Practical Course on

Machine Learning in Medical Imaging (2019WiSe)

Neural Networks

Roger David Soberanis Mukul
PhD Candidate

Chair for Computer Aided Medical Procedures
Faculty of Informatics
TU Munich
roger.soberanis@tum.de

Technische Universität München

+

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

About me

General Interests:

- Machine learning for medical applications.
- Deep learning for medical applications
  - Medical image segmentation (pancreas).
  - Localization and Classification in Medical images (polyps, colon, endoscopic sequences)

Supervised Learning and Classification

- Lets suppose we have a set of data points $\boldsymbol{x} \in \Omega \subset \mathbb{R}^n$ from some event, experiment, process, etc.
  - $\boldsymbol{x}$ is a vector representing a particular sample of the process.
  - Ω is the set of all the possible samples.

Supervised Learning and Classification

- Lets suppose we have a set of data points $\boldsymbol{x} \in \Omega \subset \mathbb{R}^n$ from some event, experiment, process, etc.
  - $\boldsymbol{x}$ is a vector representing a particular sample of the process.
  - Ω is the set of all the possible samples.

- In a general supervised learning setting, we can define a function $f: \Omega \rightarrow Y$ relating each sample in Ω with one particular target $\mathrm{y} \in \mathrm{Y}$.

Supervised Learning and Classification

- Lets suppose we have a set of data points $\boldsymbol{x} \in \Omega \subset \mathbb{R}^n$ from some event, experiment, process, etc.
  - $\boldsymbol{x}$ is a vector representing a particular sample of the process.
  - Ω is the set of all the possible samples.

- In a general supervised learning setting, we can define a function $f: \Omega \rightarrow Y$ relating each sample in Ω with one particular target $\mathrm{y} \in \mathrm{Y}$.

- For a classification problem Y is a set of discrete values (e.g. $Y = \{0, 1, 2, \dots, k-1\}$) and $|Y| = k$ the number of classes.

