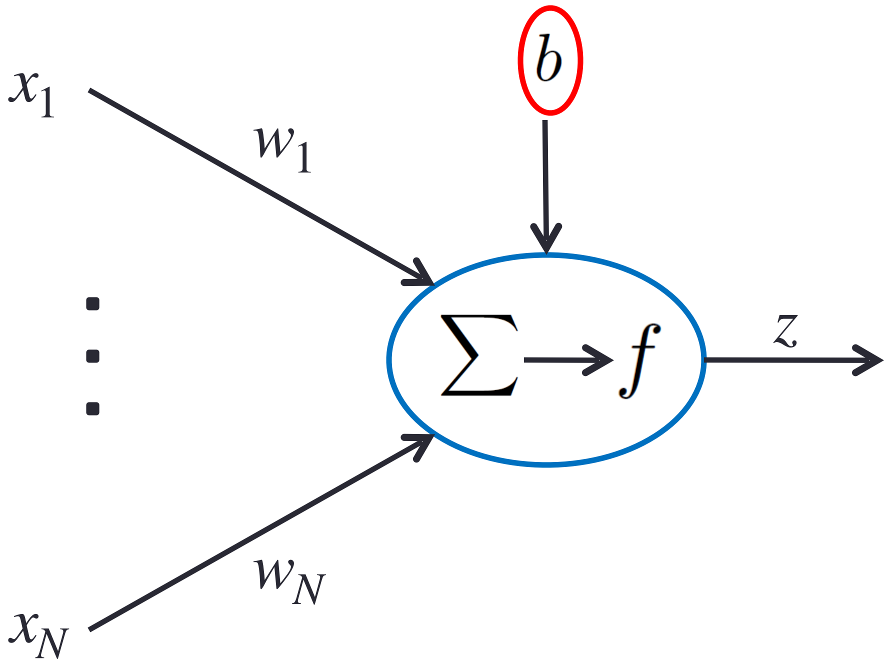
NEURAL NETWORKS (PART 1)

Artificial neuron model



Activation:

$$a = b + \sum_{i=1}^{N} x_i w_i$$

Output:

$$z = f(a)$$

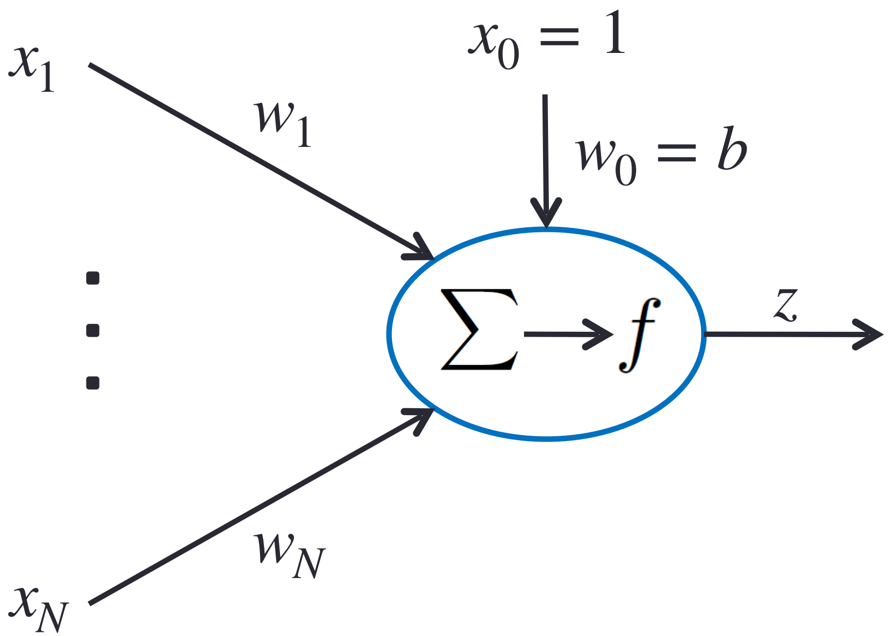
$x_1 \ldots x_N$ - inputs

$w_1 \ldots w_N$ - weights

b - bias

f - activation function

Artificial neuron model



Activation:

