

Neural Networks and Deep Learning

Script for personal use
No passing on, no publication, no reproduction!

Prof. Dr. rer. nat. Carsten Meyer

Institut für Angewandte Informatik

Fachbereich Informatik & Elektrotechnik

Fachhochschule Kiel



Christian-Albrechts-Universität zu Kiel

Lecturer

- Prof. Dr. rer. nat. Carsten Meyer
carsten.meyer@fh-kiel.de
Office: C12-1.78, Tel. 0431 / 210 4107
Office hours: by arrangement

- Curriculum vitae (1):
- 1986 – 1993: Studies of Physics
in Göttingen
and Freiburg / Brsg.

Thesis: „Neurons with nonlinear signal processing on the dendritic tree“

- 1993: Internship Management Consulting company,
Neuss



Curriculum vitae (2)

- 1993 – 1997: Ph.D. studies
 - Graduate school „Organisation and Dynamics of Neural Networks“, University of Göttingen, and
 - „Interdisciplinary Center for Neural Computation“, Hebrew University of Jerusalem
 - Ph.D. thesis: „Temporal correlations in stochastic networks of ‚spiking‘ neurons“ (Statistical Physics of Neural Networks)



Curriculum vitae (3)

- 1998 – 2011: Research Scientist / Senior Scientist
 - Philips Forschungslabor Aachen / Hamburg

Selected projects:

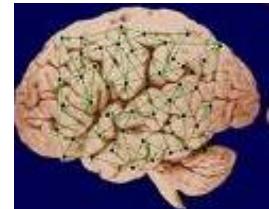
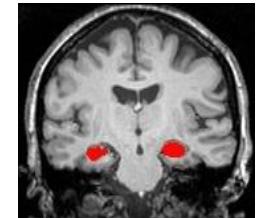
- **Automatic speech recognition** (Man-Machine interface group, Aachen)
 - Dialogue systems, directory assistance, speech-based navigation systems
 - Project leader professional dictation systems
- **Medical imaging and image processing, medical signal processing** (Molecular / X-Ray Imaging Systems group, Aachen)
 - Simulation and analysis of dynamic cardiac imaging in nuclear medicine
 - Project leader clinical decision support for clinical cardiology
 - Automatic ECG analysis
 - Automatic heart segmentation (CT, MR, 3-D X-Ray, Ultrasound)
- **Medical image processing** (Digital Imaging group, Hamburg)
 - Automatic brain segmentation in 3-D MR images

Curriculum vitae (4)

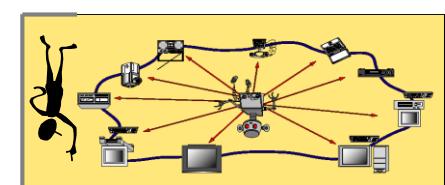
- Since March 2011: Professor at FH Kiel

Interests:

Development and application of intelligent algorithms
in image, speech and document processing:



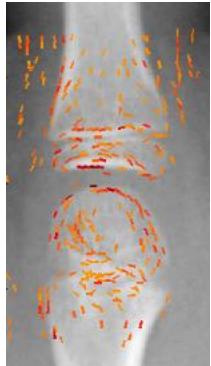
- (Medical) image processing and image analysis
- Pattern recognition and machine learning
- Deep learning and neural networks
- Medical signal processing
- Man-machine interfaces
- Automatic speech and document processing



Working group „Pattern recognition and machine learning“

- Prof. Hauke Schramm, Prof. Carsten Meyer, FH Kiel
- Advanced algorithms from pattern recognition and machine learning for **object localization / classification, image analysis, signal analysis**
- Applications: Industrial and medical image processing, video sequences
- Currently 2 PhD students (+ master and bachelor students)

Bone age
classification



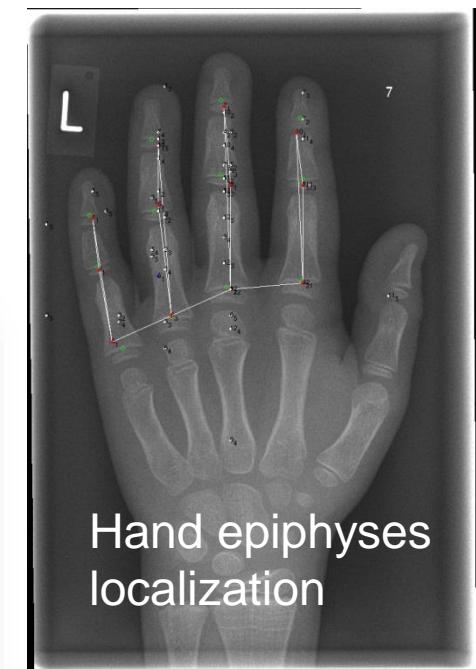
Gender
classification



Facial
landmarks



Hand epiphyses
localization



Neural Networks and Deep Learning: Overview of lecture (1)

I) Introduction

II) Biological basis

- Neurons and networks

III) Neuron models

- In particular: (Single-layer) perceptron

IV) Artificial neural networks: Introduction and architectures

V) Feedforward neural networks

- In particular: Multi-layer perceptron

VI) Learning in neural networks: Introduction

- Learning paradigms (supervised / unsupervised), overfitting, generalization
- Perceptron learning, stochastic gradient descent, backpropagation

VII) Deep Learning

- What's new? Why only recently?

Neural Networks and Deep Learning: Overview of lecture (2)

VIII) Convolutional neural networks

- Principles, common architectures, transposed convolution

X) Recurrent neural networks

- Backpropagation through time (BPTT)
- Long short term memory (LSTM)

XI) Unsupervised learning

- Autoencoders
- Generative models (variational autoencoders, generative adversarial networks)

XIII) Summary

Organization

- Structure: 2h lecture + 2h exercises (lab)
 - 6 ECTS (CAU)
 - For Master „Computer science“
- Place and time
 - **Lecture:** Wednesday, 10:00 – 11:30, room LMS6.R11
 - **Exercises:** Wednesday, 11:45 – 13:15, room HRS3 – R.105a/b
 - From April 08 until June 30
 - Some dates of both lecture AND lab are shifted, e.g. July 01 → June 30
 - For Wednesday, April 29, and Wednesday, May 6 see OLAT announcements
 - Other shifts announced via OLAT system
 - You may / should bring your own laptops to the lab
- Exam:
 - Potentially on Wednesday, July 15
 - **Please note that the date of a potential second exam is unclear; please participate in the first exam**

Criteria for credit points

- It is required to pass both the exam and the lab exercises

1. Passing written exam

- At least 50% of available points required

2. Successful completion of exercises

- It is required to submit *all* exercise sheets (before the deadline!)
- At least 5 of 6 exercise sheets have to be passed
- A single exercise sheet is passed if at least
 - 50% of the exercises are correctly solved (teams up to 2 persons)
 - 67% of the exercises are correctly solved (teams with 3 or 4 persons)
- In cases of doubt for any sheet an oral exam may be required
- If any exercise is copied from *any* source:
The complete exercise sheet is immediately considered *not passed*!
- No grades for lab (either „passed“ or „not passed“)
- Participation in lab is **mandatory!**

Allowed aids for the exam

- Single piece of paper (single-sided!) DIN-A4 with personal notes
- No further aids or devices
 - No calculators
 - No books or scripts
 - No mobile phones
- Deviations from these rules will be regarded as attempt of deception
- Emails regarding these issues will not be answered
 - In case of questions please use personal communication (latest possibility is the last lecture; no questions will be answered afterwards)

Requirements

- Strong interest in neural networks and deep learning
- Analytical and abstract thinking; affinity to math and software (python)
- Ability to work independently and to constantly and thoroughly review the lecture material also with external sources (books, papers, internet)
- Time and interest to work on the lab exercises (they need time!)
- This course is intended to prepare for a (master) thesis / project in the area of neural networks / deep learning / machine learning
- It is not easy and not an easy earning of credit points!
- Strong motivation is required!
- Exercises are (partly) based on python scripts to apply (and thereby understand) the discussed concepts
- If you want to program / implement by yourself: Do it yourself! (It's not part of the course!)

Further notes

- Lecture slides will be made available in the OLAT system (look for „Neural networks and deep learning“)

Please check from time to time for updates!

- Exercises can be downloaded from the OLAT system
- Please watch OLAT system for announcements
- If you have questions:
 - Please contact me after the lecture or ask for an appointment
 - Emails only in really urgent cases please

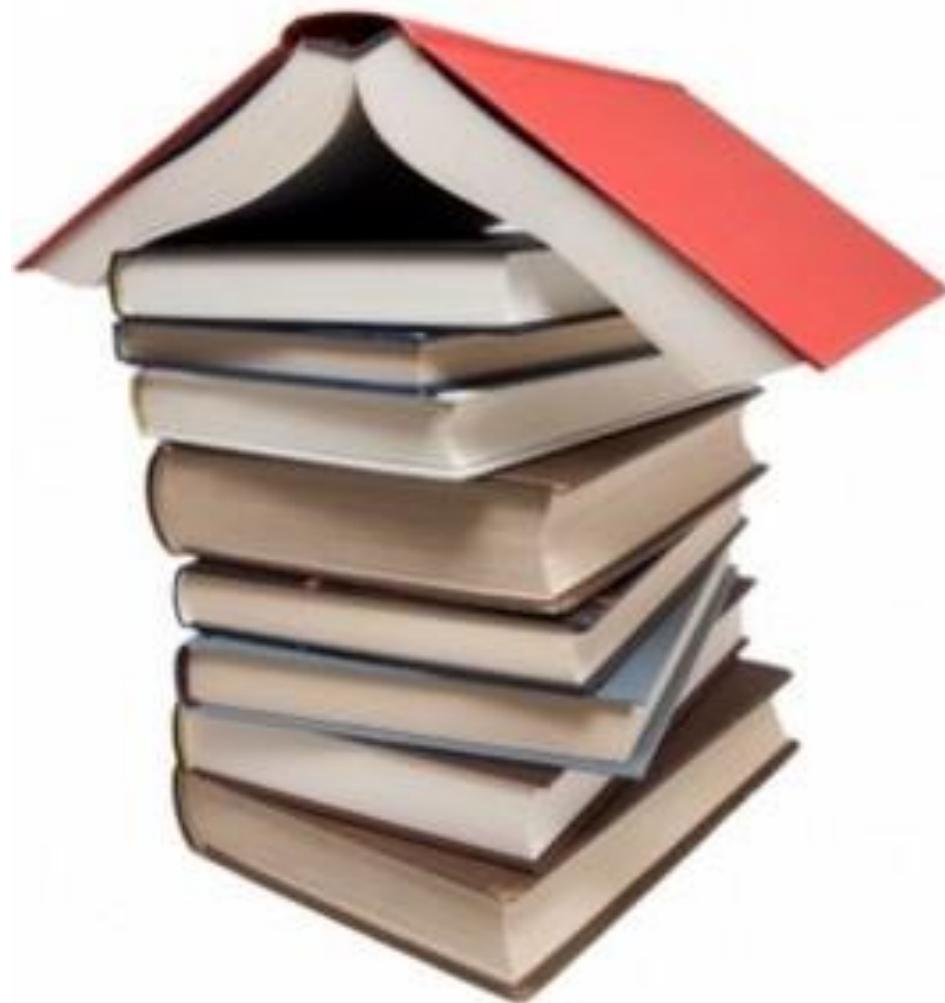
Remarks regarding the lecture slides

- The goal of the lecture slides is to relieve you from having to copy all the lecture material by yourself
- However, making notes during the lecture is *not superfluous!*
- The slides are ***no teaching book!***
 - It is still mandatory to read additional material from text books or other sources
 - Note that the lecture slides are comprehensible only with additional explanations during the lecture and with accompanying personal notes
 - It is strongly encouraged to review the lecture slides directly after each lecture and ask questions if something remains unclear!

Language

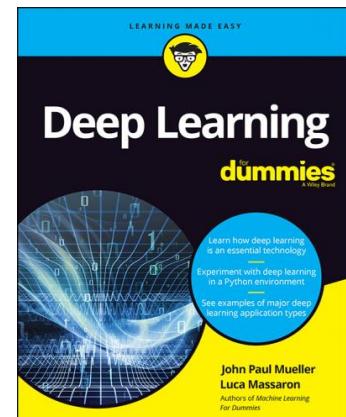
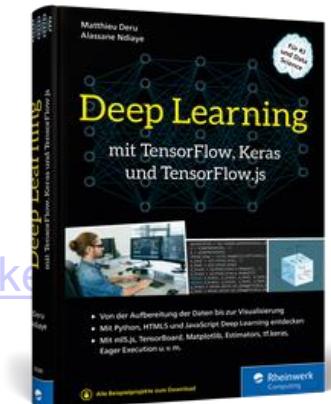
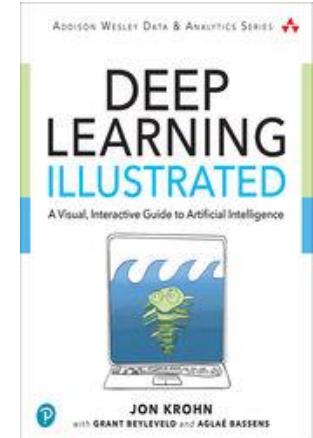
- Official course language will be English
 - if at least one English-speaking student is present
 - if not ... you can choose
- However...
 - Please tell me when I'm talking too fast or when I should repeat something in German for better understanding!
 - You may ask questions at any time in German
 - You may turn in exercises in German
- *Please ask questions and give feedback!*

Literature & Co....



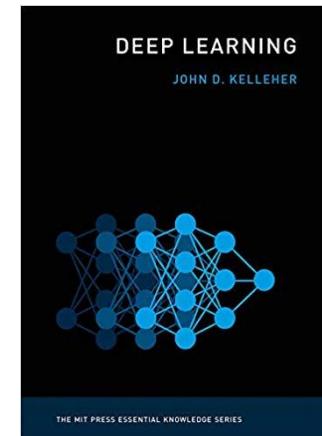
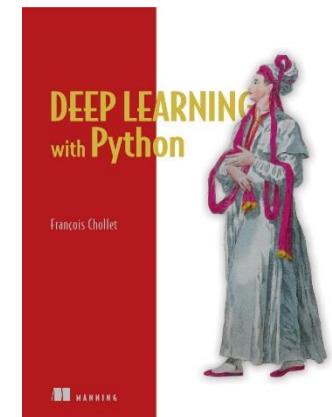
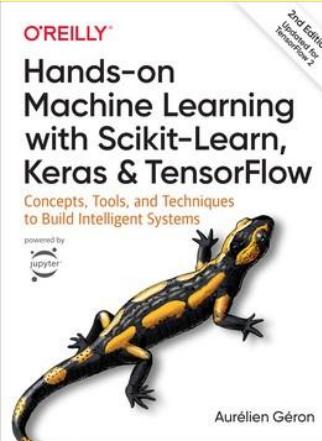
Textbooks (1): New books including deep learning

- John Krohn, Grant Beyleveld, Aglaé Bassens:
„Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence“ Addison-Wesley (2020)
 - <https://www.deeplearningillustrated.com/>
- Matthieu Deru, Alassane Ndiaye:
„Deep Learning mit TensorFlow, Keras und Tensorflow.js“
Rheinwerk (2019) (in German)
 - https://www.rheinwerk-verlag.de/deep-learning-mit-tensorflow-keras-tensorflowjs_4715/
- John Paul Mueller, Luca Massaron:
„Deep Learning for dummies“
John Wiley and Sons (2019)
Free download at
<http://www.allitebooks.org/deep-learning-for-dummies/>



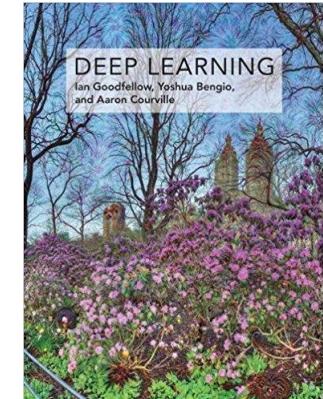
Textbooks (2): New books including deep learning

- Aurelien Geron:
„Hands-On Machine Learning with Scikit-Learn, Keras and Tensorflow“ (2nd ed.), O'Reilly (Sept. 2019)
 - <https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/#toc-start>
- Francois Chollet:
„Deep Learning with Python“
Manning Publications (2017)
 - <https://www.manning.com/books/deep-learning-with-python>
- John D. Kelleher:
„Deep Learning“ (Aug. 2019)
The MIT Press Essential Knowledge Series
Accessible introduction
 - <https://mitpress.mit.edu/books/deep-learning-1>

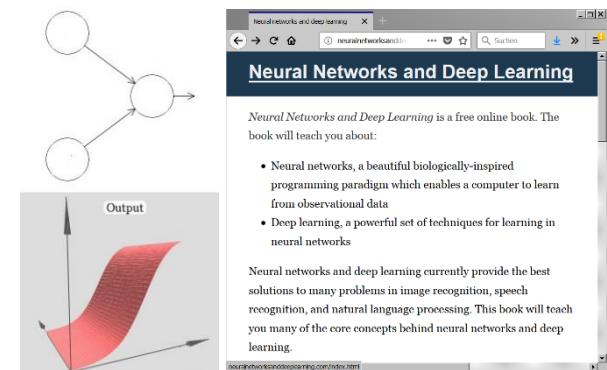


Textbooks (3): New books including deep learning

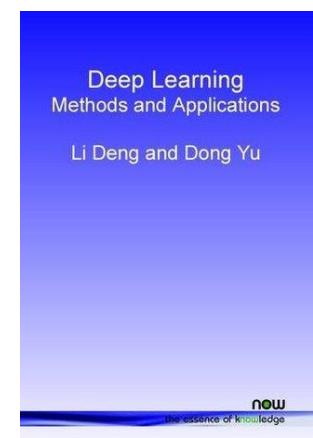
- Ian Goodfellow, Yoshua Bengio, Aaron Courville:
„Deep Learning“
MIT Press (2016)
– <https://mitpress.mit.edu/books/deep-learning>



- Michael Nielsen:
„Neural networks and deep learning“
Free online book (2016)
– <http://neuralnetworksanddeeplearning.com/>

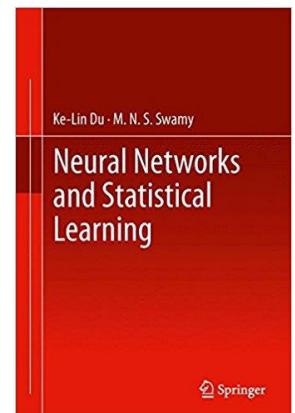
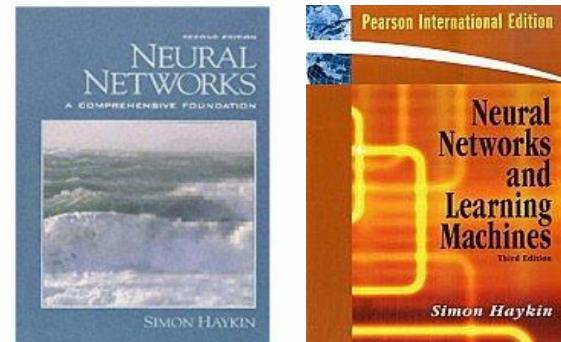
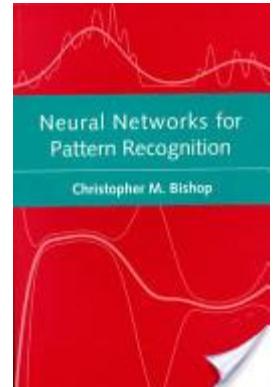


- Li Deng, Dong Yu:
„Deep Learning: Methods and Applications“
Now Publishers (2014), free pdf version available at
– <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/DeepLearning-NowPublishing-Vol7-SIG-039.pdf>



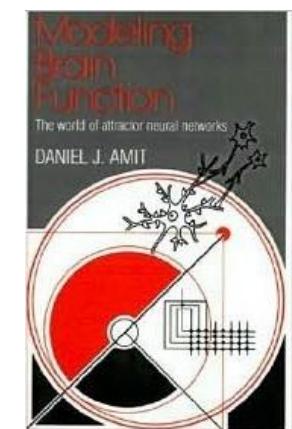
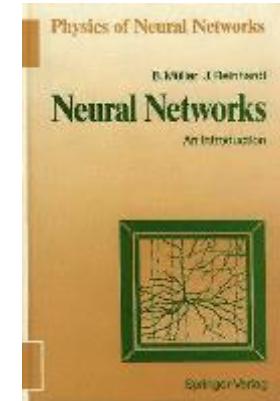
Textbooks (4): Former books without deep learning

- Christopher M. Bishop:
„Neural networks for pattern recognition“
Oxford University Press (1995)
- S. Haykin:
„Neural networks. A comprehensive foundation“
Prentice Hall (1999)
„Neural Networks and Learning Machines“
Prentice Hall (2009)
– advanced
- Ken-Lin Du, M. N. S. Swamy:
„Neural Networks and Statistical Learning“
Springer (2013)



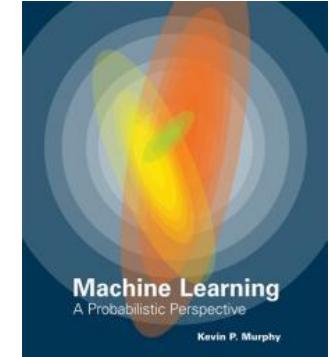
Textbooks (5): Statistical Physics of Neural Networks

- B. Müller, J. Reinhardt, M. Strickland:
„Neural Networks – An Introduction“
Springer, Heidelberg, 2nd ed. (1995)
– <http://th.physik.uni-frankfurt.de/~jr/nn/neunet.html>
- Raul Rojas:
„Neural Networks - A Systematic Introduction“
Springer, Berlin (1996)
– online version: <http://page.mi.fu-berlin.de/rojas/neural/>
- Daniel J. Amit:
„Modeling brain function –
The world of attractor neural networks“
Cambridge University Press (1992)
– Statistical physics of neural networks

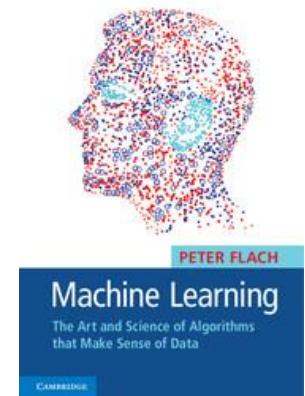


Textbooks (6): Machine learning (many, many books...!)

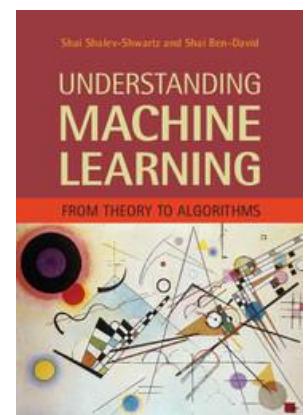
- Kevin P. Murphy:
„Machine Learning – A Probabilistic Perspective“
MIT Press (2012)
– <http://www.cs.ubc.ca/~murphyk/MLbook/>



- Peter Flach:
„Machine Learning – The Art And Science of Algorithms that Make Sense of Data“
Cambridge University Press (2012)

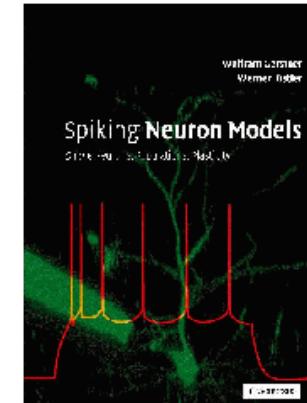


- Shai Shalev-Shwartz, Shai Ben-David:
„Understanding Machine Learning – From Theory to Algorithms“
Cambridge University Press (2014)

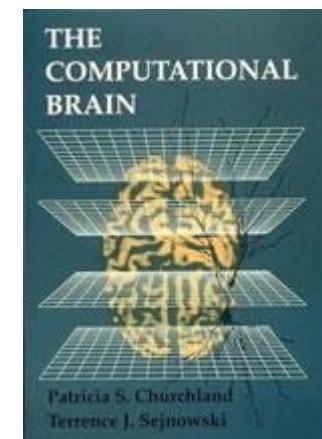


Textbooks (7): More on neurophysiology...

- Wulfram Gerstner, Werner M. Kistler:
„Spiking Neuron Models.
Single Neurons, Populations, Plasticity“.
Cambridge University Press (2002)
 - advanced
 - <http://icwww.epfl.ch/~gerstner/BUCH.html>
 - Online version:
<http://ii.fmph.uniba.sk/~farkas/Courses/Neurocomp/References/gerstner.snm.book02.pdf>

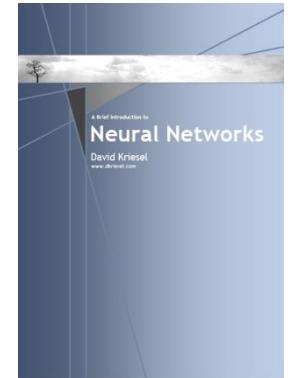


- Patricia S. Churchland, Terrence J. Sejnowski:
„The Computational Brain“
MIT Press (1992)
 - neurobiological perspective
- ... and many more...



Tutorials and websites (1)

- David Kriesel:
„A Brief Introduction to Neural Networks“
 - Available at
http://www.dkriesel.com/_media/science/neuronalenetze-en-epsilon2-dkrieselcom.pdf
 - Also in German
- Deep learning:
 - <http://deeplearning.net/>
 - http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial (Stanford)
 - <https://www.manning.com/books/deep-learning-with-python>
- ... and many more (more suggestions welcome!)



INTRODUCTION

Introduction

If the human brain was simple enough for us to understand,
we would still be so stupid that we couldn't understand it.

Jostein Gaarder

Introduction: Neural networks

- What are neural networks?

1.) Biological neural networks:

- Network of interconnected **biological** neurons (= nerve cells)
- spatially coherent / distributed, anatomically or functionally connected

2.) Artificial neural networks:

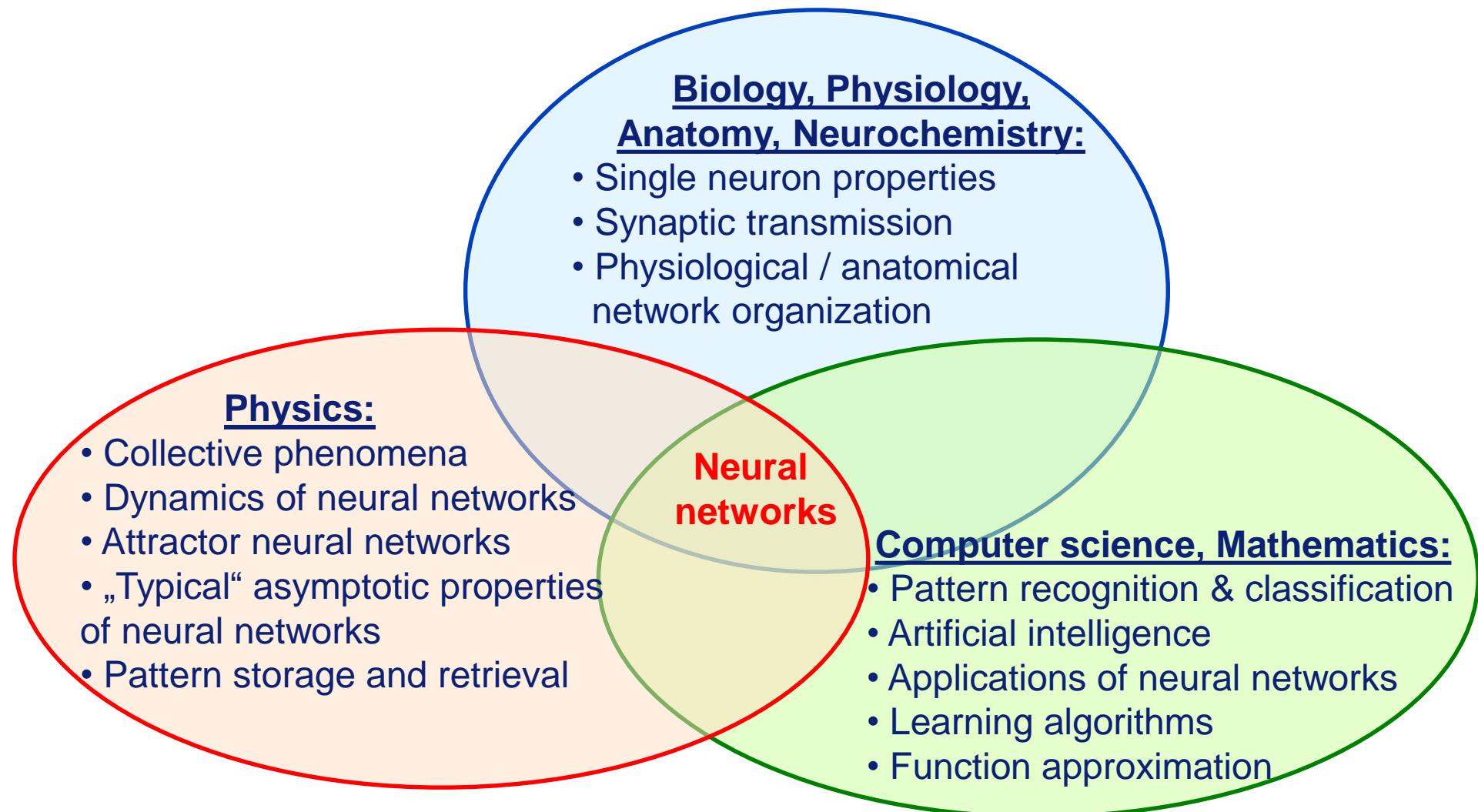
- Network of interconnected **artificial** neurons (= formal models that mimic some properties of biological neurons)
- use mathematical / computational model for information processing

Purpose:

- a) understanding biological networks
- b) solving problems in artificial intelligence, pattern recognition etc.

Disciplines involved in studying neural networks

- Multidisciplinary approach!



Why studying neural networks?

1.) Understanding some basic principles of information processing in the brain

- capability for learning and self-organization
- generalization capability, flexibility, adaptation to different environments
- ability to handle incomplete, contradictory and noisy data
- fault tolerance
- parallel processing

Some examples for the power of the brain:

- Object recognition even in noisy or partly obscured data
- Understanding of context from pieces of information
- Learning of language, cycling, social behaviour etc.
- Face recognition (in 0.1sec)

2.) Exploiting (some of) those principles to solve problems in artificial intelligence and pattern recognition

- solving problems by learning, not by implementation
- generalization, fault tolerance, association, parallel processing

(Flawed) comparison between brain and computer

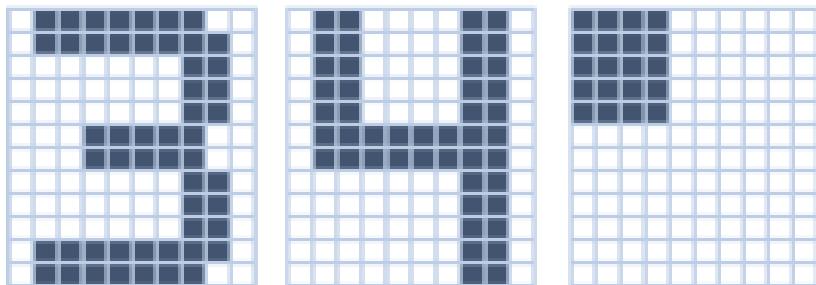
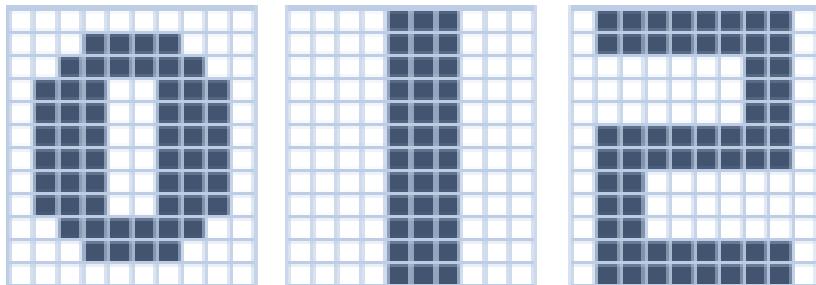
	Brain / neural systems	Computer
No. of processing units	$\approx 10^{11}$	$\approx 10^9$
Type of processing units	Neurons	Transistors
Type of calculation	Massively parallel	Usually serial
Data storage	Associative	Address-based
Switching time	$\approx 10^{-3}$ s	$\approx 10^{-9}$ s
Possible switching operations	$\approx 10^{13}$ s ⁻¹	$\approx 10^{18}$ s ⁻¹
Actual switching operations	$\approx 10^{12}$ s ⁻¹	$\approx 10^{10}$ s ⁻¹
Type of instruction	Learning by examples	Explicit programming
Fault tolerance	Distinct (noisy / fuzzy data)	Small
Robustness against failure	→ restricted functionality	→ complete breakdown

- So is the computer superior?
 - No, e.g. due to the brain's amazing learning and generalization abilities, i.e. solving *new* tasks by means of abilities learned in *other* tasks
- learning and generalization

Application example of a Hopfield neural network: Associative memory

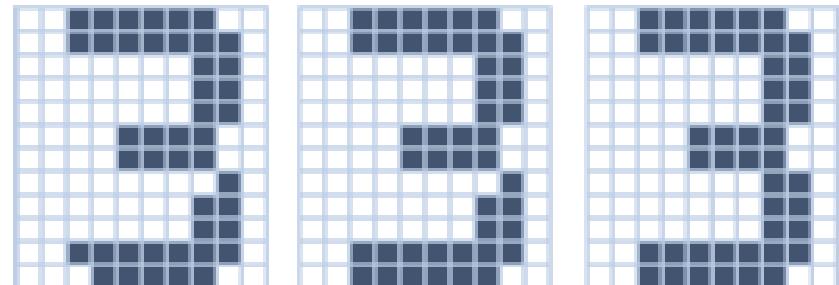
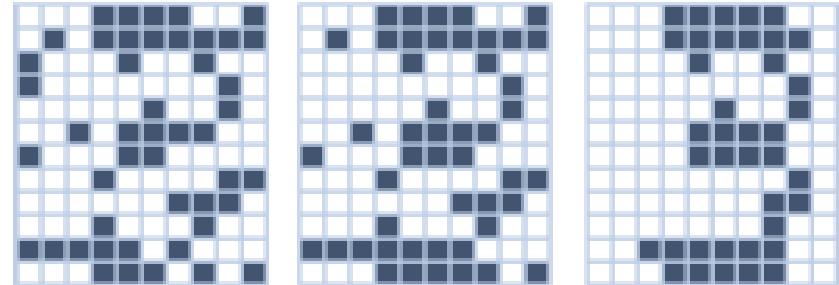
- **Hopfield network:** Example of a recurrent neural network (not deep!)
- **Associative memory:** An initial pattern „close“ to some stored memory is retrieved, *partial* input sufficient for memory retrieval
- Example:

Stored patterns (10x12 pixel each)



(Static) images of the learned target states in the network (which should become stable, stationary states in the network)

Noisy initial state converges to „3“

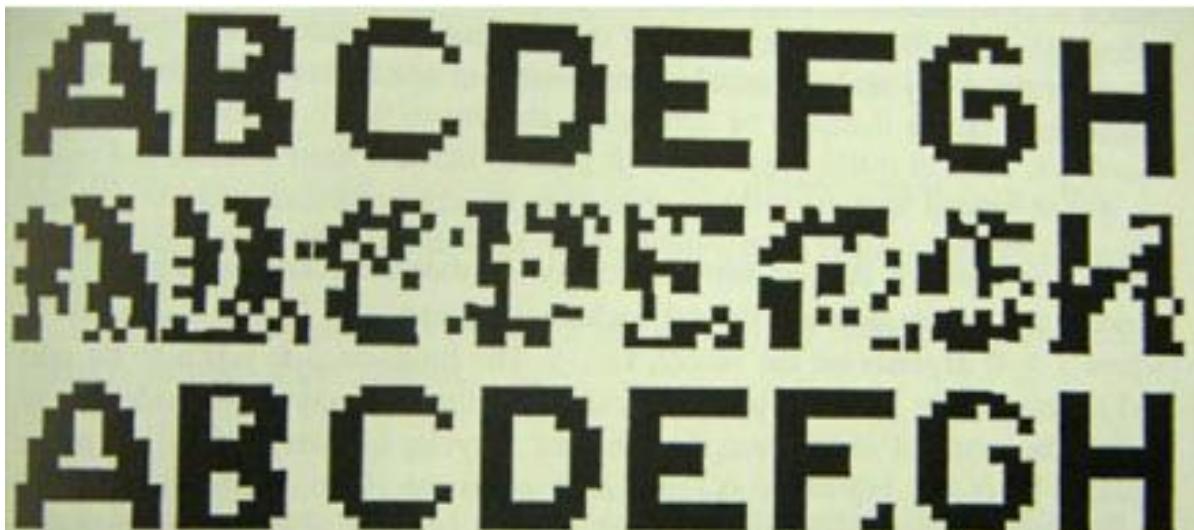


Dynamics (time evolution) of the output of the neurons in the network, given an initial stimulus (first image)

Application example of a Hopfield neural network: Associative memory

- Hopfield network with 100 neurons
- Learning of 8 patterns (shown in first row)
- Then: Network starts in an incomplete / noisy starting pattern (second row)
- Final network output: „stationary“ state corresponding to one of the stored patterns (potentially with minor deviations, third row)
 - „Associative memory“

Learned patterns



Starting pattern

Network output

Fig. 1.12. The first eight letters of the alphabet, A–H (top), have been stored in a network of $N = 100$ formal neurons. After unlearning, corrupted versions (middle) with about 12% noise are presented to the network, which then retrieves the original patterns (bottom). Note that the retrieved patterns closely resemble but need not coincide with their prototypes

From: Domany, van Hemmen, Schulten, „Models of Neural Networks I, Springer 1995

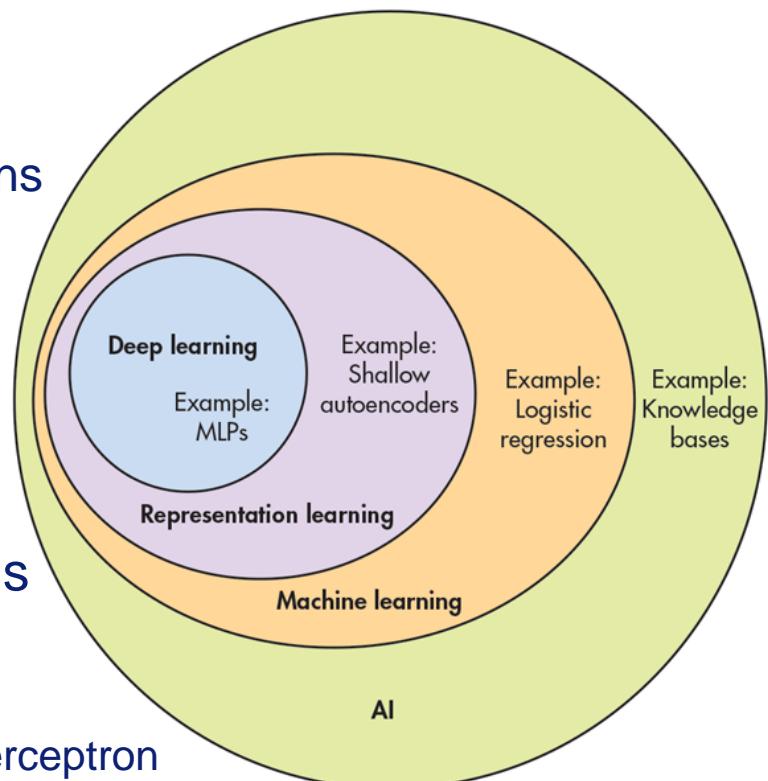
32

Introduction: Deep learning

- What is deep learning?

Deep learning are machine learning methods based on learning *hierarchical* data representations

- Subfield of „representation learning“
 - But hierarchical (multi-layer) representations
- Representations are learned
 - Not *engineered* as in standard machine learning
- Learned models used in many applications
 - Artificial intelligence, pattern recognition...

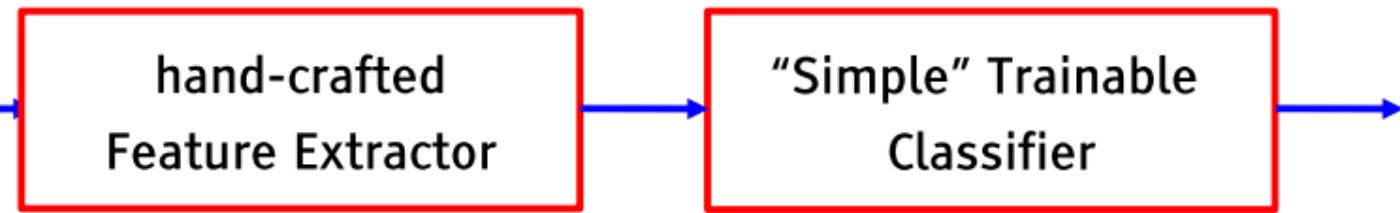


MLP: Multi-layer perceptron
AI: artificial intelligence

Deep learning versus standard machine learning

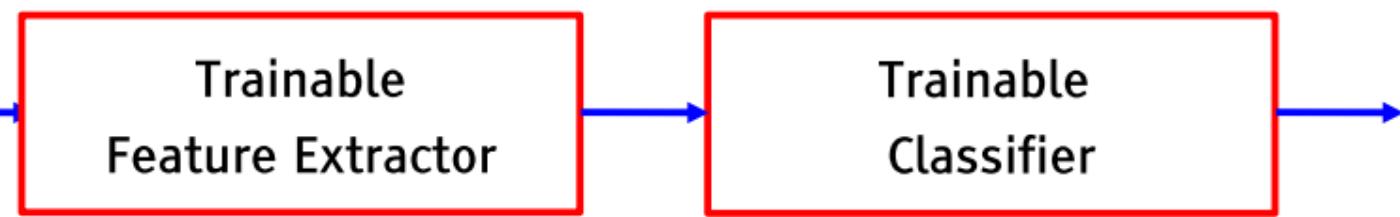
■ The traditional model of pattern recognition (since the late 50's)

- ▶ Fixed/engineered features (or fixed kernel) + trainable classifier



■ End-to-end learning / Feature learning / Deep learning

- ▶ Trainable features (or kernel) + trainable classifier



Deep learning = learning representations / features

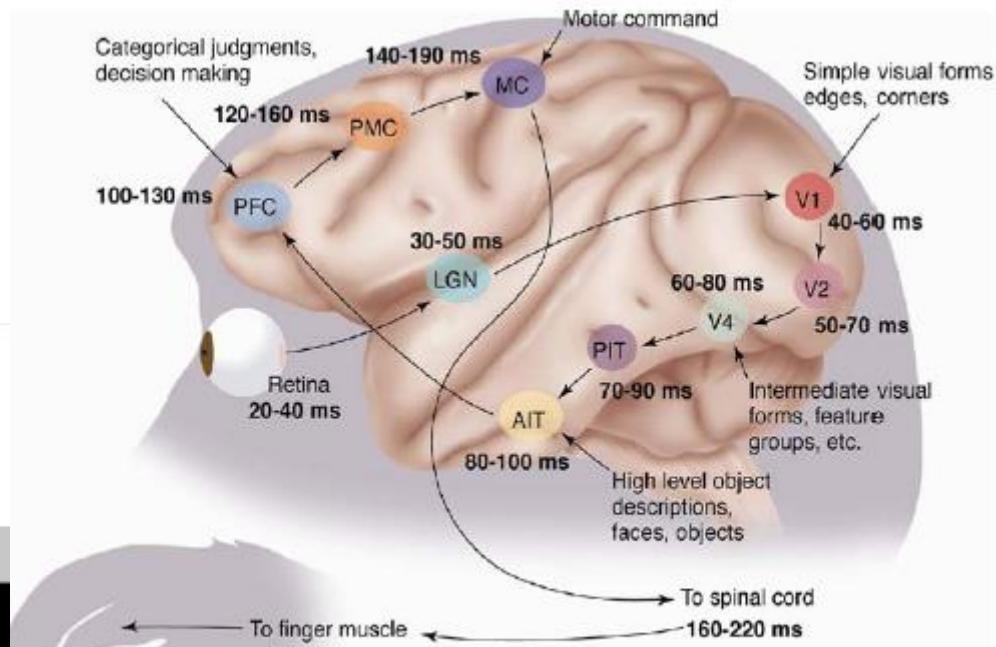
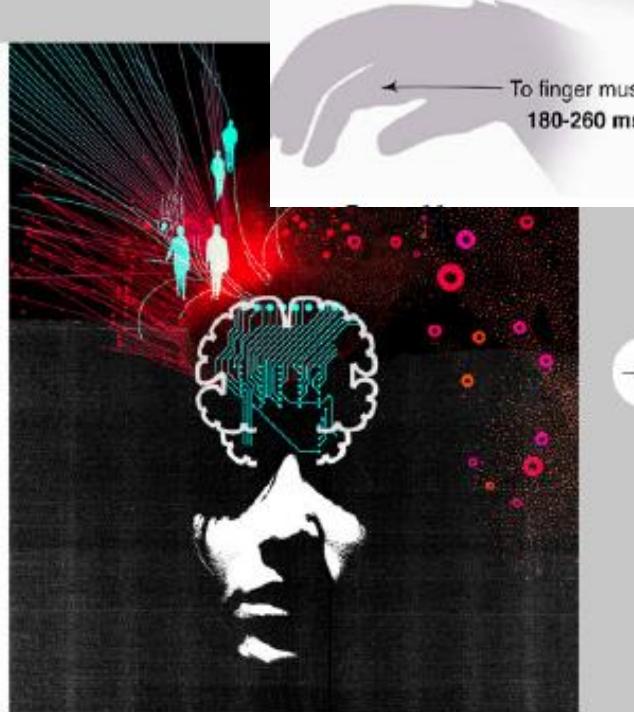
Why studying deep learning?

- Breakthrough technology...!



Deep Learning

With massive amounts of computational power, machines can now recognize objects and translate speech in real time. Artificial intelligence is finally getting smart.

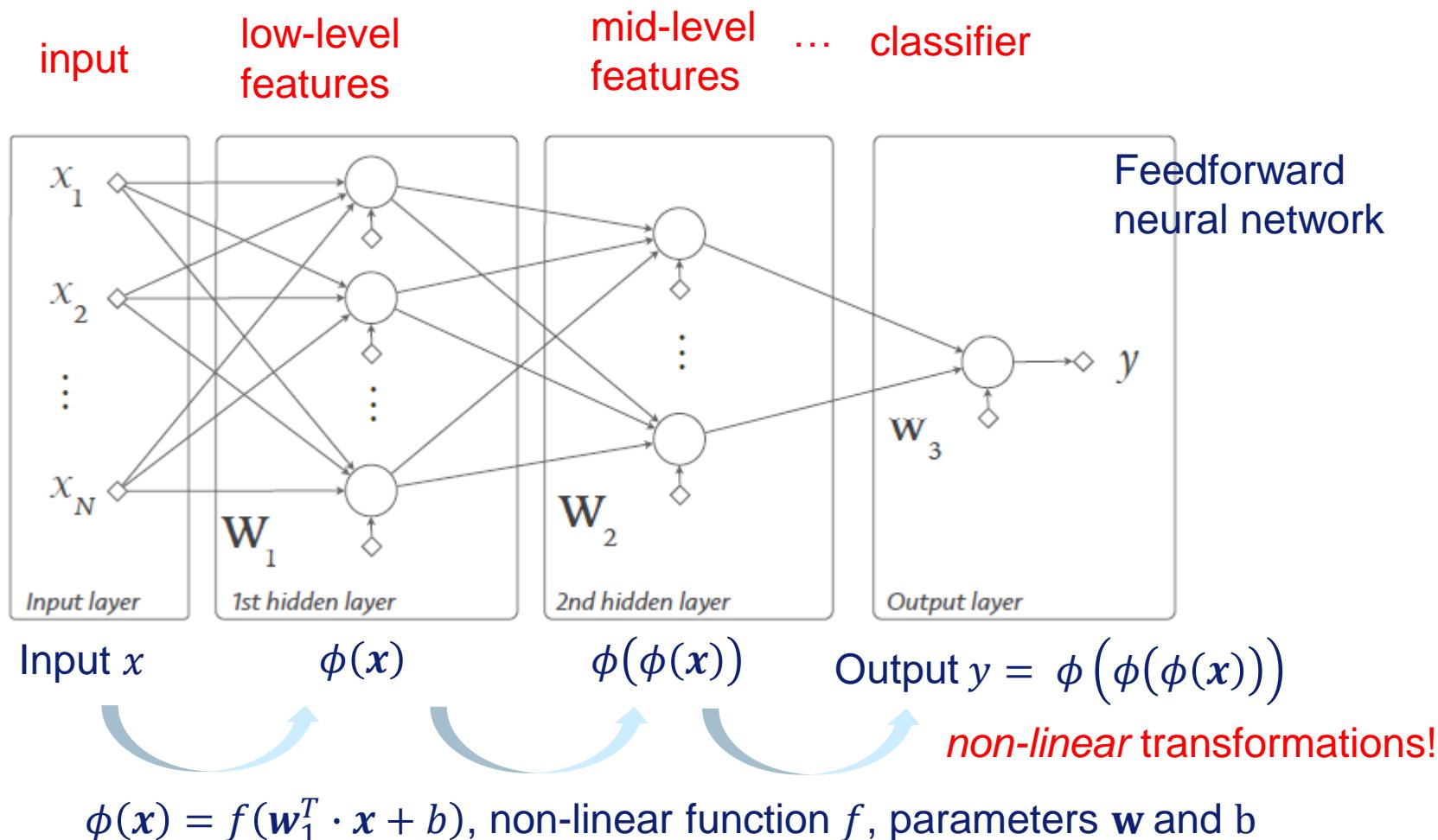


[picture from Simon Thorpe]

Introduction

- What is the **relationship between neural networks and deep learning?**
- Deep learning mainly originated in the context of **artificial neural networks**
 - e.g. Multi-layer perceptron with „many“ hidden layers
 - Often with some **additional features**, e.g.
 - new architectures (convolutional instead of fully connected layers),
 - more „tweaks“ (unsupervised pre-training, dropout, batch normalization...)
 - much more training data and better hardware (GPUs)
 - But deep models other than feedforward networks also exist
 - E.g. bi-directional (Deep Boltzmann Machines, Stacked Auto-Encoders)
- The term „deep learning“ changed from a **feature of neural networks** to a **collection of models** based on learning hierarchical data representations
 - „Deep learning“ not only refers to neural networks (e.g. graphical models)

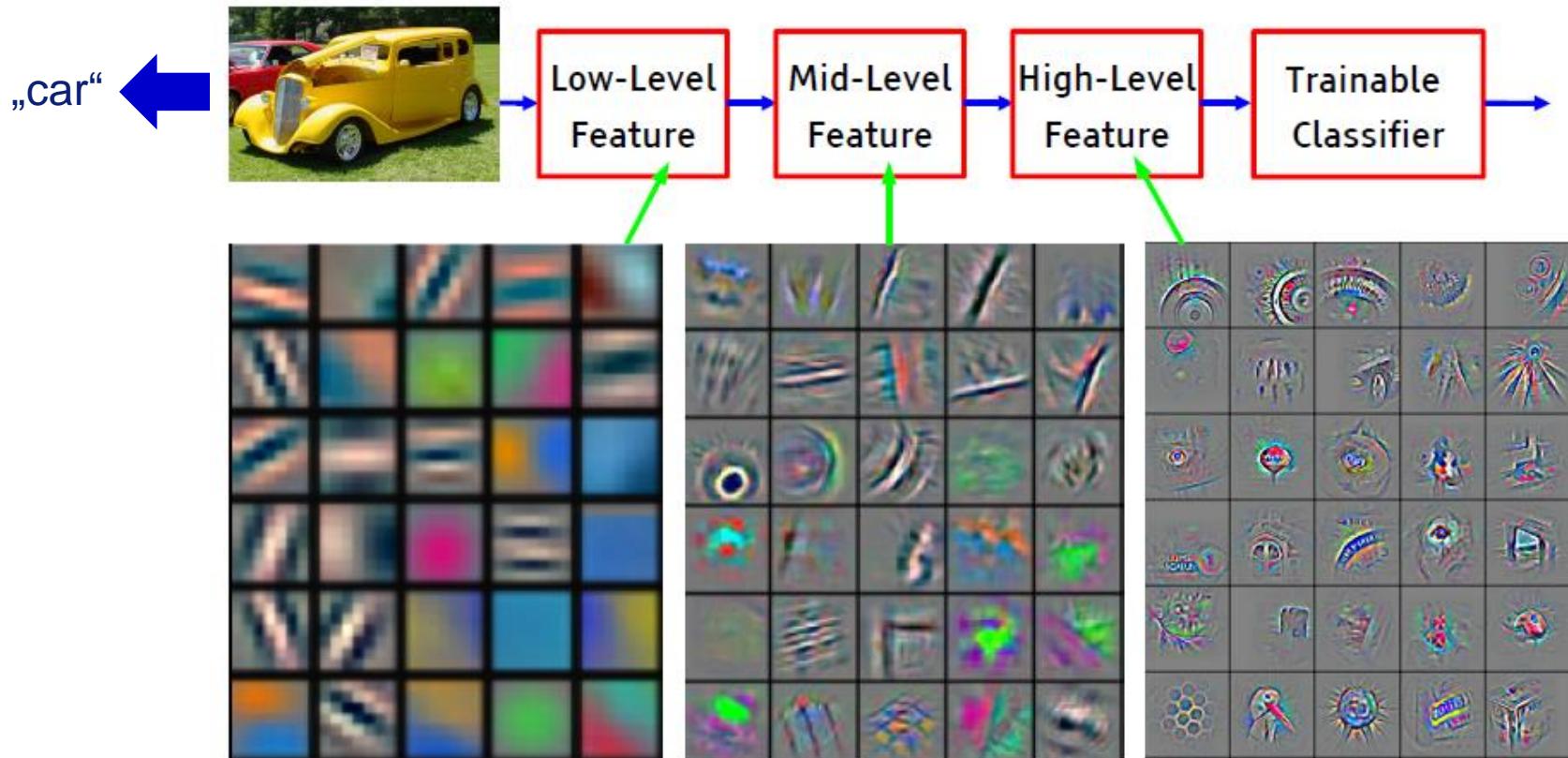
Illustration: Feedforward neural network and deep learning



- Idea: Subsequent layers **learn** increasingly „higher“ input representations ϕ („features of features of features...“) \rightarrow „deep“ model

Application example of deep learning (1): Computer vision

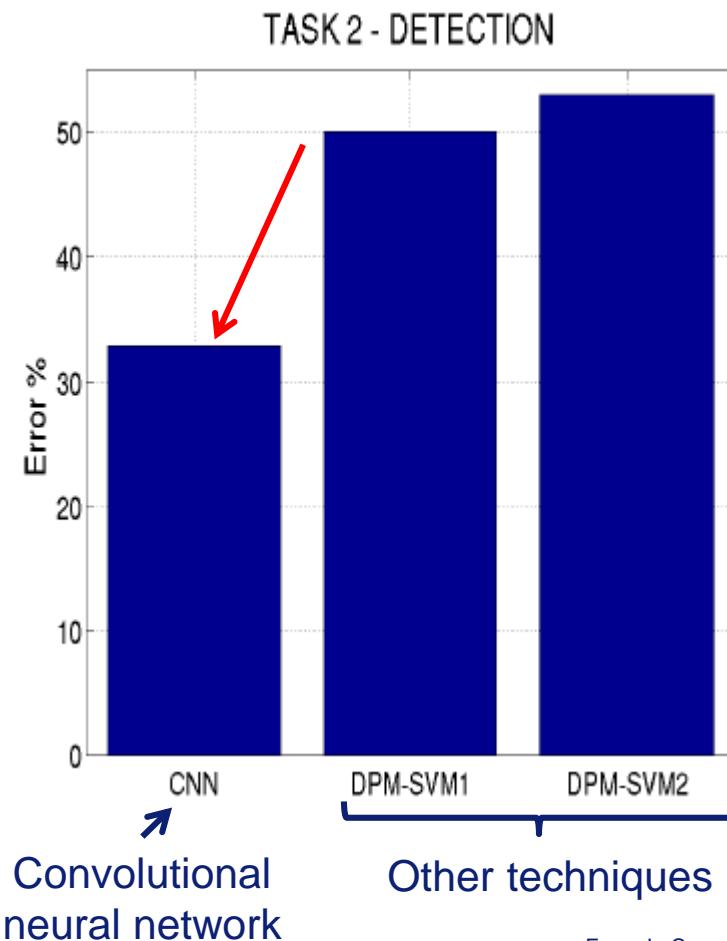
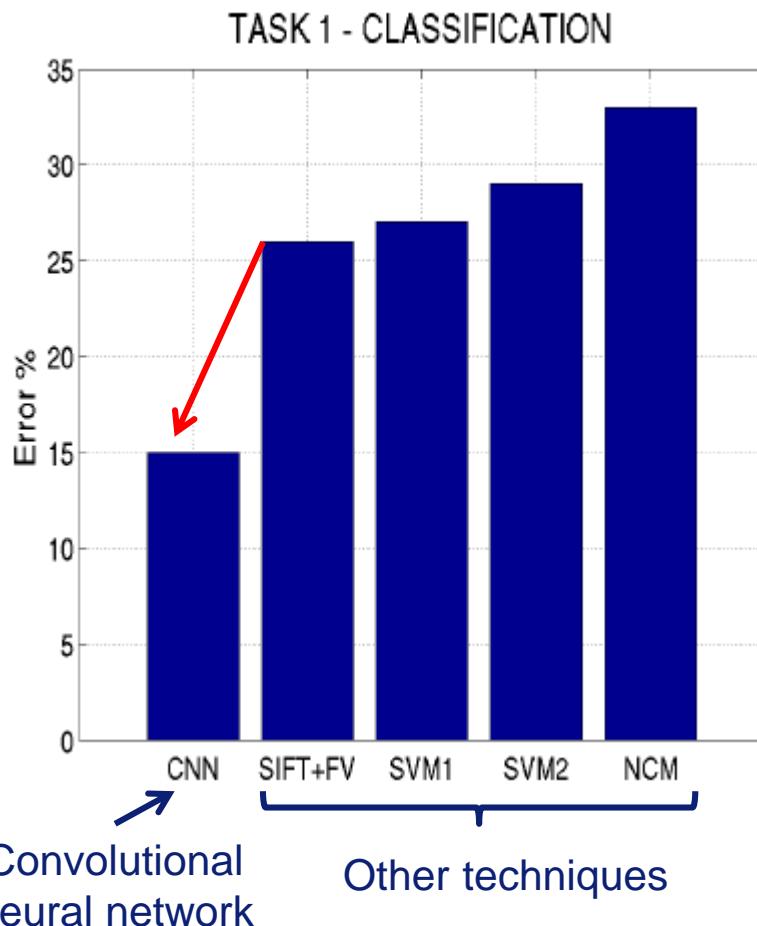
- Task: Image classification (1000 object categories; 5 guesses)
 - 2012 rank 1: SuperVision (deep convolutional neural network): 0.153% error
 - 2012 rank 2: ISI (various features, fisher vectors, linear classifier): 0.262% error
 - 2013: top 17 placed teams used deep convolutional networks (except rank 13)



Feature visualization of convolutional net trained on ImageNet (Zeiler & Fergus, 2013)

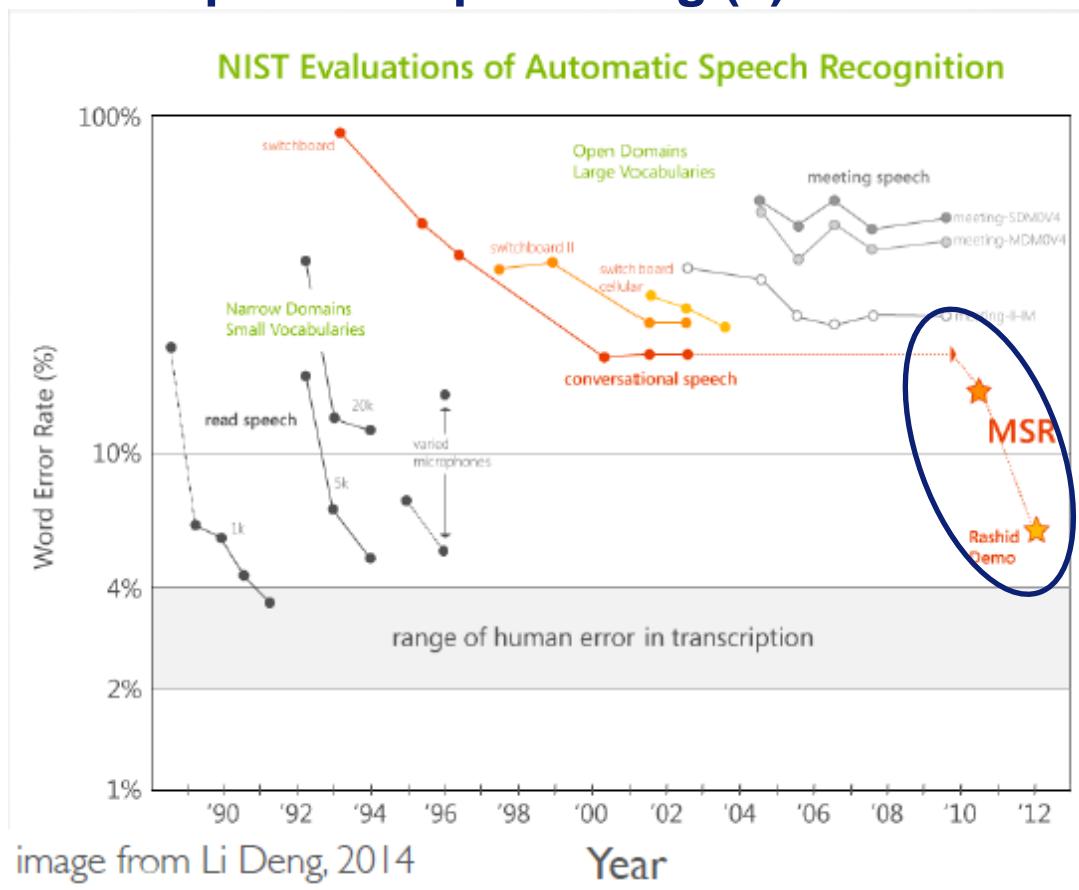
Convolutional neural networks: Evaluation

- ImageNet large scale visual recognition challenge (ILSVRC) 2012
- 1000 categories, 1.5 million labeled training samples



From: LeCun

Application example of deep learning (2): Automatic speech recognition

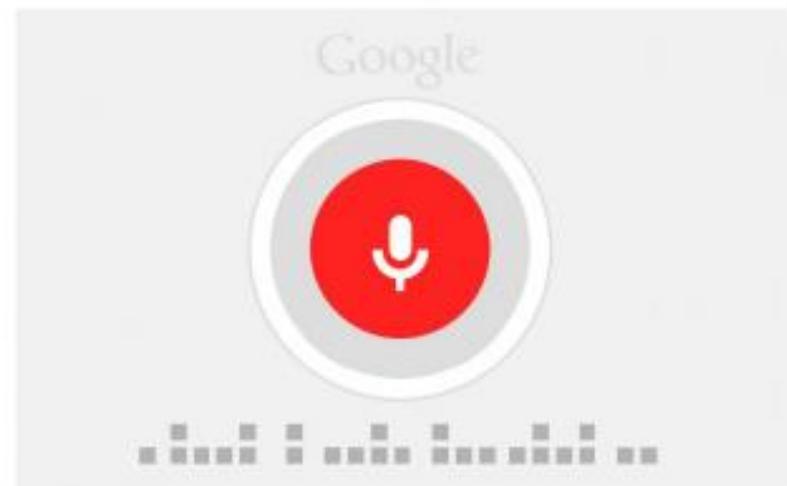


- Deep learning reduced word error rate from $\approx 23\%$ to $< 15\%$ (!!)
- Recently, unsupervised pretraining appears to be irrelevant
- But: Big system trained on 2000h of data, needed 59 days on GPUs (!)
- Extension: Multilingual speech recognition

From: Larochelle

Application example of deep learning (2): Automatic speech recognition

- You're already using deep learning!

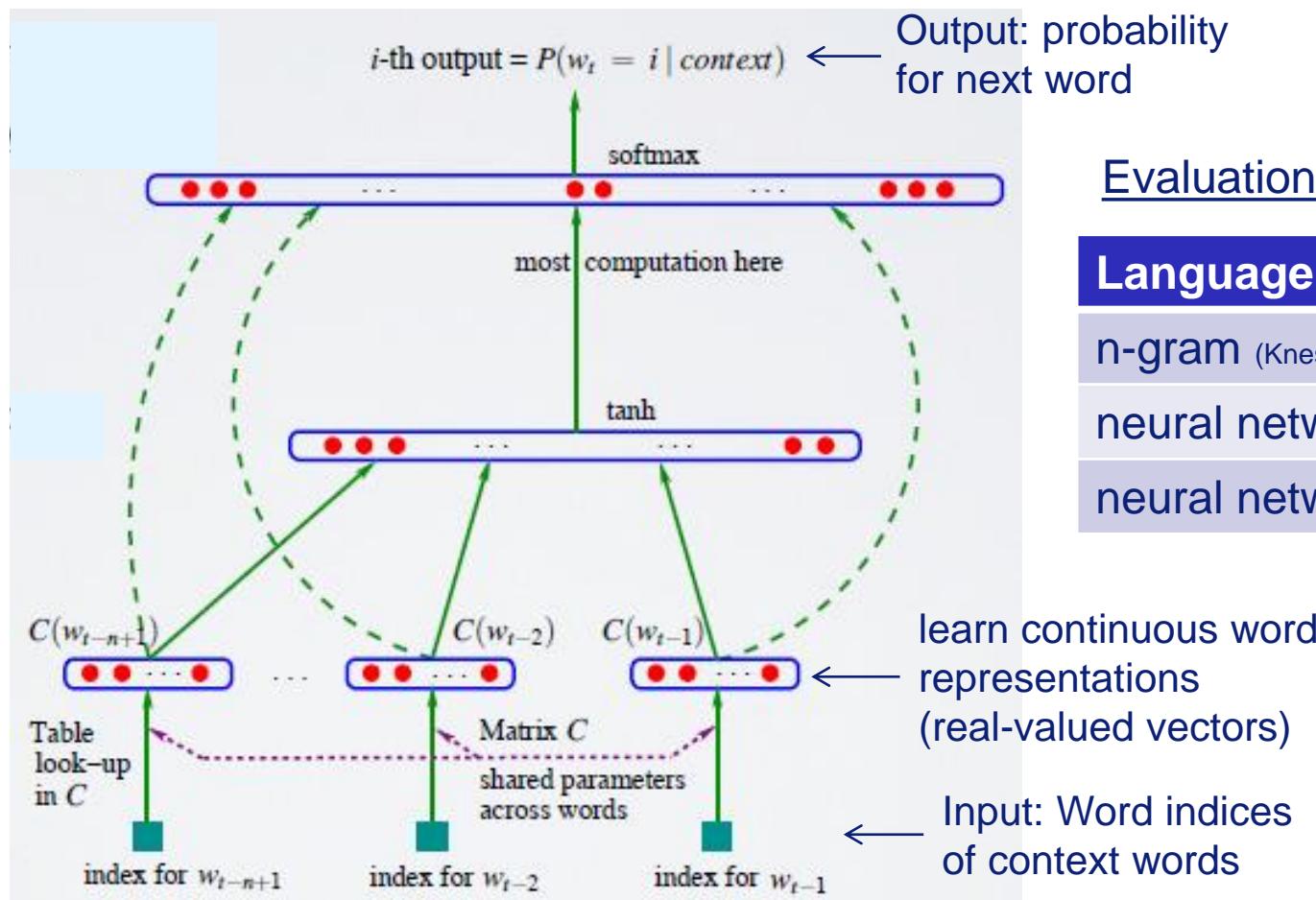


From: Cho

41

Application example of deep learning (3): Natural language processing

- Task: Incrementally predict probability for next word given predecessors: „Language model“; evaluated with metric „perplexity“ PP
 - „Effective vocabulary size“ under language model; the smaller, the better



Evaluation on Brown corpus:

Language model	PP
n-gram (Kneser-Ney smoothing)	321
neural network	276
neural network + n-gram	252

From: Larochelle

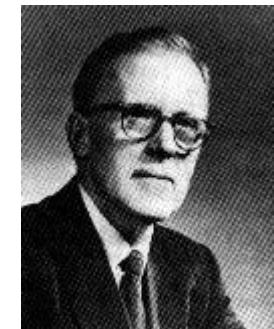
More application examples of neural networks and deep learning

- Visual recognition
 - Objects, faces, characters
- Speech and language processing
 - Automatic speech and speaker recognition, natural language processing
 - Machine translation
- Autonomous systems
 - Localisation in space, planning, avoiding obstacles
- Industrial process optimization, planning and control
 - Control engineering
 - Coordination of robotic arm
- Filtering, Clustering
 - Data coding, data compression
- Regression, data analysis
 - Time series prediction
 - Medical decision support

Brief history of neural networks

Beginning

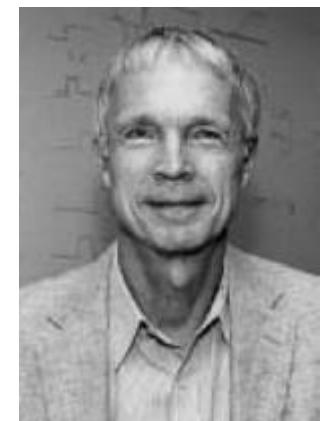
- **1943:** Warren McCulloch and Walter Pitts define some kind of a first neural network; neurons as logical elements
- **1949:** Formulation of Hebbian learning (**Donald O.Hebb**)
- **1957-1958:** Perceptron described by Frank Rosenblatt



Donald O.Hebb

Disillusion

- **1969:** Marvin Minsky and Seymour Papert investigate the Perceptron mathematically and demonstrate its limits e.g. regarding the XOR-problem



Renaissance

- **1982:** Description of the first self-organizing maps (along the lines of *biology*) by Teuvo Kohonen and publication of a seminal article by **John Hopfield** describing the first Hopfield networks (along the lines of *physics*)
- **1986:** The backpropagation learning algorithm is developed for multi-layer perceptrons

Hype

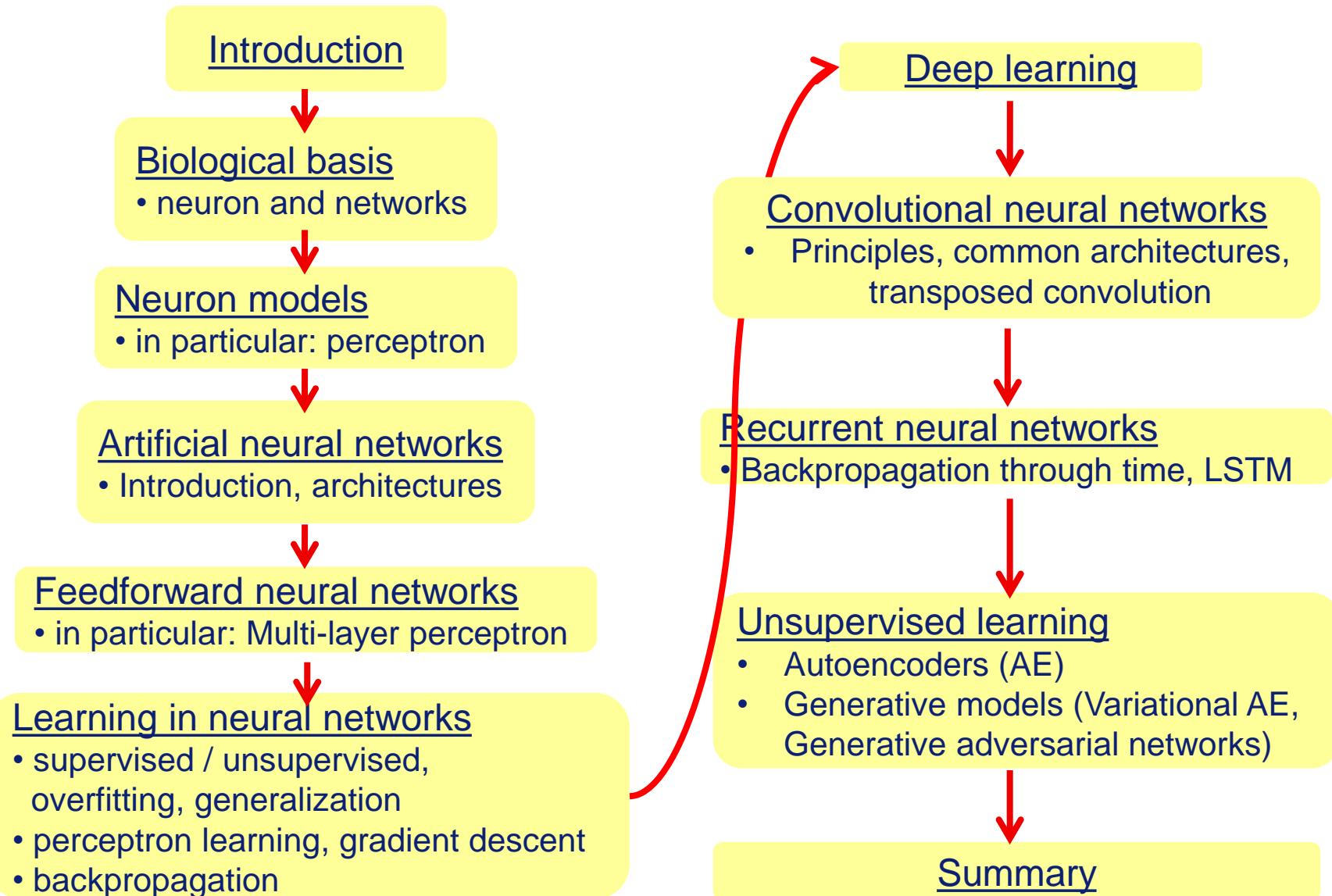
- ... but: „shallow learning“ prevails (e.g. support vector machines)
- **Since 2006:** Major breakthrough with „deep learning“

From: Hartmann

Important issues in neural networks

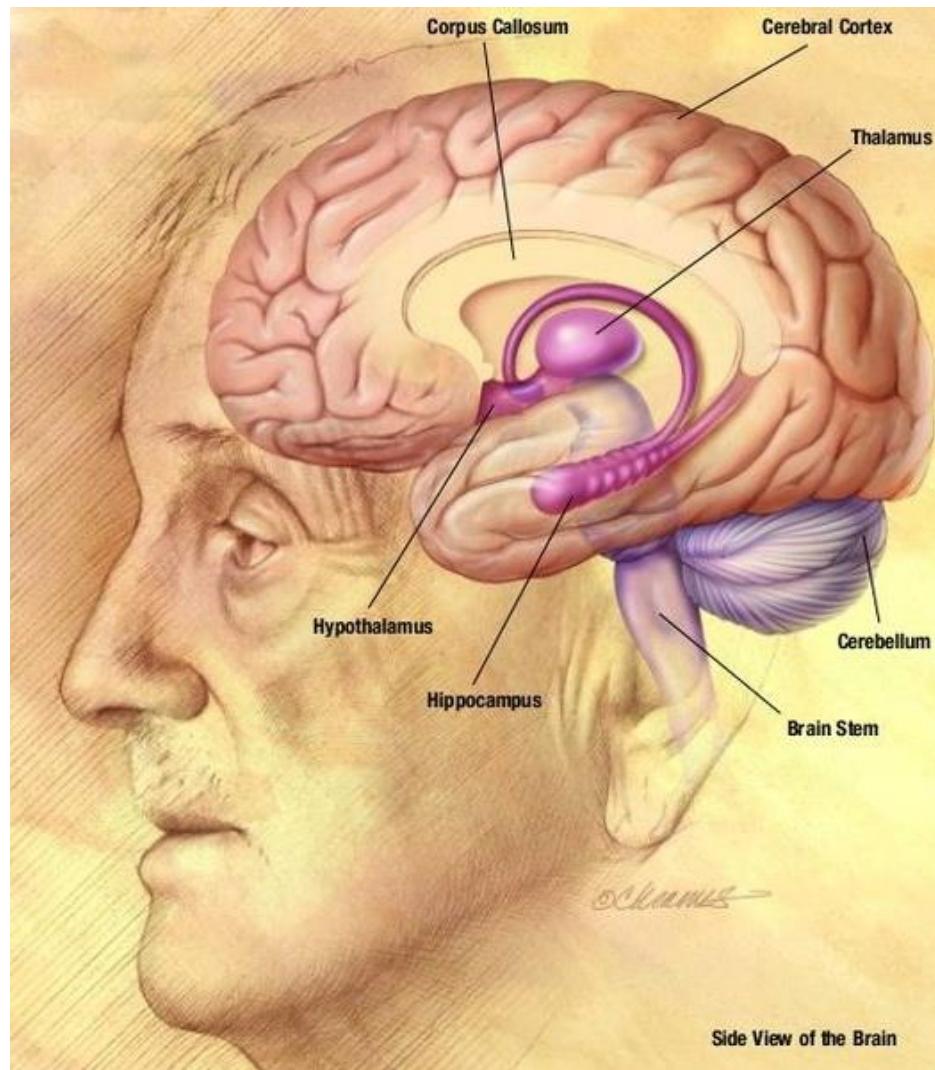
- „Computation“:
 - Given a neural network and an input pattern: What is the network response to the input (i.e. what is the network output)?
 - What is the computational ability of an artificial neuron, i.e. which input / output relations can be realized and which cannot?
- „Learning“:
 - How to choose network parameters so that the network realizes desired input / output relations?
- „Typical properties“:
 - What are typical properties of neural networks (e.g. dynamical and asymptotic properties in recurrent neural networks)?
 - What is the effect of noise, removal of neurons or connections?

Neural networks and deep learning: Overview of lecture



BIOLOGICAL BASIS

2. Biological basis: Basic anatomy of the human brain



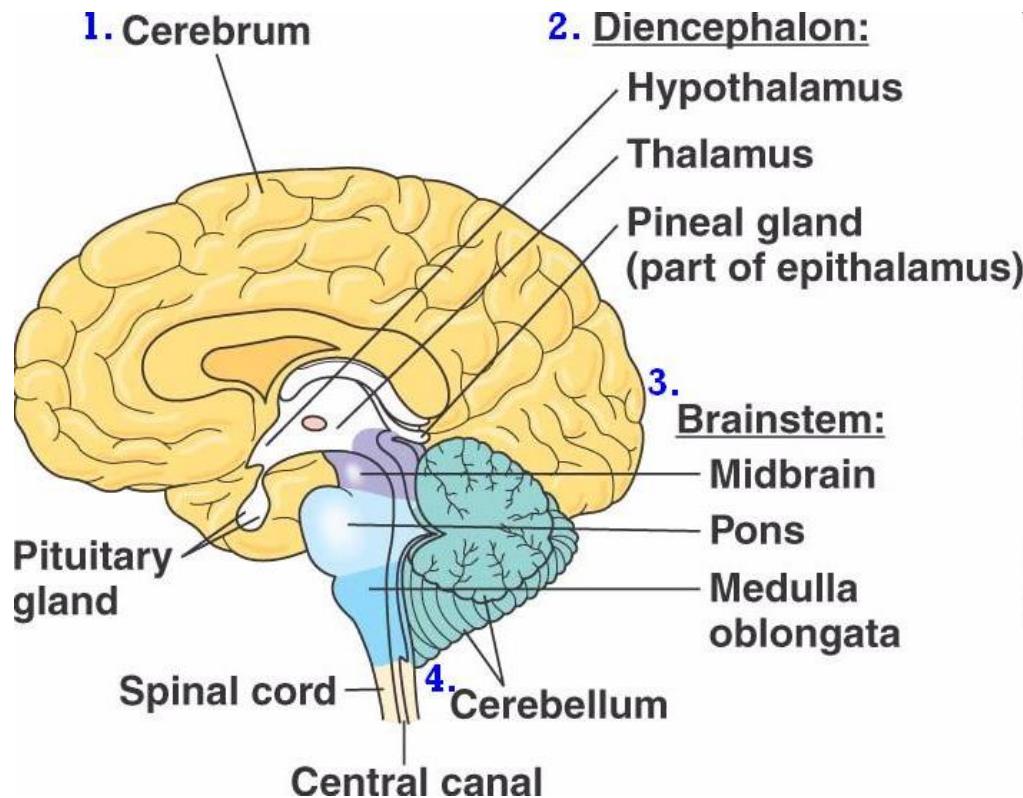
http://en.wikipedia.org/wiki/File:NIA_human_brain_drawing.jpg

Regions of the brain:

- **Cerebrum (telencephalon)**
 - responsible for abstract thinking
 - outer part: **cerebral cortex**
- **Diencephalon (interbrain)**
 - controls fundamental physiological processes
 - thalamus, hypothalamus etc.
- **Brainstem**
 - connects the brain with the spinal cord, controls reflexes
- **Cerebellum**
 - controls and coordinates motor functions

2. Biological basis: Some basics of the human brain

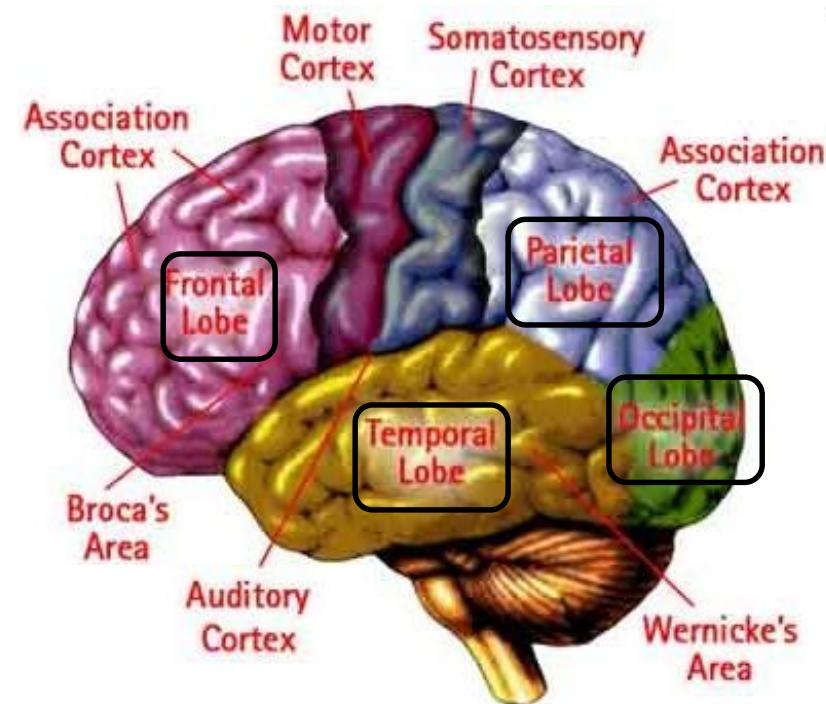
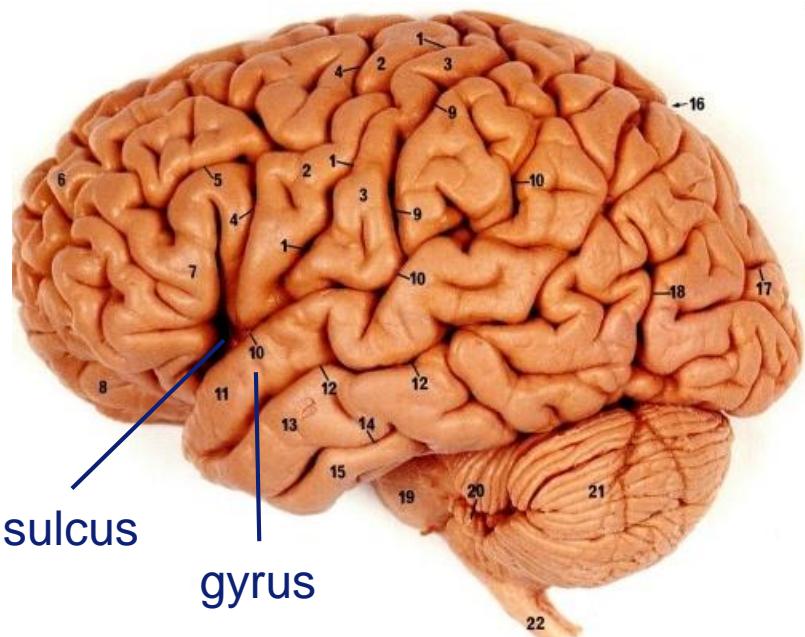
- Regions of the human brain:



http://www.nicerweb.com/bio1152/Locked/media/ch49/49_09bBrainDevelopmtAdlt-L.jpg

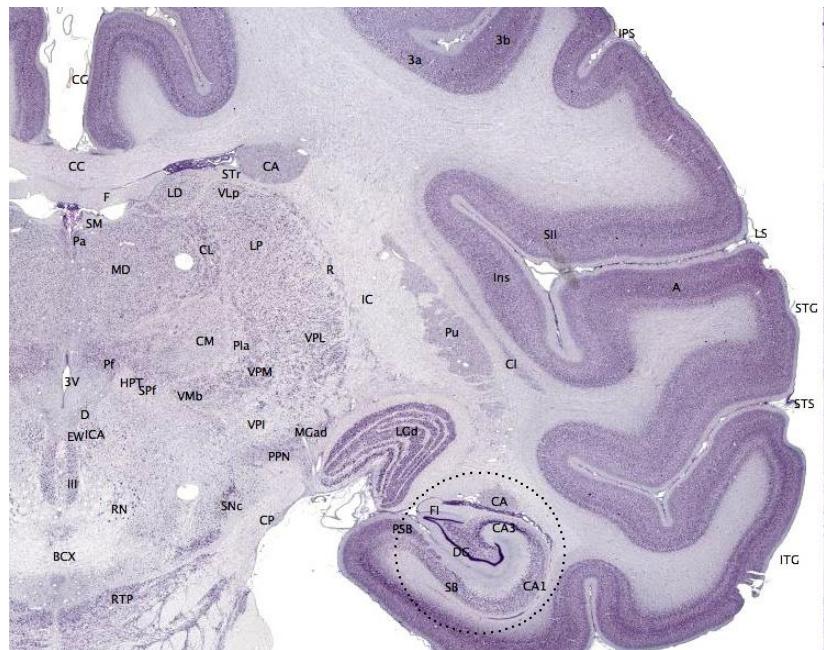
The cerebral cortex (1)

- Outer part of the cerebrum (highly convoluted, 2-4mm thick, area: 1.5m^2)
- Composed of **gyri** and **sulci**
- divided into **lobes**: frontal lobe, parietal lobe, occipital lobe, temporal lobe

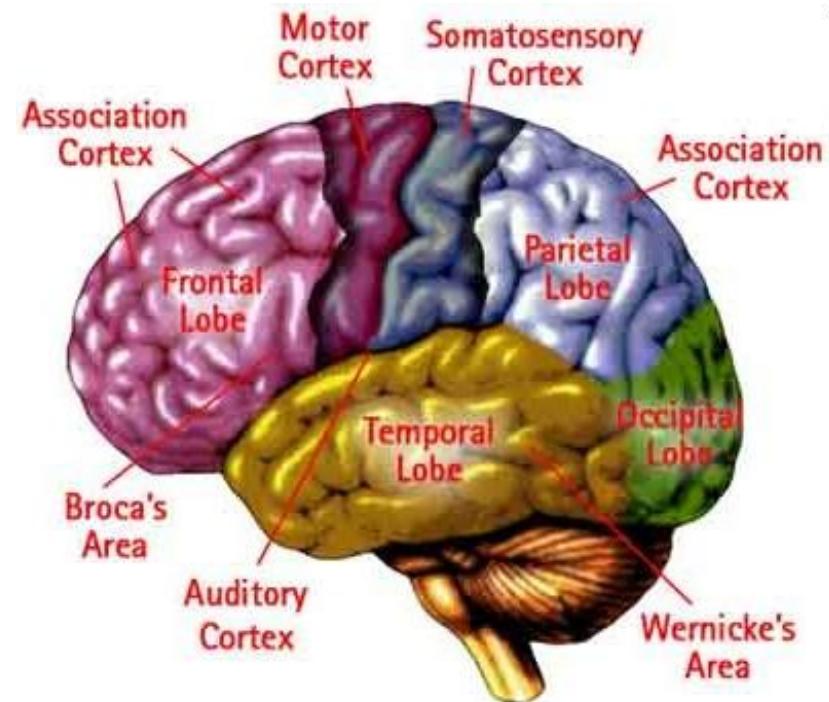


The cerebral cortex (2)

- Functional organisation into different **cortical areas with specific tasks**
 - primary areas: visual, auditory, sensory, motor cortex; information processing
 - association areas: more abstract association and thinking; memory
- key role in memory, attention, awareness, thought, language, consciousness



Cerebral cortex: outer layer (dark violet)

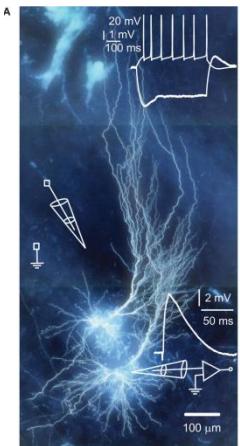


http://dericbownds.net/uploaded_images/cortex.jpg

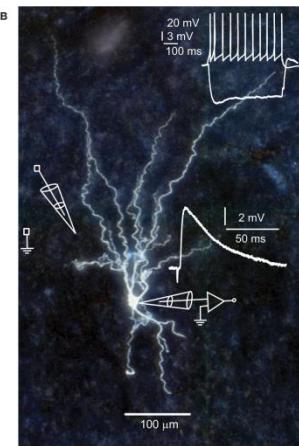
From: <http://en.wikipedia.org/wiki/File:Brainmaps-macaque-hippocampus.jpg>

The cerebral cortex: Cell types

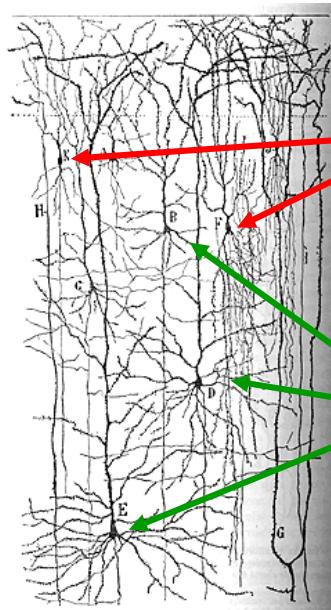
- **Nerve cells (Neurons)**
 - about 10 - 100 billion nerve cells ($10^{10} - 10^{11}$)
 - about 10.000 connections per nerve cell (in total: $10^{14} - 10^{15}$ connections)
 - different types (according to the shape of their cell body):
 - pyramidal cells: pyramid cell bodies, large
 - non-pyramidal (granular/stellate cells): small, local projections, $\approx 25\%$ of neurons
 - In addition: interneurons (communicate only with adjacent neurons)



pyramidal neuron



non-pyramidal neuron



non-pyramidal neuron

pyramidal neuron

- **Glia cells**
 - provide support and protection for the neurons

The cerebral cortex: Cortical layer structure

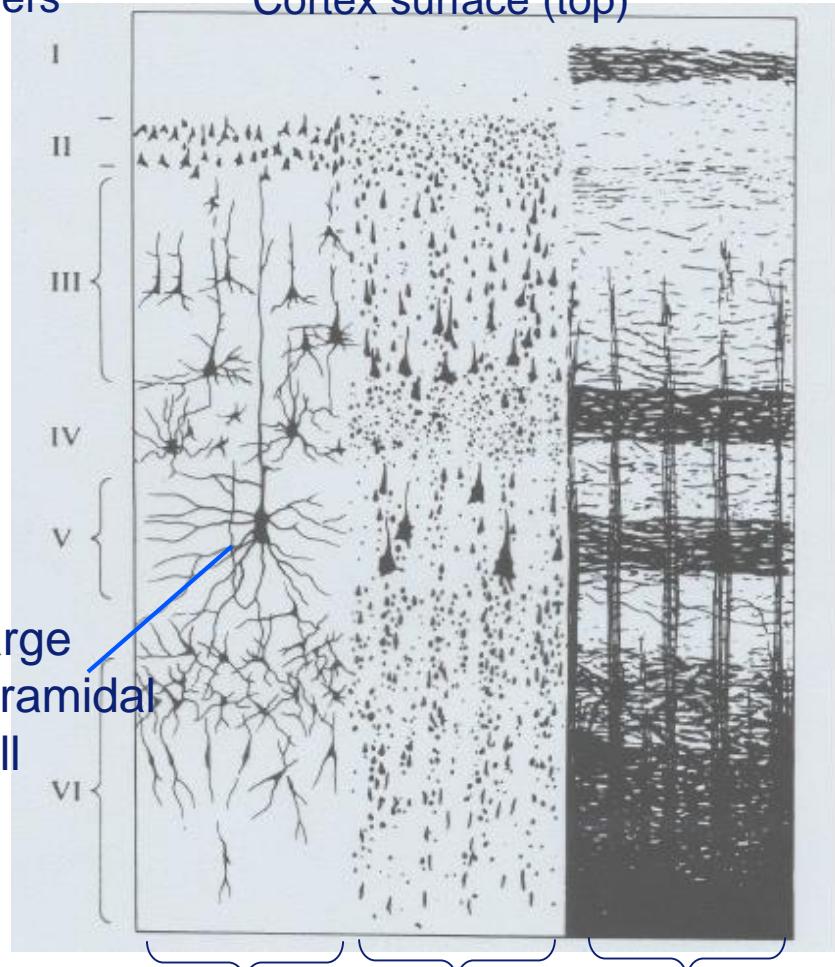
(according to cell type composition and fiber organisation)

Layers

Cortex surface (top)

From: Schwenker
I
II
III
IV
V
VI

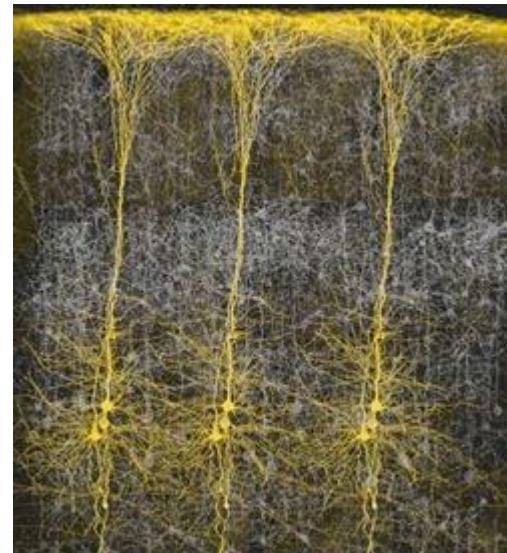
Large
pyramidal
cell



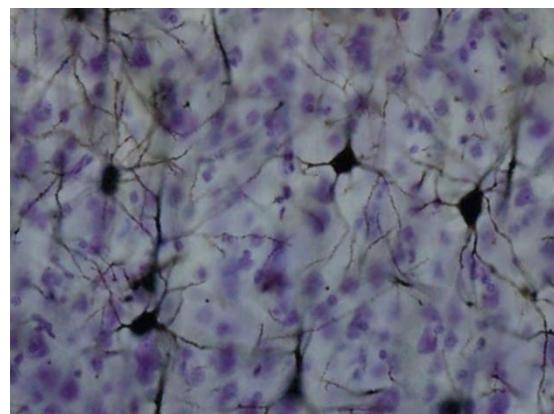
Neurons
with fibers

Distribution
of cell bodies

Organisation
of nerve fibers



Neurons in the neocortex

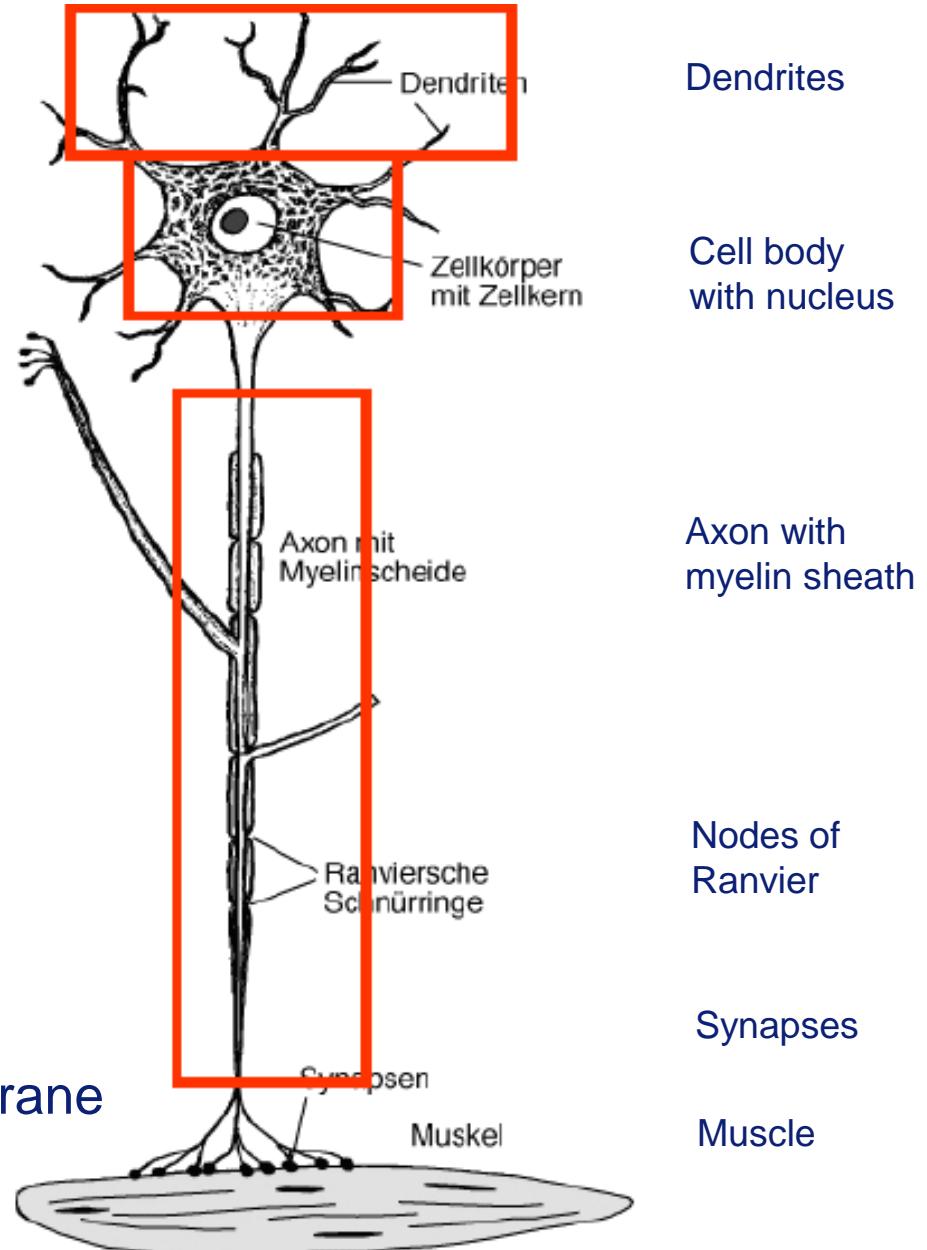


Neurons in the somatosensory cortex

The neuron: Structure

The neuron consists of:

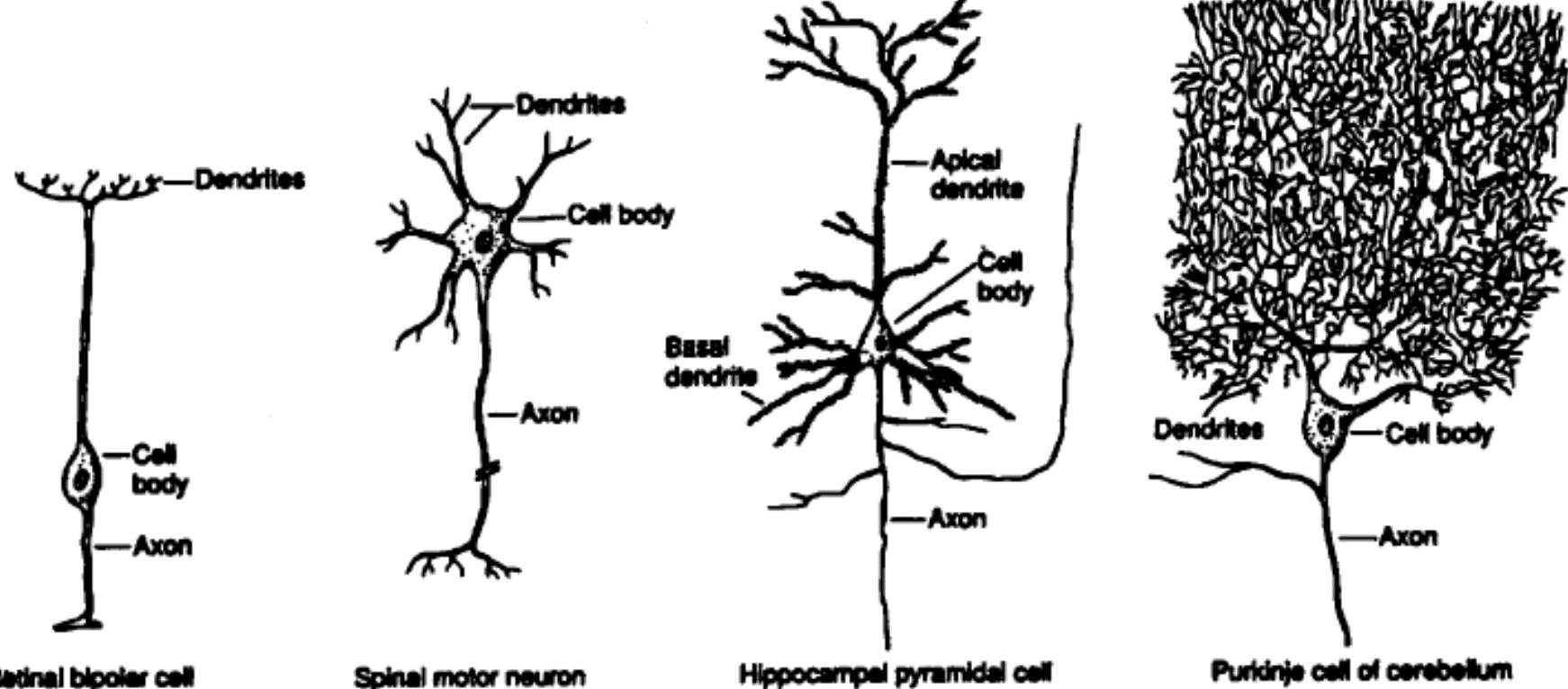
- **Cell body (soma)**
 - **dendrites / dendritic tree (input)**
 - one **axon** (output; potentially branching at the end)
 - axon connects via **synapses** to dendrites of other neurons
-
- ion channels in axon cell membrane maintain **electrical potential**



From: Schwenker

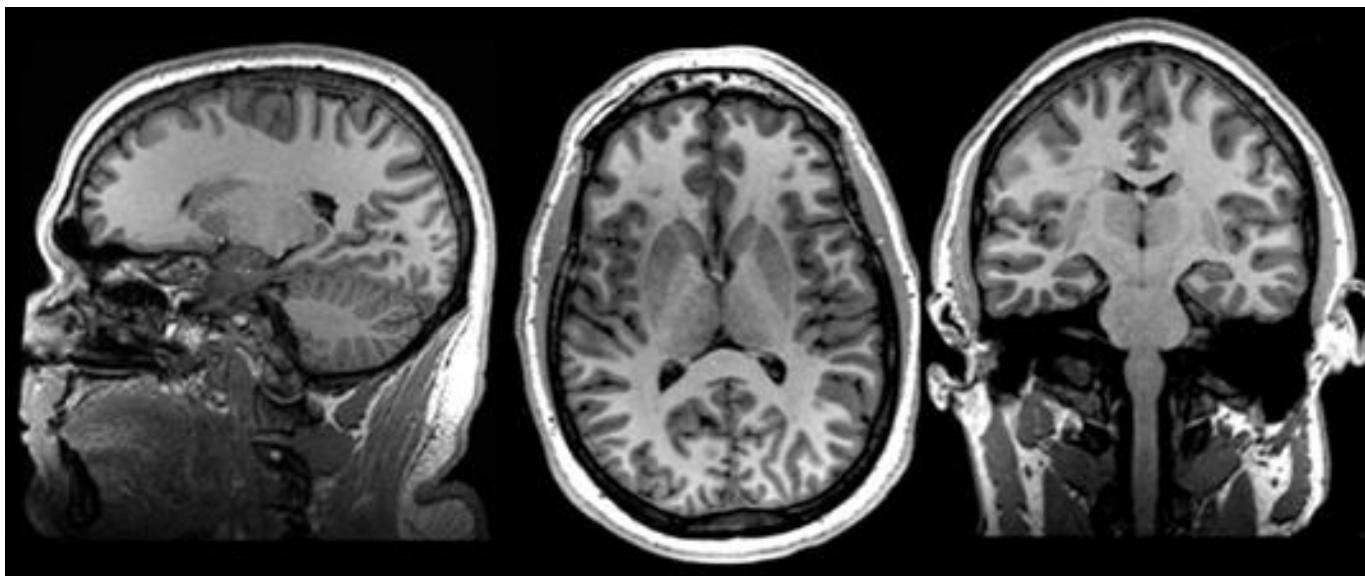
The neuron: Structure

- Different types of neurons
 - All with one axon



From: Lippe

Gray matter and white matter

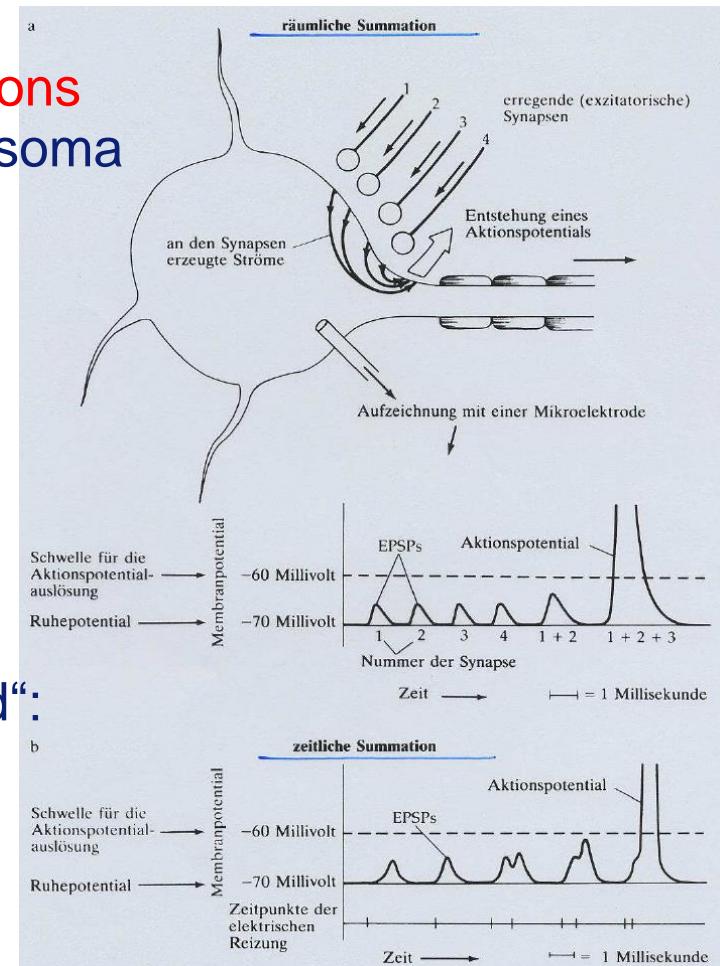


Source: Ron Killiany, ADNI

- **Gray matter**
 - neuron (nerve cells) bodies and unmyelinated fibers
 - Cerebral cortex is part of gray matter
- **White matter**
 - myelinated axons (neuron connections to different regions of the cerebral cortex or to other parts of the central nervous system)

The neuron: Function

- Neuron membrane maintains **electrical potential**
- **Dendrites receive impulses from other neurons leading to postsynaptic potentials (PSP) at soma**
 - excitatory PSP (EPSP): potential increased
 - inhibitory PSP (IPSP): potential decreased
- At cell body: **Spatiotemporal integration** of incoming impulses (PSPs) of all dendrites
 - integration time: msec – several seconds
- If (integrated) PSP exceeds „firing threshold“: release of **action potential (AP; spike)**
 - AP propagates along axon
 - after AP: **refraction** („neuronal recovery“)

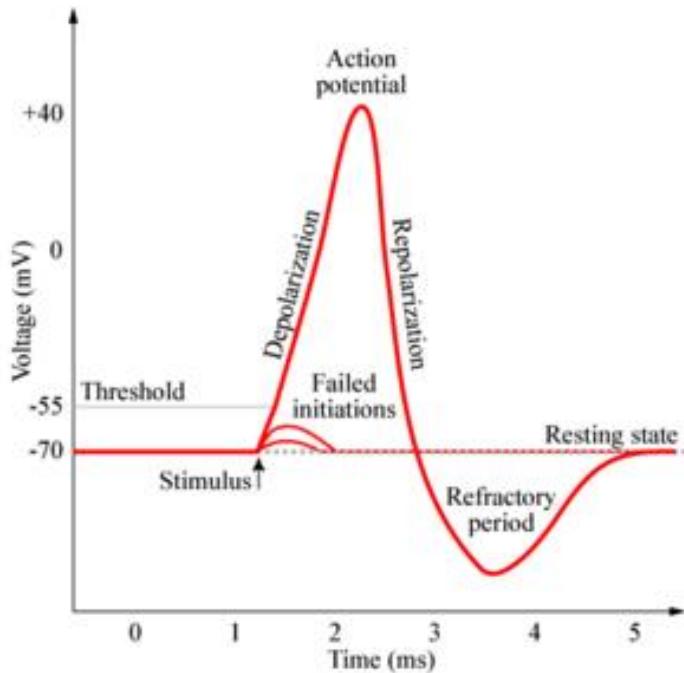


From: Schwenker

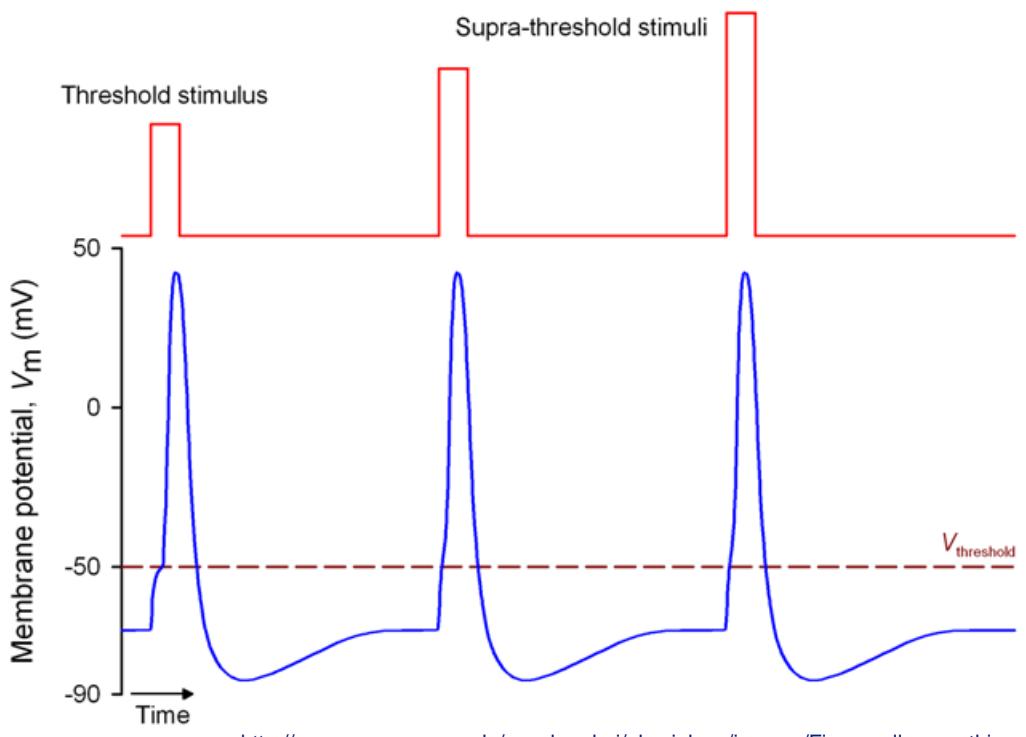
The action potential (AP, „spike“)

- all-or-none (binary) event; duration: $\approx 1\text{msec}$
- larger stimulation \rightarrow influences only spike frequency (not shape)

Action potential:
Generation and phases



Form of action potential **independent** of the stimulus size (if above threshold)



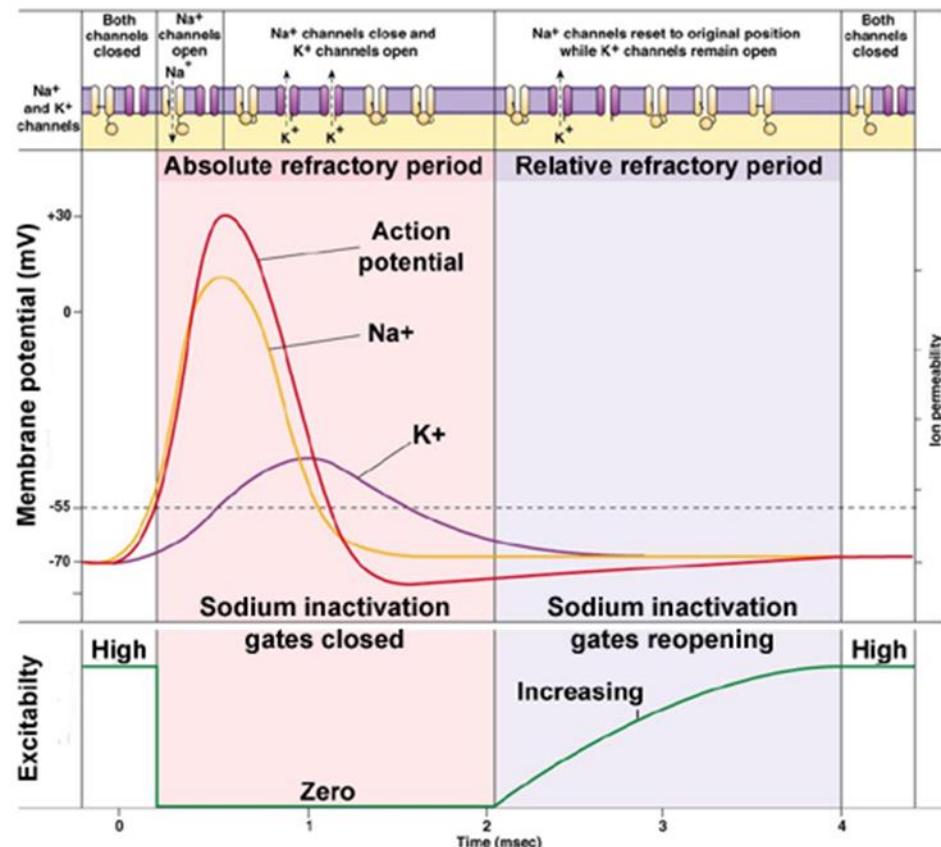
http://www.csupomona.edu/~seskandari/physiology/images/Figure_all_or_nothing.gif

Refractory period

After release of action potential (AP): Refraction (restauration of resting state)

- **absolute refractory period:**
no further spike release
(second AP *cannot* be initiated)
 - duration: 1-2 msec
- **relative refractory period:**
threshold increased
→ second AP *inhibited*,
but not impossible
 - duration: additional few msec

„neuronal recovery“



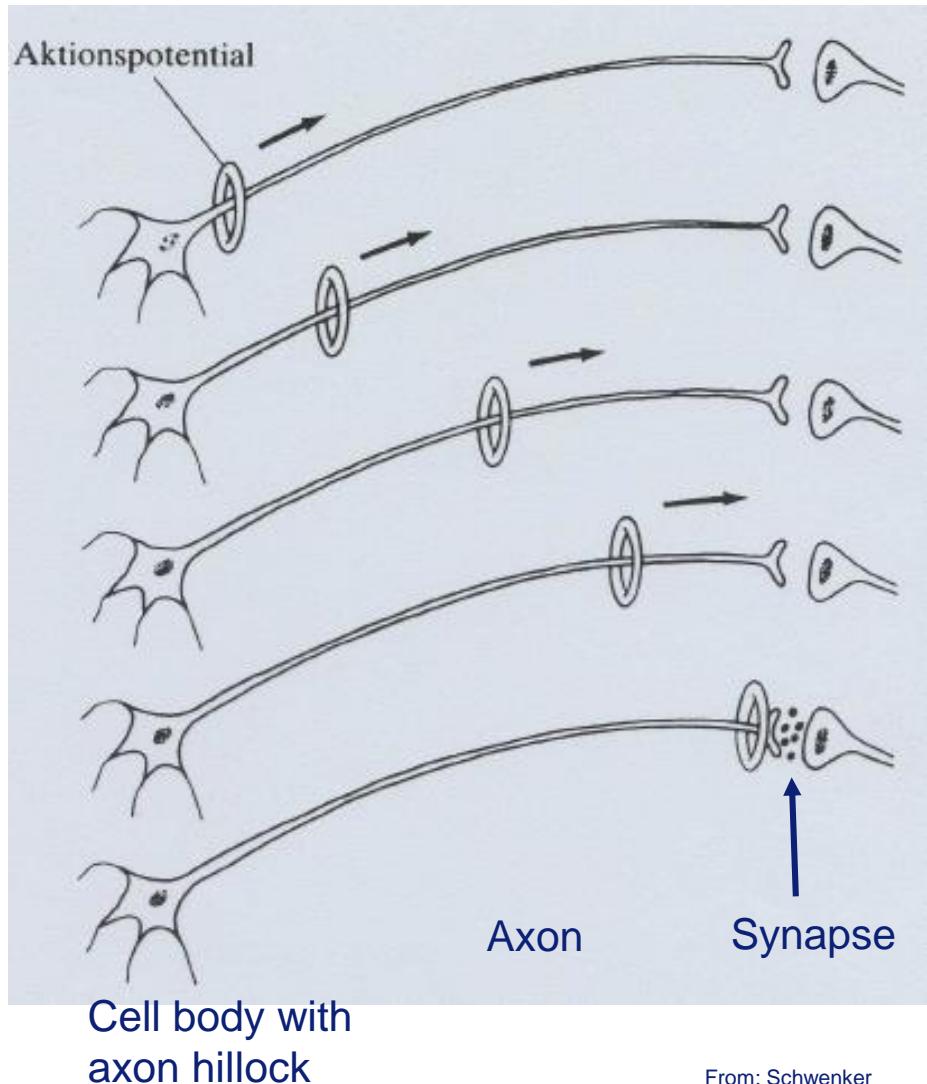
Copyright © 2007 Pearson Education, Inc., publishing as Benjamin Cummings.

Fig. 8-12

<http://www.colorado.edu/intphys/Class/IPHY3430-200/image/08-12.jpg>
<http://www.colorado.edu/intphys/Class/IPHY3430-200/image/08-12.jpg>

Action potential propagation

- 1.) Release of action potential (AP) at axon hillock
- 2.) Propagation of AP along axon with constant amplitude
 - transmission velocity:
10-100 m/sec (myelinated axons),
< 1m/s (unmyelinated axons)
- 3.) Branching of AP at end of axon (axonal tree)
- 4.) Release of neurotransmitter at (chemical) synapse

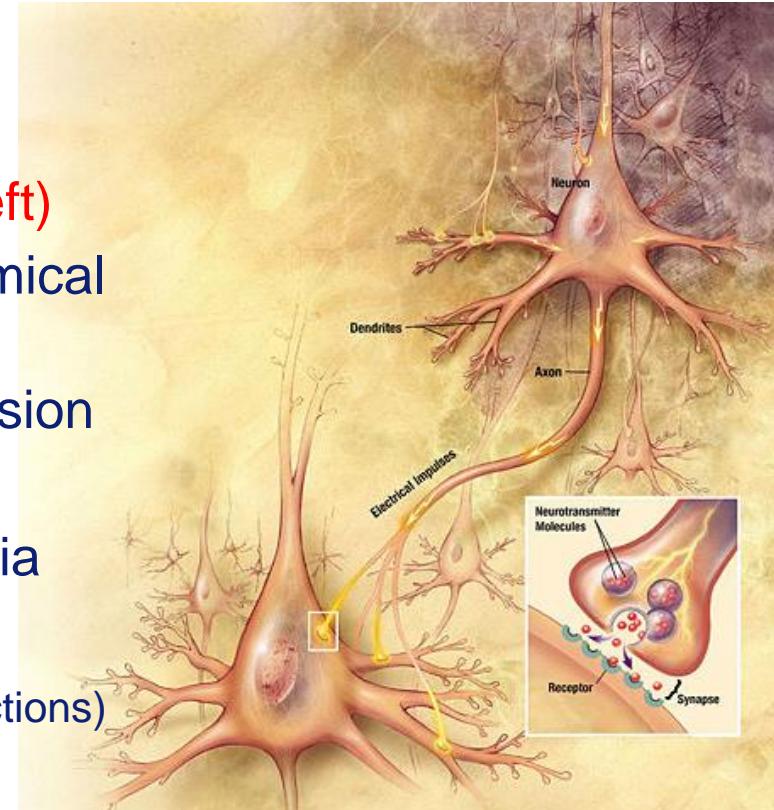


From: Schwenker

Synaptic transmission

Signal (AP) from axon of neuron A transmitted to dendrites of neuron B via (chemical) **synapses**:

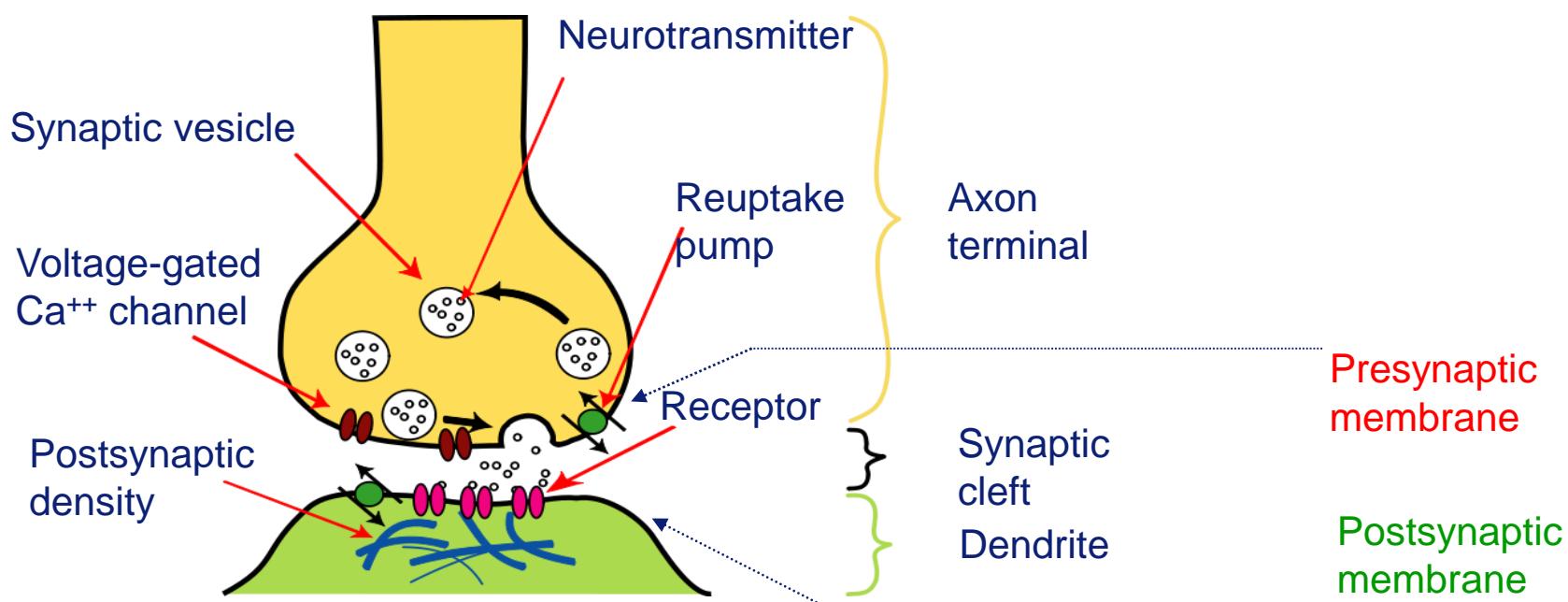
- **connections between neurons**
- neurons **physically separated (synaptic cleft)**
- signal transmission via (≈ 30 types of) **chemical neurotransmitters**
- conversion **electrical \rightarrow chemical** transmission
- synaptic transmission: ≈ 5 msec
- **integration of signals** due to chemical inertia
- Less frequent: **Electrical synapses** (gap junctions)
- **Information transmission:**
 - **divergence:** one neuron to many (output)
 - **convergence:** many neurons to one (input)



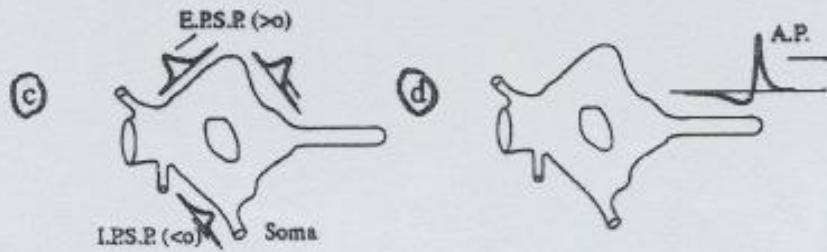
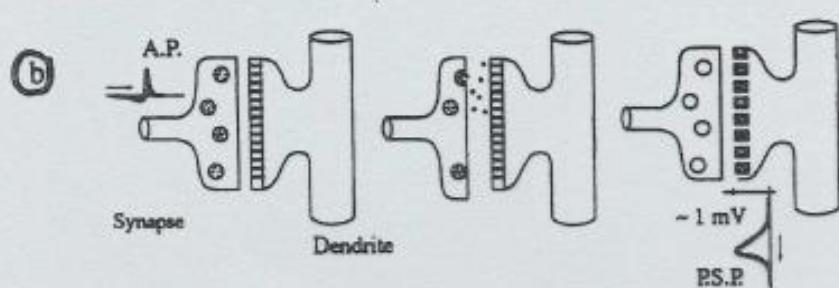
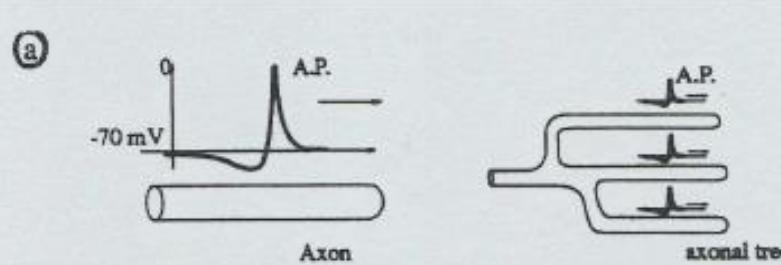
<http://svr225.stepx.com:3388/action-potential/file/78242.jpg>

Structure of a (chemical) synapse

- upon arrival of AP at presynaptic membrane: **release of neurotransmitter into synaptic cleft**
- neurotransmitter then binds to receptors at postsynaptic membrane**
 - depending on receptor type, Na^+ (Cl^-) ion channels open → influx of positive (negative) ions → **excitatory (inhibitory) postsynaptic potential**, respectively
 - electrical signal propagates along dendrite to cell body
 - leads to excitatory (inhibitory) postsynaptic potential at axon hillock



Cycle of neuronal dynamics



a) Propagation of action potential along axon

b) Synaptic transmission

c) Spatiotemporal input integration

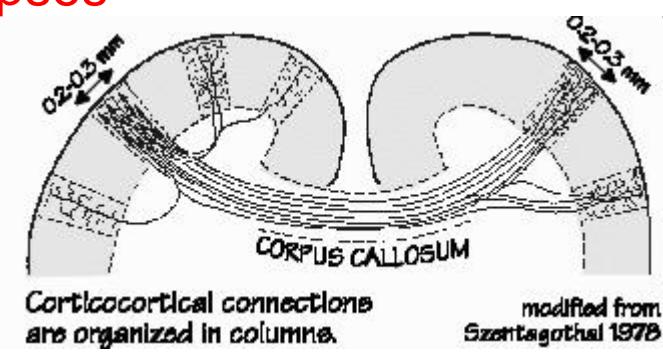
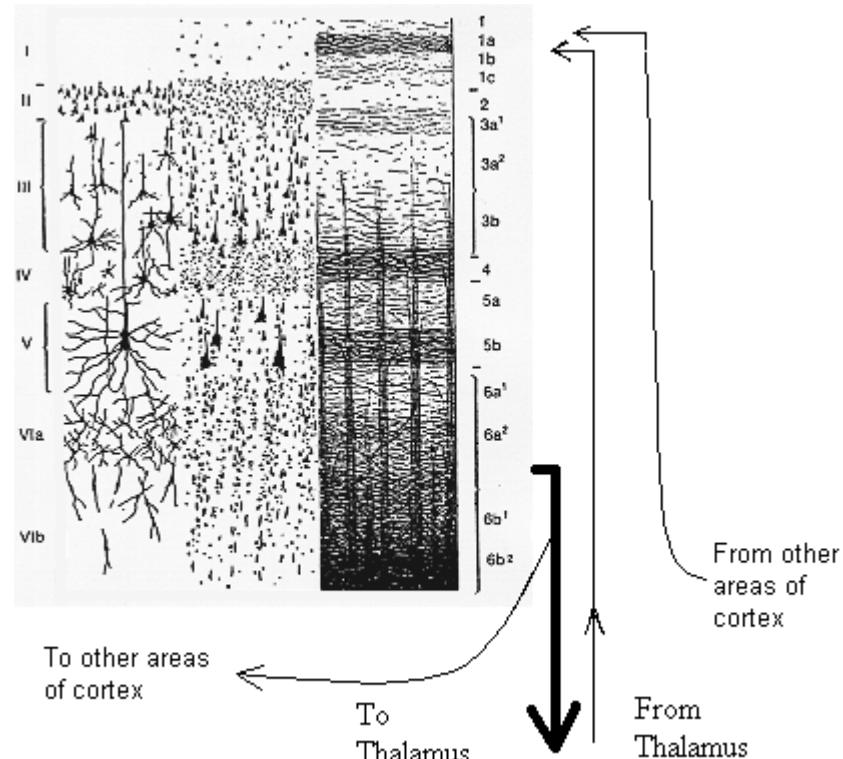
d) Release of action potential

From: Amit

Cortical networks

Neurons form **neural networks**:

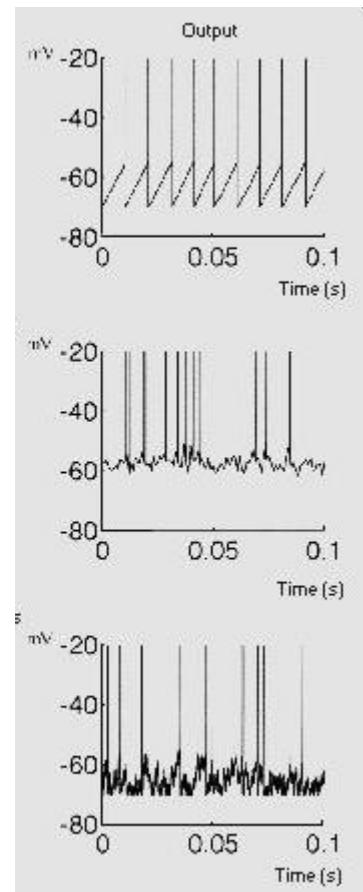
- **Local networks**
 - 1 mm^3 cortical tissue: 10^5 neurons, 10^9 synapses
 - cortex: vertical columns (approxim.)
- **Distant connections**
 - to other cortical areas
 - to subcortical areas (e.g. thalamus)
- **Excitatory / inhibitory receptors at synapses**
 - increase / decrease likelihood for action potential of postsynaptic neuron
- **Anatomical / functional networks**
 - „weak“ synaptic transmission, network dynamics



The neural code

- Basic neuronal „event“: action potential
- But: **How is the information represented and transmitted?**
- **Rate coding:**
 - information in *average* number of spikes per unit time („**firing rate**“); temporal structure of spike train ignored
 - time-dependent firing rate: averaging over short interval ($\approx 10\text{ms}$); analysis: peristimulus time histogram
- **Temporal coding:**
 - precise spike timing; timing fluctuations carry information
 - analysis: higher moments of interspike interval distribution
- **Population coding:**
 - averaging over a number of neurons
 - fast: reflect stimulus changes nearly instantaneously
- or rank code, code combinations...

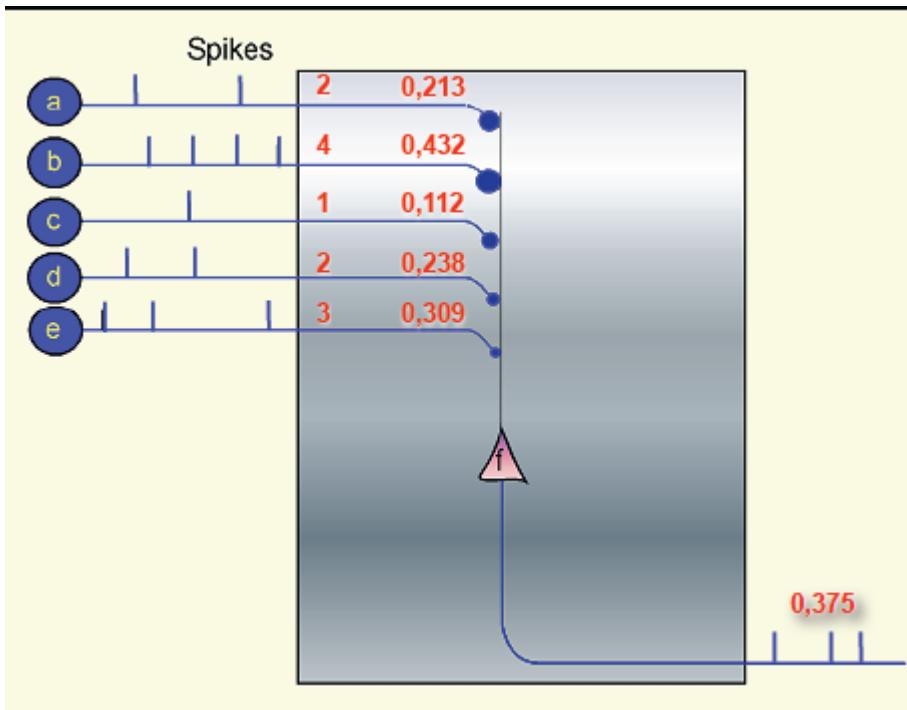
Examples for spike trains



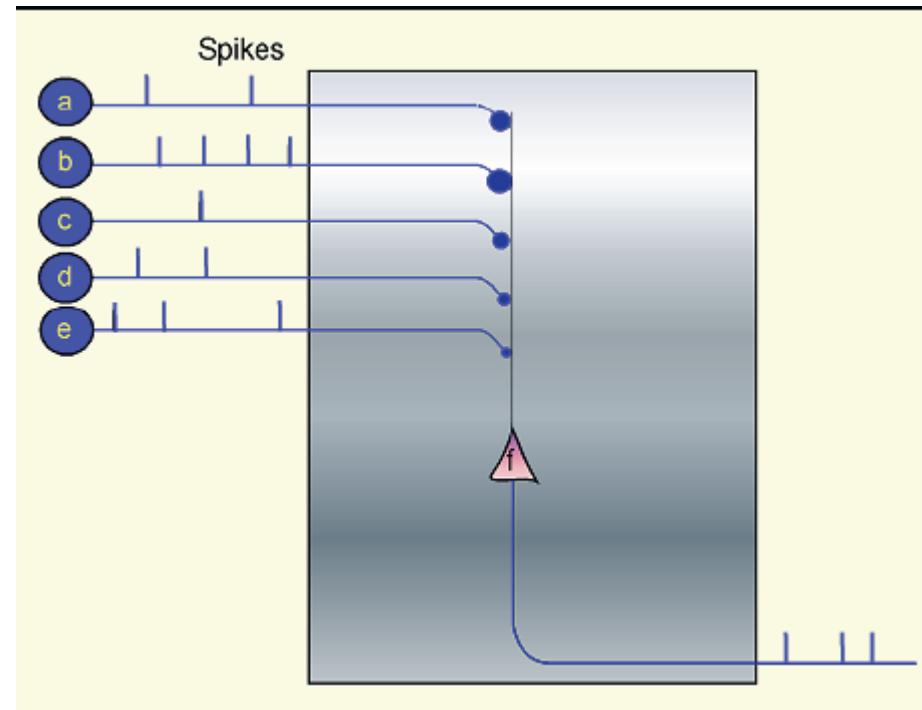
From: http://www.shadlen.org/mike/papers/mine/Current_Opinion1994/nb44031.jpg

The neural code: Illustration

Rate coding



Temporal coding



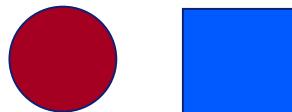
- Neurons send floating point numbers (firing rate)
- Temporal patterning of spikes across neurons critical for computation (synchrony, repeating patterns etc.)
- Apparent noise in spiking is unexplained variation

From: Thorpe, ECCV Tutorial 2008

Neural coding: the binding problem

Representation of *different* features of the same object (e.g. color / shape)?

- *(Spatial) binding problem*: pairing of features belonging to the same object
- Example: red circle versus blue square



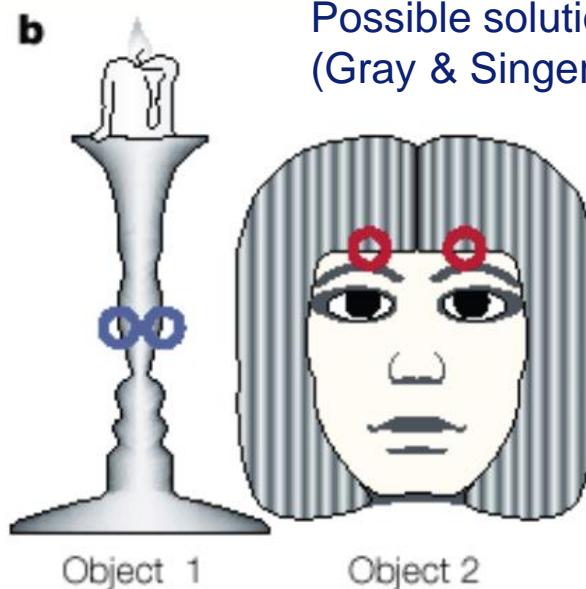
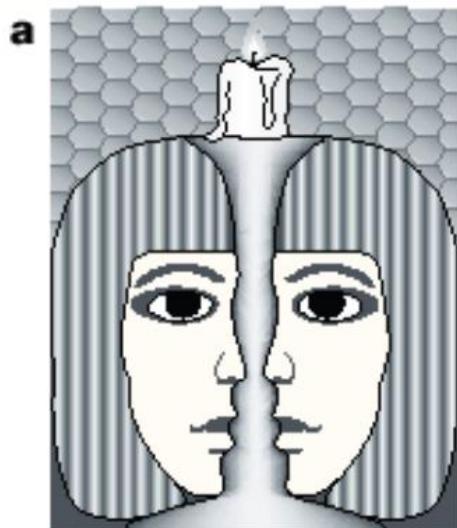
a) single neuron / representational unit for each possible feature combination

- e.g. „grandmother cell“; neuron responds only to red circle (not to blue circle)
- but: enormous number of possible feature combinations!

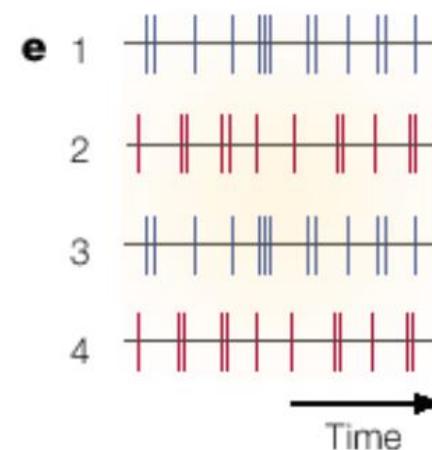
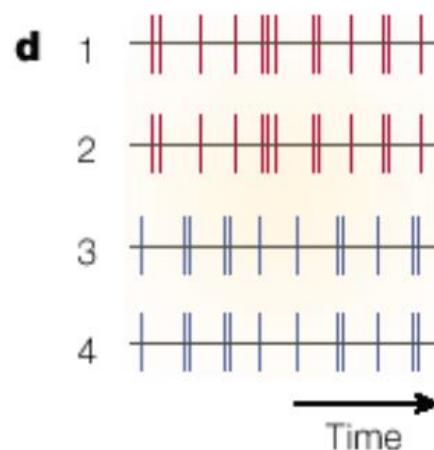
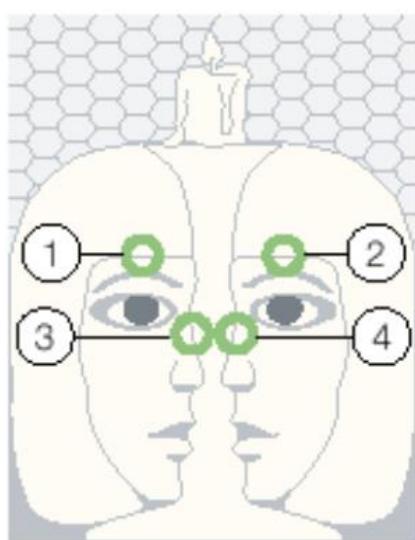
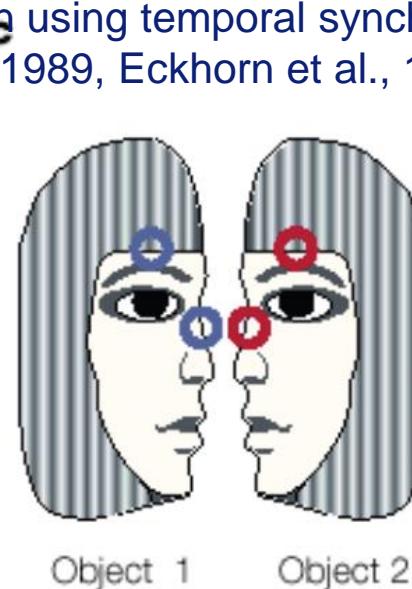
b) distributed representation: each feature coded individually (color vs. shape)

- by single neuron or group of neurons
 - but: how is binding accomplished? Possible explanations:
 - Synchronous activity of „color“ and „shape“ neurons if same object
 - Location tags: location of object in visual field recorded for each feature
- not completely clarified (see e.g. [Holcombe, 2009])

The binding problem: Illustration (temporal synchrony)



Possible solution using temporal synchrony
(Gray & Singer, 1989, Eckhorn et al., 1988)

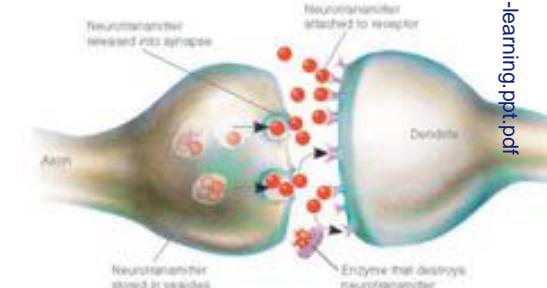


Neurobiology of learning and memory: (Very) basic hypotheses

- Memory: no new neuron production, but **modifying synaptic connections**:
 - Santiago Ramon y Cajal (1894): Memory might be formed by strengthening the connections between existing neurons to improve the effectiveness of their communication
 - Donald O. Hebb (1949): cells may grow connections or undergo metabolic changes that enhance their ability to communicate

„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“
 (D.O. Hebb; „**Hebb's rule**“)

- Synaptic plasticity**
 - Activity-dependent modification* of synaptic efficacy
- Long term potentiation (LTP)**
 - Persistent *increase* in synaptic strength resulting from synchronous stimulation of two neurons



From: Wikipedia

From: <http://sonify.psych.gatech.edu/~walkerb/classes/intro/pdf/05B-learning.ppt.pdf>

Learning: A difficult problem ...

