Outline

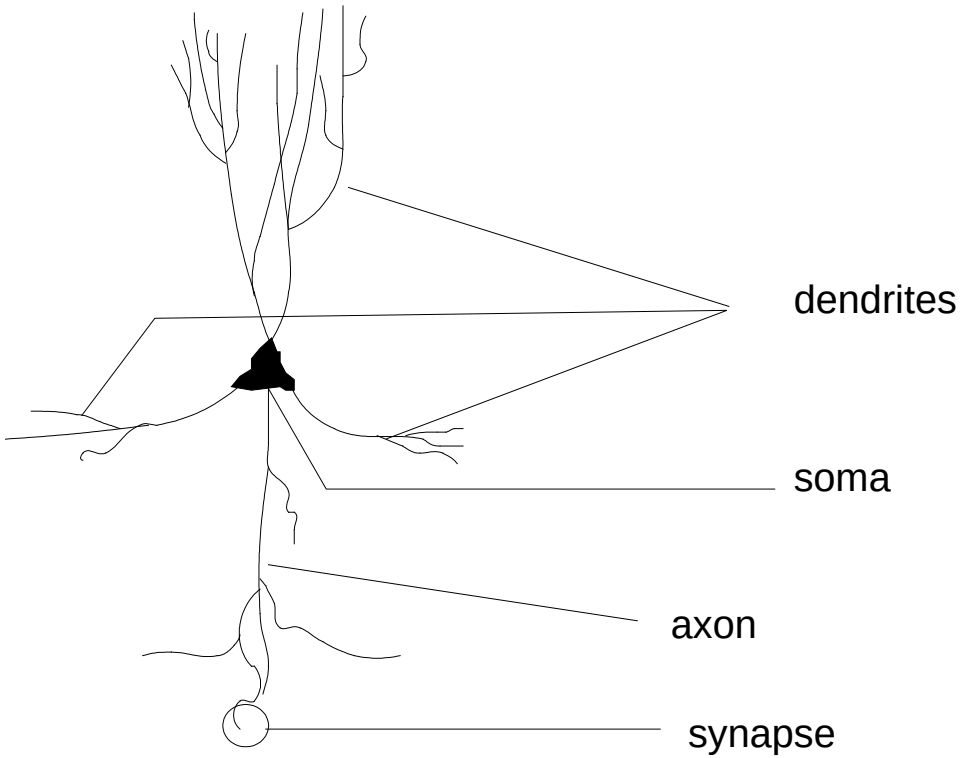
Motivation: biological neural networks

- Humans are able to process complex tasks efficiently (perception, pattern recognition, reasoning, etc.).
- Learning from examples.
- Adaptivity and fault tolerance.

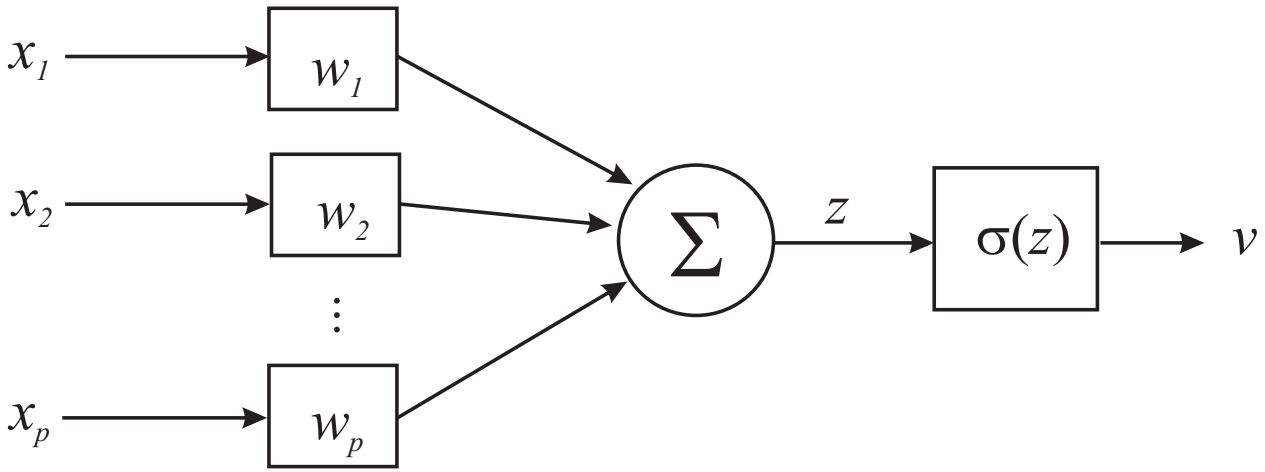
In engineering applications:

- Nonlinear approximation and classification.
- Learning and adaptation from data (black-box models).
- High dimensional inputs / outputs

Biological neuron



Artificial neuron



x_i	:	ith input of the neuron
w_i	:	adaptive weight (synaptic strength) for x_i
z	:	weighted sum of inputs: $z = \sum_{i=1}^{p} w_i x_i = \mathbf{w}^T \mathbf{x}$
$\sigma(z)$	:	activation function
v	:	output of the neuron

Activation functions

Purpose: transformation of the input space (squeezing).
Two main types:

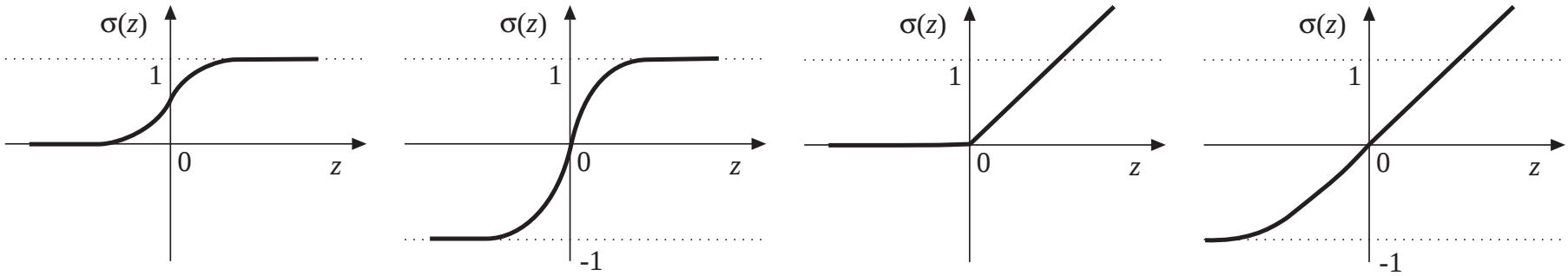
- **Projection functions**: threshold function, piece-wise linear function, tangent hyperbolic, sigmoidal, rectified linear, ... function:

$$\sigma(z) = 1/(1 + \exp(-2z))$$

- **Kernel functions** (radial basis functions):

$$\sigma(\mathbf{x}) = \exp\left(-(\mathbf{x} - \mathbf{c})^2/s^2\right)$$

Activation functions: some common choices



Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$

Tangent hyperbolic: $\sigma(z) = \frac{2}{1+e^{-2z}} - 1$

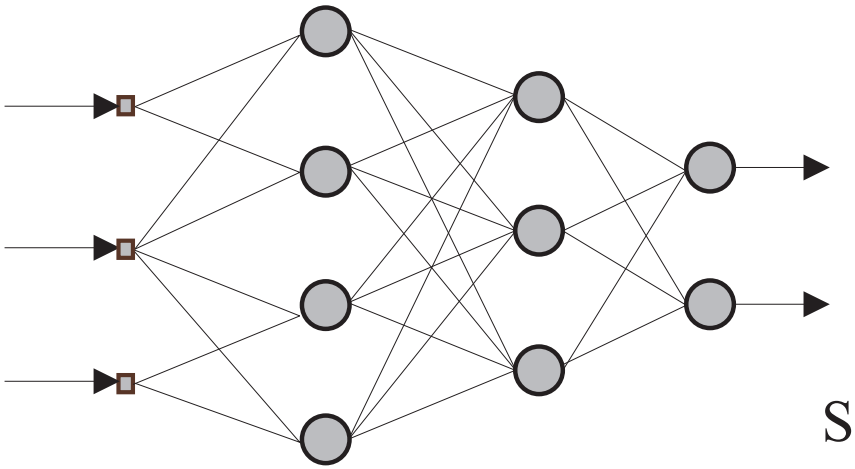
Rectified Linear Unit (ReLU): $\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$

