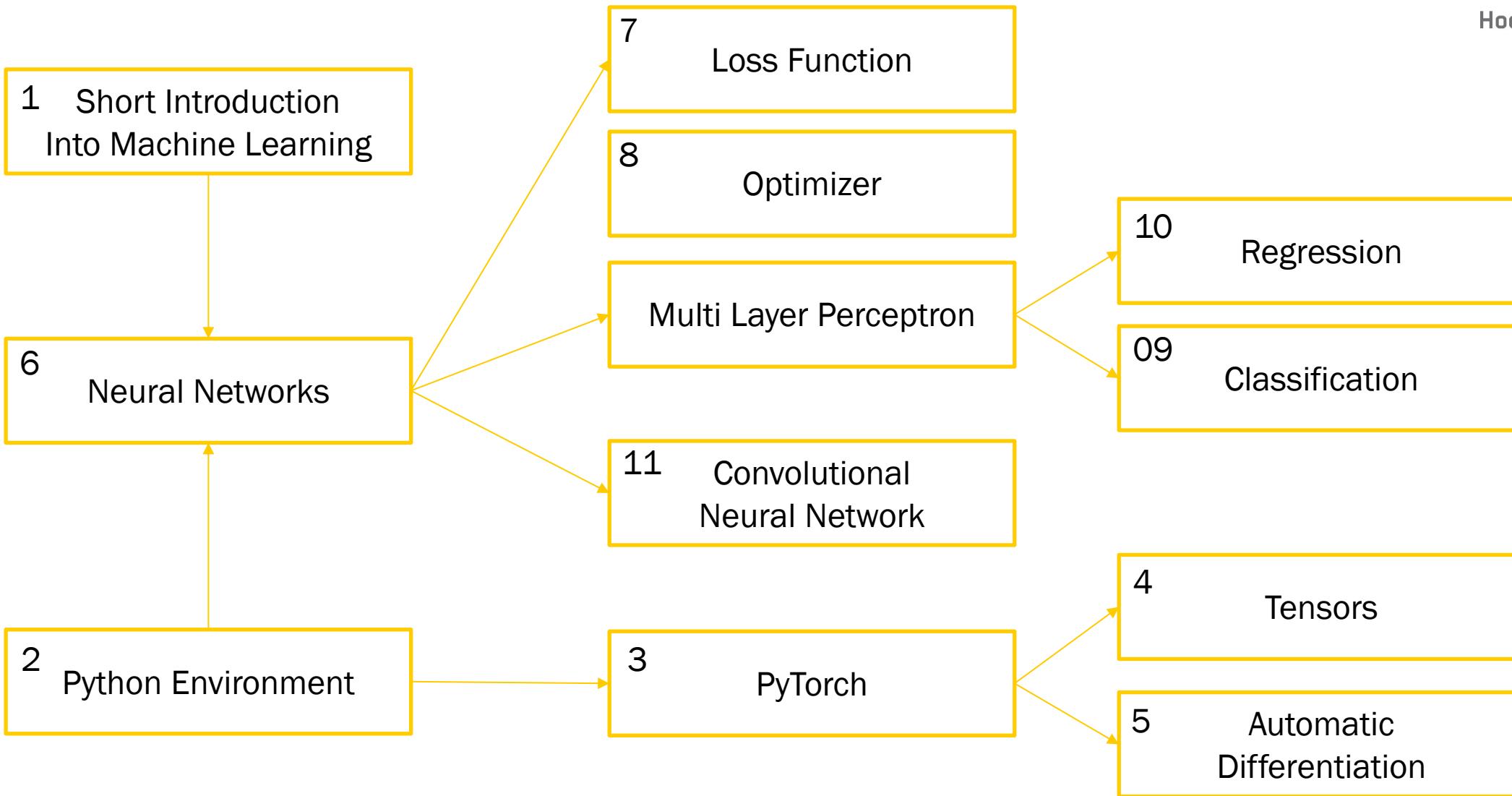


Intelligente Informationssysteme

1 – Neuronale Netze

Dominik Neumann

Bock 1 – Neural Network



01.01

Kurze Einführung in Maschinelles Lernen



A historical perspective

Arthur Samuel between 1952 and 1959 investigated the question of whether a computer can be enabled to do something without explicit instructions.

Is a computer able to learn?

That question leads to the first formal definition of machine learning.

Definition:

Machine Learning is the field of study that gives computers the ability to learn without explicitly programmed.



A modern perspective

Several decades later, Tom Mitchel specified machine learning in his 1997 definition:

Definition:

Machine Learning is the science that is concerned with the question of how to construct computer programs that automatically improve with experience.

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .



A Data-driven perspective

Building on this, Yaser Abu-Mostafa describes the essence of machine learning very impressively in his Machine Learning lecture from 2012.

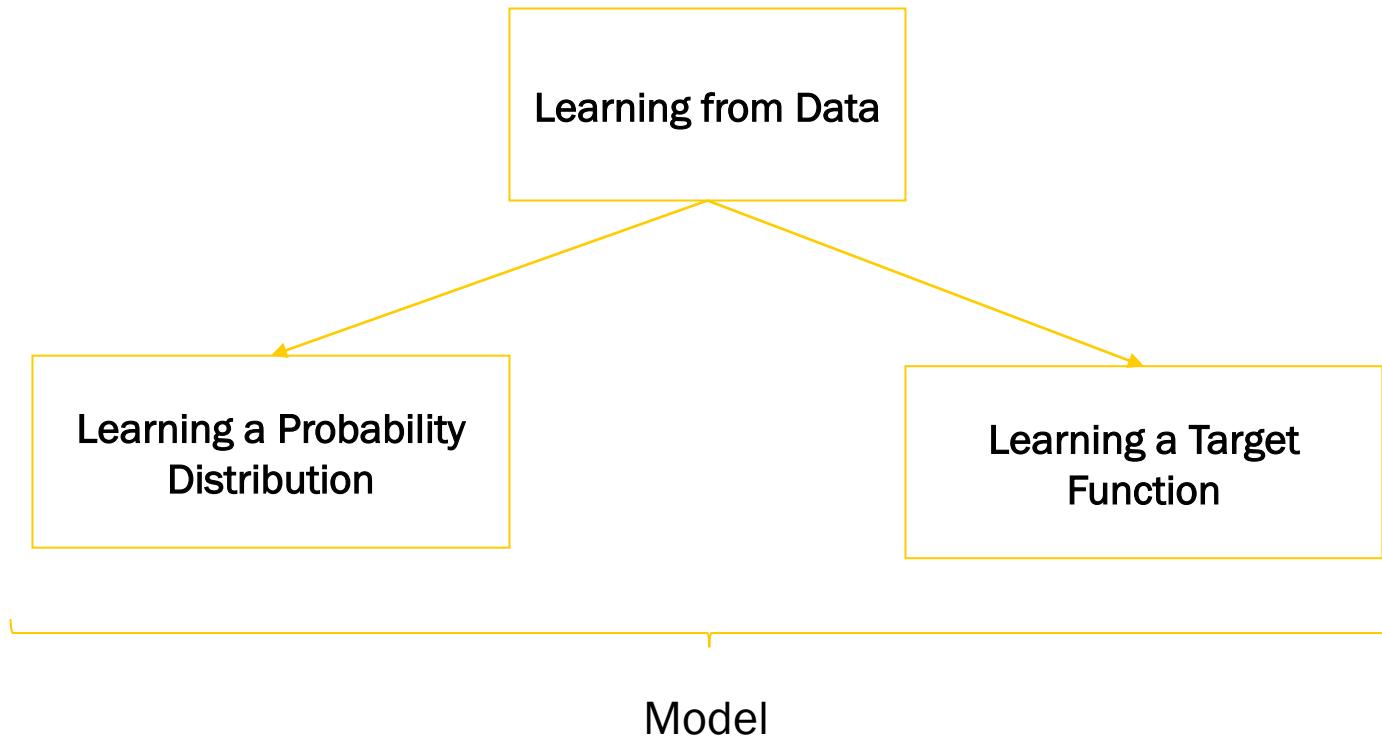
Machine Learning is feasible when:

1. A pattern exists
2. And we are not able to pin it down. We do not know the maths or rules behind that pattern.
3. But we have a lot of meaningful data or observations that can be used to learn the hidden pattern.



A Data-driven perspective

Learning from data can be seen from a **probabilistic** or a numerical **analytic** perspective.



The Learning Problem

To use Arthur Samuel's words, our goal in this lecture (today) is to understand how to enable a computer to learn.

To do this, we obviously need 3 elementary building blocks:

1. We always start with **data**
2. Then we need a parameterized **model** that can explain the data.
(given an appropriate choice of parameters.)
3. At least an evaluation procedure, the **loss** function, that allows us to adjust the parameters of our model.



Learning from Data can be seen as learning a function **f** or a probability distribution **P**.

From a probabilistic point of view it is our goal to learn a **joint probability distribution P** that explains our data. With the knowledge of **P** it would be possible to predict based on observations made:

$$P(y|x) = \frac{P(x,y)}{P(x)}, \quad P(x) \neq 0, x \in X, y \in Y.$$

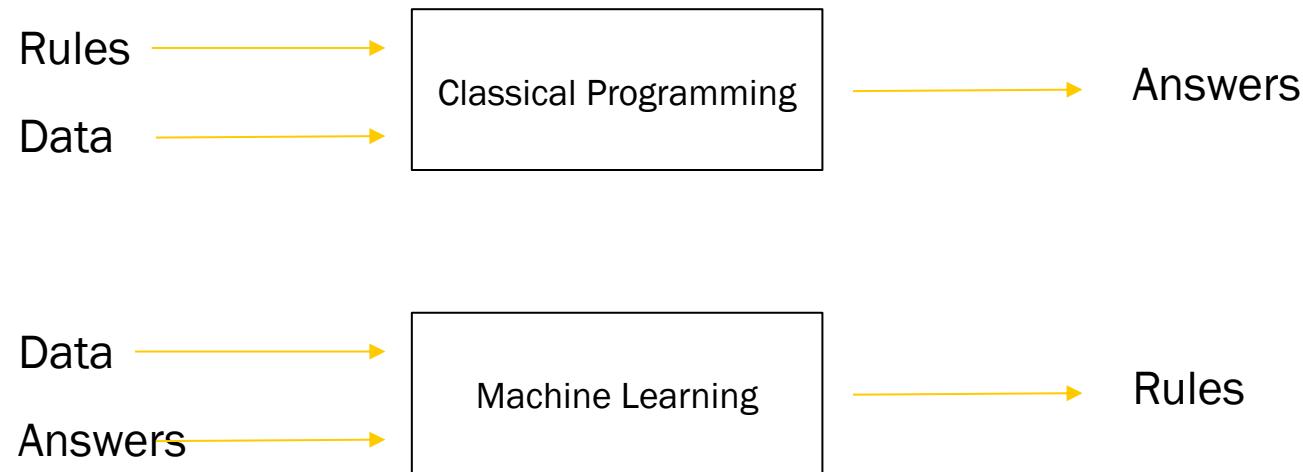
On the other side the problem could be seen as a numeric optimization problem.

Then we are searching for a function $f: X \rightarrow Y$ with $f(x) = y$ that explains the data.

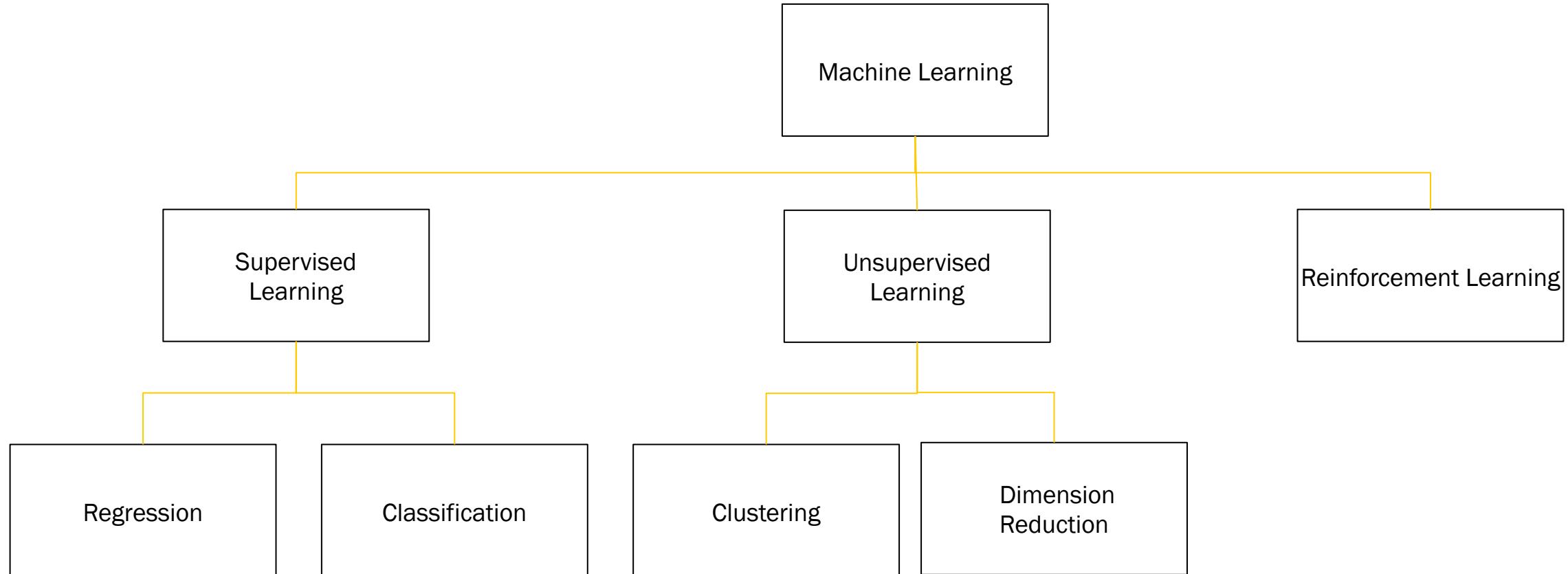
Both the probability distribution P and the function f are unknown. But there is a chance to learn P or f based on data.