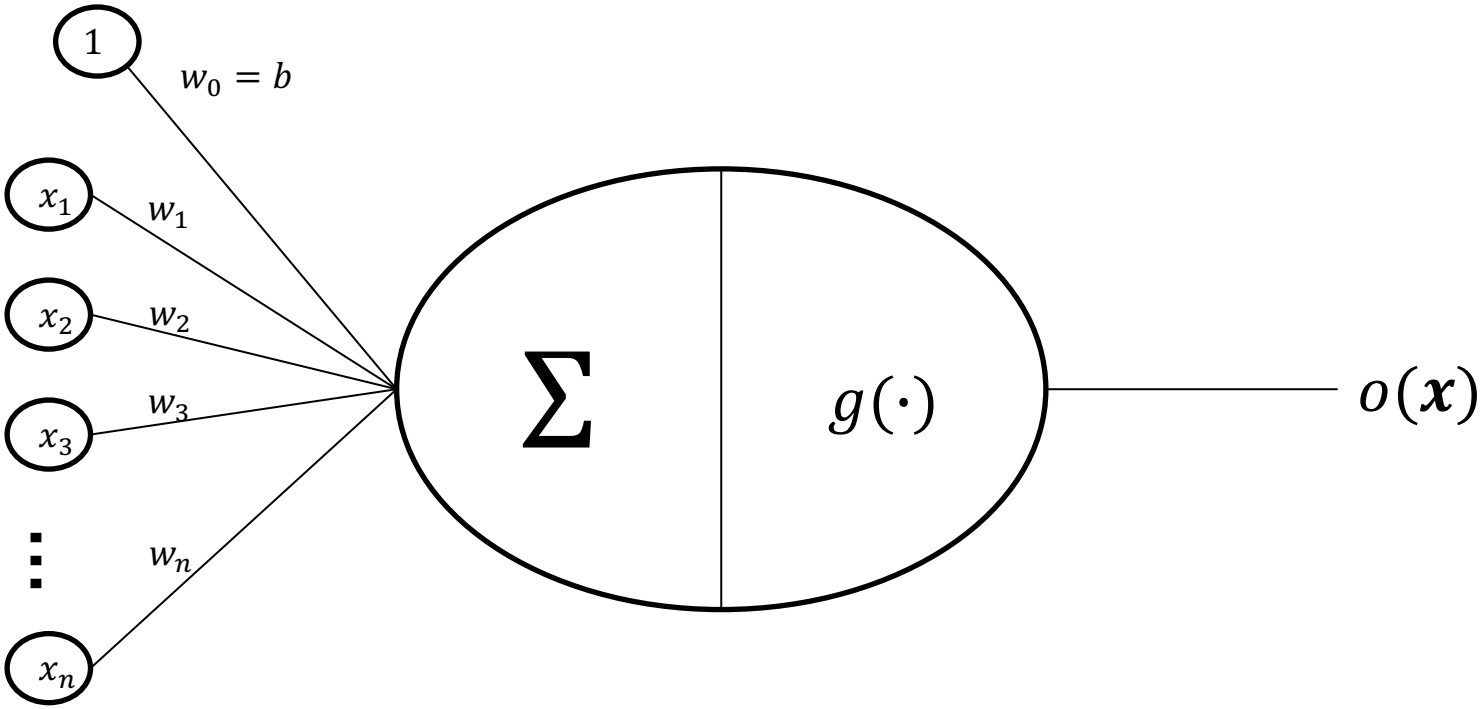
Supervised Learning and Classification

Supervised Learning: Given a training set $T = \{(\boldsymbol{x}_j, y_j) | j = 1, \dots, m,\ \boldsymbol{x}_j \in \Omega,\ y_j \in \mathrm{Y}, f(\boldsymbol{x}_j) = y_j\}$, the problem is to use T to find a function $h: \Omega \to Y$, so h is a "good" approximation of f.

- How to define h (artificial neural networks).
- How to measure how "good" is h (loss function).
- How to use T to find a "good" h (training algorithm).

Artificial Neurons

- An artificial neuron is a unit that takes a set of real-valued inputs, performs some operation $o(\cdot)$ and produces a single real-valued output.



Artificial Neurons: Perceptron

The perceptron neuron is defined by the following function:

$$o(\boldsymbol{x}) = o(x_1, x_2, \dots, x_n) = \begin{cases} 1, & \text{if } \boldsymbol{w}\boldsymbol{x} + b > 0 \\ -1, & \text{otherwise} \end{cases}$$

or

$$o(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}\boldsymbol{x} + b)$$

Where $\boldsymbol{w} = [w_1, w_2, \dots, w_n]$ and b are the weights and bias of the neuron and define a particular instance of the perceptron unit. We can also simplify the notation by writing:

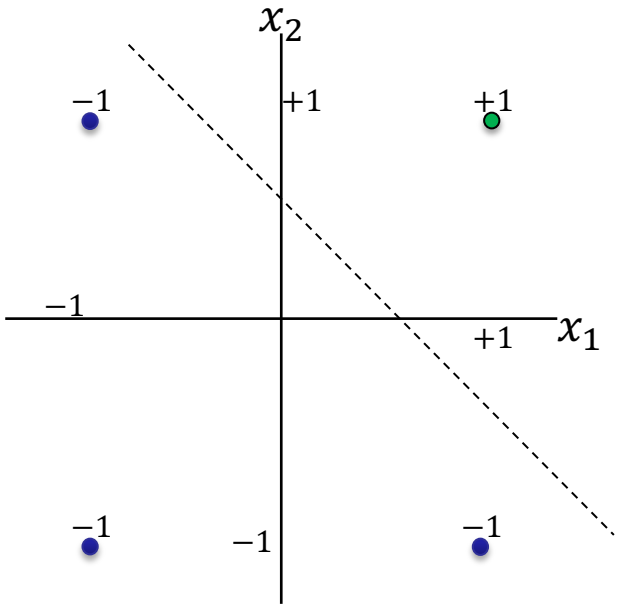
$$o(\boldsymbol{x}) = \text{sgn}(\boldsymbol{w}\boldsymbol{x})$$

with $w_0 = b$ and $x_0 = 1$.

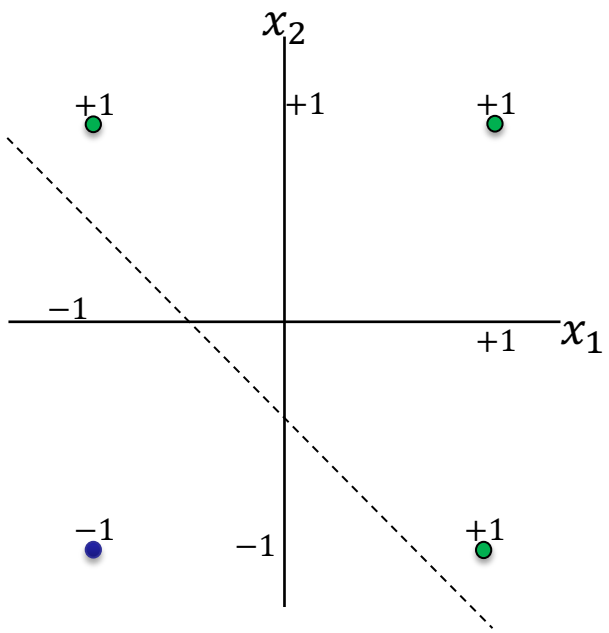
Artificial Neurons: Perceptron

Considering the sets $\Omega = \{-1, 1\}^2$ and $Y = \{-1, 1\}$ and the values -1 and 1 representing the logic values false and true we can use perceptron to represent Boolean functions.

