

Biologically Inspired Computation

Dr Marta Vallejo

m.vallejo@hw.ac.uk

Lecture 3

Multi-layer Perceptron

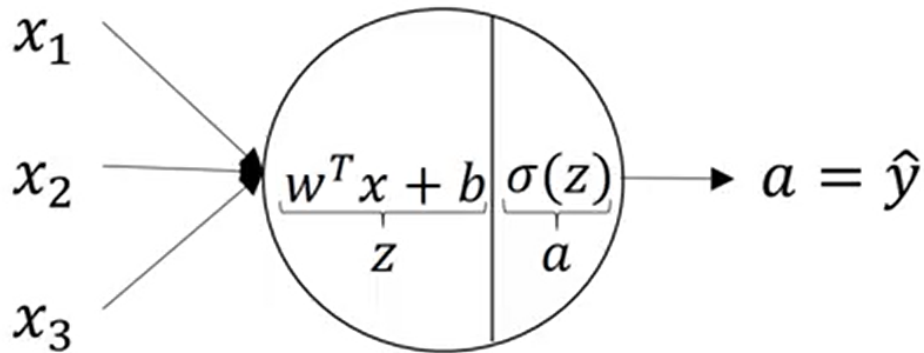
- Multilayer Perceptron
- Hyperparameters
- Different Separable Problems
- Number of Hidden Units
- Universal Approximation
- More about Activation Functions
 - Monotonicity
 - Sigmoid Function: Vanishing Gradients
 - Tanh Activation Function
 - Linear Activation Function
 - Selecting an Activation Function
- The bias
- Multi-class Classification
- Multilayer Perceptron Implementation

Multi-layer Perceptron

Also called **shallow neural networks** in the deep learning community

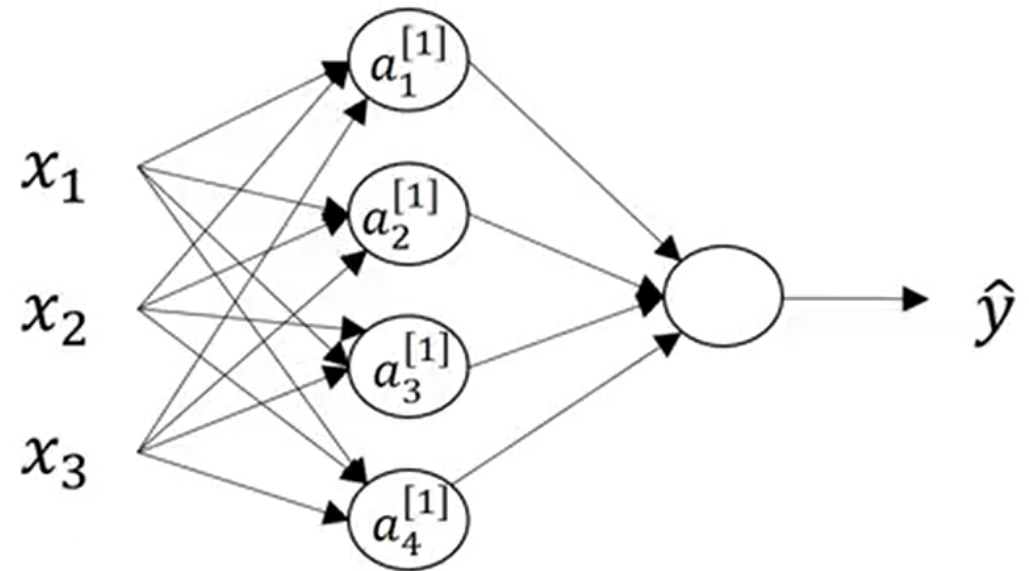
New complexity step:

Single Neuron



we stuck together different sigmoid units

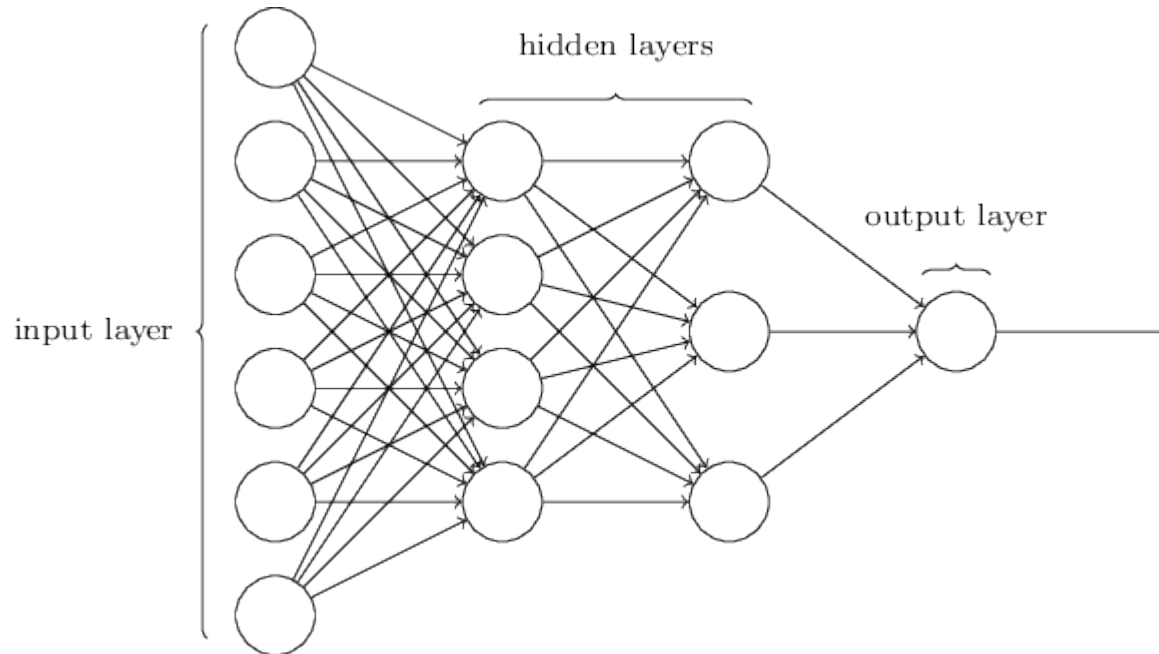
2-layer Neural Network



It is only two layers because we do not count the input layer

Dimensionality of a layer: number of units in this layer

Multi-layer Perceptron



Neuron layers:

- **1 input layer:** just contains the input vector and does not perform any computation
- **1 or more hidden layers:** transmit information from the input layer to the output layer
- **1 output layer:** After applying their activation function, the neurons in the output layer contain the output vector

The **hidden layer** is a layer that represents hidden features that influence the final outcome of the prediction.

Multi-layer Perceptron

Fully connected architecture:

Each node is connected to EVERY node in the adjacent layers and NONE of the nodes in the same layers

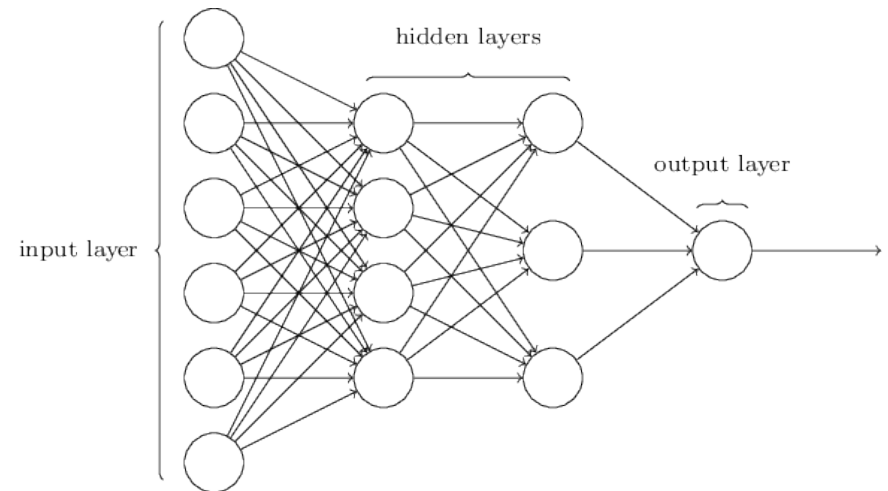
Features are mutually independent

Type of connections: **Feedforward**
(arrows represent the direction of the data)

Learning Method: Supervised

Representation that is mathematical convenient → matrix representation

Layers' values can be calculated by
multiplication of matrices



Hyperparameters

Hyperparameters are the variables which determines the network structure and how the network is trained (we have already seen the learning rate).

In terms of architecture, we have:


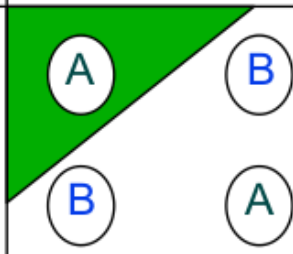
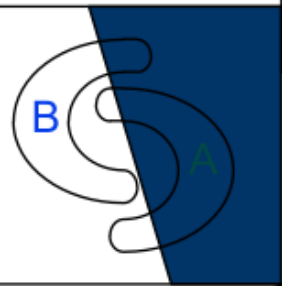

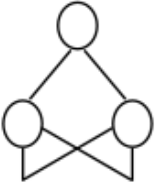
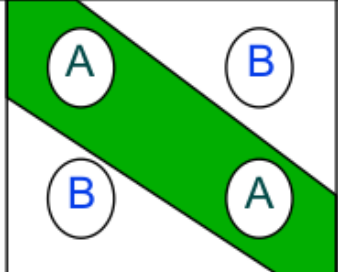
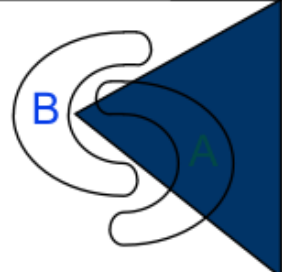

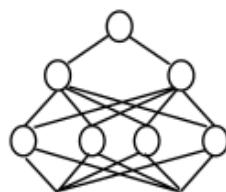
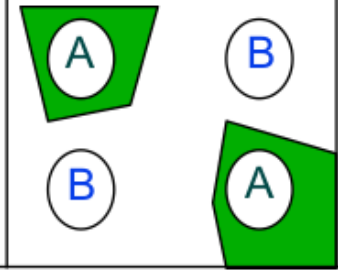

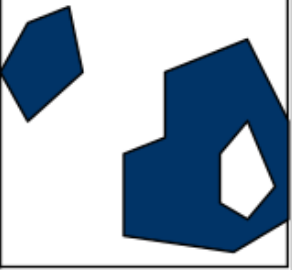
- The number of nodes of the hidden layer
- The number of layers
- Activation functions

Hyperparameters should be set before training (before optimising the weights and bias).

There is **not** a systematic way of choosing them. Knowledge is based on previous experience and trial and error methods.

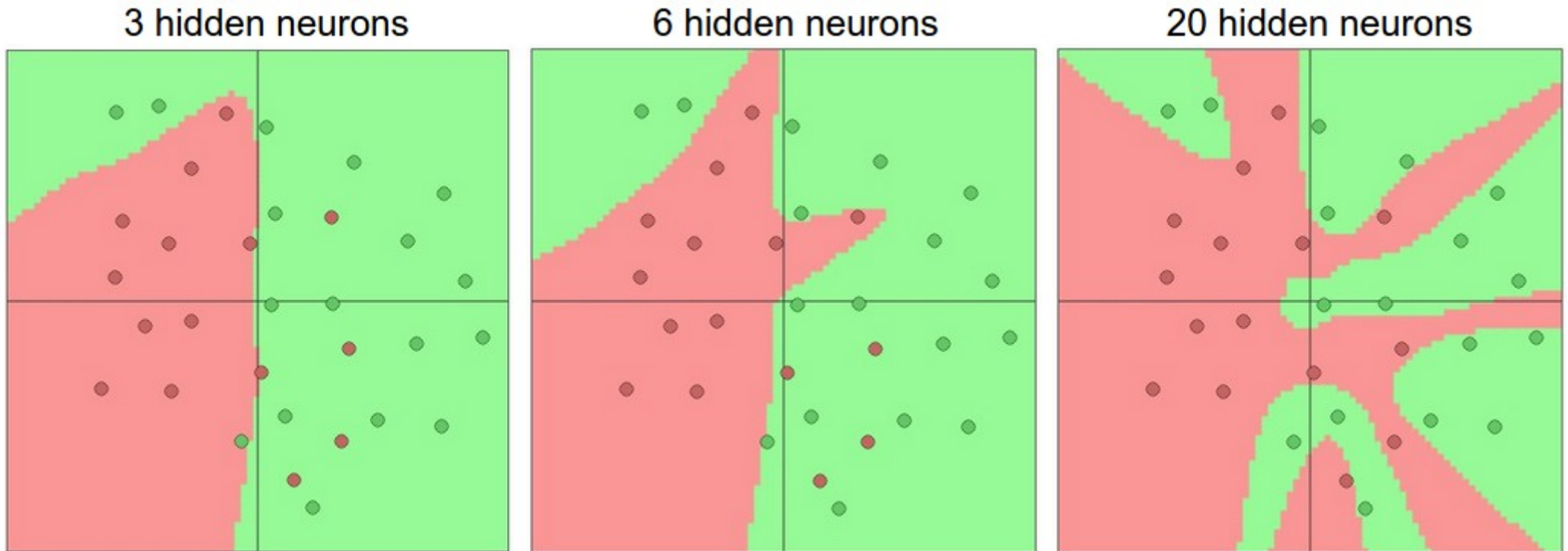
Number of Hidden Layers

Complexity of the problem in terms of the **number of layers** of the MLP.

Structure	Types of Decision Regions	Exclusive-OR Problem	Classes with Meshed regions	Most General Region Shapes
Single-Layer 	Half Plane Bounded By Hyperplane			
Two-Layer 	Convex Open Or Closed Regions			
Three-Layer 	Arbitrary (Complexity Limited by No. of Nodes)			

Number of Hidden Neurons

Effects of increasing the **number of neurons** in your hidden layer



The number of nodes affect the level of generalisation of your ANN.

Can you see a case of underfitting? And overfitting?

Play with this demo: <https://phiresky.github.io/neural-network-demo/>

Universal Approximation

Kurt Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks", Neural Networks, 4(2), 251–257

Universal approximation theorem: a feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous function.

The results applies for sigmoid and many other hidden layer activation functions

But the theorem does not give any hint on how to design the topology and the activation functions of any problem.

Learning with Hidden Units

Networks without hidden units are very limited in the input-output mappings they can model. Adding more layers of linear units do not help, **the result is still linear**.

We need multiple layers of adaptive non-linear hidden units. This gives us a **universal** approximator. But how can we train such nets?

- We need an efficient way of adapting all the weights, not just the last layer. This is hard. Learning the weights going into hidden units is equivalent to learning features.
- Nobody is telling us directly the topology of your hidden units.

Learning the hidden to output weights is **easy**.
Learning the input to hidden weights is **hard**.

Randomly perturb one weight and see if it improves performance. If so, save the change is **very inefficient**.