Exponential Linear Unit (ELU): $\sigma(z) = \begin{cases} z & \text{if } z > 0 \\ \alpha(e^z - 1) & \text{if } z \leq 0 \end{cases}$

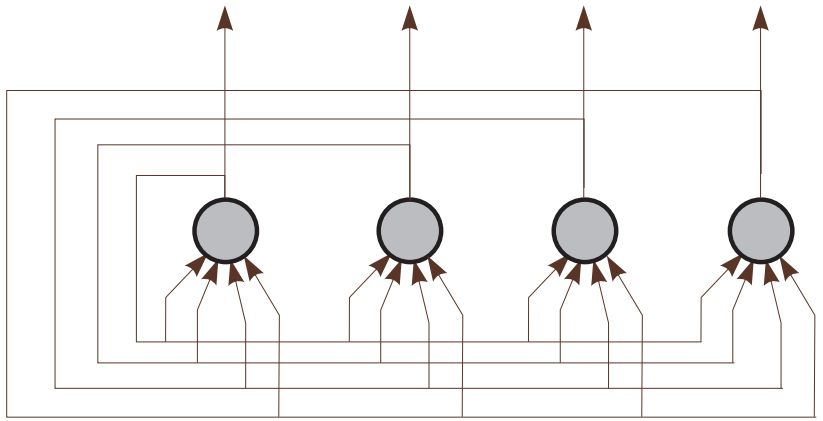
Outline

Neural Network: Interconnected Neurons

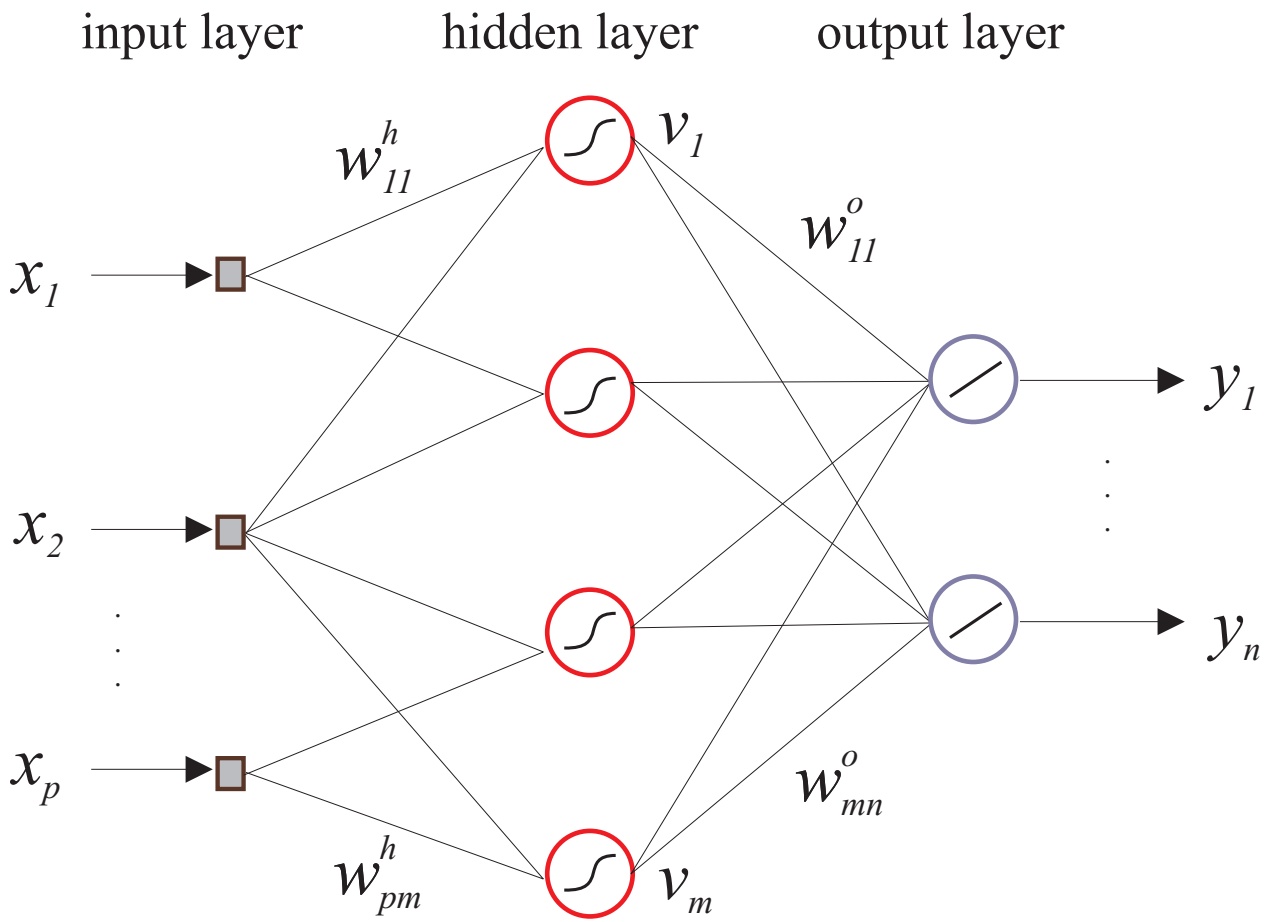
Multi-layer ANN



Single-layer recurrent ANN



Feedforward neural network example



Feedforward neural network example (cont'd)

