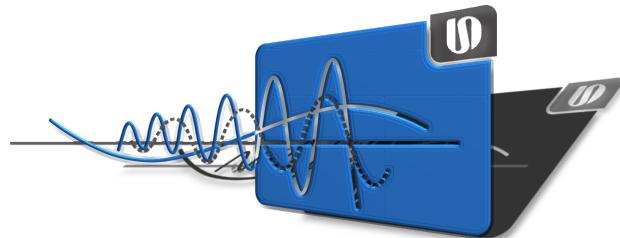

MACHINE LEARNING METHODS IN MECHANICS

ANDRÉ MIELKE



INSTITUTE OF MECHANICS, STRUCTURAL ANALYSIS,
AND DYNAMICS OF AEROSPACE STRUCTURES

University of Stuttgart

Preface

This L^AT_EX-script was automatically generated from the Jupyter notebooks used in the summer semester 2021 course MACHINE LEARNING METHODS IN MECHANICS with *Pandoc* and *XeLaTeX*. This was the second time the course took place so naturally, many errors and inconsistencies plague the content of this document. E.g., some tables look wonky and some equations may be rendered incorrectly. Take everything with a grain of salt and be cautious. There's no guarantee for anything in this document being correct. It will be improved continuously and suggestions are always welcome.

- André

Copyright Notice

THE CONTRIBUTIONS AND MATERIALS PROVIDED HERE ARE PROTECTED BY COPYRIGHT AND SHALL SOLELY BE USED FOR STUDY PURPOSES EXCLUSIVELY BY PARTICIPANTS WHO HAVE ENROLLED FOR THE TEACHING CLASS IN QUESTION WITHIN THE SCOPE OF UNIVERSITY TEACHING. ANY OTHER USE, IN PARTICULAR DISCLOSURE OF CONTRIBUTIONS AND MATERIALS OR PASSWORDS TO OUTSIDE PARTIES (INCLUDING STUDENTS ENROLLED IN OTHER CLASSES) AND ANY PUBLICATION (INCLUDING ELECTRONIC PUBLICATION ON THE INTERNET) ARE STRICTLY PROHIBITED FOR COPYRIGHT REASONS. VIOLATIONS WILL BE PURSUED LEGALLY. IN SUCH CASES, YOU WILL BE SUBJECT TO LEGAL SANCTIONS (IN PARTICULAR COMPENSATION AND CRIMINAL PROSECUTION).

DIE HIER ZUR VERFÜGUNG GESTELLTEN BEITRÄGE UND MATERIALIEN SIND URHEBERRECHTLICH GESCHÜTZT UND AUSSCHLIESSLICH ZUM STUDENTISCHEN GEBRAUCH DURCH AUSSCHLIESSLICH DIE TEILNEHMER*INNEN DER JEWELIGEN LEHRVERANSTALTUNG IM RAHMEN DER UNIVERSITÄREN LEHRE BESTIMMT. JEGLICHE ANDERE NUTZUNG, INSbesondere die WEITERGABE DER BEITRÄGE UND MATERIALIEN ODER DES PASSWORTS AN AUSSENSTEHENDE (EINSCHLIESSLICH STUDIERENDE ANDERER KURSE) UND JEDE VERÖFFENTLICHUNG (INSbesondere auch elektronische Veröffentlichungen im Internet) SIND AUS URHEBERRECHTLICHEN GRÜNDEN STRIKT UNTERSAGT. ZUWIDERHANDLUNGEN WERDEN VERFOLGT. IN SOLCHEN FÄLLEN MÜSSEN SIE MIT RECHTLICHEN SANKTIONEN (INSbesondere SCHADENSERSATZ UND STRAFRECHTLICHER VERFOLGUNG) RECHNEN.

Contents

1 Python Introduction and Jupyter	1
1.1 Python	1
1.2 Jupyter	1
1.3 Operators and strings	2
1.4 Modules	3
1.5 Variables	4
1.6 Data structures	6
1.7 Loops	7
1.8 Control Structures	8
1.9 Functions	9
1.10 File Handling	10
1.11 Classes	13
2 Linear Algebra	16
2.1 NumPy	16
2.2 Why Linear Algebra	17
2.3 Array operations	21
2.4 Linear combinations	25
2.5 Span	25
2.6 Linear independence	27
2.7 Matrix operations	27
2.8 Norms	31
2.9 Quadratic forms	32
2.10 Data Visualization with Matplotlib	33
2.10.1 Plotting	33
2.10.2 3D Plotting	41
2.11 Derivatives	42
2.12 Gradient	48
2.13 Hessian	53
2.14 Taylor Series	55
2.15 Pandas	57
2.15.1 Plotting	62
2.15.2 Saving and Loading Data	65
3 Optimization	66
3.1 Derivatives	66
3.2 Gradient	70
3.3 Hessian	75
3.4 Taylor Series	77
3.5 Matrices	80
3.5.1 Vectors and Scalars	80
3.5.2 Two Vectors	84
3.5.3 Derivative of a Matrix Product	84
3.6 Unconstrained Optimization	84
3.6.1 Scalar Optimization	85
3.6.2 Multivariate Optimization	86

3.7	Constrained Optimization	89
3.8	Numerical Optimization	92
3.8.1	Gradient Descent	93
3.8.2	Stopping Criteria	99
4	Statistics	101
4.1	Probability	101
4.1.1	Definitions	101
4.1.2	Discrete Sample Space	102
4.1.3	Continuous Sample Space	103
4.2	Conditional, Joint, and Marginal Probability	106
4.2.1	Bayes' Theorem	109
4.3	Bayes' Theorem	109
4.3.1	Prior Probability	109
4.3.2	Posterior Probability	109
4.3.3	Example	110
4.4	(Conditional) Independence and the Chain Rule of Probability	112
4.4.1	Conditional Independence	113
4.4.2	Chain Rule of Conditional Probability	113
4.5	Expectation	114
4.6	Variance and Covariance	115
4.6.1	Variance	115
4.6.2	Covariance	117
4.6.3	Correlation	118
4.6.4	Statistical Independence and Covariance	120
4.6.5	Further Relations	121
4.6.6	Affine Transformation of RVs	122
5	Linear Regression	123
5.1	The Machine Learning Paradigm	123
5.1.1	Feed-forward Process / Forward Modeling	126
5.1.2	Feedback	126
5.1.3	Machine Learning Pipeline	126
5.2	Linear Regression	127
5.3	Least Squares and Gradient Descent	132
5.4	Quantifying Fitness	137
5.5	Generalization	139
5.6	Bias and Variance	147
5.7	Regularization	154
5.8	Gradient Descent Variants	158
5.8.1	Momentum Method	159
5.8.2	Nesterov Accelerated Gradient	159
5.8.3	AdaGrad	159
5.8.4	RMSPROP and AdaDelta	160
5.9	Logistic Regression	160
5.9.1	Deep Learning Teaser	165
6	Logistic Regression and Artificial Neural Networks	165

6.1	Binary Cross-Entropy Cost Function	165
6.2	Logic Gates	167
6.2.1	OR gate	167
6.2.2	NOR, AND, NAND	171
6.2.3	XOR gate	173
6.3	Gradient of the Binary Cross-Entropy Loss Function	181
6.4	Multinomial Classification	186
6.5	Artificial Neural Networks	190
6.5.1	Artificial Neuron	190
6.5.2	Feedforward Neural Network	195
6.6	Backpropagation	197
6.7	Summary	201
7	Convolutional Neural Networks	203
7.1	Convolutional Neural Networks	203
7.2	Convolutions	209
7.3	Pooling	215
7.4	Types of Convolutions	218
7.4.1	Volume Convolutions	218
7.4.2	Padded Convolutions	221
7.4.3	Strided Convolutions	223
7.4.4	Dilated Convolutions (Atrous Convolution)	223
7.4.5	Transposed Convolution	224
7.5	CNN Forward and Backward Pass	226
7.6	Transfer Learning	227
7.6.1	Starting Points for Optimization	227
7.6.2	Transfer Learning	234
7.7	Examples	236
7.8	Activation Functions	243
7.8.1	Sigmoid	243
7.8.2	tanh	244
7.8.3	ReLU	244
7.8.4	Leaky ReLU/Parametric Rectifier	245
7.8.5	ELU	246
7.8.6	SELU	247
7.9	Data Normalization/Preprocessing	248
7.10	Batch Normalization	251
7.11	Weight Initialization	252
7.11.1	SELU	258
7.12	Hyperparameter Optimization	259
7.12.1	Cross-validation	260
7.12.2	Grid Search	262
7.12.3	Randomized Search	268
7.12.4	Bayesian Search	272
7.12.5	Evolutionary Hyperparameter Optimization	280
7.13	Visualizing Hyperparameter Search Results (optional)	281
7.13.1	TensorFlow	281
7.14	Data Augmentation	284

7.14.1	Reflections	285
7.14.2	Translations	287
7.14.3	Rotations	288
7.14.4	Noise	289
7.14.5	Automatic Augmentation with Keras	290
7.14.6	Relevant Data	302
7.15	Visualizing Convolutional Neural Networks	302
7.15.1	Saliency Maps	302
7.15.2	Dense Layer ActivationMaximization	309
7.16	Semantic Segmentation	313
7.16.1	Bitmasks	313
7.16.2	Segmentation Models	314
7.16.3	Example	315
7.16.4	Segmentation Loss Functions	325
7.16.5	Intersection over Union Loss	326
8	Recurrent Neural Networks (RNNs)	328
8.1	Recurrent Neural Networks	328
8.2	RNN Loss and Backpropagation Through Time	333
8.2.1	Loss	333
8.2.2	Backpropagation Through Time (BPTT)	335
8.3	Vanishing Gradients and Truncated BPTT	337
8.3.1	The Problematic Gradient	337
8.3.2	Gradient Clipping	339
8.3.3	Truncated BPTT	339
8.3.4	Different Architectures	341
8.4	Gated Recurrent Units	341
8.5	Long Short-Term Memories	343
8.6	Deep RNNs and Bidirectional RNNs	345
8.7	Bidirectional RNNs	347
9	Probabilistic and Statistical Methods	350
9.1	Bernoulli Distribution	350
9.2	Gaussian/Normal Distribution	351
9.2.1	Gaussian Parameter Estimation	356
9.2.2	Multivariate Maximum Likelihood Estimation	357
9.2.3	Central Limit Theorem	358
9.3	Naive Bayes Algorithms	358
9.3.1	Training Naive Bayes	359
9.4	Maximum Likelihood Estimation in Regression	363
9.5	Maximum a Posteriori Estimation and Regression	364
9.6	Bayesian Regression	365
9.7	Advanced Regularization	371
9.7.1	Early Stopping	371
9.7.2	Dropout	373
10	Dimensionality Reduction and Clustering	378
10.1	The Curse of Dimensionality	378

10.2 Eigendecomposition	380
10.3 Singular Value Decomposition	386
10.4 Principal Component Analysis	389
10.4.1 The Algorithm	394
10.4.2 PCA by SVD	395
10.4.3 Summary	395
10.5 Generative PCA	396
10.6 t-distributed Stochastic Neighborhood Embedding	399
10.7 K-Nearest Neighbors	411
10.8 K-Means	417
11 Checkpoint I Repetition	423
11.1 Linear Regression	423
11.2 Classification	423
11.3 Artificial Neural Networks	424
11.4 Convolutional Neural Networks	429
11.5 Semantic Segmentation	432
11.6 Recurrent Neural Networks	433
11.7 Probabilistic and Statistical Methods	435
11.8 Dimensionality Reduction and Clustering	436
12 Generative Modeling	441
12.1 Generative Models	441
12.2 Generative Adversarial Networks	446
12.2.1 Zero Sum Game	447
12.2.2 DCGANs	450
12.2.3 Conditional GANs	451
12.2.4 Data Augmentation	451
12.3 Autoencoders	452
12.4 Variational Autoencoders	454
12.4.1 VAE Loss	455
12.5 Reparametrization	456
13 Machine Learning Hard- and Software	459
13.1 Machine Learning Hardware	459
13.1.1 GPUs	460
13.1.2 ASICs	461
13.1.3 FPGAs	462
13.1.4 Summary	463
13.2 How to Identify ML Opportunities	463
13.3 Machine Learning Platforms	466
13.4 Quantum Machine Learning	467
13.4.1 Quantum Computing	467
13.4.2 Quantum-enhanced Machine Learning	472
13.4.3 Quantum Neural Networks	472
14 Reinforcement Learning	476
14.1 Reinforcement Learning	476

14.2 Markov Decision Processes	477
14.3 Q -Learning	481
14.3.1 Experience Replay	484
14.4 Policy Gradients	485
14.4.1 Variance Reduction	487
14.5 The Actor-Critic Algorithm	489
14.6 Summary and Examples	489
14.6.1 Summary	489
14.6.2 Examples	490
15 Physics-informed Machine Learning	492
15.1 Solving ODEs with Parameterized Functions	492
15.2 Physics-informed Neural Networks	493
15.3 Common Problems with PINNs	496
15.3.1 Spectral Bias	496
15.3.2 Vanishing Gradients	501
15.4 Noether's Theorem	506
15.5 Equivariant Neural Networks	508
16 Checkpoint II	509
16.1 Generative Modeling	509
16.2 ML Hard- and Software	510
16.3 Reinforcement Learning	512
17 Where to Go from Here	514
17.1 Papers	514
17.2 Online Info	514
17.3 Further Techniques	514
17.4 Current Developments	515
17.5 Create Your Own Solutions	515

1 Python Introduction and Jupyter

1.1 Python

Python's design goals are easy accessibility, easy development processes, and generality. Unlike other programming languages used in science and engineering, like C++ or Fortran, it is not a compiled language. The processes of compilation and linking is completely omitted from the development cycle when running pure Python. Python is an interpreted language. The interpreter is also called Python and can be used on almost any operating system.

There are two ways to use Python. In a shell, you can start the interpreter by invoking the python interpreter by simply typing `python` and hitting return. On Windows, the interpreter is often in `C:\pythonxx`, where `xx` is the version number. The easiest solution to get python for Windows systems is to install `anaconda`, which includes the `conda` package manager for python. To quit your session, press `ctrl+d`. This mode is usually only good for quick testing or small calculations, since nothing is saved and to reuse code you'd have to program functions or classes. The second way is to write a script. This is simple text in a textfile, which can later be executed by invoking the Python interpreter and providing the filename as an argument. The most comfortable way, and this is highly recommended for developing, is using an integrated development environment (IDE). These programs offer rich features and comfortable shortcuts, as well as good error recognition and solution suggestions. Big names for Python are Spyder and pyCharm.

1.2 Jupyter

In this course, you won't need to install Python on your system. Instead, we will work with Jupyter notebooks, such as the one you're looking at right now. Jupyter notebooks are successors to IPython (interactive Python) and were initially developed to support JUlia, PYThon, and R, hence the name. Since then, support for a multitude of kernels was added and it's difficult to find a widely adopted language that isn't supported. Jupyter notebooks aim to provide an easy-to-use interface for programming tasks in a presentable fashion. The notebook is subdivided into cells, of which you will find two kinds: markdown cells, such as this one, or code cells, such as the one below:

```
print("This is a code cell")
```

This is a code cell

You can change the type of a cell under "Cell" -> "Type". Markdown cells allow, well, usage of markdown, which enables rich text formatting options. See for example this markdown cheat sheet: [markdown cheat sheet](#) To edit the text of a markdown cell, double-click on it. To submit your changes and also to execute code in a code cell click on it, then hit "shift" + "enter". Try this with the following cell.

```
print(5*100/5.9)
```

84.7457627118644

The output is shown directly below the code cell. You can also directly input L^AT_EX-code (mainly math-related code) by either enclosing it in `$dollar signs\$` or starting an environment like `\begin{equation} \end{equation}`, just like you're probably used to.

That's pretty much all you need to know to work with the notebooks. For more info and a guided tour, click on "Help", then "User Interface Tour". For shortcuts, see "Help", then "Keyboard Shortcuts".

1.3 Operators and strings

Python can be used as a calculator. Standard operations include $+, -, *, /, **, //, \%$. Let's see what these do:

```
3+7
5+8
```

13

```
print("Addition: with '+': \t\t3 + 11 = ", 3 + 11)
print("Subtraction with '-': \t\t7 - 11 = ", 7 - 11)
print("Multiplication with '*': \t\t78.2 * 99.3 = ", 78.2 * 99.3)
print("Division with '/': \t\t38 / 14 = ", 38/14)
print("Exponentiation with **: \t\t2**10 = ", 2**10)
```

```
Addition: with '+':           3 + 11 = 14
Subtraction with '-':         7 - 11 = -4
Multiplication with '*':     78.2 * 99.3 = 7765.26
Division with '/':           38 / 14 = 2.7142857142857144
Exponentiation with **:      2**10 = 1024
```

Notice a few things here. We used the "print"-function, which prints all kinds of things and tries to make the output as nice as possible. The first thing printed here are **strings**, which are characters enclosed in quotes. Strings can also be manipulated by some of the operations above:

```
print("first part" + "second part")
print(3*"three times")
print("number is " + str(3))  # concatenation only works with strings
```

```
first partsecond part
three timesthree timesthree times
number is 3
```

The `\t` are tab characters, which help align output nicely. To concatenate numbers to a string, you need to use the `str()` function on it first.

We missed two of the operators mentioned above. What do they do?

```
print("?: \t67 // 11 = ", 67 // 11)
print("?: \t67 % 11 = ", 67 % 11)
```

```
?:      67 // 11 = 6
?:      67 % 11 = 1
```

A convenient way to add or subtract things is using the following abbreviated operators:

```
a = 17

a *= 7 # a = a + 7
print(a)

a -= 7
print(a)
```

119
112

1.4 Modules

To extend the base functionality of Python, extension modules can be imported. Many useful modules are delivered with the standard package, but many of those we will use have to be installed from external sources, like for example pyTorch, TensorFlow, numpy, scikit-learn and so on, which we will use at a later point. The preferred way to do this is to use a package manager. Starting with native modules, `math` offers many useful tools:

```
import math

print(math.sin(math.pi))
print(math.cos(math.pi))
print(math.exp(0.7))
```

1.2246467991473532e-16
-1.0
2.0137527074704766

Sometimes directly importing modules clutters the workspace. If you only need a small subset of the functions, classes, or constants provided by a module, you can restrict the import to those:

```
from math import sin, cos, exp, pi

print(sin(pi))
print(cos(pi))
print(exp(0.7))
```

1.2246467991473532e-16
-1.0
2.0137527074704766

Much nicer. In many cases, you do need a lot of different functions from a module and sometimes modules have long names. The workaround to this is to give an imported module a shorter alias. This is the preferred method of importing modules. It also helps avoiding masking issues, where two functions from different packages have the same name and in the end you don't know which of them is used in your script. An example would be the sine functions from `math` and `numpy`.

```
import math as m
import numpy as np

print(m.sin(0))
print(np.sin(0))
```

0.0
0.0

1.5 Variables

We already used `pi` above from the `math` module as a variable. Variables are easily initiated in Python, since Python is a dynamically typed language, meaning you don't have to tell it what kind of variable you want to save in memory. It will deduce that from the value you're assigning to the variable:

```
import math as m

a = 78.2
i = 15
s = "characters"
f = m.sin

print(type(a))
print(type(i))
print(type(s))
print(type(f))
```

```
<class 'float'>
<class 'int'>
<class 'str'>
<class 'builtin_function_or_method'>
```

This does come with detriments. E.g., you can never be sure what type a variable gets after assignment, so you should make sure that if you need a floating point variable, that you initiate it with a floating point number. This is just a safety measure, usually Python does the thinking here for the programmer. Speaking of floating point variables, perhaps you noticed the sine of π wasn't exactly zero in the example from last lesson. This is caused by machine precision, or the finite representation of numbers in computers. Some languages do not tell the truth regarding this limitation. See for example what Excel/LibreOffice Calc or similar tell you, when you ask `=IF(0,1 + 0,1 + 0,1 = 0,3;"TRUE";"FALSE")`. Compare this to the answer that Python gives:

```
print(0.1 + 0.1 + 0.1 == 0.3)
```

False

This causes some surprising effects, for example cancellation, where subtracting numbers close in value might lead to significant loss of precision. The following example is taken from <https://math.stackexchange.com/questions/1920525/why-is-catastrophic-cancellation-called->

so. Take for example the polynomial $x^2 - 1000.0x + 1.0 = 0.0$. The midnight formula gives $x_{1/2} = \frac{1000.0 \pm 999.99}{2}$

```
# roots:
x1 = (1000 + 999.99)/2
x2 = (1000 - 999.99)/2

print("Analytical roots:\t", x1, "and", x2)

# While the first is almost correct, the second one isn't at all
x1c = 1000
x2c = 0.0010005

print("x1 error:\t\t", 100*(x1 - x1c)/x1c, "%")
print("x2 error:\t\t", 100*(x2 - x2c)/x2c, "%")
```

```
Analytical roots:      999.995 and 0.004999999999954525
x1 error:            -0.000499999999995453 %
x2 error:            399.7501249370767 %
```

For more information regarding floating point arithmetic and its limitations and implications, see this exhaustive document: https://docs.oracle.com/cd/E19422-01/819-3693/ngc_goldberg.html

The `==` here is a comparison operator. These work quite intuitively. Same applies to gate operators (`^` is `xor`):

```
a = 14
b = True

print(11 < 13)
print(11 > 13)
print(11 == 13)
print(11 != a)
print(b)
print(not b)
print(11 < 13 and b)
print(11 < 13 and not b)
print(b or not b)
print((not b) ^ (not b))
```

```
True
False
False
True
True
False
True
False
True
```

```
False
```

1.6 Data structures

Many useful data structures are implemented in Python. The ones we will use most often are **lists** and **dicts**. Lists are pretty much self-explanatory, they contain a set of arbitrary things, even other lists, or even themselves:

```
a = 7

testlist = []
print(type(testlist))

testlist.append("apple")
testlist.append(15)
testlist.append(a)
testlist.append(testlist)

print(testlist)
print(testlist[-2])
```

```
<class 'list'>
['apple', 15, 7, ...]
7
```

Square brackets access a certain element in a list. You can also access list elements using negative numbers, in which case they'll start referencing elements starting from the last element in a list. You can also use slicings, choosing a range of elements from a list. We will see how these work after getting to know one of the most beautiful concepts in Python; **list comprehensions**:

```
import math as m

n = 15

squares = [x**2 for x in range(n)]

list_of_lists = [[x**2, m.sin(y)] for x in range(3) for y in range(3)]

print(squares)
print(list_of_lists)

print("Slicing: ", squares[1:3])
print("Negative elements: ", squares[-1])
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
[[0, 0.0], [0, 0.8414709848078965], [0, 0.9092974268256817], [1, 0.0], [1,
0.8414709848078965], [1, 0.9092974268256817], [4, 0.0], [4, 0.8414709848078965],
[4, 0.9092974268256817]]
```

```
Slicing: [1, 4]
Negative elements: 196
```

dicts are an extension of lists to pairs of **keys** and **values**. The notation for them is similar to the **json** format:

```
d = {"first": "some text", \
      "second": a, \
      "third": squares}

print(d["first"])
print(d["second"])
print(d["third"])
```

```
some text
7
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

You can print all available keys and values:

```
print(d.keys())
print(d.values())
```

```
dict_keys(['first', 'second', 'third'])
dict_values(['some text', 7, [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]])
```

1.7 Loops

There are two main ways to do loops. Important to notice here is *indentation*. Instead of brackets or other means to mark blocks of code, Python simply discerns these by indentation. Each level of indentation consists of **4 spaces**. This serves readability since you'll instantly see what block certain code belongs to.

```
a, b = 0, 1

while b < 10:
    print(b)
    a, b = b, a + b
```

```
1
1
2
3
5
8
```

While it's possible to do multiple assignments per row, this should be avoided for readability.

The second way to do loops is using the powerful “for” construct. A basic example uses the `range` generator to generate a list of integers to loop over:

```
f = range(10, 20)

print(type(f))
print(*f)
print(*range(0, 5, 2))
print(*range(7, 1, -2))

# our squares list from last lesson
squares = [x**2 for x in range(15)]

for i in range(0, 5):
    print(i, ":", squares[i])
```

```
<class 'range'>
10 11 12 13 14 15 16 17 18 19
0 2 4
7 5 3
0 : 0
1 : 1
2 : 4
3 : 9
4 : 16
```

`for` iterates over all elements of a list:

```
for element in squares:
    print(element, end=", ")
```

```
0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196,
```

1.8 Control Structures

It's often necessary to check if a certain condition has been met. This is done using `if`, `elif` and `else` structures:

```
a = 15000000

if a <= 20:
    print(a)
elif a > 5000001:
    print("a is really large")
else:
    print("a must be smaller than 21: ", a)
```

```
a is really large
```

Try to avoid loops like this:

```
b = 11

while True:
    print(b)
    b -= 1
    if b < 10:
        break
```

11
10

Why is this a bad idea?

1.9 Functions

Usually, you need a certain functionality more often than once. This functionality can then be defined as a function. Let's use the Fibonacci numbers as an example:

```
def fibonacci(n):
    """Print the Fibonacci series up to argument n."""
    a, b = 0, 1

    while a < n:
        print(a, end=' ')
        a, b = b, a + b
    print()

fibonacci(5)
fibonacci(81)
```

0 1 1 2 3
0 1 1 2 3 5 8 13 21 34 55

There's no check to see if the input `n` is smaller than zero. A better approach would be:

```
def fibonacci(n):
    """Print the Fibonacci series up to argument n."""
    if n > 0:
        a, b = 0, 1
        while a < n:
            print(a, end=' ')
            a, b = b, a + b
        print()
    else:
        print("Error: n must be larger than 0")

fibonacci(10)
fibonacci(-10)
```

```
0 1 1 2 3 5 8
Error: n must be larger than 0
```

Functions may also return some values:

```
def multiply(a, b):
    return a * b

print(multiply(5, 10))

result = multiply(7, 7)
print(result)
```

```
50
49
```

As an example, let's combine a few techniques here:

```
def fibonacciList(n):
    """Get the Fibonacci series up to argument n in a list."""
    if n > 0:
        result = []      # an empty list
        a, b = 0, 1

        while a < n:
            result.append(a)
            a, b = b, a + b

    return result
else:
    print("Error: n must be larger than 0")
    return []

print(fibonacciList(20))

for fibonacciNumber in fibonacciList(20):
    print(fibonacciNumber, end=', ')
```

```
[0, 1, 1, 2, 3, 5, 8, 13]
0, 1, 1, 2, 3, 5, 8, 13,
```

1.10 File Handling

Although most of the libraries we will use in the course have inbuilt functionality to save and load files, it is sometimes useful to being able to handle files manually.

In Python, `open()` will open a file, whether it is for reading or for writing, and calling `close()` on the variable referencing the file (this is called a *file handler*) will close it again. It's important to make sure that a file is closed after usage to avoid stale file handlers and memory hogging. `open()` has many arguments it can take, we'll just look at the most important ones here. One way to

open a file for writing is the following, where the argument "w" is provided to indicate that the file should be *overwritten*:

```
textlist = ["some", "text in form of", "a", "list"]

fh = open("test.txt", "w")

fh.write("testfile\n")
fh.write("some more text\n")
fh.writelines(textlist)

fh.close()
```

Since `test.txt` did not exist before executing the above cell, it was created. The `write` statements write the provided strings into the file, *without* a newline at the end. `writelines` write a sequence (here, a list) of strings into a file line by line. Let's see what happened by opening the file in read mode with "r":

```
filereader = open("test.txt", "r")

print(filereader.read())

filereader.close()
```

```
testfile
some more text
sometext in form ofalist
```

`read` gets every line separately. We can also use `readlines` to get a list of all the lines in the file:

```
filereader = open("test.txt", "r")

print(filereader.readlines())

filereader.close()
```

```
['testfile\n', 'some more text\n', 'sometext in form ofalist']
```

For appending to a file instead of overwriting it, the option "a" has to be given for `open()`:

```
fh = open("test.txt", "a")

fh.write("A new line\n")
fh.write("without changing the rest\n")

fh.close()
```

Let's check the result:

```
filereader = open("test.txt", "r")
print(filereader.read())
filereader.close()
```

```
testfile
some more text
sometext in form of a listA new line
without changing the rest
A new line
without changing the rest
```

Sometimes it's useful to iterate over all the lines in a file:

```
filereader = open("test.txt", "r")

for line in filereader:
    print(line, end='')

filereader.close()
```

```
testfile
some more text
sometext in form of a listA new line
without changing the rest
A new line
without changing the rest
```

Sometimes it's not clear whether a file already exists and overwriting it would yield catastrophic data loss. For creating new files without overwriting existing files, the option "x" is useful. In this case, an error is thrown since "test.txt" already exists:

```
fh = open("test.txt", "x")

fh.write("New file\n")
fh.write("but only if it didn't exist before\n")
fh.writelines(textlist)

fh.close()
```

↳

```
FileExistsError
last)
Traceback (most recent call
↳
```

```

<ipython-input-20-32c15536a01c> in <module>
----> 1 fh = open("test.txt", "x")
      2
      3 fh.write("New file\n")
      4 fh.write("but only if it didn't exist before\n")
      5 fh.writelines(textlist)

FileExistsError: [Errno 17] File exists: 'test.txt'

```

1.11 Classes

Although you don't need to use object-oriented programming in this course, it is good to have a rough understanding of what classes do and how to interact with them in order to understand what some of the modules we will use are doing in the background.

Classes encapsulate functionality. They have attributes and methods. Attributes are what the function “has” or “knows” (variables), methods are what a class can do (functions). Think for example of example of a car. A car has a number of things it “has”, like the number of seats, horsepower, ... and it also can do something, like accelerating, decelerating, use the brakes, steer, ...

Let's use a much simpler example here. A Window has a status that indicates whether it is opened or closed, and it can open, or close. As a class, this looks like the following:

```

class Window:                  # class names are Capitalized! This is convention.
    def __init__(self):
        self.open = False

    def openUp(self):
        if self.open:
            print("Window already open.")
        else:
            self.open = True

    def closeDown(self):
        if not self.open:
            print("Window already closed.")
        else:
            self.open = False

    def checkWindowState(self):
        if self.open:
            print("Window is open.")
        else:
            print("Window is closed.")

```

This is like a blueprint for an *instance* of a class, called an **object**. Objects are instantiated from classes similarly to initializing variables:

```
# here, we instantiate a Window in the variable "window"
window = Window()

print(type(window))

# here, we use the methods of the object
window.openUp()
window.openUp()
window.checkWindowStatus()
window.closeDown()
window.checkWindowStatus()
```

```
<class '__main__.Window'>
Window already open.
Window is open.
Window is closed.
```

Classes can also be instantiated with arguments:

```
class Window:
    def __init__(self, opened):      # __init__ function must be implemented
        self.open = opened

    def openUp(self):
        if self.open:
            print("Window already open.")
        else:
            self.open = True

    def closeDown(self):
        if not self.open:
            print("Window already closed.")
        else:
            self.open = False

    def checkWindowStatus(self):
        if self.open:
            print("Window is open.")
        else:
            print("Window is closed.")

window = Window(False)
window.checkWindowStatus()
print(window.open)
```

```
Window is closed.
False
```

This uses the inbuilt special function `__init__`, which is called when an instance of a class is created. This function *must always* be implemented, even if you don't initialize your instance with arguments.

You probably noticed the extensive use of `self`. This *must* be the first argument of every method you implement in a class. The reason is that a class is only a blueprint. When you later create an instance called, say, `w1`, and use a method like `w1.closeDown()` in your code, the `self` will contain your object name. This way, Python knows that when you call `w1.closeDown()`, it needs to perform all the actions on attributes produced by that method on those attributes belonging to `w1`. This does take some getting used to, so a little bit of practicing with your own ideas for classes goes a long way.

This is hopefully all you need to know to follow along with the lecture and the exercises. Of course, this is in no way a comprehensive Python course and we missed out many of the features and comfortable properties Python offers. Since people proficient in the language are currently highly sought after, it is probably wise to accustom oneself with it further. We will learn everything else we need as we move on.

2 Linear Algebra

2.1 NumPy

NumPy is probably the most widely used module for Python everywhere. It offers a lot of functionality for linear algebra, manipulating vectors, matrices, and some other features. We will take a quick look at the single most useful data structure in Python - numpy arrays - and learn everything else alongside the rest of this lecture. Since NumPy is a module, we need to import it before we can use it. The basic data structure NumPy works with is called a `numpy array`. You can use it pretty much like normal lists in pure Python, but they offer much more functionality. Here are a few things to know:

```
import numpy as np

vector = np.array([1, 3, 4])

matrix = np.array([[3, 4, 9],
                  [9, 6, 7],
                  [8, 3, 7]])

print(vector)
print()
print(matrix)
```

[1 3 4]

[[3 4 9]
 [9 6 7]
 [8 3 7]]

One thing to be super careful about is the type of numpy array that is created. In the above cases, both arrays are saved as integers:

```
print(vector.dtype)
print(matrix.dtype)
```

int64
int64

Numpy determines this from the values the array was instanciated with. To make sure that it has a certain type, you can force it with the `dtype` argument:

```
vector = np.array([1, 3, 4], dtype=np.float32)

matrix = np.array([[3.0, 4.0, 9.0],
                  [9.0, 6.0, 7.0],
                  [8.0, 3.0, 7.0]])

print(vector.dtype)
```

```
print(matrix.dtype)
```

```
float32
float64
```

NumPy arrays can be used like lists, so slicing and negative indexing work the same.

Sometimes you need to address a column of a matrix, which you can do like this:

```
print(matrix[:,1])
```

```
[4. 6. 3.]
```

2.2 Why Linear Algebra

Good knowledge of linear algebra is essential in understanding some results and techniques in machine learning settings. In most cases, the input to ML algorithms is expressed as a vector of some values. These values can have any meaning, and the vector doesn't need to have consistent units. Often, a part of the problem is how to express the data you work with as a vector, e.g. image or video data, which in general is qualitative data. We will talk about a few approaches to this problem later in the course. The objects we will be working with are mostly one of the following:

- Scalars: single numbers. Examples will be the learning rate $\alpha \in \mathbb{R}$, or the number of hyperparameters $n \in \mathbb{N}$. (*Side note: there are also pseudo-scalars*)
- Vectors: arrays of numbers with (in most cases) no meaning whatsoever. Example is describing a pendulum by its position and velocity, which fully determines its state
- Matrices: 2d arrays of numbers, also carrying (in most cases) no physical meaning. Examples are weight matrices $\mathbf{W} \in \mathbb{R}^{m \times n}$
- Higher order matrices: nd arrays of numbers. Examples are RGB images, or sensor data from multiple sensors, or video data with time as another degree of freedom

In ML, all of these are called **tensors** (reflected in the name **TensorFlow**) although mathematically, tensors are well-defined geometric quantities and (in most cases) different from what we are using.

Tasks that often arise in ML contexts include:

- Turning matrices into vectors and vice versa
- Stack physical quantities into a vector (state variables/degrees of freedom)
- Unrolling time-series data
- Map vectors of different dimension onto each other with non-square matrices

The latter is for example needed for neural networks. Basically, they learn a **mapping** from input data to output data. The neat thing here especially for engineering is that this mapping is in many cases **reversible**.

Turning matrices into vectors is necessary for example when processing **image data**. As an example, grayscale images are saved in a computer as matrices of grayscale values, with entries going from 0 to 255 or 0 to 1. To vectorize all the values necessary to describe a, say, 40x40 pixel grayscale image, you need a vector \mathbf{v} of dimension $\dim(\mathbf{v}) = 40 \cdot 40 = 1600$. The dimension of this vector grows quickly with increasing image size. RGB images need three values per pixel, so an image of 40x40 pixels in RGB would need a vector of dimension 4800. Transparency increases this further. Hence, we need algorithms that perform well in these high-dimensional settings.

Representing everything as a vector gives geometrical meaning to data. See for example Anderson's Iris dataset, consisting of lengths and widths of petals and sepals of Iris flowers (you don't need to understand the code here):

```
%matplotlib notebook
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import datasets
#from sklearn.decomposition import PCA

iris = datasets.load_iris()
X = iris.data[:, :4]
y = iris.target

x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5

fig = plt.figure(2, figsize=(8, 16))
ax1 = fig.add_subplot(211)
ax2 = fig.add_subplot(212, projection="3d")

ax1.scatter(X[:, 0], X[:, 1], c=y, cmap="viridis", edgecolor='black')

ax1.set_xlabel('Sepal length')
ax1.set_ylabel('Sepal width')

ax1.set_xlim(x_min, x_max)
ax1.set_ylim(y_min, y_max)
ax1.set_xticks(())
ax1.set_yticks(())

#X_reduced = PCA(n_components=3).fit_transform(iris.data)

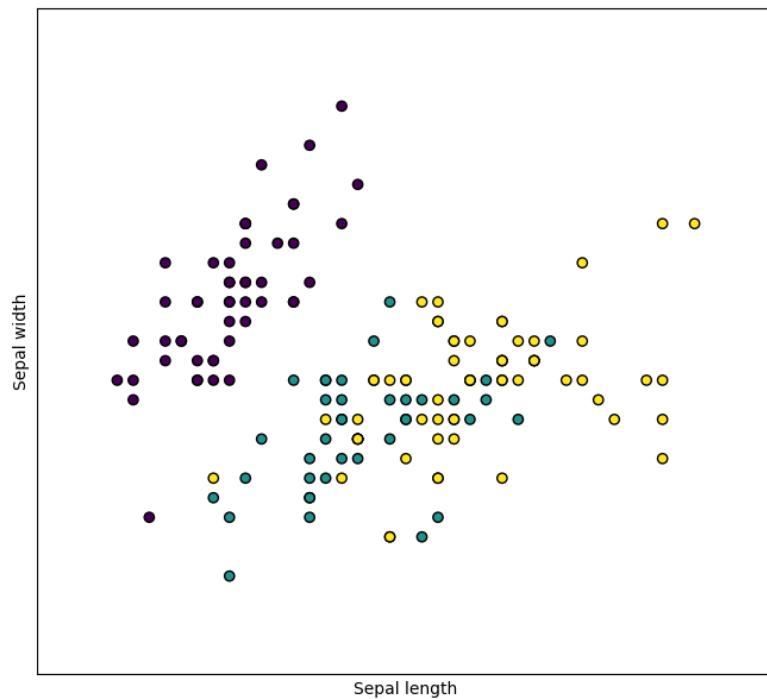
ax2.scatter(X[:, 2], X[:, 1], X[:, 0], c=y, cmap="viridis", edgecolor='black', s=40)

ax2.set_title("First three PCA directions")
ax2.set_xlabel("1st component")
ax2.w_xaxis.set_ticklabels([])
ax2.set_ylabel("2nd component")
ax2.w_yaxis.set_ticklabels([])
ax2.set_zlabel("3rd component")
ax2.w_zaxis.set_ticklabels([])

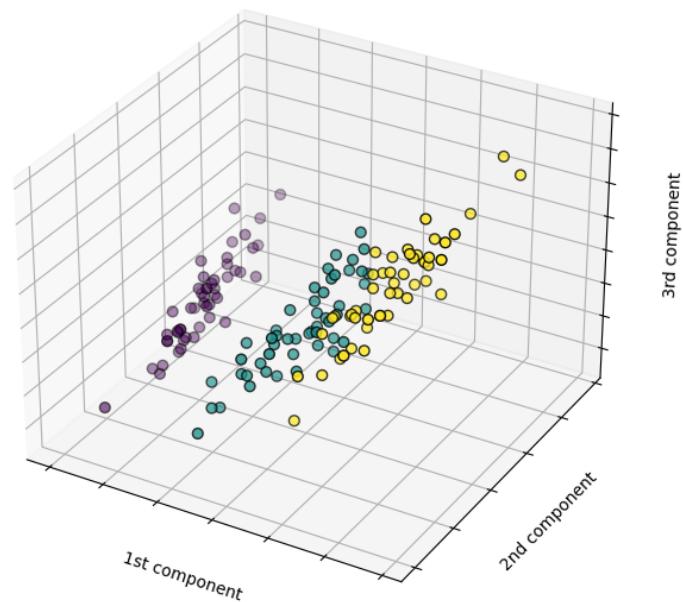
plt.savefig("img/iris_vectors.png")
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



First three PCA directions



The dataset itself is 4-dimensional, so in the above example, a reduced version of that dataset was plotted, so it could be represented in 2d and 3d (try rotating this image. It will be slow, but should be possible).

The different colors in the plot correspond to different species of Iris flower. Obviously, geometrical “closeness” of data points here suggests membership to the same species. Often, we want to find a mapping of data such that similar samples of the dataset will be close in a geometrical sense. We will investigate this closer in the lectures about model order reduction and clustering methods.

2.3 Array operations

NumPy offers several handy functions to manipulate arrays or to get some information about them. It is often necessary to get the dimensions of an array:

```
import numpy as np

vector = np.array([1, 3, 4], dtype=np.float)

matrix = numpy.array([[3.0, 4.0, 9.0],
                     [9.0, 6.0, 7.0],
                     [8.0, 3.0, 7.0]])

print(vector.shape)
print(matrix.shape)
```

```
(3,)
(3, 3)
```

Let's look at a few of the basic operations on NumPy arrays.

For adding, arrays must have the same dimensions:

```
A = np.array([[5, 3, 1], [3.0, 2, 7], [1, 3, 0]])
B = np.random.rand(3, 3)
C = np.random.rand(3, 4)

print(A + B)
print(A + C)
```

```
[[5.98790245 3.49984241 1.39192664]
 [3.11040607 2.32102182 7.495645]
 [1.37623334 3.06696395 0.62703563]]
```

```

ValueError                                Traceback (most recent call last)

->last)

<ipython-input-4-0e6835c35d69> in <module>
    4
    5 print(A + B)
----> 6 print(A + C)

ValueError: operands could not be broadcast together with shapes (3,3) <
->(3,4)

```

The error here is intentional.

Regarding addition, sometimes **broadcasting** is a handy shortcut. This looks a bit weird notation-wise, as if you would add a vector to a matrix. What actually happens is that the vector is duplicated and put into a matrix such that it can be added to **A**:

```

b = np.ones(3)

print(b)

print(A + b)
print(A + np.ones([3, 3]))

```

```
[1. 1. 1.]
[[6. 4. 2.]
 [4. 3. 8.]
 [2. 4. 1.]]
[[6. 4. 2.]
 [4. 3. 8.]
 [2. 4. 1.]]
```

There are several multiplications that can combine matrices.

- Dot/Matrix product $\mathbf{C} = \mathbf{AB} = \sum_k A_{ik}B_{kj} = C_{ij}$ or $\mathbf{v} \cdot \mathbf{w} = \sum_i v_i w_i$ or $\mathbf{v}^T \mathbf{w}$ for vectors
- Hadamard product $\mathbf{C} = \mathbf{A} \odot \mathbf{B}, A_{ij}B_{ij} = C_{ij}$, elementwise multiplication, all matrices must have same dimension
- (Frobenius) scalar product $\mathbf{A} : \mathbf{B} = \sum_{ij} A_{ij}B_{ij}$ or $\mathbf{v} : \mathbf{w} = \sum_i v_i w_i$ or $\mathbf{v}^T \mathbf{w}$ for vectors

There are different ways to perform these using NumPy:

```

v = numpy.random.rand(3)
w = numpy.random.rand(3)
A = numpy.random.rand(3, 5)
B = numpy.random.rand(5, 4)
C = numpy.random.rand(5, 4)

print("### Matrix product\n")

```

```

print(np.matmul(A, B))
print(np.dot(A, B))
print(A @ B)
print(v @ w)
print("\n\n")

print("### Hadamard product\n")
print(np.multiply(B, C))
print(np.multiply(v, w))
print("\n\n")

print("### (Frobenius) scalar product\n")
print(np.sum(np.multiply(B, C)))
print(np.trace(B.T @ C))
print(v.T @ w)
print(np.sum(np.multiply(v, w)))
print("\n\n")

```

Matrix product

```

[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
[[0.50174014 1.19098977 0.73732394 0.77027334]
 [0.58943444 2.14849181 0.95060178 0.98311974]
 [0.82874094 2.33404366 1.43521698 1.06652765]]
0.8412912756661889

```

Hadamard product

```

[[0.17681447 0.78257756 0.27551326 0.11236577]
 [0.07064154 0.16313143 0.5803325 0.02800163]
 [0.13442409 0.46555704 0.62498262 0.3243592 ]
 [0.26878177 0.17939867 0.27650369 0.73702217]
 [0.08314432 0.07775577 0.11926288 0.02877473]]
[0.58947782 0.07911717 0.17269629]

```

(Frobenius) scalar product

```
5.50934512220024
5.50934512220024
0.8412912756661889
0.8412912756661889
```

\mathbf{A}^T gives the **transpose** of \mathbf{A} , \mathbf{A}^T .

We'll also need the **inverse**, which only exists for non-singular square matrices, defined as the Matrix \mathbf{A}^{-1} that satisfies $\mathbf{A}^{-1}\mathbf{A} = \mathbf{AA}^{-1} = \mathbf{I}$:

```
A = np.random.rand(3, 3)

A_inv = np.linalg.inv(A)

Identity = np.eye(3)

print(Identity)
print(A @ A_inv)
print(A_inv @ A)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
[[ 1.0000000e+00 -2.27076691e-16  7.71880226e-18]
 [-6.24902931e-17  1.0000000e+00 -2.73750163e-16]
 [-2.88072382e-16  1.59104102e-16  1.0000000e+00]]
[[ 1.0000000e+00 -8.20568143e-16  1.08072375e-16]
 [ 4.47426168e-16  1.0000000e+00 -3.34969881e-16]
 [-1.16315276e-16 -9.81591787e-17  1.0000000e+00]]
```

Note again the numerical zeros, as mentioned in the Python introduction.

In some cases, defining the **pseudoinverse** of a matrix can be useful. The pseudoinverse \mathbf{A}^+ is defined for non-square matrices \mathbf{A} in a way that generalizes the concept of inverses of square matrices. There is no unique generalization. The most commonly used pseudoinverse is the Moore-Penrose inverse, which is defined such that it solves the least-squares problem $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{x} = \mathbf{A}^+\mathbf{b}$. You can calculate it via numpy:

```
A = np.random.rand(5, 3)

A_plus = np.linalg.pinv(A)

print(A.shape)
print(A_plus.shape)
print(A_plus @ A)
print(A @ A_plus)
```

```
(5, 3)
(3, 5)
[[ 1.0000000e+00  1.11664759e-16  1.05540083e-16]
 [-1.78234610e-16  1.0000000e+00  4.93066658e-18]
 [-8.81678620e-17  2.32567099e-16  1.00000000e+00]]
[[ 0.47608845  0.00462968  0.48928445 -0.09398551  0.03426745]
 [ 0.00462968  0.9933698   0.0119251   0.07391415 -0.03097246]
 [ 0.48928445  0.0119251   0.50298549 -0.0924462   0.04362692]
 [-0.09398551  0.07391415 -0.0924462   0.17254859  0.346313  ]
 [ 0.03426745 -0.03097246  0.04362692  0.346313   0.85500767]]
```

As you can see, only the first matrix multiplication yields an identity matrix. The pseudoinverse here is a **left inverse**. We will see this distinction again when discussing singular value decomposition.

2.4 Linear combinations

A **linear combination** \mathbf{w} of a set $V = \{\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n\}$ of n vectors \mathbf{v}_i is their weighted sum

$$\mathbf{w} = \sum_i \alpha_i \mathbf{v}_i \quad (1)$$

In FEM, the solution to a partial differential equation (PDE) for some field u (say, displacements) is calculated as the linear combination of the shape functions N^I at each node I , where the coefficients \hat{u} are the degrees of freedom calculated during the analysis:

$$u = \sum_i \hat{u}_i N_i^I \quad (2)$$

In **block matrix notation**, a linear combination of two vectors $\mathbf{w} = 3\mathbf{v}_1 + 2\mathbf{v}_2$ with $\mathbf{v}_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and $\mathbf{v}_2 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$ can be written as

$$\mathbf{w} = [\mathbf{v}_1 \ \mathbf{v}_2] \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad (3)$$

$$= \begin{bmatrix} 1 & 4 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} \quad (4)$$

So matrix multiplication can be interpreted as a linear combination of the matrix columns.

2.5 Span

The **span** of a set of vectors is the set of all the vectors that can be built by a linear combination of the given vectors. Say we have $\mathbf{e}_1 = [1; 0]$ and $\mathbf{e}_2 = [0; 1]$:

```
%matplotlib inline
```

```
import numpy as np
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(111)

#fig.patch.set_visible(False)
#ax.axis('off')

ax.set_xlim([-0.1, 2.0])
ax.set_ylim([-0.1, 2.0])

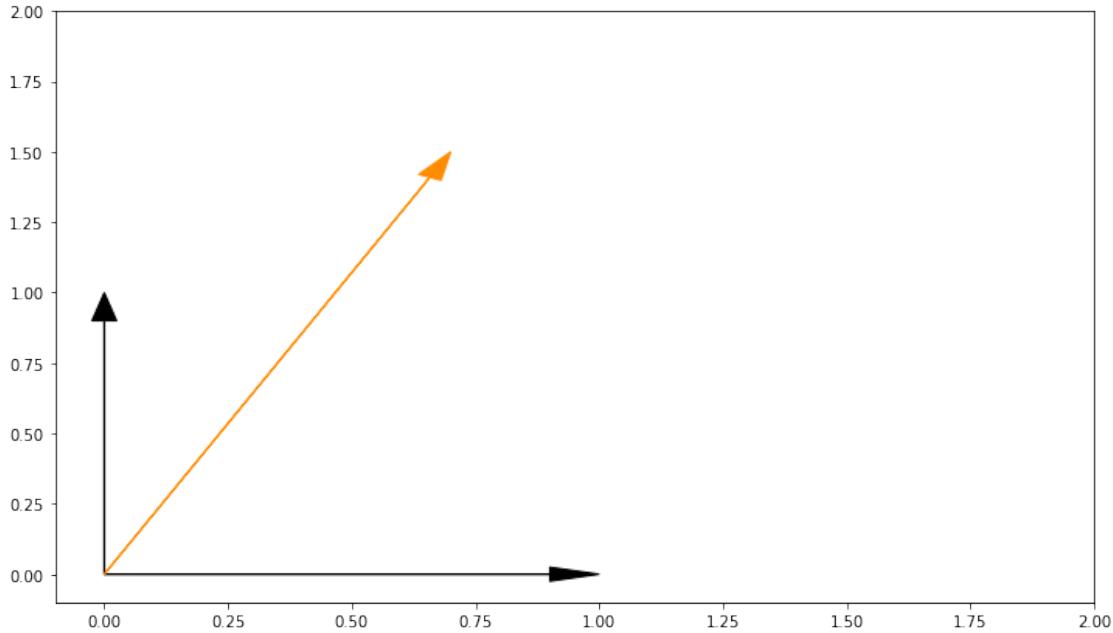
e1 = np.array([1, 0])
e2 = np.array([0, 1])
v = np.array([e1, e2]) @ np.array([0.7, 1.5])

V = np.array([e1,e2,v])
origin = np.array([0, 0])

ax.arrow(*origin, *e1, head_width=0.05, head_length=0.1, \
fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *e2, head_width=0.05, head_length=0.1, \
fc='k', ec='k', length_includes_head=True)

ax.arrow(*origin, *v, head_width=0.05, head_length=0.1, \
fc='darkorange', ec='darkorange', length_includes_head=True)

plt.show()
```



We can obviously express every 2d vector as a linear combination of \mathbf{e}_1 and \mathbf{e}_2 , so the **span** of these two vectors is the whole of \mathbb{R}^2 . This extends naturally to higher dimensions. The span of the vectors that make up the columns of a matrix is called its **column space**. $\mathbf{Ax} = \mathbf{b}$ has a solution only, if \mathbf{b} lives in the column space of \mathbf{A} . The solution \mathbf{x} then also lives in the column space of \mathbf{A} .

2.6 Linear independence

Let's look at the set of vectors V from above again. If none of the vectors in this set can be written as a linear combination of the other vectors, the set is called **linear independent**. Expressed mathematically, this means

$$\sum_i \alpha_i v_i = 0 \implies \alpha_i = 0 \quad \forall i, \tag{5}$$

so the only way this weighted sum can vanish is to make all coefficients α_i zero.

2.7 Matrix operations

We will often transform high-dimensional vectors into even higher-dimensional vectors and back, so it's imperative to think about how to get a smaller set of representative numbers that give us some information about what is going on. Examples discussed here include

norms: assign single values to matrices (see chapter about norms).

trace: the sum of the diagonal elements of a matrix $\text{tr}(\mathbf{A}) = \sum_i A_{ii}$. This extends to non-square matrices, still summing over diagonal elements, ignoring every other element. Some useful properties:

- $\text{tr}(\mathbf{A} + \mathbf{B}) = \text{tr}(\mathbf{A}) + \text{tr}(\mathbf{B})$

- $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$
- $\text{tr}(\mathbf{A}) = \text{tr}(\mathbf{A}^T)$

```
import numpy as np

A = np.array([[1, 3, 7],
              [4, 7, 11],
              [12, 6, 9]])
B = np.array([[1, 3, 7],
              [4, 7, 11],
              [12, 6, 9],
              [4, 7, 11]])
C = np.random.rand(4, 3)

print("dim(A): ", A.shape)
print("dim(B): ", B.shape)
print("dim(C): ", C.shape)
print()
print("tr(A) = ", np.trace(A))
print("tr(B) = ", np.trace(B))
print("tr(B^T) = ", np.trace(B.T))
print()
print("tr(BC^T) = ", np.trace(B @ C.T))
print("tr(CB^T) = ", np.trace(C @ B.T))
```

dim(A): (3, 3)
 dim(B): (4, 3)
 dim(C): (4, 3)

tr(A) = 17
 tr(B) = 17
 tr(B^T) = 17

tr(BC^T) = 43.63869278648197
 tr(CB^T) = 43.63869278648197

determinant: represents the volume of the parallelepiped/parallelopiped (the *nd* equivalent of a 2d rhombus/diamond shape) spanned by the column vectors of a matrix. We won't go into the details of how to calculate it here. Some useful properties are

- $\det(\mathbf{I}) = 1, \det(\mathbf{A}^T) = \det(\mathbf{A})$
- $\det(\mathbf{A}^{-1}) = \frac{1}{\det(\mathbf{A})}$
- $\det(\alpha \mathbf{A}) = \alpha^n \det(\mathbf{A})$ for $n \times n$ -matrices \mathbf{A}
- $\det(\mathbf{AB}) = \det(\mathbf{A}) \det(\mathbf{B})$ for square matrices \mathbf{A} and \mathbf{B} of equal dimension
- There's also a "pseudo - triangle inequality": $\det(\mathbf{A} + \mathbf{B}) \geq \det(\mathbf{A}) + \det(\mathbf{B})$.

Below is an example of how the determinant calculates volume (area in 2d) of a rhombus spanned by vectors. You don't need to understand the code here.

```
%matplotlib inline

import matplotlib.pyplot as plt

# plot a rhombus in 2d and calculate its volume
import matplotlib.patches as patches

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-1.0, 1.0)

# vectors spanning the rhombus
v1 = np.array([0.5, 0.5])
v2 = np.array([1.0, 0.25])
origin = np.array([0, 0])

# we need the coordinates of the endpoints
ax.add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], color="darkorange"))

# vectors spanning the rhombus
ax.arrow(*origin, *v1, head_width=0.05, head_length=0.1, \
          fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *v2, head_width=0.05, head_length=0.1, \
          fc='k', ec='k', length_includes_head=True)

# diagonals, used for geometrical calculation of the area
ax.arrow(*origin, *(v1+v2), fc='k', ec='lightgray', ls='--')
ax.arrow(*v2, *(v1-v2), fc='k', ec='lightgray', ls='--')

plt.show()

# diagonals for plotting
d1 = np.linalg.norm(v1+v2, ord=2)
d2 = np.linalg.norm(v1-v2, ord=2)

# there are two equivalent ways to calculate area geometrically
# 1. using the cross product
print("Area using the cross product: ", np.cross(v1, v2))

# 2. using  $A = |v_1| |v_2| \sin(\theta)$ 
av1 = np.linalg.norm(v1)
av2 = np.linalg.norm(v2)
= np.arccos(np.dot(v1, v2) / (av1*av2))
print("Area using the absolute form of the cross product: ", av1 * av2 * np.
      sin())

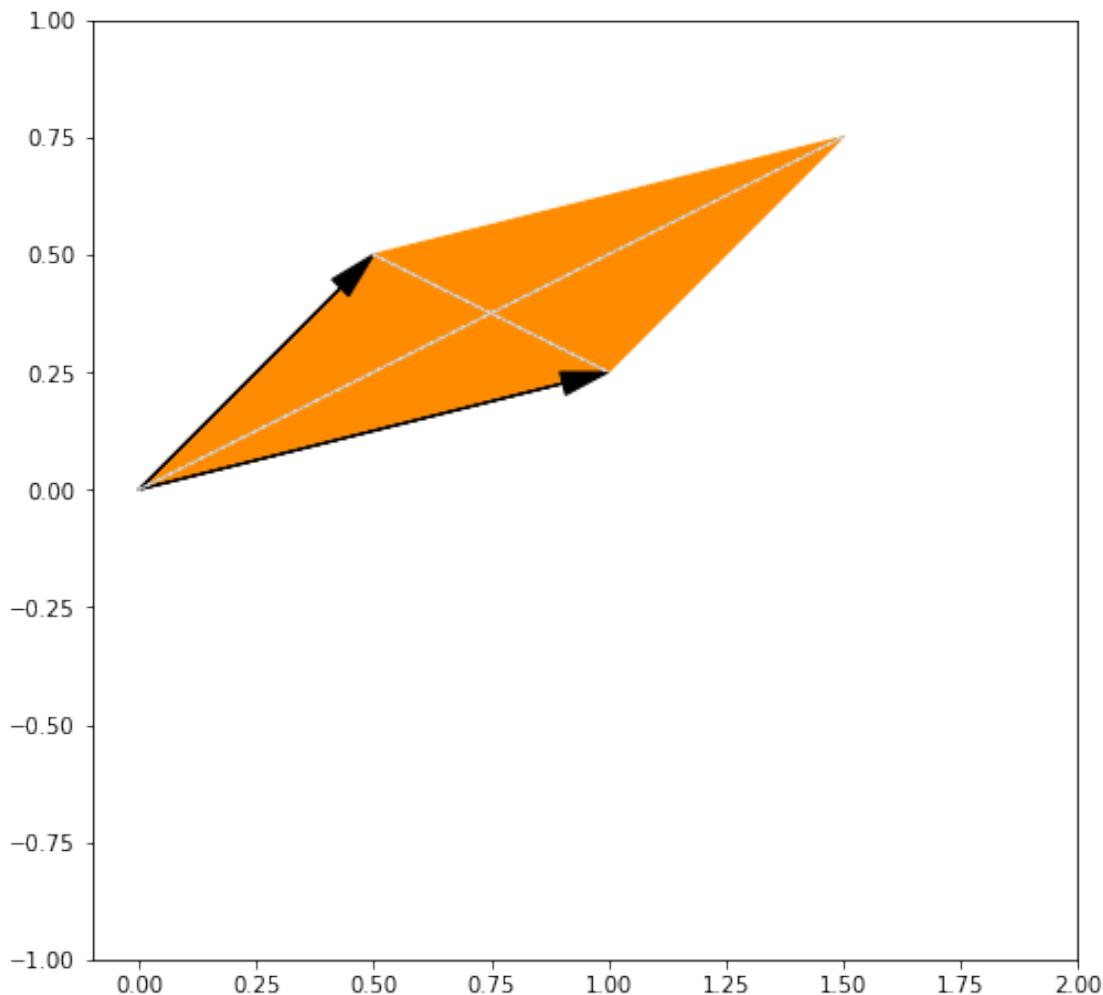
```

```
# matrix containing the vectors as columns
V = np.array([v1, v2])
print("Area using the determinant of V: ", np.linalg.det(V))

#print("Why is the determinant negative here?")

B = np.random.rand(3, 4)

#print("det(B) = ", np.linalg.det(B)) # does this work? why or why not?
```



Area using the cross product: -0.375

Area using the absolute form of the cross product: 0.37500000000000017

Area using the determinant of V: -0.375

inverse: if $\det(\mathbf{A}) \neq 0$, the inverse \mathbf{A}^{-1} of a square matrix \mathbf{A} exists such that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. A nonvanishing determinant indicates that the columns of the matrix are linear independent, or

that the column space of the $n \times n$ matrix \mathbf{A} spans the whole \mathbb{R}^n .

Intuitively, a matrix with a column space that isn't linear independent will transform a vector into a vector inside its column space. Since the dimension of this column space is lower than the dimension of the vector that is multiplied by the matrix, all components perpendicular to the column space will become 0. So, after applying the matrix, we lose all information about how the vector was embedded in the original vector space. Hence, the transformation is **not reversible**.

Let's talk about a few examples where this happens in FEM. The resulting deformations of a body, when the stiffness matrix is singular (and boundary conditions are set correctly) are called **zero modes**. A simple example of zero modes are **rigid body motions**, where your model just translates away or rotates indefinitely. How would you reverse such a motion? Since the motion is regular and unperturbed, you cannot derive the starting point or starting configuration of this kind of motion by examining the solution. This information is lost. Further examples are **locking** and **hourgassing**, which both cause internal deformations that do not cost **energy**. We will talk about zero modes/mode collapse in ML algorithms later.

```
# extremely rarely, this will produce a singular matrix
A = np.random.rand(3, 3)
A_inv = np.linalg.inv(A)

print(A @ A_inv)
print(A_inv @ A)
```

```
[[ 1.0000000e+00 -2.35699587e-17 -5.94954288e-16]
 [-2.66047701e-16  1.0000000e+00 -1.09499673e-16]
 [ 2.02447976e-16  1.42945072e-16  1.0000000e+00]]
[[ 1.0000000e+00  4.49112168e-16  6.42398285e-16]
 [-3.29809212e-16  1.0000000e+00 -9.65556270e-16]
 [-6.71502601e-16 -2.15384020e-16  1.0000000e+00]]
```

2.8 Norms

Often, we need a way to determine the “size” of some quantity. That's what norms are useful for. From a mathematical perspective, a norm must satisfy the following properties:

- $f(x) = 0 \implies x = 0$ (The only tensor of length 0 is the zero tensor)
- $f(x + y) \leq f(x) + f(y)$ (triangle inequality)
- $f(\alpha x) = |\alpha| f(x)$ (pseudo-linearity)

The most common norms all stem from the **p-norm**: $\|\mathbf{v}\|_p = (v_1^p + v_2^p + \dots + v_n^p)^{\frac{1}{p}}$. For example:

- $p = 1$: absolute sum norm
- $p = 2$: Euclidean norm
- $p \rightarrow \infty$: infinity/max norm

Let's implement the p-norm and see what it yields for different values of p :

```
import numpy as np

v = np.array([1.0, 3.0, 7.0, 14.0])
```

```

def pnorm(p, vector):
    return np.sum(vector**p)**(1/p)

for p in range(1, 10):
    print(str(p), "norm: ", pnorm(p, v))

print()
print("max: ", max(v))

```

```

1 norm: 25.0
2 norm: 15.968719422671311
3 norm: 14.604477265746313
4 norm: 14.220935686711067
5 norm: 14.087665515874415
6 norm: 14.036446523396748
7 norm: 14.015614172122747
8 norm: 14.00683203817856
9 norm: 14.003037039787143

```

```
max: 14.0
```

We see why the infinity norm is also called the max norm.

Extending these norms to matrices is straightforward by applying the definition to each matrix element, but there are more possibilities for matrix norms.

One important norm is the Frobenius norm defined as $\|\mathbf{A}\|_F = \left(\sum_{ij} A_{ij}^2\right)^{\frac{1}{2}}$. Note that this is the natural extension of the 2-norm of vectors.

Norms also give rise to a notion of distance. To quantify how close two vectors are to each other, a norm can be applied to the difference between them $\|\Delta\mathbf{v}\| = \|\mathbf{v}_2 - \mathbf{v}_1\|$. Let's say you have images of hulls of planes that you want to evaluate for crack detection. Representing the images as vectors should in the end yield the result that images of cracks are “closer” to each other than images of unharmed areas.

2.9 Quadratic forms

Quadratic forms are, in a certain sense, “weighted” versions of the notion of length we introduced with the Euclidean norm. Sometimes, some components are more “important” than others, so a scaling makes sense to give them more “weight” or “importance” relative to other components. The generalized version of the scalar product as a quadratic form is

$$\mathbf{x}^T \mathbf{A} \mathbf{x} = \sum_{ij} x_i A_{ij} x_j = x_1 A_{11} x_1 + x_1 A_{12} x_2 + \dots \quad (6)$$

Some properties follow from the numerical values of the eigenvalues of a matrix:

- **positive definite:** $\lambda_i > 0 \forall i$

- **positive semi-definite:** $\lambda_i \geq 0 \forall i$
- **negative definite:** $\lambda_i < 0 \forall i$
- **negative semi-definite:** $\lambda_i \leq 0 \forall i$

For a positive definite matrix \mathbf{A} , $\mathbf{x}^T \mathbf{A} \mathbf{x} > 0$ is true for all $\mathbf{x} \neq 0$. For a positive semi-definite matrix, the $>$ changes to \geq , so there exist non-zero vectors with zero weighted length. Think for example of the identity matrix, but set one row to all zeros. All vectors with only a component in that row will be mapped to zero. Remember here what we learned about the column space of a matrix and its span. Such a vector would live in a space exactly **perpendicular** to the column space of such a modified identity matrix. Similar corollaries follow for negative (semi-)definiteness.

This has various applications. The matrix \mathbf{A} is called a **metric** under certain conditions, and gives rise to a notion of distance on curved manifolds. This has use cases for example in relativity or curved finite elements, such as shells.

2.10 Data Visualization with Matplotlib

2.10.1 Plotting

Or rather, its `pyplot` interface. `matplotlib` is a collection of tools for 2D (and restricted 3D) visualization under Python. Its 2D capabilities are quite excessive. See for example the gallery of examples [here](#). There's also a whole lot of code available on the web if you're searching for a specific issue you have and it's likely that someone had your exact problem before.

Matplotlib has a module for visualizations in Jupyter notebooks which can be activated with the magic command `%matplotlib notebook`. We'll start with the normal mode of visualization here, which can be activated with `%matplotlib inline`:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

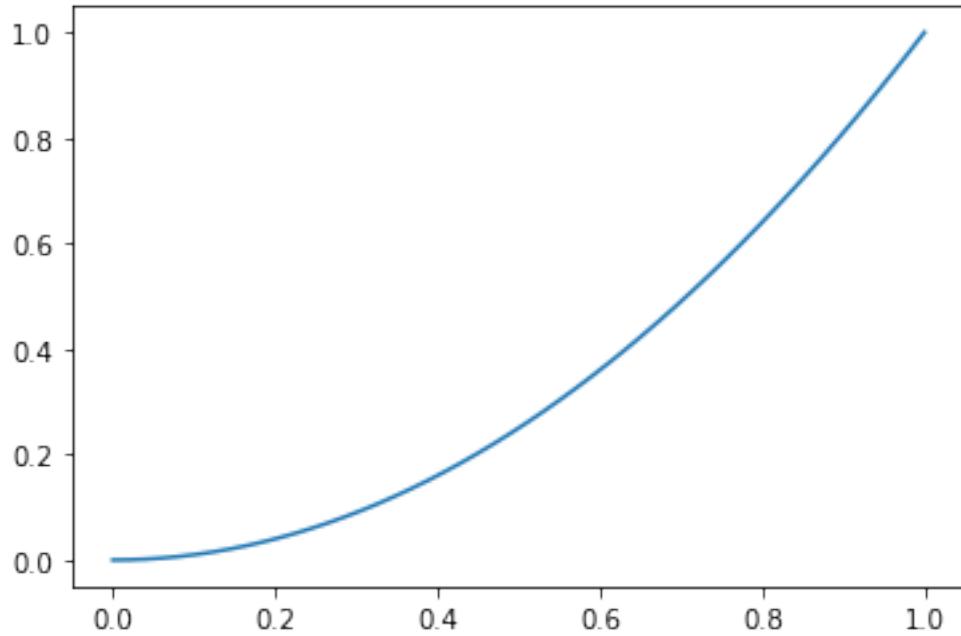
Much more often than not, functions from `numpy` are needed to generate the data necessary for plotting.

```
import numpy as np
```

Let's look at a simple example: $f(x) = x^2$. The way this is done in Python is similar to how matlab works. You need an array of plot points, which are taken as input for the function to plot, and an array of outputs, which are the y -values for the plot:

```
# create 50 points in the specified range
x = np.linspace(0, 1, 50)
# create a plot with those 50 points
plt.plot(x, x**2)

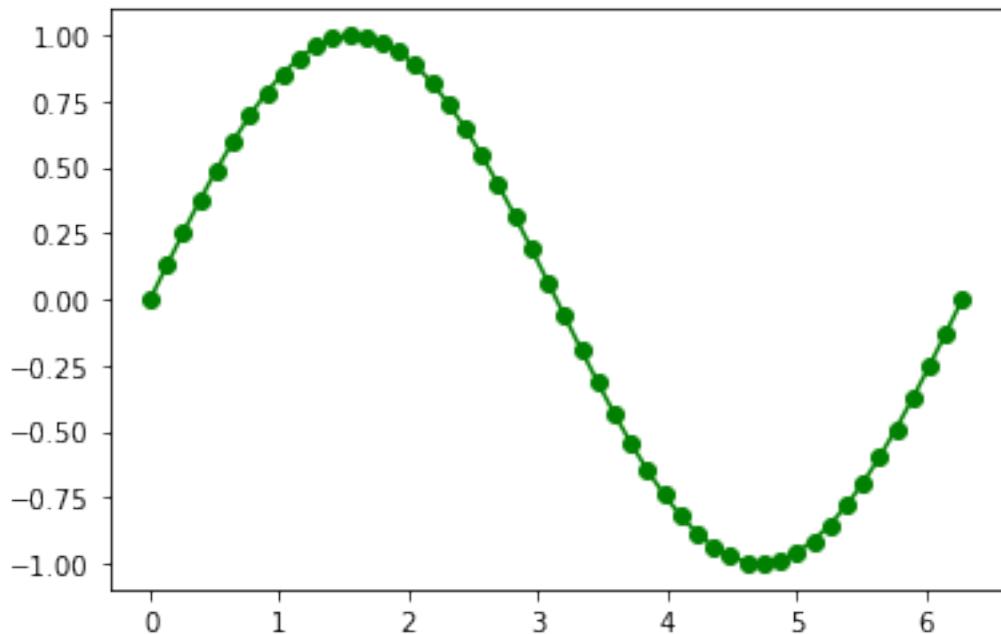
plt.show() # plt.show() can be omitted in interactive mode (%matplotlib
           ↴notebook)
```



The line style can be changed significantly by providing more parameters (a comprehensive list can be found in the [documentation](#)). A simple way to do so are *format strings*:

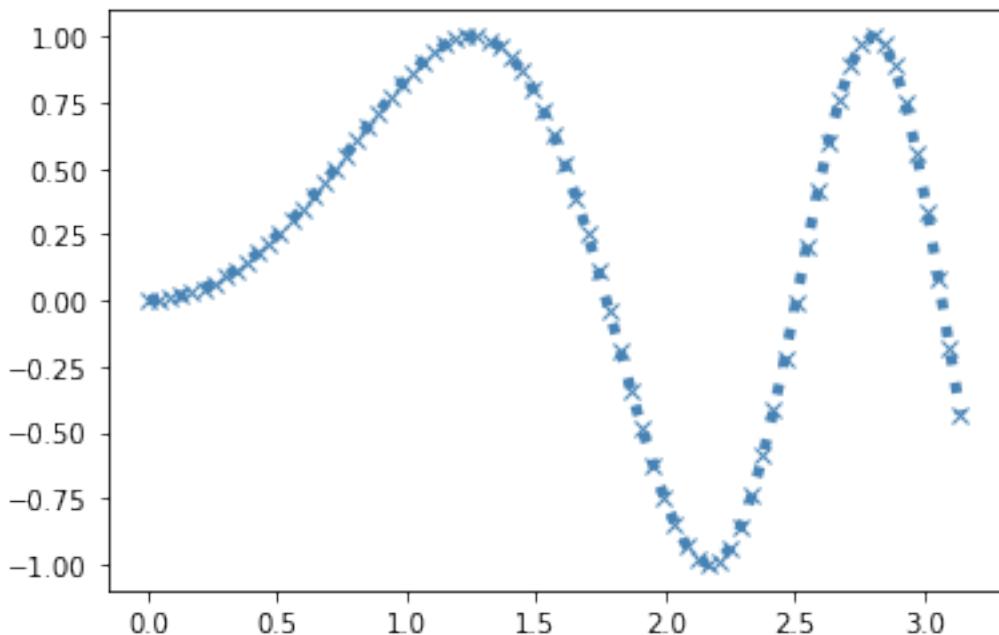
```
x = np.linspace(0, 2*np.pi, 50)
plt.plot(x, np.sin(x), 'go-')

plt.show()
```

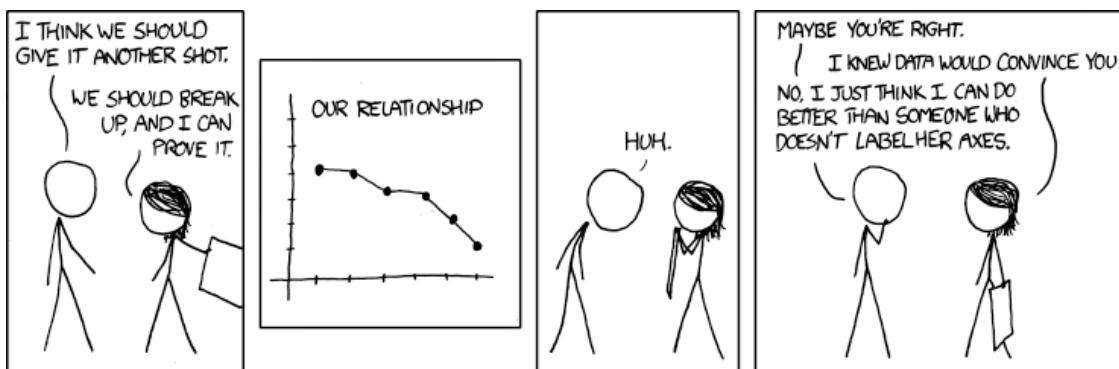


These format strings are shortcuts for other parameters which are commonly used:

```
x = np.linspace(0, np.pi, 75)
plt.plot(x, np.sin(x**2), linewidth=4, color="steelblue", marker="x", □
         ↵markersize=6, linestyle=":")
plt.show()
```



There's one important thing missing here ([obligatory XKCD](#)):



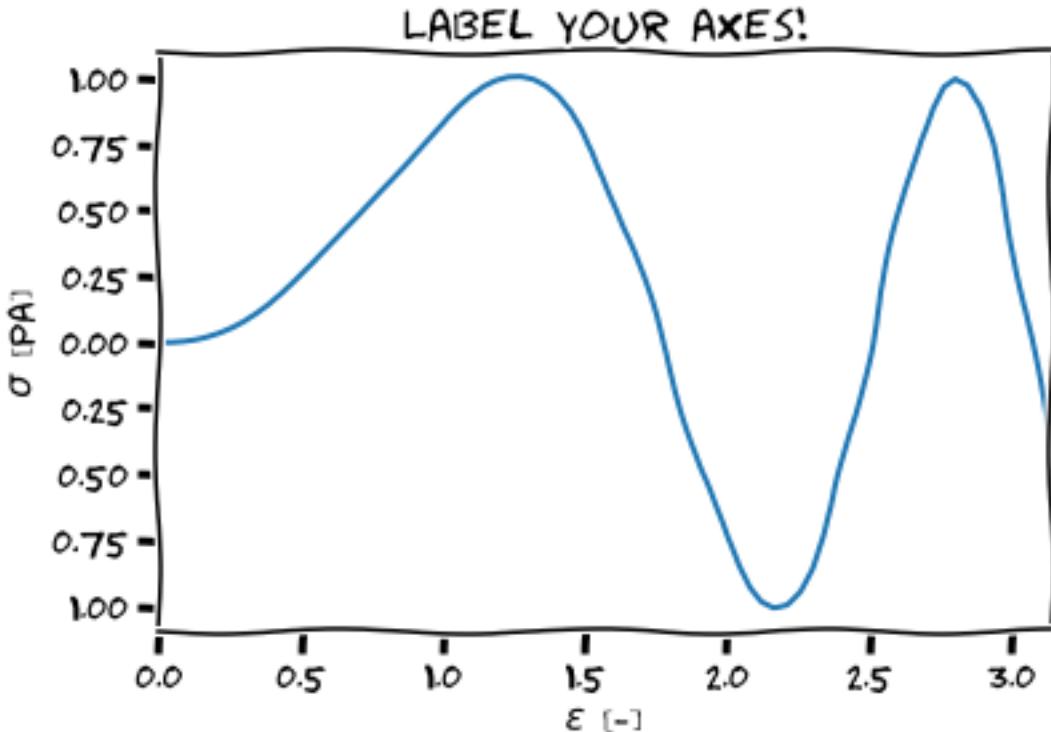
```
with plt.xkcd():
    plt.plot(x, np.sin(x**2))
```

```
# set a title for your plot
plt.title('Label your axes!')

# label axes
plt.xlabel(r'$\varepsilon$ [-]')#' [-]')
plt.ylabel(r'$\sigma$ [Pa]')#' [Pa]')

# set custom plot ranges
plt.xlim(0, np.pi)
plt.ylim(-1.1, 1.1)
```

/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:238:
RuntimeWarning: Glyph 8722 missing from current font.
font.set_text(s, 0.0, flags=flags)



Matplotlib also supports L^AT_EX rendering with the following global options set (a latex distribution must be installed on your system, e.g. MiKTeX or TeXLive):

```
from matplotlib import rc
# you can also set different fonts here:
#rc('font',**{'family':'sans-serif','sans-serif':['Helvetica']})
rc('text', usetex=True)
```

```

plt.plot(x, np.sin(x**2), linewidth=4)

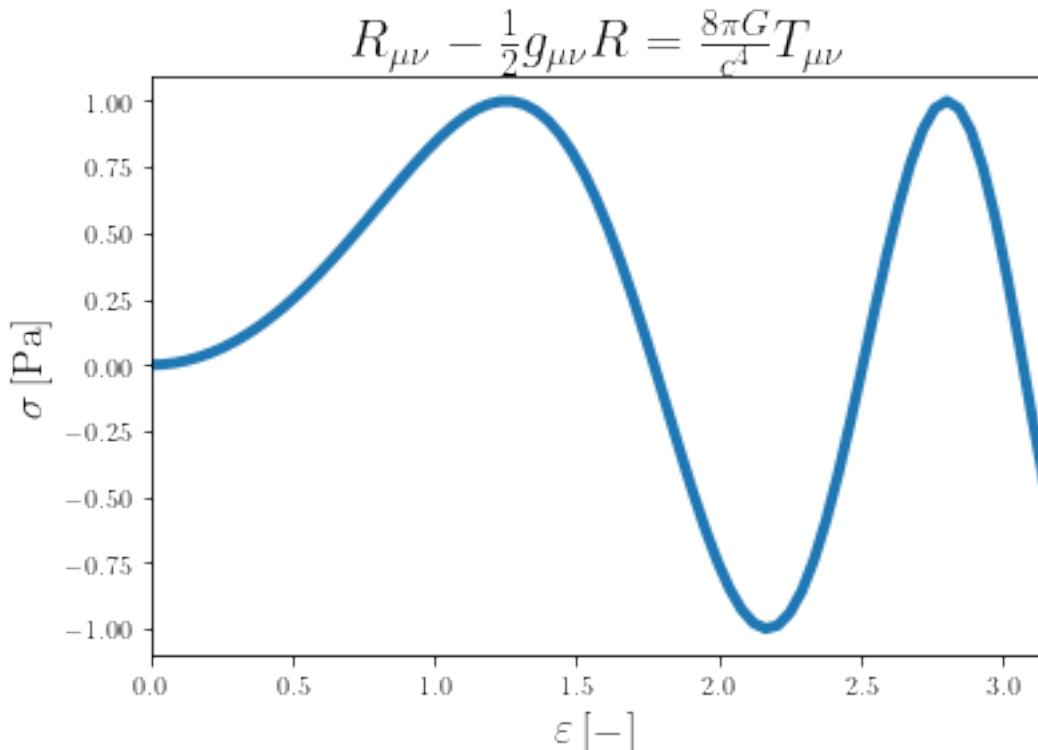
# set a title for your plot
plt.title(r'$R_{\mu\nu} - \frac{1}{2}g_{\mu\nu}R = \frac{8\pi G}{c^4}T_{\mu\nu}$', fontsize=20)

# label axes
plt.xlabel(r'$\varepsilon$ [-], [\\mathrm{-}]$', fontsize=16)
plt.ylabel(r'$\sigma$ [Pa]', [\\mathrm{Pa}]$', fontsize=16)

# set custom plot ranges
plt.xlim(0, np.pi)
plt.ylim(-1.1, 1.1)

plt.show()

```



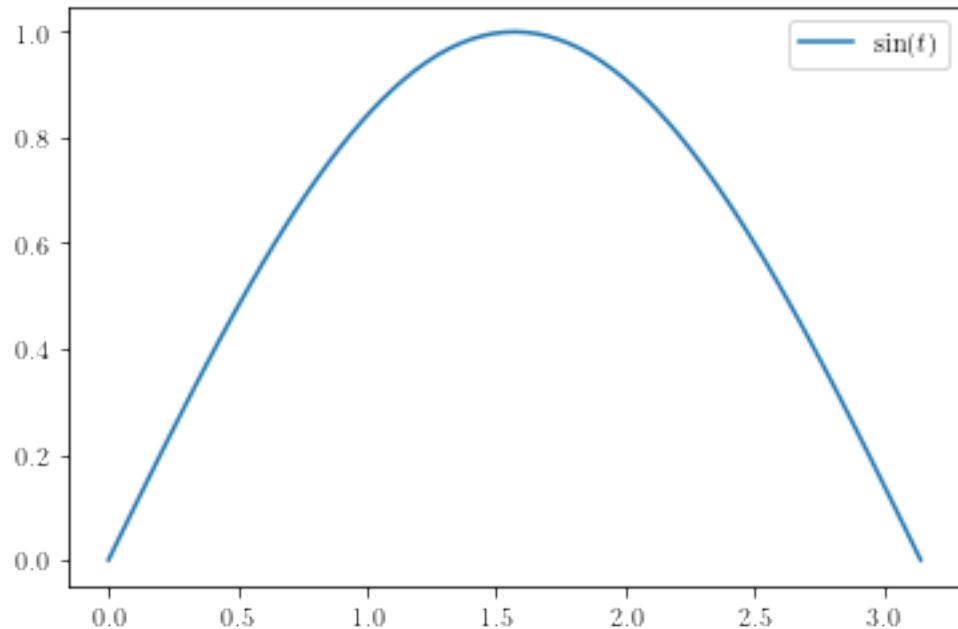
You can also place a legend in your plot:

```

plt.plot(x, np.sin(x), label=r'$\sin(t)$')
plt.legend()
#plt.legend(loc='lower left')
#plt.legend(loc='best')

```

```
plt.show()
```



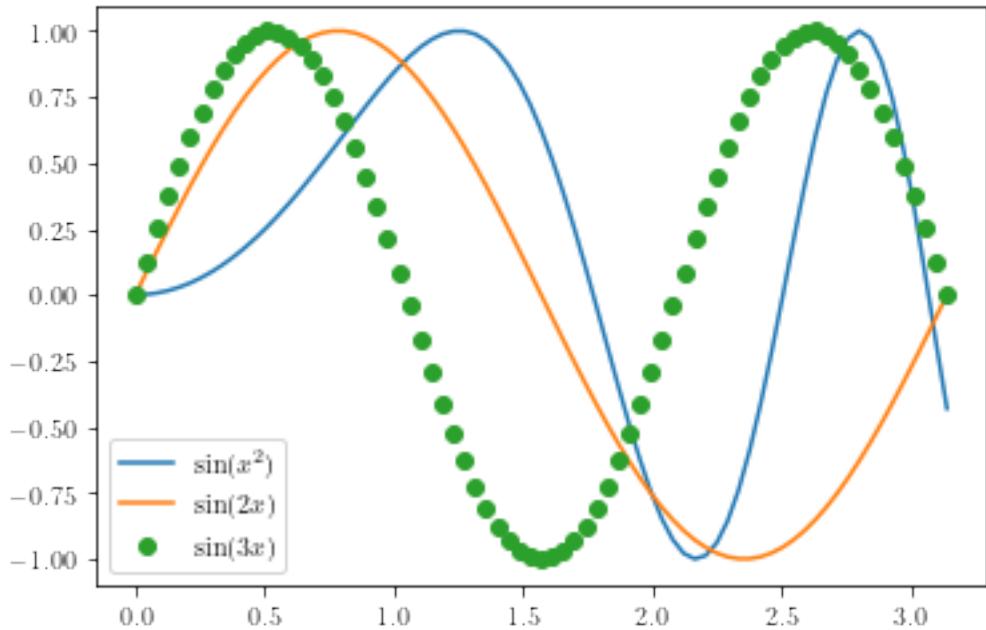
Legends usually only make sense when multiple curves are presented in a single plot:

```
rc('text', usetex=True)

plt.plot(x, np.sin(x**2), label=r'$\mathbf{\sin}(x^2)$')
plt.plot(x, np.sin(2*x), label=r'$\mathbf{\sin}(2x)$')
plt.plot(x, np.sin(3*x), 'o', label=r'$\mathbf{\sin}(3x)$')

plt.legend()

plt.show()
```



Saving a plot is easy. The file type is determined by the file suffix:

```
rc('text', usetex=False)

with plt.xkcd():
    plt.plot(x, np.sin(x**2), label=r'$\mathrm{sin}(x^2)$')
    plt.plot(x, np.sin(2*x), label=r'$\mathrm{sin}(2x)$')
    plt.plot(x, np.sin(3*x), 'o', label=r'$\mathrm{sin}(3x)$')

    # set a title for your plot
    plt.title('Label your axes!')

    # label axes
    plt.xlabel(r'$\varepsilon$ [-]')
    plt.ylabel(r'$\sigma$ [Pa]')

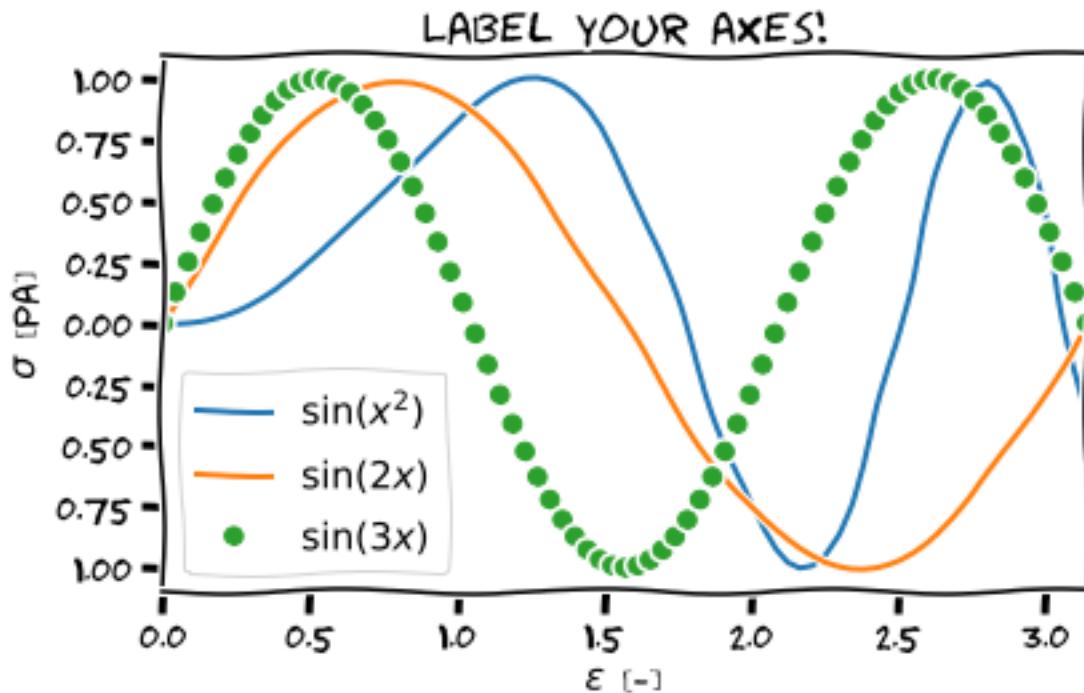
    # set custom plot ranges
    plt.xlim(0, np.pi)
    plt.ylim(-1.1, 1.1)

    # plot legend
    plt.legend()

    # sometimes the following is necessary, otherwise label might be truncated
    plt.tight_layout()
```

```
plt.savefig('weird_plot.pdf')
```

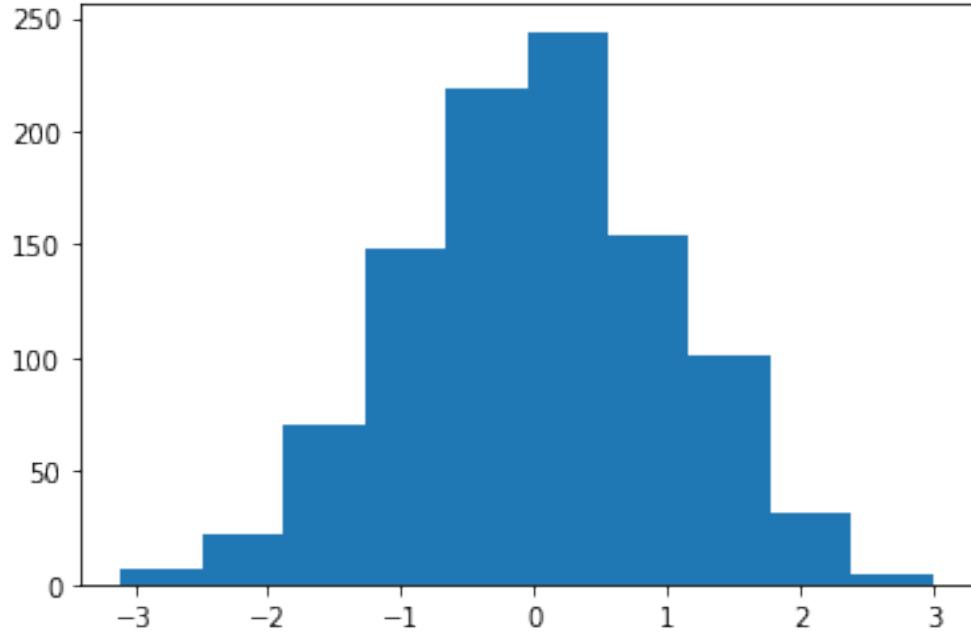
/usr/local/lib/python3.7/dist-packages/matplotlib/backends/backend_agg.py:238:
RuntimeWarning: Glyph 8722 missing from current font.
font.set_text(s, 0.0, flags=flags)



Sometimes *histograms* can yield some insight into statistical data:

```
# random data in this case
x = np.random.normal(0, 1, 1000)
plt.hist(x)

plt.show()
```



2.10.2 3D Plotting

Matplotlib's 3D capabilities are fairly restricted. It's not true 3D rendering you're seeing, but projections. This causes some objects not to be in the foreground, although they should clearly be in the foreground, among other problems. It's still quite useful in many situations. This is also a situation where it makes sense to use matplotlib's interactive mode, since it allows us to manipulate the 3D plot:

```
%matplotlib notebook

from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.gca(projection='3d')

r = np.linspace(-4 * np.pi, 4 * np.pi, 200)
z = np.linspace(-2, 2, 200)

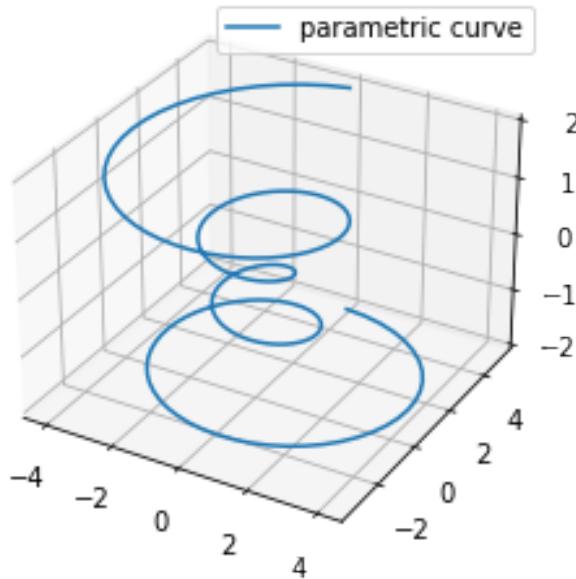
x = r * np.sin()
y = r * np.cos()

ax.plot(x, y, z, label='parametric curve')
ax.legend()

plt.savefig("img/plotting_3d.png")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```



You can find this example among other interesting things in the documentation.

2.11 Derivatives

Most optimization techniques involve taking some derivative, so we'll shortly review different derivatives and their implications and interpretations in this section.

Often, a derivative is described as measuring how much something changes when there is a small change in another quantity. See for example this image:

```
%matplotlib inline
import sympy as sy
import numpy as np
from sympy.functions import sin,cos
from math import factorial
import matplotlib.pyplot as plt

# outsource plotting arrows in a function for efficiency
def plot_arrow(origin, slope):
```

```

# need to cast to float, since sympy outputs a special type
slope = float(slope)
= np.arctan(slope)
vec = [abs(slope)*np.cos(), abs(slope)*np.sin()]
plt.arrow(*origin, *vec, head_width=0.2, head_length=0.2, \
          fc='red', ec='red', lw=3, length_includes_head=True, \
          zorder=10, label="Velocity vector")

# sympy needs all symbols that will be used declared
x = sy.Symbol('x')
# sympy functions must consist of sympy-compatible expressions
f = sin(x)*x
# this gets the analytical first derivative of a sympy-function
f_prime = f.diff(x,1)

# set up the sampling space for the curves
x_lims = [0, 8]
x_range = np.linspace(x_lims[0], x_lims[1], 100)

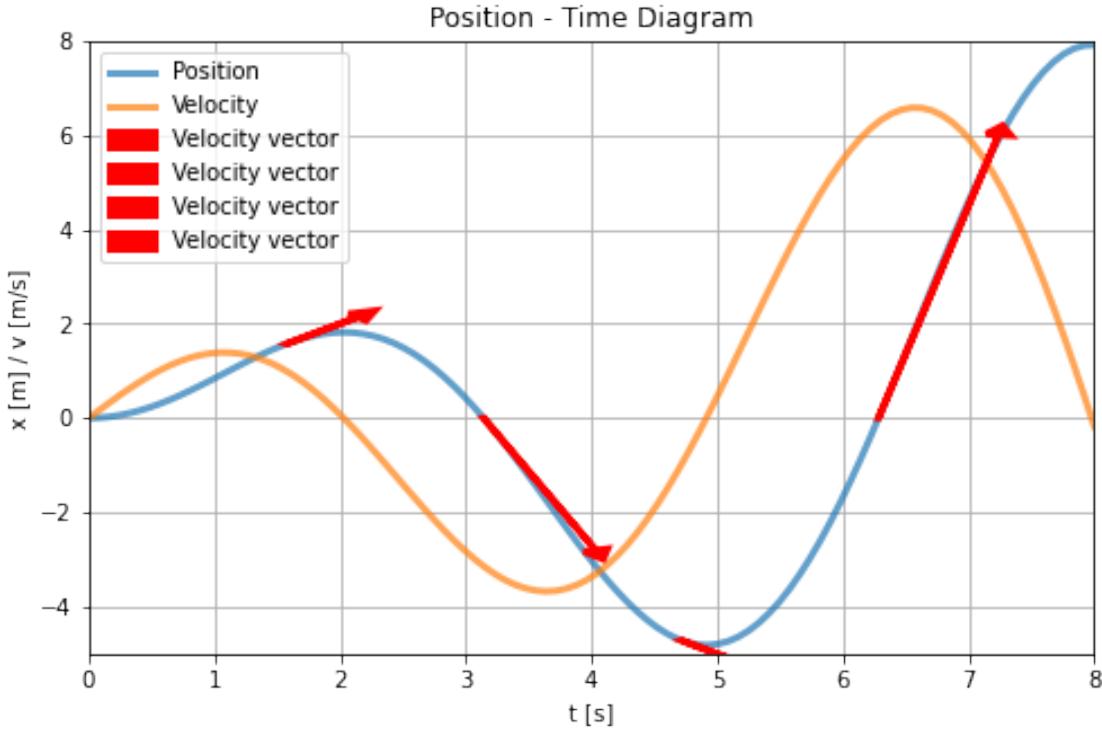
# values for the curves
y1 = np.array([f.subs(x,x_val) for x_val in x_range])
y2 = np.array([f_prime.subs(x,x_val) for x_val in x_range])

# plot the curves
plt.figure(figsize=(8,5))
plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)
plt.plot(x_range,y2,label='Velocity', lw=3, alpha=0.7)

# plot a velocity arrow at multiples of /2
for i in range(1, 5):
    plot_arrow([i*np.pi/2,f.subs(x,i*np.pi/2).evalf()], f_prime.subs(x,i*np.pi/ \
        ↵2).evalf())

# some plot options
plt.xlim(x_lims)
plt.ylim([-5,8])
plt.xlabel('t [s]')
plt.ylabel('x [m] / v [m/s]')
plt.legend()
plt.grid(True)
plt.title('Position - Time Diagram')
plt.show()

```



A more general definition would be that a derivative (of first order) is a **linear approximation** of a function at a specific point. This geometrical interpretation is often lost since for many functions, the derivative is only calculated as a number. See for example the following graph, where at $x = 3$ the derivative is plotted as a **tangent** to the curve. Use the slider to zoom closer to the point, where the tangent is attached.

