



Machine Learning

Lecture 1: foundations of machine learning
Alexander Gepperth, November 2021



"Intelligent" Systems

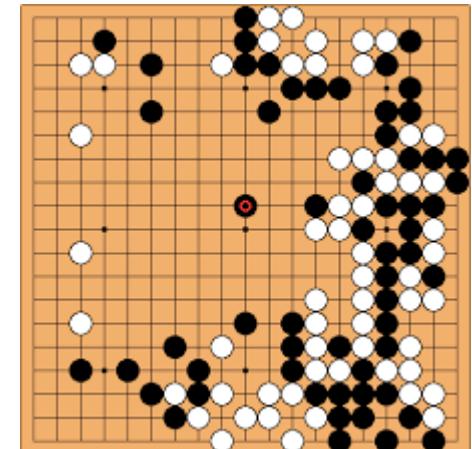
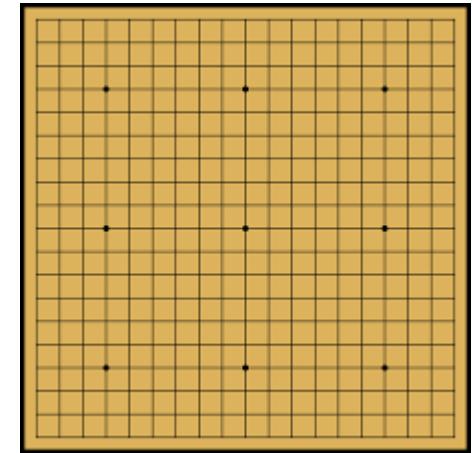
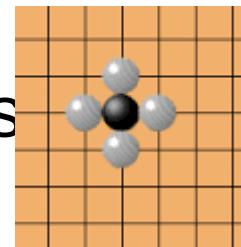
- 1996: “Deep Blue” beats world champion Garry Kasparov in the game of chess
- 2016: “AlphaGo” beats world champion Lee Sedol in the game of Go





The game of Go

- 19x19 board
- black and white pieces are placed on line intersection points
- pieces do not move..
- .. except: surrounded pieces removed from the board
- to win → “occupy” more territory!





"Intelligent" Systems

- Google Car: “autonomous” driving



- Tesla “Autopilot”





"Intelligent" Systems

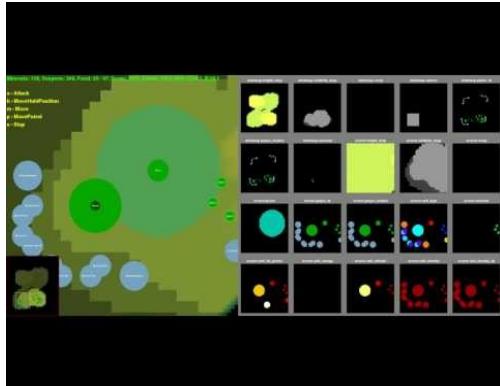
- AIs and Bots for FPs and RTS games





"Intelligent" Systems

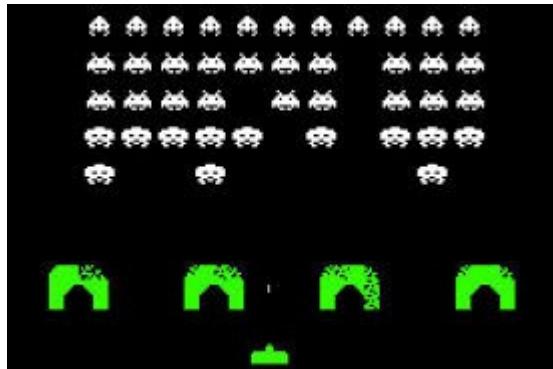
- DeepMind (Google) reports success on intelligent StarCraft II bot: AlphaStar





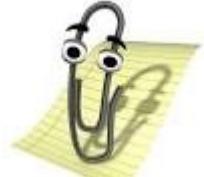
"Intelligent" Systems

- Bots play old Atari video games using “Deep Reinforcement learning” (DQN)





"Intelligent" Systems



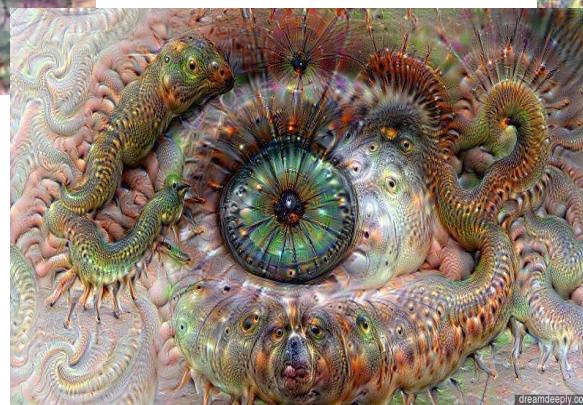
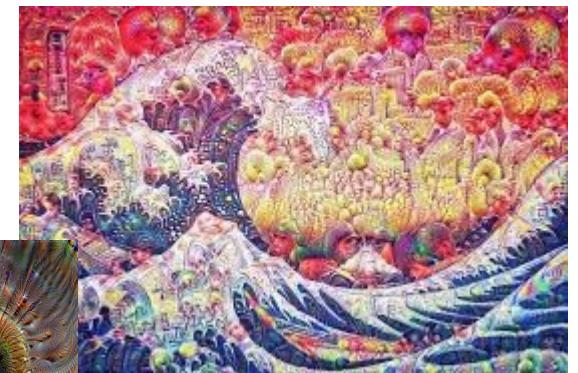
- Speech recognition and dialog systems
- Personal Assistants ("Alexa", "Siri", "Cortana", ...)
- Recommender systems
- Machine Translation





"Intelligent" Systems

- Deep dreaming: Deep Learning creates art!





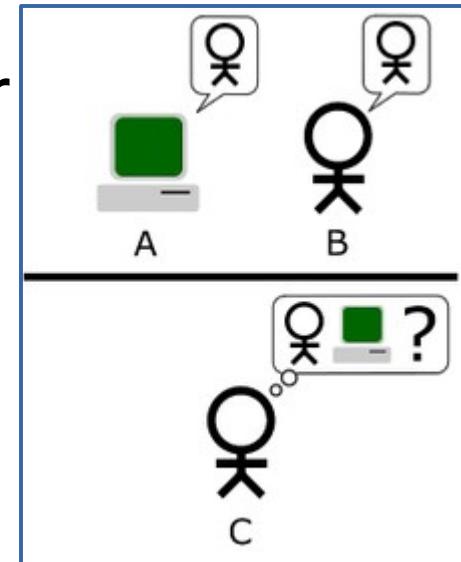
What is “Intelligence”?

- Memory
- draw conclusions
- discover regularities
- categorization
- quick on the uptake
- learn rapidly
- recognize faces
- curiosity
- do maths
- language understanding
- IQ tests
 - Language
 - Numbers
 - Visual understanding
 - meant to have mean 100, 15 stddev
- **HARD to define!!**



The Turing-Test

- Intelligence is hard to define...
- ... but there is a simple yes/no test for it
→ Turing Test (~1950)
- In modern terms: “can a computer chat with a human without the human recognizing it as such?”





Machine Learning -vs- machine intelligence

- Learning has high application relevance
- Learning is supported by a mathematical theory
 - clear **definition** of "learning"
 - methods for **measuring** learning outcome
 - methods for **comparing** learning methods
 - much better suited for study by scientists and engineers



The Turing-Test :-)

- www.xkcd.com/329/

TURING TEST EXTRA CREDIT:
CONVINCE THE EXAMINER
THAT HE'S A COMPUTER.

YOU KNOW, YOU MAKE
SOME REALLY GOOD POINTS.
I'M ... NOT EVEN SURE
WHO I AM ANYMORE.





Cut: Q&A



Mathematical notation



Notation

- We use the following shorthand expressions for sums and products:

$$\sum_{i=1}^N x_i \equiv x_1 + x_2 + \cdots + x_N$$

```
# compute sum  
s = 0 ;  
for el in x:  
    s += el ;
```

$$\prod_{i=1}^N x_i \equiv x_1 x_2 \dots x_N$$



Notation

- When it is obvious, summation ranges are often omitted:

$$\sum_{i=1}^N \dots \equiv \sum_i \dots$$

- we use the following shorthand expressions for sums and products:

$$\sum_{i,k} \dots \equiv \sum_i \sum_k \dots$$



Tools

- Arithmetic mean:

$$E(x_i) = \frac{1}{N} \left(\sum_{i=1}^N x_i \right)$$

- Scalar product:

$$\langle \vec{x}, \vec{y} \rangle \equiv \sum_{i=1}^N x_i y_i$$

- Geometric mean:

$$A(x_i) = \sqrt[n]{\prod_{i=1}^N x_i}$$



Math basics: tensors, scalars

- Scalars are real numbers: $x \in \mathbb{R}$
- Tensors are organized sets of scalars accessed by **one-based** indices: $T_{12345}, M_{12}, W_{ij}$
- We use upper-case letters for tensors
- Number and allowed ranges of tensor indices are stored in a tuple that we call **shape**: $\text{sh}(T) = \text{e.g., } (5, 1, 2, 10)$
- Alternative notation for shape: $T \in \mathbb{R}^{5,1,2,10}$
- How many elements has the tensor T??



Special case: matrices

- Two-dimensional tensors are termed matrices:
- Matrices can be seen as organized in R rows and C columns: $M \in \mathbb{R}^{R,C}$
- Convention: row index comes first: M_{34}
- Transposition flips rows and columns: $M \in \mathbb{R}^{R,C} \rightarrow M^T \in \mathbb{R}^{C,R}$

A diagram illustrating matrix indexing. An arrow points from the text "3rd row" to the row index "3" in the term M_{34} . Another arrow points from the text "4th column" to the column index "4" in the same term.

<Blackboard>



Special case: vectors

- One-dimensional tensors are termed vectors: $\vec{x} \in \mathbb{R}^n$
- Lower case letters! Arrow notation for whole vector, no arrow for component scalars: \vec{a} –vs– a_4
- Vectors can be seen as matrices with one row (row vector) or column (column vector). Default: col. vector $\vec{t} \in \mathbb{R}^{n,1}$ (important for matrix mult.)
- Transposition switches between row and column representation: $\vec{t} \in \mathbb{R}^{1,n} \rightarrow \vec{t}^T \in \mathbb{R}^{n,1}$



Tensor Slicing

- Taking out “parts” of tensor:
 - taking out row k of matrix $M \in \mathbb{R}^{4,17}$ is written as: $M_{k,:}$
 - taking out column k is written as: $M_{:,k}$
 - taking out columns 2 through 6 from row k is written as: $M_{k,2:7}$ (last index not included!)
- What are the dimensions of the resulting tensors?
- Row k of a matrix X is often written as: $\vec{x}_k^T = X_{k,:}$



Tensor reductions

- Tensor reductions: summing over a single dimension of a tensor gives a different tensor from which this dimension is removed
- Example: Let $M \in \mathbb{R}^{4,3,2}$. Now $\tilde{M}_{ij} = \sum_k M_{ijk}$
What is shape of \tilde{M} ?

More examples on blackboard



Matrix multiplication

- Special case of tensor reduction when both tensors M, L are matrices
- M.M. returns another matrix K :
$$K_{ij} = \sum_a M_{ia} L_{aj}$$
- Often simply written as: $K = ML$

examples on blackboard



Scalar and vector-valued functions

- A scalar-scalar function maps a scalar to a scalar:
 $f : x \in \mathbb{R} \mapsto y \in \mathbb{R}$
- A vector-scalar function maps a vector to a scalar:
 $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$
- A vector-vector function maps a vector to a vector:
 $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}^m, n \neq m$ or even $\vec{f}(\vec{x} \in \mathbb{R}^n) \in \mathbb{R}^m$
- Vector-vector functions are often written in component notation: $f_i : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}, i = 1, \dots, m$



Shorthand notations for functions

- Often, data are represented as a matrix, with individual data vectors represented by rows
- Scalar-scalar functions applied to a tensor are understood to be applied component-wise (**example**)
- Vector-scalar and vector-vector functions applied to a matrix are understood to be applied row-wise (**example**)



Cut: Q&A

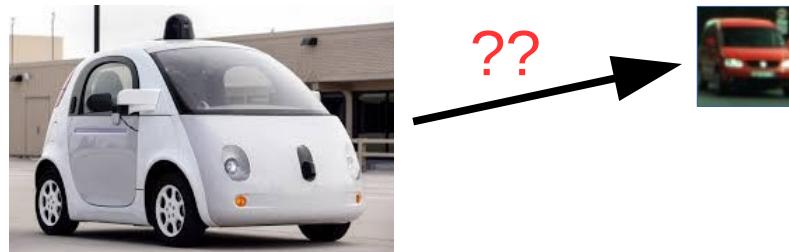


What is machine learning?



Why machine learning?

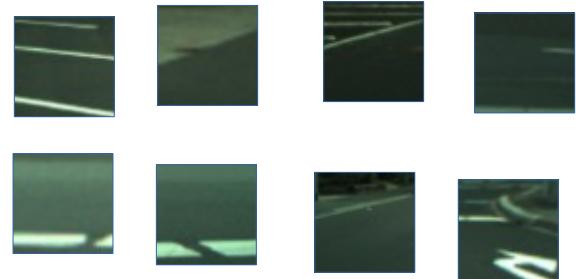
- Problem: recognition of cars in videos
 - Important for, e.g., autonomous driving
 - Goal: sort perceived objects into categories like cars, pedestrians, signs, traffic lights, ...





Example: why machine learning?

- Basic task: distinguish vehicles from background patches



transformation

"vehicle" / 2

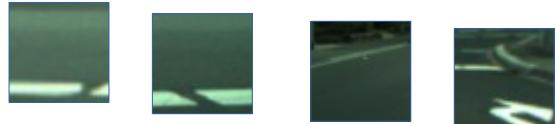


"background" / 1



Example: why machine learning?

- Basic task: distinguish vehicles from background patches

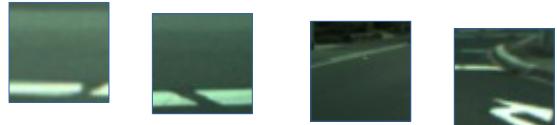


- problem: transform image → discrete (integer!) value:
"categorization" or "classification"
- Alternative: transform image → continuous value:
"regression"



Example: why machine learning?

- Basic task: distinguish vehicles from background patches



- Mathematical problem formulation:
 - single 50x50 mono image: vector $\vec{x} \in \mathbb{R}^{2500}$
 - transform: look for a function $f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$



Example: why machine learning?

- Function $f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$
 - ***model function***
 - continuous, differentiable
 - efficient (had better be...)
 - unknown!





Example: why machine learning?

- How to obtain model function?

$$f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$$

- **if-then-else by hand?**

```
// Klassifikation in Java
float [] x = getImageFromCamera() ;
if (((x[0] > 13) && (x[0]<45)) &&
    ((x[1] > 46) && (x[1] < 213)) &&
    ...
    (((x[2499] > 1) && (x[2499] < 213)) ||
     ((x[2499] > 240) && (x[2499] < 243)))
y = 1. ; else y = 0 ;
```



– another way, maybe?



Example: why machine learning?

- How to obtain model function?

$$f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$$

- automatic determination from N **training samples**

- **training samples** must represent problem "sufficiently well"
 - first approximation: enough samples in both classes





Example: why machine learning?

- How to obtain model function?
 $f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$
- automatic determination
from N training samples



machine learning!



Example: why machine learning?

- How to obtain model function?

$$f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$$

- automatic determination from **N training samples**

- images: matrix $X \in \mathbb{R}^{N,2500}$

- target values: matrix $T \in \mathbb{R}^{N,1}$

- individual images or target values often written as $\vec{x}_i^T = X_{i,:}, \vec{t}_i^T = T_{i,:}$

- here: target values are scalars: 1,2





Example: why machine learning?

- Goal of learning: determine "good" model function
$$f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$$
- “good”?
 - assign right category to all training samples
 - or at least: to as many sample as possible
 - **how to measure this?**





Example: why machine learning?

- How can we measure how "good" a model function $f : \vec{x} \in \mathbb{R}^{2500} \mapsto y \in \mathbb{R}$ is ?
- need a **loss function**
e.g, **classification error**:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \left\{ \begin{array}{ll} 1 & f(\vec{x}_i) \neq t_i \\ 0 & \text{else} \end{array} \right.$$





Example: why machine learning?

- Classification error
 - measures probability of an incorrect classification
 - clearly: smaller = better!
- other loss functions are possible, depends on problem
- **Goal of machine learning:**
find model function with smallest \mathcal{L}





How to represent model function?

- Choose a function family with parameter vector \vec{w}
 $f(\vec{x}, \vec{w})$
- parameter controls behavior of model
- better parameter \rightarrow better model!
- after functional form is fixed, improving the model is done by finding a better \vec{w}



From example to formalization

- Later :-)





Cut: Q&A



Python



Generic stuff

- supports OOP and imperative style
- makes use of indentation instead of block delimiters like { and }
- Semicolons @ end of statement possible
- interpreted language (mostly): supports program execution and interactive mode
- Python2 and Python3 → we use Python3!



OOP

- Python is extremely object-oriented: all entities are objects
 - can have methods
 - can have data field (attributes)

```
x = "string object" ;
# methods are attributes
print (x.upper) ;

# methods can be called
print( x.upper() ) ;

# non-method attributes
x = 1 ;
print (x.numerator) ;
print ((1).numerator) ;
```



Functions and arithmetics

- functions are defined by keyword `def`
- return a value with keyword `return`
- multiple values can be returned as a `tuple` (later!)
- without return, `None` is returned
- return value can be ignored!
- normal arithmetics are supported
- automatic type-casting to wider data type in all operations: $1 + 2.0 \rightarrow 3.0$
- no type casting otherwise: $5 / 7 \rightarrow 0$

```
def f(x, y):
    return x+1,x-1, (x+5)/x ;

# use return value
print ("return: ", f(2)) ;
# ignore return value
f(x) ;
```



Classes and objects

- Classes are defined by the keyword `class`
- Constructors are called `__init__`
- Constructors define and initialize instances
- Each class method takes the first parameter '`self`'
- Instantiation: "calling" the class object

```
class Test:  
    def __init__(self, start):  
        self.x = start  
  
    def f(self):  
        return x  
  
instance = Test(1)  
print( instance.f() )
```



Variables and references

- Python does not copy except when you tell it to
- variables are references (pointers in C, reference data types in Java) to their values
- assignments: no copy, reference is changed!
- check for same object: operator “`is`”
- check for same value: operator “`==`”

```
x = 1 ;  
y = 2 ;  
x = y ;  
print(x is y) ;  
print (x == y) ;
```



Lists

- Lists: 1D sequences of elements
- nr of elements: builtin function `len()`
- elements of any type, need not be same
- lists are **mutable** (can be modified)
- list access happens through indices of type int

```
x = [1,2,3] ;
print(x[0]) ;
x[0] = value ;
print(x)

y = x ;
y[2] = "xx" ;
print(x) ;
```

```
x = [1,2,3] ;
y = [1,2,3] ;
print(x is y) ;
print(x == y) ;
```



List creation

- Several standard ways to create and initialize lists
 - direct specification
 - “multiplication” (dirty)
 - “list comprehension”
 - append to empty list
 - constructor of class list:
(later!)

```
print( [1,2,3] ) ;  
  
print( [0]*5 ) ;  
  
print([i for i in range(0,10)])  
  
x=[] ;  
for i in range(0,len(x)):  
    x.append( i ) ;
```



List slicing

- slicing: copy out a part of a list, specified by
 - start index
 - stop index (not included)
 - step
- start/stop/step are optional
- creates a copy!
- will be discussed in more detail in the context of arrays!

```
x = [1,2,3] ;
print(x[2:0:-1]) ;
print(x[2::-1]) ;
x [:] is x
```



For loops

- In Python, a for loop must iterate over a data structure (an iterable)
- The function `range(start, stop, step)` creates a pseudo-structure to loop over
- Block indentation!!

```
for i in range(0,10):  
    print (i)
```

```
for el in [1,2,3]:  
    print (el)
```



Tuples

- tuples are “immutable lists” (`len()` can be used!)
- tuples are mainly created by direct specification
- or by constructor of class `tuple`
- sometimes, brackets are required to eliminate ambiguity
- tuples are useful to return multiple values from a function

```
def f(x):  
    return x+1,x+2  
  
t = 1,2,3 ;  
p = 1,  
l = len( 1,2,3 ) ;  
print( f( 2 ) ) ;  
t[0] = 1;
```

error!



Dictionaries

- Associative data type: works like a list but "indices" can be anything
- creation: extend empty dictionary by assignments

```
d = {} ;  
d["one"] = [] ;  
d[2] = 100,200 ;  
d[(1,2)] = 0.0 ;  
print (d["one"]) ;
```



Pass-By-Reference

- mutable data can be changed by whoever has a reference to them
 - function parameters
 - copied references
- since everything works by reference in Python, care must be taken to avoid side effects

```
def f(l):  
    l[0] = 15 ;  
  
i = [2,10.] ;  
f(i) ;  
print ("modified?", i);  
  
j = i ;  
j[0] = 1000 ;  
print ("modified?", i);
```



Conditionals

- if/then/else as in other languages
- block indentation is used!!

```
def relu(x):
    if x > 0:
        return x ;
    else:
        return 0. ;

print (relu(-3)) ;
print (relu(400)) ;

# list comprehension supports appended if
l = [i for i in range(0,10) if i%2==0 ] ;
```



Duck typing

- “**If it walks like a duck and quacks like a duck: it’s a duck!**”
- data types are not predefined but determined by behavior
- behavior → methods: any object with a method `__add__` can be added
- if a method expects a number, it accepts any object with the methods of a number
- special method names:
 - indexing by `__getitem__`, `__setitem__`
 - Initialization by `__init__`;
 - math by `__add__`, `__sub__`, `__mul__`, `__div__`
 - iteration by `__iter__`
 - ...



Duck typing

Example: class `list_like` that allows item access like a list although it is not a list:

```
class list_like:  
    def __init__(self, len):  
        self.len=len ;  
  
    def __getitem__(self, i):  
        return 0 ;  
  
l = list_like(3) ;  
print(l[2]) ;
```



Iterators

- iterator objects are used for sequential access to elements of a composite data type (list, tuple, ...)
- iterator objects have a method `next()` :
 - first element if called for the 1st time
 - next element if called previously
 - error if no next element exists



Python ‘iterables’

- ‘iterable’ = object that has method `__iter__()` which gives back an iterator object
- through this object, iterables can be traversed without knowing their internal structure
 - list
 - string
 - iterators
- Happens automatically in many cases!



Python ‘iterables’

- Automatic treatment of iterables: for loops
`for <variable> in <iterable>`
 - list traversal:

```
for x in [3,4,5]:  
    print (x) ;
```
 - iterator traversal: `range(start, stop)` → returns iterator whose `__iter__` method returns the iterator itself

```
for x in range(0,10):  
    print (x) ;
```



Python ‘iterables’

- Example: list creation from iterables

```
x = list ((1,2,3)) ;  
y = list ([1,2,3]) ;  
z = list (range(0,5)) ;
```



Use of libraries

- import statement at beginning of .py file

```
import package as name ;
```

- important packages:
tensorflow, numpy, math, matplotlib, sys, argparse, os
- import of sub-packages only is possible

```
import matplotlib.pyplot as plt ;  
import numpy.random as npr ;
```



Resources

- Python:
 - free basic online course (need to register):
<https://www.codecademy.com/learn/learn-python-3>
 - reference: www.python.org
- numpy:
 - tutorial: <https://docs.scipy.org/doc/numpy-1.15.1/user/quickstart.html>
 - reference: <https://docs.scipy.org/doc/numpy-1.15.1/reference/>
- tensorflow
 - tutorial(s): https://www.tensorflow.org/guide/low_level_intro
 - reference: www.tensorflow.org



Machine learning

Elements of differential calculus and
applications

Alexander Gepperth, November 2021



Organizational matters

- Python
- Practice sessions: enough places for everyone?
- Practice sessions, LinuxLab accounts and remote connection
- The OR-problem



Today

- Differential calculus for functions of one/several variables
- Applications:
 - Machine learning is a minimization/optimization problem
 - training neural networks: minimize $\mathcal{L}(\vec{w})$ by **gradient descent**



Differential calculus for functions of one variable



Differential calculus for functions of one variable

- Suppose we have a function of a single scalar argument:
$$g(x) : x \in \mathbb{R} \mapsto y \in \mathbb{R}$$
- Example: $g(x) = 3x + 15 + \cos(x)$
- By taking the derivative $g'(x)$ or $\frac{dg}{dx}(x)$, we create a new function
 - based on a few rules ...
 - ... which are relatively simple



Derivatives of simple functions

- For some functions, the derivative is trivial:

$g(x)$	$g'(x)$
x^n	nx^{n-1}
x	1
const.	0
$\cos(x)$	$-\sin(x)$
$\sin(x)$	$\cos(x)$
$e^x, \exp(x)$	$e^x, \exp(x)$
$\ln(x)$	$1/x$

does not contain x

both is possible

Memorize!!



Rules of differential calculus

- Other functions are more complex
 - derivatives are not in table
 - then: how to compute the derivative of complex functions from the derivatives of simple function?
 - **works** if complex function is composed of simple functions by linear combination, multiplication or chaining
 - a single rule for each case!



Derivatives of linear combinations

- If the **complex function**...

- is a sum of simple functions: $(g(x) + h(x))$
- simple function times constant: $(k \cdot g(x))$
- most general case:
linear combination $(k \cdot g(x) + l \cdot h(x) + \dots)$



Derivatives of linear combinations

- If the **complex function**...

- is a sum of simple functions: $(g(x) + h(x))'$
 - simple function times constant: $(k \cdot g(x))'$
 - most general case:
linear combination $(k \cdot g(x) + l \cdot h(x) + \dots)'$

- ... then the derivative is computed as:

$$(k \cdot g(x) + l \cdot h(x) + \dots)' = k \cdot g'(x) + l \cdot h'(x) + \dots$$



Examples



Derivatives of linear combinations

- More than one sum term possible
- “Derivatives of sums is the sum of derivatives”



Derivatives of products

- Sometimes a complex function is a product of simple functions...

$$h(x) = (f(x)(g(x))$$

- In such cases, the **product rule** holds:

$$h'(x) = (f(x)(g(x))' = f'(x)g(x) + g'(x)f(x)$$



Derivatives of chained functions

- Sometimes a complex functions results from the **chaining of simple functions**:

$$h(x) = f(g(x))$$

Then:
$$h'(x) = (f(g(x)))' = f'(g(x))g'(x)$$

- Examples:

$$\left(\sin(x)\right)^2 \quad e^{(x^2)} \quad \sqrt{x^3 + \cos(x) + 1}$$



Examples



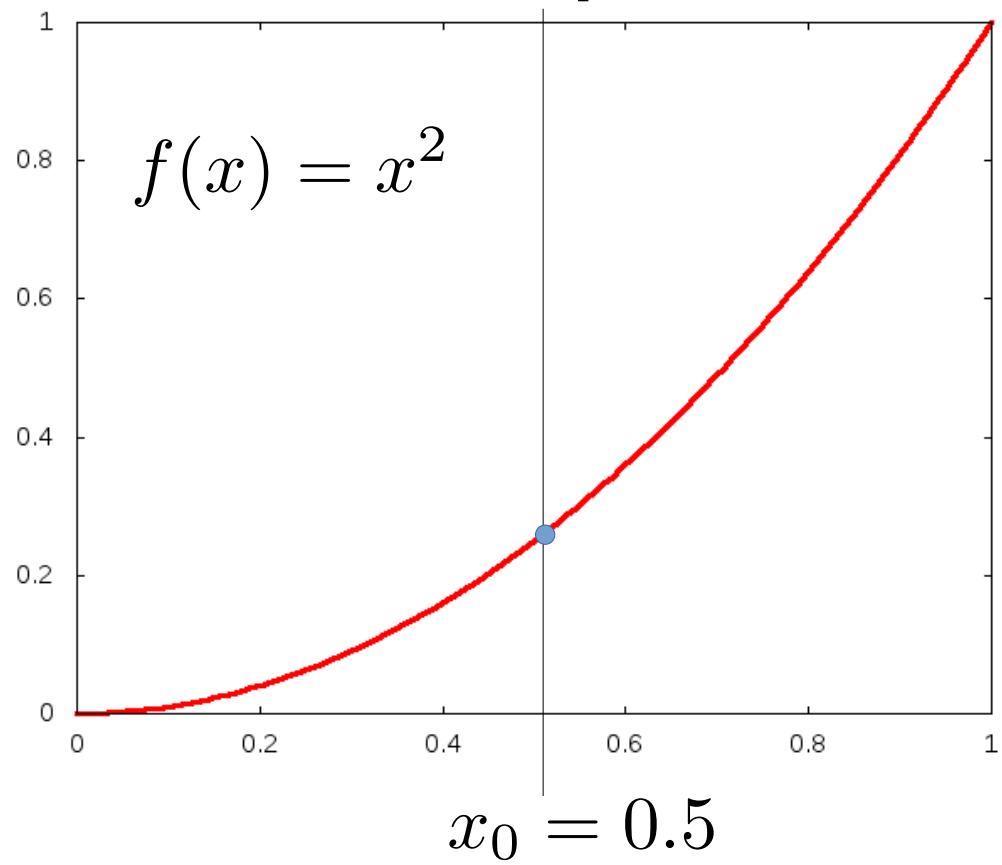
What do derivatives express?

- For a differentiable function $f(x)$, a linear approximation is possible in a small neighbourhood of each point x_0
- This approximating linear function:
 - can be written as $\tilde{f}(x) = a(x - x_0) + f(x_0)$
 - must satisfy $\tilde{f}(x_0) = f(x_0)$
- The slope a of \tilde{f} at x_0 is called the derivative of $f(x)$ at x_0 : $f'(x)\Big|_{x=x_0}$



What do derivatives express?

- Consider the point
 $x_0 = 0.5$

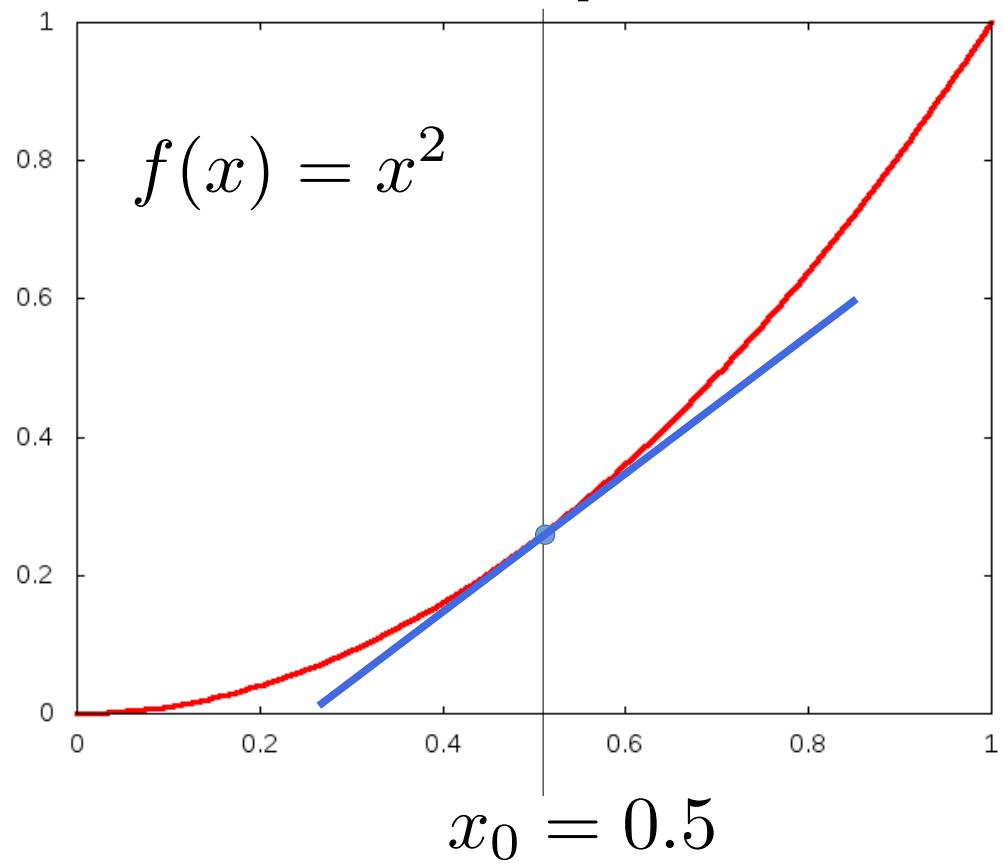




What do derivatives express?

- Consider the point
 $x_0 = 0.5$

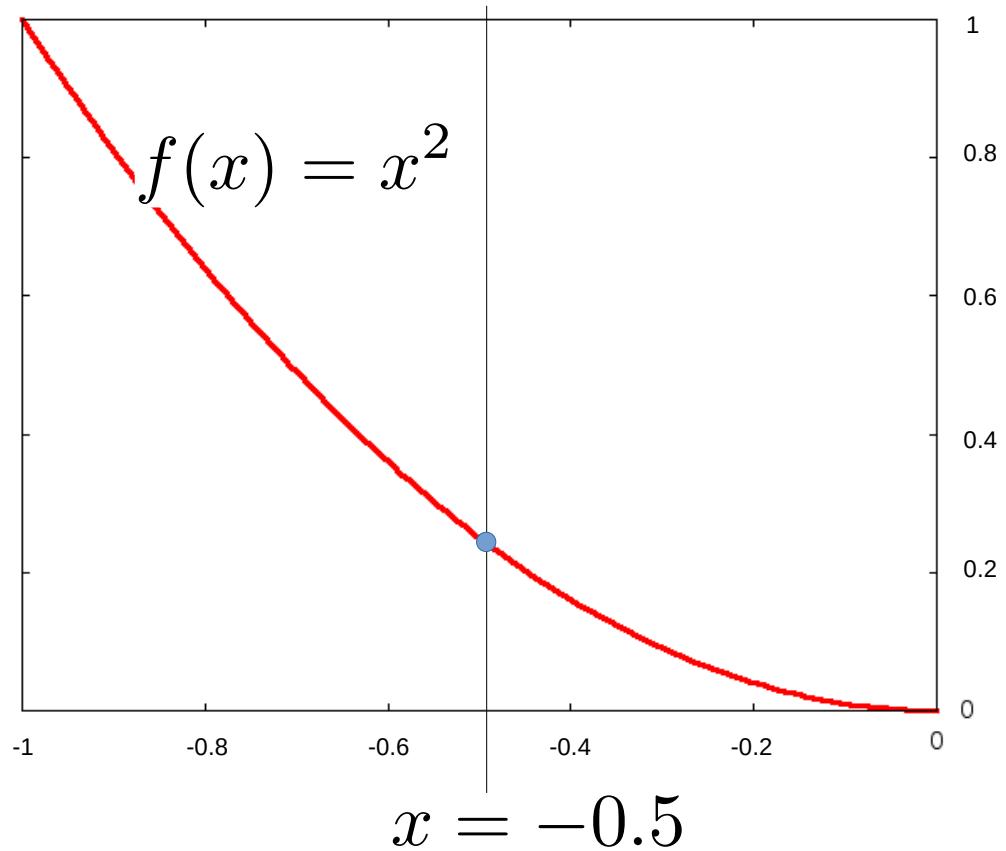
$$\begin{aligned}\tilde{f}(x) &= 2(x - x_0) + 0.25 \\ &= 2x - 0.75\end{aligned}$$





What do derivatives express?

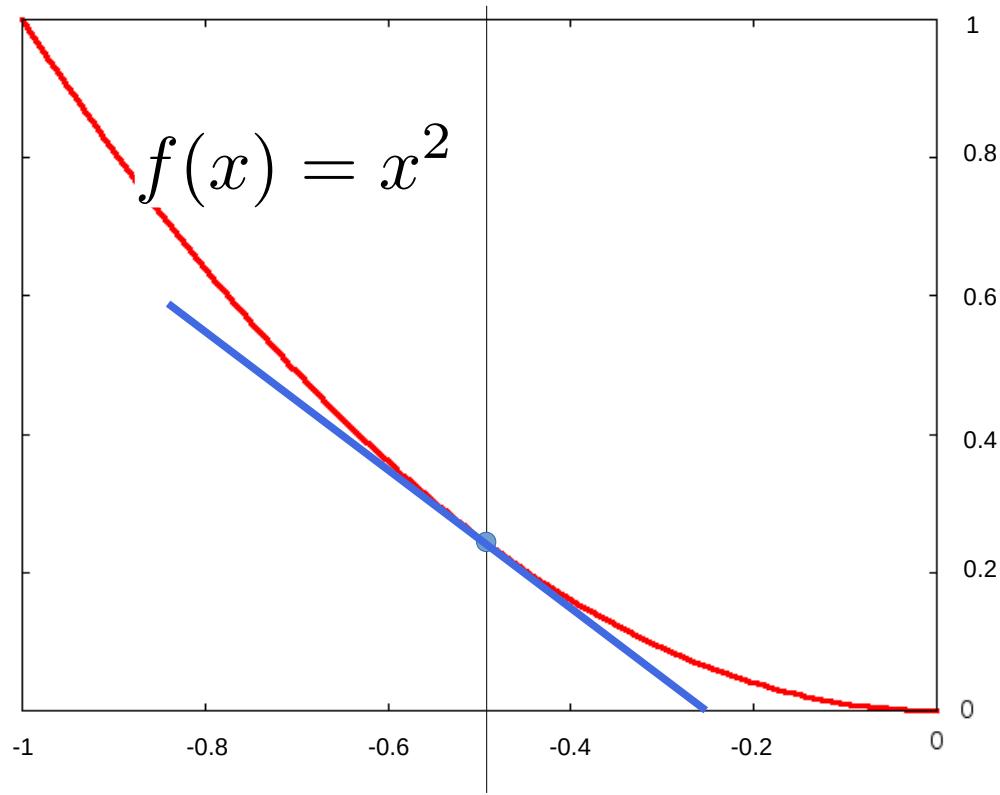
- Consider the point
 $x = -0.5$





What do derivatives express?

