CS-E4710 Machine Learning: Supervised Methods

Lecture 7: Neural networks

Juho Rousu

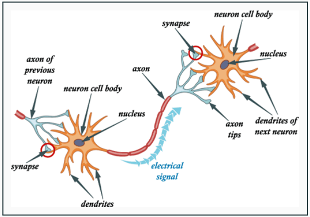
27. October, 2020

Department of Computer Science
Aalto University

Neural networks as models of the brain

Neural networks take inspiration from the human brain, with artificial neurons as computation units and edges between the units model the synapses

- However, compared to artificial neural networks, human brain has huge number of neurons (10^{11}) and synapses (10^{5})
- Each neuron is believed to operate a 'clock speed' of only 100Hz whereas computers work at clock speeds of a few Gigaherz.



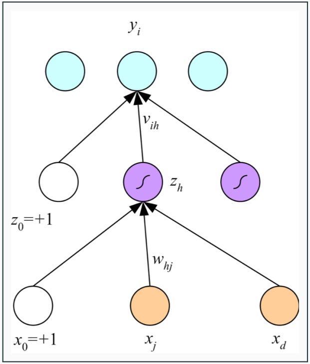
Source: https://pulpbits.net/

Multi-layer perceptrons

Multi-layer perceptrons

Multi-layer perceptron is a neural network that combines several perceptrons to achieve non-linear modelling

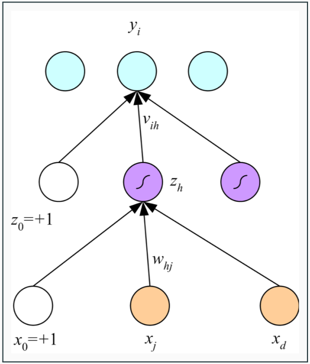
- Multi-layer perceptrons implement a layered network structure:
  - input layer
  - one or more hidden layers
  - output layer
- Nodes in adjacent layers are connected through weighted edges
- The output of each node is fed through activation functions that is typically non-linear



Multilayer perceptron

The output of a two-layer MLP is computed as follows

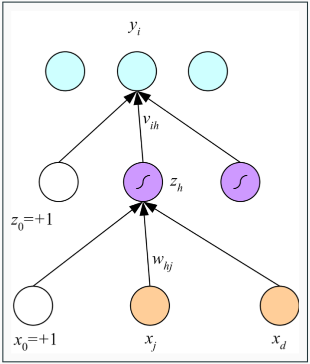
- Input $\mathbf{x}$, augmented by the constant $x_0 = 1$ is fed to the input layer
- The input values are fed to a perceptron unit h in the hidden layer, which computes the linear model $\mathbf{w}_h^T\mathbf{x}$
- An activation function σ_h (e.g. the logistic function) is then applied to obtain the activation level of the hidden unit $z_h = \sigma_h(\mathbf{w}_h^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}_h^T\mathbf{x})}$



Multilayer perceptron

- The values z_h from the hidden units are fed to the output layer through another linear model: $\mathbf{v}_i^T\mathbf{z}$
- A activation function is again computed $y_i = \sigma_i(\mathbf{v}_i^T\mathbf{z})$
- Thus, as a whole, the output y_i is the value of the function

$$y_i = \sigma_i(\mathbf{v}_i^T \left(\sigma_h(\mathbf{w}_h^T\mathbf{x})\right)_{h=1}^{H})$$



Activation functions

Each neuron v_h computes an activation function σ, which can be for example:

- Linear function (used in the output layer for regression): $\sigma(\mathbf{w}^T\mathbf{x}) = \mathbf{w}^T\mathbf{x}$
- The sign function:$\sigma(\mathbf{w}^T\mathbf{x}) = \text{sgn}\left(\mathbf{w}^T\mathbf{x}\right)$
- A threshold function (also called the rectified linear unit, ReLU): $\sigma(\mathbf{w}^T\mathbf{x}) = \begin{cases} \mathbf{w}^T\mathbf{x} & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \textit{otherwise} \end{cases}$
- Logistic function: $\sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})} \in [0, 1]$
- Hyperbolic tangent (another sigmoid function that outputs values between -1 and $+1$): $\sigma(\mathbf{w}^T\mathbf{x}) = \tanh \mathbf{w}^T\mathbf{x} = \frac{e^{\mathbf{w}^T\mathbf{x}} - e^{-\mathbf{w}^T\mathbf{x}}}{e^{\mathbf{w}^T\mathbf{x}} + e^{-\mathbf{w}^T\mathbf{x}}} \in [-1, +1]$

Why do we need non-linear activation functions?