$$a = \sum_{i=0}^{N} x_i w_i$$

$$a = w^T x$$

Output:

$$z = f(a)$$

$$z = f(w^T x)$$

$x_1 \ldots x_N$ - inputs

$w_1 \ldots w_N$ - weights

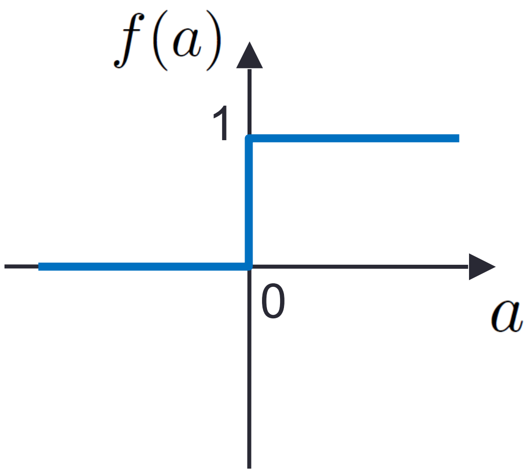
b - bias

f - activation function

$$w = \begin{bmatrix} b \\ w_1 \\ \vdots \\ w_N \end{bmatrix} \in \mathbb{R}^{N+1} \qquad x = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^{N+1}$$

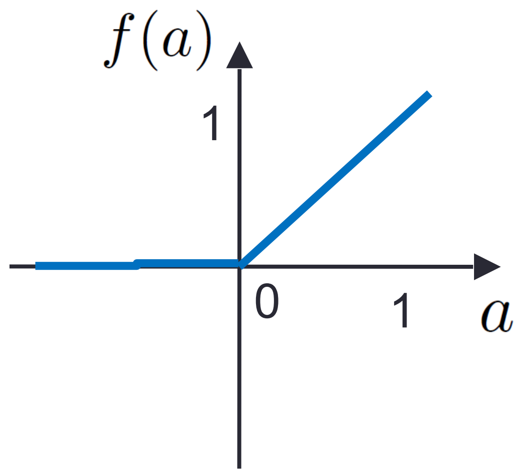
Activation functions (most common)

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



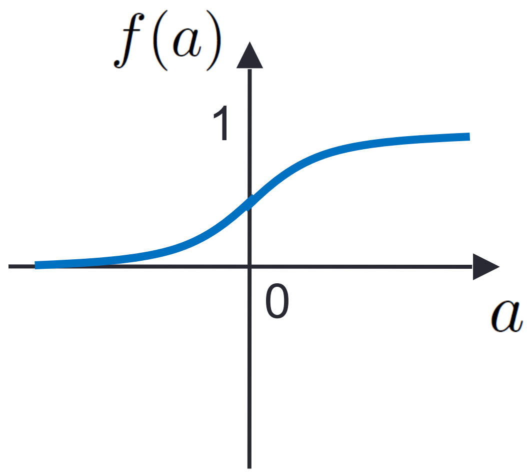
Heaviside step function
Threshold Logic Unit (TLU)
Perceptron
(classification)

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



RELU function
(nonlinear)