- Consider the point
 $x = -0.5$



$$\tilde{f}(x) = -2(x - x_0) + 0.25$$

$$x = -0.5$$



Computing the slope of the approximating function for $f(x) = x^2$

$$\begin{aligned} a(x) &= \lim_{\epsilon \rightarrow 0} \frac{\Delta f}{\epsilon} = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{(x + \epsilon)^2 - x^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{(x^2 + 2x\epsilon + \epsilon^2) - x^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{2x\epsilon + \epsilon^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} 2x + \epsilon = \\ &= 2x = f'(x) \end{aligned}$$

(1)



Computing the slope of the approximating function for $f(x) = x^2$

- Can be done for all simple functions
- Easy for polynomials
- Possible for exp, sin, cos via Taylor expansions

$$\begin{aligned} a(x) &= \lim_{\epsilon \rightarrow 0} \frac{\Delta f}{\epsilon} = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{(x + \epsilon)^2 - x^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{(x^2 + 2x\epsilon + \epsilon^2) - x^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} \frac{2x\epsilon + \epsilon^2}{\epsilon} = \\ &= \lim_{\epsilon \rightarrow 0} 2x + \epsilon = \\ &= 2x \boxed{} = f'(x) \end{aligned}$$

(1)



Differential calculus for functions of several variables



Differential calculus for functions of several variables

- Until now: treatment of functions $f(x)$ of a single variable x
$$f'(x) = \frac{df}{dx}(x)$$
- "Derivative of $f(x)$ w.r.t. x "
- If a function depends on several variables ...
... i.e., $f(x_1, x_2)$ or more generally $f(\vec{x} \in \mathbb{R}^n)$...
- ... we can take the derivate w.r.t any of these variables:

→ **partial** derivatives

$$\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots$$



Differential calculus for functions of several variables

- Notation: $f(\vec{x} \in \mathbb{R}^n) = f(x_1, x_2, \dots, x_n)$
- Example (n=2): $f(x_1, x_2) = x_1 + \sin(x_2)$
- We can take the partial derivatives w.r.t. any variable
 $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}$
- Rules/differences to the 1D case:
 - **the three rules are still applicable**
 - **variables that are not involved in derivative computations are treated as constants!**



DON'T
PANIC

calculus for functions of several variables

- Notation: $f(\vec{x} \in \mathbb{R}^n) = f(x_1, x_2, \dots, x_n)$
- Example (n=2): $f(x_1, x_2) = x_1 + \sin(x_2)$
- We can take the partial derivatives w.r.t. any variable
 $\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n}$
- Rules/differences to the 1D case:
 - **the three rules are still applicable**
 - **variables that are not involved in derivative computations are treated as constants!**



Differential calculus for functions of several variables

- Interpretation of partial derivatives
 - Approximation of $f(\vec{x})$ at point \vec{x}_0 by a linear function $\tilde{f}(\vec{x}) = \langle \vec{a}(\vec{x}_0), \vec{x} - \vec{x}_0 \rangle + f(\vec{x}_0)$
 - The slope \vec{a} is given by the **gradient** of $f(x)$:

$$\vec{a}(\vec{x}_0) = \vec{\nabla} f \Big|_{\vec{x}=\vec{x}_0} = \left(\frac{\partial f}{\partial x_1} \Big|_{\vec{x}=\vec{x}_0}, \dots, \frac{\partial f}{\partial x_n} \Big|_{\vec{x}=\vec{x}_0} \right)$$

- The gradient indicates the direction in which the function's return value grows most strongly
 - **direction of steepest ascent**



Rules for computing p.ds.

- Linear combination: $h(\vec{x}) = kf(\vec{x}) + lg(\vec{x}) + \dots$

$$\frac{\partial}{\partial x_j} \left(kf(\vec{x}) + lg(\vec{x}) + \dots \right) = k \frac{\partial f(\vec{x})}{\partial x_j} + l \frac{\partial g(\vec{x})}{\partial x_j} + \dots$$



Rules for computing p.ds.

- Product rule: $h(\vec{x}) = f(\vec{x})g(\vec{x})$

$$\frac{\partial}{\partial x_j} \left(f(\vec{x})g(\vec{x}) \right) = \frac{\partial f(\vec{x})}{\partial x_j} g(\vec{x}) + \frac{\partial g(\vec{x})}{\partial x_j} f(\vec{x})$$



Rules for computing p.ds.

- Chain rule: $h(\vec{x}) = f(g(\vec{x}))$
- Suppose that:
 - $g(\vec{x}) \in \mathbb{R}$ is a vector-scalar function: $\vec{x} \in \mathbb{R}^n$
 - $f(y) \in \mathbb{R}$ is a scalar-scalar function: $y \in \mathbb{R}$

$$\frac{\partial}{\partial x_j} \left(f(g(\vec{x})) \right) = f'(y) \Big|_{g(\vec{x})} \frac{\partial g}{\partial x_j}$$



Rules for computing p.ds.

- Chain rule: $h(\vec{x}) = f(\vec{g}(\vec{x}))$
- Suppose that:
 - $\vec{g}(\vec{x}) \in \mathbb{R}^m$ is a vector-vector function: $\vec{x} \in \mathbb{R}^n$
 - $f(\vec{y}) \in \mathbb{R}$ is a vector-scalar function: $x \in \mathbb{R}^m$

$$\frac{\partial}{\partial x_k} \left(f(\vec{g}(\vec{x})) \right) = \sum_j^m \frac{\partial f}{\partial y_j} \Big|_{g_j(\vec{x})} \frac{\partial g_j}{\partial x_k}$$



Examples



Applications of the calculus of derivatives: gradient **ascent!**

- Remember: gradient $\vec{\nabla} f$ of a function f indicates the direction of steepest ascent
 - assume we start at a point $\vec{x}(0)$
 - if we want to find a **bigger** value of f : go a small step in the direction of the gradient
$$\vec{x}(i + 1) = \vec{x}(i) + \epsilon \vec{\nabla} f \Big|_{\vec{x}(i)}$$
 - repeat!

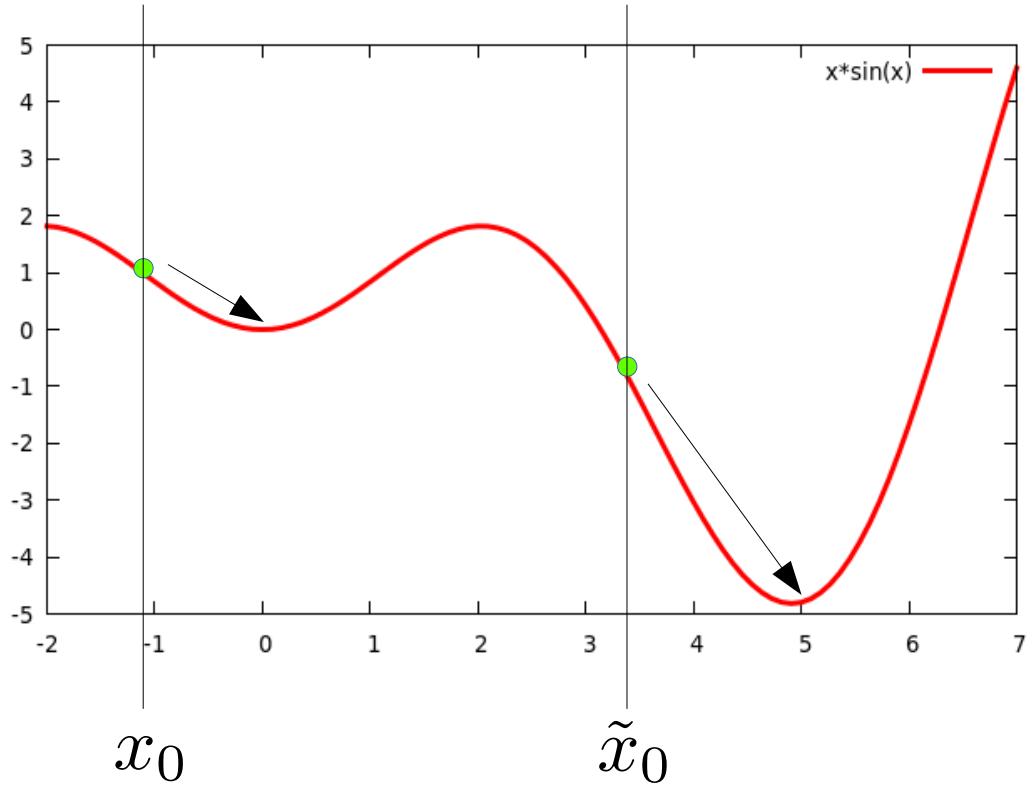


Applications of the calculus of derivatives: gradient **descent!**

- Remember: gradient $\vec{\nabla} f$ of a function f indicates the direction of steepest ascent
 - Assume we start at a point $\vec{x}(0)$
 - if we want to find a **smaller** value of f : go a small step against the direction of the gradient
- $$\vec{x}(i + 1) = \vec{x}(i) - \epsilon \vec{\nabla} f \Big|_{\vec{x}(i)}$$
- repeat!



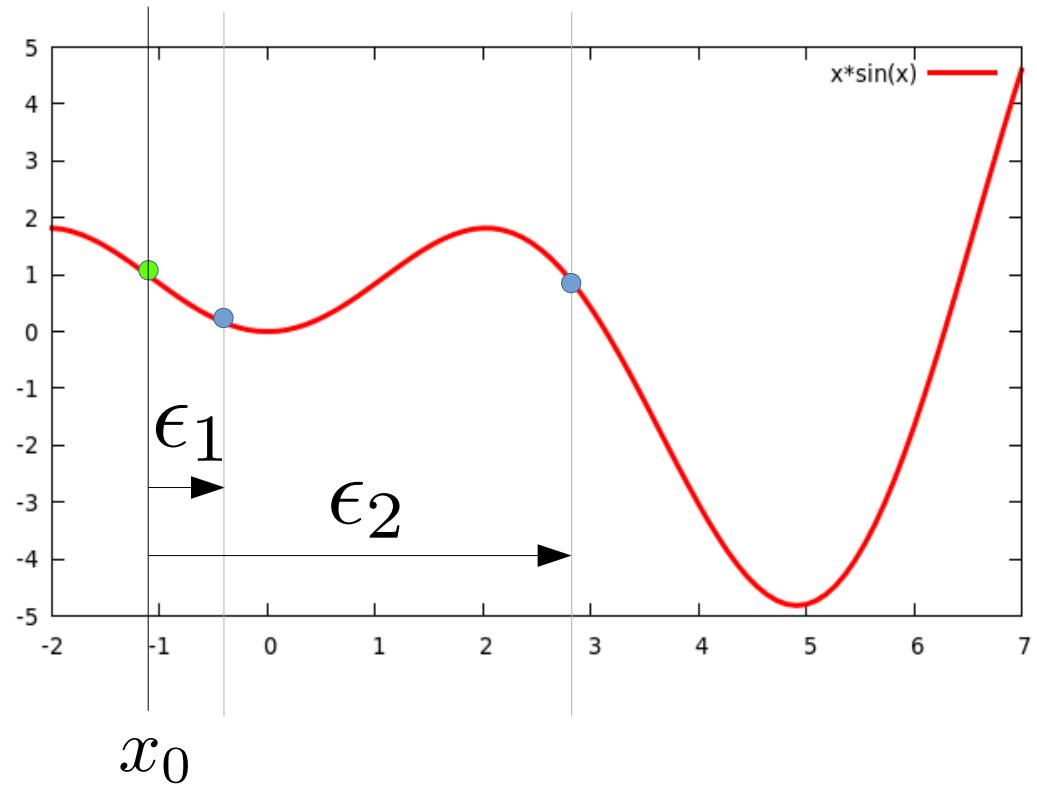
GD: dependency on starting point



- Gradientenabstieg “fällt” in lokale Minima
- Bei komplexen Funktionen mit vielen Minima entscheidet der Startwert
- lokale Minima können viel schlechter sein als das globale!



GD: dependency on step size



- ϵ too small: slow convergence
- ϵ too large: can miss extremal points



GD: toy example

- Let $f(\vec{x}) = x_1^2 + x_2^2$ and $\vec{x}(t=0) = (1, 1)$ with $\epsilon = 0.1$
- Iteration 1:
$$\vec{x}(1) = \vec{x}(0) - \epsilon 2\vec{x}(0) = (1, 1) - 0.1 (2, 2) = (0.8, 0.8)$$
- Iteration 2:
$$\vec{x}(2) = \vec{x}(1) - \epsilon 2\vec{x}(1) = (0.8, 0.8) - 0.1 (1.6, 1.6) = (0.64, 0.64)$$
- Iteration ...
- Test: $f(\vec{x}(2)) > f(\vec{x}(0))$



Summary

- Differential calculus (quick&dirty)
- Tool for optimization: gradient descent allows to minimize/maximize functions
- When training deep neural networks: minimize loss function $\mathcal{L}(\vec{w})$ using gradient descent:

$$\vec{w}(i + 1) = \vec{w}(i) - \epsilon \vec{\nabla} \mathcal{L} \Big|_{\vec{w}(i)}$$



Machine Learning

Foundations of machine learning

The linear classifier

Alexander Gepperth, November 2021



Recap: foundations of machine learning



Mathematical formalization

- Machine learning is about:
 - data: matrix $X \in \mathbb{R}^{N,n}$ with rows $\vec{x}_i \in \mathbb{R}^n$
 - targets (“labels”): matrix $T \in \mathbb{R}^{N,k}$ with row vectors $\vec{t}_i \in \mathbb{R}^k$ (why vectors? later!)
 - we wish to find a “good” model function ...
$$\vec{f}: X \in \mathbb{R}^{N,n}, \vec{w} \in \mathbb{R}^m \mapsto Y \in \mathbb{R}^{N,k}$$
 - ... such that correct target values are obtained





How to represent model function?

- Choose a function family with parameter vector \vec{w}
 $\vec{f}(X, \vec{w})$
- parameters controls behavior of model function
- better parameters \rightarrow better model!
- after functional form is fixed, improving the model
is done by finding a better \vec{w} \rightarrow **LEARNING**



Loss functions

- Learning: adapt parameters in $\vec{f}(X, \vec{w}) \in \mathbb{R}^{N,k}$ in order to find best (or better) model
- “better”: measured by **loss function**
$$\mathcal{L}\left(\vec{f}, X, T, \vec{w}\right) \equiv \mathcal{L}(\vec{w})$$
- so, effectively: learning means adapting the parameters \vec{w} so that $\mathcal{L}(\vec{w})$ is minimized!
- Loss function must be chosen according to the problem



Summary

- For supervised machine learning:
 - we need data samples
 - we need targets
 - we must specify a model function
 - we must specify a loss function



CUT: Q&A

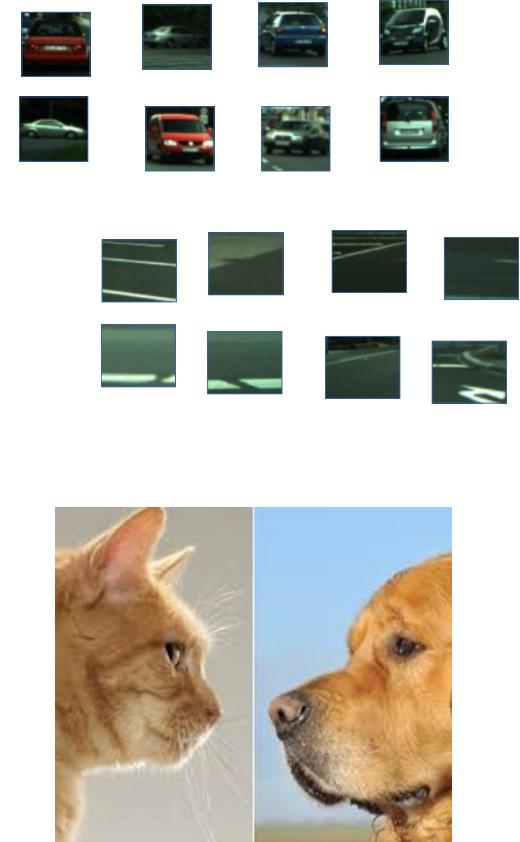


Classification problems



What are classification problems?

- Model should group samples into one out of a finite number of distinct **classes or categories**
- Examples:
 - cars or non-cars
 - cats, dogs or horses
 - hand-written digits (0 – 9)
 - ...





What are classification problems?

- Terminology:
 - binary classification problems: 2 classes
 - multi-class classification problems: >2 classes
 - classifier: model that (learns to) perform classification

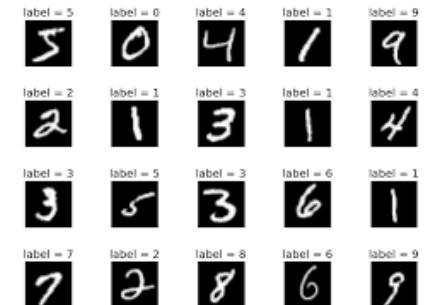
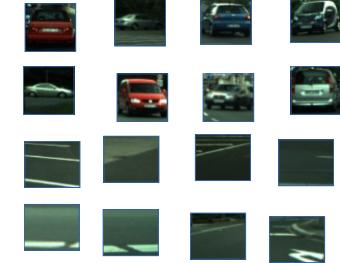


Classification problems: target encoding



Classification problems: Scalar target encoding

- Encode class as a single scalar
- Car classification example:
assign to each image one of two possible values $t_i \in \{1, 2\}$ (binary classification)
- Also possible: multi-class
classification with K classes:
 $t_i \in \{1, \dots, K\}$

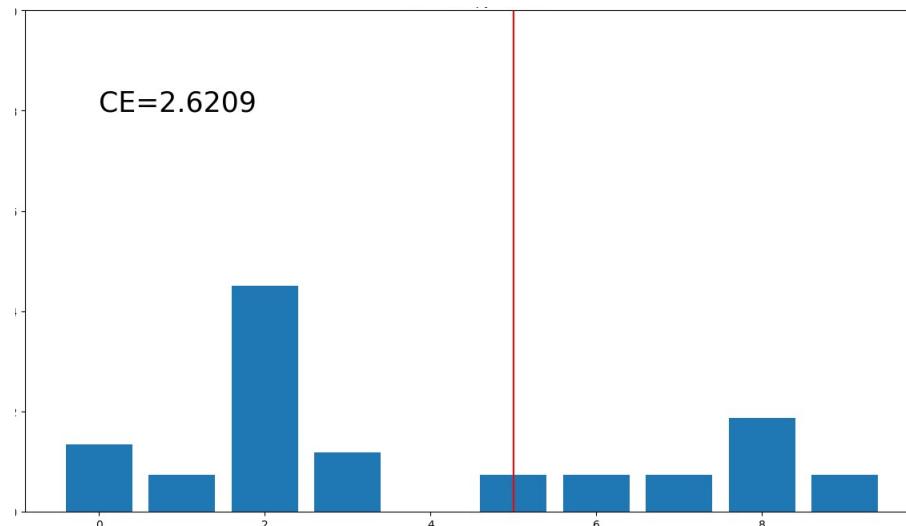




Classification problems: One-hot target encoding

- Encode individual scalar targets $t_i \in \{1, \dots, K\}$ as "one-hot"-vectors: $\rightarrow \vec{t}_i \in \mathbb{R}^k$
- Target matrix form: $T_{ij} = \begin{cases} 1 & \text{if } j = t_i \\ 0 & \text{else} \end{cases}$
- for $K=3$ classes:
 - $t_i = 3 \rightarrow \vec{t}_i = (0, 0, 1,)$
 - $t_i = 1 \rightarrow \vec{t}_i = (1, 0, 0)$

Classification problems: model properties





Classification problems: model properties

- If targets are vectors, model outputs should also be vectors: $\vec{Y} = \vec{f}(X) \in \mathbb{R}^{N,k}$
- Interpretation: largest vector component determines class: $c_i = \operatorname{argmax}_k Y_{ik}$
- advantage: separate confidence for each class!
 $t_i = 3 \rightarrow \vec{t}_i^T = (0, 0, 1,)$ $\vec{y}_i^T = (0.45, 0.05, 0.5)$

Correct classification?

Seite 16



Classification problems: model properties

- If targets are vectors, model outputs should also be vectors: $Y = \vec{f}(X) \in \mathbb{R}^{N,k}$
- Interpretation: largest vector component determines class: $c_i = \operatorname{argmax}_k Y_{ik}$
- advantage: separate confidence for each class!
 $t_i = 3 \rightarrow \vec{t}_i^T = (0, 0, 1,)$ $\vec{y}_i^T = (0.45, 0.05, 0.5)$

YES!

Seite 17



Classification problems: model properties

- If targets are vectors, model outputs should also be vectors: $Y = \vec{f}(X) \in \mathbb{R}^{N,k}$
- Interpretation: largest vector component determines class: $c_i = \operatorname{argmax}_k Y_{ik}$
- advantage: separate confidence for each class!
 $t_i = 3 \rightarrow \vec{t}_i^T = (0, 0, 1,)$ $\vec{y}_i^T = (0.5, 0.05, 0.45)$

NO!



Classification problems: loss functions



Classification error

- For binary/multi-class problems
- Assumes that model output matrix $Y = f(X)$ tries to approximate one-hot-coded target vectors

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^N \left\{ \begin{array}{ll} 1 & \text{argmax}_k Y_{ik} \neq \text{argmax}_k T_{ik} \\ 0 & \text{else} \end{array} \right.$$

- **Example!**
- not differentiable, constant almost everywhere



Cross-entropy

- For binary/multi-class classification problems
- Assumes: $Y_{ik} \in]0, 1]$ and $\sum Y_{ik} = 1 \forall i$
- Cross-entropy for single sample \vec{x} and target \vec{t}

$$\tilde{\mathcal{L}}^{CE} = - \sum_k t_k \log y_k$$

- Cross-entropy for multiple samples X and targets T

$$\mathcal{L}^{CE} = -\frac{1}{N} \sum_{i=1}^N \sum_k T_{ik} \log Y_{ik}$$

matrices



What does cross-entropy express?

- Cross-entropy is determined by log of model output at the place k^* where label = 1:

$$\tilde{\mathcal{L}}^{CE} = - \sum_k t_k \log y_k = - \log y_{k^*}$$

- log is always negative (since model outputs $y_k \leq 1$)
- lowest value is 0, unbounded from above
- if model has nonzero values aside from k^*
→ lower value at k^* → lower cross-entropy



Demo: cross-entropy



The linear classifier model



The linear classifier

- preliminaries: softmax function $S_l(\vec{x}) = \frac{\exp(x_l)}{\sum_k \exp(x_k)}$
 - ensures normalization: $\sum_l S_l(\vec{x}) = 1$
 - positive and bounded: $S_l(\vec{x}) \in [0, 1]$
 - enhances biggest values, suppresses smallest values
- Simple partial derivatives: $\frac{\partial S_i}{\partial x_j} = \delta_{ij} S_i - S_i S_j$



The linear classifier

Vector-to-vector!

- preliminaries: softmax function $S_l(\vec{x}) = \frac{\exp(x_l)}{\sum_k \exp(x_k)}$
 - ensures normalization: $\sum_l S_l(\vec{x}) = 1$
 - positive and bounded: $S_l(\vec{x}) \in [0, 1]$
 - enhances biggest values, suppresses smallest values
- Simple partial derivatives: $\frac{\partial S_i}{\partial x_j} = \delta_{ij} - S_i S_j$



The linear classifier

- Model takes a matrix $X \in \mathbb{R}^{N,n}$ and produces an output matrix $Y = \vec{f}(X) \in \mathbb{R}^{N,k}$ (k : # classes)
- normalization and boundedness ensured by softmax
- Model formula: $\vec{f}(X, W, \vec{b}) = \vec{S}\left(XW + \vec{b}^T\right)$
weights biases
- Cross-entropy loss



The linear classifier

- Model takes a matrix $X \in \mathbb{R}^{N,n}$ and produces an output matrix $Y = \vec{f}(X) \in \mathbb{R}^{N,k}$ (k : # classes)
- normalization and boundedness ensured by softmax **perfect for cross-entropy!!**
- Model formula: $\vec{f}(\vec{X}, W, \vec{b}) = \vec{S}\left(\underset{\text{weights}}{XW} + \underset{\text{biases}}{\vec{b}^T} \right)$
- Cross-entropy loss



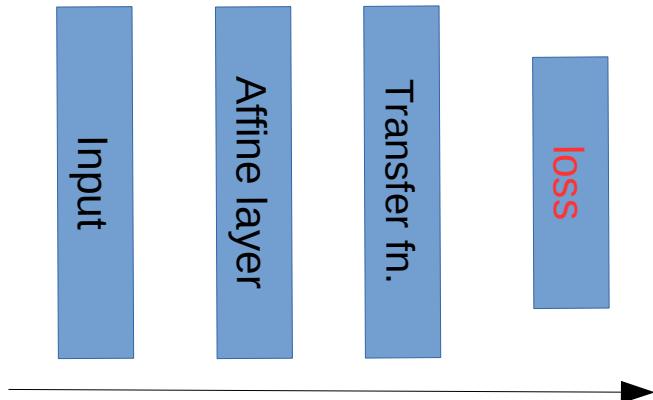
The linear softmax MC classifier as a simple DNN

- Layered model:

Affine layer: $A = XW + \vec{b}^T$

Softmax layer: $Y = \vec{S}(A)$

- Loss computed from model outputs Y





Demo: The linear softmax MC classifier on MNIST

- K=10, loss function: cross-entropy
- model outputs look like probabilities, are they?
- why such a complex model? Why one-hot coding of targets?
 - more information in classifier outputs
 - more free parameters in model, one set per class



How to optimize linear softmax MC?

- Obviously: need to adapt all the weights W and biases \vec{b}
- How?



How to optimize linear softmax MC?

- Obviously: need to adapt all the weights W and biases \vec{b}
- How?
- Gradient descent!!



Machine Learning

numpy, matplotlib and MNIST

Alexander Gepperth, November 2021





numpy

- Package for processing n-dimensional arrays ("tensors")
- ```
import numpy as np ;
```
- important class: `np.ndarray` or simply `np.array` (indexed collection of same-type elements)
- numpy objective: vectorization (calculation with whole arrays instead of elements)



# numpy

- Array creation:
  - Using the `np.array(iterable)` constructor

```
arr = np.array([1,2,3]) ;
```
  - using `np.zeros()` or  
`np.ones()`

```
arr1 = np.zeros([2,2]) ;
arr2 = np.zeros([2,2,3]) ;
arr3 = np.ones([2,4]) ;
print(arr3.shape) ;
```
  - every array has an attribute  
shape that specifies current  
array dimensions



# numpy

- Array 101:
  - Arrays have **shape**: K-tuple of **axis dimensions**
  - **axis**: k-th array dimension,  $0 \leq k < K$ , of defined size  $d_k > 0$
  - **indices** into each axis are **zero-based**: 0 to  $d_k - 1$
  - $K=1$ : 1D Array (vector)  
only a single index required to identify any element!
  - $K=2$ : 2D array: matrix, image
    - $k=0$ : row index
    - $k=1$ : column index
  - $K > 3$ : K-dimensional tensor



# Array indexing

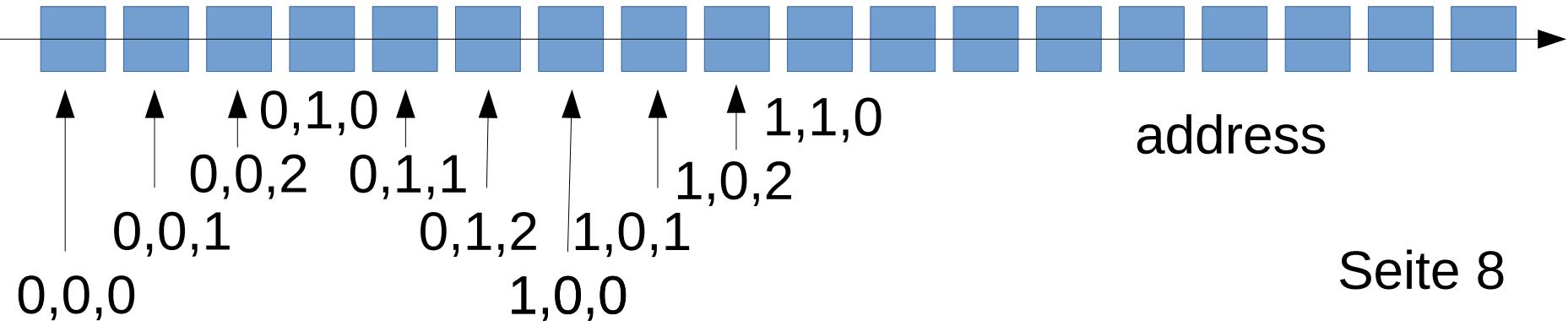
like list indexing,  
but more than  
one index  
possible!

```
a = np.ones([4,5,2]) ;
a[0,0,1] = 2 ;
a[3,4,1] = 5 ;
print (a[4,5,2]) ;
```



# numpy arrays: memory layout

- Arrays elements lie linearly in memory
- Example: 3D-Array (3 axes, shape= [3,2,3])
- when traversing the array memory linearly...
- axis 0 index varies most slowly
- axis 2 index varies most quickly





# numpy reshape

- Knowledge of memory layout is important for changing the shape of an array: `np.reshape`
- does not copy or touch data: just returns a new array header with specified dimensions
- total array size must be unchanged when resizing

```
a = np.ones([4,5,2]) ;
b = np.reshape(40,1) ;
print(b.shape) ;
```

- flatten an array: `np.ravel()`



# Vectorization

- `np.array` supports vectorization = operation on all array elements
- let A, B be Python scalars or objects of type `np.array`:
  - element-wise addition: `C = A+B` or `C = A+55.5`
  - element-wise in-place addition: `A += B`
  - element-wise multiplication (nomatrix multiplication): `C=A*B`
  - element-wise scalar multiplication: `C=2*A`
  - element-wise in-place scalar multiplication: `A *= 2`
  - element-wise exponential: `C=np.exp(A+B)`
- nearly all numpy methods work on whole arrays !



# Array comparisons

- Arrays can be compared element-wise ( $<$ ,  $\leq$ ,  $\equiv$ ,  $\geq$ ,  $>$ ) to arrays/scalars
- result is array of type uint8
- For each element: 1 if comparison yields true, 0 if false → permits counting of elements that

```
import numpy as np ;
x = np.array([1,2,3]) ;
y = np.array([1,2,0]) ;
print ((x==y).sum()) # 2
```

fulfill a condition



# Array slicing

- "cutting out" of parts of an array, representation as own array (like lists!!)
- same syntax as list slicing: start:stop:step
- except that each axis can be sliced independently!
- returns an array that allows to manipulate original data: "view" (different from lists!!)

```
source = np.ones([2, 5])
slice1 = source[0:2:1, 0:4:1]
slice2 = source[:, :, :];
```



# Array slicing

- not all elements of the `start:stop:step` expression need to be given
- generally: if missing, numpy assumes :  
`start = 0, stop = last (included), step = 1`
- except if step is negative:  
`start = last, stop = 0 (included)`
- shorthand: all elements can be missing: notation is “`::`” or “`:`”
- implicit slicing: if there are missing axes, “`:`” is assumed for each missing axis

```
source = np.ones([2, 5])
slice1 = source[0:2:1, 0:4:1]
slice2 = source[:, :]
slice3 = source[1]
```



# Array slicing: examples

- invert row order in 2D image

```
source = np.ones([4, 5])
invRows = source[::-1, :]
```

- copy out even rows from 2D image

```
source = np.ones([4, 5])
evenRows = source[:, ::2]
```



# Array slicing: examples

- 2<sup>nd</sup> column in 2D image

```
source = np.ones([4, 5])
col1 = source[:, 1]
```

- last column in 2D image

```
source = np.ones([4, 5]) ;
lastCol = source[:, -1] ;
```

- last row in 2D image

```
source = np.ones([4, 5]) ;
lastRow = source[-1, :] ;
```



# Array slicing: examples

- invert both axes and take only even positions in both axes

```
source = np.ones([4, 5])
inv2 = source[::-2, ::-2]
```

- inverting of axes possible for any tensor

```
source = np.ones([4, 5, 10])
slice = source[1, :, ::-3]
```



# Fancy indexing

- Normally, array indices are numbers → indexing returns values
- fancy indexing: several indices → indexing returns array of values
- works only if index iterable has elements of type 'int'
- can be done for all axes, combined with slicing (caution!!)
- like slicing, but you can use an enumeration instead of a range
- returns a copy!

```
import numpy as np ;
x = np.zeros([5,4]) ;
x[2,:] = 3 ;
indices = [0,2] ;
print x[indices,:] ;
```



# mask indexing

- Indexing of array with mask: array of True/False-values
- requires uint8-mask, same size and shape as array (simplest case)
- gives 1D-array of all values where mask is True
- uint8-mask can come from array comparisons
- returns copy of original array!

```
import numpy as np ;
x = np.zeros([2,3]) ;
x[1,:] = 3;
maske = (x==3)
print x[maske] ;
x[maske] = 5 ;
```



# numpy copies and views

- Some numpy operations yield array **views**
  - best example: slicing
  - modification of view modifies original array
- most numpy operations return **copies**
  - fancy/mask indexing
  - elementwise operations
- important: numpy default is to make copies, so take care of your memory!

```
source = np.ones([4,5]) ;
slc = source[0:2,:] ;
slc[0,0] = 100 ;
print (source) ;
```



# Reduction

- Application of an operation that computes a scalar from all elements of axis J
- returns a new array with changed values
- “tensor reduction” functions:
  - np.sum
  - np.argmax
  - np.min
  - np.max
  - np.argmin
  - np.argmax
  - ...

$$\tilde{a}_{ik} = \sum_{j=0}^{d_J-1} a_{ijk}$$

A diagram illustrating a summation operation. A large arrow points upwards from below, indicating the summation of elements along axis J. The formula above shows the result  $\tilde{a}_{ik}$  as a sum of elements  $a_{ijk}$  for  $j=0$  to  $d_J-1$ .

```
source = np.ones([4, 5, 6])
contr = np.sum(source, axis=1)
print contr.shape
[4, 6]
```



# Broadcasting

- Usually, element-wise vectorized operations between arrays require equal size of each axis
- exception: one axis size  $> 1$ , other  $= 1$

$$c_{ij} = a_{ij} + b_{ij}$$

```
A = np.ones([4, 5])
B = np.ones([5, 5])
C = A+B
```

$$c_{ij} = a_{ij} + b_{1j}$$

```
A = np.ones([4, 5]) ;
B = np.ones([1, 5]) ;
C = A+B ;
```



# Broadcasting: more examples

- Outer product using broadcasting:

```
A = np.ones([1,5]) ;
B = np.ones([6,1]) ;
C = A*B
print(C.shape) ;
```

- All-vs-all operations in general:

```
A = np.array([1,2,5,2,1]) ;
B = A.T ;
C = (A-B)**2 ;
```



# Random numbers

- submodule `numpy.random`
- important operations:
  - `uniform(a, b, shape)`
  - `normal(mean, sd, shape)`
  - `shuffle(x)` – in-place random shuffling along axis 0!

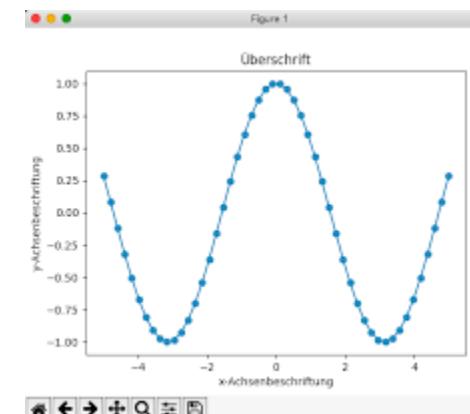
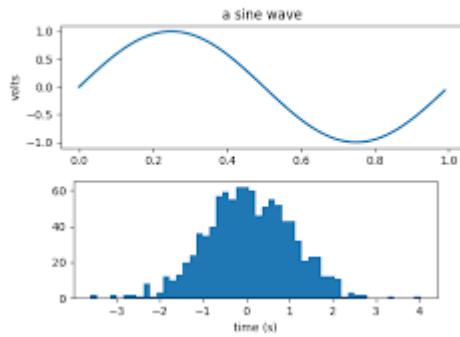
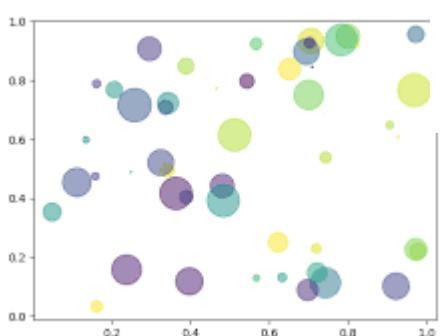
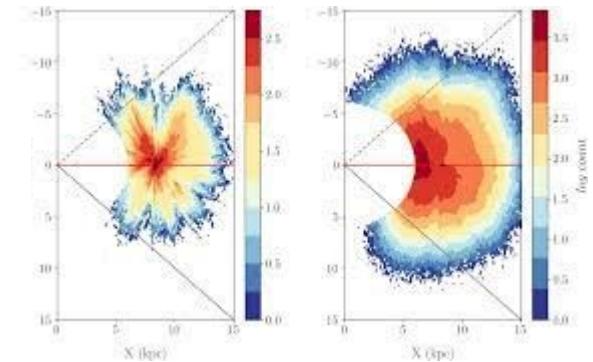
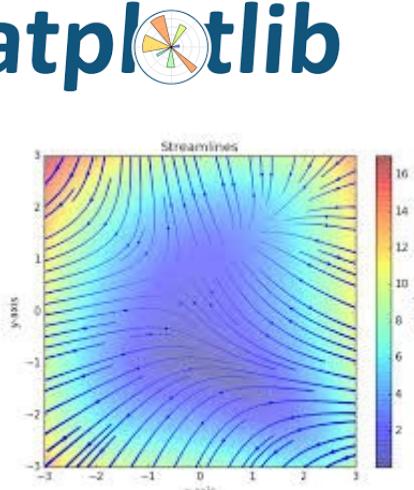
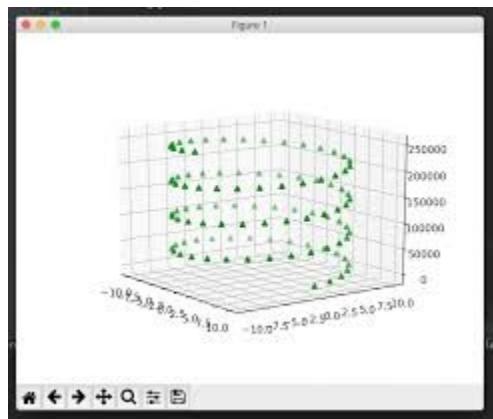
```
import numpy as np ;
import numpy.random as npr;
x = np.array([1,2,3,4]) ;
npr.shuffle(x) ;
print (x) # z.B. [2,4,3,1]
```



# CUT: Q&A



# matplotlib





# matplotlib

- Scientific 2D plotting library that can display anything in any conceivable way
  - images
  - curves
  - point sets
  - bar/pie charts, histograms, ...
- highly customizable!



