

# Deep Learning: Neural Network Architecture & Training

**Robert Legenstein**

Institute of Machine Learning and Neural Computation

[robert.legenstein@tugraz.at](mailto:robert.legenstein@tugraz.at)

Deep Learning VO - WS 25/26

Lecture 3

# Today

---

❑ Neural Network Architecture

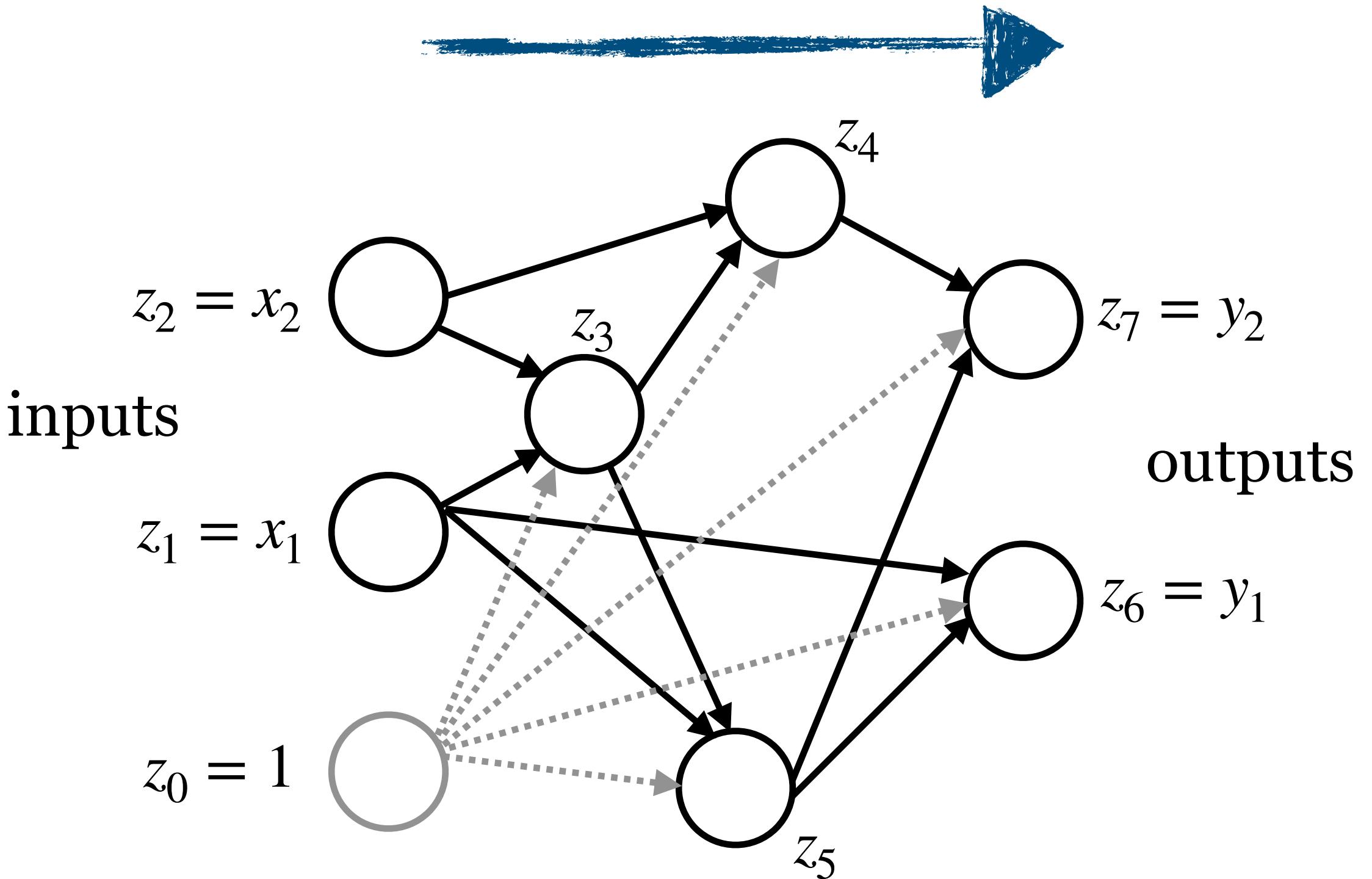
❑ Neural Network Training

    ❑ Error (Loss) Functions

    ❑ Gradient Descent

# General Network Description

- A **neural network** with  $D$  inputs and  $K$  outputs is defined by a directed graph  $G = (V, E)$  with:
  - nodes (neurons and inputs)  $V$
  - edges (connections)  $E$
- a weight  $w_{ji} \in \mathbb{R}$  for each edge  $(i, j) \in E$ ,
- an activation function  $h_j$  for each non-input node  $j$ ,
- a list of nodes  $OUT = \langle out_1, \dots, out_K \rangle$  defining  $K$  output nodes.



# Network function

- The output of node  $i$  is given by:

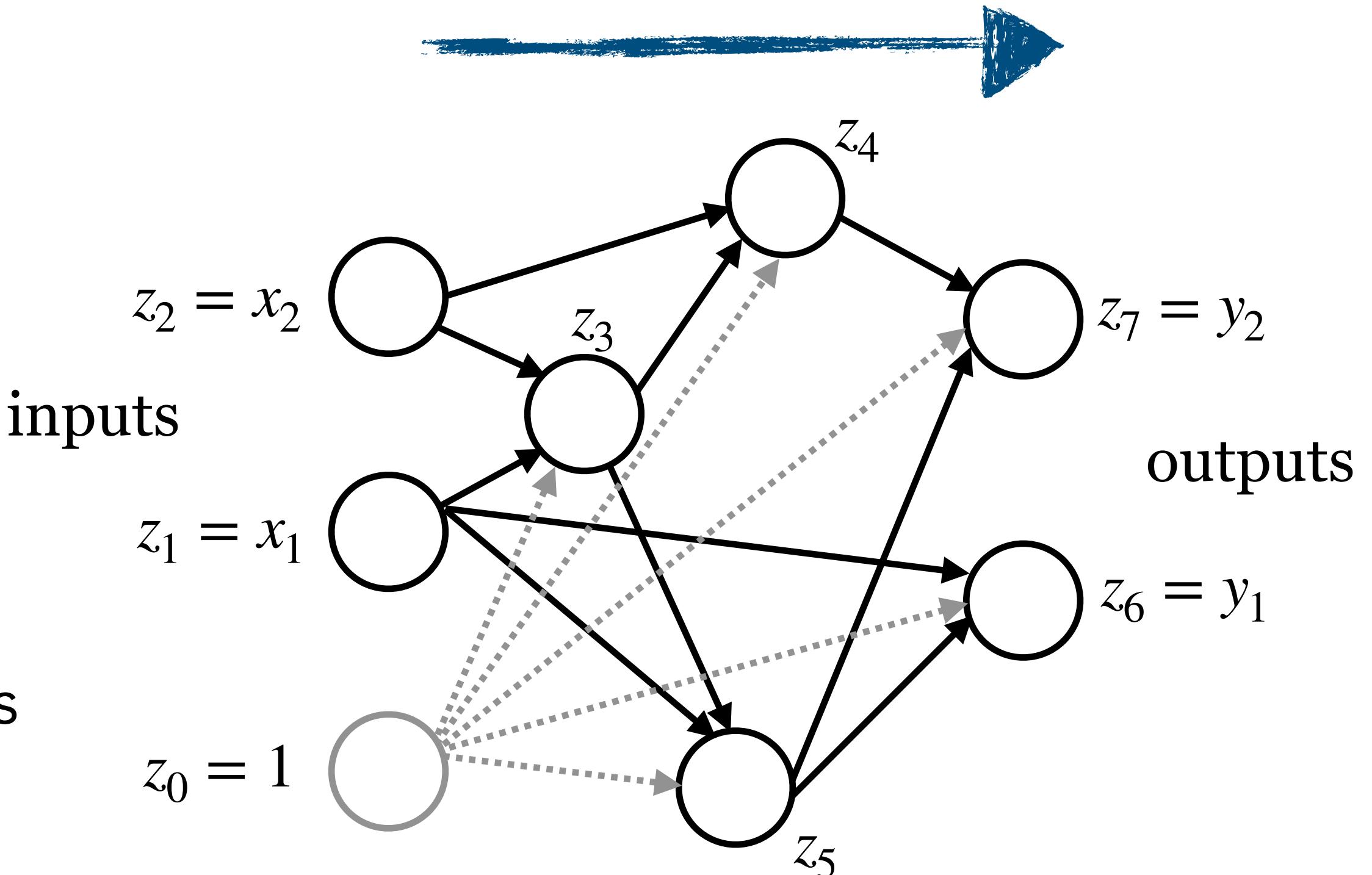
$$z_i = \begin{cases} x_i , & \text{if } i \in \{0, \dots, D\} \\ h_i(a_i) , & \text{otherwise} \end{cases}$$

with **activations**  $a_i = \sum_{j \in \text{pre}(i)} w_{ij} z_j$  and mappings:

**pre:**  $\text{pre}(i) = \{j \mid (j, i) \in E\}$  is the set of neurons which **connect to** neuron  $i$ .

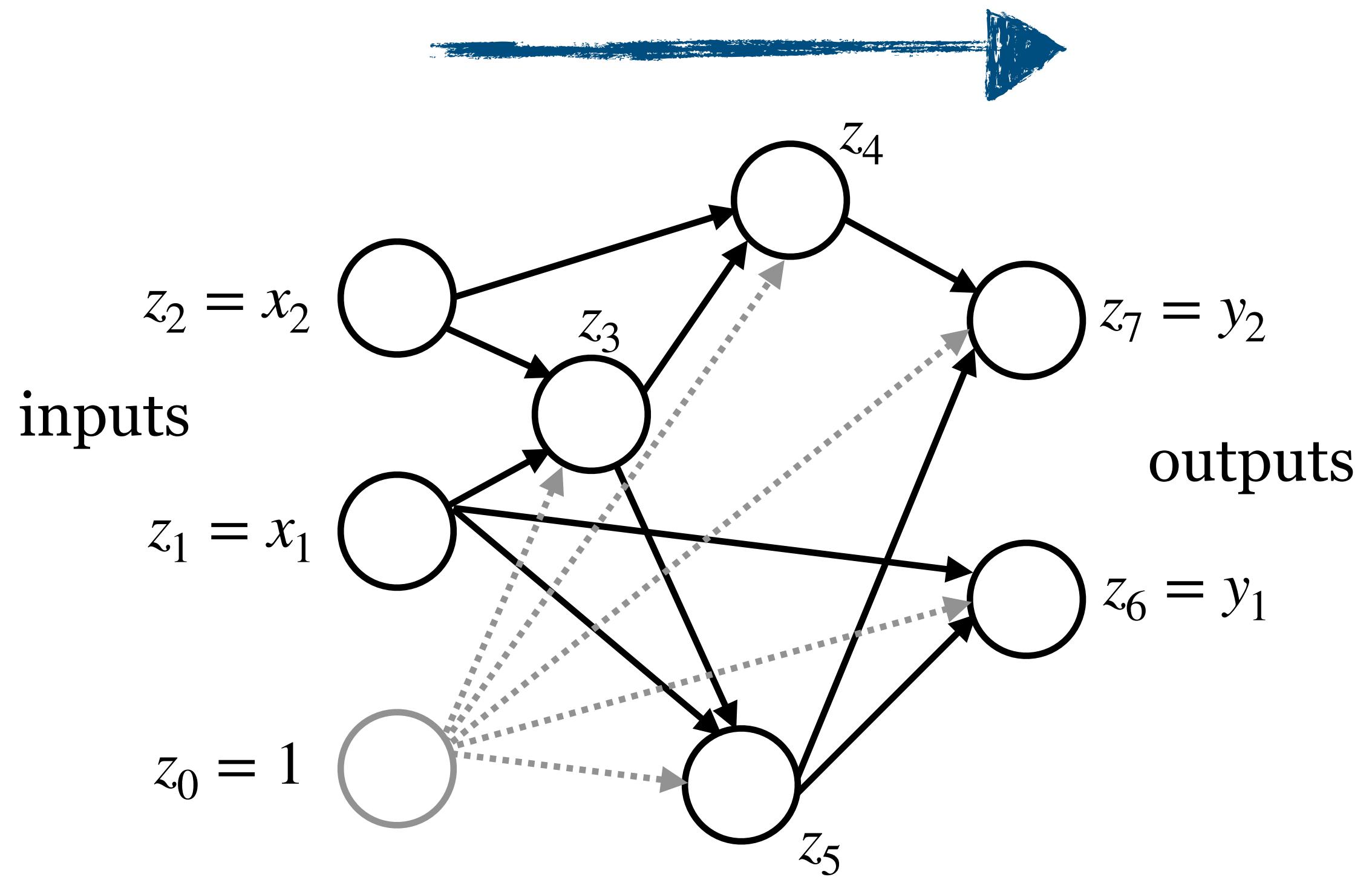
**post:**  $\text{post}(i) = \{j \mid (i, j) \in E\}$  is the set of neurons which neuron  $i$  **connects to**.

Then, the  $k$ th output of the network is given by  $y_k = z_{out_k}$ .



# Feedforward networks

- Any network can be described in this way.
- The network function  $y(\mathbf{x}, \mathbf{w})$  can be defined recursively.
- We consider here network graphs without loops, i.e., “**feedforward networks**”.
- Such networks are often called **Multilayer Perceptrons**.



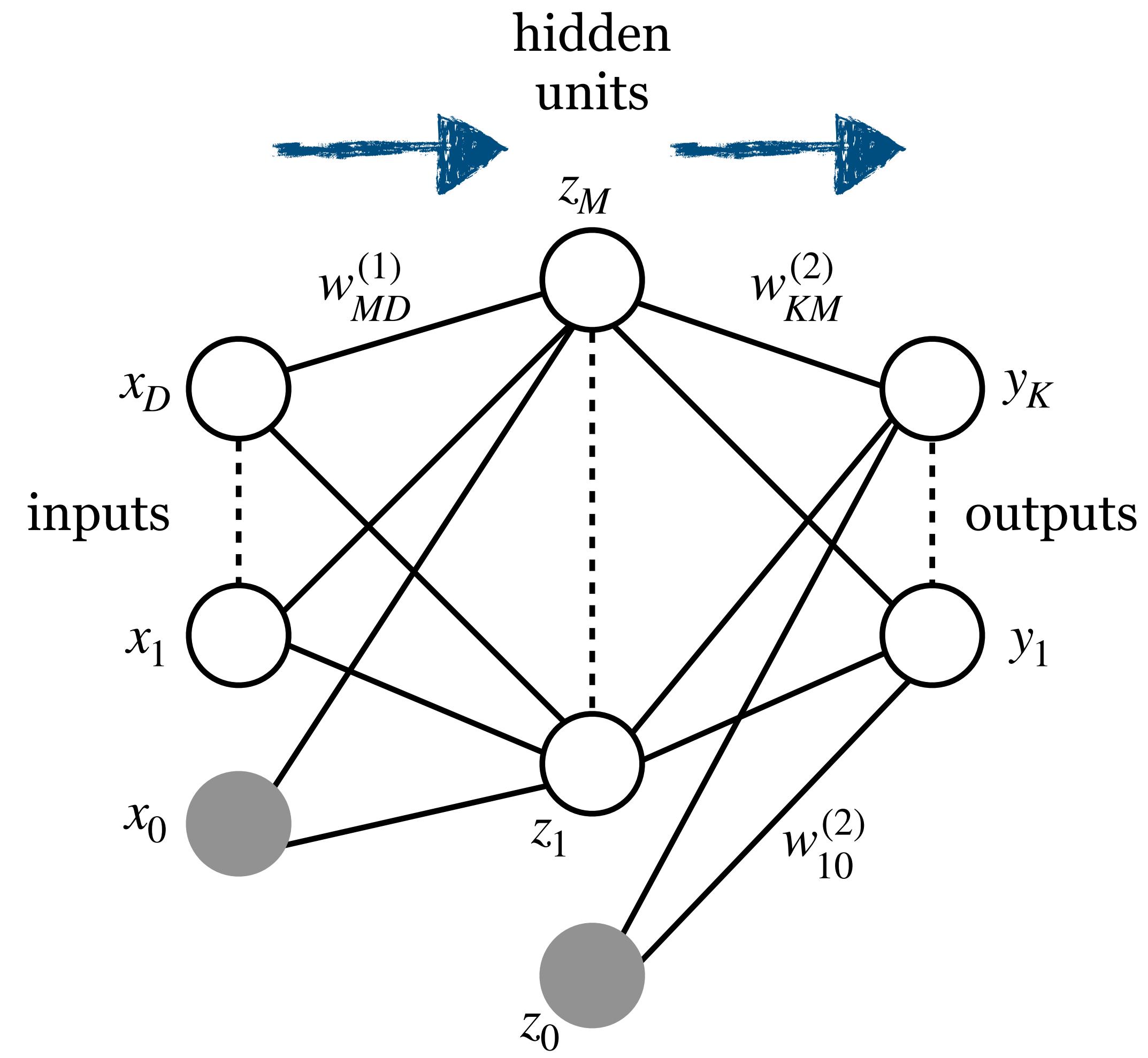
# Layered networks

- In a **layered network**, the nodes can be divided into layers, such that:
  - Input nodes are in layer 0 (input layer)
  - A node in layer  $i$  has outgoing edges only to nodes in layer  $i + 1$  for all  $i$ .

