*** Applied Machine Learning Fundamentals ***
Neural Networks / Deep Learning

Daniel Wehner, M.Sc.

SAP SE / DHBW Mannheim

Winter term 2023/2024

Find all slides on GitHub (DaWe1992/Applied_ML_Fundamentals)

Lecture Overview

Unit I	Machine Learning Introduction
Unit II	Mathematical Foundations
Unit III	Bayesian Decision Theory
Unit IV	Regression
Unit V	Classification I
Unit VI	Evaluation
Unit VII	**Classification II**
Unit VIII	Clustering
Unit IX	Dimensionality Reduction

Agenda for this Unit

Section:

Introduction

What is Deep Learning?
History of Deep Learning
Biological Motivation

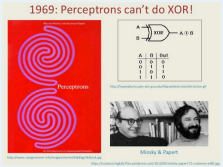
What is Deep Learning?

- **Deep Learning** is an umbrella term for all methods related to artificial neural networks
- It is a **supervised** method and **model based**
- Artificial neural networks are inspired by the human brain (but are much simpler)
- Lots of different architectures exist. Commonly used:
  - **Multi-Layer perceptrons (MLPs)**
  - **Convolutional neural networks (CNNs, ConvNets)**
  - **Recurrent neural networks (LSTMs, GRUs, etc.)**

History of Deep Learning

Early booming (1950s – early 1960s)

F. Rosenblatt suggests the **Perceptron** learning algorithm: `Click here!`



Setback I (mid 1960s – late 1970s)

M. Minsky and *S. Papert* (1969):
Serious problems with perceptron algorithm: It cannot learn the **XOR problem**.

History of Deep Learning (Ctd.)

Renewed enthusiasm (1980s)

- New techniques available
- **Backpropagation** for deep nets

Setback II (1990s – mid 2000s)

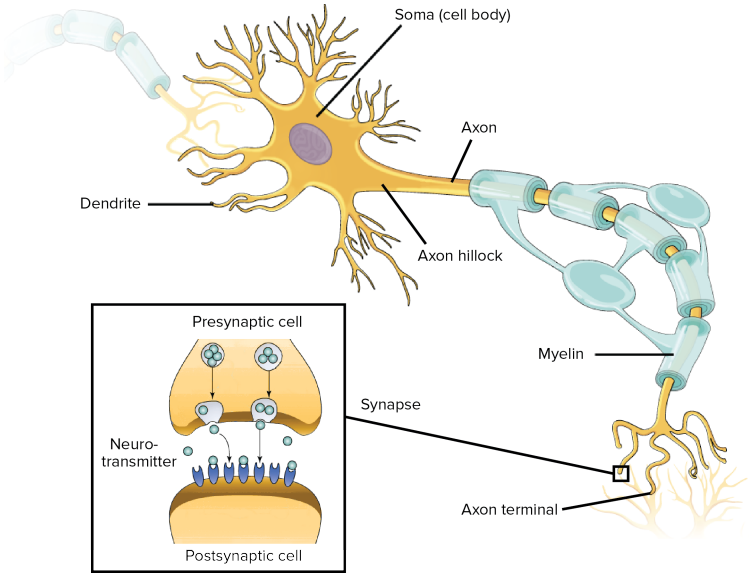
- Other techniques were considered superior (e. g. SVMs)
- CS journals rejected papers on neural networks

'Deep Learning' (since mid 2000)

More data, faster computers, better optimization techniques...

Biological Motivation

- All neurons are connected and form a complex **network**
- **Transmitter chemicals** within the fluid of the brain influence the **electrical potential** inside the body of the neurons
- If the **membrane potential** reaches some threshold, the neuron **fires**
  $\Rightarrow$ A pulse of fixed length is sent down the **axon**
- The axon connects the neuron with other neurons (via **synapses**)
- Probably there are 100 trillion (!!!) synapses in the human brain
- **Refractory period** after a neuron has fired


Soma (cell body)
Axon
Dendrite
Axon hillock
Myelin
Presynaptic cell
Synapse
Neuro-
transmitter
Axon terminal
Postsynaptic cell

How do Humans / Animals learn?

