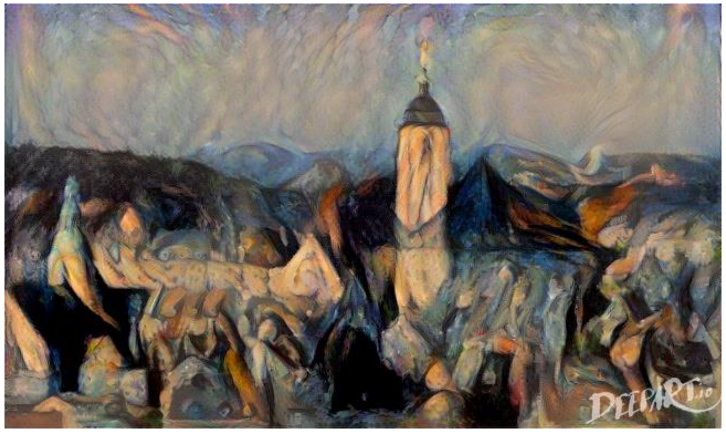
ZESS Lectures
Recent Advances in Machine Learning

State-of-the-art research in machine learning in various fields of applications

Lecture 1: Supervised Machine Learning Overview



Michael Möller – michael.moeller@uni-siegen.de

Machine Learning

Disclaimer: There will never be THE machine learning summary – the field is extremely broad and covers a lot of (sub-)disciplines.

By far the largest discipline can, however, be summarized under the name of *supervised* deep learning.

Supervised deep learning means that you have *training* examples of inputs and the desired output.

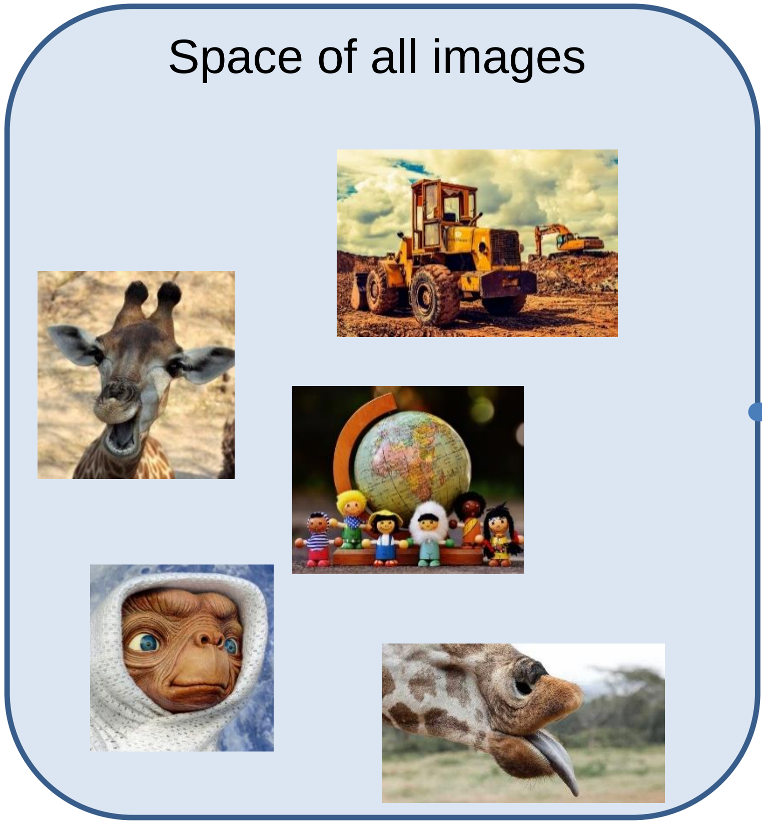
Most prominent examples for supervised deep learning applications are classifaction tasks.

Let‘s look at supervised machine learning in more details!

Machine Learning

In my perspective supervised machine learning is a nice word for function approximation

Assume there is an unknown function G that maps some kind of input data x to some kind of desired output y.



$$\xrightarrow{G}$$

Answer to the question if the image shows a giraffe

NO!

YES!

Machine Learning

In my perspective supervised machine learning is a nice word for function approximation

Assume there is an unknown function G that maps some kind of input data x to some kind of desired output y.

Assume we are given some evaluations of this (unknown) function G. This is what we call ***training data***!

Giraffe

No giraffe

No giraffe

No giraffe

Giraffe

$G\Big($ $\Big) = 0, \quad G\Big($ $\Big) = 1$

Machine Learning

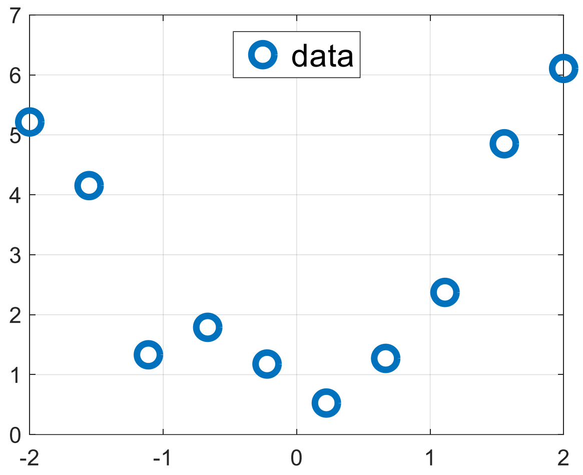
In my perspective supervised machine learning is a nice word for function approximation

Assume there is an unknown function G that maps some kind of input data x to some kind of desired output y.

Assume we are given some evaluations of this (unknown) function G. This is what we call ***training data***!

1. Choose a parameterized function $\mathcal{N}(x;\theta)$ in the hope that for the right choice of parameters θ it approximates the unknown function G well. We call $\mathcal{N}$ the ***network***, and sometimes refer to θ as the ***weights***.
2. Try to determine suitable weights θ in such a way that $\mathcal{N}(x_i;\theta) \approx y_i$ holds for all examples (x_i, y_i) from your training data set. This is referred to as ***training the network***.
3. Make try to ensure that both, the architecture as well as the training are chosen in such as way that the network makes good predictions during inference, i.e. on previously unseen data x: $\mathcal{N}(x;\theta) \approx G(x)$. We refer to this property as ***generalization***.

