FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

NATURWISSENSCHAFTLICHE
FAKULTÄT

Artificial Neural Networks

Lecture “Mathematics of Learning” 2022/23

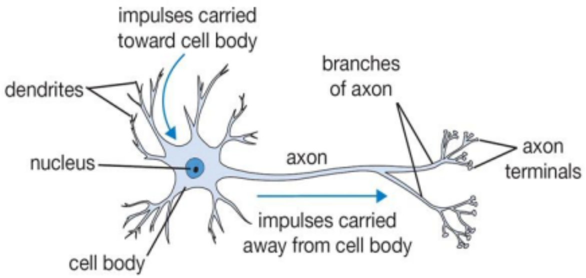
Wigand Rathmann
Friedrich-Alexander-Universität Erlangen-Nürnberg

WIRTSCHAFTS
MATHEMATIK

Towards Artificial Neural Networks

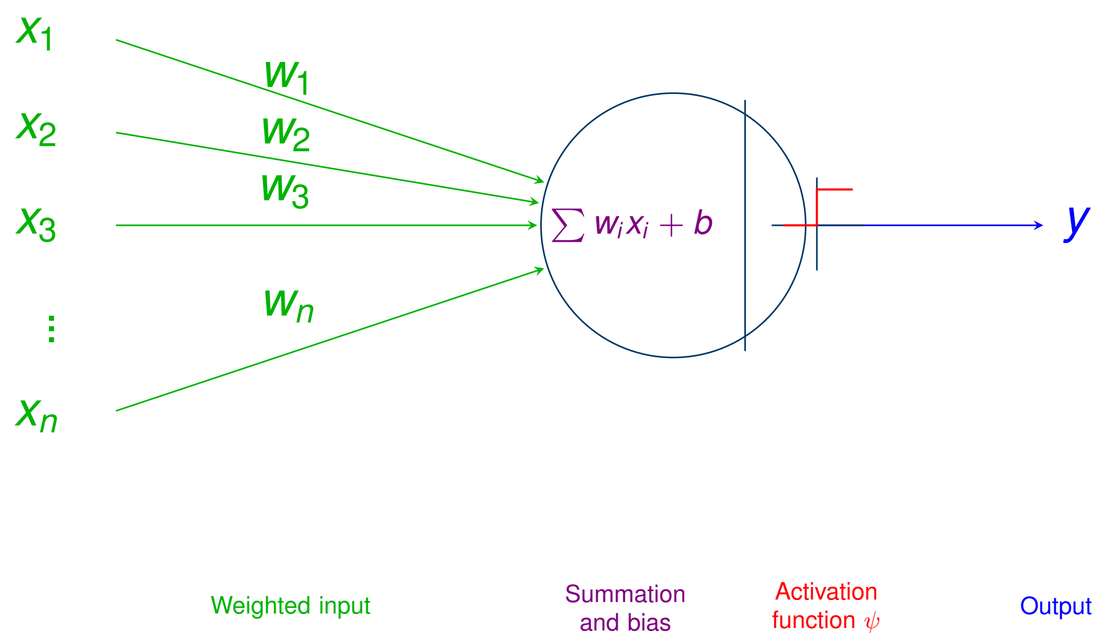
(uses material from Jonas Adler, Ozan Öktem, Daniel Tenbrinck, Philipp Wacker. See also Bishop, chapter 5.)

Historic aim (McCulloch& Pitts, 1943):
Mimic the biological processes of real neurons for Machine Learning.



Key observation: Biological neurons transmit signals **only** if the required activation energy is reached by all incoming signals.

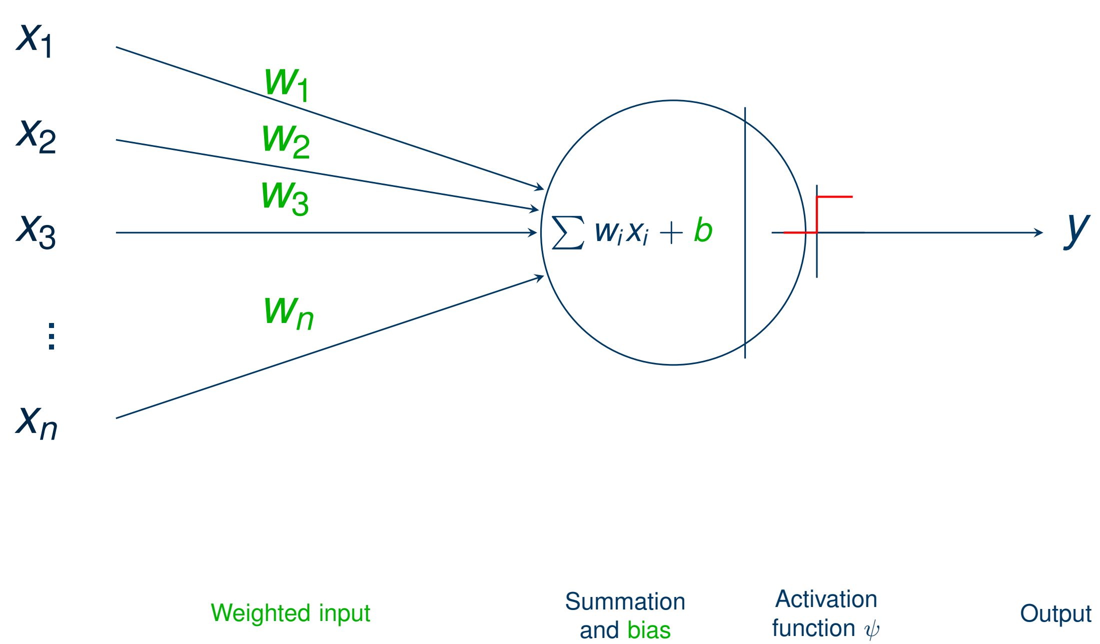
(Simple) Perceptron - an artificial neuron



The simple perceptron (Rosenblatt, 1958) is an **artificial neuron** that is able to compute mathematical functions of the form:

$$y = \psi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

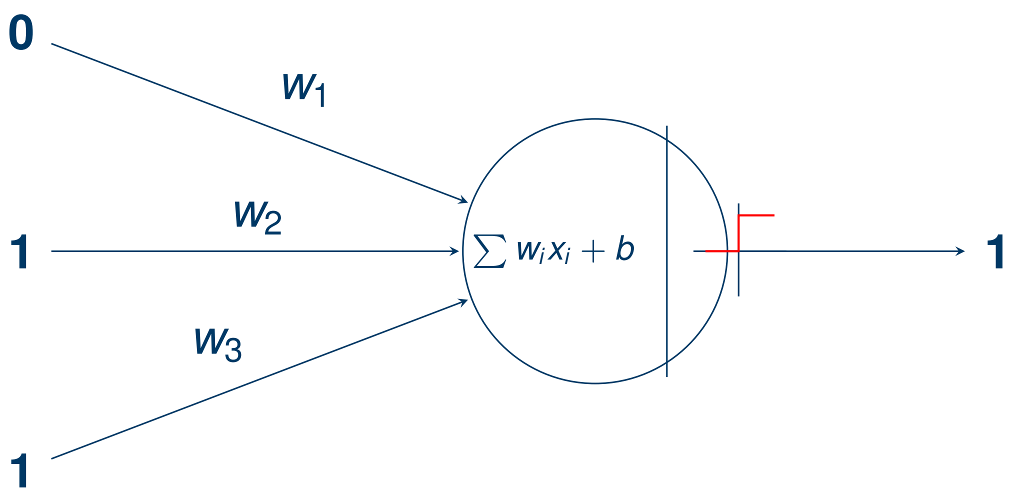
(Simple) Perceptron - an artificial neuron



For a fixed activation function $\psi\colon \mathbb{R} \to \mathbb{R}$ the behaviour of the perceptron is defined by the free parameters $(w_1, \ldots, w_n, b) = (\vec{w}, b) =: \theta \in \mathbb{R}^{n+1}$.
Thus, the perceptron realizes a parametrized map $f_\theta\colon \mathbb{R}^n \to \mathbb{R}$ with $f_\theta(\vec{x}) := f(\vec{x}; \theta) = f(x_1, \ldots, x_n; \theta)$.

An example perceptron

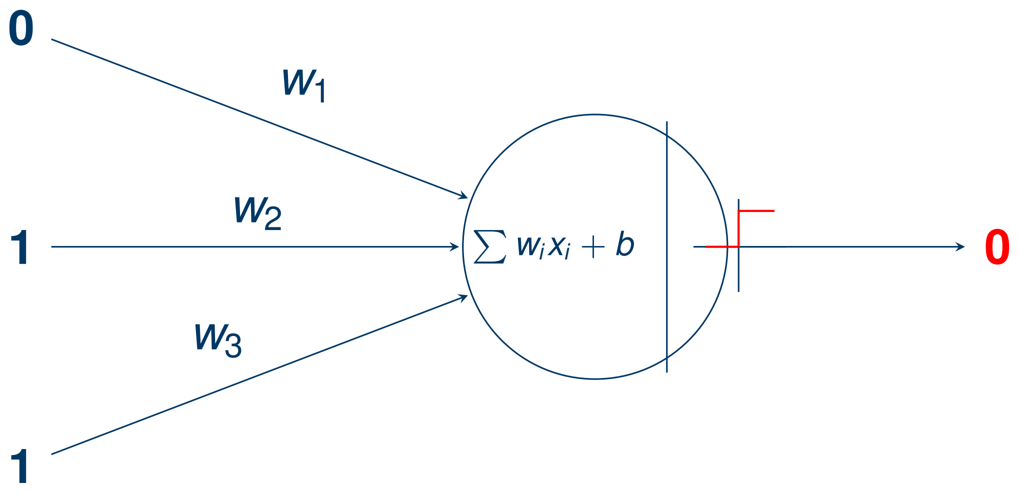
We analyze a perceptron with 3 fixed input signals $(x_1, x_2, x_3) = (0, 1, 1)$. We use the Heavyside function $H\colon \mathbb{R} \to \{0, 1\}$ as activation function ($H(z) = 0$ if $z < 0$, $H(z) = 1$ otherwise).
Set free parameters as $\theta = (w_1, w_2, w_3, b) = (1, 0, 1, -1)$.



Thus, we get $f_\theta(\vec{x}) = H([1, 0, 1] \cdot [0, 1, 1]^T - 1) = H(0) = 1$.

An example perceptron

We analyze a perceptron with 3 fixed input signals $(x_1, x_2, x_3) = (0, 1, 1)$. We use the Heavyside function $H\colon \mathbb{R} \to \{0, 1\}$ as activation function and set the free parameters as $\theta = (w_1, w_2, w_3, b) = (1, 0.5, -1, 0)$.

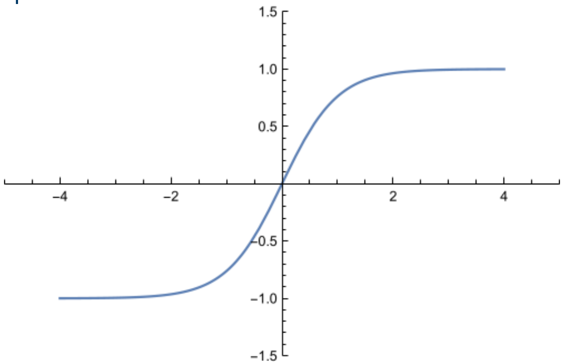


Thus, we get $f_\theta(\vec{x}) = H([1, 0.5, -1] \cdot [0, 1, 1]^T + 0) = H(-0.5) = 0$.