```
%matplotlib inline
from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets

def plot_tangent(origin, slope, length):
    slope = float(slope)
    = np.arctan(slope)
    #vec = [abs(slope)*np.cos(), abs(slope)*np.sin()]
    vec = [length*np.cos(), length*np.sin()]
    origin = [origin[i] - vec[i] for i in range(len(origin))]
    vec = [2*i for i in vec]
    plt.arrow(*origin, *vec, head_width=0, head_length=0, \
              fc='red', ec='red', lw=3, length_includes_head=True, \
              zorder=10, label="Velocity vector")

def plot_zoomable_function(factor, x_val):
    x = sy.Symbol('x')
```

```

f = sin(x)*x
f_prime = f.diff(x,1)

x_lims = [max(0,(1-factor)*x_val), min((1+factor)*x_val,8)]
x_range = np.linspace(x_lims[0], x_lims[1], 50)
y1 = [f.subs(x,x_v) for x_v in x_range]

plt.figure()
plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)

plot_tangent([x_val,f.subs(x,x_val).evalf()], f_prime.subs(x,x_val).
evalf(), 0.5*(x_lims[1] - x_lims[0]))

plt.xlim(x_lims)
plt.ylim([-5,8])
plt.xlabel('t [s]')
plt.ylabel('x [m]')
plt.legend()
plt.grid(True)
plt.title('Position - Time Diagram')
plt.savefig("img/plot_zoomable_function.png")

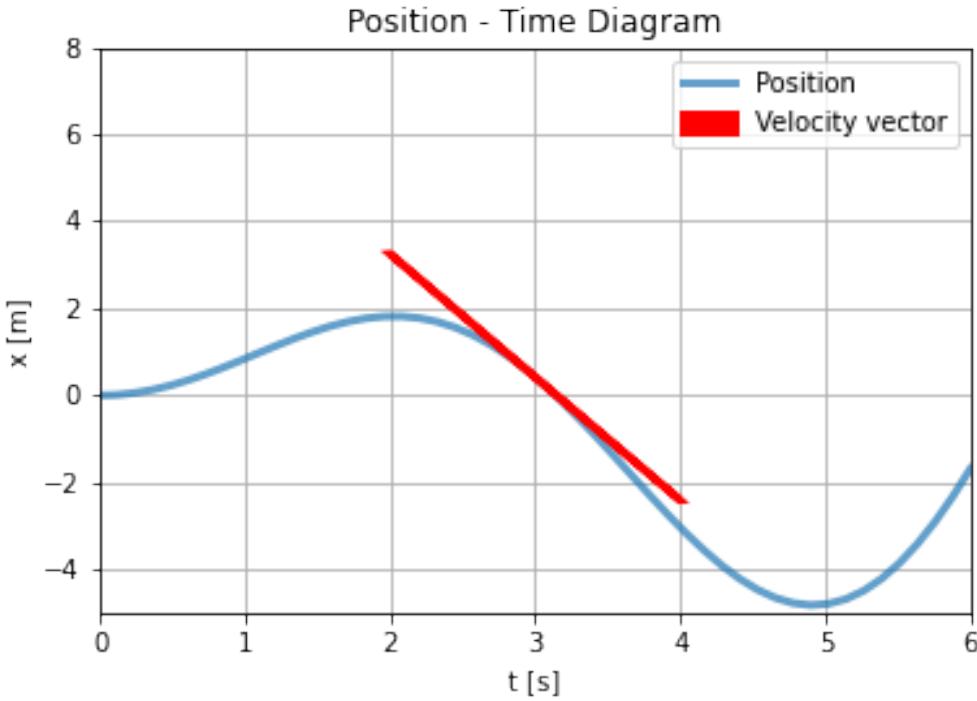
interact(plot_zoomable_function, \
    factor=widgets.FloatSlider(min=0.1, max=2, value=1, \
    continuous_update=False), \
    x_val=widgets.FloatSlider(min=2, max=7, value=3, \
    continuous_update=False));

```

```

interactive(children=(FloatSlider(value=1.0, continuous_update=False, description='factor', max

```



When zooming in you will see that the curve resembles that tangent line ever more closely. This is what is meant with “the (first-order) derivative approximates functions linearly”. This is where the notion of **manifold** stems from. Manifolds are in general *curved* n -dimensional point sets that resemble flat \mathbb{R}^n when you zoom in close enough. Our earth for example, when viewed from space, is roughly a ball (oblate spheroid, actually looks more like a potato). Yet for us small beings viewing earth from the surface, it looks rather flat.

The derivative of a function $f(x)$ is defined as

$$f'(x = p) = \frac{\partial f}{\partial x}(x = p) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}(x = p) , \quad (7)$$

where $\frac{\partial}{\partial x}$ denotes the **partial derivative** with respect to x . For multivariate functions $f(\mathbf{x})$ in \mathbb{R}^n , there are n partial derivatives $\frac{\partial}{\partial x_i}$, each for one of the n coordinates x_i of f .

$$\frac{\partial f}{\partial x_i}(\mathbf{x} = \mathbf{p}) = \lim_{h \rightarrow 0} \frac{f(x_1, x_2, \dots, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, x_2, \dots, x_n)}{h}(\mathbf{x} = \mathbf{p}) , \quad (8)$$

Using the language introduced above, the partial derivatives of a function *span* the **tangential space** at that point, or, in other words, they are the **basis vectors** of that tangential space. Any derivative is a *linear combination* of these basis vectors. In 3d, this could look like the following graph (The surface is plotted with transparency, because matplotlib's 3d capabilities are quite restricted):

```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')

# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")

# create all function points at once
z = func(x_m, y_m)

# this will plot the surface of the action on the vectors
ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)

# plot partial derivative in x-direction
ax.quiver(
    0, 0, 0, \
    float(f_x.subs(x, 1).evalf()), 0, 0, \
    color = 'red', alpha = .8, lw = 3, zorder=1000
)

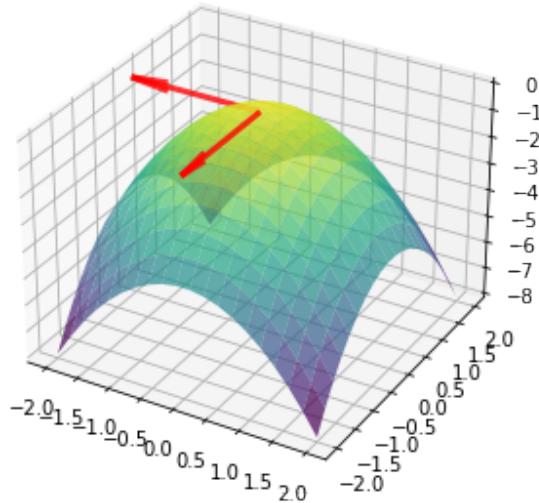
# plot partial derivative in y-direction
ax.quiver(
    0, 0, 0, \
    0, float(f_y.subs(y, 1).evalf()), 0, \
    color = 'red', alpha = .8, lw = 3, zorder=1000
)

# you might need to adjust the axes limits
#ax.set_xlim(-2, 2)
#ax.set_ylim(-2, 2)
#ax.set_zlim( 0, 8)
```

```
plt.savefig("img/tangent_space.png")
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



2.12 Gradient

The gradient of a function is the multivariate generalization of the partial derivatives.

$$\text{grad}_{\mathbf{x}} f(\mathbf{x}) = \nabla_{\mathbf{x}} f(\mathbf{x}) = \frac{\partial f}{\partial x_1} \hat{e}_1 + \frac{\partial f}{\partial x_2} \hat{e}_2 + \cdots = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \end{bmatrix} \quad (9)$$

containing the partial derivatives in the rows. Often it's abbreviated to simply ∇f , where $\nabla = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots \right]^T$ is called "nabla".

```
%matplotlib inline
import sympy as sy
import numpy as np
from sympy.functions import sin,cos
```

```

from math import factorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2 # try different factors, especially a + instead of -
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)
n = np.zeros(x_m.shape)
o = np.ones(x_m.shape)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")
func_x = sy.lambdify([x, y], f_x, "numpy")
func_y = sy.lambdify([x, y], f_y, "numpy")

z = func(x_m, y_m)

def plot_surface(surface, contours, cgrad, gradient):
    # first, a "figure" object needs to be created
    fig = plt.figure(figsize=(8,5))
    ax = fig.add_subplot(111, projection='3d')

    #ax.cla()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_zlim(np.amin(z), np.amax(z))
    if surface:
        ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)
    if contours:
        ax.contour(x_m, y_m, z, zdir='z', offset=0, cmap='viridis')
    if cgrad:
        ax.quiver(x_m, y_m, n, func_x(x_m, y_m), func_y(x_m, y_m), n, \
                  length=0.05, normalize=False, alpha=0.7)
    if gradient:
        ax.quiver(x_m, y_m, func(x_m, y_m),

```

```

func_x(x_m, y_m), func_y(x_m, y_m), n, \
length=0.05, normalize=False, alpha=0.7)

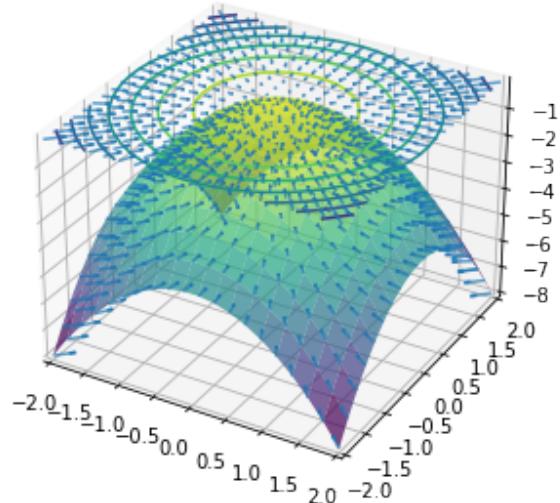
plt.savefig("img/plot_surface.png")

interact(plot_surface, surface=True, contours=True, cgrad=True, gradient=True)

interactive(children=(Checkbox(value=True, description='surface'), Checkbox(value=True, description='contours'), Checkbox(value=True, description='cgrad'), Checkbox(value=True, description='gradient')))

<function __main__.plot_surface(surface, contours, cgrad, gradient)>

```



It's often useful to plot **contour lines** of some function, which are curves of constant value, projected to a plane. The gradient is always perpendicular to the contour lines. This makes intuitive sense, after accepting the fact that the gradient always points in the direction of *steepest change*. We can easily see this after introducing the **directional derivative**

$$\text{grad}_{\mathbf{v}} f(\mathbf{x}) = \nabla_{\mathbf{v}} f(\mathbf{x}) = \mathbf{v} \cdot \nabla f(\mathbf{x}) \quad (10)$$

Since this is a scalar product, this can be written as

$$\nabla_{\mathbf{v}} f = |\mathbf{v}| |\nabla f| \cos(\theta) \quad (11)$$

This product becomes maximal, when \mathbf{v} and ∇f point in the same direction (are *colinear*), such that $\theta = 0$. Hence, the gradient always shows in the direction of *steepest ascent*. For a minimum, the vectors would have to point in opposite directions, such that $\theta = \pi$.

Let's look at an example:

```
# quadratic int grid, to make things easier
grid_size = 10
X = np.arange(0, grid_size, 1)
Y = np.arange(0, grid_size, 1)
X, Y = np.meshgrid(X, Y)

# list of x-coordinates and y-coordinates in separate lists
carriers = np.array([[2, 7, 4], [2, 7, 6]])
charges = np.array([-1, 1, -1])

def plot_elec_field(n_carriers):
    plt.figure(figsize=(8, 8))
    #plt.cla()
    Ex = np.zeros((grid_size, grid_size))
    Ey = np.zeros((grid_size, grid_size))

    for carrier in range(n_carriers):
        for i in range(grid_size):
            for j in range(grid_size):
                Ex[i, j] += charges[carrier] * (j - carriers[1][carrier]) / \
                            ((i - carriers[0][carrier])**2 + (j - carriers[1][carrier])**2)**1.5
                Ey[i, j] += charges[carrier] * (i - carriers[0][carrier]) / \
                            ((i - carriers[0][carrier])**2 + (j - carriers[1][carrier])**2)**1.5

    E = np.hypot(Ex, Ey)
    Ex /= E
    Ey /= E

    # plot charges
    plt.plot(*carriers[:, :n_carriers], 'bo', markersize=10)

    # plot field
    plt.quiver(X, Y, Ex, Ey, E, pivot='mid')

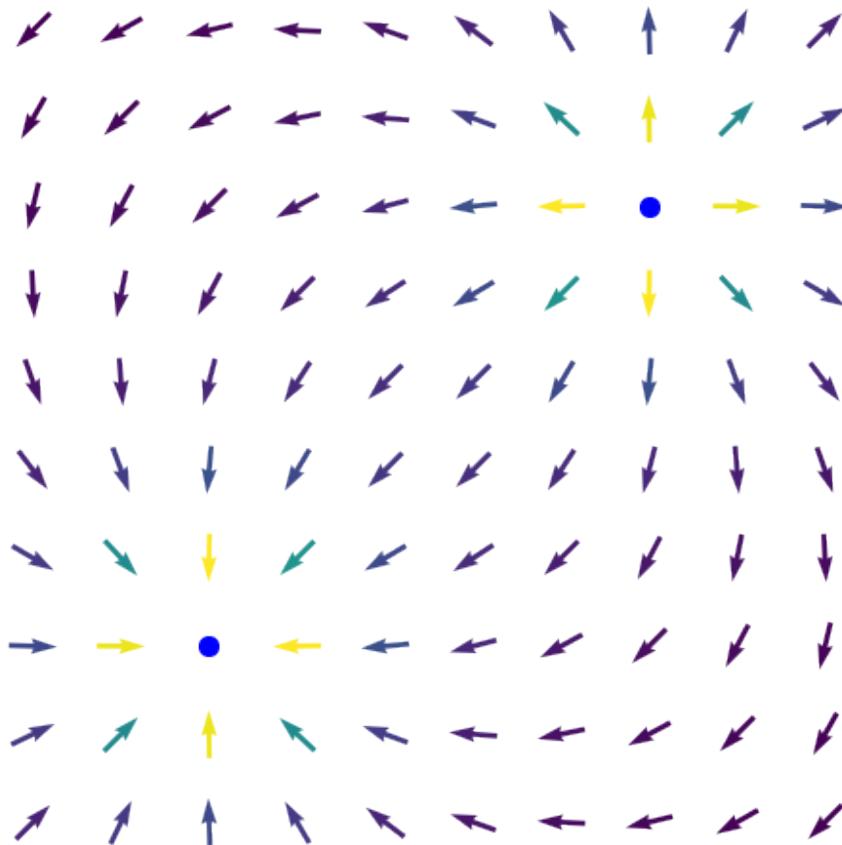
    plt.axis('equal')
    plt.axis('off')

    plt.savefig("img/plot_elec_field.png")
```

```
interact(plot_elec_field, n_carriers=(1, 3))
```

```
interactive(children=(IntSlider(value=2, description='n_carriers', max=3, min=1), Output()), _
```

```
<function __main__.plot_elec_field(n_carriers)>
```



For vector functions $\mathbf{f}(\mathbf{x})$, the gradient $\nabla \mathbf{f}(\mathbf{x})$ is often called the **Jacobian matrix**:

$$J_{\mathbf{f}}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \quad (12)$$

which is an $n \times m$ matrix. The determinant of this matrix is called the **Jacobian determinant** and both are used in transforming between coordinate system, e.g. from euclidean coordinate systems to polar coordinate system.

2.13 Hessian

The Hessian of a function f if the gradient of the gradient, measuring how much the gradient vector changes with changing spacial position. The Hessian is symmetric (see Schwarz's theorem, compare to **stiffness matrix**), so it has real *eigenvalues* and *eigenvectors*. For scalar multivariate $f(\mathbf{x})$, the Hessian is defined as

$$H_f(\mathbf{x}) = \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix} \quad (13)$$

which is an $n \times n$ -matrix. Each *column* contains the *gradient* of one component of the *gradient* of the original function, so each column tells us by how much that component of the gradient varies.

As in the 1d case, the Hessian as a second order derivative quantifies **curvature** of $f(\mathbf{x})$, with the only intricacy that curvature now depends on direction. A deep study of curvature is done in *differential geometry* and leads to several insights, that we cannot discuss in this short course.

Consider a function of the form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{b} + c$. The gradient of this function is $\nabla f(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{b}$. Its Hessian then is $H_f(\mathbf{x}) = \mathbf{A}$. Recall what we've learned about *quadratic forms*. In the following code, *eigenvalues* and *eigenvectors* are set and a corresponding quadratic form built up using a corollary of the **spectral theorem**: $\mathbf{A} = \sum_{i=1}^2 \lambda_i \mathbf{v}_i \otimes \mathbf{v}_i$, where λ_i are the eigenvalues, \mathbf{v}_i are the eigenvectors of \mathbf{A} and \otimes denotes the dyadic product.

```
mesh_points = np.linspace(-2,2,20)

x, y = np.meshgrid(mesh_points, mesh_points)

vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T

def plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2):
    fig = plt.figure(figsize=(8,5))
    ax = fig.add_subplot(111, projection='3d')

    #ax.cla()
    = np.array([eigval1, eigval2])
    v = np.array([[1, 0], [0, 1]])

    A = np.array(sum([ [i] * np.outer(v[i], v[i]) for i in range(.shape[0])]))
    q = np.array([vec.T @ A @ vec for vec in vecs])

    ax.set_xlim(np.amin(x), np.amax(x)+2)
```

```

ax.set_ylim(np.amin(y), np.amax(y)+2)
ax.set_zlim(np.amin(q)-2, np.amax(q))

q = q.reshape(20,20)

if plot_surface:
    ax.plot_surface(x, y, q, cmap='viridis', alpha=0.8)

if plot_projs:
    ax.contourf(x, y, q, zdir='x', offset=np.amax(x)+2, cmap='viridis')
    ax.contourf(x, y, q, zdir='y', offset=np.amax(y)+2, cmap='viridis')
    ax.contourf(x, y, q, zdir='z', offset=np.amin(q)-2, cmap='viridis')

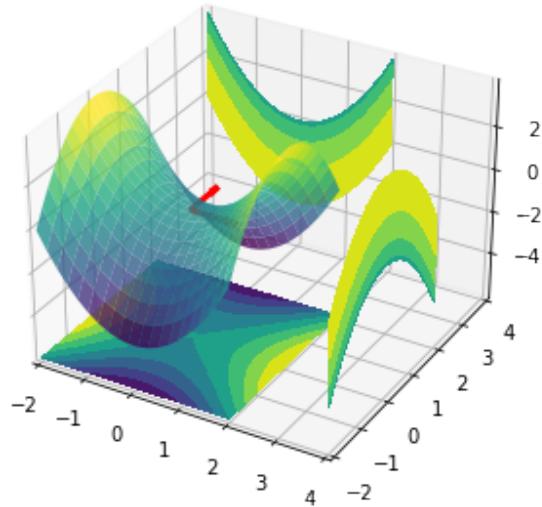
if plot_eigv:
    for vi in v:
        ax.quiver(0, 0, 0, *vi, 0, color = 'red', lw = 3, zorder=1000)

plt.savefig("img/plot_hessian.png")

interact(plot_hessian, \
         plot_surface=True, \
         plot_projs=True, \
         plot_eigv=True, \
         eigval1=widgets.IntSlider(min=-2, max=2, value=1, \
                                   continuous_update=False), \
         eigval2=widgets.IntSlider(min=-2, max=2, value=-1, \
                                   continuous_update=False))

interactive(children=(Checkbox(value=True, description='plot_surface'), Checkbox(value=True, de
<function __main__.plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2)>

```



2.14 Taylor Series

Recall that derivatives approximate functions at a specific point. The first derivative gives a linear approximation, the second derivative a quadratic approximation and equivalently for higher orders. Summing up these derivatives of a function $f(\mathbf{x})$ in a specific way gives the **Taylor series** of that function at a specific point \mathbf{a} :

$$T_f^n(\mathbf{a}) = \sum_{i=1}^n \frac{f^{(n)}(\mathbf{a})}{n!} (\mathbf{x} - \mathbf{a})^n = f(\mathbf{a}) + (\mathbf{x} - \mathbf{a})^T \nabla f + \frac{1}{2} (\mathbf{x} - \mathbf{a})^T H_f(\mathbf{a}) (\mathbf{x} - \mathbf{a}) + \dots \quad (14)$$

where $f^{(n)}(\mathbf{a})$ is the n -th derivative of f , evaluated at $\mathbf{x} = \mathbf{a}$. For “sufficiently nice” functions, this series converges to the function f itself in the limit $n \rightarrow \infty$.

The Taylor series approximates a function with **polynomials**. See a few examples below:

```
%matplotlib inline
from sympy.functions import sin, cos
from math import factorial

x = sy.Symbol('x')

f = x*sin(x)
func = sy.lambdify(x, f, "numpy")
```

```

# Taylor expansion at x0
def taylor(function,x0,n):
    p = 0
    for i in range(n):
        p += function.diff(x, i).subs(x, x0) / factorial(i) * (x - x0)**(i)

    return p

def plot_taylor(x0, order):
    plt.figure()
    #plt.cla()

    x_lims = [x0 - 5, x0 + 5]

    x1 = np.linspace(x_lims[0], x_lims[1], 100)
    y1 = []

    plt.plot(x1,func(x1),label='sin(x)', lw=3)
    plt.scatter(x0,float(f.subs(x,x0).evalf()), lw=8, color="darkorange",marker="+")

    for i in range(1,order+2):
        f_curr = taylor(f,x0,i)
        print('Taylor expansion at x0, order ' + str(i-1) + ":", f_curr)
        for k in x1:
            y1.append(f_curr.subs(x,k))
        plt.plot(x1,y1,label='order '+str(i-1), linestyle="--")
        y1 = []

    plt.xlim(x_lims)
    plt.ylim([-5,8])
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.title('Taylor series approximation')

    plt.savefig("img/plot_taylor.png")

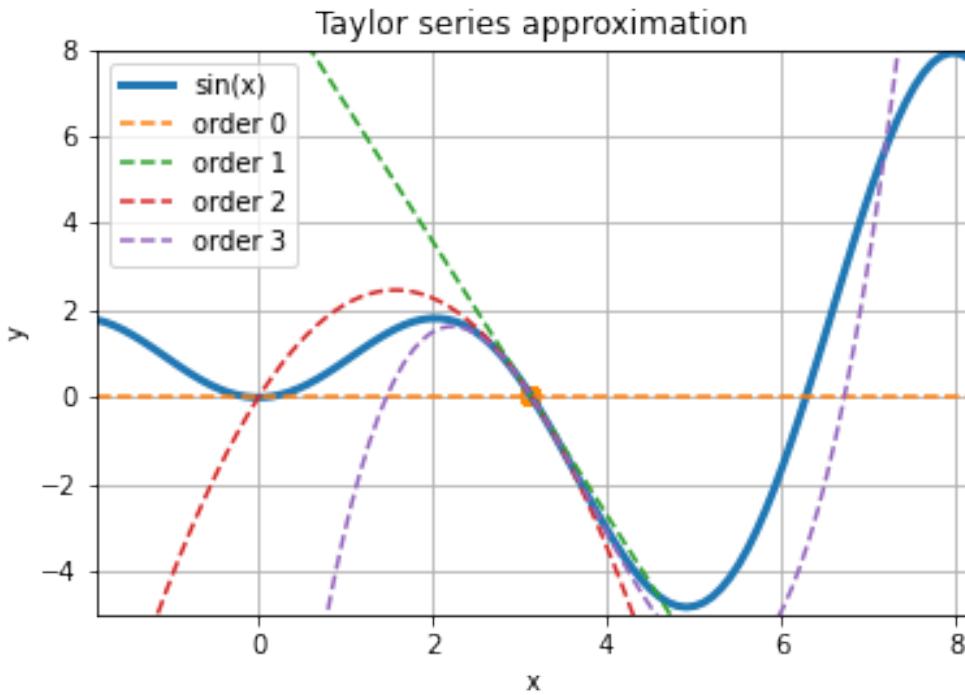
interact(plot_taylor, \
         x0=widgets.FloatSlider(min=2, max=7, value=3.1415927, \
         continuous_update=False), \
         order=widgets.IntSlider(min=0, max=8, value=3, \
         continuous_update=False));

```

```

interactive(children=(FloatSlider(value=3.1415927, continuous_update=False, description='x0', \

```



Using polynomials is obviously easier to handle than more complicated function types. Very often, a **linearization** is enough. Linearization is a Taylor series up to order 1, so a linear approximation to some function. Still, the second derivative can give us critical information about the problem. We'll see this in a later lesson today.

The Taylor series for multivariate vector functions $\mathbf{f}(\mathbf{x})$ is “simply” the Taylor series of each component.

2.15 Pandas

We've seen how NumPy can handle data and manipulate it. There are some detriments though. E.g., NumPy only stores arrays of numbers with a fixed data type (unless using structured arrays, but they're not nice to handle). If you have labeled data, say, sequential sensor data and timestamps (including dates), you'd have to either create two arrays or find a format that can turn floats into timestamps again. You also have to keep track of what the columns represent. This is tedious and error-prone. The resulting data won't be easy to read by humans or scattered among various arrays.

`pandas` solves this problem and tries to enforce a better data handling strategy. It works similarly to relational databases and allows similar operations on data. The basic data structure used by `pandas` is a *data frame*. It makes use of quite a few NumPy features, so in many cases we need to import both:

```
import numpy as np
import pandas as pd
```

Pandas supports different kinds of data structures, all managed as data frames. A simple example is a `series`, where Pandas automatically creates running integer indices:

```
ser = pd.Series([1, 3, 5, np.nan, 6, 8])
print(ser)
```

```
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

The `np.nan` is pandas standard way of representing *missing data*, so it's even possible to keep track of where, e.g., sensor data couldn't be taken or some data was lost. We also see that printing a dataframe outputs a table. This is great for quickly understanding the type of data. What helps even more is a description of data columns:

```
# pandas can automatically create a range of dates
dates = pd.date_range('20130101', periods=6)

# create a data frame with random number data, date indices and column labels
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=["Value 1", "Value 2", "Value 3", "Value 4"])

print(df)
df
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.643929	-0.941704	1.814025	0.863829
2013-01-02	-1.572498	-0.422332	-1.180399	0.381553
2013-01-03	2.024738	0.514611	-0.330865	0.123765
2013-01-04	0.927695	0.740022	-1.002011	0.095836
2013-01-05	-0.106089	-1.574821	-0.866819	0.418917
2013-01-06	1.095029	1.025670	-0.267690	0.129252

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.643929	-0.941704	1.814025	0.863829
2013-01-02	-1.572498	-0.422332	-1.180399	0.381553
2013-01-03	2.024738	0.514611	-0.330865	0.123765
2013-01-04	0.927695	0.740022	-1.002011	0.095836
2013-01-05	-0.106089	-1.574821	-0.866819	0.418917
2013-01-06	1.095029	1.025670	-0.267690	0.129252

So far, the data always had the same data type. As mentioned, this is not necessary. The following code converts a Python dict to a dataframe:

```
df2 = pd.DataFrame({'val 1': 1.,
                    'val 2': pd.Timestamp('20130102'),
                    'val 3': pd.Series(1, index=list(range(4)), name='val 3'),
                    dtype='float32'),
                    'val 4': np.array([3] * 4, dtype='int32'),
                    'val 5': pd.Categorical(["test", "train", "test", "train"]),
                    'val 6': 'foo'})
```

```
print(df2.dtypes)
df2
```

```
val 1           float64
val 2    datetime64[ns]
val 3           float32
val 4            int32
val 5            category
val 6            object
dtype: object
```

	val 1	val 2	val 3	val 4	val 5	val 6
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

There are many options to address elements in a data frame:

```
print(df2["val 5"])
print()
print(df2["val 5"][2])
print()

print(df2.iloc[:,4])
print()
print(df2.iloc[2,4])
```

```
0    test
1   train
2    test
3   train
Name: val 5, dtype: category
Categories (2, object): ['test', 'train']

test
```

```
0      test
1    train
2      test
3    train
Name: val 5, dtype: category
Categories (2, object): ['test', 'train']
```

test

Data can be converted to numpy arrays:

```
df.to_numpy()
```

```
array([[-0.64392853, -0.94170415,  1.81402525,  0.86382945],
       [-1.57249811, -0.42233239, -1.18039864,  0.38155329],
       [ 2.02473785,  0.51461143, -0.33086461,  0.12376472],
       [ 0.9276945 ,  0.74002206, -1.00201072,  0.09583597],
       [-0.10608895, -1.57482072, -0.86681926,  0.41891713],
       [ 1.09502893,  1.02567042, -0.26768965,  0.12925243]])
```

To get a quick statistical summary of the data, `describe()` can be called:

```
df.describe()
```

	Value 1	Value 2	Value 3	Value 4
count	6.000000	6.000000	6.000000	6.000000
mean	0.287491	-0.109759	-0.305626	0.335525
std	1.308588	1.033190	1.100919	0.294286
min	-1.572498	-1.574821	-1.180399	0.095836
25%	-0.509469	-0.811861	-0.968213	0.125137
50%	0.410803	0.046140	-0.598842	0.255403
75%	1.053195	0.683669	-0.283483	0.409576
max	2.024738	1.025670	1.814025	0.863829

The transpose works just like in NumPy:

```
print(df2.T)
```

	0	1	2 \
val 1	1	1	1
val 2	2013-01-02 00:00:00	2013-01-02 00:00:00	2013-01-02 00:00:00
val 3	1	1	1
val 4	3	3	3
val 5	test	train	test
val 6	foo	foo	foo

```
val 1           1
val 2 2013-01-02 00:00:00
val 3           1
val 4           3
val 5          train
val 6          foo
```

There are `head` and `tail` commands like on UNIXoid OSs, to get a quick glimpse of what data is saved. The former prints a few of the beginning rows, the latter a few of the last lines:

```
df2.head()
#df2.tail()
```

```
    val 1      val 2  val 3  val 4  val 5 val 6
0    1.0 2013-01-02    1.0      3  test   foo
1    1.0 2013-01-02    1.0      3 train   foo
2    1.0 2013-01-02    1.0      3  test   foo
3    1.0 2013-01-02    1.0      3 train   foo
```

A pretty useful feature that's also implemented in NumPy is *boolean indexing*. Comparative statements like the following create *index masks*, which can be used to address non-continuous elements of a data frame:

```
df < 1
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	True	True	True	True
2013-01-02	True	True	False	True
2013-01-03	True	False	False	True
2013-01-04	False	True	True	True
2013-01-05	True	True	True	False
2013-01-06	True	True	True	True

```
new_df = df[df < 1]
new_df
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.245117	-0.393222	0.624931	0.391709
2013-01-02	-0.281136	-0.776862	NaN	-1.910095
2013-01-03	-1.553736	NaN	NaN	-0.561621
2013-01-04	NaN	-1.244856	0.034360	0.248727
2013-01-05	-0.164869	0.672306	-0.070947	NaN
2013-01-06	-1.097641	-0.892463	-0.877066	-1.536823

When a row contains missing data, it might not be useful for further processing at all. Pandas makes it easy to quickly get rid of all rows with missing data, or to change the missing data to some other value. This is part of a process called *data cleaning*. [PyJanitor](#) is a part of the Pandas ecosystem and automates a few of the common data cleaning tasks. We don't have time to go into

details here unfortunately, but it's worth checking out all the capabilities offered here for larger projects.

```
new_df.dropna(how='any')
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.245117	-0.393222	0.624931	0.391709
2013-01-06	-1.097641	-0.892463	-0.877066	-1.536823

```
new_df.fillna(value=0)
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.245117	-0.393222	0.624931	0.391709
2013-01-02	-0.281136	-0.776862	0.000000	-1.910095
2013-01-03	-1.553736	0.000000	0.000000	-0.561621
2013-01-04	0.000000	-1.244856	0.034360	0.248727
2013-01-05	-0.164869	0.672306	-0.070947	0.000000
2013-01-06	-1.097641	-0.892463	-0.877066	-1.536823

You can get the index mask of all missing values:

```
pd.isna(new_df)
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	False	False	False	False
2013-01-02	False	False	True	False
2013-01-03	False	True	True	False
2013-01-04	True	False	False	False
2013-01-05	False	False	False	True
2013-01-06	False	False	False	False

It's possible and sometimes a great help to apply functions to the data.

```
df.apply(np.sin)
```

	Value 1	Value 2	Value 3	Value 4
2013-01-01	-0.600342	-0.808562	0.970565	0.760335
2013-01-02	-0.999999	-0.409889	-0.924758	0.372363
2013-01-03	0.898726	0.492197	-0.324861	0.123449
2013-01-04	0.800240	0.674304	-0.842556	0.095689
2013-01-05	-0.105890	-0.999992	-0.762274	0.406771
2013-01-06	0.888941	0.855062	-0.264504	0.128893

2.15.1 Plotting

Pandas can use different rendering engines as backends for plotting. The standard backend is matplotlib, which we already saw earlier today. Plotting is greatly simplified with pandas and

many useful visualizations are already implemented and ready to use. If some things are missing, Pandas can be enhanced by a number of [libraries](#).

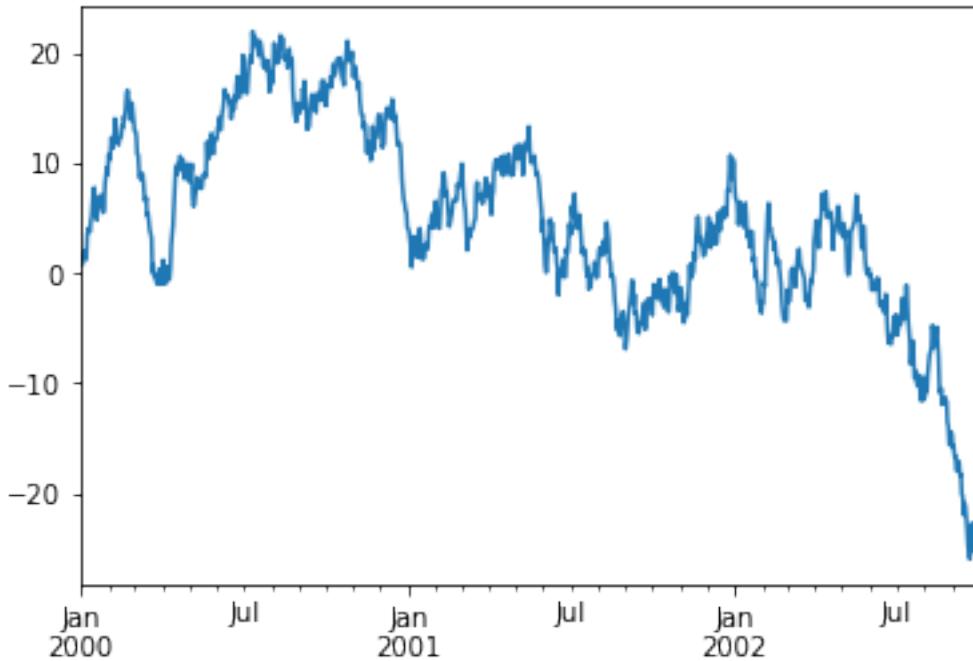
Plotting something is as simple as calling the `plot()` method of a data frame:

```
# create a random dataset with dates as indices
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000', periods=1000))

# get the cumulative sum along the columns to get a nicer plot
ts = ts.cumsum()

ts.plot()
```

`<AxesSubplot:>`



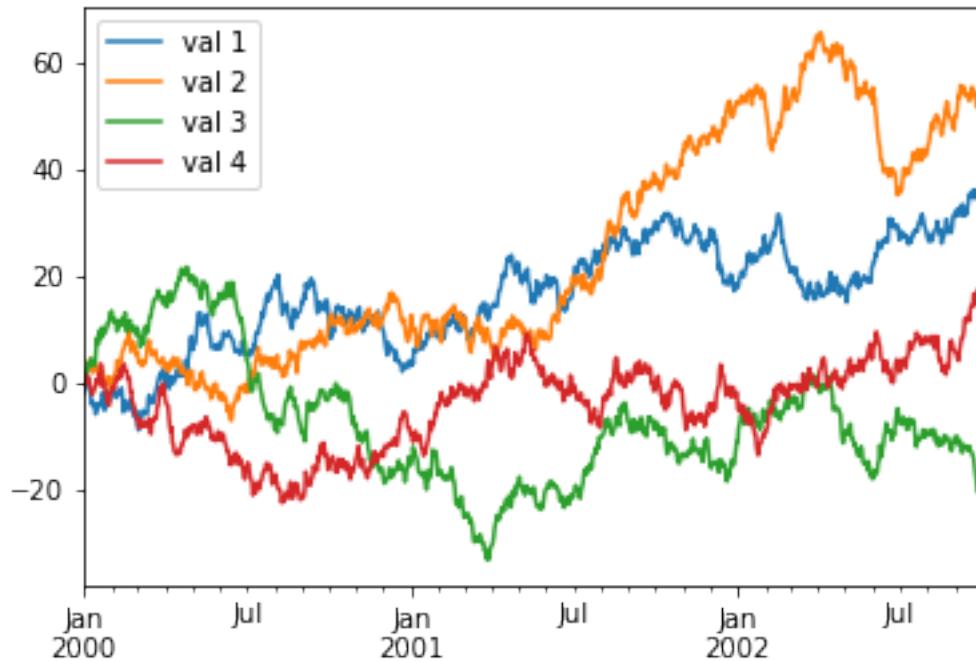
Pandas can directly plot different curves for all columns with labels, but to handle plot details, `pyplot` has to be imported first:

```
# create a random dataset with 4 dimensions and dates as indices
ts = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=["val 1", "val 2", "val 3", "val 4"])

# get the cumulative sum along the columns to get a nicer plot
ts = ts.cumsum()
```

```
ts.plot()
```

<AxesSubplot:>

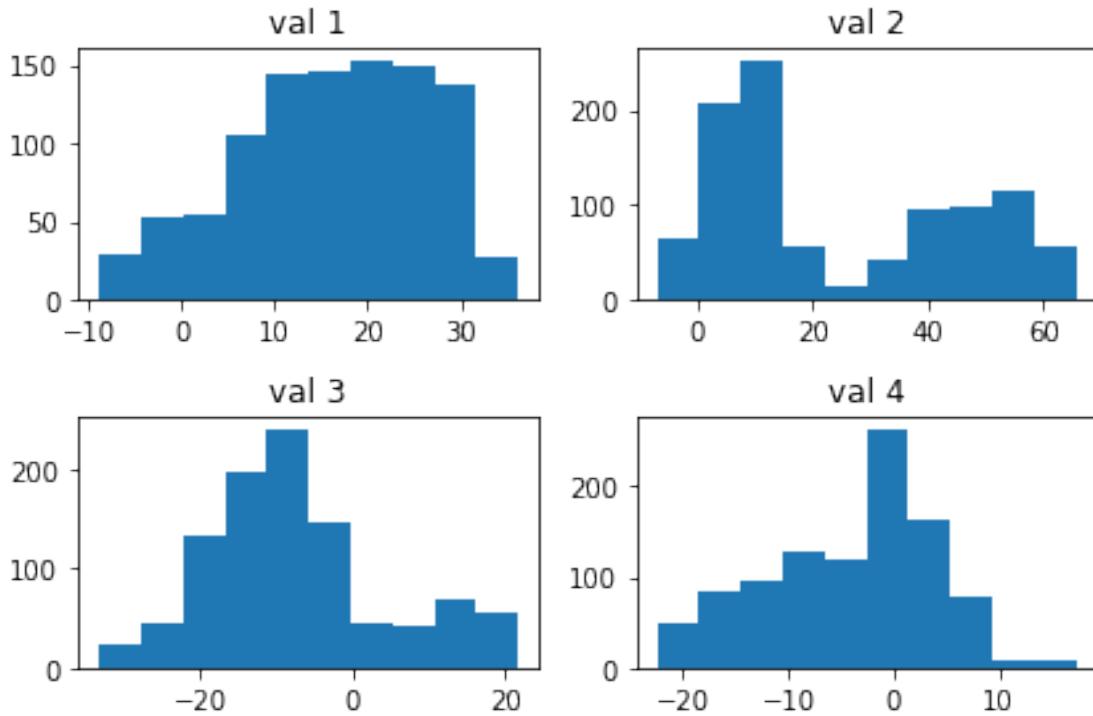


Histograms are also easy to draw. Here, we need to call the `tight_layout()` function from pyplot. To do so, it must be imported directly:

```
import matplotlib.pyplot as plt

ts.hist(grid=False)

plt.tight_layout()
```



2.15.2 Saving and Loading Data

Tabular-structured data can be exported as, e.g., a comma-separated values file:

```
ts.to_csv('ts.csv')
```

To read a csv file from disk:

```
ts_file = pd.read_csv('ts.csv')

ts_file
```

	Unnamed: 0	val 1	val 2	val 3	val 4
0	2000-01-01	-1.969445	-0.356405	1.026101	-0.762619
1	2000-01-02	-0.606335	1.367543	1.555996	-0.232897
2	2000-01-03	0.777804	1.326308	1.024626	0.017047
3	2000-01-04	0.655275	2.141760	1.549831	0.905668
4	2000-01-05	0.544901	2.625159	2.487415	1.339163
..
995	2002-09-22	36.063779	54.502244	-14.120248	16.600449
996	2002-09-23	34.513519	53.477599	-15.755223	14.546741
997	2002-09-24	35.042919	54.429797	-16.938643	14.938983
998	2002-09-25	36.122264	51.712579	-18.280598	16.089976
999	2002-09-26	35.800122	52.101227	-20.261030	17.299086

[1000 rows x 5 columns]

This was only a very quick overview of the basic capabilities of Pandas. There's much more than a part of a lecture can cover. The [documentation](#) is easy to digest and exhaustive. There's barely anything we'll need in this course that Pandas can't do.

3 Optimization

3.1 Derivatives

Most optimization techniques involve taking some derivative, so we'll shortly review different derivatives and their implications and interpretations in this section.

Often, a derivative is described as measuring how much something changes when there is a small change in another quantity. See for example this image:

```
%matplotlib inline
import sympy as sy
import numpy as np
from sympy.functions import sin,cos
from math import factorial
import matplotlib.pyplot as plt

# outsource plotting arrows in a function for efficiency
def plot_arrow(origin, slope):
    # need to cast to float, since sympy outputs a special type
    slope = float(slope)
    = np.arctan(slope)
    vec = [abs(slope)*np.cos(), abs(slope)*np.sin()]
    plt.arrow(*origin, *vec, head_width=0.2, head_length=0.2, \
              fc='red', ec='red', lw=3, length_includes_head=True, \
              zorder=10, label="Velocity vector")

# sympy needs all symbols that will be used declared
x = sy.Symbol('x')
# sympy functions must consist of sympy-compatible expressions
f = sin(x)*x
# this gets the analytical first derivative of a sympy-function
f_prime = f.diff(x,1)

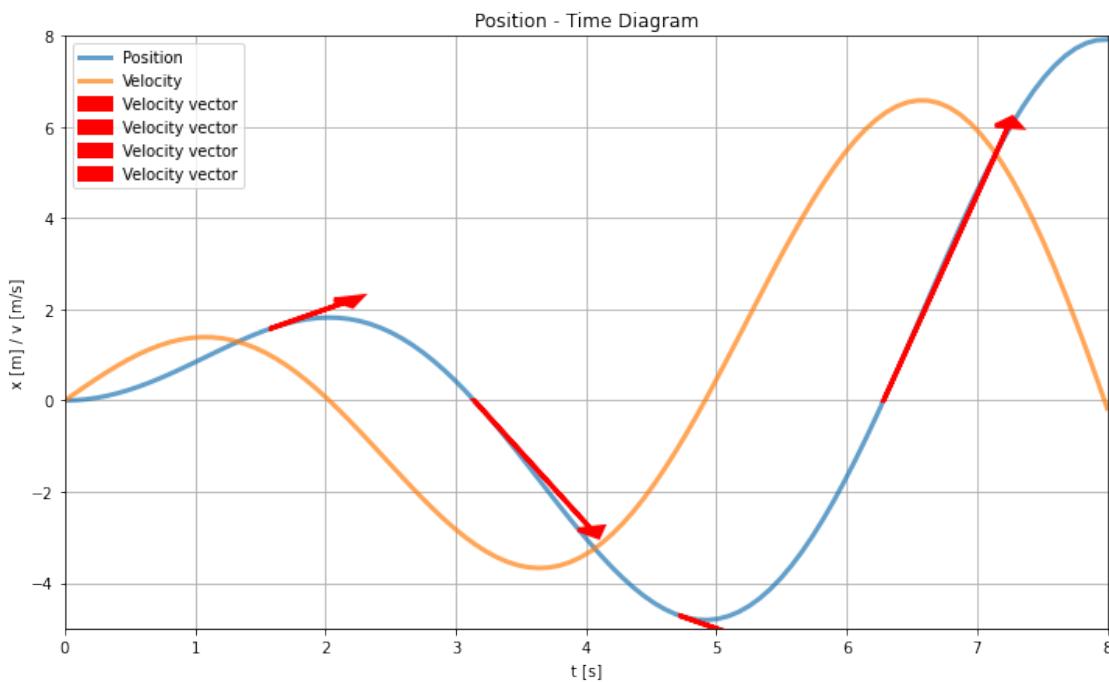
# set up the sampling space for the curves
x_lims = [0, 8]
x_range = np.linspace(x_lims[0], x_lims[1], 100)

# values for the curves
y1 = np.array([f.subs(x,x_val) for x_val in x_range])
y2 = np.array([f_prime.subs(x,x_val) for x_val in x_range])
```

```
# plot the curves
plt.figure(figsize=(12,7))
plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)
plt.plot(x_range,y2,label='Velocity', lw=3, alpha=0.7)

# plot a velocity arrow at multiples of /2
for i in range(1, 5):
    plot_arrow([i*np.pi/2,f.subs(x,i*np.pi/2).evalf()], f_prime.subs(x,i*np.pi/2).evalf())

# some plot options
plt.xlim(x_lims)
plt.ylim([-5,8])
plt.xlabel('t [s]')
plt.ylabel('x [m] / v [m/s]')
plt.legend()
plt.grid(True)
plt.title('Position - Time Diagram')
plt.show()
```



A more general definition would be that a derivative (of first order) is a **linear approximation** of a function at a specific point. This geometrical interpretation is often lost since for many functions, the derivative is only calculated as a number. See for example the following graph, where at $x = 3$ the derivative is plotted as a **tangent** to the curve. Use the slider to zoom closer to the point,

where the tangent is attached.

```

from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets

def plot_tangent(origin, slope, length):
    slope = float(slope)
    = np.arctan(slope)
#vec = [abs(slope)*np.cos(), abs(slope)*np.sin()]
    vec = [length*np.cos(), length*np.sin()]
    origin = [origin[i] - vec[i] for i in range(len(origin))]
    vec = [2*i for i in vec]
    plt.arrow(*origin, *vec, head_width=0, head_length=0, \
              fc='red', ec='red', lw=3, length_includes_head=True, \
              zorder=10, label="Velocity vector")

def plot_zoomable_function(factor, x_val):
    x = sy.Symbol('x')
    f = sin(x)*x
    f_prime = f.diff(x,1)

    x_lims = [max(0,(1-factor)*x_val), min((1+factor)*x_val,8)]
    x_range = np.linspace(x_lims[0], x_lims[1], 50)
    y1 = [f.subs(x,x_v) for x_v in x_range]

    plt.figure()
    plt.plot(x_range,y1,label='Position', lw=3, alpha=0.7)

    plot_tangent([x_val,f.subs(x,x_val).evalf()], f_prime.subs(x,x_val).\
    ↪evalf(), 0.5*(x_lims[1] - x_lims[0]))

    plt.xlim(x_lims)
    plt.ylim([-5,8])
    plt.xlabel('t [s]')
    plt.ylabel('x [m]')
    plt.legend()
    plt.grid(True)
    plt.title('Position - Time Diagram')
    plt.show()

interact(plot_zoomable_function, \
         factor=widgets.FloatSlider(min=0.1, max=2, value=1, \
         ↪continuous_update=False), \
         x_val=widgets.FloatSlider(min=2, max=7, value=3, \
         ↪continuous_update=False));

interactive(children=(FloatSlider(value=1.0, continuous_update=False, description='factor', max

```

When zooming in you will see that the curve resembles that tangent line ever more closely. This is what is meant with “the (first-order) derivative approximates functions linearly”. This is where the notion of **manifold** stems from. Manifolds are in general *curved* n -dimensional point sets that resemble flat \mathbb{R}^n when you zoom in close enough. Our earth for example, when viewed from space, is roughly a ball (oblate spheroid, actually looks more like a potato). Yet for us small beings viewing earth from the surface, it looks rather flat.

The derivative of a function $f(x)$ is defined as

$$f'(x = p) = \frac{\partial f}{\partial x}(x = p) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}(x = p), \quad (15)$$

where $\frac{\partial}{\partial x}$ denotes the **partial derivative** with respect to x . For multivariate functions $f(\mathbf{x})$ in \mathbb{R}^n , there are n partial derivatives $\frac{\partial}{\partial x_i}$, each for one of the n coordinates x_i of f .

$$\frac{\partial f}{\partial x_i}(\mathbf{x} = \mathbf{p}) = \lim_{h \rightarrow 0} \frac{f(x_1, x_2, \dots, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, x_2, \dots, x_n)}{h}(\mathbf{x} = \mathbf{p}), \quad (16)$$

Using the language introduced above, the partial derivatives of a function *span* the **tangential space** at that point, or, in other words, they are the **basis vectors** of that tangential space. Any derivative is a *linear combination* of these basis vectors. In 3d, this could look like the following graph (The surface is plotted with transparency, because matplotlib’s 3d capabilities are quite restricted):

```
from mpl_toolkits.mplot3d import axes3d

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')

# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")

# create all function points at once
z = func(x_m, y_m)
```

```
# this will plot the surface of the action on the vectors
ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)

# plot partial derivative in x-direction
ax.quiver(
    0, 0, 0, \
    float(f_x.subs(x, 1).evalf()), 0, 0, \
    color = 'red', alpha = .8, lw = 3, zorder=1000
)

# plot partial derivative in y-direction
ax.quiver(
    0, 0, 0, \
    0, float(f_y.subs(y, 1).evalf()), 0, \
    color = 'red', alpha = .8, lw = 3, zorder=1000
)

# you might need to adjust the axes limits
#ax.set_xlim(-2, 2)
#ax.set_ylim(-2, 2)
#ax.set_zlim( 0, 8)
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

<mpl_toolkits.mplot3d.art3d.Line3DCollection at 0x7fbca76dfd30>

For our purposes, partial derivatives form linear approximations to manifolds.

3.2 Gradient

The gradient of a function is the multivariate generalization of the partial derivatives.

$$\text{grad}_{\mathbf{x}} f(\mathbf{x}) = \nabla_{\mathbf{x}} f(\mathbf{x}) = \frac{\partial f}{\partial x_1} \hat{e}_1 + \frac{\partial f}{\partial x_2} \hat{e}_2 + \dots = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \end{bmatrix} \quad (17)$$

containing the partial derivatives in the rows. Often it's abbreviated to simply ∇f , where $\nabla = \left[\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \dots \right]^T$ is called “nabla”.

```
%matplotlib notebook
import sympy as sy
import numpy as np
```

```

from sympy.functions import sin,cos
from math import factorial
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

# sympy setup
x = sy.Symbol('x')
y = sy.Symbol('y')
f = -x**2 - y**2 # try different factors, especially a + instead of -
f_x = f.diff(x,1)
f_y = f.diff(y,1)

# first, a "figure" object needs to be created
fig = plt.figure(figsize=(8,5))
ax = fig.add_subplot(111, projection='3d')

# before something can be plotted, you need a space of inputs
mesh_points = np.linspace(-2,2,20)

# to create coordinate matrices out of this space (similar to matlab):
x_m, y_m = np.meshgrid(mesh_points, mesh_points)
n = np.zeros(x_m.shape)
o = np.ones(x_m.shape)

# lambdify makes it possible to apply a sympy function to whole arrays
func = sy.lambdify([x, y], f, "numpy")
func_x = sy.lambdify([x, y], f_x, "numpy")
func_y = sy.lambdify([x, y], f_y, "numpy")

z = func(x_m, y_m)

def plot_surface(surface, contours, cgrad, gradient):
    ax.cla()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_zlim(np.amin(z), np.amax(z))
    if surface:
        ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)
    if contours:
        ax.contour(x_m, y_m, z, zdir='z', offset=0, cmap='viridis')
    if cgrad:
        ax.quiver(x_m, y_m, n, func_x(x_m, y_m), func_y(x_m, y_m), n, \
                  length=0.05, normalize=False, alpha=0.7)
    if gradient:

```

```

    ax.quiver(x_m, y_m, func(x_m, y_m),
               func_x(x_m, y_m), func_y(x_m, y_m), n,
               length=0.05, normalize=False, alpha=0.7)

    plt.savefig("img/plot_surface.png")

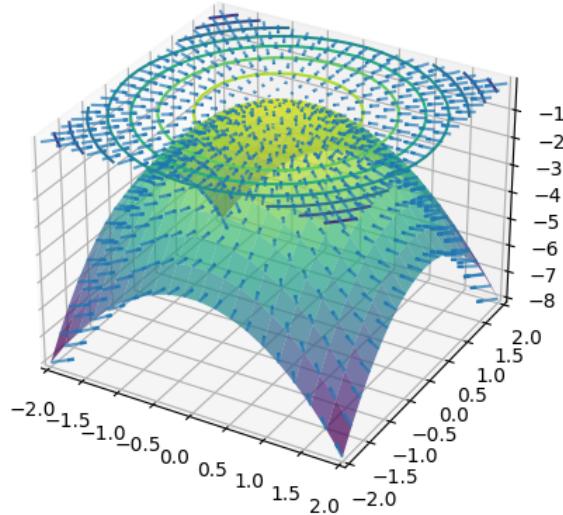
interact(plot_surface, surface=True, contours=True, cgrad=True, gradient=True)

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

interactive(children=(Checkbox(value=True, description='surface'), Checkbox(value=True, descrip
<function __main__.plot_surface(surface, contours, cgrad, gradient)>

```



It's often useful to plot **contour lines** of some function, which are curves of constant value, projected to a plane. The gradient is always perpendicular to the contour lines. This makes intuitive sense, after accepting the fact that the gradient always points in the direction of *steepest change*. We can easily see this after introducing the **directional derivative**

$$\text{grad}_{\mathbf{v}} f(\mathbf{x}) = \nabla_{\mathbf{v}} f(\mathbf{x}) = \mathbf{v} \cdot \nabla f(\mathbf{x}) \quad (18)$$

Since this is a scalar product, this can be written as

$$\nabla_{\mathbf{v}} f = |\mathbf{v}| |\nabla f| \cos(\theta) \quad (19)$$

This product becomes maximal, when \mathbf{v} and ∇f point in the same direction (are *colinear*), such that $\theta = 0$. Hence, the gradient always shows in the direction of *steepest ascent*. For a minimum, the vectors would have to point in opposite directions, such that $\theta = \pi$.

Let's look at an example:

```
# quadratic int grid, to make things easier
grid_size = 10
X = np.arange(0, grid_size, 1)
Y = np.arange(0, grid_size, 1)
X, Y = np.meshgrid(X, Y)

# list of x-coordinates and y-coordinates in separate lists
carriers = np.array([[2, 7, 4], [2, 7, 6]])
charges = np.array([-1, 1, -1])

plt.figure(figsize=(8, 8))

def plot_elec_field(n_carriers):
    plt.cla()
    Ex = np.zeros((grid_size, grid_size))
    Ey = np.zeros((grid_size, grid_size))

    for carrier in range(n_carriers):
        for i in range(grid_size):
            for j in range(grid_size):
                Ex[i, j] += charges[carrier] * (j - carriers[1][carrier]) / \
                            ((i - carriers[0][carrier])**2 + (j - carriers[1][carrier])**2)**1.5
                Ey[i, j] += charges[carrier] * (i - carriers[0][carrier]) / \
                            ((i - carriers[0][carrier])**2 + (j - carriers[1][carrier])**2)**1.5

    E = np.hypot(Ex, Ey)
    Ex /= E
    Ey /= E

    # plot charges
    plt.plot(*carriers[:, :n_carriers], 'bo', markersize=10)

    # plot field
    plt.quiver(X, Y, Ex, Ey, E, pivot='mid')
```

```
plt.axis('equal')
plt.axis('off')

plt.savefig("img/plot_elec_field.png")

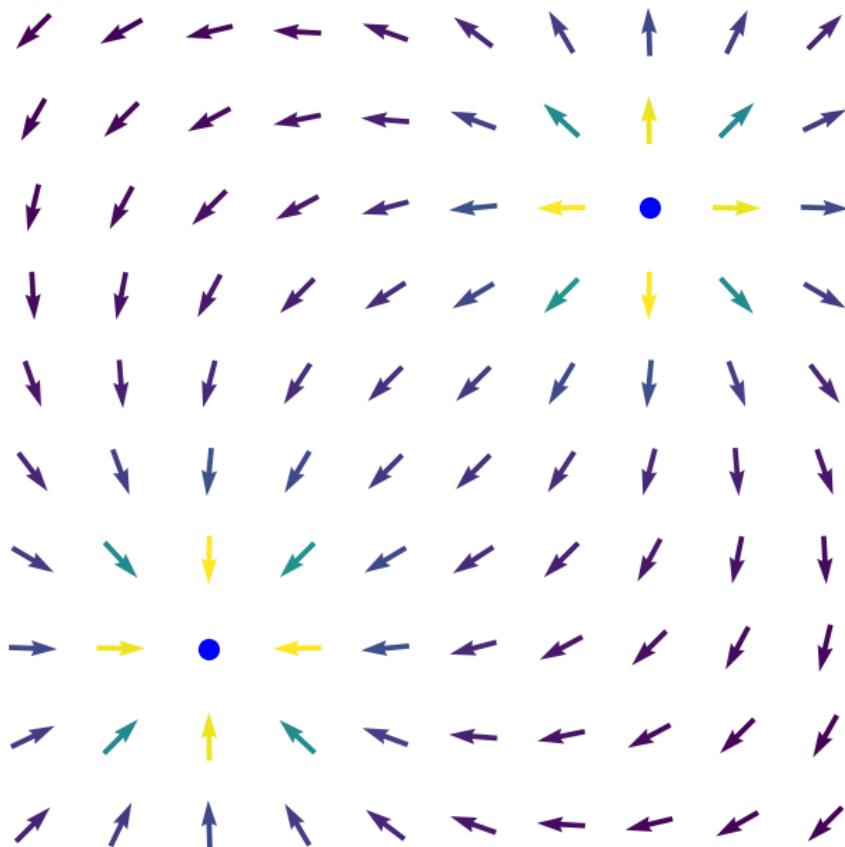
interact(plot_elec_field, n_carriers=(1, 3))
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```
interactive(children=(IntSlider(value=2, description='n_carriers', max=3, min=1), Output()), _
```

<function __main__.plot_elec_field(n_carriers)>



For vector functions $\mathbf{f}(\mathbf{x})$, the gradient $\nabla \mathbf{f}(\mathbf{x})$ is often called the **Jacobian matrix**:

$$J_{\mathbf{f}}(\mathbf{x}) = \frac{\partial f_i}{\partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \cdots & \frac{\partial f_n}{\partial x_m} \end{bmatrix} \quad (20)$$

which is an $n \times m$ matrix. The determinant of this matrix is called the **Jacobian determinant** and both are used in transforming between coordinate system, e.g. from euclidean coordinate systems to polar coordinate system.

3.3 Hessian

The Hessian of a function f is the gradient of the gradient, measuring how much the gradient vector changes with changing spacial position. The Hessian is symmetric (see Schwarz's theorem, compare to **stiffness matrix**), so it has real *eigenvalues* and *eigenvectors*. For scalar multivariate $f(\mathbf{x})$, the Hessian is defined as

$$H_f(\mathbf{x}) = \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n} \end{bmatrix} \quad (21)$$

which is an $n \times n$ -matrix. Each *column* contains the *gradient* of one component of the *gradient* of the original function, so each column tells us by how much that component of the gradient varies.

As in the 1d case, the Hessian as a second order derivative quantifies **curvature** of $f(\mathbf{x})$, with the only intricacy that curvature now depends on direction. A deep study of curvature is done in *differential geometry* and leads to several insights, that we cannot discuss in this short course.

Consider a function of the form $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{x}^T \mathbf{b} + c$. The gradient of this function is $\nabla f(\mathbf{x}) = \mathbf{A} \mathbf{x} + \mathbf{b}$. Its Hessian then is $H_f(\mathbf{x}) = \mathbf{A}$. Recall what we've learned about *quadratic forms*. In the following code, *eigenvalues* and *eigenvectors* are set and a corresponding quadratic form built up using a corollary of the **spectral theorem**: $\mathbf{A} = \sum_{i=1}^n \lambda_i \mathbf{v}_i \otimes \mathbf{v}_i$, where λ_i are the eigenvalues, \mathbf{v}_i are the eigenvectors of \mathbf{A} and \otimes denotes the dyadic product.

```
fig = plt.figure(figsize=(8,5))

ax = fig.add_subplot(111, projection='3d')

mesh_points = np.linspace(-2,2,20)

x, y = np.meshgrid(mesh_points, mesh_points)

vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T
```

```

def plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2):
    ax.cla()
    = np.array([eigval1, eigval2])
    v = np.array([[1, 0], [0, 1]])

    A = np.array(sum([ [i] * np.outer(v[i], v[i]) for i in range(.shape[0])]))
    q = np.array([vec.T @ A @ vec for vec in vecs])

    ax.set_xlim(np.amin(x), np.amax(x)+2)
    ax.set_ylim(np.amin(y), np.amax(y)+2)
    ax.set_zlim(np.amin(q)-2, np.amax(q))

    q = q.reshape(20,20)

    if plot_surface:
        ax.plot_surface(x, y, q, cmap='viridis', alpha=0.8)

    if plot_projs:
        ax.contourf(x, y, q, zdir='x', offset=np.amax(x)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='y', offset=np.amax(y)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='z', offset=np.amin(q)-2, cmap='viridis')

    if plot_eigv:
        for vi in v:
            ax.quiver(0, 0, 0, *vi, 0, color = 'red', lw = 3, zorder=1000)

    plt.savefig("img/plot_hessian.png")



interact(plot_hessian, \
         plot_surface=True, \
         plot_projs=True, \
         plot_eigv=True, \
         eigval1=widgets.IntSlider(min=-2, max=2, value=1, \
         continuous_update=False), \
         eigval2=widgets.IntSlider(min=-2, max=2, value=-1, \
         continuous_update=False))

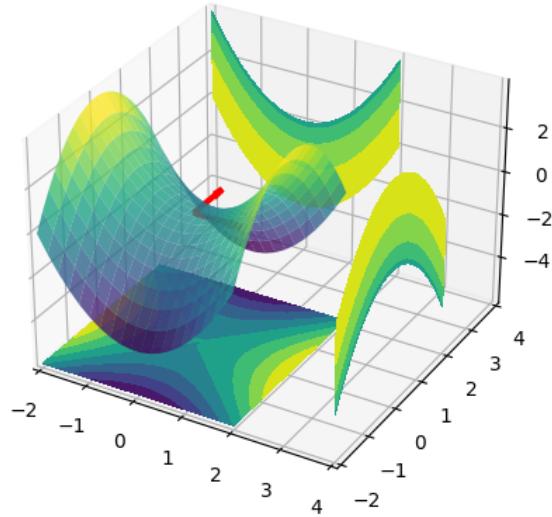
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```
interactive(children=(Checkbox(value=True, description='plot_surface'), Checkbox(value=True, d
```

```
<function __main__.plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1,
eigval2)>
```



3.4 Taylor Series

Recall that derivatives approximate functions at a specific point. The first derivative gives a linear approximation, the second derivative a quadratic approximation and equivalently for higher orders. Summing up these derivatives of a function $f(\mathbf{x})$ in a specific way gives the **Taylor series** of that function at a specific point \mathbf{a} :

$$T_f^n(\mathbf{a}) = \sum_{i=1}^n \frac{f^{(n)}(\mathbf{a})}{n!} (\mathbf{x} - \mathbf{a})^n = f(\mathbf{a}) + (\mathbf{x} - \mathbf{a})^T \nabla f + \frac{1}{2} (\mathbf{x} - \mathbf{a})^T H_f(\mathbf{a}) (\mathbf{x} - \mathbf{a}) + \dots \quad (22)$$

where $f^{(n)}(\mathbf{a})$ is the n -th derivative of f , evaluated at $\mathbf{x} = \mathbf{a}$. For “sufficiently nice” functions, this series converges to the function f itself in the limit $n \rightarrow \infty$.

The Taylor series approximates a function with **polynomials**. See a few examples below:

```
from sympy.functions import sin,cos
from math import factorial

plt.figure()

x = sy.Symbol('x')
```

```

f = x*sin(x)
func = sy.lambdify(x, f, "numpy")

# Taylor expansion at x0
def taylor(function,x0,n):
    p = 0
    for i in range(n):
        p += function.diff(x, i).subs(x, x0) / factorial(i) * (x - x0)**(i)

    return p

def plot_taylor(x0, order):
    plt.cla()

    x_lims = [x0 - 5, x0 + 5]

    x1 = np.linspace(x_lims[0], x_lims[1], 100)
    y1 = []

    plt.plot(x1,func(x1),label='sin(x)', lw=3)
    plt.scatter(x0,float(f.subs(x,x0).evalf()), lw=8, color="darkorange", marker="+")

    for i in range(1,order+2):
        f_curr = taylor(f,x0,i)
        print('Taylor expansion at x0, order ' + str(i-1) + ": ", f_curr)
        for k in x1:
            y1.append(f_curr.subs(x,k))
        plt.plot(x1,y1,label='order '+str(i-1), linestyle="--")
        y1 = []

    plt.xlim(x_lims)
    plt.ylim([-5,8])
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend()
    plt.grid(True)
    plt.title('Taylor series approximation')

    plt.savefig("img/plot_taylor.png")

interact(plot_taylor, \
         x0=widgets.FloatSlider(min=2, max=7, value=3.1415927, continuous_update=False), \

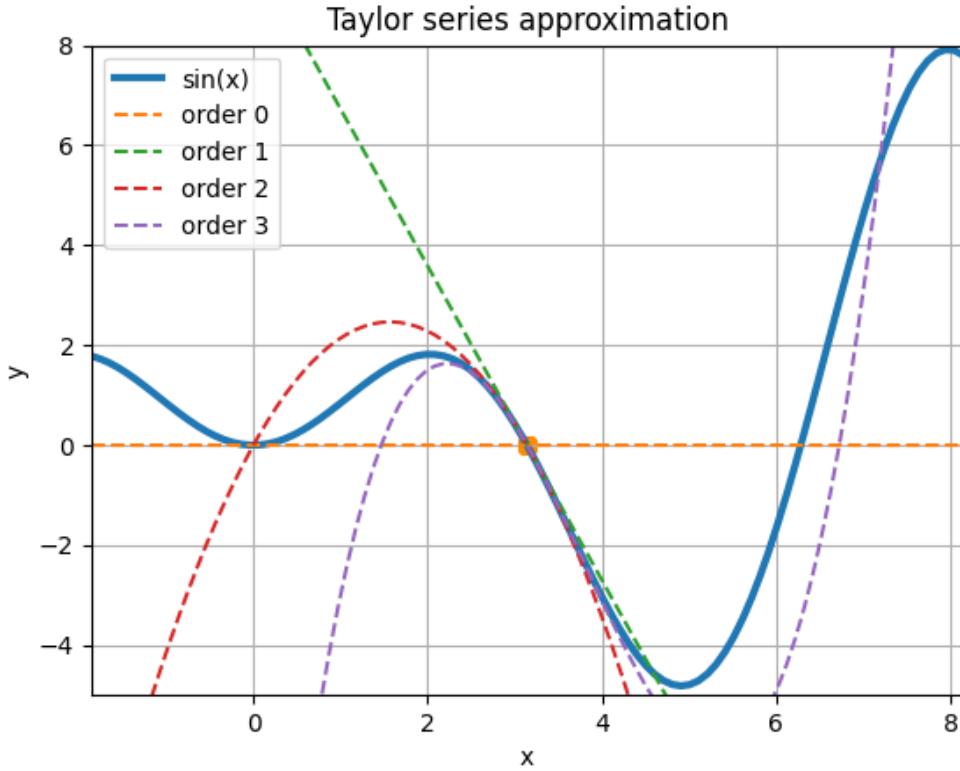
```

```
order=widgets.IntSlider(min=0, max=8, value=3,
    continuous_update=False));
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>

```
interactive(children=(FloatSlider(value=3.1415927, continuous_update=False, description='x0',
```



Using polynomials is obviously easier to handle than more complicated function types. Very often, a **linearization** is enough. Linearization is a Taylor series up to order 1, so a linear approximation to some function. Still, the second derivative can give us critical information about the problem. We'll see this in a later lesson today.

The Taylor series for multivariate vector functions $\mathbf{f}(\mathbf{x})$ is “simply” the Taylor series of each component.

3.5 Matrices

Training in ML and analyzing the process requires evaluating how a vector changes with respect to another vector, say, how an output vector changes with respect to the model parameters $\frac{\partial \mathbf{v}}{\partial \mathbf{w}}$. This requires some basic knowledge of *matrix calculus*. There are different constellations:

3.5.1 Vectors and Scalars

The derivative of a vector $\mathbf{v} \in \mathbb{R}^n$ with respect to a scalar $\alpha \in \mathbb{R}$: $\mathbf{w} = \frac{\partial \mathbf{v}}{\partial \alpha}$ is a vector $\mathbf{w} \in \mathbb{R}^n$ again. This is often seen in parameterized curves, such as

$$\mathbf{c}(\alpha) = \begin{bmatrix} \cos(\alpha) \\ \alpha \\ \alpha^2 \end{bmatrix} \quad \mathbf{c}'(\alpha) = \begin{bmatrix} -\sin(\alpha) \\ 1 \\ 2\alpha \end{bmatrix} \quad (23)$$

where α could be an *arc length* or *time* parameter for example.

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact#, interactive, fixed, interact_manual
#import ipywidgets as widgets

= np.linspace(-7, 7, 100)

x = np.cos( )
y =
z = **2

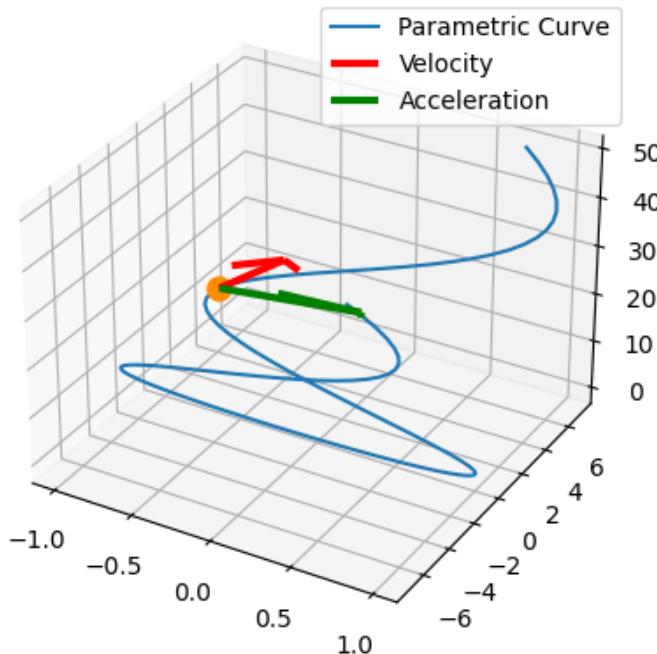
def plot_pos( =3.5):
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    #ax.cla()
    pos = np.cos( ), , **2
    ax.plot(x, y, z, label='Parametric Curve')
    ax.scatter(*pos, color='darkorange', lw=6)
    ax.quiver(*pos, -np.sin( ), 1, 2*, color='red', lw = 3, zorder=1000, u
    ↪label='Velocity')
    ax.quiver(*pos, -np.cos( ), 0, , color='green', lw = 3, zorder=1000, u
    ↪label='Acceleration')
    ax.legend()

    plt.savefig("img/plot_pos.png")

interact(plot_pos, =(-7.0, 7.0))
```

```
interactive(children=(FloatSlider(value=3.5, description=' ', max=7.0, min=-7.0), Output()), _d
```

```
<function __main__.plot_pos(=3.5)>
```



In the inverse case, where the derivative of a scalar f is taken with respect to a vector, this is simply the (directional) gradient again. Take for example the function $f(x, y, z) = x \cdot y^2 \cdot z^3$, then

$$\nabla f = \begin{bmatrix} y^2 z^3 \\ 2xyz^3 \\ 3xy^2 z^2 \end{bmatrix} \quad (24)$$

```
import sympy as sy
from sympy.vector import gradient, CoordSys3D
import matplotlib.patches as patches
from matplotlib import cm, colors, colorbar
from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection

x, y, z = sy.symbols('x, y, z')
X = [x, y, z]
```

```

f = x * y**2 * z**3
f_grad = sy.derive_by_array(f, X)
# it's possible to repeat the last line to get the analytical Hessian,
# but plotting its action on vectors at each gridpoint just looks messy

print("Function: f(x,y,z) = ", f)
print("Gradient: f'(x,y,z) = ", f_grad)

func = sy.lambdify([x, y, z], f, "numpy")
func_grad = sy.lambdify([x, y, z], f_grad, "numpy")

# this is the old create_parallelpipiped_3d function, where origin is added ↪
everywhere
def create_cube_3d(origin, sidelength):
    v1, v2, v3 = [sidelength * np.eye(3)[i] for i in range(3)]
    origin -= 0.5*(v1 + v2 + v3)
    return [[origin, origin+v1, origin+v1+v2, origin+v2], \
            [origin+v3, origin+v1+v3, origin+v1+v2+v3, origin+v2+v3], \
            [origin, origin+v1, origin+v1+v3, origin+v3], \
            [origin+v1+v2, origin+v2, origin+v2+v3, origin+v1+v2+v3], \
            [origin+v1, origin+v1+v2, origin+v1+v2+v3, origin+v1+v3], \
            [origin+v3, origin+v2+v3, origin+v2, origin]]]

fig = plt.figure()
ax = fig.gca(projection='3d')
cax = fig.add_axes([0.95, 0.1, 0.5, 0.45])
cmap = cm.bwr_r

cuts = 5

xs = np.linspace(-2, 2, cuts)
ys = np.linspace(-2, 2, cuts)
zs = np.linspace(-2, 2, cuts)
xg, yg, zg = np.meshgrid(xs,
                           ys,
                           zs)

data = func(xg, yg, zg)
grad_data = func_grad(xg, yg, zg)

ax.cla()
ax.quiver(xg, yg, zg, *grad_data, length=0.01)

norm = colors.Normalize(vmin=data.min(), vmax=data.max())
colors = lambda i,j,k : cm.ScalarMappable(norm=norm,cmap = "bwr_r") .
    ↪to_rgba(data[i,j,k])

for i, xi in enumerate(xs):

```

```

for j, yj in enumerate(ys):
    for k, zk in enumerate(zs):
        ax.add_collection3d(Poly3DCollection(create_cube_3d([xi,yj,zk], 5/
→cuts), \
            facecolors=colors(i,j,k), linewidths=1, edgecolors=None, alpha=.
→25))

cbar = colorbar.ColorbarBase(cax, cmap=cmap, norm=norm, orientation='vertical')
cbar.set_ticks(np.unique(data))
cbar.solids.set(alpha=0.15)

plt.savefig("img/grad_3d.png")

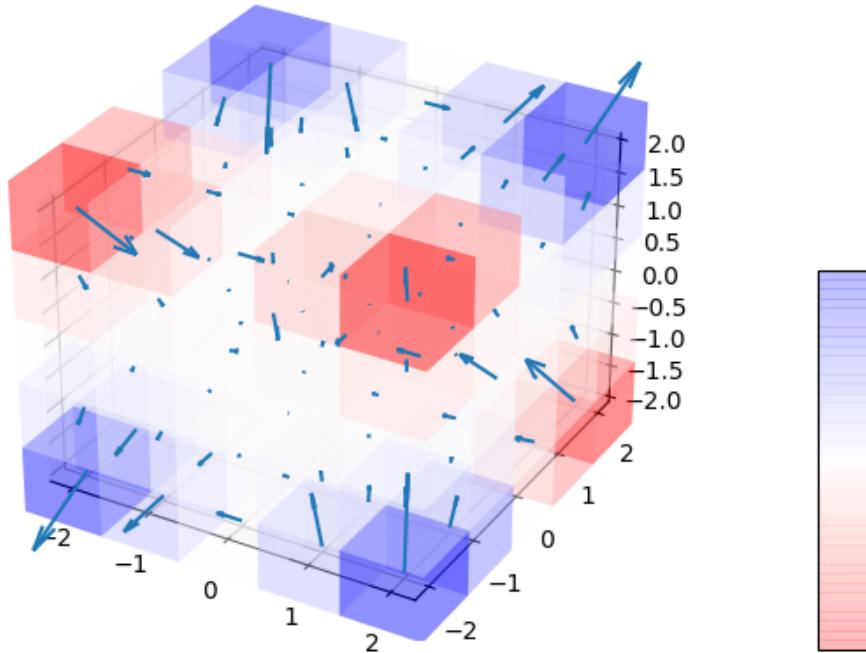
```

Function: $f(x,y,z) = x*y**2*z**3$

Gradient: $f'(x,y,z) = [y**2*z**3, 2*x*y*z**3, 3*x*y**2*z**2]$

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



3.5.2 Two Vectors

The derivative of a vector \mathbf{v} with respect to some other vector \mathbf{w} (or the directional derivative of a vector field) $\nabla_{\mathbf{w}} \mathbf{v} = \frac{\partial \mathbf{v}}{\partial \mathbf{w}}$ measures the rate of change of each component of that vector field with respect to each component of the vector \mathbf{w} . An example could be the derivative of a velocity field wrt position. The result of this operation is a tensor of 2nd order, or a matrix. The directional derivative of a product of two vectors is

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x} \cdot \mathbf{w}) = \frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T \cdot \mathbf{w}) = \frac{\partial}{\partial \mathbf{x}}(\mathbf{w}^T \cdot \mathbf{x}) = \mathbf{w}^T \quad (25)$$

where the transpose converts a (contravariant) vector into an element of the *dual vector space* (covariant). Elements of this dual space are linear maps, that map a vector of the original vector space to the reals, a scalar. The relation above gets more complicated when a different directional derivative is asked for, it looks like the one for the *derivative of a Matrix Product*.

3.5.3 Derivative of a Matrix Product

The gradient of a matrix product is

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{A} \cdot \mathbf{B}) = \frac{\partial \mathbf{A}}{\partial \mathbf{x}} \cdot \mathbf{B} + \mathbf{A} \cdot \frac{\partial \mathbf{B}}{\partial \mathbf{x}} \quad (26)$$

although this is not the full truth. This relation will suffice though. The reason that this is a sum of two terms that cannot be simplified is that in general, matrices do not *commute* (the commutator of two operators/matrices is $[\mathbf{A}, \mathbf{B}] = \mathbf{A} \cdot \mathbf{B} - \mathbf{B} \cdot \mathbf{A}$). For quadratic forms, the derivative is

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T \mathbf{A} \mathbf{x}) = (\mathbf{A} + \mathbf{A}^T) \mathbf{x} \quad (27)$$

The quadratic form gives a scalar, whereas its gradient gives a vector, as it does here. The expression on the right side is twice the symmetric part of \mathbf{A} , so for *symmetric matrices* this simplifies to

$$\frac{\partial}{\partial \mathbf{x}}(\mathbf{x}^T \mathbf{A} \mathbf{x}) = 2\mathbf{A}\mathbf{x} \quad \text{given symmetric } \mathbf{A} \quad (28)$$

3.6 Unconstrained Optimization

As we've already talked about, most ML algorithms map input features to some output, where the parameters of the model are determined by a *training* process that tries to find the best fit for given data. Hence, almost every ML algorithm can be thought of as an optimization problem and can explicitly be written as one.

In general, an optimization task would be to find the minimum or maximum of a function $f(\mathbf{x})$ by varying \mathbf{x} . f is called the *objective function*, or in the context of ML usually **loss function** or **cost function**. It might be a scalar function, in which case the optimization would be *single-objective*, or it might be multivariate, which would constitute a *multi-objective* optimization. In ML, cost functions are in the majority of cases scalar, so we'll restrict ourselves to single-objective optimization here. An example would be simply taking the length of a vector, which is done for example when training *autoencoders*: $f(\mathbf{x}) = \sqrt{x_1^2 + x_2^2 + \dots}$.

All optimization problems can be reduced to finding the minimum of some function, as in finding the \mathbf{x} that makes $f(\mathbf{x})$ take its minimum ($\max(f) = \min(-f)$). The notation for such a problem is

$$\mathbf{x}_{\text{opt}} = \arg \min_{\mathbf{x}} f(\mathbf{x}) \quad (29)$$

This is not to be confused with the minimum itself. Take for example the shifted parabola $f(x) = x^2 + 2$. The $\arg \min$ of f is $x_{\text{opt}} = 0$, but the minimum is $f(x = 0) = 2$.

3.6.1 Scalar Optimization

Let's examine the graph below. Since here we're talking about unconstrained optimization, $x \in \mathbb{R}$ can take on any value on the real line.

```
### some function with a few minima/maxima,
### mark with scatter in legend, print values of ' and ''
%matplotlib inline
import sympy as sy
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

x      = sy.Symbol('x')
f      = sy.exp(-0.04*x**2)*sy.sin(x)
f_x   = f.diff(x,1)
f_xx = f.diff(x,2)

func    = sy.lambdify([x], f,      "numpy")
func_x = sy.lambdify([x], f_x,   "numpy")
func_xx = sy.lambdify([x], f_xx, "numpy")

print("f(x)    = ", f)
print("f'(x)   = ", f_x)
print("f''(x)  = ", f_xx)

X = np.linspace(-3*np.pi, 3*np.pi, 100)

def plot_point(x_val):
    fig = plt.figure(figsize=(8,5))
    ax = fig.add_subplot(111)
    ax.cla()
    ax.plot(X, func(X))
    ax.scatter(x_val, func(x_val), lw=3, color='darkorange', zorder=10)
    print("f(" + str(x_val) + ")    = ", func(x_val))
    print("f'(" + str(x_val) + ")   = ", func_x(x_val))
    print("f''(" + str(x_val) + ") = ", func_xx(x_val))
```

```

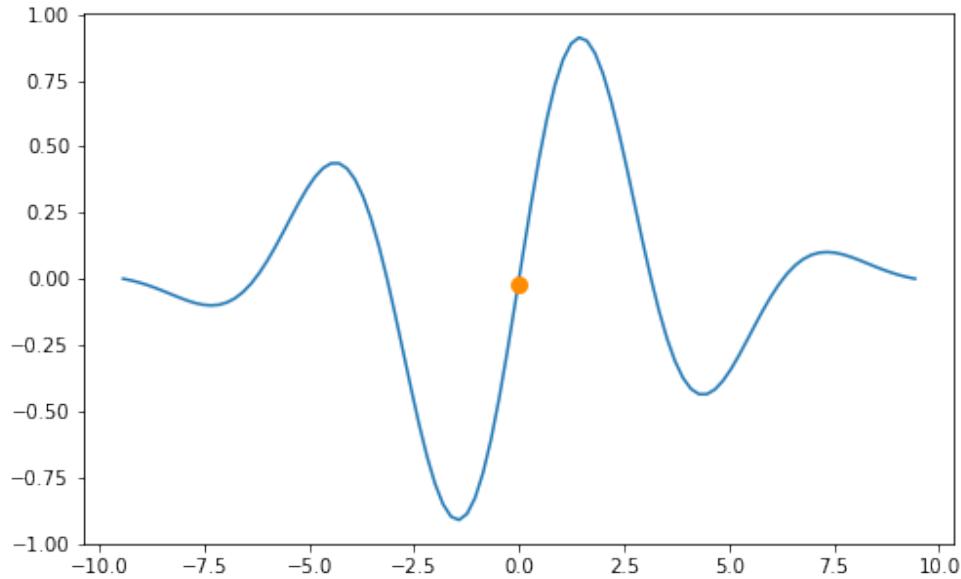
plt.savefig("img/plot_point.png")

interact(plot_point, x_val=(-3*np.pi, 3*np.pi, 0.1))

f(x)    =  exp(-0.04*x**2)*sin(x)
f'(x)   =  -0.08*x*exp(-0.04*x**2)*sin(x) + exp(-0.04*x**2)*cos(x)
f''(x)  =  (-0.16*x*cos(x) + (0.0064*x**2 - 0.08)*sin(x) -
sin(x))*exp(-0.04*x**2)

interactive(children=(FloatSlider(value=-0.024777960769378993, description='x_val', max=9.4247
<function __main__.plot_point(x_val)>

```



The points x_c at which $f'(x_c) = 0$ and $f(x_c)$ takes on its *extremal* values are called **critical points** or **stationary points**. To confirm whether one such critical point makes f take on a maximum or minimum, the second derivative of f is investigated at those points. If $f''(x_c) > 0$, $f(x_c)$ takes on a minimum. If $f''(x_c) < 0$, $f(x_c)$ takes on a maximum. If $f''(x_c) = 0$, $f(x_c)$ has a saddle point at that x_c . We have seen this behavior in previous lessons.

3.6.2 Multivariate Optimization

When x is a vector and $f(\mathbf{x})$ still a scalar function, the gradient is used instead of the simple derivative. The points minimizing or maximizing $f(\mathbf{x})$ are still called stationary or critical points, where f takes on local minima, local maxima or saddle points. The type can be determined by

examining the *Hessian* of f . If the Hessian is positive/negative definite (meaning *all* its eigenvalues are absolutely positive/negative) at a point $\mathbf{x} = \mathbf{p}$, $f(\mathbf{x} = \mathbf{p})$ has a local minimum/maximum. Compare these notions to the visualization for the Hessian in lesson 03-1, where you could preset the eigenvalues. When both were negative, the shape was mountain-like, if both were positive, it formed a valley. When the eigenvalues had different signs, a saddle shaped surface was built. Note that also the shapes that formed for one eigenvalue of $\lambda_i = 0$ are called saddle points. To summarize, if the Hessian is neither positive definite, nor negative definite, the function has a saddle point at that critical point.

```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets

mesh_points = np.linspace(-2,2,20)

x, y = np.meshgrid(mesh_points, mesh_points)

vecs = np.array([x.reshape(400), \
                 y.reshape(400)]).T

def plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2):
    fig = plt.figure(figsize=(8,5))
    ax = fig.add_subplot(111, projection='3d')
    #ax.cla()
    = np.array([eigval1, eigval2])
    v = np.array([[1, 0], [0, 1]])

    A = np.array(sum([ [i] * np.outer(v[i], v[i]) for i in range(.shape[0])]))

    q = np.array([vec.T @ A @ vec for vec in vecs])

    ax.set_xlim(np.amin(x), np.amax(x)+2)
    ax.set_ylim(np.amin(y), np.amax(y)+2)
    ax.set_zlim(np.amin(q)-2, np.amax(q))

    q = q.reshape(20,20)

    if plot_surface:
        ax.plot_surface(x, y, q, cmap='viridis', alpha=0.8)

    if plot_projs:
        ax.contourf(x, y, q, zdir='x', offset=np.amax(x)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='y', offset=np.amax(y)+2, cmap='viridis')
        ax.contourf(x, y, q, zdir='z', offset=np.amin(q)-2, cmap='viridis')

    if plot_eigv:
```

```

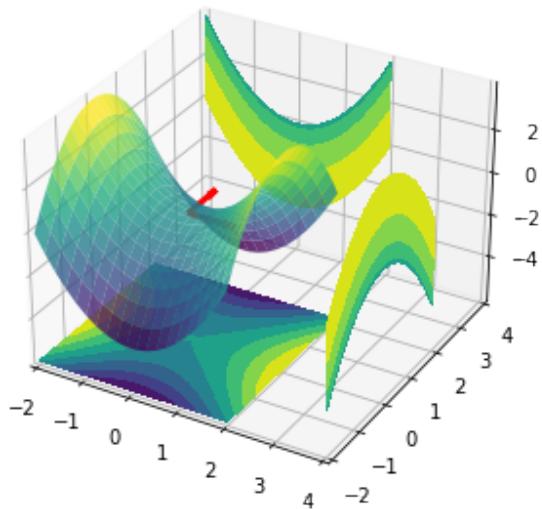
for vi in v:
    ax.quiver(0, 0, 0, *vi, 0, color = 'red', lw = 3, zorder=1000)

plt.savefig("img/plot_hess.png")

interact(plot_hessian, \
    plot_surface=True, \
    plot_projs=True, \
    plot_eigv=True, \
    eigval1=widgets.IntSlider(min=-2, max=2, value=1, \
    continuous_update=False), \
    eigval2=widgets.IntSlider(min=-2, max=2, value=-1, \
    continuous_update=False))

interactive(children=(Checkbox(value=True, description='plot_surface'), Checkbox(value=True, d
<function __main__.plot_hessian(plot_surface, plot_projs, plot_eigv, eigval1, eigval2)>

```



3.7 Constrained Optimization

In contrast to last lesson, here the minimization of the objective function happens for a constrained x . This is very usual in engineering, where for example you wouldn't just ask which rocket design would be the best to get the farthest with lowest fuel consumption, but also leave some room for payload and perhaps human operators. Another example would be to design not the fastest airplane, but the fastest airplane given some restrictions for fuel efficiency. A mathematical example would be to find the minimum of, say, $f(\mathbf{x}) = x_1x_2 + x_3^2$, where $\|\mathbf{x}\| < 1$, so where all solutions must lie inside the unit ball.

An important fact is that *all* constraints can be converted to either equality constraints, like $\arg \min_{\mathbf{x}} f(\mathbf{x})$ subject to $|x_1| + |x_2| + |x_3| = 1$ (say, a string of certain length with which an area is to be surrounded that must be as large as possible) or to inequality constraints. For the latter, one could change the equality sign in the above example to $<$ (maximize velocity while keeping fuel consumption below some threshold). The canonical form to write this is “minimize $f(\mathbf{x})$ subject to the constraint that $\mathbf{x} \in S$, where $S = \{\mathbf{x} | \forall i g_i(\mathbf{x}) = 0 \text{ and } \forall j h_j(\mathbf{x}) \leq 0\}$ ”. These \mathbf{x} are called **feasible points** and can be subjected to multiple constraints at once. In the string example from above, one could fix the string length, but restrict the width of such an area to be less than some threshold.

There are many approaches to these types of problems, but the most well-known approach is using **generalized Lagrange functions**:

$$L(\mathbf{x}, \lambda, \alpha) = f(\mathbf{x}) + \lambda \cdot \mathbf{g}(\mathbf{x}) + \alpha \cdot \mathbf{h}(\mathbf{x}) \quad (30)$$

The minimum of the constrained f becomes

$$\min_{\mathbf{x} \in S} f(\mathbf{x}) = \min_{\mathbf{x}} \max_{\lambda} \max_{\alpha > 0} L(\mathbf{x}, \lambda, \alpha) \quad (31)$$

where λ and α are called **Lagrange-multipliers**. Their physical interpretation is that they are the **constraining forces** keeping the system on its constrained trajectory in *configuration space*. An example are centripetal forces that keep rotating bodies on their circular trajectory. We will see more of this when coming to *Support Vector Machines*.

This technique is used in various theories, such as the *Theory of Porous Media*, where saturation rates of multiple phases must add up to zero, such that no net material is produced: $n'_S + n'_F = 0$. This constraint is added to the entropy inequality using a Lagrange-multiplier that in further analysis turns out to be the pore pressure. It's also used for modeling *contact* problems.

*Side note: the geometrical interpretation of the Lagrange function is that it is something like an “energy” of a system in configuration space. The Euler-Lagrange equation finds the **geodesics** in configuration space. In other words, it finds the paths of least “distance” between two states, where distance is induced by the Lagrange function.*

Let's look at a simple example, to see what Lagrange-multipliers mean:

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
```

```

from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact#, interactive, fixed, interact_manual
# import ipywidgets as widgets
from scipy.optimize import minimize

# this is the objective function we want to minimize
def objective(X):
    x, y = X
    return -x**2 + y**2

# this is the constraint equality
def constr(X):
    x, y = X
    return 1 * np.sin(6*x-0.7) - y - 0.5

def grad_obj(X):
    x, y = X
    return np.array([-2*x, 2*y])

def grad_con(X):
    x, y = X
    return np.array([6*np.cos(6*x-0.7), -1])

sol = minimize(objective, [1, -0.5], constraints={'type': 'eq', 'fun': constr})

mesh_points = np.linspace(-2,2,50)
x_m, y_m = np.meshgrid(mesh_points, mesh_points)
ys = 1*np.sin(6*mesh_points-0.7) - 0.5
z = objective([x_m, y_m])
g = constr([x_m, y_m])

def plot_lagrange_multi(surface, contour, gradients, solution):
    fig = plt.figure(figsize=(8,5))
    ax = fig.add_subplot(111, projection='3d')
    #ax.cla()
    ax.set_xlim(-2, 2)
    ax.set_ylim(-2, 2)
    ax.set_zlim(np.amin(z)-1, np.amax(z))
    if surface:
        ax.plot_surface(x_m, y_m, z, cmap='viridis', alpha=0.7)
    if contour:
        ax.contour(x_m, y_m, z, zdir='z', offset=np.amin(z)-1, levels=18,
        cmap='viridis', alpha=0.7)
        ax.plot(mesh_points, ys, zs=np.amin(z)-0.99, zdir='z',
        color='darkorange', lw=3)
    if constraint:

```

```

        ax.plot_surface(x_m, y_m, np.zeros(x_m.shape), color='lightgray', □
→alpha=0.5)
        ax.plot_surface(x_m, y_m, g, color='steelblue', zorder=1000, alpha=0.3)
        ax.plot(mesh_points, ys, zs=0, zdir='z', color='darkorange', lw=4)
    if gradients:
        ax.quiver(sol.x[0], sol.x[1], objective(sol.x), *grad_obj(sol.x), □
→objective(sol.x)+1, \
            length=0.15, normalize=False, alpha=0.7)
        ax.quiver(sol.x[0], sol.x[1], np.amin(z)-0.99, *grad_obj(sol.x), 0, \
            length=0.15, normalize=False, alpha=0.7)
        ax.quiver(sol.x[0], sol.x[1], constr(sol.x), *grad_con(sol.x), □
→constr(sol.x), \
            length=0.15, normalize=False, color='darkorange', alpha=0.7)
        ax.quiver(sol.x[0], sol.x[1], np.amin(z)-0.99, *grad_con(sol.x), 0, \
            length=0.15, normalize=False, color='darkorange', alpha=0.7)
    if solution:
        ax.scatter(sol.x[0], sol.x[1], objective(sol.x), zorder=2000, □
→color='red')
        ax.scatter(sol.x[0], sol.x[1], zs=np.amin(z)-1, color='red')

plt.savefig("img/plot_lagrange_multi.png")

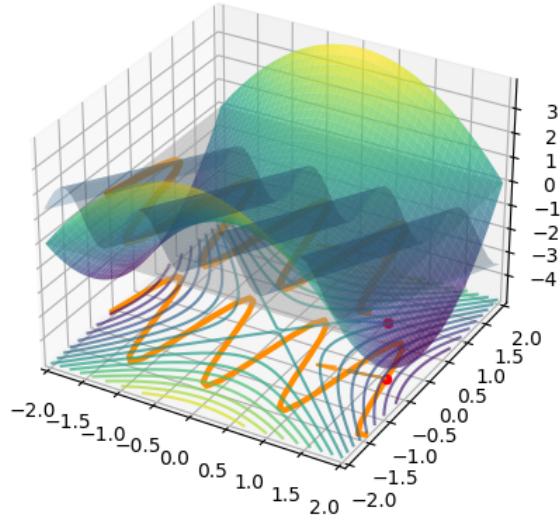
print(sol.x, objective(sol.x), constr(sol.x))
interact(plot_lagrange_multi, \
    surface=True, \
    contour=True, \
    constraint=True, \
    gradients=True, \
    solution=True)

```

[1.65082188 -0.28191999] -2.6457340021830453 -1.946231897864692e-08

interactive(children=(Checkbox(value=True, description='surface'), Checkbox(value=True, descrip

<function __main__.plot_lagrange_multi(surface, contour, constraint, gradients, solution)>



In the first state of this plot, you will see the constraint surface plotted in `steelblue`. From this surface, only those points are of interest which intersect the zero plane perpendicular to the z-axis, colored `lightgray` here. The intersection curve is plotted in `darkorange`. To make things more clear, this curve is projected onto the bottom of the graph. Here, you also see the contour plot of the surface of the objective function. To turn on the plot of the objective function, click on `surface`. In this case, it's a saddle. Feel free to test this with other objective functions, although the gradient is not yet calculated analytically, you'll have to define it yourself (same goes for the constraint function).

To see the solution that `scipy` found, click on `solution` and maybe turn off everything else except for the contour plot. What makes this point so special? You'll see this when turning on the `gradients` of both functions. This is the only point where they are `colinear`, and that is exactly what constraint optimization does - finding the optimal point for the objective function at a place, where the gradient is colinear with the gradient of the constraint function.

3.8 Numerical Optimization

So far, we only determined extremal points analytically, by evaluating where the gradient of some function had its zeros. This requires an explicit expression for the objective function, expressed by its *features*, or variables, like $J(\mathbf{x}) = \sin(x_1) + x_2^2 + \dots$. In the majority of cases, this is absolutely impossible. The most obvious reason will be that our feature space, or the number of variables, will be extremely large (think millions), such that we cannot write down the complete objective function at all. Hence, we will work with the objective function, or *loss function*, like with a black box. Something goes in, something comes out. To get the gradient of a function like this, numerical methods are mandatory.

One way to do this is by using *iterative optimization*. The first step here is to *guess* some initial $\mathbf{x}^{(0)}$, evaluate $J(\mathbf{x}^{(0)})$, find the gradient $\nabla_{\mathbf{x}}$, stop if the gradient vanishes or make a new guess if it doesn't. Better than simple guessing is to improve the guess by some technique. Calculating the gradient could be done simply by using the *finite difference method*:

$$\frac{\partial J}{\partial x_i} = \frac{J(x_1, x_2, \dots, x_i + \Delta x_i, \dots, x_n) - J(x_1, x_2, \dots, x_i, \dots, x_n)}{\Delta x_i} \quad (32)$$

which is just the definition of the partial derivative, without taking the limit. This isn't really feasible, since input vectors will often be very large, say 3600 elements for a 60x60 pixel grayscale image. The finite difference method would have to be applied 3600 times. If J can be expressed analytically and if it is a composition of functions, as in $J(g(h(\dots)))$, then a technique called **automatic differentiation** can be used, which is a computationally cheap numerical technique. We will look at this later when talking about *neural networks*, where this technique is known as **backpropagation**.

The most common and important example for an iterative technique is:

3.8.1 Gradient Descent

We've seen the gradient on contour plots before. In the following plot, some more complicated loss function was chosen and its contours plotted. The idea here is to randomly choose one of the contour lines, get the gradient at that point and move into the *opposite* direction. This relies on the fact that the gradient always points in the direction of *steepest ascent*, which we have seen in the first lesson.

In the scalar case, the algorithm for gradient descent is

$$x^{(k+1)} = x^{(k)} - \alpha \frac{\partial}{\partial x} J(x^{(k)}) \quad (33)$$

where α is the **learning rate** (or **step size** in a more general context). This is a value chosen by the user of this algorithm *before* starting it, which is an example of a **hyperparameter**. We will discuss these types of parameters in more detail later in the course.