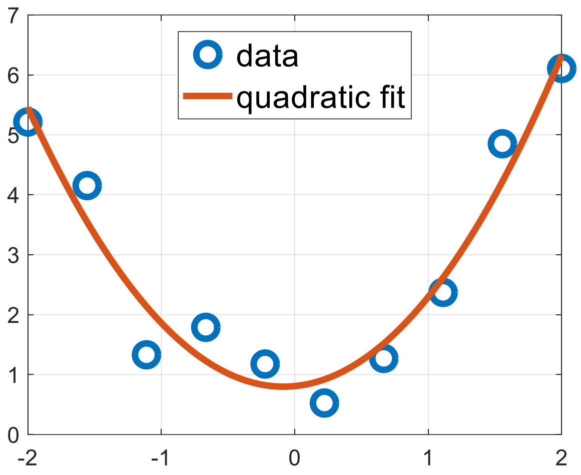
Shallow 1d Example



Training data (x,y) = evaluations of an unknown function G

Choose a "network architecture"

Train network = determine good parameters

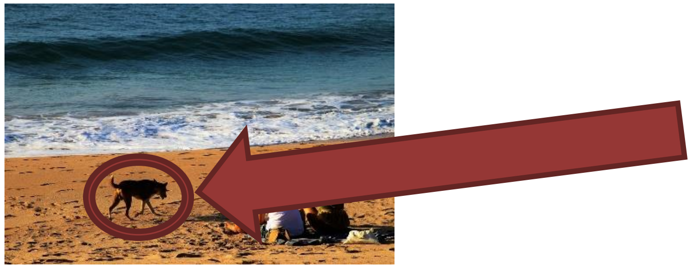


$$\mathcal{N}(x;\theta) = \theta_1 x^2 + \theta_2 x + \theta_3$$

weights / free parameters

This is a very simple 1d example! The power of machine learning, and the reason it receives a lot of attention are that similar concepts seem to work extremely well for incredibly complex functions G!!

Why are these problems difficult?



Yes, here it is!

Is there a dog in this image?

Unfortunately, this is how the dog looks like for the computer (red channel only)

178	189	190	187	182	174	120	139	172	175	141	124	149	168	183	176	184	182	184	192	192	195	185
170	174	181	162	144	178	174	183	178	180	183	181	180	186	183	181	185	189	190	190	170	172	208
186	187	185	185	188	187	186	185	181	182	170	176	188	189	189	185	187	194	192	182	182	180	193
182	188	185	178	164	150	159	170	180	184	173	153	128	114	119	153	182	137	145	195	188	191	191
186	137	75	45	33	20	25	32	50	63	41	22	2	0	0	18	46	29	38	154	186	191	175
119	26	2	2	2	0	0	0	0	0	0	0	0	3	9	2	0	0	3	17	52	91	120
19	9	7	5	5	3	6	8	8	4	3	2	2	3	6	7	4	6	4	0	0	0	1
3	0	4	6	7	4	6	9	8	1	4	5	6	4	5	6	6	8	11	10	10	6	0
118	60	1	4	6	7	6	6	6	5	8	6	9	10	10	7	3	8	27	29	23	11	5
204	98	2	7	8	13	13	12	12	14	11	4	4	6	8	9	6	8	13	30	38	18	16
113	104	26	9	11	11	13	13	13	15	12	7	6	5	7	7	9	21	44	68	50	23	13
38	102	56	14	16	13	26	31	20	22	11	7	9	8	8	10	42	101	179	165	113	29	4
52	23	40	29	14	4	78	160	82	46	25	6	7	7	9	40	121	166	126	92	127	85	26
37	41	51	25	16	9	77	204	180	134	107	60	14	16	36	114	182	161	131	120	168	160	134
76	174	144	31	12	9	87	187	194	198	198	186	59	36	39	122	197	189	204	207	200	203	212
156	201	202	98	4	7	124	197	191	191	194	206	79	92	65	67	185	194	202	212	200	145	125
209	201	206	136	6	4	143	196	182	193	195	186	69	169	156	39	137	219	202	201	190	137	137
140	113	93	93	22	44	174	179	188	199	206	186	79	172	206	97	70	159	122	106	169	190	194
107	134	155	148	31	116	203	199	194	192	210	173	77	135	160	175	60	80	141	130	167	190	194
132	157	195	134	20	156	197	172	151	176	181	94	96	168	163	177	114	49	172	161	132	139	137
137	172	212	105	25	164	188	171	153	88	74	53	117	164	143	130	147	101	93	134	155	143	111
183	186	177	106	31	157	196	188	95	63	139	144	152	166	158	175	175	188	123	69	67	118	132

Why are these problems difficult?