1. Activation of hidden-layer neuron j:

$$z_j = \sum_{i=1}^{p} w_{ij}^h x_i + b_j^h$$

2. Output of hidden-layer neuron j:

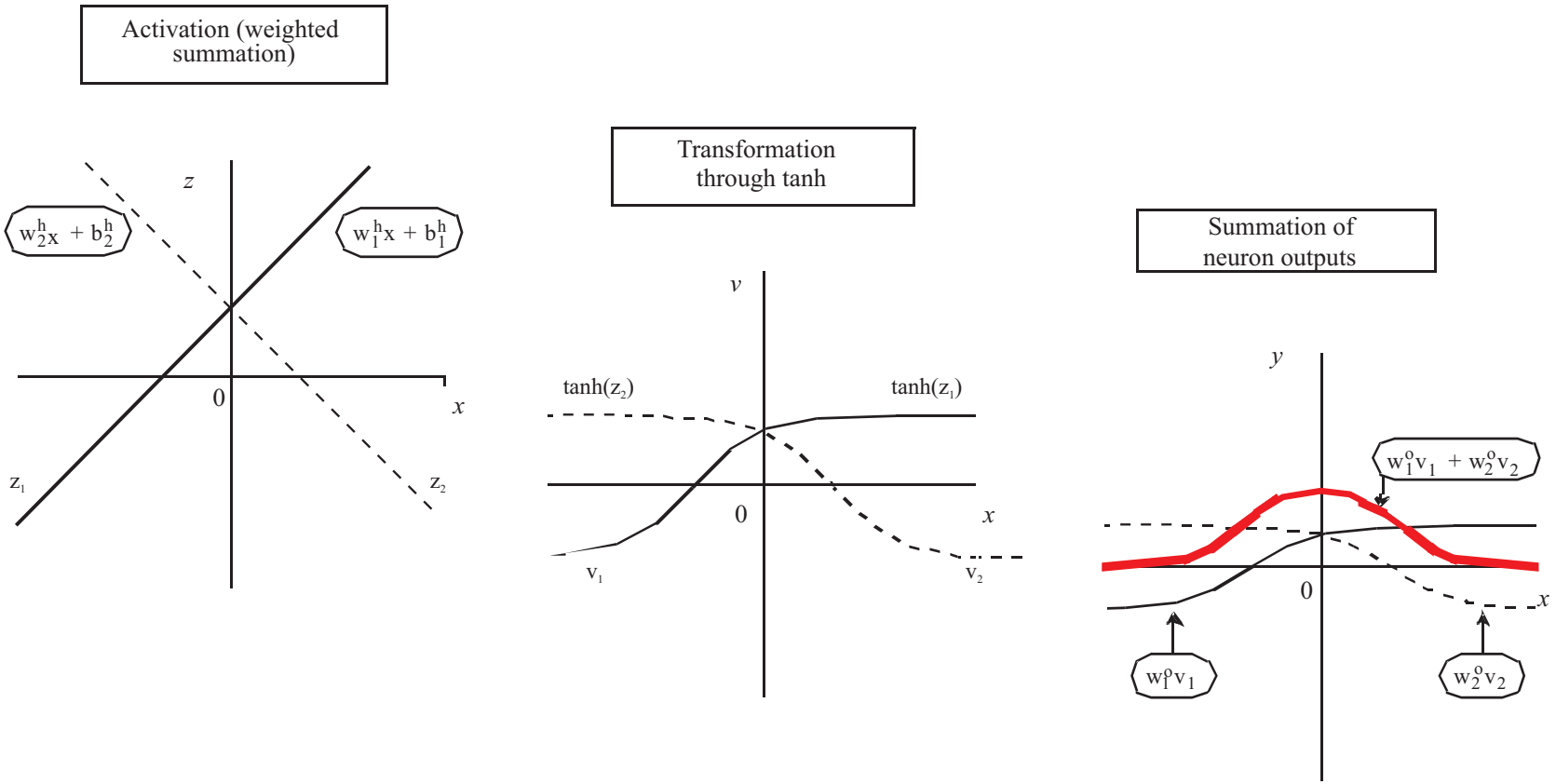
$$v_j = \sigma\left(z_j\right)$$

3. Output of output-layer neuron l:

$$y_l = \sum_{j=1}^{h} w_{jl}^o v_j + b_l^o$$

Function approximation with neural nets: examples

$$y = w_1^o \tanh\left(w_1^h x + b_1^h\right) + w_2^o \tanh\left(w_2^h x + b_2^h\right)$$



Warping space

Need for nonlinearities

Input–Output Mapping

Matrix notation:

$$\begin{aligned} \mathbf{Z} &= \mathbf{X}_b \mathbf{W}^h \\ \mathbf{V} &= \sigma(\mathbf{Z}) \\ \mathbf{Y} &= \mathbf{V}_b \mathbf{W}^o \end{aligned}$$

with $\mathbf{X}_b = [\mathbf{X}\ \mathbf{1}]$ and $\mathbf{V}_b = [\mathbf{V}\ \mathbf{1}]$.

Compact formula (1-layer feedforward net):

$$\mathbf{Y} = [\sigma([\mathbf{X}\ \mathbf{1}]\mathbf{W}^h)\ \mathbf{1}]\mathbf{W}^o$$

Approximation properties of neural nets

[Cybenko, 1989]: A feedforward neural net with at least one hidden layer can approximate any continuous nonlinear function $\mathbb{R}^p \to \mathbb{R}^n$ arbitrarily well, provided that sufficient number of hidden neurons are available (not constructive).

Approximation properties of neural nets

[Barron, 1993]: A feedforward neural net with one hidden layer with sigmoidal activation functions can achieve an integrated squared error of the order

$$J = \mathcal{O}\left(\frac{1}{h}\right)$$

independently of the dimension of the input space p, where h denotes the number of hidden neurons.

Approximation properties of neural nets

[Barron, 1993]: A feedforward neural net with one hidden layer with sigmoidal activation functions can achieve an integrated squared error of the order

$$J = \mathcal{O}\left(\frac{1}{h}\right)$$

independently of the dimension of the input space p, where h denotes the number of hidden neurons.

For a basis function expansion (polynomial, trigonometric expansion, singleton fuzzy model, etc.) with h terms, in which only the parameters of the linear combination are adjusted

$$J = \mathcal{O}\left(\frac{1}{h^{2/p}}\right)$$

Approximation properties: example

1) $p = 2$ (function of two variables):

$$\text{polynomial} \quad J = \quad \mathcal{O}\left(\frac{1}{h^{2/2}}\right) = \mathcal{O}\left(\frac{1}{h}\right)$$

$$\text{neural net} \quad J = \quad \mathcal{O}\left(\frac{1}{h}\right)$$

$\longrightarrow$ no difference

Approximation properties: example

2) $p = 10$ (function of ten variables) and $h = 21$:

$$\text{polynomial} \quad J = \quad \mathcal{O}\left(\frac{1}{21^{2/10}}\right) = 0.54$$

$$\text{neural net} \quad J = \quad \mathcal{O}\left(\frac{1}{21}\right) = 0.048$$

Approximation properties: example

2) $p = 10$ (function of ten variables) and $h = 21$:

$$\text{polynomial} \quad J = \quad \mathcal{O}(\frac{1}{21^{2/10}}) = 0.54$$

$$\text{neural net} \quad J = \quad \mathcal{O}(\frac{1}{21}) = 0.048$$

To achieve the same accuracy:

$$\mathcal{O}(\frac{1}{h_n}) = \mathcal{O}(\frac{1}{h_b})$$

$$h_n = h_b^{2/p} \quad \Rightarrow \quad h_b = \sqrt{h_n^p} = \sqrt{21^{10}} \approx 4 \cdot 10^6$$