```
%matplotlib inline
import sympy as sy
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact#, interactive, fixed, interact_manual
# import ipywidgets as widgets

def plot_arrow(ax, origin, target):
    origin = np.array(origin, dtype=np.float)
    target = np.array(target, dtype=np.float)
    ax.arrow(*origin, *(target-origin), head_width=0.2, head_length=0.2, \
             fc='darkorange', ec='darkorange', lw=3, \
             length_includes_head=True, \
             zorder=10, label="Velocity vector")
```

```

def plot_gd(ax, , function, start, n_steps, x_lims):
    ax.cla()
    points = [start] # this will be filled with more points later
    f_prime = function.diff(x,1)

    x_range = np.linspace(x_lims[0], x_lims[1], 30)

    y = np.array([function.subs(x,x_val) for x_val in x_range])
    y_lims = [float(np.amin(y))-0.5, float(np.amax(y))]

    ax.plot(x_range, y, lw=3, alpha=0.7)

    for i in range(n_steps):
        next_point = points[-1] - *f_prime.subs(x,points[-1]).evalf()
        # plot position in current step
        ax.scatter(points[-1],function.subs(x,points[-1]).evalf(), color='red', ▾
        ↳lw=4, zorder=20)
        # plot distance arrow
        plot_arrow(ax, [points[-1],function.subs(x,points[-1]).evalf()], \
                   [next_point,function.subs(x,next_point).evalf()])
        # append new point to "points" array
        points.append(next_point)

    ax.set_xlim(x_lims)
    ax.set_ylim(y_lims)
    ax.grid(True)

x = sy.Symbol('x')
f1 = sy.sqrt(x**2)
f2 = x**2

def plot_grads( , starting_point, n_steps):
    fig = plt.figure(figsize=(12,7))
    ax1 = fig.add_subplot(121, aspect='equal')
    ax2 = fig.add_subplot(122, aspect='equal')

    plot_gd(ax1, , f1, starting_point, n_steps, [-3, 3])
    plot_gd(ax2, , f2, starting_point, n_steps, [-1.8, 1.8])

    plt.savefig("img/plot_grads.png")

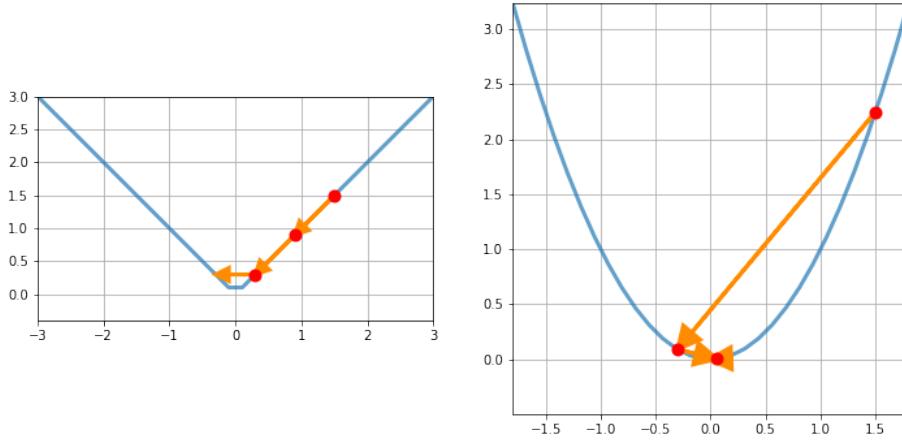
interact(plot_grads, \
         =(0.1, 1.1, 0.1), \

```

```
starting_point=(0.5, 2.5, 0.1), \
n_steps=(1, 6))
```

```
interactive(children=(FloatSlider(value=0.6, description=' ', max=1.1, min=0.1), FloatSlider(va
```

```
<function __main__.plot_grads( , starting_point, n_steps)>
```



We see that on the left plot, the gradient has constant value. This easily results in **overshooting** the minimum we would like to find. On the right side, the gradient becomes smaller when getting closer to the minimum, and larger when leaving it. This situation is more stable, because being further away shoots you closer to the minimum, while when being close to the minimum, only small steps are taken.

In case \mathbf{x} is a vector, the algorithm above is simply applied to each component:

$$x_i^{(k+1)} = x_i^{(k)} - \alpha \frac{\partial}{\partial x_i} J(x^{(k)}) \quad (34)$$

or, expressed as a vector equation:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \alpha \nabla_{\mathbf{x}} J(x^{(k)}) \quad (35)$$

You will take a closer look at what gradient descent does in this case in the assignment, but let's quickly look at an example (you don't need to understand the code):

```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d
import matplotlib.pyplot as plt
from matplotlib import cm
from sympy.vector import gradient, CoordSys3D

x, y = sy.symbols('x, y')
X = [x, y]

f = sy.exp(-0.05*x**2 -0.05*y**2) * sy.sin(x) * sy.cos(y)
f_grad = sy.derive_by_array(-f, X) # the "--" is introduced here already

print("Function: f(x,y,z) = ", f)
print("Gradient: f'(x,y,z) = ", f_grad)

func = sy.lambdify([x, y], f, "numpy")
func_grad = sy.lambdify([x, y], f_grad, "numpy")

X,Y = np.meshgrid(np.arange(-5,5,0.1), \
                  np.arange(-5,5,0.1))
Z = func(X,Y)

def plot_grad_landscape( , surface, contours, start_p, next_p, snext_p):
    fig = plt.figure(figsize=(9,8))
    ax = fig.add_subplot(111, projection='3d')

    #starting_point = np.array([1, -0.5])
    starting_point = np.array([-0.5, -0.5])
    next_point = starting_point + np.array([]) * func_grad(*starting_point)
    s_next_point = next_point + np.array([]) * func_grad(*next_point) # "+", ↵
    ↵because grad is "--" here

    plt.cla()
    if surface:
        ax.plot_surface(X, Y, Z, rstride=8, cstride=8, alpha=0.5, cmap=cm.ocean)

    if contours:
        ax.contour(X, Y, Z, zdir='z', offset=np.min(Z)-1, cmap=cm.ocean, ↵
        ↵levels=15)
        #ax.contourf(X, Y, Z, zdir='x', offset=-5, cmap=cm.ocean)
        #ax.contourf(X, Y, Z, zdir='y', offset=5, cmap=cm.ocean)

    if start_p:
```

```

        ax.scatter(*starting_point, np.min(Z)-1, color='red', lw=4)
        ax.scatter(*starting_point, func(*starting_point)+0.1, color='red', □
→lw=4, zorder=10000)
        ax.quiver(*starting_point, np.min(Z)-1,
                  *func_grad(*starting_point), 0, \
                  length= *np.linalg.norm(func_grad(*starting_point)), lw=3, □
→color='red', normalize=False, alpha=1)
        ax.quiver(*starting_point, func(*starting_point)+0.1,
                  *func_grad(*starting_point), □
→func(*func_grad(*starting_point)), \
                  length= *np.linalg.norm(func_grad(*starting_point)), lw=3, □
→color='darkorange', normalize=False, alpha=1)
        print("starting f = ", func(*starting_point))
        print("starting f_grad = ", func_grad(*starting_point))

    if next_p:
        ax.scatter(*next_point, np.min(Z)-1, color='red', lw=4)
        ax.scatter(*next_point, func(*next_point)+0.1, color='red', lw=4, □
→zorder=10000)
        ax.quiver(*next_point, np.min(Z)-1,
                  *func_grad(*next_point), 0, \
                  length=2* *np.linalg.norm(func_grad(*next_point)), lw=3, □
→color='red', normalize=False, alpha=1)
        ax.quiver(*next_point, func(*next_point)+0.1,
                  *func_grad(*next_point), func(*func_grad(*next_point)), \
                  length=2* *np.linalg.norm(func_grad(*next_point)), lw=3, □
→color='darkorange', normalize=False, alpha=1)
        print("next f = ", func(*next_point))
        print("next f_grad = ", func_grad(*next_point))

    if snext_p:
        ax.scatter(*s_next_point, np.min(Z)-1, color='red', lw=4)
        ax.scatter(*s_next_point, func(*s_next_point)+0.1, color='red', lw=4, □
→zorder=10000)
        ax.quiver(*s_next_point, np.min(Z)-1,
                  *func_grad(*s_next_point), 0, \
                  length=2* *np.linalg.norm(func_grad(*s_next_point)), lw=3, □
→color='red', normalize=False, alpha=1)
        ax.quiver(*s_next_point, func(*s_next_point)+0.1,
                  *func_grad(*s_next_point), func(*func_grad(*s_next_point)), \
                  length=2* *np.linalg.norm(func_grad(*s_next_point)), lw=3, □
→color='darkorange', normalize=False, alpha=1)
        print("second f = ", func(*s_next_point))
        print("second f_grad = ", func_grad(*s_next_point))

#fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)

```

```

ax.set_xlabel('X')
ax.set_xlim(-5, 5)
ax.set_ylabel('Y')
ax.set_ylim(-5, 5)
ax.set_zlabel('Z')
ax.set_zlim(np.min(Z)-1, np.max(Z))

plt.savefig("img/plot_grad_landscape.png")

interact(plot_grad_landscape, \
         surface=True, \
         contours=True, \
         start_p=True, \
         next_p=True, \
         snext_p=True, \
         =(0.1, 5.0, 0.1))

```

Function: $f(x,y,z) = \exp(-0.05*x^2 - 0.05*y^2)*\sin(x)*\cos(y)$
 Gradient: $f'(x,y,z) = [0.1*x*\exp(-0.05*x^2 - 0.05*y^2)*\sin(x)*\cos(y) - \exp(-0.05*x^2 - 0.05*y^2)*\cos(x)*\cos(y), 0.1*y*\exp(-0.05*x^2 - 0.05*y^2)*\sin(x)*\cos(y) + \exp(-0.05*x^2 - 0.05*y^2)*\sin(x)*\sin(y)]$

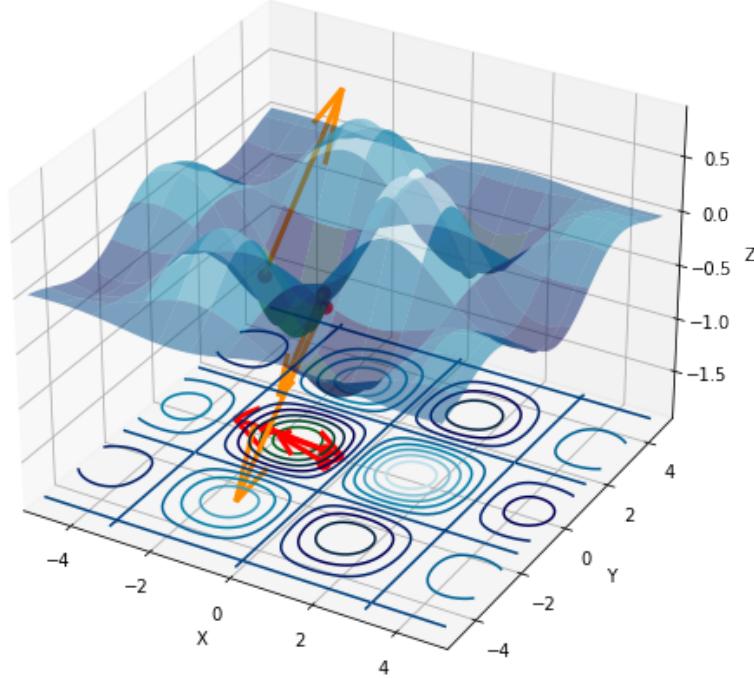
```

interactive(children=(FloatSlider(value=2.5, description=' ', max=5.0, min=0.1),
```

```

<function __main__.plot_grad_landscape( , surface, contours, start_p, next_p,
snext_p)>

```



3.8.2 Stopping Criteria

Due to machine precision and sometimes suboptimal loss landscapes, the algorithm won't ever terminate if the only criterion to stop is getting the gradient to exactly zero. In practice, one would decide on a sensible **error limit**, such as $\epsilon = 1e - 6 = 10^{-6}$, which would mean 6 decimal places of precision on results. For setting a stop criterion, there are multiple options available. One could say that convergence is reached when $\|x^{(k+1)} - x^{(k)}\| \leq \epsilon$, which would be based on the *closeness* of two vectors. One could also use $\|J(x^{(k+1)}) - J(x^{(k)})\| \leq \epsilon$ or $\|\nabla_x J(x^{(k)})\| \leq \epsilon$.

For tracking the convergence of such a process, a **convergence graph** is used, which plots the error criterion for each iteration. This should be familiar to you from FEM, where such graphs show that, or if at all, the analysis converged. There are different kinds of convergences (quadratic, linear, ...), giving a notion of how well some algorithm gets to the solution. In FEM, an example of something you could plot would be the norm of the displacements in the current iteration of a Newton-Raphson solution scheme.

The graph above shows such a convergence graph. In this case, the *cumulative error* over the whole *test set* was plotted against the number of *training samples* used for training a neural network as a

surrogate model for a complex TPM multiscale-multiphase simulation. The training was performed multiple times to get some statistics on the stability of the process. The gray margins are the *standard deviation* of the cumulative error for these runs. These graphs help to understand how many samples you really need for a model to be useful, and where the tradeoff for time/resources consumed and applicability of the model starts becoming unrewarding. We will talk about these things in more detail in the lecture on evaluation of ML models.

To summarize; when using such an algorithm one has to decide on a learning rate α and an error limit ϵ , both of which are *hyperparameters*, before starting the algorithm. Then, an initial guess $x^{(0)}$ is made, starting the gradient descent that is (numerically) calculated as $x^{(k+1)} = x^{(k)} - \alpha \nabla_x J$. The algorithm stops if it found a zero of the gradient ($x^{(k+1)} = x^{(k)}$), or if the difference of $x^{(k+1)}$ and $x^{(k)}$ is below the error threshold ϵ . Otherwise, the gradient step is repeated.

4 Statistics

4.1 Probability

Probability theory presents a mathematical framework for representing **uncertainty**. We will encounter uncertainty in experimental as well as simulation data, and in both again in input data as well as output data. This whole lesson is only a short introduction to the presented concepts, an in-depth analysis could fill whole lecture series.

There are various sources for uncertainty in data.

The system under consideration could be **inherently random** (Brownian motion/Langevin dynamics, quantum mechanics, ...), such that a whole probabilistic framework is needed to formulate models in these circumstances.

Regarding modelling, often one tries to model only the most important aspects of a system, thus disregarding many of its properties in an **incomplete model**. In other cases, a complete model of underlying physics might be computationally impossible. This happens for example in multiscale systems, where a gigantic microscopic state space necessitates a phenomenological macroscopic description. Take for example a model of water flow; 1 cm^3 of water contains roughly $6 \cdot 10^{23}$ particles. Using current computer technology, tracking all particle trajectories is impossible, so a macroscopic description is the only thing that can be done here. The same is true for 1 cm^3 of solid material. The **material models** describing the behavior of such a cube under load could, in principle, be derived by the fundamental laws describing solid state physics for each particle in the cube and their geometric arrangement. Since this is only possible for minimal amounts of matter, we are stuck with using phenomenological models giving an inherently inaccurate description of reality (*side note: all theories of physics are incomplete and probably will be so forever*).

The greatest source for uncertainty is **incomplete data** or **incomplete observability** of parts of a system. From the same argument as above, you can never observe the complete microscopic state of a system of sufficient size, like tracking the molecules of a gas.

We will need these notions mainly in two parts of ML:

- **constructing** learned models mimicking uncertain human reasoning, e.g., say there's a 5% that an aircraft engine is going to fail in the next two flights, which probabilistic models and results can you derive from this information?
- **evaluating** learned models. None of the models we will get to know will output a correct result with complete certainty, so we need a way to analyze and quantify their behavior (say, a model that is supposed to identify cracks from images of plane hulls, it would output "I'm 90% sure this is a crack")

4.1.1 Definitions

We need to introduce a few terms for the mathematical treatment of probability and statistics:

- **Random Experiment**: an experiment with different outcomes despite same/similar conditions
- **Random Variables**: result of a random experiment. When the result is categorical (a "word"), there are various ways to assign numbers to these categories instead (e.g. 1-hot

encoding -> later). RVs are denoted by capital letters, e.g. X , whereas the value an RV takes is denoted by a small letter, e.g. x .

- **Sample Space:** the set of all possible results (states) of a random experiment.
- **probability Distribution:** a function that quantifies how likely an RV is to take a possible state. There is a distinction:
 - **Probability Mass Function** for discrete sample spaces, “table” of probabilities for each state
 - **Probability Density Function** for continuous sample spaces, probability per unit “length”

4.1.2 Discrete Sample Space

A simple example for a random experiment with a discrete sample space is a coin toss. The sample space here would be $S = \{h, t\}$, where the RV S (for *side*) here can take on the values h for heads, and t for tails. If the coin is unbiased and always falls on one of its sides, the results of consecutive trials follow a **uniform distribution**. Hence, the PMF is $P(S) = \frac{1}{N}$, where N is the number of states in the state space.

Mathematically, a PMF must be injective (its domain must contain all values in the state space), such that the sample space is completely covered. It must give a value $0 \leq P(X = x) \leq 1$ and sum to $\sum_{x \in X} P(X = x) = 1$ (some outcome from the sample space *must* happen).

Below is an simple implementation of this example. **Probability** is sometimes also defined via the law of large numbers, that says that the number a certain result occurred divided by the number of trials T approaches the probability of that result for large, but *finite* T . One would expect the histogram below to show two equally sized bins for large T , but for small T , you will often get a skewed distribution of outcomes.

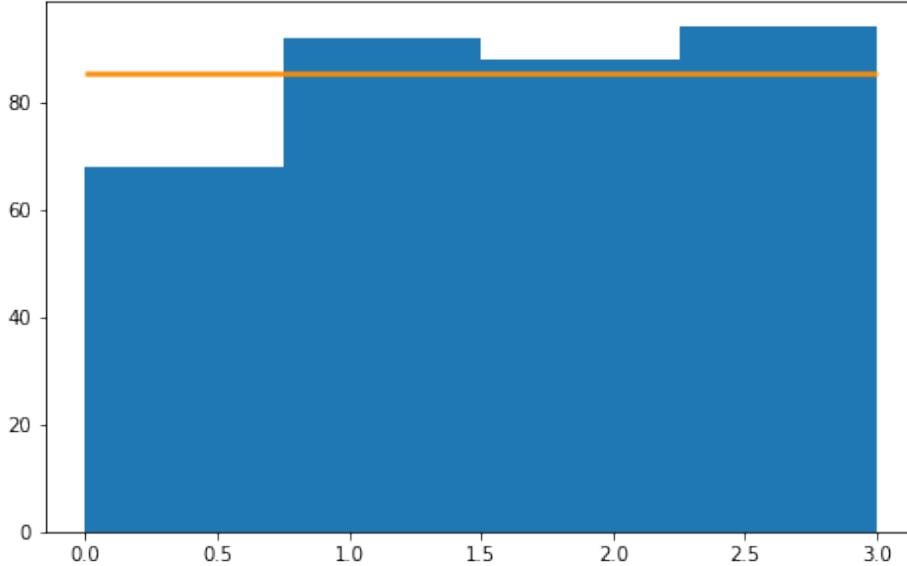
```
from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets
import matplotlib.pyplot as plt
from random import randint

def plot_histogram(sample_space_size, trials):
    results = [randint(0, sample_space_size-1) for trial in range(trials)]

    plt.figure(figsize=(8,5))
    plt.hist(results, bins = sample_space_size)
    plt.hlines(trials/sample_space_size, 0, sample_space_size-1, color='darkorange', lw=2.5)
    #plt.show()
    plt.savefig("img/plot_histogram.png")

interact(plot_histogram, sample_space_size=widgets.IntSlider(min=2, max=10, value=2, continuous_update=False), trials=widgets.IntSlider(min=2, max=500, value=5, continuous_update=False))
plt.close()
```

```
interactive(children=(IntSlider(value=2, continuous_update=False, description='sample_space_size')))
```



4.1.3 Continuous Sample Space

An example for a random experiment with a continuous sample space could be manufacturing steel beams. For a customer, the length of these must fall into a desired interval, say $0.95 \text{ m} < L < 1.05 \text{ m}$, so the RV here is L . The sample space is continuous this time and encompasses all possible beam lengths that can be produced. Asking for the probability of beams coming out with a length in this interval would be calculated as

$$P(0.95 < L < 1.05) = P(0.95 \leq L \leq 1.05) = \int_{0.95}^{1.05} p(l)dl \quad (36)$$

where the first equality is true, because the probability for an exact value is zero. The explanation for why becomes clear in the mathematical criteria for a PDF, since probability is the area, and the area of a line is zero.

Mathematically, a PDF must also be injective, such that $\forall x \in X : 0 \leq p(x)$, but $p(x)$ must not necessarily be smaller than one. The reason is that the probability P here is the area under a portion of the graph of $p(x)$, but see for yourself:

```
def plot_bin(width):
    results = [width*randint(0, 1) for i in range(200)]
    plt.hist(results, bins=1, density=True)
```

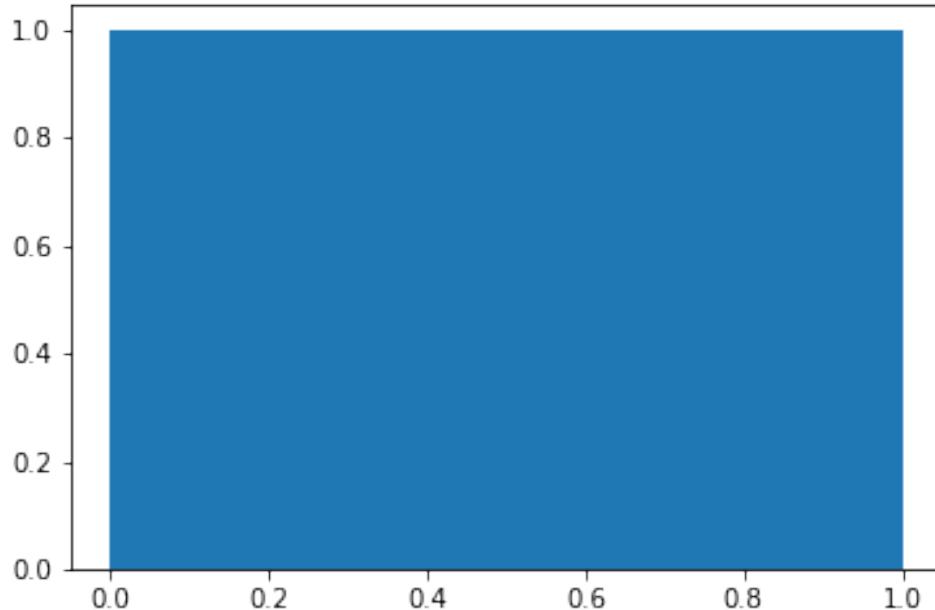
```

plt.savefig("img/plot_bin.png")

interact(plot_bin, width=widgets.FloatSlider(min=0.001, max=10, value=1, ↴continuous_update=False))
plt.close()

interactive(children=(FloatSlider(value=1.0, continuous_update=False, description='width', max=10),)

```



No matter how large you choose the interval to be, the area of this bar must always be equal to 1, since the last mathematical requirement for a PDF is that $\int_{-\infty}^{\infty} p(x)dx = 1$, which is a normalization.

Plotting the number of randomly chosen values that were in a certain interval is usually done in these bars called *bins*. Increasing the number of bins (the number of intervals) and decreasing their width (decreasing the size of an interval, i.e., creating a finer resolution) will converge to the underlying statistical distribution.

In the visualization below, you can adjust the number of `bins` and the number of `samples` separately. If your number of samples is lower than the number of bins, there will be empty spots in the graph (due to empty bins), so this has to be chosen sufficiently high to see the actual distribution. The example below is a beta-distribution.

```

from numpy.random import beta
from numpy import linspace
import scipy.stats as stats

```

```

def plot_beta(bins, samples):
    results = beta(2, 5, samples)

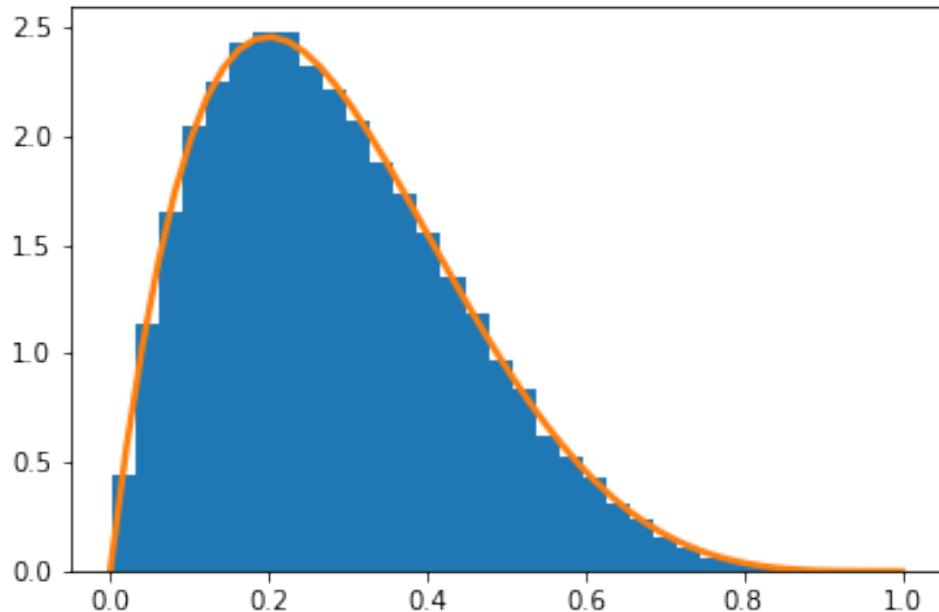
    x = linspace(0, 1)
    plot_points = stats.beta.pdf(x, 2, 5)

    plt.hist(results, bins=bins, density=True)
    plt.plot(x, plot_points, lw=2.5)
    plt.savefig("img/plot_beta.png")

interact(plot_beta, bins=widgets.IntSlider(min=10, max=100, value=10, continuous_update=False), samples=widgets.IntSlider(min=100, max=100000, value=100, continuous_update=False))
plt.close()

interactive(children=(IntSlider(value=10, continuous_update=False, description='bins', min=10)

```

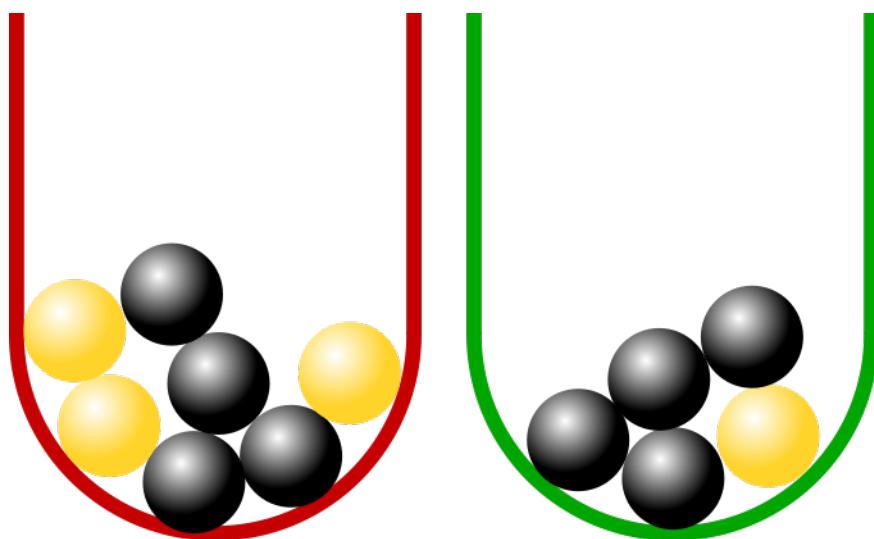


Remember the definition of the Riemann integral, where a sum is transformed into an integral as a limiting process by converting the summands into densities and introducing an infinitesimal factor. This should give you a sense for the correspondence between these examples and especially the definition of the PMF and PDF. This is the same correspondence that transforms point loads into line loads.

4.2 Conditional, Joint, and Marginal Probability

These concepts are explained in the easiest way with an example. In introductory texts to statistics, this is often visualized by urns U_i containing differently colored balls B_j . Let's say we have two urns, 1 and 2. Urn 1 contains 7 balls, of which 4 are black and 3 are golden. Urn 2 contains 5 balls, of which 4 are black and 1 is golden.

```
from IPython.display import Image
Image("img/urns.png", width="700")
```



The RVs here are $U = 1, 2$ and $B = b, g$. The random experiment is picking one of the balls randomly, so all balls are equally likely to be chosen. Apart from asking what the probability is for choosing a certain color, some conditional questions also make sense now. For example, when you pick a ball and it turns out to be black, what is the probability that it came from urn 1? We can make sense of this question with a probability table. To make things more interesting, let's say picking the urns isn't uniform, but urn 1 has a chance of $P(U = 1) = 0.4$ to be picked, and urn 2 has a chance of $P(U = 2) = 0.6$ to be picked (you don't need to understand the code, it's a bit hacky):

```
from random import choice
import numpy as np

class Urn:
    def __init__(self, dict_of_balls):
        self.balls = dict_of_balls
        self.total = sum(dict_of_balls.values())
        self.probs = {key : value/self.total \
                     for key, value in dict_of_balls.items()}
        self.cop = 1.0
```

```

def set_chance_of_picking(self, chance):
    self.cop = chance

def get_chance_of_picking(self):
    return self.cop

def pick_random_ball(self):
    return choice(self.balls)

def get_balls(self):
    return self.balls.keys()

def get_prob(self, ball):
    return self.probs[ball]

def get_amount(self, ball):
    return self.balls[ball]

## change problem setup here
trials = 1
u = [ \
    {"black" : 4, "golden" : 3}, \
    {"black" : 4, "golden" : 1} \
]

# enter a chance for each urn to be picked, sum must equal 1
chance_of_picking = [0.4, 0.6]
##

urns = [Urn(urn) for urn in u]
for num, chance in enumerate(chance_of_picking):
    urns[num].set_chance_of_picking(chance)
balls = [*urns[0].get_balls()]

prob_matrix = np.zeros([len(chance_of_picking), len(balls)])
for u, urn in enumerate(urns):
    for b, ball in enumerate(balls):
        prob_matrix[u,b] = trials * urn.get_chance_of_picking() * urn.
        ↪get_prob(ball)

print("\t\t#### probability table ####")
print()
print("      | ")
for ball in balls:
    print("B =", ball, " | ", end=" ")

```

```

print("Total      ")
for u,urn in enumerate(urns):
    print("\t" + len(balls)*"-----" + " | ")
    print("U =", u+1, " | ", end=" ")
    for b,ball in enumerate(balls):
        print(" ", round(prob_matrix[u,b],2), " ", end=" | ")
    print(round(np.sum(prob_matrix, axis=1)[u],2))
print("\t" + len(balls)*"-----" + " | ")
print("Total", end="")
for r in range(prob_matrix.shape[1]):
    print("      ", round(np.sum(prob_matrix, axis=0)[r],2), " ", end="")

```

probability table

	B = black	B = golden	Total
U = 1	0.23	0.17	0.4
U = 2	0.48	0.12	0.6
Total	0.71	0.29	

The probability of choosing a black ball can be seen in the table in the *margin* of the first column. To answer the question of how likely it is that the ball came from urn 1 if it turned out to be black, we'd have to look in row 1, column 1 of the table. These are the **joint probabilities** p_{ij} , which in general describe the probability that some RV X will take the value x_i AND some RV Y will take the value y_j . Regarding the above example, a joint probability would be $P(U = 1, B = b) \approx 0.23$.

To generalize this notion to N urns and M differently colored balls, the table would be an $N \times M$ -matrix with entries $p_{ij} = P(X = x_i, Y = y_j)$. More RVs would create a higher dimensional table with entries $p_{ijk\dots}$.

Let's get back to the question what the probability for getting a black ball or having chosen urn 1 is. These correspond to the sum of their respective row or column and are denoted in the margin. Hence, they're called **marginal probabilities** r_j or c_i . The fact that the values in a row/column are added to get these is also called the **sum rule**; $c_i = \sum_j p_{ij}$ with $P(X = x_i) = c_i = \sum_j p_{ij}$. Calculating these is sometimes called **marginalization**. P is the marginal probability.

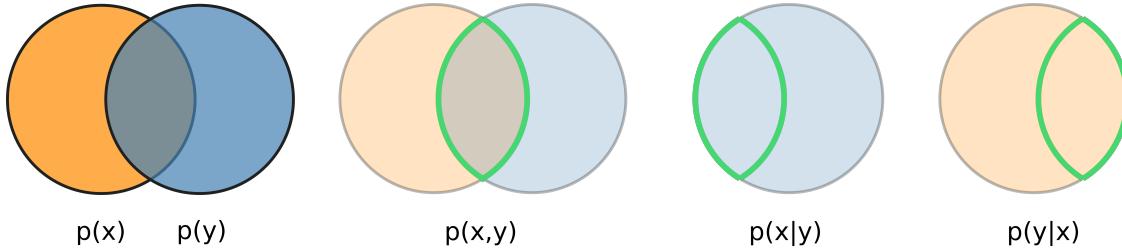
Coming back to the question from above, what the chances are that our ball came from urn 1, given we picked a black ball. This is denoted as $P(U = 1|B = b)$ and called **conditional probability**. Think back of the example with the images of cracks. The application here would be asking “given the values of pixels in an image, what is the probability that they depict a crack?”. Here, $P(U = 1|B = b) \approx \frac{0.23}{0.71} \approx 0.32$ from the table above. Generally, this is calculated by $P(Y = y_j|X = x_i) = \frac{p_{ij}}{c_i}$. Looking at this in reverse gives:

$$p_{ij} = \frac{p_{ij}}{c_i} c_i = P(Y = y_j|X = x_i) \cdot P(X = x_i) \quad (37)$$

This is the conditional probability times the marginal probability, also called the **product rule**. Some say the symbol “|” represents a division symbol.

The different kinds of probabilities can be visualized by *Venn diagrams*:

```
Image("img/venn_probability.png", width="800")
```



They all have a geometrical interpretation.

4.2.1 Bayes' Theorem

Last thing we need is **Bayes' theorem**. Thinking of the product rule, we similarly get $P(Y, X) = P(X|Y) \cdot P(Y)$ by symmetry. Also $P(X, Y) = P(Y, X)$. From this, we get that

$$P(Y|X)P(X) = P(X|Y)P(Y) \implies P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \quad (38)$$

This is an important relation that sometimes leads to very counterintuitive results. We will look at an example in the next part. For a great explanation and visualization, see [this](#) video from 3blue1brown.

4.3 Bayes' Theorem

4.3.1 Prior Probability

Let's think about the example with the urns from before again. Say we pick a ball at random and ask how likely it is, that it came from the 2nd urn. The probabilities for the urns in the last lesson were $P(U = 1) = 0.4$ and $P(U = 2) = 0.6$. So without looking at the color of the ball we picked, the probability is 0.6. Since this is calculated *before* looking at the retrieved information, this is called the **prior probability**.

4.3.2 Posterior Probability

Looking at the ball and seeing that it is golden, this probability changes, since we have more information available now. Compare this to an example. Someone tells you that there is a plane flying in the air and asks you, what are the chances that it is from Lufthansa? To get an answer, you might compare fleet sizes of different companies and how many of them are in the air at a certain time and you could get an approximate probability from that. Now say that you get another piece of information, namely that the plane was in the air over Frankfurt (Main). Now the chances shift significantly. Coming back to the question above, using Bayes theorem, the answer is

$$P(U = 2|B = g) = P(B = g|U = 2) \cdot P(U = 2) / P(B = g) \approx 0.12/0.6 \cdot 0.6 / 0.29 \approx 0.41. \quad (39)$$

Since this is a probability we got using the information we retrieved, it's called the **posterior probability**.

An interesting application of these concepts is to use live information on rescue mission. Suppose you detect a distress signal giving you a rough location. You can (using domain knowledge of signal technology) apply probabilities to each square in a grid around the area where you detected the signal and start your search on a high-probability square, see that the source is not to be found there, update the probabilities of surrounding squares and continue your search in an optimal way. [Here](#) is a good video explanation of this application, using information from [informs.org](#)

The last definition we need here is that of **likelihood**. Notice how the posterior probability is proportional to the prior probability:

$$P(Y|X) \propto P(X|Y) \cdot P(Y) \quad (40)$$

The proportionality factor $P(X|Y)$ here is called the likelihood.

4.3.3 Example

Assume someone developed a test for seeing whether an aircraft engine would fail on the next flight. How can we quantify its accuracy? Usually, the accuracy is the number of engines that are defective and correctly identified as being defective by the test. Let's just assume the accuracy here is 99%, so 99/100 defective engines are correctly identified. These are called the **true positives**. Engines that are functional are also identified by the test with 99% accuracy, these are called the **true negatives**.

This seems like a good value, but we should analyze the other probabilities we are left with. The random variables are $T = \{+, -\}$ and $E = \{d, f\}$, denoting the test result and whether an engine is defective or functional. In these types of problems, there is often a huge **bias** in the sample population. Say only 0.5% of engines being tested are defective. We are given that $P(+|d) = 0.99$, $P(-|f) = 0.99$ and $P(d) = 0.005$ (without test results) and their complements. What is the probability that an engine that was tested positive is really defective?

$$P(d|+) = \frac{P(+|d)P(d)}{P(+)} = \frac{P(+|d)P(d)}{P(+|d)P(d) + P(+|f)P(f)} \quad (41)$$

$$P(d|+) = \frac{0.99 \cdot 0.005}{0.99 \cdot 0.005 + 0.01 \cdot 0.995} \approx 0.33 \quad (42)$$

This is quite low! The reason is the low **prior**. Suppose there are 10000 engines to be tested, of which 50 are defective. 99% will be detected, so roughly 50 of them will test positive, but 9950 engines are functional. Out of them, $0.01 \cdot 9950 \approx 100$ test positive. So we have 50 correctly identified defective engines vs. 100 functional engines, that were incorrectly tested positive, called **false positives**. Out of 150 positive results, only 50 were correct, which gives the 0.33 result above. The 100 false positives are so large, because $P(d) = 0.005$ is so low. This is an example of **sample bias** and tuning the priors will be important for us later on.

```
import numpy as np
import seaborn as sn
```

```

import pandas as pd

base_population = 10000
defect_rate = 0.005
defective = int(base_population*defect_rate)
effectivity = 0.99

ground_truth = np.zeros(base_population)
for i in range(defective):
    ground_truth[i] = 1

predictions = np.zeros(base_population)
for i in range(1,1+int((1-effectivity)*(base_population-defective)+defective)):
    predictions[i] = 1

data = np.array([ground_truth,
                 predictions]).T

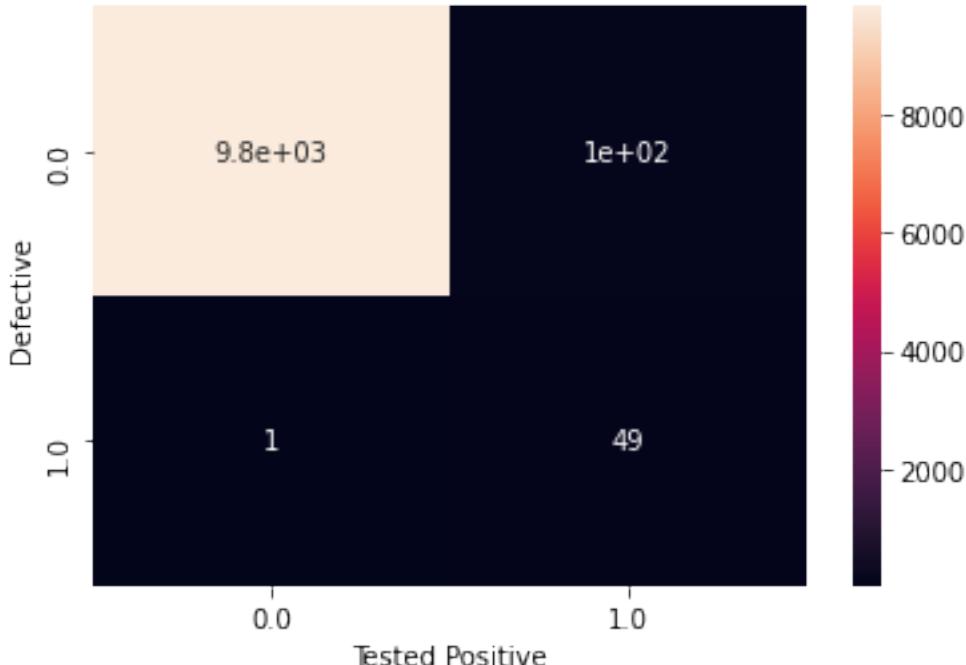
df = pd.DataFrame(data, columns=['Defective', 'Tested Positive'])

confusion_matrix = pd.crosstab(df['Defective'], df['Tested Positive'])

sn.heatmap(confusion_matrix, annot=True)

```

<AxesSubplot:xlabel='Tested Positive', ylabel='Defective'>



4.4 (Conditional) Independence and the Chain Rule of Probability

Two RVs are statistically independent iff[sic!] $P(X, Y) = P(X)P(Y)$, or more rigorously

$$P(X = x, Y = y) = P(X = x)P(Y = y) \quad \forall x \in X, y \in Y \quad (43)$$

or in other words, when the joint probability can be factorized. This has important implications later on (*side note: dependent variables resemble the behavior of entangled observables in quantum mechanics*). Thinking back of the coin toss, independence means that two tosses do not influence each other.

Let's take two coin tosses as an example. The RVs here are $T_1 = \{h_1, t_1\}$ and $T_2 = \{h_2, t_2\}$, or $T = \{hh, ht, th, tt\}$. In the latter, each result is equally likely if the tosses are independent:

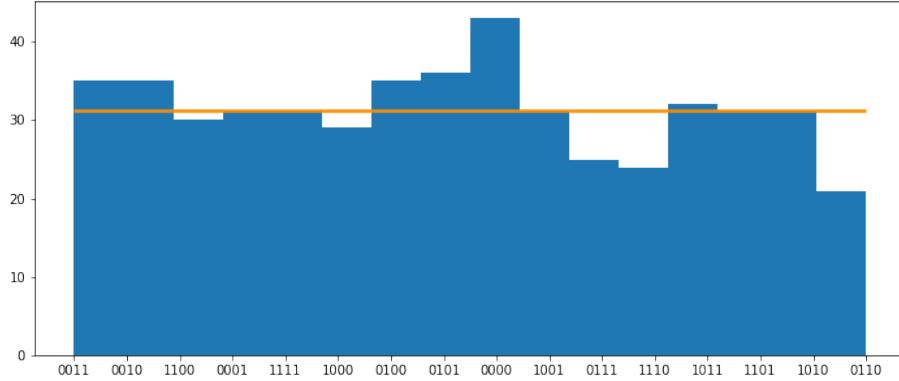
```
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
import matplotlib.pyplot as plt
from random import randint

def plot_coin_histogram(number_of_coins, trials):
    results = ["".join([str(randint(0, 1)) for noc in range(number_of_coins)])]
    for trial in range(trials):
        sample_space_size = 2**number_of_coins

        plt.figure(figsize=(12,5))
        plt.hist(results, bins = sample_space_size)
        plt.hlines(trials/sample_space_size, 0, sample_space_size-1, color='darkorange', lw=2.5)
        plt.savefig("img/plot_coin_histogram.png")

    interact(plot_coin_histogram, number_of_coins=widgets.IntSlider(min=1, max=5, value=2, continuous_update=False), trials=widgets.IntSlider(min=2, max=500, value=10, continuous_update=False))
    plt.close()

interactive(children=(IntSlider(value=2, continuous_update=False, description='number_of_coins'))
```



Note that the sample space grows exponentially.

An example for dependent variables is taking the height and weight of people. Of course there are tall, slim builds but usually, taller people weigh more than smaller people and hence, there is a **correlation** (later).

An equivalent definition of independence is $P(Y|X) = P(Y)$, or $P(X|Y) = P(X)$ which are sometimes more intuitive. Compare these to Bayes theorem and the product rule. The notation for statistically independent variables X and Y is $X \perp Y$, reminiscent of linear independence in linear algebra.

4.4.1 Conditional Independence

The RVs X and Y are independent given Z iff $P(X, Y|Z) = P(X|Z)P(Y|Z)$, or more rigorously

$$P(X = x, Y = y|Z = z) = P(X = x|Z = z)P(Y = y|Z = z) \quad \forall x \in X, y \in Y, z \in Z \quad (44)$$

or, in other words, when the probability can be factorized if a certain condition is met. Let's look at a few examples:

- Three random experiments, say X : coin toss, Y : die throw, Z : card draw. Here, X and Y are independent and conditionally independent
- X : height, Y : vocabulary size, Z : age. X and Y might seem independent at first, but take for example a person of height 1m. The probability that this person is a child is quite high and hence, vocabulary size probably low. Given a certain age, X and Y become independent.
- Two things originally independent can become dependent by a third RV, say X : die throw, Y : second die throw, Z : sum of the eyes. Suppose we fix Z . Then X and Y are dependent.

Conditional independence is denoted as $X \perp Y|Z$.

4.4.2 Chain Rule of Conditional Probability

This is an extension of the product rule to three or more variables:

$$P(X, Y, Z) = P(Z)P(Y|Z)P(X|Y, Z) \quad (45)$$

or, in general for N RVs:

$$P(X^{(1)}, X^{(2)}, \dots) = P(X^{(1)}) \prod_{n=2}^N P(X^{(n)}|X^{(1)}, X^{(2)}, \dots, X^{(n-1)}) \quad (46)$$

An example would be an image of, say 40x40 pixels, so 1600 pixels total, and asking for the probability that the pixels take on the values they do. The probability of a particular image forming can be interpreted as a joint probability. More on that comes later in the course.

4.5 Expectation

Doing a random experiment results in different outcomes regarding the RVs. The variation in outcomes is captured by the PMF or PDF, which give information on how probable it is to get a specific value or a value in a certain interval.

To quantify some properties of distributions and RVs in single numbers, summarizing statistics can be used, such as calculating the **expectation** or **variance** (next lesson) of a distribution. The expectation E gives the expectation value (or **mean** / **average**) given a certain distribution. Think for example of expected return on investment at the stock market, or expected mean directions in random walks.

The expectation of some function $f(x)$ with respect to some distribution $p(x)$ is the mean value $f(x)$ takes, when x is randomly drawn from $p(x)$ (denoted as $[x|p]$). The expectation itself is denoted as

$$E_{x \sim P}[f(x)] = \sum_x f(x)P(x) \quad (47)$$

or in the continuous case

$$E_{x \sim p}[f(x)] = \int_X f(x)p(x)dx \quad (48)$$

In ML, we deal with vectors \mathbf{x} a lot. For **multivariate expectations** of vectors, simply consider each component separately, such that

$$E_{\mathbf{x} \sim \mathbf{p}}[f(\mathbf{x})] = \begin{bmatrix} E_{x_1}[f(x_1)] \\ E_{x_2}[f(x_2)] \\ \vdots \end{bmatrix} \quad (49)$$

The expectation value of a coin toss can be calculated easily, with $X = \{0, 1\}$, $P(x) = 0.5$, such that $E_{x \sim P}[x] = \sum_x xP(x) = 0 \cdot 0.5 + 1 \cdot 0.5 = 0.5$.

What happens if we throw 2 dice and take their eye sum as the RV? Now $X = \{2, 3, \dots, 12\}$, but now the probability distribution is not uniform anymore. This can be seen by looking at the number of ways a certain sum can be reached. A 2 only has a single possibility, namely getting two

“1”s. A 7 on the other hand can be built using “1” and “6”, “2” and “5” and so on (see assignment 02). The probabilities here are

sum	2,12	3,11	4,10	5,9	6,8	7
prob	1/36	2/36	3/36	4/36	5/36	6/36

The expectation value here gives $E_{x \sim p}[x] = \dots = 7$. Multiplying and summing here is quite tedious and this gets worse with larger dice, but there is an easier way to calculate this, using the **linearity of expectation**.

The expectation operator is linear, which can be seen easily by the linearity of sums or integrals (a sum of two integrals equals the integral of the sums). Since we’re throwing two dice here, we could simply calculate the expectation value of a single die throw and multiply by two: $E[D_1 + D_2] = E[D_1] + E[D_2] = 7$.

In general, linearity here means that if $f(\mathbf{x}) = \alpha \cdot g(\mathbf{x}) + \beta \cdot h(\mathbf{x})$, then

$$E[f] = \alpha E[g] + \beta E[h] \quad (50)$$

giving a much simpler way to calculate the expectation value, since the explicit distribution $P(\mathbf{x})$ must not necessarily be calculated.

4.6 Variance and Covariance

4.6.1 Variance

As we have seen, random experiments result in different outcomes for the RVs. The average of these outcomes is captured by the expectation value, but we can also formulate a measure for the **variance** of the outcomes, which gives the variation (**fluctuation**) from the mean. Thinking back to the stock market example from last lesson, the variance here would give a measure for the *risk* of the investment (*modern portfolio theory*).

Two different distributions can have the same expectation value, but different variances. It is very important to also give at least the variance of a distribution as a measure of how sure you are that the data will be close to the expectation value. For example:

```
from ipywidgets import interact#, interactive, fixed, interact_manual
import ipywidgets as widgets
from math import sqrt
from random import gauss
from numpy import linspace
import matplotlib.pyplot as plt
import scipy.stats as stats

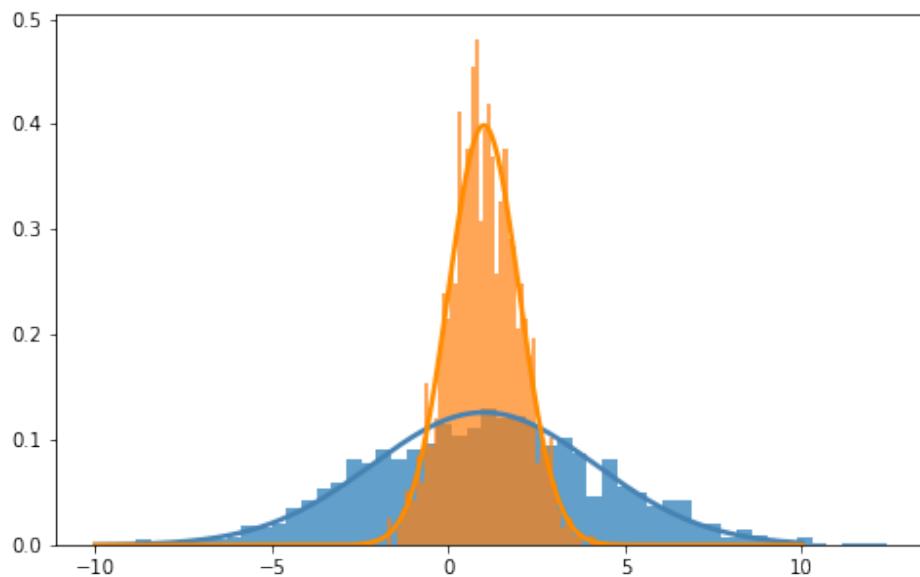
def plot_dists(mean, variance1, variance2, samples):
    x = linspace(-max(variance1, variance2), max(variance1, variance2), samples)
    data1 = [gauss(mean, sqrt(variance1)) for s in range(samples)]
    data2 = [gauss(mean, sqrt(variance2)) for s in range(samples)]
```

```
plt.figure(figsize=[8,5])
plt.hist(data1, density=True, alpha=0.7, bins=50)
plt.plot(x, stats.norm.pdf(x, loc=mean, scale=sqrt(variance1)), color="steelblue", lw=2.5)
plt.hist(data2, density=True, alpha=0.7, bins=50)
plt.plot(x, stats.norm.pdf(x, loc=mean, scale=sqrt(variance2)), color="darkorange", lw=2.5)

plt.savefig("img/plot_dists.png")

interact(plot_dists, mean=widgets.IntSlider(min=-5, max=5, value=0, continuous_update=False), \
         variance1=widgets.IntSlider(min=0, max=10, value=10, continuous_update=False), \
         variance2=widgets.IntSlider(min=0, max=10, value=1, continuous_update=False), \
         samples=widgets.IntSlider(min=10, max=1000, value=100, continuous_update=False))
plt.close()

interactive(children=(IntSlider(value=0, continuous_update=False, description='mean', max=5, m...
```



Playing around with `variance2` should change the shape of the orange distribution significantly, making it sharper or wider. Assume two manufacturer offer you steel beams of a certain size with expectation value of that size as ordered, but one of them offers you beams with a length variance of 1, while the other manufacturer offers you beams with a variance of 0.2. The best offer here still depends on pricing and actual use case, but hopefully you see how variance can give a good hint at the quality of things.

$$V_{x \sim p}[f(x)] = E_{x \sim p}[f(x) - E_{x \sim p}[f(x)]^2] = E_{x \sim p}[(f(x) - \bar{f}(x))^2] \quad (51)$$

Sometimes the **standard deviation** σ is given or necessary, as in the code block above for the distribution functions. The standard deviation is the square root of the variance and measures how much the value of some function $f(x)$ varies for various samples when x is drawn from a distribution p : $\sigma_{x \sim p}[f(x)] = \sqrt{V_{x \sim p}[f(x)]}$. The variance is often called the **mean square**, the standard deviation the **root mean square** respectively.

4.6.2 Covariance

Covariance is a generalization of variance to a pair of variables. You can think of the variance $V[x] = E[(x - \bar{x})^2]$ as a “self-covariance”. We’ll give this a meaning later. Covariance is defined as

$$C[x, y] = E[(x - \bar{x})(y - \bar{y})] \quad (52)$$

or, equivalently

$$C[x, y] = \bar{x} \cdot \bar{y} - \bar{x} \cdot \bar{y} \quad (53)$$

and analogous for $f(x)$ and $f(y)$. It quantifies how much x varies from its mean, when y varies from its own mean, or how much x and y vary together.

For vectors, this becomes a covariance matrix, as in the example below.

```
import numpy as np

samples = 400

mu = np.array([5.0, 5.0])

C1 = np.array([[4, -4], \
              [-4, 5]])

C2 = np.array([[4, 4], \
              [4, 5]])

C3 = np.array([[4, 0.1], \
              [0.1, 5]])

neg_vals = np.random.multivariate_normal(mu, C1, size=samples)
```

```

pos_vals = np.random.multivariate_normal(mu, C2, size=samples)
non_vals = np.random.multivariate_normal(mu, C3, size=samples)

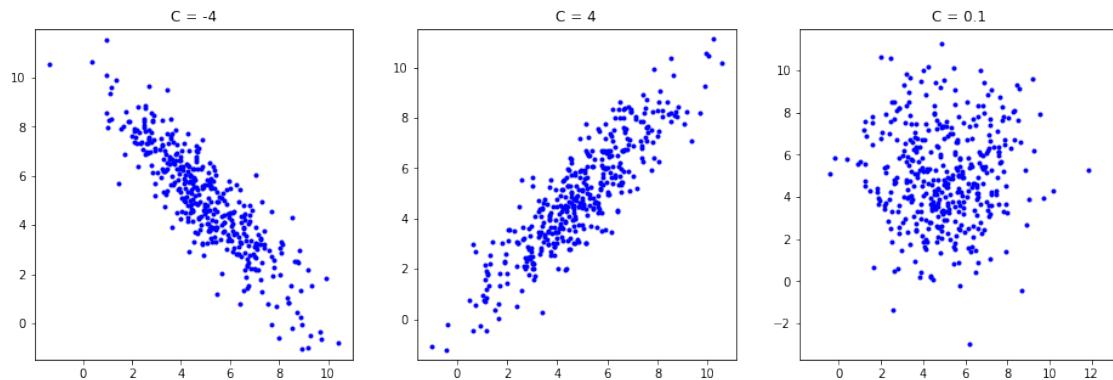
plt.figure(figsize=[16,5])
plt.subplot(131)
plt.plot(neg_vals[:,0], neg_vals[:,1], 'b.')
plt.title("C = " + str(C1[0,1]))
plt.axis('equal')

plt.subplot(132)
plt.plot(pos_vals[:,0], pos_vals[:,1], 'b.')
plt.title("C = " + str(C2[0,1]))
plt.axis('equal')

plt.subplot(133)
plt.plot(non_vals[:,0], non_vals[:,1], 'b.')
plt.title("C = " + str(C3[0,1]))
plt.axis('equal')

plt.show()

```



It's difficult, if not impossible, to make sense of the numbers the covariance gives us here. A much more intuitive and useful concept is that of correlation.

4.6.3 Correlation

Correlation is closely related to the concept of covariance:

$$c[x, y] = \frac{C[x, y]}{\sqrt{V[x]V[y]}} \quad (54)$$

This is basically a **normalized** covariance, that will always lie in the interval $-1 \leq c \leq 1$. Correlation measures how *linearly* correlated two RVs are. With $C[x, x] = V[x] \implies c[x, x] = 1$ meaning that x is maximally correlated with itself.

```
samples = 400

mu = np.array([5.0, 5.0])

C1 = np.array([[ 4, -4], \
              [-4,  5]])

C2 = np.array([[4, 4], \
              [4, 5]])

C3 = np.array([[4, 0.1], \
              [0.1, 5]])

c1 = C1[0,1] / sqrt(C1[0,0]*C1[1,1])
c2 = C2[0,1] / sqrt(C2[0,0]*C2[1,1])
c3 = C3[0,1] / sqrt(C3[0,0]*C3[1,1])

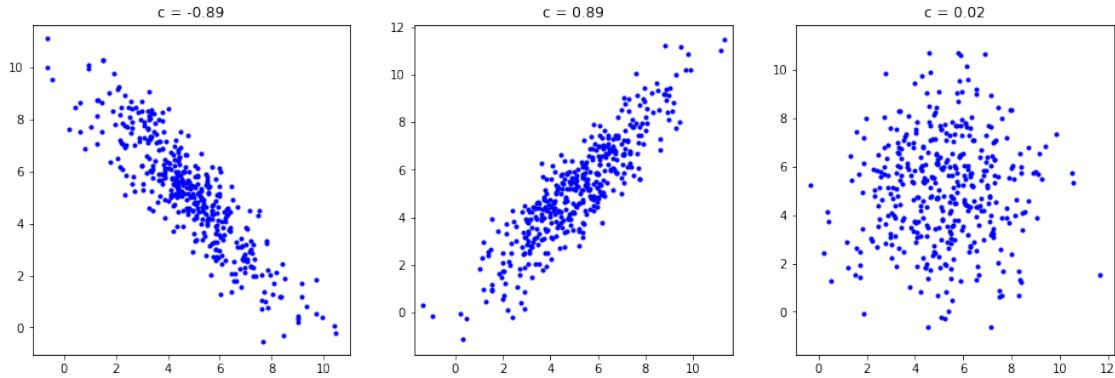
neg_vals = np.random.multivariate_normal(mu, C1, size=samples)
pos_vals = np.random.multivariate_normal(mu, C2, size=samples)
non_vals = np.random.multivariate_normal(mu, C3, size=samples)

plt.figure(figsize=(16,5))
plt.subplot(131)
plt.plot(neg_vals[:,0], neg_vals[:,1], 'b.')
plt.title("c = " + str(round(c1,2)))
plt.axis('equal')

plt.subplot(132)
plt.plot(pos_vals[:,0], pos_vals[:,1], 'b.')
plt.title("c = " + str(round(c2,2)))
plt.axis('equal')

plt.subplot(133)
plt.plot(non_vals[:,0], non_vals[:,1], 'b.')
plt.title("c = " + str(round(c3,2)))
plt.axis('equal')

plt.show()
```



Here, we can make much more sense of the value given by the correlation. When x increases, the decrease/increase in y deviates from the increase in x by roughly 11%. In the third image, there is no (linear) correlation at all.

4.6.4 Statistical Independence and Covariance

If X and Y are independent, their covariance vanishes: $C[X, Y] = 0$. This is what (roughly) happens in the third image in the examples above. The inverse of this statement is not true, a vanishing covariance does not imply independence. See for example the following distribution, where $E[x] \approx 0$ and $C[x] \approx 0$.

```
from random import random

samples = 100

x = linspace(-1, 1, samples)

vals = np.array([2*random()-1 for s in range(samples)])

plt.figure(figsize=[12,5])

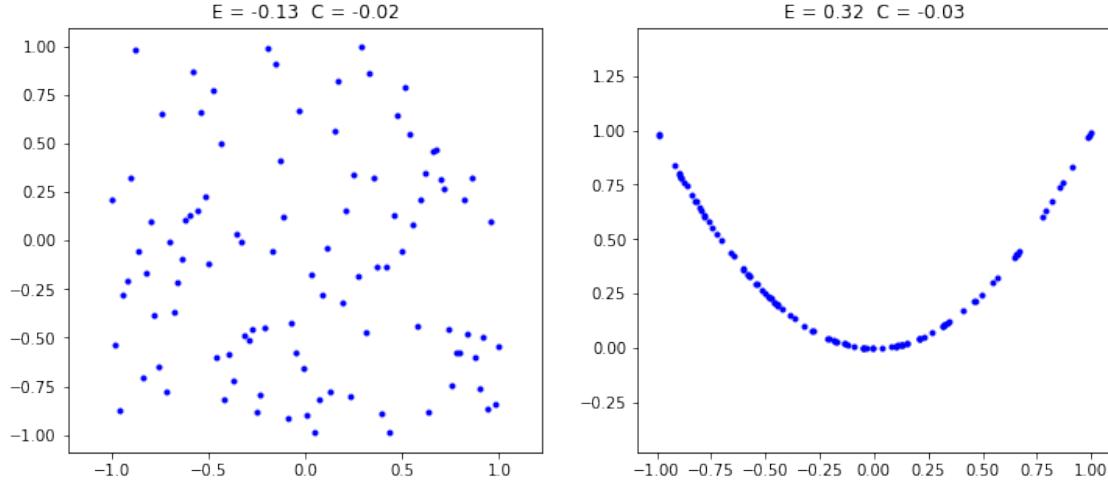
plt.subplot(121)
plt.plot(x, vals, 'b.')
plt.title("E = " + str(round(np.mean(vals),2)) + \
          " C = " + str(np.round(np.cov(x, vals)[0,1],2)))
plt.axis('equal')

plt.subplot(122)
plt.plot(vals, vals**2, 'b.')
plt.title("E = " + str(round(np.mean(vals**2),2)) + \
          " C = " + str(np.round(np.cov(vals, vals**2)[0,1],2)))
plt.axis('equal')

print("E[x] = ", round(np.mean(vals),2))
```

```
print("E[x^3] = ", round(np.mean(vals**3),2))
```

$E[x] = -0.13$
 $E[x^3] = -0.07$



For the second plot, a second RV $Y = x^2$ was introduced. Clearly, X and Y are not independent but still, the covariance is close to zero. We printed the expectation values for x and x^3 , which can be used to verify this:

$$C[x, x^2] = \overline{x \cdot x^2} - \bar{x} \cdot \bar{x^2} = E[x^3] - E[x]E[x^2] = 0 \quad (55)$$

Hence, a covariance of zero does not imply that variables are independent. It only means that there's no *linear* relationship between two variables.

We will often use the covariance matrix $C[\mathbf{x}, \mathbf{x}]_{ij} = C[x_i, x_j]$ for input vectors.

4.6.5 Further Relations

Using the definitions above and $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^m$, we get

$$C[\mathbf{x}, \mathbf{y}] = E[\mathbf{x}\mathbf{y}^T] - E[\mathbf{x}]E[\mathbf{y}]^T = C[\mathbf{y}, \mathbf{x}]^T \in \mathbb{R}^{n \times m} \quad (56)$$

and

$$C[\mathbf{x}, \mathbf{x}] = V[\mathbf{x}] = E[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T] = E[\mathbf{x}\mathbf{x}^T] - E[\mathbf{x}]E[\mathbf{x}]^T \in \mathbb{R}^{n \times n} \quad (57)$$

where $\boldsymbol{\mu}$ is the expectation value of the vector \mathbf{x} .

The expectation operator is linear, but what happens to the variance of the sum of two variables is a bit more involved:

$$V[x \pm y] = V[x] + V[y] \pm C[x, y] \pm C[y, x] \quad (58)$$

so linearity here only applies iff x and y are independent. It's always true that $V[\alpha x] = \alpha V[x]$.

4.6.6 Affine Transformation of RVs

Sometimes an RV \mathbf{x} is transformed by $\mathbf{y} = \mathbf{Ax} + \mathbf{b}$. This is an affine transformation (this is often called “linear”, but linear functions always map zero onto itself). So given the mean and variance for \mathbf{x} , can we derive them for \mathbf{y} ? Since expectation is linear, this is obvious:

$$E_{\mathbf{y}}[\mathbf{y}] = E_{\mathbf{x}}[\mathbf{Ax} + \mathbf{b}] = \mathbf{A}E_{\mathbf{x}}[\mathbf{x}] + \mathbf{b} = \mathbf{A}\boldsymbol{\mu} + \mathbf{b} \quad (59)$$

For the variance, the transformation is as follows:

$$V_{\mathbf{y}} = \mathbf{A}V_{\mathbf{x}}\mathbf{A}^T \quad (60)$$

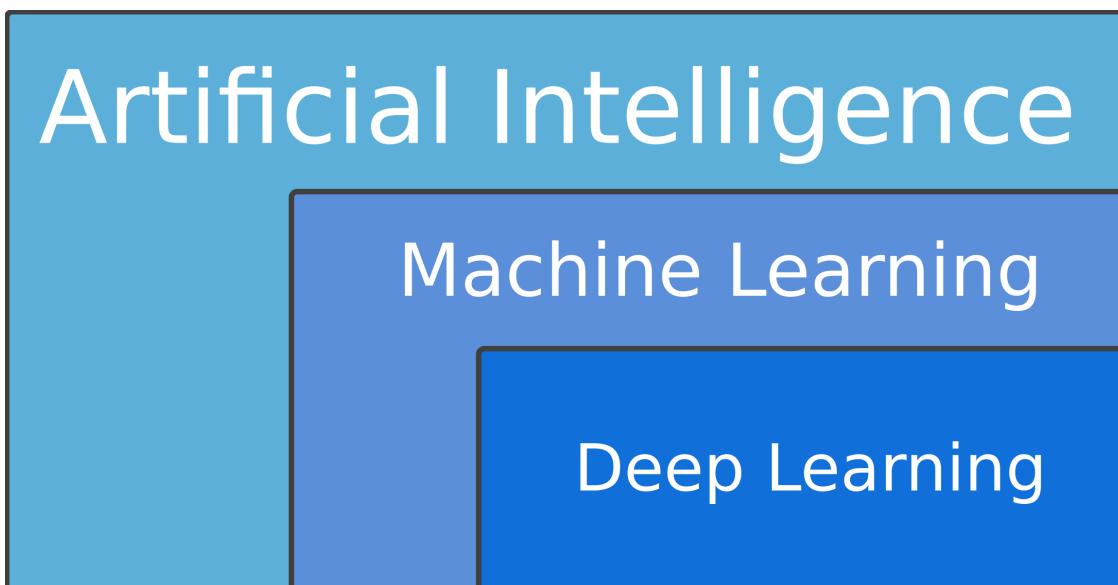
without a proof here. This should remind you of something from linear algebra.

5 Linear Regression

5.1 The Machine Learning Paradigm

Let's first take a look at how artificial intelligence (AI), machine learning (ML), and deep learning (DL) relate to each other in the modern definition of these terms:

```
from IPython.display import Image
Image("img/AI.png", width="500")
```



AI became pretty much a synonym for statistics in general these days, especially in industrial settings. What used to be called AI is *artificial general intelligence (AGI)* nowadays, where the idea is pretty much to develop human-like (or superhuman) reasoning. AI is an umbrella term for *algorithms performing a specific task which they weren't explicitly programmed for*, such as detecting objects in images with convolutional neural networks. AGI is a generalization of that, where the algorithms can not only handle a single specific task, but develop solution strategies for any task. The modern definition of ML was coined by Prof. Tom Mitchell:

A computer program is said to learn from experience E with respect to some task \mathbf{T} and some performance measure \mathbf{P} , if its performance on \mathbf{T} , as measured by \mathbf{P} , improves with experience \mathbf{E} .

or, in the short and less rigorous version:

Machine Learning is the study of computer algorithms that improve automatically through experience.

A common example for \mathbf{T} , \mathbf{P} , and \mathbf{E} is a spam-detector. The task is to categorize incoming mail as spam or ham, the performance measure is the ratio of correctly classified emails and the experience it gets is the end user labeling emails as spam or ham.

Deep learning refers to artificial neural networks (ANNs) with at least one hidden layer. The

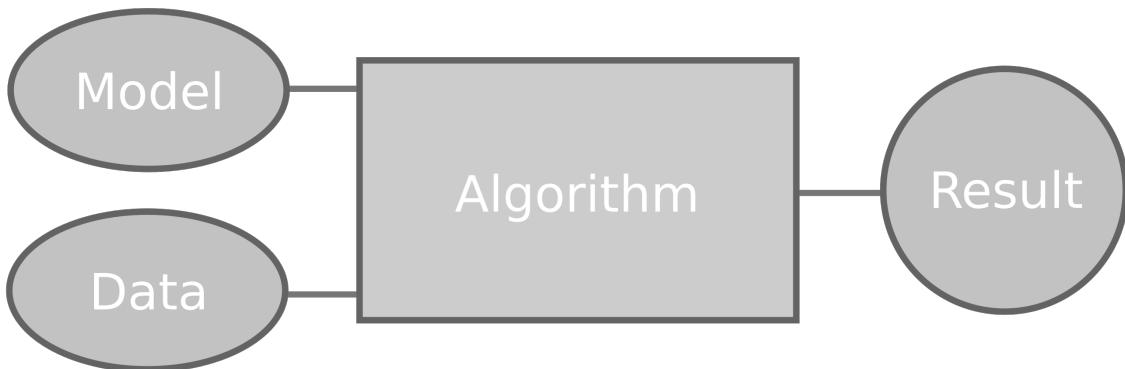
importance of these models comes from the *universal approximation theorem*, which states that ANNs can approximate (almost) any function/mapping arbitrarily well, given a single hidden layer with sufficiently many neurons and a nonlinearity. We'll come back to this later.

Classically, programming solutions to problems requires a **model** of the problem. Together with **data** about the problem, an algorithm can then output the answer or solution the programmer is looking for. Think for example of constitutive laws you have to provide for a simulation to predict the behaviour of some geometric body under load. The data would be the geometric properties, boundary conditions, and loads.

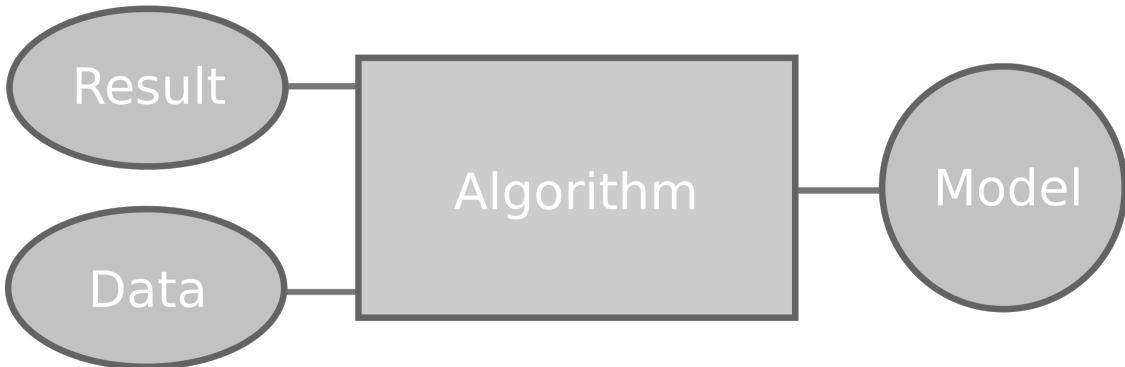
Very often, we don't have access to a good model or there simply doesn't exist a good model. So a different strategy would involve providing data and answers/solutions (e.g. from an experiment), and let an algorithm determine the *model*.

```
Image("img/ml-approach.png", width="500")
```

Classical Approach



ML Approach



This is the basic idea of *Machine Learning*; instead of providing the relationship between inputs and outputs, then calculating the outputs given some inputs, we let the computer **learn** the relationship by itself. Many people try this with financial data. Given the data from today (plus earlier times), having learned a model from past developments on the stock market, predict the developments for tomorrow. While sometimes this works on *short* time scales, this is generally a bad idea. The reason is that there naturally is *no underlying analytical connection* between data points other than psychology (which is the reason for the short-term success stories). A more appropriate application could be weather prediction. Building models in these highly chaotic settings is especially difficult.

Some work has been done in this regard [with astonishing results](#). A simple example would be deriving material laws from data gathered by an experiment.

The relationship between inputs and outputs is assumed to be a function, called the *model* or the **hypothesis**. For example, it can look like

$$h(\mathbf{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1x_2 \quad (61)$$

Characterizing a hypothesis/model are its **form**, and its **parameters/weights** w_i . The form here is a constant term, plus two linear terms, plus a quadratic term. Obviously, infinite forms are possible for such a hypothesis. This has to be chosen by the programmer (for now...), while Machine Learning takes care of the parameters. Choosing a good form usually requires **domain knowledge**, so knowledge in the specificities of the problem to be modeled. Say you want to model the stress-strain relationship of some material under load. Without knowing what this relationship (at least roughly) looks like, you will have to guess. Knowing that your material behaves linearly, you can instantly use a linear form for the hypothesis and get fitting results. You probably know this as *curve fitting* and in a certain way, machine learning is some form of that, albeit much more sophisticated.

5.1.1 Feed-forward Process / Forward Modeling

Given \mathbf{x} and a choice of w_i for some hypothesis $h(\mathbf{x}; \mathbf{w})$ (which could be an educated guess at some relationship), we can simply calculate the result $\hat{y} = h(\mathbf{x}; \mathbf{w})$. This is a forward pass/forward process/forward model or **feed-forward**.

5.1.2 Feedback

In contrast to feed-forward, here a huge **data set** is gathered (e.g. by an experiment). Let's say we want to model a stress-strain curve of an unknown material. To simplify this, let the experiment be designed such that we can prescribe a certain strain ε and the sensors give back the corresponding stress σ . The **ground truth** is the results of the experiment, usually denoted as y , so this is what actually exists in reality. A model always gives different values, denoted as \hat{y} . Guessing a hypothesis (*guessed model*), say, $h(\mathbf{x}; \mathbf{w}) = w_0 + w_1x_1 + w_2x_2$, $\hat{y} = h(\mathbf{x}_p; \mathbf{w})$ will not exactly hit the ground truth found by the experiment.

In this circumstance, \mathbf{x} and $h(\mathbf{x}; \mathbf{w})$ are fixed, while the parameters \mathbf{w} are guessed, so \hat{y} really depends on the parameters \mathbf{w} . Then we can define a **cost function** $J(y, \hat{y}(\mathbf{w}))$, that somehow depends on the *difference* between the ground truth and the model prediction such that it vanishes if both values match. We can see this as an *objective function*, with which we can *optimize* the weights \mathbf{w} . This process of using a cost function J to improve \mathbf{w} is called **feedback**. Optimization mostly happens via gradient descent, and most of machine learning really is optimization of such a loss function using some variant of gradient descent.

5.1.3 Machine Learning Pipeline

The process of developing a machine learning model is usually not very different from the following:

All problems are data, all solutions are functions/maps!

- Identify appropriate inputs (**features**) and outputs

- Create a sufficient (in size, prevalence, ...) **data set** following that convention
- Clean the data (NaNs etc.)
- Identify the necessary features (feature engineering)
- Explore the dataset
- Decide on the form of the *forward model* $\hat{y} = h(\mathbf{x}; \mathbf{w})$
- Choose a *loss function*, e.g. least squares: $J(y, \hat{y}) = (y - \hat{y})^2$
- Decide on an optimization algorithm, such as *gradient descent* and appropriate hyperparameters

A few caveats:

- Each choice of hypothesis function corresponds to a different purpose, say, linear functions for simple polynomial regression problems, logistic functions for binary classification problems, deep neural networks for (almost) any nonlinear problem (*see Universal Approximation Theorem*, later), convolutional neural networks for image-based problems (thanks to their translational invariance), recurrent neural networks for transient problems.
- There's also an appropriate choice of loss function for different problems, since they define the goal of the optimization.
- Machine-learned models aren't automatically the best solutions! Analyze the posed problem using as much domain knowledge as possible.

Although some say that a lot of domain knowledge is necessary to do machine learning, this is not quite true. Getting results, even good results, is actually quite easy once you understand what machine learning algorithms do and how to apply them, but it is very dangerous because you *will always* get *some* solution to a problem.

5.2 Linear Regression

Recall the introductory presentation on machine learning from the very first lecture. Most *supervised learning* problems can be formulated as either a **classification** problem, where data points are to be categorized or as a **regression** problem, where a continuous range of values is to be predicted. Recall the example where we wanted to identify cracks on images of hulls. A classification, learned on examples with **labeled** data of images with either a crack or no crack, could sort a new image into one of these categories.

```
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt

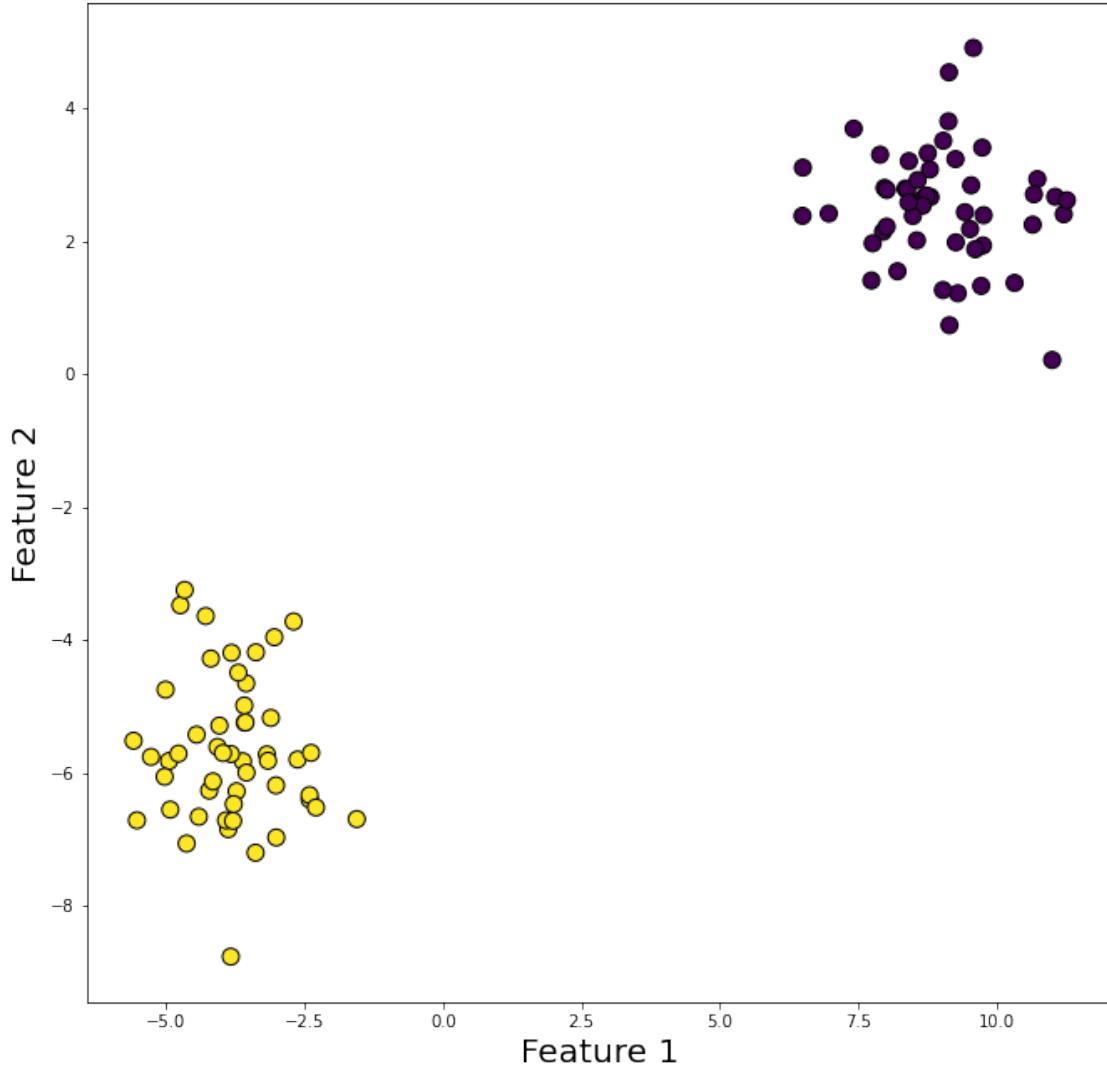
plt.figure(figsize=(11,11))
plt.subplot(111)

X1, Y1 = make_blobs(n_features=2, centers=2)

plt.xlabel("Feature 1", fontsize=20)
plt.ylabel("Feature 2", fontsize=20)

plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=100, edgecolor='k')
```

```
<matplotlib.collections.PathCollection at 0x7f22026fcc18>
```



Regression on the other hand works with data as real number values.

```
import numpy as np

plt.figure(figsize=(12, 8))
plt.subplot(111)

samples = 20
noise = np.random.normal(0, 0.1, samples)
= np.linspace(0, 2, samples)/10
= (10* -2)**3 + 8 + noise
[0] = 0.0
```

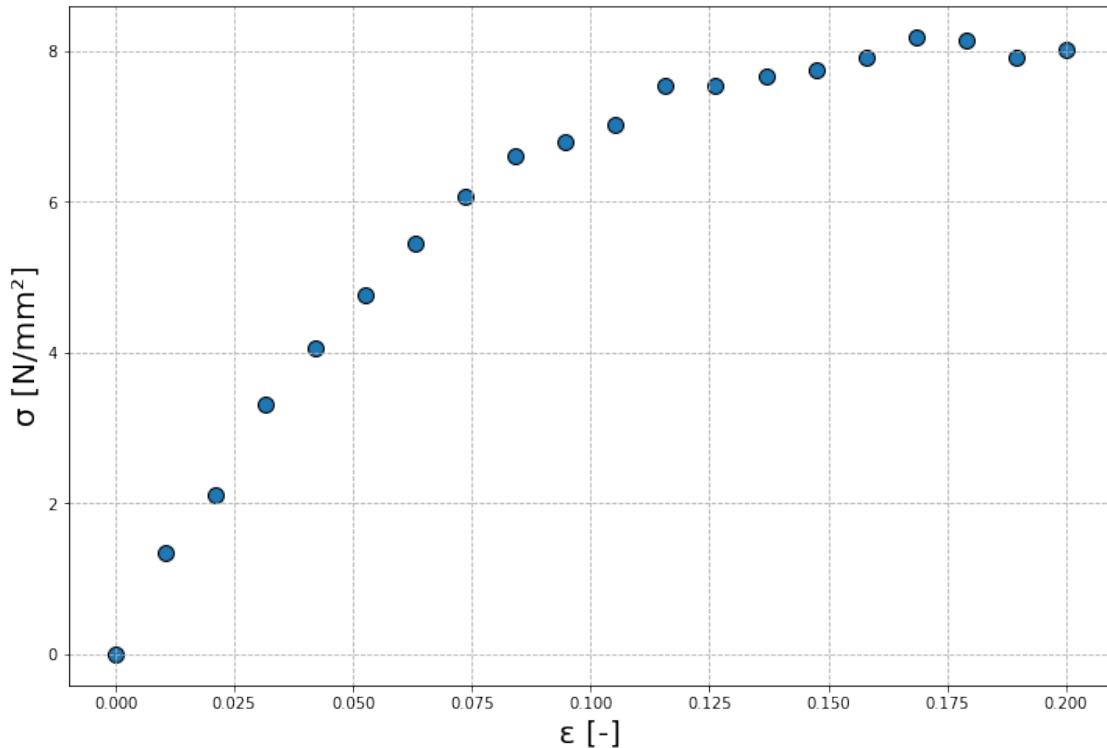
```

plt.xlabel(" [-]", fontsize=20)
plt.ylabel(" [N/mm²]", fontsize=20)
plt.xlim([-0.01, 0.21])
plt.grid(True, linestyle="--")

plt.scatter( , , marker='o', s=100, edgecolor='k')

```

<matplotlib.collections.PathCollection at 0x7f22026925c0>



This example shows a stress-strain curve of ideally elastic, but nonlinear behavior. The plotted points were determined by an experimental setup (and are *not exact* in real experiments, but we'll ignore this for now), the measured stresses σ are called the **ground truth**, at predetermined strains ϵ . What can we do if we wanted to predict the stress at some intermediate point $\epsilon = 0.1$, that wasn't measured? One way would be to interpolate linearly between $\epsilon = 0.05$ and $\epsilon = 0.10$ by adding the stresses σ at those points and dividing by 2. Not all the available data is used in this variant and hence, the result perhaps not in congruence with the general material behavior.

Instead, we can form a *hypothesis* and see how well it fits the data. Since we don't know which hypothesis will model the data well, we have to try out a few. You can test this in the graph below.

```

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

```

```

from ipywidgets import interact
import ipywidgets as widgets

plt.figure(figsize=(12, 8))
#plt.subplot(111)

# the regression and results will be precalculated to speed up plotting
# the way this works is that the data points are transformed,
# then a linear regression is performed on the transformed data
# we will take a closer look at this in the assignment

# this list will contain the fitted polynomials
regression_result = []
# this list will contain transformed "training data"
_poly = []
# predetermine the polynomials
poly = [PolynomialFeatures(degree=degree) for degree in range(1, 4)]

# precalculate all regressions for polynomials up to degree 3
for degree in range(1, 4):
    # transforms "training data" into polynomial data,
    # such that -> [1, , ^2] for example
    # these are the "features", the "1" is for bias
    _poly.append(poly[degree-1].fit_transform(.reshape(-1, 1), ))

    # perform linear regression on polynomial data
    lin = LinearRegression()
    lin.fit(_poly[degree-1], )
    # append results to regression_result list
    regression_result.append(lin)

def plot_reg(degree, new_point):
    new_approx = regression_result[degree-1].predict(poly[degree-1].
    ↪fit_transform([[new_point]]))

    #plt.cla()
    plt.scatter(1000* , , color = 'steelblue')
    plt.plot(1000* , regression_result[degree-1].predict(_poly[degree-1]), \
              color = 'darkorange', lw=3)
    plt.scatter(1000*new_point, *new_approx, color='red')
    plt.xlabel(r"  $[\$10^{-3}\$]$ ", fontsize=20)
    plt.ylabel(r"  $[N/mm^2]$ ", fontsize=20)

    index = np.searchsorted( , new_point, side="left")
    if index < .shape[0]-1:
        print("Linear interpolation: ( = 0.1) = ", ( [index] + [index+1])/2)

```

```

print("Regression degree", degree, ":", ( = 0.1) = ", *new_approx)

plt.tight_layout()
plt.savefig("img/plot_reg.png")

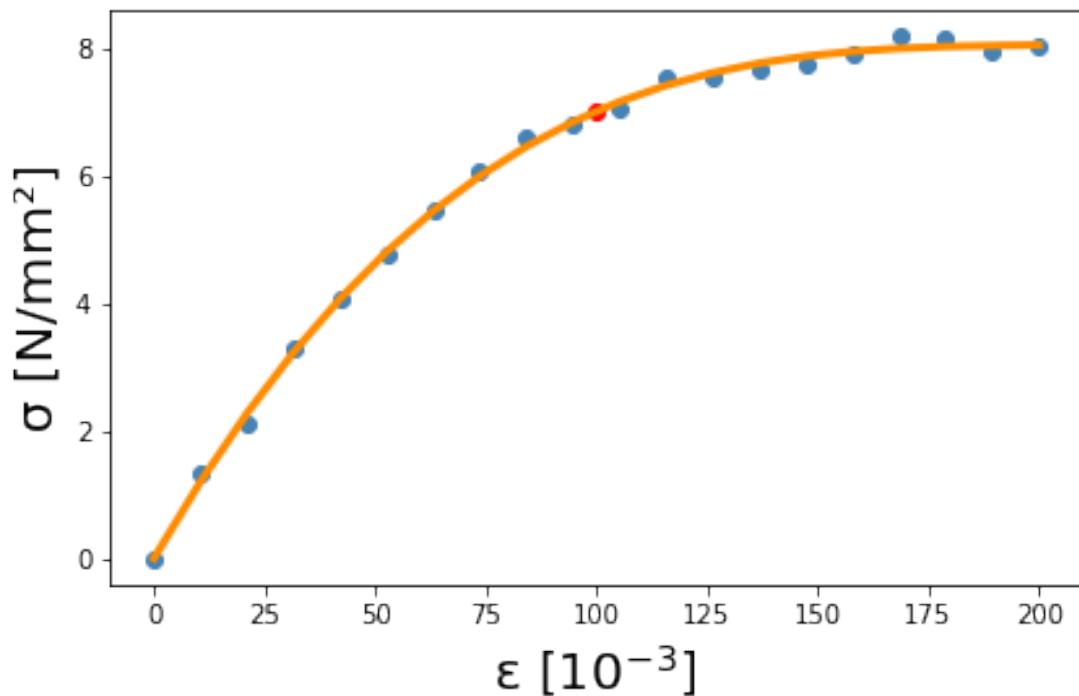
interact(plot_reg, degree=(1, 3), new_point=(0.0, 0.2, 0.02))

```

<Figure size 864x576 with 0 Axes>

interactive(children=(IntSlider(value=2, description='degree', max=3, min=1), FloatSlider(value=0.0, step=0.2, max=0.02, min=0.0), FunctionOutput(function=_main_.plot_reg, name='plot')))

<function _main_.plot_reg(degree, new_point)>



This obviously necessitates *polynomial* regression, since a single line cannot capture the nonlinear behaviour displayed here. The feature space (the input vector components) depend on the degree of the polynomial. For the quadratic case, the **feature space** is $S = [1 \ \epsilon \ \epsilon^2]$. For a quadratic polynomial in two variables, it would be $S = [1 \ x \ y \ xy \ x^2 \ y^2]$. You can think of each feature as a *basis vector* or *basis function*. While for example the projection onto a vector $[0, 0, 1]$ in a 3d space gives information about the “z-ness” of an arbitrary vector, the “projection” onto the vector $[0, 0, \epsilon^2]$ would give information about the “quadraticness” of an arbitrary vector in this function space.

A quadratic hypothesis fits the data much better, and a cubic hypothesis performs even better at the endpoints. It is not true that the higher degree a polynomial has, the better its performance. It's rarely a good idea to fit data this well (*overfitting*), but we'll see this in a later lesson today.

5.3 Least Squares and Gradient Descent

Let's formalize the notions from last lesson a bit and see how we can quantify how well a hypothesis fits some data.

The stresses σ_i , as measured by the experiment, were the **ground truth** of the problem. The problem itself is called the *general univariate linear regression problem*, where there's a *single* input, and a *single* output. $(x^{(i)}, y^{(i)})$ is called the *i*th **example**, or data point. The input for the algorithm is (\mathbf{x}, \mathbf{y}) , where \mathbf{x} and \mathbf{y} are vectors with dimension the same as the number of examples in the data set, and a *form* of hypothesis. The output of the algorithm is the model $h(\mathbf{x}; \mathbf{w})$ and its predictions $\hat{\mathbf{y}}$.

In the first try we used a linear (actually *affine*) model $\hat{y} = h(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1$. This always produces a line in the plot. There are infinitely many choices for w_0 and w_1 , so which one is the best?

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact#, interactive, fixed, interact_manual
# import ipywidgets as widgets

samples = 20
noise = np.random.normal(0, 0.1, samples)
x = np.linspace(0, 2, samples)
y = x + noise

w_0_m, w_1_m = np.meshgrid(np.linspace(-2, 2, 100), np.linspace(-2, 2, 100))
J_m = 1/(2*samples) * np.array(np.sum([(y[i] - w_1_m*x[i] - w_0_m)**2 for i in range(samples)]), axis=0)

def plot_line(w_0, w_1):
    fig = plt.figure(figsize=(8, 8))
    ax1 = fig.add_subplot(211)
    ax2 = fig.add_subplot(212, projection='3d')
    ax1.set_aspect(aspect=1)
    #ax1.cla()
    ax1.scatter(x, y, color = 'steelblue')
    ax1.plot(x, w_1*x + w_0, color = 'darkorange', lw=3)

    #ax2.cla()
```

```
J = 1/(2*samples) * np.sum((y - w_1*x - w_0)**2) #np.sum([(y[i] - w_1*x[i] - w_0)**2 for i in range(samples)])
ax2.plot_surface(w_0_m, w_1_m, J_m, rstride=8, cstride=8, alpha=0.5, cmap="viridis")
ax2.scatter(w_0, w_1, J, lw=4, alpha=1, color='darkorange', zorder=100)

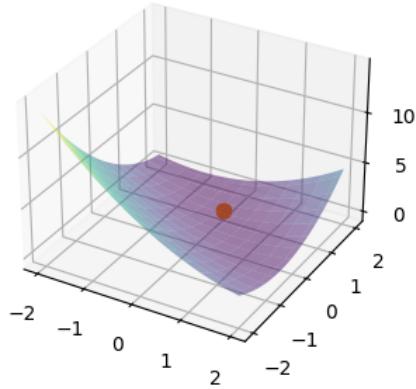
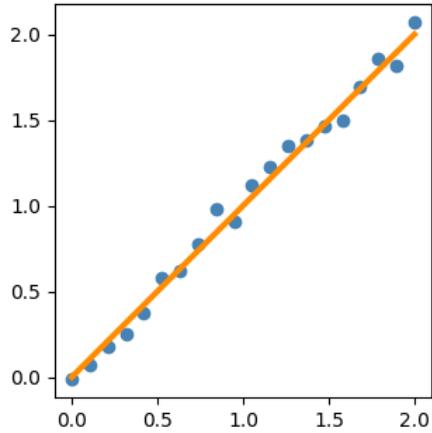
print("Cost function J = ", J)

plt.savefig("img/plot_line.png")

interact(plot_line, w_0=(-2.0, 2.0), w_1=(-2.0, 2.0))

interactive(children=(FloatSlider(value=0.0, description='w_0', max=2.0, min=-2.0), FloatSlider(value=0.0, description='w_1', max=2.0, min=-2.0), Output()), _js=[{"w_0": "w_0", "w_1": "w_1"}])

<function __main__.plot_line(w_0, w_1)>
```



In the above graph on the left, you can choose different parameters to get different lines. As you probably guessed, w_0 determines where the line cuts the y -axis, also called the *bias*, and w_1 determines the *slope* of the line. If there wasn't a bias term, all lines would cross the origin, no matter what other choice of parameters is taken. On the right is the **loss/cost landscape**. Underneath the graph the value of the **loss/cost function** is calculated and also shown in the right plot.

$$J = \frac{1}{2N} \sum_i^N \left(y^{(i)} - \hat{y}^{(i)} \right)^2 \quad (62)$$

where N is the number of samples in the data set. The best line has the lowest J , so the optimal choice of \mathbf{w} is the one that minimizes J . The factor $\frac{1}{2N}$ isn't mandatory here, but the 2 will later get cancelled out and taking the mean of the squared error avoids overflow problems. The result

without the factor would be the same. This is called a **least squares fit**, J is the least mean square, or **LMS cost function**.

To optimize J , we can use *gradient descent*. The procedure would go as follows:

- choose an $x^{(i)}$
- guess w_0 and w_1
- feed-forward both through the hypothesis $h(x^{(i)}; \mathbf{w})$
- calculate J using the ground truth $y^{(i)}$ corresponding to $x^{(i)}$
- calculate the gradient of J
- improve w_0 and w_1 using this gradient (decide on learning rate and stopping criterion beforehand)
- repeat until convergence or maximum number of steps is reached
- final w_0 and w_1 are the regression coefficients

To calculate the gradient of J :

$$\frac{\partial J}{\partial w_j} = -\frac{1}{N} \sum_i^N (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)} \quad (63)$$

Update rule for the weights:

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \alpha \nabla_w J \quad (64)$$

so that for the weights in our case we get

$$w_0^{(k+1)} = w_0^{(k)} - \alpha \frac{\partial J}{\partial w_0} \quad (65)$$

$$w_1^{(k+1)} = w_1^{(k)} - \alpha \frac{\partial J}{\partial w_1} \quad (66)$$

and for J

$$J = \frac{1}{2N} \sum_i^N (y^{(i)} - w_0 - w_1 x^{(i)})^2 \quad (67)$$

```
import pandas as pd
import random
from IPython.display import display

def J(w_0, w_1):
    return 1/(2*N) * np.sum((y - w_1*x - w_0)**2)

N = samples # for brevity
```

```

= 0.7
max_steps = 30
precision = 5e-3
w_0 = 2*random.random() - 1
w_1 = 2*random.random() - 1

# list of values to be printed as table later
w = [[1, w_0, w_1, J(w_0, w_1)]]


for i in range(1, max_steps):
    Δw_0 = /N * np.sum((y - w_1*x - w_0))
    Δw_1 = /N * np.sum((y - w_1*x - w_0)*x)

    w_0 += Δw_0
    w_1 += Δw_1

    w.append([i+1, w_0, w_1, J(w_0, w_1)])

    if np.linalg.norm([Δw_0, Δw_1]) < precision:
        break

headers = [r"Step", r"$w_0$", r"$w_1$", r"$J$"]

display(pd.DataFrame(w, columns=headers))

```

	Step	\$w_0\$	\$w_1\$	\$J\$
0	1	0.303893	0.582810	0.044530
1	2	0.390753	0.782115	0.025268
2	3	0.277297	0.729706	0.017217
3	4	0.279947	0.806918	0.012977
4	5	0.226693	0.808314	0.010268
5	6	0.209740	0.845650	0.008331
6	7	0.178518	0.859090	0.006872
7	8	0.159744	0.881511	0.005749
8	9	0.138417	0.895597	0.004879
9	10	0.122159	0.911119	0.004202
10	11	0.106416	0.923153	0.003674
11	12	0.093270	0.934679	0.003263
12	13	0.081257	0.944367	0.002943
13	14	0.070871	0.953184	0.002693
14	15	0.061584	0.960825	0.002499
15	16	0.053449	0.967648	0.002347
16	17	0.046233	0.973630	0.002229
17	18	0.039880	0.978933	0.002137

```

18   19  0.034262  0.983603  0.002065
19   20  0.029308  0.987733  0.002009
20   21  0.024931  0.991374  0.001966
21   22  0.021069  0.994592  0.001932
22   23  0.017658  0.997431  0.001905

```

5.4 Quantifying Fitness

In the previous example we could intuitively see that the cubic hypothesis performed better than the linear one, but this intuition won't work in higher dimensions, since we are confined to plotting at most 3 dimensions.