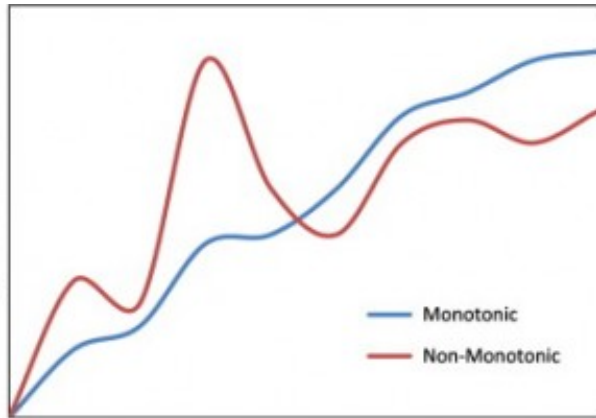
More about Activation Functions

- Activation Function: monotonicity
- Sigmoid Function problem
- Tanh – Hyperbolic Tangent Activation Function
- Linear Activation Function
- Relu Activation Function: next lecture
- Variants of Relu Activation Function: next lecture
- Selection of an Activation Function
- Summary of the most common Activation Functions

Activation Functions: monotonicity

A **monotonic** function is a function which is either entirely **nonincreasing** or **nondecreasing**. But why is important for the activation function?

We know that:

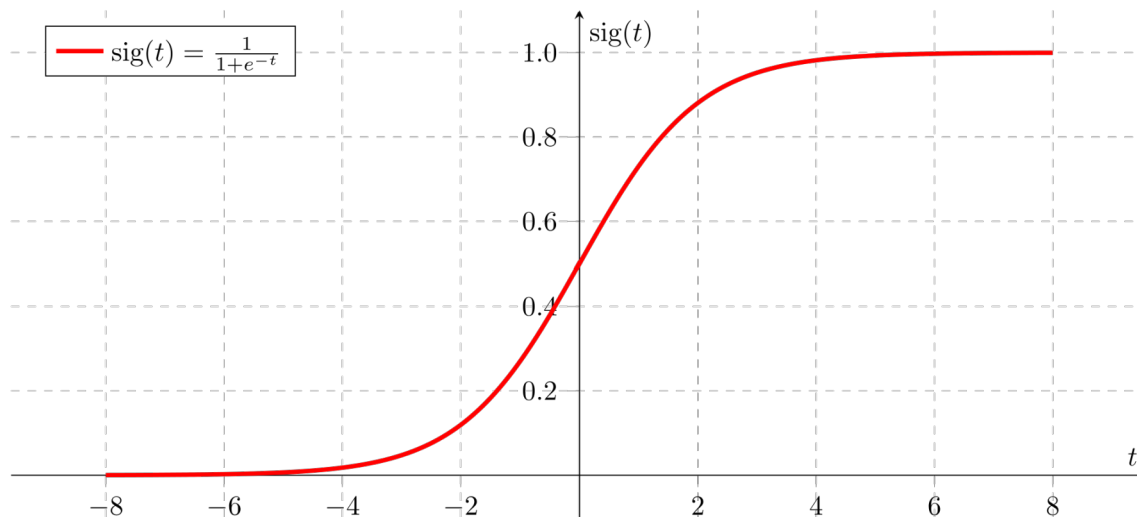


1. The value of a given weight represents the **importance** of the associated factor for the correct outcome.
2. During the training phase, the learning algorithm informs each neuron how much it should influence.

If the activation function is not monotonic, then increasing the neuron's weight might cause the opposite effect: to have less influence. The result would lead to a **chaotic behaviour** during training, with the network unlikely to converge to an accurate outcome.

Sigmoid Function: Vanishing Gradients

We know that the sigmoid function is continuous, derivable and monotonous.



From -2 to 2, the function is very sensitive to changes

However, it can cause that the neural network **get stuck** at the training time. This is due by the fact that if a strongly-negative input is provided to the sigmoid, its output values are very near zero. This is called **vanishing gradients**.

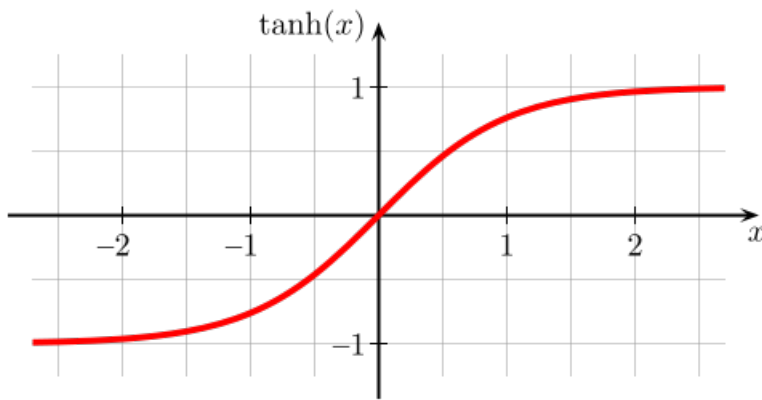
Since neural networks use gradients to learn, this can result in model parameters that are updated less regularly than it should be, **slowing down** the learning process.

Tanh or Hyperbolic Tangent Activation Function

The tanh function is mainly used for binary classification. It is a shifted version of sigmoid (they shared similar characteristics).

$$\tanh(z) = 2\sigma(2z) - 1$$

Both tanh and sigmoid are used in feed-forward networks.



The range of the tanh is from -1 to 1.
tanh is also sigmoidal (s - shaped)

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

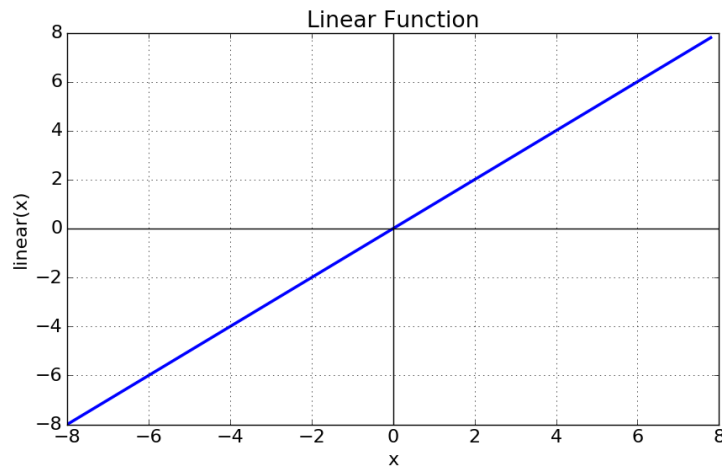
The mean of the values of the activation function is close to zero: this makes the learning easier for forward layers (the training is faster).

The function is differentiable and monotonic. Derivatives larger than sigmoid → faster

Tanh function also suffers from the vanishing gradient problem.

Linear Activation Function

Linear activation function has a range from -infinity to infinity. It is not a binary activator and it is a continuous function.



$$f(x) = x$$

The derivative is a constant, then changes do not depend on x , only on the input. This means that all the neurons are linear, no matter how many layers we implement.

N layers with all linear functions = single layer ANN

Since the composition of two linear functions is a linear function, then, N layers can be replaced by a single layer.

Selecting an Activation Function

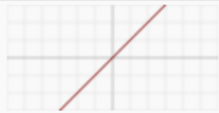







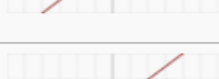
Remember that if we want to train the neural network with backpropagation, the activation function needs to be **continuous** and **differentiable**. Be aware also that the option should depend on the type of problem you are facing on: regression or classification.

It is also important to point out that you can combine different ones:

- In regression or function approximation: since we want to have a real output, it could be convenient to have a linear activation function in the output, and other types in the hidden layer.
- In classification we could choose a sigmoid for the output (it gives a probability value) and tanh function for the hidden layer, since it has several advantages compared to sigmoid
- Remember that for multi-class classification, you cannot choose sigmoid or tanh for the output.

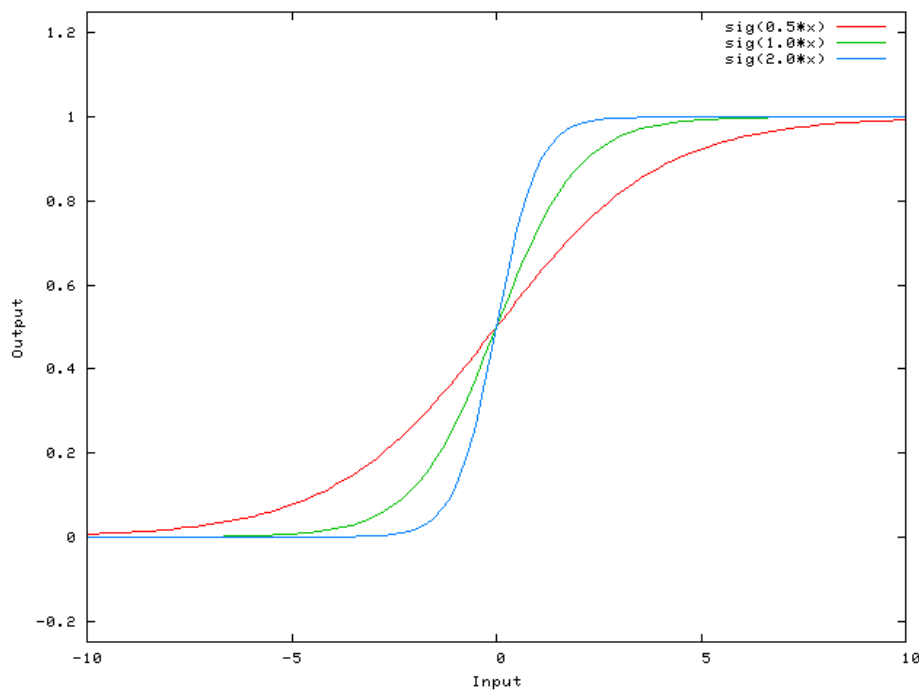
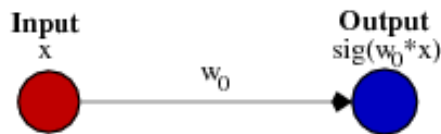
The activation function has a critical impact on the **speed** of the training

Activation Functions

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parametric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

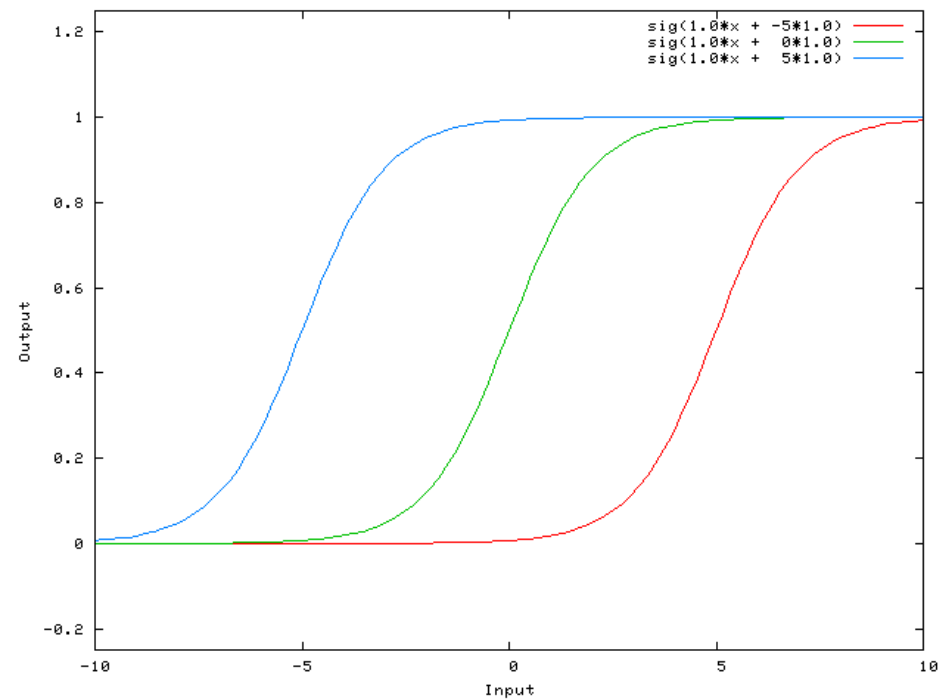
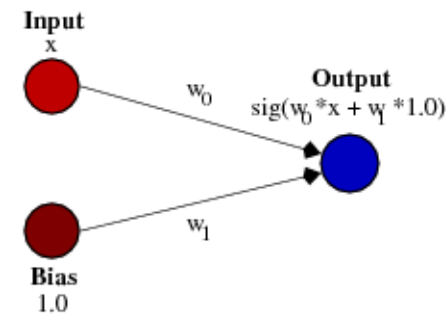
Multilayer Perceptron: the bias

Changes in the function computed for various values of w_0 :



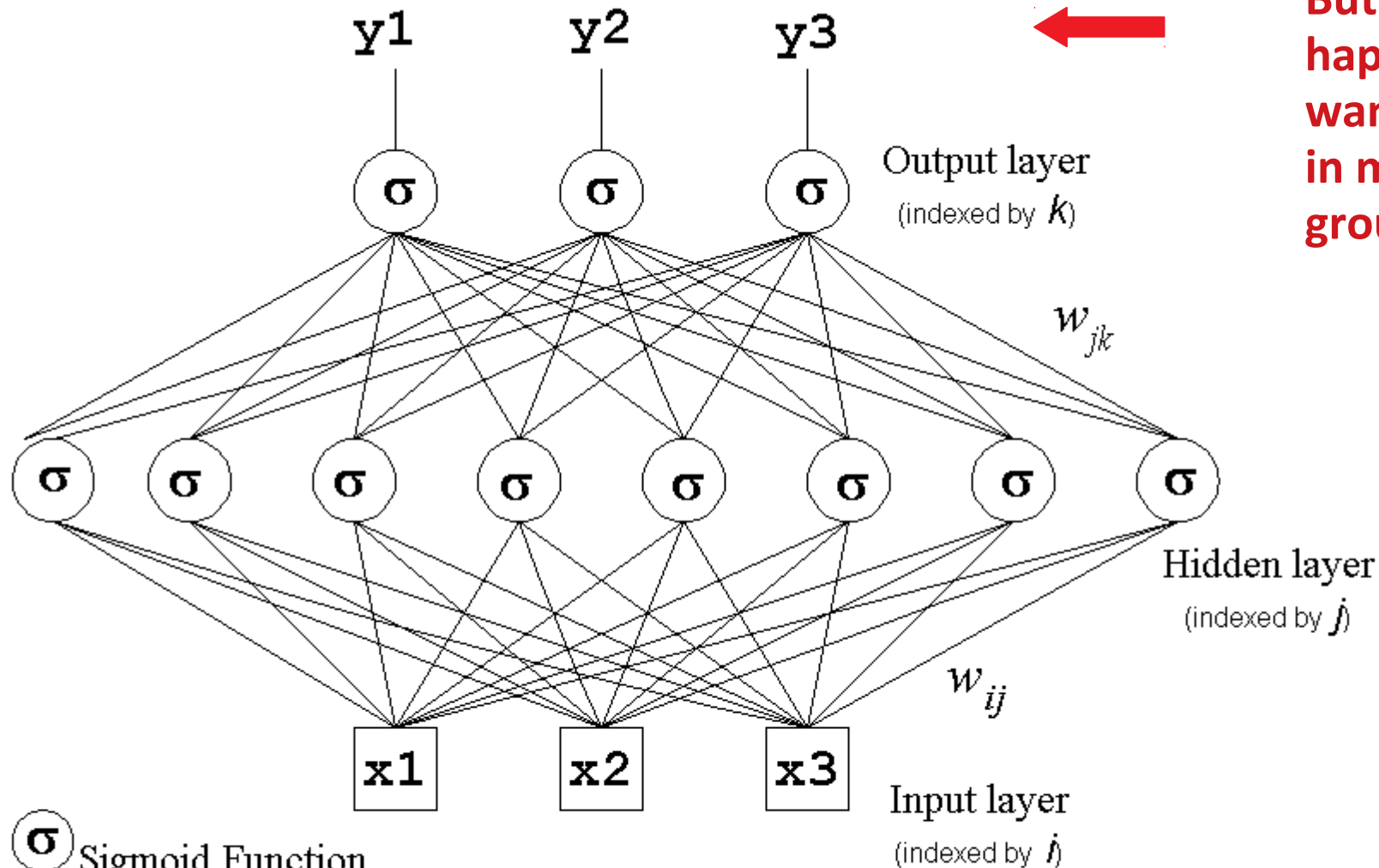
Changes in the smoothness of the sigmoid function

To be able to shift the entire curve:



we add a bias to the network

Multilayer Perceptron



But what happens if we want to classify in more than two groups?