- Consider having two layer network with first layer computing $z_h = \sum_j w_{hj} x_j$ and the second layer computing $y_i = \sum_j v_{ih} z_h$
- The total function is thus:

$$y_i = \sum_h v_{ih} \sum_j w_{hj} x_j = \sum_j \sum_h v_{ih} w_{hj} x_j$$

- We can compute the same with a linear function:

$$y_i = \sum_j u_{ij} x_j$$

  where $u_{ij} = \sum_h v_{ih} w_{hj}$
- Thus there is no real non-linearity in the model and our model reduces to learning a linear hyperplane

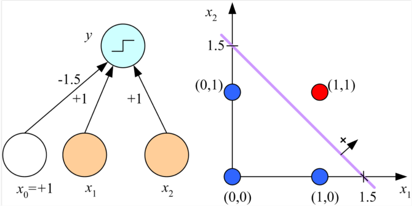
To make the network structure useful, we need non-linear activation functions

Expressive power of neural networks

Computing Boolean AND with the perceptron

- Perceptron can compute the Boolean AND function as follows
- Set the bias $w_0 = -1.5$ and the weights $w_1 = w_2 = 1$
- Now the function $w_1 x_1 + w_2 x_2 + w_0 > 0$ if and only if $x_1 = x_2 = 1$
- The function is a hyperplane (line) that linearly separates the point $(1, 1)$ from the other three possible input combinations

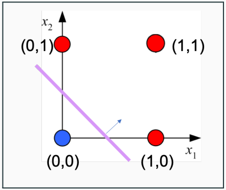
x_1	x_2	r
0	0	0
0	1	0
1	0	0
1	1	1



Computing Boolean OR with the perceptron

- Boolean OR function can be computer similarly
- Set the bias $w_0 = -0.5$ and the weights $w_1 = w_2 = 1$
- Now the function $w_1 x_1 + w_2 x_2 + w_0 > 0$ if and only if $x_1 = 1$ or $x_2 = 1$
- The function is a hyperplane separating the point $(0, 0)$ from the other input combinations

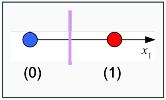
X_1	X_2	r
0	0	0
0	1	1
1	0	1
1	1	1



Computing Boolean NOT with the perceptron

- Boolean NOT function is simple to compute with a neuron with only one input
- Set the bias $w_0 = 0.5$ and the weight to $w_1 = -1$
- Now the function $w_1 x_1 + w_0 > 0$ if and only if $x_1 = 0$
- The function linearly separates 0 from 1

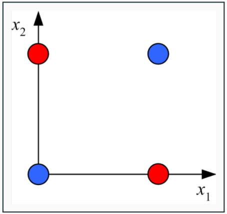
X_1	r
1	0
0	1



XOR with perceptron

- The exclusive or, or XOR operator cannot be represented by the perceptron
- This is because the output XOR function is not linearly separable: there is no hyperplane that can separate $(0, 0), (1, 1,)$ from $(0, 1), (1, 0)$

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

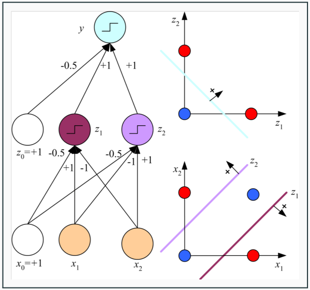


XOR with MLP

XOR can be computed by a simple neural network consisting of three neurons

$$XOR(x_1, x_2) = (x_1 \text{ AND NOT}(x_2)) \text{ OR } (\text{NOT}(x_1) \text{ AND } x_2)$$

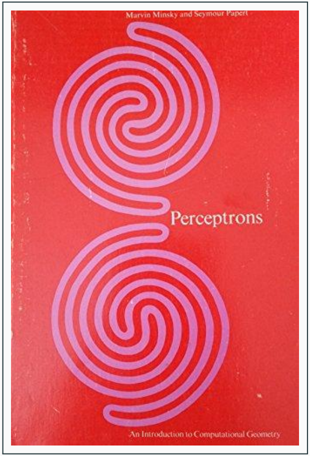
- The first layer computes two hyperplanes:
  - $z_1 = x_1 - x_2 - 0.5 > 0$ if and only if (x_1 AND NOT(x_2))
  - $z_2 = -x_1 + x_2 - 0.5 > 0$ if and only if (NOT(x_1) AND x_2)
- The second layer computes a single hyperplane implementing the OR $z_1 + z_2 - 0.5 > 0$ if and only if z_1 OR z_2 is true