In the following plot, which of the depicted lines is a better regression model of the data cloud?

```

import numpy as np
import matplotlib.pyplot as plt

samples = 30

noise = np.random.normal(0, 0.3, samples)
#noise = 0

mu = np.array([5.0, 5.0])

C3 = np.array([[4, 0.9], \
              [0.9, 5]])

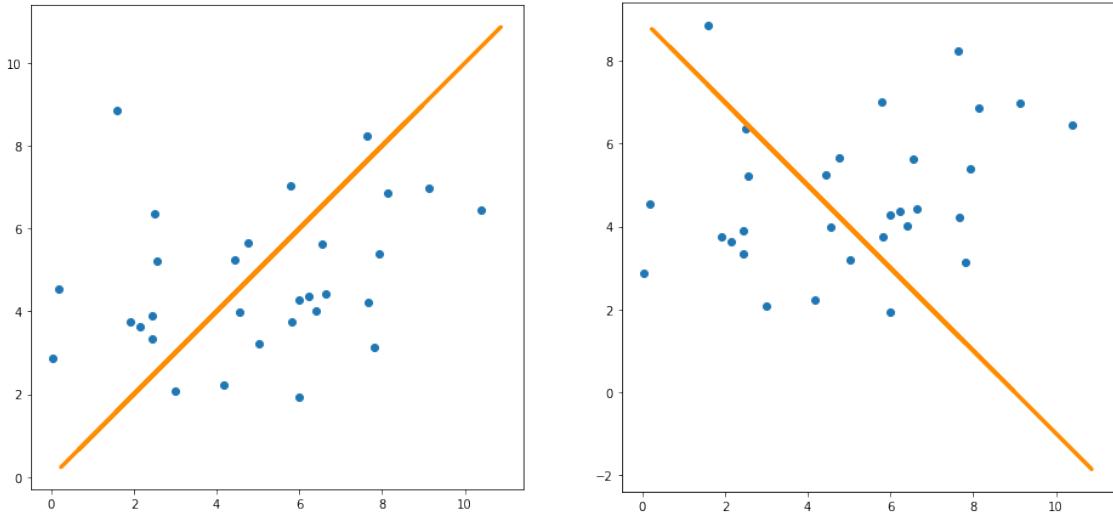
vals = np.random.multivariate_normal(mu, C3, size=samples)

fig = plt.figure(figsize=(16,11))
ax1 = fig.add_subplot(121)
ax1.set_aspect(aspect=1)
ax2 = fig.add_subplot(122)
ax2.set_aspect(aspect=1)

# implicitly, the dataset here is vals[:,0] + noise
# hypothesis 1 is hat y = vals[:,0]
ax1.scatter(vals[:,0]+noise, vals[:,1])
ax1.plot(vals[:,0], vals[:,0], lw=3, color='darkorange')
# hypothesis 2 is hat y = 9 - vals[:,0]
ax2.scatter(vals[:,0]+noise, vals[:,1])
ax2.plot(vals[:,0], -vals[:,0] + 9, lw=3, color='darkorange')

```

[<matplotlib.lines.Line2D at 0x7ff9370ebd30>]



We could use the loss function J (the Mean Squared Error, MSE) itself, but often this is not a good measure at all. It's especially difficult to interpret when large numbers are involved. Large parameters lead to a large loss function and large values are difficult to interpret. A useful quantifier would give a value between 0, denoting a bad fit, and 1, denoting a good fit. Think of this as some kind of normalization.

A better, more reliable measure is the **coefficient of determination** value R^2 , which *should* lie in $R^2 \in [0, 1]$. There are multiple ways to calculate it, but they all involve two of the following quantities:

- **Sum Square Total:** $SST = \sum_i^N (y_i - \bar{y})^2$, also known as *total variance*, calculates total amount of variance in the data
- **Sum Square Error:** $SSE = \sum_i^N (y_i - \hat{y}_i)^2$, the total amount of squared error in the data, also known as *sum squared predictive error*
- **Sum Square Regression:** $SSR = \sum_i^N (\hat{y}_i - \bar{y})^2$, calculating how much predictions vary from the mean, also known as *variance in prediction*, or *amount of variance captured by the model*

The R^2 value is then defined as

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST} \quad (68)$$

which is the amount of variance explained/captured by the model, divided by the total amount of variance present in the data. Using $SST = SSE + SSR$, the form on the right can be derived, which is the implementation found in most algorithms.

```
import pandas as pd
from IPython.display import display

SST1 = np.sum((vals[:,0]+noise - np.mean(vals[:,0]+noise))**2)
SSE1 = np.sum((vals[:,0]+noise - vals[:,0])**2)
```

```

SSR1 = np.sum((vals[:,0] - np.mean(vals[:,0]+noise))**2)
R2_1 = SSR1/SST1

SST2 = np.sum((vals[:,0]+noise - np.mean(vals[:,0]+noise))**2)
SSE2 = np.sum((vals[:,0]+noise - (-vals[:,0]+9))**2)
SSR2 = np.sum((-vals[:,0]+9 - np.mean(vals[:,0]+noise))**2)
R2_2 = SSR2/SST2

data = [[1, SST1, SSE1, SSR1, R2_1], \
         [2, SST2, SSE2, SSR2, R2_2]]
headers = ["Curve", "SST", "SSE", "SSR", "R2"]

display(pd.DataFrame(data, columns=headers))

```

	Curve	SST	SSE	SSR	R2
0	1	206.065991	1.750646	207.491435	1.006917
1	2	206.065991	854.238153	236.365382	1.147037

The fact that we get values higher than 1 or lower than 0 here means that our curve fits the data *worse than a horizontal hyperplane*, or, in other words, worse than a constant.

5.5 Generalization

Let's try to fit a polynomial to a sine curve by sampling some points directly on the curve, then adding some noise:

```

%matplotlib inline
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
from ipywidgets import interact
import ipywidgets as widgets

plt.figure(figsize=(8, 8))
#plt.subplot(111)

samples = 15
max_degree = 15
noise_level = 0.3

noise = np.random.normal(0, noise_level, samples)
x = np.linspace(-np.pi, np.pi, samples)
y = np.sin(x) + noise

# for testing the generalization capability of the model,
# some extra data points are generated from the same distribution

```

```

# usually, this is done differently, but we'll come back to this later
noise_test = np.random.normal(0, noise_level, 10)
x_test = np.linspace(-np.pi, np.pi, 10)
y_test = np.sin(x_test)

regression_result = []
x_poly = []
x_test_poly = []
# R2 scores are used in a later plot
R2 = []
R2_test = []
rms_error = []
rms_error_test = []

poly = [PolynomialFeatures(degree=degree) for degree in range(1, max_degree+1)]

for degree in range(1, max_degree+1):
    x_poly.append(poly[degree-1].fit_transform(x.reshape(-1, 1), y))
    x_test_poly.append(poly[degree-1].fit_transform(x_test.reshape(-1, 1), y_test))

    lin = LinearRegression()
    lin.fit(x_poly[degree-1], y)

    regression_result.append(lin)

    R2.append(regression_result[degree-1].score(x_poly[degree-1], y))
    R2_test.append(regression_result[degree-1].score(x_test_poly[degree-1], y_test))

    rms_error.append(np.sqrt(2/samples * np.sum((regression_result[degree-1].predict(x_poly[degree-1]) - y)**2)))
    rms_error_test.append(np.sqrt(2/samples * np.sum((regression_result[degree-1].predict(x_test_poly[degree-1]) - y_test)**2)))

def plot_reg(degree, plot_sine, plot_test):
    #plt.cla()
    plt.scatter(x, y, color = 'steelblue', label=r"Training data")
    plt.plot(x, regression_result[degree-1].predict(x_poly[degree-1]), \
              color = 'darkorange', lw=3, label=r"Prediction")
    plt.fill_between(x, np.sin(x) - noise_level, np.sin(x) + noise_level,
                     color='gray', alpha=0.2, label=r"Standard deviation of noise")

    if plot_sine:

```

```

plt.plot(x, np.sin(x), color='green', linestyle="--", lw=3, label=r"Ground truth")

if plot_test:
    plt.scatter(x_test, y_test, color='red', zorder=100, label=r"Test data")

print("Training R2 = ", R2[degree-1])
print("Test R2      = ", R2_test[degree-1])

plt.savefig("img/plot_reg_deg.png")

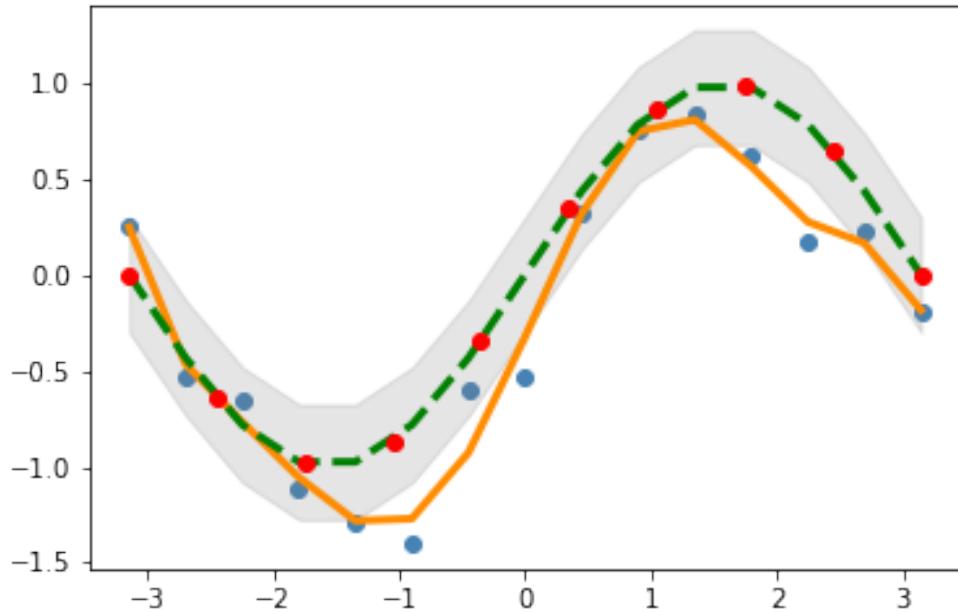
interact(plot_reg, degree=(1, max_degree), plot_sine=False, plot_test=False)

```

<Figure size 576x576 with 0 Axes>

interactive(children=(IntSlider(value=8, description='degree', max=15, min=1), Checkbox(value=False, checked=False, description='plot sine'), Checkbox(value=False, checked=False, description='plot test')))

<function __main__.plot_reg(degree, plot_sine, plot_test)>



If the degree of the polynomial is too high, we intuitively see that this is not really what we wanted to achieve, the curve is not at all close to a sine. In fact, when the polynomial degree reaches

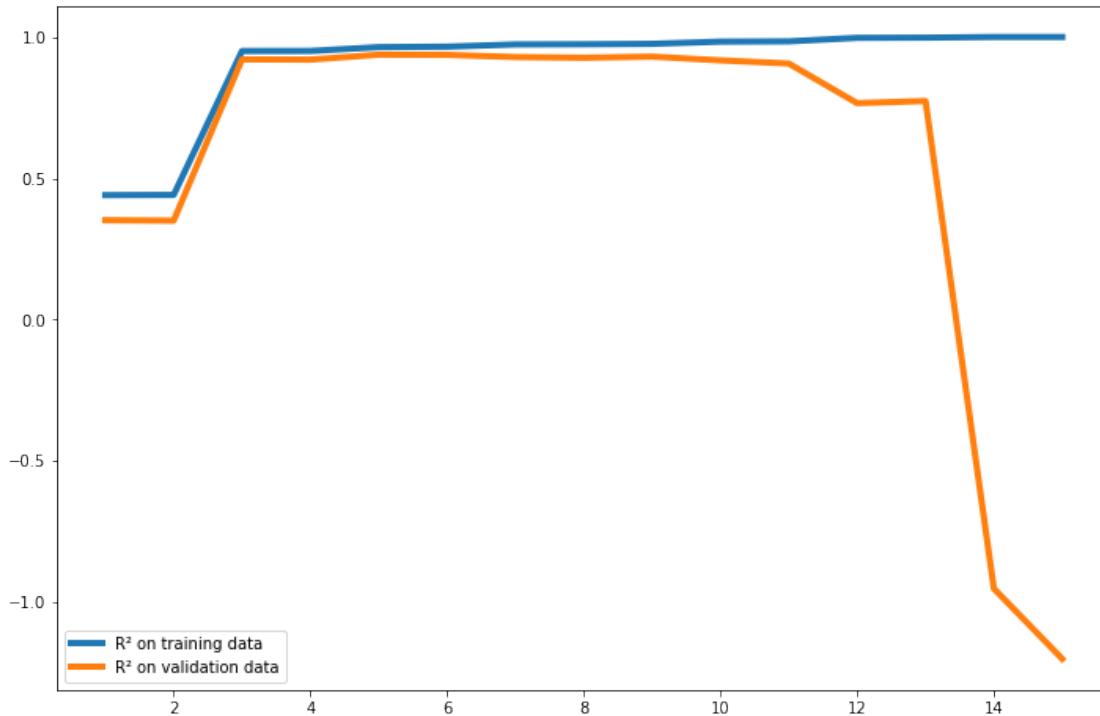
the number of samples, we just built a very inefficient way to save the samples in a “polynomial database”.

We can plot the R^2 -value against the degree of the polynomial:

```
degrees = [i for i in range(1, max_degree+1)]

plt.figure(figsize=(12,8))
plt.plot(degrees, R2      , lw=4, label='R2 on training data')
plt.plot(degrees, R2_test, lw=4, label='R2 on validation data')
plt.legend()
```

<matplotlib.legend.Legend at 0x7efcdd8be908>

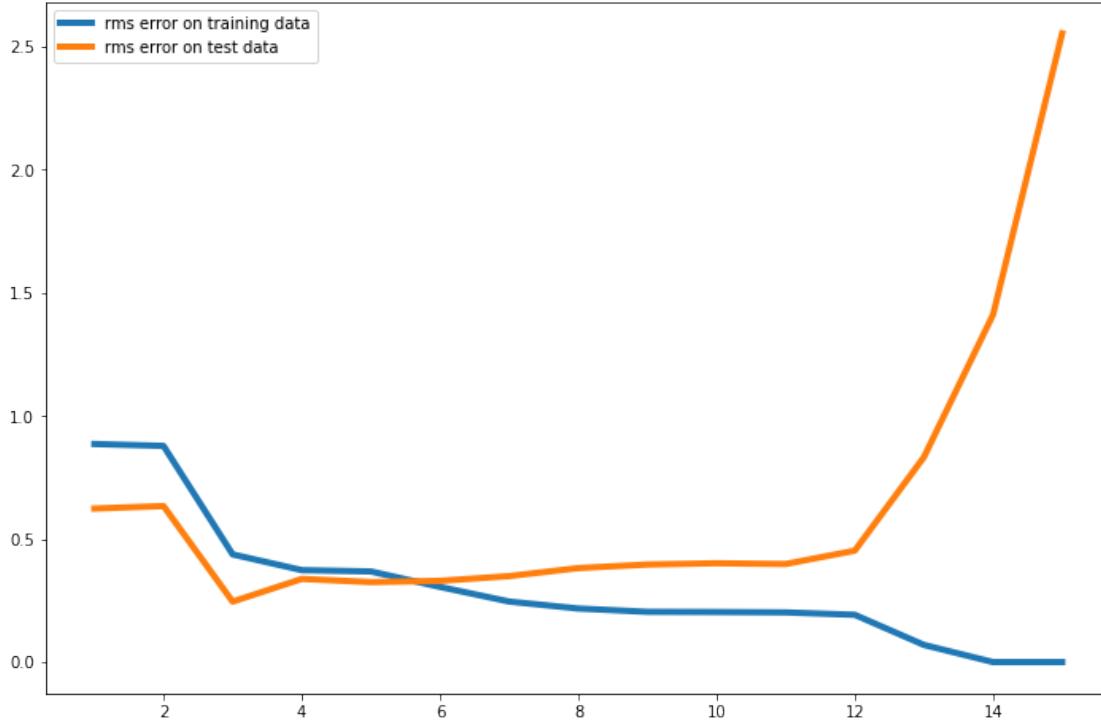


The R^2 -value becomes very close to 1 on the training data, but with increasing degree of the polynomial used for fitting becomes ever worse. We can also plot the *root mean square* error against the polynomial degree:

```
degrees = [i for i in range(1, max_degree+1)]

plt.figure(figsize=(12,8))
plt.plot(degrees, rms_error      , lw=4, label='rms error on training data')
plt.plot(degrees, rms_error_test, lw=4, label='rms error on test data')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7efcdd8db198>
```



The error on the training data becomes very small, but the error on the test data becomes extremely high. This is called **overfitting**. On the left side of the graph, we see **underfitting** with a high training error and a high test error. This is why in machine learning you really *don't want to have zero error*. Your model won't **generalize** to new data sampled from the same distribution. In order for things to work, there *must* be a training error.

This is a very important point. In settings with sufficiently large data sets, you are almost guaranteed to be able to find situations in which this unavoidable error will cause a severe error in a prediction for some parameter combinations. Imagine for example you're using machine learning in order to develop a new drug for some disease. The algorithm might find astonishing results, a new drug that will heal 99.99% of all patients. But: a rare mixture of conditions will lead to the patients death. Examples for this problem are legion, and actually more problems like this contribute to some scepticism regarding the use of machine learning algorithms in critical situations. We will discuss a few other problems later.

Observing this behavior in a different data set:

```
samples = 45
noise = np.random.normal(0, 0.1, samples)
x = np.random.random(samples)
x.sort()
y = np.sin(np.pi*x) + noise
```

```

x_test = np.random.random(1)
# you can define different functions here
# you might have to adjust noise levels above
y_test = np.sin(np.pi*x_test)

degrees = [1, 3, samples]
regression_results = []
poly = [PolynomialFeatures(degree=degree) for degree in degrees]
x_poly = []

for i in range(len(degrees)):
    x_poly.append(poly[i].fit_transform(x.reshape(-1, 1), y))
    #x_test_poly.append(poly[degree-1].fit_transform(x_test.reshape(-1, 1), y))

lin = LinearRegression()
lin.fit(x_poly[i], y)

regression_results.append(lin)

fig = plt.figure(figsize=(14,6))
ax1 = fig.add_subplot(131)
ax1.set_title("Underfitting, degree " + str(degrees[0]))

ax2 = fig.add_subplot(132)
ax2.set_title("Appropriate Capacity " + str(degrees[1]))

ax3 = fig.add_subplot(133)
ax3.set_title("Overfitting, degree " + str(degrees[2]))

# underfitting
ax1.scatter(x, y)
ax1.plot(x, regression_results[0].predict(x_poly[0]), \
          color = 'darkorange', lw=3)
ax1.scatter(x_test, y_test, color='red', zorder=100)
ax1.plot(x, np.sin(np.pi * x), linestyle='--', color='green')

# appropriate complexity
ax2.scatter(x, y)
ax2.plot(x, regression_results[1].predict(x_poly[1]), \
          color = 'darkorange', lw=3)
ax2.scatter(x_test, y_test, color='red', zorder=100)
ax2.plot(x, np.sin(np.pi * x), linestyle='--', color='green')

# overfitting
ax3.scatter(x, y)
ax3.plot(x, regression_results[2].predict(x_poly[2]), \

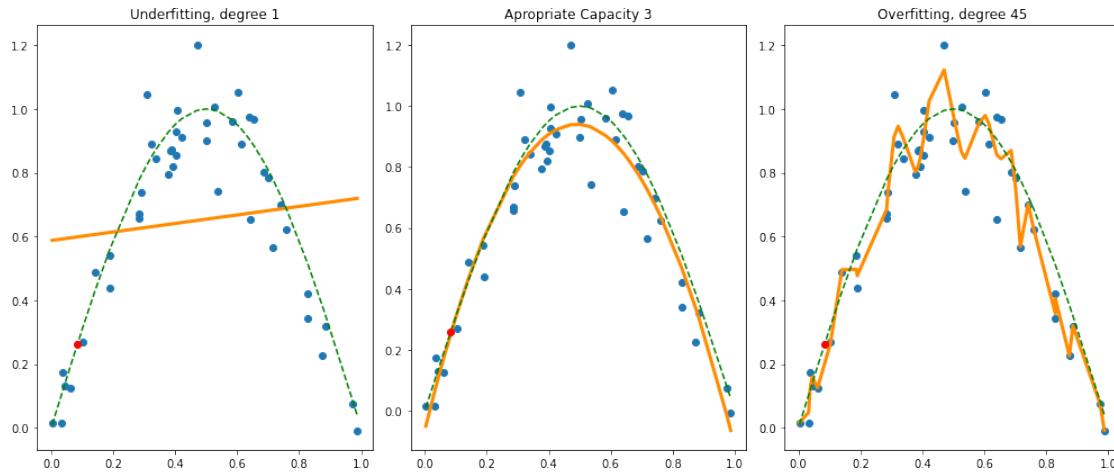
```

```

        color = 'darkorange', lw=3)
ax3.scatter(x_test, y_test, color='red', zorder=100)
ax3.plot(x, np.sin(np.pi * x), linestyle='--', color='green')

plt.tight_layout()

```



In the above graphs, if we introduce new example points, we see that in the underfitting and overfitting case, the curve rarely hits them if it does so at all. Hence here, the cubic model has appropriate **capacity**. How can we determine the necessary capacity for a given problem?

To control whether a model is more likely to under- or overfit, we can alter its capacity, where capacity is basically a model's ability to fit a certain class of functions.

```

plt.figure(figsize=(12, 7))
#plt.subplot(111)

samples = 30
max_degree = 28#samples//2

noise = np.random.normal(0, 0.3, samples)
x = np.linspace(-np.pi, np.pi, samples)
y = np.sin(1.5 * x) + noise

# for testing the generalization capability of the model,
# some extra data points are generated from the same distribution
# usually, this is done differently, but we'll come back to this later
noise_test = np.random.normal(0, 0.3, samples//3)
x_test = np.linspace(-np.pi, np.pi, samples//3)
y_test = np.sin(1.5 * x_test) + noise_test

regression_result = []

```

```

x_poly = []
x_test_poly = []
# R2 scores are used in a later plot
R2 = []
R2_test = []
rms_error = []
rms_error_test = []

poly = [PolynomialFeatures(degree=degree) for degree in range(1, max_degree+1)]

for degree in range(1, max_degree+1):
    x_poly.append(poly[degree-1].fit_transform(x.reshape(-1, 1), y))
    x_test_poly.append(poly[degree-1].fit_transform(x_test.reshape(-1, 1),  

→y_test))

    lin = LinearRegression()
    lin.fit(x_poly[degree-1], y)

    regression_result.append(lin)

    R2.append(regression_result[degree-1].score(x_poly[degree-1], y))
    R2_test.append(regression_result[degree-1].score(x_test_poly[degree-1],  

→y_test))

    rms_error.append(np.sqrt(2/samples * np.sum((regression_result[degree-1].  

→predict(x_poly[degree-1]) - y)**2)))
    rms_error_test.append(np.sqrt(2/samples * np.  

→sum((regression_result[degree-1].predict(x_test_poly[degree-1]) -  

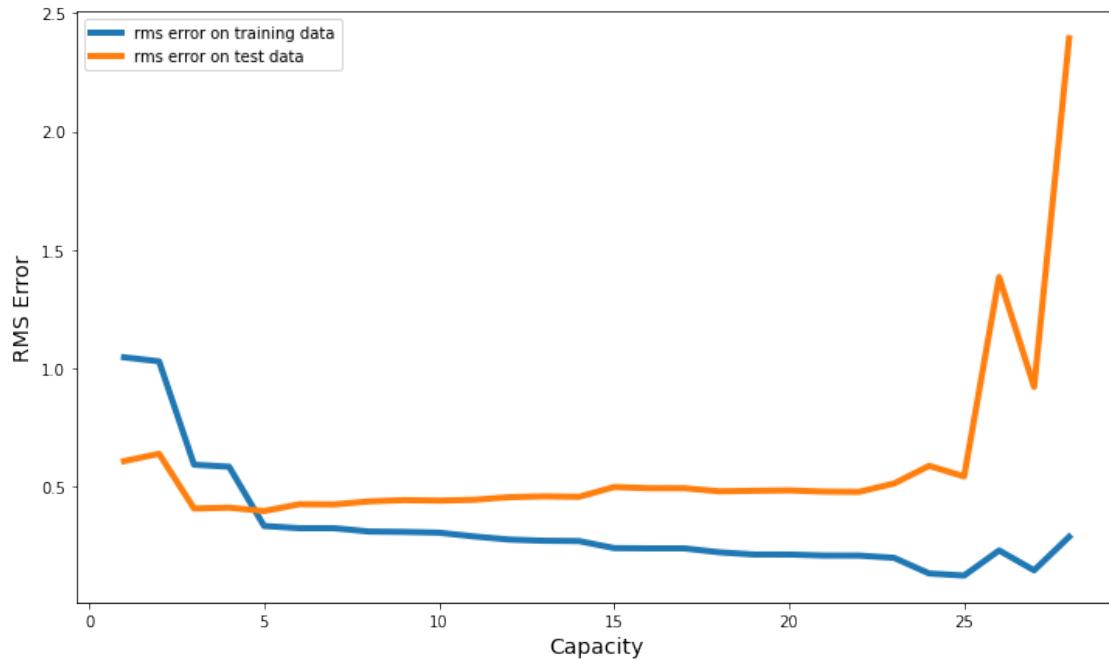
→y_test)**2)))

degrees = [i for i in range(1, max_degree+1)]

plt.plot(degrees, rms_error      , lw=4, label='rms error on training data')
plt.plot(degrees, rms_error_test, lw=4, label='rms error on test data')
plt.xlabel("Capacity", fontsize=14)
plt.ylabel("RMS Error", fontsize=14)
plt.legend()

```

<matplotlib.legend.Legend at 0x7efcdd471be0>



On the right side again, we see a high **generalization error**, which means our model won't accurately represent data points that weren't used for training. To get a good capacity, the training error should be low, as well as the gap between the training and generalization error.

5.6 Bias and Variance

A good illustration of **bias** and **variance** can be gained by plotting 2d Gaussian distributions with different biases and variances:

```
import numpy as np
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 2, sharex='col', sharey='row', figsize=(12,12))

samples = 15

# biases
mu1 = [0.5, 0.5]
mu2 = [0.7, 0.7]
mu3 = [0.5, 0.5]
mu4 = [0.7, 0.7]

# variances (compare to Lecture 02-6)
C1 = np.diag([0.0002, 0.0002])
C2 = np.diag([0.0002, 0.0002])
C3 = np.diag([0.003, 0.003])
```

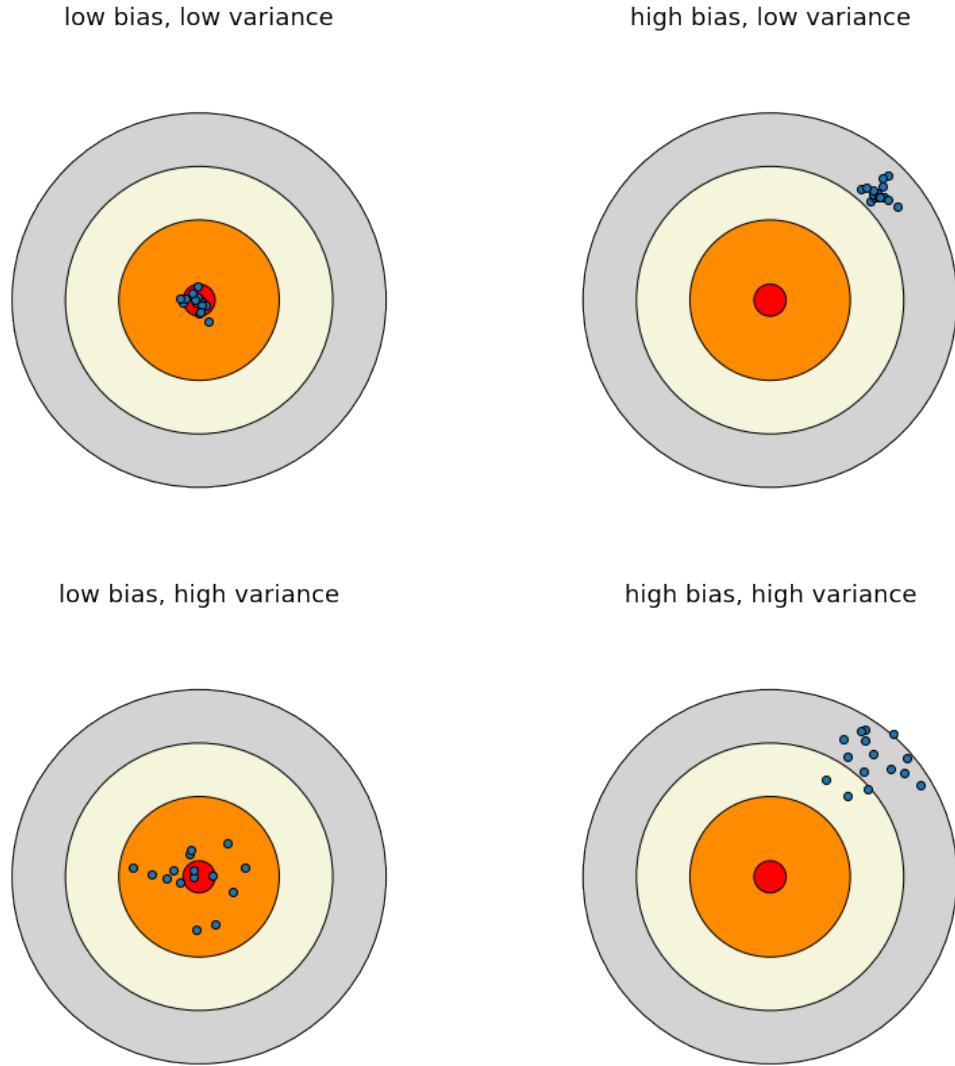
```
C4 = np.diag([0.003, 0.003])

# create data sets
distributions = [[np.random.multivariate_normal(mu1, C1, size=samples), \
                  np.random.multivariate_normal(mu2, C2, size=samples)], \
                  [np.random.multivariate_normal(mu3, C3, size=samples), \
                  np.random.multivariate_normal(mu4, C4, size=samples)]]


# set titles for plots
titles = [["low bias, low variance", \
           "high bias, low variance"], \
           ["low bias, high variance", \
           "high bias, high variance"]]

# plot everything
for i in range(2):
    for j in range(2):
        axs[i,j].add_artist(plt.Circle((0.5, 0.5), 0.35, ec='black', \
                                       fc='lightgray'))
        axs[i,j].add_artist(plt.Circle((0.5, 0.5), 0.25, ec='black', \
                                       fc='beige'))
        axs[i,j].add_artist(plt.Circle((0.5, 0.5), 0.15, ec='black', \
                                       fc='darkorange'))
        axs[i,j].add_artist(plt.Circle((0.5, 0.5), 0.03, ec='black', fc='red'))
        axs[i,j].scatter(distributions[i][j][:,0], distributions[i][j][:,1], \
                          ec='black', zorder=100)
        axs[i,j].set_xlim([0, 1])
        axs[i,j].set_ylim([0, 1])
        axs[i,j].set_title(titles[i][j], fontsize=18)
        axs[i,j].set_aspect(aspect='equal')
        axs[i,j].axis('off')

plt.tight_layout()
```



Here, **bias** is the distance of the distribution *mean* from the bullseye, **variance** is the distance of points from the mean of the distribution (recall the variance example distributions from 02-6, but imagine this in 2D).

The **complexity** of a model is basically the number of parameters and the basis functions used in the model. Both bias (dominant for low complexity) and variance (dominant for high complexity) contribute to the error in the whole model.

\hat{y} is also called a *statistical estimate of the true model y* . Then bias is the *expectation value* of the difference between the model prediction and the correct value (the hypothesis - the ground truth):

$$\text{Bias} = \sqrt{E[(\hat{y} - y)^2]} \quad (69)$$

This is the bias term. Variance is

$$\text{Variance} = \sqrt{E[(\hat{y} - \bar{y})^2]} \quad (70)$$

Both terms contribute to the overall error of the model. The third contribution comes from **noise** in the data itself (later). The sum of these three errors explains the model error.

```

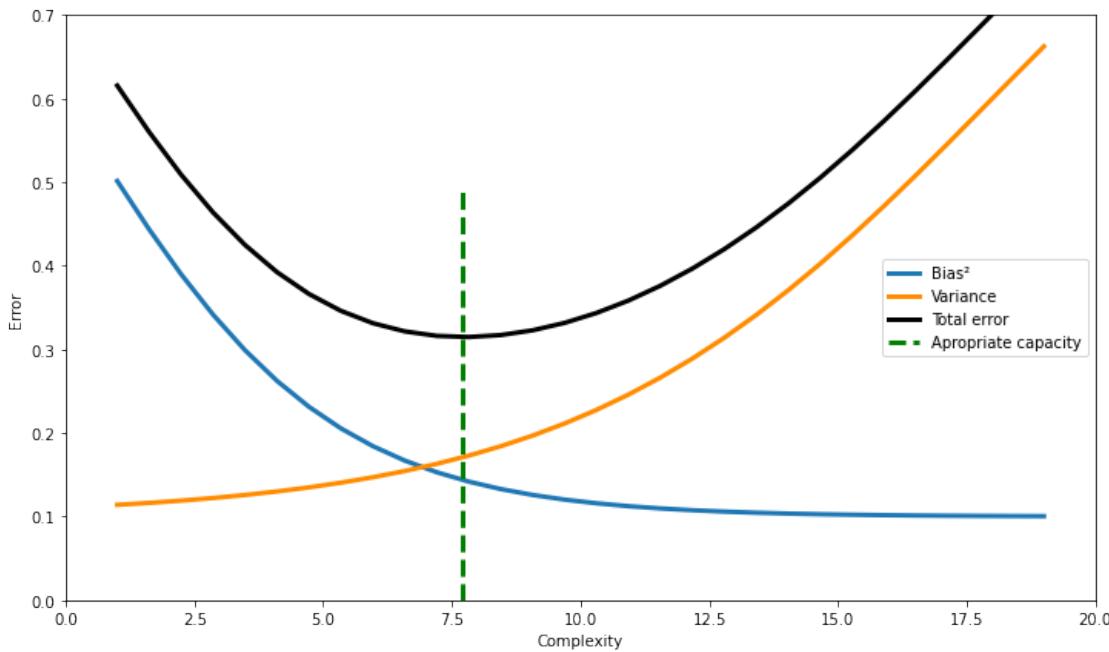
def sigmoid( , z):
    return 1/(1+np.exp(- *z))

x      = np.linspace(1, 19, 30)
Bias2   = sigmoid(0.4, -x)+0.1 # squared Bias
Variance = sigmoid(0.25, x-18)+0.1
error   = Bias2 + Variance

plt.figure(figsize=(12,7))
plt.plot(x, Bias2, lw=3, label="Bias²")
plt.plot(x, Variance, lw=3, color='darkorange', label="Variance")
plt.plot(x, error, lw=3, color='black', label="Total error")
plt.axvline(x=7.7, ymin=0, ymax=0.7, lw=3, linestyle='--', color='green', ↴
            label="Appropriate capacity")
plt.xlim([0, 20])
plt.ylim([0, 0.7])
plt.xlabel("Complexity")
plt.ylabel("Error")
plt.legend()

```

<matplotlib.legend.Legend at 0x7fa3d7535a58>



Simple models with few parameters usually show high bias, but low variance, and complex models with many parameters usually have small bias, but high variance.

```

from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

fig = plt.figure(figsize=(15, 7))
ax1 = fig.add_subplot(121)
ax2 = fig.add_subplot(122)

samples = 15
max_degree = 15
noise_level = 0.3

noise = np.random.normal(0, noise_level, samples)
x = np.linspace(-np.pi, np.pi, samples)
y = np.sin(x) + noise

# for testing the generalization capability of the model,
# some extra data points are generated from the same distribution
# usually, this is done differently, but we'll come back to this later
noise_test = np.random.normal(0, noise_level, 10)
x_test = np.linspace(-np.pi, np.pi, 10)
y_test = np.sin(x_test)

regression_result = []

```

```

x_poly = []
x_test_poly = []
# R2 scores are used in a later plot
R2 = []
R2_test = []
rms_error = []
rms_error_test = []

poly = [PolynomialFeatures(degree=degree) for degree in range(1, max_degree+1)]

for degree in range(1, max_degree+1):
    x_poly.append(poly[degree-1].fit_transform(x.reshape(-1, 1), y))
    x_test_poly.append(poly[degree-1].fit_transform(x_test.reshape(-1, 1), y))
    lin = LinearRegression()
    lin.fit(x_poly[degree-1], y)
    regression_result.append(lin)

    R2.append(regression_result[degree-1].score(x_poly[degree-1], y))
    R2_test.append(regression_result[degree-1].score(x_test_poly[degree-1], y))

    rms_error.append(np.sqrt(2/samples * np.sum((regression_result[degree-1].predict(x_poly[degree-1]) - y)**2)))
    rms_error_test.append(np.sqrt(2/samples * np.sum((regression_result[degree-1].predict(x_test_poly[degree-1]) - y)**2)))

def plot_reg(ax, degree, plot_sine, plot_test):
    #plt.cla()
    ax.scatter(x, y, color = 'steelblue')
    ax.plot(x, regression_result[degree-1].predict(x_poly[degree-1]), \
            color = 'darkorange', lw=3)

    if plot_sine:
        ax.plot(x, np.sin(x), color='green', linestyle="--", lw=3)

    if plot_test:
        ax.scatter(x_test, y_test, color='red', zorder=100)

    #ax.set_aspect(aspect='equal')

    print("Training R2 = ", R2[degree-1])

```

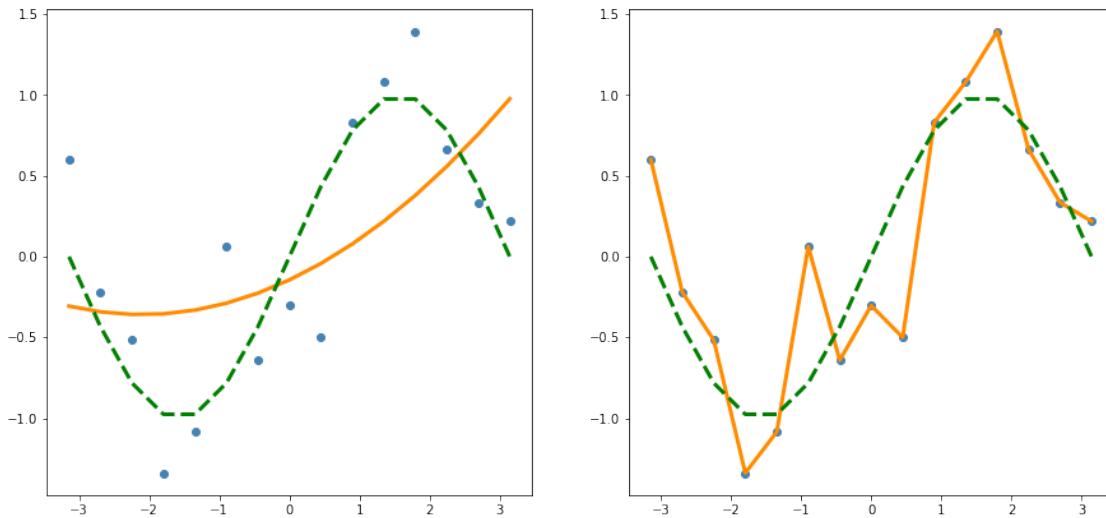
```

print("Test R2      = ", R2_test[degree-1])

plot_reg(ax1, 2, True, False)
plot_reg(ax2, 15, True, False)

Training R2 = 0.3128691543770026
Test R2     = 0.33917733546613493
Training R2 = 1.0
Test R2     = -16.40278621715949

```



Bias and variance can be measured in data. One way to do it, is to **bootstrap** a dataset into variants. For this, multiple sets of samples are chosen from some distribution. In the sine example, for each bootstrap say 10 points are chosen from the whole set. Then, each bootstrap data set is split into a training set T_b and a test set S_b . The same model (e.g. a quadratic model) is then trained for each bootstrap data set and tested on the respective test set. Then, for each sample $(x^{(i)}, y^{(i)})$ there are several predictions $\hat{y}_1(x^{(i)}), \hat{y}_2(x^{(i)}), \dots$, such that we can estimate the bias as

$$\text{Bias} = \sqrt{\text{average}[\hat{y}_1(x^{(i)}), \dots, \hat{y}_n(x^{(i)})] - y}^2 \quad (71)$$

and the variance as just the variance of the set $\hat{y}_1(x^{(i)}), \dots, \hat{y}_n(x^{(i)})$. This is obviously not feasible for large data sets. In practice, this is done differently.

There is no consensus on where to set the ratios, but usually, existing data is split into 60% **training data**, 20% **validation data**, and 20% **test data**. Usually, training and validation error are plotted simultaneously for each training step to spot these errors.

```

# this is not a real example, just a qualitative graph
x          = np.linspace(1, 19, 30)
error_train = sigmoid(0.4, -x)

```

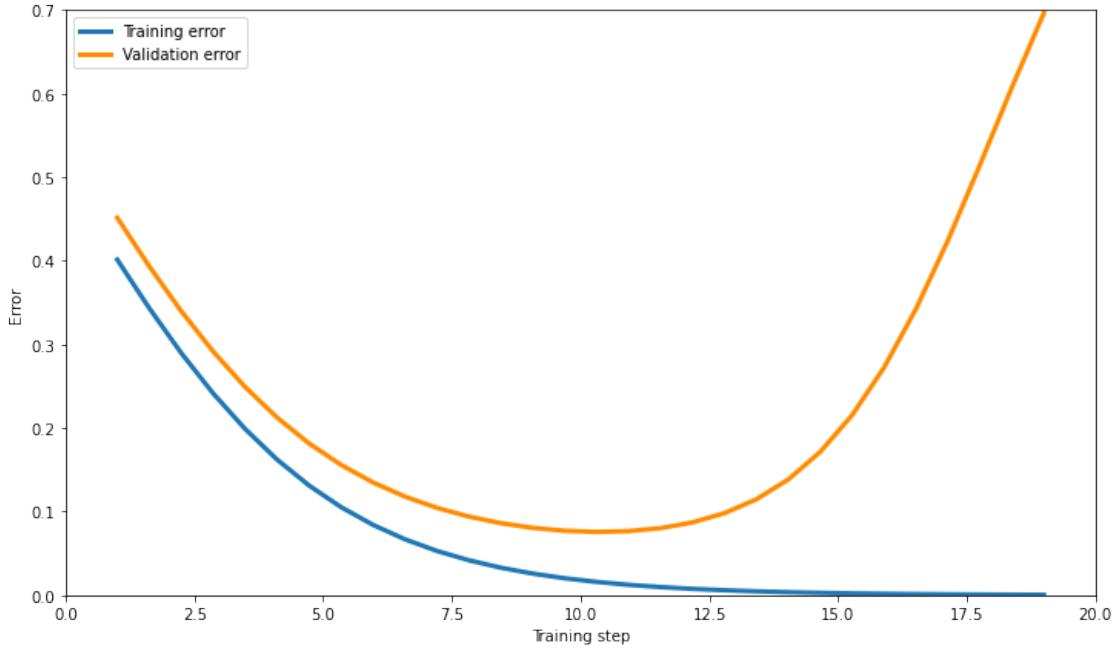
```

error_valid = error_train + sigmoid(0.6, x-18) + 0.05

plt.figure(figsize=(12,7))
plt.plot(x, error_train, lw=3, label="Training error")
plt.plot(x, error_valid, lw=3, color='darkorange', label="Validation error")
plt.xlim([0, 20])
plt.ylim([0, 0.7])
plt.xlabel("Training step")
plt.ylabel("Error")
plt.legend()

```

<matplotlib.legend.Legend at 0x7fa3d7113d68>



When both values are high, there is a high bias problem. If training error is very low, but the validation error very high, then there is a high variance problem (overfitting).

5.7 Regularization

Regularization can be used to soothe overfitting, although it can also lead to bias and variance problems again. The basic idea of regularization is to **penalize** large coefficient values:

$$J = (y - \sum_i^N w_i x_i)^2 + \lambda \|\mathbf{w}\|_2^2 \quad (72)$$

where the norm is the 2-norm. If we look at the sine example again, we see that this regularized

version gives a much smoother result:

```

from sklearn.linear_model import Ridge
from ipywidgets import interact
import ipywidgets as widgets

plt.figure(figsize=(15, 7))

samples = 15
max_degree = 15
noise_level = 0.3
ln_min = -20
ln_max = 20

noise = np.random.normal(0, noise_level, samples)
x = np.linspace(-np.pi, np.pi, samples)
y = np.sin(x) + noise

noise_test = np.random.normal(0, noise_level, 10)
x_test = np.linspace(-np.pi, np.pi, 10)
y_test = np.sin(x_test)

regression_result = [[None for i in range(ln_min+ln_max, 2*ln_max+1)] for j in range(max_degree+1)]

x_poly = []
x_test_poly = []

# R2 scores and rms_errors are used in a later plot
R2 = [[None for i in range(ln_min+ln_max, 2*ln_max+1)] for j in range(max_degree+1)]
R2_test = [[None for i in range(ln_min+ln_max, 2*ln_max+1)] for j in range(max_degree+1)]
rms_error = [[None for i in range(ln_min+ln_max, 2*ln_max+1)] for j in range(max_degree+1)]
rms_error_test = [[None for i in range(ln_min+ln_max, 2*ln_max+1)] for j in range(max_degree+1)]

# setup all polynomial spaces
poly = [PolynomialFeatures(degree=degree) for degree in range(1, max_degree+1)]

# precalculate all solutions
for degree in range(1, max_degree+1):
    x_poly.append(poly[degree-1].fit_transform(x.reshape(-1, 1), y))
    x_test_poly.append(poly[degree-1].fit_transform(x_test.reshape(-1, 1), y_test))
    for ln in range(ln_min, ln_max+1):
        regression_result[ln][j] = Ridge().fit(x_poly[ln], y).predict(x_test_poly[ln])

```

```

lin = Ridge(np.exp(ln), normalize=True)
lin.fit(x_poly[degree-1], y)

regression_result[degree][ln+ln_max] = lin

R2[degree][ln+ln_max] = regression_result[degree][ln+ln_max].score(x_poly[degree-1], y)
R2_test[degree][ln+ln_max] = regression_result[degree][ln+ln_max].score(x_test_poly[degree-1], y_test)

rms_error[degree][ln+ln_max] = np.sqrt(2/samples * np.sum((regression_result[degree][ln+ln_max].predict(x_poly[degree-1]) - y)**2))
rms_error_test[degree][ln+ln_max] = np.sqrt(2/samples * np.sum((regression_result[degree][ln+ln_max].predict(x_test_poly[degree-1]) - y_test)**2))

def plot_reg_reg(ln, degree, plot_sine, plot_test):
    #plt.cla()
    plt.scatter(x, y, color = 'steelblue', label=r"Training data")
    plt.plot(x, regression_result[degree][ln+ln_max].predict(x_poly[degree-1]), \
              color = 'darkorange', lw=3, label=r"Model prediction")
    plt.fill_between(x, np.sin(x) - noise_level, np.sin(x) + noise_level,
                     color='gray', alpha=0.2, label=r"Noise standard deviation")

    if plot_sine:
        plt.plot(x, np.sin(x), color='green', linestyle="--", lw=3, \
                  label=r"Ground truth")

    if plot_test:
        plt.scatter(x_test, y_test, color='red', zorder=100, label=r"Test data")

    print("Training R2 = ", R2[degree][ln+ln_max])
    print("Test R2      = ", R2_test[degree][ln+ln_max])

    plt.legend()
    plt.savefig("img/plot_regularization.png")

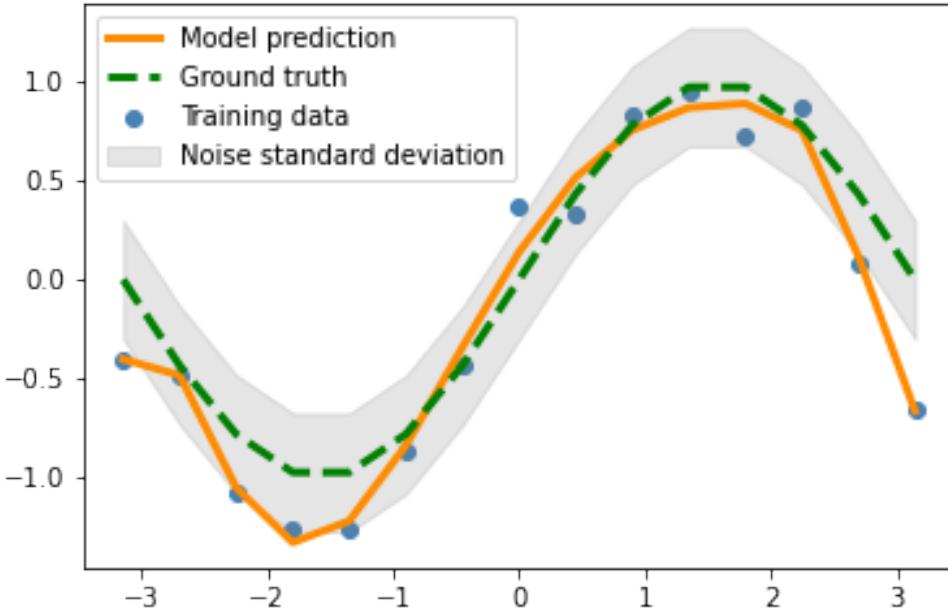
interact(plot_reg_reg, ln=(ln_min, ln_max), degree=(1, max_degree), \
         plot_sine=False, plot_test=False)

```

<Figure size 1080x504 with 0 Axes>

```
interactive(children=(IntSlider(value=0, description='ln ', max=20, min=-20), IntSlider(value=8,
```

```
<function __main__.plot_reg_reg(ln , degree, plot_sine, plot_test)>
```



λ is also a hyperparameter and as such, chosen before starting the training. Since the actual λ would be quite high here, its natural logarithm is given instead for ease of display. Choosing a large negative $\ln\lambda$, we get the unregularized model again, but showing high bias, roughly displaying a diagonal line. A higher λ helps closing the gap between training and validation error. Tuning it controls the **bias-variance tradeoff**:

```
plt.figure()

def plot_rms(degree):
    #plt.cla()
    plt.plot([ln for ln in range(ln_min, ln_max+1)], rms_error[degree], lw=4, u
             ↳label=r"rms error training")
    plt.plot([ln for ln in range(ln_min, ln_max+1)], rms_error_test[degree], u
             ↳lw=4, label=r"rms error test")

    plt.xlabel(r"\ln\lambda")
    plt.ylabel(r"rms error")

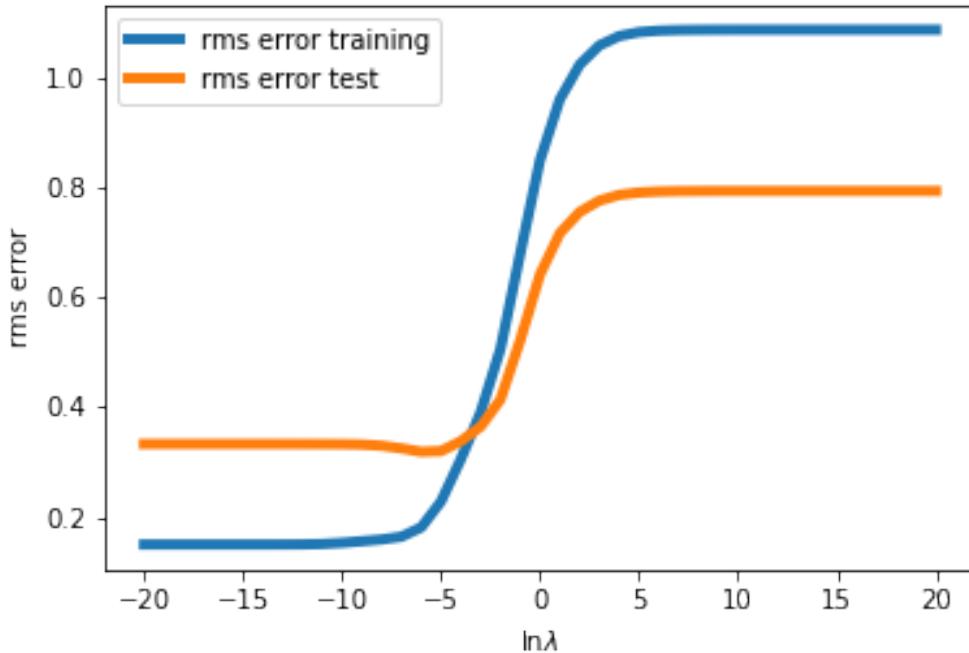
    plt.legend()
```

```
plt.savefig("img/plot_bias-variance.png")
interact(plot_rms, degree=(1, max_degree))
```

<Figure size 432x288 with 0 Axes>

```
interactive(children=(IntSlider(value=8, description='degree', max=15, min=1), Output()), _dom_
```

```
<function __main__.plot_rms(degree)>
```



The most commonly used norms are the $L1$ -norm, where instead of the 2 -norm the 1 -norm is used, and the $L2$ -norm from above, or the sum of both of them. It is also possible to regularize only certain weights w_i , by only penalizing these individual weights.

We will get to know more regularization options later.

5.8 Gradient Descent Variants

There are several ways to perform gradient descent:

- **Batch Gradient Descent:** calculate gradient of the cost function using the entire data set to make parameter updates. Online updates impossible.

- **Stochastic Gradient Descent:** parameter updates are calculated for *every* training sample, so online learning is possible. This causes large oscillations in the loss function due to the frequent updates.
- **Mini-batch Gradient Descent:** Combination of both. Performs update for a mini-batch of the training data.

The original gradient descent algorithm is rarely used, due to several problems it produces. Instead, you'll most probably find one of these variants in practice:

5.8.1 Momentum Method

$$\Delta \mathbf{w}^{(k+1)} = \gamma \Delta \mathbf{w}^{(k)} + \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)}) \quad (73)$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \Delta \mathbf{w}^{(k)} \quad (74)$$

Here, a fraction γ of the previous update is added to the current update, which helps taking larger steps in the relevant direction and prevents oscillations commonly found in normal stochastic gradient descent. γ is the scaling factor, a new hyperparameter that needs to be chosen.

5.8.2 Nesterov Accelerated Gradient

$$\Delta \mathbf{w}^{(k+1)} = \gamma \Delta \mathbf{w}^{(k)} + \alpha \nabla_{\mathbf{w}} J(\mathbf{w}^{(k)} + \gamma \Delta \mathbf{w}^{(k)}) \quad (75)$$

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \Delta \mathbf{w}^{(k)} \quad (76)$$

The loss gradient is now evaluated at a *lookahead* point, that is determined using the current gradient.

5.8.3 AdaGrad

$$w_i^{(k+1)} = w_i^{(k)} + \frac{\alpha}{\sqrt{G_i^{(k)} + \varepsilon}} g_i^{(k)} \quad (77)$$

AdaGrad uses a different learning rate for each parameter, where every update is still computed by a gradient. The *general learning rate* α is scaled at a particular iteration by using accumulated squared gradients from previous iterations. This leads to rapidly diminishing updates. $g_i^{(k)} = \nabla_{w_i} J$ is the partial derivative of J with respect to the component of \mathbf{w} at the current iteration, $G_i^{(k)} = \sqrt{(g_i^{(1)})^2 + (g_i^{(2)})^2 + \dots + (g_i^{(k)})^2}$ is the root accumulated squared gradient sum. This approach *normalizes* the base learning rate for each parameter, since otherwise some parameters will have very large and frequent, other parameters very small and rare updates. ε is a small value that makes sure no division by zero occurs.

The disadvantage here is that the denominator will become very large with increasing iteration numbers, so that when this doesn't converge quickly, updates will become extremely small.

5.8.4 RMSProp and AdaDelta

$$w_i^{(k+1)} = w_i^{(k)} + \frac{\alpha}{\sqrt{E[g^2]^{(k+1)} + \varepsilon}} g_i^{(k+1)} \quad (78)$$

$$E[g^2]^{(k+1)} = \rho E[g^2]^{(k)} + (1 - \rho)(g^{(k)})^2 \quad (79)$$

This is an extension of AdaGrad, where instead of storing the running sum of squared gradients, the weighted average of the squared gradients is used. The current average depends on the average from the previous iteration instead of $G_i^{(k)}$. $E[g^2]^{(k)}$ is a weighted running sum, $g^{(k)}$ is the current gradient. $\rho \in [0, 1]$ is the weight. In the first iteration $E[g^2]^{(k)} = 0$, so ε must be nonzero. This causes exponentially decreasing weighted average learning rates.

The only difference is that in AdaDelta, the base learning rate is also scaled with the weighted average updates of the last step. This will make sure that units match, given that the weights do have a unit.

$$w_i^{(k+1)} = w_i^{(k)} + \frac{\alpha E[(\Delta w)^2]^{(k)}}{\sqrt{E[g^2]^{(k+1)} + \varepsilon}} g_i^{(k+1)} \quad (80)$$

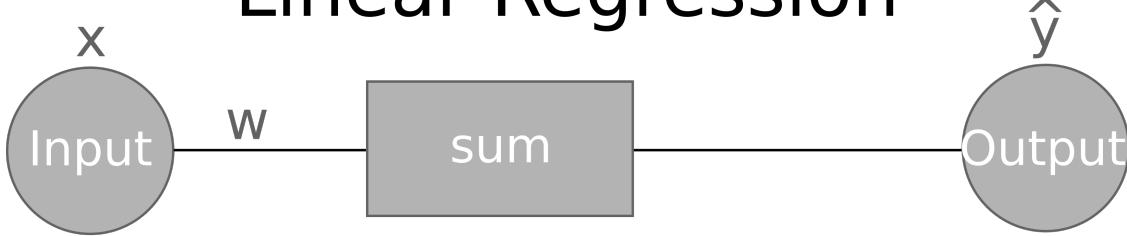
$$E[g^2]^{(k+1)} = \rho E[g^2]^{(k)} + (1 - \rho)(g^{(k)})^2 \quad (81)$$

5.9 Logistic Regression

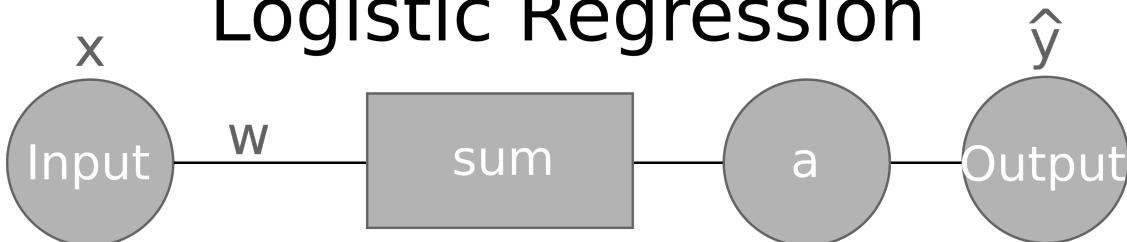
In linear regression, we connected inputs \mathbf{x} to outputs \mathbf{y} by a linear map, so $h(\mathbf{x}; \mathbf{w})$ is a linear function. We had a ground truth y and predictions \hat{y} , calculated the gradient of the loss function, optimized the loss function, and that was pretty much it. Here, we make a small, but crucial, addition to that recipe:

```
from IPython.display import Image
Image("img/lin-log-reg.png", width="700")
```

Linear Regression



Logistic Regression



a is a nonlinear **activation function** $a(x) = \sigma(Wx + b)$.

In linear regression, we had real-valued output data. Sometimes we need more qualitative outputs, like when deciding whether an image of an airplane hull shows a crack or not, or whether something produced at a factory is ready for shipping or defective. We need to think of a way to assign a number to these **classes**. Choosing a nonlinear a is not straight-forward, as is true for the loss function J and its gradient.

```

import numpy as np
from sklearn.datasets import make_blobs
import matplotlib.pyplot as plt
from ipywidgets import interact
import ipywidgets as widgets

x = np.linspace(-8, 8, 20)

fig = plt.figure(figsize=(8,8))
#ax = fig.add_subplot(111)

X1, Y1 = make_blobs(n_features=2, centers=2)

def plot_line(w_0, w_1):
    plt.cla()
  
```

```

plt.scatter(X1[:, 0], X1[:, 1], marker='o', c=Y1,
            s=100, edgecolor='k')

plt.plot(x, w_0 + w_1*x, lw=4, color='darkorange')
plt.xlim(min(X1[:,0]), max(X1[:,0]))
plt.ylim(min(X1[:,1]), max(X1[:,1]))
plt.xlabel("Feature 1", fontsize=20)
plt.ylabel("Feature 2", fontsize=20)

plt.savefig("img/plot_linreg.png")

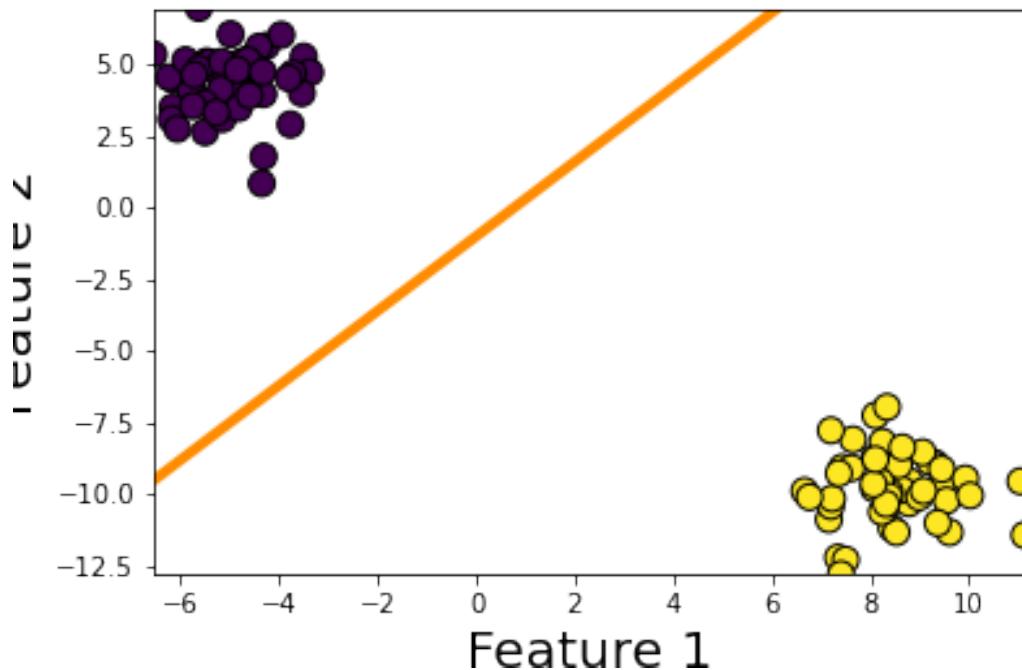
interact(plot_line, w_0=(-20.0, 20.0), w_1=(-20.0, 20.0))

```

<Figure size 576x576 with 0 Axes>

interactive(children=(FloatSlider(value=0.0, description='w_0', max=20.0, min=-20.0), FloatSlider

<function __main__.plot_line(w_0, w_1)>



This is a **binary classification problem**. We don't want to make any quantitative statements about the data, but instead want to assign labels to each point and generalize this to new data

points (in other words, drawing a new sample from the same distribution should make our algorithm decide, say, which urn something came from). If the data is nicely clustered as in the graph above, we can simply draw a **separating/classifying line** between the clusters, but we can also use the linear regression idea for finding such a line, by assigning each class a **label** of either 0 or 1. So we are given N examples x_1, x_2, \dots with ground truth $y_i \in \{0, 1\}$.

```
from IPython.display import display
import pandas as pd

headers = [r"$$\text{Example}$$", r"$$\text{Features } (x_1, x_2)$$", r"$$\text{GT } y$$"]
data = [[r"$1$", r"$$x_1^{(1)}$$", r"$$x_2^{(1)}$$", r"$0$]", r"$0.11$"], \
         [r"$2$", r"$$x_1^{(2)}$$", r"$$x_2^{(2)}$$", r"$1$]", r"$0.98$"], \
         [r".", r"...", r"...", r"..."], \
         [r".", r"...", r"...", r"..."], \
         [r"N", r"$$x_1^{(N)}$$", r"$$x_2^{(N)}$$", r"$0$]", r"$0.07$"]]

display(pd.DataFrame(data, columns=headers))
```

	\$\$\text{Example}\$\$	\$\$\text{Features } (x_1, x_2)\$\$	\$\$\text{GT } y\$\$
0	\$1\$	\$\$x_1^{(1)}\$\$, $x_2^{(1)}$	\$0\$
1	\$2\$	\$\$x_1^{(2)}\$\$, $x_2^{(2)}$	\$1\$
2
3
4	\$N\$	\$\$x_1^{(N)}\$\$, $x_2^{(N)}$	\$0\$

	\$\$\text{H } \hat{y}\$\$
0	\$0.11\$
1	\$0.98\$
2	...
3	...
4	\$0.07\$

In the end we also want to do this for newly chosen examples. If we introduced a new point in the far right top in the graph above, it will have a very high value, when linear regression is used, and not something between 0 and 1. A linear model never reliably gives values between 0 and 1. So the idea here is to compress all values to the range $[0, 1]$ by using the **sigmoid function** $\sigma(z) = \frac{1}{1+\exp(-\beta z)}$, which looks like this:

```
def sigmoid( , z):
    return 1/(1 + np.exp(- * z))

x = np.linspace(-8, 8, 20)

plt.figure(figsize=(12,7))

def plot_sig():
```

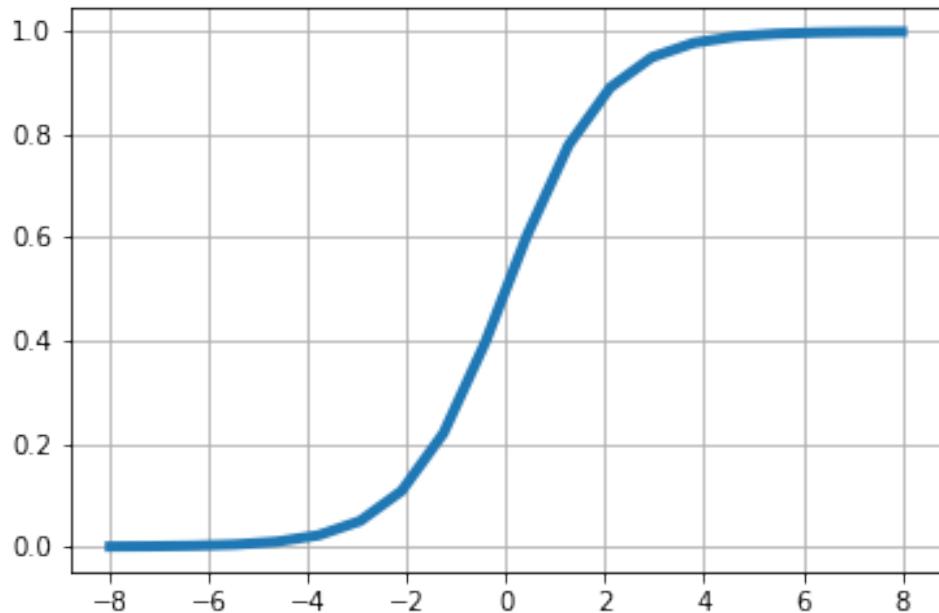
```
#plt.cla()
plt.plot(x, sigmoid( , x), lw=4)
plt.grid(True)
plt.savefig("img/plot_sigmoid.png")

interact(plot_sig, =(-1.0, 3.0))
```

<Figure size 864x504 with 0 Axes>

interactive(children=(FloatSlider(value=1.0, description=' ', max=3.0, min=-1.0), Output()), _d

<function __main__.plot_sig()>



For $z \rightarrow -\infty$, σ approaches 0, while for $z \rightarrow \infty$ it approaches 1, as we desired. At $t = 0$, $\sigma(0) = 0.5$. This is a monotonic function. Sometimes a scaling factor β is also set, but mostly it's omitted.

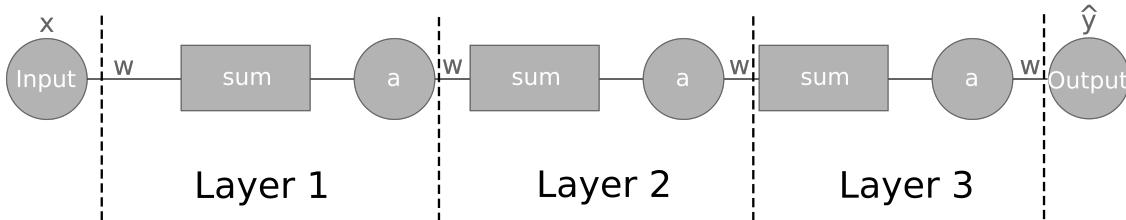
We can now interpret results of our hypothesis $h(\mathbf{x}; \mathbf{w}) = \sigma(z) \in (0, 1)$ (where $z = w_0 + w_1 x_1 + w_2 x_2$) as *probabilities* $p(y = 1|\mathbf{x})$. For new points to be in class 1, z must have a very high value while if it belongs to class 0, z must be really low, both of which become more probable the further away this new point is from the separating line. The separating line is the line $z = 0$. The closer a point lies to this line, the more uncertain we are about which class it belongs to.

5.9.1 Deep Learning Teaser

Look at the following schematic, which is simply repeated logistic regression:

```
Image("img/dl.png", width="800")
```

Deep Learning



Iterated logistic regression like this can already be thought of as a *neural network* with 3 layers. Neural networks with more than one layer are called *deep neural networks*. While this might seem like a vast oversimplification, there really isn't a whole lot more to it. We will talk about neural networks in the next lecture.

6 Logistic Regression and Artificial Neural Networks

6.1 Binary Cross-Entropy Cost Function

The setup in binary classification problems, or logarithmic regression, is that the hypothesis now has *nonlinear* function applied to the accumulated weighted inputs $\hat{y} = \sigma(\mathbf{w} \cdot \mathbf{x})$, with $\mathbf{w} = [w_0, w_1, \dots, w_F]$ and $\mathbf{x} = [1, x_1, x_2, \dots, x_F]$ (sometimes instead of 1, x_0 is used to indicate the bias term). The cost function we used in linear regression $J = \frac{1}{2N}(y - \hat{y})^2$ is not appropriate here. The reason is that y now is *always* either 0 or 1: $y \in \{0, 1\}$, and $\hat{y} \in [0, 1]$. E.g., let $y = 0$ and $\hat{y} = 1$, then the cost for this misclassification is only $J = \frac{1}{2N}$, which is extremely low. A better cost function would turn the cost of such a misclassification up extremely.

Instead, a function with the following features is preferred:

- $J = 0$ for $y = \hat{y}$
- $J \gg 0$ for $y \neq \hat{y}$
- $J \geq 0$ always

The **binary cross-entropy** cost function satisfies these demands. It's defined as

$$J = -[y \ln(\hat{y}) + (1 - y) \ln(1 - \hat{y})] \quad (82)$$

Since the sigmoid function is used as the activation function here, $\hat{y} \in [0, 1] \forall \mathbf{w}, \mathbf{x}$. Labels are always $y \in \{0, 1\}$. Using the above example for misclassification in this formula, the first term vanishes, since $y = 0$. The second term becomes $J = (1 - 0) \ln(1 - 1)$, and $\lim_{x \rightarrow 0} \ln(x) = -\infty$, so this becomes extremely large, the closer \hat{y} is to 1. This also works the other way, where $y = 1$ and $\hat{y} = 0$. You can test this in the graph below:

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact

plt.figure(figsize=(8,5))

def BCE(y, y_hat):
    return -(y*np.log(y_hat) + (1-y)*np.log(1-y_hat))

y_hat = np.linspace(0.02, 0.98, 50)

def plot_BCE(y):
    #plt.cla()
    plt.plot(y_hat, BCE(y, y_hat), lw=4)
    plt.xlabel(r"\hat{y}", fontsize=16)
    plt.ylabel(r"J", fontsize=16)

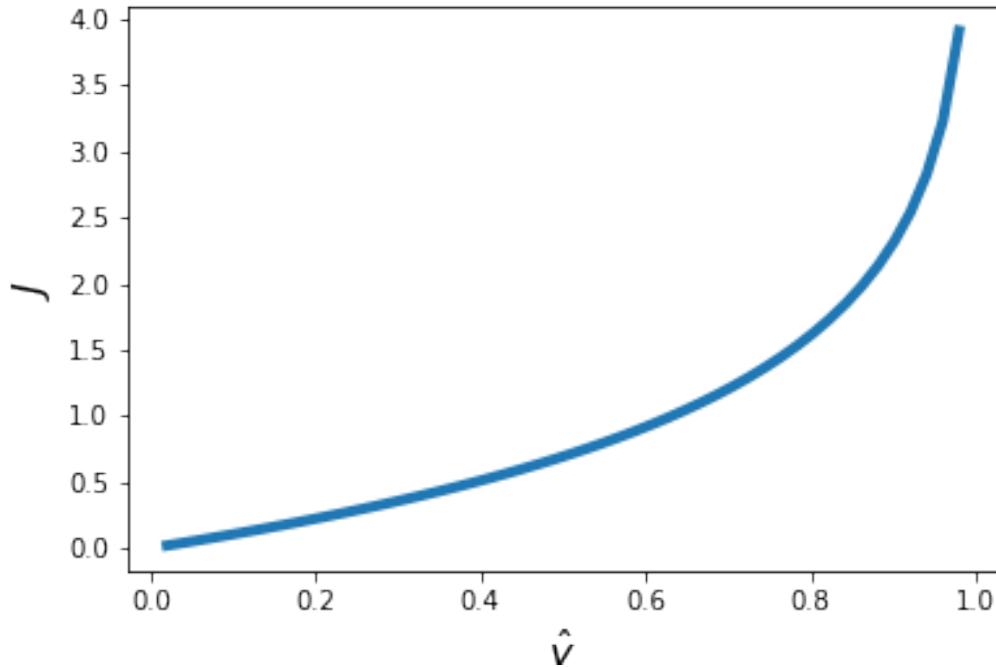
    plt.savefig("img/plot_bce.png")

interact(plot_BCE, y=(0,1))
```

<Figure size 576x360 with 0 Axes>

```
interactive(children=(IntSlider(value=0, description='y', max=1), Output()), _dom_classes='wid
```

```
<function __main__.plot_BCE(y)>
```



6.2 Logic Gates

To see the limitations of logistic regression for classification problems, simple *logic gates* are a good and simple way to evaluate and visualize what is going on behind the scenes. Logic gates come from computer science and process a number of binary inputs to a number of binary outputs. We will only look at simple logic gates taking 2 bits as input and producing 1 bit of output. They are usually described in a logic table like this:

6.2.1 OR gate

```
import numpy as np
import pandas as pd

def z(w_0, w_1, w_2, x_1, x_2):
    return w_0 + w_1*x_1 + w_2*x_2

def sigmoid(z):
    return 1/(1+np.exp(-z))

w_or = [-1.0, 2.0, 2.0]

headers = [r"$$x\_1, x\_2$$", r"$$y$$", r"$$z$$", r"$$\hat{y} = \text{sig}(z)$$"]
```

```
data_or = [[str(i)+" "+str(j), i or j, z(*w_or, i, j), sigmoid(z(*w_or, i, j))] for i in range(2) for j in range(2)]
```

```
table = pd.DataFrame(data_or, columns=headers)
```

```
table
```

	x_1	x_2	y	\hat{y}
0	0, 0	0	-1.0	0.268941
1	0, 1	1	1.0	0.731059
2	1, 0	1	1.0	0.731059
3	1, 1	1	3.0	0.952574

This gate checks whether *at least* one input bit is 1. These are called *logic tables*, because usually 0 and 1 denote *False* and *True*. We can plot the results as features in a 2d plane (recall that $z = w_0 + w_1x_1 + w_2x_2 = 0$ is the separating line):

```
import matplotlib.pyplot as plt
from ipywidgets import interact

inputs = np.array([[i, j] for i in range(2) for j in range(2)])
y_or = np.array([inp[0] or inp[1] for inp in inputs])

x = np.linspace(-0.1, 1.1, 20)

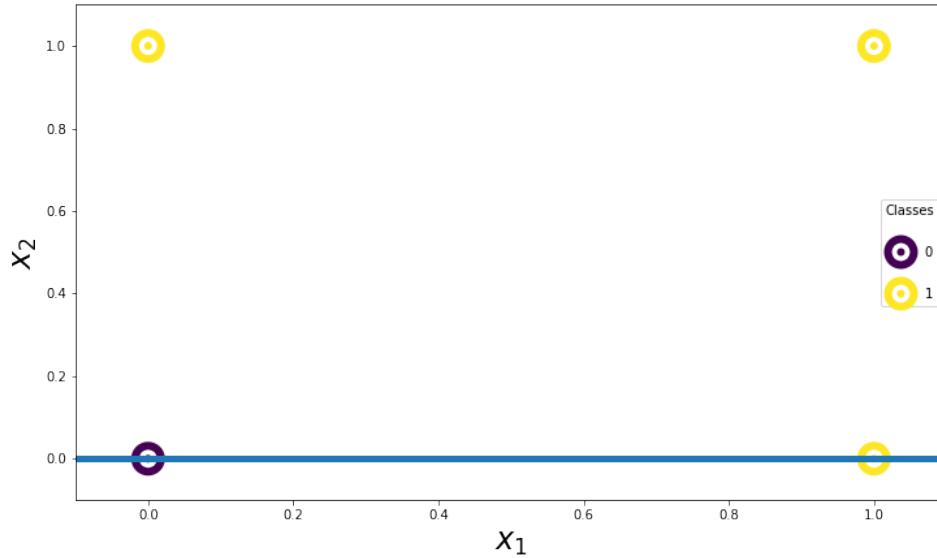
def plot_line(w_0, w_1, w_2):
    plt.figure(figsize=(12,7))
    if w_2 == 0:
        w_2 = 0.1
    #plt.cla()
    scatter = plt.scatter(inputs[:,0], inputs[:,1], lw=20, c=y_or)
    plt.plot(x, -(w_0+w_1*x)/w_2, lw=5)
    legend = plt.legend(*scatter.legend_elements(), labelspacing=2, \
                        loc="right", title="Classes")
    #plt.artist(legend)
    plt.xlabel(r"$x_1$", fontsize=24)
    plt.ylabel(r"$x_2$", fontsize=24)
    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.1, 1.1])

    plt.savefig("img/plot_OR.png")

interact(plot_line, w_0=(-10.0, 10.0), w_1=(-10.0, 10.0), w_2=(-10.0, 10.0))
```

```
interactive(children=(FloatSlider(value=0.0, description='w_0', max=10.0, min=-10.0), FloatSlider
```

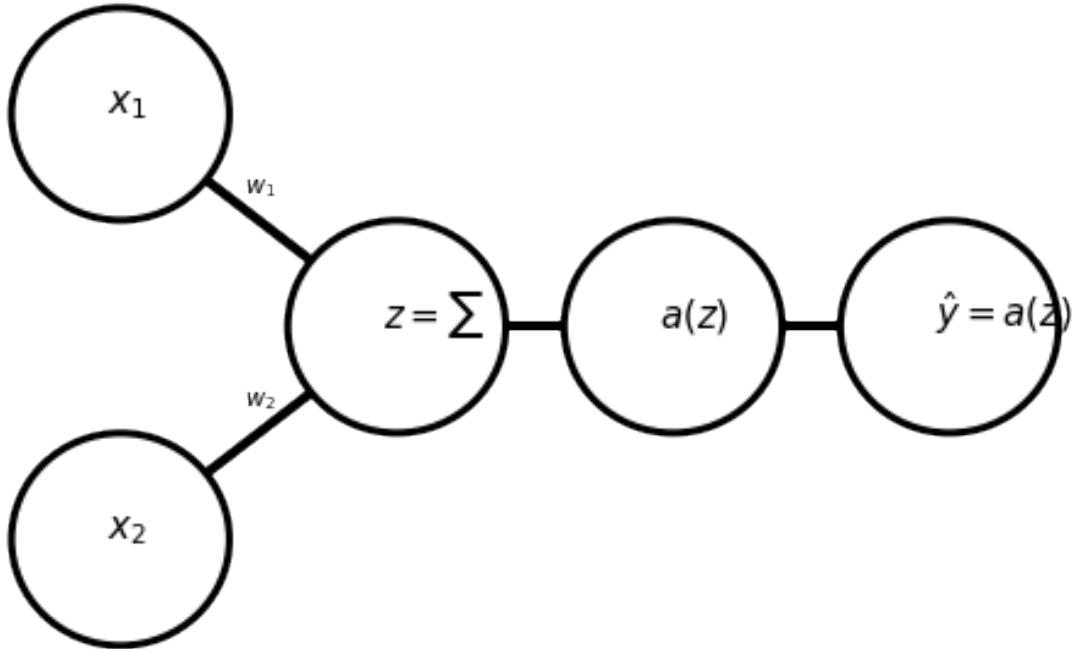
```
<function __main__.plot_line(w_0, w_1, w_2)>
```



Note that without a *bias* term w_0 , all possible lines would cross the origin and finding a separating line would be impossible.

```
import matplotlib.pyplot as plt
from scripts.draw_ann import draw_neural_net

label_list = [[r"$x\_1$"], [r"$x\_2$"], ["$z=\sum$"], [r"$a(z)$"], [r"$\hat{y} = \rightarrow a(z)$"]]
weight_list = [[[r"$w\_1$"], [r"$w\_2$"]], [[[""]], [[[""]]]]
fig = plt.figure(figsize=(8, 8))
ax = fig.gca()
ax.axis('off')
draw_neural_net(plt, ax, .12, .88, .1, .9, [2, 1, 1, 1], label_list, weight_list)
```



This is the *neural diagram* showing the forward pass in this example. When $\hat{y} \geq 0.5$, the example will be classified as belonging to class 1, and when $\hat{y} \leq 0.5$ the example will be classified as belonging to class 0.

So which weights will classify the correct examples as 0 or 1 respectively? With a linear separation $z = w_0 + w_1x_1 + w_2x_2$, we can get the line equation by setting $z = 0$, such that $-\frac{w_0 + w_1x_1}{w_2} = x_2$. So either w_0 or w_2 must be negative. Let's decide for a negative bias term $w_0 = -1$ here. Putting in an example, say, $(0, 1)$, shows that $w_0 + w_2$ must be positive, so let's take $w_2 = 2$. $w_1 = 2$ in that case. You can verify these numbers in the interactive plot above. These numbers are not unique. For example, you could multiply the line equation by some number and get the same line again. It's also possible to slightly rotate and translate the line and still get the classification correct.

6.2.2 NOR, AND, NAND

NOR is short for NOT(OR). Looking at the logic table, this gets completely opposite results compared to the OR gate above. Since this is the inverse of the OR gate, we can try flipping the weights to $w_0 = 1$, $w_1 = -2$, and $w_2 = -2$ and indeed, this works. Note that direction is important here. Everything above this line will belong to class 1, everything below will belong to class 0. This was reversed for the OR gate.

We can also do this for AND and NAND gates.

```
from IPython.display import display

x = np.linspace(-0.1, 1.1, 20)

w_nor = [1.0, -2.0, -2.0]
y_nor = np.array([1-(inp[0] or inp[1]) for inp in inputs])
data_nor = [[str(i)+" "+str(j), 1-(i or j), z(*w_nor, i, j), sigmoid(z(*w_nor, u
    →i, j))] for i in range(2) for j in range(2)]
w_and = [-3, 2, 2]
y_and = np.array([inp[0] and inp[1] for inp in inputs])
data_and = [[str(i)+" "+str(j), (i and j), z(*w_and, i, j), sigmoid(z(*w_and, u
    →i, j))] for i in range(2) for j in range(2)]
w_nand = [3, -2, -2]
y_nand = np.array([1 - (inp[0] and inp[1]) for inp in inputs])
data_nand = [[str(i)+" "+str(j), 1-(i and j), z(*w_nand, i, j), sigmoid(z(*w_nand, i, j))] for i in range(2) for j in range(2)]

def plot_line(w_0, w_1, w_2, y):
    if w_2 == 0:
        w_2 = 0.1
    #plt.clf()
    plt.scatter(inputs[:,0], inputs[:,1], lw=20, c=y)
    plt.plot(x, -(w_0+w_1*x)/w_2, lw=5)
    plt.xlabel(r"\$x_1\$", fontsize=16)
    plt.ylabel(r"\$x_2\$", fontsize=16)
    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.1, 1.1])

def choose_gate(gate, w_0, w_1, w_2):
    plt.figure()

    if gate == "OR":
        if w_0 == 0 and w_1 == 0 and w_2 == 0:
            print("Weights: ", w_or)
            plot_line(*w_or, y_or)
        else:
            plot_line(w_0, w_1, w_2, y_or)
```

```

table = pd.DataFrame(data_or, columns=headers)
display(table)

elif gate == "NOR":
    if w_0 == 0 and w_1 == 0 and w_2 == 0:
        print("Weights: ", w_nor)
        plot_line(*w_nor, y_nor)
    else:
        plot_line(w_0, w_1, w_2, y_nor)
table = pd.DataFrame(data_nor, columns=headers)
display(table)

elif gate == "AND":
    if w_0 == 0 and w_1 == 0 and w_2 == 0:
        plot_line(*w_and, y_and)
        print("Weights: ", w_and)
    else:
        plot_line(w_0, w_1, w_2, y_and)

table = pd.DataFrame(data_and, columns=headers)
display(table)

elif gate == "NAND":
    if w_0 == 0 and w_1 == 0 and w_2 == 0:
        plot_line(*w_nand, y_nand)
        print("Weights: ", w_nand)
    else:
        plot_line(w_0, w_1, w_2, y_nand)

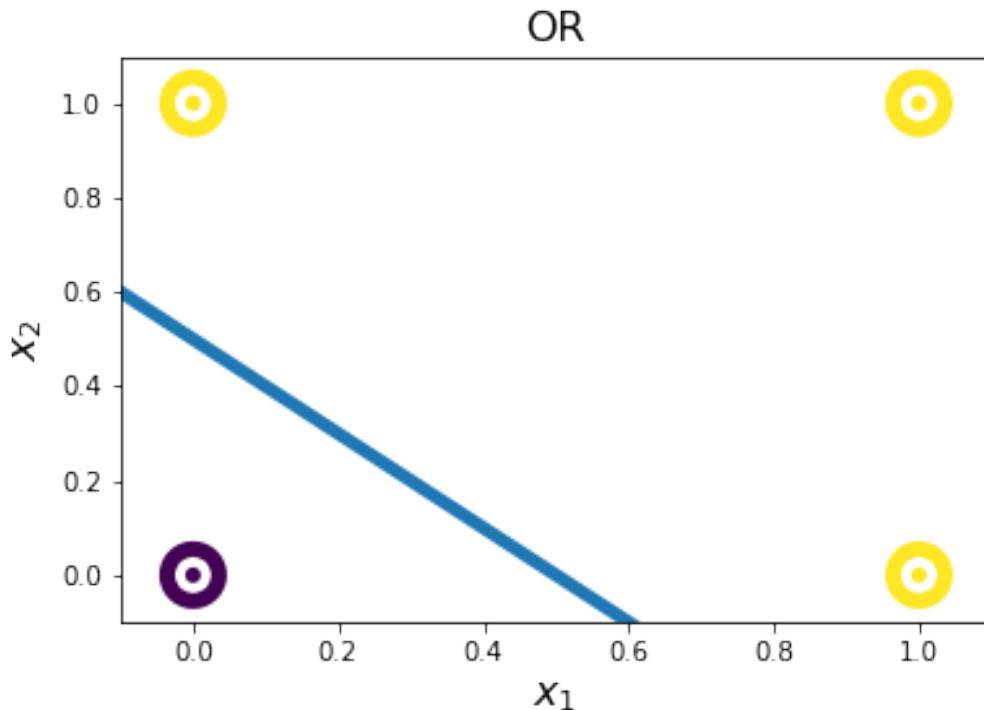
table = pd.DataFrame(data_nand, columns=headers)
display(table)

plt.title(gate, fontsize=16)
plt.savefig("img/plot_gates.png")

interact(choose_gate, gate=["OR", "NOR", "AND", "NAND"], \
         w_0=(-10.0, 10.0), w_1=(-10.0, 10.0), w_2=(-10.0, 10.0))

interactive(children=(Dropdown(description='gate', options=('OR', 'NOR', 'AND', 'NAND'), value=
<function __main__.choose_gate(gate, w_0, w_1, w_2)>

```



The combination of the weighted sum and the activation function is an **artificial neuron**.

Can all gates or circuits be represented by this simple network?

6.2.3 XOR gate

We can also try the approach from last lesson on the XOR gate, which returns 1 if *exactly* one of the inputs is 1:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
from ipywidgets import interact
import pandas as pd

def z(w_0, w_1, w_2, x_1, x_2):
    return w_0 + w_1*x_1 + w_2*x_2

def sigmoid(z):
    return 1/(1+np.exp(-z))

headers = [r"$$x\_1", r"x\_2$$", r"$$y$$", r"$$z$$", r"$$\hat{y} = \rightarrow \mathrm{sig}(z)$$"]
```

```

inputs = np.array([[i, j] for i in range(2) for j in range(2)])
y_xor = np.array([inp[0] ^ inp[1] for inp in inputs])

x = np.linspace(-0.1, 1.1, 20)

def plot_line(w_0, w_1, w_2, draw_nonlinear):
    fig = plt.figure()
    ax = fig.add_subplot(111)
    if w_2 == 0:
        w_2 = 0.1
    #ax cla()
    ax.scatter(inputs[:,0], inputs[:,1], lw=20, c=y_xor)
    ax.plot(x, -(w_0+w_1*x)/w_2, lw=5)
    ellipse = Ellipse([0.5, 0.5], 0.1+np.sqrt(2), np.sqrt(2)/2, -45, fc='w', ec='darkorange', lw=4, zorder=-10)
    if draw_nonlinear:
        ax.add_artist(ellipse)
    ax.set_xlabel(r"$x_1$", fontsize=16)
    ax.set_ylabel(r"$x_2$", fontsize=16)
    ax.set_xlim([-0.1, 1.1])
    ax.set_ylim([-0.1, 1.1])

    data = [[str(i)+", "+str(j), i ^ j, z(w_0, w_1, w_2, i, j), sigmoid(z(w_0, w_1, w_2, i, j))] for i in range(2) for j in range(2)]
    display(pd.DataFrame(data, columns=headers))

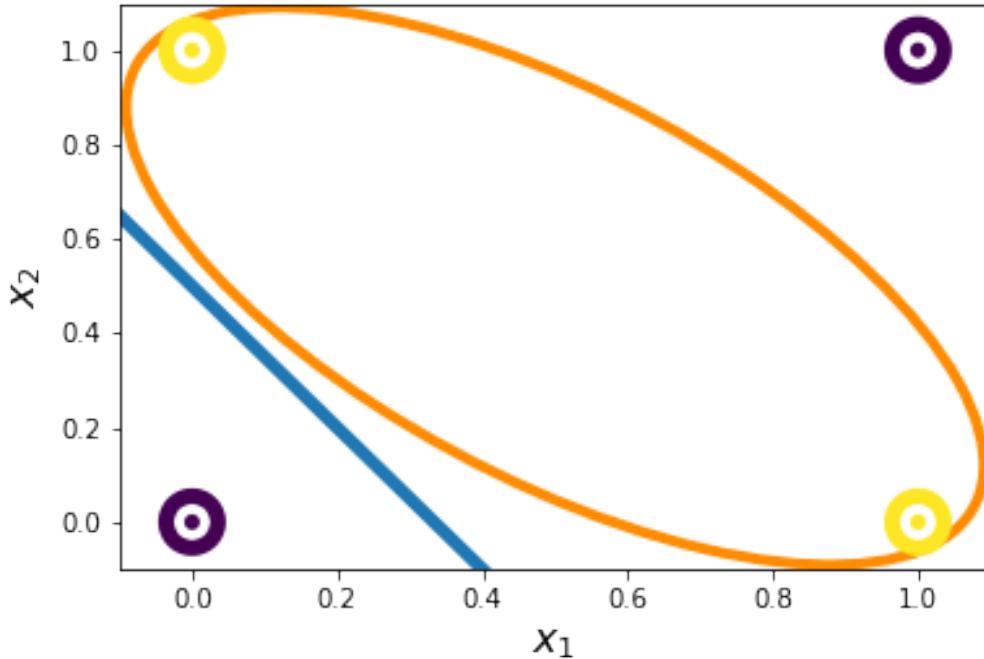
    plt.savefig("img/plot_XOR.png")

interact(plot_line, w_0=(-10.0, 10.0), w_1=(-10.0, 10.0), w_2=(-10.0, 10.0), draw_nonlinear=False)

interactive(children=(FloatSlider(value=0.0, description='w_0', max=10.0, min=-10.0), FloatSlider(value=0.0, description='w_1', max=10.0, min=-10.0), FloatSlider(value=0.0, description='w_2', max=10.0, min=-10.0), Checkbox(value=False, description='draw nonlinear')))

<function __main__.plot_line(w_0, w_1, w_2, draw_nonlinear)>

```



Regardless of which parameters we try, there is no way to find a separating line. This kind of data is called **not linearly separable**. The other gates from last lesson were *linearly separable*. Nonlinear decision boundaries are possible though. You can see an example for such a boundary by activating `draw_nonlinear`. These nonlinear decision boundaries could be computed using the **kernel trick**, which we will come to later (but which we already used implicitly by including nonlinear features in the last lecture). This necessitates manually finding a good *kernel*, that transforms the data into a higher dimensional space, where it is linearly separable again.

It turns out that deep learning automatically *learns* such a kernel (or higher dimensional representation) on its own, without us having to interfere. This is done by simply adding *extra hidden layers*, forming a *deep network*.