$$f(a) = \frac{1}{1 + e^{-a}}$$

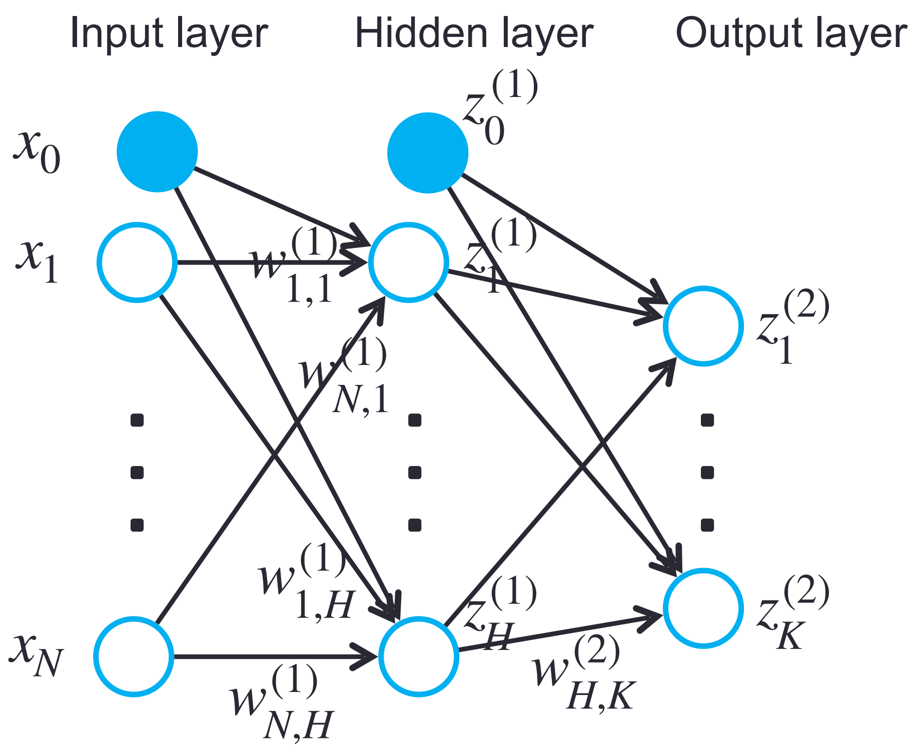


Sigmoid function
Feedforward networks
(nonlinear, classification)

FEEDFORWARD NEURAL NETWORK (MULTILAYER PERCEPTRON)

Feedforward layer architecture

- Neurons are organized in layers (Multilayer Perceptrons)
- Input information is propagated from input neurons towards the output ones



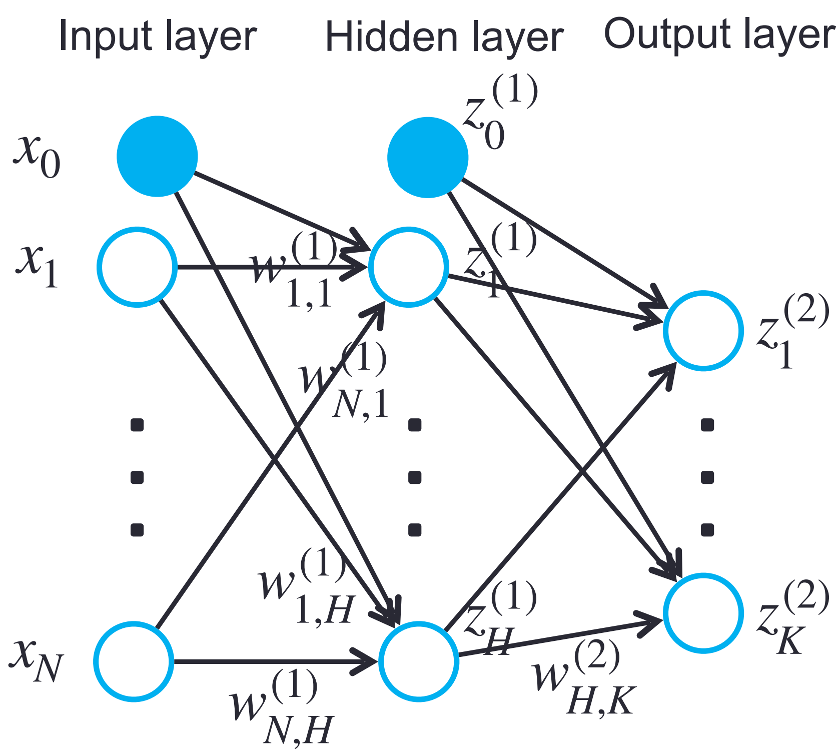
input $x \in \mathbb{R}^{N+1}$

output $z^{(2)} \in \mathbb{R}^{K}$

weights $w^{(1)} \in \mathbb{R}^{(N+1)\times H}$

$w^{(2)} \in \mathbb{R}^{(H+1)\times K}$

Feedforward layer architecture



input	$x \in \mathbb{R}^{N+1}$
output	$z^{(2)} \in \mathbb{R}^{K}$
weights	$w^{(1)} \in \mathbb{R}^{(N+1)\times H}$
	$w^{(2)} \in \mathbb{R}^{(H+1)\times K}$

