

Deep Learning I: Feedforward Neural Networks

COMP5318 Machine Learning and Data Mining

Semester 1, 2026, week 7

Nguyen H. Tran (slides prepared by Irena Koprinska)

Reference: Witten ch. 10.1-10.2, Tan ch. 4.7-4.8, Müller & Guido: ch.2.3.8, Geron: ch.10-11



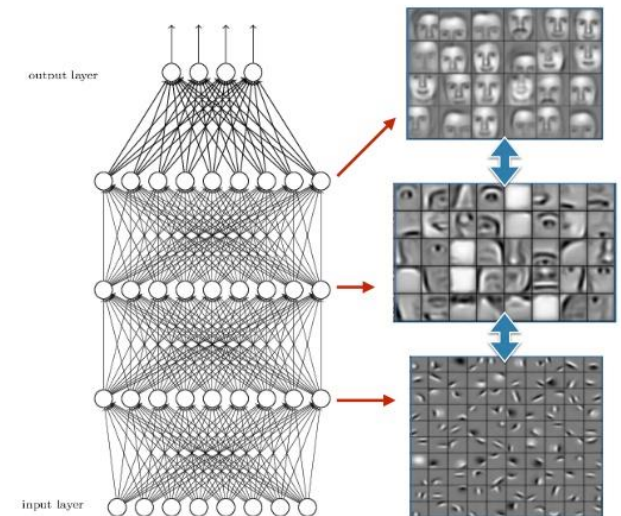
- Deep learning
- Introduction to neural networks
- Perceptrons
- Multi-layer perceptrons and the backpropagation algorithm
- Creating deep feedforward neural networks - modern techniques

- **Deep learning** is an approach to machine learning that is inspired by how the human brain operates
- It refers to **modern artificial Neural Networks (NNs)** and emphasizes that these networks are deeper - have more layers than the previous networks. This depth enables them to learn more complex input-output mappings.
- The term “deep learning” was introduced in the mid-2000s
- Before, since the 1940s - a lot of research on artificial NNs:
 - artificial neuron – introduced by McCulloch and Pitts in 1943
 - perceptron – developed by Rosenblatt in 1958
 - backpropagation algorithm – Werbos 1974; Rumelhart, Hinton and Williams 1986, Parker 1985; LeCun 1985

- Part of AI that focuses on creating **large** NNs that are capable of making accurate **data-driven** decisions

John Kelleher (Deep Learning, MIT Press, 2019)

- Suitable for applications with complex data and large datasets
- Deep NNs are able to learn hierarchical feature representations
- Making the NN deep by adding hidden layers subjects the features to a sequence of transformations and allows to learn important features



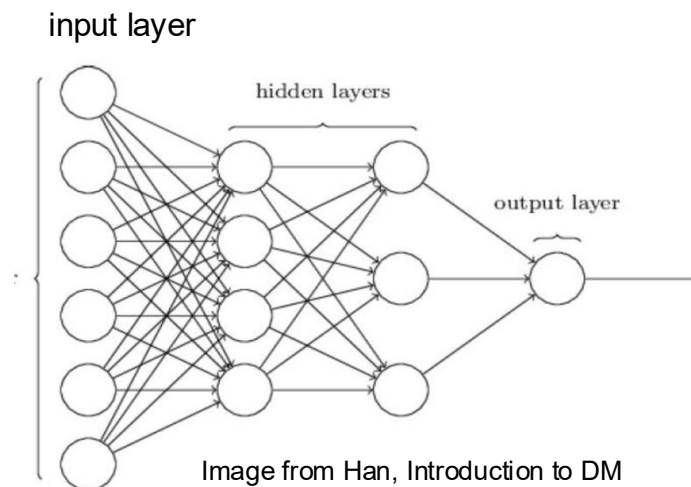
- Deep learning has been very successful in many areas and especially in image processing, computer vision, speech recognition and natural language processing
- Who uses it:
 - Facebook to analyse text in online conversations
 - Google, Baidu and Microsoft for image search and machine translation
 - Almost all smart phones for speech recognition and face detection
 - Self-driving cars – for localization, motion planning and steering
 - Healthcare and medicine – for processing medical images (X-ray, CT, MRI)

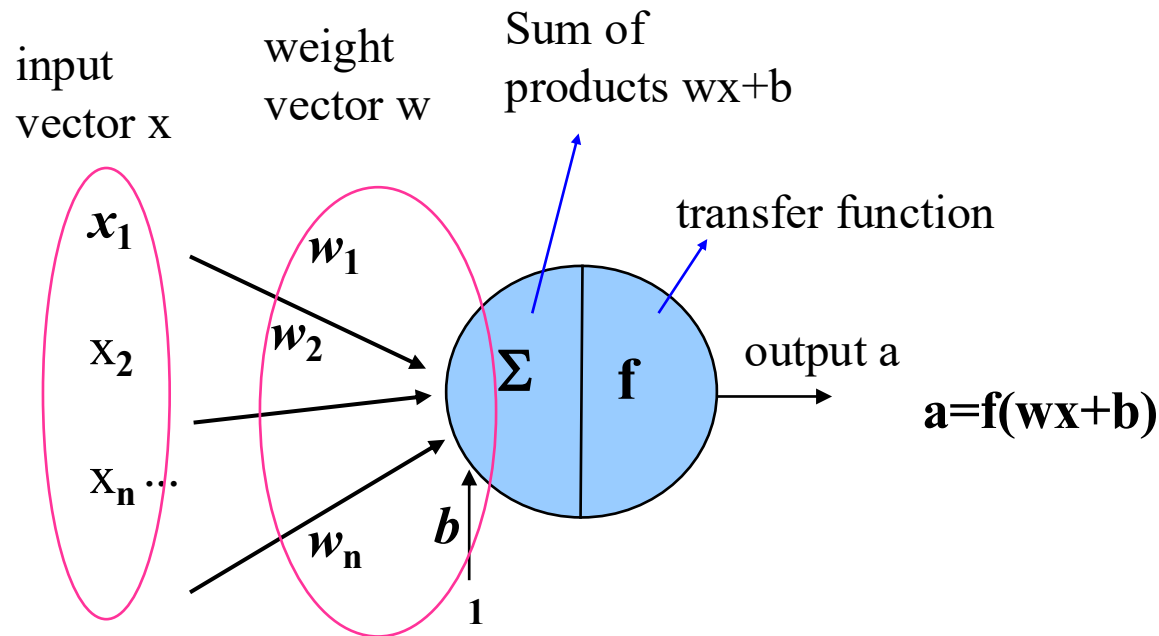
- <https://theconversation.com/can-robots-write-machine-learning-produces-dazzling-results-but-some-assembly-is-still-required-146090>
- <https://theconversation.com/ai-has-beaten-us-at-go-so-what-next-for-humanity-55945>
- <https://www.nytimes.com/2019/03/11/well/live/how-artificial-intelligence-could-transform-medicine.html>
- Find more news articles and post them on Ed!

- Most deep learning methods use multilayer perceptrons as building blocks, so we need to learn about **perceptrons**, **multilayer perceptrons** and the **backpropagation algorithm**

What is a Neural Network?

- Neural networks consist of **neurons** (units, nodes) that connected with each other with directed links where each **connection** has an associated numerical **weight**
- The neurons are typically organized into layers
 - Input layer, output layer and 1 or more hidden layers
- During training, the weights are adjusted, in order to learn to perform a certain task (e.g. to predict the correct class)

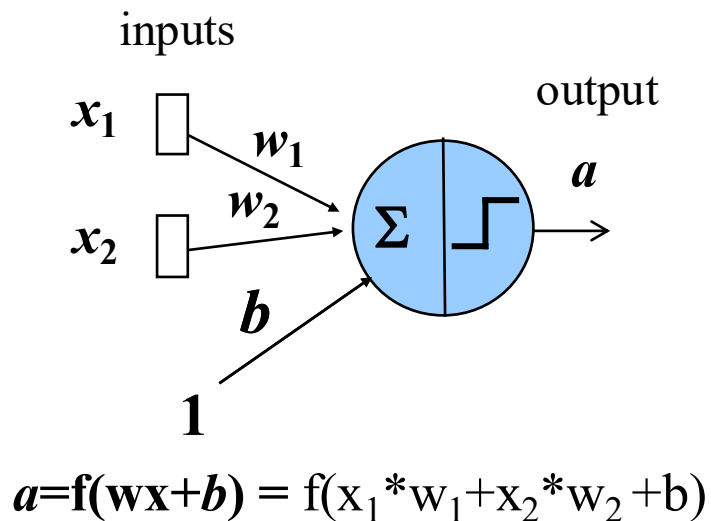




x comes from the data

w & b are the parameters of the neuron and they are learned using the learning rule for the specific type of NN

- The simplest neural network is called **perceptron**
- Uses a step transfer function
- Binary output: 0 and 1 (or -1 and 1)
- Output: weighted sum of its inputs, subject to a step transfer function



- **Step** transfer function:

$$a = f(n) = \begin{cases} 1 & \text{if } n \geq 0 \\ 0 & \text{if } n < 0 \end{cases}$$

Example:

$$x_1 = 0.2 \text{ and } x_2 = 0.3$$

$$w_1 = 2, w_2 = 1, b = -1.5$$

$$a = \text{step}(0.2 * 2 + 0.3 * 1 - 1.5) = \text{step}(-0.8) = 0$$

- t – target output, a – actual output; $\mathbf{x}$ – input vector

If $t = 1$ and $a = 0$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \mathbf{x}^T$

If $t = 0$ and $a = 1$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \mathbf{x}^T$

If $t = a$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}}$

- Define error: $e = t - a$

If $e = 1$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + \mathbf{x}^T$

If $e = -1$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} - \mathbf{x}^T$

If $e = 0$, then $\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}}$

- Perceptron rule in matrix format

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + e\mathbf{x}^T$$

$$\mathbf{b}^{\text{new}} = \mathbf{b}^{\text{old}} + e$$

- 1. Initialize the weights w and bias b to small random values, set epoch=1.
- 2. For each training example $\{x, t\}$ (input x , target output t)
 - 3. Calculate a , the output of the network for this example (also called network activation)
 - 4. Compute the output error $e = t - a$
 - 5. Update the weights:

$$\mathbf{w}^{\text{new}} = \mathbf{w}^{\text{old}} + e\mathbf{x}^T$$
$$\mathbf{b}^{\text{new}} = \mathbf{b}^{\text{old}} + e$$

- 6. At the end of each epoch check if the stopping condition is satisfied: all examples are correctly classified or a maximum number of epochs is reached; if yes - stop, otherwise epoch++ and repeat from step 2.

- The stopping condition is checked at the end of each **epoch**:
- Epoch - one pass through the whole training set – this means that:
 - training example 1 is passed, the perceptron output is computed and the weights are changed, then the next example is passed etc. – repeat for all training examples)
- The epoch numbering starts from 1: epoch 1, epoch 2, etc.
- To check if all examples are correctly classified at the end of the epoch:
 - All training examples are passed again one-by-one, the perceptron's output is calculated and compared with the target output. There is no weight change here.
 - This check does not count for another epoch as there is no weight change

- Given is the following training data:

Ex.	input	output
1	1 0 0	0
2	1 0 1	1
3	1 1 0	0

- Train a perceptron on this data. The initial weights are $w=[0.3, 0.2, 0.4]$, $b=0.1$. Stopping criterion: all examples are correctly classified or a maximum number of 2 epochs is reached.