# matplotlib

- Important principles
  - matplotlib has an object-oriented and an imperative API
  - here: imperative API (easier but slower) via pyplot submodule
  - In general: when a plot command is invoked it draws nothing.  
→ need to call `plt.show()`

```
import matplotlib.pyplot as plt ;
xval = [1,2,3,4,5] ;
plt.plot(xval,[x**2 for x in xval])
plt.show()
```



# Matplotlib: some plot commands

- Scatter plot (“points in 2D”): `scatter(x, y)`
- bar chart (“bars in 2D”): `bar(x, y)`
- histogram (visualization of a distribution):  
`hist(x, . . .)`
- Display 2D image: `imshow(img)`
- Plot graph with lines: `plot(x, y)`
- all part of `pyplot` submodule



# Matplotlib generics

- All plot commands work with **iterables**
- can plot lists, tuples, arrays, ...
- **anything that is iterable and contains numbers**

```
import matplotlib.pyplot as plt ;
plt.scatter((1,2,3), (4,4,4)) ;
plt.plot([1,2,3], [4,4,4]) ;
plt.bar(np.arange(0,3), [4,4,4]) ;
plt.show() ;
```



# Displaying multiple plots

- Create a figure instance

```
fig = plt.figure() ;
```

- Specify target axis in figure by

```
fig.add_subplot(rows,cols,index,
title="") ;
```

- Plot/draw/repeat/...

- plt.show()



# Demo

- Histogram plot
- Curve plot
- Scatter plot
- Bar plot
- Multiple plots in one figure
- Saving plots



# CUT: Q&A



# MNIST

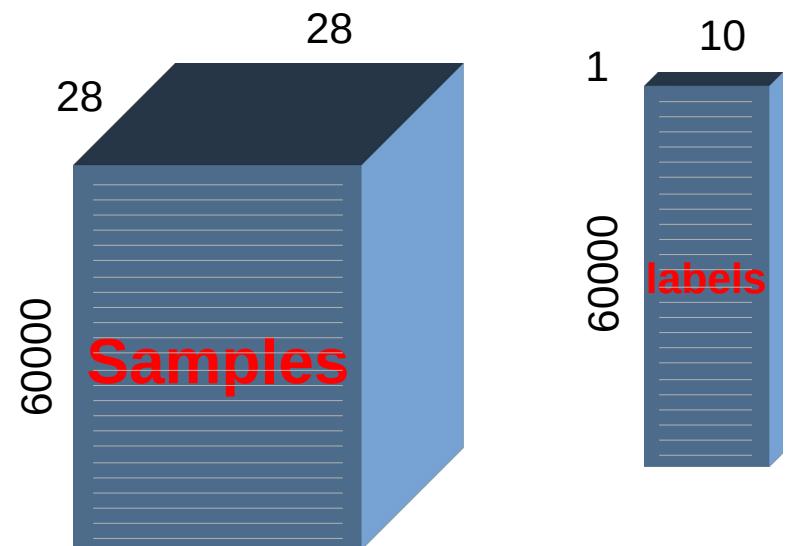
- Collection of images of handwritten digits (0-9)
- each image (or: **sample**) of dimension  $28 \times 28 = 784$
- 60.000 samples for training, 10.000 for testing
- for each sample: digit class (or: **label**) available

1 1 5 4 3  
7 5 3 5 3  
5 5 9 0 6  
3 5 2 0 0



# MNIST data format

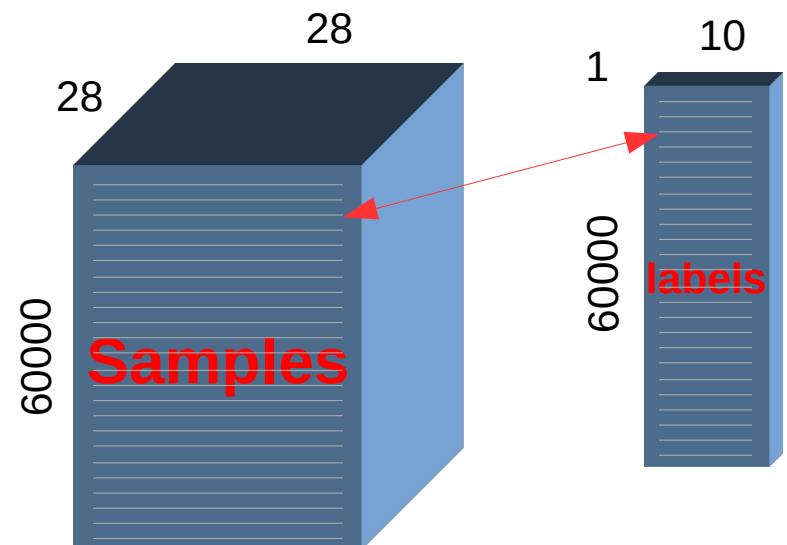
- Good example of data representation in machine learning and data science
- represented as 2 numpy arrays
  - Samples:
    - 2D-Arrays of shape: [60.000,28,28]
    - each sample (“row”) a 28x28-Tensor
    - each pixel of a sample is a float32 between 0 (black) and 1 (white)
  - Labels:
    - 2D-Arrays (numpy) of shape [60.000,10]
    - labels are tied to sample with same index 0 (“row”)
    - each row: vector with 1 at position of class, 0 elsewhere
    - e.g.: [0,0,0,1,0,0,0,0,0,0] → class 3!





# MNIST data format

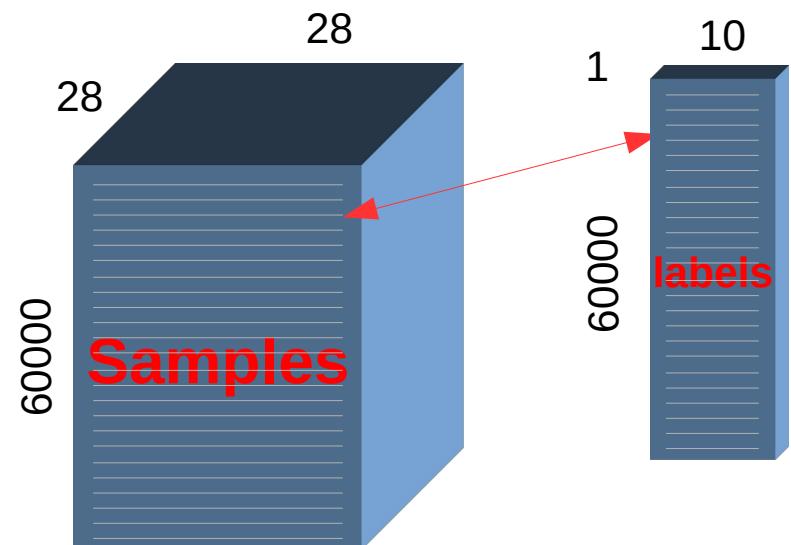
- Good example of data representation in machine learning and data science
- represented as 2 numpy arrays
  - Samples:
    - 2D-Arrays of shape: [60.000,28,28]
    - each sample (“row”) a 28x28-Tensor
    - each pixel of a sample is a float32 between 0 (black) and 1 (white)
  - Labels:
    - 2D-Arrays (numpy) of shape [60.000,10]
    - labels are tied to sample with same index 0 (“row”)
    - each row: vector with 1 at position of class, 0 elsewhere
    - e.g.: [0,0,0,1,0,0,0,0,0,0] → class 3!





# MNIST data format

- Good example of data representation in machine learning and data science
- represented as several numpy arrays
  - Samples:
    - 2D-Arrays of shape: [60.000,28,28]
    - each sample (“row”) a 28x28-Tensor
    - each pixel of a sample is a float32 between 0 (black) and 1 (white)
  - Labels:
    - 2D-Arrays (numpy) of shape [60.000,10]
    - labels are tied to sample with same index 0 (“row”)
    - each row: vector with 1 at position of class, 0 elsewhere
    - e.g.: [0,0,0,1,0,0,0,0,0,0] → class 3!

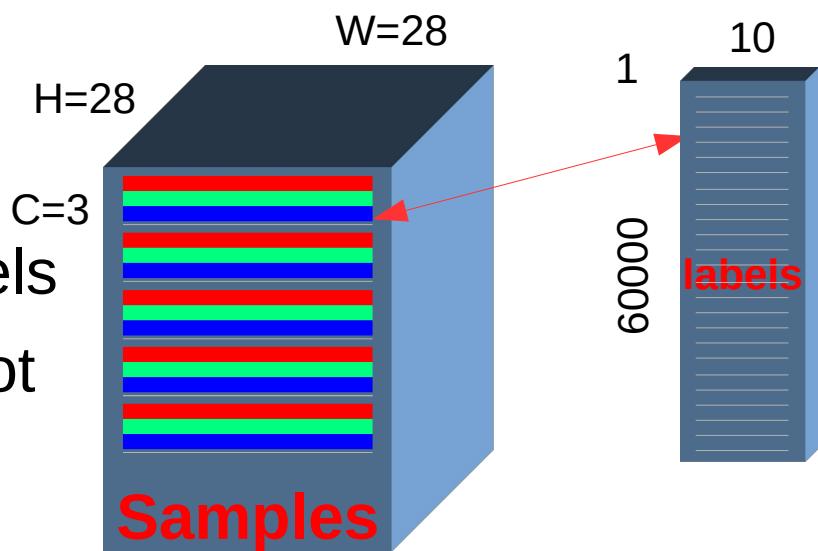


**Dimensions when samples are RGB color images??**



# NHWC data format

- Most common data representation in machine learning and data science
- Stack of N color images of dimensions H,W,C
- **Number, Height, Width, Channels**
- Label encoding is always one-hot independently of data encoding





# Basic data science exercises with MNIST

- 1) verify number and dimension of samples
- 2) copy out and visualize sample k
- 3) what are minimal and maximal pixel values?
- 4) delete (set to 0) upper half in each sample and visualize
- 5) count samples of class X?
- 6) how many classes are actually present in the data?
- 7) copy out all samples of class X into new array
- 8) copy out all samples of classes X,Y,Z... into new array
- 9) copy out K randomly selected samples



# 1) number/ dimension of MNIST samples

```
import matplotlib.pyplot as plt, numpy as np ;
...load mnist...
print ("number of samples:",traind.shape[0]) ;
print ("y dimension of samples:",traind.shape[1]) ;
print ("x dimension of samples:",traind.shape[2]) ;
```



## 2) copy out and visualize

- Display n<sup>th</sup> sample from training data:

```
import matplotlib.pyplot as plt, numpy as np ;
...load mnist...
n = 576 ;
sampleN = traind[n,:,:] ;
plt.imshow(sampleN) ;
plt.show() ;
```



# 3) minimal or maximal pixel values

Smallest/biggest pixel values in all images, anywhere...

```
print("minmax=", traind.min(), traind.max()) ;
```

Smallest/biggest pixel values in single image

```
print("minmax in sample n=",
 traind[n, :, :].min(), traind[n].max())
```

Smallest/biggest pixel values in all images at position 0,0

```
print("minmax at 0,0=", traind[:, 0, 0].min(),
 traind[:, 0, 0].max()) ;
```



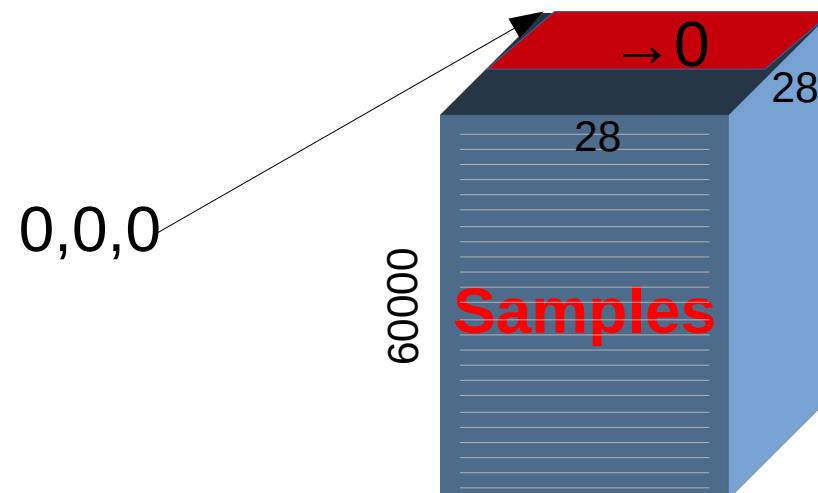
# 4) set upper half of images to 0

Treat all images

```
traind[:, 0:15, :] = 0 ; # set upper half to zero
```

Treat only image 100

```
traind[100, 0:15, :] = 0 ; # just treat sample 100
```

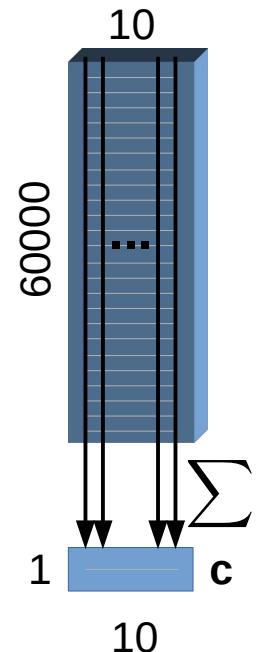




# 5) Count samples of class X

- need to analyze label tensor!
- ansatz: sum along axis 0 → **reduction** gives Tensor **c** of shape [10]. Entry i corresponds to sum over column i
  - counts ones in column i
  - therefore: counts instances of class i
- caution: label tensor needs to be cast to float32 before reduction (else: numerical overflow)

```
trainl32 = trainl.astype("float32") ;
c = trainl32.sum(axis=0) ;
print("#=", c) ;
```

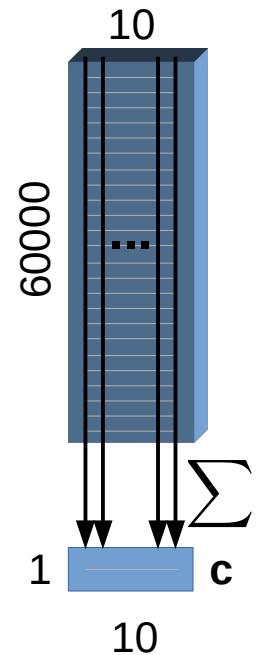




# 6) Count classes that are actually present

- starting point: Tensor **c** aus 1c)
- **c** contains number of samples in each class
- class k not present  $\longleftrightarrow c_k = 0$
- Ansatz: count positions where  $c_k > 0$ 
  - element-wise comparison, gives mask of ints
  - sum up mask

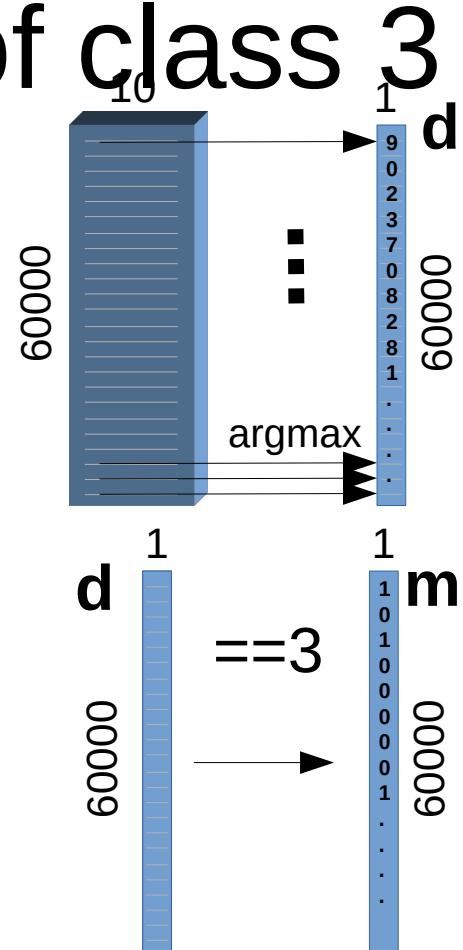
```
trainl32 = trainl.astype("float32") ;
c = trainl32.sum(axis=0) ;
mask = (c > 0).sum() ;
```





# 7) copy out all samples of class 3

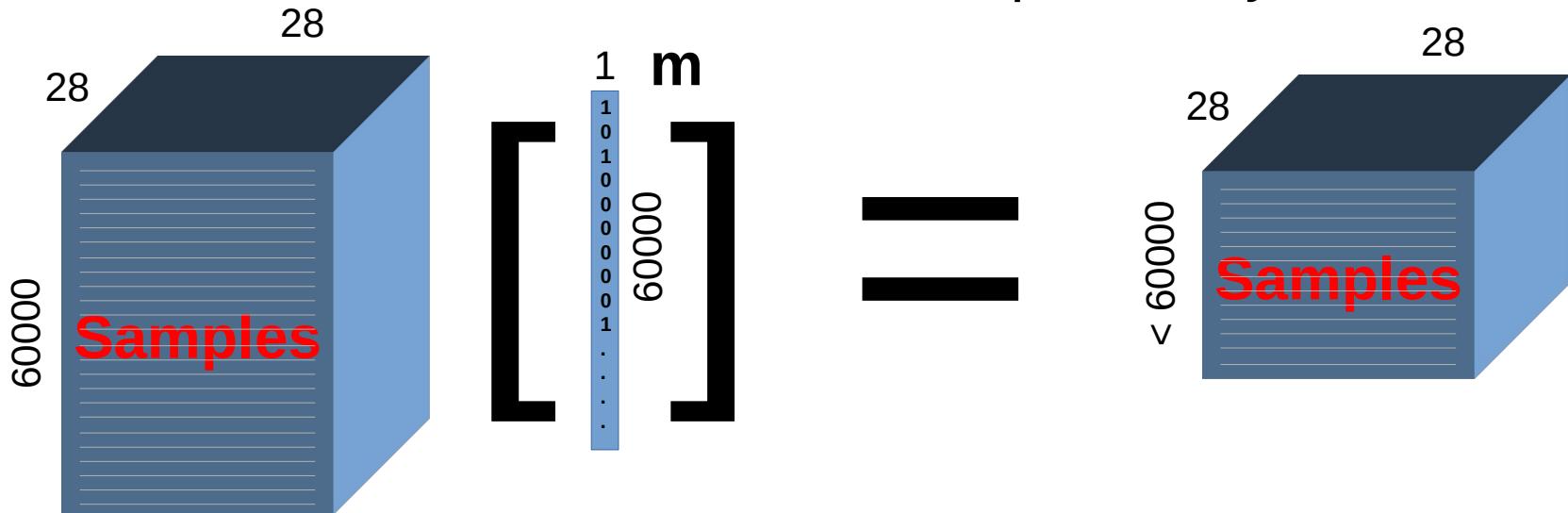
- need to consider label tensor
  - step 1: convert one-hot into single number
    - by using argmax reduction → tensor  $\mathbf{d}$
    - → each entry of reduced tensor: 0-9
  - step 2: create a mask  $\mathbf{m}_3$ 
    - 1 at positions where  $d_k == 3$
    - 0 elsewhere
  - step 3: use mask  $\mathbf{m}_3$  to copy out samples (mask indexing)





# 7) copy out all samples of class 3

- mask **m3** can be an index into samples array



```
d = trainl.argmax(axis = 1) ;
m3 = (d == 3) ;
copyOut = traind[m, :]
```



## 8) copy out samples of classes X,Y,Z

- obtain masks  $m_X, m_Y, m_Z$  for classes X,Y,Z by repeating step 7
- sum up masks to combine them into  $m$
- use this for mask indexing into samples array

```
d = train1.argmax(axis = 1) ;
mX = (d == X) ;
mY = (d == Y) ;
mZ = (d == Z) ;
m = mX + mY + mZ ;
copyOut = traind[m, :]
```



# 9) copy out K random samples

- step 1: create an array **r** of K random entries ranging from 0 to the number of samples

```
nrSamples = traind.shape[0] ;
K = 100 ;
r = np.random.randint(0, nrSamples, shape=[K]) ;
```

- step 2: use **r** for fancy indexing

```
copyOut = traind[r] ;
```

1

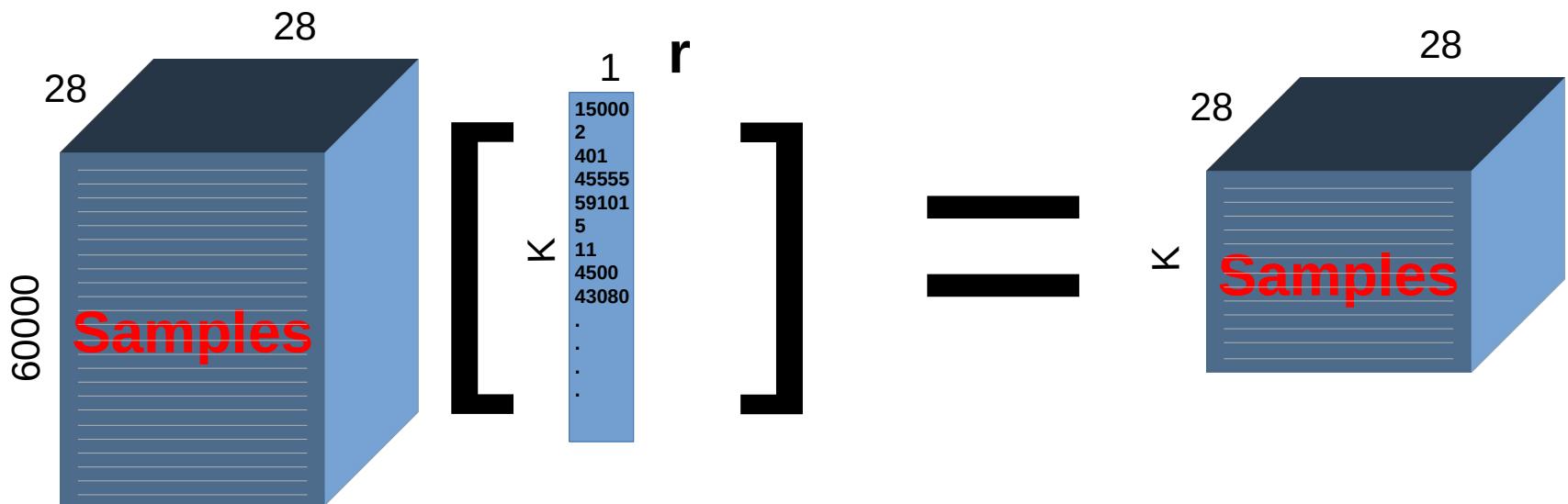
**r**

15000  
2  
401  
45555  
59101  
5  
11  
4500  
43080  
. . .

↖



# 9) copy out K random samples





# Machine Learning

Training of ML models

Alexander Gepperth, December 2021



# Today

- Recap: machine learning fundamentals
- Gradient Computation for Linear Classifier, ctd.
- Review of numpy code for training a LC
- Practical aspects of LC/ML training:
  - Gradient Descent and Stochastic Gradient Descent
  - Test sets and generalization



# Recap: foundations of machine learning



# Mathematical formalization

- Machine learning is about:
  - data: matrix  $X \in \mathbb{R}^{N,n}$  with rows  $\vec{x}_i \in \mathbb{R}^n$
  - targets (“labels”): matrix  $T \in \mathbb{R}^{N,k}$  with row vectors  $\vec{t}_i \in \mathbb{R}^k$  (why vectors? later!)
  - we wish to find a “good” model function ...  
$$\vec{f}: X \in \mathbb{R}^{N,n}, \vec{w} \in \mathbb{R}^m \mapsto Y \in \mathbb{R}^{N,k}$$
  - ... such that correct target values are obtained





# How to represent model function?

- Choose a function family with parameter vector  $\vec{w}$   
 $\vec{f}(X, \vec{w})$
- parameters controls behavior of model function
- better parameters  $\rightarrow$  better model!
- after functional form is fixed, improving the model is done by finding a better  $\vec{w}$   $\rightarrow$  **LEARNING**
- **Linear Classifier:**  $\vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b})$   




# Loss functions

- “better”: measured by **loss function**

$$\mathcal{L} \left( Y = \vec{f}(X), T \right) \equiv \mathcal{L}(\vec{w})$$

- Cross-entropy loss (for classification with K classes):

$$\mathcal{L}(Y, T) = -\frac{1}{N} \sum_n^N \sum_k^K \log Y_{nk} T_{nk}$$



# Learning

- Learning: adapt parameters in  $Y = \vec{f}(X, \vec{w}) \in \mathbb{R}^{N,k}$  in order to find best (or better) model
- so, effectively: adapting the model parameters  $\vec{w}$  so that  $\mathcal{L}\left(Y = \vec{f}(X), T\right) \equiv \mathcal{L}(\vec{w})$  is minimized!
- Minimization of loss through **gradient descent**:

$$\vec{w}(t+1) = \vec{w}(t) - \epsilon \vec{\nabla} \mathcal{L} \Big|_{\vec{w}(t)}$$

- Linear classifier:

$$W_{ab}(t+1) = W_{ab}(t) - \epsilon \frac{\partial L}{\partial W_{ab}} \quad b_c(t+1) = b_c(t) - \epsilon \frac{\partial L}{\partial b_c}$$



# Summary

- For supervised machine learning:
  - we need data samples
  - we need targets
  - we must specify a model function
  - we must specify a loss function



# CUT: Q&A



# Gradient computation for linear classifier, ctd.



# Gradient computation for linear classifier, ctd.

- Known:

$$\frac{\partial L}{\partial b_c} = -\frac{1}{N} \sum_n (T_{nc} - Y_{nc})$$

- Now: compute  $\frac{\partial L}{\partial W_{ab}}$



# CUT: Q&A



# Numpy code for training LC

- Live discussion of code
- Demonstration of LC training on MNIST



# Numpy code for training LC

- Implementing model gradients:

$$\frac{\partial L}{\partial b_c} = -\frac{1}{N} \sum_n (T_{nc} - Y_{nc})$$

and

$$\frac{\partial L}{\partial W_{ab}} = \frac{1}{N} \sum_n (T_{na} - Y_{na}) X_{nb}$$



# Cut: Q&A



# Overfitting and generalization



# Overfitting and generalization

- When training a LC on MNIST, we find:
  - train loss gets **smaller** (better) if less samples are used for training
  - explanation: smaller number of samples can be learned more easily
  - can trained LC better recognize digits?



# Overfitting and generalization

- When training a LC on MNIST, we find:
  - train loss gets **smaller** (better) if less samples are used for training
  - explanation: smaller number of samples can be learned more easily
  - can trained LC better recognize digits?  
**not at all, overfitting!**



# Overfitting

- Overfitting occurs when training data do not sufficiently describe the “true” problem
  - too few samples (see demo)
  - not representative enough
- Very small training loss, but problem not solved!

training data



true problem





# Overfitting

- Overfitting occurs when training data do not sufficiently describe the “true” problem
  - too few samples (see demo)
  - not representative enough
- Very small training loss, but problem not solved!

training data      true problem

1 5 4 3  
5 3 5 3

1 1 5 4 3  
7 5 3 5 3  
5 5 9 0 6  
3 5 2 0 0



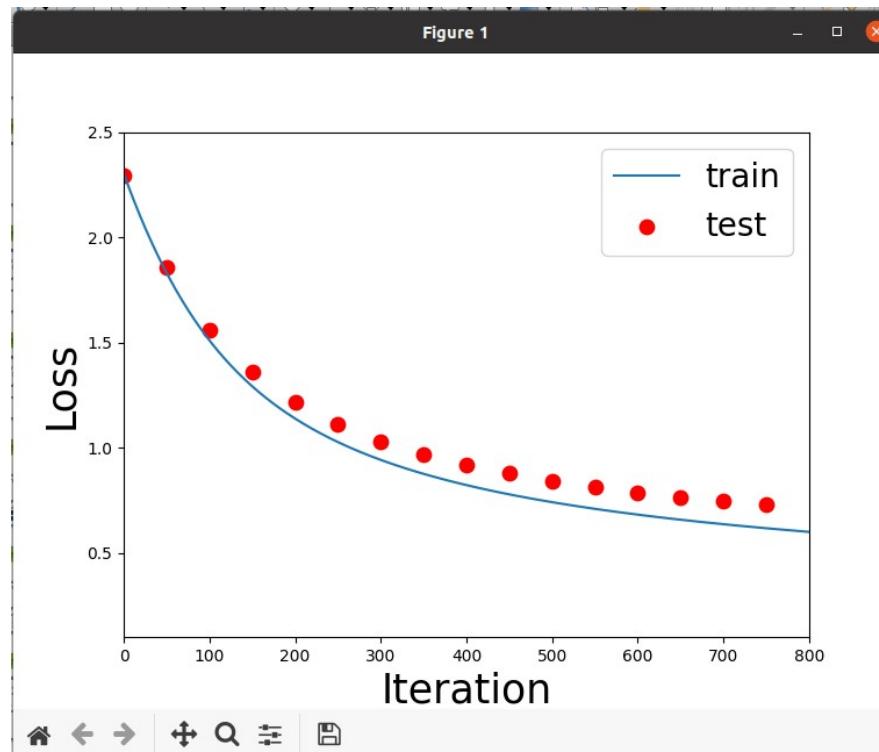
# How to detect overfitting

- We need an independent set of samples that represents the “true problem” in addition to the training data (*train set*)
- Often called *test set* or *test data*
- Procedure: perform learning using train set, measure loss on test set
- Overfitting if losses disagree!



# No overfitting (demo)

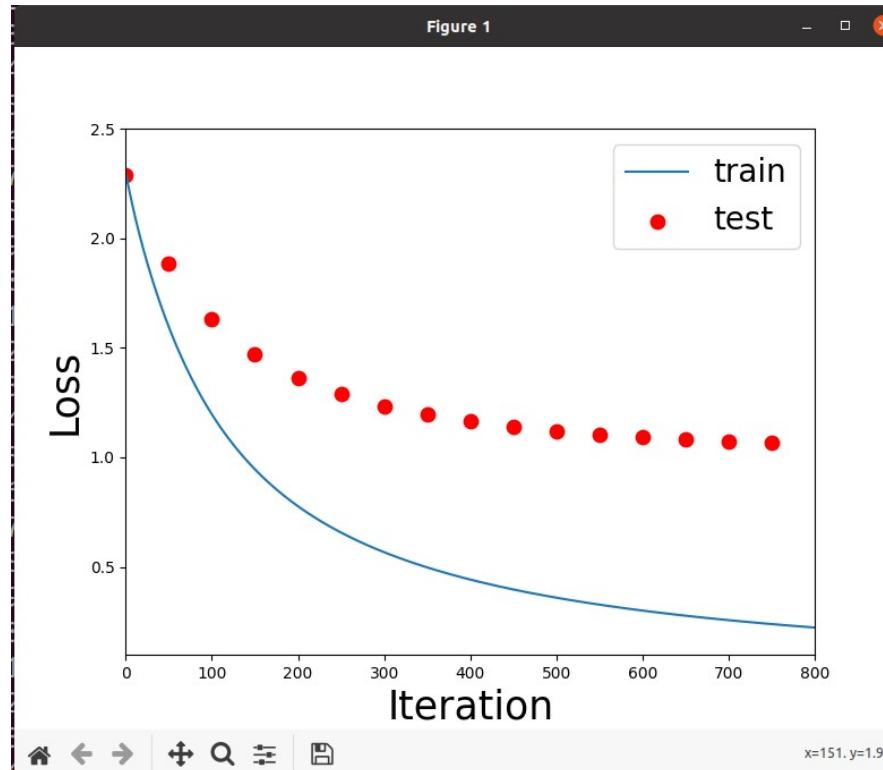
- Training with 2000 random MNIST samples





# Overfitting (demo)

- Training with 100 random MNIST samples





# Cut: Q&A



# Stochastic gradient descent (SGD)



# Stochastic gradient descent (SGD)

- Context: loss functions are usually averages over single-sample losses
- Example: cross-entropy

$$\tilde{\mathcal{L}}(\vec{y}, \vec{t}) = - \sum_k \log y_k t_k$$

$$\mathcal{L}(Y, T) = \frac{1}{N} \sum_n \tilde{\mathcal{L}}(\vec{y}_n, \vec{t}_n) = -\frac{1}{N} \sum_n \sum_k \log Y_{nk} T_{nk}$$



# Stochastic gradient descent (SGD)

- Consequence: gradients are averages of single-sample gradients as well

$$\mathcal{L}(Y, T) = \frac{1}{N} \sum_n \tilde{\mathcal{L}}(\vec{y}_n, \vec{t}_n)$$

$$\frac{\partial \mathcal{L}}{\partial W_{ab}} = \frac{1}{N} \sum_n \frac{\partial \tilde{\mathcal{L}}}{\partial W_{ab}}$$



# Stochastic gradient descent (SGD)

- Consequence: gradients are averages of single-sample gradients as well

$$\begin{aligned}\mathcal{L}(Y, T) &= \frac{1}{N} \sum_n \tilde{\mathcal{L}}(\vec{y}_n, \vec{t}_n) \\ \frac{\partial \mathcal{L}}{\partial W_{ab}} &= \frac{1}{N} \sum_n \frac{\partial \tilde{\mathcal{L}}}{\partial W_{ab}} \\ &= -\frac{1}{N} \sum_n (T_{na} - Y_{na}) X_{nb}\end{aligned}$$



# Stochastic gradient descent (SGD)

- Consequence: gradients are averages of single-sample gradients as well

$$\mathcal{L}(Y, T) = \frac{1}{N} \sum_n \tilde{\mathcal{L}}(\vec{y}_n, \vec{t}_n)$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial W_{ab}} &= \frac{1}{N} \sum_n \frac{\partial \tilde{\mathcal{L}}}{\partial W_{ab}} \\ &= -\frac{1}{N} \sum_n (T_{na} - Y_{na}) X_{nb}\end{aligned}$$

large matrix!



# Stochastic gradient descent (SGD)

- Computing gradients over all data samples is memory-intensive
- All data samples need to be stored in memory → sometimes very difficult
- Conclusion: true gradient descent is problematic



# Stochastic gradient descent (SGD)

- Solution: stochastic gradient descent (SGD): approximates full gradient by gradient over mini-batch  $\mathcal{B}$  of size  $B$

$$\frac{\partial \mathcal{L}}{\partial W_{ab}} = \frac{1}{N} \sum_n^N \frac{\partial \tilde{\mathcal{L}}}{\partial W_{ab}} \approx \frac{1}{B} \sum_{n \in \mathcal{B}} \frac{\partial \tilde{\mathcal{L}}}{\partial W_{ab}}$$

- Perform gradient descent with this gradient!

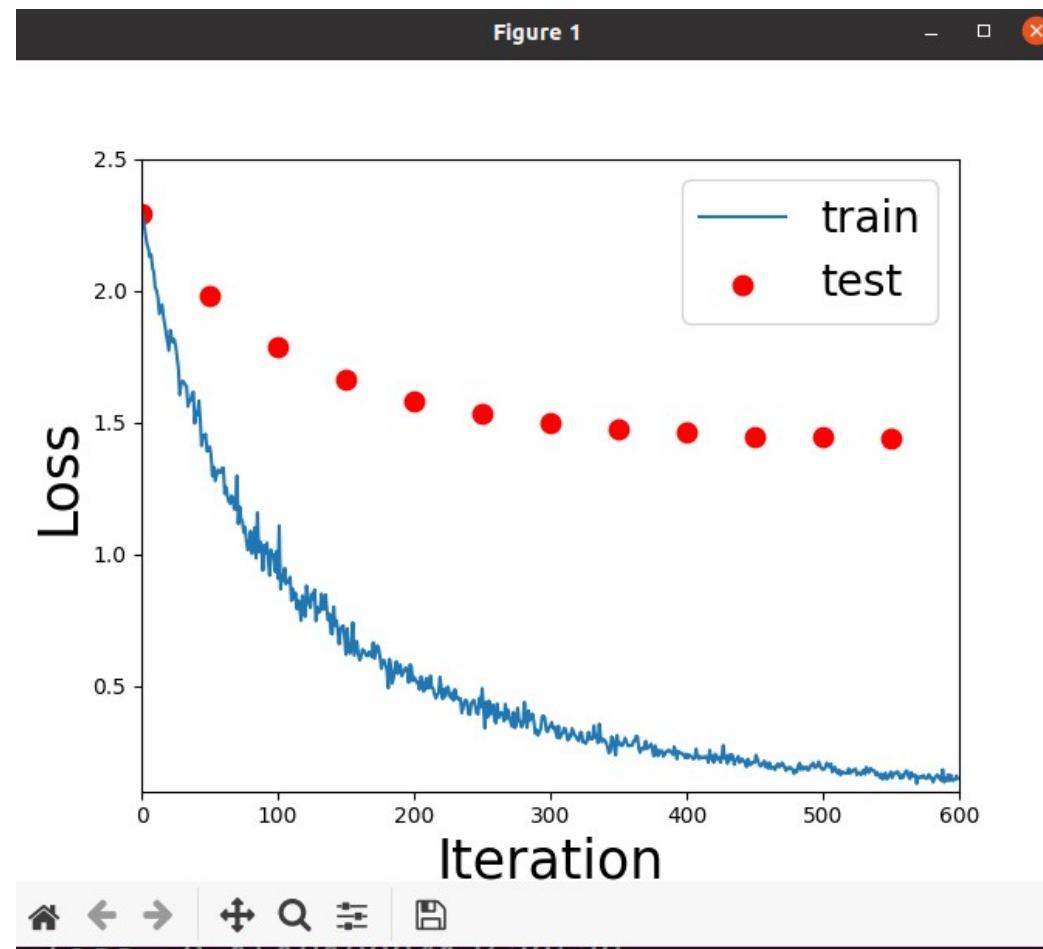


# SGD: algorithm

- Parameters: learning rate  $\epsilon$ ,  $max\_it$ , batch size B
- Repeat for  $max\_it$  iterations:
  - draw a random batch from the training data
  - compute loss gradients on mini-batch
  - update parameters by gradient descent
  - compute test loss
- Consequence: noisy gradient descent!



# SGD: demo





# Cut: Q&A



# Machine learning

## TensorFlow

Alexander Gepperth, December 2021



# Today

- Optimizers and stochastic gradient descent
- Introduction to TensorFlow 2
- tf.keras



# Tensorflow2



# What is TensorFlow2?

- Framework for accelerating complex tensor computations
  - over multiple machines
  - with multiples CPUs, GPUs (or TPUs/similar)
  - details hidden from user
- Basic entity: tensors (multi-dimensional arrays like in numpy)
- "Open Source" software with large community
- Works with any GPU architecture (but CUDA preferred)



# TF basics

- Current version: 2.7.0
- Very active development
- "Like numpy, but with GPU support"
  - vectorization is the goal
  - same, or similar, treatment of tensors/arrays
  - supports automated gradient computations!!