Continuous activation functions

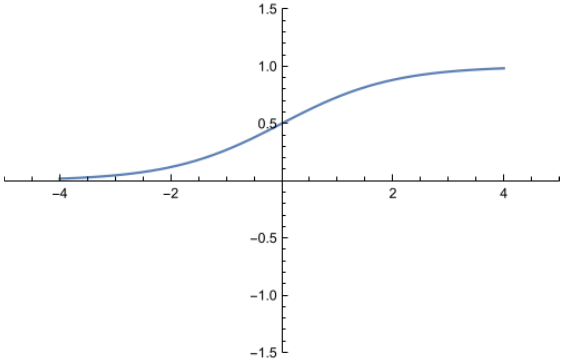
The following **continuous activation functions** are commonly used in artificial neurons due to their nice analytic properties:

Tanh



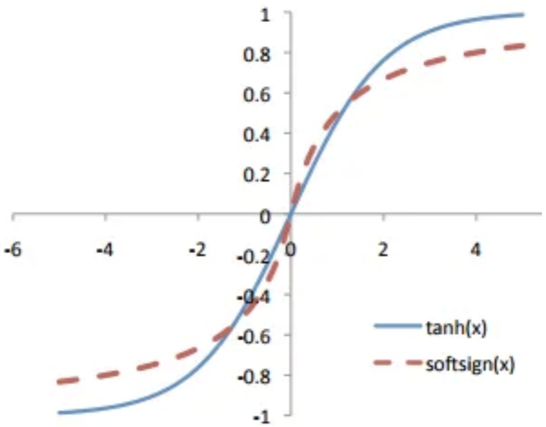
$$\psi(t) := \tanh(t)$$

Logistic



$$\psi(t) := \frac{1}{1 + e^{-t}}$$

Softsign



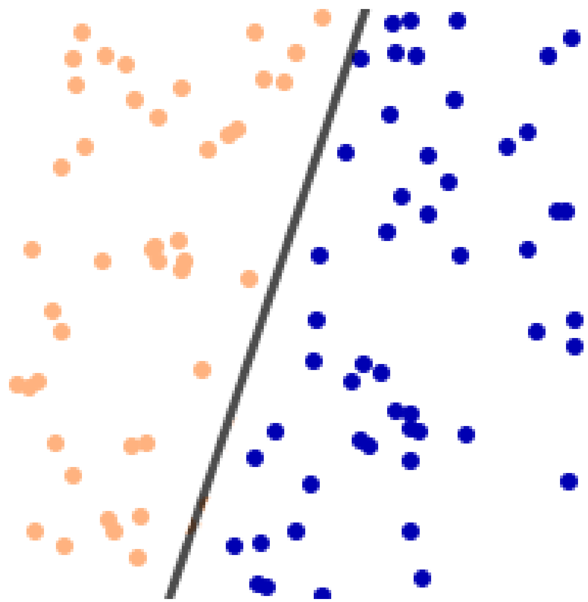
$$\psi(t) := \frac{t}{1 + |t|}$$

Observations on the perceptron

Observations:

- **weights** $\vec{w} = (w_1, \ldots, w_n)$ determine the influence of the input
  $\rightarrow$ weight $w_k = 0$ disregards respective input x_k completely
- **bias** b defines a base probability for activation of the artificial neuron
  $\rightarrow$ bias $b \ll 0$ makes an activation very unlikely
  $\rightarrow$ bias $b \gg 0$ makes an activation very likely
- simple perceptrons with Heavyside activation function realize **linear binary classifiers**
  $\rightarrow$ for more complex applications a perceptron is too restricted

Observations on the perceptron



Given a set of input data $\{\vec{x}^{(1)}, \ldots, \vec{x}^{(N)}\}$ with $x^{(i)} \in \mathbb{R}^n$, the free parameters $\theta \in \mathbb{R}^{n+1}$ induce a hyperplane that **linearly** separates the data in two classes.

$$f_\theta(\vec{x}^{(i)}) := \begin{cases} 1, & \text{if } \langle \vec{w}, \vec{x}^{(i)} \rangle + b > 0, \\ 0, & \text{otherwise.} \end{cases}$$

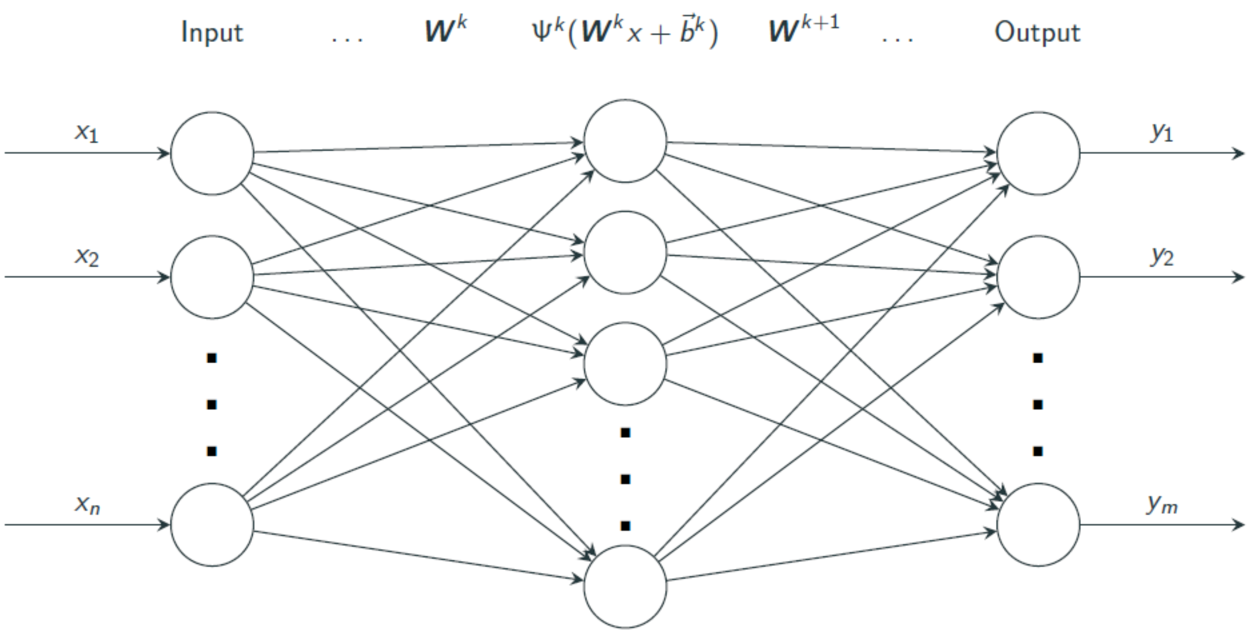
Artificial neural networks

Idea: Combine multiple perceptrons to perform more complex tasks.

- align artificial neurons in consecutive layers
  $\rightarrow$ convention: use designated input layer and output layer
  $\rightarrow$ all intermediate layers are called **hidden layer**
  $\rightarrow$ number of layers is called **depth** of the neural network
  $\rightarrow$ number of nonzero weights is called **connectivity** of the neural network
- artificial neural networks can be represented by directed graphs
- connections between neurons can be (almost) **arbitrary**
  $\rightarrow$ often there are no connections within same layer (except in recurrent neural networks)
  $\rightarrow$ certain network structures have proved to be successful for different applications, e.g., convolutional neural networks

Fully-connected feedforward neural network

Classical representation: Mappings from kth to $(k+1)$st layer:



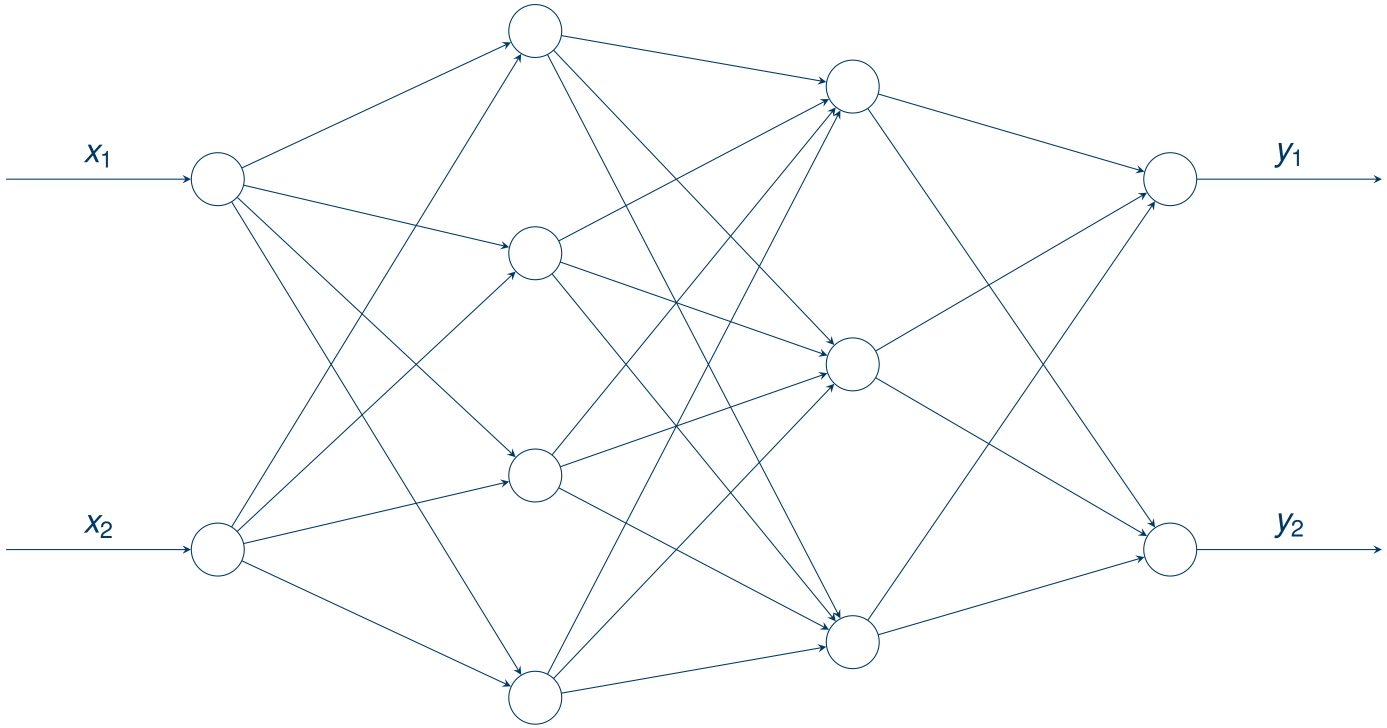
Compact representation:

Fully-connected feedforward neural network

- a fully-connected feedforward neural network can be written as a parametrized map $f_\Theta \colon \mathbb{R}^n \to \mathbb{R}^m$ that is realized by a concatenation of $d \in \mathbb{N}$ perceptron layers via
$$f_\Theta := f^d_{\Theta_d} \circ \ldots \circ f^1_{\Theta_1}$$
- each layer is a map $f^k_{\Theta_k} \colon \mathbb{R}^{n_{k-1}} \to \mathbb{R}^{n_k}$ with $f^k_{\Theta_k}(x) = \Psi^k(\boldsymbol{W}^k x + \vec{b}^k)$
- the free parameters can be written as matrix $\Theta_k = (\boldsymbol{W}^k, \vec{b}^k)$ with weights $\boldsymbol{W}^k \in \mathbb{R}^{n_k \times n_{k-1}}$ and biases $\vec{b}^k \in \mathbb{R}^{n_k}$
- the activation function Ψ^k acts pointwise on the resulting vector of the affine linear map, i.e., $\Psi^k(x_1, \ldots, x_{n_k}) := \big(\psi^k(x_1), \ldots, \psi^k(x_{n^k})\big)$ where $\psi^k \colon \mathbb{R} \to \mathbb{R}$ is the chosen activation function for this layer
- the network is fully-connected if each weight matrix $\boldsymbol{W}^k$ is fully occupied