Epoch=1, $w=[0.3 \ 0.2 \ 0.4]$, $b=0.1$

Apply Ex.1: $[1 \ 0 \ 0]$, $t=0$

$a=\text{step}([0.3 \ 0.2 \ 0.4][1 \ 0 \ 0]+0.1)=\text{step}(0.4)=1$, incorrect; $e=t-a=0-1=-1$

$w_{\text{new}}=[0.3 \ 0.2 \ 0.4] + (-1)[1 \ 0 \ 0]=[-0.7 \ 0.2 \ 0.4]$

$b_{\text{new}}=0.1+(-1)=-0.9$

Apply Ex.2: $[1 \ 0 \ 1]$, $t=1$

$a=\text{step}([-0.7 \ 0.2 \ 0.4][1 \ 0 \ 1]-0.9)=\text{step}(-1.2)=0$, incorrect; $e=t-a=1-0=1$

$w_{\text{new}}=[-0.7 \ 0.2 \ 0.4] + (1)[1 \ 0 \ 1]=[0.3 \ 0.2 \ 1.4]$

$b_{\text{new}}=-0.9+1 = 0.1$

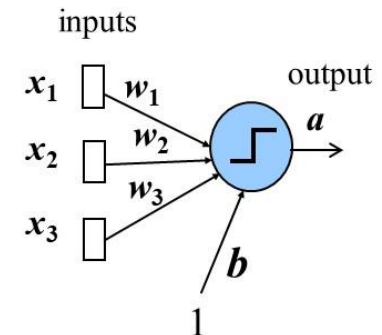
Apply Ex.3: $[1 \ 1 \ 0]$, $t=0$

$a=\text{step}([0.3 \ 0.2 \ 1.4][1 \ 1 \ 0]+0.1)=\text{step}(0.6)=1$, incorrect; $e=t-a=0-1=-1$

$w_{\text{new}}=[0.3 \ 0.2 \ 1.4] + (-1)[1 \ 1 \ 0]=[-0.7 \ -0.8 \ 1.4]$

$b_{\text{new}}=0.1-1 = -0.9$

Ex.	input	output
1	1 0 0	0
2	1 0 1	1
3	1 1 0	0



End of epoch 1. Check if the stopping condition is satisfied:

- 1) All training examples are correctly classified?
- 2) Start epoch 2:

Ex.	input	output
1	1 0 0	0
2	1 0 1	1
3	1 1 0	0

current weight vector and bias:

$w = [-0.7 \ -0.8 \ 1.4]$

$b = -0.9$

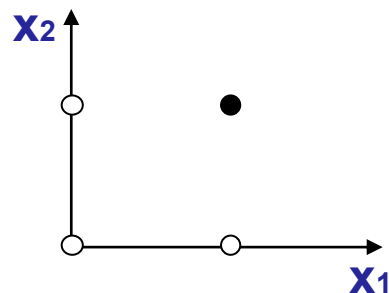
Apply Ex.1 [1 0 0], $t=0$

$a = \text{step}([-0.7 \ -0.8 \ 1.4][1 \ 0 \ 0] - 0.9) = \text{step}(-1.6) = 0$, correct

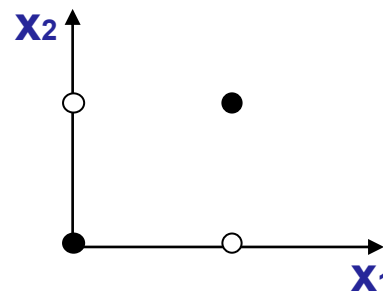
Apply Ex.2 [1 0 1], $t=1$

$a = \text{step}([-0.7 \ -0.8 \ 1.4][1 \ 0 \ 1] - 0.9) = \text{step}(-0.2) = 0$, incorrect $\Rightarrow$ condition not satisfied

- If the training examples are **linearly separable**, the perceptron learning rule is guaranteed to converge to a solution - a set of weights that correctly classifies the training examples - in a finite number of steps
 - In this case, the perceptron will find a linear decision boundary that separates the two classes
 - It doesn't try to find an "optimal" line, it will simply stop when a separating line is found
- Most of the problems are not linearly separable, so this is a limitation of the perceptrons



linearly separable

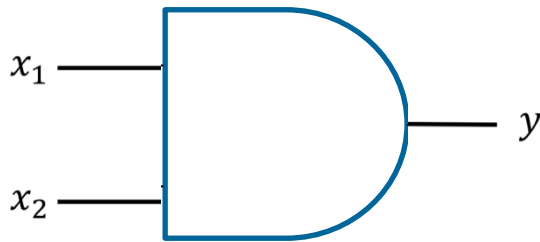


linearly non-separable

- The perceptron learning algorithm was proposed by Frank Rosenblatt in 1957
- The limitation of the perceptron were publicized by Marvin Minsky and Seymour Papert in the book “Perceptrons”
- Rosenblatt and his colleagues were aware that this limitation can be overcome by using more complex NNs (multi-layer perceptrons) but they were not able to adapt the perceptron learning rule to train these networks
- Let's see some examples of linearly separable and inseparable problems!

- Can a perceptron implement the AND function? Is this a linearly separable problem?

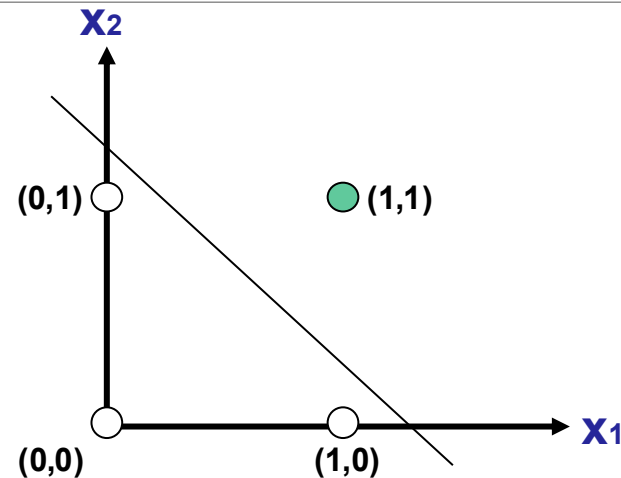
AND Gate



Output = 1, if both inputs are 1
Output = 0, otherwise

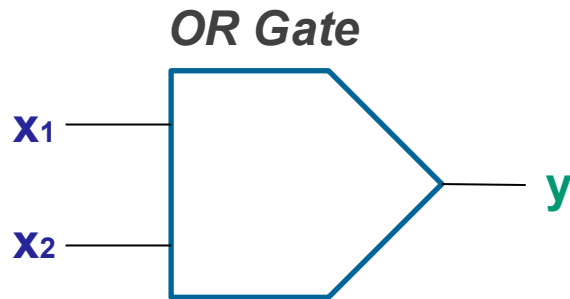
Input and Output

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



Yes!

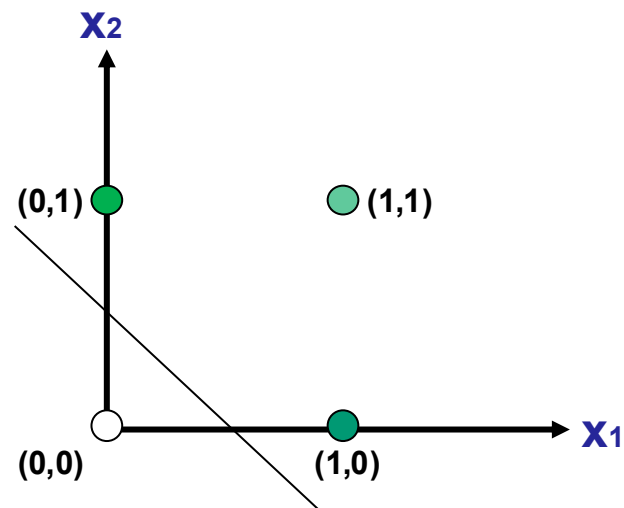
- Can a perceptron implement the OR function?



Output = 1, if 1 or more of the inputs are 1
Output = 0, if all inputs are 0

Input and output

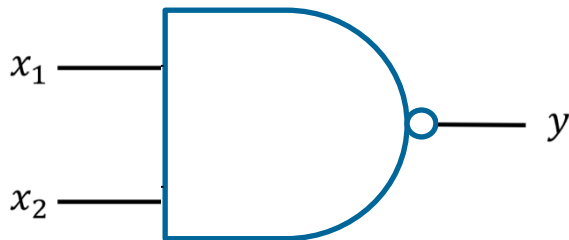
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1



Yes!

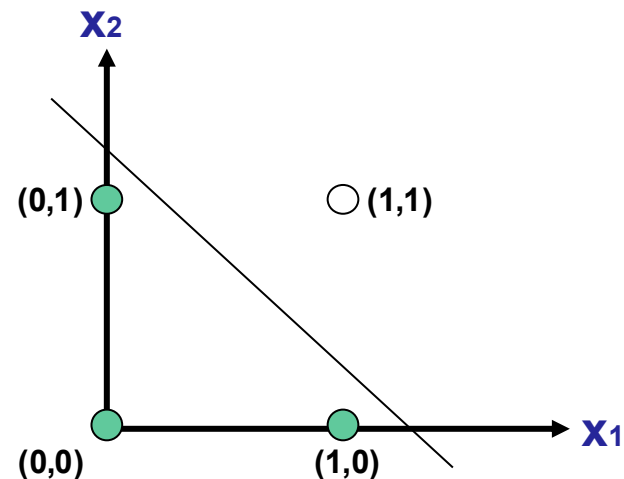
- Can a perceptron implement the NAND (negated AND) function?

NAND Gate



Input and Output

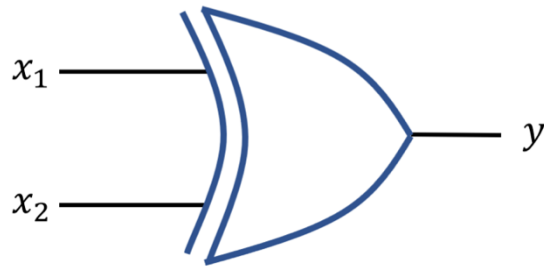
x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0



Yes!

- Can a perceptron implement the XOR function?

