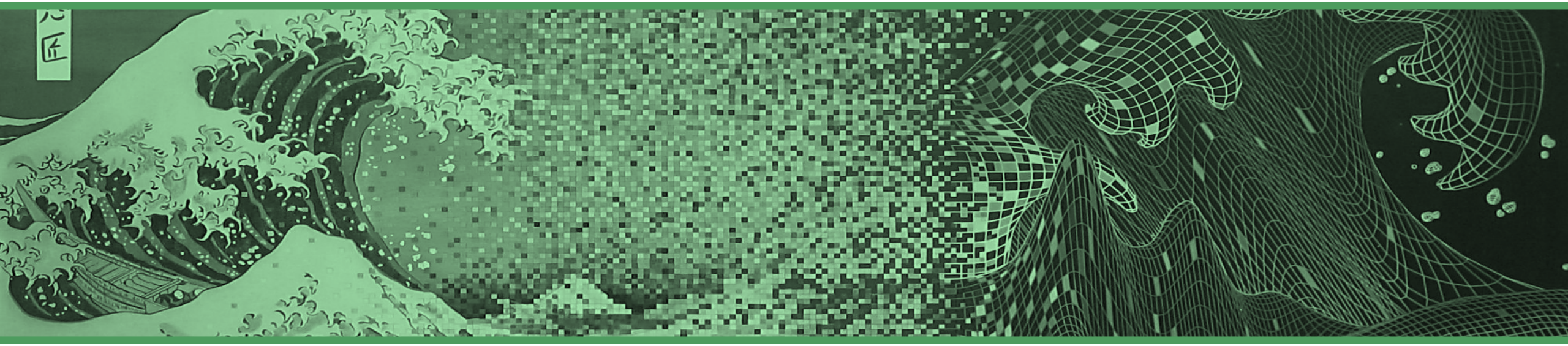
TÉCNICO LISBOA

Multilayer perceptron

Backpropagation and Neural Processing



Aprendizagem 2024

Miguel Couceiro miguel.j.couceiro@tecnico.ulisboa.pt
Andreas Wischert andreas.wichert@tecnico.ulisboa.pt

Outline

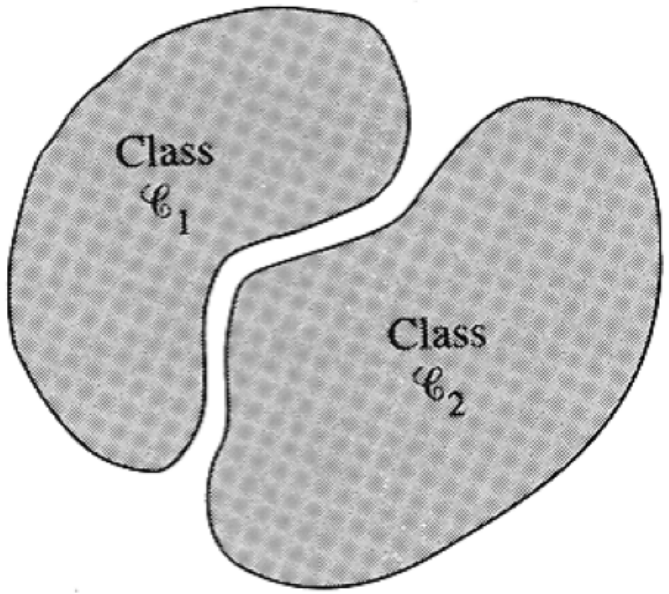
- **Multi-layer perceptron**
  - non-linearity
  - gradient descent
- **Propagation**
- **Backpropagation**
  - network updates
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- **Learning convergence**
  - optimality
  - early stopping

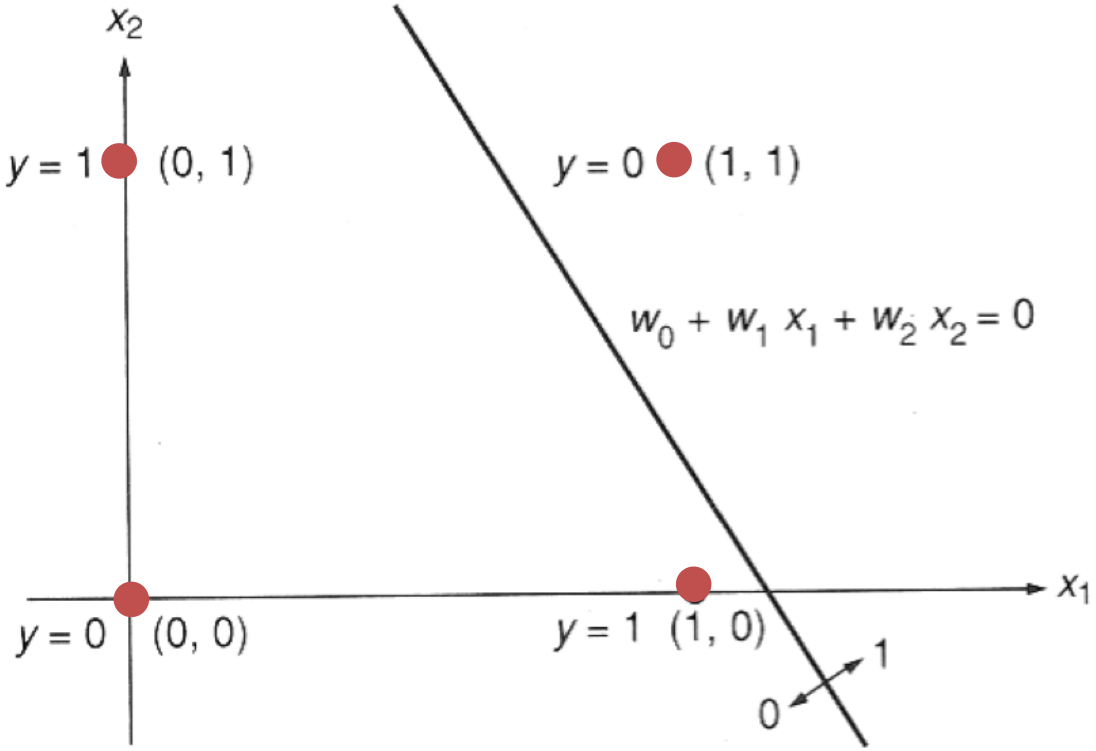
Outline

- **Multi-layer perceptron**
  - **non-linearity**
  - gradient descent
- Propagation
- Backpropagation
  - network updates
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- Learning convergence
  - optimality
  - early stopping

Perceptron limitations

- Designed for binary classification $|Z| = 2$
  - how can we extend for other predictive tasks?
- Linear separation only (the XOR problem)