Multilayer Perceptron: Multi-class classification

We are going to predict the **probability** that the output y takes a particular value given the input x , as we did before, but

Then, our predictor would be a conditional probability:

$$\hat{y} = P(y = j | x), j \in \{1, \dots, k\}$$

Subject to:

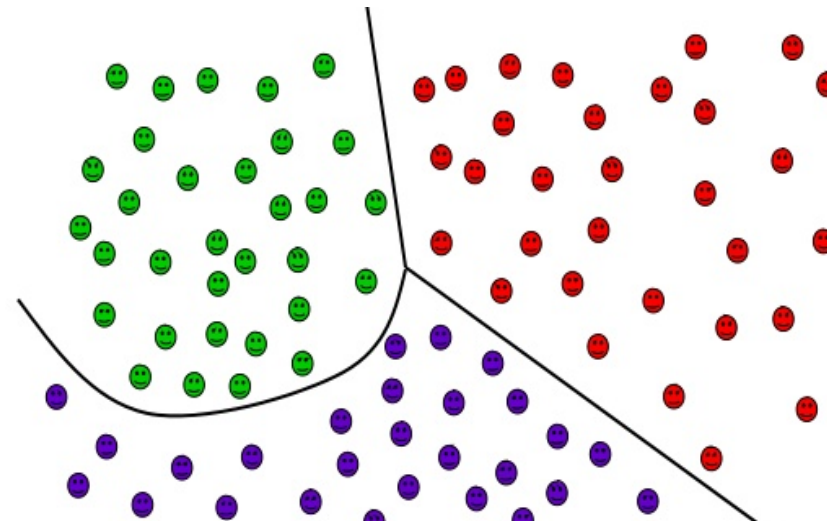
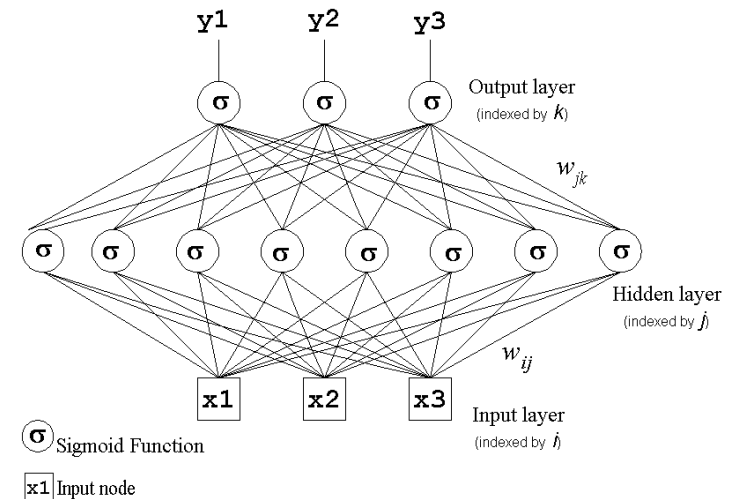
$$\sum P(y = j | x) = 1$$

The probabilities will all sum to 1

For multi-class classification:

- We need multiple outputs (1 per class)
- Sometimes is represented as a one-of-K encoding

$$Y = \underbrace{(0, \dots, 0, 1, 0, \dots, 0)}_{\text{entry } k \text{ is } 1}$$



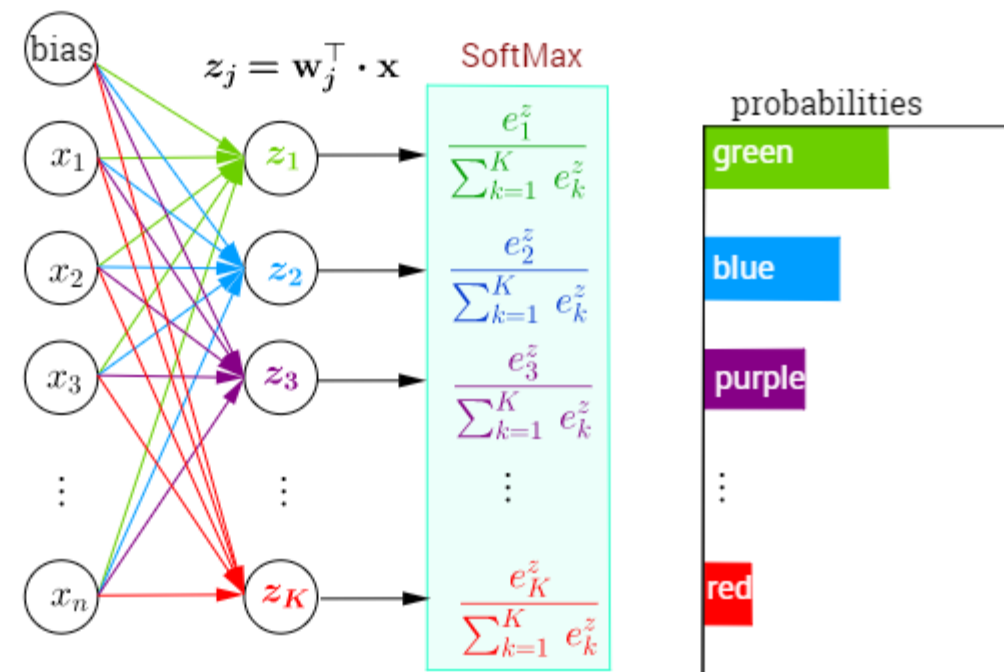
Multilayer Perceptron: Multi-class classification

The **softmax activation function** is the analogue of the sigmoid function for more than two classes.

It assigns decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0. This additional constraint helps training converge more quickly than it otherwise would.

$$\text{softmax}(x)_j = \frac{e^{z_j}}{\sum e^{(z_k)}}$$

- Strictly positive
- We will use it only in the output layer
- Softmax assumes that each example is a member of exactly one class. If a sample can simultaneously be a member of multiple classes, you should not use it.
- Predicted class is the one with the highest estimated probability

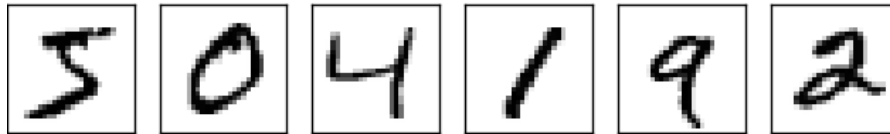


Example: classify handwriting digits

Using the MNIST dataset (1999): 60,000 training and 10,000 testing images

<http://yann.lecun.com/exdb/mnist/>

We want to recognised the following series of numbers:



Four files are available on this site:

train-images-idx3-ubyte.gz: training set images (9912422 bytes)

train-labels-idx1-ubyte.gz: training set labels (28881 bytes)

t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)

t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

How we can design a Multilayer Perceptron to perform this task?

Which elements we need to take into account?

Example: classify handwriting digits

Information:

Input: 28 by 28 pixel images (grey scale 0.0 representing white and 1.0 black)
then,

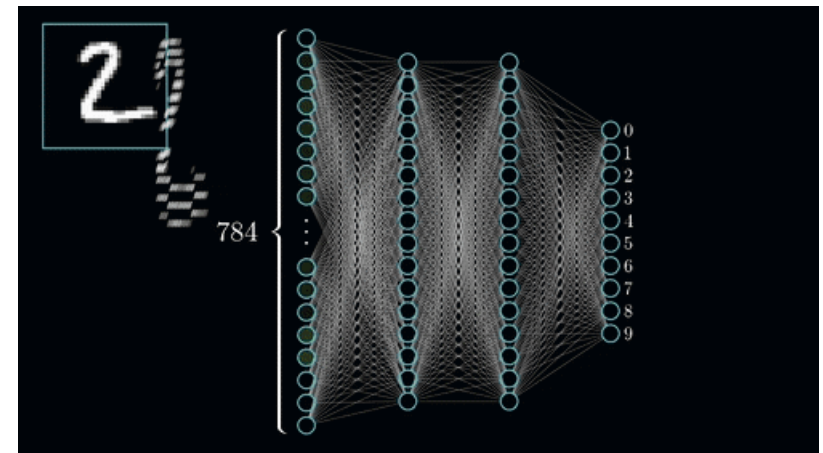
$$x_i \in \mathbb{R}^{784}$$

Output: 4 binary representation
output

$$2^4 = 16$$

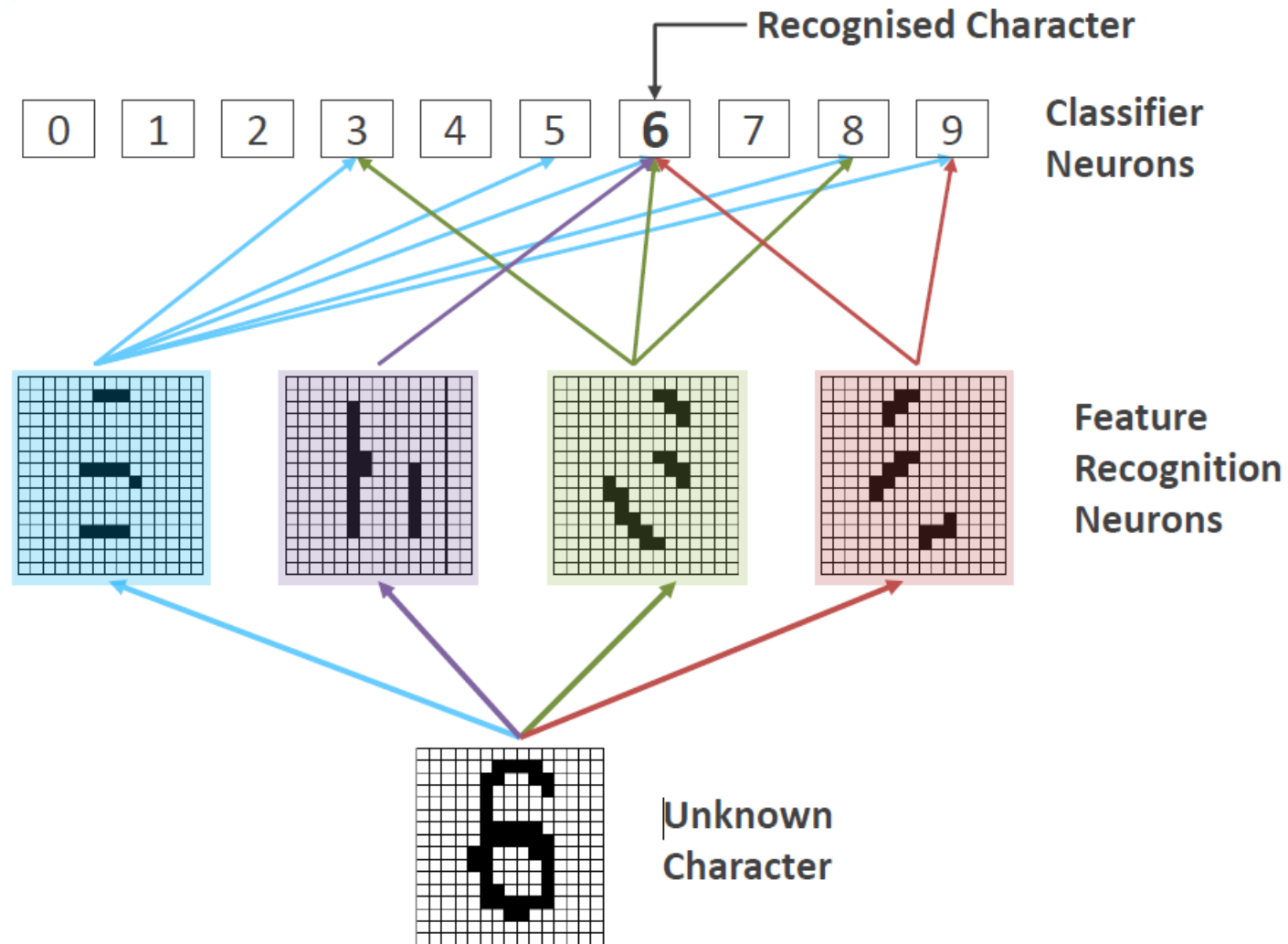
$$y_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Example of **multi-class classification** task



Example: classify handwriting digits

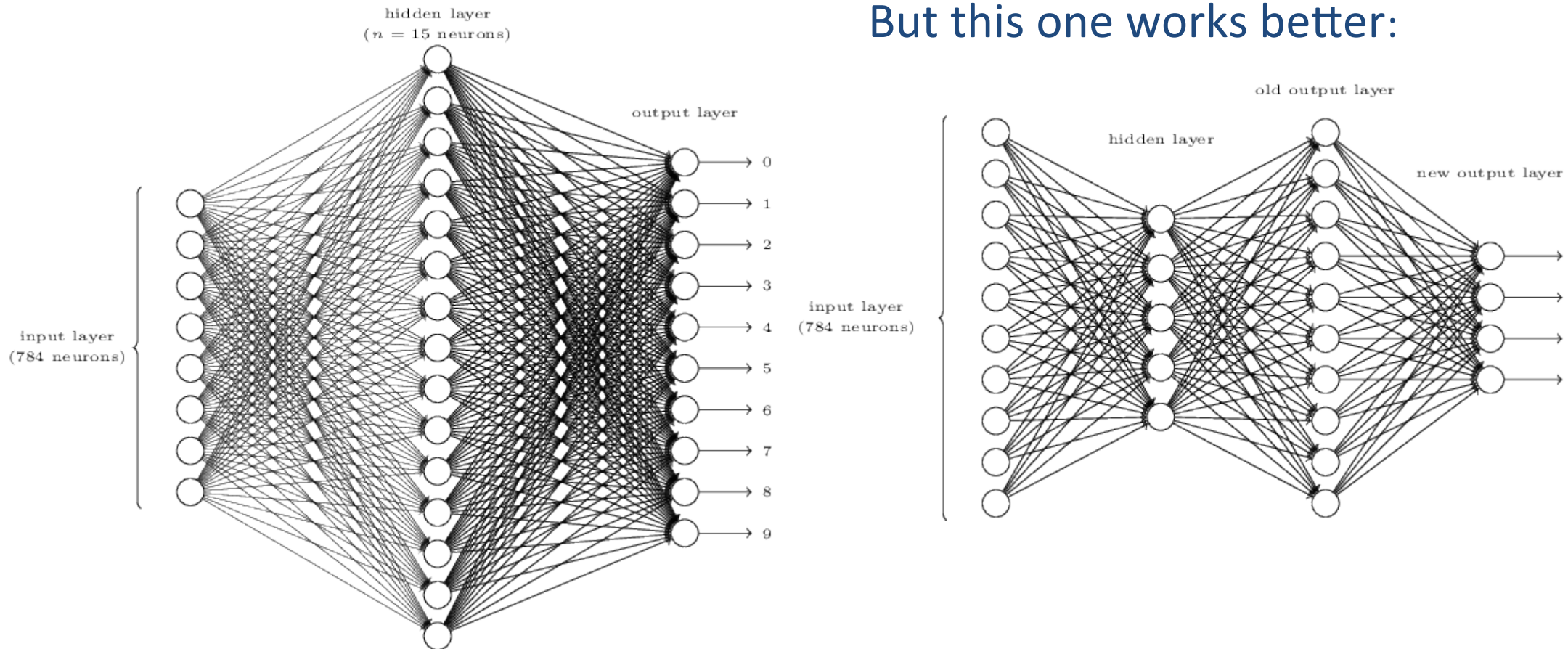
Simplification of the values within the network:



Example: classify handwriting digits

Topology: To recognise individual digits we will use a two-layer neural network.