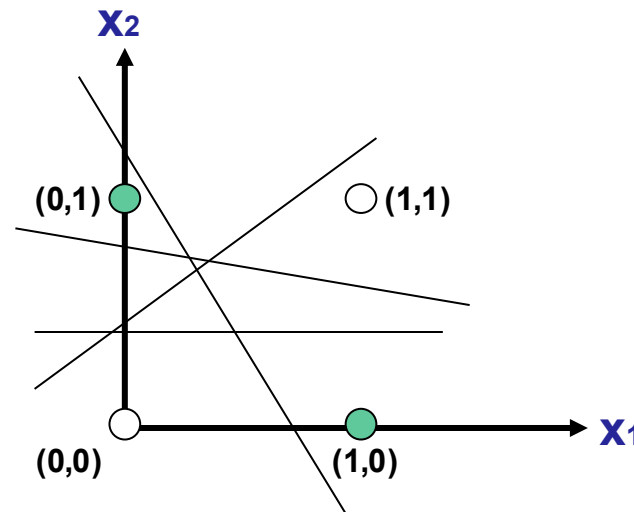
XOR Gate



Output = 0, if both inputs are the same (0 or 1)
Output = 1, otherwise

Input and Output

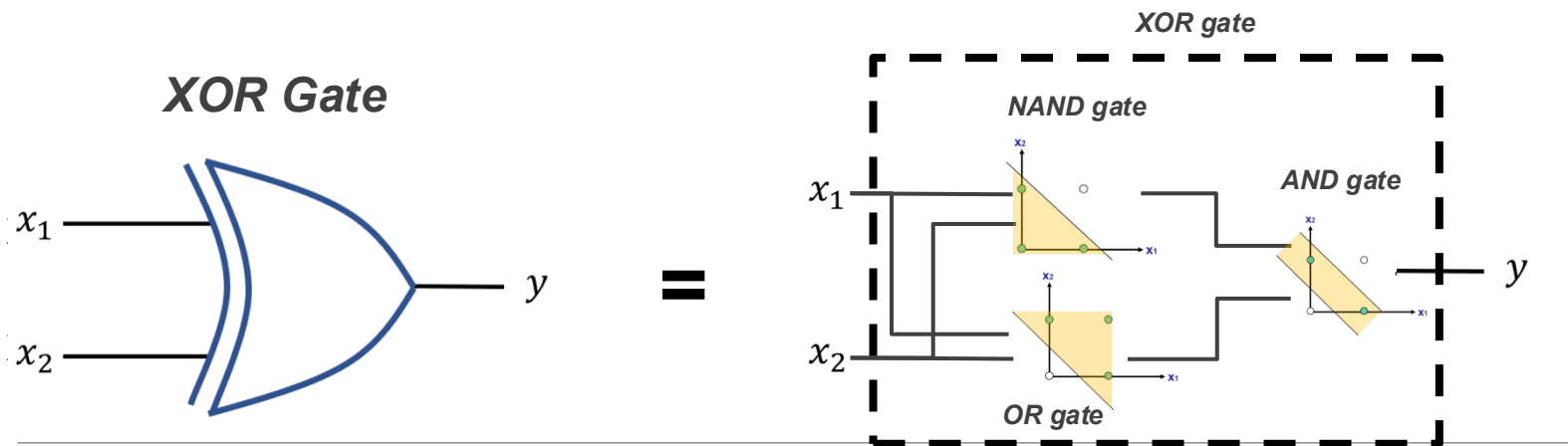
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



No!

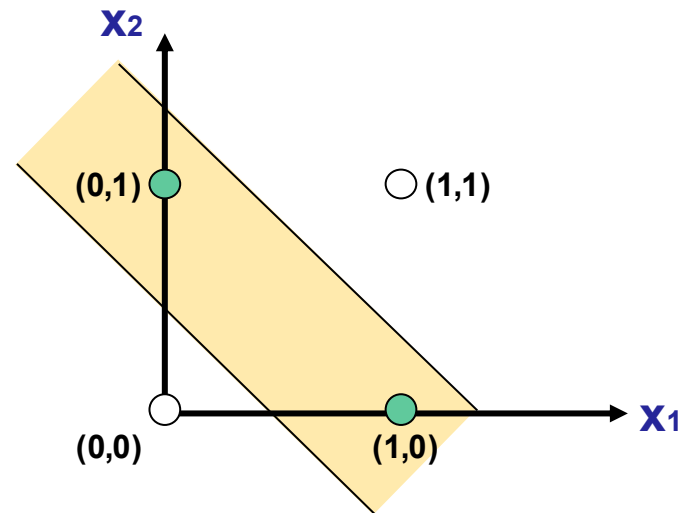
Perceptron for XOR gate 2

- XOR Gate = combination of NAND, OR and AND gates



Input and Output

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

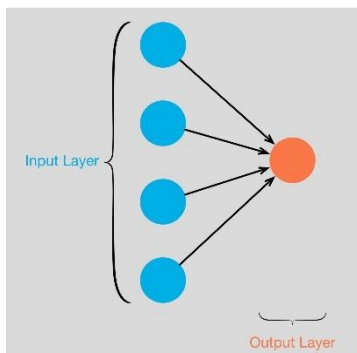
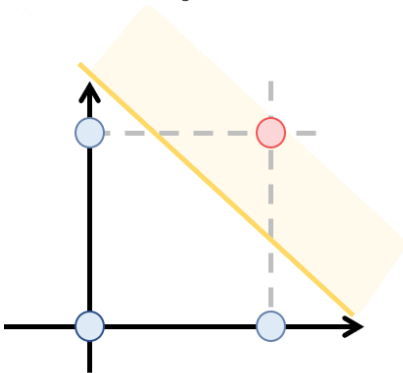


A 2-layer
perceptron
can solve
the XOR
problem!

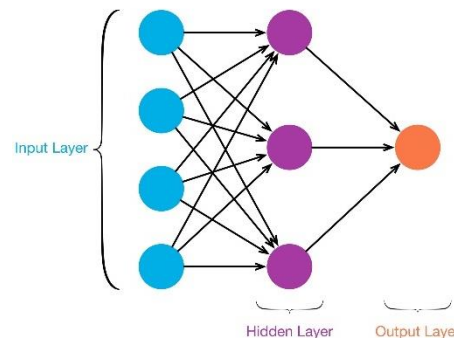
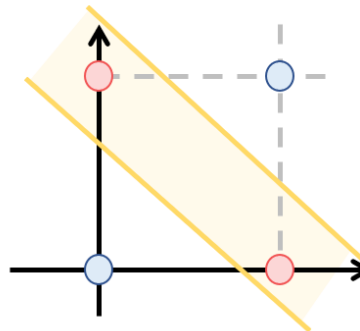
Adding more layers

- If we add more layers, we can form more complex boundaries
- But how to train these networks?

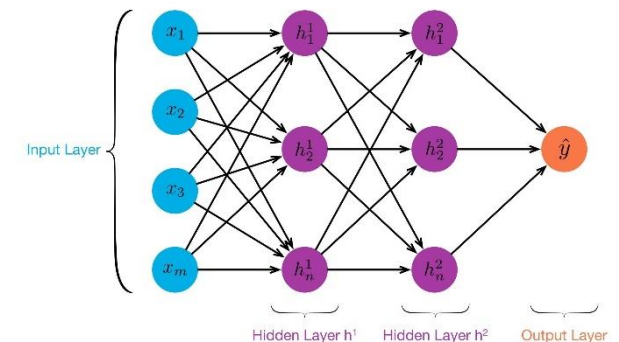
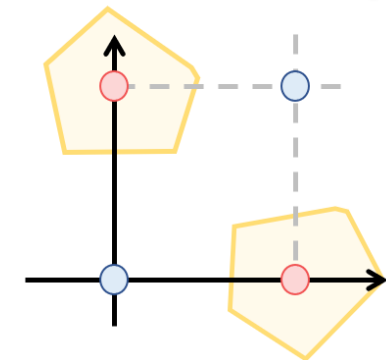
One layer



Two layers



Three layers

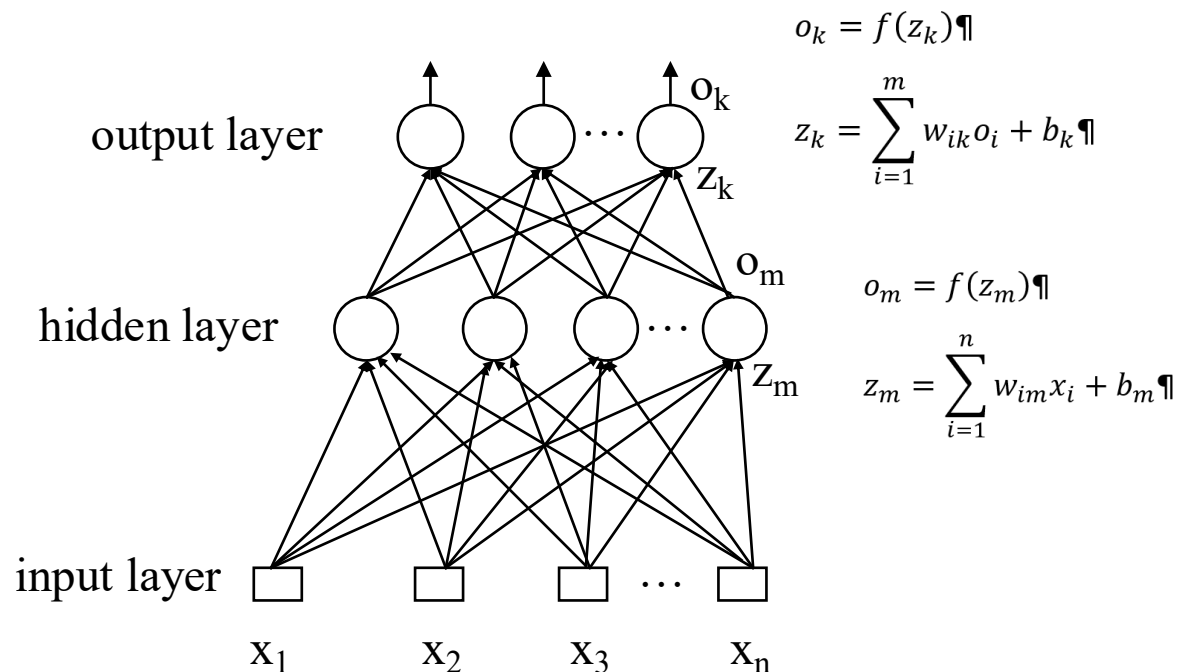




Multi-layer perceptrons and backpropagation algorithm

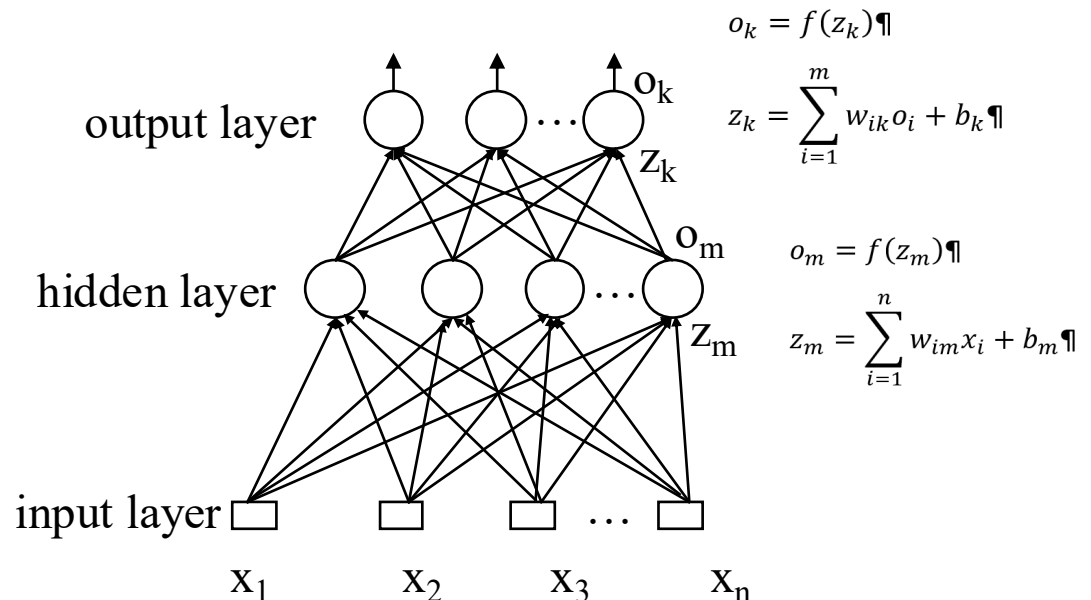
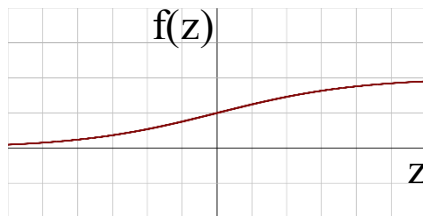
- An algorithm that is used to train multi-layer perceptron neural networks
- Proposed by Paul Werbos in 1974; later re-discovered by David Rumelhart, Geoffrey Hinton, Ronald Williams 1986; David Parker 1985; Yann LeCun 1985

