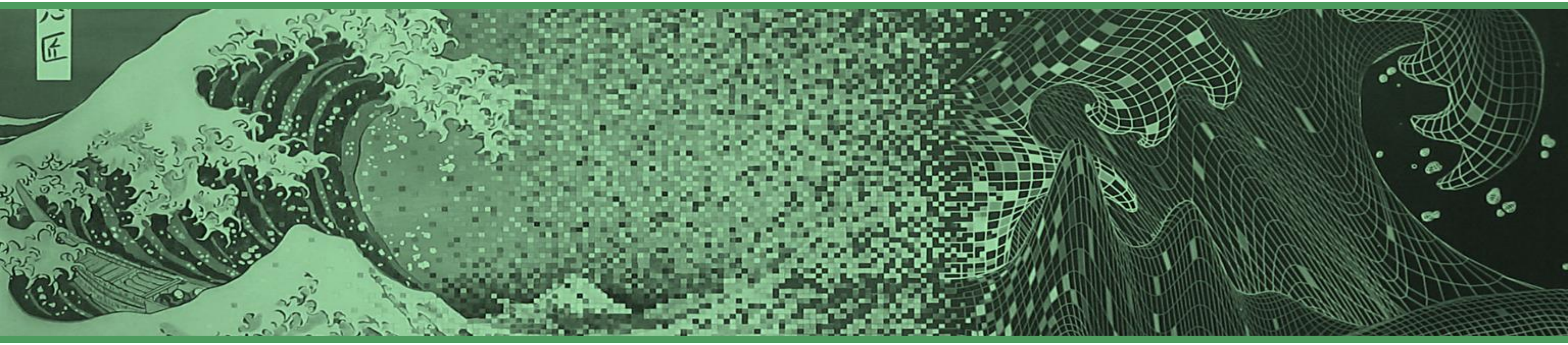
TÉCNICO LISBOA

Multilayer perceptron

Backpropagation and Neural Processing



Aprendizagem 2023

Rui Henriques rmch@tecnico.ulisboa.pt
Andreas Wischert andreas.wichert@tecnico.ulisboa.pt

Outline

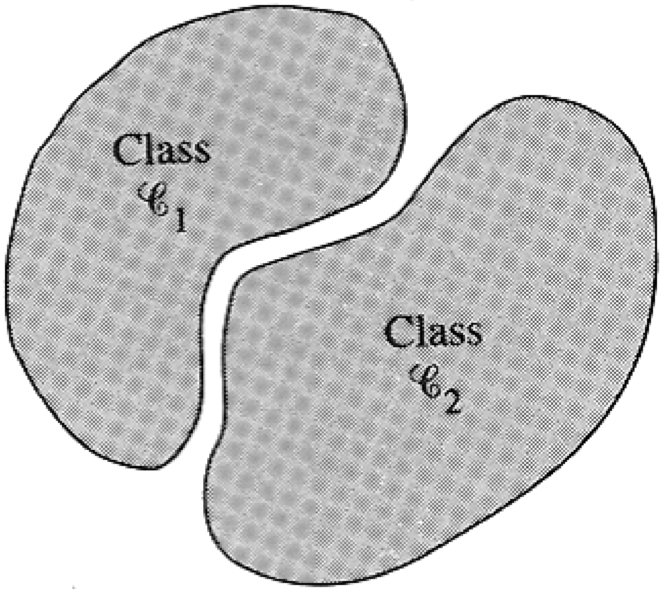
- **Multi-layer perceptron**
  - non-linearity
  - gradient descent
- **Propagation**
- **Backpropagation**
  - network updates
  - tensor operations
  - batch *versus* stochastic updates
  - cross-entropy
- **Learning convergence**
  - optimality
  - early stopping

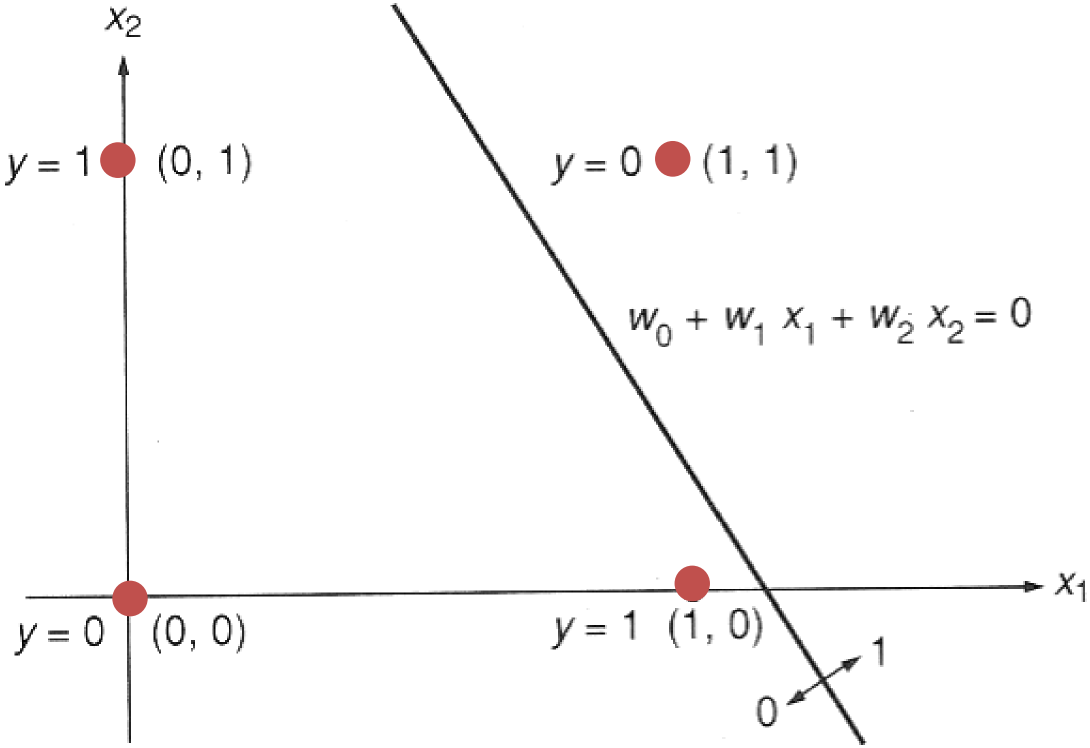
Outline

- **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

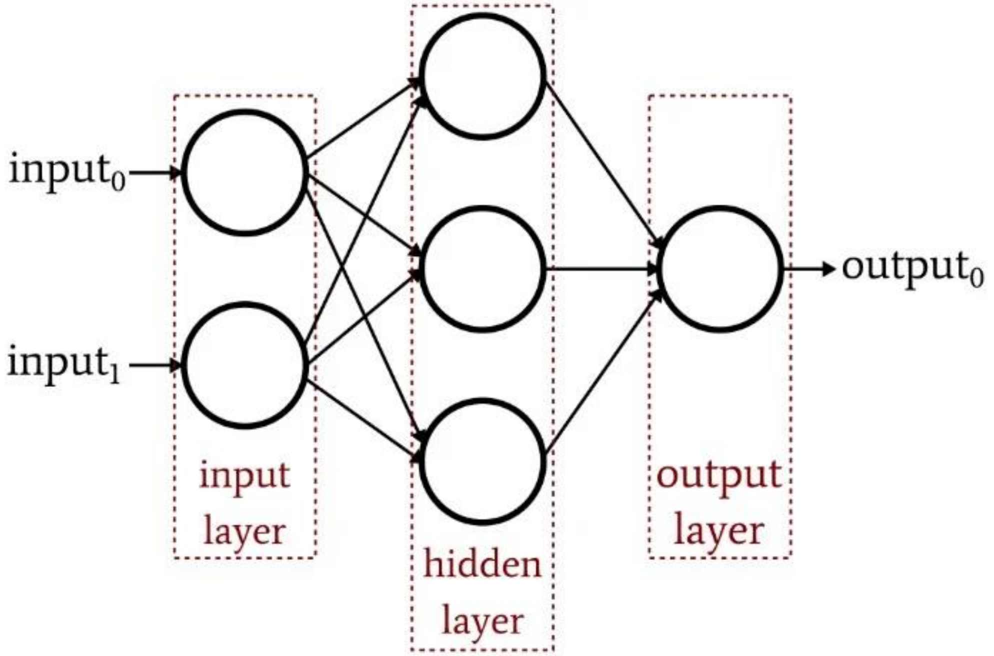
- Designed for binary classification $|Z| = 2$
  - how can we extend for other predictive tasks?
- Linear separation only (the XOR problem)





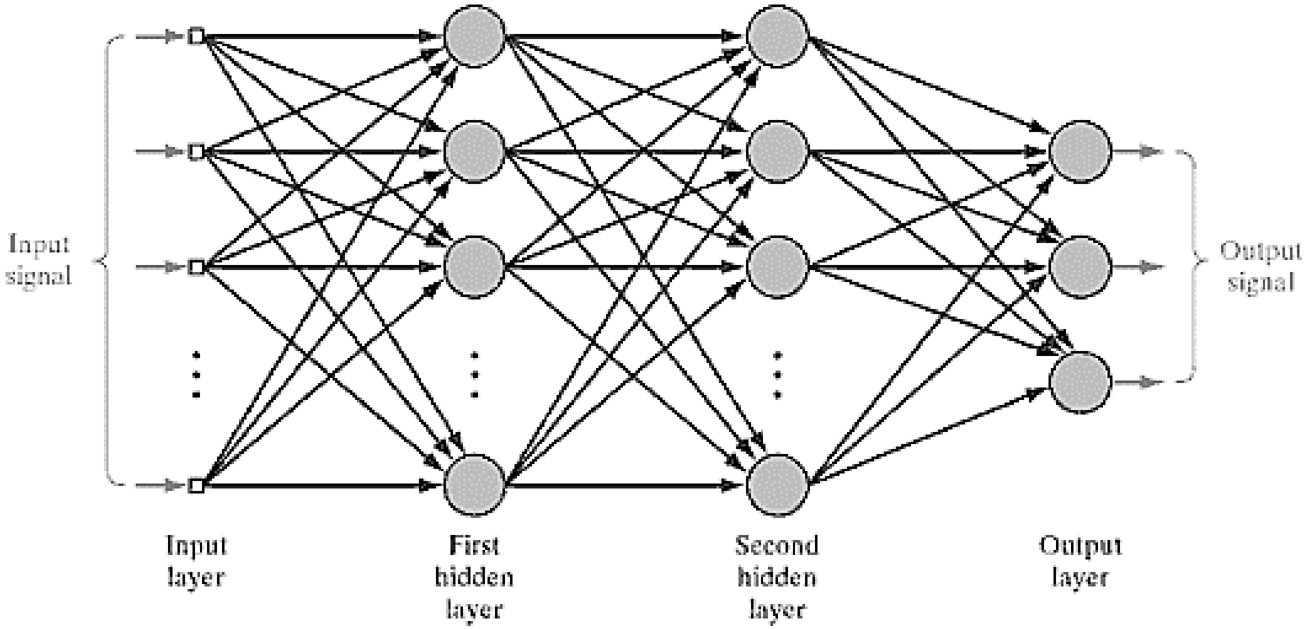
Non-linear problems

- In mathematics, we can compose ***linear*** functions, $f(x)$ ang $g(x)$, to form a ***non-linear*** function, $h(f(x), g(x))$, to model more complex behavior
- 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