- Minsky („The Society of Mind“, 1985):

„Which agents could be wise enough to guess what changes should then be made? The high level agents can't know such things; they scarcely know which lower-level processes exist. Nor can lower-level agents know which of their actions helped us to reach our high-level goals; they scarcely know that higher-level goals exist. The agencies that move our legs aren't concerned with whether we are walking toward home or toward work – nor do the agents involved with such destinations know anything of controlling individual muscle units.“

- Hebb („Physiological learning theory“, 1976):

„The more we learn about the nature of learning, the farther we seem to get from being able to give firm answers.“

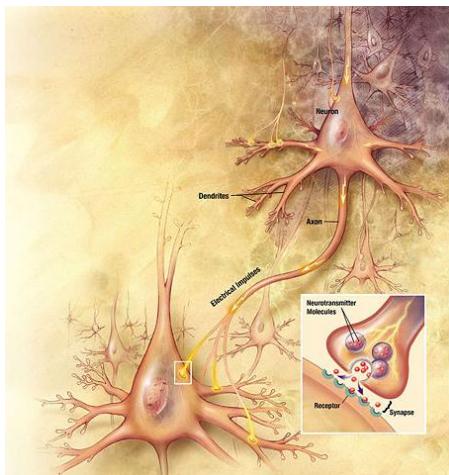
Biological basis: Summary (1)

- Basic anatomy of the human brain
 - cerebral cortex, cortical areas and layers
- Structure and function of a neuron (nerve cell)
 - cell body, axon, membrane, ion channels, dendritic tree, spatiotemporal input integration
 - action potential, electrical signal propagation, refractory period
- Synaptic transmission
 - neurotransmitters, chemical signal propagation, postsynaptic potential
- Cortical networks
 - local / distant, excitatory / inhibitory synapses, anatomical / functional
 - neural code (rate coding, temporal coding, population coding)
 - (spatial) binding problem

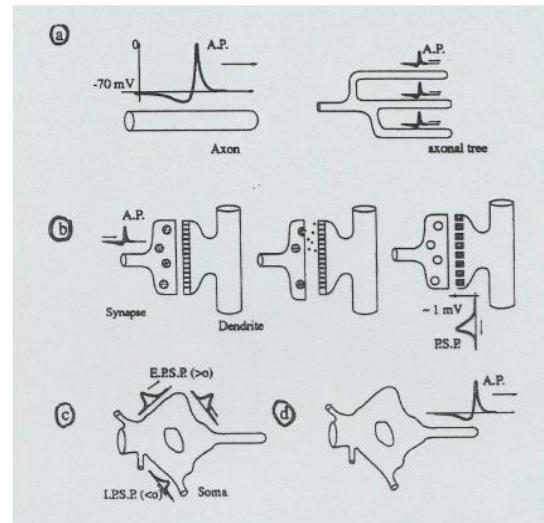
Biological basis: Summary (2)

Information transmission by biological neurons:

- Action potential arrives at **synapse** → release of **neurotransmitter**
- Neurotransmitter modifies ion permeability of postsynaptic membrane → **excitatory / inhibitory postsynaptic potential** at postsynaptic neuron
- If spatio-temporal sum of postsynaptic potentials from all synapses exceeds **threshold**: **release of action potential** at postsynaptic neuron
- **Frequency modulation**: signal strength encoded by frequency of action potentials



<http://svr225.stepx.com:3388/action-potential/file/78242.jpg>



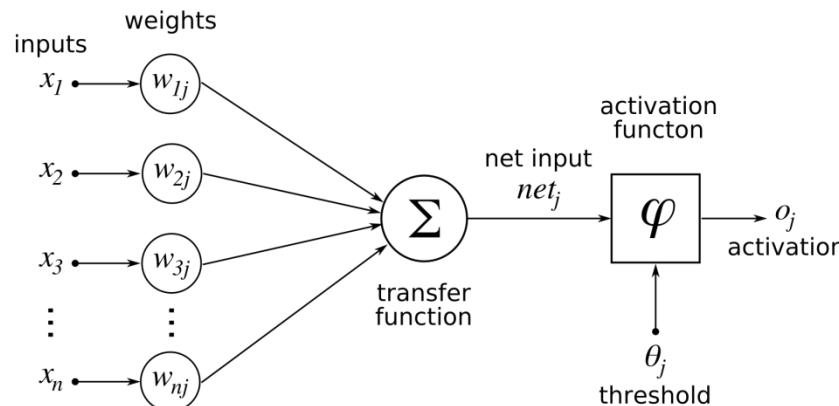
From: Amit

Biological basis: Summary (3)

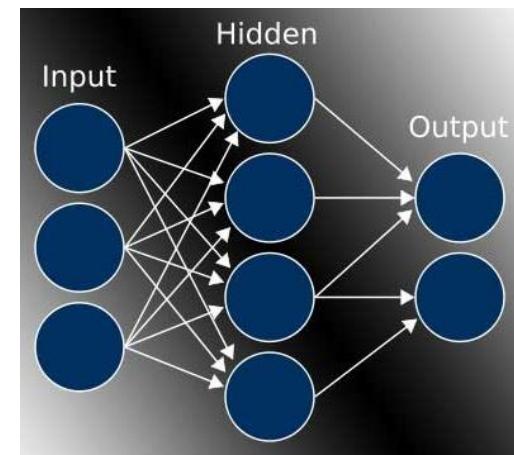
Some basic principles of information processing in the brain:

- capability for learning and self-organization
- generalization capability, flexibility, adaptation to different environments
- ability to handle incomplete, contradictory and noisy data
- fault tolerance
- parallel processing

→ motivates development of artificial neural networks!



http://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png



<http://www.neuralphysics.org/images/categories/neuralnet.jpg>

Further readings

- Müller / Reinhardt, chapter 1
- Amit, chapter 1.2
- Gerstner:
 - chapter 1 (basics, neuronal coding, rate coding, spike coding)
 - chapter 10 (learning, synaptic plasticity, long-term potentiation)
- Rojas, chapter 1.2
- Haykin, chapter 1.2
- Kriesel, chapter 2

NEURON MODELS

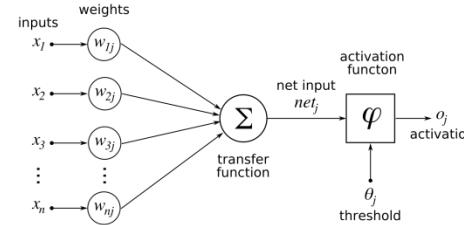
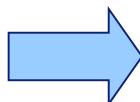
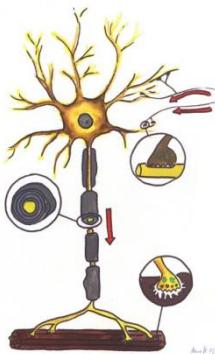
3. From biological to artificial neurons: Neuron models

abstraction / simplicity

- Simulations
- Hodgkin-Huxley model
- Integrate-and-fire model
- Renewal neuron
- Threshold element / perceptron

mathematical tractability

(and many more variants and models)

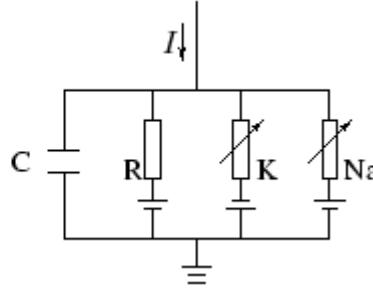
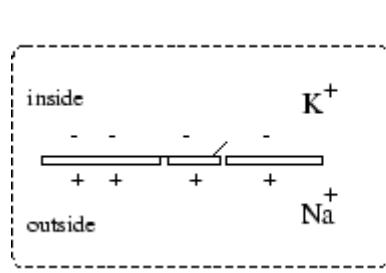


http://upload.wikimedia.org/wikipedia/commons/6/60/ArtificialNeuronModel_english.png

<http://www.webmic.de/images/neuron01.jpg>

a) Hodgkin-Huxley model (1952)

- Kinetic model: describes (squid giant) **axon membrane potential $V(t)$** in detail
- driven by **Na and K ion channels** as well as „leakage“ terms (Cl and K ions)

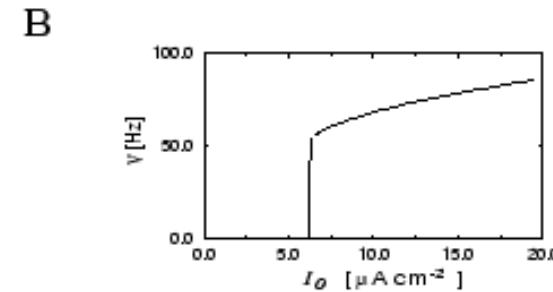
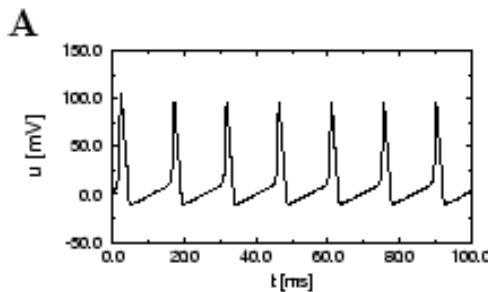


$V(t)$: membrane potential
 E : equilibrium potential (Na, K)
 C_m : membrane capacitance
 $I_{inj}(t)$: applied current
 $G=1/R$: (ion) conductance
 m, n : activation factors (function of V and t)
 h : inactivation factor (function of V and t)

$$C_m \frac{dV}{dt} = \bar{G}_{Na} m^3 h (E_{Na} - V) + \bar{G}_K n^4 (E_K - V) + G_m (V_{rest} - V) + I_{inj}(t)$$

- Example simulation:

(relative) refractory periods implicitly included



A. Spike train for constant input I_0 . B. Mean firing rate as fct. of I_0 .

But: not appropriate for all neuron types; computationally involved

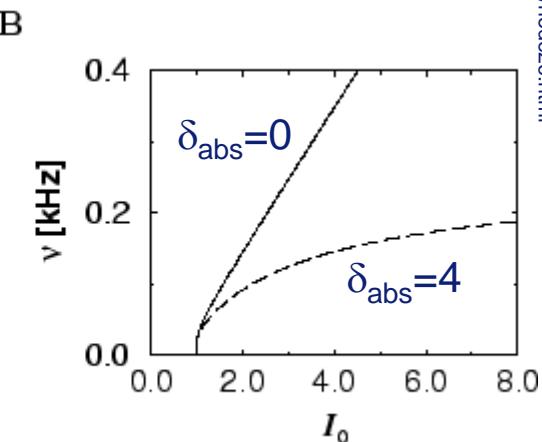
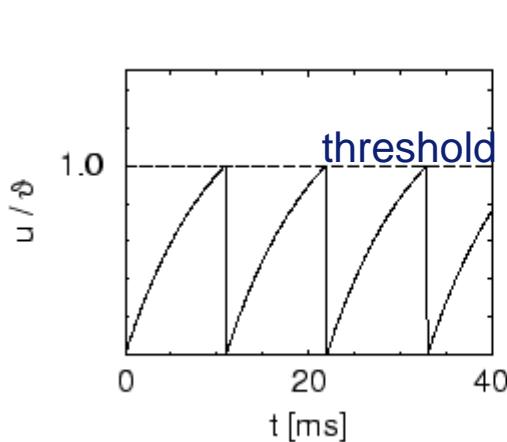
b) Integrate-and-fire model (1907)

- Kinetic model: describes membrane potential $V(t)$ more abstract (no ion channels)

$$I(t) - \frac{V_m(t)}{R_m} = C_m \frac{dV_m(t)}{dt}$$

$V_m(t)$: membrane potential
 C_m : membrane capacitance
 R_m : membrane resistance (leakage term)
 $I(t)$: input current

- At time t_i at which membrane potential reaches threshold ($V_m(t_i) = 9$): generation of a **point-like action potential** at time t_i (**spike shape not modeled**)
- after **absolute refractory period** δ_{abs} : voltage reset to resting potential
- Example simulation:



A. Membrane potential for constant input I_0 . B. Mean firing rate as fct. of I_0 .

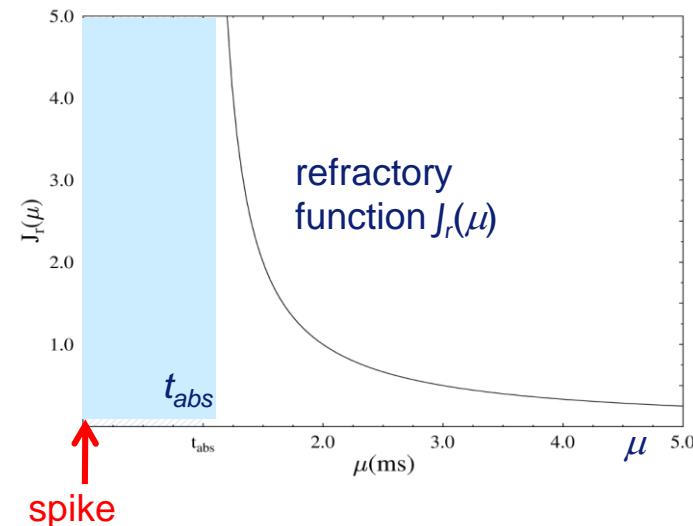
But: still computationally involved (differential equation for each neuron)

c) Renewal neuron (1)

- Differential equation for $V(t) \rightarrow$ simple equation for postsynaptic potential $h(t)$
- Introduce explicit **memory**, i.e. variable μ : time since *last* (output) spike
 - all previous spikes discarded
- Action potential: point-like event $\delta_{\mu,0}$ (as integrate-and-fire; shape not modeled)
- Neuron state: binary variable $x = \delta_{\mu,0} \in \{0,1\}$
 - $x=1$: neuron emits action potential
 - $x=0$: neuron „inactive“ (no action potential)
- Explicit modeling of **refractory function** $J_r(\mu)$ after each spike, e.g.

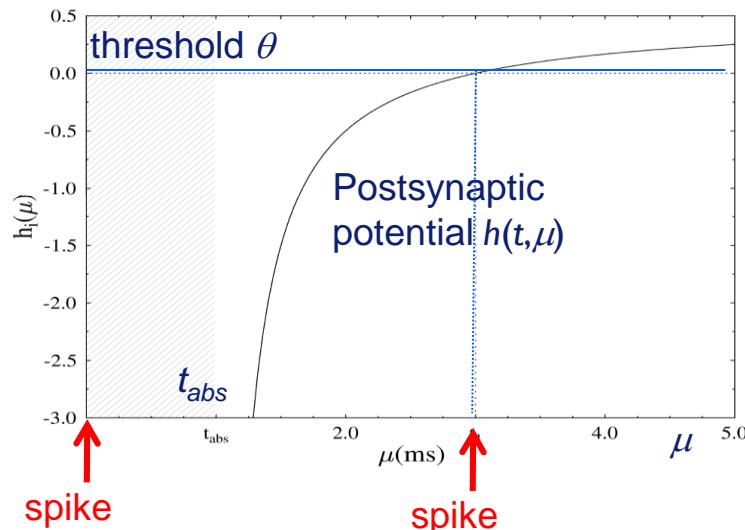
$$J_r(\mu) = \begin{cases} \infty & \text{for } \mu \leq t_{abs} \\ \frac{c}{\mu - t_{abs}} & \text{for } \mu > t_{abs} \end{cases}$$

t_{abs} : **absolute refractory time**
 $c > 0$: **refractory strength**



c) Renewal neuron (2)

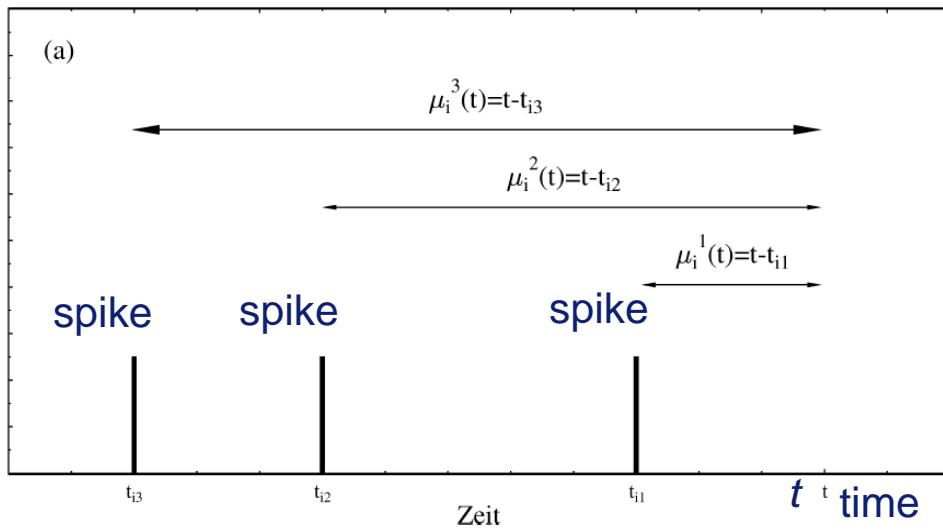
- Postsynaptic potential (threshold: $\theta \in \mathfrak{R}$): $h(t, \mu) = I(t) - J_r(\mu) - \theta$
 - depends on current time t and time μ since *last* (output) spike
- At time t_i at which $h(t_i, \mu)$ reaches threshold θ :
 - 1.) set $\mu = 0$ (i.e. reset refractory properties; same state after each spike: „renewal“)
 - 2.) generation of a point-like action potential $\delta_{\mu,0}$ at time t_i



- + Models exact times of action potentials (without spike shape, as integrate-and-fire)
- + Simple equation for postsynaptic potential (no differential equation)
- Only last spike remembered (\rightarrow no adaptation, no bursts)

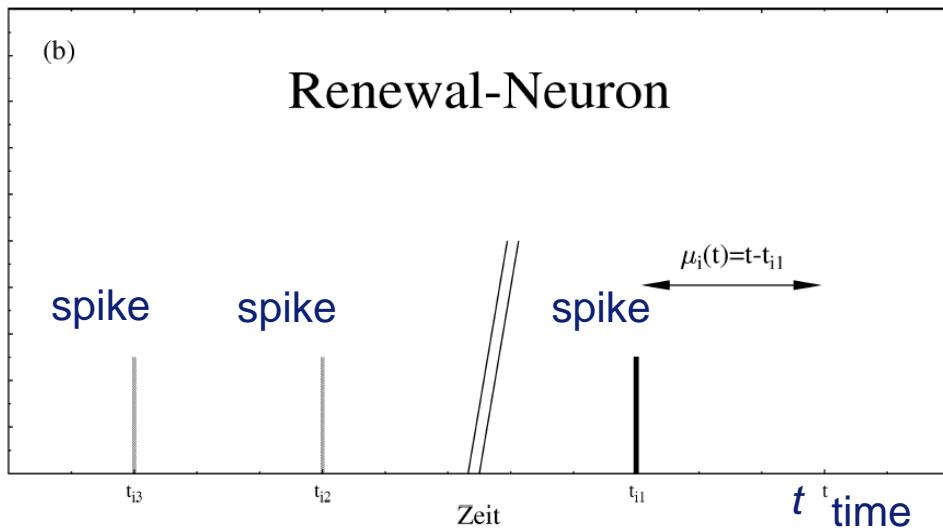
(related model: „spike-response-model“ [Gerstner])

Renewal neuron: Illustration



General case:

all previous action potentials may influence refractory state and thus postsynaptic potential at time t



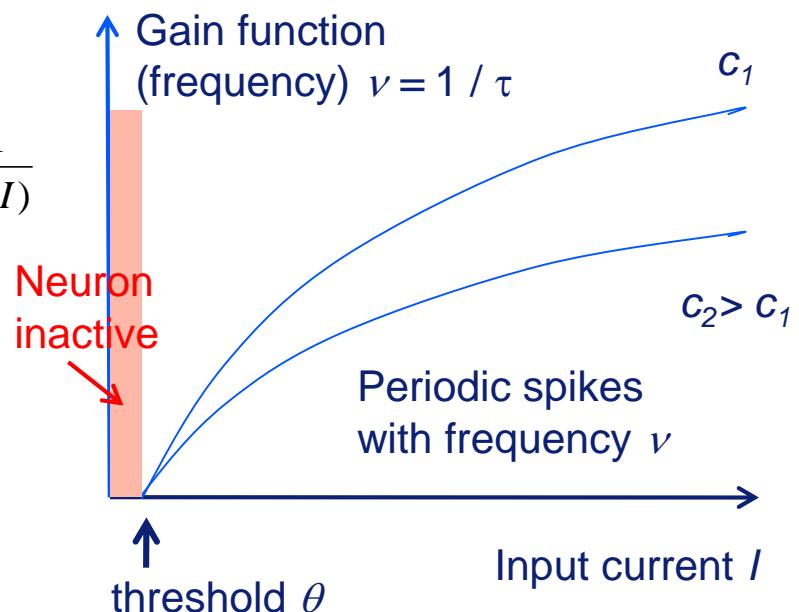
Renewal neuron:

only most recent action potential influences refractory state and thus postsynaptic potential at time t ; previous spikes are discarded

Renewal neuron: Output characteristics

Renewal neuron for *constant* input current $I(t)=I$:

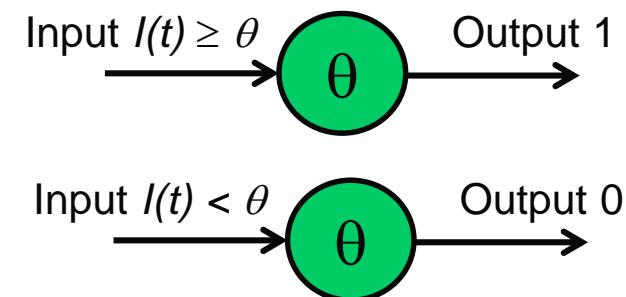
- If $I(t)=I < \theta$: Neuron inactive
- If $I(t)=I > \theta$: Neuron emits strictly **periodic spike train** (renewal property)
- **Interspike interval τ** (time between two consecutive spikes) depends on
 - Input current I : larger $I \rightarrow$ smaller τ
 - Refractory function : larger c \rightarrow larger τ
- **Gain function (spike frequency)**: $\nu(I) = \frac{1}{\tau(I)}$
 - Function of input current I
 - and refractory strength c



d) Threshold element (with single input channel)

- Further simplification: no refractory properties, no memory of last spike
- Neuron state: binary variable $x \in \{0,1\}$
 - $x=1$: neuron „active“ (one or more action potentials)
 - $x=0$: neuron „inactive“ (no action potential)
- i.e. no explicit modeling of individual action potentials
- Postsynaptic potential (threshold: $\theta \in \mathfrak{R}$): $h(t) = I(t) - \theta$
 - depends on current time t , no refractory properties
- If $h(t) \geq 0$: output $x = 1$ $x(t) = \Theta[h(t)]$
- If $h(t) < 0$: output $x = 0$ Heaviside - Fct.

→ simple threshold element (perceptron)



- + Computationally simple
- No interpretation of active state as single action potential
- No refractory properties

Modeling noise

- So far: neuron output *deterministic*
 - Postsynaptic potential above threshold \rightarrow neuron becomes immediately active
- **Physiological processes may be complex and uncertain**
 - E.g. additional effects which are not contained in the model
 - \rightarrow Potential fluctuations of spike times

\rightarrow Include **noise** in the model

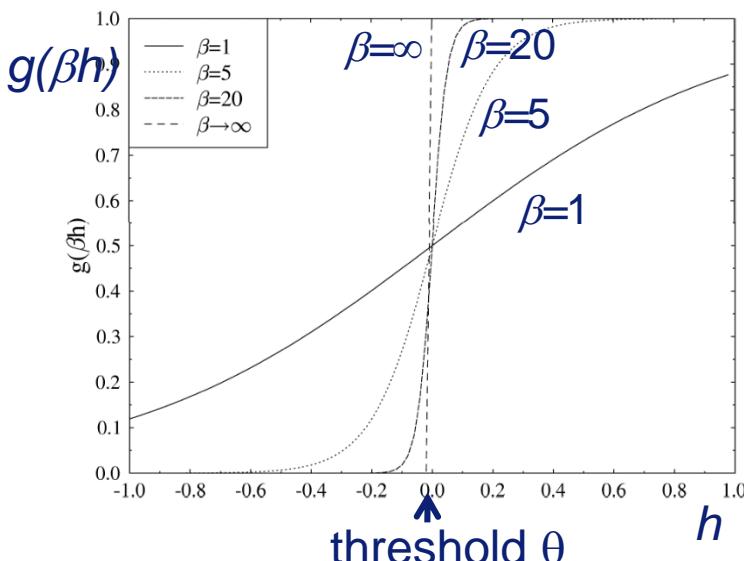
- **Probability $g(\beta h)$ to emit action potential,**
e.g.

$$g(\beta h) = \frac{1}{2} (1 + \tanh(\beta h)) \in [0;1]$$

- **Neuron output: random variable $x_{out} \in \{0,1\}$**
with $P(x_{out}=1) = g(\beta h)$, $P(x_{out}=0) = 1 - g(\beta h)$

• **Consequences:**

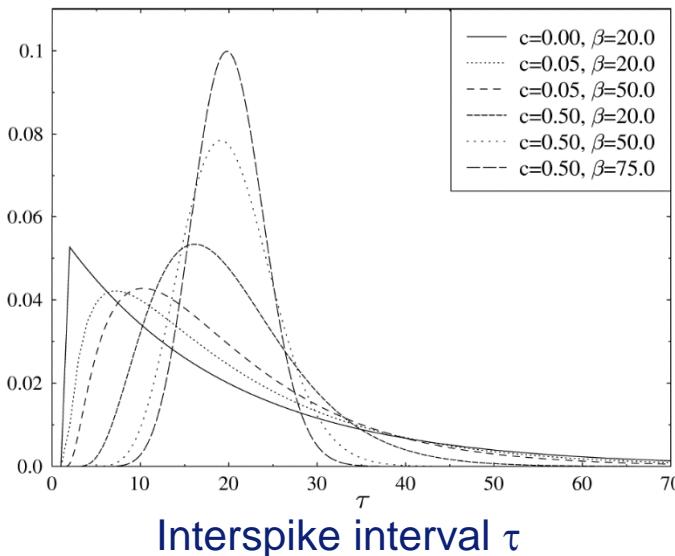
- Neuron can be active even if input below threshold
- Neuron might be inactive even if input above threshold



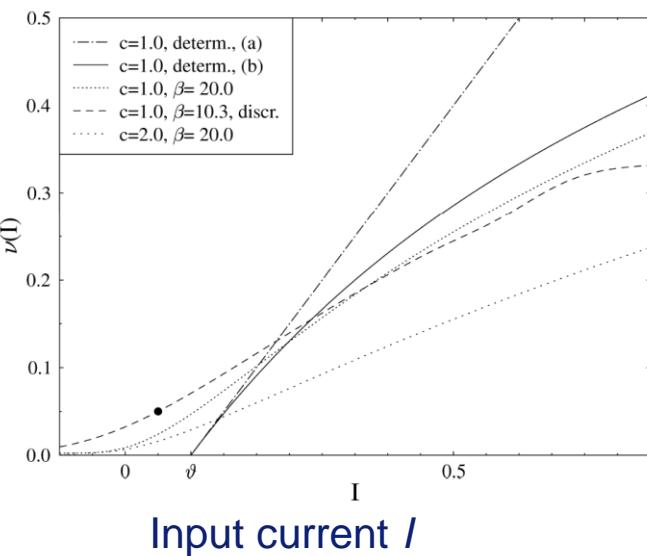
Modeling noise: Consequences

- Transitions into active state **stochastic** (instead of deterministic)
 - Smaller $\beta \rightarrow$ more randomness
- Threshold unit:
 - Mean output $\langle x_{out} \rangle \in [0;1]$ depends on input current I
- **Renewal neuron:** random fluctuation of spike time around mean
 - **Interspike interval distribution:** broader (around mean $\langle \tau \rangle$) for smaller β
 - **Gain function** $v(I) = \frac{1}{\langle \tau(I) \rangle}$: positive even if $I < \theta$ and less step for smaller β

Interspike
interval
distribution
 $D(\tau)$

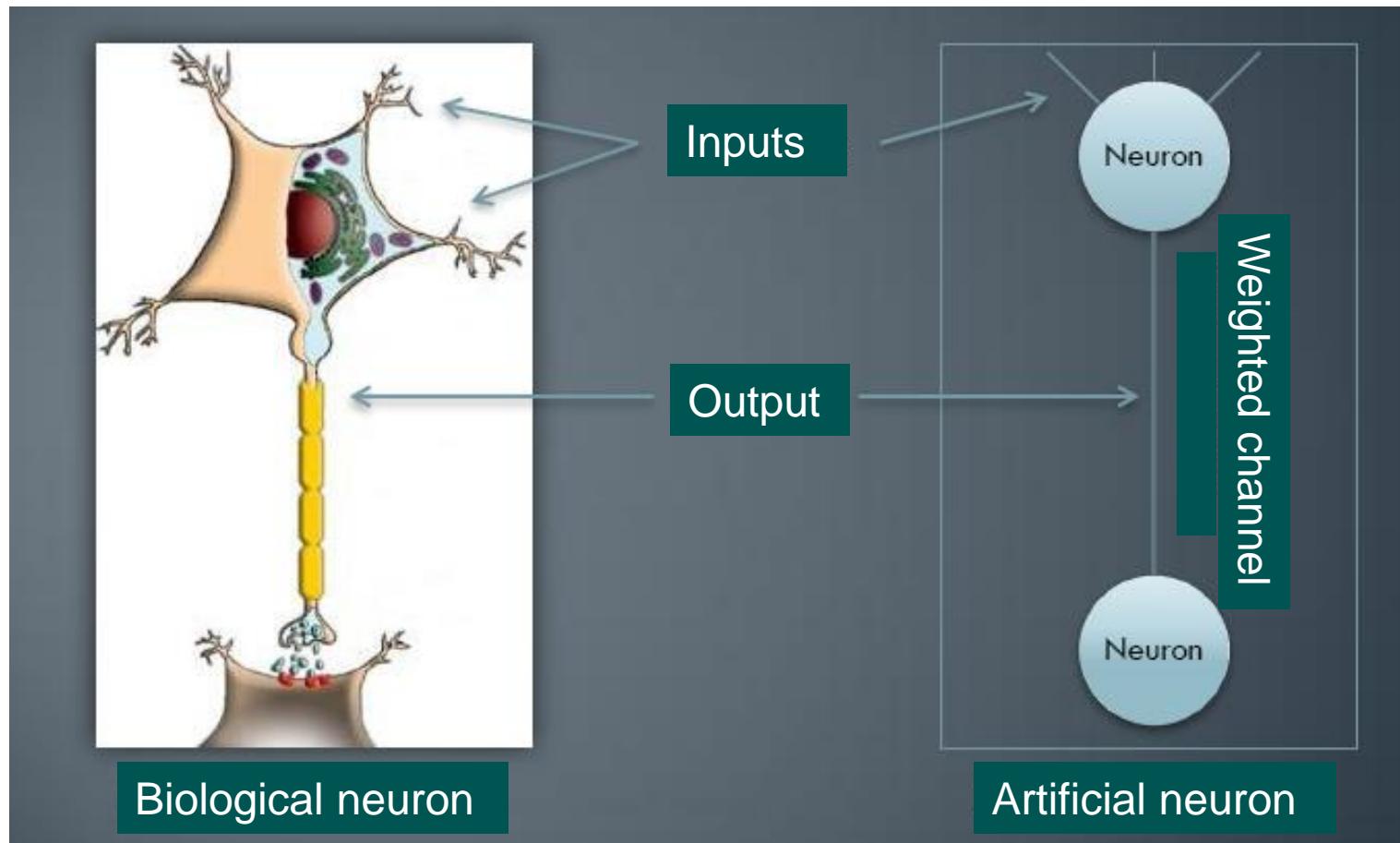


Gain
function
 $v(I)$



Modeling synaptic connections

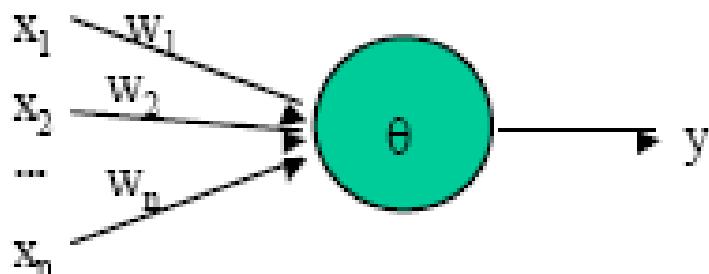
- A **synaptic connection** between two neurons is modeled as a **logical channel** between the neurons, associated with a **weight** that characterizes the synaptic strength („**synaptic efficacy**“)



From: Ruttlöff

Modeling synaptic connections

- Synaptic connection between two neurons → (directed) logical channel
 - Input channel: combination of synapse and dendrite
 - As many logical channels as synapses connecting to the neurons' dendrites
- Input channel i associated with weight parameter w_i : synaptic efficacy
 - Determines amount of post-synaptic potential added to the soma if channel i is activated (amount of post-synaptic penetrating current per pre-synaptic spike)
 - „Strength“ of the synaptic transmission characterized as single number (neglecting the influence of different neurotransmitters on synaptic strength)
- Graphical representation: Line with arrow, associated with weight w_i



$w_i > 0$: excitatory connection
 $w_i < 0$: inhibitory connection
 $w_i = 0$: no connection

From: Bittel

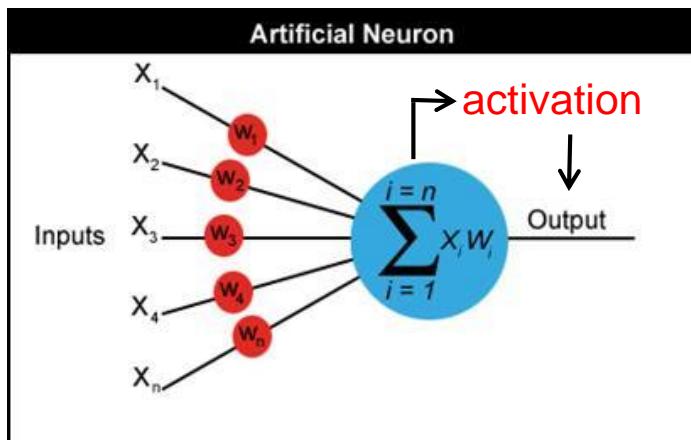
Artificial neuron: Definition

An **artificial neuron** is a tuple consisting of

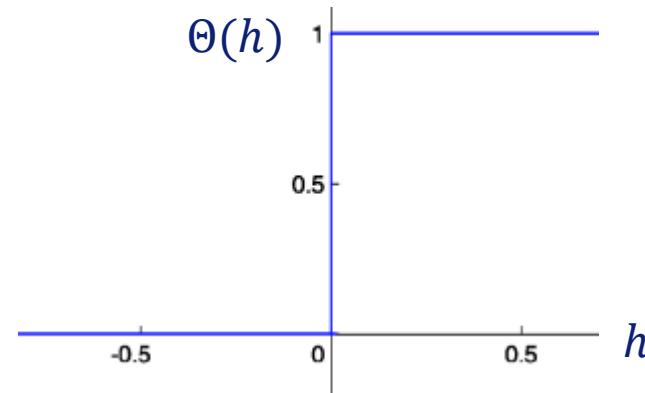
- an **input vector** $\mathbf{x} = (x_1, \dots, x_m)$ (binary or real)
- a **weight vector** $\mathbf{w} = (w_1, \dots, w_m)$ (real)
- an **activation function** $f(h)$ (binary or real)

The **neuron output** is computed as $x_{out} = f(h) = f(\mathbf{x} \cdot \mathbf{w} - \theta) = f(\sum_{i=1}^n w_i \cdot x_i - \theta)$

NOTE: several different definitions exist for activation / output / transfer function!



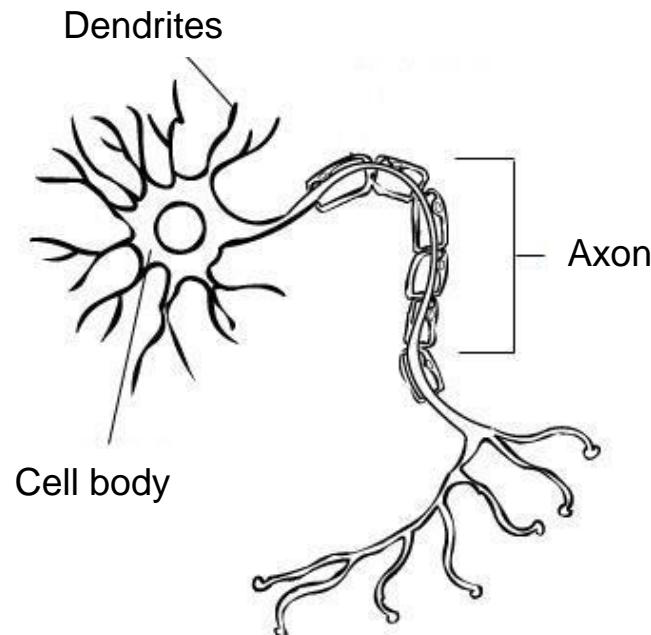
Activation function:
e.g. step function (binary) $f(h) = \Theta(h)$



http://www.ai-junkie.com/ann/evolved/nnt3_files/image002.jpg

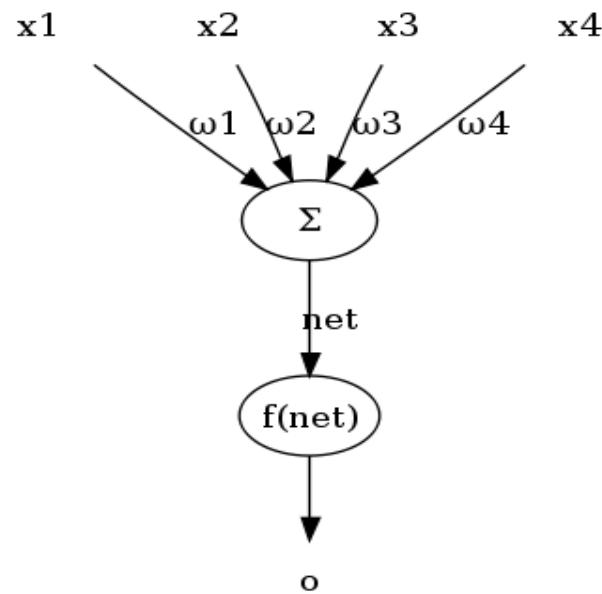
Biological and artificial neuron

Biological neuron



- Pre-synaptic action potentials lead to **post-synaptic potentials (PSPs)** at the dendrites
- **Spatio-temporal integral** of all PSPs at soma
- If the **total PSP exceeds a threshold potential**, the neuron **emits an action potential** which **propagates** along the axon to synapses connecting to other neurons

Artificial neuron

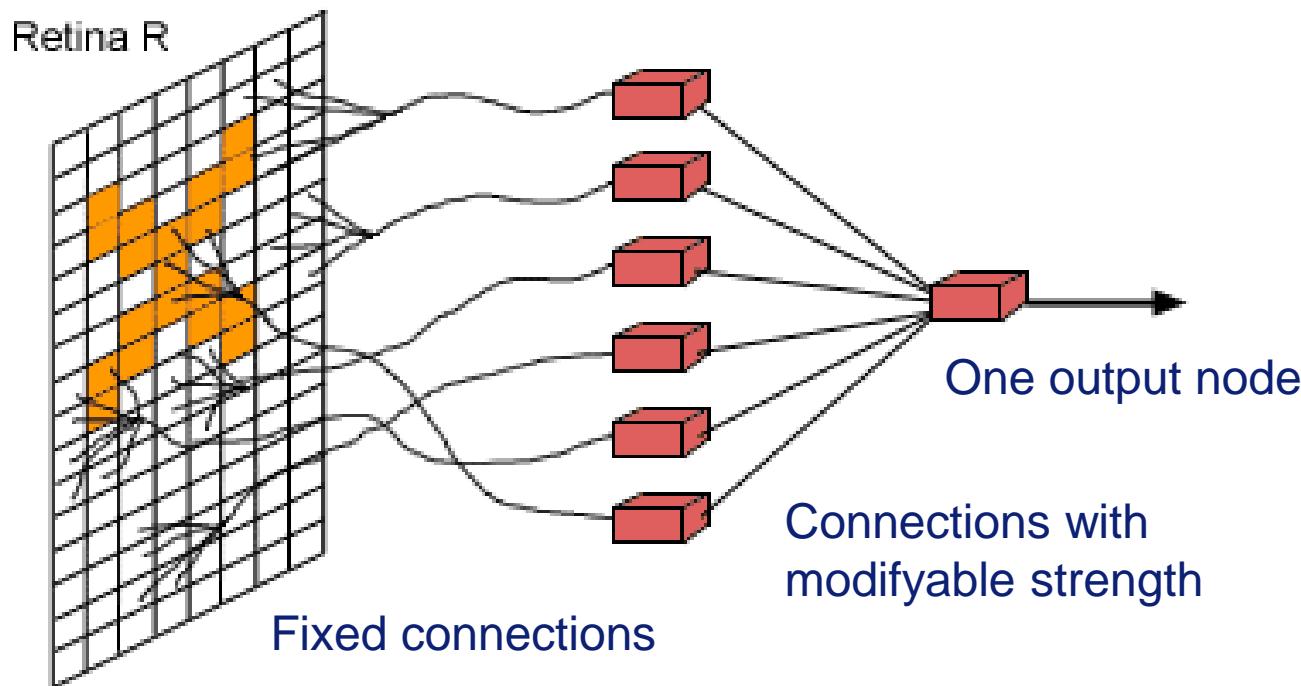


- Active pre-synaptic neurons lead via the **synaptic weights** to **post-synaptic potentials (PSPs)**
- The (total) **PSP** of the neuron is a **weighted sum** of all neuron **inputs**
- If the **sum exceeds a threshold**, the neuron **becomes active** and the activation is **propagated** to other connected neurons

The (single-layer) perceptron

(Single-layer) Perceptron

- Artificial neuron with $f(h) = \Theta[h]$ Θ : Heaviside (step) function
- Output: $x_{out} = \Theta \left[\sum_{i=1}^m x_i \cdot w_i - \theta \right]$ **Linear threshold element**
- Motivation for the (single-layer) perceptron (Rosenblatt, 1962)
 - information processing in the visual field of the retina (Minsky & Papert, 1969)



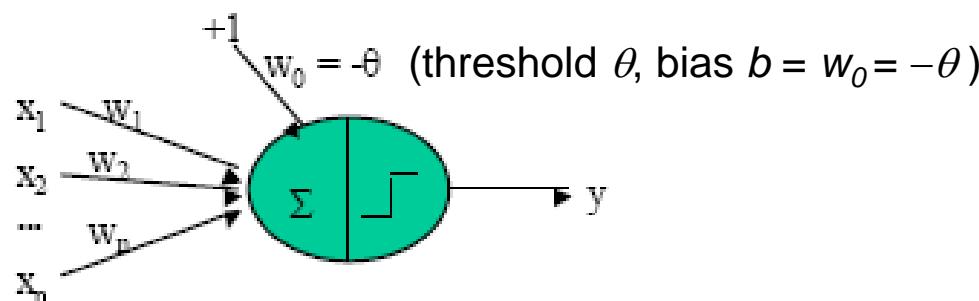
From http://www.chemgapedia.de/vsengine/media/vsc/de/ch/13/anc/daten/neuronalenetze/images/snn_107.gif 91

Simplification of notation: Threshold as „bias“

A simpler notation (but equivalent formulation) is obtained by regarding the threshold as „bias“ (i.e. similar to a synaptic weight with constant input 1):

- Introduce additional weight $w_0 = -\theta$ with fixed input $x_0 = 1$:

$$\sum_{i=1}^m x_i \cdot w_i - \theta = \sum_{i=0}^m x_i \cdot w_i \quad \text{with} \quad w_0 = -\theta \quad \text{and} \quad x_0 = 1$$

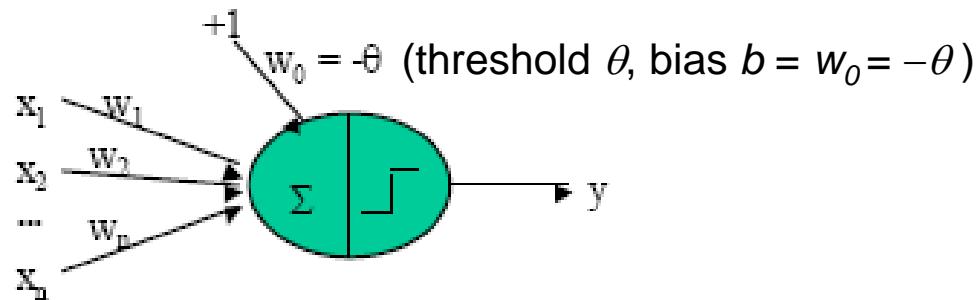


- Terminology: θ : „threshold“
 $b := w_0 = -\theta$ „bias“

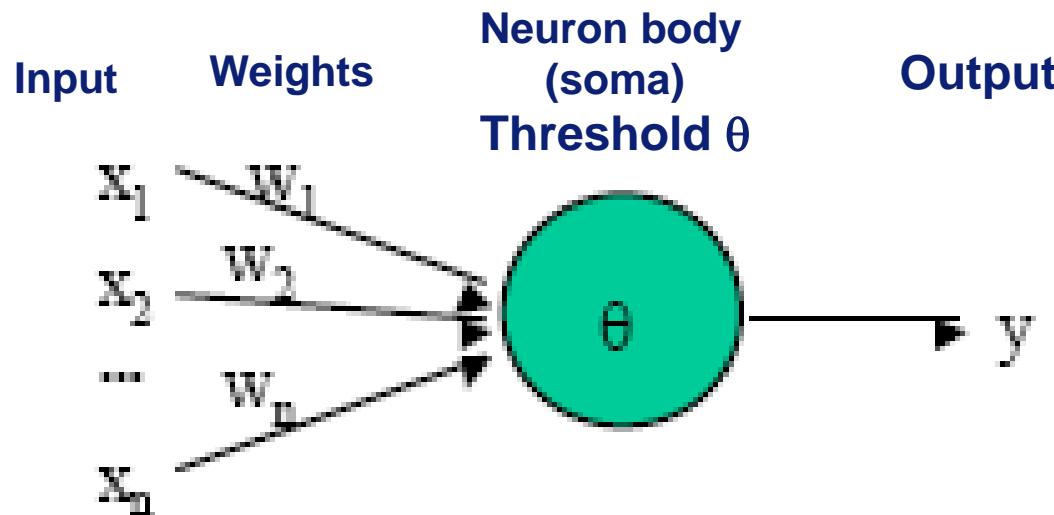
From: Bittel

Simple graphical representation of a perceptron

- Inputs are summed up, output determined as threshold operation



- Even simpler representation:



From: Bittel

Perceptron: Interpretations

1. „Neurophysiological“ interpretation:

- Perceptron as simple artificial neuron, potentially as element of large network
- collecting synaptic input and becoming „active“ (i.e. emitting action potentials) if postsynaptic potential exceeds threshold
 - Binary state 1: neuron active (emits sequence of action potentials)
 - Binary state 0: neuron inactive (no action potentials)
- **Binary** input patterns (x_1, \dots, x_n)
 - Indicates whether pre-synaptic neuron $i \in \{1, \dots, n\}$ is active or not

2. More abstract: „Logical“ interpretation:

- Perceptron as (binary) **classifier**:
- „Arbitrary“ (e.g. real) input pattern $(x_1, \dots, x_n) \rightarrow$
$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_{i=1}^m w_i \cdot x_i \geq \theta \\ 0 & \text{else} \end{cases}$$
- Perceptron classifies each (binary) input pattern as either „1“ or „0“
 - Binary states 1 and 0: abstract variables; no interpretation w.r.t. action potentials
- Example: Perceptron realising **Boolean functions** (i.e. binary inputs)

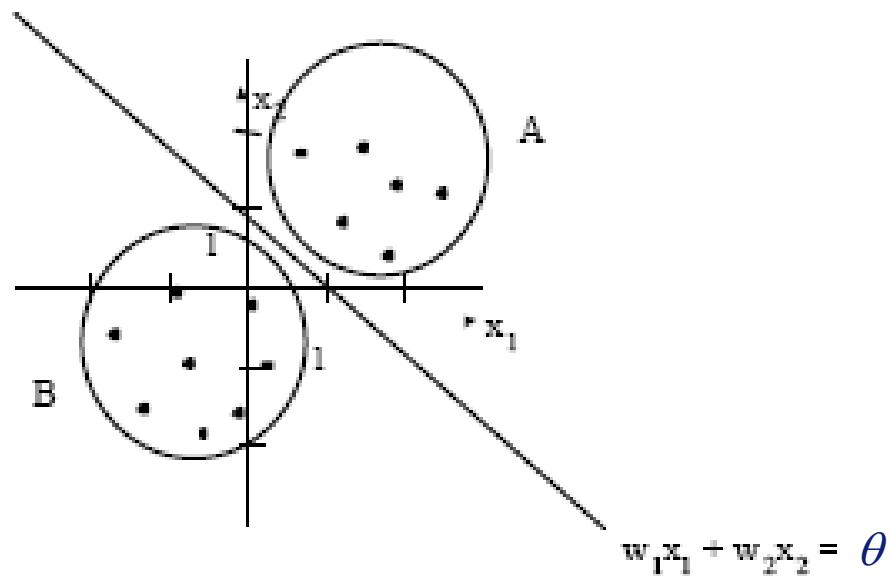
Mathematical definition: Linear separability (point sets)

Two sets of points $A, B \subseteq \mathbb{R}^n$ are **linearly separable** if w_1, \dots, w_n, θ exist so that

$$\sum_{i=1}^m w_i \cdot x_i \geq \theta \quad \text{for all points } (x_1, \dots, x_n) \in A$$

$$\sum_{i=1}^m w_i \cdot x_i < \theta \quad \text{for all points } (x_1, \dots, x_n) \in B$$

- Example:



A and B are linearly separable

From: Bittel

Mathematical definition: Linear separability (binary function)

A binary function $f: \mathbb{R}^n \rightarrow \{0,1\}$ is **linearly separable** if the two sets

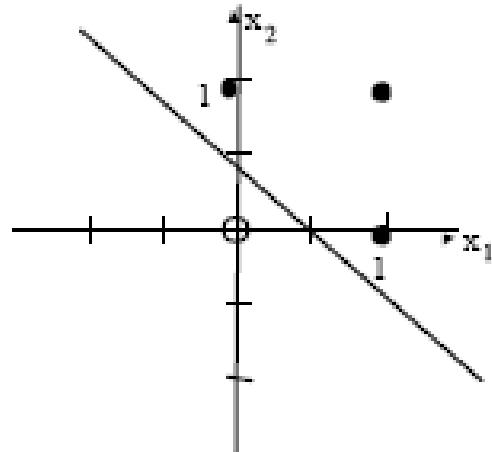
$$\{ (x_1, \dots, x_n) \mid f(x_1, \dots, x_n) = 1 \} \text{ and}$$

$$\{ (x_1, \dots, x_n) \mid f(x_1, \dots, x_n) = 0 \}$$

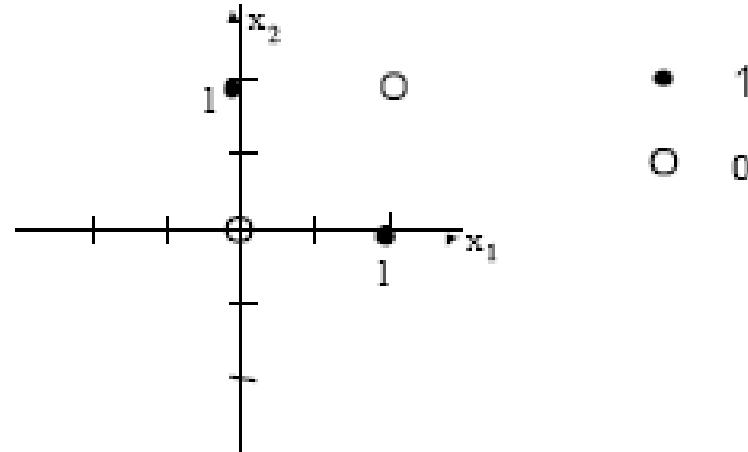
are linearly separable.

- Examples:

The OR-function is linearly separable



The XOR-function is NOT linearly separable



From: Bittel

Mathematical definition: Boolean function

- Boolean function with n inputs: function of the form $f: \{0,1\}^n \rightarrow \{0,1\}$
- Number of Boolean functions with n inputs: 2^{2^n}
- Number of Boolean functions and number / percentage of them which are **linearly separable**:

n	# Boolean functions with n inputs	# of them which are linearly separable	% of them which are linearly separable
1	4	4	100 %
2	16	14	87.5 %
3	256	104	40.6 %
4	65.536	1.772	2.7 %
5	$4.3 \cdot 10^9$	94.572	$2.2 \cdot 10^{-3} \%$
6	$1.8 \cdot 10^{19}$	5.028.134	$2.7 \cdot 10^{-11} \%$

From: Lippe

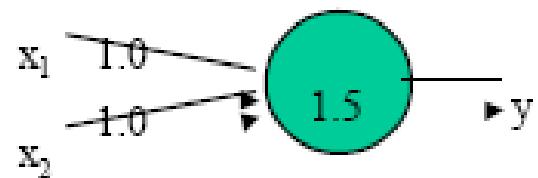
Example: Perceptron as binary classifier

- Goal: Realising Boolean functions with (noise-free single-layer) perceptron
- Example: 2 inputs
- Desired: *Weight factors, threshold* so that given Boolean function realized

1. Boolean „AND“

x_1	x_2	AND
0	0	0
0	1	0
1	0	0
1	1	1

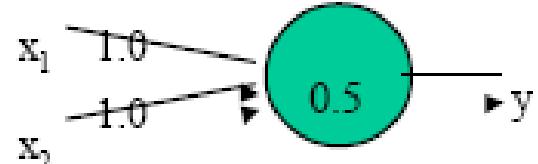
• AND-Funktion:



2. Boolean „OR“

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

• OR-Funktion:



From: Bittel

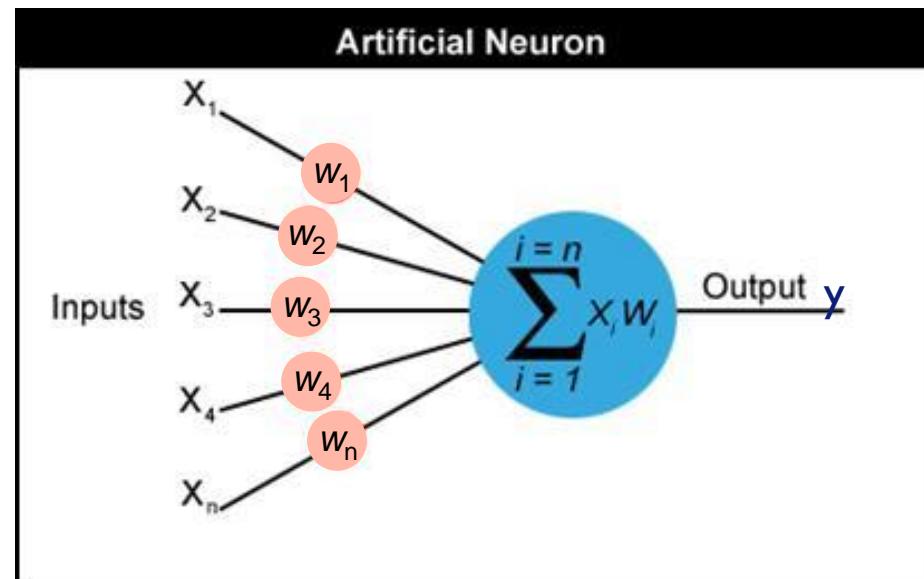
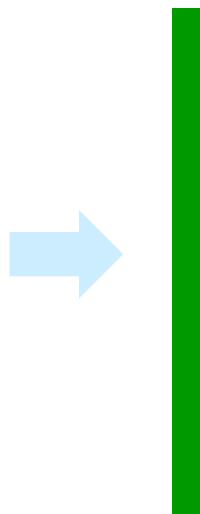
Example: Perceptron as binary classifier

- Real inputs represented as **vector**, generally with some pre-processing
 - E.g. subtract mean, normalize to $[0,1]$ or $[-1, 1]$ (i.e. real-valued inputs)

Input example



vector \mathbf{x}



A blue curved arrow pointing from the text below to the neuron diagram.

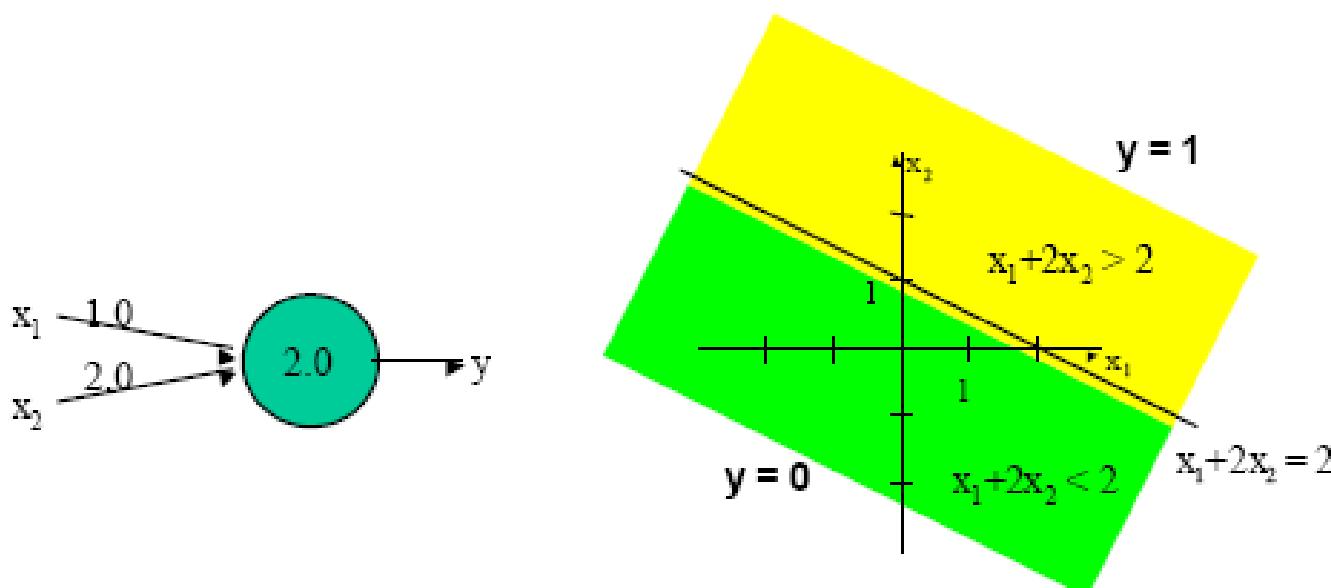
$$\text{output } y = f(h) = \Theta(\sum_{i=1}^n w_i \cdot x_i - \theta) \in \{0,1\}$$

Examples for output:

- Does the image contain a face ($y = 1$) or not ($y = 0$)?
- Is the person female ($y = 1$) or male ($y = 0$)?
- ...

Perceptron: Geometrical interpretation

- (Single-layer) perceptron linearly separates the input space into 2 regions:
 - Region 1 consists of input points yielding perceptron output 0
 - Region 2 consists of input points yielding perceptron output 1
 - Separation boundary: $(m-1)$ dimensional hyperplane satisfying $\sum_{i=1}^m w_i \cdot x_i = \theta$
- Example (2 inputs):
 - $w_1 = 1.0, w_2 = 2.0, \theta = 2.0$



From: Bittel

(Single-layer) perceptron: Strengths and weaknesses

Strengths:

- Simple (i.e. mathematically tractable) neuron model
- Realisation of basic (i.e. linear separable) classification tasks
- Very basic neurophysiological interpretation (e.g. visual field, retina)

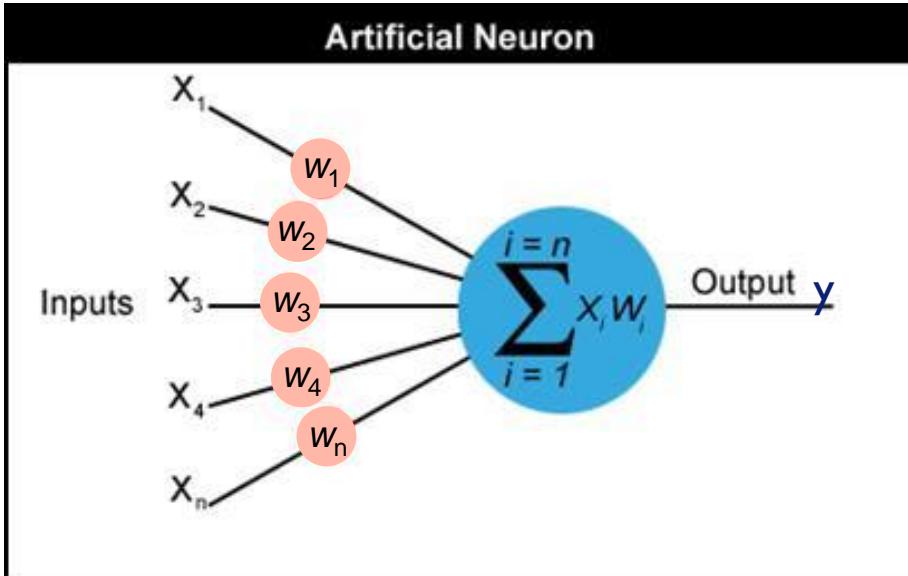
Weaknesses:

- Oversimplified neuron model
 - no refractoriness → no interpretation of active state in terms of single spikes
- Can only represent linearly separable Boolean functions
 - Linear inseparable Boolean functions (e.g. XOR): needs extension
→ Multi-layer perceptrons

Extensions:

- Spiking neurons (e.g. renewal neuron, spike-response-model [Gerstner], ...)
- Analog activation function (output)
- Networks and multi-layer perceptrons (later)

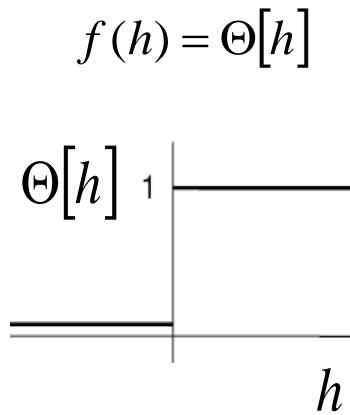
From binary to analog outputs



- So far: **Binary** activation function / output $y = f(h) = \Theta(\sum_{i=1}^n w_i \cdot x_i - \theta) \in \{0,1\}$
 - Activation function: **Heaviside** function Θ
- Now: Use more **general activation functions** f such that e.g. $y \in [0,1]$, $y \in \mathbb{R}_0^+$ or $y \in \mathbb{R}$ → allows for more general applications e.g. **regression** problems

Examples for commonly used activation functions (1)

Heaviside function

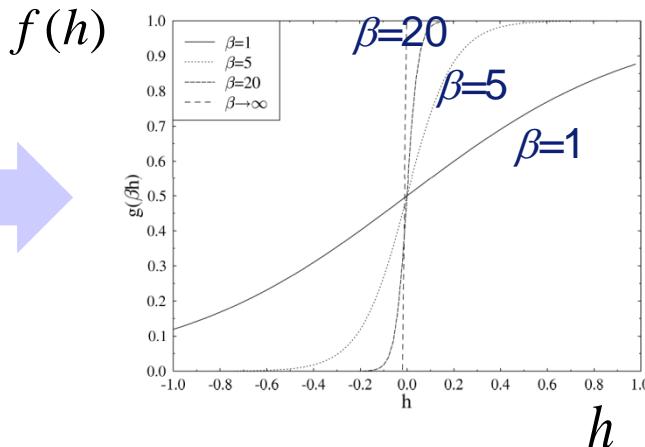


binary;
output $y \in \{0,1\}$
not differentiable at $h = 0$

binary classification

Tangens hyperbolicus

$$f(h) = \frac{1}{2}(1 + \tanh(\beta h))$$

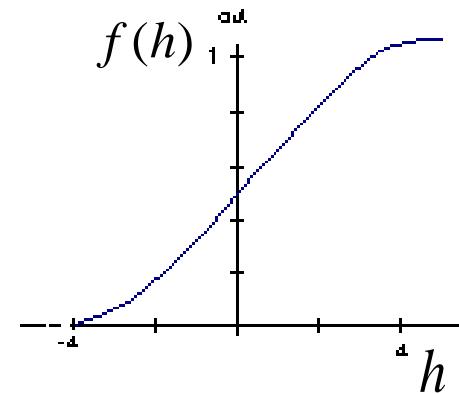


sigmoid;
output $y \in [0,1]$

binary classification
(probability for $y = 1$)

Logistic function

$$f(h) = \frac{1}{1 + e^{-kh}}$$



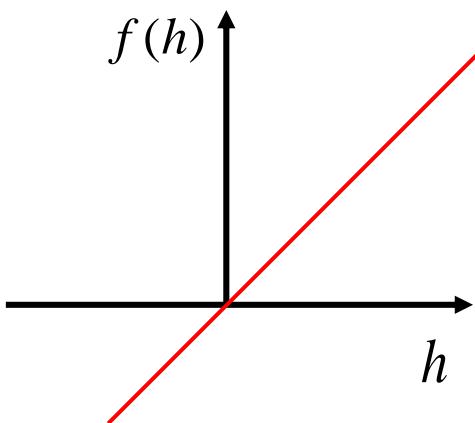
sigmoid;
output $y \in [0,1]$

binary classification
(probability for $y = 1$)
logistic regression on h

Examples for commonly used activation functions (2)

Linear function

$$f(h) = h$$

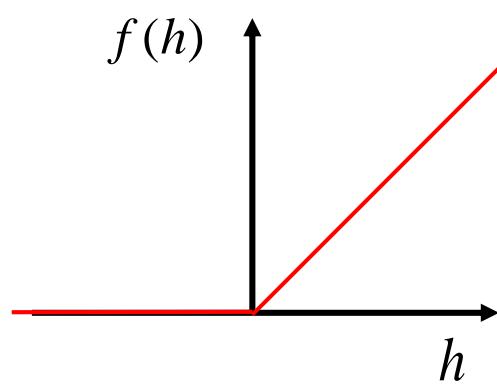


linear;
output $y \in \mathbb{R}$

(linear, multi-dim.) regression
(no nonlinearity)

Rectified linear unit
„ReLU“

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

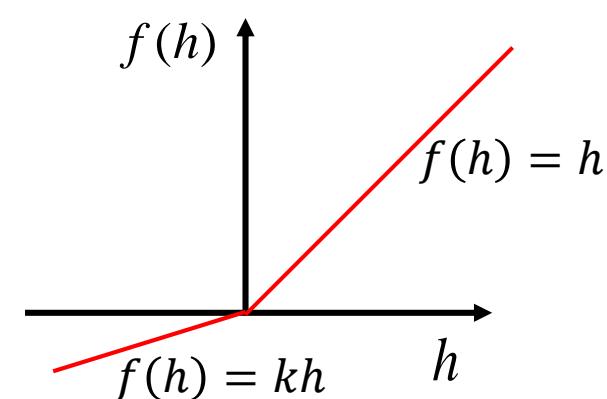


piecewise linear;
output $y \in \mathbb{R}_0^+$

non-neg. linear regression

Parametric ReLU
„PReLU“

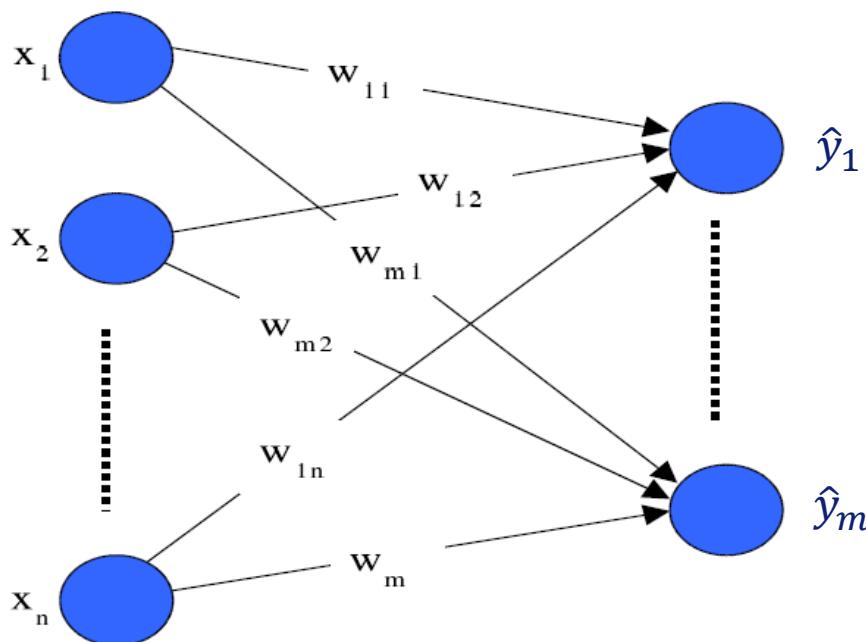
$$f(h) = \max(h, 0) + k \cdot \min(h, 0)$$



piecewise linear;
output $y \in \mathbb{R}$;
parameter k

More than one output unit

- w_{jk} : Weight from the k^{th} input neuron to the j^{th} output neuron (matrix)
- PSP: $h_j = \sum_k w_{jk} x_k - \theta_j$ or $\mathbf{h} = \mathbf{Wx} - \boldsymbol{\theta}$ (vector, $j = 1, \dots, m$)
- Network output \mathbf{y} : $y_j = f(h_j)$ or $\mathbf{y} = f(\mathbf{h})$ (vector, $j = 1, \dots, m$)

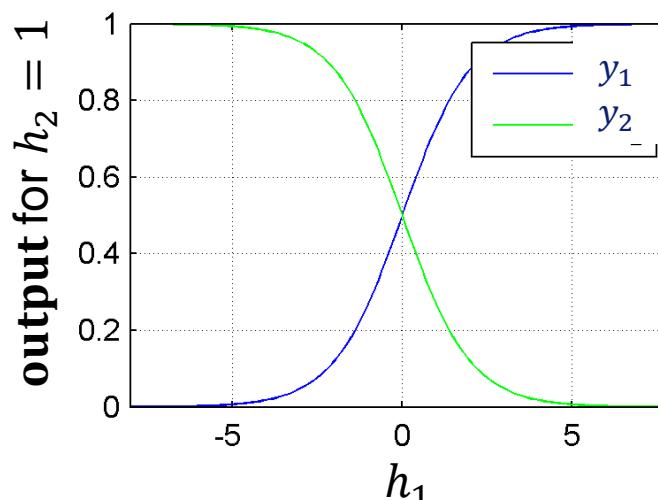


Softmax activation function

Softmax activation function $y_j = f(h_j) = s(h_j) := \frac{e^{h_j}}{\sum_{k=1}^m e^{h_k}} \quad (j = 1, \dots, m)$

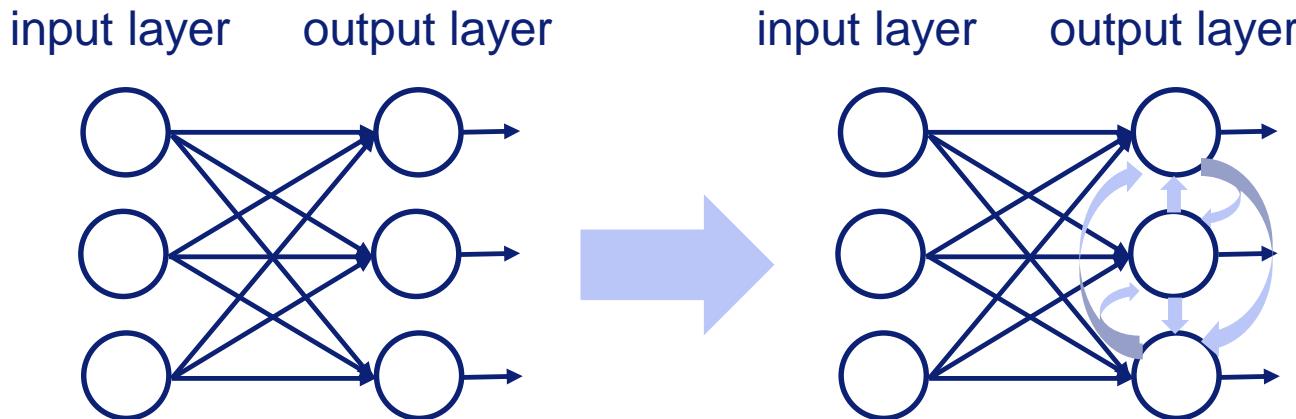
- **Positivity:** Softmax activation (strictly) *positive*: $y_j \in [0,1]$
 - **Normalization:** Outputs normalized: $\sum_{j=1}^m y_j = 1$
 - **Non-locality:** Outputs *linked* via normalization (i.e. y_j depends on *all* h_k)
 - **Monotonicity:** $\partial y_j / \partial h_k > 0$ if $j = k$, $\partial y_j / \partial h_k < 0$ if $j \neq k$
- Output from softmax layer can therefore model a **probability distribution**
- Therefore often used as output in **(multi-class) classification** problems

- **Example:**
2 classes $j = 1, 2$



Softmax activation function links output units

- Normalization links the output units:



- Softmax activation corresponds to multi-class logistic regression on h
 - Normalized probabilities for all m classes
- Example (3 classes):



vector \mathbf{x}

$$\begin{aligned}
 y_1 &= \text{Prob("dog")} = 0.6 \\
 y_2 &= \text{Prob("cat")} = 0.3 \\
 y_3 &= \text{Prob("hamster")} = 0.1
 \end{aligned}$$

Neuron models: Summary

- „Spiking“ neuron models (computationally complex!):
 - Hodgkin-Huxley model:
 - kinetic model, includes various ion channels
 - Integrate-and-fire model:
 - kinetic model, spike shape not modeled, no ion channels
 - Renewal neuron:
 - simple equation for postsynaptic potential, explicit integration of refractory function, spike shape not modeled, remembers (only) last spike
 - constant input above threshold → emits periodic spike train, period: input-dependent
 - Output characterisation: Interspike-interval distribution, gain function
- Non-spiking neuron model (computationally „simple“):
 - Threshold element / perceptron:
 - no refractory properties → no interpretation of active state as individual spike
 - binary classifier; can realise linearly separable Boolean functions
 - Extension of perceptron: more general activation function (*real* output)
- Synapses modeled as logical channels with weights (synaptic efficacies)
- Different activation functions (sigmoid, ReLU and variants, softmax...)

Further readings

- Gerstner:
 - chapter 2 (Hodgkin-Huxley model)
 - chapter 4.1 (integrate-and-fire model)
 - chapter 4.2 (spike-response-model which is closely related to renewal neuron)
 - chapter 5 (role of noise)
- Rojas, chapter 3 (perceptron)
- Amit, chapters 1.3, 1.4
- Haykin, chapter 1.3
- Müller / Reinhardt, chapters 5.2, 6.1
- Kriesel, chapters 3.2, 3.4

ARTIFICIAL NEURAL NETWORKS (introduction and architectures)

Artificial neural networks

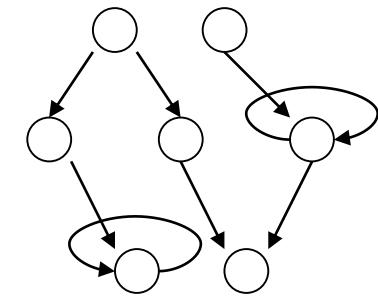
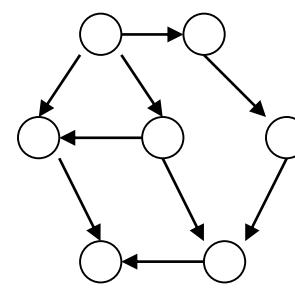
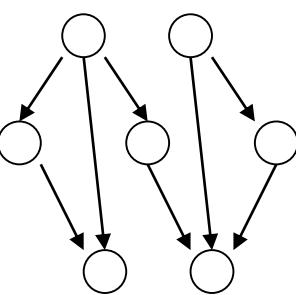
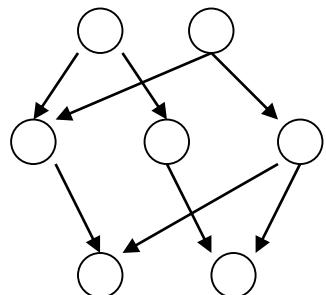
- Single artificial neuron: only small computational power
 - perceptron: can only represent **linearly separable** Boolean functions
 - „simple“ dynamic response to input

→ Increase computational power by assembling neurons in a **neural network**

- Artificial neural network: pair (N, W) with
 - N is a **set of artificial neurons**
 - W is a **set of connections (weights)** between neurons
 - the structure is a **directed graph** (nodes = neurons, edges = connections)
 - Every single neuron can receive an arbitrary set of connections (inputs)
 - Every single neuron can send its output to an arbitrary set of neurons

From: Lippe

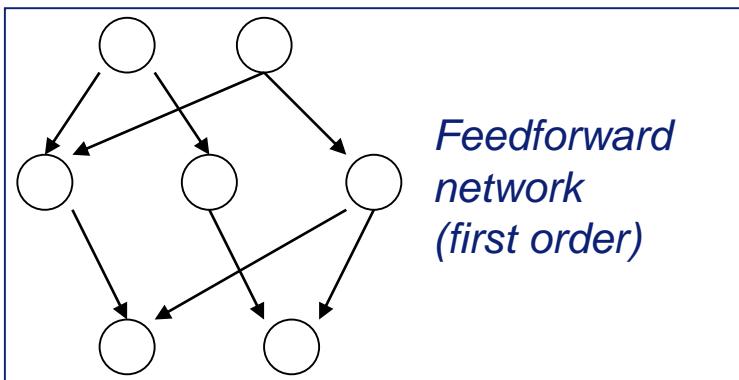
From: Hartmann



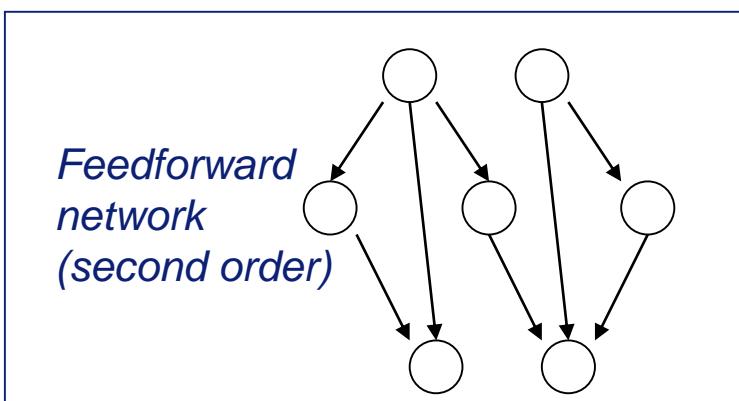
Artificial neural networks: Network architectures

Feedforward neural networks:

- Uni-directional data flow
- Layer structure: connections only from lower to higher layers



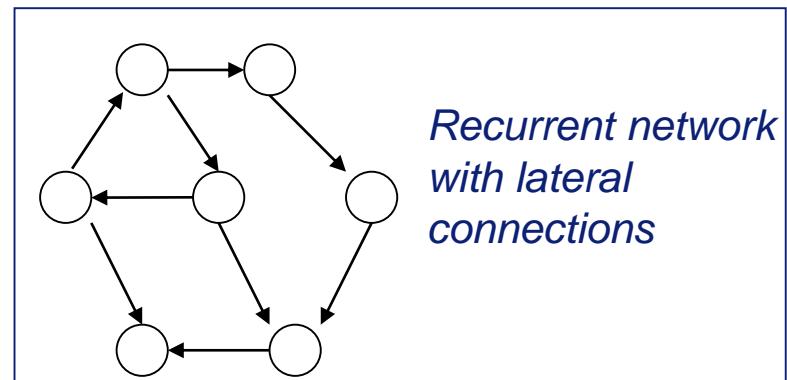
Feedforward network (first order)



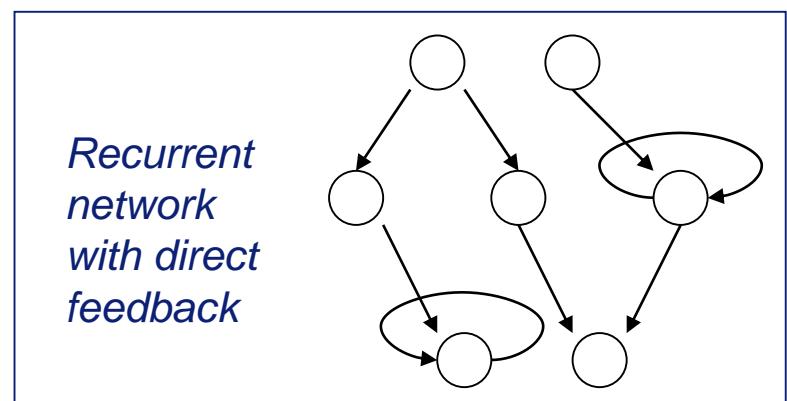
Feedforward network (second order)

Recurrent (feedback) networks:

- Bi-directional data flow
- Any connections possible; with / without layers

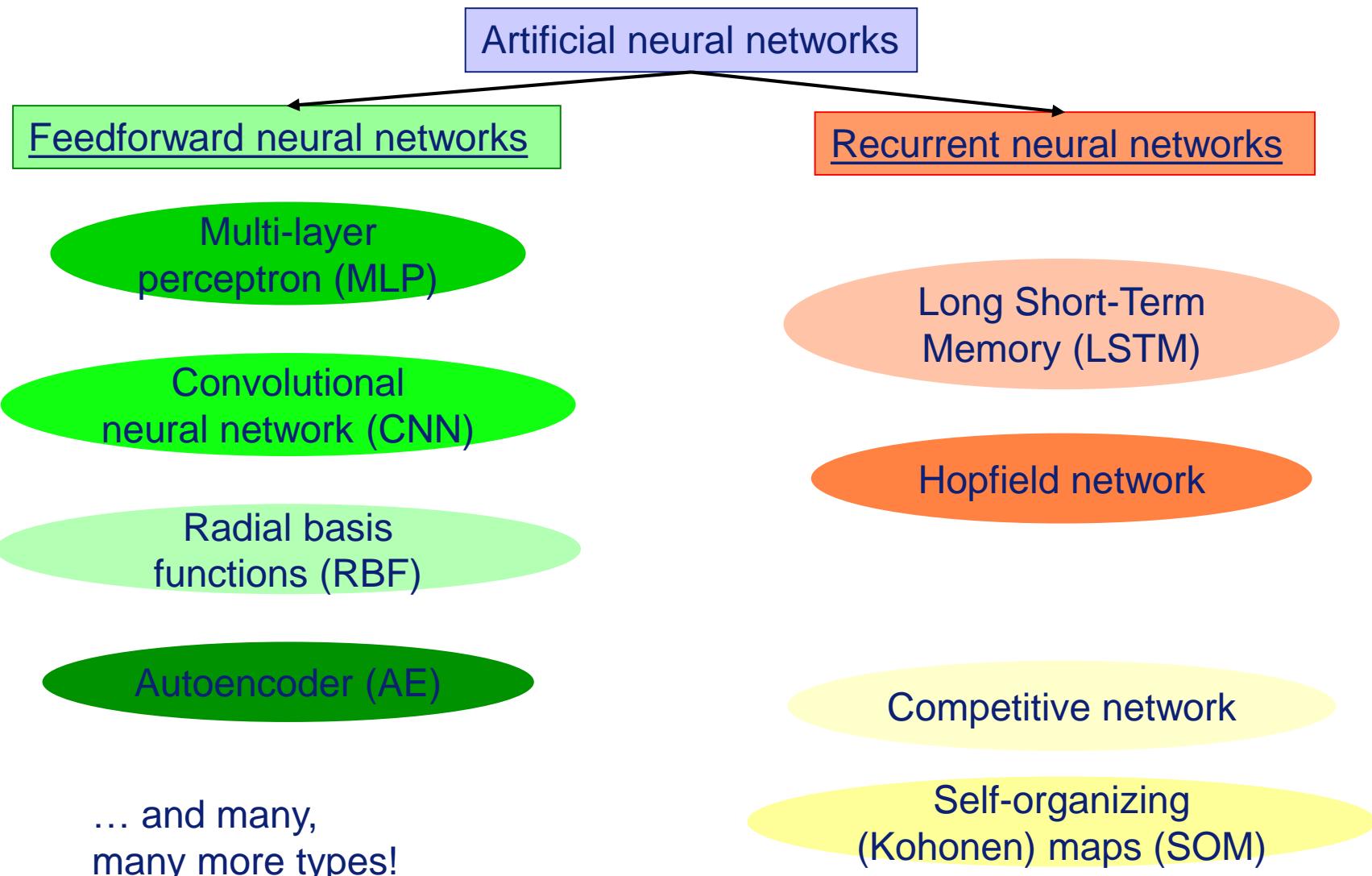


Recurrent network with lateral connections



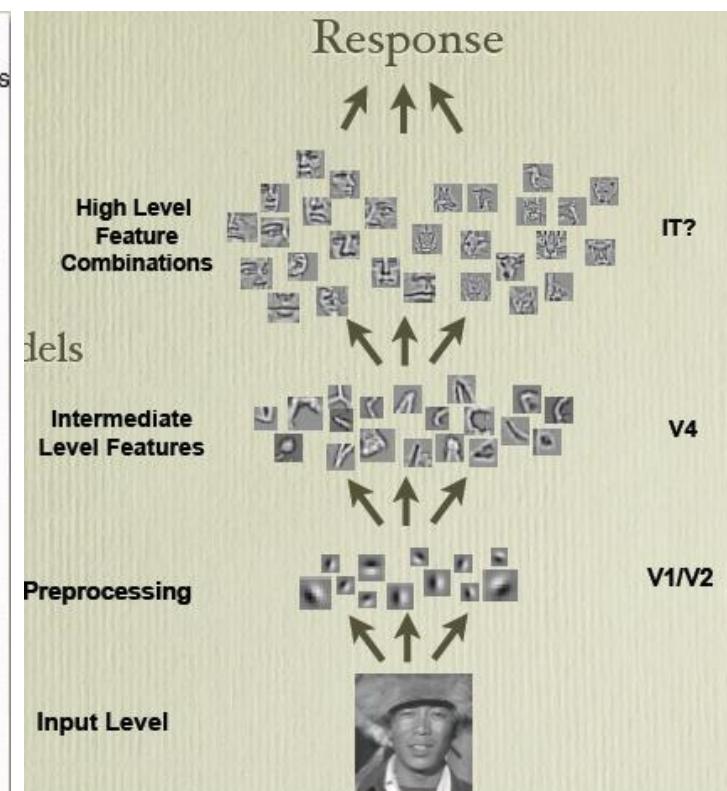
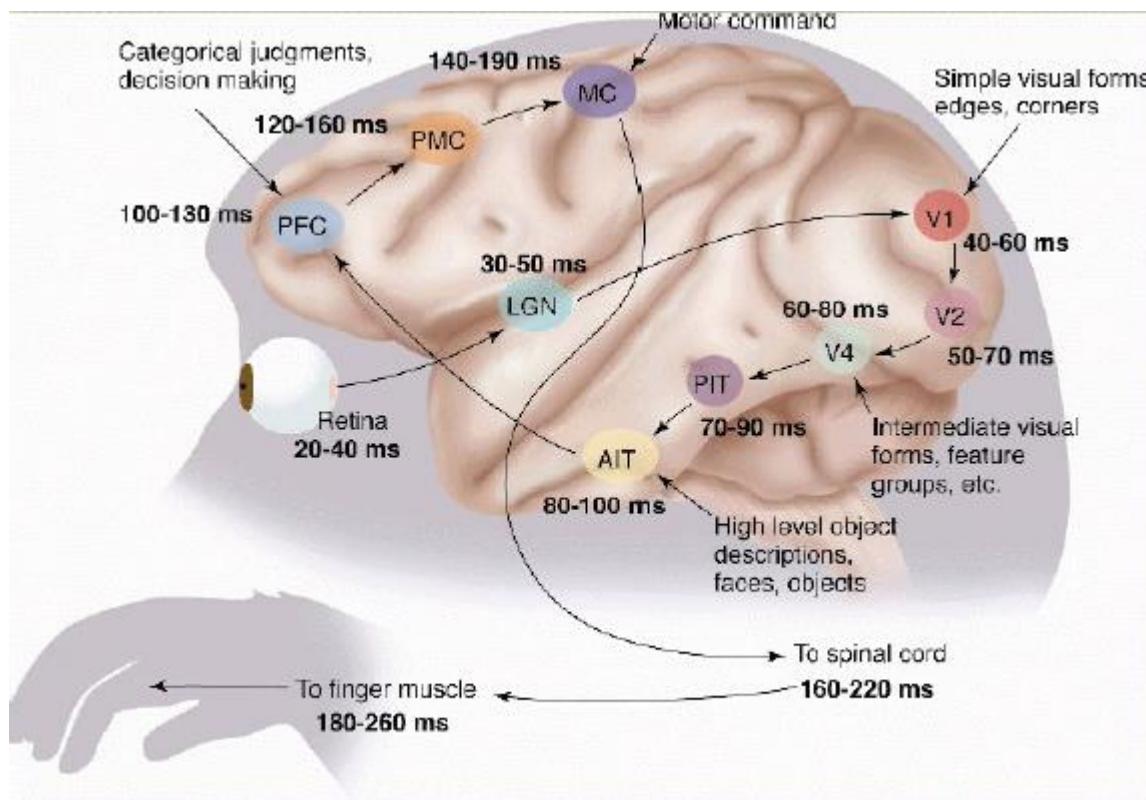
Recurrent network with direct feedback

Artificial neural networks: Examples



Network architectures in the brain

- Example: Face recognition → hierarchical network organisation
 - Feedforward architecture between hierarchy levels (?)
 - Recurrent architecture within hierarchies (?)
- Still a matter of debate ...



Artificial neural networks

- To define an artificial neural network, the following must be specified:

Topology:

- number of layers (if layers exist)
- number of neurons (per layer), especially inputs and outputs (see also later discussions)

Architecture and type:

- feedforward (MLP, CNN, ...)
- recurrent (LSTM, Hopfield, ...)

Activation function:

- sigmoid, (P)ReLU, linear, softmax...
- its parameters (if exist)

Learning algorithms

Network parameters:

- synaptic weights, biases

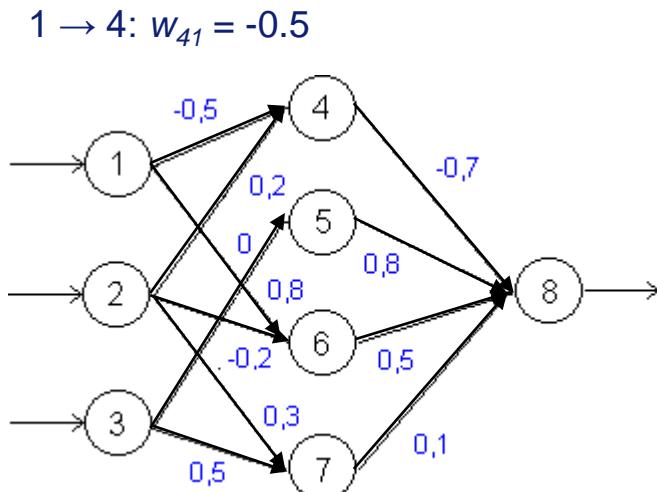
Potentially: type of dynamics (e.g. Hopfield):

- deterministic vs. stochastic
- synchronous vs. asynchronous

Artificial neural networks: Notation

- Network connections can be described by **weight matrix $W = [w_{ij}]$**
 - describes synaptic transmission neuron $j \rightarrow$ neuron i
 - $w_{ij} > 0$: **excitatory connection** from neuron j to neuron i
 - $w_{ij} < 0$: **inhibitory connection** from neuron j to neuron i
 - $w_{ij} = 0$: no connection from neuron j to neuron i
- Example: Neural network

weight matrix $W = [w_{ij}]$:



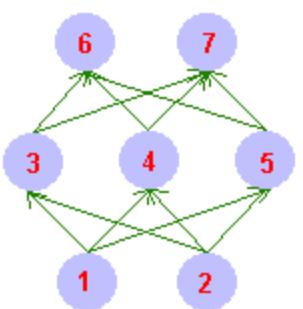
(Number in circles: number of neuron)

	1	2	3	4	5	6	7	8
1								
2								
3								
4	-0.5	0.2						
5					0			
6	0.8	-0.2						
7		0.3	0.5					
8				-0.7	0.8	0.5	0.1	

(remaining elements are 0)

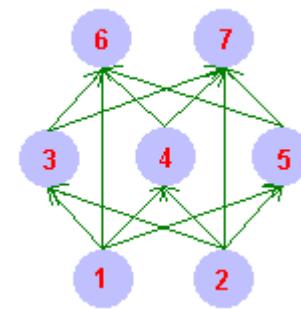
From: <http://cs.uni-muenster.de/Professoren/Lippe/lehre/skripte/wwwnnscript/bilder/netz2.gif>

Artificial neural networks: Example topologies and weight matrices



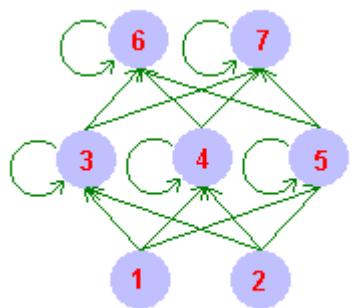
		1	2	3	4	5	6	7
1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1

Feedforward network (first order)



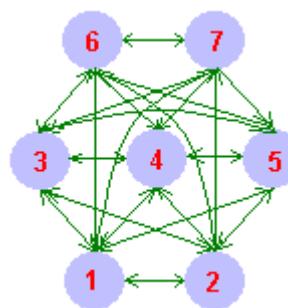
		1	2	3	4	5	6	7
1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1

Feedforward network (second order)



		1	2	3	4	5	6	7
1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1

Recurrent network with direct feedback



		1	2	3	4	5	6	7
1	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
4	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
5	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
6	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1
7	1	1	1	1	1	1	1	1
	2	1	1	1	1	1	1	1

Fully connected recurrent network (without direct feedback)

Artificial neural networks: Issues

1.) Given network topology, weights W and input (initial state):

- Computation of **network output**
 - feedforward network: state of **output layer**
 - recurrent network: **stationary state** of the network

Computation

Computer science,
Pattern recognition

Physics,
Biology

2.) Given network topology, network input / output:

- How to select the weights W to realize the input / output relations?

Learning

• Learning algorithms

Computer science

• „Storage capacity“ of the network

Physics

3.) How to select an appropriate network topology?

- (Topic not covered in this lecture)

(Design/ Learning)

Artificial neural networks, introduction and architectures: Summary

- Artificial neural network: Connected set of artificial neurons
 - structure: directed graph
 - Connections described by weight matrix $W = [w_{ij}]$
 - w_{ij} describes synaptic transmission neuron $j \rightarrow$ neuron i
- Important network types:
 - Feedforward neural network (only uni-directional data flow)
 - Recurrent neural network (including feedback, i.e. bidirectional data flow)
- The architecture, type, topology and characteristics of an artificial neural network have to be specified!
- Important issues:
 - Computation: Computation of network response given input, topology, weights
 - Learning: Selection of synaptic weights and thresholds (biases) to realize given input / output relations

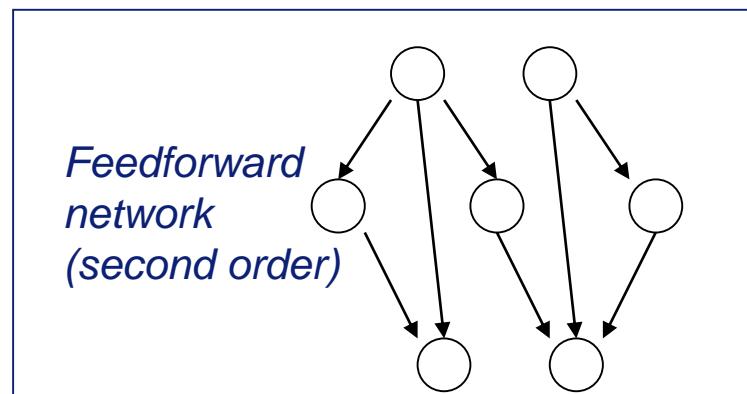
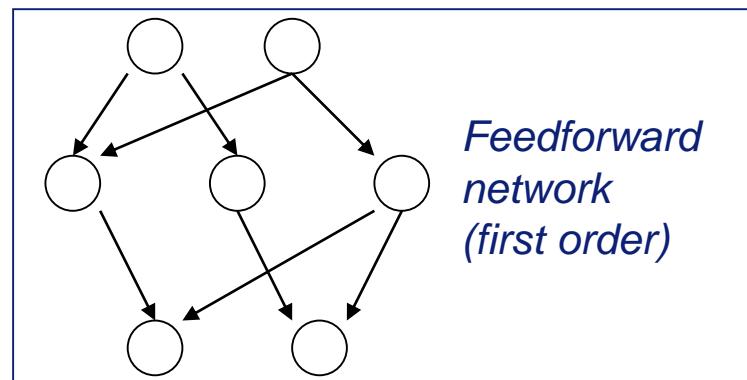
FEEDFORWARD NEURAL NETWORKS

Feedforward neural networks: Introduction

- Simple network structure: **feedforward neural network**
 - neurons organized in various layers (multi-layer)
 - connections only from lower to higher layers („feedforward“)
 - i.e. no loops / no cycles, no feedback (even no lateral feedback!)
 - no direct or indirect path from a neuron back to this neuron
 - mathematically: **directed acyclic graph**
 - first order networks (no **shortcuts**, i.e. only neighboring layers are connected) versus higher order networks
- Motivation for layer organization:
 - serialisation of information processing
 - increasingly abstract representation

Feedforward neural networks:

Directed connections only from lower to higher layers

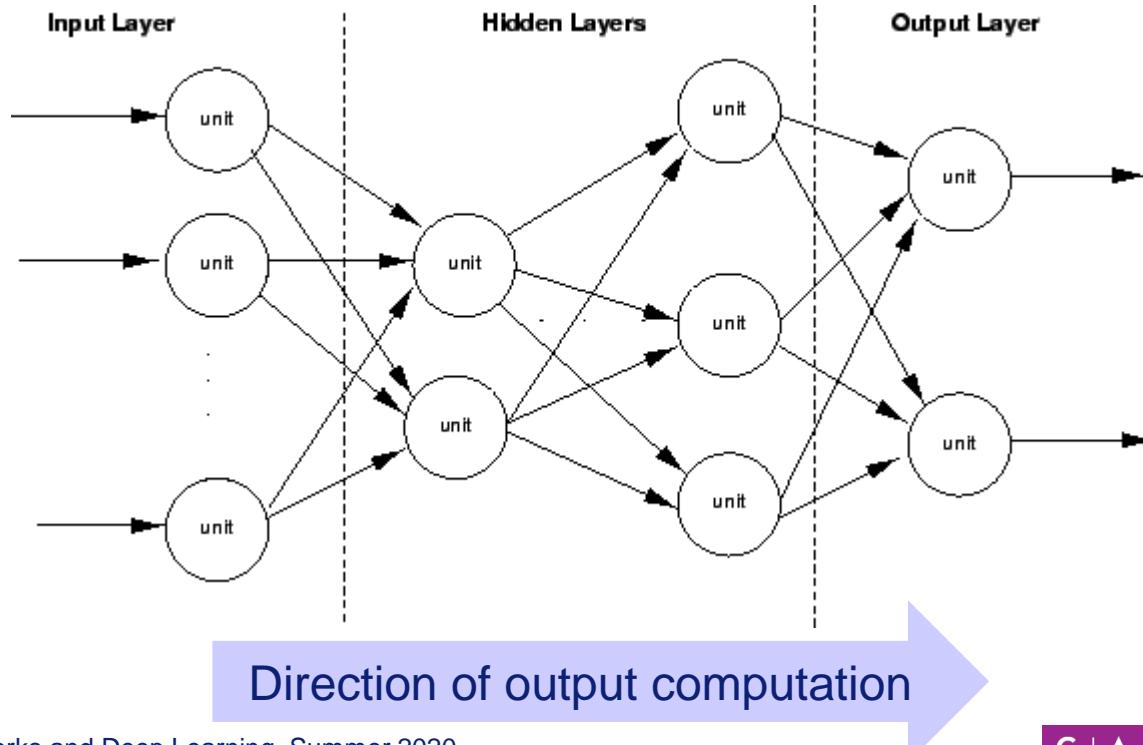


From: Hartmann

Multi-layer perceptron

Multi-layer perceptron (MLP)

- Combination of various perceptrons in a **feed-forward network**
 - Organized in **layers** (input layer, hidden layers, output layer)
 - Input layer („lowest layer“) receives input stimulus / pattern, n neurons
 - Output layer represents response of the network to input pattern, m neurons
 - Hidden layer: remaining layers; no direct contact to outside world („hidden“)
 - Each layer receives input only from lower layers (no feedback, no lateral connection)
 - Output computation follows layer structure from input to output



Multi-layer perceptron: Notation

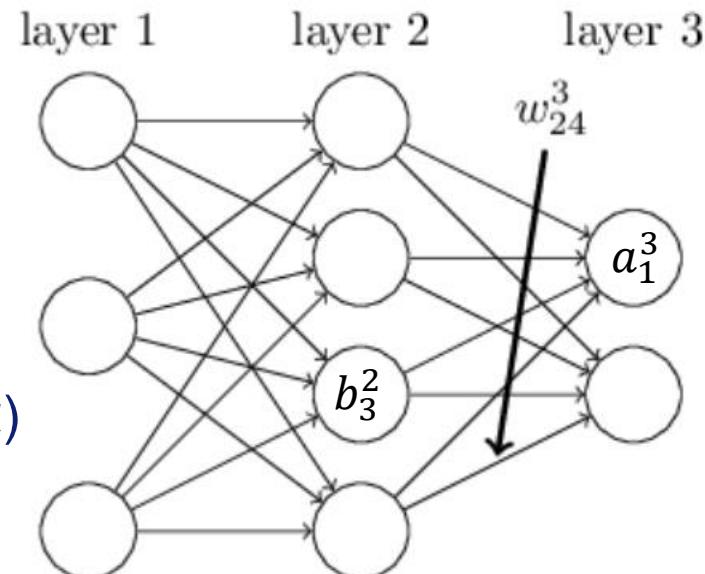
- **Layers** $l = 1, \dots, L$ (superscript)
- w_{jk}^l : **Weight** from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer (layer l contains n_l neurons)

in matrix form (row j : all weights to neuron j , layer l)

$$\mathbf{W}^l = \begin{bmatrix} w_{11}^l & w_{12}^l & \cdots & w_{1n_{l-1}}^l \\ w_{21}^l & \ddots & & w_{2n_{l-1}}^l \\ \vdots & & & \vdots \\ w_{n_l 1}^l & w_{n_l 2}^l & & w_{n_l n_{l-1}}^l \end{bmatrix}$$

- b_j^l : **Bias** of the j^{th} neuron in the l^{th} layer, in vector form:

- Output of j^{th} neuron in l^{th} layer: **activation** a_j^l , in vector form $\mathbf{a}^l =$



$$\mathbf{b}^l = \begin{bmatrix} b_1^l \\ b_2^l \\ \vdots \\ b_{n_l}^l \end{bmatrix}$$

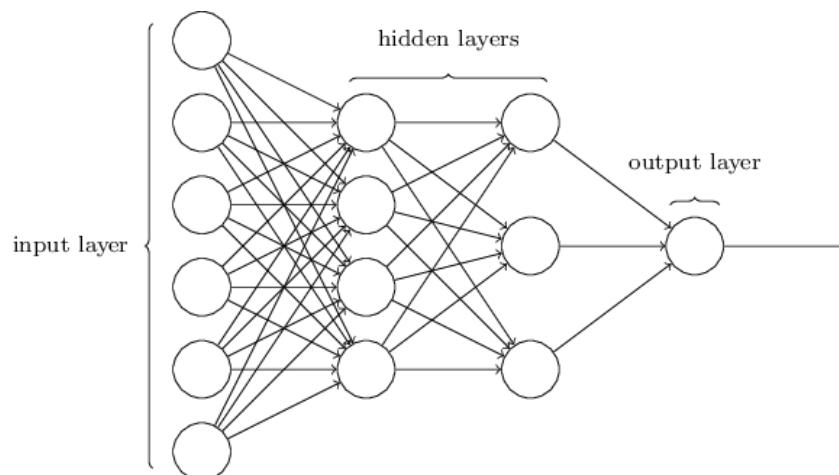
$$\mathbf{a}^l = \begin{bmatrix} a_1^l \\ a_2^l \\ \vdots \\ a_{n_l}^l \end{bmatrix}$$

Multi-layer perceptron: Output computation

- Input vector $x =: \mathbf{a}^0$
- Postsynaptic potential (PSP) / input z_j^l of neuron (“unit”) j in layer l :

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \quad \text{or} \quad \mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$$
- PSP z_j^l leads to **activation** a_j^l via **activation function** f : $a_j^l = f(z_j^l)$

$$\mathbf{a}^l = f(\mathbf{z}^l) = f(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l) \quad \text{for } l = 1, \dots, L$$
- **Network output** $\hat{\mathbf{y}}$ for input \mathbf{x} is activation of last layer L : $\hat{\mathbf{y}}(\mathbf{x}) = \mathbf{a}^L(\mathbf{x})$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{

for all neurons j in layer l :

{

Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

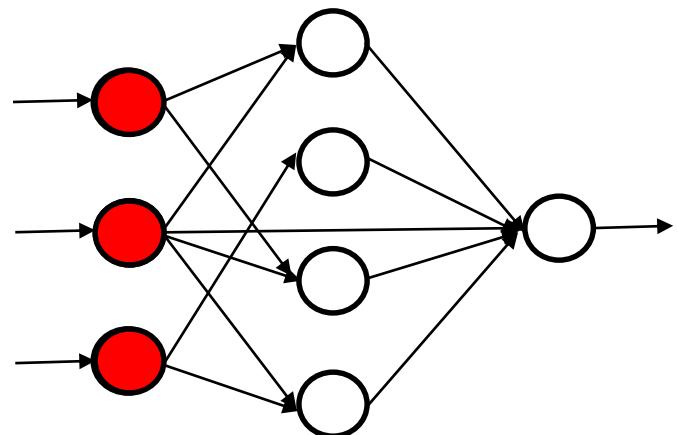
}

Example:

Red: current neuron
Grey: already computed
Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

1.) Input $l=0$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{

for all neurons j in layer l :

{

Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

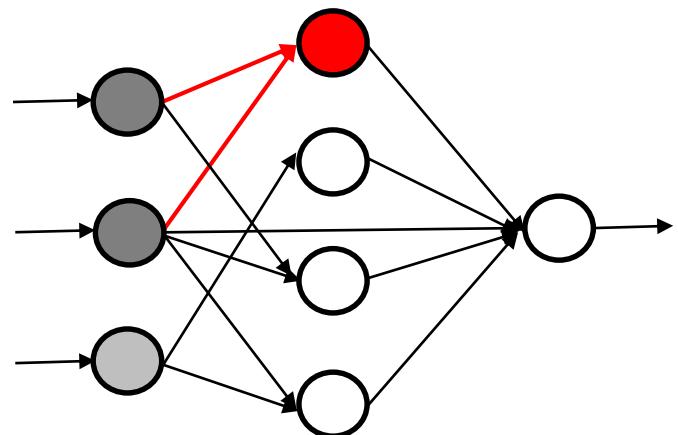
}

Example:

Red: current neuron
Grey: already computed
Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

2.) Layer $l=1$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{
 for all neurons j in layer l :
 {
 Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

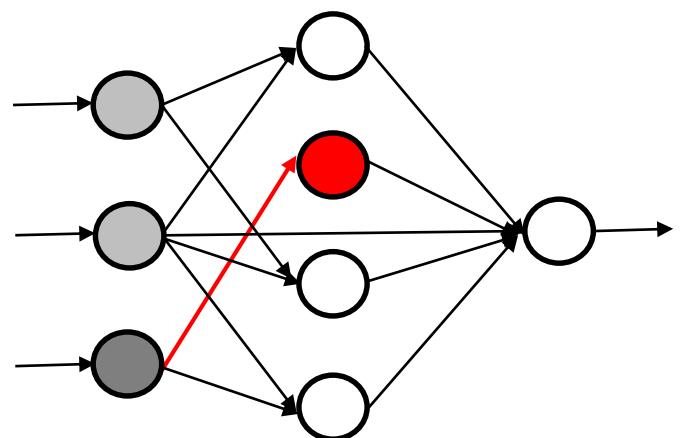
$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

Example: Red: current neuron
 Grey: already computed
 Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

2.) Layer $l=1$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{
 for all neurons j in layer l :
 {
 Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

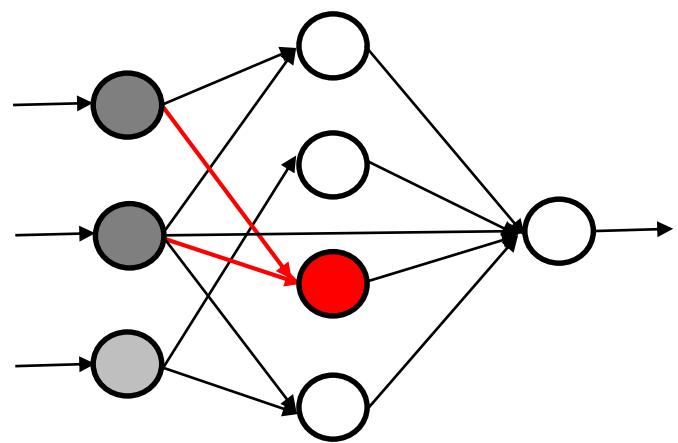
$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

Example: Red: current neuron
 Grey: already computed
 Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

2.) Layer $l=1$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{

for all neurons j in layer l :

{

Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

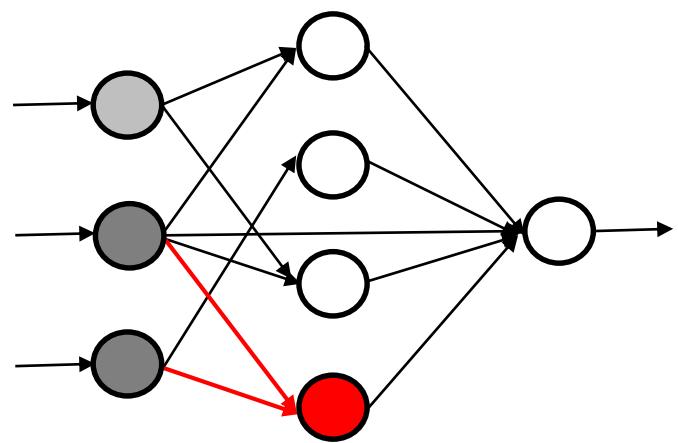
}

Example:

Red: current neuron
Grey: already computed
Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

2.) Layer $l=1$



Computing a multi-layer perceptron

- parallel (for neurons within a layer), i.e. at time t , for layer l (order arbitrary)
- sequential (from layer to layer), i.e. $t_l \rightarrow t_{l+1}$

Input layer $l=0$, output layer $l=L$:

RF(j): set of predecessors („receptive field“) of neuron j (which is in layer l)

PSP: postsynaptic potential

for layers $l=1, \dots, L$

{

for all neurons j in layer l :

{

Compute PSP z_j^l and activation a_j^l :

$$z_j^l = \sum_{k \in RF(j^l)} w_{jk}^l a_k^{l-1} + b_j^l$$

$$\mathbf{a}^l = f(\mathbf{z}^l)$$

}

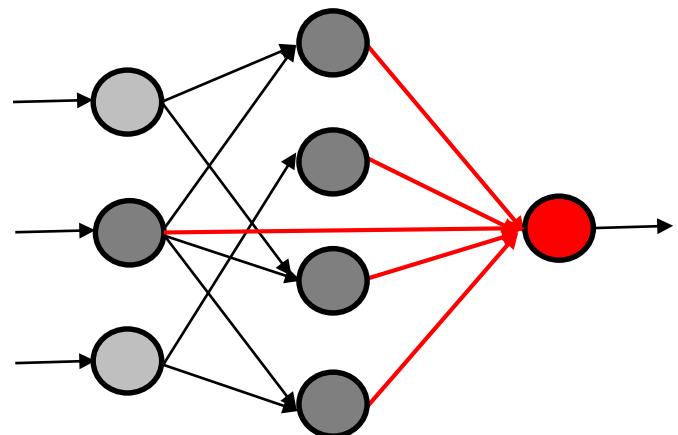
}

Example:

Red: current neuron
Grey: already computed
Dark grey: receptive field

Input layer ($l=0$)	Hidden layer ($l=1$)	Output layer ($l=2$)
--------------------------	---------------------------	---------------------------

3) Output layer $l=2$

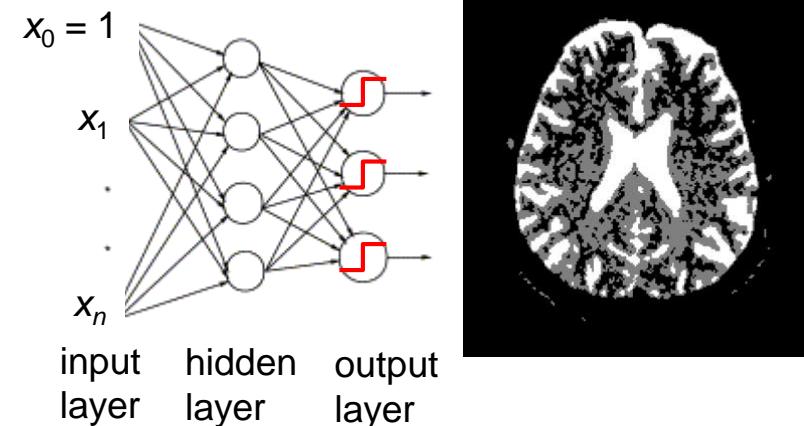


Multi-layer perceptron: Application examples

1) Classification

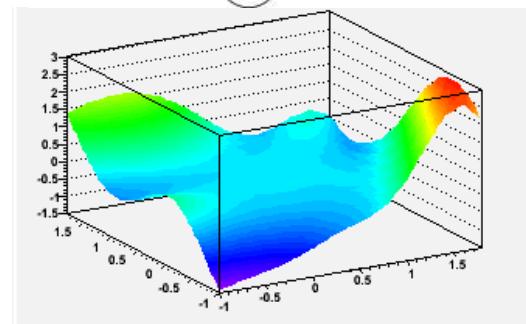
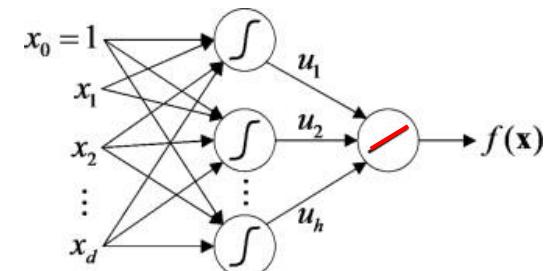
- Output activation function:
 - Step function (i.e. **binary** output)
 - Sigmoid, softmax (**probability** for a class)
- Number of output neurons: $m \geq 1$
 - E.g. number of classes
 - Or some binary code

Example: 3 classes



2) Regression / function approximation

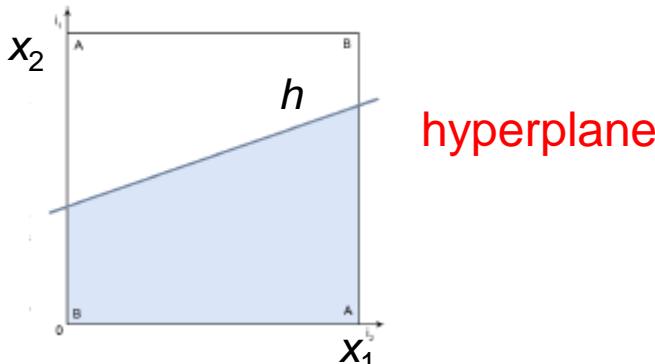
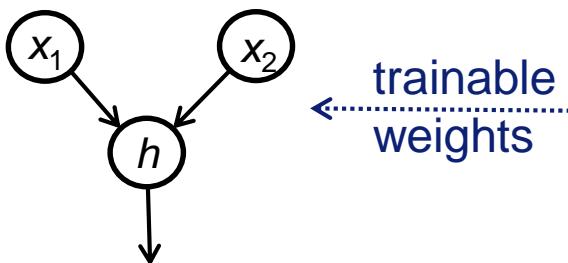
- Activation function: In general sigmoid / linear
 - i.e. **real** output
- Number of output neurons: $m \geq 1$
 - i.e. output $\in \mathbb{R}^m$
 - m depends on dimensionality of regression / function approximation problem



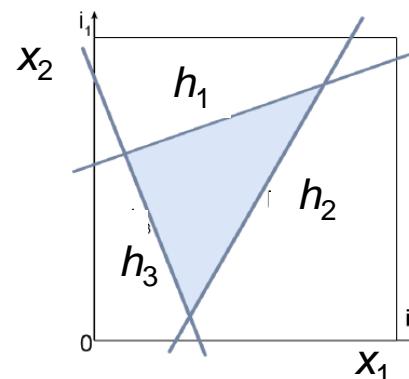
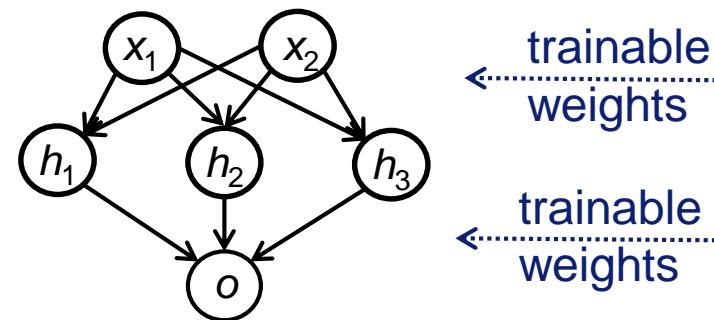
Classification with multi-layer perceptrons (1)

- Remember: Single-layer perceptron (one set of trainable weights): represents data separated by hyperplane (linearly separable)
 - Multi-layer perceptron with one hidden layer (two sets of trainable weights): combination of hyperplanes into (convex) polygon

Single-layer perceptron



Multi-layer perceptron (one hidden layer)

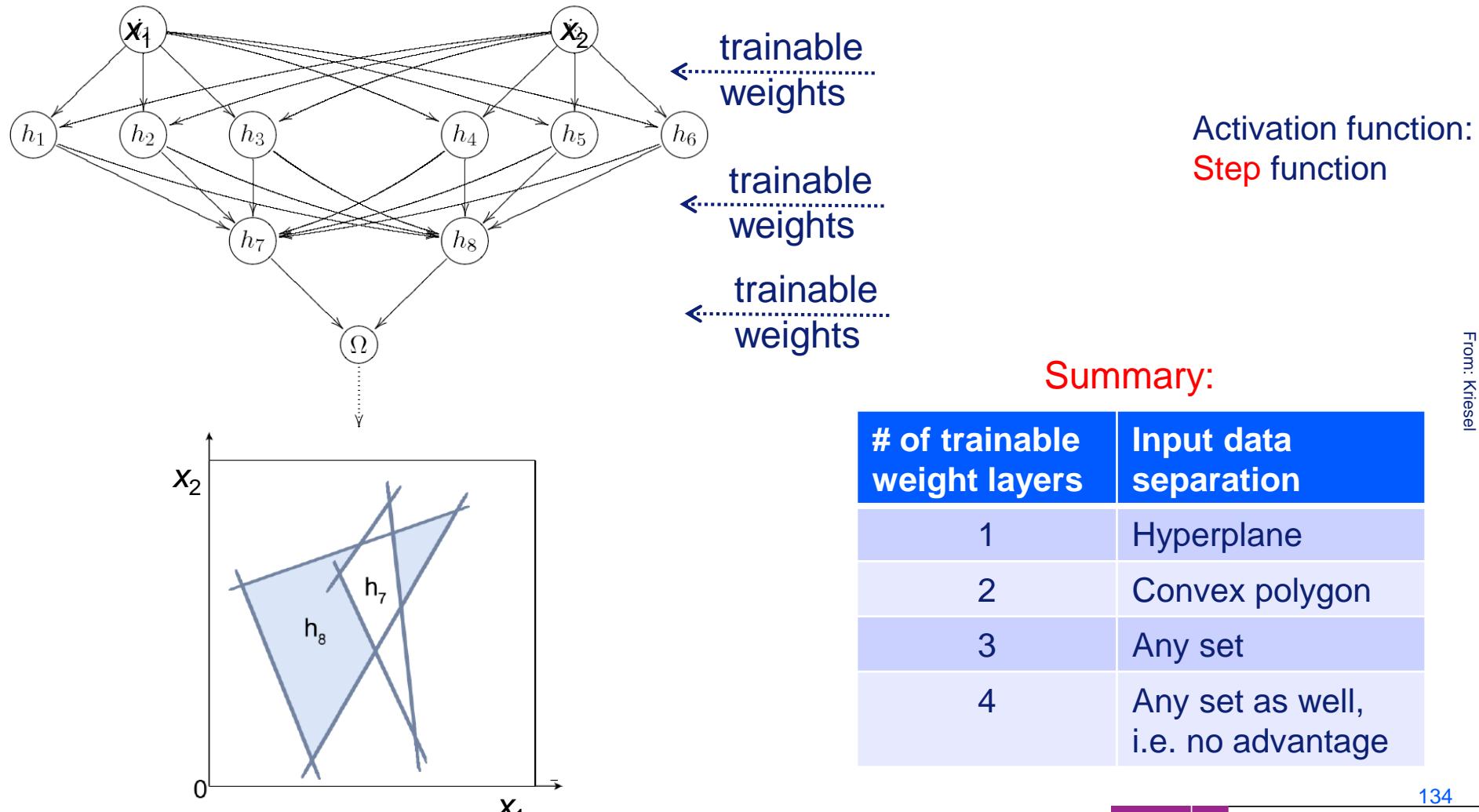


Polygon

Activation function: Step function

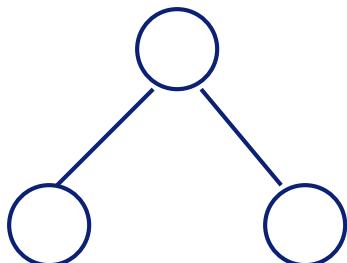
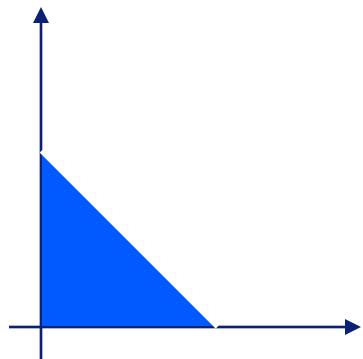
Classification with multi-layer perceptrons (2)

- Multi-layer perceptron with *two hidden layers* (three sets of trainable weights): combination of (convex) polygons into **arbitrary point sets**

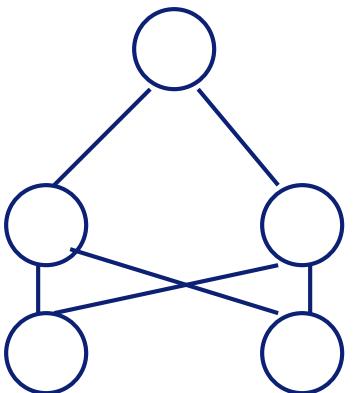
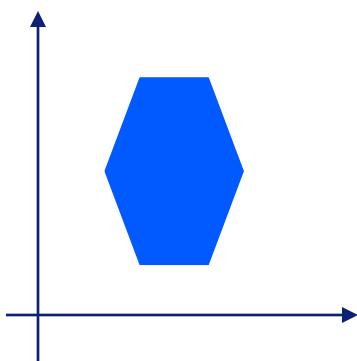


From: Kriegel

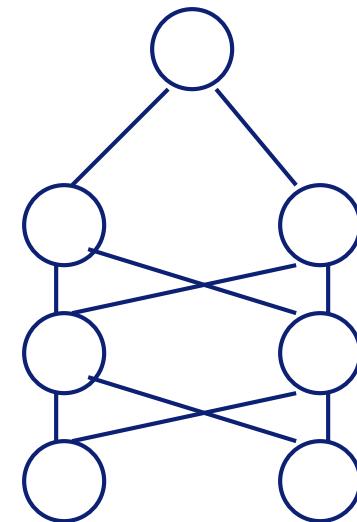
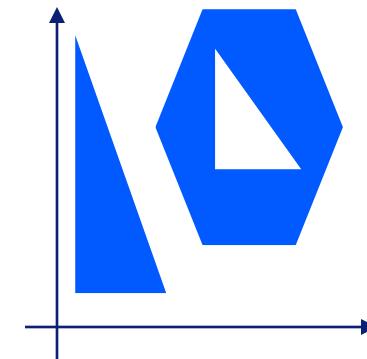
What do each of the layers do?



1st layer draws linear boundaries



2nd layer combines the boundaries



3rd layer can generate arbitrarily complex boundaries

Regression: Multi-layer perceptron as universal function approximator

Consider multi-layer perceptron with **single hidden layer**

- Non-constant, bounded, monotonically-increasing activation function

Universal function approximator (Cybenko, 1989)

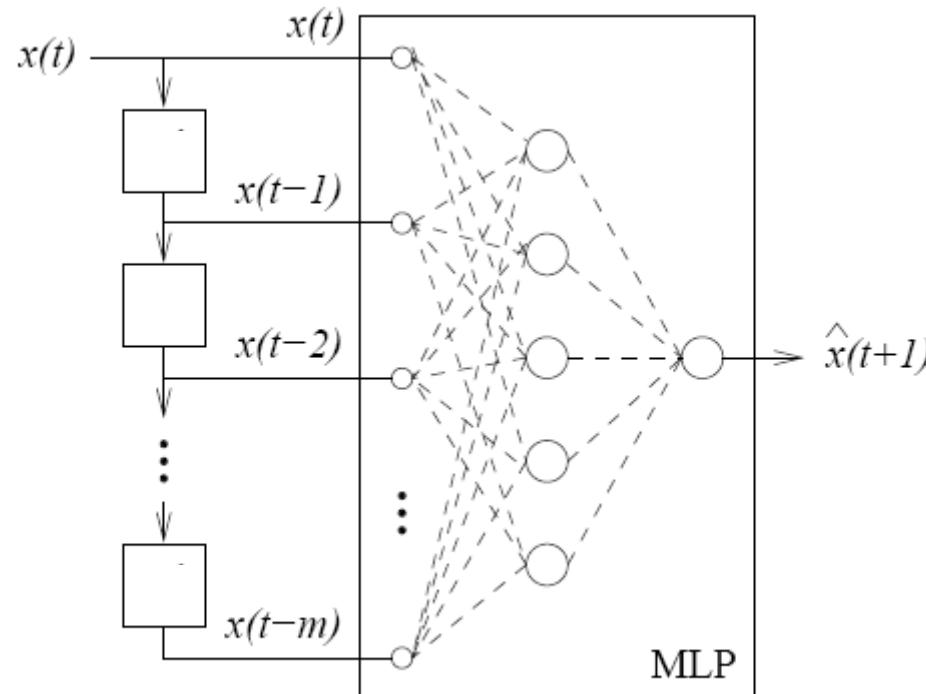
- i.e. an arbitrary continuous function on a compact interval can be approximated with any desired accuracy
- Guarantees broad applicability of a multi-layer perceptron (in contrast to a single-layer perceptron which is limited to linearly separable data)
- However, this only guarantees the **existence** of a multi-layer perceptron; it is not specified how to choose the number of hidden neurons!

Summary: Multi-layer perceptron:

- Complex enough to represent complex (linearly inseparable) functions
- Simple enough to be „tractable“ (i.e. a „learning algorithm“ exists, see next)

Example for regression: Time series prediction

- Task: Learning to predict time series $x(t+1) = f(x(t), x(t-1), x(t-2), \dots)$
- Simple idea: $x(t), x(t-1), \dots, x(t-m)$ as input of a **multi-layer perceptron**
 - m input unit, h hidden units, 1 output unit (linear, sigmoid etc.)



- But: *fixed* time window (length m), restricted context (m identical previous values leads to same output, independent of long-range context)

From: Schwenker

Feedforward neural networks: Summary

- **Uni-directional data flow** (network structure: directed acyclic graph)
- **Layer structure**: input layer, hidden layers, output layer

Examples:

- **Multi-layer perceptron (MLP)** (can have various hidden layers)
 - Universal function approximator: an arbitrary continuous function on a compact interval can be approximated with any desired accuracy

Further examples (discussed later):

- **Convolutional neural networks**
- **Radial basis function (RBF) networks** (single hidden layer)

LEARNING IN NEURAL NETWORKS

Learning in neural networks: Motivation

- So far: assuming **given network structure** (architecture, synaptic weights): given input $x \rightarrow$ calculate network output y
- Problem: How to **specify the network structure**?
- In particular: How to specify the **synaptic weights / thresholds**?
- **Goal:** Algorithms for specifying the parameters of a neural network in order to solve a given task: „learning“ / „training“
- Learning algorithms depend on the network type, e.g.:
 - Feedforward network: Iterative algorithms („delta rule“), e.g. backpropagation
 - Hopfield network: Direct specification of synaptic weights from wanted patterns („Hebbian learning“)

Representing input / output relations

Variant a): *Explicit formulation*

- Input / output relationship must be known explicitly (formulated as rule)
- Explicit formulation as mathematical relation, mapping, algorithm etc.
→ *explicit* computer program

Variant b): *Implicit learning from examples*

- Input / output relationship not known in advance, but **learned from examples (training set)**
 - Input / output relationship represented in the parameters of the learner
→ *Training phase* of a neural network!
- **Advantage:** No explicit algorithm to solve a specific task necessary!
- Instead:
 - Presentation of (suitable) data to the network
 - Solution of a specific task by application of a general learning algorithm

Learning in the context of neural networks

- In the context of neural networks, „learning“ refers to **specifying the organization of the network** (connectivity, neuronal elements etc.) in such a way that a desired network response is achieved for a given set of input patterns (the „*training set*“)
- We will not deal with the dynamical process of learning itself
 - Instead, we assume that there is a separate „*training phase*“ in which the network organization is specified, and a subsequent „*test phase*“ in which the network is run on new examples using the pre-defined network organization
- The hope is that by learning from examples, the network implicitly detects some structure in the data so that for a new (previously unseen) input pattern a correct response is generated („**generalization**“)
 - Necessary condition: the training set is representative of the overall data

Learning what?

- Number of layers
- Number of neurons

Learning of
topology

- Synaptic weights
- Thresholds (biases)

Learning of
weights

- Activation function
- Parameters (e.g. of activation function)

Learning of
model

Learning what?

- Number of layers
- Number of neurons

Learning of
topology

- Synaptic weights
- Thresholds (biases)

Learning of
weights

- Activation function
- Parameters (e.g. of activation function)

Learning of
model

Learning from examples: Paradigms

What is presented to the network:

- **Supervised learning:**
 - Input patterns *and* corresponding target outputs are presented to the network
 - The network adjusts its parameters to realize input / output pairs
- **Unsupervised learning:**
 - Only input patterns are presented to the network (without target output)
 - The network detects similarities / generates classes by itself and adapts its parameters to them
- Reinforcement learning:
 - Training set: input patterns and assessment of network output (success/failure)

How is it presented to the network:

- **Online (incremental) learning:**
 - Learning (parameter modification) after every training sample
- **Offline (batch) learning:**
 - Learning (parameter modification) only after presenting *all* training samples

Or compromise
(„mini-batch“)

Learning in neural networks: Supervised and unsupervised learning

Supervised learning

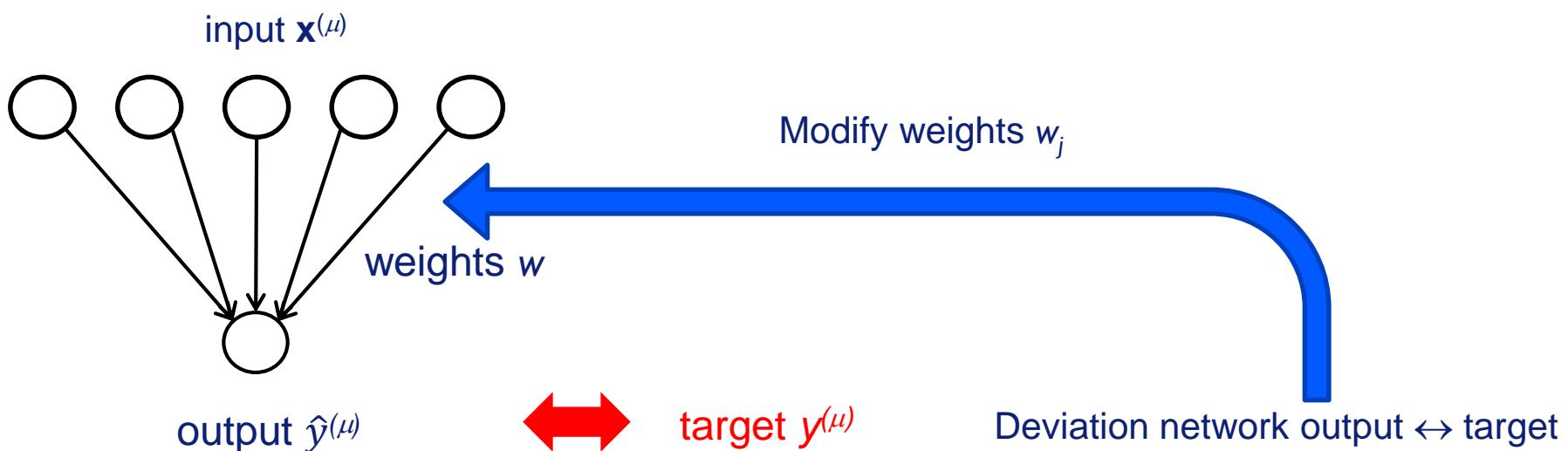
- „Teacher“, i.e. correct answers, required
- Learning by trying to match calculated output with target
- Disadvantage: Needs annotated data (big manual effort!)
- Examples:
 - Perceptron learning
 - Backpropagation

Unsupervised learning

- Does not need correct answers, learns from inputs
- Learning by „association“
- Advantage: Un-annotated data are cheap and abundant
- Examples:
 - Winner takes all (WTA)
 - Self-organizing maps (SOM)
 - Autoencoder

Supervised learning

- Learning supervised by „teacher“ (i.e. the correct outputs $y^{(\mu)}$ are known)
- Training set: $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(p)}, y^{(p)})\}$
- Network is exposed to samples, presents its output $\hat{y}^{(\mu)}$ for each input $\mathbf{x}^{(\mu)}$
- The target of learning is to minimize the discrepancy between the actual network output $\hat{y}^{(\mu)}$ and the correct target output $y^{(\mu)}$
- General procedure: Define an **error function E** and search for the set of network parameters (synaptic connections) minimizing the error



Training and generalization error

- **Error** between network output $\hat{y}(\mathbf{x})$ and target y (other definitions possible):

$$\text{classification: } E(\mathbf{x}) = 1 - \delta(y - \hat{y}(\mathbf{x}))$$

$$\delta(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} = 0 \\ 0 & \text{if } \mathbf{x} \neq 0 \end{cases}$$

$$\text{regression: } E(\mathbf{x}) = (y - \hat{y}(\mathbf{x}))^2$$

↑
(sum squared error SSE)

- **Training error:** computed on training set $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(p)}, y^{(p)})\}$
 - Depends on choice of training set D !

$$E_{train}(\mathbf{x}) = \sum_{\mu=1}^p E(\mathbf{x}^{(\mu)}) \quad \text{sum over training patterns } \mu = 1, \dots, p$$

- Goal of learning: Small error on **unseen examples** $\mathbf{x} \notin D$: generalization
- **Generalization error:** computed on **all** inputs \mathbf{x} (or new, randomly chosen \mathbf{x})

$$E_{generalisation}(\mathbf{x}) = \int E(\mathbf{x}) P(\mathbf{x}) d\mathbf{x} \quad \text{(expectation value over all possible } \mathbf{x})$$

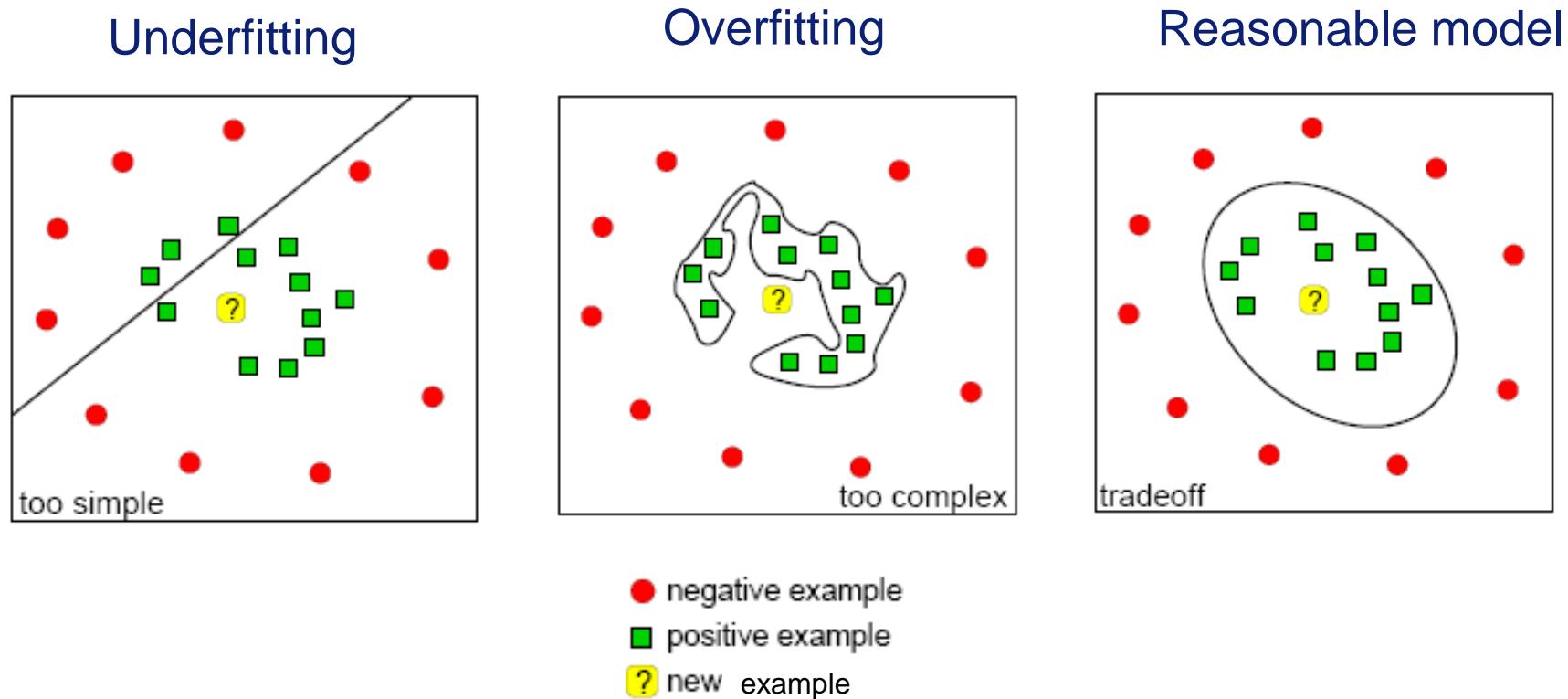
- **Generalisation error unknown** → must be approximated

Training and validation

- Practical approach: Obtain *independent* measure of generalisation error on an independent **validation set**
 - Before training, split available data into disjoint training and validation set
 - Apply training on training set
 - Test learned hypothesis on validation set (→ **validation error**)
 - Advantage:
 - Indicator whether hypothesis really improves (on independent data) or just learns specifics of training data („overfitting“)
- Overfitting: Details of some training patterns are learned which are not relevant for most of the remaining patterns („too detailed“)
 - Underfitting: Model / hypothesis are not detailed enough
- Disadvantages:
 - Division into training / validation set reduces available data for training
 - Unclear whether validation error is a good estimate of generalisation error

Overfitting and underfitting: Illustration

- Overfitting: Details of some training patterns are learned which are not relevant for most of the remaining patterns („too detailed“)
- Underfitting: Model / hypothesis are not detailed enough

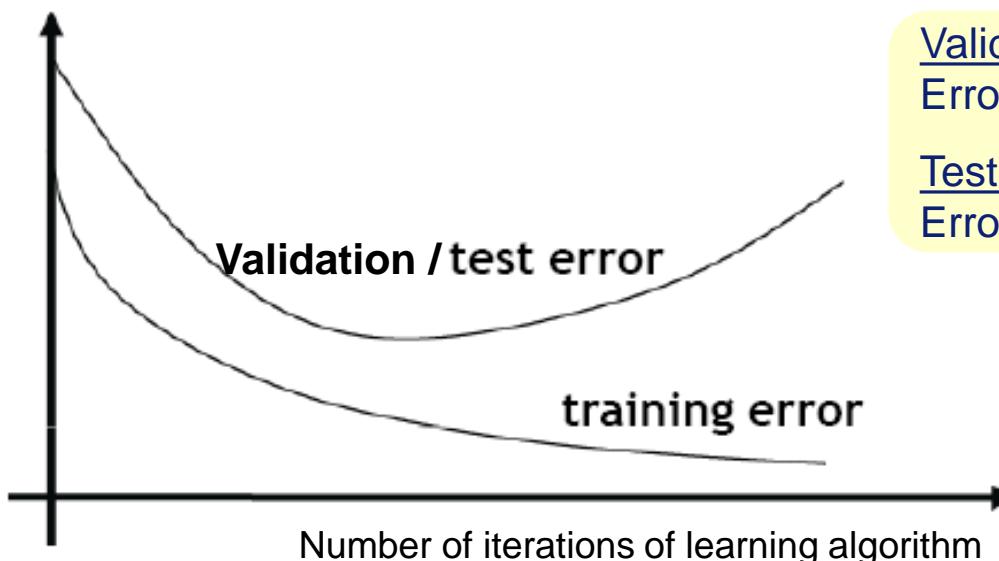


From: Markowetz

150

Generalisation and overfitting

- Goal of learning: Predict labels / output on **new observations**
 - Train classification model on limited training set
 - The further we optimize the model parameters, the more the **training error** will decrease (until saturation)
 - However, at some point the generalisation (validation / test) error may go up again
- **Overfitting to the training set**



Validation error:
Error on (indep.) validation set

Test error:
Error on independent test set

Overfitting and underfitting: Training / test error behaviour

1. Example of **overfitting**:

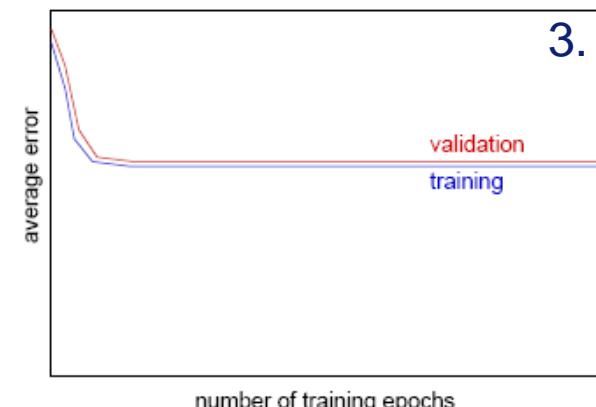
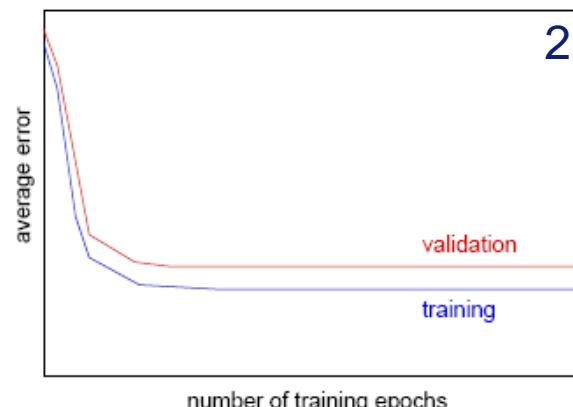
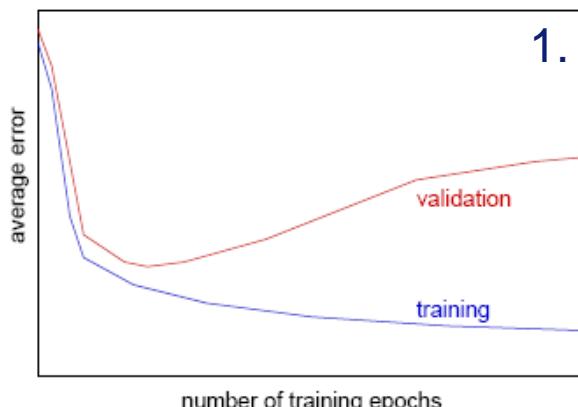
validation error increases while training error decreases

2. Example of **successful learning**:

validation error and training error monotonically decrease

3. Example of **underfitting**:

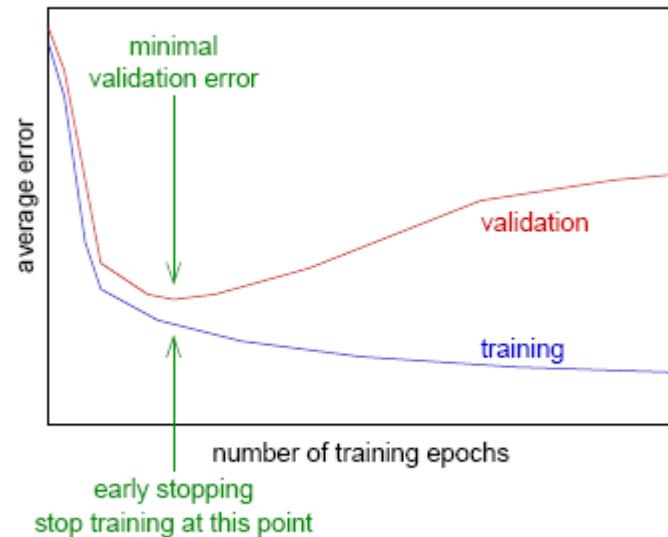
validation error and training error remain large



From: Riedmiller

Some techniques to avoid overfitting

- Select appropriate and sufficient training data
 - Representing the full problem sufficiently
- Early stopping
 - Stop training when validation error has reached minimum
- Regularisation
 - Reducing the degrees of freedom (parameters); see later



Cross-validation

- Extension of the validation set method:

k-fold cross validation:

Require: training set D with p input patterns, integer k with $2 \leq k \leq p$

1. split D into k disjoint subsets of (approx.) equal size: D_1, \dots, D_k
2. **for** $i = 1$ to k **do**
3. Initialize learning algorithm
4. Apply learning algorithm on sets $D_1 \cup \dots \cup D_{i-1} \cup D_{i+1} \cup \dots \cup D_k$
5. calculate test error e_i on D_i
6. **end for**
7. **return** weighted average of test errors according to size of sets D_i

- Advantage: model learned on $\frac{k-1}{k}p$ examples, evaluated on all examples
- Disadvantage: model has to be trained k times
- k -fold cross-validation with $k = p$ yields **leave-one-out error**
- Note: If subsets do not have equal size: Take **weighted** error average

Cross-validation: Example

- 5-fold cross-validation:
 - Split available data into 5 sets
 - Training on 4 sets, test on remaining set
 - Repeat in cyclic fashion

train	train	train	train	test
train	train	train	test	train
train	train	test	train	train
train	test	train	train	train
test	train	train	train	train

Perceptron learning

Perceptron learning: Idea

Binary threshold activation function: $f(h) = \Theta[h]$

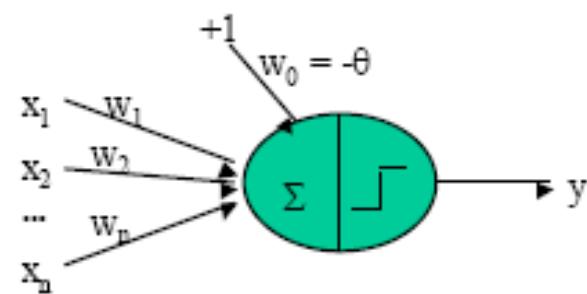
Iterate over training set:

Present example $\mathbf{x}^{(\mu)}$ and calculate perceptron output $\hat{y}^{(\mu)} = \Theta[\mathbf{w}(t) \cdot \mathbf{x}^{(\mu)}]$

- if output is **correct** ($\hat{y}^{(\mu)} = y^{(\mu)}$) \Rightarrow **no weight modification**
- if output is **incorrect** ($\hat{y}^{(\mu)} \neq y^{(\mu)}$) \Rightarrow **modify weights** by subtracting (adding) the input sample from $\mathbf{w}(t)$ depending on type of error:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{(\mathbf{y}^{(\mu)} - \hat{y}^{(\mu)}) \cdot \mathbf{x}^{(\mu)}}_{\Delta \mathbf{w}(t) = 0 \text{ or } \pm \eta \mathbf{x}^{(\mu)}} \quad \eta: \text{„learning rate“} \quad (\text{step size of weight modification})$$

- If $\hat{y}^{(\mu)} = 1$, but $y^{(\mu)} = 0$ („false positive“) $\Rightarrow \mathbf{w}(t+1) = \mathbf{w}(t) - \eta \mathbf{x}^{(\mu)}$
i.e. reduce synaptic weights so that $\mathbf{w}(t) \cdot \mathbf{x}^{(\mu)}$ is reduced
- If $\hat{y}^{(\mu)} = 0$, but $y^{(\mu)} = 1$ („false negative“) $\Rightarrow \mathbf{w}(t+1) = \mathbf{w}(t) + \eta \mathbf{x}^{(\mu)}$
i.e. increase synaptic weights so that $\mathbf{w}(t) \cdot \mathbf{x}^{(\mu)}$ is increased
- Proceed with next example (online learning)



Perceptron learning: Algorithm

Given: (Linearly separable) training set $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(p)}, y^{(p)})\}$

1. Initialise all weights w_j (e.g. small random numbers)
2. Select training pattern $(\mathbf{x}^{(\mu)}, y^{(\mu)}) \in D$ (online learning):
 - Calculate perceptron output $\hat{y}^{(\mu)}$ for input $\mathbf{x}^{(\mu)}$: $\hat{y}^{(\mu)} = \Theta[\mathbf{w}(t) \cdot \mathbf{x}^{(\mu)}]$
 - Calculate deviation $\varepsilon^{(\mu)}$ between output $\hat{y}^{(\mu)}$ and target output $y^{(\mu)}$: $\varepsilon^{(\mu)} = y^{(\mu)} - \hat{y}^{(\mu)}$
 - Modify weights according to $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \varepsilon^{(\mu)} \cdot \mathbf{x}^{(\mu)}$, i.e.
 - Synaptic weights: $w_i(t+1) = w_i(t) + \eta \cdot \varepsilon^{(\mu)} \cdot x_i^{(\mu)}$
 - Threshold $\theta(t+1) = \theta(t) - \eta \cdot \varepsilon^{(\mu)}$ (since $w_0 = -\theta$)
 - Batch learning: all training samples, sum over error terms
3. If there are still training patterns in D which are classified incorrectly:
GOTO 2 else END

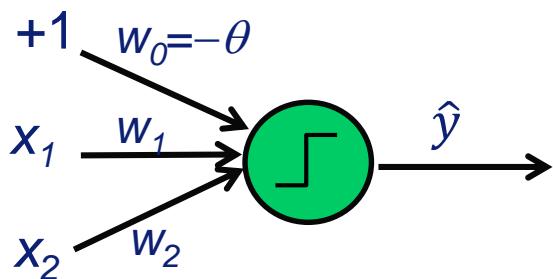
Perceptron Convergence Theorem:

If the classification problem is linearly separable, then the perceptron learning algorithm finds a solution (weight vector) after a finite number of steps.