**Network depth:** The number of layers of a network excluding the input layer.

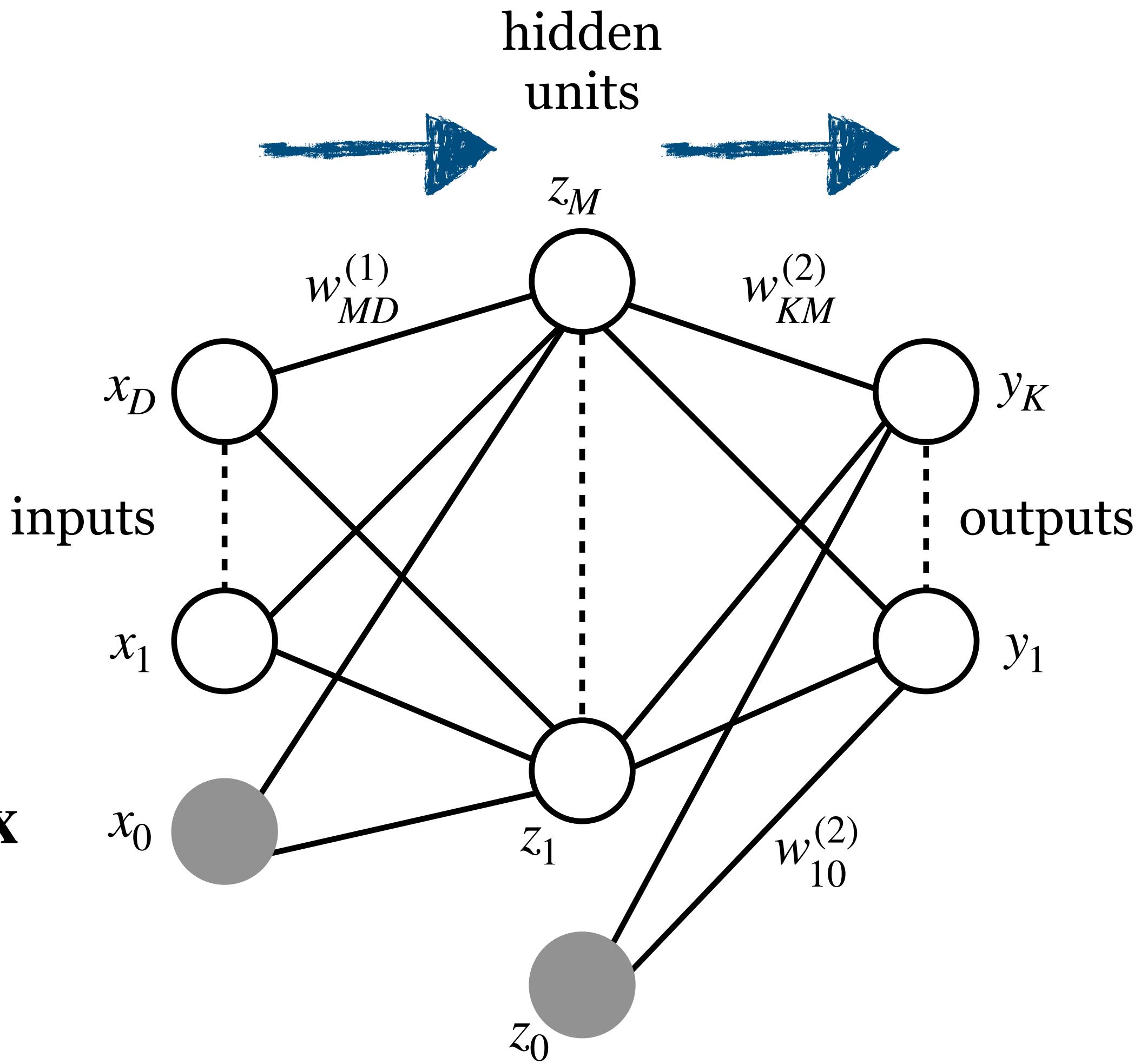
**Hidden units:** Neurons which are neither input nor output units.

**Hidden layers:** Layers containing hidden units (for layered network).



# Layered networks

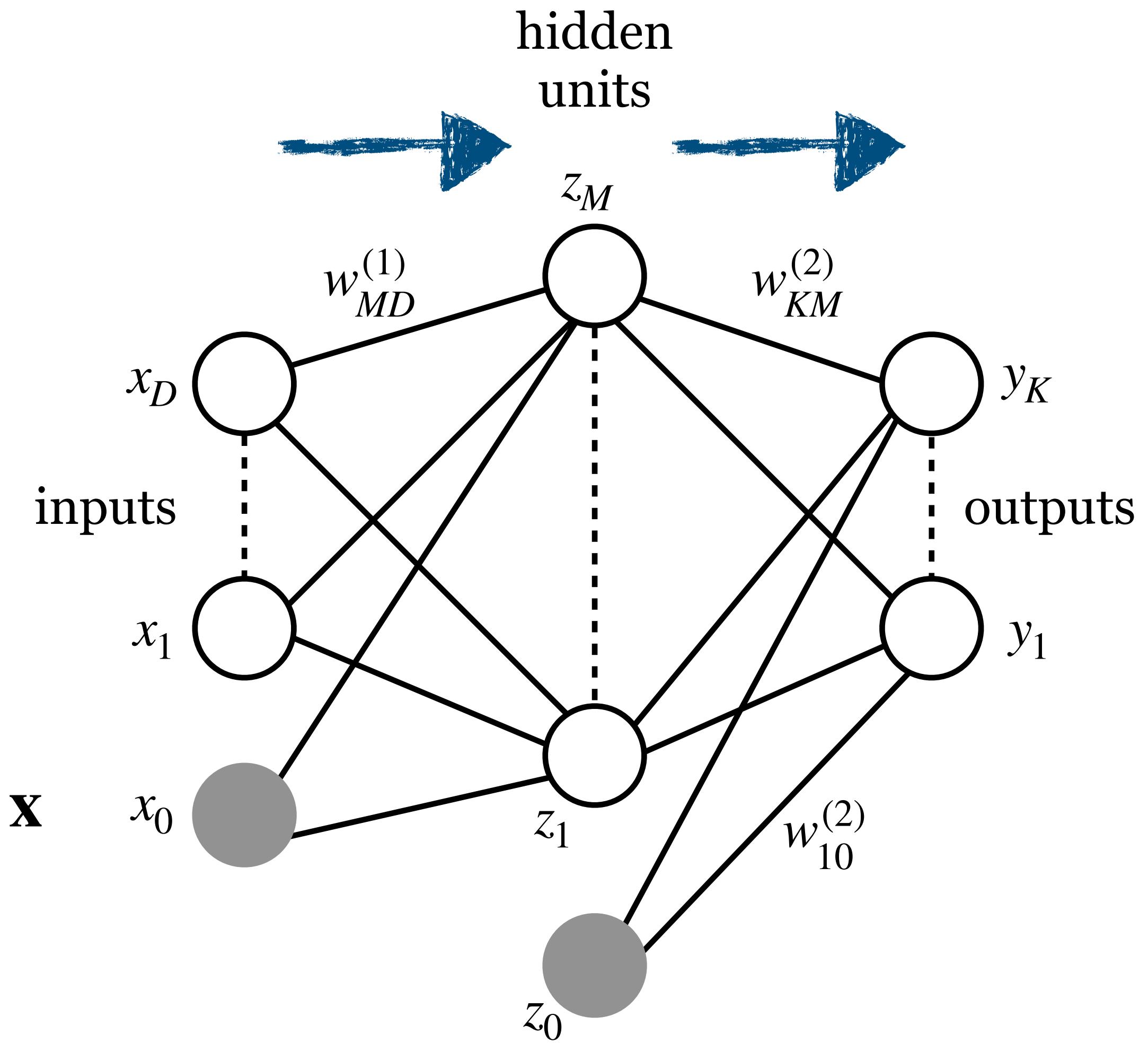
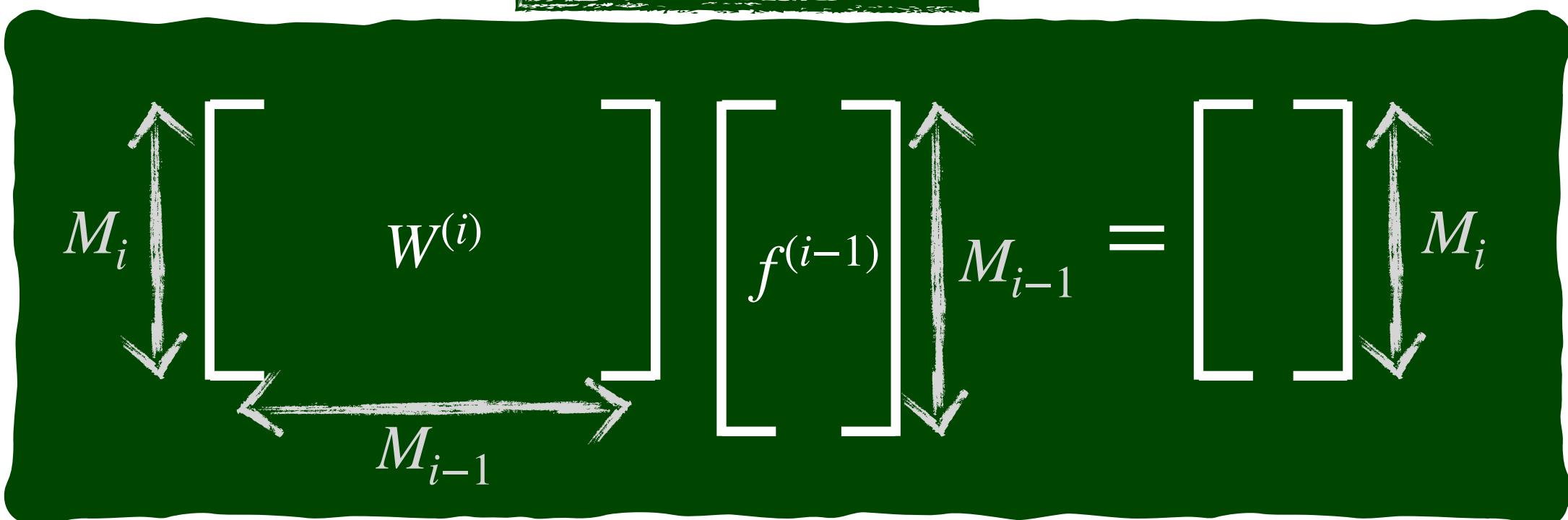
- Let  $M_i$  be the number of neurons in layer  $i$ .
  - $M_0 = D$  (inputs)
  - $M_d = K$  (outputs),  $d$  is the network depth.
- In a layered network, each layer  $i > 0$  computes a function  $f^{(i)} : \mathbb{R}^{M_{i-1}} \rightarrow \mathbb{R}^{M_i}$ .
- It applies the activation function  $h^{(i)}$  component-wise to an affine transformation of the previous layer output:
$$f^{(i)}(\mathbf{x}) = h^{(i)} \left( W^{(i)} f^{(i-1)}(\mathbf{x}) + \mathbf{b}^{(i)} \right) \quad \text{where } f^{(0)}(\mathbf{x}) = \mathbf{x}$$



# Layered networks

- Let  $M_i$  be the number of neurons in layer  $i$ .
  - $M_0 = D$  (inputs)
  - $M_d = K$  (outputs),  $d$  is the network depth.
- In a layered network, each layer  $i > 0$  computes a function  $f^{(i)} : \mathbb{R}^{M_{i-1}} \rightarrow \mathbb{R}^{M_i}$ .
- It applies the activation function  $h^{(i)}$  component-wise to an affine transformation of the previous layer output:

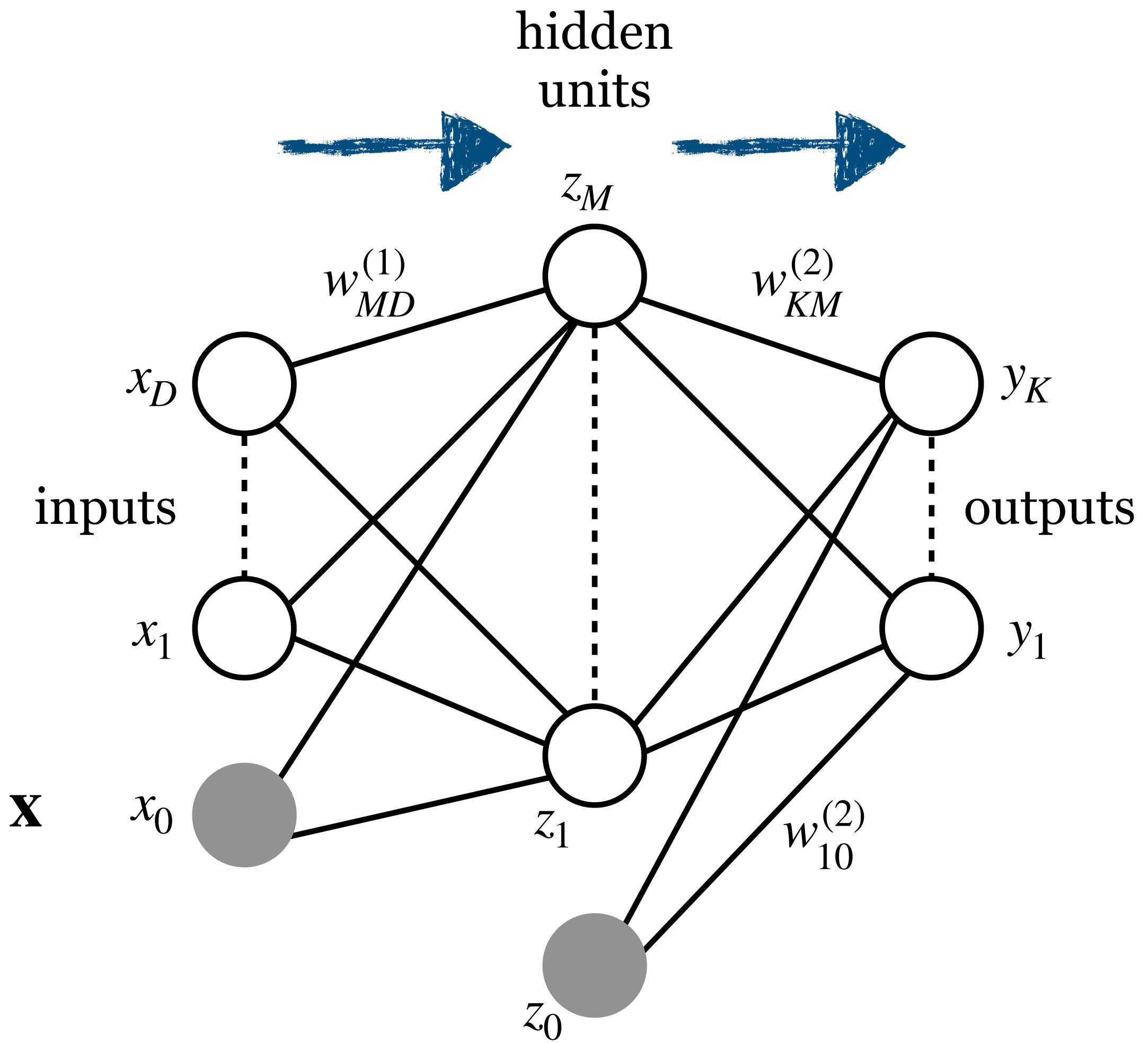
$$f^{(i)}(\mathbf{x}) = h^{(i)} \left( W^{(i)} f^{(i-1)}(\mathbf{x}) + \mathbf{b}^{(i)} \right) \quad \text{where } f^{(0)}(\mathbf{x}) = \mathbf{x}$$



