Neural Network Basic

Non-probabilistic discriminative classifier

Content

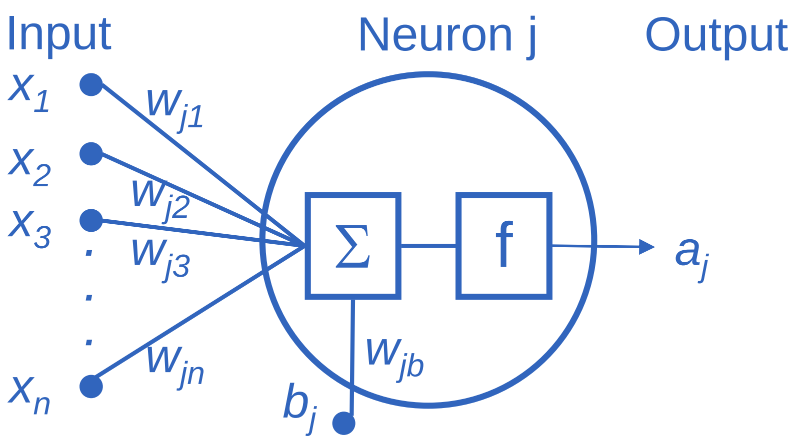
- Artificial Model of a Neuron
- The Perceptron
- Neural Networks: Multilayer Perceptron
- Training Neural Networks
- Probabilities
- Discussion

Artifical Model of a Neuron: Motivation

- The human brain is very good at interpreting scenes
- The human brain consists of relatively simple nerve cells (neurons), but these are strongly interconnected
- Assumption: The performance of the brain is related to this strong connectedness
- Attempt to simulate these network structures in pattern recognition → neural networks
- Research on neural networks started in the 1940s
- Since the 1960s, they have gone in and out of fashion several times
- Currently: Convolutional Neural Networks (CNN), deep learning

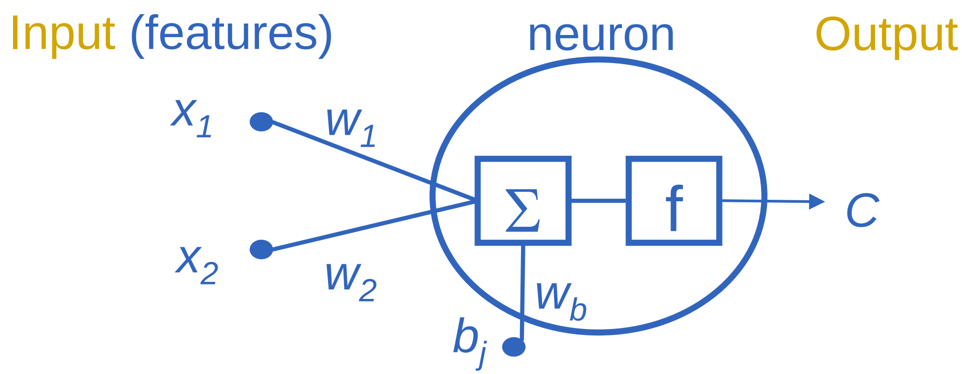
Artificial Model of a Neuron

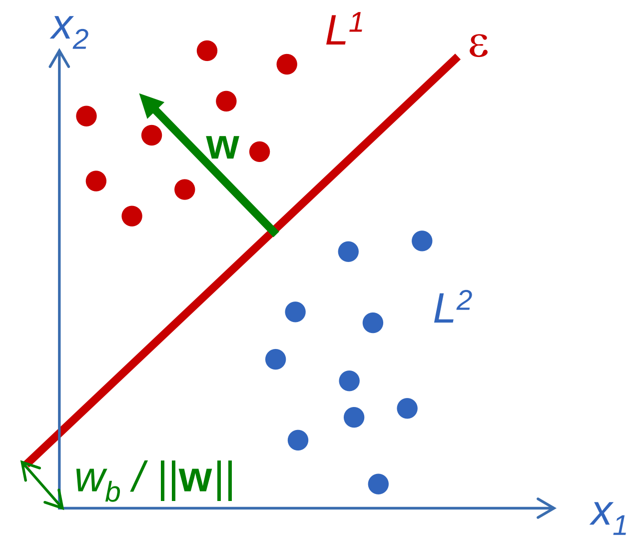
- Input variables x_i: Components of the feature vector $\underline{\mathbf{x}}$
- Each input variable is multiplied with a weight $\underline{w_{ji}}$;
  Determine weighted sum $z_j = \sum \underline{w_{ji}} \cdot \underline{x_i} + b_j = \underline{\mathbf{w}}_j^T \cdot \underline{\mathbf{x}} + b_j$
- b_j: Bias, considered to be a component of each feature vector
  → $\mathbf{x} = [\underline{\mathbf{x}}^T \; 1]^T$ and $\mathbf{w}_j = [\underline{\mathbf{w}}_j^T \; b_j]^T$
  → Simplified notation :
  $z_j = \mathbf{w}_j^T \cdot \mathbf{x}$
- Output a_j of the neuron j:
  $a_j = f(z_j) = f(\mathbf{w}_j^T \cdot \mathbf{x})$
  with $f(z_j)$ … activation function



The Perceptron: Binary Classification example

- Binary classification, Class $C = f(\mathbf{x})$, i.e. $C \in \{-1, +1\}$,
- Perceptron(can be interpreted as): a binary classifier based on a single neuron
- Example (two features x_1, x_2):