The idea here is to express the gate as $\text{XOR} = \text{NOR}(\text{NOR}(x_1, x_2), \text{AND}(x_1, x_2))$. This is a **concatenation** of gates. We have all the weights for these “elementary” gates and can simply use the output of the inner gates as input for the outer gate:

```
from IPython.display import display

inputs = [[i, j] for i in range(2) for j in range(2)]

def plot_circle_and_line(ax, start, end, radius, xlabel, ylabel, labelshift):
    circle = plt.Circle(start, radius, color='w', ec='k', zorder=4, lw=3)
    ax.add_artist(circle)
    ax.text(start[0]-0.02, start[1]-0.01, xlabel, fontsize=15, zorder=10)
```

```

line = plt.Line2D([start[0], end[0]], [start[1], end[1]], c='k', lw=2, ls='--')
ax.text((end[0]+start[0])/3.5 + labelshift[0], (end[1]+start[1])/2+labelshift[1], \
         xlabel, fontsize = 10)
ax.add_artist(line)

headers = [r"$$x_1, x_2$$", r"NOR, AND", r"NOR"]
data = [[str(inp[0]) + ", " + str(inp[1]), \
          str(1-(inp[0] or inp[1])) + ", " + str(inp[0] and inp[1]), \
          1-(1-(inp[0] or inp[1]) or (inp[0] and inp[1]))] \
         for inp in inputs]

table = pd.DataFrame(data, columns=headers)
display(table)

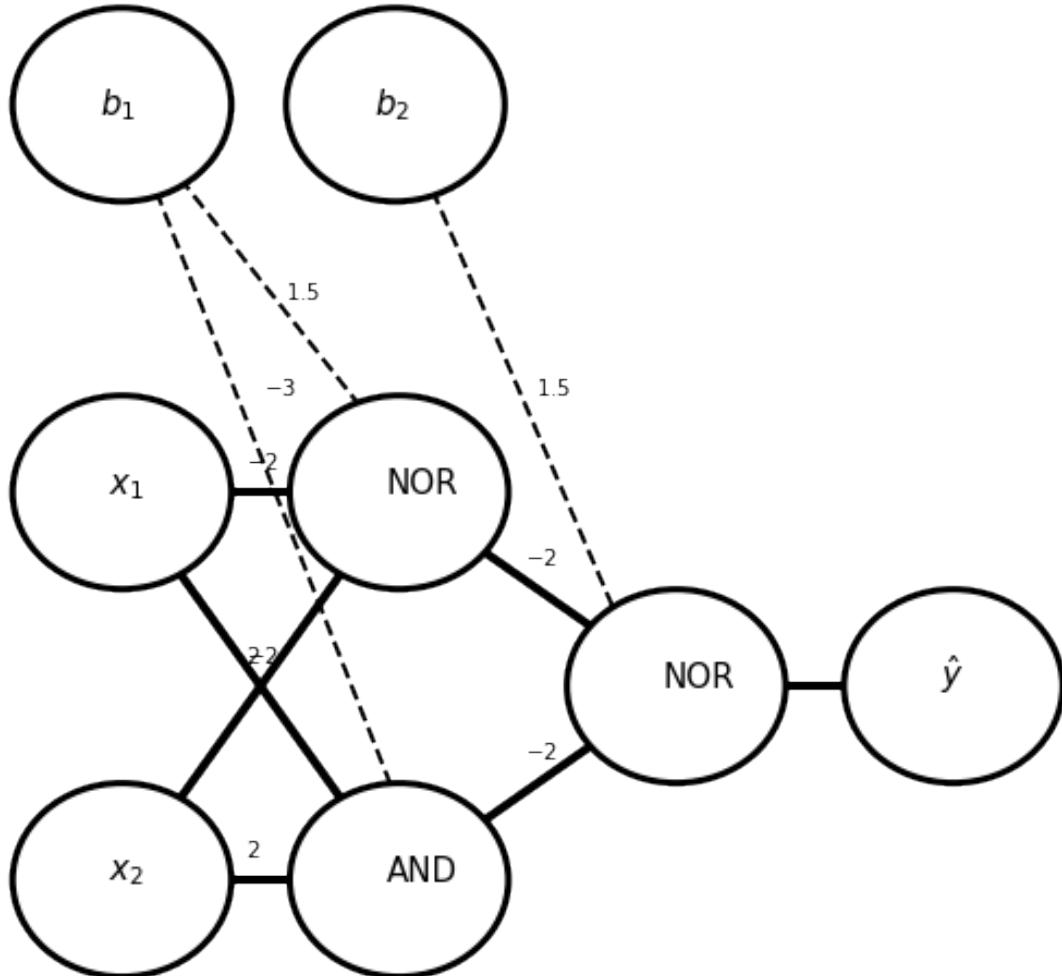
from scripts.draw_ann import draw_neural_net

label_list = [[r"x_1", r"x_2"], [r"NOR", r"AND"], [r"\hat{y}"]]
weight_list = [[[r"-2", r"2"], [r"-2", r"2"]], [[r"-2"], [r"-2"]], \
               [[]]]
fig = plt.figure(figsize=(9, 9))
ax = fig.gca()
ax.axis('off')
draw_neural_net=plt, ax, .12, .88, .1, .9, [2, 2, 1, 1], label_list, \
weight_list)
ax.set_ylim([0.15, 1.25])

plot_circle_and_line(ax, [0.12, 1.1], [0.4, 0.7], 0.1, r"$b_1$", r"$1.5$", [0.12, 0.0])
plot_circle_and_line(ax, [0.37, 1.1], [0.6, 0.5], 0.1, r"$b_2$", r"$1.5$", [0.22, 0.0])
plot_circle_and_line(ax, [0.12, 1.1], [0.4, 0.3], 0.0, "", r"$-3$", [0.1, 0.1])

```

	\$\$x_1, x_2\$\$	NOR	AND	NOR
0	0, 0	1, 0	0	
1	0, 1	0, 0	1	
2	1, 0	0, 0	1	
3	1, 1	0, 1	0	



In total, we have used 9 weights here. The output is the XOR of the inputs, which you can also see in this truth table:

```
w_1 = np.array([[ 1.5, -2, -2], \
                 [-3,  2,  2]])  
w_2 = np.array( [ 1.5, -2, -2])  
  
inputs = [[i, j] for i in range(2) for j in range(2)]  
  
headers = [r"$$x\_1, x\_2$$", r"$$y$$", r"$$\mathrm{sig}(z\_NOR)$$",  
          r"$$\mathrm{sig}(z\_AND)$$",  
          r"$$\mathrm{sig}(z\_TOT)$$"]  
data = [[str(inp[0]) + ", " + str(inp[1]), inp[0] ^ inp[1], \
```

```

        sigmoid(z(*w_1[0], inp[0], inp[1])), sigmoid(z(*w_1[1], inp[0],  

→inp[1])), \  

        sigmoid(z(*w_2, sigmoid(z(*w_1[0], inp[0], inp[1])), sigmoid(z(*w_1[1],  

→inp[0], inp[1])))) ] \  

    for inp in inputs]

display(pd.DataFrame(data, columns=headers))

    $$x\_1, x\_2$$  $$y$$  $$\mathrm{sig}(z_{\mathrm{NOR}})$$ \
0      0, 0      0          0.817574
1      0, 1      1          0.377541
2      1, 0      1          0.377541
3      1, 1      0          0.075858

    $$\mathrm{sig}(z_{\mathrm{AND}})$$  $$\mathrm{sig}(z_{\mathrm{TOT}})$$
0          0.047426          0.442752
1          0.268941          0.551575
2          0.268941          0.551575
3          0.731059          0.471572

```

Let us formalize the notation a little bit:

```

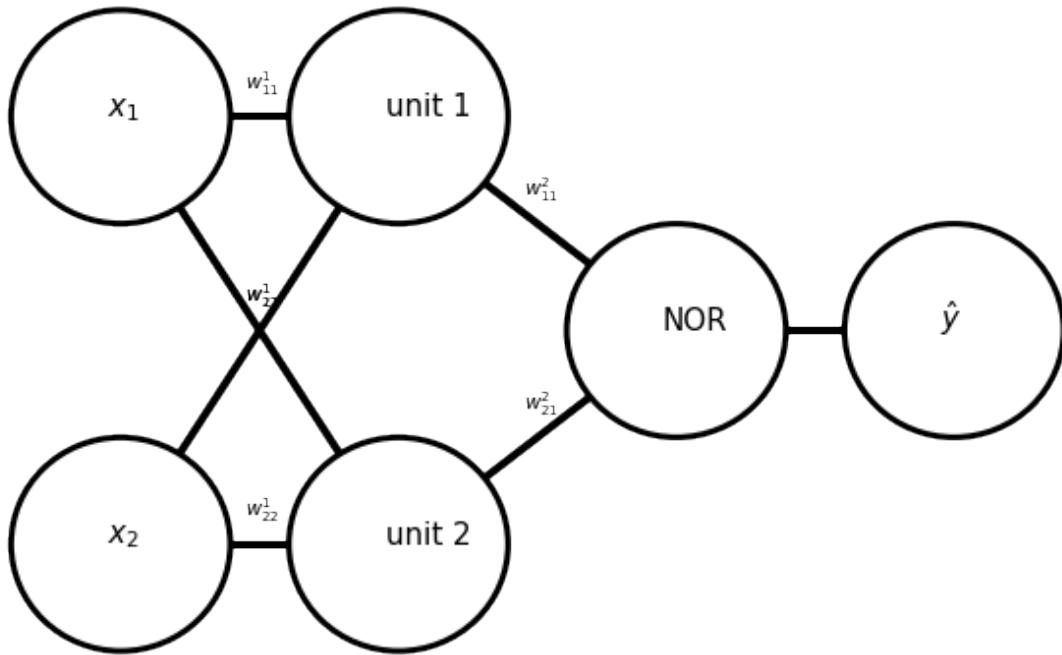
label_list = [[r"$x\_1$", r"$x\_2$"], [r"unit 1", r"unit 2"], [r"NOR"],  

→[r"$\hat{y}$"]]
weight_list = [[[r"$w_{11}^1$"], r"$w_{12}^1$"], [r"$w_{21}^1$"],  

→[r"$w_{22}^1$"]], [[r"$w_{11}^2$"], [r"$w_{21}^2$"]], [[]]]
fig = plt.figure(figsize=(10, 10))
ax = fig.gca()
ax.axis('off')
draw_neural_net(plt, ax, .12, .88, .1, .9, [2, 2, 1, 1], label_list,  

→weight_list)

```



Here, the first layer is connected to the hidden layer by 6 weights, where $w_{01}^{(1)}$ is the weight from the input bias to unit 1, the NOR gate, $w_{11}^{(1)}$ is the weight from input x_1 to the NOR gate, and so on. In the last layer, $w_{01}^{(2)}$ connects the bias from the hidden layer to the last NOR gate, $w_{11}^{(2)}$ connects the output of unit 1 to the last NOR gate, and $w_{21}^{(2)}$ connects the output of unit 2 to the last NOR gate.

So this can successfully represent an XOR gate. There is a theorem called the **universal approximation theorem** that in rough terms says:

Any function from a class that is invariant under concatenation can be expressed by a neural network with one hidden layer and arbitrary nonlinear activation function up to some desired accuracy.

More details on this will come in a later lecture. Note that this does not say anything about how many neurons are needed for this to work.

What introducing a hidden layer with more neurons does is introduce new *dimensions*. The network then automatically *learns* a higher dimensional representation of the input data, that is linearly separable. For this to work, the dimension of the hidden layer must be high enough (sometimes introducing several hidden layers helps too). You can see the result of this in the following plot, alongside the 2D decision plane.

```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d

W1 = np.array([[1.5, -3], [-2, 2], [-2, 2]])
W2 = np.array([[1.5], [-2], [-2]])

inputs = np.array([[1, i, j] for i in range(2) for j in range(2)])

layer1out = []
layer2out = []

for inp in inputs:
    layer1out.append([1, *sigmoid(W1.T @ inp)])

for l1 in layer1out:
    layer2out.append(sigmoid(W2.T @ l1))

layer1out = np.array(layer1out)
layer2out = np.array(layer2out).reshape(-1)

print(layer1out)
print(layer2out)

fig = plt.figure(figsize=(9,8))
ax = fig.add_subplot(111, projection='3d')

x_m, y_m = np.meshgrid(np.linspace(0, 1, 20), \
                       np.linspace(0, 1, 20))

ax.scatter(inputs[:,1], inputs[:,2], layer2out, c=y_xor, lw=30)
#ax.scatter(layer1out[:,1], layer1out[:,2], layer2out.reshape(-1), c=y_xor, lw=30)
ax.plot_surface(x_m, y_m, np.zeros(x_m.shape)+0.5, alpha=0.5)
ax.set_zlim([0.4, 0.6])

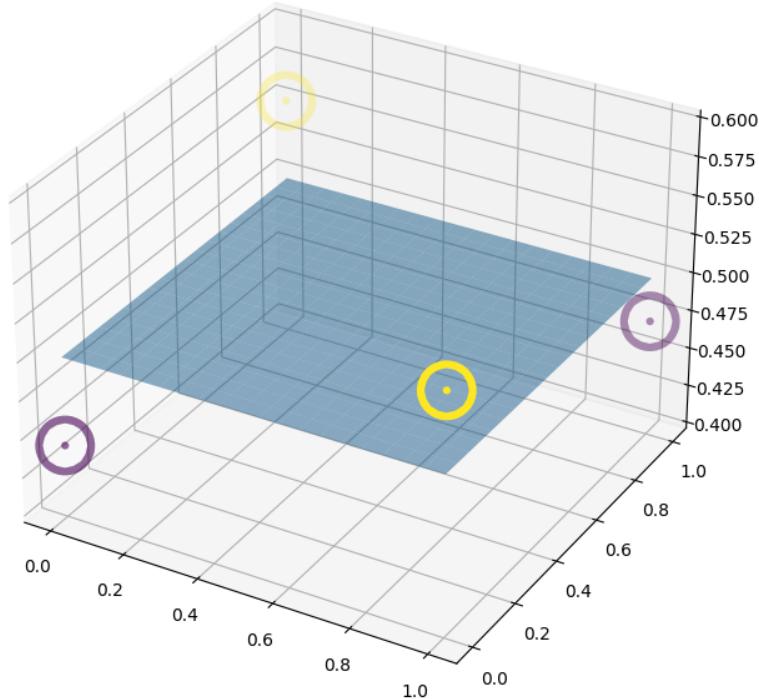
plt.savefig("img/plot_XOR_3d.png")
```

[[1. 0.81757448 0.04742587]
 [1. 0.37754067 0.26894142]
 [1. 0.37754067 0.26894142]
 [1. 0.07585818 0.73105858]]

```
[0.44275197 0.55157486 0.55157486 0.47157231]
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```



6.3 Gradient of the Binary Cross-Entropy Loss Function

In the last lessons, we guessed the weights for the gates without using any optimization technique. Obviously this will be absolutely impossible in higher dimensional settings. For using gradient descent, we need the gradient of the loss function J . This necessitates using the **chain rule**:

$$\frac{\partial J}{\partial w} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w} \quad (83)$$

```

import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))
ax = plt.subplot(111)
ax.set_aspect("equal")

circle = plt.Circle((0.4, 0.4), 0.1, color='w', ec='k', zorder=4, lw=3)
ax.add_artist(circle)
ax.text(0.4-0.02, 0.4-0.01, "x", fontsize=15, zorder=10)

line = plt.Line2D([0.4, 0.7], [0.4, 0.1], c='k', lw=4)
ax.text(0.55, 0.35, "w", fontsize = 10)
ax.add_artist(line)

circle = plt.Circle((0.75, 0.05), 0.1, color='w', ec='k', linestyle='--', zorder=4, lw=1)
ax.add_artist(circle)
ax.text(0.75-0.02, 0.05-0.01, "z", fontsize=15, zorder=10)

line = plt.Line2D([0.75, 0.95], [0.05, 0.4], c='k', lw=4)
ax.text(0.85-0.05, 0.25+0.05, "a", fontsize = 10)
ax.add_artist(line)

circle = plt.Circle((0.95, 0.4), 0.1, color='w', ec='k', zorder=4, lw=3)
ax.add_artist(circle)
ax.text(0.95-0.02, 0.4-0.01, "ŷ", fontsize=15, zorder=10)

line = plt.Line2D([0.95, 1.2], [0.4, 0.4], c='k', lw=4)
#ax.text(0.85-0.05, 0.25+0.05, "a", fontsize = 10)
ax.add_artist(line)

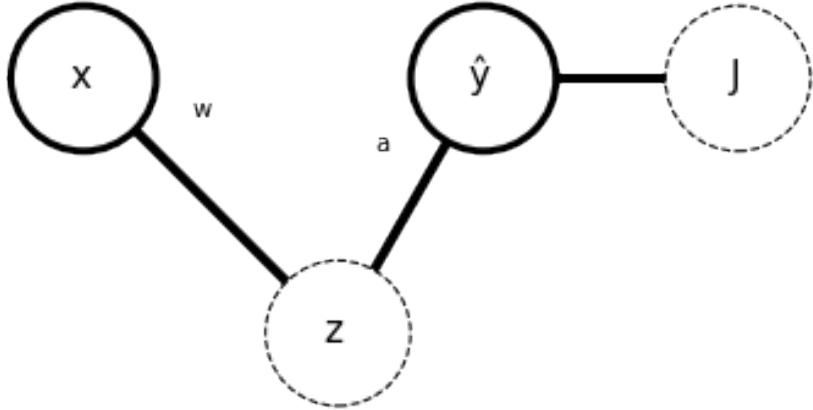
circle = plt.Circle((1.3, 0.4), 0.1, color='w', ec='k', linestyle='--', zorder=4, lw=1)
ax.add_artist(circle)
ax.text(1.3-0.01, 0.4-0.01, "J", fontsize=15, zorder=10)

ax.set_xlim([0, 1.5])
ax.set_ylim([-0.1, 0.6])

ax = fig.gca()
ax.axis('off')

```

(0.0, 1.5, -0.1, 0.6)



Something very similar will also be used in artificial neural networks, which is then called **back-propagation**.

The loss function itself was

$$J = - \sum_{i=1}^N [y^{(i)} \ln(\hat{y}^{(i)}) + (1 - y^{(i)}) \ln(1 - \hat{y}^{(i)})] \quad (84)$$

Now the partial derivative of this with respect to the weights is the sum of the partial derivatives for each term

$$\frac{\partial J}{\partial \mathbf{w}} = \sum_{i=1}^N \frac{\partial J^{(i)}}{\partial \mathbf{w}} \quad (85)$$

so we only need to apply the chain rule to the loss function for each example i . First derivative we need is with respect to $\hat{y}^{(i)}$ ($y^{(i)}$ is fixed):

$$\frac{\partial J^{(i)}}{\partial \hat{y}^{(i)}} = - \left[\frac{y^{(i)}}{\hat{y}^{(i)}} - \frac{1 - y^{(i)}}{1 - \hat{y}^{(i)}} \right] \quad (86)$$

The next one is

$$\frac{\partial \hat{y}^{(i)}}{\partial z} = \frac{\partial \sigma(z)}{\partial z} = \sigma(z) \cdot (1 - \sigma(z)) = \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \quad (87)$$

where the gradient of the sigmoid function is $\sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$. Combined, these two give

$$\frac{\partial J^{(i)}}{\partial \hat{y}^{(i)}} \frac{\partial \hat{y}^{(i)}}{\partial z} = -[y^{(i)} - \hat{y}^{(i)}] \quad (88)$$

which is simply the negative *error* itself. The last part is

$$\frac{\partial z}{\partial \mathbf{w}} = \mathbf{x}^{(i)} \quad (89)$$

So the result for the whole derivative is

$$\frac{\partial J^{(i)}}{\partial \mathbf{w}} = -[y^{(i)} - \hat{y}^{(i)}] \cdot \mathbf{x}^{(i)} = -\text{error} \cdot \text{input} \quad (90)$$

and for the complete loss function gradient for logistic regression with binary cross entropy loss:

$$\frac{\partial J}{\partial \mathbf{w}} = - \sum_{i=1}^N [y^{(i)} - \hat{y}^{(i)}] \cdot \mathbf{x}^{(i)} \quad (91)$$

Recall the gradient we used for linear regression. Apart from a factor $\frac{1}{2N}$, this is exactly the same gradient. Using regularization leads to the exact same results here and can be applied just like it was applied in linear regression. Note that although the expression here is the same as before, J itself isn't, because here, the y and \hat{y} values are restricted by the sigmoid activation function.

An important distinction here is that in logistic regression, the loss function is **not convex**. Convex loss landscapes have a unique global minimum which *will* be found by gradient descent (given acceptable hyperparameters and starting points). Think of the parabolic landscape from before. Here, there is (at least in general) not a unique minimum. Recall the logic gate examples, where different curves did the job just fine. So logistic regression does not give a unique solution.

```
%matplotlib notebook
import numpy as np
from mpl_toolkits.mplot3d import axes3d
from sklearn.datasets import make_blobs
from ipywidgets import interact

def sigmoid(z):
    return 1/(1 + np.exp(-z))

def h(w, x):
    return w.T @ x

def BCE(w, x, y):
    return -(y*np.log(h(w, x)) - (1-y)*np.log(1-h(w, x)))

X, Y = make_blobs(n_features=2, centers=2)
x = np.array([0.5, 0.5])
```

```
w1 = np.linspace(0.1, 0.9, 40)
w2 = np.linspace(0.1, 0.9, 40)
w1m, w2m = np.meshgrid(w1, w2)

def plot_log_ls(x1=0.1, x2=0.1):
    Z = BCE(np.array([w1m, w2m]), np.array([x1, x2]), 0.99*Y[:40])
    Z = np.nan_to_num(Z.reshape(w1m.shape))

    fig = plt.figure(figsize=(9,8))
    ax = fig.add_subplot(111, projection='3d')

    #ax.cla()
    #ax.scatter(X[:, 0], X[:, 1], marker='o', c=Y, s=100, edgecolor='k')

    ax.plot_surface(w1m, w2m, Z, lw=4, cmap='viridis')

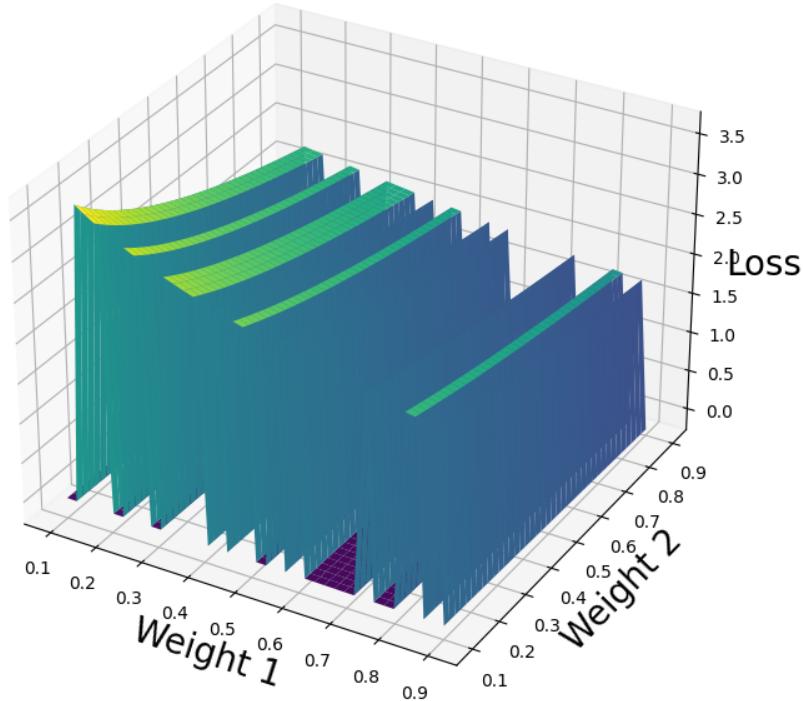
    ax.set_xlabel("Weight 1", fontsize=20)
    ax.set_ylabel("Weight 2", fontsize=20)
    ax.set_zlabel("Loss", fontsize=20)

    plt.savefig("img/plot_BCE_grad.png")

interact(plot_log_ls, x1=(-2.0, 2.0), x2=(-2.0, 2.0))
```

```
interactive(children=(FloatSlider(value=0.1, description='x1', max=2.0, min=-2.0), FloatSlider
```

```
<function __main__.plot_log_ls(x1=0.1, x2=0.1)>
```



This extends to artificial neural networks, which will rarely attain their global optimum. In the example above, there are many local minima in which gradient descent can get stuck. This is usually much worse in the case of neural networks.

6.4 Multinomial Classification

Let's generalize binary classification, where we had only two classes, to an arbitrary number of classes K . One example that is used in almost any course is the [MNIST data set of handwritten numbers](#), that contains 70,000 images of handwritten numbers as 28x28 pixel grayscale images.

```

import sklearn
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn import datasets

# if you want to try this yourself, you need to fetch the data set first

```

```

# this takes some time though, since the data set is almost 500MB in size
# hence, it makes sense to save the data in numpy arrays, with the
# extension ".npy" and load them when you rerun the code.
#X, y = datasets.fetch_openml('mnist_784', version=1, return_X_y=True)
#np.save("X.npy", X)
#np.save("y.npy", y)

# allow_pickle here is needed because of a current bug,
# astype(np.float) loads the data as floats instead of strings
try:
    X
# only load this if X isn't already loaded, to save computation time
except NameError:
    X = np.load("/data/X.npy", allow_pickle=True).astype(np.float)
    y = np.load("/data/y.npy", allow_pickle=True).astype(np.float)

# 70000 examples of 28x28 pixel images, saved as 784 float arrays
print(X.shape, y.shape)

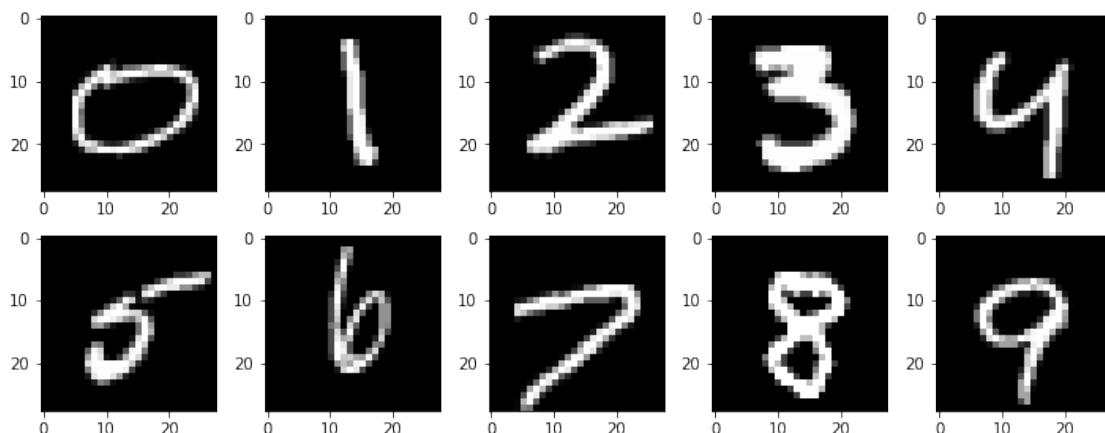
fig, ax = plt.subplots(2,5, figsize=(10,4))
ax = ax.flatten()

for i in range(10):
    indices = np.argwhere(y == i).flatten()
    index = np.random.choice(indices)
    image = np.reshape(X[index], (28, 28))
    ax[i].imshow(image, cmap='gray')

plt.tight_layout()
plt.show()

```

(70000, 784) (70000,)



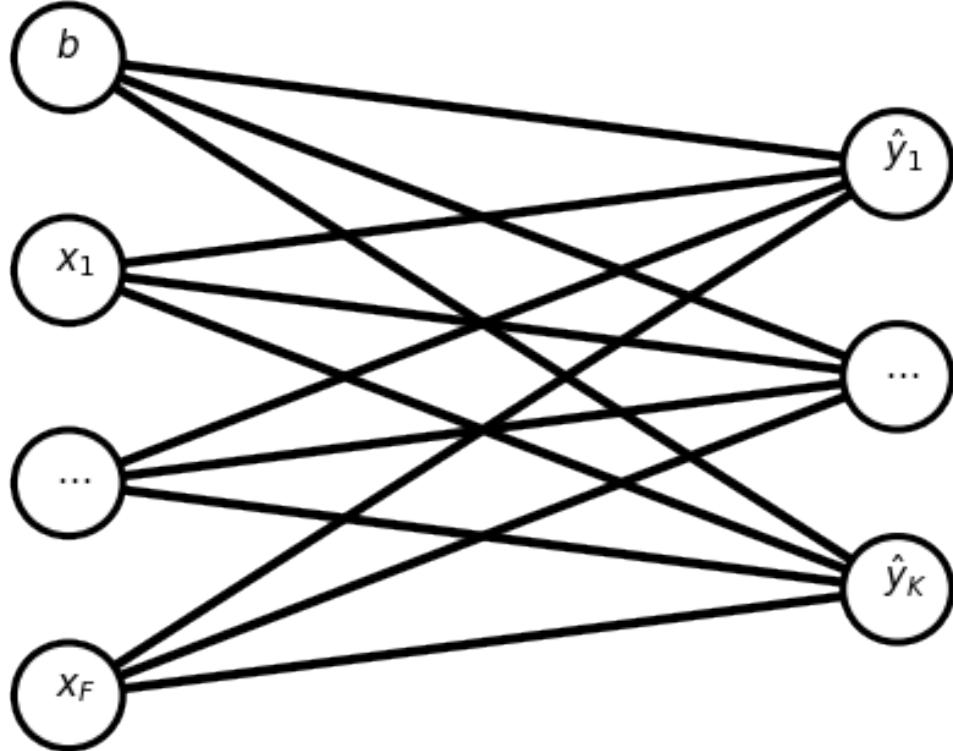
This dataset consists of $K = 10$ classes. How can we represent the labels here as numbers? Strings are not a good idea at all. Real-valued output is also difficult to handle, since the numbers are very close and there is no clear separation (is 8 a very definite 8, or a badly detected 7?). A better possibility is to use **one-hot encoding**.

In the binary case, say, for the images of the cracked hull example, we had a single number output of either 0 or 1. Now instead, the output will be either the vector $[1, 0]$ for class 0, or the vector $[0, 1]$ for class 1 (of course for binary classification this isn't really necessary). This is easily extendable to K classes. In the *MNIST* example above, 0 would be output as $[1, 0, 0, 0, \dots, 0]$, 1 as $[0, 1, 0, 0, \dots, 0]$ and so on. In general for K classes, this would be $\hat{y} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_K]$. This type of representation is called “one-hot” (short synonym for “one out of x”), because only one of the values is close to 1, while the others are close to 0 (cold). At least if the classification algorithm worked. So $\hat{y}_i \in [0, 1] \forall i$ as usual. Then, we can interpret it as a probability $\hat{y}_i = p(y_i = 1|x)$, where x is an example.

So far, we only discussed “networks” with single-value outputs. What must happen in the last layer nonlinear activation-wise to ensure such an outcome? In the *MNIST* dataset, x is an image of 28x28 pixels, flattened to a vector of size 784, which is the input.

```
from scripts.draw_ann import draw_neural_net

label_list = [[r"$b$", r"$x_1$", r"$\dots$", r"$x_F$"], [r"$\hat{y}_1$",
    r"$\dots$", r"$\hat{y}_K$"]]
weight_list = [4*[3*[""]]]
fig = plt.figure(figsize=(8, 8))
ax = fig.gca()
ax.axis('off')
draw_neural_net=plt, ax, .12, .88, .1, .9, [4, 3], label_list, weight_list)
```



The \hat{y}_i will now be probabilities, so their sum should be $\sum_i \hat{y}_i = 1$, while all $\hat{y}_i \in [0, 1]$. In the above image, the action of the lines shall be represented by a matrix operation of the weights themselves. Here, \mathbf{x} has dimension 785×1 , while $\hat{\mathbf{y}}$ has dimension $K \times 1$, so the **weight matrix** \mathbf{W} has to have dimension $785 \times K$ to make things work (remember, we took $\mathbf{w}^T \cdot \mathbf{x}$, but sometimes this is defined differently). *Side note: the action of multiplying the weights and inputs and then doing the summation is called “multiply and accumulate” (MAC). This is done extensively in machine learning and is a combination of two operations implemented in a CPU/GPU. Many companies succeeded in implementing MAC in hardware to speed things up. Worth mentioning here are especially Google with their TPUs (tensor processing units) and NVIDIA with their Tensor Cores on current graphics cards. These are still limited to small portions of MAC, such that they can only process 2×2 matrices, but it does speed up computation considerably.*

Again, we need a function that squishes the values into the range $[0, 1]$, while simultaneously making sure that the sum of all probabilities is 1. Using sigmoid again on the product $\mathbf{W}^T \mathbf{x}$ won't work,

since it doesn't ensure the sum to be 1. Instead, softmax is used, defined as

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^K \exp(z_j)} \quad (92)$$

where each component is normalized by the sum of all exponents. softmax returns a vector, where each component is computed as defined above $\text{softmax}(\mathbf{z}) = [e^{z_1}, e^{z_2}, \dots]/Z$ with the *state sum Z*. *Side note: this is actually a point in machine learning where the connection to physics (especially statistical mechanics) itself is very obvious. There might be a short lesson about this in the future, if interest is high.*

Adjusting the loss function is straightforward:

$$J = - \sum_{k=1}^K y_k \ln(\hat{y}_k) - (1 - y_k) \ln(1 - \hat{y}_k) \quad (93)$$

This is the **cross-entropy loss function** for a general number of classes K . This reduces to binary cross-entropy loss for $K = 2$, because complementary probabilities $\hat{y}_2 = 1 - \hat{y}_1$ ensure that the sum of all occurring probabilities is 1.

The general procedure is thus the following: * Design the forward model $[1 \ x_1 \ x_2 \ \dots] \rightarrow [\hat{y}_1 \ \hat{y}_2 \ \dots]$, where $\hat{\mathbf{y}} = \text{softmax}(w_{01}^{(1)} + w_{02}^{(1)}x_1 + \dots, w_{11}^{(1)} + w_{12}^{(1)}x_1 + \dots, \dots)$ * Get gradient of the loss and adjust the weights until convergence or the maximum number of steps is reached.

Here, $W_{ij}^{(l)}$ is a component of the weight matrix, with input indicator i , output indicator j , and layer indicator l . Bias is often denoted as b_1 instead of $w_{01}^{(1)}$ and so on, and the equation system as $\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}^T \mathbf{x} + \mathbf{b})$ with $\dim(\hat{\mathbf{y}}) = K \times 1$, $\dim(\mathbf{W}) = K \times F$, $\dim(\mathbf{x}) = F \times 1$, and $\dim(\mathbf{b}) = K \times 1$, where K is the number of classes and F is the number of features. Otherwise, the equation system is $\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}^T \mathbf{x})$, where $\mathbf{x} = [1 \ x_1 \ x_2 \ \dots]$ with dimensions changed accordingly.

6.5 Artificial Neural Networks

6.5.1 Artificial Neuron

The artificial neuron is inspired by the biological neuron, where a number of dendrites pass on input currents to the axon, which then *MACs* the inputs and axon terminals output them if a certain threshold current is reached after activation (this is very roughly speaking). The artificial neuron looks like this:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def plot_circle_and_line(ax, start, end, radius, xlabel, ylabel, labelshift):
    circle = plt.Circle(start, radius, color='w', ec='k', zorder=4, lw=3)
    ax.add_artist(circle)
    ax.text(start[0]-0.02, start[1]-0.01, xlabel, fontsize=15, zorder=10)
    line = plt.Line2D([start[0], end[0]], [start[1], end[1]], c='k', lw=4)
```

```

    ax.text((end[0]+start[0])/3.5, (end[1]+start[1])/2+labelshift, elabel, u
    ↪fontsize = 10)
    ax.add_artist(line)

fig = plt.figure(figsize=(8,8))
ax = plt.subplot(111)
ax.set_aspect("equal")

plot_circle_and_line(ax, [1, 1], [2, 1], 0.5, "", "", 0)

plot_circle_and_line(ax, [0, 1.6], [1, 1], 0.2, r"$b$", r"$w_0$", 0.18)
plot_circle_and_line(ax, [0, 1], [1, 1], 0.2, r"$x_1$", r"$w_1$", 0.05)
plot_circle_and_line(ax, [0, 0.4], [1, 1], 0.2, r"$x_2$", r"$w_2$", -0.05)

line = plt.Line2D([1, 2], [1, 1.6], c='k', lw=4, zorder=-4)
ax.add_artist(line)
line = plt.Line2D([1, 2], [1, 0.4], c='k', lw=4, zorder=-4)
ax.add_artist(line)

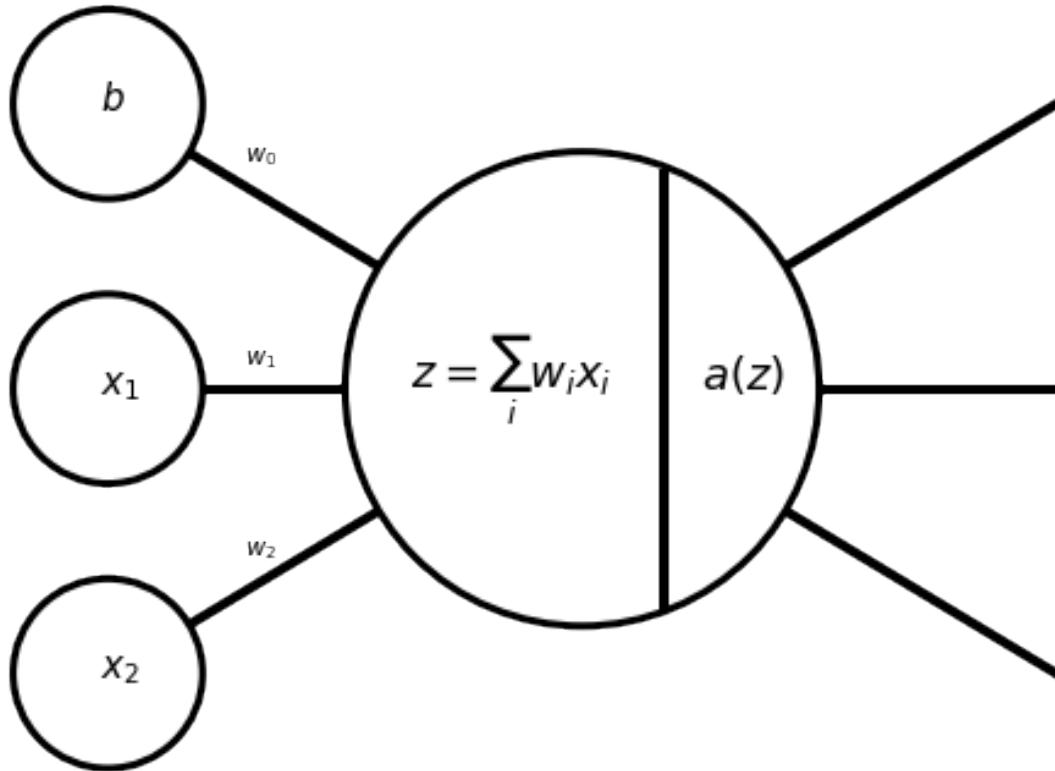
line = plt.Line2D([1-np.cos(np.pi/1.8), 1-np.cos(np.pi/1.8)], [1+0.46*np.sin(np.
    ↪pi/1.8), 1-0.46*np.sin(np.pi/1.8)], c='k', lw=4, zorder=11)
ax.add_artist(line)
ax.text(1-0.45, 1, r"\quad z = \sum_i w_i x_i", fontsize = 18, zorder=11)
ax.text(1+0.25, 1, r"$a(z)$", fontsize = 18, zorder=11)

ax.set_xlim([-0.3, 2])
ax.set_ylim([0, 2])

ax = fig.gca()
ax.axis('off')

```

(-0.3, 2.0, 0.0, 2.0)



The inputs themselves do not belong to the neuron itself, but the weights do. What the neuron does is it takes a *linear combination* (not thinking of the inputs as vectors in this case) of the inputs x_i with the weights w_i which returns a scalar value z , then applies a *nonlinear activation function* $a(z)$ to that combination. So input is supplied to the neuron, which then MACs, activates, and supplies the output. In general, different kinds of activation functions are used, like the sigmoid function we used in logistic regression, but also tanh, ReLU, leaky ReLU, linear and other functions can be used with different effects. These will be discussed later. You can play around with the code below to see what outputs a neuron with different activations and input supplies:

```
%matplotlib notebook
from mpl_toolkits.mplot3d import axes3d
from ipywidgets import interact
import ipywidgets as widgets

def linear(z):
    return z
```

```

def relu(z):
    return z.clip(min=0)

def sigmoid(z):
    return 1/(1+np.exp(-z))

def tanh(z):
    return np.tanh(z)

class Neuron():
    def __init__(self, n_input, weights, activation="linear"):
        self.n_input = n_input
        self.weights = np.random.random(n_input) if weights == None else weights
        if activation == "relu":
            self.activation = relu
        elif activation == "sigmoid":
            self.activation = sigmoid
        elif activation == "tanh":
            self.activation = tanh
        else:
            self.activation = linear

    def update_weights(self, weights):
        self.weights = np.array(weights)

    def get_z(self, inputs): #inputs includes bias term
        return self.weights.T @ inputs

    def get_output(self, inputs): #inputs includes bias term
        return self.activation(self.weights.T @ inputs[1:])

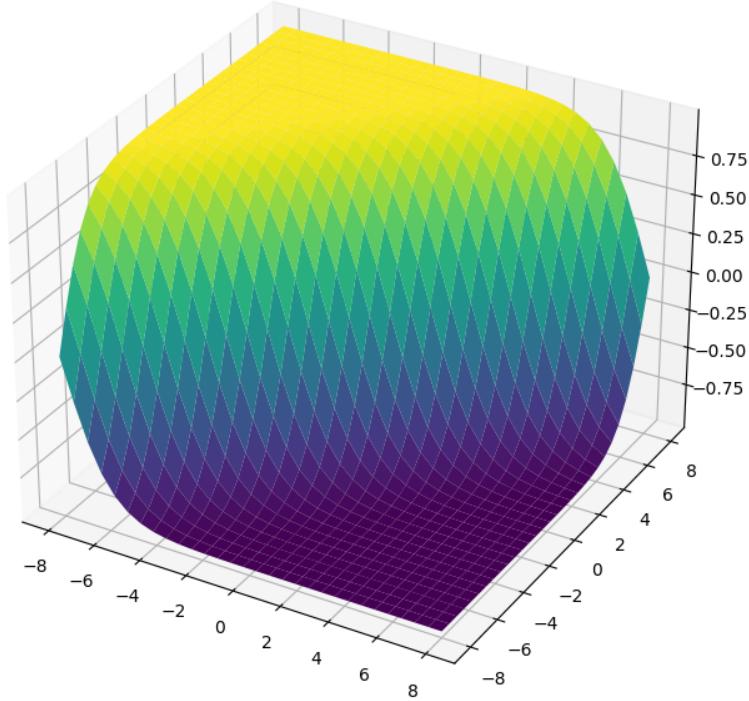
w1 = np.linspace(-8, 8, 30)
w2 = np.linspace(-8, 8, 30)
w1m, w2m = np.meshgrid(w1, w2)

n1 = Neuron(2, None, "linear")
n2 = Neuron(2, None, "relu")
n3 = Neuron(2, None, "sigmoid")
n4 = Neuron(2, None, "tanh")
neurons = [n1, n2, n3, n4]
inputs = [1, 0.5, -0.5]
J = [[], [], [], []]

for i in range(len(neurons)):

```

```
for w1i in w1:  
    for w2i in w2:  
        neurons[i].update_weights([w1i, w2i])  
        J[i].append(neurons[i].get_output(inputs))  
  
J = np.array(J)  
J = J.reshape((len(neurons), *w1m.shape))  
  
def plot_output(activation):  
    fig = plt.figure(figsize=(9,8))  
    ax = fig.add_subplot(111, projection='3d')  
    #ax.cla()  
    ax.plot_surface(w1m, w2m, J[activation], cmap='viridis')  
  
    plt.savefig("img/plot_activation_landscapes.png")  
  
interact(plot_output, activation=(0, 3))  
  
interactive(children=(IntSlider(value=1, description='activation', max=3), Output()), _dom_classes=['mlmech'])  
  
<function __main__.plot_output(activation)>
```



6.5.2 Feedforward Neural Network

```
%matplotlib inline
from scripts.draw_ann import draw_neural_net

label_list = [[r"$b$",
               r"$x_1$",
               r"$x_2$",
               r"$x_2$",
               r"$\dots$",
               r"$x_F$"],
              ["",
               "",
               "",
               ""],
              ["",
               "",
               "",
               ""],
              [r"$\hat{y}_1$",
               r"$\hat{y}_2$",
               r"$\hat{y}_3$",
               r"$\hat{y}_4$",
               r"$\dots$",
               r"$\hat{y}_K$"]]
weight_list = [6*[4*[""]],
               4*[4*[""]],
               4*[6*[""]]]
fig = plt.figure(figsize=(12, 12))
ax = fig.gca()
ax.axis('off')
draw_neural_net(plt, ax, .12, .88, .1, .9, [6, 4, 4, 6], label_list,
                weight_list)
```

```

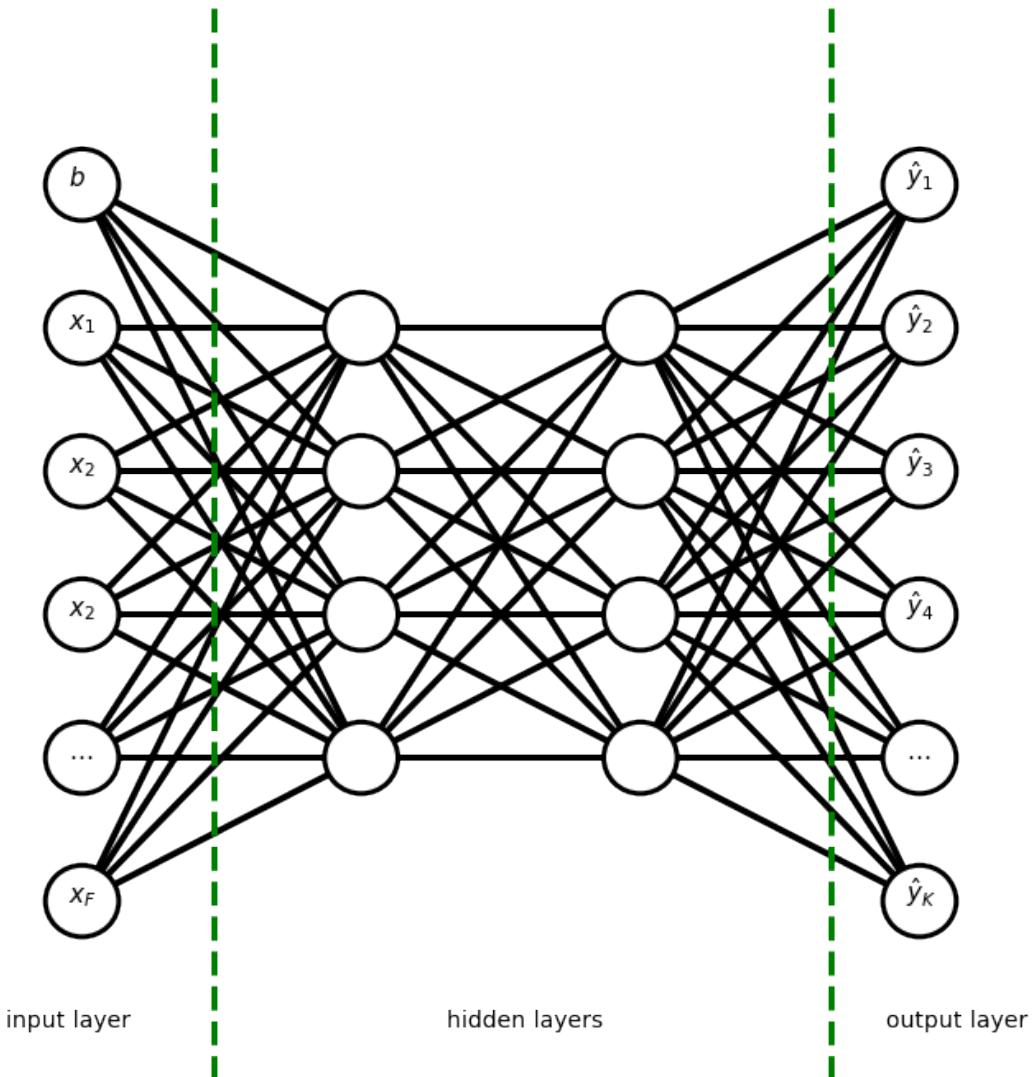
line = plt.Line2D([0.24, 0.24], [0, 1], c='green', lw=4, linestyle='--', ↵
                  zorder=11)
ax.add_artist(line)

line = plt.Line2D([0.8, 0.8], [0, 1], c='green', lw=4, linestyle='--', ↵
                  zorder=11)
ax.add_artist(line)

ax.text(0.05, 0.05, "input layer", fontsize = 14, zorder=11)
ax.text(0.45, 0.05, "hidden layers", fontsize = 14, zorder=11)
ax.text(0.85, 0.05, "output layer", fontsize = 14, zorder=11)

```

Text(0.85, 0.05, 'output layer')



Each unit in the image above is an artificial neuron. The first layer here is called the *input layer*, the last layer is the *output layer*. Every layer inbetween is called a *hidden layer*. If there is at least one hidden layer, this is called a **deep neural network**. Each neuron in layer n_{l+1} has its own set of $n_l + 1$ weights, where n_l is the number of neurons in layer l . Assuming layer l has N weights, and that layer $l + 1$ has M weights, the number of weights required (given that *all* neurons are connected to neurons in neighboring layers) is $N \times M$. Each neuron is assigned a different bias. Networks like this are called **fully-connected networks**. It's not necessary that neurons are all connected, which we will discuss in the next lecture and again later.

6.6 Backpropagation

In diagrams of neural networks, the bias unit is usually omitted, because it is always 1 and has no incoming connections from previous layers. For now, we are only looking at fully-connected networks, meaning that each node of a layer is connected to each node in the next layer. Nodes $n_i^{(l)}$ and $n_j^{(l+1)}$ are connected by weight $W_{ij}^{(l)}$ in the l th layer. Input nodes are $n_i^{(1)}$. At the end of a forward pass, the loss function $J(\mathbf{y}, \hat{\mathbf{y}})$ is calculated, depending on the weights \mathbf{W} . At first, these are all guessed. What is needed now is a way to investigate which weight was responsible for what part of the loss, so we need the derivative $\frac{\partial J}{\partial \mathbf{W}}$ in order to redistribute a part of the gradient to the weights. This redistribution process is called **backpropagation**. We could do the following:

- Guess \mathbf{W}
- Do a forward pass
- Calculate $J(\mathbf{y}, \hat{\mathbf{y}}; \mathbf{W})$
- Do the same with another guess $\mathbf{W} + \Delta \mathbf{W}$
- Use the finite difference method to get a gradient $\frac{\partial J}{\partial \mathbf{W}} = \frac{J(\mathbf{W} + \Delta \mathbf{W}) - J(\mathbf{W})}{\Delta \mathbf{W}}$

We already found out that this isn't feasible at all for larger problems, since this has to be computed for each weight, each input and so on. Large networks have numbers of weights easily in the millions. This is the reason for why before *automatic differentiation* was found, artificial neural networks weren't successful at all.

Instead, in usual *backpropagation*, **automatic differentiation** is used (developed by Hinton, among others). This is basically "just" using the chain rule, but there is a bit more to it. It's a very cheap method, but difficult to program. Pretty much every package for machine learning implements a routine for this.

To get some intuition for how this works, we will omit the bias unit for now and look at a simple case of scalar input and output, as well as scalar weights.

```
import matplotlib.pyplot as plt

def plot_circle_and_line(ax, start, end, radius, xlabel, ylabel, labelshift):
    circle = plt.Circle(start, radius, color='w', ec='k', zorder=4, lw=3)
    ax.add_artist(circle)
    ax.text(start[0]-0.1, start[1]-0.05, xlabel, fontsize=18, zorder=10)
    line = plt.Line2D([start[0], end[0]], [start[1], end[1]], c='k', lw=4)
    ax.text((end[0]+start[0])/3.5 + labelshift[0], (end[1]+start[1])/
→2+labelshift[1], ylabel, fontsize = 16)
```

```

ax.add_artist(line)

fig = plt.figure(figsize=(8,8))
ax = plt.subplot(111)
ax.set_aspect("equal")

plot_circle_and_line(ax, [1, 1], [3, 1], 0.7, r"$x$", r"$w^{(1)}$",
                     [0.6, 0.3])
plot_circle_and_line(ax, [3.5, 1], [5.5, 1], 0.7, r"$n^{(2)}$",
                     r"$w^{(2)}$",
                     [1.7, 0.3])
plot_circle_and_line(ax, [6, 1], [8, 1], 0.7, r"$n^{(3)}$",
                     r"$w^{(3)}$",
                     [2.8, 0.3])
plot_circle_and_line(ax, [8.5, 1], [10.5, 1], 0.7, r"$\hat{y}$", "", [1.7, 0.3])

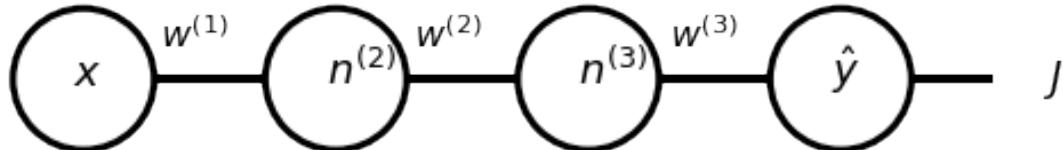
ax.text(10.5, 0.9, r"$J$",
        fontsize = 18, zorder=11)

ax.set_xlim([-0.1, 10])
ax.set_ylim([-0.1, 2])

ax = fig.gca()
ax.axis('off')

```

(-0.1, 10.0, -0.1, 2.0)



In the end, J is calculated and we want $\frac{\partial J}{\partial w}$. With $n^{(1)} = x$, $z^{(2)} = w^{(1)}n^{(1)}$, $n^{(2)} = a(z^{(2)})$ and so on, we can calculate $\frac{\partial J}{\partial w^{(1)}}$, $\frac{\partial J}{\partial w^{(2)}}$ and $\frac{\partial J}{\partial w^{(3)}}$ using the chain rule. The following image might help reconstructing the chain rule path:

```

fig = plt.figure(figsize=(12,12))
ax = plt.subplot(111)
ax.set_aspect("equal")

plot_circle_and_line(ax, [1, 1], [3.5, -1], 0.7, r"$n^{(1)}$",
                     r"$w^{(1)}$",
                     [0.7, 0.3])
plot_circle_and_line(ax, [3.5, -1], [3.5, 1], 0.7, r"$z^{(2)}$",
                     r"$a$",
                     [1.7, 0])
plot_circle_and_line(ax, [3.5, 1], [6, -1], 0.7, r"$n^{(2)}$",
                     r"$w^{(2)}$",
                     [1.7, 0.3])

```

```

plot_circle_and_line(ax, [6, -1], [6, 1], 0.7, r"$z^{(3)}$", r"$a$", [2.8, 0])
plot_circle_and_line(ax, [6, 1], [8.5, -1], 0.7, r"$n^{(3)}$", r"$w^{(3)}$", [2.
    ↪8, 0.3])
plot_circle_and_line(ax, [8.5, -1], [8.5, 1], 0.7, r"$z^{(4)}$", r"$a$", [4, 0])
plot_circle_and_line(ax, [8.5, 1], [10.5, 1], 0.7, r"$n^{(4)}$", "", [2.8, 0.3])

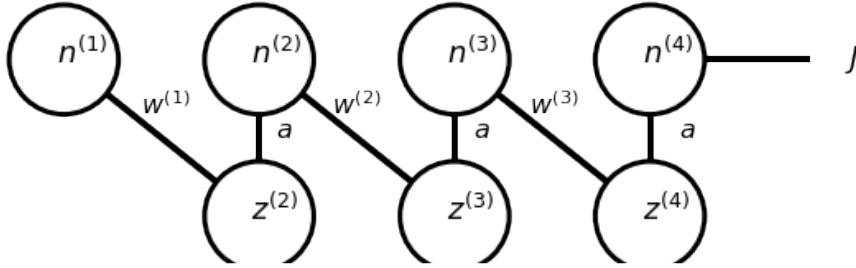
ax.text(11, 0.9, r"$J$", fontsize = 18, zorder=11)

ax.set_xlim([-0.1, 14])
ax.set_ylim([-1.6, 2])

ax = fig.gca()
ax.axis('off')

```

(-0.1, 14.0, -1.6, 2.0)



We need $\frac{\partial J}{\partial w^{(3)}}$ first, since this is the closest weight. Without loss of generality, let's assume that $a = \text{sigmoid}$ and J is the binary cross-entropy loss. Then

$$\frac{\partial J}{\partial w^{(3)}} = \frac{\partial J}{\partial n^{(4)}} \frac{\partial n^{(4)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial w^{(3)}} = -[y - \hat{y}] \cdot n^{(3)} \quad (94)$$

since $\frac{\partial J}{\partial z^{(4)}} = -[y - \hat{y}] = -[y - n^{(4)}]$, which is the *error in output activation*, and $\frac{\partial z^{(4)}}{\partial w^{(3)}} = n^{(3)}$. The former term is often abbreviated to $\frac{\partial J}{\partial z^{(l)}} = \delta^{(l)}$, so here $\frac{\partial J}{\partial w^{(3)}} = \delta^{(4)} n^{(3)}$.

The next term we need is

$$\frac{\partial J}{\partial w^{(2)}} = \frac{\partial J}{\partial n^{(4)}} \frac{\partial n^{(4)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial n^{(3)}} \frac{\partial n^{(3)}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial w^{(2)}} = \delta^{(3)} \cdot n^{(2)} \quad (95)$$

where $\frac{\partial J}{\partial n^{(4)}} \frac{\partial n^{(4)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial n^{(3)}} \frac{\partial n^{(3)}}{\partial z^{(3)}} = \frac{\partial J}{\partial z^{(3)}} = \delta^{(3)}$ and $\frac{\partial z^{(3)}}{\partial w^{(2)}} = n^{(2)}$. In general

$$\frac{\partial J}{\partial w^{(l)}} = \delta^{(l+1)} \cdot n^{(l)} \quad (96)$$

where $n^{(l)}$ is known from the forward pass and $\delta^{(l+1)}$ is not known, except for the last one $\delta^{(4)}$, since this is simply the output error. So we need to find a way to get all the other δ s. We have

$$\delta^{(4)} = \frac{\partial J}{\partial z^{(4)}} = \frac{\partial J}{\partial n^{(4)}} \frac{\partial n^{(4)}}{\partial z^{(4)}} \quad (97)$$

and

$$\delta^{(3)} = \frac{\partial J}{\partial z^{(3)}} \quad (98)$$

$$= \frac{\partial J}{\partial n^{(4)}} \frac{\partial n^{(4)}}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial n^{(3)}} \frac{\partial n^{(3)}}{\partial z^{(3)}} \quad (99)$$

$$= \delta^{(4)} \frac{\partial z^{(4)}}{\partial n^{(3)}} \frac{\partial n^{(3)}}{\partial z^{(3)}} \quad (100)$$

$$= \delta^{(4)} w^{(3)} a'(z^{(3)}) \quad (101)$$

Generalized, this is

$$\delta^{(l)} = \delta^{(l+1)} w^{(l)} a'(z^{(l)}) \quad (102)$$

such that

$$\frac{\partial J}{\partial w^{(l)}} = \delta^{(l+1)} n^{(l)} \quad (103)$$

and using this, all the gradients can be computed recursively.

In contrast to the finite difference method, you don't have to go through each J and each w and calculate different updates. Instead, only one single feedforward is needed along with a single feedback. Note that no approximation was used anywhere, this is simply applying the chain rule.

The above formula only works for scalar artificial neural networks. For vectors, it looks like this:

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \delta_j^{(l+1)} n_i^{(l)} \quad (104)$$

where $\delta^{(l+1)}$ is now a vector

$$\delta^{(l)} = [\mathbf{W}^{(l)} \delta^{(l+1)}] \odot [a'(\mathbf{z}^{(l)})] \quad (105)$$

and \odot is the *Hadamard* product (element-wise multiplication). The second part of this is also a vector, since the gradient of the loss function is applied to vector $\mathbf{z}^{(l)}$ component-wise. A nice visualization of what is happening was created by [3blue1brown](#)

6.7 Summary

The greatest difficulty for using gradient descent in neural networks is computing the gradient $\frac{\partial J}{\partial \mathbf{W}}$. We solved this using *backpropagation*. The remaining problem here is that in very *deep networks*, the gradient becomes ever smaller the further back it is propagated from the output layer. This happens because of the sigmoid activation, which produces exceedingly small number for inputs sufficiently smaller than 0. This causes *vanishing gradients* or with other activation functions *exploding gradients*, and results in suboptimal weight configurations.

Generally, networks with *skip connections* are possible:

```
from matplotlib.patches import Arc

fig = plt.figure(figsize=(8,8))
ax = plt.subplot(111)
ax.set_aspect("equal")

plot_circle_and_line(ax, [1, 1], [3, 1], 1.1, r"$\dots$", "", [0.6, 0.3])
plot_circle_and_line(ax, [3.5, 1], [5.5, 1], 1.1, r"$n^{(l)}$", "", [1.7, 0.3])
plot_circle_and_line(ax, [6, 1], [8, 1], 1.1, "$n^{(l+1)}$", "", [2.8, 0.3])
plot_circle_and_line(ax, [8.5, 1], [10.5, 1], 1.1, r"$n^{(l+2)}$", "", [1.7, 0.
    ↪3])

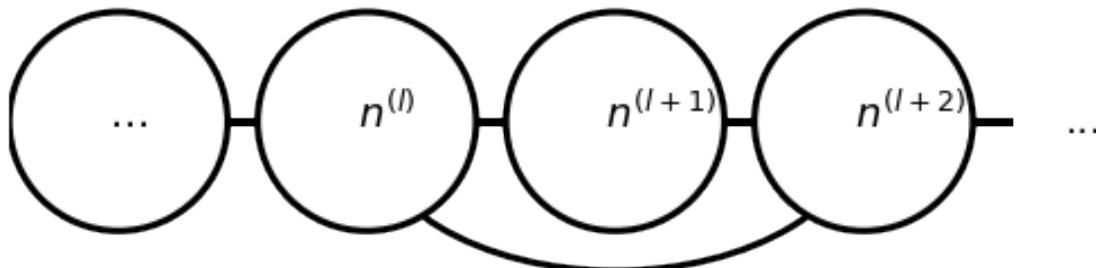
arc = Arc([6, 1], 5, 3, angle=0.0, theta1=180.0, theta2=360.0, lw=3)
ax.add_artist(arc)

ax.text(10.5, 0.9, "...", fontsize = 18, zorder=11)

ax.set_xlim([-0.1, 10])
ax.set_ylim([-0.7, 2.2])

ax = fig.gca()
ax.axis('off')
```

(-0.1, 10.0, -0.7, 2.2)



Packages like *TensorFlow* implement these networks as *graphs* and compute the derivative on that

graph.

We saw in lecture 03 that the starting point for gradient descent can have a huge impact on where the algorithm converges (in non-convex landscapes), so we need to think about how to initialize the weights sensibly. We will discuss this later.

You might have asked yourself how many layers and how many neurons per layer are a sensible choice for a given task. Unless the actual function mapping your inputs to the desired outputs is known (or a good portion of general information about it), this is an open problem. These numbers are also hyperparameters and there are a few techniques for finding a suitable configuration.

So far, we only talked about using sigmoid and softmax as nonlinear activations. Often used alternatives include tanh, which produces outputs in the range $[-1, 1]$, ReLU, which is a linear activation for positive numbers and 0 elsewhere, leakyReLU, which is a modification of ReLU trying to mitigate the vanishing gradients problem (and not quite succeeding), and SELU, which finally does the trick. These will be discussed in a later lecture.

7 Convolutional Neural Networks

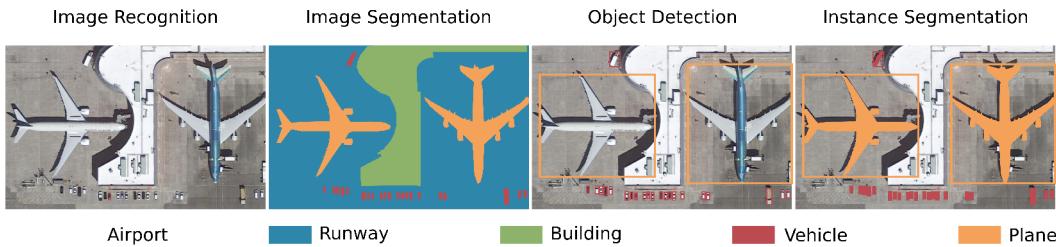
7.1 Convolutional Neural Networks

We've seen in assignment 02 that ANNs are perfectly capable of performing image-based tasks. They are not adjusted to this kind of input and hence, suffer from detriments. For example, a lot of neurons and layers are needed to capture the whole image and the information contained, all the while the layers are fully-connected.

A type of network better adjusted to these kinds of tasks is a **convolutional neural network**, which uses the same type of elementary units as the normal ANN, i.e. neurons, activation functions, weights, and a loss function. The format in which these are combined and connected allows for some architectural modifications, that in turn allow sparse connections between layers and weight sharing between neurons in hidden layers. CNNs are mostly used for image-based tasks, like image recognition, object detection, semantic segmentation, or image analysis. They can be combined with any other approach discussed later in the lecture, like recurrent neural networks or generative models such as autoencoders or GANs.

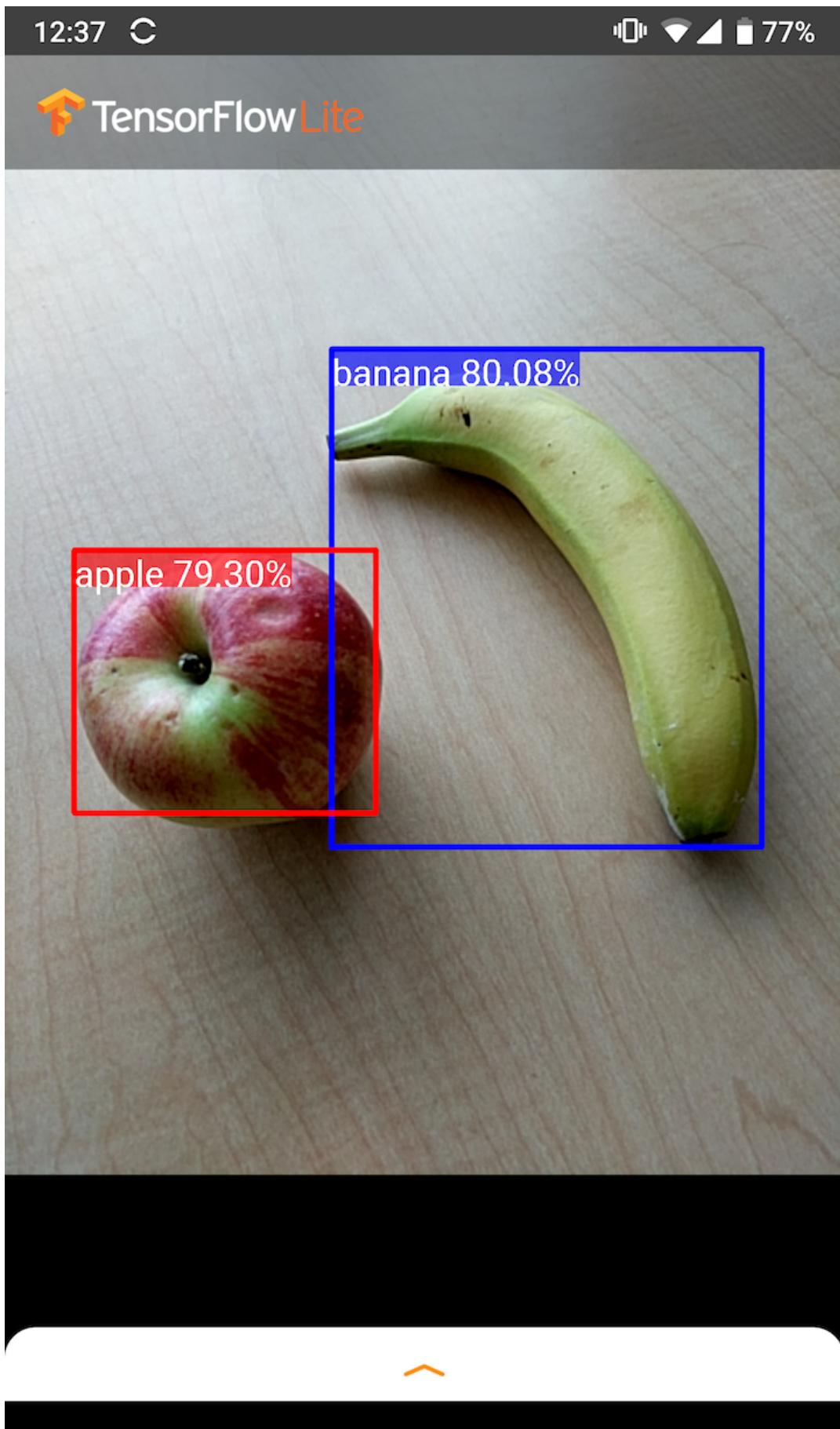
An example for a semantic segmentation is the following image (taken from [mdpi](#)):

```
from IPython.display import Image
Image("img/image_segmentation_example.png", width=1000)
```



Here, an example segment from the [DOTA dataset](#) was used to identify objects and mark pixels belonging to those objects with different colors. This can be interpreted as a classification problem, where each color belongs to a class. If you want to try how well these things work, there are example apps from TensorFlow Lite, which is a light TensorFlow framework for “edge devices” (the devices where models are actually used on, but not trained on), free to use that give some astonishing results. An example taken from the [TensorFlow Lite object detection example page](#):

```
Image("img/android_apple_banana.png", height=900, width=400)
```



Usually, this works much better. There are huge caveats though. While detection works well in “friendly” environments, *adversarial examples* can be generated easily. These are techniques created to fool image detection networks, and they work remarkably well. We will discuss some techniques for creating such examples later in the lecture. An example for such a fooled network is the following, which also works in 3d:

```
Image("img/turtle_rifle.png", width=800)
```



In this example, the texture of the turtle had to be adjusted completely, but there are less obvious examples. One team managed to fool Facebook’s image recognition by only changing a few pixel values in an image, such that every image was classified as an ostrich. These manipulations weren’t visible to the human eye.

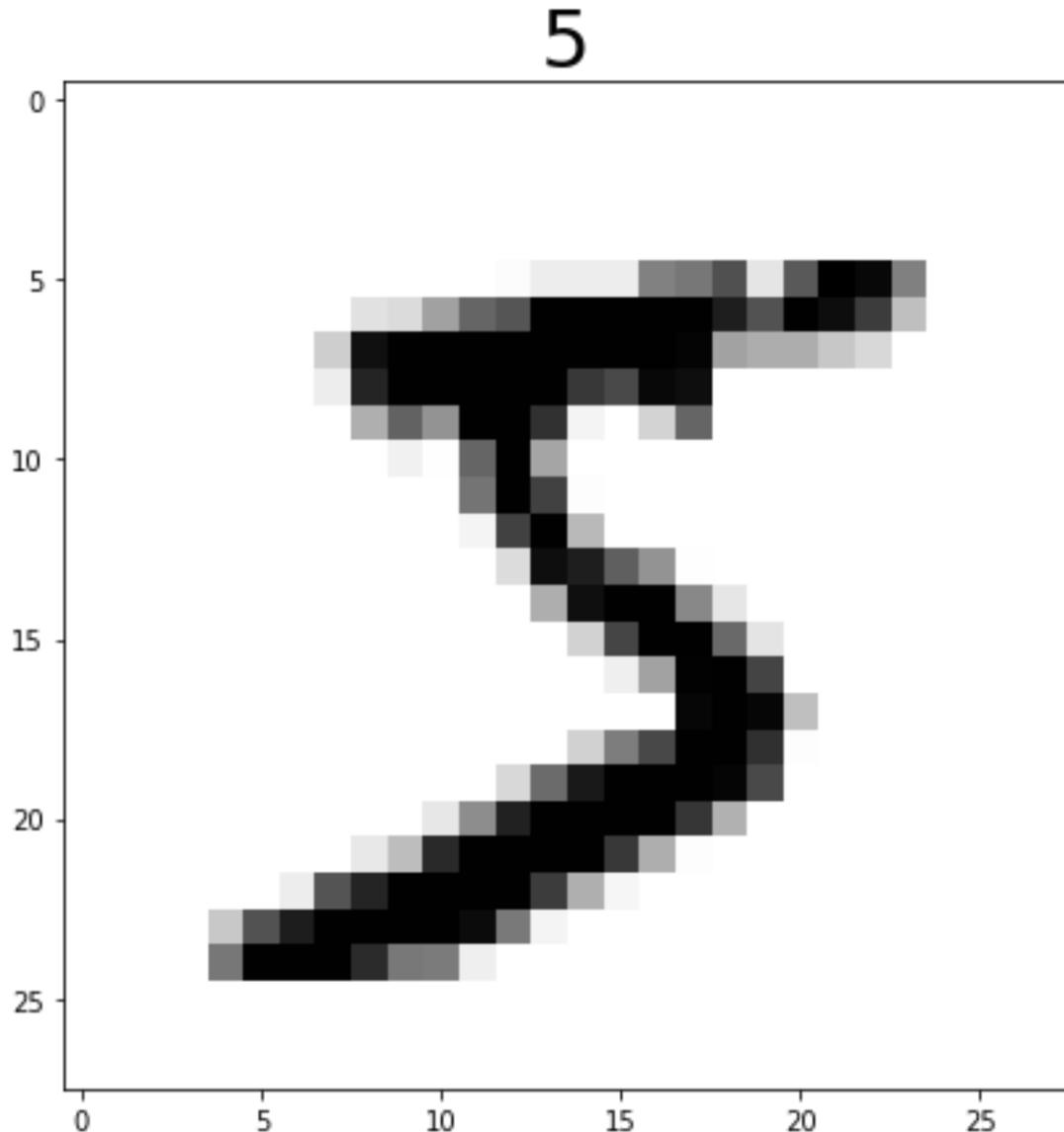
It is very important to make sure that these kind of techniques are not naively employed in high-security or life-threatening situations without making absolutely sure that the system is impenetrable at least to the most obvious adversarial techniques. But let’s continue exploring what CNNs actually are.

The rise of CNNs for image classification started when it was found that they performed extraordinarily well on the [ImageNet dataset](#). ImageNet is a database of over 14 million labeled images. Every year, a challenge is posed of proposing a model that has the highest accuracy in detecting the objects displayed in the images. For each label, there are 500 images on average. You can find a table at the [Keras application page](#), where several common models are compared by their top-1 accuracy, which measures how many times the label with the largest probability determined by the model hit the correct ground truth. They are also compared regarding their top-5 accuracy, which is the percentage of times the ground truth was among the 5 most probable suggestions made by

the model. The table also shows the number of parameters that were trained in the model, and they regularly exceed 10 million. The error rate for human performance on the ImageNet dataset is $\approx 5\%$, and the models mentioned on the Keras website approach that value, one even exceeding it.

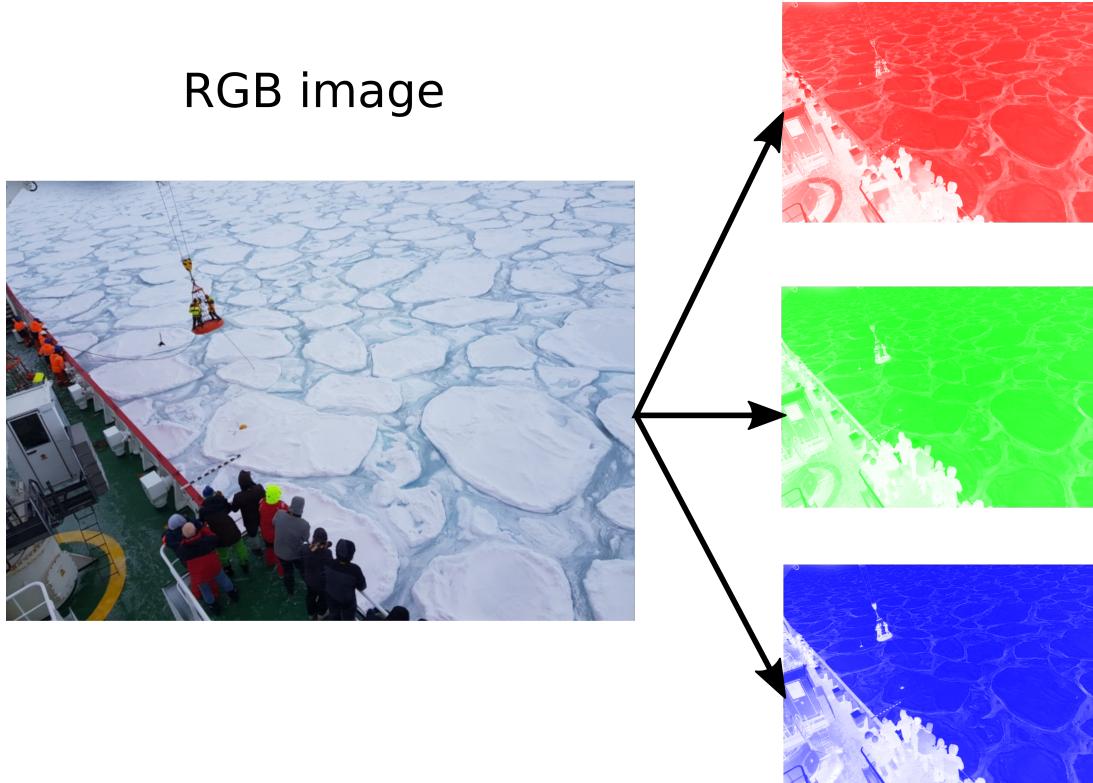
Recall the image format from the MNIST dataset from last lecture:

```
Image("img/5.png", height=400, width=400)
```



Last time, we fed this into a neural network by flattening the matrix to a vector, but actually this is a 2D matrix of pixels with 8bit values from 0 to 255 (or 0 to 1, depending on the format). For RGB images, the matrix is 3D, with RGB values for each pixel. Each matrix for a certain color is called a *channel*.

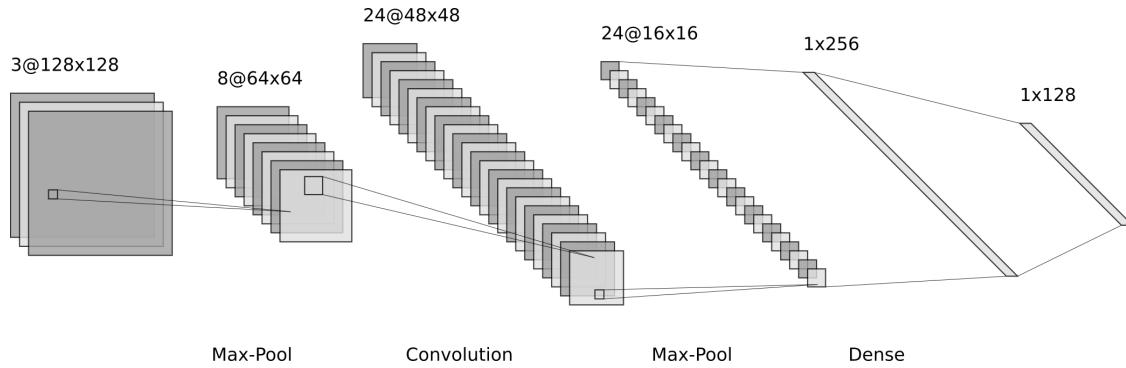
```
Image("img/rgb_example.png", height=800, width=1000)
```



The flattening process obfuscates (but does not destroy) spatial relations between parts of the image. **Convolutional neural networks**, as opposed to standard artificial neural networks, exploit the spatial composition of an image, which in turn leads to *sparse connections* between input and output neurons, as well as *parameter sharing*. A vanilla neural network with an input of, say, a 256x256 pixel RGB image needs $\sim 10^5$ neurons for the input. With a conservative 1000 hidden neurons, $\sim 10^8$ weights would be necessary to model the connections. This kind of network becomes extremely large very fast. In contrast, a CNN has a lot fewer weights, leveraging sparse connections and parameter sharing (basically the local connectivity of the network).

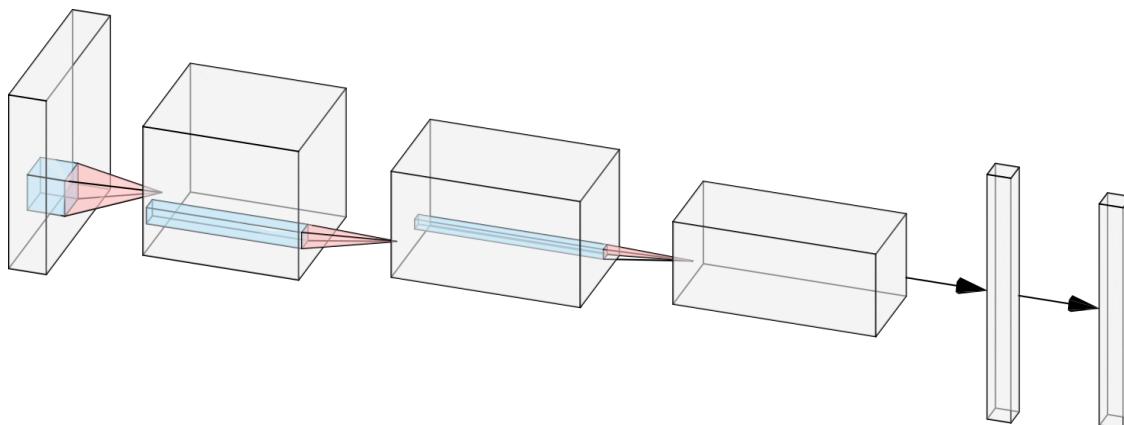
CNNs still consist of stacks of layers, followed by an output layer (mostly a classification layer), where each layer performs either a *convolution* or a *pooling* operation. These layers are stacked alternatively and finally lead to a series of fully connected layers, which in turn lead to the output layer. Sometimes the final layers are also convolutional, which is the case for example when another image should be produced as the output.

```
Image("img/g228.png", width=1000)
```



The CNN in the image above (created with [NN-SVG](#)), the input is an image “volume” (basically a subvolume of the full RGB image), and each layer outputs another 3D volume after a convolution or pooling operation. This is in contrast to standard neural networks, where each layer outputs a vector of a certain size.

```
Image("img/path800.png", width=1000)
```

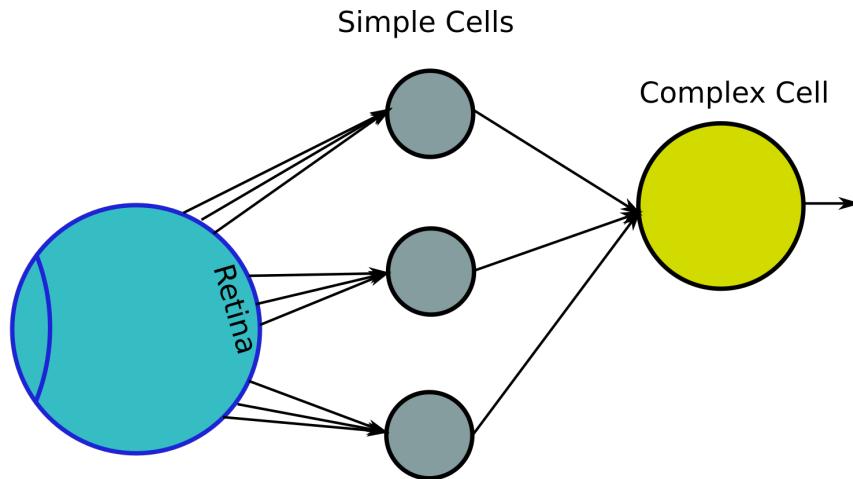


In the above picture, every 3D image “volume” can be sliced into 2D matrix outputs. Each of these outputs is called a **feature map/activation map**. Even if the input is single-channel (say, a grayscale image), the output can be multi-channel. This happens for example in image segmentation with grayscale inputs and colored classifications.

7.2 Convolutions

So why are we using convolutions? This is again inspired by biology, this time by neurons responsible for eyesight.

```
from IPython.display import Image
Image("img/cortex_cells.png", width=1000)
```



The finding depicted in the image above was awarded with (half) a Nobel prize in biology in 1981 for *David H. Hubel* and *Torsten Wiesel*. What happens in optical neurons is invariant and equivariant under translation. We will see how that works in a moment. Here, the “simple cells” respond linearly to elementary geometric features, like edges. “Complex cells” perform linear combinations and produce rectified output from the activations of the simple cells.

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(12,8))

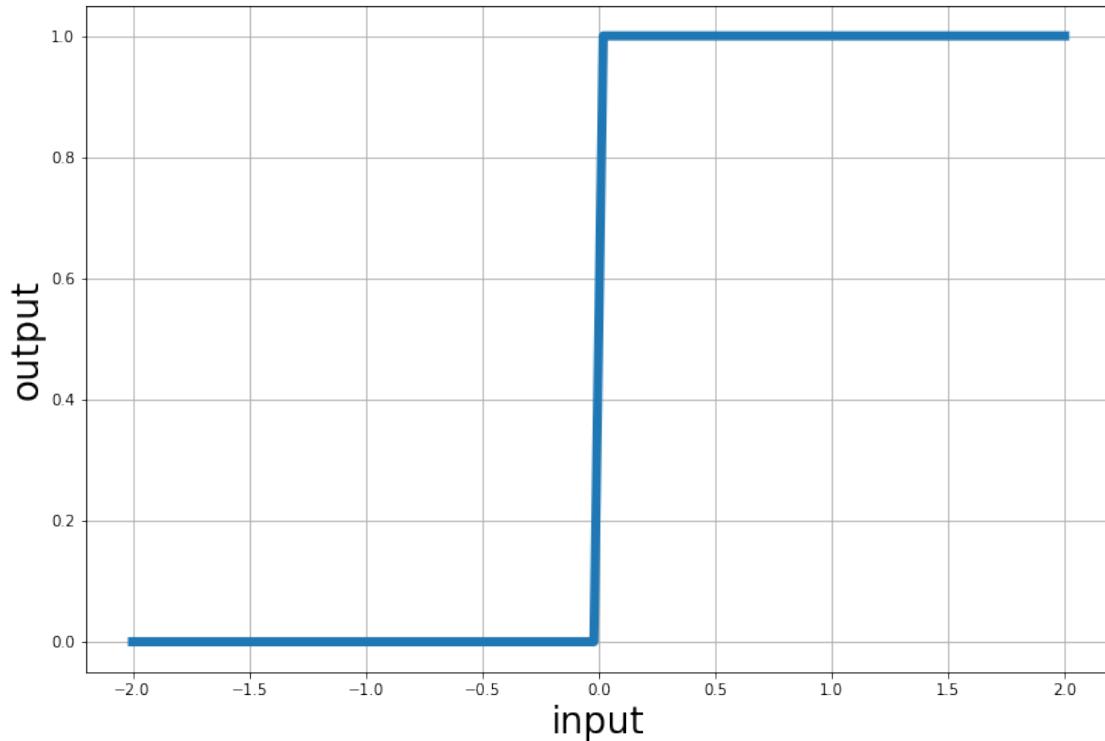
xrange = np.linspace(-2, 2, 201)

plt.plot(xrange, 0.5*(np.sign(xrange)+1), lw=6)

plt.xlabel(r"input", fontsize=24)
plt.ylabel(r"output", fontsize=24)

plt.grid()

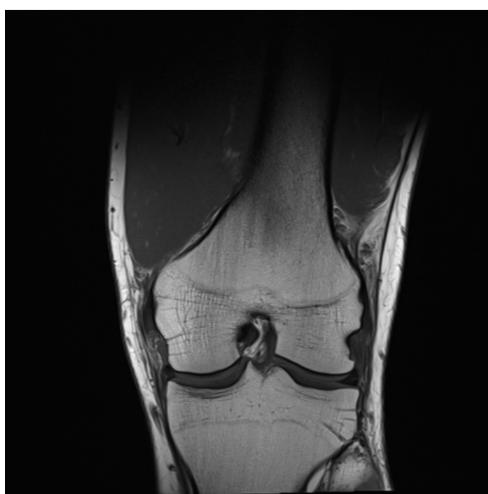
plt.show()
```



CNNs were designed to mimick this process by using **convolutions**, for which **filters/kernels** need to be defined. A simple way to understand the edge detection mechanism is the following image:

```
Image("img/knee_edges.png", width=1000)
```

1	-1
---	----



The *kernel* $[1 - 1]$ in this case detects abrupt changes from black to white, or vice versa. It produces a map of all edges in the original image. There are a multitude of manually generated filters like this (Sobel, Previn, Laplace, ...). The *receptive field* is the dimension of a filter kernel. In this case, the receptive field is of size 1×2 , but in general, filters can be much larger than that. Applying a single filter to the whole image is the first *convolution* of a series of convolutions that will happen in a CNN. What makes CNNs stand out here is that they will *learn* good filters for a given task, instead of necessitating the user to define them.

Why is this called a convolution? [Wikipedia](#) has a nice animation to show what happens in a standard convolution:

```
Image("img/Convolution_of_box_signal_with_itself2.gif", width=700)
```

```
<IPython.core.display.Image object>
```

The convolution itself measures the overlap of a moving filter and a given function over “time”. Convolutions for CNNs are a weighted, discrete version of this, and do pretty much the exact same thing. In this gif, it’s clear that the output is translationally invariant. If the given function moves to the left or right, the resulting graph is also moved left or right, but won’t change its form in any way. Edges in an image are always basically the same, just rotated and/or translated (or reversed). Edges are also found at multiple spots in an image. Still, only a single filter is needed to detect all of them. This is meant with **parameter sharing**. Instead of having to define multiple filters/kernels, a single one is enough to detect all edges in an image.

In convolution layers, every neuron in the next layer is connected to a small neighborhood in the input layer by a weight matrix. In this case, this is the kernel/filter itself. It is possible to define multiple kernels for every convolutional layer, where each produces a different output and each kernel is swept over the input image. This happens for example with RGB images, where every color channel is probed with a different filter. Each kernel produces 2D output, and outputs correspond to each kernel. These 2D outputs (feature maps) are usually stacked to a 3D output, such that the output of the whole convolutional layer is 3D. The filtering process looks like this:

```
%matplotlib inline
from ipywidgets import interact
import ipywidgets as widgets

input_size  = (5, 5)
kernel_size = (3, 3)

input_matrix = np.random.randint(0, 10, size=input_size)
kernel_matrix = np.random.randint(0, 10, size=kernel_size)
conv_matrix = np.zeros((input_size[0] - kernel_size[0] + 1, input_size[1] - kernel_size[1] + 1), dtype=np.int)
```

```

def convolve(step=0):
    fig, axs = plt.subplots(1,3, figsize=(8,8))
    axs.flatten()

    axs[0].matshow(input_matrix, cmap=plt.cm.Blues)
    axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

    axs[0].cla()
    axs[2].cla()

    axs[0].matshow(input_matrix, cmap=plt.cm.Blues)
    axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

    for i in range(input_size[0]):
        for j in range(input_size[1]):
            c = input_matrix[j,i]
            axs[0].text(i, j, str(c), va='center', ha='center')

    for i in range(kernel_size[0]):
        for j in range(kernel_size[1]):
            c = kernel_matrix[j,i]
            axs[0].text(i + step%(input_size[0]-kernel_size[0]+1), j + step//(
                input_size[1]-kernel_size[1]+1), str(c), va='top', ha='left', color='darkorange')

    for i in range(kernel_size[0]):
        for j in range(kernel_size[1]):
            c = kernel_matrix[j,i]
            axs[1].text(i, j, str(c), va='center', ha='center')

    A = input_matrix[step//(input_size[1]-kernel_size[1]+1):step//(
        input_size[1]-kernel_size[1]+1)+kernel_size[1], \
                    step%(input_size[0]-kernel_size[0]+1):\
                    step%(input_size[0]-kernel_size[0]+1)+kernel_size[0]]
    print(A)
    conv_matrix[step//conv_matrix.shape[1], step%conv_matrix.shape[0]] = np.
    sum(np.multiply(A, kernel_matrix))

    axs[2].matshow(conv_matrix, cmap=plt.cm.Blues)

    for i in range(conv_matrix.shape[0]):
        for j in range(conv_matrix.shape[1]):
            c = conv_matrix[j,i]
            if c:
                axs[2].text(i, j, str(c), va='center', ha='center')

```

```

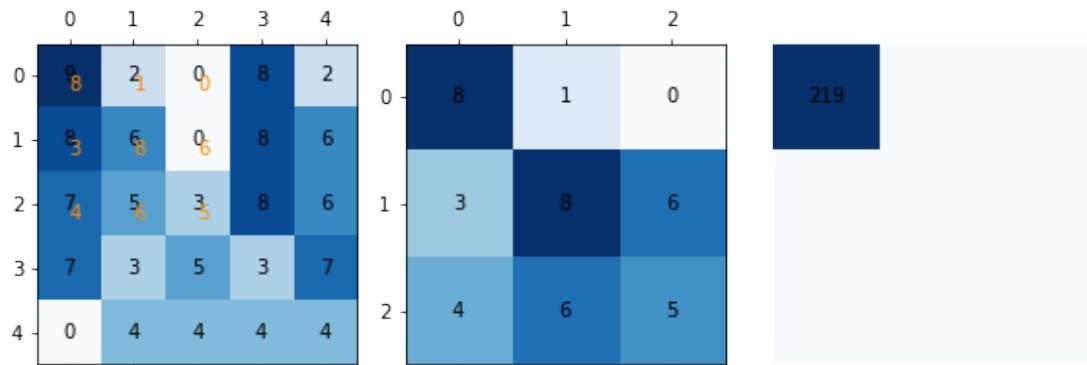
plt.axis('off')
plt.tight_layout()
plt.savefig("img/plot_conv.png")

interact(convolve, step=(0,_
    ↳(input_size[1]-kernel_size[1]+1)*(input_size[0]-kernel_size[0]+1)-1))

interactive(children=(IntSlider(value=0, description='step', max=8), Output()), _dom_classes=()

<function __main__.convolve(step=0)>

```



The kernel/filter is swept across the matrix. Then, a multiply accumulate (mac) is performed with

the 3×3 portion of the image and the result saved in the corresponding component of a new matrix. The filter is then moved to the right (below and to the start when it reaches the right side) and the process repeated for the next component of the resulting matrix. The output size for an $n \times m$ matrix with a kernel of size $f \times g$ is

$$(n \times m) * (f \times g) \implies (n - f + 1) \times (m - g + 1)$$

(can you see why?). The values in the kernel itself are the *weights*. Instead of being user-defined, these will be *learned* in CNNs.

In the end, the output will consist of K feature maps. Usually, they all have the same size.

The fact that the kernel weights are shared by all neurons in a hidden layer is called **weight sharing** and causes the network to be **translationally equivariant**, meaning that when you shift an object over an image, some neuron will still react the same way as before to its presence. We'll check this behavior in the last lesson today.

7.3 Pooling

This is what provides the translational invariance. Currently, feature maps produced by the convolutional layers have a size that depends on the size of the input image. **Pooling** will reduce the size of these feature maps by **subsampling**. Let's say we want to locate a crack in an image. Subsampling could for example reduce a feature map of size 256×256 to a size of 32×32 , which would be a reduction by a factor of 8. Then, if the crack "moves" by less than 8 pixels in the original image, the output here won't change by much. This reduces the size of feature maps significantly. As more layers are traversed in a network, the output becomes more and more invariant to finite translations of objects in the original image. The most common types of pooling are **average pooling** and **maxpooling**.

The code below shows what maxpooling does:

```
%matplotlib inline
from ipywidgets import interact
import ipywidgets as widgets
import numpy as np
import matplotlib.pyplot as plt

input_size = (4, 4)
kernel_size = (2, 2)

input_matrix = np.random.randint(0, 10, size=input_size)
kernel_matrix = np.ones(kernel_size)
pool_matrix = np.zeros((input_size[0] - kernel_size[0] + 1, input_size[1] - kernel_size[1] + 1), dtype=np.float)

def convolve(stride=2, step=0, pooling="maxpooling"):
    # dirty hack, vertical stride doesn't work yet
    if step == 2 & stride == 2:
        pass
```

```

else:
    fig, axs = plt.subplots(1,2, figsize=(8,8))
    axs.flatten()
    axs[0].matshow(input_matrix, cmap=plt.cm.Blues)
    axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

    axs[0].cla()
    axs[1].cla()

    axs[0].matshow(input_matrix, cmap=plt.cm.Blues)

    for i in range(input_size[0]):
        for j in range(input_size[1]):
            c = input_matrix[j,i]
            axs[0].text(i, j, str(c), va='center', ha='center')

    for i in range(kernel_size[0]):
        for j in range(kernel_size[1]):
            c = kernel_matrix[j,i]
            axs[0].text(i + (stride*step)%(input_size[0]-kernel_size[0]+1), \
→ j + (stride*step)//(input_size[1]-kernel_size[1]+1), " ", va='top', \
→ ha='left', color='darkorange')

    A = input_matrix[(stride*step)//(input_size[1]-kernel_size[1]+1): \
→ (stride*step)//(input_size[1]-kernel_size[1]+1)+kernel_size[1], \
→ (stride*step)%(input_size[0]-kernel_size[0]+1): \
→ (stride*step)%(input_size[0]-kernel_size[0]+1)+kernel_size[0]]
    print(A)
    if pooling == "maxpooling":
        pool_matrix[step//pool_matrix.shape[1], step%pool_matrix.shape[0]] \
→= np.max(A)
    else:
        pool_matrix[step//pool_matrix.shape[1], step%pool_matrix.shape[0]] \
→= np.mean(A)

    axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

    for i in range(pool_matrix.shape[0]):
        for j in range(pool_matrix.shape[1]):
            c = pool_matrix[j,i]
            if c:
                axs[1].text(i, j, str(c), va='center', ha='center')

plt.axis('off')
plt.tight_layout()

```

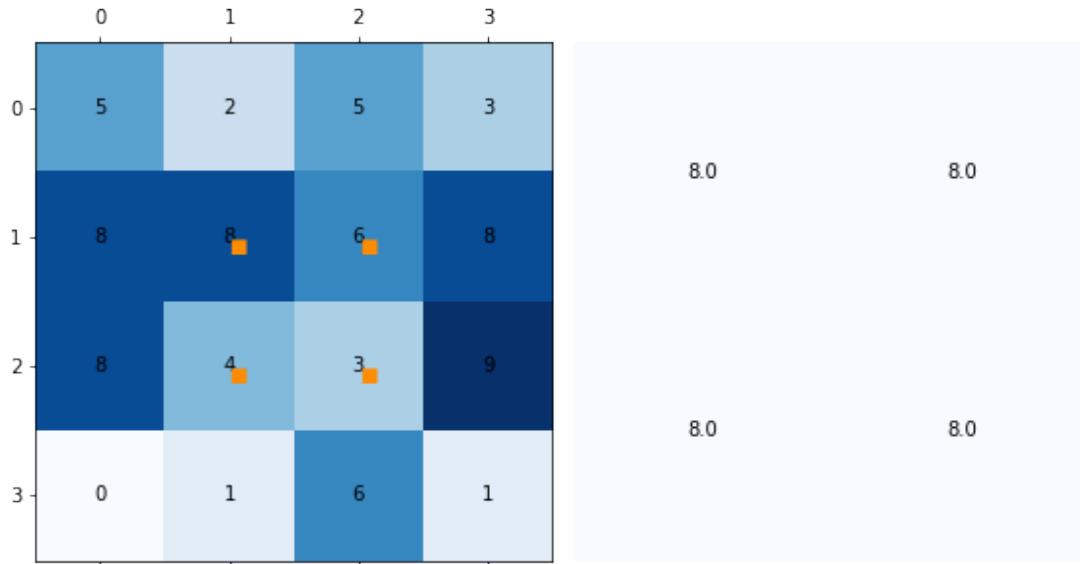
```

plt.savefig("img/plot_pool_stride.png")

interact(convolve, stride=(1, 2), \
         step=(0, ↗((input_size[1]-kernel_size[1]+1)*(input_size[0]-kernel_size[0]+1)-1)//2), \
         pooling=["maxpooling", "average pooling"])

interactive(children=(IntSlider(value=2, description='stride', max=2, min=1), IntSlider(value=1, description='step', max=1, min=0), IntSlider(value=0, description='pooling', options=['maxpooling', 'average pooling']), Output()), _main_=convolve)

```



You can think of this again as having a kernel, in this case of size 2×2 , that is swept across the input matrix. The **stride** is the number of steps the kernel moves in each iteration. A stride of 2 in

the above example would make the kernel never overlap, and this is the standard for maxpooling. The result of maxpooling is that just the *largest value* in the area of the kernel is taken as the corresponding component of the output matrix.

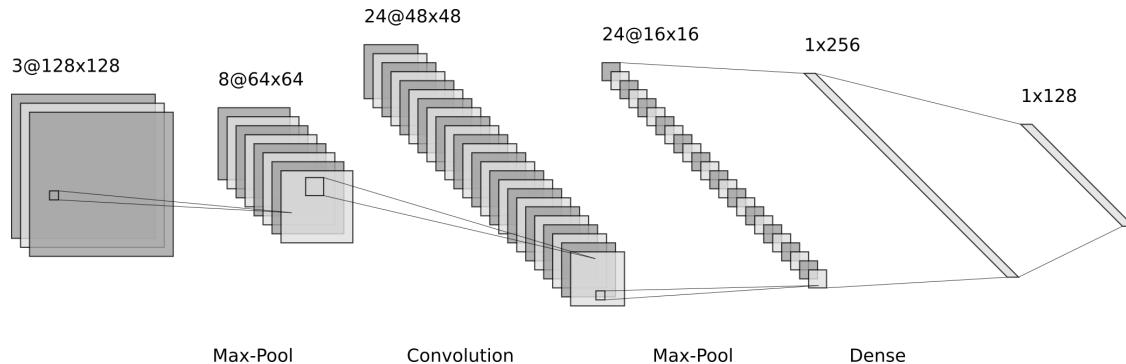
For *average pooling*, the resulting value would be the *average* of the values in the kernel region instead.

This behavior causes (very limited) **translational invariance**, meaning that if we slightly translate the image into one (or more) direction(s), the pooling operation will yield an extremely similar output. We'll see this in the last lesson today.

7.4 Types of Convolutions

7.4.1 Volume Convolutions