$$o_{\text{and}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} - 0.3)$$



$$o_{\text{or}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} + 0.3)$$



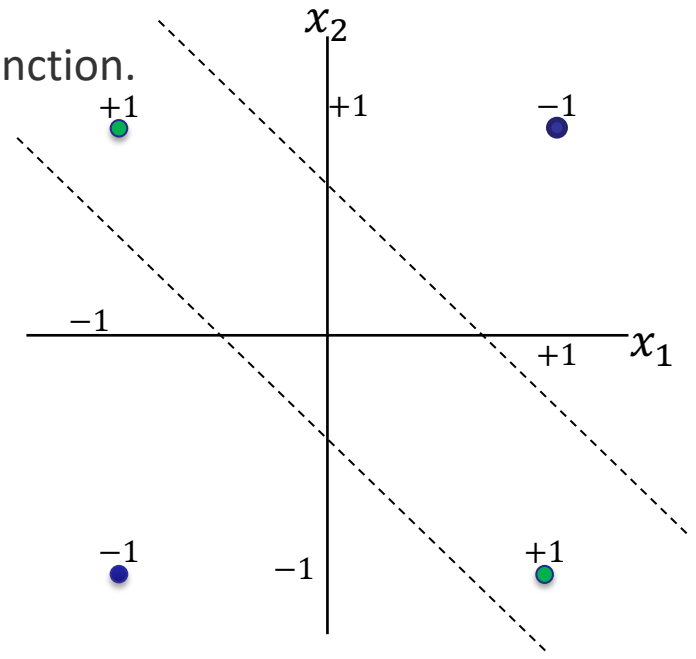
Artificial Neurons: Perceptron

Considering the sets $\Omega = \{-1, 1\}^2$ and $Y = \{-1, 1\}$ and the values -1 and 1 representing the logic values false and true we can use perceptron to represent Boolean functions.

$$o_{\text{and}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} - 0.3)$$

$$o_{\text{or}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} + 0.3)$$

One single unit can not represent the XOR Boolean function.



