CSCE 633: Machine Learning

Lecture 7

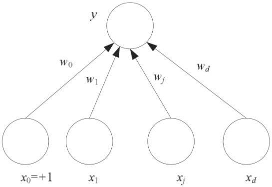
Overview

- Perceptron
  - Representation
  - Learning
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Overview

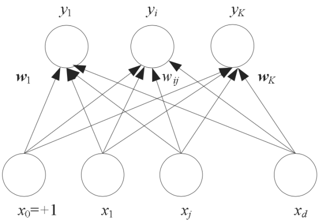
- Perceptron
  - Representation
  - Training
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Perceptron: Basic processing unit



- Inputs $x_d \in \mathbb{R}$, $d = 1, \ldots, D$
  - might come from the environment
  - might be the output of other perceptrons
- Associated with a connection weight $w_d \in \mathbb{R}$, $d = 1, \ldots, D$
- Output is some function of the linear combination of inputs
  - $y = s\left(\sum_{j=1}^{D} w_d x_d + w_0\right) = s(\mathbf{w}^T \mathbf{x})$
    where $s(\alpha) = 1$, if $\alpha > 0$, $s(\alpha) = 0$, otherwise
    e.g. sigmoid activation: $s(\mathbf{x}, \mathbf{w}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$
- can be used for classification, i.e. choose C_1, if $s(\alpha) > 0.5$

Perceptron: Basic processing unit



- Multiclass: $K > 2$ outputs
  - $y_k = s\left(\sum_{d=1}^{D} w_{kd} x_d + w_{k0}\right) = s(\mathbf{w_k}^T \mathbf{x})$
    where w_{kj} is the weight from input x_j to output y_k
    e.g. $s(\mathbf{x}, \mathbf{w_1}, \ldots, \mathbf{w_K}) = \frac{\exp(\mathbf{w_k}^T \mathbf{x})}{1+\sum_{k=1}^{K} \exp(\mathbf{w_k}^T \mathbf{x})}$
  - 0/1 encoding for output vector
    - e.g. in a 4-class problem: if class=3, then $y = [0, 0, 1, 0]$

Overview

- Perceptron
  - Representation
  - Training
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Perceptron: Training

Online training

- Cost-efficient (computationally and memory-wise)
- Nature of data can change over time
- Error function expressed in terms of individual samples
- Weight update performed after each instance is seen

Perceptron: Training

Online training

- Evaluation: cross-entropy function for 1 instance $(\mathbf{x_n}, y_n)$
  $\mathcal{E}(\mathbf{w}) = -y_n \log \left[\sigma(\mathbf{w}^T \mathbf{x_n})\right] - (1 - y_n) \log \left[1 - \sigma(\mathbf{w}^T \mathbf{x_n})\right]$
  $\mathcal{E}(\mathbf{w_1}, \ldots, \mathbf{w_K}) = -\sum_{k=1}^{K} y_{nk} \log p(y_{nk} = 1 | \mathbf{w_1}, \ldots \mathbf{w_K})$
- Optimization: gradient descent
  $\frac{\vartheta \mathcal{E}(\mathbf{w})}{\vartheta w_d} = \left(\sigma(\mathbf{w}^T \mathbf{x_n}) - y_n\right) x_{nd}$
  $\frac{\vartheta \mathcal{E}(\mathbf{w})}{\vartheta w_{kd}} = \left(\sigma(\mathbf{w}^T \mathbf{x_n}) - y_{nk}\right) x_{nd}$

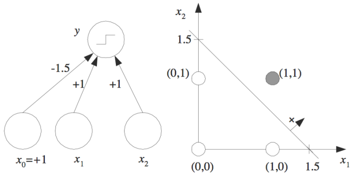
Overview

- Perceptron
  - Representation
  - Training
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Approximating linear functions

Example: Boolean AND

x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1



Example of a perceptron implementing AND

$y = s(x_1 + x_2 - 1.5)$

$\mathbf{w} = [-1.5\ 1\ 1]^T$

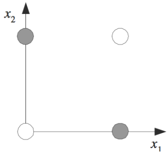
$\mathbf{x} = [1\ x_1\ x_2]^T$

The above weights were empirically selected, but we could have also learned them through gradient descent