```
from IPython.display import Image
Image("img/g228.png", width=1000)
```



Usually, not the simple type of convolution we talked about before is used, but a more advanced one. In general convolutions are *volume convolutions*. Every layer of a CNN has the same basic function. It takes as input n 2D slices of numbers and produces as ouput n 2D slices of numbers of a different size. So actually when we say that a kernel has size 3×3 , we actually mean it has size $3 \times 3 \times 3$ (for RGB images with 3 channels). Each kernel slice operates on its corresponding channel. A value in the feature map is the sum of all the outputs. Each filter also has a bias. If K kernels are defined, they will produce K feature maps as outputs. The next filter then is of size $3 \times 3 \times K$, which works across all feature maps (slices of the last output). Typically filters have a size of 3×3 or 5×5 . All values in a filter will be unique for each channel, and each kernel for a channel will be learned separately.

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from matplotlib import cm, colors, colorbar
from mpl_toolkits.mplot3d.art3d import Poly3DCollection, Line3DCollection
```

```

# this is the old create_paralleliped_3d function, where origin is added ↵
↪everywhere
def create_cube_3d(origin, sidelength):
    v1, v2, v3 = [sidelength * np.eye(3)[i] for i in range(3)]
    origin -= 0.5*(v1 + v2 + v3)
    return [[origin, origin+v1, origin+v1+v2, origin+v2], \
            [origin+v3, origin+v1+v3, origin+v1+v2+v3, origin+v2+v3], \
            [origin, origin+v1, origin+v1+v3, origin+v3], \
            [origin+v1+v2, origin+v2, origin+v2+v3, origin+v1+v2+v3], \
            [origin+v1, origin+v1+v2, origin+v1+v2+v3, origin+v1+v3], \
            [origin+v3, origin+v2+v3, origin+v2, origin]]
fig = plt.figure(figsize=(10,10))
ax = fig.gca(projection='3d')

cuts = 9

xs = np.linspace(-cuts//2, cuts//2, cuts)
ys = np.linspace(-cuts//2, cuts//2, cuts)
zs = np.linspace(-cuts//2, cuts//2, cuts)
xg, yg, zg = np.meshgrid(xs,
                           ys,
                           zs)

data = np.zeros((cuts,cuts,cuts))
for i in range(-cuts//2, cuts//2, 1):
    for j in range(-cuts//2, cuts//2, 1):
        for k in range(-cuts//2, cuts//2, 1):
            if abs(i)+abs(j)+abs(k) < 2.5: #np.sqrt(i**2+j**2+k**2) < 2.5:
                data[i+cuts//2,j+cuts//2,k+cuts//2] = 1

norm = colors.Normalize(vmin=-2, vmax=2)
colors = lambda i,j,k : cm.ScalarMappable(norm=norm,cmap = "bwr_r") .
↪to_rgba(data[i,j,k])

for i, xi in enumerate(xs):
    for j, yj in enumerate(ys):
        for k, zk in enumerate(zs):
            ax.add_collection3d(Poly3DCollection(create_cube_3d([xi,yj,zk], 1.
↪1), \
                                           facecolors="steelblue" if data[i,j,k]==1 else "white", ↵
↪linewidths=1, \
                                           edgecolors="gray" if data[i,j,k]==1 else None, alpha=.15 if ↵
↪data[i,j,k]==1 else 0.0))

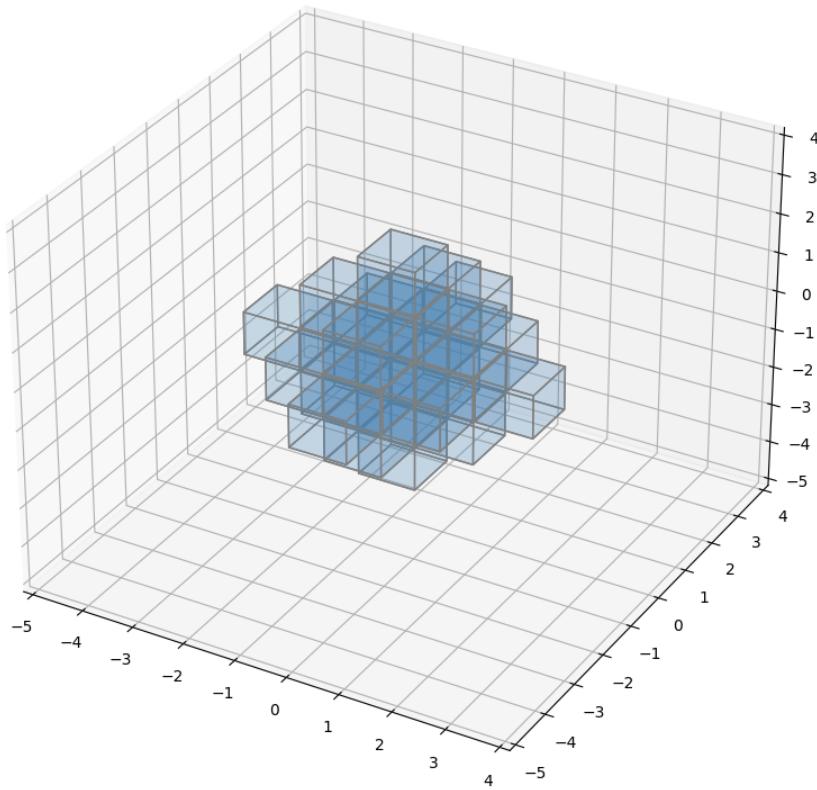
```

```
ax.set_xlim([-cuts//2, cuts//2])
ax.set_ylim([-cuts//2, cuts//2])
ax.set_zlim([-cuts//2, cuts//2])

plt.savefig("img/plot_binvox.png")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```



There are also higher-dimensional kernels that produce higher-dimensional convolutions. E.g., 3D kernels that act on 3D matrices. Suitable inputs could be materials with a specific geometry,

expressed as a *binary voxel matrix*. Such a matrix is boolean, meaning it has entries `True` where there is material in a volume and `False` otherwise. Only using 2D-convolutions on such inputs would lose information about connections in the remaining direction of the material.

7.4.2 Padded Convolutions

As the number of convolutions in a deep CNN increases, the feature maps will become ever smaller. At some point, filtering will become impossible. An extreme example is a feature map of size 3×3 that is probed by a filter of the same size, resulting in an output of a 1×1 feature map. To mitigate this issue, **padding** is used, where usually zeros pad an input such the the size of the output feature map is preserved. One layer of zeros around a feature map corresponds to a padding of $p = 1$. The kernel can then probe the matrix just like before. The size of the output for an input matrix of size $n \times m$ and kernel of size $f \times g$ is now

$$(n \times m)_p * (f \times g) \implies (n - f + 1 + 2p) \times (m - g + 1 + 2p)$$

You can test a few configurations with the code below:

```
%matplotlib inline
from ipywidgets import interact
import ipywidgets as widgets

input_size = (4, 4)

input_matrix = np.random.randint(0, 10, size=input_size)

def pad_mat(padding):
    fig, axs = plt.subplots(1,2, figsize=(8,8))
    axs.flatten()
    axs[0].matshow(input_matrix, cmap=plt.cm.Blues)

    axs[1].cla()
    padded_matrix = np.pad(input_matrix, padding)

    for i in range(input_size[0]):
        for j in range(input_size[1]):
            c = input_matrix[j,i]
            axs[0].text(i, j, str(c), va='center', ha='center')

    axs[1].matshow(padded_matrix, cmap=plt.cm.Blues)

    for i in range(padded_matrix.shape[0]):
        for j in range(padded_matrix.shape[1]):
            c = padded_matrix[j,i]
            axs[1].text(i, j, str(c), va='center', ha='center')

    plt.axis('off')
```

```

plt.savefig("img/plot_padding.png")

interact(pad_mat, padding=(0, 3))

interactive(children=(IntSlider(value=1, description='padding', max=3), Output()), _dom_classes=['padding'])
<function __main__.pad_mat(padding)>

```

	0	1	2	3
0	2	9	2	1
1	0	9	7	3
2	2	7	0	9
3	8	2	3	5

	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	0	2	9	2	1	0	0
3	0	0	0	9	7	3	0	0
4	0	0	2	7	0	9	0	0
5	0	0	8	2	3	5	0	0
6	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Usually, padding is chosen such that there are no fractions in the new feature map size, say $p = \frac{f-1}{2}$.

7.4.3 Strided Convolutions

A stride of 1 is typical for normal kernels, but very large feature maps often take a long time to be filtered. Striding then makes sense to reduce computation time. This effectively corresponds to subsampling the image. A stride of $s = 1$ is the standard configuration that we've seen before, such that one step is taken to the right. The size of the output feature map for an input matrix of size $n \times m$ and kernel size $f \times g$ is now

$$(n \times m)_p * (f \times g)_{s,t} \implies \left(\frac{n-f+2p}{s} + 1 \right) \times \left(\frac{m-g+2p}{t} + 1 \right)$$

where t is a stride value for vertical movement. For convenience, filters mostly have odd-valued receptive fields. We have seen the effect of different strides in the last lesson.

7.4.4 Dilated Convolutions (Atrous Convolution)

This type of convolution increases the receptive field. An additional **dilation rate** d is introduced, that denotes the number of rows and columns added to a filter. A 3×3 kernel with $d = 2$ “sees” the same area a 5×5 kernel does, but only has 9 parameters/weights like before. So the receptive field is increased without increasing computational need. Instead of 9 convolutions, only a single one is performed in the example below:

```
fig, axs = plt.subplots(1,2, figsize=(14,14))
axs.flatten()

input_size  = (7, 7)
kernel_size = (3, 3)

input_matrix  = np.random.randint(0, 10, size=input_size)
kernel_matrix = np.random.randint(0, 10, kernel_size)
pool_matrix = np.zeros((3, 3), dtype=np.float)

stride = 2
step = 0
d = 2
kernel_mask = np.ones(input_matrix.shape, dtype=np.int)

for i in range(input_size[0]):
    for j in range(input_size[1]):
        c = input_matrix[j,i]
        axs[0].text(i, j, str(c), va='center', ha='center')

for i in range(kernel_size[0]):
    for j in range(kernel_size[1]):
        c = kernel_matrix[j,i]
        axs[0].text(i*d + (stride*step)% (input_size[0]-kernel_size[0]+1), \
                    j*d + (stride*step)//(input_size[1]-kernel_size[1]+1), \
                    str(c), va='top', ha='left', color='darkorange')
```

```

kernel_mask[i*d + (stride*step)%(input_size[0]-kernel_size[0]+1), \
            j*d + (stride*step)// \
            ↳(input_size[1]-kernel_size[1]+1)] = 0

for i in range(pool_matrix.shape[0]):
    for j in range(pool_matrix.shape[1]):
        c = pool_matrix[j,i]
        if i+j != 0:
            axs[1].text(i, j, str(c), va='center', ha='center')

A = np.ma.array(input_matrix, mask=kernel_mask)
A = A.compressed().reshape(3,3)

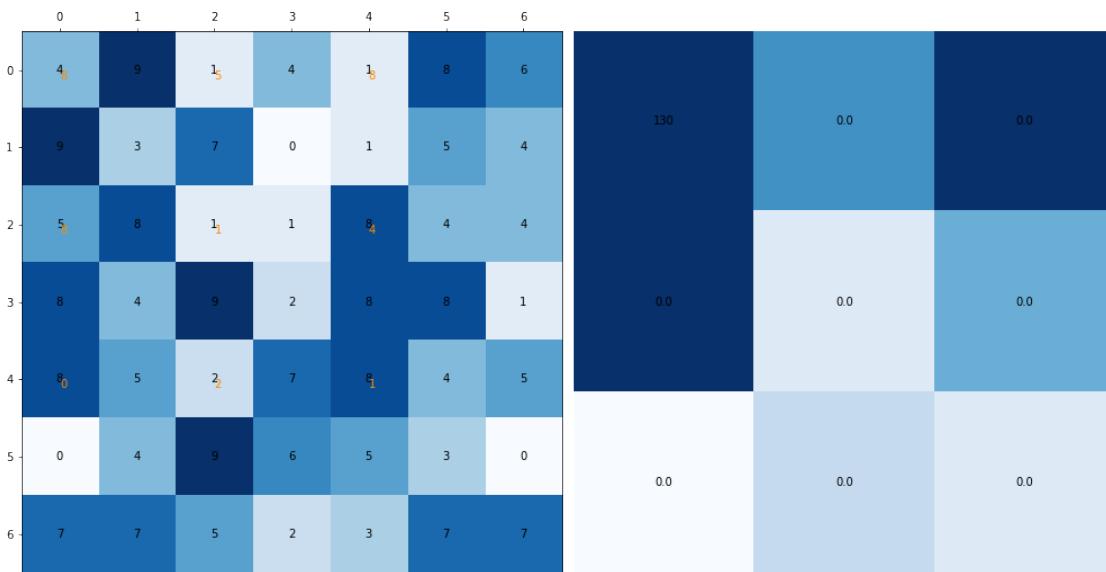
axs[1].text(0, 0, str(np.sum(np.multiply(A, kernel_matrix))), va='center', \
            ↳ha='center')

axs[0].matshow(input_matrix, cmap=plt.cm.Blues)
axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

plt.axis('off')

plt.tight_layout()

```



7.4.5 Transposed Convolution

A **transposed convolution** is the inverse of a convolution. So instead of creating output feature maps from input feature maps, output feature maps are transpose convoluted to regain a 3D feature map pack of the original size. Transposed convolutions are used to help increasing the size of feature

map outputs and are mostly found in encoder-decoder networks. The goal here is to perform a convolution in such a way that the original spatial resolution is regained, so the input has to be padded appropriately and a specific dilation has to be found. With an input feature map of size 2×2 , a kernel of size 3×3 and a desired output feature map size of $m \times m$, the question is what input feature map size $m \times m$ for a transposed convolution is necessary to get an output feature map size of 2×2 when this is run backwards. So basically the question is how to reverse this operation. Assume we want an output feature map size of 5×5 . What input feature map size, when convolved with a kernel of size 3×3 , yields an output feature map of size 5×5 ? Here, a zero padding of $p = 2$ and a dilation of $d = 2$ is necessary, which expands the input feature map size to 7×7 .

```

fig, axs = plt.subplots(1,2, figsize=(8,8))
axs.flatten()

input_size  = (7, 7)
kernel_size = (3, 3)

input_matrix  = np.zeros(input_size)
kernel_matrix = np.random.randint(0, 10, kernel_size)
pool_matrix = np.zeros((5, 5), dtype=np.float)

input_matrix[2, 2] = 5
input_matrix[4, 2] = 7
input_matrix[2, 4] = 3
input_matrix[4, 4] = 1

step = 0
d = 1 # this is the dilation for the kernel, not the input matrix
kernel_mask = np.ones(input_matrix.shape, dtype=np.int)

for i in range(input_size[0]):
    for j in range(input_size[1]):
        c = input_matrix[j,i]
        axs[0].text(i, j, str(c), va='center', ha='center')

for i in range(kernel_size[0]):
    for j in range(kernel_size[1]):
        c = kernel_matrix[j,i]
        axs[0].text(i*d + (stride*step)%(input_size[0]-kernel_size[0]+1), \
                    j*d + (stride*step)//(input_size[1]-kernel_size[1]+1), \
                    str(c), va='top', ha='left', color='darkorange')
        kernel_mask[i*d + (stride*step)%(input_size[0]-kernel_size[0]+1), \
                    j*d + (stride*step)//(input_size[1]-kernel_size[1]+1)] = 0

for i in range(pool_matrix.shape[0]):
    for j in range(pool_matrix.shape[1]):
```

```

c = pool_matrix[j,i]
if i+j != 0:
    axs[1].text(i, j, str(c), va='center', ha='center')

A = np.ma.array(input_matrix, mask=kernel_mask)
A = A.compressed().reshape(3,3)

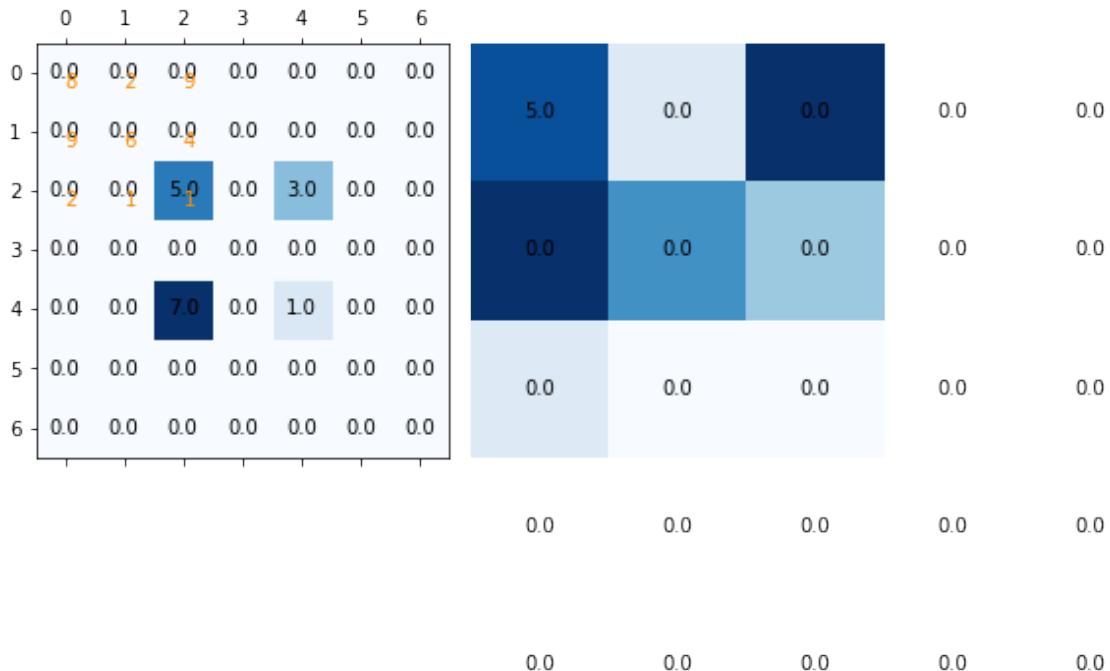
axs[1].text(0, 0, str(np.sum(np.multiply(A, kernel_matrix))), va='center',
            ha='center')

axs[0].matshow(input_matrix, cmap=plt.cm.Blues)
axs[1].matshow(kernel_matrix, cmap=plt.cm.Blues)

plt.axis('off')

plt.tight_layout()

```



7.5 CNN Forward and Backward Pass

A *feedforward* or *forward pass* in a CNN from feature map to feature map is again a sequence of matrix multiplications (applying a nonlinearity is always implicitly assumed in CNNs) to get the output. The *backward pass* or *backpropagation* are fed back by *transposed* matrix multiplications. The transformations from smaller size feature maps to larger size feature maps is a transposed convolution.

7.6 Transfer Learning

7.6.1 Starting Points for Optimization

Before coming to transfer learning, let us quickly review a few things about optimization. You don't need to understand the code below, but it may help in understanding gradient descent a little bit better. It uses the SymPy package, which allows symbolic manipulations of formulas. The next cell contains the setup for the symbolic calculations.

```

import sympy as sy
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from mpl_toolkits.mplot3d import axes3d
from matplotlib import cm
from sympy.vector import gradient, CoordSys3D

# initialize symbolic variables
x, y = sy.symbols('x, y')
# put them in a list for further processing
X = [x, y]
# define the symbolic function
f = sy.exp(-0.05*x**2 - 0.05*y**2) * sy.sin(x) * sy.cos(y)
# and its gradient
f_grad = sy.derive_by_array(f, X)
# and Hessian
f_hess = sy.derive_by_array(f_grad, X)

# lambdify makes the symbolic function callable, especially with numpy arrays
func      = sy.lambdify(X, f,      'numpy')
func_grad = sy.lambdify(X, f_grad, 'numpy')
func_hess = sy.lambdify(X, f_hess, 'numpy')

```

Just to check whether all these terms make sense, we can try a few different values (what does the Hessian tell you about the zeros of the cost function?):

```

print(func(0, 0))
print(func_grad(0, 0))
print(func_hess(0, 0))
print()
print(func(np.pi/2, np.pi/2))
print(func_grad(np.pi/2, np.pi/2))
print(func_hess(np.pi/2, np.pi/2))
print()
print(func(1000, 1000))
print(func_grad(1000, 1000))
print(func_hess(1000, 1000))

```

```

0.0
[1.0, -0.0]
[[0.0, 0.0], [0.0, 0.0]]

4.784350493243898e-17
[-7.515240180886863e-18, -0.7813437305474442]
[[-5.144736425857103e-17, 0.12273318619081464], [0.12273318619081464,
0.2454663723816293]]]

0.0
[0.0, -0.0]
[[0.0, 0.0], [0.0, 0.0]]

```

Next, we need an optimizer. We'll just use standard gradient descent that we introduced in the first lecture:

```

def take_grad_step(point, grad_f, ):
    return point - * np.array(grad_f(*point))

def gradient_descent(x_init, grad_f, , max_steps, precision):
    points = [x_init]
    for i in range(max_steps):
        points.append(take_grad_step(points[-1], grad_f, ))
        if np.allclose(points[-1], points[-2], atol=precision):
            print("Convergence after " + str(i) + "steps.")
            print("Convergence criterion " + str(points[-1] - points[-2]))
            break

    return np.array(points)

```

And visualizing the process for some loss landscape:

```

%matplotlib notebook
# choose a starting point here, restricted to [-5, 5] in both directions
# this is a very good starting point:
#x_init = [0.5, 0.0]

# this gives a suboptimal local minimum
#x_init = [2.0, 0.0]

# this is a rather bad one, why?
#x_init = [0.0, np.pi/2]

# this also leads to trouble, why?
#x_init = [-np.pi/2+0.2, -np.pi+0.2]

# learning rate, max_steps, and precision can be set by sliders below

```

```

# For plotting, we need z-values for all x- and y-values on a grid,
# as we've seen before. Thanks to `lambdify`, we can simply apply
# our function to numpy arrays and get a resulting array with all
# values we need:
x_m, y_m = np.meshgrid(np.arange(-5,5,0.1), \
                       np.arange(-5,5,0.1))

Z = func(x_m, y_m)

def plot_grad_landscape( , max_steps, precision, surface, contours, ↴
    starting_point):
    if starting_point == "Good":
        x_init = [0.5, 0.0]
    elif starting_point == "Suboptimal":
        x_init = [2.0, 0.0]
    elif starting_point == "Bad":
        x_init = [0.0, np.pi/2]
    elif starting_point == "Unstable":
        x_init = [-np.pi/2+0.2, -np.pi+0.2]

    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111, projection='3d')

    points = gradient_descent(x_init, func_grad, , max_steps, precision)

    # give infor on what kind of critical point was found:
    hess = np.array(func_hess(*points[-1]), dtype=np.float)
    , v = np.linalg.eig(hess)
    if np.allclose(func_grad(*points[-1]), np.zeros(2), atol=precision):
        if [0] * [1] > 0:
            if [0] > 0:
                print("Hessian is positive definite, this is a minimum")
            else:
                print("Hessian is negative definite, this is a maximum")
        else:
            print("Hessian is indefinite, this is a saddle point")
    else:
        print("This is not an extremal value within the set precision")

    # this is needed to plot consecutive points in different colors
    colors = cm.rainbow(np.linspace(0, 1, points.shape[0]))

    #plt.cla()
    if surface:
        ax.plot_surface(x_m, y_m, Z, rstride=8, cstride=8, alpha=0.5, cmap=cm.
            ↴ocean)

```

```

    ax.scatter(points[:,0], points[:,1], np.array(func(points[:,0]), points[:,1]))+0.1, \
               color=colors, lw=4, alpha=1, zorder=100)
#ax.quiver(*point, func(*point)+0.1,
#           *func_grad(*point), func(*func_grad(*point)), \
#           length=np.linalg.norm(func_grad(*point)), lw=3, \
#           color='darkorange', normalize=False, alpha=1)

if contours:
    ax.contour(x_m, y_m, Z, zdir='z', offset=np.min(Z)-1, cmap=cm.ocean, levels=15, alpha=0.7)
    #ax.contourf(X, Y, Z, zdir='x', offset=-5, cmap=cm.ocean)
    #ax.contourf(X, Y, Z, zdir='y', offset=5, cmap=cm.ocean)
    ax.scatter(points[:,0], points[:,1], points.shape[0]*[np.min(Z)-1], \
               color=colors, lw=4, alpha=1, zorder=100)
    #ax.quiver(*x_init, np.min(Z)-1,
    #           *func_grad(*x_init), 0, \
    #           length=np.linalg.norm(func_grad(*x_init)), lw=3, \
    #           color='red', normalize=False, alpha=1)

    #fig.colorbar(surf, ax=ax, shrink=0.5, aspect=5)

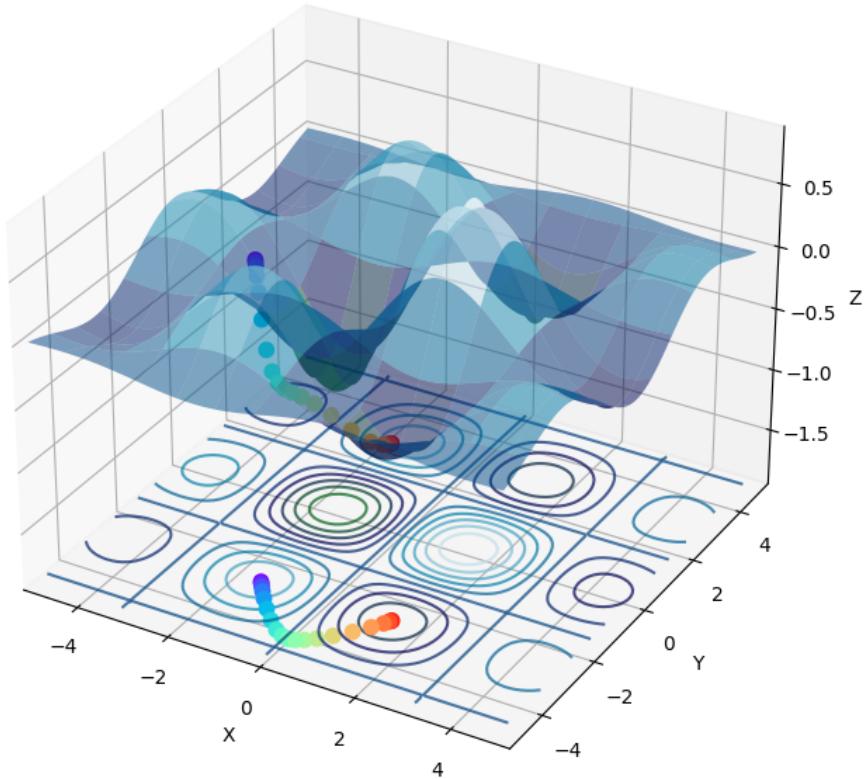
    ax.set_xlabel('X')
    ax.set_xlim(-5, 5)
    ax.set_ylabel('Y')
    ax.set_ylim(-5, 5)
    ax.set_zlabel('Z')
    ax.set_zlim(np.min(Z)-1, np.max(Z))

    plt.savefig("img/plot_opti_init.png")

interact(plot_grad_landscape, \
         =(0.1, 2.0, 0.1), \
         max_steps=(1, 20), \
         precision=(0.01, 0.1, 0.01), \
         surface=True, \
         contours=True, \
         starting_point=["Good", "Suboptimal", "Bad", "Unstable"])

interactive(children=(FloatSlider(value=1.0, description=' ', max=2.0, min=0.1), IntSlider(value
<function __main__.plot_grad_landscape(, max_steps, precision, surface, contours, starting_point)>

```



Try the different given starting points above. Each will lead to a different minimum, or to none at all. The second example gets this simple gradient descent algorithm stuck in a suboptimal local minimum, while the third example stays on a saddle point. The fourth example starting point starts pretty close to a maximum, where the direction of steepest descent depends on minor changes to the exact coordinates. In this case, it will swing towards a saddle point, slightly miss it and end up in a suboptimal local minimum, taking a lot of steps to converge.

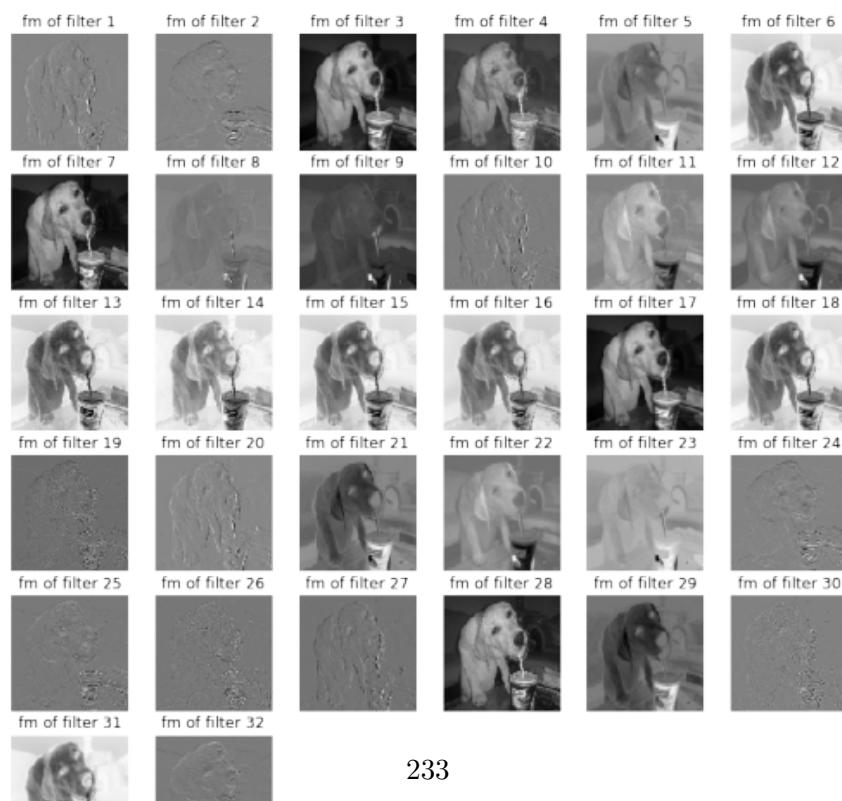
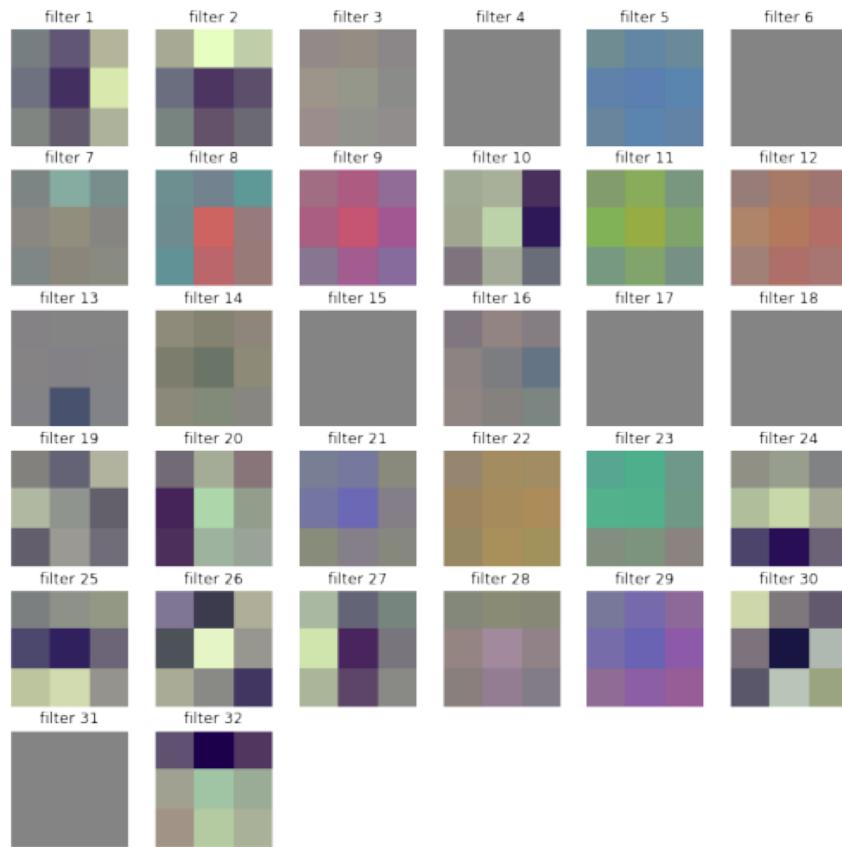
CNNs are difficult to train. In most cases, it makes sense to use a pretrained CNN as a feasible *starting point* for the actual task that is to be performed. The idea is the same as above; choosing a good starting point will help the optimizer finding a good minimum and hence produce better results.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```
filters = mpimg.imread('img/filters.png')
effects = mpimg.imread('img/filter_effect.png')

# use first for lecture
#fig, ax = plt.subplots(1,2, figsize=(20,12))
fig, ax = plt.subplots(2,1, figsize=(12,20))

ax[0].imshow(filters)
ax[1].imshow(effects)
ax[0].axis("off")
ax[1].axis("off");
```



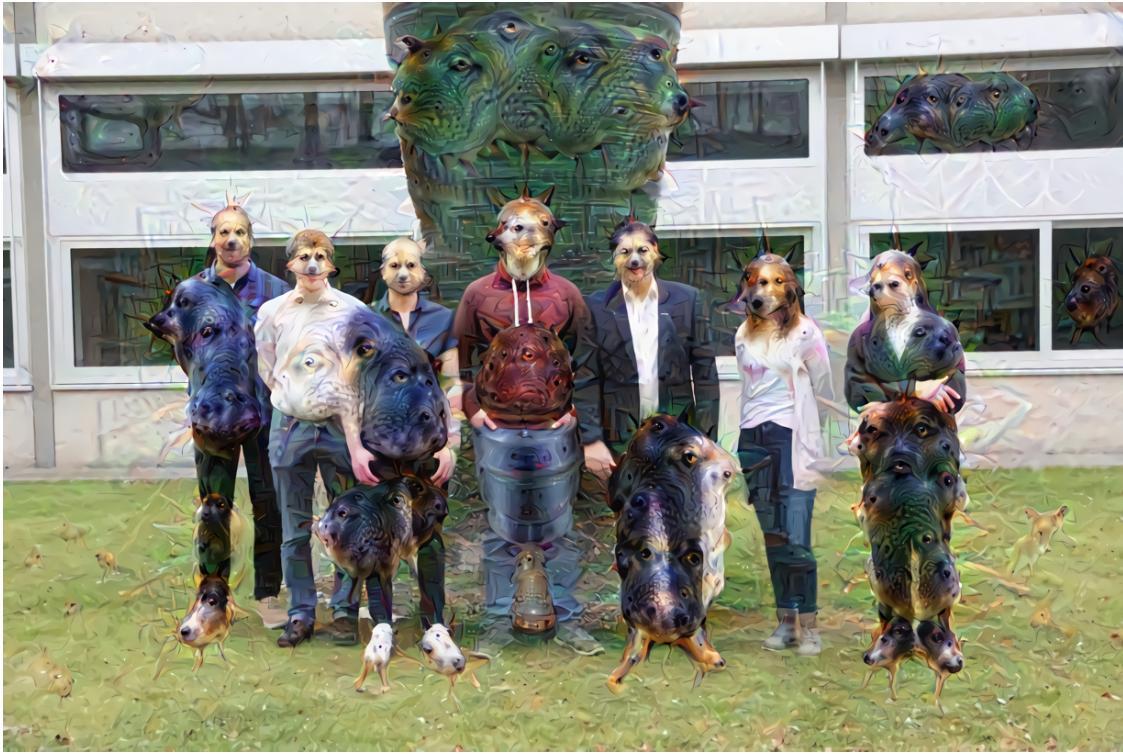
CNNs are generally difficult to train and quite data-hungry. It is possible to remove the need for full training by using pretrained CNNs as **feature extractors**. The idea here is to remove the classification layer in a pretrained network, feedforward your own data through this network, then feed this output to another ANN or CNN or other model. We will implore a bit why this is called feature extraction in the assignment today, but generally with features here prominent image features like edges and combinations of edges and even higher-level abstract geometries are meant. The feature extractor works by producing an embedding/representation of your data in terms of the high-level features in the last layer of the network, and this representation is then used to train other machine learning algorithms like support vector machines, ANNs, or logistic regression to classify the data.

7.6.2 Transfer Learning

Another possibility is to fine-tune pretrained networks. This works by loading a pretrained network completely. The output layer then has to be modified to have a number of neurons that is equal to the number of classes you're training for in the new dataset (where the weights for these new neurons are initialized randomly). Then, the network can be retrained with the new data. This is called **transfer learning**. Both versions work quite well. We will test this in one of the assignments today.

CNNs need a lot of data to be trained accurately and generating a large database of labeled data is quite expensive (there are some online services like Amazon's mechanical turk??). In deep CNNs, the last layers learn high-level, task-specific features. An example would be Google's ??net, that was initially trained for classifying certain breeds of dogs. The result of this can still be seen by taking the output of the last layers when a certain image is fed into such a network. This is called *deep dream*:

```
from IPython.display import Image  
Image("img/isd_dream.jpg", width=1000)
```



There are services that can extract such images ([Deep Dreams](#)) for given inputs. In this example, many dogs can be seen which is a result of the overabundance of dog breeds for the initial training of the network. It was later used to classify all kinds of animals without any specificity, so other animals can be seen as well. This can also be applied to [video input](#). Us humans aren't immune to this effect either, see e.g. the [uncanny valley effect](#). These last layers are easier to train, since they are inclose proximity to the classification layer and the error can be backpropagated to them easier than to earlier layers, where the *vanishing gradient* problems makes it difficult to adapt then. Earlier layers as said learn more generic features like edges, geometric patterns and combinations of these. Effectively, transfer learning applies the "knowledge" of an already trained network to a different, but related problem.

There are two possible strategies here: The initial layers of a pretrained model can be set constant and only the last few layers are retrained, or the entire model is set to "trainable mode", such that the pretrained weights are the starting point of the optimization (which should be a rather good starting point, as long as the problems are related). Take for example medical images, which are often provided in DICOM or some voxel format. Obviously, these are very different from the Imagenet training data, but they are certainly related, so using the weights of a pre-trained model which has seen the Imagenet data should be a feasible starting point for the optimization.

In practice, if your new dataset is not that large but similar to Imagenet data and you're trying to tackle a classification task, you could use a pretrained model, switch the last layer to one with a number of neurons corresponding to the number of classes, then train the last layer to fine-tune them to your dataset, given that the low-level features of the pretrained model are transferable to the new dataset. Otherwise, if data is abundant but not that similar to the Imagenet data, you would retrain the entire network with the new dataset and make use of pretrained weights as good

starting points for your optimization.

7.7 Examples

Translational invariance isn't perfect in CNNs. Let's check that statement. "Pose" meaning position, orientation, lighting etc. relative to viewer is not built in, needs to be learned from data, hence a lot of data is necessary.

First, let's load the first 4 samples in the MNIST dataset:

```
import numpy as np

X = np.load("/data/X.npy").astype(float)[:4].reshape(4,28,28,1)
```

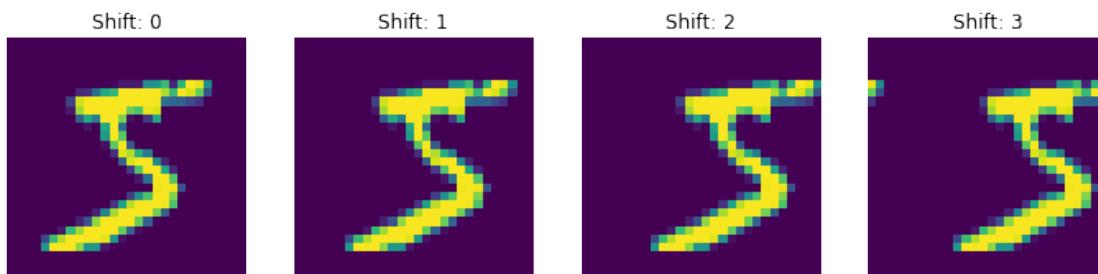
And shift one of the images to the right by one pixel four times:

```
import matplotlib.pyplot as plt

fig,axs = plt.subplots(1,4, figsize=(12,4))

X_rolled = np.array([np.roll(X[0], shift=2*i, axis=1) for i in range(4)])

for i in range(4):
    axs[i].imshow(X_rolled[i])
    axs[i].set_title(f"Shift: {i}")
    axs[i].axis(False)
```



Below, a convolutional layer is initialized. Instead of adding it to a sequential model like in last lecture's assignment, we can also just give it a name. It becomes a callable function now:

```
from tensorflow.keras.layers import Conv2D, MaxPooling2D

# first value doesn't matter, second two
# values are the image dimensions, the last
# value is the number of channels
input_shape = (1, 28, 28, 1)
```

```
convlayer = Conv2D(
    filters=1,
    kernel_size=(3,3),
    strides=(1, 1),
    padding="valid",
    dilation_rate=(1, 1),
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    input_shape=input_shape
)
```

The code above shows a lot of the options available for tuning convolutional layers. We don't need most of them for now.

Check it on the four shifted images of the 5:

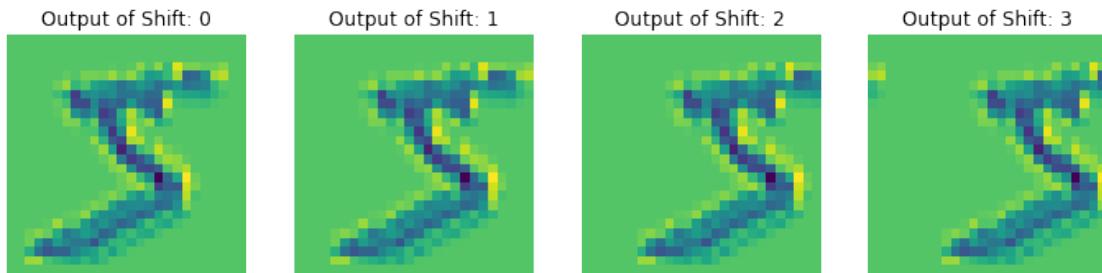
```
conv_out = convlayer(X_rolled)
print(conv_out.shape)
```

(4, 26, 26, 1)

And plotting the results:

```
fig,axs = plt.subplots(1,4, figsize=(12,4))

for i in range(4):
    axs[i].imshow(conv_out[i,:,:,:])
    axs[i].set_title(f"Output of Shift: {i}")
    axs[i].axis(False)
```



Nothing surprising is happening here. The translational invariance should come from the pooling layers. Let's try a simple one here:

```
maxpool = MaxPooling2D(
    pool_size=(3, 3),
    strides=(3,3),
    padding="valid"
)

pool_out = maxpool(conv_out)

print(pool_out.shape)
```

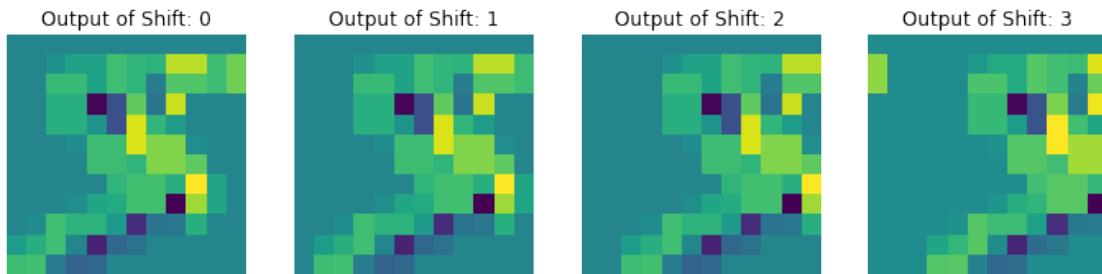
(4, 12, 12, 1)

The pool size is the size of the “kernel”, that performs the maxpooling operation, and the stride means just the same as in the convolutional layer case.

Plotting the results:

```
fig,axs = plt.subplots(1,4, figsize=(12,4))

for i in range(4):
    axs[i].imshow(pool_out[i,:,:,:])
    axs[i].set_title(f"Output of Shift: {i}")
    axs[i].axis(False)
```



Two of the images should look rather similar in the positions of the most notable and most neglectable features. But let's introduce two more layers:

```
convlayer2 = Conv2D(
    filters=1,
    kernel_size=(3,3),
    strides=(1, 1),
    padding="valid",
    dilation_rate=(1, 1),
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
```

```
    input_shape=input_shape
)

maxpool2 = MaxPooling2D(
    pool_size=(2, 2), strides=(2,2), padding="valid"
)
```

And test them on the results from before:

```
output = maxpool2(convlayer2(pool_out))

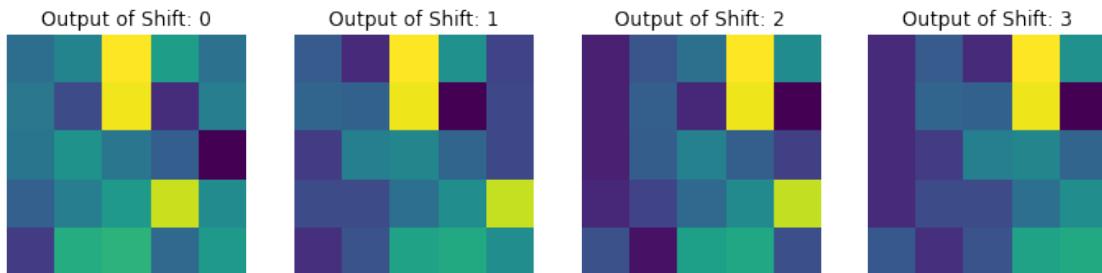
print(output.shape)
```

(4, 5, 5, 1)

Plot them:

```
fig,axs = plt.subplots(1,4, figsize=(12,4))

for i in range(4):
    axs[i].imshow(output[i,:,:,:])
    axs[i].set_title(f"Output of Shift: {i}")
    axs[i].axis(False)
```



Again, at least two images should look quite similar, although not exactly the same. This is the *limited* translational invariance of CNNs. It's only approximate, but the most important information is at the same position after small shifts. Experiment with the pool sizes and kernel strides and dilations here to get a feeling for what they influence.

CNNs have problems relating different “poses” of an object to each other. A shift in “pose” in this case might mean placing an object at a different position, rotating it, or putting it into different lighting conditions or any other transformation that doesn’t change the object itself. The advantage of this behavior is that we can use *data augmentation* to generate more training data for CNNs by simple translating, mirroring and rotating training images, since the CNN won’t recognize them as the same objects. The detriment is that these symmetries have to be *learned* by the network and hence, much more data is necessary to teach it.

We'll explore data augmentation in the next lecture.

As a last point, let's see what pooling does to some input image. We can use the `pillow` library for opening an image and convert it into a `numpy` array:

```
from PIL import Image
from numpy import asarray

image = Image.open('img/pepsi_dog.jpg')

print("Image type:", image.mode)

data = asarray(image)

print("Image shape", data.shape)

image
```

```
Image type: RGB
Image shape (600, 800, 3)
```



The pooling layer to play around with:

```
mp = MaxPooling2D(
    pool_size=(4, 4), strides=(4,4), padding="valid"
)
```

And now we can apply it to the data numpy array containing the image information. The layer actually returns a TensorFlow Tensor object, the `.numpy()` method converts this into a numpy array again:

```
pooled_img = mp(data.reshape(1,600,800,3)).numpy()

print("Pooled image dimensions:", pooled_img.shape)
```

Pooled image dimensions: (1, 150, 200, 3)

Now we can transform this back into a pillow image to see the full thing and plot the result for each channel:

```
pooled = Image.fromarray(pooled_img[0])

fig,axs = plt.subplots(1,4, figsize=(12,4))

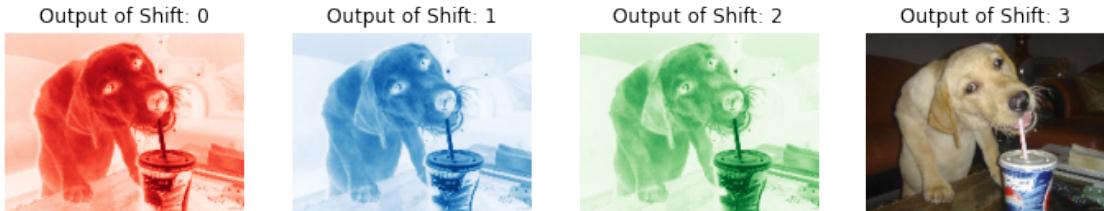
cmaps = ["Reds", "Blues", "Greens", None]

for i in range(4):
    axs[i].imshow(pooled_img[0,:,:,:i] if i < 3 else pooled_img[0],  

    ↪cmap=cmaps[i])
    axs[i].set_title(f"Output of Shift: {i}")
    axs[i].axis(False)

pooled
```





```
plt.figure(figsize=(12,12))
```

```
plt.imshow(pooled)  
plt.axis("off")
```

```
(-0.5, 199.5, 149.5, -0.5)
```



With a pool size of $(2, 2)$ not a whole lot changes, except for making the image smaller, but play around with the values yourself to see what happens in other configurations.

7.8 Activation Functions

7.8.1 Sigmoid

- Output range of $\sigma(z) \in [0, 1]$
- Saturation is easily possible
- In saturation, the gradient quickly vanishes ($\sigma'(z) = \sigma(z)(1 - \sigma(z))$)
- Outputs are always positive, so updates of all weights are in the same direction
- Acts linearly around the origin, but is not the identity function

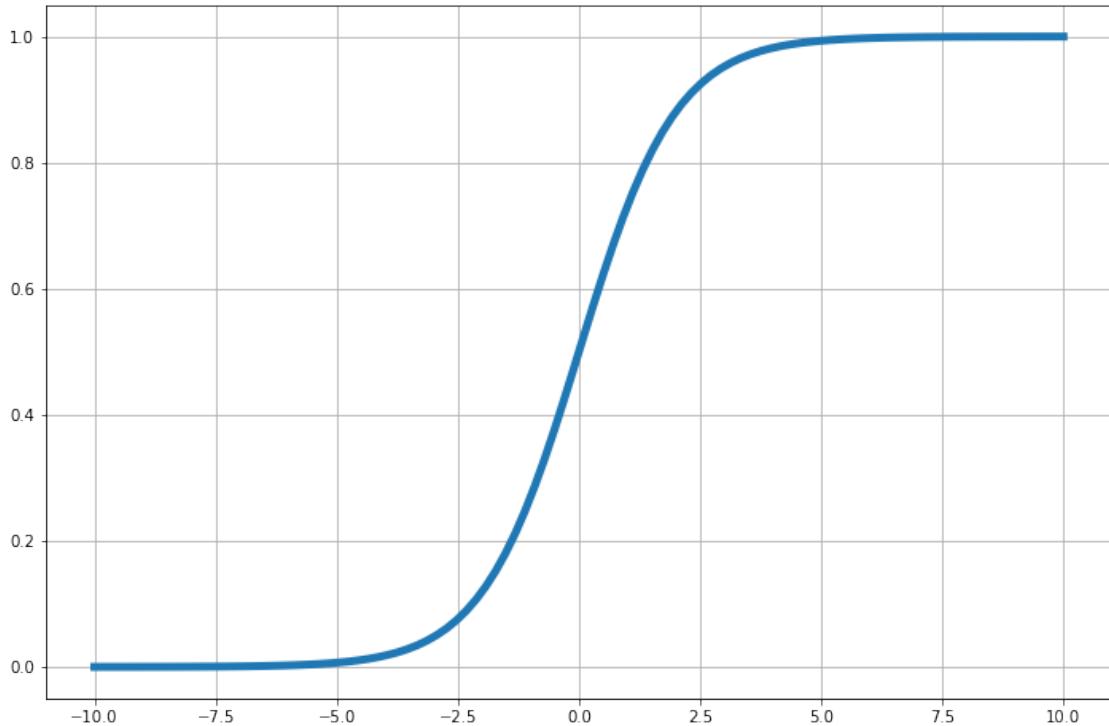
```
import numpy as np
import matplotlib.pyplot as plt

def sigmoid(z):
    return 1/(1+np.exp(-z))

x = np.linspace(-10, 10, 100)

fig = plt.figure(figsize=(12,8))
plt.grid()
plt.plot(x, sigmoid(x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e6ec0b00>]



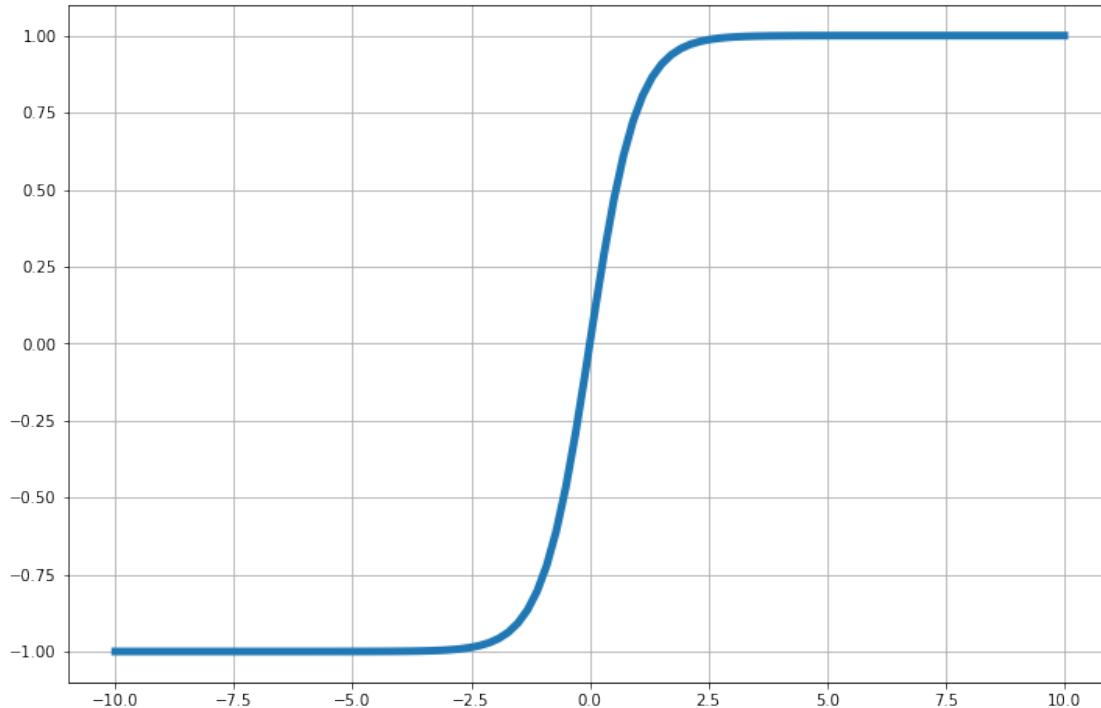
7.8.2 tanh

- Output range of $\tanh(z) \in [-1, 1]$
- Saturation problem like with sigmoid
- Acts like identity near the origin ($\tanh(z) \approx z$)
- Used in RNNs for their finite output range (taking powers of tanh won't make values explode) and when applied repeatedly, they do not become the zero function too quickly.

```
fig = plt.figure(figsize=(12,8))

plt.grid()
plt.plot(x, np.tanh(x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e6e3a828>]



7.8.3 ReLU

- Rectified Linear Unit $f(x) = \max(0, x)$
- No saturation problem for large inputs
- “Neuron death” (neuron weights receiving no updates) rare
- Must be initialized with positive bias, so that a gradient exists at all

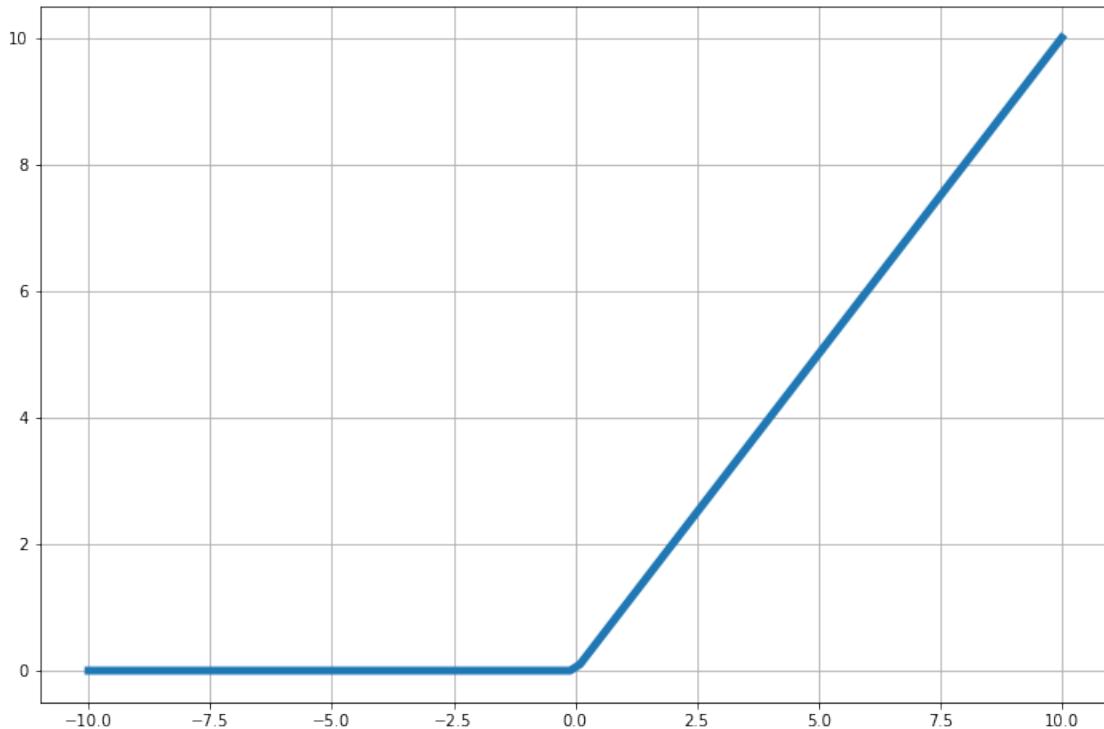
```
def relu(z):
    r = z.copy()
    idxs = r < 0
```

```
r[ixs] = 0

return r

fig = plt.figure(figsize=(12,8))
plt.grid()
plt.plot(x, relu(x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e6d55eb8>]



7.8.4 Leaky ReLU/Parametric Rectifier

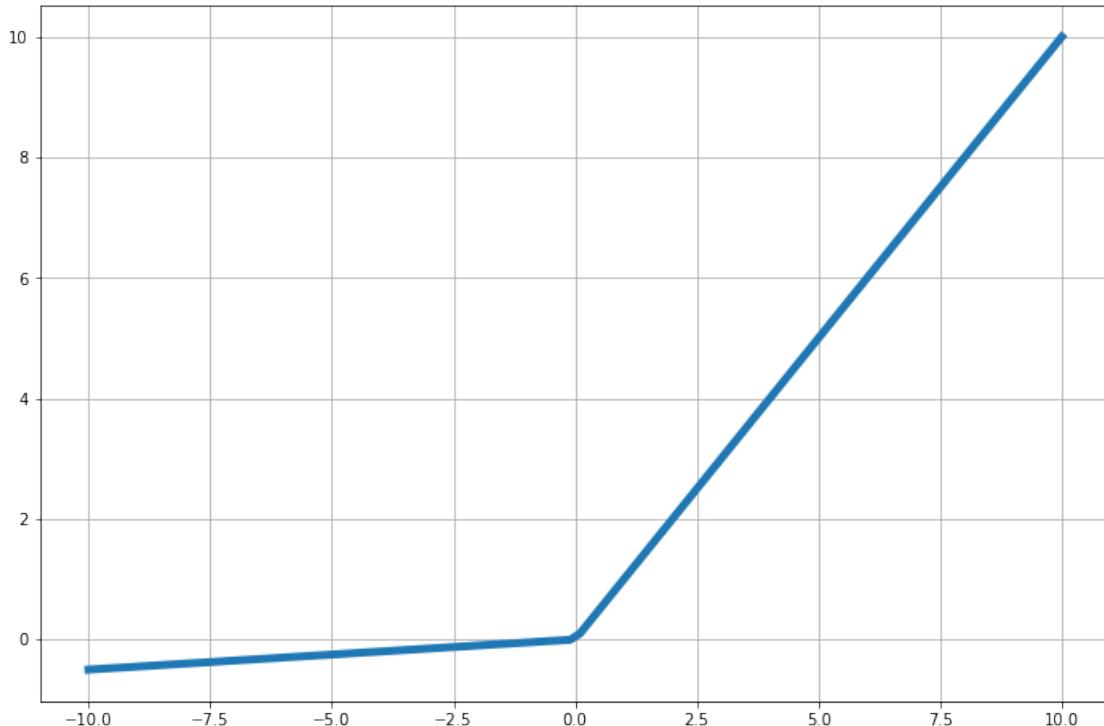
- $f(x) = \begin{cases} 0.01x & x \leq 0 \\ x & x > 0 \end{cases}$ or as a PReLU $f(x) = \begin{cases} \alpha x & x \leq 0 \\ x & x > 0 \end{cases}$
- No saturation problem for large inputs
- “Leakiness” has to be determined empirically, or via hyperparameter optimization

```
def prelu( , z):
    r = z.copy()
    ixs = r < 0
    r[ixs] = *r[ixs]

    return r
```

```
fig = plt.figure(figsize=(12,8))
plt.grid()
plt.plot(x, prelu(0.05, x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e6d29860>]



7.8.5 ELU

- Exponential Linear Unit

$$f(x) = \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$$

- Saturates for large negative values
- Otherwise same properties as ReLU

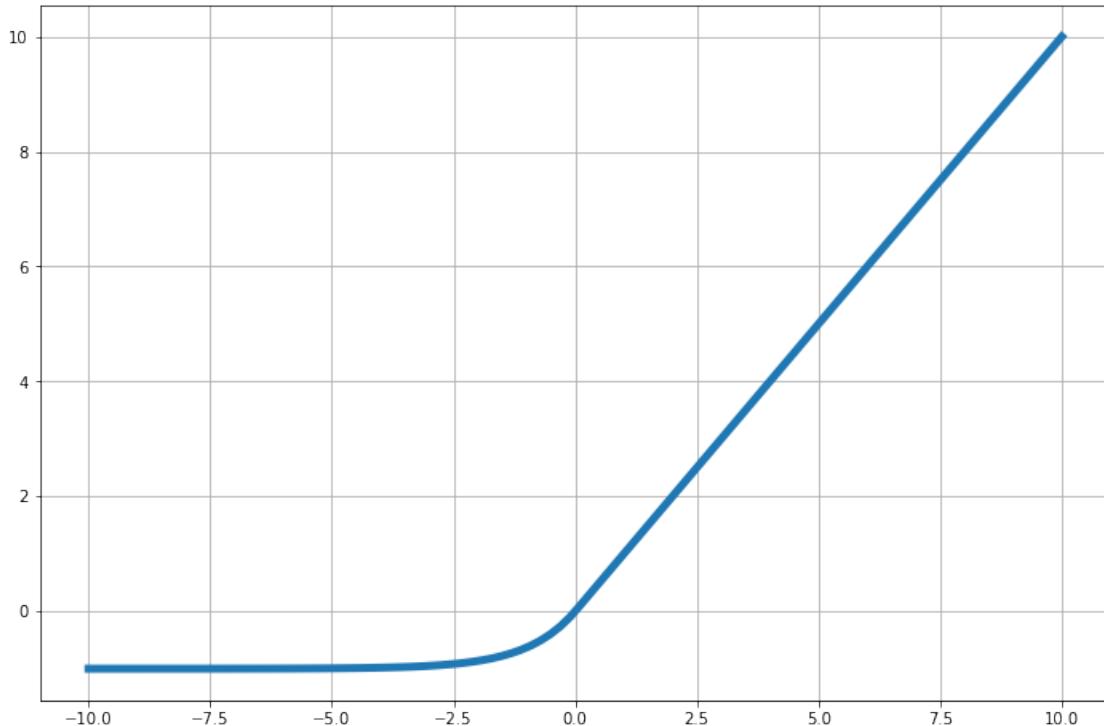
```
def elu( , z):
    r = z.copy()
    ixs = r < 0
    r[ixs] = *(np.exp(r[ixs])-1)

    return r

fig = plt.figure(figsize=(12,8))
```

```
plt.grid()
plt.plot(x, elu(1, x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e70437f0>]



7.8.6 SELU

- Scaled Exponential Linear Unit Klambauer 2017
- Modification of ELU

$$f(x) = \lambda \begin{cases} \alpha(e^x - 1) & x \leq 0 \\ x & x > 0 \end{cases}$$

- Optimized to create a gradient field that automatically keeps weights normalized with $\lambda \approx 1.0507$, $\alpha \approx 1.6733$
- For this to work, needs normalized inputs (zero mean, unit variance)
 - initialized weights in each layer with variance $1/n$
 - regularization with α -dropout
- Can have unintended side-effects like mode collapse in some situations, but works extraordinarily well especially for very deep networks
- Not smooth at the origin, but continuous

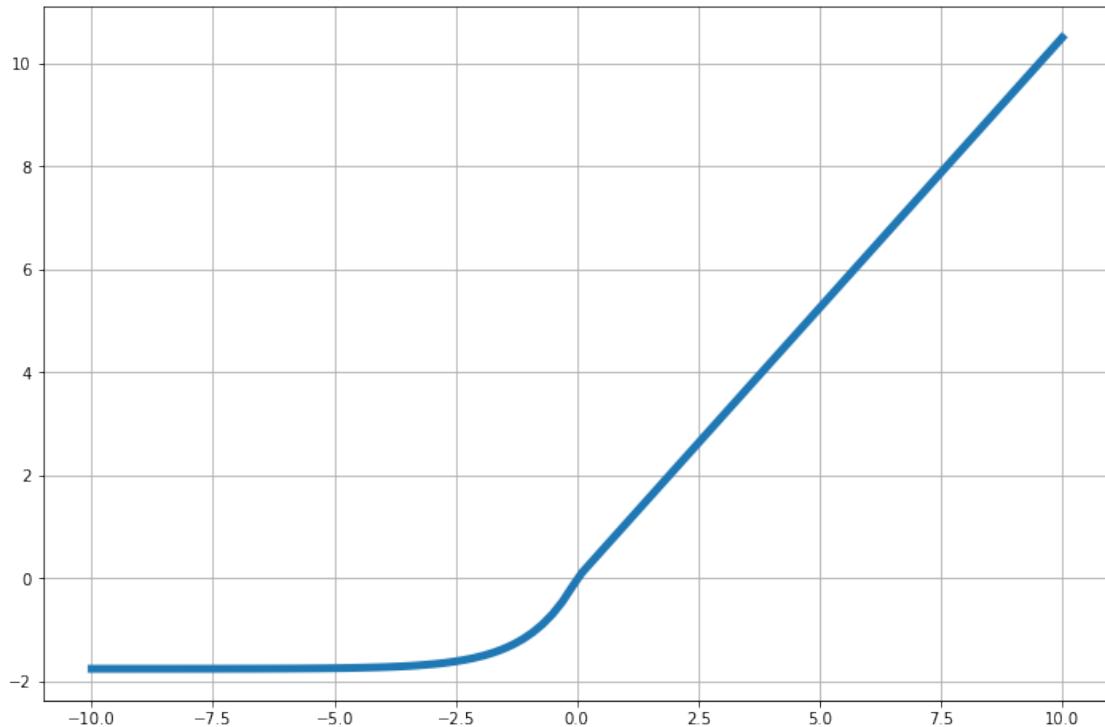
```
def selu( , , z):
    r = z.copy()
    ixs = r < 0
```

```
r[ixs] = *(np.exp(r[ixs])-1)

return *r

fig = plt.figure(figsize=(12,8))
plt.grid()
plt.plot(x, selu(1.0507, 1.6733, x), lw=5)
```

[<matplotlib.lines.Line2D at 0x7f21e6e0a978>]



Note that the form of the activation itself is not that important for what happens in an ANN, although it does have some effects directly (e.g. softmax for classification problems). What matters more is what the *gradient* of an activation function looks like.

7.9 Data Normalization/Preprocessing

The usual way to preprocess data is **z-score normalization/feature scaling**. The idea is to normalize the training data to have zero mean and unit variance. The way to achieve this is to calculate the mean over the training set for each dimension/feature and subtract the corresponding mean from *all* training input features:

$$\mu_j = \frac{1}{f} \sum_{i=1}^f x_j^{(i)} \quad (106)$$

$$\hat{x}_j = x_j - \mu_j \quad \text{or} \quad (107)$$

$$\hat{\mathbf{x}} = \mathbf{x} - \boldsymbol{\mu} \quad (108)$$

The next step is to divide all features by the *standard deviation* over all the training data

$$\sigma_j^2 = \frac{1}{f} \sum_{i=1}^f (\mu_j - \hat{x}_j^{(i)})^2 \quad (109)$$

$$\hat{x}_j \rightarrow \frac{\hat{x}_j}{\sigma_j} \quad (110)$$

This is the most common normalization technique. You can see the effects in the code below.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import PatchCollection
from matplotlib.patches import Rectangle

fig, axs = plt.subplots(1, 3, figsize=(18,6))

r1 = 2.5*(2*np.random.random(30)-1)+2
r2 = np.random.random(30)+2
m1 = r1 - np.mean(r1)
m2 = r2 - np.mean(r2)
n1 = m1 / np.sqrt(np.var(m1))
n2 = m2 / np.sqrt(np.var(m2))

rect1 = Rectangle([np.mean(r1)-2*np.sqrt(np.var(r1)), np.mean(r2)-2*np.sqrt(np.
    ↵var(r2))], 4*np.sqrt(np.var(r1)), 4*np.sqrt(np.var(r2)))
pc1 = PatchCollection([rect1], facecolor="", edgecolor='darkorange', lw=3)
rect2 = Rectangle([np.mean(m1)-2*np.sqrt(np.var(m1)), np.mean(m2)-2*np.sqrt(np.
    ↵var(m2))], 4*np.sqrt(np.var(m1)), 4*np.sqrt(np.var(m2)))
pc2 = PatchCollection([rect2], facecolor="", edgecolor='darkorange', lw=3)
rect3 = Rectangle([np.mean(n1)-2*np.sqrt(np.var(n1)), np.mean(n2)-2*np.sqrt(np.
    ↵var(n2))], 4*np.sqrt(np.var(n1)), 4*np.sqrt(np.var(n2)))
pc3 = PatchCollection([rect3], facecolor="", edgecolor='darkorange', lw=3)

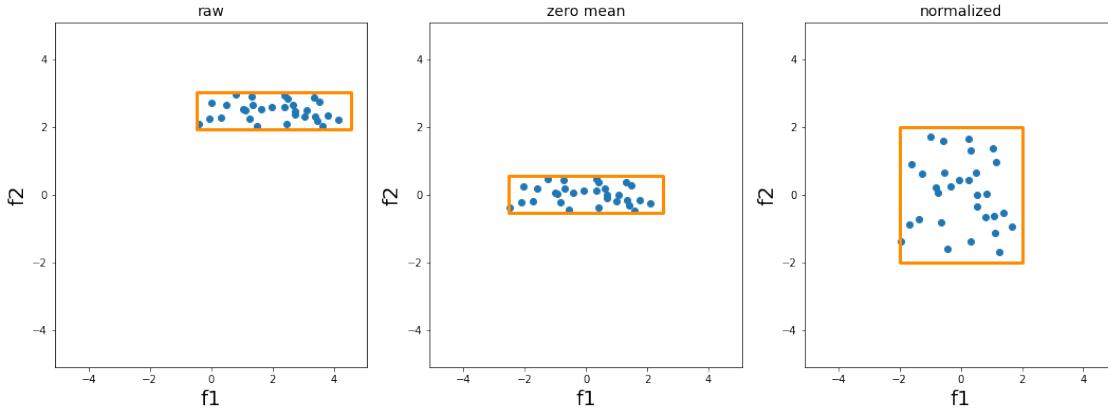
axs[0].scatter(r1, r2)
axs[0].add_collection(pc1)
axs[0].set_title("raw", fontsize=14)
axs[1].scatter(m1, m2)
```

```

    axs[1].add_collection(pc2)
    axs[1].set_title("zero mean", fontsize=14)
    axs[2].scatter(n1, n2)
    axs[2].add_collection(pc3)
    axs[2].set_title("normalized", fontsize=14)

    for ax in axs:
        ax.set_xlabel("f1", fontsize=20)
        ax.set_ylabel("f2", fontsize=20)
        ax.set_xlim([-5.1, 5.1])
        ax.set_ylim([-5.1, 5.1])

```



This is just a basic example with 2 features and mediocre *distortion*. The difference in values could well be several orders of magnitude (say, when having temperatures and high loads as inputs at the same time). Compare this to distorted elements in FEM, where a strong disbalance between the width and height of an element causes the Jacobian to have a bad condition number, resulting in various numerical problems. In general, the *condition number* says something about the ratio of column vector lengths of a matrix. The more they deviate from one another, the worse numerical problems become and the worse algorithms will work, if they work at all.

Recall that the input to an activation function is

$$z = 1/f \sum w_i x_i = w_1 x_1 + w_2 x_2 + \dots + w_f x_f$$

with f the number of features in the input vector \mathbf{x} . Without normalization, some of these terms would blow up or vanish, completely overshadowing all other values in this sum. It is very difficult to find an appropriate weight initialization for such a situation or for gradient-based optimization to find such a combination during training. Hence, normalization always makes sense.

Other techniques include *PCA-whitening*, which also *decorrelates* the data (we'll talk about PCA in a later lecture), or scaling to $[0, 1]$ instead of $[-1, 1]$. All these techniques are meant for the input layer alone, but they can also be applied to later layers. This is usually done in the form

of *batch normalization*, which will be the last lesson of today's lecture. This effectively prevents weights from becoming too large and saturating activation functions.

7.10 Batch Normalization

If nothing is done to prevent this, the distributions of each layer's weights changes during training and can lead to saturation issues, depending on the activation used. One solution possibility is an *internal covariate shift*, where each activation input is normalized just like training data was normalized above:

$$\tilde{x} \rightarrow \frac{\tilde{x} - \mu}{\sqrt{\text{var}[\tilde{x}]}} \quad (111)$$

where \tilde{x} means the output from the previous layer. In practice, this is implemented by **batch normalization**. The algorithm works as follows, for simplicity with a single layer with N neurons and some *mini batch* (a small set of training data) of size M . The inputs to the algorithm are the outputs of a layer \mathbf{x} over a mini batch $\mathfrak{B} = \mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(M)}$ and it outputs batch-normalized values $\tilde{y}_i = \text{BN}_{\beta, \gamma}(\mathbf{x}_i)$:

$$\mu_{\mathfrak{B}} = \frac{1}{M} \sum_{i=1}^M \tilde{x}_i \quad \text{mini batch mean} \quad (112)$$

$$\sigma_{\mathfrak{B}}^2 = \frac{1}{M} \sum_{i=1}^M (\tilde{x}_i - \mu_{\mathfrak{B}})^2 \quad \text{mini batch variance} \quad (113)$$

$$\tilde{y}_i = \gamma \frac{\tilde{x}_i - \mu_{\mathfrak{B}}}{\sqrt{\sigma_{\mathfrak{B}}^2 + \varepsilon}} + \beta \quad \text{normalize, then scale and shift} \quad (114)$$

(115)

β and γ are parameters to be learned, estimated by backpropagation, ε is a safeguard to make sure no bad things happen for extremely small variances. This whole process is done for every neuron and looking from above the network, looks like a linear layer added inbetween two normal hidden layers in an ANN. This is what it is usually treated like; an added inbetween layer that performs batch normalization. This data transformation makes sure that the weight distributions in layers don't change too drastically during training. Ideally, one would compute the mean and standard deviation for the whole training set, but they are usually extremely large and applying them would be rather inefficient. Instead, a *moving average* and *standard deviation* is calculated of mini batches of the training data during the training process (for each neuron in a layer). This can also be used in CNNs, but it's much more difficult to implement there.

The advantages of using batch normalization include increased learning speed, less need for dropout (later) or other forms of regularization, improved stability during training (fewer saturation effects), supposedly a higher accuracy after training (according to [the authors 2015](#), although I am not so sure this is universally true. These advantages necessitate using more computational resources, since more computation is needed for one extra layer in between each two layers in a model. The batch size also shouldn't be too small, otherwise no effect would be noticeable.

7.11 Weight Initialization

In assignment 02 we initialized the weights of our simple ANN completely randomly from a uniform distribution. Is this the best approach? Recall lesson 03-4 Transfer Learning, where we talked briefly about how choosing a good initialization point makes or breaks a good optimization.

Although ML frameworks usually take care of a good choice of weight initializations, it makes sense to at least give it a superficial look to understand how weight initialization and activation functions interact, since sometimes we may want to use custom activations.

Let's just see what happens in a hypothetical network with 50 layers, if we initialize the weights in a standard distribution with zero mean and unit variance. For the start, we'll see what happens with linear activations (so no activation at all). The inputs are multiplied with the first weight matrix to produce the first output, then that output is multiplied by the weight matrix of the next layer and so on.

```
import numpy as np

# randn produces random numbers sampled from a standard distribution
output = np.random.randn(50)

for i in range(50):
    # random weights sampled from standard distribution in each layer
    W = np.random.randn(50,50)
    # emulates the weight multiplication in each layer
    output = W @ output

print(output)
```

```
[-5.65571144e+40 -3.73813777e+41  3.68608416e+40 -9.88161125e+41
 -3.26504833e+41  3.13601953e+41 -5.02849474e+41  4.19997898e+41
 -1.04156456e+42  3.00314992e+41 -7.07353373e+41 -2.45616510e+41
 -1.67900158e+41  8.75893316e+41 -7.54062800e+40 -3.40924808e+41
 -1.87427205e+41  3.02362391e+41 -7.51396411e+41 -1.56717459e+41
 5.19544202e+41 -5.04505512e+41  6.94236844e+41 -2.09463334e+41
 2.86241658e+40 -6.71311295e+41  2.35629649e+41 -7.24607607e+41
 1.82399484e+41 -5.62767008e+40  8.17269412e+41 -5.12893484e+40
 -8.24660734e+41  2.64620272e+41 -2.41864969e+41 -3.63612377e+41
 -2.02809715e+41  3.74868869e+41 -7.30748435e+41 -6.93436048e+41
 -4.83193664e+41 -7.70479903e+41 -2.00449012e+41 -8.29552192e+41
 -4.45113843e+41 -3.35873614e+41 -1.31792274e+41  1.59709276e+42
 2.20476441e+41  1.86884401e+41]
```

The outputs exploded. It's extremely difficult if not impossible to update weights this large. What happens if we sample the initial random weights with a much smaller variance?

```
output = np.random.randn(50)

for i in range(50):
```

```

W = np.random.randn(50,50) * 0.001
output = W @ output

print(output)

```

```

[-4.05044213e-108  9.40943643e-109 -2.21013800e-108 -5.27481650e-109
 1.65720835e-108 -1.31924522e-108  2.54339792e-109  2.69487933e-108
 1.19378008e-109 -4.09343365e-108  1.08883033e-108 -2.84128986e-108
-3.33092334e-109 -2.01466660e-108  3.18875808e-108  2.65982320e-108
-1.66829744e-108  2.26644665e-108  1.45067300e-108  2.91378882e-108
 1.61469540e-108  4.01548613e-108 -4.54100585e-109 -8.01311961e-110
-2.36158449e-108 -9.60153998e-109 -1.64603223e-108  3.08724227e-108
-2.75160864e-108  4.84217702e-109  1.73722056e-108  3.51378848e-108
 7.58001644e-109  1.86352566e-108 -1.81574827e-108  3.46483893e-108
 2.55235018e-109  1.78215985e-108  2.03789119e-108  9.48870175e-109
-4.46901533e-108 -1.91040700e-108  2.57914524e-108 -7.48540059e-109
-3.08663951e-108 -1.94855157e-109 -3.12426986e-108 -1.82448701e-108
 2.83789703e-108 -2.57799783e-109]

```

The weights almost vanish. With more layers or even smaller weights, they will become zero and hence, produce zero output. This is no surprising results if we take a look at the mean and variance of the outputs in each layer and recall the computation rules for variance. The input of a layer is multiplied with each row of the weight matrix, so in this case this happens 50 times. Since each value is sampled from a standard distribution with zero mean and variance one, the output will still have zero mean, but variance 50. In the second layer, the variance becomes 2500 (50 times values with variance 50) and so on. The way to actually initialize the weights must be somewhere between these extrema. A good output should have a variance of one again.

A simple way to do so would be dividing the weights in each layer by the number of neurons:

```

output = np.random.randn(50)

for i in range(50):
    W = np.random.randn(50,50) / np.sqrt(50)
    output = W @ output

print(output)
print(np.mean(output))
print(np.var(output))

```

```

[ 2.1069863 -1.50147835  0.77843366 -1.0414192 -1.84186858 -0.61380669
-1.49760164  0.40935345  0.80535745  1.73008417  0.55557758 -2.11510371
-0.75060807 -0.48503801 -1.17388712 -2.06108467  0.32874392 -1.28268594
 1.021169    1.12247987  0.33104009 -0.32985982  0.47648031 -0.38193451
 1.17131599 -0.82415255  0.2009456   -0.93001206 -1.1539732   0.0699543
 0.65530778 -0.52381509  0.79573071 -0.05310976  1.18177383  1.36215959
 0.35554997 -1.57830173  1.40581872  0.83842203  0.01113606 -0.46696634
-0.40102354 -0.76489099  1.04240405 -0.85446014 -0.91466858 -0.8255044

```

```
0.77027958 -1.46598151]
-0.1261346439991442
1.0897539463619836
```

The outputs neither exploded nor vanished. This seems sufficient for linear activations, but what about the nonlinear activation functions we got to know? Let's see what happens with the anti-symmetric ones:

```
output = np.random.randn(50)

for i in range(50):
    W = np.random.randn(50,50) / np.sqrt(50)
    output = np.tanh(W @ output)

print(np.mean(output))
print(np.var(output))
```

```
-0.009713068175383581
0.00868893484997242
```

The weights are not vanishing, but they are very small. The standard way to initialize weights for antisymmetric activations is to use **Xavier** or **Glorot** initialization (named after Xavier Glorot), which samples the weights $w_{ij}^{(l)}$ from a *uniform* distribution with the bounds

$$w_{ij}^{(l)} \in [-\sqrt{\frac{6}{n_i + n_{i+1}}}, +\sqrt{\frac{6}{n_i + n_{i+1}}}]$$

where n_i is the number of connections from the previous layer, and n_{i+1} is the number of connections to the next layer (sometimes called “fanin” and “fanout”, inspired by logic gates). Let's see what happens in our ANN mockup:

```
def xavier(n, m):
    # sample uniformly from -1 to 1, divide by Xavier factor
    return (2*np.random.random((n,m))-1) / np.sqrt(6/(n+m))

output = np.random.randn(50)

for i in range(50):
    W = xavier(50,50)
    output = np.tanh(W @ output)

print(np.mean(output))
print(np.var(output))
```

```
0.18956333315247406
0.9448106794090045
```

We're back to being pretty close to zero mean and unit variance again.

What about ReLU et al? Let's see what the mean and variance per layer are:

```
def relu(x):
    z = x.copy()
    z[z<0] = 0
    return z

output = np.random.randn(50)

for i in range(1):
    W = np.random.randn(50,50)
    output = relu(W @ output)

print(np.mean(output))
print(np.var(output))
```

3.4797507804436827

15.685935255463573

On average, the variance should be something like $\sqrt{\frac{2}{n_i}}$, giving rise to the **Kaiming** or **He** initialization:

$$w_{ij}^{(l)} \in \mathcal{N}(0, 1) \cdot \sqrt{\frac{2}{n_i}}$$

The weights are sampled from a standard distribution and multiplied by $\sqrt{2/n_i}$. Let's see what happens:

```
def kaiming(n, m):
    return np.random.randn(n,m) * np.sqrt(2 / n)

output = np.random.randn(50)

for i in range(50):
    W = kaiming(50,50)
    output = relu(W @ output)

print(np.mean(output))
print(np.var(output))
```

0.5651593272797193

0.6580111556910514

On average, this should produce sane outputs. See for yourself what happens when using Xavier initialization in this case.

To see how initializations affect the weights in a small ANN, we can get the weight gradients in each epoch and plot their minimum and maximum:

```

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.backend import one_hot, clear_session

clear_session()

X_MNIST = np.load("/data/X.npy", allow_pickle=True)[:100]
y_MNIST = np.load("/data/y.npy", allow_pickle=True)[:100].astype(int)
y_oh = one_hot(y_MNIST, 10)

vals = ["mean", "min", "max"]
activations = ["linear", "tanh", "relu"]
initializations = ["uniform", "glorot_normal", "he_uniform"]
results = {}

epochs = 100

for activation in activations:
    results[activation] = {}
    for initialization in initializations:
        w_mean = []
        w_min = []
        w_max = []

        model = Sequential()
        model.add(Dense(400, input_shape=(X_MNIST.shape[1],), kernel_initializer=initialization, activation=activation))
        model.add(Dense(256, kernel_initializer=initialization, activation=activation))
        model.add(Dense(128, kernel_initializer=initialization, activation=activation))
        model.add(Dense(64, kernel_initializer=initialization, activation=activation))
        model.add(Dense(10, kernel_initializer=initialization, activation="softmax"))

        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

        for i in range(epochs):
            model.fit(X_MNIST, y_oh, epochs=1, batch_size=4, verbose=0)
            w_mean.append([np.mean(layer.get_weights()[0]) for layer in model.layers])
            w_min.append([np.min(layer.get_weights()[0]) for layer in model.layers])

```

```
w_max.append( [np.max( layer.get_weights()[0]) for layer in model.
→layers])
```

```
results[activation][initialization] = np.array([w_mean, w_min, w_max])
```

```
import matplotlib.pyplot as plt

fig,axs = plt.subplots(len(activations),len(initializations), figsize=(12,12),_
→sharex='col', sharey='row')

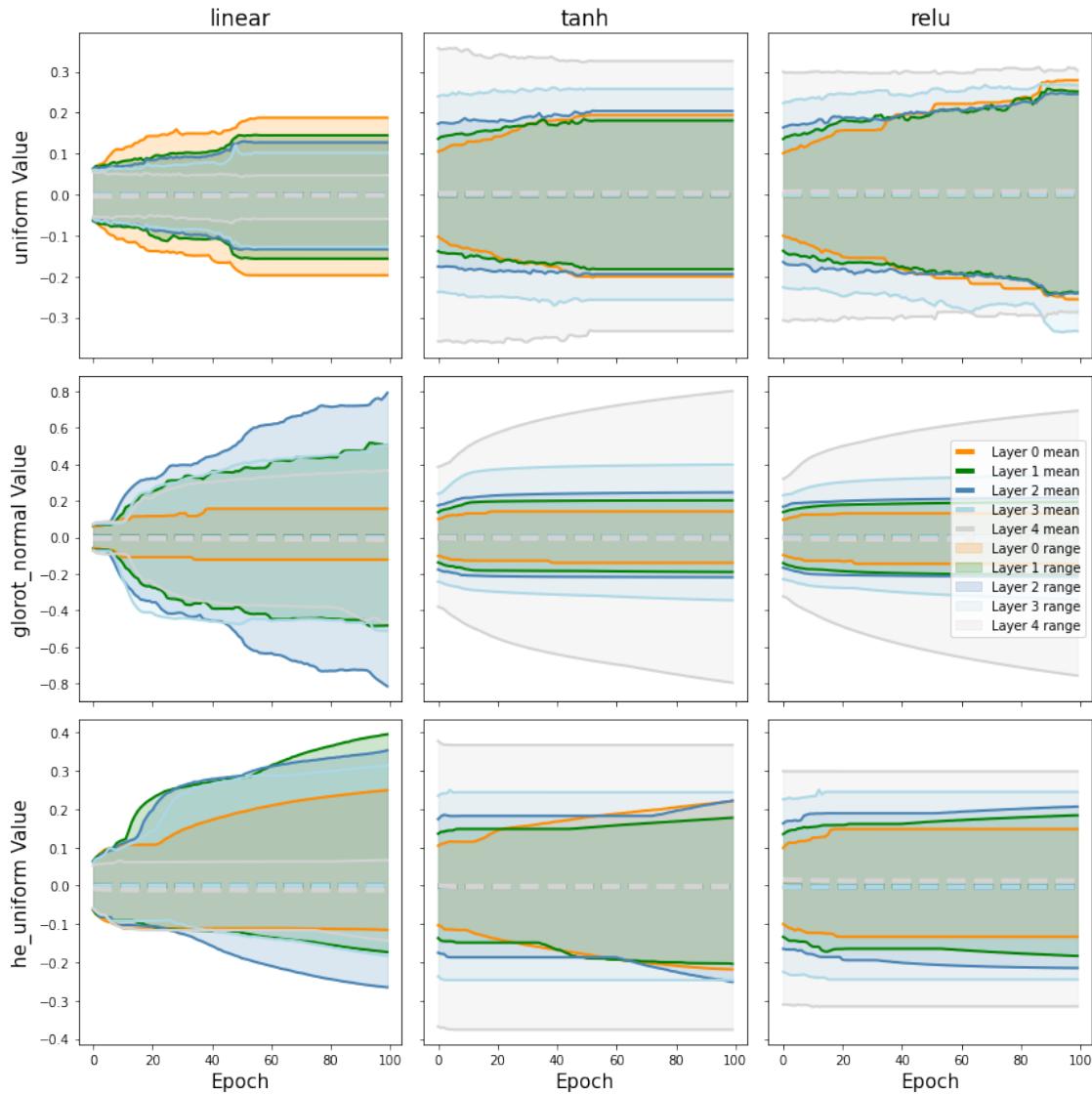
colors = ["darkorange", "green", "steelblue", "lightblue", "lightgray"]

for i,activation in enumerate(results.keys()):
    for j,initialization in enumerate(results[activation]):
        for l,layer in enumerate(np.
→transpose(results[activation][initialization], [2,1,0])):
            axs[i,j].fill_between(range(epochs), layer[:,1], layer[:,2],_
→color=colors[l], alpha=0.2, label=f"Layer {l} range")
            axs[i,j].plot(range(epochs), layer[:,0], lw=4, ls="--",_
→color=colors[l], label=f"Layer {l} mean")
            axs[i,j].plot(range(epochs), layer[:,1], lw=2, color=colors[l]),_
→label=f"Layer {l} min")
            axs[i,j].plot(range(epochs), layer[:,2], lw=2, color=colors[l]),_
→label=f"Layer {l} max")

for ax in axs[-1]:
    ax.set_xlabel("Epoch", fontsize=15)
for i,ax in enumerate(axs[:,0]):
    ax.set_ylabel(f"{initializations[i]} Values", fontsize=15)
for i,ax in enumerate(axs[0]):
    ax.set_title(activations[i], fontsize=17)

axs[1,-1].legend(loc="center right")

plt.tight_layout()
```



7.11.1 SELU

A special way of making sure weights retain their normalization is to use the **SELU activation**. According to the [original paper](#), it automatically keeps all weights in an ANN normalized if:

- input data is normalized
- weights are initialized with zero mean and variance $1/n_i$ (n_i is the number of incoming connections)
- AlphaDropout with a rate between 0.05 and 0.1

AlphaDropout is a slightly modified Dropout regularization technique, which we'll discuss in the next lecture. Using SELU as a normalization technique has the advantage that no mini-batch normalization layers are needed to keep the weights normalized. It just happens automatically. It does come with more robust training and better results in general, but may be more prone to mode

collapse. (*side note: the original paper is only 9 pages long, but has an infamously long appendix with 90 pages of proofs. There's even a [Twitter account](#) tracking mentions of it.*)

7.12 Hyperparameter Optimization

Hyperparameters of a model are the values that have to be chosen and fixed *before* starting the training process. Choosing a good set of hyperparameters is a difficult task and either requires heaps of experience or a good searching approach. There are various ways to optimize hyperparameters. The obvious one is **manual hyperparameter tuning** (sometimes called *grad student descent* (haha...)), meaning trial and error until a good set is found. Obviously, this is not a feasible approach for most problems, especially those of the larger variety.

As an example for all the techniques mentioned here, we'll use a simple ANN and the MNIST dataset from Lecture 02:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense
from tensorflow.keras.optimizers import SGD, Adam
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

X = np.load("/data/X.npy", allow_pickle=True)[:10000]
y = np.load("/data/y.npy", allow_pickle=True)[:10000].astype(int)

X_train = X[:6000]
y_train = y[:6000]
X_val = X[6000:8000]
y_val = y[6000:8000]
X_test = X[8000:]
y_test = y[8000:]
```

For the hyperparameter optimization, we need a function that abstracts the parameters we'd like to optimize as arguments to the function itself. The example below will allow varying the number of hidden layers, the neurons per layer (you can modify the function to allow for different numbers of neurons per layer), the activations, and the optimizer. The latter two are categorical variables, while the former are discrete but in principle infinite. Instead of the learning rate itself, the logarithm of the learning rate is provided here. The reason is that this way it's easier to find the order of magnitude it should be in. Testing 0.1, 0.001, 0.0001 and so on will give us a hint at where to continue the search later on more efficiently than testing 0.0001, 0.0002, 0.0003 and so on.

The function will construct a model in just the same way we already know. The first layer is a `Flatten()` layer, which takes something of some provided shape as input and *flattens* it, meaning converting it into a vector.

A slight specialty is perhaps the treatment of the optimizer here, where the argument is supposed to be a *class*. Python handles functions and classes just like variables (actually, everything is a *class* in Python), and calling the function below with something like `optimization=Adam` will simply alias the original function.

```

def build_classification_model(hidden_layers=1,
                               neurons_per_layer=10,
                               log_learning_rate=-3,
                               activation="relu",
                               optimization=Adam,
                               input_shape=(1,784)):

    # base model
    model = Sequential()

    # flatten the input
    model.add(Flatten(input_shape=input_shape))

    # add hidden layers with activations
    for layer in range(hidden_layers):
        model.add(Dense(neurons_per_layer, activation=activation))

    # add output layer with softmax activation for categorical data
    model.add(Dense(10, activation="softmax"))

    # initialize the optimizer
    optimizer = optimization(lr=10**log_learning_rate)

    # compile model with appropriate loss, chosen optimizer and some desired
    ↪metrics
    # the metric can be almost any function that takes certain arguments and
    ↪will
    # be evaluated in each step
    model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
    ↪metrics=["accuracy"])

    return model

# this wrapper lets us call the ANN like a scikit-learn classifier
# it would work without the wrapper but is more convenient this way
ann_clf = KerasClassifier(build_classification_model)

```

7.12.1 Cross-validation

We saw how splitting the data into training, validation, and test sets helps to evaluate a model in the end. During training, the validation set can provide some information about the bias-variance tradeoff, and whether overfitting happens. For hyperparameter optimization, this cross validation is usually done as a ***K*-Fold** cross validation, since using the same data as a validation set for each combination of hyperparameters may cause some trouble. Say for example the problem is a classification problem with 10 classes, and the validation set only contains data about 7 of these classes.

The idea of *K*-Folds is to split the data set used for training (test set aside) into *K* parts, then use

$K - 1$ of these subsets for training, and 1 for validation:

```

import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from matplotlib.colors import ListedColormap

K = 5

X = range(100)
y = np.zeros(100)
y[10:30] = 1
y[30:] = 2

plt.figure(figsize=(13,7))

plt.scatter(X, len(X)*[-1], c=y, marker='_', lw=30, cmap='Paired')

for i in range(K):
    folds = np.zeros(len(X))
    fold_size = len(X) // 5

    folds[i*fold_size:(i+1)*fold_size] = 1

    plt.scatter(X, len(X)*[i], c=folds, cmap=ListedColormap(["steelblue", ↴
    "darkorange"]), marker='_', lw=30)

plt.gca().set_yticklabels(["", "Data"] + [f"Fold {i+1}" for i in range(K)], ↴
    fontsize=14)
plt.gca().set_xticks([])
plt.xlim([-10, 140])
plt.ylim([-2, 5])

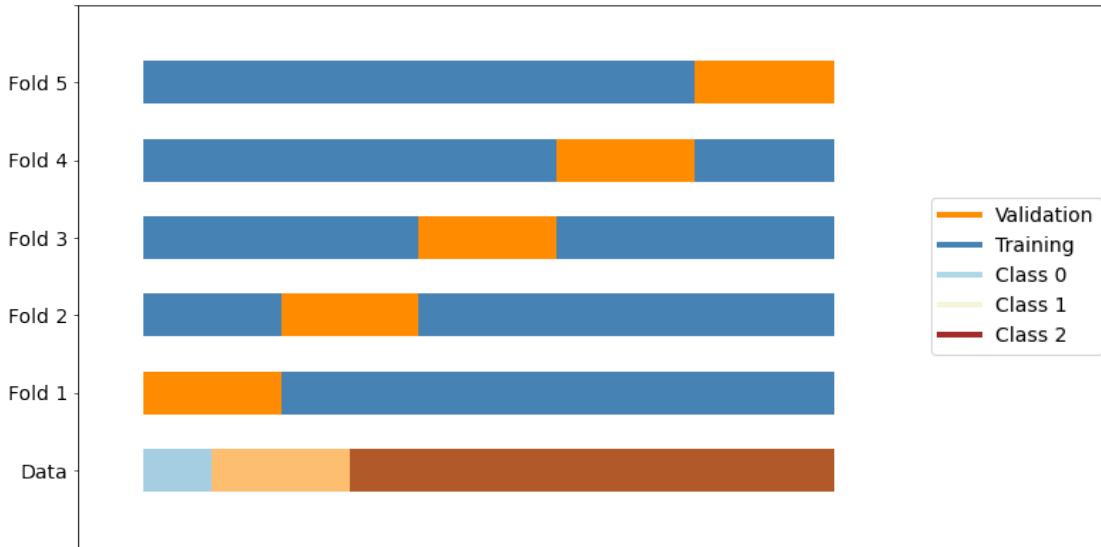
legend_colors = ["darkorange", "steelblue", "lightblue", "beige", "brown"]

legend_lines = [Line2D([0], [0], color=color, lw=4) for color in legend_colors]

plt.legend(legend_lines, ['Validation', 'Training']+['Class {i}' for i in ↴
    range(3)], fontsize=14, loc="center right")
plt.show()

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:25: UserWarning:
FixedFormatter should only be used together with FixedLocator



The orange parts mark which subset of the data is used for validation, the blue parts are used for training. This way, the actual distribution of classes among the data won't affect the hyperparameter optimization too much. In practice, more sophisticated folding methods like stratified folds are used, which don't create these subsets in deterministic and connected chunks, but randomly distributed and in smaller sets.