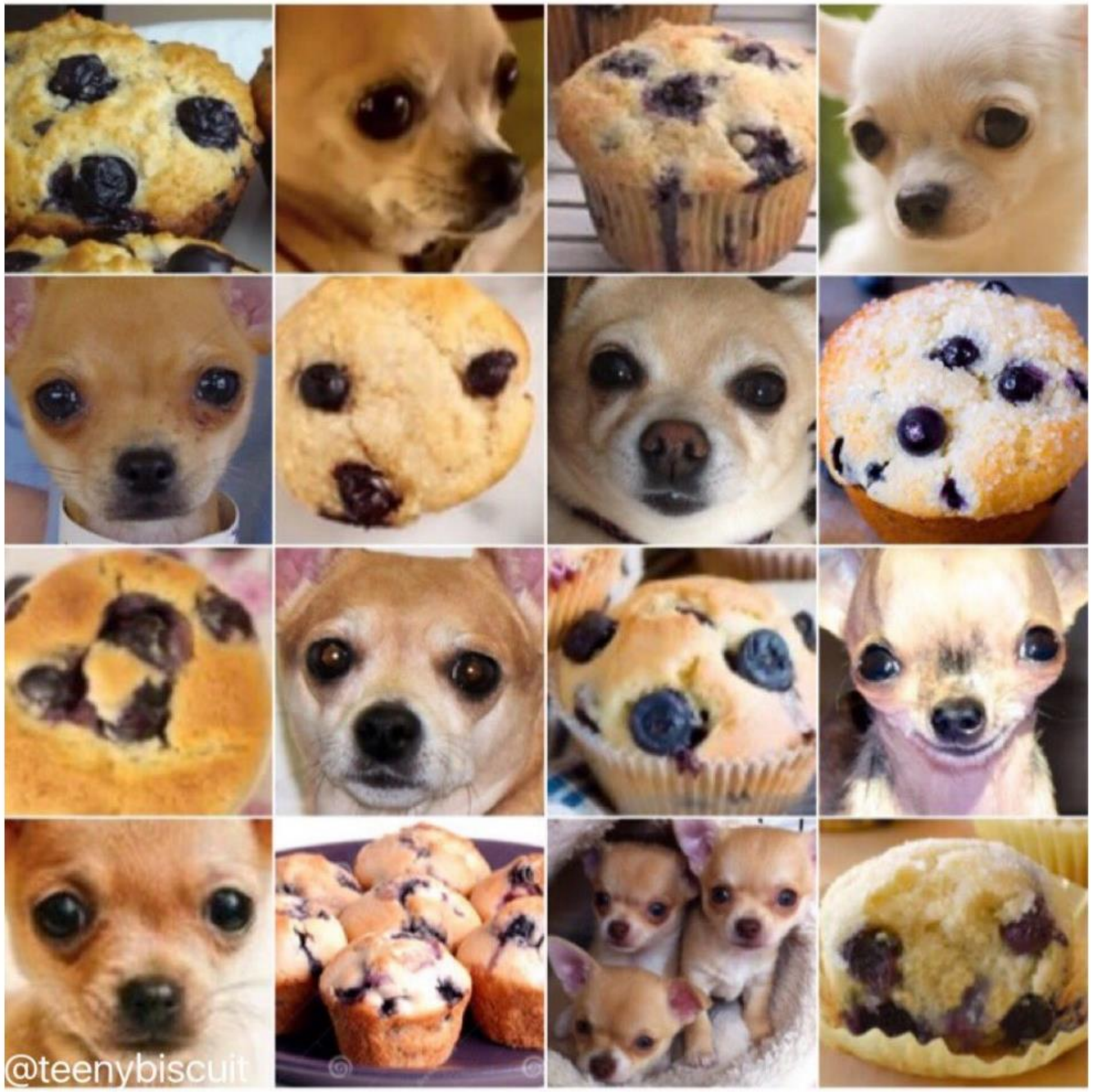


Image Source: Karen Zack, twitter.com/teenybiscuit

It is amazing how small visual differences can change our interpretation of images entirely!

Regression example

Simple example: We have some inputs x and want to predict some outputs y.

First step: Pick a network architecture

How do you want to parameterize the function that makes your prediction?

Easiest exemplary choice $\mathcal{N}(x;\theta) = \theta_w^T x + \theta_b$

Second step: Pick a loss function

How do you want to compare the ground truth to your prediction?

Examples:

$$\mathcal{L}(\mathcal{N}(x,\theta), y) = \|\mathcal{N}(x,\theta) - y\|^2$$

Squared loss

$$\mathcal{L}(\mathcal{N}(x,\theta), y) = -\sum_i y(i) \log(\mathcal{N}(x,\theta)(i))$$

Cross entropy loss

Linear regression

Together with the training examples (x_j, y_j) the loss function yields the training costs or energy, which measures the quality of the current parameters θ via

$$E(\theta) = \frac{1}{N} \sum_{\text{training examples } j=1}^{N} \mathcal{L}(\mathcal{N}(x_j, \theta), y_j)$$

Third step: Training

Choose a suitable algorithm to approximate the solution of

$$\hat{\theta} = \arg\min_{\theta} E(\theta)$$

Fourth step: Inference

Make predictions for new/unseen input data x via $\mathcal{N}(x; \hat{\theta})$

Common architectures

What are common choices for the architecture of networks?

Some approaches, e.g. support vector machines (SVMs) keep the architecture rather simple, e.g. linear.

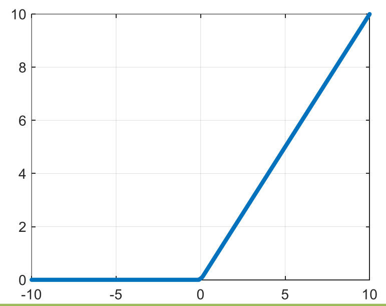
Recently very successful: **Deeply nested** functions (leading to deep learning)

(Shallow) example for mapping some $x \in \mathbb{R}^{11}$ to some $y \in \mathbb{R}$:

$$\mathcal{N}(x;\theta) = \underbrace{\theta_2^b}_{\in \mathbb{R}^{1\times m}} \; \underbrace{\sigma}_{\text{Activation function to break the linearity}}(\underbrace{\theta_1^w}_{\in \mathbb{R}^{m\times 11}} x + \underbrace{\theta_1^b}_{\in \mathbb{R}^{m}}) + \underbrace{\theta_2^b}_{\in \mathbb{R}}$$

Rectified linear unit (ReLU)

$$(\sigma(z))_j = \max(z_j, 0)$$



m is the number of *hidden neurons* (and an architectural design choice)

Common architectures

More generally: **Deeply** nested functions:

$$\mathcal{N}(x;\theta) = \ell^L(\ell^{L-1}(\dots(\ell^1(x;\theta^1)\dots);\theta^{L-1});\theta^L)$$

- Separate functions ℓ^i are often called *layers*.
- Each layer may or may not have *learnable parameters* θ^i.
- Typical architecture: Alternate between (affine) linear functions and simple nonlinear activations.

Common visualization:

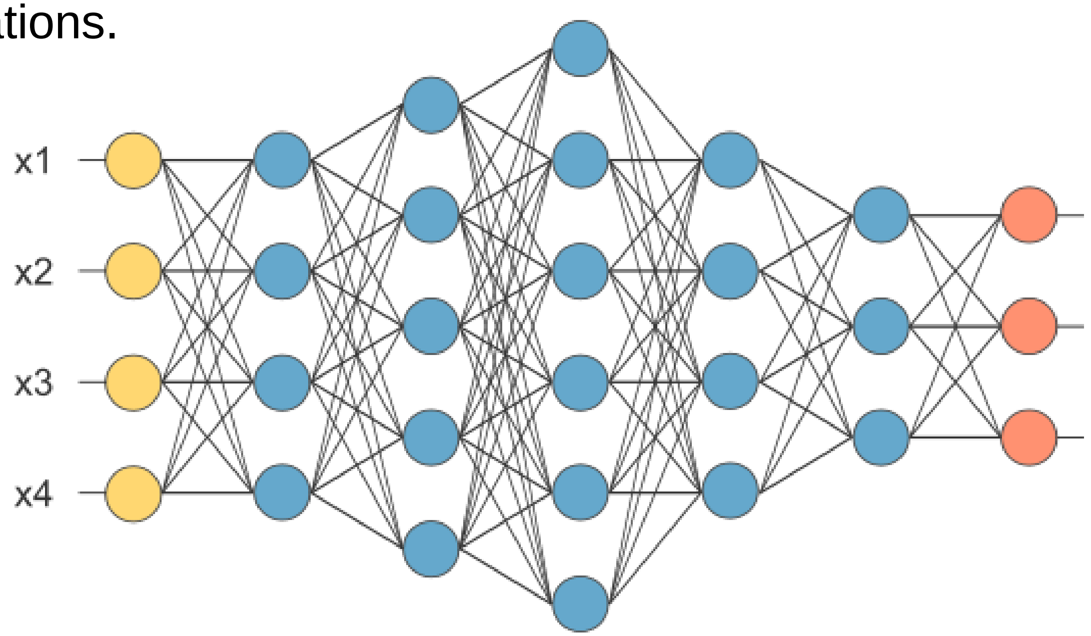


Image taken from https://www.neuraldesigner.com/

Fully connected networks

More technical, a *fully connected network* is of the form

$$\mathcal{N}(x;\theta) = \ell^L(\ell^{L-1}(\dots(\ell^1(x;\theta^1)\dots);\theta^{L-1});\theta^L)$$