Approximating linear functions

Example: Boolean XOR

x_1	x_2	r
0	0	0
0	1	1
1	0	1
1	1	0



Not linearly separable

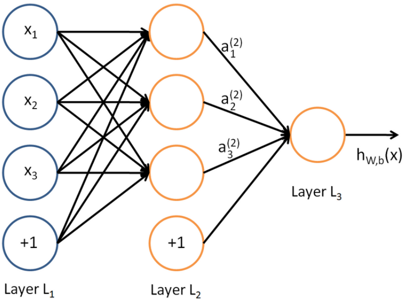
Need combination of more than one perceptrons $\rightarrow$ multilayer perceptrons

Overview

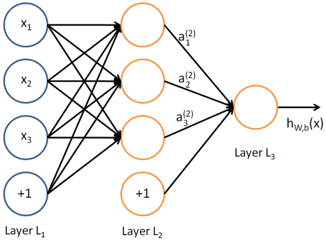
- Perceptron
  - Representation
  - Learning
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Multilayer Perceptron

- Type of feedforward neural network
- Can model non-linear associations
- “Multi-level combination” of many perceptrons



Multilayer Perceptron: Representation



$$\alpha_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$\alpha_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$\alpha_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \alpha_1^{(3)} =$$
$$f(W_{11}^{(2)}\alpha_1^{(2)} + W_{12}^{(2)}\alpha_2^{(2)} + W_{13}^{(2)}\alpha_3^{(2)} + b_1^{(2)})$$

Terminology

$W_{ij}^{(l)}$: connection between unit j in layer l to unit i in layer $l+1$

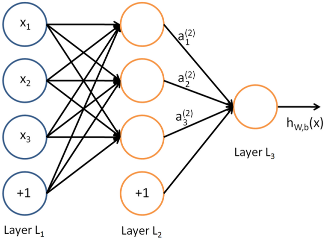
$\alpha_i^{(l)}$: activation of unit i in layer l

$b_i^{(l)}$: bias connected with unit i in layer $l+1$

Forward propagation: The process of propagating the input to the output through the activation of inputs and hidden units to each node

Multilayer Perceptron: Representation

Matrix notation



$$\boldsymbol{\alpha}^{(2)} = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) = \alpha^{(3)} = f(\mathbf{W}^{(2)}\boldsymbol{\alpha}^{(2)} + \mathbf{b}^{(2)})$$

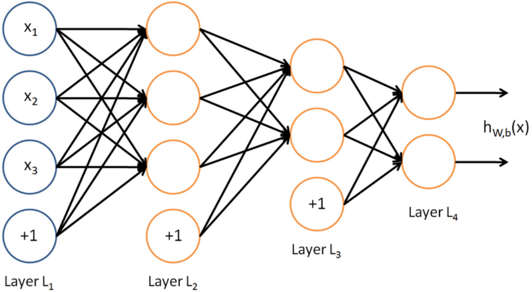
$$\mathbf{W}^{(1)} = \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \\ W_{31}^{(1)} & W_{32}^{(1)} & W_{33}^{(1)} \end{bmatrix}, \mathbf{b}^{(1)} = [b_1^{(1)}\ b_2^{(1)}\ b_3^{(1)}], \text{ etc.}$$

Multilayer Perceptron: Representation

Alternative architectures

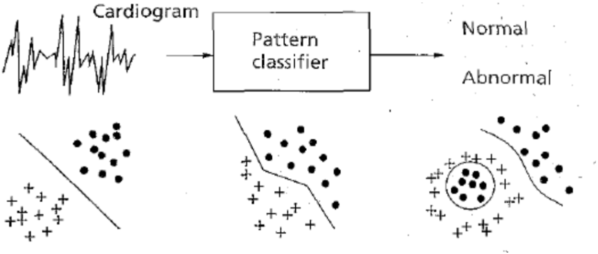
2 hidden layers, multiple output units

e.g. medical diagnosis: different outputs might indicate presence or absence of different diseases