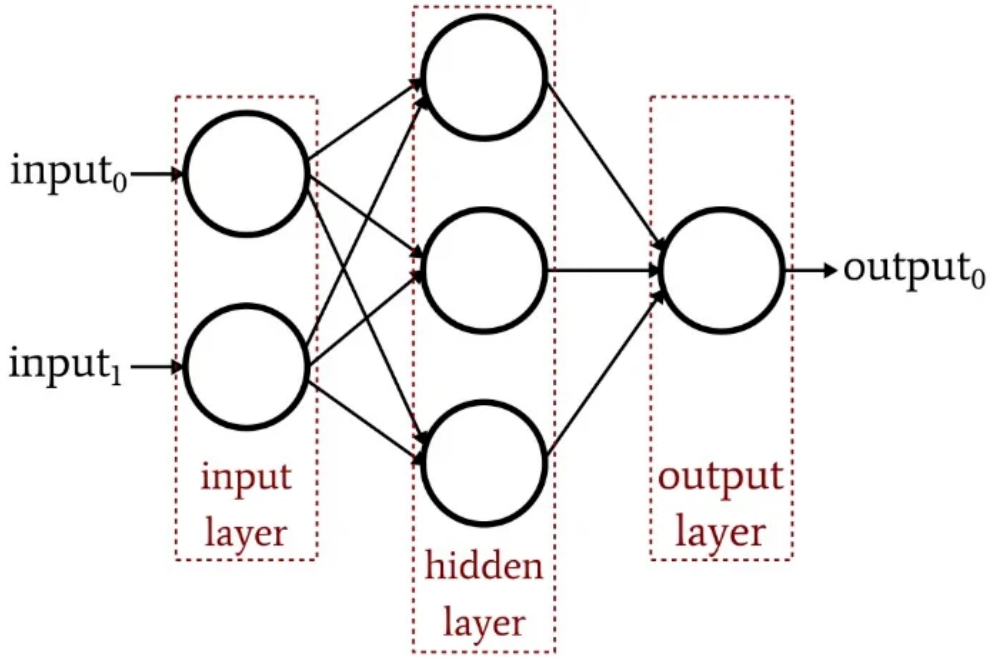




Non-linear problems

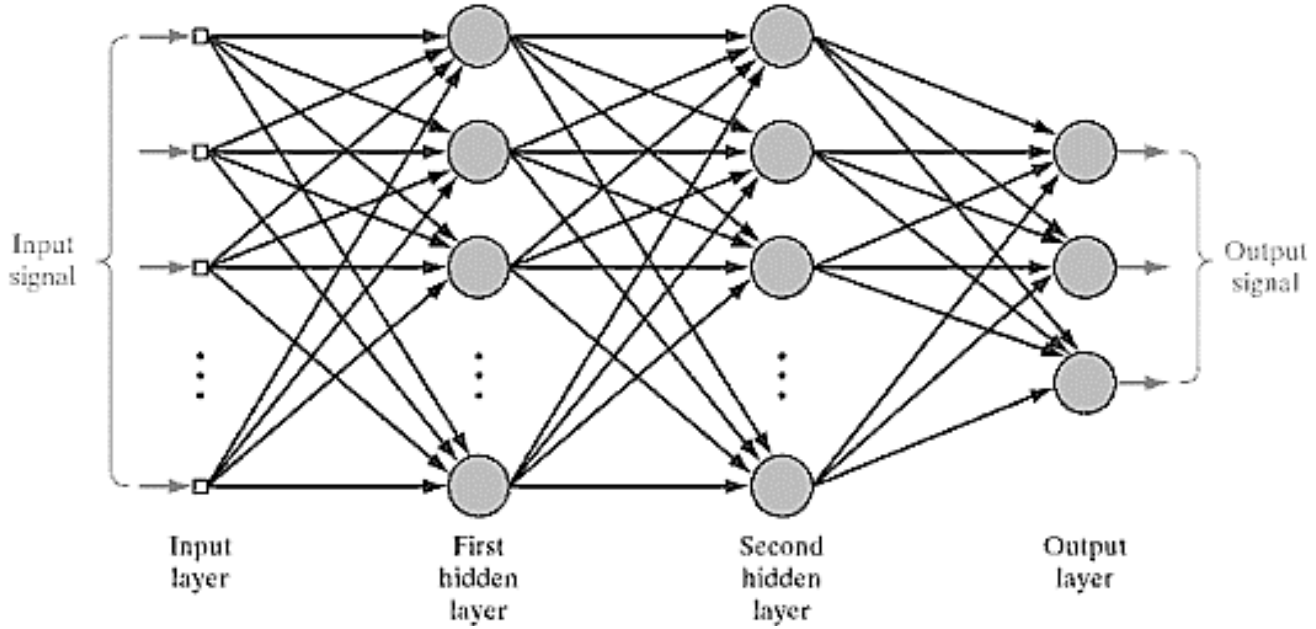
- In mathematics, we can compose ***activations*** and ***linear*** functions, $f(x)$ and $g(x)$, to form a ***non-linear*** function, $h(f(x), g(x))$, to model more complex behaviors
- What if we combine multiple perceptrons? Are we able to solve the XOR problem?
  - YES!

- We can create an intermediate or hidden layer of perceptrons
  - a network with a single hidden unit can already represent any Boolean function!



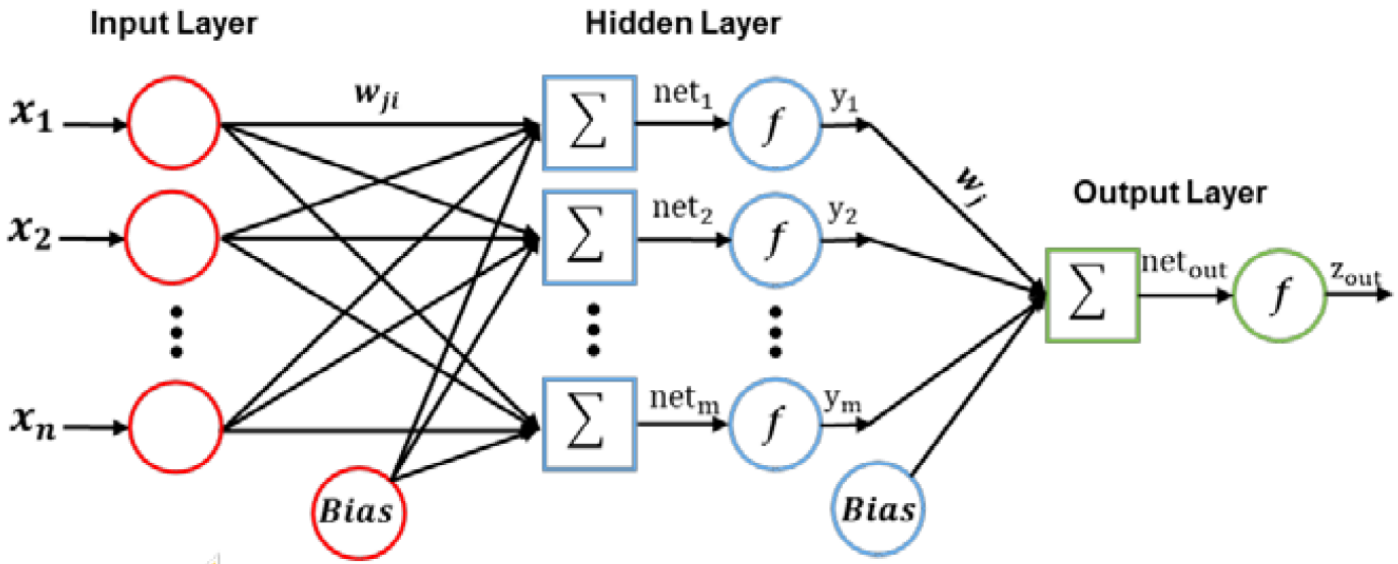
Multi-layer networks

- The composition of perceptrons can be **organized in layers**
  - the outputs of perceptrons in one layer feed the input of perceptrons in the next layer
    - hence **multi-layer perceptrons** (MLPs) are also termed **feed forward networks**
    - a simplified architecture to facilitate the learning
    - the presence of cycles (e.g. two perceptrons feeding each other) brings complexities
  - the first and last layers are referred as **input** and **output** layers



Non-linear models

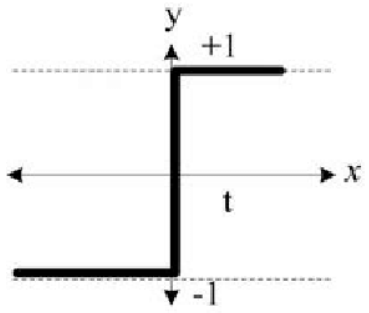
- **Recall:** multiple layers of cascade linear units still produce only linear functions: $z_{out} = \sum w \left(\sum w \, x_i \right)$



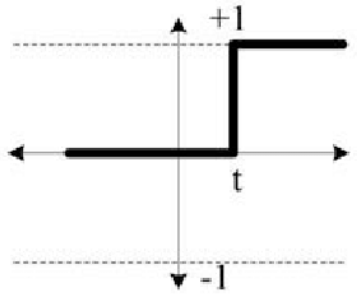
Solution: $z_{out} = f\left(\sum w \, f\left(\sum w \, x_i \right) \right)$

Activation functions

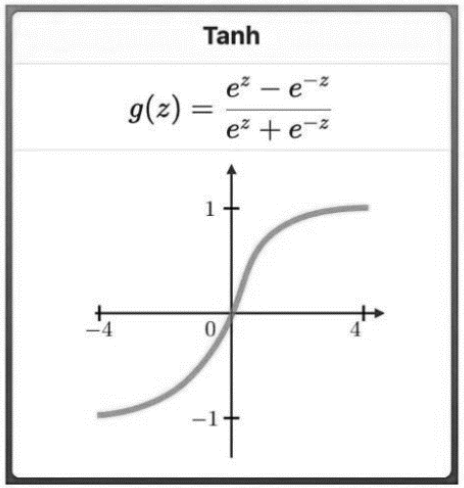
- networks able of representing nonlinear functions:
  - use nonlinear activation functions
  - activations should be continuous and differentiable
  - problems of sign and step?