A new programming paradigm

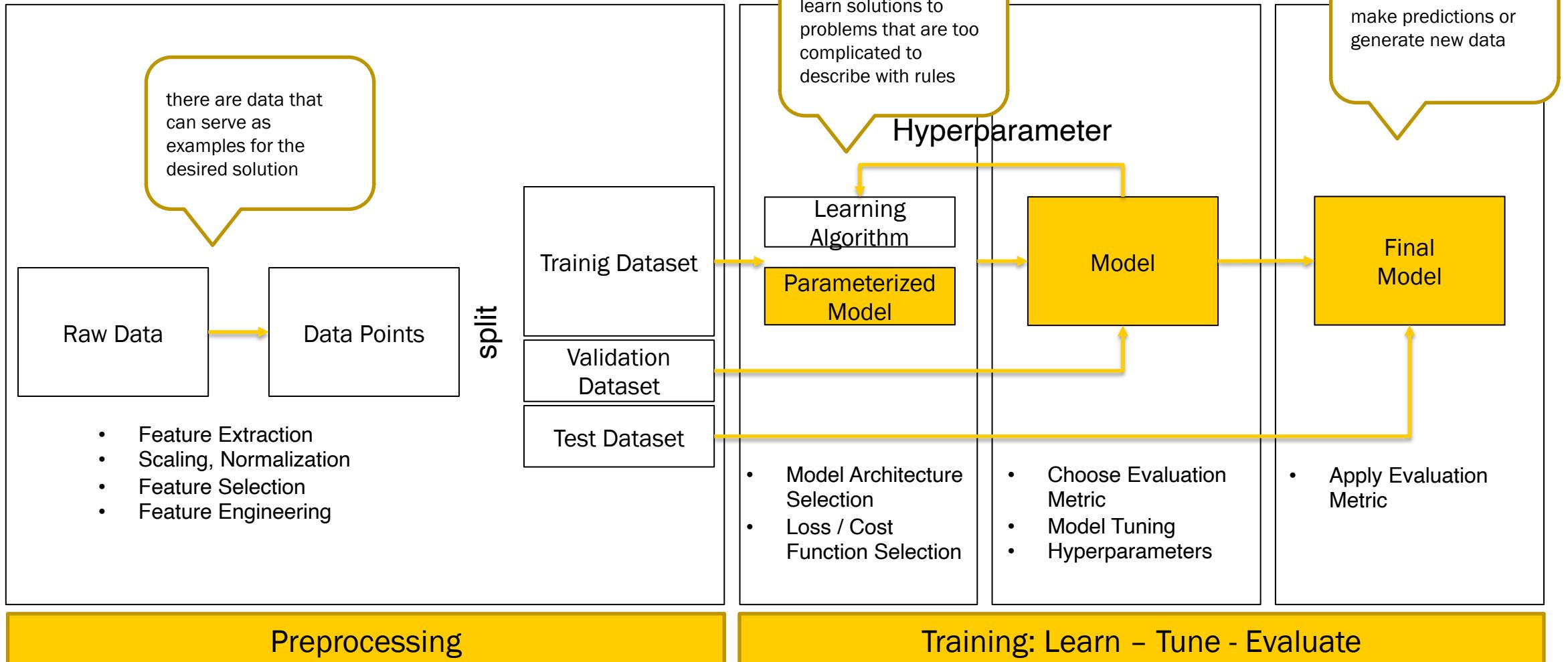


Three Types of Machine Learning



Concept of a Model in Machine Learning

- learned in some machine learning process



01.02

Python Environment



Create for each project its own Python environment to omit dependency conflicts:

\$ python -m venv envpath creates a virtual environment (in `envpath` directory)

Beispiel:

```
(base) done@Dominiks-MBP ~ % python -m venv Documents/tmpenv/test1
(base) done@Dominiks-MBP ~ % cd Documents/tmpenv/test1
(base) done@Dominiks-MBP test1 % ls -l
total 8
drwxr-xr-x 12 done  staff  384  2 Okt 10:46 bin
drwxr-xr-x   2 done  staff    64  2 Okt 10:46 include
drwxr-xr-x   3 done  staff    96  2 Okt 10:46 lib
-rw-r--r--   1 done  staff    89  2 Okt 10:46 pyvenv.cfg
```



Python Environment

```
(base) done@Dominiks-MBP ~ % tree -dL 4 Documents/tmpenv/test1  
Documents/tmpenv/test1
```

```
└── bin    ← Link to system python  
└── include  
└── lib  
    └── python3.10  
        └── site-packages  
            ├── _distutils_hack  
            ├── pip  
            ├── pip-22.3.1.dist-info  
            ├── pkg_resources  
            ├── setuptools  
            └── setuptools-65.5.0.dist-info
```



Activate and deactivate the virtual environment:

On Unix/MacOS:

\$ source envpath/bin/activate	activates the environment
\$ source envpath/bin/deactivate	deactivates the environment

On Windows:

C:\> envpath/Scripts/activate.bat

Activation does many things:

1. Add virtualenv's *bin* directory at the beginning of shell PATH
2. Defines a deactivate command to return the environment to its former state
3. Modifies shell prompt: puts the environment name in front of it
4. Defines a VIRTUAL_ENV environment variable as the path to virtual environments root directory.

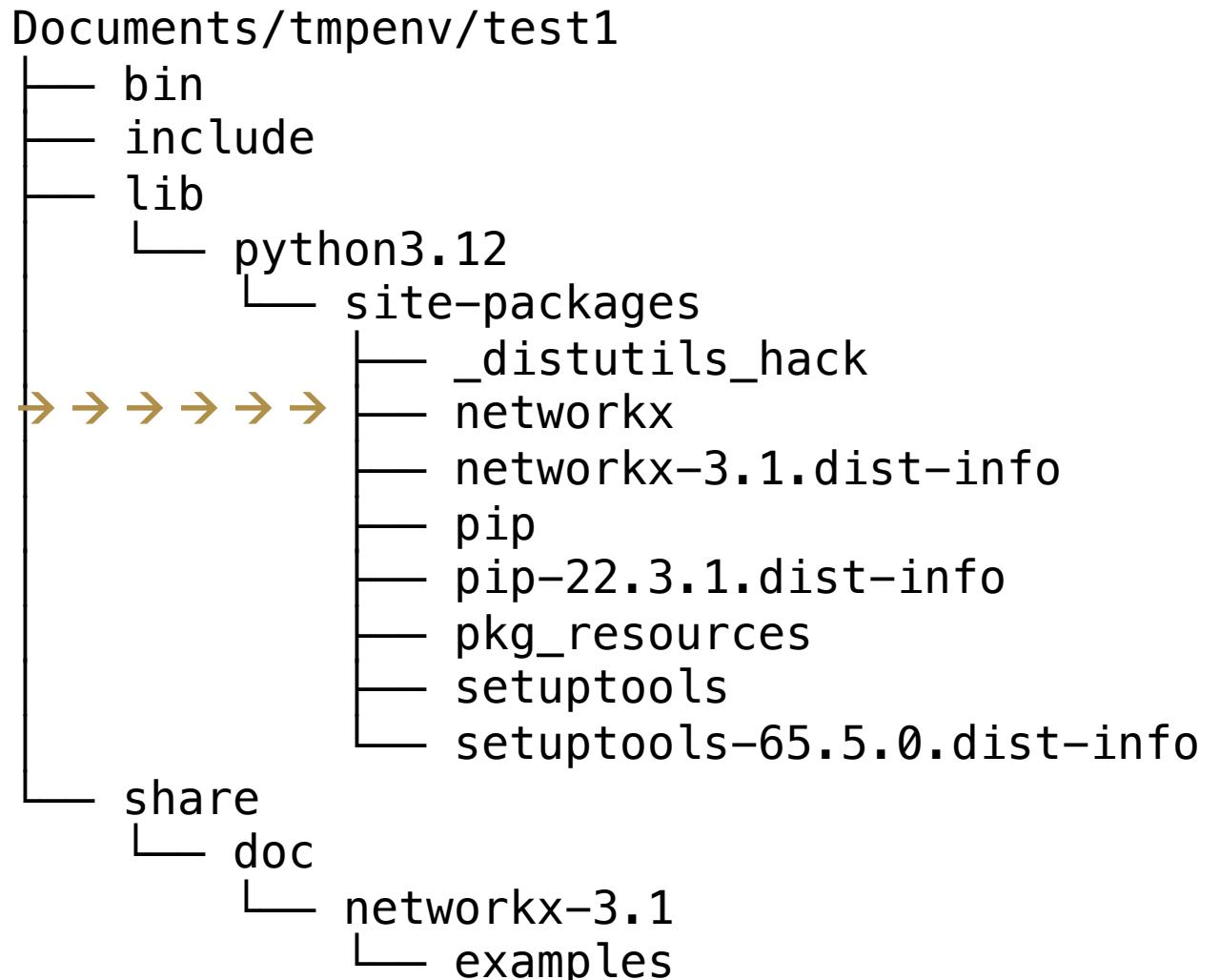


Install a package from <https://pypi.org> into the virtual environment:

`pip install packagename`

Example:

`pip install networkx`



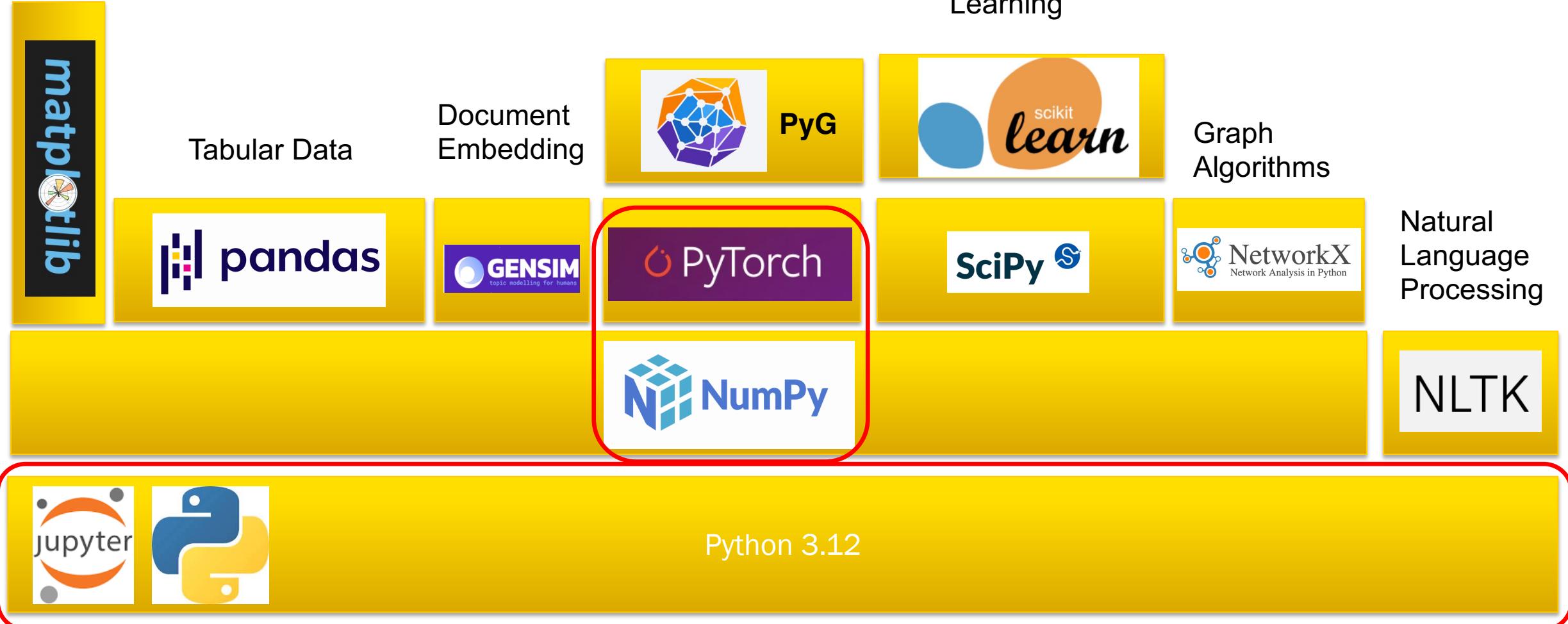
A smart way to manage environments is conda:

<https://docs.conda.io/en/latest/>

```
conda create --name envname python=3.12
conda activate envname
conda deactivate
```



MACHINE LEARNING PYTHON STACK



MACHINE LEARNING PYTHON STACK

- Jupyter: <https://jupyter.org> □ pip install jupyter
- Gensim: <https://radimrehurek.com/gensim/> □ pip install gensim
- NetworkX: <https://networkx.org> □ pip install networkx
- NLTK: <https://www.nltk.org> □ pip install nltk
- NumPy: <https://numpy.org> □ pip install numpy
- Pandas: <https://pandas.pydata.org> □ pip install pandas
- Pytorch: <https://pytorch.org> □ pip install torch torchvision
- SciPy: <https://scipy.org> □ pip install scipy
- Scikit-Learn: <https://scikit-learn.org> □ pip install scikit-learn

Übung:

- Installiere Python 3.12 auf dem Notebook
- Erstelle mit Conda ein Environment, z.B. ws25

```
conda create --name ws25 python=3.12
```

```
conda activate ws25
```

```
pip install jupyter
```

```
pip install numpy
```

```
pip install torch
```

- Starte Jupyter Notebook mit

```
jupyter notebook
```



01.03

PyTorch



PyTorch is hosted by the PyTorch Foundation: <https://pytorch.org>

„PyTorch is an open source deep learning framework built to be flexible and modular for research, with the stability and support needed for production deployment.“