A historical note: XOR with perceptron

A historical note:

- The inability of perceptron to compute the XOR was highlighted by Marvin Minsky and Seymour Papert in their book on Perceptrons published in 1969
- This finding contributed to the research on neural networks going out of fashion in the 1970's
- At the time, the representation power of MLPs was not widely understood
- Also, good algorithms to train MLPs were not known, so they were dismissed by the research community at the time



Representing arbitrary boolean functions with neural nets

- Perceptron can represent all three basic logical operators AND, OR and NOT
- All Boolean functions can be represented by combinations of these basic operations
- Thus, MLPs can in principle represent arbitrary Boolean functions
- However, **learning** arbitrary Boolean functions may still require prohibitive amount of data and time (e.g. the VC dimension of arbitrary Boolean functions of d variables is 2^d)

Representing arbitrary boolean functions with neural nets

- In fact already a MLP with a single hidden layer can represent all Boolean functions
- Construction of the network is simple: there is a hidden unit h_i, for each $\mathbf{x}_i$ for which $f(\mathbf{x}_i) = 1$, that will output 1 if the input equals $\mathbf{x}_i$ (the unit computes an AND over all input variables)
- The output layer outputs $+1$ if any of the hidden units outputs 1 (OR over the hidden units)

Representing arbitrary boolean functions with neural nets

- The network described before is fully memorizing the Boolean function, no learning or generalization is happening
- This network has exponential size in the number of variables
- Exponential size is not an artifact: one can prove that any network structure that allows representing any Boolean function must have exponential size in the input dimension (Shalev-Shwartz and Ben-David, 2014)

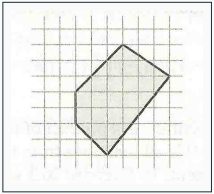
MLPs as universal approximators

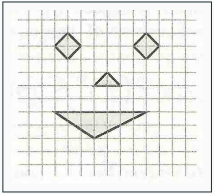
- Besides the ability of represent arbitrary Boolean functions MLPs can also approximate real valued functions that have bounded gradients (called Lipschitz functions) with arbitrary precision
- Given a function $f(\mathbf{x})$ the network will output value between $f(\mathbf{x}) - \epsilon$ and $f(\mathbf{x}) + \epsilon$, where $\epsilon > 0$ is the desired precision.
- However, again the price to pay is the size of the network: it will necessarily be of exponential size in the input dimension (Shalev-Shwartz and Ben-David, 2014)

As a whole these results tell us that neural networks are extremely flexible and thus are potentially prone to overfitting, unless there is sufficient training data

Geometric intuition

- A two layer network can represent convex polytopes, through an intersection of half-spaces defined by hyperplanes (top picture)
  - Each face of the polytope is defined by a single neuron in the hidden layer, the output layer computes an AND of the hidden layer activations
- A three layer network can represent disjunctions of convex polytopes: the final layer computes an OR of the second hidden layer outputs (bottom picture)





(Source: Shalev-Shwartz and Ben-David, 2014)

Learning Multi-Layer Perceptrons

Hardness of training MLPs

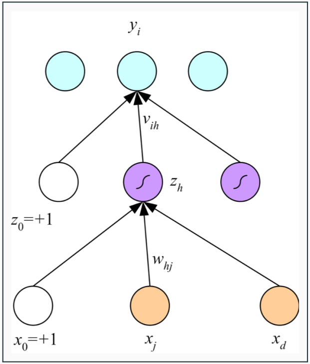
Learning optimal weights for MLPs and other neural networks is computationally hard (Shalev-Shwartz and Ben-David, 2014):

- It is NP-hard to find the parameters that minimizes the empirical error, for a network with a single hidden layer that contains 4 neurons or more
- Even close-to-minimal error is NP-hard to achieve
- Changing the structure of the network is not likely to make learning easier, since any function class that can represent intersections of halfspaces is NP-hard under some cryptographic assumptions