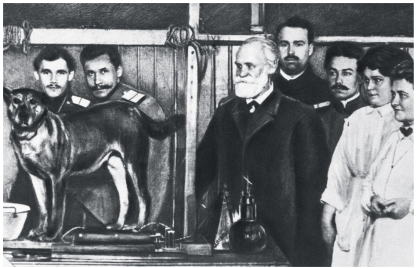
- **Idea:** Mechanism of learning is **association**
- **Hebbian learning:** If the firing of one neuron repeatedly assists in firing another neuron, their synaptic connection will be strengthened

*'When an axon of cell A is near enough to excite a cell B and repeatedly or persistently **takes part in firing it**, some growth process or **metabolic change** takes place in one or both cells such that A's **efficiency**, as one of the cells firing B, is **increased**.'*

*'The general idea is an old one, that any two cells or systems of cells that are **repeatedly active at the same** time will tend to become **'associated'**, so that activity in one facilitates activity in the other.'* *Hebb*

Classical / Pavlovian Conditioning

- Dog salivates when given food
- Food is an **unconditioned stimulus (US)**
- Salivation in response to food is **unconditioned response (UR)**
- Food is paired with the sound of a bell
- Bell is **conditioned stimulus (CS)**
- Bell will eventually elicit salivation event without food
- Salivation is **conditioned response (CR)**



Blocking

Group A	train N+	train LN+	test L-	$\Rightarrow$ no conditioning
Group B		train LN+	test L-	$\Rightarrow$ conditioning

- CS is a light (L), a noise (N), or a combination of both (LN)
- US is a mild shock that is paired with the CS in the training phase (+)
- Fear response is tested after training when only L is presented without shock (-)
- Group B shows conditioning; Group A does not: **N blocks L**
- This is hard to explain with Hebbian learning
- **Idea:** Learning only happens, if there is a **prediction error**

Section:

Perceptrons

The original Perceptron Algorithm
'New' Perceptron Learning Algorithm

Original Perceptron Algorithm [*Rosenblatt*, 1957]

The **Perceptron** algorithm was originally proposed by *Frank Rosenblatt* in 1957

Frank Rosenblatt's (1928 – 1971) perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in `Principles of Neurodynamics` published in 1962. Frank Rosenblatt died in July 1971 on his 43rd birthday, in a boating accident in Chesapeake Bay.

Original Perceptron Algorithm - Model Function

- The Perceptron is a binary classification model
- The (linear) model function is given by:

$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = g(\boldsymbol{w}^\intercal \boldsymbol{x} + b) \tag{1}$$

- The nonlinear activation function g is given by a **step function** of the form

$$g(p) = \begin{cases} +1 & \text{if } p \geqslant 0 \\ -1 & \text{if } p < 0. \end{cases} \tag{2}$$

Original Perceptron Algorithm - Perceptron Criterion

- For the labels $y^{(i)}$ we use the values +1 (positive) and -1 (negative)
- We seek a weight vector $\boldsymbol{w}$ such that $\left(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}^{(i)} + b\right)y^{(i)} > 0$, i. e.
  - $\boldsymbol{w}^\mathsf{T}\boldsymbol{x}^{(i)} + b > 0$ if $y^{(i)} = +1$
  - $\boldsymbol{w}^\mathsf{T}\boldsymbol{x}^{(i)} + b < 0$ if $y^{(i)} = -1$
- **Perceptron criterion** (error function):

$$\mathcal{J}_P(\boldsymbol{w}, b) = -\sum_{n \in \mathcal{M}} \left(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}^{(i)} + b\right)y^{(i)} \tag{3}$$

- $\mathcal{M}$ is the set of misclassified training examples

Original Perceptron Algorithm

Algorithm 1: Perceptron Algorithm proposed by Rosenblatt (1962)

Input: Input data $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y}), y \in \{-1, +1\}$