If the index i is odd:

$$\ell^i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_{i+1}\times(n_i+1)} \to \mathbb{R}^{n_{i+1}}$$

$$\ell^i(z;\theta^i) = \theta^i \begin{pmatrix} z \\ 1 \end{pmatrix}$$

Output vector of size n^{i+1}

Parameter matrix of size $n^{i+1} \times (n^i + 1)$

Input vector concatenated with 1

This is called a *fully connected layer*!

If the index i is even:

Use a simple, componentwise nonlinear function!

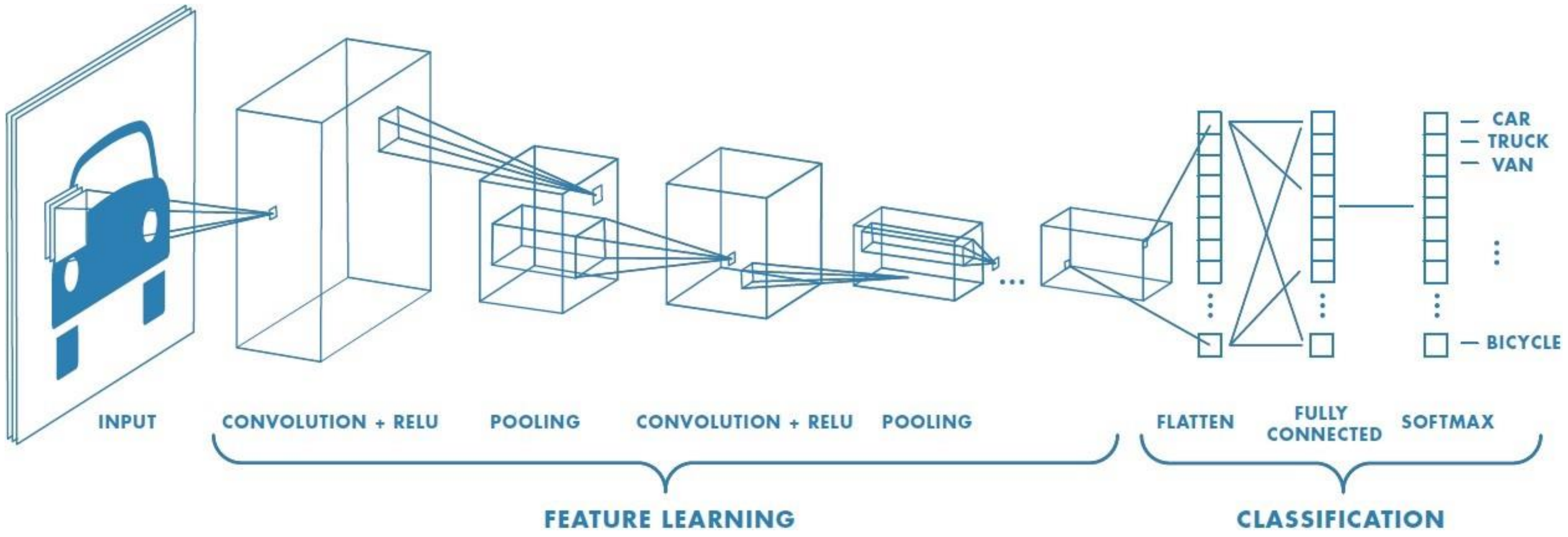
Example: *rectified linear unit*

$$\ell^i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$$

$$(\ell^i(z))_j = \max(z_j, 0)$$

This is called an activation function!

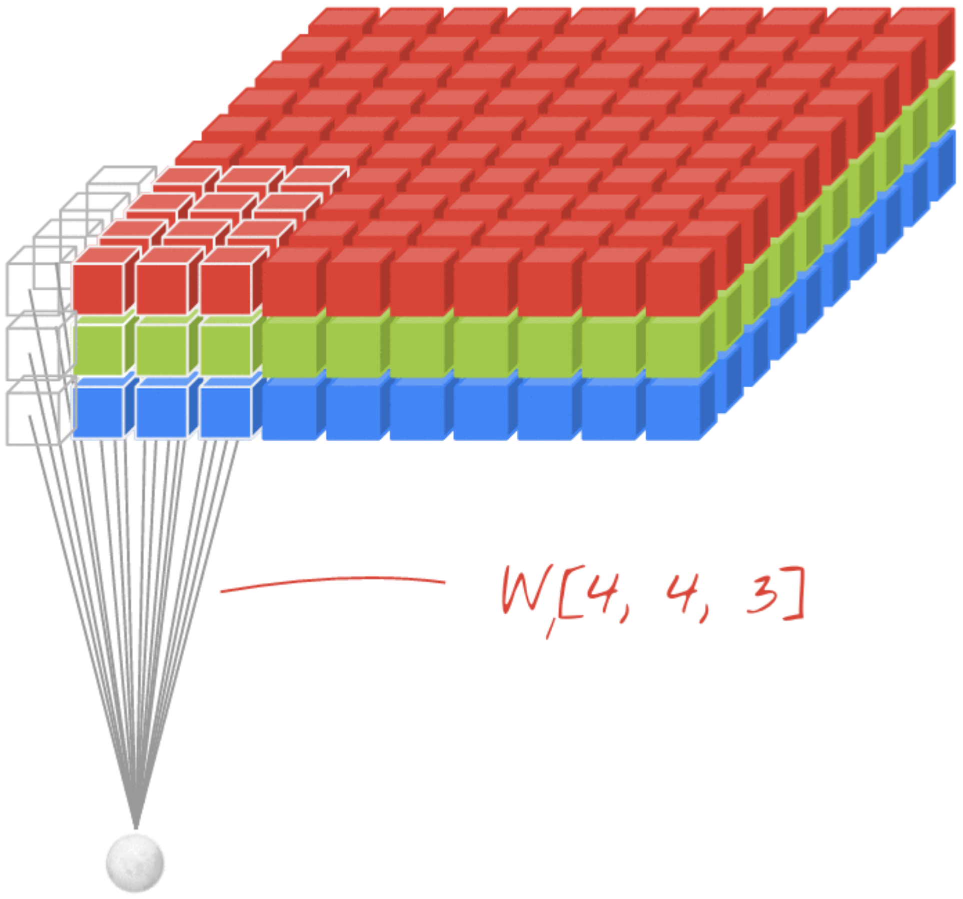
And for images?



https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148

Arbitrary affine linear functions are now restricted to convolutional filters!