(XOR can *not* be learned!)

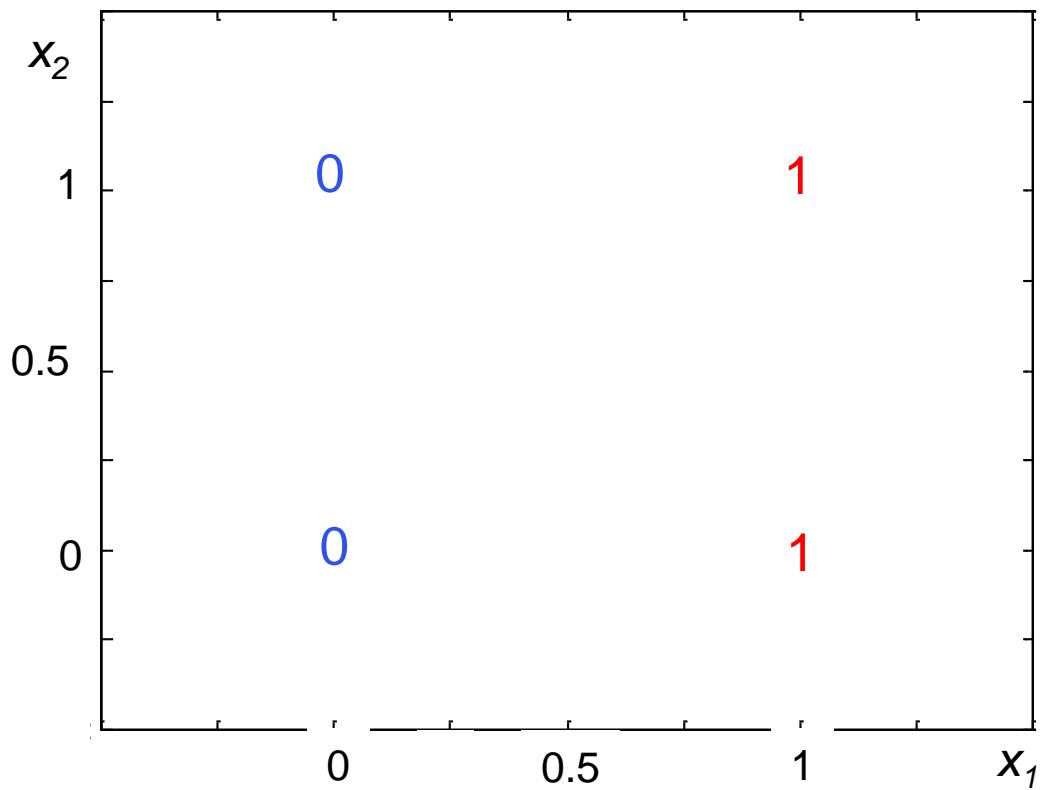
Perceptron learning: Example

- Perceptron with two inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Blue: target perceptron output 0, red: target perceptron output 1



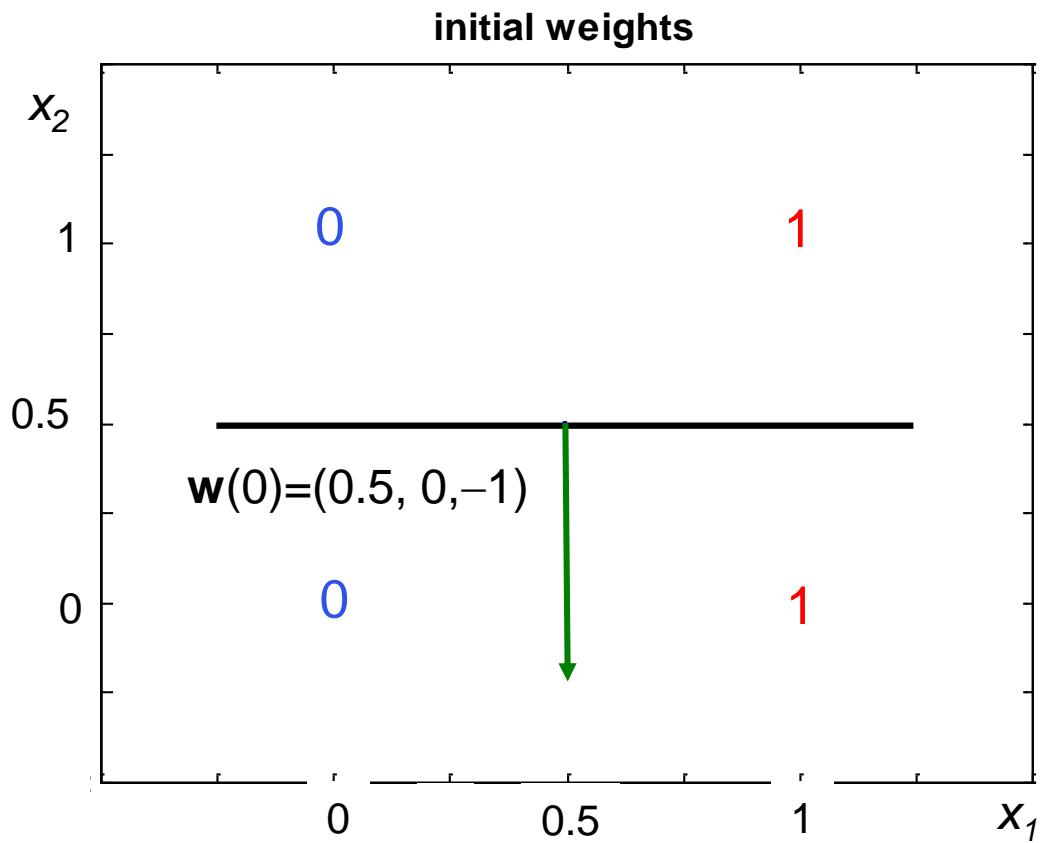
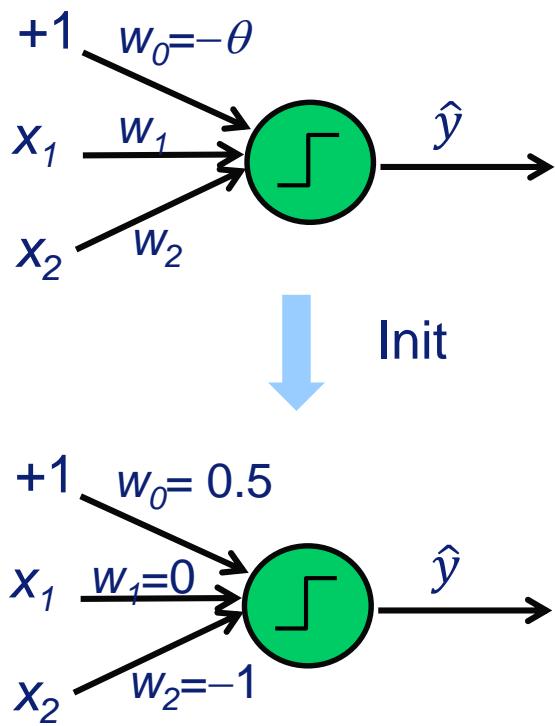
Perceptron output:

$$\begin{aligned}\hat{y} &= \Theta[\mathbf{w} \cdot \mathbf{x}] = \Theta\left[\sum_{j=0}^n w_j \cdot x_j\right] \\ &= \Theta[w_0 + w_1 \cdot x_1 + w_2 \cdot x_2]\end{aligned}$$



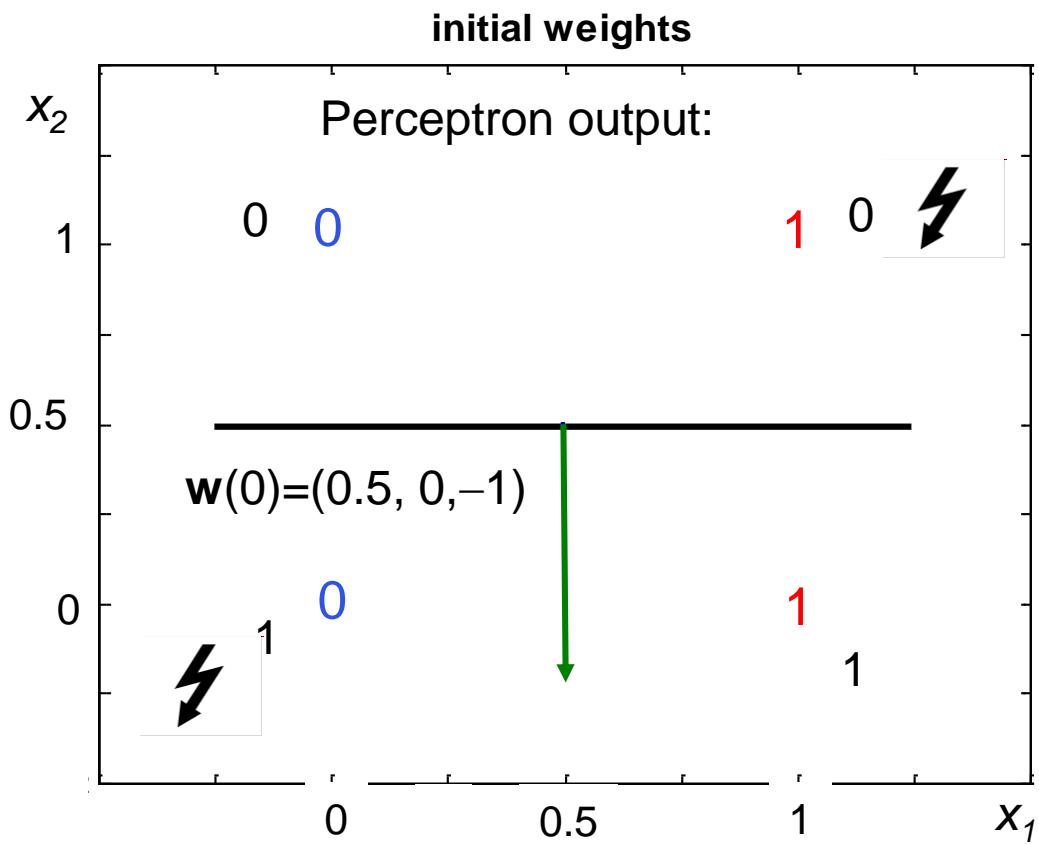
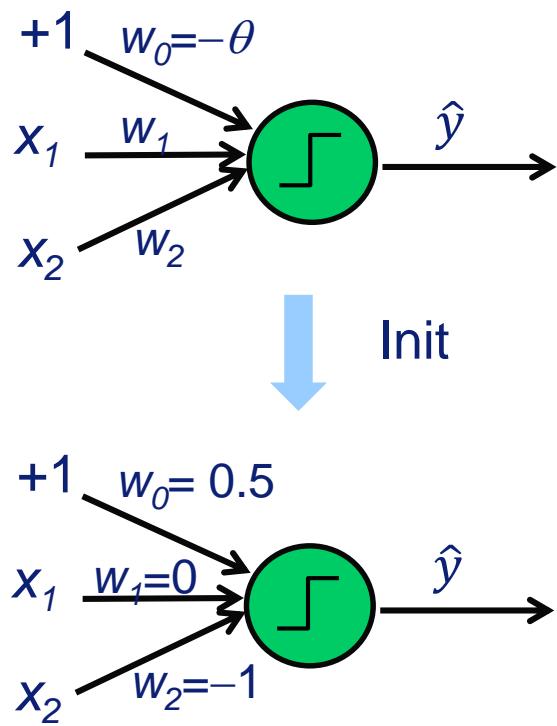
Perceptron learning: Example

- Perceptron with two inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Initialisation: $w_1 = 0, w_2 = -1$, threshold $\theta = -0.5$



Perceptron learning: Example

- Perceptron with two inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Initialisation: $w_1 = 0, w_2 = -1$, threshold $\theta = -0.5$



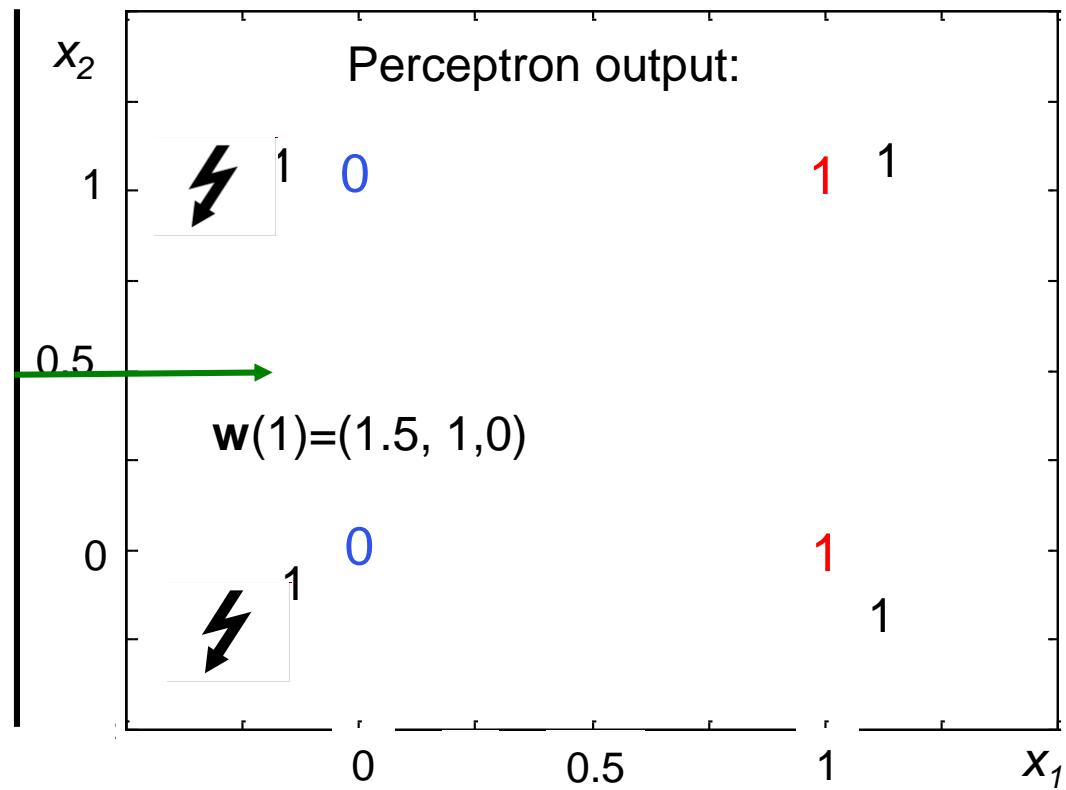
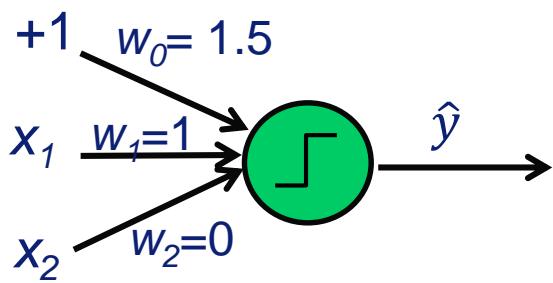
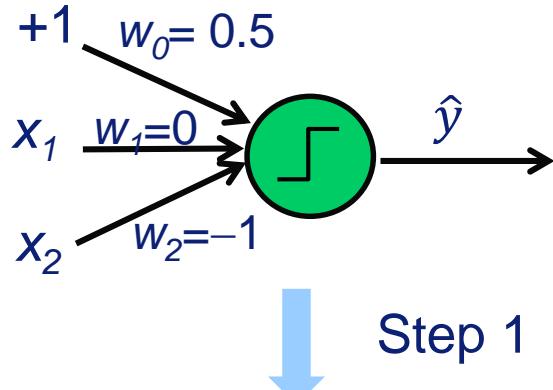
Perceptron learning: Example

Step 1: Select input $(x_1, x_2) = (1, 1)$ which is wrongly classified; set $\eta = 1$

- output $\hat{y} = 0$, target output $y = 1 \Rightarrow$ deviation $\varepsilon = 1$

Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0 \\ -1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1 \\ 0 \end{pmatrix}$$



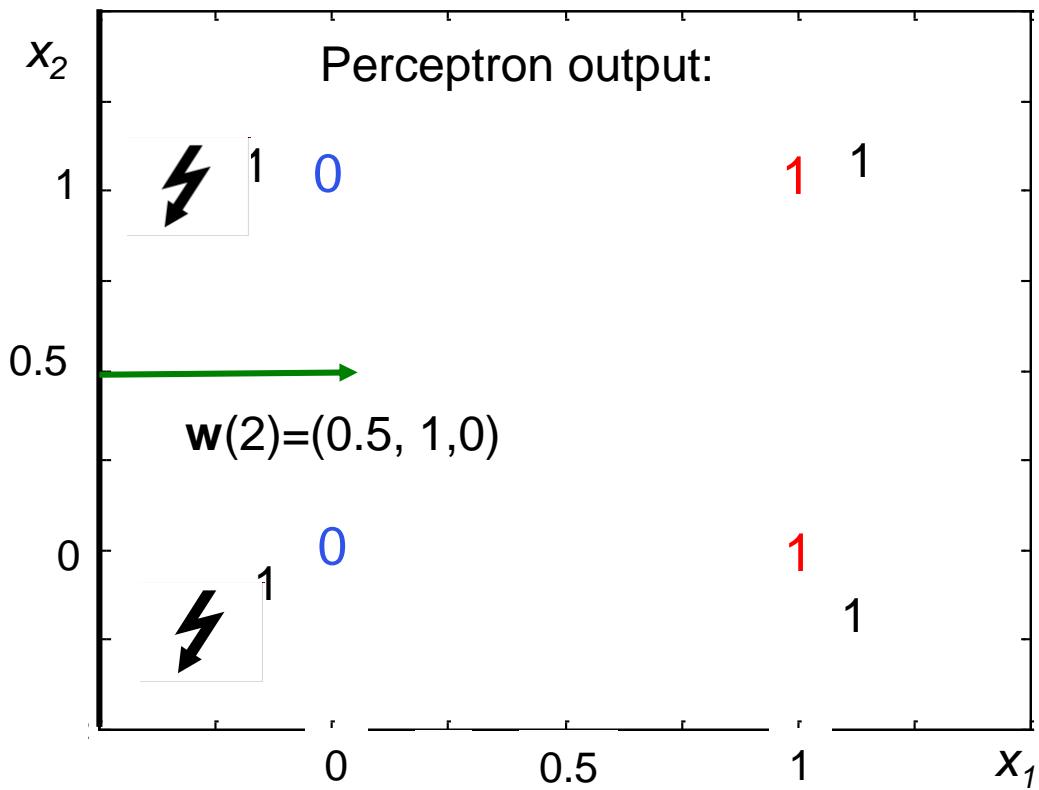
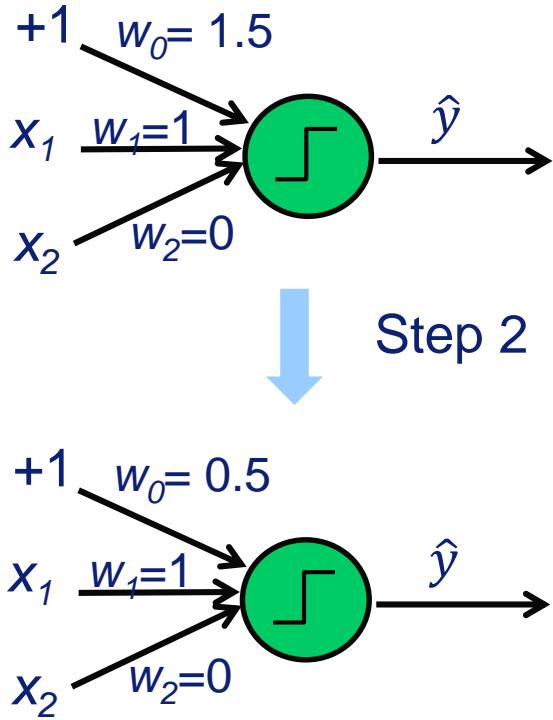
Perceptron learning: Example

Step 2: Select input $(x_1, x_2) = (0,0)$ which is wrongly classified; set $\eta = 1$

- output $\hat{y} = 1$, target output $y = 0 \Rightarrow$ deviation $\varepsilon = -1$

Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} - \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 1 \\ 0 \end{pmatrix}$$



Perceptron learning: Example

Step 3: Select input $(x_1, x_2) = (0,0)$ which is still wrongly classified; set $\eta = 1$

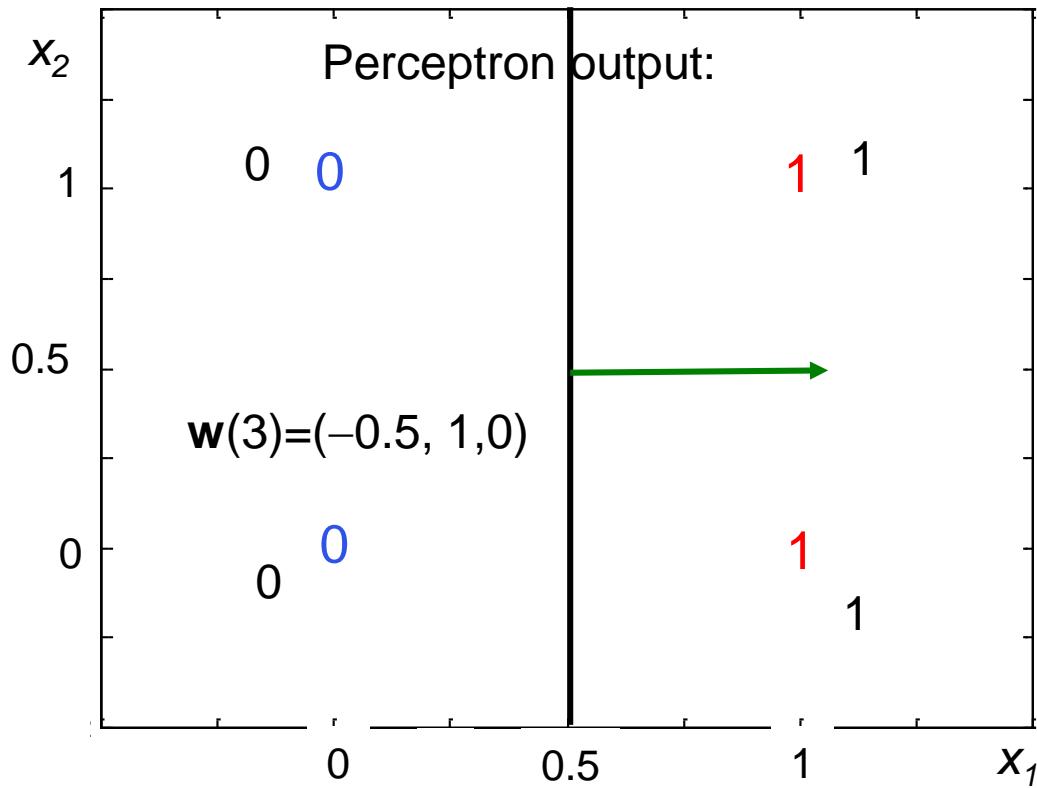
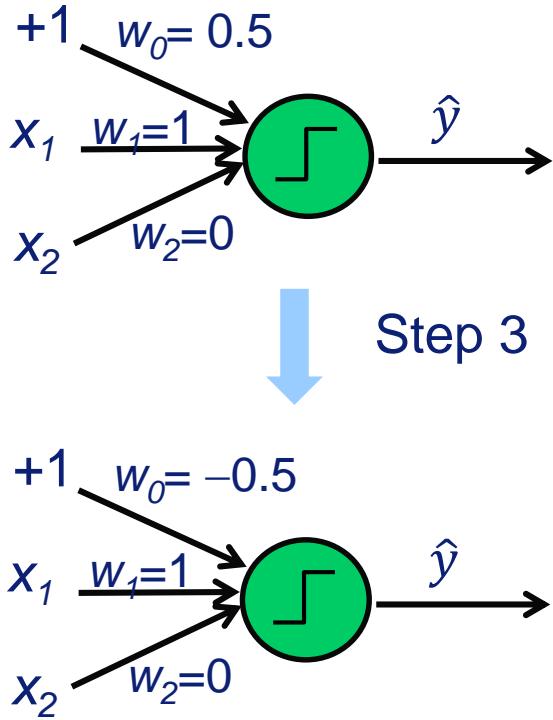
- Still output $\hat{y} = 1$, target output $y = 0 \Rightarrow$ deviation $\varepsilon = -1$

Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} - \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 1 \\ 0 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -0.5 \\ 1 \\ 0 \end{pmatrix}$$



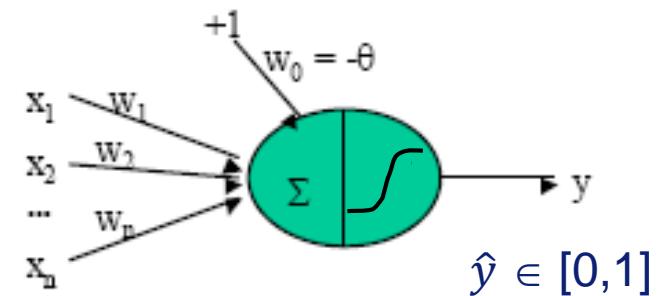
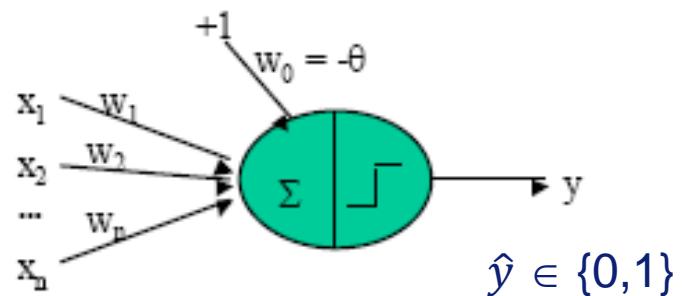
All 4 patterns correct!



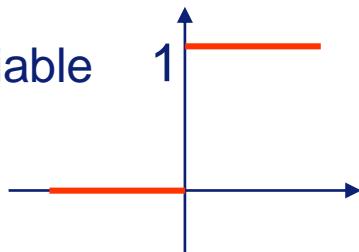
Learning based on gradient descent

From binary to real-valued outputs

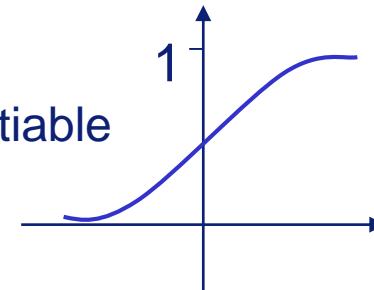
- Release assumption of *binary* activation function, i.e. $y \in \{0,1\}$
- Instead: **differentiable** (monotonically increasing) activation fct. $f(h)$
 - E.g. linear or sigmoid
- Also target value can be real: $y \in [0,1]$ (sigmoid) or $y \in \mathbb{R}$ (linear)
- Perceptron output now given by $\hat{y} = f(\mathbf{w} \cdot \mathbf{x})$



Not differentiable
(at $h=0$)



differentiable



- General (e.g. sigmoid) f needs extension of the perceptron learning rule

Training objective: Minimizing loss function

- Idea: Estimate parameters in **iterative fashion** by minimizing energy / loss
 - (since for non-linear perceptrons, no closed-form solution available!)
- Ultimate goal: **Minimize generalization error**
 - But: generalization error unknown since data generating distribution unknown
 - Besides, generalization error not often differentiable w.r.t. network parameter
- Instead, **minimize „surrogate loss function“**
 - Calculated on finite data sample (training data): „empirical risk minimization“
 - Differentiable with regard to network parameters
 - Related to task and to generalization error
- Assumption: Minimizing surrogate loss minimizes generalization error
 - Not always true → risk of overfitting, adequate **stopping criterion** needed!
- Common loss functions:
 - **Mean squared error loss** (e.g. for regression problems)
 - **Cross-entropy loss** (e.g. for logistic functions)
 - **Log-likelihood loss** (e.g. for softmax output)

Mean squared error (MSE) loss

- Define loss function, e.g. (half) mean squared error (MSE) loss

$$L_{MSE}(\mathbf{w}, y, \hat{y}) = \frac{1}{2p} \sum_{\mu=1}^p L_{MSE}^{(\mu)}(\mathbf{w}, y, \hat{y}) = \frac{1}{2p} \sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)})^2$$

- Loss function quantifies deviation between perceptron output $\hat{y}^{(\mu)}$ and target $y^{(\mu)}$ over training set D with p input patterns
 - Output $\hat{y}^{(\mu)} = \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})$ for input $x^{(\mu)}$ depends on parameters (weights, bias)
 - Goal: Find parameters $\mathbf{w}=\mathbf{w}^*$ minimizing (MSE) loss:
- $$\mathbf{w}^* = \arg \min_{\mathbf{w}} L(\mathbf{w}, y, \hat{y}) = \arg \min_{\mathbf{w}} \underset{\text{MSE}}{\frac{1}{2p} \sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)})^2}$$
- Other loss functions exist, depending on the task (see later)

Minimizing the loss function with gradient descent

- Idea: Iterative procedure (index t) to minimize L : $\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta\mathbf{w}(t)$
→ How to choose $\Delta\mathbf{w}$?
- Gradient (partial derivative) of L indicates direction of steepest ascent of L
⇒ to minimize L , we should make steps into the direction of *negative gradient*
- Formally: If we change a parameter w_j by Δw_j , the function L changes by

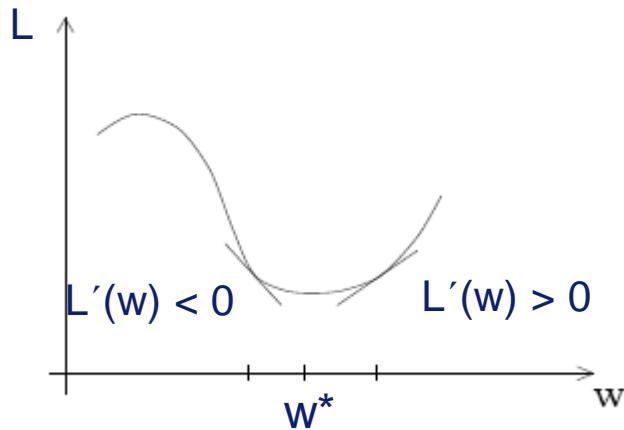
$$\Delta L = \frac{\partial L}{\partial w_j} \Delta w_j \quad \text{so if we choose } \Delta w_j = -\eta \frac{\partial L}{\partial w_j} \quad \text{we get} \quad \Delta L = -\eta \left\| \frac{\partial L}{\partial w_j} \right\|^2 < 0$$

→ parameter update: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w})$

- Learning rate $\eta > 0$ controls step size in iteration process
 - Must be chosen appropriately
- Loss function (and thus activation function) must be differentiable!
 - With regard to the perceptron parameters (weights, biases)

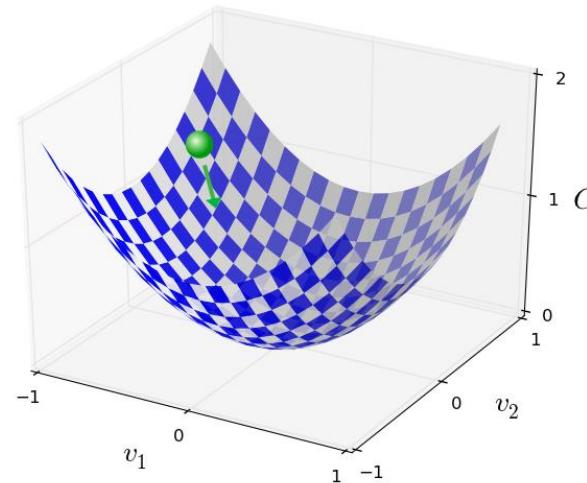
Minimizing the loss function with gradient descent: Illustration

- Parameter update: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w})$
- Example (1-dim.):



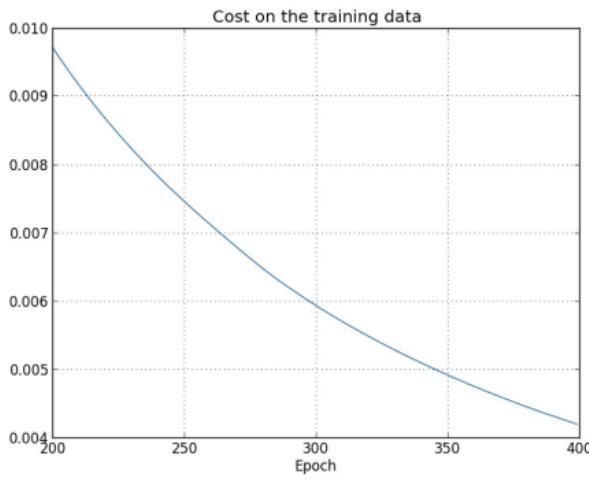
If $L'(w) > 0$: reduce w
 If $L'(w) < 0$: increase w
 If $L'(w) = 0$: no change (*local* minimum w^* found)

- Analogously in more dimensions:
- Finds only *local* minimum!

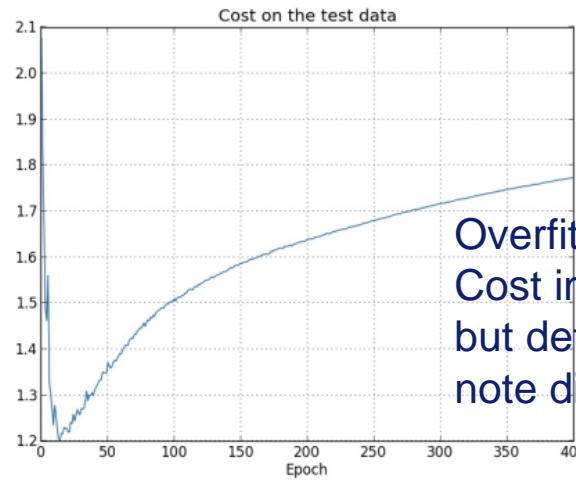


Regularization

- Problem: Too many network parameters may lead to overfitting
 - If not sufficient training data available
- Many different techniques / strategies:
 - Weight regularization by adding weight penalty (L1, L2 regularization)
 - Parameter sharing, sparsity
 - Dropout (\rightarrow later)
 - Data augmentation (\rightarrow later)
 - Early stopping: Use validation set to diagnose overfitting / underfitting



Epoch 200 400



Epoch 0 400

Weight regularization

- Impose additional constraint (penalty) on the weights
 - Make weights in similar range (no neurons dominate) / make small weights 0
 - Large weights relate to more complex models: small input change \rightarrow large effect
 - Biases not penalized (less a cause for overfitting; sometimes large bias desired)

→ Add weight regularization term to loss function:

→ „cost function“ $J(\mathbf{w}, y, \hat{y}) = L(\mathbf{w}, y, \hat{y}) + \lambda \cdot \Omega(\mathbf{w})$

→ additional regularization parameter λ , estimated via (cross-)validation

→ gradient descent on cost function leads to additional regularization term:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) - \eta \cdot \lambda \cdot \nabla_{\mathbf{w}} \Omega(\mathbf{w})$$

$$b(t+1) = b(t) - \eta \cdot \nabla_b L(b) - \eta \cdot \lambda \cdot \nabla_b \Omega(b)$$

$$b(t+1) = b(t) - \eta \cdot \nabla_b L(b)$$

(if regularization includes bias)

(without bias regularization)



L1 / L2 regularization

L1 regularization

$$\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_i |w_i|$$

- Weight update:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) - \eta \cdot \lambda \cdot \text{sgn}(\mathbf{w})$$

- Drives smaller weights more likely towards zero
- Thus favors sparse weights

- Choice of regularization depends on task

- In the following, regularization may be applied
 - Then, replace the loss function L by the cost function J (incl. regularization term)

L2 regularization

$$\Omega(\mathbf{w}) = \frac{1}{2} \|\mathbf{w}\|_2^2 = \frac{1}{2} \sum_i w_i^2$$

„weight decay“

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w}) - \eta \cdot \lambda \cdot \mathbf{w}$$

- Strongly penalizes large weights (proportional to w)
- Small weights penalized less than for L1 regularization

Single-layer perceptron: Training setup

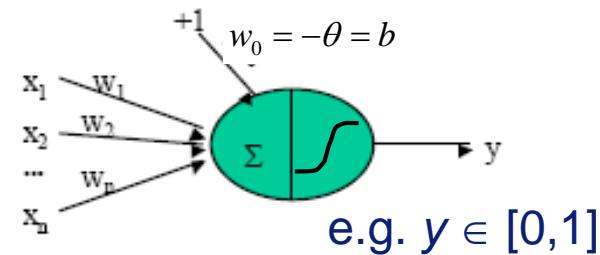
- Training set (input $\mathbf{x}^{(\mu)}$, true label $y^{(\mu)}$):

$$D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(p)}, y^{(p)})\}$$

- Postsynaptic potential $z^{(\mu)}$ for input $\mathbf{x}^{(\mu)}$:

$$z^{(\mu)} = \sum_{k=1}^n w_k \cdot x_k^{(\mu)} - \theta = \sum_{k=1}^n w_k \cdot x_k^{(\mu)} + b = \mathbf{w} \cdot \mathbf{x}^{(\mu)} + b$$

- Perceptron output for input $\mathbf{x}^{(\mu)}$: $\hat{y}^{(\mu)} = f(\mathbf{w} \cdot \mathbf{x}^{(\mu)} + b)$
 - With differentiable activation function f
- Goal: Find perceptron parameters $\Theta = \{\mathbf{w}, b\}$ by supervised learning on the training set using gradient descent with the loss function L
- Define auxiliary quantity: „Error term“ $\Delta := \frac{\partial L}{\partial z}$
 - Quantifies impact on loss function L if postsynaptic potential z is modified (e.g. by changing a synaptic weight or bias)



Single-layer perceptron: Derivation of „least mean squares learning“

Strategy:

- Calculate Δ using chain rule w.r.t. perceptron output \hat{y}
- Then calculate partial derivatives of L w.r.t. w and b using chain rule

$$1.) \quad \Delta = \frac{\partial L}{\partial z} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\partial L}{\partial \hat{y}} \cdot f'(z) \quad \text{since } \hat{y} = f(z) \Rightarrow \frac{\partial \hat{y}}{\partial z} = f'(z)$$

$$2.) \quad \frac{\partial L}{\partial w_k} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w_k} = \Delta \cdot x_k^{(\mu)} \quad \text{since } \Delta = \frac{\partial L}{\partial z} \quad \text{and} \quad z^{(\mu)} = \sum_{k=0}^n w_k \cdot x_k^{(\mu)} + b \Rightarrow \frac{\partial z}{\partial w_k} = x_k^{(\mu)}$$

$$3.) \quad \frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b} = \Delta \quad \text{since } \Delta = \frac{\partial L}{\partial z} \quad \text{and} \quad z^{(\mu)} = \sum_{k=0}^n w_k \cdot x_k^{(\mu)} + b \Rightarrow \frac{\partial z}{\partial b} = 1$$

- Using MSE loss: $L_{MSE}(w, y, \hat{y}) = \frac{1}{2p} \sum_{\mu=1}^p (\hat{y}(w, x^{(\mu)}) - y^{(\mu)})^2 \Rightarrow \frac{\partial L_{MSE}}{\partial \hat{y}} = \frac{1}{p} \sum_{\mu=1}^p (\hat{y}(w, x^{(\mu)}) - y^{(\mu)})$

$$\frac{\partial L_{MSE}}{\partial w_k} = \Delta \cdot x_k^{(\mu)} = \frac{\partial L_{MSE}}{\partial \hat{y}} \cdot f'(z^{(\mu)}) \cdot x_k^{(\mu)} = \frac{1}{p} \sum_{\mu=1}^p (\hat{y}^{(\mu)} - y^{(\mu)}) \cdot f'(z^{(\mu)}) \cdot x_k^{(\mu)}$$

$$\nabla_w L_{MSE} = \frac{1}{p} \sum_{\mu=1}^p (\hat{y}^{(\mu)} - y^{(\mu)}) \cdot f'(z^{(\mu)}) \cdot x^{(\mu)}$$

... and similarly for bias b

Single-layer perceptron: „Gradient learning“ (without regularization)

- **Batch learning:** $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L_{MSE}(\mathbf{w}) = \mathbf{w}(t) + \eta \cdot \frac{1}{p} \sum_{\mu=1}^p (y^{(\mu)} - \hat{y}^{(\mu)}) \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$
- **Online learning:** $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})) \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$
 $b(t+1) = b(t) + \eta \cdot (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})) \cdot f'(z^{(\mu)})$
 - After a *single* training example $\mathbf{x}^{(\mu)}$
 - But: does not necessarily minimize the *total* training error!
 - Order of patterns might influence results (\rightarrow random order after each epoch)

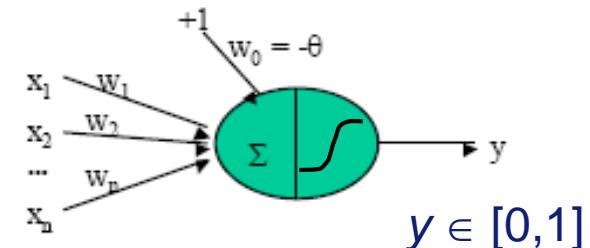
Weights are modified

- proportional to the **deviation between target and actual perceptron output**
- proportional to the **derivative of the activation function** at the input PSP:
 - Sigmoid activation function: large contribution if the PSP is near threshold
 - Sigmoid activation function: small contribution if the PSP is far from threshold
- proportional to the **learning rate η**

„Gradient learning“ versus perceptron learning

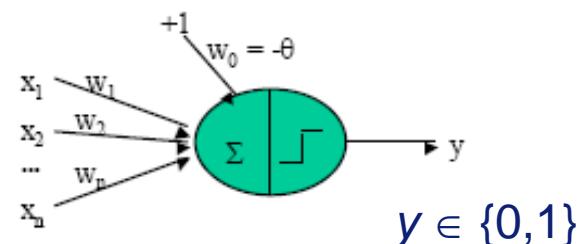
- „Gradient learning“ (online mode, training pattern $\mathbf{x}^{(\mu)}$):
 - Single-layer perceptron with **differentiable** activation function $f(h) \in [0,1]$ (e.g. linear, sigmoid)

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{\left(y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) \right)}_{\text{Difference: } \in [-1, 1] \text{ (sigmoid)} \\ \in \mathbb{R} \text{ (linear)}} \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$$



- Perceptron learning (online mode, training pattern $\mathbf{x}^{(\mu)}$):
 - Single-layer perceptron with **binary** activation function $f(h) \in \{0,1\}$

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{\left(y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) \right)}_{\text{Difference: } 0 \text{ or } \pm 1} \cdot \mathbf{x}^{(\mu)}$$



„Gradient learning“: Algorithm

Given: Training set $D = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(p)}, y^{(p)})\}$

- Initialise all weights w_j (e.g. small random numbers)
 - For all training patterns $(\mathbf{x}^{(\mu)}, y^{(\mu)})$:
 - Calculate $z^{(\mu)}$ and perceptron output $\hat{y}^{(\mu)}$ for input $\mathbf{x}^{(\mu)}$: $z^{(\mu)} = \sum_{j=0}^n w_j x_j^{(\mu)}$; $\hat{y}^{(\mu)} = f(z^{(\mu)})$
 - Calculate deviation $\varepsilon^{(\mu)}$ between target output $y^{(\mu)}$ and output $\hat{y}^{(\mu)}$: $\varepsilon^{(\mu)} = y^{(\mu)} - \hat{y}^{(\mu)}$
 - Calculate derivative of activation function f at activation value $z^{(\mu)}$: $f'(z^{(\mu)})$
 - for online learning: Modify weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \varepsilon^{(\mu)} \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$
 - for batch learning: Modify weights: $\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \sum_{\mu} \varepsilon^{(\mu)} \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$
 - Repeat training until (total) training error small enough
 - Test perceptron on examples not belonging to training set (generalisation)
 - In practice: Parameter updates based on randomly chosen single training samples („online learning“) / small „mini-batches“
- „Stochastic gradient descent“

Stochastic gradient descent

- Gradient descent where the gradient is computed against a randomly chosen training sample / „small“ mini-batch:
- **Initialization:** learning rate η , initial parameter \mathbf{w}
- **While** stopping criterion not met **do**
 - Sample a mini-batch of m examples from the training set $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(\mu)}$ (in practice: shuffle data \rightarrow then keep that order)
 - Compute gradient estimate based on the per-sample loss on the mini-batch:
$$\hat{g} = \frac{1}{m} \nabla_{\mathbf{w}} \sum_{\mu=1}^m L^{(\mu)}(\mathbf{w}, y, \hat{y}) = \frac{1}{m} \sum_{\mu=1}^m \nabla_{\mathbf{w}} L^{(\mu)}(\mathbf{w}, y, \hat{y})$$
 - Update $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \hat{g}$
- **End while**
- Note: Learning rate η needs to be gradually decreased over time
 - since stochastic gradient estimate introduces noise also in a minimum of L

„Optimal“ batch size for gradient estimation / parameter update?

- Larger batches → more accurate gradient estimation (but less than linear return!)
- But: **Faster convergence** if gradient rapidly approximated on **few samples**
 - instead of slowly computing exact gradient on all samples
 - E.g. if there is redundancy in the training data → not all samples really needed
 - Smaller batch sizes have regularizing effect (add noise to learning process)

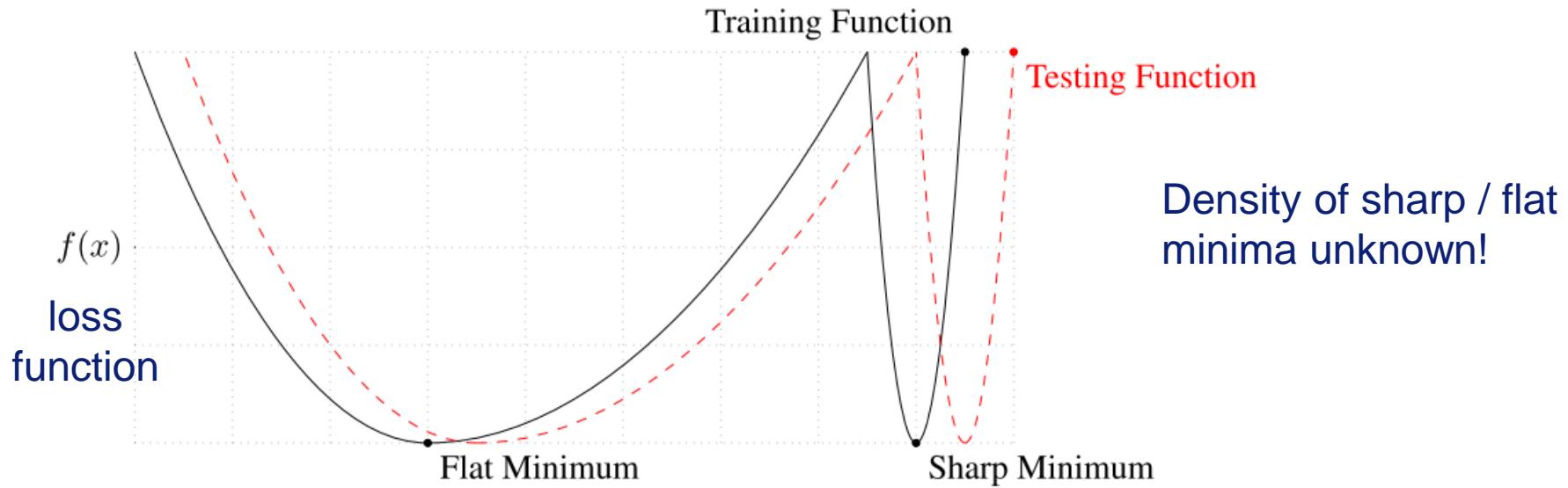
→ Use smaller batch sizes: „mini-batches“

- Examples in a batch must be **independent** (→ data shuffling)
- If batch size too small:
 - No efficient use of parallel processing in multi-core architectures → min. size (e.g. 32)
 - Second order methods need more samples (e.g. 10000) for robustness / accuracy
- If batch size too large:
 - Amount of memory in parallel processing scales with batch size (prohibitive?!)
 - As long as only new and independent samples are used, the estimated gradient follows the unbiased gradient of the generalization error
 - In practice: Often several iterations over complete training data: „**epochs**“
 - To further reduce training error



Mini-batch size and generalization

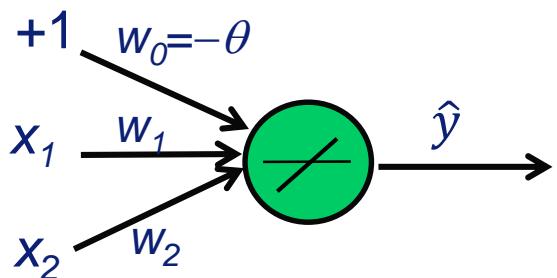
- Sharp minimum: Large number of large positive eigenvalues in Hessian
- Flat minimum: Numerous small eigenvalues in Hessian of loss function
- Sharp minima generalize less well than flat minima!



- Large-batch methods tend to converge to sharp minima of the training fct.
 - Can't escape basins of attraction of sharp minima → initial values important!
- Small-batch methods converge to flat minima of the training function
- Wanted: Large-batch method (to parallelize training) leading to flat minima

Gradient learning: Example

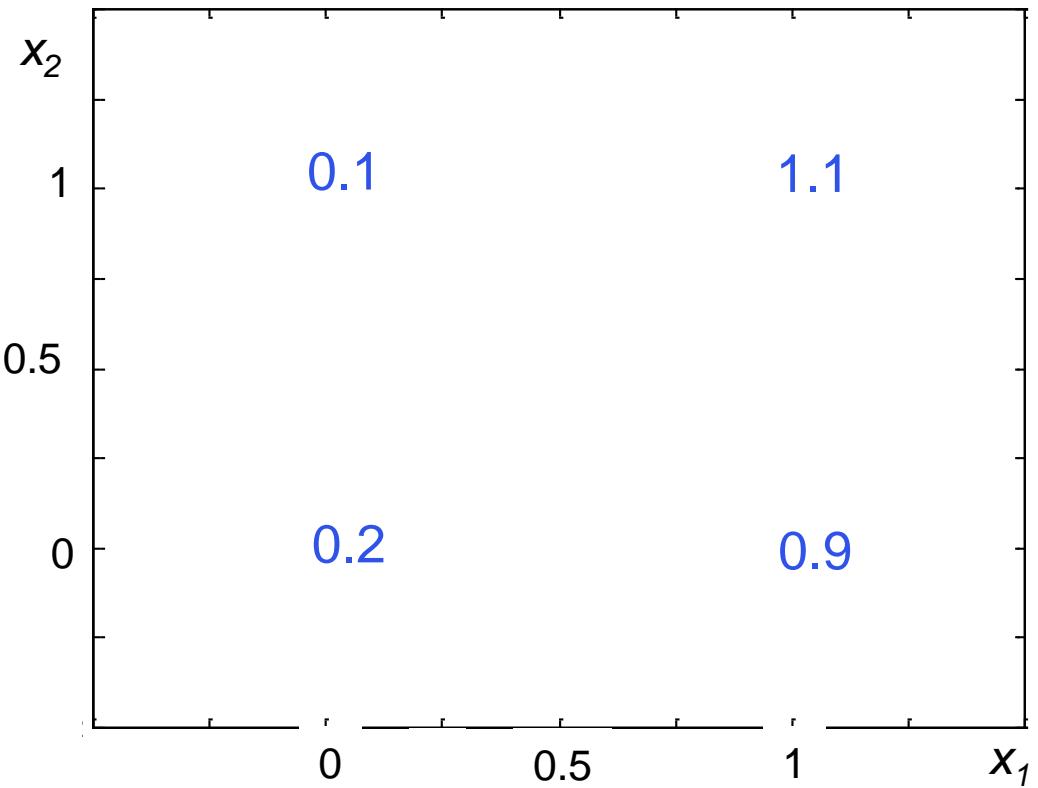
- Linear perceptron with inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Blue: target perceptron output (real number, not a two-class-problem)



Perceptron output:

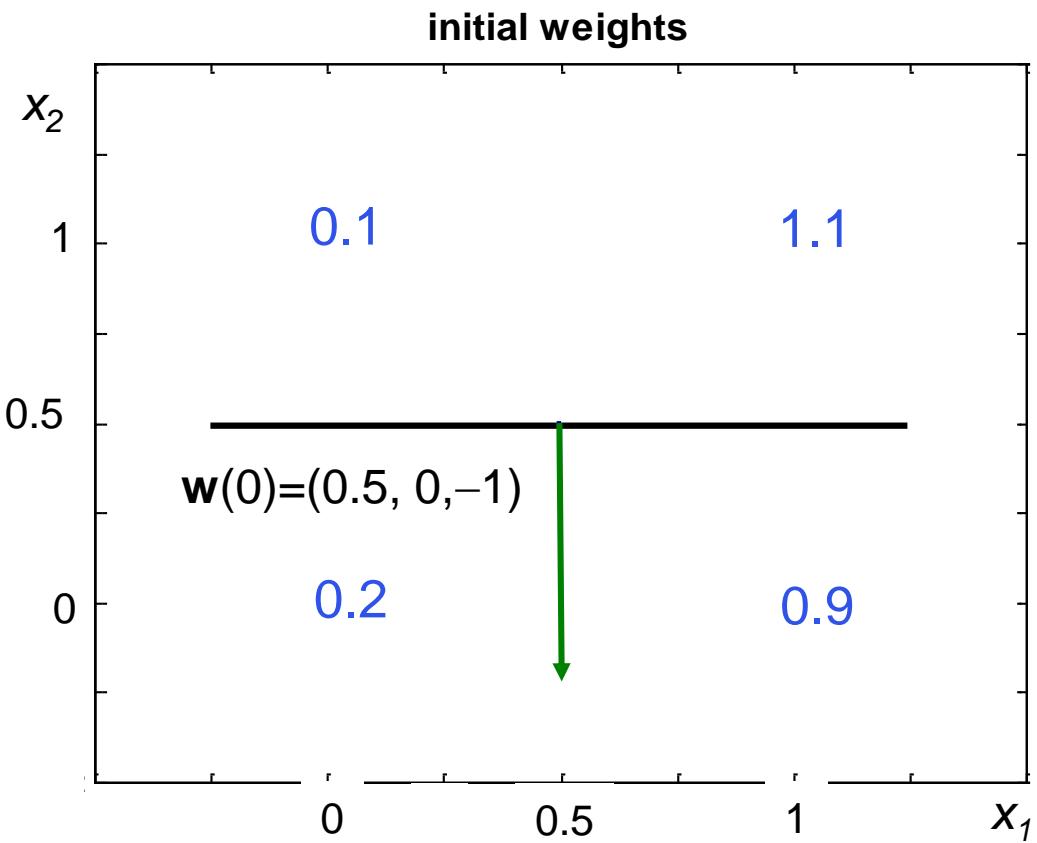
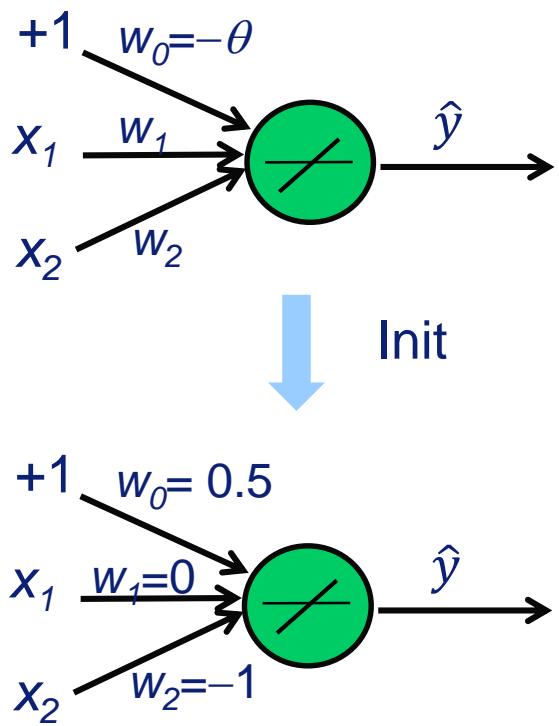
$$\begin{aligned} y &= \mathbf{w} \cdot \mathbf{x} = \sum_{j=0}^n w_j \cdot x_j \\ &= w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 \end{aligned}$$

Derivative: $f(h) = h \Rightarrow f'(h) = 1$



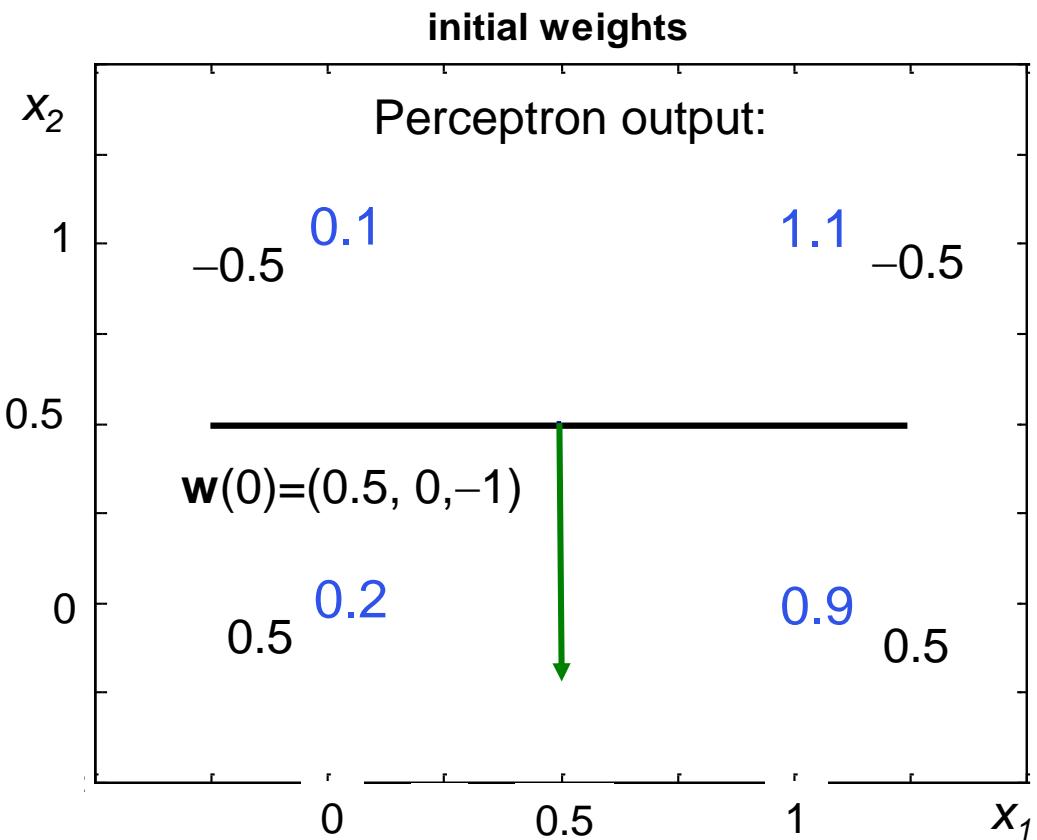
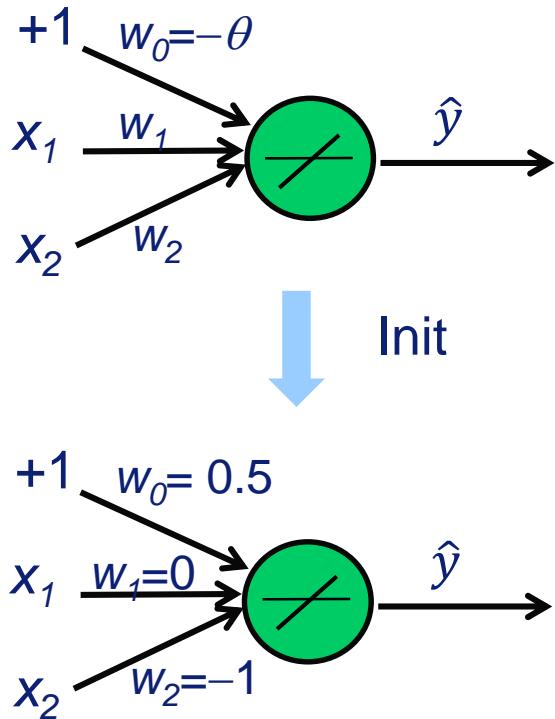
Gradient learning: Example

- Linear perceptron with inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Initialisation: $w_1 = 0, w_2 = -1$, threshold $\theta = -0.5$



Gradient learning: Example

- Linear perceptron with inputs x_1, x_2 , weights w_1, w_2 , threshold $\theta = -w_0$
- Initialisation: $w_1 = 0, w_2 = -1$, threshold $\theta = -0.5$



Gradient learning: Example (online learning)

Step 1: Online learning: Select input $(x_1, x_2) = (1, 1)$; set $\eta = 1$

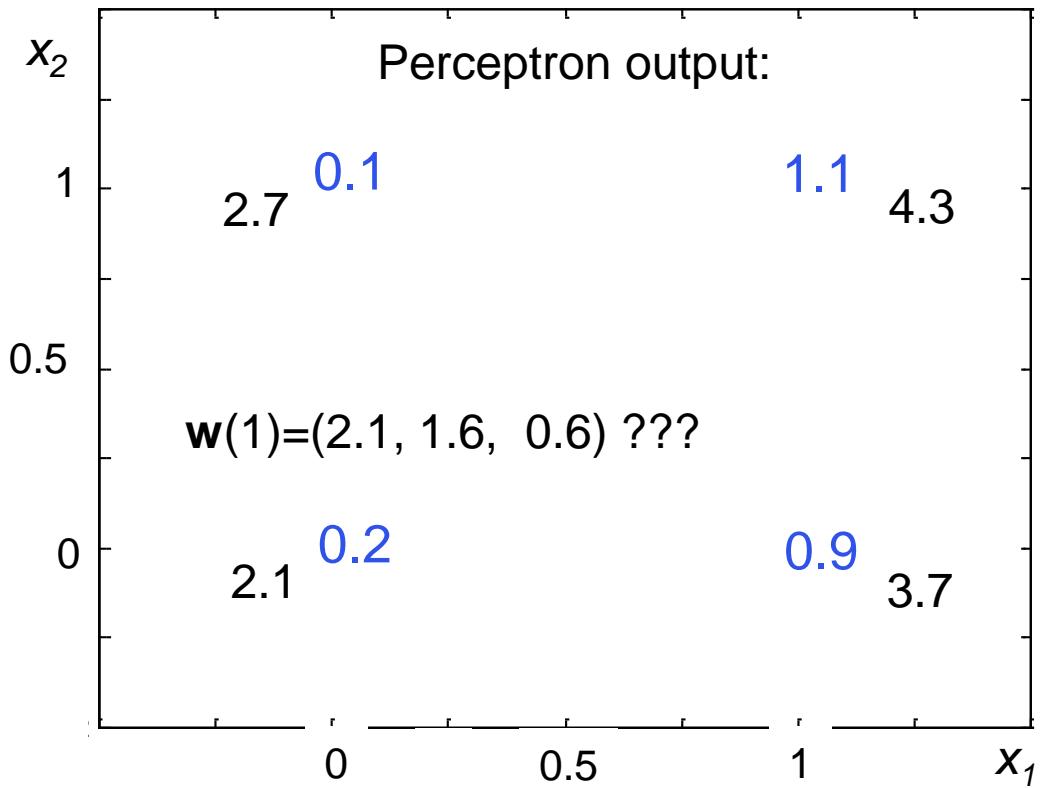
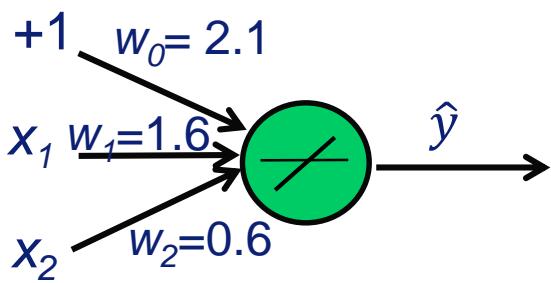
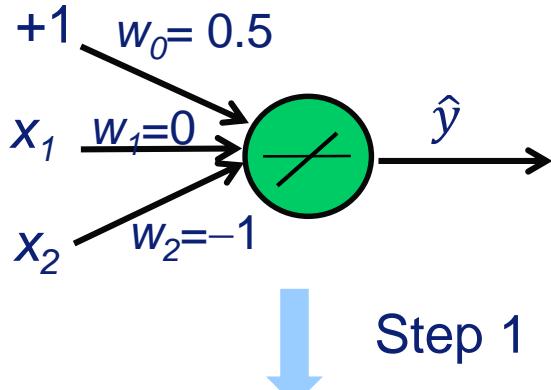
- output $\hat{y} = -0.5$, target output $y = 1.1 \Rightarrow$ deviation $\varepsilon = 1.6$

Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + 1.6 \cdot \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0 \\ -1 \end{pmatrix} + \begin{pmatrix} 1.6 \\ 1.6 \\ 1.6 \end{pmatrix} = \begin{pmatrix} 2.1 \\ 1.6 \\ 0.6 \end{pmatrix}$$

?

- Outputs too large
- Weights too large
- η too large



Gradient learning: Example (online learning)

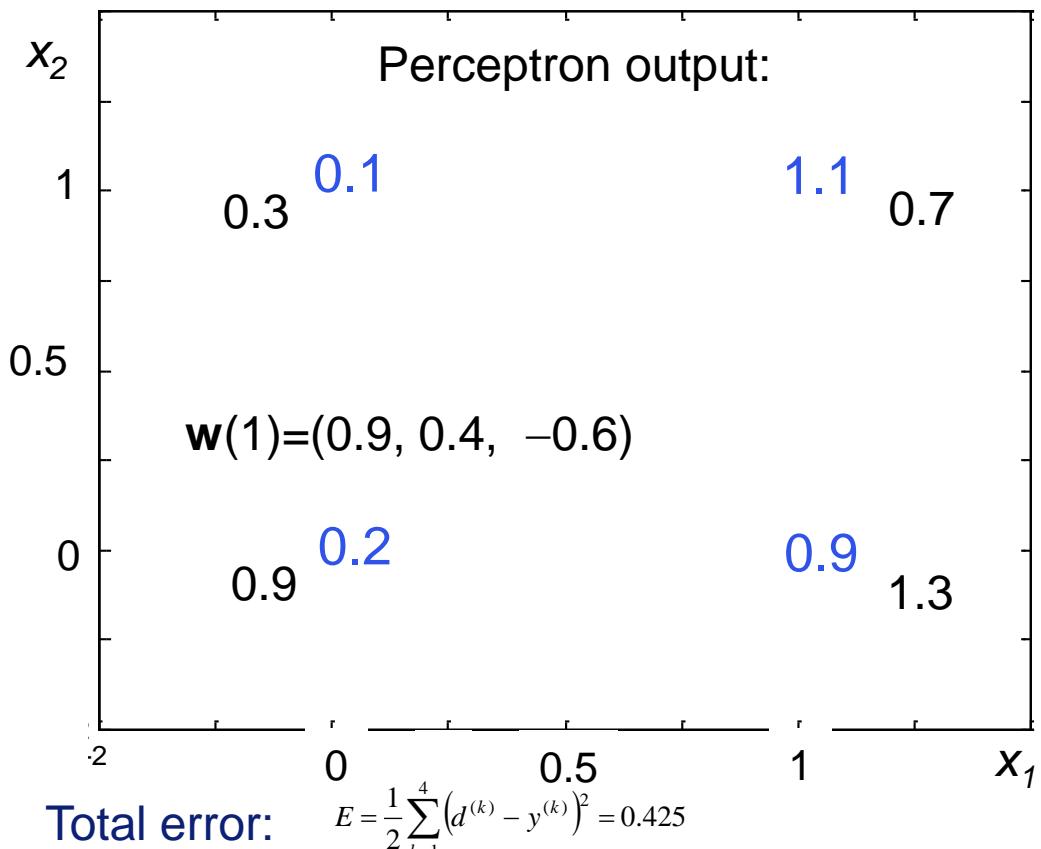
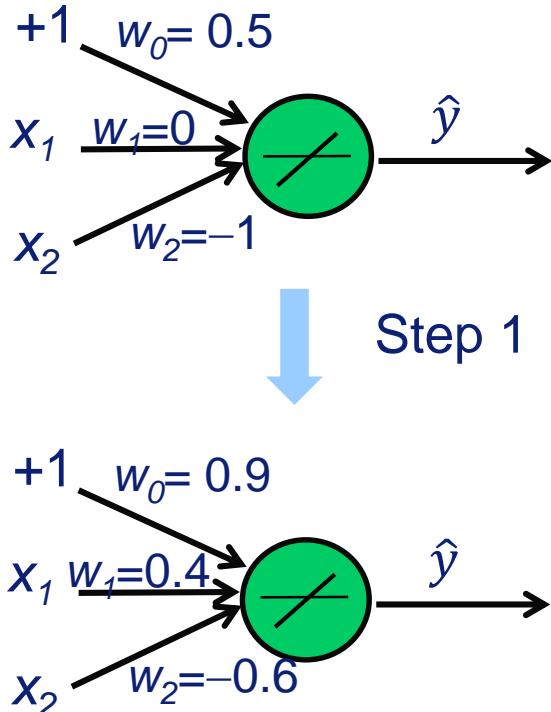
Step 1: Online learning: Learning rate too large; repeat step with $\eta = 0.25$

- output $\hat{y} = -0.5$, target output $y = 1.1 \Rightarrow$ deviation $\varepsilon = 1.6$

Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + 1.6 \cdot \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0 \\ -1 \end{pmatrix} + \begin{pmatrix} 0.4 \\ 0.4 \\ 0.4 \end{pmatrix} = \begin{pmatrix} 0.9 \\ 0.4 \\ -0.6 \end{pmatrix}$$

Recall: $f'(z) = 1$
since *linear* perceptron!



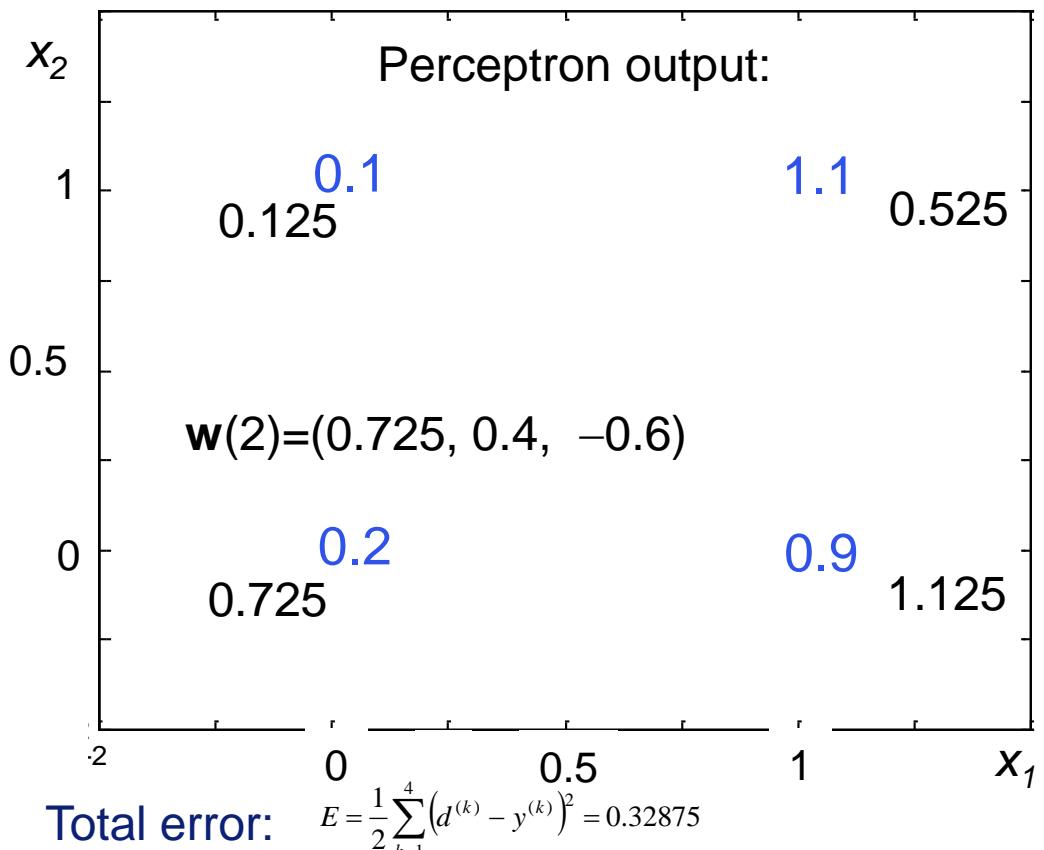
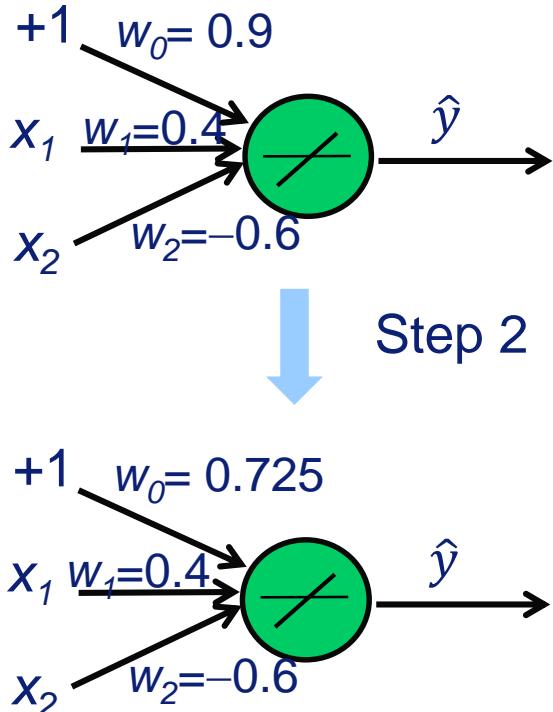
Gradient learning: Example (online learning)

Step 2: Online learning: Now select input $(x_1, x_2) = (0,0)$; set $\eta = 0.25$

- output $\hat{y} = 0.9$, target output $y = 0.2 \Rightarrow$ deviation $\varepsilon = -0.7$

Update:

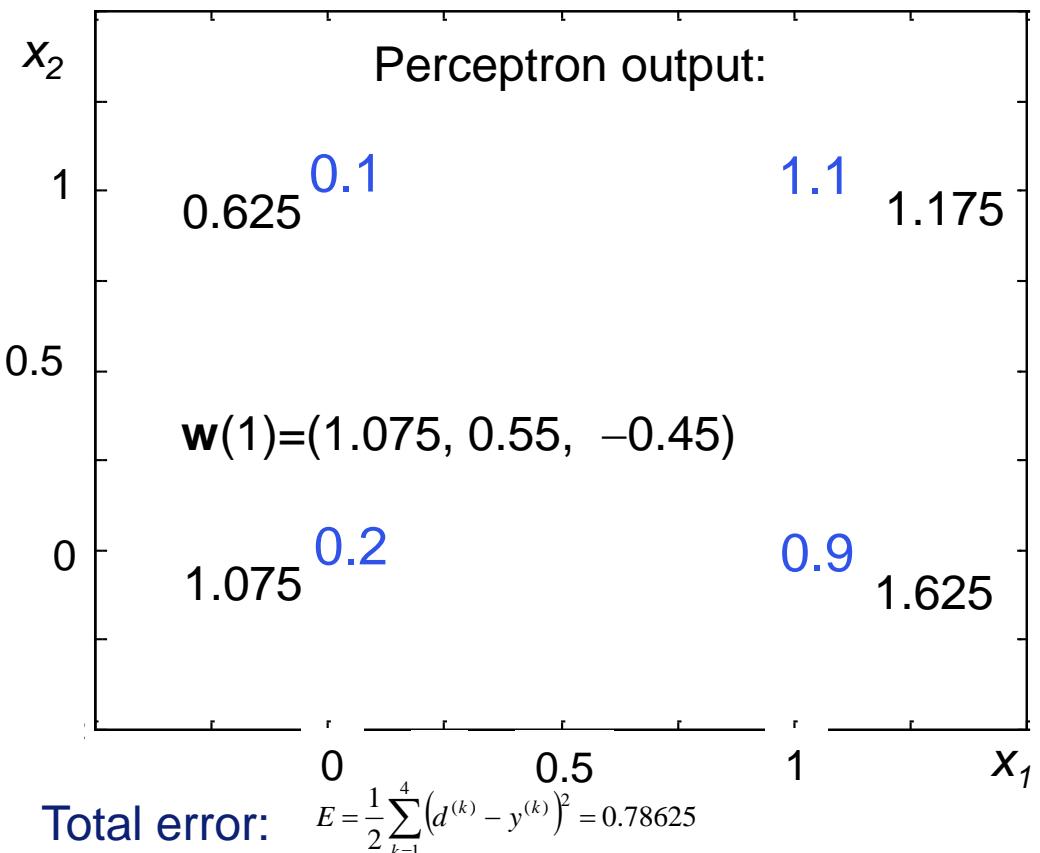
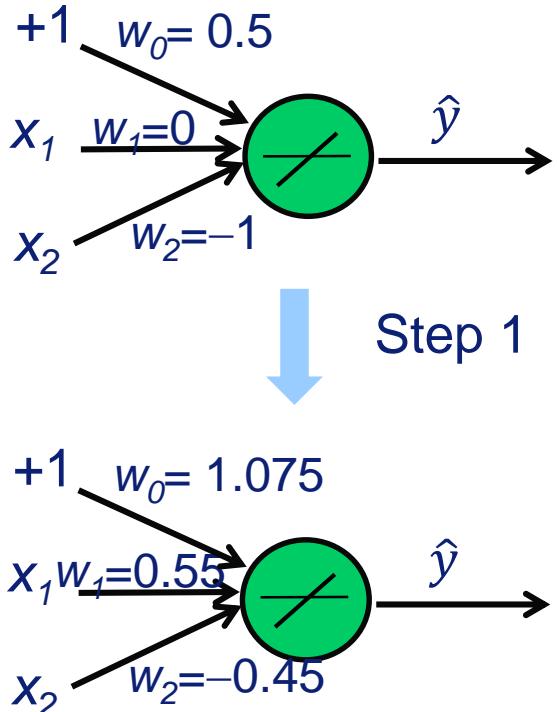
$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} - 0.7 \cdot \eta \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.9 \\ 0.4 \\ -0.6 \end{pmatrix} - 0.175 \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0.725 \\ 0.4 \\ -0.6 \end{pmatrix}$$



Gradient learning: Example (batch learning)

Alternative: Batch learning using *all* patterns; Step 1: select $\eta = 0.25$

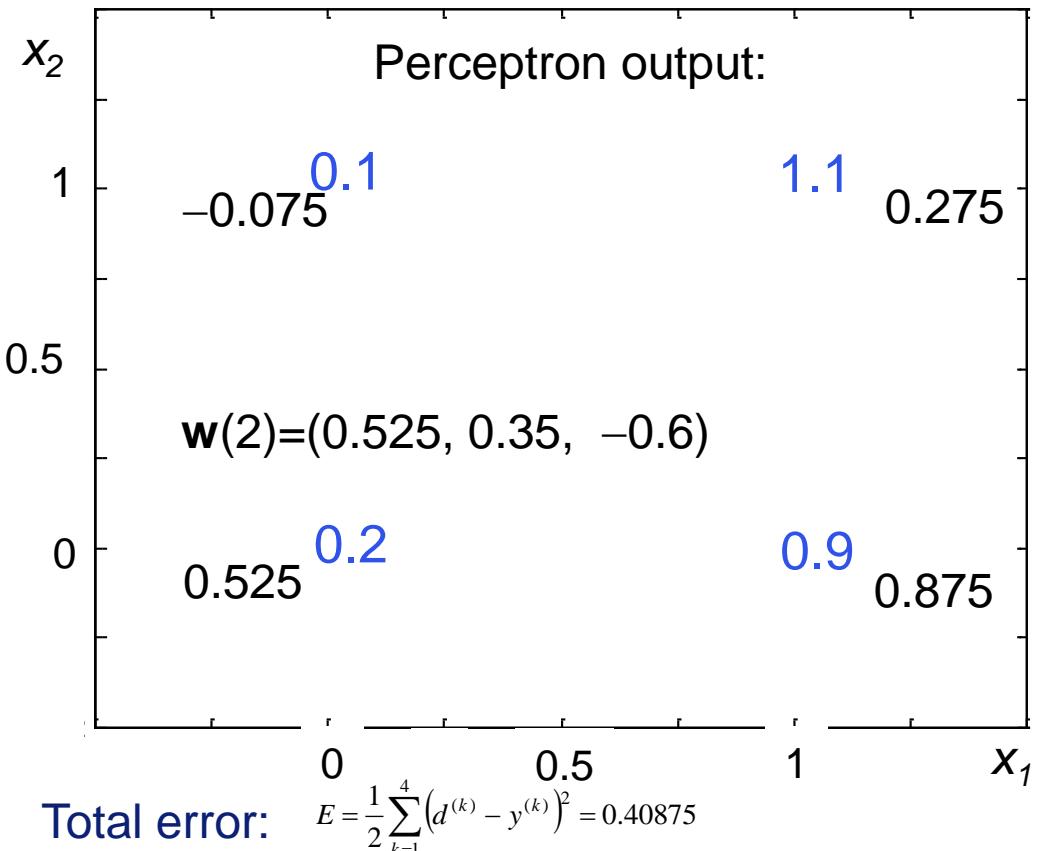
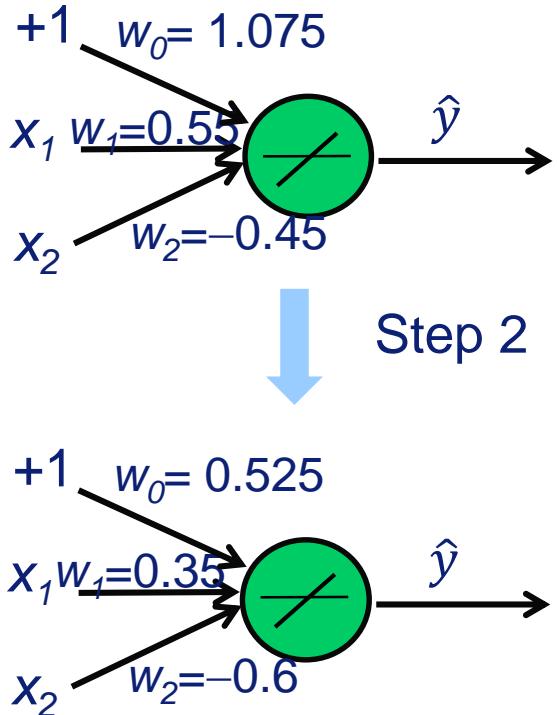
Update:
$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + \eta \cdot \sum_{k=1}^4 (d^{(k)} - y^{(k)}) \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ 0 \\ -1 \end{pmatrix} + 0.25 \cdot \left\{ -0.3 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 0.6 \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + 0.4 \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + 1.6 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\} = \begin{pmatrix} 1.075 \\ 0.55 \\ -0.45 \end{pmatrix}$$



Gradient learning: Example (batch learning)

Step 2: Batch learning: next loop over *all* patterns; select $\eta = 0.25$; Update:

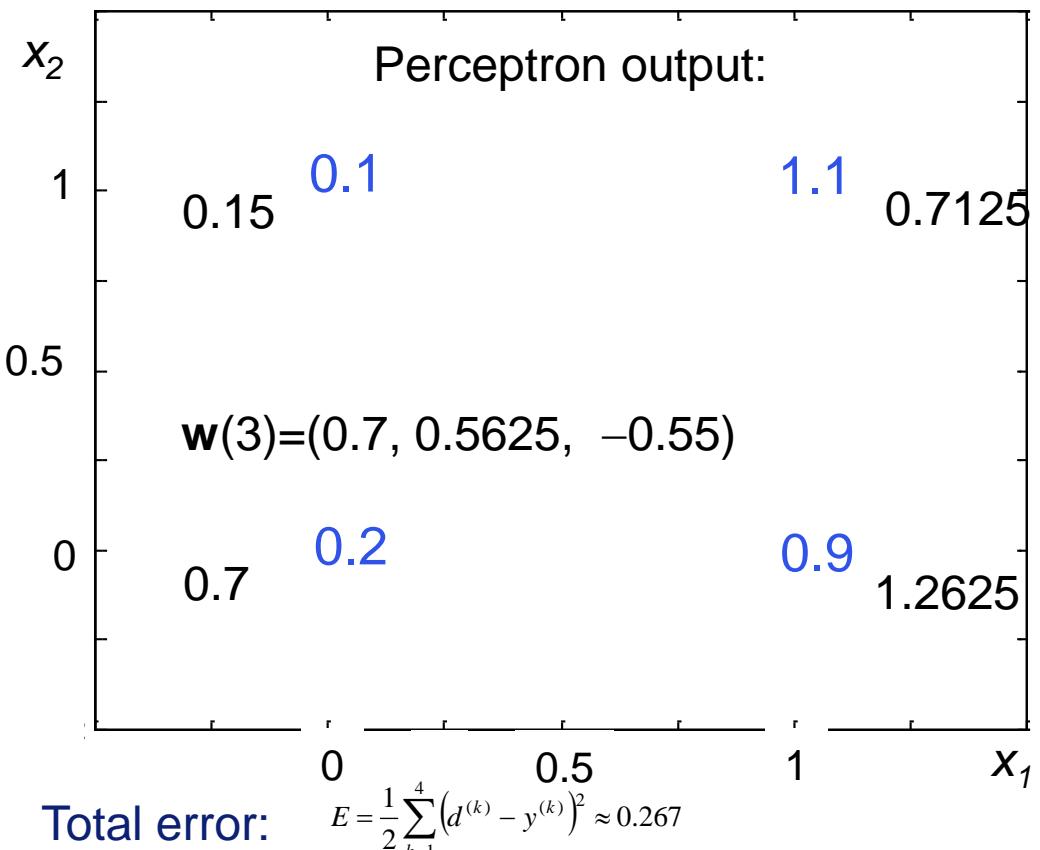
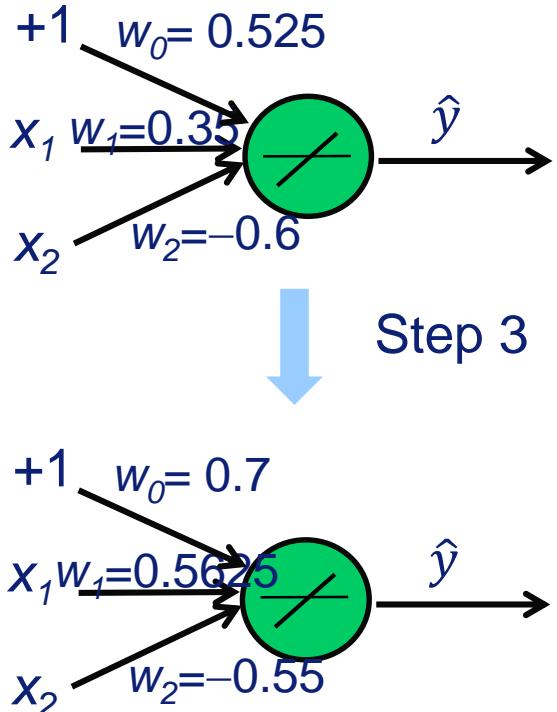
$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + \eta \cdot \sum_{k=1}^4 (d^{(k)} - y^{(k)}) \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1.075 \\ 0.55 \\ -0.45 \end{pmatrix} + 0.25 \cdot \left\{ -0.875 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} - 0.525 \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} - 0.725 \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} - 0.075 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\} = \begin{pmatrix} 0.525 \\ 0.35 \\ -0.6 \end{pmatrix}$$



Gradient learning: Example (batch learning)

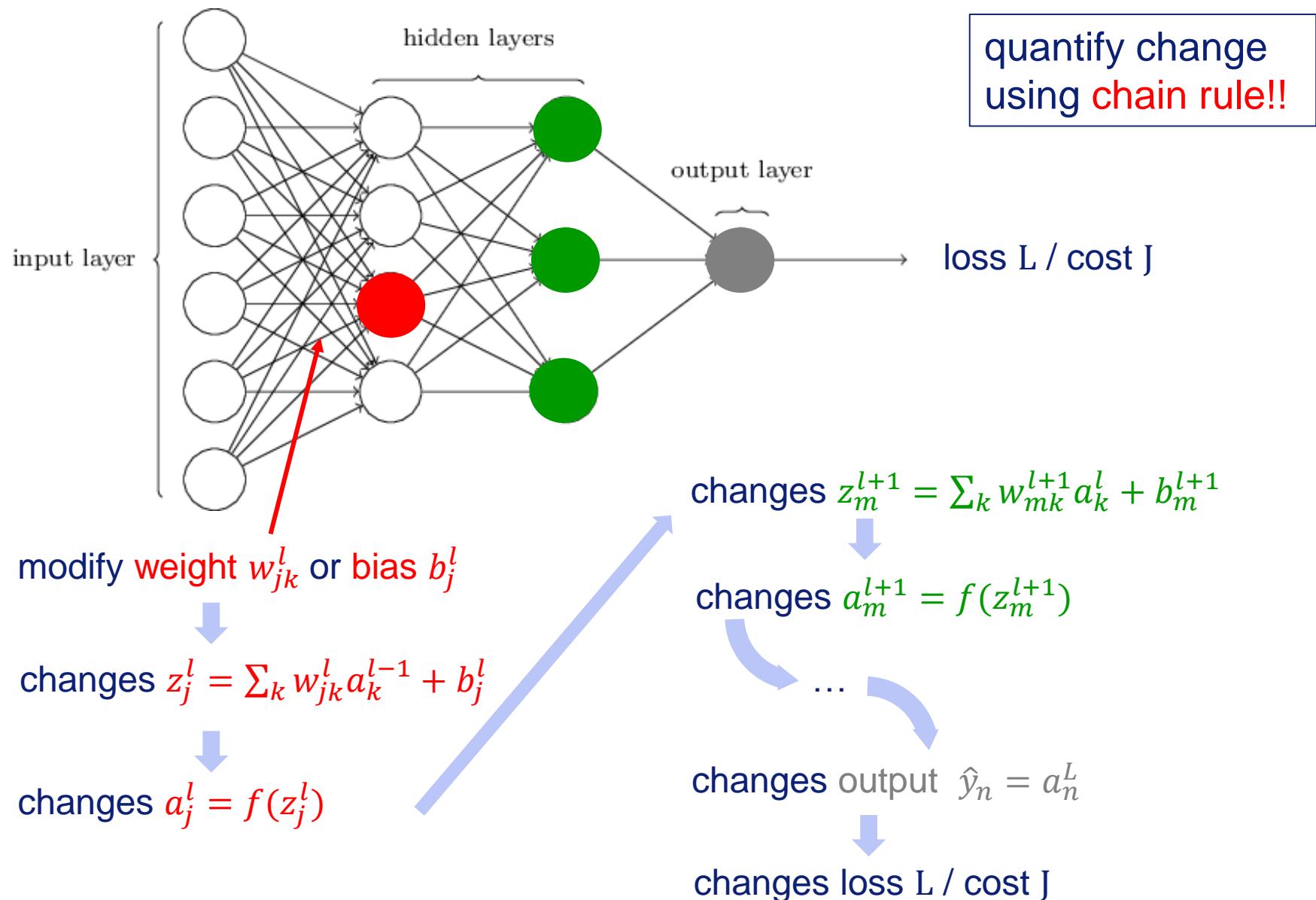
Step 3: Batch learning: next loop over *all* patterns; select $\eta = 0.25$; Update:

$$\begin{pmatrix} w_0(t+1) \\ w_1(t+1) \\ w_2(t+1) \end{pmatrix} = \begin{pmatrix} w_0(t) \\ w_1(t) \\ w_2(t) \end{pmatrix} + \eta \cdot \sum_{k=1}^4 (d^{(k)} - y^{(k)}) \cdot \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0.525 \\ 0.35 \\ -0.6 \end{pmatrix} + 0.25 \cdot \left\{ -0.325 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + 0.175 \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} + 0.825 \cdot \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix} + 0.025 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \right\} = \begin{pmatrix} 0.7 \\ 0.5625 \\ -0.55 \end{pmatrix}$$



Backpropagation

Multi-layer perceptron: Backpropagation (illustration)



Multi-layer perceptron: Strategy to derive backpropagation

- Definition of an error term for neuron j in layer l : $\Delta_j^l := \frac{\partial L}{\partial z_j^l}$
 - Quantifies impact on loss function L if postsynaptic potential z_j^l modified (e.g. by changing a synaptic weight or bias of neuron j in layer l)

Strategy:

1. Calculate error Δ_j^L in last layer L using chain rule w.r.t. perceptron output
 2. Calculate error in previous layers $l < L$ using chain rule
 3. Then calculate partial derivatives of L w.r.t. w and b using chain rule
- we need gradient of the loss (cost) function w.r.t. network parameters

Note:

- In Theano / Tensorflow nearly all layers can be differentiated symbolically and in Pytorch automatically
- Enables easy implementation of new functions!

Multi-layer perceptron: Derivation of backpropagation (1)

1. Error Δ_j^L in last layer L :

$$\Delta_j^L = \frac{\partial L}{\partial z_j^L} = \frac{\partial L}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L} = \frac{\partial L}{\partial a_j^L} f'(z_j^L) \text{ since } a_j^L = f(z_j^L) \quad (\text{eq. 1a})$$

$$\text{or } \Delta^L = \nabla_{\hat{y}} L \odot f'(\mathbf{z}^L) \quad \text{since } \mathbf{a}^L = \hat{\mathbf{y}} \quad (\text{eq. 1b})$$

Hadamard product \odot (elementwise multiplication):

component j of $s \odot t$: $(s \odot t)_j = s_j \cdot t_j$, e.g. $\begin{pmatrix} 1 \\ 2 \end{pmatrix} \odot \begin{pmatrix} 3 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \cdot 3 \\ 2 \cdot 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \end{pmatrix}$

2. Error Δ_j^l in previous layers $l < L$:

$$\Delta_k^l = \frac{\partial L}{\partial z_k^l} = \sum_{j=1}^{n_{l+1}} \underbrace{\frac{\partial L}{\partial z_j^{l+1}}}_{\Delta_j^{l+1}} \underbrace{\frac{\partial z_j^{l+1}}{\partial a_k^l}}_{f'(z_k^l) \text{ since } a_k^l = f(z_k^l)} = \sum_{j=1}^{n_{l+1}} \Delta_j^{l+1} w_{jk}^{l+1} f'(z_k^l)$$

$$z_j^{l+1} = \sum_k w_{jk}^{l+1} a_k^l + b_j^{l+1} \Rightarrow \frac{\partial z_j^{l+1}}{\partial a_k^l} = w_{jk}^{l+1}$$

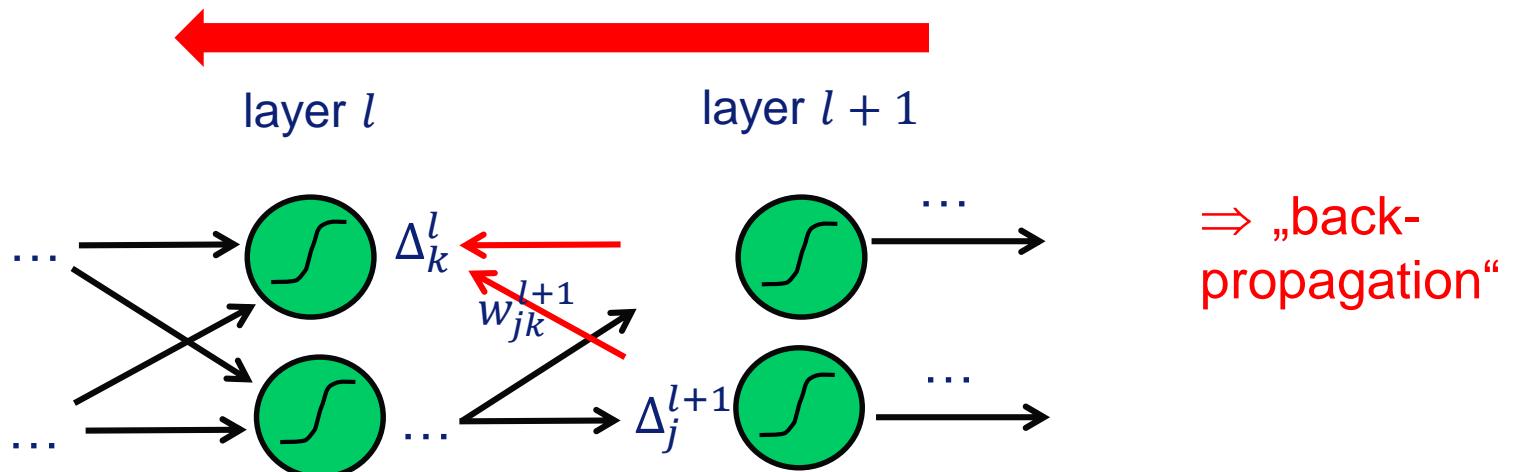
Multi-layer perceptron: Derivation of backpropagation (2)

2. Error Δ_k^l in previous layers $l < L$:For $l = 1, \dots, L - 1$:

$$\Delta_k^l = \sum_{j=1}^{n_{l+1}} \Delta_j^{l+1} w_{jk}^{l+1} f'(z_k^l) \quad (\text{eq. 2a})$$

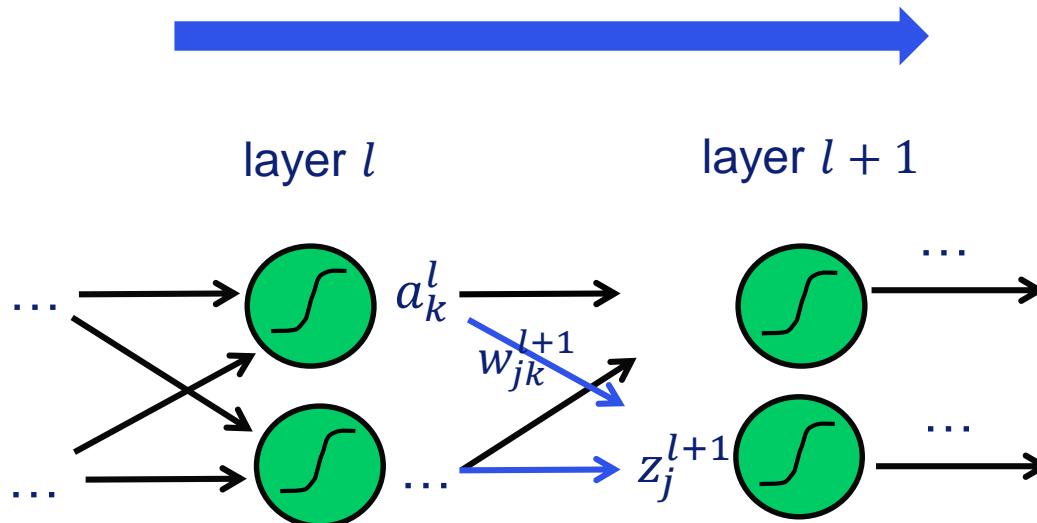
$$\text{or } \Delta^l = \underbrace{\left((\mathbf{w}^{l+1})^T \cdot \Delta^{l+1} \right)}_{\substack{\text{vector} \\ \text{vector}}} \odot \underbrace{f'(\mathbf{z}^l)}_{\text{vector}} \quad (T: \text{transpose}) \quad (\text{eq. 2b})$$

- Error in layer l expressed in terms of errors in „later“ layer $l + 1$



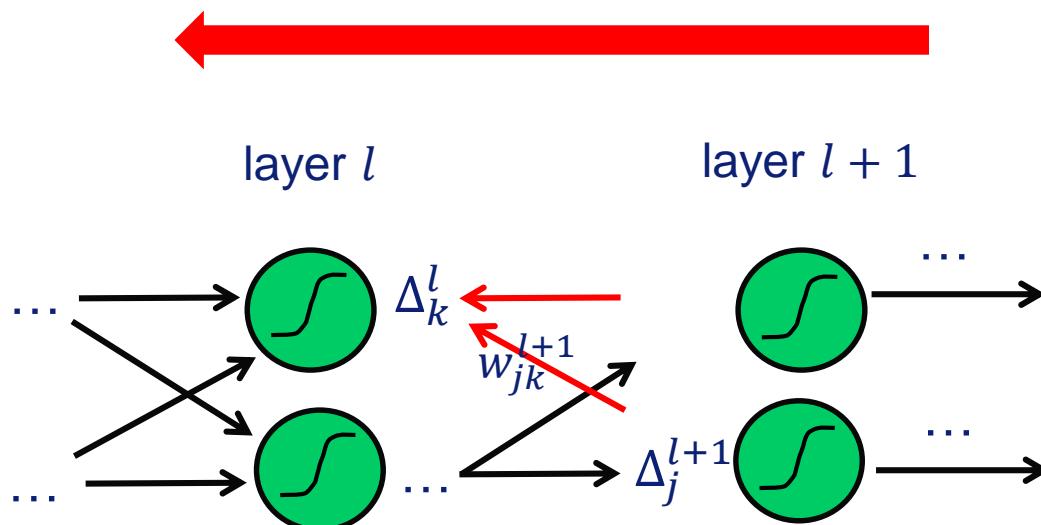
Multi-layer perceptron: Illustration of error backpropagation

- Compare $z_j^{l+1} = \sum_{k=1}^{n_l} w_{jk}^{l+1} a_k^l + b_j^{l+1}$ and $\Delta_k^l = \sum_{j=1}^{n_{l+1}} \Delta_j^{l+1} w_{jk}^{l+1} f'(z_k^l)$:
- Forward propagation of activations: Activations a_k^l of layer l propagated through synaptic weights w_{jk}^{l+1} to PSPs z_j^{l+1} in layer $l + 1$



Multi-layer perceptron: Illustration of error backpropagation

- Compare $z_j^{l+1} = \sum_{k=1}^{n_l} w_{jk}^{l+1} a_k^l + b_j^{l+1}$ and $\Delta_k^l = \sum_{j=1}^{n_{l+1}} \Delta_j^{l+1} w_{jk}^{l+1} f'(z_k^l)$:
- Forward propagation of activations: Activations a_k^l of layer l propagated through synaptic weights w_{jk}^{l+1} to PSPs z_j^{l+1} in layer $l + 1$
- Backward propagation of errors: Errors Δ_j^{l+1} of layer $l + 1$ propagated back through synaptic weights w_{jk}^{l+1} to errors Δ_k^l in layer l



Multi-layer perceptron: Derivation of backpropagation (3)

3. Partial derivative of L w.r.t. bias:

For $l = 1, \dots, L$:

$$\frac{\partial L}{\partial b_j^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \Delta_j^l \text{ since } z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \Rightarrow \frac{\partial z_j^l}{\partial b_j^l} = 1; \frac{\partial L}{\partial z_j^l} = \Delta_j^l \quad (\text{eq. 3a})$$

$$\text{or } \nabla_{b^l} L = \Delta^l \quad (\text{eq. 3b})$$

4. Partial derivative of L w.r.t. weights:

For $l = 1, \dots, L$:

$$\frac{\partial L}{\partial w_{jk}^l} = \frac{\partial L}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \text{ since } \frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \text{ (see above)}; \frac{\partial L}{\partial z_j^l} = \Delta_j^l \quad (\text{eq. 4a})$$

$$\text{or } \nabla_{W^l} L = \Delta^l \cdot (a^{l-1})^T \quad (T: \text{transpose}) \quad (\text{eq. 4b})$$

Note: $a^0 = x$ (network input)

→ insert eq. 3b and 4b into stochastic gradient descent parameter update

MLP: Backpropagation algorithm with stochastic gradient descent

- **Initialization:** Network parameter $\Theta = \{\mathbf{W}^l, \mathbf{b}^l\}_{l=1}^L$, learning rate η
- **While** stopping criterion not met **do**
 1. Sample mini-batch $\{x^{(1)}, \dots, x^{(m)}\}$ from training set with targets $y^{(i)}$
 2. For each training example $x^{(\mu)}$: Set input activation $a^{0,\mu} = x^{(\mu)}$ and do:
 - i. **Forward propagation:** For $l = 1, \dots, L$ compute

$$\mathbf{z}^{l,\mu} = \mathbf{W}^l \mathbf{a}^{l-1,\mu} + \mathbf{b}^l \quad \text{and} \quad \mathbf{a}^{l,\mu} = f(\mathbf{z}^{l,\mu})$$
 - ii. **Output error:** Compute $\Delta^{L,\mu} = \nabla_{\hat{y}^\mu} L \odot f'(\mathbf{z}^{L,\mu})$ (eq. 1b)
 - iii. **Error backpropagation:** For $l = L-1, \dots, 1$ compute

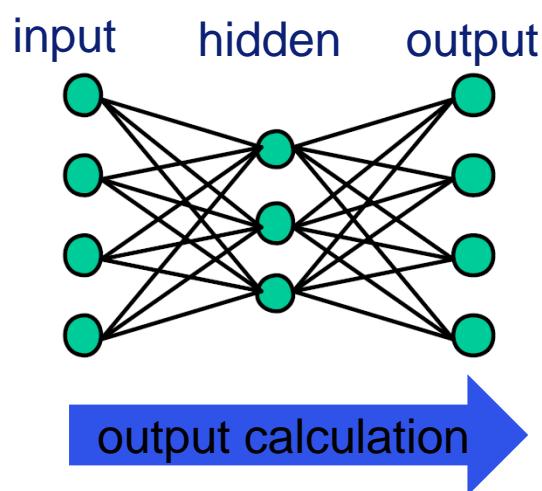
$$\Delta^{l,\mu} = \left((\mathbf{W}^{l+1})^T \cdot \Delta^{l+1,\mu} \right) \odot f'(\mathbf{z}^{l,\mu})$$
 (eq. 2b)
 3. **Stochastic gradient descent:** For $l = L, \dots, 1$ update (regularisation optional)

$$\mathbf{W}^l(t+1) = \mathbf{W}^l(t) - \frac{\eta}{m} \sum_{\mu=1}^m \Delta^{l,\mu} \cdot (\mathbf{a}^{l-1,\mu})^T - \eta \lambda \frac{\partial \Omega(\Theta)}{\partial \mathbf{W}^l(t)}$$
 (using eq. 4b)

$$\mathbf{b}^l(t+1) = \mathbf{b}^l(t) - \frac{\eta}{m} \sum_{\mu=1}^m \Delta^{l,\mu} - \eta \lambda \frac{\partial \Omega(\Theta)}{\partial \mathbf{b}^l(t)}$$
 (using eq. 3b)
- **End while**

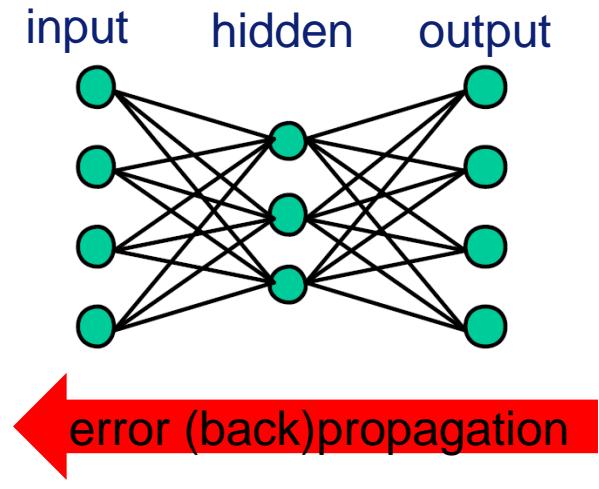
Multi-layer perceptron: Backpropagation

- Generalisation of gradient learning to multi-layer perceptrons
 - Supervised learning algorithm
1. Forward pass: Forward propagation of input, calculation of output
 2. Error calculation: Compare network output with target, compute loss / error
 3. Backward pass: backward propagation of errors through the network
→ “backpropagation”



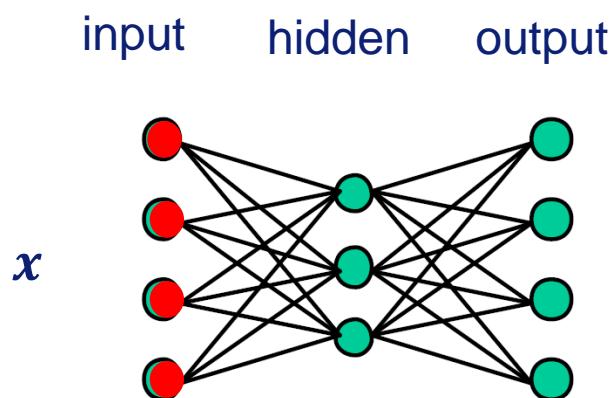
target
 \hat{y}_i \leftrightarrow y_i
 \leftrightarrow
 \leftrightarrow
 \leftrightarrow

error calculation



Backpropagation: Illustration (single input, x , i.e. mini-batch of size 1)

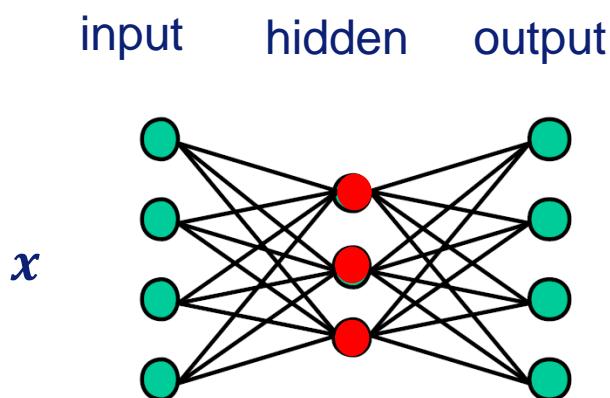
Forward pass: Compute network output for input $x = a^0$:



Backpropagation: Illustration (single input, x , i.e. mini-batch of size 1)

Forward pass: Compute network output for input $x = a^0$:

$$z^1 = W^1 x + b^1 \quad \text{and} \quad a^1 = f(z^1)$$

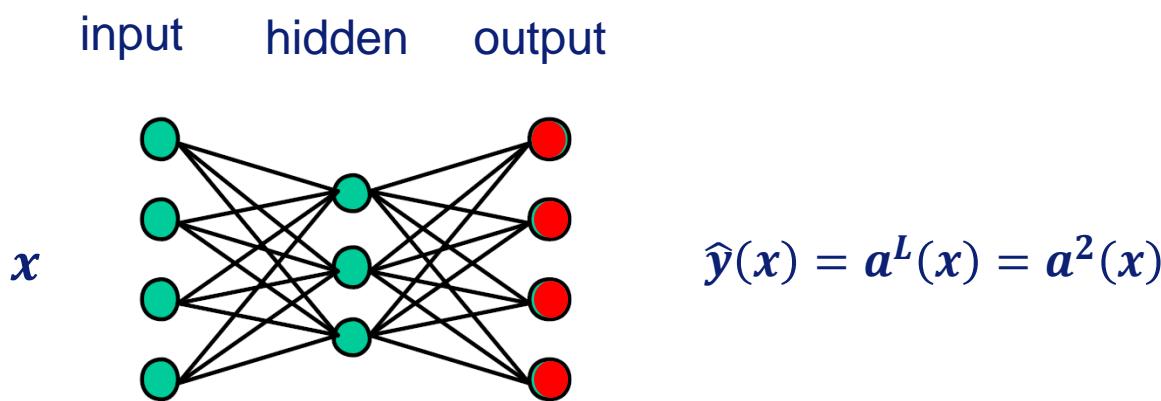


From: Bittel

Backpropagation: Illustration (single input, x , i.e. mini-batch of size 1)

Forward pass: Compute network output for input $x = \mathbf{a}^0$:

$$\mathbf{z}^2 = \mathbf{W}^2 \mathbf{a}^1 + \mathbf{b}^2 \quad \text{and} \quad \mathbf{a}^2 = f(\mathbf{z}^2)$$

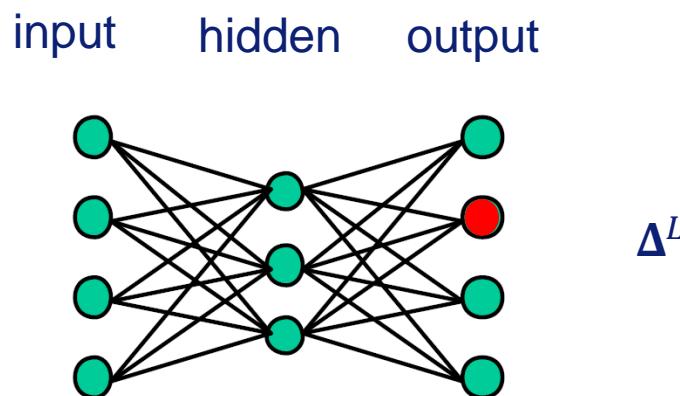


Backpropagation: Illustration (single input, x , i.e. mini-batch of size 1)

Error calculation:

1. Compute error signal Δ^L involving comparison between actual network output \hat{y} and target output y (in definition of loss function) and derivative of activation function f :

$$\Delta^L = \nabla_{\hat{y}} L \odot f'(\mathbf{z}^L)$$



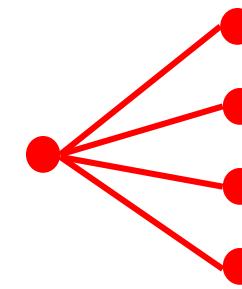
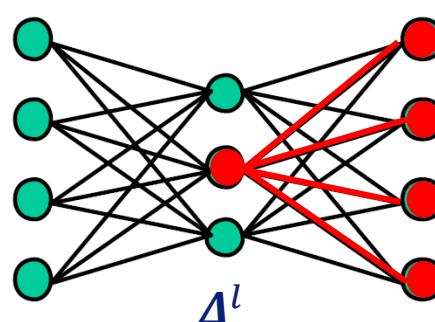
Backpropagation: Illustration (single input, x, i.e. mini-batch of size 1)

Backward pass:

2. Backpropagate error signal:

$$\Delta^l = \left((\mathbf{W}^{l+1})^T \cdot \Delta^{l+1} \right) \odot f'(\mathbf{z}^l)$$

input hidden output



From: Bittel

Backpropagation: Illustration (single input, x, i.e. mini-batch of size 1)

Weight modification: Adjust weights according to

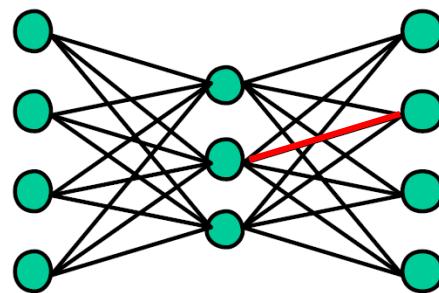
$$\mathbf{W}^l(t+1) = \mathbf{W}^l(t) - \eta \cdot \Delta^l \cdot (\mathbf{a}^{l-1})^T$$

$$\mathbf{b}^l(t+1) = \mathbf{b}^l(t) - \eta \cdot \Delta^l$$

(optionally: add regularization)

$$x_j \quad j \quad \xrightarrow{w_{ij}} \quad i \quad \Delta_i \quad \Delta w_{ij}^l = -\eta \cdot \Delta_i^l \cdot a^{l-1}$$

input hidden output



From: Bittel

Backpropagation: Illustration (mini-batch of size m)

Weight modification: Adjust weights according to

$$\mathbf{W}^l(t+1) = \mathbf{W}^l(t) - \frac{\eta}{m} \sum_{\mu=1}^m \Delta^{l,\mu} \cdot (\mathbf{a}^{l-1,\mu})^T$$

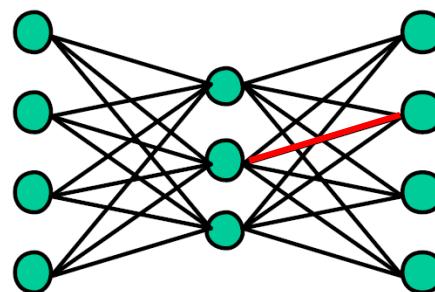
$$\mathbf{b}^l(t+1) = \mathbf{b}^l(t) - \frac{\eta}{m} \sum_{\mu=1}^m \Delta^{l,\mu}$$

(optionally: add regularization)

$$x_j \quad j \quad \xrightarrow{w_{ij}} \quad i \quad \Delta_i$$

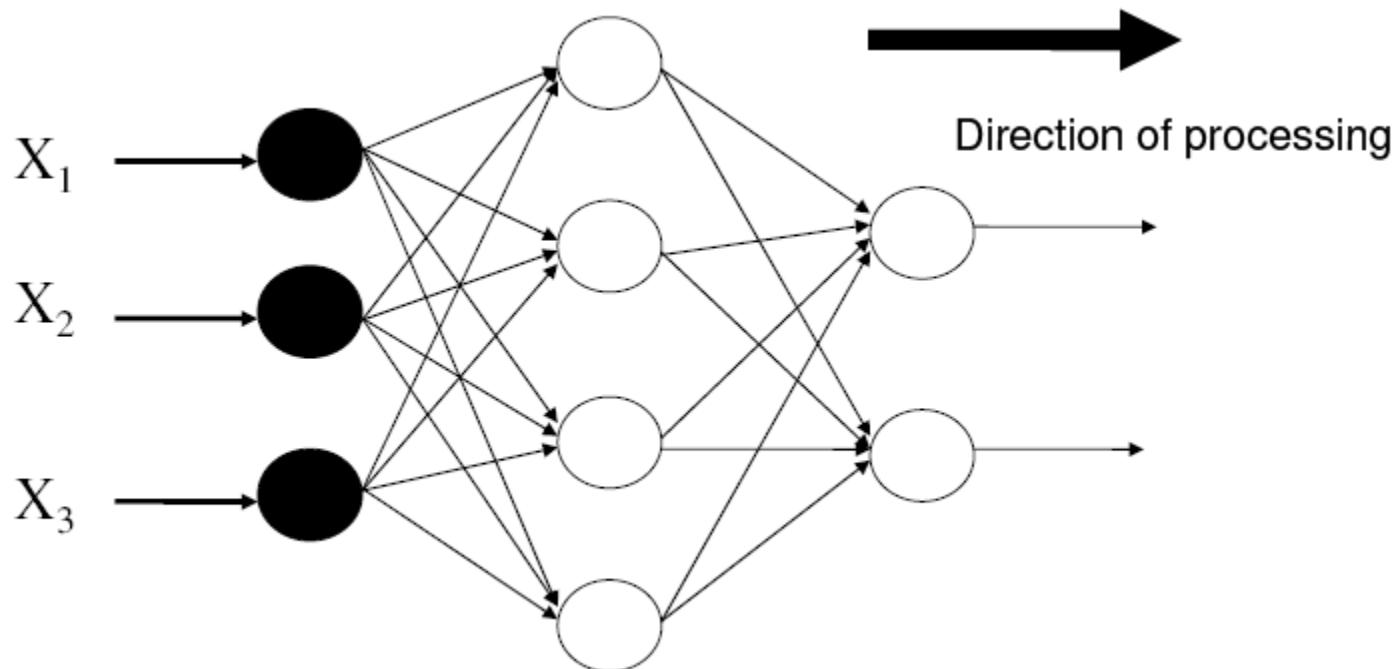
$$\Delta w_{ij}^l = -\frac{\eta}{m} \sum_{\mu=1}^m \Delta_i^{l,\mu} \cdot a_j^{l-1,\mu}$$

input hidden output



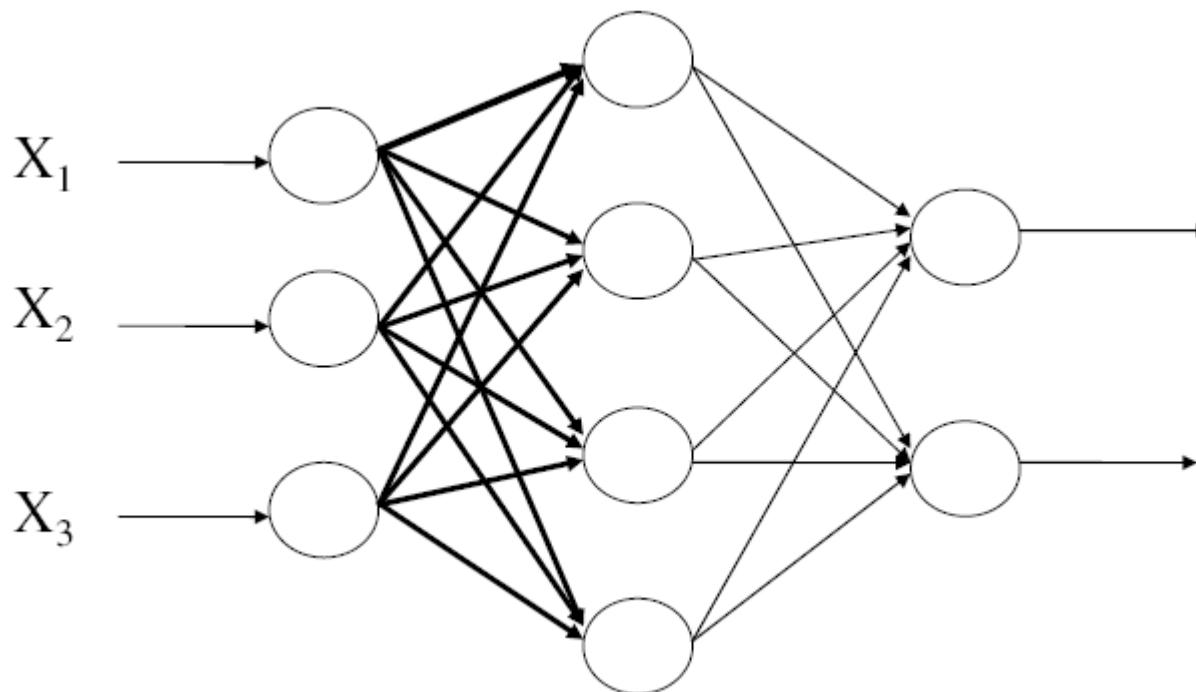
Backpropagation: Illustration

- Step 1: Forward pass (compute network output to given input)



Backpropagation: Illustration

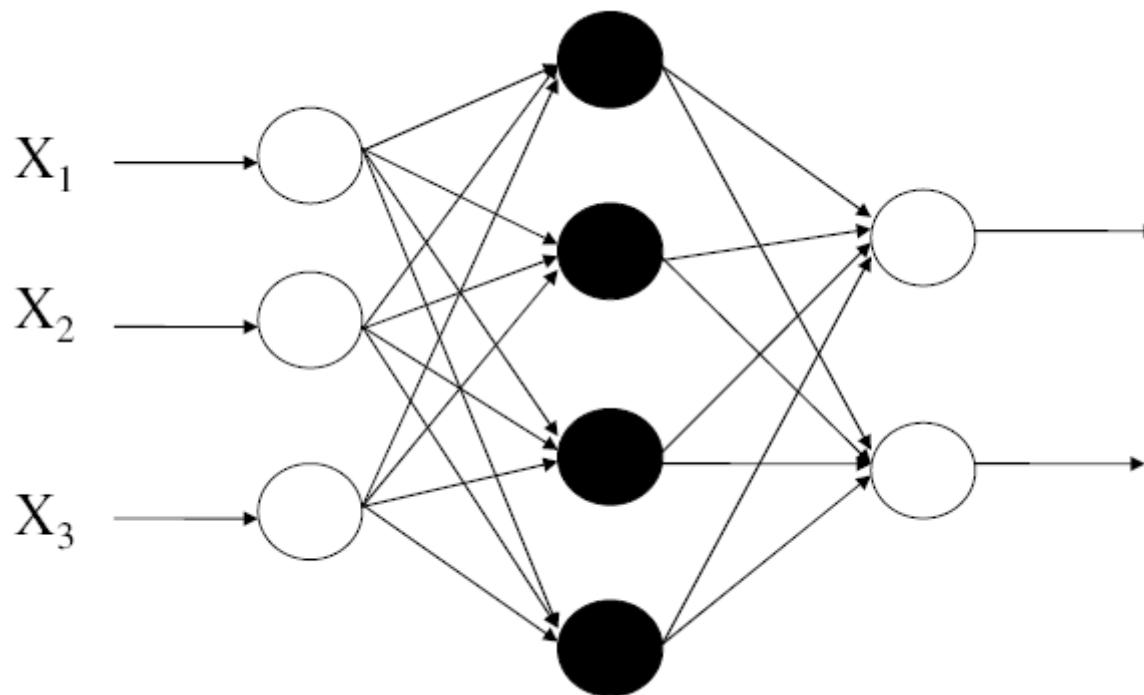
- Step 1: Forward pass (compute network output to given input)



From: Lippe

Backpropagation: Illustration

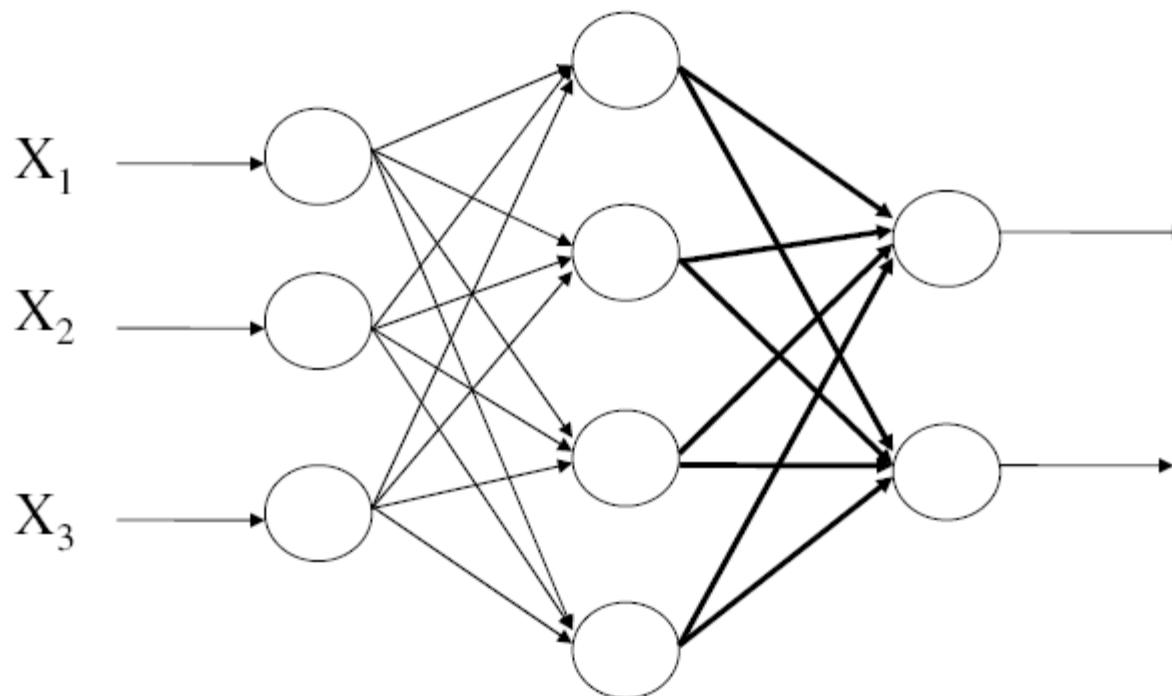
- Step 1: Forward pass (compute network output to given input)



From: Lippe

Backpropagation: Illustration

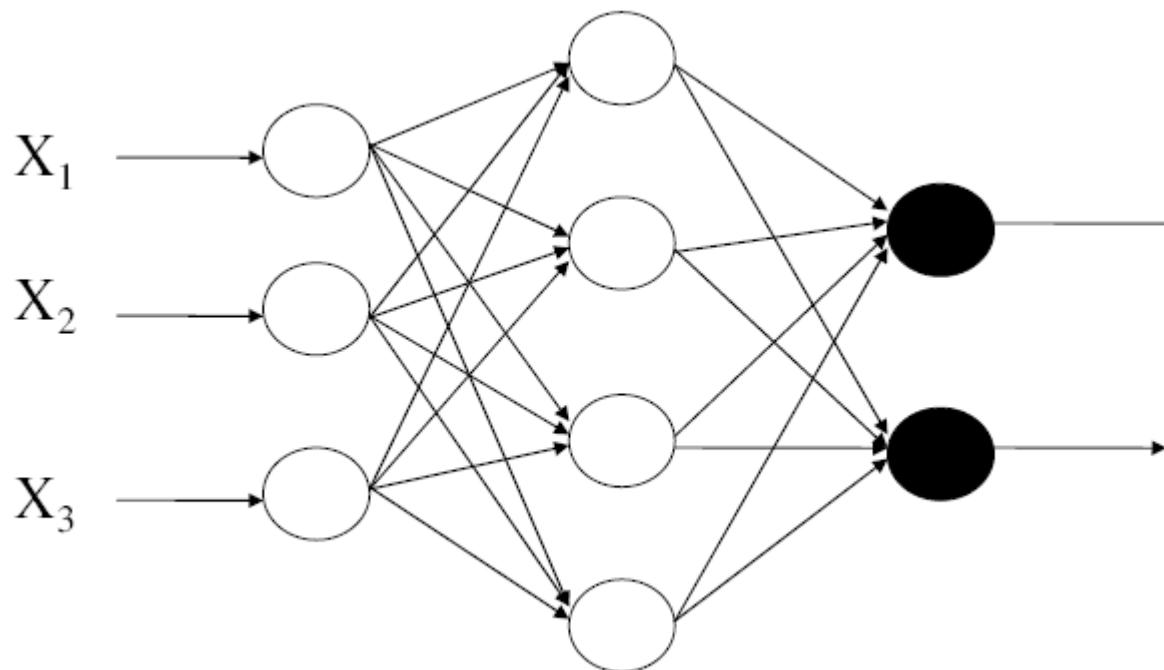
- Step 1: Forward pass (compute network output to given input)



From: Lippe

Backpropagation: Illustration

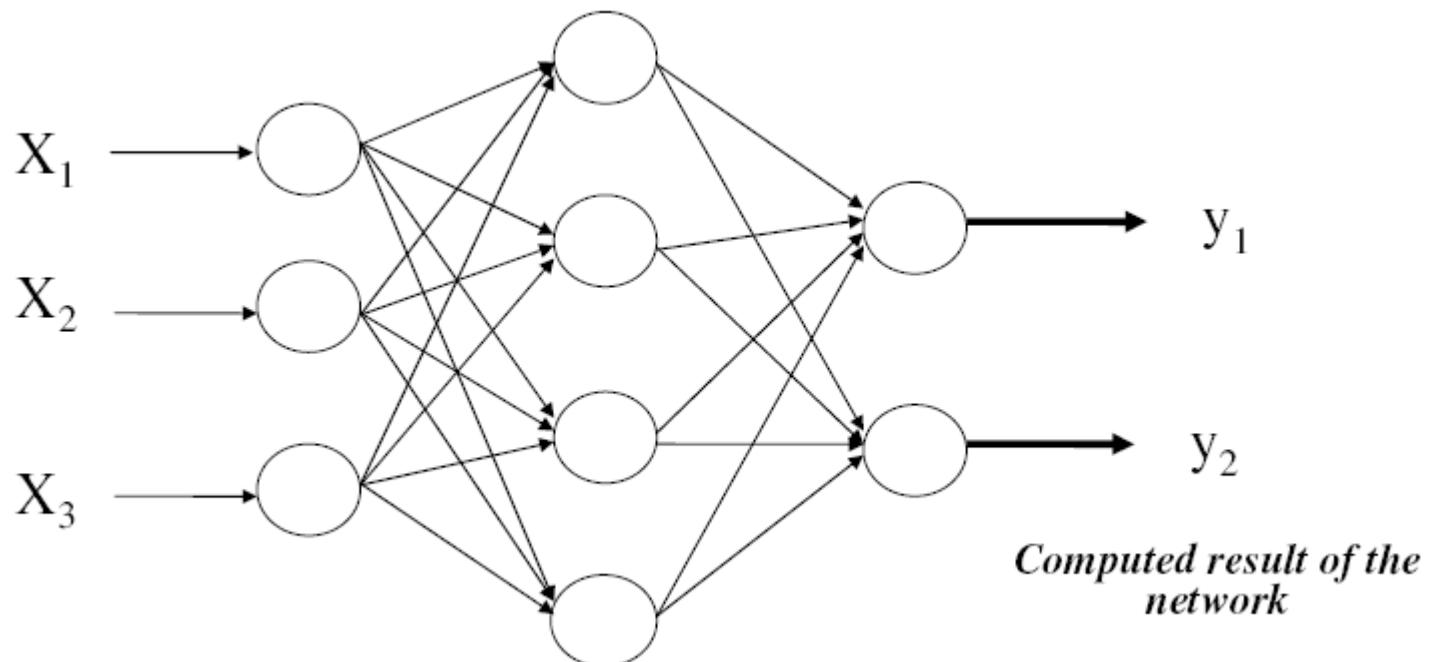
- Step 1: Forward pass (compute network output to given input)



From: Lippe

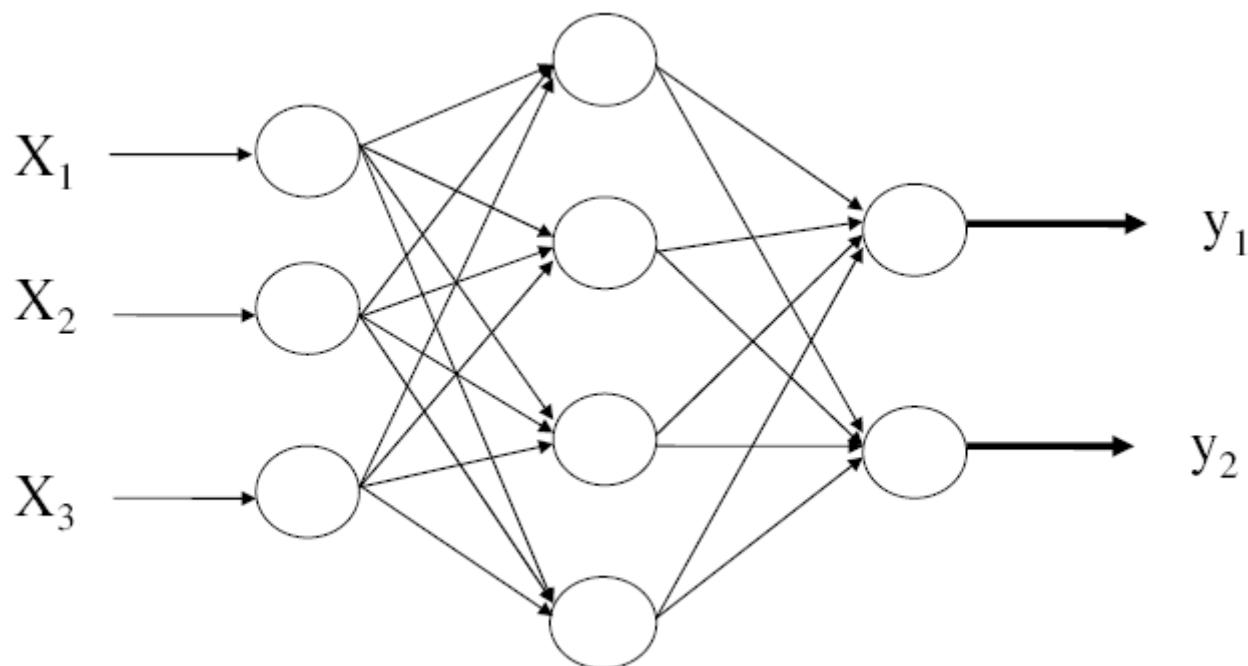
Backpropagation: Illustration

- Step 1: Forward pass (compute network output to given input)



Backpropagation: Illustration

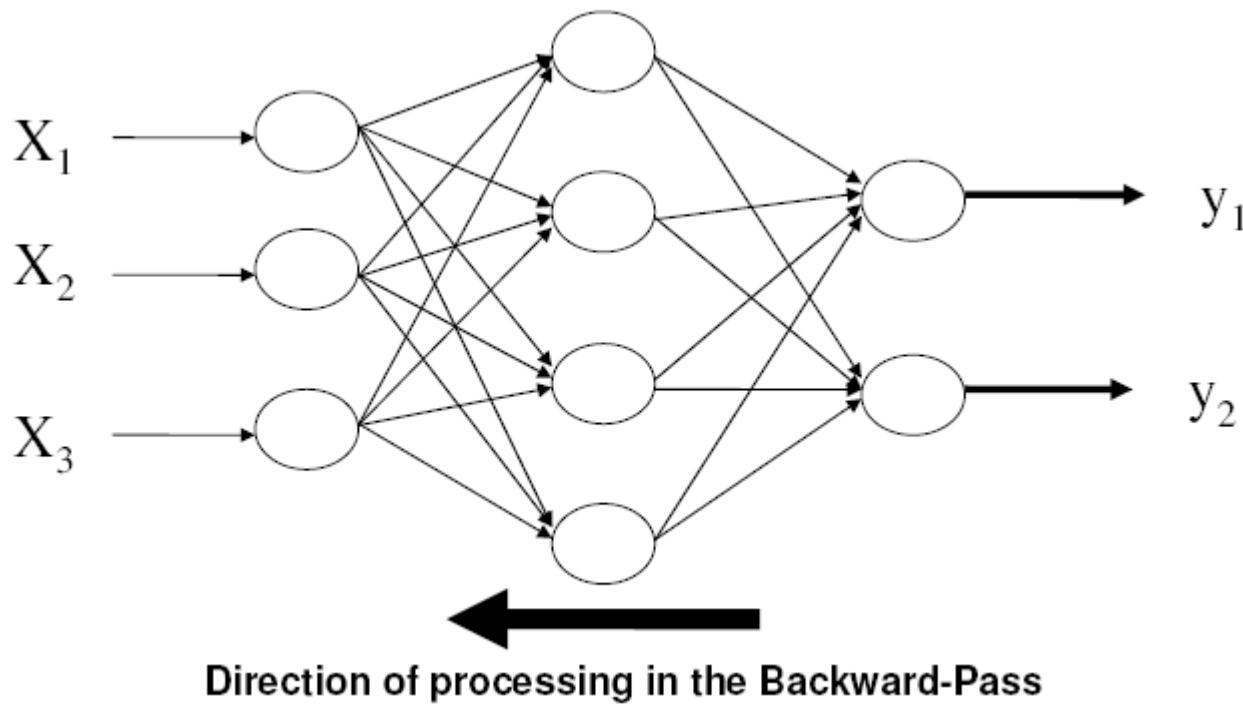
- Step 2: Error calculation



From: Lippe

Backpropagation: Illustration

- Step 3: Backward pass (modify network weights from output to input layer)



Gradient descent: Problems (1)

- **Iterative method**, finds only **local minimum** (no guarantee for global min.)
 - Thus: adequate initialization required
 - Or: several runs with different initializations
 - No problem if local and global minimum equivalent (w.r.t. generalization error)
- **Saddle points, plateaus / flat regions** (gradient zero, but not at a minimum)
 - flat error function: very small gradient → very small steps
- Very **steep regions** in loss function → very large („exploding“) gradients
- **Ill-conditioning**
 - If 2nd derivative (Hessian) too large → even small steps may increase loss
- **Local structure** might not reflect **global structure**
 - Local gradients might not always point towards global solution
- **Inexact gradients**
 - Since gradients are estimated on a mini-batch or based on an approximation

Gradient descent: Problems (2)

- Step size not adequate
 - If too large: risk of stepping out of a good minimum / oscillations
 - If too small: Convergence too slow, may reach non-optimal local minimum
- Oscillations
 - Example: Steep valley, minimum at end of valley; very slow progress
- Long-term dependencies (applies mainly for recurrent neural networks)
 - If same matrix \mathbf{W} is multiplied repeatedly, any eigenvalue not close to 1 in magnitude will either explode or vanish
- Note: Weight modification very small (thus slow convergence) if
 - Activation or error function „flat“ (i.e. derivative nearly zero)
- Note: Weight modification very large (thus minimum may be missed) if
 - Activation or error function „steep“ (i.e. derivative very large)
- Problems if error function behaves differently in different dimensions...