# Layered networks

- Let  $M_i$  be the number of neurons in layer  $i$ .
  - $M_0 = D$  (inputs)
  - $M_d = K$  (outputs),  $d$  is the network depth.
- In a layered network, each layer  $i > 0$  computes a function  $f^{(i)} : \mathbb{R}^{M_{i-1}} \rightarrow \mathbb{R}^{M_i}$ .
- It applies the activation function  $h^{(i)}$  component-wise to an affine transformation of the previous layer output:

$$f^{(i)}(\mathbf{x}) = h^{(i)}(W^{(i)}f^{(i-1)}(\mathbf{x}) + \mathbf{b}^{(i)}) \quad \text{where } f^{(0)}(\mathbf{x}) = \mathbf{x}$$



**Example:** A network with three layers computes:

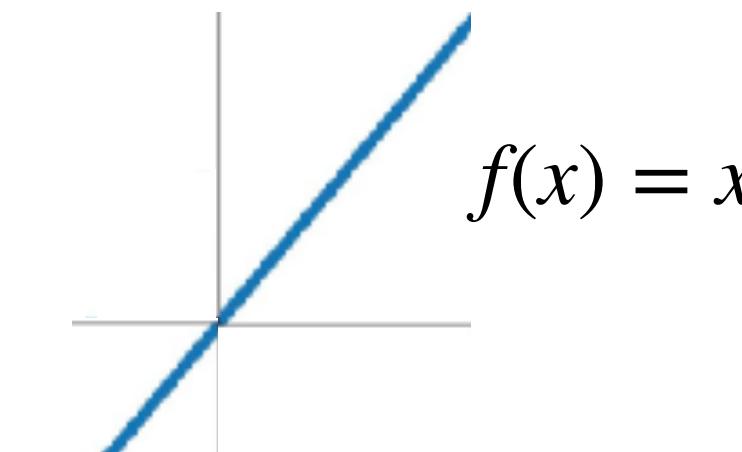
$$y(\mathbf{x}) = f^{(3)}(\mathbf{x}) = h^{(3)} \left( W^{(3)} h^{(2)} \left( W^{(2)} h^{(1)}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)} \right) + \mathbf{b}^{(3)} \right)$$

# Output Activation functions

The choice of **output activation functions** depends on the nature of the problem:

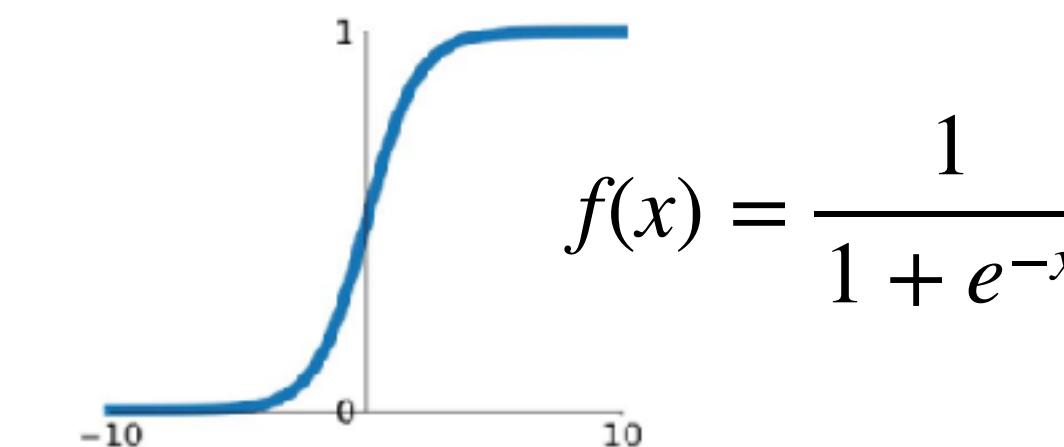
- **Linear** for regression problems.

$$h(a) = a$$



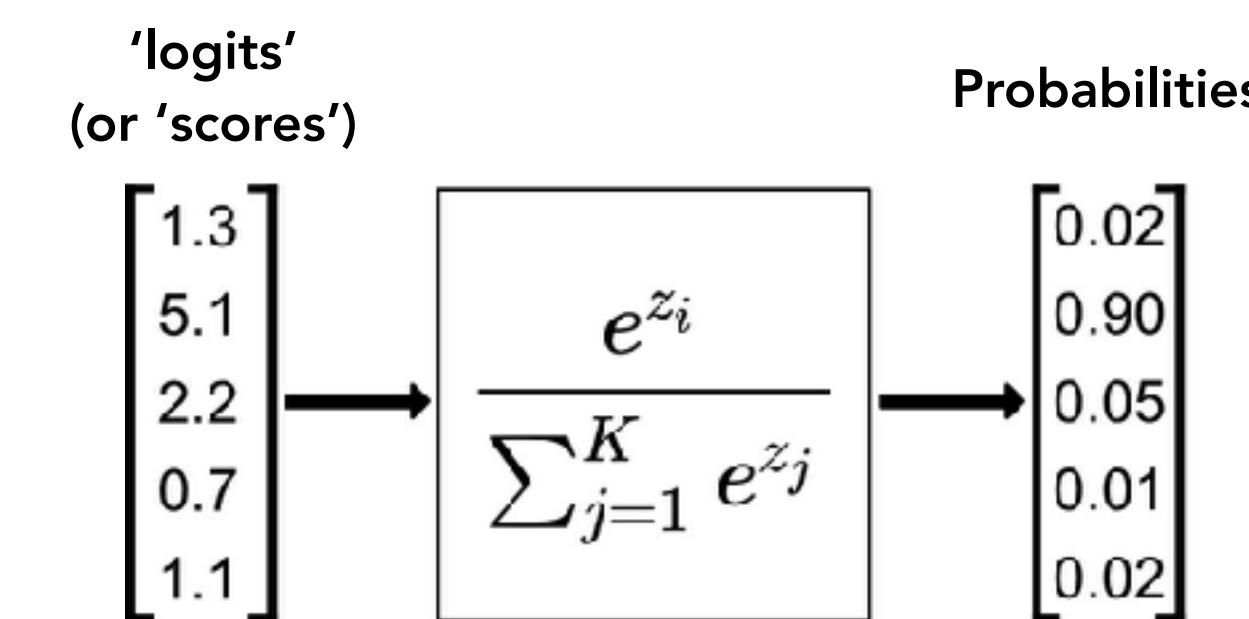
- **Sigmoid** for (possibly multiple) binary classification problems.

$$h(a) = \frac{1}{1 + \exp(-a)}$$



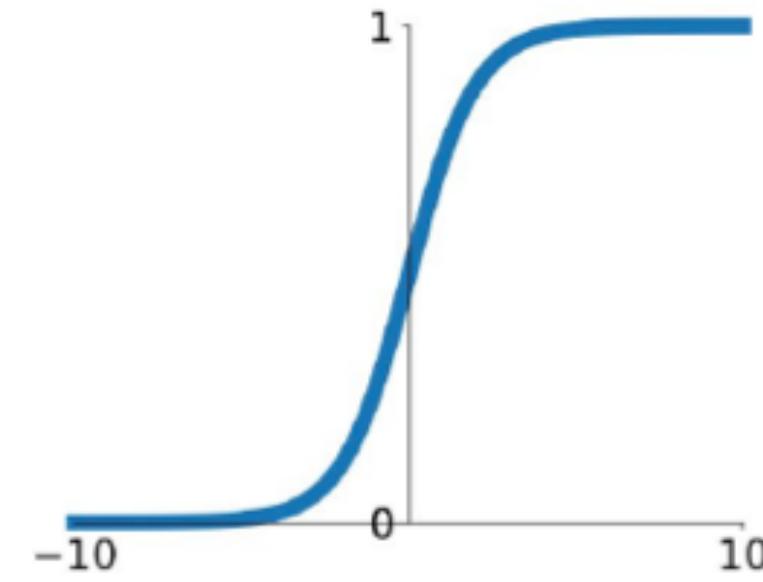
- **Softmax** for multiclass classification problems.

$$h_k(a_1, \dots, a_K) = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$



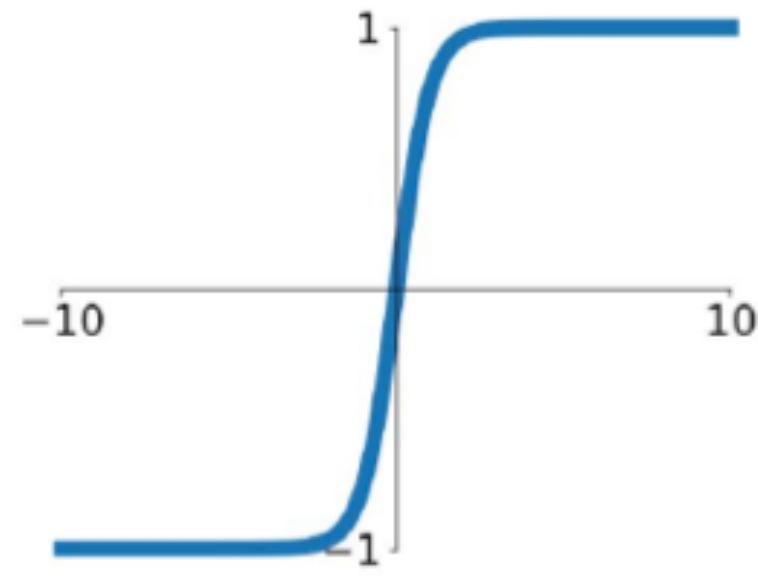
# Hidden Unit Activation functions

Sigmoid



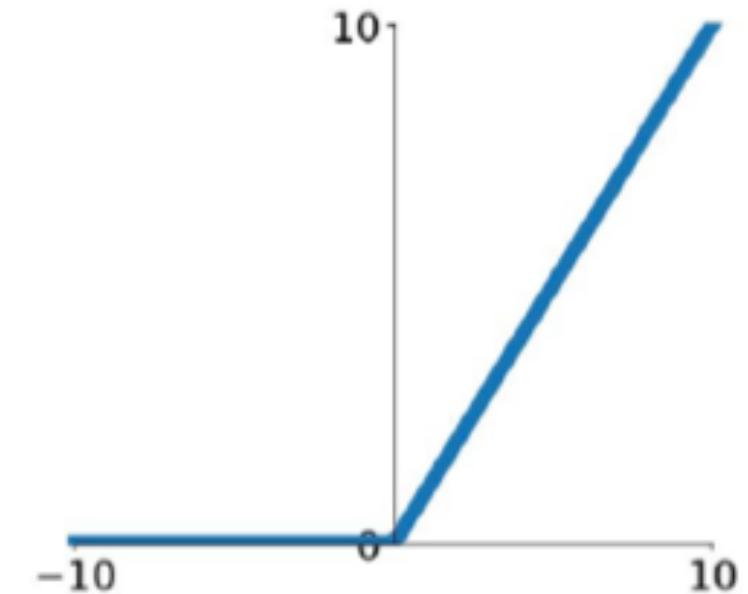
$$\sigma(x) = \frac{1}{1 + \exp(-x)}$$

Hyperbolic Tangent (tanh)



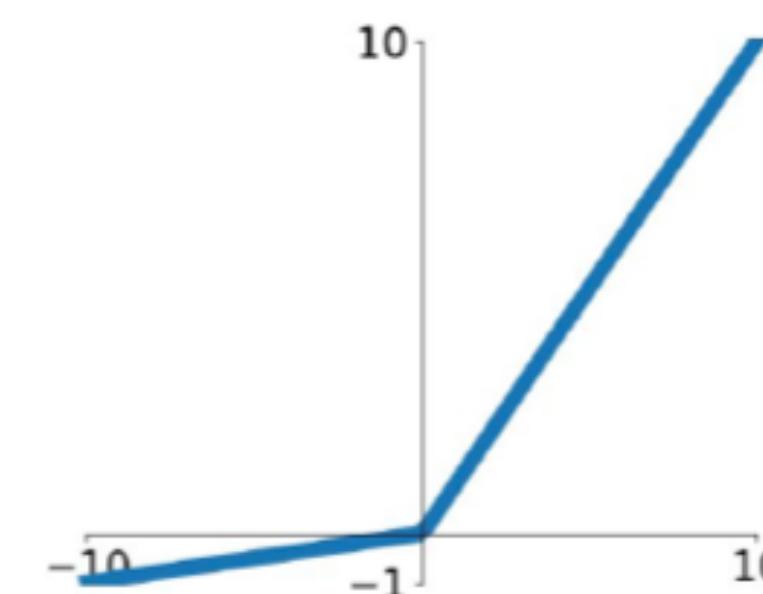
$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Rectified Linear Unit (ReLU)



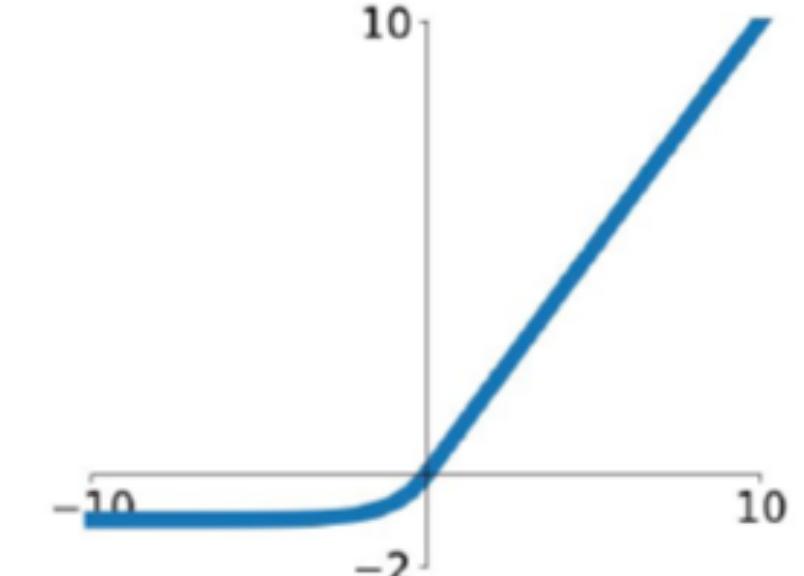
$$h(x) = \max\{0, x\}$$

Leaky ReLU



$$h(x) = \max\{\alpha x, x\}$$

Exponential Linear Unit (ELU)