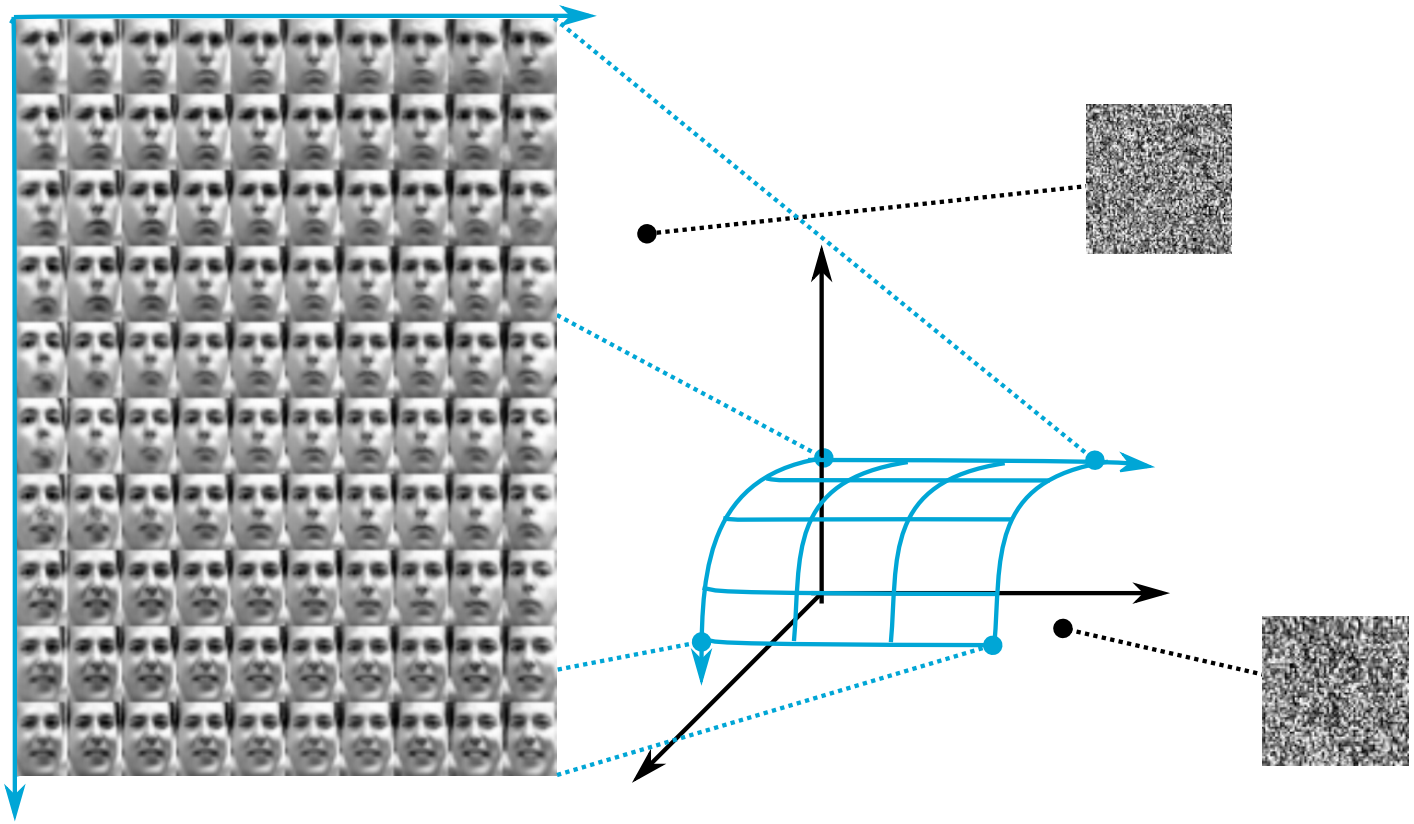
Approximation properties in practice

What does the fact that *a neural network with one layer can theoretically achieve a low approximation error independently of the dimensionality of the input space* mean in practice?

- **Does** mean neural networks are suitable for a range of problems with high dimensional inputs.
- **Does not** mean it is always possible to get near the theoretical limit.
- **Does not** mean adding more neurons per layer always results in a lower approximation error.
- **Does not** mean one layer is optimal.

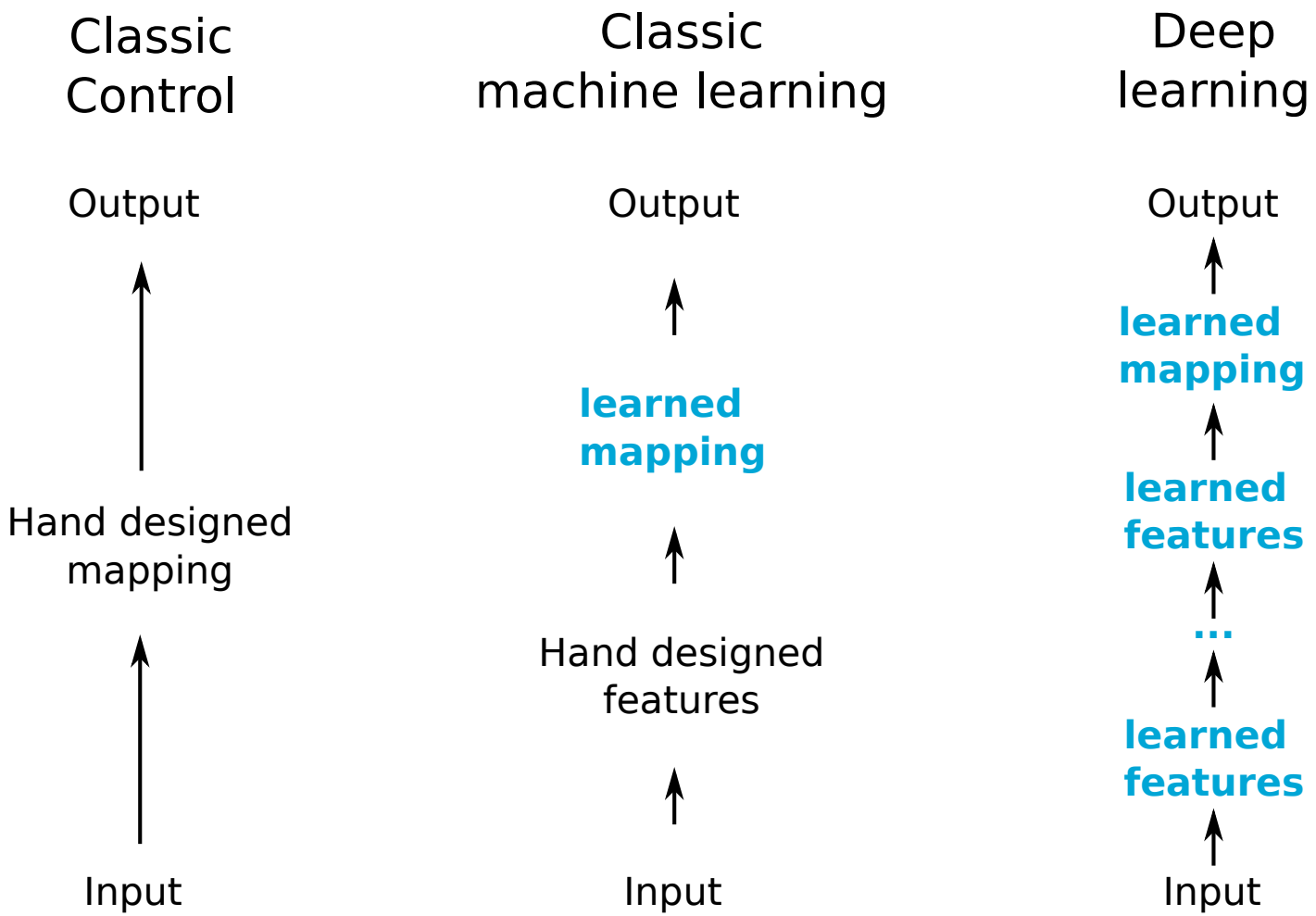
Outline

Manifold hypothesis



For many problems defined in very high dimensional spaces, the data of interest lie on low dimensional manifolds embedded in the high dimensional space. Demo: moving over a faces manifold

Deep Learning



Outline

Neural network training

Goal: find the weight vector W that minimizes some cost function $J(f(x; W))$ for all (especially unseen) inputs x.