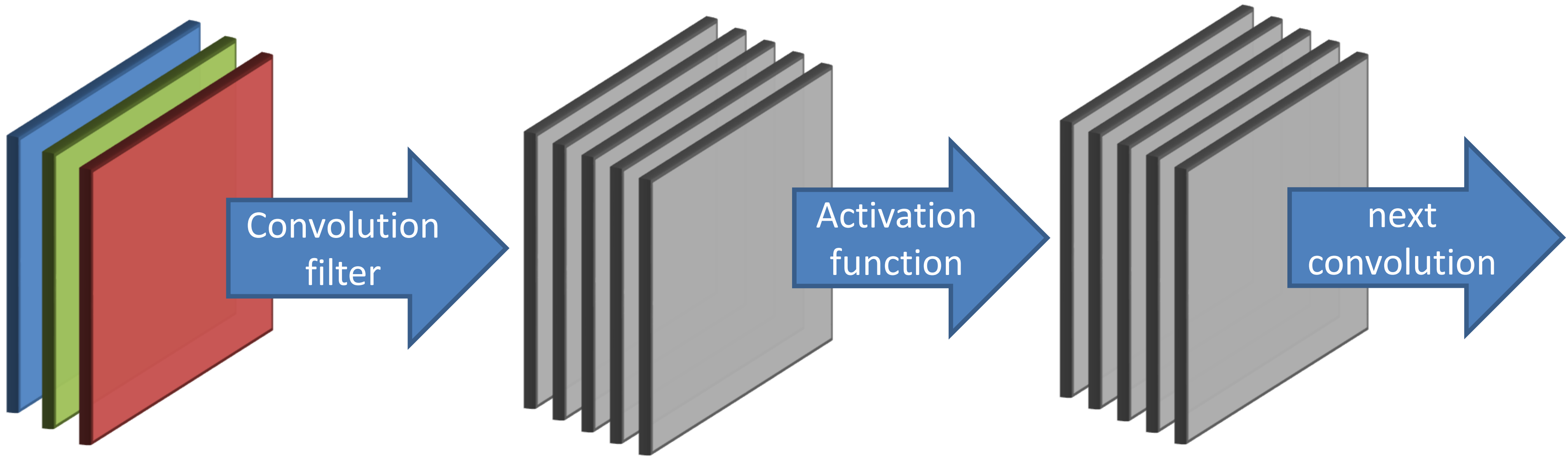
Convolution / Correlation



Animation taken from https://sites.google.com/site/nttrungmtwiki/home/it/data-science---python/tensorflow/tensorflow-and-deep-learning-part-3

Convolutional Neural Networks

Network architecture design for images:



RGB-image

Result of several filters (5 in this case)

Result, e.g. after ReLU

The number of filters determines the number of channels in the next layer!

Convolutions with stride

Input image

f

1	2	-1	4	4	2
4	-1	2	3	6	2
2	1	4	1	3	3
1	5	2	4	8	7
3	5	2	1	9	8
6	5	7	6	6	6

Convolution result without padding and stride 3

-13	12
5	13

kernel

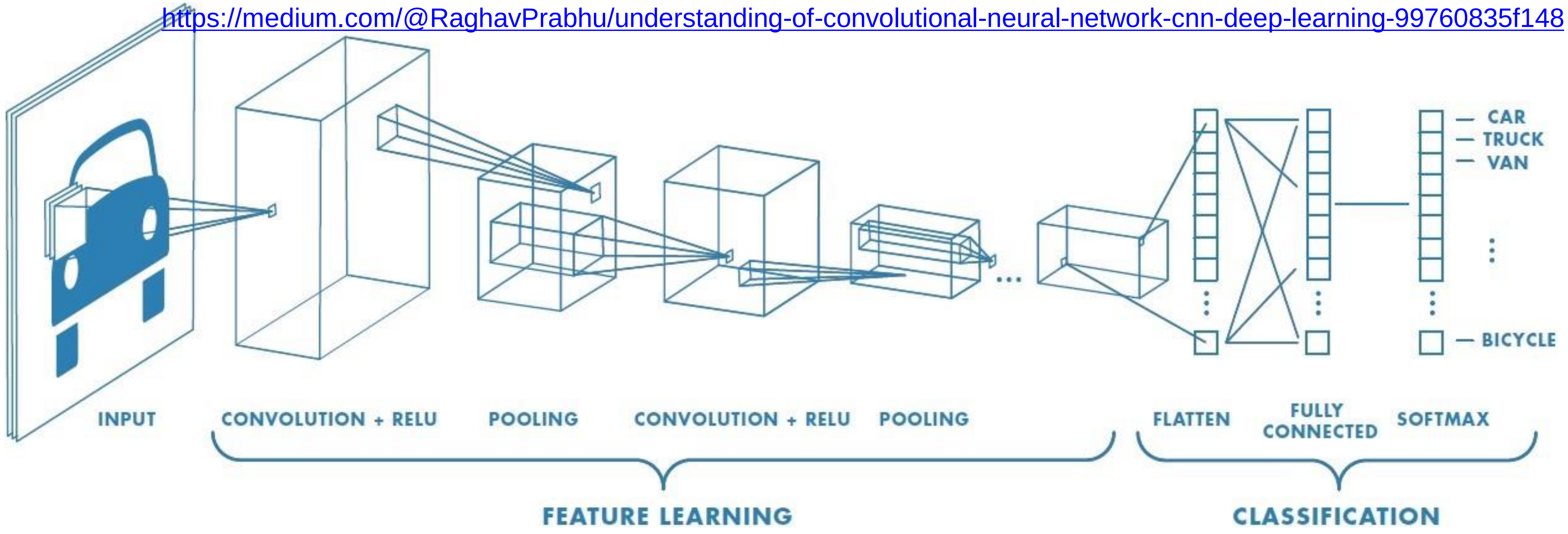
g

0	-1	0
-1	4	-1
0	-1	0

Of course the computation blocks may also overlap (e.g. stride 2)

And for images?

https://medium.com/@RaghavPrabhu/understanding-of-convolutional-neural-network-cnn-deep-learning-99760835f148



Pooling layers use a sliding window over the image (similar to a convolution), but often in a non-overlapping fashion. Each window (of which one can specify the size), gets reduced to a single number, by

- Taking the maximum value among the entries within the window (*max.-pooling*)
- Taking the average value among the entries within the window (*avg.-pooling*)
- Less frequent: Taking the ℓ^p norm of each window (*fractional max-pooling*)

How does training work?