Gradient descent: Illustration

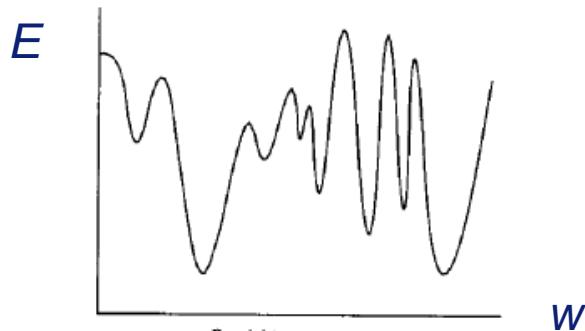
- The learning rate η must be chosen appropriately:

If η too small:

- slow convergence
- may find *local* (not global) minimum

If η too large:

- minimum may be missed
- thus may not converge (or slow)

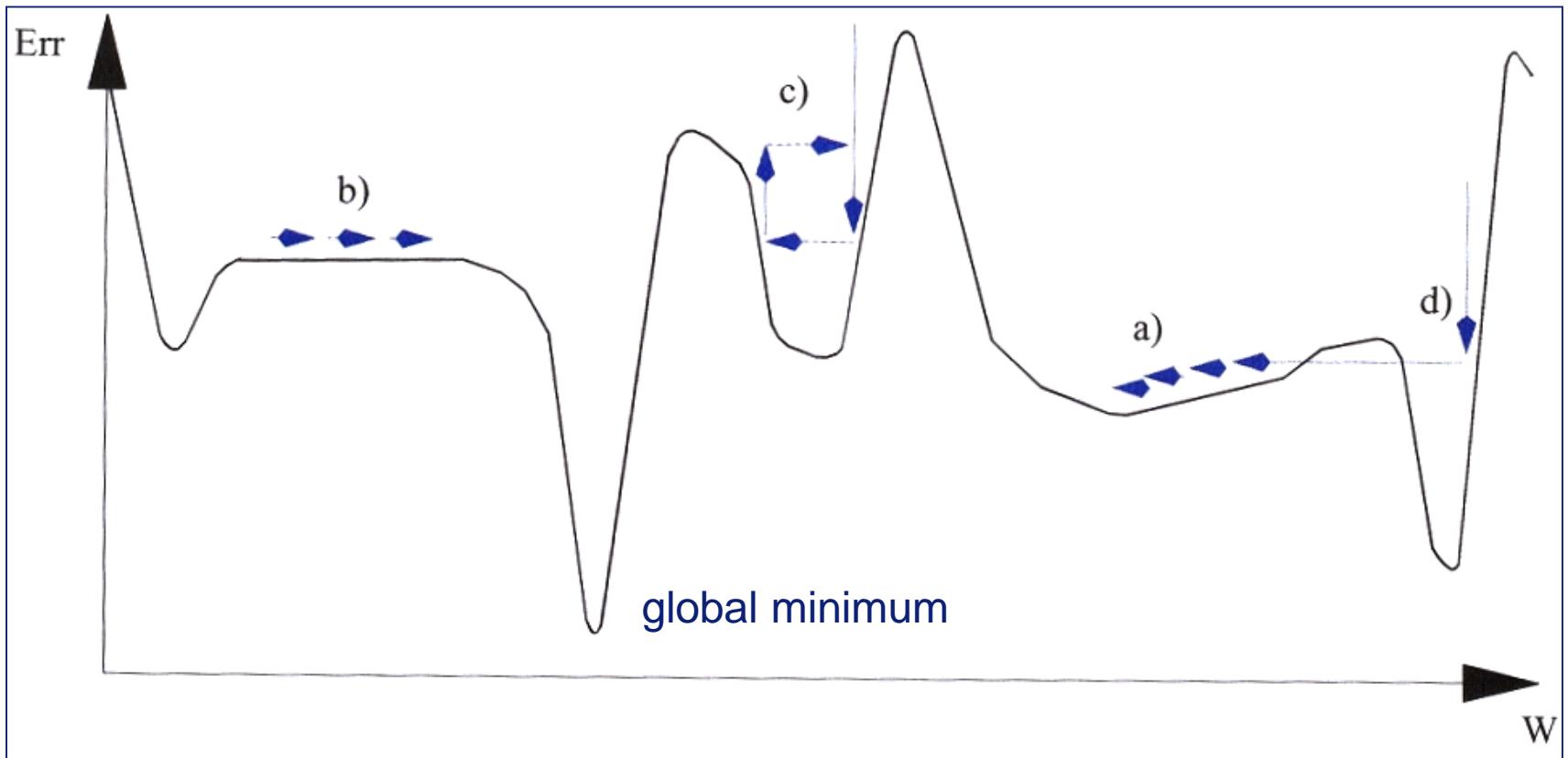


Error as function of the weights may show many (local) minima

- Is solution near local / global minimum?
- Does this provide good generalization?

- The learning rate η is one of the most important parameters!
- Remember: Ultimate goal: Generalization to *unseen* examples!

Gradient descent: Illustration of problems



- a) local minimum
- b) „flat error function“: small gradient \rightarrow very small steps
- c) oscillation
- d) stepping out of a „good“ minimum (step size too large)

From: Hartmann

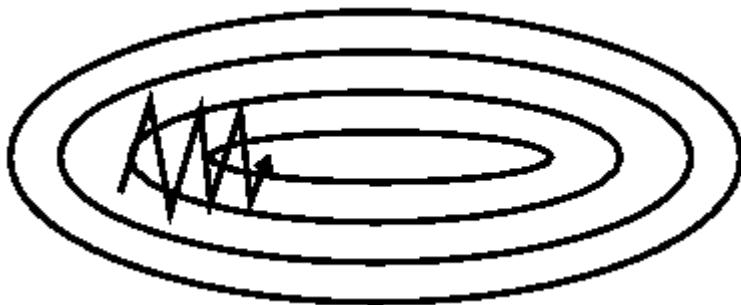
Gradient descent: Extensions (1): Learning with momentum

- Previous weight update: $\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \hat{g}$ \hat{g} : estimated gradient
- Add additional **momentum** term (corresponds to **velocity** if mass = 1):
 $\mathbf{v}(t+1) = \alpha \mathbf{v}(t) - \eta \cdot \hat{g}$ keeps moving average of past gradients; e.g. $\alpha \in \{0.5, 0.9, 0.99\}$
 $\mathbf{w}(t+1) = \mathbf{w}(t) + \mathbf{v}(t+1)$ i.e. $\mathbf{w}(t+1) = \mathbf{w}(t) + \alpha \mathbf{v}(t) - \eta \cdot \hat{g}$
- Weight update not based only on current gradient, but **gradient sequence**:
 - Step size increased if gradients point in same direction: build up „speed“ in directions with consistent gradient
 - Step size reduced if gradients point in opposite directions: oscillations damped in directions of high curvature by combining gradients with opposite signs
 - Like a ball rolling on the error surface; its momentum keeps it going in the previous direction
- But: one more parameter α , not always necessary or useful
 - Might be adapted over time (starting with a small value, then increased)

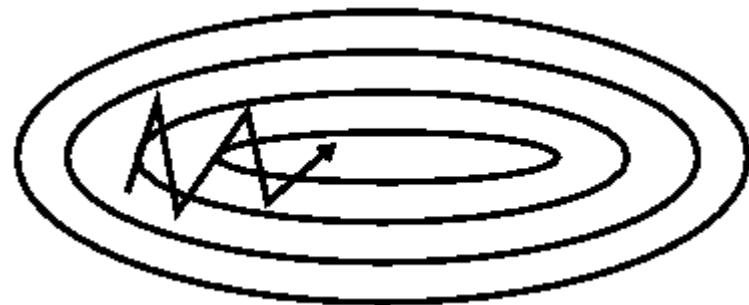
Gradient descent: Extensions (1): Learning with momentum

Illustration:

Without momentum: oscillations



With momentum: oscillations damped



Momentum term increases for dimensions whose gradients point in the same direction / decreases for dimensions whose gradients change direction

- Momentum smoothes parameter updates,
- increases stability (reduced oscillations),
- increases speed (faster convergence)

Gradient descent: Extensions (1): Nesterov momentum

- Motivation: Instead of a ball blindly rolling down a hill, we want a „smart ball“ that knows to slow down before the hill slopes up again
- Thus: Instead of evaluating the gradient at the current parameter values, **evaluate the gradient at an estimation of the next parameter value, i.e. after applying momentum**

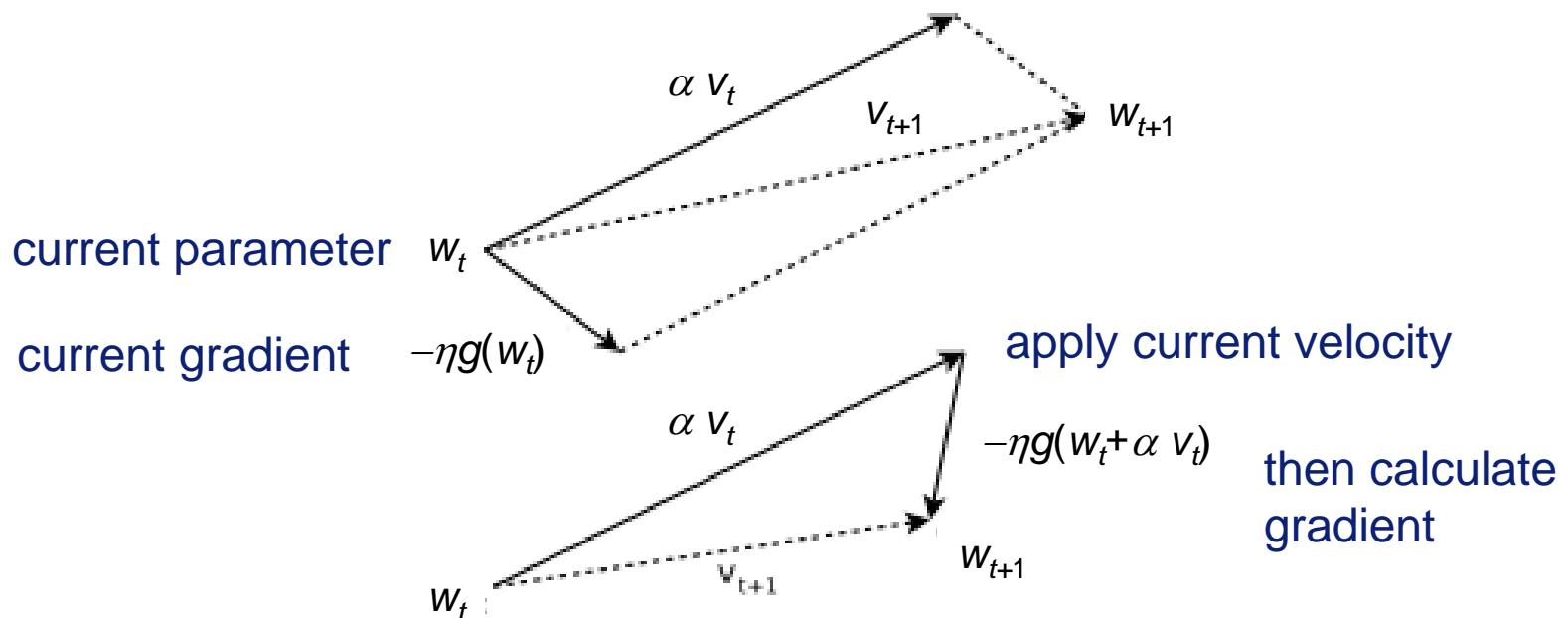


Figure 1. (Top) Classical Momentum (Bottom) Nesterov Accelerated Gradient

Gradient descent: Extensions (2): Adaptive learning rates

AdaGrad (Duchi et al. 2011): Adapts learning rate to the parameters:

- Rapid learning rate decrease in dimensions with largest summed gradient
- Slow learning rate decrease in dimensions with smallest summed gradient
- Previous weight update: $\mathbf{w}(t + 1) = \mathbf{w}(t) - \eta \hat{\mathbf{g}}$; $\hat{\mathbf{g}}$: estimated gradient
- Replaced by: $\mathbf{w}(t + 1) = \mathbf{w}(t) - \frac{\eta}{\delta + \sqrt{\mathbf{r}}} \odot \hat{\mathbf{g}}$ δ : for numerical stability
 \odot : elementwise multiplication
- \mathbf{r} accumulates (sums up) all the historical squared values of the gradient:
$$\mathbf{r}(t + 1) = \mathbf{r}(t) + \hat{\mathbf{g}} \odot \hat{\mathbf{g}}$$
- Disadvantage: Premature and excessive decrease in effective learning rate
- AdaDelta (Zeiler 2012) and RMSProp (Hinton 2012) try to reduce its aggressive, monotonically decreasing learning rate:
 - AdaDelta restricts window of accumulated past gradients to fixed-size window
 - RMSProp uses exponentially weighted moving average (details see literature)

Gradient descent: Extensions (2): Adaptive learning rates

- Adam (Kingma and Ba 2014): Combine advantages of
 - AdaGrad: Works well with sparse gradients
 - RMSProp: Works well in non-stationary settings
- Maintain exponentially decaying averages of gradient (new in Adam) and its square (as in AdaDelta and RMSProp)

→ Update proportional to $\frac{\text{average gradient}}{\sqrt{\text{average squared gradient}}}$

- First moment („momentum“): $\mathbf{v}(t) = \beta_1 \mathbf{v}(t-1) + (1 - \beta_1) \hat{\mathbf{g}}(t)$
- Second moment: $\mathbf{r}(t) = \beta_2 \mathbf{r}(t-1) + (1 - \beta_2) \hat{\mathbf{g}}(t) \odot \hat{\mathbf{g}}(t)$
- Bias correction (otherwise the estimates may initially be biased towards 0):

$$\hat{\mathbf{v}}(t) = \frac{\mathbf{v}(t)}{1 - \beta_1^t} \quad ; \quad \hat{\mathbf{r}}(t) = \frac{\mathbf{r}(t)}{1 - \beta_2^t} \quad (\beta_1^t, \beta_2^t : t \text{ is exponent!})$$

- Update parameters:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \frac{\eta}{\delta + \sqrt{\hat{\mathbf{r}}(t)}} \odot \hat{\mathbf{v}}(t) \quad (\text{operations applied element-wise})$$

Gradient descent: Extensions (3): Second order methods

Newton's Method: Expand loss function L up to second order:

$$L(\mathbf{w}_{t+1}) \approx L(\mathbf{w}_t) + (\mathbf{w}_{t+1} - \mathbf{w}_t)^T \nabla L(\mathbf{w}_t) + \frac{1}{2} (\mathbf{w}_{t+1} - \mathbf{w}_t)^T H(\mathbf{w}_t) (\mathbf{w}_{t+1} - \mathbf{w}_t) \quad H: \text{Hessian matrix}$$

At extremum: Derivative w.r.t. $\Delta\mathbf{w}_t := \mathbf{w}_{t+1} - \mathbf{w}_t$ is zero:

$$\begin{aligned} L(\mathbf{w}_{t+1}) &\approx L(\mathbf{w}_t) + \Delta\mathbf{w}_t^T \nabla L(\mathbf{w}_t) + \frac{1}{2} \Delta\mathbf{w}_t^T H(\mathbf{w}_t) \Delta\mathbf{w}_t \quad \text{Take derivative w.r.t } \Delta\mathbf{w}_t \\ \Rightarrow 0 &= \nabla L(\mathbf{w}_t) + H(\mathbf{w}_t) \Delta\mathbf{w}_t = \nabla L(\mathbf{w}_t) + H(\mathbf{w}_t) (\mathbf{w}_{t+1} - \mathbf{w}_t) \end{aligned}$$

Solve for \mathbf{w}_{t+1} : $\mathbf{w}_{t+1} = \mathbf{w}_t - H(\mathbf{w}_t)^{-1} \nabla L(\mathbf{w}_t)$ Newtons method

Problem: Inversion of the (high-dimensional) matrix H

Solution: Approximation: only keep diagonal elements

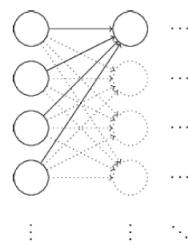
$$w_{ij}(t+1) := w_{ij}(t) - \frac{\frac{\partial L}{\partial w_{ij}}(\mathbf{w}(t))}{\frac{\partial^2 L}{\partial w_{ij} \partial w_{ij}}(\mathbf{w}(t))}$$

Replace gradients by difference quotients
→ Quickprop learning algorithm

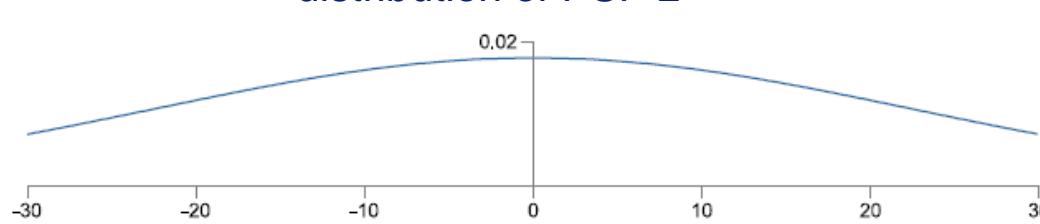
Parameter initialization: Motivation

- Assume the weights are initialized as Gaussian normal variables: $\mathcal{N}(0,1)$
- Variance of z scales with size m of receptive field (for uncorrelated inputs)
- i.e. distribution of postsynaptic potentials quite broad; values often large
- Large values of z may drive neurons into **saturation** → **slow learning**

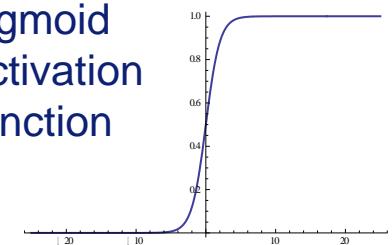
m inputs PSP z



distribution of PSP z

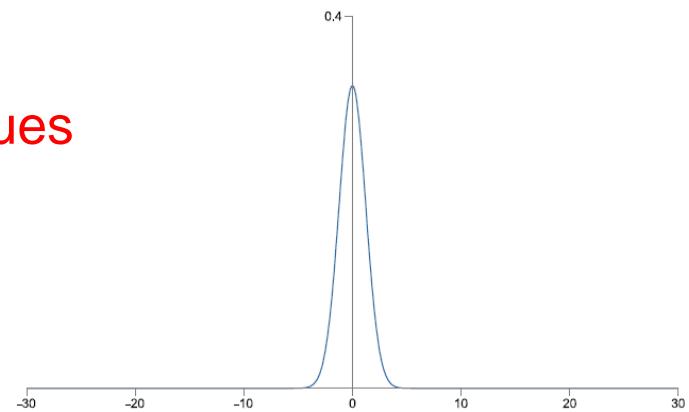


sigmoid activation function



- Instead, initialize weights as $\mathcal{N}(0, \sqrt{1/m})$
- Distribution of PSP z much sharper, **lower values**
- Neurons less likely to saturate
- Less problems with slow learning
- Highlights importance of initializations!

distribution of PSP z



Parameter initialization

Goals: Initial parameter values should

- drive network into the operation (not saturation) regime
- „break symmetry“ between different units
- lead to fast convergence to a minimum with good generalization
 - This, however, can of course not be guaranteed

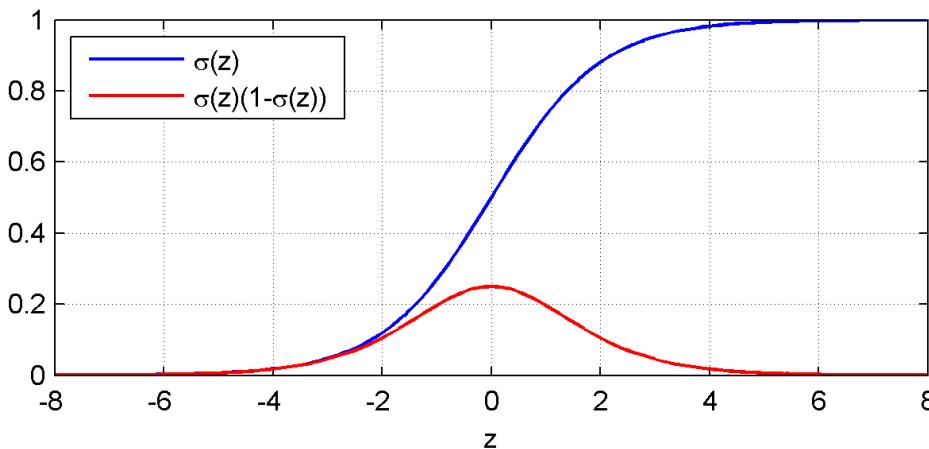
Methods:

- Weights drawn randomly from a normal / Gaussian / uniform distribution
 - Scale of the distribution (\rightarrow „typical“ values) is important!
 - Glorot / Bengio: Initialize all layers (fully connected) to have the same activation variance and same gradient variance:
- $$\mathbf{w}_{ij} \sim \mathcal{N} \left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}} \right) \quad \begin{matrix} m: \text{number of inputs} \\ n: \text{number of outputs} \end{matrix} \quad \left. \right\} \text{of fully connected layer}$$
- But: Weights may become too small if m or n are too large...
 - Alternative: Weight initialization using **pretraining** (see later)
- Biases often set to heuristically chosen constants
 - Often 0 or 0.1 for hidden layers, expected class probability for softmax output

Gradient descent with MSE loss: „Learning slowdown“ for sigmoid f

- Remember MSE loss: $L_{MSE}(\mathbf{w}, \mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2p} \sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)})^2$
- $\rightarrow \frac{\partial L_{MSE}}{\partial w_j} = \frac{1}{p} \sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)}) \cdot f'(z^{(\mu)}) \cdot x_j^{(\mu)}$ weight update in *last* layer L
 $\frac{\partial L_{MSE}}{\partial b} = \frac{1}{p} \sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)}) \cdot f'(z^{(\mu)})$ proportional to **derivative** of activation function (see before, e.g. eq. 1b backpr.)

- Sigmoid (logistic) activation function $\hat{y} = f(z) = \sigma(z) := \frac{1}{1+e^{-z}}$
- $\Rightarrow f'(z) = \frac{e^{-z}}{(1+e^{-z})^2} = \sigma(z) \cdot (1 - \sigma(z)) = \hat{y}(1 - \hat{y})$



Consequence:
 If PSP z is very large (small)
 → gradient very small
 → **nearly no learning**
 (even if output wrong)
 „learning slowdown“

\rightarrow For MSE loss, choose **linear activation function** in final layer $\Rightarrow f'(z^{(\mu)}) = 1$

The problem of vanishing / exploding gradients

- For previous layers $l < L$, remember from backpropagation (eq. 2b):

$$\Delta^{l,\mu} = \left((\mathbf{W}^{l+1})^T \cdot \Delta^{l+1,\mu} \right) \odot f'(\mathbf{z}^{l,\mu}) \quad (*) \quad \text{and} \quad \mathbf{z}^{l,\mu} = \mathbf{W}^l \mathbf{a}^{l-1,\mu} + \mathbf{b}^l \quad (**)$$

Observations:

- Each layer in a multi-layer neural network involves multiplication with f'
- For a logistic f , f' is smaller than 0.25 for all arguments
 - If the weights are „small“, the error terms $\Delta^{l,\mu}$ (and therefore the modifications of weights during learning) get smaller with each layer: „Vanishing gradients“
 - Increasing the weights increases $\mathbf{z}^{l,\mu}$ via eq. (**) and may lead to saturation (due to the sigmoid activation function), so this doesn't help
 - Conversely, if the weights are „too big“, gradients may become too large: „Exploding gradients“
 - In any case gradients are „unstable“, i.e. gradients in early layers may have a different magnitude than gradients in later layers, making learning with many layers difficult

The problem of vanishing / exploding gradients: Illustration

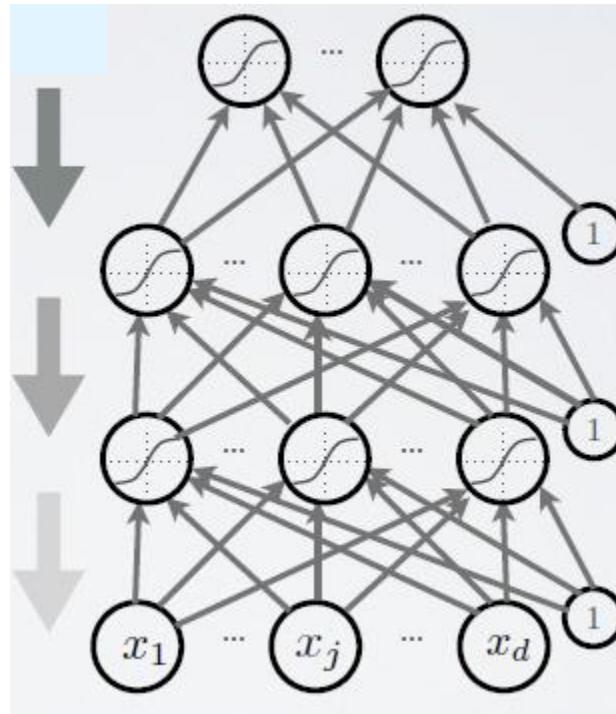
Vanishing gradients

large gradient

smaller gradient

even smaller /
vanishing gradient

(schematic representation!)



Exploding gradients

Small gradient

Larger gradient

even larger /
exploding gradient

- With sigmoid activation functions, vanishing gradients prevail

Consequences:

next slides

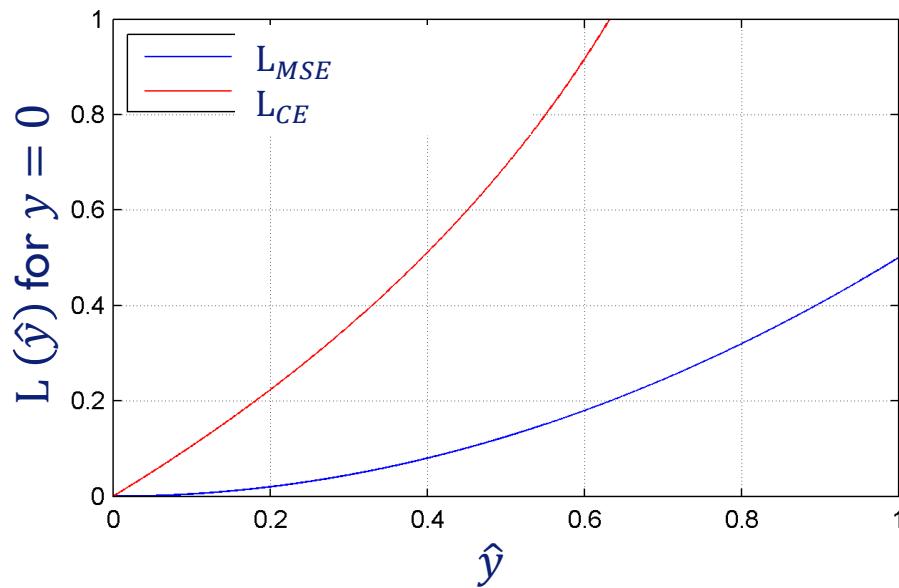
- Use different loss function to mitigate learning slowdown at last layer
- Use different activation function to mitigate vanishing gradient problem
- Use reasonable initial values for the weights and data scaling

Cross-entropy loss

- Alternative loss function: „Cross-entropy (CE) loss“

$$L_{CE}(\mathbf{w}, y, \hat{y}) = -\frac{1}{p} \sum_{\mu=1}^p [y^{(\mu)} \ln \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) + (1 - y^{(\mu)}) \ln(1 - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}))]$$

- Cross-entropy is non-negative if network output \hat{y} and target output $y \in [0,1]$
- Cross-entropy becomes minimal if network output \hat{y} identical to target output y

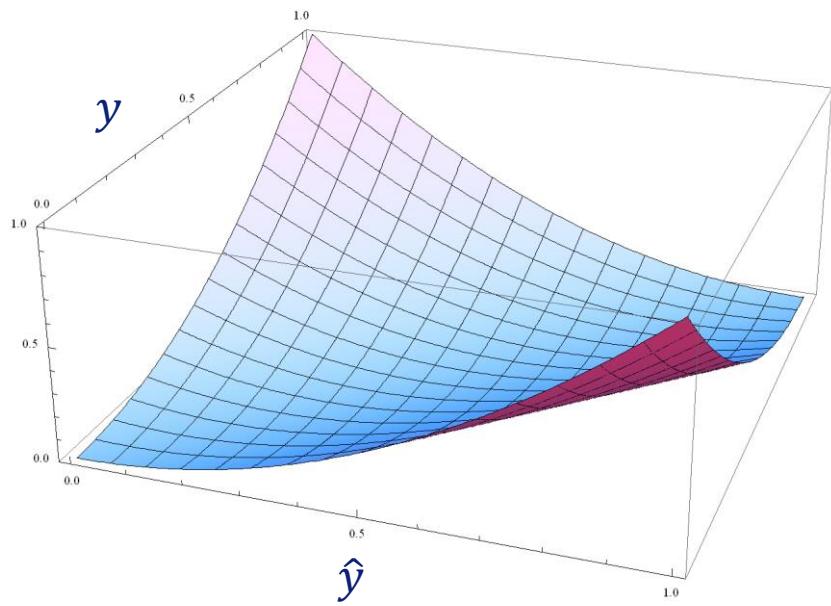


No derivative of activation function,
i.e. no learning slowdown

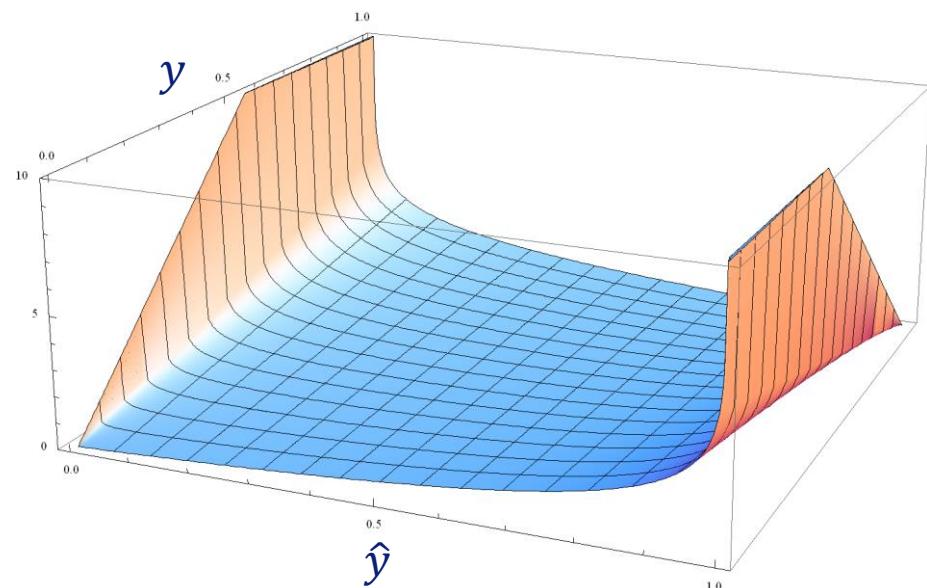
- Parameter update:
$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})) \cdot \mathbf{x}^{(\mu)}$$

Mean squared error loss versus cross-entropy loss

$L_{MSE}(\hat{y})$
(scale from 0 to 1)



$L_{CE}(\hat{y})$
(scale from 0 to 10)



Cross-entropy loss: Parameter updates (for logistic activation function)

- Cross-entropy loss:

$$L_{CE}(\mathbf{w}, y, \hat{y}) = -\frac{1}{p} \sum_{\mu=1}^p [y^{(\mu)} \ln \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) + (1 - y^{(\mu)}) \ln(1 - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}))]$$

With $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}} \Rightarrow \frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$ and $z = \sum_j w_j x_j + b \Rightarrow \frac{\partial z}{\partial w_j} = x_j$:

$$\frac{\partial L_{CE}}{\partial w_j} = \frac{\partial L_{CE}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(\mu)}} \frac{\partial z^{(\mu)}}{\partial w_j} = -\frac{1}{p} \sum_{\mu=1}^p \left[\frac{y^{(\mu)}}{\hat{y}} - \frac{1-y^{(\mu)}}{1-\hat{y}} \right] \hat{y}(1 - \hat{y}) x_j^{(\mu)} = -\frac{1}{p} \sum_{\mu=1}^p \frac{y^{(\mu)} - \hat{y}}{\hat{y}(1 - \hat{y})} \hat{y}(1 - \hat{y}) x_j^{(\mu)}$$

- Thus $\nabla_{\mathbf{w}} L_{CE} = -\frac{1}{p} \sum_{\mu=1}^p (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})) \mathbf{x}^{(\mu)}$
- Similarly $\frac{\partial L_{CE}}{\partial b} = -\frac{1}{p} \sum_{\mu=1}^p (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}))$

no derivative of activation function
 → no learning slowdown
 (for logistic activation function)

$$\Rightarrow \mathbf{w}(t+1) = \mathbf{w}(t) + \eta \frac{1}{p} \sum_{\mu=1}^p (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)})) \mathbf{x}^{(\mu)}$$

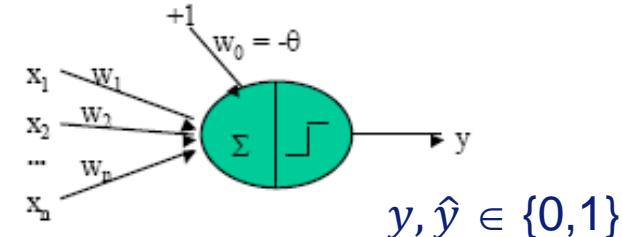
$$\Rightarrow b(t+1) = b(t) + \eta \frac{1}{p} \sum_{\mu=1}^p (y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}))$$

Comparing the learning rules

- Binary perceptron (single training sample $\mathbf{x}^{(\mu)}$): Perceptron learning

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{\left(y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) \right)}_{\text{Difference: 0 or } \pm 1} \cdot \mathbf{x}^{(\mu)}$$

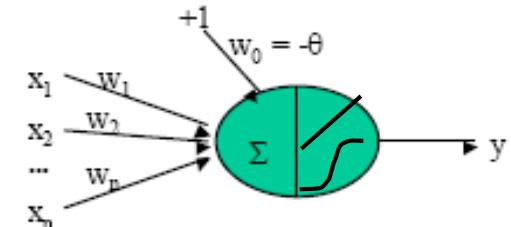
Difference: 0 or ± 1



- Perceptron with differentiable activation function f , MSE loss: LMS rule

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{\left(y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) \right)}_{\text{Difference: } \in [-1, 1] \text{ (f sigmoid)}} \cdot f'(z^{(\mu)}) \cdot \mathbf{x}^{(\mu)}$$

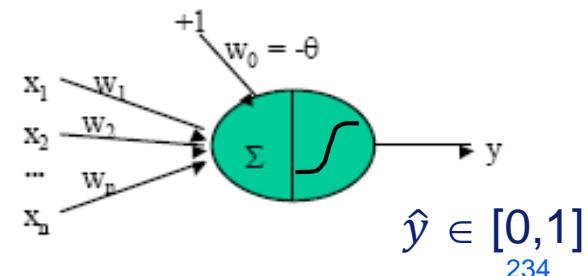
Difference: $\in [-1, 1]$ (f sigmoid)
 $\in \mathbb{R}$ (f linear)



- Perceptron with logistic activation function, cross-entropy loss

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta \cdot \underbrace{\left(y^{(\mu)} - \hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) \right)}_{\text{Difference: } \in [-1, 1] \text{ (f logistic)}} \cdot \mathbf{x}^{(\mu)}$$

Difference: $\in [-1, 1]$ (f logistic)



Log-likelihood loss for softmax activation function

- Assume p training samples $\mathbf{x}^{(\mu)}$ with targets $\mathbf{y}^{(\mu)}$, $\mu = 1, \dots, p$
 - Targets $\mathbf{y}^{(\mu)}$ as „one-hot encoding“ for m output classes:
 - 1 for the correct class $j(\mu)$ for sample μ : $y_j^{(\mu)} = 1$ if $j = j(\mu)$
 - 0 for any other class: $y_j^{(\mu)} = 0$ if $j \neq j(\mu)$
 - Probability to output true class $j(\mu)$ for sample μ (softmax): $\hat{y}_{j(\mu)}(\mathbf{w}, \mathbf{x}^{(\mu)})$
 - All training samples considered independent
- Joint probability to output the true class for all p training samples:

$$\text{Likelihood } (\mathbf{w}, \mathbf{y}, \hat{\mathbf{y}}) = \prod_{\mu=1}^p \hat{y}_{j(\mu)}(\mathbf{w}, \mathbf{x}^{(\mu)})$$

- Negative logarithm → log-likelihood loss:

$$L_{LL}(\mathbf{w}, \mathbf{y}, \hat{\mathbf{y}}) = - \sum_{\mu=1}^p \ln \hat{y}_{j(\mu)}(\mathbf{w}, \mathbf{x}^{(\mu)})$$

Minimal loss corresponds to maximal output probability for the true class!

Log-likelihood loss: Parameter updates (softmax activation function)

- Parameter updates for log-likelihood loss:

$$L_{LL}(\mathbf{W}, y, \hat{y}) = -\sum_{\mu=1}^p \ln \hat{y}_{j(\mu)}(\mathbf{W}, \mathbf{x}^{(\mu)}) \quad \text{with} \quad \hat{y}_{j(\mu)} = \frac{e^{z_{j(\mu)}}}{\sum_{k=1}^m e^{z_k}}$$

$$\Rightarrow \frac{\partial \hat{y}_{j(\mu)}}{\partial z_j} = \frac{\delta_{j,j(\mu)} e^{z_{j(\mu)}} \sum_{k=1}^m e^{z_k} - e^{z_{j(\mu)}} e^{z_j}}{(\sum_{k=1}^m e^{z_k})^2} = \delta_{j,j(\mu)} \hat{y}_{j(\mu)} - \hat{y}_{j(\mu)} \hat{y}_j$$

$$\frac{\partial L_{LL}}{\partial w_{jk}} = \frac{\partial L_{LL}}{\partial \hat{y}_{j(\mu)}} \frac{\partial \hat{y}_{j(\mu)}}{\partial z_{j(\mu)}} \frac{\partial z_{j(\mu)}}{\partial w_{jk}} = - \sum_{\mu=1}^p \frac{1}{\hat{y}_{j(\mu)}} (\delta_{j,j(\mu)} \hat{y}_{j(\mu)} - \hat{y}_{j(\mu)} \hat{y}_j) x_k^{(\mu)} = \sum_{\mu=1}^p (\hat{y}_j - y_j^{(\mu)}) x_k^{(\mu)}$$

- Similarly $\frac{\partial L_{LL}}{\partial b_j} = \sum_{\mu=1}^p (\hat{y}_j - y_j^{(\mu)})$ so we get for the parameter updates:

$$\mathbf{W}(t+1) = \mathbf{W}(t) - \eta \sum_{\mu=1}^p (\hat{\mathbf{y}} - \mathbf{y}^{(\mu)}) \cdot \mathbf{x}^{(\mu)T}$$

$$\mathbf{b}(t+1) = \mathbf{b}(t) - \eta \sum_{\mu=1}^p (\hat{\mathbf{y}} - \mathbf{y}^{(\mu)})$$

- Same form as for logistic activation function with cross-entropy loss
 - Without derivative of activation function, so no learning slowdown

Log-likelihood loss for softmax activation function: Summary

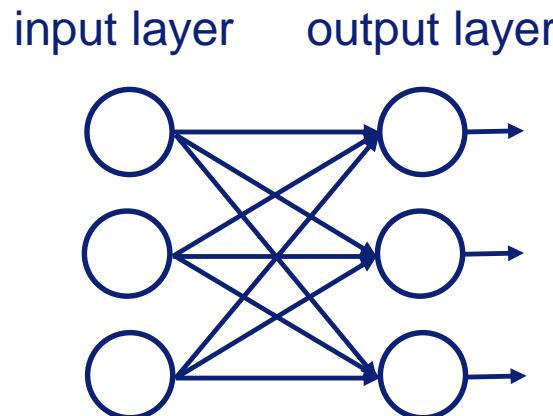
- Loss function for softmax activation function: **log-likelihood loss**:

$$L_{LL}(\mathbf{W}, y, \hat{y}) = - \sum_{\mu=1}^p \ln \hat{y}_{j(\mu)}(\mathbf{W}, \mathbf{x}^{(\mu)})$$

Minimal loss corresponds to maximal output probability for the true class!

- Parameter update:

$$\mathbf{W}(t + 1) = \mathbf{W}(t) - \eta \sum_{\mu=1}^p (\hat{y} - y^{(\mu)}) \cdot \mathbf{x}^{(\mu)T}$$



Log-likelihood loss also called categorical cross-entropy loss

Summary of loss functions with corresponding activation functions

- m output units $\hat{y}_j, j = 1, \dots, m$ (vector $\hat{\mathbf{y}}$)

- p training samples $\mathbf{x}^{(\mu)}$ with targets $\mathbf{y}^{(\mu)}, \mu = 1, \dots, p$

- Mean squared error loss for linear activation function $\hat{\mathbf{y}} = f(\mathbf{z})$:

$$L_{MSE}(\mathbf{W}, \mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2p} \sum_{\mu=1}^p (\hat{\mathbf{y}}(\mathbf{W}, \mathbf{x}^{(\mu)}) - \mathbf{y}^{(\mu)})^2 = \frac{1}{2p} \sum_{\mu=1}^p \sum_{j=1}^m (\hat{y}_j(\mathbf{W}, \mathbf{x}^{(\mu)}) - y_j^{(\mu)})^2$$

- Cross-entropy loss for logistic activation function $\hat{\mathbf{y}} = f(\mathbf{z})$:

$$L_{CE}(\mathbf{W}, \mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{p} \sum_{\mu=1}^p \sum_{j=1}^m \left[y_j^{(\mu)} \ln \hat{y}_j(\mathbf{W}, \mathbf{x}^{(\mu)}) + (1 - y_j^{(\mu)}) \ln (1 - \hat{y}_j(\mathbf{W}, \mathbf{x}^{(\mu)})) \right]$$

- Log-likelihood loss for softmax activation function $\hat{\mathbf{y}} = f(\mathbf{z})$:

$$L_{LL}(\mathbf{W}, \mathbf{y}, \hat{\mathbf{y}}) = - \sum_{\mu=1}^p \ln \hat{y}_{j(\mu)}(\mathbf{W}, \mathbf{x}^{(\mu)})$$

Sum of negative log of probability for correct class
(also referred to as „categorical cross-entropy loss“)

Activation versus loss functions: Summary

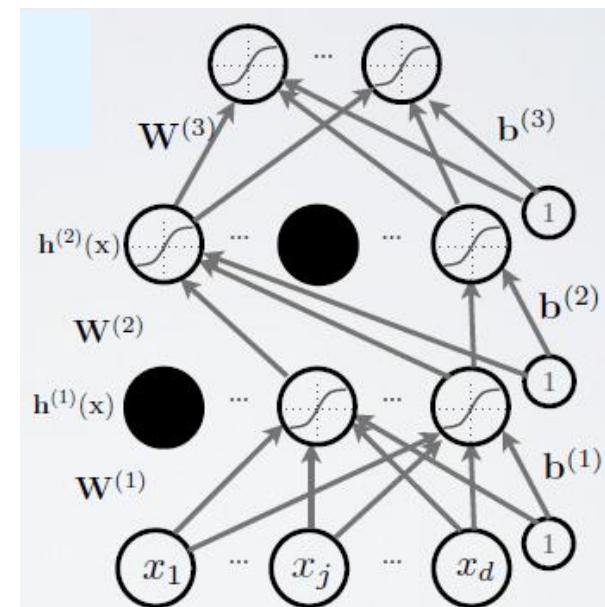
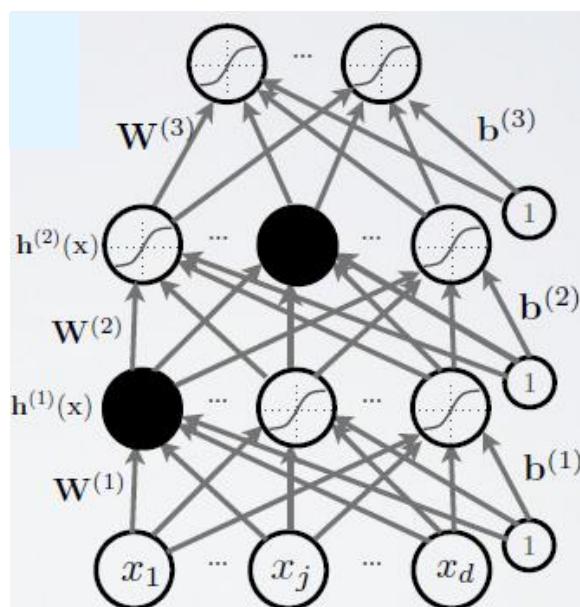
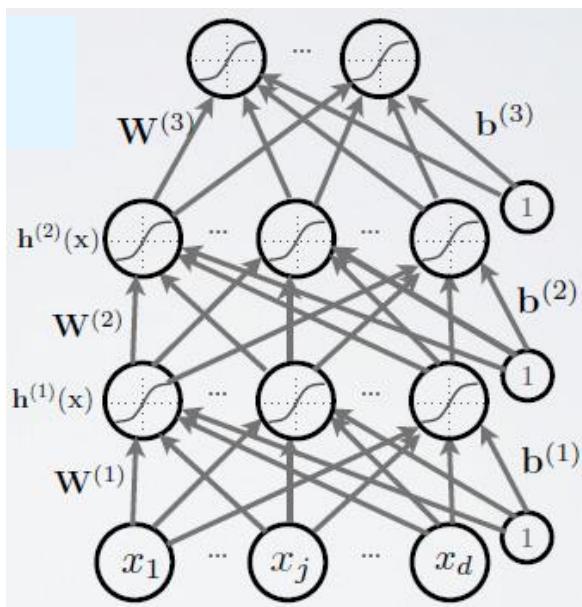
- Reasonable combinations for tackling the learning slowdown problem (in the output layer) are:

Activation Function $f(z)$	Loss function L
Linear	Mean squared error
Logistic	Cross-Entropy
Softmax	Log-likelihood

- Note: When switching between these scenarios, the learning rates are not directly comparable (must be optimized again)

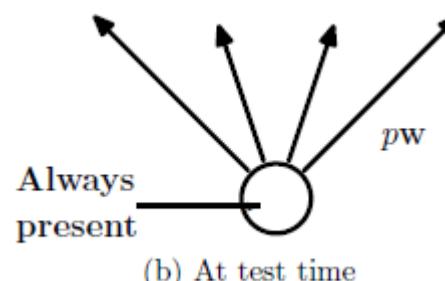
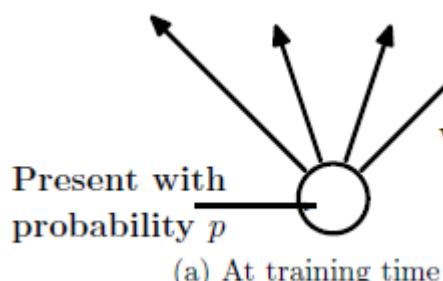
Dropout (1)

- Idea: „Cripple“ neural network by removing hidden units stochastically
 - Each hidden unit is removed with probability $1 - p$ (kept with prob. p), e.g. 0.5
 - Theoretically, a new thinned network is sampled for every training case
 - Hidden units cannot co-adapt to other units
 - Hidden units must be more generally useful



Dropout (2)

- For a neural network of N units, there are 2^N possible thinned networks
- However, these networks share weights \rightarrow still $O(N^2)$ parameters in total
- Dropout is more efficient than training a diverse *ensemble* of neural networks, which is averaged over at test time
 - Both methods generally lead to better generalization
 - However, training an ensemble of networks is prohibitive for large scale tasks
- Realization:
 - Training: Divide training data into **mini-batches** of training cases
 - Sample a thinned network for each mini-batch
 - Use standard forward / backpropagation on thinned network
 - Test: Use **all units** with weights multiplied with p



often implemented in training instead of test

Batch normalization (1)

- Input normalization: Distribution of network inputs assumed to be normal
- Problem: The *activations* (outputs of first layer) are *not* normal („whitened“)
- During training, network parameters change → distribution of activations at internal nodes change during training: „internal covariate shift“
 - For sigmoid activations: Learning may drive inputs into saturation
 - Then, learning must „calibrate“ before discriminating between samples
 - Effect is amplified as network depth increases
 - Therefore, ReLUs and small learning rates are often used
- Backpropagation has to cope with internal covariate shift → slow learning

... slowing down learning

Solution:

- Normalize („whiten“) the layer outputs (activations) for every layer and every dimension independently, based on statistics estimated from the current minibatch
- For inference, a long-term estimate is used (based on a low-pass-filtered version of the minibatch statistics)

Batch normalization (2)

Details: Whitening, scale + shift inputs \mathbf{x} for each layer / minibatch (gives \mathbf{y}):

- Input dimensions treated independently
- For each feature dimension x^k of input \mathbf{x} (index k omitted):

$$\text{Minibatch } B = \{x_1 \dots x_m\}: \quad \mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad ; \quad \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2$$

$$\text{Normalize: } \hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \quad (\varepsilon \text{ for stability}); \quad \text{scale and shift: } y_i = \gamma \hat{x}_i + \beta$$

- Scale and shift transform to preserve network capacity (includes identity transf.)
 - Else, for sigmoid activ., normalized inputs would only operate in linear regime
- γ and β are learned per dimension k (minimizing loss, similar to weights)
- ConvNets: Use \mathbf{z} for \mathbf{x} (not \mathbf{a}), average also over feature map locations

Benefits:

- Batch normalization **accelerates training** by allowing higher learning rates
- Also **acts as regularizer**, in some cases eliminating the need for dropout
- Reduces dependence of gradients **on scale of parameters or initial values**
- Allows to use **saturating** (sigmoid) activation functions

Factors influencing learning performance

General factors:

- **Training data**
 - number of samples, consistency, pre-processing
 - do they sufficiently well represent the data / problem?

Additional factors relevant for iterative learning (gradient, backpropagation):

- **Application of learning algorithm**
 - Initialisation
 - Selection / modification of meta-parameters (e.g. learning rate, mini-batch size)
 - Stopping criterion

Additional factors relevant for perceptrons / multi-layer perceptrons:

- **Network architecture**
 - Number and type of neurons (activation function)
 - Number of hidden layers
 - ...

Influence of training data

The result of learning strongly depends on the quality of the training data!

Ideally, training data should be

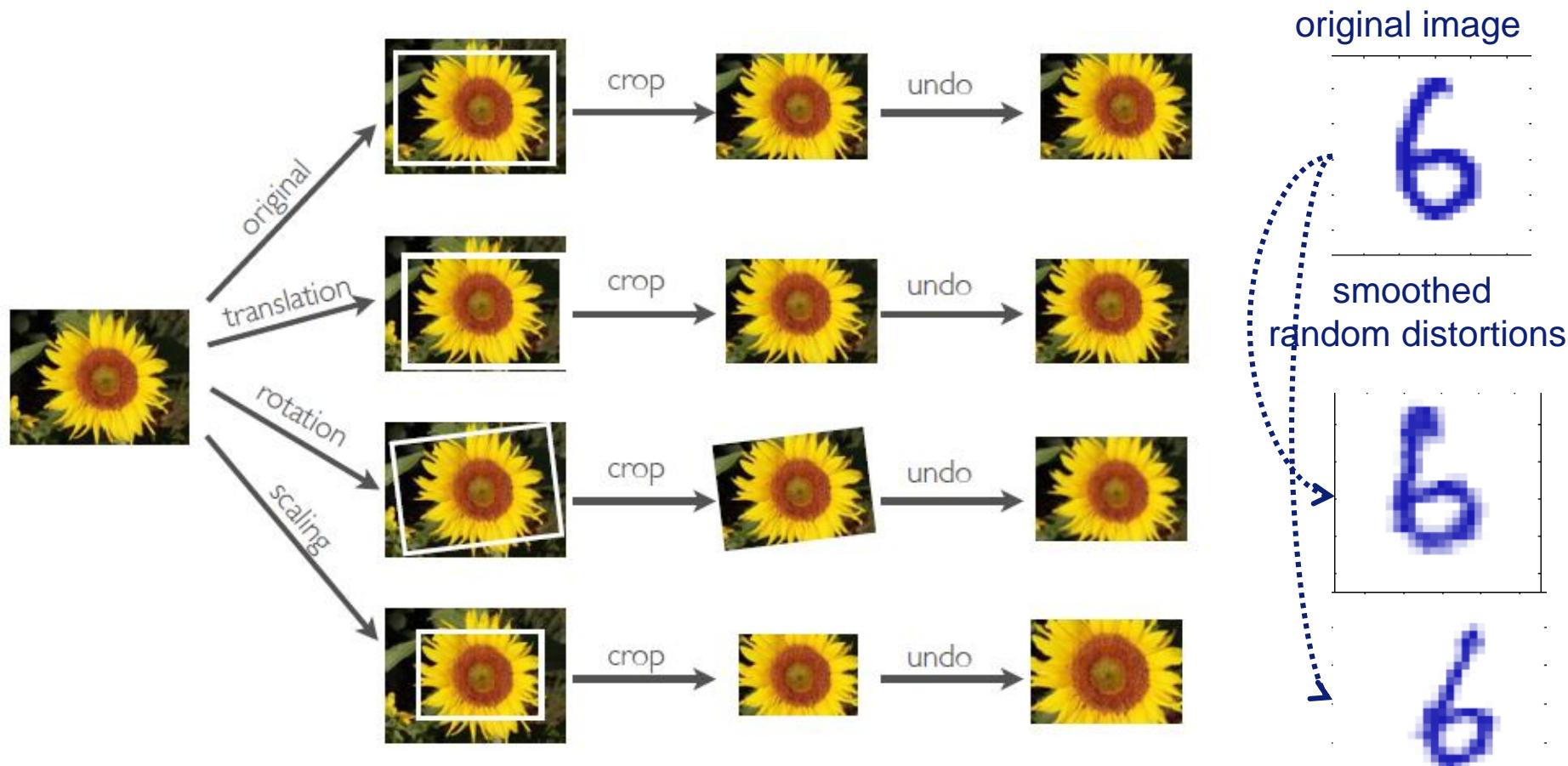
- **Representative** of the data / problem
- **Consistent** (similar input patterns should lead to similar outputs)
- **Sufficient** (learning algorithms are „data hungry!“)

Scaling of training data:

- If there is strong variation in the data range of the different input nodes: learning behaviour might be dominated by the node with the largest inputs
- Potential solution: **Normalization** of input values
 - Linear transformation to standard interval (e.g. [0,1])
 - Normalization to mean 0 and standard deviation 1
- Additional „tricks“:
 - Data augmentation
 - Pre-training / fine-tuning

Training: Data augmentation

- Idea: Artificially generate additional training data with more variation (rotation, scale etc.), by applying some distortion to original training images
→ Neural network will learn to be invariant to such transformations



Training: Data augmentation

- Transformations must not conflict with the task to be solved!

Examples:

- Classification of images: transform images, keep labels
- Transformations depend on wanted invariance:
 - changing colors may help if colors shall not be classified
 - scaling may help if classification does not aim at size
 - rotation may help unless you transform '6' → '9' ...
 - adding noise may help (noise on inputs, hidden layer's output, weights...)
- Note: In segmentation problems, the labels (segmented images) have to be transformed as well!

Selecting training samples / class imbalance

- For optimal performance, classes generally should be *balanced*
 - i.e. similar number of samples for each class
- This is not always the case, e.g.
 - Classifying breast cancer: Many images without cancer
 - Lesion segmentation: Only few foreground (lesion) / many background voxel

Strategies:

- Restrict number of background samples to number of foreground samples
- Modify loss function, e.g. for binary targets $y^{(\mu)} \in \{0, 1\}$:
 - Weights sensitivity and specificity with „sensitivity ratio“ r

$$L(\mathbf{w}, y, \hat{y}) = r \frac{\sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)})^2 y^{(\mu)}}{\sum_{\mu=1}^p y^{(\mu)}} + (1 - r) \frac{\sum_{\mu=1}^p (\hat{y}(\mathbf{w}, \mathbf{x}^{(\mu)}) - y^{(\mu)})^2 (1 - y^{(\mu)})}{\sum_{\mu=1}^p (1 - y^{(\mu)})}$$

- In any case: Carefully select the background samples („hard-to-classify“)

Supervised pretraining / models to aid optimization

- Idea: Instead of directly training a large network, start with a **smaller network** (less layers) and **gradually increase the number of layers**
 - May be followed by **supervised fine-tuning** (joint optimization, full problem)
- Related approach: **Start with simpler models that are easier to optimize**
 - E.g. lower depth; when training the full model, the activations of hidden layers of the simpler model can become additional targets to guide optimization
- Alternative: **Linear paths or skip connection between layers**
 - to mitigate vanishing gradient problem

„It is more important to choose a model family that is easy to optimize than to use a powerful optimization algorithm“ (Goodfellow)

Unsupervised pre-training (1)

- Idea: Initialize hidden layers using unsupervised learning
 - Force network to represent latent structure of input distribution
 - Encourage hidden layers to encode that structure



Why is one a character image



and the other not?

character image

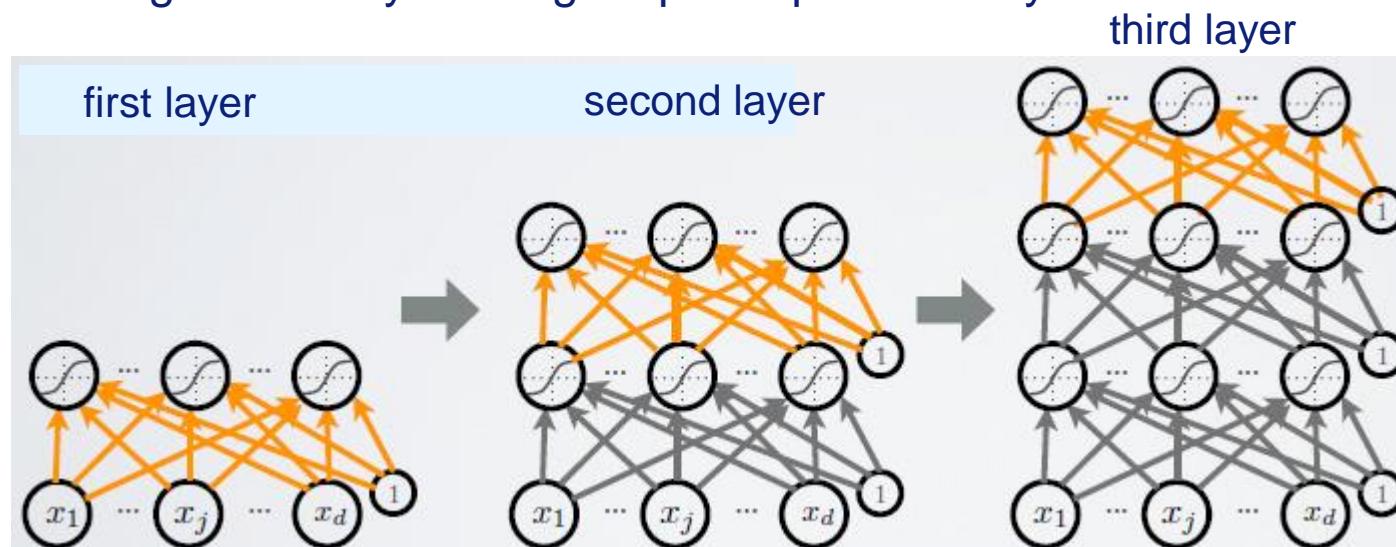


random image

- Harder task than supervised learning (classification)
→ Hence, we expect less overfitting

Unsupervised pre-training (2)

- Greedy, layer-wise procedure:
 - Train one layer at a time, from first to last, with **unsupervised learning**
 - Freeze parameters of previous layer (viewed as feature extraction); start training of next layer using output of previous layer



Find hidden unit features that are more common in training inputs than in random inputs

Find *combinations* of hidden unit features that are more common than in random inputs

Find *combinations of combinations* of ...

- Pre-training initializes the parameters in a region such that the near local optima overfit less the data

What kind of unsupervised learning?

- (Stacked) autoencoders
- (Stacked) denoising autoencoders
- (Stacked) restricted Boltzmann machines (RBM)
- And more:
 - Stacked semi-supervised embeddings
 - Stacked kernel PCA
 - Stacked independent subspace analysis...

Fine-tuning

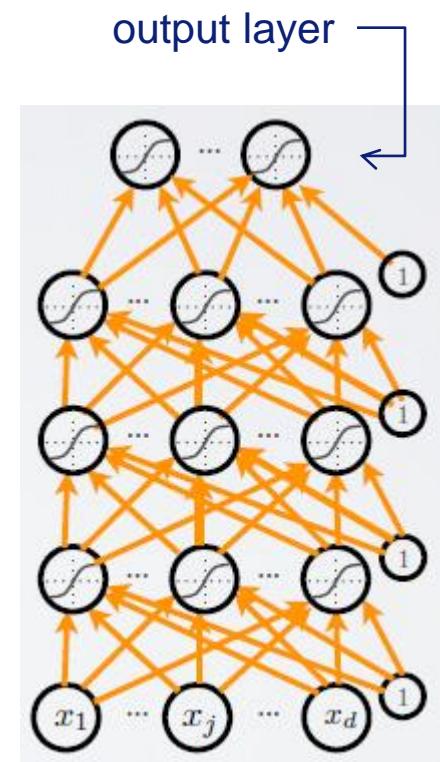
- Once all layers are pre-trained:
 - Add output layer (e.g. classifier)
 - Train the *whole network* using **supervised learning** (as in a regular feed-forward network):
 - Forward propagation, backpropagation, update

→ „fine-tuning“:

- All parameters „tuned“ for the supervised task at hand
- Representation is adjusted to be more discriminative

Advantages of pre-training / fine-tuning:

- Each layer gets full learning focus in its turn
- Can take advantage of the unlabeled data
- Fine-tuning starts from adjusted network weights (good „error basin“)



Neural network training: Where does generalization come from?

Some hypotheses (following Zhang et al., ICLR 2017):

- A low training error does *not* imply a low test error
 - Most deep neural networks can achieve a low training error even on random labels / random inputs, since # of parameters \gg # of data points
- Explicit or implicit regularization can improve the test error, but **are not the main reason to achieve good generalization**
 - Even without regularization, a low test error can often be achieved
 - This test error can often further be reduced by e.g. data augmentation, weight decay, batch normalization etc.
- Complexity measures from Statistical Learning Theory (e.g. VC dimens.) do not seem to explain the generalization properties of neural networks
 - Since deep networks can learn / memorize *any* training set (finite sample expressivity) if number of parameters exceeds number of data points
- Important: Relation between training and test input / label distributions (?)
- **Still insufficient insight into the generalization properties of deep nets!!!**

Parameters of a multi-layer perceptron

- Number and range of input values (real, binary)
- Number and range of output values (real, binary)
- Synaptic weights: range, initialisation
- Number of hidden layers and hidden neurons
- Parameters of learning algorithm (e.g. learning rate)
- Regularisation
- ...

Summary: For neural network training, take care of...

Network architecture:

- Number of hidden layers
- Number / range / type of inputs and outputs, number of hidden units
- Activation functions of hidden and output units
- Input normalization (zero centered, unit variance) / pre-processing

Training:

- Choice of loss function / regularization
- Choice of optimization algorithm (momentum, adaptive, second order) and required parameters (e.g. learning rate schedule, mini-batch size)
- Parameter initialisation (especially weights, biases less important)
- Further optimization strategies (batch normalization, dropout, data augmentation, supervised pretraining, start with simpler models...)
- Stopping criterion (early stopping)
- Hyperparameter search (cross-validation)

Learning in neural networks: Summary (1)

- Instead of *explicit* programming: *implicit* learning from examples
- Neural networks: Learning mostly refers to specifying the network parameters (synaptic weights, thresholds) in order to realise target outputs
 - Network topology and neuron model have to be pre-selected
- Learning paradigms:
 - **Supervised learning**: target output known and presented to network
 - **Unsupervised learning**: only input patterns presented to network
 - **Offline (batch) learning**: Weight update after *all* training samples
 - **Online (incremental) learning**: Weight update after every training sample
- Goal: correct network output for *new* examples: „**generalisation**“
 - Low training error does not necessarily imply low generalisation error: **overfitting**
 - Methods to reduce overfitting: train data selection, early stopping, regularization

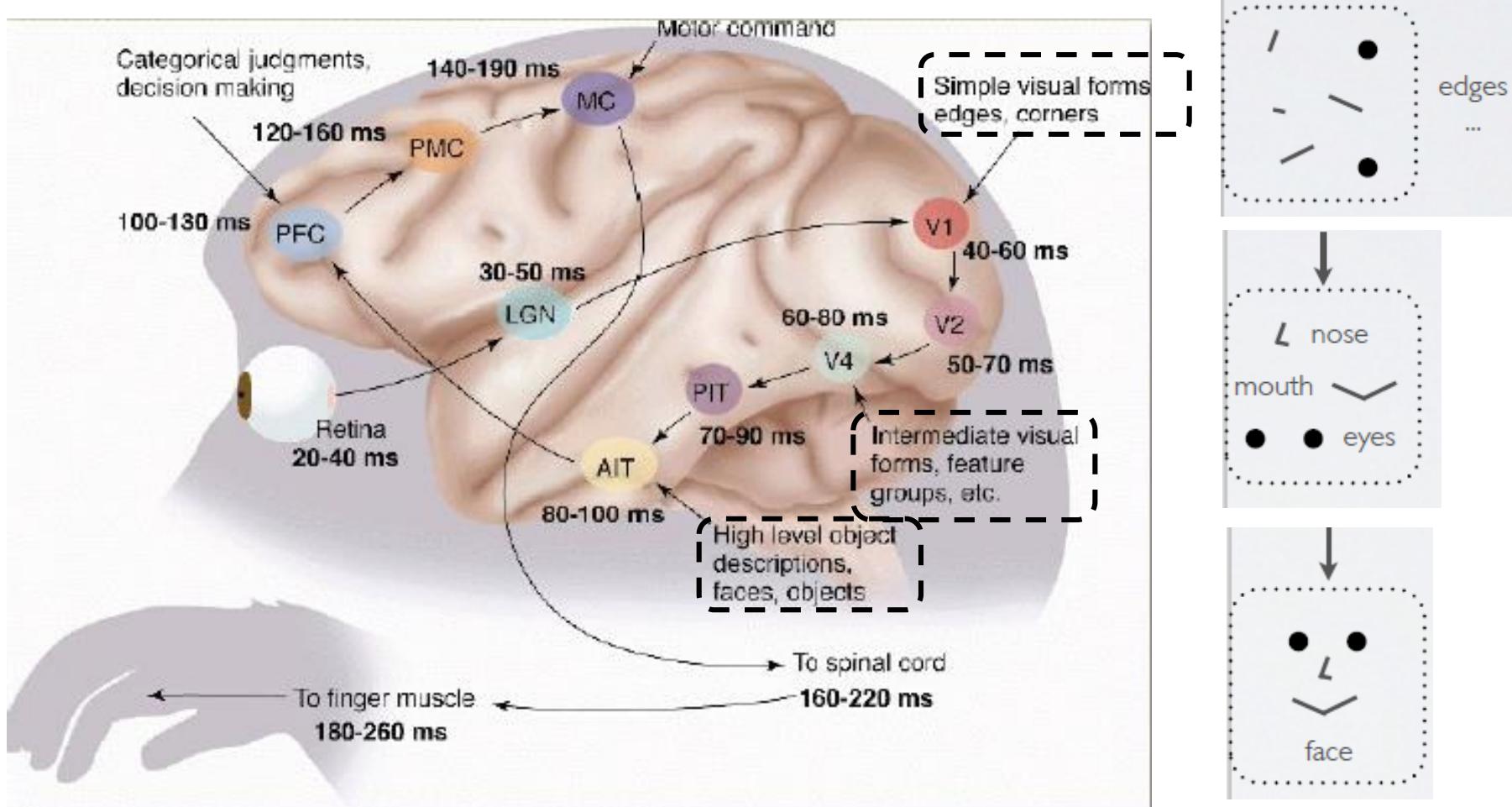
Learning in neural networks: Summary (2)

- (Single-layer) perceptron:
 - Perceptron learning: for threshold (binary) activation function
 - Gradient learning: for differentiable (e.g. sigmoid) activation function
- (Multi-layer) perceptron:
 - Backpropagation:
 - Forward step: computation of network output
 - Backward step: computation of error contribution of each neuron from error contribution of successor neurons → weight update
- Iterative algorithms; potential problems:
 - *local* minima, small gradients → very small steps, oscillations, initialisation
- Extensions:
 - Learning with momentum, adaptive learning rate, 2nd order methods
- Training data should be representative, consistent, concise
- Appropriate network topology has to be chosen in advance
- Avoid overfitting (early stopping, regularisation etc.)

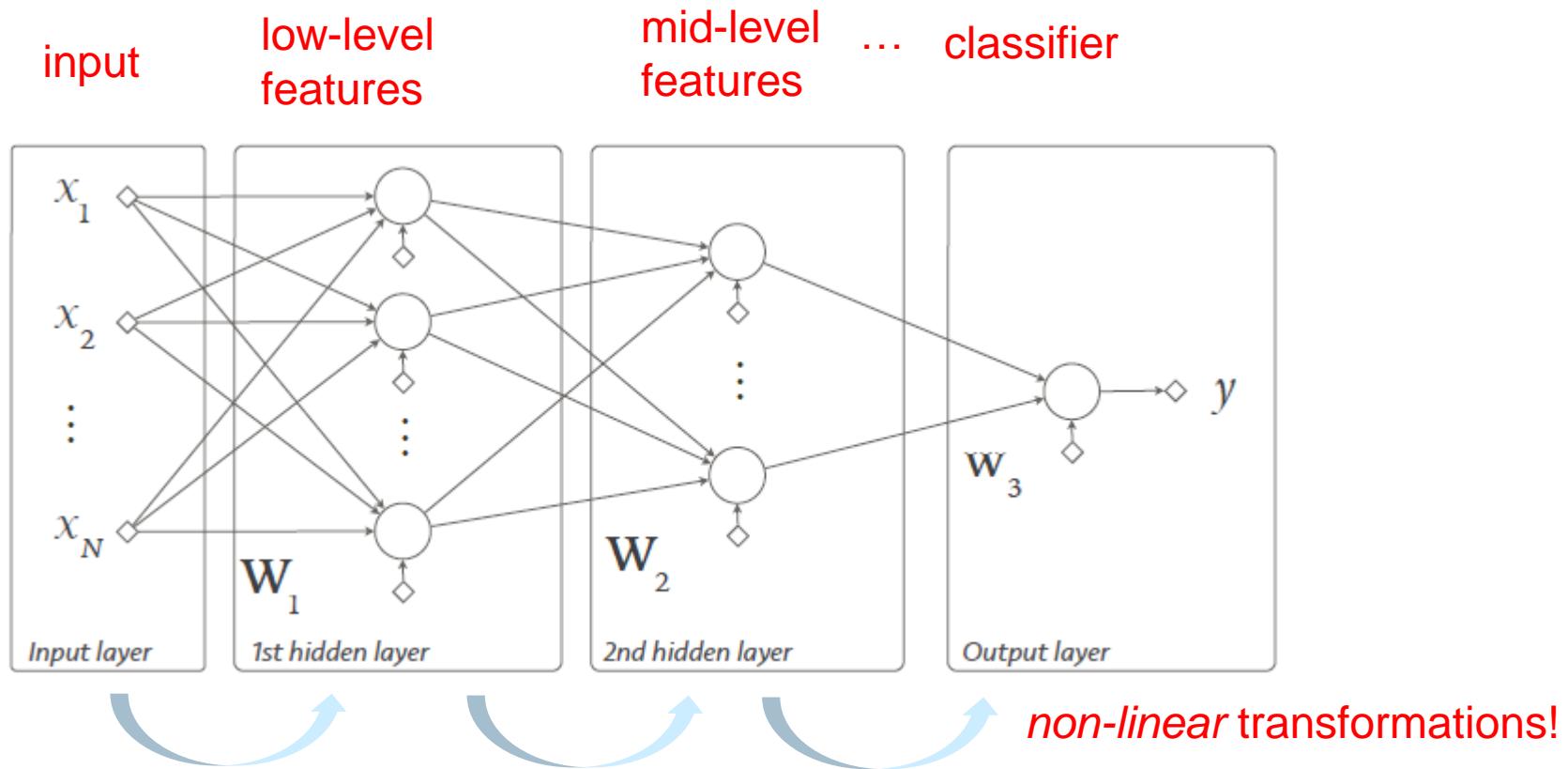
DEEP LEARNING

Towards deep learning: Inspiration from visual cortex

- Layer structure in visual processing with increasingly complex feature representation



So just use a multi-layer neural network with many layers ...?



- Idea: Subsequent layers learn increasingly „higher“ input representations („features of features of features...“)

→ „deep“ model

(see later for definition)

But: Why should we use more than a few layers?

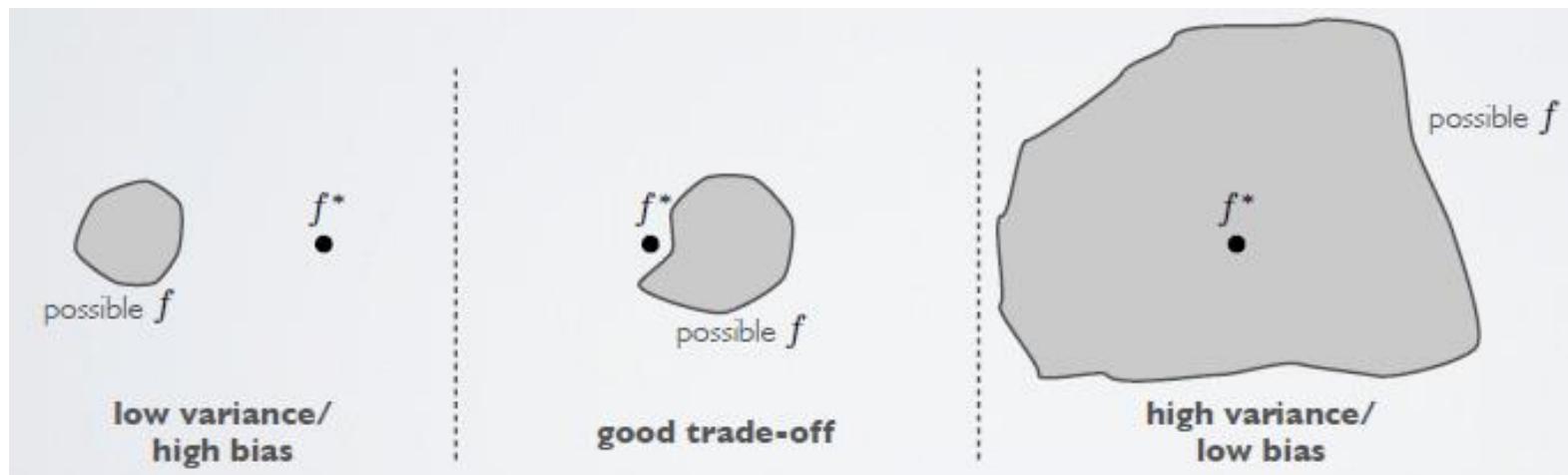
- We've seen that 3 layers are enough for anything... – why using more?
 - The brain has a deep architecture (see before)
 - Architectures with multiple levels naturally provide *sharing* and *re-use* of components
 - A deep architecture can represent certain functions (exponentially) more compactly

Example: Boolean functions

- Any Boolean funct'n can be represented by a network with **one hidden layer**
 - This may however require an **exponential** number of hidden units
- It can be shown that there are Boolean functions which
 - Require an **exponential** number of hidden units in case of a single hidden layer
 - Require a **polynomial** number of hidden units if the **number of layers** is adapted

Using more layers – is it that easy? (1)

- Deep architectures have **lots of parameters** (millions to billions!)
 - We are exploring a space of complex functions
 - **Lots of local minima** (even for shallow networks this might be a problem...)
 - **overfitting** (parameters tuned too much to the data)
 - high variance / low bias situation



- The learning time does not scale well
 - It is very slow in networks with multiple hidden layers
- It requires labeled training data
 - Almost all data is unlabeled

From: Illinois, Larochelle

263

Using more layers – is it that easy? (2)

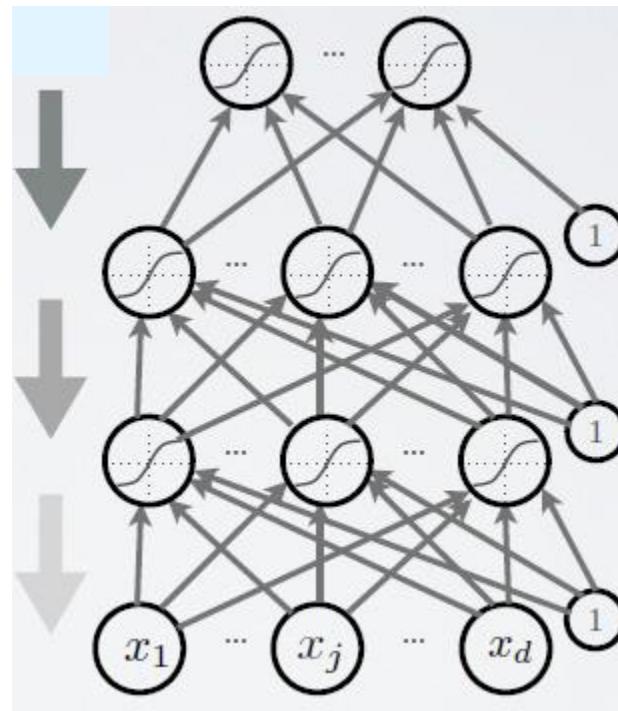
- Remember: Through many layers gradient may become too small / large
 - E.g. saturated units block gradient propagation („vanishing“ gradient)
 - we can't propagate errors too far back
 - **underfitting** (since the true parameter configuration is not found)

Vanishing gradients

large gradient

smaller gradient

even smaller /
vanishing gradient



(schematic representation!)

Exploding gradients

Small gradient

Larger gradient

even larger /
exploding gradient

Using more layers – solution attempts

- Deep architectures have lots of parameters (millions to billions!)
→ overfitting (parameters tuned too much to the data)

→ Use better regularization

- Parameter regularization, dropout
- Initialization of layers with unsupervised pre-training

- Through many layers gradient may become too small
→ underfitting (since the true parameter configuration is not found)

→ Use better optimization and suitable parameter initialization

- The learning time does not scale well

→ Use GPUs and more computer networks

- It requires labeled training data

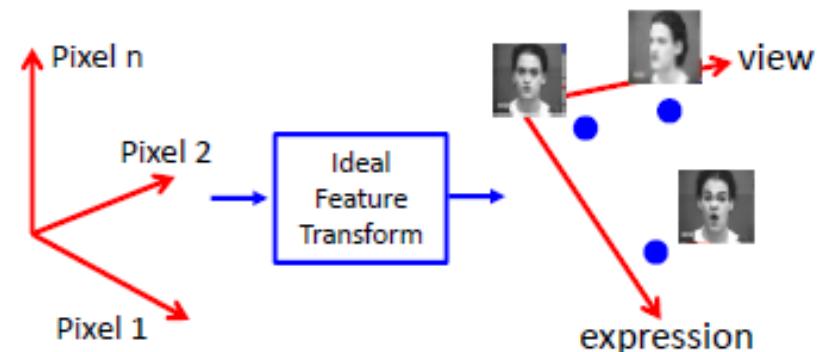
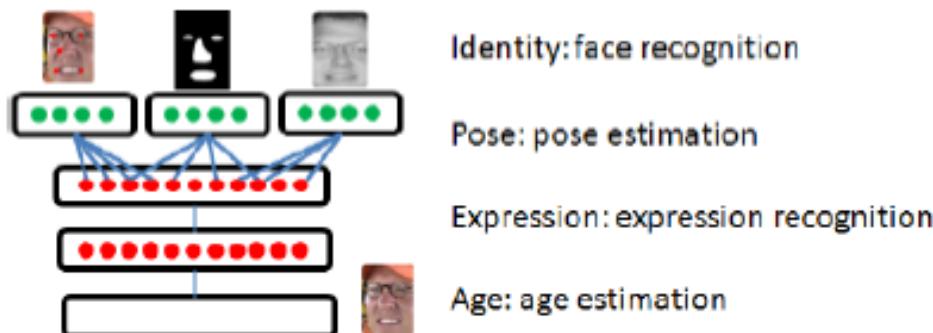
→ Unsupervised pre-training (see above), data augmentation

From: Larochelle

265

What is deep learning?

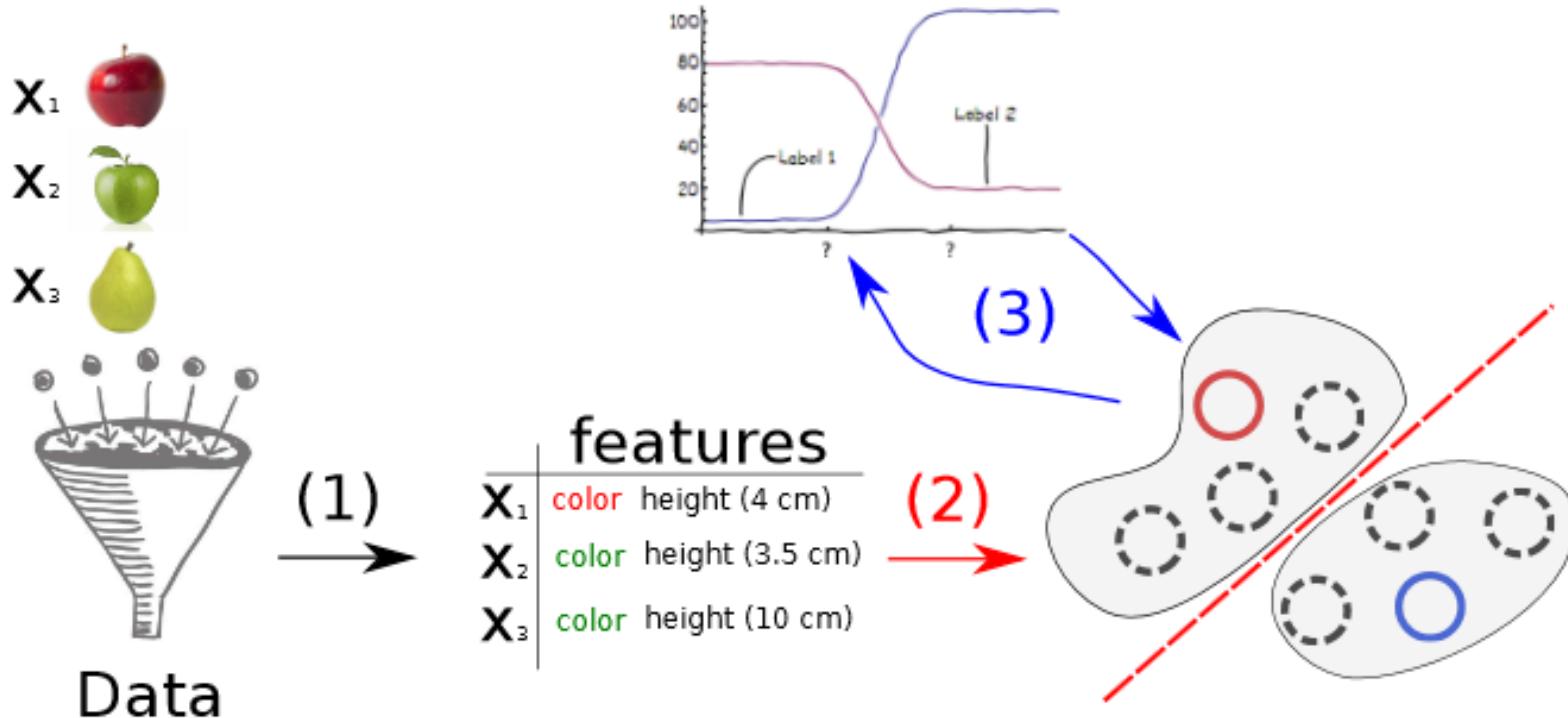
- Deep learning is research on learning models with multi-layer representations
 - Multi-layer (feed-forward) neural networks
 - Multi-layer graphical models (deep belief networks, deep Boltzmann machines)
- In this way, **hierarchical feature representations** are learned
 - Multiple levels of representation / abstraction
- Each layer corresponds to a „distributed“ feature representation
- Note: Good feature representations should be able to disentangle multiple factors coupled in the data



From: Larochelle, Wang

Standard machine learning pipeline ...

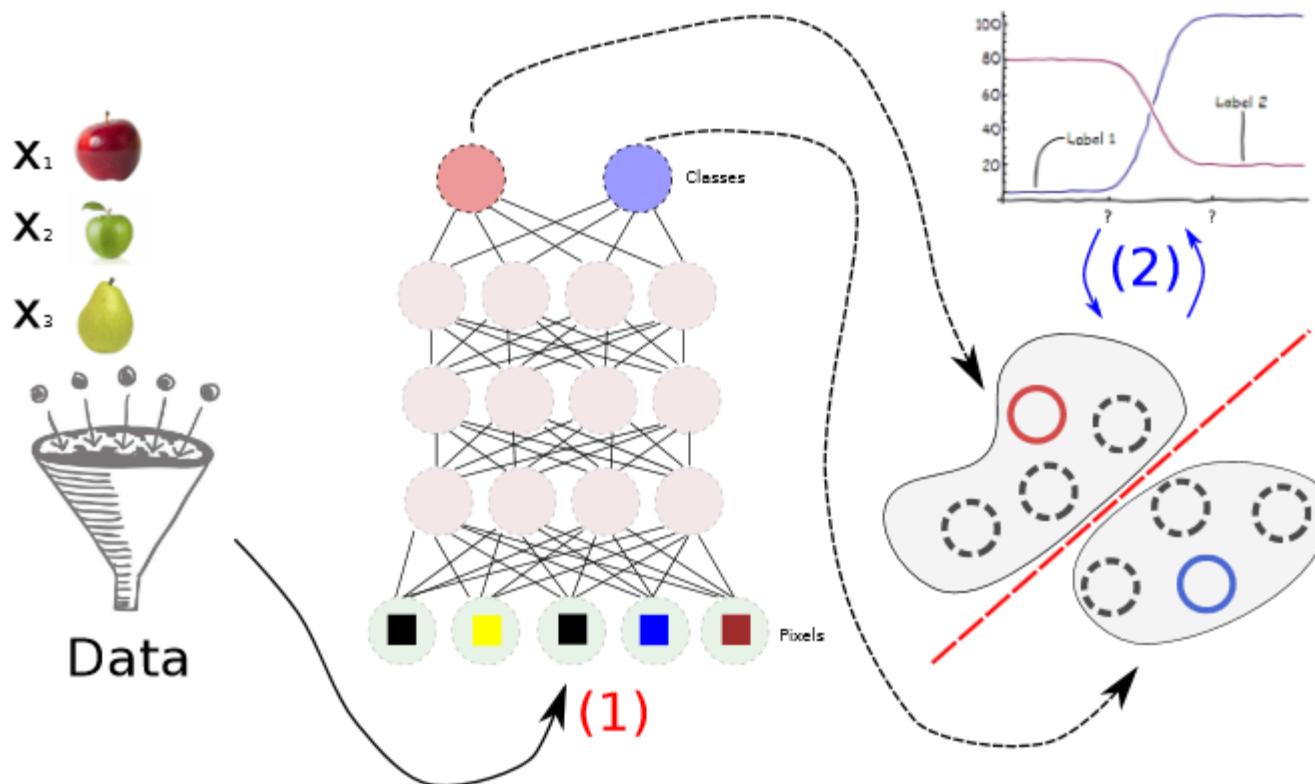
- (1) Engineer good features (*not learned*)
- (2) Learn model
- (3) Inference e.g. classes of unseen data



Separation between domain knowledge (features) and general machine learning; preprocessing and feature design not part of an end-to-end learning system

... versus deep learning

- (1) Jointly **learn** everything with a deep architecture
- (2) **Inference** e.g. classes of unseen data



Hierarchical feature transformations are *jointly* learned with classifier (part of an end-to-end learning system); „The data decides“ (Yoshua Bengio)

From: Riedmiller

Deep learning: Why now and not in the 80s?

Major success factors:

- Exponential growth of amount of
 - (labeled) data
 - computational power
- The ability to train very large neural networks fast by using GPUs
- Better designs for modeling and training
 - Input normalization
 - Using unsupervised pre-training and fine-tuning
 - Better regularization, e.g. by dropout
 - Use special (rectified linear) activation units that make training faster
 - Optimizing large networks (non-convex optimization): Not so scary as thought (but still not enough theory)

Software frameworks for deep learning

- Tensorflow
 - <https://www.tensorflow.org>
- Theano
 - <http://www.deeplearning.net/software/theano>
- Torch
 - <http://torch.ch>
- Caffe
 - C++ library, hooks from Python -> notebooks
 - <http://caffe.berkeleyvision.org>
- Keras
 - Python front-end APIs mapped either on Tensorflow or Theano back-end
 - <https://keras.io>
- Lasagne
 - Lightweight library to build and train neural nets in Theano
 - <http://lasagne.readthedocs.io>
- CNTK
 - <https://github.com/Microsoft/CNTK>

Deep learning: Summary

- Deep learning: Learning models with multi-layer representations
 - Idea: Learn distributed, hierarchical feature representation
 - Multiple levels of representation / abstraction
 - Instead of feature engineering
 - Major success factors:
 - Initialization of layers with unsupervised pre-training
 - E.g. via (denoising / sparse ...) autoencoders
 - Recent research: Not absolutely necessary in every task
 - Dropout
 - (Supervised) fine-tuning
 - Better optimization
 - Using GPUs and more computing power
 - Using more data in training (potentially via training set expansion)
 - Achieved breakthrough in many application fields, e.g.
 - Automatic speech recognition
 - Natural language processing
 - Computer vision (convolutional neural networks)
- 
- better regularization

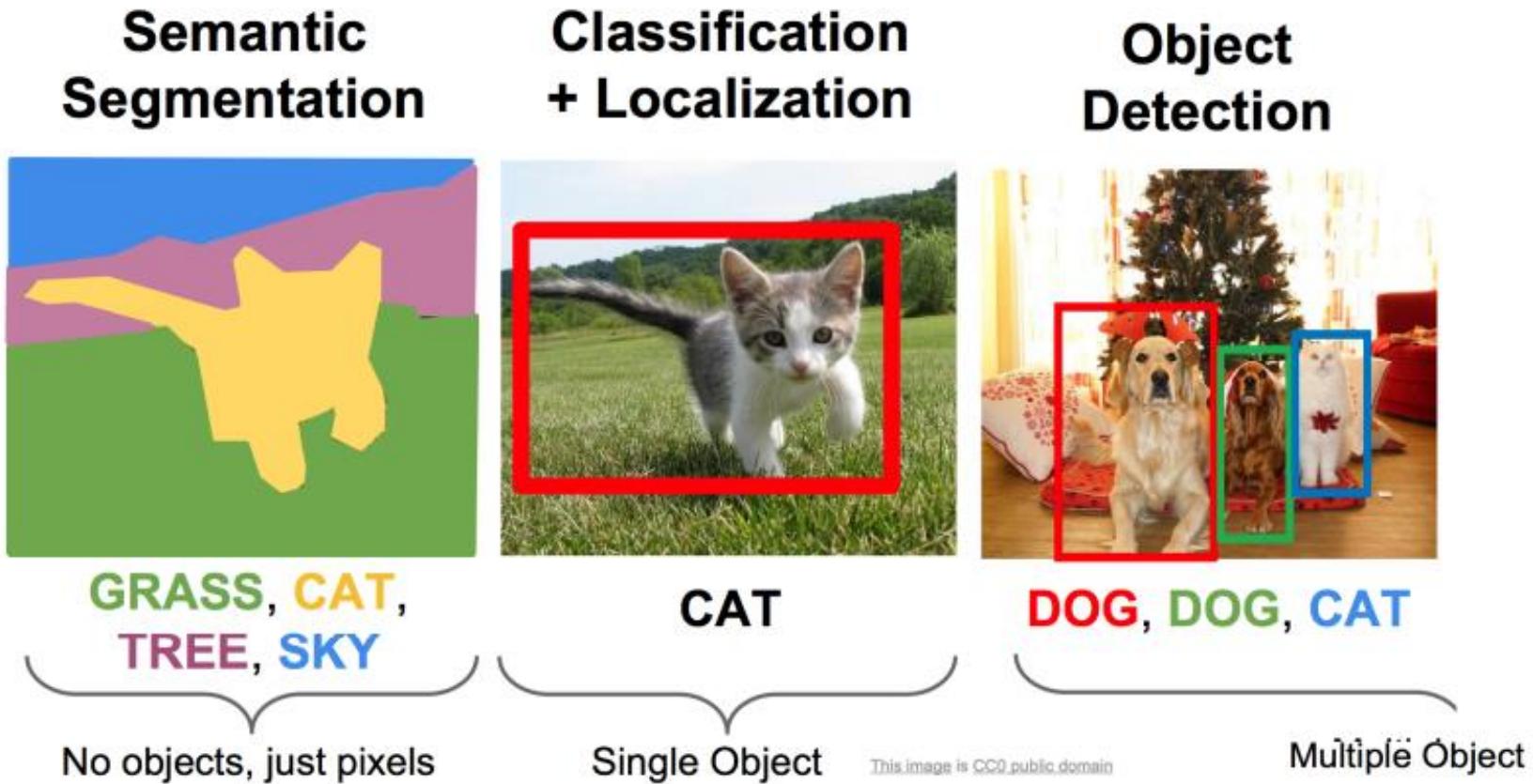
Further readings

- Aaron Courville, Hugo Larochelle: „A Deep learning tutorial“:
 - https://dl.dropboxusercontent.com/u/19557502/ecml-pkdd_slides.pdf
 - Video lectures:
<https://www.youtube.com/playlist?list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH>
- Yoshua Bengio: „Learning Deep Architectures for AI“
 - http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf
- Li Deng, Dong Yu: „Deep Learning: Methods and Applications“:
 - <http://research.microsoft.com/pubs/209355/DeepLearning-NowPublishing-Vol7-SIG-039.pdf>
- Stanford course on deep learning and unsupervised feature learning:
 - http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial
- and many tutorials from internet, e.g.
 - <http://deeplearning.net/>
 - <http://neuralnetworksanddeeplearning.com/>

CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks: Motivation

- Common computer vision tasks:

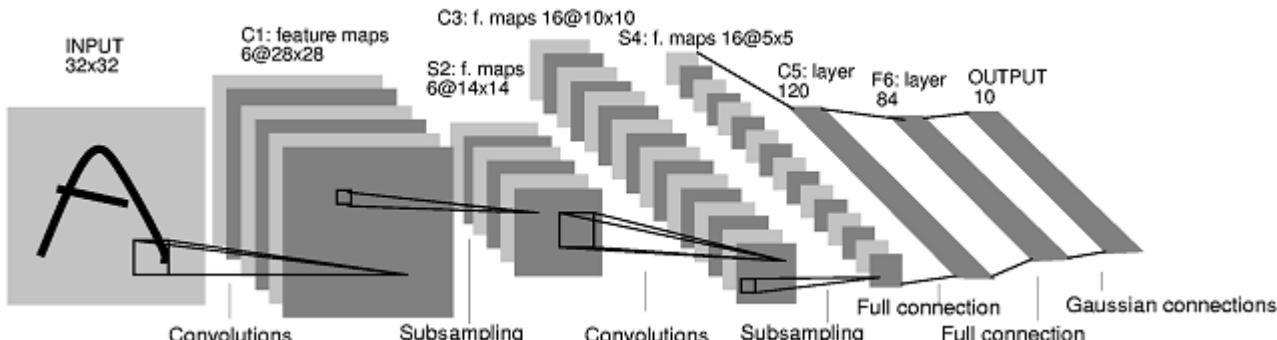


→ Can all be solved using some form of (deep) convolutional neural network

Convolutional neural networks: Introduction

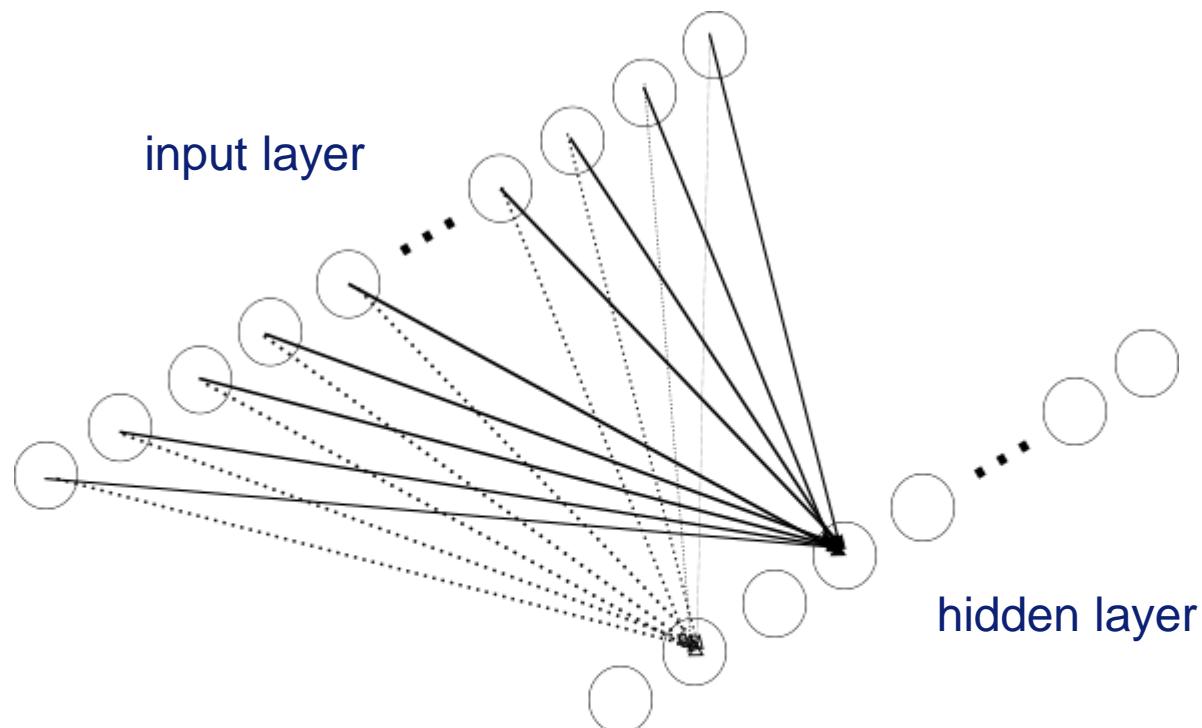
Too many parameters!

- Example: Computer vision
 - E.g. $10^3 \times 10^3$ input pixel; 10^6 hidden units, fully connected $\rightarrow 10^{12}$ params
- Key ideas to design neural networks adapted to such problems:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space
 - Pooling / subsampling hidden units
- Convolutional neural network (CNN):
 - Neural network with convolution layers
 - Special type of feedforward neural network
- Deep convolutional network:
 - Convolutional neural network with multiple convolution layers



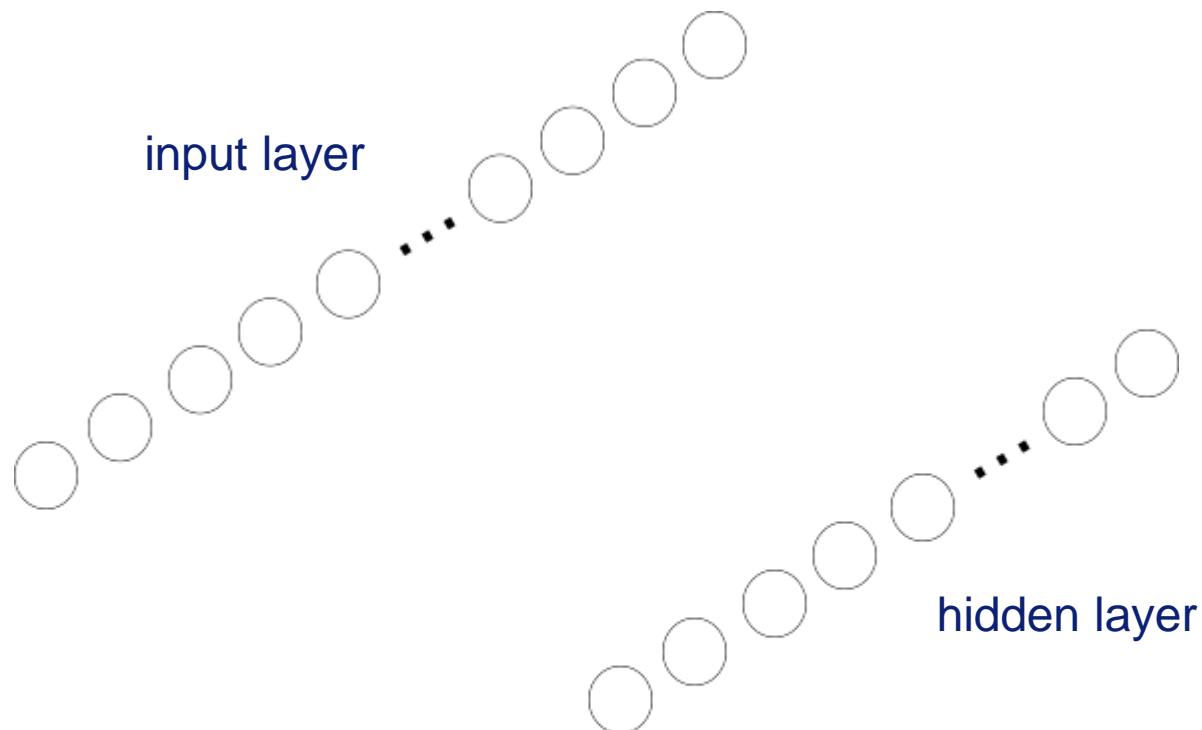
Convolutional neural networks: Introduction

multi-layer perceptron



Convolutional neural networks: Introduction

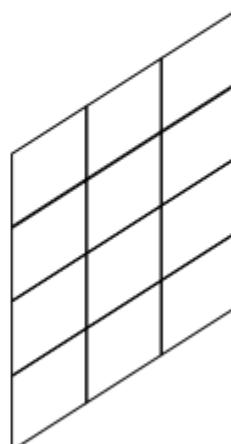
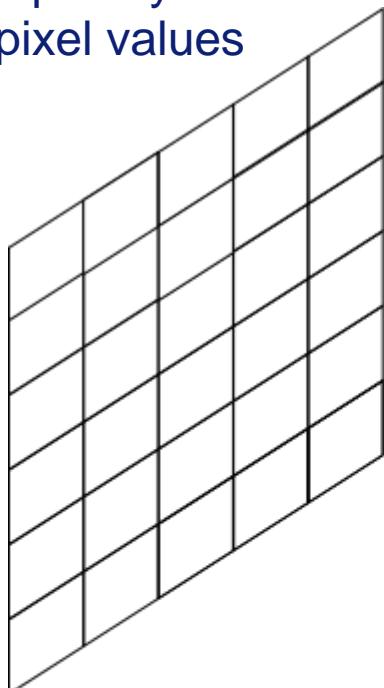
multi-layer perceptron



Convolutional neural networks: Introduction

Re-arrange neurons in form of two-dimensional arrays (images)
(representing pixel values in input layer)

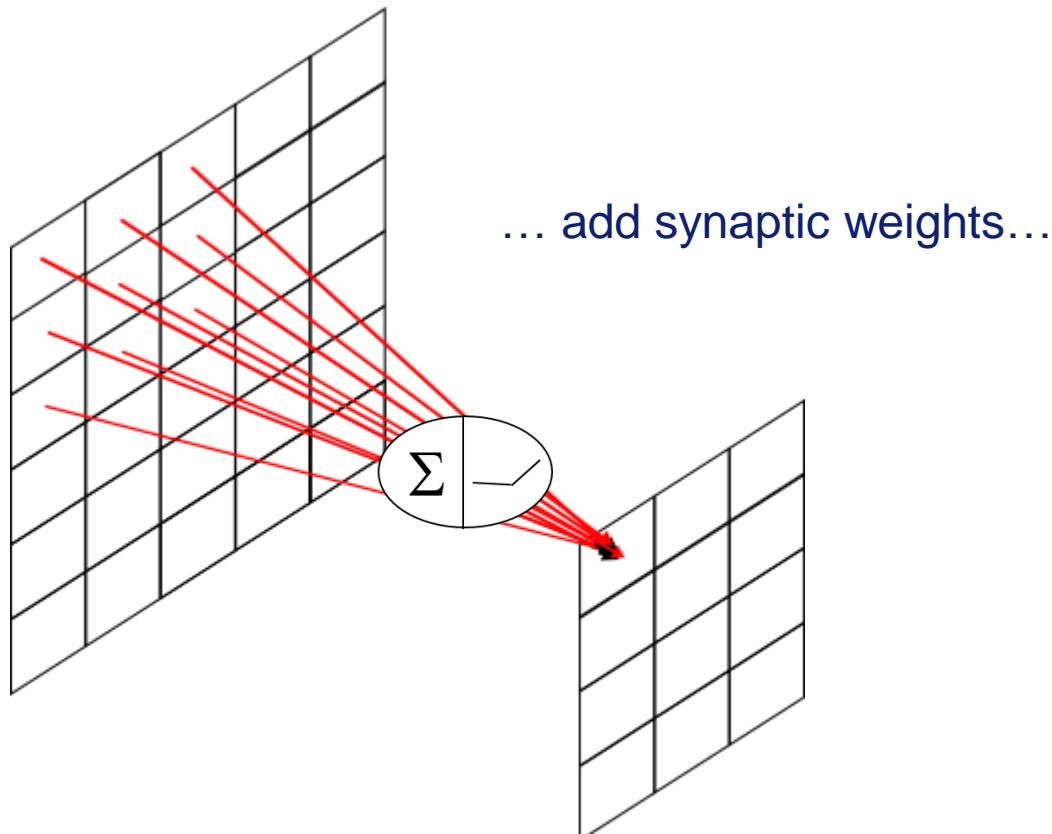
input layer:
pixel values



hidden layer

Convolutional neural networks: Introduction

Re-arrange neurons in form of two-dimensional arrays (images)
(representing pixel values in input layer)

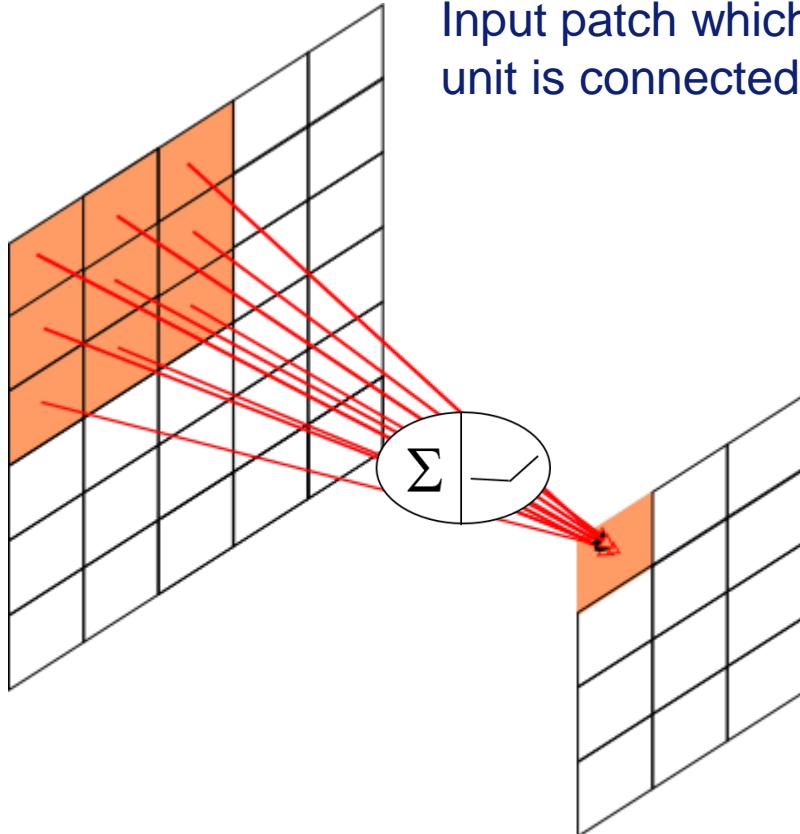


Convolutional neural networks: Introduction

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch

Receptive field:

Input patch which the hidden unit is connected to (orange in input layer)



Example weights:

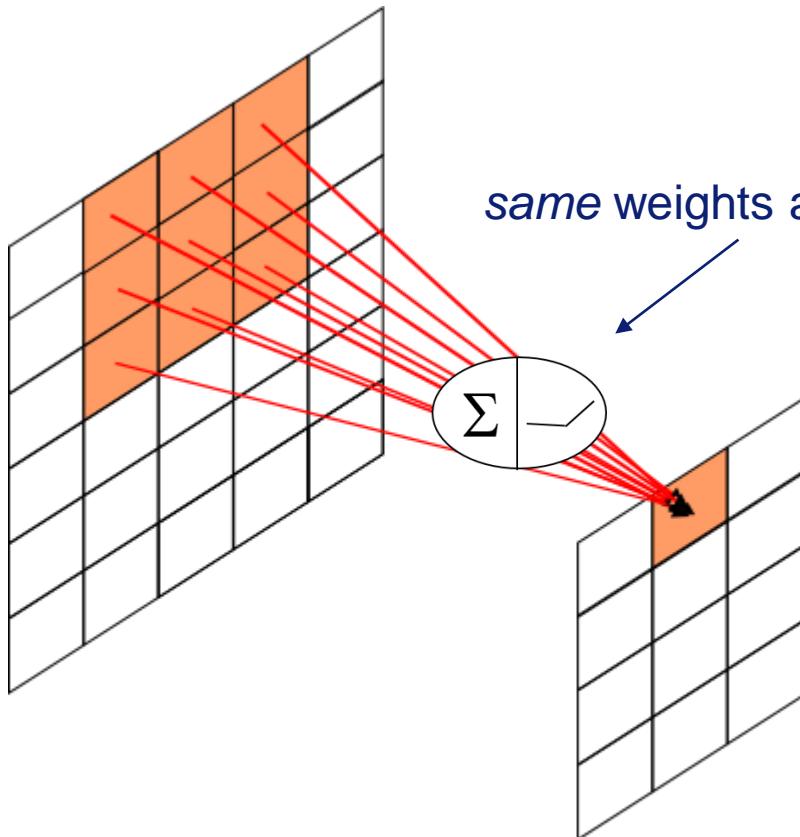
$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Output: Activation computed by

$$y = f(\mathbf{w} \cdot \mathbf{x} - \theta)$$

Convolutional neural networks: Introduction

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space
- Much less parameters than in a multi-layer perceptron!



Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

same weights w , different location x

$$y = f(w \cdot x - \theta)$$

What is a convolution?

- Continuous **input x** and **kernel w** : $s(t) = (x * w)(t) = \int x(a)w(t - a)da$
- Discrete t : $s(t) = (x * w)(t) = \sum_a x(a)w(t - a)$
 - x and w are only defined on integer t
- CNN terminology: s is referred to as **feature map**
- For a 2D image $I(m,n)$ as input:

$$s(i,j) = (I * w)(i,j) = \sum_m \sum_n I(m,n)w(i - m, j - n)$$

Commutativity

$$s(i,j) = (I * w)(i,j) = \sum_m \sum_n I(i - m, j - n)w(m,n) = (w * I)(i,j)$$

- Implementations of convolutional networks use **cross-correlation** *

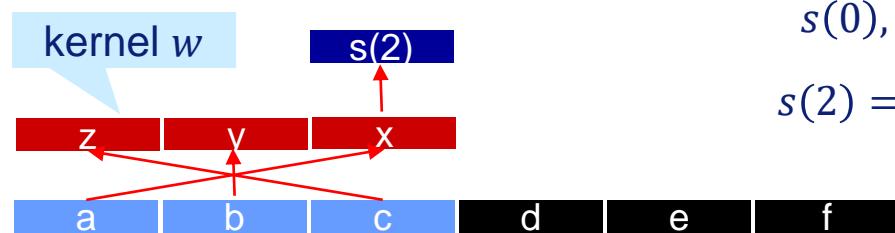
$$s(i,j) = (I \star w)(i,j) = \sum_m \sum_n I(i + m, j + n)w(m,n)$$

- Same as convolution with flipped kernel (at shifted output indices)

Convolution: 1-D illustration

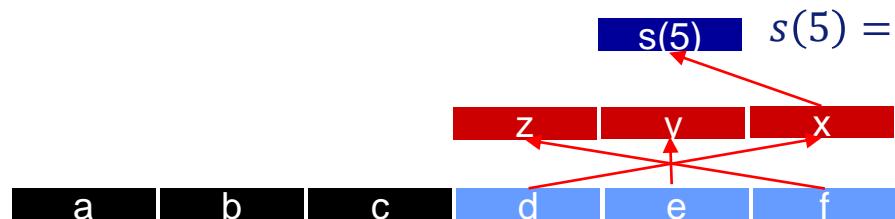
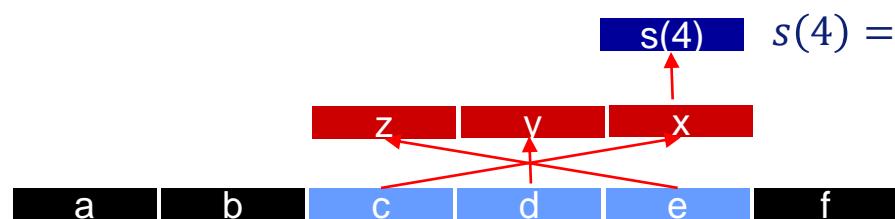
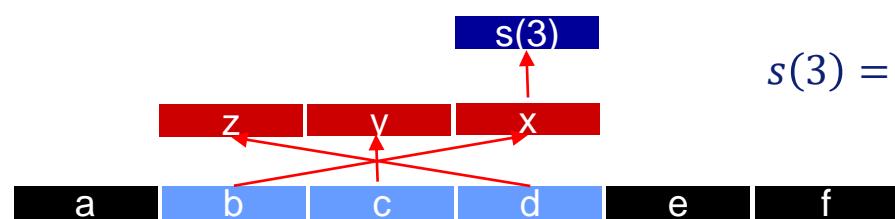
$$s(t) = (x * w)(t) = \sum_a x(t-a)w(a)$$

$x = [a, b, c, d, e, f], w = [z, y, x]$ (convolution)



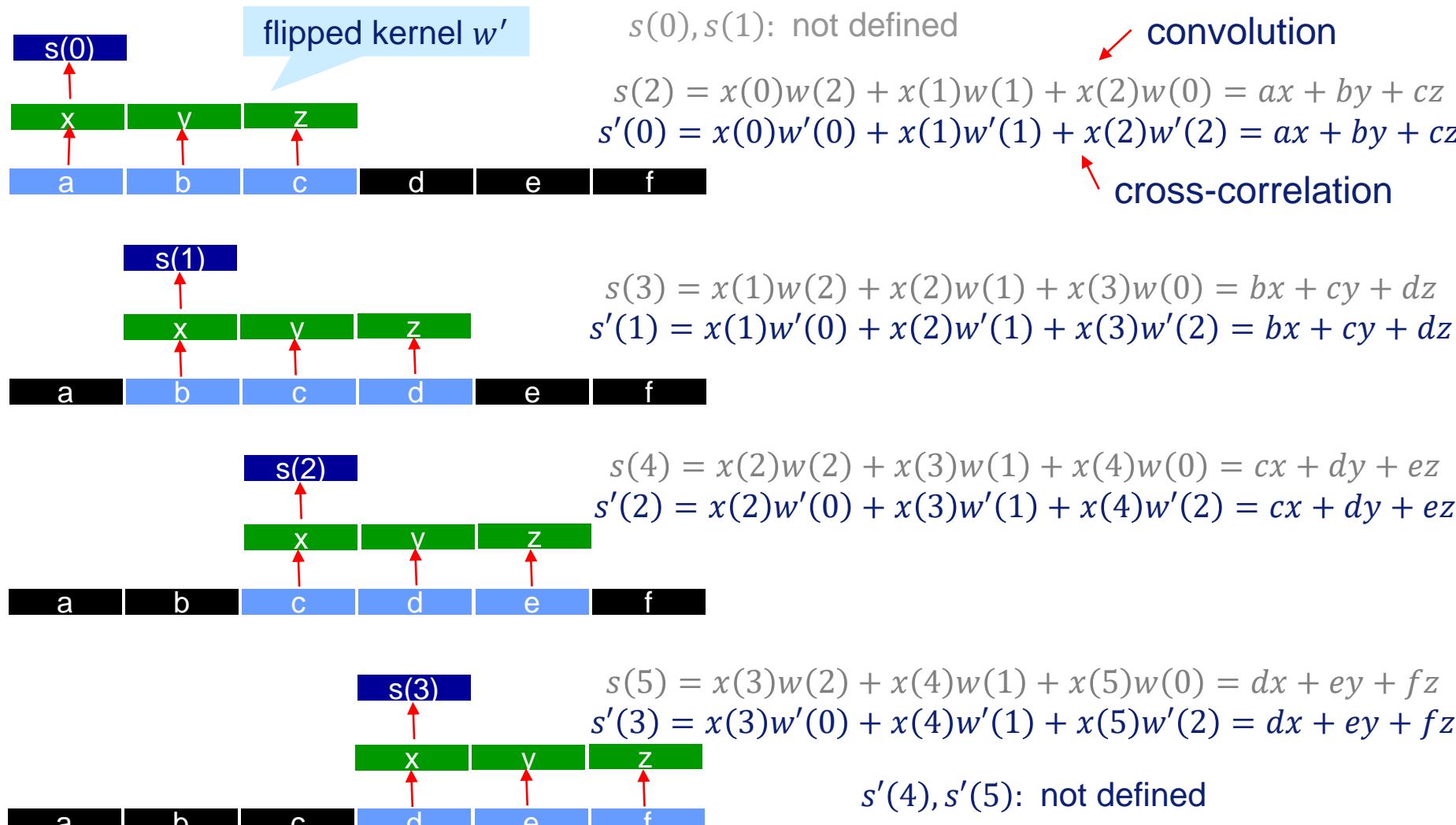
$s(0), s(1)$: not defined

convolution

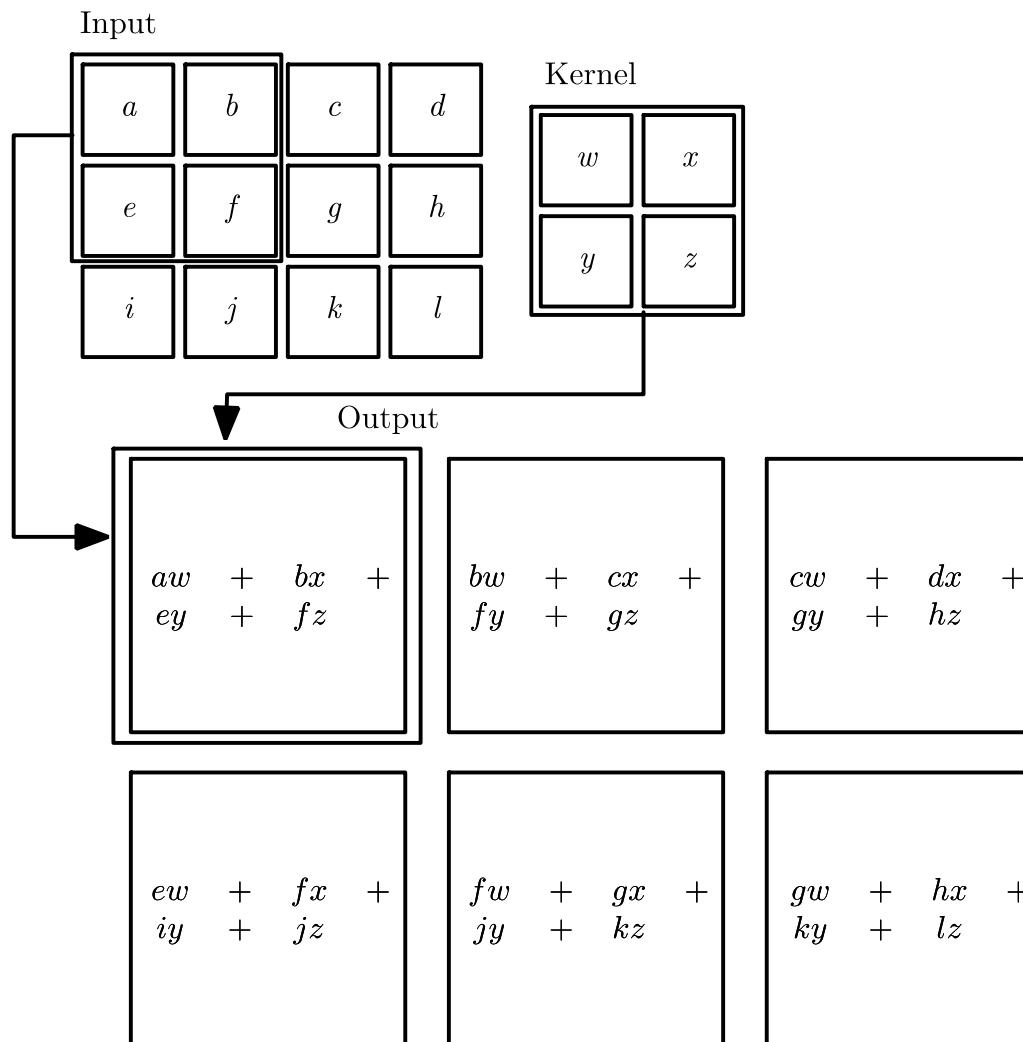


Cross-correlation: 1-D illustration

$x = [a, b, c, d, e, f], w = [z, y, x]$ (convolution), $w' = [x, y, z]$ (cross-correlation)



„Convolution“ (actually cross-correlation): 2-D Illustration



Kernel applied at all locations inside image;

here:

input: 3×4

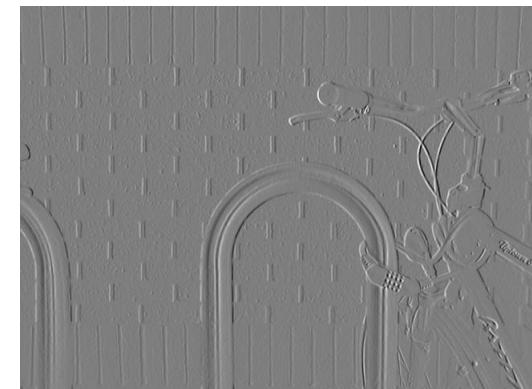
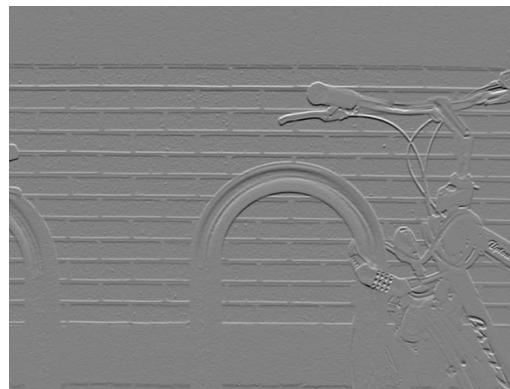
kernel: 2×2

→ output: 2×3

i.e. output feature map smaller than input image (depending on kernel)

Why convolution?

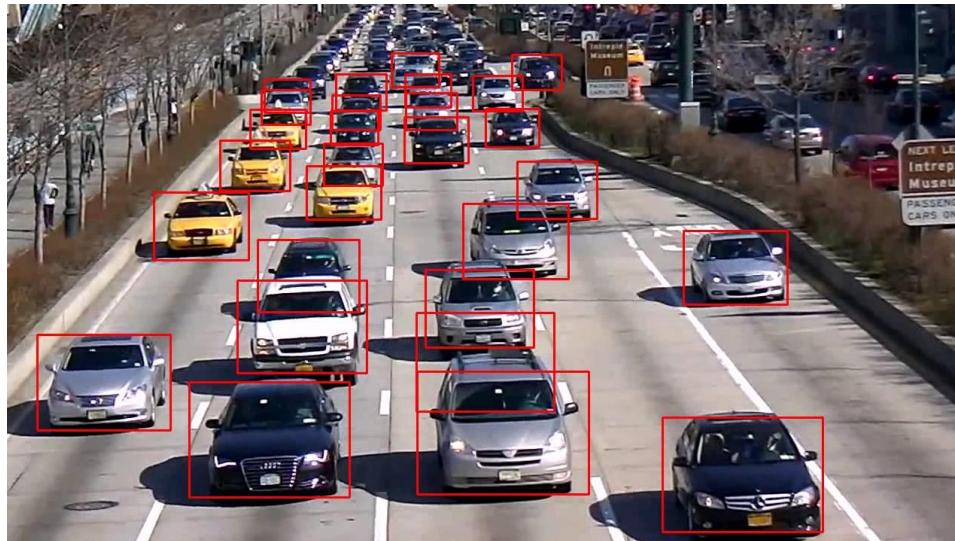
- 3 x 3 kernel (Sobel operator) $K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$, $K^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$
- Convolve input image with K : with K^T :



- Convolution with kernel (filter) K highlights horizontal edges
- Convolution with kernel (filter) K^T highlights vertical edges
- Different kernels amplify different image properties → multiple feature maps
- The same kernel is applied at all image locations (parameter sharing)
- Subsequent layers activate for specific (but more complex) patterns

Why convolution (2)? Equivariance

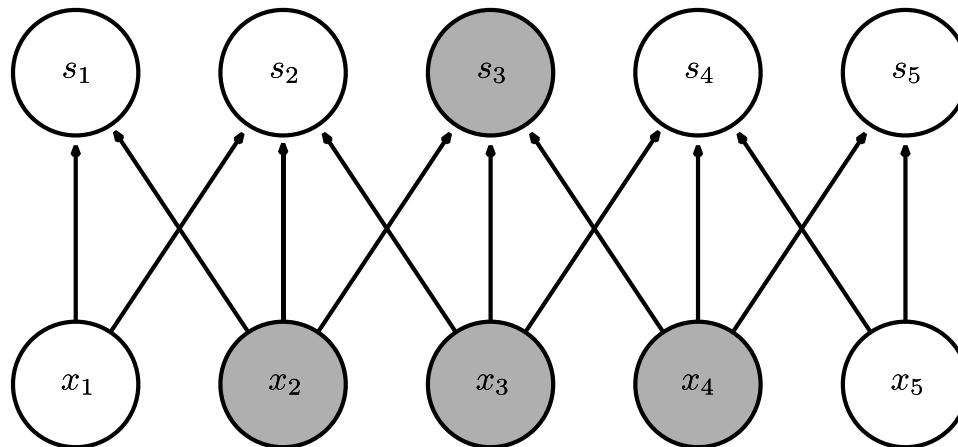
- A function $f(x)$ is equivariant if $f(g(x)) = g(f(x))$
- Example: $f(x)$ is convolution, $g(x)$ translation
- Convolution is equivariant to translations
 - Translation of object in image followed by convolution is same as convolution followed by translation of (convolved) object
- Necessary for images where objects can be anywhere within the image



<https://i.ytimg.com/vi/xVws19p3irA/maxresdefault.jpg>

Illustration of local connectivity (1)

Local connectivity
due to small kernel
("sparse connections")



„Dense connections“
(fully connected)

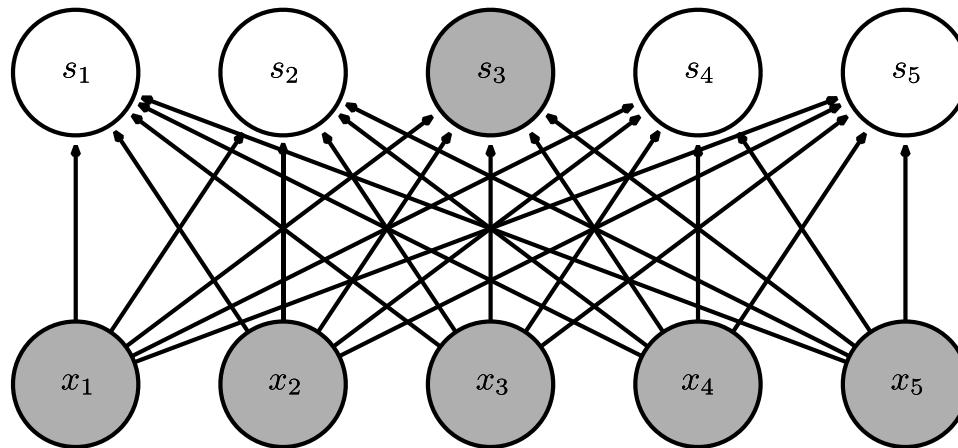
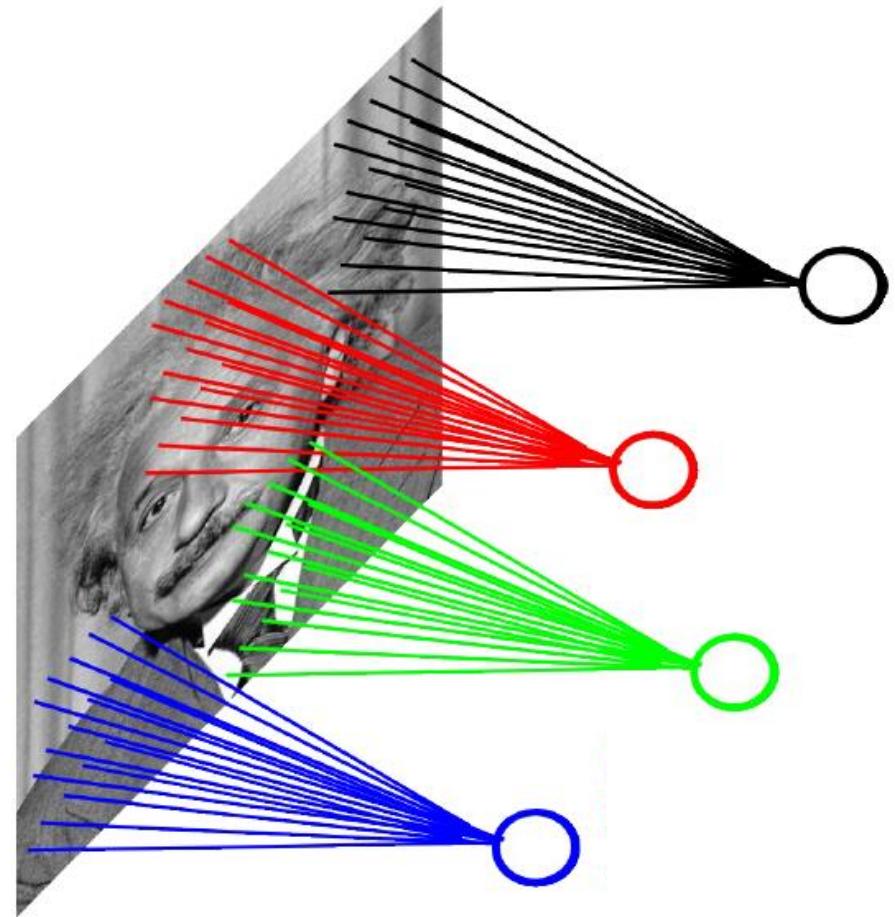


Illustration of local connectivity (2)

- Each hidden unit computes convolution with kernel at one image location
- Each hidden unit is connected only to subregion (patch) of input image
 - Separately for each image channel: 1 for grayscale, 3 (red, green, blue) for color



Example:

$10^3 \times 10^3$ input pixel

10^6 hidden units

Filter (patch) size: 10×10

→ 10^8 parameters

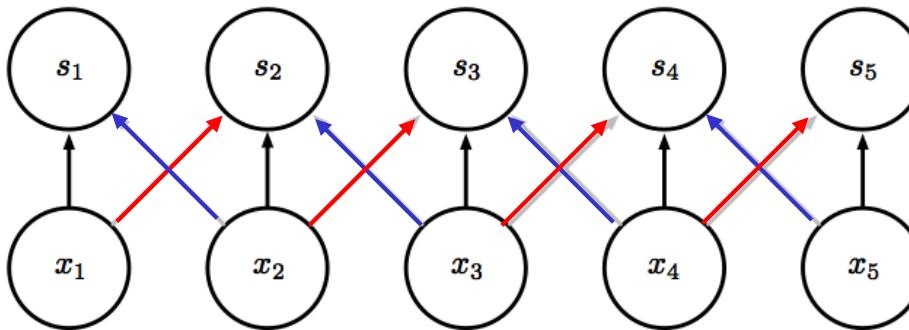
→ small kernels (instead of fully connected) lead to parameter reduction

Receptive field:

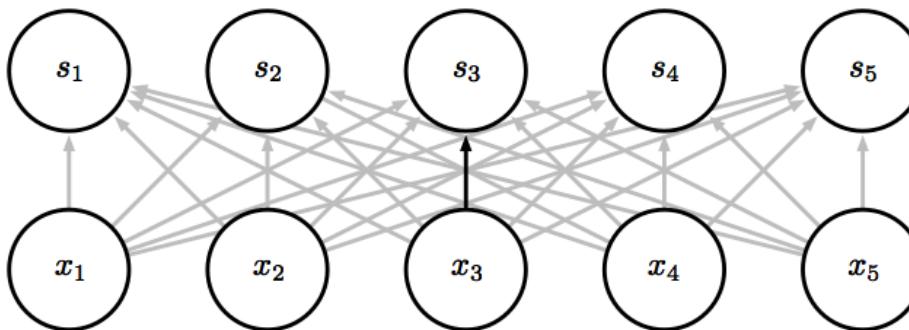
Input patch which the hidden unit is connected to

Illustration of parameter sharing (1)

Convolution
shares the same
parameters
across all spatial
locations



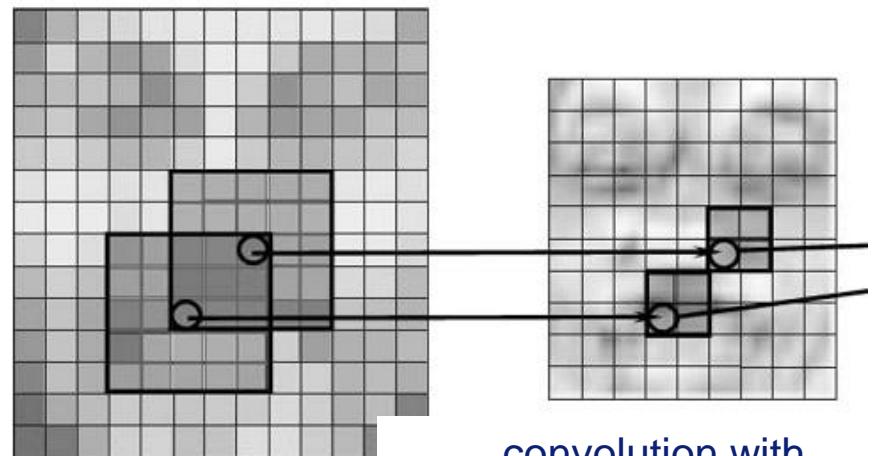
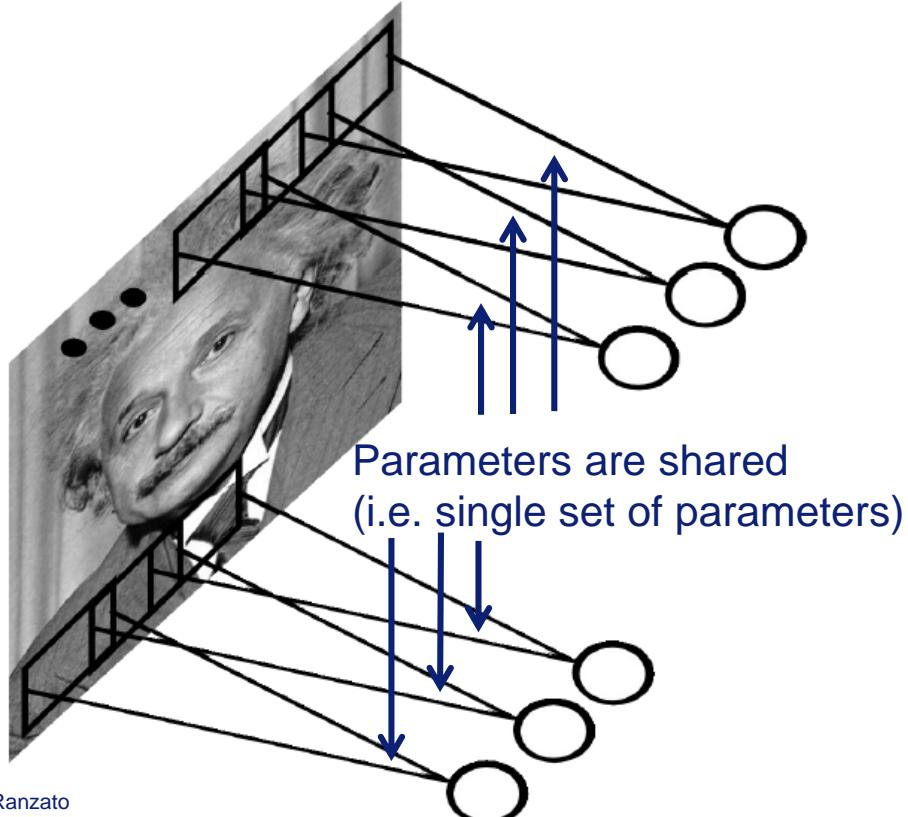
Traditional
matrix
multiplication
does not share
any parameters



- Local connectivity and parameter sharing → much less parameters
 - Reduces overfitting!
- However, input structure must be appropriate (e.g. images, spectrum, ...)

Illustration of parameter sharing (2)

- Share the kernel parameters (synaptic weights, biases) across the different spatial positions → same features extracted at every location
- Each hidden unit computes a **convolution** of the input patch with some kernel (given by the synaptic weight parameters and bias)
- „convolution layer“



Input image
(14×13)

convolution with
5 \times 5 kernel for all
possible positions in image
→ 10 \times 9 feature map

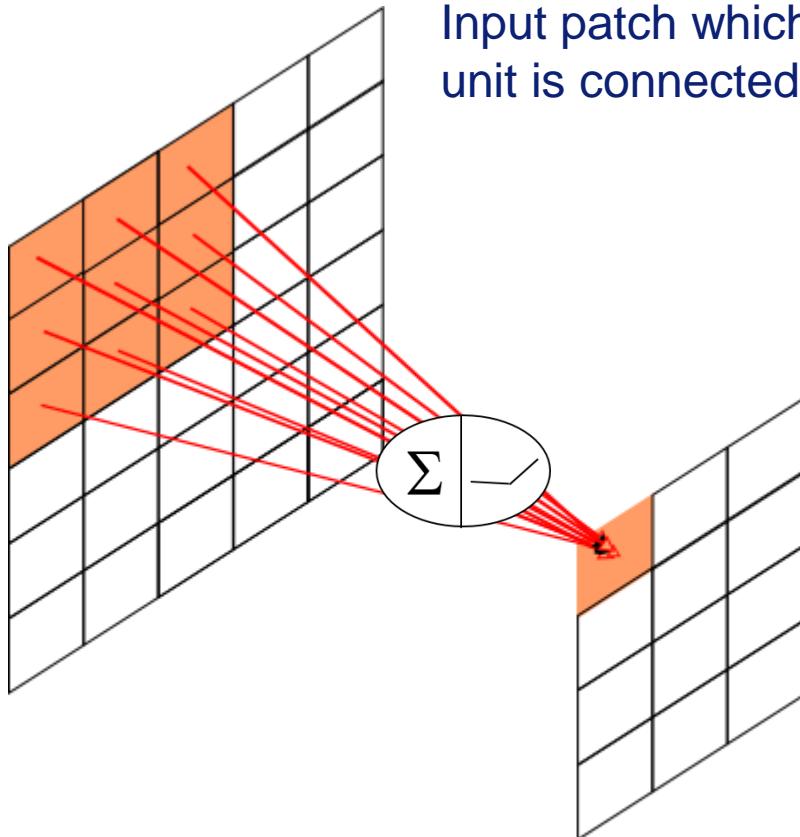
convolution layer in this example:
hidden layer with 10×9 units; each unit
computes convolution at some image position
(„feature map“)

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch

Receptive field:

Input patch which the hidden unit is connected to (orange in input layer)



Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Output: Activation computed by

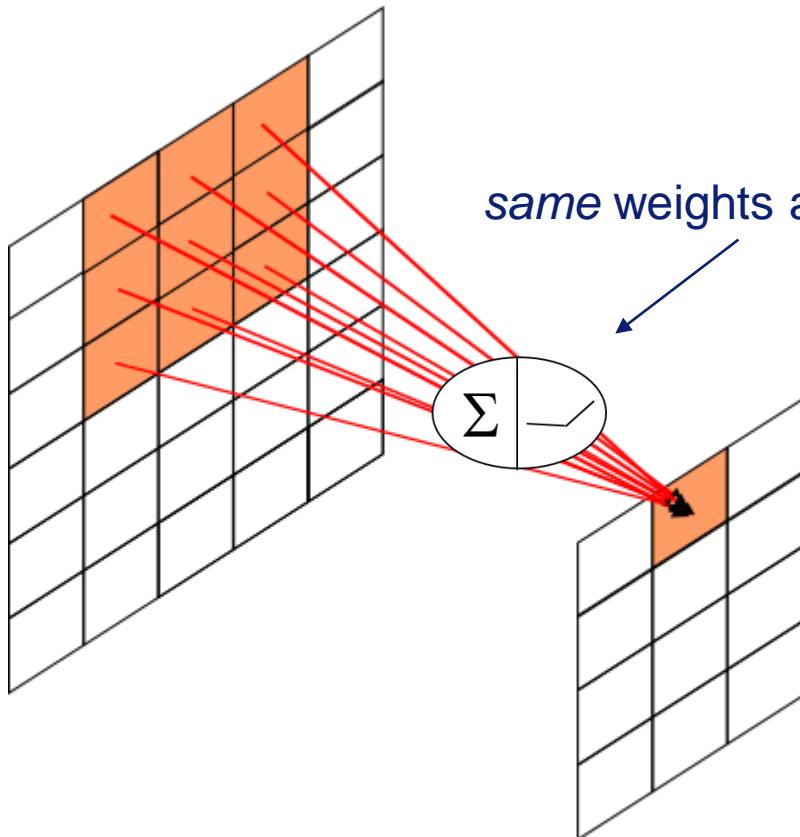
$$y = f(\mathbf{w} \cdot \mathbf{x} - \theta)$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:

- Local connectivity: Connect each hidden unit to a *small* input patch
- Parameter sharing: Share the synaptic weights across space

→ Much less parameters than
in a multi-layer perceptron!