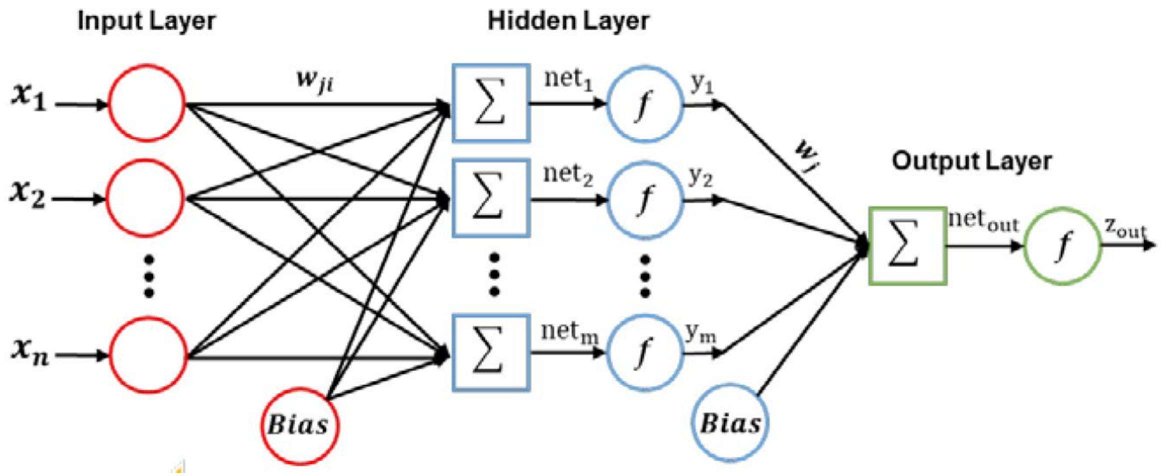
- The composition of perceptrons can be **organized in layers**
  - the outputs of perceptrons in one layer feed the input of perceptrons in the next layer
    - hence **multi-layer perceptrons** (MLPs) are also termed **feed forward networks**
    - a simplified architecture to facilitate the learning
    - 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

- Yet… multiple layers of cascade linear units still produce only linear functions

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

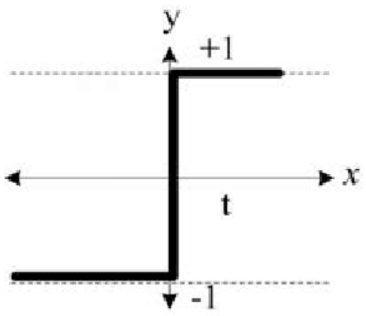


  – solution?

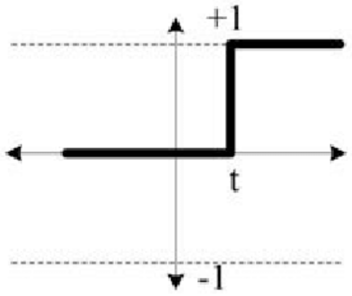
$$z_{out} = f\left(\sum w \, f\left(\sum w \, x_i \right) \right)$$

Activation functions

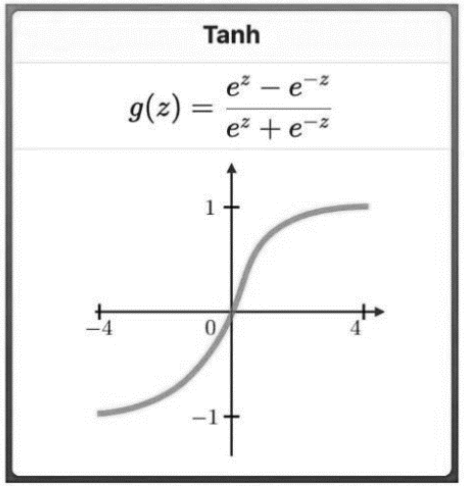
- networks able of representing nonlinear functions:
  - use nonlinear activation functions
  - activations should be continuous and differentiable
  - problems of sign and step?