Initialize parameters $(\boldsymbol{w}, b)$ to small random numbers

forall $(\boldsymbol{x}, y) \in \mathcal{D}$ *until convergence* **do**

  if $\boldsymbol{x} \notin \mathcal{M}$ *(correct classification)* **then**

    do nothing

  else

    if $y = +1$ **then**

      $\boldsymbol{w} \longleftarrow \boldsymbol{w} + \boldsymbol{x}$

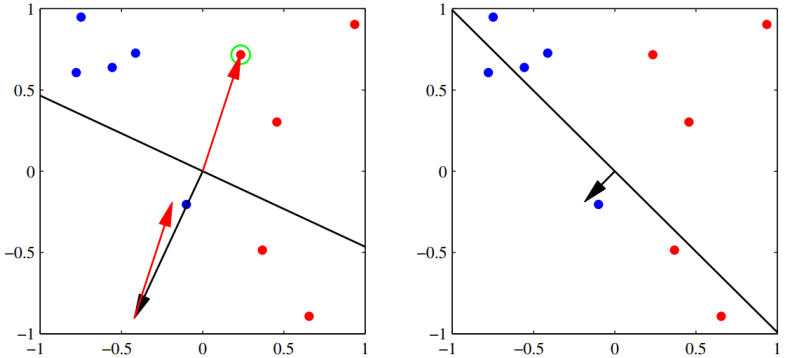
      $b \longleftarrow b + 1$

    if $y = -1$ **then**

      $\boldsymbol{w} \longleftarrow \boldsymbol{w} - \boldsymbol{x}$

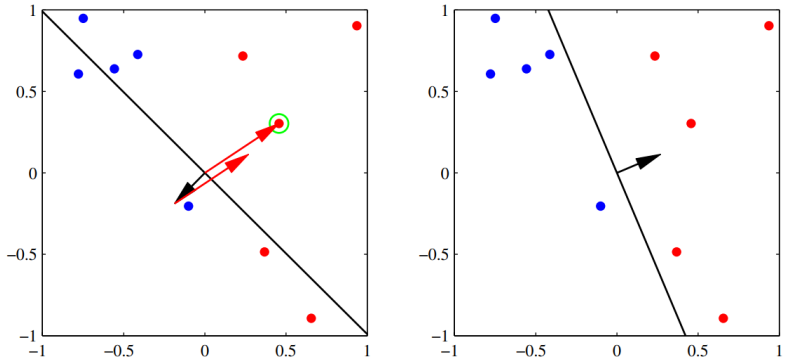
      $b \longleftarrow b - 1$

Original Perceptron Algorithm Example



cf. Bishop.2006, page 195

Original Perceptron Algorithm Example (Ctd.)



cf. Bishop.2006, page 195

Mark 1 Perceptron Hardware

Figure 4.8 Illustration of the Mark 1 perceptron hardware. The photograph on the left shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focussed onto a 20×20 array of cadmium sulphide photocells, giving a primitive 400 pixel image. The perceptron also had a patch board, shown in the middle photograph, which allowed different configurations of input features to be tried. Often these were wired up at random to demonstrate the ability of the perceptron to learn without the need for precise wiring, in contrast to a modern digital computer. The photograph on the right shows one of the racks of adaptive weights. Each weight was implemented using a rotary variable resistor, also called a potentiometer, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

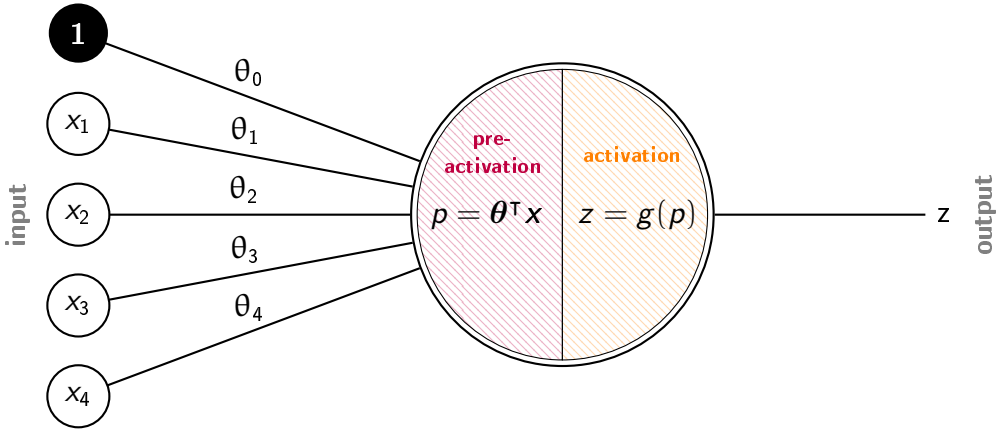
cf. Bishop.2006, page 196

Perceptron Convergence Theorem

Perceptron Convergence Theorem:
If the training data is **linearly separable**, then the perceptron learning algorithm will **converge after a finite amount of time** and classifies **all training data examples correctly**.

- **The number of steps can be substantial** (until convergence we cannot distinguish between a nonseparable problem and one that is simply slow to converge)
- **The solution is not unique** and depends on the initialization of $\boldsymbol{w}$ and b and on the order of presentation of the data points!