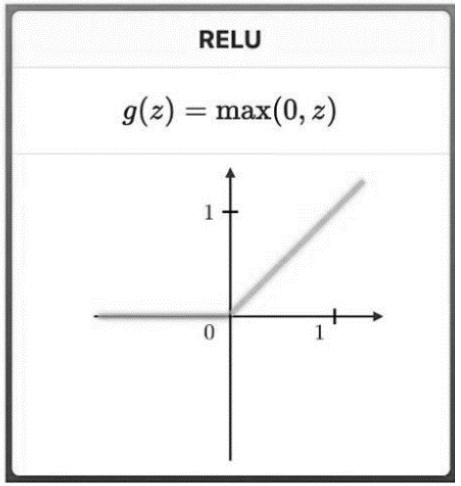


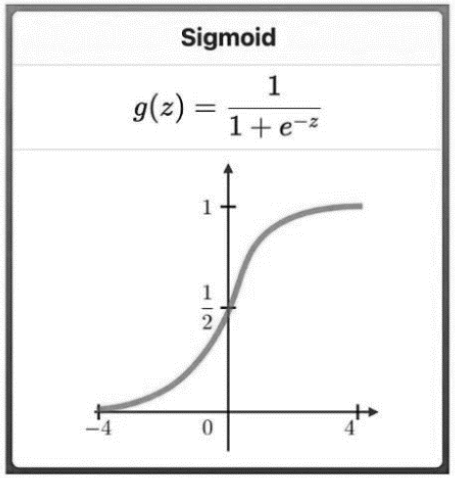
Sign Function

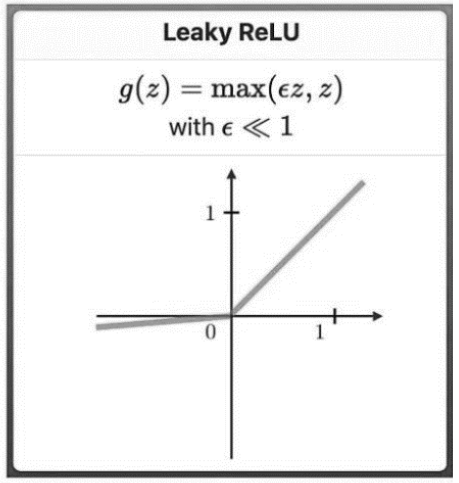


Step Function

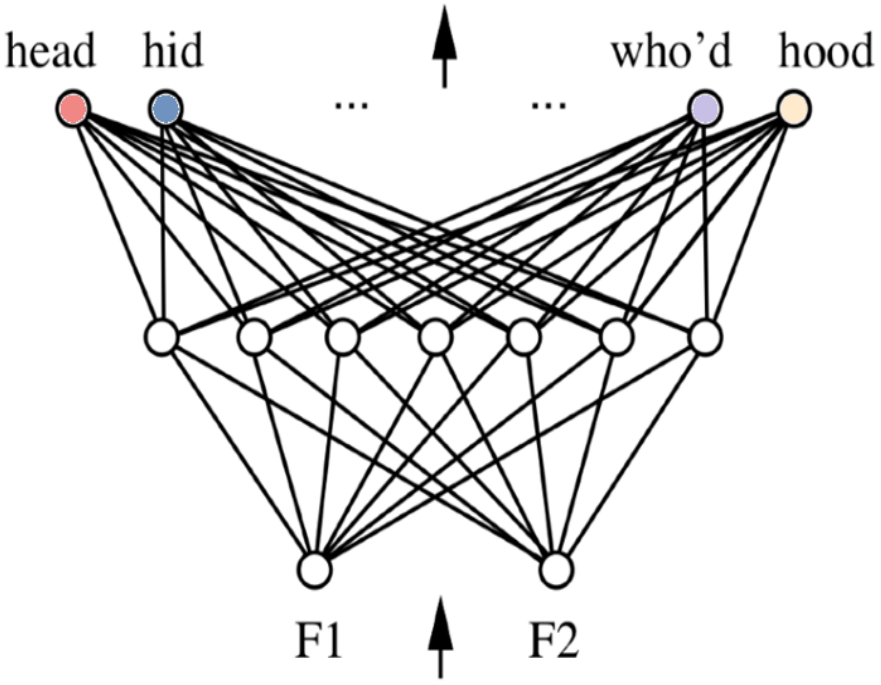


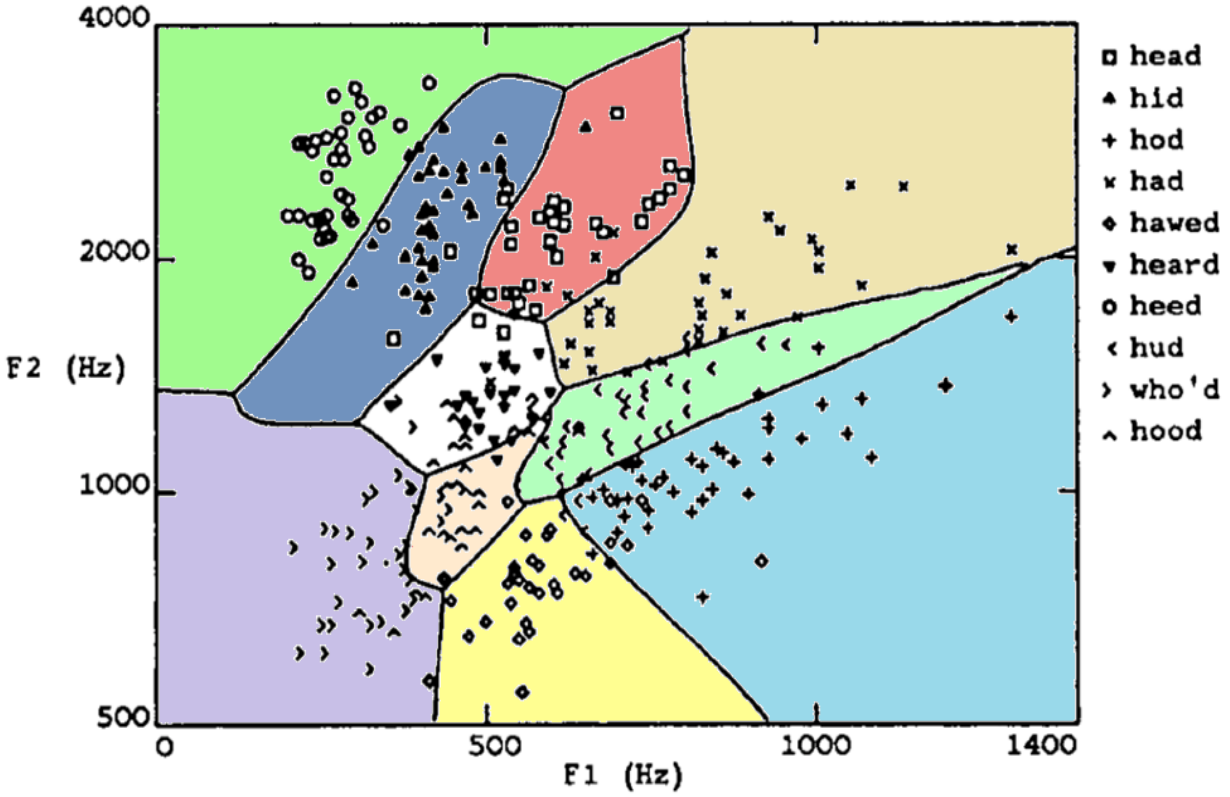




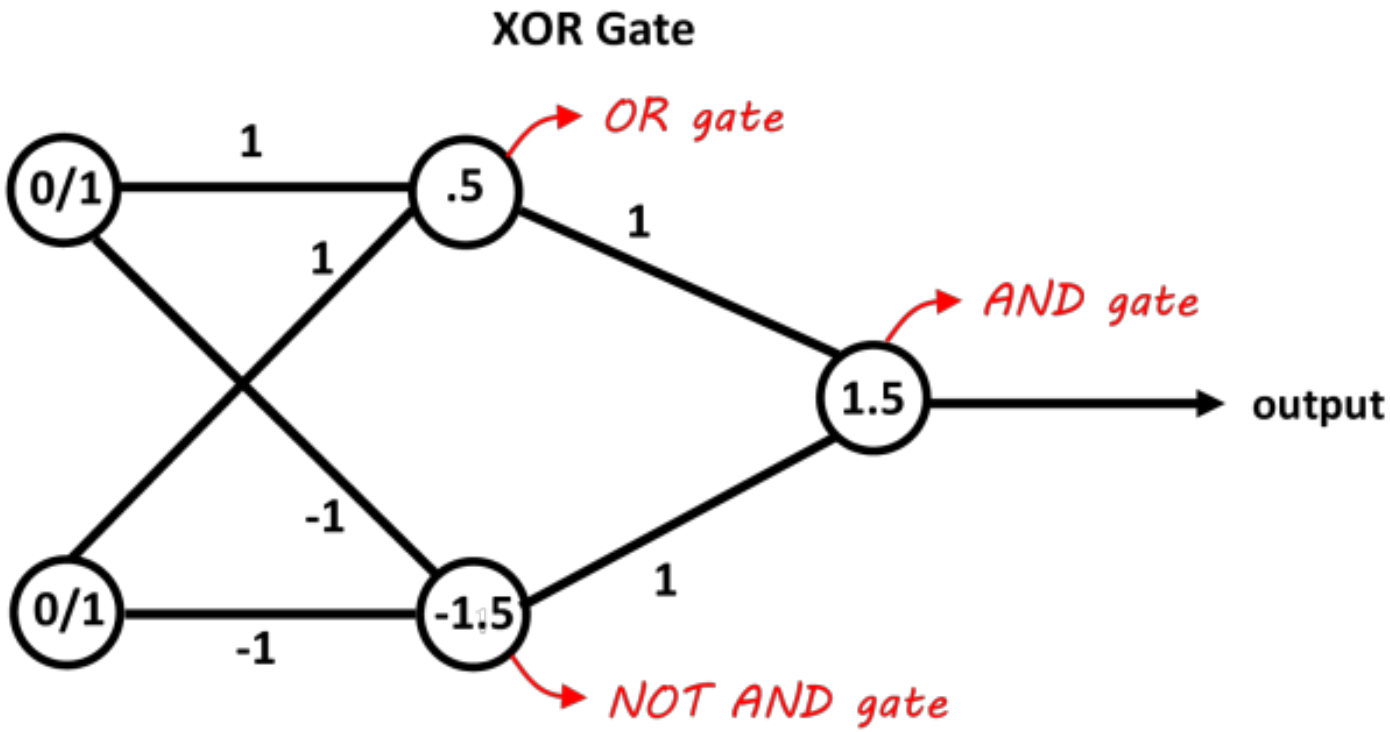


Non-linear boundaries





Solving the XOR problem

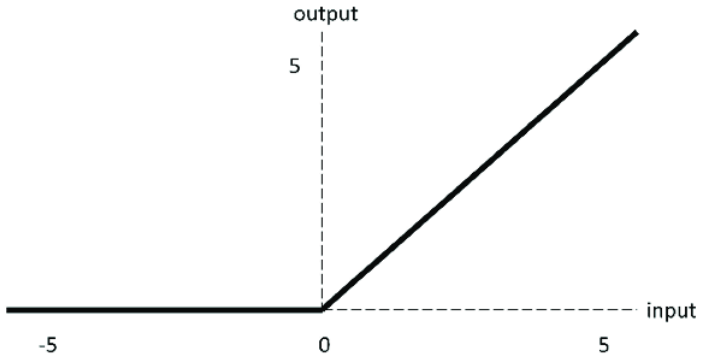


- *Exercise*: place the necessary activation functions on the perceptrons to yield the desired result

MLPs for regression and classification

- Until now… a single perceptron can only answer a binary classification task
- Neural networks can be extended to handle more general predictive tasks
  - *multiclass classification*
  - ***regression***
    - replace the last sign/sigmoid/tanh activation by other function – e.g. [rectified] linear unit
  - ***multiple-output prediction***
    - multiple targets