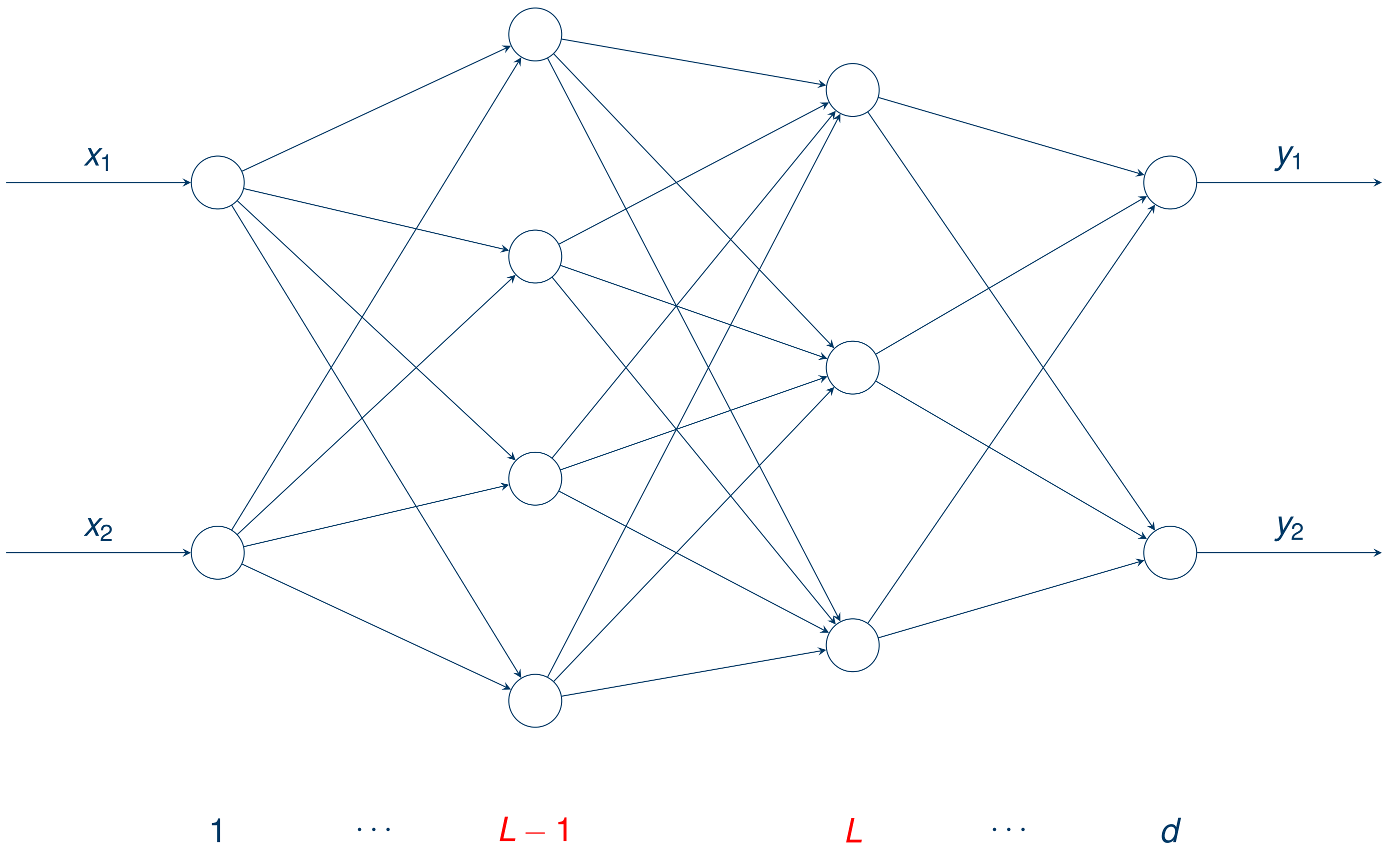
Notation

Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



Notation

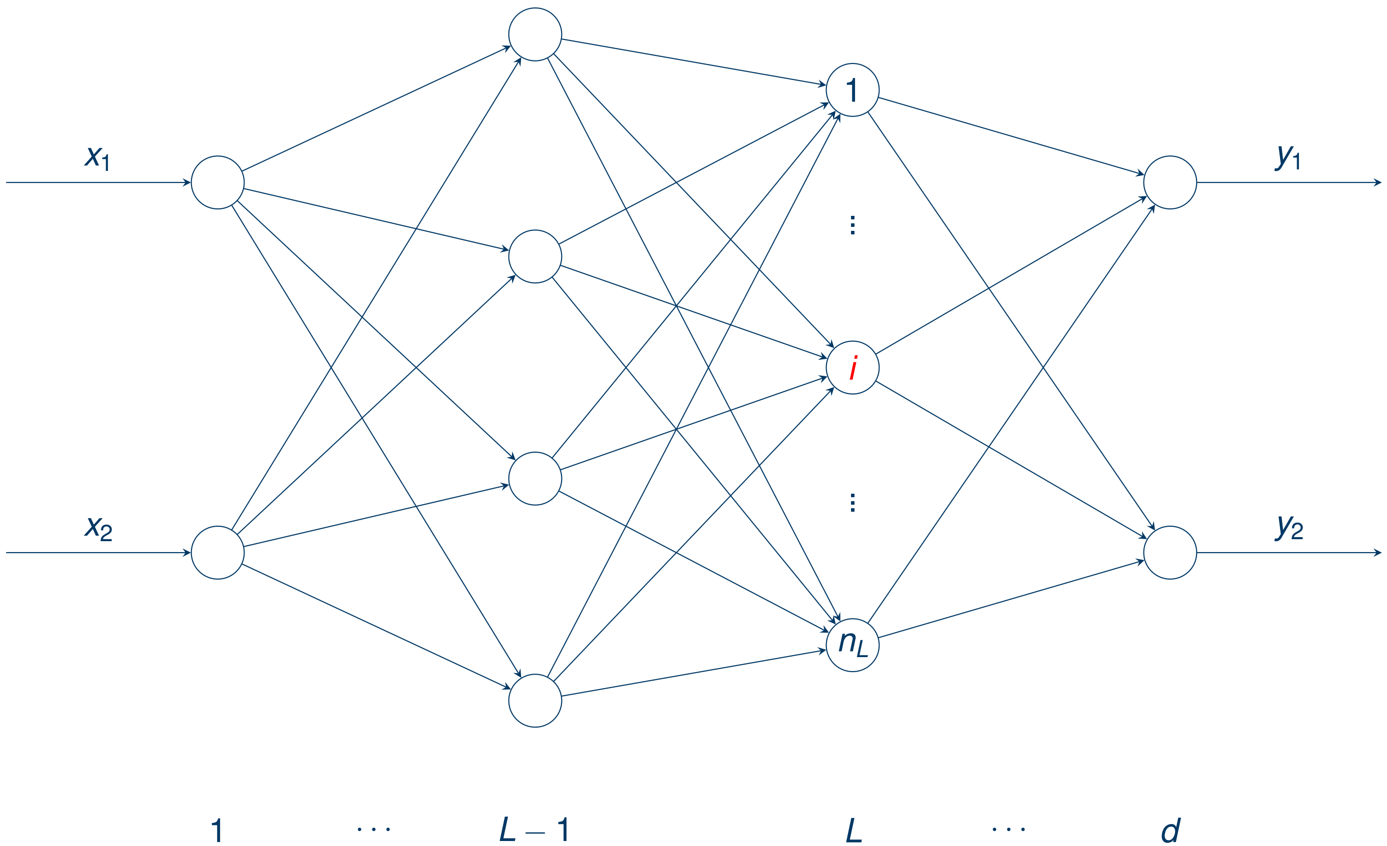
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$L = 1, \ldots, d$ is the **layer index**

Notation

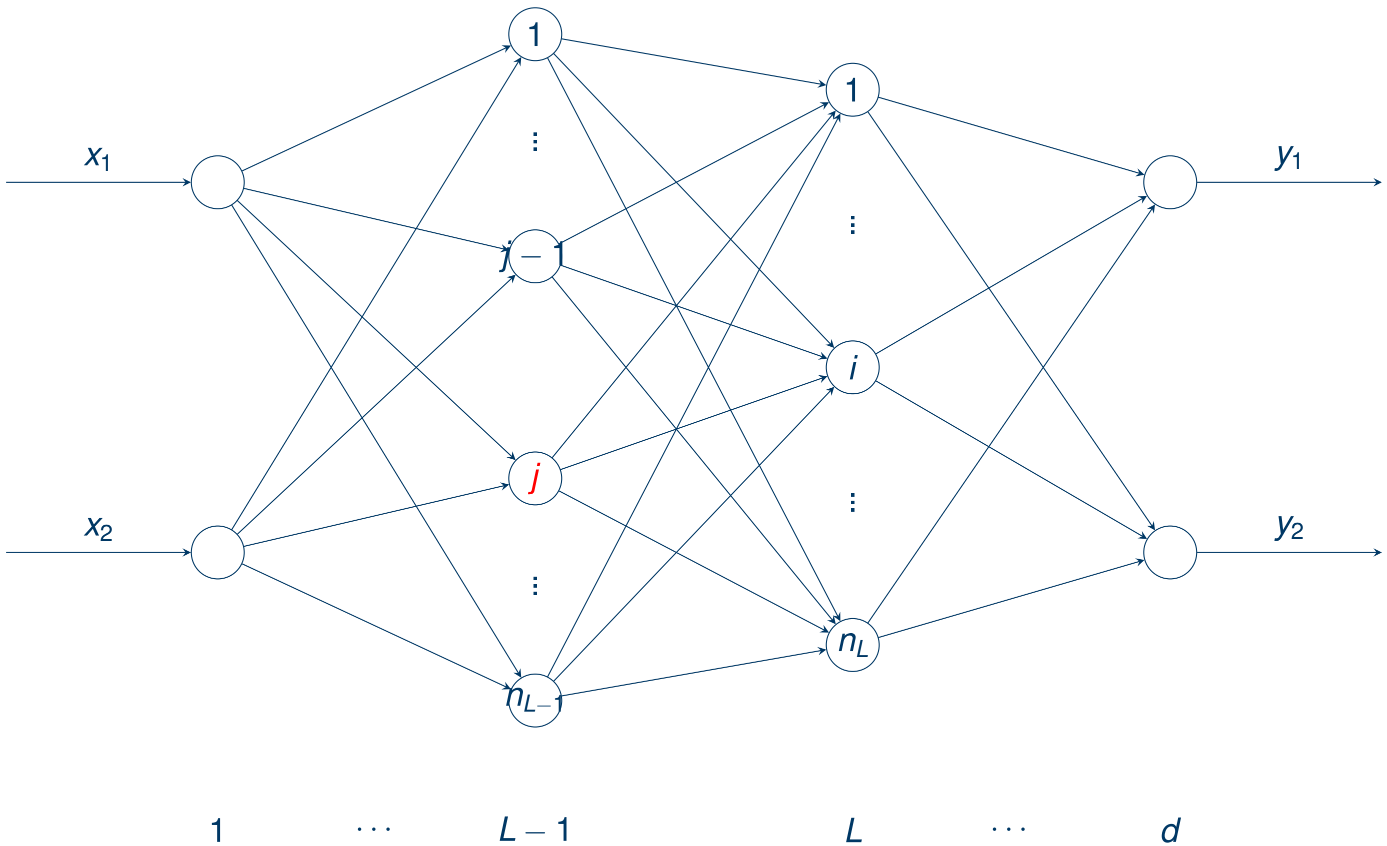
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$i = 1, \ldots, n_L$ is the **neuron index** for a layer L