- Layers of neurons – input layer, output layer, 1 or more hidden layers
- Feedforward NN – each neuron receives input only from the previous layer
- Fully connected network – each neuron in the current layer is connected with all neurons from the previous



- A single neuron calculates the weighed sum of the outputs of the previous layer, which is passed through a transfer function
- The transfer function needs to be **differentiable**
- Most widely used transfer function: sigmoid

$$f(z) = 1/(1 + e^{-z})$$



Backpropagation algorithm - idea

- For each training example $\{\mathbf{x}, \mathbf{t}\}$, $\mathbf{x} = \{x_1, x_2, \dots, x_n\}$
 - Propagate $\mathbf{x}$ through the network and calculate the output $\mathbf{o}$. Compare it with the target output $\mathbf{t}$ and calculate the error.
 - Update weights of the network to reduce the error
- Until error over all examples $<$ threshold
- Adjusts the weights backwards - from the output to the input neurons by propagating the **weight change** to minimize the error

$$w_{pq}^{new} = w_{pq}^{old} + \Delta w_{pq}$$

- How to calculate the weight change?
- By defining an error function and using the **gradient descent** algorithm to calculate it

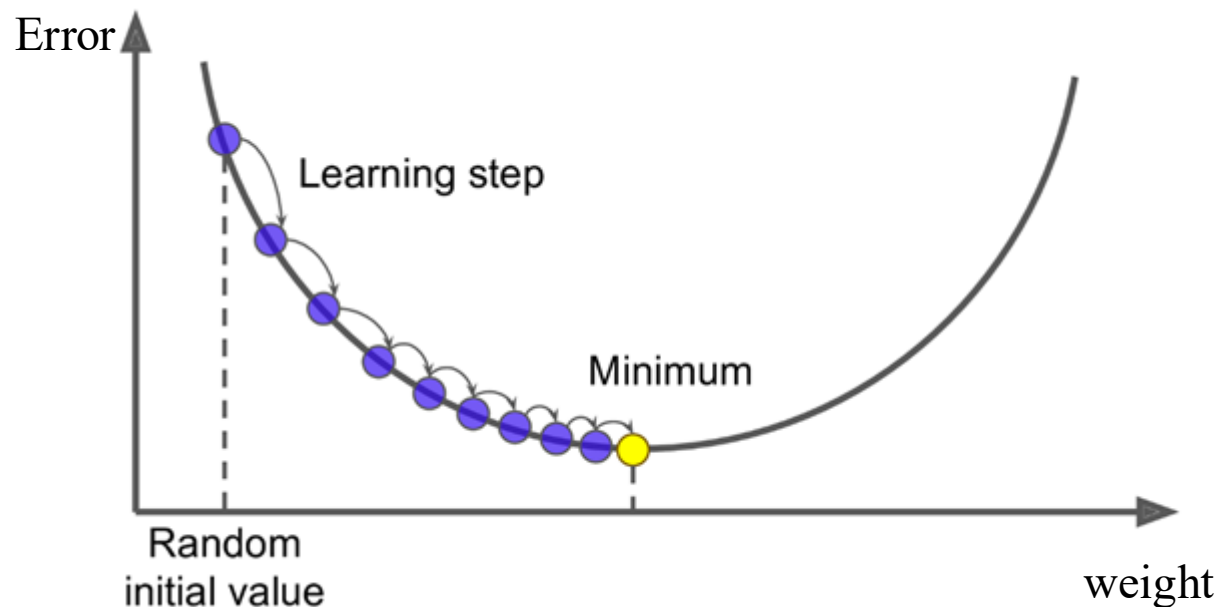
Steepest gradient descent

- We define an error function (also called cost or loss function), e.g. MSE over all training examples
- We can minimize it by using the steepest gradient descent algorithm (standard optimization method)
- The NN weights are iteratively updated by moving downhill - in the direction that reduces the error the most – this is the direction of the negative of the **gradient**



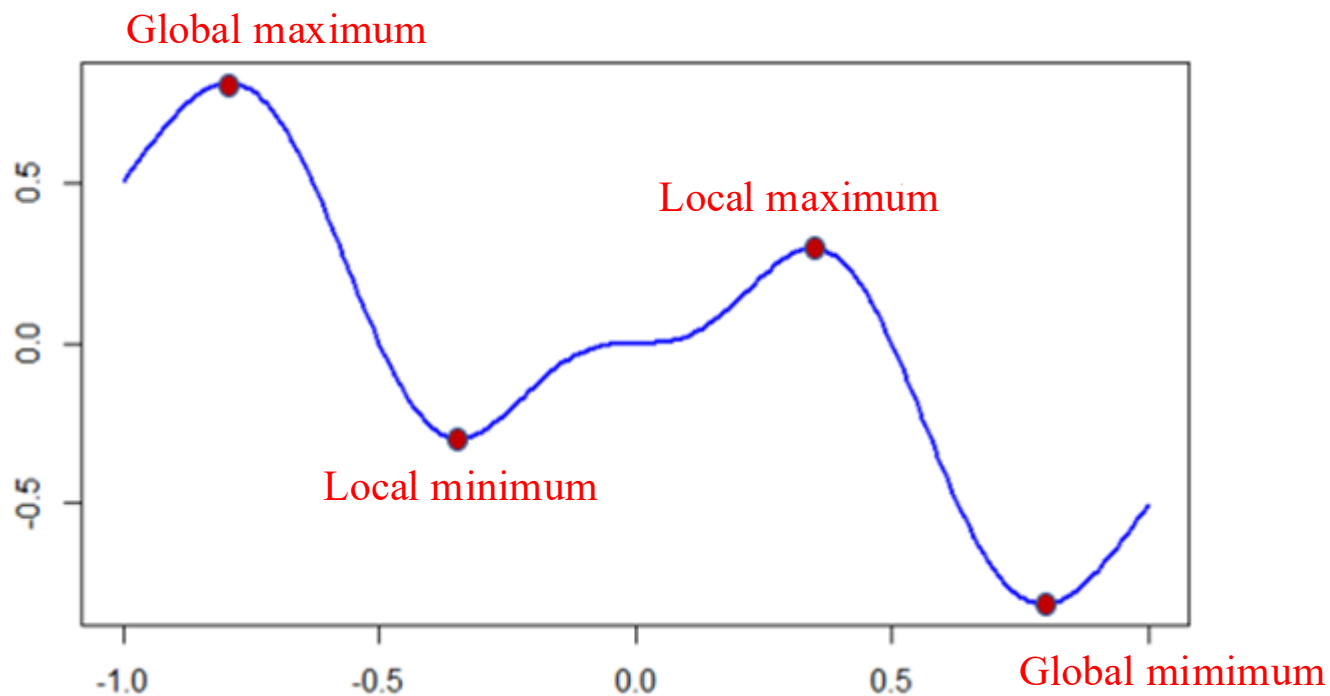
- Error landscape: the error as a function of the weights
- 1 state is 1 NN configuration (with all weights)

- The step that is used to move downhill is called **learning rate**; it is a hyperparameter of the algorithm
- Learning rate η



Local and global minimum

- The gradient descent algorithm is not guaranteed to find the global minimum, it converges to the closest local minimum depending on the starting position



$w_{pq}(t)$ - weight from neuron p to neuron q at time t

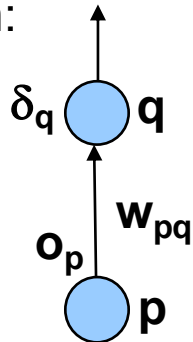
$$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}$$

$\Delta w_{pq} = \eta \cdot \delta_q \cdot o_p$ - weight change

- The weight change is proportional to the output activation of neuron p and the error δ of neuron q
- δ is calculated in 2 different ways:

- 1) q is an output neuron:

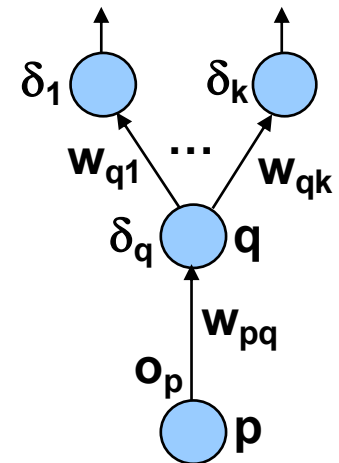
$$\delta_q = (t_q - o_q) f'(z_q)$$



- 2) q is a hidden neuron:

$$\delta_q = f'(z_q) \sum_i w_{qi} \delta_i$$

- (i is over all neurons in the layer above q)

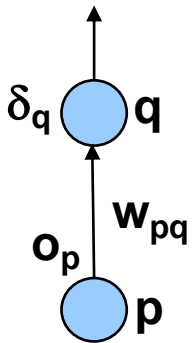


- $f'(z_q)$ is the first derivative of the activation function used in neuron q with respect to its input z_q

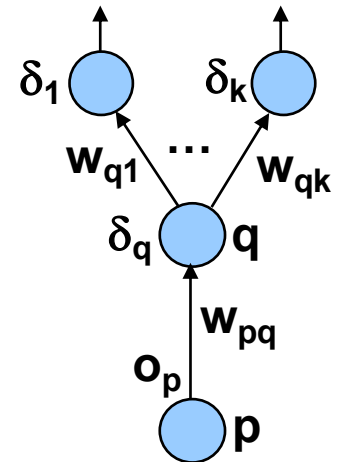
Weight update for sigmoid transfer function

- It can be shown that: $f'(x) = f(x)(1 - f(x))$
- => Weigh update formulas for **sigmoid transfer function**:

- 1) q is an output neuron:
 $\delta_q = (t_q - o_q)o_q(1 - o_q)$



- 2) q is a hidden neuron:
 $\delta_q = o_q(1 - o_q) \sum_i w_{qi} \delta_i$
- (i is over all neurons in the layer above q)