- Output: Class label $C = f(\mathbf{w}^T \cdot \mathbf{x} + w_b)$
  - Use step function as activation function → $C = (\mathbf{w}^T \cdot \mathbf{x} + w_b) > 0$
  - The decision boundary is a (hyper-) plane

The Perceptron

- Simplest possible neural network, consisting of one neuron
- Input: Vector $\mathbf{\Phi}(\mathbf{x})$
  - Derived by some (pre-defined) feature space mapping
  - One component of $\mathbf{\Phi}(\mathbf{x})$ is equivalent to the bias (value 1)
- Activation function: $f(a) = \begin{cases} +1 \text{ if } a \geq 0 \\ -1 \text{ if } a < 0 \end{cases}$
- Output: $a(\mathbf{x}) = f(\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}))$
- Wanted: Weights **w** of the perceptron
- One could try to determine **w** by minimizing the number of training samples that are assigned to the wrong class
- Problem: the activation function is a step function

Supervised Learning: Perceptron

- Causes due to the activation function problem: one cannot compute gradients, and Gradient descent is impossible
- Better choice: apply the perceptron criterion according to Rosenblatt (1962!)
- Perceptron criterion: Minimize the error function
  $E_p(\mathbf{w}) = \Sigma \max(0, -[\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}_n)] \cdot C_n)$
- Note that for a sample that is classified correctly, the max() function will return 0 and the sample will not contribute to the error
- The error $[\mathbf{w}^T \cdot \mathbf{\Phi}(\mathbf{x}_n)] \cdot C_n$ is a linear function in regions where $\mathbf{x}_n$ is misclassified

  → the error function is piecewise linear

  → gradient descent methods can be applied

Supervised Learning: Perceptron

- Minimize the error function $E_n(\mathbf{w}) = -\Sigma\ [\mathbf{w}^T \cdot \Phi(\mathbf{x}_n)] \cdot C_n$ using stochastic gradient descent:
  - Initialize the weights with random values: $\mathbf{w}^{(0)}$
  - As long as the minimum of $E_n(\mathbf{w})$ is not found, loop through the training data:
    - Select a training sample $\mathbf{x}_n$ with class C_n
    - Classify $\mathbf{x}_n$ using the current values of $\mathbf{w}$ → class C'_n
    - If $C'_n \neq C_n$: Determination of new weights:

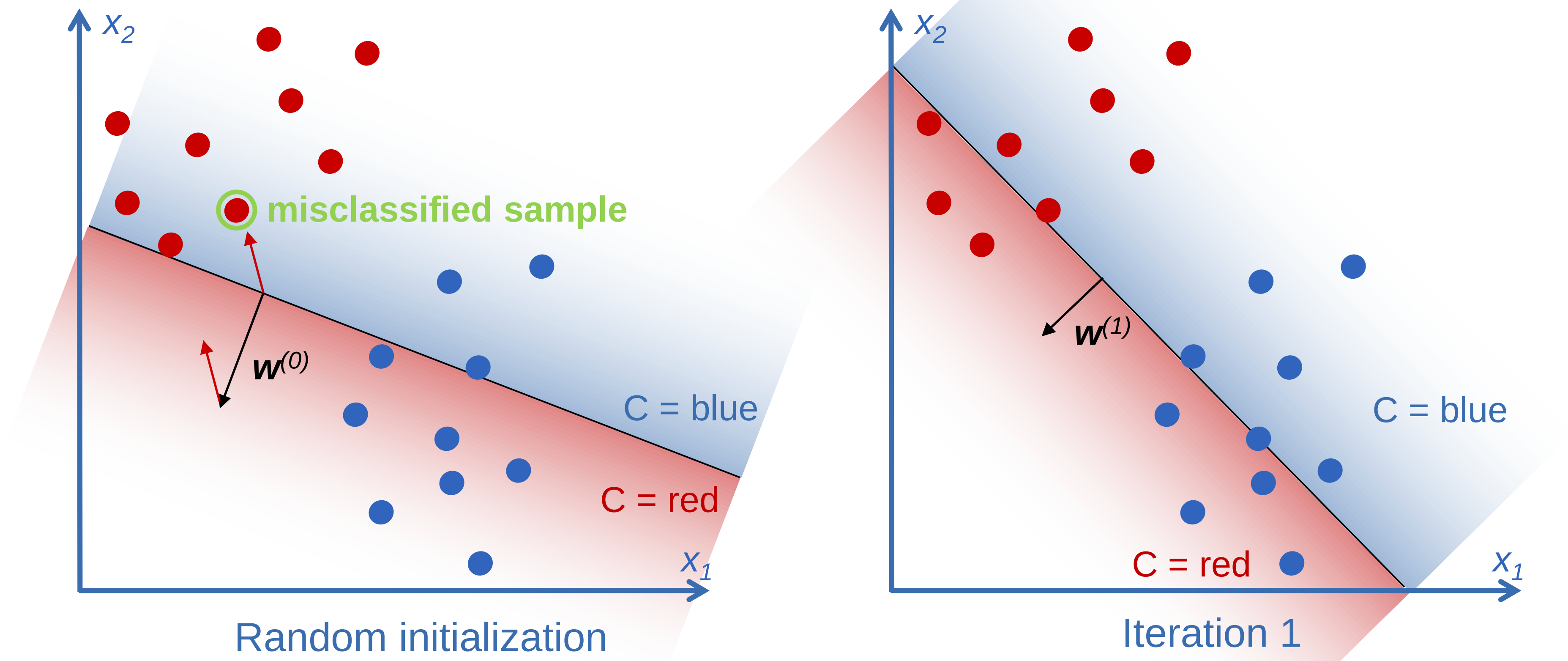
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \cdot \nabla E_n(\mathbf{w}^{(\tau)}) = \mathbf{w}^{(\tau)} + \eta \cdot \Phi(\mathbf{x}_n) \cdot C_n$$

with η … learning rate (can be set to 1 in this case)