# TF principles

- Data (all with `shape` and `dtype` attributes):
  - immutable tensors/constants
  - mutable variables
- Same “always copy” policy as numpy
- TF is strongly typed, strict checking
- Data types for tensors: `tf.uint8`, `tf.int16`,  
`tf.int32`, `tf.int64`, `tf.float16`,  
`tf.float32`, `tf.float64`, `tf.complex`, ...



# Tensors

- Class: `tf.Tensor`, data resides on GPU!
- Immutable multi-dimensional array
- Creation: `tf.zeros`, `tf.ones`,
- Important attributes: `shape`, `dtype`
- Important member function: `numpy()`
- Idea:
  - placeholder for computation result on GPU
  - produced by all TF functions



# Tensor creation

- Creating immutable tensors:
  - `tf.zeros`
  - `tf.ones`
  - `tf.range`
  - `tf.linspace`
  - `tf.constant`



## tf.constant

- Like variables, but immutable
- Can be created explicitly:  
`tf.constant(iterable,  
 dtype, shape)`
- Other data types, like numpy arrays, are automatically promoted to `tf.constant`



# Variables

- Class: `tf.Variable`, data resides on GPU
- Creation:  
`tf.Variable(dtype, shape, initial_value)`
- Idea:
  - Placeholder for **mutable** data on GPU
  - Supports in-place operations
  - Network weights, biases, etc...
- `tf.assign_add`, `tf_assign_mul` for manipulation



# Tensor comparison

- Comparison function produce bool tensors
- Analogous to numpy operators
  - `tf.greater`
  - `tf.less`
  - `tf.equal`



# Important TF functions

- Tensor manipulation:
  - `tf.reshape`, `tf.expand_dims`
  - `tf.cast`
  - `tf.slice(slicing)`
  - `tf.gather` (fancy indexing)
  - `tf.boolean_mask` (mask indexing)
  - `tf.where` (mask multiplexing)



# Tensor creation/manipulation demo



# Important TF functions

- Reductions (using axis parameter!):
  - `tf.reduce_sum`
  - `tf.reduce_max`
  - `tf.reduce_min`
  - `tf.argmax`
  - `tf.reduce_mean`



# Important TF functions

- Math:
  - `tf.math.sin`
  - `tf.math.cos`
  - `tf.matmul`
  - `tf.math.log`



# Important TF functions

- Neural networks:
  - `tf.nn.relu`
  - `tf.nn.softmax`
  - `tf.nn.softmax_cross_entropy_with_logits`



# TF functions/tensors demo



# Automated gradients

- `tf.GradientTape` context manager
- Keeps track of all tensors within its scope
- Afterwards, can compute a gradient of/w.r.t all tensors in its scope
- Works for arbitrary expression composed of **TF functions only!**
- **Demo**



# Demo



# Cut: Q&A



# tf.keras

- Keras: sub-package of TF2
- Goal: simple, portable construction and training of DNNs
- Especially: gradient computation and training hidden from user
- But: details **can** be controlled by experts



# tf.keras and the shortest linear classifier code EVER

```
model = tf.keras.Sequential() ;
model.add(tf.keras.layers.Dense(10)) ;
model.add(tf.keras.layers.Softmax()) ;
model.compile(optimizer="sgd",
loss = tf.keras.losses.CategoricalCrossentropy()) ;
model.fit(trainind, trainl, epochs=1) ;
```



# Eager mode and graph mode

- Default in TF2 is eager mode:
  - commands are executed immediately
  - commands are executed as specified
- In particular: “always copy” behavior of numpy is strictly adhered to



# Memory consumption in eager mode

- How many tensors are created?

```
def compute_something(x) :
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```



# Memory consumption in eager mode

- How many tensors are created? **3**

```
def compute_something(x) :
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```



# Memory consumption in eager mode

- How many copies are required?

```
def compute_something(x) :
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```



# Memory consumption in eager mode

- How many copies are required? **0**

```
def compute_something(x) :
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```

- **Reason: in-place modification of x would be sufficient**



# Memory consumption in eager mode

- How many copies are required? **0**

```
def compute_something(x) :
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```

- **Memory consumption is a huge problem in TF → TF graph mode!**



# TF2 graph mode

- Applied to functions using a decorator

```
@tf.function
def compute_something(x):
 y = x + 1 ;
 x = 2 * x + y ;
 return x ;
```

- **TF2 performs a graph analysis of the code and optimizes it → 0 copies!** Seite 29



# TF2 graph mode

- Complex construct, has restrictions:
  - applicable to whole functions only
  - function can only use TF functions and tensors, no switching between np and tf
- **Potentially huge gain w.r.t. GPU memory**



# TF2 graph mode

- Examples: OK

```
@tf.function
def compute_something(A, x) :
 x = tf.expand_dims(x, 0) ;
 return tf.matmul(A, x) ;
```



# TF2 graph mode

- Examples: not OK (but no error)

