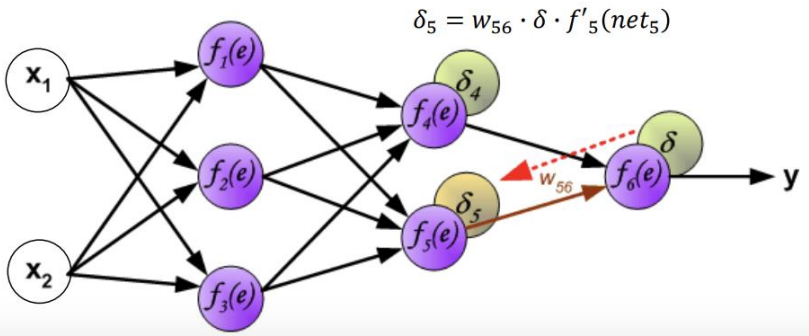
- We do the delta calculation step at every unit, back-propagating the loss into the neural net, and finding out what loss every node/unit is responsible for.

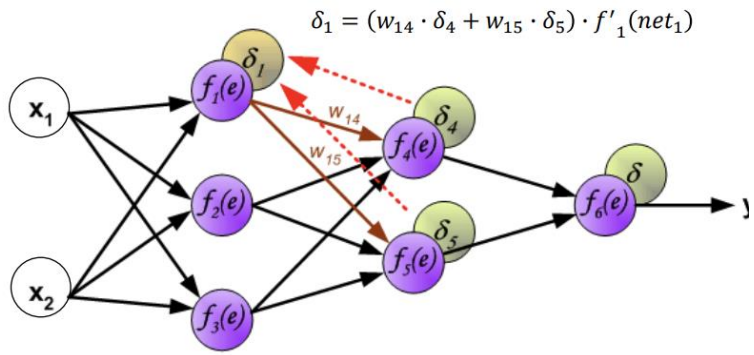


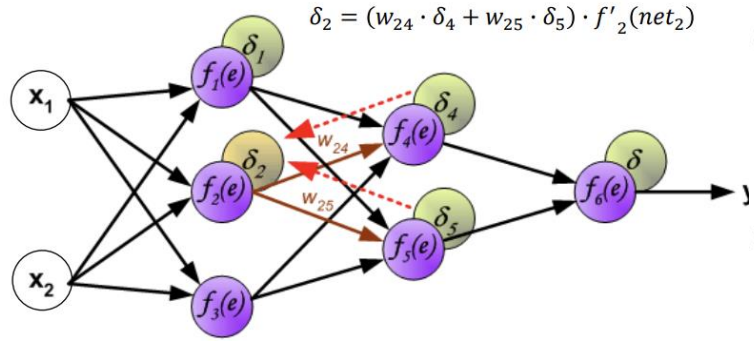
Backpropagation

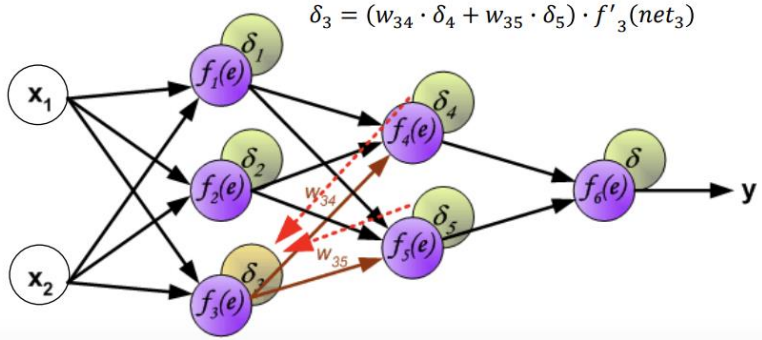






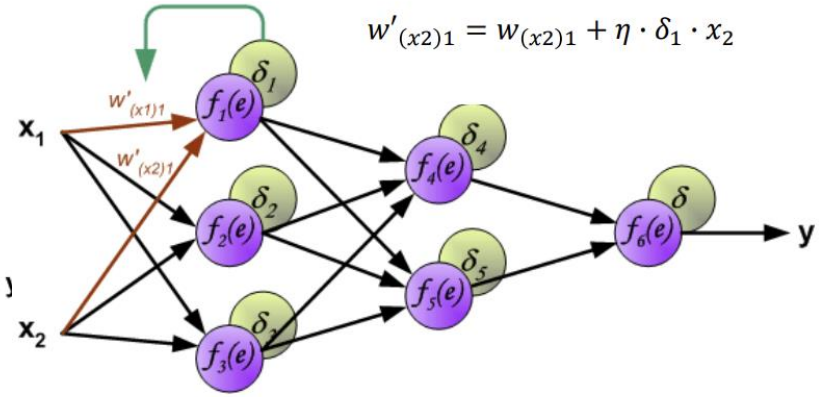






$$w'_{(x1)1} = w_{(x1)1} + \eta \cdot \delta_1 \cdot x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \cdot \delta_1 \cdot x_2$$



Epoch

An epoch is one complete pass through the entire training set.

Training for 20 epochs means processing each training example 20 times, usually reshuffling the training set before each epoch.

Mini-batch

The mini-batch size is the number of training examples processed before updating the network's weights once.

For each mini-batch, the network:

1. Performs forward and backward passes for its examples.
2. Accumulates their gradients, usually taking the average.
3. Updates the weights once, using that average gradient.