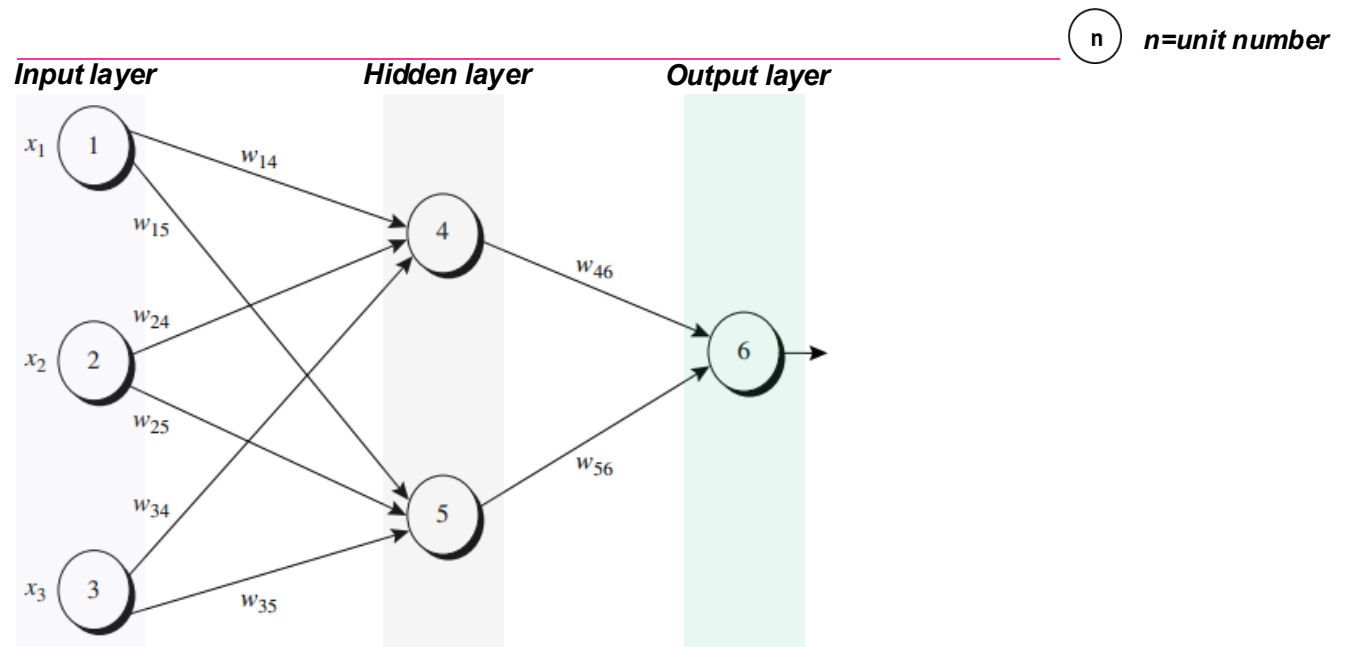


Training NN with the backpropagation algorithm

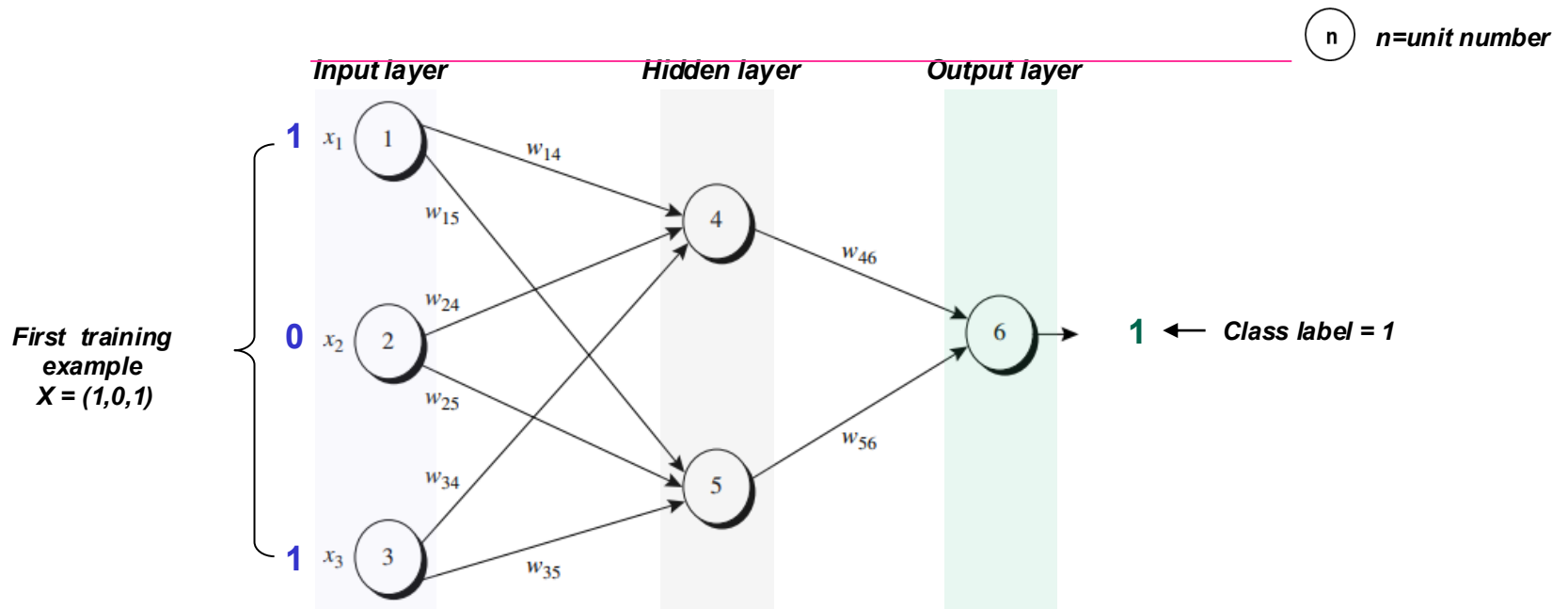
- 1. Initialize all weights (biases incl.) to small random values, e.g. $\in [-1, 1]$
- 2. Repeat until stopping condition is satisfied:
 - (forward pass)
 - --Present a training example and compute the network output
 - (backward pass)
 - --Compute the δ values for the output neurons and update the weights to the output layer
 - --Starting with output layer, repeat for each layer in the network:
 - propagate the δ values back to the previous layer
 - update the weights between the two layers
- 3. Check the stopping condition at the end of each epoch – e.g. the error on the training set is below a threshold or a maximum number of epochs has been reached

Backpropagation algorithm - example

- NN with 1 hidden layer, trained with the backpropagation algorithm
- Assume the learning rate is 0.9

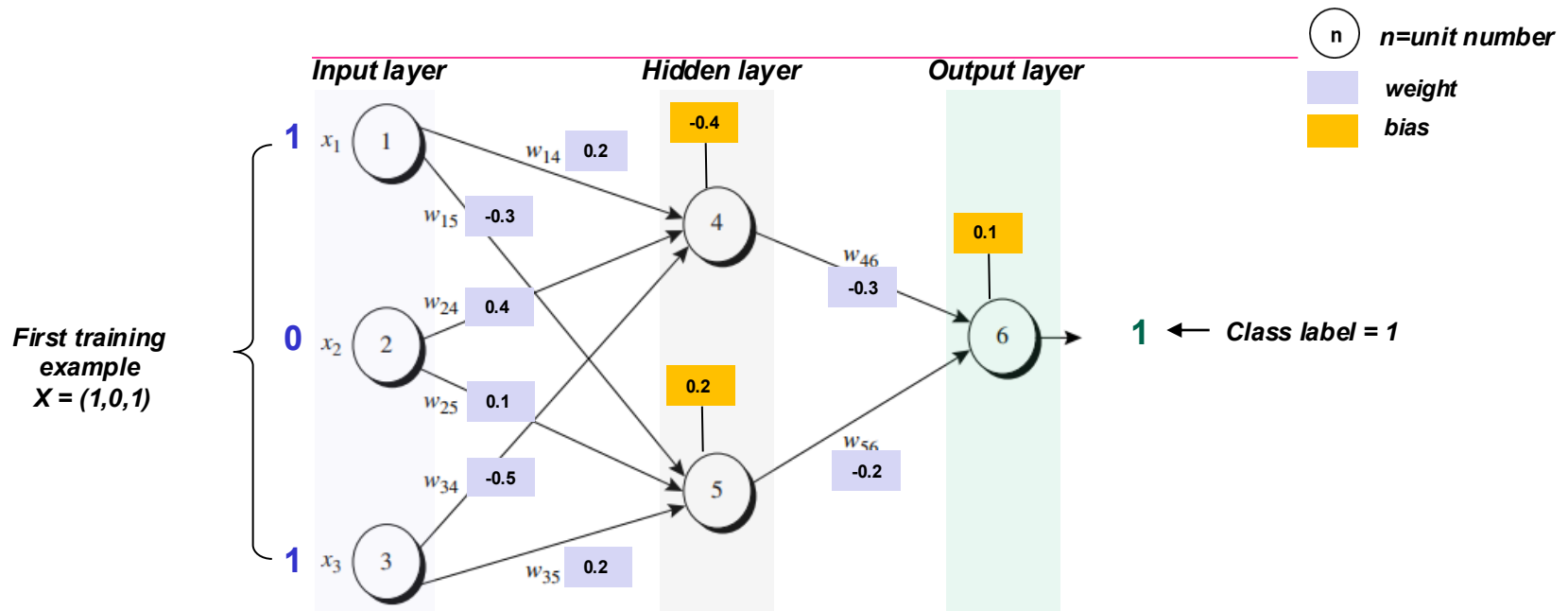


- Input example $x=(1,0,1)$, target: 1



2 – Initialization of weights

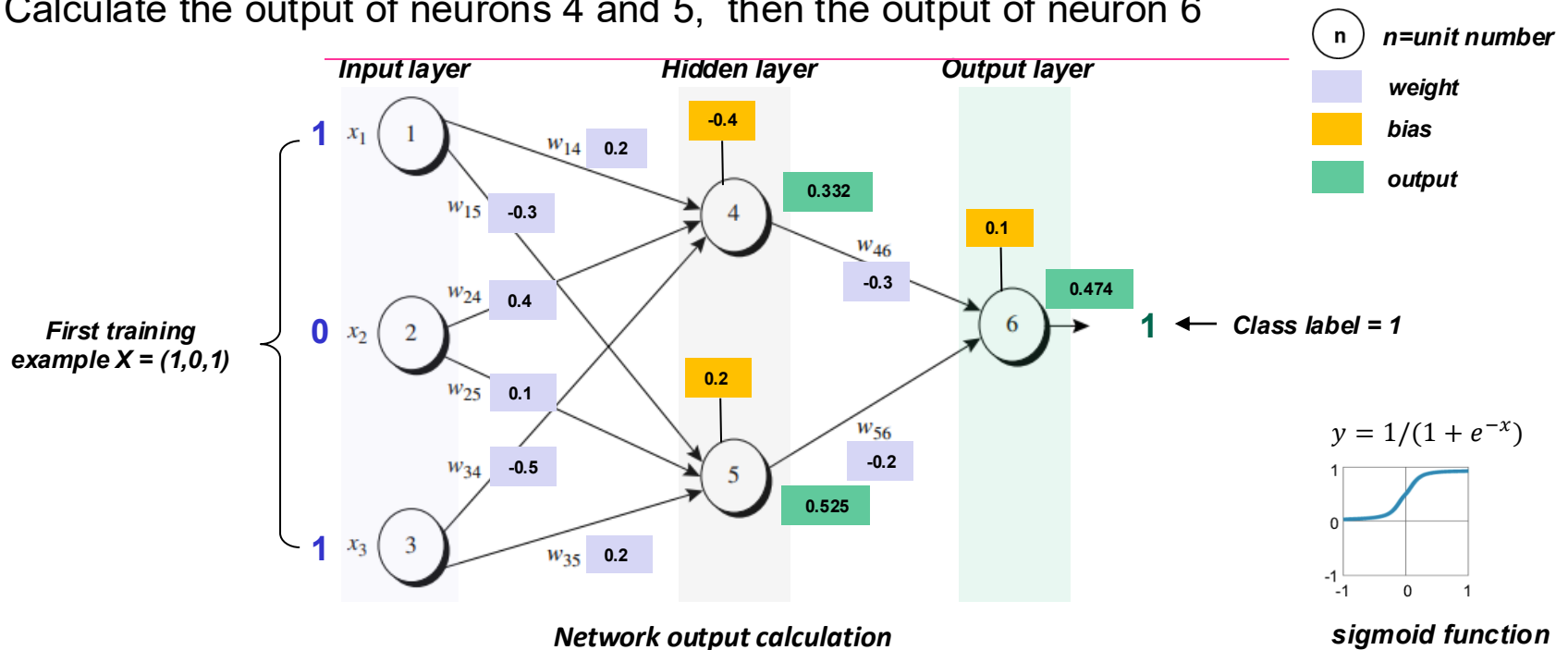
- Initialize the weights w and biases θ to small random values
- (The biases are denoted with θ , not b)



input vector x			initial weights w								initial biases θ		
x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	-0.3	0.4	0.1	-0.5	0.2	-0.3	-0.2	-0.4	0.2	0.1