```
@tf.function
def compute_something(x, y):
 a = x.numpy().max() ;
 b = y.numpy().max() ;
 return a + b ;
```



# Machine Learning

Lecture 7: deep learning

Alexander Gepperth, December 2021



# Notation for DNNs



# Single-sample notation

- Instead of using matrices for samples, targets, model outputs, ...
- ... formulate everything in terms of one-row matrices (aka vectors):
  - $\vec{x}$  instead of  $X$
  - $\vec{y}$  instead of  $Y$
  - $\vec{t}$  instead of  $T$



# Single-sample notation examples

- Linear softmax MC:

$$\vec{y} = \vec{f}(\vec{x}, W, \vec{b}) = \vec{S}\left(W\vec{x} + \vec{b}\right)$$

- Cross-Entropy:

$$\tilde{\mathcal{L}}(\vec{y}, \vec{t}) = - \sum_j \log(y_j) t_j$$



# Multi-sample notation

- Machine learning involves treatment of
  - multiple data samples
  - multiple target values
  - multiple layer activities (later today...)



# Multi-sample notation

- Treat all quantities as matrices
- Matrix rows represent individual samples, targets, activities, model outputs
- Functions (e.g., model, softmax or adding a vector) are applied row-wise



# Multi-Sample notation examples

- Linear classifier:  $Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$
- Cross entropy:  $\mathcal{L} = -\frac{1}{N} \sum_{i=1}^N \sum_j \log(Y_{ij}) T_{ij}$
- Sometimes, we mix notations for clarity:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^N \sum_j \log(f_j(\vec{x}_i)) T_{ij}$$



# Deep Learning



# Generalizing linear softmax MC to a DNN classifier

- Linear classifier:

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

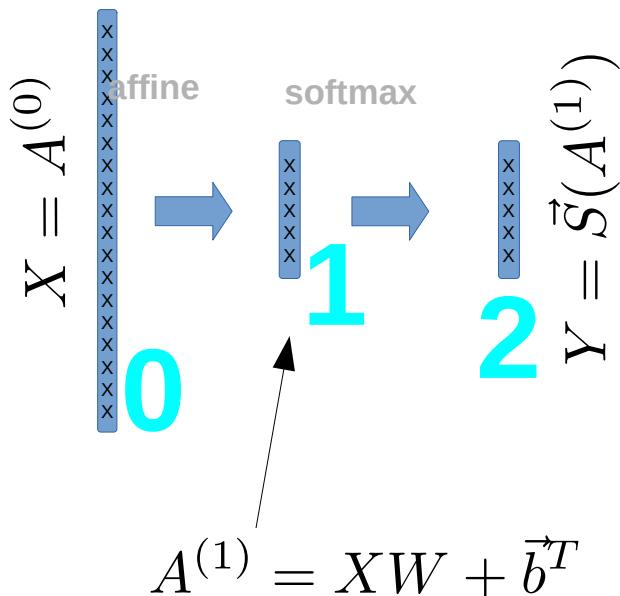
- Two successive transformations:
  - affine Transformation (**affine layer**)
  - softmax transformation (**softmax layer**)



# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

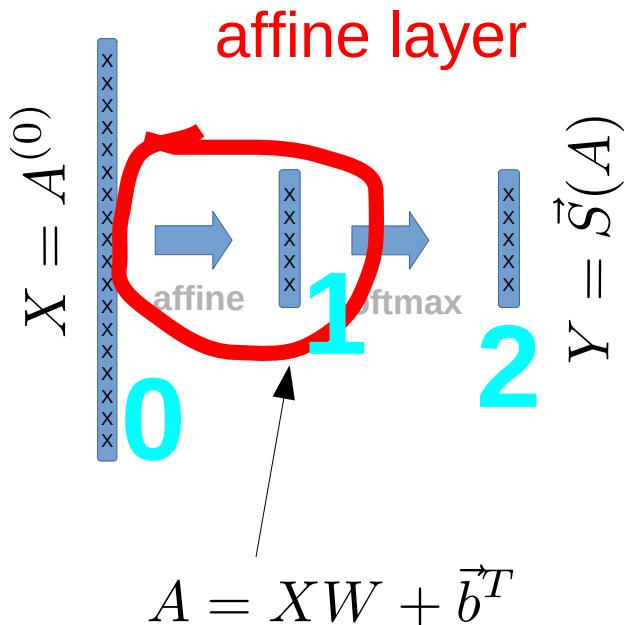




# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

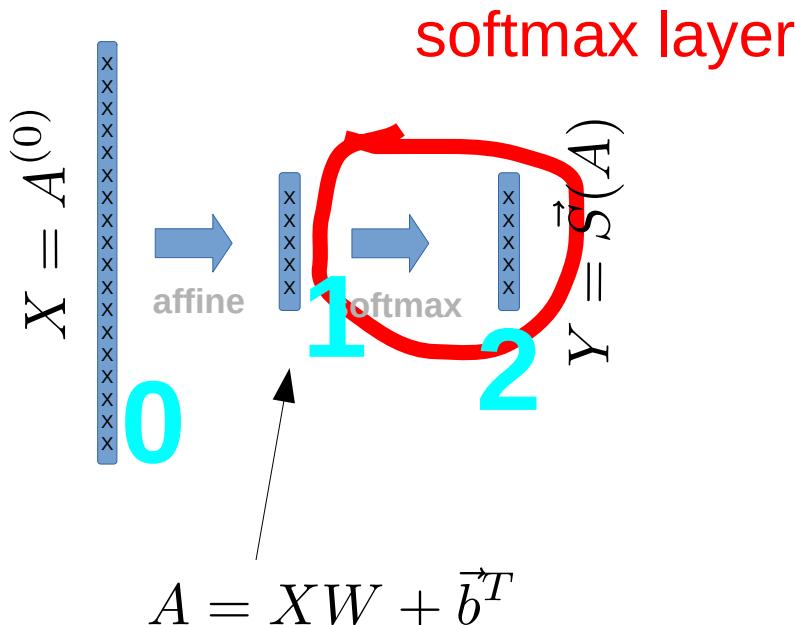




# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

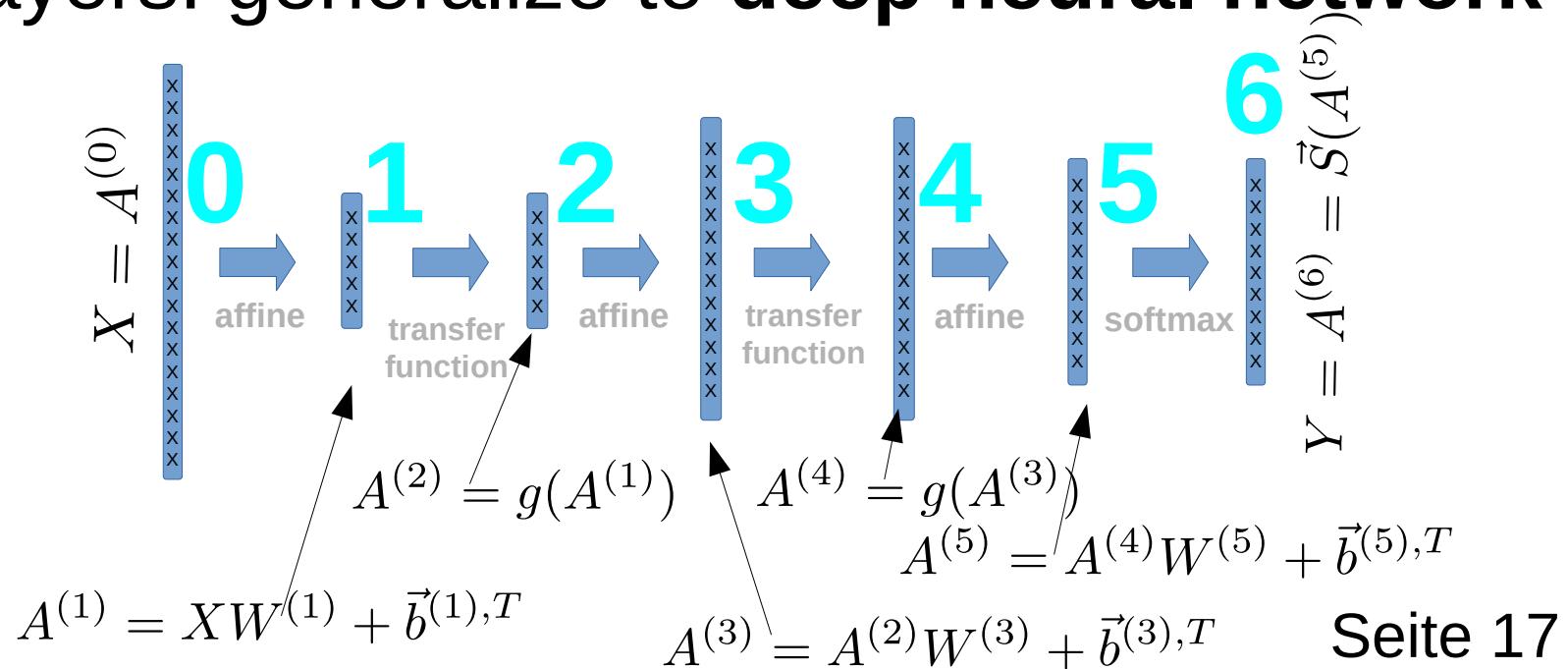
$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$





# Generalizing linear softmax MC to a DNN classifier

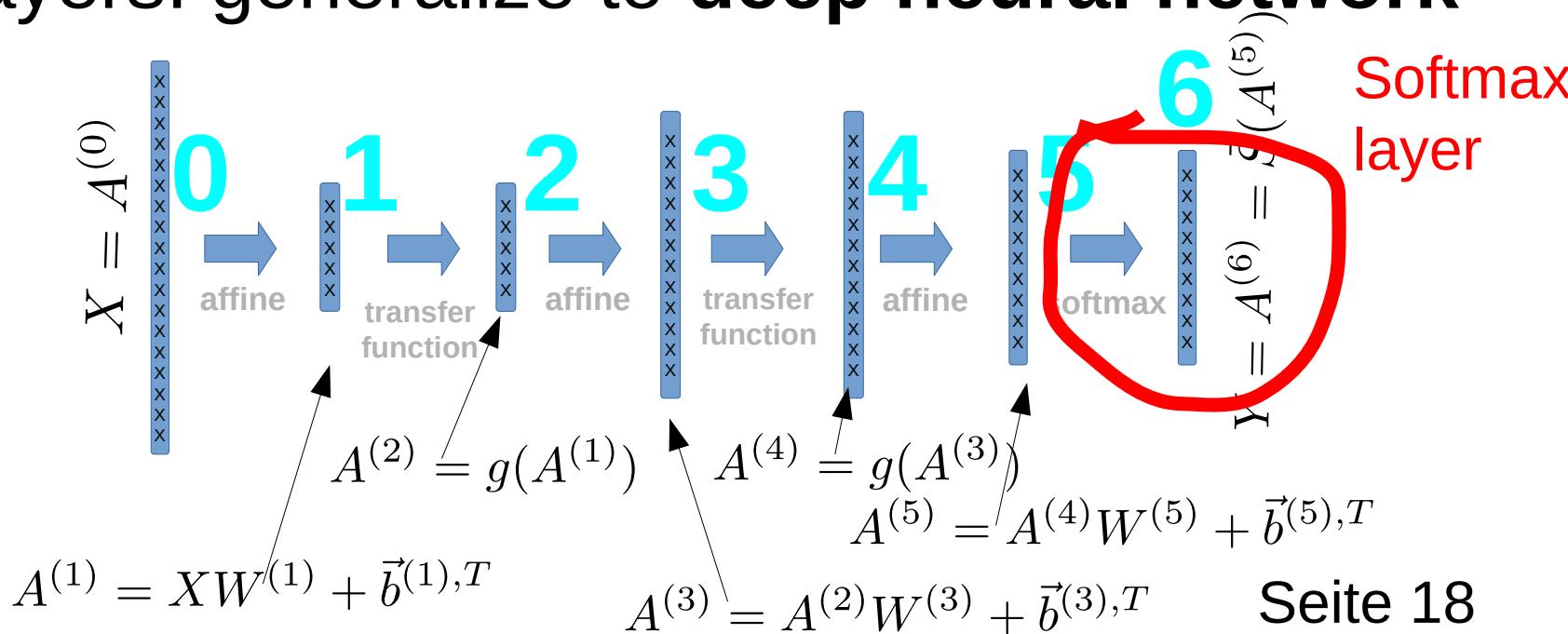
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

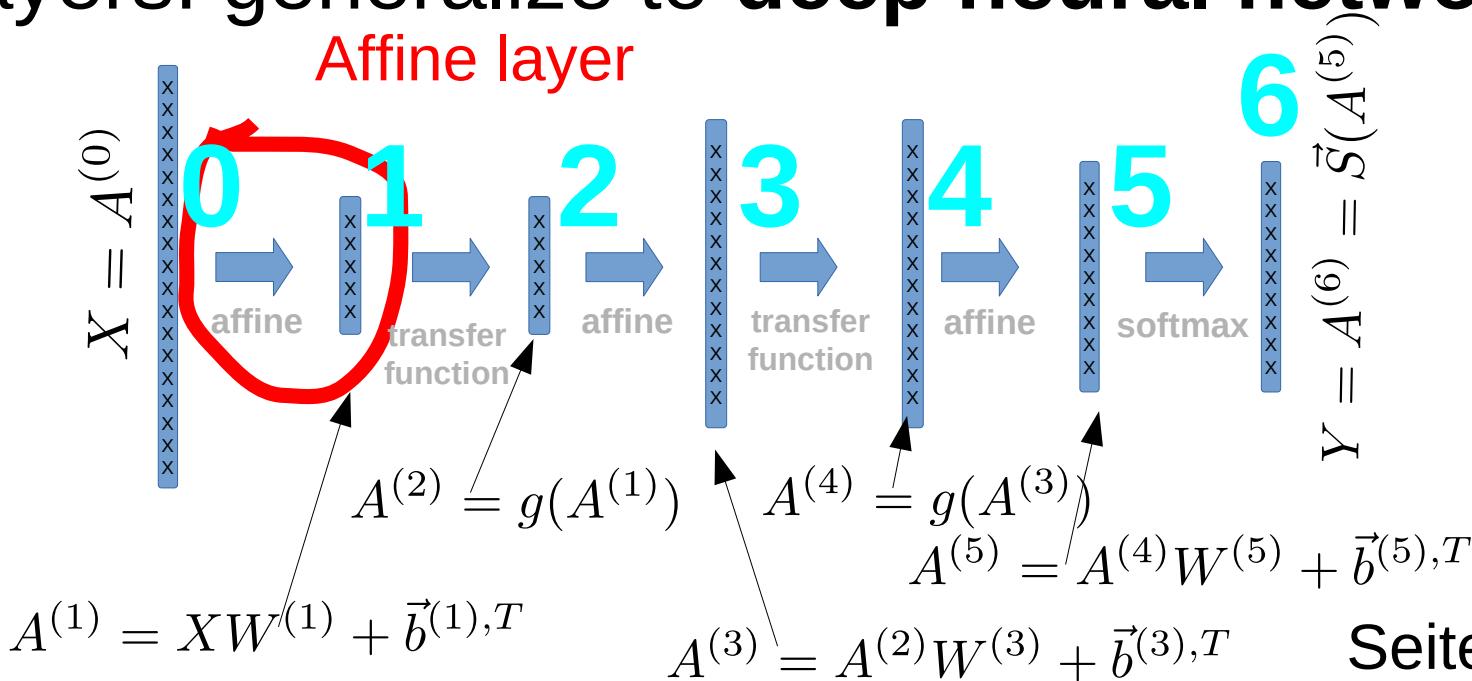
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

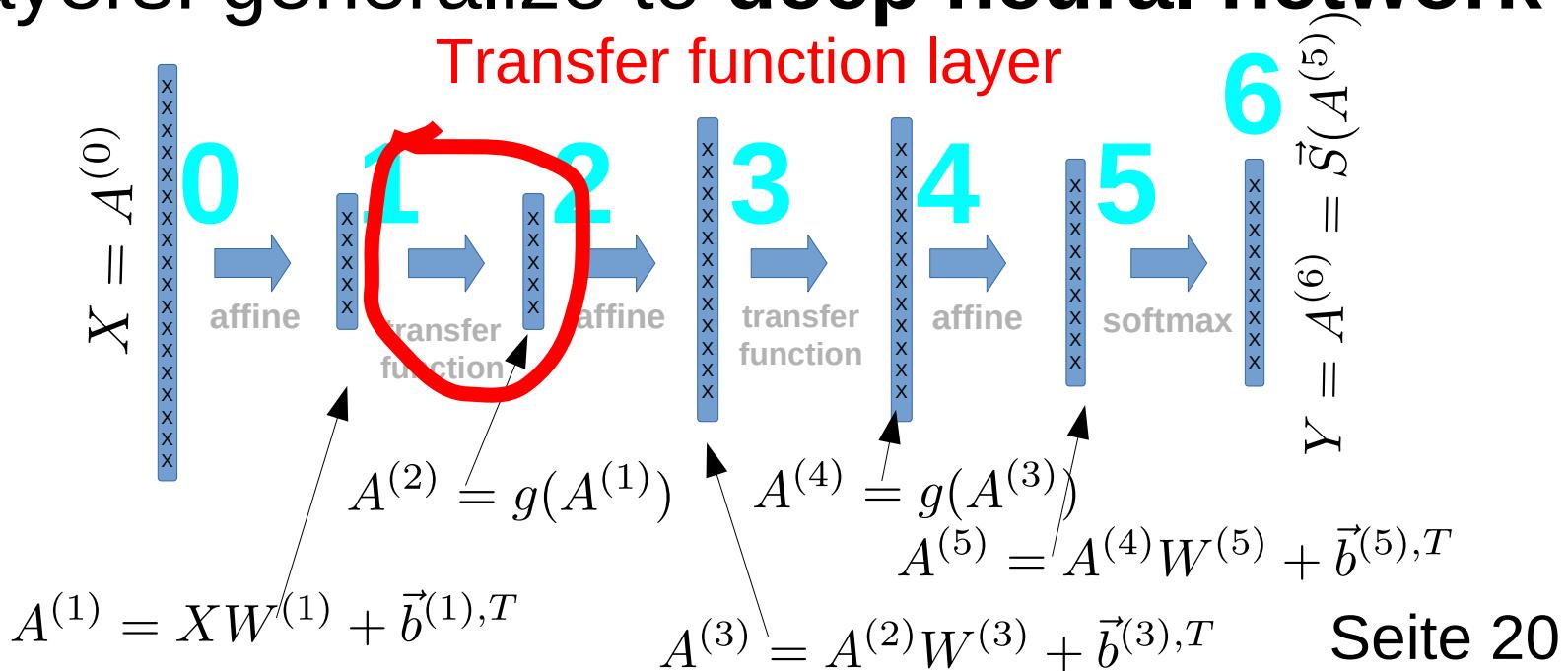
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

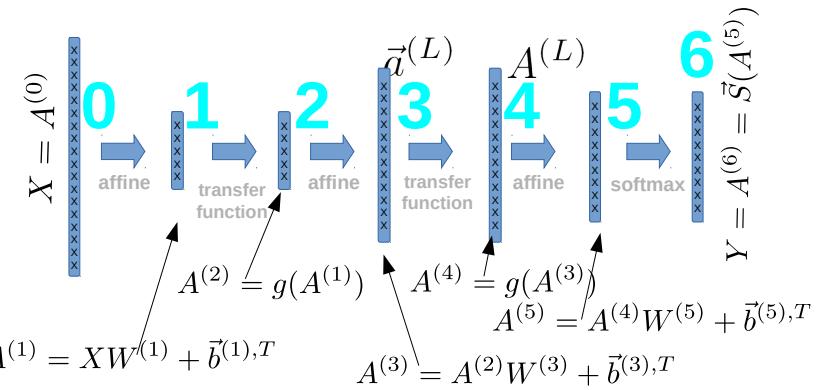
- No reason why there should be only two layers: generalize to **deep neural network**





# Deep neural networks / deep learning

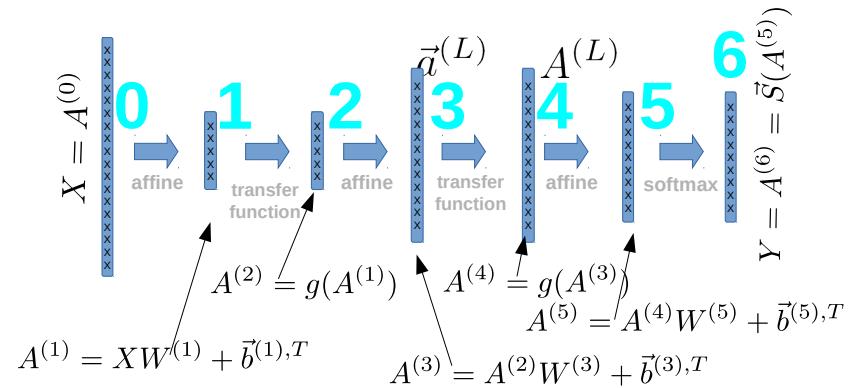
- DNNs are chains of successive transformations (layers)
- Output of layer L: **activity**  $A^{(L)}$
- Still a machine learning model, output  $Y$  at last layer
- Layer types are usually:
  - **affine layer**: affine, has adaptable parameters
  - **transfer function layer**: non-linear, no parameters
  - **softmax layer layer**: non-linear, no parameters





# Deep neural networks / deep learning

- Transfer functions: ReLU, softmax, Dropout, pooling
- Choice of last layer(s) determines type of model:
  - affine+MSE loss for regression
  - affine/softmax + CE loss for classification
- General principle in deep learning: more layers are better!





# Notation for DNNs

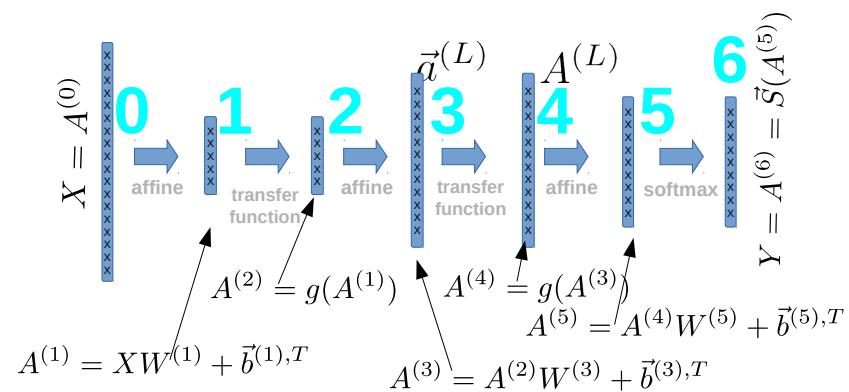
- All relevant quantities are numbered layer-wise:

$$\vec{b} \rightarrow \vec{b}^{(L)}$$

$$W \rightarrow W^{(L)}$$

$$A \rightarrow A^{(L)}, \text{ shape: } N \times Z^{(L)}$$

(mini-)batch size



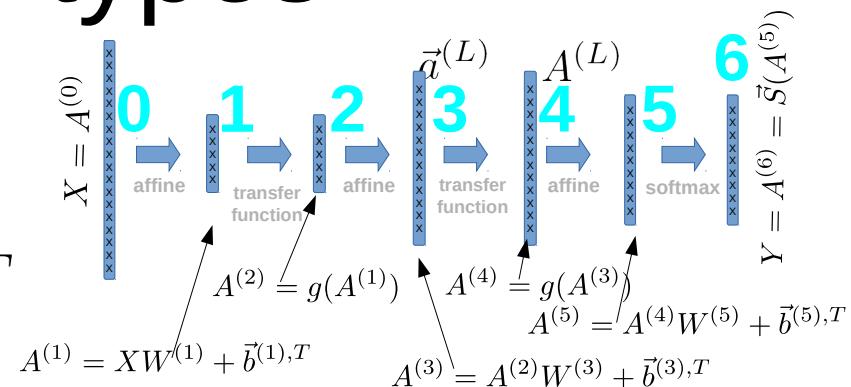
- Convention: layers start at 1, layer 0 is input:  $A^{(0)} \equiv X$
- Model output is activity of last layer:  $Y \equiv A^{(O)}$



# DNN layer types

- **Affine layers:**

$$A^{(L)} = A^{(L-1)}W^{(L)} + \vec{b}^{(L),T}$$



- Component-wise:

$$A_{ij}^{(L)} = \sum_l A_{il}^{(L-1)} W_{lj}^{(L)} + b_j^{(L)}$$

- Changes shape!
- Weight matrix has shape  $Z^{(L-1)} \times Z^{(L)}$   
Bias vector has shape  $1 \times Z^{(L)}$



# DNN layer types

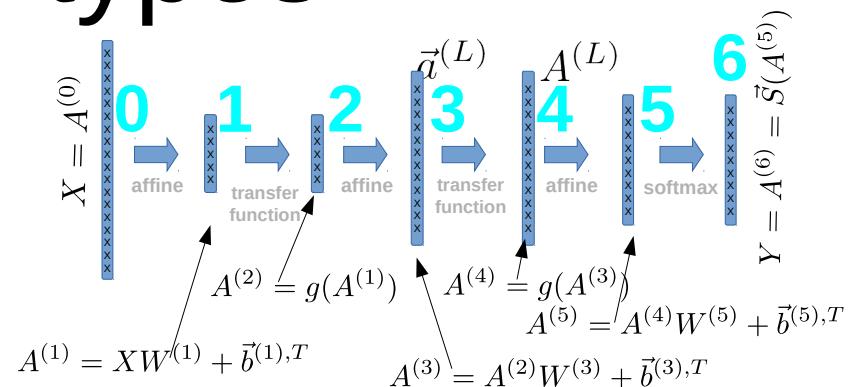
- **Softmax layers:**

$$A^{(L)} = \vec{S}(A^{(L-1)})$$

- Component-wise:

$$A_{ij}^{(L)} = S_j(A_{i,:}^{(L-1)})$$

- No change of shape!





# DNN layer types

- **Transfer function layers:**

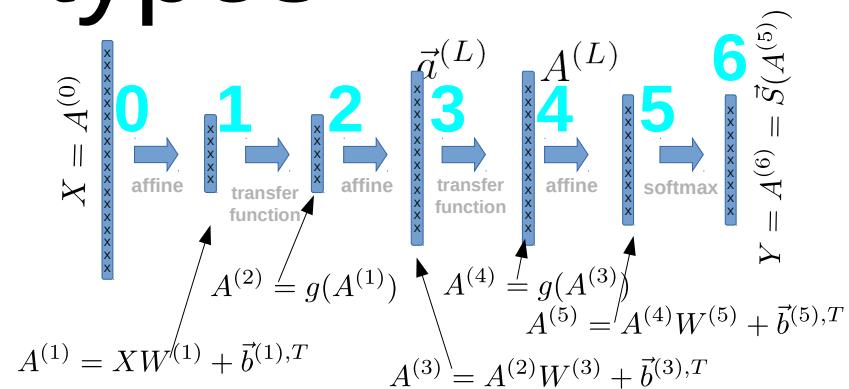
$$A^{(L)} = g\left(A^{(L-1)}\right)$$

with  $g : x \in \mathbb{R} \mapsto y \in \mathbb{R}$  applied element-wise

- Component-wise:

$$A_{ij}^{(L)} = g\left(A_{ij}^{(L-1)}\right)$$

- No change of shape!





# Cut: Q&A



# Layer type details

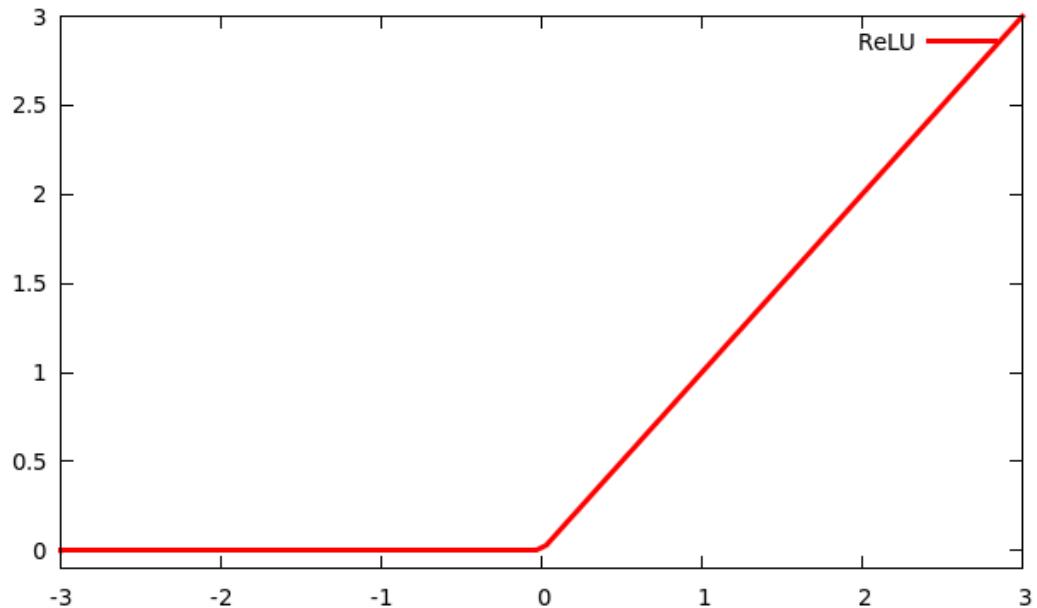


# The ReLU transfer function

- Rectified Linear Unit (ReLU):

$$g(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{sonst} \end{cases}$$

- default transfer function for DNNs
- works well in practice
- no theory to justify it!





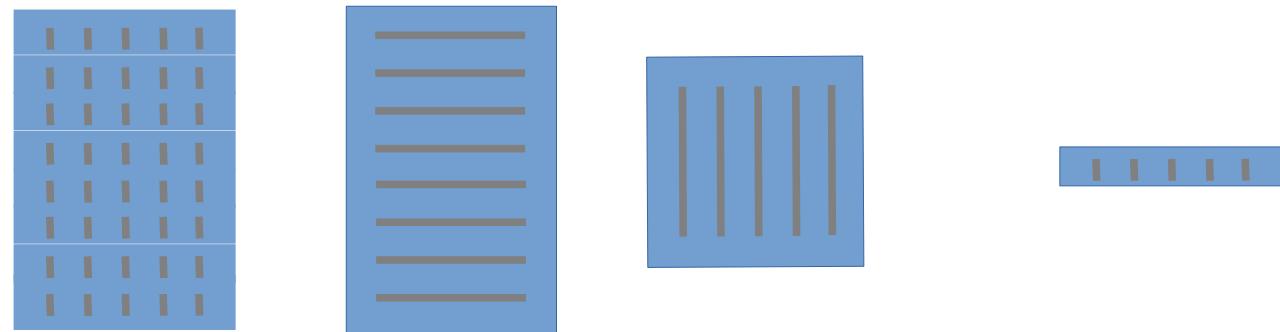
# Convolutional layers

- Special case of affine layers
- Treat preceding layer activity as stack of RGB images:
- $N \times Z^{(L-1)} \rightarrow N \times \underbrace{H^{(L-1)} \times W^{(L-1)} \times C^{(L-1)}}_{Z^{(L-1)}}$



# ConvLayers: basics

- Affine layer transformation:

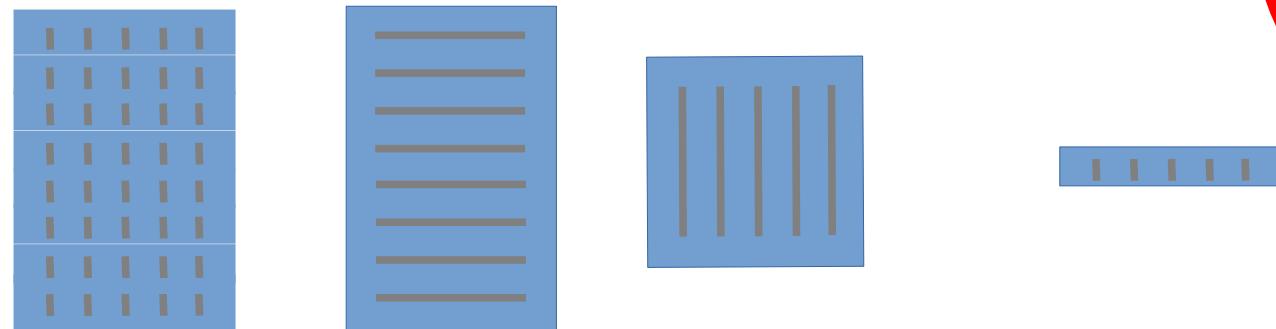


$$A^{(L)} = A^{(L-1)} W^{(L)} + \vec{b}^{(L),T}$$



# ConvLayers: basics

- Affine layer transformation:



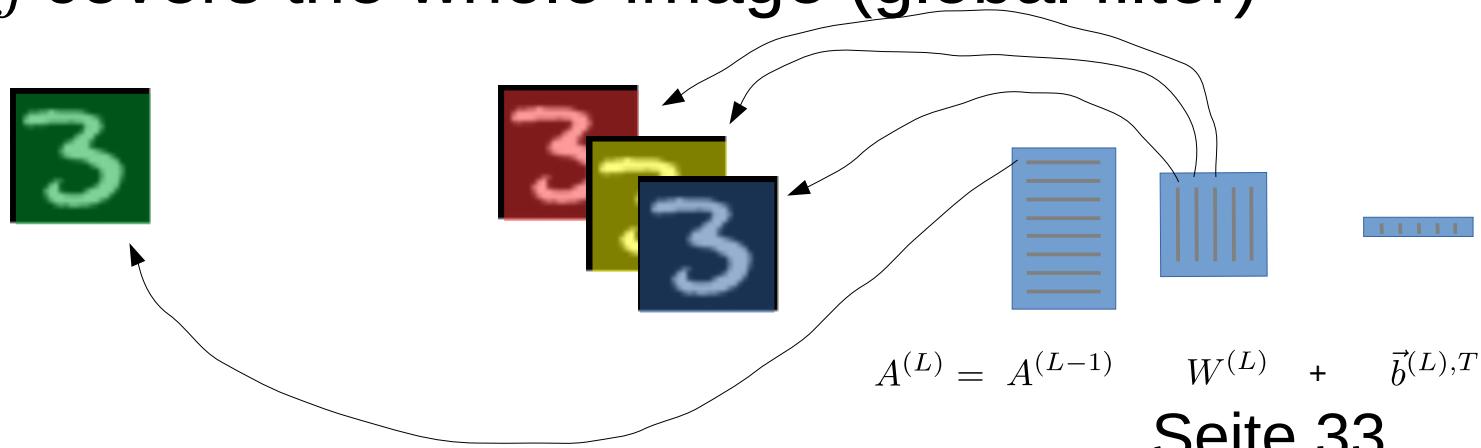
If each row of  $A^{(L-1)}$   
is a RGB image,  
then so are the  
columns of  $W^{(L)}$ .  
→ **filters**

$$A^{(L)} = A^{(L-1)} \cdot W^{(L)} + \vec{b}^{(L),T}$$



# ConvLayers: global filters

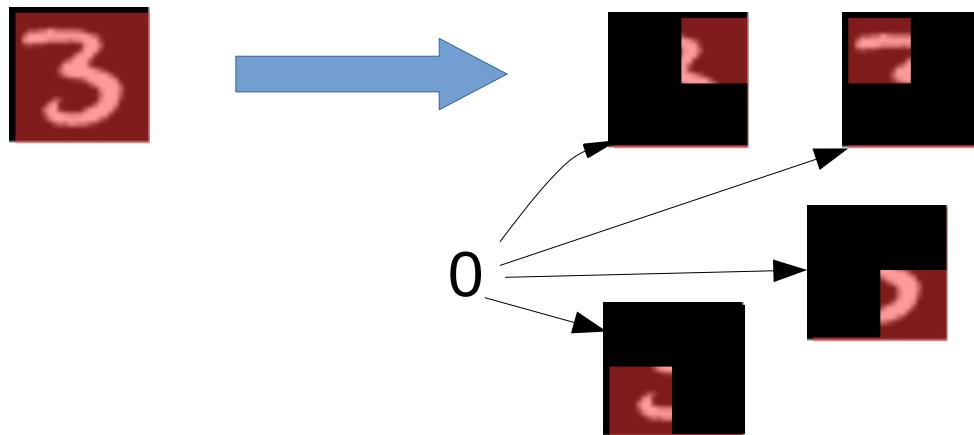
- ConvLayers (affine layers) have filters:
  - each element  $A_{ij}^{(L)}$  is the result of a scalar product between column  $W_{:,j}$  and  $A^{(L-1)}$
  - filter  $j$  covers the whole image (global filter)





# ConvLayers: local filters

- Restriction: filters do not cover the whole image but only a part → filter size  $f_X, f_Y$

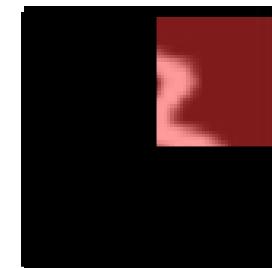
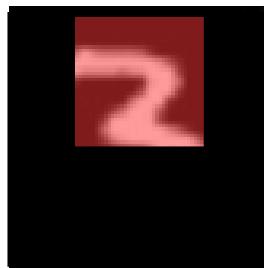
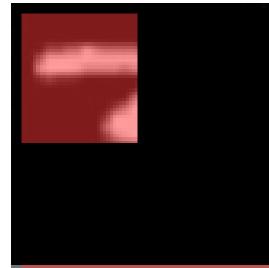




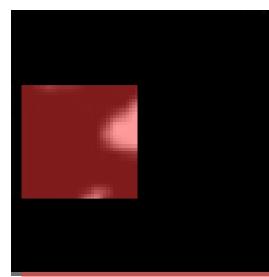
# ConvLayers: strides

- Local filters are applied in steps (**strides**)

$\Delta_X, \Delta_Y$



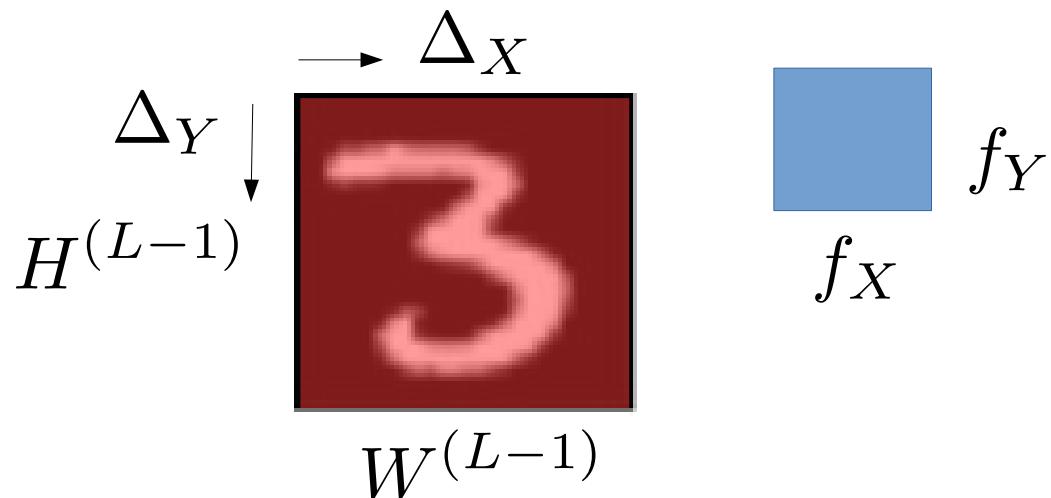
$\Delta_Y \downarrow$





# ConvLayers: quiz

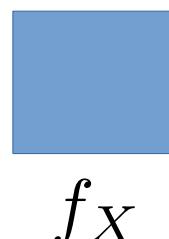
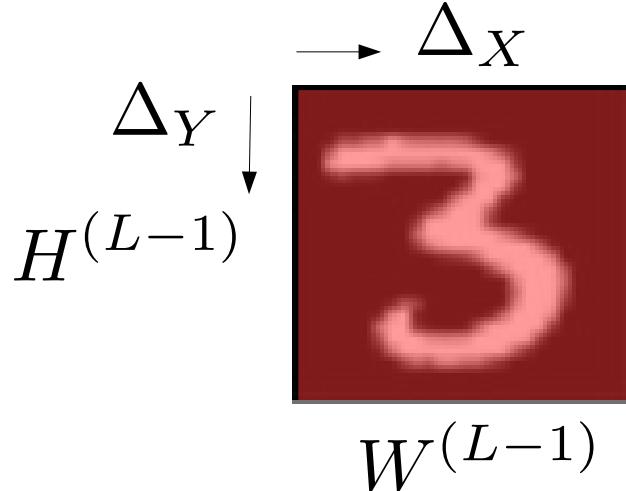
- If images have size  $H^{(L-1)}, W^{(L-1)}$ , filters are  $f_X, f_Y$  and strides are  $\Delta_X, \Delta_Y$ : how many filters do we need to cover the whole image?





# ConvLayers: quiz

- If images have size  $H^{(L-1)}, W^{(L-1)}$ , filters are  $f_X, f_Y$  and strides are  $\Delta_X, \Delta_Y$ : how many filters do we need to cover the whole image?



$$(1 + \frac{H^{(L-1)} - f_Y}{\Delta_Y}) \times (1 + \frac{W^{(L-1)} - f_Y}{\Delta_X})$$



# ConvLayers: multiple filters and weight-sharing

- Up to now, we require a number of neurons determined by
$$(1 + \frac{H^{(L-1)} - f_Y}{\Delta_Y}) \times (1 + \frac{W^{(L-1)} - f_Y}{\Delta_X})$$
- Now: for each filter position, apply  $C^{(L)}$  filters instead of a single one
- Weights matrix entries between filters at different positions are shared!



# Typical CNN transfer function: Max-Pooling/subsampling

- Type of transfer function, inspired by cortical visual processing
- meant to limit number of parameters
- each channel is subdivided into  $k \times k$  non-overlapping squares ( $k$ : kernel size)
- just keep maximal activity from each kernel  
→ strong size reduction!

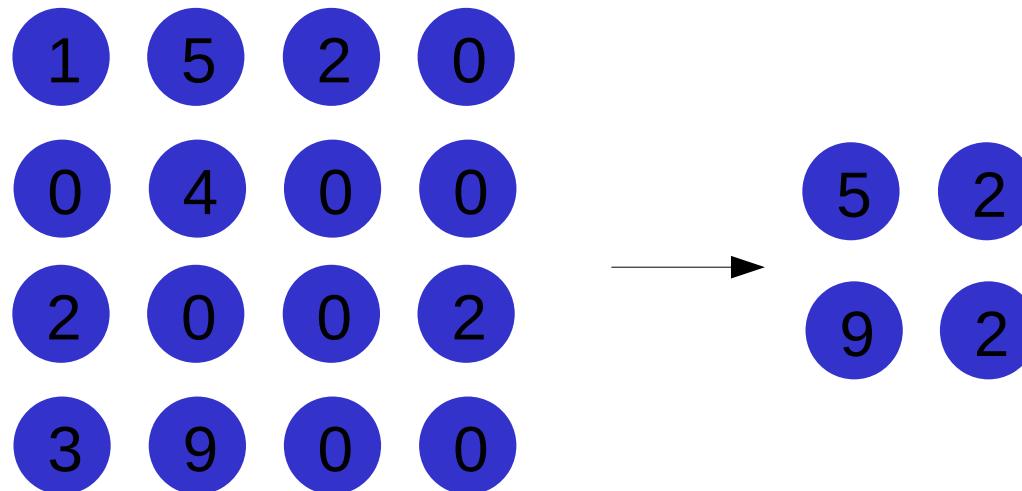


# Cut: Q&A



# Max-Pooling/subsampling

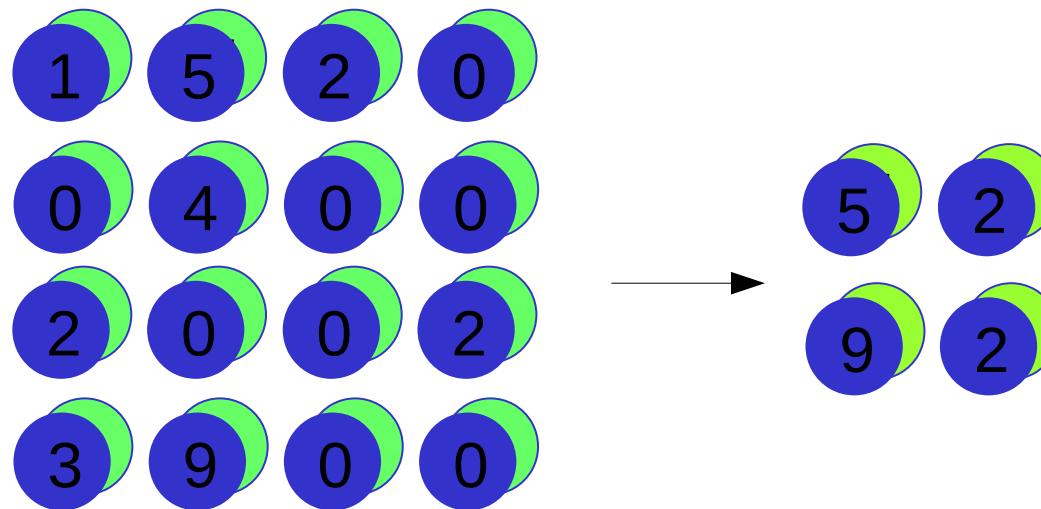
- Example for a single channel and kernel size 2:





# Max-Pooling/subsampling

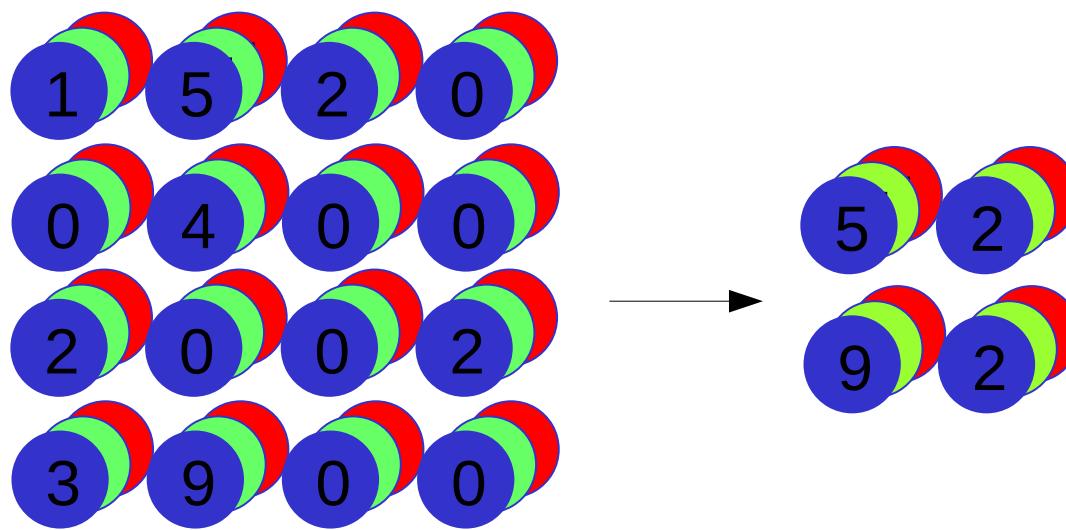
- Example for two channels and kernel size 2:





# Max-Pooling/subsampling

- Example for three channels and kernel size 2:





# Cut: Q&A



# DNNs can dream

- Deep dreaming: technique to visualize what individual layer's activities represent
- approach: after training a DNN ...
  - select a layer X and introduce new loss:  
$$\mathcal{L}^{DD} = \sum_n \sum_i \left( A_{ni}^{(X)} \right)^2$$
 (high if high layer X activities)
  - Leave weights and biases fixed, perform gradient ascent on inputs to maximize  $\mathcal{L}^{DD}$  :  
“what input produces highest activities in layer X?”

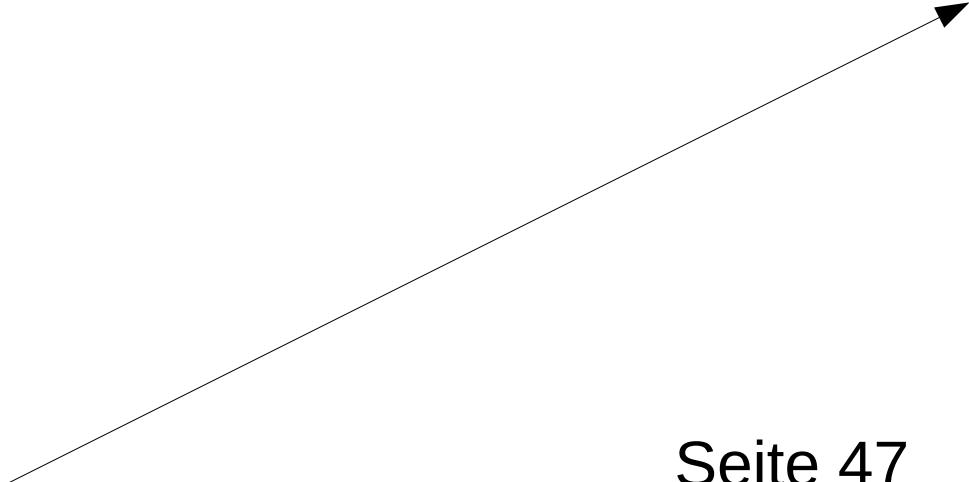


## DNNs can dream

- Consequences: none, enjoy the show!



# Deep dreaming: Beispiele





# Deep dreaming: Beispiele





# Deep dreaming: Beispiele



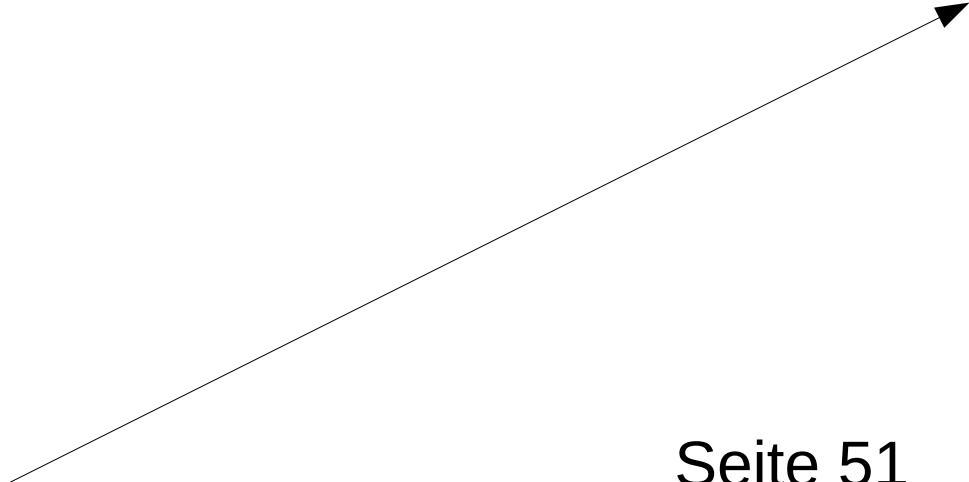


# Deep dreaming: Beispiele



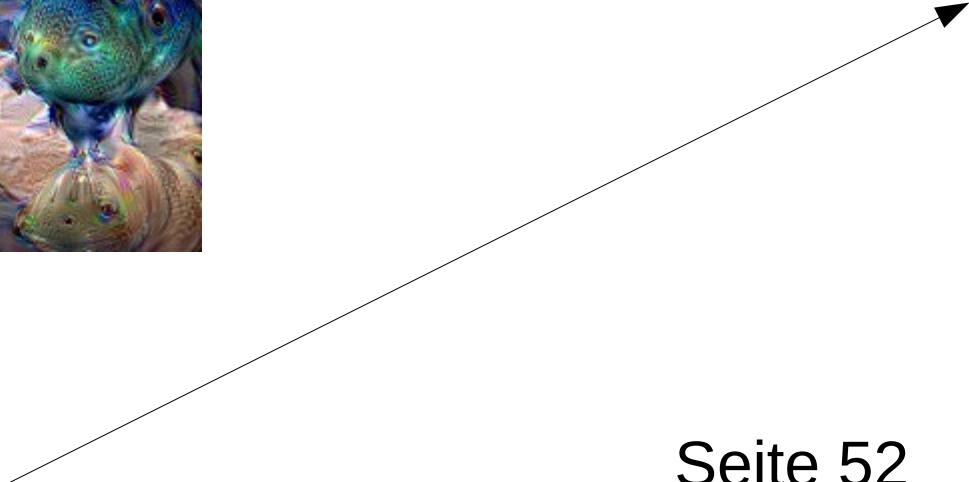


# Deep dreaming: Beispiele



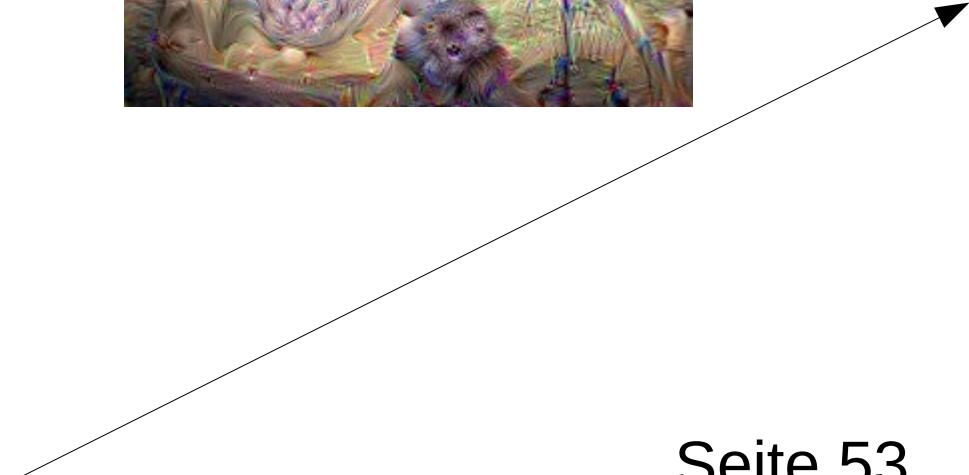


# Deep dreaming: Beispiele



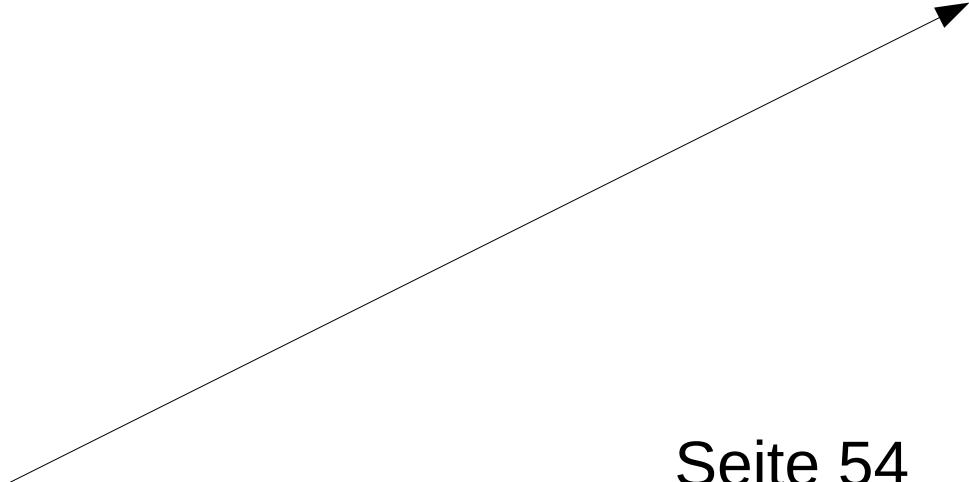
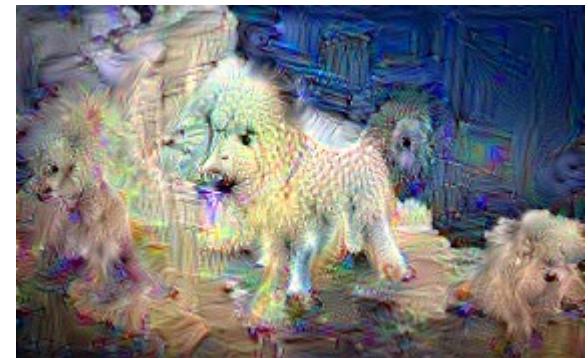


# Deep dreaming: Beispiele



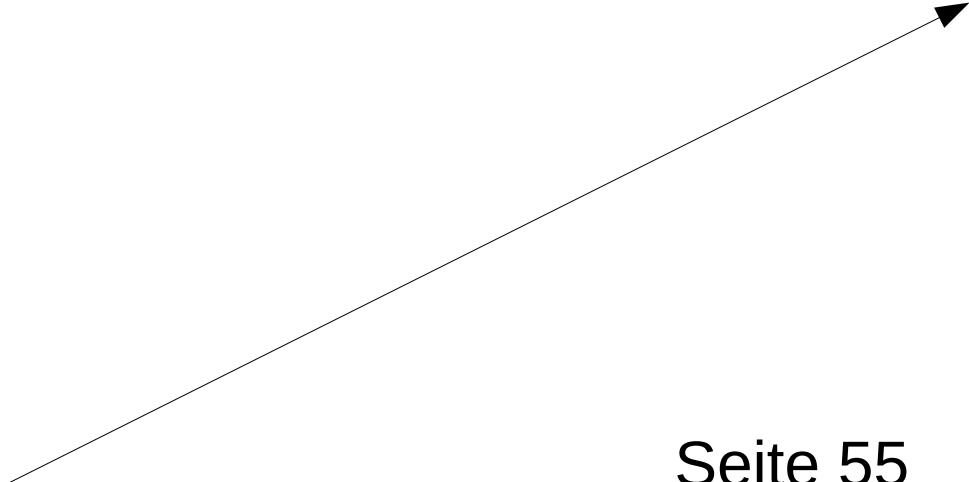


# Deep dreaming: Beispiele



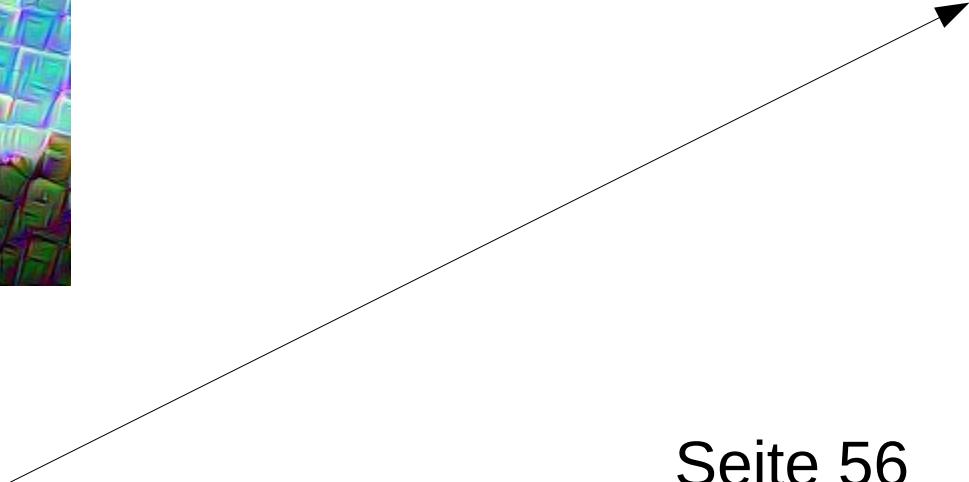
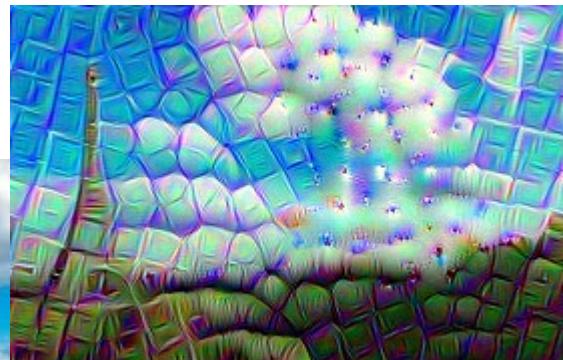


# Deep dreaming: Beispiele



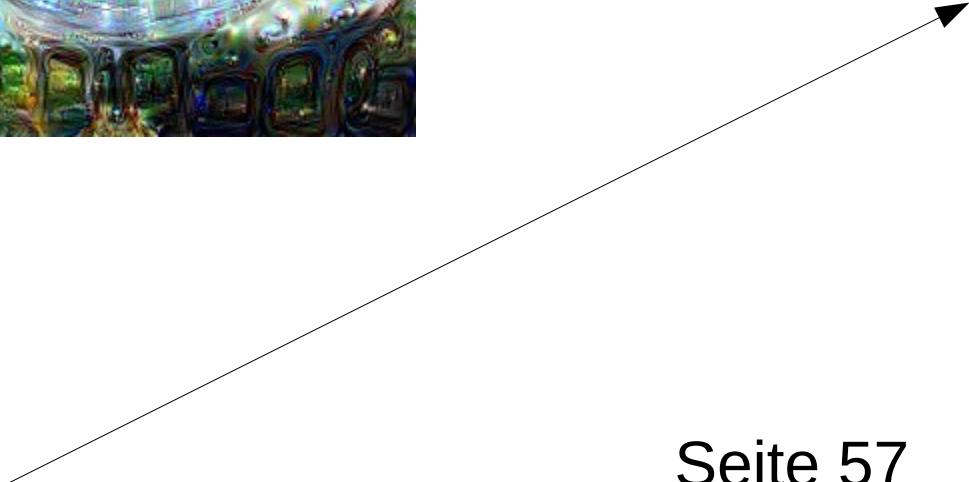


# Deep dreaming: Beispiele



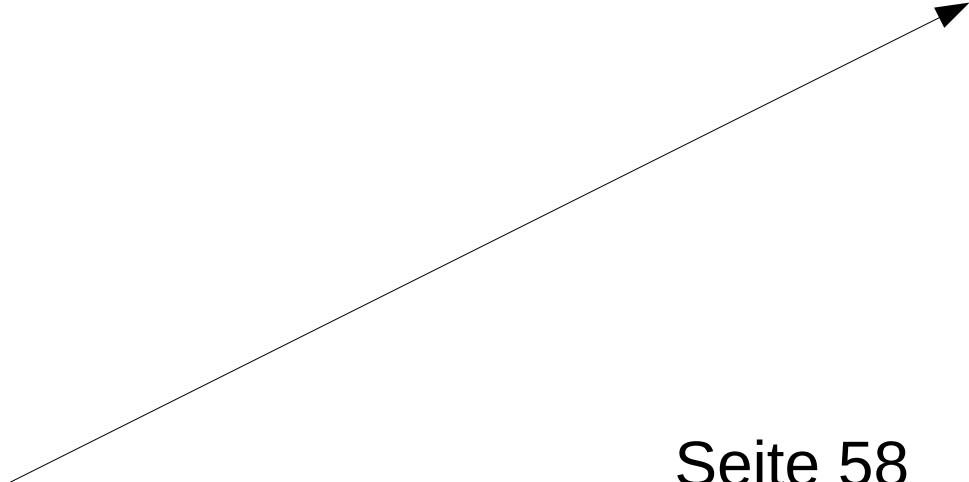


# Deep dreaming: Beispiele



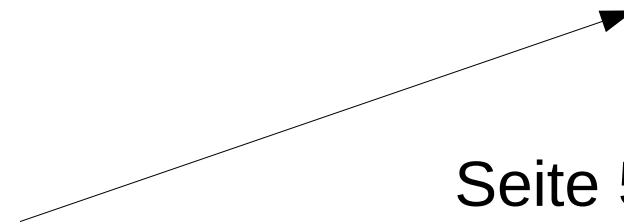


# Deep dreaming: Beispiele



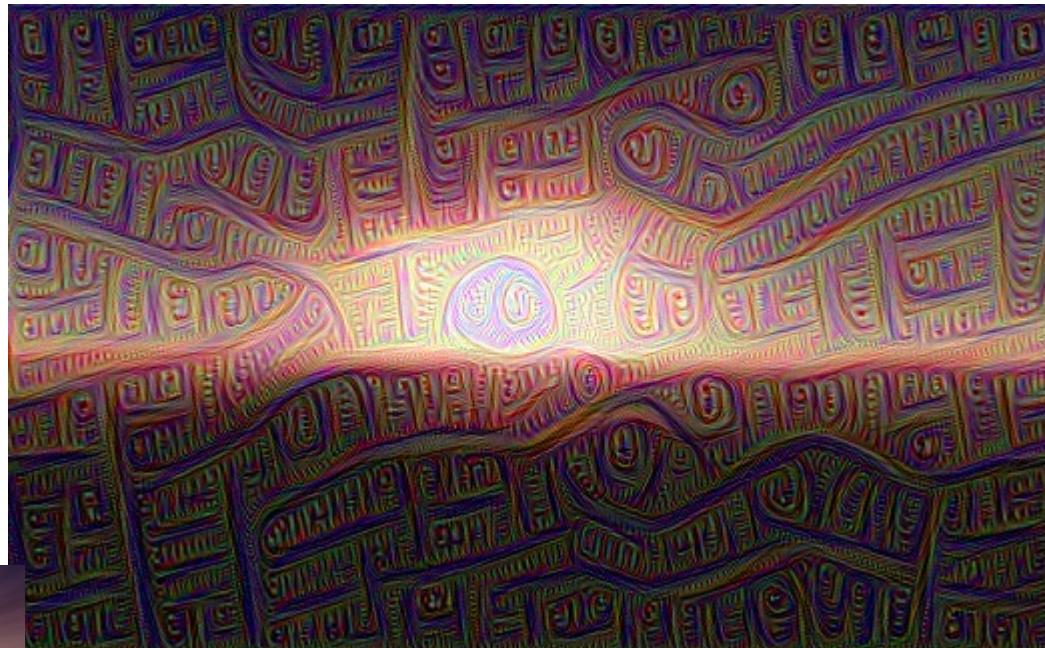


# Deep dreaming: Beispiele



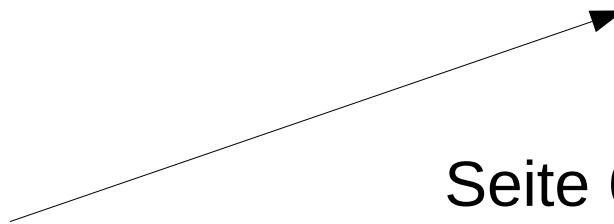


# Deep dreaming: Beispiele





# Deep dreaming: Beispiele





# Deep dreaming: Beispiele





# Deep dreaming: Beispiele



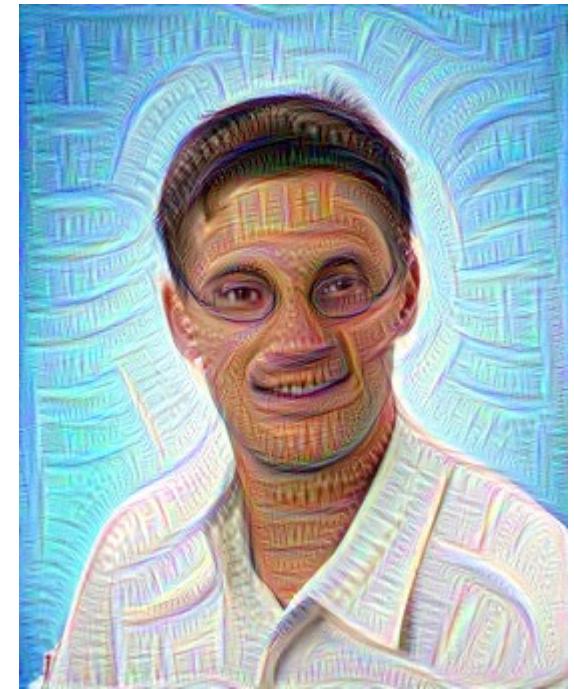


# Deep dreaming: Beispiele





# Deep dreaming: Beispiele





# Deep dreaming: Beispiele



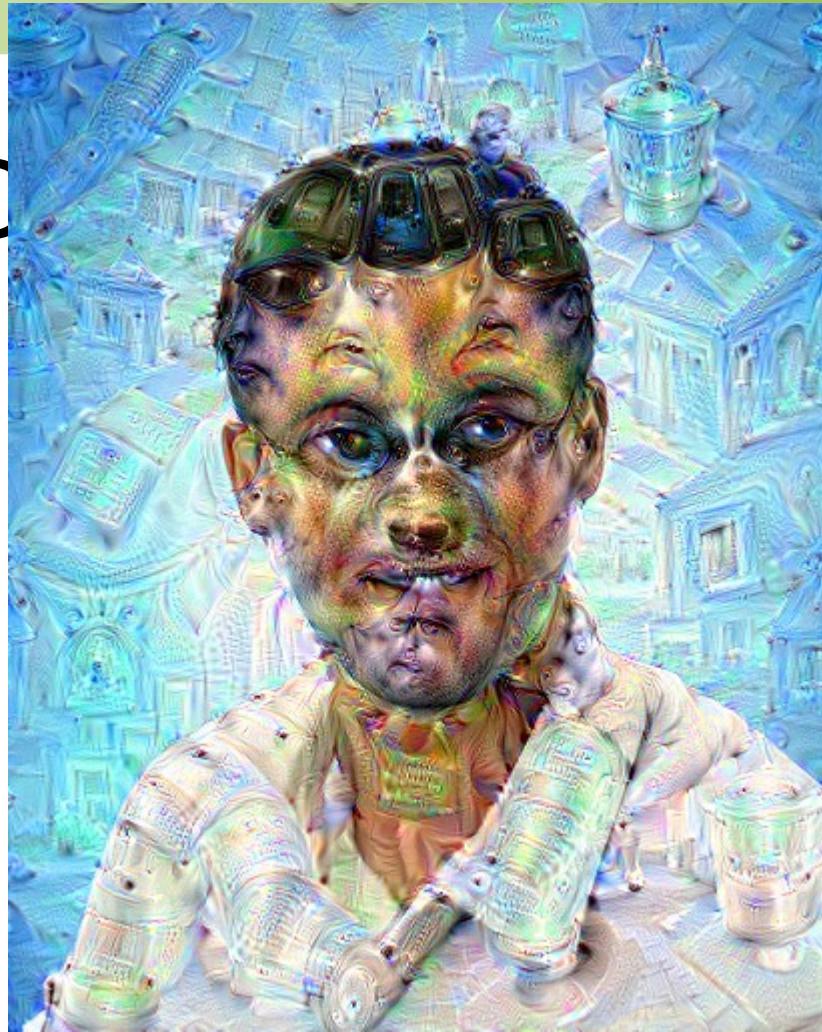


# Deep dreaming: Beispiele





D

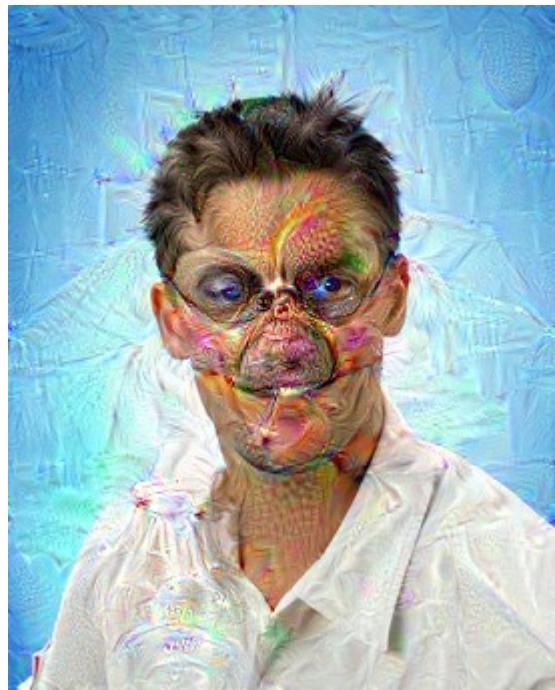


# Beispiele



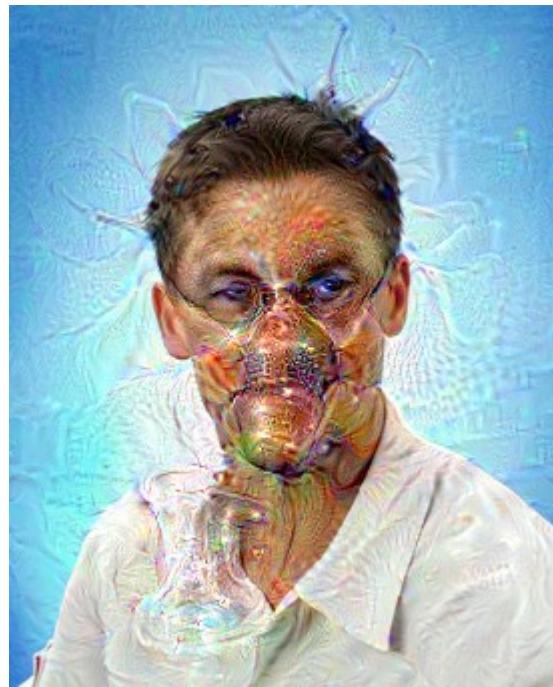


# Deep dreaming: Beispiele



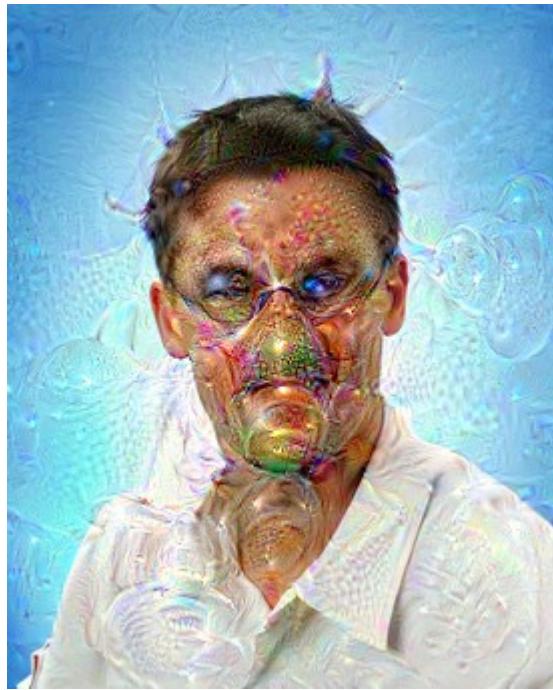


# Deep dreaming: Beispiele





# Deep dreaming: Beispiele





# Final remarks

- Most derivatives contain layer activities  
→ computation of derivatives requires previous computation of activities
- Training a DNN consists of a:
  - forward pass: computation of activities
  - backward pass: computation of derivatives and gradient descent
- Next lectures and exercises: complete example of forward and backward pass!



# Machine Learning

Lecture 7: deep learning and CNNs

Alexander Gepperth, January 2022



# Deep Learning



# Generalizing linear softmax MC to a DNN classifier

- Linear classifier:

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

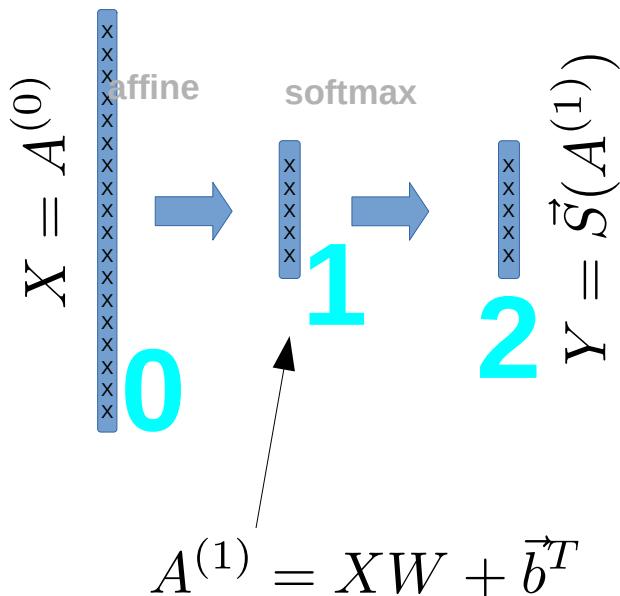
- Two successive transformations:
  - affine Transformation (**affine layer**)
  - softmax transformation (**softmax layer**)



# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

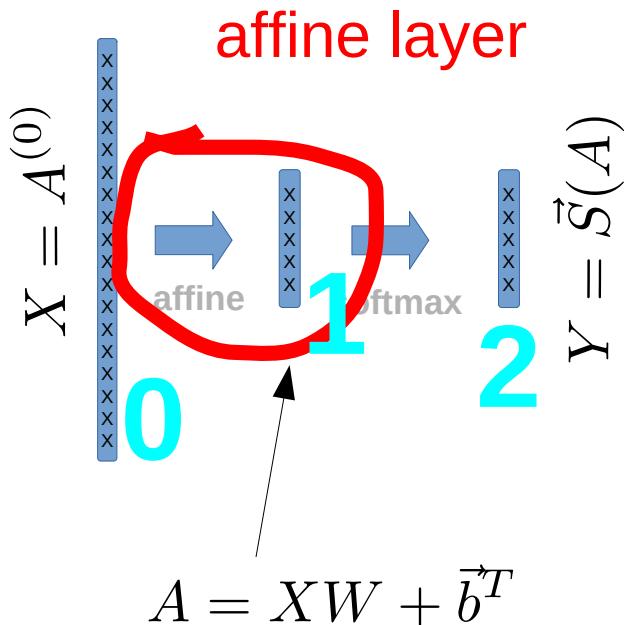




# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$

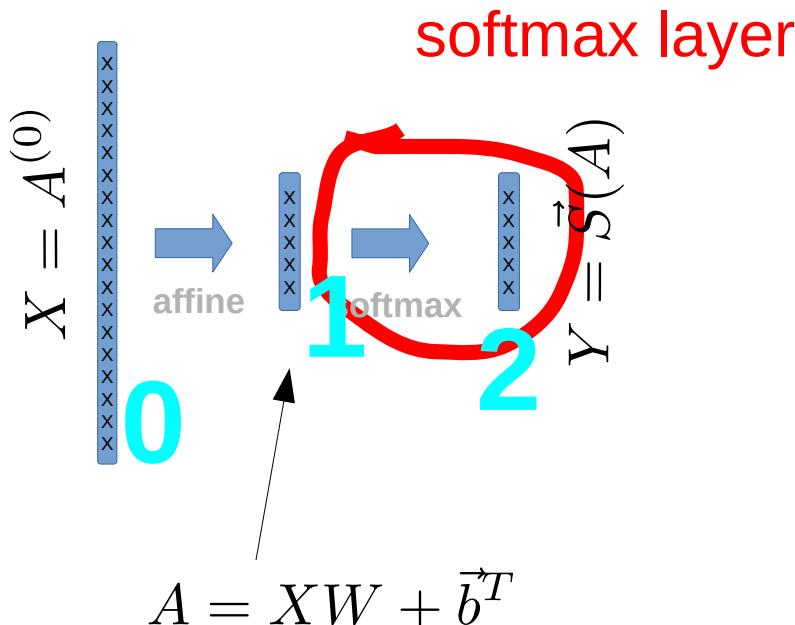




# Generalizing linear softmax MC to a DNN classifier

- Graphical notation (data flow: left → right)

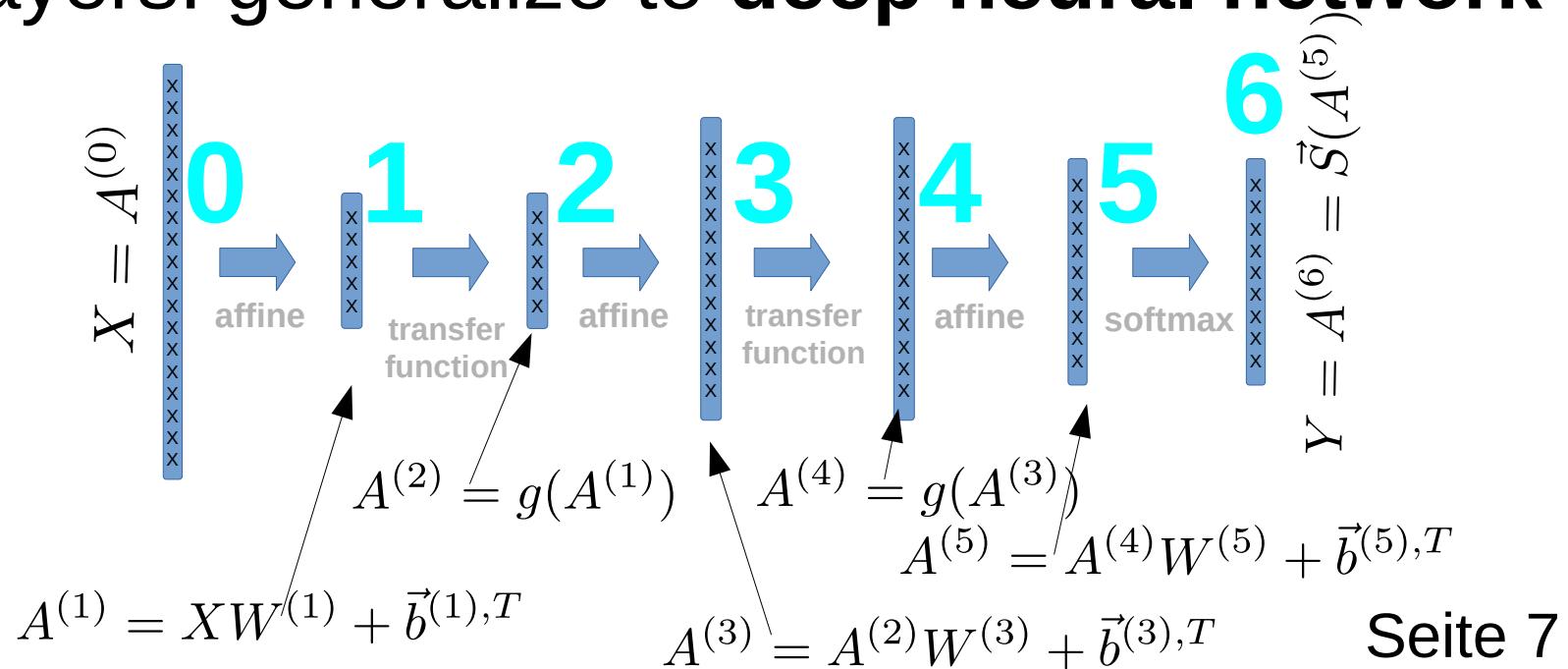
$$Y = \vec{f}(X, W, \vec{b}) = \vec{S}(XW + \vec{b}^T)$$





# Generalizing linear softmax MC to a DNN classifier

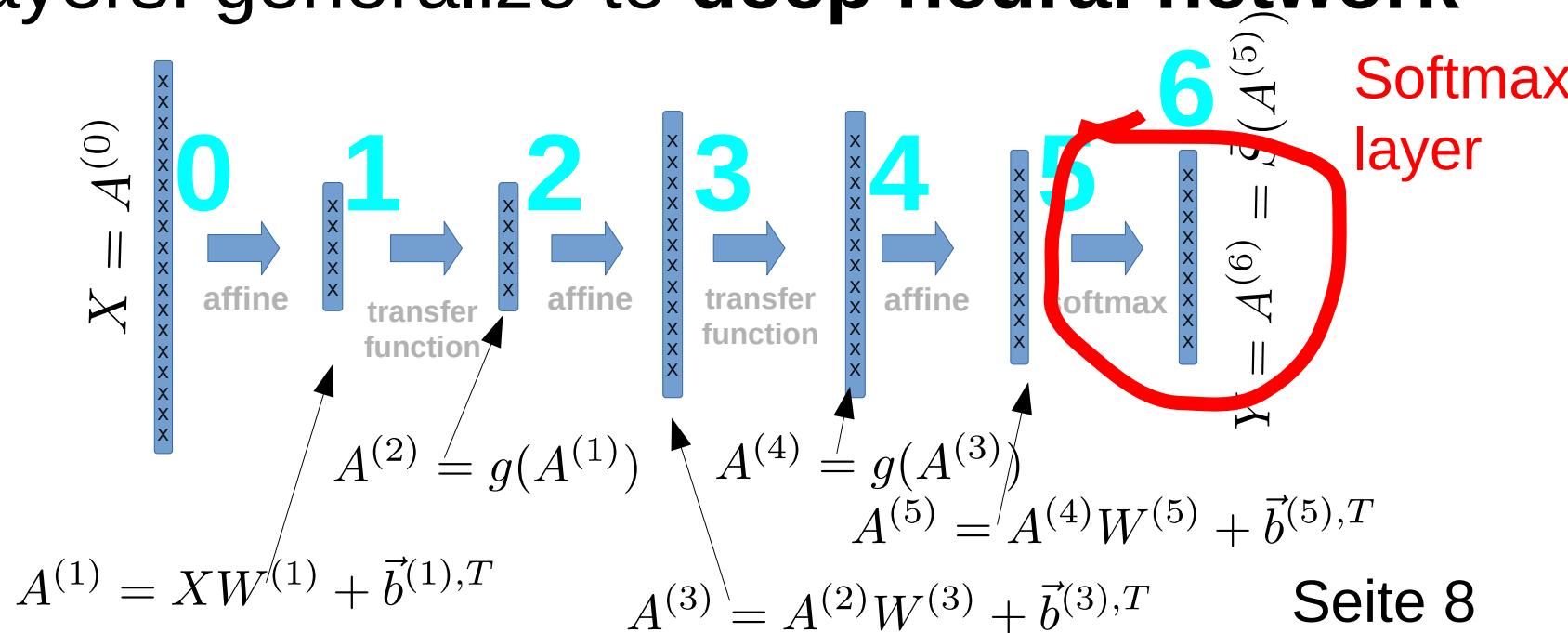
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

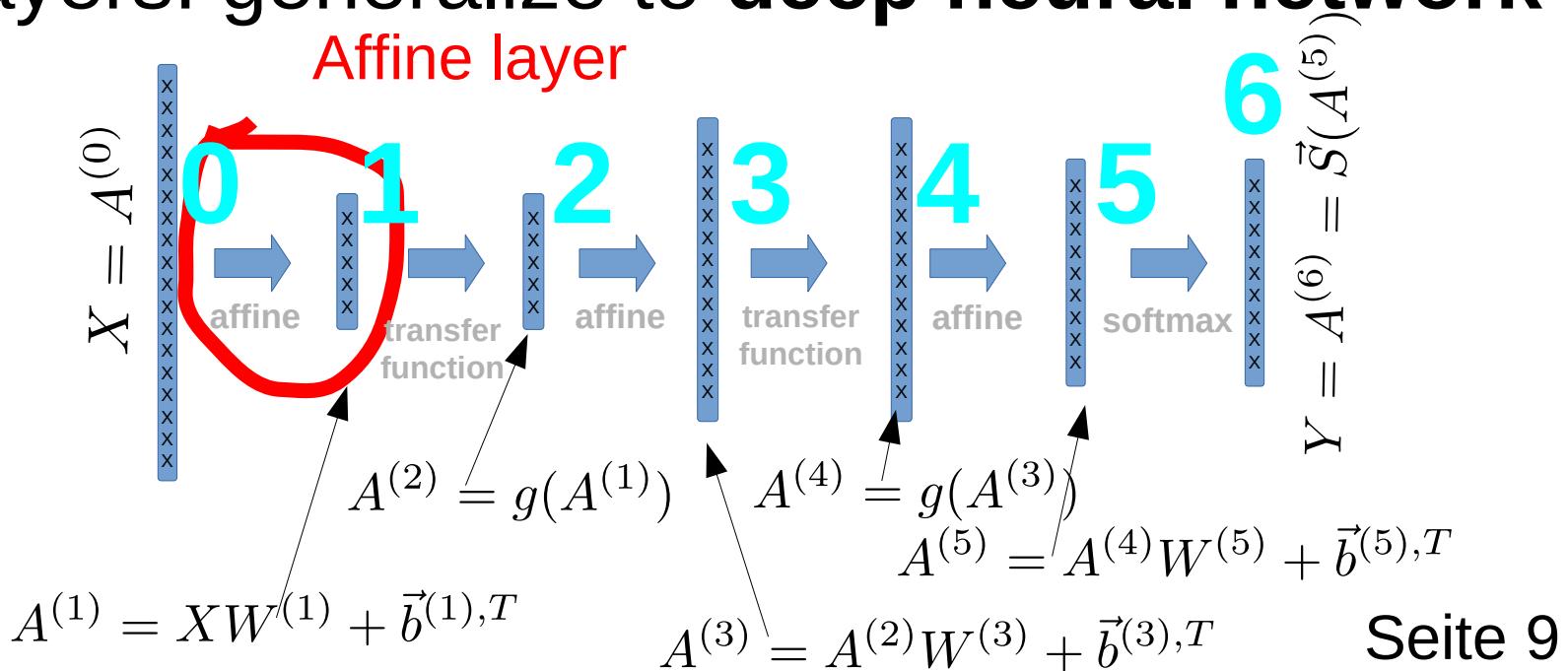
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

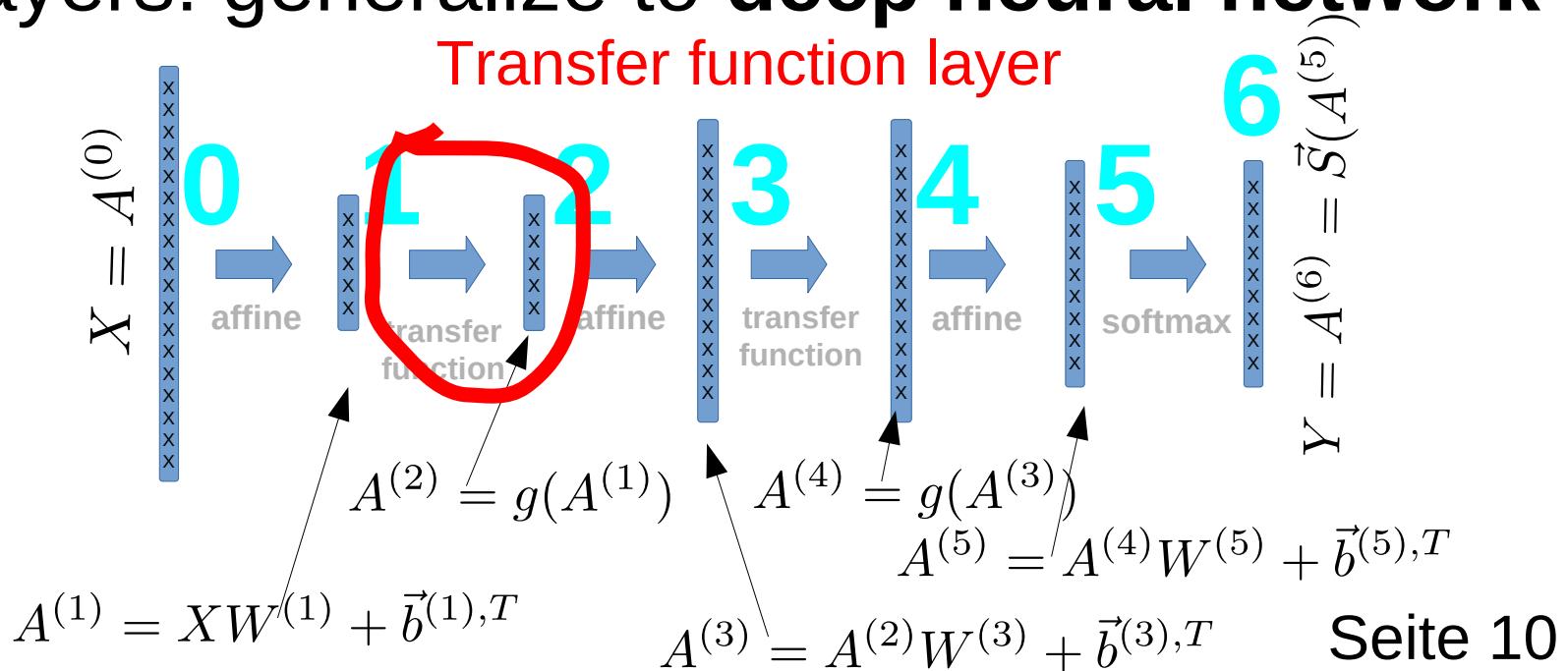
- No reason why there should be only two layers: generalize to **deep neural network**





# Generalizing linear softmax MC to a DNN classifier

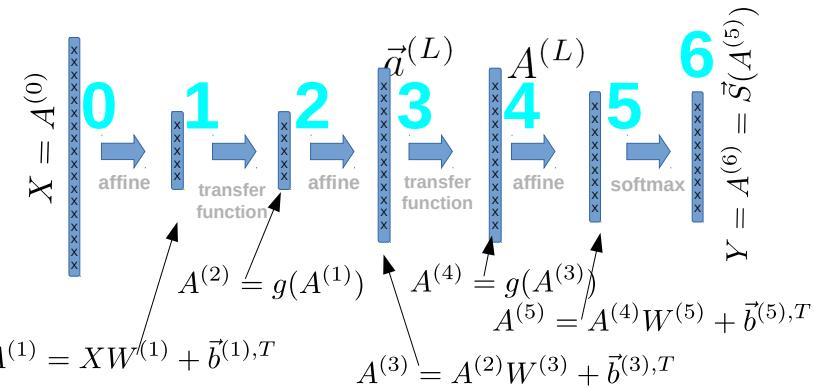
- No reason why there should be only two layers: generalize to **deep neural network**





# Deep neural networks / deep learning

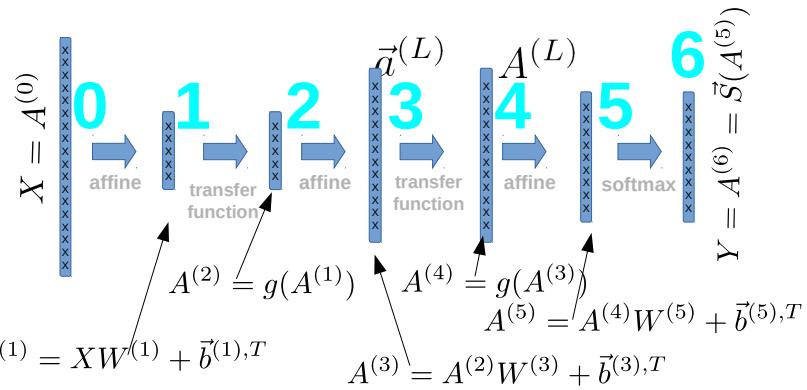
- DNNs are chains of successive transformations (layers)
- Output of layer L: **activity**  $A^{(L)}$
- Still a machine learning model, output  $Y$  at last layer
- Layer types are usually:
  - **affine layer**: affine, has adaptable parameters
  - **transfer function layer**: non-linear, no parameters
  - **softmax layer layer**: non-linear, no parameters
  - **pooling layer**: non-linear, no params





# Deep neural networks / deep learning

- Transfer functions are element-wise operations: dropout, ReLU
- Choice of last layer(s) determines type of model:
  - affine+MSE loss for regression
  - affine/softmax + CE loss for classification
- General principle in deep learning: more layers are better!





# Notation for DNNs

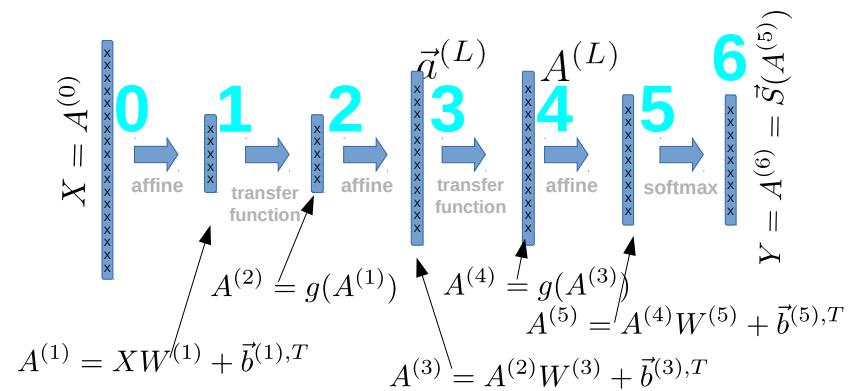
- All relevant quantities are numbered layer-wise:

$$\vec{b} \rightarrow \vec{b}^{(L)}$$

$$W \rightarrow W^{(L)}$$

$$A \rightarrow A^{(L)}, \text{ shape: } N \times Z^{(L)}$$

(mini-)batch size



- Convention: layers start at 1, layer 0 is input:  $A^{(0)} \equiv X$
- Model output is activity of last layer:  $Y \equiv A^{(O)}$



# Cut: Q&A



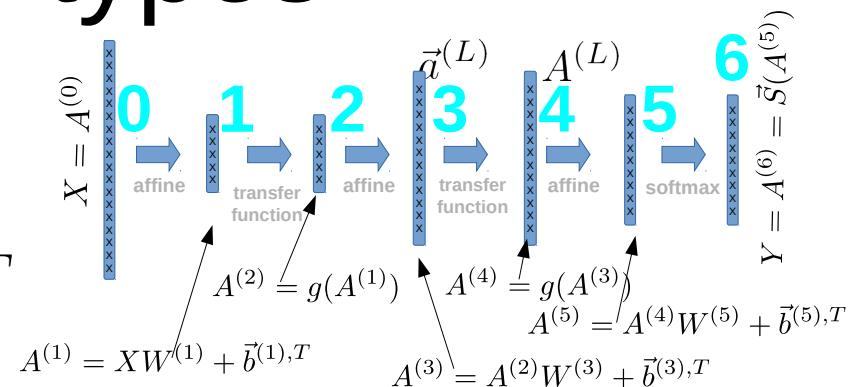
# DNN Layer types



# DNN layer types

- **Affine layers:**

$$A^{(L)} = A^{(L-1)}W^{(L)} + \vec{b}^{(L),T}$$



- Component-wise:

$$A_{ij}^{(L)} = \sum_l A_{il}^{(L-1)} W_{lj}^{(L)} + b_j^{(L)}$$

- Changes shape!
- Weight matrix has shape  $Z^{(L-1)} \times Z^{(L)}$   
Bias vector has shape  $1 \times Z^{(L)}$



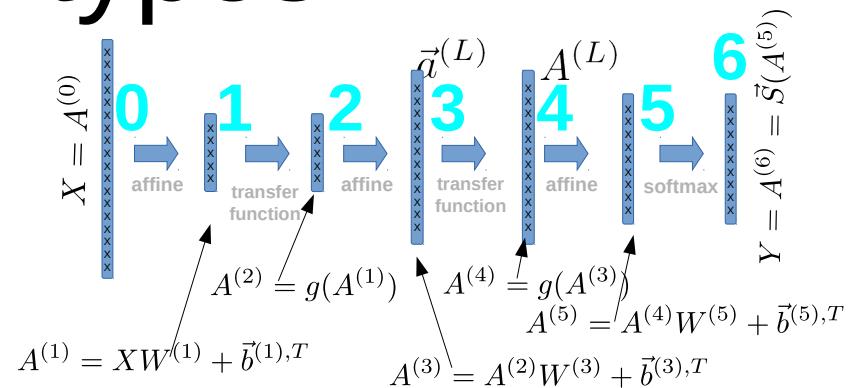
# DNN layer types

- **Softmax layers:**

$$A^{(L)} = \vec{S}(A^{(L-1)})$$

- Component-wise:

$$A_{ij}^{(L)} = S_j(A_{i,:}^{(L-1)})$$



- No change of shape!



# DNN layer types

- **Transfer function layers:**

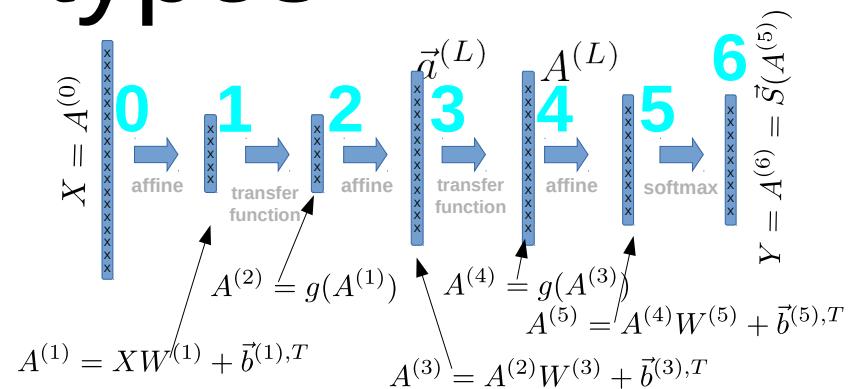
$$A^{(L)} = g\left(A^{(L-1)}\right)$$

with  $g : x \in \mathbb{R} \mapsto y \in \mathbb{R}$  applied element-wise

- Component-wise:

$$A_{ij}^{(L)} = g\left(A_{ij}^{(L-1)}\right)$$

- No change of shape!



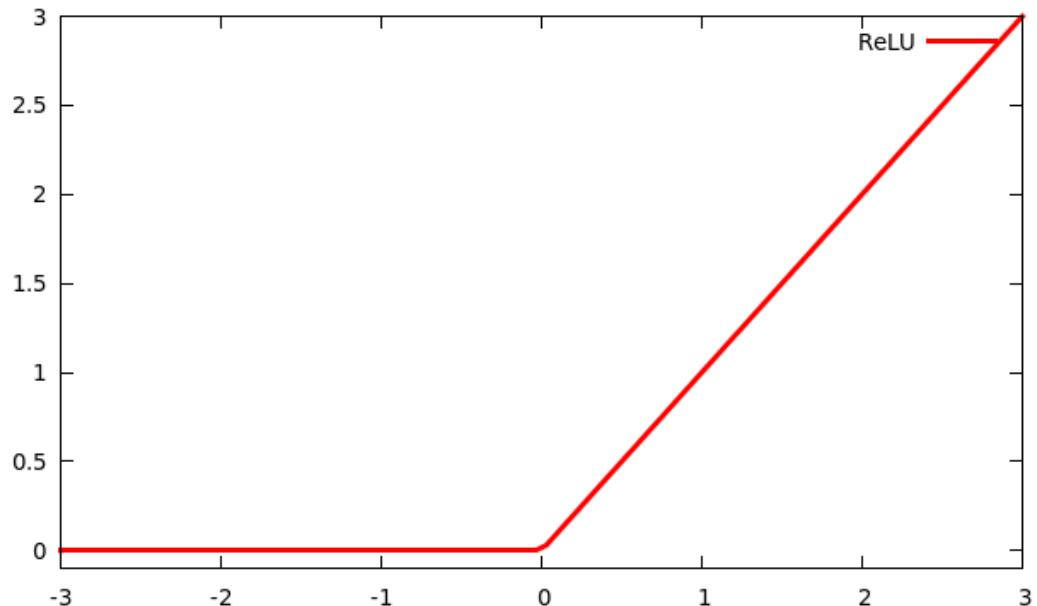


# The ReLU transfer function

- Rectified Linear Unit (ReLU):

$$g(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{sonst} \end{cases}$$

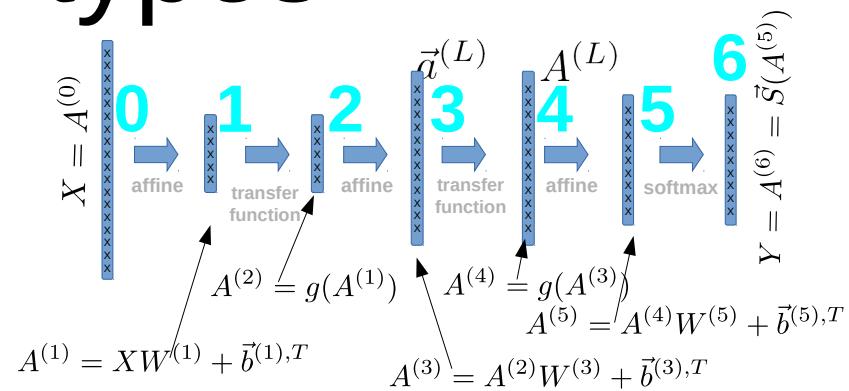
- default transfer function for DNNs
- works well in practice
- no theory to justify it!





# DNN layer types

- **Special layer types:  
convolutional layer,  
pooling layer**
- Only used when constructing convolutional neural networks (CNNs)





# Cut: Q&A



# Convolutional neural networks



# CNNs

- Convolutional neural networks were proposed around 1998 by Yann LeCun
- Optimized for processing (RGB) images
- Special type of DNNs, with two particular layer types:
  - convolutional layer
  - pooling layer



# Convolutional layers

- Special case of affine DNN layers
- Activities are still 2D matrices ...
- But: **act** as if preceding layer activity  $A^{(L-1)}$  was a stack of multi-channel (e.g., RGB) images:

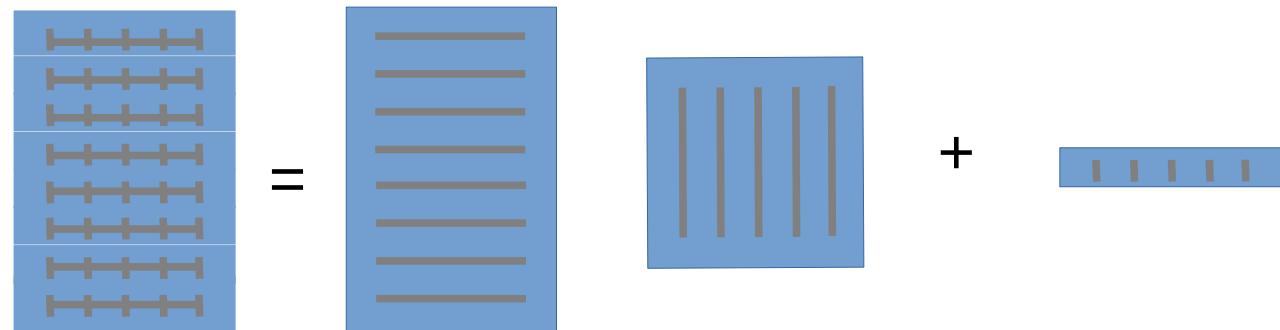
$$N \times Z^{(L-1)} \rightarrow N \times \underbrace{H^{(L-1)} \times W^{(L-1)} \times C^{(L-1)}}_{Z^{(L-1)}}$$

- Similar for current layer activity



# ConvLayers are affine layers

- Affine layer transformation:

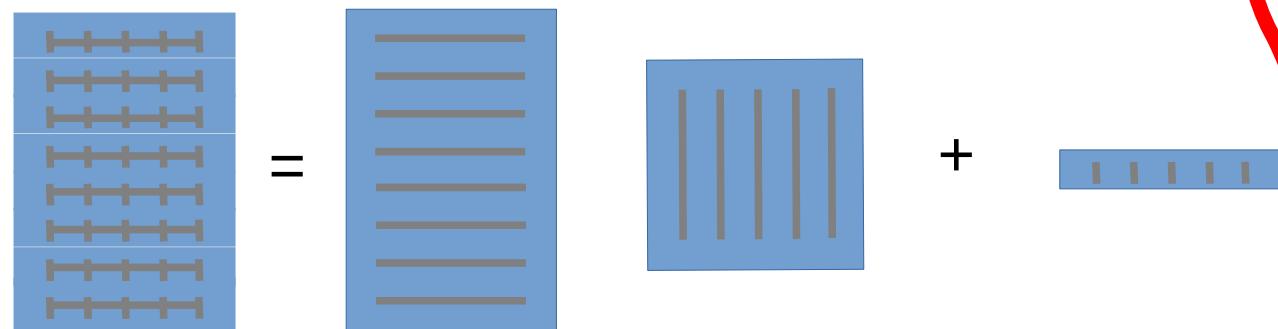


$$A^{(L)} = A^{(L-1)} W^{(L)} + \vec{b}^{(L),T}$$



# ConvLayers are affine layers

- Affine layer transformation:



If each row of  $A^{(L-1)}$  is a flat RGB image, then so are the columns of  $W^{(L)}$ .  
→ **filters**

$$A^{(L)} = A^{(L-1)} W^{(L)} + \vec{b}^{(L),T}$$



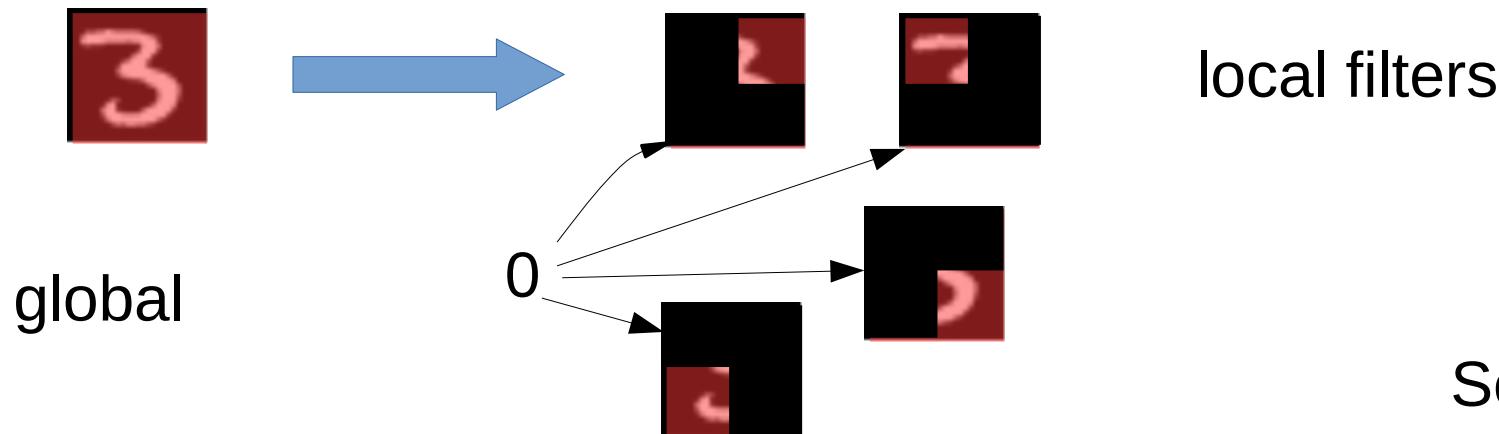
# ConvLayers: global filters

- Affine layers have global filters:
    - each element  $A_{ij}^{(L)}$  is the result of a scalar product between filter  $j$  ( $W_{:,j}$ ) and prev. layer image  $i$  ( $A_{i,:}^{(L-1)}$ )
    - this measures similarity of all prev. layer images to all filters
- $$A^{(L)} = A^{(L-1)} \cdot W^{(L)} + \vec{b}^{(L),T}$$
-



# ConvLayers: local filters

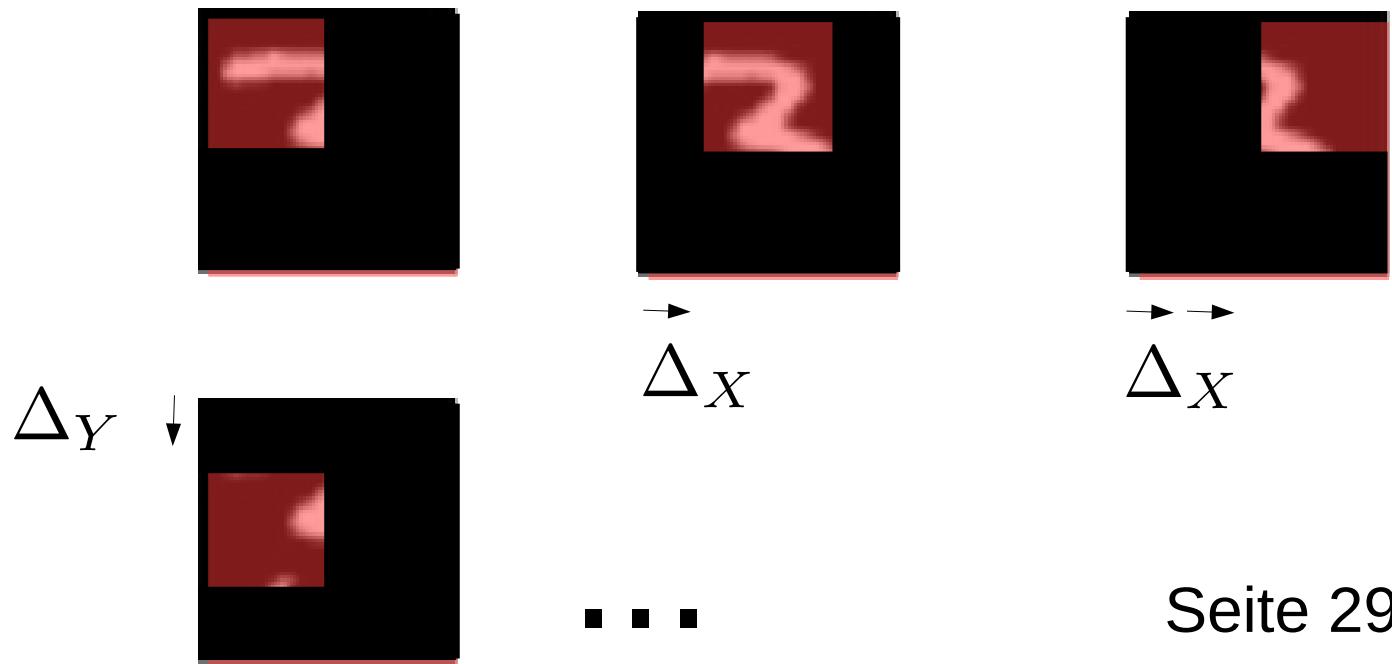
- Restriction: filters do not “cover” the whole image but only a part → local filters
- Local filters have position and size  $f_X, f_Y$





# ConvLayers: strides

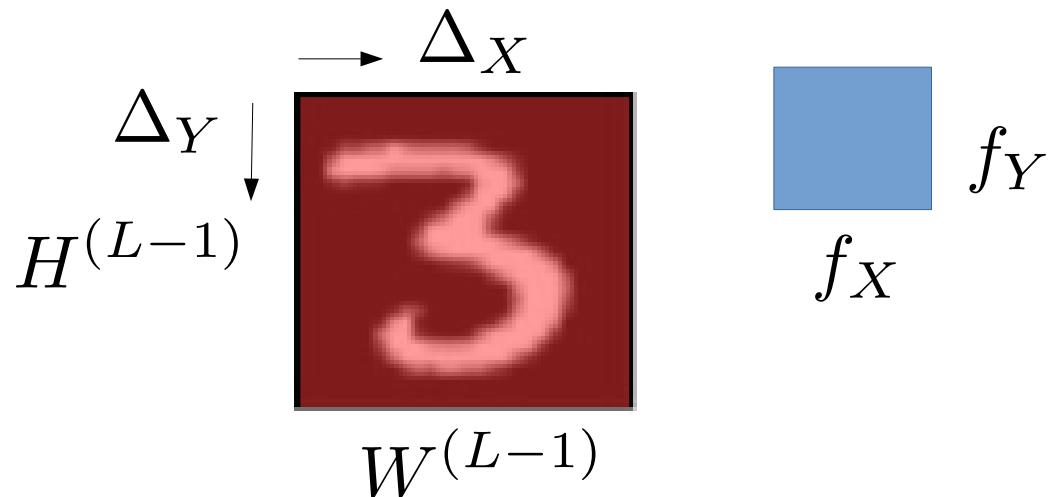
- Local filter positions are defined by **strides**  
 $\Delta_X, \Delta_Y$





# ConvLayers: quiz

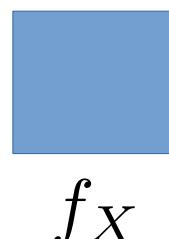
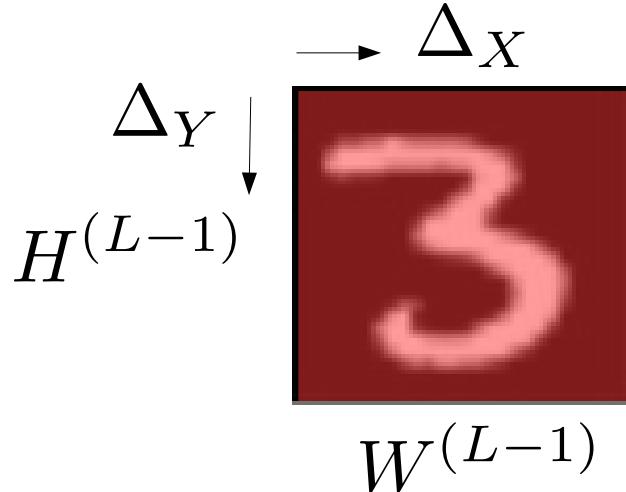
- If images have size  $H^{(L-1)}, W^{(L-1)}$ , filters are  $f_X, f_Y$  and strides are  $\Delta_X, \Delta_Y$ : how many filters do we need to cover the whole image?





# ConvLayers: quiz

- If images have size  $H^{(L-1)}, W^{(L-1)}$ , filters are  $f_X, f_Y$  and strides are  $\Delta_X, \Delta_Y$ : how many filters do we need to cover the whole image?



$$(1 + \frac{H^{(L-1)} - f_Y}{\Delta_Y}) \times (1 + \frac{W^{(L-1)} - f_X}{\Delta_X})$$



# ConvLayers: multiple filters and channels

- Number of filters determines size of next layer:

$$Z^{(L)} = \left(1 + \frac{H^{(L-1)} - f_Y}{\Delta_Y}\right) \times \left(1 + \frac{W^{(L-1)} - f_X}{\Delta_X}\right) \times 1$$

- Now: for each filter position, apply  $C^{(L)}$  filters instead of a single one → next layer has size

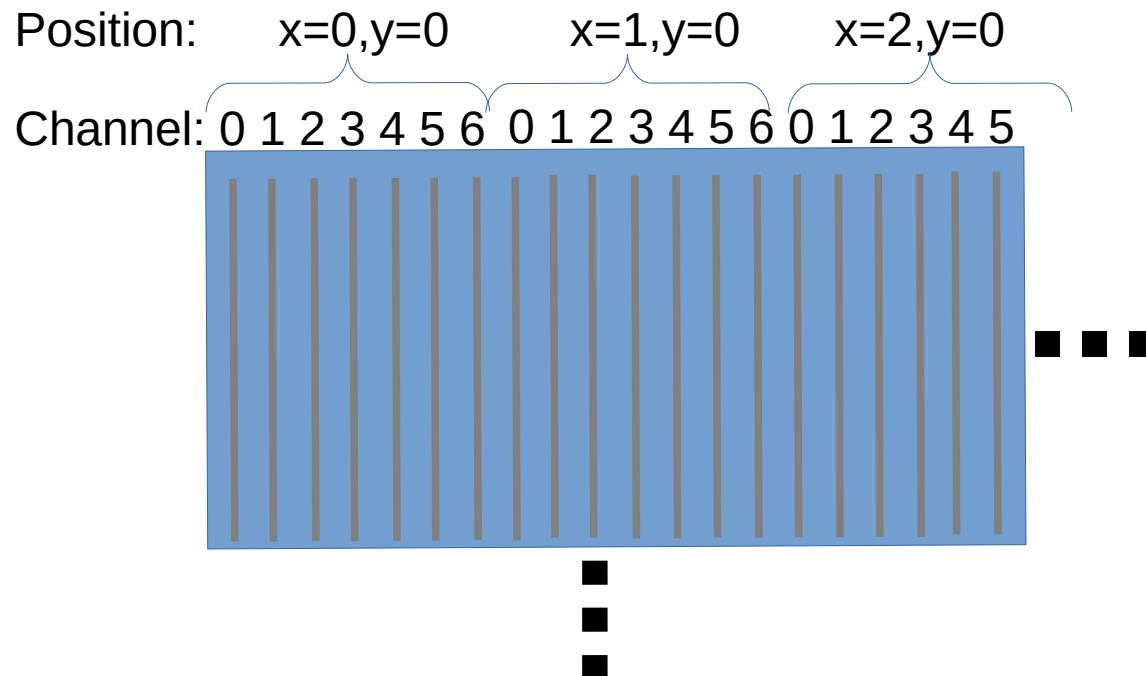
$$Z^{(L)} = \left(1 + \frac{H^{(L-1)} - f_Y}{\Delta_Y}\right) \times \left(1 + \frac{W^{(L-1)} - f_X}{\Delta_X}\right) \times C^{(L)}$$

- New **channel** dimension of size  $C^{(L)}$



# ConvLayers: weight-sharing

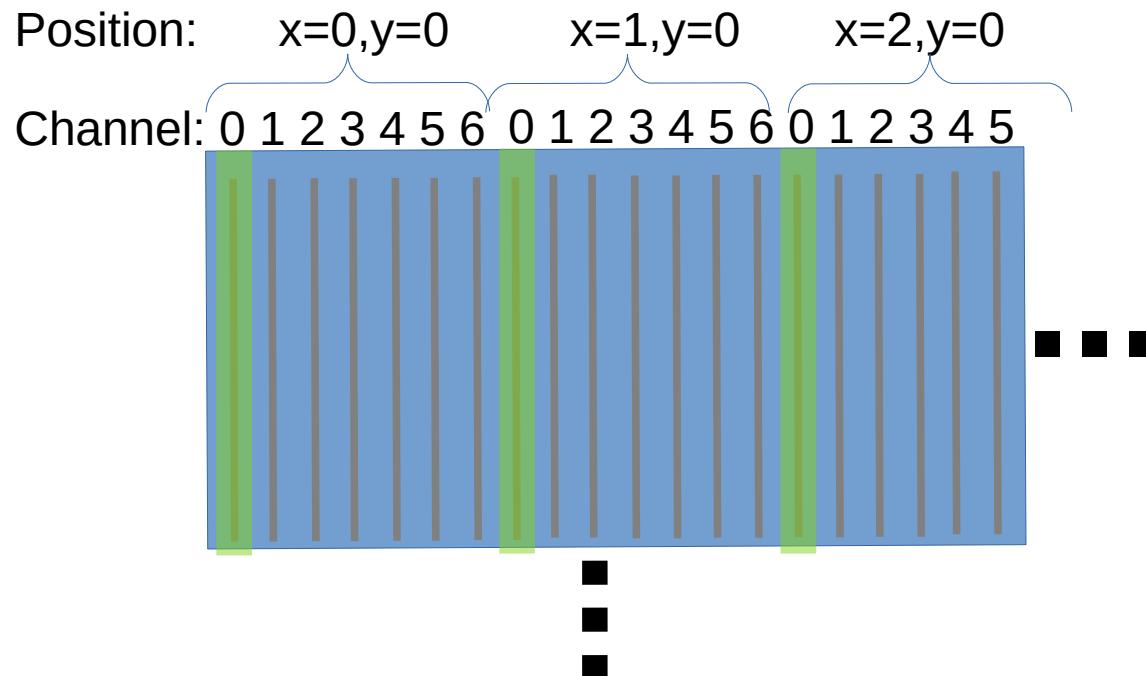
- Filters = rows of weight matrix are shared between same channel indices





# ConvLayers: weight-sharing

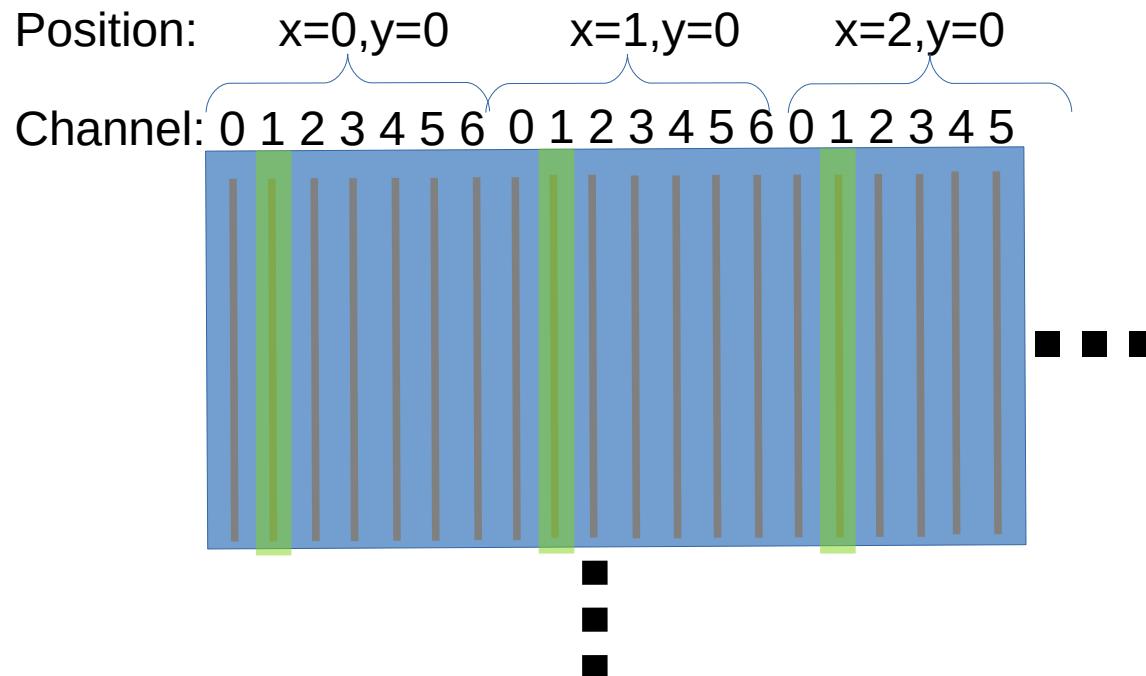
- Filters = rows of weight matrix are **shared** between same channel indices





# ConvLayers: weight-sharing

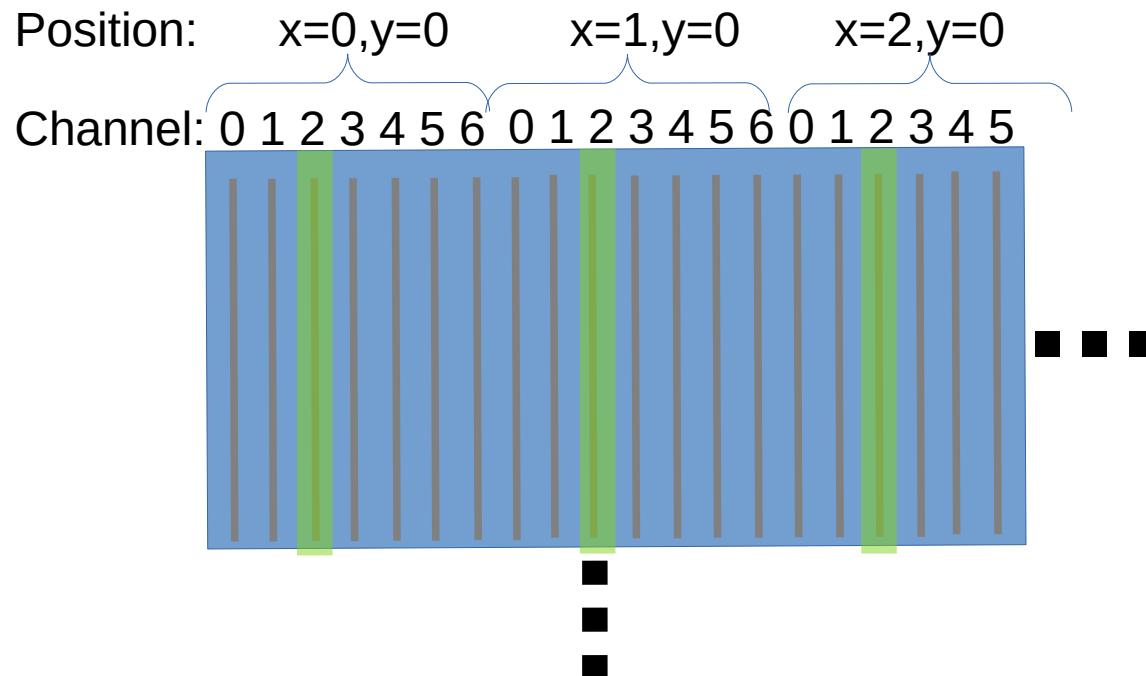
- Filters = rows of weight matrix are **shared** between same channel indices





# ConvLayers: weight-sharing

- Filters = rows of weight matrix are **shared** between same channel indices





# ConvLayers: weight-sharing

- Quiz: how many independent filters does a conv. Layer have a) with and b) without weight sharing?
- Let parameters be:

$$f_X = f_Y = 5, \Delta_X = \Delta_Y = 3, C^{(L)} = 10$$
$$H^{(L-1)} = 20, W^{(L-1)} = 20, C^{(L-1)} = 3$$



# ConvLayers: weight-sharing

- Quiz: how many independent filters does a conv. Layer have a) with and b) without weight sharing?

- Let parameters be:

$$f_X = f_Y = 5, \Delta_X = \Delta_Y = 3, C^{(L)} = 10$$

$$H^{(L-1)} = 20, W^{(L-1)} = 20, C^{(L-1)} = 3$$

- a)  $6 \times 6 \times 10 = 360$       b) 10  
→ weight sharing reduces parameters!!



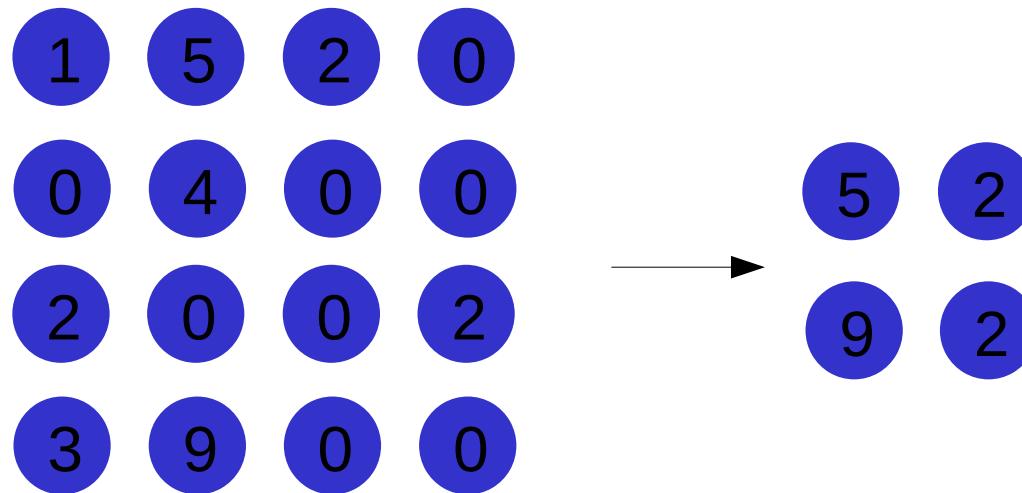
# Typical CNN transfer function: Max-Pooling/subsampling

- Used directly after a convolutional layer
  - preceding layer activities can be seen as  
 $N^{(L-1)} \times H^{(L-1)} \times W^{(L-1)} \times C^{(L-1)}$
- each channel is subdivided into non-overlapping squares of size  $k \times k$  ( $k$ : kernel size)
- just keep maximal activity from each kernel
  - strong size reduction!



# Max-Pooling/subsampling

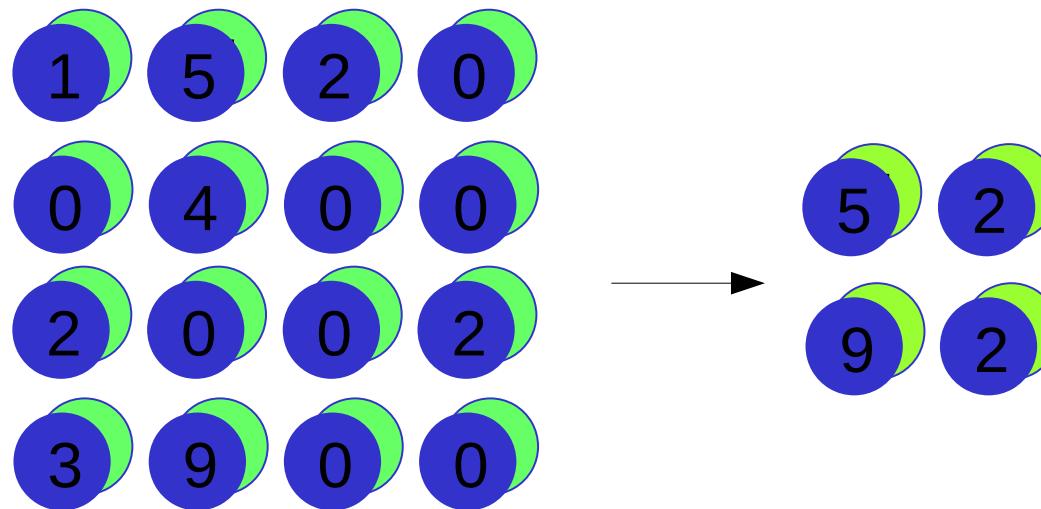
- Example for a single channel and kernel size 2:





# Max-Pooling/subsampling

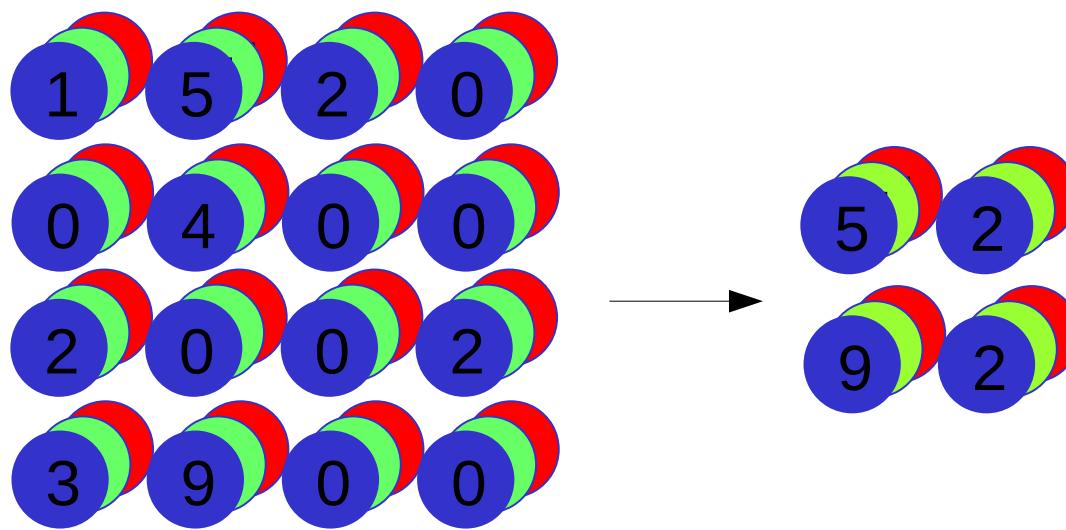
- Example for two channels and kernel size 2:





# Max-Pooling/subsampling

- Example for three channels and kernel size 2:





# Cut: Q&A



# CNN structure

- Mostly used for classifying RGB images
- Typical structure:
  - sequence of (conv+ReLU)
  - flatten last conv. Layer output to 2D shape
  - sequence of affine + ReLU
  - softmax
- Many details, most notably **padding** for conv. layers: enlarge input layer with zeros so that activities (for stride 1) are same size as input

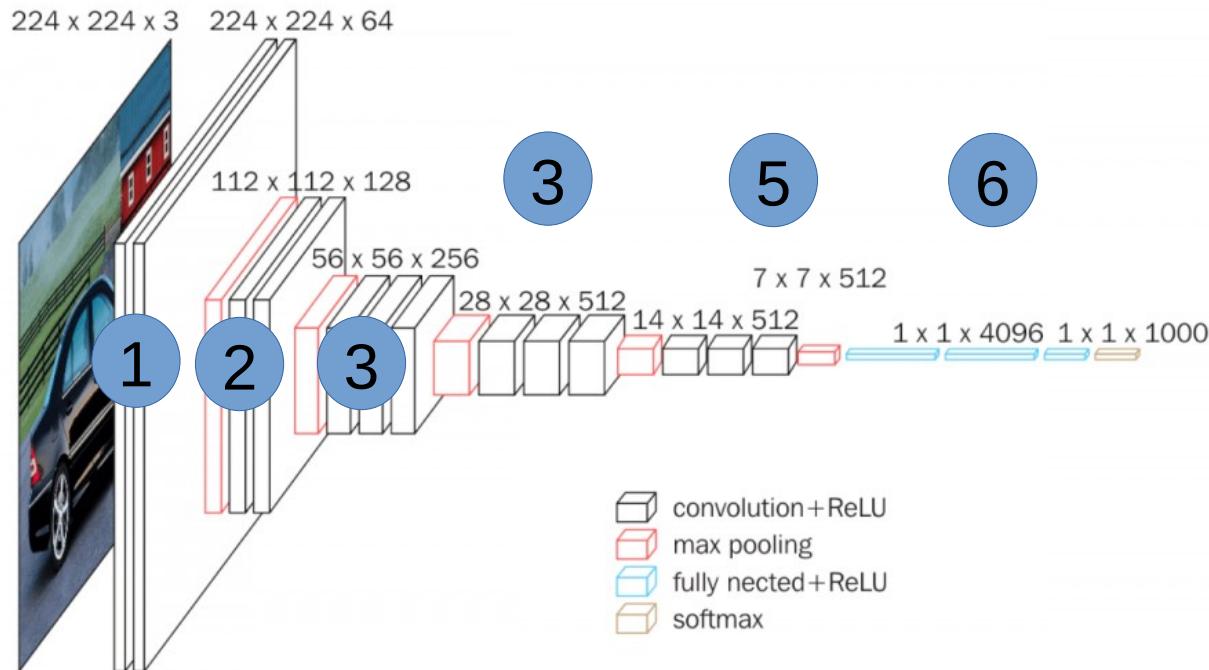


# VGG-16

- Proposed in 2016
- Very competitive in ImageNet benchmark
- Combination of convLayers, max-pooling, ReLU and fully-connected (dense) layers
- (Almost) linear softmax MC on top
- ?? parameters

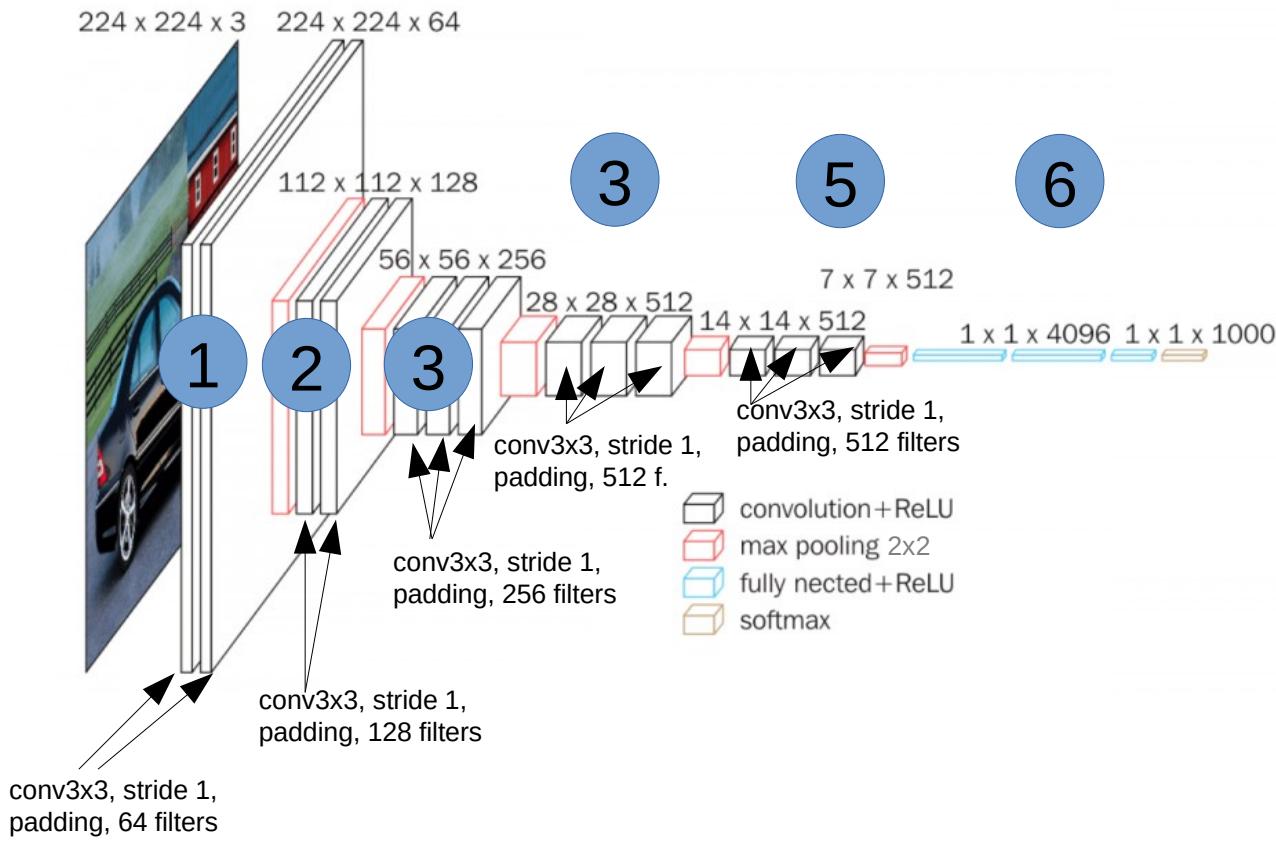


# VGG-16





# VGG-16 topology





# Quiz: parameters

- How many params in each layer of block 1,2,3,4,5,6?
- How many parameters in total for each block?
- How many parameters in the whole CNN?



# Cut: Q&A



# Project



# Project description

- Duration: 2-3 weeks
- Overall goal: train a classifier on data I give you!
- What you get:
  - zip file with png images
  - label is encoded in file name
- Expected results: CNN classifier that performs well on a holdout dataset that I will provide



# Project description

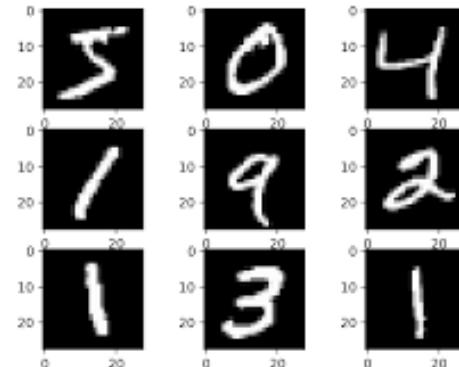
- Details:
  - Create a Python script, e.g., cnn.py
  - Invocation:

```
python3 cnn.py <path_to_images> imgW imgH
 imgC train|test
```
  - If parameter 4 is “train” → train DNN on images, save weights
  - If parameter 4 is “test” → load weights, compute accuracy on provided image data only



# Datasets

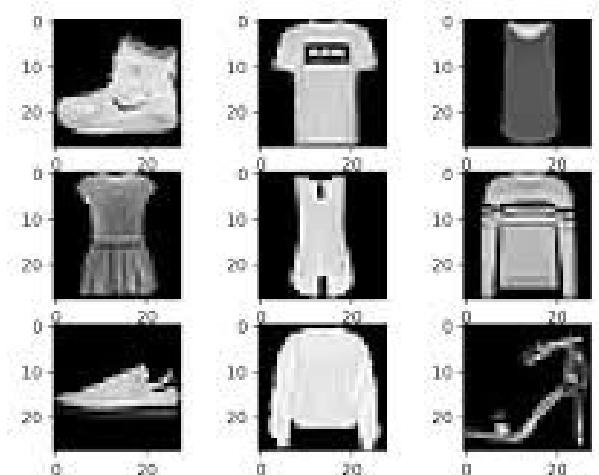
- Nr classes: find out!
- Datasets: MNIST
  - 28x28 mono
  - 70.000 images
  - expected accuracy: ~97%





# Datasets

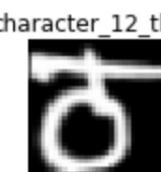
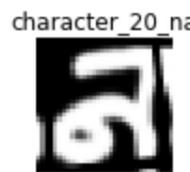
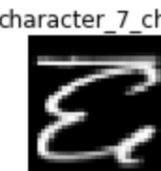
- Number of classes: find out!
- Datasets: FashionMNIST
  - 28x28 mono
  - 70.000 images
  - expected accuracy: ~ 88%





# Datasets

- Nr of classes: find out!
- Datasets: Devanagari
  - 32x32 mono
  - 22.000 images
  - expected accuracy: ~ 98%





# What you need to do

- Read images into numpy, extract label from file names. Take image path and size (W/H/C) from command line (using `sys.argv`)
- Data analysis and preprocessing, remove problems from data
- Data shuffling, train/test split
- Train/evaluate DNN



# Machine Learning

Important practical aspects of DNNs

Alexander Gepperth, January 2022



# Today

- Key weaknesses of DNNs + proofs
- Simplified construction of DNNs/CNNs with keras



# Fundamental issues of DNNs

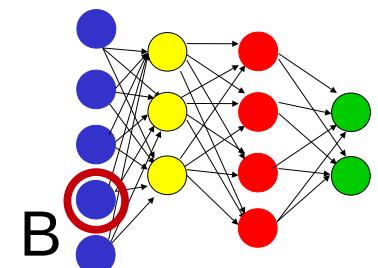
- DNNs are very sensitive to input ranges
- DNNs are sensitive to class imbalance
- DNNs are prone to catastrophic forgetting
- (DNNs may suffer from adversarial samples)



# DNNs are sensitive to input ranges

- Assume problem in which all input components  $a_i^{(0)}$  are in range 0...1 except one:  $a_B^{(0)}$  which is in range 0...100
- Compute derivatives of loss w.r.t any first hidden layer weight:

$$\frac{\partial \tilde{\mathcal{L}}}{\partial W_{ij}^{(1)}} =$$

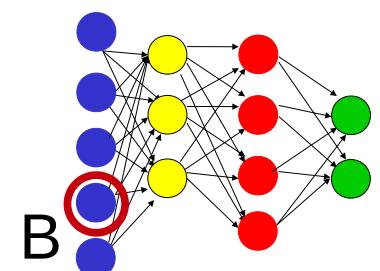




# DNNs are sensitive to input ranges

- Assume problem in which all input components  $a_i^{(0)}$  are in range 0...1 except one:  $a_B^{(0)}$  which is in range 0...100
- Compute derivatives of loss w.r.t any first hidden layer weight:

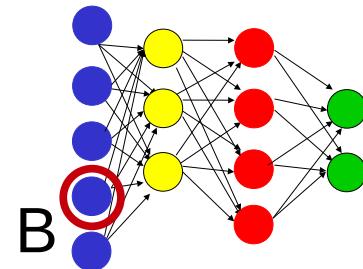
$$\frac{\partial \tilde{\mathcal{L}}}{\partial W_{ij}^{(1)}} = \sum_k \frac{\partial \tilde{\mathcal{L}}}{\partial a_k^{(1)}} \frac{\partial a_k^{(1)}}{\partial W_{ij}^{(1)}}$$





# DNNs are sensitive to input ranges

- Assume problem in which all input components  $a_i^{(0)}$  are in range 0...1 except one:  $a_B^{(0)}$  which is in range 0...100
- Compute derivatives of loss w.r.t any first hidden layer weight:



$$\frac{\partial \tilde{\mathcal{L}}}{\partial W_{ij}^{(1)}} = \sum_k \frac{\partial \tilde{\mathcal{L}}}{\partial a_k^{(1)}} \frac{\partial a_k^{(1)}}{\partial W_{ij}^{(1)}} = \frac{\partial \tilde{\mathcal{L}}}{\partial a_j^{(1)}} a_i^{(0)}$$



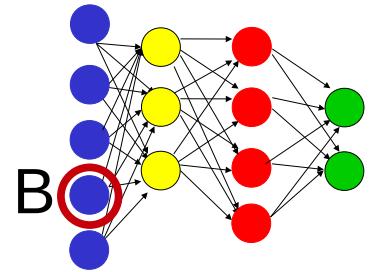
# DNNs are sensitive to input ranges

- Result:

$$\frac{\partial \tilde{\mathcal{L}}}{\partial W_{ij}^{(1)}} = \frac{\partial \tilde{\mathcal{L}}}{\partial a_j^{(1)}} a_i^{(0)} \quad \text{or} \quad \frac{\partial \mathcal{L}}{\partial W_{ij}^{(1)}} = \frac{1}{N} \sum_n \frac{\partial \mathcal{L}}{\partial A_{nj}^{(1)}} A_{ni}^{(0)}$$

- Interpretation:

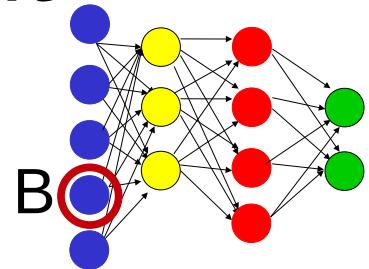
- derivatives of loss w.r.t. weight  $W_{ij}^{(1)}$  much higher if  $j = B$
- weights  $W_{iB}^{(1)}$  grow larger than all others
- activity in next layer eventually dominated by  $a_B^{(0)}$





# DNNs are sensitive to input ranges

- **Consequence:** need to normalize all input components to same range!





# DNNs are sensitive to class imbalance

- Consider a single weight  $W_{ij}^{(X)}$  in a DNN trained on two classes
  - assume that derivative  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  is -1 for class 1
  - assume that derivative  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  is +1 for class 2
- what is value of  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$        $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$ 
  - if both classes have frequencies  $z=0.5$  ?
  - for class frequencies  $z$  and  $1-z$  ?



# DNNs are sensitive to class imbalance

- Consider a single weight  $W_{ij}^{(X)}$  in a DNN trained on two classes
  - assume that derivative  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  is -1 for class 1
  - assume that derivative  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  is +1 for class 2
- what is value of  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$        $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$ 
  - if both classes have frequencies  $z=0.5$  ? **0**
  - for class frequencies  $z$  and  $1-z$  ? **1-2z**



# DNNs are sensitive to class imbalance

- In other words: different classes contribute to all gradients in proportion to their frequencies!
- more frequent classes have a higher impact!
- another way to look at this: what is classification accuracy for a binary classifier that always chooses class 1?
  - balanced classes?
  - unbalanced classes  $z$  and  $1-z$  ?



# DNNs are sensitive to class imbalance

- In other words: different classes contribute to all gradients in proportion to their frequencies!
- more frequent classes have a higher impact!
- another way to look at this: what is classification accuracy for a binary classifier that always chooses class 1?
  - balanced classes? **50% wrong**
  - unbalanced classes  $z$  and  $1-z$ ?  **$z\%$  wrong**



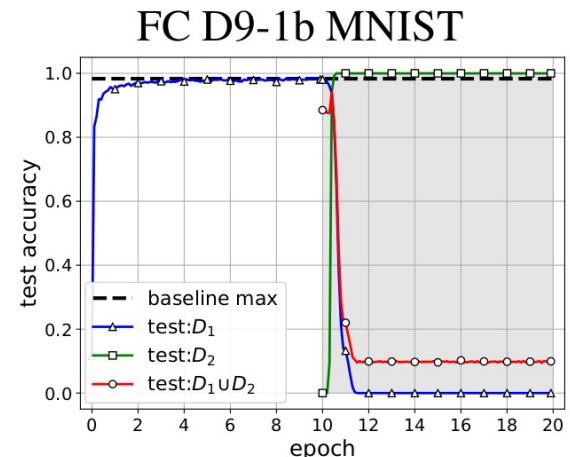
DNNs are sensitive to class imbalance

- **Consequence:** need balanced classes!
- If impossible: perform **oversampling**
  - determine samples-per-class value  $T$
  - randomly draw  $T$  samples from each class  
(even if there are more/less than  $T$ )
  - merge all drawn samples for training  
→ balanced dataset!
- $T$  is usually average or max. sample count



# DNNs are prone to catastrophic forgetting

- Assume a DNN is trained only on classes 0-8
- after successful training (evaluation on test set  $\mathbf{D}_1$ ):
  - retrain with samples from class 9, test on test set  $\mathbf{D}_2$
  - in parallel: test classification accuracy on  $\mathbf{D}_1$
- Catastrophic forgetting: classes 0-8 are forgotten quickly!



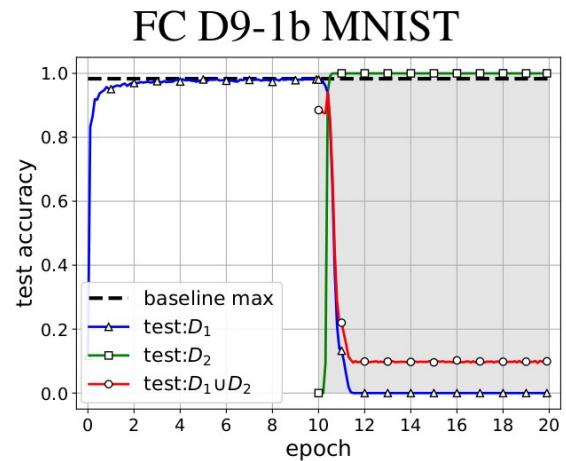


# DNNs are prone to catastrophic forgetting

- Why CF?
- consider loss CE function:

$$\mathcal{L}^{CE} = -\frac{1}{N} \sum_{ni} \log(A_{ni}^{(O)}) T_{ni}$$

- loss contains only samples from current batch (or mini-batch): if those are only from class 9  $\rightarrow$  classes 0-8 are disregarded!
- or:  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  depends only on samples from current batch!  
So  $\frac{\partial \mathcal{L}}{\partial W_{ij}^{(X)}}$  weights are adapted regardless of 0-8!

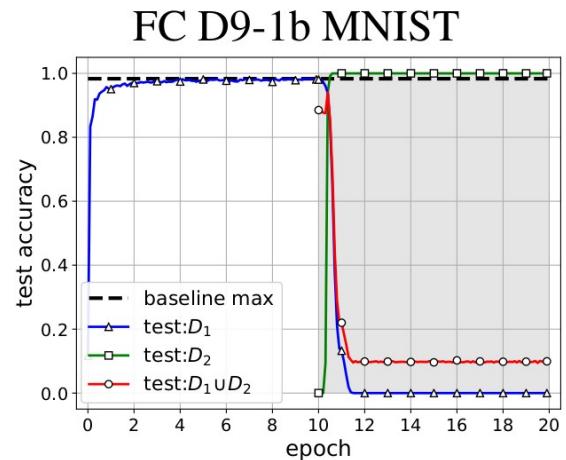




DNNs are prone to catastrophic forgetting

- **Consequence:**

- mini-batches must be drawn **randomly** from training data!
- if classes are balanced, each **mini-batch** will contain all classes with same probability
- randomly shuffle data once, and draw consecutive mini-batches later





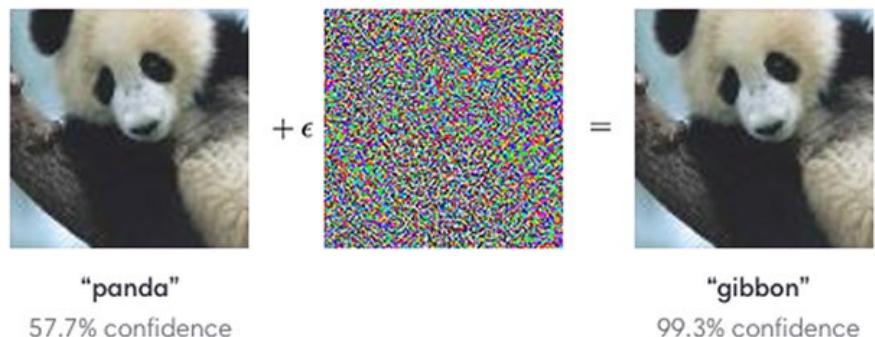
# DNNs may suffer from adversarial samples

- Big surprise in 2016: discovery of adversarial samples
- suppose a DNN is trained successfully
- then: compute  $\frac{\partial \mathcal{L}}{\partial a_i^{(0)}}$  **(what's that??)**



# DNNs may suffer from adversarial samples

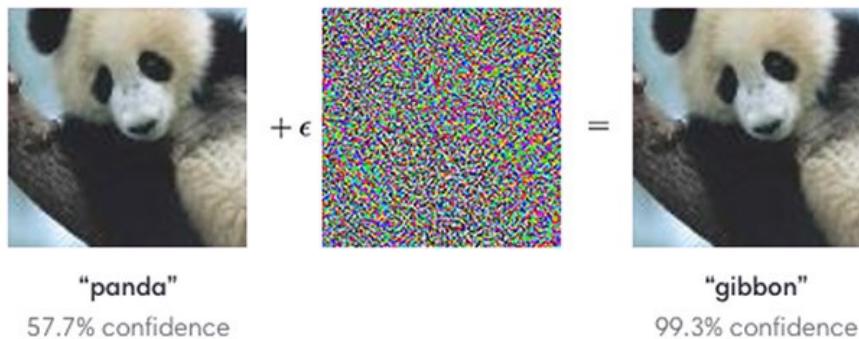
- Big surprise in 2016: discovery of adversarial samples
- suppose a DNN is trained successfully
- then: compute  $\frac{\partial \mathcal{L}}{\partial a_i^{(0)}}$  **(deriv. of loss w.r.t. input)**
- perform gradient ascent on inputs to **increase** loss while leaving weights unchanged
- leads to samples that are very similar, yet get assigned another class!





DNNs may suffer from adversarial samples

- **Consequence:** add adversarial samples to training data if necessary (adversarial training)!





# CUT: Q&A



# DNNs with keras



# tf.keras

- Keras: sub-package of TF2
- Goal: simple, portable construction and training of DNNs
- Especially: gradient computation and training hidden from user (no GradientTape)
- But: details **can** be controlled by experts



# tf.keras and the shortest linear classifier code EVER