If all partial derivatives of a function $E : \mathbb{R}^n \to \mathbb{R}$ exist and are continuous, we call E *continuously differentiable*, and call

$$\nabla E := \begin{pmatrix} \frac{\partial E}{\partial \theta_1} \\ \vdots \\ \frac{\partial E}{\partial \theta_n} \end{pmatrix} : \mathbb{R}^n \to \mathbb{R}^n$$

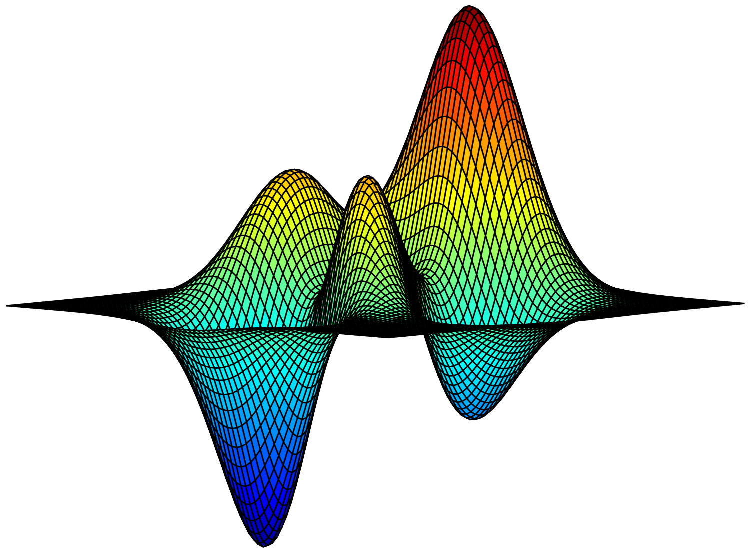
the *gradient* of E. Evaluating the gradient at a point θ yields a vector in $\mathbb{R}^n$.

Necessary condition for local minima: If $E : \mathbb{R}^n \to \mathbb{R}$ has a local minimum at some point θ, then it holds that

$$\nabla E(\theta) = 0$$

Training algorithms

Unfortunately, sufficient conditions for global optimality are impossible to verify as soon as the network architecture is difficult (more precisely, as soon as the optimization problem does not have very special properties such as convexity)



Common deep learning approach: Resign the desire to compute global minimizers! Iteratively reduce the training costs – at most until you reach $\nabla E(\hat{\theta}) = 0$. One never even checks sufficient conditions for *local* minima.

Most commonly used algorithms are variants of

gradient descent!

UNIVERSITÄT SIEGEN

Gradient descent

ZESS

Basic idea: For a continuously differentiable $E : \mathbb{R}^n \to \mathbb{R}$, the quantity $-\nabla E(\theta)$ points into the direction of steepest descent.

Move into this direction!

$$\underbrace{\theta(k+1)}_{\text{New parameters}} = \underbrace{\theta(k)}_{\text{Previous parameters}} \underbrace{- \tau \nabla E(\theta(k))}_{\text{Direction of steepest descent}}$$

The parameter τ is called *step-size* or *learning rate*.

Gradient computation

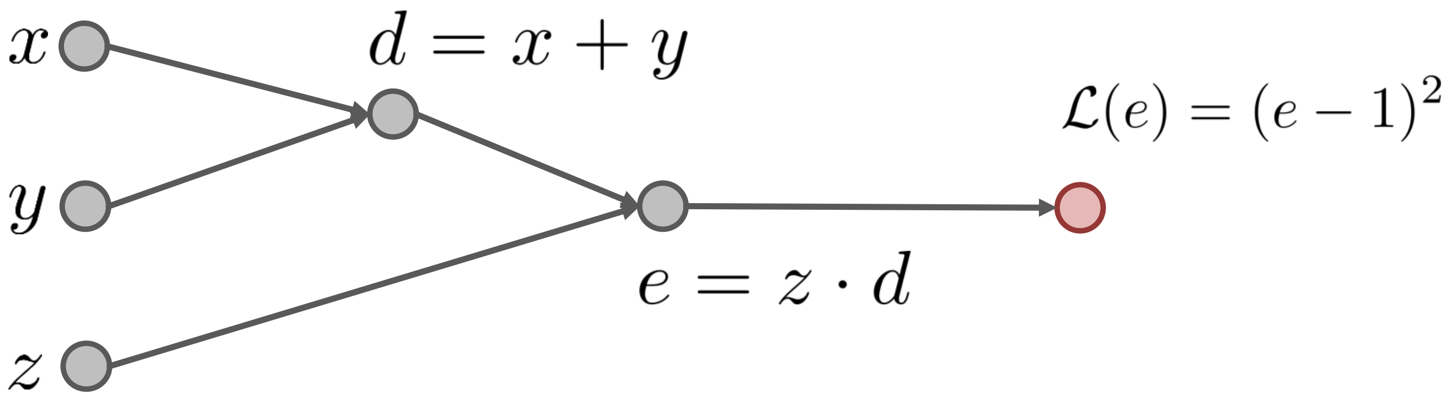
How do we compute the gradient of a deeply nested function?