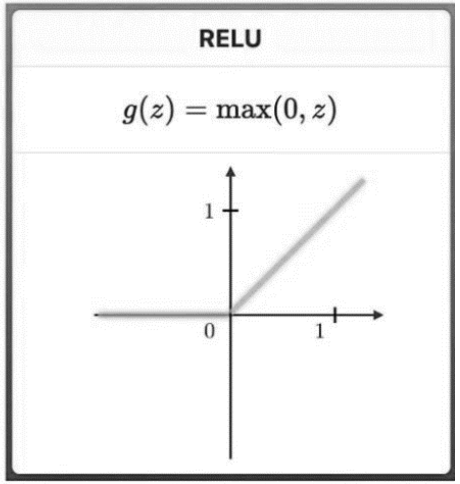


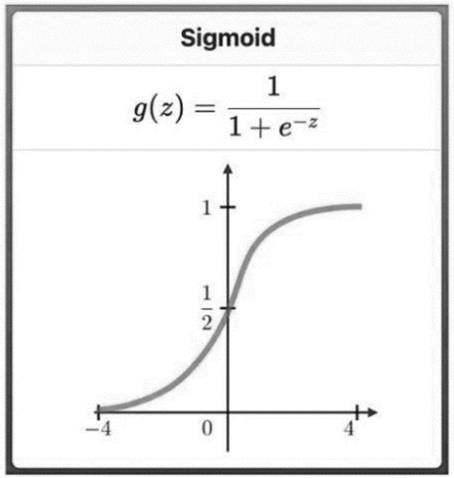
Sign Function

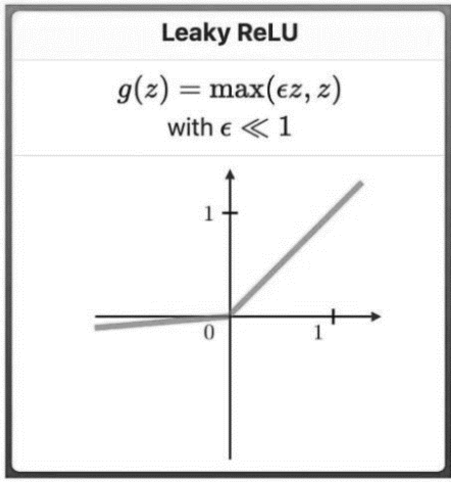


Step Function

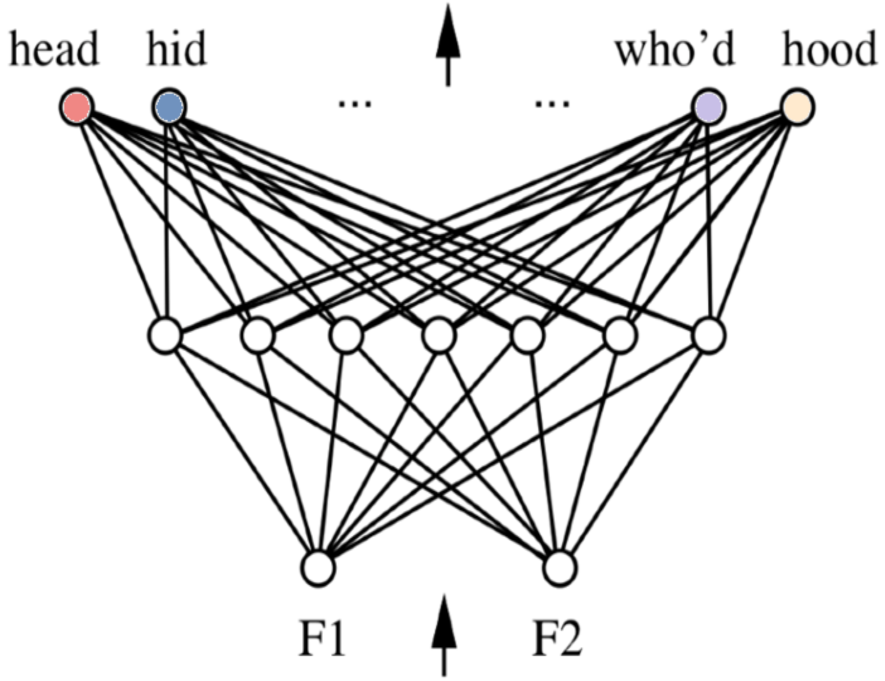


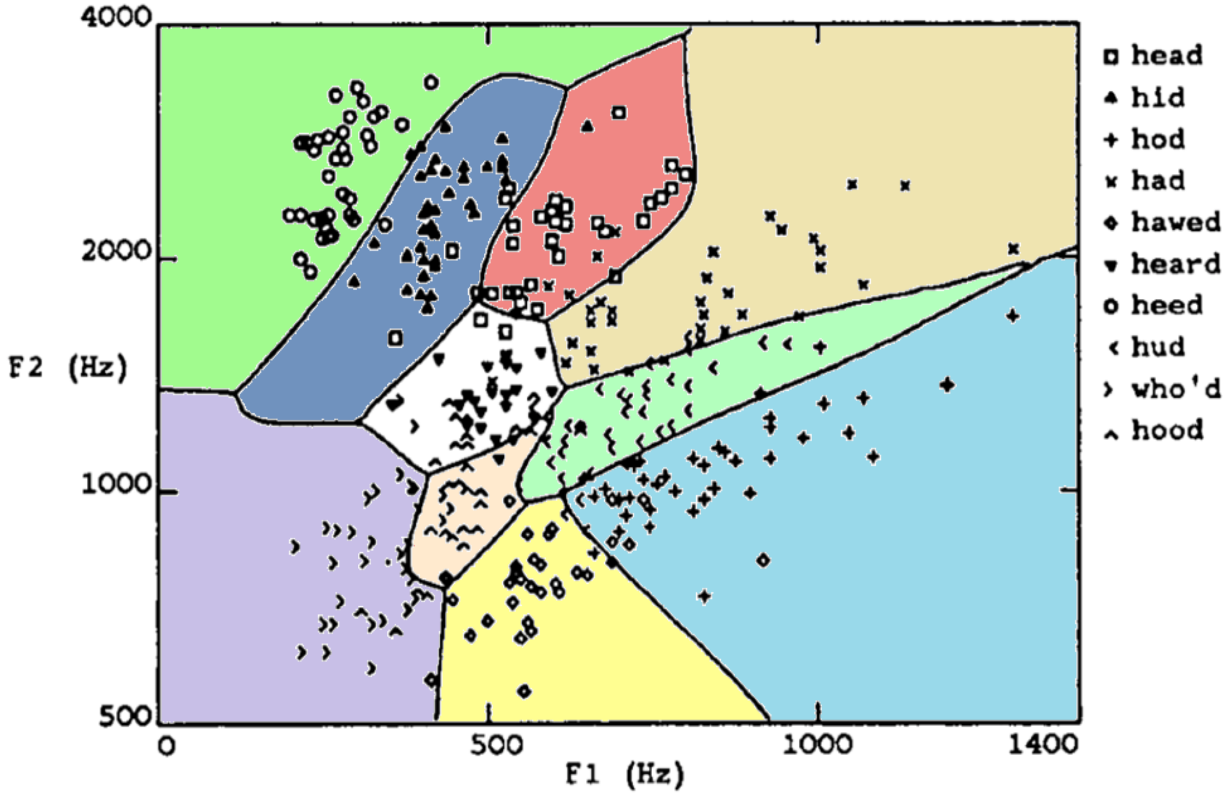




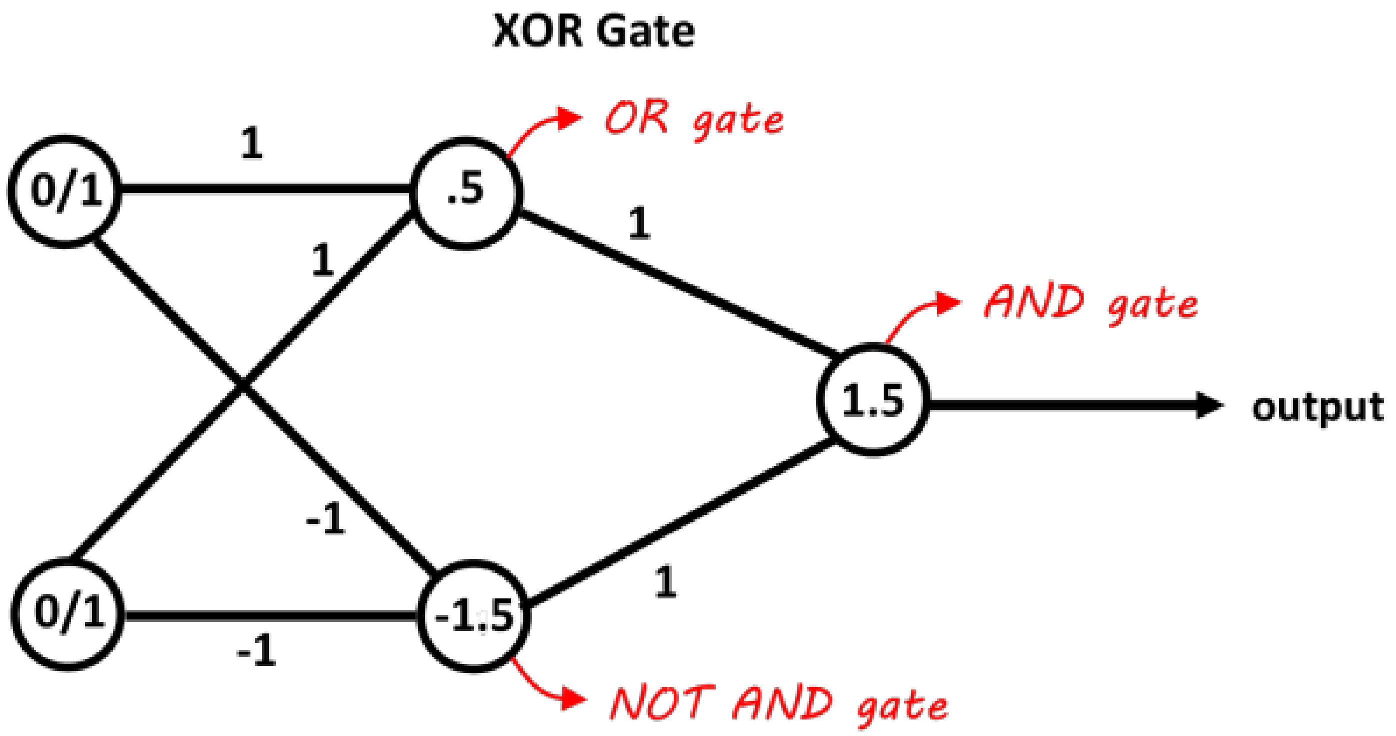


Non-linear boundaries





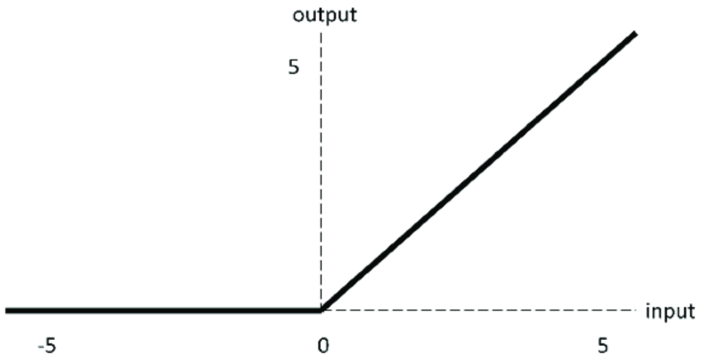
Solving the XOR problem



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

MLPs for regression and classification

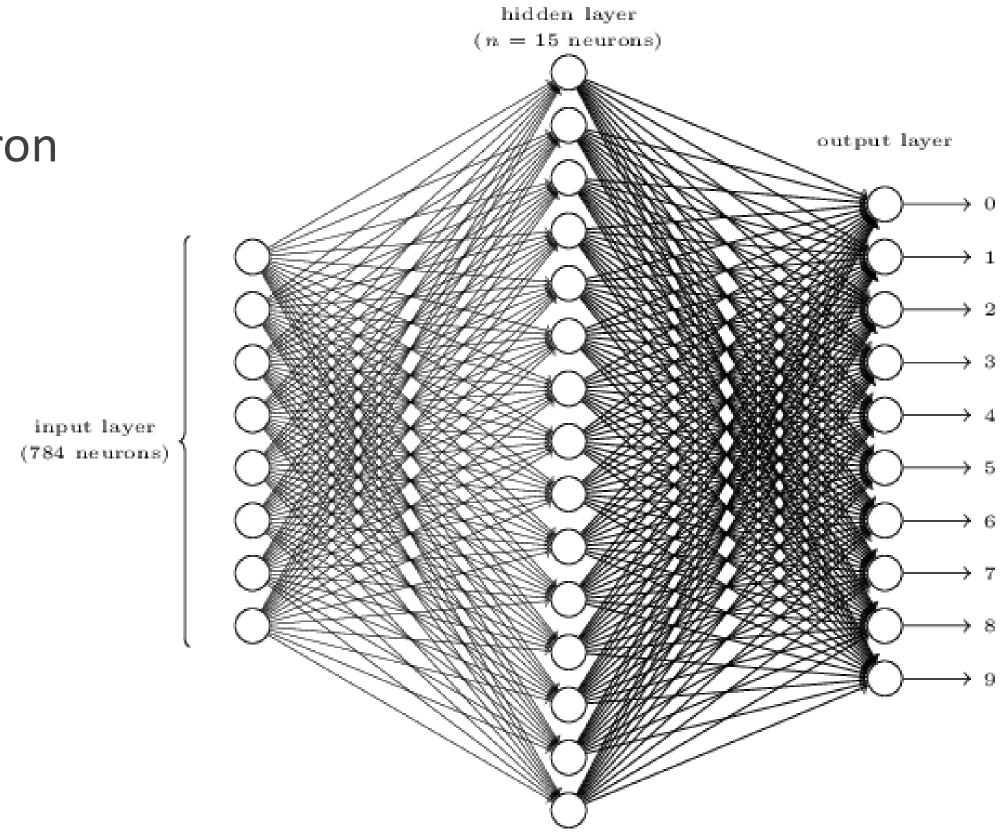
- Until now… a single perceptron can only answer a binary classification task
- Neural networks can be extended to handle more general predictive tasks
  – *multiclass classification*
  – ***regression***
    – replace the last sign/sigmoid/tanh activation by other function – e.g. [rectified] linear 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 $\{\mathbf{x}_i, \mathbf{t}_i\}$

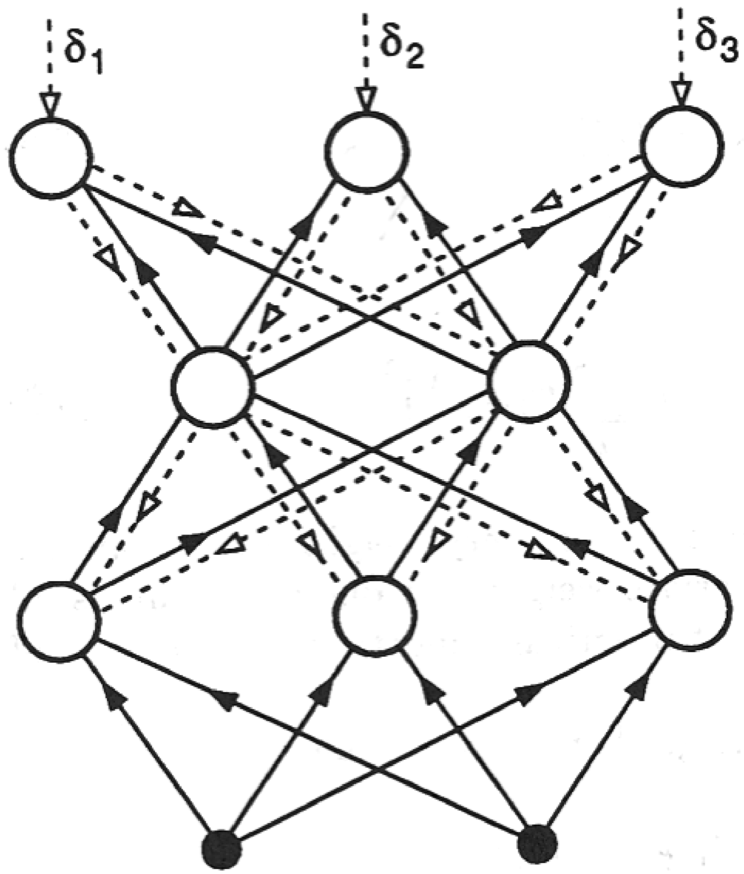
$$E(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{l=1}^{L}\left(t_l^{(i)} - o_l^{(i)}\right)^2$$

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**)

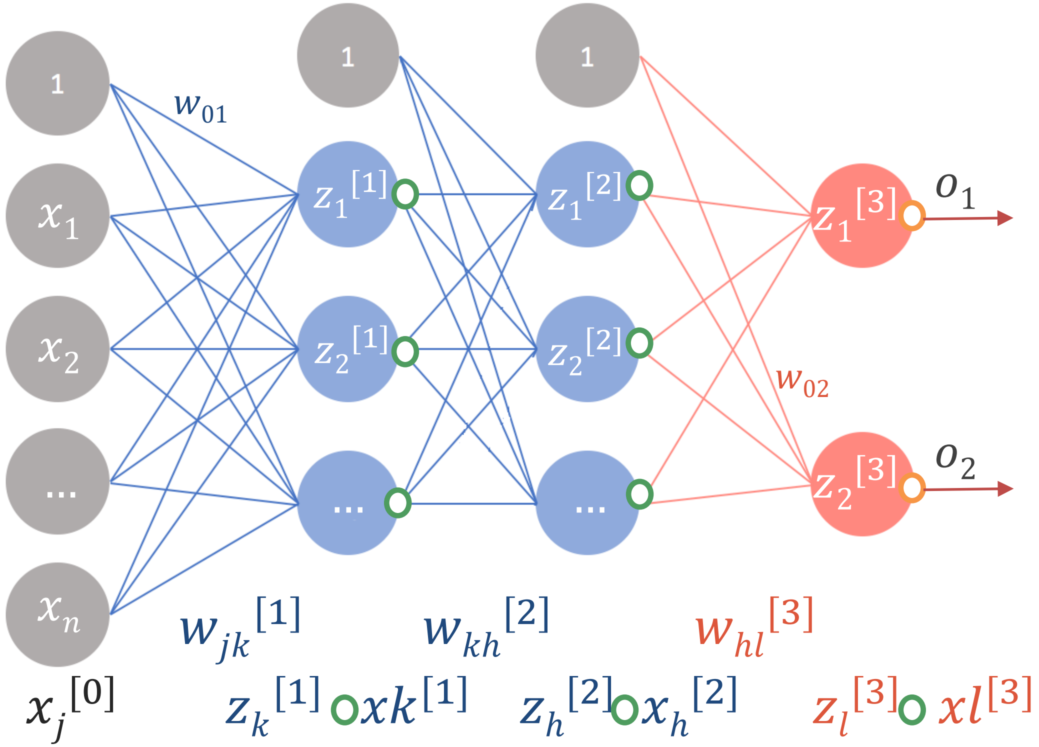


Outline

- 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**, $p = 1,2,..,P$
- $z_k{}^{[p]}$ is the **net value** of the k^{th} node of the p^{th} layer
- $\phi^{[p]}$ is the activation function of the p^{th} layer
- $x_k{}^{[p]}$ is the **signal** of the k^{th} node of the p^{th} layer
  - $x_k{}^{[0]}$ is a synonym for x_k, i.e. the kth **input feature**
  - $x_k{}^{[P]}$ is a synonym for o_k, i.e. the kth **network output**
- $w_{ij}{}^{[p]}$ is the **weight** of the connection from i^{th} node of the p-1th layer to the j^{th} node of the p^{th} layer
- **bias** is the product of $x_0{}^{[p]} = 1$ 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