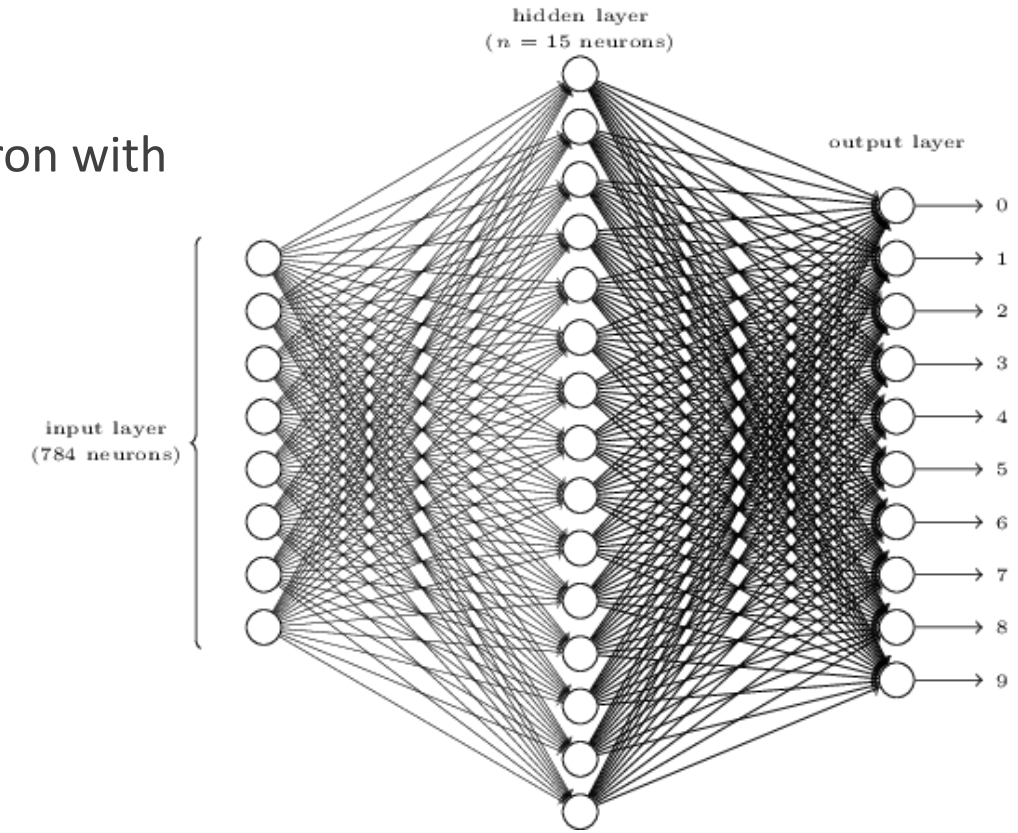
      (e.g., autonomous driving – speed and direction)



- What about the learning? The same principles apply as we will see later

Multi-class classification

- Many classification tasks have high cardinality
  - e.g. document categorization, product recommendation, character recognition
- Given a set L of class labels
  - $|L|$ output nodes, one per class
  - predicted class is the output neuron with the higher value



Outline

- **Multi-layer perceptron**
  - non-linearity
  - **gradient descent**
- Propagation
- Backpropagation
  - network updates
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- Learning convergence
  - optimality
  - early stopping

Learning multi-layer networks

- The great power of multi-layer networks was realized a long ago
  - yet only in the 80s it was shown how to make them learn!