But this one works better:

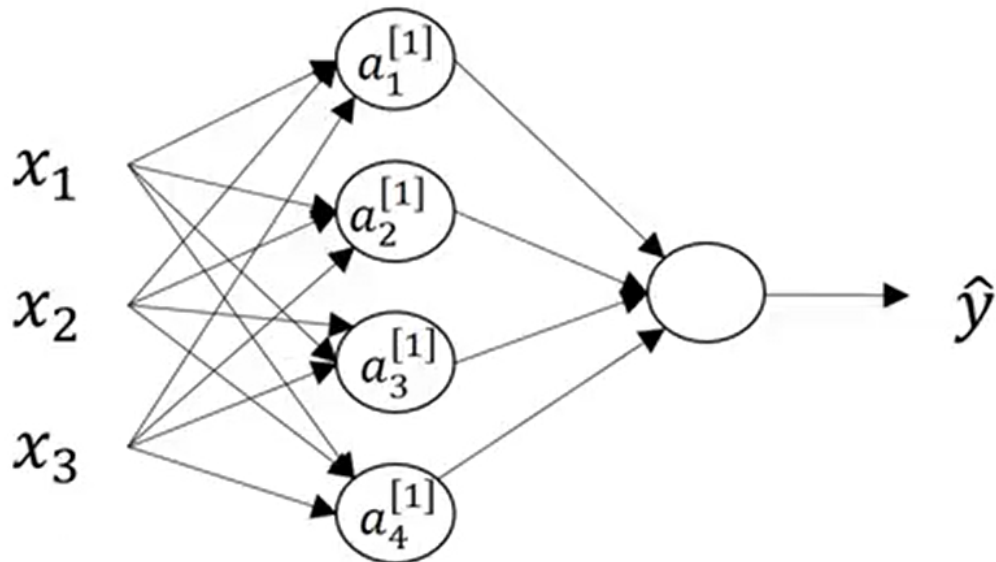
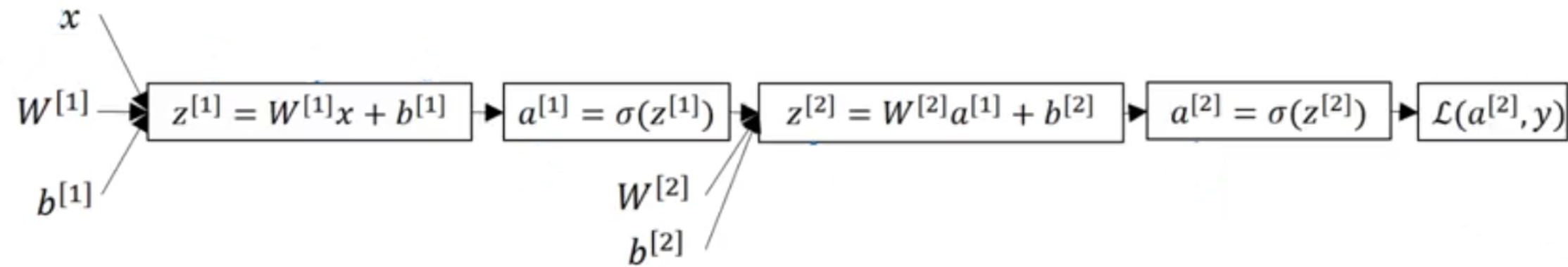


<https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html>

In 2006 DNN broke Support Vector Machine supremacy for this dataset.

Multilayer Perceptron

The multilayer perceptron is an extension of our previous neural model.



Changes/additions in nomenclature:

$[]$ – Number of Layer

$()$ – Number of Sample

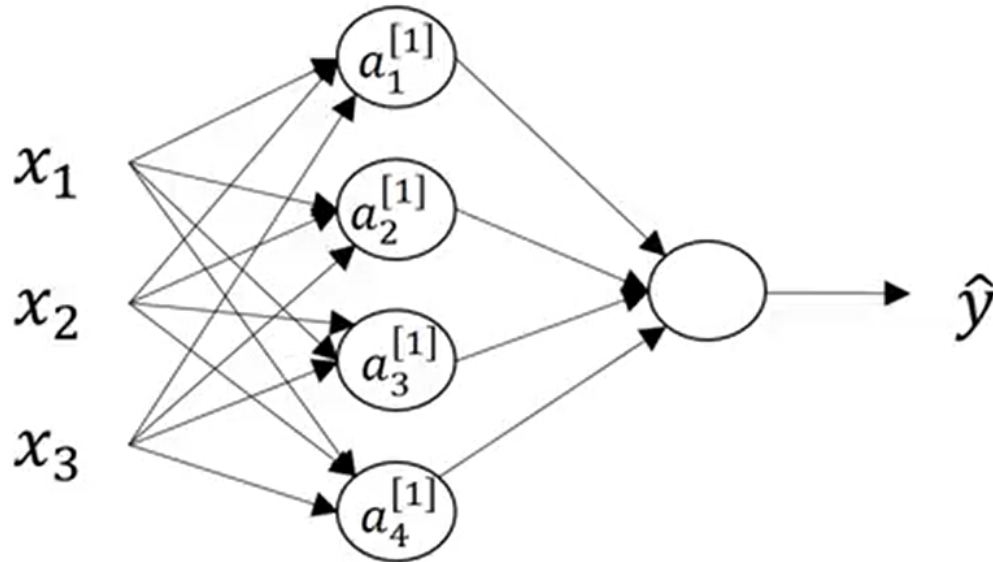
$a_i^{[l]}$ layer
node in the layer

$$\hat{y} = a \rightarrow \hat{y} = a^{[2]}$$

where [2] is the number of the last layer 24

Multilayer Perceptron

For a single training example:



For the first layer:

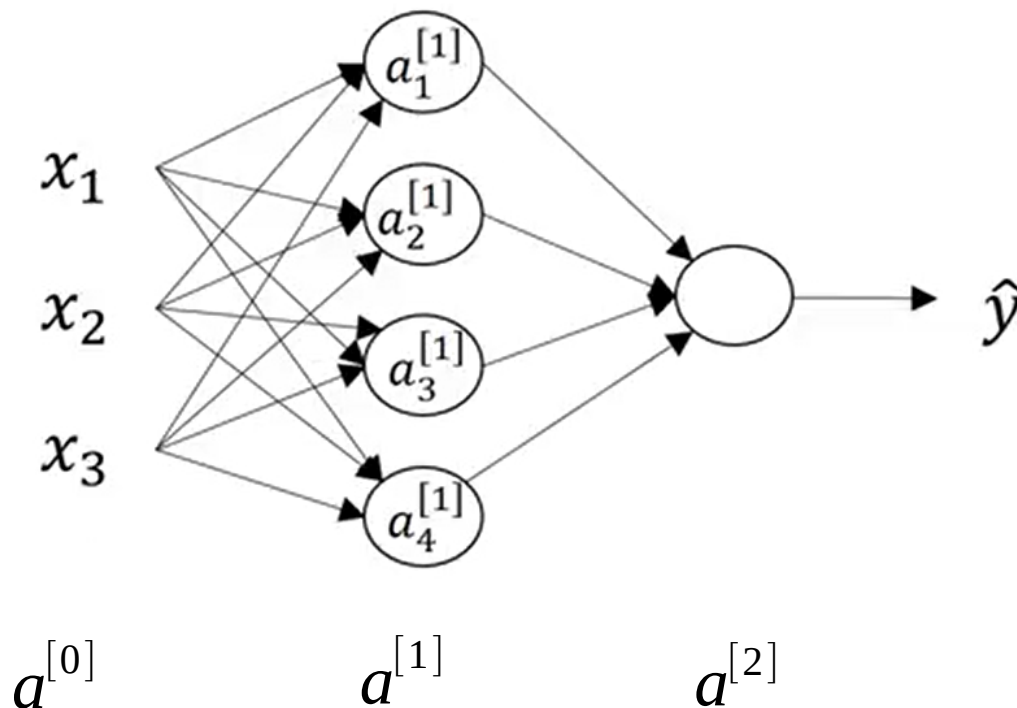
$$\begin{aligned} z_1^{[1]} &= w_1^{[1]T} x + b_1^{[1]}, & a_1^{[1]} &= \sigma(z_1^{[1]}) \\ z_2^{[1]} &= w_2^{[1]T} x + b_2^{[1]}, & a_2^{[1]} &= \sigma(z_2^{[1]}) \\ z_3^{[1]} &= w_3^{[1]T} x + b_3^{[1]}, & a_3^{[1]} &= \sigma(z_3^{[1]}) \\ z_4^{[1]} &= w_4^{[1]T} x + b_4^{[1]}, & a_4^{[1]} &= \sigma(z_4^{[1]}) \end{aligned}$$

Vectorisation of this first layer:

$$z^{[1]} = \begin{bmatrix} - & - & w_1^{[1]} & - & - \\ - & - & w_2^{[1]} & - & - \\ - & - & w_3^{[1]} & - & - \\ - & - & w_4^{[1]} & - & - \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]}x + b_1^{[1]} \\ w_2^{[1]}x + b_2^{[1]} \\ w_3^{[1]}x + b_3^{[1]} \\ w_4^{[1]}x + b_4^{[1]} \end{bmatrix} = \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} \quad a^{[1]} = \sigma(z^{[1]})$$

Multilayer Perceptron

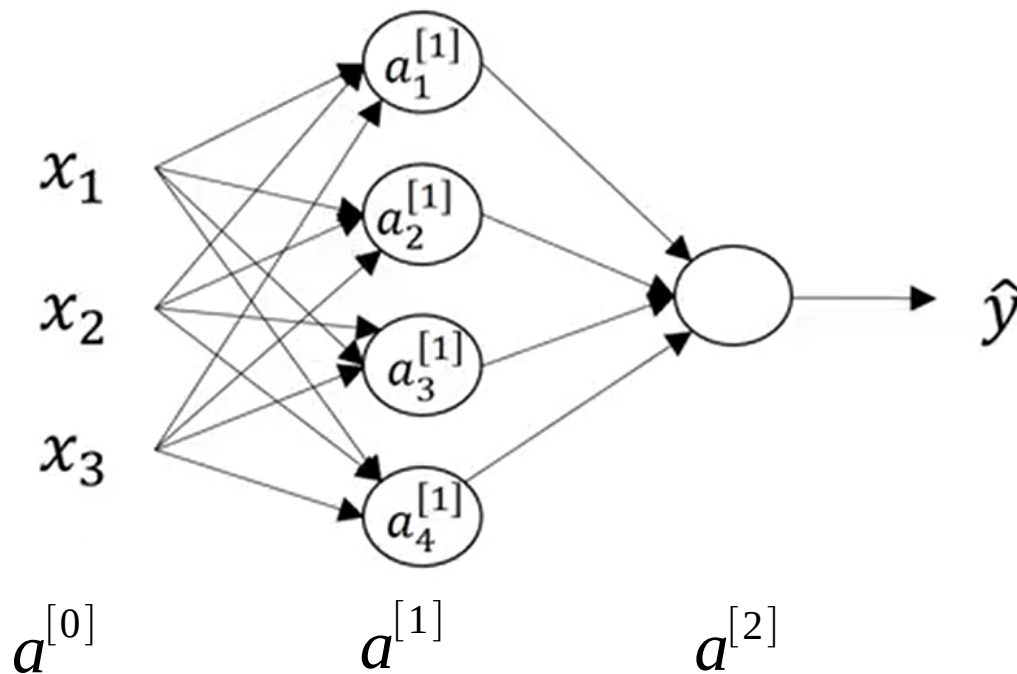
For the entire network (first and second layer) using only one training sample:



$$\begin{aligned}z^{[1]} &= w^{[1]T} a^{[0]} + b^{[1]} \\a^{[1]} &= \sigma(z^{[1]}) \\z^{[2]} &= w^{[2]T} a^{[1]} + b^{[2]} \\a^{[2]} &= \sigma(z^{[2]})\end{aligned}$$

Multilayer Perceptron

For the entire network and the entire training set without vectorisation:



for $i=1$ to m

$$z^{[1]}(i) = w^{[1]T} a^{[0]}(i) + b^{[1]}$$

$$a^{[1]}(i) = \sigma(z^{[1]}(i))$$

$$z^{[2]}(i) = w^{[2]T} a^{[1]}(i) + b^{[2]}$$

$$a^{[2]}(i) = \sigma(z^{[2]}(i))$$

end for

We need to add the following information:

$a^{[0]}(i)$ Layer 0, training sample i

Multilayer Perceptron

$$Z^{[1]} = w^{[1]T} A^{[0]} + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = w^{[2]T} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$

nx: number of hidden units

m: number of samples

$$X = \begin{bmatrix} \vdots & \vdots & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

(nx, m)

$$Z^{[1]} = \begin{bmatrix} \vdots & \vdots & \vdots \\ z^{1} & z^{[1](2)} & \dots & z^{[1](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

(nx, m)

$$Z^{[2]} = \begin{bmatrix} \vdots & \vdots & \vdots \\ z^{[2](1)} & z^{2} & \dots & z^{[2](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

(nx, m)

$$A^{[1]} = \begin{bmatrix} \vdots & \vdots & \vdots \\ a^{1} & a^{[1](2)} & \dots & a^{[1](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