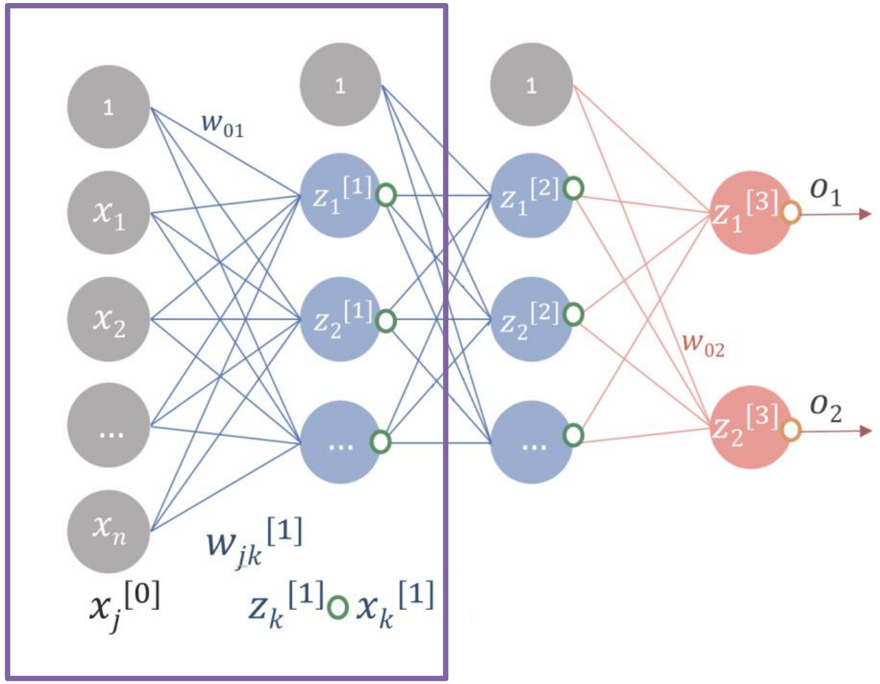
Propagation

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

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

  – … and produces the output

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



Propagation

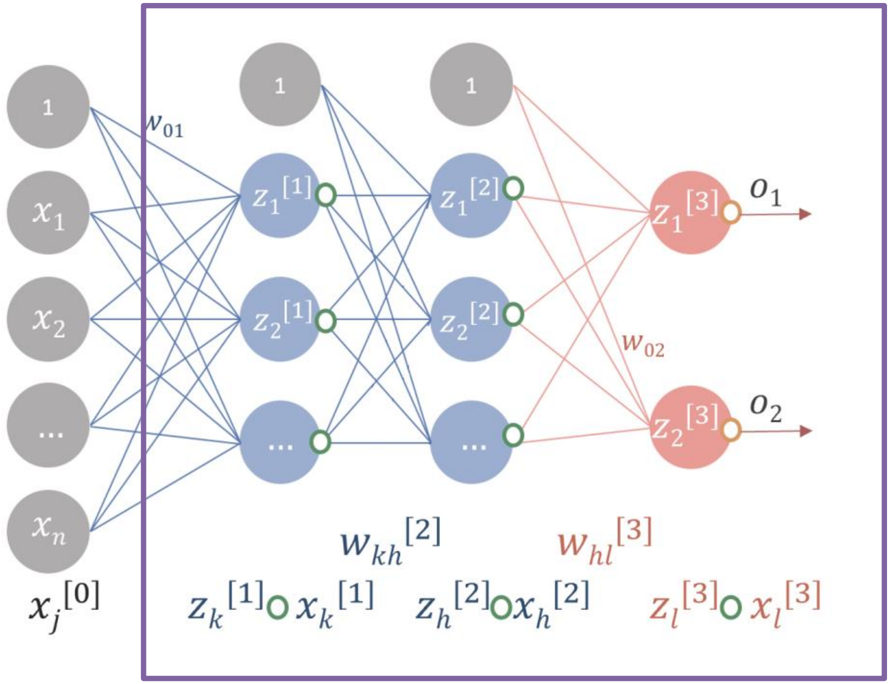
– similarly considering the second layer…

$$x_h{}^{[2]} = \phi^{[2]}\left(z_h{}^{[2]}\right) = \phi^{[2]}\left(\sum_{k=1}^{m^{[1]}} w_{kh}{}^{[2]}\, x_k{}^{[1]}\right)$$

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

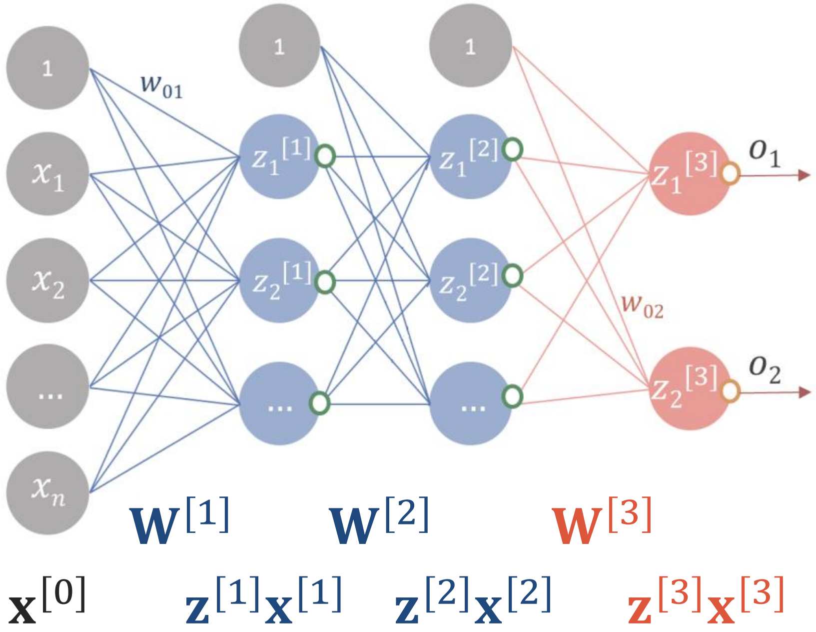
– and the output layer…

$$o_l = xl^{[3]} = \phi^{[3]}\left(z_l{}^{[3]}\right) = \phi^{[3]}\left(\sum_{h=1}^{m^{[2]}} w_{hl}{}^{[3]}\, x_h{}^{[2]}\right) = \phi^{[3]}\left(\sum_{h=1}^{m^{[2]}} w_{hl}{}^{[3]}\,(\dots)\right)$$



Vector notation

- network with P **layers**, $p = 1,2,..,P$
- **parameters:**
  - $\mathbf{W}^{[p]}$ contains the **weights** connecting p-1^{th} layer and p^{th} 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 p^{th} layer
  - $\mathbf{x}^{[p]}$ is the activated/output values of the p^{th} layer, $\mathbf{x}^{[p]} = \phi^{[1]}(\mathbf{z}^{[p]})$
    - $\mathbf{x}^{[0]}$ 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:
  - $\mathbf{W}^{[1]} = \begin{pmatrix} {w_{11}}^{[1]} & \dots & {w_{41}}^{[1]} \\ \dots & & \dots \\ {w_{13}}^{[1]} & \dots & {w_{43}}^{[1]} \end{pmatrix}, \mathbf{W}^{[2]} = \begin{pmatrix} {w_{11}}^{[2]} & \dots & {w_{31}}^{[2]} \\ \dots & & \dots \\ {w_{13}}^{[2]} & \dots & {w_{33}}^{[2]} \end{pmatrix}, \mathbf{W}^{[3]} = \begin{pmatrix} {w_{11}}^{[3]} & {w_{21}}^{[3]} & {w_{31}}^{[3]} \\ {w_{12}}^{[3]} & {w_{22}}^{[3]} & {w_{32}}^{[3]} \end{pmatrix}$
  - $\mathbf{b}^{[1]} = \begin{pmatrix} {b_1}^{[1]} \\ \dots \\ {b_3}^{[1]} \end{pmatrix}, \ \mathbf{b}^{[2]} = \begin{pmatrix} {b_1}^{[2]} \\ \dots \\ {b_3}^{[2]} \end{pmatrix}, \mathbf{b}^{[3]} = \begin{pmatrix} {b_1}^{[3]} \\ {b_2}^{[3]} \end{pmatrix}$
- The **nodes** of the neural network
  - $\mathbf{x} = \mathbf{x}^{[0]} = \begin{pmatrix} x_1 \\ \dots \\ x_4 \end{pmatrix}, \mathbf{x}^{[1]} = \phi^{[1]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) = \begin{pmatrix} {x_1}^{[1]} \\ \dots \\ {x_3}^{[1]} \end{pmatrix}, \ \mathbf{x}^{[2]} = \phi^{[2]}(\mathbf{W}^{[2]}\ \mathbf{x}^{[1]} + \mathbf{b}^{[2]}) = \begin{pmatrix} {x_1}^{[2]} \\ \dots \\ {x_3}^{[2]} \end{pmatrix}$
  - $\boldsymbol{o} = \mathbf{x}^{[3]} = \phi^{[3]}(\mathbf{W}^{[3]}\mathbf{x}^{[2]} + \mathbf{b}^{[3]}) = \begin{pmatrix} o_1 \\ o_2 \end{pmatrix}$

Propagation: example

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

  – $\mathbf{W}^{[1]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{W}^{[2]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{W}^{[3]} = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}, \mathbf{b}^{[1]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b}^{[2]} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \mathbf{b}^{[3]} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$

  – $\mathbf{x}^{[1]} = \phi^{[1]}(\mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 & 0.1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\right) = \sigma\begin{pmatrix} 0.2 \\ 0.2 \\ 0.2 \end{pmatrix},$

  – $\mathbf{x}^{[2]} = \phi^{[2]}(\mathbf{W}^{[2]}\mathbf{x}^{[1]} + \mathbf{b}^{[2]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}\begin{pmatrix} \sigma(0.2) \\ \sigma(0.2) \\ \sigma(0.2) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}\right) = \sigma\begin{pmatrix} 0.165 \\ 0.165 \\ 0.165 \end{pmatrix}$

  – $\boldsymbol{o} = \mathbf{x}^{[3]} = \phi^{[3]}(\mathbf{W}^{[3]}\mathbf{x}^{[2]} + \mathbf{b}^{[3]}) = \sigma\left(\begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix}\begin{pmatrix} \sigma(0.165) \\ \sigma(0.165) \\ \sigma(0.165) \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \end{pmatrix}\right) = \sigma\begin{pmatrix} 0.162 \\ 0.162 \end{pmatrix} = \begin{pmatrix} 0.54 \\ 0.54 \end{pmatrix}$

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 $p \in \{1,2,\dots,P\}$
- **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:

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

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:

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

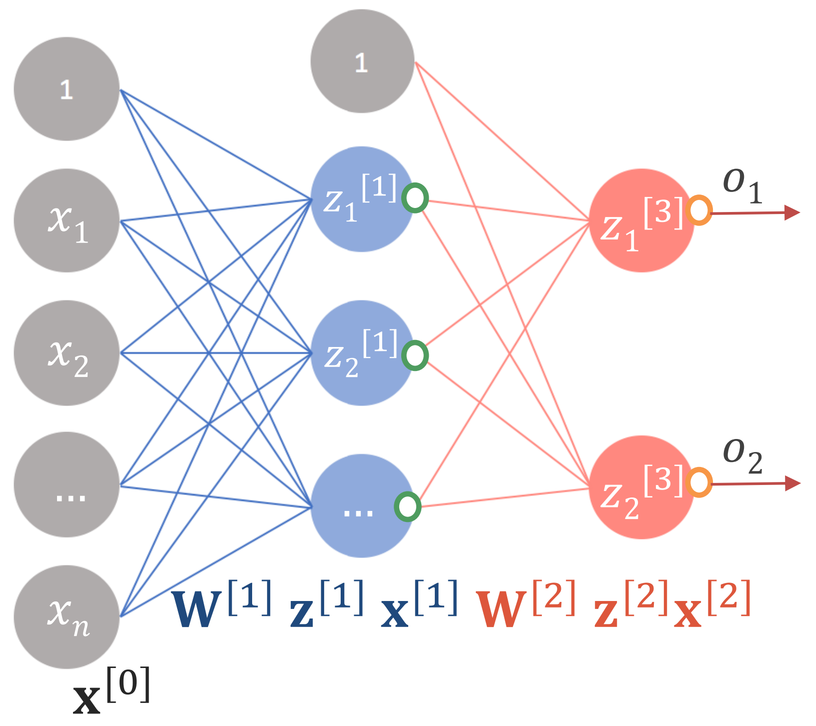
$$\nabla_{\mathbf{x}} f\big(g(\mathbf{x})\big) = \frac{\partial f(g(\mathbf{x}))}{\partial \mathbf{x}} = \frac{\partial f(g(\mathbf{x}))}{\partial g} \cdot \frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}$$

  - example:

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

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 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 $\frac{\partial E}{\partial \mathbf{W}^{[p]}} = \boldsymbol{\delta}^{[p]}$



Backpropagation

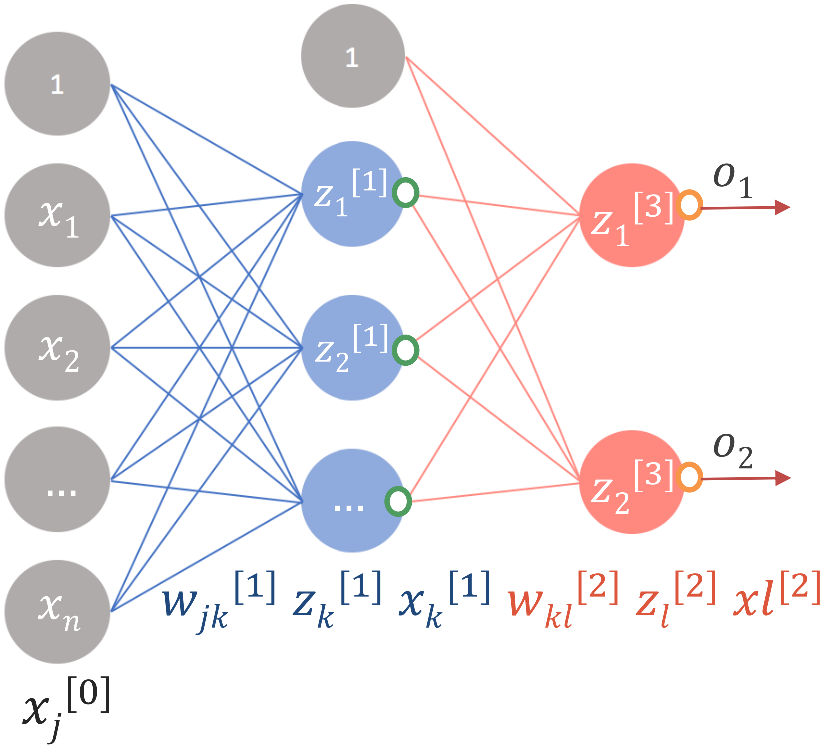
- To compute $\boldsymbol{\delta}^{[p]}$, we need first to infer the update rule…
- Let us infer the update rules on the right
  - let us start with the **last layer**…

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

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

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

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



Backpropagation

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

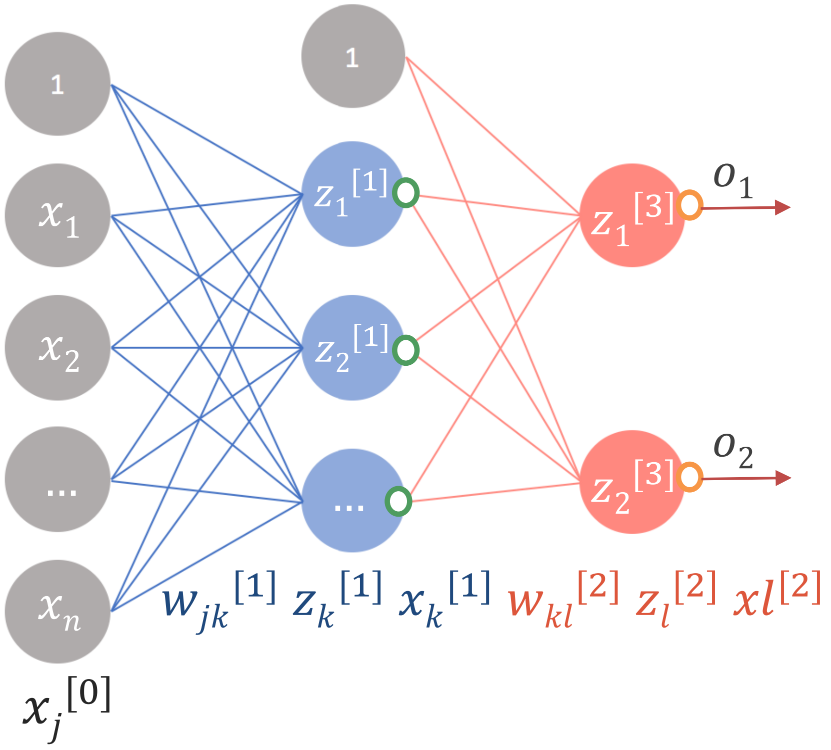
$$\Delta w_{jk}{}^{[1]} = -\eta \frac{\partial E}{\partial w_{jk}{}^{[1]}} = -\eta \sum_{i=1}^{n} \frac{\partial E}{\partial x_k{}^{(i)[1]}} \cdot \frac{\partial x_k{}^{(i)[1]}}{\partial w_{jk}{}^{[1]}}$$

$$\frac{\partial x_k{}^{(i)[1]}}{\partial w_{jk}{}^{[1]}} = \frac{\partial}{\partial w_{jk}{}^{[1]}}\left(\phi^{[1]}\left(\sum_{j=1}^{4} w_{jk}{}^{[1]} x_j{}^{(i)}\right)\right) = \phi'^{[1]}\left(z_k{}^{(i)[1]}\right) \cdot x_j{}^{(i)}$$

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

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

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



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

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

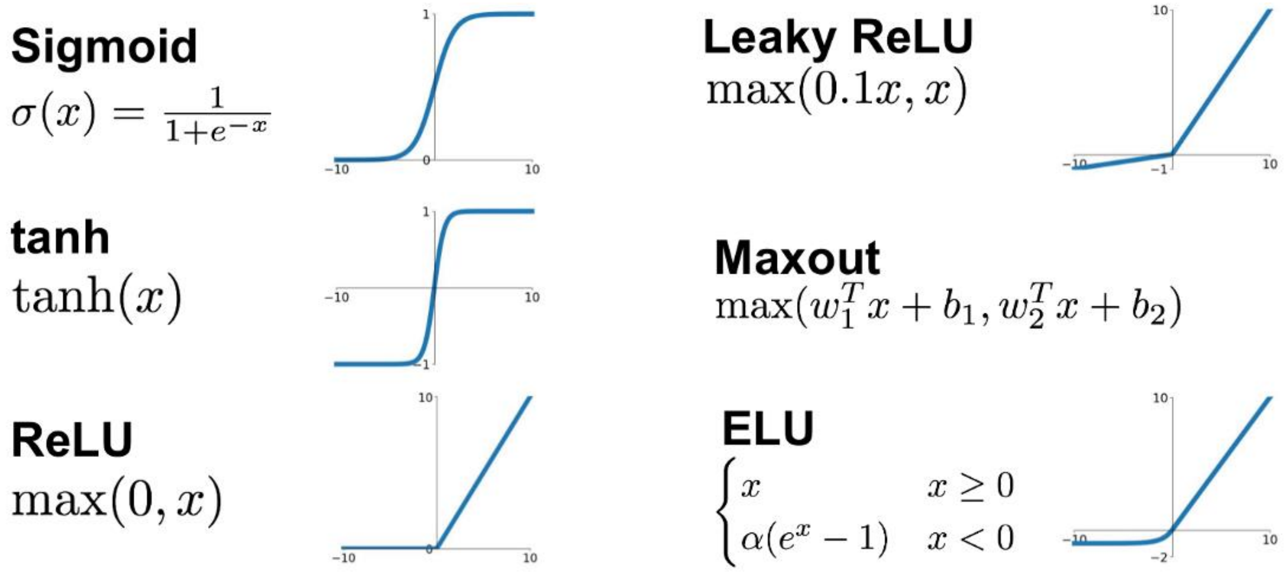
  - and the remaining δ given by

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

- 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



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

Vector notation

- Let us recover important matricial operations
  - **Hadamard product**: element-wise product of vectors or matrices

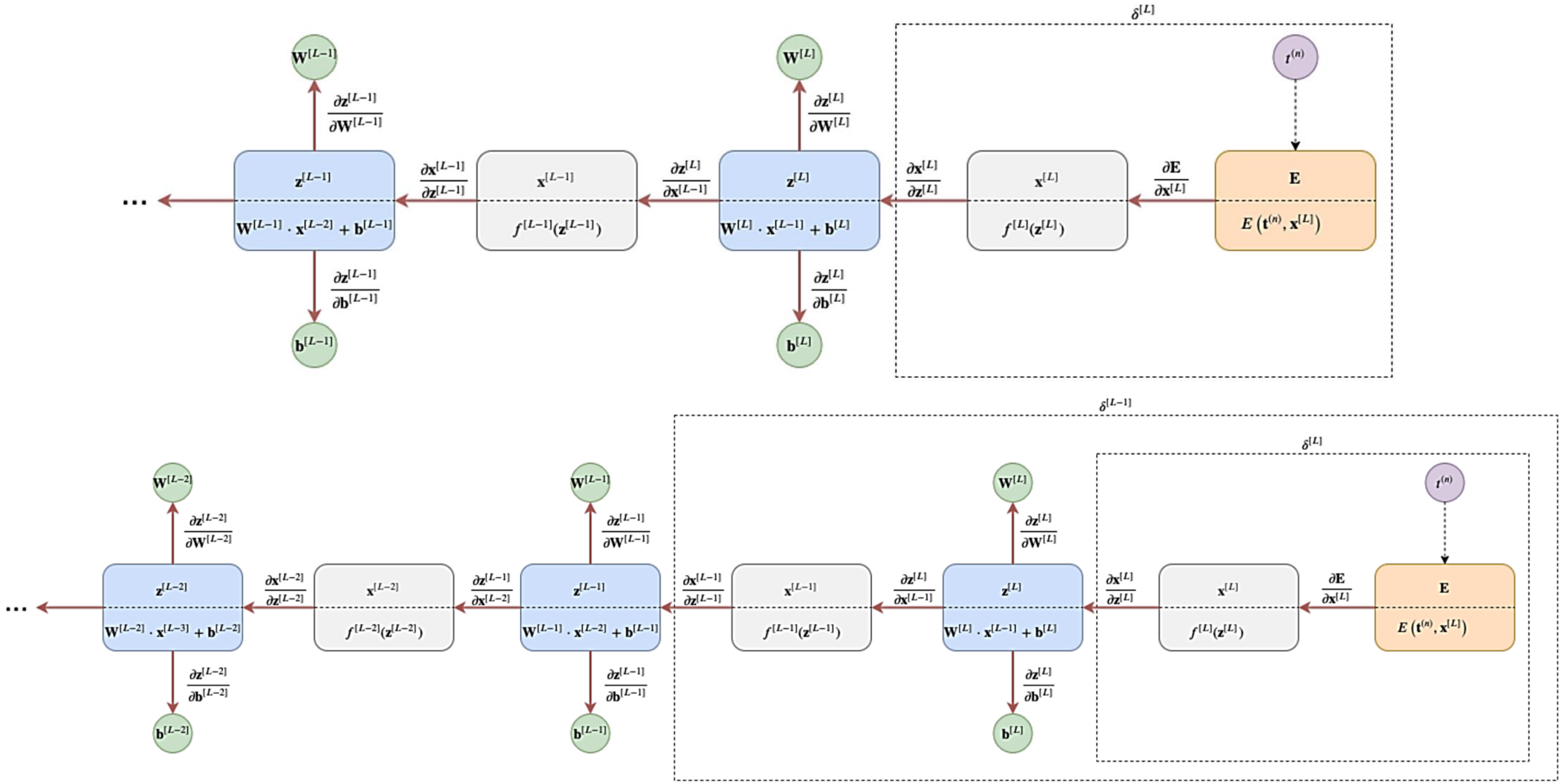
$$\begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \circ \begin{pmatrix} b_1 & b_2 \\ b_3 & b_4 \end{pmatrix} = \begin{pmatrix} a_1 b_1 & a_2 b_2 \\ a_3 b_3 & a_2 b_4 \end{pmatrix}$$