The Architecture of a 'modern' Neuron



Perceptron

- The neuron receives an input vector $\boldsymbol{x}$:

$$\boldsymbol{x} = (1, x_1, x_2, \ldots, x_m)^\intercal \in \mathbb{R}^{m+1}$$

- Each input signal is weighted by a factor θ_j: *(weight of synaptic strength)*

$$\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2, \ldots, \theta_m)^\intercal \in \mathbb{R}^{m+1}$$

- We compute the **pre-activation** p and the **activation** z according to:

$$p = \boldsymbol{\theta}^\intercal \boldsymbol{x} = \sum_{j=0}^{m} \theta_j x_j \qquad z = g(p) \tag{4}$$

Perceptron (Ctd.)

- The simplest activation function is to use a threshold ρ: (not used in practice, since this function **not differentiable**)

$$g_\rho(p) = \begin{cases} 0 & \text{if } p \leqslant \rho \\ 1 & \text{if } p > \rho \end{cases} \tag{5}$$

- Quick example: • $\boldsymbol{x} = (1, 0, 0.5)^\intercal$ • $\boldsymbol{\theta} = (1, -0.5, -1)^\intercal$ • $\rho = 0$

$$p = \boldsymbol{\theta}^\intercal \boldsymbol{x} = 1 \cdot 1 + (-0.5) \cdot 0 + (-1) \cdot 0.5 = 0.5$$
$$z = g_{\rho=0}(0.5) = 1$$

Perceptron Learning

- Learning means choosing the correct weights $\boldsymbol{\theta}^*$ from a set of possible hypotheses $\mathcal{H}$ (**hypothesis space**):

$$\mathcal{H} := \left\{ \boldsymbol{\theta} : \boldsymbol{\theta} \in \mathbb{R}^{m+1} \right\}$$

- How to learn the weights from a dataset $\mathcal{D} = (\boldsymbol{X}, \boldsymbol{y})$?
- **Algorithm outline**: Stochastic gradient descent
  1. Pick a training example $(\boldsymbol{x}, y) \in \mathcal{D}$
  2. Calculate the activation z for that training example
  3. Update the weights $\boldsymbol{\theta}$ based on the error

Perceptron Learning (Ctd.)

- How can we compute the error? $\Rightarrow$ We need a loss function $\mathcal{J}(\boldsymbol{\theta})$:

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \ell\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}^{(i)}), y^{(i)}\right) \tag{6}$$

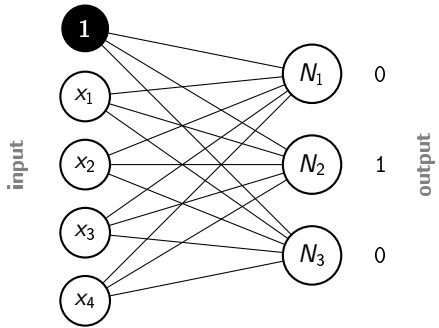
- ℓ is the loss of a single training example, e. g. squared error or cross entropy
- **Gradient descent**: Compute the gradient and go into the negative direction of the gradient:

$$\boldsymbol{\theta}^{(t+1)} \longleftarrow \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{J}(\boldsymbol{\theta}) \tag{7}$$

Multi-Class Perceptron

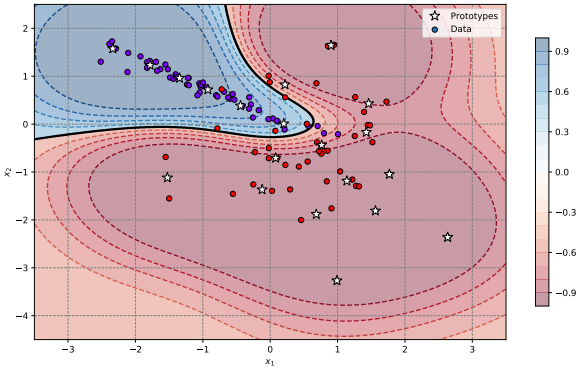
- A single neuron can only distinguish two classes
- If there are more than two classes: Simply use more neurons!
- Use **one-hot encoding** for the classes and **softmax** as activation function (later)
- Example for three classes:

$\mathcal{C}_1$	1 0 0
$\mathcal{C}_2$	0 1 0
$\mathcal{C}_3$	0 0 1



What about non-linear Datasets?