- This procedure is guaranteed to converge (… but may be slow)

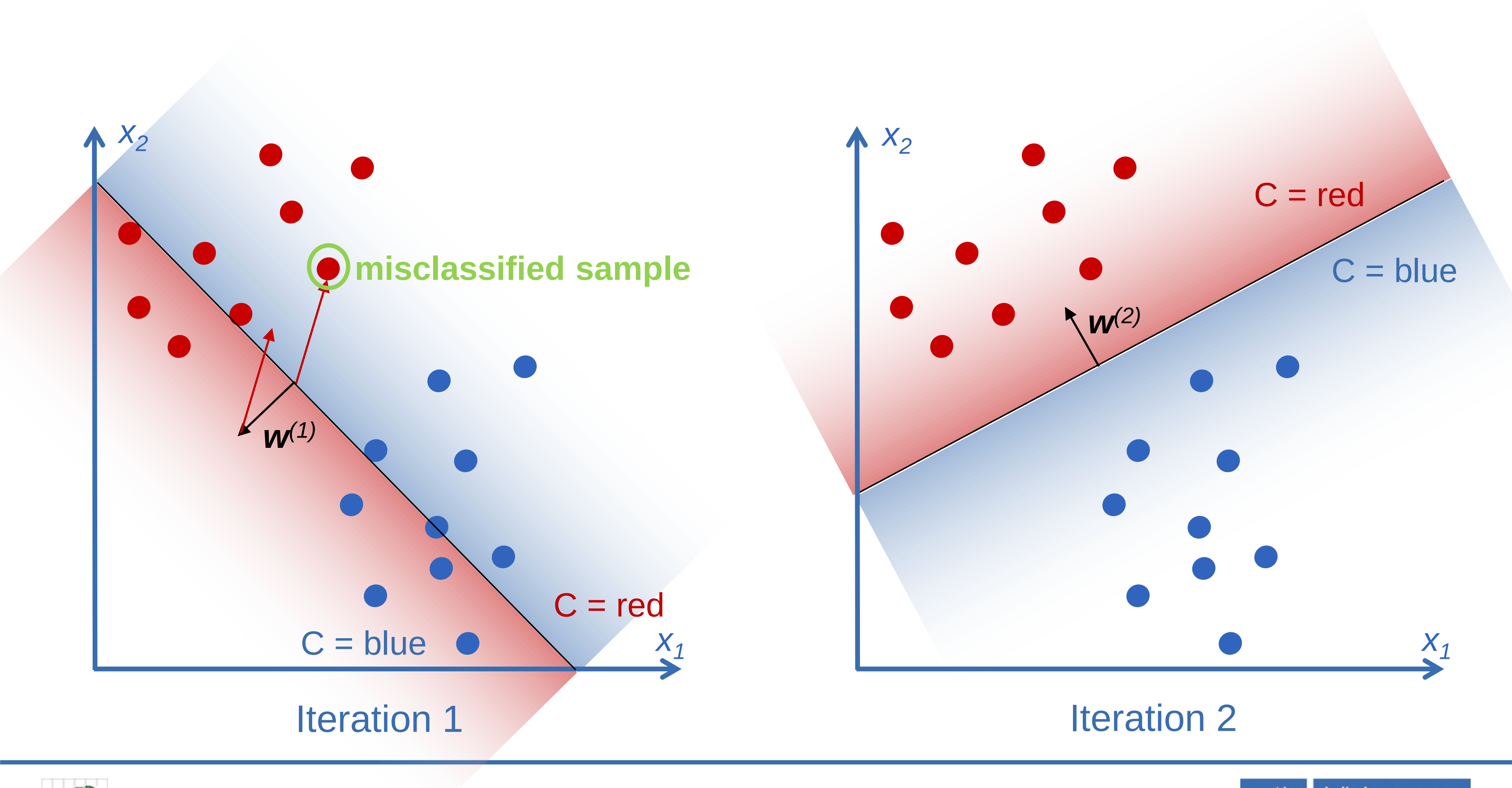
Supervised Learning: Perceptron - Example

- **Start**: initial vector $\mathbf{w}^{(0)}$ (black), randomly selected training sample $\mathbf{x}_n$ assigned to the wrong class (green circle).
- Red vector: error vector of the misclassified sample (η =1), it is added to $\mathbf{w}^{(0)}$ to obtain $\mathbf{w}^{(1)}$ in iteration 1