Thus in practice we need to resort in heuristic optimization approaches with no theoretical guarantees of optimality

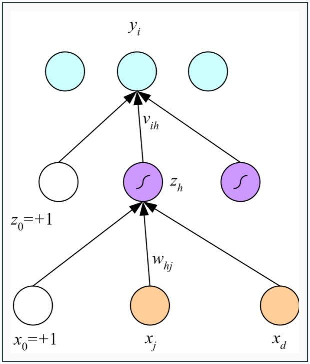
Stochastic Gradient Descent for MLPs

- Most training algorihms for MLPs are variants of stochastic gradient decent (SGD)
- Unlike with logistic regression and SVM problems, MLP optimization is a non-convex optimization problem
  - SGD generally converges to a local optimum
  - No theoretical guarantees how close to the global optimum we are
- In practise, SGD needs to be run many times with different initializations to find a good local optimum



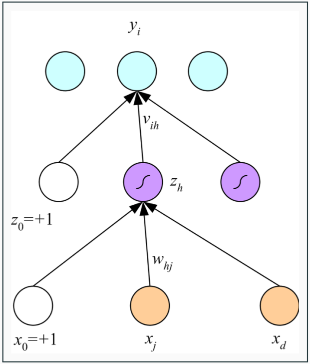
Stochastic Gradient Descent for MLPs

- SGD requires us to compute the gradient of the loss function with respect to a training example
- Unlike Logistic regression or SVM, there is no analytical expression for the gradient
- The expression for the gradient will be in general a expression involving nested sums and products
- The computation of the gradient and the update of the weights needs to be incrementally, layer by layer



Stochastic Gradient Descent for MLPs

- Let us study a two-layer MLP for regression
- There is one output neuron that has a linear activation function $y = y_i = \mathbf{v}^T \mathbf{z}$
- There are H hidden neurons with a logistic activation function $z_h = \frac{1}{1+\exp(-\mathbf{w}_h^T \mathbf{x})}$
- Squared loss is used as the loss function: $L(y, r) = \frac{1}{2}(y - r)^2$



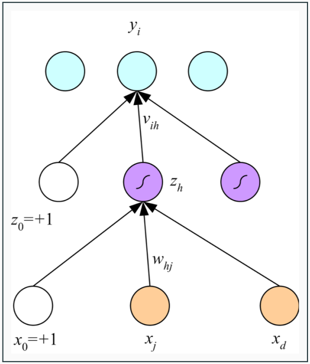
Stochastic Gradient Descent for MLPs

We traverse the network backwards from the output layer, first taking the hidden layer as fixed, considering z_k of the hidden units as inputs

- The gradient of the loss function with respect to the output layer weights is

$$\frac{\partial}{\partial v_{ih}} L(r, y_i) = \frac{\partial}{\partial v_{ih}} \frac{1}{2}(r - \sum_{h=0}^{H} v_{ih} z_h)^2$$

$$= (r - \sum_{h=0}^{H} v_{ih} z_h)(-z_h)$$



- The SGD update to the weight v_{ih} is a step along the negative gradient

$$\Delta v_{ih} = \eta(r - y_i) z_h$$

Stochastic Gradient Descent for MLPs

- The update for the hidden layer weights w_{hj} is not as simple, as we do not have a "desired output" for hidden layer neurons and thus no loss function either
- We can use the chain rule of differentiation

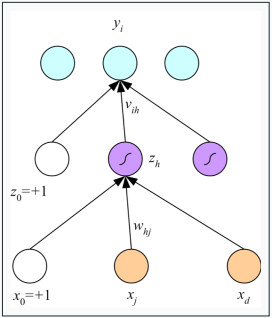
$$\frac{\partial L(r, y_i)}{\partial w_{hj}} = \frac{\partial L(r, y_i)}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}}$$

The derivatives of the three factors are given by:

$$\frac{\partial L(r, y_i)}{\partial y_i} = \frac{\partial}{\partial y_i} \frac{1}{2}(r - y_i)^2 = -(r - y_i)$$

$$\frac{\partial y_i}{\partial z_h} = \frac{\partial}{\partial z_h} \sum_{h=1}^{H} v_{ih} z_h = v_{ih}$$

$$\frac{\partial z_h}{\partial w_{hj}} = \frac{\partial}{\partial w_{hj}} \sigma_h(\sum_{k=0}^{I} w_{hk} x_k)$$