The network implements the function:

Hidden neuron $z_j^{(1)} = f_j^{(1)}\left(\sum_{i=0}^{N} w_{ij} x_i\right)$

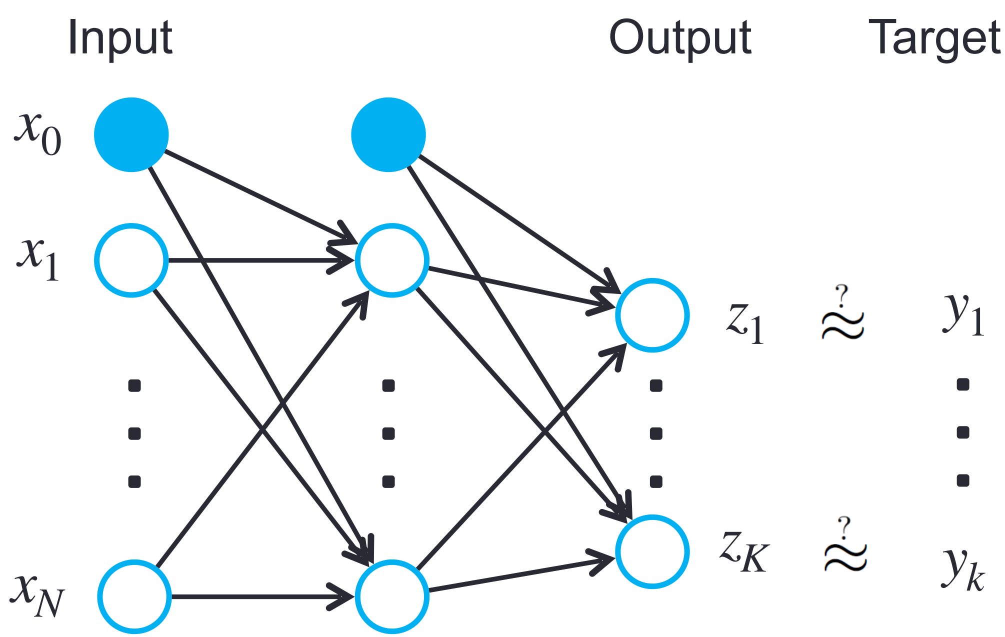
Output neuron $z_k^{(2)} = f_k^{(2)}\left(\sum_{j=0}^{H} w_{jk}^{(2)} z_j^{(1)}\right) = f_k^{(2)}\left(\sum_{j=0}^{H} w_{jk}^{(2)} f_j^{(1)}\left(\sum_{i=0}^{N} w_{ij}^{(1)} x_i\right)\right)$

Hidden neurons (units)

- Situated in **hidden layers** between the input and the output.
- They allow a network to learn **non-linear** functions and to represent combinations of the input features.

TRAINING (LEARNING) AND TESTING

Learning = minimizing training error (loss)

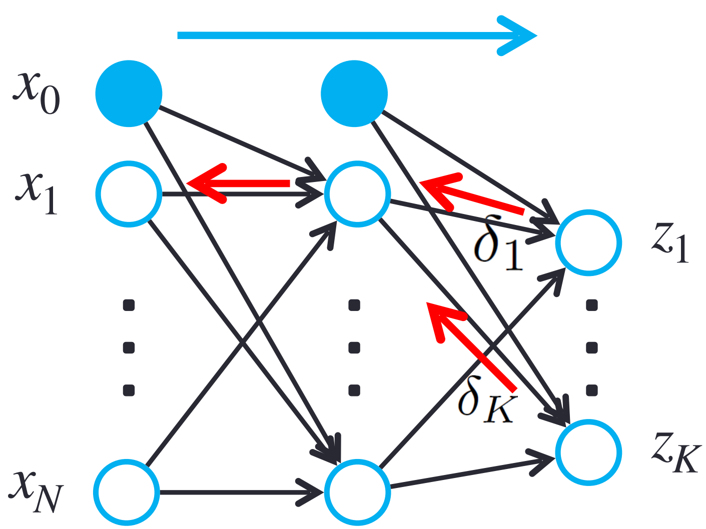


Learning:
weights are optimized (iteratively updated) while the training error is minimized.