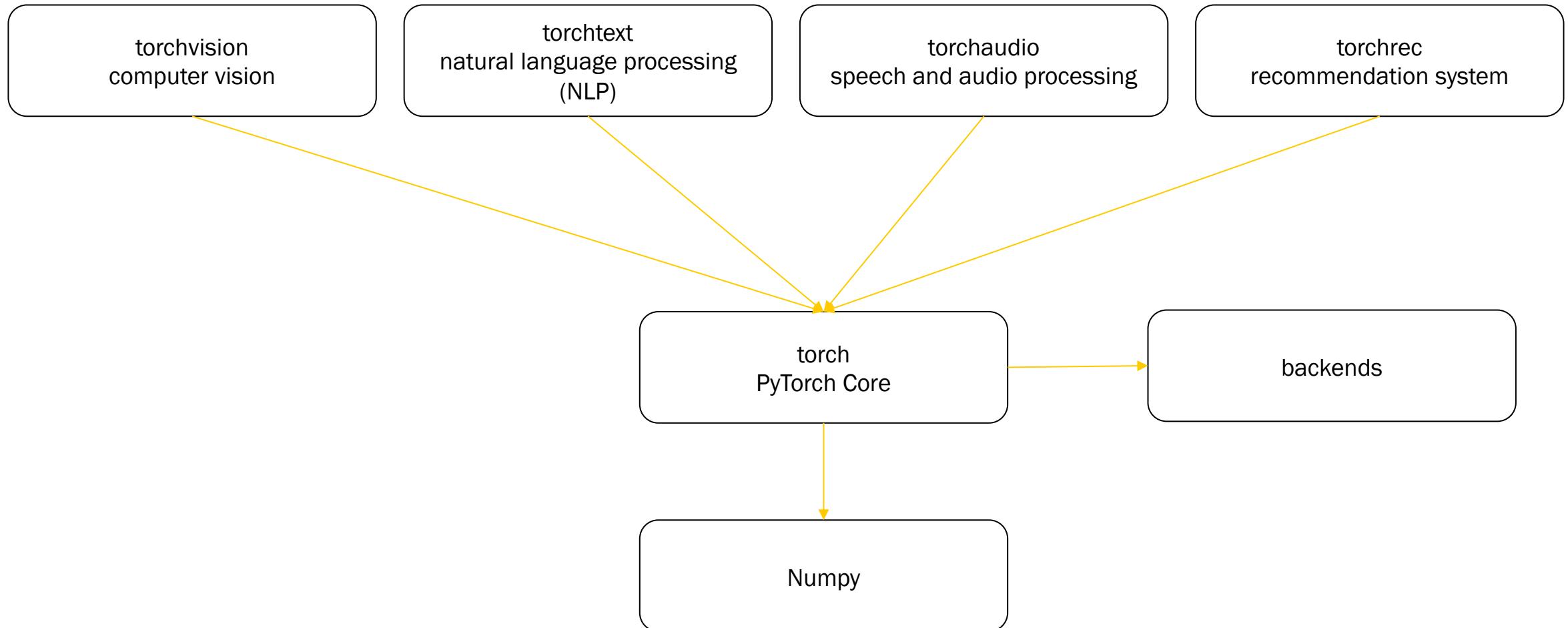
History:

- Before PyTorch, there was Torch — an open-source machine learning framework written in Lua and C. Torch was developed around 2002 by researchers at Idiap Research Institute, NYU, and others.
- PyTorch 0.1 was released in September 2016 (publicly in early 2017) by Facebook’s AI Research lab (FAIR).
- PyTorch 1.0 launched at NeurIPS 2018, marking the transition from a “research tool” to a production-ready framework. Facebook, Microsoft, and others began using PyTorch in production systems.
- In September 2022, Meta announced that PyTorch would join the Linux Foundation under the PyTorch Foundation.
- PyTorch 2.0 (released in March 2023).
- As today PyTorch is the industry standard in AI research and deployment.



PyTorch

Modules, Packes and Dependencies



In PyTorch, a **backend** refers to the mechanism that determines on which hardware and with which low-level API tensor operations are executed. Each backend is therefore a combination of:

- memory management (Speicherverwaltung),
- kernel implementations (Kernel-Implementierungen),
- and device drivers (Gerätetreiber).

Hardware	Verfügbare Backends
CPU (x86, ARM)	CPU, XLA
NVIDIA GPU	CUDA, XLA
AMD GPU	ROCm (HIP)
Apple Silicon GPU	MPS, MLX (experimentell)
Intel GPU	XPU (oneAPI / Level-Zero)
TPU (Google)	XLA
Windows-GPU allgemein	DirectML
Mobile / Embedded	Vulkan, IREE

PyTorch

Core Packages

Core Packages and Classes

<code>torch.Tensor</code>	the core multi-dimensional array object (similar to NumPy arrays but with GPU acceleration)
<code>torch.nn.Module</code>	the fundamental building block for all neural networks.
<code>torch.cuda</code>	GPU support (CUDA utilities, device management).
<code>torch.autograd</code>	automatic differentiation engine for gradient computation
<code>torch.nn</code>	neural network layers, loss functions, and utilities
<code>torch.optim</code>	optimization algorithms (SGD, Adam, RMSProp, etc.)
<code>torch.utils</code>	helper tools (data loading, checkpointing, etc.).
<code>torch.jit</code>	Just-In-Time compilation (TorchScript).
<code>torch.distributed</code>	distributed training utilities.
<code>torch.multiprocessing</code>	parallel data loading and multiprocessing.
<code>torch.onnx</code>	exporting PyTorch models to the ONNX format.
<code>torch.fx</code>	symbolic tracing and program transformation
<code>torch.compile</code>	dynamic compiler to optimize models with TorchDynamo and TorchInductor



01.04

Tensoren in PyTorch



Tensoren sind die grundlegenden Datenstrukturen in allen Machine Learning Frameworks (Numpy, TensorFlow, JAX, Keras, PyTorch). Ein Tensor ist ein Container für numerische Daten:

```
# 0D Tensor (a single number)
torch.tensor(3)

# 1D Tensor (a list of numbers)
torch.tensor([1, 2, 3])

# 2D Tensor (a matrix)
torch.tensor([[1, 2, 3],
             [4, 5, 6]])

# 3D Tensor (a cube of numbers)
torch.tensor([[[1, 2], [3, 4]],
             [[5, 6], [7, 8]]])
```

A **torch.Tensor** is a multi-dimensional matrix containing elements of a single data type:

- <https://docs.pytorch.org/docs/stable/tensors.html>
- Shape tensor.shape, tensor.size()
- Axis tensor.ndim
- Type: tensor.dtype



Tensoren können auf vielfältige Weise erzeugt werden:

- Direkt aus einer Python Liste: der zugrunde liegende Datentyp wird dabei antizipiert.

```
data = [[1, 2], [3, 4]]  
t = torch.tensor(data)
```

- Aus einem Numpy Array

```
data = numpy.array([[1, 2], [3, 4]])  
t = torch.tensor(data)
```

- Oder mit einem factory methode:

```
ones      = torch.tensor.ones(size=(2,2))  
zeros    = torch.tensor.zeros(size=(2,2))
```



Zufalls-Tensoren:

- Normalverteilung: „Mittelwert und Standardabweichung als Input PyTorch-Tensoren führen automatisch zur erwarteten Shape“

```
t = torch.normal( mean=torch.zeros(size=(3, 1)),      #Mittelwert
                  std=torch.ones(size=(3, 1))       #Standardabweichung
                  )
```

- Gleichverteilung:

```
t = torch.rand(3,1) #erstellt einen Tensor mit Zufallszahlen zwischen 0 und 1
                      (gleichmäßig verteilt) mit Shape = [3,1]
```



Tensor-Operationen:

- Matrix-Multiplikation: $C = A * B$

```
A = torch.tensor([[1,2],[3,4]], dtype=torch.int32)
B = torch.tensor([[1,0],[0,1]], dtype=torch.int32)
C = torch.matmul(A,B)
```

- Affine Transformation: $y = W * x + b$

```
W = torch.rand(2,2, dtype=torch.float32)
b = torch.rand(2,1, dtype=torch.float32)
x = torch.tensor([1,1], dtype=torch.float32)
y = torch.matmul(W,x) + b
```



Elementweise Tensor-Operationen:

```
A = torch.ones((2, 2))  
B = torch.square(A)      #Quadratur, analog A^2  
C = torch.sqrt(B)        #Quadratwurzel  
D = B + C               #Addition und Subtraktion
```

Konkatenation Tensor-Operationen:

```
E = torch.cat((A, B), dim=0) # Konkatenation entlang der Achse 0  
E.shape  
torch.Size([4, 2])  
F = torch.cat((A, B), dim=1) # Konkatenation entlang der Achse 1  
F.shape  
torch.Size([2, 4])
```



Slicing and reshaping Tensors:

- Slicing (Teile eines Tensors auswählen)

```
x = torch.arange(10)
print(x[2:6])      # Elemente 2 bis 5
print(x[:5])       # Erste 5 Elemente
print(x[::2])       # Jedes zweite Element
print(x[-3:])      # Letzte 3 Elemente
```

- Reshaping (ändert die Form ändern)

```
x = torch.arange(12)
y = x.reshape(3, 4)
print(y)
print(y.shape)    # torch.Size([3, 4])
```



Um Tensoren effizient verarbeiten zu können, besteht die Möglichkeit die Verarbeitung an ein **device** auszulagern: GPU, TPU, ...

```
device = torch.device("cpu") # cpu, cuda, ipu, xpu, mkldnn, opengl, opencl, ideep,  
# hip, ve, fpga, maia, xla, lazy, vulkan, mps, meta,  
# hpu, mtia  
  
if torch.cuda.is_available():  
    device = torch.device("cuda")  
  
elif torch.backends.mps.is_available():  
    device = torch.device("mps")  
  
print(device)  
  
T = torch.tensor([1.,2.], dtype=torch.float32)  
T = T.to(device)
```

`torch.backends.mps.is_built()` means "PyTorch was compiled with MPS support";
`torch.backends.mps.is_available()` means "and it's actually usable right now".

Übung:

- Führe im Jupyter Notebook verschiedene Tensor Operationen durch

01.05

Automatic Differentiation



Motivation:

Automatic Differentiation ermöglicht es in PyTorch automatisiert Gradienten von Tensor Operationen zu berechnet. Das bedeutet, man muss die Ableitung mit der Kettenregel nicht mehr von Hand ausrechnen.

Das Trainieren von neuronalen Netzwerken entspricht einem Optimierungsproblem.

Das Ergebnis y_{hat} eines NN wird mittels einer Loss Funktion mit dem zu erwarteten Ergebnis y verglichen und dabei wird der Verlust $\text{loss} = L(y_{\text{hat}}, y)$ berechnet. (Ein Maß dafür, wie gut oder schlecht der errechnete Wert den wahren Wert trifft.)

Ziel ist es den loss zu minimieren.

Um das (lokale) Minimum der Loss Funktion zu finden, kann man das Gradientenabstiegsverfahren (Gradient Descent) nutzen. Hierfür muss man Gradienten berechnen (Ableitungen der Loss-Funktion nach den Parametern des NN) und das nimmt einem PyTorch Automatic Differentiation ab.



Automatic Differentiation

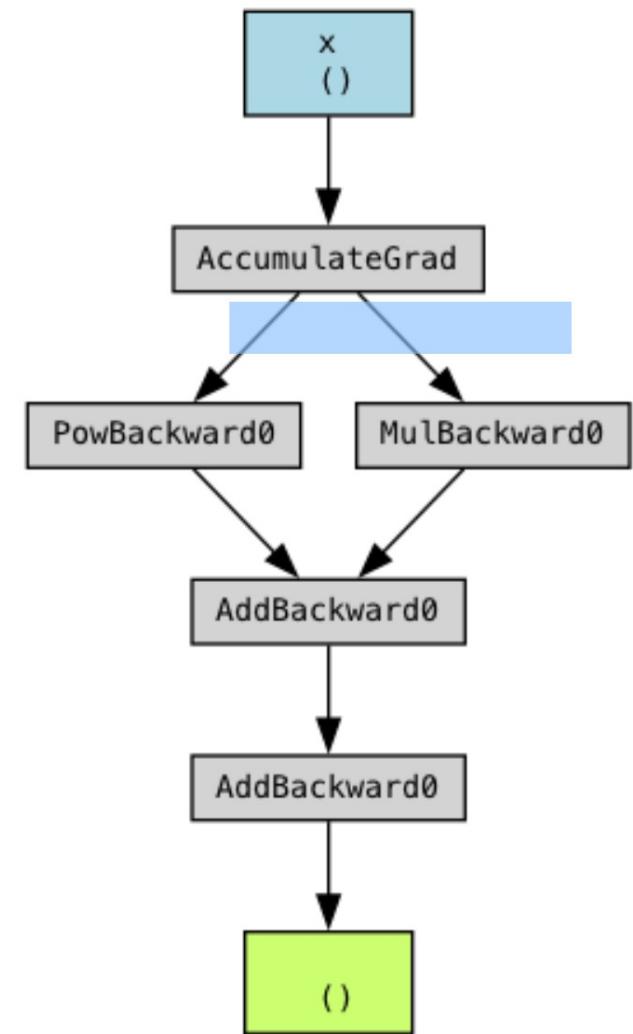
Any `torch.Tensor` with `requires_grad=True` tells PyTorch:

“Track all operations on this tensor so I can compute gradients later.”

```
input_var = torch.tensor(3.0, requires_grad=True)
result = torch.square(input_var)
result.backward()
gradient = input_var.grad
```

```
-----
x = torch.tensor(2.0, requires_grad=True)
y = x ** 3 + 2 * x + 1 # Build computation graph
print(y).                # tensor(13., grad_fn=<AddBackward0>)
```

At this point torch has built a computation graph.



Automatic Differentiation

Backpropagation

Betrachten wir die Funktion $loss = L(\hat{y}, y)$. Mit $\hat{y} = \sigma(z)$ und $z = xW + b$, W, b Parameter

Um das Gradientenabstiegsverfahren anwenden zu können, müssen wir die Gradienten $\frac{\partial L}{\partial W}$ und $\frac{\partial L}{\partial b}$ bezüglich der Parameter W und b von L berechnen. Mit der Kettenregel lässt sich der Gradient berechnen:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial \sigma} * \frac{\partial \sigma}{\partial z} * \frac{\partial z}{\partial W}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial \sigma} * \frac{\partial \sigma}{\partial z} * \frac{\partial z}{\partial b}$$

PyTorch nimmt uns die Arbeit ab:



Automatic Differentiation

Backpropagation

Wir starten mit einer Loss-Funktion L und einer forward-Funktion, die einen Vorhersage-Wert $y_{\text{hat}} = \text{forward}(W, b, x)$ liefert. Weil wir die Parameter W und b so anpassen wollen, dass die forward-Funktion für gegebene Paare aus (x,y) möglichst korrekte Ergebnisse zurückliefert, sind wir an den Gradienten $\frac{\partial L}{\partial W}$ und $\frac{\partial L}{\partial b}$ interessiert.

```
W = torch.randn(out_features, in_features, requires_grad=True) # (out, in)
b = torch.randn(out_features, requires_grad=True) # (out,)
y_hat = forward(W, b, x) #für gegebenes x berechnen wir y_hat
loss = (y_hat - y).pow(2).mean() #für gegebenes x berechnen wir die mittlere
                                 quadratische Abweichung
if W.grad is not None: W.grad.zero_() #Falls der Gradient existiert,
if b.grad is not None: b.grad.zero_() #dann setzen wir ihn auf Null

loss.backward() # Hier beginnt die PyTorch Magie. Es wird der
                Gradient mit Hilfe der Kettenregel zurück
                propagiert.
```

Jetzt können wir den Gradienten im Gradientenabstiegsverfahren nutzen um W und b schrittweise anzupassen.