(nx, m)

$$A^{[2]} = \begin{bmatrix} \vdots & \vdots & \vdots \\ a^{[2](1)} & a^{2} & \dots & a^{[2](m)} \\ \vdots & \vdots & \vdots \end{bmatrix}$$

(nx, m)

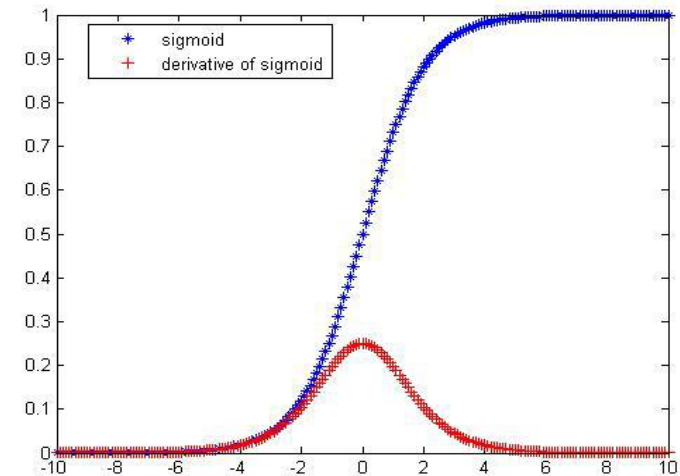
Multilayer Perceptron: gradients

The training algorithm for MLP requires **differentiable**, continuous, nonlinear activation functions

Sigmoid function

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

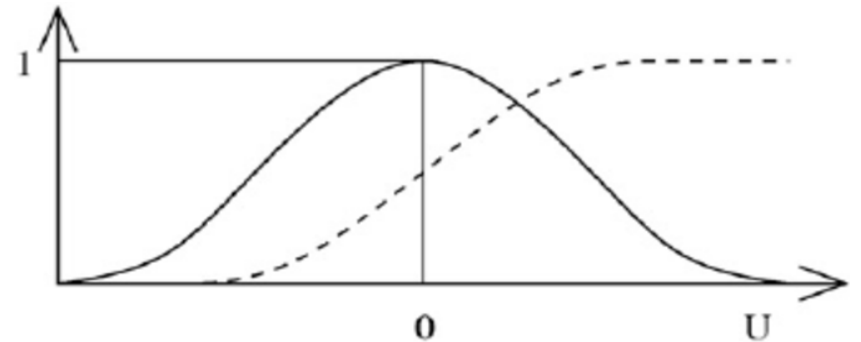
$$\begin{aligned}\frac{d\sigma(z)}{dz} &= \frac{d}{dz} \frac{1}{1+e^{-z}} \\ &= \frac{1}{(1+e^{-z})^2} (e^{-z}) \\ &= \frac{1}{(1+e^{-z})} \cdot \left(1 - \frac{1}{(1+e^{-z})}\right) \\ &= \sigma(z)(1-\sigma(z))\end{aligned}$$



Hyperbolic tangent

$$\tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh'(z) = \frac{d}{dz} \frac{\sinh(z)}{\cosh(z)} = 1 - \tanh^2(z)$$




Backpropagation Algorithm

Weights initialisation:

- If we initialise all the weights to the same value, e.g zero, then no matter how long the network is trained, by symmetry, each hidden unit will compute the same function. We need to use a random initialisation.
- The bias has no problems of symmetry. It can be initialised to zero.
- Values should be small in size to avoid problems if we use any type of activation function that slow down the learning for large values, like sigmoid or tanh

General algorithm is the same than in logistic regression:



```
define the network structure
initialise the model parameters
loop
    compute predictor (forward propagation)
    compute the loss
    compute gradients
    update learning rules (gradient descent)
until convergence
```


Backpropagation Algorithm

Parameters:

$$W_{(n^{[1]}, n^{[0]})}^{[1]}, b_{(n^{[1]}, 1)}^{[1]}, W_{(n^{[2]}, n^{[1]})}^{[2]}, b_{(n^{[2]}, 1)}^{[2]}$$

$n^{[0]}$: input units
 $n^{[1]}$: hidden units
 $n^{[2]}$: output units

Cost Function:

$$J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}, y)$$

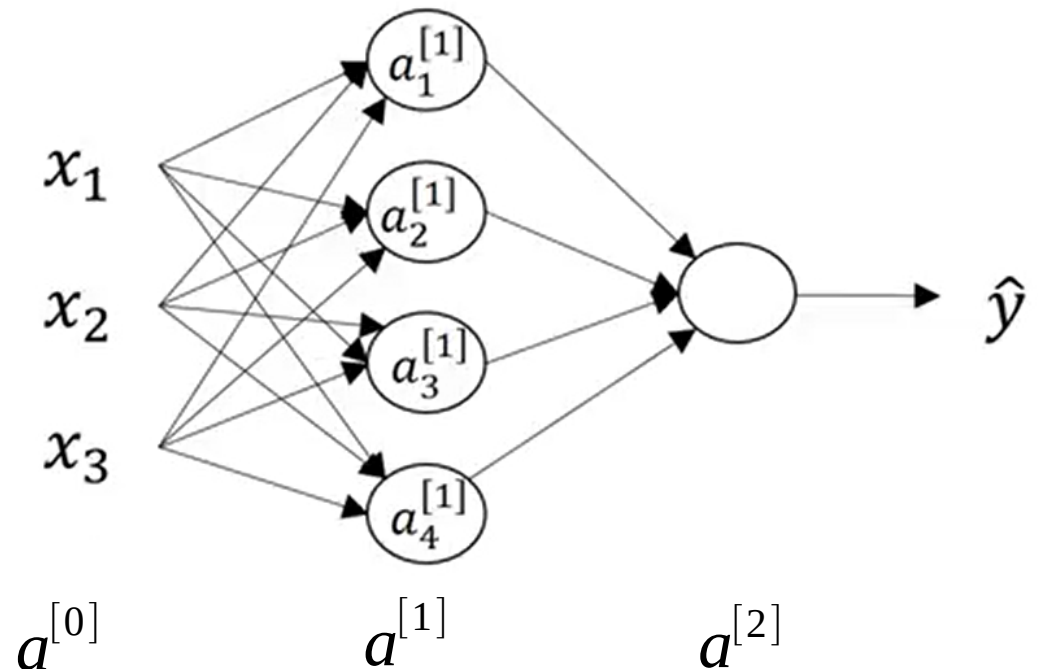
Forward Propagation:

$$Z^{[1]} = w^{[1]T} A^{[0]} + b^{[1]}$$

$$A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = w^{[2]T} A^{[1]} + b^{[2]}$$

$$A^{[2]} = \sigma(Z^{[2]})$$



Backpropagation Algorithm

Back Propagation:

$$dZ^{[2]} = A^{[2]} - Y$$

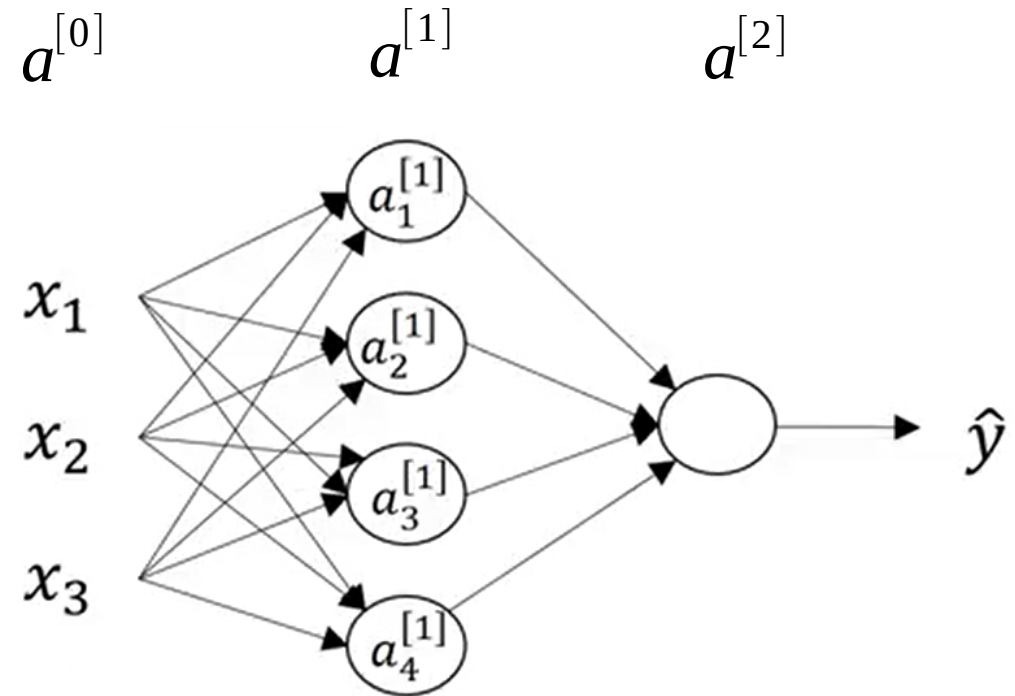
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$

$$db^{[2]} = \frac{1}{m} dZ^{[2]}$$

$$\underline{dZ^{[1]} = W^{[2]} dZ^{[2]} * g^{[1]'}(Z^{[1]})}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$

$$db^{[1]} = \frac{1}{m} dZ^{[1]}$$



$$dW^{[1]} \quad dZ^{[1]}$$

$$dW^{[2]} \quad dZ^{[2]}$$

as we did for
logistic regression

Learning Rate in a MLP

Not a trivial problem.

Low value – slow learning rate (risk of stagnation)

High value – risk of oscillations (could be no learning at all)

More critical in online learning

An ongoing line of research

Different approaches to overcome this risk:

Second order gradients. Computational expensive

Optimal algorithms for adaptive networks: Second order back propagation, second order direct propagation, and second order Hebbian learning. DB Parker 1987

Heuristics

Momentum

Learning rate variation methods

Learning Rate in a MLP

Momentum

How important previous update is in calculating current update

$$w_i = w_i - \alpha \frac{\partial J(w, b)}{\partial w} + \underbrace{\eta * \Delta w_{i-1}}_{\text{momentum}}, \quad 0 \leq \eta < 1$$

Momentum constant

typically $\eta = 0.9$

Learning rate variation methods

LR Decay Make smaller steps near optimum

$$\alpha(n+1) = k \alpha(n), \quad 0 < k < 1$$

Typically apply it each 10-100 iterations with $k = 0.1$

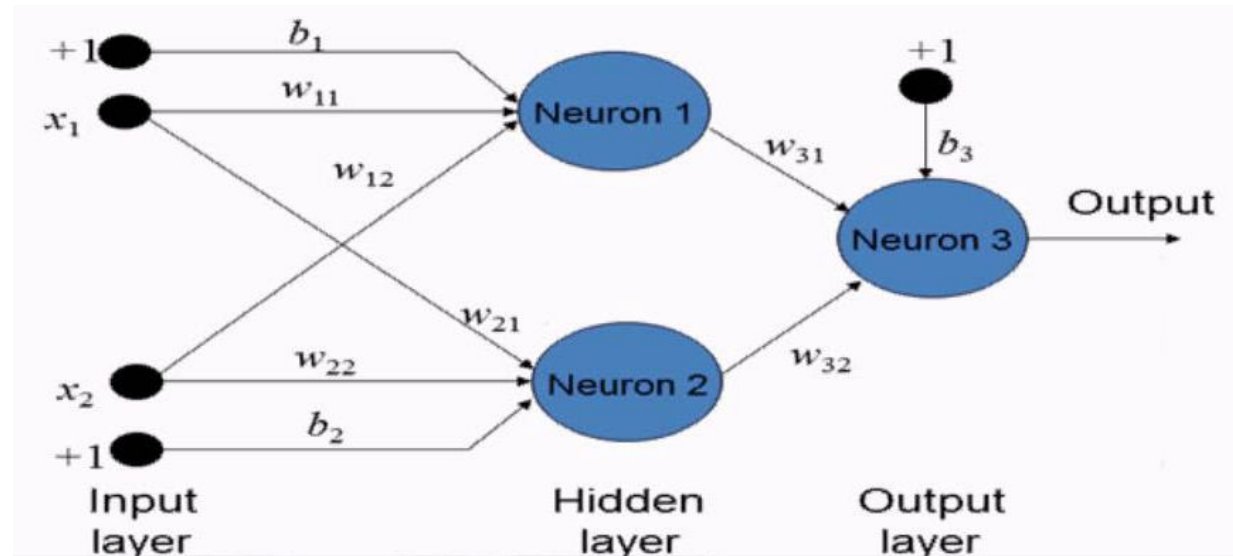
Exponential Decay Exponential rate of change of the learning rate as a function of the number of iteration

$$\alpha(n+1) = \alpha(n) e^{-kt}, \quad 0 < k < 1$$

...and much more: Adam, Adagrad, Adamax, Adadelata

Example of training a MLP

In a MLP with architecture 2-2-1, we are going to do a complete forward and backward sweep



We assume that the activation function in all the three neurons is a sigmoid function:

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

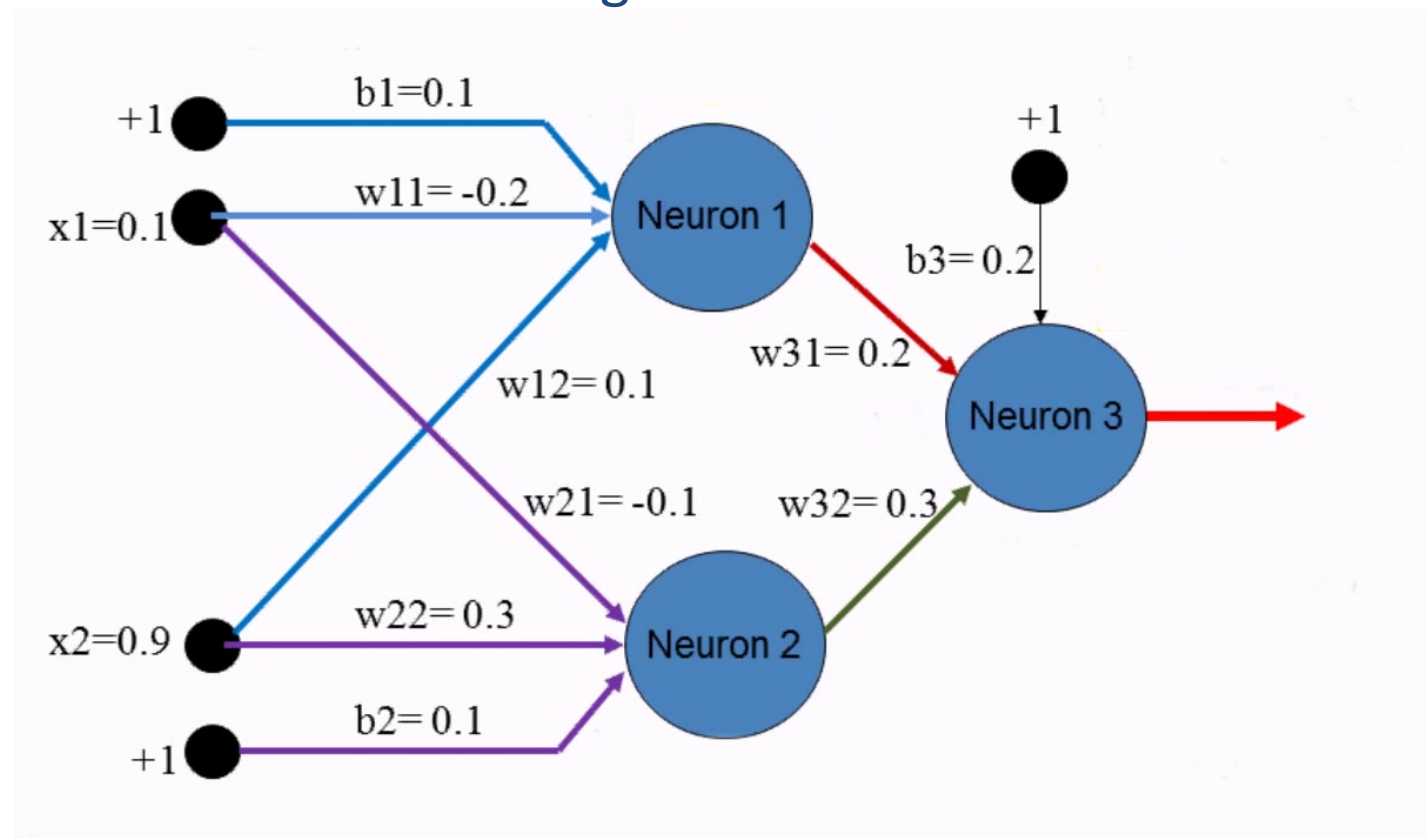
We assume also:

- Output value: 0.9
- Input values: $x_1=0.1$ and $x_2=0.9$
- Learning rate $\alpha=0.25$

Example of training a MLP

- 1.- Forward Pass
- 2.- Backward Pass (training)

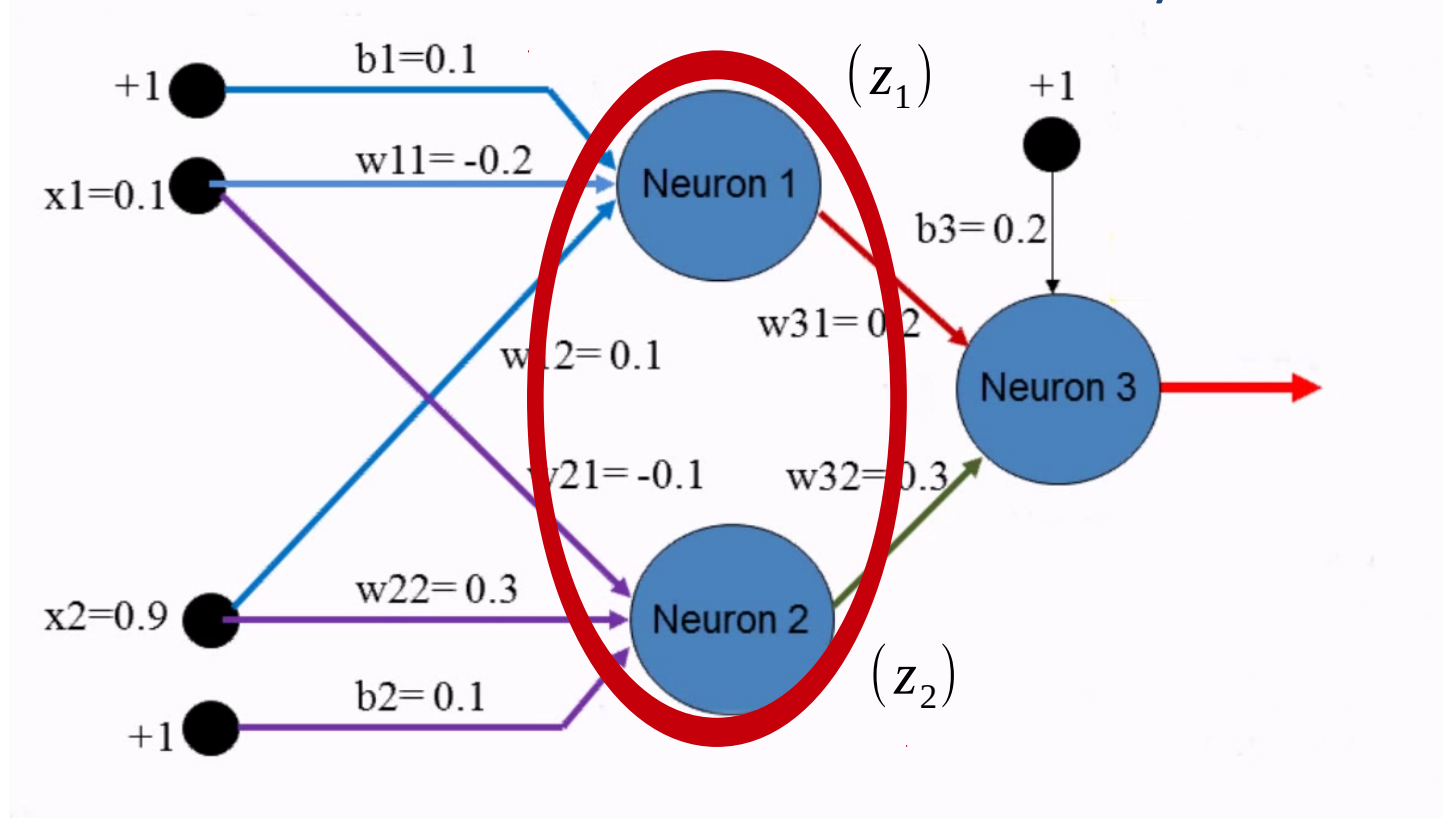
Random initialisation of the weights



Example of training a MLP

Forward Pass

Calculate the values of the units of the hidden layer: Neuron 1 and Neuron 2



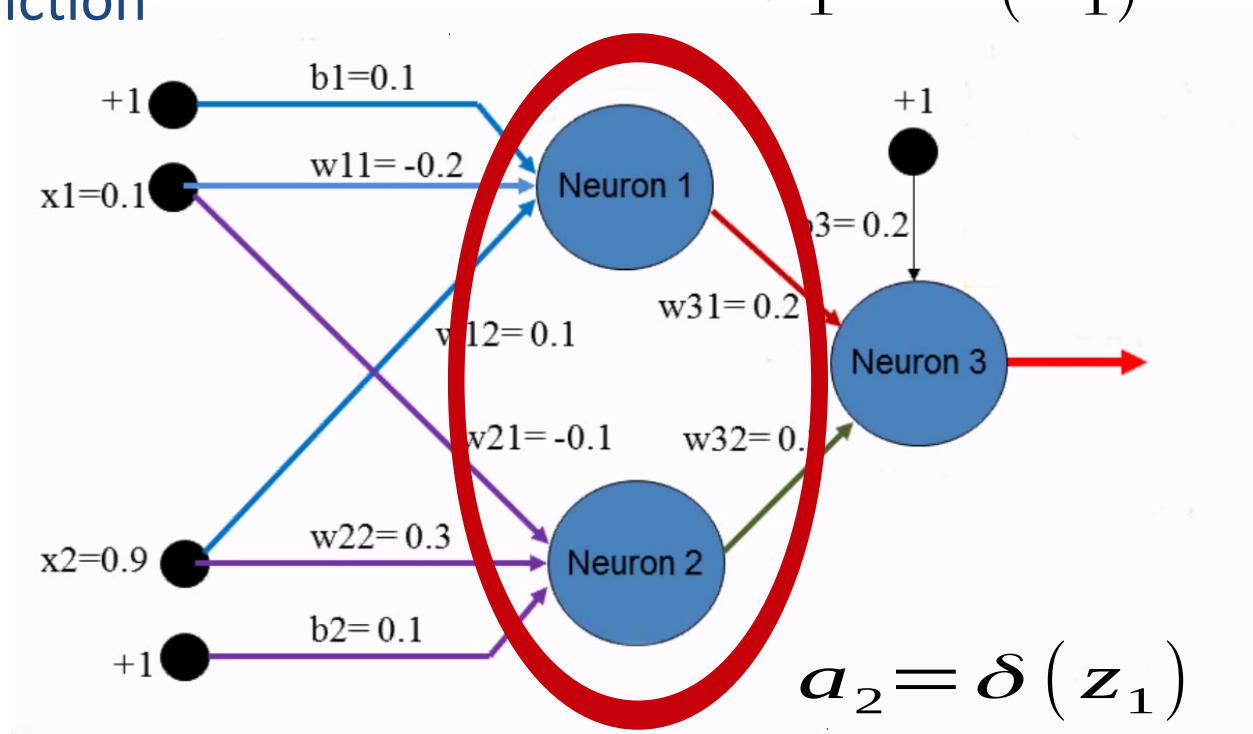
$$z_1 = 1 * 0.1 + 0.1 * (-0.2) + 0.9 * 0.1 = 0.17$$

$$z_2 = 1 * 0.1 + 0.1 * (-0.1) + 0.9 * 0.3 = 0.36$$

Example of training a MLP

Forward Pass

Activation function



$$z_1 = 0.17$$

$$z_2 = 0.36$$

$$a_1 = \sigma(0.17) = \frac{1}{1 + e^{(-0.17)}} = 0.542$$

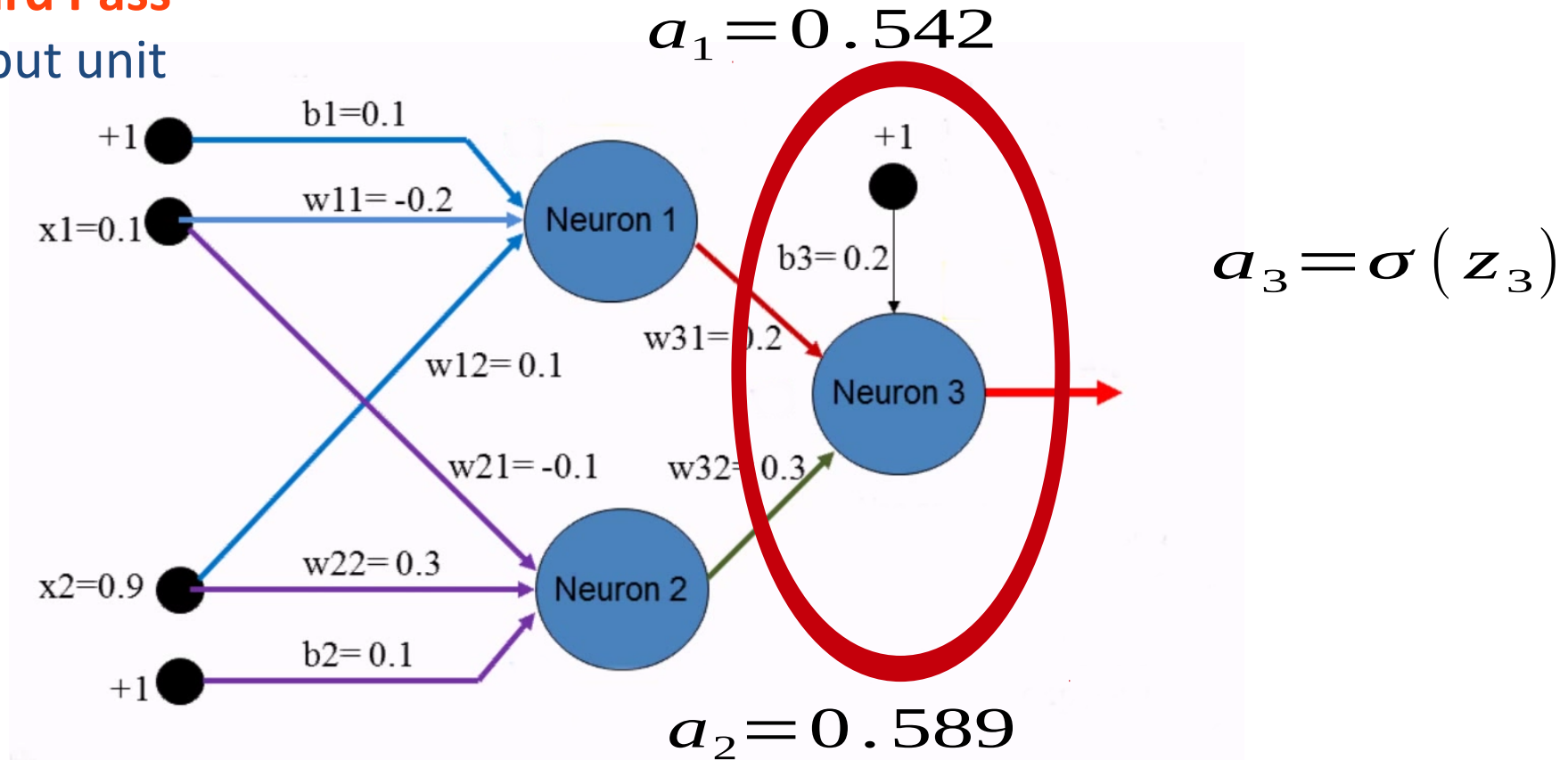
$$a_2 = \sigma(0.36) = \frac{1}{1 + e^{(-0.36)}} = 0.589$$