e.g., $$E = \frac{1}{2}\sum_{k=1}^{K} e_k^2 = \frac{1}{2}\sum_{k=1}^{K}(z_k - y_k)^2$$

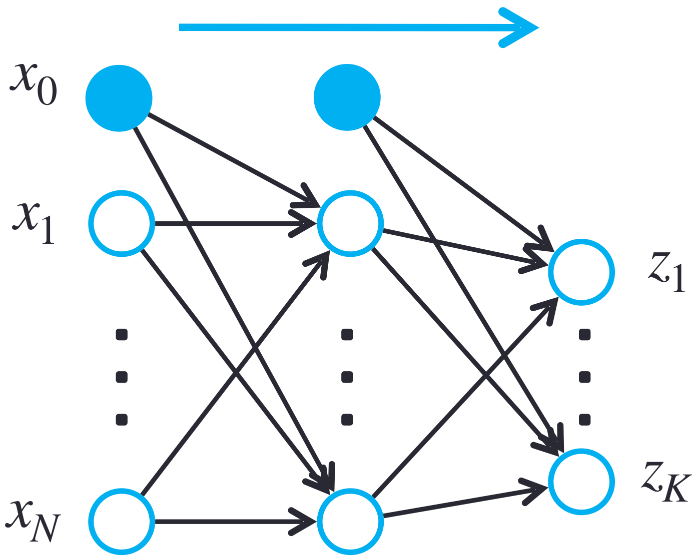
Training: Backpropagation algorithm

- For learning (updating of randomly initialized weights) the gradient of the error function is needed.
- The gradient of the error function is calculated by the local exchange of messages in 2 passes:
  - **Forward:** Calculate activations and outputs of all neurons z
  - **Backward:** Calculate errors δ and propagate them back



Testing

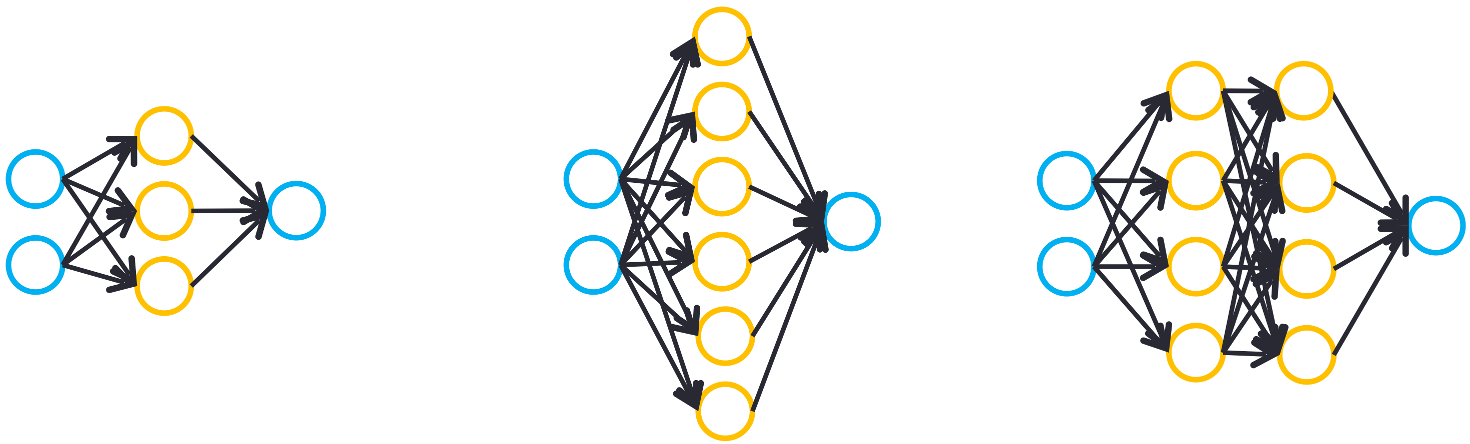
- **ONLY Forward pass:** Calculate activations and outputs of all neurons