Übung:

- Backpropagation im Jupyter Notebook

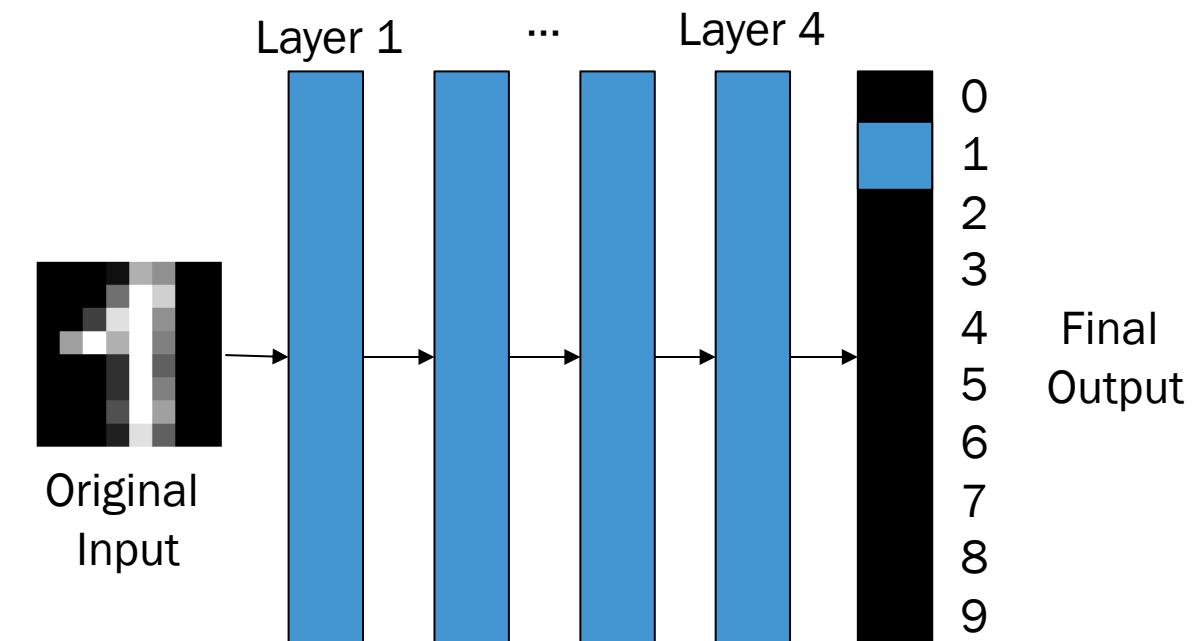
01.06
Neural Networks
& Deep Learning



Neuronale Netze & Deep Learning

- Deep Learning ist ein spezifischer Teilbereich des maschinellen Lernens.
- Ziel ist das Erlernen von Repräsentationen aus Daten, bei dem der Schwerpunkt auf dem Erlernen aufeinanderfolgender Schichten (Layer) von zunehmend aussagekräftigen Repräsentationen liegt.
- Die Anzahl der Layer definiert die Tiefe des Modells.
- Beim Deep Learning werden diese layered Repräsentationen mithilfe von Modellen gelernt, die als neuronale Netze bezeichnet werden.
- Für unsere Zwecke ist Deep Learning ein mathematisches Framework zum Erlernen von Repräsentationen aus Daten.
- Man kann sich ein tiefes Netzwerk als einen mehrstufigen Prozess der Informationsdestillation vorstellen, bei dem Informationen mehrere Filter durchlaufen und zunehmend verfeinert (verbessert) werden.

- Loading handwritten digits:



Neuronale Netze & Deep Learning

Deep Learning in 3 Schritten (1/3)

Transfrom

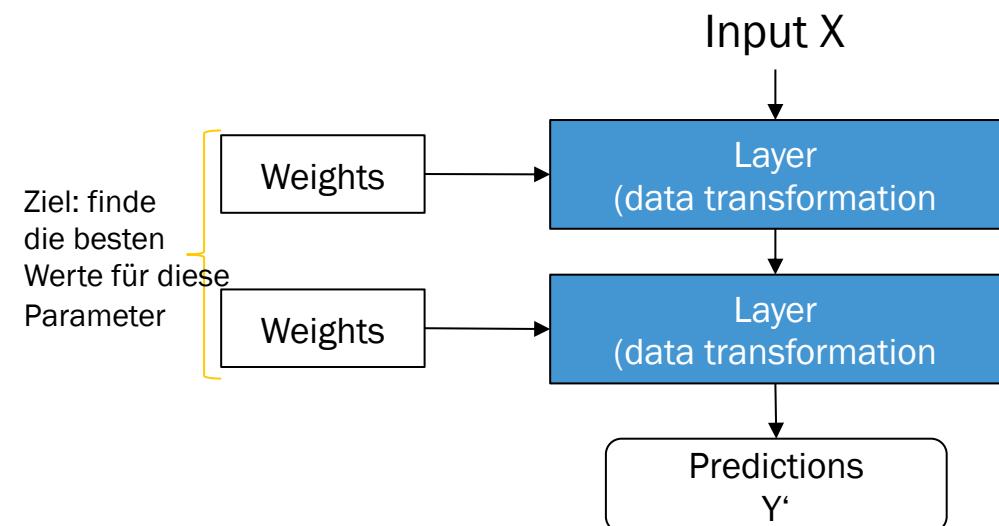
- Beim maschinellen Lernen geht es darum, Eingaben (z. B. Bilder) Zielwerten (z. B. der Bezeichnung „Katze“) zuzuordnen.
- Dies geschieht durch die Beobachtung vieler Beispiele für Eingaben und Zielwerte.
- Tiefe neuronale Netze führen diese Zuordnung von Eingaben zu Zielwerten über eine tiefe Abfolge einfacher Datentransformationen (Layers) durch, wobei diese Transformationen durch die Konfrontation mit Beispielen gelernt werden.
- Die von einem Layer durchgeführte Transformation wird durch ihre Gewichte parametrisiert. Gewichte werden als Parameter eines Layers bezeichnet.
- Lernen bedeutet, einen Satz von Werten für die Gewichte aller Layer in einem Netzwerk zu finden, sodass das Netzwerk Beispiel-Eingaben korrekt den zugehörigen Zieldaten zuordnet.

- Herausforderung: Wenn es Millionen von Parametern gibt und jeder Parameter Werte in R (reelle Zahlen) hat, dann gibt es $R * R * \dots * R$ mögliche Parameterkombinationen.

millionen

Das sind unter Umständen mehr Kombinationen als es Atome im Universum gibt.

Das sieht danach aus, als ob man die Nadel im Heuhaufen finden müsste.

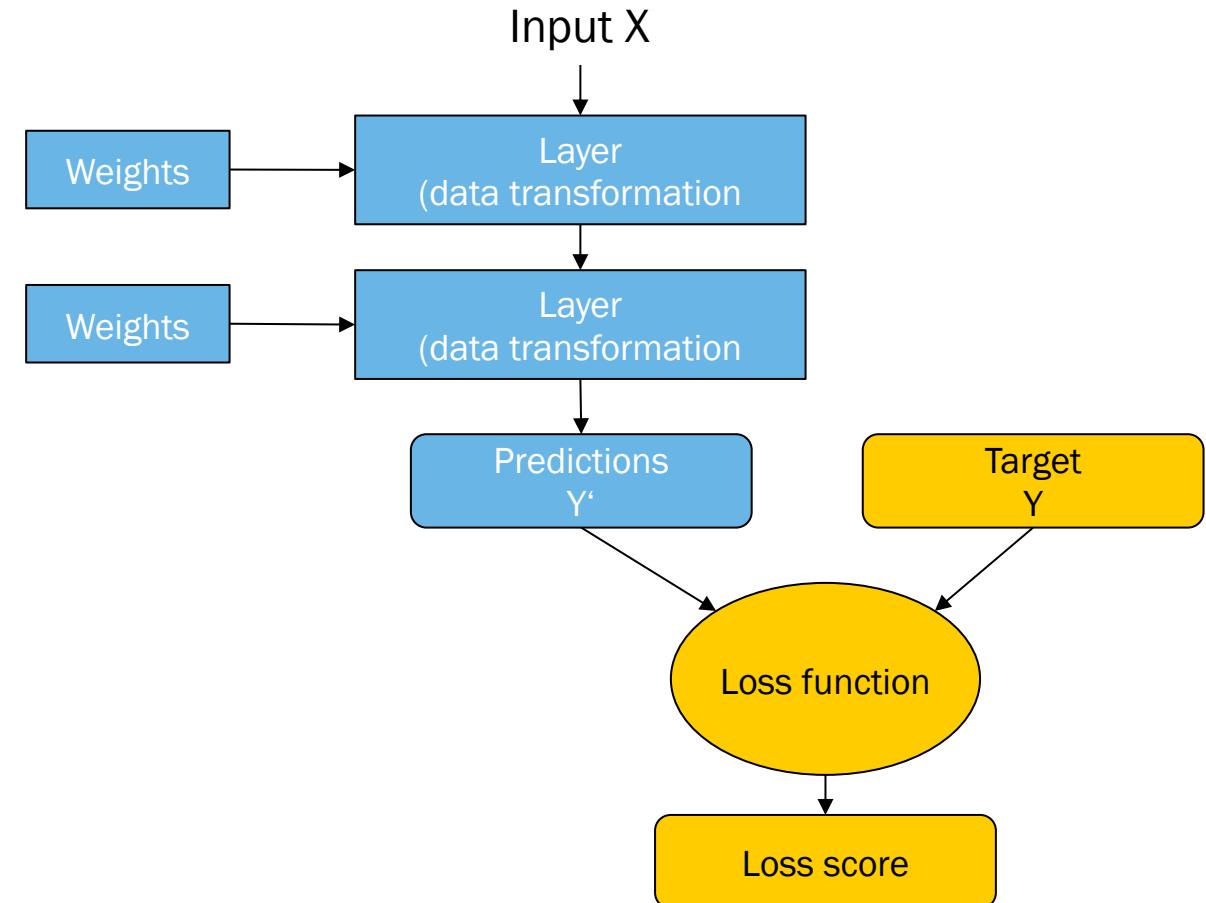


Neuronale Netze & Deep Learning

Deep Learning in 3 Schritten (2/3)

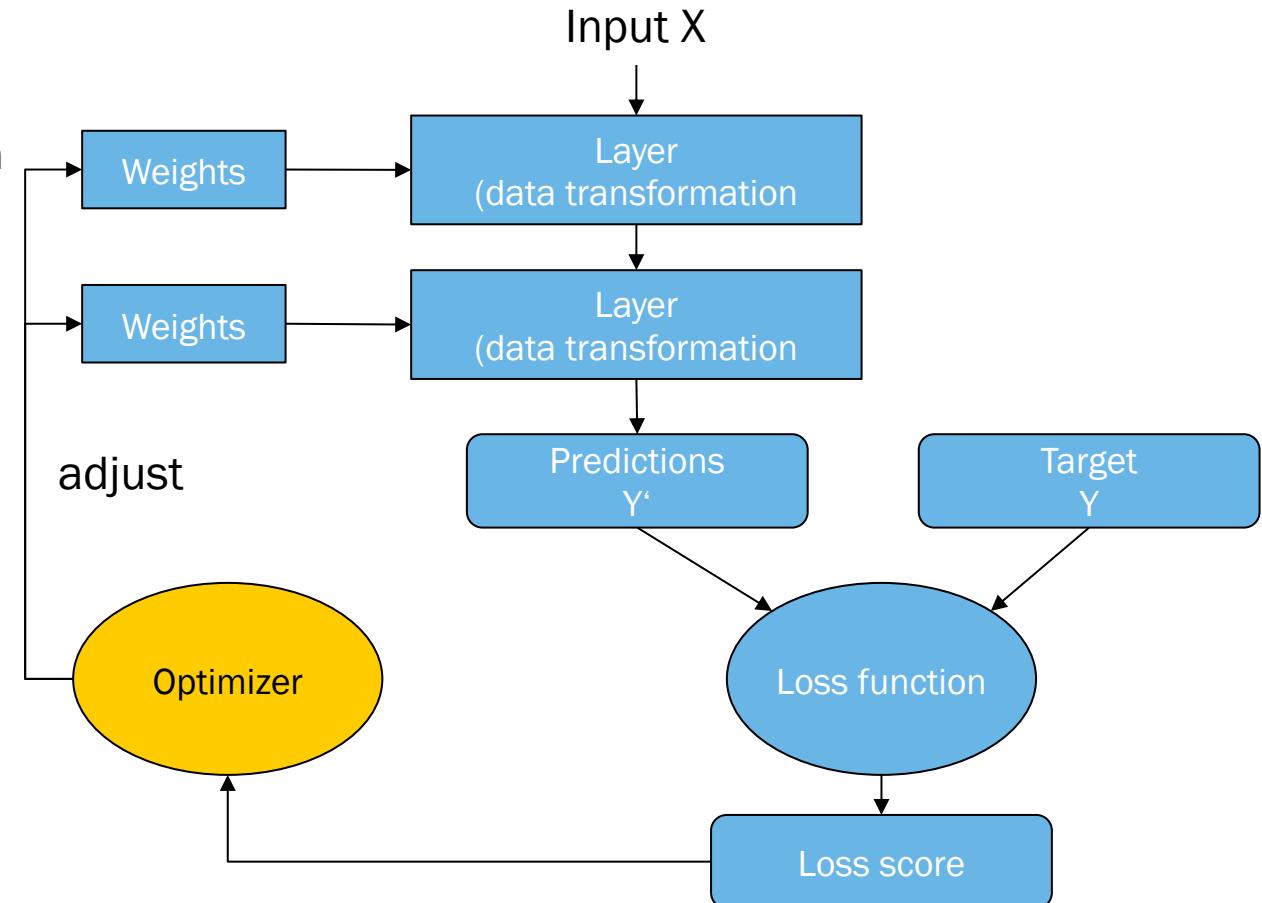
Control

- Um etwas zu kontrollieren, muss man es zunächst beobachten und messen können.
- Um die Ausgabe (output, predictions) eines neuronalen Netzwerks zu kontrollieren, muss man messen können, wie weit diese Ausgabe von den Erwartungen (target) abweicht.
- Dies ist die Aufgabe der Verlustfunktion (loss function) des Netzwerks, die manchmal auch als objective function oder cost function bezeichnet wird.
- Die Verlustfunktion nimmt die Vorhersagen (predictions) des Netzwerks und das tatsächliche Ziel (target) und berechnet einen Abstandsscore (loss score), der angibt, wie gut das Netzwerk bei diesem speziellen Beispiel abgeschnitten hat.