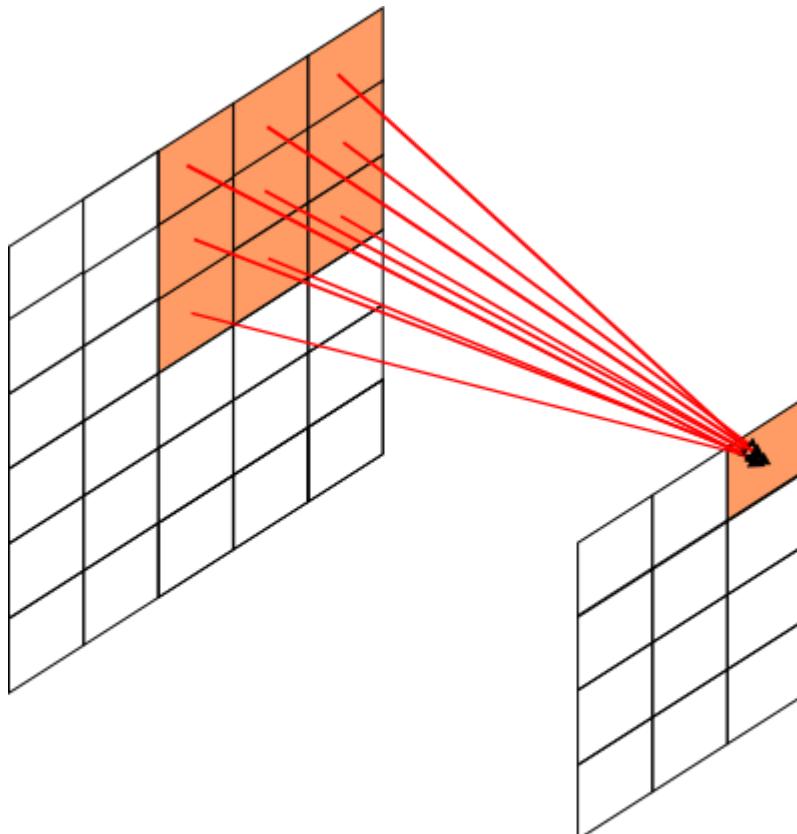


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

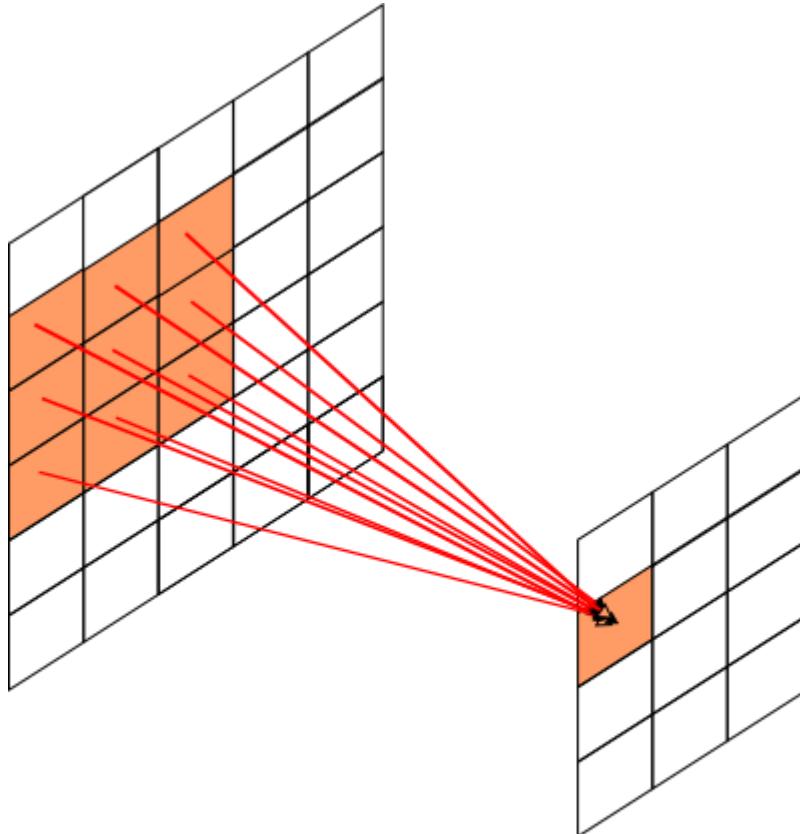


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

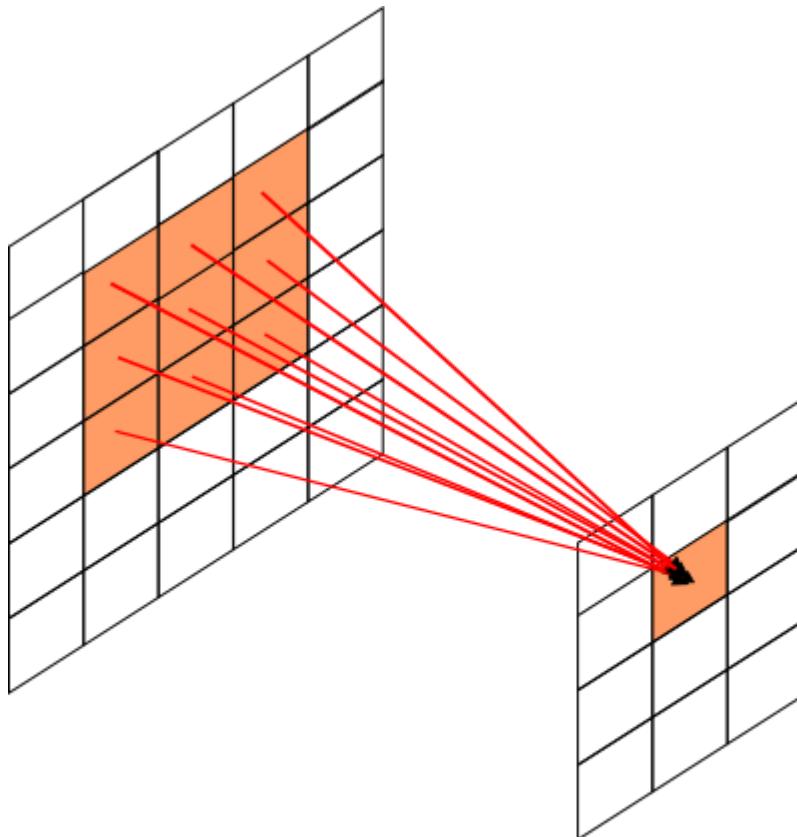


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

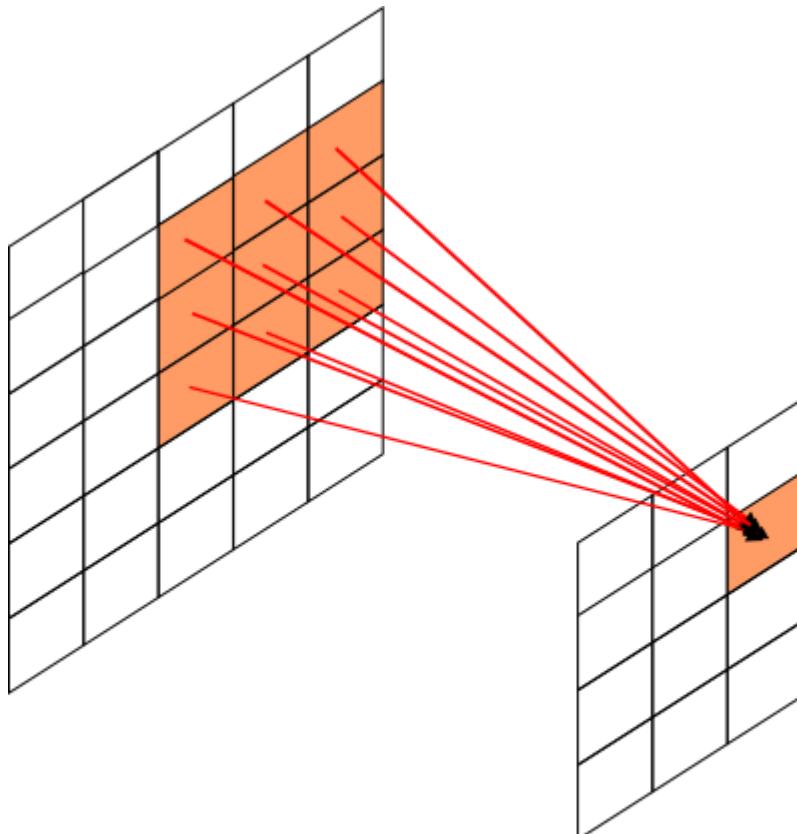


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

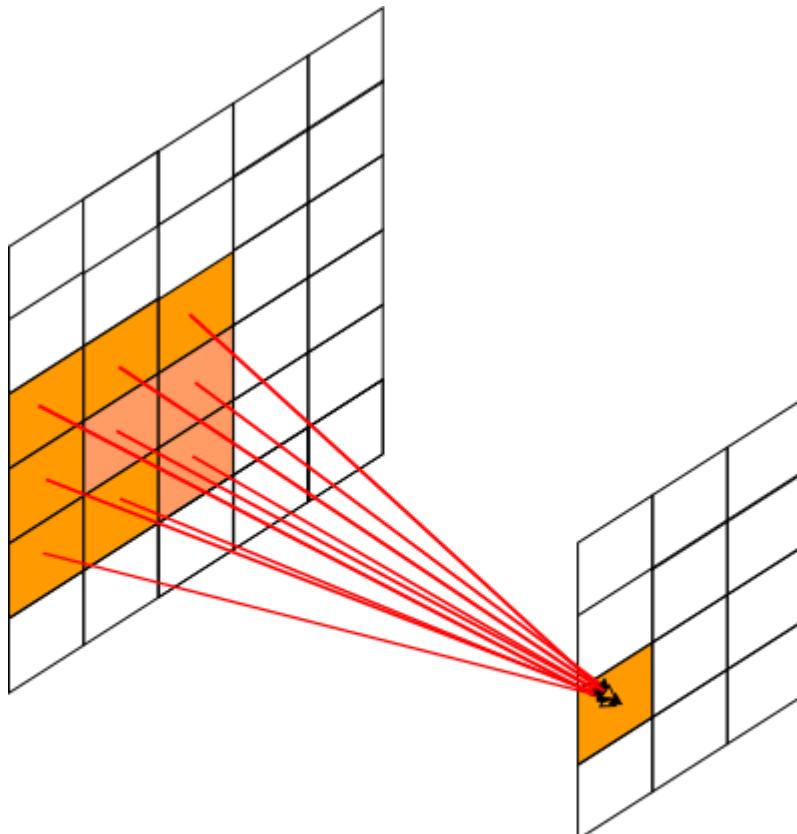


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

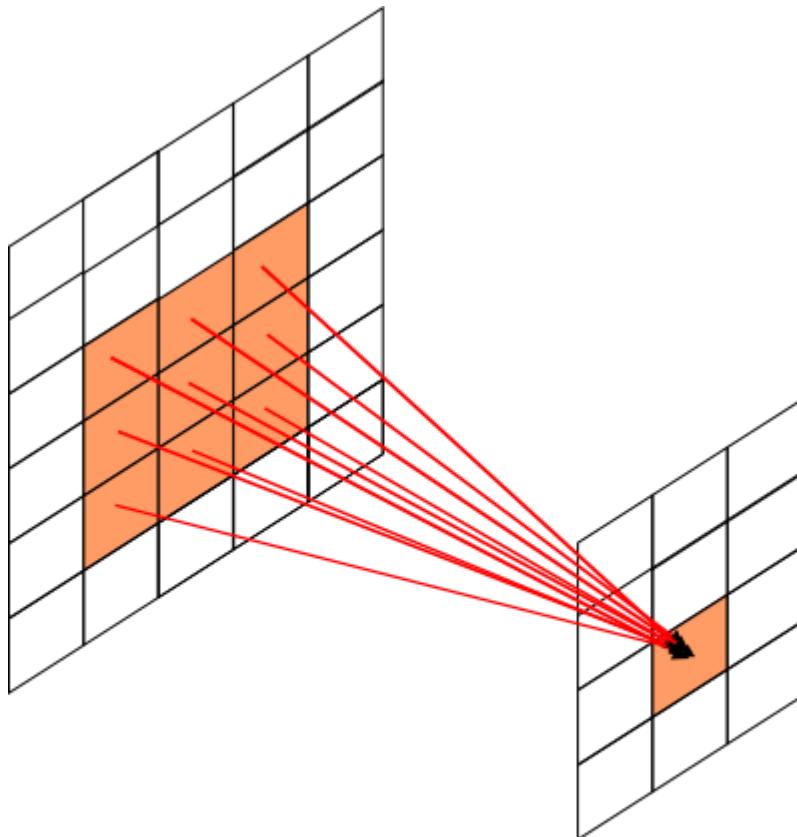


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

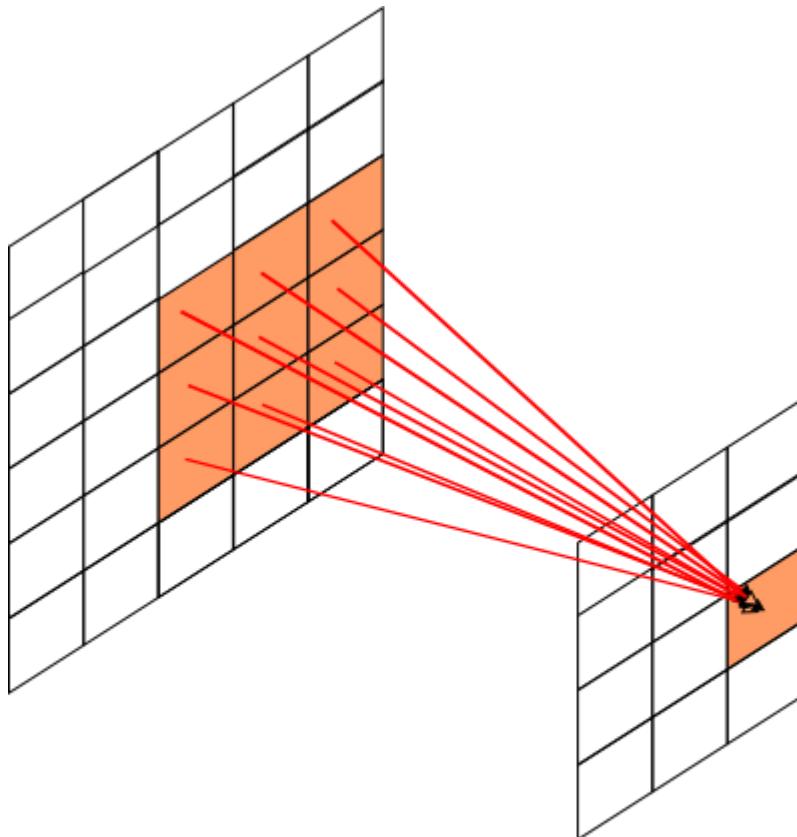


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

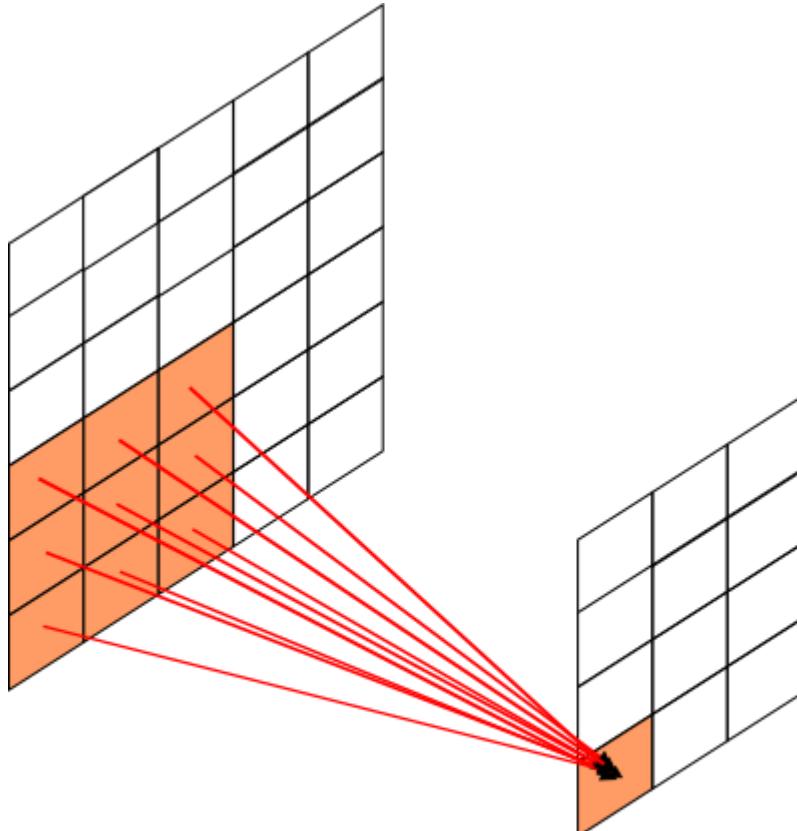


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

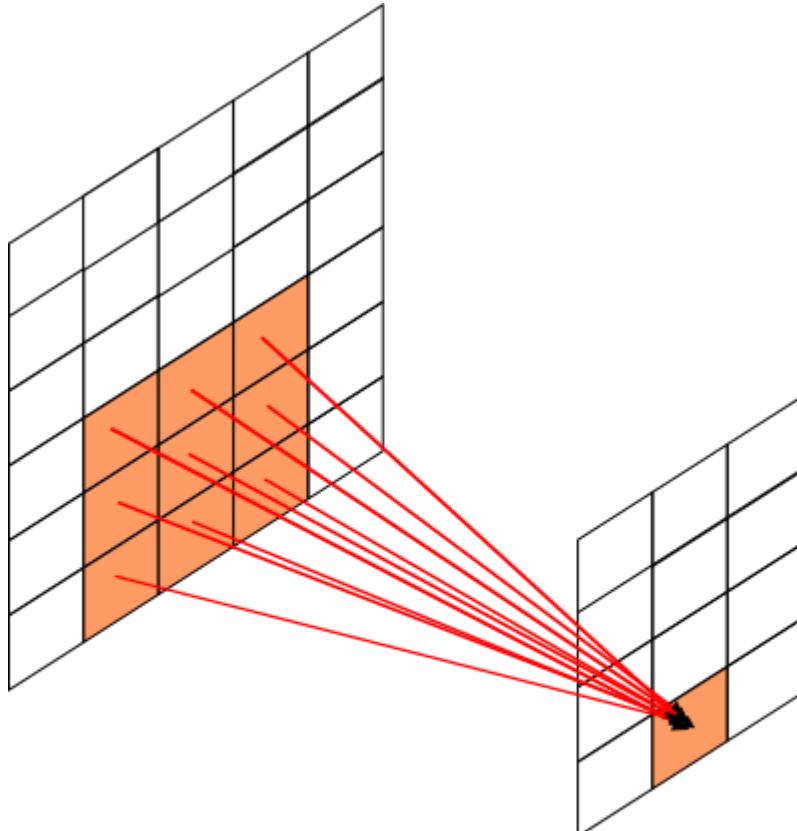


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

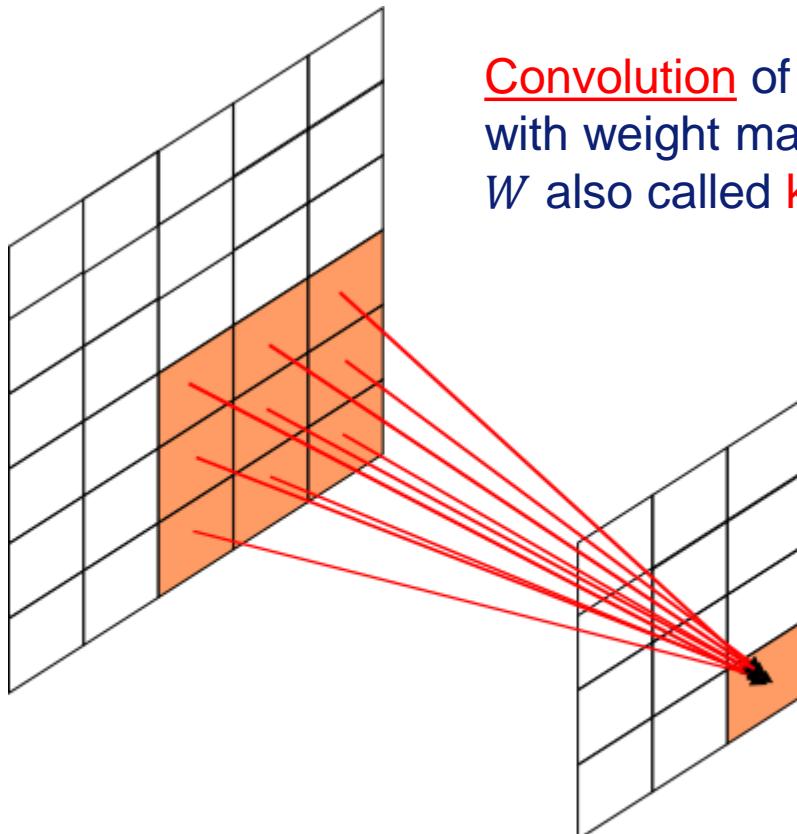


Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



Convolution of input image
with weight matrix W ;
 W also called **kernel**

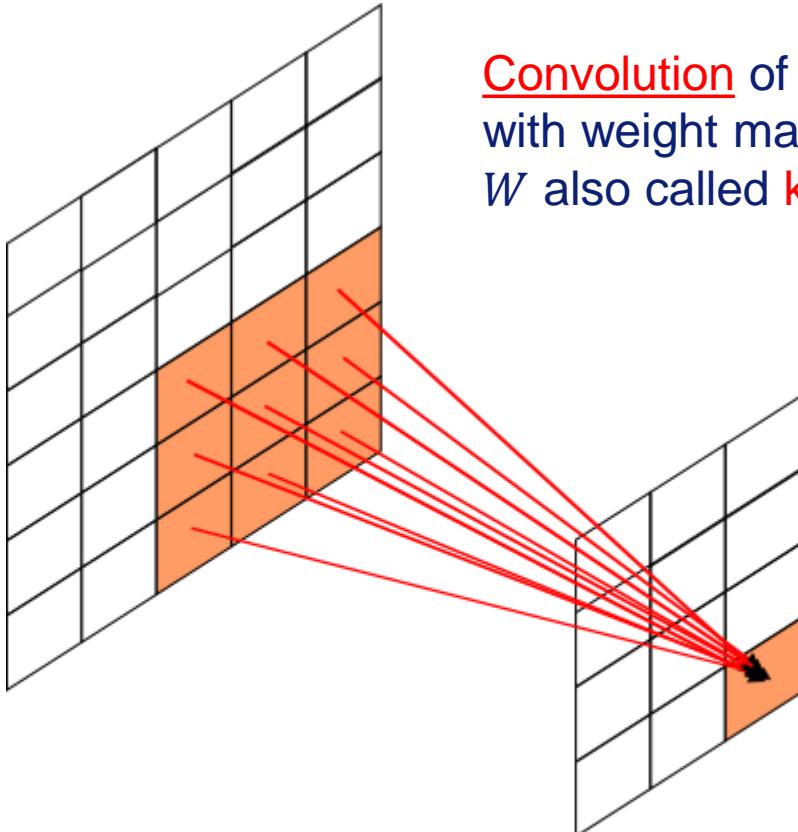
Example weights:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

activation map /
feature map

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space

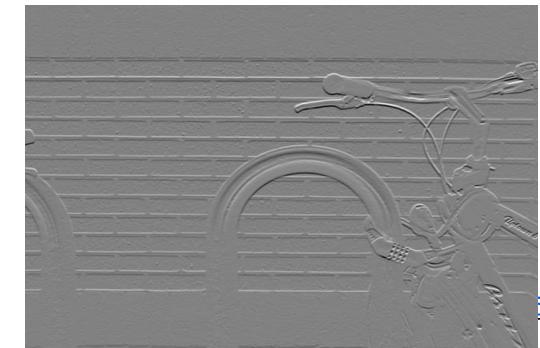


Convolution of input image
with weight matrix W ;
 W also called **kernel**

activation map /
feature map

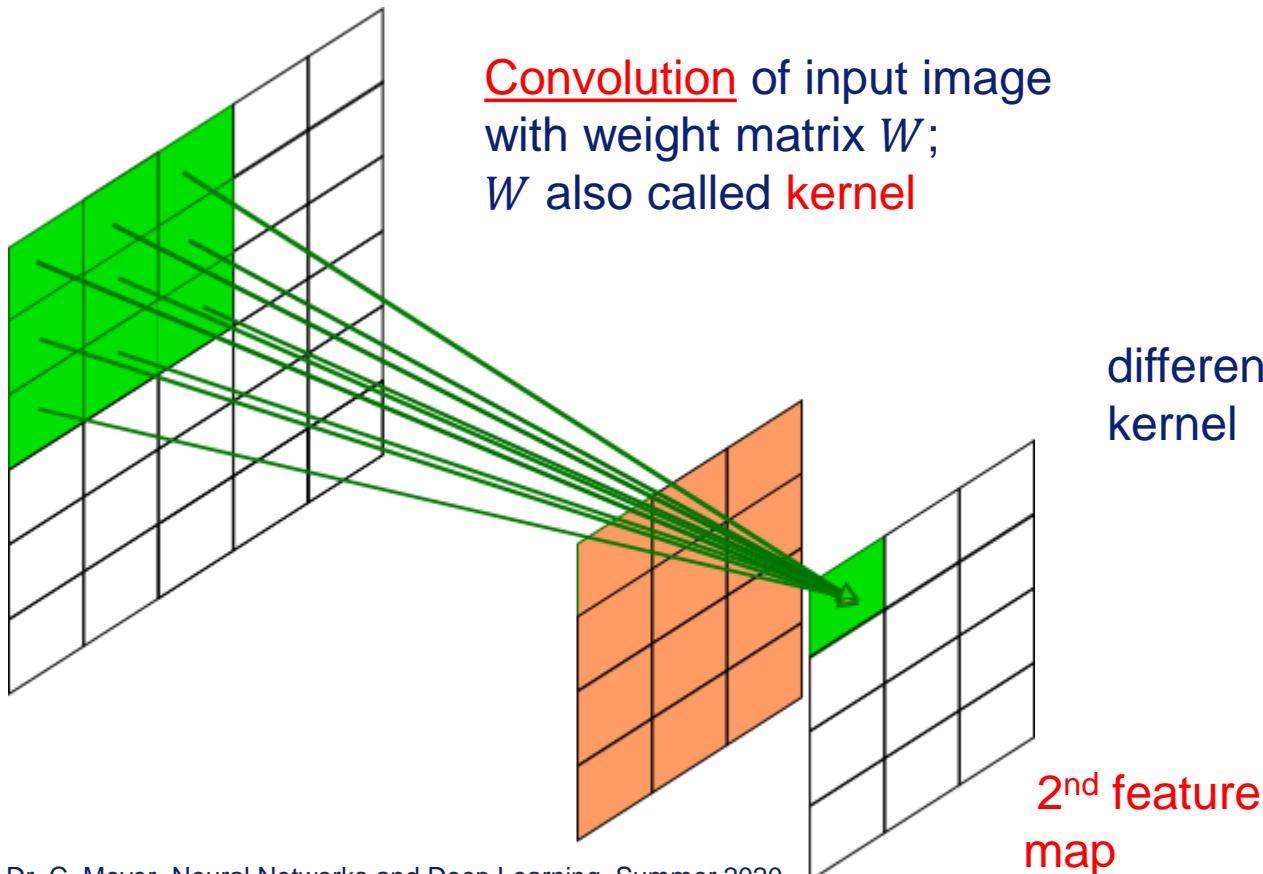


$$W = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



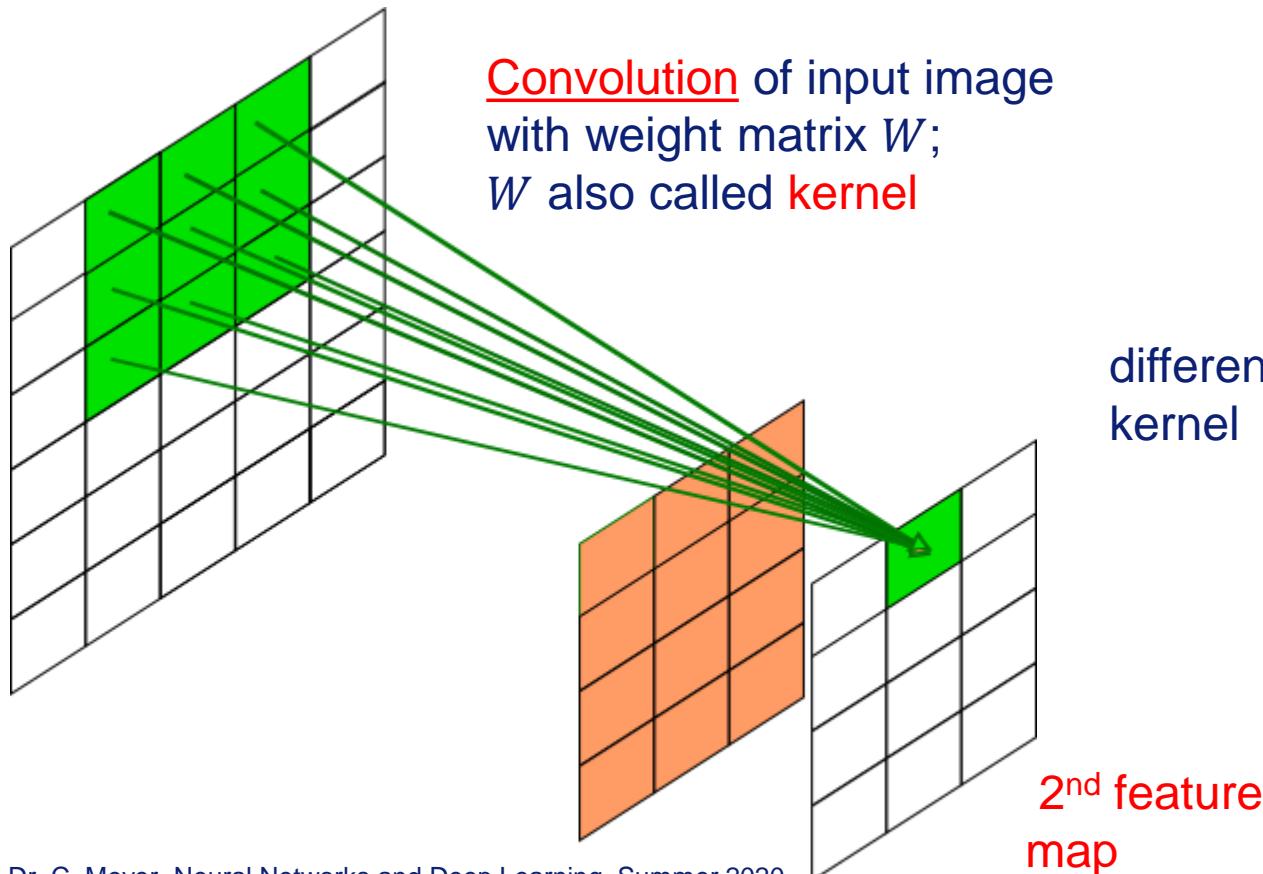
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



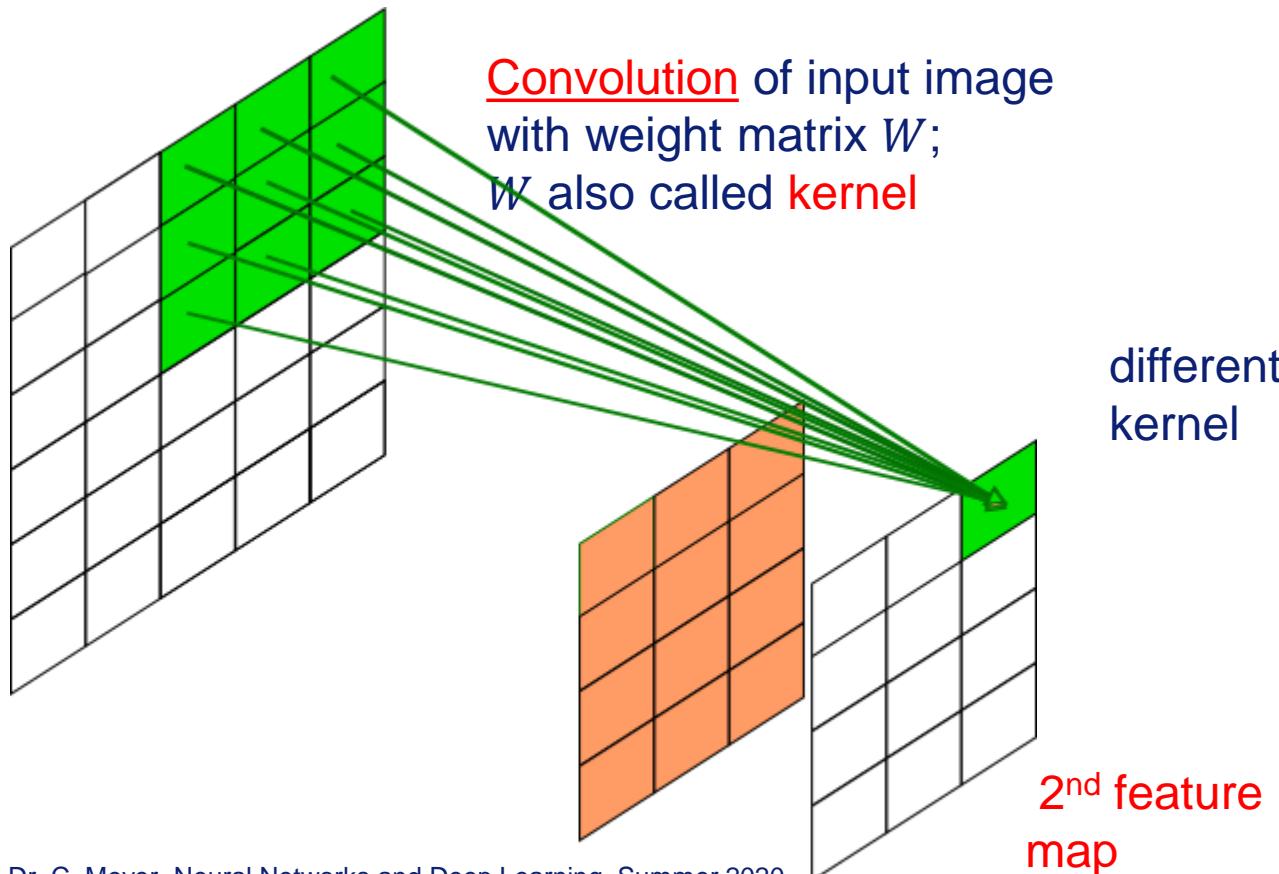
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



Convolutional neural networks: Illustration

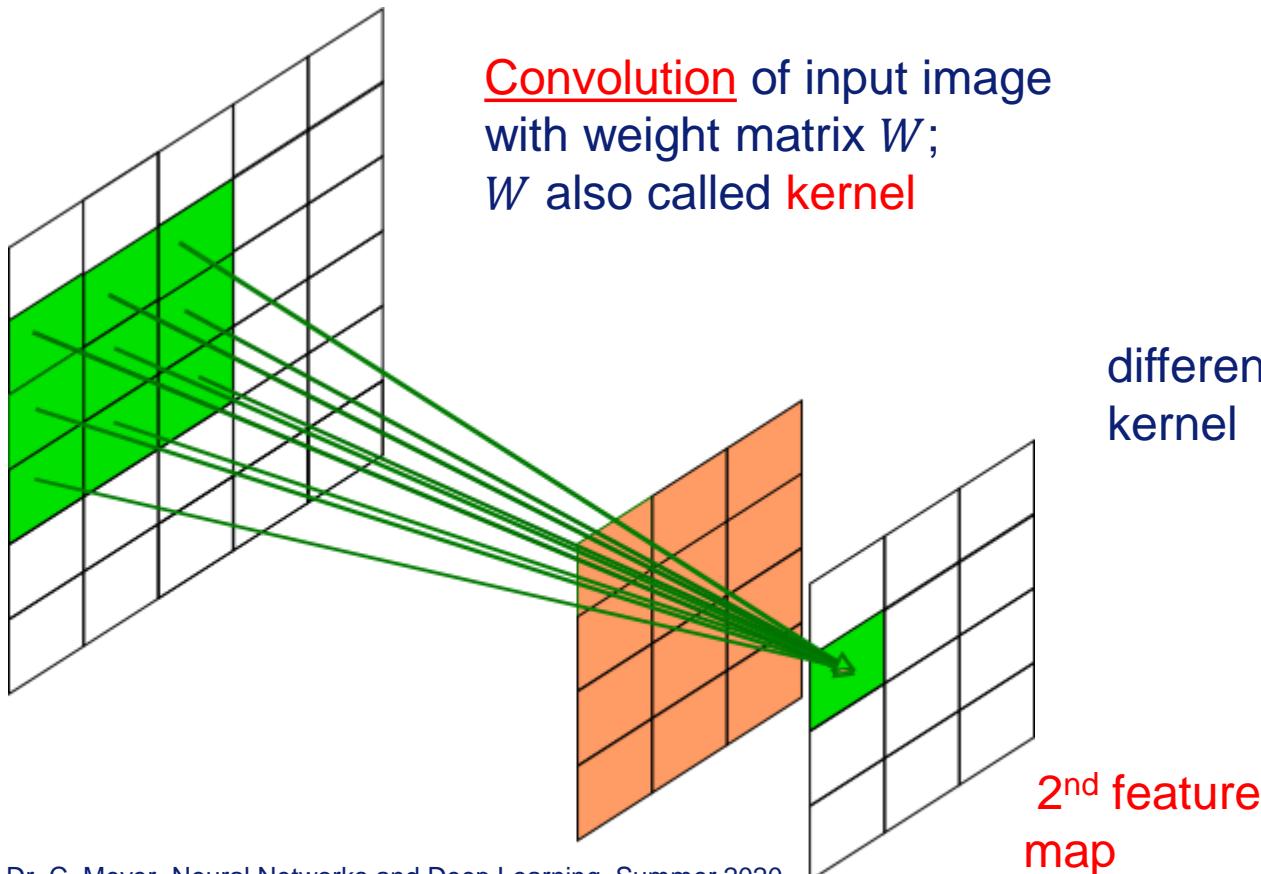
- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

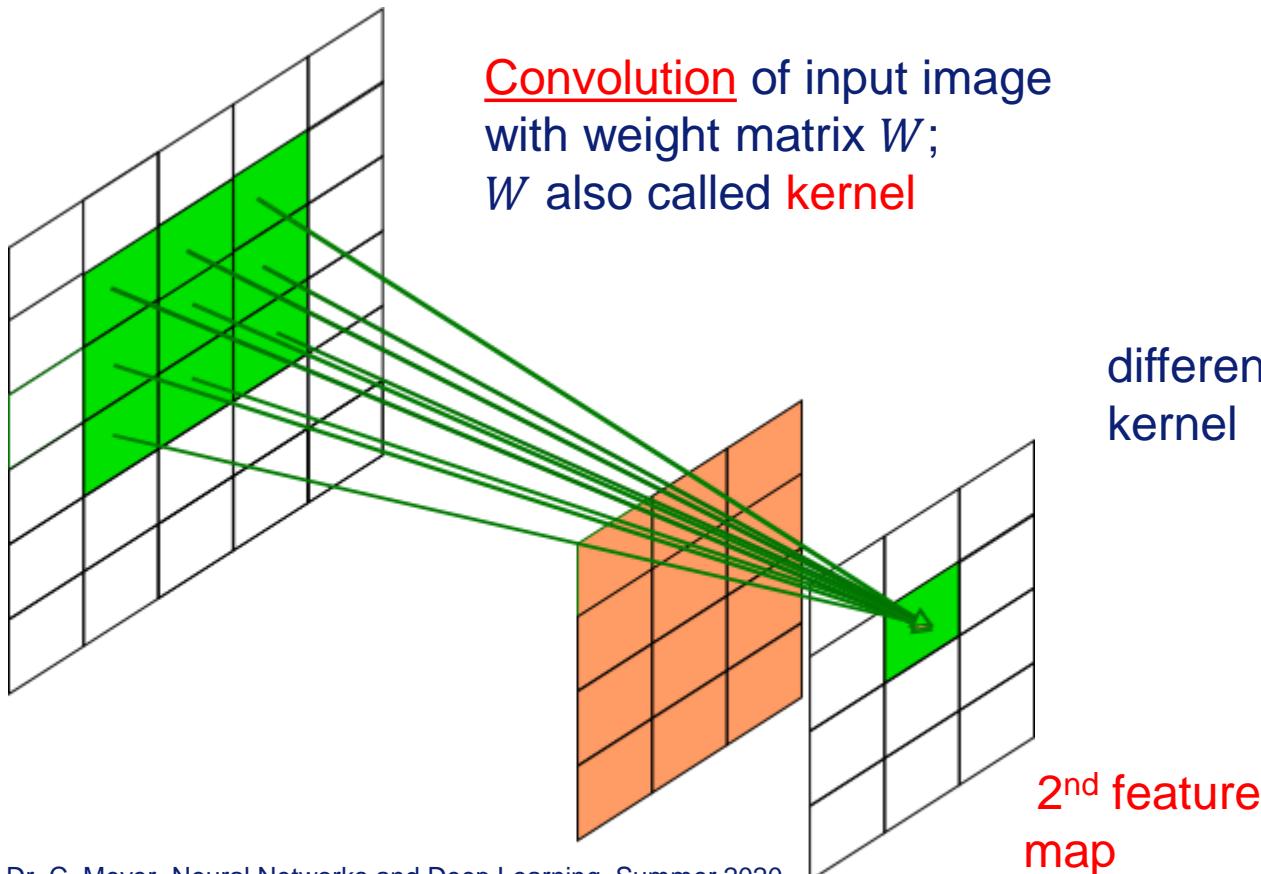
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



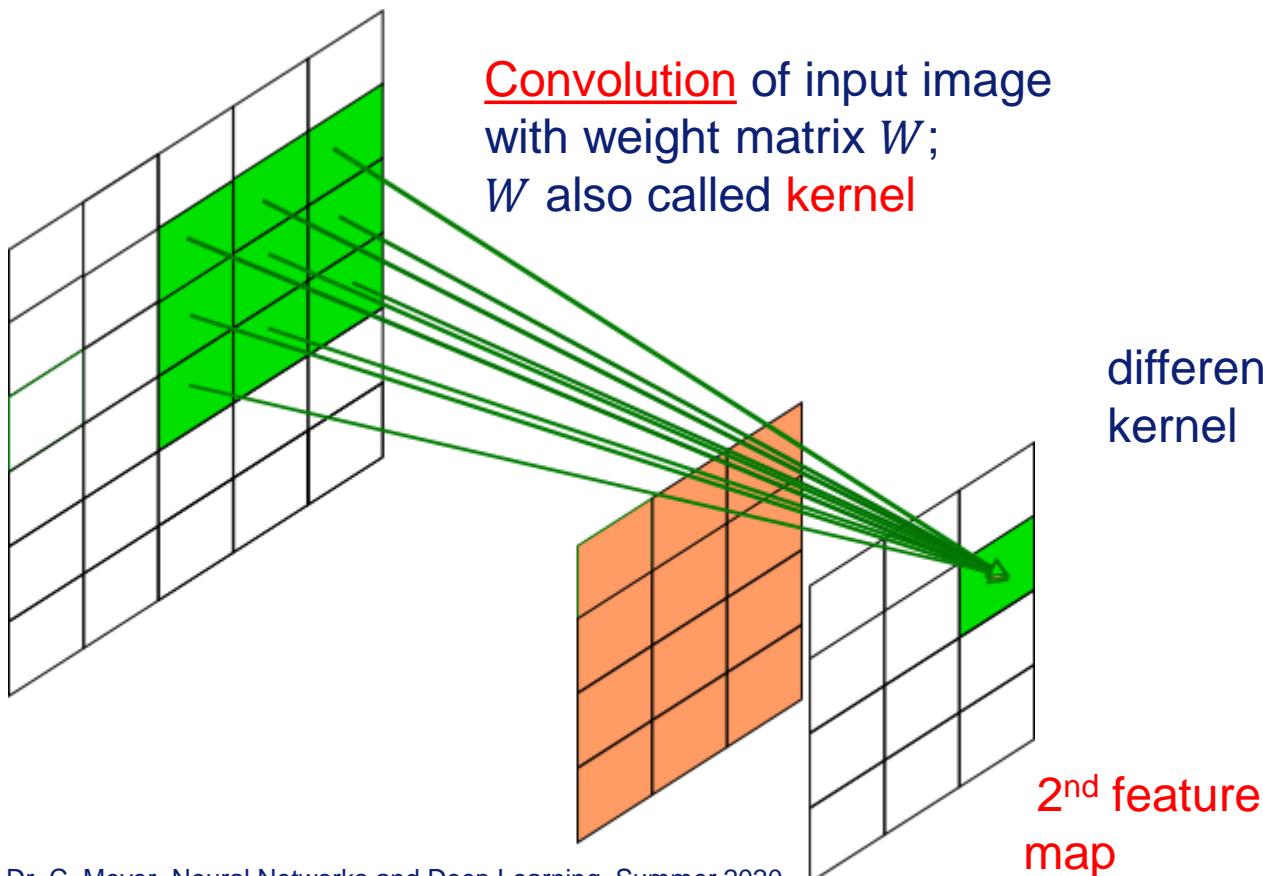
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space

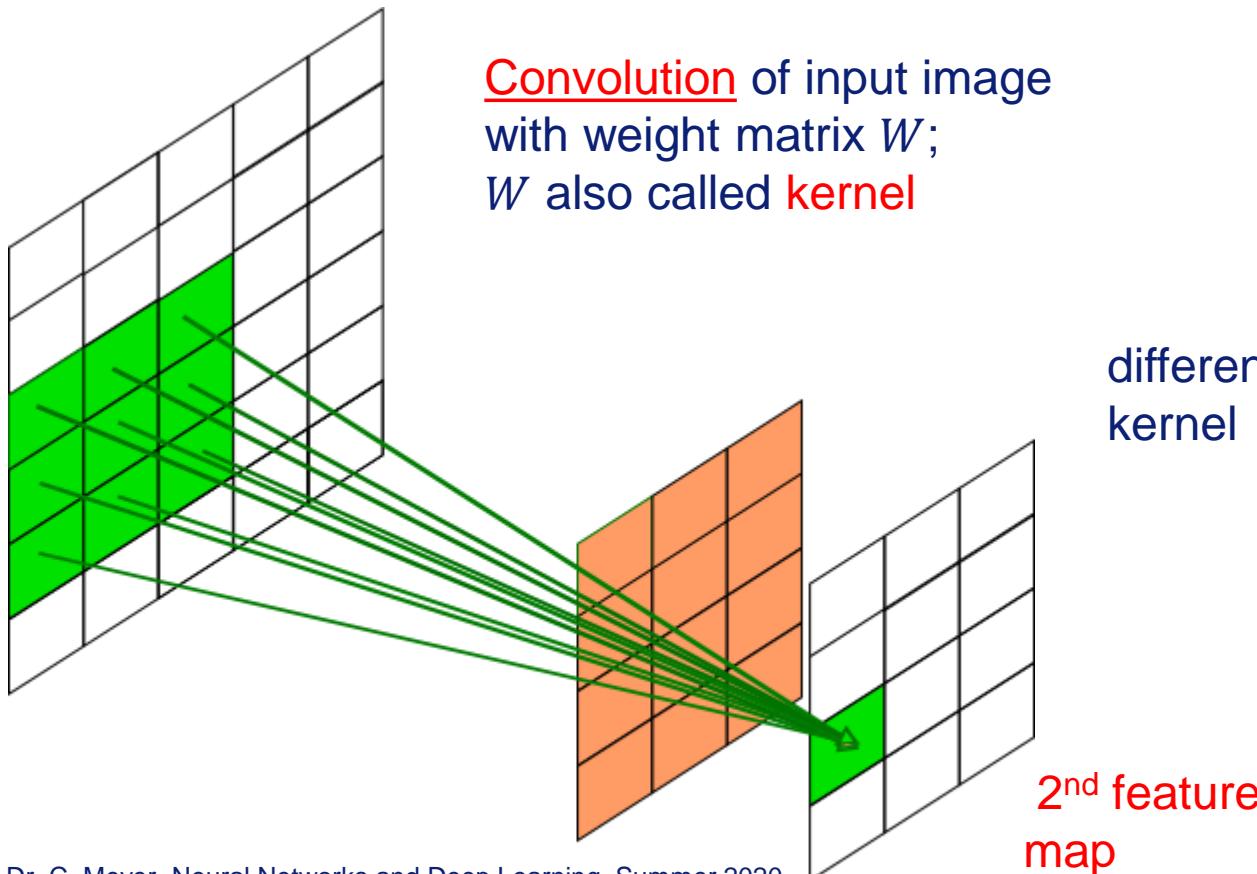


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

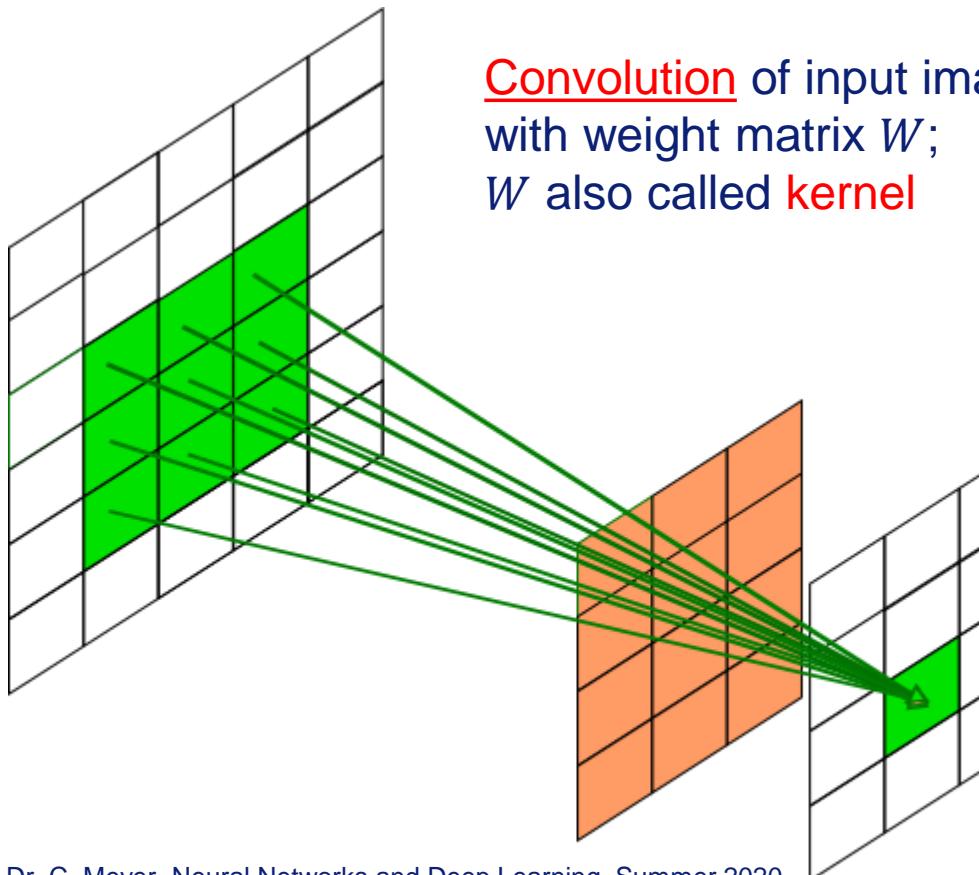
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space



Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space

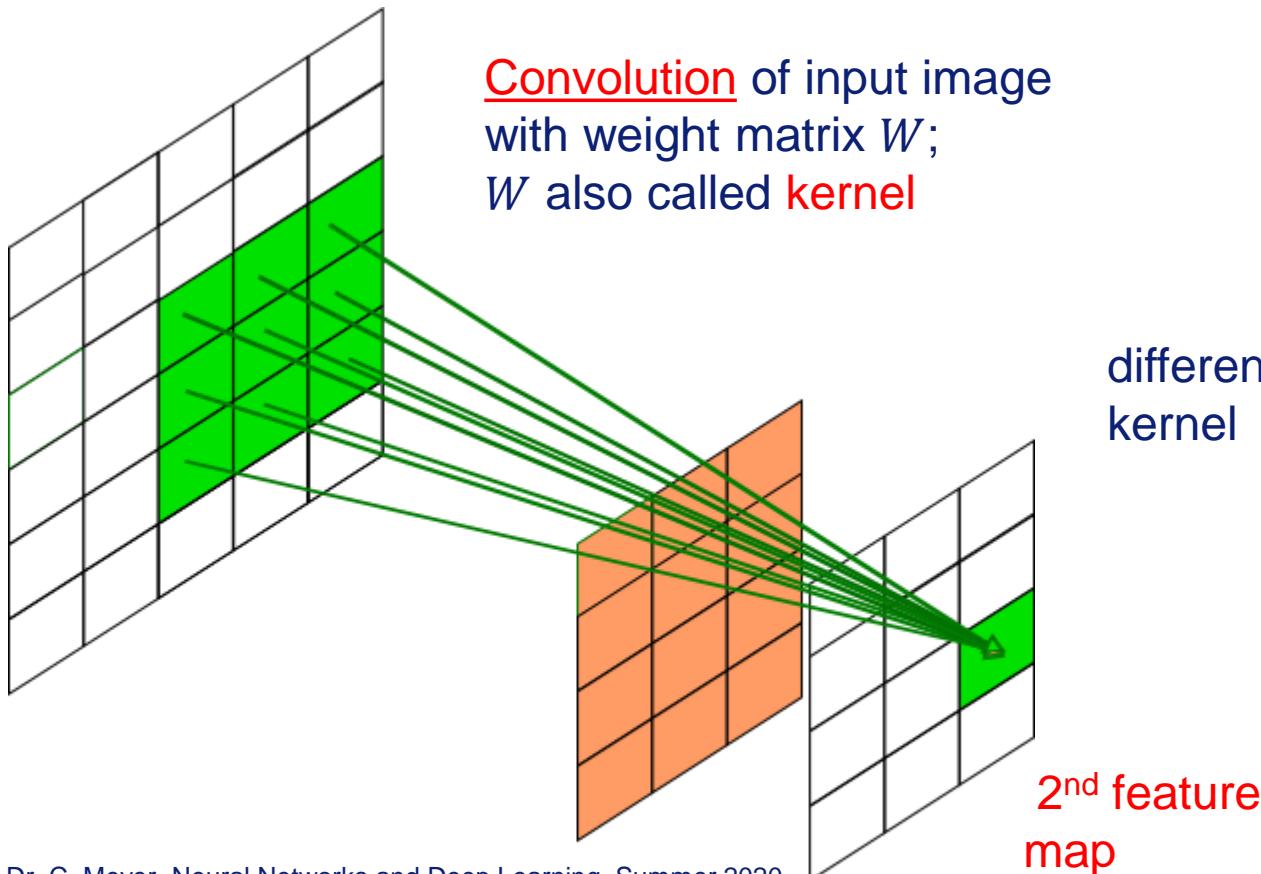


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

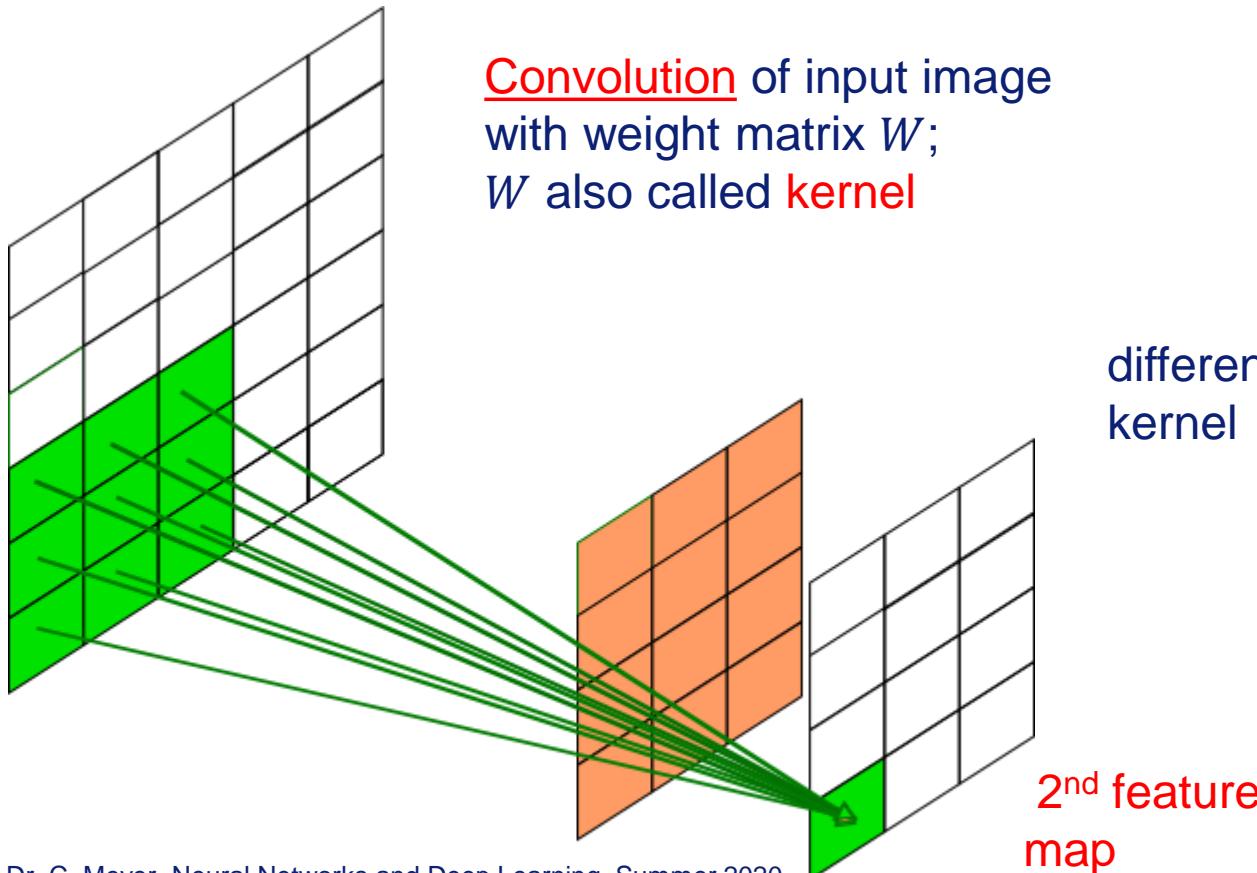


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

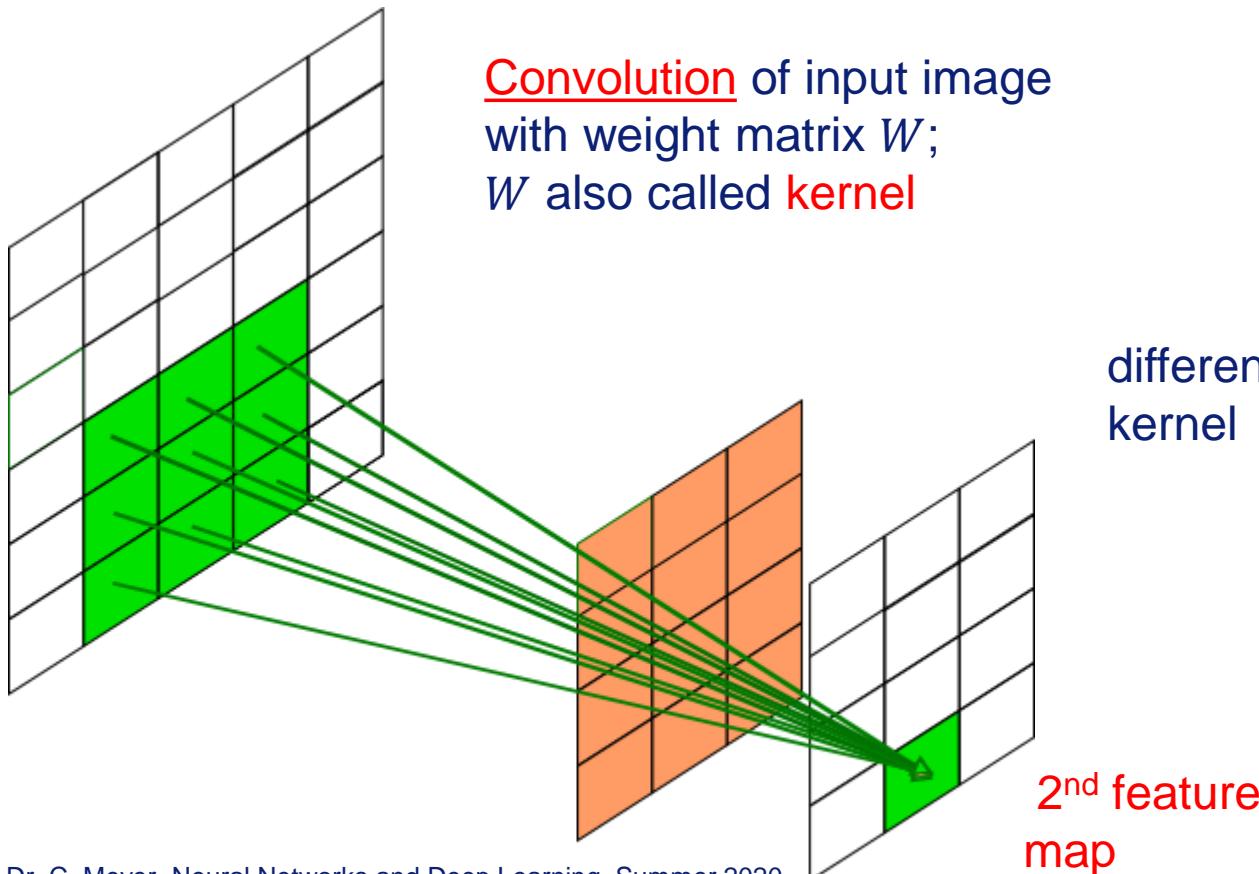


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space

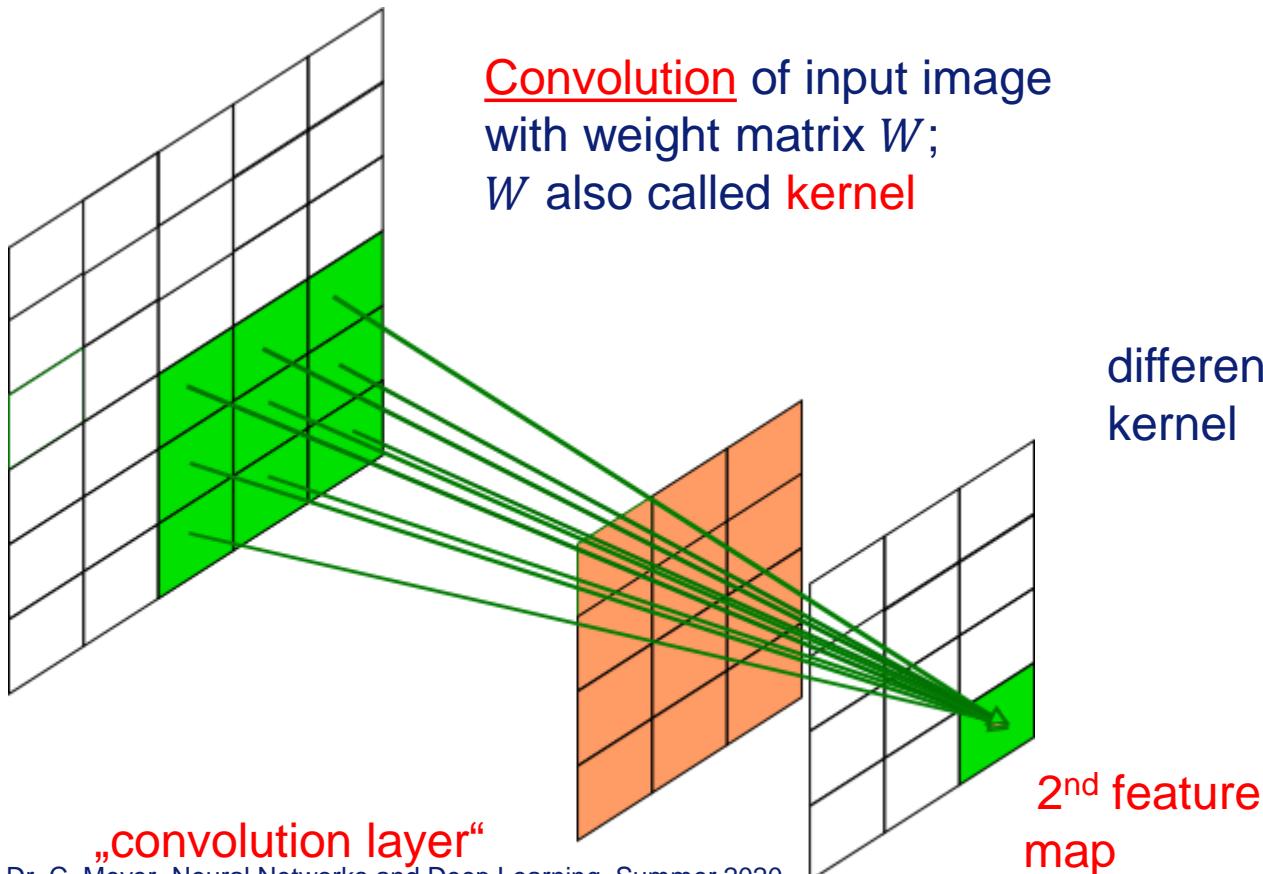


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space

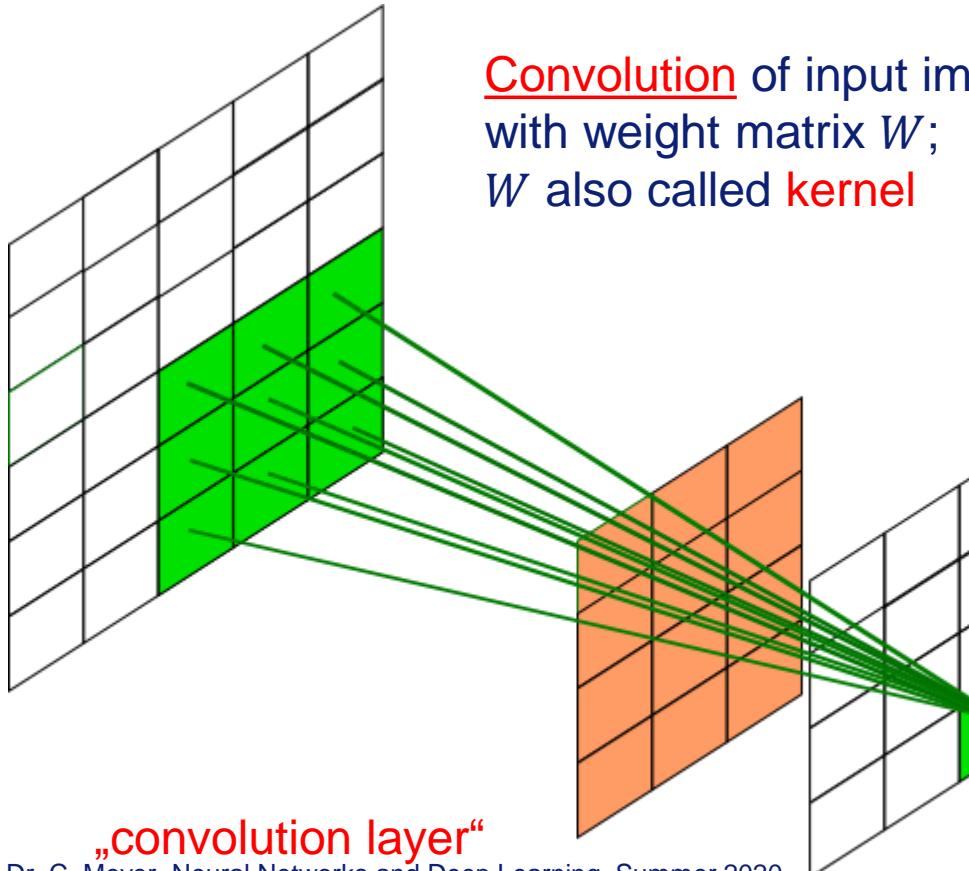


different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

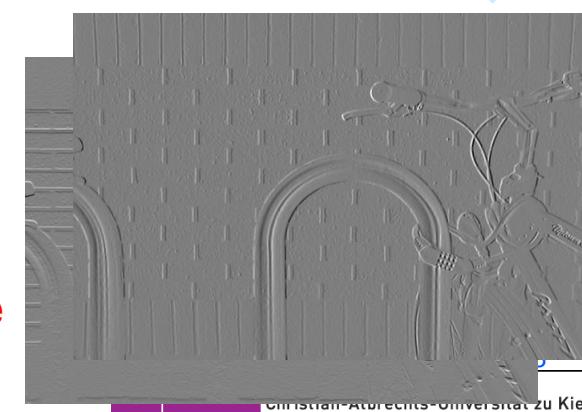
Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - Local connectivity: Connect each hidden unit to a *small* input patch
 - Parameter sharing: Share the synaptic weights across space



different kernel

$$W^T = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$



Convolutional neural networks: Illustration

- Key ideas of convolutional neural networks:
 - a) Local connectivity: Connect each hidden unit to a *small* input patch
 - b) Parameter sharing: Share the synaptic weights across space
 - c) Pooling / subsampling hidden units

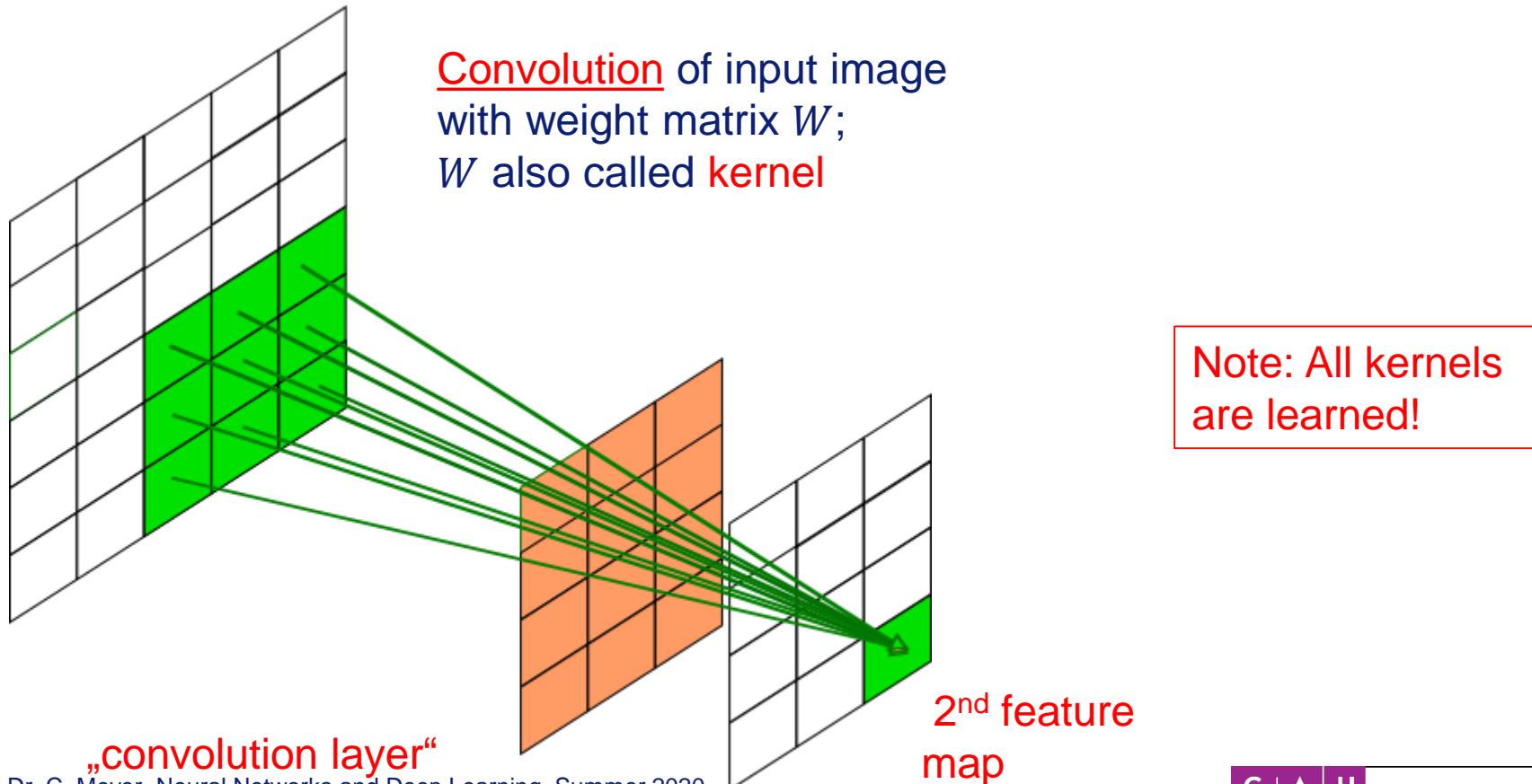
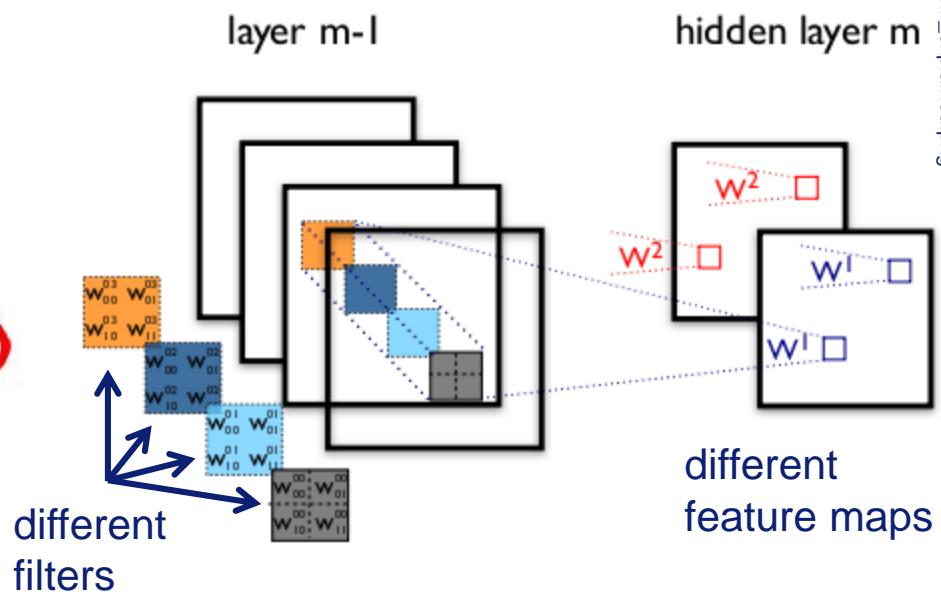
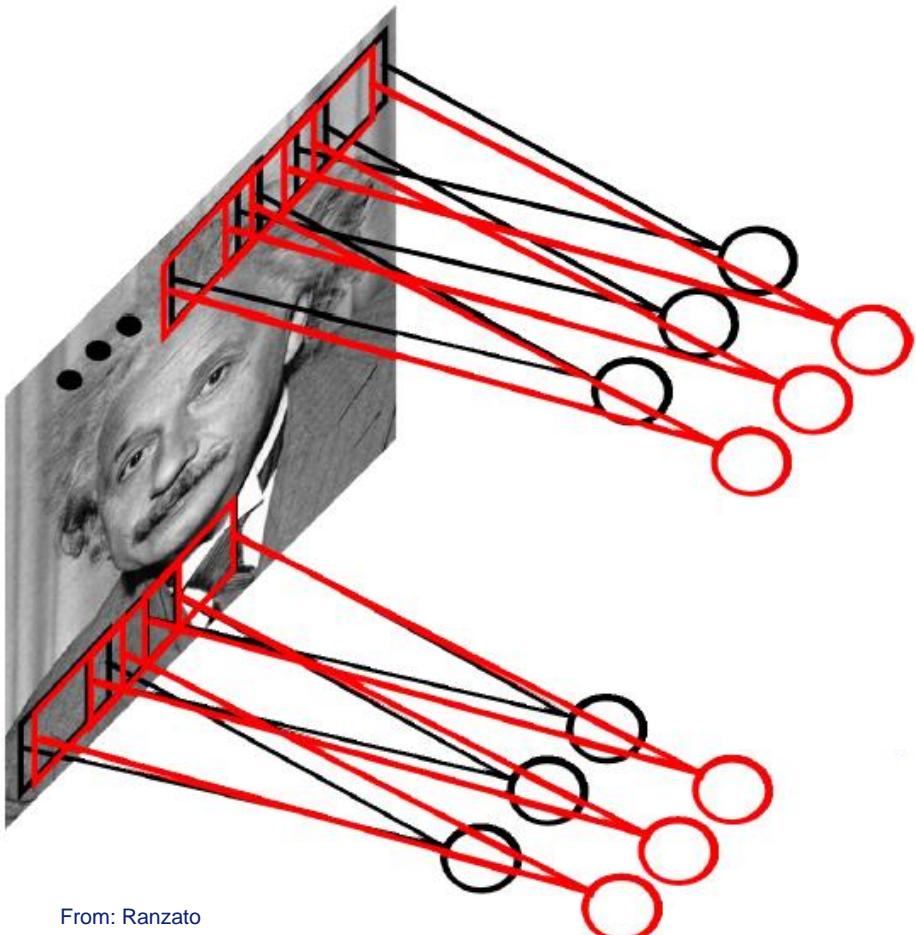


Illustration of multiple feature maps

- Use **different kernels (filter)** with distinct parameters
→ i.e. compute **multiple feature maps**, highlighting different aspects in the input image



Example:

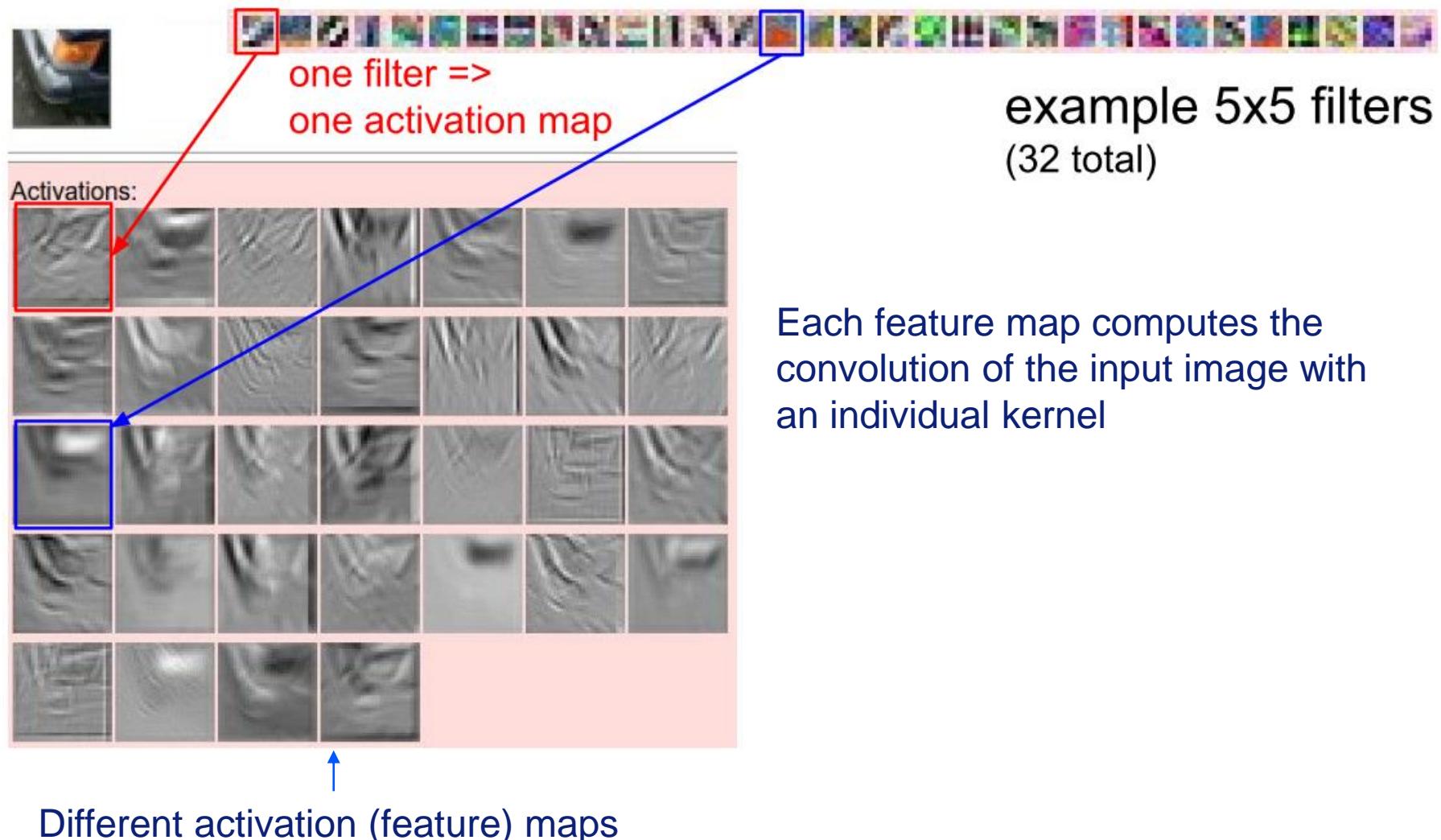
$10^3 \times 10^3$ input pixel
 Filter (patch) size: 10×10
 100 filters
 → 10^4 parameters

Convolution layer and feature maps: Illustration



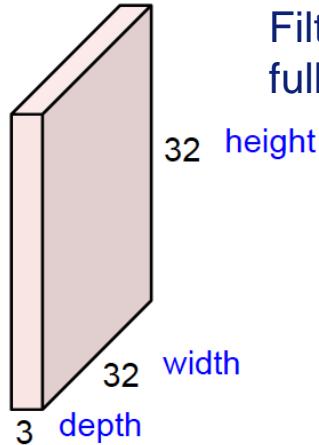
Input

Illustration of multiple feature maps

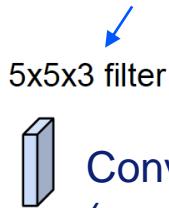


Convolution layer: Applying convolution with kernel

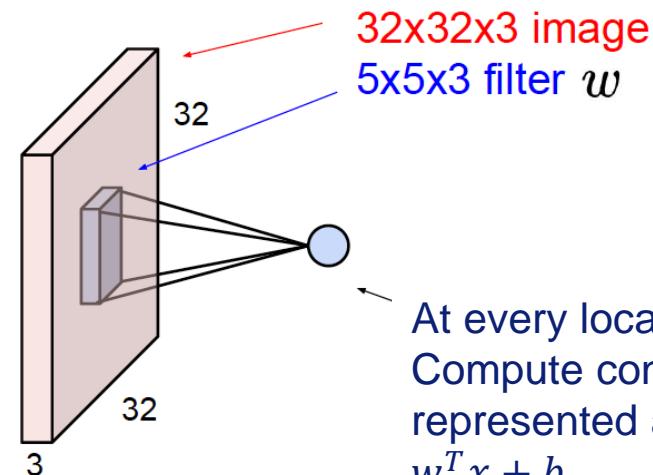
32 x 32 x 3 (H x W x D) image, 5 x 5 x 3 (H' x W' x D) kernel (filter)



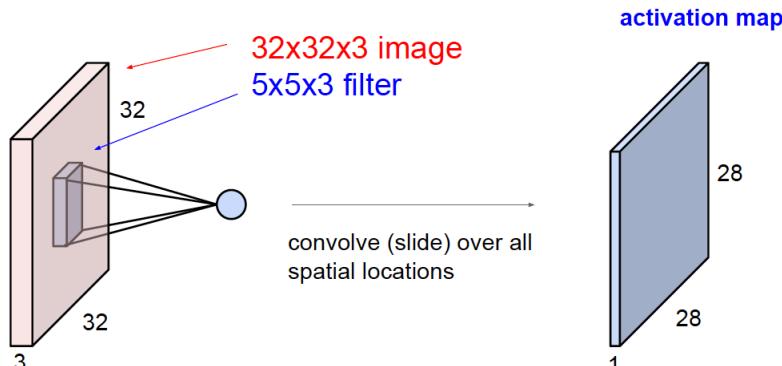
Filters always extend up to the full depth of the input volume



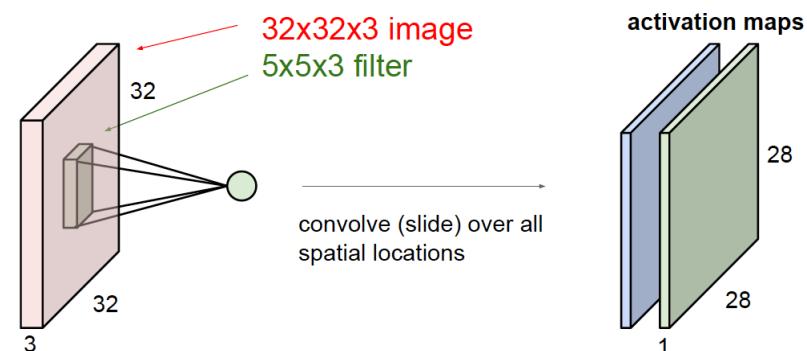
Convolve filter with image (convolution at every Image location)



single kernel \rightarrow single feature map:

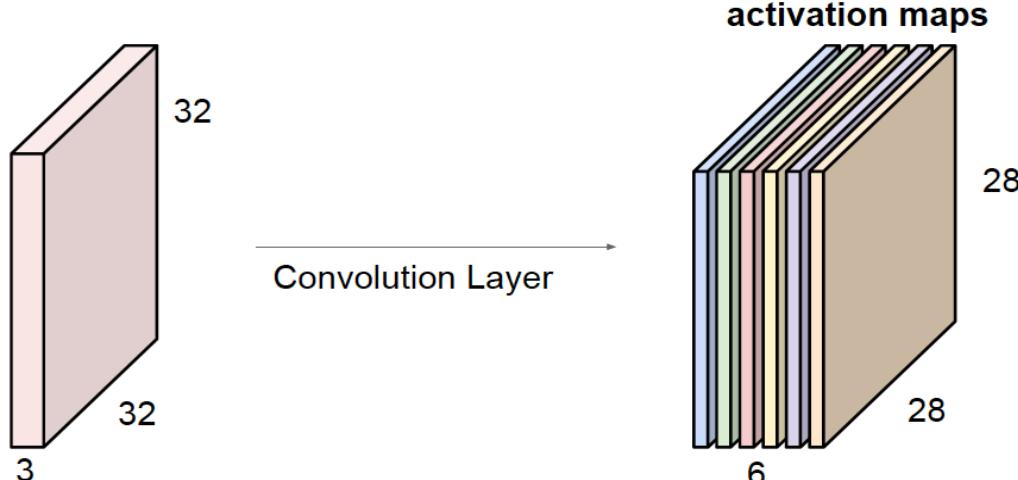


2nd kernel \rightarrow 2nd feature map:



Convolution layer: Result of applying multiple kernels

- 6 activation (feature) maps from six 5×5 filters



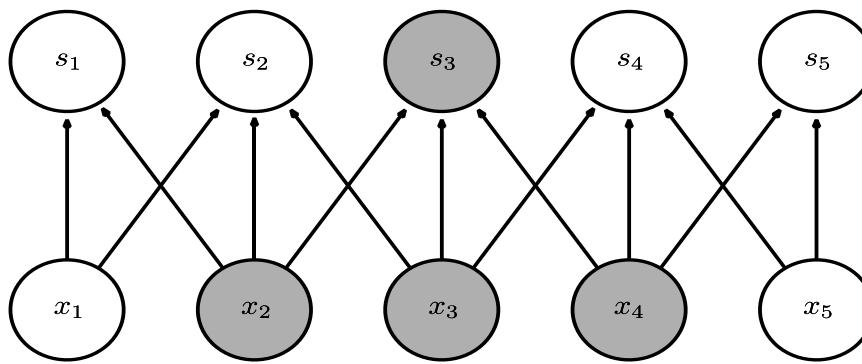
Note on terminology:

- Feature / activation map:** Result of applying a kernel (*output* of a layer)
- Channel:** Individual *input* ($H \times W$) of a layer
- Depth:** Number of channels
→ (output) feature maps of a layer are the (input) channels of the next layer

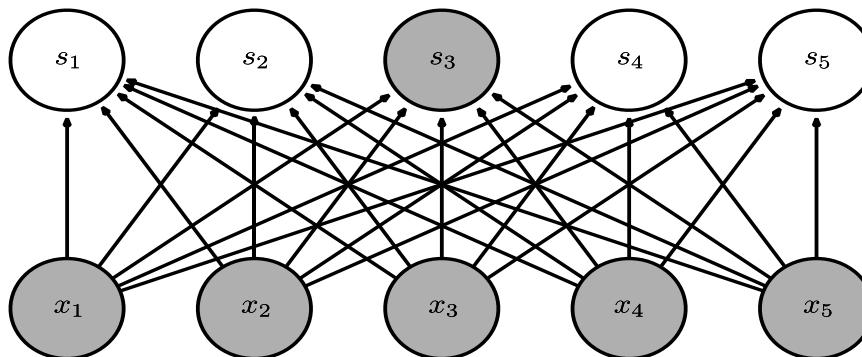
- Activations are stacked to new image tensor, size $28 \times 28 \times 6$ ($H \times W \times D$)
 - ConvLayer:
 - Accepts volume of size $H_1 \times W_1 \times D_1$ (e.g. $32 \times 32 \times 3$)
 - Has K filters with spatial extent F (e.g. $K = 6, F = 5$)
- Produces volume of size $H_2 \times W_2 \times D_2$
- $$H_2 = H_1 - F + 1 \quad (\text{e.g. } H_2 = 32 - 5 + 1 = 28)$$
- $$W_2 = W_1 - F + 1 \quad (\text{e.g. } W_2 = 32 - 5 + 1 = 28)$$
- $$D_2 = K \quad (\text{e.g. } D_2 = 6)$$

Convolutional versus fully connected layer (1)

- Convolution layer is a fully connected layer where the weights are non-zero only over a small neighborhood



Convolution layer:
Local connectivity,
parameter sharing



Fully connected layer:
All possible connections,
no parameter sharing

Convolutional versus fully connected layer (2)

- The convolution (cross-correlation) of image \mathbf{x} and kernel \mathbf{w} is a sum over pointwise multiplications between each kernel element w_i and a respective image pixel x_i , and adding a bias term b :

$$z = \sum_{i=1}^n w_i x_i + b = \mathbf{w} \cdot \mathbf{x} + b$$

- This is formally equivalent to a fully connected (fc) layer in a MLP
 - Only differences between a CONV layer and a fully connected layer: CONV layer is only locally connected and shares many parameters
 - Any CONV layer could be expressed by an fc layer, where the fc weight matrix is mostly zero (due to local connectivity) and where the weights in many blocks are equal (due to parameter sharing)
 - Any fc layer could be expressed by a CONV layer, by setting the filter size to match the complete image height and width, the number of filters K to the number of neurons in the fc layer, no padding and stride 1 (see later)
 - fc \rightarrow CONV conversion especially useful in fully convolutional networks (\rightarrow later)
- Each kernel described by synaptic weights \mathbf{w} ($F \times F \times D_1$) and single bias b

Convolution (cross-correlation) as matrix multiplication

- Fully connected layers: $\mathbf{z}^l = \mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l$ (layer l)

$$n_l \text{ outputs} \quad \left[\begin{array}{c} \mathbf{z} \\ \vdots \end{array} \right] = \left[\begin{array}{c} \mathbf{W} \\ \vdots \end{array} \right] n_l \quad \left[\begin{array}{c} \mathbf{a} \\ \vdots \end{array} \right] \quad n_{l-1} \text{ inputs}$$

$$\mathbf{W} \quad (n_l \times n_{l-1}) \cdot (n_{l-1} \times 1) \rightarrow (n_l \times 1)$$

- CONV layers: Write image patches to be convolved into columns (im2col)

$$K \quad \left[\begin{array}{c} \mathbf{z}^T \\ \vdots \end{array} \right] = \left[\begin{array}{c} \text{Filter 1} \\ \text{Filter 2} \\ \vdots \\ \mathbf{W}^T \end{array} \right] K \quad \left[\begin{array}{c} \text{Patch 1} \\ \text{Patch 2} \\ \vdots \\ \mathbf{a} \end{array} \right] \quad F*F*D_1 \text{ inputs}$$

num. patches F*F*D₁ num. patches

reshape to image

$\mathbf{W}^T \quad \mathbf{a} \quad \mathbf{z}^T$
 $(K \times (F*F*D_1)) \cdot ((F*F*D_1) \times \text{num. patches.}) \rightarrow K \times \text{num. patches.}$

→ very efficient execution on GPUs (but: memory increase)!

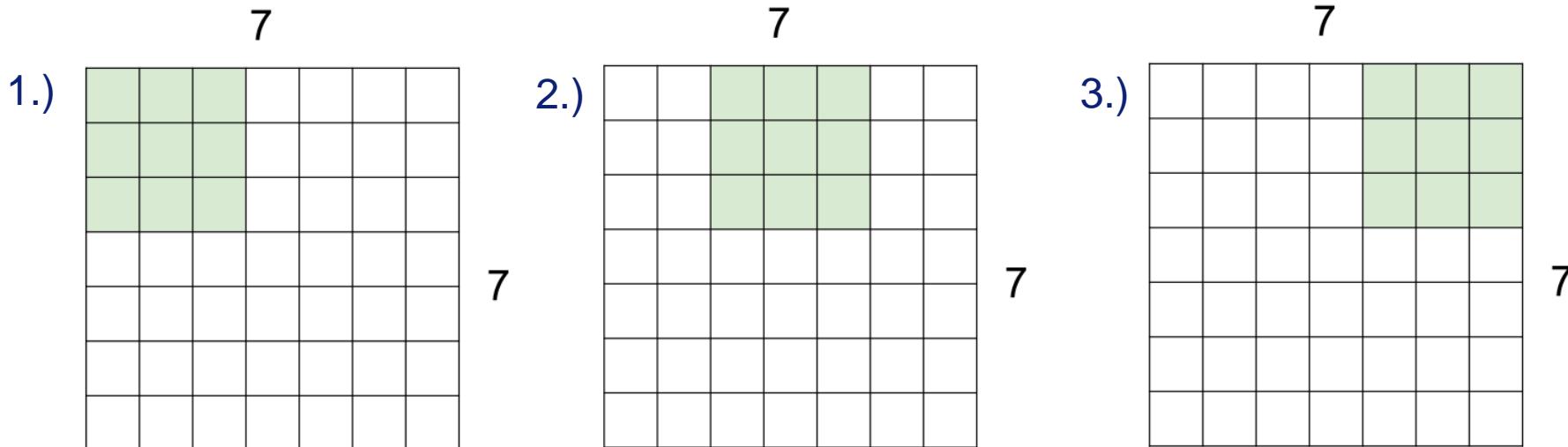
Input Image

im2col

$\left[\begin{array}{c} \text{Patch 1} \\ \text{Patch 2} \\ \vdots \\ \mathbf{a} \end{array} \right] \quad F*F*D_1 \text{ inputs}$
 num. patches

Convolution layer: Stride

- Apply filter not at every image location, but in larger steps: „Stride“
- Example: Stride 2 (illustrated for rows, but applied similarly to columns!)



→ Output: 3 x 3 feature map (instead of 5 x 5 feature map for stride 1)

- Output size = $(\text{image size} - \text{filter size})/\text{stride} + 1 = (N - F)/S + 1$
- Stride 3? Doesn't fit, i.e. cannot apply 3 x 3 filter on 7 x 7 input with stride 3

Convolution layer: Padding

- Often, image borders are **padded** (extended), mostly with value 0

0	0	0	0	0	0			
0								
0								
0								
0								
0								

Example:

Input: 7×7 ($N = 49$)

Filter: 3×3 ($F = 9$)

Stride: $S=1$

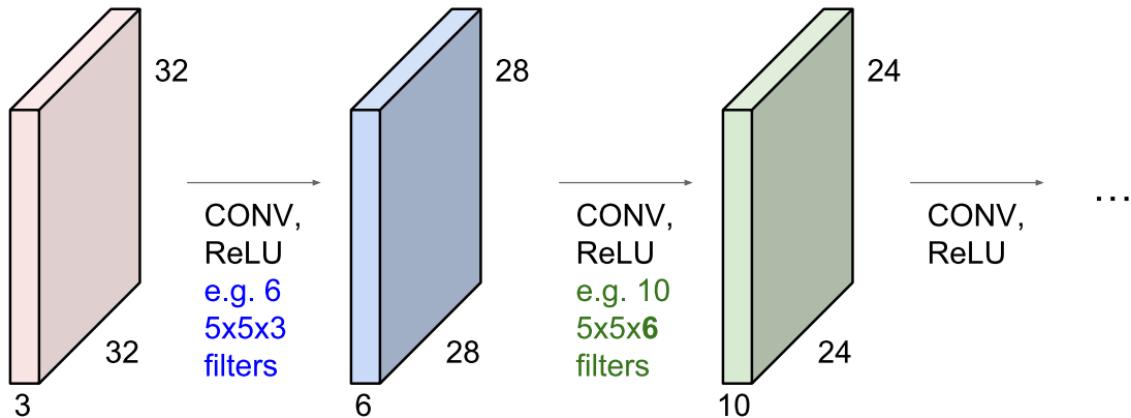
Padding with 1 pixel border ($P = 1$)

→ Output: 7×7 (preservation of image size!)

- Common: CONV layer with stride 1, filter size $F \times F \rightarrow$ pad with $P=(F-1)/2$
 - „same conv.“; preserves input size, e.g. pad with 1, 2, 3 for $F = 3, 5, 7$, resp.
- Output size: $(N - F+2P) / S + 1$ (N : image size, i.e. height or width)
- But: Padding introduces image artifacts! Zero padding often not adequate**

Valid / same / full convolution

- „valid convolution“: corresponds to no padding ($P = 0$)
 - resulting in output size which is smaller than the input size

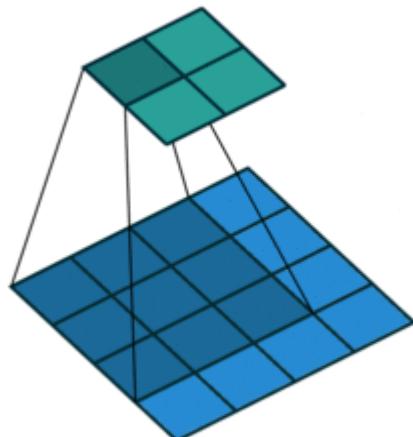


- „same convolution“: padding with half the filter size (rounded)
 - If stride = 1, the output size is identical to the input size
- „full convolution“: padding with one less than filter size on both sides
 - Convolution is computed wherever input and filter overlap by at least 1 pixel
 - For $F > 1$ and $S = 1$, the output size is larger than the input size

Convolution layer: Examples for strides and padding (1)

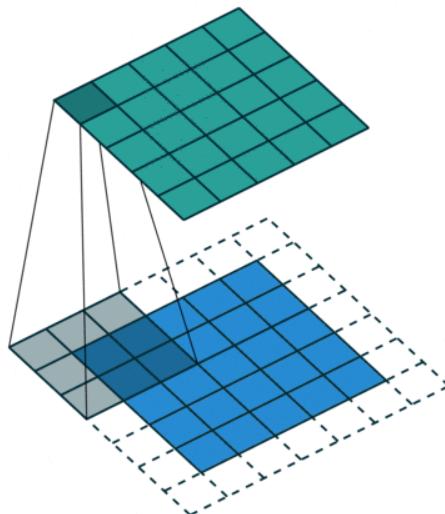
- Input: Blue map, output: cyan map
- Output size = $(N - F + 2P) / S + 1$

No padding ($P = 0$, „valid“)
Stride $S = 1$



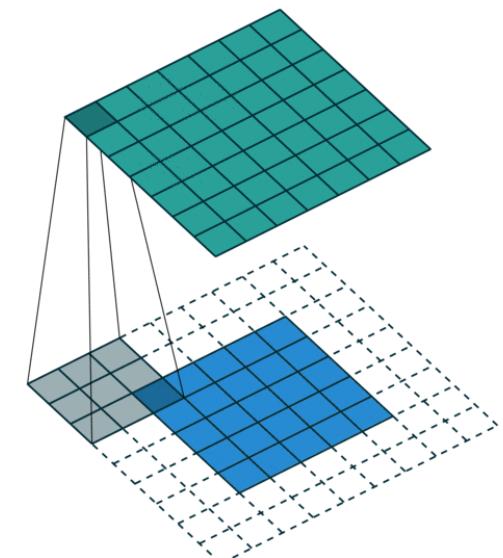
Input: 4×4
Filter: 3×3
Output: 2×2

Padding ($P = 1$, „same“),
Stride $S = 1$



Input: 5×5
Filter: 3×3
Output: 5×5

Padding ($P = 2$, „full“),
Stride $S = 1$

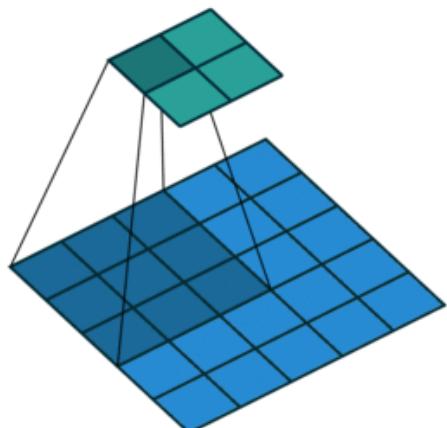


Input: 5×5
Filter: 3×3
Output: 7×7

Convolution layer: Examples for strides and padding (2)

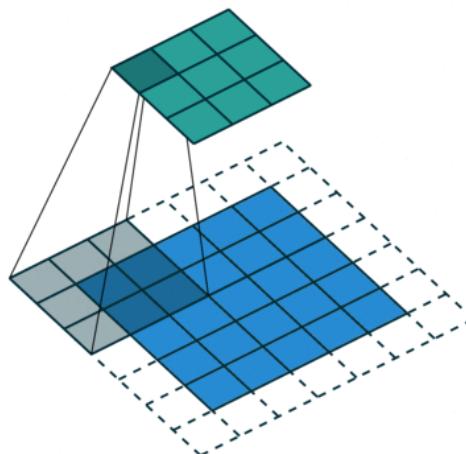
- Input: Blue map, output: cyan map
- Output size = $(N - F + 2P) / S + 1$

No padding ($P = 0$, „valid“),
Stride $S = 2$



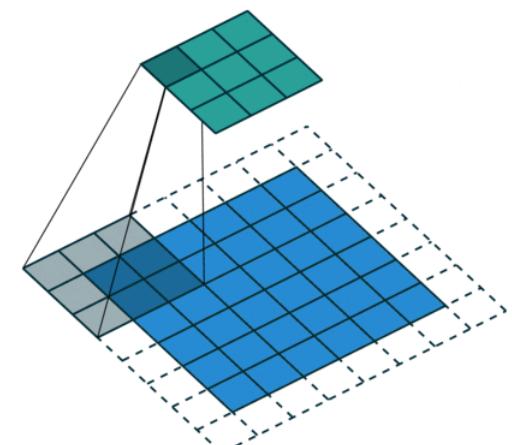
Input: 5×5
Filter: 3×3
Output: 2×2

Padding ($P = 1$, „same“),
Stride $S = 2$



Input: 5×5
Filter: 3×3
Output: 3×3

Padding ($P = 1$, „same“),
Stride $S = 2$



Input: 6×6
Filter: 3×3
Output: 3×3

Convolution layer: Summary

- To summarize, the CONV layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
 - Number of filters K ,
 - their spatial extent F ,
 - the stride S ,
 - the amount of zero padding P .
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F + 2P)/S + 1$
 - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
 - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d -th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d -th filter over the input volume with a stride of S , and then offset by d -th bias.

Common settings:

$K = (\text{powers of 2, e.g. } 32, 64, 128, 512)$

- $F = 3, S = 1, P = 1$
- $F = 5, S = 1, P = 2$
- $F = 5, S = 2, P = ?$ (whatever fits)
- $F = 1, S = 1, P = 0$

Sequence of convolution layers: Growing receptive field

- Example (single dimension):

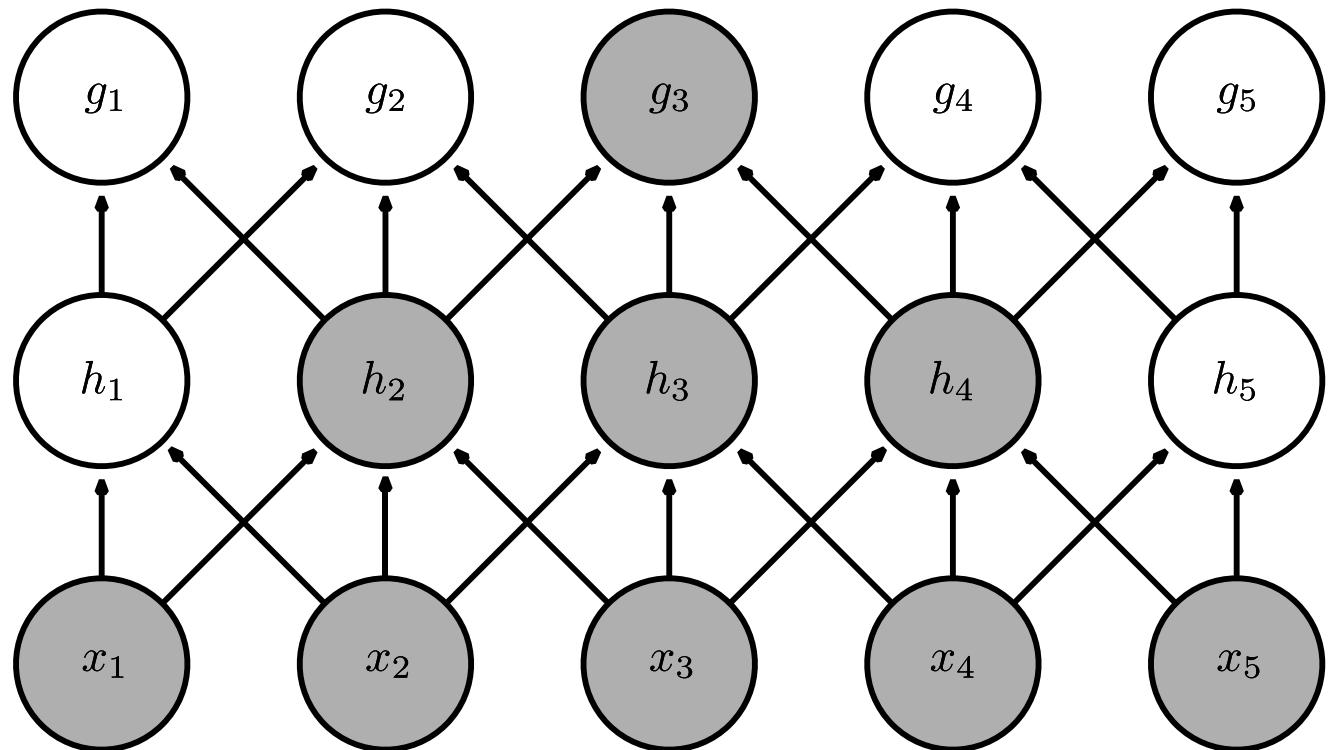
2nd hidden layer

filter size 3

1st hidden layer:
Receptive field size 3

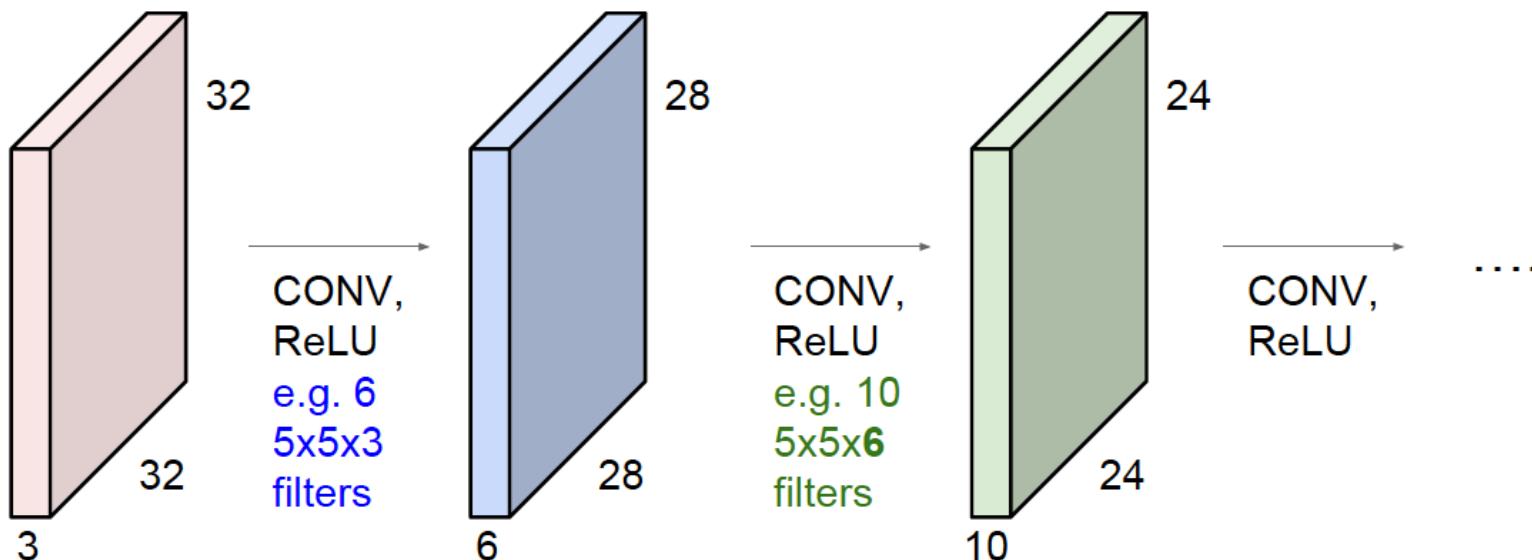
filter size 3

Input layer:
Receptive field size 5



Convolutional neural network

- A convolutional neural network is a sequence of convolution layers interspersed with activation functions:

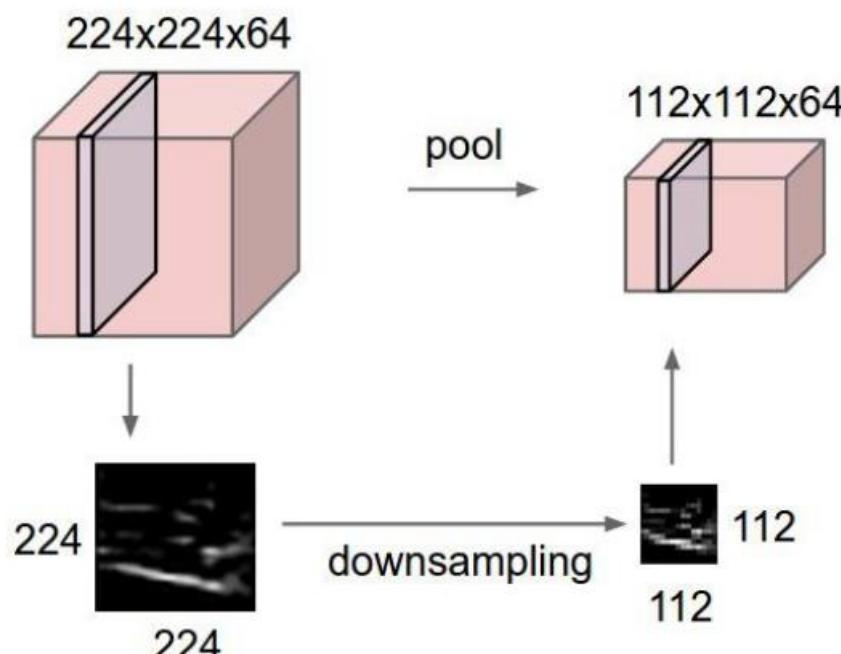


Pooling / subsampling

Motivation:

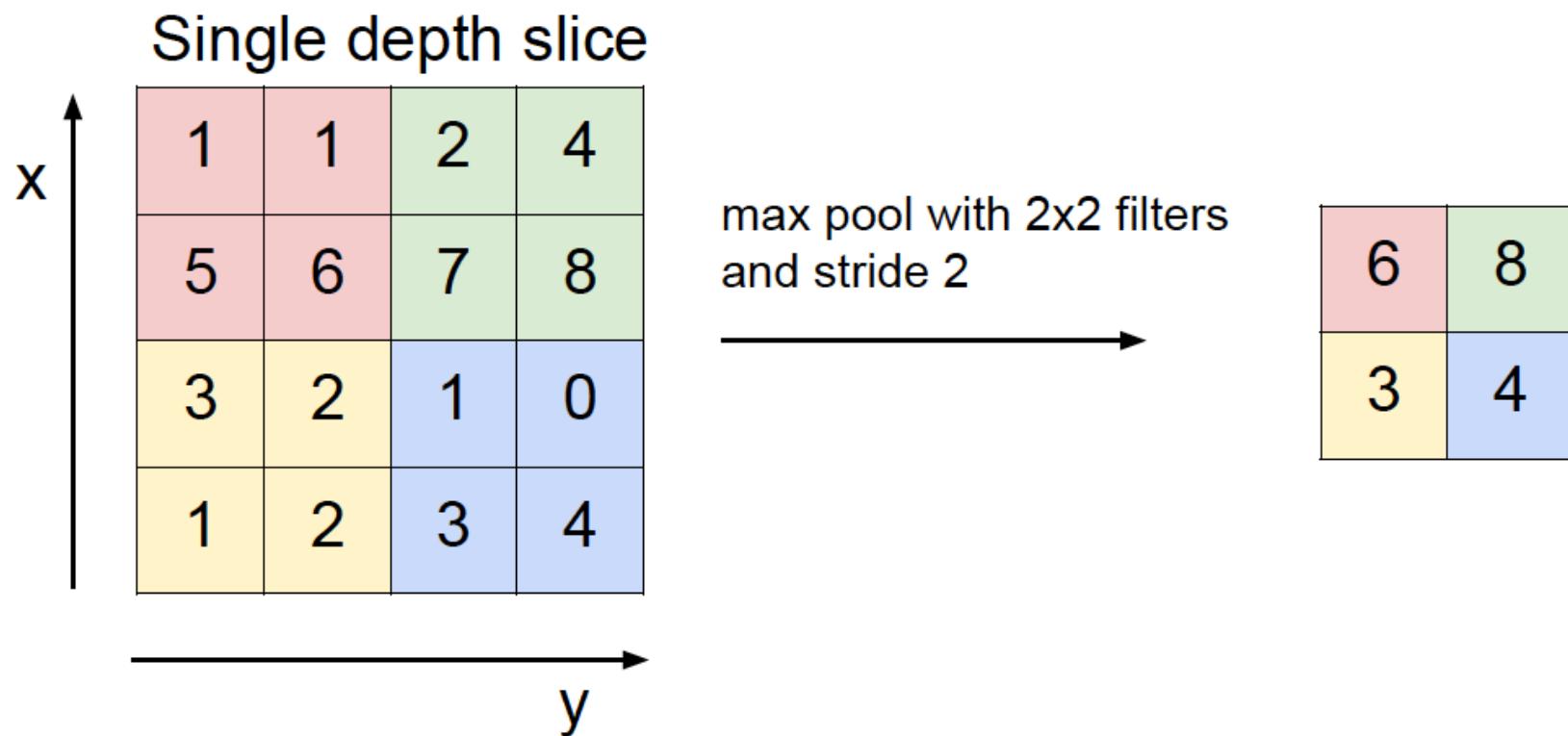
- Make representation **smaller** / reduce parameters
- Increase **size of receptive field** (for *classification* tasks)
- Helps representation to be **invariant to slight translations** of the input
 - If we care about whether a feature is present (not precisely where)

Operates over each activation map independently!

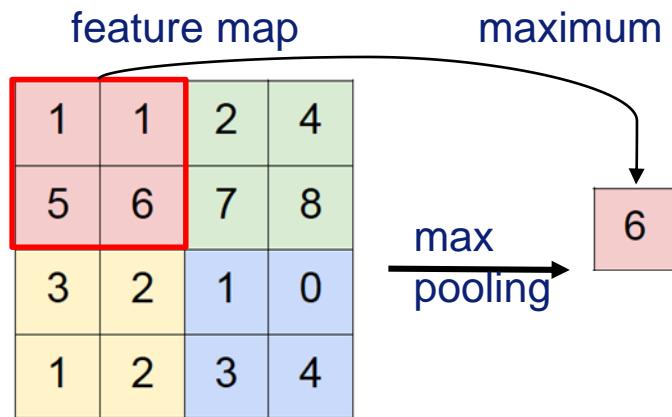


Pooling / subsampling

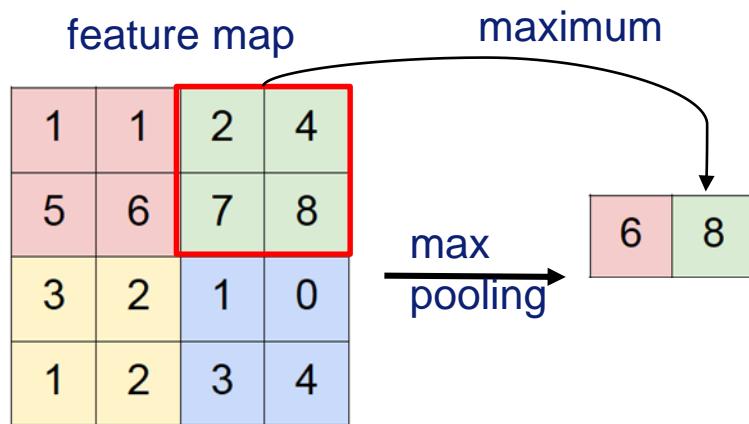
- Types of pooling:
 - Max pooling: $y = \max\{x_1, x_2, \dots x_k\}$
 - Average pooling: $y = \text{mean}\{x_1, x_2, \dots x_k\}$ / sum pooling $y = \text{sum}\{x_1, x_2, \dots x_k\}$
- Example max pooling:



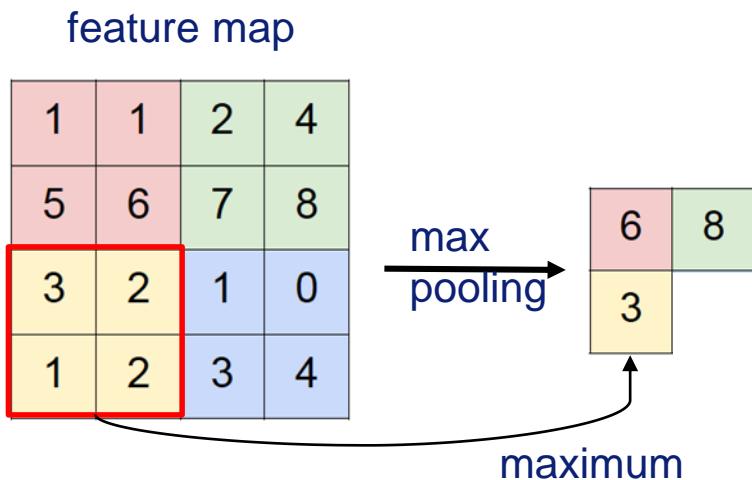
Pooling / subsampling: Illustration of max pooling



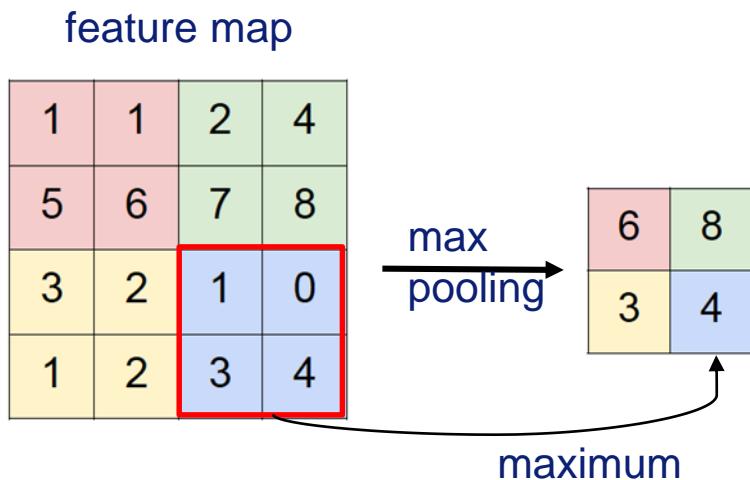
Pooling / subsampling: Illustration of max pooling



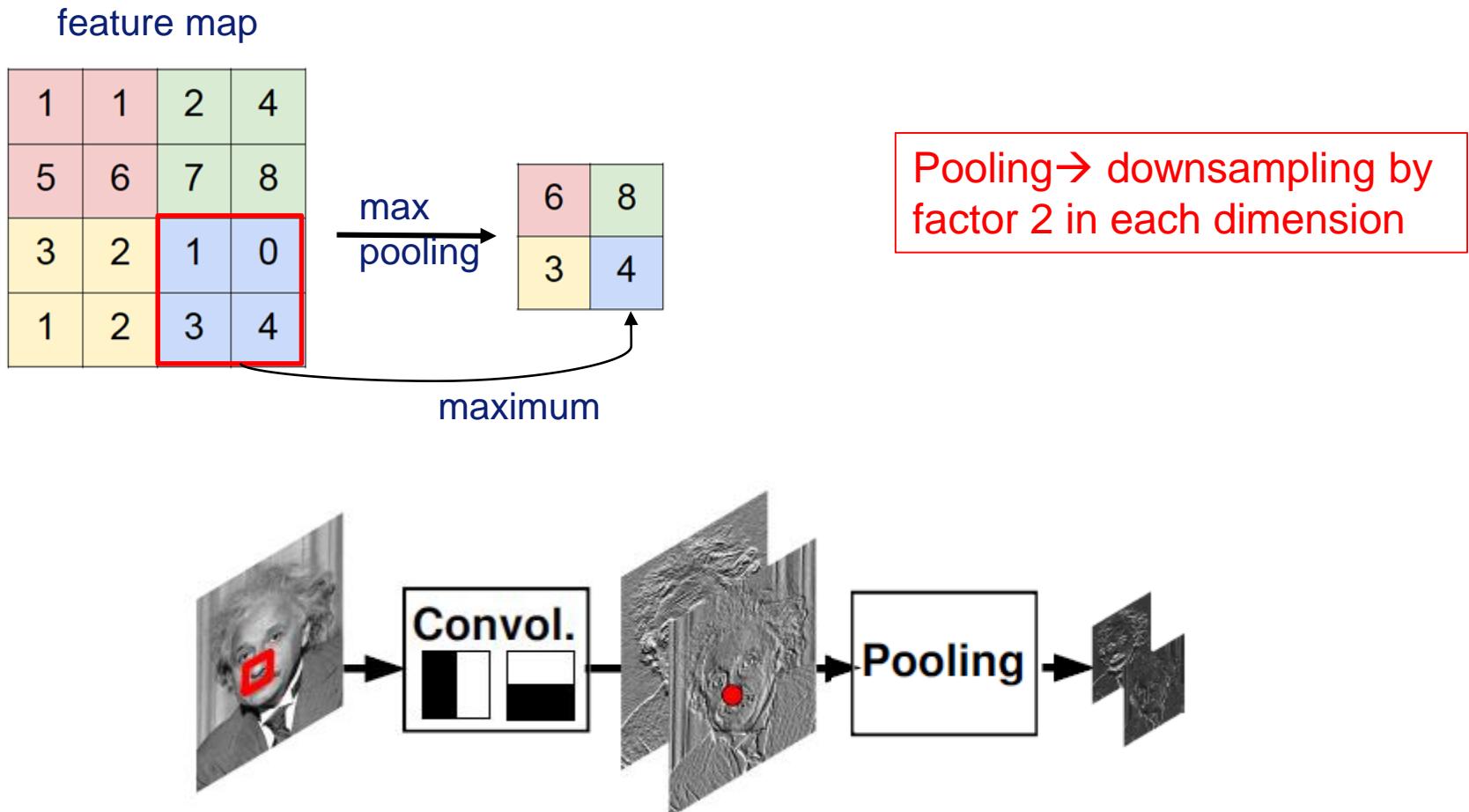
Pooling / subsampling: Illustration of max pooling



Pooling / subsampling: Illustration of max pooling

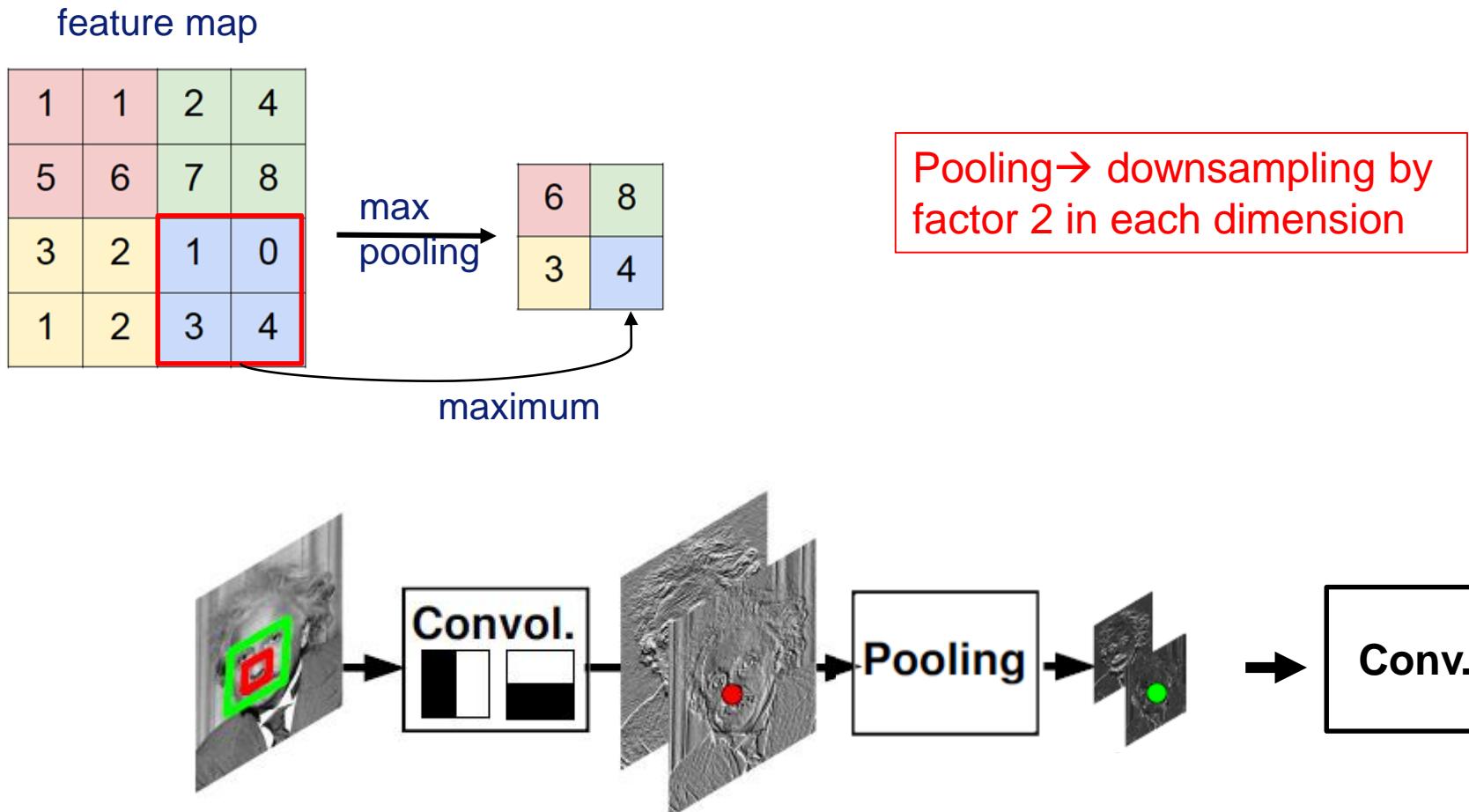


Pooling / subsampling: Illustration of max pooling



- Each unit in first layer is influenced by small input region (filter size)

Pooling / subsampling: Illustration of max pooling



- Layers after pooling: Each unit is influenced by a larger input image region

Pooling / subsampling

A pooling layer

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires three hyperparameters:
 - their spatial extent F ,
 - the stride S ,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
 - $W_2 = (W_1 - F)/S + 1$
 - $H_2 = (H_1 - F)/S + 1$
 - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

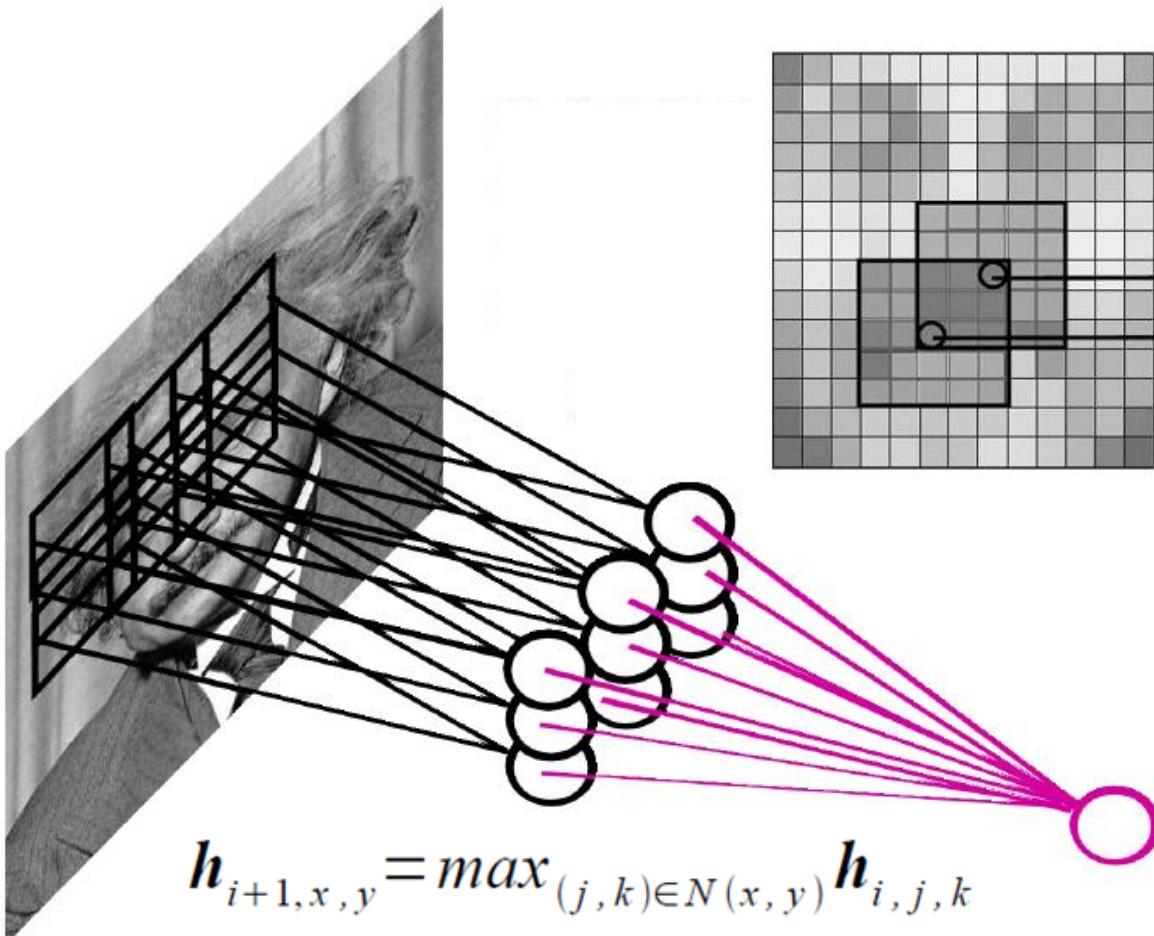
Common settings:

$F = 2, S = 2$

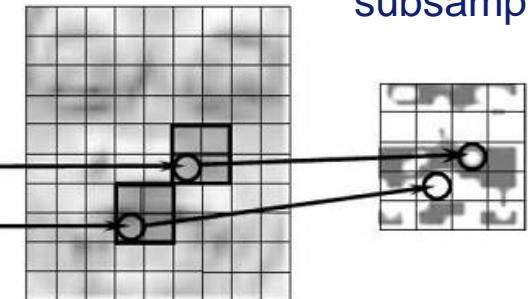
$F = 3, S = 2$

Pooling / subsampling: Illustration

- **Pooling** computes maximal feature response in neighborhood
 - Pooling performed in (mostly) non-overlapping neighborhoods (subsampling)



convolution → pooling /
subsampling

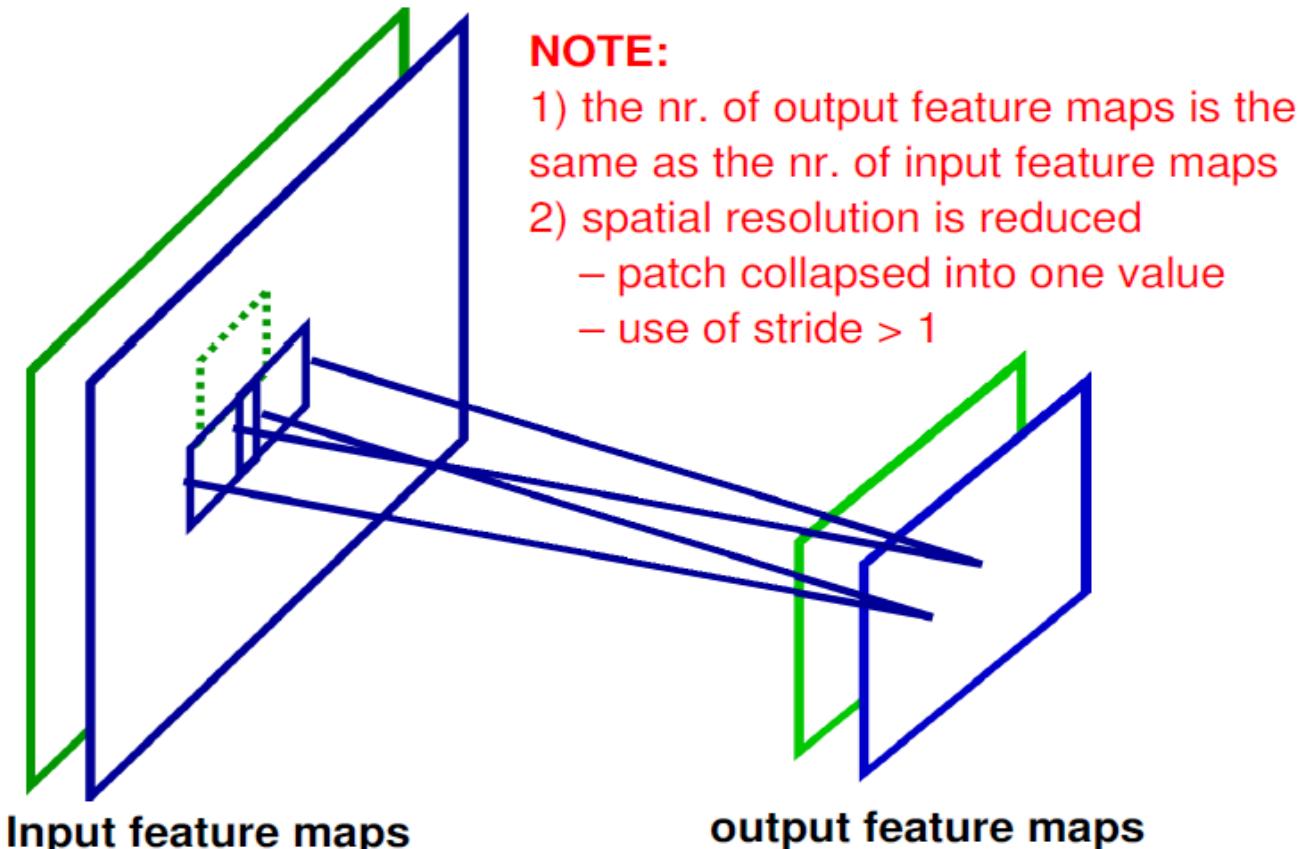


max pooling in 2×2
non-overlapping neighborhood
→ 5×4 pooling feature map

„max pooling“

Pooling / subsampling

- Introduces invariance to local translations
- Reduces number of hidden units in (next) hidden layer



- A similar effect can be achieved by using a stride > 1 instead of pooling!

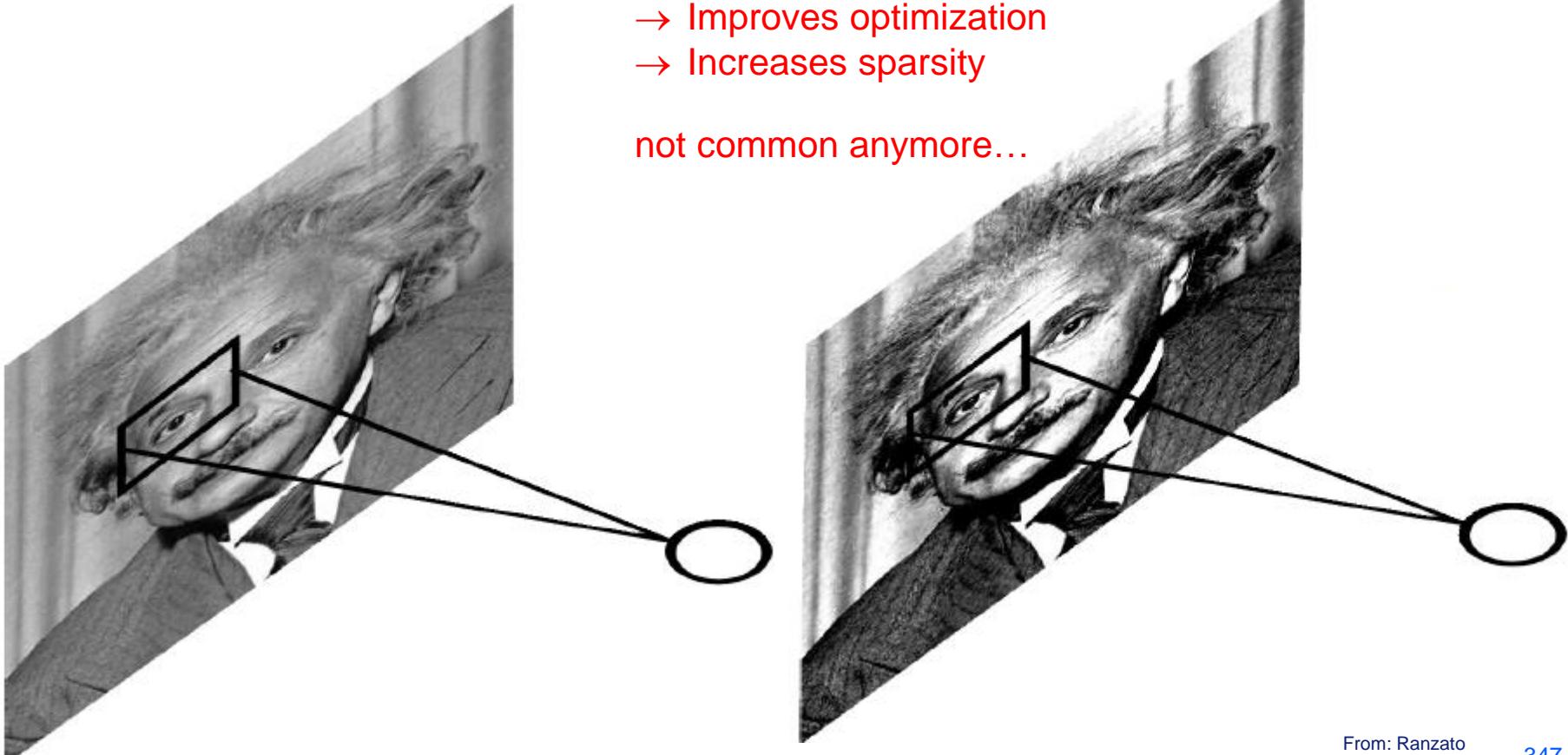
Local contrast normalization (LCN)

- Different contrast should yield the same response
- Local contrast normalization

$$h_{i+1,x,y} = \frac{h_{i,x,y} - m_{i,N(x,y)}}{\sigma_{i,N(x,y)}}$$

- Improves invariance
- Improves optimization
- Increases sparsity

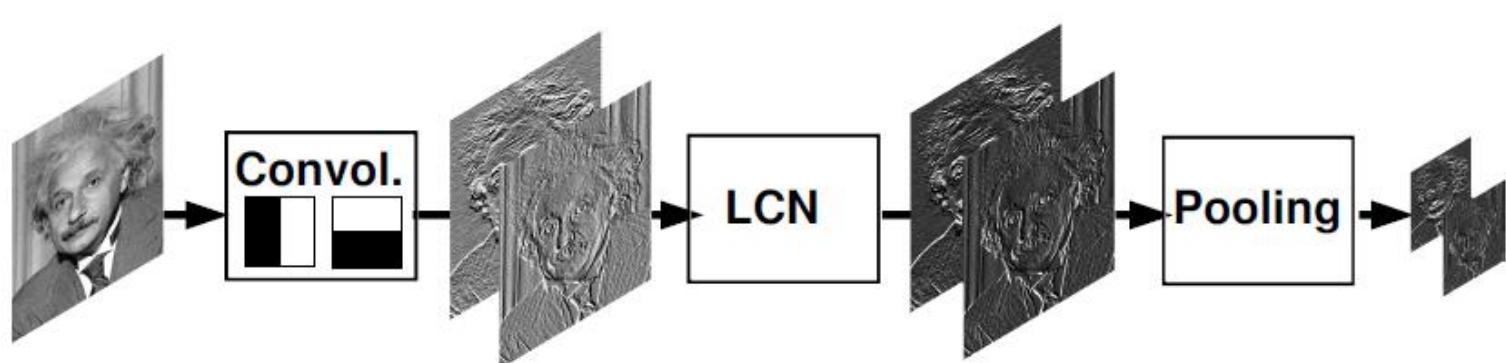
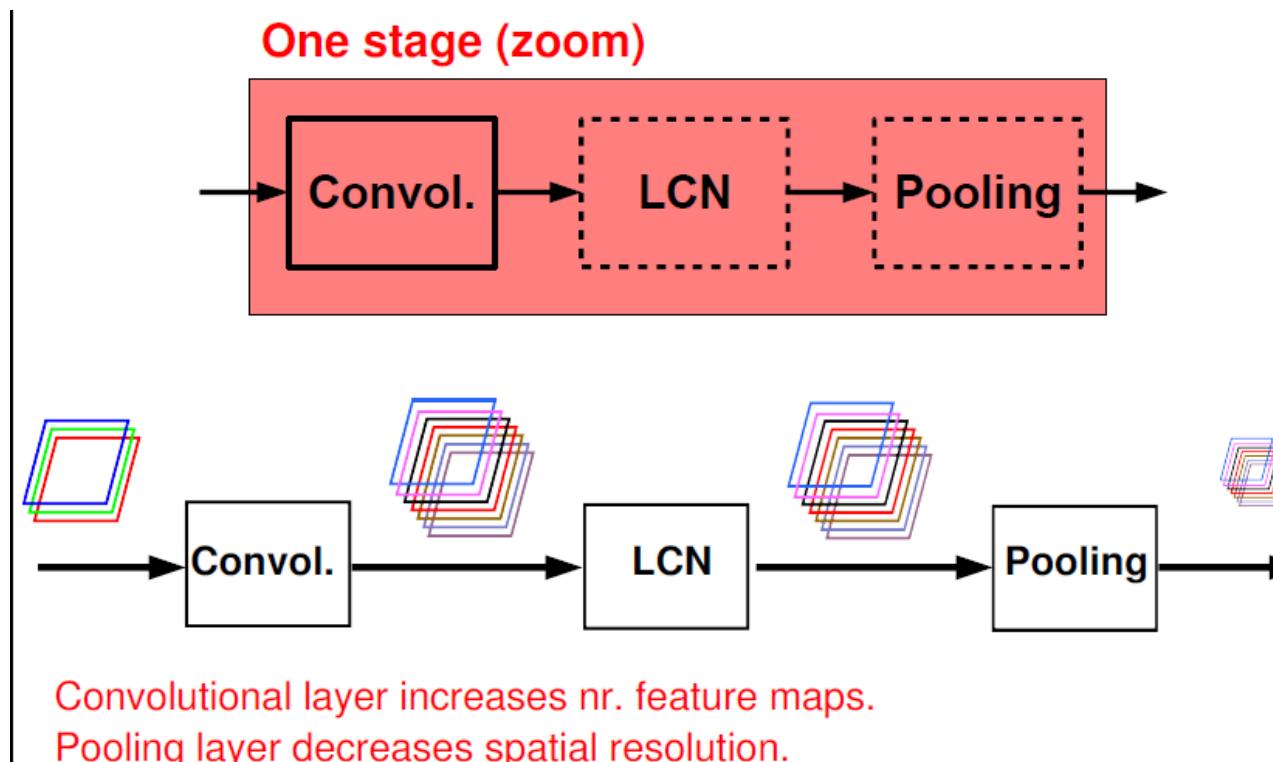
not common anymore...



From: Ranzato

347

Convolutional neural networks: Typical architecture

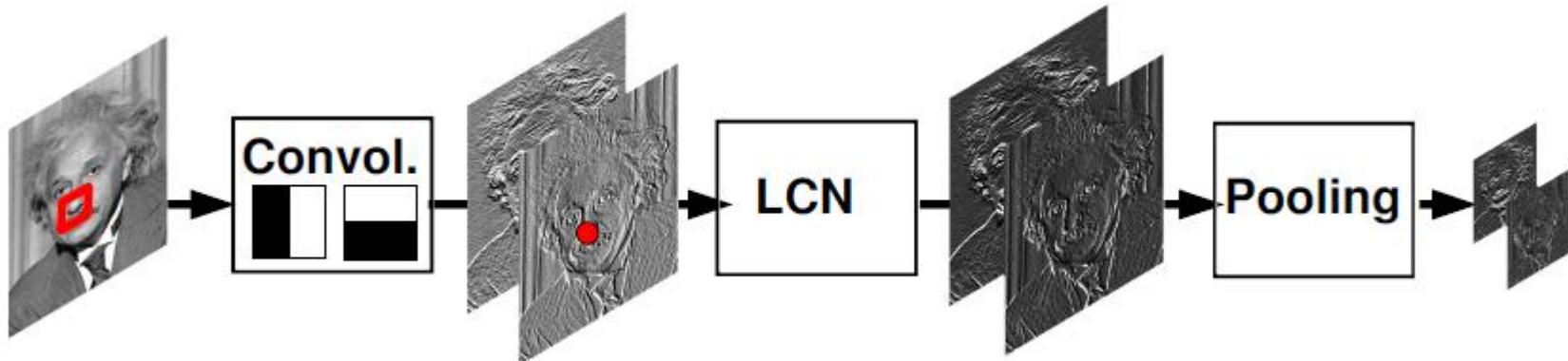


Example with only two filters.