REGULARIZATION IN NEURAL NETWORKS

Regularization of NN

- How many hidden layers and how many neurons?
  - Fewer – risk of underfitting
  - More – risk of overfitting



Weight decay

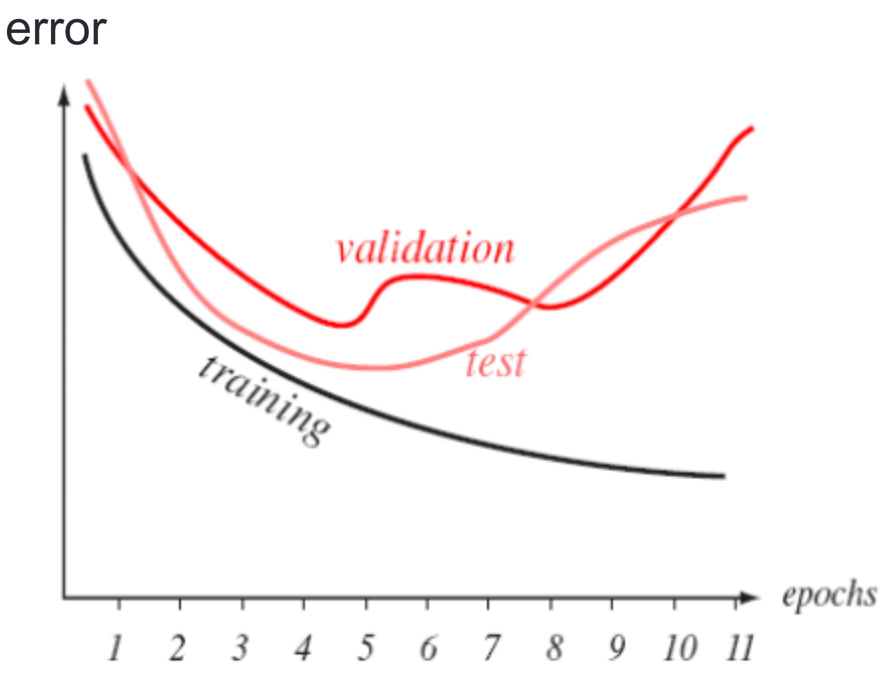
- "Weight decay" is an L_2 norm regularization for neural networks.
- The weights of a NN will be an additional term in an Error function:

$$E(\boldsymbol{w}) = MSE(\boldsymbol{w}) + \frac{\lambda}{2}||\boldsymbol{w}||^2$$

Early stopping

- A form of regularization based on the scheme of model selection

- Steps:
  - The weights are initialized to small values
  - Stop when the error on validation data increases



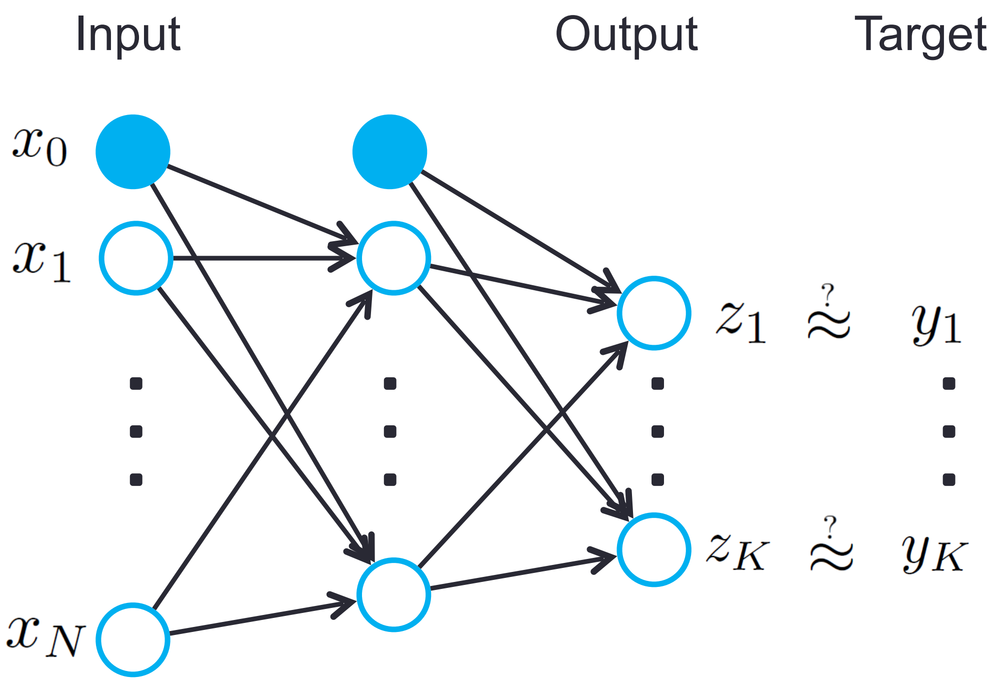
IMPORTANT DETAILS OF NEURAL NETWORK TRAINING