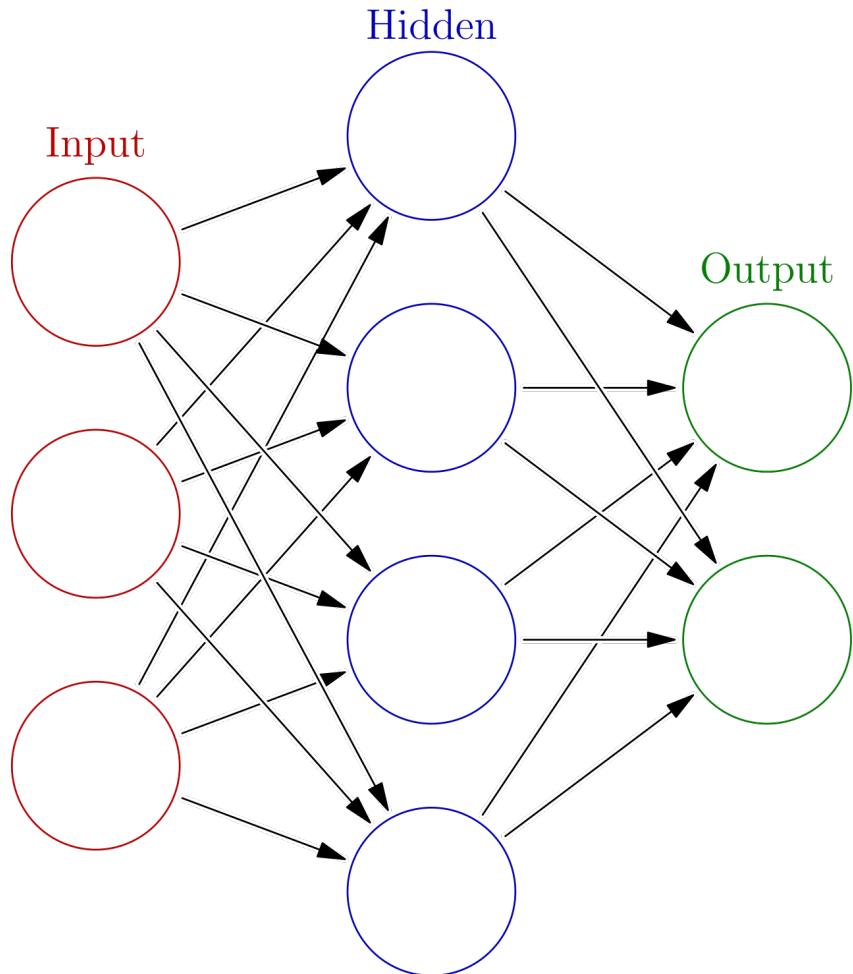
Adjust

- Der Trick beim Deep Learning besteht darin, den Loss score als Feedback-Signal zu verwenden, um die Gewichte ein wenig anzupassen, und zwar in einer Richtung, die den Loss score für das aktuelle Beispiel reduziert.
- Diese Anpassung ist die Aufgabe des Optimizer.
- Der Optimizer nutzt den Backpropagation-Algorithmus
- Backpropagation ist der zentrale Algorithmus im Deep Learning.



Neuronale Netze & Deep Learning

A very short Introduction into Neural Networks

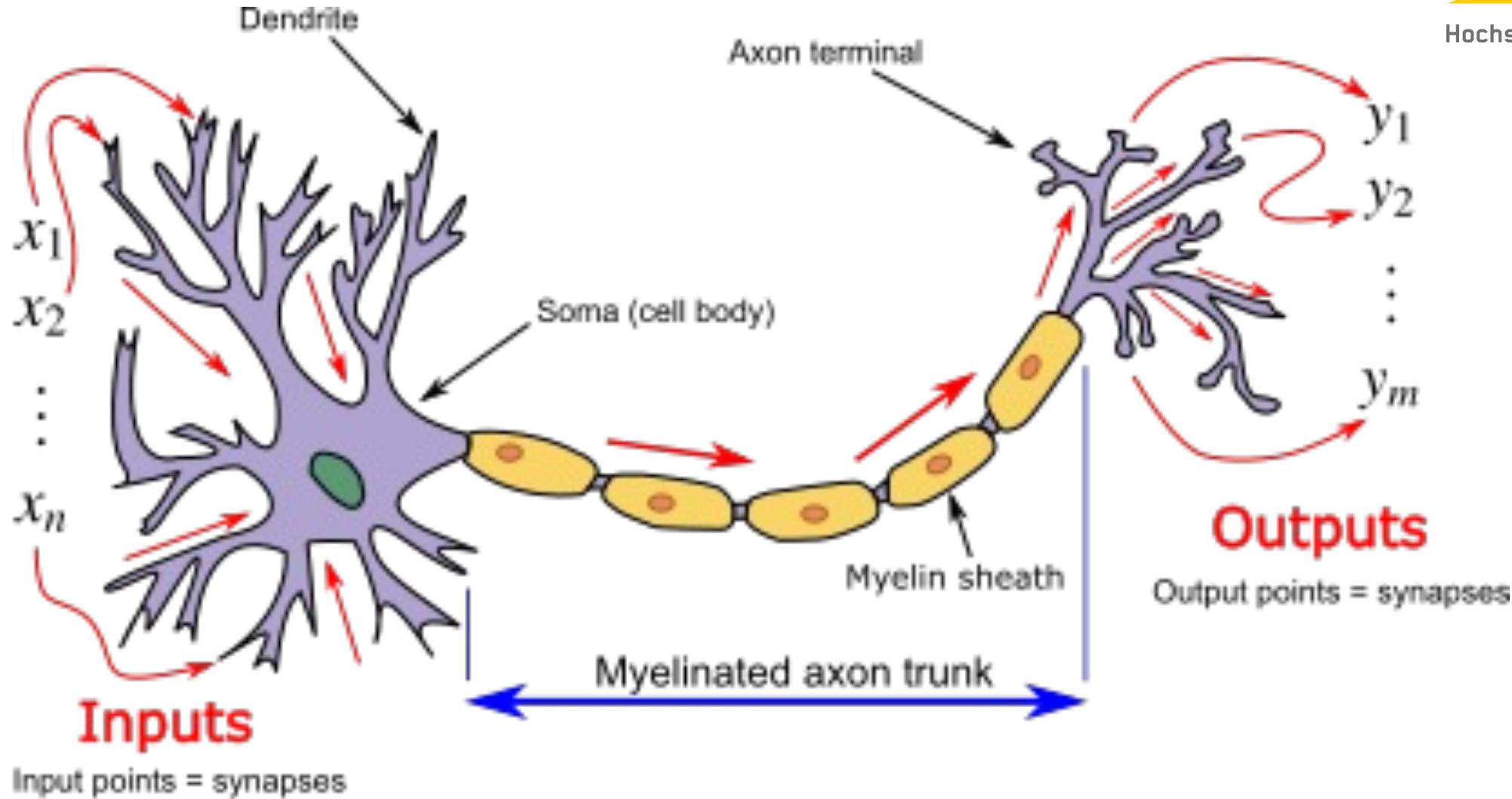


- In machine learning, a **neural network (NN)** is a model inspired by the structure and function of biological neural networks in animal brains.
- An NN consists of connected units or **nodes** called artificial neurons, which loosely model the neurons in the brain.
- These are connected by **edges**, which model the synapses in the brain.
- Each artificial neuron receives **signals** from connected neurons, then processes them and sends a signal to other connected neurons. The "signal" is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs, called the **activation function**.
- The strength of the signal at each connection is determined by a **weight**, which adjusts during the learning process.



Neuronale Netze & Deep Learning

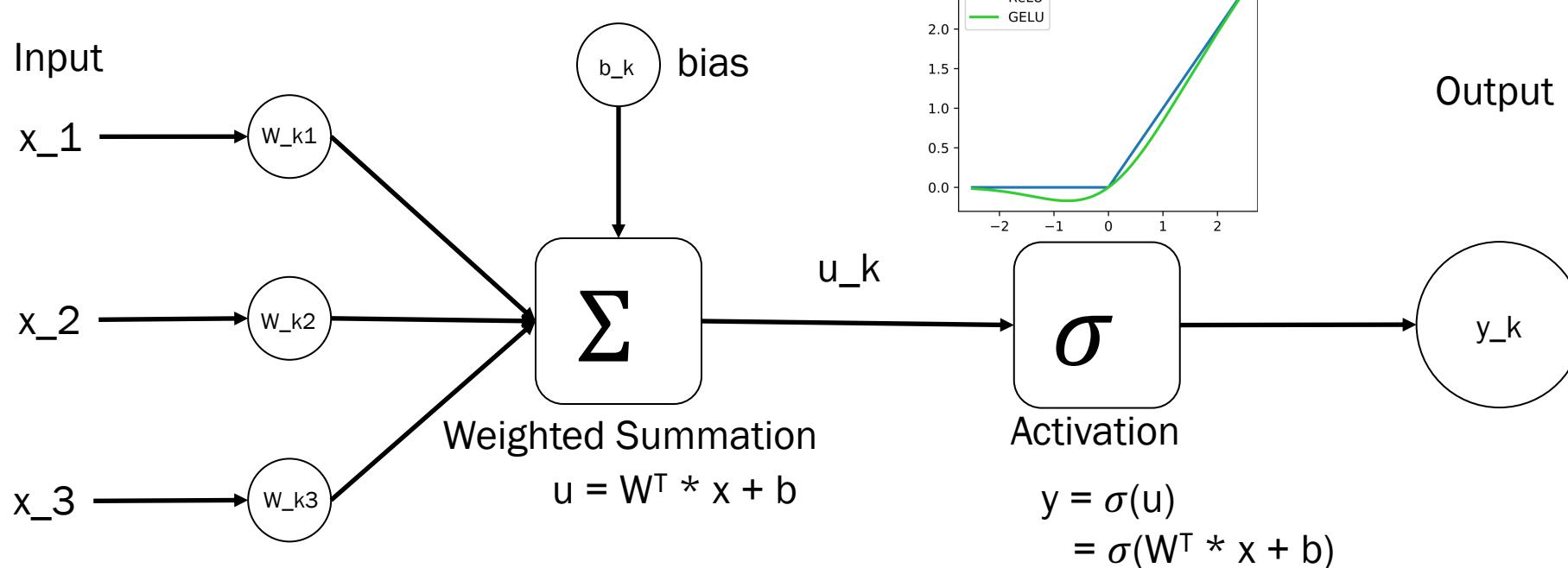
A very short Introduction into Neural Networks



Neuronale Netze & Deep Learning

A very short Introduction into Neural Networks

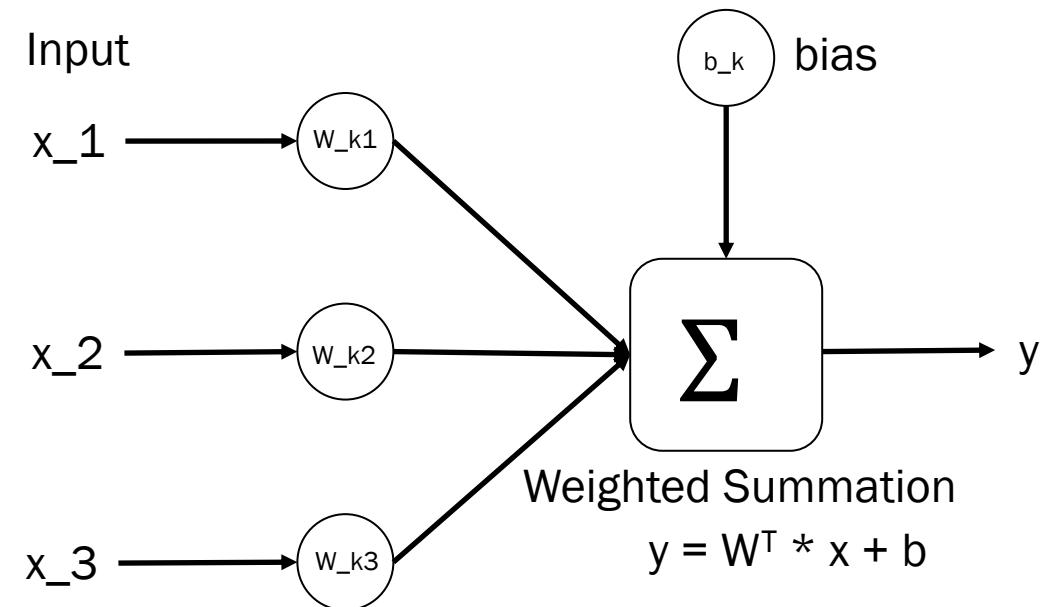
Each neuron of the hidden layer does the following computation



Neuronale Netze & Deep Learning

Lineare Layer in PyTorch

```
import torch  
from torch import nn  
L = nn.Linear(3, 1, bias=True)  
y = L(x)
```

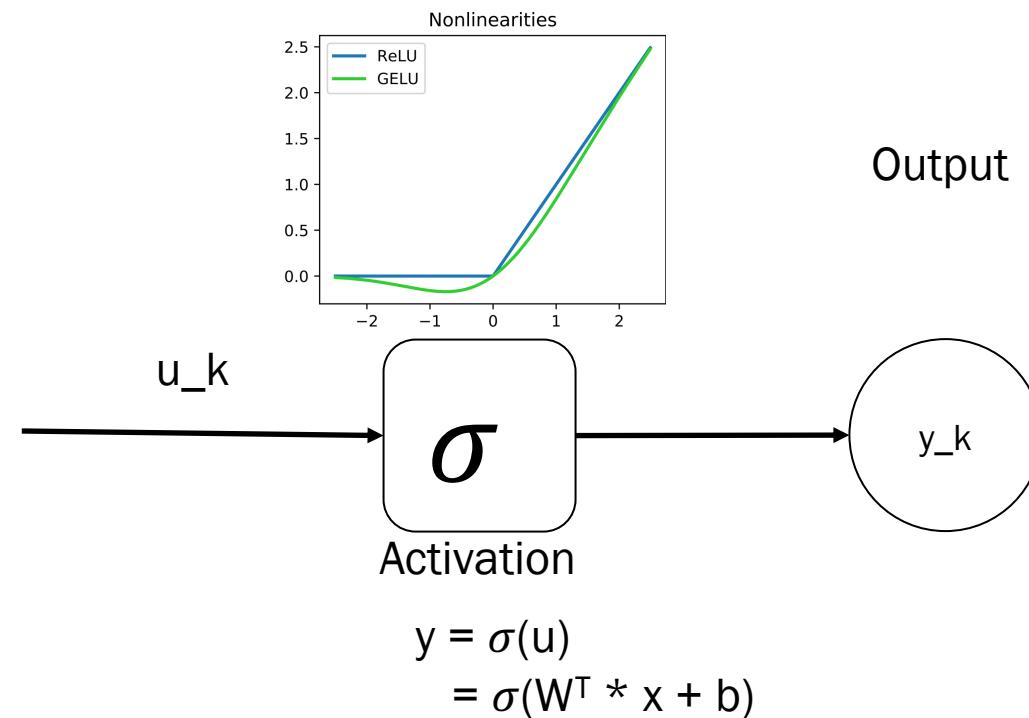


Neuronale Netze & Deep Learning

Aktivierungsfunktionen

Das Schachteln von Linearen Layern ist ausreichend dafür, dass ein NN zu einem universellen Funktions-Approximator wird. Dafür ist es notwendig Nicht-Linearitäten einzubauen. Das erfolgt mit sogenannten Aktivierungsfunktionen:

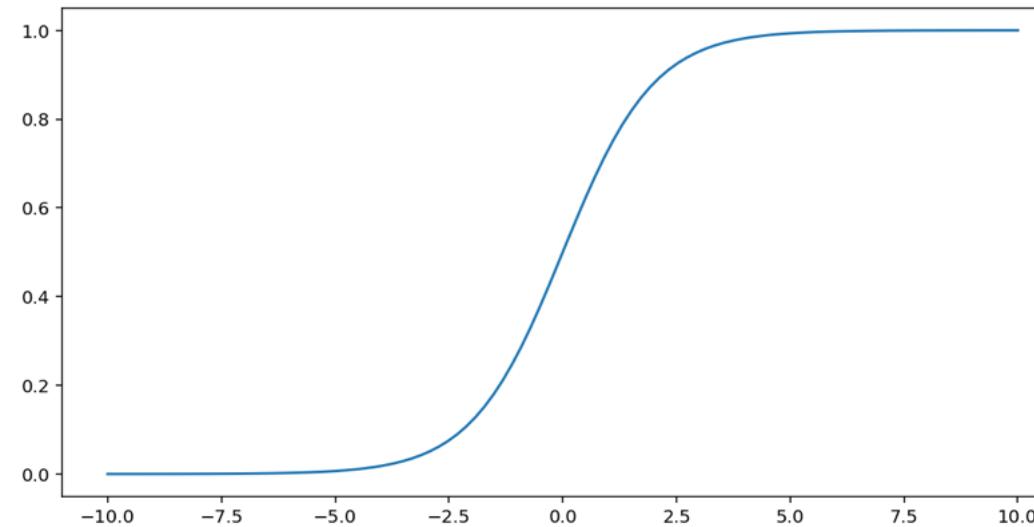
- Sigmoid: torch.nn.functional.sigmoid
- Tanh: torch.nn.functional.tanh
- ReLU: torch.nn.functional.relu
- GELU: torch.nn.functional.gelu



Aktivierungsfunktionen

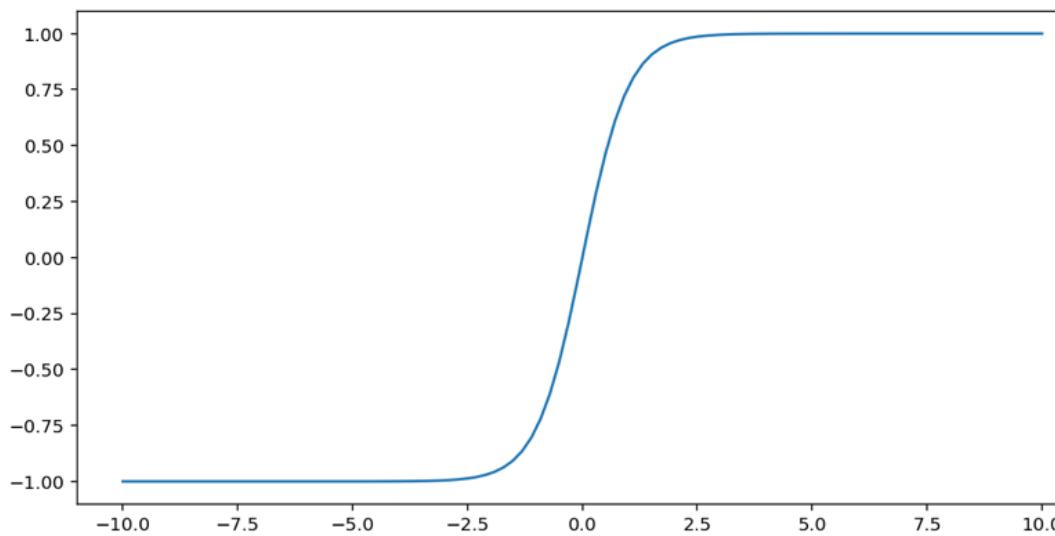
Sigmoid-Funktion:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



Tanh-Funktion

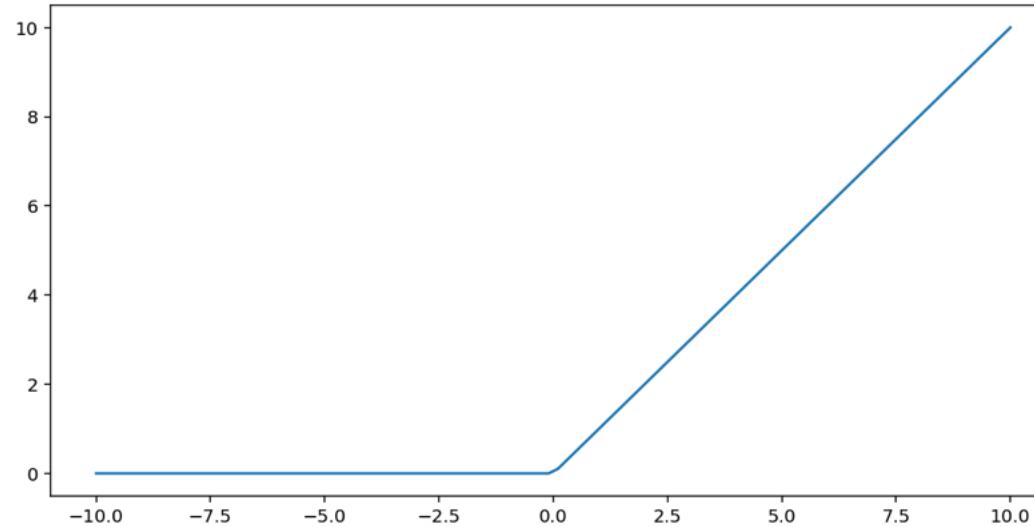
$$\sigma(z) = \tanh z$$



Aktivierungsfunktionen

ReLU:

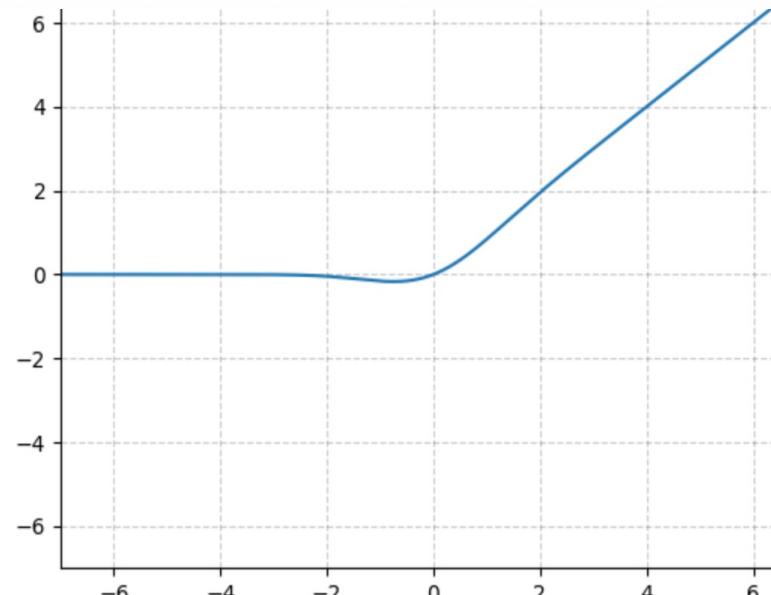
$$\sigma(z) = \max(0, z)$$



GELU:

$$\sigma(z) = z * \phi(z)$$

where $\Phi(x)$ is the Cumulative Distribution Function for Gaussian Distribution



Aktivierungsfunktion

Die Wahl der Aktivierungsfunktion hängt stark vom Lernproblem, der Architektur des NN und den Eigenschaften der Daten ab:

Aktivierung	Formel / Idee	Typische Verwendung	Vorteile	Nachteile
Sigmoid	$\sigma(x) = \frac{1}{1 + e^{-x}}$	Klassische binäre Klassifikation (z. B. am Ausgang)	Ausgabe zwischen 0 und 1 (gut interpretierbar als Wahrscheinlichkeit)	Vanishing Gradient, langsames Lernen
Tanh	$\tanh(x) \in [-1,1]$	Rekurrente Netze, ältere Architekturen	Zentriert um 0 (besser als Sigmoid)	Immer noch vanishing gradients
ReLU	$\max(0, x)$	Standard in CNNs, MLPs	Einfach, effizient, kein vanishing gradient für positive Werte	"Dead neurons" (Gradient = 0 bei $x < 0$)
Leaky ReLU / Parametric ReLU	$\max(\alpha x, x)$	CNNs, tiefe MLPs	Verhindert dead neurons	Leicht mehr Rechenaufwand
ELU / SELU	Glatt bei 0, ähnlich wie ReLU aber mit negativen Werten	Selbst-normalisierende Netze	Bessere Stabilität	Etwas teurer
GELU	$x \cdot P(X \leq x), \text{ mit } X \sim N(0,1)$	Transformer, BERT, GPT usw.	Glatter als ReLU, theoretisch motiviert (stochastisch gewichtetes ReLU)	Etwas teurer in der Berechnung
Softmax	$\frac{e^{x_i}}{\sum_j e^{x_j}}$	Am Ausgang bei Mehrklassen-Klassifikation	Wahrscheinlichkeitsverteilung über Klassen	Nicht in Hidden Layers nutzen

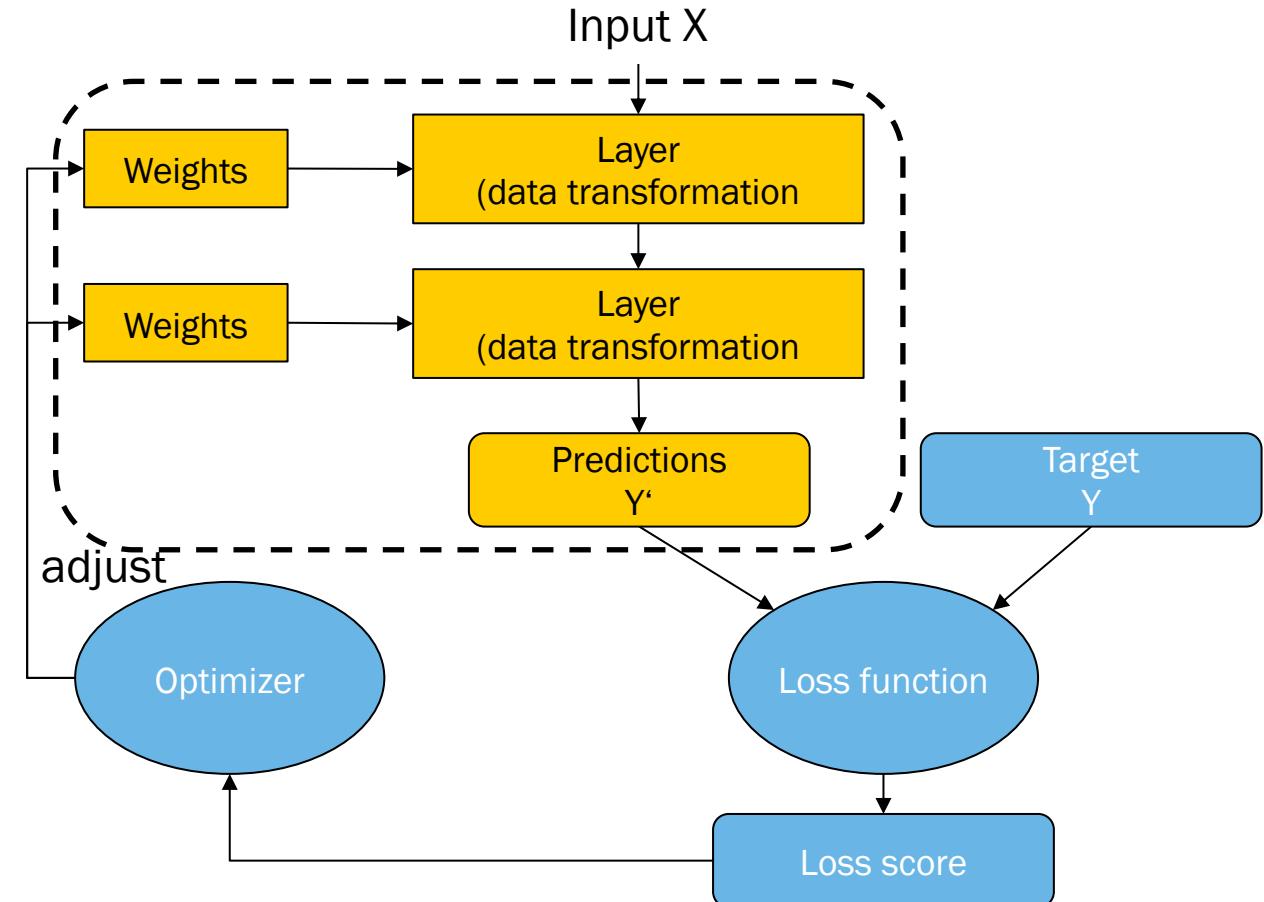
Aktivierungsfunktion

Welche Aktivierungsfunktion für welches Lernproblem?

Lernproblem	Merkmale	Empfohlene Aktivierungen
Binäre Klassifikation	1 Ausgangsneuron, $y \in 0,1$	Hidden: ReLU / GELU / LeakyReLU Output: Sigmoid
Mehrklassen-Klassifikation	K Ausgangsneuron, $y \in 1,2, \dots, K$	Hidden: ReLU / GELU / ELU Output: Softmax
Regression (kontinuierlich)	n Ausgangsneuron, $y \in R^n$	Hidden: ReLU / LeakyReLU / Tanh (je nach Datenbereich) Output: linear (keine Aktivierung)
Autoencoder / Rekonstruktion	Eingabe = Ausgabe	Hidden: ReLU / LeakyReLU / GELU Output: Sigmoid (wenn Werte in [0,1])
Transformer	Token-Sequenzen, Token-Embeddings	Hidden: GELU (Standard in BERT, GPT usw.)
Bildklassifikation (CNN)	Input: Pixelbilder	Hidden: ReLU oder LeakyReLU
Zeitreihen / RNNs / LSTMs	Sequenzen	Innerhalb RNN/LSTM: tanh und sigmoid
GANs (Generator/Discriminator)	Generator/Discriminator	Generator: LeakyReLU, Tanh (Output) Discriminator: LeakyReLU, Sigmoid (Output)

Das Herzstück von PyTorch ist die Klasse `torch.nn.Module`:

- Wiederverwendbarkeit: Module sind wie Lego-Steine.
Man kann Module erweitern, kombinieren,
verschachteln, speichern, laden und exportieren.