In the code examples below, the folds are controlled by the `cv` parameter.

7.12.2 Grid Search

A naive approach is to use **grid search**, which takes a set of hyperparameters, keeps all of them fixed except for one. This last hyperparameter is then varied in a predefined range and as soon as that *sweep* is done, one of the other hyperparameters is changed to start another sweep. The gridlike nature of this approach may hide minima in-between the grid points.

Let's see what this looks like for the small ANN:

```
from sklearn.model_selection import GridSearchCV

# dictionary that will tell the search algorithm which parameters to
# vary and which combinations to try
param_space = {"hidden_layers": [1, 2],
               "neurons_per_layer": [10, 20, 30],
               "log_learning_rate": [-3, -2, -1],
               "activation": ["relu", "tanh"]}
}

# instantiate a GridSearchCV object that can be used like a
# sciki-learn classifier
```

```

ann_grid_search = GridSearchCV(ann_clf,
                               param_space,
                               cv=5,
                               scoring="accuracy",
                               return_train_score=True,
                               verbose=True,
                               n_jobs=4)

# "fit" all the classifiers to the data
ann_grid_search.fit(X_train, y_train, epochs=3,
                     validation_data=(X_val, y_val))

```

Fitting 5 folds for each of 36 candidates, totalling 180 fits

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent workers.

[Parallel(n_jobs=2)]: Done 46 tasks | elapsed: 49.6s

[Parallel(n_jobs=2)]: Done 180 out of 180 | elapsed: 3.2min finished

Epoch 1/3

WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32, name='flatten_input'), name='flatten_input', description="created by layer 'flatten_input'"), but it was called on an input with incompatible shape (None, 784).

WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32, name='flatten_input'), name='flatten_input', description="created by layer 'flatten_input'"), but it was called on an input with incompatible shape (None, 784).

175/188 [=====>...] - ETA: 0s - loss: 2.1182 - accuracy: 0.3123

WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32, name='flatten_input'), name='flatten_input', description="created by layer 'flatten_input'"), but it was called on an input with incompatible shape (None, 784).

188/188 [=====] - 1s 4ms/step - loss: 2.0845 - accuracy: 0.3229 - val_loss: 1.1830 - val_accuracy: 0.6340

Epoch 2/3

188/188 [=====] - 0s 2ms/step - loss: 1.1125 - accuracy: 0.6561 - val_loss: 0.8561 - val_accuracy: 0.7300

Epoch 3/3

188/188 [=====] - 0s 2ms/step - loss: 0.8265 - accuracy: 0.7572 - val_loss: 0.7433 - val_accuracy: 0.7630

```

GridSearchCV(cv=5,
             estimator=<tensorflow.python.keras.wrappers.scikit_learn.KerasClassifier object
at 0x7fa1233bb240>,
             n_jobs=2,

```

```
param_grid={'activation': ['relu', 'tanh'],
            'hidden_layers': [1, 2],
            'log_learning_rate': [-3, -2, -1],
            'neurons_per_layer': [10, 20, 30]},
            return_train_score=True, scoring='accuracy', verbose=True)
```

This should take a short while, depending on the server load. We can get the resulting best combination of hyperparameters by:

```
print("Best parameters:", ann_grid_search.best_params_)

print("Best score: ", ann_grid_search.best_score_)
```

```
Best parameters: {'activation': 'tanh', 'hidden_layers': 1, 'log_learning_rate': -3, 'neurons_per_layer': 30}
Best score:  0.746
```

We can also directly use the model with the best parameters without manually setting them again, since it's already trained:

```
ann_grid_search.best_estimator_.predict(X_test)
```

```
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_input'), name='flatten_input', description="created by layer
'flatten_input'"), but it was called on an input with incompatible shape (None,
784).
```

```
/usr/local/lib/python3.7/dist-
packages/tensorflow/python/keras/engine/sequential.py:450: UserWarning:
`model.predict_classes()` is deprecated and will be removed after 2021-01-01.
Please use instead: * `np.argmax(model.predict(x), axis=-1)`, if your model
does multi-class classification (e.g. if it uses a `softmax` last-layer
activation). * `(model.predict(x) > 0.5).astype("int32")`, if your model does
binary classification (e.g. if it uses a `sigmoid` last-layer activation).
warnings.warn(`model.predict_classes()` is deprecated and '
```

```
array([0, 6, 2, ..., 6, 9, 7])
```

And evaluate the best model on the test set:

```
ann_grid_search.best_estimator_.score(X_test, y_test)
```

```
63/63 [=====] - 0s 1ms/step - loss: 0.7239 - accuracy:
0.7855
```

```
0.7854999899864197
```

There are different kinds of visualizations that can help to understand the results. A common

technique is a **parallel coordinates plot**. The idea is to plot lines for each combination of hyperparameters and introduce parallel axes denoting the quantity of a hyperparameter.

The `cv_results_` dictionary has a slightly weird form, so we need to adjust it a little bit such that `plotly` can create the plot. The first thing is to convert it into a `pandas` dataframe:

```
import pandas as pd

grid_scores = pd.DataFrame(ann_grid_search.cv_results_)

grid_scores.head()

mean_fit_time std_fit_time mean_score_time std_score_time \
0 1.826210 0.290822 0.129712 0.015833
1 1.618566 0.032389 0.114856 0.004062
2 1.612695 0.025639 0.116547 0.004402
3 1.561959 0.026647 0.114704 0.007238
4 1.617028 0.023619 0.113990 0.006094

param_activation param_hidden_layers param_log_learning_rate \
0 relu 1 -3
1 relu 1 -3
2 relu 1 -3
3 relu 1 -2
4 relu 1 -2

param_neurons_per_layer params \
0 10 {'activation': 'relu', 'hidden_layers': 1, 'lo...
1 20 {'activation': 'relu', 'hidden_layers': 1, 'lo...
2 30 {'activation': 'relu', 'hidden_layers': 1, 'lo...
3 10 {'activation': 'relu', 'hidden_layers': 1, 'lo...
4 20 {'activation': 'relu', 'hidden_layers': 1, 'lo...

split0_test_score ... mean_test_score std_test_score rank_test_score \
0 0.140000 ... 0.155667 0.041400 27
1 0.307500 ... 0.297500 0.042736 16
2 0.535833 ... 0.534500 0.059336 7
3 0.118333 ... 0.138500 0.030688 28
4 0.198333 ... 0.174333 0.072903 21

split0_train_score split1_train_score split2_train_score \
0 0.151458 0.238125 0.167292
1 0.311875 0.274583 0.370417
2 0.575000 0.575833 0.528750
3 0.106042 0.198958 0.177083
4 0.180833 0.114375 0.149583

split3_train_score split4_train_score mean_train_score std_train_score
```

```

0          0.122083      0.123958      0.160583      0.042343
1          0.281042      0.283750      0.304333      0.035426
2          0.639583      0.458958      0.555625      0.059832
3          0.112708      0.111250      0.141208      0.038906
4          0.107500      0.306875      0.171833      0.072460

[5 rows x 24 columns]

```

Next, `plotly` cannot handle categorical data, so we need to assign numbers instead of names to the activation functions:

```

grid_scores.param_activation[grid_scores.param_activation=="tanh"] = 0
grid_scores.param_activation[grid_scores.param_activation=="relu"] = 1

grid_scores.head()

```

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

```

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
 """Entry point for launching an IPython kernel.

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

```

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```

mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
0            1.826210      0.290822      0.129712      0.015833
1            1.618566      0.032389      0.114856      0.004062
2            1.612695      0.025639      0.116547      0.004402
3            1.561959      0.026647      0.114704      0.007238
4            1.617028      0.023619      0.113990      0.006094

param_activation param_hidden_layers param_log_learning_rate  \
0                  1                      1                      -3
1                  1                      1                      -3
2                  1                      1                      -3
3                  1                      1                      -2
4                  1                      1                      -2

param_neurons_per_layer                                params  \
0           10  {'activation': 'relu', 'hidden_layers': 1, 'lo...

```

```

1                      20  {'activation': 'relu', 'hidden_layers': 1, 'lo...
2                      30  {'activation': 'relu', 'hidden_layers': 1, 'lo...
3                      10  {'activation': 'relu', 'hidden_layers': 1, 'lo...
4                      20  {'activation': 'relu', 'hidden_layers': 1, 'lo...

    split0_test_score ... mean_test_score std_test_score rank_test_score \
0          0.140000   ...      0.155667     0.041400        27
1          0.307500   ...      0.297500     0.042736        16
2          0.535833   ...      0.534500     0.059336         7
3          0.118333   ...      0.138500     0.030688        28
4          0.198333   ...      0.174333     0.072903        21

    split0_train_score split1_train_score split2_train_score \
0          0.151458           0.238125      0.167292
1          0.311875           0.274583      0.370417
2          0.575000           0.575833      0.528750
3          0.106042           0.198958      0.177083
4          0.180833           0.114375      0.149583

    split3_train_score split4_train_score mean_train_score std_train_score
0          0.122083           0.123958      0.160583      0.042343
1          0.281042           0.283750      0.304333      0.035426
2          0.639583           0.458958      0.555625      0.059832
3          0.112708           0.111250      0.141208      0.038906
4          0.107500           0.306875      0.171833      0.072460

[5 rows x 24 columns]

```

And using plotly for the actual plot:

```

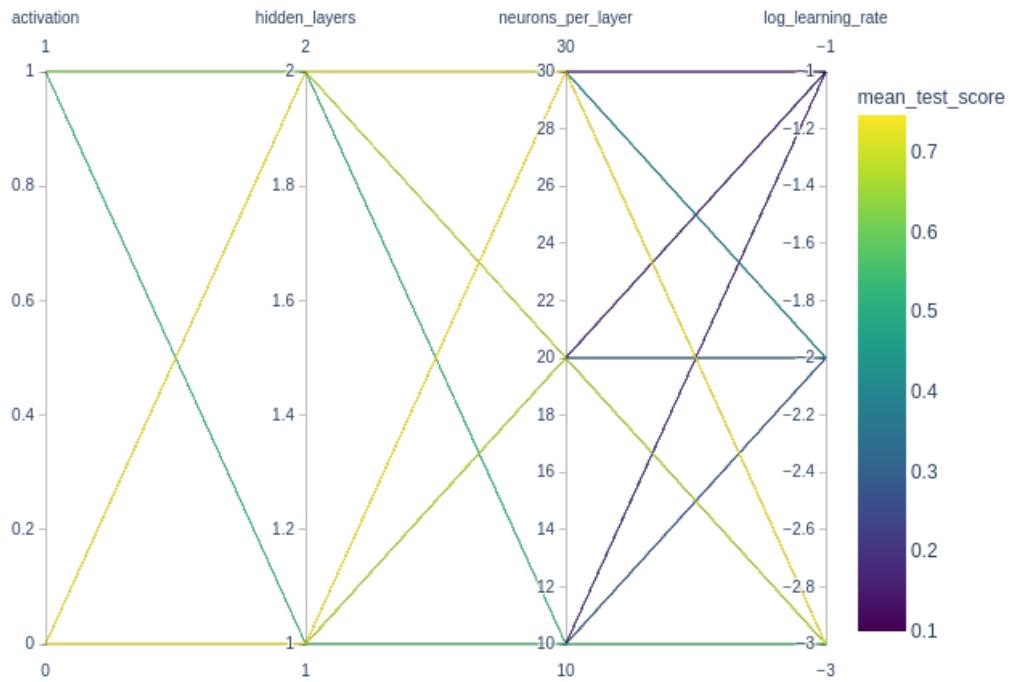
import plotly.express as px

dimensions=["param_activation",
            "param_hidden_layers",
            "param_neurons_per_layer",
            "param_log_learning_rate"]

fig = px.parallel_coordinates(grid_scores.to_dict(),
                               color="mean_test_score",
                               dimensions=dimensions,
                               labels={s: s[6:] for s in dimensions},
                               color_continuous_scale="viridis")#px.colors.
                               ↪diverging.Tealrose)

fig.show()
fig.write_image("img/plot_grid_par_coords.png")

```



The colorbar on the right denotes the mean loss score on the test set, so the lower this value is, the better. `plotly` creates interactive graphs. You can mark interesting areas in the graph above by creating a vertical line on some axis with the left mouse button. This will mark all lines that fall into that range. Of course the above graph is for a very small set of tested parameters and hence quite uninteresting. Generally, this can help in deciding where to continue the search.

7.12.3 Randomized Search

Instead of going over each parameter one-by-one, they can also be sampled randomly. This is what **Randomized Search** does. It's basically grid search, but only takes a small number `n_iter` of samples to probe the full hyperparameter space. Hence, we can increase the search space a little bit:

```
from sklearn.model_selection import RandomizedSearchCV

param_space = {"hidden_layers": [1, 2, 3],
               "neurons_per_layer": [5, 10, 20, 30, 50],
               "log_learning_rate": [-4, -3, -2, -1, 0],
               "activation": ["relu", "tanh", "linear"]}
}

ann_random_search = RandomizedSearchCV(ann_clf,
```

```

        param_space,
        cv=5,
        scoring="accuracy",
        return_train_score=True,
        verbose=True,
        n_iter=36,
        n_jobs=4)

ann_random_search.fit(X_train, y_train, epochs=3,
                      validation_data=(X_val, y_val))

```

Fitting 5 folds for each of 36 candidates, totalling 180 fits

```

[Parallel(n_jobs=2)]: Using backend LokyBackend with 2 concurrent workers.
[Parallel(n_jobs=2)]: Done  46 tasks      | elapsed:   49.0s
[Parallel(n_jobs=2)]: Done 180 out of 180 | elapsed:  3.3min finished

Epoch 1/3
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_1_input'), name='flatten_1_input', description="created by layer
'flatten_1_input'"), but it was called on an input with incompatible shape
(None, 784).
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_1_input'), name='flatten_1_input', description="created by layer
'flatten_1_input'"), but it was called on an input with incompatible shape
(None, 784).
162/188 [=====>...] - ETA: 0s - loss: 63.7850 - accuracy:
0.5493WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for
input KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_1_input'), name='flatten_1_input', description="created by layer
'flatten_1_input'"), but it was called on an input with incompatible shape
(None, 784).
188/188 [=====] - 1s 4ms/step - loss: 58.5119 -
accuracy: 0.5733 - val_loss: 9.1737 - val_accuracy: 0.8160
Epoch 2/3
188/188 [=====] - 0s 2ms/step - loss: 6.4504 -
accuracy: 0.8543 - val_loss: 5.7461 - val_accuracy: 0.8330
Epoch 3/3
188/188 [=====] - 0s 3ms/step - loss: 3.4429 -
accuracy: 0.8773 - val_loss: 4.2632 - val_accuracy: 0.8435

```

```

RandomizedSearchCV(cv=5,
estimator=<tensorflow.python.keras.wrappers.scikit_learn.KerasClassifier object
at 0x7fa1233bb240>,
n_iter=36, n_jobs=2,
param_distributions={'activation': ['relu', 'tanh'],

```

```

        'linear'],
    'hidden_layers': [1, 2, 3],
    'log_learning_rate': [-4, -3, -2, -1,
                           0],
    'neurons_per_layer': [5, 10, 20, 30,
                           50]},
    return_train_score=True, scoring='accuracy', verbose=True)

```

The results can be plotted like above:

```

random_scores = pd.DataFrame(ann_random_search.cv_results_)

random_scores.param_activation[random_scores.param_activation=="tanh"] = 0
random_scores.param_activation[random_scores.param_activation=="relu"] = 1
random_scores.param_activation[random_scores.param_activation=="linear"] = 2

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:3:
SettingWithCopyWarning:

```

A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4:
SettingWithCopyWarning:

```

A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:5:
SettingWithCopyWarning:

```

A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

```

fig = px.parallel_coordinates(random_scores.to_dict(),
                               color="mean_test_score",
                               dimensions=dimensions,

```

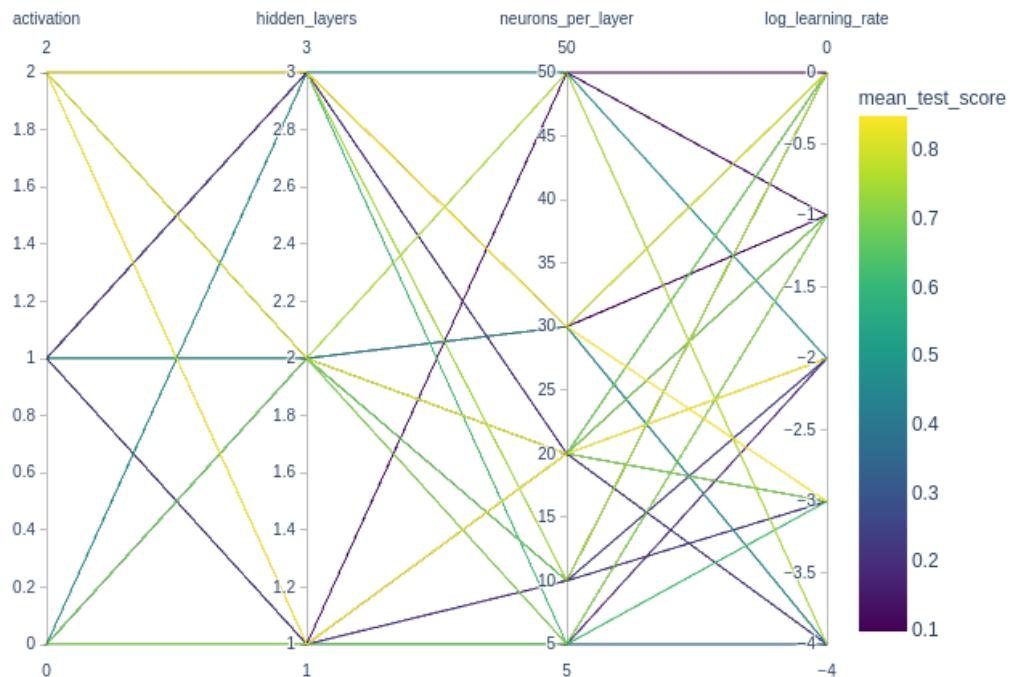
```

        labels=[s: s[6:] for s in dimensions],
        color_continuous_scale="viridis")#px.colors.

→diverging.Tealrose)

fig.show()
fig.write_image("img/plot_random_par_coords.png")

```



And the best score:

```

print("Best parameters:", ann_random_search.best_params_)

print("Best score: ", ann_random_search.best_score_)

ann_random_search.best_estimator_.score(X_test, y_test)

```

```

Best parameters: {'neurons_per_layer': 30, 'log_learning_rate': -3,
'hidden_layers': 3, 'activation': 'linear'}
Best score:  0.8498333333333333
63/63 [=====] - 0s 1ms/step - loss: 4.8608 - accuracy:
0.8480

```

0.8479999899864197

This is directly much better than the normal grid search result.

Normally, you'd have to manually "zoom in" to parameter space areas that yield good results, but there are ways to automate this adaptability.

7.12.4 Bayesian Search

The "zooming" process into interesting regions can be automated in several ways. One way is to use **Bayesian Search**, which randomly samples from the parameter space, then calculates an *uncertainty*. The next sampling points are chosen so that this uncertainty is optimally reduced. This should suffice as a short explanation for now, we'll cover Bayesian search again in a later lecture.

`scikit-learn` itself doesn't offer a simple class for performing Bayesian hyperparameter optimization, but there's a package that mimicks its interfaces called `scikit-optimize`, which additionally offers some interesting plotting capabilities on top of providing the optimization.

```
from skopt import BayesSearchCV
from skopt.space import Real, Categorical, Integer

param_space = {"hidden_layers": Integer(1, 3),
               "neurons_per_layer": Integer(5, 50),
               "log_learning_rate": Integer(-4, 0),
               "activation": Categorical(["relu", "tanh", "linear"])}
}

# we can use this function as a "callback", meaning that it
# will be executed after each iteration in the algorithm
def each_step(optim_result):
    print(f"Current best score: {ann_bayes_search.best_score_}")
    # The following line implements an "early stop", which we'll
    # discuss in the next lecture
    if ann_bayes_search.best_score_ >= 0.9:
        print('Score higher than given limit, stopped.')
        return True

ann_bayes_search = BayesSearchCV(ann_clf,
                                 param_space,
                                 n_iter=36,
                                 scoring="accuracy",
                                 n_jobs=4,
                                 cv=5)

ann_bayes_search.fit(X_train, y_train, callback=each_step)
```

Current best score: 0.4688333333333333

```
Current best score: 0.4688333333333333
Current best score: 0.4688333333333333
Current best score: 0.791
Current best score: 0.8181666666666667
Current best score: 0.8181666666666667
Current best score: 0.8181666666666667
Current best score: 0.8225
Current best score: 0.8225
Current best score: 0.8408333333333333

/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8408333333333333
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8408333333333333
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
Current best score: 0.8461666666666666
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666  
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

The objective has been evaluated at this point before.

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```
Current best score: 0.8461666666666666
```

```
/usr/local/lib/python3.7/dist-packages/skopt/optimizer/optimizer.py:449:  
UserWarning:
```

```
The objective has been evaluated at this point before.
```

```

Current best score: 0.8461666666666666
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_2_input'), name='flatten_2_input', description="created by layer
'flatten_2_input'"), but it was called on an input with incompatible shape
(None, 784).
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_2_input'), name='flatten_2_input', description="created by layer
'flatten_2_input'"), but it was called on an input with incompatible shape
(None, 784).
188/188 [=====] - 1s 2ms/step - loss: 37.2591 -
accuracy: 0.6447

```

```

BayesSearchCV(cv=5,
estimator=<tensorflow.python.keras.wrappers.scikit_learn.KerasClassifier object
at 0x7fbec8a125f8>,
    n_iter=36, n_jobs=-1, scoring='accuracy',
    search_spaces={'activation': Categorical(categories=('relu',
'tanh', 'linear'), prior=None),
                    'hidden_layers': Integer(low=1, high=3,
prior='uniform', transform='identity'),
                    'log_learning_rate': Integer(low=-4, high=0,
prior='uniform', transform='identity'),
                    'neurons_per_layer': Integer(low=5, high=50,
prior='uniform', transform='identity')})

```

We can plot the best parameters and evaluation score just like before:

```

print("Best parameters:", ann_bayes_search.best_params_)

print("Best score: ", ann_bayes_search.best_score_)

ann_bayes_search.best_estimator_.score(X_test, y_test)

```

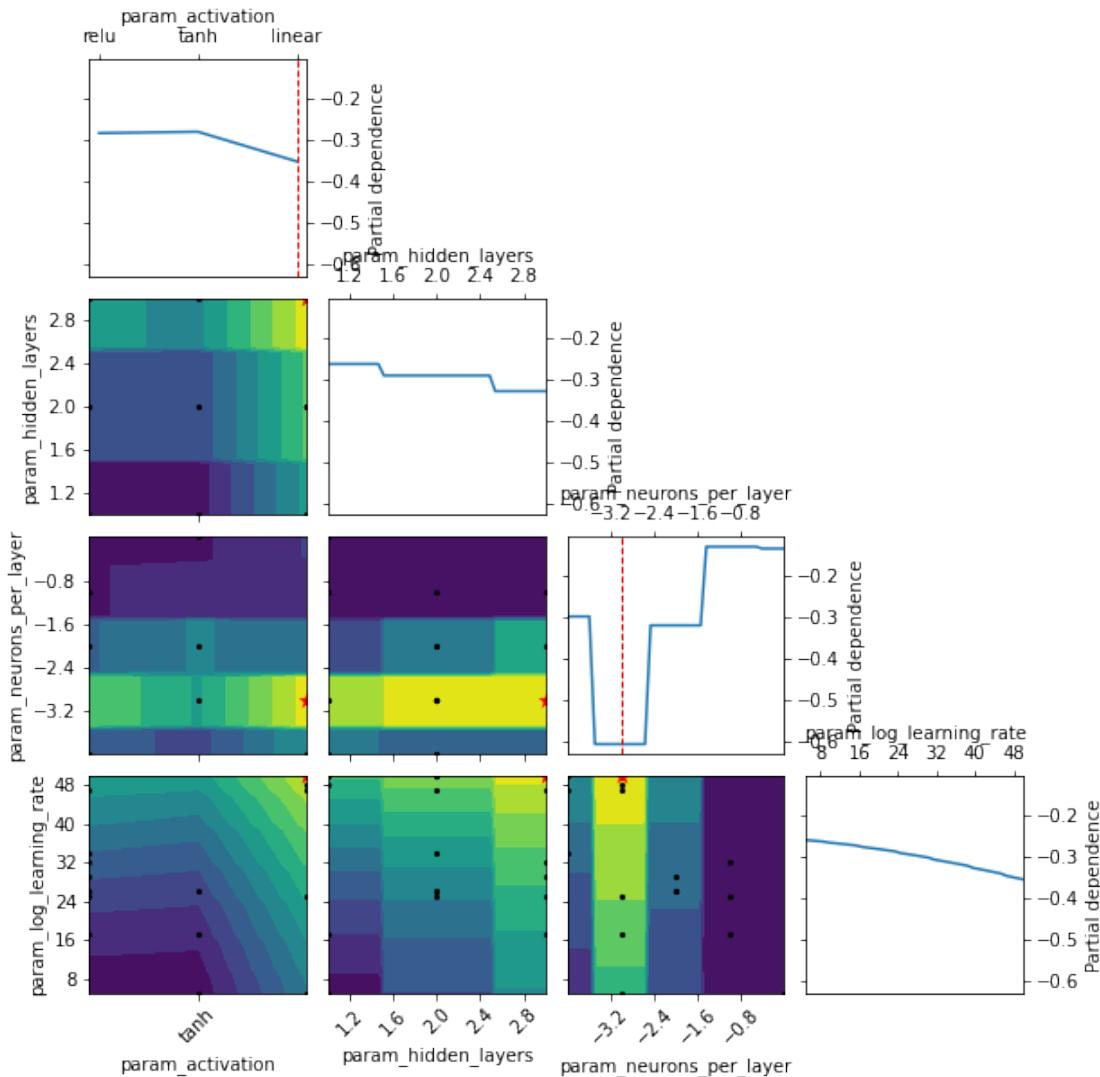
```

Best parameters: OrderedDict([('activation', 'linear'), ('hidden_layers', 3),
('log_learning_rate', -3), ('neurons_per_layer', 50)])
Best score:  0.8461666666666666
WARNING:tensorflow:Model was constructed with shape (None, 1, 784) for input
KerasTensor(type_spec=TensorSpec(shape=(None, 1, 784), dtype=tf.float32,
name='flatten_2_input'), name='flatten_2_input', description="created by layer
'flatten_2_input'"), but it was called on an input with incompatible shape
(None, 784).
63/63 [=====] - 0s 1ms/step - loss: 8.6168 - accuracy:
0.8240
0.8240000009536743

```

```
from skopt.plots import plot_objective

_ = plot_objective(ann_bayes_search.optimizer_results_[0],
                   dimensions=dimensions,
                   n_minimum_search=int(1e8))
```



The plot below shows a histogram of how many samples were drawn by the optimizer, hinting at the approximated importance of parameters:

```
import matplotlib.pyplot as plt
from skopt.plots import plot_histogram

fig, axs = plt.subplots(2,2, figsize=(12,7))
axs = axs.flatten()
```

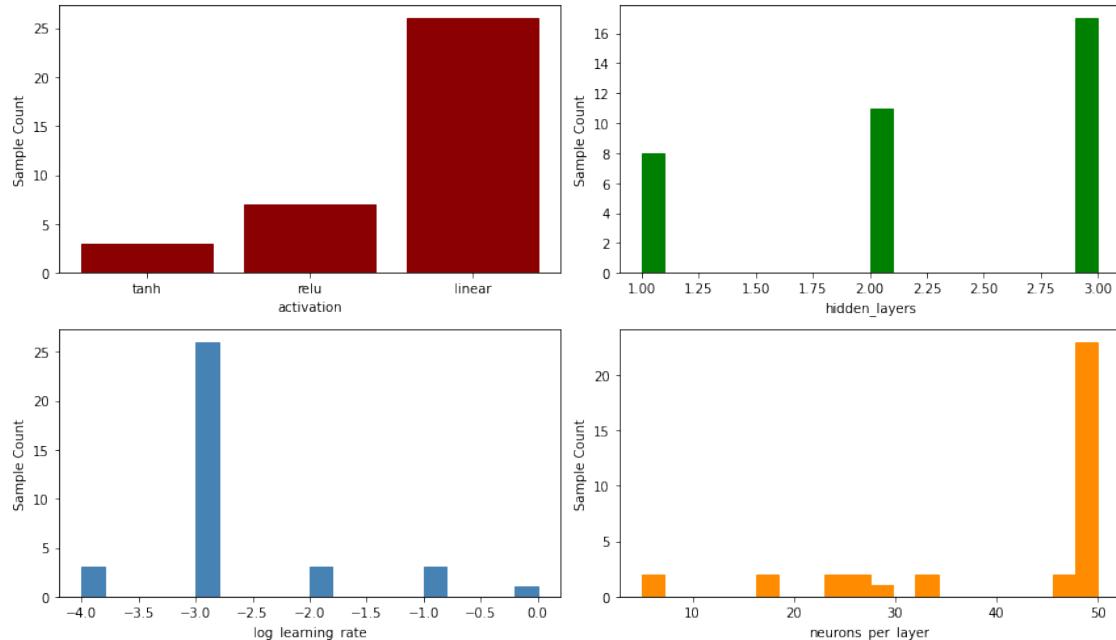
```

colors = ["darkred", "green", "steelblue", "darkorange"]

for i in range(4):
    plot_histogram(ann_bayes_search.optimizer_results_[0], i, ax=axs[i])
    for child in axs[i].properties()['children']:
        if "Rectangle" in str(child) and not "xy=(0, 0)" in str(child):
            child.set_color(colors[i])

plt.tight_layout()
# plt.show()

```



The same plot along with the actual points sampled can be created with the following function:

```

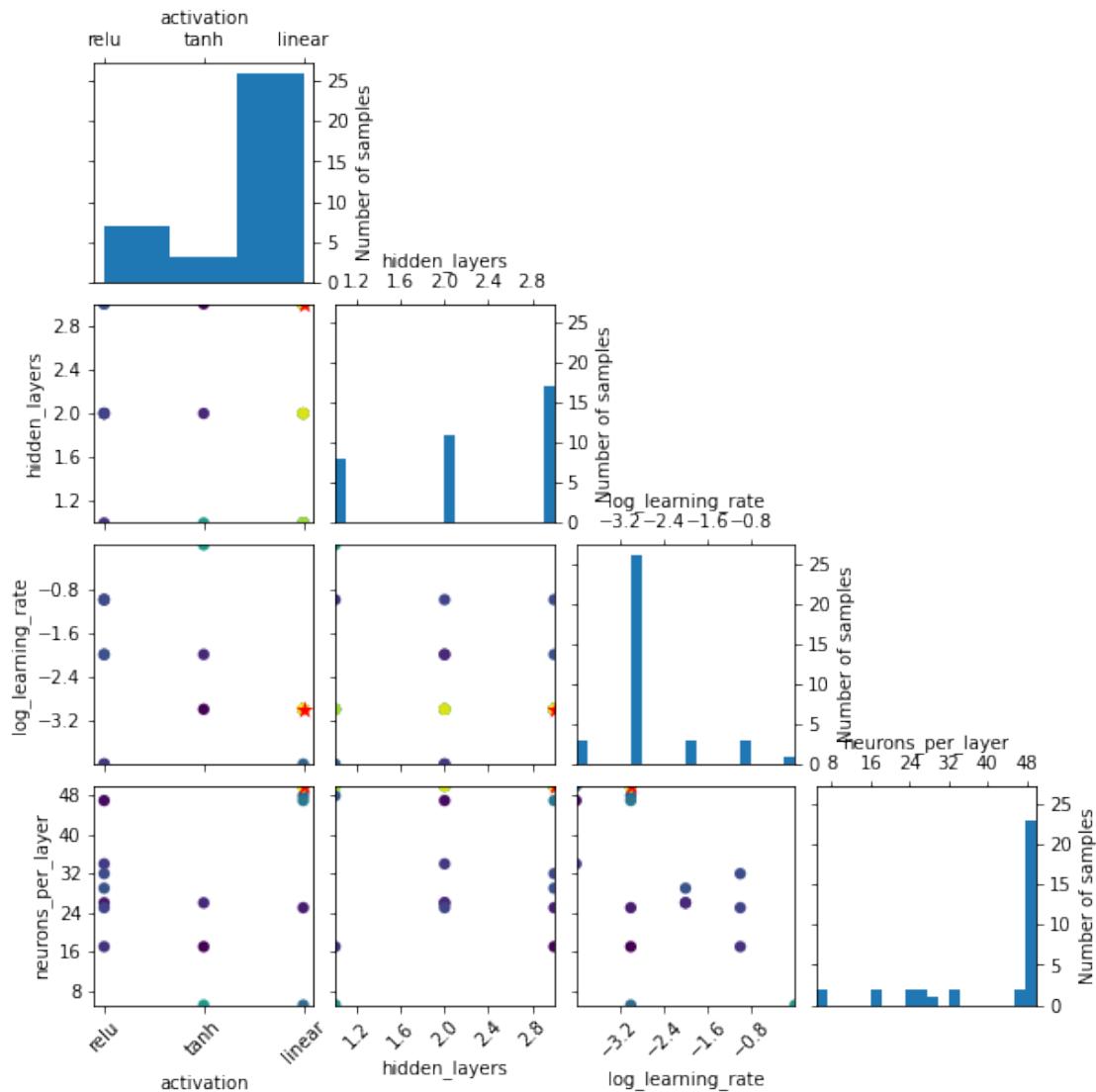
from skopt.plots import plot_evaluations

plt.figure()

_ = plot_evaluations(ann_bayes_search.optimizer_results_[0])

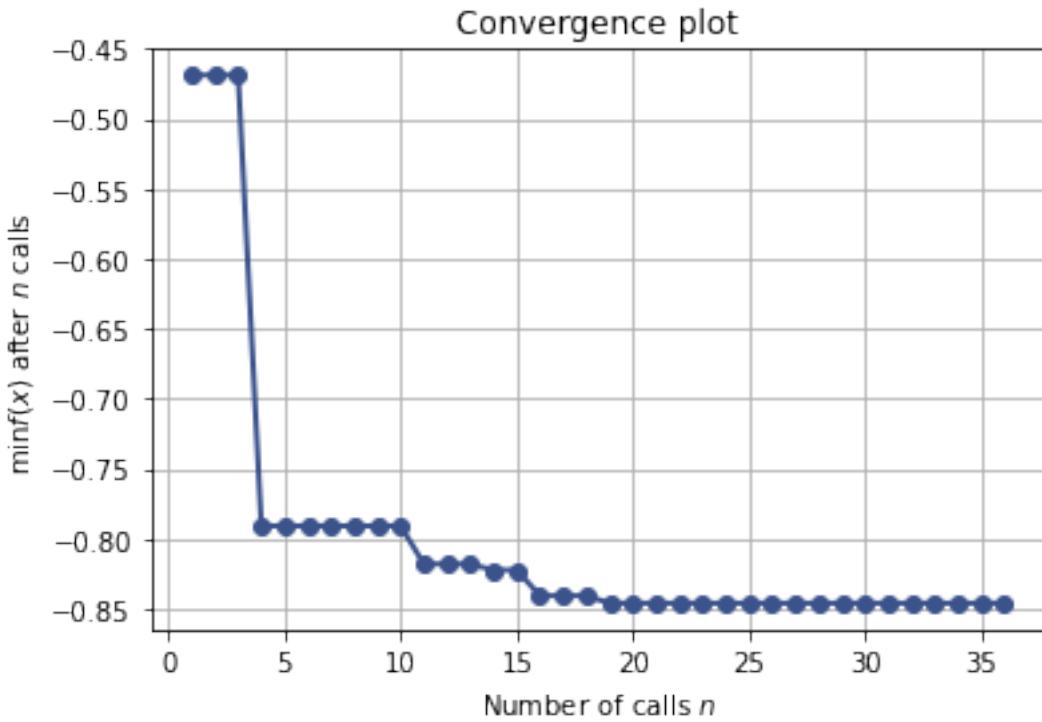
```

<Figure size 432x288 with 0 Axes>



We can also plot a convergence graph:

```
from skopt.plots import plot_convergence
_ = plot_convergence(ann_bayes_search.optimizer_results_[0])
```



7.12.5 Evolutional Hyperparameter Optimization

This is another option to automatically “zoom in” to interesting regions of the parameter space. A few algorithms were mentioned in the AML lecture, so we won’t go into too much detail here. In summary, the rough idea is the following:

- create a *population* of random sample configurations
- evaluate all of them
- take the (e.g.) 25% best of them
- sample new configurations around these best configurations
- repeat

In the AML lecture, covariance-controlled evolutionary strategies were discussed where the covariance of the distribution around the 25% best examples is adjusted to “get more samples closer to local minima”.

Notable libraries for these types of strategies are Google’s **AutoML** (40 hours of training free, then costs something) and **autokeras** (free and open source), but there are others for different frameworks (e.g. **AutoWeka** for Java programmers). We won’t use them in this course, but they are easy to use and easily outperform human experts in the construction of generic ANNs. The cost is that these strategies are extremely time-consuming and optimize for days on HPC clusters. A quick test with **autokeras** for creating a simple surrogate model of a complex simulation used at the ISD produced a better model than what was done manually in around 36 hours of training on a desktop computer with a small GPU. It’s a probabilistic process, so in general longer optimization runs might be necessary.

7.13 Visualizing Hyperparameter Search Results (optional)

7.13.1 TensorFlow

Let's quickly glance over two larger and popular frameworks for hyperparameter optimization and visualization. These are not important for this course, but may be helpful in the future.

The first is `tensorboard`, which was developed for the `TensorFlow` framework to visualize properties of (mainly) ANNs. The following part is simplified from the [TensorFlow documentation](#). In Jupyter, the extension can be loaded with a *magic command*:

```
%load_ext tensorboard
```

All the data necessary for visualizations in `tensorboard` are saved in a directory called `logs`, so at the beginning of an optimization, it makes sense to delete it just to make sure we'll see the right data after the optimization:

```
!rm -rf ./logs/
```

Next, we'll load the training data (MNIST again) and import everything we'll need:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Dense, Dropout
from tensorboard.plugins.hparams import api as hp
from tensorflow.summary import create_file_writer, scalar
import datetime

X = np.load("/data/X.npy", allow_pickle=True)[:1000]
y = np.load("/data/y.npy", allow_pickle=True)[:1000].astype(int) // 10

X_train = X[:600]
y_train = y[:600]
X_val = X[600:800]
y_val = y[600:800]
X_test = X[800:]
y_test = y[800:]
```

The syntax for defining the parameter space is similar to that of `scikit-optimize`. The following code writes the hyperparameter optimization into a file. It's binary encoded, so you can't read it directly.

```
HP_NUM_UNITS = hp.HParam('num_units', hp.Discrete([16, 32]))
HP_DROPOUT = hp.HParam('dropout', hp.RealInterval(0.1, 0.2))
HP_OPTIMIZER = hp.HParam('optimizer', hp.Discrete(['adam', 'sgd']))

METRIC_ACCURACY = 'accuracy'

with create_file_writer('logs/hparam_tuning').as_default():
    hp.hparams_config(hparams=[HP_NUM_UNITS, HP_DROPOUT, HP_OPTIMIZER],
```

```
metrics=[hp.Metric(METRIC_ACCURACY, ↴
→display_name='Accuracy')],)
```

The following code automates creating and running the models for each set of hyperparameters:

```
# this is the objective function to optimize
def train_test_model(hparams):
    model = Sequential([Flatten(),
                        Dense(hparams[HP_NUM_UNITS], activation="relu"),
                        Dropout(hparams[HP_DROPOUT]),
                        Dense(10, activation="linear"),])

    model.compile(optimizer=hparams[HP_OPTIMIZER],
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(X_train, y_train, epochs=1)

    _, accuracy = model.evaluate(X_test, y_test)

    return accuracy

def run(run_dir, hparams):
    with create_file_writer(run_dir).as_default():
        hp.hparams(hparams) # record the values used in this trial
        accuracy = train_test_model(hparams)
        scalar(METRIC_ACCURACY, accuracy, step=1)
```

Unfortunately, we have to manually loop over all parameters:

```
session_num = 0

for num_units in HP_NUM_UNITS.domain.values:
    for dropout_rate in (HP_DROPOUT.domain.min_value, HP_DROPOUT.domain.
→max_value):
        for optimizer in HP_OPTIMIZER.domain.values:
            hparams = {
                HP_NUM_UNITS: num_units,
                HP_DROPOUT: dropout_rate,
                HP_OPTIMIZER: optimizer,
            }
            run_name = f"run-{session_num}"
            print(f'--- Starting trial: {run_name}')
            print({h.name: hparams[h] for h in hparams})
            run('logs/hparam_tuning/' + run_name, hparams)
            session_num += 1
```

--- Starting trial: run-0

```
{'num_units': 16, 'dropout': 0.1, 'optimizer': 'adam'}
19/19 [=====] - 0s 2ms/step - loss: 5.2751 - accuracy:
0.3888
7/7 [=====] - 0s 1ms/step - loss: 0.9558 - accuracy:
0.9300
--- Starting trial: run-1
{'num_units': 16, 'dropout': 0.1, 'optimizer': 'sgd'}
19/19 [=====] - 0s 1ms/step - loss: 1.9812 - accuracy:
0.7455
7/7 [=====] - 0s 1ms/step - loss: 1.1322 - accuracy:
0.9950
--- Starting trial: run-2
{'num_units': 16, 'dropout': 0.2, 'optimizer': 'adam'}
19/19 [=====] - 0s 2ms/step - loss: 4.8082 - accuracy:
0.0376
7/7 [=====] - 0s 1ms/step - loss: 2.4482 - accuracy:
0.1450
--- Starting trial: run-3
{'num_units': 16, 'dropout': 0.2, 'optimizer': 'sgd'}
19/19 [=====] - 0s 1ms/step - loss: 11.7530 - accuracy:
0.0024
7/7 [=====] - 0s 1ms/step - loss: 6.1205 - accuracy:
0.0100
--- Starting trial: run-4
{'num_units': 32, 'dropout': 0.1, 'optimizer': 'adam'}
19/19 [=====] - 0s 2ms/step - loss: 4.2391 - accuracy:
0.5451
7/7 [=====] - 0s 2ms/step - loss: 0.5289 - accuracy:
0.9550
--- Starting trial: run-5
{'num_units': 32, 'dropout': 0.1, 'optimizer': 'sgd'}
19/19 [=====] - 0s 1ms/step - loss: 6.4987 - accuracy:
0.0915
7/7 [=====] - 0s 1ms/step - loss: 1.1170 - accuracy:
0.9750
--- Starting trial: run-6
{'num_units': 32, 'dropout': 0.2, 'optimizer': 'adam'}
19/19 [=====] - 0s 2ms/step - loss: 4.8190 - accuracy:
0.1116
7/7 [=====] - 0s 1ms/step - loss: 2.1636 - accuracy:
0.2150
--- Starting trial: run-7
{'num_units': 32, 'dropout': 0.2, 'optimizer': 'sgd'}
19/19 [=====] - 0s 1ms/step - loss: 4.5900 - accuracy:
0.2892
7/7 [=====] - 0s 1ms/step - loss: 0.7134 - accuracy:
1.0000
```

Tensorboard is implemented as a small server, which you can start with the following line (if you're doing all of this on your own system, you can issue the same command without the "!" in a shell)::

```
!tensorboard --logdir logs/hparam_tuning --host ↴  
→b2858809-a4c2-46dc-a0d7-01e48260a138.ma.bw-cloud-instance.org --port 6006
```

```
2021-05-18 10:35:42.479529: W tensorflow/stream_executor/platform/default/dso_loader.cc:60] Could not load dynamic library 'libcudart.so.11.0'; dlerror: libcudart.so.11.0: cannot open shared object file: No such file or directory  
2021-05-18 10:35:42.479588: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.  
TensorBoard 2.4.1 at http://b2858809-a4c2-46dc-a0d7-01e48260a138.ma.bw-cloud-instance.org:6006/ (Press CTRL+C to quit)  
^C
```

You can click on the link above to open it. To close the tensorboard, click on the black square on top of this page (interrupt the kernel) and execute the following cell. You should always do this to close the server, since otherwise you may get errors the next time you run it and also to save resources on this course server.

```
!pkill tensorboard
```

Another larger package for hyperparameter tuning is `optuna`, which offers various ways to visualize results and the importance of different parameters.

7.14 Data Augmentation

We talked about how CNNs show (limited!) translational invariance and equivariance in the last lecture. There is a higher theory to this behavior that is connected to *Noether's Theorem*, which we'll discuss in the last lecture of this course. The gist is that we basically have two options to create models that are invariant under some transformation of the dataset:

- 1.: build invariance into the network by architectural means
- 2.: make the dataset contain samples that are transformed in a way the network would deem different

The second point means that if the network itself isn't invariant under some transformation of the dataset, it can *learn* that invariance *from the dataset*, given that the samples reflect that symmetry. If we have a low amount of data available, and this is the standard case for image-based tasks, we can make use of that property to create more training data by applying the transformations we'd like to have the network invariant to. This is called **data augmentation**.

There are various transformations that especially CNNs should be invariant to. The most common ones are

- translations (significant ones)
- rotations
- reflections
- scaling
- cropping

- lighting
- noise
- perspective

and all of these can be used to create larger *augmented* datasets for training.

When your original dataset is small, each transformation you apply doubles the size of that dataset if combinations are allowed. Not all combinations make sense, since some of them produce the same output, e.g. rotating counterclockwise by 180°, then reflecting vertically creates the same image as a horizontal reflection (*side note: this is formalized in group theory*). This approach is the **offline** approach and really makes the dataset size explode in size, so may not be the best option in all circumstances.

When the original dataset is already quite large, an **online** approach of applying the transformations is preferable. Here, the transformations are applied onto the mini-batches fed into the network directly before entering the first layer. Most frameworks offer a way to do this by introduced “layers” that handle the transformations.

`keras`, which is a part of tensorflow since version 2.0, offers an automated way to do things. Anyway it still makes sense to take a quick look at what is going on behind the scenes before moving to the automatic version. At least for a few of the transformations.

7.14.1 Reflections

Let’s start with reflections, since these are the easiest to implement because the image size won’t change. First, we need to load some data:

```
import numpy as np
import matplotlib.pyplot as plt

X = np.load("/data/X.npy", allow_pickle=True)[:10000]
y = np.load("/data/y.npy", allow_pickle=True)[:10000].astype(int)

# reflecting numbers doesn't make much sense, since they are oriented
# let's take 0 as an example, as it provides the maximal possible symmetry
img = X[1].reshape(28,28)
y[1]
```

0

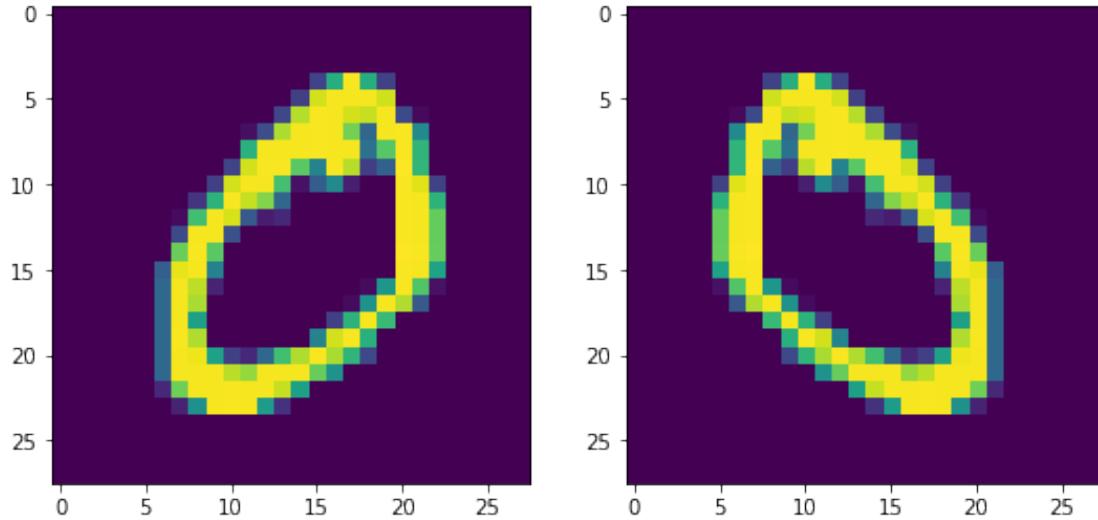
There’s no single right way to do things, but we can use NumPy:

```
# type in "np.flip" and press the tab key to see flip operations implemented in
# numpy
img_flipped = np.fliplr(img)

fig,axs = plt.subplots(1,2, figsize=(9,5))

axs[0].imshow(img)
axs[1].imshow(img_flipped)
```

<matplotlib.image.AxesImage at 0x7fc0f00cccf8>



or TensorFlow:

```
from tensorflow import image

# similar to above, type in "image." and press the tab button
# to see a list of operations that can be applied to images

# tensorflow needs 3D image arrays, where the last dimension is
# the number of channels in the image:
tf_img = img.reshape(28,28,1)

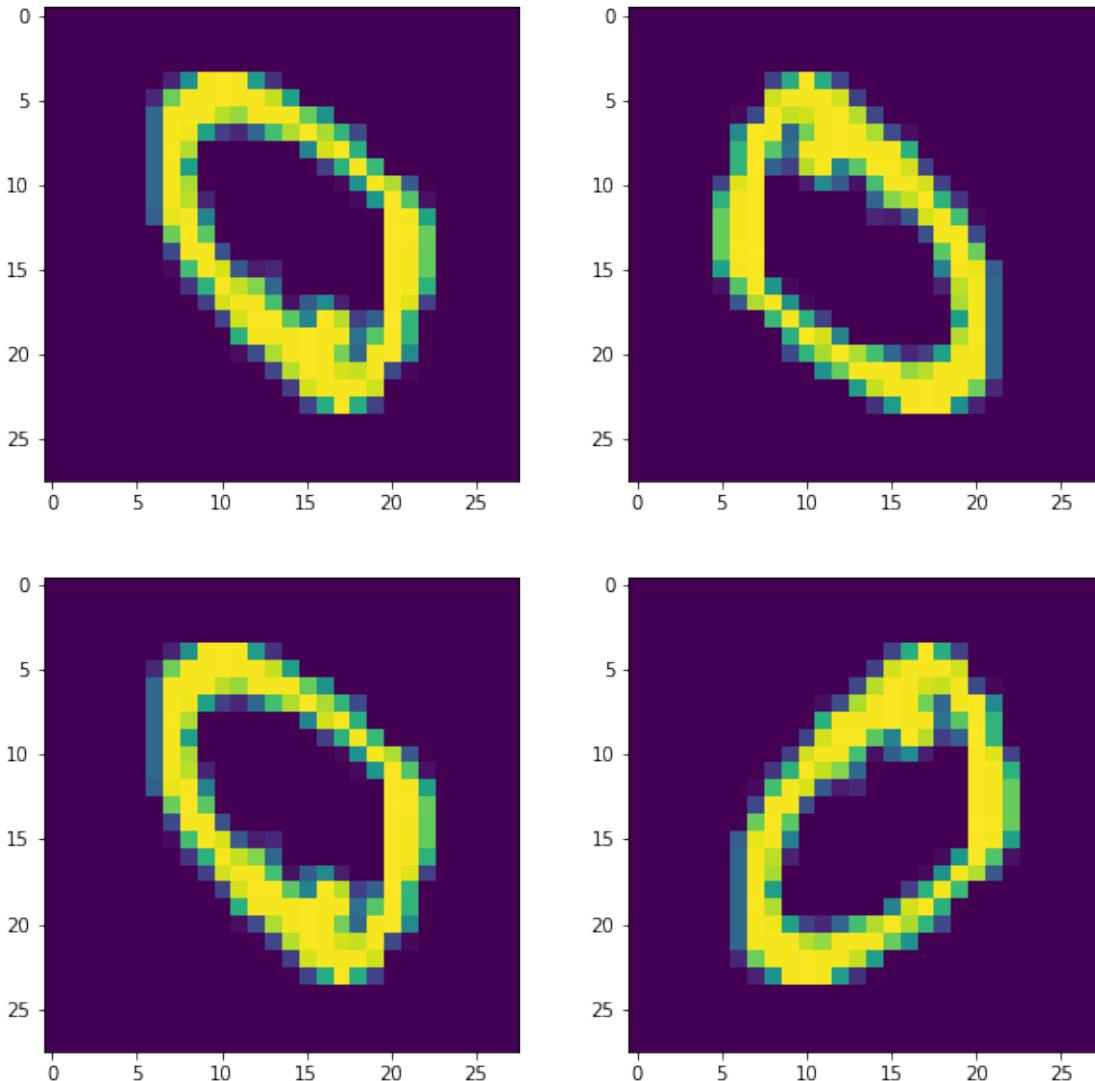
img_ud  = image.flip_up_down(tf_img)
img_lr  = image.flip_left_right(tf_img)

# tensorflow also offers functions that randomly flip an image
img_rud = image.random_flip_up_down(tf_img)
img_rlr = image.random_flip_left_right(tf_img)

fig,axs = plt.subplots(2,2, figsize=(10,10))
axs = axs.flatten()

axs[0].imshow(img_ud)
axs[1].imshow(img_lr)
axs[2].imshow(img_rud)
axs[3].imshow(img_rlr)
```

<matplotlib.image.AxesImage at 0x7fc0f84650f0>



`scikit-learn` has an `image` module that also provides rich functionality for manipulating images, but we'll stick to tensorflow here for the next examples.

7.14.2 Translations

We could use numpy's `roll` function like we did in the examples in the last lecture, but that wouldn't translate the image freely. Instead, it's moved on a *torus*, where the part leaving the bounding box turns up on the other side again. That's not what we want here.

In tensorflow, the idea is to introduce a *padding* to one of the sides and move the rest of the pixels accordingly, to preserve the original image size:

```
from ipywidgets import interact
```

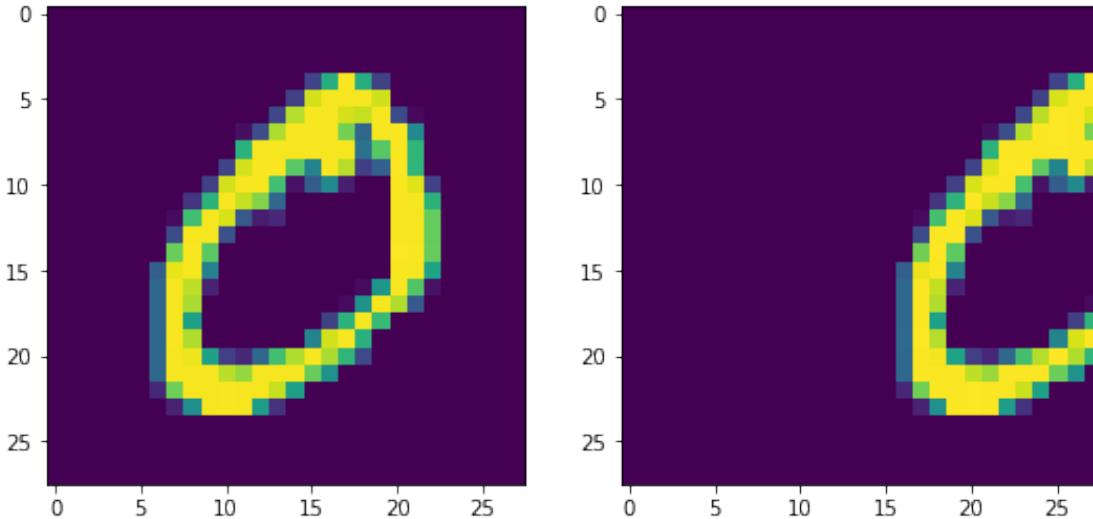
```
# determine the padding. Set one of the to some value
# to induce a translation, leave the rest to 0. Vertical
# and horizontal movement can be combined.
pad_left, pad_right, pad_top, pad_bottom = 10, 0, 0, 0

# the following line adds the above paddings to the image
x = image.pad_to_bounding_box(tf_img, pad_top, pad_left, height + pad_bottom +_
    pad_top, width + pad_right + pad_left)
# crop the image to original size to emulate a translation
output = image.crop_to_bounding_box(x, pad_bottom, pad_right, height, width)

fig,axs = plt.subplots(1,2, figsize=(9,5))

axs[0].imshow(tf_img)
axs[1].imshow(output)
```

<matplotlib.image.AxesImage at 0x7fc0f0486fd0>



Of course this only works if the original image has a constant background, like a single object and nothing else. Otherwise, more thought has to be put into inducing translations as well as the following operations. For example, you may need to crop the original image at different positions. We'll see an example in the assignment today.

7.14.3 Rotations

In general, image dimensions aren't retained when rotating, especially with non-right angles. Hence, the images must either be appropriately cropped from the original image (rotate original image, then crop), or padding like above has to be used.

The `k` argument controls how many times an image is rotated by 90 degrees. For arbitrary rotations,

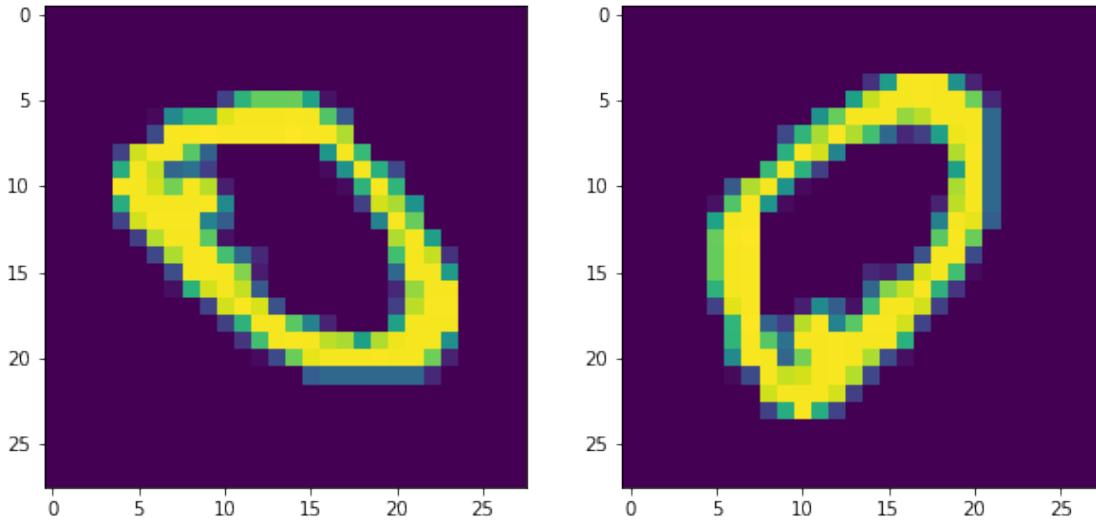
addons are necessary.

```
rot_1 = image.rot90(tf_img, k=1)
rot_2 = image.rot90(tf_img, k=2)

fig,axs = plt.subplots(1,2, figsize=(10,10))
axs = axs.flatten()

axs[0].imshow(rot_1)
axs[1].imshow(rot_2)
```

<matplotlib.image.AxesImage at 0x7fc0e02e1cf8>



7.14.4 Noise

In most cases, models should also be robust to *noise* in the dataset. E.g., changing weather and lighting conditions when taking pictures as data, or general noise in sensors creating data. This can be emulated with **Gaussian noise**, which means adding a random value sampled from a distribution with zero mean and some tunable variance. This also has a regularizing effect, which we'll briefly talk about in the next lecture.

Sometimes a simpler version of noise is used called *salt and pepper* noise, which distributes black and white pixels randomly on the image.

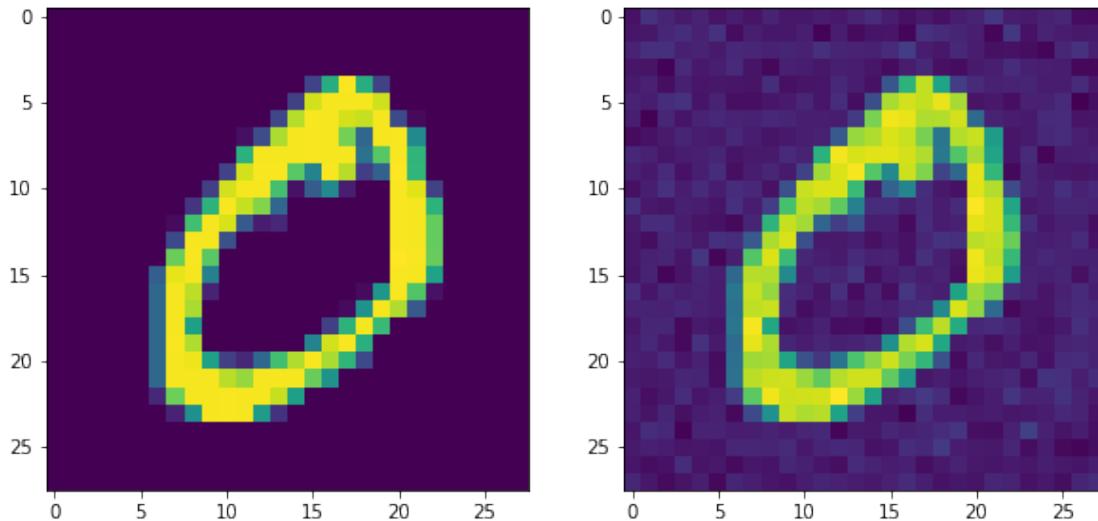
```
from tensorflow.random import normal
from tensorflow.math import add

noise = normal(shape=tf_img.shape, mean=0.0, stddev=10.0, dtype=np.float64)
img_noise = add(tf_img, noise)
```

```
fig,axs = plt.subplots(1,2, figsize=(10,10))

axs[0].imshow(tf_img)
axs[1].imshow(img_noise)
```

<matplotlib.image.AxesImage at 0x7fbfdc0b3160>



7.14.5 Automatic Augmentation with Keras

The above examples should suffice for a short demonstration, since the other operations are rather straightforward to apply. We actually already used the function we need in assignment 02, but here, we'll use it in an online way, since it provides us with a *generator*. Let's first look at how generators work in Python in general, to get a feeling for what is going on. The difference between a normal function and a generator is that `yield` is used instead of a `return` statement. The following example creates a sequence of the squares of numbers in the list `numbers`:

```
def square(numbers):
    for number in numbers:
        yield number**2
```

There are different ways to use generators. One could directly loop over them:

```
numberlist = [1,4,9,11]

for res in square(numberlist):
    print(res)
```

81
121

Or use the `next` keyword to get the next item that is generated. For doing so, an iterator has to be defined first:

```
iterator = square(numberlist)
```

After which we can use `next` to actually produce the outputs:

```
next(iterator)
```

```
StopIteration
↳-----last
              Traceback (most recent call)
              <ipython-input-9-4ce711c44abc> in <module>
              ----> 1 next(iterator)

StopIteration:
```

repeatedly to get all the values. Note that when the number sequence is iterated over, `next` will throw an exception. There are also infinite generators, like the following that creates all natural numbers:

```
def natural_numbers():
    number = 0
    while True:
        yield number
        number += 1

iterator = natural_numbers()
```

We can use this in the same way as before, but the for loop will be infinite. The `next` keyword gets the sequence step by step:

```
next(iterator)
```

29

And it will never stop. The `ImageDataGenerator` from `keras` works the same way, by generating batches of images to which random augmentation techniques are applied, never stopping. You can consult the documentation to see which options are available. Let's see how to use it with a small selection of augmentations. First, we need to prepare the data:

```
from tensorflow import one_hot

X_train = X.reshape(100, 28, 28, 1)
y_train = one_hot(y, 10)
```

Then, a simple convolutional model:

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dropout, Flatten, ↴
    ↴Dense

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=X_train. ↴
    ↴shape[1:]))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

model.compile(loss="categorical_crossentropy", optimizer="adam", ↴
    ↴metrics=['accuracy'])
```

And the data generator:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    brightness_range=[0.3,1],
    zoom_range=0.4,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True,
    vertical_flip=True,
    validation_split=0.2)

# fit the normalization values to the data
datagen.fit(X_train)
```

The following line shows how to fit a model to the data generated by the `datagen` iterator. It

doesn't work as directly as in the simple Python example from before. The actual iterator is called with the `flow` method, which then yields batches of augmented images.

```
model.fit(datagen.flow(X_train, y_train, batch_size=32, subset='training'),
          validation_data=datagen.flow(X_train, y_train, batch_size=8,
→subset='validation'),
          steps_per_epoch=len(X_train) // 32, epochs=100)
```

```
Epoch 1/100
3/3 [=====] - 0s 79ms/step - loss: 2.2570 - accuracy: 0.1250 - val_loss: 2.3623 - val_accuracy: 0.0500
Epoch 2/100
3/3 [=====] - 0s 62ms/step - loss: 2.2547 - accuracy: 0.1625 - val_loss: 2.3937 - val_accuracy: 0.0500
Epoch 3/100
3/3 [=====] - 0s 66ms/step - loss: 2.2514 - accuracy: 0.1375 - val_loss: 2.3863 - val_accuracy: 0.0500
Epoch 4/100
3/3 [=====] - 0s 54ms/step - loss: 2.2804 - accuracy: 0.1375 - val_loss: 2.3334 - val_accuracy: 0.1000
Epoch 5/100
3/3 [=====] - 0s 60ms/step - loss: 2.2599 - accuracy: 0.1250 - val_loss: 2.3056 - val_accuracy: 0.0500
Epoch 6/100
3/3 [=====] - 0s 56ms/step - loss: 2.2508 - accuracy: 0.1625 - val_loss: 2.3058 - val_accuracy: 0.0500
Epoch 7/100
3/3 [=====] - 0s 55ms/step - loss: 2.2580 - accuracy: 0.1375 - val_loss: 2.3058 - val_accuracy: 0.1000
Epoch 8/100
3/3 [=====] - 0s 57ms/step - loss: 2.2418 - accuracy: 0.1875 - val_loss: 2.3184 - val_accuracy: 0.1000
Epoch 9/100
3/3 [=====] - 0s 57ms/step - loss: 2.2609 - accuracy: 0.1375 - val_loss: 2.3333 - val_accuracy: 0.1500
Epoch 10/100
3/3 [=====] - 0s 64ms/step - loss: 2.2140 - accuracy: 0.1375 - val_loss: 2.3264 - val_accuracy: 0.1000
Epoch 11/100
3/3 [=====] - 0s 66ms/step - loss: 2.2380 - accuracy: 0.1250 - val_loss: 2.3258 - val_accuracy: 0.2000
Epoch 12/100
3/3 [=====] - 0s 56ms/step - loss: 2.2425 - accuracy: 0.1875 - val_loss: 2.3343 - val_accuracy: 0.0500
Epoch 13/100
3/3 [=====] - 0s 56ms/step - loss: 2.2382 - accuracy: 0.1125 - val_loss: 2.3146 - val_accuracy: 0.0500
Epoch 14/100
```

```
3/3 [=====] - 0s 55ms/step - loss: 2.2652 - accuracy:  
0.1500 - val_loss: 2.2981 - val_accuracy: 0.1000  
Epoch 15/100  
3/3 [=====] - 0s 60ms/step - loss: 2.2432 - accuracy:  
0.1750 - val_loss: 2.3225 - val_accuracy: 0.0500  
Epoch 16/100  
3/3 [=====] - 0s 56ms/step - loss: 2.2172 - accuracy:  
0.2125 - val_loss: 2.3390 - val_accuracy: 0.0500  
Epoch 17/100  
3/3 [=====] - 0s 64ms/step - loss: 2.2150 - accuracy:  
0.1750 - val_loss: 2.3413 - val_accuracy: 0.0500  
Epoch 18/100  
3/3 [=====] - 0s 59ms/step - loss: 2.2338 - accuracy:  
0.1625 - val_loss: 2.3681 - val_accuracy: 0.0500  
Epoch 19/100  
3/3 [=====] - 0s 53ms/step - loss: 2.2575 - accuracy:  
0.1500 - val_loss: 2.3232 - val_accuracy: 0.1500  
Epoch 20/100  
3/3 [=====] - 0s 54ms/step - loss: 2.2497 - accuracy:  
0.1000 - val_loss: 2.3035 - val_accuracy: 0.1000  
Epoch 21/100  
3/3 [=====] - 0s 62ms/step - loss: 2.2539 - accuracy:  
0.1250 - val_loss: 2.3004 - val_accuracy: 0.2500  
Epoch 22/100  
3/3 [=====] - 0s 57ms/step - loss: 2.2168 - accuracy:  
0.1625 - val_loss: 2.3243 - val_accuracy: 0.1500  
Epoch 23/100  
3/3 [=====] - 0s 63ms/step - loss: 2.1964 - accuracy:  
0.2375 - val_loss: 2.3446 - val_accuracy: 0.1500  
Epoch 24/100  
3/3 [=====] - 0s 66ms/step - loss: 2.2069 - accuracy:  
0.1875 - val_loss: 2.3696 - val_accuracy: 0.1000  
Epoch 25/100  
3/3 [=====] - 0s 62ms/step - loss: 2.2721 - accuracy:  
0.1375 - val_loss: 2.3785 - val_accuracy: 0.0500  
Epoch 26/100  
3/3 [=====] - 0s 56ms/step - loss: 2.2385 - accuracy:  
0.1250 - val_loss: 2.3431 - val_accuracy: 0.1000  
Epoch 27/100  
3/3 [=====] - 0s 54ms/step - loss: 2.2244 - accuracy:  
0.2250 - val_loss: 2.3220 - val_accuracy: 0.0500  
Epoch 28/100  
3/3 [=====] - 0s 53ms/step - loss: 2.2133 - accuracy:  
0.2000 - val_loss: 2.3176 - val_accuracy: 0.0500  
Epoch 29/100  
3/3 [=====] - 0s 58ms/step - loss: 2.1995 - accuracy:  
0.1750 - val_loss: 2.3859 - val_accuracy: 0.1000  
Epoch 30/100
```

```
3/3 [=====] - 0s 59ms/step - loss: 2.2343 - accuracy: 0.2000 - val_loss: 2.4288 - val_accuracy: 0.0500
Epoch 31/100
3/3 [=====] - 0s 66ms/step - loss: 2.2329 - accuracy: 0.1625 - val_loss: 2.4184 - val_accuracy: 0.0500
Epoch 32/100
3/3 [=====] - 0s 62ms/step - loss: 2.2607 - accuracy: 0.1500 - val_loss: 2.3251 - val_accuracy: 0.0500
Epoch 33/100
3/3 [=====] - 0s 55ms/step - loss: 2.2301 - accuracy: 0.1625 - val_loss: 2.3020 - val_accuracy: 0.1000
Epoch 34/100
3/3 [=====] - 0s 60ms/step - loss: 2.2495 - accuracy: 0.1500 - val_loss: 2.2849 - val_accuracy: 0.1500
Epoch 35/100
3/3 [=====] - 0s 191ms/step - loss: 2.2482 - accuracy: 0.1625 - val_loss: 2.3040 - val_accuracy: 0.1000
Epoch 36/100
3/3 [=====] - 0s 55ms/step - loss: 2.2191 - accuracy: 0.2375 - val_loss: 2.3211 - val_accuracy: 0.1500
Epoch 37/100
3/3 [=====] - 0s 56ms/step - loss: 2.2368 - accuracy: 0.1625 - val_loss: 2.3124 - val_accuracy: 0.1500
Epoch 38/100
3/3 [=====] - 0s 54ms/step - loss: 2.2183 - accuracy: 0.1750 - val_loss: 2.3427 - val_accuracy: 0.0500
Epoch 39/100
3/3 [=====] - 0s 62ms/step - loss: 2.2516 - accuracy: 0.1250 - val_loss: 2.3563 - val_accuracy: 0.0500
Epoch 40/100
3/3 [=====] - 0s 58ms/step - loss: 2.2702 - accuracy: 0.1125 - val_loss: 2.3484 - val_accuracy: 0.0500
Epoch 41/100
3/3 [=====] - 0s 66ms/step - loss: 2.2495 - accuracy: 0.1500 - val_loss: 2.3363 - val_accuracy: 0.0500
Epoch 42/100
3/3 [=====] - 0s 66ms/step - loss: 2.2579 - accuracy: 0.1625 - val_loss: 2.3052 - val_accuracy: 0.0500
Epoch 43/100
3/3 [=====] - 0s 57ms/step - loss: 2.2437 - accuracy: 0.1750 - val_loss: 2.2972 - val_accuracy: 0.1000
Epoch 44/100
3/3 [=====] - 0s 58ms/step - loss: 2.2326 - accuracy: 0.2000 - val_loss: 2.3195 - val_accuracy: 0.0500
Epoch 45/100
3/3 [=====] - 0s 59ms/step - loss: 2.2140 - accuracy: 0.1750 - val_loss: 2.3043 - val_accuracy: 0.1000
Epoch 46/100
```

```
3/3 [=====] - 0s 58ms/step - loss: 2.1908 - accuracy: 0.2000 - val_loss: 2.3261 - val_accuracy: 0.0500
Epoch 47/100
3/3 [=====] - 0s 62ms/step - loss: 2.2237 - accuracy: 0.1625 - val_loss: 2.3352 - val_accuracy: 0.1500
Epoch 48/100
3/3 [=====] - 0s 62ms/step - loss: 2.2121 - accuracy: 0.1500 - val_loss: 2.3268 - val_accuracy: 0.0500
Epoch 49/100
3/3 [=====] - 0s 66ms/step - loss: 2.2027 - accuracy: 0.2000 - val_loss: 2.3490 - val_accuracy: 0.0500
Epoch 50/100
3/3 [=====] - 0s 69ms/step - loss: 2.2255 - accuracy: 0.1625 - val_loss: 2.2740 - val_accuracy: 0.2000
Epoch 51/100
3/3 [=====] - 0s 65ms/step - loss: 2.2285 - accuracy: 0.1750 - val_loss: 2.3122 - val_accuracy: 0.1000
Epoch 52/100
3/3 [=====] - 0s 55ms/step - loss: 2.1884 - accuracy: 0.1875 - val_loss: 2.3387 - val_accuracy: 0.0500
Epoch 53/100
3/3 [=====] - 0s 62ms/step - loss: 2.1893 - accuracy: 0.2125 - val_loss: 2.2746 - val_accuracy: 0.1500
Epoch 54/100
3/3 [=====] - 0s 57ms/step - loss: 2.2152 - accuracy: 0.2375 - val_loss: 2.3092 - val_accuracy: 0.1000
Epoch 55/100
3/3 [=====] - 0s 58ms/step - loss: 2.2264 - accuracy: 0.1625 - val_loss: 2.3236 - val_accuracy: 0.0500
Epoch 56/100
3/3 [=====] - 0s 65ms/step - loss: 2.2391 - accuracy: 0.1625 - val_loss: 2.3213 - val_accuracy: 0.1000
Epoch 57/100
3/3 [=====] - 0s 55ms/step - loss: 2.2313 - accuracy: 0.1500 - val_loss: 2.3033 - val_accuracy: 0.1500
Epoch 58/100
3/3 [=====] - 0s 56ms/step - loss: 2.2316 - accuracy: 0.1500 - val_loss: 2.2813 - val_accuracy: 0.2000
Epoch 59/100
3/3 [=====] - 0s 53ms/step - loss: 2.2110 - accuracy: 0.1875 - val_loss: 2.3007 - val_accuracy: 0.1000
Epoch 60/100
3/3 [=====] - 0s 55ms/step - loss: 2.1973 - accuracy: 0.1625 - val_loss: 2.3122 - val_accuracy: 0.1000
Epoch 61/100
3/3 [=====] - 0s 58ms/step - loss: 2.2059 - accuracy: 0.1500 - val_loss: 2.3210 - val_accuracy: 0.0500
Epoch 62/100
```

```
3/3 [=====] - 0s 57ms/step - loss: 2.1951 - accuracy: 0.2250 - val_loss: 2.3007 - val_accuracy: 0.1000
Epoch 63/100
3/3 [=====] - 0s 57ms/step - loss: 2.2020 - accuracy: 0.1625 - val_loss: 2.3133 - val_accuracy: 0.1000
Epoch 64/100
3/3 [=====] - 0s 64ms/step - loss: 2.2326 - accuracy: 0.1875 - val_loss: 2.3320 - val_accuracy: 0.1500
Epoch 65/100
3/3 [=====] - 0s 58ms/step - loss: 2.1955 - accuracy: 0.2000 - val_loss: 2.3275 - val_accuracy: 0.1000
Epoch 66/100
3/3 [=====] - 0s 59ms/step - loss: 2.2284 - accuracy: 0.2125 - val_loss: 2.3125 - val_accuracy: 0.1000
Epoch 67/100
3/3 [=====] - 0s 59ms/step - loss: 2.1726 - accuracy: 0.2250 - val_loss: 2.3162 - val_accuracy: 0.0500
Epoch 68/100
3/3 [=====] - 0s 57ms/step - loss: 2.2194 - accuracy: 0.1750 - val_loss: 2.2886 - val_accuracy: 0.1000
Epoch 69/100
3/3 [=====] - 0s 59ms/step - loss: 2.2189 - accuracy: 0.1500 - val_loss: 2.2968 - val_accuracy: 0.1000
Epoch 70/100
3/3 [=====] - 0s 64ms/step - loss: 2.2072 - accuracy: 0.1750 - val_loss: 2.2155 - val_accuracy: 0.2500
Epoch 71/100
3/3 [=====] - 0s 65ms/step - loss: 2.2050 - accuracy: 0.1500 - val_loss: 2.2344 - val_accuracy: 0.1500
Epoch 72/100
3/3 [=====] - 0s 58ms/step - loss: 2.2165 - accuracy: 0.2000 - val_loss: 2.2650 - val_accuracy: 0.1000
Epoch 73/100
3/3 [=====] - 0s 58ms/step - loss: 2.2193 - accuracy: 0.1750 - val_loss: 2.2530 - val_accuracy: 0.2500
Epoch 74/100
3/3 [=====] - 0s 62ms/step - loss: 2.2157 - accuracy: 0.2125 - val_loss: 2.2484 - val_accuracy: 0.2000
Epoch 75/100
3/3 [=====] - 0s 55ms/step - loss: 2.2168 - accuracy: 0.2125 - val_loss: 2.2909 - val_accuracy: 0.1500
Epoch 76/100
3/3 [=====] - 0s 55ms/step - loss: 2.1884 - accuracy: 0.1500 - val_loss: 2.2732 - val_accuracy: 0.3000
Epoch 77/100
3/3 [=====] - 0s 61ms/step - loss: 2.2177 - accuracy: 0.2125 - val_loss: 2.2792 - val_accuracy: 0.1000
Epoch 78/100
```

```
3/3 [=====] - 0s 62ms/step - loss: 2.1972 - accuracy: 0.2125 - val_loss: 2.2682 - val_accuracy: 0.1000
Epoch 79/100
3/3 [=====] - 0s 57ms/step - loss: 2.1826 - accuracy: 0.1375 - val_loss: 2.3299 - val_accuracy: 0.0500
Epoch 80/100
3/3 [=====] - 0s 58ms/step - loss: 2.1853 - accuracy: 0.2125 - val_loss: 2.3142 - val_accuracy: 0.1500
Epoch 81/100
3/3 [=====] - 0s 62ms/step - loss: 2.1566 - accuracy: 0.2500 - val_loss: 2.3177 - val_accuracy: 0.1000
Epoch 82/100
3/3 [=====] - 0s 65ms/step - loss: 2.1889 - accuracy: 0.2375 - val_loss: 2.2166 - val_accuracy: 0.2500
Epoch 83/100
3/3 [=====] - 0s 59ms/step - loss: 2.1878 - accuracy: 0.2125 - val_loss: 2.2979 - val_accuracy: 0.1000
Epoch 84/100
3/3 [=====] - 0s 69ms/step - loss: 2.1612 - accuracy: 0.2125 - val_loss: 2.2936 - val_accuracy: 0.1500
Epoch 85/100
3/3 [=====] - 0s 184ms/step - loss: 2.2064 - accuracy: 0.2125 - val_loss: 2.2916 - val_accuracy: 0.1500
Epoch 86/100
3/3 [=====] - 0s 62ms/step - loss: 2.2178 - accuracy: 0.1500 - val_loss: 2.2843 - val_accuracy: 0.1000
Epoch 87/100
3/3 [=====] - 0s 63ms/step - loss: 2.1772 - accuracy: 0.2250 - val_loss: 2.2473 - val_accuracy: 0.2500
Epoch 88/100
3/3 [=====] - 0s 59ms/step - loss: 2.1876 - accuracy: 0.2250 - val_loss: 2.3148 - val_accuracy: 0.1000
Epoch 89/100
3/3 [=====] - 0s 67ms/step - loss: 2.2195 - accuracy: 0.1875 - val_loss: 2.2641 - val_accuracy: 0.2000
Epoch 90/100
3/3 [=====] - 0s 61ms/step - loss: 2.1635 - accuracy: 0.2250 - val_loss: 2.2403 - val_accuracy: 0.2000
Epoch 91/100
3/3 [=====] - 0s 64ms/step - loss: 2.1893 - accuracy: 0.1500 - val_loss: 2.2410 - val_accuracy: 0.1500
Epoch 92/100
3/3 [=====] - 0s 60ms/step - loss: 2.2128 - accuracy: 0.1125 - val_loss: 2.2952 - val_accuracy: 0.1500
Epoch 93/100
3/3 [=====] - 0s 65ms/step - loss: 2.2001 - accuracy: 0.1625 - val_loss: 2.2778 - val_accuracy: 0.1500
Epoch 94/100
```

```

3/3 [=====] - 0s 59ms/step - loss: 2.2179 - accuracy: 0.1375 - val_loss: 2.3069 - val_accuracy: 0.1500
Epoch 95/100
3/3 [=====] - 0s 65ms/step - loss: 2.2032 - accuracy: 0.1750 - val_loss: 2.2353 - val_accuracy: 0.0500
Epoch 96/100
3/3 [=====] - 0s 61ms/step - loss: 2.1876 - accuracy: 0.2125 - val_loss: 2.2568 - val_accuracy: 0.1000
Epoch 97/100
3/3 [=====] - 0s 66ms/step - loss: 2.1627 - accuracy: 0.2250 - val_loss: 2.2935 - val_accuracy: 0.1000
Epoch 98/100
3/3 [=====] - 0s 67ms/step - loss: 2.1826 - accuracy: 0.2000 - val_loss: 2.2174 - val_accuracy: 0.2000
Epoch 99/100
3/3 [=====] - 0s 58ms/step - loss: 2.1938 - accuracy: 0.2125 - val_loss: 2.2458 - val_accuracy: 0.2000
Epoch 100/100
3/3 [=====] - 0s 57ms/step - loss: 2.1718 - accuracy: 0.2250 - val_loss: 2.2231 - val_accuracy: 0.2500

<tensorflow.python.keras.callbacks.History at 0x7fc0180d8ac8>

```

This is not a great model, obviously, but just supposed to showcase how to use these iterators. It's more interesting to look at which kind of images are generated:

```

# here's a more "manual" example
for epoch in range(2):
    fig,axs = plt.subplots(4,3, figsize=(12,15))
    batch = 0
    for X_batch, y_batch in datagen.flow(X_train, y_train, batch_size=4):
        for i,img in enumerate(X_batch):
            axs[i,batch].imshow(img)
            axs[i,batch].axis("off")

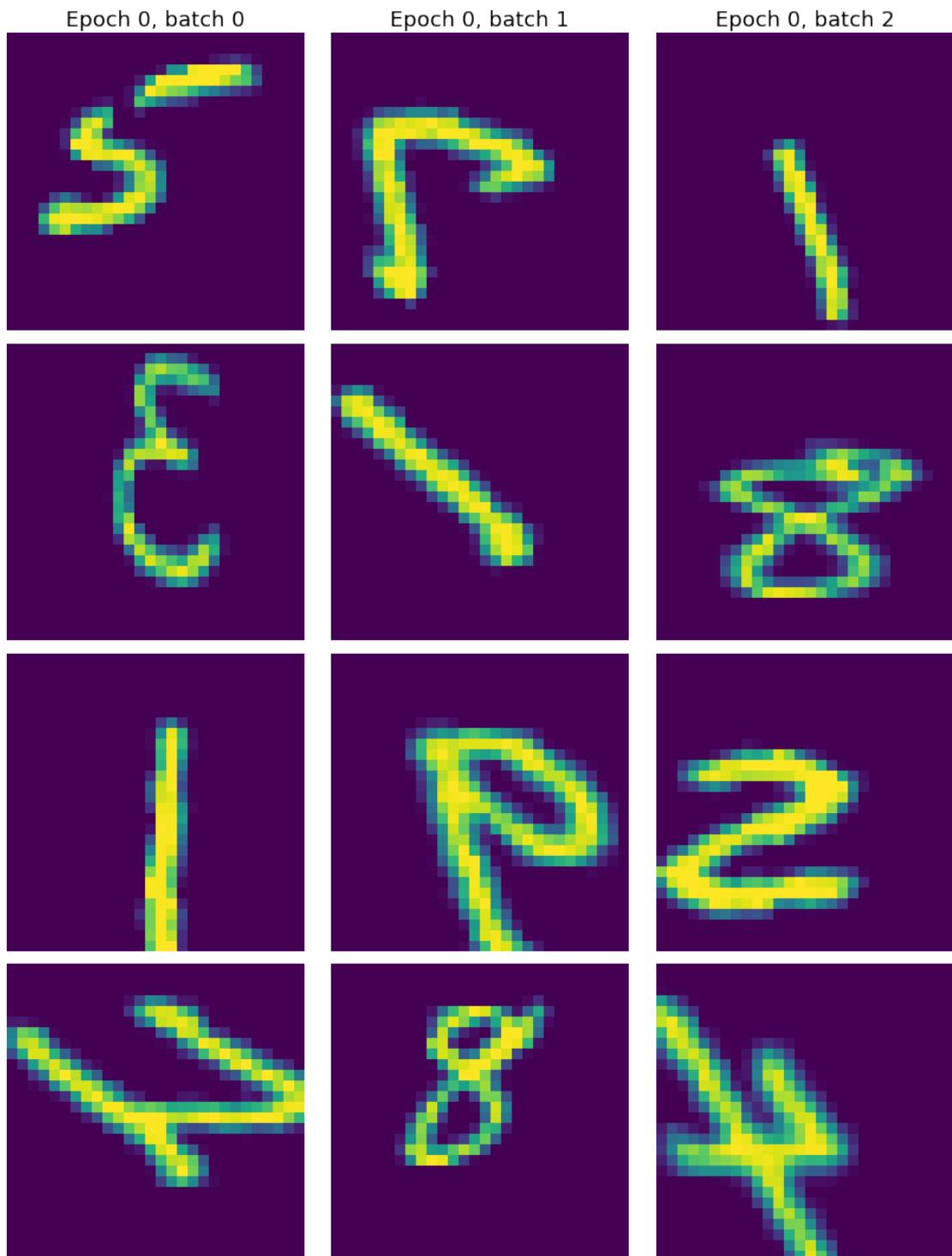
        axs[0,batch].set_title(f"Epoch {epoch}, batch {batch}", fontsize=18)

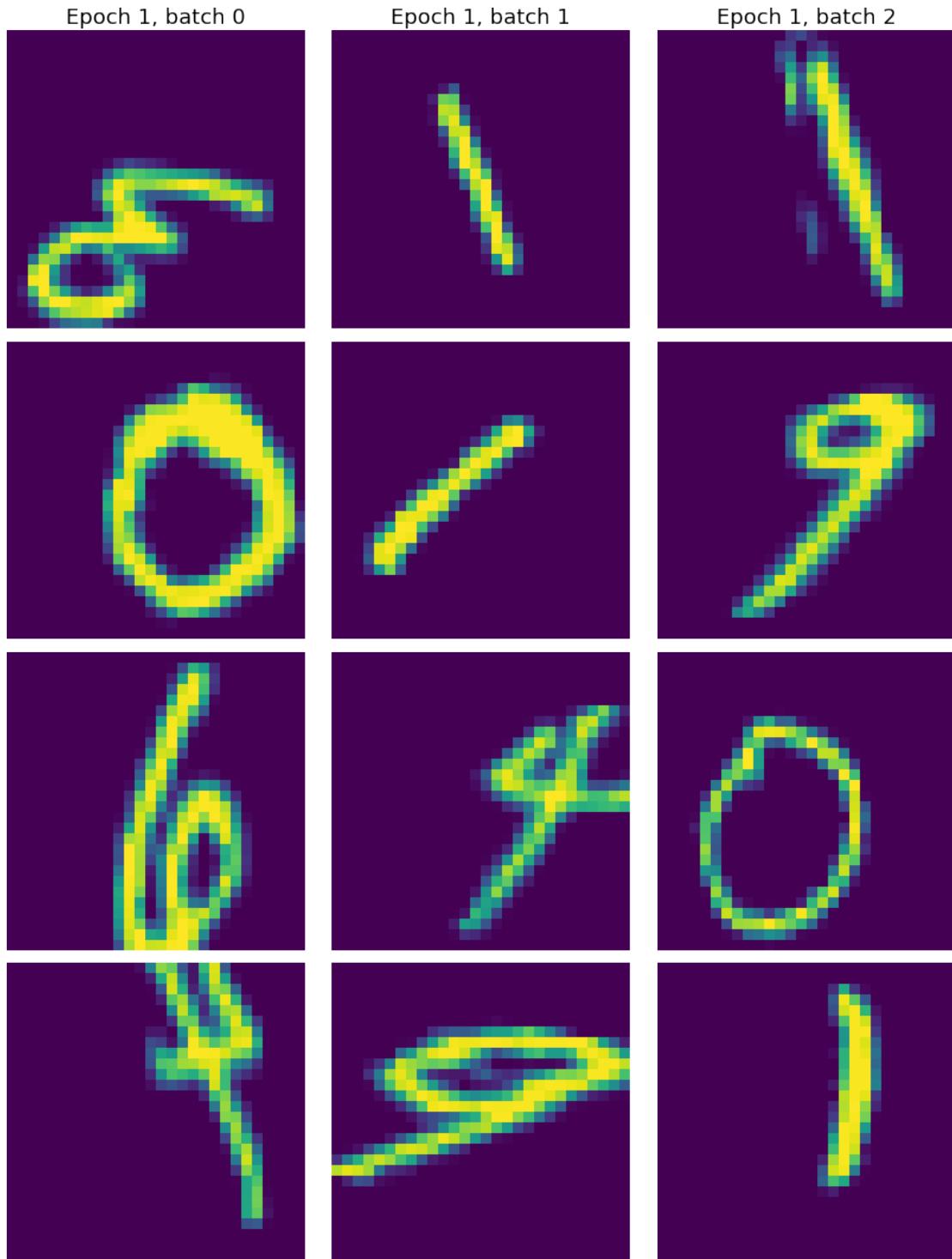
    plt.tight_layout()

    # since infinitely many examples are generated, we need to
    # manually break the loop here
    if batch == 2:
        break

    batch += 1

```





There are also specialized libraries for data augmentation, like `Augmentor`, `Albumentation`, `ImgAug` (parallelizable), or `AutoAugment` (non-free)/`DeepAugment`, which use ML techniques to automatically determine the best augmentation strategy for datasets. These offer even more ways to augment

images than the `keras` inbuilt options. For us, these internal options will suffice but for your own projects it may make sense to look at these options.

7.14.6 Relevant Data

Is data augmentation even useful when plenty of data is available? Yes, in several ways. Some of the techniques, like adding noise, have a regularizing effect and help the training process in general (see next lecture). Others create data in positions that may not have been originally recorded as data points, hence increasing generalizability.

Say for example that you're training a dog classifier, but for some reason all the dogs look to the left. If now a new data point is fed into the network, where a dog looks to the right, it will not recognize the image as a dog at all. Data augmentation therefore also helps creating **relevant** data for training, reducing *bias* problems in the dataset.

7.15 Visualizing Convolutional Neural Networks

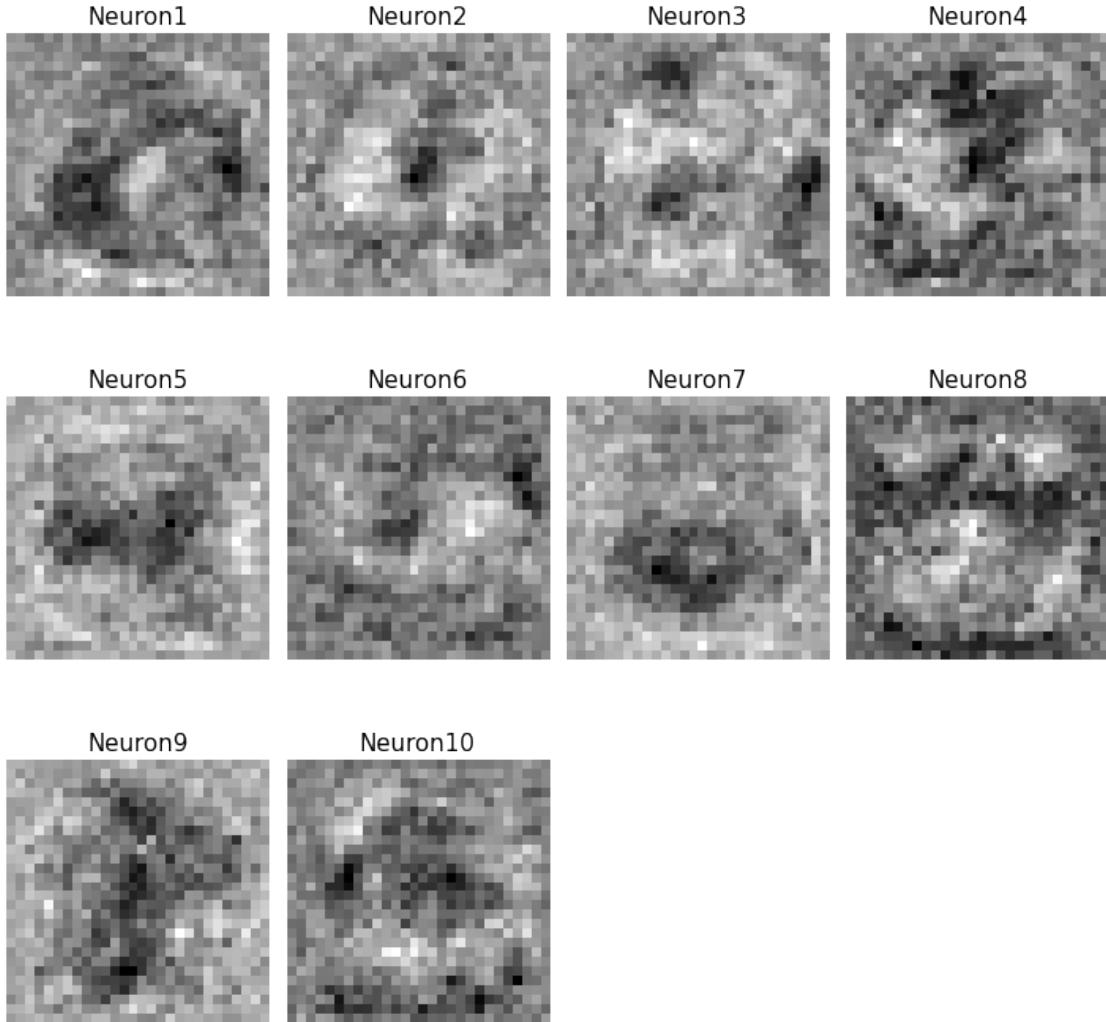
ANNs in general are often used as *black box* models that are trained for a specific task, but not very well investigated regarding their inner workings and “reasoning” for how to come to a certain conclusion about an image. In recent years “explainable AI” became a mainstream issue. It’s a wide field that tries to develop a foundation for analyzing ANNs in general, but it’s still not very far. A few things are already possible, and we’ll take a look at very few of the techniques, that are mostly used for getting an understanding of CNNs. For a good overview of more techniques, check out [Christopher Olah’s Blog](#), which offers super helpful and plenty of content regarding a lot of topics around CNNs.