$$e = 0.281$$

Example of training a MLP

Forward Pass

Output unit



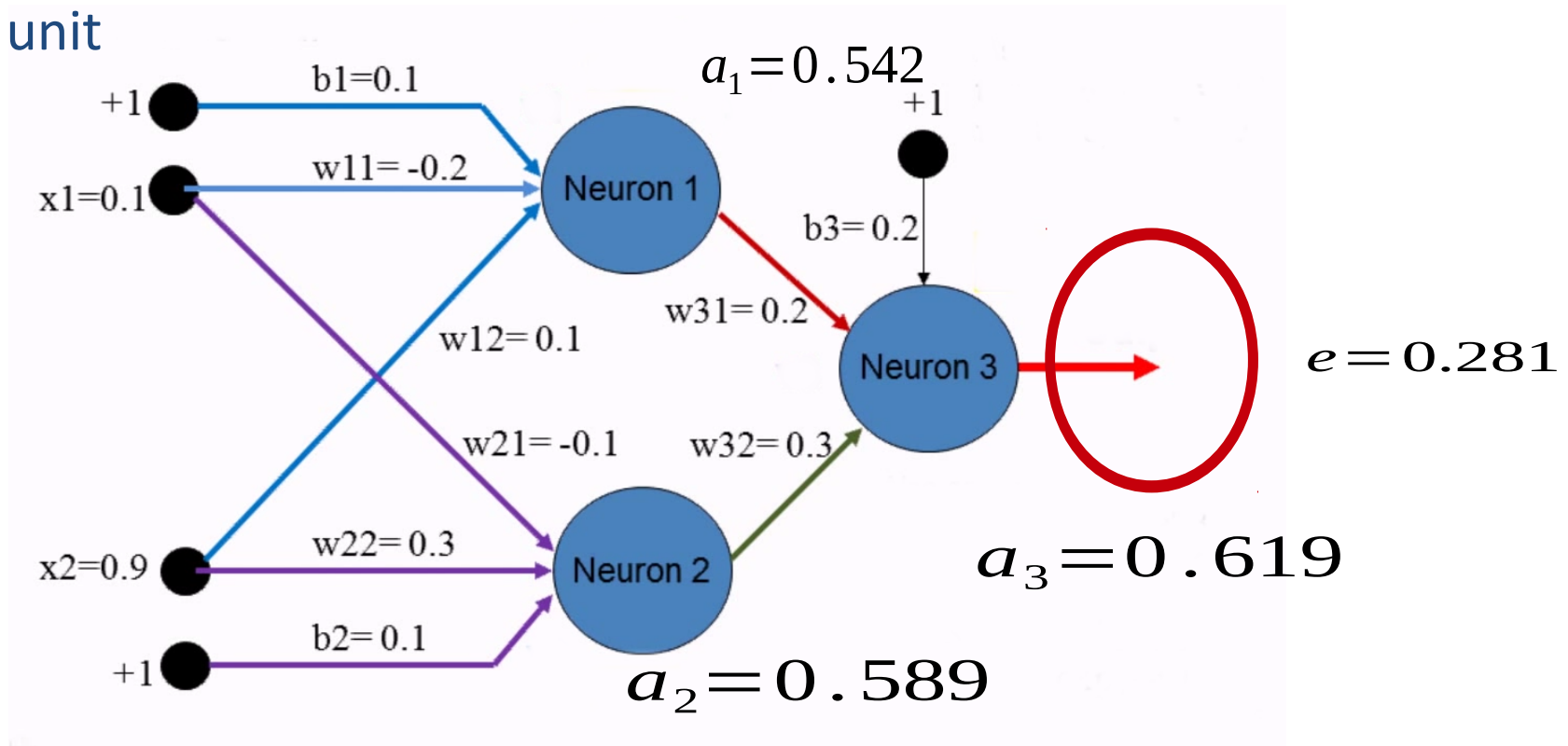
$$z_3 = 1 * 0.2 + 0.542 * 0.2 + 0.589 * 0.3 = 0.485$$

$$a_3 = \sigma(0.485) = \frac{1}{1 + e^{(-0.485)}} = 0.619$$

Example of training a MLP

Error Calculation

Output unit

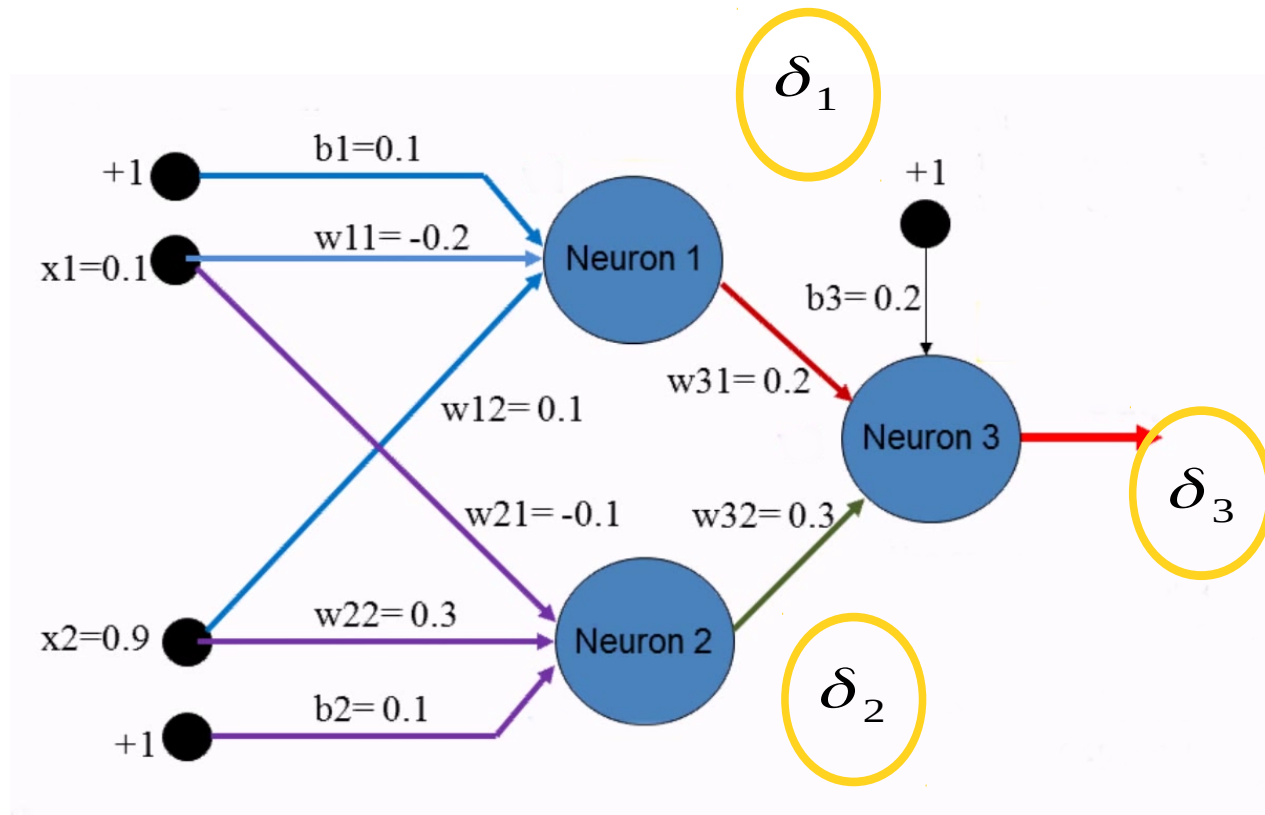


$$e = y - a_3$$

$$e = 0.9 - 0.619 = 0.281$$

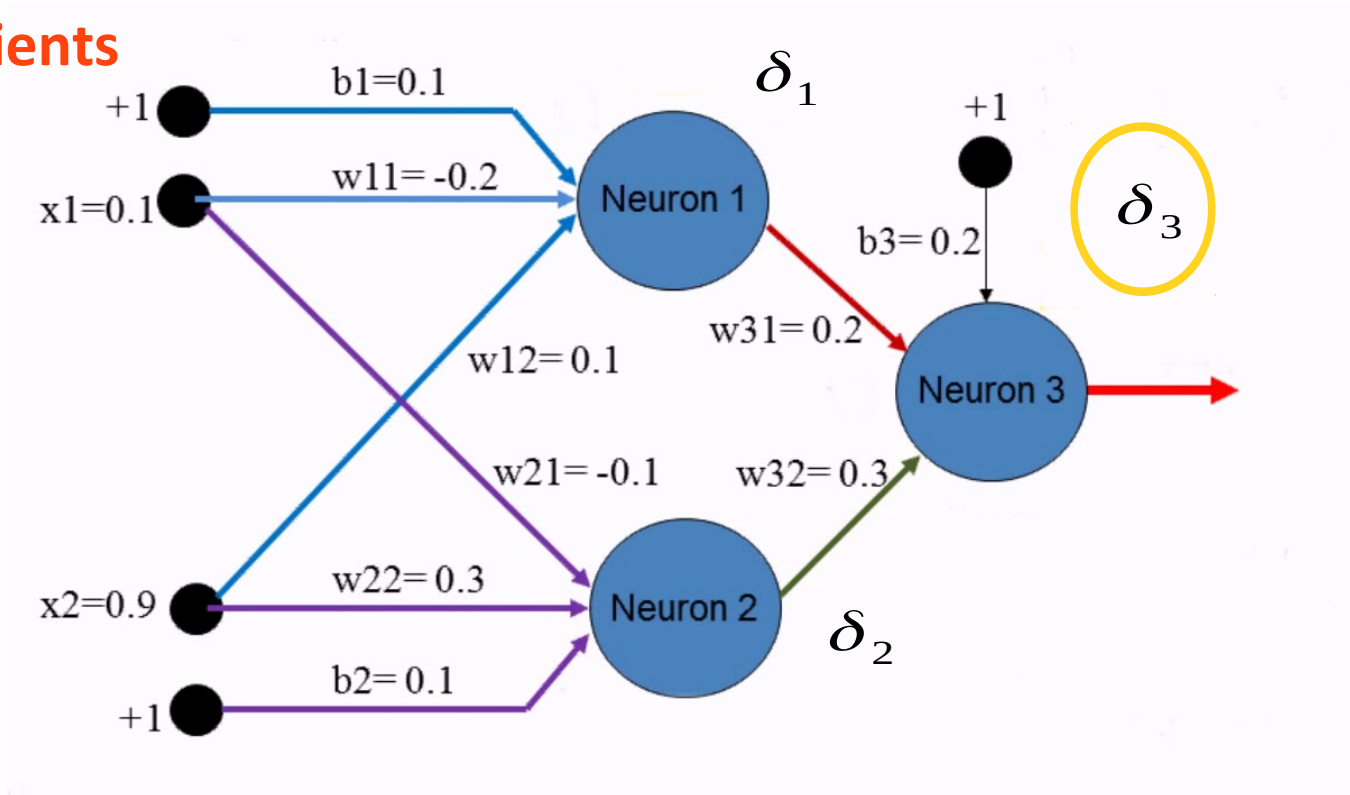
Example of training a MLP

Local Gradients



Example of training a MLP

Local Gradients

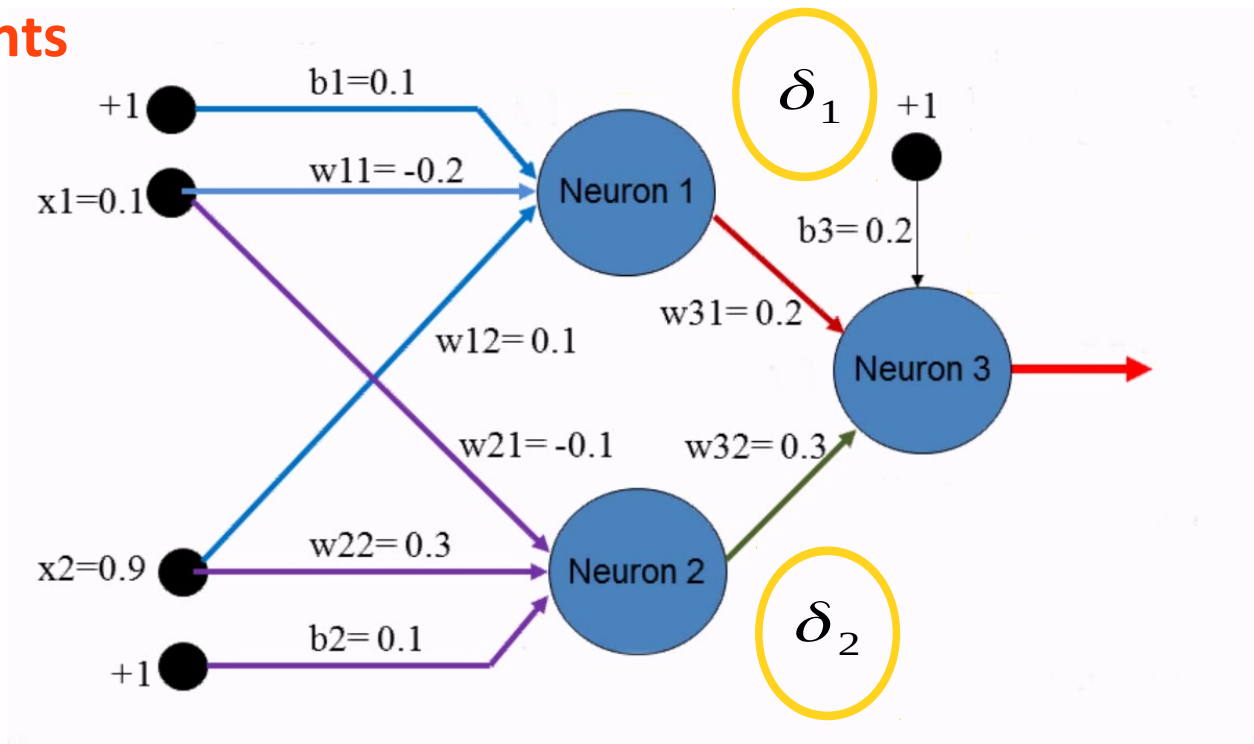


$$e = 0.281$$

$$\begin{aligned}\delta_3 &= \sigma'(z_3) * e = \sigma'(0.485) * 0.281 = \sigma(0.485) [1 - \sigma(0.485)] * 0.281 \\ &= 0.619 (1 - 0.619) * 0.281 = 0.0663\end{aligned}$$

Example of training a MLP

Local Gradients



$$\delta_3 = 0.0663$$

$$\delta_1 = \sigma'(z_1) * (\delta_3 * w_{31}) = \sigma'(0.17) * (0.0663 * 0.2)$$

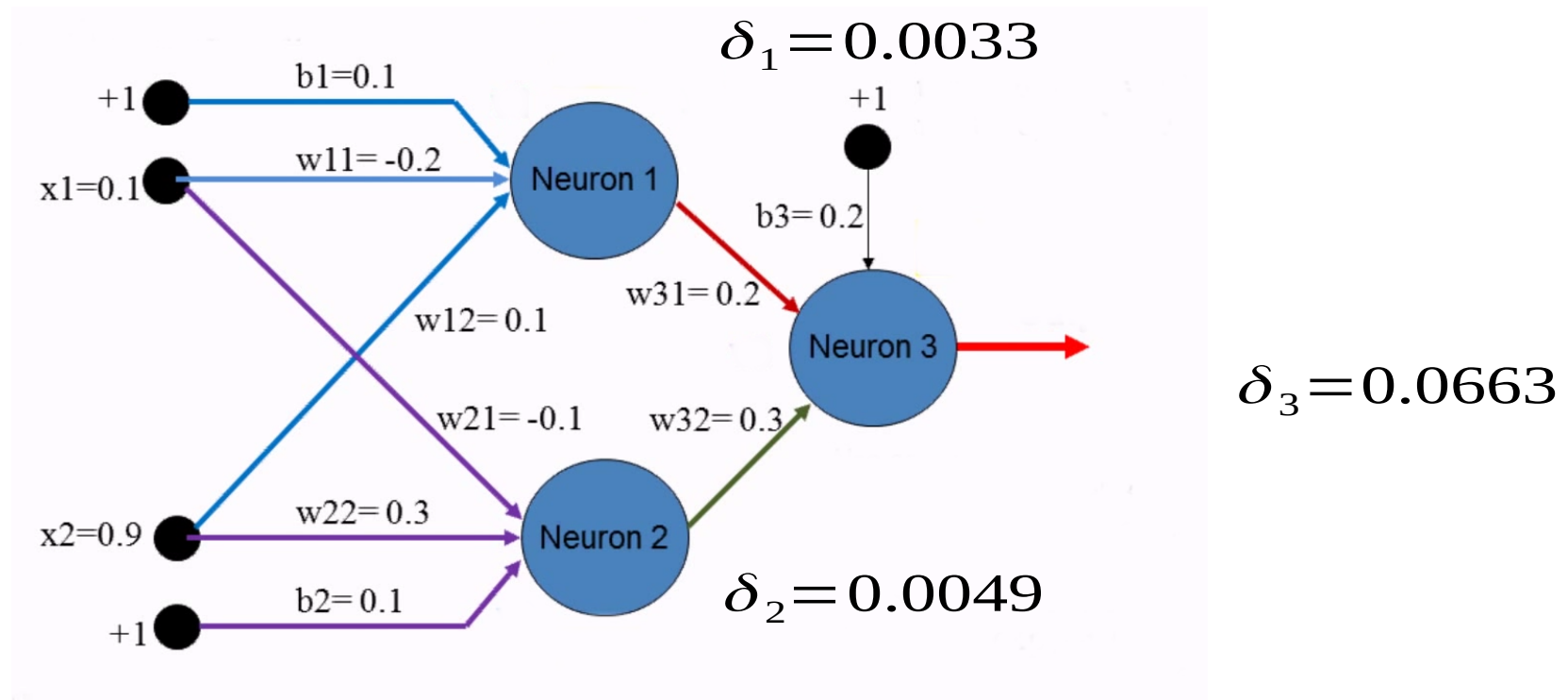
$$= \sigma(0.17) [1 - \sigma(0.17)] * 0.01362 = 0.542 [1 - 0.542] * 0.01362 = 0.0033$$

$$\delta_2 = \sigma'(z_2) * (\delta_3 * w_{32}) = \sigma'(0.36) * (0.0663 * 0.3)$$

$$= \sigma(0.36) [1 - \sigma(0.36)] * 0.01989 = 0.589 [1 - 0.589] * 0.01989 = 0.0049$$