Module Klasse

Das Herzstück von PyTorch ist die Klasse `torch.nn.Module`:

- Wiederverwendbarkeit: Module sind wie Lego-Steine.
Man kann Module erweitern, kombinieren,
verschachteln, speichern, laden und exportieren.
- Layer und Parameter werden rekursiv registriert.
- Klare Trennung von Definition `__init__` und Berechnung `forward`.
- Forward wird durch Python-Magie innerhalb `__call__` aufgerufen

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.act = nn.GELU()

    def forward(self, x):
        x = self.act(self.fc1(x))
        x = self.fc2(x)
        return x

# Modell instanzieren
model = SimpleMLP(128, 256, 10)

# Modell ausführen
x = torch.rand(128, dtype=torch.float32)
y = model(x)

# Alle trainierbaren Parameter anzeigen
print(sum(p.numel() for p in model.parameters()))
```



Übung: SimpleNN

- Unser erstes Neuronales Netz in PyTorch

01.07

Loss Function



Loss Function

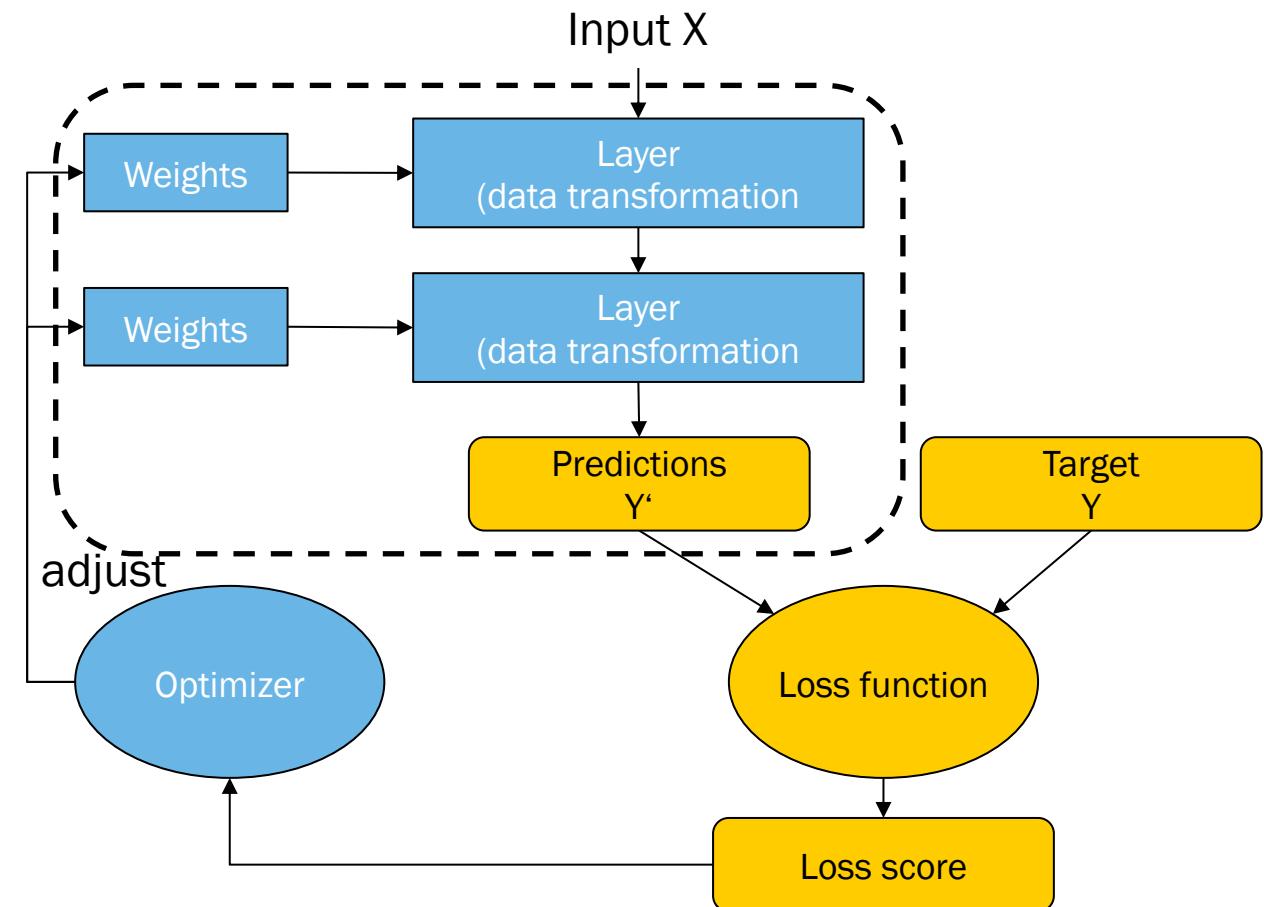
Die Loss Function, auch Verlustfunktion genannt, quantifiziert, den Unterschied zwischen den vorhergesagten Werten eines Modells und den tatsächlichen Werten.

Je kleiner der Wert der Loss Funktion, desto genauer ist das Modell; eine Anpassung des Modells erfolgt dann, um diesen Loss-score zu minimieren.

Es existieren für die verschiedenen Lernprobleme verschiedene Arten von Loss Funktionen, wie beispielsweise die Mean Squared Error (MSE) oder Cross-Entropy, und deren Wahl hängt von der spezifischen Aufgabe ab, die zu lösen ist:

Grundsätzlich gilt:

- Mean Squared Error (MSE) wird bei Regressions- und
- Cross-Entropy bei Klassifikations-Problemen genutzt.



Loss Function

Klassifikation mit Cross-Entropy Loss

One-Hot Vektor:

Angenommen wir haben C Klassen. Ein Label $y=j$ wird als One-Hot Vektor dargestellt. Das ist ein C-dimensionaler Vektor bestehend aus 0-Einträgen und **einem** 1-Eintrag an der Stelle j: $y = (0, 0, \dots, 1, 0, \dots, 0) \in \mathbb{R}^C$

Allgemein:

$$y_j = \begin{cases} 1, & \text{wenn } j = \text{true class} \\ 0, & \text{sonst} \end{cases}$$

Beispiel:

- 3 Klassen (rot, grün, blau): rot = (1,0,0), grün = (0,1,0), blau = (0,0,1)
- 10 Klassen (0,1,2,3,4,5,6,7,8,9): 0 = (1,0,0,0,0,0,0,0,0,0), 1 = (0,1,0,0,0,0,0,0,0,0), ... 9 = (0,0,0,0,0,0,0,0,0,1)



Loss Function

Klassifikation mit Cross-Entropy Loss

Logits:

Ein Klassifikations-Modell gibt als Ergebnis je Klasse einen Wert aus. Diesen nennen wir **logit**. Wir haben als Output einen Vektor mit logits: $z = (z_1, \dots, z_C)$

Softmax:

Mit der Softmax-Funktion können wir jedem logit eine Wahrscheinlichkeit zuordnen:

$$p_j = \frac{e^{z_j}}{\sum_{k=1}^C e^{z_k}} \quad \text{für } j = 1, \dots, C$$

Damit gilt: $p_j \in [0,1]$ und $\sum p_j = 1$.



Loss Function

Klassifikation mit Cross-Entropy Loss

Der Cross-Entropy Loss für ein Sample ist der negative Logarithmus der Wahrscheinlichkeit der richtigen Klasse:

$$L(z, y) = - \sum_{j=1}^C y_j * \log(p_j)$$

Alle y_j sind 0 bis auf einen (true class). Daraus ergibt sich bei $j = \text{true class}$:

$$L(z, y) = -\log p_j$$

Das bedeutet:

- Ist p_j hoch, also nahe an 1, dann ist der Logarithmus nahe an 0. Modell ist sicher und richtig → Loss ist klein
- Ist p_j niedrig, also nahe an 0, dann ist der Logarithmus sehr groß. Modell ist unsicher und falsch → Loss ist groß



Loss Function

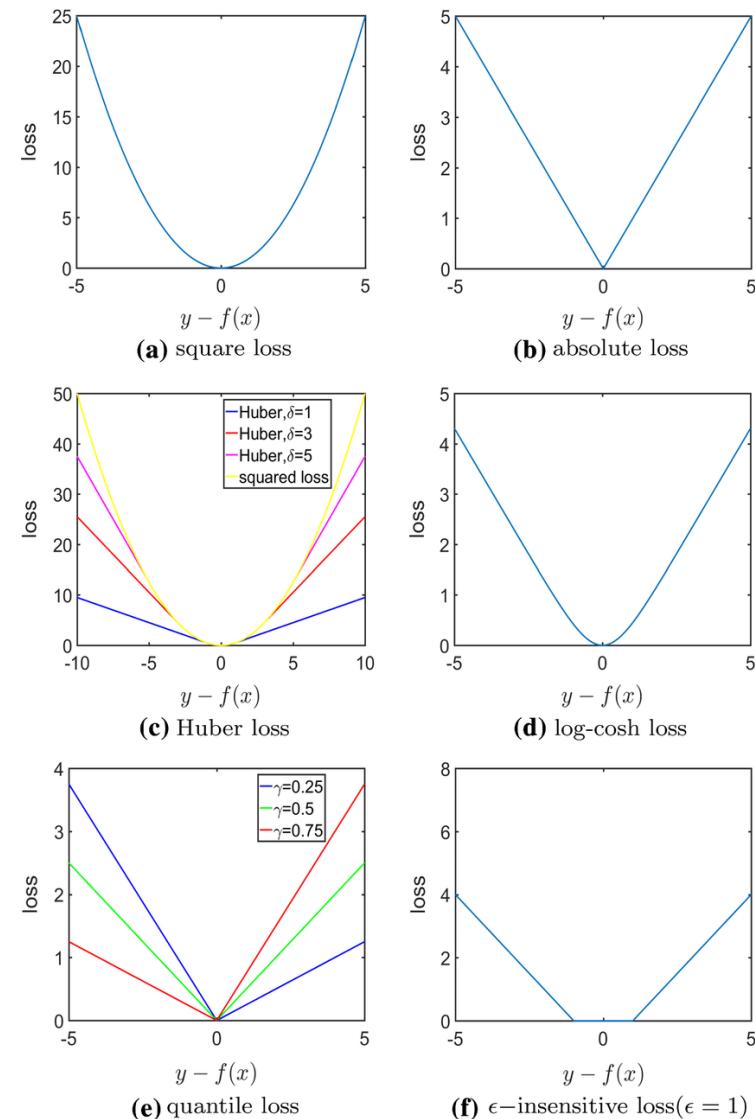
Regression mit Mean Squared Error Loss

Die Mean Squares Error Loss (MSE) quantifiziert die mittlere quadratische Abweichung zwischen Vorhersage y_{hat} und tatsächlichem Wert y . Sie ist definiert als:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y - \hat{y})^2$$

Das führt dazu, dass größere Fehler deutlich stärker bestraft werden als kleinere Abweichungen.

Die Kurvenform des MSE ist eine U-Kurve (Parabel)



Weitere Verlust-Funktionen

<https://www.ibm.com/de-de/think/topics/loss-function>

<https://www.datacamp.com/de/tutorial/loss-function-in-machine-learning>



01.08 Optimizer



Optimizer

Der Optimizer in PyTorch aktualisiert die Gewichte (Parameter) des Modells, damit die Loss-Funktion minimiert wird.

Er nutzt also die Gradienten, die durch Backpropagation berechnet wurden, und führt einen Schritt in Richtung geringerer Fehler aus.

Ein Modell hat Parameter θ (Gewichte, Bias).

Die Backpropagation berechnet die Gradienten $\partial L / \partial \theta$.

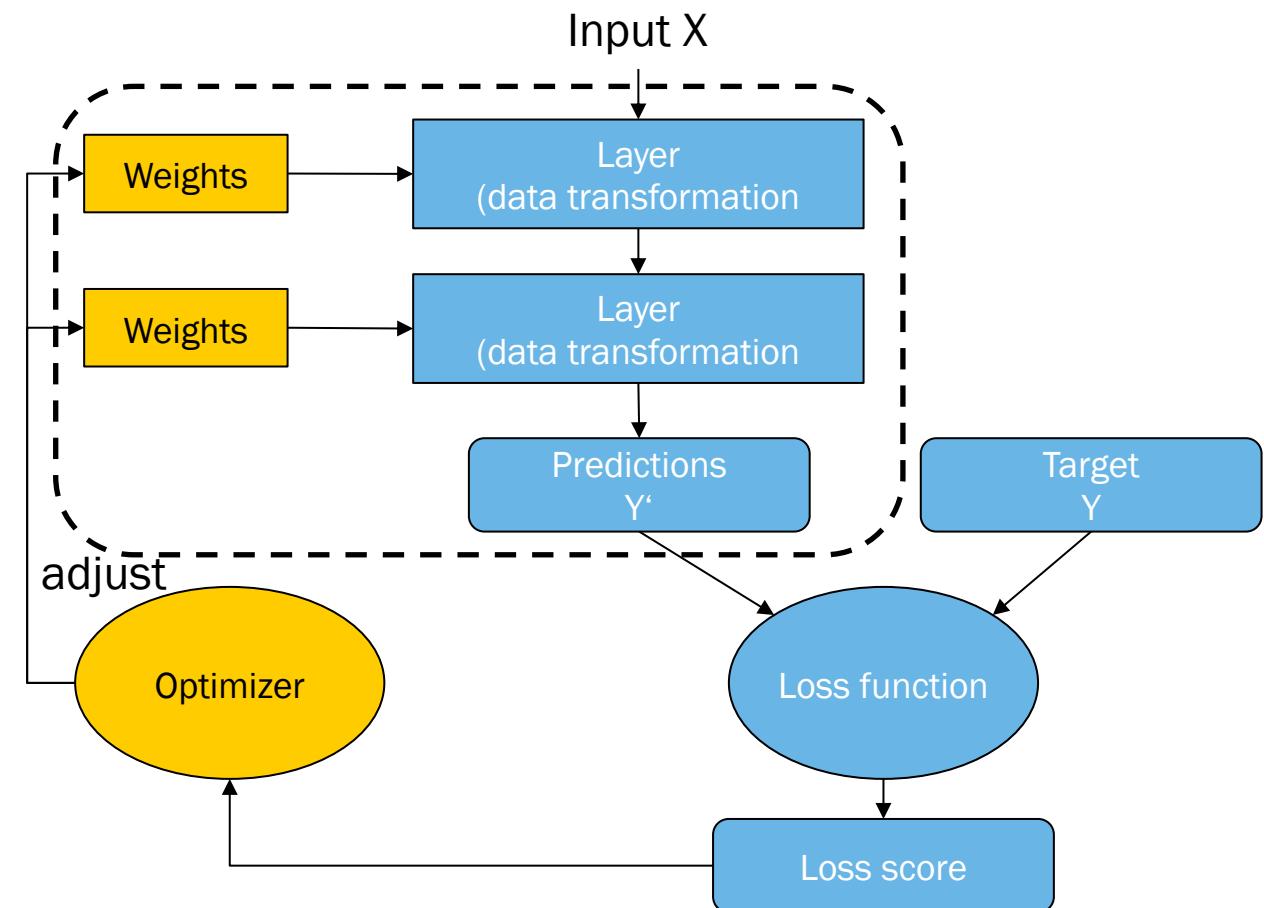
Der Optimizer wendet dann eine Regel an, um θ zu aktualisieren:

$$\theta \leftarrow \theta - \eta * \partial L / \partial \theta.$$

η = Lernrate (learning rate)

$\partial L / \partial \theta$ = Gradient des Losses

Das ist der Kern von Gradient Descent.



Optimizer