Remember: **Chain rule**

$$E = f \circ g \quad \Rightarrow \quad \nabla E(x) = \nabla g(x) \cdot \nabla f(g(x))$$

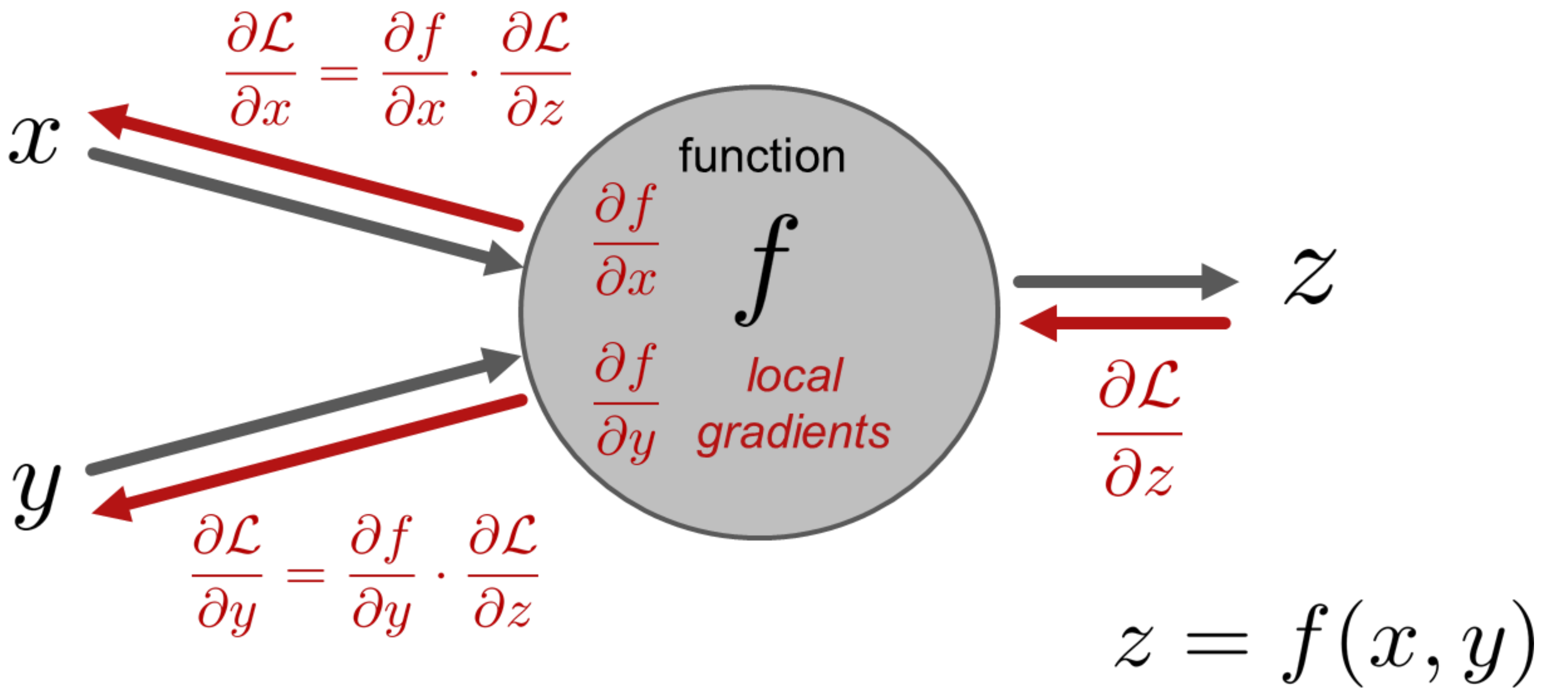
Essetially, gradient computations (known under the name of ***backpropagation***) are just a repeated application of the chain rule!

Scalar example:



Backpropagation – “local gradients”

Key ingredient: Every function only needs to know the derivatives with respect to its own inputs!



$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial \mathcal{L}}{\partial z}$
x
function
$\frac{\partial f}{\partial x}$
f
$\frac{\partial f}{\partial y}$
local gradients
z
$\frac{\partial \mathcal{L}}{\partial z}$
y
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial f}{\partial y} \cdot \frac{\partial \mathcal{L}}{\partial z}$

$$z = f(x, y)$$

Stochastic gradient descent (SGD)

Problem:

$$E(\theta) = \sum_{\text{training examples } j} \mathcal{L}(\mathcal{N}(x_j;\theta), y_j)$$

Can easily be a sum over 1,000,000 terms

Idea: Use only a few random summands to compute an approximate gradient:

$$E_k(\theta) = \sum_{j \in I(k)} \mathcal{L}(\mathcal{N}(x_j,\theta), y_j)$$ for a ***very small index set*** $I(k)$

Update the parameters using this approximation

$$\theta(k+1) = \theta(k) - \tau \nabla E_k(\theta(k)) \approx \theta^k - \tau \nabla E(\theta^k)$$

Randomly selecting entries in the index set $I(k)$ leads to the name ***stochastic gradient descent***. The training examples (x_j, y_j) with $j \in I(k)$ are called a ***mini-batch***.

GD with Nesterov Momentum

$$\theta(k+1) = \theta(k) - \tau(k)\nabla E(\theta(k)) + \alpha * v(k+1)$$

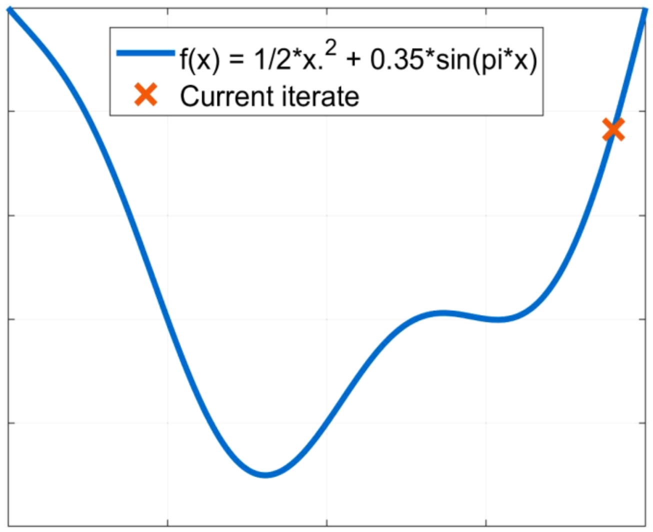
Gradient descent | + additional velocity

$$v(k+1) = \alpha \cdot v(k) - \tau(k)\nabla E(\theta(k))$$

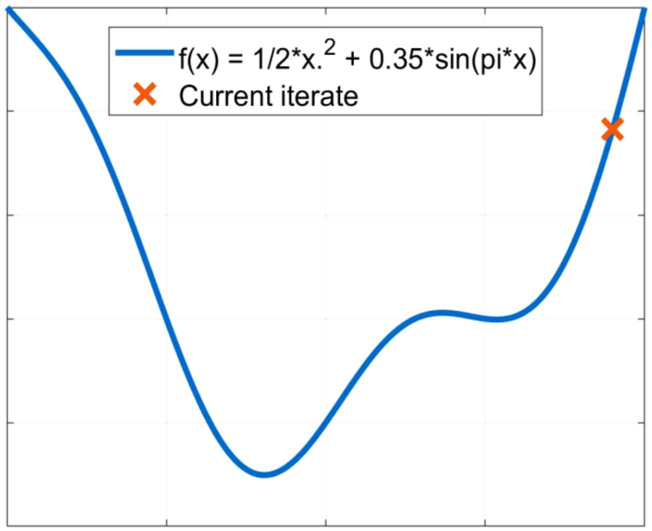
new velocity | old velocity | accumulating gradients, i.e. ``speeds + directions''