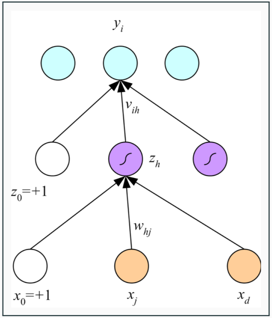
Stochastic Gradient Descent for MLPs

Using the logistic function as the activation function for the hidden layer $z_h = \sigma_h(\sum_{k=0}^{d} w_{hk}x_k) = \frac{1}{1+\exp(-\sum_{k=0}^{d} w_{hk}x_k)}$ we get:

$$\frac{\partial z_h}{\partial w_{hj}} = \frac{\partial}{\partial w_{hj}}\sigma_h(\sum_{k=0}^{d} w_{hk}x_k)$$

$$= \sigma_h(\sum_{k=0}^{d} w_{hk}x_k)(1-\sigma_h(\sum_{k=0}^{d} w_{hk}x_k))x_j$$

$$= z_h(1-z_h)x_j$$



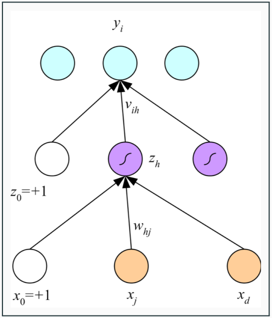
Stochastic Gradient Descent for MLPs

Now we have all the factors of the derivative of he loss function:

$$\frac{\partial L(r_i, y_i)}{\partial w_{hj}} = \frac{\partial L(r_i, y_i)}{\partial y_i} \frac{\partial y_i}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}} \qquad = -(r_i - y_i) v_{ih} z_h (1 - z_h) x_j$$

Interpretation

- $(r_i - y_i)v_h$ can be seen as a error term of hidden unit h
- This error is **backpropagated** from the output layer to the hidden unit
- The larger the weight v_h, the larger "responsibility" of the error is given to unit h



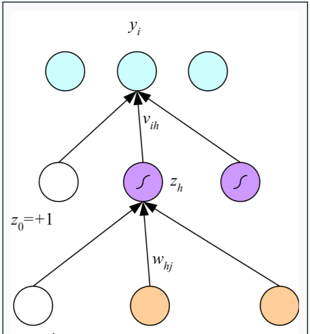
Stochastic Gradient Descent for MLPs

$$\frac{\partial L(r_i, y_i)}{\partial w_{hj}} = \frac{\partial L(r_i, y_i)}{\partial y_i}\frac{\partial y_i}{\partial z_h}\frac{\partial z_h}{\partial w_{hj}} = -(r_i - y_i)v_{ih}z_h(1 - z_h)x_j$$

The update for the weight is a step along the negative gradient

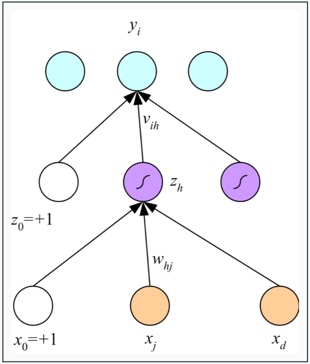
$$\Delta W_{hj} = -\eta\frac{\partial L(r_i, y_i)}{\partial w_{hj}} = \eta(r_i - y_i){\color{red}v_{ih}}z_h(1 - z_h)x_j$$

- Note the update of the hidden layer weight refers to the output layer weight v_{ih}
- We should first update w_{hj} the hidden layer weights using the old values of v_{ih}, then update the output layer weights



Backpropagation training algorithm for two-layer MLP for regression

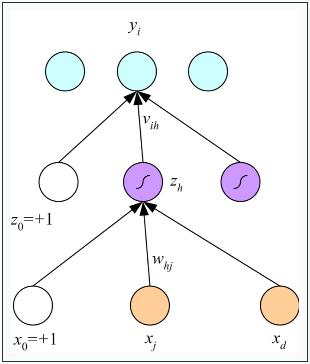
Initialize all w_{hj}, v_{ih} randomly to range $[-0.01, 0.01]$
repeat
  Draw a training example $(\mathbf{x}, r)$ at random
  Forward propagation of activation:
  Set $z_h = \sigma_h(\mathbf{w}_h^T \mathbf{x})$ for $h = 1, \ldots, H$
  $y = \mathbf{v}^T \mathbf{z}$
  Backpropagation of error:
  $\Delta \mathbf{v} = \eta(r - y)\mathbf{z}$
  $\Delta \mathbf{w}_h = \eta(r - y_i) v_{ih} z_h (1 - z_h)\mathbf{x}$, for $h = 1, \ldots, H$
  Update weights:
  $\mathbf{v} = \mathbf{v} + \Delta \mathbf{v}$
  $\mathbf{w}_h = \mathbf{w}_h + \Delta \mathbf{w}_h$
until stopping criterion is satisfied