- Perceptrons cannot learn non-linear boundaries
- Remember *Minsky* and *Papert*?
- What can we do?
  1. Add feature mapping (cf. right)
  2. Add hidden layers **(Multi-Layer Perceptrons)**



Section:

Multi-Layer-Perceptrons (MLPs)

Overview
Backpropagation
Activation Functions

Multi-Layer Perceptrons

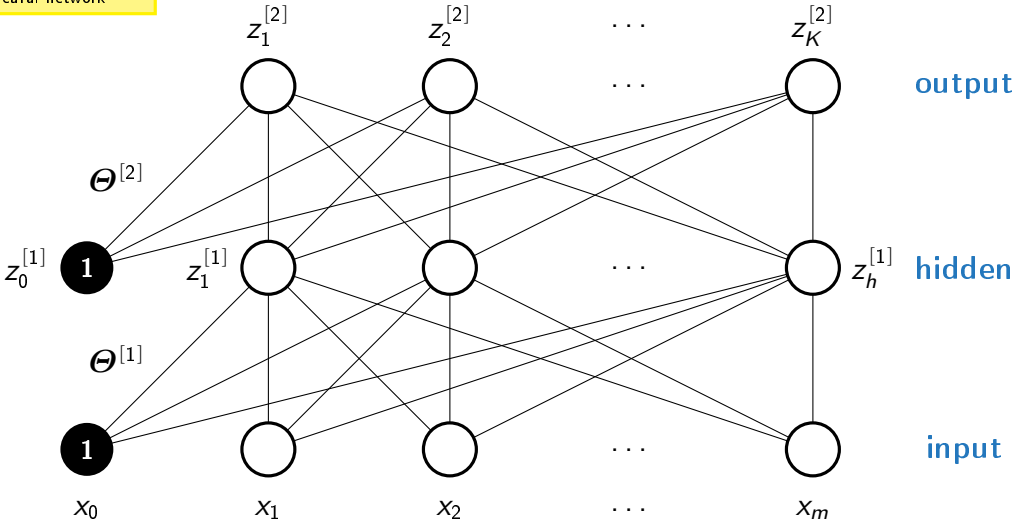
- In theory, a **Multi-Layer Perceptron (MLP)** can approximate any continuous function arbitrarily well
- An MLP with λ layers is a function $h : \mathbb{R}^{m+1} \rightarrow \mathbb{R}^K$, parameterized by network parameters $\boldsymbol{\Theta}^{[1]}, \boldsymbol{\Theta}^{[2]}, \ldots, \boldsymbol{\Theta}^{[\lambda]}$
- In each layer, a non-linearity is applied: $g^{[1]}, g^{[2]}, \ldots, g^{[\lambda]}$

$$z^{[1]}(x) = g^{[1]}\left(\boldsymbol{\Theta}^{[1]} x\right)$$
$$\ldots$$
$$z^{[\lambda]}(x) = g^{[\lambda]}\left(\boldsymbol{\Theta}^{[\lambda]} z^{[\lambda-1]}(x)\right)$$

Prediction:

$$y_p = \arg\max_k z^{[\lambda]}(x)$$

This is a fully connected neural network



MLP Learning

1. Start by randomly initializing the network weights
2. **Do not set an initial value of 0 for all weights! (Why?)**
   $\Rightarrow$ Initialize to small random numbers
3. Perform a forward pass through the network, i. e. make predictions for the elements of the training dataset
4. Using the predictions, compute a scalar loss value $\mathcal{J}(\boldsymbol{\Theta})$
5. Calculate the gradients of the loss w. r. t. each network parameter and update all parameters (gradient descent, backpropagation)

Forward Pass (for one Hidden Layer, $\lambda = 2$)

- **Step 1**: Perform a **forward pass**:

$$z_l^{[1]}(\boldsymbol{x}) = g^{[1]}\left(\sum_{j=0}^{m} \Theta_{lj}^{[1]} x_j^{(i)}\right) \qquad \text{(hidden activation)}$$

$$z_k^{[2]}(\boldsymbol{x}) = g^{[2]}\left(\sum_{l=0}^{h} \Theta_{kl}^{[2]} z_l^{[1]}\right) \qquad \text{(output activation)}$$

- $g(\cdot)$ is the activation function, e. g. sigmoid, tanh, ReLU (see section **Activation Functions** for more details)
- $\boldsymbol{\Theta}$ are the network parameters (to be learned)

Backpropagation (for one Hidden Layer, $\lambda = 2$)