Notation

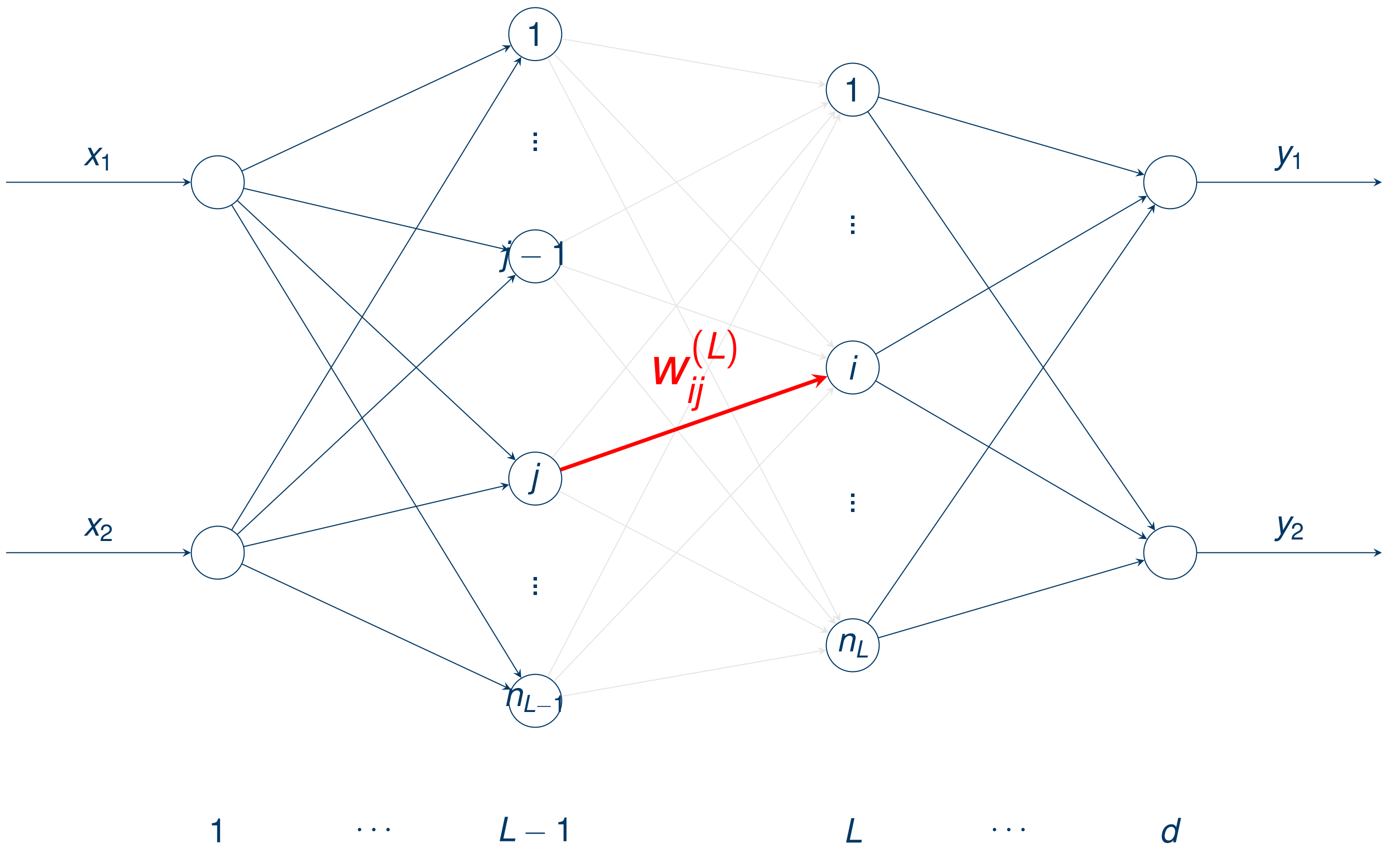
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$j = 1, \dots, n_{L-1}$ is the **neuron index** for the preceding layer $L-1$

Notation

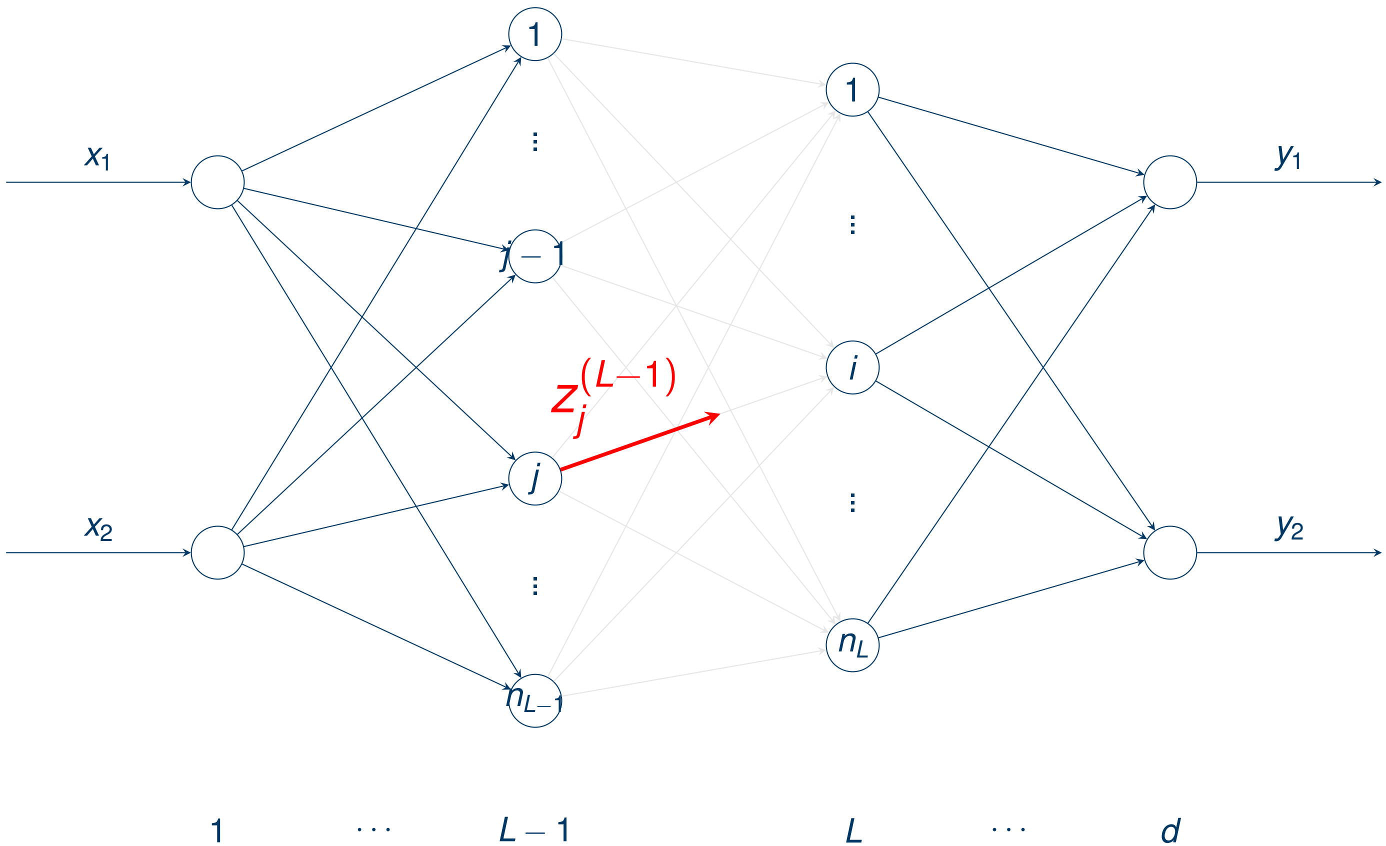
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$w_{ij}^{(L)}$ is the **weight** between neuron j in layer $L-1$ and neuron i in layer L

Notation

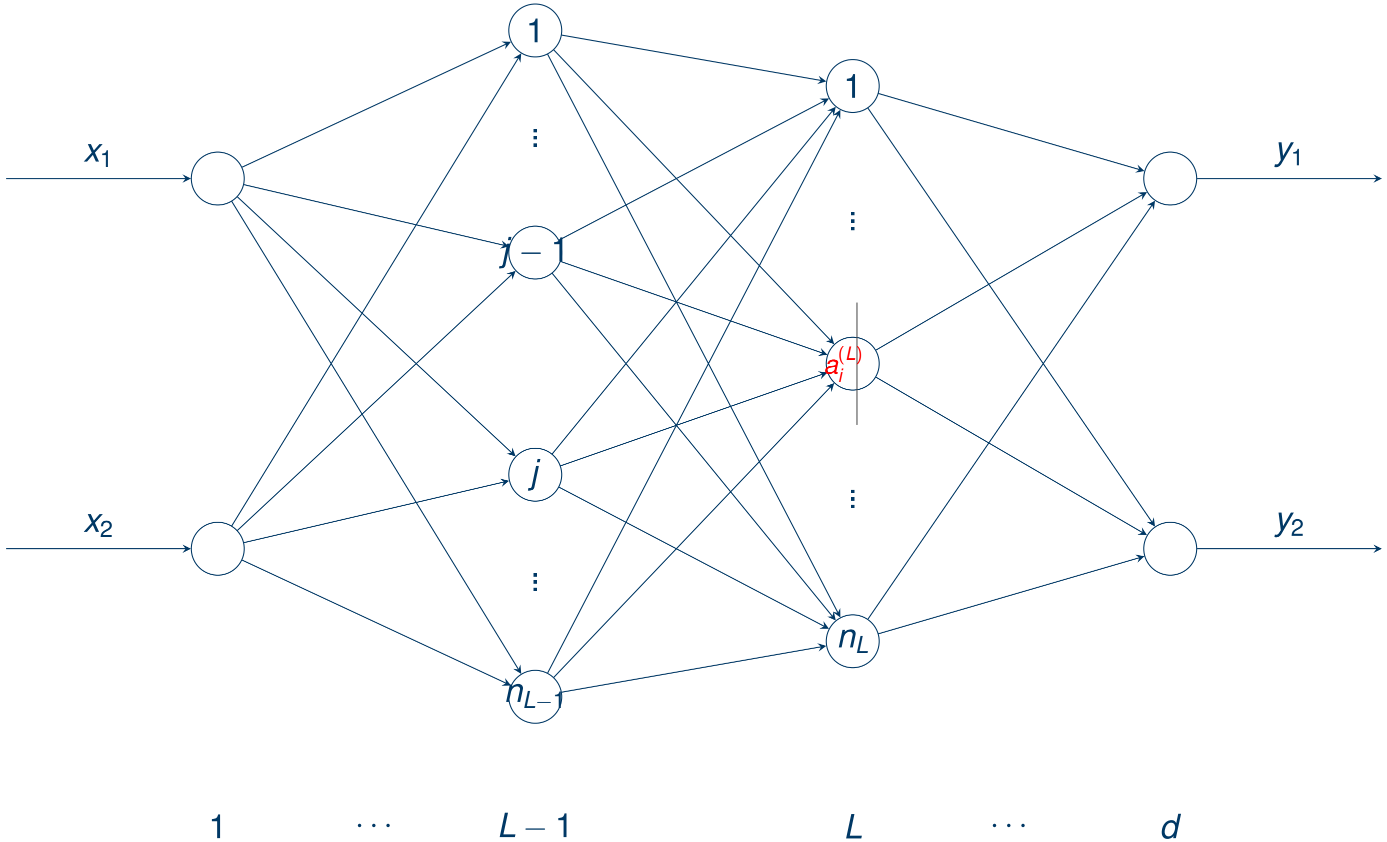
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$z_j^{(L-1)}$ is the **output** of neuron j in layer $L-1$