Important details of training

- If you use another library (than sklearn), or implement everything from scratch, you will need to choose:
  - Activation function also in the *output* layer
  - Loss function appropriate for your data set
  - Batch size - number of training examples used in forward-pass, after which the backward-pass is performed.

  - In case of multi-class classification, sometimes the labels are discrete numbers (e.g., if number of classes n = 10, labels are discrete numbers 0-9), but sometimes they need to be transformed into “1-out-of-n” coding scheme.

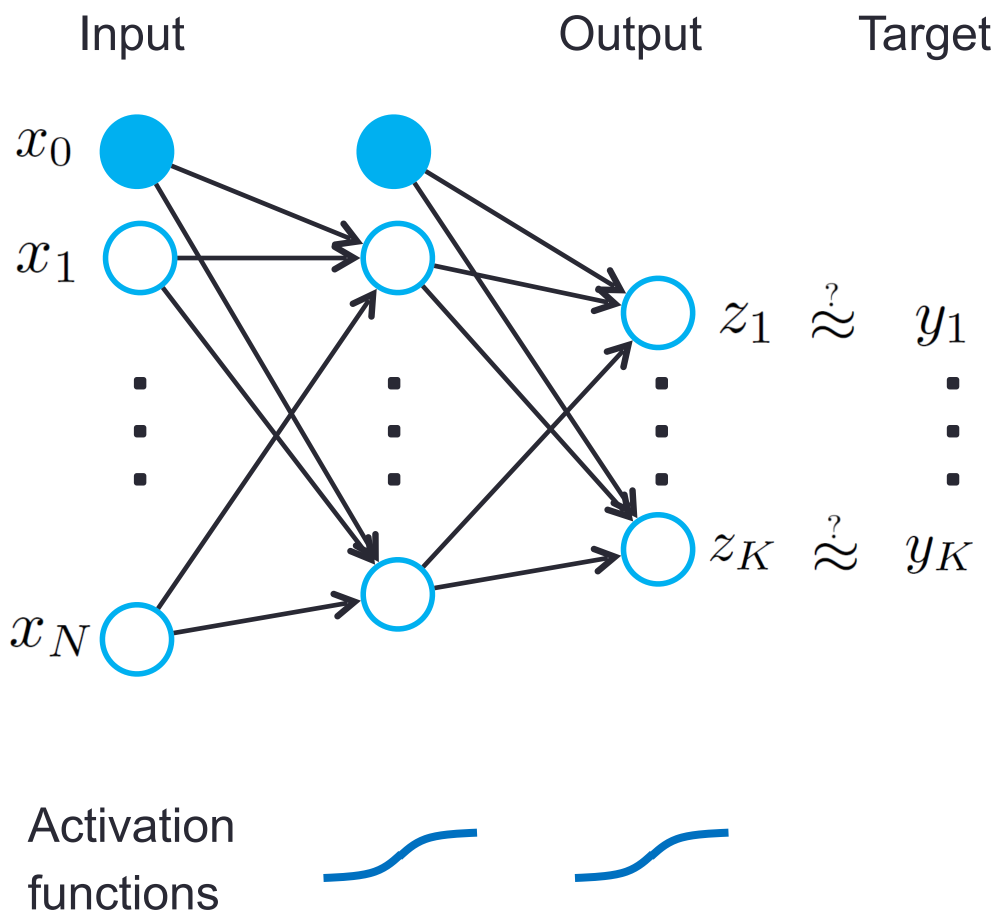
Regression



- Linear activation function in output layer ensures arbitrary output range

Activation functions

Classification



- Sigmoid activation function in output layer ensures outputs between 0 and 1

Classification: binary vs multiclass

- Binary classification:
  - Round the output of a single neuron (with sigmoidal activation) to 0 or 1 (threshold at 0.5) and interpret it as a class 0 or class 1

- Multiclass classification:
  - Multiple output neurons: use 1-out-of-n encoding for the target value $\boldsymbol{y}^{(i)}$ (one of $y_k^{(i)}$ values is 1, all others are 0)
  - This means there is one output neuron for each class
  - Use a softmax encoding to code the output as probabilities

$$softmax(z)_k = \frac{e^{z_k}}{\sum_{l=1}^{K} e^{z_l}}$$

For example if:

$\boldsymbol{y}^{(i)}$ is one of: $\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\1\\0\end{pmatrix}$ or $\begin{pmatrix}0\\0\\1\end{pmatrix}$

We would want the output of ANN to be:

$$\boldsymbol{z} \approx \begin{pmatrix}1\\0\\0\end{pmatrix}, \boldsymbol{z} \approx \begin{pmatrix}0\\1\\0\end{pmatrix}, \boldsymbol{z} \approx \begin{pmatrix}0\\0\\1\end{pmatrix}$$