  - **tensor product**: outer product

$$\begin{pmatrix} \omega_0 \\ \omega_1 \end{pmatrix} \otimes \begin{pmatrix} \omega_0 \\ \omega_1 \end{pmatrix} = \begin{pmatrix} \omega_0 \cdot \omega_0 \\ \omega_0 \cdot \omega_1 \\ \omega_1 \cdot \omega_0 \\ \omega_1 \cdot \omega_1 \end{pmatrix}$$

$$A \otimes B = \begin{pmatrix} a_{11} \cdot B & a_{12} \cdot B \\ a_{21} \cdot B & a_{22} \cdot B \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_{11} & a_{11} \cdot b_{12} & a_{12} \cdot b_{11} & a_{12} \cdot b_{12} \\ a_{11} \cdot b_{21} & a_{11} \cdot b_{22} & a_{12} \cdot b_{21} & a_{12} \cdot b_{22} \\ a_{21} \cdot b_{11} & a_{21} \cdot b_{12} & a_{22} \cdot b_{11} & a_{22} \cdot b_{12} \\ a_{21} \cdot b_{21} & a_{21} \cdot b_{22} & a_{22} \cdot b_{21} & a_{22} \cdot b_{22} \end{pmatrix}$$

Backpropagation in vector notation



Backpropagation

- Let us now rewrite the updating rules with gradient descent using vector notation
  - let us minimize the error: $E(\mathbf{w}) = \frac{1}{2}(\boldsymbol{t} - \boldsymbol{o})^2$
  - applying the chain rule taking into attention matricial operators (see previous slide)…

$$\frac{\partial E}{\partial \mathbf{W}^{[p]}} = \frac{\partial E}{\partial \mathbf{x}^{[p]}} \circ \frac{\partial \mathbf{x}^{[p]}}{\partial \mathbf{z}^{[p]}} \cdot \frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{W}^{[p]}}$$

$$\frac{\partial \mathbf{x}^{[p]}}{\partial \mathbf{z}^{[p]}} = \phi'^{[p]}(\mathbf{z}^{[p]})$$

$$\frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{W}^{[p]}} = \mathbf{x}^{[p-1]}$$

$$(\text{for last layer } p = P)\ \frac{\partial E}{\partial \mathbf{x}^{[P]}} = \frac{1}{2}\frac{\partial}{\partial \mathbf{x}^{[P]}}\left(\boldsymbol{t} - \mathbf{x}^{[P]}\right)^2 = \mathbf{x}^{[P]} - \boldsymbol{t}$$

Backpropagation

- Finally, the update of **weights**:

$$\mathbf{W}^{[p]} = \mathbf{W}^{[p]} - \eta \frac{\partial E}{\partial \mathbf{W}^{[p]}}, \text{where } \frac{\partial E}{\partial \mathbf{W}^{[p]}} = \boldsymbol{\delta}^{[p]} \frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{W}^{[p]}} = \boldsymbol{\delta}^{[p]} \left(\mathbf{x}^{[p-1]}\right)^T$$

  – where the delta from the ***last layer*** is

$$\boxed{\boldsymbol{\delta}^{[P]} = \frac{\partial E}{\partial \mathbf{x}^{[P]}} \circ \frac{\partial \mathbf{x}^{[P]}}{\partial \mathbf{z}^{[P]}} = (\mathbf{x}^{[P]} - \boldsymbol{t}) \circ \phi'^{[P]}(\mathbf{z}^{[P]})}$$

  – and the deltas from the ***remaining layers*** (using recursion):

$$\boxed{\boldsymbol{\delta}^{[p]} = \left(\frac{\partial \mathbf{z}^{[p+1]}}{\partial \mathbf{x}^{[p]}}\right)^T \cdot \boldsymbol{\delta}^{[p+1]} \circ \frac{\partial \mathbf{x}^{[p]}}{\partial \mathbf{z}^{[p]}} = \mathbf{W}^{[p+1]^T} \cdot \boldsymbol{\delta}^{[p+1]} \circ \phi'^{[p]}(\mathbf{z}^{[p]})}$$

Backpropagation

- The update of **biases** is analogous:

$$\mathbf{b}^{[p]} = \mathbf{b}^{[p]} - \eta \frac{\partial E}{\partial \mathbf{b}^{[p]}}, \text{where } \frac{\partial E}{\partial \mathbf{b}^{[p]}} = \delta^{[p]} \frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{b}^{[p]}} = \delta^{[p]}$$

– since $\frac{\partial E}{\partial \mathbf{b}^{[p]}} = \frac{\partial E}{\partial \mathbf{x}^{[p]}} \circ \frac{\partial \mathbf{x}^{[p]}}{\partial \mathbf{z}^{[p]}} \cdot \frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{b}^{[p]}}$ and $\frac{\partial \mathbf{z}^{[p]}}{\partial \mathbf{b}^{[p]}} = \mathbf{1}$

- Recall the full learning process

do following steps **until** convergence (e.g. loss not improving on the last d steps):

– select a set of observations $\{\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_q}\}$ from X

– updated the parameters $\mathbf{W}^{[p]}$, $\mathbf{b}^{[p]}$ of the network using the backpropagation rules

Backpropagation: example

- Let us recover our example (network on the right)
  - hidden layers with 3 nodes each, all weights at 0.1, no biases
  - sigmoid activation function $\phi^{[p]}(x) = \sigma(x)$
  - learning rate $\eta = 1$ and squared error loss
  - **quest**: update parameters using $\mathbf{x}^T = (1\ 1\ 0\ 0)$ with $\mathbf{t} = (0\ 1)$
- *Solution* **notes**
  - for convenience, let us recall the sigmoid derivative

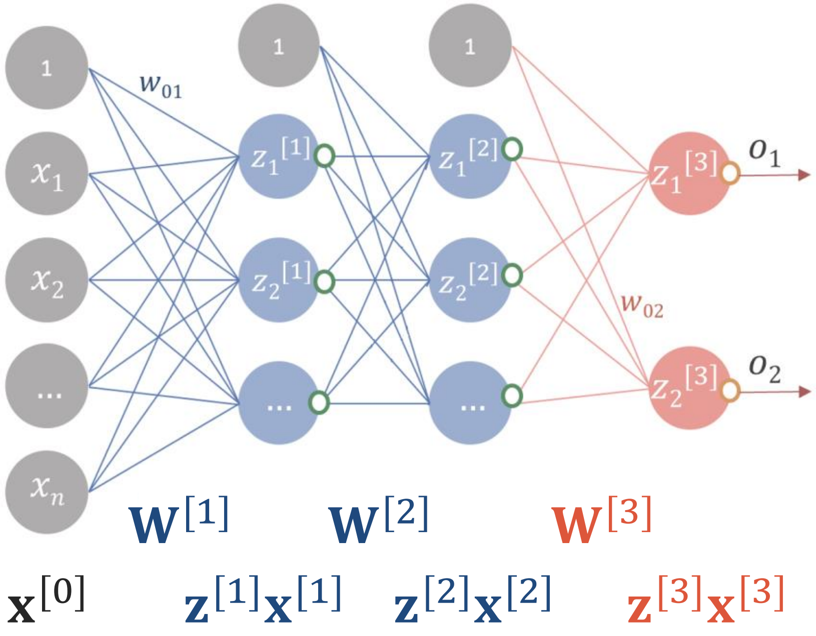
$$\frac{\partial\sigma(x)}{\partial x} = \frac{\partial}{\partial x}\frac{1}{1+exp(-x)} = \sigma(x)(1-\sigma(x))$$

Note: $\sigma(x) = f(g(h(q(x))))$

where $q(x)$=$-x$, $h(x)$=$\exp(x)$, $g(x)$=1+$\exp(x)$, $f(x)$=$\frac{1}{x}$

  - first, let us recover the propagated values

$$\mathbf{x}^{[1]} = \sigma\begin{pmatrix}0.2\\0.2\\0.2\end{pmatrix}, \mathbf{x}^{[2]} = \sigma\begin{pmatrix}0.165\\0.165\\0.165\end{pmatrix}, \boldsymbol{o} = \mathbf{x}^{[3]} = \sigma\begin{pmatrix}0.162\\0.162\end{pmatrix}$$



Backpropagation: example

- Third, let us compute the deltas:
  - $\boldsymbol{\delta}^{[3]} = \left(\mathbf{x}^{[3]} - \boldsymbol{t}\right) \circ \sigma\left(\mathbf{z}^{[3]}\right) \circ \left(1 - \sigma\left(\mathbf{z}^{[3]}\right)\right) = \left(\begin{pmatrix}0.54\\0.54\end{pmatrix} - \begin{pmatrix}1\\0\end{pmatrix}\right) \circ \begin{pmatrix}0.54\\0.54\end{pmatrix} \circ \left(1 - \begin{pmatrix}0.54\\0.54\end{pmatrix}\right) = \begin{pmatrix}-0.114\\0.134\end{pmatrix}$
  - $\boldsymbol{\delta}^{[2]} = \mathbf{W}^{[3]^T} \cdot \boldsymbol{\delta}^{[3]} \circ \sigma\left(\mathbf{z}^{[2]}\right) \circ \left(1 - \sigma\left(\mathbf{z}^{[2]}\right)\right) = \begin{pmatrix}0.1 & 0.1\\0.1 & 0.1\\0.1 & 0.1\end{pmatrix} \cdot \begin{pmatrix}-.114\\0.134\end{pmatrix} \circ \sigma\begin{pmatrix}0.165\\0.165\\0.165\end{pmatrix}\left(1 - \sigma\begin{pmatrix}0.165\\0.165\\0.165\end{pmatrix}\right) = \begin{pmatrix}0.0005\\0.0005\\0.0005\end{pmatrix}$
  - $\boldsymbol{\delta}^{[1]} = \mathbf{W}^{[2]^T} \cdot \boldsymbol{\delta}^{[2]} \circ \sigma\left(\mathbf{z}^{[1]}\right) \circ \left(1 - \sigma\left(\mathbf{z}^{[1]}\right)\right) = \begin{pmatrix}0.1 & 0.1 & 0.1\\0.1 & 0.1 & 0.1\\0.1 & 0.1 & 0.1\end{pmatrix} \cdot \begin{pmatrix}0.0005\\0.0005\\0.0005\end{pmatrix} \circ \sigma\begin{pmatrix}0.2\\0.2\\0.2\end{pmatrix}\left(1 - \sigma\begin{pmatrix}0.2\\0.2\\0.2\end{pmatrix}\right) = \begin{pmatrix}3.7E{-}5\\3.7E{-}5\\3.7E{-}5\end{pmatrix}$