Backpropagation algorithm for classification tasks

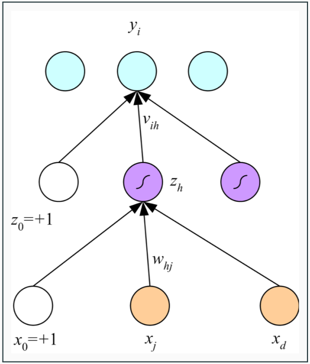
The backpropagation algorithm described can be adapted for classification tasks:

- For binary classification task we change the output activation function to sigmoid function, either logistic (with 0/1 labels) or tanh ($-1/+1$ labels)
- Multiclass classification can be implemented by using K output units and applying a softmax-function $y_i = \frac{\exp(\mathbf{v}_i^T \mathbf{z})}{\sum_k \exp(\mathbf{v}_k^T \mathbf{z})}$



Multiple hidden layers

- Adding hidden layers to the network means that both forward propagation of activation and the backward propagation of error needs to be iterated for more layers
- The error backpropagation then involves a chain-rule over all hidden layers



Improving convergence

A few simple tricks can be used to speed up convergence:

- Momentum: The SGD update may cause oscillation; subsequent update directions may be very different to each other. This can be helped by computing a running average of the current negative gradient direction and the previous update direction

$$\Delta \mathbf{w}^{(t)} = -\eta \frac{\partial L(r_t, y_t)}{\partial \mathbf{w}} + \alpha \Delta \mathbf{w}^{(t-1)}$$

- Adaptive learning rate: the stepsize η can be changed based on whether error on the training set has been decreasing during the last few passes over the training data (epochs):

$$\Delta \eta = \begin{cases} +a & \text{if } \hat{R}^{(T)} < \frac{1}{p} \sum_{k=1}^{p} \hat{R}^{(T-k)} \\ -b\eta & \text{otherwise} \end{cases},$$

  where $\hat{R}^{(t)}$ denotes the average loss over the training data on epoch t

Using GPUs

- The use of Graphical Processing Units (GPU) is widely spread in neural network research
- GPUs can process especially matrix operations (esp. matrix products) very efficiently
- The operations in the backpropagation algorithm can be written so that the majority of computation is in the form of matrix products

Avoiding overfitting

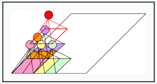
Due to their flexibility neural networks are prone to overfitting. This can be alleviated by certain techniques

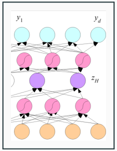
- Early stopping: the weights in the network tend to increase during training and grdually overfitting becomes more likely. Stopping training prior convergence can help.
- Dropout: during training, randomly fixing some weights during an update stops the network adjusting to the noise too well. This technique is widely used in current deep learning algorithms

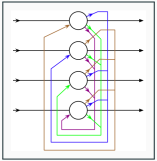
Other neural network architectures

Particular architectures of neural networks can be used for specific purposes

- **Convolutional Neural Networks** are used e.g. for image input. The instead of fully connected layers, a local neighborhood is cross-connected, but the neighborhoods can overlap
- **Autoencoder networks** have an "hourglass" structure, where a middle hidden layer is much narrower than the input and output layers. This is used for learning new representations for data.
- **Recurrent networks** are used for data that has variable length e.g. speech and natural language







Summary

- Neural networks are a model family inspired by the human brain
- Multi-layer perceptrons can represent and approximate remarkably complex functions
- Large training data is generally needed to avoid overfitting
- Finding optimal weights for a neural network is generally NP-hard problem
- Variants of stochastic gradient descent are generally used to train neural networks

The Course CS-E4890 - Deep Learning (Spring 2021) `https://mycourses.aalto.fi/course/view.php?id=28212` is recommmended to those who wish to learning more about neural networks