damping with $\alpha < 1$, could be interpreted as *friction*

<u>Gradient descent</u>



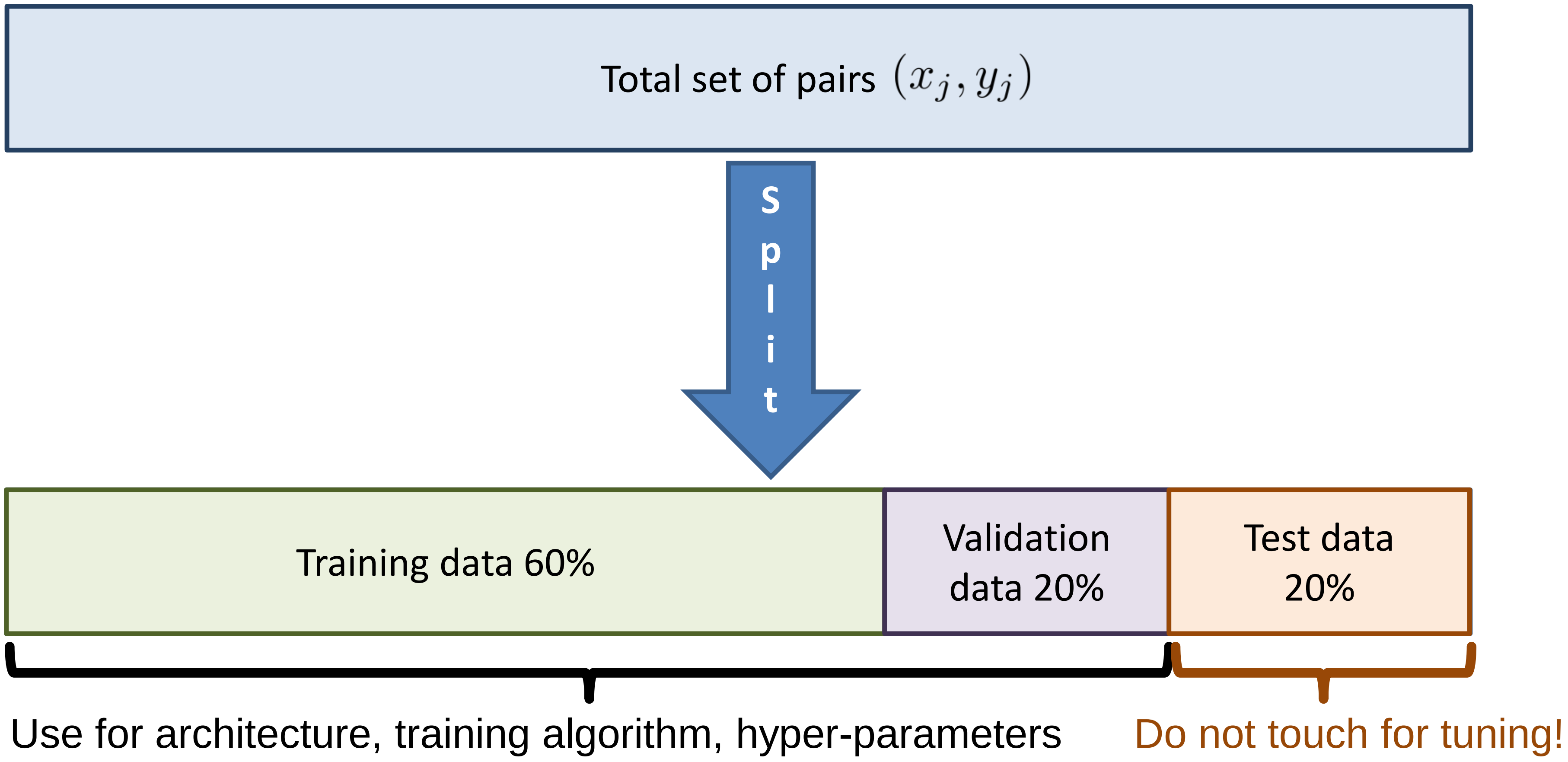
<u>Gradient descent with *Momentum*</u>



An even more advanced technique with similar ideas is the Adam optimizer

Good practice: Validate!

Without validation it is impossible to judge whether your model is reasonable or starts overfitting the data! Therefore:



Total set of pairs (x_j, y_j)
Split
Training data 60%
Validation data 20%
Test data 20%
Use for architecture, training algorithm, hyper-parameters
Do not touch for tuning!

Good practice: Validate!

How do you use the validation set?

Deep Learning Recipe

Do you have a small training error?

No

Does the training error decay reasonably at the beginning of the training?

Yes

Increase the network's expressiveness.

No

Tune your training.

Yes

Do you have a small validation error?

No

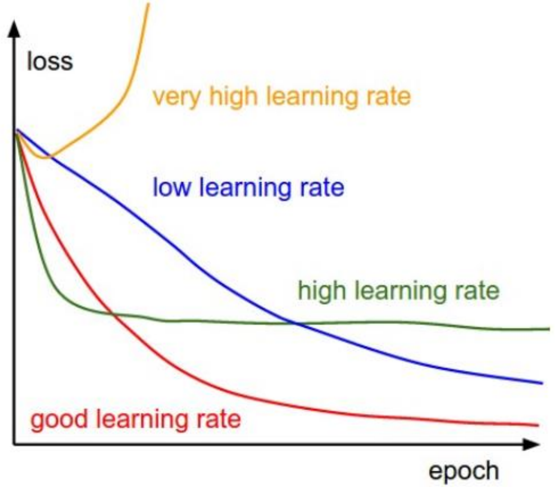
Use more data (if possible).

Regularization techniques.

Use prior knowledge (modelling parts of the network)

Yes

Perfect! You're done!



If nothing works, try a different network architecture.

What else do you need to know?

I expect all lectures to be pretty self-contained, but they do of course require you to be familiar with ML concepts.

There might be terms like *transpose convolutions* in the presentation or in a paper relevant to your project without you knowing what it is.

Important skill: Being able to look these things up and deciding the level of detail you need to know!

Up next: A small hands-on image classification example in Matlab