Neural network training

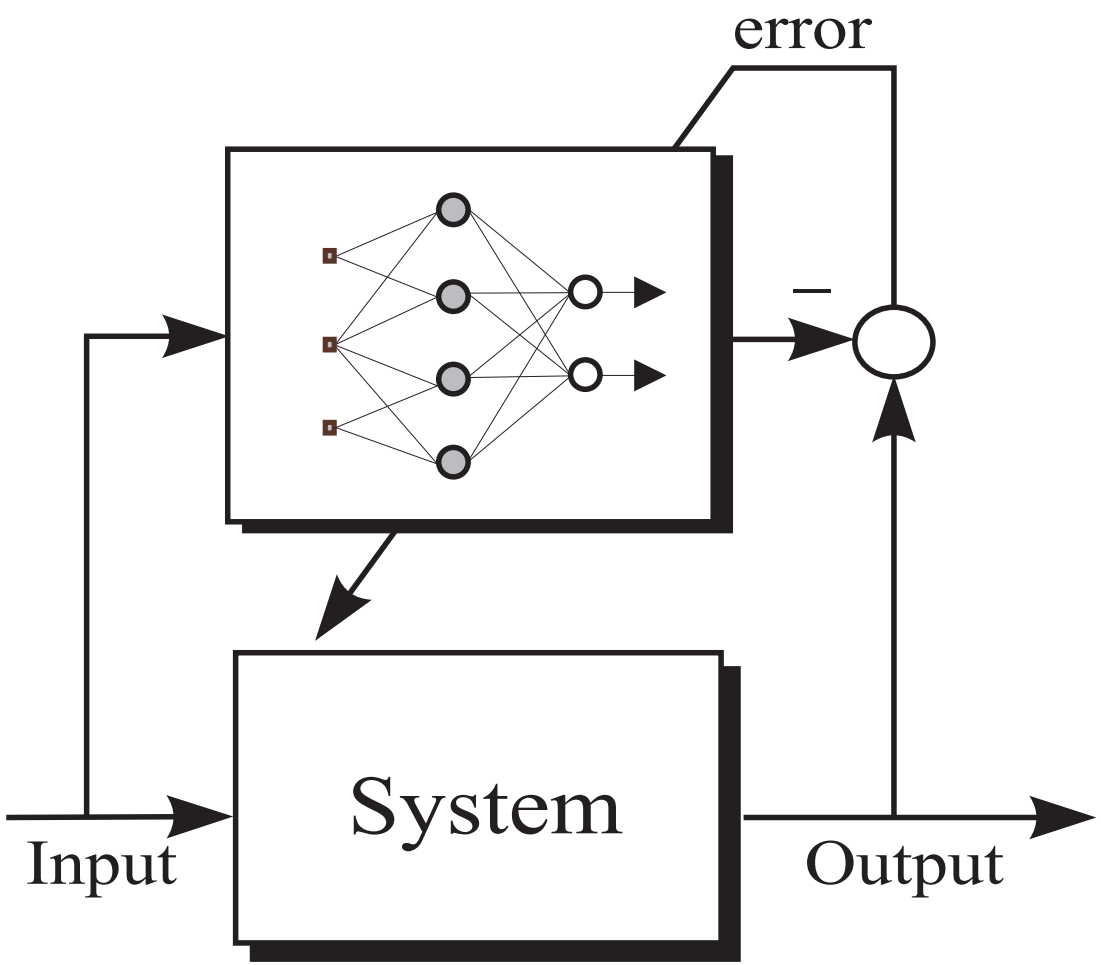
Goal: find the weight vector W that minimizes some cost function $J(f(x; W))$ for all (especially unseen) inputs x.

Supervised learning example: Make a neural network approximate a known function $x \to t$ by minimizing: $J(W) = \frac{1}{2}(f(x; W) - t)^2$

Neural network training - general algorithm

1. Initialize W to small random values
2. Repeat until the performance (on a separate test-set) stops improving:
   1. **Forward pass:** Given an input x, calculate the neural network output $y = f(x; W)$. Then calculate the cost $J(y, t)$ of predicting y instead of the *target* output t.
   2. **Backward pass:** Calculate the gradient of the cost with respect to the weights: $\nabla J(y(x; W), t) = \left[\frac{\partial J}{\partial w_1}, \quad \ldots, \quad \frac{\partial J}{\partial w_n}\right]^T$
   3. **Optimization step:** Change the weights based on the gradient to reduce the cost.

Supervised learning



Learning in feedforward nets

1. **Feedforward computation.** From the inputs proceed through the hidden layers to the output.

$$\mathbf{Z} = \mathbf{X}_b \mathbf{W}^h, \quad \mathbf{X}_b = [\mathbf{X}\ \mathbf{1}]$$

$$\mathbf{V} = \sigma(\mathbf{Z})$$

$$\mathbf{Y} = \mathbf{V}_b \mathbf{W}^o, \quad \mathbf{V}_b = [\mathbf{V}\ \mathbf{1}]$$

Learning in feedforward nets

2 **Weight adaptation.** Compare the net output with the desired output:

$$\mathbf{E} = \mathbf{T} - \mathbf{Y}$$

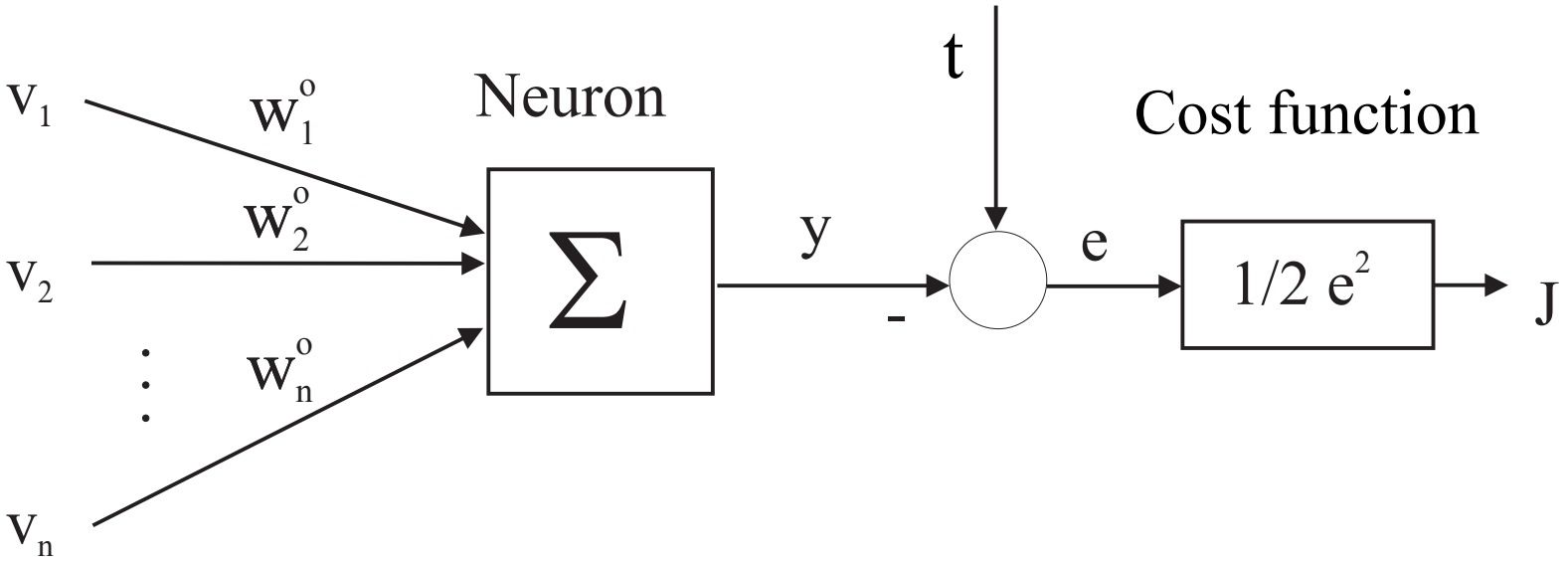
Adjust the weights such that the following cost function is minimized:

$$J(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^{N} \sum_{j=1}^{l} e_{kj}^2$$

$$\mathbf{w} = \left[\mathbf{W}^h \, \mathbf{W}^o \right]$$

(This is the *empirical loss*: the loss over the examples in the dataset. We actually want to minimize the loss over the true underlying distribution of examples. We come back to this difference in the next lecture.)