- **Step 2:** Compute the network loss
  For simplicity we assume square loss: $\ell = \left(z_k^{[2]}(\boldsymbol{x}^{(i)}), y_k^{(i)}\right)^2$

$$\mathcal{J}(\boldsymbol{\Theta}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{J}^{(i)}(\boldsymbol{\Theta}) = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K}\ell\left(z_k^{[2]}(\boldsymbol{x}^{(i)}), y_k^{(i)}\right)$$

- **Step 3:** Compute the error gradient for the **output layer** w. r. t. $z_k^{[2]}$:

$$\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_k^{[2]}(\boldsymbol{x}^{(i)})} = \ell'\left(z_k^{[2]}(\boldsymbol{x}^{(i)}), y_k^{(i)}\right) =: \delta_k^{(i)}$$

Backpropagation (Ctd.)

- **Step 4:** Compute the weight gradient for the **output layer**:

$$\begin{aligned}\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial \Theta_{kl}^{[2]}} &= \frac{\mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_k^{[2]}(\boldsymbol{x}^{(i)})} \frac{\partial z_k^{[2]}(\boldsymbol{x}^{(i)})}{\partial \Theta_{kl}^{[2]}} \\ &= \ell'\left(z_k^{[2]}(\boldsymbol{x}^{(i)}), y_k^{(i)}\right) \cdot g'^{[2]} \left(\sum_{t=0}^{h} \Theta_{kt}^{[2]} z_t^{[1]}(\boldsymbol{x}^{(i)})\right) \cdot z_l^{[1]}(\boldsymbol{x}^{(i)}) \\ &= \delta_k^{(i)} \cdot g'^{[2]} \left(\sum_{t=0}^{h} \Theta_{kt}^{[2]} z_t^{[1]}(\boldsymbol{x}^{(i)})\right) \cdot z_l^{[1]}(\boldsymbol{x}^{(i)})\end{aligned}$$

Backpropagation (Ctd.)

- **Step 5**: Compute the error gradient for the **hidden layer**:

$$\begin{aligned}\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_l^{[1]}} &= \sum_{k=1}^{K} \frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_k^{[2]}(\boldsymbol{x}^{(i)})} \frac{\partial z_k^{[2]}(\boldsymbol{x}^{(i)})}{\partial z_l^{[1]}} \\ &= \sum_{k=1}^{K} \ell'\left(z_k^{[2]}(\boldsymbol{x}^{(i)}), y_k^{(i)}\right) \cdot g'^{[2]}\left(\sum_{t=0}^{h} \Theta_{kt}^{[2]} z_t^{[1]}(\boldsymbol{x}^{(i)})\right) \cdot \Theta_{kl}^{[2]} \\ &= \sum_{k=1}^{K} \delta_k^{(i)} \cdot g'^{[2]}\left(\sum_{t=0}^{h} \Theta_{kt}^{[2]} z_t^{[1]}(\boldsymbol{x}^{(i)})\right) \cdot \Theta_{kl}^{[2]} =: \widehat{\delta}_l^{(i)}\end{aligned}$$

Backpropagation (Ctd.)

- **Step 6**: Compute the weight gradient for the **hidden layer**:

$$\frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial \Theta_{lj}^{[1]}} = \frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial z_l^{[1]}} \cdot g'^{[1]} \left(\sum_{t=0}^{m} \Theta_{lt}^{[1]} x_t^{(i)} \right) \cdot x_j^{(i)}$$

$$= \widehat{\delta}_l^{(i)} \cdot g'^{[1]} \left(\sum_{t=0}^{m} \Theta_{lt}^{[1]} x_t^{(i)} \right) \cdot x_j^{(i)}$$

- **Step 7**: Update the weights using gradient descent, e. g.:

$$\Theta_{lj}^{[1]} \longleftarrow \Theta_{lj}^{[1]} - \alpha \frac{\partial \mathcal{J}^{(i)}(\boldsymbol{\Theta})}{\partial \Theta_{lj}^{[1]}}$$

Some Remarks

- **Check your gradients:**

$$\nabla h(x) \overset{!}{\approx} \frac{1}{\varepsilon}\big(h(x+\varepsilon) - h(x)\big)$$

- **Hyper-parameter optimization is absolutely necessary:**
  - # hidden layers
  - # hidden units
  - activation functions
  - learning rate
  - batch size
  - # training epochs
  - regularization
  - ...
- Have a look at: `https://playground.tensorflow.org`

Sigmoid Function

$$g(x) = \frac{1}{1 + e^{-x}} \qquad g'(x) = \sigma(x)(1 - \sigma(x))$$

- Value range: 0 to 1
- **Problem 1:** Gradient can become very small **(vanishing gradient problem)**
- **Problem 2:** Output is not zero-centered (makes optimization harder)

Sigmoid Function (Ctd.)