Multilayer Perceptron

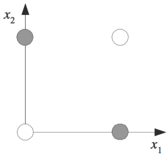
Non-linear feature learning

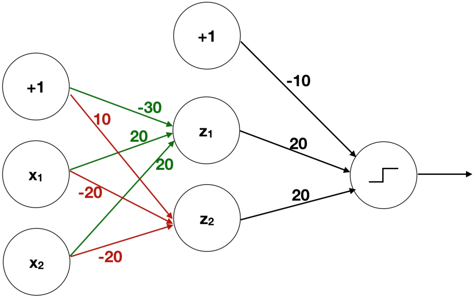


Multilayer Perceptron: Approximating non-linear functions

Example: Boolean XOR with multilayer perceptrons

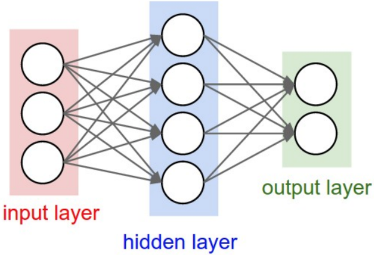
x_1	x_2	z_1	z_2	r
0	0	0	1	1
0	1	0	0	0
1	0	0	0	0
1	1	1	0	1





Multilayer Perceptron

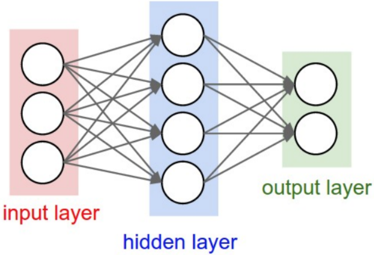
Question: How many parameters does this network have to learn?



A) 20
B) 26
C) 6
D) 12

Multilayer Perceptron

Question: How many parameters does this network have to learn?



A) 20
B) 26
C) 6
D) 12

The correct answer is B

[3 x 4] + [4 x 2] = 20 weights, 4 + 2 = 6 biases

Overview

- Perceptron
  - Representation
  - Learning
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Backpropagation

Multilayer Perceptron: Representation

- Input: $\mathbf{x} \in \mathbb{R}^D$
- Output:
  $y \in \{0, 1\}$ or $y \in \{1, \ldots, K\}$ (classification)
  $y \in \mathbb{R}$ or $y \in \mathbb{R}^K$ (regression)
- Training data: $\mathcal{D}^{train} = \{(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_N}, y_N)\}$
- Model: $h_{\mathbf{W},\mathbf{b}}(\mathbf{x})$
  represented through forward propagation (see previous slides)
- Model parameters: weights $\mathbf{W^{(1)}}, \ldots, \mathbf{W^{(L)}}$ and biases $\mathbf{b^{(1)}}, \ldots, \mathbf{b^{(L)}}$

Multilayer Perceptron: Evaluation criterion

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = \frac{1}{2} \| h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) - y \|_2^2$ (regression)

$J(\mathbf{W}, \mathbf{b}, \mathcal{D}^{train}) = y \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x}) + (1 - y) \log(1 - h_{\mathbf{W},\mathbf{b}}(\mathbf{x}))$ (classification)

Backpropagation

Multilayer Perceptron: Evaluation criterion

Regression

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M} \frac{1}{2}\|h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n\|_2^2 + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

s_l: # nodes in l^{th} layer

Classification

$$J(\mathbf{W},\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{M}(y_n \log h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) + (1-y_n)\log(1-h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n})))$$
$$+ \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$$

We will perform gradient descent

Backpropagation

Gradient descent for regression

$$J(\mathbf{W}, \mathbf{b}) = \frac{1}{N} \sum_{n=1}^{M} \frac{1}{2} \| h_{\mathbf{W},\mathbf{b}}(\mathbf{x_n}) - y_n \|_2^2 + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2$$

$$W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$$

$$b_i^{(l)} := b_i^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$$

Note: Initialize the parameters randomly $\rightarrow$ symmetry breaking

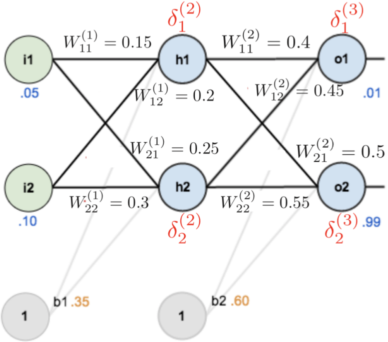
Use backpropagation to compute partial derivatives $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$ and $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