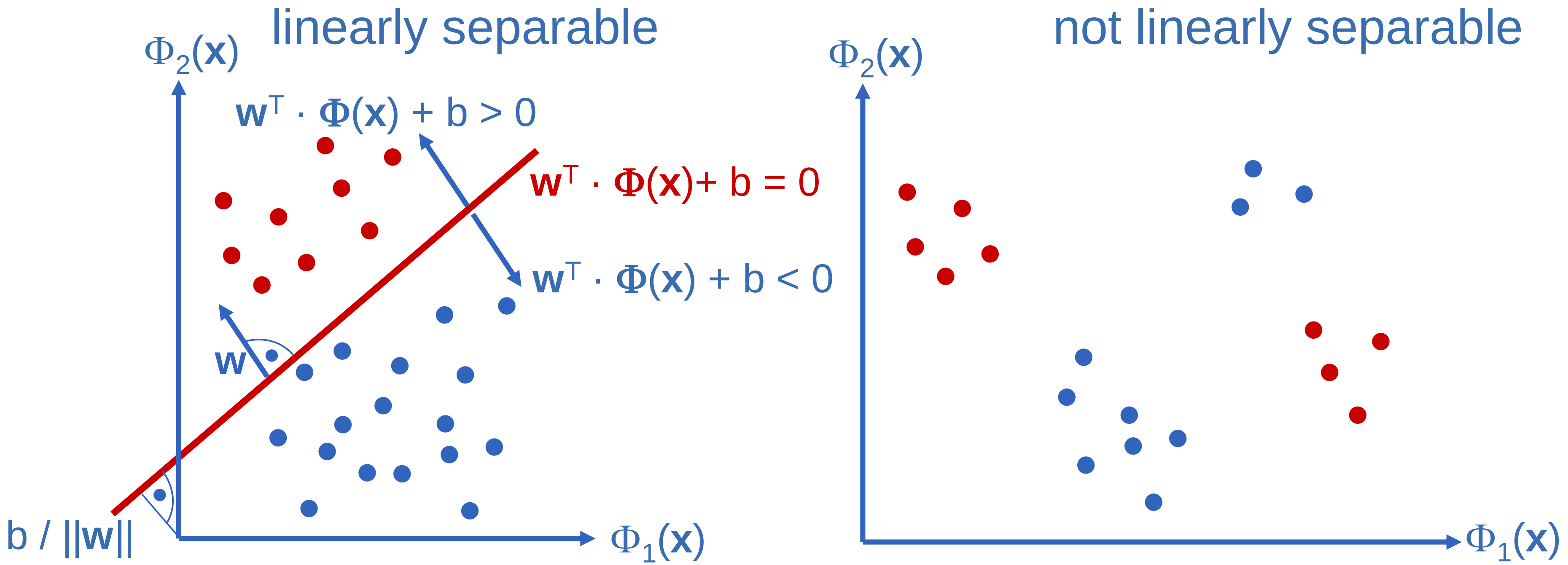
Supervised Learning: Perceptron - Example

- **Centre**: Red vector: error vector of another misclassified sample (η =1), it is added to $\mathbf{w}^{(1)}$ to obtain $\mathbf{w}^{(2)}$ in iteration 2



Geometrical Interpretation of the Perceptron

- Perceptron delivers a hyperplane as decision boundary
- Single layer perceptron only works if the classes are linearly separable in feature space
- Example for 2D feature space mapping $\mathbf{\Phi}(\mathbf{x}) = (\Phi_1(\mathbf{x}), \Phi_2(\mathbf{x}))^T$



Neural Networks: Multilayer Perceptron

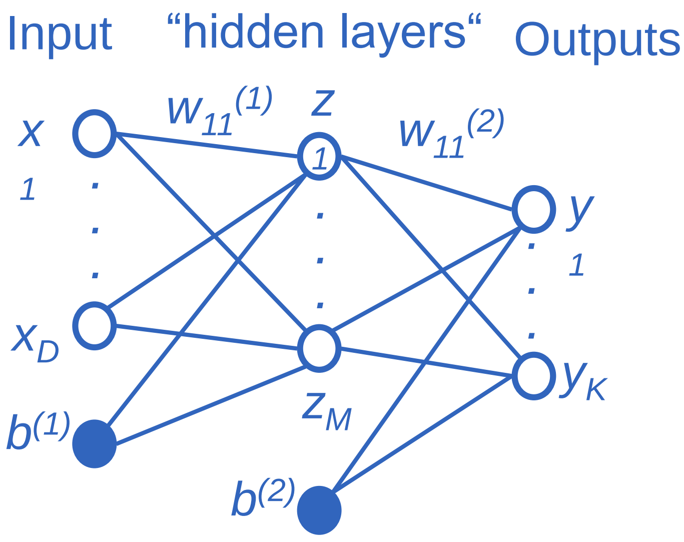
- What if more complex decision boundaries are needed: Networks consisting of several layers of neurons
- Example: two layers and "feed forward" - architecture
- Input: features x_i
- Hidden layer with neurons z_j:

  $z_j = f(\Sigma\, w_{ji}^{(1)} \cdot x_i)$

- Output: degree of membership to class C^k

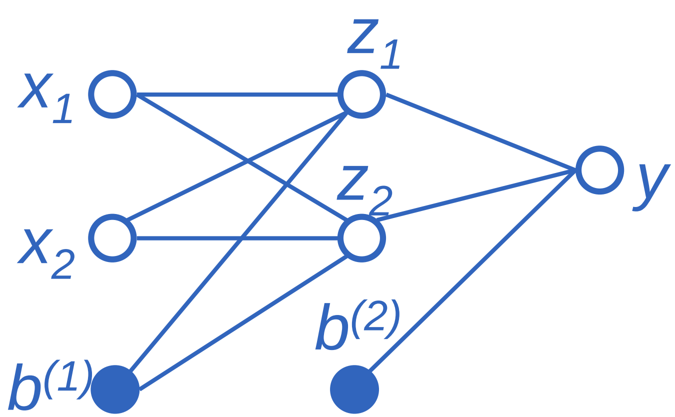
  $y_k = f(\Sigma\, w_{ki}^{(2)} \cdot z_i)$

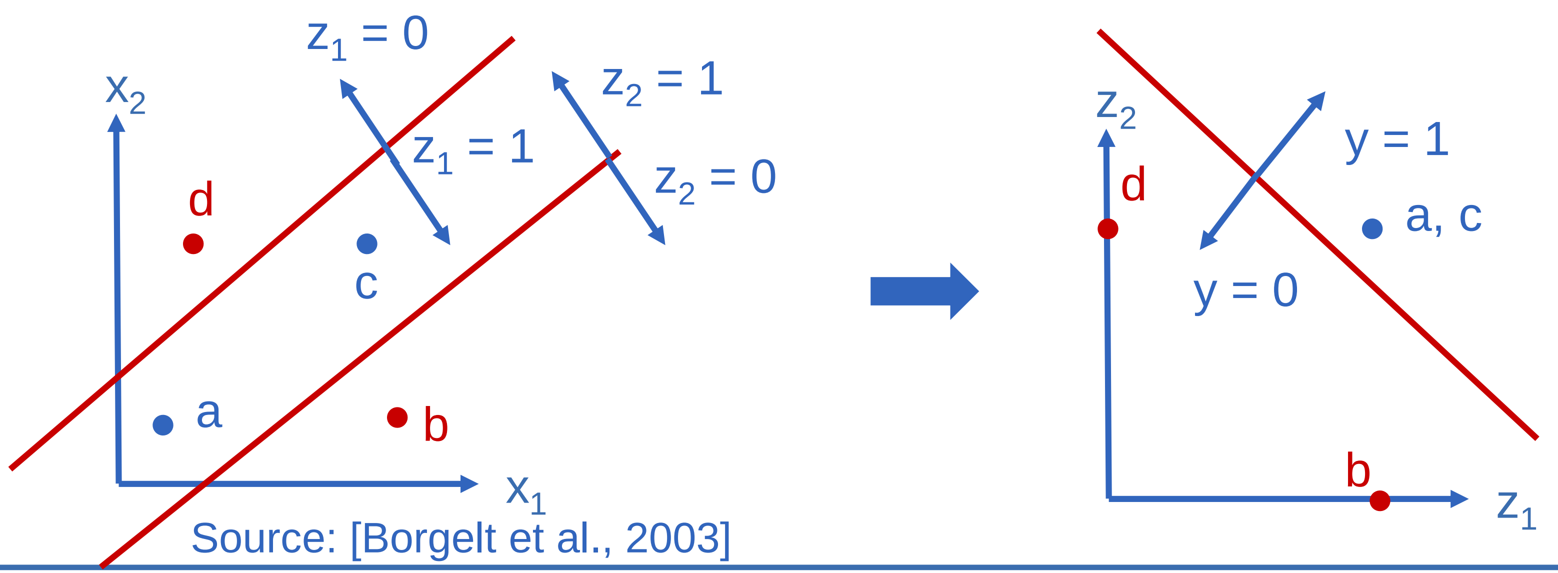
- Extension to more "hidden layers" → **Multilayer Perceptron (MLP)**



Geometrical Interpretation of Multi-layered Networks

- Hidden layers act as feature space mapping with adaptive functions Φ
- **Feature space mapping can be learnt**
- Example:



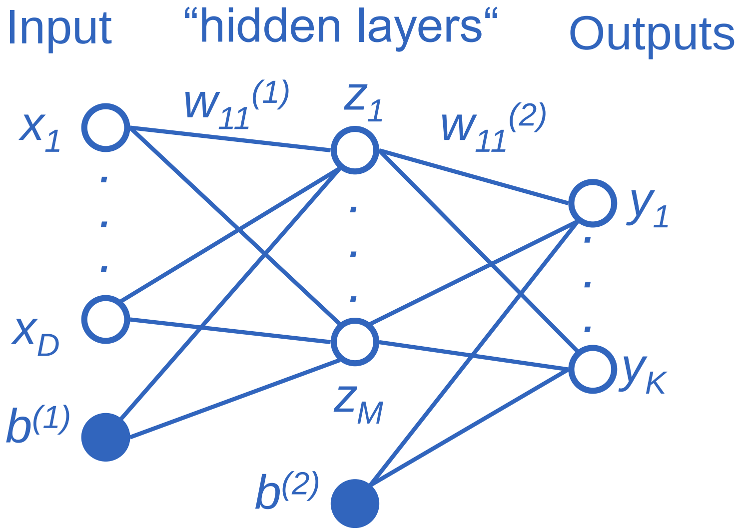


Source: [Borgelt et al., 2003]