'Convergence is usually faster if the average of each input variable over the training set is close to zero. To see this, consider the extreme case where all the inputs are positive. Weights to a particular node in the first weight layer are updated by an amount proportional to δx where δ is the (scalar) error at that node and x is the input vector. When all of the components of an input vector are positive, all of the updates of weights that feed into a node will have the same sign (i. e. sign(δ)). As a result, these weights can only all decrease or all increase together for a given input pattern. Thus, if a weight vector must change direction it can only do so by zigzagging which is inefficient and thus very slow.' - **Yann LeCun et al., Efficient BackProp, 1998** (`http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`)

Tangent Hyperbolic (tanh)

$$g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad g'(x) = 1 - \tanh(x)^2$$

- Value range: -1 to 1
- Zero-centered
- **Still suffers from the vanishing gradient problem**

Rectified Linear Unit (ReLU)

$$g(x) = \max(0, x) \qquad g'(x) = \begin{cases} 0 & \text{if } x \leqslant 0 \\ 1 & \text{otherwise} \end{cases}$$

- ReLU does not lead to vanishing gradients and is very efficient to compute
- Use ReLU as a hidden layer activation function!
- **But:** Pay attention to the initialization of your parameters to avoid *'dying ReLUs'* (parameter settings where single neurons will always output 0)
- **Use batch-normalization!**

Activation Functions Overview

Name	Function	Derivative
Sigmoid	$g(x) = \frac{1}{1+e^{-x}}$ 	$g'(x) = \sigma(x) \cdot (1 - \sigma(x))$ 
Tanh	$g(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ 	$g'(x) = 1 - \tanh(x)^2$ 
ReLU	$g(x) = \max(0, x)$ 	$g'(x) = \begin{cases} 0 & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$ 
Leaky ReLU	$g(x) = \begin{cases} \alpha x & \text{if } x \leqslant 0 \\ x & \text{if } x > 0 \end{cases}$ 	$g'(x) = \begin{cases} \alpha & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$ 
ELU	$g(x) = \begin{cases} \alpha(e^{x} - 1) & \text{if } x \leqslant 0 \\ x & \text{if } x > 0 \end{cases}$ 	$g'(x) = \begin{cases} \alpha e^{x} & \text{if } x \leqslant 0 \\ 1 & \text{if } x > 0 \end{cases}$ 

Global Activation: Softmax Activation

$$softmax_k(\boldsymbol{x}) = \frac{\exp\{x_k\}}{\sum_{j=1}^{K} \exp\{x_j\}}$$

- Softmax is a **global activation function** (i. e. it also depends on the preactivations of the other units in the layer)
- It is used to squash the last layer's activations into a probability distribution, **such that the activations sum to 1**

Section:

Further Network Architectures

Convolutional Neural Networks
Recurrent Neural Networks

Convolutional Neural Networks (CNNs)

- A fully-connected MLP network has got lots of parameters, i. e. it might be **too complex** or **computationally inefficient** for some tasks
- The input position does not matter in all cases
  (e. g. word positions when classifying e-mails, pixels in images)
- An MLP is fed with a fixed-size input vector, but we might have **variable-size input data**
  (e. g. images in different resolutions, text sequences of different lengths, etc.)
- **Solution:** Use Convolutional Neural Networks (CNNs)!

CNN: General Idea

'A convolutional neural network is designed to identify indicative local predictors in a large structure, and combine them to produce a fixed size vector representation of the structure, capturing these local aspects that are most informative for the prediction task at hand.' - **Yoav Goldberg**

- A CNN consists of multiple convolutional layers
- A convolutional layer consists of a filter and a non-linear activation function
- A Pooling layer extracts the most important features **independent of their input position**

Convolution Layer

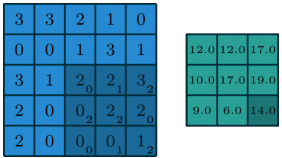
Definition of convolution:

$$(x \star g)[i] = \sum_{j=-w}^{w} x[i-j] \cdot g[j]$$

- $\star$ is the convolution operator
- x is the convolution input
- i denotes the current position in the input
- w is the filter size / window size
- g is the convolution filter (kernel)