- Fourth, and finally, let us perform the updates of the weights…
  - $\mathbf{W}^{[1]} = \mathbf{W}^{[1]} - \eta\frac{\partial E}{\partial \mathbf{W}^{[1]}} = \mathbf{W}^{[1]} - \eta\boldsymbol{\delta}^{[1]}\mathbf{x}^T = \begin{pmatrix}0.1 & 0.1 & 0.1 & 0.1\\0.1 & 0.1 & 0.1 & 0.1\\0.1 & 0.1 & 0.1 & 0.1\end{pmatrix} - 1\begin{pmatrix}3.7E{-}5\\3.7E{-}5\\3.7E{-}5\end{pmatrix}\begin{pmatrix}1 & 1 & 0 & 0\end{pmatrix} = \begin{pmatrix}.09996 & .09996 & 0.1 & 0.1\\.09996 & .09996 & 0.1 & 0.1\\.09996 & .09996 & 0.1 & 0.1\end{pmatrix}$
  - $\mathbf{b}^{[1]} = \mathbf{b}^{[1]} - \eta\frac{\partial E}{\partial \mathbf{b}^{[1]}} = \mathbf{b}^{[1]} - \eta\boldsymbol{\delta}^{[1]} = \begin{pmatrix}-3.7E{-}5\\-3.7E{-}5\\-3.7E{-}5\end{pmatrix}$

Backpropagation: example

- $\mathbf{W}^{[2]} = \mathbf{W}^{[2]} - \eta \frac{\partial E}{\partial \mathbf{W}^{[2]}} = \mathbf{W}^{[2]} - \eta \boldsymbol{\delta}^{[2]} (\mathbf{x}^{[1]})^T = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} - 1 \begin{pmatrix} 0.0005 \\ 0.0005 \\ 0.0005 \end{pmatrix} (0.55 \quad 0.55 \quad 0.55) = \begin{pmatrix} 0.0997 & 0.0997 & 0.0997 \\ 0.0997 & 0.0997 & 0.0997 \\ 0.0997 & 0.0997 & 0.0997 \end{pmatrix}$
- $\mathbf{b}^{[2]} = \mathbf{b}^{[2]} - \eta \frac{\partial E}{\partial \mathbf{b}^{[2]}} = \mathbf{b}^{[2]} - \eta \boldsymbol{\delta}^{[2]} = \begin{pmatrix} -0.0005 \\ -0.0005 \\ -0.0005 \end{pmatrix}$
- $\mathbf{W}^{[3]} = \mathbf{W}^{[3]} - \eta \frac{\partial E}{\partial \mathbf{W}^{[3]}} = \mathbf{W}^{[3]} - \eta \boldsymbol{\delta}^{[3]} (\mathbf{x}^{[2]})^T = \begin{pmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.1 & 0.1 \end{pmatrix} - 1 \begin{pmatrix} -0.114 \\ 0.134 \end{pmatrix} (0.54 \quad 0.54 \quad 0.54) = \begin{pmatrix} 0.162 & 0.162 & 0.162 \\ 0.027 & 0.027 & 0.027 \end{pmatrix}$
- $\mathbf{b}^{[3]} = \mathbf{b}^{[3]} - \eta \frac{\partial E}{\partial \mathbf{b}^{[3]}} = \mathbf{b}^{[3]} - \eta \boldsymbol{\delta}^{[2]} = \begin{pmatrix} 0.114 \\ -0.134 \end{pmatrix}$
- $E(\mathbf{w}_{init}) = -\frac{1}{2}\left((1 - 0.54)^2 + (0 - 0.54)^2\right) = 0.25, \quad E(\mathbf{w}_{new}) = -\frac{1}{2}\left((1 - 0.593)^2 + (0 - 0.544)^2\right) = 0.23$

- Notes:
  - considering the classification of the given observation, we observe that it decreased from 0.25 to 0.23
  - random weight initializations will prevent repeated values on $\boldsymbol{\delta}^{[p]}$, promoting a more expressive learning
  - first layers update more slowly than last layers (we will return to this point later!)

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

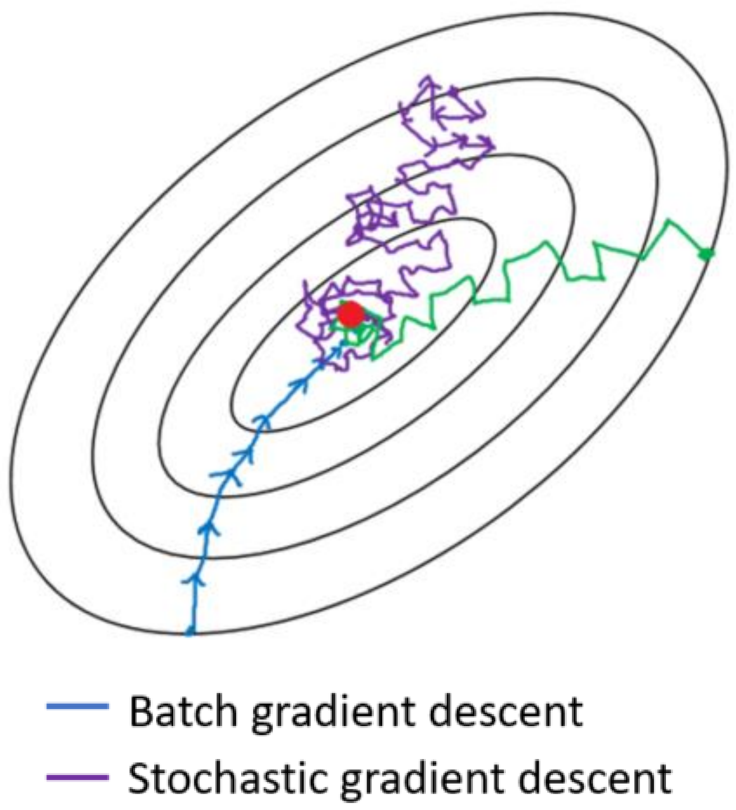
Gradient descent variants

- **Batch**, classic or steepest gradient descent
  - use all n observations to compute $\frac{\partial E}{\partial \mathbf{W}^{[p]}}$ contributions
  - sum all the contributions and only after update weights

$$\mathbf{W}^{[p]} = \mathbf{W}^{[p]} - \eta \left(\frac{\partial E}{\partial \mathbf{W}^{[p]}}^{(1)} + \cdots \frac{\partial E}{\partial \mathbf{W}^{[p]}}^{(n)} \right)$$

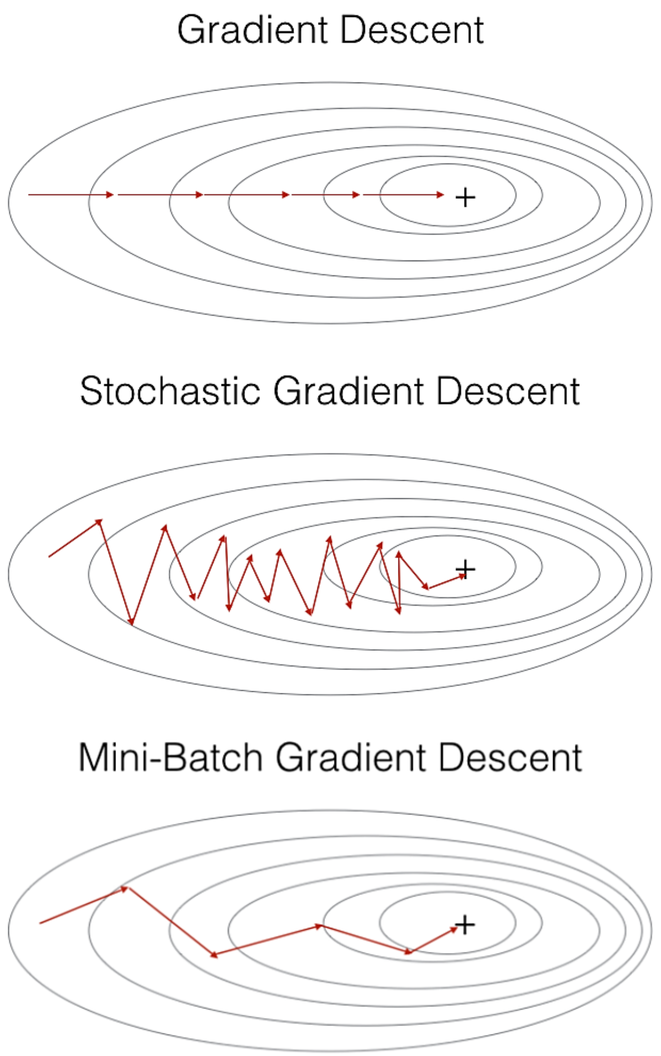
  - some disadvantages:
    - the low stochasticity can lead to be stuck in local minima
    - efficiency: updates can be slow to compute for high n

- **Stochastic** gradient descent
  - use one observation at a time $\mathbf{W}^{[p]} = \mathbf{W}^{[p]} - \eta \frac{\partial E}{\partial \mathbf{W}^{[p]}}^{(i)}$
  - disadvantage: weak stability can be associated with a heightened number of iterations until convergence



Gradient descent variants

- Can we trade-off the limitations of batch and stochastic GD?
  - Yes: mini-batch gradient descent
- A **mini-batch** is a compact subset of observations
- Mini-batches are randomly selected to provide:
  - more stable updates than stochastic alternatives
  - more efficient updates than the classic GD
    while more able to escape local minimum
- How to choose the #observations in a mini-batch?
- The number of **epochs** is generally used in reference
  to the number of (mini-batch) network updates



Mini-batch backpropagation

- Mini-batch backpropagation with batches of size q:
  1. Initialize the weights to small random values
  2. Randomly choose q observations from X, $\{\mathbf{x}_{i_1}, \dots, \mathbf{x}_{i_q}\}$
  3. Propagate the signals through the network, $\mathbf{x}^{(i)[p]} = \phi^{[p]}(\mathbf{W}^{[p]}\mathbf{x}^{(i)[p-1]} + \mathbf{b}^{[p]})$
  4. Compute the deltas for the output layer $\boldsymbol{\delta}^{[P]} = (\mathbf{x}^{[(i)P]} - \boldsymbol{t}^{(i)}) \circ \phi'^{[P]}(\mathbf{z}^{(i)[P]})$
  5. Compute the deltas for the preceding layers $\boldsymbol{\delta}^{[p]} = \mathbf{W}^{[p+1]^T} \cdot \boldsymbol{\delta}^{(i)[p+1]} \circ \phi'^{[p]}(\mathbf{z}^{(i)[p]})$
  6. Update all connections, $\mathbf{W}^{[p]} = \mathbf{W}^{[p]} - \eta \left(\frac{\partial E}{\partial \mathbf{w}^{[p]}}^{(1)} + \cdots \frac{\partial E}{\partial \mathbf{w}^{[p]}}^{(q)} \right)$, where $\frac{\partial E}{\partial \mathbf{w}^{[p]}}^{(i)} = \boldsymbol{\delta}^{(i)[p]} \left(\mathbf{x}^{(i)[p-1]} \right)^T$
  7. Compute the loss $E(\mathbf{w})$
  8. **If** there is evidence of convergence (loss not improving for past iterations) **then** terminate
     Else go to (2) and repeat for the next batch

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

Cross-entropy

- Until here, our focus has been placed on minimizing squared error function

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

  - yet **alternative loss functions** yield relevant properties
  - cross-entropy is a measure of concordance, classically $H(P,Q) = -\sum_{i=1}^{|C|} p_i log(q_i)$

    where P is our ground truth distribution (*targets*) and Q is the learnt distribution (*estimates*)
  - it can be used as the loss: **cross-entropy error**…

$$E(\mathbf{w}) = -\sum_{i=1}^{n}\sum_{l=1}^{|C|} t_l^{(i)}\, log\left(o_l^{(i)}\right)$$

  - Exercise: given targets $\left\{\begin{pmatrix}0\\1\\0\end{pmatrix},\begin{pmatrix}1\\0\\0\end{pmatrix}\right\}$ and MLP estimates $\left\{\begin{pmatrix}0.3\\0.5\\0.2\end{pmatrix},\begin{pmatrix}0.8\\0.1\\0.1\end{pmatrix}\right\}$ compute the MLP's cross-entropy

Cross-entropy and activations

- An important property of cross entropy:
  - outputs o_l should be in $[0,1]$. Why? Inspect the loss
  - furthermore, the classic cross-entropy formulation, $H(P,Q)$, generally assumes: $\sum_i p_i = 1$ and $\sum_i q_i = 1$ (in other words $\sum_l t_l = 1$ and $\sum_l o_l = 1$)

- Important implications:
  - the activation functions at the output layer, i.e. , should be also in $[0,1]$
    - e.g. $\mathrm{sigmoid}(x)$ is eligible, $\mathrm{ReLU}(x)$ is not eligible
  - although not mandatory, the output activations should $\sum_l o_l = 1$
    - can we simply normalize the output values $\frac{o_k}{\sum_l o_l}$?
      - not as simple! If we use sigmoid activation and post-normalize the outputs, normalization will affect the learning – need to revise the update rules!