Training Neural Networks: MLP

- Given: N feature vectors $\mathbf{x}_\mathrm{n}$ with a class membership vector $\mathbf{C}_\mathrm{n}$
  - 1-in-K representation: $\mathbf{C}_\mathrm{n} = [C_n^1, \ldots C_n^K]^\mathrm{T}$ and $C_n^k \in \{0,1\}$,
  - $C_n^k = 1$, if $\mathbf{x}_\mathrm{n}$ belongs to class C^k
- Wanted: Weights $\mathbf{w}$ of the multi-layer network
- Activation function:
  - Today, usually ReLu
  - Output layer: softmax
- Output layer delivers membership y_{nk} of the $\mathbf{x}_\mathrm{n}$ for each class C^k:

$$y_{nk} = f(\mathbf{w}, \mathbf{x}_\mathrm{n})$$



- What are the options for the activation functions

Activation Functions

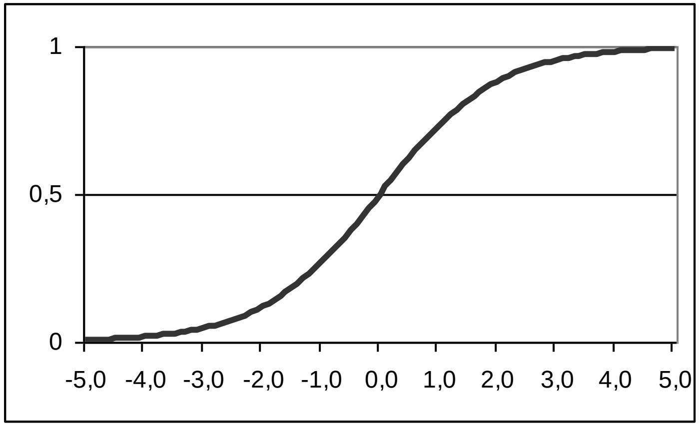
- Step function:

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

- Logistic sigmoid function:

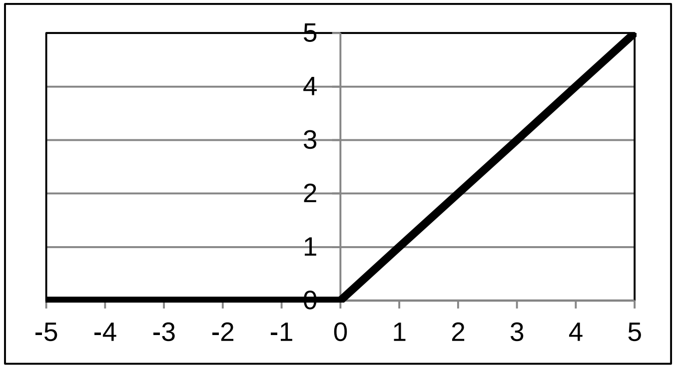
$$f(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$

with $f'(a) = f(a) \cdot [1 - f(a)]$



- Rectified Linear Unit (ReLu):

$$f(a) = \max(0, a)$$

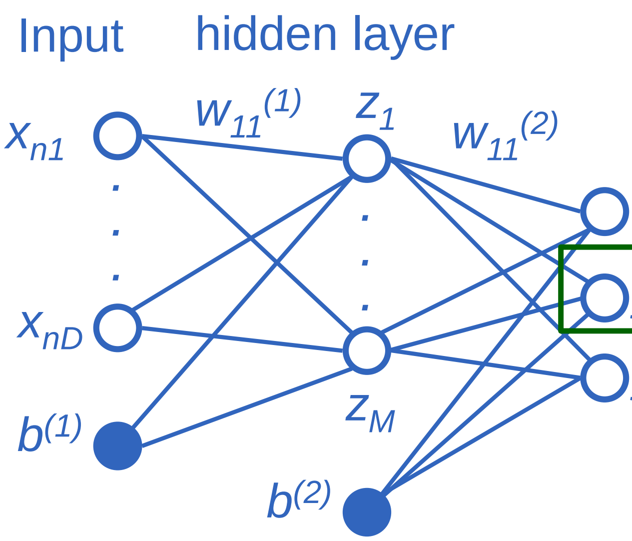


Training Neural Networks: Loss Function

- Membership y_{nk} of the feature vector $\mathbf{x}_n$ to class L^k: $y_{nk} = f(\mathbf{w}, \mathbf{x}_n)$
- Definition of an loss (error) function for $\mathbf{x}_n$, e.g.:

$$E_n(\mathbf{w},\mathbf{x}_n) = \frac{1}{2} \cdot \sum_k \left(y_{nk}(\mathbf{w},\mathbf{x}_n) - L_n^k \right)^2$$

- Example (3 classes; training sample belongs to class L^2)



Output	Training label	Error	½ · Error²
$y_{n1} = 0.1$	$L_n^1 = 0$	0.1	0.005
$y_{n2} = 0.6$	$L_n^2 = 1$	-0.4	0.080
$y_{n3} = 0.3$	$L_n^3 = 0$	0.3	0.045
		$E_n(\mathbf{w}, \mathbf{x}_n)$	Σ 0.130

Training Neural Networks: Minibatch Learning

- Total loss function: sum over all training samples:

$$E(\mathbf{w}) = \sum_n E_n(\mathbf{w}, \mathbf{x}_n) = \frac{1}{2} \cdot \sum_{n,k} \left(y_{nk}(\mathbf{w}, \mathbf{x}_n) - L_n^k \right)^2 \to \min$$

Optimization: stochastic gradient descent [Bishop, 2006]

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \cdot \nabla E(\mathbf{w}^{(\tau)})$$

- Gradient descent is mainly used in the **minibatch version**
  - Minibatch: a small (random) subset of the training samples
  - Minibatch size: e.g.128; important hyperparameter
  - In iteration τ, the sum in $E(\mathbf{w})$ is taken over all samples of the minibatch

Training Neural Networks: Initialisation

- **Preprocessing** of all training samples:
  - Subtract mean from feature values → features with **zero mean**
  - Numerical reasons!