Example of training a MLP

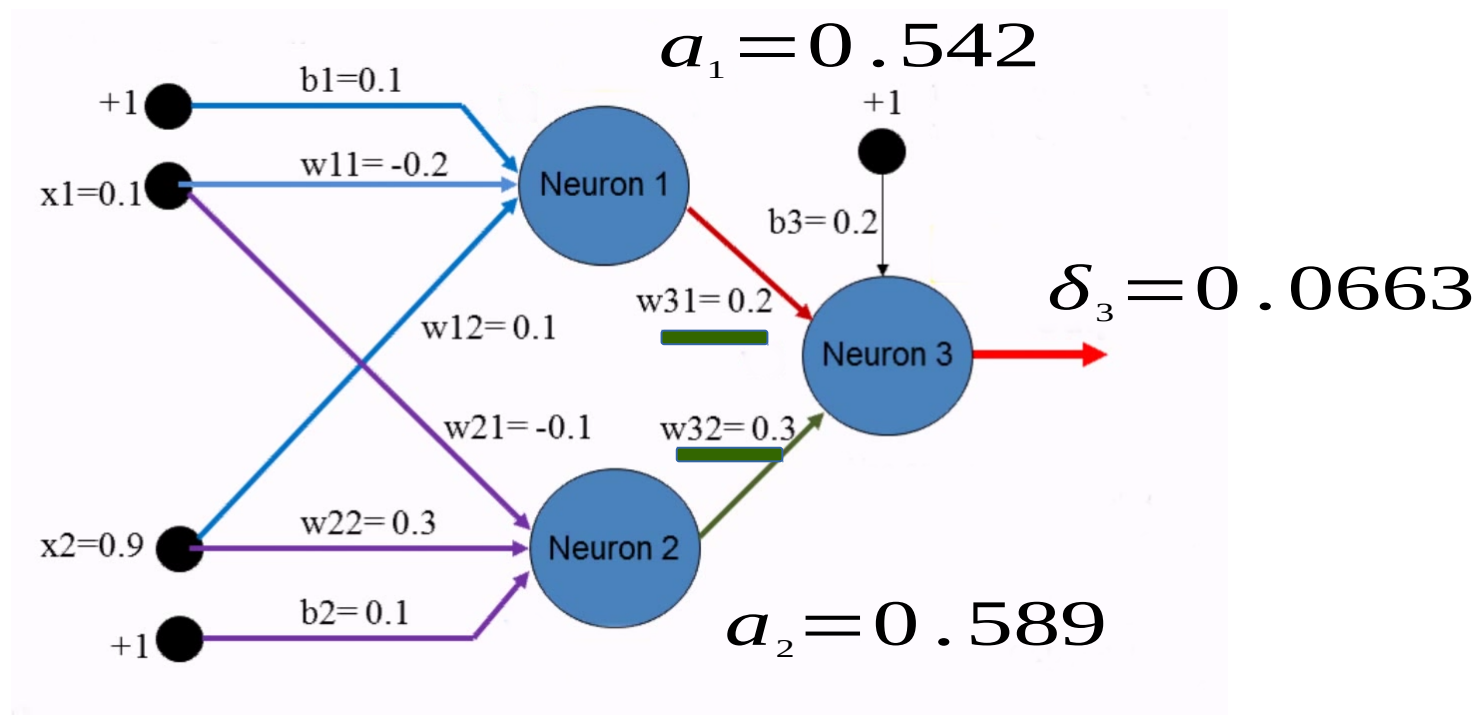
Adjustments of weights



$$w = w - \alpha * \delta * a$$

Example of training a MLP

Adjustments of weights



$$w_{31} = w_{31} - \alpha * \delta_3 * a_1 = 0.2 - 0.25 * 0.0663 * 0.542$$

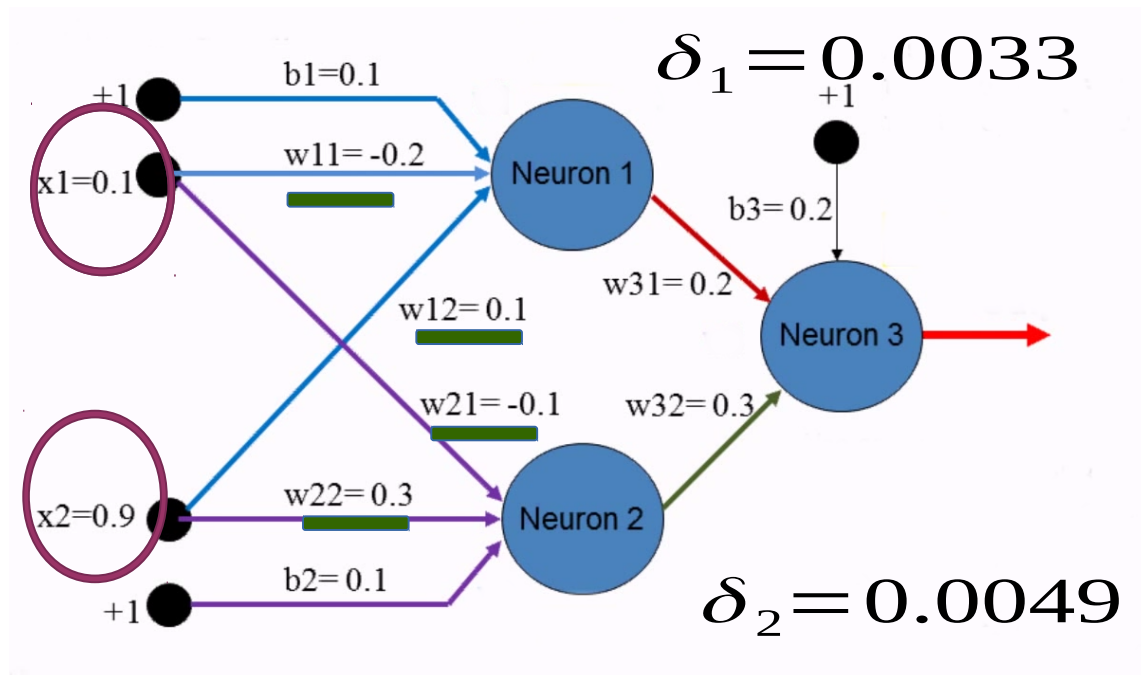
$$= 0.2 - 0.0089 = 0.1911$$

$$w_{32} = w_{32} - \alpha * \delta_3 * a_2 = 0.3 - 0.25 * 0.0663 * 0.589$$

$$= 0.3 - 0.0097 = 0.2903$$

Example of training a MLP

Adjustments of weights



$$w_{11} = w_{11} - \alpha * \delta_1 * x_1 = (-0.2) - 0.25 * 0.0033 * 0.1 = -0.2$$

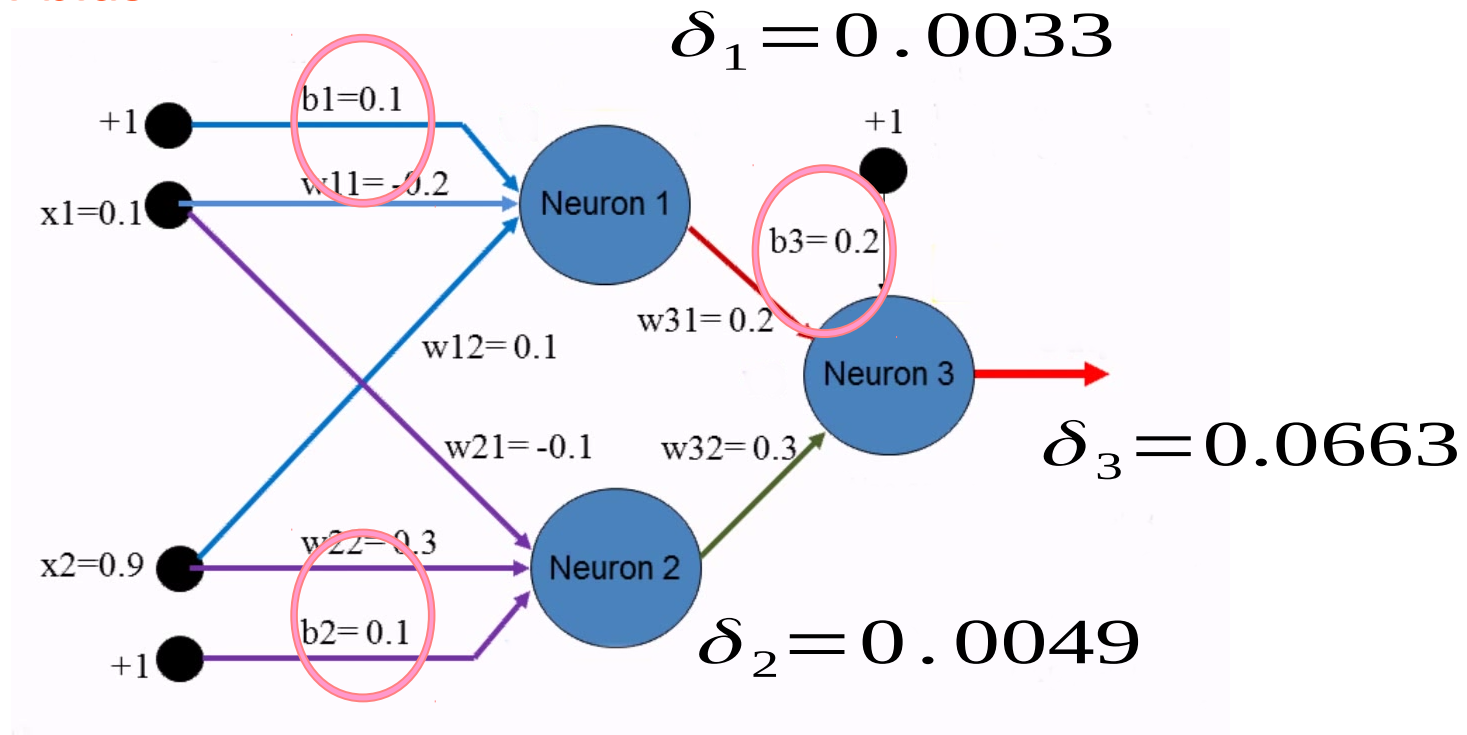
$$w_{21} = w_{21} - \alpha * \delta_2 * x_1 = (-0.1) - 0.25 * 0.0049 * 0.1 = -0.1$$

$$w_{12} = w_{12} - \alpha * \delta_1 * x_2 = 0.2 - 0.25 * 0.0033 * 0.9 = 0.1992$$

$$w_{22} = w_{22} - \alpha * \delta_2 * x_2 = 0.3 - 0.25 * 0.0049 * 0.9 = 0.2988$$

Example of training a MLP

Adjustments of bias



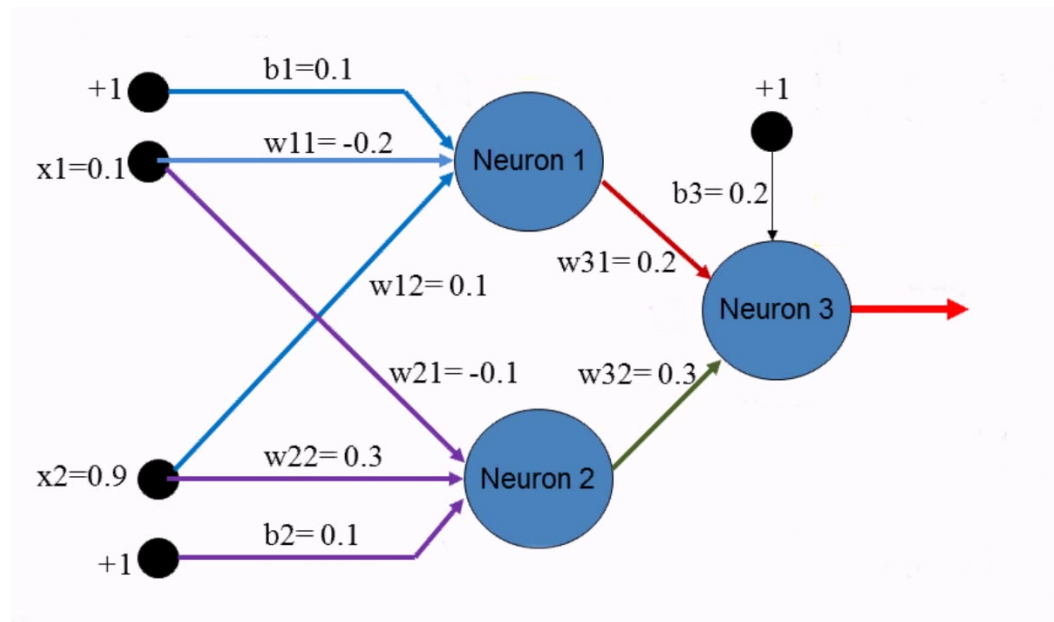
$$b_3 = b_3 - \alpha * \delta_3 * 1 = 0.2 - 0.25 * 0.0663 * 1 = 0.1834$$

$$b_2 = b_2 - \alpha * \delta_2 * 1 = 0.1 - 0.25 * 0.0049 * 1 = 0.0987$$

$$b_1 = b_1 - \alpha * \delta_1 * 1 = 0.1 - 0.25 * 0.0033 * 1 = 0.0991$$

Example of training a MLP

Example with more than one sample



Accumulation of the gradients

$$\theta = \sum \delta_l^{(l+1)} * a^{(l)}$$

For m = number of samples

$$w = w - \frac{1}{m} * \alpha * \theta$$