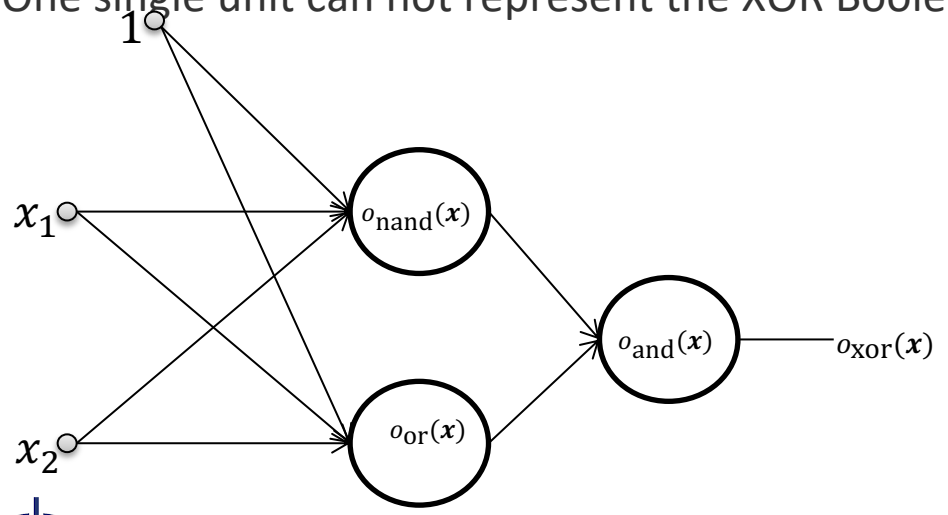
Artificial Neurons: Perceptron

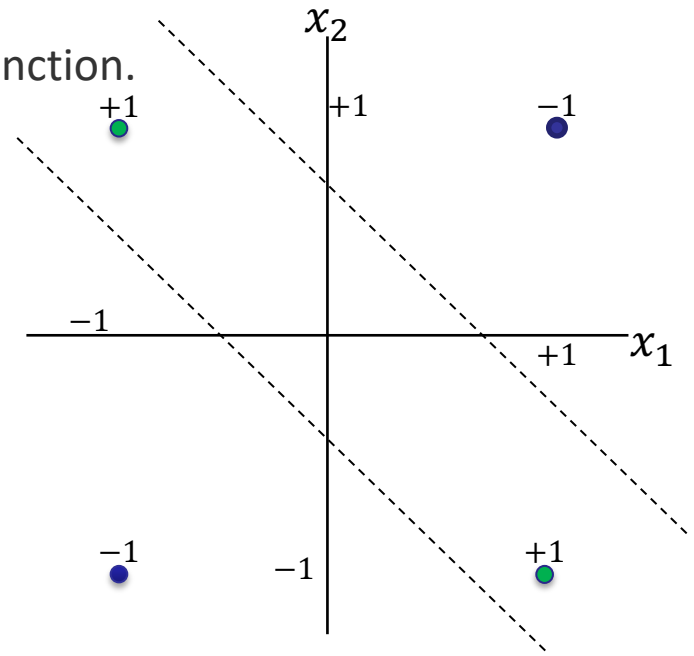
Considering the sets $\Omega = \{-1, 1\}^2$ and $Y = \{-1, 1\}$ and the values -1 and 1 representing the logic values false and true we can use perceptron to represent Boolean functions.

$$o_{\text{and}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} - 0.3)$$

$$o_{\text{or}}(\boldsymbol{x}) = \text{sgn}([0.5, 0.5]\boldsymbol{x} + 0.3)$$

One single unit can not represent the XOR Boolean function.





Learning Perceptron's Weights

Learning the weights for one single perceptron.

- Randomly initialize all the weights.
- Iteratively update the weights according to:

$$w_i = w_i + \alpha\big(y - o(\boldsymbol{x})\big)x_i \qquad \text{(perceptron rule)}$$

For a particular input $\boldsymbol{x}$ with target (or class) $y \in \{-1, 1\}$ and with α the learning rate.

Works well with linearly separable data.

Learning Perceptron's Weights

Learning the weights for one single perceptron using gradient descent.

- We will consider units with no thresholding (linear units): $o(\boldsymbol{x}) = \boldsymbol{w}\boldsymbol{x}$
- Loss function (e.g.): $E(\boldsymbol{w}) = \frac{1}{2}\sum_{j=1\ldots,m}[y_j - o(\boldsymbol{x}_j)]^2$
- The weights will be updated according its gradient (moving in the opposite direction):

$$w_i = w_i - \alpha \frac{\partial E}{\partial w_i}$$

$$w_i = w_i + \alpha \sum_{j=1\ldots,m} [y_j - o(\boldsymbol{x}_j)] x_{ij}$$

With x_{ij} the component x_i of input example $\boldsymbol{x}_j$.

Learning Multilayer Networks: Backpropagation

- Differentiable non linear inputs, e.g. sigmoid units.