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

# Hidden Unit Activation functions

- **Sigmoidal Units:** Traditional non-linear activations. Discouraged by recent research.

$$\text{Logsig: } \sigma(x) = \frac{1}{1 + e^{-x}} \quad \text{or} \quad \text{Hyperbolic tangent (tanh): } h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Note that:

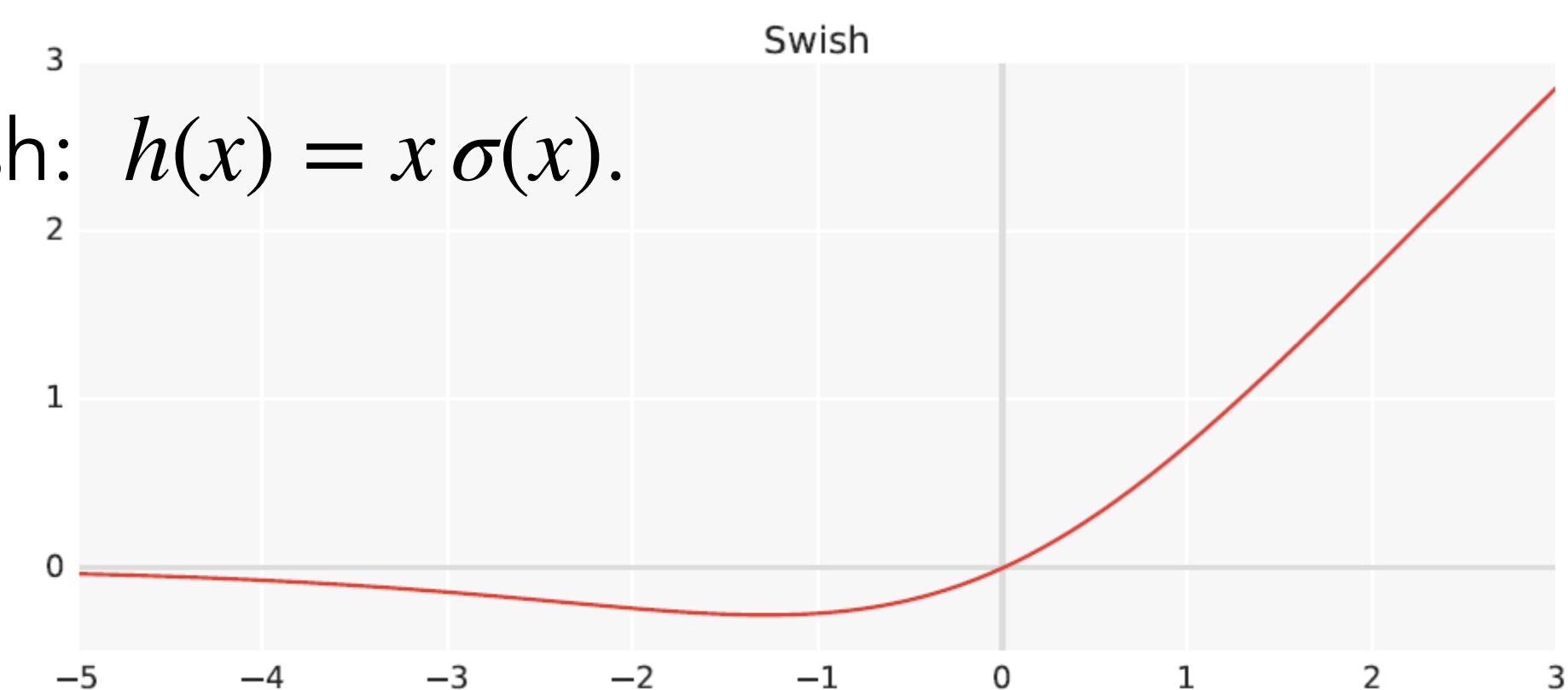
$$\tanh(x) = 2\sigma(2x) - 1$$

- **Rectified Linear Unit (ReLU):** Excellent default choice.  $\rightarrow h(x) = \max\{0, x\}$

- **Leaky ReLU:**  $h(x) = \max\{\alpha x, x\}$  for small  $\alpha$  (e.g.,  $\alpha = 0.1$ ).

- **Exponential Linear Unit (ELU):**  $h(x) = \begin{cases} x, & x \geq 0 \\ \alpha(e^x - 1), & x < 0 \end{cases}$

- **Others:** Maxout, Hard tanh:  $h(x) = \max\{-1, \min\{1, x\}\}$ , Swish:  $h(x) = x \sigma(x)$ .



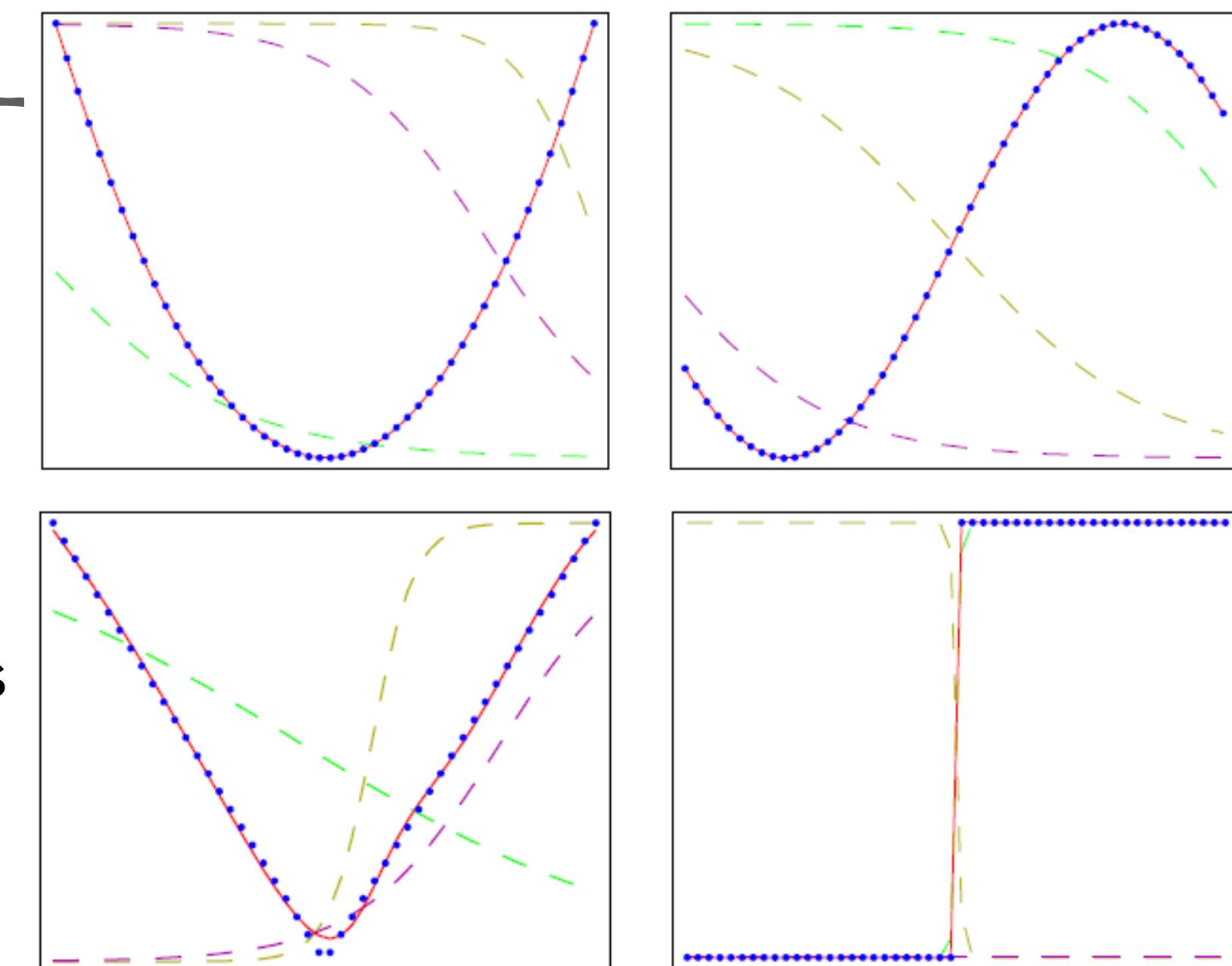
# Approximation Properties

- Neural networks define a very general class of functions.
- **Neural networks are universal approximators:**

A two-layer neural network with linear output unit and sigmoidal hidden units can approximate any continuous function on a compact input domain to arbitrary accuracy provided a sufficiently large number of hidden units.

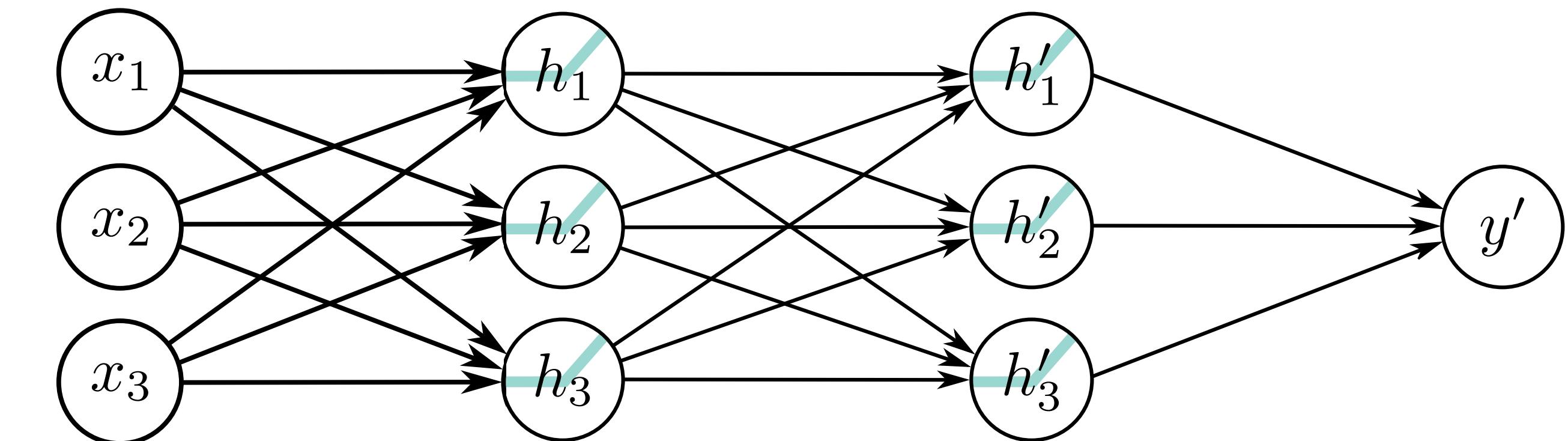
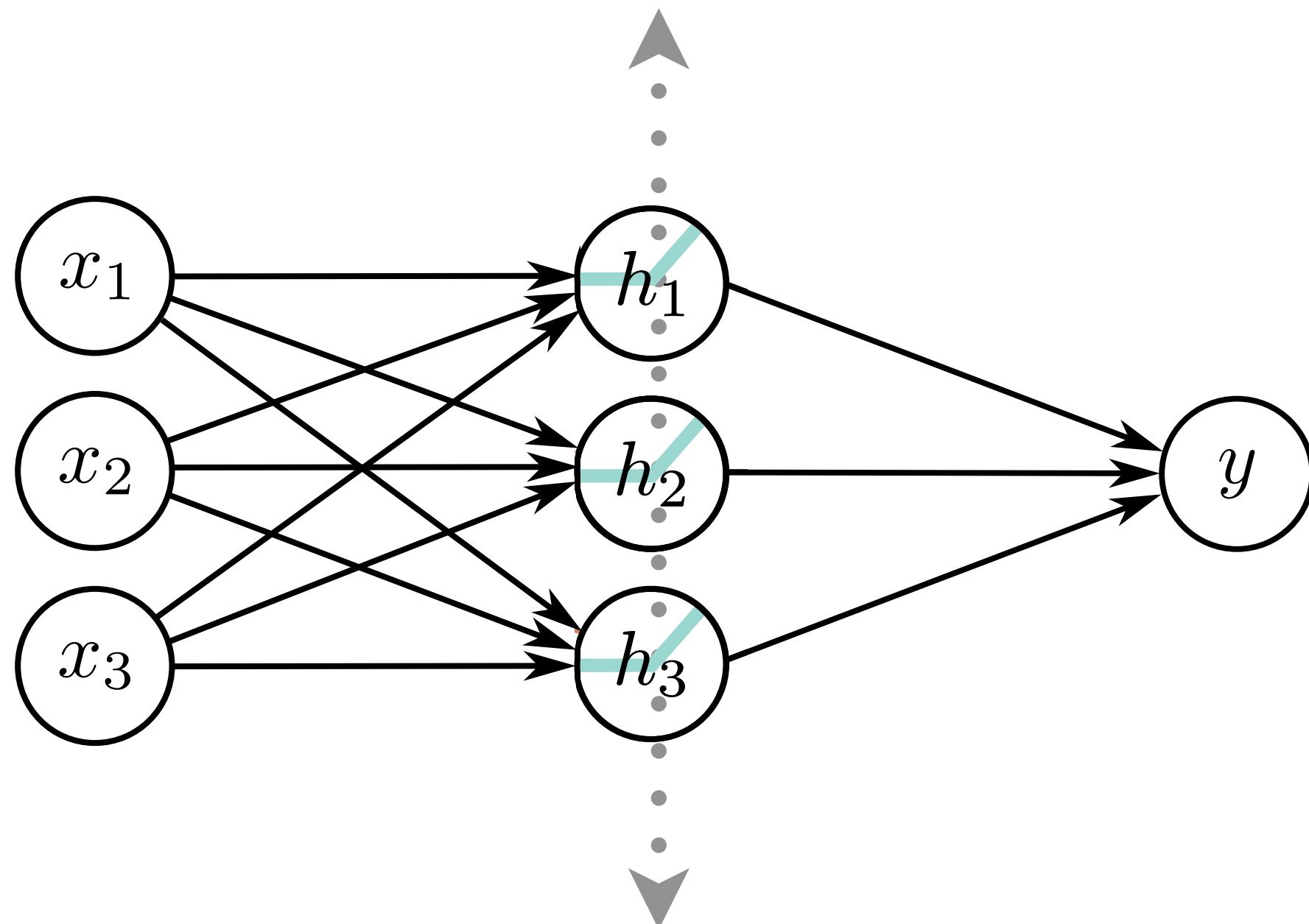
Network with 3 hidden neurons:

**blue** target function  
**red** approximation  
— — — output of hidden neurons



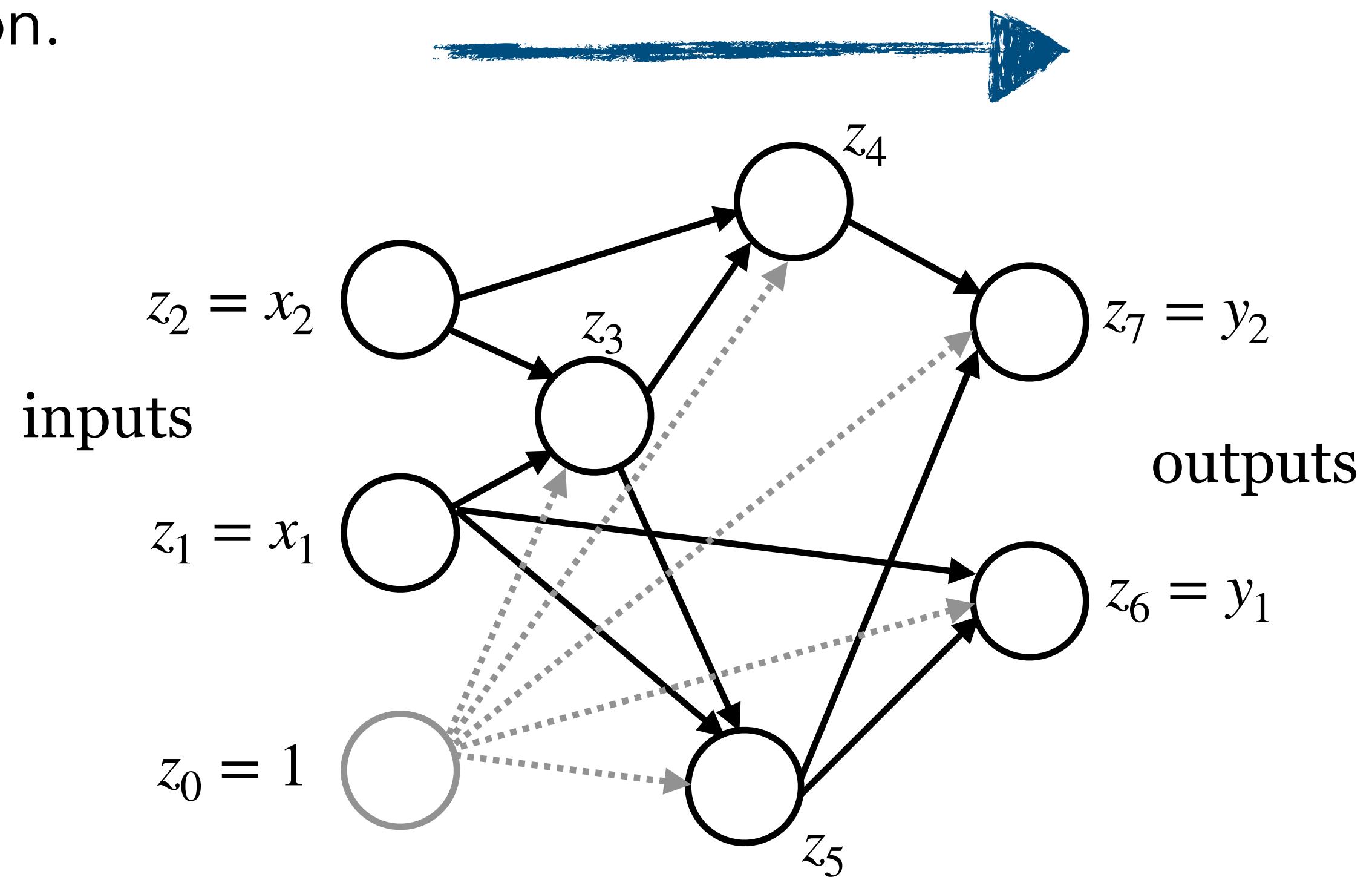
# Why deep neural networks are preferable?

- A two-layer neural network can approximate any function.
  - However, the number of neurons necessary may be extremely large.
- In many cases, deeper networks can reduce the number of units and improve generalization.
- Using a deep model encodes a general belief that the functions we want to learn involve composition of several simpler functions (multi-step programs).



# Summary: Neural Network Architectures

- Neural network architectures can be defined as network graphs.
- For layered networks, we can simplify the definition.
- Many activation functions are possible.
- Neural networks are universal approximators.
- Deep neural networks are preferred.



# Today

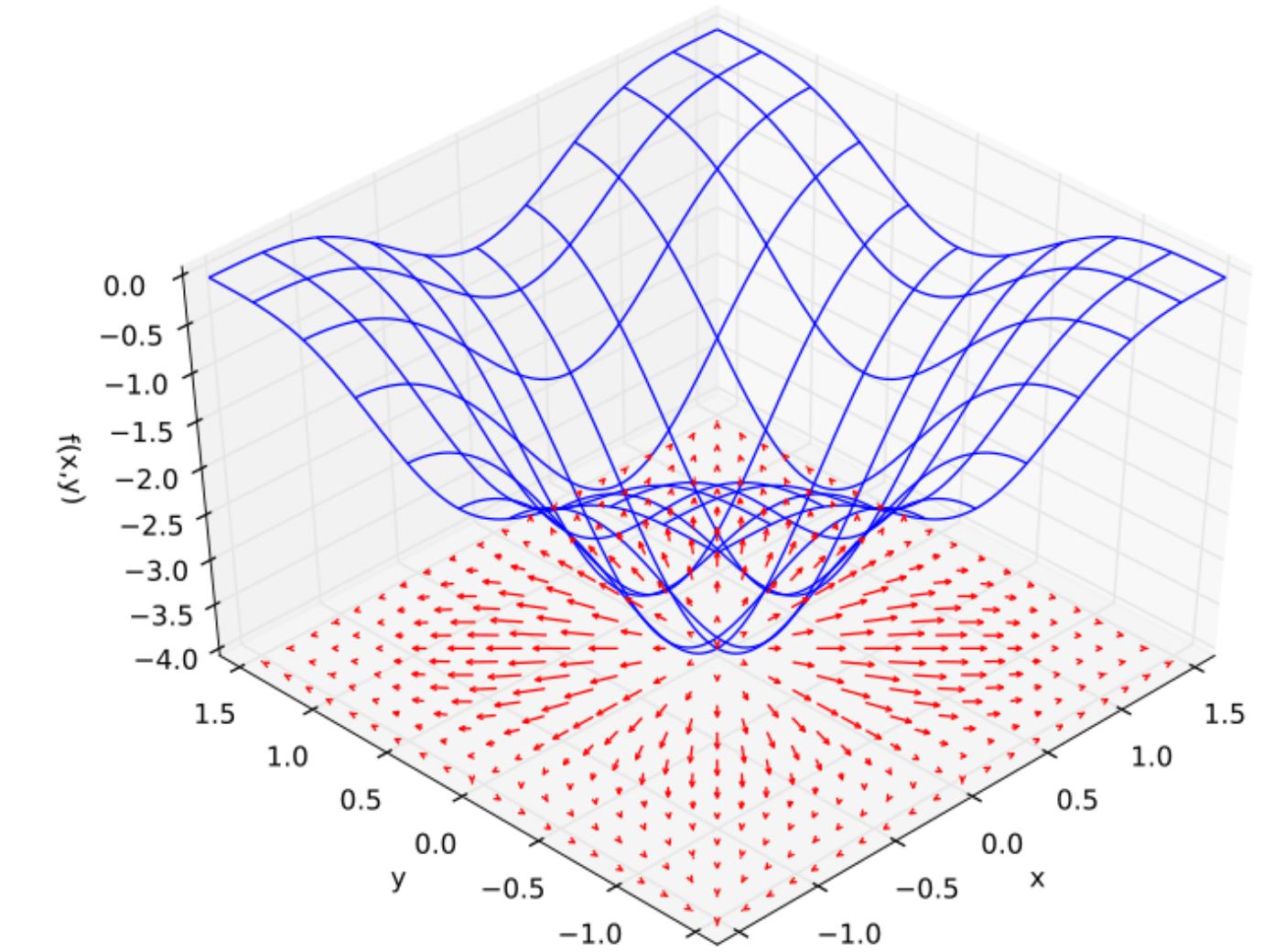
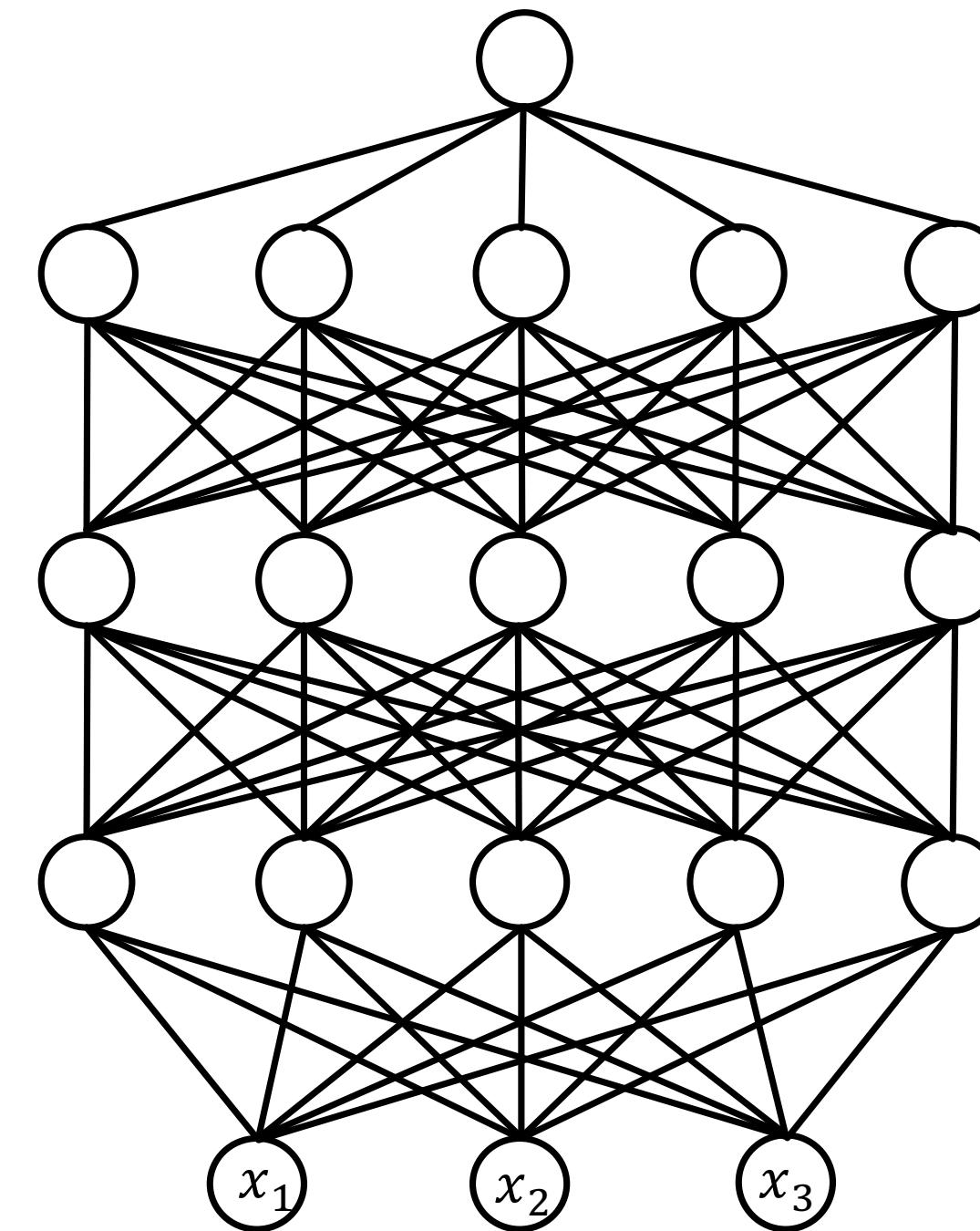
---

Neural Network Architecture

Neural Network Training

Error (Loss) Functions

Gradient Descent

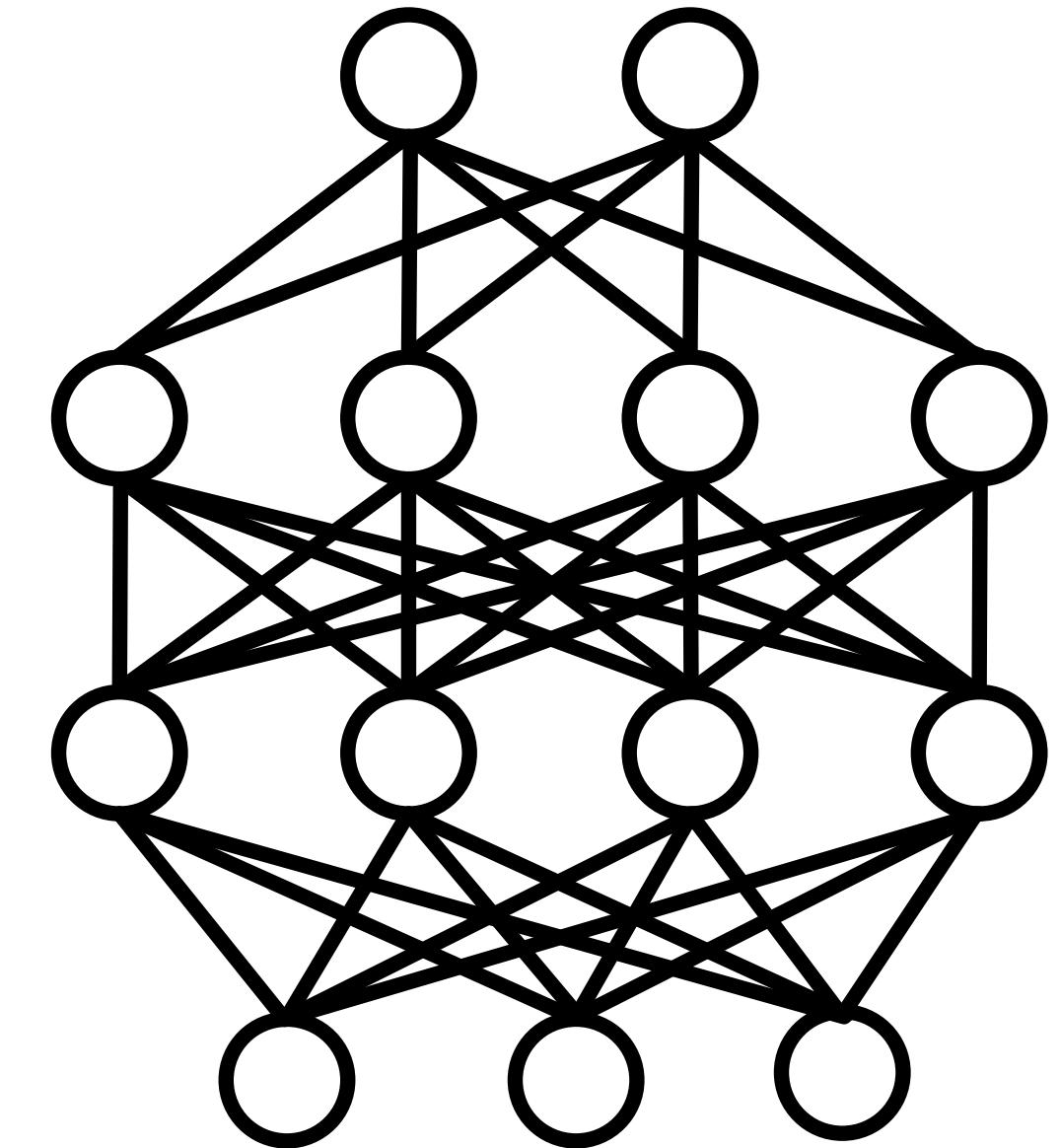


# Error (Loss) functions

---

**For training we try to minimize a loss function.**

- Consider a neural network with  $D$  inputs and  $K$  outputs.
- We have  $M$  training examples.
  - Network inputs:  $X = \langle \mathbf{x}^{(1)}, \dots, \mathbf{x}^{(M)} \rangle$
  - Target vectors:  $T = \langle \mathbf{t}^{(1)}, \dots, \mathbf{t}^{(M)} \rangle$ .
- The loss functions for different types of problems can be derived from maximum likelihood and assumptions on the data distribution.
- Corresponding output activation functions arise naturally such that the network output can be interpreted as the (mean of the) posterior distribution.