3 – Compute NN output

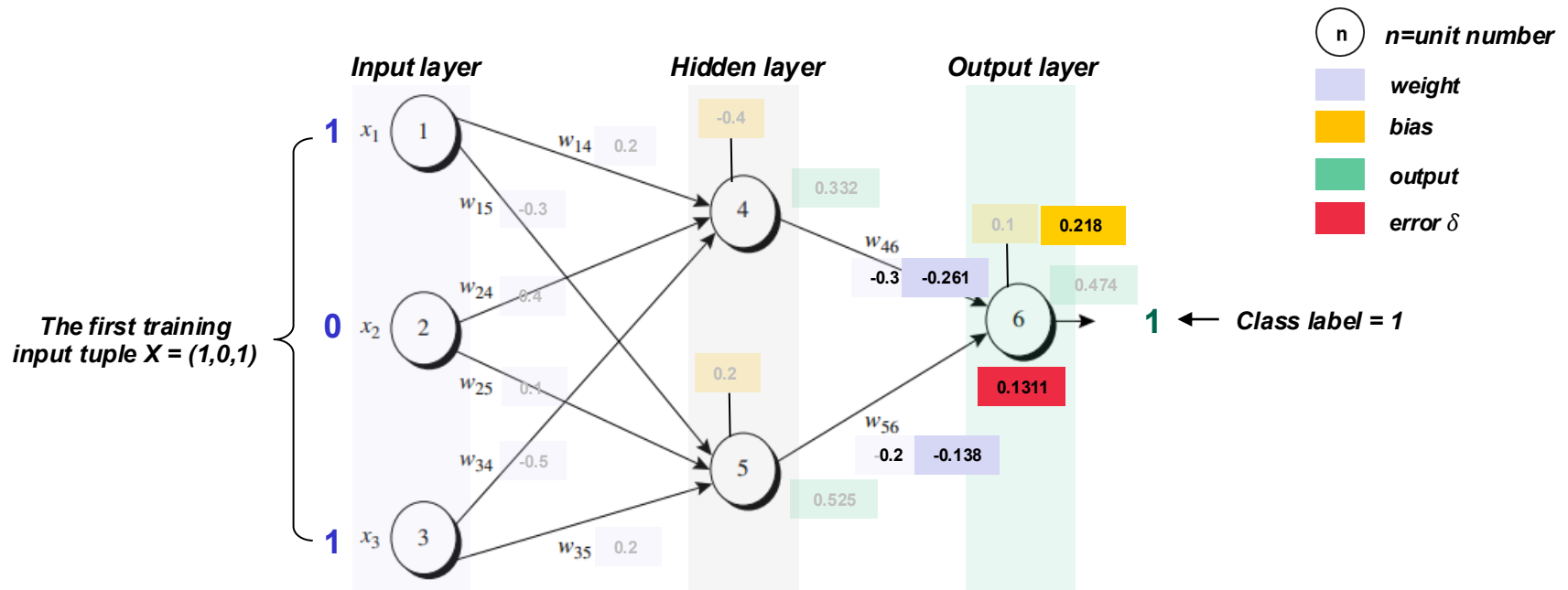
- Use sigmoid transfer functions for hidden and output neurons (input neurons don't have transfer functions)
- Calculate the output of neurons 4 and 5, then the output of neuron 6



- Input to neuron 4: $z_4 = 1 \cdot 0.2 + 0 \cdot 0.4 + 1 \cdot (-0.5) - 0.4 = -0.7$, output of neuron 4: $o_4 = 1 / (1 + e^{0.7}) = 0.332$
- Input to neuron 5: $z_5 = 1 \cdot (-0.3) + 0 \cdot 0.1 + 1 \cdot 0.2 + 0.2 = 0.1$, output of neuron 5: $o_5 = 1 / (1 + e^{-0.1}) = 0.525$ NN output
- Input to neuron 6: $z_6 = 0.332 \cdot (-0.3) + 0.525 \cdot (-0.2) + 0.1 = -0.105$, output of neuron 6: $o_6 = 1 / (1 + e^{0.105}) = \mathbf{0.474}$

4 – Compute errors δ of output layer and update the weights to output layer

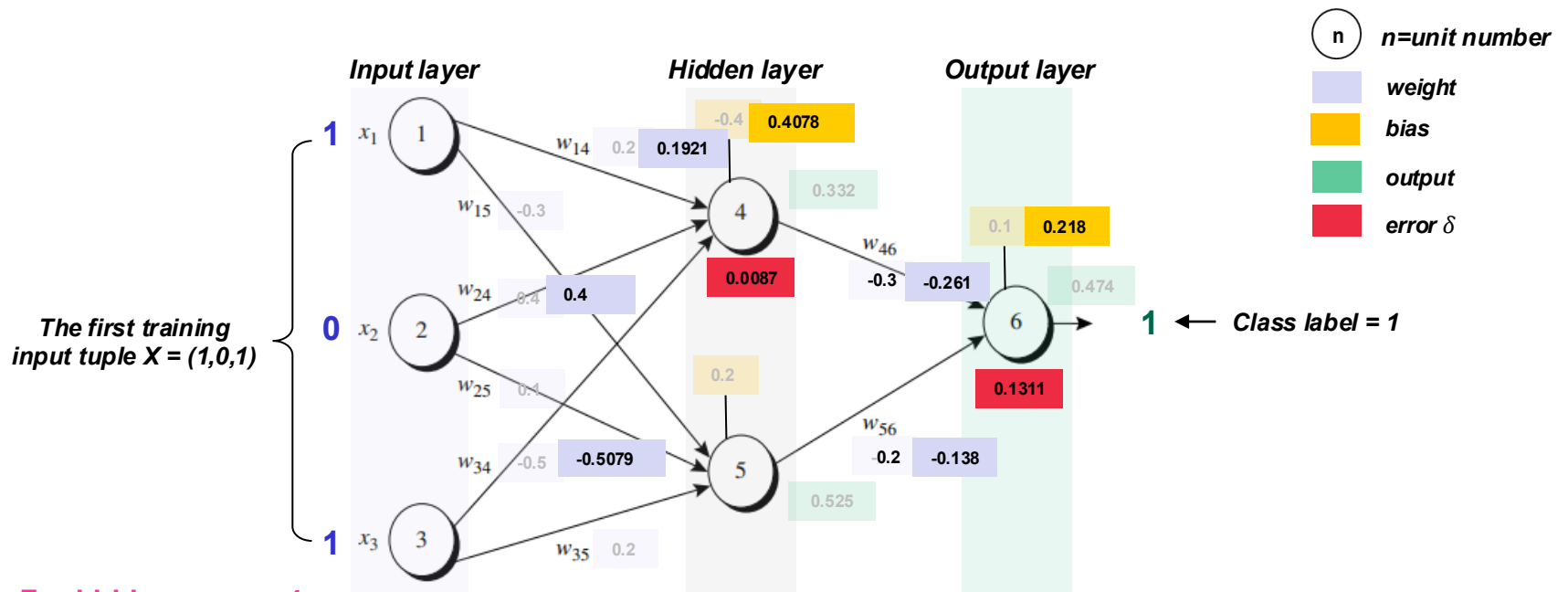
- Compute the δ values for the output layer neurons (neuron 6) and update the weights to the output neuron (between the hidden neurons 4 and 5 and output neurons 6)



- $\delta_6 = (t_6 - o_6) * o_6 * (1 - o_6) = (1 - 0.474) * 0.474 * (1 - 0.474) = 0.1311$
- $\Delta w_{46} = \eta * \delta_6 * o_4 = 0.9 * 0.1311 * 0.332 = 0.039$, $w_{46}^{new} = w_{46}^{old} + \Delta w_{46} = -0.3 + 0.039 = -0.261$
- $\Delta w_{56} = \eta * \delta_6 * o_5 = 0.9 * 0.1311 * 0.525 = 0.0619$, $w_{56}^{new} = w_{56}^{old} + \Delta w_{56} = -0.2 + 0.0619 = -0.138$
- $\theta_6^{new} = \theta_6^{old} + \Delta \theta_6 = \theta_6^{old} + \eta * \delta_6 * 1 = 0.1 + 0.9 * 0.1311 * 1 = 0.218$

5 – Compute errors δ of hidden layer and update the weights to hidden layer (1)

- Compute the δ values for the hidden neurons (neurons 4 and 5) and update the weights to the hidden layer (between input neurons 1, 2 and 3 and hidden neurons 4 and 5)

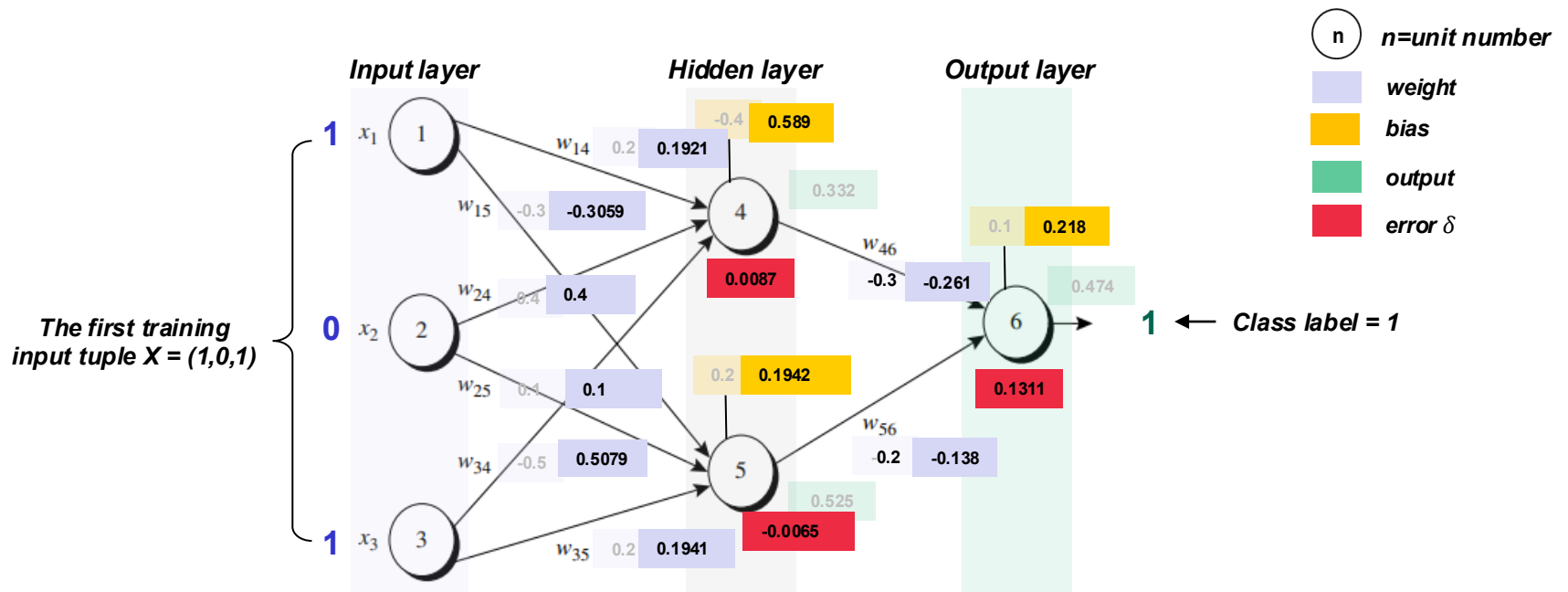


For hidden neuron 4:

- $\delta_4 = o_4 * (1 - o_4) * w_{46} * \delta_6 = 0.332 * (1 - 0.332) * (-0.3) * 0.1311 = -0.0087$
- $\Delta w_{14} = \eta * \delta_4 * o_1 = 0.9 * (-0.0087) * 1 = -0.0079$, $w_{14\text{new}} = w_{14\text{old}} + \Delta w_{14} = 0.2 - 0.0079 = 0.1921$
- $\Delta w_{24} = \eta * \delta_4 * o_2 = 0.9 * (-0.0087) * 0 = 0$, $w_{24\text{new}} = w_{24\text{old}} + \Delta w_{24} = 0.4 + 0 = 0.4$
- $\Delta w_{34} = \eta * \delta_4 * o_3 = 0.9 * (-0.0087) * 1 = -0.0079$, $w_{34\text{new}} = w_{34\text{old}} + \Delta w_{34} = -0.5 - 0.0079 = -0.5079$
- $\theta_{4\text{new}} = \theta_{4\text{old}} + \Delta \theta_4 = \theta_{4\text{old}} + \eta * \delta_4 * 1 = -0.4 + 0.9 * (-0.0087) * 1 = -0.4078$

5 – Compute errors δ of hidden layer and update the weights to hidden layer (2)

- Compute the δ values for the hidden neurons (neurons 4 and 5) and update the weights to the hidden layer (between input neurons 1, 2 and 3 and hidden neurons 4 and 5)



For hidden neuron 5:

- $\delta_5 = o_5(1 - o_5)w_{56} \delta_6 = 0.525(1 - 0.525)(-0.2)0.1311 = -0.0065$
- $\Delta w_{15} = \eta \delta_5 o_1 = 0.9(-0.0065)1 = -0.0059$, $w_{15} \text{ new} = w_{15} \text{ old} + \Delta w_{15} = -0.3 - 0.0059 = -0.3059$
- $\Delta w_{25} = \eta \delta_5 o_2 = 0.9(-0.0065)0 = 0$, $w_{24} \text{ new} = w_{25} \text{ old} + \Delta w_{25} = 0.1 + 0 = 0.1$
- $\Delta w_{35} = \eta \delta_5 o_3 = 0.9(-0.0065)1 = -0.0059$, $w_{35} \text{ new} = w_{35} \text{ old} + \Delta w_{34} = 0.2 - 0.0059 = 0.1941$
- $\theta_5 \text{ new} = \theta_5 \text{ old} + \Delta \theta_5 = \theta_5 \text{ old} + \eta \delta_5 * 1 = 0.2 + 0.9(-0.0065)1 = 0.1942$

End of backward pass for input example X

- The standard gradient descent algorithm sums the error of all training examples and then updates the weights
- The Stochastic Gradient Descent (SGD) updates the weights after each example – this is the algorithm we used; SGD is usually faster
- Mini-batch SGD (also called batch gradient descent) – sum the error of mini-batches of training examples, update the weights

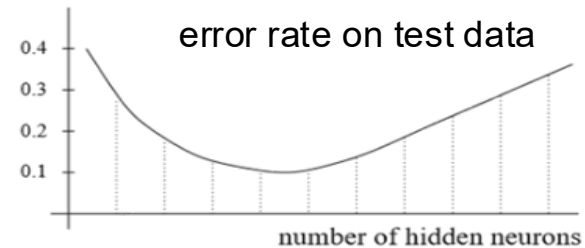
Universality of multi-layer perceptrons

- Multi-layer perceptrons trained with the backpropagation algorithm are **universal approximators** – theorems:
 - Any continuous function can be approximated with arbitrary small error by a network with one hidden layer (Cybenko 1989, Hornik et al. 1989):
 - Any function (including discontinuous) can be approximated to arbitrary small error by a network with two hidden layers (Cybenko 1988)
- These are existence theorems – they don't say how to choose the network architecture and hyperparameters

- Number of neurons in the input layer
 - Numerical attributes - 1 neuron per attribute
 - Categorical attributes with k values – k neurons per attribute, one-hot encoding
 - One-hot encoding - represent a k -valued attribute with k binary attributes, only 1 of which is set to 1 (“hot”) and all the others are 0
 - E.g. outlook had 3 values – sunny, overcast, rainy; one-hot encoding: 1 0 0 for outlook=sunny, 0 1 0 for outlook=overcast and 0 0 1 for outlook=rainy
- Number of neurons in the output layer
 - k -class problem: k neurons, 1 for each class, one-hot encoding
 - binary class problem:
 - 1 neuron for each class (=2 neurons) as k -class problem or
 - 1 neuron only with sigmoid transfer function – output close to 0 -> class1, close to 1-> class2

- Number of hidden layers and neurons in them – trial and error

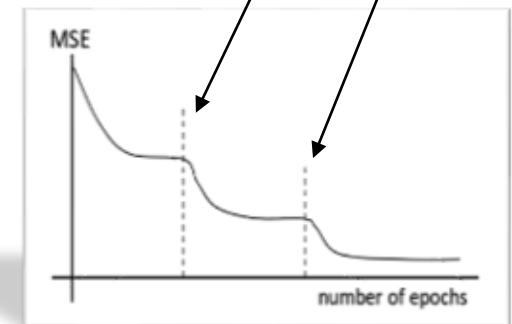
- Too many – overfitting
- Too few – underfitting



- Idea: Grow the hidden layer
- Start with a small network, train until the error rate no longer improves, then add new neurons (randomly initialized)

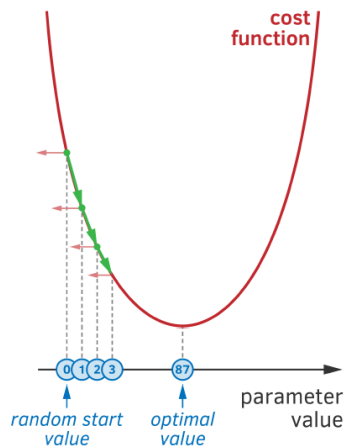
1. At the beginning, use only a few hidden neurons.
2. Train the network until the mean squared error no longer seems to improve.
3. At this moment, add a few neurons to the hidden layer, each with randomly initialized weights, and resume training.
4. Repeat the previous two steps until a termination criterion has been satisfied; for instance, when the new addition does not result in a significant error reduction, or when the hidden layer exceeds a user-set maximum size.

MSE levels off, adding new neurons reduced the error

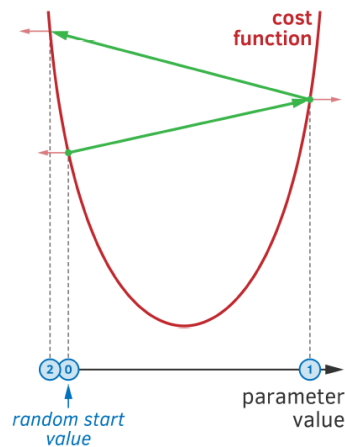


- Standard approach:
 - Each epoch runs through the same training examples, one by one, always in the same sequence
- Alternatives:
 1. For each epoch, permute the training examples
 2. Present more often the examples with higher error and less often the examples with lower error
 3. Present the examples not one by one, but in batches of N examples, summing up their individual errors and updating after each batch (mini-batch)

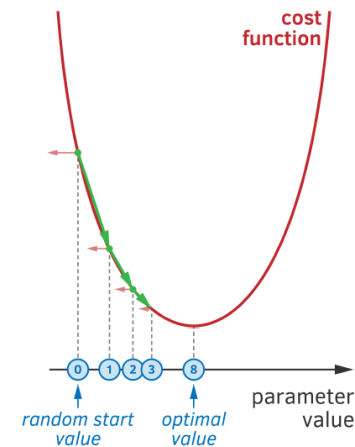
- The performance is very sensitive to the learning rate
 - Too small – slow convergence
 - Too big – oscillation, overshooting of the minimum
 - It is not possible to determine the best learning rate before training as it changes and depends on the error surface



(a) too small learning rate
slow convergence



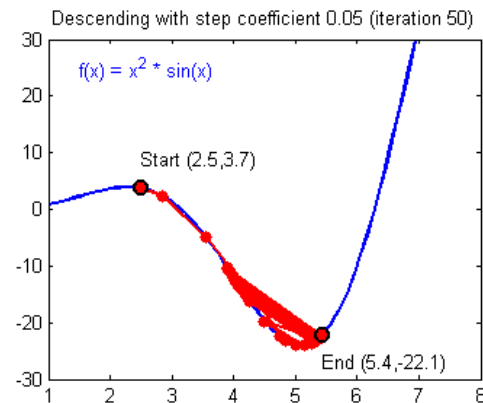
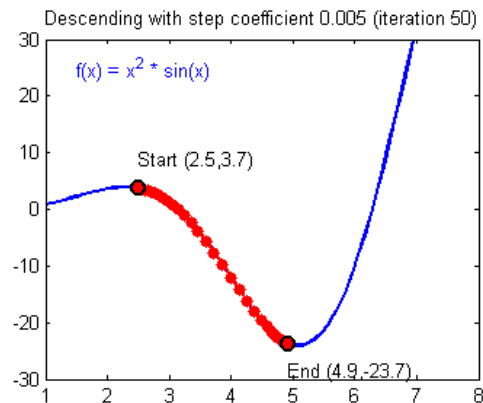
(b) too large learning rate
no convergence, oscillations



(c) good learning rate

Image from Burkov, ML Engineering, 2020

- Standard approach: constant learning rate during training



Constant learning rate – different values

Demo from <https://towardsai.net/p/machine-learning/analysis-of-learning-rate-in-gradient-descent-algorithm-using-python>

- Alternatives: variable learning rate – time-dependent
 - Start with a high learning rate, gradually decrease it – motivation:
 - Big learning rate initially -> greater weight change - can reduce the number of training epochs and may even help to jump over some local minima
 - Later - decrease the learning rate to prevent overshooting the global minimum
- Different decay schedules:
- $\eta_n = \frac{\eta_{n-1}}{1+d*n}$ (time-based) $\eta_n = \eta_0 e^{-dn}$ (exponential)
 n – epoch number, d - decay rate (hyperparameter)
e.g. if $\eta_0=0.3$ (initial learning rate), $d=0.5$, values in the first 5 epochs:
 - epoch 1: $\eta_1=0.2$
 - epoch 2: $\eta_2=0.1$
 - epoch 3: $\eta_3=0.04$
 - epoch 4: $\eta_4=0.013$

- Make the current update dependent on the previous by introducing a hyperparameter called momentum μ and a momentum term in the weight update equation
- Reduces oscillations and allows to use a larger learning rate

- Without momentum:

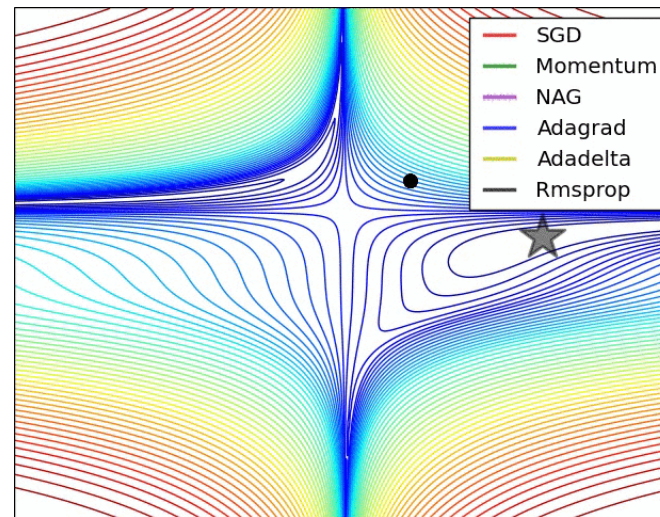
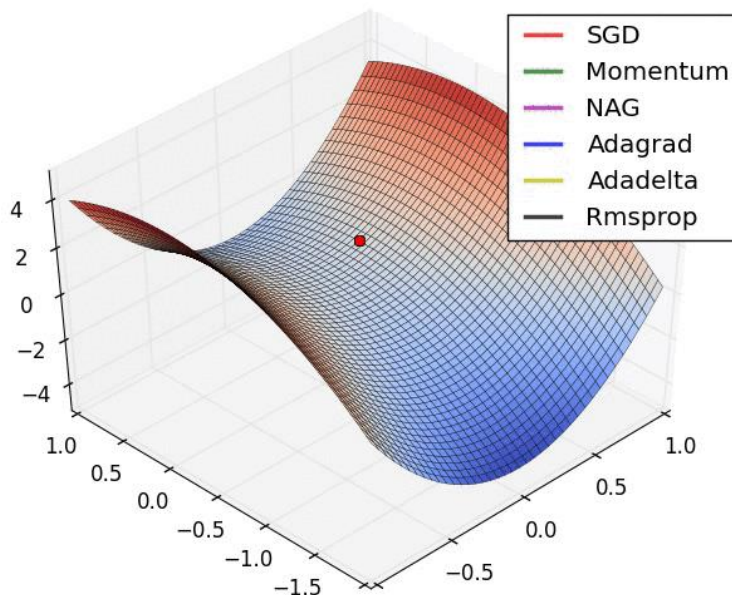
$$w_{pq}(t + 1) = w_{pq}(t) + \Delta w_{pq}(t), \text{ where } \Delta w_{pq}(t) = \eta \delta_q o_p$$

- With momentum: Momentum term

$$\Delta w_{pq}(t) = \eta \delta_q o_p + \mu(w_{pq}(t) - w_{pq}(t - 1))$$

Gradient descent algorithm - variations

- Also called optimisers
 - Standard gradient descent
 - Stochastic gradient descent (SGD)
 - Momentum
 - Adagrad
 - NAG
 - RMSProp
 - AdaDelta
 - Adam



- The performance is very sensitive to the weights initialization (w and b)
- Different strategies – most commonly used:
 - Random (standard approach) - small random values, e.g. $\in [-1, 1]$
 - Xavier initialization - the weights are sampled from normal distribution, centered at 0 with standard deviation $\sigma = \sqrt{\frac{2}{N_{in} + N_{out}}}$
- N_{in} – number of **input** neurons (neurons in the previous layer to which the current neuron is connected)
- N_{out} – number of **output** neurons (neurons in the next layer to which the current neuron is connected)
- The “current neuron” is the one being initialized



Creating deep feedforward neural networks - modern techniques

Deep learning – why now and not before?

- Deep learning – NNs with many hidden layers
 - Motivation: complex high-level features can be constructed by combining lower-level features
 - Greater number of hidden layers -> deeper hierarchy of features
- What are the enabling factors for the success of deep learning?
 - 1) Computational power - fast and powerful computers; powerful GPUs (Graphics Processing Units)
 - 2) Availability of large and high-quality datasets, especially labelled datasets such as ImageNet – 15 million images from 20,000 categories – enabled deep learning to go big! <https://www.image-net.org/>



Deep learning – why now and now before? (2)

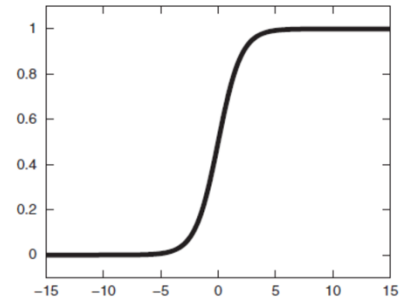
- 3) Algorithmic advances
 - Overcoming the vanishing gradient problem
 - Dropout to avoid overfitting
 - New initialization methods, e.g. using autoencoders for pre-training
 - Convolution and shared weights (next week)

- **Sigmoid** transfer function; recall the formulas for weight change between neuron p and q:

$$\Delta w_{pq} = \eta \cdot \delta_q \cdot o_p \quad \text{- weight change}$$

$$\text{For output neuron } q: \delta_q = (t_q - o_q) o_q (1 - o_q)$$

$$\text{For hidden neuron } q: \delta_q = o_q (1 - o_q) \sum_i w_{qi} \delta_i$$



- Saturation of outputs
 - The range of the sigmoid function is (0,1)
 - If the NN output o_q is close to 0 or 1, δ_q will be very small – close to 0
 $\delta_q = (t_q - o_q) o_q (1 - o_q)$ – very small weight update
- These small delta values δ_i will be propagated to the previous levels – multiplication – values close to 0; very small weight change

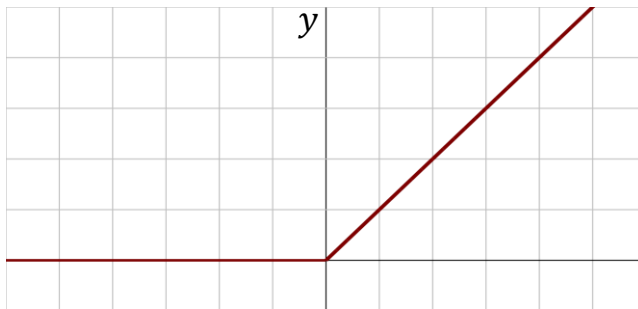
- Especially problematic if there are many hidden layers – diminishing gradients, slow convergence
- Even if the activation in the output layer does not saturate, the repeated multiplications performed as we backpropagate the gradients from the output to the hidden layers may lead to diminishing gradients
- **The vanishing gradient problem:** The weight changes for the lower levels are very small; these layers learn slower than the higher hidden layers
- This has been a major problem in training deep NNs

Solution: using other activation functions

- Piece-wise linear functions
 - **Rectified Linear Unit (ReLU)**
 - **Leaky Rectified Linear Unit (LReLU)**
- No upper bound, no saturation of the output
- Gradient of ReLU is 1 for input >0 – no saturation of hidden nodes in this case

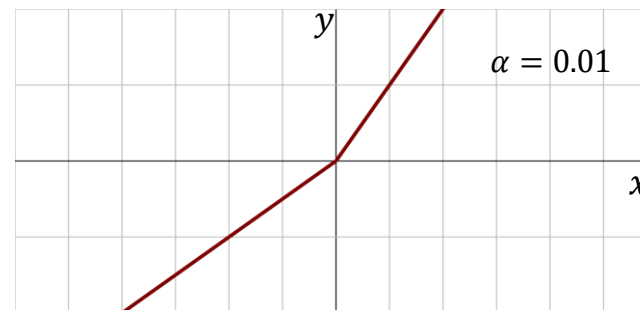
ReLU

$$y = \max(0, x)$$



LReLU

$$y = \max(\alpha x, x)$$



- A method for preventing overfitting
- Idea: Avoid learning spurious features at the hidden nodes – intuition:
 - Relevant features are more resilient to the removal of neurons; they perform well for different combinations of neurons, while spurious features depend only on certain neurons
- Dropout forces the NN to be less dependent on certain neurons, to collect more evidence from other neurons => to be more robust to noise
- During training, at each iteration of the backpropagation, we select randomly neurons in each layer and set their values to 0 (i.e. we drop them out from the weight adjustment = we temporarily disable them)

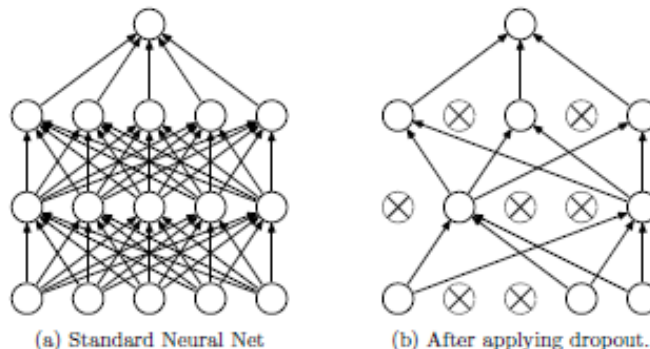
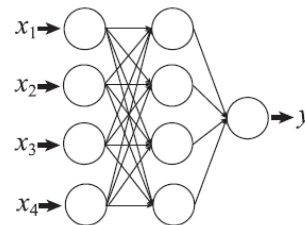


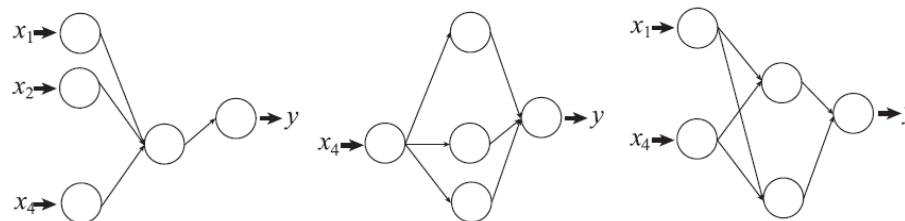
Image from Shrivastava et al. (2014). Dropout: A simple way to prevent neural networks from overfitting,
<https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>

- This results in a “thinned” sub-network of a smaller size
- During training, we update its weights (w , b) using the backpropagation algorithm and then add the new weight values to the original network
- During testing, we do not drop out any neurons; we scale down the weights based on the dropout rate



(a) Original network.

- The dropout rate is a hyperparameter, e.g. $\gamma=0.5$ means that 50% of the neurons will be dropped out



(b) Sub-networks.

Figure 4.30. Examples of sub-networks generated in the dropout method using $\gamma = 0.5$.

Softmax: Converting output values to probabilities

- The NN outputs can be post-processed to turn them into probabilities
- Motivation: interpret the outputs as probabilities that sum up to 1
- Let the NN outputs are $(o_1, \dots, o_n)$; n – number of output neurons
- Softmax transformation: $p_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$
- Example: 3 output neurons with values: $o_1=0.3$, $o_2=0.8$ and $o_3=0.2$
- Applying softmax: $p_1=0.28$, $p_2=0.46$, $p_3=0.26$
- Checking: $0.28+0.46+0.26=1$

- Using **cross entropy loss** instead of MSE
- Classification task, one-hot encoding to represent class labels
 - C – number of classes, N - number of examples
 - y_i – a one-hot encoded class of example i (C-dimensional)
 - $\hat{y}_i$ - predicted class (C-dimensional vector)

- Categorical cross entropy loss for the classification of example i :

$$CCE_i = - \sum_{j=1}^C y_{ij} \cdot \log \hat{y}_{ij}$$

- Cross entropy loss function - sum of losses of individual examples:

$$CCE = \sum_{i=1}^N CCE_i$$

- Perceptrons form linear decision boundaries
- Multi-layer perceptrons trained with the backpropagation algorithm can form arbitrary decision boundaries - both linear and non-linear
- However, they require careful tuning as they can get stuck in a bad local minimum
- Their performance is sensitive to the:
 - starting conditions (weights initialization)
 - architecture (number of hidden layers and neurons
 - Too few neurons – underfitting, unable to learn what you want it to learn
 - Too many – overfitting, learns slowly
 - other hyperparameters such as: learning rate (too small – slow convergence, too big – oscillations), momentum, number of training epochs
- Modern neural networks (especially deep NNs - networks with many hidden layers) use techniques such as ReLu activation functions to reduce the vanishing gradient problem, dropout to reduce overfitting, better initialization of weights and more sophisticated optimisers (variations of the backpropagation algorithm)