```
import torch
import torch.nn as nn
import torch.optim as optim

model = nn.Linear(10, 1)
Loss_fn = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

for epoch in range(100):
    x = torch.randn(32, 10)
    y = torch.randn(32, 1)
    optimizer.zero_grad()                      # Gradienten löschen
    y_hat = model(x)                          # Vorwärtsdurchlauf
    loss = loss_fn(y_hat, y)
    loss.backward()                            # Gradienten berechnen
    optimizer.step()                           # Parameter aktualisieren
```



PyTorch bietet verschiedene Varianten, die von `torch.optim.Optimizer` erben:

Optimizer	Formel / Idee	Typische Anwendung
SGD (Stochastic Gradient Descent)	$\theta = \theta - \eta \cdot \nabla L$	Mini-Batch Gradient Descent
SGD + Momentum	Momentum fügt dem Optimierungsschritt eine Art Trägheit hinzu. Momentum (speichert den gleitenden Mittelwert der Gradienten), damit werden vergangene Gradienten-Werte weitergetragen. Vermeidet Zickzackbewegungen.	Schnellere Konvergenz
Adam	Adaptive Lernrate + Momentum	In Deep Learning der Standard
RMSProp	Skaliert Lernrate basierend auf Gradient-Varianz	Recurrent Networks, instabile Loss-Landschaften
AdamW	Wie Adam, aber mit sauberem Weight Decay: „Adam with decoupled Weight Decay“	State-of-the-art für NLP, Computer Vision



Loss Function & Optimizer

Übung: Eigener Optimizer (4a) und weiteres Neuronales Netz (4b) in PyTorch



01.09

Classification



Lerne, handgeschriebene Ziffern zu erkennen:

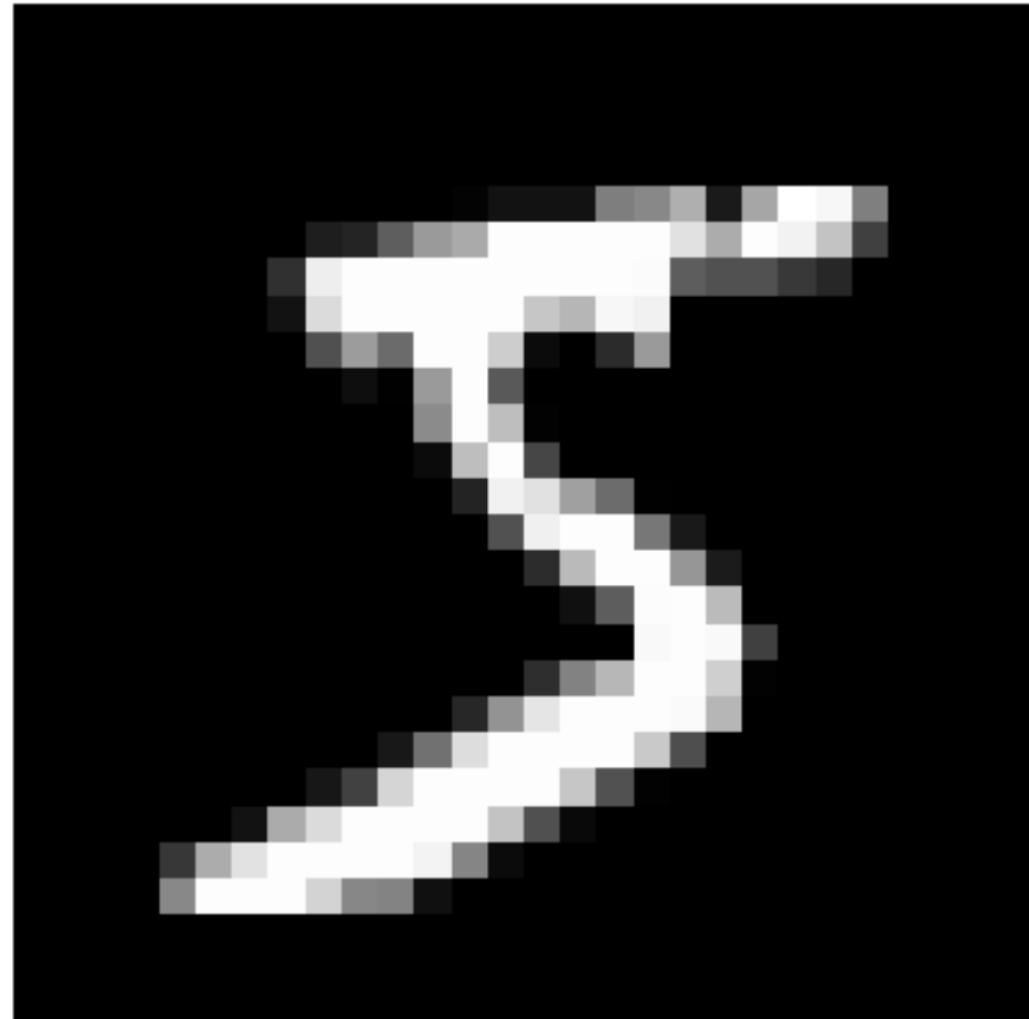
1. Lade die MINST Daten von openML

<https://www.openml.org>

verwende dafür die Funktion

```
from sklearn.datasets import  
fetch_openml  
mnist = fetch_openml('mnist_784',  
version=1, as_frame=False)
```

Rohdaten von MNIST sind Graustufenbilder mit Pixelwerten in [0, 255]



Neuronale Netze - MINST

2. Normalisiere die Daten:

i) Teile alle Werte durch 255, dann haben wir Werte im Intervall [0,1]

ii) Normalisiere mit Mittelwert und Standardabweichung:

$$\mu = \frac{1}{N} \sum_{i=1}^N x_i$$

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2}$$

$$x' = \frac{x - \mu}{\sigma}$$

3. Ändere die Datentypen:

Im Deep Learning verwendet man in der Regeln Float32 anstatt Float64 zur Speicherplatz-Optimierung:

- float64 = 8 Bytes pro Wert
- float32 = 4 Bytes pro Wert

4. Unterteile die Daten in Train, Validation und Test:

Man teilt die Daten in Train, Validation und Test. Jedes Set erfüllt im maschinellen Lernen eine andere Rolle.

Set	Zweck
Train	<ul style="list-style-type: none">• Anpassen der Modellparameter (Gewichte im Neuronalen Netz, Koeffizienten in einer Regression usw.)• Typisch: 60–80 % der Daten
Validation	<ul style="list-style-type: none">• Abstimmen der Hyperparameter (z. B. Lernrate, Netzarchitektur, Regularisierung)• Early Stopping (Training abbrechen, wenn sich die Validierungsleistung nicht mehr verbessert)• Typisch: 10–20 % der Daten
Test	<ul style="list-style-type: none">• Nur ganz am Ende zur unabhängigen Evaluation des finalen Modells• Wird niemals ins Training oder Hyperparameter-Tuning einbezogen• Typisch: 10–20 % der Daten

nn.Linear

nn.Linear is the basic fully-connected (dense) layer in PyTorch. It implements a linear mapping:

$$y = x \ W^T + b$$

where

- x = input of shape (in_features)
- W = weight matrix of shape (out_features, in_features)
- b = bias vector of shape (out_features,) (optional)

For a single input vector $x \in \mathbb{R}^{in}$ and output $y \in \mathbb{R}^{out}$:

$$y_j = \sum_{i=1}^{in} W_{j,i} x_i + b_j$$

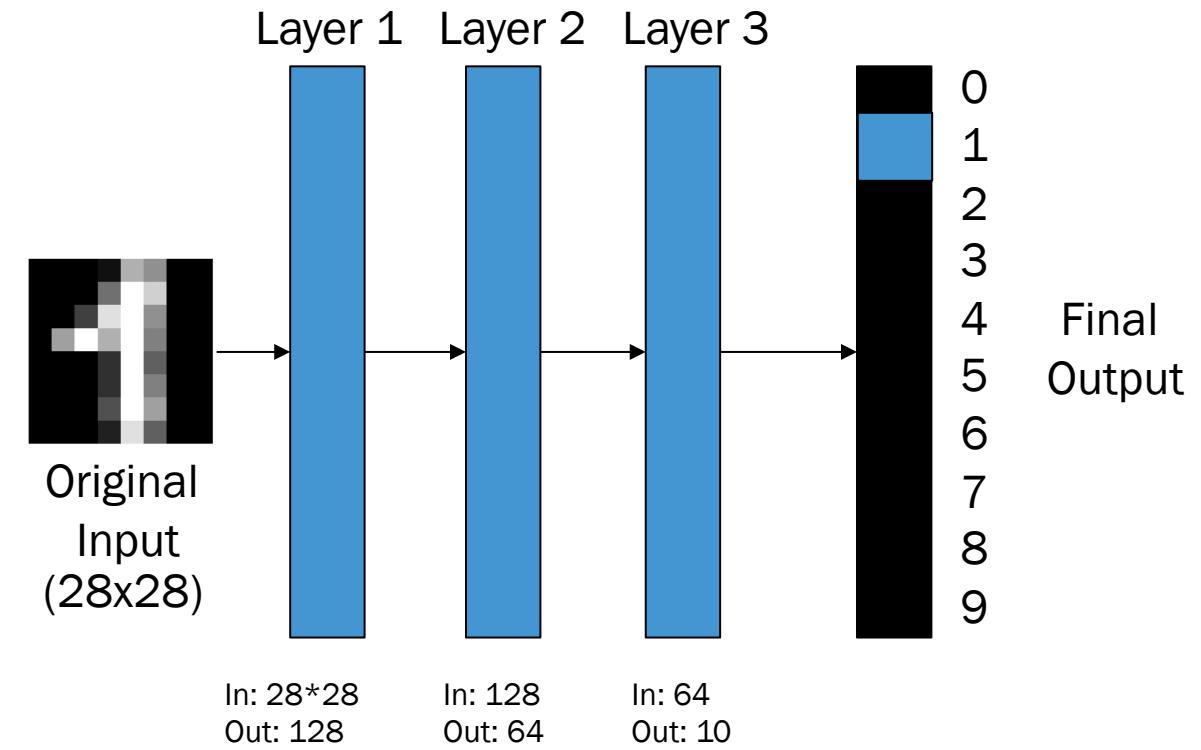
In matrix form for a batch $X \in \mathbb{R}^{N \times in}$:

$$Y = XW^T + \mathbf{1}b^T$$

(where b is broadcast across rows).

Multi Layer Perceptron

```
class MLP(nn.Module):  
  
    def __init__(self):  
        super(MLP, self).__init__()  
  
        self.fc1 = nn.Linear(28*28, 128)  
  
        self.fc2 = nn.Linear(128, 64)  
  
        self.fc3 = nn.Linear(64, 10)  
  
    def forward(self, x):  
  
        x = F.relu(self.fc1(x))  
  
        x = F.relu(self.fc2(x))  
  
        x = self.fc3(x)  
  
    return x
```



Classification

Übung: MNIST



01.10

Regression



Regression

Übung: Regression with California House Prices

01.12

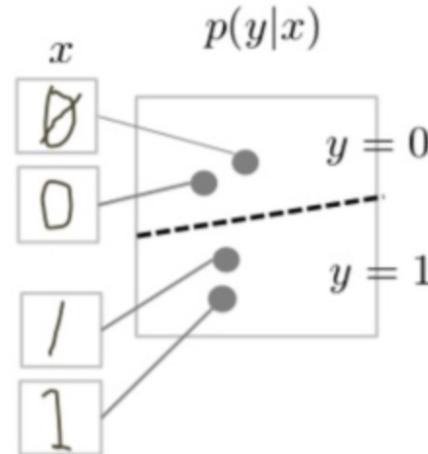
Ausblick



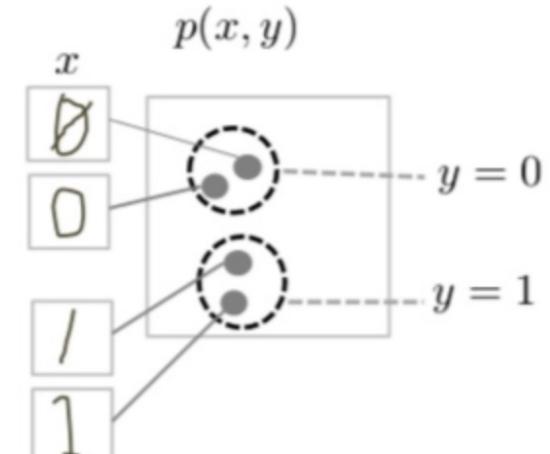
Generative versus Discriminative models

- Generative AI refers to deep-learning models that can take raw data and “learn” to generate statistically probable outputs when prompted.
- At a high level, generative models encode a simplified representation of their training data and draw from it to create a new data that are similar, but not identical, to the original data.
- A generative model could generate new photos of animals that look like real animals, while a discriminative model could tell a dog from a cat.
- A generative model includes the distribution of the data itself and tells you how likely a given example is.

- **Discriminative Model**



- **Generative Model**



Discriminative and generative models of handwritten digits.

Given a set of data instances X and a set of labels Y :

- Generative models capture the joint probability $p(X, Y)$, or just $p(X)$ if there are no labels.
- Discriminative models capture the conditional probability $p(Y | X)$.

<https://developers.google.com/machine-learning/gan/generative>

Bock 2 – Generative AI & Transformer