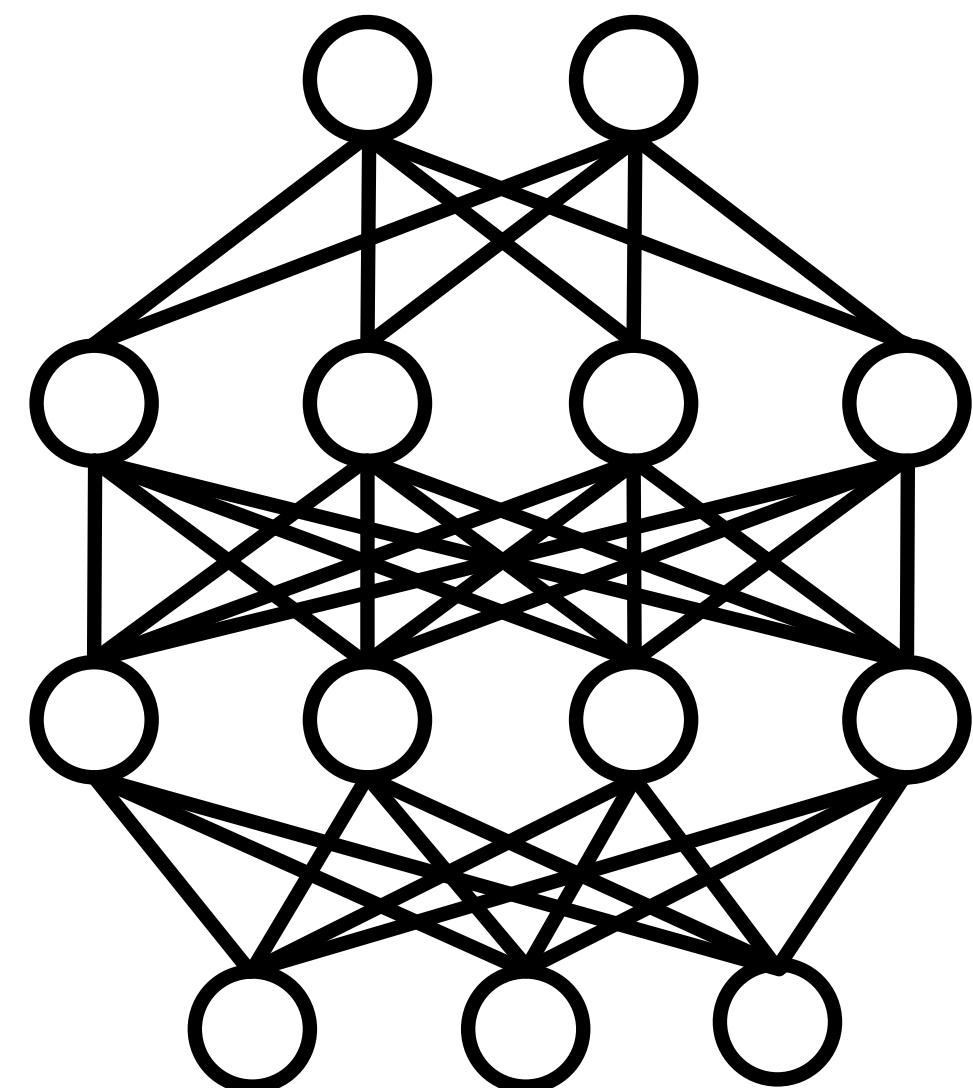
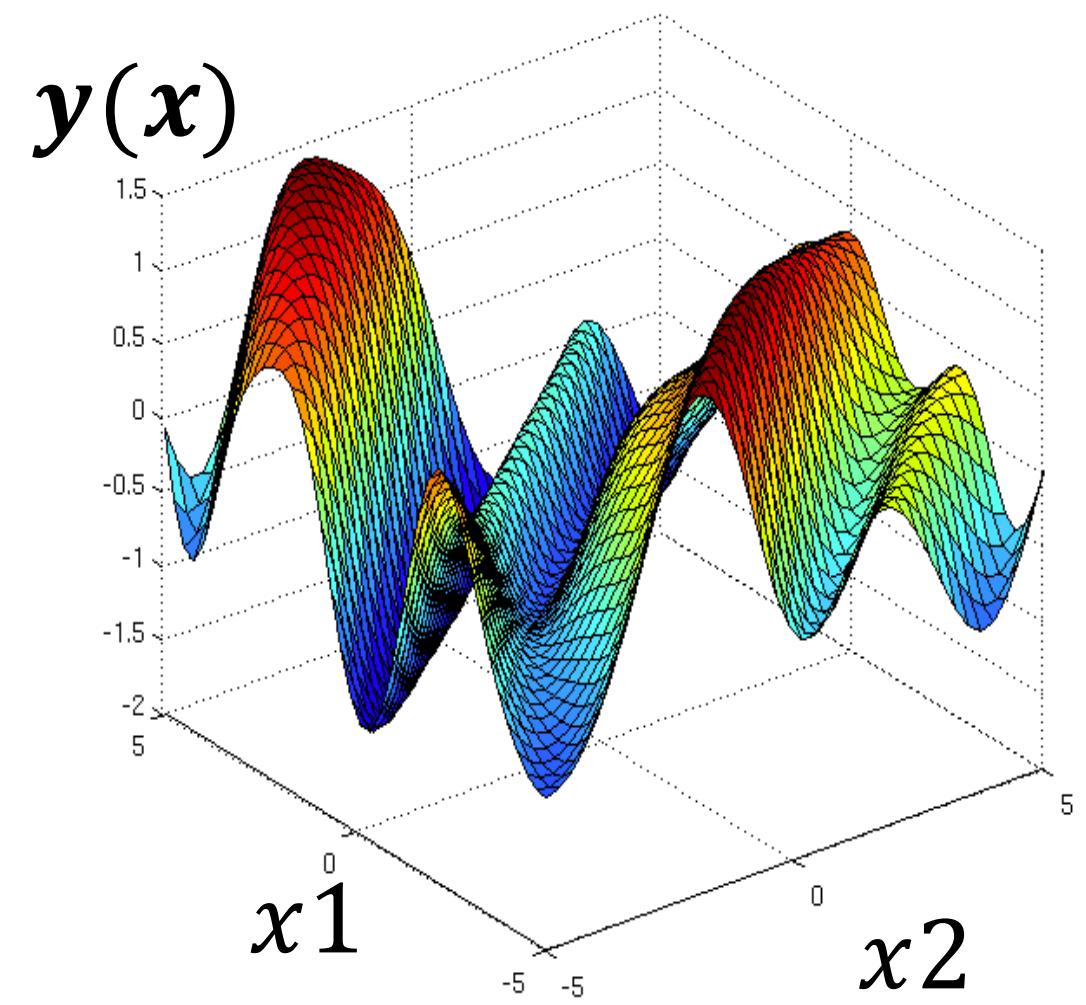


# Error (Loss) functions

## K-dimensional regression

- **Activation function:** identity
- **Data-sample Likelihood:**  $N(\mathbf{t}^{(m)} | \mathbf{y}(\mathbf{x}^{(m)}), \sigma^2 \mathbf{I})$
- **Error function:** Sum squared error

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2} \sum_m \|\mathbf{y}(\mathbf{x}^{(m)}) - \mathbf{t}^{(m)}\|^2 \\ &= \frac{1}{2} \sum_m \sum_k (y_k(\mathbf{x}^{(m)}) - t_k^{(m)})^2 \end{aligned}$$

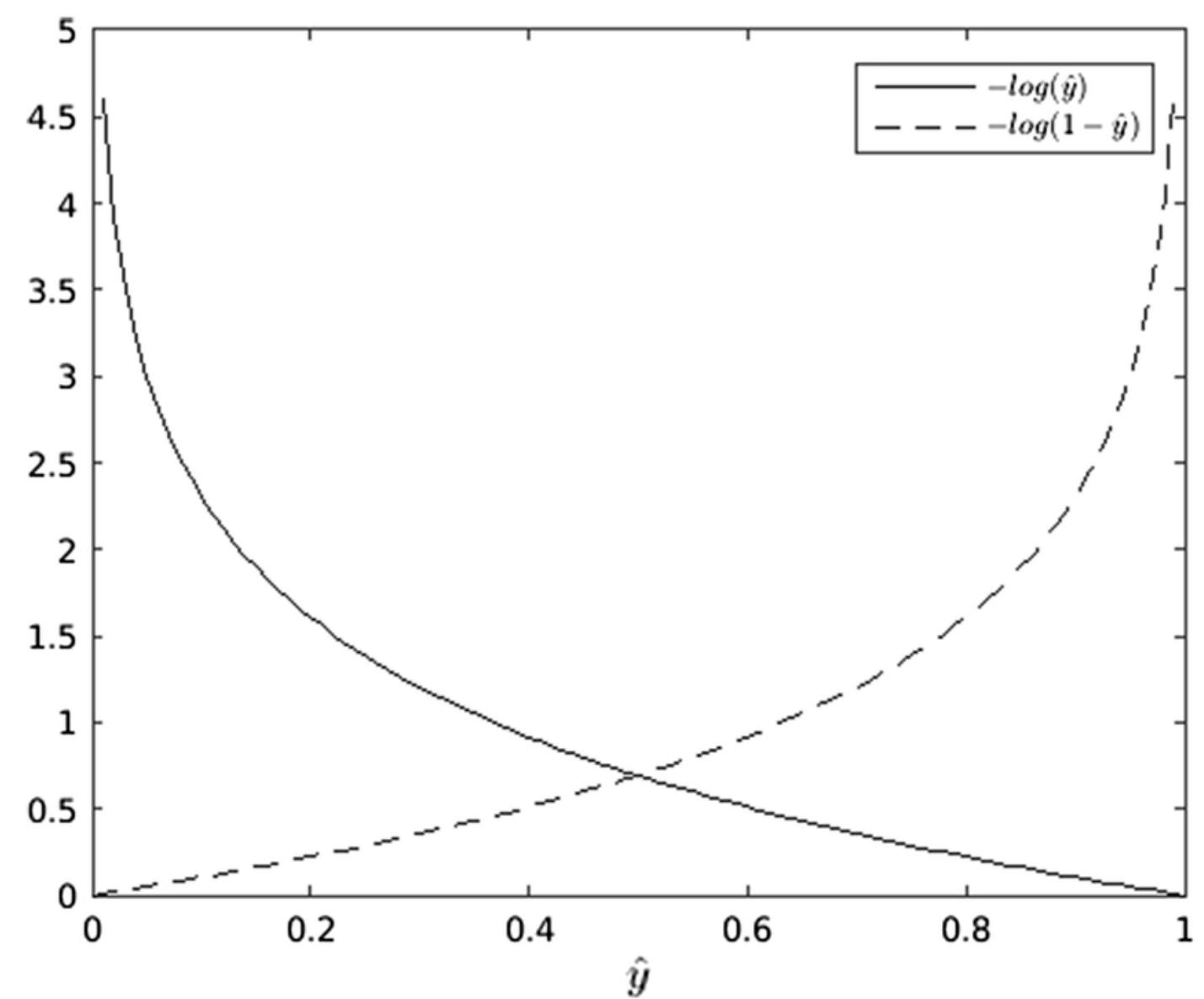
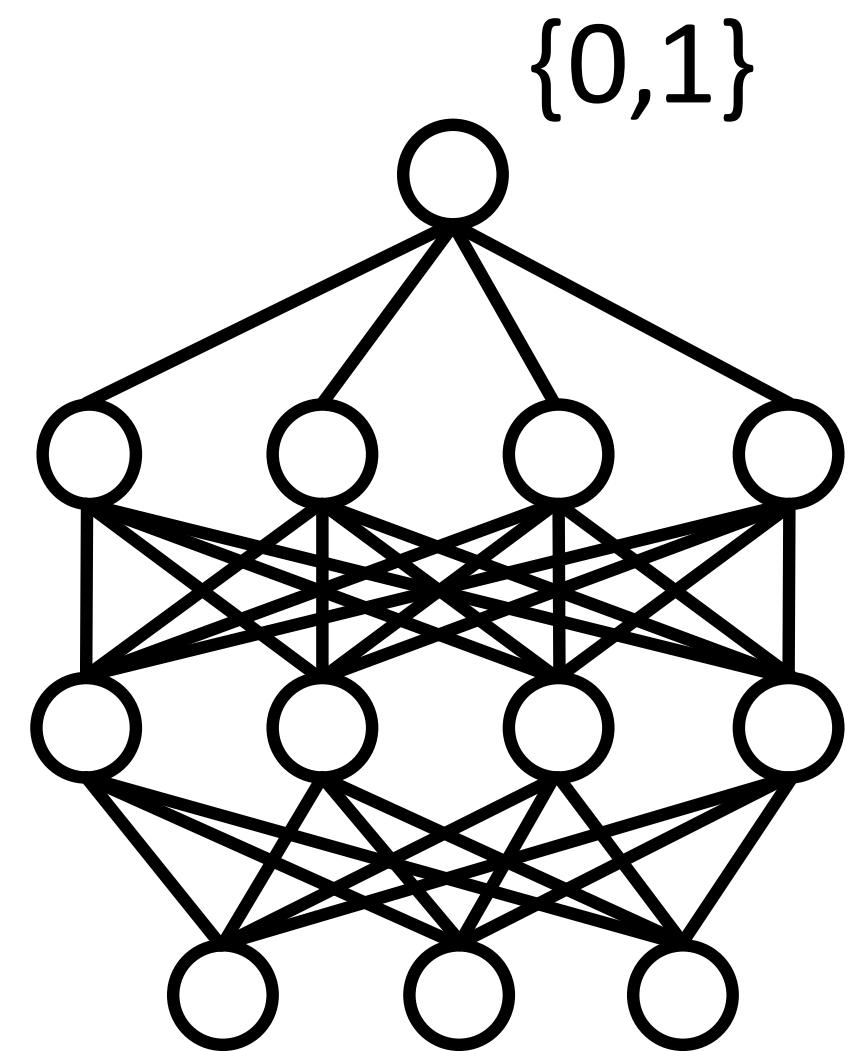


# Error (Loss) functions

## Binary classification

- **Activation function:** logistic sigmoid
- **Data-sample Likelihood:**  $y(\mathbf{x}^{(m)})^{t^{(m)}}(1 - y(\mathbf{x}^{(m)}))^{(1-t^{(m)})}$
- **Error function:** Cross-entropy error

$$E(\mathbf{w}) = - \sum_m \left[ t^{(m)} \log y(\mathbf{x}^{(m)}) + (1 - t^{(m)}) \log (1 - y(\mathbf{x}^{(m)})) \right]$$