- **Backpropagation** is the most used learning algorithm for multi-layer neural networks
  - invented independently several times
    - Bryson an Ho [1969]
    - Werbos [1974]
    - Parker [1985]
    - Rumelhart et al. [1986]
      - See Parallel Distributed Processing - Vol. 1, Foundations

Backpropagation

- **Goal**
  - given a training set of input-output pairs $\{\mathbf{x}_i, t_i\}$
  - learn the parameters of the network, w_{ij}

- **How?**
  - finding the weights that minimize the loss
    - back to the usual error function to assess our network

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(t^{(i)} - o^{(i)}\right)^2$$

    - for L outputs and n input-output pairs $\{\mathbf{x}_i, \mathbf{t}_i\}$

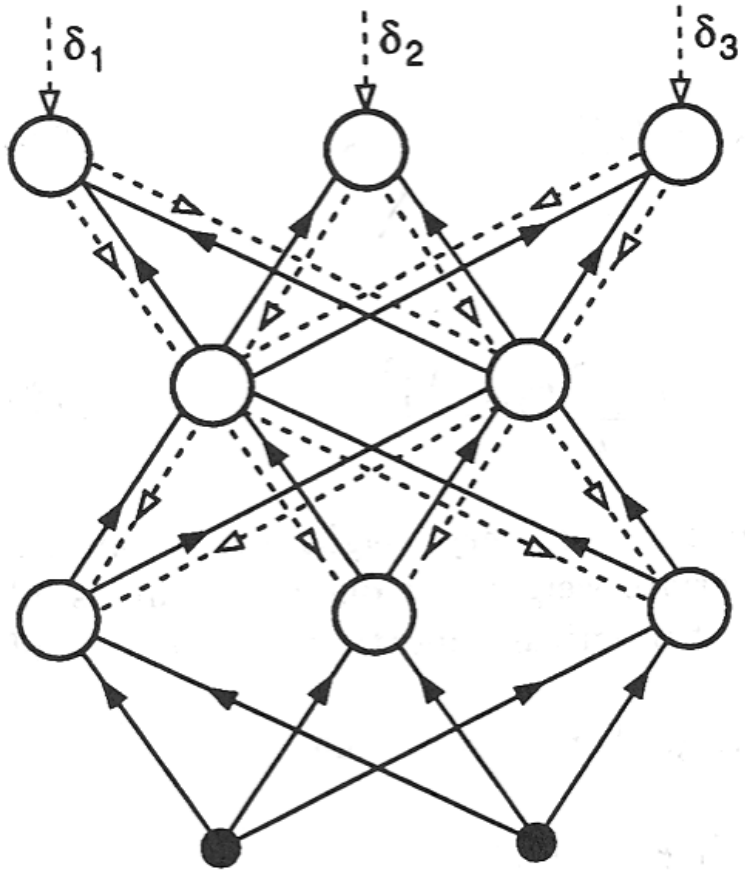
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{l=1}^{L}\left(t_l^{(i)} - o_l^{(i)}\right)^2$$

Backpropagation

- How do we find the weights that minimize the given loss?
  - gradient descent! Adjust the weights in the direction where the error decreases

$$\Delta w_{ij} = -\eta \frac{\partial E(\mathbf{w})}{\partial w_{ij}}$$

  - two major steps:
    - *forward* **propagation** of inputs until the end of the network
    - compute the observed errors δ and propagate them *backwards* (hence **backpropagation**)

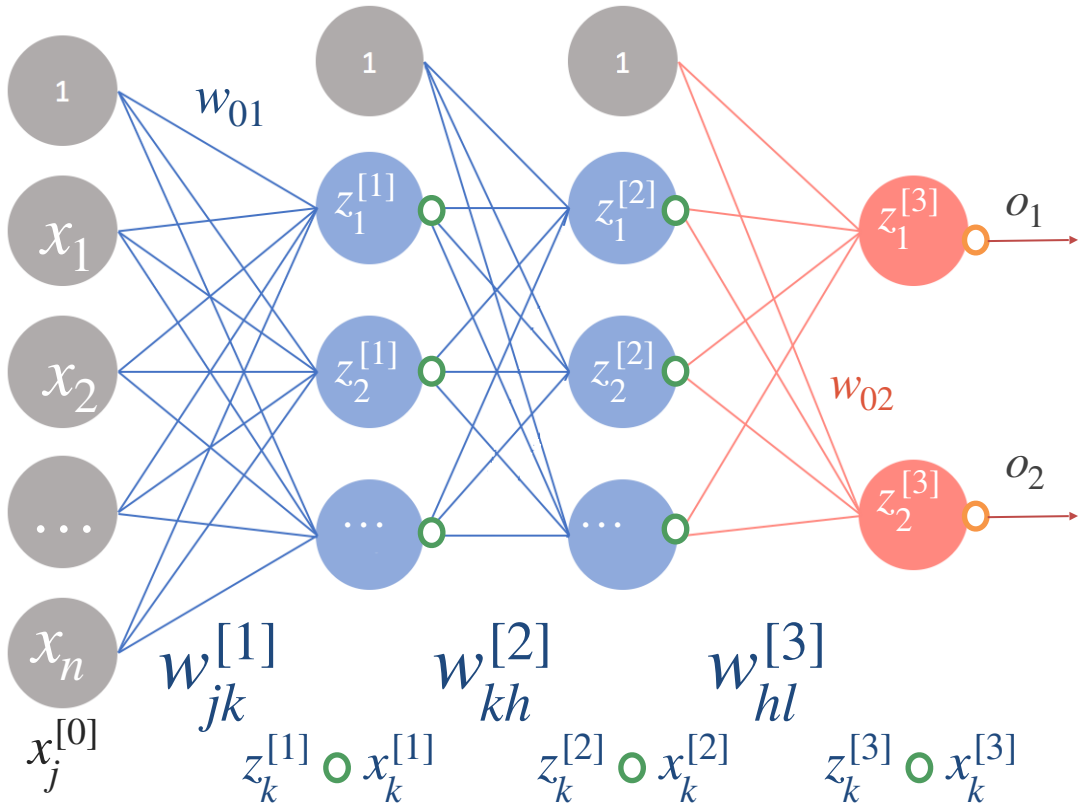


Outline

- Multi-layer perceptron
  - non-linearity
  - task formulations
- **Propagation**
- Backpropagation
  - gradient descent updates
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- Learning convergence
  - optimality
  - early stopping

Notation

- network with P **layers**, $p = 1,2,\ldots,P$
- $z_k^{[p]}$ is the **net value** of the kth node of the pth layer
- $\phi^{[p]}$ is the activation function of the pth layer
- $x_k^{[p]}$ is the **signal** of the kth node of the pth layer
  - $x_k^{[0]}$ is just x_k, i.e. the **input feature**
  - $x_k^{[P]}$ is just the output o_k, i.e. the kth **network output**
- $w_{ij}^{[p]}$ is the **weight** of the connection from ith node of the $(p-1)$th layer to the jth node of the pth layer
- **bias** is the product of $x_0^{[p]}$ and $w_{0k}^{[p]}$
  - similarly to the perceptron and linear regression, we consider a bias term to aid the learning
  - the bias is present on *every* layer