Backpropagation

Intuition

- Given a training example $(\mathbf{x_n}, y_n)$, we run a "forward pass" to compute all the activations
- For each node i in layer l, we compute an error term $\delta_i^{(l)}$ that measures how much that node was "responsible" for any errors in the output
    - Output node: difference between activation and target value
    - Hidden nodes: weighted average of the error terms of the nodes from the previous layer (i.e. $l+1$)

Backpropagation



Backpropagation Implementation

- For each node i in output layer L
  - $\delta_i^{(L)} = (\alpha_i^{(L)} - y_n) f'(z_i^{(L)})$
- For each node i in layer $l = L-1, L-2, \ldots, 2$
  - Hidden nodes: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$
- Compute the desired partial derivatives as:

  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}} = \alpha_j^{(l)} \delta_i^{(l+1)}$

  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}} = \delta_i^{(l+1)}$
- Update the weights as:

  $W_{ij}^{(l)} := W_{ij}^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}}$

  $b_i^{(l)} := b_i^{(l)} - \alpha \frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}}$

[Detailed solution of example in Handouts]

Backpropagation

Implementation

- Given a training example $(\mathbf{x_n}, y_n)$, we run a "forward pass" to compute all the activations
- For each node i in output layer L
  - $\delta_i^{(L)} = (y_n - \alpha_i^{(L)}) f'(z_i^{(L)})$
- For each node i in layer $l = L-1, L-2, \ldots, 2$
  - Hidden nodes: $\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$
- Compute the desired partial derivatives as:
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta W_{ij}^{(l)}} = \alpha_j^{(l)} \delta_i^{(l+1)}$
  $\frac{\vartheta J(\mathbf{W},\mathbf{b})}{\vartheta b_i^{(l)}} = \delta_i^{(l+1)}$

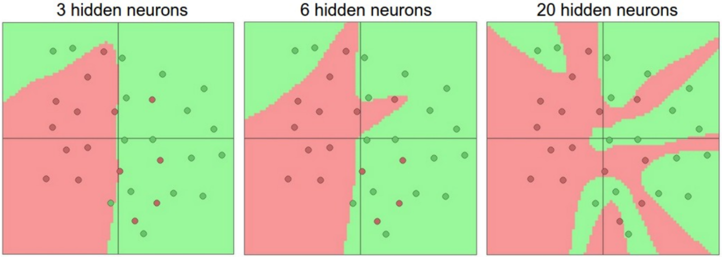
Overview

- Perceptron
  - Representation
  - Learning
  - Examples
- Multilayer Perceptron
  - Representation
  - Learning: Backpropagation
  - Practical issues
  - Activation Function

Determining number of layers and their sizes

Implementation

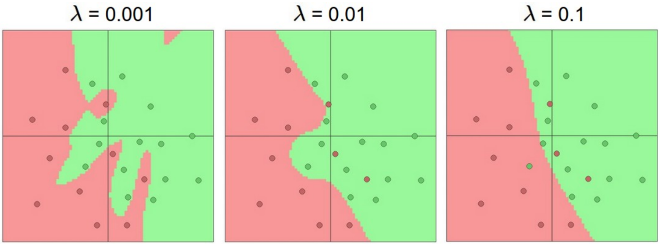
- The capacity of the network (i.e. the number of representable functions) increases as we increase the number of layers
- How to avoid overfittting?



Determining number of layers and their sizes

How to avoid overfitting

- Limit # layers and #hidden units per layers
- Early stopping: start with small weights and stop learning early
- Weight decay: penalize large weights (regularization)
- Noise: add noise to the weights



The effects of regularization strength: Each neural network above has 20 hidden neurons, but changing the regularization strength makes its final decision regions smoother with a higher regularization. You can play with these examples in this ConvNetsJS demo.