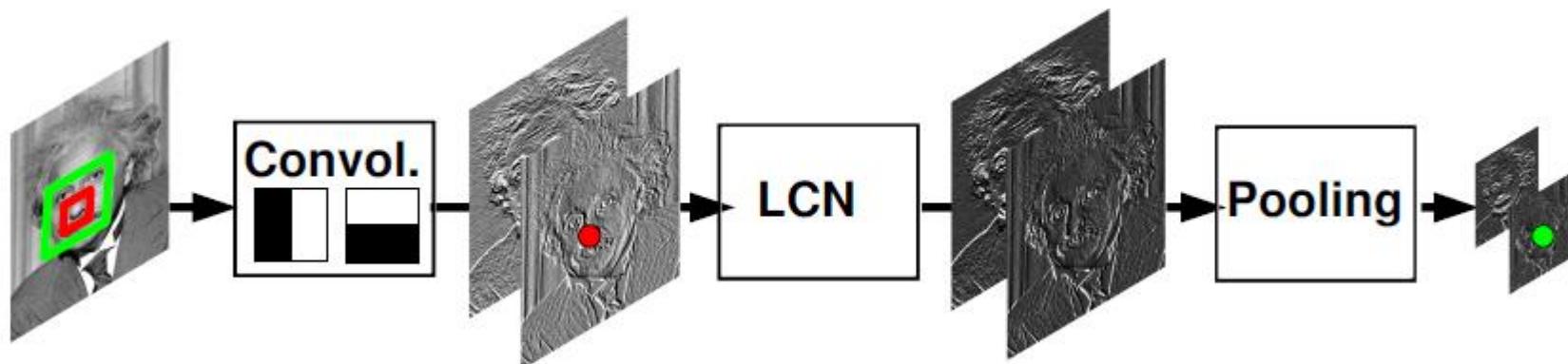
From: Ranzato

348

Convolutional neural networks: Receptive field



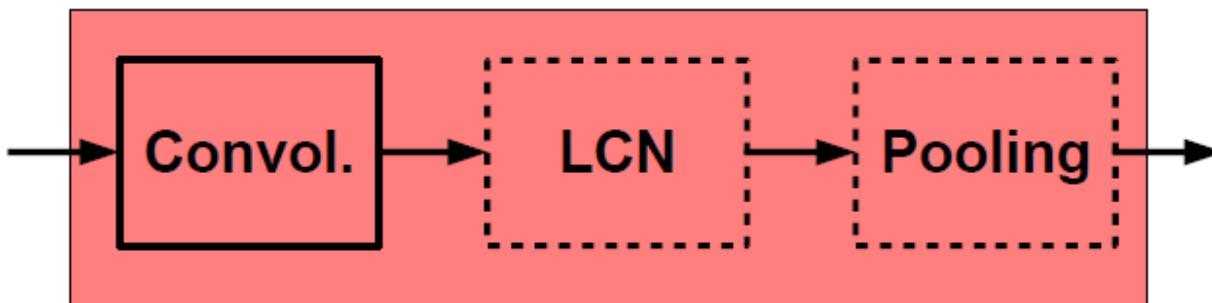
A hidden unit in the first layer is influenced by a small neighborhood (equal to size of filter)



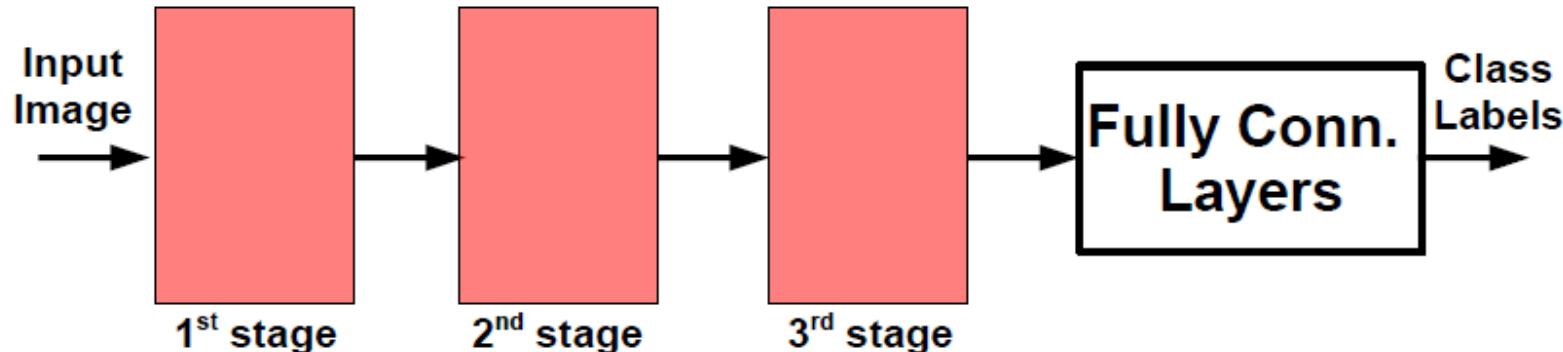
A hidden unit after the pooling layer is influenced by a larger neighborhood (depending on filter sizes and strides)

Convolutional neural networks with multiple convolution layers

One stage (zoom)



Whole system



- After a few stages: very low residual spatial resolution
→ We have learned a descriptor for the whole image

Normalization layer: Different variants

- Local contrast normalization (not used anymore)
 - Normalize values in local neighborhood of single feature map (see before)
- Local response normalization (not used anymore)
 - Normalize activations at given position across feature maps
 - Implements „lateral inhibition“, creating competition for big activities
- Batch normalization
- Layer normalization
- Instance normalization
- Group normalization

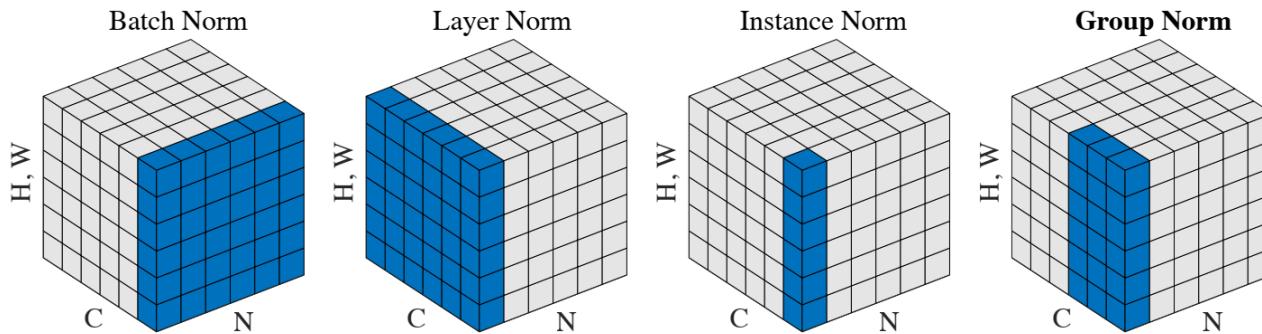
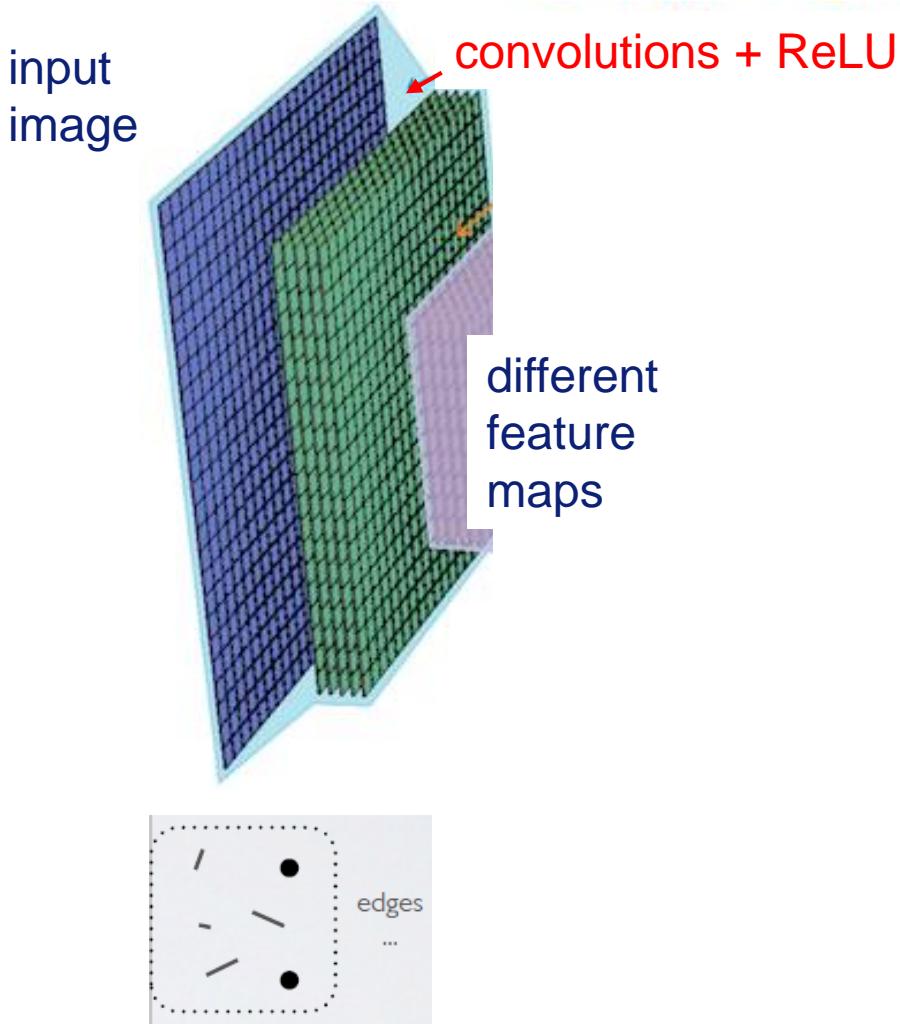


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

Convolutional neural network (CNN)

optional

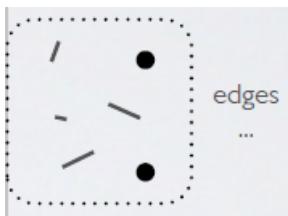
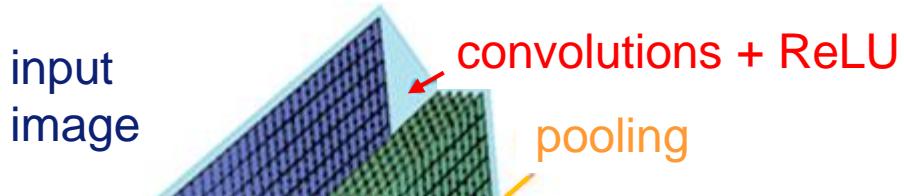
- Multiple convolution layers interspersed with pooling / normalization layers



Convolutional neural network (CNN)

optional

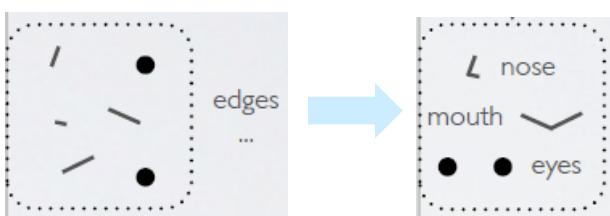
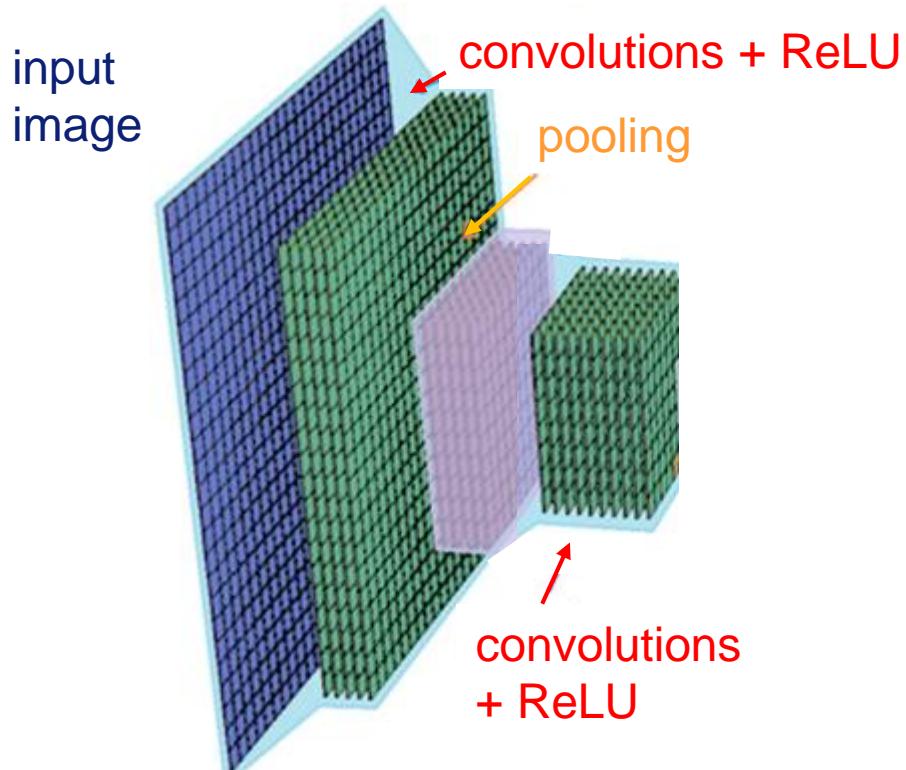
- Multiple convolution layers interspersed with pooling / normalization layers



Convolutional neural network (CNN)

optional

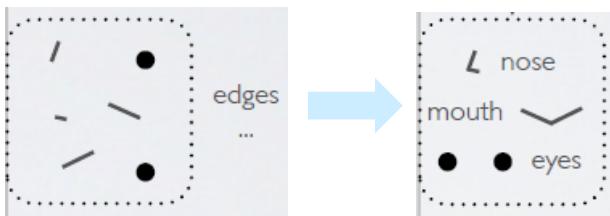
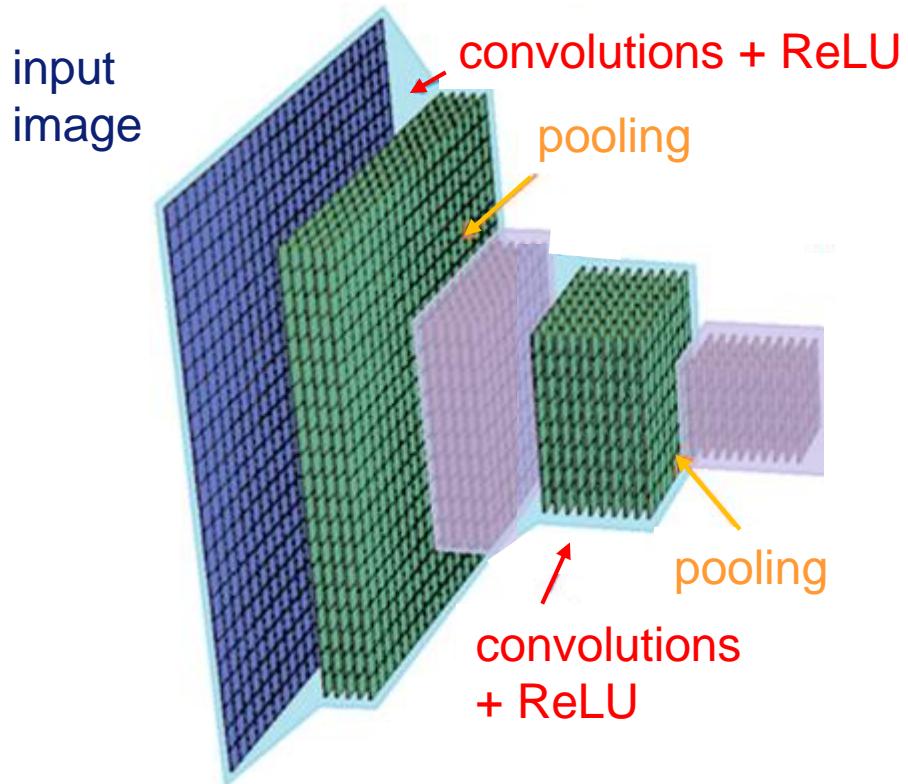
- Multiple convolution layers interspersed with pooling / normalization layers



Convolutional neural network (CNN)

optional

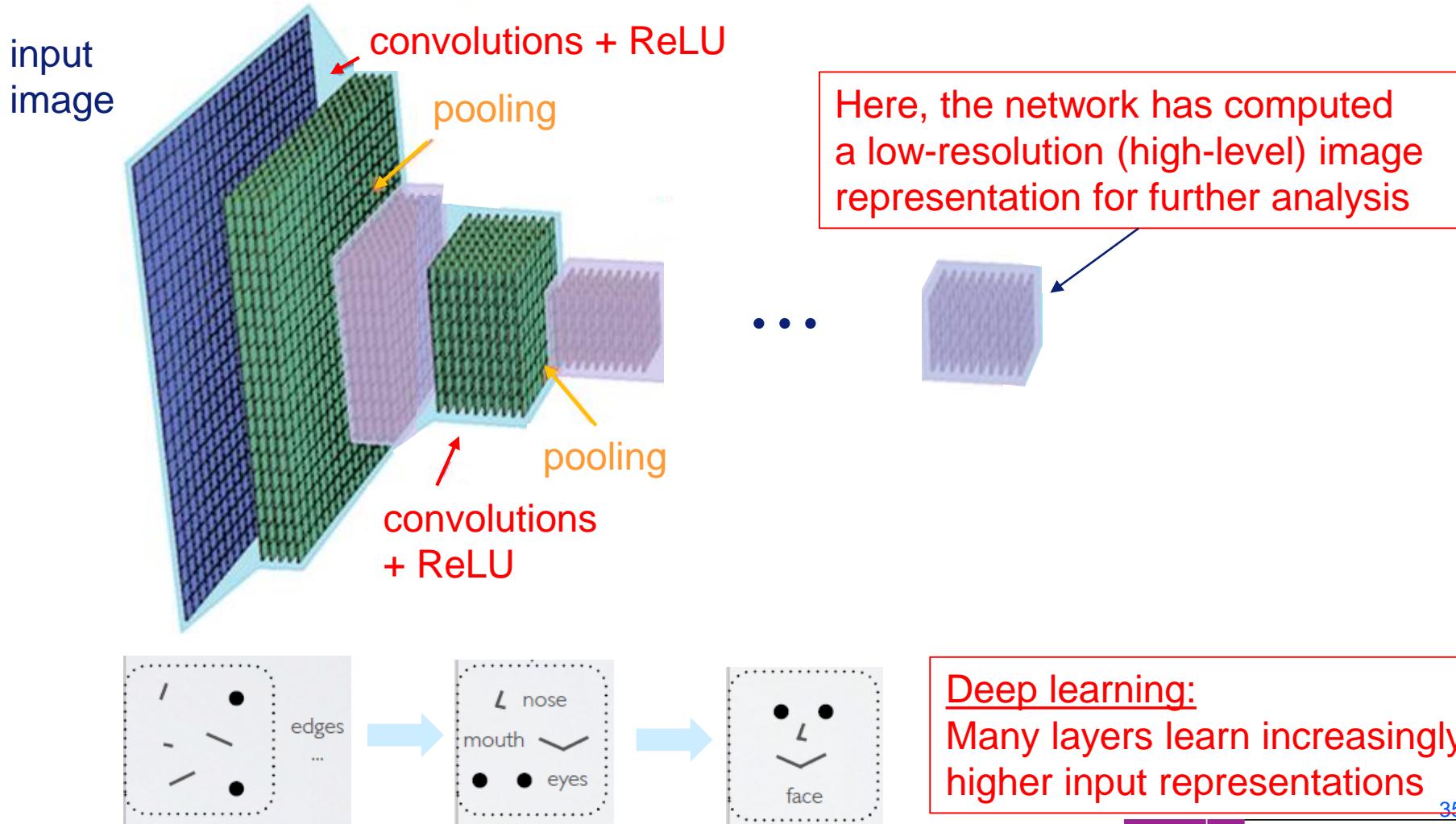
- Multiple convolution layers interspersed with pooling / normalization layers



Convolutional neural network (CNN)

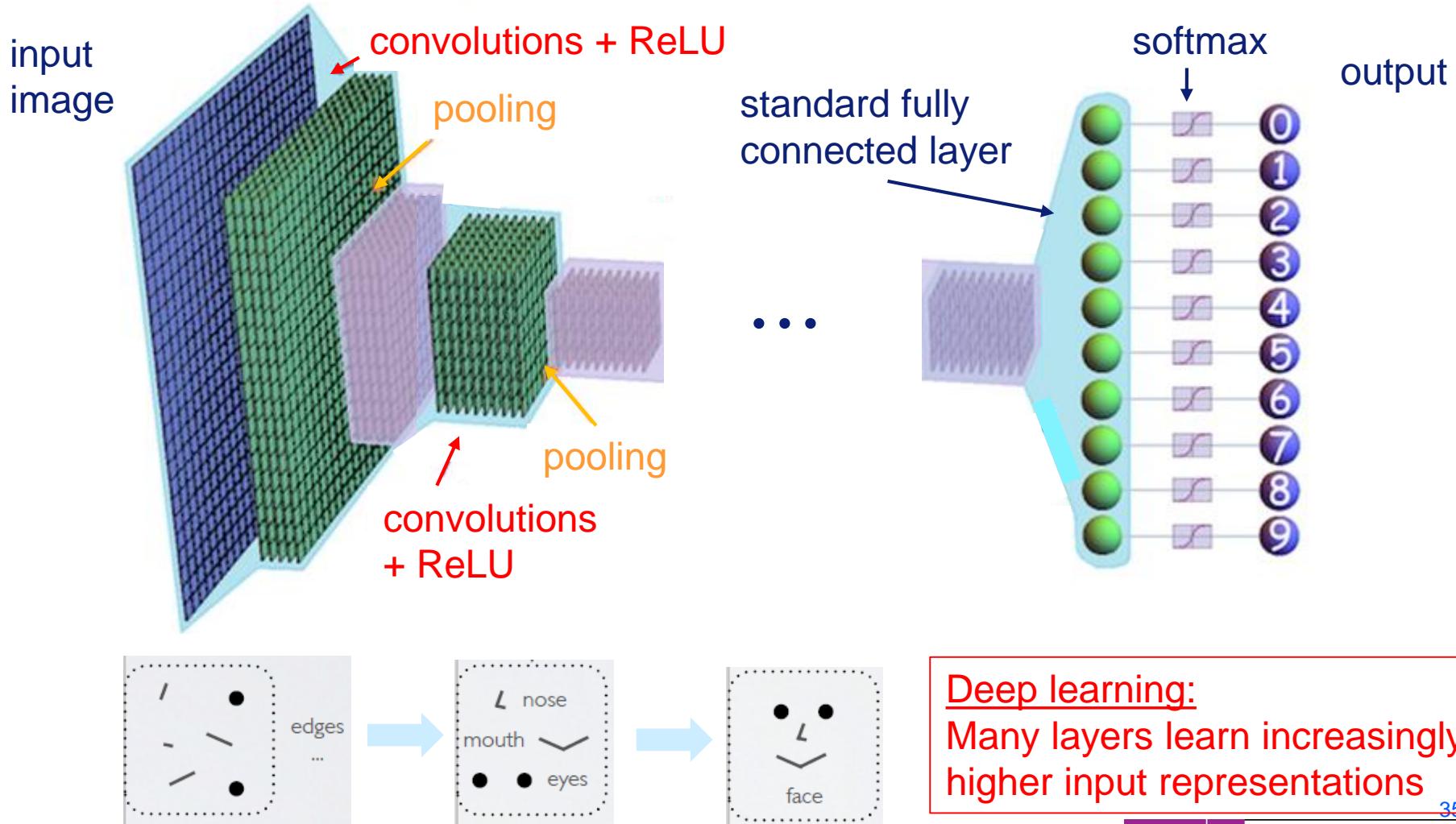
optional

- Multiple convolution layers interspersed with pooling / normalization layers



Convolutional neural network (CNN)

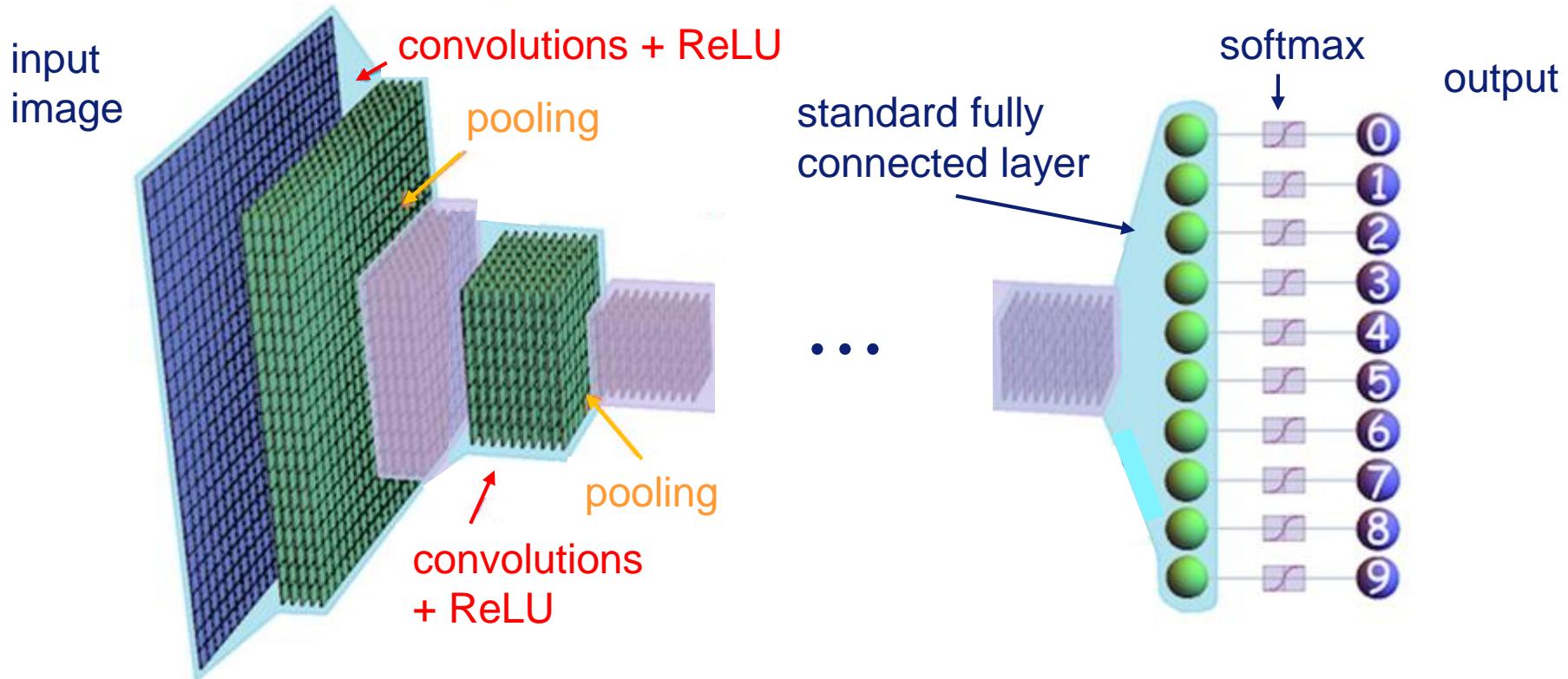
- Multiple convolution layers interspersed with pooling / normalization layers
- At the end: E.g. fully connected layer(s) for image classification



Convolutional neural network (CNN)

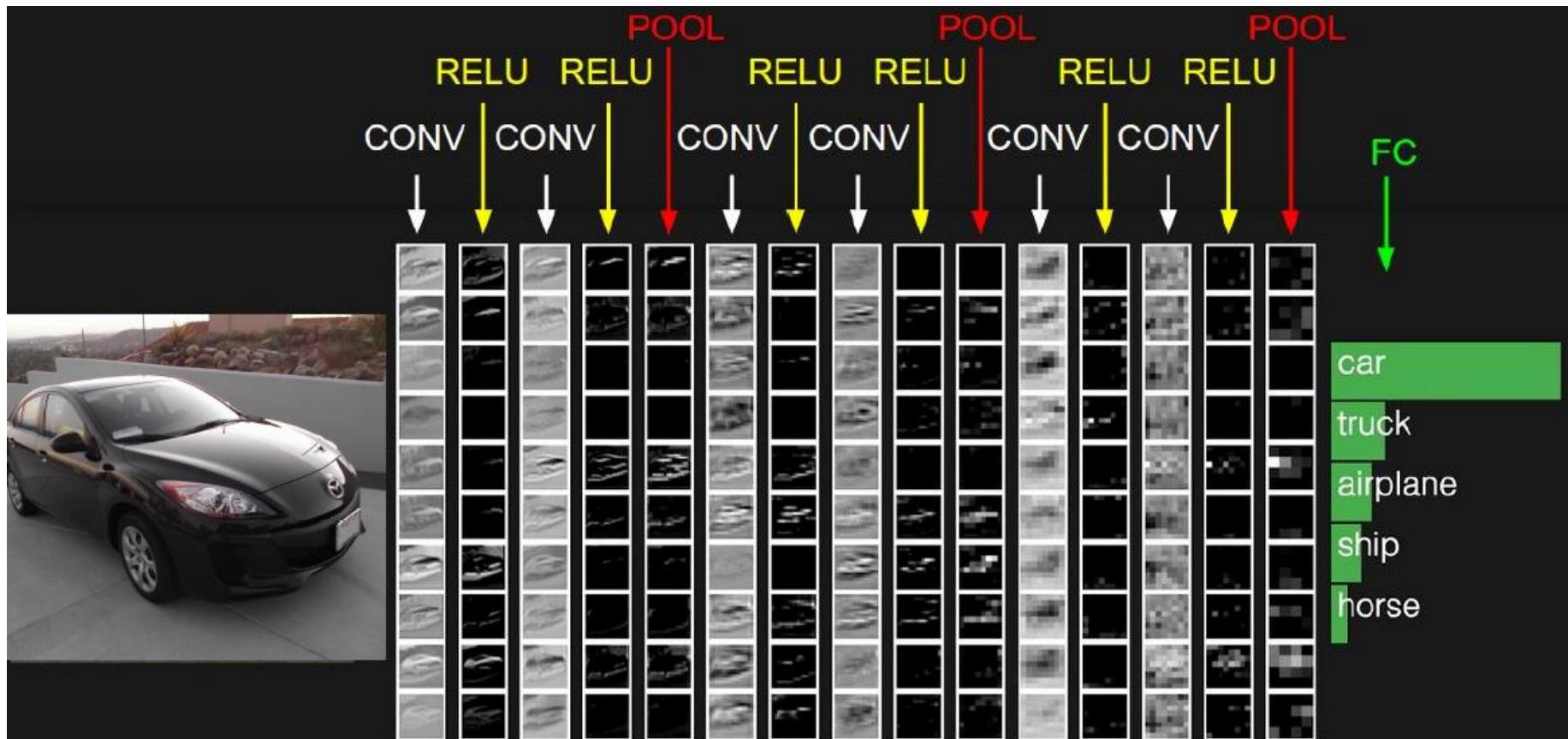
optional

- Multiple convolution layers interspersed with pooling / normalization layers
- At the end: E.g. fully connected layer(s) for image classification



- *Learn appropriate hierarchy of features for given problem (backpropagation)*
 - Instead of using hand-crafted features requiring domain expertise

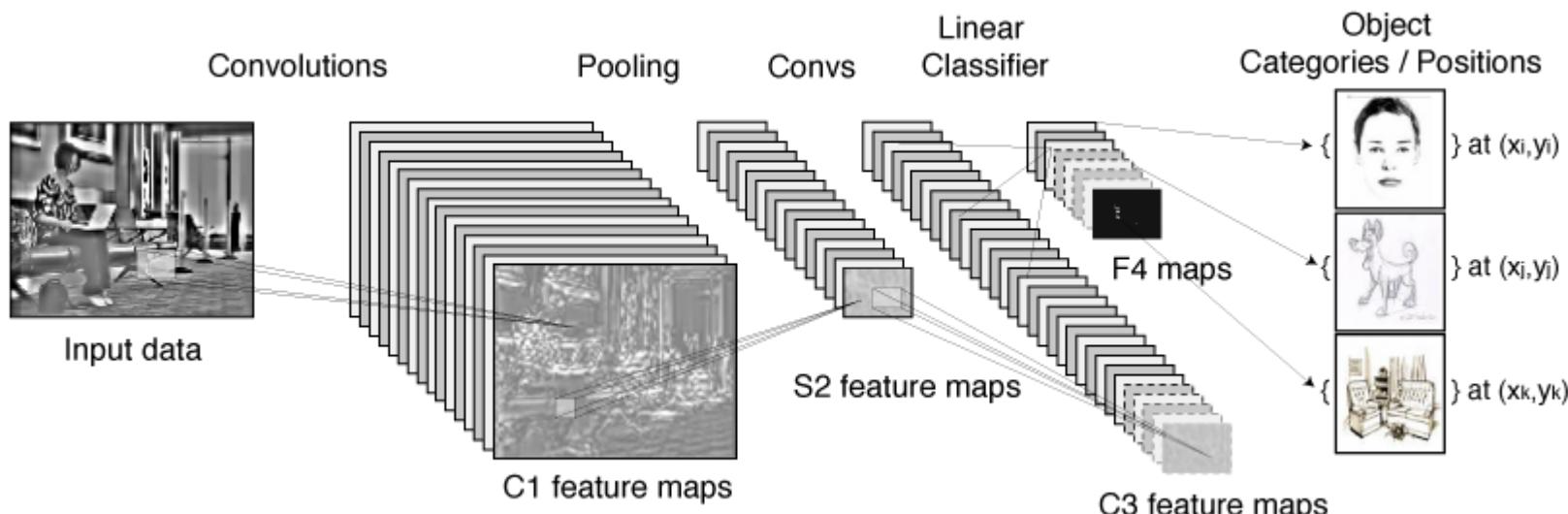
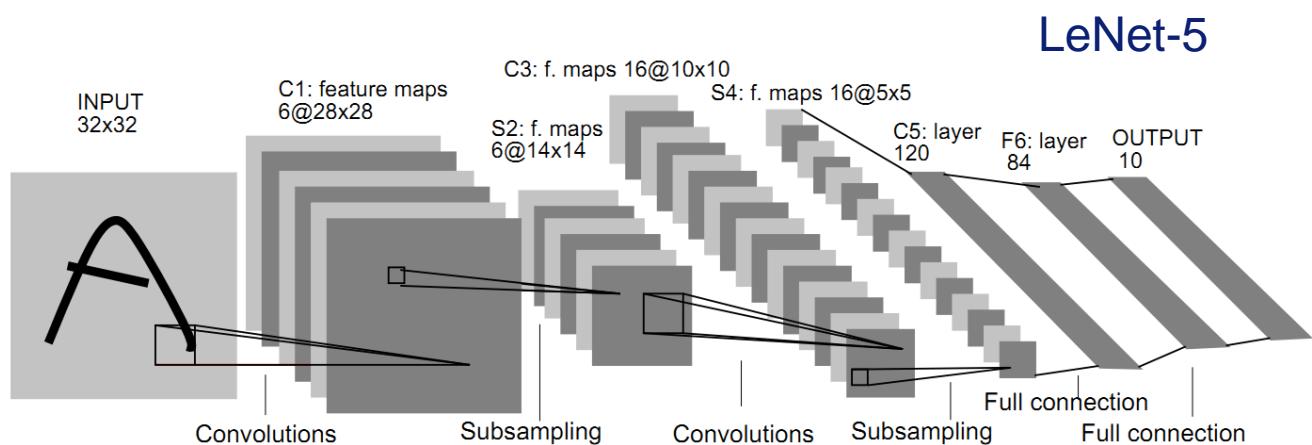
Convolutional neural network: Illustration



Convolutional neural network: Example

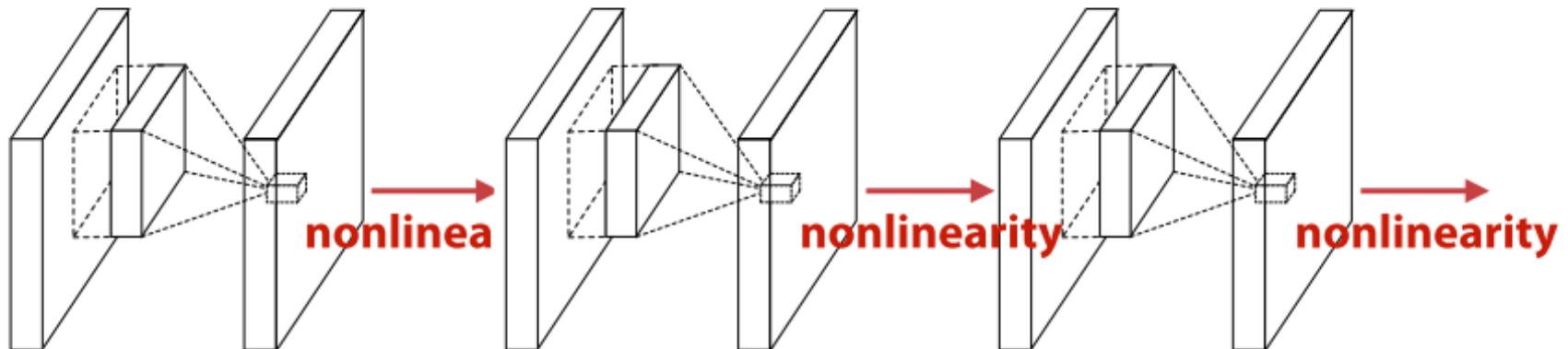
Alternating between convolutional and pooling layers

Note: Input image with **fixed size** due to fully connected layers!



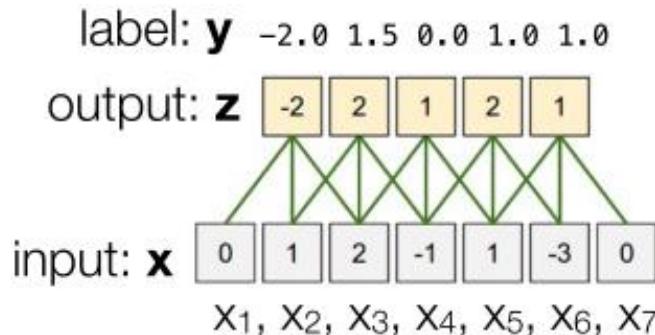
Convolutional neural networks: Training

key idea: **learn convolutional kernels for features extraction**



- Learning of kernel (filter) weights:
 - **Backpropagation** (similarly as in MLP), stochastic gradient descent
 - Need to pass gradients through convolution and pooling layers
 - Equations written as matrix multiplication (similarly to forward pass)
 - ReLU activation function accelerates training and avoids saturation issues
- Trainings usually „large scale“; examples:
 - Computer vision (AlexNet, 1000 classes): trained on 1.2M images (\approx 1 week) on 2 GPUs (650.000 neurons, 60.000.000 params, 630.000.000 connections)
 - Automatic speech recognition (Microsoft): training e.g. on 2000h of data, needed 59 days on GPUs (!) (ICML 2011)

Convolutional neural networks: 1-dim. backpropagation example



Single filter (no bias):

weights $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ 

linear activation function

- forward propagation: $\mathbf{z}^T = \mathbf{w}^T \cdot \text{im2col}(\mathbf{x}) = (w_1 \quad w_2 \quad w_3) \cdot \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ x_2 & x_3 & x_4 & x_5 & x_6 \\ x_3 & x_4 & x_5 & x_6 & x_7 \end{pmatrix}$

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ x_2 & x_3 & x_4 & x_5 & x_6 \\ x_3 & x_4 & x_5 & x_6 & x_7 \end{pmatrix}$$

- $\frac{\partial \mathbf{z}^T}{\partial \mathbf{w}} = \text{im2col}(\mathbf{x}) = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \\ x_2 & x_3 & x_4 & x_5 & x_6 \\ x_3 & x_4 & x_5 & x_6 & x_7 \end{pmatrix}$

Weight update

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \eta \cdot \nabla_{\mathbf{w}} L(\mathbf{w})$$

$$\text{Chain rule: } \frac{\partial L}{\partial \mathbf{w}^T} = \frac{\partial L}{\partial \mathbf{z}^T} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{w}^T}$$

- SSE loss: $L(\mathbf{w}, \mathbf{y}, \mathbf{z}) = \frac{1}{2} \sum_{i=1}^5 (z_i(\mathbf{w}) - y_i)^2 \Rightarrow \frac{\partial L}{\partial z_i} = (z_i - y_i)$

- $\frac{\partial L}{\partial \mathbf{w}^T} = \left(\frac{\partial L}{\partial w_1} \quad \frac{\partial L}{\partial w_2} \quad \frac{\partial L}{\partial w_3} \right) = \left(\frac{\partial L}{\partial z_1} \quad \frac{\partial L}{\partial z_2} \quad \frac{\partial L}{\partial z_3} \quad \frac{\partial L}{\partial z_4} \quad \frac{\partial L}{\partial z_5} \right) \cdot \text{im2col}(\mathbf{x})^T$

matrix multipl.

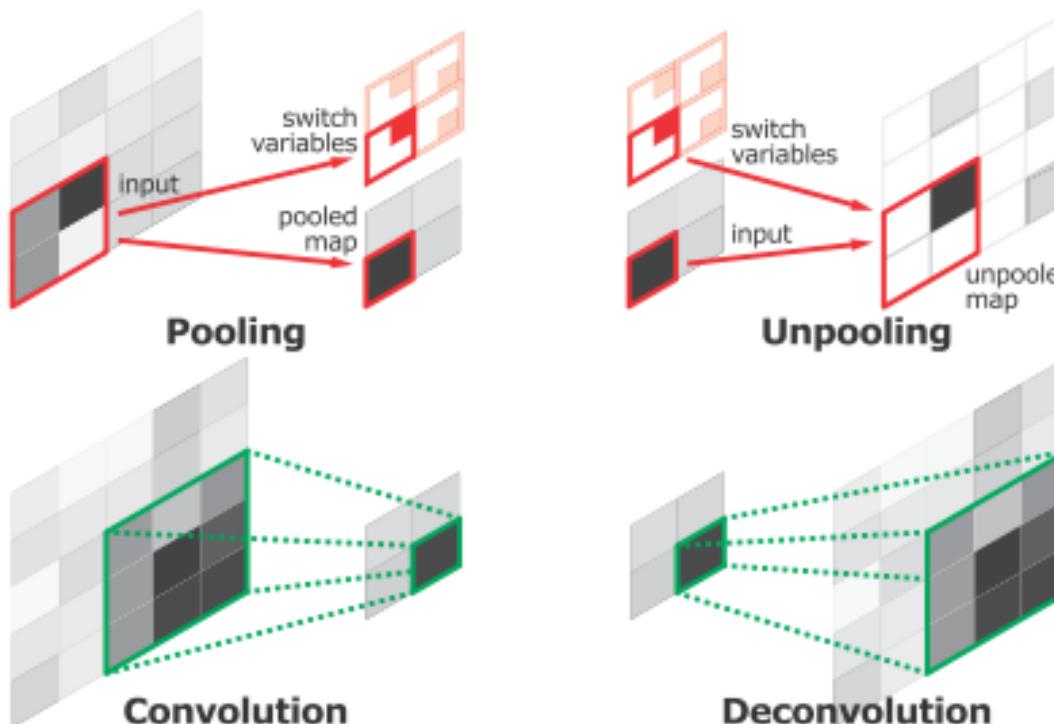
1 x 3

1 x 5

5 x 3

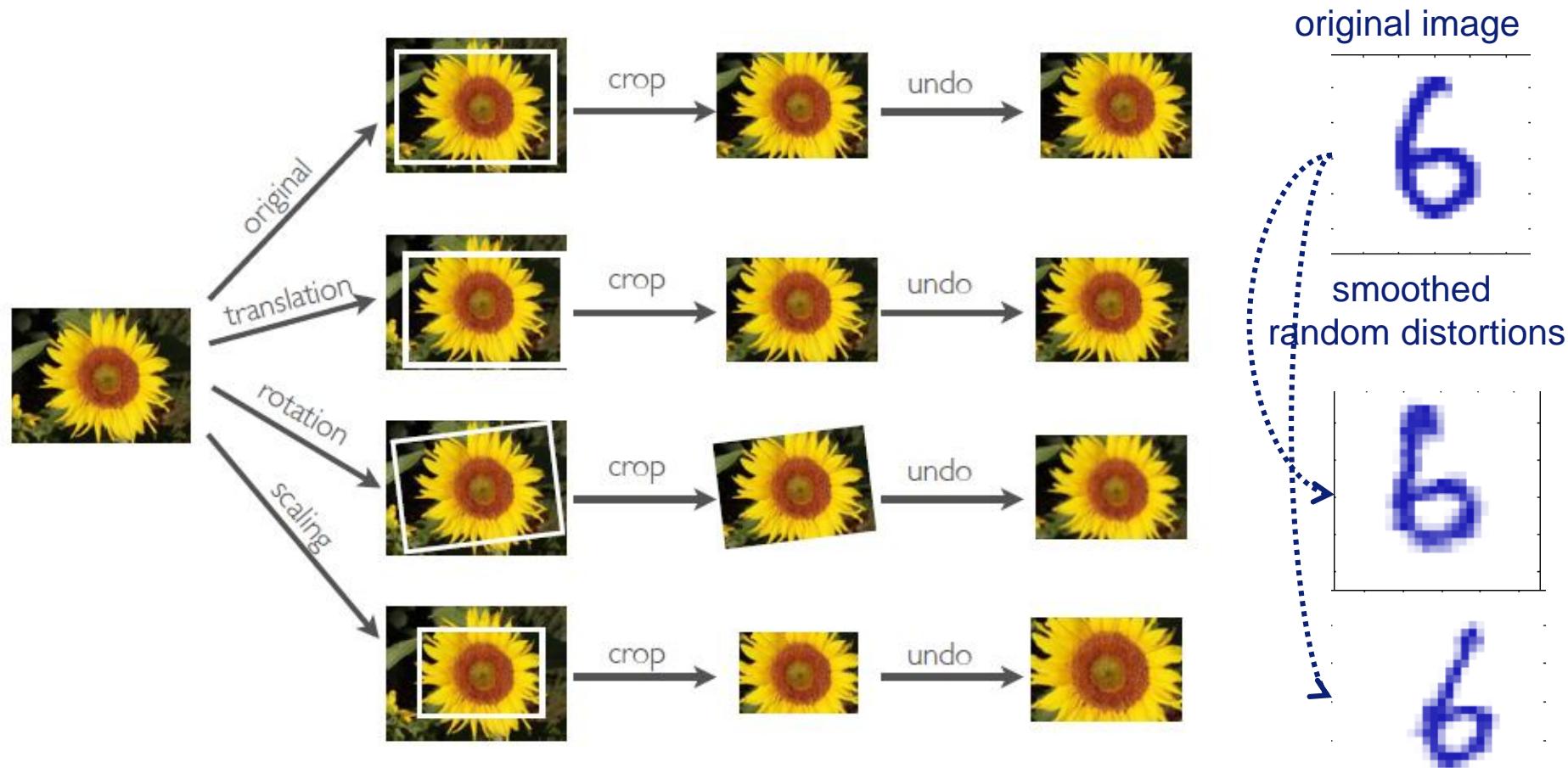
Convolutional neural networks: Backpropagation for pooling layers

- Pooling layers:
 - Error signals for each example are distributed over an upsampled region
 - **Average pooling:** Uniform distribution of error for single pooling unit among the units which feed into it in the previous layer
 - **Max pooling:** Unit which was chosen as the **max** receives all the error, remembered by „switch variables“
 - very small change in input perturbs result only through that unit



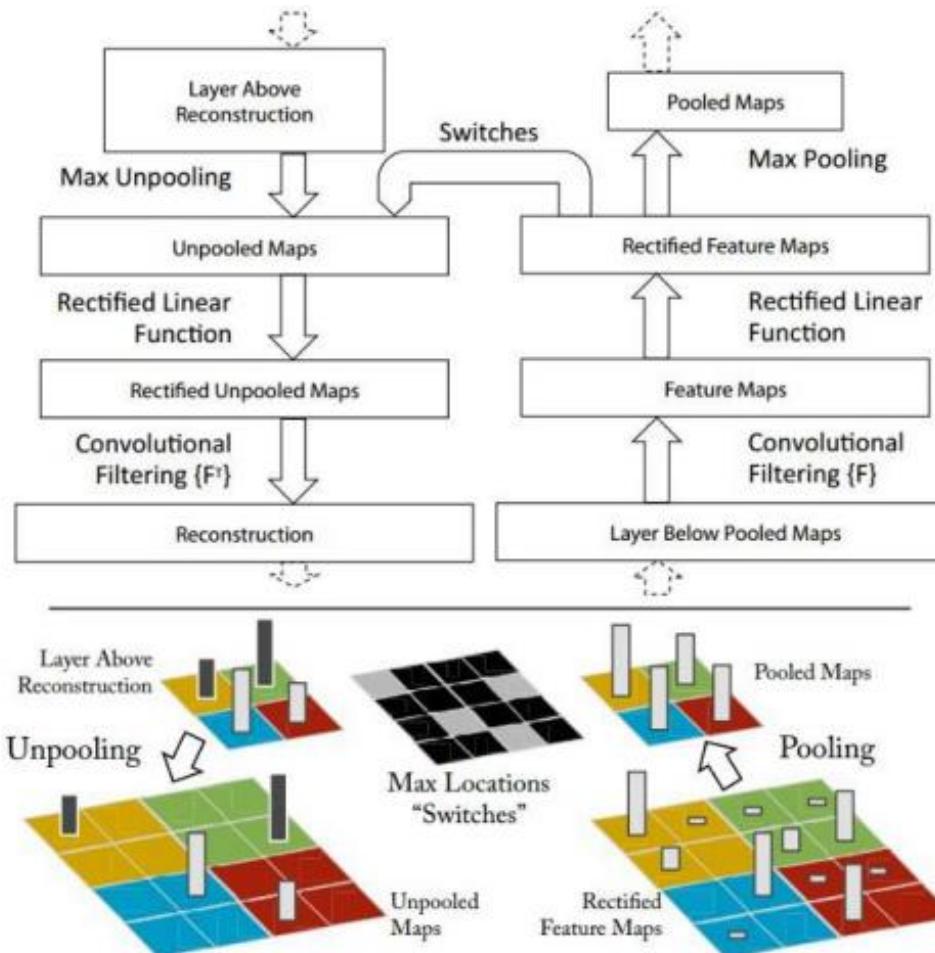
Strategies to avoid overfitting

- Dropout (generally in fully connected layers)
- Weight regularization
- Data augmentation (especially appropriate for image data!)



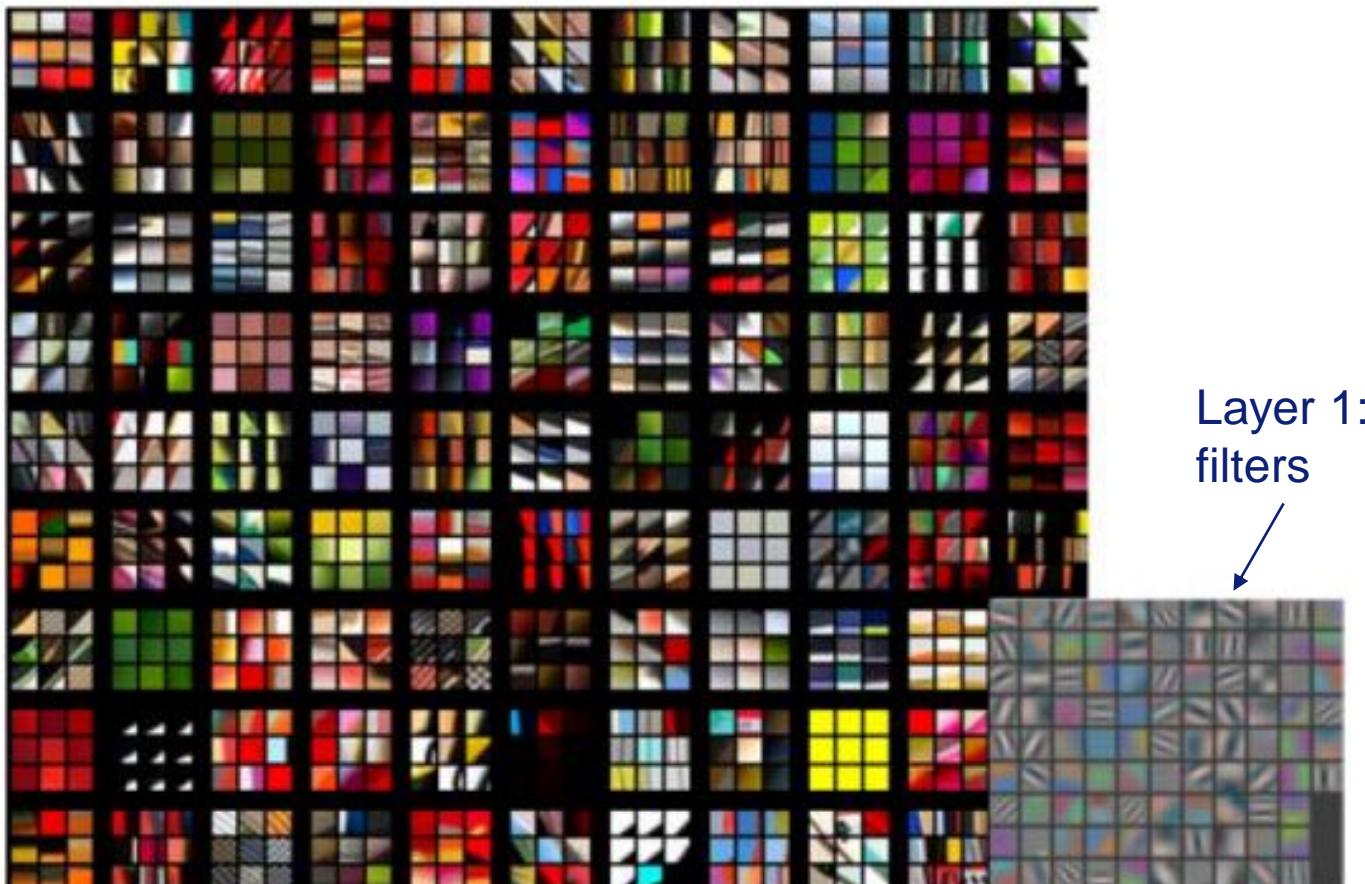
Convolutional neural networks: Visualisation of features

- What input pattern originally caused a given activation in the feature maps?



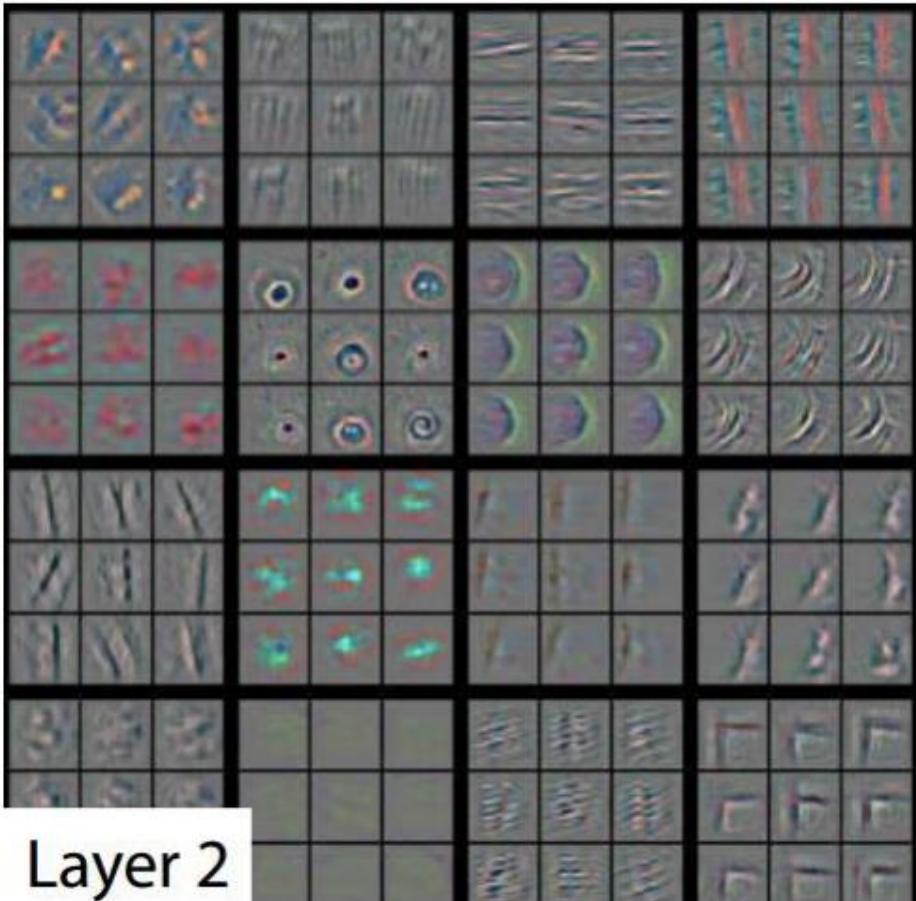
Convolutional neural networks: Visualisation of features

- Main idea: Mapping activations at high layers back to input pixel space
- Show which input patterns caused a given activation in the feature maps
- Layer 1: Top 9 activations (projected down to pixel space by deconvolution)



Convolutional neural networks: Visualisation of features

- Layer 2: Top 9 activations, projected to pixel space... corresponding image patches causing the activation



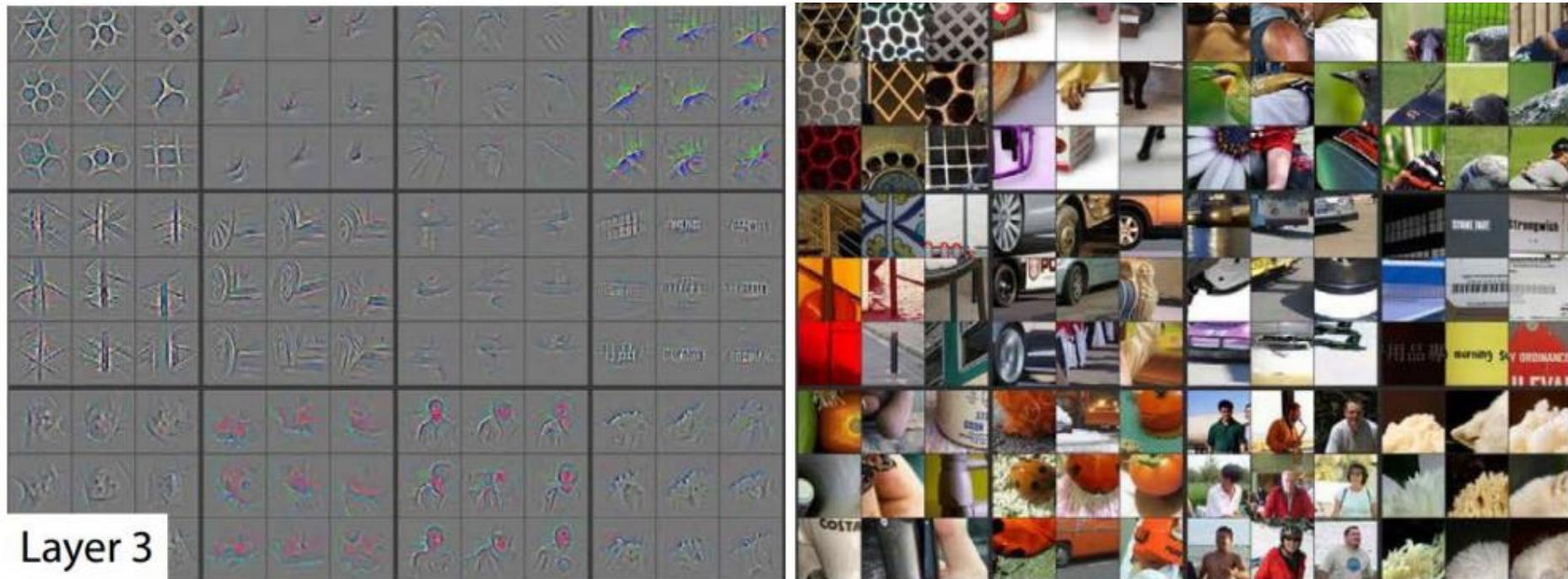
... shown for 16 different randomly selected feature maps from validation set



Layer 2: more complex patterns...

Convolutional neural networks: Visualisation of features

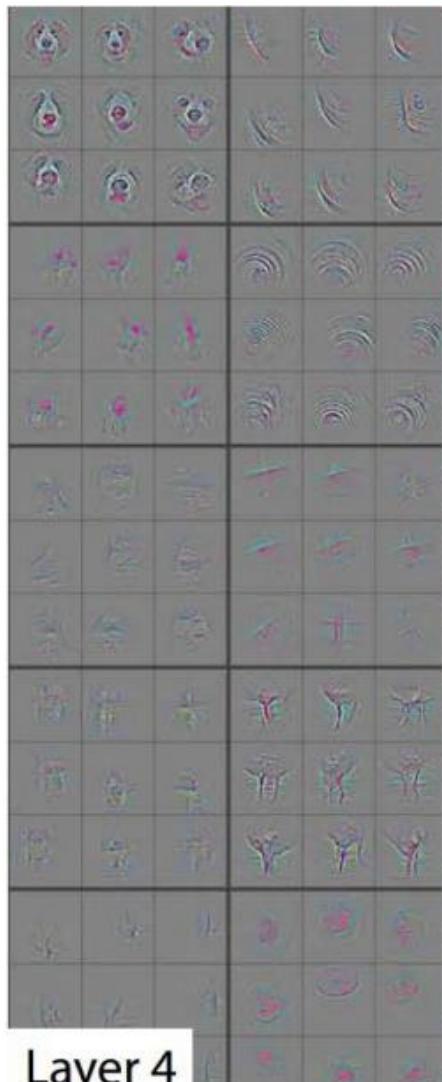
- Layer 3: Top 9 activations



... even more complex patterns and object parts

Convolutional neural networks: Visualisation of features

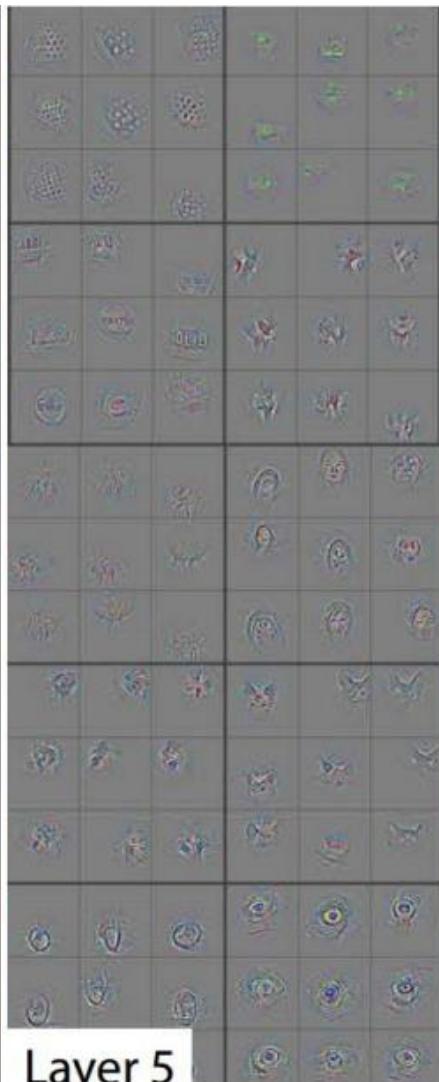
- Layer 4 and 5: Top 9 activations



Layer 4



Layer 5



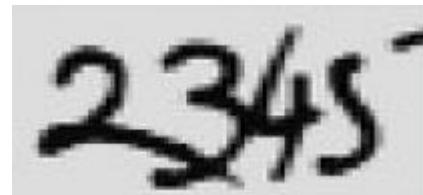
Convolutional networks: Application examples in computer vision

- Image classification (e.g. ImageNet)
- Object detection (e.g. pedestrian detection)
- Optical character recognition, house numbers
- Traffic sign classification
- Texture classification
- Scene parsing
- Segmentation (2D / 3D)
- Action recognition from videos
- Pose estimation
- Image captioning
- Denoising
- ...

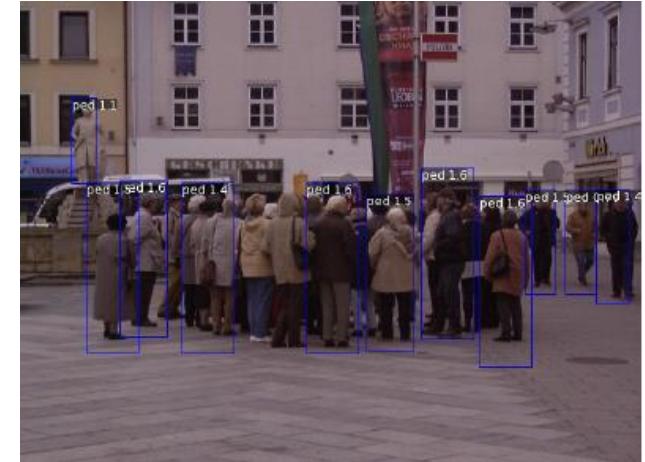


Traffic sign classification

From: Ranzato



Optical character recognition



Pedestrian detection



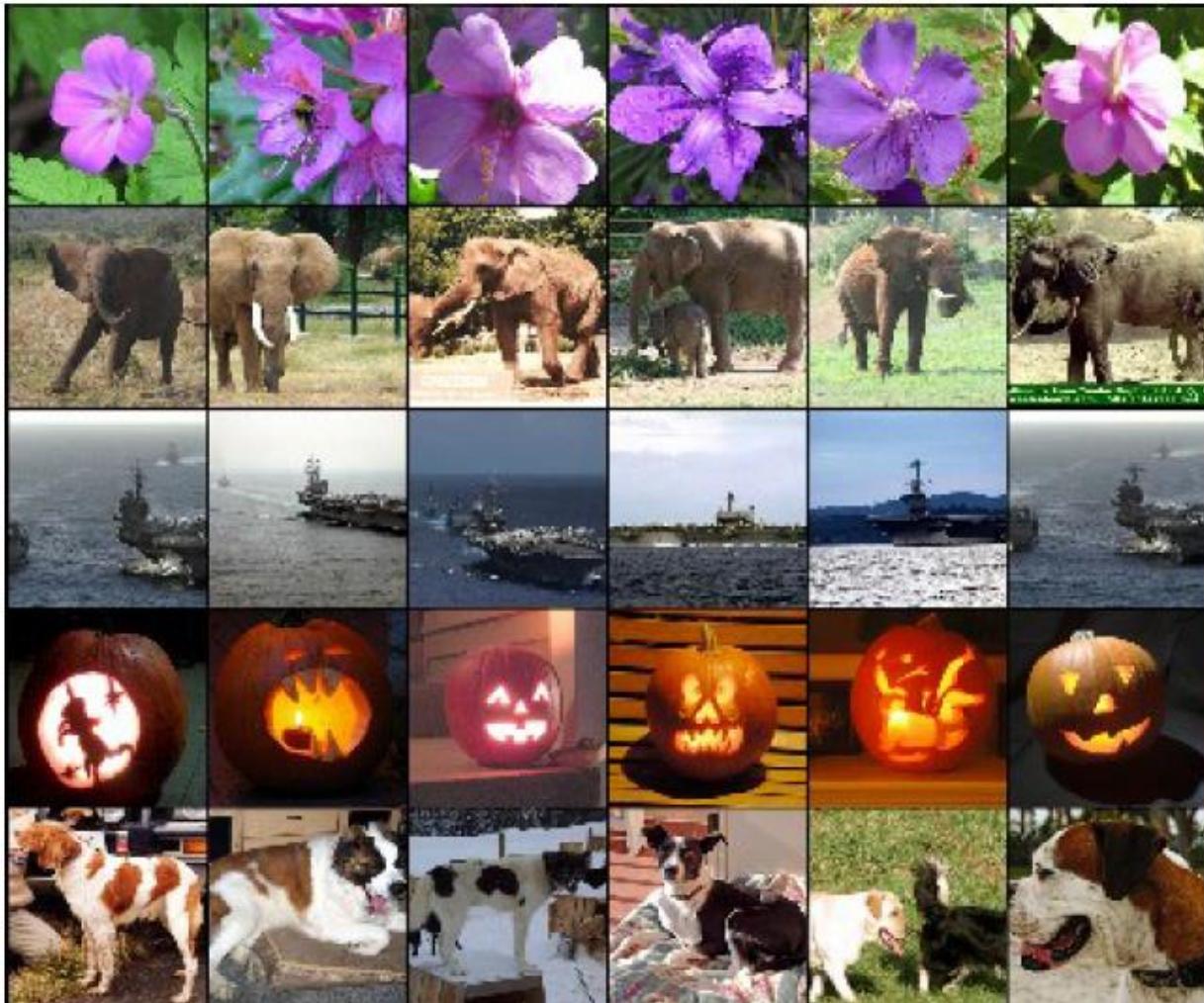
Scene parsing

Convolutional neural network: Example object retrieval

TEST
IMAGE

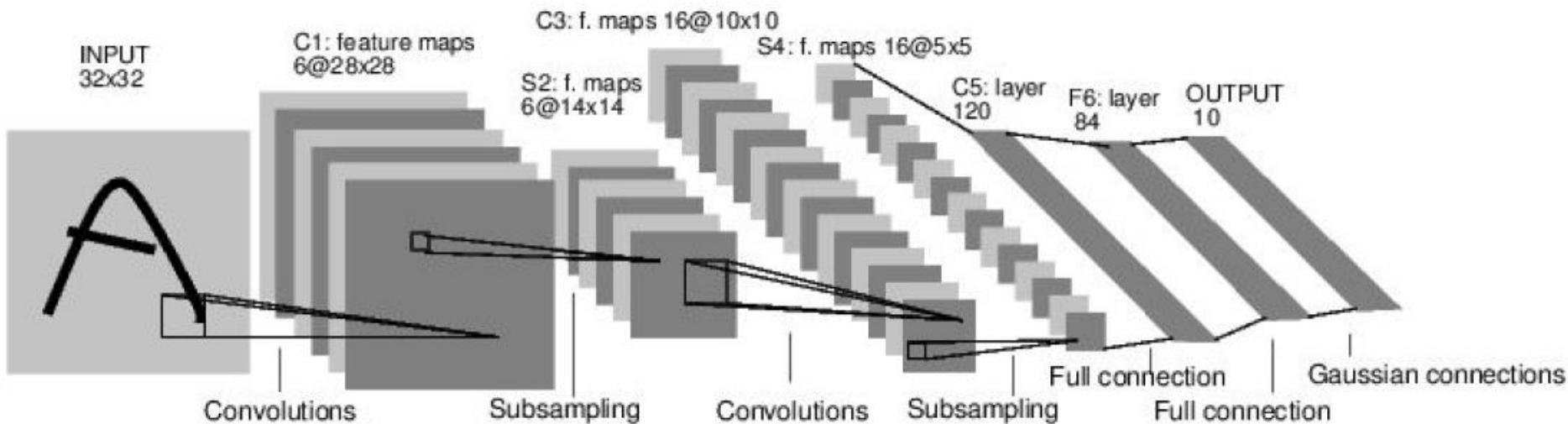


RETRIEVED IMAGES



CNN architectures: LeNet (LeCun et al., 1998)

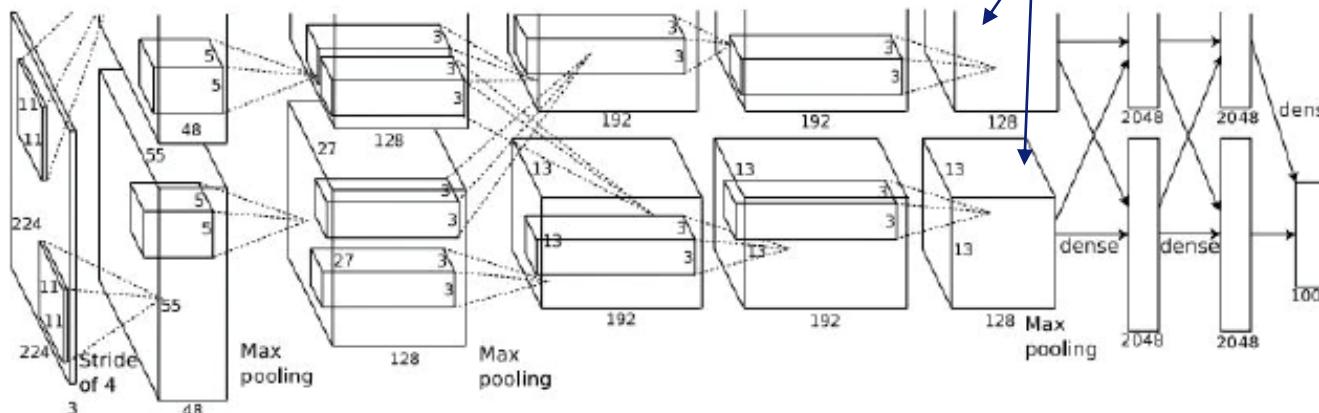
- Applied to handwritten digit recognition (MNIST)
- Input: 32 x 32 image



- Filters were 5×5 , applied at stride 1
- Subsampling (pooling) layers were 2×2 , applied at stride 2
- Architecture: [CONV – POOL – CONV – POOL – CONV – FC]

CNN architectures: AlexNet

division into 2 separate paths for memory reasons



Input:
224 x 224 x 3 image

Full (simplified) AlexNet architecture:

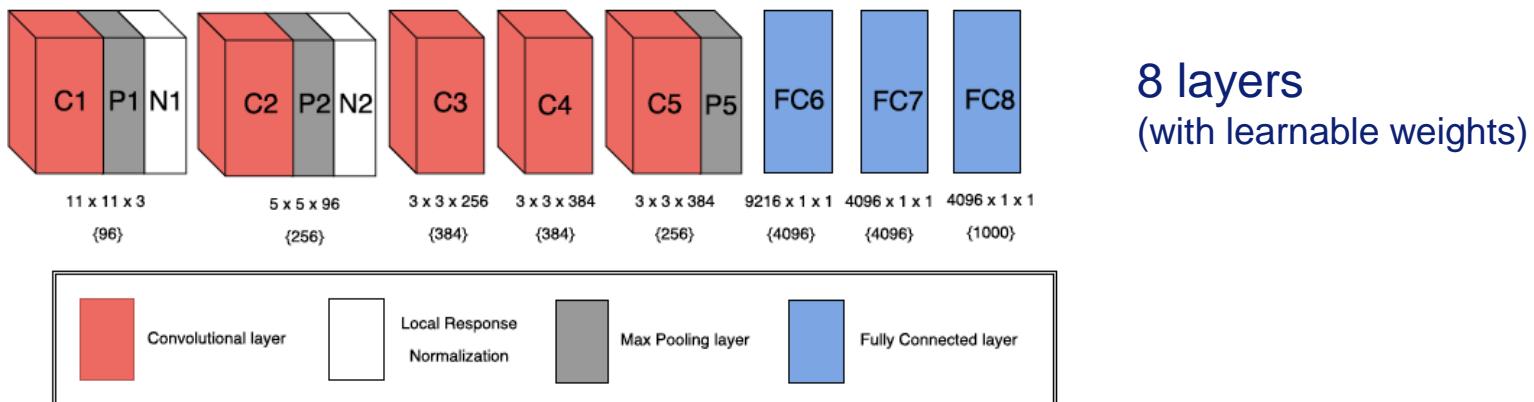
- **INPUT** [227x227x3] INPUT
- **CONV1** [55x55x96] **CONV1**: 96 11x11 filters at stride 4, pad 0
- **MAXPOOL1** [27x27x96] **MAX POOL1**: 3x3 filters at stride 2
- **NORM1** [27x27x96] **NORM1**: Normalization layer
- **CONV2** [27x27x256] **CONV2**: 256 5x5 filters at stride 1, pad 2
- **MAXPOOL2** [13x13x256] **MAX POOL2**: 3x3 filters at stride 2
- **NORM2** [13x13x256] **NORM2**: Normalization layer
- **CONV3** [13x13x384] **CONV3**: 384 3x3 filters at stride 1, pad 1
- **CONV4** [13x13x384] **CONV4**: 384 3x3 filters at stride 1, pad 1
- **CONV5** [13x13x256] **CONV5**: 256 3x3 filters at stride 1, pad 1
- **MAXPOOL3** [6x6x256] **MAX POOL3**: 3x3 filters at stride 2
- **FC6** [4096] **FC6**: 4096 neurons
- **FC7** [4096] **FC7**: 4096 neurons
- **FC8** [1000] **FC8**: 1000 neurons (class scores)

Output size:
 $(N - F + 2P) / S + 1$

Number of params:
 $F \cdot F \cdot D_1 \cdot K + K$

N: image height / width
D₁: input depth
F: filter (kernel) size
K: number of kernels
P: padding
S: stride

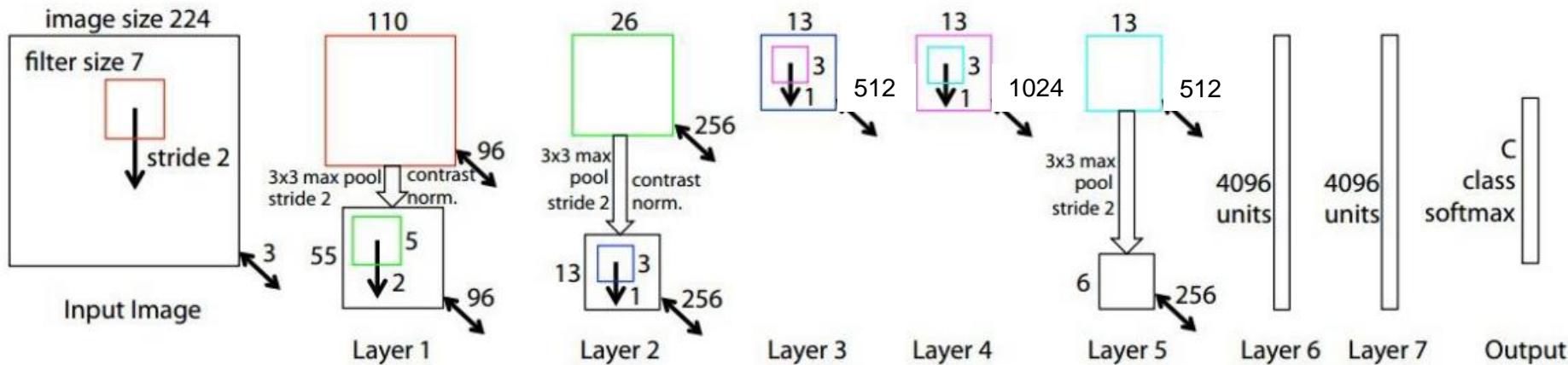
CNN architectures: AlexNet (Krizhevsky et al., 2012)



- 60 million parameters, 650.000 neurons
- Use local response normalization layers (not common anymore)
- Trained on 2 GTX-580 GPUs (3GB) (5 – 6 days)
- First use of ReLU
- Heavy data augmentation: Image flips, corner + center crops, color jitters
- Aggressive dropout: 0.5
- Batch size: 128
- SGD momentum: 0.9 (SGD: stochastic gradient descent)
- Learning rate $1e-2$, reduced by 10 manually on val. accuracy plateau
- L2 weight decay $5e-4$
- Further top-5 ImageNet valid.error red. ($18.2\% \rightarrow 15.4\%$) by ensemble of 7 CNNs

CNN architectures: ZFNet (Zeiler and Fergus, 2013)

- A variant of AlexNet



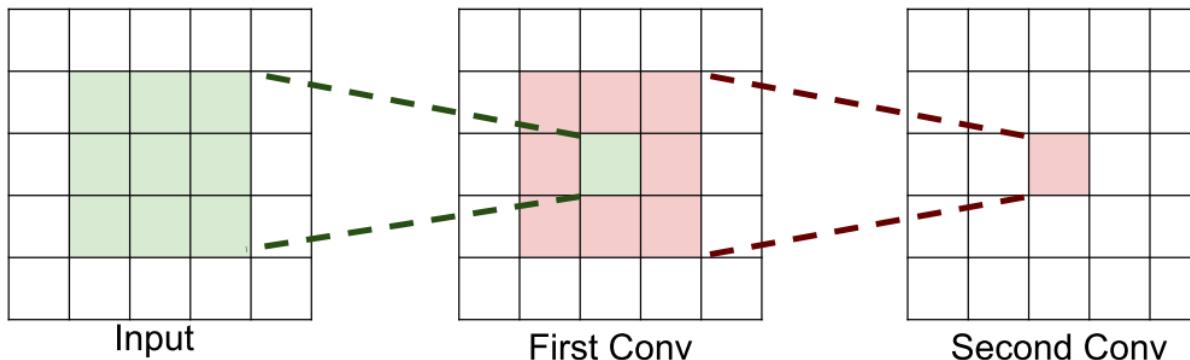
AlexNet but

- CONV1: change from (11 x 11 stride 4) to (7 x 7 stride 2)
- CONV3,4,5: instead of 384, 384, 256 filters use 512, 1024, 512
- Trained on 1.3 million images (instead of 15 million images for AlexNet); 12 days on GTX 580 GPU

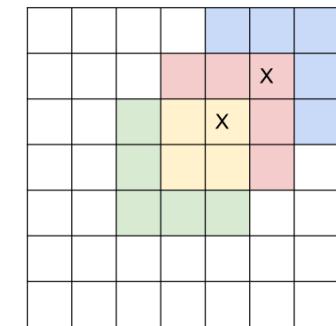
→ ImageNet top 5 error: 16.4% → 11.7%

The power of small filters (1)

- Suppose we stack **two 3×3** convolution layers with stride 1
 - Each neuron sees 3×3 region of previous activation map
 - Each neuron in *second conv layer* sees 5×5 region in the input



- Suppose we stack **three 3×3** convolution layers with stride 1
 - Each neuron in *third conv layer* sees 7×7 region in the input
- If we use a **single 7×7** convolution layer instead:
 - Each neuron in first conv layer sees 7×7 region in input
 - Similar representational power (w.r.t. receptive field)



The power of small filters (2)

- Compare number of parameters:
 - Input $H \times W \times C$, C filter (preserve depth), stride 1, padding to preserve H , W

One CONV with 7×7 filters:

$$\begin{aligned} \text{Number of weights} \\ = C \times (7 \times 7 \times C) = 49 C^2 \end{aligned}$$

Three CONV with 3×3 filters:

$$\begin{aligned} \text{Number of weights} \\ = 3 \times C \times (3 \times 3 \times C) = 27 C^2 \end{aligned}$$

Fewer parameters, more nonlinearity
(3 instead of 1)

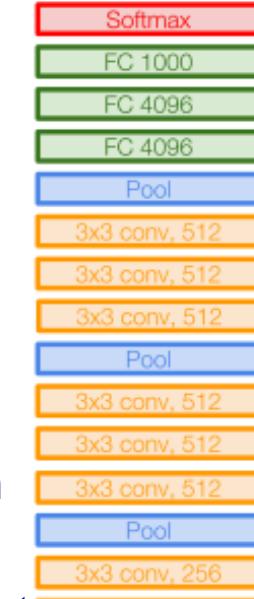
$$\begin{aligned} \text{Number of multiply-adds} \\ = (H \times W \times C) \times (7 \times 7 \times C) = 49 HWC^2 \end{aligned}$$

$$\begin{aligned} \text{Number of multiply-adds} \\ = 3 \times (H \times W \times C) \times (3 \times 3 \times C) = 27 HWC^2 \end{aligned}$$

Less computations, more nonlinearity

→ Use more smaller filters instead of larger ones!

CNN architectures: VGG (Simonyan and Zisserman, 2014)

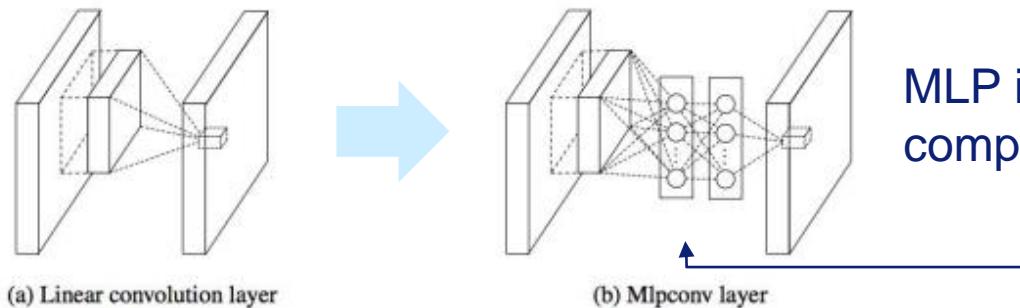
- Principle: Use smaller filters and more layers (deeper networks)
 - Only 3×3 convolution with stride 1, pad 1
 - 5 pooling layers (2×2 max pooling, stride 2)
 - 16 / 19 layers (VGG16 / VGG19)
 - More details:
 - 3 fully connected layers
 - No local response normalization
 - Similar training procedure than AlexNet
 - ILSVRC 2014 2nd in classification, 1st in localization
 - VGG19 slightly better than VGG16, more memory
 - Use ensembles for best results
 - FC7 features generalize well to other tasks

(1 more convolutional layer,
i.e. altogether 3 layers!)

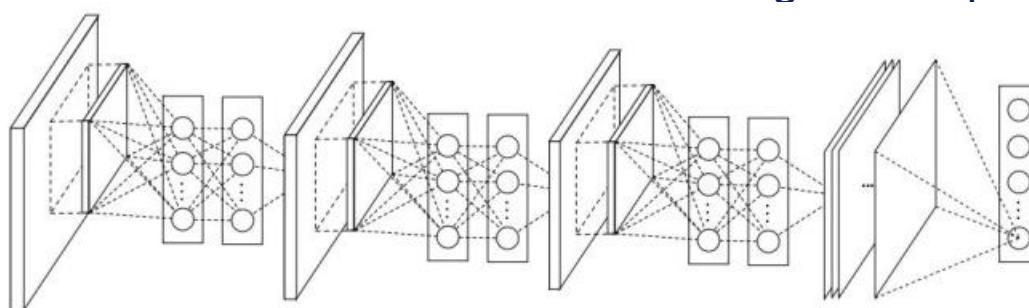


CNN architectures: Network in Network („NIN“, Lin et al., 2014)

- Replace linear conv. layer by mlpconv layer including a MLP: micro network
 - Motivation: conv layer is a generalized *linear* model for underlying data patch, whereas MLP is a more potent *nonlinear* function approximator which can represent more general concepts (due to nonlinearity in each layer)
 - MLP consists of multiple fully connected layers with nonlinear activ. fct. (ReLU)



- MLP is shared among all local receptive fields and slided over the input similarly as the CNN to compute the output feature map
- Overall structure of NIN is a stacking of multiple mlpconv layers:



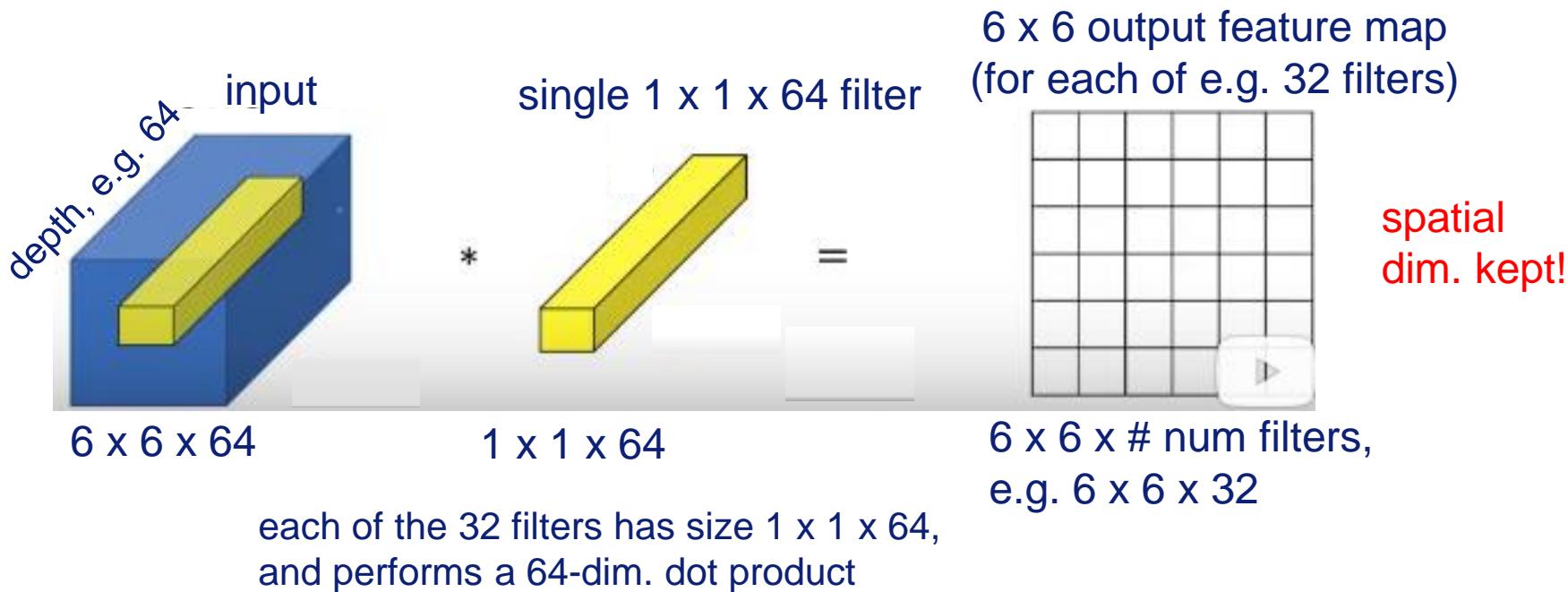
„Network in Network“

CNN architectures: Global average pooling (Lin et al., 2014)

- Classification traditionally performed in the last layers (fully connected fc)
 - But: FC layers prone to overfitting, hampering generalization (\rightarrow dropout)
- Idea: Replace fully connected layers by **global average pooling (GAP)**:
 - Generate **one feature map per classification category** in the last mlpconv layer
 - For each feature map, GAP computes **spatial average** over input height / width
 - Then, feed vector of spatial averages into softmax activation function
- Advantages:
 - Feature maps can be interpreted as categories confidence maps
 - No parameters in global average pooling \rightarrow no overfitting
 - Spatial information summed out \rightarrow robust to spatial input translations
 - But: Not clear whether GAP is useful in standard CNNs; seems to need NIN

CNN architectures: 1 x 1 convolutions (Lin et al., 2014)

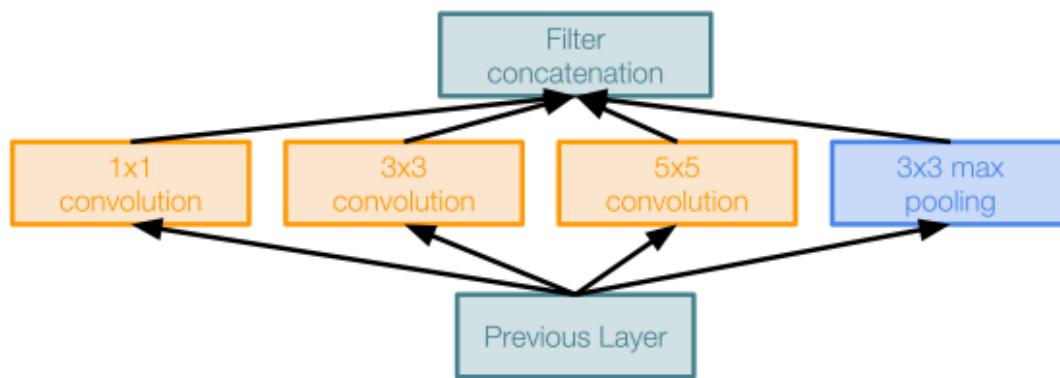
- Goal: Reduce depth (# feature channels), preserve spatial dimensions
 - Depth projected to lower dimension (smaller depth by combining feature maps)



- Each filter m computes $z_{ij}^{(m)} = \sum_{k=1}^{64} w_k^{(m)} x_{ijk} + b^{(m)}$ **fully connected along depth dimension**
- Network in Network: Precursor to GoogLeNet („philosophical inspiration“) and ResNet bottleneck layers (both excessively use 1 x 1 convolutions)

CNN architectures: Inception (Szegedy et al., 2014) – Motivation (1)

- Idea: Instead of choosing which convolutional filter size to use:
Use a variety of convolutions and concatenate all inputs



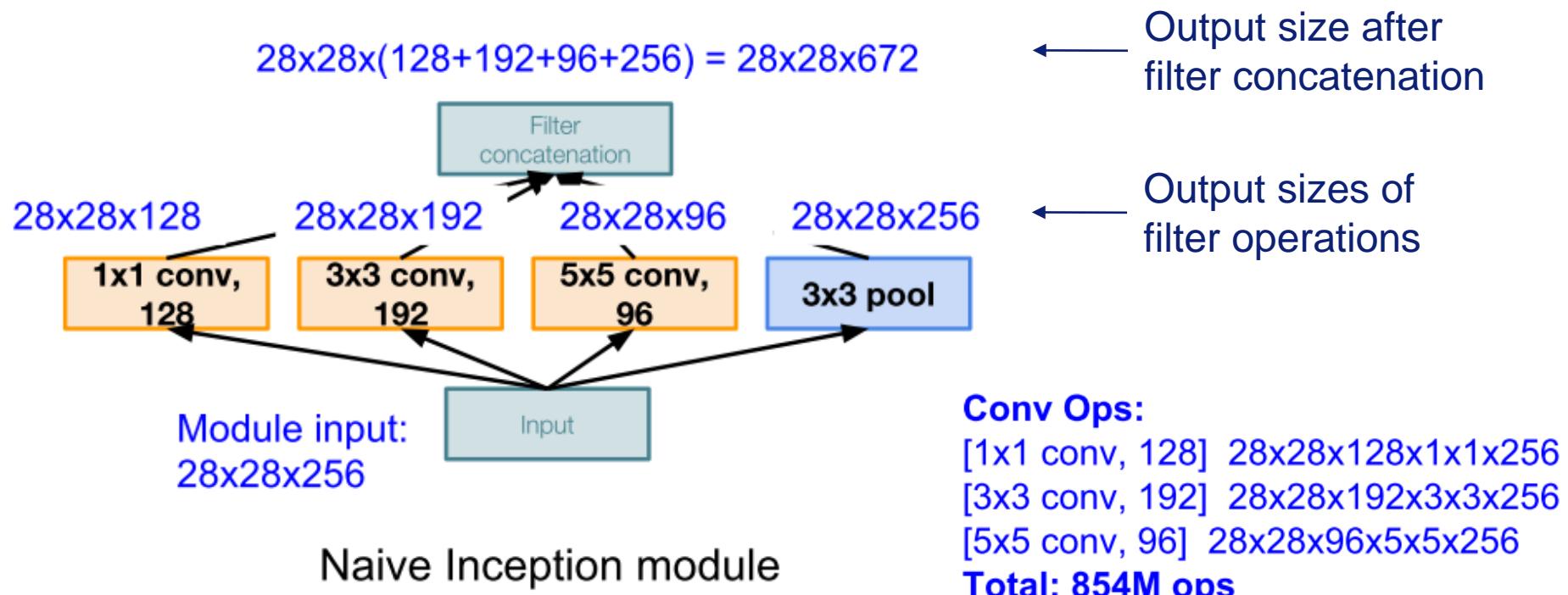
- Apply parallel filter operations on the input from previous layer: different filter sizes (1×1 , 3×3 , 5×5) and pooling (3×3)
- Concatenate all filter outputs together depth-wise

Naive Inception module

- What is the problem?
- Assume input of size $28 \times 28 \times 256$ and 128, 192, 96, 256 filter maps, resp.

CNN architectures: Inception (Szegedy et al., 2014) – Motivation (2)

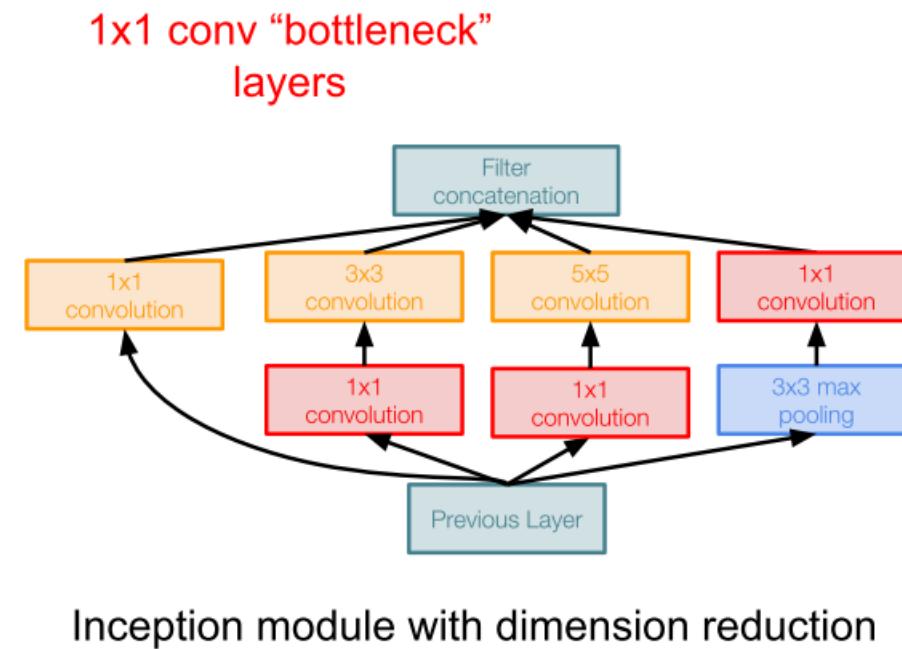
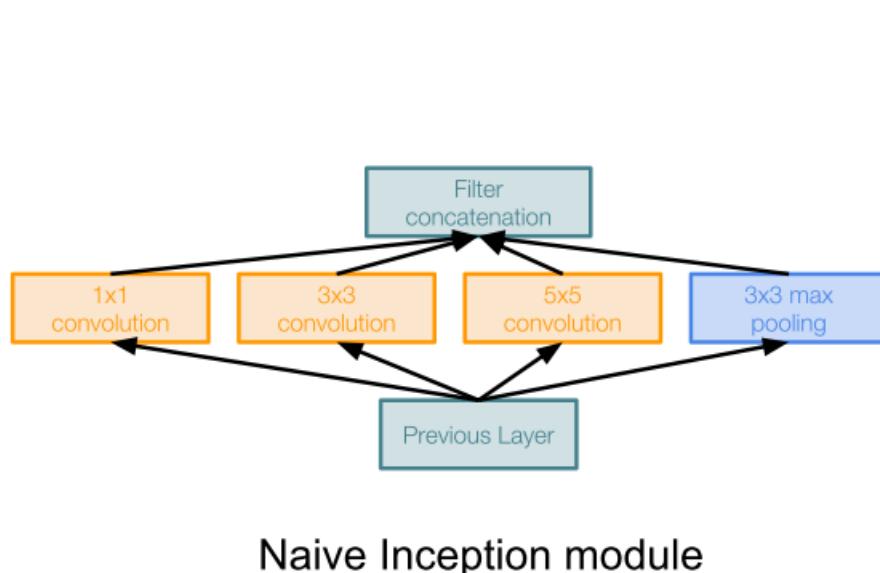
- Idea: Instead of choosing which convolutional filter size to use:
Use a variety of convolutions and concatenate all inputs



- What is the problem?
- Assume input of size $28 \times 28 \times 256$ and 128, 192, 96, 256 filter maps, resp.

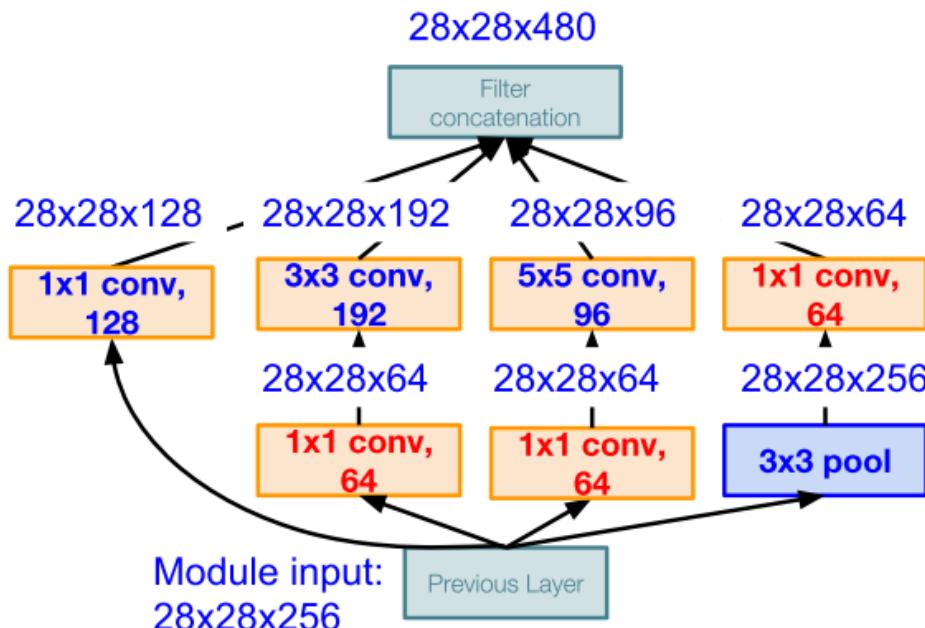
CNN architectures: Inception (Szegedy et al., 2014) – Motivation (3)

- Naive Inception module:
 - Very large number of computations
 - Pooling layer preserves feature depth
 - total depth after concatenation can only grow at every layer!
- Solution: „bottleneck“ layers using 1×1 convolutions to reduce feature depth



CNN architectures: Inception (Szegedy et al., 2014) – Motivation (4)

- Efficient inception module:
 - Calculation using same parallel layers as naive example, and adding „1x1 conv, 64 filter“ bottlenecks:



Inception module with dimension reduction

Conv Ops:

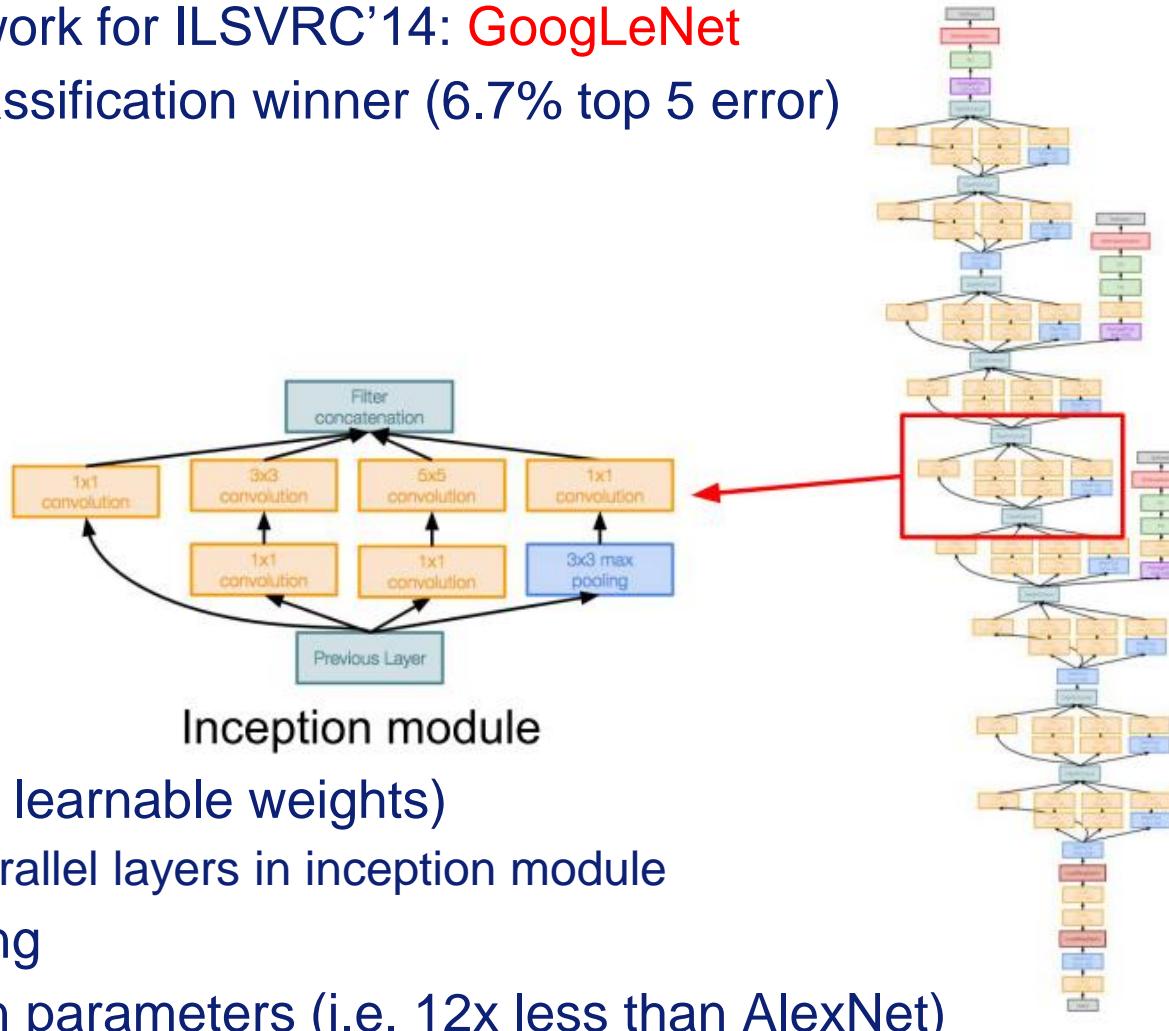
[1x1 conv, 64] 28x28x64x1x1x256
 [1x1 conv, 64] 28x28x64x1x1x256
 [1x1 conv, 128] 28x28x128x1x1x256
 [3x3 conv, 192] 28x28x192x3x3x64
 [5x5 conv, 96] 28x28x96x5x5x64
 [1x1 conv, 64] 28x28x64x1x1x256

Total: 358M ops

Compared to 854M ops for naive version
 Bottleneck can also reduce depth after pooling layer

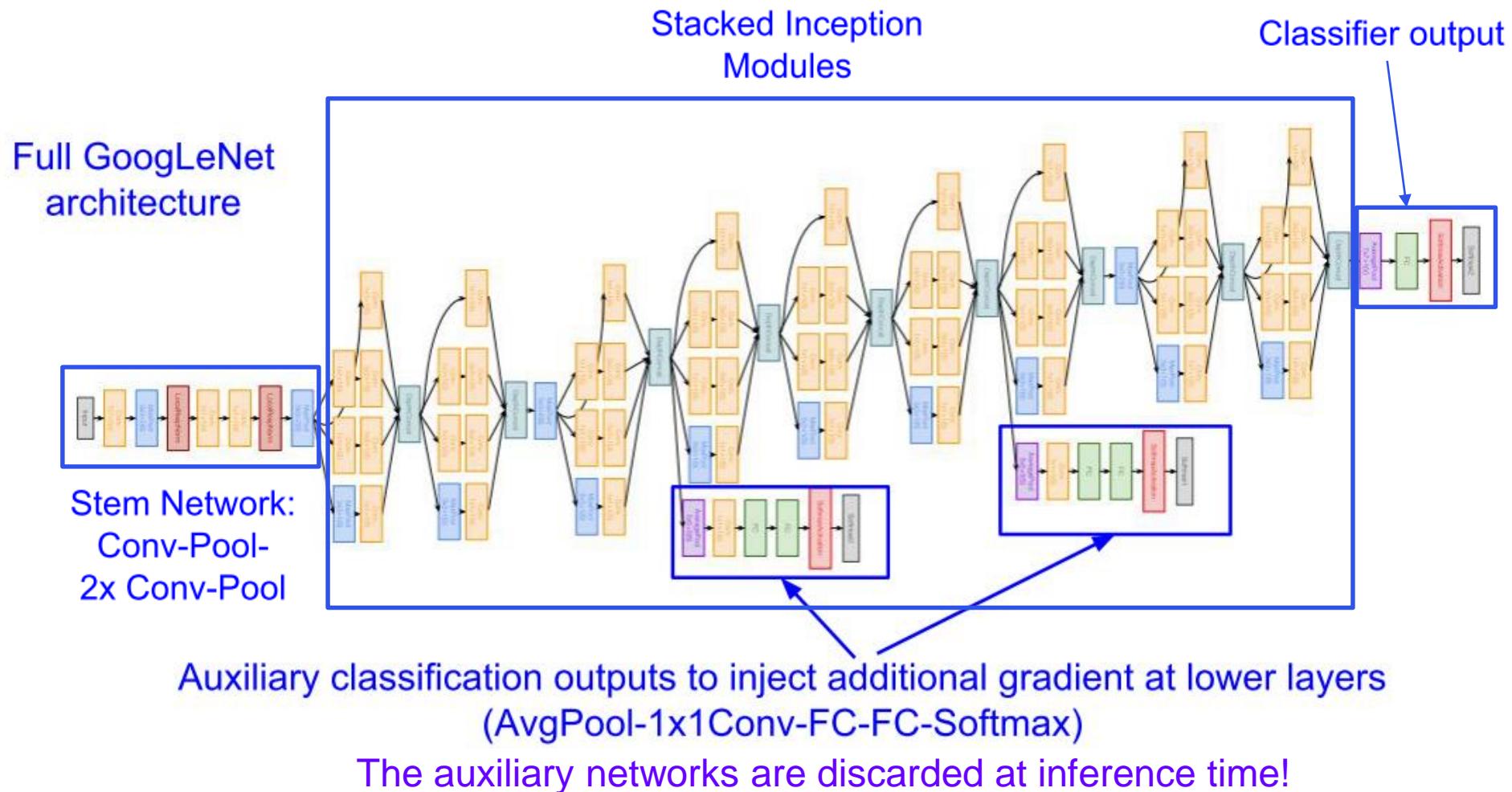
CNN architectures: Inception / GoogLeNet (Szegedy et al., 2014)

- Stack inception modules with dimension reduction on top of each other
- Particular network for ILSVRC'14: **GoogLeNet**
- ILSVRC'14 classification winner (6.7% top 5 error)



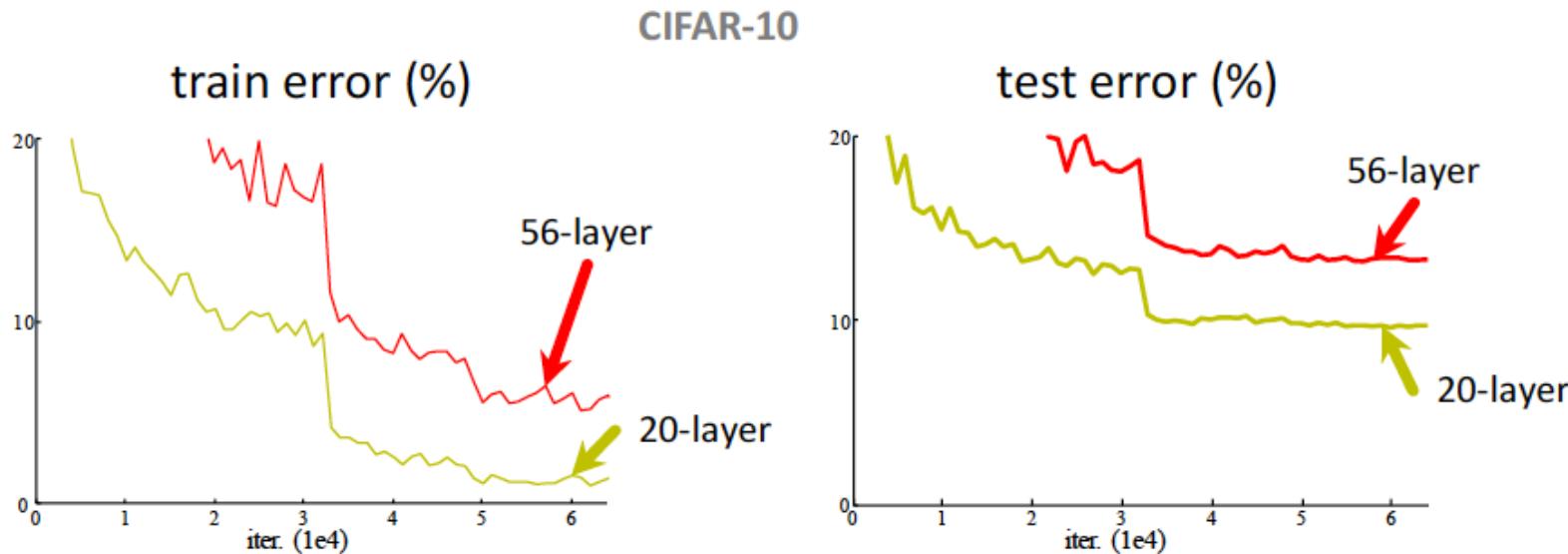
- 22 layers (with learnable weights)
 - Including parallel layers in inception module
- Average pooling
- „Only“ 5 million parameters (i.e. 12x less than AlexNet)

CNN architectures: Inception / GoogLeNet (Szegedy et al., 2014)



CNN architectures: ResNet (He et al., 2015) – Motivation (1)

- Stacking more layers is easy, but is it also easy to **train** those networks?
- Experiments using „plain nets“, i.e. just stacking 3×3 conv layers:

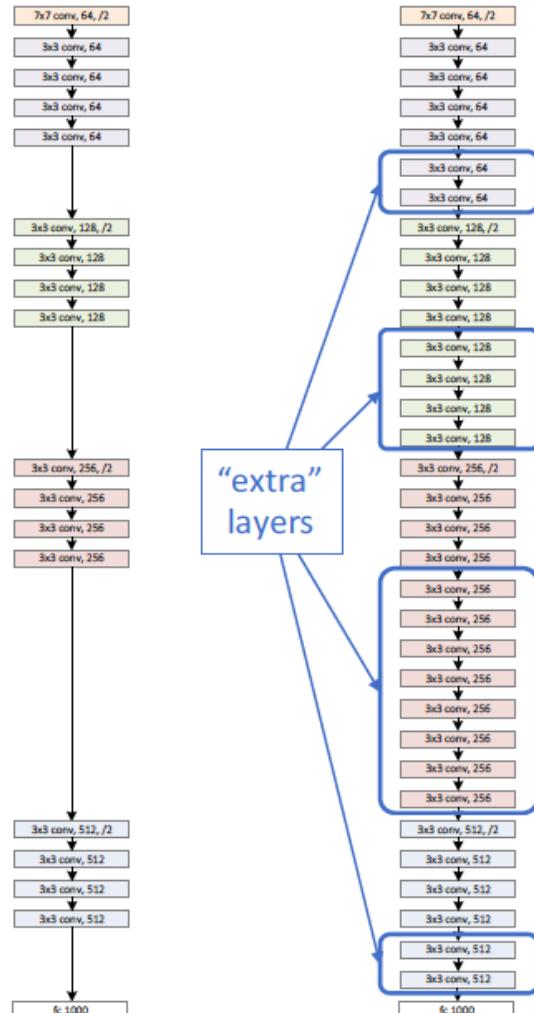


- Observation: 56-layer net: **higher training and test error** than 20-layer net
 - Deeper model performs worse, but it's not caused by overfitting!
- Hypothesis: **Optimization problem** (deeper models harder to optimize)
 - Deeper model should be able to perform at least as well as shallower model!

CNN architectures: ResNet (He et al., 2015) – Motivation (2)

- Solution by construction: Copy learned layers from shallower model, set additional layers to identity mapping

a shallower
model
(18 layers)



a deeper
counterpart
(34 layers)

Let's construct a deep model by copying layers from a pre-trained shallower model

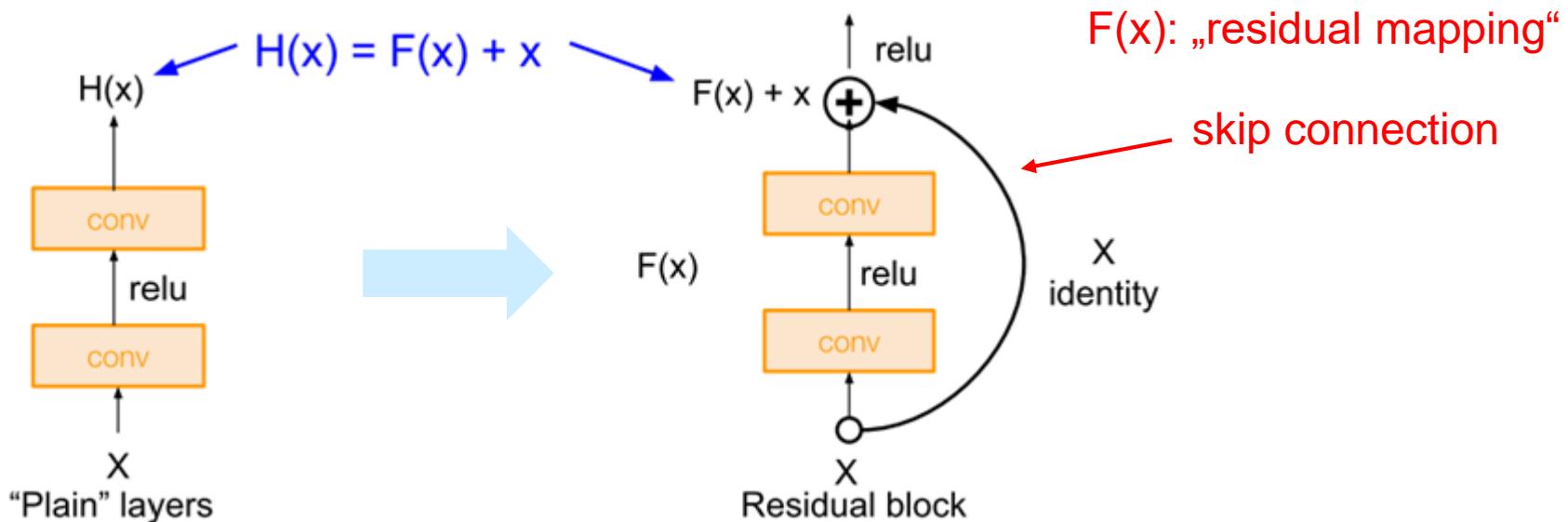
Extra layers set as identity

Should ensure at least same training error

Optimization difficulty: Solvers cannot find a solution, while going deep.

CNN architectures: ResNet (He et al., 2015) – Motivation (3)

- Use network layers to **fit residual mapping** instead of directly trying to fit a desired underlying mapping („make it easy to represent identity mapping“)



$H(x)$ is any desired mapping;
hope the 2 conv layers fit $H(x)$

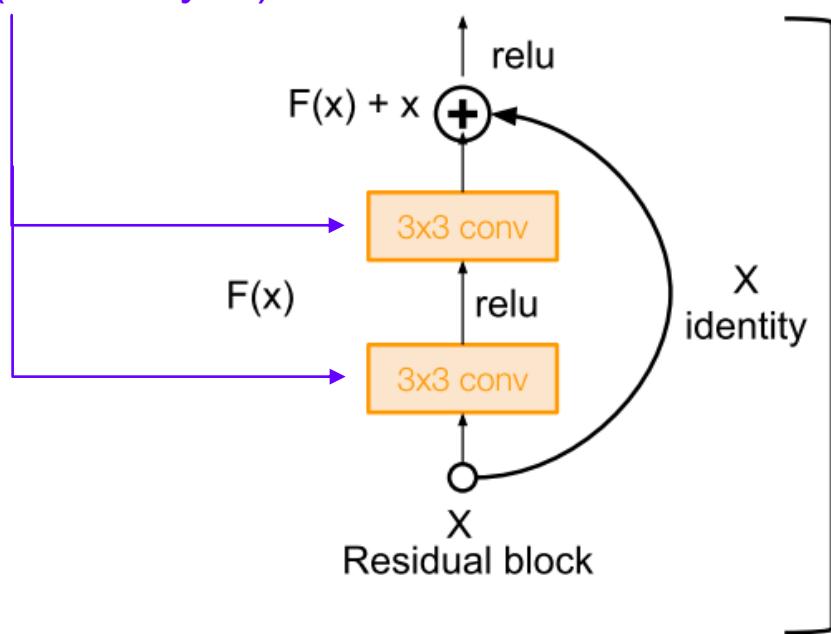
$H(x)$ is any desired mapping;
hope the 2 conv layers fit $F(x) = H(x) - x$

- If identity was optimal, all weights can just be set to 0
- If optimal mapping is close to identity, $F(x)$ is small and can be better learned

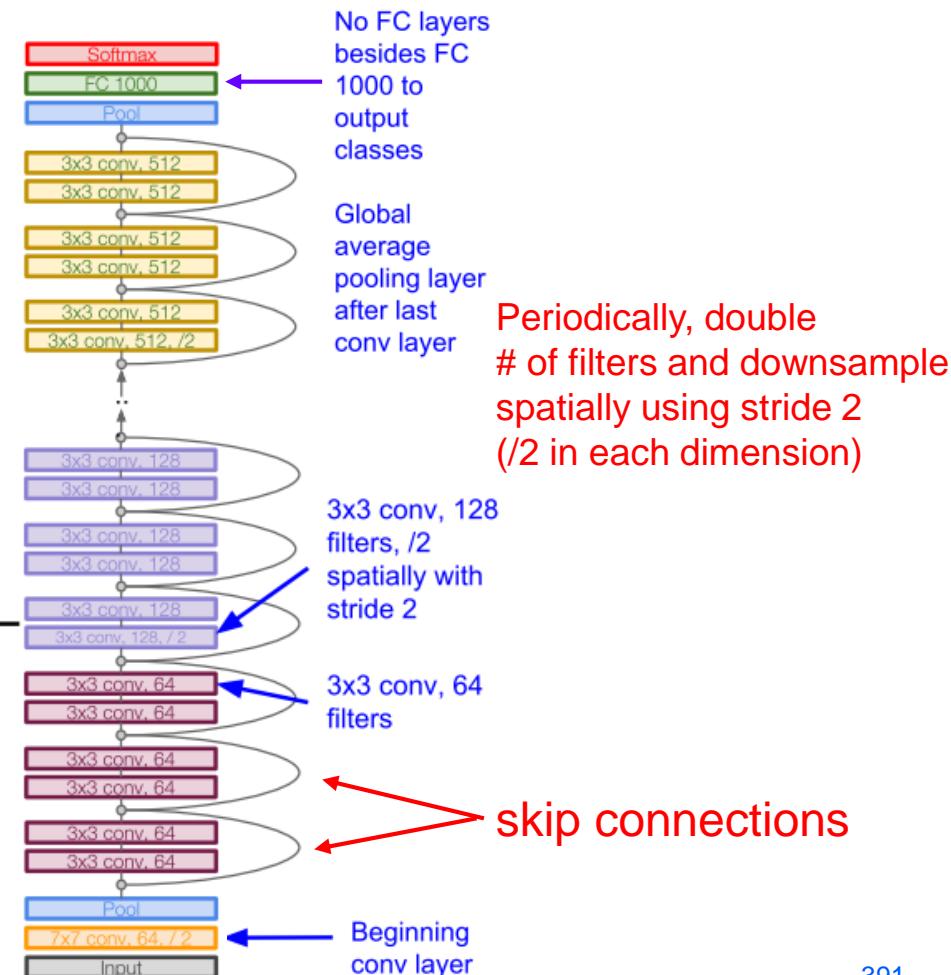
CNN architectures: ResNet (He et al., 2015) – Principle

- Stack residual blocks (with skip connections)
- No fully connected layers (only FC 1000 to output), no dropout
- Heavy use of batch normalization

Every residual block has only 3 x 3 convolutions („VGG style“)



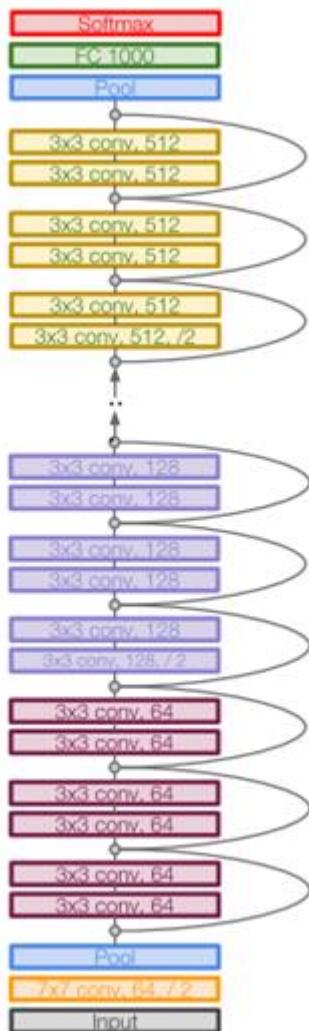
2-3 weeks training on 8 GPU machine!



CNN architectures: ResNet (He et al., 2015)

- State-of-the-art CNN (ILSVRC 2015 winner: 3.57% top-5-error)

Total depth of
34, 50, 101 or 152 →
layers for ImageNet

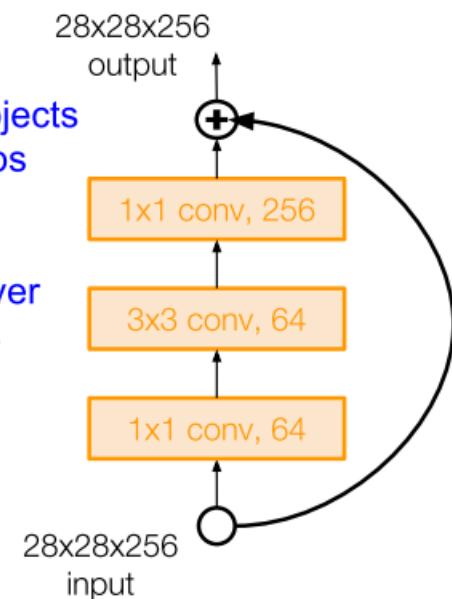


For deeper networks
(ResNet-50+), use
„bottleneck“ layer to
improve efficiency
(similar to GoogLeNet):

1x1 conv, 256 filters projects
back to 256 feature maps
(28x28x256)

3x3 conv operates over
only 64 feature maps

1x1 conv, 64 filters
to project to
28x28x64



CNN architectures: ResNet (He et al., 2015)

- From https://github.com/PaddlePaddle/book/tree/develop/03.image_classification

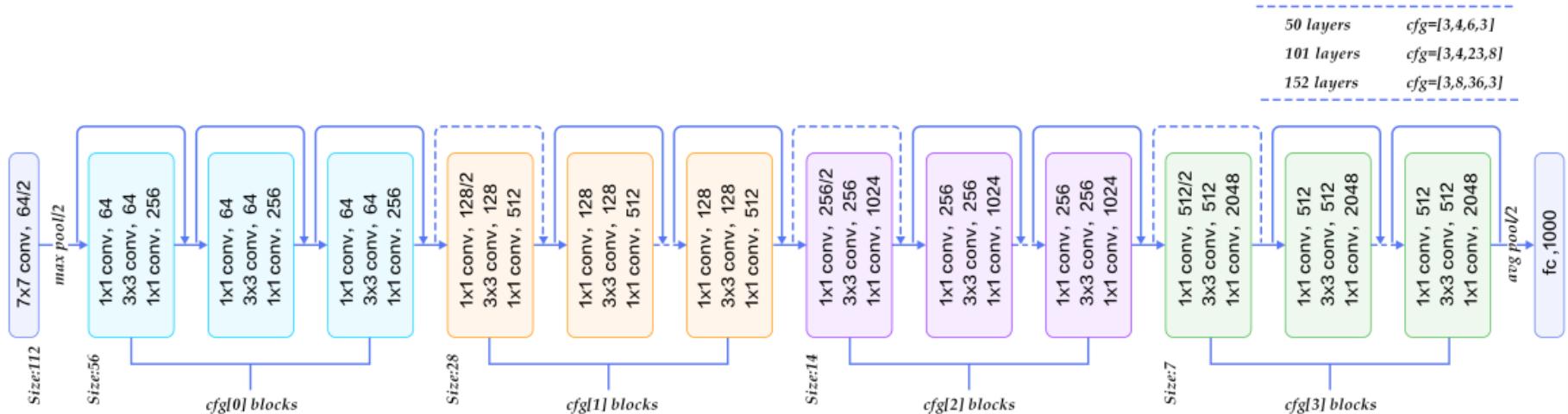
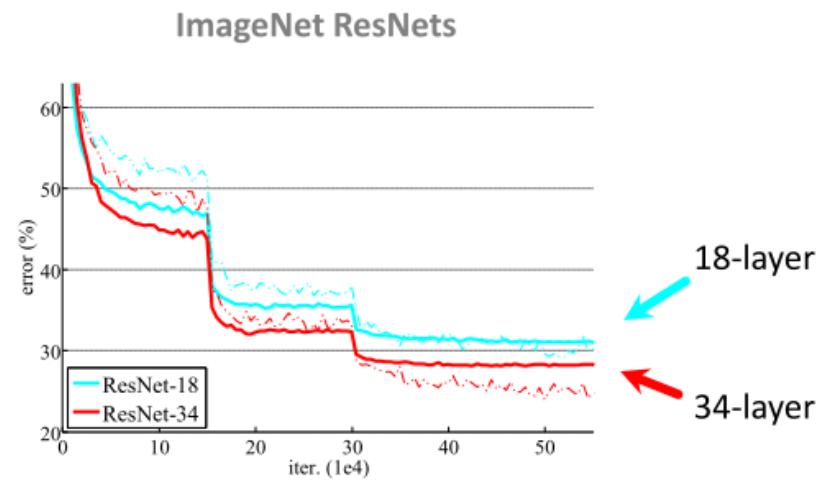
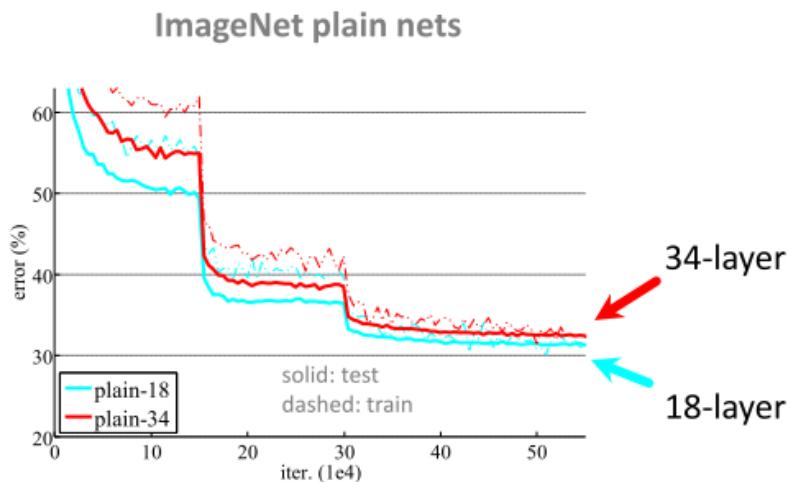
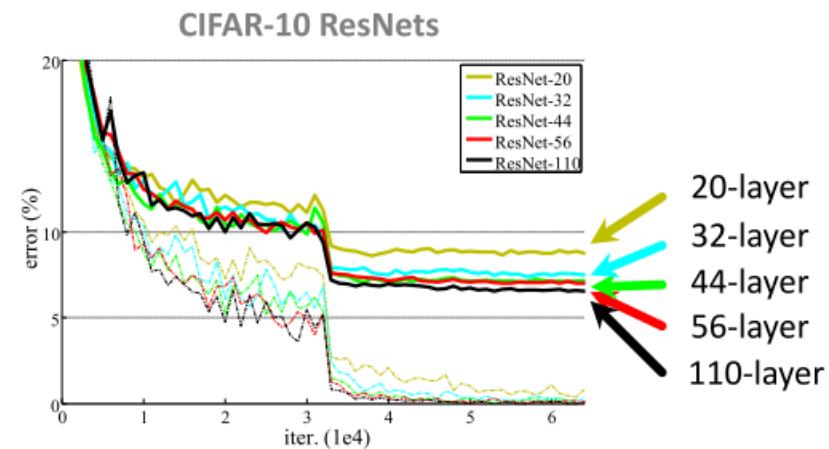
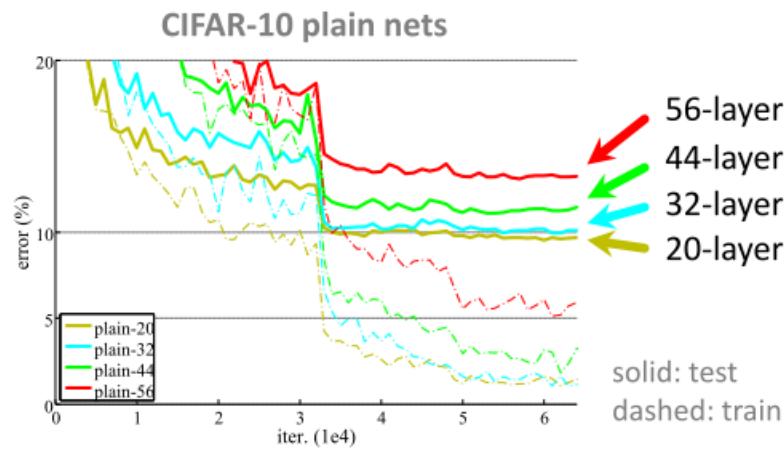


Fig. 4: Architecture of ResNet. In this project ResNet-101 has been used. Reprinted from [2].

The ResNet architecture is shown in Fig 4. The initial input data is rescaled by a 7x7 convolutional layer and a stride 2 max-pooling to a size of 112px. The following path can generally be divided into four major parts, labeled as cfg[0-3] blocks in the image. From one block type to the next the amount of kernels is doubled (256, 512, 1024, 2048) and the input size is halved (56, 28, 14, 7). The additional circumventing paths can be seen in the graphic as well. The version used in this project consists of 101 layers – 3 cfg[0] layers, 4 cfg[1] layers, 23 cfg[2] layers and 8 cfg[3] layers.

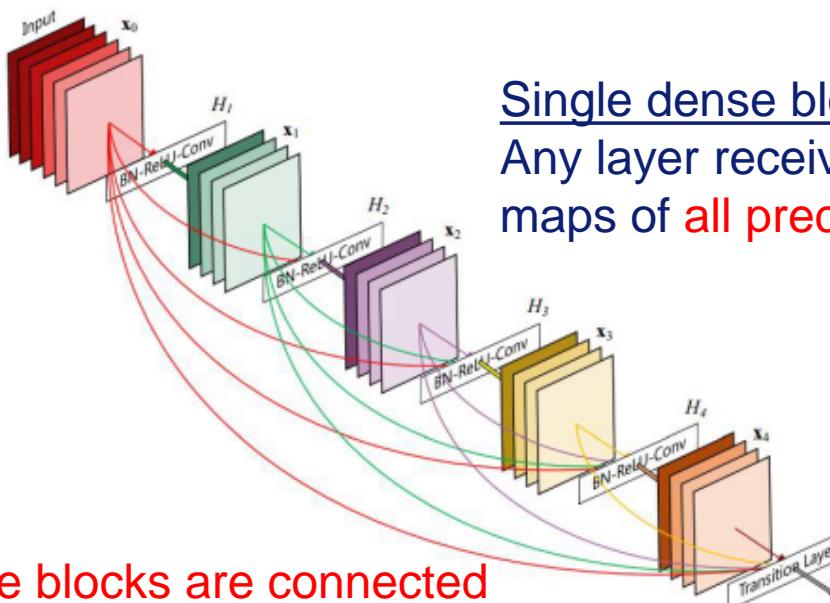
CNN architectures: ResNet (He et al., 2015)



- Deep ResNets can be trained without difficulties
- Deeper ResNets have **lower training error**, and also lower test error

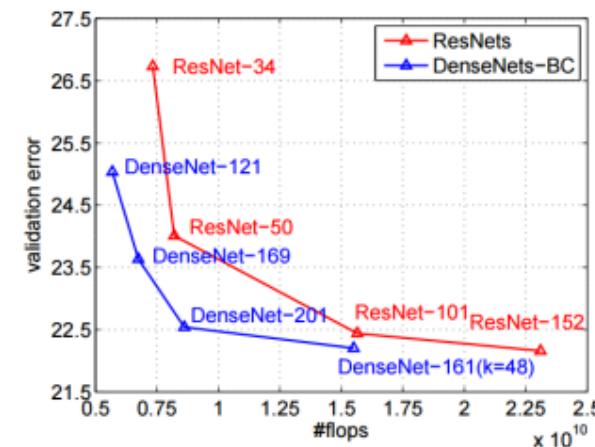
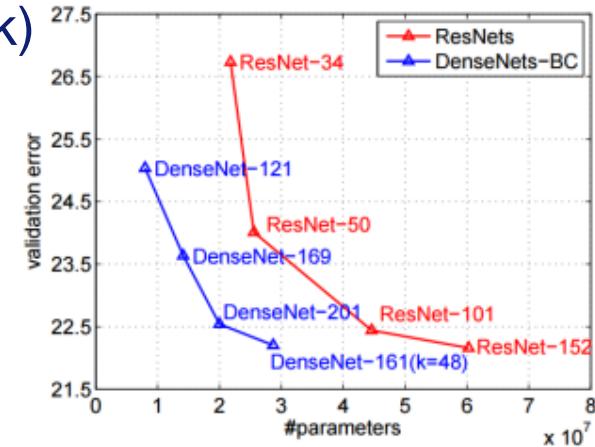
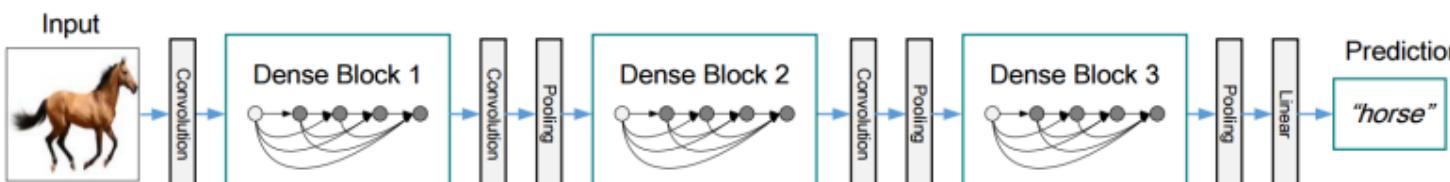
CNN architectures: DenseNet (Huang et al., 2017)

- Skip connections (like ResNet) support direct gradient flow
→ Connect layer to **all subsequent layers** (within block)
 - Concatenate feature map with „earlier“ feature maps
 - Only few feature maps per layer → few parameters



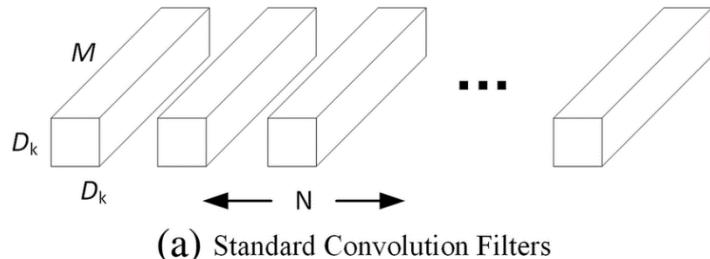
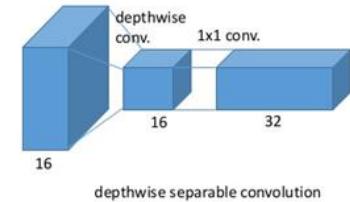
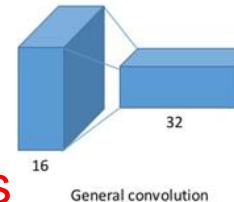
Single dense block:
Any layer receives feature maps of **all preceding layers**

Dense blocks are connected via (e.g.) 1x1 conv. & pooling

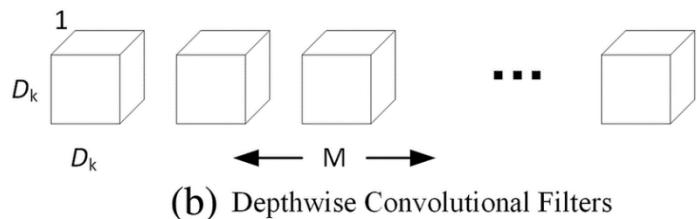


MobileNet (Howard et al., 2017)

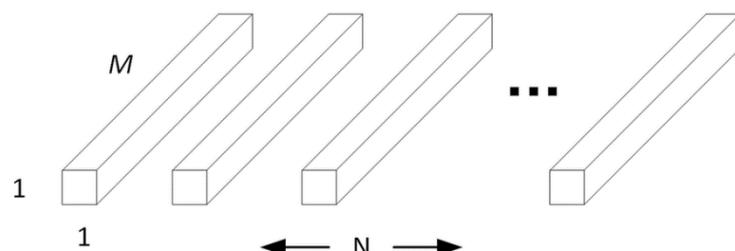
- For embedded or mobile computer vision tasks
- Less multiplications and parameters by depthwise separable convolution:



Standard convolution („Conv“):
Each filter has same depth as the input (M)



First step in depthwise separable convolution:
Depthwise convolution („Conv dw“):
Each filter has depth 1; there are M such filters



Second step in depthwise separable convolution:
Pointwise convolution („Conv“):
1 x 1 convolutions (with depth M), using N filters

Same output size in both cases
(output size x output size x N)

MobileNet (Howard et al., 2017)

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
5× Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

First layer: standard convolution

Depthwise convolution ...