Notation

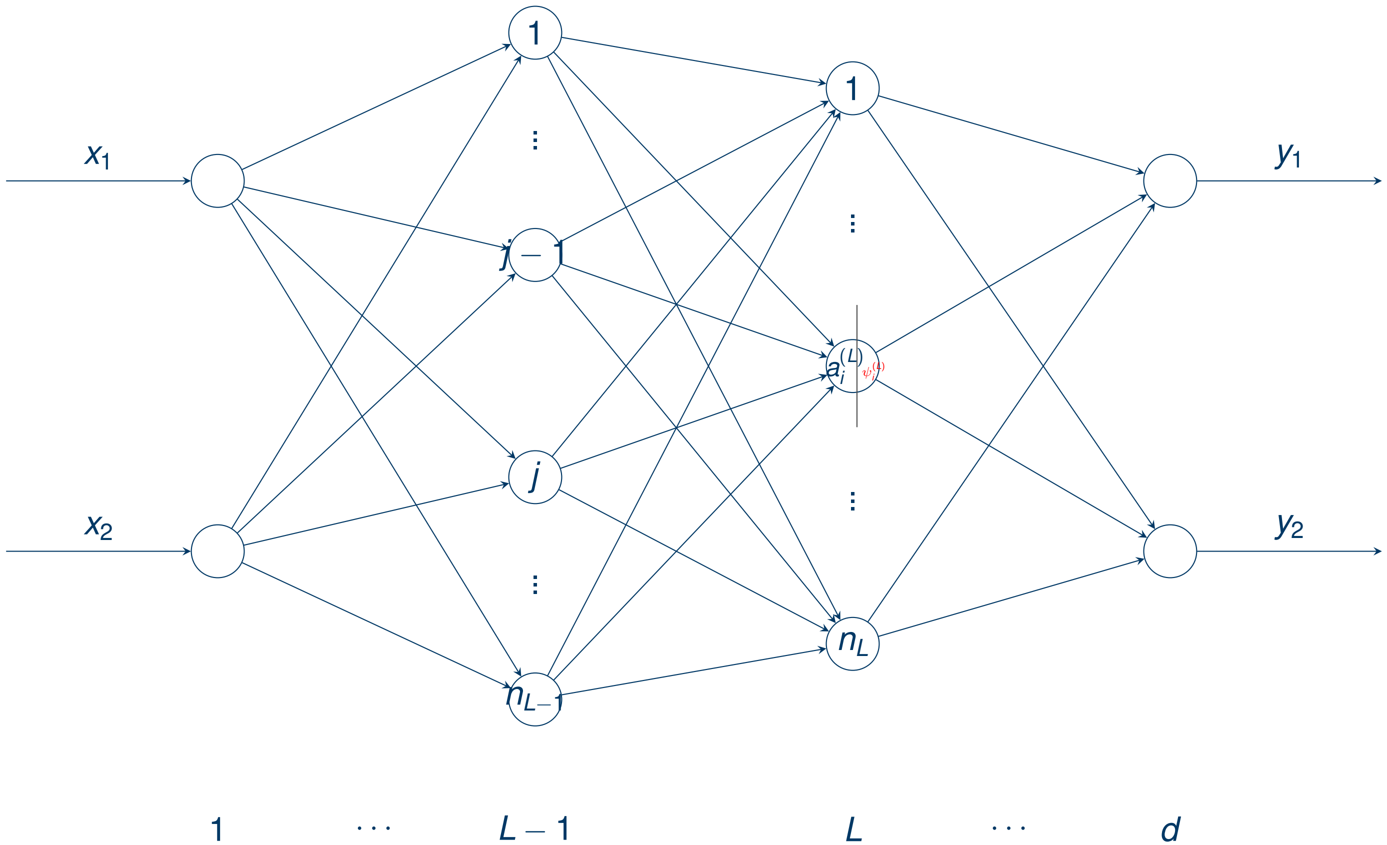
Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$a_i^{(L)} = \sum_{j=1}^{n_{L-1}} w_{ij}^{(L)} z_j^{(L-1)} + b_i^{(L)}$ is the **affine linear map** of neuron i in layer L

Notation

Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$.



$\psi_i^{(L)}$ is the **activation function** of neuron i in layer L

Notation

Let f_θ be a fully-connected feedforward network of depth $d \in \mathbb{N}$. Then we denote:

- $L = 1, \dots, d$ is the **layer index**
- $i = 1, \dots, n_L$ is the **neuron index** for a layer L
- $j = 1, \dots, n_{L-1}$ is the **neuron index** for the preceding layer $L-1$
- $w_{ij}^{(L)}$ is the **weight** between neuron j in layer $L-1$ and neuron i in layer L
- $z_j^{(L-1)}$ is the **output** of neuron j in layer $L-1$
- $a_i^{(L)} = \sum_{j=1}^{n_{L-1}} w_{ij}^{(L)} z_j^{(L-1)} + b_i^{(L)}$ is the **affine linear map** of neuron i in layer L
- $\psi_i^{(L)}$ is the **activation function** of neuron i in layer L,

For an index-free notation see the blog article by Dirk Lorenz (TU Braunschweig):

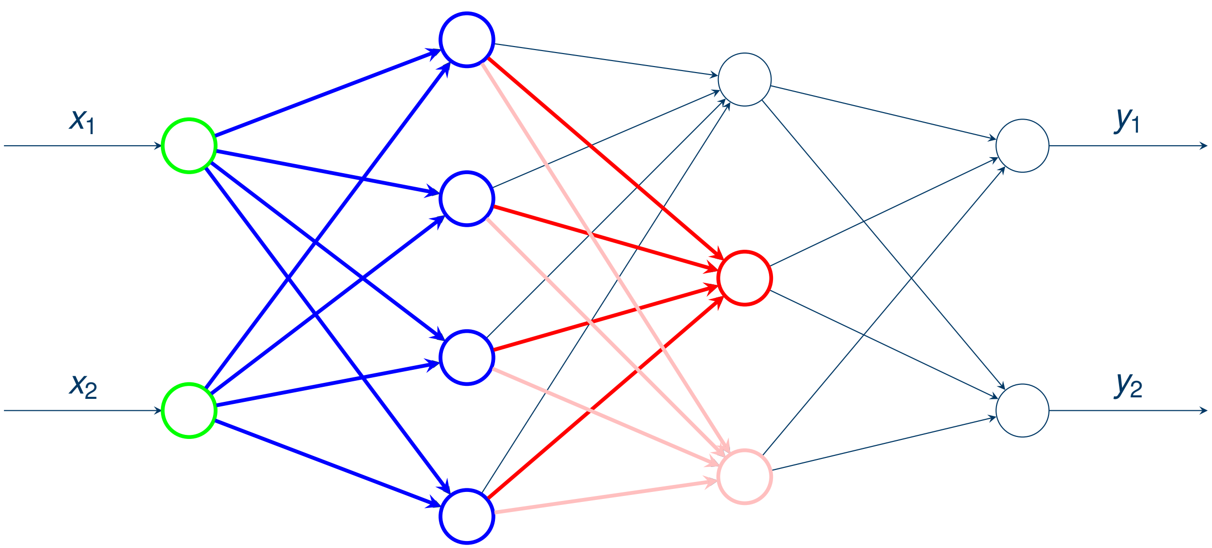
https://regularize.wordpress.com/2018/12/13/an-index-free-way-to-take-the-gradient-of-a-neural-network/

Forward propagation

We can now express the output $z_i^{(L)}$ of any neuron i in any layer L as:

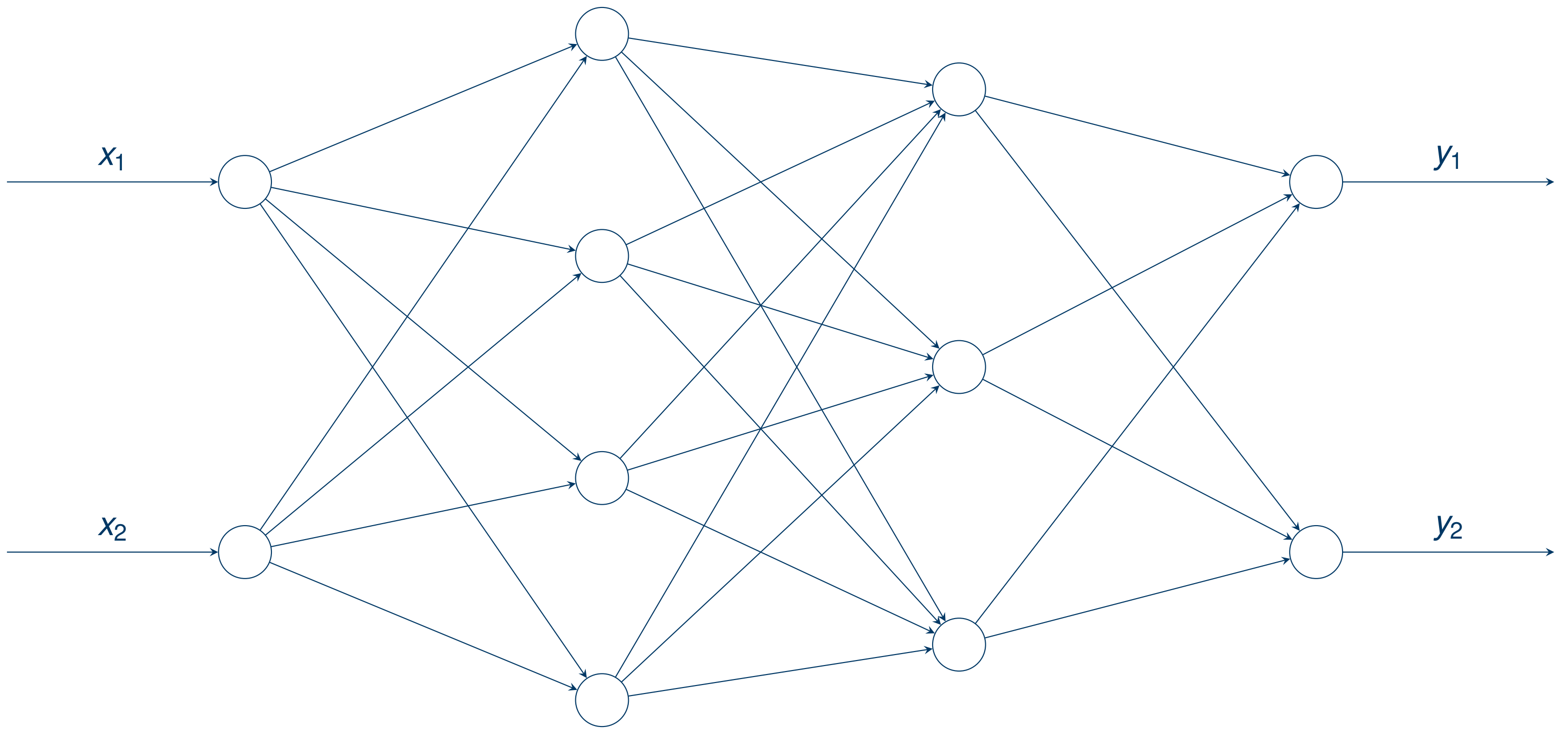
$$z_i^{(L)} = \psi_i^{(L)}(a_i^{(L)}) = \psi_i^{(L)}\left(\sum_{j=1}^{n_{L-1}} w_{ij}^{(L)} z_j^{(L-1)} + b_i^{(L)}\right)$$

To compute this expression one first has to compute the output $z_j^{(L-1)}$ of all neurons in layer $L-1$. $\rightarrow$ **Recursion**