For example, with 1,000 training examples and mini-batch size 100, each epoch contains 10 mini-batches, so the weights are updated 10 times per epoch.

Batch size	**Training approach**	**Weight updates per epoch**
1	Stochastic gradient descent	n
$1 < b < n$	Mini-batch gradient descent	$\lceil n/b \rceil$
n	Full-batch gradient descent	1

Learning a MLNN

Initialize:

- nH = number of hidden neurons
- w = randomly initialized weights and biases
- λ = learning rate
- Num_epochs = maximum number of epochs
- size_mb = mini-batch size

repeat:
    epoch = epoch + 1

    Randomly shuffle the training set
    Divide it into mini-batches of size size_mb
    (the last mini-batch may contain fewer examples)

    for each mini-batch B:
        accumulated_gradient = 0

        for each training example (x, y) in B:
            Forward pass: compute the network prediction
            Compute the loss for this example
            Backward pass: compute the loss gradient with respect to w
            Add the gradient to accumulated_gradient

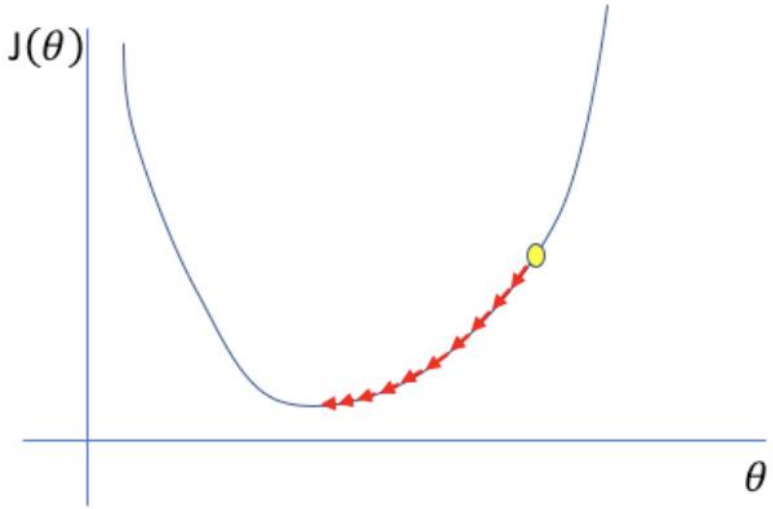
        mean_gradient = accumulated_gradient / |B|
        Update parameters: $w = w - \lambda \cdot$ mean_gradient