Propagation

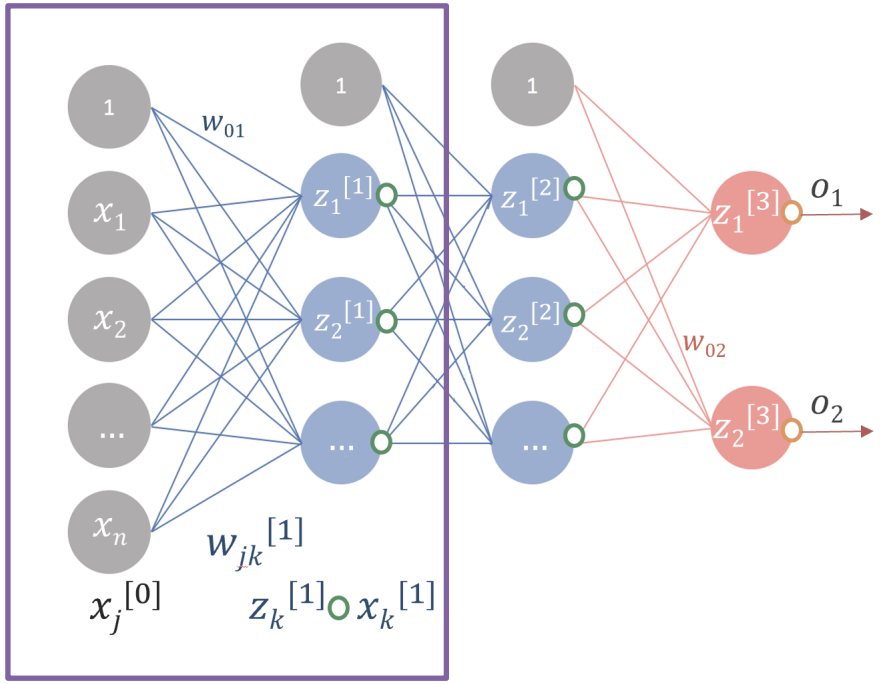
- Let us consider a simple three-layered network
- Given the observation $\mathbf{x}$ the net values of the first layer

$$net_k^{[1]} = z_k^{[1]} = \sum_{j=0}^{m} w_{jk}^{[1]} x_j^{[0]} = \sum_{j=0}^{m} w_{jk}^{[1]} x_j$$

NB: $\forall p \in \{0,\ldots,P\}, w_{0k}^{[p]} x_0^{[p-1]} = w_{0k}^{[p]}$ is a constant term!

– … and produces the output

$$x_k^{[1]} = f(z_k^{[1]}) = \phi^{[1]}\left(\sum_{j=0}^{m} w_{jk}^{[1]} x_j \right)$$



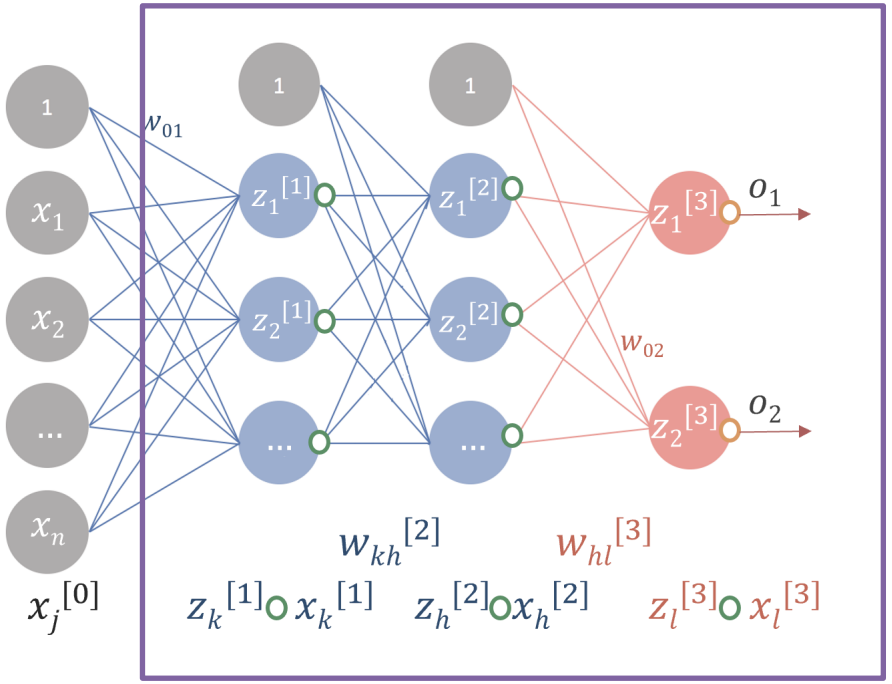
Propagation

– similarly, considering the second layer:

$$x_h^{[2]} = \phi^{[2]}\left(\sum_{k=0}^{m^{[1]}} w_{kh}^{[2]} x_k^{[1]}\right) = \phi^{[2]}\left(\sum_{k=0}^{m^{[1]}} w_{kh}^{[2]} \phi^{[1]}\left(\sum_{j=0}^{m} w_{jk}^{[1]} x_j\right)\right)$$

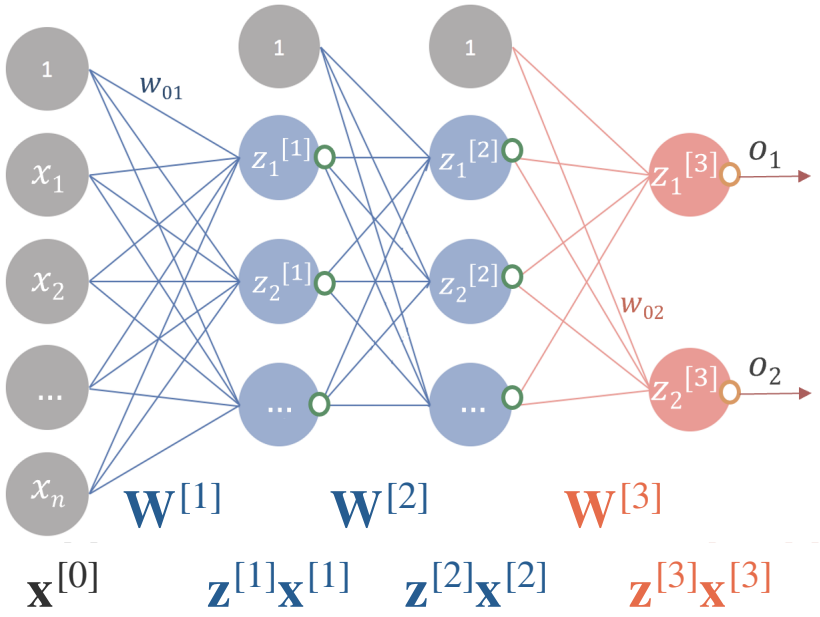
– and the output layer…

$$o_l = x_l^{[3]} = \phi^{[3]}(z_l^{[3]}) = \phi^{[3]}\left(\sum_{h=0}^{m^{[2]}} w_{hl}^{[3]} x_h^{[2]}\right) = \phi^{[3]}\left(\sum_{h=0}^{m^{[2]}} w_{hl}^{[3]}(\dots)\right)$$



Vector notation

- network with P **layers**, $p = 1,2,..,P$
- **parameters:**
  - $\mathbf{W}^{[p]}$ contains the **weights** connecting (p-1)th layer and pth layer
  - either $\mathbf{W}^{[p]}$ contains all weights or **biases** are isolated in $\mathbf{b}^{[p]}$
- variables:
  - $\mathbf{z}^{[p]}$ are the **net values** of the pth layer
  - $\mathbf{x}^{[p]}$ is the activated/output values of the pth layer, $\mathbf{x}^{[p]} = \phi^{[1]}\left(\mathbf{z}^{[p]}\right)$
    - $\mathbf{x}^{[0]}$ is a synonym for $\mathbf{x}$ (input **observation**)
    - $\mathbf{x}^{[P]}$ is a synonym for $\mathbf{o}$, i.e., the network **output**