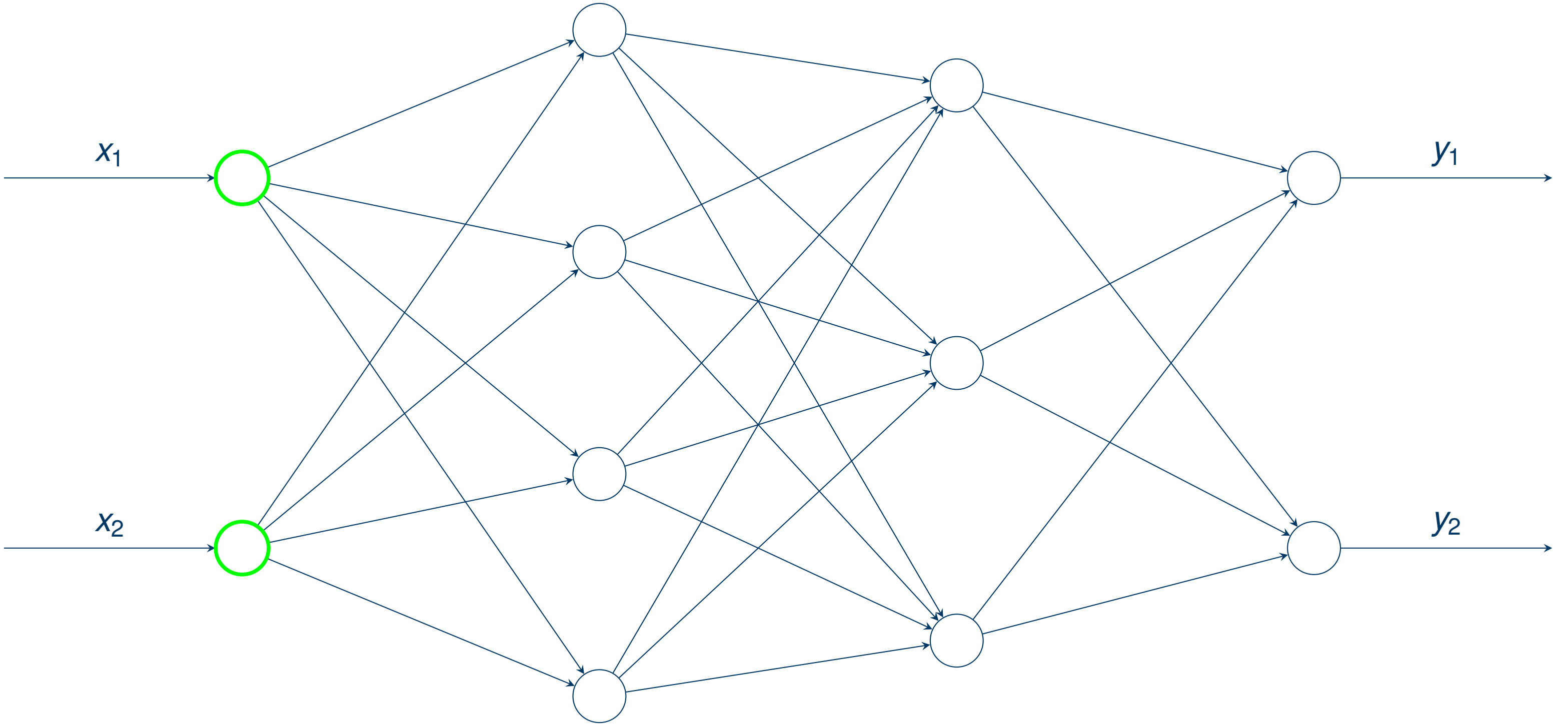
Iterative computation in forward propagation

Observation:
Instead of performing the computation of an arbitrary output $z_i^{(L)}$ recursively, it is better to start at layer $L = 1$ and **compute** and **store** the output of each layer **iteratively**.



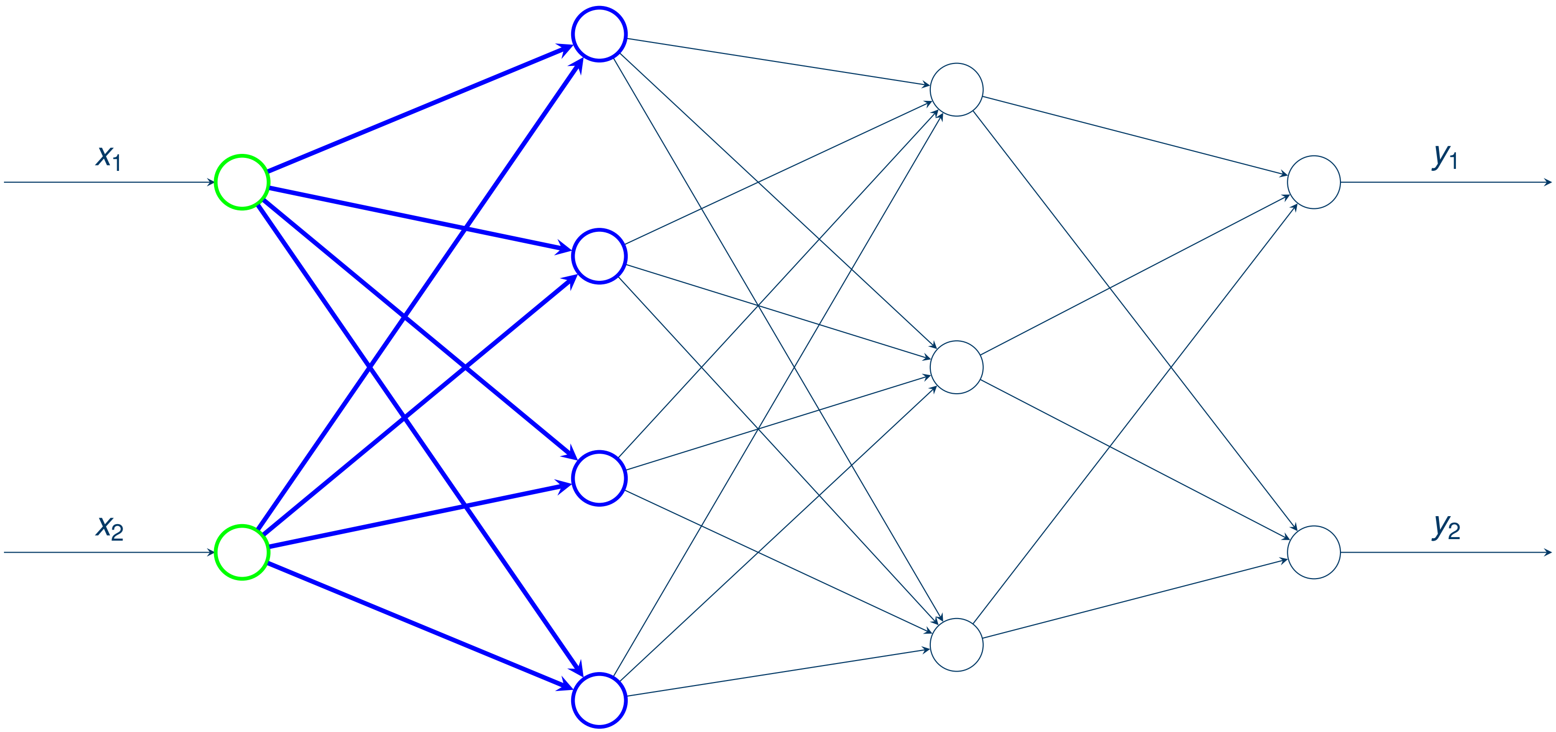
Iterative computation in forward propagation

Observation:
Instead of performing the computation of an arbitrary output $z_i^{(L)}$ recursively, it is better to start at layer $L = 1$ and **compute** and **store** the output of each layer **iteratively**.



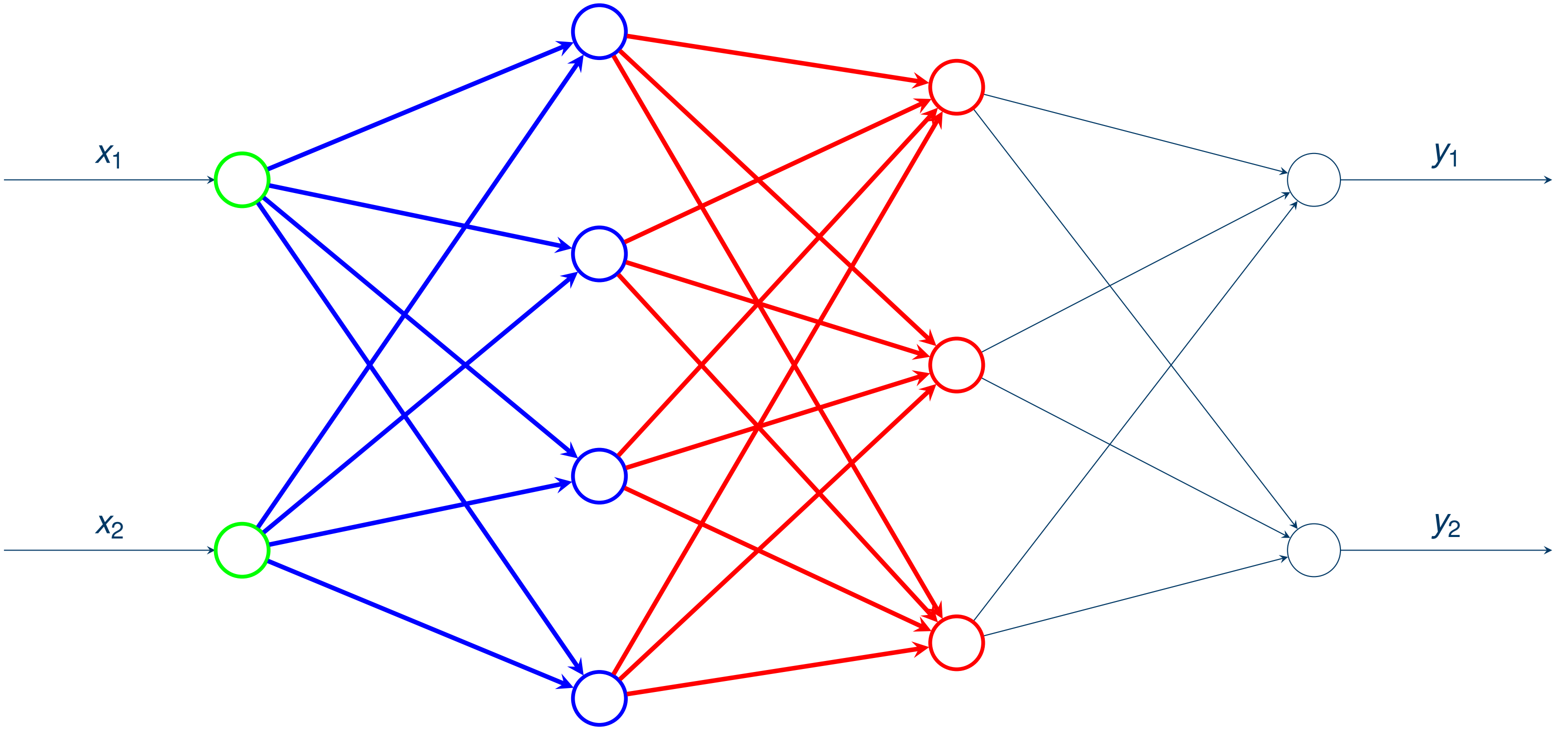
Iterative computation in forward propagation

Observation:
Instead of performing the computation of an arbitrary output $z_i^{(L)}$ recursively, it is better to start at layer $L = 1$ and **compute** and **store** the output of each layer **iteratively**.