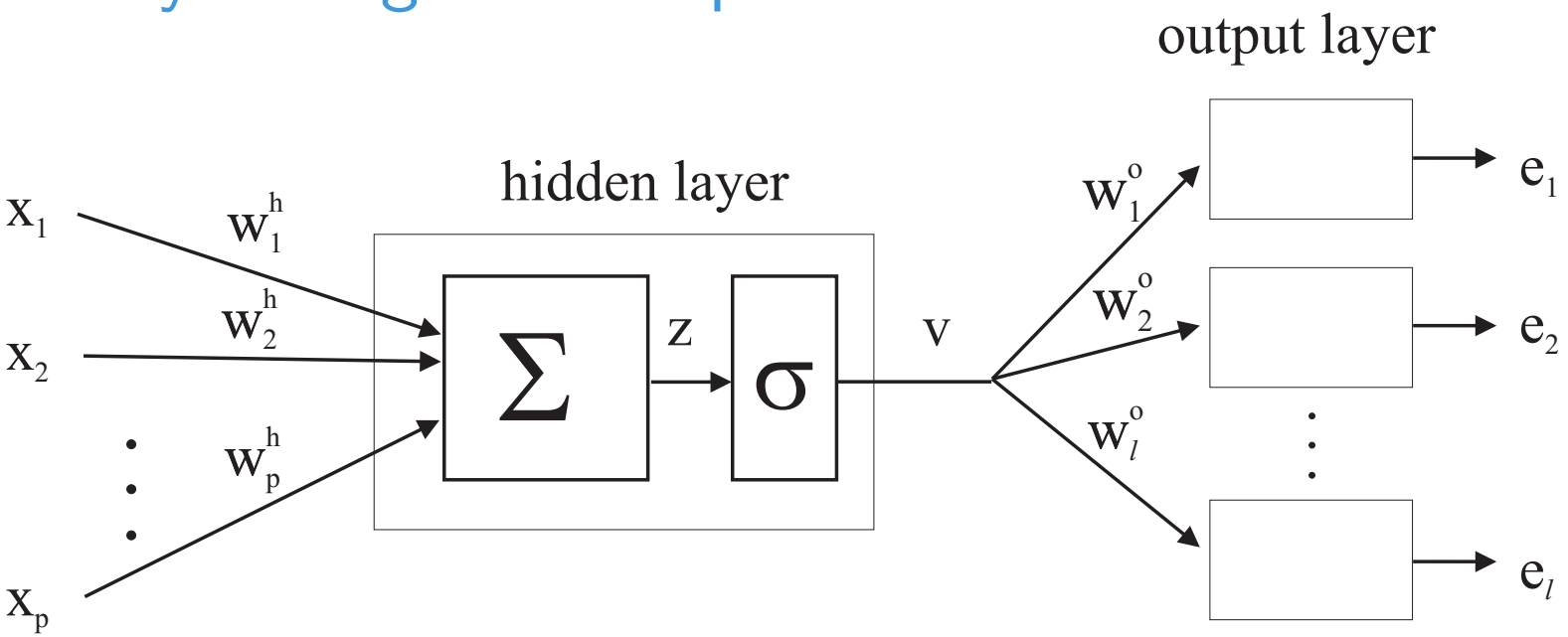
Output-layer weights example



$$J = \frac{1}{2}e^2, \quad e = t - y, \quad y = \sum_j w_j^o v_j$$

$$\frac{\partial J}{\partial w_j^o} = \frac{\partial J}{\partial e} \cdot \frac{\partial e}{\partial y} \cdot \frac{\partial y}{\partial w_j^o} = -v_j e$$

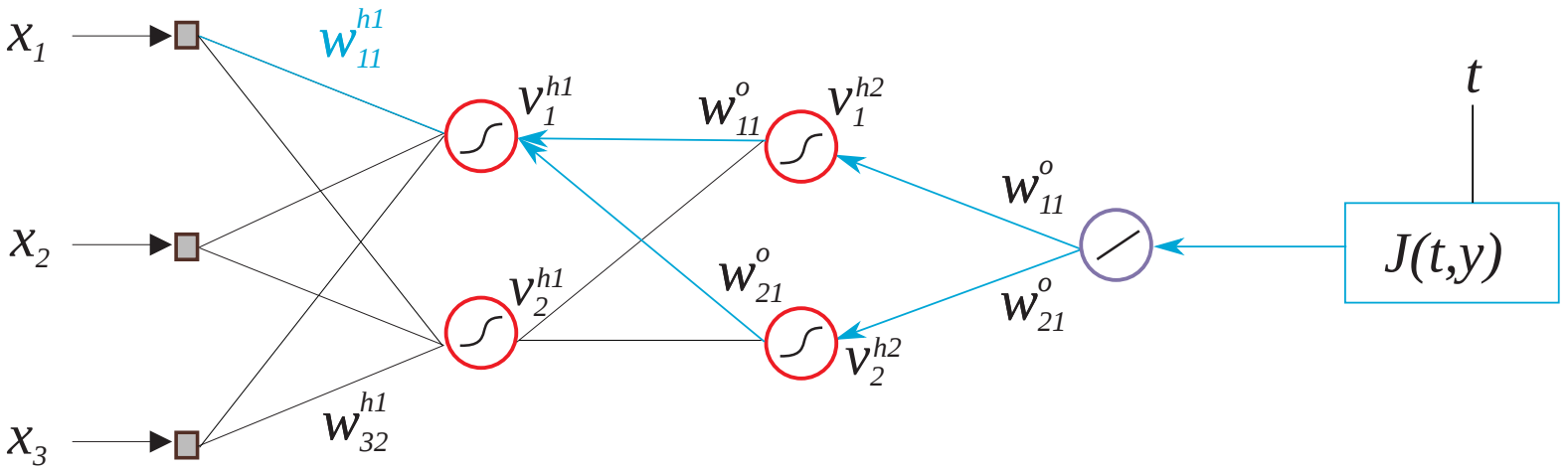
Hidden-layer weights example



$$\frac{\partial J}{\partial w_{ij}^h} = \frac{\partial J}{\partial v_j} \cdot \frac{\partial v_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}^h} = -x_i \cdot \sigma_j'(z_j) \cdot \sum_l e_l w_{jl}^o$$

$$\frac{\partial J}{\partial v_j} = \sum_l -e_l w_{jl}^o, \quad \frac{\partial v_j}{\partial z_j} = \sigma_j'(z_j), \quad \frac{\partial z_j}{\partial w_{ij}^h} = x_i$$

Back-propagating further



$$\frac{\partial J}{\partial w_{11}^{h1}} = \left(\frac{\partial J}{\partial v_1^{h2}} \frac{\partial v_1^{h2}}{\partial v_1^{h1}} + \frac{\partial J}{\partial v_2^{h2}} \frac{\partial v_2^{h2}}{\partial v_1^{h1}} \right) \frac{\partial v_1^{h1}}{\partial w_1^{h1} 1}$$

Cost functions

Cost function term types:

- Classification
- Regression
- Regularization *(next lecture)*

Two main criteria:

1. The minimum of the cost function $w^* = \arg\min_w J(f(w))$ should correspond to desirable behavior.
   examples:
   - $J(y,t) = \|y-t\|^2 \quad : \; y = f(x; w^*) \rightarrow$ mean t for each x
   - $J(y,t) = \|y-t\|_1 \quad : \; y = f(x; w^*) \rightarrow$ median of t for each x
2. The error gradient should be informative.

(Stochastic) Gradient Descent

Update rule for the weights:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \alpha_n \nabla J(\mathbf{w_n})$$

with the gradient $\nabla J(\mathbf{w_n})$

$$\nabla J(\mathbf{w}) = \left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \ldots, \frac{\partial J}{\partial w_M} \right)^T$$

- **Gradient Descent:** use $\nabla J(\mathbf{w}) = \frac{1}{K} \sum_{i=1}^{K} \left(\frac{\partial J(t_i, f(x_i; W))}{\partial W} \right)$ with K = the size of the database
- **Stochastic Gradient Descent:** use $\hat{\nabla} J(\mathbf{w}) = \frac{1}{k} \sum_{i=1}^{k} \left(\frac{\partial J(t_i, f(x_i; W))}{\partial W} \right)$ with $k << K$ the *batch* size