Propagation

- Consider the given network, assuming $m = 4$ and hidden layers with 3 nodes each
- The **parameters** of the given neural network:
  - $\mathbf{W}^{[1]} = \begin{pmatrix} w11^{[1]} & \dots & w41^{[1]} \\ \dots & & \dots \\ w13^{[1]} & \dots & w43^{[1]} \end{pmatrix}, \mathbf{W}^{[2]} = \begin{pmatrix} w11^{[2]} & \dots & w31^{[2]} \\ \dots & & \dots \\ w13^{[2]} & \dots & w33^{[2]} \end{pmatrix}, \mathbf{W}^{[3]} = \begin{pmatrix} w11^{[3]} & w21^{[3]} & w31^{[3]} \\ w12^{[3]} & w22^{[3]} & w32^{[3]} \end{pmatrix}$
  - $\mathbf{b}^{[1]} = \begin{pmatrix} b_1^{[1]} \\ \dots \\ b_3^{[1]} \end{pmatrix}, \quad \mathbf{b}^{[2]} = \begin{pmatrix} b_1^{[2]} \\ \dots \\ b_3^{[2]} \end{pmatrix}, \mathbf{b}^{[3]} = \begin{pmatrix} b_1^{[3]} \\ b_2^{[3]} \end{pmatrix}$
- The **nodes** of the neural network
  - $\mathbf{x} = \mathbf{x}^{[0]} = \begin{pmatrix} x_1 \\ \dots \\ x_4 \end{pmatrix}, \quad \mathbf{x}^{[1]} = \phi^{[1]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) = \begin{pmatrix} x_1^{[1]} \\ \dots \\ x_3^{[1]} \end{pmatrix}, \quad \mathbf{x}^{[2]} = \phi^{[2]}(\mathbf{W}^{[2]}\,\mathbf{x}^{[1]} + \mathbf{b}^{[2]}) = \begin{pmatrix} x_1^{[2]} \\ \dots \\ x_3^{[2]} \end{pmatrix}$
  - $o = \mathbf{x}^{[3]} = \phi^{[3]}(\mathbf{W}^{[3]}\mathbf{x}^{[2]} + \mathbf{b}^{[3]}) = \begin{pmatrix} o_1 \\ o_2 \end{pmatrix}$

Propagation: example

- Considering all weights initialized at 0.1, no biases, and sigmoid σ activation function
  What is the output for observation $\mathbf{x}^T = (1\ 1\ 0\ 0)$?

  - $\mathbf{W}^{[1]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{W}^{[2]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{W}^{[3]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{b}^{[1]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b}^{[2]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b}^{[3]} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$
  - $\mathbf{x}^{[1]} = \phi^{[1]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right) = \sigma \begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix},$
  - $\mathbf{x}^{[2]} = \phi^{[2]}(\mathbf{W}^{[2]}\mathbf{x}^{[1]} + \mathbf{b}^{[2]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} \sigma(0.2) \\ \sigma(0.2) \\ \sigma(0.2) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right) = \sigma \begin{pmatrix} 0.165 \\ 0.165 \\ 0.165 \end{pmatrix}$
  - $o = \mathbf{x}^{[3]} = \phi^{[3]}(\mathbf{W}^{[3]}\mathbf{x}^{[2]} + \mathbf{b}^{[3]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} \begin{pmatrix} \sigma(0.165) \\ \sigma(0.165) \\ \sigma(0.165) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) = \sigma \begin{pmatrix} 0.\ 162 \\ 0.162 \end{pmatrix} = \begin{pmatrix} 0.54 \\ 0.54 \end{pmatrix}$

Outline

- Multi-layer perceptron
  - non-linearity
  - gradient descent
- Propagation
- **Backpropagation**
  - **network updates**
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- Learning convergence
  - optimality
  - early stopping

Loss: sum of squared errors

- Recall our goal: learn the parameters of the network
  - optimize weights $\mathbf{w}$ – matrices $\mathbf{W}^{[p]}$ and biases $\mathbf{b}^{[p]}$ for $p \in \{1,2,\ldots,P\}$
- **How?**
  - minimize error $E(\mathbf{w})$ to estimate $\mathbf{w}$! Back to our sum of squared errors (SSE)

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\left(t^{(i)} - o^{(i)}\right)^2$$

  - for L outputs and n input-output pairs $\{\mathbf{x}_i, \mathbf{t}_i\}$

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{l=1}^{L}\left(t_l^{(i)} - o_l^{(i)}\right)^2$$

  - for our previous example:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{l=1}^{2}\left(t_l^{(i)} - \phi^{[3]}\left(\sum_{h=1}^{3} w_{hl}^{[3]}\phi^{[2]}\left(\sum_{k=1}^{3} w_{kh}^{[2]}\phi^{[1]}\left(\sum_{j=1}^{m} w_{jk}^{[1]}x_j^{(i)}\right)\right)\right)\right)^2$$

Backpropagation

- $E(\mathbf{w})$ is differentiable if $\phi^{[p]}$ are differentiable
  - **Gradient Descent** can be applied!
  - similarly to what we did for the perceptron, we can infer the update rules for a MLP
- Yet before advancing a central tool – **chain rule** – to differentiate composite functions:

$$\frac{\partial f(g(x))}{\partial x} = f'\big(g(x)\big)\frac{\partial g(x)}{\partial x} = f'\big(g(x)\big)g'(x)$$

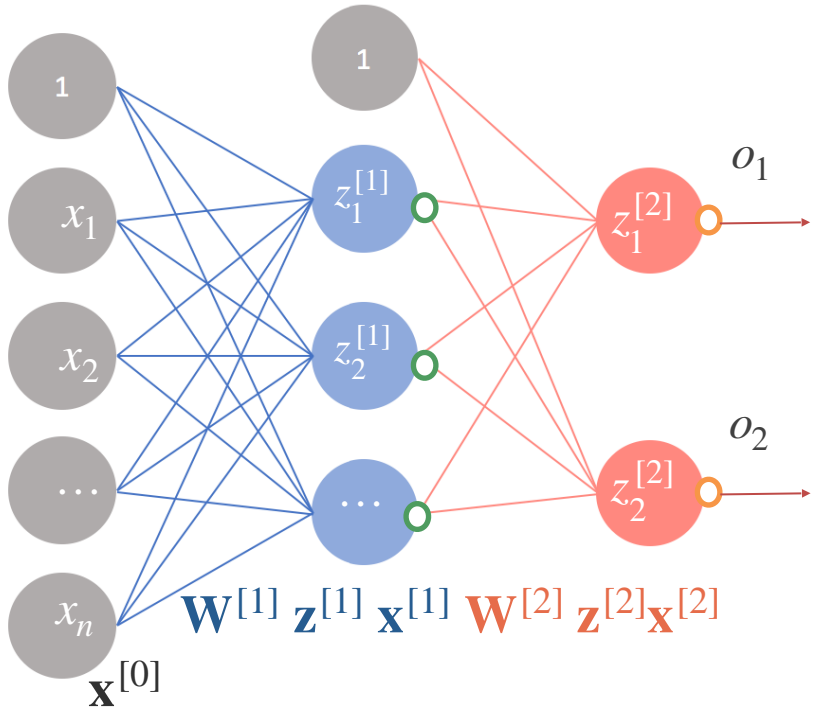
$$\nabla_{\mathbf{x}} f\big(g(\mathbf{x})\big) = \frac{\partial f(g(\mathbf{x}))}{\partial \mathbf{x}} = \frac{\partial f(g(\mathbf{x}))}{\partial g} \bullet \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}$$

  - example:

$$\frac{\partial \exp(3x^2)}{\partial x} = \frac{\exp(f(x))}{\partial f}\frac{\partial\left(3x^2\right)}{\partial x} = \exp\left(3x^2\right) \times 6x$$

Backpropagation

- To learn a MLP we iteratively select observations and:
  - (forward) propagation of observation values
    - $\mathbf{x}^{[p]} = \phi^{[p]}(\mathbf{W}^{[p]}\mathbf{x}^{[p-1]} + \mathbf{b}^{[p]})$
  - backpropagation of the errors
    - compute $\boldsymbol{\delta}^{[p]}$ differences to expectations from last to first layer
  - update the weights
    - $\mathbf{W}^{[p]} = \mathbf{W}^{[p]} - \eta \frac{\partial E}{\partial \mathbf{W}^{[p]}}$ where $\frac{\partial E}{\partial \mathbf{W}^{[p]}} = \boldsymbol{\delta}^{[p]}(\mathbf{x}^{[p-1]})^T$
    - $\mathbf{b}^{[p]} = \mathbf{b}^{[p]} - \eta \frac{\partial E}{\partial \mathbf{b}^{[p]}}$ where $\frac{\partial E}{\partial \mathbf{b}^{[p]}} = \delta^{[p]}$