# Error (Loss) functions

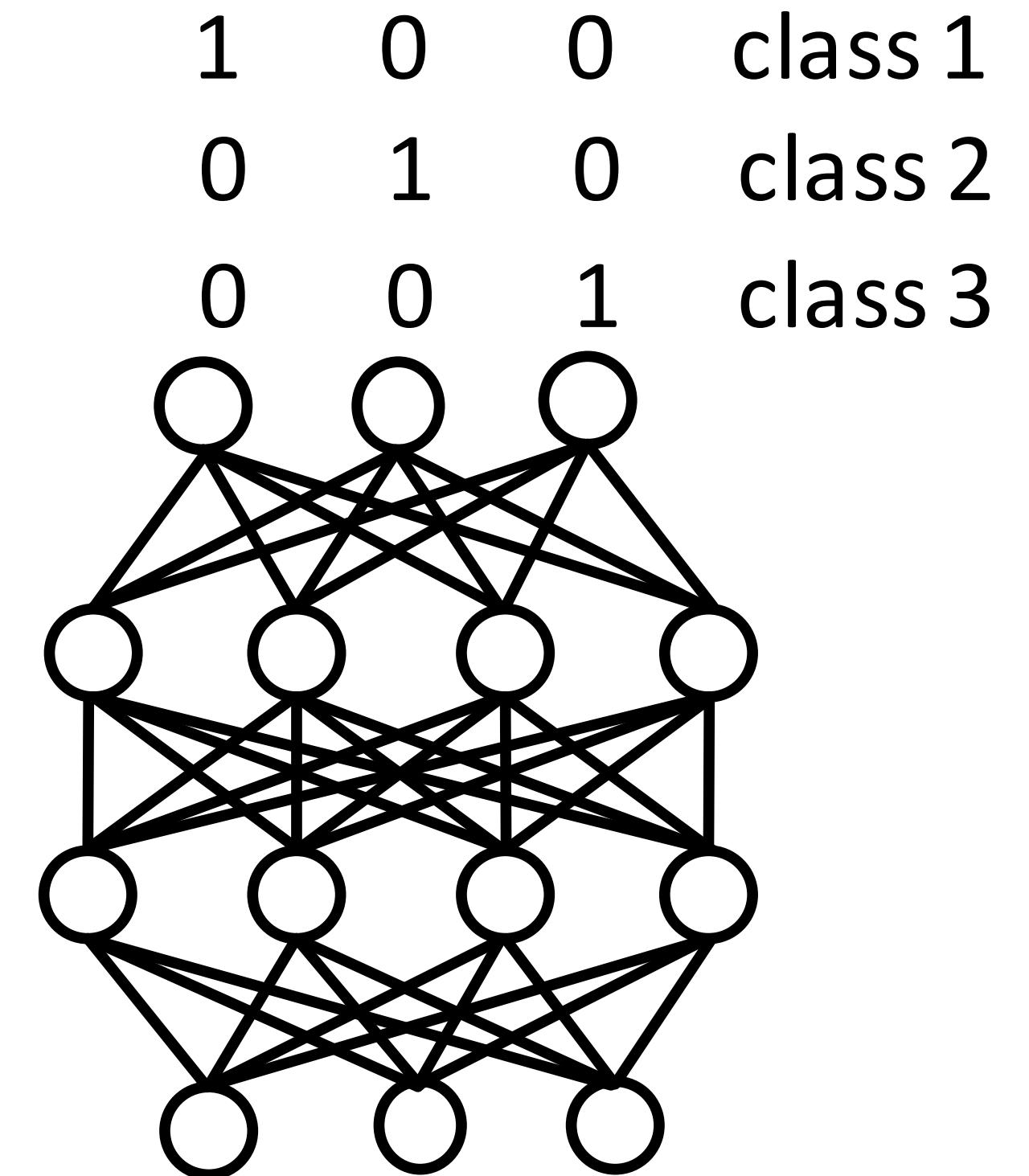
## K-class classification

- **Activation function:** softmax
- **Data-sample Likelihood:**  $\prod_k y_k(\mathbf{x}^{(m)})^{t_k^{(m)}}$
- **Error function:** Cross-entropy error

$$E(\mathbf{w}) = - \sum_m \sum_k \left[ t_k^{(m)} \log y_k(\mathbf{x}^{(m)}) \right]$$

Target vectors are one-hot encodings of the target class:

$$t_k^{(m)} = \begin{cases} 1, & \text{if target class is } k, \\ 0, & \text{otherwise} \end{cases}$$



# Today

---

Neural Network Architecture

Neural Network Training

Error (Loss) Functions

Gradient Descent

# The Gradient

- The error (a.k.a. loss) is a **scalar field**:

$$E : \mathbb{R}^D \rightarrow \mathbb{R}$$

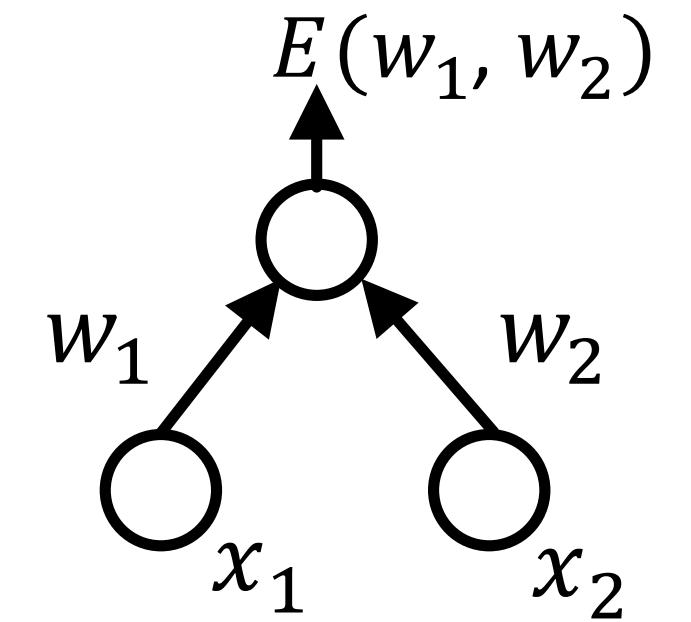
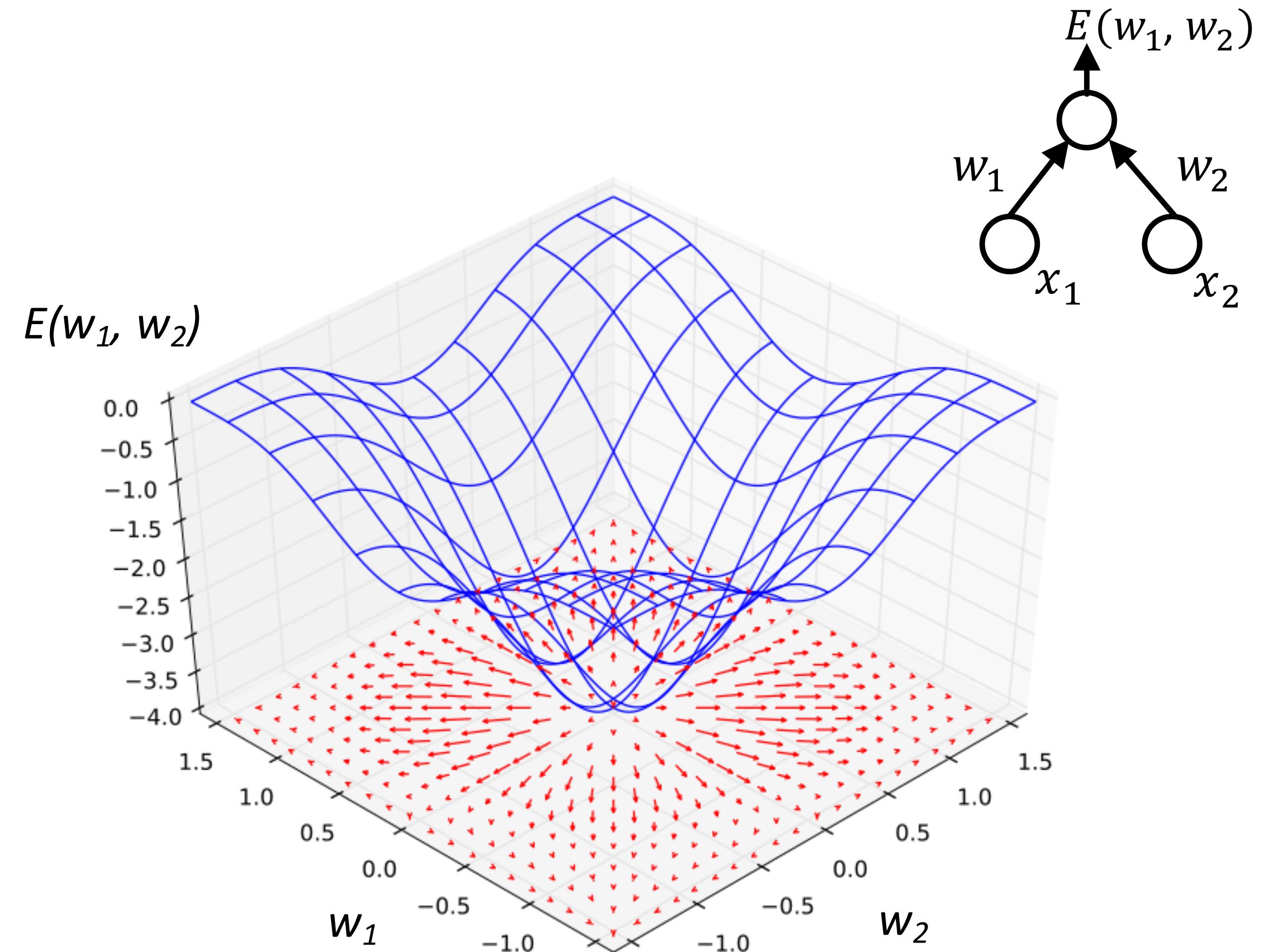
$D$  : number of parameters

- The gradient is a **vector field**:

$$\nabla_{\mathbf{w}} E : \mathbb{R}^D \rightarrow \mathbb{R}^D$$

$$\nabla_{\mathbf{w}} E = \begin{pmatrix} \frac{\partial E}{\partial w_1} \\ \vdots \\ \frac{\partial E}{\partial w_D} \end{pmatrix}$$

- $(\nabla_{\mathbf{w}} E)(\mathbf{w}^*)$  points in the direction of the steepest increase of the error at  $\mathbf{w}^*$ .



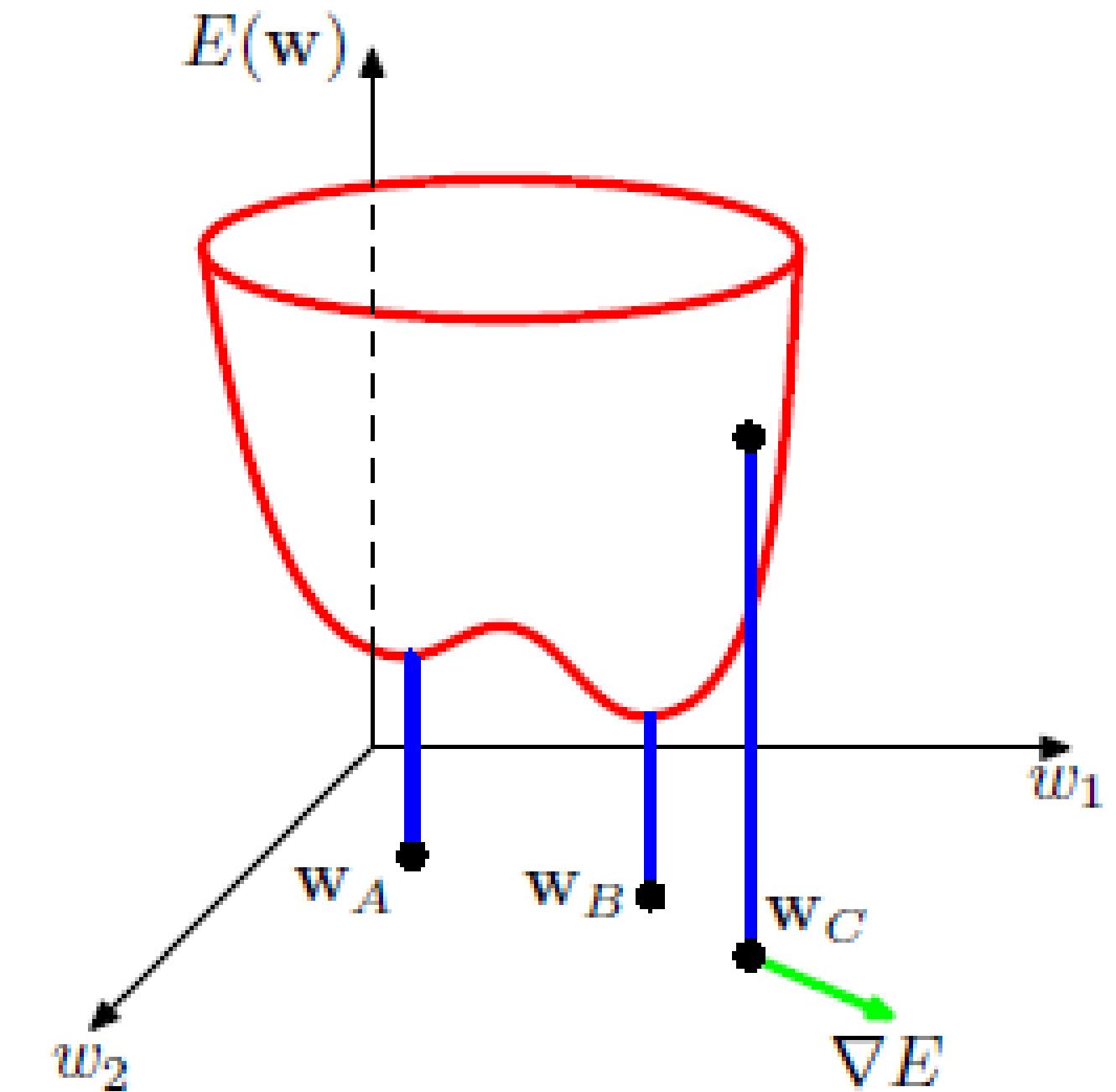
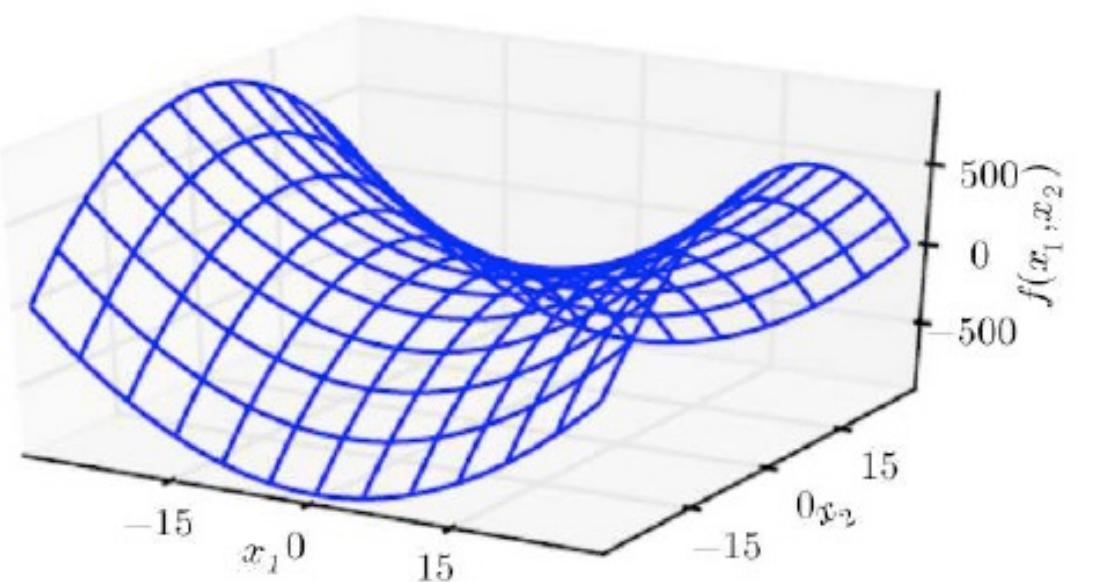
# Loss minimization

For training we try to minimize the loss function.

The gradient  $\nabla_{\mathbf{w}} E$  provides useful information.