- Initialization of the weights $\mathbf{w}^{(0)}$:
  - small random numbers , e.g. Gaussians with zero mean
  - Xavier initialization [Glorot et al., 2010]:

    $\sigma = 1/\sqrt{N_i}$ with N_i: number of input neurons of layer i

  - Better option for ReLu [He et al., 2015]: $\sigma = \sqrt{2/N_i}$
  - Initialisation is important, but may be tricky

Training Neural Networks: Gradient Descent

- Gradient descent with minibatches to minimize the error function

$$E(\mathbf{w}) = \frac{1}{2} \cdot \sum_{n,k} \left(y_{nk}(\mathbf{w}, \mathbf{x}_n) - L_n^k \right)^2$$

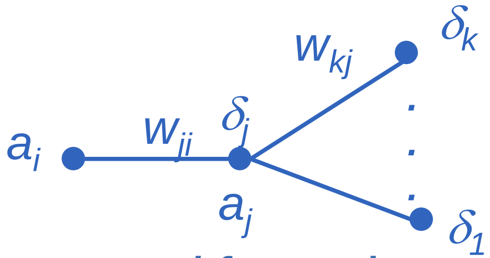
- As long as the minimum of $E(\mathbf{w})$ is not found:
    - Randomly choose a minibatch
    - Determine output y_{nk} of the neuronal network for each sample $\mathbf{x}_n$ of the current minibatch using the current values of $\mathbf{w}$
    - New weights: $\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \cdot \nabla[\Sigma_n E_n(\mathbf{w}^{(\tau)})]$ with η … learning rate, τ… Iteration count
    - The sum to compute the gradient is taken over all samples of the minibatch.

Training Neural Networks: Gradients

- The components of the gradients are the derivatives $\frac{\partial E_n(\mathbf{w})}{\partial w_{ji}}$
- Remember: in a neuron j, the signals coming from the input layer a_i are converted into an output a_j:

$$a_j = f(\mathrm{I_j}) = f\,(\Sigma\, w_{ji} \cdot a_i)$$

- Chain rule: $\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial I_j} \cdot \frac{\partial I_j}{\partial w_{ji}}$



  - $\frac{\partial I_j}{\partial w_{ji}} = a_i$ i.e. the signal arriving at neuron j from the neuron i
  - $\frac{\partial E_n}{\partial I_j} \equiv \delta_\mathrm{j}$: Different for hidden layers and the output layer

Training Neural Networks: Gradients

- $\delta_k \equiv \dfrac{\partial E_n}{\partial I_k}$ for neuron k in the output layer:

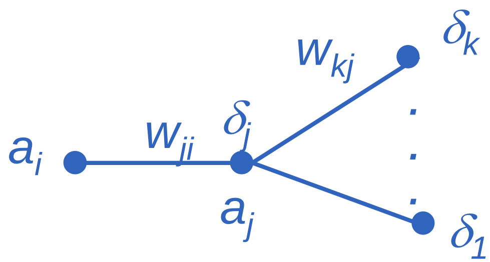
$$\delta_k = [y_{nk}(\mathbf{w}, \mathbf{x}_n) - L_n^{\,k}] \cdot f'(I_k)$$

i.e. δ_k is proportional to the classification error

- $\delta_j \equiv \dfrac{\partial E_n}{\partial I_j}$ for neuron j in a hidden layer:

$$\delta_j = \sum_k \frac{\partial E_n}{\partial I_k} \cdot \frac{\partial I_k}{\partial I_j} = f'(I_j) \cdot \sum_k w_{kj} \cdot \delta_k$$

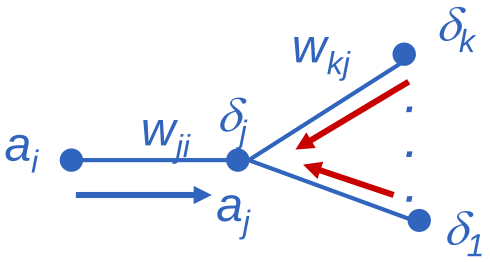
where k is an index running over all units to which neuron j sends an output



Training Neural Networks: Back-propagation

- Back-propagation for computing the gradients:
  - Forward step:
    - Calculate output y_{nk} from $\mathbf{x}_n$ and the current values of $\mathbf{w}$
    - Save the output a_j as well as $f'(l_j)$ in every neuron j
    - The classification error and δ_k is calculated from y_{nk}
  - Actual back-propagation:
    - δ_j is calculated from δ_k and $f'(l_j)$ successively for each layer from δ_j and a_j:

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j \cdot a_j$$



Training Neural Networks: Regularisation

- Gradient descent might lead to overfitting
- Regularisation: weights should not take very large numerical values
- Expansion of the loss function:

$$E(\mathbf{w}) = \underbrace{\frac{1}{2} \cdot \sum_{n,k} \left(y_{nk}(\mathbf{w}, \mathbf{x}_n) - L_n^k \right)^2}_{\text{classification loss}} + \underbrace{\lambda \cdot \sum_{i,j} w_{ij}^2}_{\text{regularisation term}}$$

- This type of regularisation is called “weight decay“ with parameter λ
- Also has to be considered in gradient computation

Training Neural Networks: Momentum

- Gradients from minibatches may be noisy
  - May result in slow convergence, may get stuck in local minima
- Solution: Use **momentum**!
  - Consider "velocity" $\mathbf{v}$ from average change in previous updates
  - Initialisation: $\mathbf{w}^{(0)}$ as discussed earlier, $\mathbf{v}^{(0)} = \mathbf{0}$
  - Update: $\mathbf{v}^{(\tau+1)} = \rho \cdot \mathbf{v}^{(\tau)} + \nabla E(\mathbf{w}^{(\tau)})$

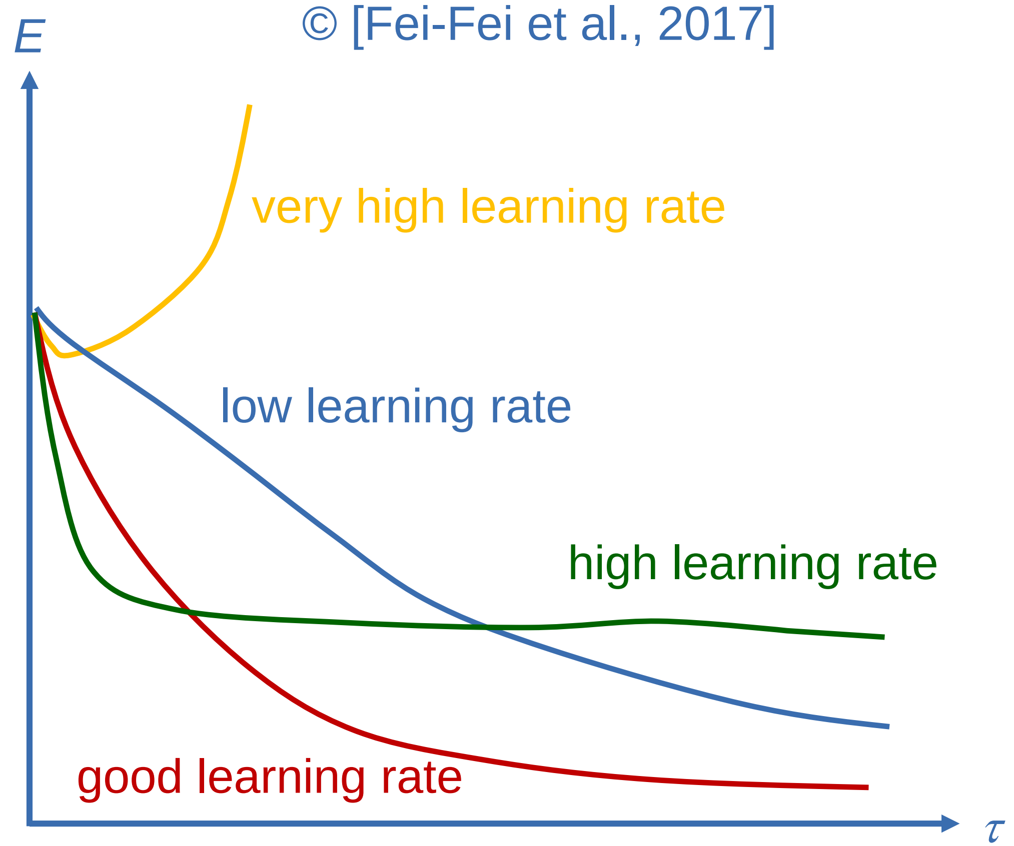
$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \cdot \mathbf{v}^{(\tau+1)}$$

  - *Friction* parameter ρ : e.g. 0.9 or 0.99
  - Faster convergence: Nesterov momentum [Suskever et al., 2013]

Training Neural Networks: Learning rate

- Learning rate η in gradient descent is an important hyperparameter
- Needs to be tuned carefully!
- Good η leads to …
  - Fast convergence
  - Strong minimum of *E*
- Adapt η in the iteration process
- Example: exponential decay with small ε

$$\eta = \eta_0 \cdot (1 - \varepsilon)^{k \cdot \tau}$$



Probabilities

- For multiclass-problems, for each class L^k there is a neuron y_k in the output layer
- The output of y_k is interpreted as the membership value of class L^k
- An interpretation as a posterior probability can be derived if the outputs are normalized

$$P\left(C = L^k \mid \mathbf{x}\right) = \frac{y_k}{\sum_k y_k}$$

Discussion

- Neural networks had gone out of fashion compared to procedures such as SVM or random forests:
  - Networks with few layers: not adaptable enough
  - Networks with many neurons: numerical problems in the determination of the parameters
- Neural networks have come back in the context of "**Deep Learning**"
  - Networks with many layers ("deep" networks), many neurons
  - Sharing of weights → Convolutional Neural Networks (CNN)
  - Improved initialisation and learning
  - Implementation on graphics card (GPU)
  - Availability of large databases of annotated images for training