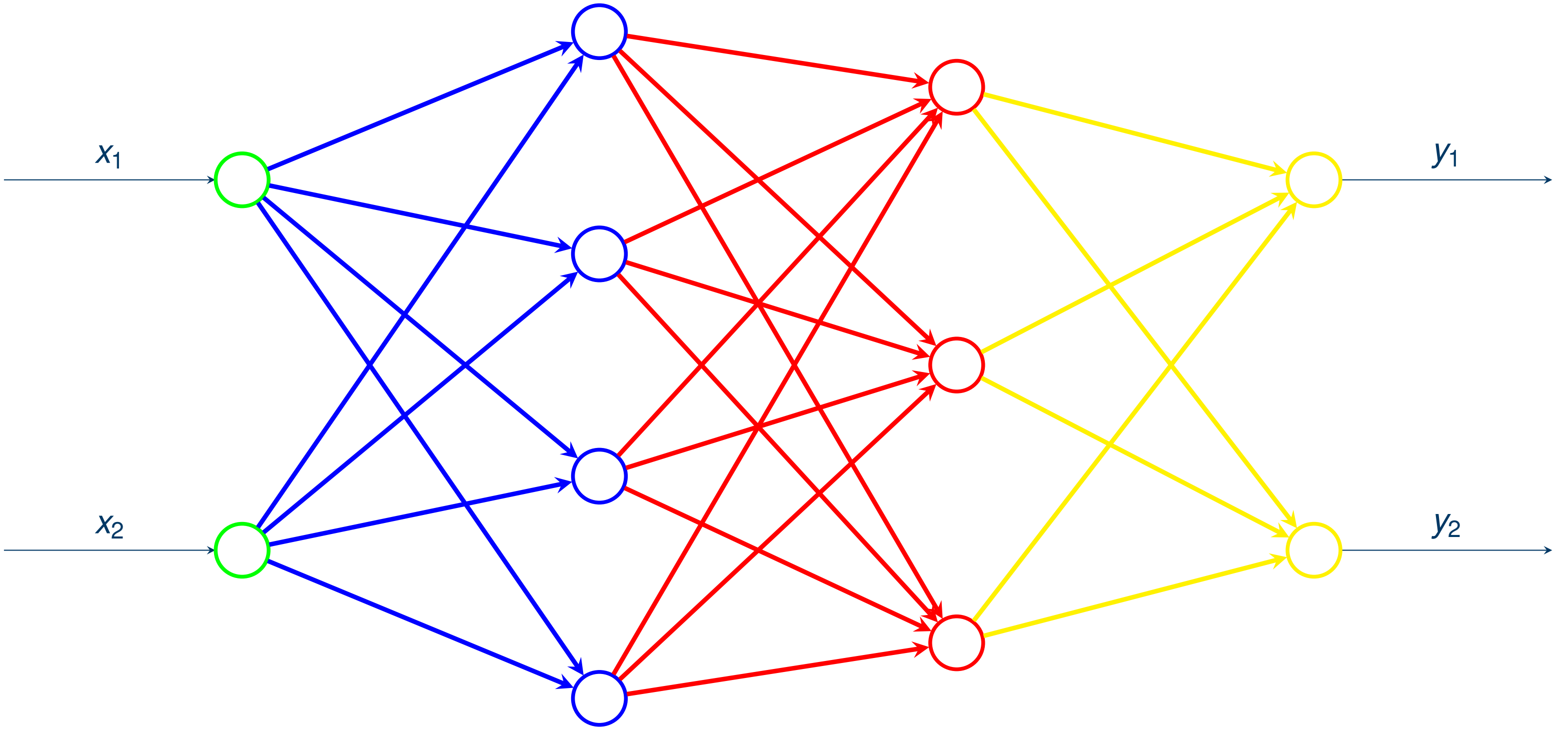
Iterative computation in forward propagation

Observation:
Instead of performing the computation of an arbitrary output $z_i^{(L)}$ recursively, it is better to start at layer $L = 1$ and **compute** and **store** the output of each layer **iteratively**.



Iterative computation in forward propagation

Observation:
Instead of performing the computation of an arbitrary output $z_i^{(L)}$ recursively, it is better to start at layer $L = 1$ and **compute** and **store** the output of each layer **iteratively**.



Motivation for neural network training

Aim:
Find good parameters Θ such that for a given set of input-output data $\{(\vec{x}^{(1)}, \vec{y}^{(1)}), \ldots, (\vec{x}^{(N)}, \vec{y}^{(N)})\} \subset X \times Y$ (training data) the parameterized map approximates the given data pairs well, i.e.,

$$f_\Theta(\vec{x}^{(i)}) \approx \vec{y}^{(i)}, \quad i = 1, \ldots, N.$$

Questions:

1. How far is the approximation by f_Θ off from the *training data*?
2. How can we measure how well the artificial neural network is performing for *other data*?

Loss function

Idea:
We measure the performance of the artificial neural network using a metric $d\colon Y \times Y \to \mathbb{R}^+$.

- many metrics are possible, which lead to different realizations of the free parameters Θ
- typical examples: mean squared error, cross-entropy, ...
- based on a chosen metric d one defines the loss function C of f_Θ with respect to the free parameters Θ as:

$$C(\Theta) \ := \sum_{i=1}^{N} d[\, f_\Theta(\vec{x}^{(i)}), \vec{y}^{(i)}].$$

Minima of the loss function

Question:
How does the loss function help us to find good parameters Θ for f_Θ?

(Ambitiuous) Goal:
Compute the global minimum of the loss function $C(\Theta)$!

- computing the global minimum for a large system of nonlinear equations is challenging (at least!)
- in the presence of at least one hidden layer: loss function C is typically non-convex

Gradient descent for neural network training

Aim:
Use iterative optimization methods to train the artificial neural network, i.e., to decrease the loss function value of C.

Idea:
Use the gradient descent method to optimize the loss function C:

$$\Theta^{k+1} = \Theta^k - \eta \nabla C(\Theta^k),$$

where Θ^0 are randomly initialized parameters.

Intuitively, we update our parameter vector Θ^k in direction of the steepest change of the loss function. The step size parameter η is often denoted as **learning rate** in the context of training artificial neural networks.

Gradient of the loss function

For a fully-connected feedforward network of depth $d \in \mathbb{N}$ the gradient of the loss function ∇C can be written as:

$$\nabla C = \begin{pmatrix} \frac{\partial C}{\partial \Theta_1} \\ \\ \vdots \\ \\ \frac{\partial C}{\partial \Theta_d} \end{pmatrix} = \begin{pmatrix} \frac{\partial C}{\partial W^1} \\ \frac{\partial C}{\partial \vec{b}^1} \\ \vdots \\ \frac{\partial C}{\partial W^d} \\ \frac{\partial C}{\partial \vec{b}^d} \end{pmatrix}$$

Question:
How can we compute the gradient ∇C of the loss function?

Simplified training setup: single training data

- f_Θ is a fully-connected feedforward neural network of depth $d \in \mathbb{N}$
- we try to train f_Θ with a single pair of training data $(\vec{x}, \vec{y}) \in X \times Y$.
- Y is a normed vector space. We choose the loss function C as the squared Euclidean distance, i.e.,

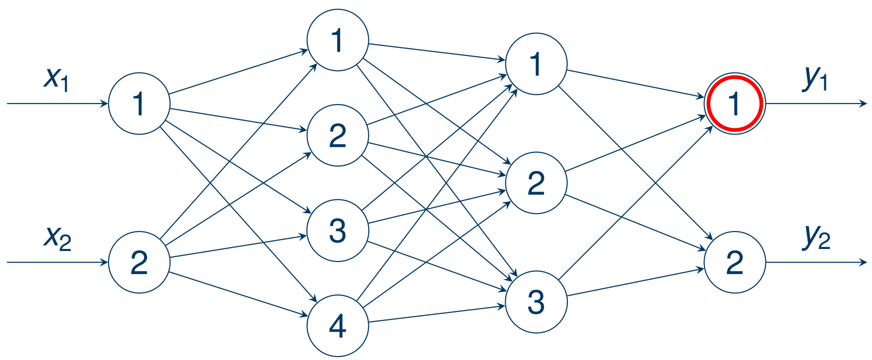
$$C(\Theta) \;=\; \frac{1}{2}\|f_\Theta(\vec{x}) - \vec{y}\|^2$$

To perform one step of the gradient descent method, we need to compute the gradient $\nabla C(\Theta)$ of the loss function, i.e., we need to compute all partial derivatives

$$\nabla C(\Theta) = \left(\frac{\partial C}{\partial \theta_k}(\Theta)\right)_{k=1,\dots,K},$$

where each θ_k is a free parameter of the neural network f_Θ, i.e., θ_k is either a **weight** or a **bias**.

Partial derivative with respect to the bias

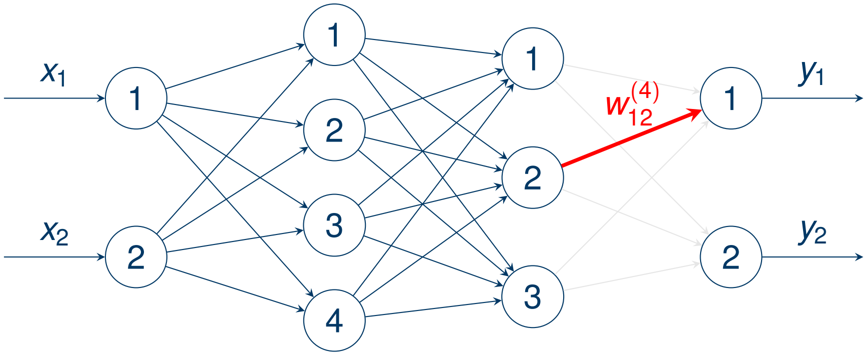


$$\frac{\partial C}{\partial b_1^{(4)}} = \frac{\partial}{\partial b_1^{(4)}}\left(\frac{1}{2}\|f_\Theta(\vec{x}) - \vec{y}\|^2\right) = (f_\Theta(\vec{x}) - \vec{y}) \cdot \left(\frac{\partial z_1^{(4)}}{\partial b_1^{(4)}}, \overbrace{\frac{\partial z_2^{(4)}}{\partial b_1^{(4)}}}^{=0}\right)^T = (z_1^{(4)} - \vec{y}_1) \cdot \frac{\partial z_1^{(4)}}{\partial b_1^{(4)}}$$

$$= (z_1^{(4)} - \vec{y}_1) \cdot \frac{\partial}{\partial b_1^{(4)}}\left(\psi_1^{(4)}(a_1^{(4)})\right) = (z_1^{(4)} - \vec{y}_1) \cdot (\psi_1'^{(4)}(a_1^{(4)})) \cdot \frac{\partial a_1^{(4)}}{\partial b_1^{(4)}}$$

$$= (z_1^{(4)} - \vec{y}_1) \cdot (\psi_1'^{(4)}(a_1^{(4)})) \cdot \underbrace{\frac{\partial}{\partial b_1^{(4)}}\left(\sum_{j=1}^{3} w_{1j}^{(4)} z_j^{(3)} + b_1^{(4)}\right)}_{=1} = \underbrace{(z_1^{(4)} - \vec{y}_1)}_{=\frac{\partial C}{\partial z_1^{(4)}}} \cdot \underbrace{(\psi_1'^{(4)}(a_1^{(4)}))}_{\frac{\partial z_1^{(4)}}{\partial a_1^{(4)}}}$$

Partial derivative with respect to weights

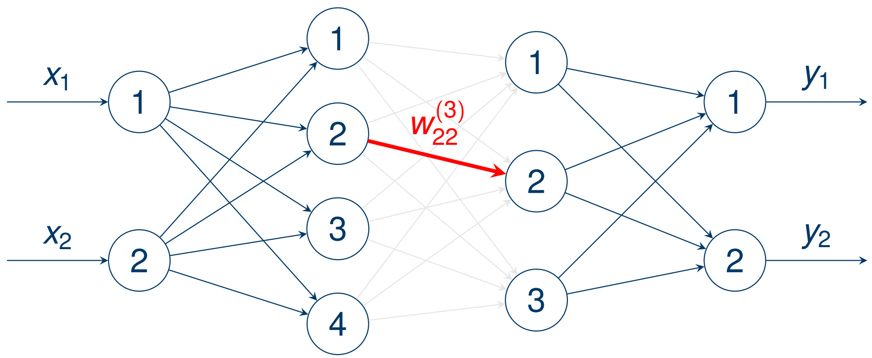


$$\frac{\partial C}{\partial w_{12}^{(4)}} = \underbrace{\frac{\partial C}{\partial z_1^{(4)}} \cdot \frac{\partial z_1^{(4)}}{\partial a_1^{(4)}}}^{=\frac{\partial C}{\partial a_1^{(4)}}} \cdot \frac{\partial a_1^{(4)}}{\partial w_{12}^{(4)}} = (z_1^{(4)} - \vec{y}_1) \cdot ({\psi'_1}^{(4)}(a_1^{(4)})) \cdot \overbrace{\frac{\partial}{\partial w_{12}^{(4)}} \left(\sum_{j=1}^{3} w_{1j}^{(4)} z_j^{(3)} + b_1^{(4)} \right)}^{=z_2^{(3)}}$$

$$= (z_1^{(4)} - \vec{y}_1) \cdot ({\psi'_1}^{(4)}(a_1^{(4)})) \cdot z_2^{(3)}$$

Observation: We already computed two terms of this partial derivative.