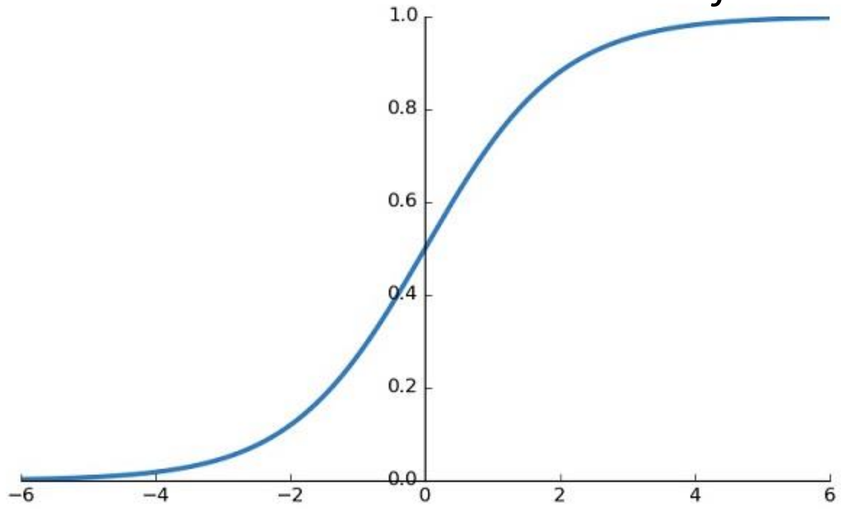
Softmax

- At the light of previous limitations – solution?
  - **softmax** activation

$$\mathrm{softmax}(z_i^{[P]}|\mathbf{z}^{[P]}) = \frac{e^{z_i^{[P]}}}{\sum_{l=1}^{L} e^{z_l^{[P]}}}$$

  - preferred activation on the output layer to use with cross-entropy error
  - properties:
    - ensures $\sum_l o_l = 1$
    - focus on differences instead of absolute values
      - exercise: find the softmax outputs for $\mathbf{z}^{[P]} = (1\ 4\ 0)^T$, $\mathbf{z}^{[P]} = (11\ 14\ 10)^T$ and $\mathbf{z}^{[P]} = (0.25\ 1\ 0)^T$
      - this confers expressivity to the learning

softmax



Minimizing cross-entropy

- Let us find the update rules for cross-entropy using general activations $\phi^{[p]}$
- Knowing $o_l = \phi^{[P]}\left(\sum_{k=1}^{K} {w_{kl}}^{[P]}\, {x_k}^{[P-1]}\right)$ … we get

$$E(\mathbf{w}) = -\sum_{l=1}^{2}\sum_{i=1}^{n} t_l^{(i)}\, log\left(\phi^{[P]}\left(\sum_{k=1}^{K} {w_{kl}}^{[P]}\, {v_k}^{[P-1]}\right)\right)$$

– recalling that $\frac{\partial\ \log_a x}{\partial x} = \frac{1}{x}\ln(a)$ and considering $a = e$ for simplicity

– when minimizing the error…

$$\frac{\partial E}{\partial {w_{kl}}^{[P]}} = \ \dots\ \text{some steps later}\ \dots = -\sum_{i=1}^{n}\left(t_l^{(i)} - o_l^{(i)}\right)\cdot {x_k}^{(i)[P-1]}$$

$$\frac{\partial E}{\partial \mathbf{W}^{[P]}} = \sum_{i=1}^{n} \boldsymbol{\delta}^{[P]}\cdot {\mathbf{x}^{(i)[P-1]}}^{T}\ \ \text{with}\ \ \boldsymbol{\delta}^{[P]} = \left(\boldsymbol{o}^{(i)} - \boldsymbol{t}^{(i)}\right)$$

Minimizing cross-entropy

- For the former layer, by applying the chain rule… and a few mathematics… we get

$$\frac{\partial E}{\partial {w_{jk}}^{[p]}} = \sum_{i=1}^{n} \frac{\partial E}{\partial {x_k}^{(i)[p]}} \cdot \frac{\partial {x_k}^{(i)[p]}}{\partial {w_{jk}}^{[p]}} = \cdots = -\sum_{i=1}^{n} \sum_{l=1}^{2} \left(t_l^{(i)} - o_l^{(i)}\right) \cdot {w_{kl}}^{(i)[p+1]} \cdot \phi'({z_k}^{(i)[p]}) \cdot {x_j}^{(i)[p-1]}$$

– yielding the same updates for any former layer:

$$\frac{\partial E}{\partial \mathbf{W}^{[p]}} = \boldsymbol{\delta}^{[p]} \mathbf{x}^T \quad \text{with } \boldsymbol{\delta}^{[p]} = {\mathbf{W}^{[p+1]}}^T \cdot \boldsymbol{\delta}^{[p+1]} \circ \phi'^{[p]}(\mathbf{z}^{[p]})$$

- What is then the **difference** between cross-entropy and SSE?
  - just $\boldsymbol{\delta}^{[P]}$ (which then affects every other delta $\boldsymbol{\delta}^{[p]}$ for $1,2,\dots,P-1$)
  - while for the cross-entropy error we have $\boldsymbol{\delta}^{[P]} = (\boldsymbol{o} - \boldsymbol{t})$
  - for the old squared error we have $\boldsymbol{\delta}^{[P]} = (\boldsymbol{o} - \boldsymbol{t}) \circ \phi'^{[P]}(\mathbf{z}^{[P]})$
  - this difference in red makes cross-entropy more attractive in some scenarios as it can yield faster convergence

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**

Convergence of backpropagation

- As we saw with perceptron learning, we can control η to promote convergence:
  - when η is small: smooth path (yet slow)
  - when η is large: oscillatory path
  - when η exceeds a certain critical value, backpropagation can become unstable

- Still for small η… gradient descent can find hard to leave **local minimum**
  - Problem: perhaps not a global minimum
  - Solutions?
    - **stochastic** gradient descent
    - train multiple networks with **different initial weights** (i.e., run multiple times)
    - add **momentum** (next class!)

Validation and test data

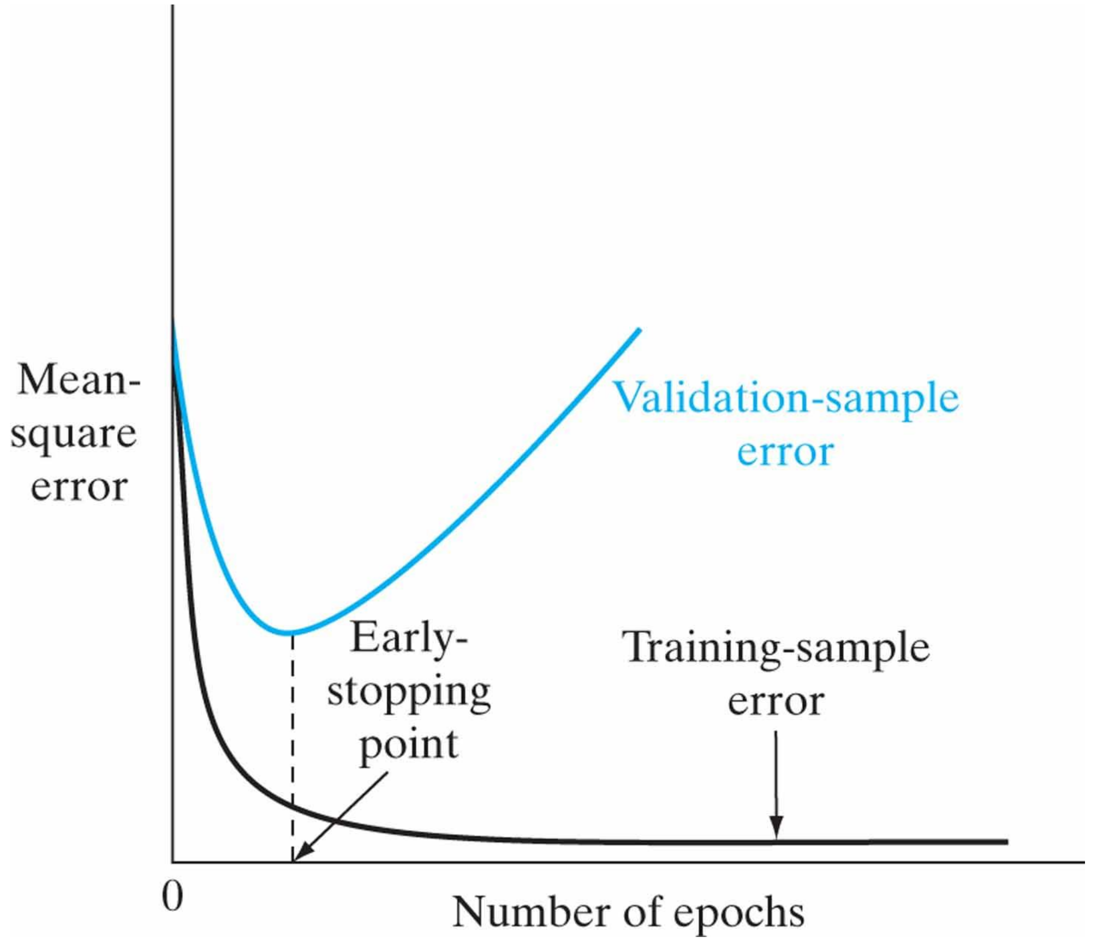
- In neural processing, data is generally divided into:
  - **training data**: to update the network
  - **validation data**: to decide the best network (parameters)…
    - more later!
  - **test data**: to get a final unbiased assessment of the network performance
    - generally we expect testing error to be worse than validation error

  - k-fold Cross-Validation (multiple train-test partitions) can be as well considered for a more comprehensive comparison of neural networks
    - for each iteration, the training partition is then subdivided onto train and validation sets
      - exercise: given 1000 observations, 10-fold CV, and 90-10 train-validation split.
        How many observations are set aside for validation per CV iteration? Answer: 810!
    - CV is nevertheless less common for very large networks since efficiency is hampered k times

Convergence: early stopping

- We can keep optimizing the network weights…
  - … until the network perfectly overfits the training data, hampering the ability to generalize to unseen data

- One possible solution? **Early stopping**
  - stop convergence before MLP overfits data
  - **how?**
    - optimize weights with training data, yet **assess the loss on the validation set**
      - stop learning when the validation error increases along few iterations (evidence of overfitting)



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

Thank You

rmch@tecnico.ulisboa.pt
andreas.wichert@tecnico.ulisboa.pt