```
model = tf.keras.Sequential() ;
model.add(tf.keras.layers.Dense(10)) ;
model.add(tf.keras.layers.Softmax()) ;
model.compile(optimizer=tf.keras.optimizers.Adam(),
 loss = tf.keras.losses.CategoricalCrossentropy(),
 metrics = tf.keras.metrics.CategoricalAccuracy()) ;
model.fit(traind,trainl,epochs=1) ;
metrics = model.evaluate(testd, testl) ;
```



# The Sequential model for constructing CNNs/DNNs

- The `tf.keras.Sequential` class describes a standard feed-forward CNN/DNN without shortcuts or recurrence
- A keras model is an **aggregation of instances** of
  - layers: class in `tf.keras.layers`
  - a loss: classes in `tf.keras.losses`
  - a SGD optimizer: classes in `tf.keras.optimizers`
  - optionally, one or more performance metrics: classes in `tf.keras.metrics`



# Creating a Sequential model

- Instantiate
- Define model by adding `tf.keras.layers` instances with `.add()`
- call `.compile` and provide instances of loss, optimizer and metrics (optionally)
- Model is ready for training!



# Training a Sequential model

- Using the method `.fit(...)`
- Common arguments:
  - `x, y`: training data and labels. Can be `np.array`, `tf.Tensor`, `tf.data.Dataset`, generators, ...
  - `epochs`: passes through training data
  - `batch_size`: mini-batch size
  - `validation_data`: validation(test) data, tuple of (`x,y`)
  - `validation_split`: percentage for train/test split
  - `validation_freq`: perform evaluation on test data every n epochs



# Sequential example

- Demo and code in E-Learning



# Keras layer classes

- Keras provides all CNN/DNN layers:
  - affine: `tf.keras.layers.Dense`
  - ReLU: `tf.keras.layers.ReLU`
  - max-Pooling: `tf.keras.layers.MaxPool2D`
  - conv-Layer: `tf.keras.layers.Conv2D`
  - Softmax: `tf.keras.layers.Softmax`



# Keras loss functions

- Keras provides all relevant loss functions as separate classes in package `tf.keras.losses`
- CE with one-hot encoded targets:  
`CategoricalCrossEntropy`.  
By default, assumes that softmax has been applied. If not, applies it (using the `from_logits` argument in constructor)
- CE with scalar labels: `SparseCategoricalCrossEntropy`
- MSE: `MeanSquaredError`



# Keras Metrics

- Keras provides all relevant metrics for evaluating a CNN/DNN.
  - we discuss classification accuracy only:  
`tf.keras.metrics.CategoricalAccuracy()`



# Keras Optimizers

- Keras provides many pre-defined optimizers for performing SGD
- Adam is usually best
- Important classes:
  - Plain SGD: `tf.keras.optimizers.SGD`
  - Adam: `tf.keras.optimizers.Adam`



# Keras layers: affine

- Typically instantiated as:

```
tf.keras.layers.Dense(num_elements)
```



# Keras layers: ReLU

- Typically instantiated as:

```
tf.keras.layers.ReLU()
```



# Keras layers: convolutional layer

- Typically instantiated as:

```
tf.keras.layers.Conv2D(filters,
 kernel_size,
 strides = (1,1),
 ...)
```



# Keras layers: max-pooling layer

- Typically instantiated as:

```
tf.keras.layers.MaxPool2D(
 kernel_size,
 ...)
```



# Keras layers: softmax layer

- Typically instantiated as:

```
tf.keras.layers.Softmax()
```



# What can we do with a trained Sequential model?

- Assume we have trained a Sequential model instance, e.g., seq\_model
- Apply to data by using model as a callable:  
`predicted_classes = seq_model(testd)`
- Evaluate using provided metrics:  
`m_values = seq_model.evaluate(testd)`
- Save entire model to disk: `seq_model.save(..)`
- Save all parameters to disk:  
`seq_model.saveWeights()`



# Differences between applying and evaluating a model

- Applying `seq_model` to data:
  - produces the last layer activities (no loss etc.)
  - is performed for the provided data in a single step
    - memory issues for large data
- Calling `seq_model.evaluate()`:
  - is performed in mini-batches
    - no memory issues
  - produces a list of performance metrics



# Customizing `tf.keras` models

- Important concept: callbacks = actions to be executed at specific points
  - before/after processing an epoch
  - before/after processing a mini-batch
  - before/after training
- To provide callbacks, we instantiate subclasses of `tf.keras.callbacks.Callback` and provide them to a model when calling `.fit()`



# Customizing `tf.keras` models

- Object-oriented design using aggregation
  - all involved classes can be sub-classed to modify their behavior
- Example: `sub.class tf.keras.Sequential`, e.g., for overwriting `.fit` or `.train_step()`
- We can create own classes for
  - losses: subclass `tf.keras.losses.Loss`
  - optimizers:  
`subclass tf.keras.optimizers.Optimizer`
  - metrics: subclass `tf.keras.metrics.Metric`





# Project hints

- Read data to numpy
- Visualize and correct, no oversampling
- Construct a DNN/CNN: try LeNet-5 or similar architecture
- Measure classification accuracy, adapt learning rates, optimizers, layer sizes, ...
- Perform oversampling



# Machine Learning

Classification measures & probability

Alexander Gepperth, January 2022



# Today

- Discrete probability theory
- ... applied to the evaluation of classifiers!



# Elements of discrete probability theory



# (Discrete) probabilities

- Situation: experiment with finite number  $E$  of possible outcomes
- Repeated  $N$  times
- Outcomes: mutually exclusive
- Description by random variable  $X$
- Empirical probability for outcome  $A$ :  $x = a$

$$p(A) \equiv p(X = a) = \frac{\#(A)}{N} = \frac{\#(X = a)}{N}$$

count



# Examples

**x:** 1 2 0 3 4 7 0 0 0 1 2 3 4

- $A: X \geq 4 \rightarrow p(A) =$
- $A: 0 <= X <= 1 \rightarrow p(A) =$
- $A: X = 1 \rightarrow p(A) =$
- $A: x^2 > 4 \rightarrow p(A) =$



# Examples

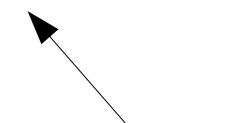
**x:** 1 2 0 3 4 7 0 0 0 1 2 3 4

- $A: X \geq 4 \rightarrow p(A) = 3 / 13$
- $A: 0 <= X <= 1 \rightarrow p(A) = 6 / 13$
- $A: X = 1 \rightarrow p(A) = 2 / 13$
- $A: x^2 > 4 \rightarrow p(A) = 5 / 13$



# Joint probabilities

- Now: 2 experiments, with  $N$  outcomes each
- Described by 2 random variables (RVs), e.g.,  $X, Y$
- Described by joint probabilities
- Empirical probability for joint event  $A, B : X = a, Y = b$   
given by  $p(A, B) \equiv \frac{\#(A, B)}{N}$

logical “and”  




# Example: joint probabilities

**x:** 1 2 0 3 4 7 0 0 0 1 2 3 4

**y:** 0 0 1 1 0 3 0 2 2 2 0 1 0

$$p(A, B) \begin{cases} p(X = 2, Y = 1) \\ p(X = 2, Y = 0) \\ p(X = 7, Y = 2) \end{cases}$$



# Example: joint probabilities

**x:** 1 2 0 3 4 7 0 0 0 1 2 3 4

**y:** 0 0 1 1 0 3 0 2 2 2 0 1 0

$$p(A, B) \begin{cases} p(X = 2, Y = 1) = \frac{0}{13} \\ p(X = 2, Y = 0) = \frac{2}{13} \\ p(X = 7, Y = 2) = \frac{0}{13} \end{cases}$$



# Conditional probabilities

- In a 2-experiment setting, it often makes sense to restrict the space of possible outcomes
- We ask: what is the probability of outcome  $A: X=1$  when restricted to outcomes with  $B: Y=2$  ?
- Restrict space of outcomes to those satisfying a certain condition
- Notation:

$p(A|B)$

Outcome of interest  
e.g.,  $X = 1$

restriction/condition, e.g.,  
 $Y = 2$

Two red arrows point from the descriptive text below to the 'A' and the '|B' in the mathematical notation.



# Example 1

**X:** 1 2 0 3 4 7 0 0 0 1 2 3 4

**Y:** 0 0 1 1 0 3 0 2 2 2 1 1 0

$$p(A|B) \left\{ \begin{array}{l} p(X = 2|Y = 1) \\ p(X = 2|Y = 0) \\ p(X = 7|Y = 2) \end{array} \right.$$

Procedure: 1) remove all outcomes not satisfying condition  $\rightarrow \#(B)$  remaining  
2) how many of the remaining outcomes correspond to the outcome of interest?  $\rightarrow \#(A,B)$   
3)  $\#(A,B) / \#(B)$



# Example 1

**X:** 1 2 0 3 4 7 0 0 0 1 2 3 4

**Y:** 0 0 1 1 0 3 0 2 2 2 1 1 0

$$p(A|B) \left\{ \begin{array}{l} p(X = 2|Y = 1) = \frac{1}{4} \\ p(X = 2|Y = 0) = \frac{1}{5} \\ p(X = 7|Y = 2) = \frac{0}{3} \end{array} \right.$$

Procedure: 1) remove all outcomes not satisfying condition  $\rightarrow \#(B)$  remaining  
2) how many of the remaining outcomes correspond to the outcome of interest?  $\rightarrow \#(A,B)$   
3)  $\#(A,B) / \#(B)$



# Computing conditional probabilities

- Algorithm for computing  $p(A/B)$ :
  - count outcomes satisfying  $B$ :  $\#(B)$
  - count outcomes satisfying  $A, B$ :  $\#(A, B)$
  - compute  $p(A/B) = \#(A, B) / \#(B)$
- Elementary algebra:

$$p(A|B) = \frac{\#(A, B)}{\#(B)} = \frac{\#(A, B)}{\#(B)} \frac{N}{N} = \frac{p(A, B)}{p(B)}$$



# CUT



# Demonstration 1

**X:** 1 3 3 3 4 0 0 0 0 0 1 3 1

**Y:** 2 3 3 1 5 3 0 2 2 2 1 1 0

Compute:

- $p(X=3)$
- $p(X=3, Y=3)$
- $p(X=3|Y=3)$



# Demonstration 2

**X:** 1 3 3 3 4 0 0 0 0 1 3 1

**Y:** 2 3 3 1 5 3 0 2 2 2 1 1 0

Compute:

- $p(Y=3)$
- $p(X=3, Y=3)$
- $p(Y=3|X=3)$



# CUT



# Classifiers

- Known classification models:
  - Linear classifier
  - DNN/CNN classifier
- Estimate class from model outputs  $Y = \vec{f}(X)$
- True class given by target matrix (one-hot format!)  $T$
- Scalar classes given by:

$$\hat{y}_i = \operatorname{argmax}_k Y_{ik}$$

$$\hat{t}_i = \operatorname{argmax}_k T_{ik}$$

Math convention:  $\geq 1$





# Performance measures for MC classification

- MC classifier: treat model outputs  $\hat{y}_i$  and targets  $\hat{t}_i$  as independent experimental outcomes

$\hat{y}_i:$  1 2 1 3 4 7 1 4 6 1 2 3 4

$\hat{t}_i:$  2 2 1 3 4 3 1 2 2 2 1 1 4

Which  
classifications  
are correct?  
Incorrect?

- Same situation as in discrete probability theory
- How to summarize a classifier's performance succinctly?
- How do we query for certain results?



# Representation of classification outcomes in a confusion matrix

- Confusion matrix:  $c_{jk} \equiv \#(\hat{t} = j, \hat{y} = k)$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \end{array} \right)$$

- Entry at position  $(j,k)$ : how often did  $\hat{t} = j, \hat{y} = k$  occur?



# Interpretation of the confusion matrix

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \dots \end{array} \right)$$

- (Off-)diagonal elements: (in)correct classifications
- Sum of all entries: total number of classifications/samples  $N$



# How to extract information from a confusion matrix

- Classification error:  $\chi = p(\hat{y} \neq \hat{t}) = \frac{\sum_{i,j=1, i \neq j}^K c_{ij}}{N} = 1 - \frac{\sum_{i=1}^K c_{ii}}{N}$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \end{array} \right)$$



# How to extract information from a confusion matrix

- Elementary probabilities:  $p(\hat{y} = k) = \frac{\sum_j c_{jk}}{N}$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & | \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & | \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & | \hat{t} = 3 \\ \dots & \dots & \dots & \dots & | \end{array} \right)$$



# How to extract information from a confusion matrix

- Elementary probabilities:  $p(\hat{t} = j) = \frac{\sum_k c_{jk}}{N}$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \end{array} \right)$$



# How to extract information from a confusion matrix

- Joint probabilities:  $p(\hat{y} = k, \hat{t} = j) = \frac{c_{jk}}{N}$

$$C = \left( \begin{array}{cccc|c} & \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & \\ \hline c_{11} & c_{12} & c_{13} & \dots & | \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & | \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & | \hat{t} = 3 \\ \dots & \dots & \dots & \dots & | \end{array} \right)$$



# How to extract information from a confusion matrix

- Conditional probabilities:  $p(\hat{y} = k | \hat{t} = j) = \frac{p(\hat{y} = k, \hat{t} = j)}{p(\hat{t} = j)}$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \end{array} \right)$$



# How to extract information from a confusion matrix

- Conditional probabilities:  $p(\hat{t} = j | \hat{y} = k) = \frac{p(\hat{t} = j, \hat{y} = k)}{p(\hat{y} = k)}$

$$C = \left( \begin{array}{cccc|c} \hat{y} = 1 & \hat{y} = 2 & \hat{y} = 3 & & \\ \hline c_{11} & c_{12} & c_{13} & \dots & \hat{t} = 1 \\ c_{21} & c_{22} & c_{23} & \dots & \hat{t} = 2 \\ c_{31} & c_{32} & c_{33} & \dots & \hat{t} = 3 \\ \dots & \dots & \dots & \dots & \end{array} \right)$$



# Complex example

- Compute:
  - #(samples), #(classes)
  - Classification error
  - Probability of incorrectly classifying samples from class 4
  - Probability of wrongly classifying a sample from class 2 as class 3
  - Probability of correct classifications for samples from class 1?

$$C = \begin{pmatrix} 5 & 3 & 2 & 0 \\ 1 & 5 & 1 & 3 \\ 0 & 1 & 7 & 2 \\ 4 & 1 & 0 & 5 \end{pmatrix}$$



# CUT: Q&A



# Binary classification

- Binary → only two classes: negative (1) and positive (2) class
- Confusion matrix becomes  $C = \begin{array}{cc} \hat{y} = 1 & \hat{y} = 2 \\ \hline tn & fp \\ fn & tp \end{array} \left| \begin{array}{l} \hat{t} = 1 \\ \hat{t} = 2 \end{array} \right.$
- negatives:  $n = \#(\hat{t} = 1)$
- positives  $p = \#(\hat{t} = 2)$
- negative rate:  $p_n = p(\hat{t} = 1) = n/N$
- positive rate:  $p_p = p(\hat{t} = 1) = p/N$



# Binary classification

- Binary → only two classes: negative(1) and positive(2) class
- Confusion matrix becomes  $C = \begin{pmatrix} \hat{y} = 1 & \hat{y} = 2 \\ tn & fp \\ fn & tp \end{pmatrix} \left| \begin{array}{l} \hat{t} = 1 \\ \hat{t} = 2 \end{array} \right.$
- true negatives:  $tn = \#(\hat{y} = 1, \hat{t} = 1)$
- false positives:  $fp = \#(\hat{y} = 2, \hat{t} = 1)$
- true positives:  $tp = \#(\hat{y} = 2, \hat{t} = 2)$
- false negatives:  $fn = \#(\hat{y} = 1, \hat{t} = 2)$



# Binary classification

- Binary → only two classes: negative(1) and positive(2) class
- Confusion matrix becomes  $C = \begin{pmatrix} \hat{y} = 1 & \hat{y} = 2 \\ tn & fp \\ fn & tp \end{pmatrix} \left| \begin{array}{l} \hat{t} = 1 \\ \hat{t} = 2 \end{array} \right.$
- true negative rate:  $tpr = p(\hat{y} = 1 | \hat{t} = 1)$
- false positive rate:  $fpr = p(\hat{y} = 2 | \hat{t} = 1)$
- true positive rate:  $tpr = p(\hat{y} = 2 | \hat{t} = 2)$
- false negative rate:  $fpr = p(\hat{y} = 1 | \hat{t} = 2)$



# Corollaries I

- Remember: conditional probabilities are normalized:  $\sum_a p(X = a|Y = b) = 1$
- Therefore:  $p(\hat{y} = 1|\hat{t} = 1) + p(\hat{y} = 2|\hat{t} = 1) = 1$   
 $p(\hat{y} = 1|\hat{t} = 2) + p(\hat{y} = 2|\hat{t} = 2) = 1$
- Or:



# Corollaries I

- Remember: conditional probabilities are normalized:  $\sum_a p(X = a|Y = b) = 1$
- Therefore:  $p(\hat{y} = 1|\hat{t} = 1) + p(\hat{y} = 2|\hat{t} = 1) = 1$   
 $p(\hat{y} = 1|\hat{t} = 2) + p(\hat{y} = 2|\hat{t} = 2) = 1$
- Or:  
 $tpr + fpr = 1$   
 $fnr + tpr = 1$



# Classification error

- What is the classification error?

$$E = p(\hat{y} \neq \hat{t})$$

$$= p(\hat{y} = 1, \hat{t} = 2) + p(\hat{y} = 2, \hat{t} = 1)$$

- Can we express this using cond. probs.?



# Classification error

- What is the classification error?

$$E = p(\hat{y} \neq \hat{t})$$

$$= p(\hat{y} = 1, \hat{t} = 2) + p(\hat{y} = 2, \hat{t} = 1)$$

- Can we express this using cond. probs.?

$$E = p(\hat{y} = 1, \hat{t} = 2) + p(\hat{y} = 2, \hat{t} = 1)$$

$$= p(\hat{y} = 1 | \hat{t} = 2)p(\hat{t} = 2) + p(\hat{y} = 2 | \hat{t} = 1)p(\hat{t} = 1)$$



# Classification error

- What is the classification error?

$$E = p(\hat{y} \neq \hat{t})$$

$$= p(\hat{y} = 1, \hat{t} = 2) + p(\hat{y} = 2, \hat{t} = 1)$$

- Can we express this using cond. probs.?

$$E = p(\hat{y} = 1, \hat{t} = 2) + p(\hat{y} = 2, \hat{t} = 1)$$

$$= p(\hat{y} = 1 | \hat{t} = 2)p(\hat{t} = 2) + p(\hat{y} = 2 | \hat{t} = 1)p(\hat{t} = 1)$$

$$= fnr * p_p + fpr * p_n$$



# Example: testing for COVID-19

- Why are  $fpr$ ,  $fnr$  relevant?
- Covid test: binary classifier, returns  $y_i \in \{1, 2\}$ 
  - negative: no COVID  $t_i = 1$
  - positive: COVID  $t_i = 2$
- “accuracy is 99%”, is that good?  
→ compute error for a test that always says “negative”!





# Example: testing for COVID-19

- Why are  $fpr$ ,  $fnr$  relevant?
- Covid test: binary classifier, returns  $y_i \in \{1, 2\}$ 
  - negative: no COVID  $t_i = 1$
  - positive: COVID  $t_i = 2$
- “accuracy is 99%”, is that good?
  - compute error for a test that always says “negative”!
- Since  $p_n = 0.99$  and  $p_p = 0.01$ :  $E = 1\% = fpr p_n + fnr p_p$ 
  - test with  $fpr = 0$  und  $fnr = 100\%$  achieves 1% error
  - very bad test!
  - $fpr, fnr$  are relevant,  $E$  is not enough for unbalanced data!





# Common aliases and measures

- Sensitivity/recall:

true positive rate  $tpr = p(\hat{y} = 2 | \hat{t} = 2)$

- Specificity/selectivity:

true negative rate  $tnr = p(\hat{y} = 1 | \hat{t} = 1)$

- Precision:  $p(\hat{t} = 2 | \hat{y} = 2) = \frac{tp}{fp + tp} = \frac{tpr}{fpr \frac{p_n}{p_p} + tpr}$

- Error  $E$  : previous slide

- Accuracy:  $1-E$



# Common aliases and measures

- Sensitivity/recall:

true positive rate  $tpr = p(\hat{y} = 2 | \hat{t} = 2)$

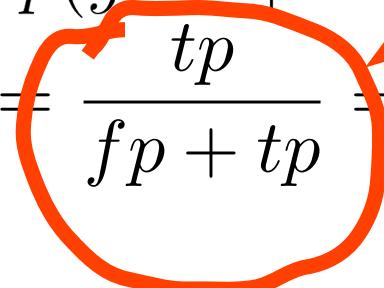
- Specificity/selectivity:

true negative rate  $tnr = p(\hat{y} = 1 | \hat{t} = 1)$

- Precision:  $p(\hat{t} = 2 | \hat{y} = 2) = \frac{tp}{fp + tp}$

$$= \frac{tpr}{fpr \frac{p_n}{p_p} + tpr}$$

from  
CM!



- Error  $E$  : previous slide

- Accuracy:  $1-E$



# Example 1

- A classifier is evaluated to having a precision of 0.8 and  $fpr=0.1$ . The frequencies of positive and negatives are 0.2 and 0.8, respectively.
- Compute  $tpr$ ,  $fpr$ ,  $tnr$ ,  $fnr$ ,  $E$ !



# Example 2

- A classifier is evaluated to having a precision of 0.8 and a recall of 0.5. The frequencies of positive and negatives are 0.2 and 0.8, respectively.
- Compute  $tpr$ ,  $fpr$ ,  $tnr$ ,  $fnr$ ,  $E!$



# Example 3

- Consider the experimental outcomes given by:  $C = \begin{pmatrix} 30 & 1 \\ 20 & 20 \end{pmatrix}$
- Compute:
  - $N, p_p, p_n, E$
  - $fnr, fpr$
  - precision, recall



# Machine Learning

Lecture 13: Sampling bias and other stuff

Alexander Gepperth, February 2022



# Today

- Technical test for online exam
- Discussion of project, demo code
- Discussion of exercise pool 2
- Interesting ML topic: sampling bias, ethical AI



# About data



# About data

- ML-Verfahren hängen kritisch von den Daten ab, die zum Training benutzt werden
- Probleme mit Daten:
  - unvollständig → Overfitting
  - tendenziös → "Sampling Bias"
  - zu komplex für das Modell → Underfitting
  - ...



# About data: theory

- We assume that we have data samples  $\vec{x}_i$  and targets  $\hat{t}_i$
- fundamental assumption: there **is** a “true” statistical relationship between data samples and targets:  $p(\hat{t}|\vec{x})$ 
  - if relationship were known  $\rightarrow$  problem solved!
  - In reality: unknown, so use M.L. to approximate it
- Another fundamental assumption: training data follow this “true” relationship



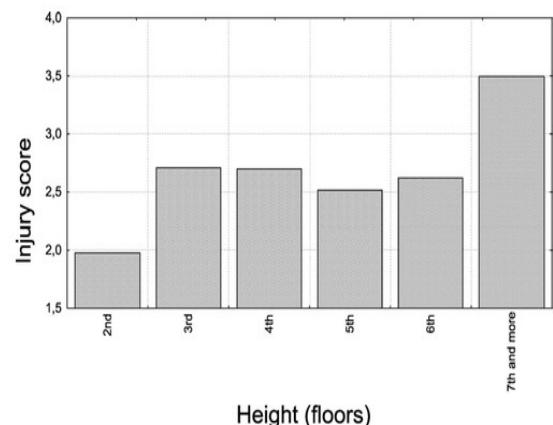
# About data: in practice

- Usually, the available training data  $\vec{x}_i \ \hat{t}_i$  do **not** follow this unknown distribution
- Consequence: ML does not model the true relationship between samples and targets
  - Sampling bias: bad data  $\rightarrow$  bad model
  - Overfitting: not enough data, or not everywhere
  - ...
- Can have important consequences for society!



# Sampling bias: cats

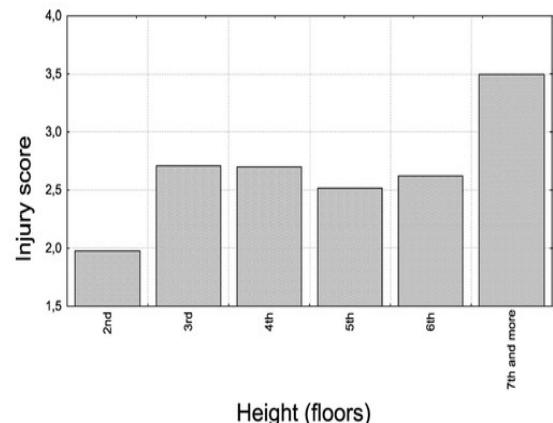
- High-rise syndrome:  
Empirical study of cats falling  
from skyscrapers (in NY)
- Collected data: height of fall  
vs. injury score (10 = death)
- Conclusion:





# Sampling bias: cats

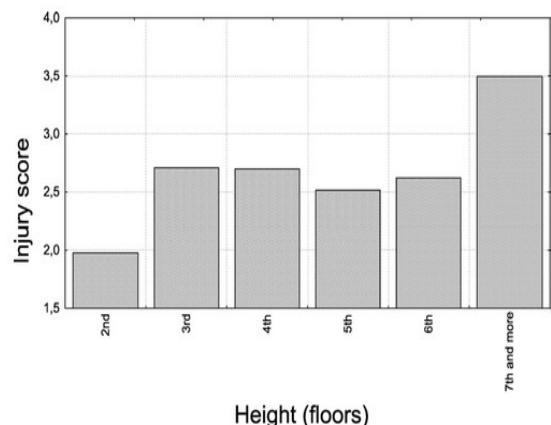
- High-rise syndrome:  
Empirical study of cats falling  
from skyscrapers (in NY)
- Collected data: height of fall  
vs. injury score (10 = death)
- Conclusion: cats can survive  
falls from great heights!





# Sampling bias: cats

- High-rise syndrome:  
Empirical study of cats falling  
from skyscrapers (in NY)
- Collected data: height of fall  
vs. injury score (10 = death)
- Conclusion: cats can survive  
falls from great heights!
- Careful: **dead cats not included!**





# Sampling bias: risk assessment in justice systems

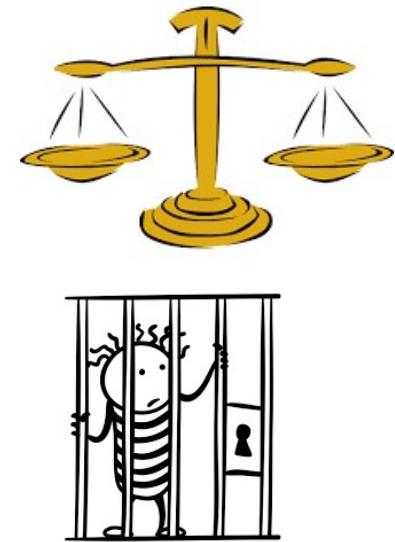
- Context: courts want to assess risk of relapse for convicted criminals





# Sampling bias: risk assessment in justice systems

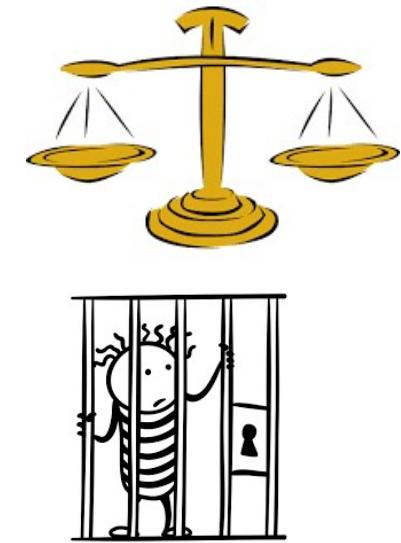
- Context: courts want to assess risk of relapse for convicted criminals
  - high risk: keep criminals in prison





# Sampling bias: risk assessment in justice systems

- Context: courts want to assess risk of relapse for convicted criminals
  - high risk: keep criminals in prison
  - low risk: release them on probation



## Ereigniskarte

Du kommst aus dem Gefängnis frei.

Diese Karte muss behalten werden,  
bis sie gebraucht oder verschenkt wird.



# Sampling bias: risk assessment in justice systems

- Context: courts want to assess risk of relapse for convicted criminals
  - high risk: keep criminals in prison
  - low risk: release them on probation
- Decision by ML: compute  $p(\hat{r} = \{0, 1\} | \vec{x})$
- Training data: past court records





# Sampling bias: risk assessment in justice systems

- Context: courts want to assess risk of relapse for convicted criminals
  - high risk: keep criminals in prison
  - low risk: release them on probation
- Decision by ML: compute  $p(\hat{r} = \{0, 1\} | \vec{x})$
- Training data: past court records
- **If data is biased → ML will be, too!**





# Sampling bias: vehicle detection

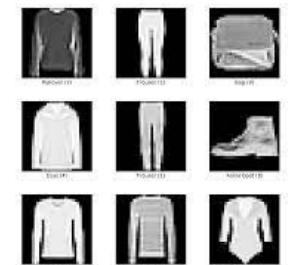
- Simple example: cars / background
- Problem: rather hard to obtain car images
- Non-car images are simple
  - non-cars may be over-represented
  - car class may be insufficiently covered





# Sampling bias: projects

- Always, class 0 was severely underrepresented  
→ another form of sampling bias!
- Yet, CMs look the same when performing oversampling. Why?
- $p(\hat{y} \neq \hat{t} | \hat{t} = 0) = \frac{\#(\hat{y} \neq \hat{t})}{\#(\hat{t} = 0)}$
- Should test on original data!





### Ereigniskarte

Du kommst aus dem Gefängnis frei.

Diese Karte muss behalten werden,  
bis sie gebraucht oder verschenkt wird.

# Sampling bias

- Obtaining data for training ML methods is a complex process
- Consequence: data may posses systematic biases!
- Consequence: ML will be biased!
- Ethical AI: automatic bias detection
- Google for “CEO”!!