Partial derivative with respect to weight

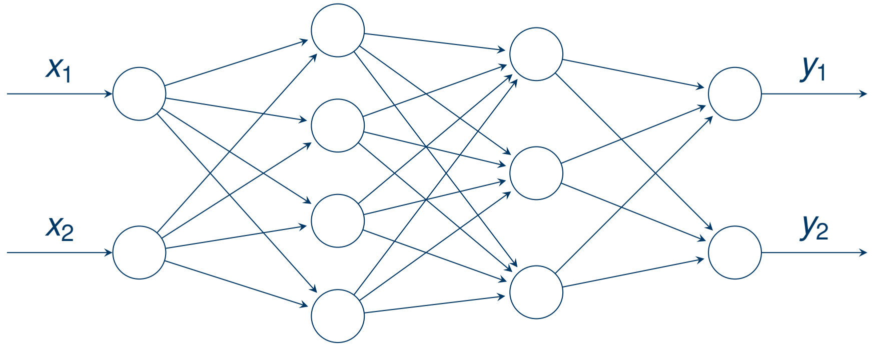


$$\frac{\partial C}{\partial w_{22}^{(3)}} = \sum_{i=1}^{2}\left(\frac{\partial C}{\partial z_i^{(4)}} \cdot \frac{\partial z_i^{(4)}}{\partial a_i^{(4)}} \cdot \frac{\partial a_i^{(4)}}{\partial z_2^{(3)}} \cdot \frac{\partial z_2^{(3)}}{\partial a_2^{(3)}} \cdot \frac{\partial a_2^{(3)}}{\partial w_{22}^{(3)}}\right) = \left(\sum_{i=1}^{2}\frac{\partial C}{\partial z_i^{(4)}} \cdot \frac{\partial z_i^{(4)}}{\partial a_i^{(4)}} \cdot \frac{\partial a_i^{(4)}}{\partial z_2^{(3)}}\right) \cdot \frac{\partial z_2^{(3)}}{\partial a_2^{(3)}} \cdot \frac{\partial a_2^{(3)}}{\partial w_{22}^{(3)}}$$

$$= \underbrace{\left(\sum_{i=1}^{2}(z_i^{(4)} - \vec{y}_i) \cdot (\psi_i'^{(4)}(a_i^{(4)})) \cdot w_{i2}^{(4)}\right) \cdot (\psi_2'^{(3)}(a_2^{(3)}))}_{=\frac{\partial C}{\partial a_2^{(3)}}} \cdot z_2^{(2)}$$

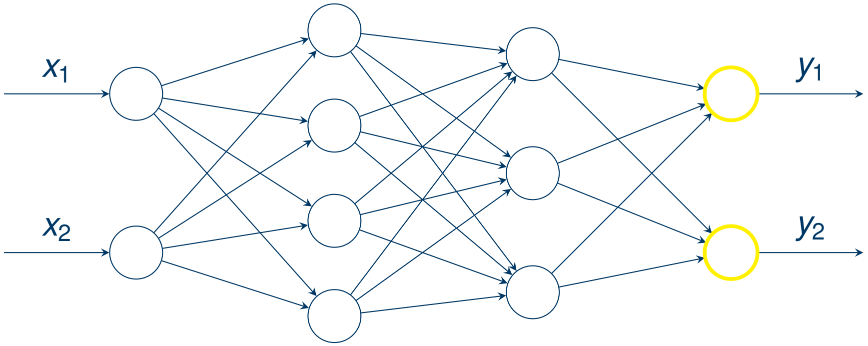
Observation: If we precompute the term $\frac{\partial C}{\partial a_2^{(3)}}$ once, we need only one multiplication for the partial derivative.

Backpropagation



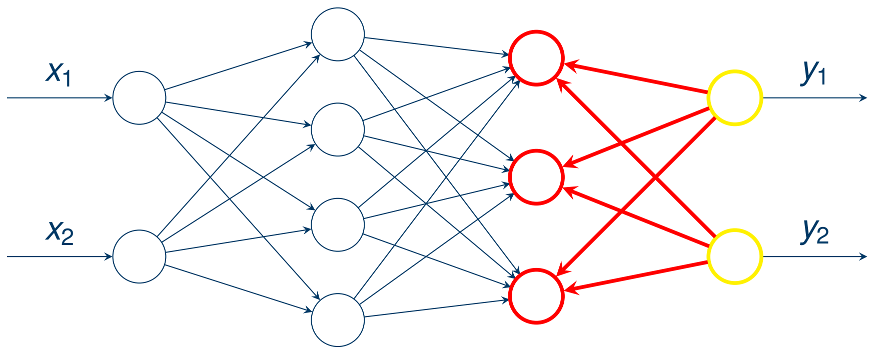
If we start computing the partial derivatives at layer *d* we can perform **backpropagation**, i.e., we can iteratively compute partial derivatives of previous layers very efficiently via the following recursive relationships.
These formula are obtained easily writing the formulae from the earlier slides in general form and summarized next. (please check!)

Backpropagation



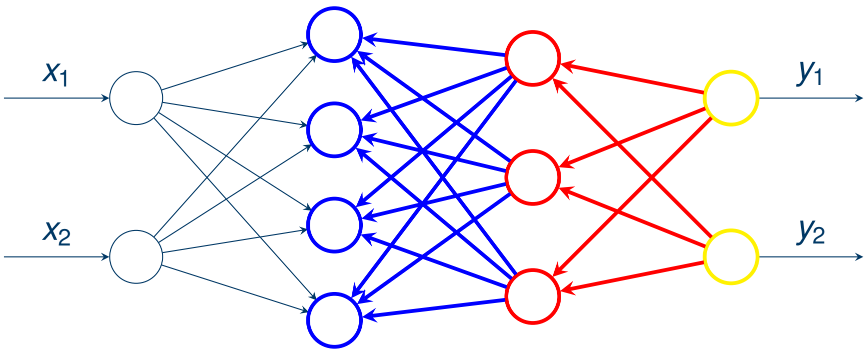
If we start computing the partial derivatives at layer *d* we can perform **backpropagation**, i.e., we can iteratively compute partial derivatives of previous layers very efficiently via the following recursive relationships.
These formula are obtained easily writing the formulae from the earlier slides in general form and summarized next. (please check!)

Backpropagation



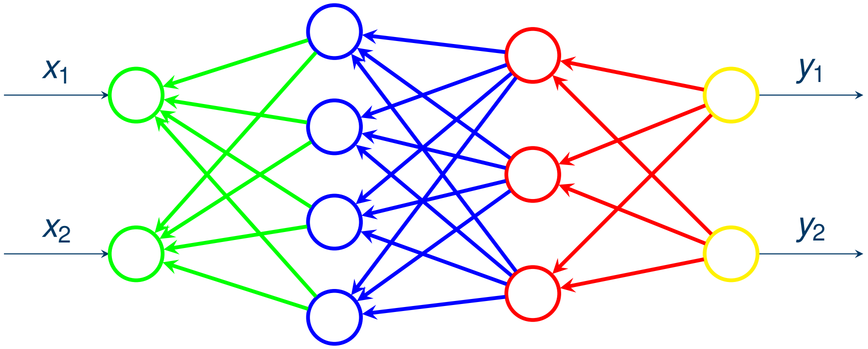
If we start computing the partial derivatives at layer *d* we can perform **backpropagation**, i.e., we can iteratively compute partial derivatives of previous layers very efficiently via the following recursive relationships. These formula are obtained easily writing the formulae from the earlier slides in general form and summarized next. (please check!)

Backpropagation



If we start computing the partial derivatives at layer *d* we can perform **backpropagation**, i.e., we can iteratively compute partial derivatives of previous layers very efficiently via the following recursive relationships.
These formula are obtained easily writing the formulae from the earlier slides in general form and summarized next. (please check!)

Backpropagation



If we start computing the partial derivatives at layer *d* we can perform **backpropagation**, i.e., we can iteratively compute partial derivatives of previous layers very efficiently via the following recursive relationships. These formula are obtained easily writing the formulae from the earlier slides in general form and summarized next. (please check!)

The mean squared loss function

Observation:
Using backpropagation as just derived we are able to compute the gradient ∇C of the loss function and hence can finally train the artificial neural network f_Θ.

...not yet really!

So far we have chosen $C(\Theta) = \frac{1}{2}\|f_\Theta(\vec{x}) - \vec{y}\|^2$ with respect to only one pair of training data. However, this would lead the neural network f_Θ to only approximate one data point.

To train with all available training data $\{(\vec{x}^{(i)}, \vec{y}^{(i)}) | i = 1, \ldots, N\}$ one typically uses the **mean squared loss function**:

$$C(\Theta) \;=\; \frac{1}{2N}\sum_{i=1}^{N}(f_\Theta(\vec{x}^{(i)}) - \vec{y}^{(i)})^2 \;=:\; \frac{1}{N}\sum_{i=1}^{N} C_0(\Theta; \vec{x}^{(i)})$$

Summary Backpropagation

Consider loss function for single data point and mean squared loss function

$$C(\theta; x^{(i)}) = \frac{1}{2}||f_\theta(x^{(i)}) - y^{(i)}||_2^2$$

$$\overline{C}(\theta) = \frac{1}{2N}\sum_{i=1}^{N}||f_\theta(x^{(i)}) - y^{(i)}||_2^2$$

For every single data point (x, y):
Forward passes: Compute $a_i^{(L)}$ and $z_i^{(L)}$ in every node i in layer L (note that $z_i^{(L)} = \psi_i^{(L)}(a_i^{(L)})$)
Backpropagation: Compute partial derivatives for ∇C in a recursive way:

- Last layer: $\frac{\partial C}{\partial b_i^{(d)}} = (z_i^{(d)} - y_i) \cdot (\psi_i'^{(d)}(a_i^{(d)}))$
- Recursively all partial derivative for the biases:
  $$\frac{\partial C}{\partial b_j^{(L-1)}} = \left(\sum_{i=1}^{n_L} \frac{\partial C}{\partial b_i^{(L)}} \cdot w_{ij}^{(L)}\right) \cdot \psi_j'^{(L-1)}(a_j^{(L-1)})$$
- Recursively all partial derivative for the weights: $\frac{\partial C}{\partial w_{ij}^{(L)}} = \frac{\partial C}{\partial b_i^{(L)}} \cdot z_j^{(L-1)}$

Summary for a simple training algorithm

Train an artificial neural network:

1. Set a fixed **learning rate** η, e.g., $\eta = 0.001$
2. **Initialize** all weights and biases of Θ^0 **randomly**, e.g., $\theta_i = \mathcal{N}(0, 1)$ i.i.d. normally distributed
3. Repeat until **no significant improvement** is achieved
   a) Loop over all pairs $(\vec{x}^{(i)}, \vec{y}^{(i)})$ in training set
      i) Compute a forward pass of the neural network $f_{\Theta^k}(\vec{x}^{(i)})$ and store intermediate results
      ii) Use intermediate results of forward pass and backpropagation to compute the gradient $\nabla C_0(\Theta^k; \vec{x}^{(i)})$
   b) compute average $\nabla \overline{C}(\Theta^k)$ of all N gradients $\nabla C_0(\Theta^k; \vec{x}^{(i)})$
   c) update parameters using **gradient descent**:
   $\Theta^{k+1} = \Theta^k - \eta \nabla \overline{C}(\Theta^k)$

Conclusions

- Training a neural network involves nonconvex optimization

  $\rightarrow$ many local minima!
- gradient descent method can be used to update the parameters Θ
- a forward pass can be computed efficiently by iteratively updating each layer starting at $L = 1$
- backpropagation can efficiently compute the gradient of the loss function by iteratively updating each layer starting at $L = d$
- gradient descent methods for training are computationally expensive when the training data set is large
- Next week, we will see a way out via *stochastic gradient methods.*

Thank you for your attention!