    Compute loss and accuracy on the training set
    Compute loss and accuracy on the validation set
    Check the convergence criterion

until convergence or epoch ≥ Num_epochs

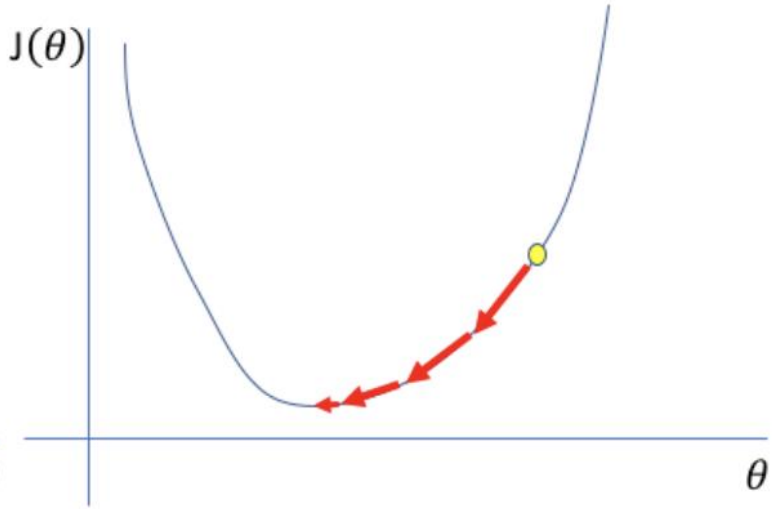
Learning Rate

Too low



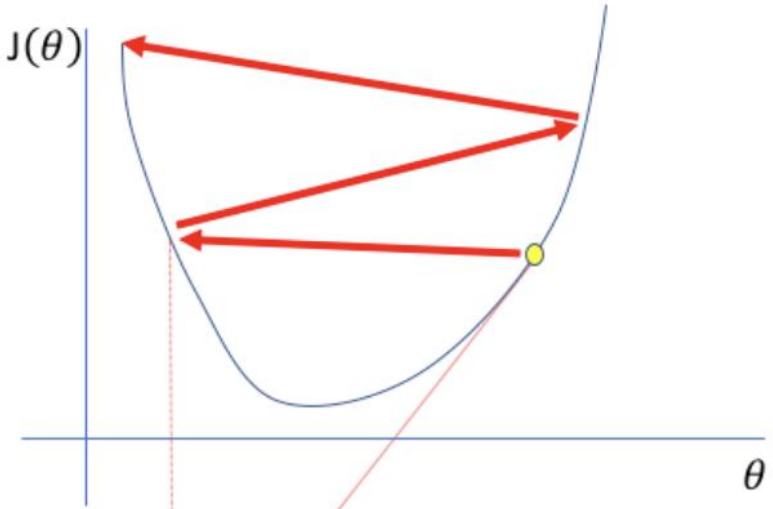
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

On the Key Importance of Error Functions

- The error/loss/cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be compared.
- It is important, therefore, that **the function faithfully represents our design goals**.
- If we choose a poor error function and obtain unsatisfactory results, the fault is ours for badly specifying the goal of the search.