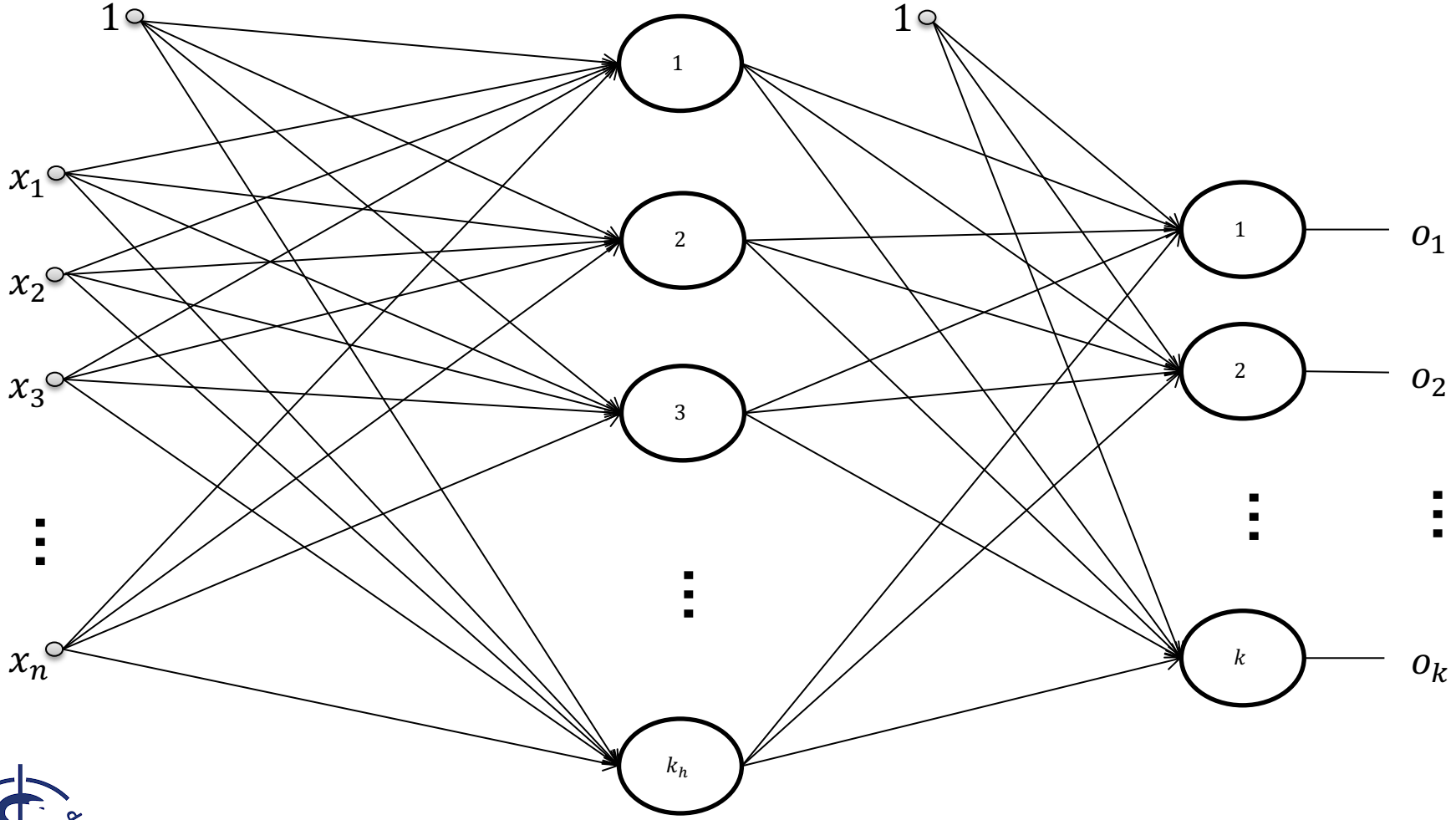
$$o(\boldsymbol{x}) = \sigma(\boldsymbol{wx}) = \frac{1}{1 + e^{-\boldsymbol{wx}}}$$

- Loss function considering k outputs units $[o_0, \dots, o_{k-1}]$, e.g.

$$E(\boldsymbol{w}) = \frac{1}{2} \sum_{j=1\dots,m-1} \sum_{i=1,\dots,k} [\boldsymbol{y}_{ij} - o_{ij}]^2$$

Where y_{ij} is the component y_i or the target vector $\boldsymbol{y}_j = [y_0, y_1, \dots, y_{k-1}]$ and $o_i(\boldsymbol{x_j})$ is the component o_i of the final output vector of the network when evaluated on the sample $\boldsymbol{x}_j$.

Learning Multilayer Networks: Backpropagation



Learning Multilayer Networks: Backpropagation

For a network with one hidden layer with k_h neurons and an output layers with k units:

- Initialize the weights to small random numbers.
- Do until termination conditions is met:
  - For each training example $(\boldsymbol{x}, \boldsymbol{y})$:
    - Propagate the input forward through the network (evaluate the output for every neuron).
    - Propagate the errors backward through the network:

    1. Compute the error term δ_r for each output unit o_r:

$$\delta_r = o_r(1 - o_r)(y_r - o_r)$$

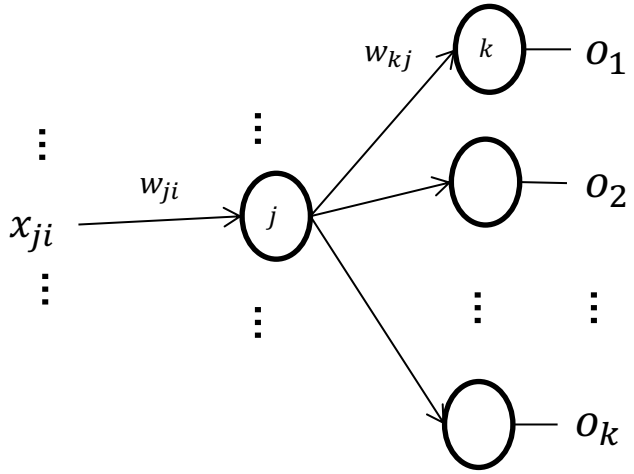
    2. Compute the error δ_h for each hidden unit o_h:

$$\delta_h = o_h(1 - o_h) \sum_{r=1,\ldots,k} w_{rh}\delta_r$$

    3. Update the weighs: $w_{ji} = w_{ji} + \alpha\delta_j x_{ji}$, with x_{ji} and w_{ji} the ith input component to unit j and its corresponding weight, respectively.

Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

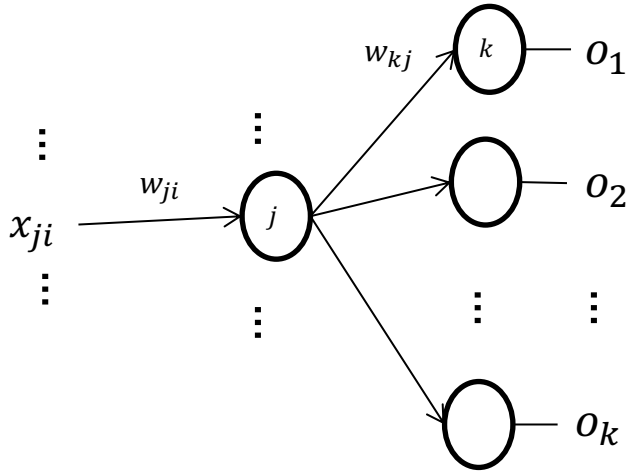


Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}}$$

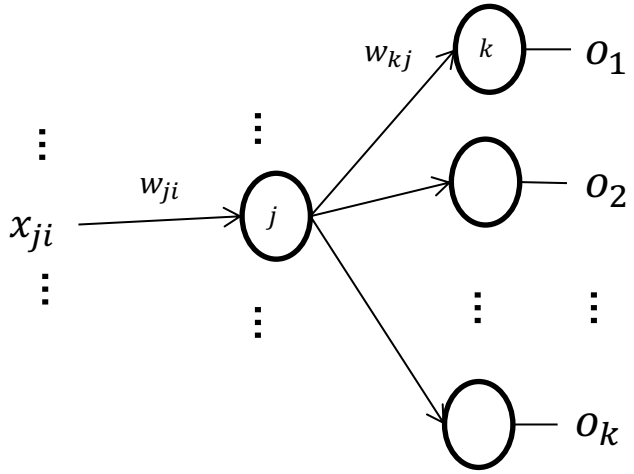


Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}}$$

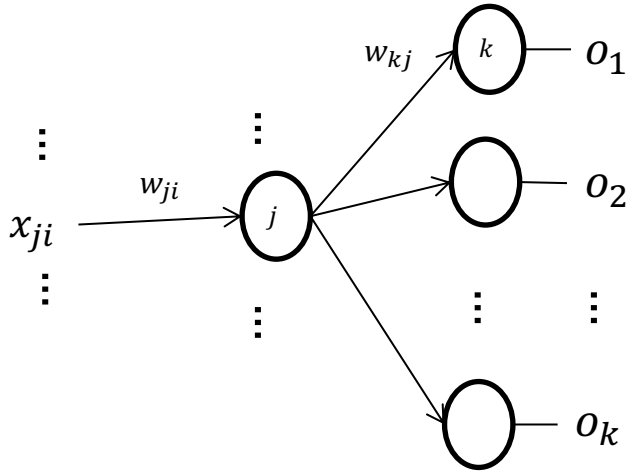


Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} x_{ji}$$



Learning Multilayer Networks: Backpropagation

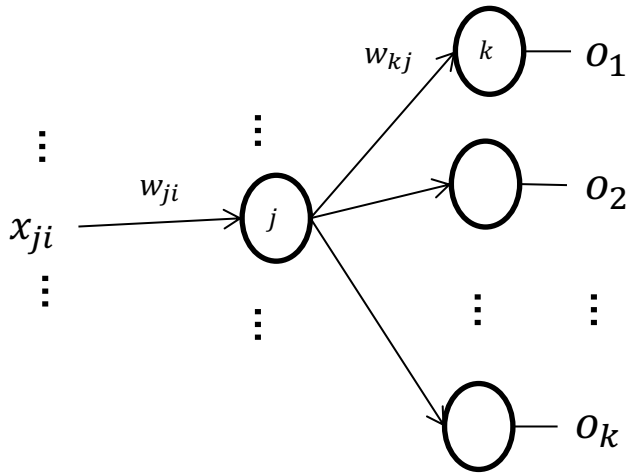
- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} x_{ji}$$

For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j}$$



Learning Multilayer Networks: Backpropagation

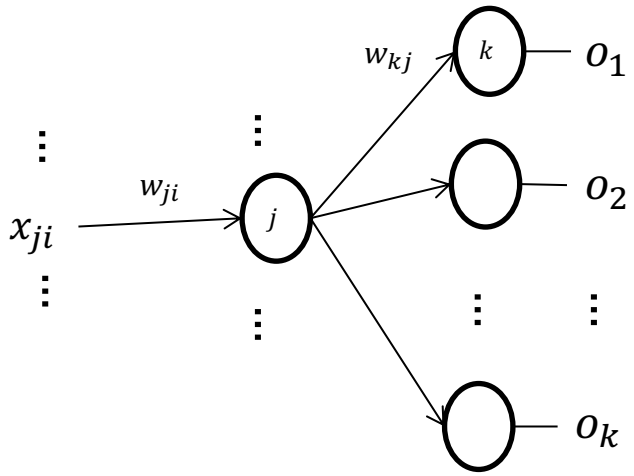
- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j} x_{ji}$$