... followed by pointwise convolution

Altogether 13 (dw + pw) conv layers;
followed by AvgPool, FC and Softmax

Accuracy on ImageNet:

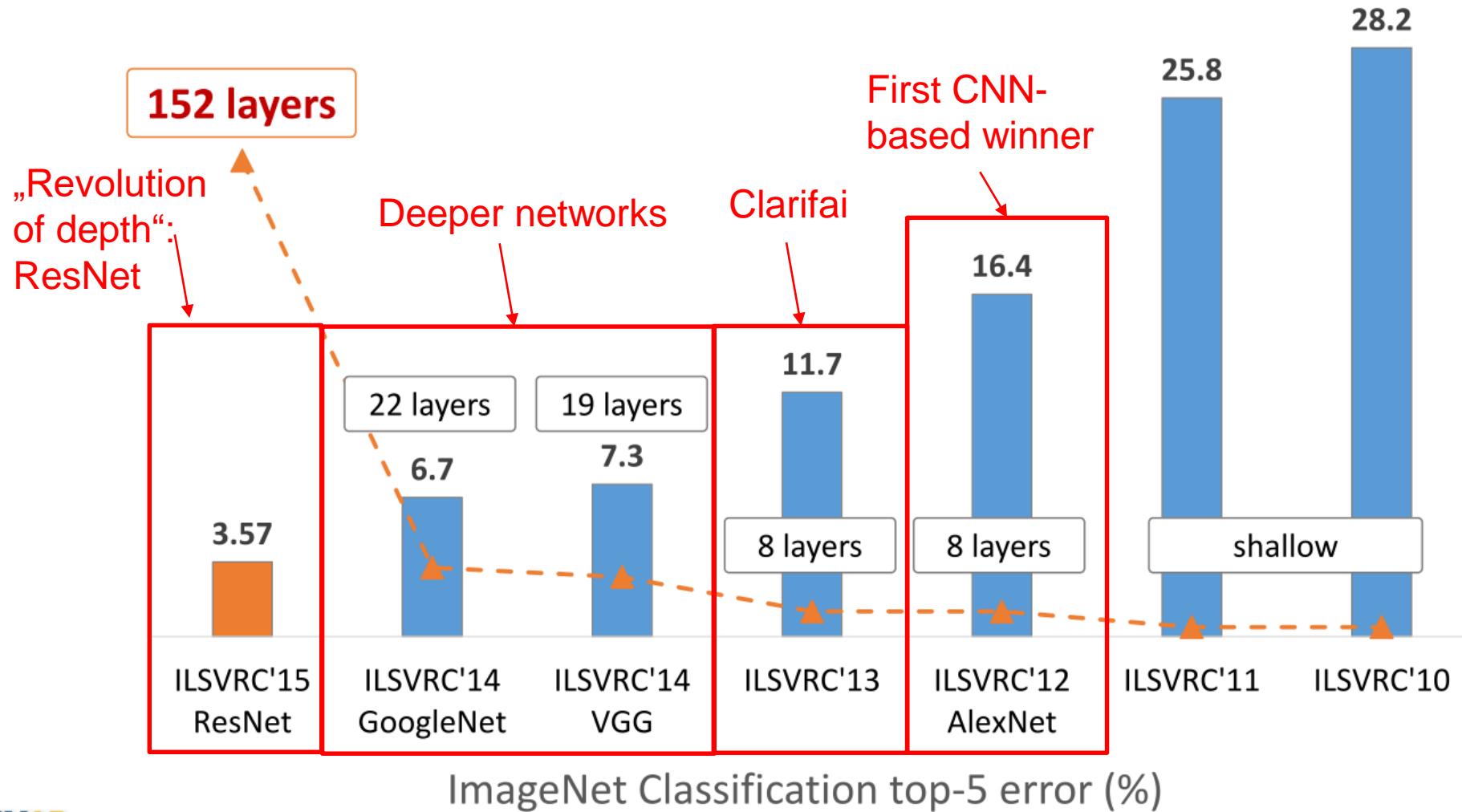
- MobileNet (4.2×10^6 Params): 70.6%
- VGG16 (138×10^6 Params): 71.5%

s1, s2: stride

- **Further complexity reduction** by width multiplier α (to downscale network at each layer) and resolution multiplier ρ (downscale image size)
 - New training required when changing value of any of the two parameters

Convolutional neural networks: Evaluation

- ImageNet large scale visual recognition challenge (ILSVRC)
- 1000 categories, 1.5 million labeled training samples (2012)

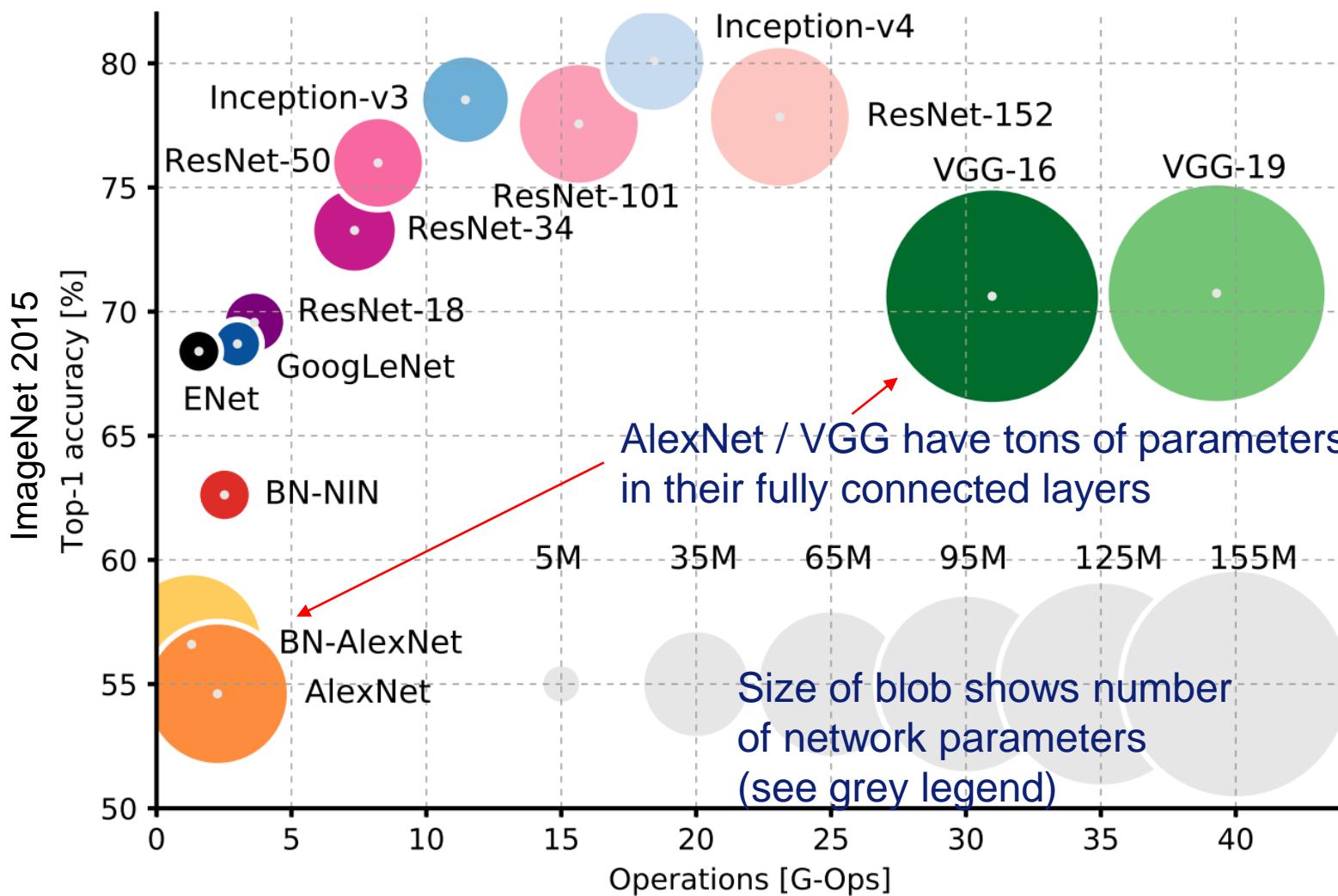


Computational / Model complexity

BN: Batch normalization

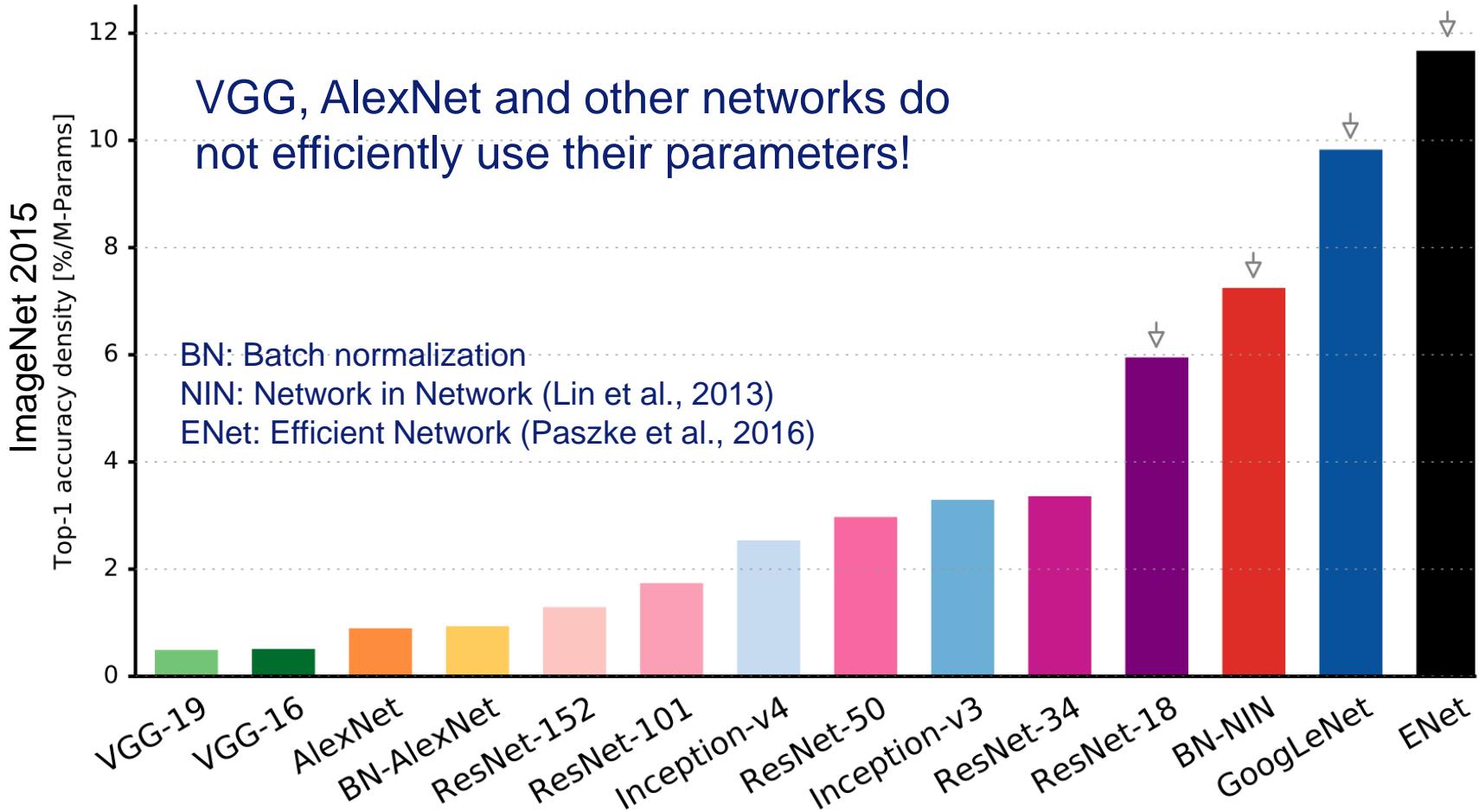
NIN: Network in Network (Lin et al., 2013)

ENet: Efficient Network (Paszke et al., 2016)



Canziani et al., „An Analysis of Deep Neural Network Models For Practical Applications“, 2017

Accuracy per parameter versus network type

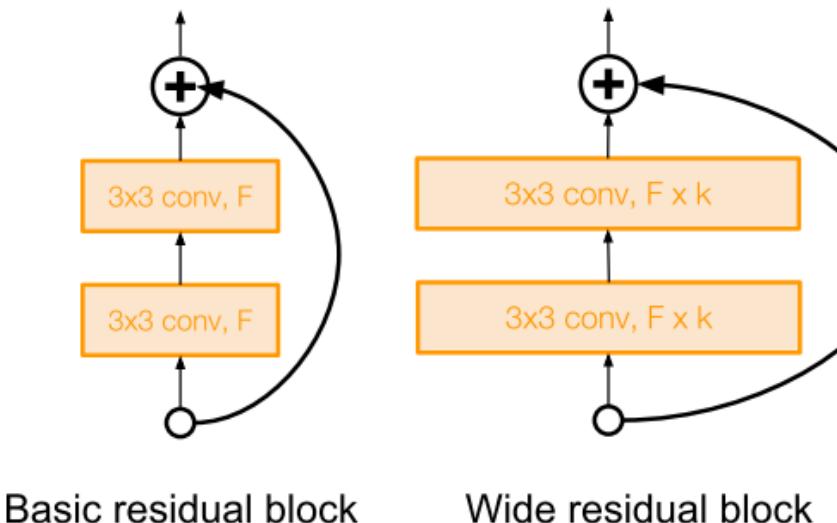


Canziani et al., „An Analysis of Deep Neural Network Models For Practical Applications“, 2017

400

CNN architectures: Summary

- VGG, GoogLeNet, ResNet all in **wide use**, available in model zoos
- ResNet current best default
- Trend towards **extremely deep networks**
- **Research w.r.t design of layer / skip connections, improving gradient flow**
- Current trend: Examining necessity of **depth vs width** / residual connections
 - Width: $F \times k$ filters instead of F filters in each layer of a residual block



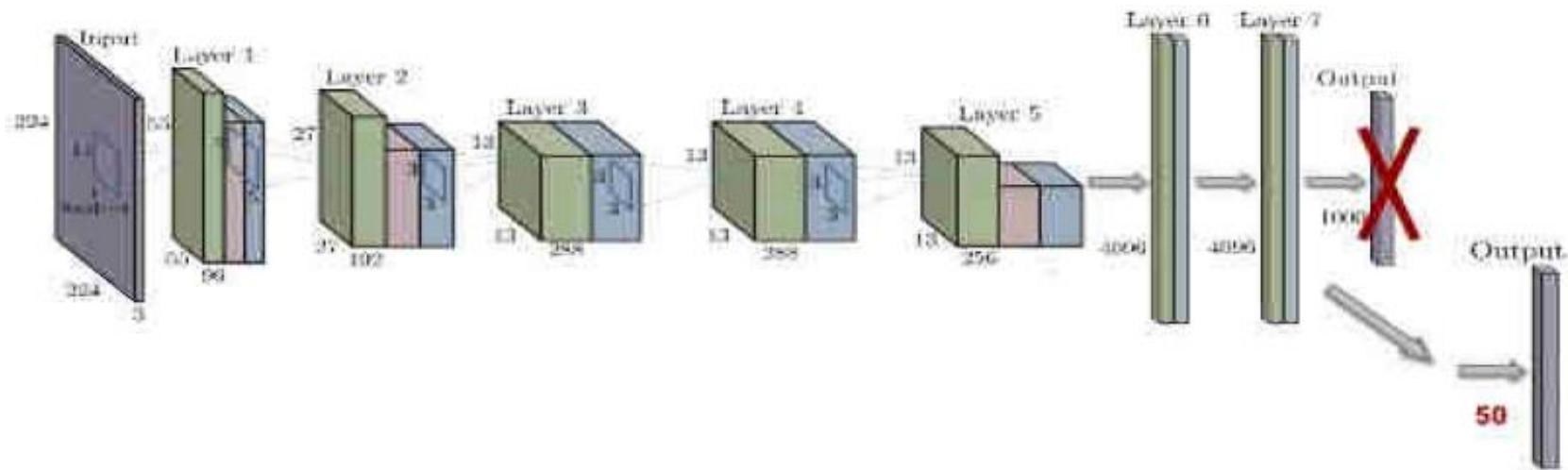
Zagoruyko et al., 2016:

- 50-layer wide ResNet outperforms 152-layer original ResNet
- Increasing width instead of depth computationally more efficient (parallelizable)

Transfer learning

Starting from CNN e.g. trained on ImageNet:

- Remove last (classification) layers
- Obtain transformation of any input image into semi-abstract representation
- **Can be used for learning something else („transfer learning“):**
 - Either by just using learned representations as features
 - Or by creating new CNN output layers and perform learning of those additional layers plus fine-tuning of the other layers



- Using CNN pre-trained on large dataset, it's possible to adapt it to another task, using only a small training set! (Example: Semantic segmentation)

Common computer vision tasks

- So far: Image classification



[This image is CC0 public domain](#)

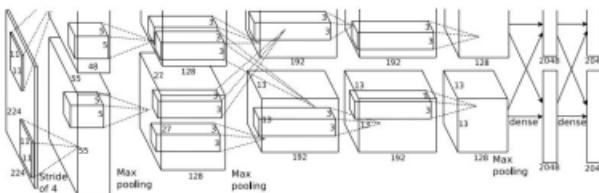


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

Image size often *fixed*
due to fully connected layers

Class Scores

Cat: 0.9
Dog: 0.05
Car: 0.01
...

Fully-Connected:
4096 to 1000

Vector:
4096

- Other computer vision tasks:

Semantic Segmentation



GRASS, CAT,
TREE, SKY

Classification + Localization



CAT

Object Detection



DOG, DOG, CAT

Instance Segmentation



DOG, DOG, CAT

No objects, just pixels

Single Object

Multiple Object

[This image is CC0 public domain](#)

Common computer vision tasks

- **Object detection:** Localize different objects by placing bounding box around each instance of a pre-defined category
- **Semantic segmentation:** Assign object category label to each pixel in image
- **Instance segmentation:** Assign unique identifier to each segmented object in image

Combined with Natural Language Processing:

- **Image captioning:** Describe an image in words
- **Visual Question-Answering:** Answering textual questions from images



Object Detection



Semantic Segmentation



Instance Segmentation

Tags: Person, Dining Table

A group of people sitting at a table

Q: What were the people doing?
A: Eating dinner

Image Classification

Image Captioning

Visual Question-Answering

Object classification and localization

- Tasks:

Classification: C classes

Input: Image

Output: Class label

Evaluation metric: Accuracy



→ CAT

From: Fei-Fei Li

Localization:

Input: Image

Output: Box in the image (x, y, w, h)

Evaluation metric: Intersection over Union



→ (x, y, w, h)

Bounding box:
 x, y : position
 w, h : width, height

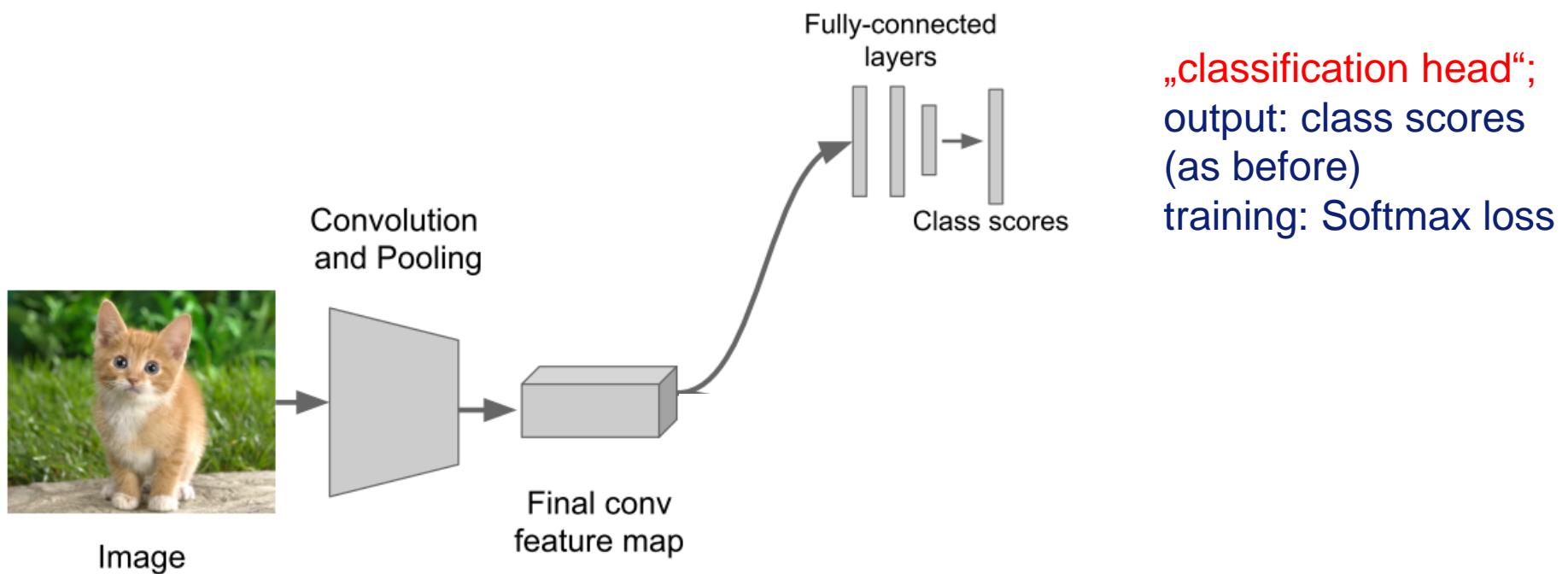
Classification + Localization: Do both

Object localization:

- As **regression**: Estimating bounding box(es), only pre-defined # objects)
- Using **region proposals** (generated by CNN or by other method)
- As **segmentation** (pixel-wise) → see „semantic segmentation“

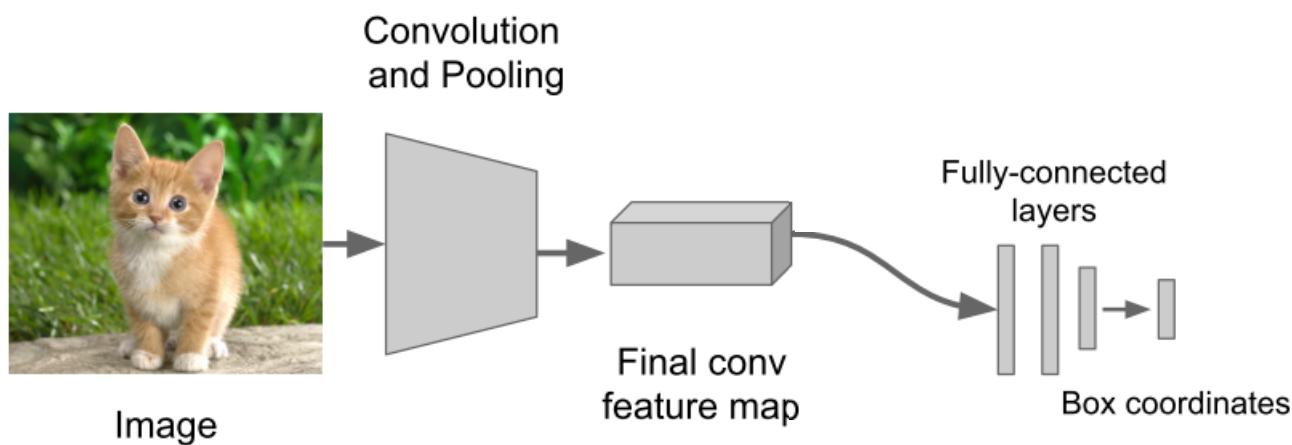
Object localization as regression

- Only one object (simpler than detection which may contain many objects):
 - Replace „classification head“ by „regression head“; fine-tuning with MSE loss



Object localization as regression

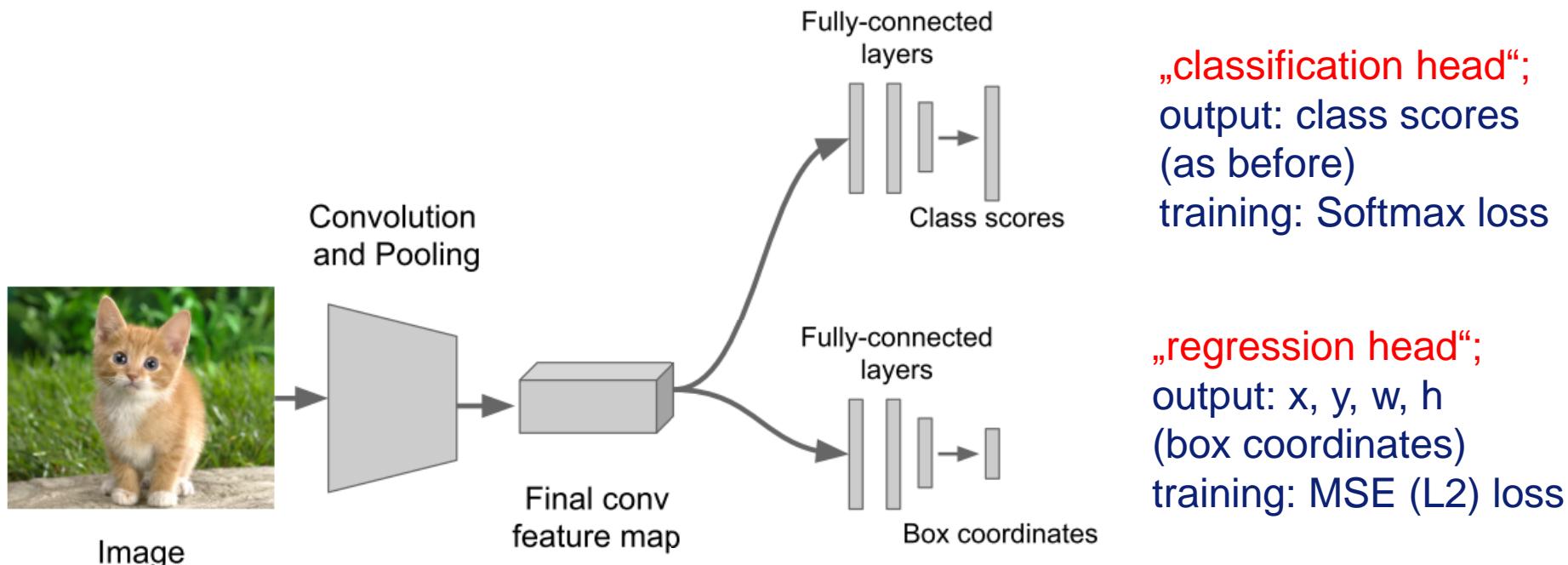
- Only one object (simpler than detection which may contain many objects):
 - Replace „classification head“ by „regression head“; fine-tuning with MSE loss
- Multiple (i.e. exactly K) objects: Output $K \times 4$ numbers (one box per object)



„regression head“;
output: x, y, w, h
(box coordinates)
training: MSE (L2) loss

Object localization as regression

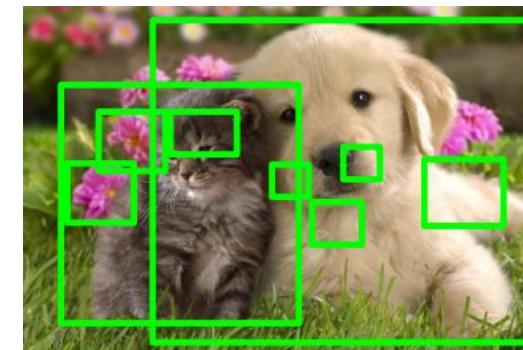
- Only one object (simpler than detection which may contain many objects):
 - Replace „classification head“ by „regression head“; fine-tuning with MSE loss
- Multiple (i.e. exactly K) objects: Output $K \times 4$ numbers (one box per object)
- Joint classification and localization: Use both heads at test time



- Object detection: via **region proposals** or **semantic segmentation**
 - Variable number of objects

Object detection by sliding window and region proposals

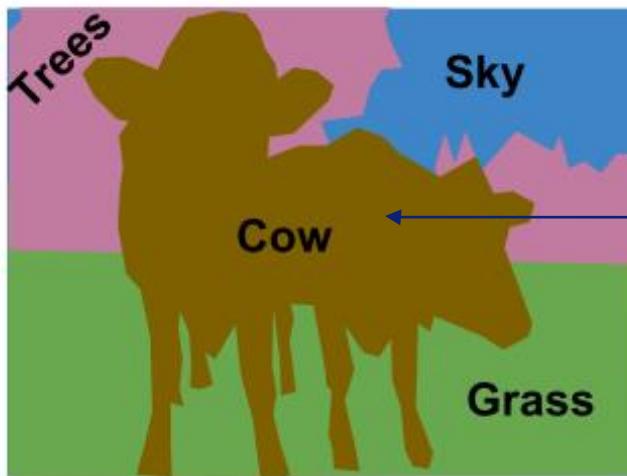
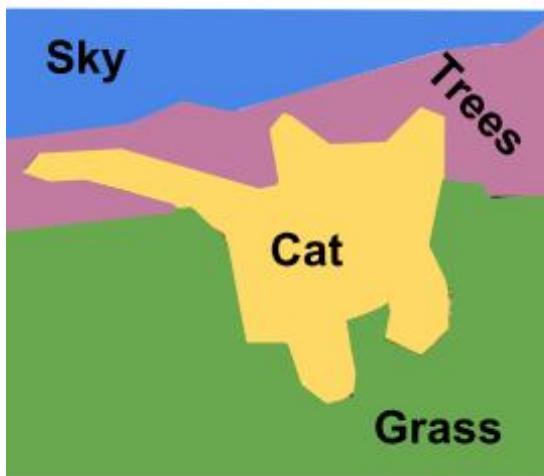
- **Sliding window:** Apply CNN to many different crops of the image
 - Classification of each crop as object or background
 - Potentially regression of bounding box
- Problem: Huge number of locations and scores → too many computations
- Alternative: **Region proposals** (regions likely to contain objects):



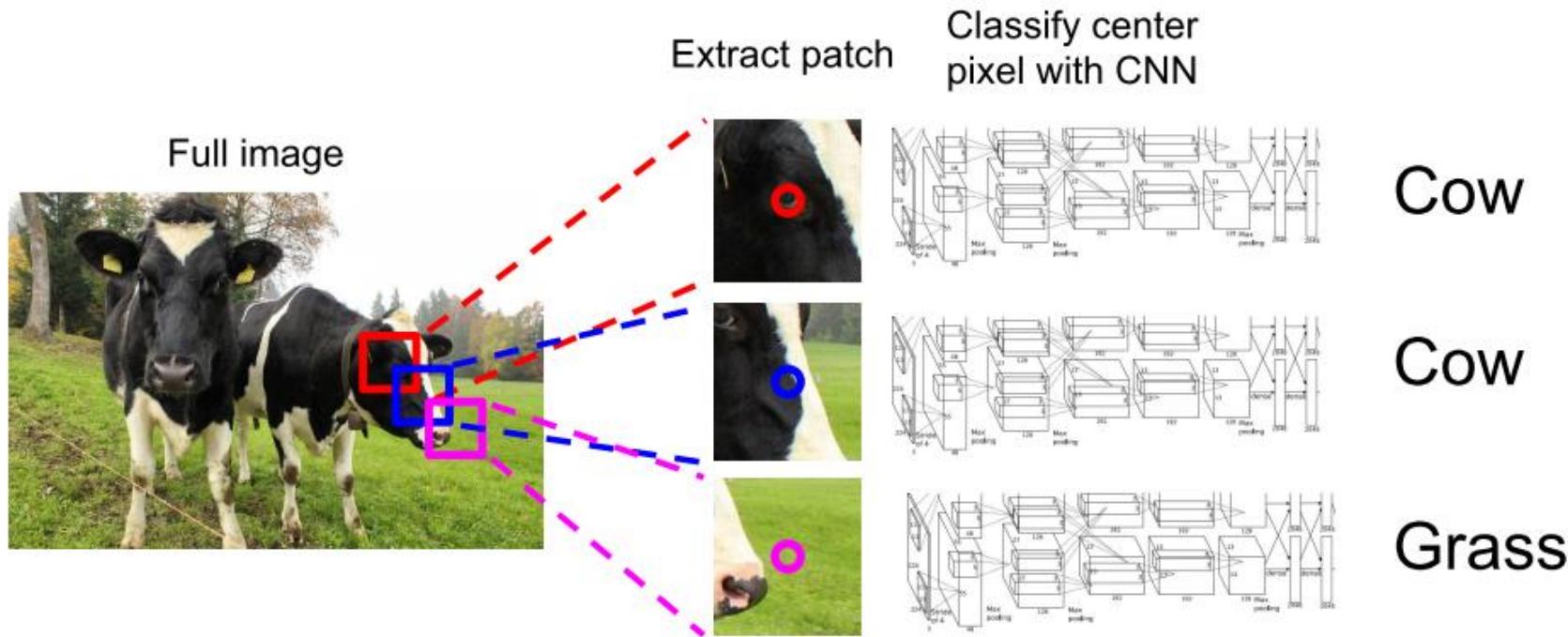
- Region proposals by CNN or by other method (e.g. selective search)
- Examples:
 - R-CNN, Fast / Faster R-CNN („two-stage methods“, region proposal network)
 - YOLO, SSD („one-stage method“)

Semantic segmentation

- Task: Label each pixel in the image with a category label
 - Don't differentiate between instances (e.g. 2 cows), only care about pixels



Semantic segmentation idea: Sliding window



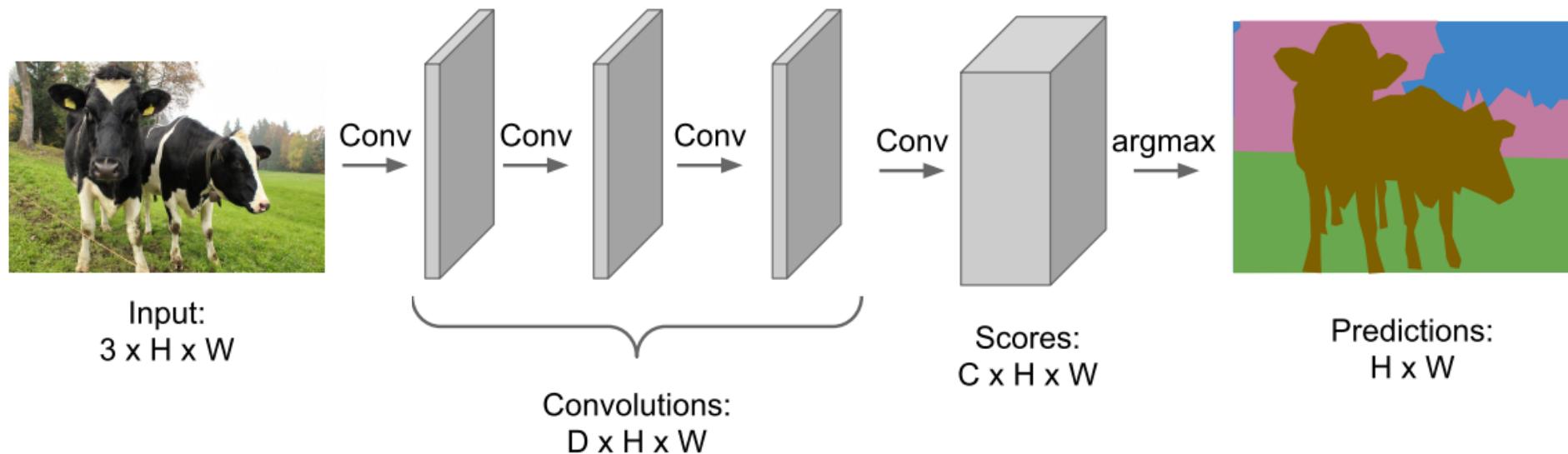
Farabet et al, "Learning Hierarchical Features for Scene Labeling," TPAMI 2013

Pinheiro and Collobert, "Recurrent Convolutional Neural Networks for Scene Labeling", ICML 2014

- Problem: Very inefficient
 - not reusing shared features between overlapping patches
- Solution: Fully convolutional networks

Fully convolutional networks: Idea

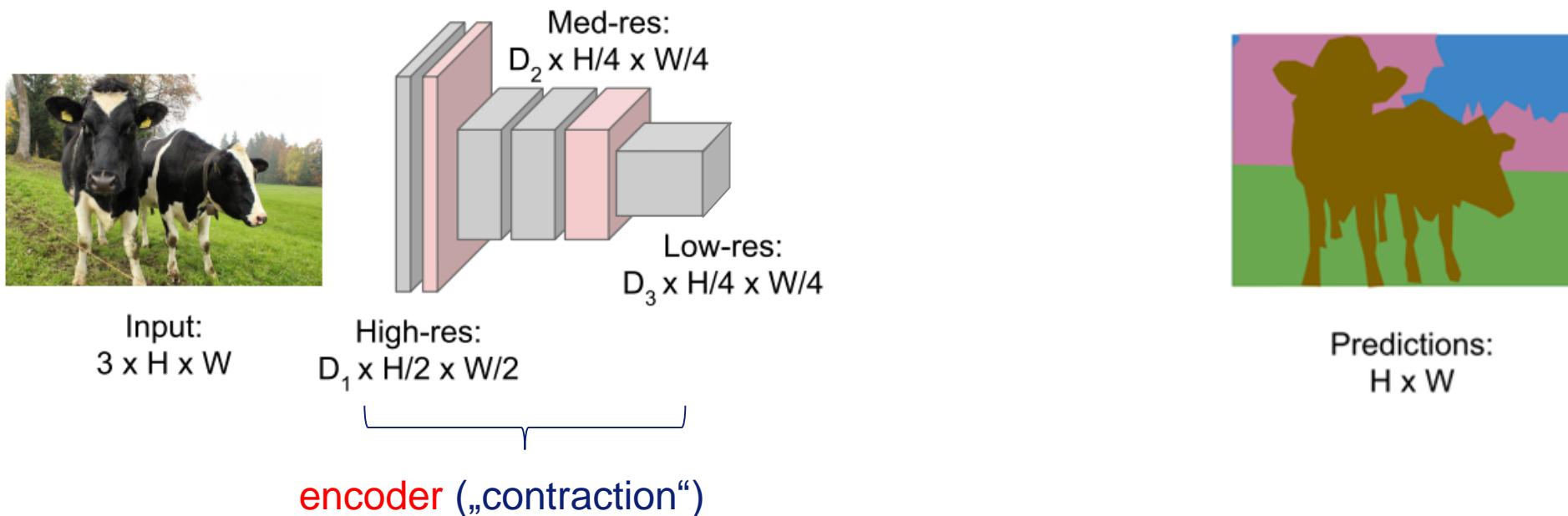
- Idea: Design a **network as bunch of convolution layers** to make predictions for pixels all at once
 - In particular: Remove fully connected layers!
- Naive approach:



- Problems:
 - Large filters are computationally too expensive
 - Small filters may not capture enough image detail (receptive field too small)
- **Downsampling**: Pooling or strided convolution

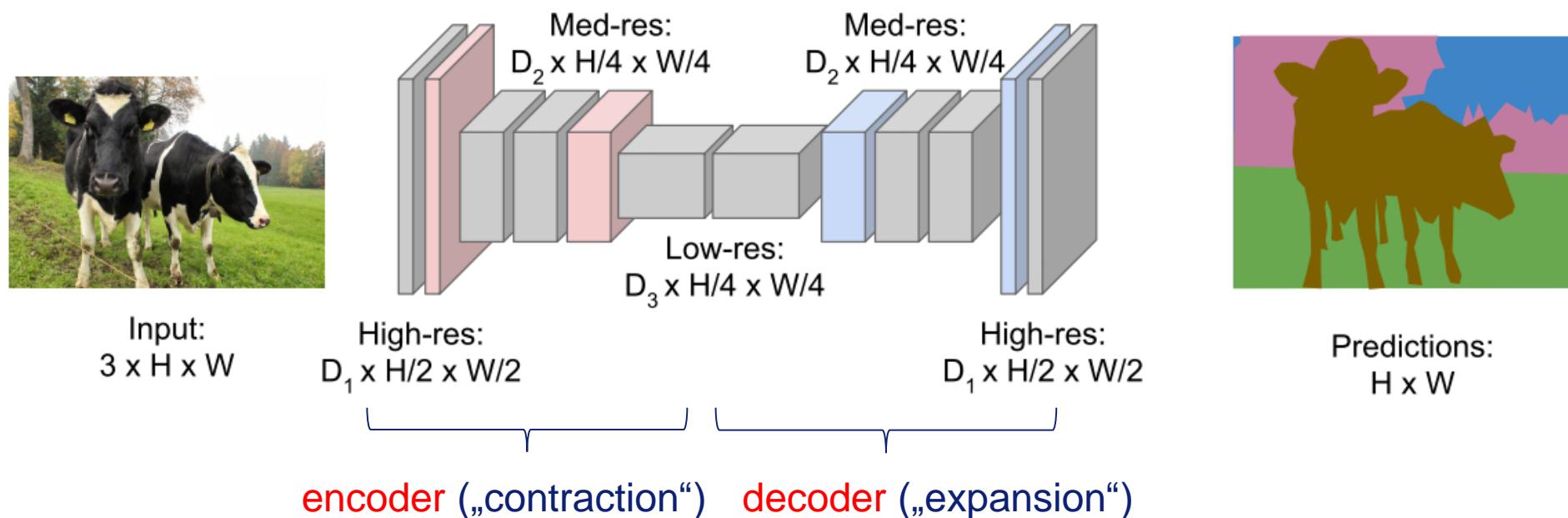
Principle of encoder-decoder architectures

- So far: CNN computes a **low-resolution image representation („encoder“)**
 - Covering sufficiently large receptive field due to multiple convolutions / pooling
 - Idea: Use representation to **predict labels for all pixels at once („decoder“)**
 - But: Must restore original image resolution, i.e. „undo“ pooling / downsampling
- For each downsampling (pooling), apply a corresponding „upsampling“
- Interspersed with convolutional layers as before, i.e. „mirror“ the CNN



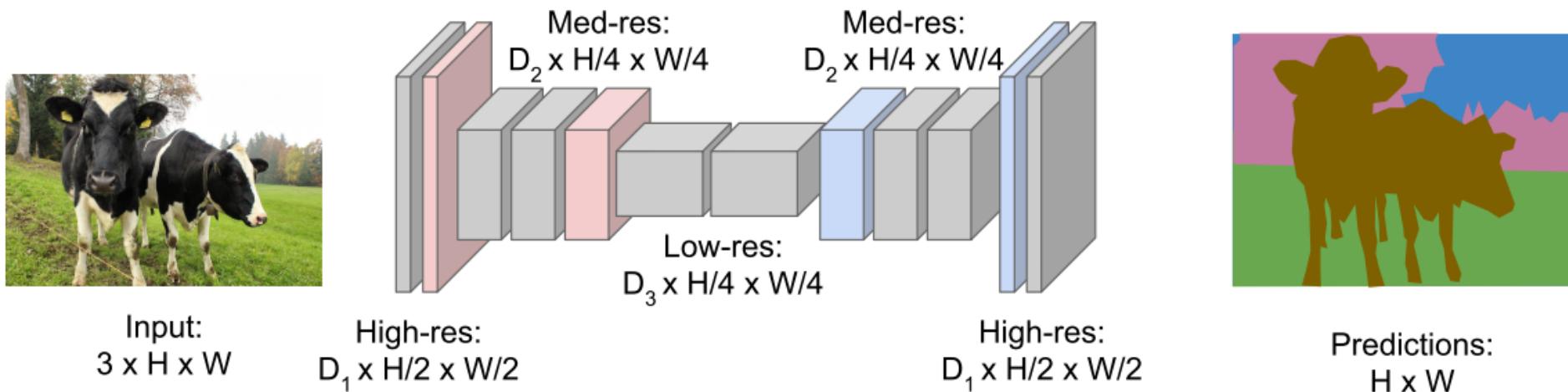
Principle of encoder-decoder architectures

- So far: CNN computes a **low-resolution image representation** („encoder“)
 - Covering sufficiently large receptive field due to multiple convolutions / pooling
 - Idea: Use representation to **predict labels for all pixels at once** („decoder“)
 - But: Must restore original image resolution, i.e. „undo“ pooling / downsampling
- For each downsampling (pooling), apply a corresponding „upsampling“
 - Interspersed with convolutional layers as before, i.e. „mirror“ the CNN



Principle of encoder-decoder architectures

- So far: CNN computes a **low-resolution image representation („encoder“)**
 - Covering sufficiently large receptive field due to multiple convolutions / pooling
 - Idea: Use representation to **predict labels for all pixels at once („decoder“)**
 - But: Must restore original image resolution, i.e. „undo“ pooling / downsampling
- For each downsampling (pooling), apply a corresponding „upsampling“
- Interspersed with convolutional layers as before, i.e. „mirror“ the CNN



- But: How to „upsample“, restoring enough image detail?

Upsampling: Strategies

- **Interpolation**
 - Fill in missing values in high-resolution data by interpolation (e.g. bilinear)
 - Not very exact / detailed; no learning of interpolation filters
- **Max unpooling**
 - Remember which elements in high-resolution feature map yielded maximum
 - To those positions the values from low-resolution data are copied, rest set to 0
- **Transposed convolution („learnable upsampling“)**
 - Use **(learned) filter** to get high-resolution output from low-resolution input
 - Other names: Upconvolution, „deconvolution“ (bad), backward strided convol., fractionally strided convolution

Normal convolution $*$:

$$X * W = z$$

input X : high-resolution input
 filter W : filter kernel
 output z : low-resolution output

Transposed convolution $*^T$:

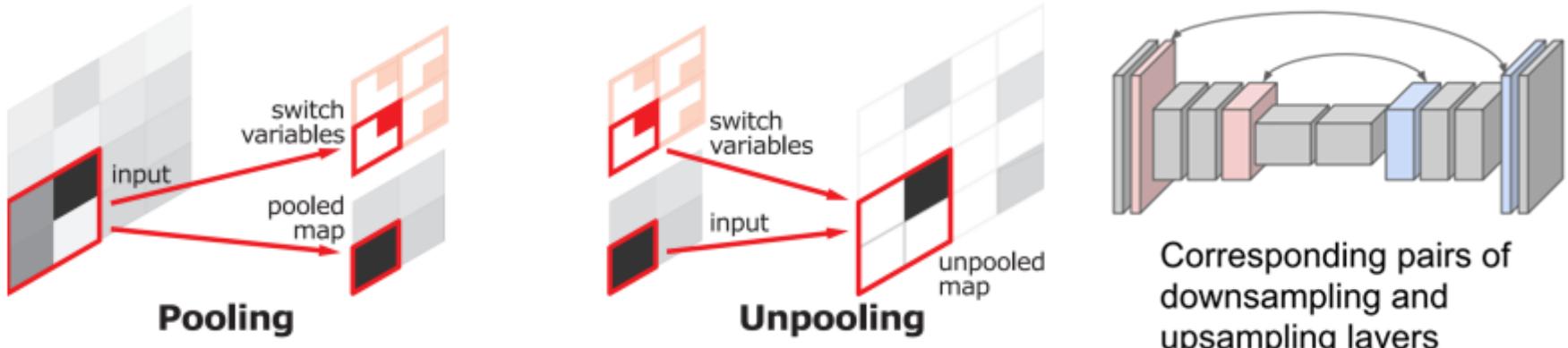
$$z *^T W^T = X$$

input z : low-resolution input
 filter W : filter kernel (transposed)
 output X : high-resolution output

Note: Filter matrix W here as spatial kernel (not vectorized), see subsequent examples

Upsampling: Max unpooling - illustration

- Remember which elements in high-resolution feature map yielded maximum
- To those positions the values from low-resolution data are copied, rest set to 0

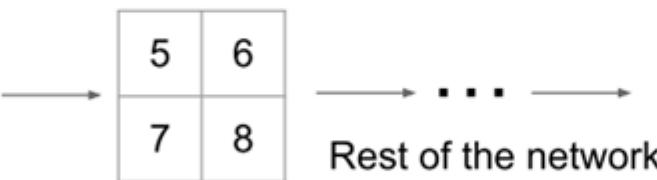


Max Pooling

Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

Input: 4 x 4



Output: 2 x 2

Max Unpooling

Use positions from pooling layer

1	2
3	4

Input: 2 x 2

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

Output: 4 x 4

Illustration: Normal convolution (cross-correl.) as matrix multiplication

Input image (3 x 3): single filter (2 x 2)

output (2 x 2):

X_1	X_2	X_3
X_4	X_5	X_6
X_7	X_8	X_9

*
convol.

w_1	w_2
w_3	w_4



Z_1	Z_2
Z_3	Z_4

 W^T

filter dim.

matrix multipl.

im2col(X):

X_1	X_2	X_4	X_5
X_2	X_3	X_5	X_6
X_4	X_5	X_7	X_8
X_5	X_6	X_8	X_9

filter dim.

 z^T

#patches

z_1	z_2	z_3	z_4
-------	-------	-------	-------

#filter

$$\begin{aligned}
 z_1 &= w_1 x_1 + w_2 x_2 + w_3 x_4 + w_4 x_5 \\
 z_2 &= w_1 x_2 + w_2 x_3 + w_3 x_5 + w_4 x_6 \\
 z_3 &= w_1 x_4 + w_2 x_5 + w_3 x_7 + w_4 x_8 \\
 z_4 &= w_1 x_5 + w_2 x_6 + w_3 x_8 + w_4 x_9
 \end{aligned}$$

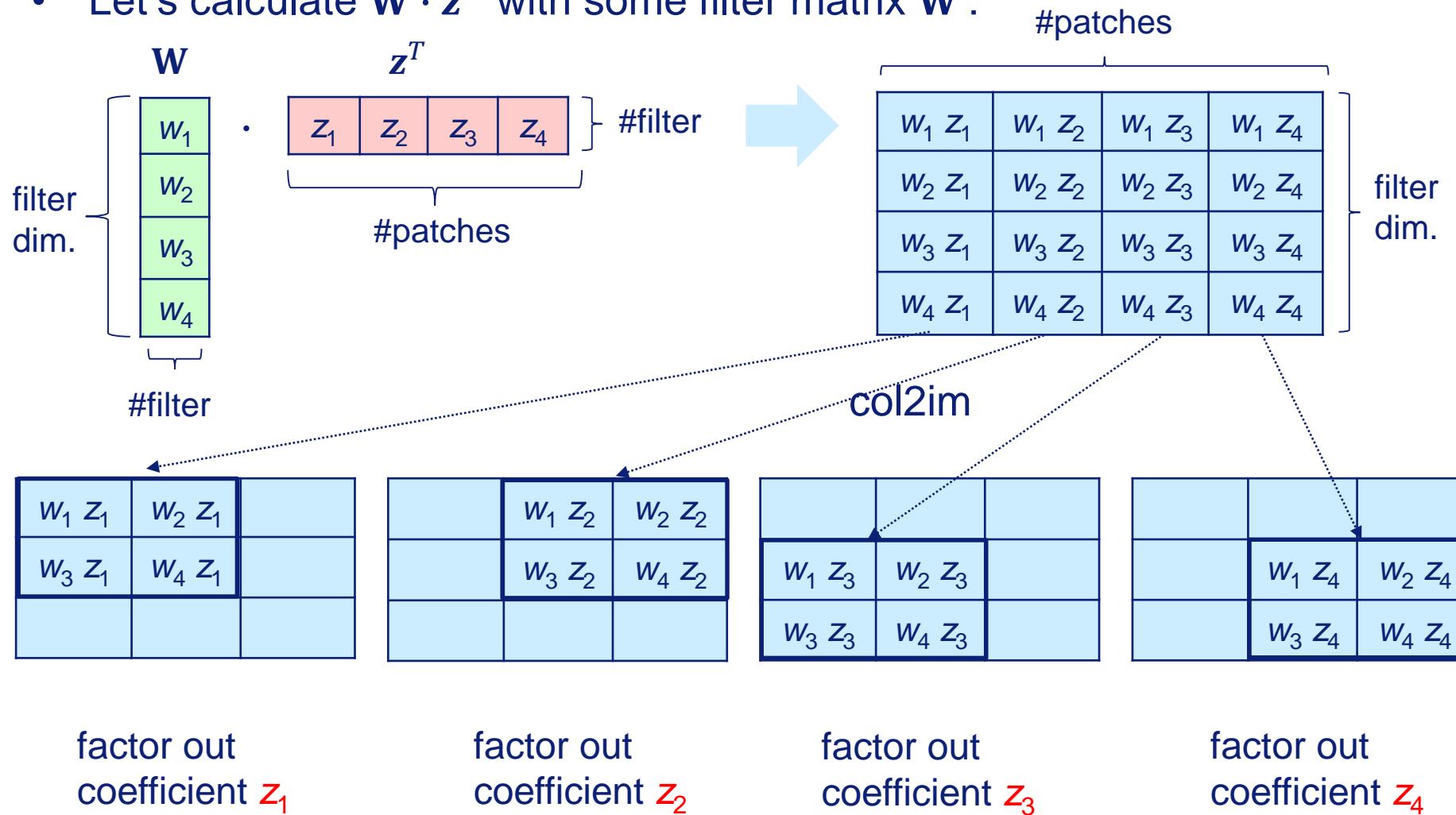
$$W^T \cdot \text{im2col}(X) = z^T \quad (\text{#filter} \times \text{filter dim.}) \cdot (\text{filter dim.} \times \text{#patches}) = (\text{#filter} \times \text{#patches})$$

$$\text{im2col}(X') = W' \cdot z^T \quad (\text{filter dim.} \times \text{#patches}) = (\text{filter dim.} \times \text{#filter}) \cdot (\text{#filter} \times \text{#patches})$$

→ multiply „low-dimensional“ vector z^T with matrix W' gives „high-dimensional“ output X'

Illustration: Transposed convolution

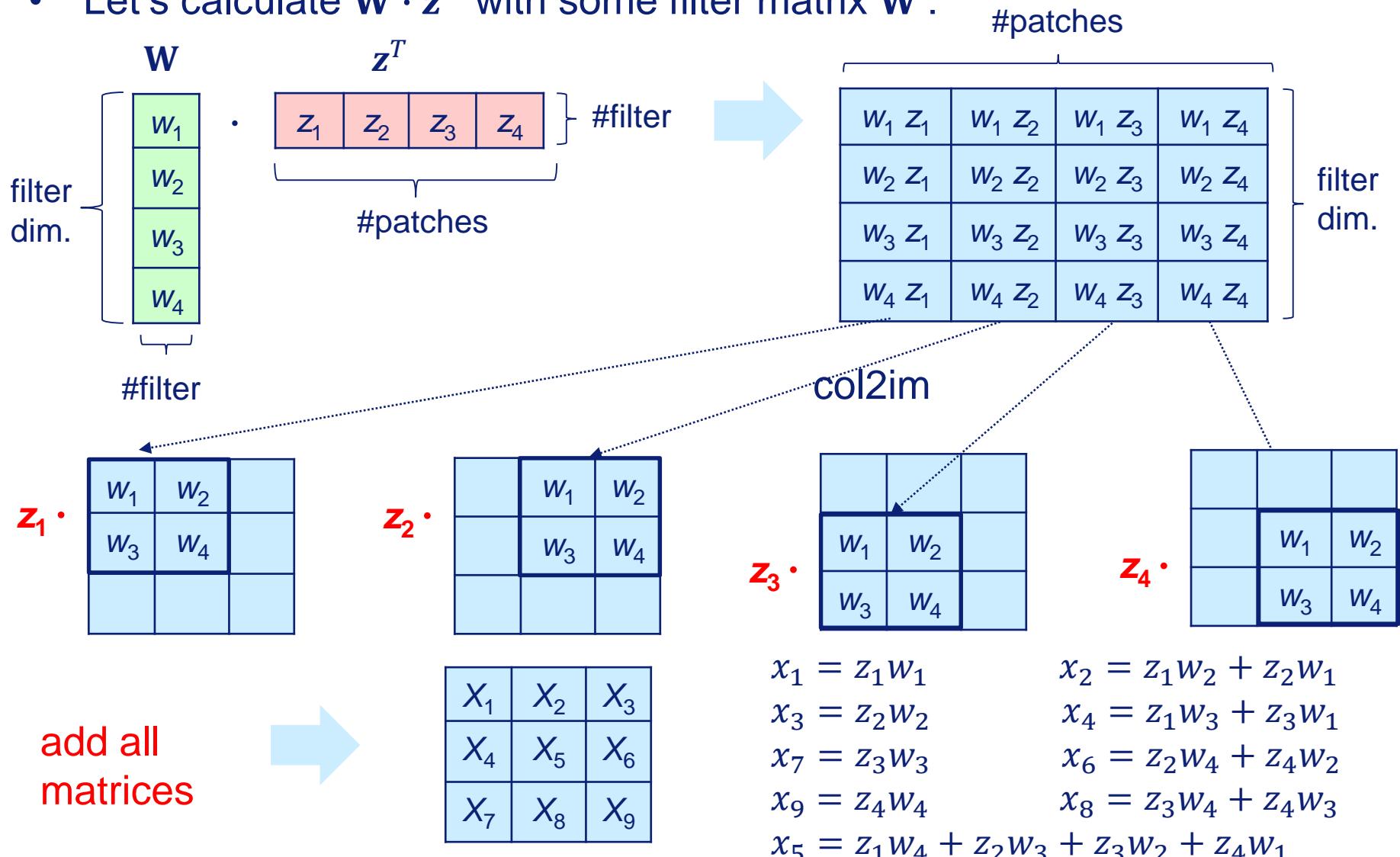
- Let's calculate $\mathbf{W} \cdot \mathbf{z}^T$ with some filter matrix \mathbf{W} :



It's NOT the math. inverse of convolution

Illustration: Transposed convolution → no „deconvolution“!

- Let's calculate $\mathbf{W} \cdot \mathbf{z}^T$ with some filter matrix \mathbf{W} :



Other example: Normal convolution (cross-correlation)

Input image (4 x 4):

4	5	8	7
1	8	8	8
3	6	6	4
6	5	7	8

single filter (3 x 3)

1	4	1
1	4	3
3	3	1

output (2 x 2):

122	148
126	134

*

Mask 1

4	5	8	7
1	8	8	8
3	6	6	4
6	5	7	8

*

1	4	1	0
1	4	3	0
3	3	1	0
0	0	0	0

dot product

122	

Mask 2

0	1	4	1
0	1	4	3
0	3	3	1
0	0	0	0

dot product

	148

Mask 3

0	0	0	0
1	4	1	0
1	4	3	0
3	3	1	0

dot product

126	

Mask 4

0	0	0	0
0	1	4	1
0	1	4	3
0	3	3	1

dot product

	134

It's NOT the math. inverse of convol.
→ no „deconvolution“!

Other example: Transposed convolution

Input (2 x 2):

2	1
4	4

single filter (3 x 3)

1	4	1
1	4	3
3	3	1

output (4 x 4):

2	9	6	1
6	29	30	7
10	29	33	13
12	24	16	4

$*^T$

mask
weight

2	

Mask 1

1	4	1	0
1	4	3	0
3	3	1	0
0	0	0	0

copy filter (weighted)

2	8	2	0
2	8	6	0
6	6	2	0
0	0	0	0

mask
weight

		1	

Mask 2

0	1	4	1
0	1	4	3
0	3	3	1
0	0	0	0

copy filter (weighted)

0	1	4	1
0	1	4	3
0	3	3	1
0	0	0	0

mask
weight

4	

Mask 3

0	0	0	0
1	4	1	0
1	4	3	0
3	3	1	0

copy filter (weighted)

0	0	0	0
4	16	4	0
4	16	12	0
12	12	4	0

mask
weight

		4	

Mask 4

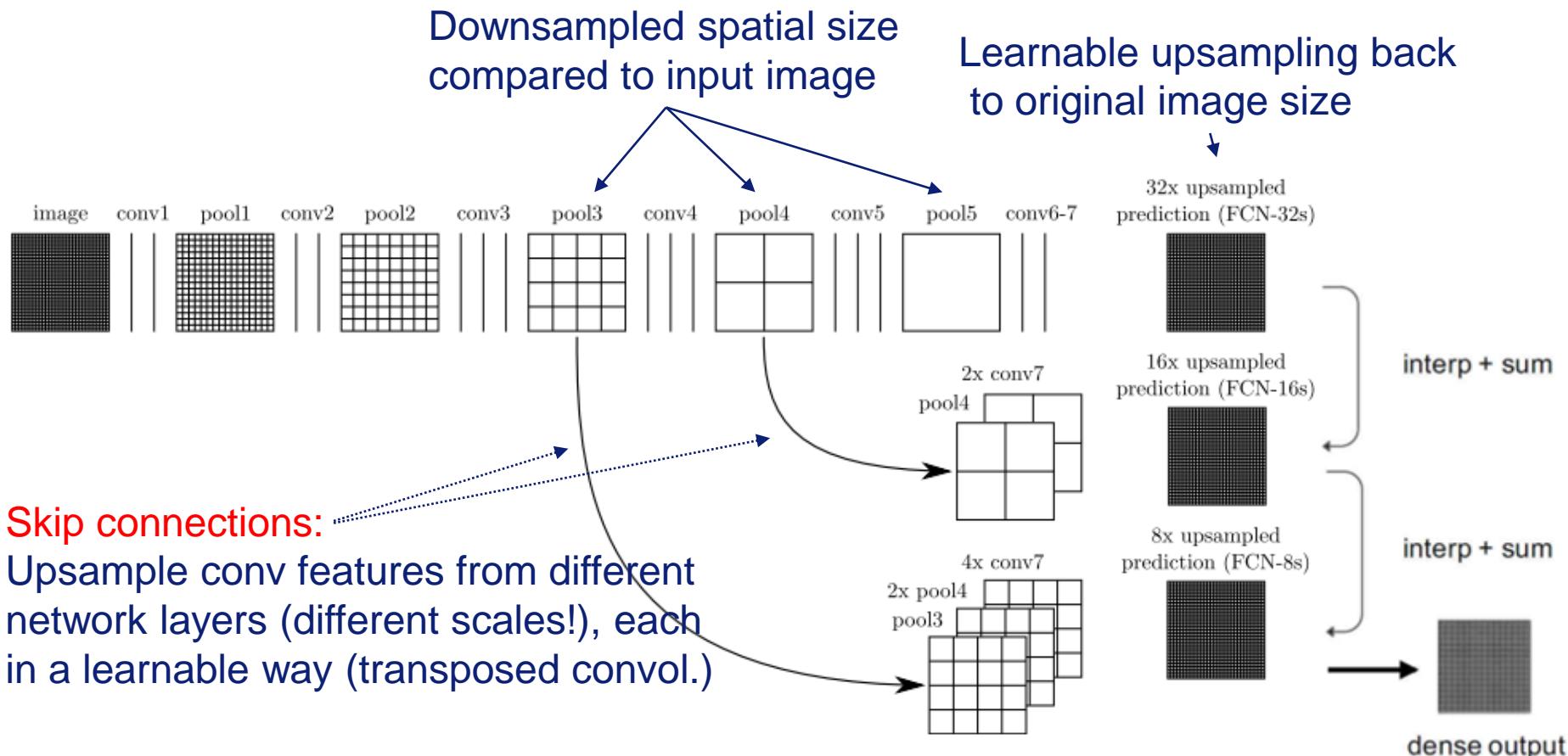
0	0	0	0
0	1	4	1
0	1	4	3
0	3	3	1

copy filter (weighted)

0	0	0	0
0	4	16	4
0	4	16	12
0	12	12	4

Fully Convolutional Network (FCN; Long, Shelhamer, Darrell, 2015)

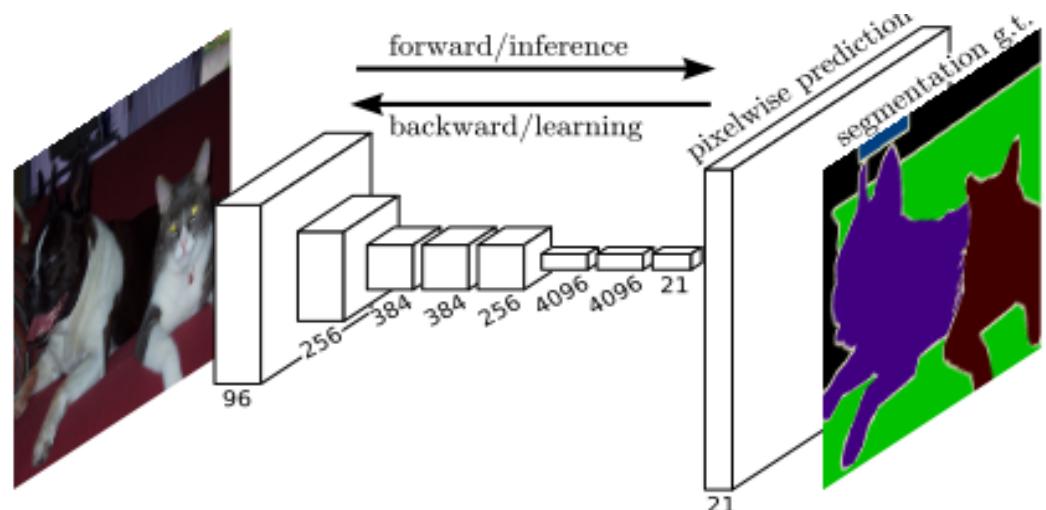
- Learn upsampling as part of the network (using transposed convolution)
- Additional „skip connections“ to use CONV features at different scales



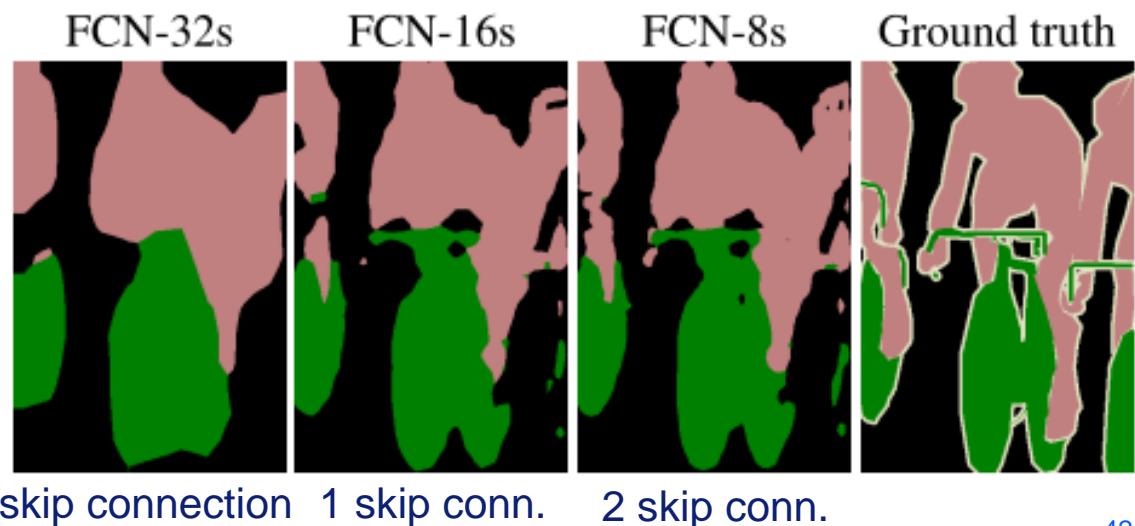
- End-to-end, joint learning of semantics and location

Fully Convolutional Network (FCN; Long, Shelhamer, Darrell, 2015)

- Network overview:



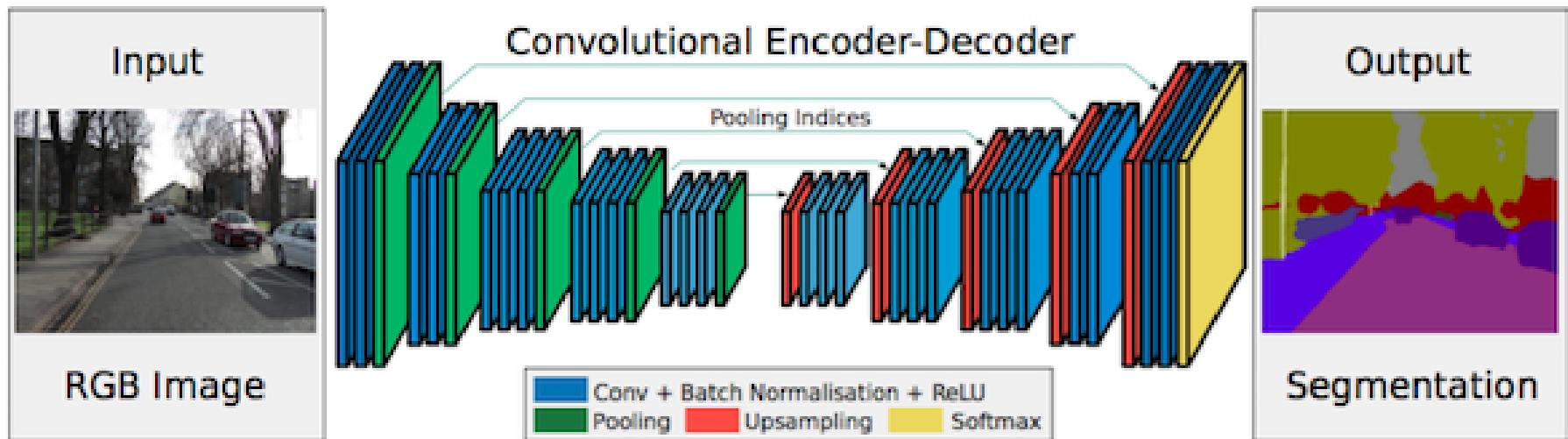
- Adding skip connections gives fine-grained information about spatial locations, helps to clean up boundaries



Skip connections:
Fuse information from layers
with different strides

SegNet (Badrinarayanan et al, 2015)

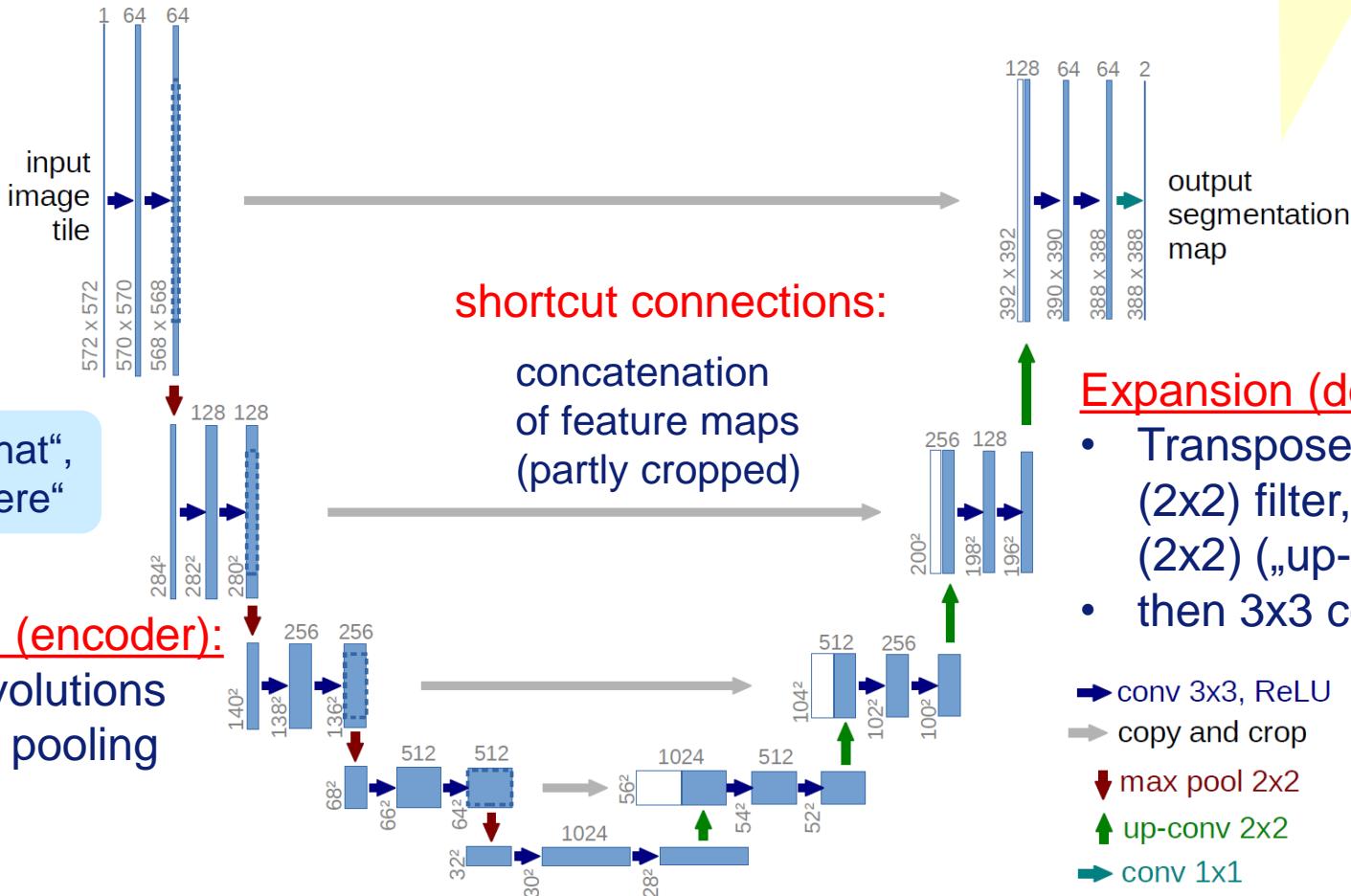
- Decoder: Max unpooling, followed by normal convolutions
 - Output layer: Softmax with as many feature maps as categories
- Label of each pixel given by (pixel-wise) max operation over feature maps



U-Net (Ronneberger et al., 2015)

create high-resolution
segmentation map
(binary: object / backgr.)

- Encoder – decoder architecture

Contraction (encoder):

- 3x3 convolutions
- 2x2 max pooling
- ReLU

- Created with medical image segmentation in mind
 - Heavy data augmentation, dropout layers at the end

up-conv.: counterpart
to max pooling

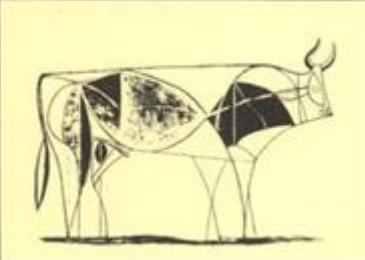
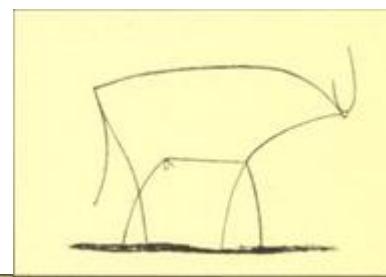
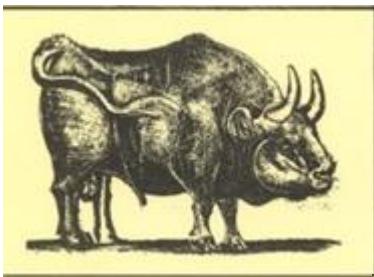
Semantic segmentation: Summary

General problem:

- Convolutions on high-resolution (larger filters) are prohibitively expensive
- Small filters cannot represent image context, e.g. object spatial relations
- Semantic segmentation needs right balance between local (fine-grained) information and global context!
- Encoder-decoder architecture:
 - Encoder gradually reduces the spatial dimension (pooling or stride > 1)
 - Decoder gradually recovers object details and spatial dimension
 - Shortcut connections encoder \rightarrow decoder to help recover object details better
 - Examples: U-Net, SegNet
- Alternative: Dilated / atrous convolution architecture (for global context)
 - Potentially post-processing by conditional random field (for fine-grained details)
 - Examples: DeepLab, Dilated convolution
- Further alternatives: Multi-scale aggregation, additional recurrent network

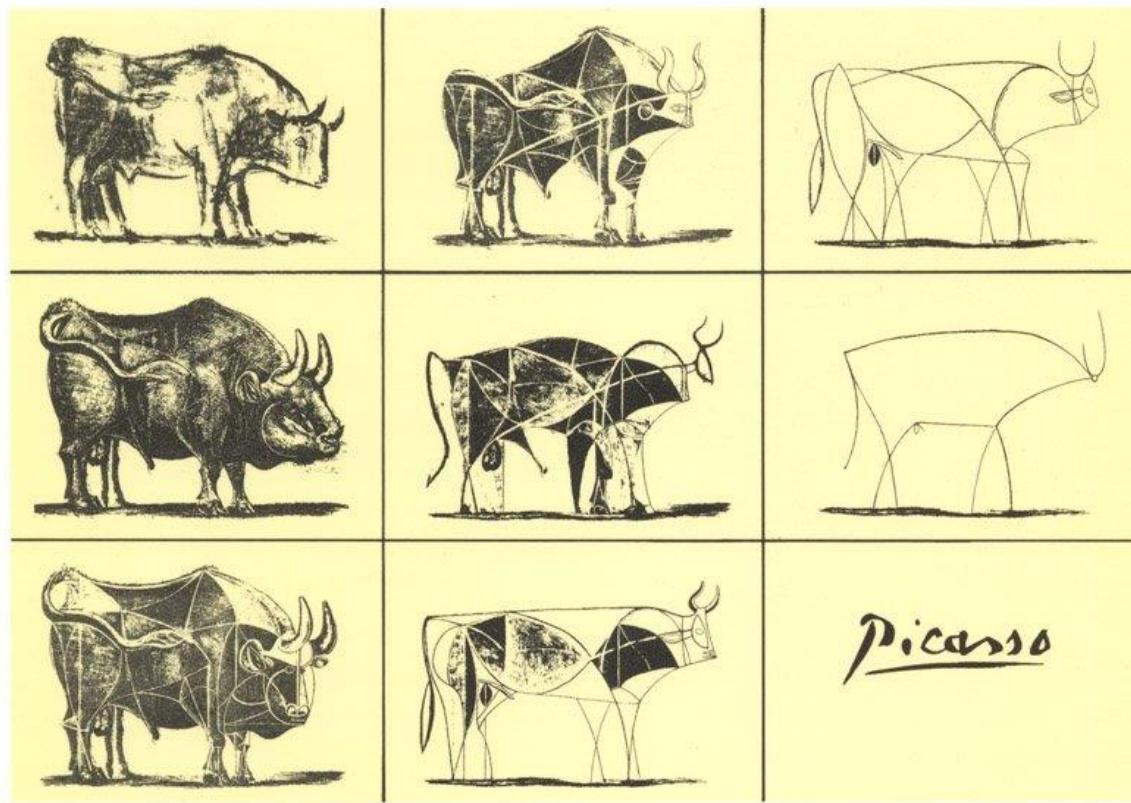
Computer vision: Is the problem solved?

- What do we see?



Computer vision: Is the problem solved?

- What do we see? (from Picasso: Bull series)



- Humans are very good at recognizing **abstract concepts**
 - And neural networks...?
- Find it out yourself: <http://www.clarifai.com> etc....

From: Riedmiller

Computer vision: Is the problem solved? Adversarial examples (1)

original input image

→ CNN output: „panda“



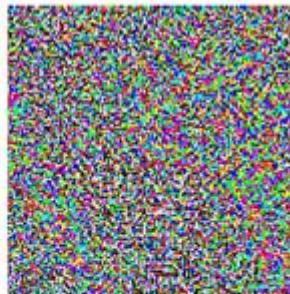
x

“panda”

57.7% confidence

add (selected) noise

$+ .007 \times$



$\text{sign}(\nabla_x J(\theta, x, y))$
“nematode”
8.2% confidence

perturbed input image

→ CNN output: „gibbon“

=



$x + \epsilon \text{sign}(\nabla_x J(\theta, x, y))$
“gibbon”
99.3 % confidence

J : cost / loss function

θ : model parameters

x : input image

y : target value

- **Adversarial example:**
 - Input x' visually similar to an input x but with very different classifier output
 - Images x' and x are undistinguishable to human eye
- Goodfellow: Explanation by **linear behavior in high-dimensional spaces**:
 - $x' = x + \delta x \Rightarrow \text{choose } \delta x = \epsilon \cdot \text{sgn}(\nabla_x J(\theta, x, y))$ **fast gradient sign method**
 - Each pixel is modified only slightly such that the loss increases; due to summing all input dimensions (linear!), this eventually leads to adversarial examples

Computer vision: Is the problem solved? Adversarial examples (2)

- Misclassification stable across architectures trained on different data splits
- Adversarial examples not limited to neural networks or deep learning
 - Other learning machines behave similarly: vulnerable to adversarial examples: Models favoring linearity (networks with ReLU activation functions, LSTMs, logistic regression...)
 - More resistant to adversarial examples: Radial basis function networks
- Modification of even a single pixel may suffice to deceive a network!
 - See e.g. <https://christophm.github.io/interpretable-ml-book/adversarial.html>
- Direction of perturbation (aligned with model weight vector) more important than position, i.e. adversarial examples are not dense in the input space
- Adversarial training (regularisation): Add adversarial examples to the training data to learn more robust models, using modified loss / cost:

$$\tilde{J}(\theta, x, y) = \alpha J(\theta, x, y) + (1 - \alpha) J(\theta, x + \epsilon \text{sign}(\nabla_x J(\theta, x, y)))$$

- „Models that are easy to optimise are easy to perturb“ (Goodfellow)

Convolutional neural networks: Summary

- Convolutional neural network: Neural network with convolution layer
- Deep convolutional neural network: Multiple convolution layers
 - Specific neural architecture for which the concept of deep learning can be applied to
- Key ideas:
 - Local connectivity
 - Parameter sharing
 - Pooling / subsampling
 - Rectified linear units
 - (Local contrast) normalization
- Deep network: Convolutional and pooling layers (+ normalization) alternate
- Multiple feature maps are computed from an input
 - By applying multiple convolutional kernels (filters) which are learned from data
- Since 2006: State-of-the-art in various fields of computer vision and beyond
 - Automatic speech processing, natural language processing ...

} „further
tricks“

RECURRENT NEURAL NETWORKS

Feedforward neural networks for sequences?

- So far: Feedforward networks; fixed input → fixed output: „one to one“
- Simple example (for simplicity with a single hidden layer only):



Fixed-sized output vector $\hat{y} = a^2$

hidden layer

Fixed-sized input vector x

$$\begin{aligned} z^2 &= W^2 a^1 + b^2 \\ a^2 &= f^2(z^2) = f^2(W^2 a^1 + b^2) \end{aligned}$$

$$\begin{aligned} z^1 &= W^1 x + b^1 \\ a^1 &= f^1(z^1) = f^1(W^1 x + b^1) \end{aligned}$$

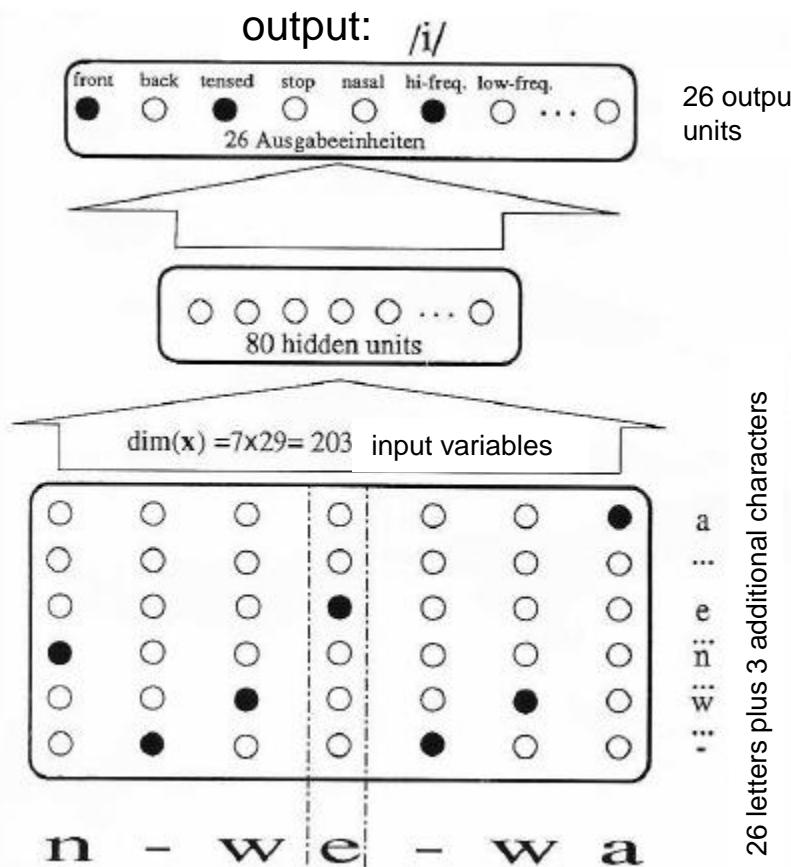
Superscript:
layer index!

x ($=: a^0$)

- How to handle **sequences of variable length** (in input, output or both)?
- Simple approach: Use *fixed number of elements* from sequence as input
 - „sliding window approach“; corresponds to *fixed context length* (not variable!)
- Example: NETtalk

Example feedforward neural network for input sequences: NETtalk

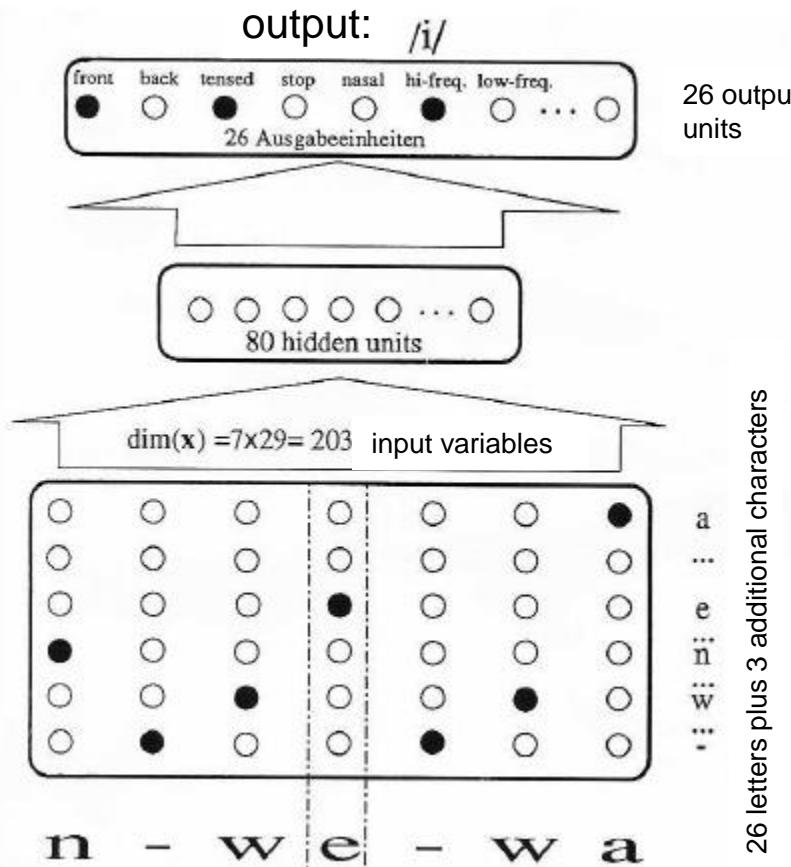
- Given a sequence of 7 characters (letters), predict phonetic pronunciation of middle character (input: letter sequence, output: phoneme)
 - Need to know the letters preceding and following the middle character
- Multi-layer perceptron architecture (Sejnowski & Rosenberg, 1986):



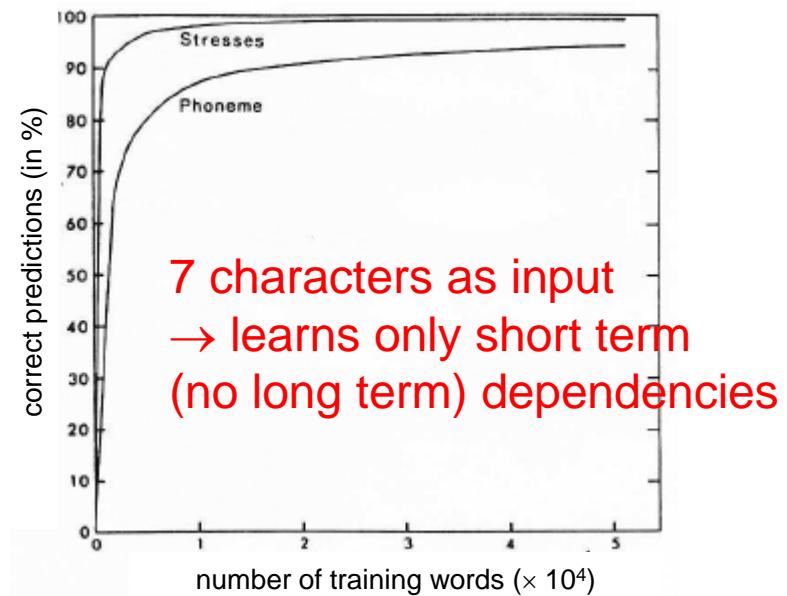
- 29x7 input units
 - 29 letters, context of 7 letters
- 80 hidden units
- 26 output units
- about 18000 weights
- training with several 100 pairs of word / pronunciation
- quality similar to rule-based systems (but: less effort for development!)
- still serious errors...

Example feedforward neural network for input sequences: NETtalk

- Given a sequence of 7 characters (letters), predict phonetic pronunciation of middle character (input: letter sequence, output: phoneme)
 - Need to know the letters preceding and following the middle character
- Multi-layer perceptron architecture (Sejnowski & Rosenberg, 1986):



Number of correctly pronounced words:

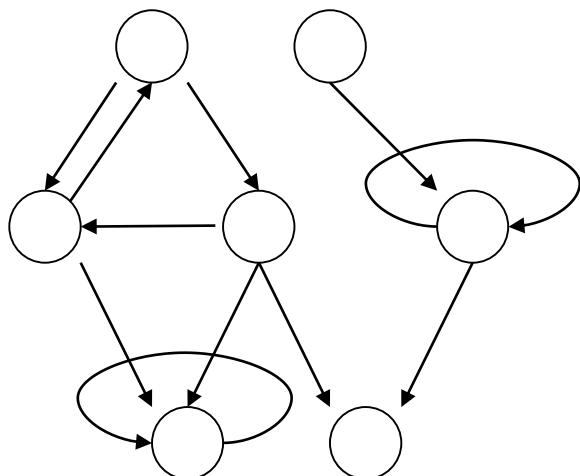


Recurrent neural networks: Motivation

- Goal: **Process sequences of arbitrary length**, e.g.
 - time series prediction: sequence of input data → future element in time series
 - image captioning: image → sequence of words
 - machine translation: sequence of words → sequence of words
- Approach: Add feedback loop(s) → **recurrent neural network**
 - Use state of any neuron (input / hidden / output) as input in next step
 - Neuron states become a function of time
 - same update rule is applied at every time step
 - **sequences of arbitrary length can be handled**
 - **recurrence (in hidden states) allows for longer memory**

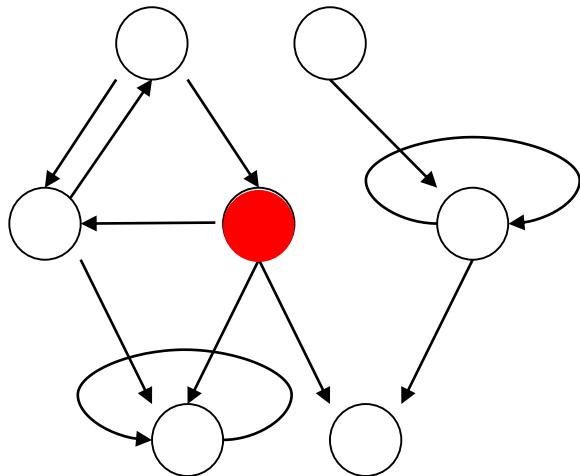
Recurrent neural networks: Feedback

- Networks with direct / indirect / lateral feedback
- If state of any neuron i changes:
 - Influences all other connected neurons (as in feedforward networks)
 - Input of same neuron i later affected via direct / indirect / lateral feedback



Recurrent neural networks: Feedback

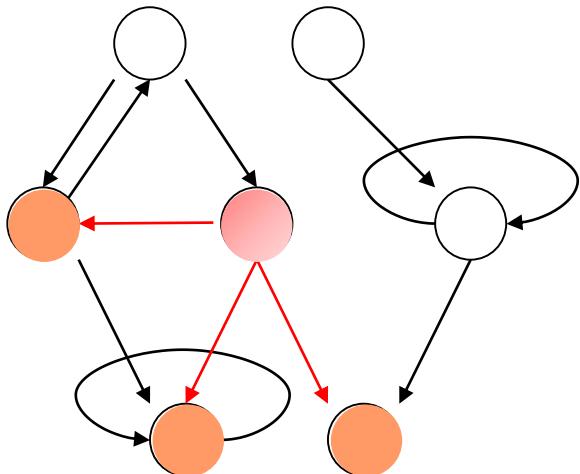
- Networks with direct / indirect / lateral feedback
- If state of any neuron i changes:
 - Influences all other connected neurons (as in feedforward networks)
 - Input of same neuron i later affected via direct / indirect / lateral feedback



Example:
Update neuron i at time t

Recurrent neural networks: Feedback

- Networks with direct / indirect / lateral feedback
- If state of any neuron i changes:
 - Influences all other connected neurons (as in feedforward networks)
 - Input of same neuron i later affected via direct / indirect / lateral feedback

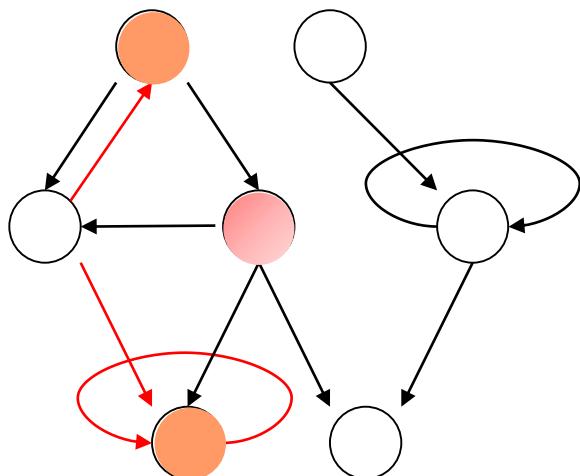


Example:

Affects PSP of other neurons at time $t + 1$
(via synaptic connections)

Recurrent neural networks: Feedback

- Networks with direct / indirect / lateral feedback
- If state of any neuron i changes:
 - Influences all other connected neurons (as in feedforward networks)
 - Input of same neuron i later affected via direct / indirect / lateral feedback

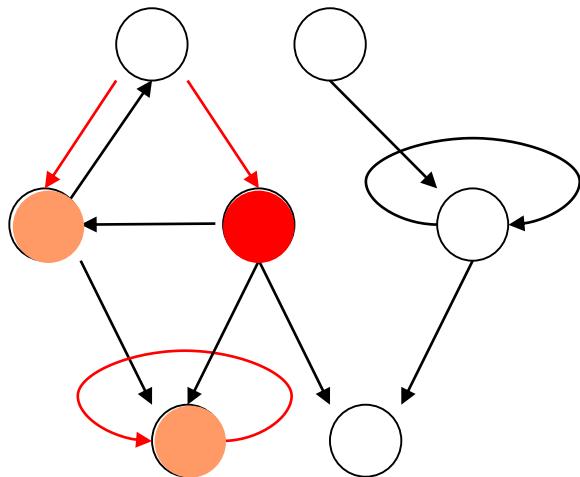


Example:

Affects PSP of other neurons at time $t + 2$
(via synaptic connections)

Recurrent neural networks: Feedback

- Networks with direct / indirect / lateral feedback
- If state of any neuron i changes:
 - Influences all other connected neurons (as in feedforward networks)
 - Input of same neuron i later affected via direct / indirect / lateral feedback



Example:

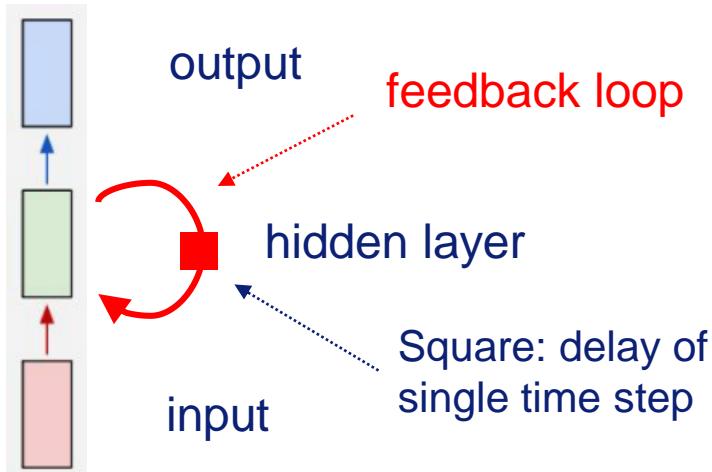
Neuron i affected again at time $t + 3$
(via indirect feedback)

Due to feedback loops:

- The state of any neuron becomes a function of time (indicated by subscript)

Recurrent neural networks: Simple example

- Add feedback loop to hidden layer



Hidden variable a_t^1 is influenced by value a_{t-1}^1 at previous time step:
 $a_t^1 = g(a_{t-1}^1, x_t)$

⇒ variables become a function of time
(indicated by subscript)

Subscript:
time index!

Feedforward network:

$$a^1 = f(z^1) = f(W^1 x + b^1) = g(x)$$



Recurrent network:

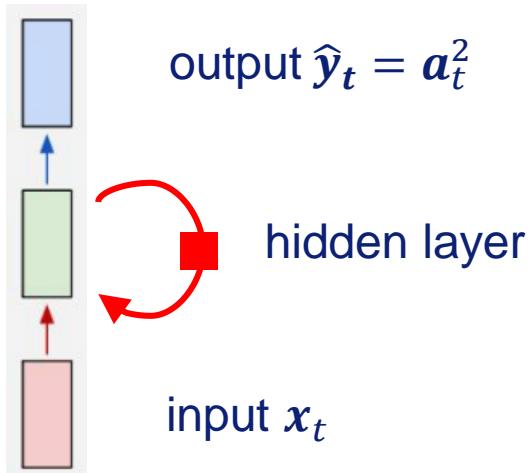
$$a_t^1 = g(a_{t-1}^1, x_t) = f(W^1 x_t + U^1 a_{t-1}^1 + b^1)$$

g : Function of x with parameters W, b

Additional weight matrix U^1 (or W_{hh}) connecting hidden layer at $t-1$ to hidden layer at t (first layer)

Recurrent neural networks: Simple example (mathematical formulation)

- Add feedback loop to hidden layer



$$\begin{aligned} z_t^2 &= \mathbf{W}^2 a_t^1 + \mathbf{b}^2 \\ a_t^2 &= f^2(z_t^2) = f^2(\mathbf{W}^2 a_t^1 + \mathbf{b}^2) \end{aligned}$$

Superscript:
layer index!

$$\begin{aligned} z_t^1 &= \mathbf{W}^1 x_t + \mathbf{U}^1 a_{t-1}^1 + \mathbf{b}^1 \\ a_t^1 &= f^1(z_t^1) = f^1(\mathbf{W}^1 x_t + \mathbf{U}^1 a_{t-1}^1 + \mathbf{b}^1) \end{aligned}$$

x_t

Subscript:
time index!

- Weights between input and hidden layer: \mathbf{W}^1 (or \mathbf{W}_{xh} or \mathbf{W})
- Feedback weights at hidden layer: \mathbf{U}^1 (or \mathbf{W}_{hh} or \mathbf{U})
- Weights between hidden and output layer: \mathbf{W}^2 (or \mathbf{W}_{hy} or \mathbf{V})
- Bias at hidden layer: \mathbf{b}^1
- Bias at output layer: \mathbf{b}^2

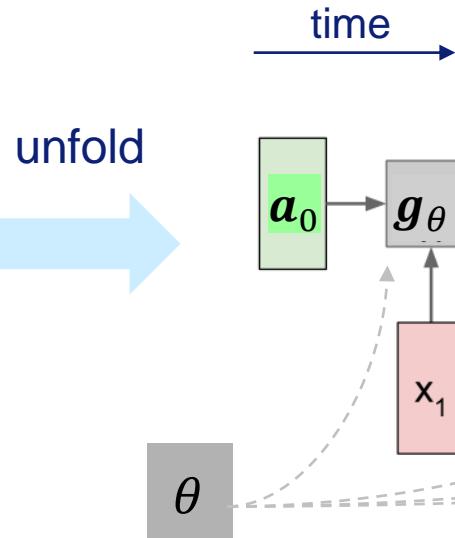
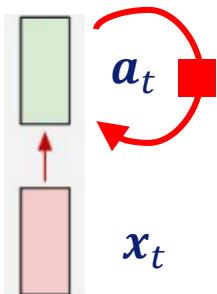
different books / articles
use different notation!

Unfolding the recurrence formula

- For now: only consider input and hidden neuron(s)
- Recurrence formula: $a_t = g_\theta(a_{t-1}, x_t)$ g_θ : function of parameters θ
(weights, biases)
- Apply definition several times, e.g. three times:

$$a_3 = g_\theta(a_2, x_3) = g_\theta(g_\theta(a_1, x_2), x_3) = g_\theta(g_\theta(g_\theta(a_0, x_1), x_2), x_3)$$
whole input sequence x_1, x_2, x_3
- Illustration of unfolding:

circuit
diagram



computational graph:

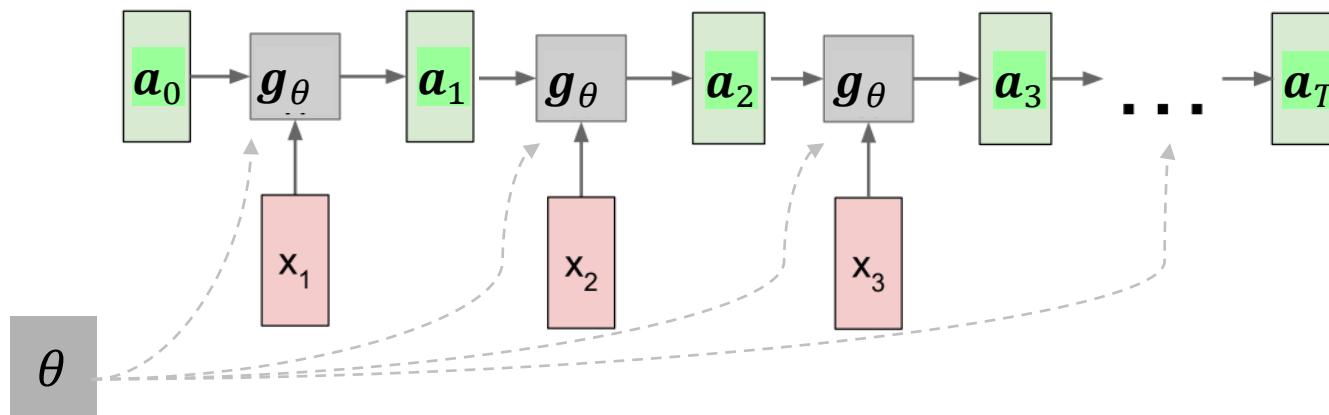
(each node associated with one particular time step)

Subscript:
time index!

- Same function g_θ and same set of parameters θ used at all time steps!
 - Weights and biases do *not* depend on time!

Simple example: Interpretation

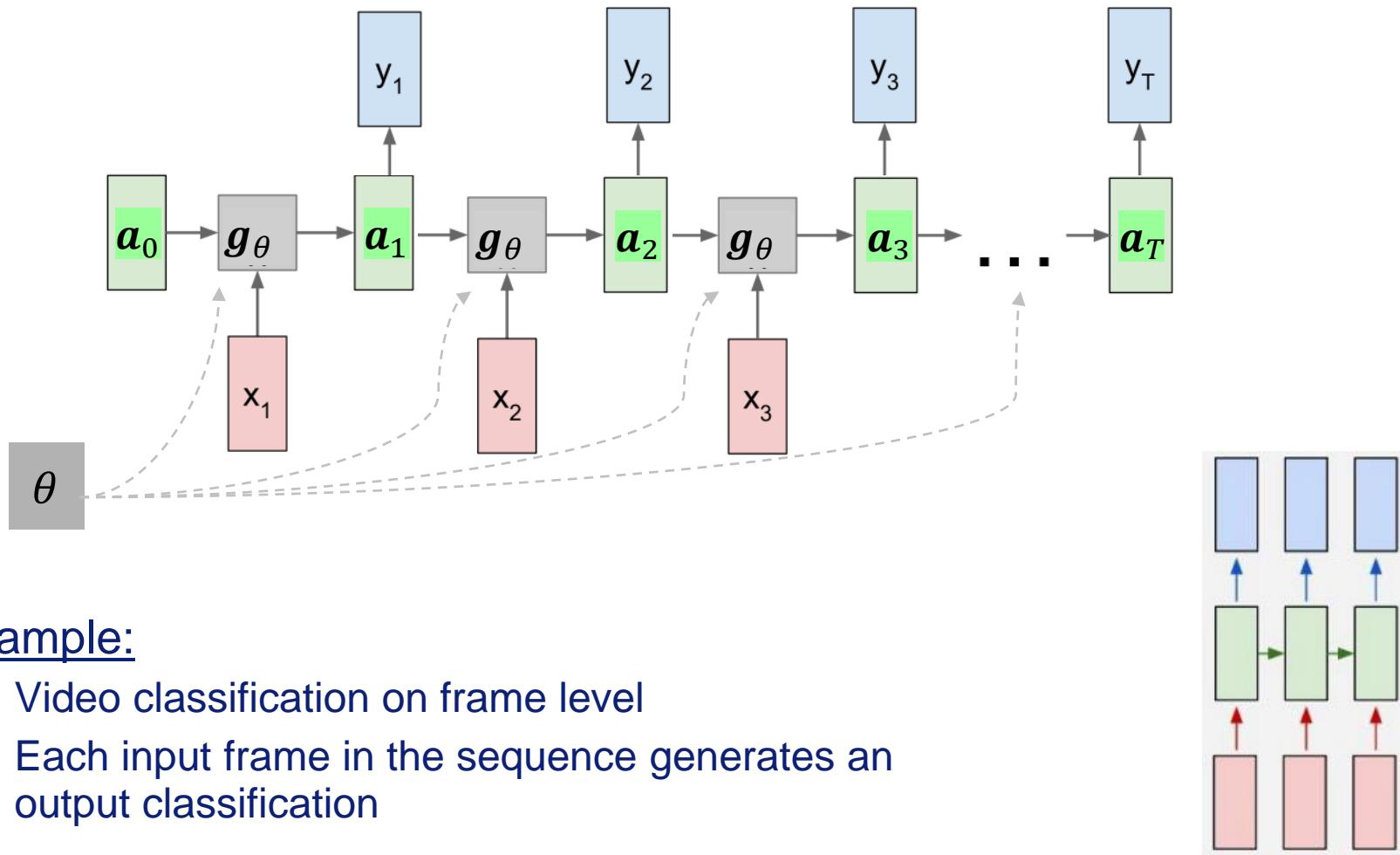
- Network typically learns to use a_t as **lossy summary** of the task-relevant aspects of the past input sequence up to t
 - Lossy since an arbitrary length input sequence $(x_t, x_{t-1}, \dots, x_2, x_1)$ is mapped to a fixed length vector a_t



- **Shared functions / parameters** → less capacity, **better generalization**
 - Uses **prior knowledge** that same function can be applied each time step (e.g NLP)
- Typical RNNs will add extra architectural features such as **output layers** to read information out of the state a to make predictions (see later)
- Note: x and a are **vectors** → may also refer to many input / hidden neurons

RNN computational graph: Many-to-many

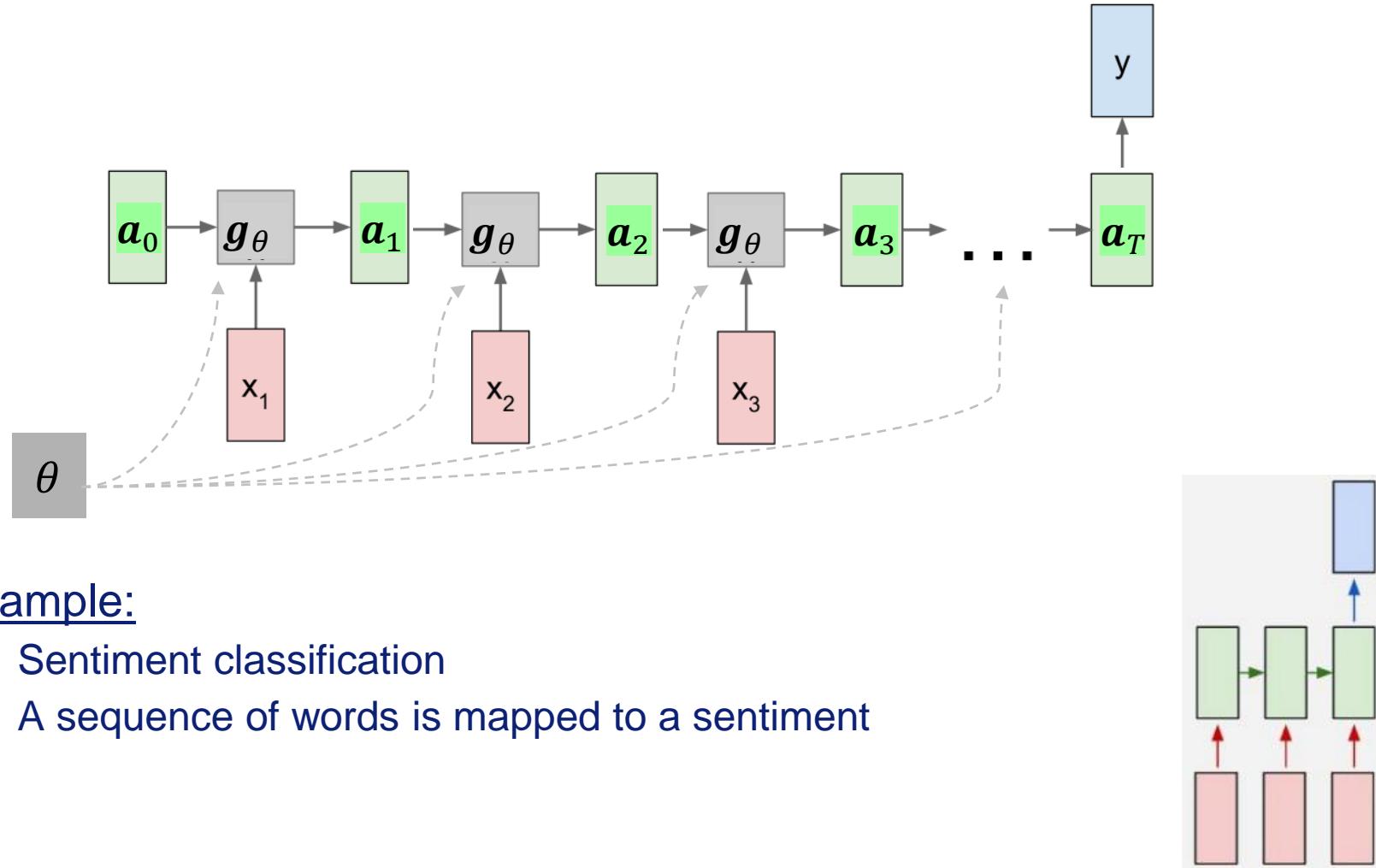
- First variant: Input and output sequence have same length
 - But: Each prediction shall be a function of the whole preceding input sequence



- Example:
 - Video classification on frame level
 - Each input frame in the sequence generates an output classification

RNN computational graph: Many-to-one

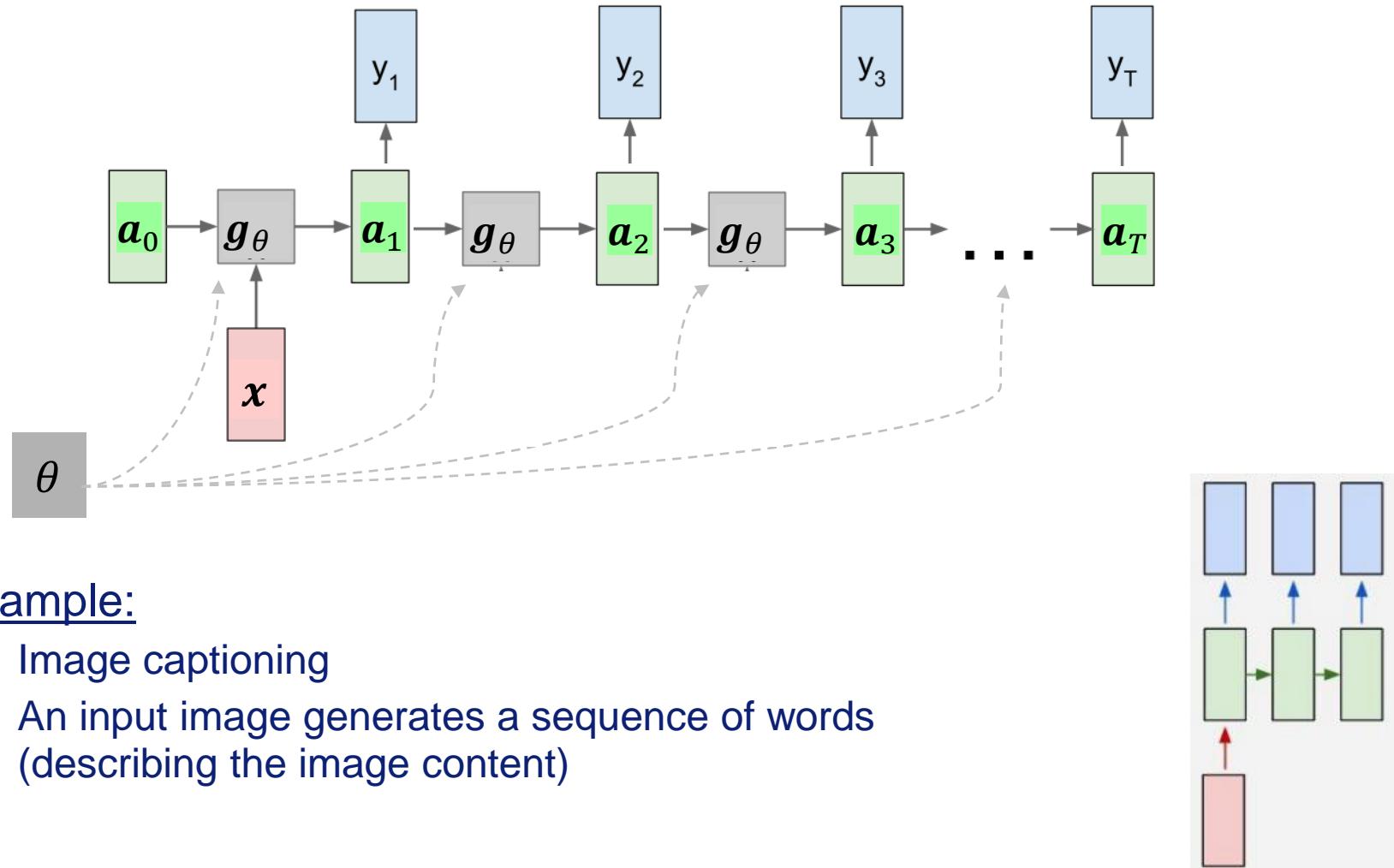
- Input sequence generates a single output



- Example:
 - Sentiment classification
 - A sequence of words is mapped to a sentiment

RNN computational graph: One-to-many

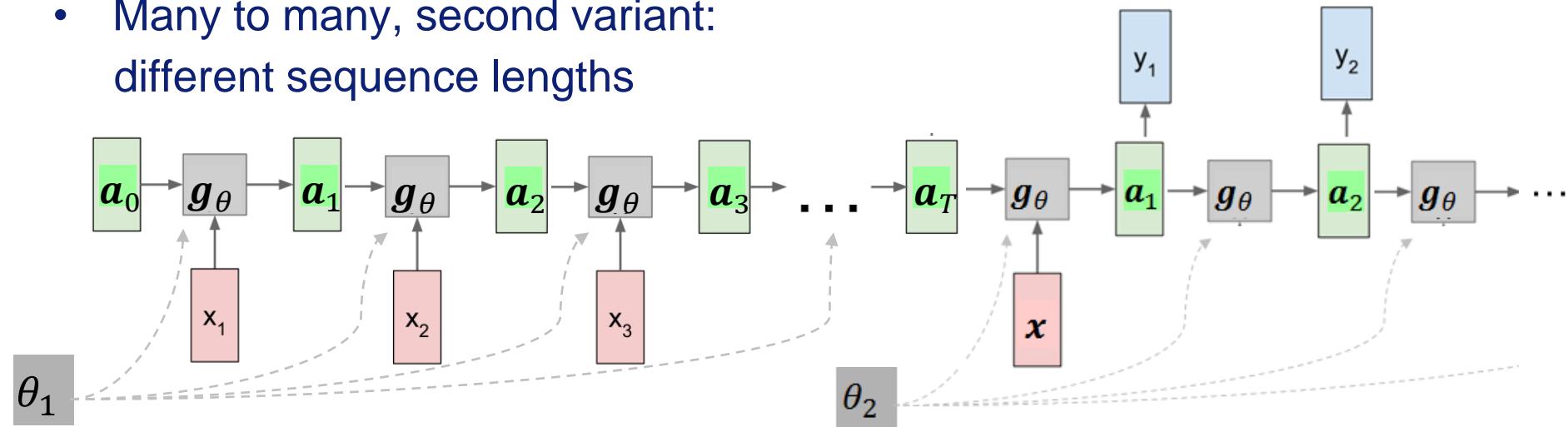
- A single input generates an output sequence



- Example:
 - Image captioning
 - An input image generates a sequence of words (describing the image content)

RNN computational graph: Many-to-one + one-to-many

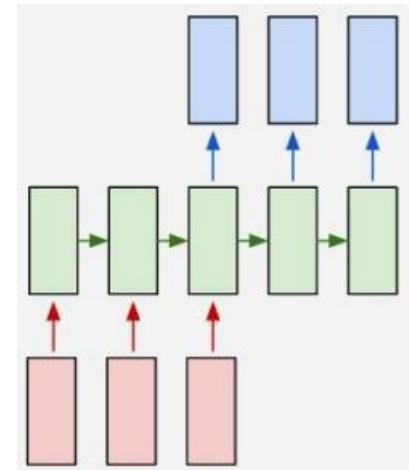
- Many to many, second variant:
different sequence lengths



Many to one: Encode input sequence in a single vector a_T

One to many: Produce output sequence from single input vector a_T

- Example:
 - Machine translation
 - Sequence of words is mapped to sequence of words (potentially with different number of words)



Example: Character-level language model, introduction

- Example vocabulary: [h,e,l,o]
- One-hot encoding:

	input chars: "h"	"e"	"l"	"o"																
input layer	<table border="1"><tr><td>1</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	1	0	0	0	<table border="1"><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>0</td></tr><tr><td>0</td></tr></table>	0	1	0	0	<table border="1"><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>0</td></tr></table>	0	0	1	0	<table border="1"><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>0</td></tr><tr><td>1</td></tr></table>	0	0	0	1
1																				
0																				
0																				
0																				
0																				
1																				
0																				
0																				
0																				
0																				
1																				
0																				
0																				
0																				
0																				
1																				

- Input sequence: „hell“, i.e. $x_1 \cong \text{"h"}$, $x_2 \cong \text{"e"}$, $x_3 \cong \text{"l"}$, $x_4 \cong \text{"l"}$
- Goal: Predict next letter in sequence; here: „o“: „hell“ + „o“ \rightarrow „hello“
 - Idea: Train model on huge text corpus containing relevant words
- Update equations:

$$\mathbf{z}_t^1 = \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1}^1 + \mathbf{b}^1$$

$$\mathbf{a}_t^1 = \tanh(\mathbf{z}_t^1)$$

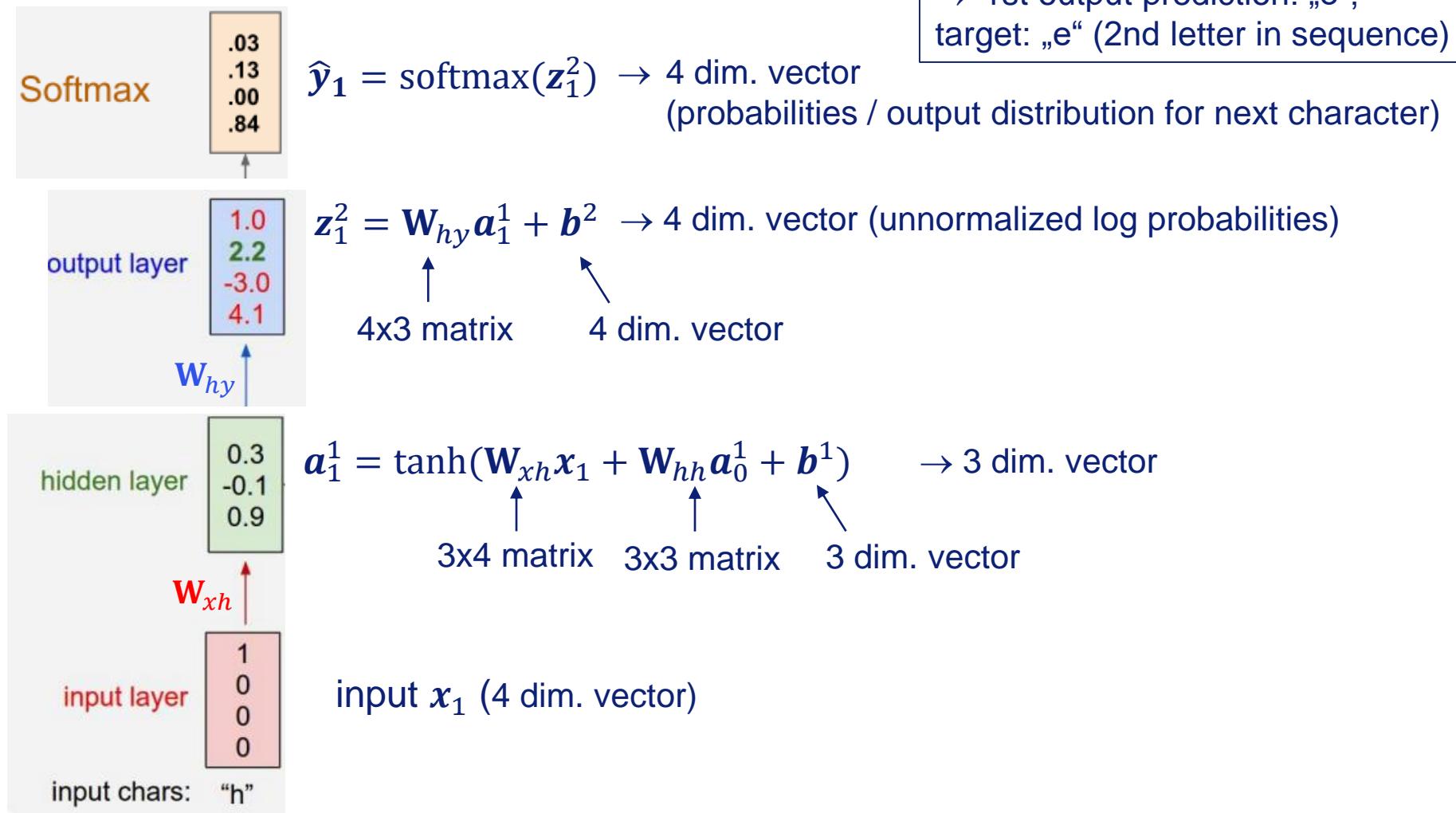
$$\mathbf{z}_t^2 = \mathbf{W}_{hy} \mathbf{a}_t^1 + \mathbf{b}^2 \quad \text{unnormalized log probability}$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{z}_t^2) \quad \text{output probability}$$

Change in notation:
 $\mathbf{W}^1 \rightarrow \mathbf{W}_{xh}$, $\mathbf{U}^1 \rightarrow \mathbf{W}_{hh}$, $\mathbf{W}^2 \rightarrow \mathbf{W}_{hy}$

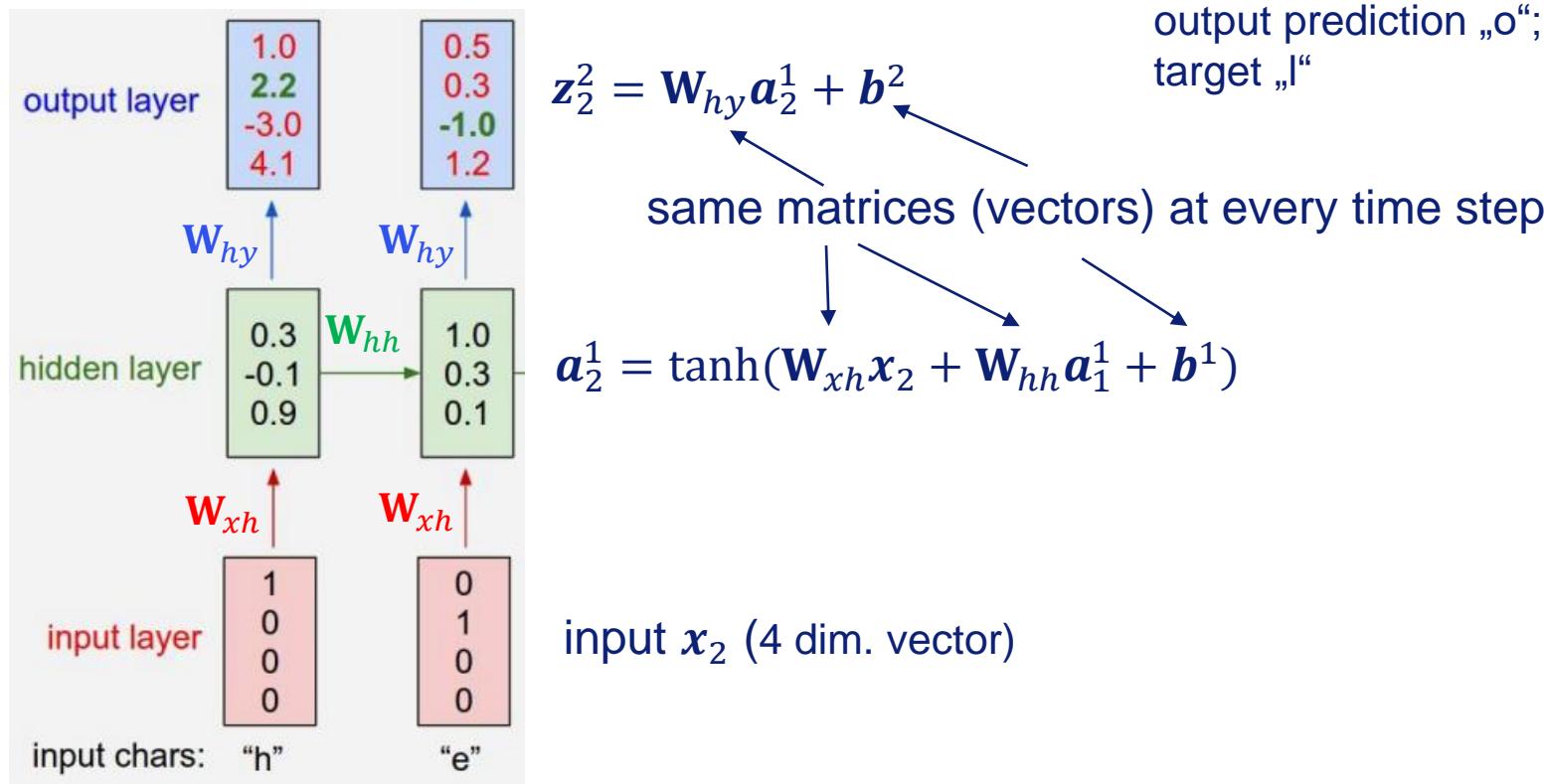
- 3 hidden neurons, i.e. 3-dimensional vector \mathbf{a}_t ; initialisation: $\mathbf{a}_0^1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

Example: Character-level language model, calculations (1)



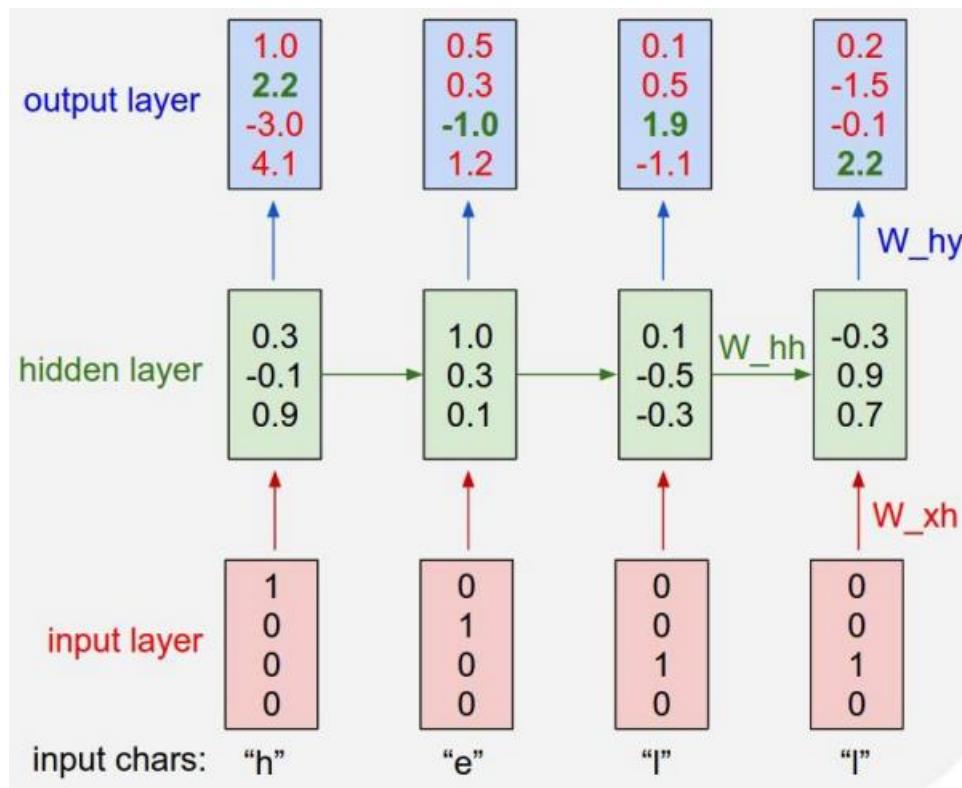
Example: Character-level language model, calculations (2)

- State of hidden neurons at any time point influenced by current input and previous state of hidden neurons
- Output at any time influenced by current state of hidden neurons



Example: Character-level language model, calculations (3)

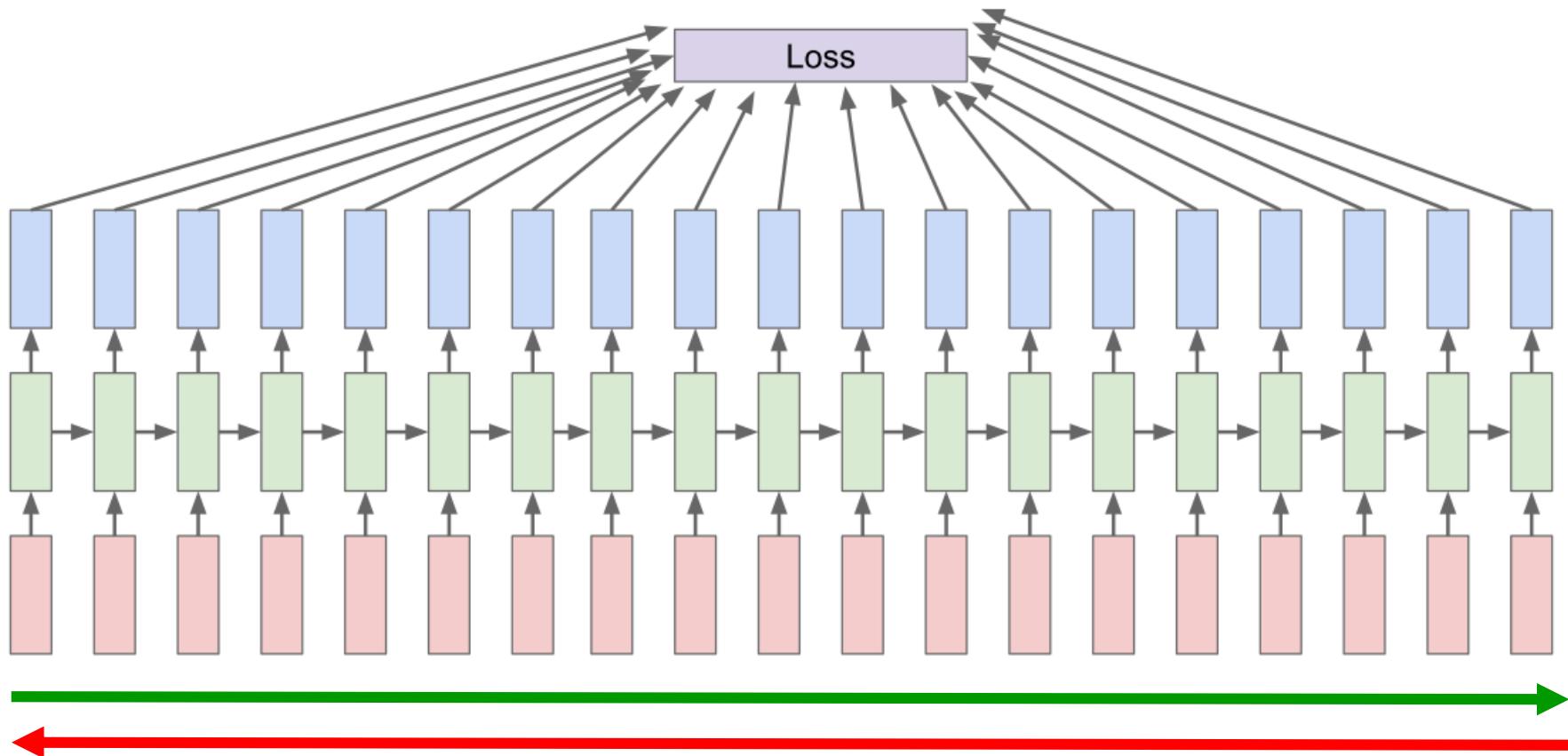
- State of hidden neurons at any time point influenced by current input and previous state of hidden neurons
- Output at any time influenced by current state of hidden neurons



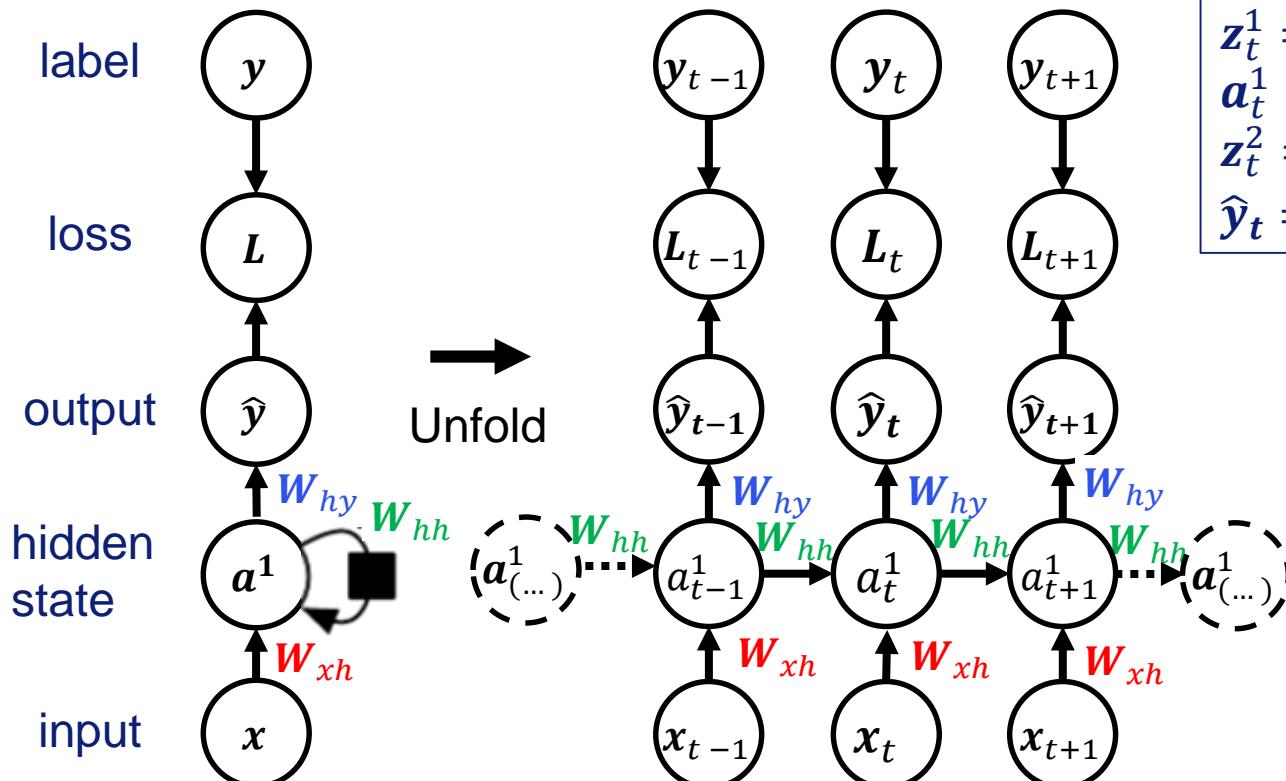
weights and biases must be adjusted such that the output corresponds to the target
 → needs training algorithm
 → „backpropagation through time“

Training RNN: Backpropagation through time

- Principle: Unfold the computational graph and use **backpropagation**
 - Same parameters used at all time steps → **total loss** composed of all time steps
 - must **forward through entire sequence** to compute loss
 - then **backward through entire sequence** to compute gradient



Training RNN: Backpropagation through time



Math. formulae:

$$\begin{aligned} \mathbf{z}_t^1 &= \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1}^1 + \mathbf{b}^1 \\ \mathbf{a}_t^1 &= \tanh(\mathbf{z}_t^1) \\ \mathbf{z}_t^2 &= \mathbf{W}_{hy} \mathbf{a}_t^1 + \mathbf{b}^2 \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{z}_t^2) \end{aligned}$$

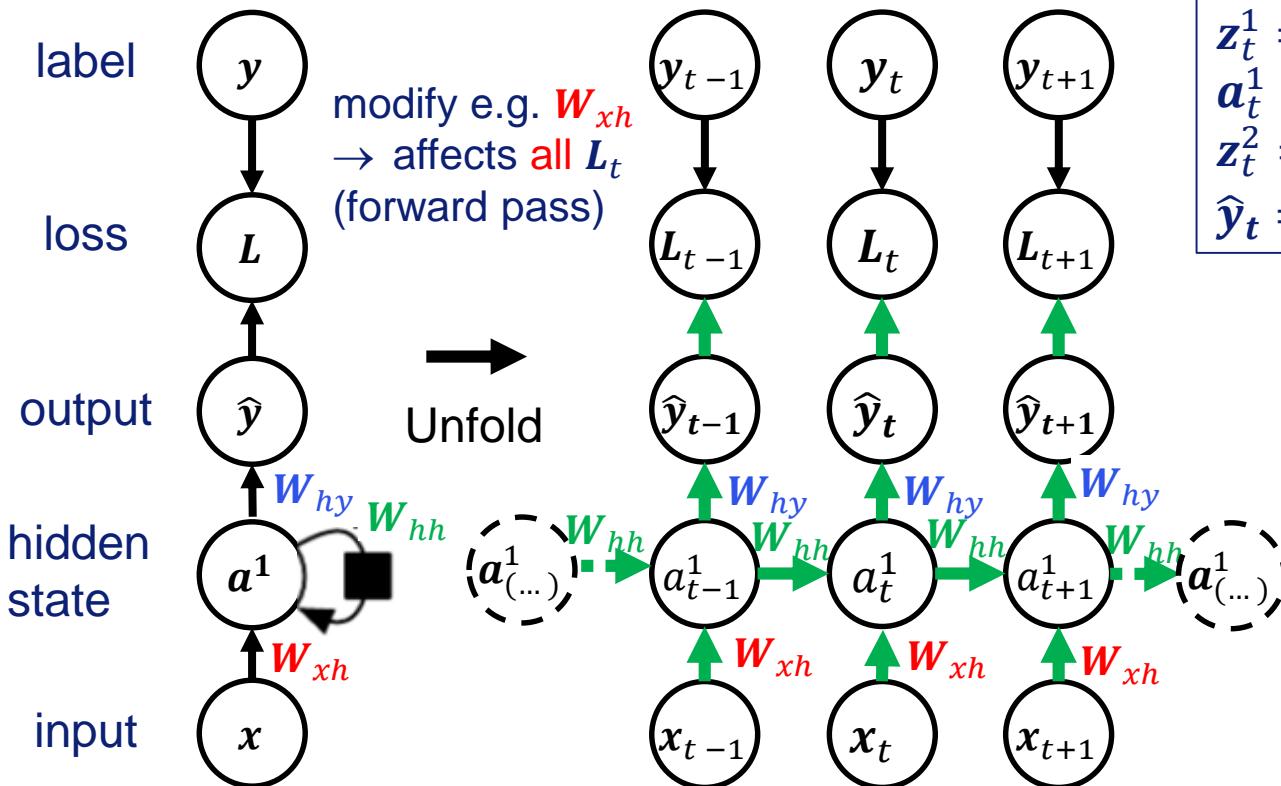
total loss: $L = \sum_{t=1}^T L(\theta, y_t, \hat{y}_t)$

e.g. log-likelihood loss
(for softmax output):

$$L(\theta, y_t, \hat{y}_t) = -\ln \hat{y}_{y_t}$$

↑
Prob. (softmax output)
for correct class y_t at time t

Training RNN: Backpropagation through time



Math. formulae:

$$\begin{aligned} \mathbf{z}_t^1 &= \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1}^1 + \mathbf{b}^1 \\ \mathbf{a}_t^1 &= \tanh(\mathbf{z}_t^1) \\ \mathbf{z}_t^2 &= \mathbf{W}_{hy} \mathbf{a}_t^1 + \mathbf{b}^2 \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{z}_t^2) \end{aligned}$$

total loss: $L = \sum_{t=1}^T L(\theta, y_t, \hat{y}_t)$

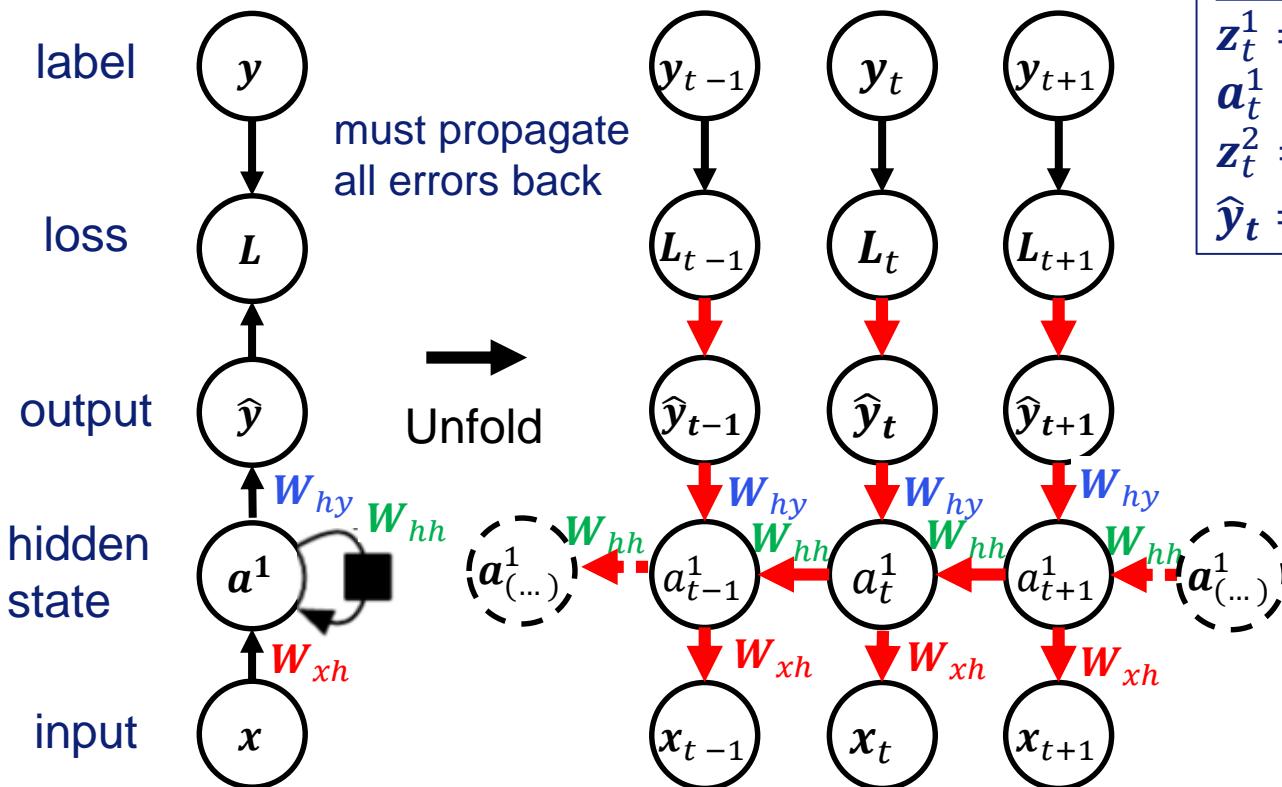
e.g. log-likelihood loss
(for softmax output):

$$L(\theta, y_t, \hat{y}_t) = -\ln \hat{y}_{y_t}$$

↑
Prob. (softmax output)
for correct class y_t at time t

- Modify any parameter → influences hidden / output / loss at all time steps

Training RNN: Backpropagation through time



Math. formulae:

$$\begin{aligned} \mathbf{z}_t^1 &= \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1}^1 + \mathbf{b}^1 \\ \mathbf{a}_t^1 &= \tanh(\mathbf{z}_t^1) \\ \mathbf{z}_t^2 &= \mathbf{W}_{hy} \mathbf{a}_t^1 + \mathbf{b}^2 \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{z}_t^2) \end{aligned}$$

total loss: $L = \sum_{t=1}^T L(\theta, y_t, \hat{y}_t)$

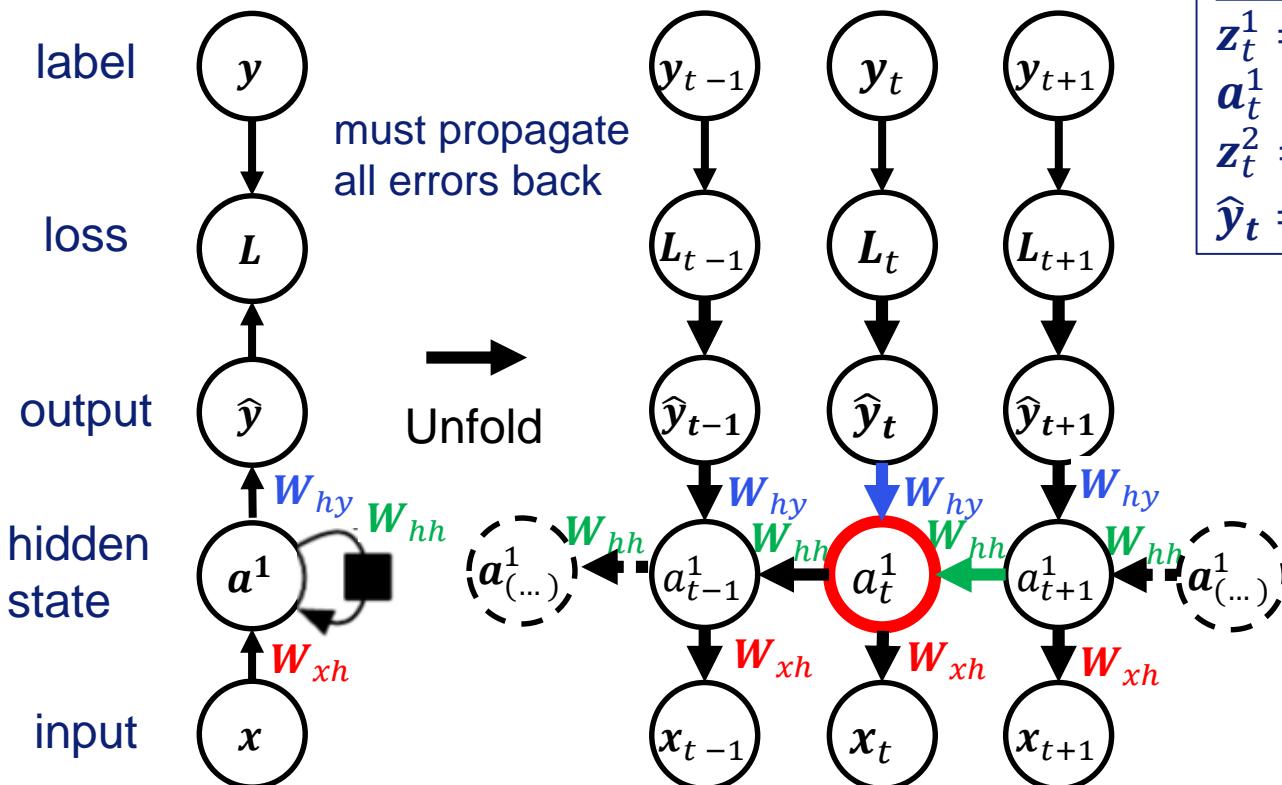
e.g. log-likelihood loss
(for softmax output):

$$L(\theta, y_t, \hat{y}_t) = -\ln \hat{y}_{y_t}$$

↑
Prob. (softmax output)
for correct class y_t at time t

- Modify any parameter → influences hidden / output / loss at all time steps
- These changes have to be traced back in backpropagation
 - In an **ordered** way (not parallel!)