**Stationary points:** Points  $\mathbf{w}^*$  where  $\nabla_{\mathbf{w}} E(\mathbf{w}^*) = 0$

- Minima:
  - Global minima:  $E(\mathbf{w}^*) \leq E(\mathbf{w})$  for all  $\mathbf{w}$ .
  - Local minima: Smaller errors exist.
- Maxima
- Saddle points:



**Our task:** Find a good local minimum.

# Training: Gradient Descent

- The negative gradient

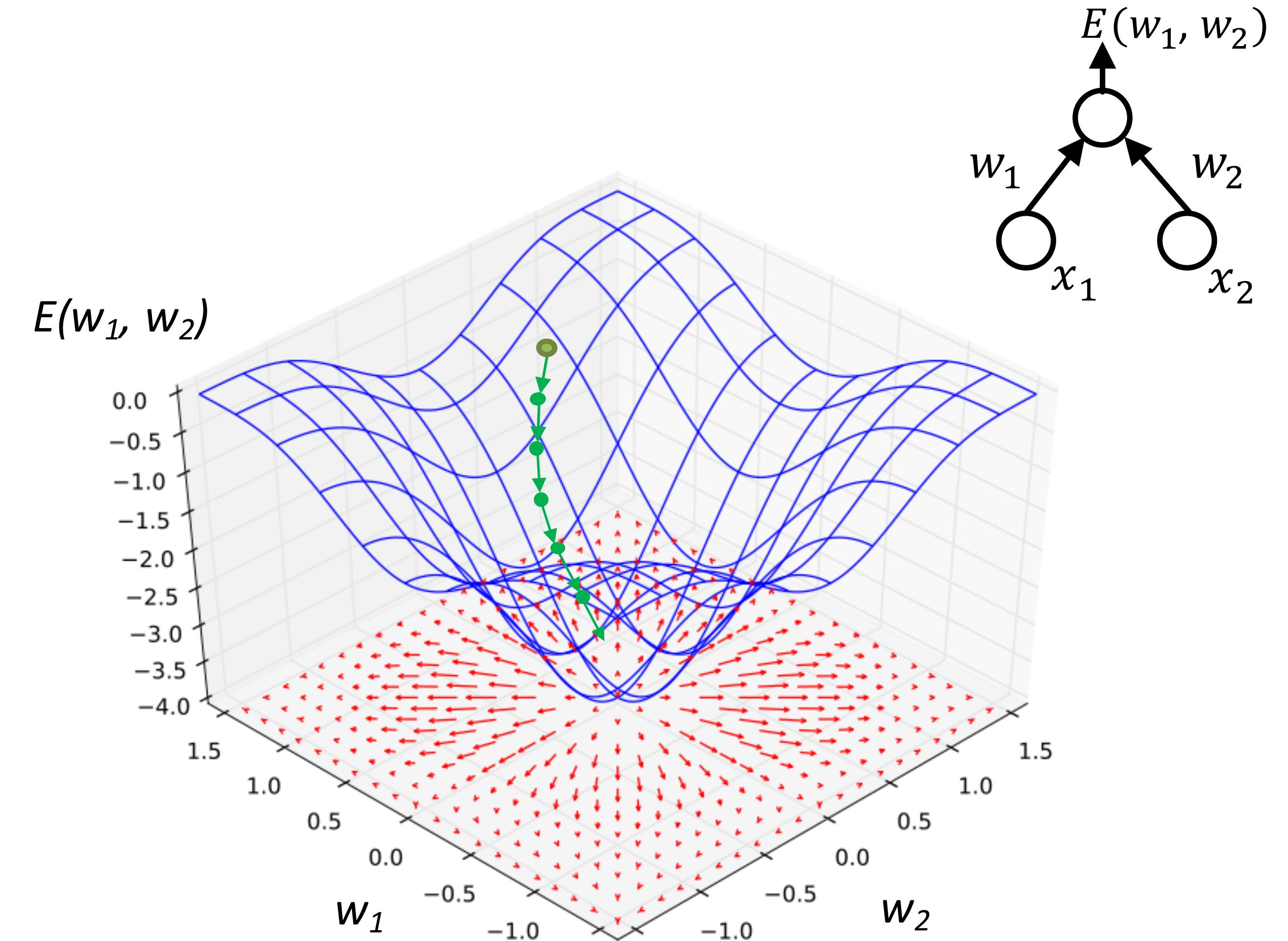
$$-\nabla_{\mathbf{w}} E(\mathbf{w}_{old})$$

at  $\mathbf{w}_{old}$  points in the direction where the error **decreases** most strongly.

- We slightly change parameters to reduce error.

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}_{old})$$

$\eta > 0$  is the **learning rate**.



# Batch Gradient Descent

```
1 Choose  $w$  randomly  
2 i  $\leftarrow 0$   
3 REPEAT  
4      $w \leftarrow w - \eta(i) \nabla_w E(w)$   
5     i  $\leftarrow i+1$   
6 UNTIL Stopping Criterion met
```

- This follows the true gradient of the error.
- Error function has the form:  $E(w) = \sum_{m=1}^M E_m(w)$
- Hence, we have  $\nabla_w E(w) = \sum_m \nabla_w E_m(w)$ .

$E_m(w)$  is the error for  
the training sample  $m$



# Batch Gradient Descent

```
1 Choose  $w$  randomly  
2 i  $\leftarrow 0$   
3 REPEAT  
4    $w \leftarrow w - \eta(i) \nabla_w E(w)$   
5   i  $\leftarrow i+1$   
6 UNTIL Stopping Criterion met
```

For example in the case of regression  
with the error function:

$$E(w) = \sum_m \frac{1}{2} ||y(x^{(m)}) - t^{(m)}||^2$$

- This follows the true gradient of the error.

- Error function has the form:  $E(w) = \sum_{m=1}^M E_m(w)$

- Hence, we have  $\nabla_w E(w) = \sum_m \nabla_w E_m(w)$ .

$E_m(w)$  is the error for  
the training sample  $m$



A single descent step has time complexity  $O(M)$ .

$M$ : number of training examples

# Mini-batch (Stochastic) Gradient Descent

```
1 Choose  $w$  randomly  
2 Let  $B$  be the set of minibatches  
2  $i \leftarrow 0$   
3 REPEAT  
4   FOR  $b$  in  $B$   
4      $w \leftarrow w - \eta(i) \nabla_w E(w, b)$   
5    $i \leftarrow i+1$   
6 UNTIL Stopping Criterion
```

“epoch”: one complete pass over the whole training set.

- Divide the training set into **mini-batches** of size  $M'$ .
- Approximate the true gradient by the gradient over the mini-batch.
- Here,  $E(w, b)$  denotes the error on mini-batch  $b$ .

# Mini-batch (Stochastic) Gradient Descent

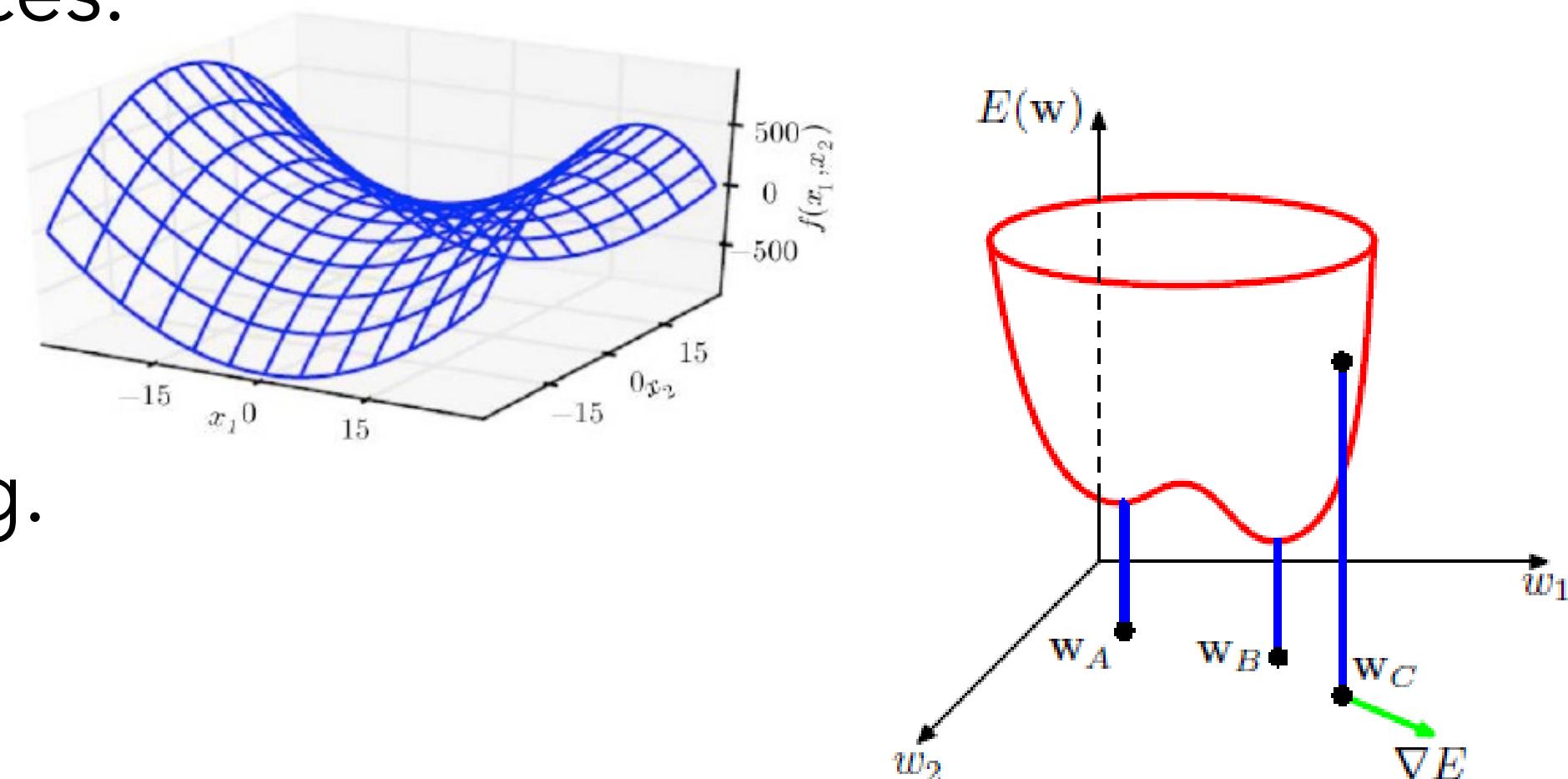
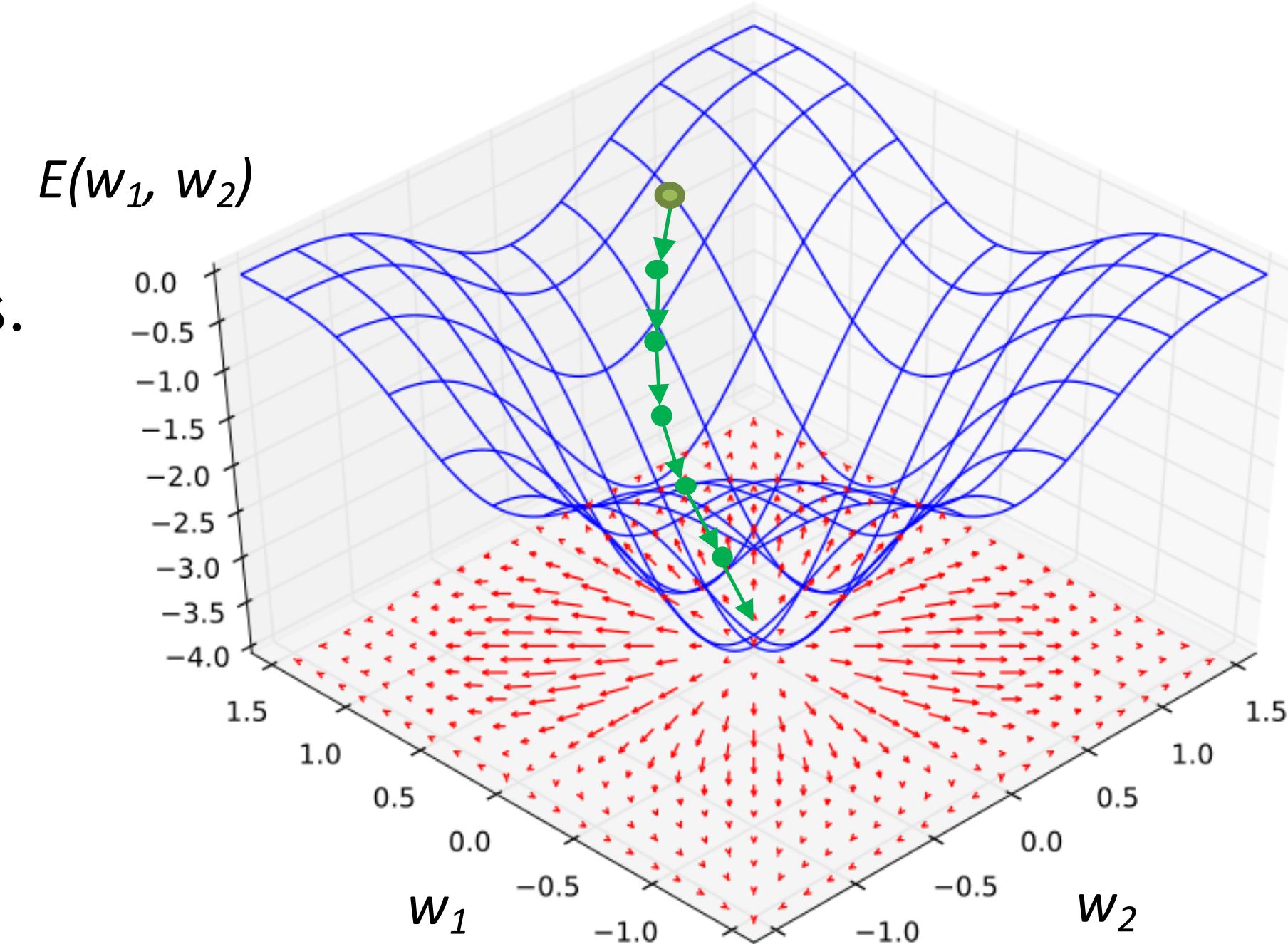
---

- Initially, SGD referred to a batch-size of 1.  
i.e., update the weights after every training example.
- Today, mini-batch GD is commonly called SGD.
- For large training sets, batch GD is too slow.
- In SGD, complexity of one update is independent of training set size.  
Instead, depends on the mini-batch size  $M'$ .
- SGD converges faster.
- Gradients within one mini-batch can be computed in parallel (e.g., on GPUs).

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \sum_m \nabla_{\mathbf{w}} E_m(\mathbf{w})$$

# When Gradient Descent struggles

- Local minima: Usually not a problem in neural networks.
  - In high-dimensional spaces, there are many downhill paths.
- Plateaus:
  - Regions of small gradients
  - Gradient scaling techniques help
- Saddle points:
  - They are much more numerous in high-dimensional spaces.
  - Small gradients.
- Ragged error landscapes:
  - Gradients can become very large → unstable training.
  - We can clip/rescale large gradients.



# Summary: Gradient Descent

---

- We want to **minimize a loss function**.
  - But we can only find local minima.
- The loss function is often dictated by the problem-type.
- Training with **gradient descent**

$$\mathbf{w}_{new} = \mathbf{w}_{old} - \eta \nabla_{\mathbf{w}} E(\mathbf{w}_{old})$$

- Mini-batch (stochastic) gradient descent  Much more efficient

*Next lecture: How do we compute the gradients in a neural network?*

# Today

---

Neural Network Architecture

Neural Network Training

Error (Loss) Functions

Gradient Descent

# Questions?