For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = \frac{\partial E}{\partial g_j} g_j'$$



Learning Multilayer Networks: Backpropagation

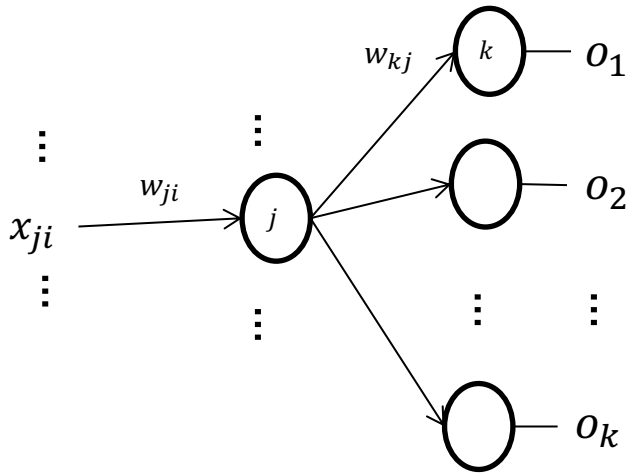
- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}x_{ji}$$

For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = \frac{\partial E}{\partial g_j}g_j' = g_j'\frac{\partial E_j}{\partial o_j}$$



Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

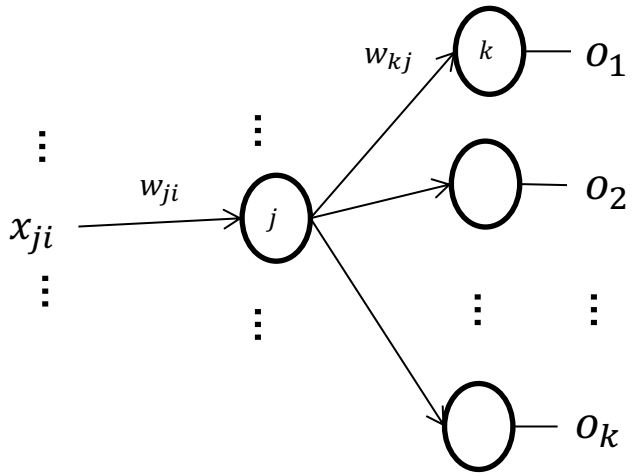
$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}x_{ji}$$

For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = \frac{\partial E}{\partial g_j}g_j' = g_j'\frac{\partial E_j}{\partial o_j}$$

For a hidden unit, we need to consider all the neurons in the next layer:

$$\frac{\partial E}{\partial g_j} = \sum_{k\in \mathrm{nxt}(j)} \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial g_j}$$



Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

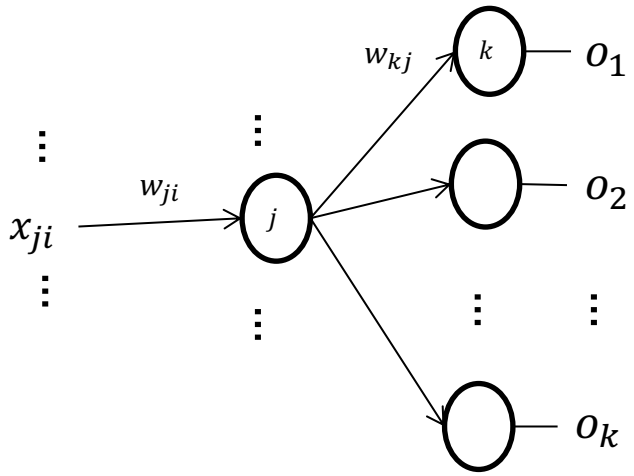
$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}x_{ji}$$

For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = \frac{\partial E}{\partial g_j}g_j' = g_j'\frac{\partial E_j}{\partial o_j}$$

For a hidden unit, we need to consider all the neurons in the next layer:

$$\frac{\partial E}{\partial g_j} = \sum_{k\in \mathrm{nxt}(j)}\frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial g_j} = \sum_{k\in \mathrm{nxt}(j)}\frac{\partial E}{\partial z_k}w_{kj}$$



Learning Multilayer Networks: Backpropagation

- Consider:
  - x_{ji} the input i to unit j.
  - w_{ji} the weight of unit j to input i
  - $z_j = \boldsymbol{w_j x_j}$ the weighted sum of inputs.
  - $g_j = g(z_j)$ the activation function of unit j.