Stochastic Gradient Descent

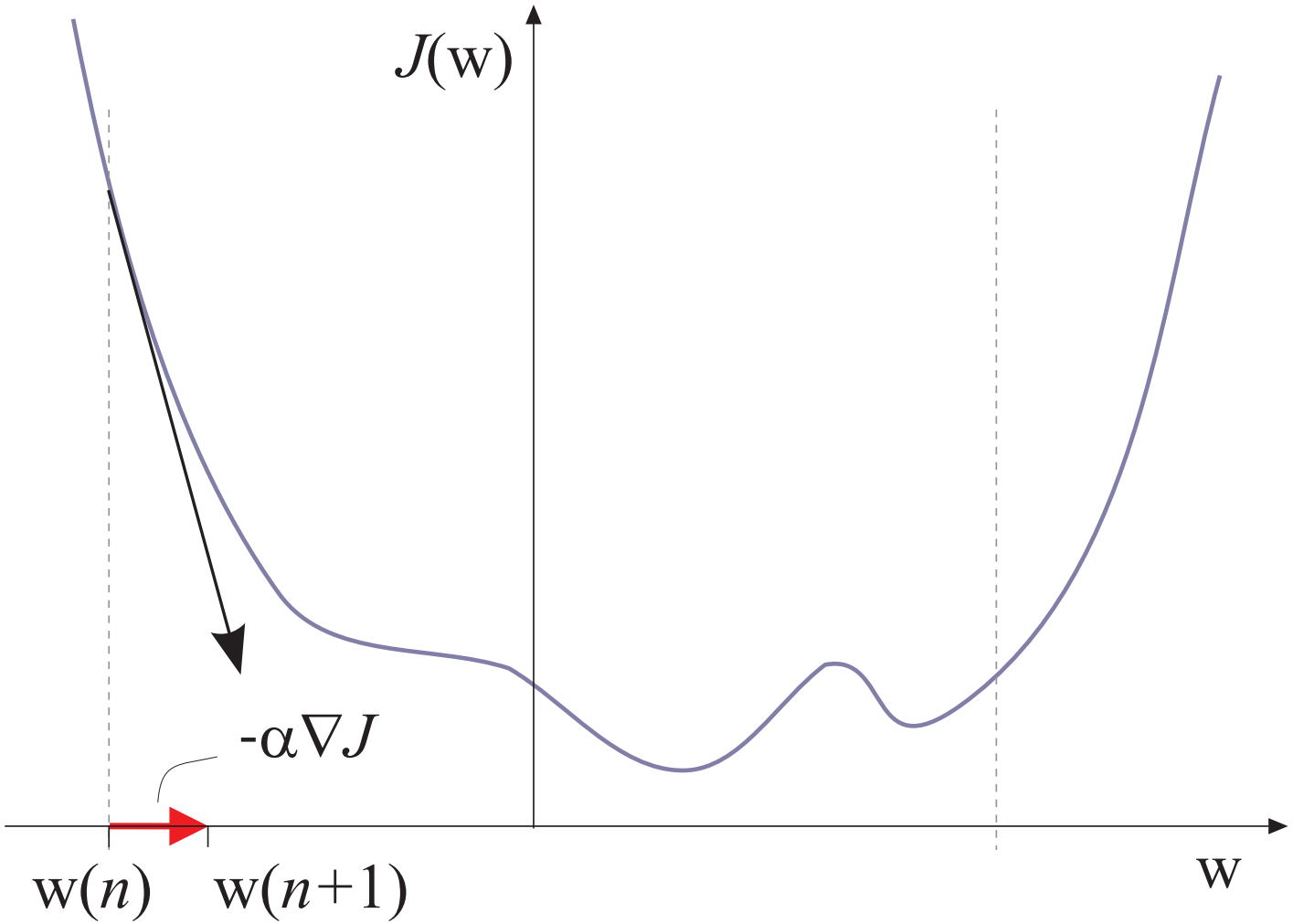
$\hat{\nabla} J(\mathbf{w}) = \frac{1}{k} \sum_{i=1}^{k} \left(\frac{\partial J(t_i, f(x_i; W))}{\partial W} \right)$ with $k << K$

In practice: $k \approx 10^0 - 10^2, \quad K \approx 10^4 - 10^9$.

The x_i, t_i data points in the batches should be independent and identically distributed (i.i.d.).

What might go wrong when learning online?

What step size?



Second-order gradient methods

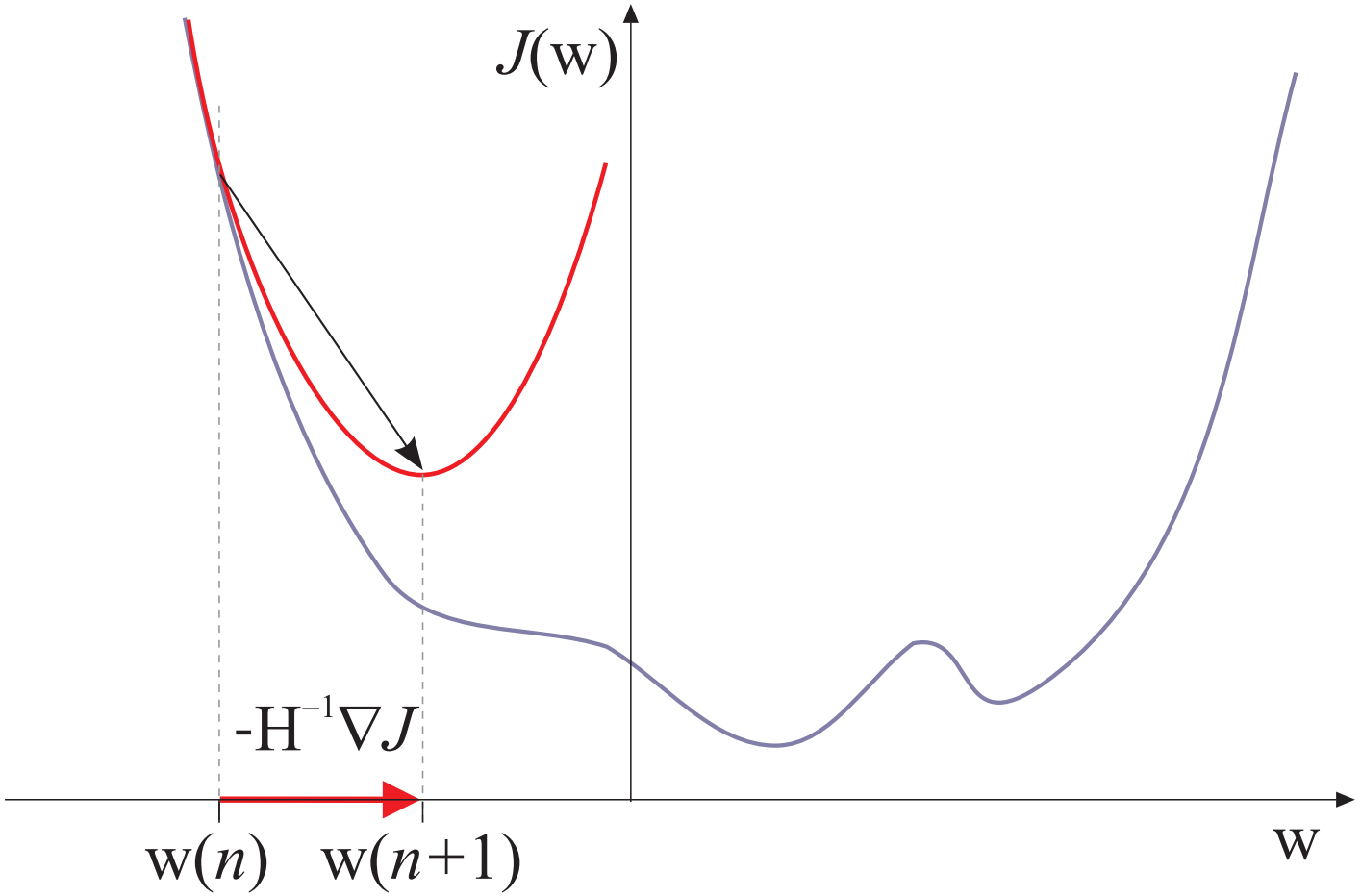
$$J(\mathbf{w}) \approx J(\mathbf{w}_0) + \nabla J(\mathbf{w}_0)^T(\mathbf{w} - \mathbf{w}_0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T \mathbf{H}(\mathbf{w}_0)(\mathbf{w} - \mathbf{w}_0)$$

where $\mathbf{H}(\mathbf{w}_0)$ is the Hessian in $\mathbf{w}_0$.