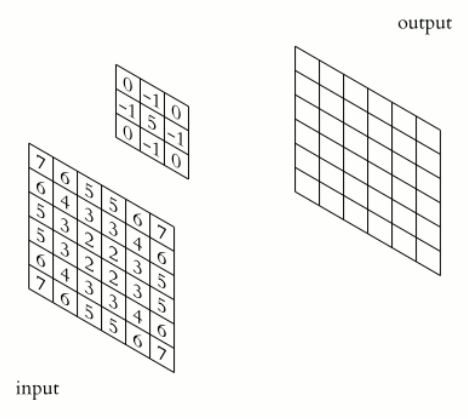
Convolution Layer (Ctd.)

- We shift a filter across the input data (e. g. image pixels) with a certain **stride** and multiply the input with the filter weights at each position
- The resulting feature map is usually smaller in size



$$(x \star g)[i] = \sum_{j=-w}^{w} x[i-j] \cdot g[j]$$

Convolution Animation



Pooling Layer

- The outputs of the convolutions are passed through non-linear activation functions (e. g. ReLU, tanh, ...)
- Pooling is applied to extract only the most important activations:
  - Max-pooling (extracts the maximum activation)
  - Mean-pooling (extracts the mean activation)
- **There are no parameters involved in the pooling operation**

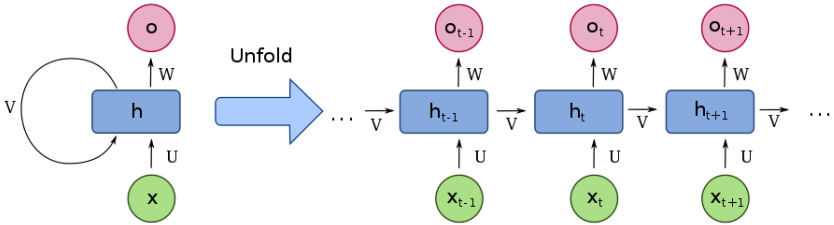
In image processing, pooling is necessary to reduce the dimensionality

Recurrent Neural Networks (RNNs)

- **Recurrent Neural Networks (RNNs)** can be used to process sequences
  - Sequence labeling (e. g. POS-tagging)
  - Sequence transduction (e. g. machine translation)
  - Sequence classification
- They are similar to feed-forward networks, but have a **recurrent loop in the computational graph**
- By unfolding the network, it can be trained exactly like a feed-forward MLP using backpropagation

RNN Architecture

$$\boldsymbol{h}_t = \sigma_h(\boldsymbol{U}\boldsymbol{x} + \boldsymbol{V}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h)$$
$$\boldsymbol{o}_t = \sigma_o(\boldsymbol{W}\boldsymbol{h}_t + \boldsymbol{b}_o)$$



Section:

Wrap-Up

Summary
Self-Test Questions
Lecture Outlook

Summary

- Neural Networks are powerful models for pattern recognition
- The perceptron can classify all training examples correctly, **if the training data is linearly separable**
- **MLPs are universal function approximators**
- Backpropagation is a recursive procedure based on the **chain rule of calculus** to obtain the gradients for neural network learning
- Different neural network architectures like CNNs and RNNs exist for solving different kinds of problems

Self-Test Questions

1. What is the relation between neural networks and logistic regression?
2. What is a perceptron? Which problems can it solve and which not?
3. Why do we often use multiple layers instead of a simple perceptron?
4. How does backpropagation work? How does a neural network learn?
5. What are advantages and disadvantages of using neural networks?
6. What are CNNs and RNNs? For which tasks are they suitable?

What's next...?

Unit I	Machine Learning Introduction
Unit II	Mathematical Foundations
Unit III	Bayesian Decision Theory
Unit IV	Regression
Unit V	Classification I
Unit VI	Evaluation
Unit VII	Classification II
Unit VIII	**Clustering**
Unit IX	Dimensionality Reduction

Thank you very much for the attention!

Topic: *** Applied Machine Learning Fundamentals *** Neural Networks / Deep Learning
Term: Winter term 2023/2024

Contact:
Daniel Wehner, M.Sc.
SAP SE / DHBW Mannheim
daniel.wehner@sap.com

Do you have any questions?