In order to update the weights, we need to compute the gradients

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}x_{ji}$$

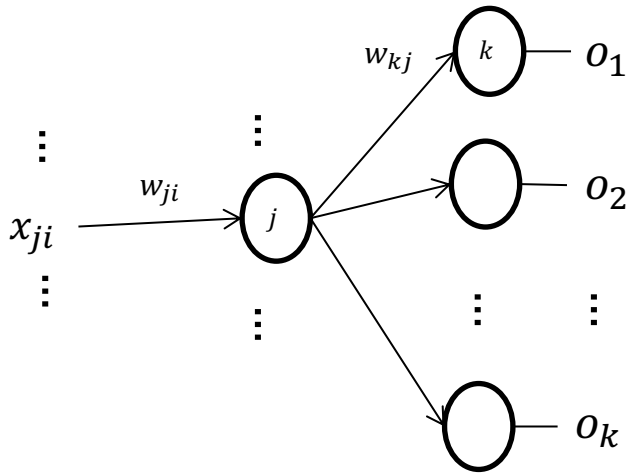
For a output unit:

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial g_j}\frac{\partial g_j}{\partial z_j} = \frac{\partial E}{\partial g_j}g_j' = g_j'\frac{\partial E_j}{\partial o_j}$$

For a hidden unit, we need to consider all the neurons in the next layer:

$$\frac{\partial E}{\partial g_j} = \sum_{k\in \mathrm{nxt}(j)} \frac{\partial E}{\partial z_k}\frac{\partial z_k}{\partial g_j} = \sum_{k\in \mathrm{nxt}(j)} \frac{\partial E}{\partial z_k} w_{kj}$$

$$\frac{\partial E}{\partial z_j} = g_j' \sum_{k\in \mathrm{nxt}(j)} w_{kj}\frac{\partial E}{\partial z_k}$$



Learning Multilayer Networks: Backpropagation

- The backpropagation algorithm does not guarantee to converge to a local maxima, but does guarantee to converge at least to a local minima.

Learning Multilayer Networks: Backpropagation

- The backpropagation algorithm does not guarantee to converge to a local maxima, but does guarantee to converge at least to a local minima.

- Adding momentum in the the weight update rule could help in this problem.

$$\Delta w_{ji}(n) = \alpha \delta_j x_{ji} + \beta \Delta w_{ji}(n-1)$$
$$w_{ji} = w_{ji} + \Delta w_{ji}(n)$$

With $\Delta w_{ji}(n)$ the weight update in the nth learning iteration.

Learning Multilayer Networks: Backpropagation

- The backpropagation algorithm does not guarantee to converge to a global minima, but does guarantee to converge to a local minima.

- Adding momentum in the the weight update rule could help in this problem.

$$\Delta w_{ji}(n) = \alpha \delta_j x_{ji} + \beta \Delta w_{ji}(n-1)$$
$$w_{ji} = w_{ji} + \Delta w_{ji}(n)$$

With $\Delta w_{ji}(n)$ the weight update in the nth learning iteration.

- In order to avoid overfitting we can add a regularization term to the loss function (not the only option, e.g. dropout)

$$\tilde{E}(\boldsymbol{w}) = E(\boldsymbol{w}) + \lambda \sum_{i,j} w_{ji}^2$$

Learning Multilayer Networks: Activation and Loss

- The next are common activation functions for use in multilayered neural networks
- Sigmoid unit.

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

- Rectified Linear Unit

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

- Softmax output layer

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{c=1,\ldots k} e^{z_c}}$$

Learning Multilayer Networks: Activation and Loss

- The next are common loss functions used in multilayered neural networks

- MSE Loss

$$E = \frac{1}{2} \sum_{j=1\ldots,m} [y_j - H(\boldsymbol{x}_j)]^2$$

- Cross Entropy Loss

$$E = - \sum_{j=1\ldots,m} y_j \log\left(H(\boldsymbol{x}_j)\right)$$

References

- Tom M. Mitchel. *Machine Learning*. McGraw-Hill Science/Engineering/Math; 1999.
- Srivastava, Hinton, Krizhevsky, Sutskever, and Salakhutdinov. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014
- Nair and Hinton. *Rectified Linear Units Improve Restricted Boltzmann Machines*. Proceedings of the 27th International Conference on International Conference on Machine Learning. 2010.
- Goh. *Why Momentum Really Works*. Distill, 2017. https://distill.pub/2017/momentum/