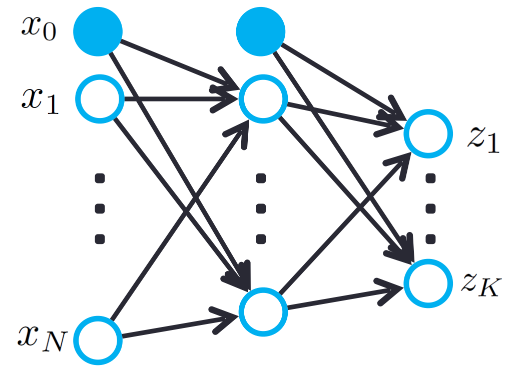
Different error functions for different tasks

- **Regression**
  - Quadratic loss:

$$E^{(i)} = \frac{1}{2}\sum_{k=1}^{K}(z_k^{(i)} - y_k^{(i)})^2$$

Training data:

$\langle \boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)} \rangle \ldots \langle \boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)} \rangle$



- **Binary Classification**
  - Binary cross-entropy:

$$E^{(i)} = -y^{(i)}\log(sigmoid(z^{(i)})) - (1 - y^{(i)})\log(1 - sigmoid(z^{(i)}))$$

- **Multi-class Classification**
  - Multi-class cross-entropy:

$$E^{(i)} = -\sum_{k=1}^{K} y_k^{(i)} \log(softmax(z^{(i)})_k)$$

Batch vs online learning

- **Batch learning**
  - The error gradient for each sample from training set is calculated and accumulated. The weight update is done after all samples are seen.

$$E = \sum_{i}^{m} E^{(i)} \qquad w_{kj} := w_{kj} - \eta \nabla E$$

- **Online learning**
  - After presentation of each sample i from the training set we use the calculated error gradient for weight update:

$$w_{kj} := w_{kj} - \eta \nabla E^{(i)}$$

  - It can be used when there is no fixed training set (new data keeps coming in)
  - The noise in the gradient can help to escape from **local minimum**
  - “Stochastic Gradient Descent”

Mini-batch learning

- Error gradient is calculated and accumulated over samples of a mini-batch from the training set. Weight update is done each time after a **mini-batch** of samples are seen.

$$w_{kj} := w_{kj} - \eta \sum_{i=1}^{N_B} \nabla E^{(i)}$$

- The algorithm is executed (usually in epochs during which a batch of samples from the training set is presented to the network) until a stopping criteria (e.g., error is smaller then some threshold) is satisfied.

- Results may converge to a local minimum.