Design Issues in ANN

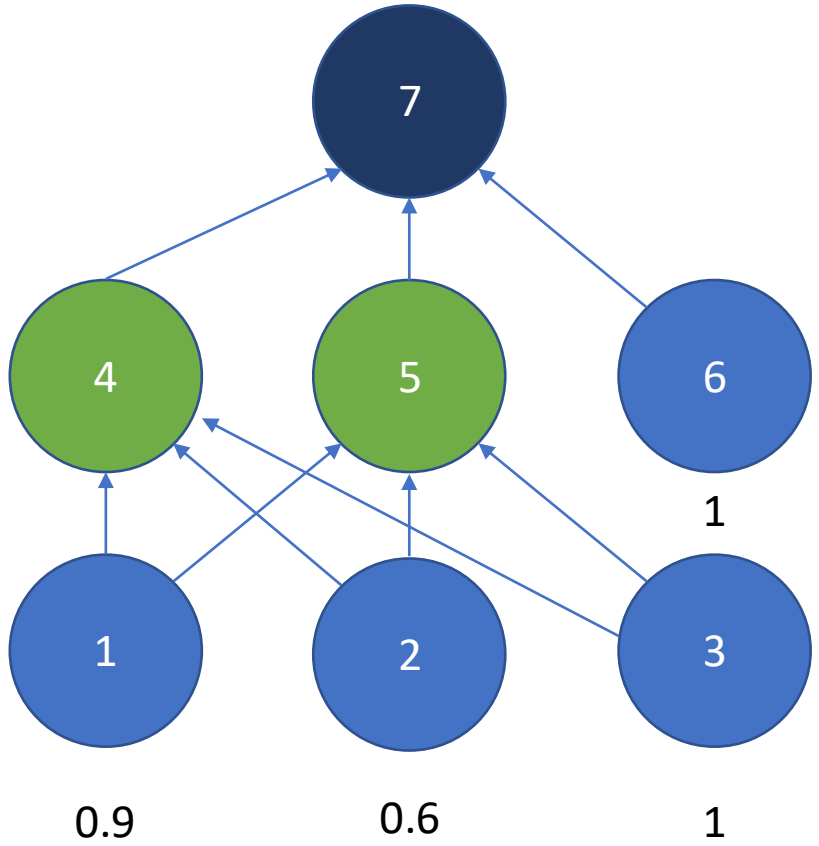
- Number of nodes in input layer
  - One input node per binary/continuous attribute
  - k or $log_2 k$ nodes for each categorical attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k nodes for k-class problem
- Number of nodes in hidden layer
- Initial weights and biases

Characteristics of ANN

- Multilayer ANN are universal approximators but could suffer from ***overfitting*** if the network is too large.
- Gradient descent may converge to ***local minimum***.
- Model building can be very time consuming, but testing can be very fast.
- Can handle redundant attributes because weights are automatically learnt.
- Sensitive to noise in training data.
- Difficult to handle missing attributes.

Backpropagation: example

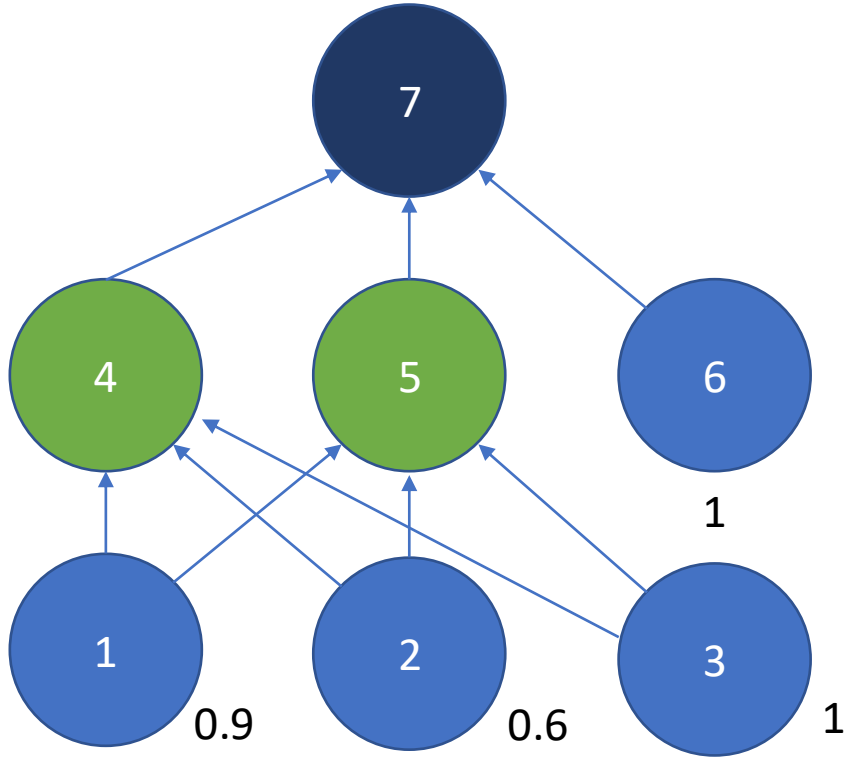
- All the weights are initialized to 0.5
- The learning rate is 1
- Activation function: $Z_j = \mathrm{f}(net_j) = \frac{1}{1+e^{-net_j}}$
- $f'(net) = Z_j(1 - Z_j)$
- $\Delta w_{ij} = \lambda \delta_j Z_{\mathrm{i}}$
- Output node: $\delta_j = (T_j - Z_j) f'(net)$
- Hidden node: $\delta_j = \sum_k (\delta_k w_{jk}) f'(net)$