Training RNN: Backpropagation through time



Math. formulae:

$$\mathbf{z}_t^1 = \mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1}^1 + \mathbf{b}^1$$

$$\mathbf{a}_t^1 = \tanh(\mathbf{z}_t^1)$$

$$\mathbf{z}_t^2 = \mathbf{W}_{hy} \mathbf{a}_t^1 + \mathbf{b}^2$$

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{z}_t^2)$$

total loss: $L = \sum_{t=1}^T L(\theta, y_t, \hat{y}_t)$

e.g. log-likelihood loss
(for softmax output):

$$L(\theta, y_t, \hat{y}_t) = -\ln \hat{y}_{y_t}$$

↑
Prob. (softmax output)
for correct class y_t at time t

Example: Gradient at a_t^1 (for $t < T$):

$$\nabla_{a_t^1} L = (\nabla_{a_{t+1}^1} L) \frac{\partial a_{t+1}^1}{\partial a_t^1} + (\nabla_{\hat{y}_t} L) \frac{\partial \hat{y}_t}{\partial a_t^1}$$

← contribution from output at t , involves W_{hy}

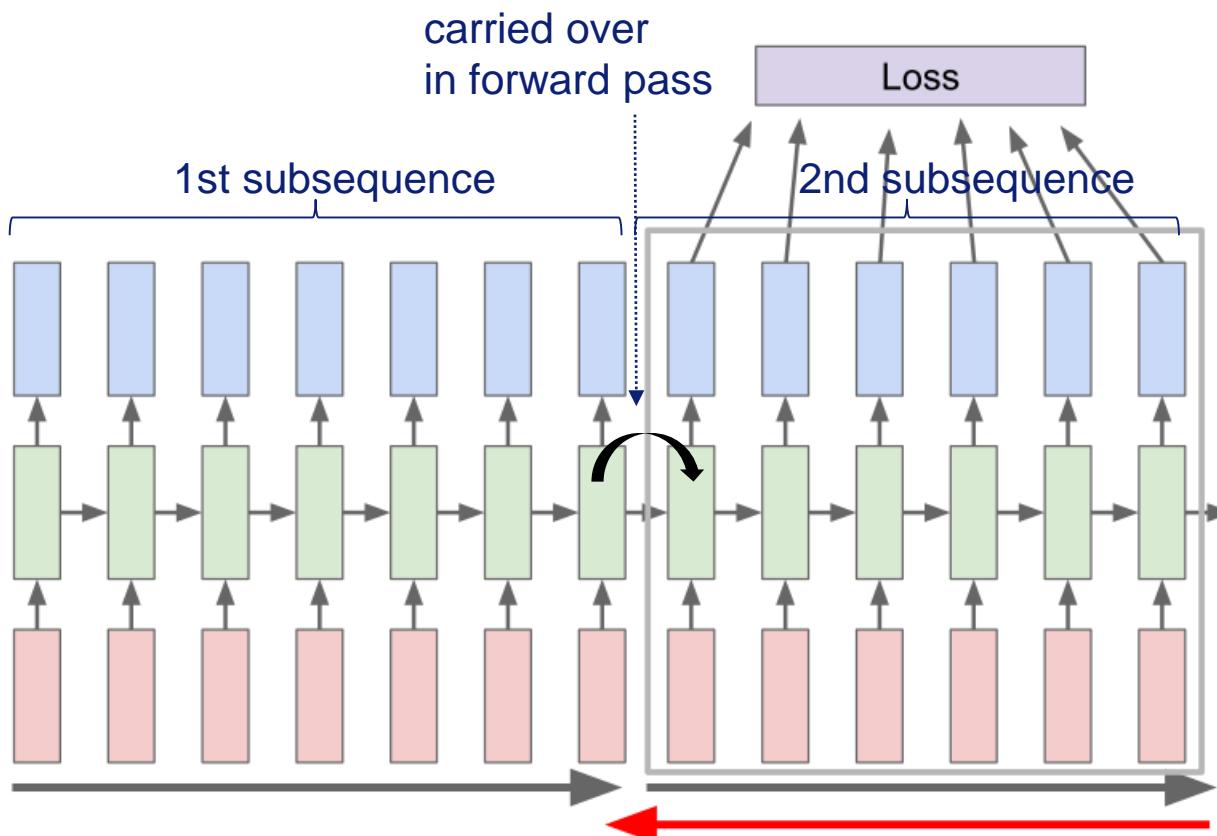
contribution from recurrent connection $t+1 \rightarrow t$, involves W_{hh}

Training RNN: Backpropagation through time

- Same parameters (weights, biases) used at all time steps
 - each time step contributes to updating the parameters
 - For a single weight update, $O(T)$ derivatives must be calculated (ordered!)
 - Complexity of BPTT:
 - Time: $O(T)$, i.e. slow learning
 - Memory: $O(T)$, i.e. large memory demand→ BPTT is expensive to train!
 - Example: Sequence of length 1000 → calculations are the equivalent of forward and backward pass in a network of 1000 layers (!)
 - Naive solution approach:
 - split 1000-long sequence into e.g. 50 sequences of length 20 each
 - treat these 50 sequence as separate training cases
 - But: Method is blind to temporal dependencies of more than 20 time steps
- Truncated backpropagation through time (TBPTT)

Truncated Backpropagation Through Time (TBPTT)

- Idea: Process subsequences, i.e. **backpropagate smaller number of steps**
- but at each following subsequence: **carry hidden states forward in time**
 - Same per-iteration cost as naive approach (less memory & time than BPTT)
 - But more adept at utilizing temporal dependencies (carry over in forward pass)



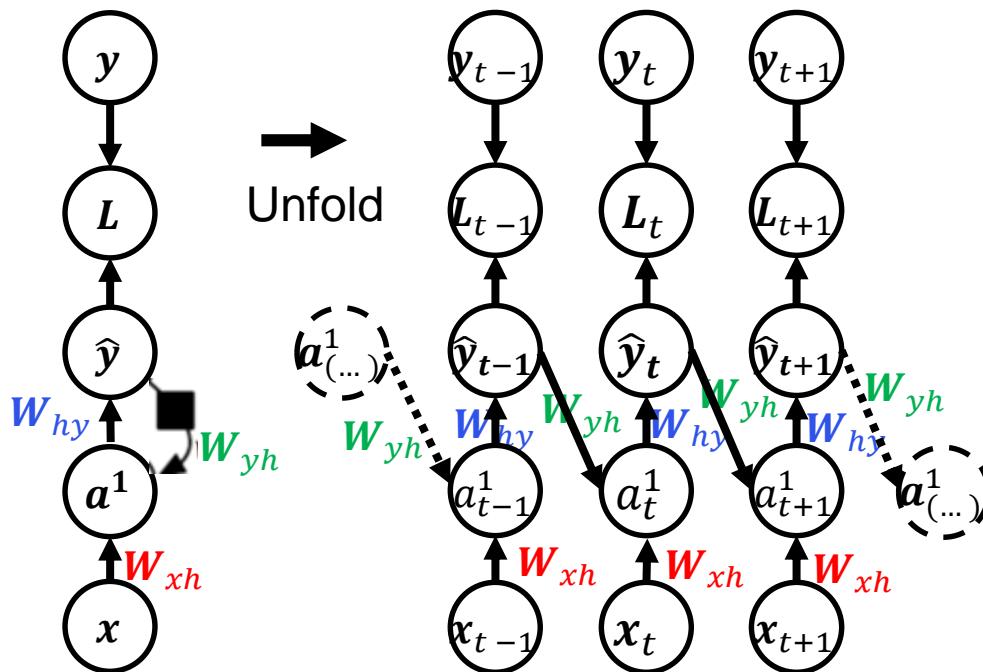
TBPTT: parameters k_1, k_2

```

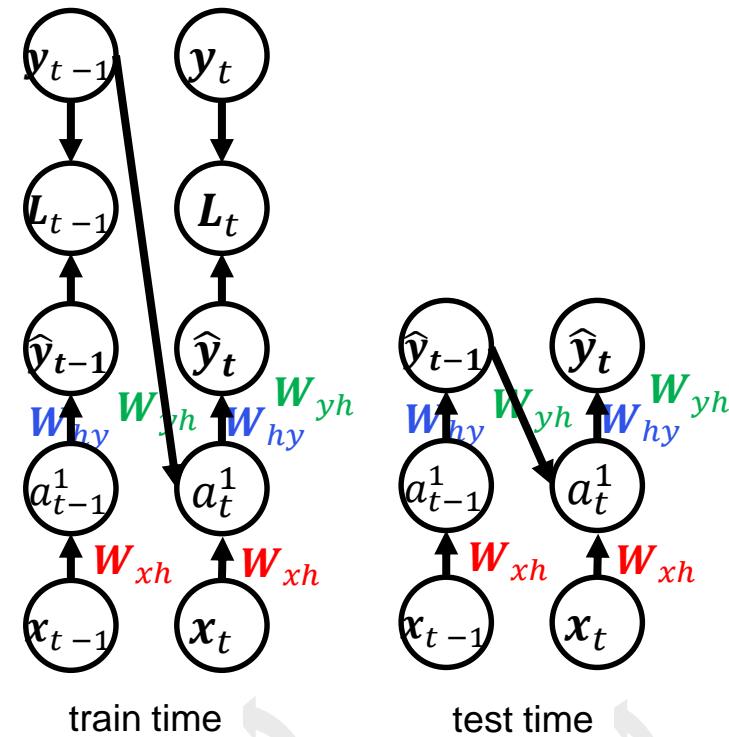
For  $t$  from 1 to  $T$  do
  Run RNN for time step  $t$ ,
  computing  $a_t$  and  $\hat{y}_t$ 
  If  $t$  divides  $k_1$  then
    Run BPTT from  $t$ 
    down to  $t - k_2$ 
  end if
end for
(often:  $k_1 = k_2$ )

```

RNN variant: Feedback output \rightarrow hidden



Teacher forcing:



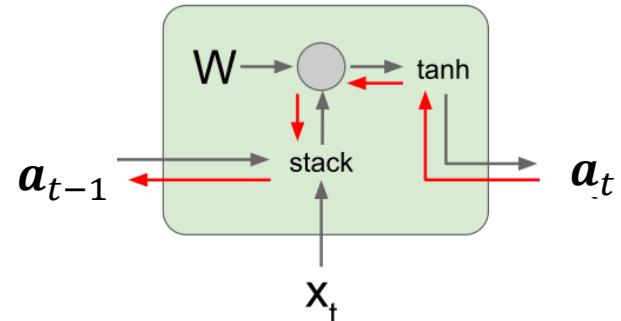
- **Less powerful than previous architecture**
 - Output unit must capture all information about past needed to predict future
- **But easier to train:**
 - **Teacher forcing:** use true output y_t (instead of predicted output) as input at $t + 1$
 \rightarrow each time step can be computed in isolation!
 - at test time: „normal“ architecture, i.e. model’s output fed back at $t + 1$

The challenge of long-term dependencies (1)

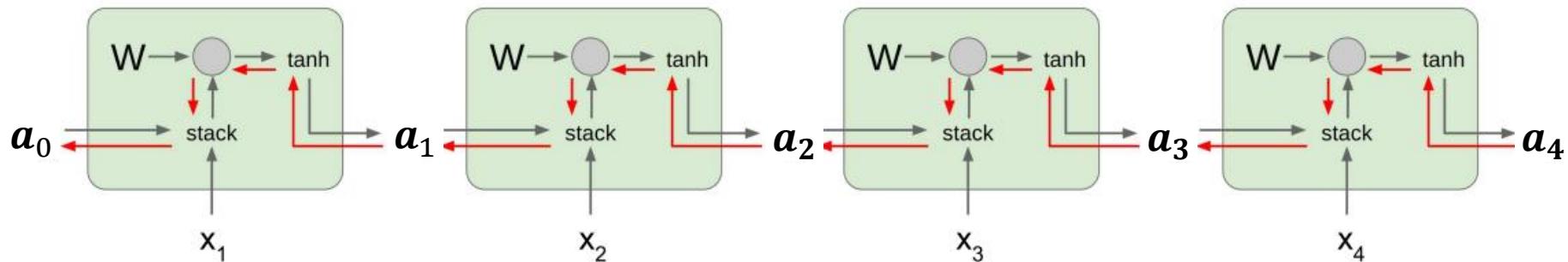
- Forward propagation:

$$\begin{aligned}
 a_t &= \tanh(\mathbf{W}_{hh}a_{t-1} + \mathbf{W}_{xh}x_t) \\
 &= \tanh\left(\begin{pmatrix} \mathbf{W}_{hh} & \mathbf{W}_{xh} \end{pmatrix} \begin{pmatrix} a_{t-1} \\ x_t \end{pmatrix}\right) \\
 &= \tanh\left(\mathbf{W} \begin{pmatrix} a_{t-1} \\ x_t \end{pmatrix}\right) \quad (\text{bias included in } \mathbf{W})
 \end{aligned}$$

Backpropagation from a_t to a_{t-1} multiplies by W (actually W_{hh}^T)



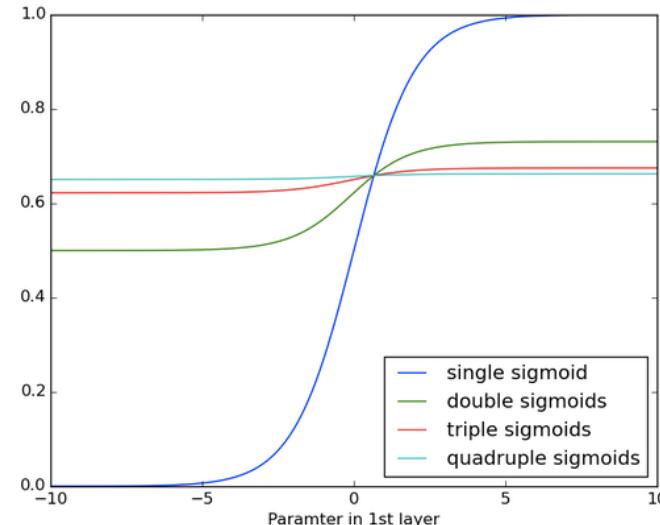
- Backpropagation involves derivative of tanh and multiplication with \mathbf{W}_{hh}^T
- Backprop. over many time steps: Many factors of \mathbf{W}_{hh}^T and repeated tanh



- If largest singular value of $\mathbf{W}_{hh}^T > 1$: exploding gradient
- if largest singular value of $\mathbf{W}_{hh}^T < 1$: vanishing gradient

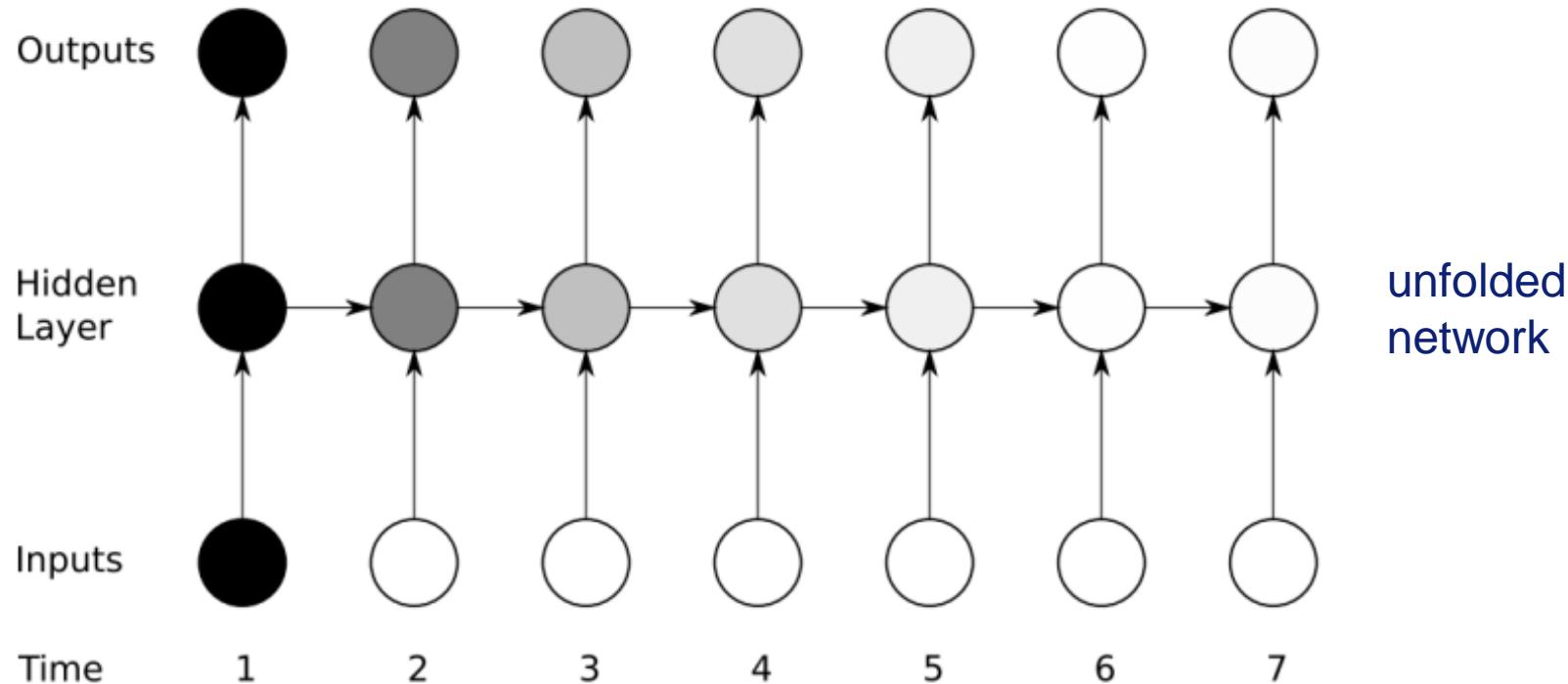
The challenge of long-term dependencies (2)

- Repeated application of sigmoid:



- Gradients propagated over many stages tend to either vanish or explode
 - Stages: time steps / layer in computational graph
 - if vanishing → difficult to know the gradient direction improving cost function
 - if exploding → can make learning unstable (→ might apply gradient clipping)
- Gradient of long-term interaction has exponentially smaller magnitude than gradient of short-term interaction
 - Difficult to train long-term dependencies („long“ means length 10 or 20)
- different RNN architectures (e.g. LSTM)

Vanishing gradient problem for RNNs: Illustration



From: Alex Graves, „Supervised sequence labeling“, Springer 2012

- Shading of nodes indicates sensitivity to inputs at time 1
 - Darker shade → greater sensitivity
- **Sensitivity decays over time** as new inputs overwrite the activations of hidden layer, and the network „forgets“ the first inputs

Long short term memory (LSTM)

Long short term memory (1)

- Motivation: Alleviate the vanishing gradient problem of RNNs

Standard RNN (single cell):

$$\mathbf{a}_t = \tanh(\mathbf{W}_{xh} \mathbf{x}_t + \mathbf{W}_{hh} \mathbf{a}_{t-1} + \mathbf{b})$$

Note: Layer superscript omitted for simplicity

Vanishing / exploding gradient caused
by (repeated) multiplication with \mathbf{W}_{hh}
when unfolding in time



Long short term memory (2) – „basic version“

- Motivation: Alleviate the vanishing gradient problem of RNNs

→ Introduce second hidden state („cell state“ c_t) which

- accumulates the input activations
- is LSTM-internal (not exposed to outside world)

$$g_t = \tanh(\mathbf{W}_{xg}x_t + \mathbf{W}_{hg}a_{t-1} + \mathbf{b}_g)$$

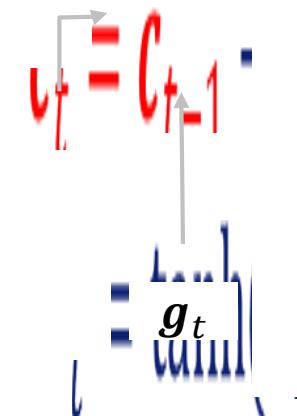
$$c_t = c_{t-1} + g_t$$

$$a_t = \tanh(c_t)$$



Note:

- no matrix multiplication of $c_t \Rightarrow \frac{\partial c_t}{\partial c_{t-1}} = 1$
- no vanishing gradient, but constant error flow
„constant error carousel“



compare with standard RNN activation

$$a_t = \tanh(\mathbf{W}_{xh}x_t + \mathbf{W}_{hh}a_{t-1} + \mathbf{b}) \rightarrow \frac{\partial a_t}{\partial a_{t-1}} \text{ may vanish / explode}$$

a_{t-1}

Long short term memory (3) – original version (Hochreiter, Schmidhuber 1997)

- Motivation: Control what to write to and read from cell

→ Introduce **gates** to control what to

- write to cell state: input gate i_t
 - read from cell state: output gate o_t

$$\boldsymbol{i}_t = \sigma(\mathbf{W}_{xi}\boldsymbol{x}_t + \mathbf{W}_{hi}\boldsymbol{a}_{t-1} + \boldsymbol{b}_i)$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo} \mathbf{x}_t + \mathbf{W}_{ho} \mathbf{a}_{t-1} + \mathbf{b}_o)$$

$$\mathbf{g}_t = \tanh(\mathbf{W}_{xg} \mathbf{x}_t + \mathbf{W}_{hg} \mathbf{a}_{t-1} + \mathbf{b}_g)$$

$$c_t = c_{t-1} + i_t \odot g_t$$

$$a_t = o_t \odot \tanh(c_t)$$

⊕ Hadamard product

(elementwise multiplication)

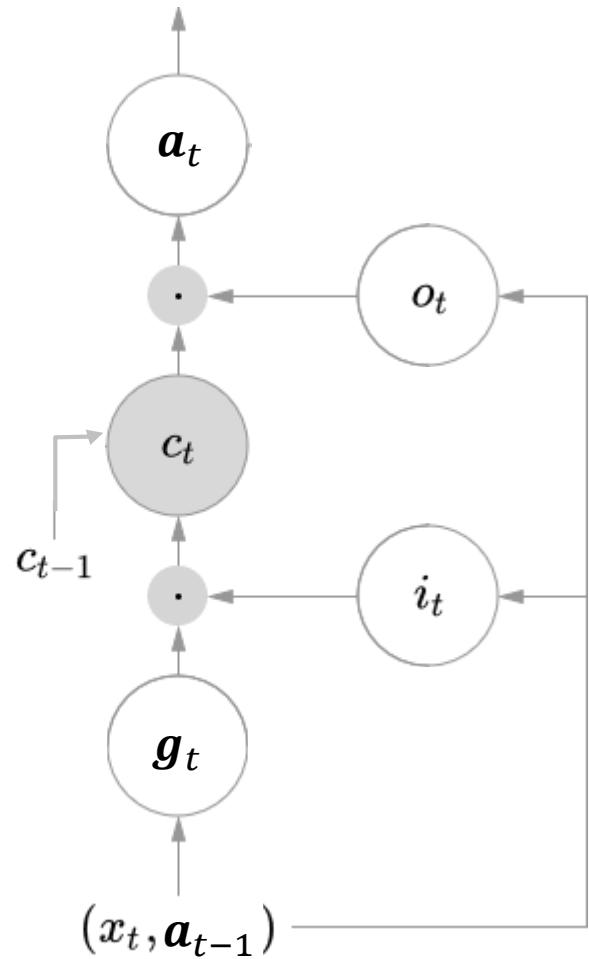
σ : sigmoid function

dark blue: no change to before

$i_t, o_t \in [0,1]$ (therefore sigmoid σ)

$i_t, o_t = 1$: gate fully open

$i_t, o_t = 0$: gate fully closed

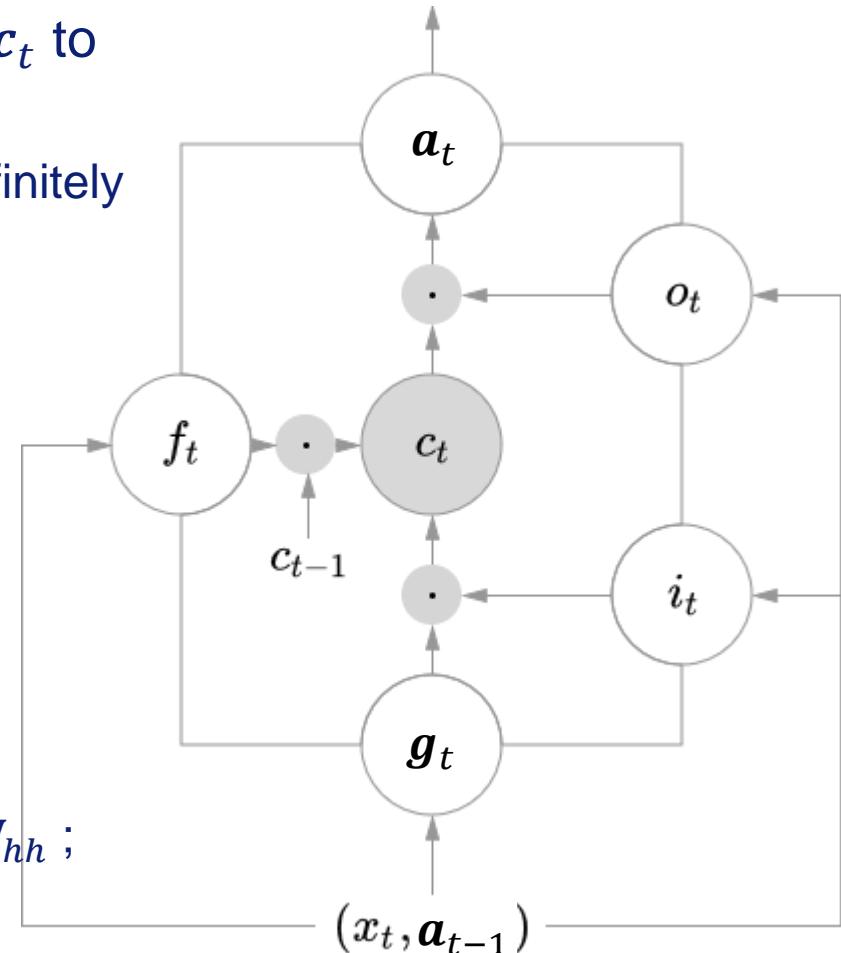


Long short term memory (4) – full version (Gers, Schmidhuber, Cummins 1999)

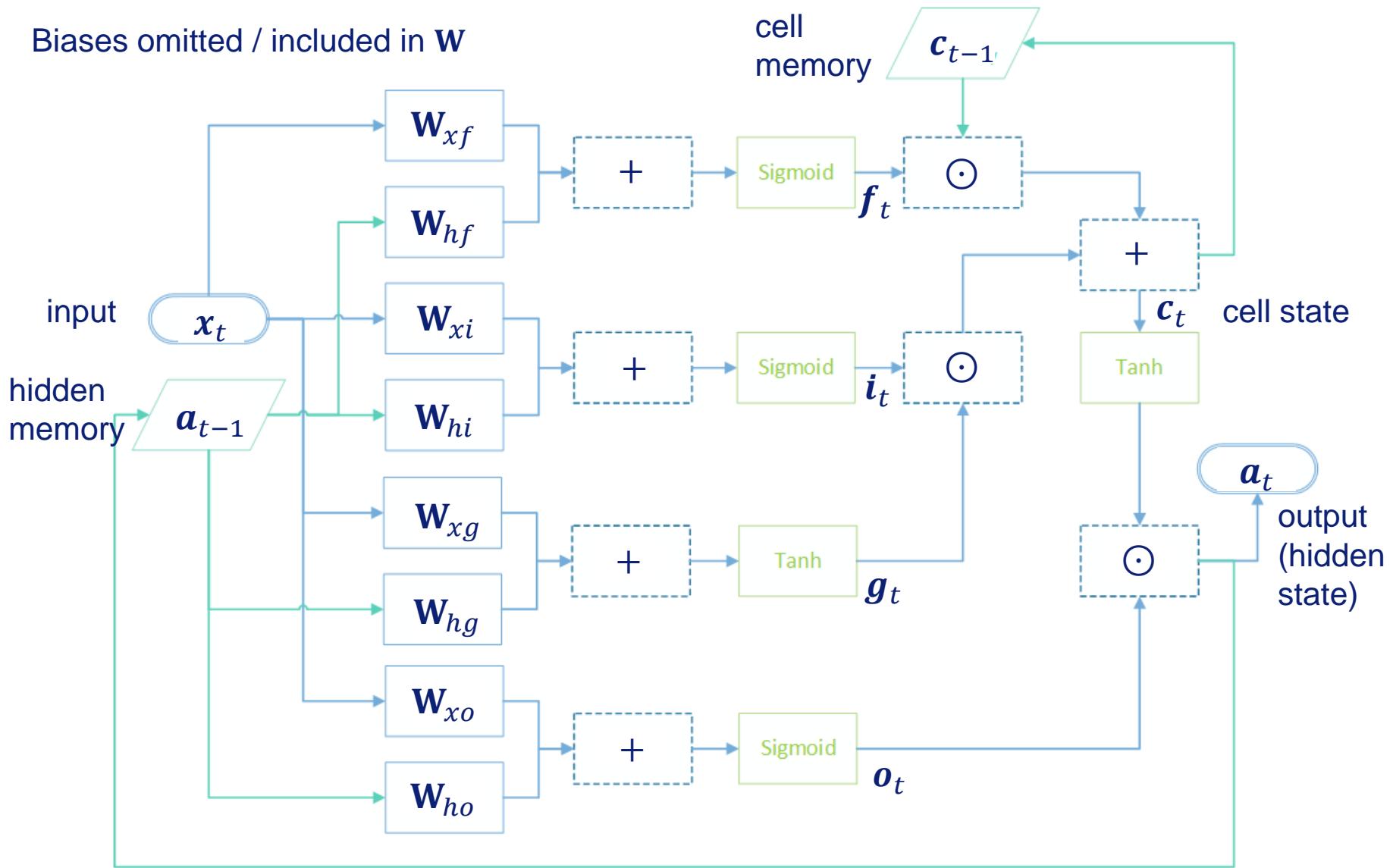
- Motivation: Allow cell state to reset itself (to segment input into subsequences)
 - Introduce **forget gate** f_t for cell c_t to control what to remember / erase
 - Otherwise, cell may grow indefinitely

$$\begin{aligned}
 i_t &= \sigma(\mathbf{W}_{xi}x_t + \mathbf{W}_{hi}a_{t-1} + \mathbf{b}_i) \\
 f_t &= \sigma(\mathbf{W}_{xf}x_t + \mathbf{W}_{hf}a_{t-1} + \mathbf{b}_f) \\
 o_t &= \sigma(\mathbf{W}_{xo}x_t + \mathbf{W}_{ho}a_{t-1} + \mathbf{b}_o) \\
 g_t &= \tanh(\mathbf{W}_{xg}x_t + \mathbf{W}_{hg}a_{t-1} + \mathbf{b}_g) \\
 c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
 a_t &= o_t \odot \tanh(c_t)
 \end{aligned}$$

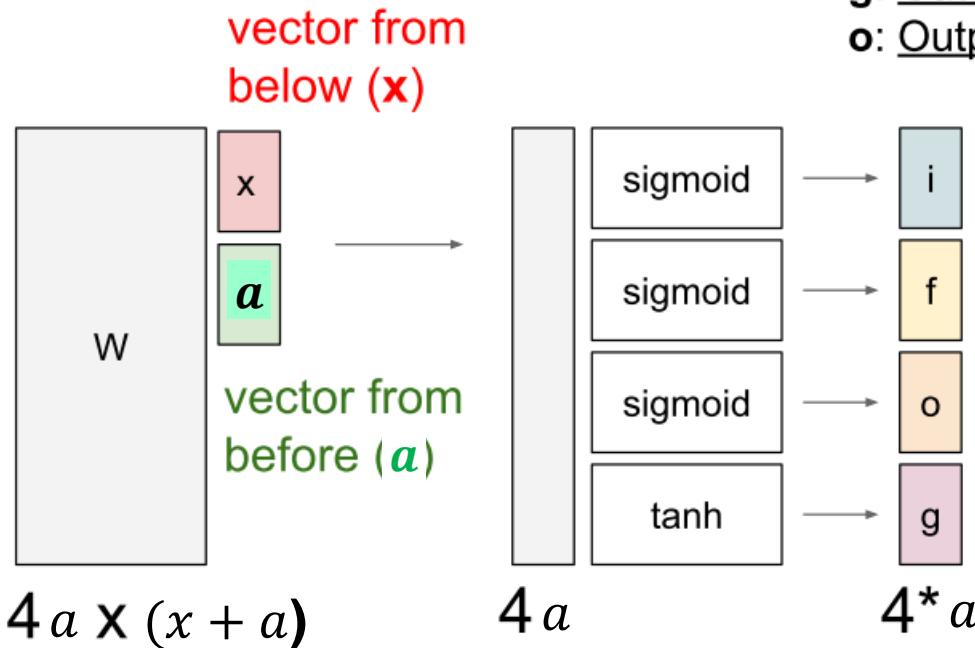
$f_t \odot c_{t-1}$: elementwise multiplication, no (repeated) matrix multiplication with \mathbf{W}_{hh} ; potentially different factor f_t at each t



Long short term memory: Diagram of computations (single LSTM unit)

Biases omitted / included in \mathbf{W} 

Long short term memory: Alternative representation



$a = \text{len}(a)$, i.e. number of hidden units
 $x = \text{len}(x)$, i.e. length of input

- f: Forget gate, Whether to erase cell
- i: Input gate, whether to write to cell
- g: Gate gate (?), How much to write to cell
- o: Output gate, How much to reveal cell

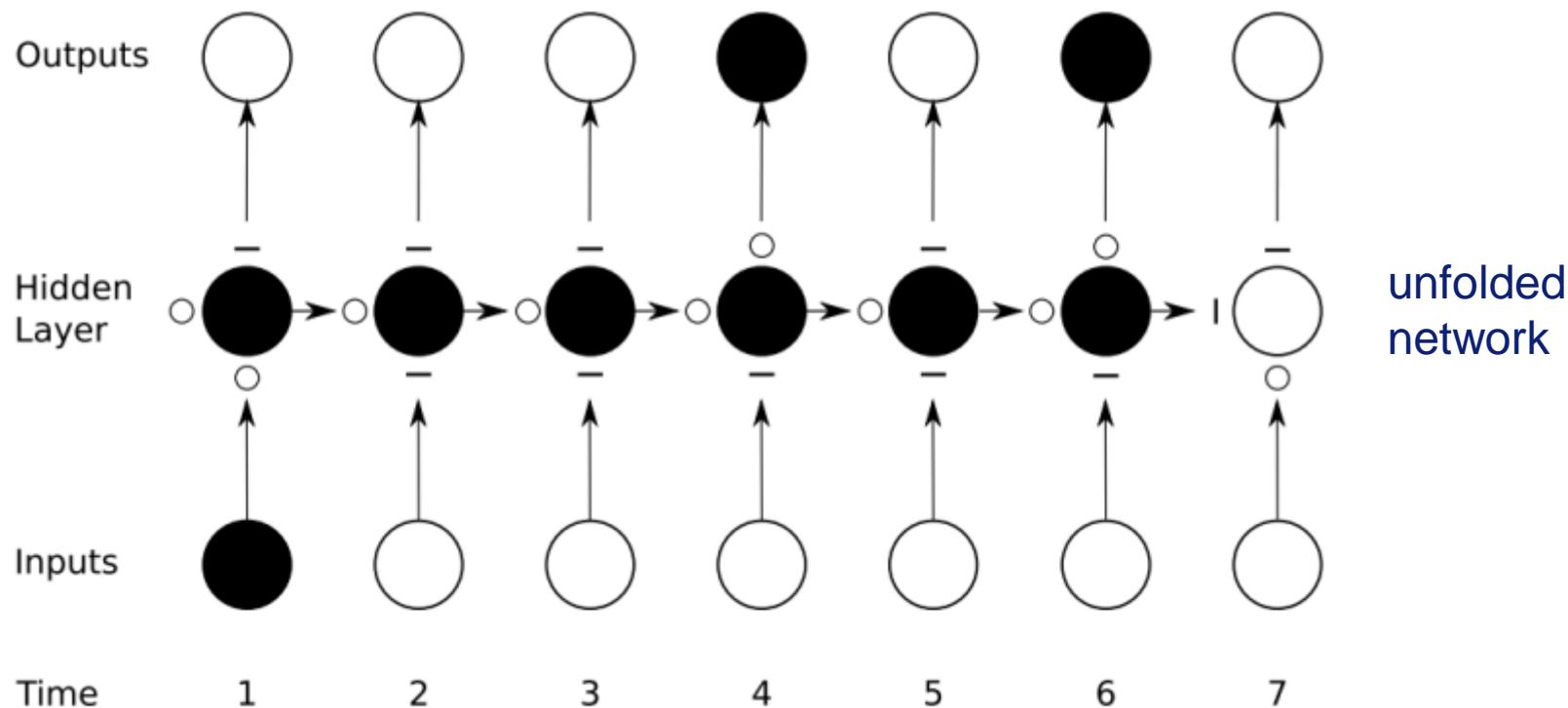
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} a_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$a_t = o \odot \tanh(c_t)$$

- Standard RNN (bias b included in W): $a_t = \tanh \left(W \begin{pmatrix} a_{t-1} \\ x_t \end{pmatrix} \right)$

Preservation of gradient information by LSTM: Illustration



- Shading of nodes indicates sensitivity to inputs at time 1
 - Black nodes: Maximally sensitive, white nodes: insensitive (binary for simplicity)
- Input / forget / output gates shown below / left / above hidden layer, respect.
 - o: gate entirely open, -: gate closed (binary for simplicity)
- **Cell „remembers“ first inputs as long as forget gate is open & input closed**
- Sensitivity of output layer can be switched on / off by output gate (doesn't affect cell)

Long short term memory: Discussion (1)

Motivation: Alleviate the vanishing gradient problem of RNNs

- $\frac{\partial c_t}{\partial c_{t-1}} = f_t \Rightarrow$ no gradient decay as long as f_t is close to 1

Example: Input x_1 at $t = 1$, but no further input for $t > 1$; let $f_t = 1 \forall t$

\Rightarrow circulating memory that never decays: $c_t = c_{t-1}$,
(the cell remembers its input over some time interval \rightarrow „memory“)

\Rightarrow any error backpropagated into this loop would also never decay
(standard RNN: decay due to gradient of activation function / W_{hh})

\rightarrow Better gradient flow properties than in standard RNNs

- in analogy to fancy CNNs like ResNet

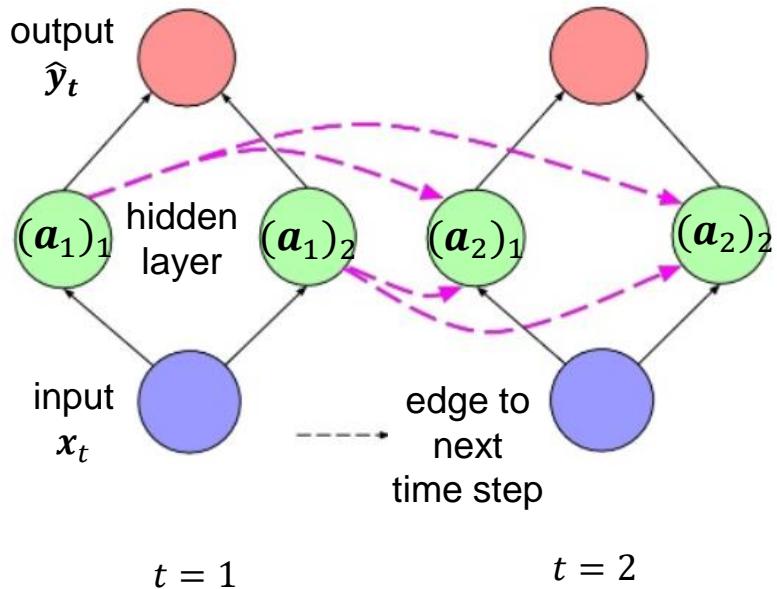
Long short term memory: Discussion (2)

- LSTM is well suited to **classify, process and predict time series**
 - where important events are separated by time lags of unknown size & duration
- Relative **insensitivity to gap length** gives LSTM an advantage over other sequence learning methods like Hidden Markov Models
- Further LSTM modifications exist, e.g.
 - Peephole LSTM (Gers and Schmidhuber, 2000)
 - Convolutional LSTM (Shi et al., 2015)
- **Hidden layers are composed of many LSTM units and several hidden layers can be stacked on top of each other**
 - But: Increased training time and memory demand, i.e. computational limits

Example: RNN with two hidden standard / LSTM units (unfolded in time)

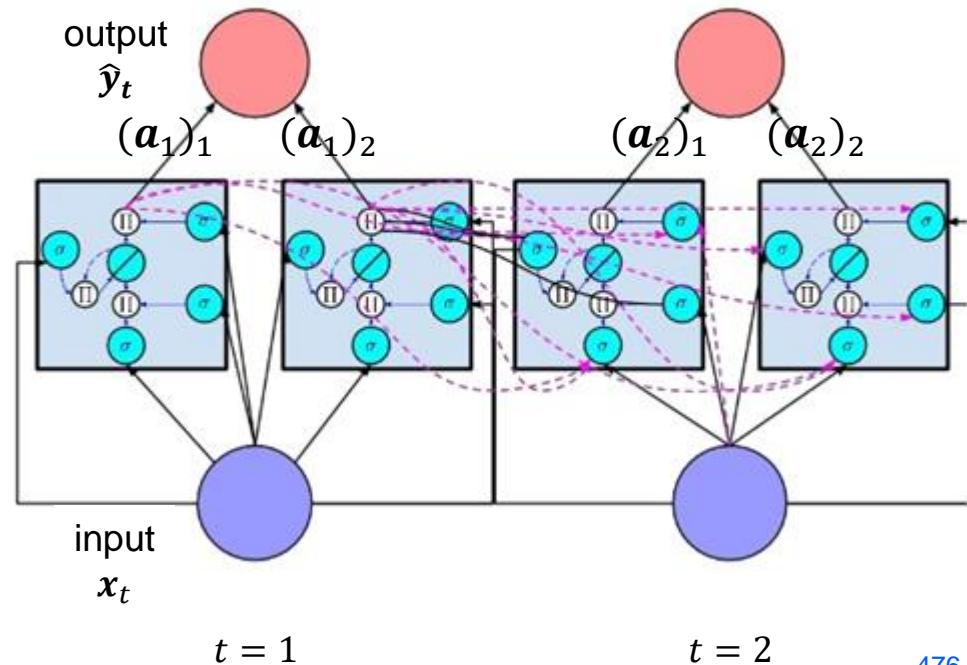
RNN with two hidden standard units:

$$\begin{aligned} \mathbf{a}_t &= \tanh(\mathbf{W}_{xh}x_t + \mathbf{W}_{hh}\mathbf{a}_{t-1} + \mathbf{b}^1) \\ \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{W}_{hy}\mathbf{a}_t + \mathbf{b}^2) \end{aligned}$$



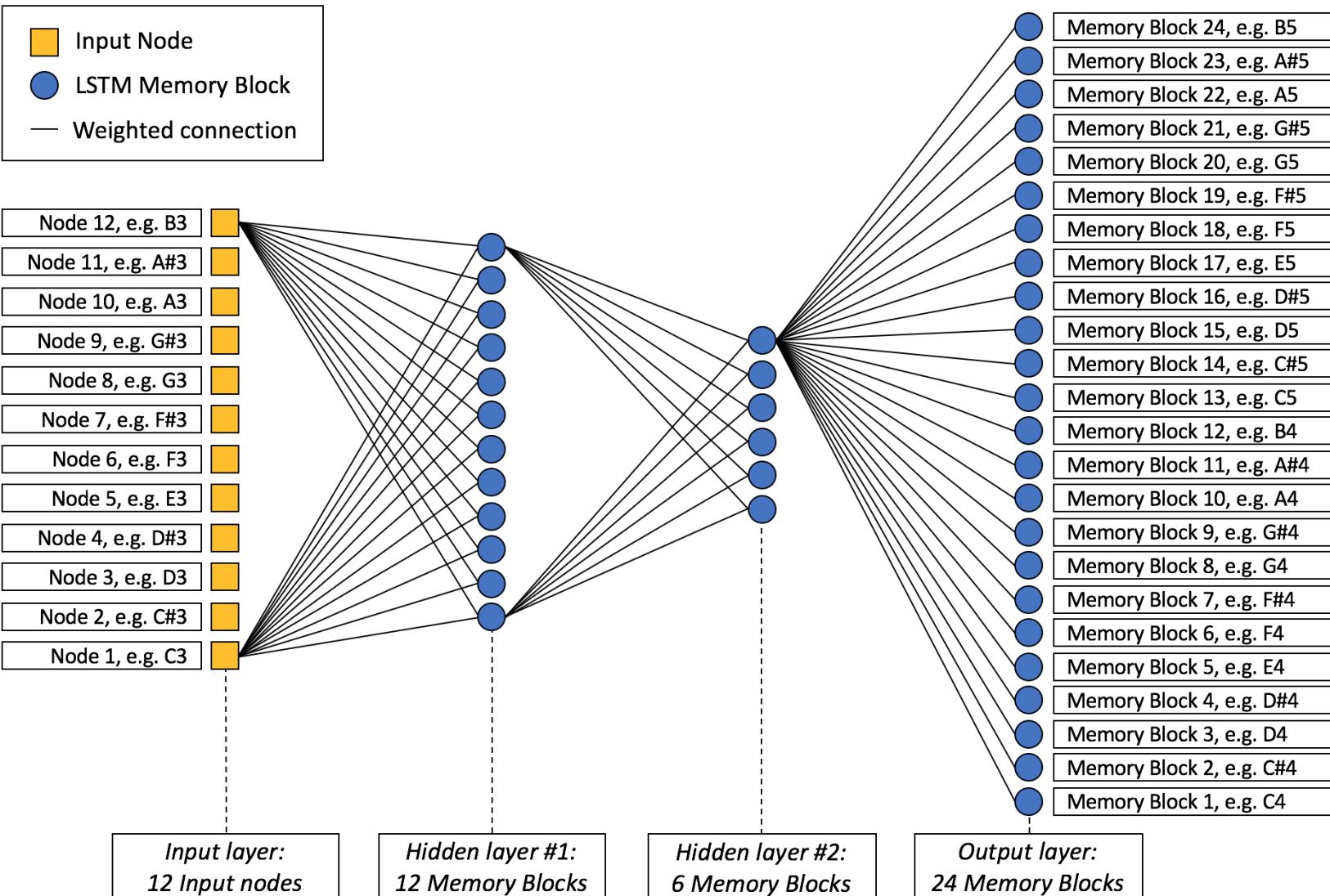
RNN with two hidden LSTM cells:

$$\begin{aligned}
 \mathbf{i}_t &= \sigma(\mathbf{W}_{xi} \mathbf{x}_t + \mathbf{W}_{hi} \mathbf{a}_{t-1} + \mathbf{b}_i) \\
 \mathbf{f}_t &= \sigma(\mathbf{W}_{xf} \mathbf{x}_t + \mathbf{W}_{hf} \mathbf{a}_{t-1} + \mathbf{b}_f) \\
 \mathbf{o}_t &= \sigma(\mathbf{W}_{xo} \mathbf{x}_t + \mathbf{W}_{ho} \mathbf{a}_{t-1} + \mathbf{b}_o) \\
 \mathbf{g}_t &= \tanh(\mathbf{W}_{xg} \mathbf{x}_t + \mathbf{W}_{hg} \mathbf{a}_{t-1} + \mathbf{b}_g) \\
 \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \\
 \mathbf{a}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \\
 \hat{\mathbf{y}}_t &= \text{softmax}(\mathbf{W}_{hy} \mathbf{a}_t + \mathbf{b}^2)
 \end{aligned}$$



Example LSTM application: Music composition

Goal: Compose a melody to a given chord sequence



Long short term memory: Application examples

- Handwriting recognition
- Speech recognition
 - E.g. Google's speech recognition and machine translation, Apple's Siri, Amazon Alexa
- Robot control
- Time series prediction, time series anomaly detection
- Human action recognition
- Semantic parsing
- ...

Recurrent neural networks: Summary

- Different architectures of recurrent neural networks:
 - With layers: Recurrent connection often within hidden layer (or from output to hidden layer); Example: Standard units, **LSTM units**
 - Application: Many-to-many, many-to-one or one-to-many sequence problems
 - Learning: (Truncated) backpropagation through time
 - Standard unit: Vanishing gradient problem for long-term dependencies
 - LSTM aims to overcome vanishing gradient problem by introducing cell state
 - Without layers (e.g. fully connected): Example **Hopfield network**
 - Hebbian learning
 - Associative memory

UNSUPERVISED LEARNING

Recall: Supervised versus unsupervised learning

Supervised learning

- Data: $D = \{(\mathbf{x}_i, y_i)\}$ (data + labels)
- Goal: Learn **function** to map $\mathbf{x} \rightarrow y$

- Disadvantage: Needs annotated data (big manual effort!)

- Examples:

- Classification
- Regression
- Object detection
- Semantic segmentation
- Image captioning

Unsupervised learning

- Data: $D = \{\mathbf{x}_i\}$ (just data, no labels)
- Goal: Learn some underlying hidden **structure** of the data

Solve unsupervised learning
→ **understand structure of visual world**

- Advantage: Un-annotated data are cheap and abundant

- Examples:

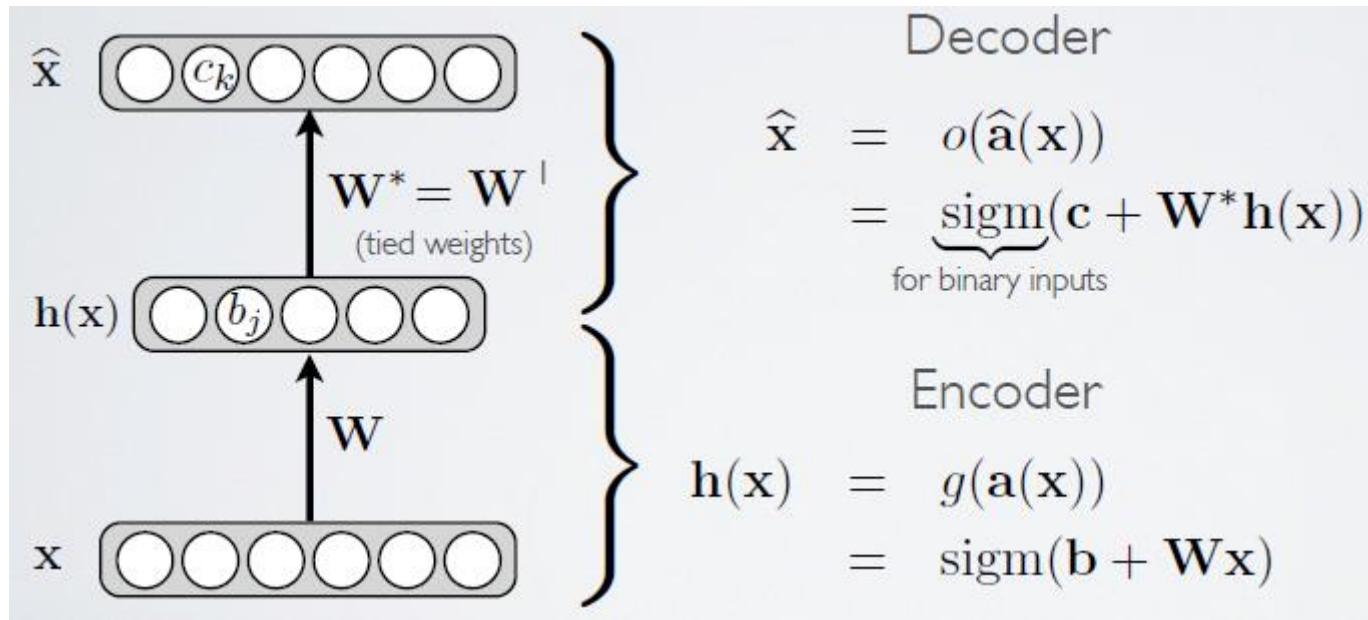
- Clustering
- Dimensionality reduction
- Feature learning (e.g. autoencoder)
- Density estimation (e.g. variational autoencoder, GANs)
- ...

Autoencoder

(slides based on lecture of Hugo Larochelle and others)

Autoencoder: Definition

- Feed-forward neural network trained to reproduce its input at output layer



- Goal: Learn a distributed representation (encoding) for a set of data
 - e.g. for the purpose of feature learning or dimensionality reduction
- Autoencoder: Unsupervised approach for learning a lower-dimensional feature representation from unlabeled training data

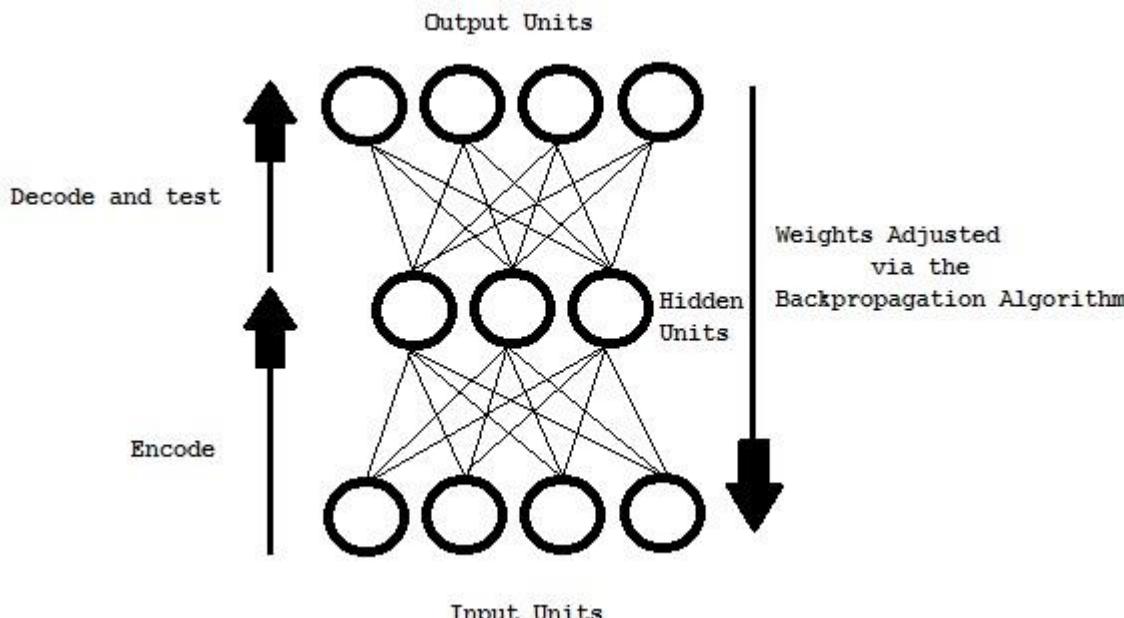
Training

- Define a loss function, e.g. for real-valued inputs: $l(f(x)) = \frac{1}{2} \sum_{\mu=1}^p (\hat{x}_\mu - x_\mu)^2$

- Sum of squared Euclidean distance between input and reconstruction
- Using a *linear* activation function at the output:

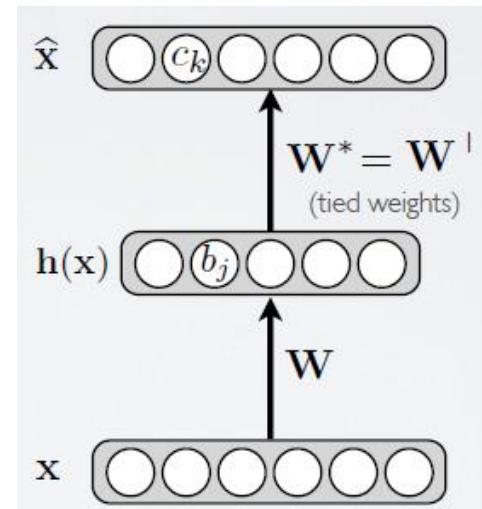
$$\hat{x} = f(x) = \mathbf{W}^T \cdot h(\mathbf{x}) + \mathbf{c} = \mathbf{W}^T (\mathbf{W} \cdot \mathbf{x} + \mathbf{b}) + \mathbf{c}$$

- Compute gradient with respect to parameters **W**, **b** and **c**
- Backpropagate gradient like in a regular network

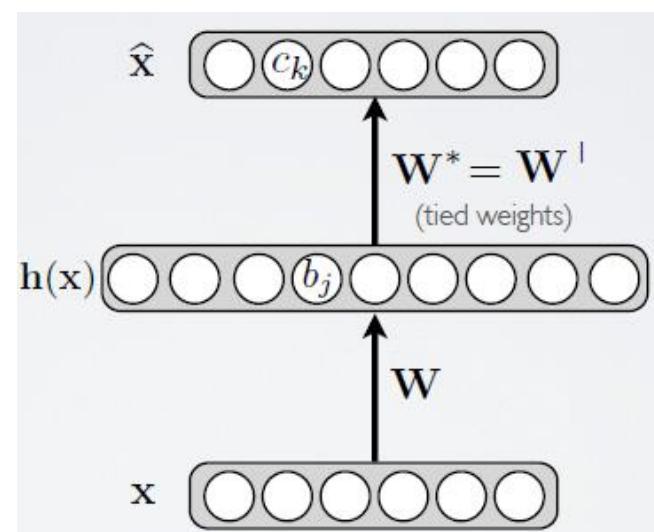


How many units in hidden layer?

- Less than in input layer („undercomplete“)
 - Hidden layer „compresses“ the input
 - Will compress well only for training distribution
 - Hidden units good for training distribution, but bad for other types of input

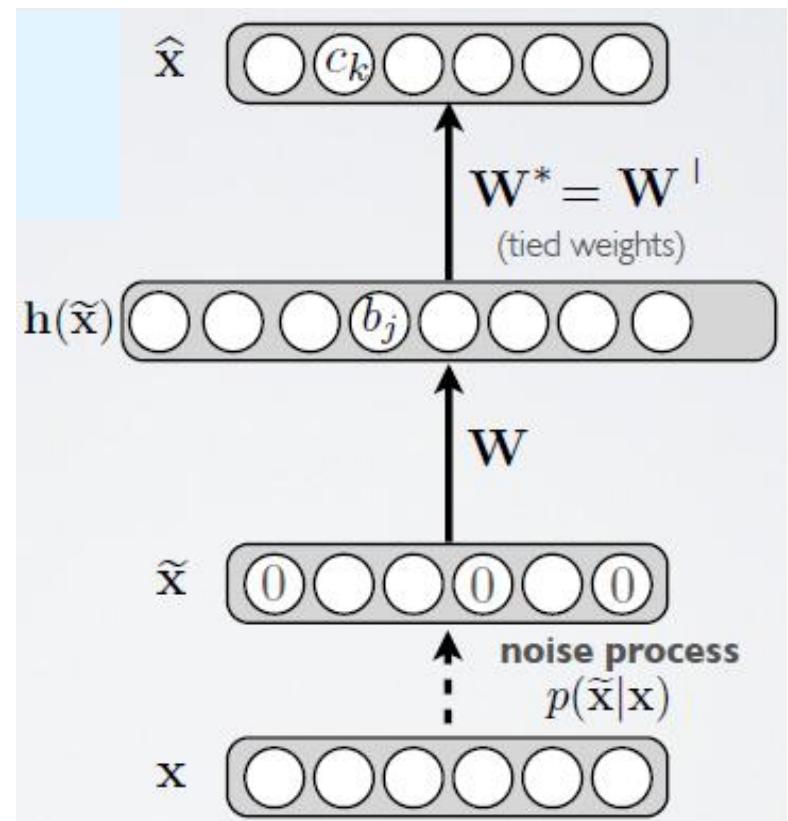


- More than in input layer („overcomplete“)
 - No compression in hidden layer
 - Each hidden unit could simply copy a different input component
 - No guarantee that hidden units will extract meaningful structure

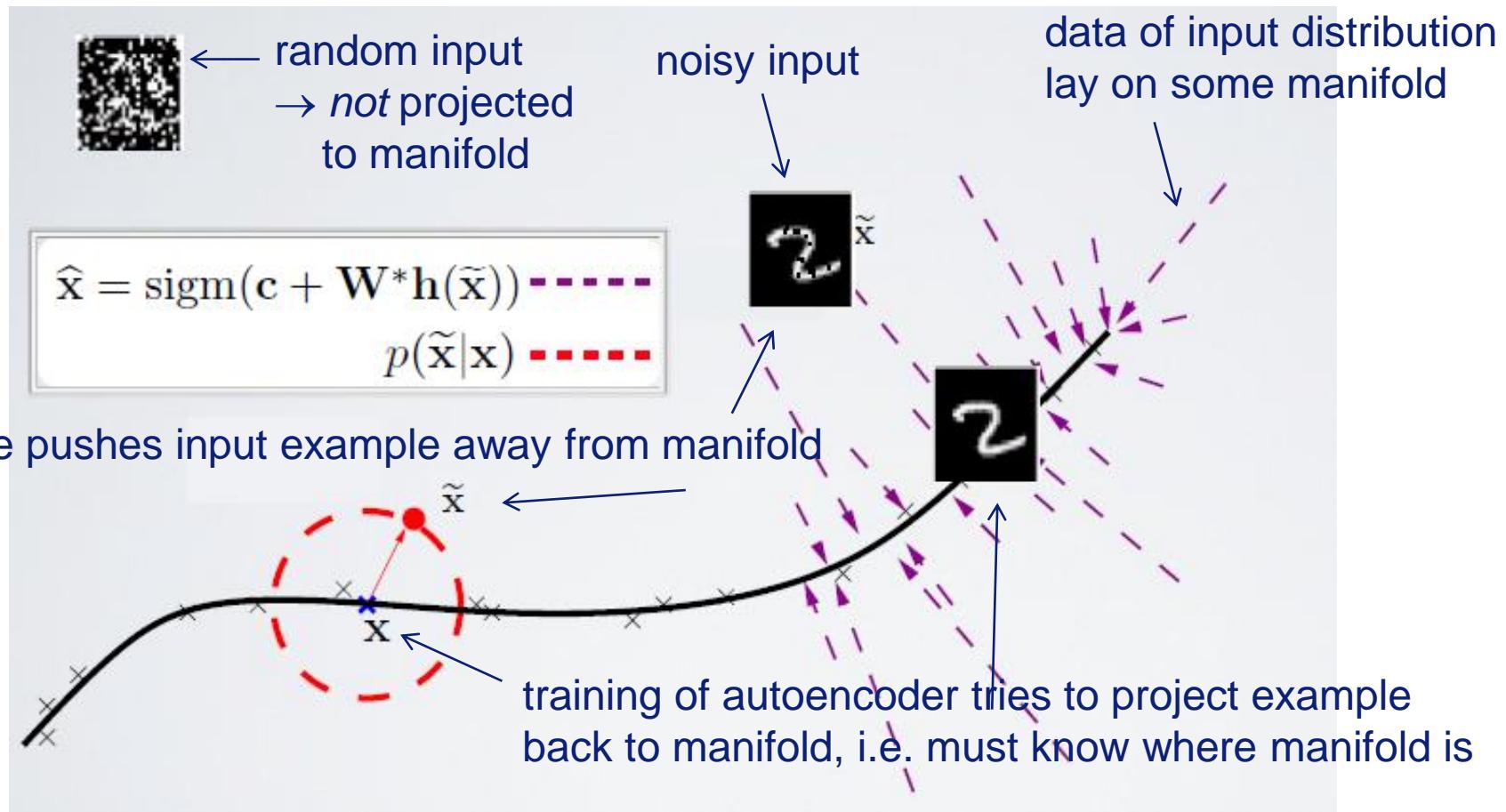


Denoising autoencoder

- Idea: More meaningful input structure is learned if hidden representation is forced to be robust to introduction of noise into the input
 - Randomly set subset of inputs to 0 (with probability ν)
 - Alternative: Add Gaussian noise
- Loss function compares $\hat{\mathbf{x}}$ reconstruction with noiseless input \mathbf{x}
- Reconstruction $\hat{\mathbf{x}}$ computed from the corrupted input $\tilde{\mathbf{x}}$
 - Input components cannot just be copied (because *noisy* inputs are seen, which may not correspond to *true* input)
 - Instead: Hidden units must learn about *relationship* between input components, i.e. learn structure of input distribution



Denoising autoencoder: Illustration (handwritten digit recognition)



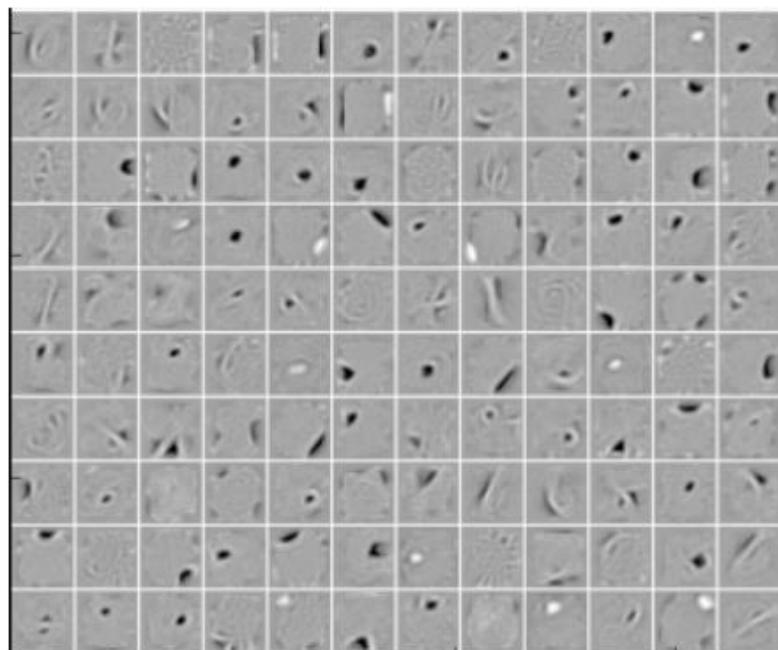
- Autoencoder must learn about **conditional dependencies / structure** within inputs (e.g. characters in contrast to non-character images / noise)

Denoising autoencoder: Example

- Visualisation of weight vectors between inputs and specific hidden unit
 - Every single square corresponds to one hidden unit in autoencoder
 - White: positive weight, gray: zero weight, black: negative weight

Images of handwritten digits (MNIST):

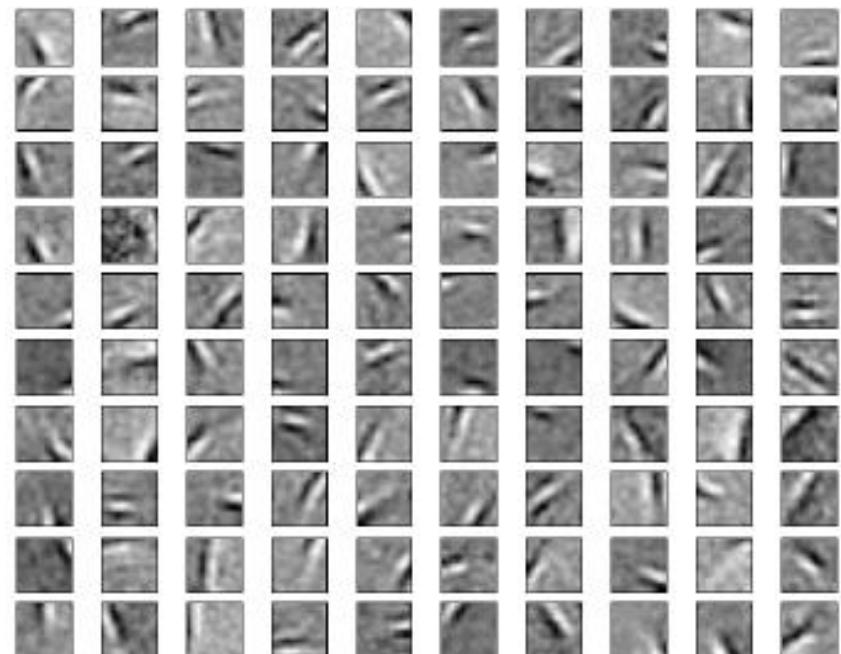
50% corrupted inputs



filters act like penstroke detectors

Natural image patches

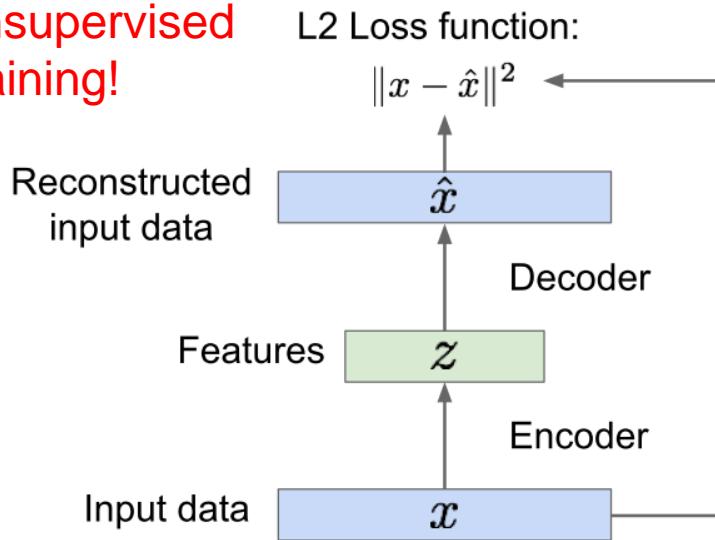
(add Gaussian noise, squared difference loss)



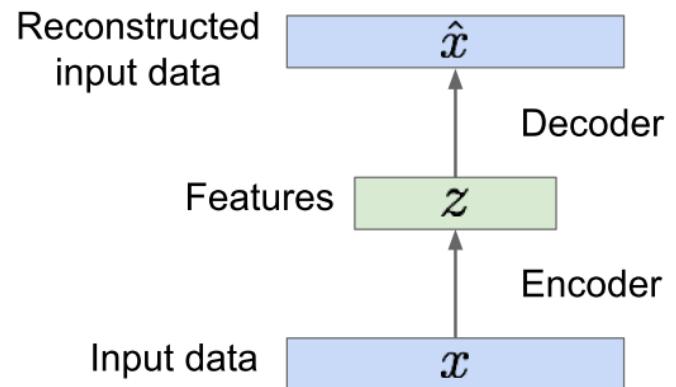
filters act like edge detectors

Autoencoder: Application in unsupervised pre-training

- Train autoencoder such that features can be used to reconstruct original data:
unsupervised training!



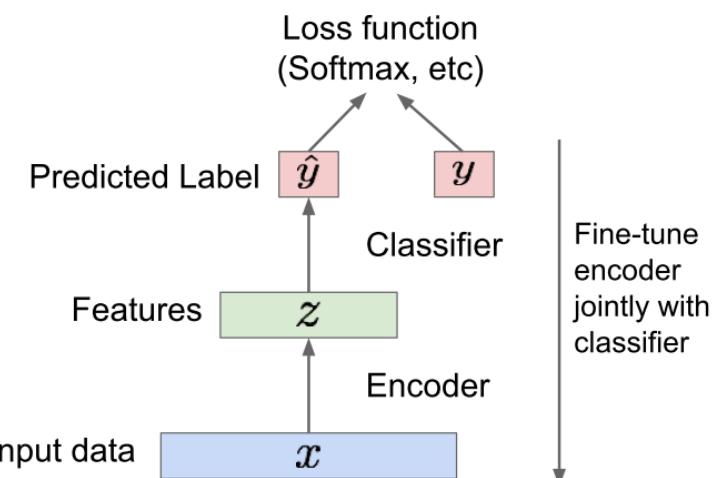
after training, throw away decoder



replace decoder e.g. by classifier, trained (jointly with features) with supervised training; („fine-tuning“)

Encoder / Decoder:

- Linear / nonlinear (sigmoid) layer
- Later: Deep, fully connected
- Later: ReLU CNN



Autoencoder: Summary

- Autoencoder are an example for **unsupervised learning** where some (feature) representation of an input image is learned from which the input can be reconstructed
 - Learned features can be used to initialize a supervised model
 - Learned features capture factors of variation in training data
- **Training** can be performed by optimizing a loss function with respect to the parameters (synaptic weights and biases)
 - Similar to backpropagation
- **Regularization** is necessary to learn conditional dependencies / structure in the input data
 - Denoising autoencoder
 - Alternatives: Sparse autoencoder etc.
- Learned representations may have an intuitive interpretation
 - At least in selected applications

Further readings

- Vincent et al.: „Extracting and composing robust features with denoising autoencoders“ (ICML 2008, <http://icml2008.cs.helsinki.fi/papers/592.pdf>)
- Andrew Ng, „Sparse autoencoder“ (CS294A Lecture notes, <http://web.stanford.edu/class/cs294a/sparseAutoencoder.pdf>)
- Bengio et al.: „Generalized denoising auto-encoders as generative models“ (NIPS 2013, <http://papers.nips.cc/paper/5023-generalized-denoising-auto-encoders-as-generative-models.pdf>)
- G. Hinton et al.: „Transforming autoencoders“ (ICANN 2011, <http://www.cs.toronto.edu/~fritz/absps/transauto6.pdf>)
- A very fast denoising autoencoder: <http://fastml.com/very-fast-denoising-autoencoder-with-a-robot-arm/>

Generative models

Discriminative versus generative models

- Data: Pixels (features \mathbf{x})
- Labels: y (zebra / no zebra)

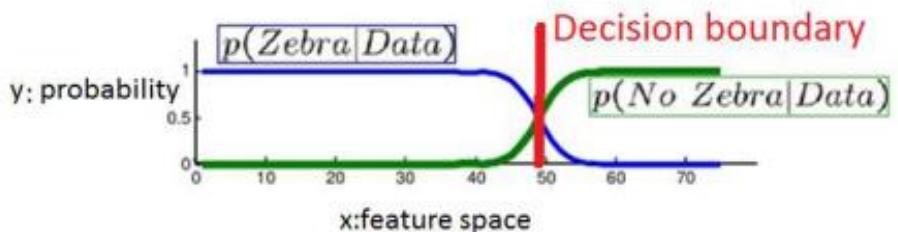


From: <https://www.pexels.com/photo/selective-focus-photography-of-zebra-750539/> (public domain)
 From https://cseweb.ucsd.edu/classes/sp17/cse252c-a/CSE252C_20170412.pdf

Discriminative model: Models $p(y|\mathbf{x})$

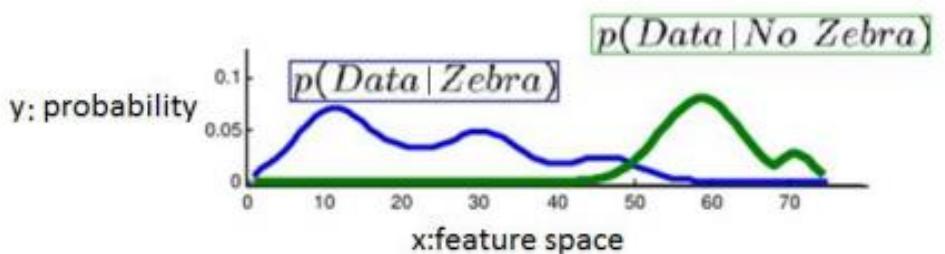
- Learns **decision boundary** in feature space
- For each point \mathbf{x} in feature space:

$$p(\text{zebra}|\mathbf{x}) + p(\text{no zebra}|\mathbf{x}) = 1$$



Generative model: Models $p(\mathbf{x}, y)$ ($p(y|\mathbf{x})$ may be obtained via Bayes' rule)

- Learns **probability distribution** of each class of data
- $\int p(\mathbf{x}|\text{zebra})dx = 1$
- $\int p(\mathbf{x}|\text{no zebra})dx = 1$



Generative models: Goal and motivation

- Idea: Given training data, generate new samples from same distribution



Training data $\sim p_{\text{data}}(\mathbf{x})$



Generated samples $\sim p_{\text{model}}(\mathbf{x})$

- Goal: Learn $p_{\text{model}}(\mathbf{x})$ similar to $p_{\text{data}}(\mathbf{x})$

Variational autoencoder

- density estimation (core problem in unsupervised learning)
- Explicit density estimation: explicitly define and solve for $p_{\text{model}}(\mathbf{x})$
- Implicit density estimation: learn model that can sample from $p_{\text{model}}(\mathbf{x})$ without explicitly defining it

GANs

- Motivation:

- Generate realistic samples for artwork super-resolution, colorization etc.
- Artificially expand training set (\rightarrow semi-supervised learning)
- Enable inference of latent representations which are useful as general features
- Time series: Generate data for simulation / planning (reinforcement learning!) ⁴⁹⁴

Generative models: Variational autoencoder (VAE)

Generating data: The problem with standard autoencoders

Autoencoders...

- calculate compact representations of their input
- Reconstruct their inputs (seen in training) well (i.e. replicate known images)

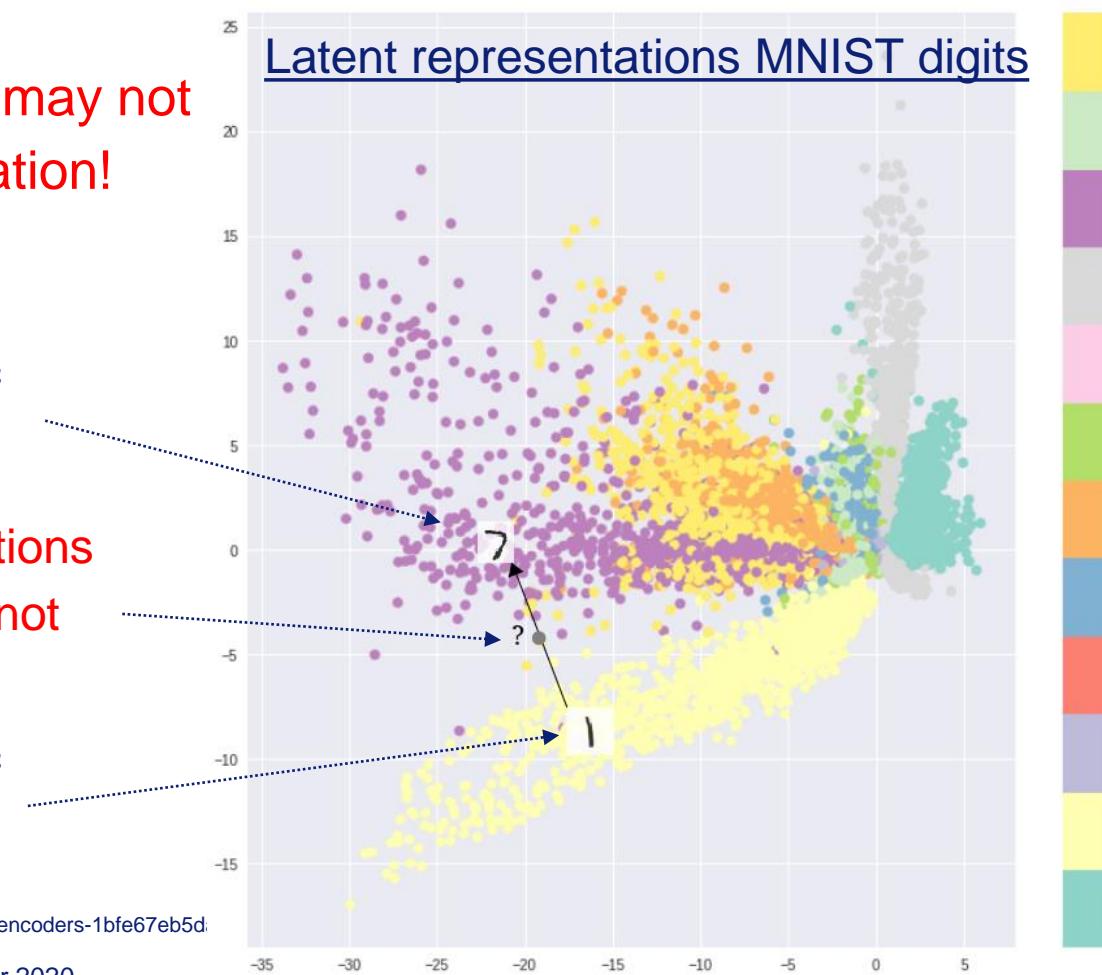
But the generated latent space may not be continuous or allow interpolation!

Example:

Cluster of latent representations of handwritten digit „7“

How to deal with latent representations „in between“? (Those points have not been observed during training!)

Cluster of latent representations of handwritten digit „1“

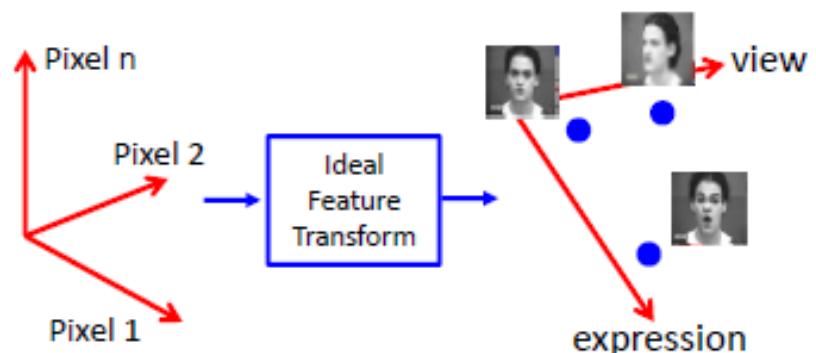
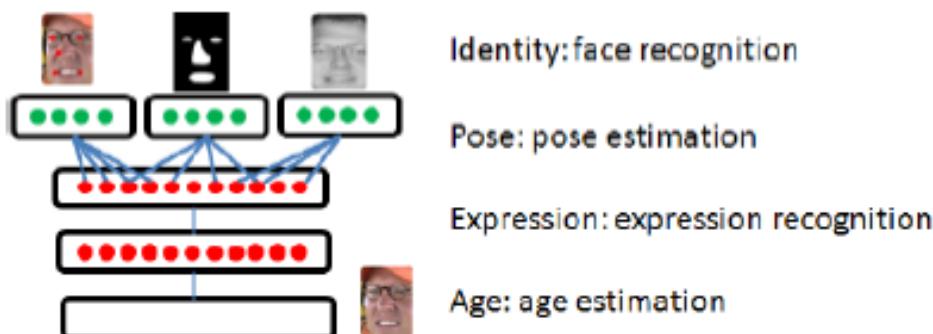
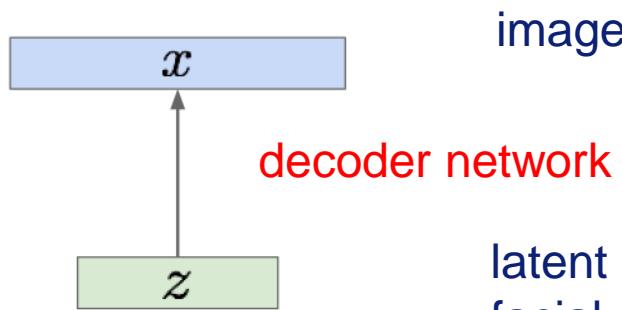


Variational autoencoder: Idea

- How can we sample from the model to **generate new representative data**?
 - Needs **continuous** latent space, e.g. characterised by probability distribution $p_{\theta^*}(z)$ the parameters of which have to be learned
 - Varying latent variable z then leads to **meaningful** variation in generated image

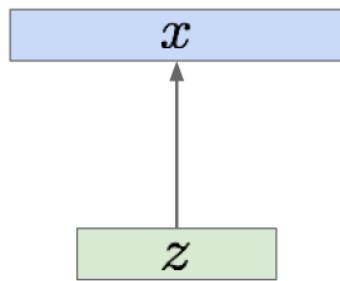
sample from
true conditional
 $p_{\theta^*}(x|z)$

sample from
true prior
 $p_{\theta^*}(z)$



Variational autoencoder: Problem

- Goal: Estimate the true parameters θ^* of the generative model
- How to represent the model?



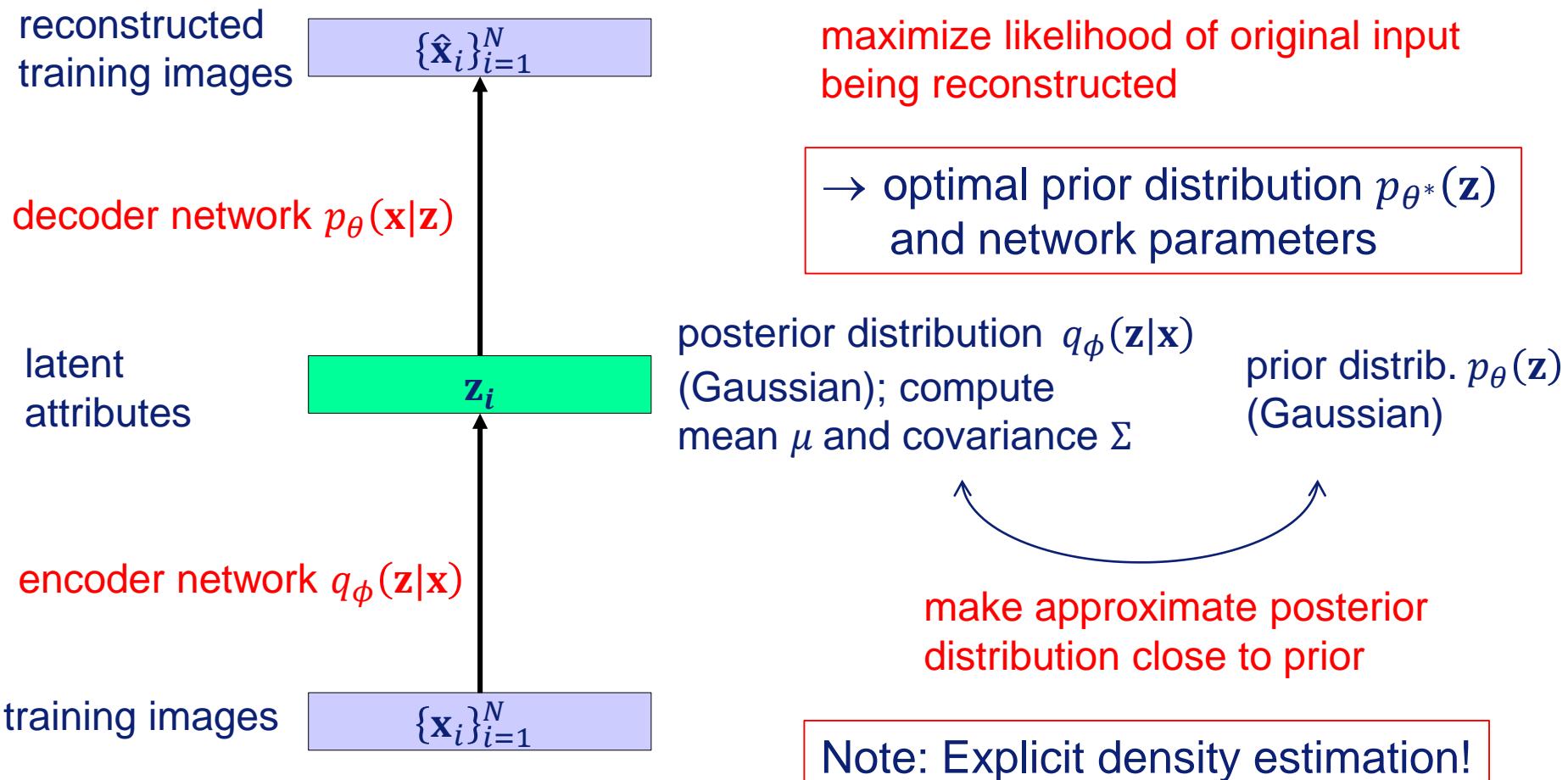
Conditional $p_{\theta^*}(x|z)$ is complex (generates image)
 → represent with neural network (decoder)

Choose prior $p_{\theta^*}(z)$ to be simple, e.g. Gaussian
 (reasonable for latent attributes)

- How to train the model parameters? **Maximum likelihood**
 - Estimate parameters to maximize likelihood for generating the training images
- **Likelihood:** $p_{\theta}(x) = \int p_{\theta}(z)p_{\theta}(x|z)dz$
- Problem: **Intractable** to compute $p_{\theta}(x|z)$ for every z
 - $p_{\theta}(z|x)$ also intractable due to intractable data likelihood $p_{\theta}(x)$
- Solution: Use additional encoder network $q_{\phi}(z|x)$ approximating $p_{\theta}(z|x)$

Variational autoencoder: Training

- Thus, instead of maximizing the likelihood of the training data directly: choose model parameters θ^* to **maximize a lower (variational) bound** on the training data → yields optimal prior distribution $p_{\theta^*}(\mathbf{z})$



Variational autoencoder: Training (full picture)

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbb{E}_z \left[\log p_\theta(x^{(i)} | z) \right] - D_{KL}(q_\phi(z | x^{(i)}) || p_\theta(z))}_{\mathcal{L}(x^{(i)}, \theta, \phi)}$$

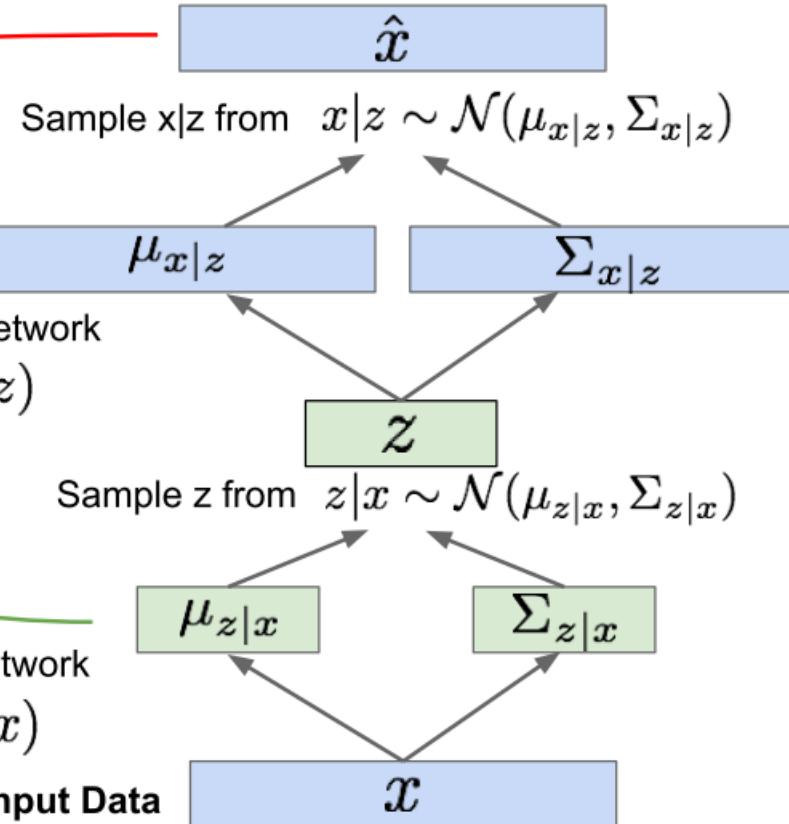
Make approximate posterior distribution close to prior

For every minibatch of input data: compute this forward pass, and then backprop!

Maximize likelihood of original input being reconstructed

Decoder network
 $p_\theta(x|z)$

Encoder network
 $q_\phi(z|x)$



From: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf

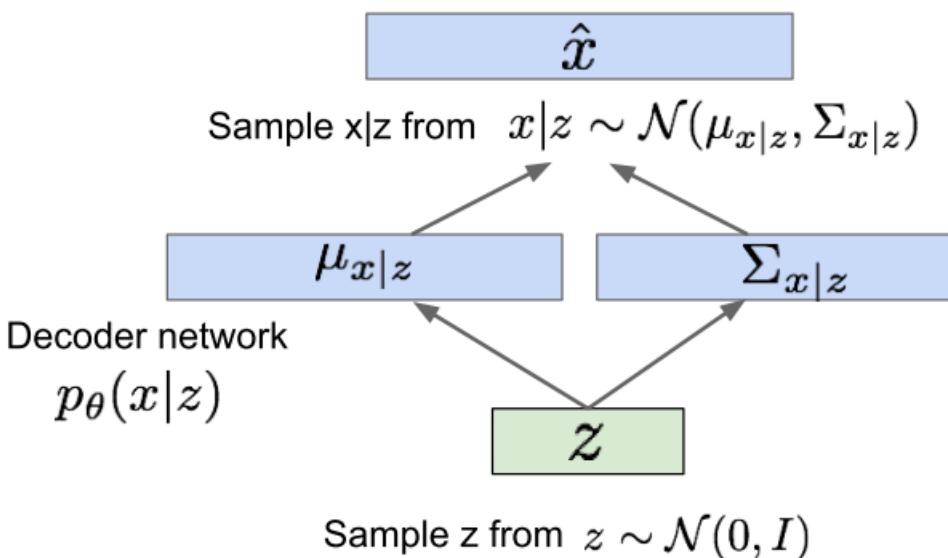
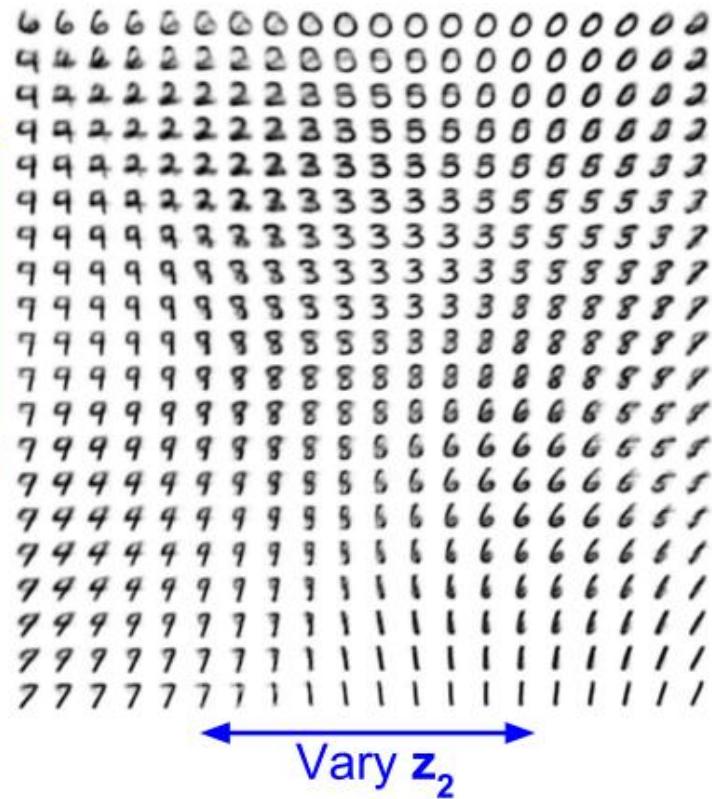
Variational autoencoder: Generating new data

- Now sample z from prior distribution $p_{\theta^*}(z)$
 → Use decoder network to generate new image x

sample z from prior distribution $p_{\theta^*}(z)$:

Example:

Data manifold for 2-d z



Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

From: http://cs231n.stanford.edu/slides/2017-02-05_vae.pdf

Variational autoencoder: Generating new data

- Further example:

Diagonal prior on \mathbf{z}
 \Rightarrow independent
latent variables

Different
dimensions of \mathbf{z}
encode
interpretable factors
of variation

Also good feature representation that
can be computed using $q_{\phi}(z|x)$!

Degree of smile
Vary z_1



Vary z_2
Head pose

Kingma and Welling, "Auto-Encoding Variational Bayes", ICLR 2014

Variational autoencoder: Summary

- Allows to generate new data
- Due to intractable probability distribution: derive and maximize a (variational) lower bound on training likelihood

Advantages:

- Principled approach to generative models
- Allows inference of $q_\phi(z|x)$, can be useful feature representation for other tasks

Disadvantages:

- Maximizes only lower bound on likelihood (suboptimal)
- Samples blurrier and lower quality compared to state-of-the-art (GANs)

Research: Better approximation, incorporating structure in latent variables

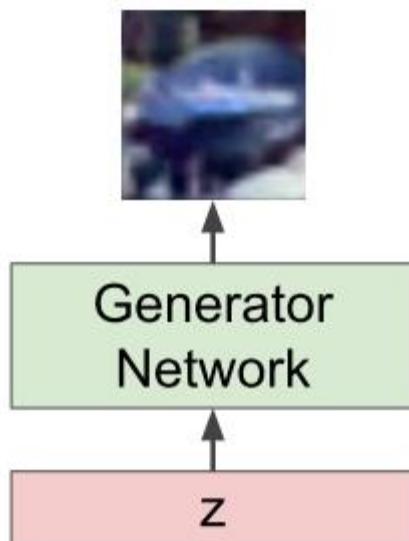
Alternative: Generative adversarial networks (GANs)

Generative models: Generative adversarial networks (GANs)

Generative adversarial networks: Idea (1)

- Goal: Sample from a (complex, high-dimensional) training distribution
 - No direct way to do this!
 - VAE: Explicit density, maximize likelihood to generate the training data
- Alternative:
 1. Sample from a simple distribution (e.g. random noise; „generator“)
 2. Let additional agent („discriminator“) „judge on quality“

Ad 1.:



Neural network (decoder, „generator“)
generates image x

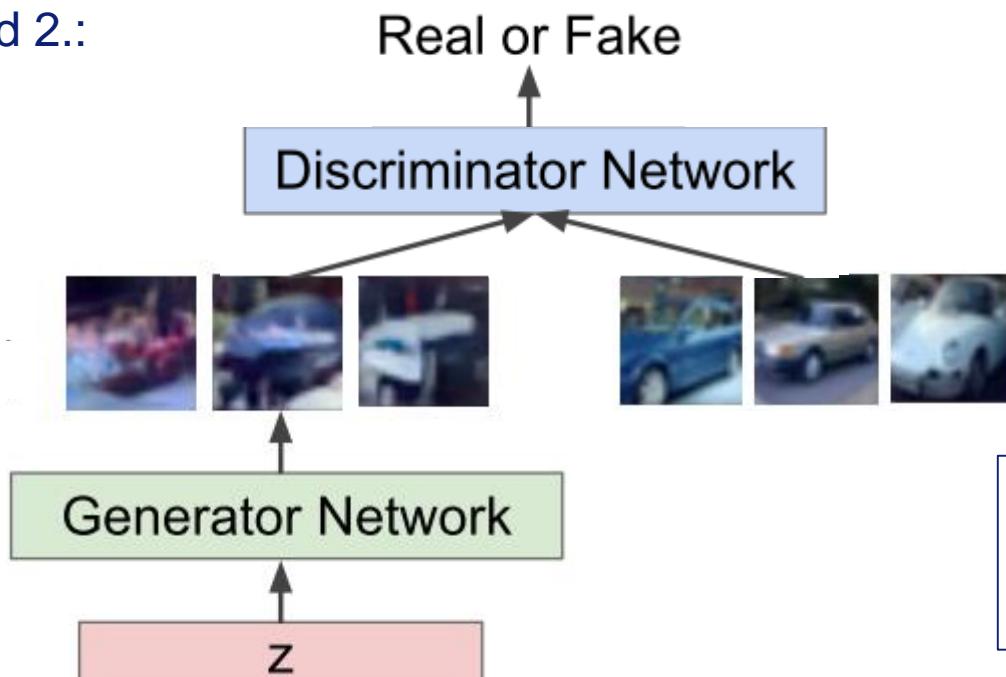
Input: randomly chosen z

Is this really a car image?
→ Must assure that generated
images are similar to training images

Generative adversarial networks: Idea (2)

- Goal: Sample from a (complex, high-dimensional) training distribution
 - No direct way to do this!
 - VAE: Explicit density, maximize likelihood to generate the training data
- Alternative:
 1. Sample from a simple distribution (e.g. random noise; „generator“)
 2. Let additional agent („discriminator“) „judge on quality“

Ad 2.:



Discriminator network:
Try to **distinguish** between
real and fake images

No explicit density!

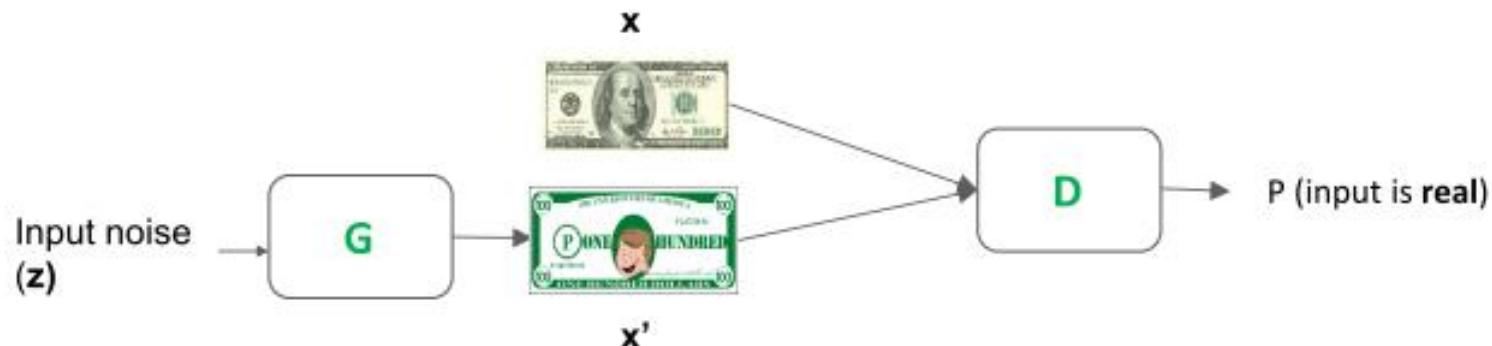
Generator network:
Try to **fool** the discriminator by
generating **real-looking** images

Generator network

- Input: Random noise z
- Output: A fake image $x' = G(z)$

Discriminator network

- Input: A real image x or a fake image $x' = G(z)$
- Output: Probability that input is **real**



Goal of generator:

- $D(G(z)) = 1$
- **minimize** $\log(1 - D(G(z)))$

Goal of discriminator:

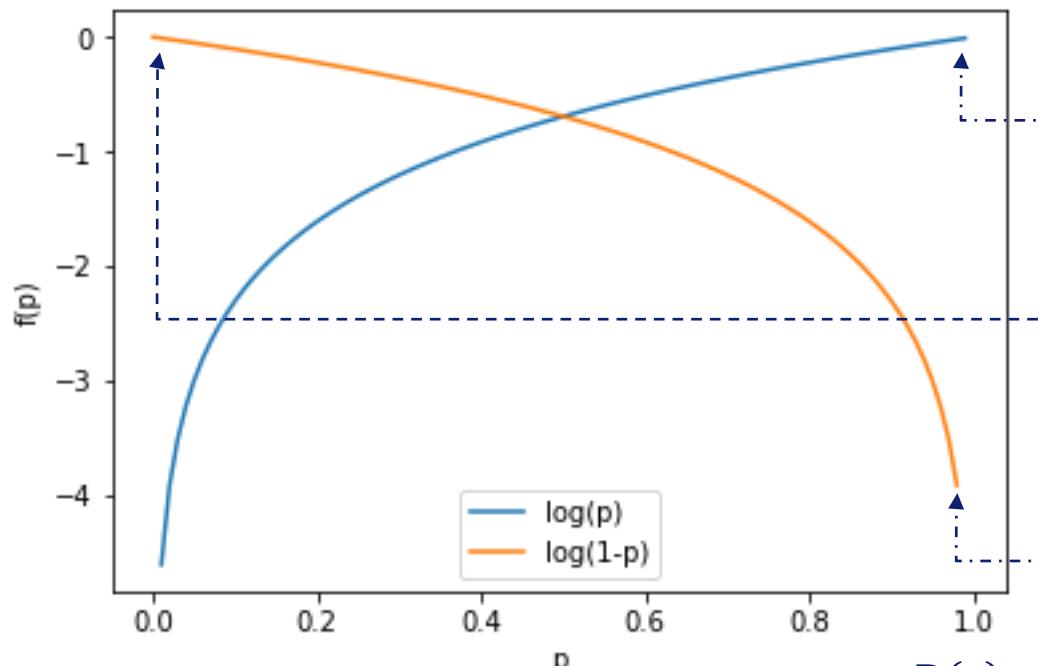
- $D(x) = 1$
 - $D(G(z)) = 0$
- **maximize** $\log(D(x))$
- **maximize** $\log(1 - D(G(z)))$

Minimax objective function

„minimax game“

 θ_g : parameters of generator
 θ_d : parameters of discriminator

- $$\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(x) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d}(\mathbf{G}_{\theta_g}(z)) \right) \right]$$
- „minimax objective“
- ↑
generator pushes down ↑
discriminator pushes up ↑
discriminator's ability to
recognize data as being real ↑
discriminator's ability to
recognize generator
samples as being fake

Discriminator outputs probability of real image, i.e. $\mathbf{D}(x)$, $\mathbf{D}(\mathbf{G}(z)) \in [0, 1]$ Discriminator wants $\mathbf{D}(x) = 1$
→ maximum of $\log \mathbf{D}(x)$ in $[0,1]$ Discriminator wants $\mathbf{D}(\mathbf{G}(z)) = 0$
→ max. of $\log(1 - \mathbf{D}(\mathbf{G}(z)))$ in $[0,1]$ Generator wants $\mathbf{D}(\mathbf{G}(z)) = 1$
→ min. of $\log(1 - \mathbf{D}(\mathbf{G}(z)))$ in $[0,1]$

Training GANs: Two-player game

- $\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$ „minimax objective“

Alternate between:

1. Gradient ascent on discriminator

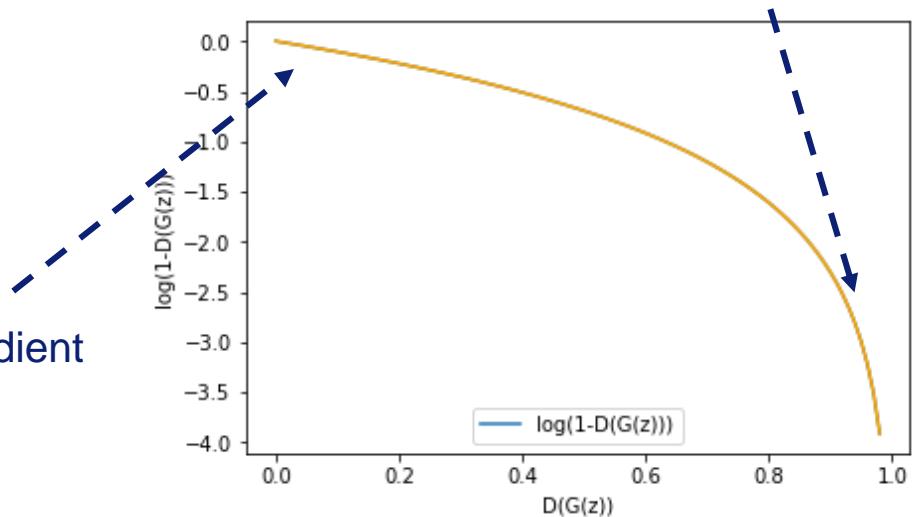
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$$

Gradient is *large* if $\mathbf{D}(\mathbf{G}(\mathbf{z}))$ is close to 1, i.e. when generated sample is already good

2. Gradient descent on generator

$$\min_{\theta_g} \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right)$$

Gradient is *small* if $\mathbf{D}(\mathbf{G}(\mathbf{z}))$ is close to 0, i.e. when generated sample is bad, we want to learn to improve, but get small gradient



→ in practice, optimizing this generator objective does not work well

Training GANs: Two-player game with modified generator objective

- $\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$ „minimax objective“

Alternate between:

1. Gradient ascent on discriminator (as before)

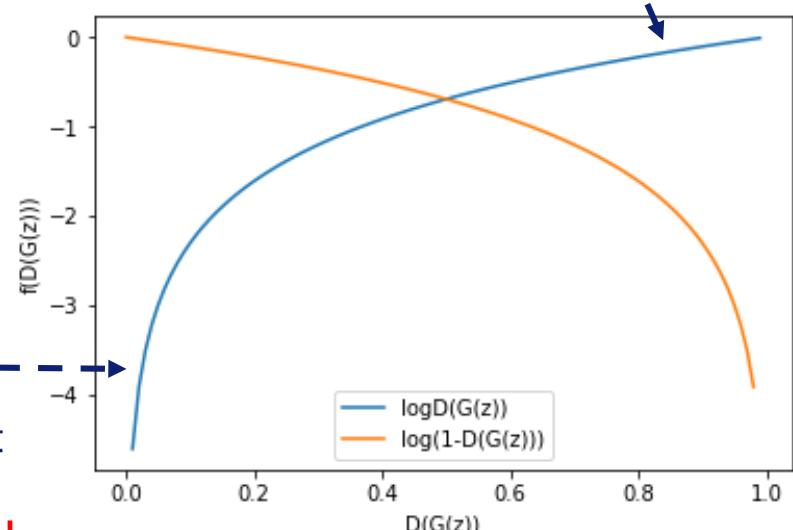
$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$$

2. Gradient ascent on generator, different objective (blue curve)

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log \left(\mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right)$$

Gradient is *large* if $\mathbf{D}(\mathbf{G}(\mathbf{z}))$ is close to 0, i.e. when generated sample is bad, we want to learn to improve and get *large* gradient

Gradient is *small* if $\mathbf{D}(\mathbf{G}(\mathbf{z}))$ is close to 1, i.e. when generated sample is already good



→ works much better → standard in practice!

Training GANs: Two-player game with modified generator objective

- $\min_{\theta_g} \max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$ „minimax objective“

Alternate between:

1. Gradient ascent on discriminator (as before)

$$\max_{\theta_d} \left[\mathbb{E}_{x \sim p_{data}} \log \mathbf{D}_{\theta_d}(\mathbf{x}) + \mathbb{E}_{z \sim p(z)} \log \left(1 - \mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right) \right]$$

2. Gradient ascent on generator, different objective (blue curve)

$$\max_{\theta_g} \mathbb{E}_{z \sim p(z)} \log \left(\mathbf{D}_{\theta_d} \left(\mathbf{G}_{\theta_g}(\mathbf{z}) \right) \right)$$

Instead of minimizing probability of discriminator being correct, now maximize likelihood of discriminator being wrong!

→ gives higher gradient signal for bad samples

→ facilitates learning

(jointly learning two networks is challenging and can be unstable!)

GAN training algorithm

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_z(\mathbf{z})$
- Sample minibatch of m examples $\{\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\mathbf{x})$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D_{\theta_d}(\mathbf{x}^{(i)}) + \log(1 - D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)}))) \right]$$

end for

- Sample minibatch of m noise samples $\{\mathbf{z}^{(1)}, \dots, \mathbf{z}^{(m)}\}$ from noise prior $p_g(\mathbf{z})$.
- Update the generator by ascending its stochastic gradient (improved objective):

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log(D_{\theta_d}(G_{\theta_g}(\mathbf{z}^{(i)})))$$

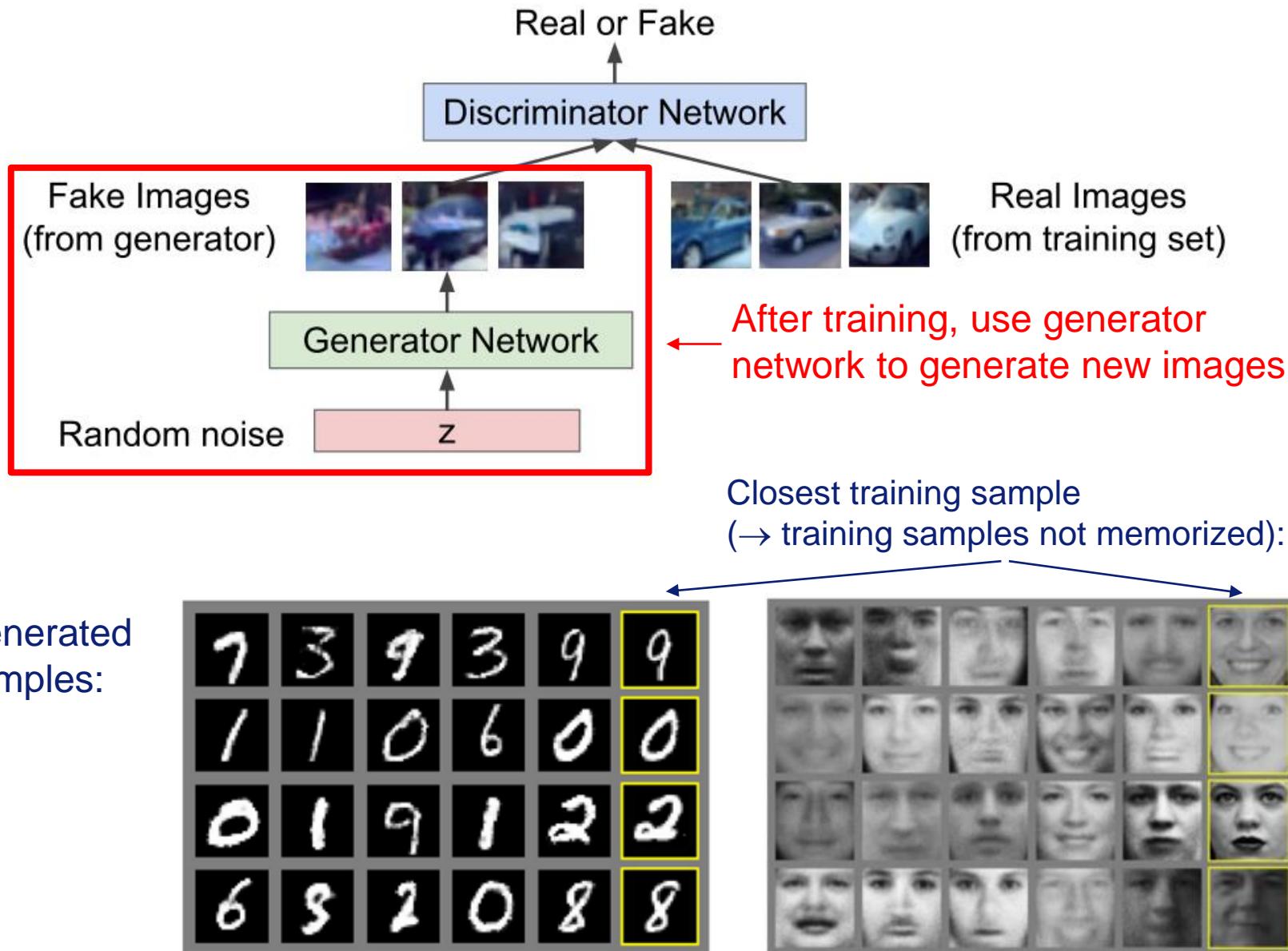
end for

Modify parameters of generator network
such that the objective function is maximized

Modify parameters of discriminator network
such that the objective function is maximized

number of steps k : some find $k = 1$ more stable, others use $k > 1$, no best rule

GANs: Generating images



DCGAN: Deep convolutional GAN (Radford et al., 2015) – architecture

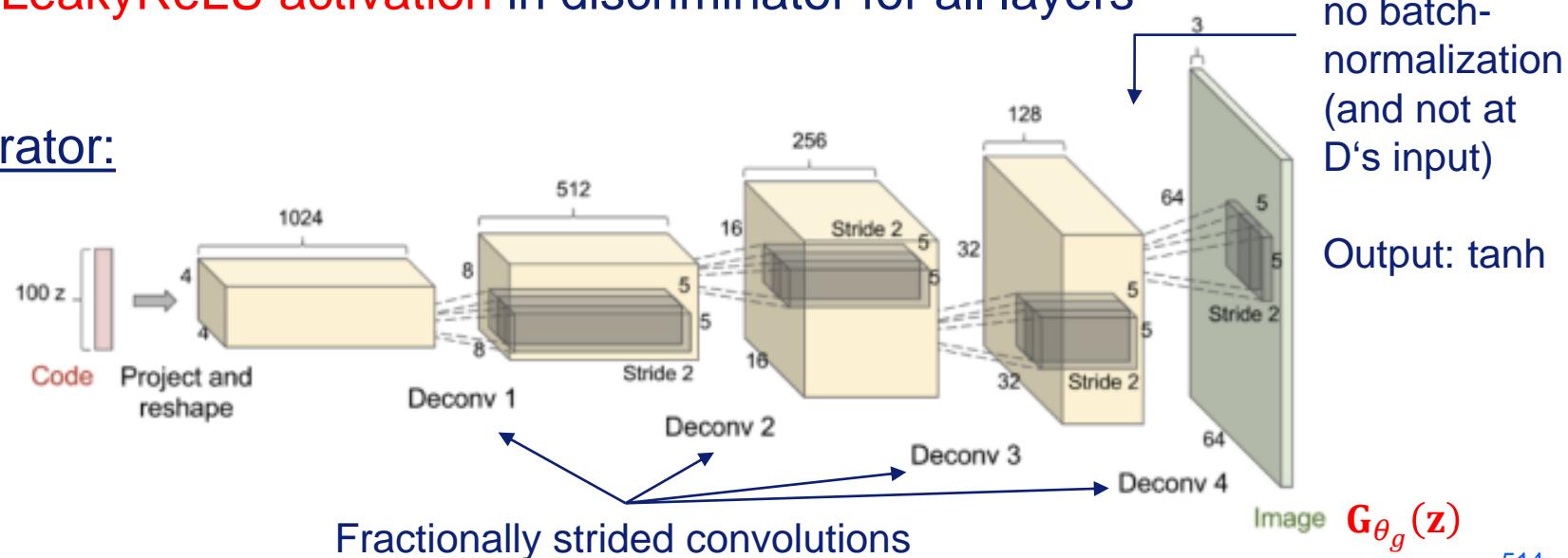
- Generator: Upsampling network with fractionally-strided convolutions
- Discriminator: Convolutional network

Architecture guidelines:

- Replace pooling layers with **strided convolutions (discriminator)** and **fractionally-strided convolutions (generator)** – these are *learned*
- Use **batch normalization** in both generator and discriminator
- Use **ReLU activation** in generator for all layers except for the output (tanh)
- Use **LeakyReLU activation** in discriminator for all layers

From: Radford, ICLR 2016

Generator:



DCGAN: Deep convolutional GAN (Radford et al., 2015) – examples (1)

- DCGAN can generate high quality images if trained in restricted domains
- Example: Bedroom images generated by a DCGAN trained on LSUN

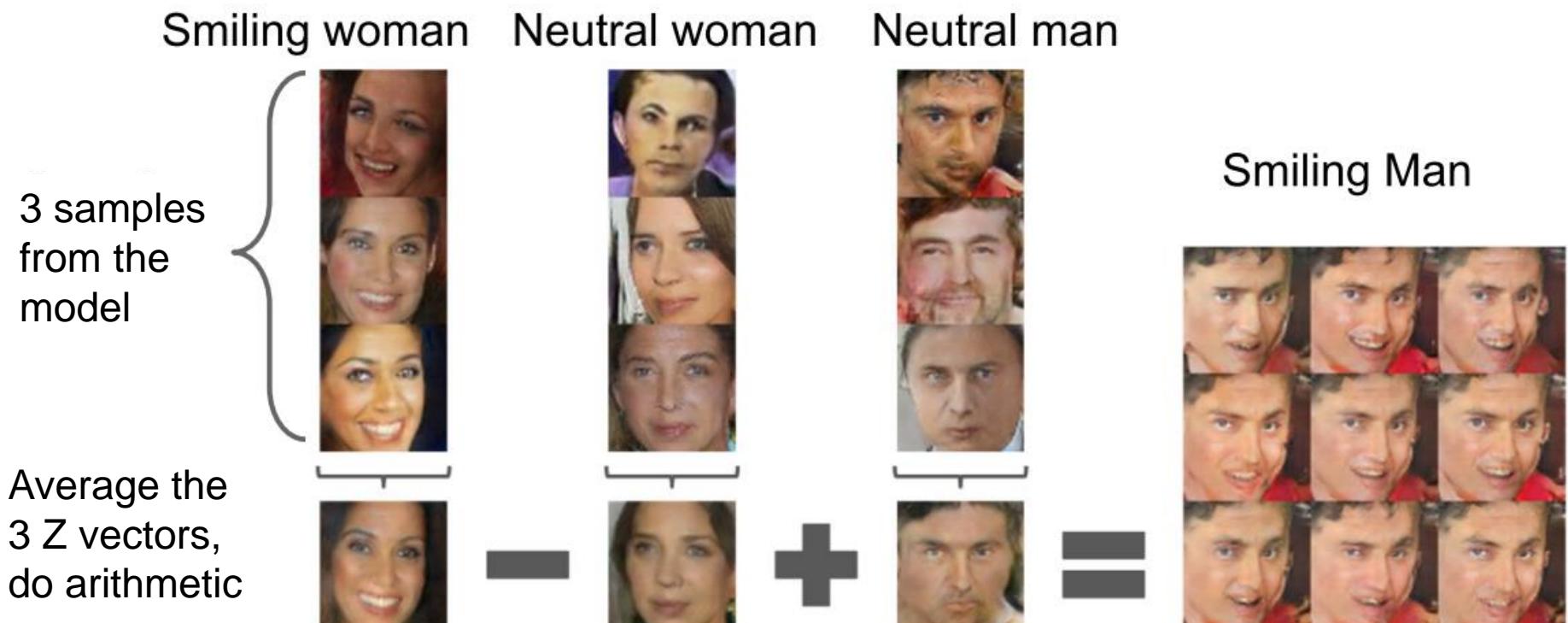


From: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf

515

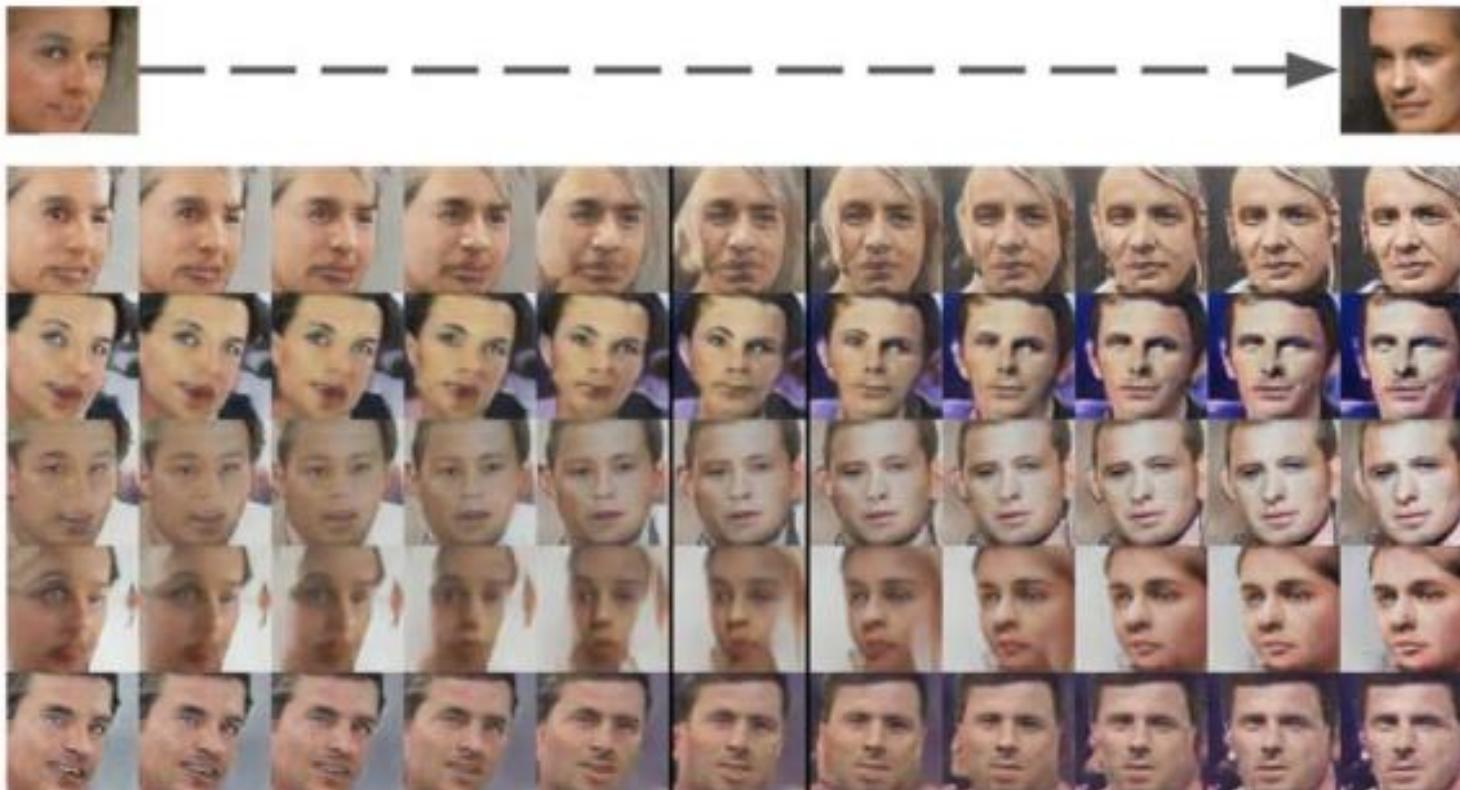
DCGAN: Deep convolutional GAN (Radford et al., 2015) – examples (2)

- DCGAN learns to use their latent code in meaningful ways, with simple arithmetic operations in latent space having clear interpretation as arithmetic operations on semantic attributes of the input



DCGAN: Deep convolutional GAN (Radford et al., 2015) – examples (3)

- „Turn vector“: By adding interpolations to random samples Z in latent space their pose can be transformed (Radford et al., 2015):

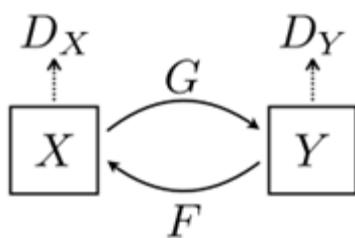


From: https://cseweb.ucsd.edu/classes/sp17/cse252C-a/CSE252C_20170412.pdf

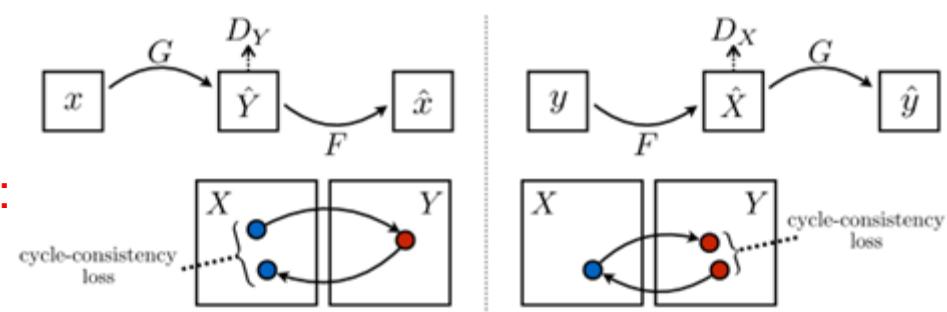
Turn Vector, from Radford, Alec, Luke Metz, and Soumith Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." *arXiv preprint arXiv:1511.06434* (2015).

CycleGANs: Unpaired Image-to-image translation

- CycleGAN: GAN using **two generators and two discriminators**
 - Plus adding a **cycle consistency loss**
- **Goal:** Translate image from domain X to another domain Y
 - E.g. photos to paintings, zebras to horses, summer to winter ...
 - Unpaired data: Independent samples from both domains needed, but no pair of data present in both domains
- **Idea:**
 - Generator $\mathbf{G}: X \rightarrow Y$ maps input image $x \in X$ to output image $\mathbf{G}(x) \in Y$
 - Discriminator \mathbf{D}_Y encourages \mathbf{G} to translate x into „real-like“ outputs in domain Y
 - Generator $\mathbf{F}: Y \rightarrow X$ maps input image $y \in Y$ to output image $\mathbf{F}(y) \in X$
 - Discriminator \mathbf{D}_X encourages \mathbf{F} to translate y into „real-like“ outputs in domain X
 - Plus **cycle consistency**



cycle
consistency:



CycleGAN: Applications (1)

- CycleGANs discover special characteristics of each image collection and how these characteristics should be translated into other image collection
- Examples: Collection style transfer

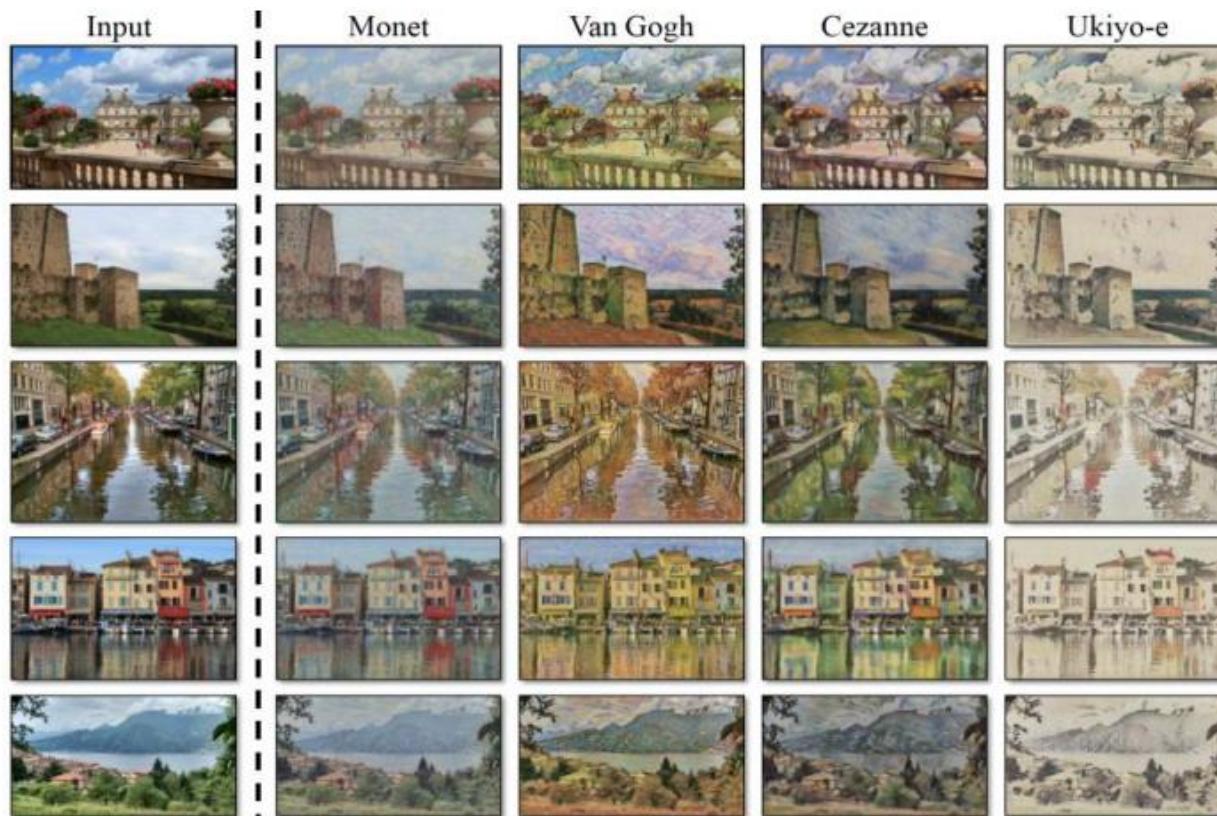
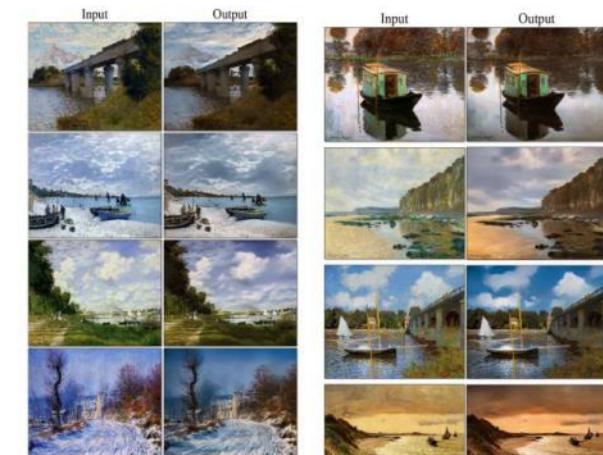
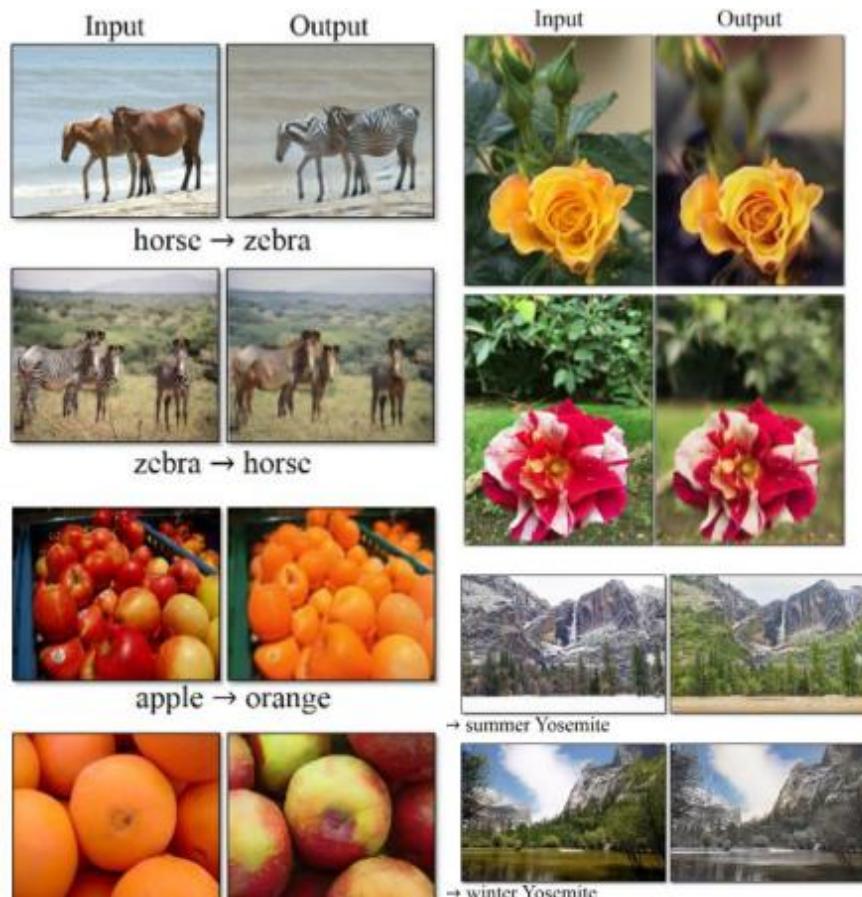


Photo generation
from paintings

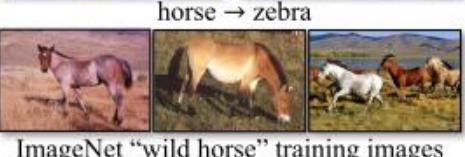
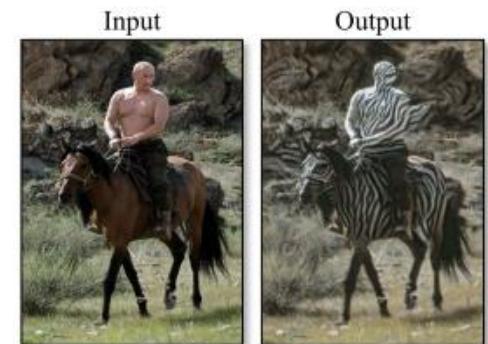


CycleGAN: Applications (2)

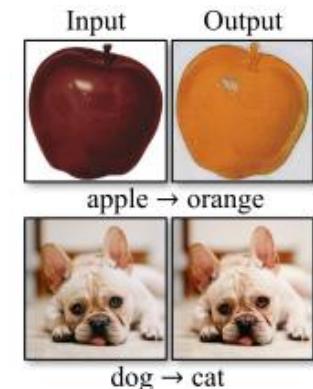
- CycleGANs discover special characteristics of each image collection and how these characteristics should be translated into other image collection
- Examples: source → target domain transfer



Failure cases:



ImageNet “wild horse” training images



GANs: Further applications

- Text → image synthesis:
 - StackGAN (Zhang et al., 2016)
- Semi-supervised learning
 - Feature-matching GANs (Salimans et al., 2016)
 - Use (small) set of supervised training data (K classes) in addition to many generated images („generated“ class $K + 1$), modified loss function
- Representation learning
 - DCGAN; use discriminator features from all layers (Radford et al., ICLR 2016)
- The „GAN Zoo“: <https://github.com/hindupuravinash/the-gan-zoo>
 - Lists more than 500 GAN variants (!! with applications (June 2019))
- When training GANs: **Use labels wherever possible**
 - GANs trained on specific classes work better than GANs free to generate from any class
- Further tips and tricks: see <https://github.com/soumith/ganhacks>

input: text

output: image
(Reed et al., 2017)

this small bird has a pink breast and crown, and black primaries and secondaries.



this magnificent fellow is almost all black with a red crest, and white cheek patch.



Generative adversarial networks: Summary

- GANs don't work with an explicit density function
- Instead, they apply a **game-theoretic approach** (2-player game)
 - Generator learns to generate samples which the discriminator cannot discriminate from real images

Pros:

- Can generate beautiful, state-of-the-art samples

Cons:

- Tricky, partly unstable to train (convergence not guaranteed)
- **Mode collapse** (generator maps several different z 's to the same output x)
 - Leads to relatively low output diversity, i.e. generated outputs are „too similar“
 - Potential solution: **Minibatch features** (\rightarrow use it's Theano / Tensorflow code)
- Can't solve inference queries such as $p(x)$, $p(z | x)$
- Sometimes problems with **counting, perspective, global structure**



SUMMARY

Learning: Overview

Feedforward
network

Recurrent
network

Unsupervised

Autoencoders

Self-organizing maps

Hopfield networks /
Associative memory:
• Hebbian learning

Supervised

Single-layer perceptron:

- perceptron learning
- gradient learning

Multi-layer perceptron:

- backpropagation

Convolutional neural networks:

- (unsupervised pre-training)
- supervised fine-tuning

Radial basis functions

Properties of neural networks

- Complex network behaviour by (non-trivial) connection of simple elements
- Parallel processing
- Information distributed over many neurons
- Neural networks generate an implicit model of the input data (without hypotheses)

General applications of neural networks

Pattern classification / recognition:

- Single-layer / multi-layer perceptron (softmax)
- Deep networks, CNNs
- Self-organizing map

Function approximation, regression:

- Single-layer / multi-layer perceptron (linear or sigmoid activation)
- Radial basis function networks

Pattern association / associative memory:

- Heteroassociative: associative memory
- Autoassociative: Hopfield network

Modeling of time sequences, prediction:

- Multi-layer perceptron
- Recurrent networks (LSTMs, Hopfield)

Applications of neural networks

Visual (image) recognition

- objects, faces, characters / text, images
- 2-dim. edge detection with perceptrons
- NETtalk (text to speech)

Speech and language processing

- Automatic speech / speaker recognition
- Natural language processing
- Machine translation

Autonomous systems

- Vehicle control, localisation in space, planning, avoiding obstacles
- robotics, e.g. coordination of robotic arm

Industrial process optimization, planning and control

- Control engineering

Filtering, Clustering

- Data coding, data compression
- Spam filter
- Data mining, information retrieval

Data analysis

- Medical diagnoses & decision support
- Time series prediction, e.g. finance (stock quotes), weather

Neural networks in practice

Problem selection

- problems for which no rule-based or deterministic solution is available

Data preprocessing

- e.g. normalization, filtering etc. (if appropriate)

Network topology

- appropriate activation function (matching problem at hand)
- network should be as small as possible (to minimize overfitting)

Selection of training data

- Representative
- Consistent
- Concise

Selection of training algorithm

- matching activation function
- Matching patterns (uncorrelated?)
- extension of basic algorithms?

Stopping criterion / generalization

- cross-validation
- *local* minimum?

Advantages of neural networks

- Tolerance against internal defects and external perturbations / errors
 - noise, missing neurons or connections
- High learning potential
- High generalisation ability
- Associative memory, i.e. retrieval with *partial* information
- High speed of classification / retrieval
- Deep learning: Top performer in many application fields

Disadvantages of neural networks

- Knowledge *only* from learning
- No guarantee for training success
 - Network topology might not be adequate
 - Training algorithm might not be adequate
 - Training data might be inconsistent or non-representative
 - Algorithm might be trapped in local minimum
 - Potentially long training times (local minima)
- No guarantee for generalisation ability (overfitting)
- Training result and network output not always intuitive („black box“)
- Many parameters

Some controversial issues (according to Simon Thorpe, 2008)

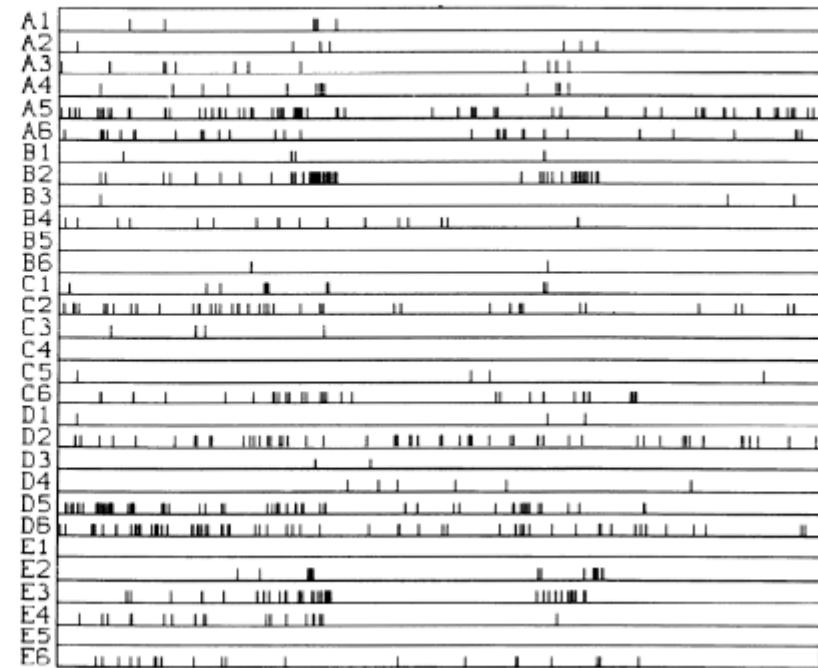
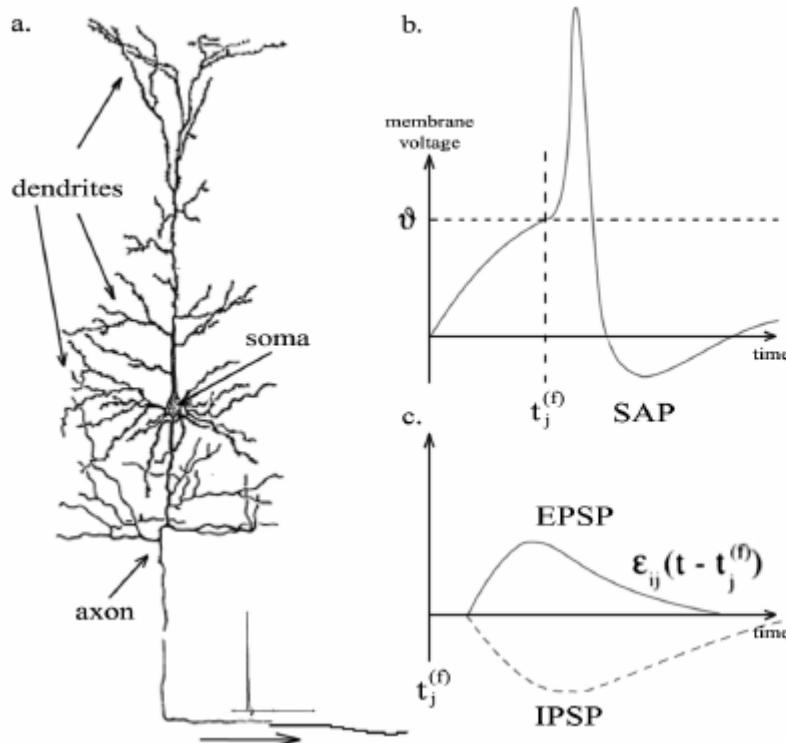
- Complex neurons vs. simple integrate-and-fire models
 - Do we need to model very detailed neuronal properties to understand vision?
→ Maybe not. See how far we can get with the simplest ideas!
- Feed-forward vs. feed-back processing
 - How much vision can be done with a feed-forward pass?
→ Quite a lot!
- Local vs. distributed coding
 - Are there „grandmother cells“ in the visual system?
→ Yes, for highly familiar and important stimuli
- Rate-based vs. temporal coding
 - Do neurons send analog values as a rate code?
→ Relative spike timing is a better solution
- Noise and visual computation
 - Does the brain need to add noise to allow computation?
→ Maybe not – variability may be entirely due to lack of experimental control

Some alternatives to neural networks

- Pattern classification and recognition:
 - Support vector machines
 - Decision trees
 - Probabilistic models
- Clustering:
 - Mixture models
 - Principal component analysis
 - Hierarchical clustering
- Regression:
 - Linear regression
- Temporal sequences:
 - Hidden Markov Models

The future of neural networks... (1)

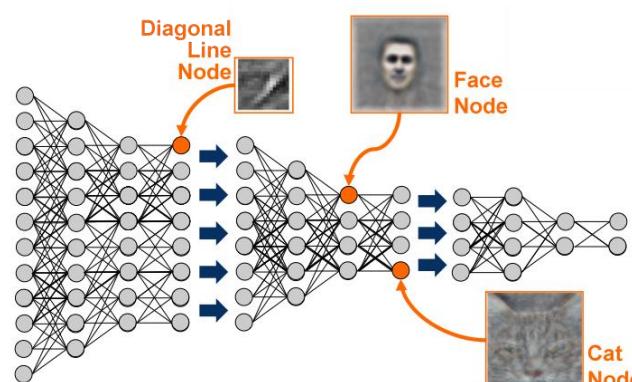
- 3rd generation of neural networks:
Spiking neural networks / pulse-coupled neural networks
- Goals:
 - Simulation of biological neural networks
 - Understanding the principles of information processing in the brain



The future of neural networks... (2)

- Further improvements in deep learning / convolutional neural networks:
 - Better unsupervised learning
 - New algorithms
 - Deep reinforcement learning
 - Attentional models for computer vision
 - More complex computer vision systems
 - Object detection / localisation
 - Video classification
 - Image caption generation
 - More complex natural language processing systems
 - End-to-end machine translation
 - Question answering

From: <http://theanalyticstore.com/wp-content/uploads/2013/04/DeepNetwork.png>



From: Larochelle

Sources (1)

- Amit: Modeling brain function - The world of attractor neural networks, Cambridge University Press (1992)
- Berg: http://wwwhep.physik.uni-freiburg.de/physik_am_samstag/2009/vortrag_berg.pdf
- Bittel: http://www-home.fh-konstanz.de/~bittel/nndl/NeuroNetze_1.pdf
- Bruse: http://www.staff.uni-mainz.de/bruse/10Info_MAS/PDF/MAS_05_neuroaleNetze.pdf
- Hartmann: <http://www.caesar.de/kursmaterialien.html?&L=unjnagfl#KNN>
- Kriesel: http://www.dkriesel.com/science/neural_networks
- Lippe: <http://cs.uni-muenster.de/u/shirvanian/lehre/ws1011/neuroinformatik/index.html>
- Martinetz: <http://www.inb.uni-luebeck.de/lehre-de/ss08/ni1>
- Riedmiller (not available anymore)
- Ruttloff: http://www-user.tu-chemnitz.de/~stj/lehre/prosem0910/Neuronale_Netze.pdf
- Schwarz: <http://wwwstud.fh-zwickau.de/~sibsc/lehre/ss10/nn/>
- Schwenker: <http://www.informatik.uni-ulm.de/ni/Lehre/WS06/NI1/>
- TU Graz: <http://www.igi.tugraz.at/lehre/CI/lectures/lecture2.pdf>
- U Hildesheim: www.uni-hildesheim.de/.../IV.%20Neuronale%20Netze.ppt
- U Rostock: wwwteo.informatik.uni-rostock.de/~le/Lehre/WS1011/ws1011-9.ppt
- Larochelle: https://dl.dropboxusercontent.com/u/19557502/ecml-pkdd_slides.pdf
- Illinois: <https://courses.engr.illinois.edu/cs598ps/ewExternalFiles/Lecture%2020%20-%20Deep%20learning.pdf>
- Riedmiller: http://ml.informatik.uni-freiburg.de/_media/teaching/ws1415/presentation_dl_lect1.pdf
- Wang: <https://s3.amazonaws.com/piazza-resources/i48o74a0lqu0/i4jo66igh6k3pw/Introduction.pdf?AWSAccessKeyId=AKIAJKOQYKAYOBKKVTQ&Expires=1425405822&Signature=qFpAZD5aNbq8wu%2Bv8GKKn3%2BqE98%3D>
- Choi: <http://people.cs.pitt.edu/~milos/courses/cs3750/lectures/class26.pdf>
- Ranzato: http://www.cs.toronto.edu/~ranzato/publications/ranzato_cvpr13.pdf
- LeCun: <http://www.cs.nyu.edu/~yann/talks/lecun-ranzato-icml2013.pdf>
- Huang (Virginia Tech): https://filebox.ece.vt.edu/~jbhuang/teaching/ece5554-4554/fa17/lectures/Lecture_20_CNN.pdf

Sources (2)

- Karpathy: <http://cs231n.stanford.edu/>