Update rule for the weights:

$$\mathbf{w}_{n+1} = \mathbf{w}_n - \mathbf{H}^{-1}(\mathbf{w}_n)\nabla J(\mathbf{w_n})$$

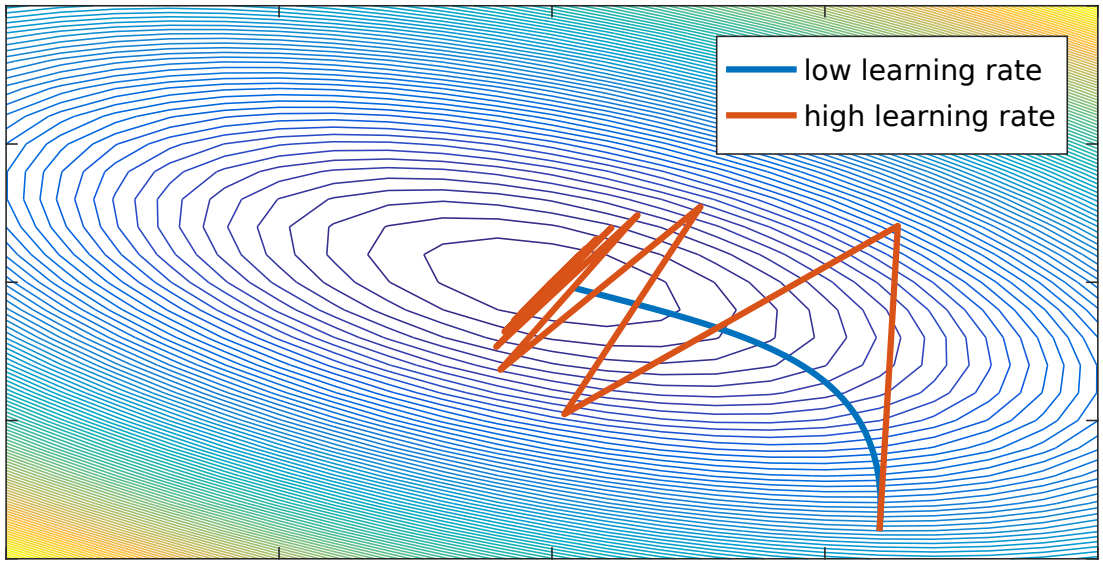
Second-order gradient methods



Step size per weight

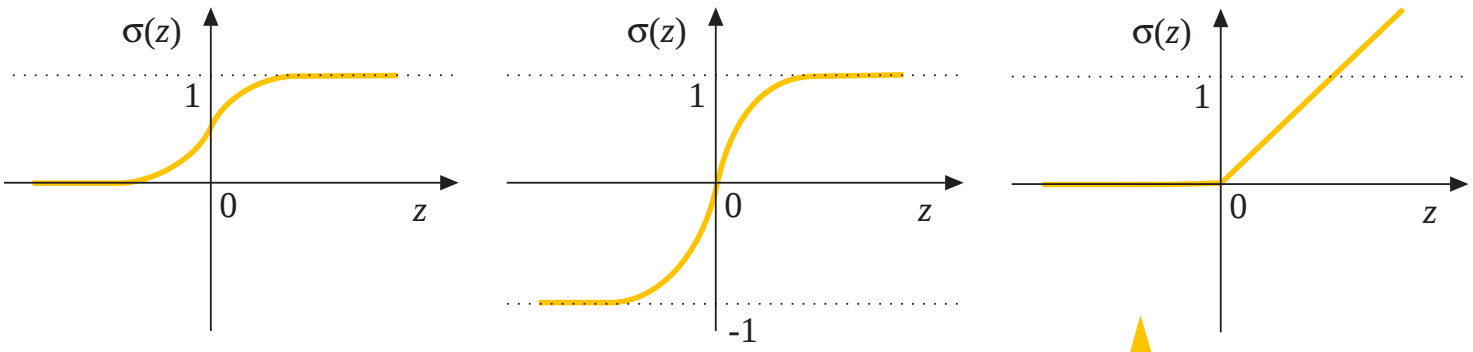
$\mathbf{H}^{-1} \nabla J$ computes a good step for each weight, only feasible for (very) small networks.

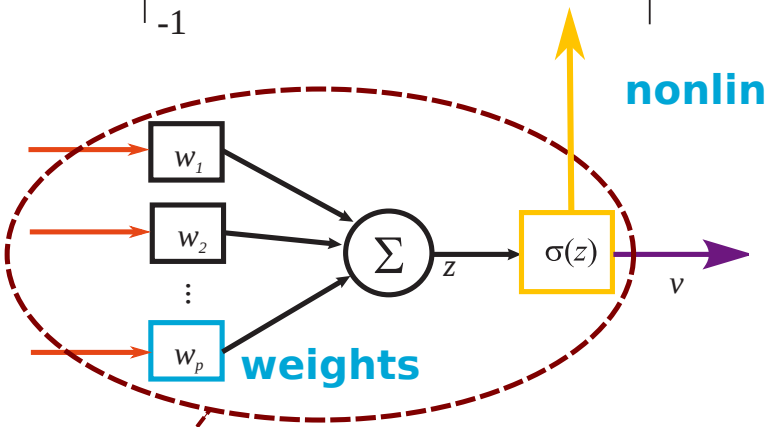
Can we do better than Gradient Descent without using second order derivatives?

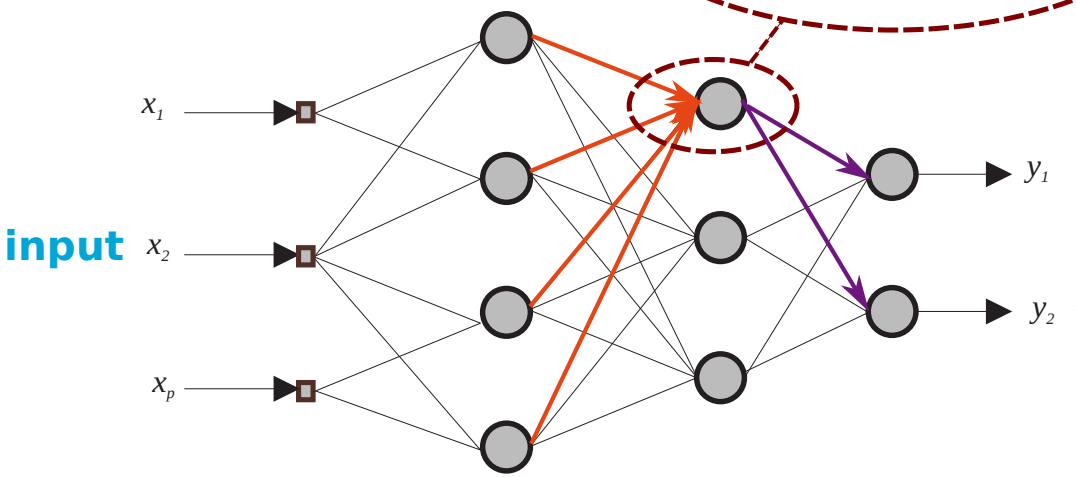


Gradient Descent

Summary artificial neural networks part 1







Foward pass:

$$y = f(x; w)$$

input

output

network structure

weights

Summary artificial neural networks part 1

Backward pass: calculate ∇J and use it in an optimization algorithm to iteratively update the weights of the network to minimize the loss J.