Backpropagation: example

- $net_4 = 0.9 * 0.5 + 0.6 * 0.5 + 1 * 0.5 = 1.25$
- $net_5 = 0.9 * 0.5 + 0.6 * 0.5 + 1 * 0.5 = 1.25$
- $z_4 = {}^{1}/_{(1+e^{-1.25})} = 0.77$
- $z_5 = {}^{1}/_{(1+e^{-1.25})} = 0.77$
- $net_7 = 0.77 * 0.5 + 0.77 * 0.5 + 1 * 0.5 = 1.27$
- $z_7 = {}^{1}/_{(1+e^{-1.27})} = 0.78$

- $Z_j = f(net_j) = \frac{1}{1+e^{-net_j}}$
- $f'(net) = Z_j(1 - Z_j)$
- $\Delta w_{ij} = \lambda \delta_j Z_j$
- Output node: $\delta_j = (T_j - Z_j) f'(net)$
- Hidden node: $\delta_j = \sum_k (\delta_k w_{jk}) f'(net)$



Backpropagation: example

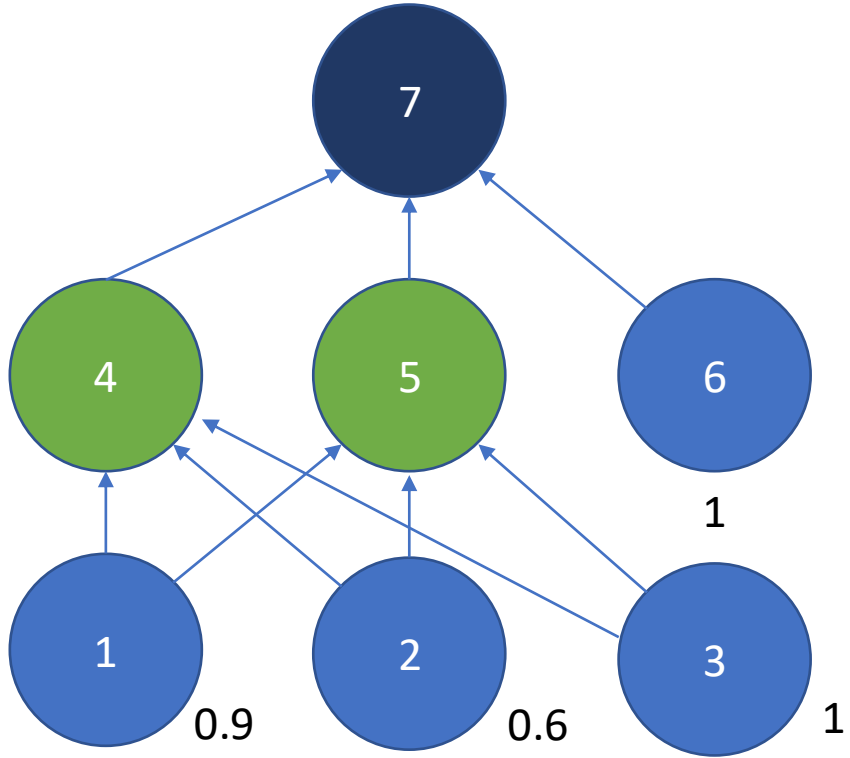
- $net_4 = 1.25,\ net_5 = 1.25,\ z_4 = 0.77,\ z_5 = 0.77,$
  $net_7 = 1.27,\ z_7 = 0.78$