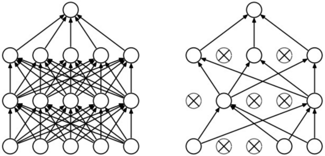
5

http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

Determining number of layers and their sizes

How to avoid overfitting

- An alternative method that complements the above is dropout
- While training, dropout keeps a neuron active with some probability p (a hyperparameter), or sets it to zero otherwise



(a) Standard Neural Net (b) After applying dropout.

https://machinelearningmastery.com/dropout-for-regularizing-deep-neural-networks/

Determining number of layers and their sizes

How to chose the number of layers and nodes

- No general rule of thumb, this depends on:
    - Amount of training data available
    - Complexity of the function that is trying to be learned
    - Number of input and output nodes
- If data is linearly separable, you don't need any hidden layers at all
- Start with one layer and hidden nodes proportional to input size
- Gradually increase

Overview

- Perceptron
    - Representation
    - Learning
    - Examples
- Multilayer Perceptron
    - Representation
    - Learning: Backpropagation
    - Activation Function

Activation Function

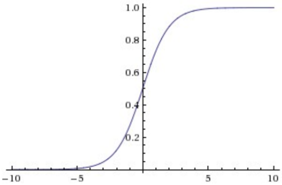
Transforms the activation level of a node (weighted sum of inputs) to an output signal

- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $s(x) = \tanh(x) = 2\sigma(2x) - 1$
- Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$
- Leaky ReLU: $f(x) = (ax) \cdot \mathbb{I}(x < 0) + (x) \cdot \mathbb{I}(x \geq 0)$ (e.g. $a = 0.01$)

Activation Function

Sigmoid: $s(x) = \frac{1}{1+e^{-x}}$

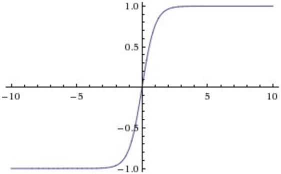
- Transforms a real-valued number between 0 and 1
- Large negative numbers become 0 (not firing at all)
- Large positive numbers become 1 (fully-saturated firing)
- Used historically because of its nice interpretation
- Saturates gradients: The gradient at either extremes (0 or 1) is almost zero, “killing” the signal will flow
- Non-zero centered output: Can be problematic during training, since it can bias outputs toward being always positive or always negative, causing unnecessary oscillations during the optimization



Activation Function

Hyperbolic tangent: $s(x) = \tanh(x) = 2\sigma(2x) - 1$

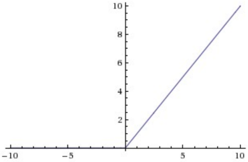
- Scaled version of sigmoid
- Transforms a real-valued number between -1 and 1
- Saturates gradients: Similar to sigmoid
- Output is zero-centered, avoiding some oscillation issues

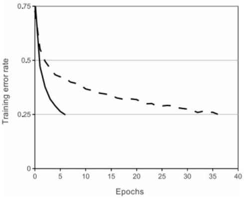


Activation Function

Rectified Linear Unit (ReLU): $f(x) = \max(0, x)$

- Activation simply thresholded at zero
- Very popular during the last years
- Accelerates convergence (e.g. a factor of 6, see bellow) compared to the sigmoid/tanh (due to its linear, non-saturating form)
- Cheap implementation by simply thresholding at zero
- Activation can "die": a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any datapoint again, proper adjustment of learning rate can mitigate that

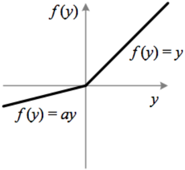




Activation Function

Leaky ReLU: $f(x) = (ax) \cdot \mathbb{I}(x < 0) + (x) \cdot \mathbb{I}(x \geq 0)$

- Instead of the function being zero when $x < 0$, leaky ReLU will have a small negative slope (e.g. $a = 0.01$)
- Some successful results, but not always consistent



What have we learnt so far

- Perceptrons are the basic processing unit of neural networks
- Simulate the "neural connectivity"
- Implemented by the linear combination of input features followed by an activation function, e.g. sigmoid
- Online learning
  - updating weights based on one sample at a time
- Examples implementing boolean functions
  - XOR: non-linear $\rightarrow$ impossible to implement with single perceptron

What have we learnt so far

- Multilayer perceptron is the basic feedforward neural network
- Hidden nodes simulate non-linear associations
- Backpropagation to find network weights
- Different activation functions
- Readings: Alpaydin 11.1-11.8.2