Backpropagation

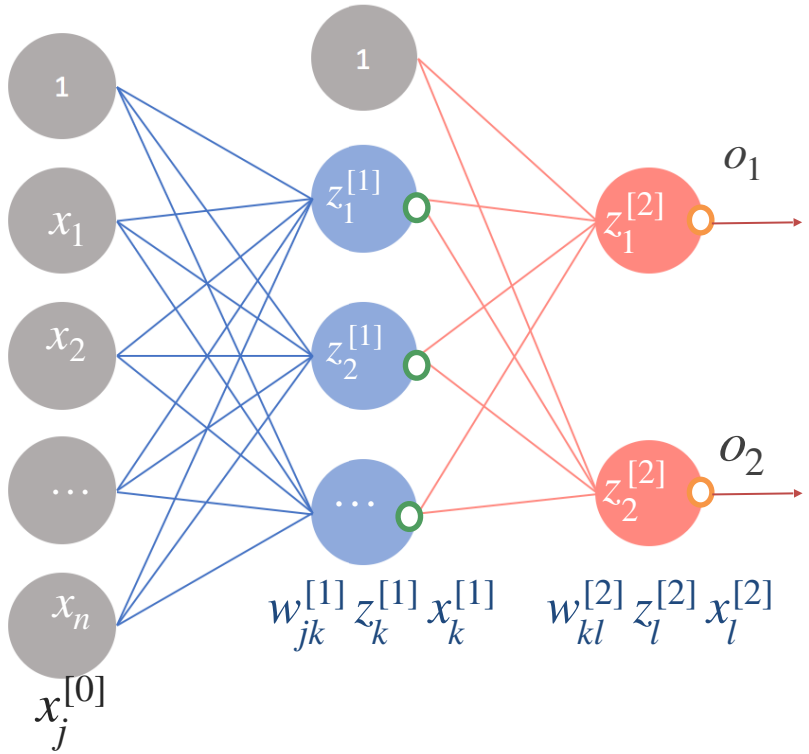
- To compute $\boldsymbol{\delta}^{[p]}$, we need first to infer the update rule…
- Let us infer the update rules on the right
  - let us start with the **last layer**…

$$\Delta w_{kl}^{[2]} = -\eta \frac{\partial E}{\partial w_{kl}^{[2]}} = -\eta \frac{\partial}{\partial w_{kl}^{[2]}} \sum_{i=1}^{n} \left(t_l^{(i)} - o_l^{(i)} \right)^2$$

$$= -\eta \sum_{i=1}^{n} \left(t_l^{(i)} - O_l^{(i)} \right) \cdot \left(-\phi'^{[2]}(z_l^{(i)[2]}) \right) \cdot x_k^{(i)[1]})$$

$$= \eta \sum_{i=1}^{n} \delta_l^{(i)[2]} \cdot x_k^{(i)[1]}$$

where $\delta_l^{[2]} = (t_l - o_l)\phi'^{[2]}(z_l^{[2]})$



Backpropagation

- … let us continue with the **hidden layer**. Using the chain rule:

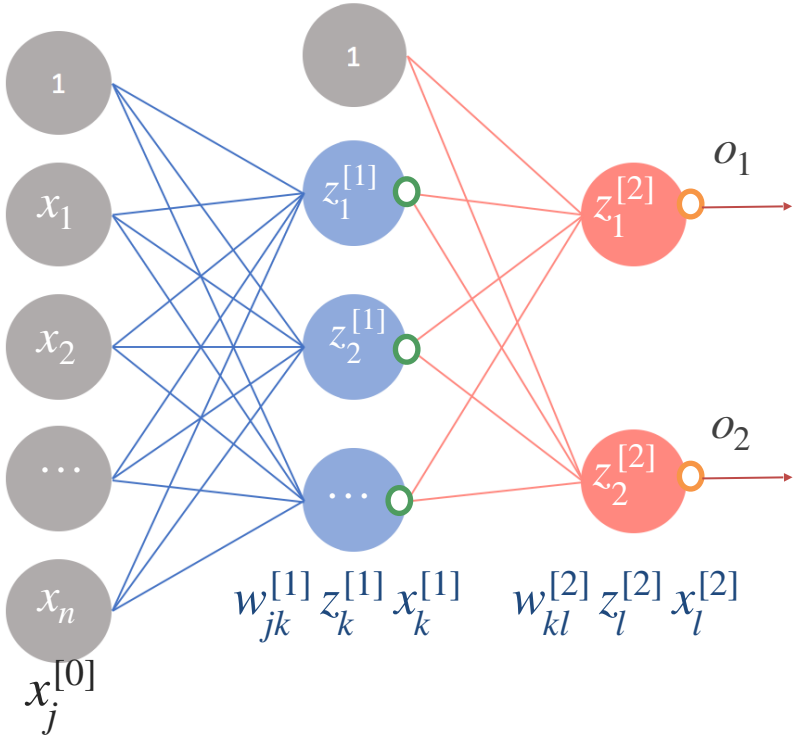
$$\Delta w_{jk}^{[1]} = -\eta \frac{\partial E}{\partial w_{jk}^{[1]}} = -\eta \sum_{i=1}^{n} \frac{\partial E}{\partial xk^{(i)[1]}} \bullet \frac{\partial xk^{(i)[1]}}{\partial w_{jk}^{[1]}}$$

$$\frac{\partial xk^{(i)[1]}}{\partial w_{jk}^{[1]}} = \frac{\partial}{\partial w_{jk}^{[1]}} \left(\phi^{[1]} \left(\sum_{j=1}^{4} w_{jk}^{[1]} xj^{(i)} \right) \right) = \phi'^{[1]} \left(zk^{(i)[1]} \right) \bullet xj^{(i)}$$

$$\frac{\partial E}{\partial xk^{(i)[1]}} = \sum_{l=1}^{2} \delta l^{(i)[2]} w_{kl}^{(i)[2]}$$

$$\Delta w_{jk}^{[1]} = \eta \sum_{i=1}^{n} \delta k^{(i)[1]} \, x_j^{(i)}$$

where $\delta_k^{[1]} = \phi'^{[1]}(z_k^{[1]}) \sum_{l=1}^{2} \delta_l^{[2]} w_{kl}^{[2]}$



Backpropagation

- In general, with an **arbitrary number of layers**, the back-propagation **update rule** has the form

$$\Delta w_{jk}^{[p]} = \eta \sum_{i=1}^{n} \delta_k^{(i)[p]} x_j^{(i)[p-1]}$$

  - where the δ of the last layer is given by

$$\delta_k^{[P]} = (t_k - o_k)\phi'^{[P]}(z_k^{[P]})$$

  - and the remaining δ given by

$$\delta_k^{[p]} = \phi'^{[p]}(z_k^{[p]}) \sum_{h=1}^{m^{[p+1]}} \delta_h^{[p+1]} w_{kh}^{[p+1]}$$

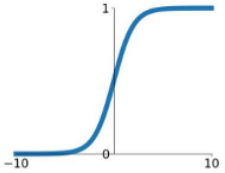
- The deltas essentially propagate the difference between expected and observed outputs with correction factors – defining the strength to change weights

Activation functions

- we have to use a *differentiable* activation functions $\phi^{[p]}$
  - sigmoid, hyperbolic tangent and rectified linear unit (ReLU) are common options
  - we can use different activations for different layers (e.g. $\phi^{[p]} \neq \phi^{[p+1]}$)
  - yet the nodes within the same layer generally have the same activation function

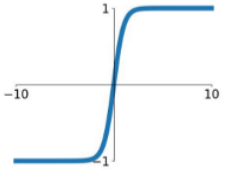
Sigmoid

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



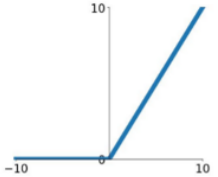
tanh

$\tanh(x)$



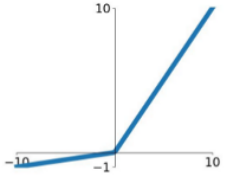
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

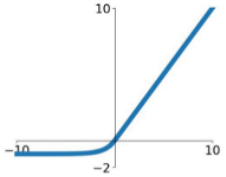


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Thank You

miguel.j.couceiro@tecnico.ulisboa.pt
andreas.wichert@tecnico.ulisboa.pt