- $\delta_7 = (0 - 0.78) * 0.78 * (1 - 0.78) = -0.13$

- $\delta_4 = (-0.13 * 0.5) * 0.77 * (1 - 0.77) = -0.01$

- $\delta_5 = (-0.13 * 0.5) * 0.77 * (1 - 0.77) = -0.01$

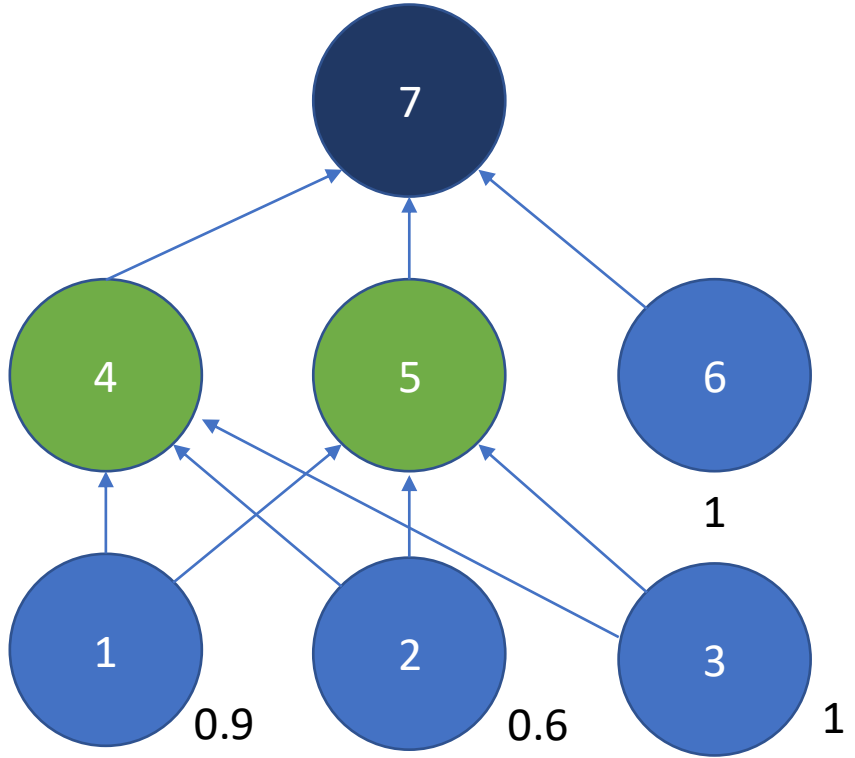
- $Z_j = \mathrm{f}(net_j) = \frac{1}{1+e^{-net_j}}$
- $f'(net) = Z_j(1 - Z_j)$
- $\Delta w_{ij} = \lambda \delta_j Z_j$
- Output node: $\delta_j = (T_j - Z_j) f'(net)$
- Hidden node: $\delta_j = \sum_k (\delta_k w_{jk})\, f'(net)$



Backpropagation: example

- $net_4 = 1.25,\ net_5 = 1.25,\ z_4 = 0.77,\ z_5 = 0.77,\ net_7 = 1.27,\ z_7 = 0.78$
- $\delta_7 = -0.13$
- $\delta_4 = -0.01$
- $\delta_5 = -0.01$
- $w_{47} = 0.5 + (1 * -0.13 * 0.77) = 0.39$
- $w_{57} = 0.5 + (1 * -0.13 * 0.77) = 0.39$
- $w_{67} = 0.5 + (1 * -0.13 * 1) = 0.36$

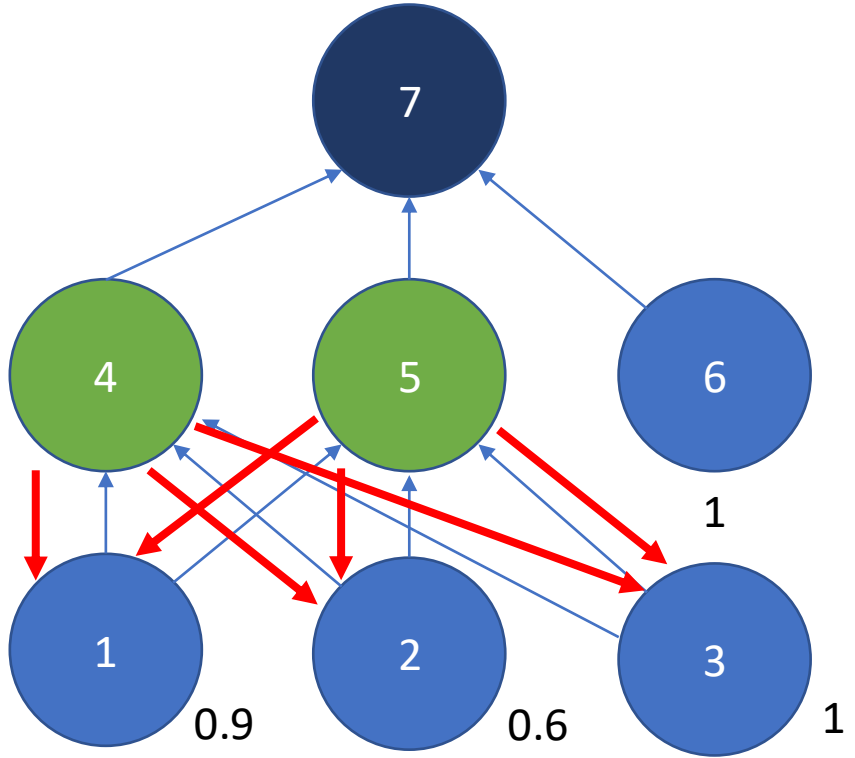
- $Z_j = \mathrm{f}(net_j) = \frac{1}{1+e^{-net_j}}$
- $f'(net) = Z_j(1 - Z_j)$
- $\Delta w_{ij} = \lambda \delta_j Z_j$
- Output node: $\delta_j = (T_j - Z_j) f'(net)$
- Hidden node: $\delta_j = \sum_k (\delta_k w_{jk})\, f'(net)$



Backpropagation: example

- $\delta_7 = -0.13, \delta_4 = -0.11, \delta_5 = -0.11$
- $w_{47} = 0.39, w_{57} = 0.39, w_{67} = 0.36$
- $w_{14} = 0.5 + (1 * -0.11 * 0.9) = 0.48$
- $w_{15} = 0.5 + (1 * -0.11 * 0.9) = 0.48$
- $w_{24} = 0.5 + (1 * -0.11 * 0.6) = 0.49$
- $w_{25} = 0.5 + (1 * -0.11 * 0.6) = 0.49$
- $w_{34} = 0.5 + (1 * -0.11 * 1) = 0.48$
- $w_{35} = 0.5 + (1 * -0.11 * 1) = 0.48$

- $Z_j = \mathrm{f}(net_j) = \frac{1}{1+e^{-net_j}}$
- $f'(net) = Z_j(1 - Z_j)$
- $\Delta w_{ij} = \lambda \delta_j Z_j$
- Output node: $\delta_j = (T_j - Z_j) f'(net)$
- Hidden node: $\delta_j = \sum_k (\delta_k w_{jk}) f'(net)$



References

- Artificial Neural Network. Chapter 5.4 and 5.5. Introduction to Data Mining.