There is no standard way of visualizing the concepts we'll use in this lesson, and various packages exist for this purpose. We'll use `tf-keras-vis` here, which seems to be the most popular one.

7.15.1 Saliency Maps

In assignment 02, we plotted the weights of the network to visualize a bunch of “filters”, that give us a hint at which parts of an image activate certain neurons the most. For standard ANNs, this is rather easy and straightforward. For CNNs, not so much thanks to their locality and complex layers.

```
from IPython.display import Image
Image("img/ann_saliency.png", width=900)
```



Saliency Maps offer a way to analyze regions of an image that a CNN deems noteworthy for deciding the class of an input. Roughly speaking, every pixel in a saliency map gets assigned a value that visualizes the pixelwise dependence of a class label \hat{y}_{ij}^c from the pixel value x_{ij} in the original image:

$$s_{ij} = \frac{\Delta \hat{y}_{ij}^c}{\Delta x_{ij}}$$

This is done by slightly changing the pixel values and recording where class labels change drastically in an image.

Saliency maps cannot completely explain why a CNN came to a certain conclusion, but they do provide some insight into which parts of an image were most important for the decision. For example, instead of identifying a leaf on an image, a CNN may actually “see” that the leaf is

attached to a tree and derive the information from that part of the image, instead of from the leaf itself. If that's the case, the model won't be able to identify leaves that are not on a tree anymore. You could even get more information than that in specific circumstances. Say for example you'd like to estimate the age of a tree by looking at a cross section of its trunk. From a well-trained CNN one would expect the saliency map to show the network's attention to be on the rings of the trunk, instead of, say, the background because all samples were taken in different labs.

Let's try this with the crack detection model (it was slightly modified for this, containing more Dense layers and dropout regularization in-between, also using one-hot-encoded output and softmax instead of a single output with sigmoid) from assignment 03 (loading takes some time):

```
from tensorflow.keras.models import load_model
from tensorflow.keras import backend as K

# since we defined a custom metric, we have to provide it to the model again
# it's not saved with the model
def recall_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    recall = true_positives / (possible_positives + K.epsilon())
    return recall

def precision_m(y_true, y_pred):
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    return precision

def f1_m(y_true, y_pred):
    precision = precision_m(y_true, y_pred)
    recall = recall_m(y_true, y_pred)
    return 2*((precision*recall)/(precision+recall+K.epsilon()))

model = load_model("/data/crack_detector", custom_objects={"f1_m": f1_m})
```

Loading a few example images:

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

num_samples = 3

image_generator = ImageDataGenerator(rescale=1/255)
train_data_gen = image_generator.flow_from_directory(directory="/data/
→Crack_data", classes=["Positive", "Negative"])

images, labels = [], []
for i in range(num_samples):
```

```

im, lab = next(train_data_gen)
images.append(im[0])
labels.append(lab[0][0])

images = np.array(images)
labels = np.array(labels)

fig,axs = plt.subplots(1,num_samples, figsize=(12,7))

for i,img in enumerate(images):
    axs[i].imshow(img)
    axs[i].axis("off")

```

Found 200 images belonging to 2 classes.



Two functions are necessary to provide for calculating the saliences. The “loss” provides the neuron from the classification to look out for, i.e. “[0]” if there is a crack, and “[1]” if there is none. This has to be done manually currently, but can be automated using the labels. The `model_modifier` removes the softmax activation and switches it for linear activation, since softmax rescales each output.

```

from tensorflow.keras.activations import linear

# output means the output of the model here, i.e. the prediction
# in this modified model, it's a 2D vector, where the first element indicates
# a crack and the second element indicates no crack
# the shape of output is (3, 2), where 3 is the number of samples, and 2 is the
# number of classes detected by the model
def loss(output):
    return (output[0][1], output[1][0], output[2][0])

# if softmax was used, this function is necessary to strip off the softmax
# activation and exchange it for linear
def model_modifier(m):
    m.layers[-1].activation = linear

```

```
    return m
```

The following code creates a raw, standard saliency map for the input images above. The `%%time` magic command provides an automatic timing analysis of the code executed in the cell. It's not necessary, but gives a feeling for how long it takes to traverse the network.

```
%%time
from tensorflow.keras import backend as K
from tf_keras_vis.saliency import Saliency
from tf_keras_vis.utils import normalize

# helper function for plotting results
def plot_saliency(saliency_map, num_samples, overlay=False):
    fig, axs = plt.subplots(1,num_samples, figsize=(12,7))

    for i in range(num_samples):
        if overlay:
            axs[i].imshow(images[i])
        axs[i].imshow(saliency_map[i], cmap='jet', alpha=0.5 if overlay else 1)
        axs[i].axis("off")

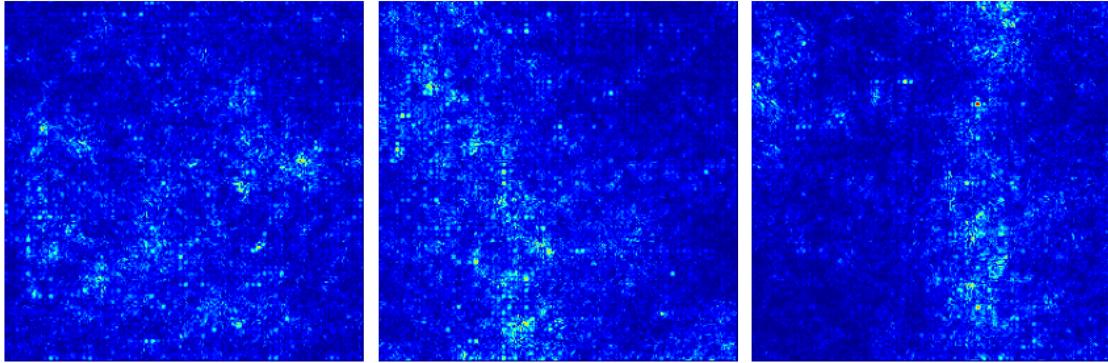
    plt.tight_layout()
    #plt.show()

# creates a saliency object. clone=True creates a copy of the model,
# so that the original model is not modified by the model_modifier
saliency = Saliency(model,
                     model_modifier=model_modifier,
                     clone=False)

# generate saliency map
saliency_map = saliency(loss, images)
# normalize all values
saliency_map = normalize(saliency_map)

plot_saliency(saliency_map, 3)
```

CPU times: user 13.3 s, sys: 265 ms, total: 13.6 s
Wall time: 11.5 s



The raw saliency is quite chunky and disconnected, but already provides some information about which parts of the image were important for the networks's decision. To smooth the results and make them more clear and focused, we can use the *SmoothGrad* method. It feeds the same image multiple times into the network but with added Gaussian noise each time. Counterintuitively, adding noise to the input reduces noise in the output, because the saliency is calculated as an average over the sampled outputs. This takes much more time than the standard saliency maps.

```
%%time

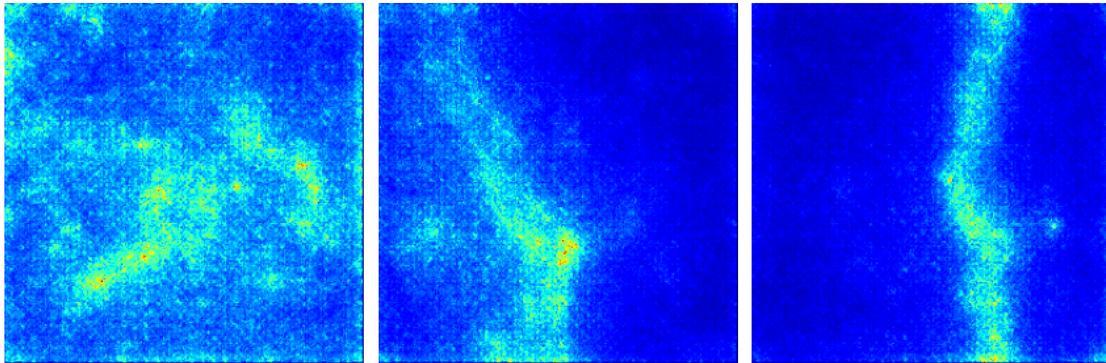
# just like before
saliency_sg = Saliency(model,
                       model_modifier=model_modifier,
                       clone=False)

# two new parameters create noisy inputs for SmoothGrad
# smooth_samples determines the number of times the input image
# is fed into the network, and smooth_noise determines the noise level
saliency_map_sg = saliency(loss,
                           images,
                           smooth_samples=20,
                           smooth_noise=0.20)

saliency_map_sg = normalize(saliency_map_sg)

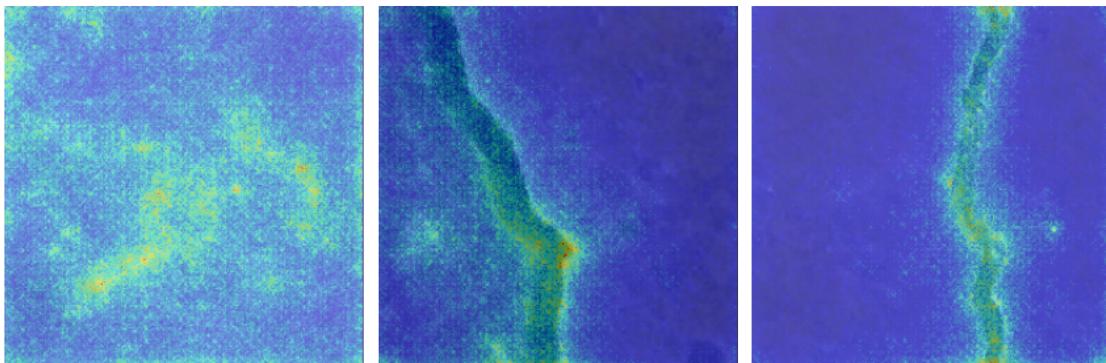
plot_saliency(saliency_map_sg, 3)
```

CPU times: user 1min 27s, sys: 4.41 s, total: 1min 31s
Wall time: 51.2 s



Let's overlay the result with the original images to see if the maps make sense:

```
plot_saliency(saliency_map_sg, 3, True)
```



`tf_keras_vis` offer many more visualization methods, e.g. `GradCAM` and `ScoreCAM`, which provide *class activation maps*. These are class-specific saliencies, calculated as the weighted sum

$$M_c(x, y) = \sum_k w_k^c f_k(x, y)$$

where M_c is the class activation map. $f_k(x, y)$ is the value at position (x, y) of the k th feature map after the last pooling layer, and w_k^c is the weight connecting the k feature maps after the last pooling layer to the class neurons in the one-hot-encoded output layer. The class activation map shows which inputs would approximately maximize a certain class output.

```
%%time
from tf_keras_vis.scorecam import ScoreCAM

# create ScoreCAM object
scorecam = ScoreCAM(model,
```

```

        model_modifier=model_modifier,
        clone=False)

# generate heatmap with ScoreCAM
# experience shows that the most important feature maps after the last pooling
# layer are those with the largest variances in the weights
# the penultimate_layer argument determines how many of the highest-variance
# feature maps to use for approximating the full class activation map
cam = scorecam(loss,
                 images,
                 penultimate_layer=-1,
                 max_N=10
                )
cam = normalize(cam)

```

CPU times: user 40.3 s, sys: 1.38 s, total: 41.7 s
Wall time: 19.7 s

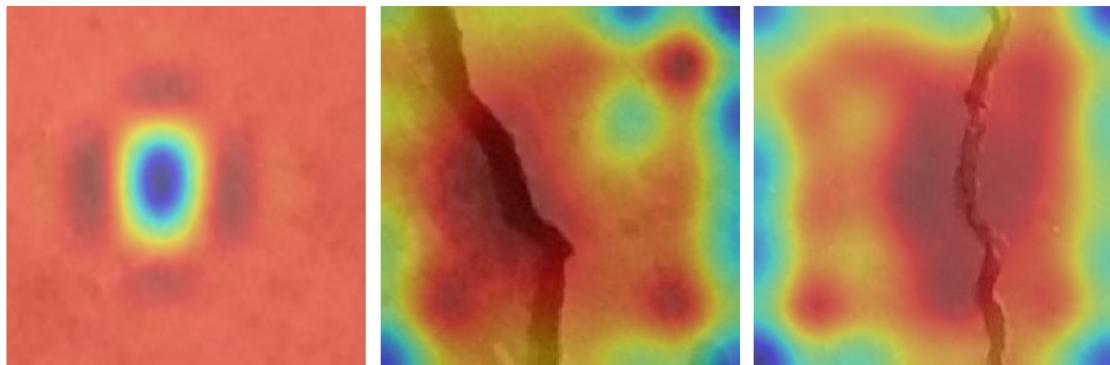
And overlaying this with the original images:

```

fig, axs = plt.subplots(1,num_samples, figsize=(12,7))

for i in range(num_samples):
    heatmap = np.uint8(cm.jet_r(cam[i])[..., :3] * 255)
    axs[i].imshow(images[i])
    axs[i].imshow(heatmap, cmap='jet', alpha=0.5)
    axs[i].axis("off")
plt.tight_layout()
plt.show()

```



7.15.2 Dense Layer ActivationMaximization

Similarly, we can plot these activation maps that maximize the neuron activations of the dense layers in our network:

```

from tensorflow.random import uniform
from tf_keras_vis.activation_maximization import ActivationMaximization

activation_maximization = ActivationMaximization(model,
                                                    model_modifier=model_modifier,
                                                    clone=False)

%%time
from tf_keras_vis.utils.callbacks import Print

# we need to define random seed inputs, used for calculating the activations
# the shape should be (samples, height, width, channels), the rest are
# the pixel values
seed_input = uniform((3, 256, 256, 3), 0, 255)

# Do 500 iterations and Generate an optimizing animation
activations = activation_maximization(loss,
                                         seed_input=seed_input,
                                         steps=1024,
                                         callbacks=[Print(interval=32)])

```

Steps: 032 Losses: [[207.7827606201172, 22.71770477294922, 23.66503143310547]], Regularizations: [('TotalVariation', 35.543121337890625), ('L2Norm', 0.01293756254017353)]
 Steps: 064 Losses: [[374.5107421875, 12.142592430114746, 28.300365447998047]], Regularizations: [('TotalVariation', 34.307334899902344), ('L2Norm', 0.012914523482322693)]
 Steps: 096 Losses: [[505.4567565917969, 36.633968353271484, 48.526947021484375]], Regularizations: [('TotalVariation', 34.99290466308594), ('L2Norm', 0.012874385342001915)]
 Steps: 128 Losses: [[573.2018432617188, 31.37478256225586, 47.91239929199219]], Regularizations: [('TotalVariation', 35.05389404296875), ('L2Norm', 0.012839166447520256)]
 Steps: 160 Losses: [[573.9132080078125, 48.34168243408203, 45.382179260253906]], Regularizations: [('TotalVariation', 34.574615478515625), ('L2Norm', 0.012802090495824814)]
 Steps: 192 Losses: [[740.5380249023438, 34.533233642578125, 53.38788604736328]], Regularizations: [('TotalVariation', 34.42466354370117), ('L2Norm', 0.012768782675266266)]
 Steps: 224 Losses: [[588.017578125, 45.39036178588867, 8.432276725769043]], Regularizations: [('TotalVariation', 34.283390045166016), ('L2Norm', 0.012732259929180145)]
 Steps: 256 Losses: [[706.9357299804688, 42.22456359863281, 44.4503059387207]], Regularizations: [('TotalVariation', 33.858184814453125), ('L2Norm', 0.012697350233793259)]
 Steps: 288 Losses: [[631.39404296875, 40.37201690673828, 46.36805725097656]], Regularizations: [('TotalVariation',

```

33.962257385253906), ('L2Norm', 0.012666814960539341)]
Steps: 320      Losses: [[635.7527465820312, 47.76375961303711,
33.9408073425293]],      Regularizations: [('TotalVariation', 37.88405227661133),
('L2Norm', 0.01262736413627863)]
Steps: 352      Losses: [[536.3544921875, 51.87144470214844,
58.84349822998047]],      Regularizations: [('TotalVariation',
33.17377853393555), ('L2Norm', 0.012597757391631603)]
Steps: 384      Losses: [[724.1409301757812, 60.85784912109375,
49.44327163696289]],      Regularizations: [('TotalVariation', 33.43707275390625),
('L2Norm', 0.0125576788559556)]
Steps: 416      Losses: [[795.671630859375, 69.33103942871094,
29.854598999023438]],      Regularizations: [('TotalVariation',
33.417659759521484), ('L2Norm', 0.012512262910604477)]
Steps: 448      Losses: [[781.5609741210938, 82.93732452392578,
88.21493530273438]],      Regularizations: [('TotalVariation',
33.826812744140625), ('L2Norm', 0.012465167790651321)]
Steps: 480      Losses: [[806.391357421875, 83.32705688476562,
72.92233276367188]],      Regularizations: [('TotalVariation',
34.11659622192383), ('L2Norm', 0.012416478246450424)]
Steps: 512      Losses: [[692.92529296875, 89.17378234863281,
81.7132568359375]],      Regularizations: [('TotalVariation',
34.055206298828125), ('L2Norm', 0.012393542565405369)]
Steps: 544      Losses: [[700.9462890625, 69.55590057373047,
87.51813507080078]],      Regularizations: [('TotalVariation',
33.873756408691406), ('L2Norm', 0.012356076389551163)]
Steps: 576      Losses: [[706.4295043945312, 126.99260711669922,
74.84227752685547]],      Regularizations: [('TotalVariation', 33.5162353515625),
('L2Norm', 0.012314063496887684)]
Steps: 608      Losses: [[803.4898071289062, 188.34378051757812,
115.66326141357422]],      Regularizations: [('TotalVariation', 35.41212463378906),
('L2Norm', 0.01225835457444191)]
Steps: 640      Losses: [[685.532958984375, 111.8447265625,
137.33621215820312]],      Regularizations: [('TotalVariation',
37.947166442871094), ('L2Norm', 0.012208173051476479)]
Steps: 672      Losses: [[895.0794067382812, 209.82827758789062,
171.71401977539062]],      Regularizations: [('TotalVariation', 35.18159484863281),
('L2Norm', 0.012158291414380074)]
Steps: 704      Losses: [[823.501708984375, 211.46365356445312,
160.96905517578125]],      Regularizations: [('TotalVariation', 37.58407974243164),
('L2Norm', 0.012103999964892864)]
Steps: 736      Losses: [[954.4452514648438, 233.90159606933594,
189.28115844726562]],      Regularizations: [('TotalVariation', 35.211036682128906),
('L2Norm', 0.012049548327922821)]
Steps: 768      Losses: [[1000.0418090820312, 216.3817138671875,
238.2342529296875]],      Regularizations: [('TotalVariation', 36.03767776489258),
('L2Norm', 0.011993806809186935)]
Steps: 800      Losses: [[778.4735107421875, 252.789306640625,
203.5623016357422]],      Regularizations: [('TotalVariation',

```

```

35.688751220703125), ('L2Norm', 0.011960520409047604)]
Steps: 832      Losses: [[1033.3634033203125, 263.18463134765625,
244.20680236816406]], Regularizations: [('TotalVariation', 36.93369674682617),
('L2Norm', 0.011915473267436028)]
Steps: 864      Losses: [[856.5927734375, 287.8738098144531,
236.85260009765625]], Regularizations: [('TotalVariation',
35.07530212402344), ('L2Norm', 0.011875276453793049)]
Steps: 896      Losses: [[908.4691162109375, 210.12042236328125,
239.30809020996094]], Regularizations: [('TotalVariation', 35.00328826904297),
('L2Norm', 0.011827466078102589)]
Steps: 928      Losses: [[793.8474731445312, 227.6829833984375,
232.35682678222656]], Regularizations: [('TotalVariation', 35.78455352783203),
('L2Norm', 0.01178510021418333)]
Steps: 960      Losses: [[1046.0330810546875, 309.98480224609375,
275.6339111328125]], Regularizations: [('TotalVariation', 35.25912857055664),
('L2Norm', 0.011734514497220516)]
Steps: 992      Losses: [[867.1292114257812, 266.7157287597656,
269.9614562988281]], Regularizations: [('TotalVariation', 34.89230728149414),
('L2Norm', 0.01168928574770689)]
Steps: 1024     Losses: [[959.529541015625, 167.28170776367188,
224.65403747558594]], Regularizations: [('TotalVariation', 35.41402816772461),
('L2Norm', 0.011641942895948887)]

```

□
→-----

NameError
↳last)

<timed exec> in <module>

<timed exec> in <listcomp>(.0)

NameError: name 'np' is not defined

```

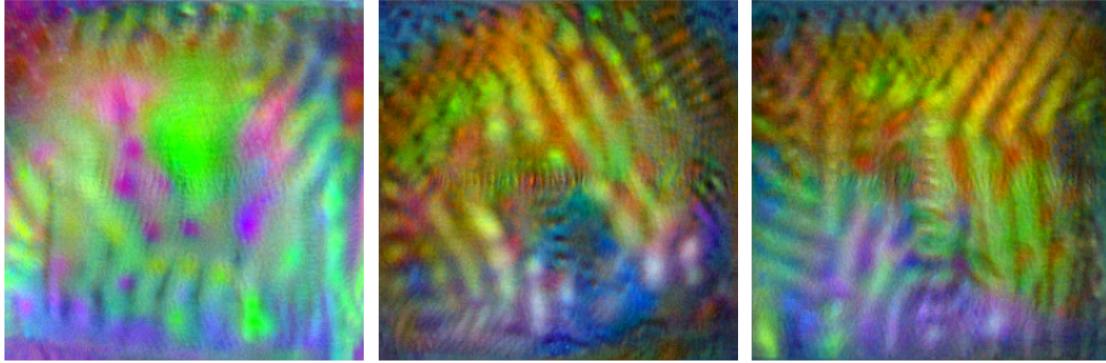
acti_ims = [activation.astype(np.uint8) for activation in activations]

fig, axs = plt.subplots(1,num_samples, figsize=(12,7))

for i in range(num_samples):
    axs[i].imshow(acti_ims[i])
    axs[i].axis("off")

```

```
plt.tight_layout()
plt.show()
```



These activation maximizations don't tell us much at all about how the CNN determines classes, but when this is done for full NASNetMobile or VGG-19 or similar network, the images will depict typical shapes found on the objects, like eyes or head shapes. We do again see that the network tends to look towards the middle of images and that the concept of "crack" is abstracted as a lot of ripples on the right and middle image, while the "no-crack" concept contains fewer ripples.

The activation maximization can also be plotted for filters of convolutional layers and will then depict the features they detect. The further to the end of a network the convolutional layers are, the higher level and more abstract the filter activation maps will become.

7.16 Semantic Segmentation

7.16.1 Bitmasks

We introduced the chapter about CNNs with semantic segmentation as an example application. The idea was to mark pixels in an image with a fixed color, if they belong to an object in the original image. This is effectively a pixelwise classification. There are two options for semantic segmentation, either the standard one, that simply classifies objects in an image, or *instance segmentation*, which marks each *instance* of an object differently. I.e., if there are several cats in an image, the standard method will mark all of the pixels belonging to cats with the same color. In instance segmentation, the goal would be to color each cat differently.

The desired pixelwise classification images are called **bitmasks**. These are the ground truths y_i of a segmentation problem. See for example an image sample from the [Pascal VOC2012 dataset](#) consisting of 9963 images with 20 different classe bitmaps:

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

original = mpimg.imread('img/voc2012_airplane_original.jpg')
bitmask = mpimg.imread('img/voc2012_airplane_objects.png')
instances = mpimg.imread('img/voc2012_airplane_instances.png')
```

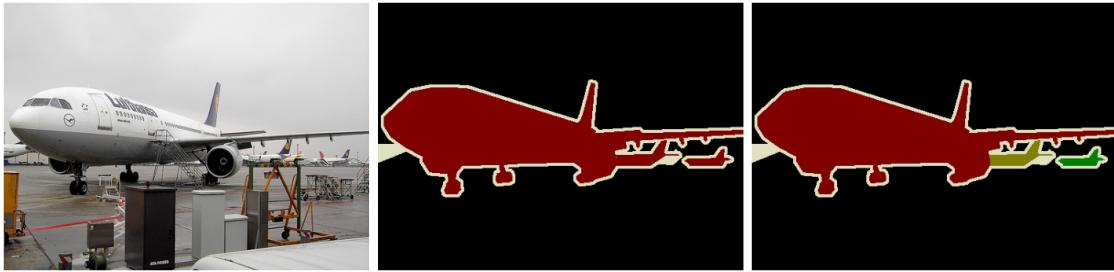
```

fig, ax = plt.subplots(1,3, figsize=(20,7))

ax[0].imshow(original)
ax[1].imshow(bitmask)
ax[2].imshow(instances)
ax[0].axis("off")
ax[1].axis("off");
ax[2].axis("off");

plt.tight_layout()

```



The left is the original image, the middle bitmask marks all the planes in the original image and the right bitmask colors all instances of planes differently.

7.16.2 Segmentation Models

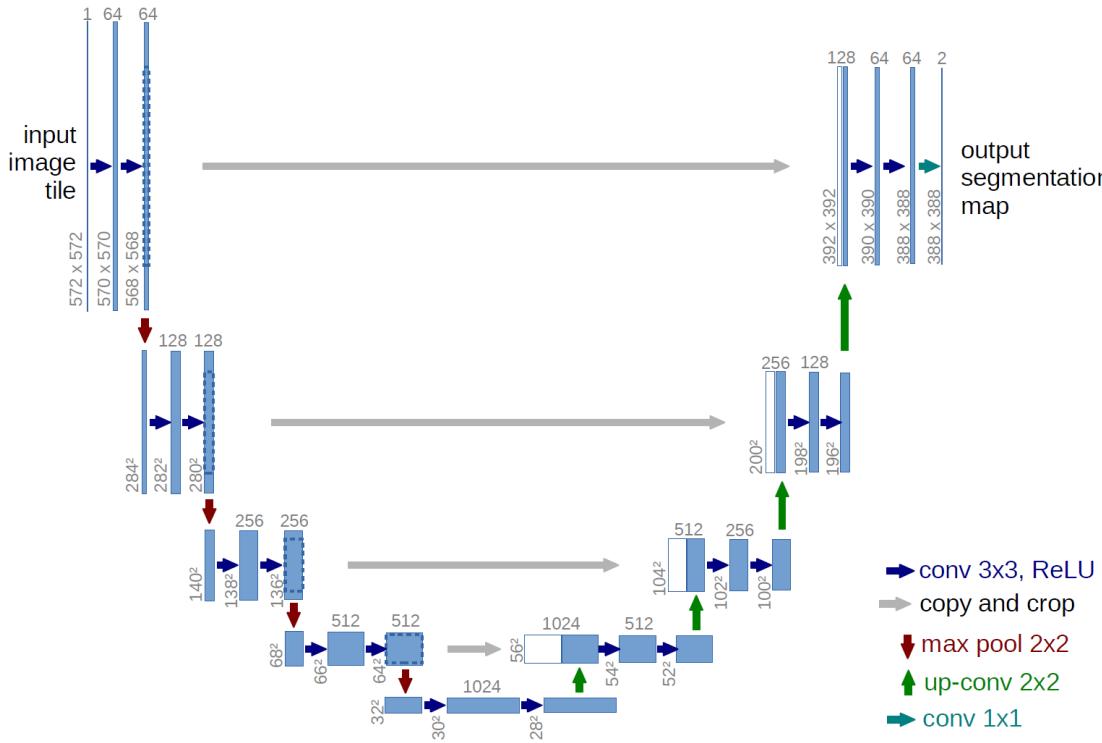
Creating these bitmasks is a tedious process, but there are some libraries and GUIs that can help the process, like the [Sefexa image segmentation tool](#), which also helps analyzing images, [Facebook's DeepMask](#), which proposes objects in an image to help build masks, [openCV](#), which is a commonly used Python module for image-based tasks in general and offers especially many classical segmentation algorithms, and many many more. [MISCCNN](#) is a framework engineered for medical image analysis and also contains models, but can surely be used for other tasks as well.

Several models have been proposed to handle this kind of mapping. Almost all of them have the following idea in common:

- use a CNN to compress the information contained in an image into a relatively low-dimensional feature vector
- create the bitmask starting from this feature vector by upscaling

This kind of structure is usually called *encoder-decoder* architecture, of which the defining property is a so-called *bottleneck layer* somewhere in the middle of the model. This captures the essence of the data in a *latent space*, that is the feature vector mentioned above. We'll talk about these concepts in the lecture about generative models. Perhaps the most widely known segmentation model is the **U-Net architecture**, which looks like this (taken from the original [paper](#)):

```
from IPython.display import Image
Image("img/u-net-architecture.png", width=1000)
```



The encoder part consists of blocks with two convolutions and one maxpooling operation, downscaling the input into a bottleneck fully-connected layer with 1024 neurons. The decoder part upscales from here with a transpose convolution and two convolutions. The gray lines connecting the encoder and decoder blocks copy over feature maps from the encoder to the decoder part (depicted as white boxes), helping to include some information from the original image into the generation of the bitmasks.

There are some other models, such as using a random forest classifier after a CNN, FastFCN, which uses Joint Pyramid Upsampling instead of dilated transpose convolutions to save computational time, Gated-SCNN, which uses two “stream” CNNs to work with segmentation and shape information, and many many more. Here, we’ll concentrate on the U-Net.

Image segmentation can be used in various fields related to engineering, e.g. in material science for identifying microstrutures, or generating mechanical models from CT scans (different materials require different material laws), or for *generating* microstructures from bitmasks.

7.16.3 Example

The easiest way to build a U-Net is to use keras *functional layers*, which we already used in assignment 03.

```

from tensorflow.keras.backend import clear_session
from tensorflow.keras.layers import Conv2D, SeparableConv2D, Conv2DTranspose, ↴
    ↪MaxPooling2D, UpSampling2D, BatchNormalization, Activation, add
from tensorflow.keras import Model, Input

clear_session()

def get_model(img_size, num_classes, activation):
    inputs = Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = BatchNormalization()(x)
    x = Activation(activation)(x)

    previous_block_activation = x # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = Activation(activation)(x)
        x = SeparableConv2D(filters, 3, padding="same")(x)
        x = BatchNormalization()(x)

        x = Activation(activation)(x)
        x = SeparableConv2D(filters, 3, padding="same")(x)
        x = BatchNormalization()(x)

        x = MaxPooling2D(3, strides=2, padding="same")(x)

    # Project residual
    residual = Conv2D(filters, 1, strides=2, padding="same")(
        previous_block_activation
    )
    x = add([x, residual]) # Add back residual
    previous_block_activation = x # Set aside next residual

    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = Activation(activation)(x)
        x = Conv2DTranspose(filters, 3, padding="same")(x)
        x = BatchNormalization()(x)

        x = Activation(activation)(x)

```

```

x = Conv2DTranspose(filters, 3, padding="same")(x)
x = BatchNormalization()(x)

x = UpSampling2D(2)(x)

# Project residual
residual = UpSampling2D(2)(previous_block_activation)
residual = Conv2D(filters, 1, padding="same")(residual)
x = add([x, residual]) # Add back residual
previous_block_activation = x # Set aside next residual

# Add a per-pixel classification layer
outputs = Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

# Define the model
model = Model(inputs, outputs)

return model

# Build model
model = get_model((256,256), 2, "selu")

```

The model description is quite large:

```
model.summary()
```

Model: "model"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 256, 256, 3) 0]		
conv2d (Conv2D)	(None, 128, 128, 32) 896		input_1[0] [0]
batch_normalization (BatchNorma	(None, 128, 128, 32) 128		conv2d[0] [0]
activation (Activation)	(None, 128, 128, 32) 0		
batch_normalization[0] [0]			
activation_1 (Activation)	(None, 128, 128, 32) 0		

```
activation[0] [0]
-----
separable_conv2d (SeparableConv (None, 128, 128, 64) 2400
activation_1[0] [0]
-----
batch_normalization_1 (BatchNor (None, 128, 128, 64) 256
separable_conv2d[0] [0]
-----
activation_2 (Activation)      (None, 128, 128, 64) 0
batch_normalization_1[0] [0]
-----
separable_conv2d_1 (SeparableCo (None, 128, 128, 64) 4736
activation_2[0] [0]
-----
batch_normalization_2 (BatchNor (None, 128, 128, 64) 256
separable_conv2d_1[0] [0]
-----
max_pooling2d (MaxPooling2D)   (None, 64, 64, 64)  0
batch_normalization_2[0] [0]
-----
conv2d_1 (Conv2D)             (None, 64, 64, 64)  2112
activation[0] [0]
-----
add (Add)                    (None, 64, 64, 64)  0
max_pooling2d[0] [0]          conv2d_1[0] [0]
-----
activation_3 (Activation)     (None, 64, 64, 64)  0           add[0] [0]
-----
separable_conv2d_2 (SeparableCo (None, 64, 64, 128) 8896
activation_3[0] [0]
-----
batch_normalization_3 (BatchNor (None, 64, 64, 128) 512
separable_conv2d_2[0] [0]
-----
activation_4 (Activation)     (None, 64, 64, 128)  0
```

```
batch_normalization_3[0] [0]
-----
separable_conv2d_3 (SeparableCo (None, 64, 64, 128) 17664
activation_4[0] [0]
-----
batch_normalization_4 (BatchNor (None, 64, 64, 128) 512
separable_conv2d_3[0] [0]
-----
max_pooling2d_1 (MaxPooling2D) (None, 32, 32, 128) 0
batch_normalization_4[0] [0]
-----
conv2d_2 (Conv2D) (None, 32, 32, 128) 8320 add[0] [0]
-----
add_1 (Add) (None, 32, 32, 128) 0
max_pooling2d_1[0] [0]
conv2d_2[0] [0]
-----
activation_5 (Activation) (None, 32, 32, 128) 0 add_1[0] [0]
-----
separable_conv2d_4 (SeparableCo (None, 32, 32, 256) 34176
activation_5[0] [0]
-----
batch_normalization_5 (BatchNor (None, 32, 32, 256) 1024
separable_conv2d_4[0] [0]
-----
activation_6 (Activation) (None, 32, 32, 256) 0
batch_normalization_5[0] [0]
-----
separable_conv2d_5 (SeparableCo (None, 32, 32, 256) 68096
activation_6[0] [0]
-----
batch_normalization_6 (BatchNor (None, 32, 32, 256) 1024
separable_conv2d_5[0] [0]
-----
max_pooling2d_2 (MaxPooling2D) (None, 16, 16, 256) 0
batch_normalization_6[0] [0]
```

```
-----  
conv2d_3 (Conv2D)           (None, 16, 16, 256) 33024      add_1[0] [0]  
-----  
add_2 (Add)                 (None, 16, 16, 256) 0  
max_pooling2d_2[0] [0]          conv2d_3[0] [0]  
-----  
activation_7 (Activation)    (None, 16, 16, 256) 0      add_2[0] [0]  
-----  
conv2d_transpose (Conv2DTranspo (None, 16, 16, 256) 590080  
activation_7[0] [0]  
-----  
batch_normalization_7 (BatchNor (None, 16, 16, 256) 1024  
conv2d_transpose[0] [0]  
-----  
activation_8 (Activation)    (None, 16, 16, 256) 0  
batch_normalization_7[0] [0]  
-----  
conv2d_transpose_1 (Conv2DTrans (None, 16, 16, 256) 590080  
activation_8[0] [0]  
-----  
batch_normalization_8 (BatchNor (None, 16, 16, 256) 1024  
conv2d_transpose_1[0] [0]  
-----  
up_sampling2d_1 (UpSampling2D) (None, 32, 32, 256) 0      add_2[0] [0]  
-----  
up_sampling2d (UpSampling2D)   (None, 32, 32, 256) 0  
batch_normalization_8[0] [0]  
-----  
conv2d_4 (Conv2D)           (None, 32, 32, 256) 65792  
up_sampling2d_1[0] [0]  
-----  
add_3 (Add)                 (None, 32, 32, 256) 0  
up_sampling2d[0] [0]          conv2d_4[0] [0]  
-----
```

```
-----  
activation_9 (Activation)      (None, 32, 32, 256)  0           add_3[0] [0]
```

```
-----  
conv2d_transpose_2 (Conv2DTrans (None, 32, 32, 128)  295040  
activation_9[0] [0]
```

```
-----  
batch_normalization_9 (BatchNor (None, 32, 32, 128)  512  
conv2d_transpose_2[0] [0]
```

```
-----  
activation_10 (Activation)     (None, 32, 32, 128)  0  
batch_normalization_9[0] [0]
```

```
-----  
conv2d_transpose_3 (Conv2DTrans (None, 32, 32, 128)  147584  
activation_10[0] [0]
```

```
-----  
batch_normalization_10 (BatchNo (None, 32, 32, 128)  512  
conv2d_transpose_3[0] [0]
```

```
-----  
up_sampling2d_3 (UpSampling2D) (None, 64, 64, 256)  0           add_3[0] [0]
```

```
-----  
up_sampling2d_2 (UpSampling2D) (None, 64, 64, 128)  0  
batch_normalization_10[0] [0]
```

```
-----  
conv2d_5 (Conv2D)             (None, 64, 64, 128)  32896  
up_sampling2d_3[0] [0]
```

```
-----  
add_4 (Add)                  (None, 64, 64, 128)  0  
up_sampling2d_2[0] [0]          conv2d_5[0] [0]
```

```
-----  
activation_11 (Activation)    (None, 64, 64, 128)  0           add_4[0] [0]
```

```
-----  
conv2d_transpose_4 (Conv2DTrans (None, 64, 64, 64)   73792  
activation_11[0] [0]
```

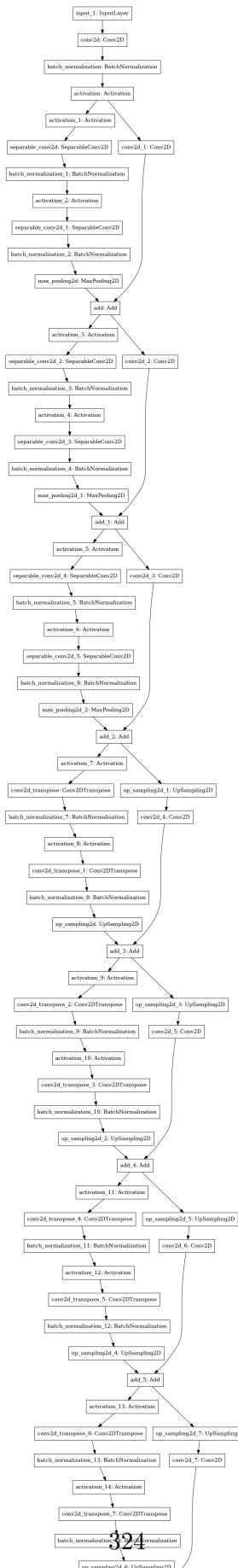
```
-----  
batch_normalization_11 (BatchNo (None, 64, 64, 64)   256
```

```
conv2d_transpose_4[0] [0]
-----
activation_12 (Activation)      (None, 64, 64, 64)  0
batch_normalization_11[0] [0]
-----
conv2d_transpose_5 (Conv2DTrans (None, 64, 64, 64)  36928
activation_12[0] [0]
-----
batch_normalization_12 (BatchNo (None, 64, 64, 64)  256
conv2d_transpose_5[0] [0]
-----
up_sampling2d_5 (UpSampling2D)  (None, 128, 128, 128 0           add_4[0] [0]
-----
up_sampling2d_4 (UpSampling2D)  (None, 128, 128, 64) 0
batch_normalization_12[0] [0]
-----
conv2d_6 (Conv2D)              (None, 128, 128, 64) 8256
up_sampling2d_5[0] [0]
-----
add_5 (Add)                   (None, 128, 128, 64) 0
up_sampling2d_4[0] [0]          conv2d_6[0] [0]
-----
activation_13 (Activation)    (None, 128, 128, 64) 0           add_5[0] [0]
-----
conv2d_transpose_6 (Conv2DTrans (None, 128, 128, 32) 18464
activation_13[0] [0]
-----
batch_normalization_13 (BatchNo (None, 128, 128, 32) 128
conv2d_transpose_6[0] [0]
-----
activation_14 (Activation)    (None, 128, 128, 32) 0
batch_normalization_13[0] [0]
-----
conv2d_transpose_7 (Conv2DTrans (None, 128, 128, 32) 9248
activation_14[0] [0]
```

```
-----  
batch_normalization_14 (BatchNo (None, 128, 128, 32) 128  
conv2d_transpose_7[0] [0]  
  
-----  
up_sampling2d_7 (UpSampling2D) (None, 256, 256, 64) 0 add_5[0] [0]  
  
-----  
up_sampling2d_6 (UpSampling2D) (None, 256, 256, 32) 0  
batch_normalization_14[0] [0]  
  
-----  
conv2d_7 (Conv2D) (None, 256, 256, 32) 2080  
up_sampling2d_7[0] [0]  
  
-----  
add_6 (Add) (None, 256, 256, 32) 0  
up_sampling2d_6[0] [0] conv2d_7[0] [0]  
  
-----  
conv2d_8 (Conv2D) (None, 256, 256, 2) 578 add_6[0] [0]  
=====  
=====  
Total params: 2,058,690  
Trainable params: 2,054,914  
Non-trainable params: 3,776
```

A perhaps slightly better overview of the connectivity can be produced with the `plot_model` function from keras:

```
from tensorflow.keras.utils import plot_model  
  
plot_model(model, show_shapes=False, show_dtype=False,  
           show_layer_names=True, expand_nested=False)
```



This is a large model. It can now be trained as usual, by calling `model.fit()` with appropriate data, and analyzed just like we did before. We'll see an example in the assignment today.

7.16.4 Segmentation Loss Functions

As always, there is a variety of loss functions available to choose from when performing image segmentation tasks, each with a focus on different tasks.

Focal Loss This is effectively a downweighting of categorical cross entropy. The problem with using simple cross entropy is that it makes the model very confident on its predictions, meaning that when a prediction probability is around, say, 0.8 while the ground truth is 1, the loss will still be rather high. When the predicted probability is a bit lower, say, 0.6, then the loss will become much higher compared to before. This tips the model towards only predicting things it's very confident about. The **focal loss** rescales the loss, making the model more risk-taking in its predictions:

$$L_{\text{FL}}(p_t) = -\alpha_t(1 - p_t)^\gamma \log(p_t)$$

```
import numpy as np
import matplotlib.pyplot as plt

def cr(p_t):
    return -np.log(p_t)

def fl(p_t, , ):
    return -*(1-p_t)** * np.log(p_t)

p_t = np.linspace(0.01, 0.99)

plt.figure(figsize=(12,7))

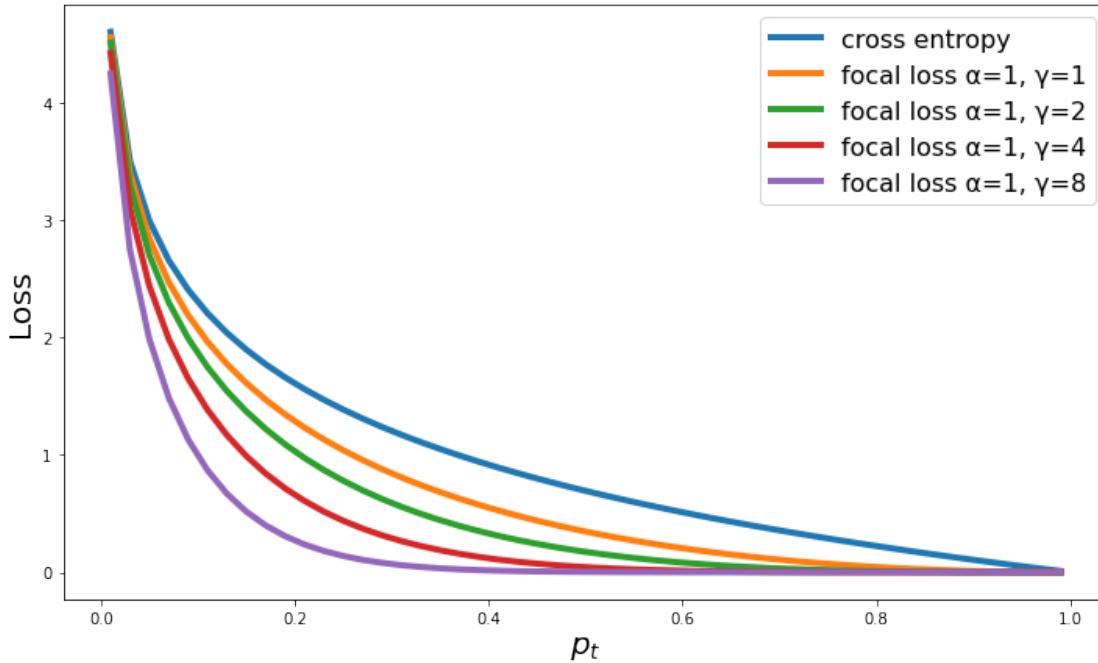
plt.plot(p_t, cr(p_t), lw=4, label="cross entropy")

for in [1]:
    for in [1, 2, 4, 8]:
        plt.plot(p_t, fl(p_t, , ), lw=4, label=f"focal loss ={}, ={ }")

plt.xlabel(r"$p_t$", fontsize=20)
plt.ylabel("Loss", fontsize=20)

plt.legend(fontsize=16)
```

<matplotlib.legend.Legend at 0x7fdc90f4aef0>



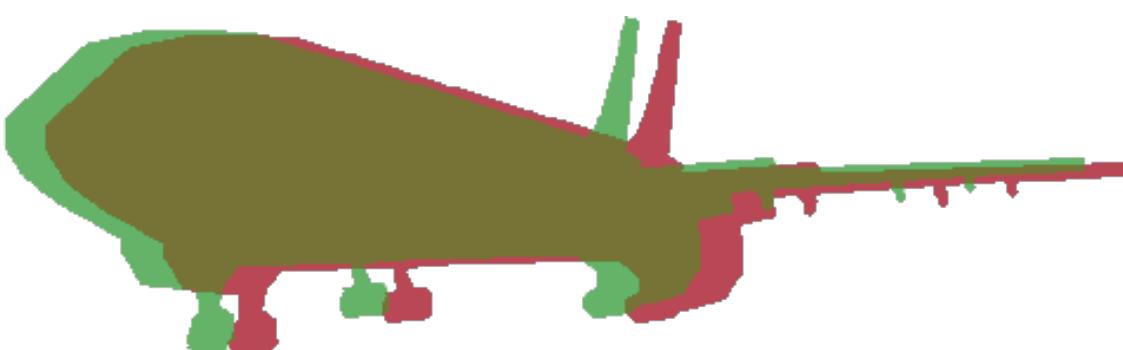
The cross entropy loss for $p_t \approx 0.5$ is pretty close to $L_{\text{CR}} = 1$, while the focal loss is very close to zero. This prevents the model from becoming *overconfident*, which is a common problem in deep learning.

In *unbalanced* problems, where one class is underrepresented or much smaller than other classes, a *class-specific loss coefficient* is often introduced to boost the loss that misclassifying smaller objects causes. This is subject to optimization and needs tuning. Another option is to use a loss from the following class.

7.16.5 Intersection over Union Loss

The idea for this class of losses is relatively simple. Instead of measuring some kind of numerical distance itself, the *overlap* of ground truth and prediction is measured relative to their combined size:

```
Image("img/IoU.png", width=600)
```



The IoU loss is defined as the ratio of the brown area $A_{\text{brown}} = |A_{\text{red}} \cap A_{\text{green}}|$ to the combined red and green areas $|A_{\text{red}} \cup A_{\text{green}}|$. This way, the absolute size of an object is irrelevant to the loss and the overlap of prediction and ground truth is the optimization goal. This helps against class imbalances.

There are various flavors of IoU losses. A common choice is the **Dice loss**, which optimizes $L_{\text{Dice}} = 1 - D$, where D is the *Dice coefficient*:

$$L_{\text{Dice}} = 1 - 2 \frac{|Y_{\text{pred}} \odot Y_{\text{GT}}|}{|Y_{\text{pred}} + Y_{\text{GT}}|}$$

The 2 counterweights the double-counted overlapping areas.

There are many more interesting loss functions to check out, like Lovász-Softmax loss, TopK loss or Hausdorff distance loss, which each tackle a specific problem that may plague a dataset.

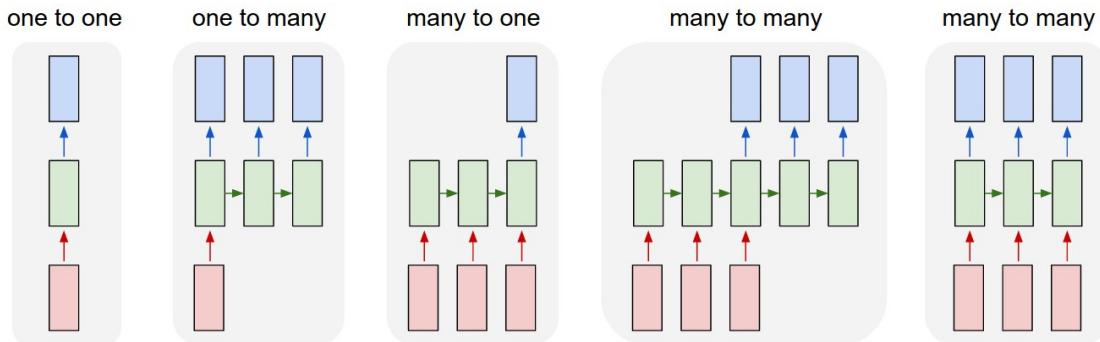
8 Recurrent Neural Networks (RNNs)

8.1 Recurrent Neural Networks

As we have seen, ANNs are great when dealing with purely numerical data and modeling one-to-one correspondences between some inputs and outputs (given there is a correspondence). CNNs are special kinds of ANNs with skip connections and weight sharing, which has huge advantages when dealing with image input, such that the number of weights is decreased drastically compared to fully-connected ANNs. In both cases, the inputs are generally of fixed size and the input has to be available to the network completely, say, an image has to be given as a whole input. There are situations in which this is not possible, like in speech recognition or video analysis, where sequential input of data and sequential processing is needed and input can be variable in size. Sequential input means “time-like” here, or transient, or serial. For example when translating a sentence from one language to another, the order in which the words appear in a sentence is of utmost importance to the meaning and hence, the resulting translation.

A way to deal with these issues is to use **Recurrent Neural Networks**:

```
from IPython.display import Image
Image("img/x-to-x.jpeg", width=1000)
```



In the above image (taken from Andrej Karpathy's extremely insightful blog post [The Unreasonable Effectiveness of RNNs](#)), several possible configurations are shown, each one is a standard RNN. Simple RNNs can be interpreted as ANNs or CNNs with input layer, hidden layer, and output layer. On the left, a *one-to-one* RNN is shown, that takes input from a single point in time and outputs values for a single point in time, say, the next time step. The second image shows a *one-to-many* RNN. This kind of relation can also be found in CNNs, e.g. in image captioning, where a single image is used as input and several captions are computed by the network. Recall the example for image segmentation from last lecture. Instead of segmenting the image, the CNN could have simply only output the classes it found in the image. Another example is weather prediction, where it could take as input the current weather data and produce weather predictions for several days. An example for image captioning can be found below, taken from [Image Captioning with Keras](#).

```
Image("img/image_captioning_example.png", width=1000)
```



```
Greedy: man in black shirt is skateboarding down ramp
```

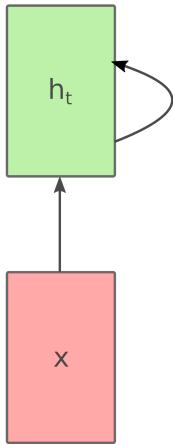
The third image shows a *many-to-one* RNN, which is used in sentiment analysis. The task there is to decide whether, say, a review of a movie is inclining to be positive or negative. This is also used for stock market analysis, where especially news articles and social media posts are analyzed (a few years ago, someone took over the twitter account of a news agency and tweeted something about a bomb having been found in the white house, which caused an almost instant 10% drop in the Dow Jones, costing some people a lot of money).

The fourth image shows a *many-to-many* RNN, which is used for example for translations and speech recognition, where a set of words is translated to a different set of words or an audio signal is “translated” into a sequence of commands. The output and input size can differ. In frame-to-frame video analysis, it is usually the same size as shown in the fifth image.

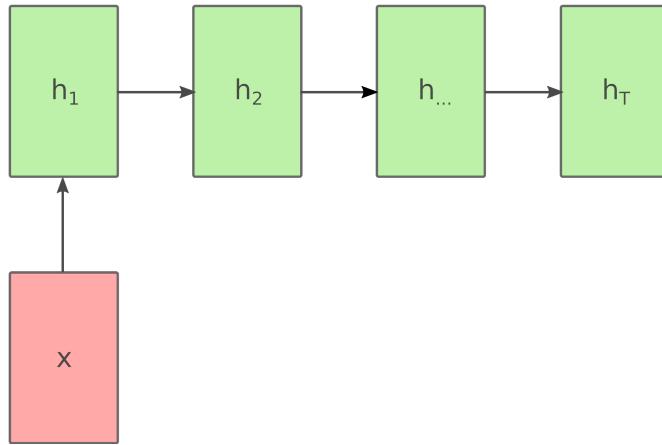
So in general, RNNs handle variably sized, sequential data. Their idea is to combine an input vector of sequential data with a *state vector* via a fixed function to produce a new state, where the number of functions is fixed, but the size of the data is not. We’ll see what exactly that means in a moment.

```
Image("img/rnn_unrolled.png", width=1000)
```

Standard



Unrolled

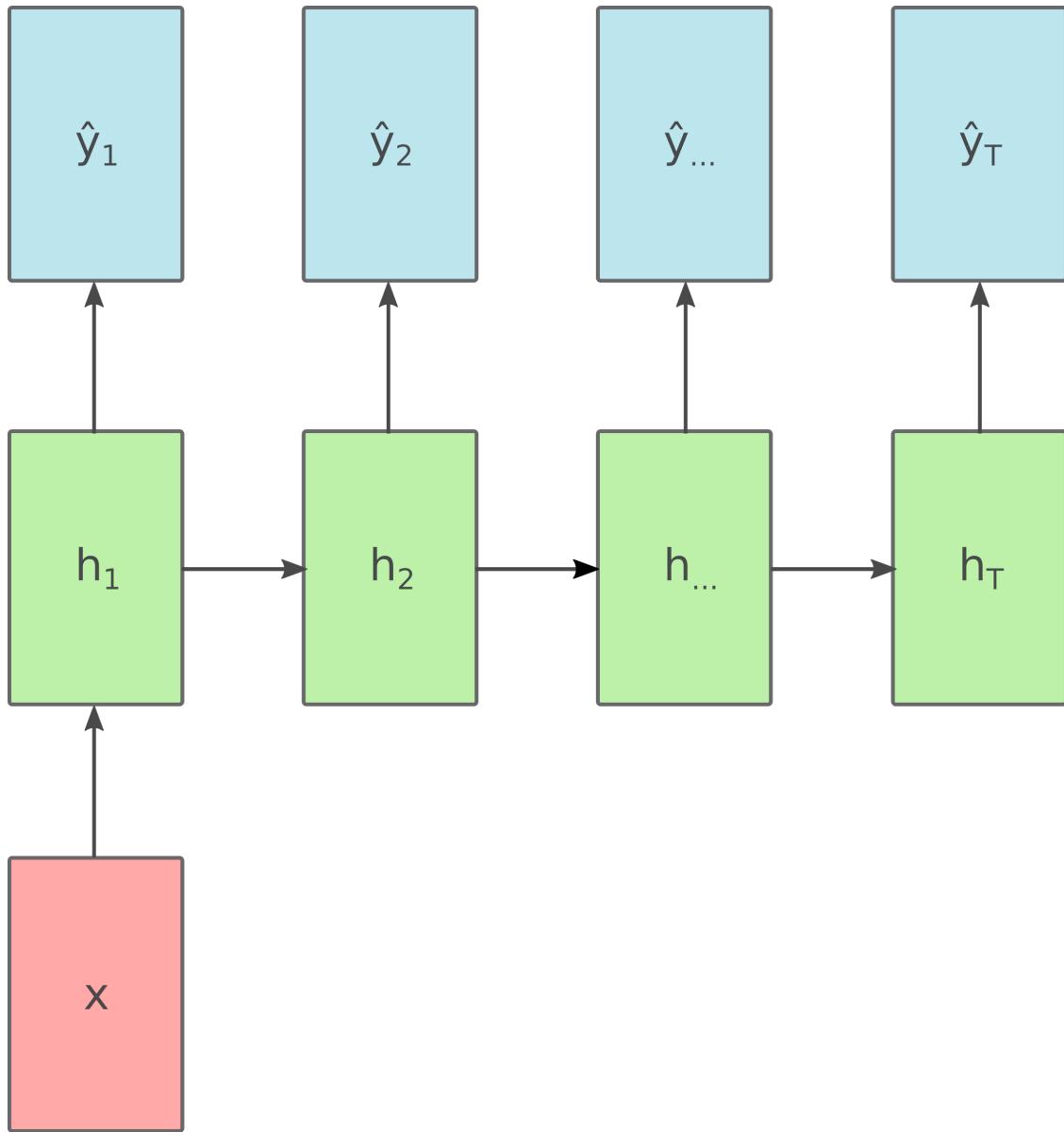


The image above shows typical representations of RNNs. The left one is the actual RNN (we'll see what the self-referential arrow means later), the right one is an *unrolled* RNN, where each column represents what happens in a single time step t . The latter is easier to understand intuitively, so we will mostly work with this visualization.

Side note: what makes RNNs work so well on sequential data is their (theoretical) Turing-completeness, which means they can approximate any given algorithm arbitrarily well. Siegelmann and Sontag showed this property in 1992 for finite size RNNs with rational weights. In later papers they even showed that giving an RNN real weights (including uncomputable irrational weights), it would be even more powerful than a Turing machine which has deep implications for computer science in general. Unfortunately, this is just a mathematical model and we cannot represent irrational numbers numerically, in fact we cannot even represent every rational number since we're confined by the available machine precision and as we have seen in the 0th lecture, some rational numbers (like 0.1) necessitate infinite binary representations. We will later talk about generalizations of RNNs, namely LSTMs and GRUs. Although they are proper generalizations in theory, it is not clear whether they maintain the Turing-completeness of standard RNNs, at least practically.

Let's look at an example. For weather predictions, currently only classical methods are used, since deep learning models aren't well understood yet, but research is focusing a lot on these kinds of things since they show promising results for chaotic systems, like in [flame propagation](#). So although this is not done in practice, a way to use RNNs is to take as input weather-related data from today (say pressure, temperature, humidity, precipitation) and predict the temperature for the following days (scalar outputs for simplicity, otherwise the outputs would be vectors). The idea here is that given fixed differences in the time steps, the predicted values for the day after tomorrow should be close to the values predicted for tomorrow. This assumed similarity for temporal relationships in equally-spaced, repetitive data is inherent in RNNs. So roughly speaking, the relationship between h_3 and h_2 should be the same as between h_2 and h_1 in the following image:

```
Image("img/rnn_weather.png", width=500)
```



This is just for a basic intuition about how RNNs work (*Turing-completeness is what is actually going on here*). In this example, we only have an input x_1 consisting of four features (pressure, temperature, humidity, precipitation), but we don't have an x_2 for tomorrow. The information for tomorrow's predicted weather is encoded in h_2 :

$$\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{h}_{t-1}, \mathbf{x}_t) = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t) \quad (116)$$

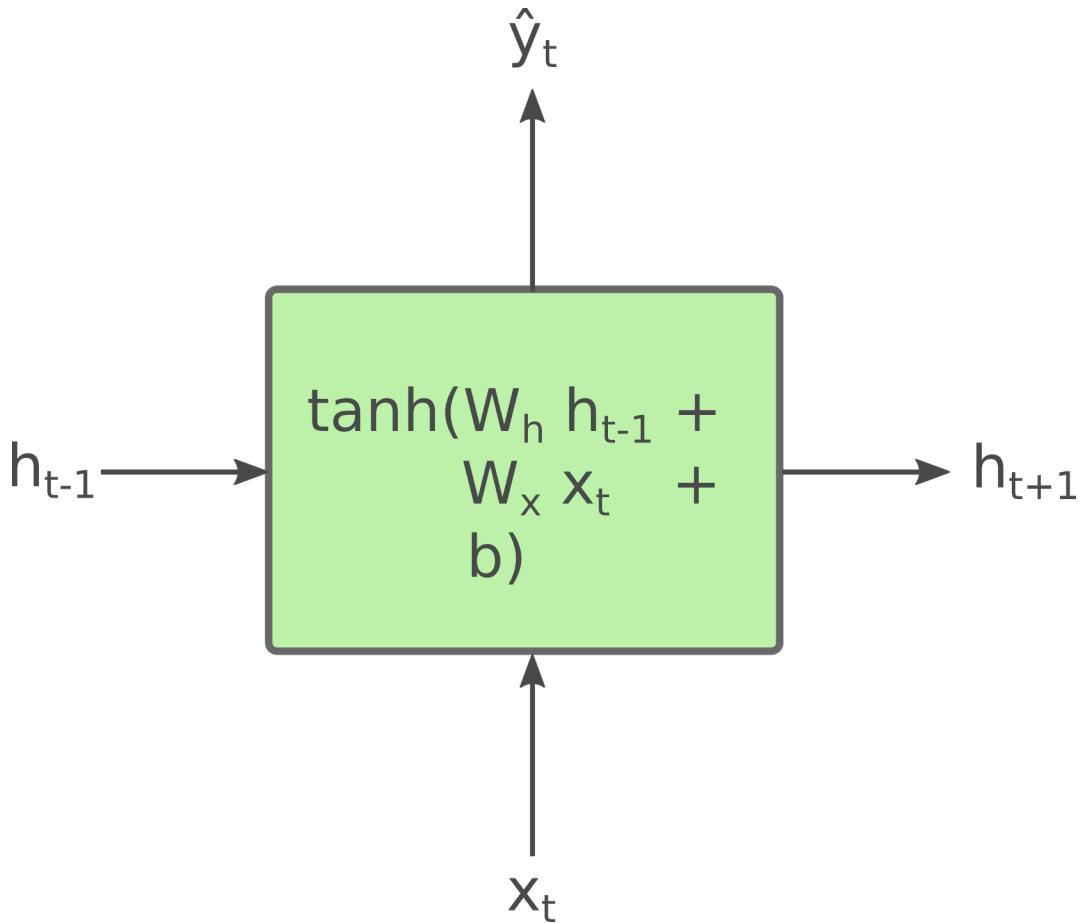
where $f_{\mathbf{W}}$ is the aforementioned fixed function, which is simply the activated mac of some inputs as before (since this is the most general form of function in an ANN), \mathbf{W}_h are the weights connecting different **hidden states** \mathbf{h}_t and \mathbf{W}_x are the weights connecting the hidden states and the input \mathbf{x}_t . The activation can be arbitrary, but usually tanh is used in RNNs. The prediction is then

$$\hat{\mathbf{y}}_t = a(\mathbf{W}_y \mathbf{h}_t) \quad (117)$$

where often the activation a is just linear $a(z) = z$, but depends on the problem. For binary classification the activation would be sigmoid, for multiclassification softmax and so on. We can extract predictions at arbitrary points in time, since the predictions do not affect the network itself. Sometimes it makes sense to not create predictions for every time step, but only a few, or only make a prediction at the end of the sequence. It's also possible to insert new information as soon as an update comes. Say we created an RNN model for the weather example and tomorrow comes, so we get the data for $t = 2$ as well. It can now be inserted at $t = 2$ into the network to increase predictive performance without having to train a new model, but reusing the old trained model. This will then become a many-to-many prediction. Google (or rather DeepMind) recently published a [paper](#) where they used such a model to predict energy output of wind farms, which makes the produced energy more valuable for the grid compared to wind farms where energy output can not be predicted. So now $\mathbf{h}_2 = \tanh(\mathbf{W}_h \mathbf{h}_1 + \mathbf{W}_x \mathbf{x}_2 + \mathbf{b})$ (bias is mostly omitted in this lecture). The size of the \mathbf{h}_t is chosen by the developer, \mathbf{x}_2 is 4×1 and $\hat{\mathbf{y}}_t$ is 1×1 . It is currently not known how to interpret the data in the \mathbf{h}_t or what exactly they mean, since this is a forward model, which means the relationship between inputs and outputs is guessed and the weights adjusted for this guess.

Recall the idea of constant relationships between data for equally-spaced consecutive points in time (especially in engineering situations a fixed Δt makes sense). The result is that the weights and biases \mathbf{W}_y , \mathbf{W}_h , \mathbf{W}_x , and \mathbf{b} are *fixed* along the time steps. The weight matrices are not updated while going forward in time, otherwise computational cost would increase drastically. At some point in time t we have

```
Image("img/single_ht.png", width=600)
```



where \mathbf{h}_t is of size 100×1 , \mathbf{x}_t is of size 4×1 , and \hat{y}_t is of size 1×1 . What are the corresponding sizes of the weight and bias matrices? To be compatible, \mathbf{W}_y must be of size 1×100 , \mathbf{W}_x must be of size 100×4 , \mathbf{W}_h of size 100×100 , and \mathbf{b} of size 100×1 , so together there are $100 + 400 + 10000 + 100 = 10600$ trainable parameters in this model. Since weights are fixed according to the abovementioned idea, this is all that's necessary to train for such a model. Otherwise the number of trainable parameters would scale with the number of time steps in the system and increase quickly. This is what keeps RNNs compact while retaining the predictive capabilities for time-series data.

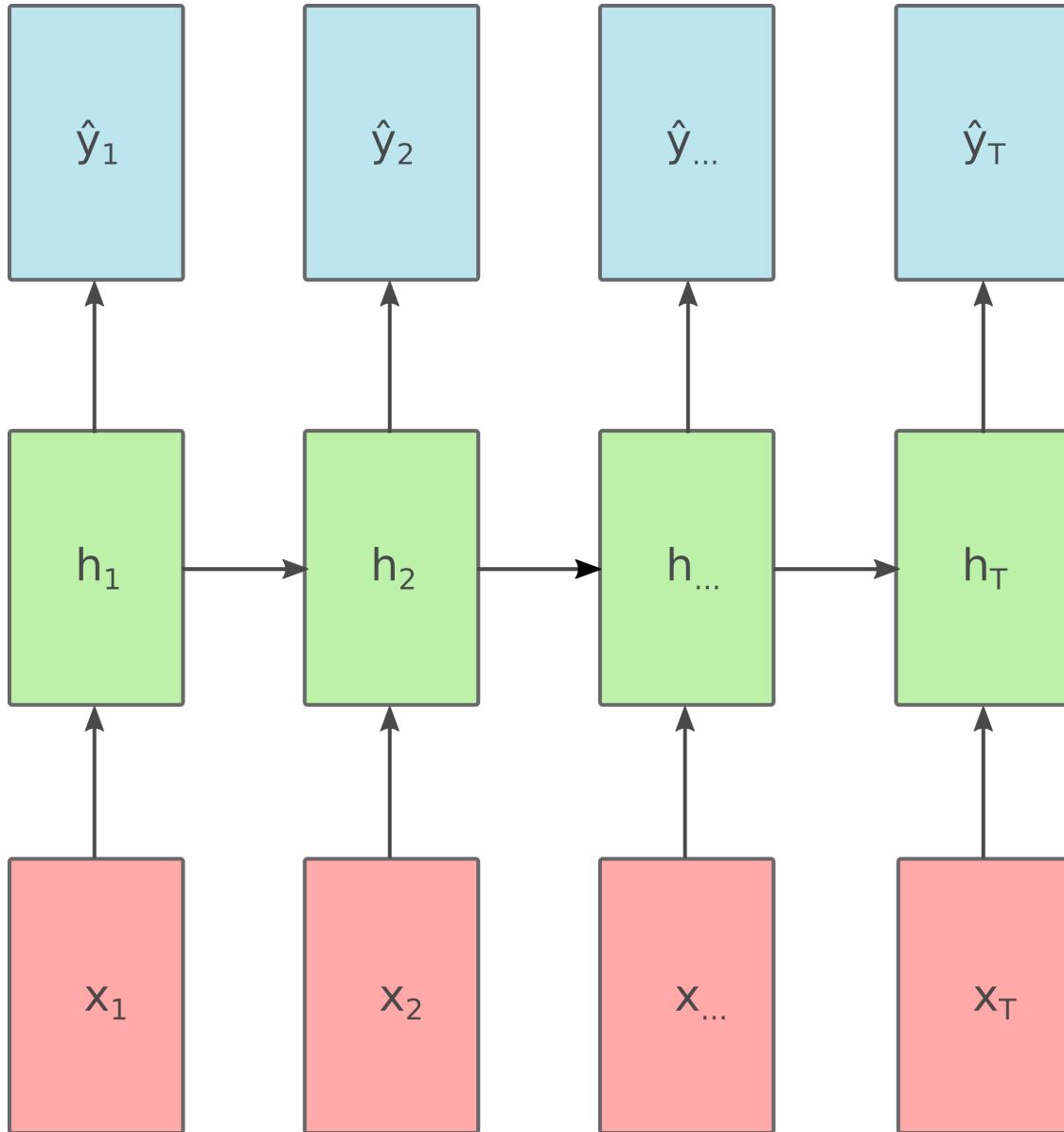
8.2 RNN Loss and Backpropagation Through Time

Training an RNN at a first glance is quite similar to training ANNs or CNNs, but there are a few fine differences in how loss is calculated and in how error is backpropagated. Since now the networks has a sense of time, the error needs to be backpropagated not only through the layers in the network, but also through time.

8.2.1 Loss

The number of total time steps T processed by the network determines how often the recurrent layer iterates over the computation shown in the last lesson.

```
from IPython.display import Image
Image("img/rnn_unrolled_complete.png", width=500)
```



It's not necessary to provide input at every time step, as we've seen. It's also not necessary to make predictions at every timestep (making a prediction does not change anything inside the network, but does influence the loss). Where to provide inputs and make predictions is completely problem-specific. It is perfectly possible to only make a prediction at the very last time step $t = T$ or every second time step for example. If for every prediction \hat{y}_t made, there exists a corresponding *ground truth* y_t , then at every such time step there is a natural loss function induced by the loss functions we already got to know (in principle, you could define a different loss function for every time step, but that rarely makes sense, if at all). This could be the *mean square loss* for example, or *cross*

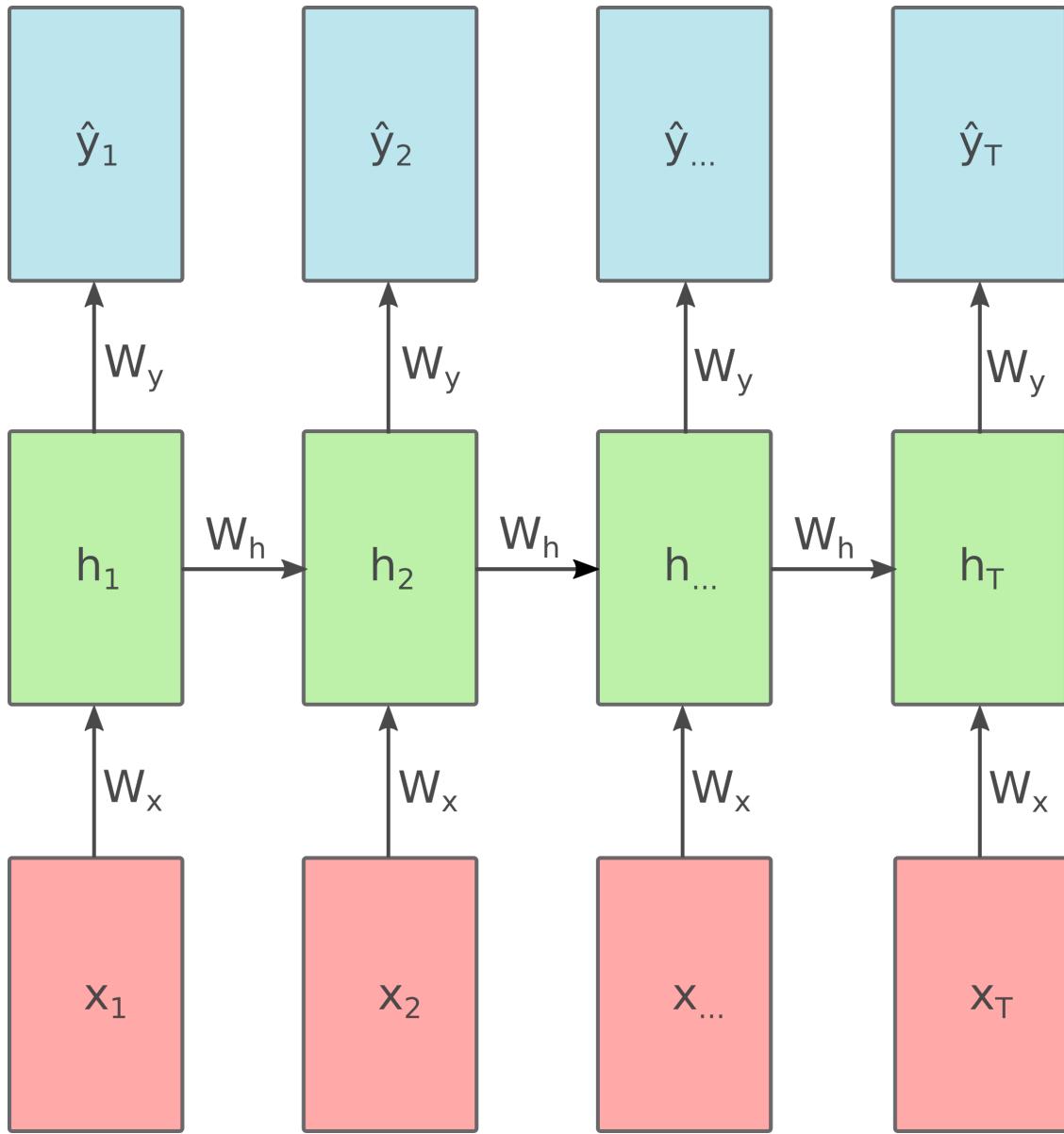
entropy, depending on the problem. The total loss for the network is then simply the sum of all “local”/intermediate losses

$$L = \sum_{t=1}^T L_t \quad (118)$$

8.2.2 Backpropagation Through Time (BPTT)

For simplicity, let’s assume we’re taking outputs at every time step. As before we have $h_t = a(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t)$ (with implicit bias) and $\hat{\mathbf{y}}_t = \tilde{a}(\mathbf{W}_y \mathbf{h}_t)$, where a and \tilde{a} can be different activation functions. According to the idea of similarity between consecutive time steps, the three weights matrices *do not change through time* (meaning across “layers”).

```
Image("img/rnn_unrolled_complete_matrices.png", width=500)
```



To define backpropagation for this kind of structure, we need to find the partial derivatives $\frac{\partial L}{\partial \mathbf{W}_h}$, $\frac{\partial L}{\partial \mathbf{W}_x}$, and $\frac{\partial L}{\partial \mathbf{W}_y}$. Starting with the intermediate loss $\frac{\partial L_3}{\partial \mathbf{W}_y}$ and assuming \tilde{a} to be linear activation and the loss to be mean square loss we get

$$\frac{\partial L_3}{\partial \mathbf{W}_y} = \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial \mathbf{W}_y} \quad (119)$$

$$= -(y_3 - \hat{y}_3) \otimes \mathbf{h}_3 \quad (120)$$

The other ones are a bit more tricky:

$$\frac{\partial L_3}{\partial \mathbf{W}_h} = \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_h} \quad (121)$$

$$= -(y_3 - \hat{y}_3) \mathbf{W}_y a'(\mathbf{z}_3) \frac{\partial \mathbf{z}_3}{\partial \mathbf{W}_h} \quad (122)$$

with $\mathbf{h}_3 = a(\mathbf{z}_3)$ and $\mathbf{z}_3 = \mathbf{W}_h \mathbf{h}_2 + \mathbf{W}_x \mathbf{x}_3$. So the problem now is that the weights are the same in every layer and this becomes a recursive derivative. Calculating the gradient L_t involves calculating *all* the gradients for time steps before t . The same thing happens for the remaining derivative:

$$\frac{\partial L_3}{\partial \mathbf{W}_x} = \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial \mathbf{h}_3} \frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_x} \quad (123)$$

$$= -(y_3 - \hat{y}_3) \mathbf{W}_y a'(\mathbf{z}_3) \frac{\partial \mathbf{z}_3}{\partial \mathbf{W}_x} \quad (124)$$

For cases where inputs and predictions aren't provided/taken at every time step, tracking the flow of derivatives becomes tedious quickly. Frameworks like TensorFlow represent these networks as *graphs* and traverse the graph using efficient algorithms to perform automatic differentiation.

When the gradients are calculated, the error is then backpropagated as usual:

$$W^{(k+1)} = W^{(k)} - \alpha \frac{\partial L}{\partial W} \quad (125)$$

where $L = \sum L_t$.

8.3 Vanishing Gradients and Truncated BPTT

Calculating gradients in RNNs and backpropagating them correctly is computationally expensive, and they can suffer from problems called **vanishing gradients** or **exploding gradients**. There are various solutions to this problems and we will explore a few here.

8.3.1 The Problematic Gradient

The dependency of the gradient on earlier time steps is the cause of these problems. To give a brief and rough explanation of why this happens (just for intuition), we make a few assumptions.

Inputs \mathbf{x}_t are ignored completely, and all activation functions are linear, such that now $\mathbf{h}_t = \mathbf{W}_h \mathbf{h}_{t-1}$. \mathbf{h}_t now scales with \mathbf{W}_h , such that $\mathbf{h}_{t+1} = \mathbf{W}_h^2 \mathbf{h}_{t-1}$, or in general $\mathbf{h}_{t+n} = \mathbf{W}_h^n \mathbf{h}_{t-1}$. Now taking any matrix norm (it doesn't really matter which one), this becomes

$$\|\mathbf{h}_{t+n}\| \approx \lambda_h^n \|\mathbf{h}_{t-1}\| \quad (126)$$

so $\|\mathbf{h}_t\|$ scales roughly with λ_h , which is an eigenvalue of \mathbf{W}_h (either the largest or the smalles one). This scaling means that with increasing time, the values in \mathbf{h}_t become larger and larger (or smaller and smaller) in magnitude, depending on whether $|\lambda_h| > 1$ or $|\lambda_h| < 1$. So with increasing n $\|\mathbf{h}_{t+n}\| \rightarrow \infty/0$.

A similar argument holds for the gradient of the loss function, since $\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}} = W \frac{\partial \mathbf{h}_2}{\partial \mathbf{W}}$ and so on, such that $\left\| \frac{\partial L}{\partial \mathbf{W}} \right\| \rightarrow \infty / 0$ with increasing n .

This is what causes *exploding/vanishing gradients*. Since we are constrained to finite machine precision (a computer cannot represent arbitrary numbers arbitrarily exact, see for example 0.1 in the very first lecture 00), overflow/underflow errors will appear at some point and completely interrupt the training process. This is sometimes also called *saturation*. The same things happens when introducing tanh as the activation function:

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact

fig = plt.figure(figsize=(8,8))

x = np.linspace(-5, 5, 40)

def plot_tanh(power=1):
    plt.cla()
    plt.plot(x, np.tanh(x)**power, lw=4, label=fr"tanh$^{{{{power}}}}()$")
    plt.xlim([-5, 5])
    plt.ylim([-1.1, 1.1])
    plt.legend(fontsize=16)

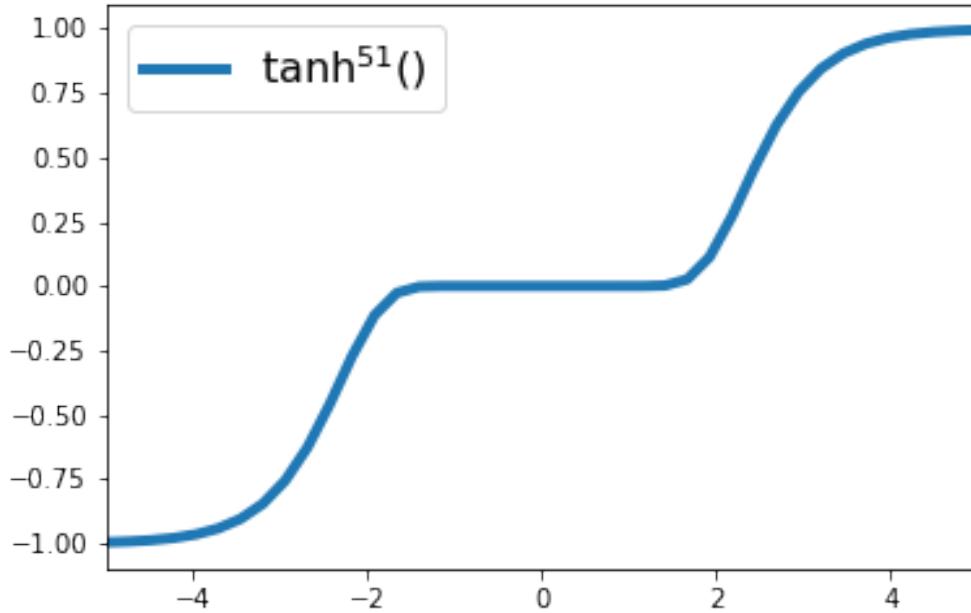
    plt.savefig("img/plot_vanishing_grad_tanh.png")

interact(plot_tanh, power=(1, 51, 5))
```

<Figure size 576x576 with 0 Axes>

interactive(children=(IntSlider(value=1, description='power', max=51, min=1, step=5), Output()))

<function __main__.plot_tanh(power=1)>



So high powers of tanh will make activations approach zero for a growing range of values around zero.

8.3.2 Gradient Clipping

To help the exploding gradient problem, a *threshold* or *cutoff* value G_{\max} is given by the developer, e.g. $\frac{\partial L}{\partial \mathbf{W}} = \mathbf{g}$ with $\max \|\mathbf{g}\| = G_{\max}$. While performing gradient descent, $\|\mathbf{g}\|$ is calculated and compared to the given threshold value. If it is below, the iteration continues normally. Otherwise, \mathbf{g} is set to

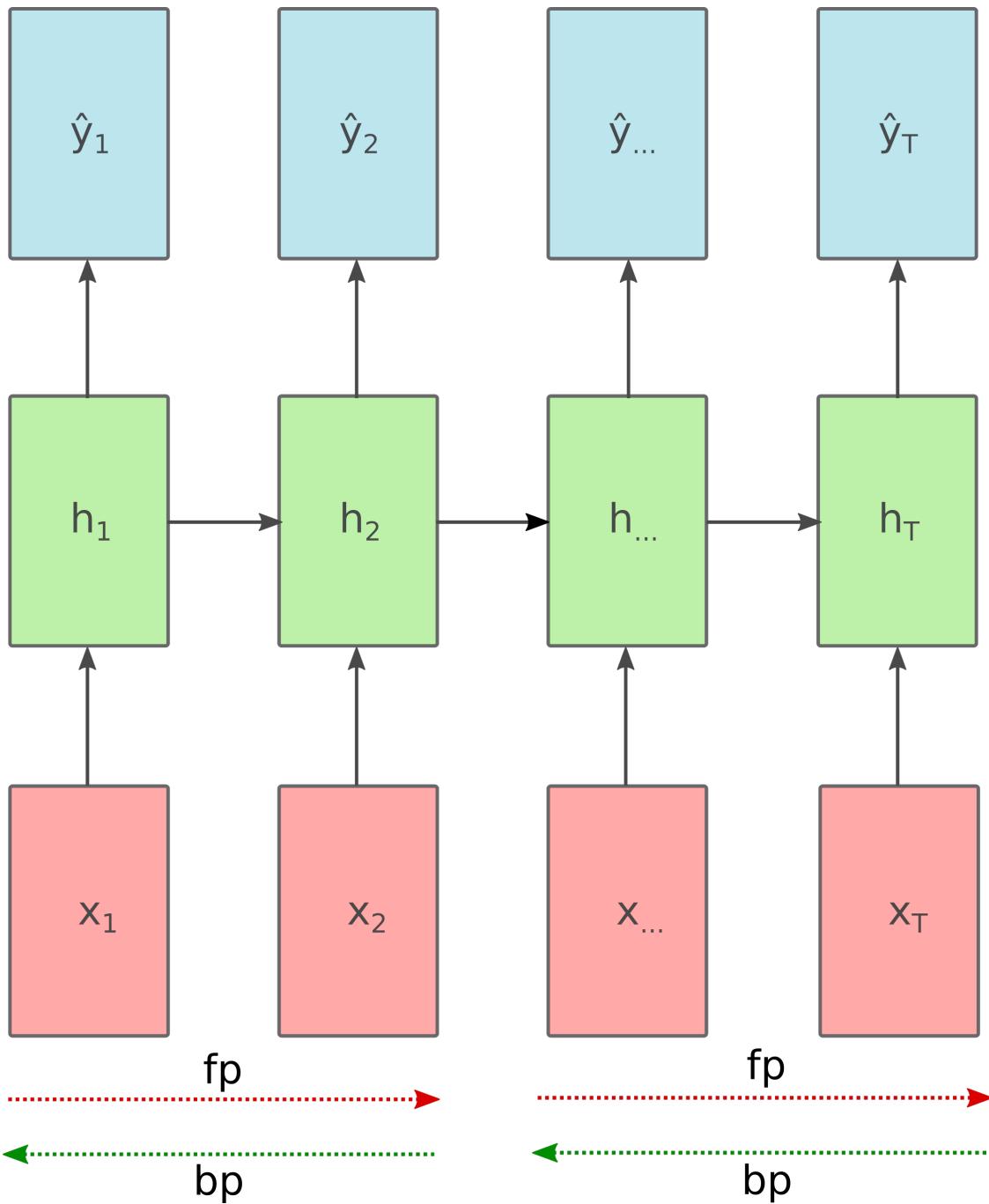
$$\mathbf{g} = \frac{\mathbf{g}}{\|\mathbf{g}\|} G_{\max} \quad (127)$$

such that the new gradient shows in the exact same direction as before, but is scaled down significantly.

8.3.3 Truncated BPTT

This helps with all the problems mentioned in the beginning of this lesson to a certain extent.

```
from IPython.display import Image
Image("img/tbptt.png", width=500)
```



Imagine the above image shows an unrolled network for several thousand time steps. To backpropagate prediction error, one would have to perform a complete forward pass, calculate the complete loss gradient, then backpropagate completely through all layers. Hence, it's extremely expensive to do just a single weight update. An idea similar to stochastic gradient descent is to use **truncated BPTT** (see arrows in the image above). Given the time-invariant nature of RNNs, the cuts can be set pretty much anywhere in the unrolled network, the arrows here are just an example. The sequences are then a split version of the full backpropagation. Here, we would forward pass

through 2 layers, then backprop through these two layers, do the same thing for the next two layers, backpropagate through the last 2 layers, and so on. In general, the forward pass would be done through k_1 layers, the backpropagation back through k_2 layers. In this scenario (given small enough k_1 and k_2), the gradient would neither vanish nor explode. Choosing good values for the truncation is specific to every problem setup and needs experimentation (as it is the case for most ML techniques).

Generally, k_2 has to be chosen sufficiently large to capture the temporal structure of the problem. If after 4 time steps the state of the problem is decorrelated from the beginning, a $k_2 = 4$ may make sense. Anything smaller would obfuscate the temporal information in the sequence. If k_2 becomes too large, the vanishing or exploding gradients problem takes over again.

8.3.4 Different Architectures

These help with the vanishing gradient problem. We will discuss this approach in the next lesson with *GRUs* and *LSTMs*.

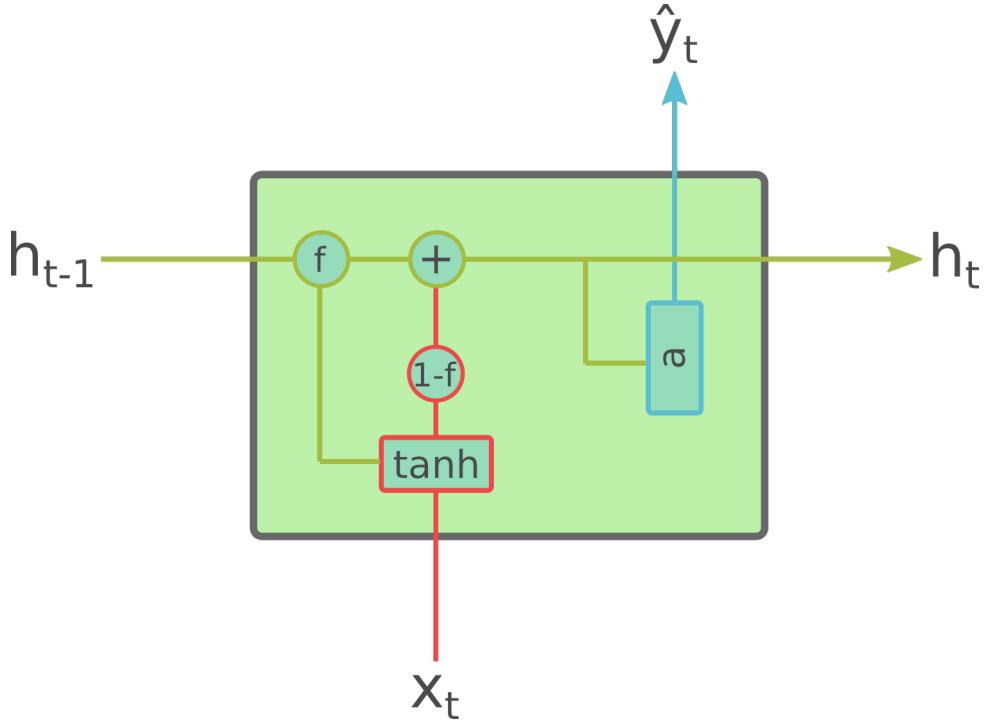
8.4 Gated Recurrent Units

The problems of vanishing/exploding gradients are caused by repeatedly applying \mathbf{W} (and \tanh) in standard RNNs. This made updating the weights in RNNs extremely difficult up until the late 90s. Gradient clipping/thresholding could be used against exploding gradients, but the first good solution to vanishing gradients was found a lot later in the form of *Long Short-Term Memory units* (LSTMs) by Hochreiter, Schmidhuber 1997. **Gated Recurrent Units** (GRUs) are a simplified version of them, found by KyungHyun 2014. Typically, instead of standard RNNs, one of these architectures is used in transient problems.

The basic idea is that we need some form of *memory* or *history-awareness* in ANNs, roughly similar to how we process speech for example. When a sentence is spoken, e.g. “don’t do this”, our brain uses its short-term memory to keep the spoken words in mind and relate them to each other in order to derive the correct meaning. Effectively, in the algorithm we looked at so far, we have a memory of a single time step (\mathbf{h}_t only depends on \mathbf{h}_{t-1} , not on earlier time steps), everything else is lost. A solution could be to introduce a separate memory cell which store earlier computation states, as is the case for LSTMs (this is very similar to relaxation schemes found in some numerical algorithms), or to perform “memory-like” operations, as is the case for GRUs.

Here, we will start with a simplified version of GRUs to develop an intuition for how they work in the way that Andrew Ng does in his courses (see recommendations in Ilias, this is not used in practice).

```
from IPython.display import Image
Image("img/gru_simple.png", width=700)
```



\mathbf{h}_t is the output of a standard RNN. Relabeling $\mathbf{g} := \mathbf{h}_t = \tanh(\mathbf{z}_g)$ with $\mathbf{z}_g = \mathbf{W}_{hg}\mathbf{h}_{t-1} + \mathbf{W}_{xg}\mathbf{x}_t$ and giving an index g to the matrices \mathbf{W}_{hg} and \mathbf{W}_{xg} , nothing has changed so far. This simplified version of a GRU takes a linear combination of the output of a standard RNN and older computations in the following way:

$$\mathbf{h}_t = f\mathbf{h}_{t-1} + (1 - f)\mathbf{g} \quad (128)$$

where f is some scalar (for now), the first part is older output from a standard RNN and the second part is current output from a standard RNN. This is very similar to the *momentum method* we got to know as an alternative to standard gradient descent. For $f = 1$, this is a pure memory without any computation happening. For $f = 0$ we get back a standard RNN, so this is an interpolation between these two and a generalization of RNNs. More generally, f is a vector \mathbf{f} and $\mathbf{f} \odot$ is called a **forget gate**:

$$\mathbf{h}_t = \mathbf{f} \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}) \odot \mathbf{g} \quad (129)$$

If \mathbf{h}_t is of size 100×1 , so is \mathbf{g} and in turn, \mathbf{f} has to be of size 100×1 as well. So \mathbf{f} operates independently on each component of the state vectors.

Instead of manually choosing values for \mathbf{f} , we let the algorithm *learn* good values. In order for the interpolation to work, the components of \mathbf{f} must be $\in [0, 1]$, which can easily be ensured by using the sigmoid function:

$$\mathbf{f} = \sigma(\mathbf{z}_f) = \sigma(\mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t) \quad (130)$$

Note that the W s here are different from the matrices before.

So again, we have an affine map and a nonlinearity applied. This introduces an additional number of parameters to train.

The full GRU is an only slightly modified version of this:

$$\mathbf{h}_t = \mathbf{f} \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}) \odot \mathbf{g} \quad (131)$$

with $\mathbf{g} = \tanh(\mathbf{z}_g)$, $\mathbf{f} = \sigma(\mathbf{z}_f)$ and $\mathbf{z}_f = \mathbf{W}_{hf}\mathbf{h}_{t-1} + \mathbf{W}_{xf}\mathbf{x}_t$, $\mathbf{z}_g = \mathbf{W}_{hg}(\mathbf{r} \odot \mathbf{h}_{t-1}) + \mathbf{W}_{xg}\mathbf{x}_t$ with the “remember gate” $\mathbf{r} \odot$, for which $\mathbf{r} = \sigma(\mathbf{z}_r)$ and $\mathbf{z}_r = \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{W}_{xr}\mathbf{x}_t$.

Again, $\mathbf{r} \in [0, 1]^{\dim(\mathbf{r})}$, ensured by the sigmoid function. Now, instead of having 3 weight matrices like for standard RNNs, we have 7 weight matrices, increasing the number of trainable parameters.

GRUs work much better for deeper networks compared to simple RNNs, justifying this increase in trainable parameters. If for some problem an RNN of 10 layers produced promising results, GRUs can go up to almost 100 layers for the same problem, at least very roughly speaking. LSTMs work very similarly to this. Despite their old age, they are much more commonly found compared to GRUs and they work at least as well as they do.

8.5 Long Short-Term Memories

In the last lesson we saw how GRUs handle vanishing and exploding gradients with a simple architectural change. In a simplified GRU, we had $\mathbf{h}_t = \mathbf{f} \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}) \odot \mathbf{g}$, where the first term retains some information from before. In LSTMs, this is done by using a separate memory cell, with

$$\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{i} \odot \mathbf{g} \quad (132)$$

with the **input gate** $\mathbf{i} \odot$, and $\mathbf{i} \in [0, 1]^{\dim(h)}$ and a *forget gate* $\mathbf{f} \odot$ with the same property again. Now

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t) \quad (133)$$

with the **output gate** $\mathbf{o} \odot$ and $\mathbf{o} \in [0, 1]^{\dim(h)}$. To summarize:

$$\mathbf{h}_t = \mathbf{o} \odot \tanh(\mathbf{c}_t) \quad (134)$$

$$\mathbf{c}_t = \mathbf{f} \odot \mathbf{c}_{t-1} + \mathbf{i} \odot \mathbf{g} \quad (135)$$

$$\mathbf{g} = \tanh(\mathbf{z}_g) \quad (136)$$

$$\mathbf{o} = \sigma(\mathbf{z}_o) \quad (137)$$

$$\mathbf{f} = \sigma(\mathbf{z}_f) \quad (138)$$

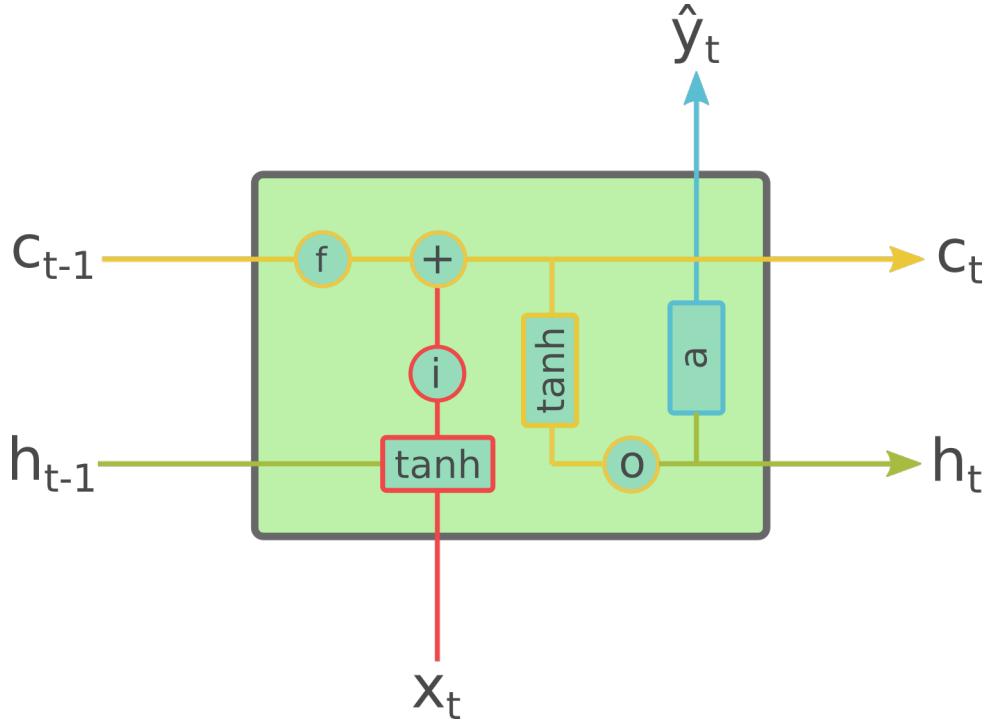
$$\mathbf{i} = \sigma(\mathbf{z}_i) \quad (139)$$

$$\mathbf{z}_\gamma = \mathbf{W}_{h\gamma}\mathbf{h}_{t-1} + \mathbf{W}_{x\gamma}\mathbf{x}_t \quad (140)$$

All of these equations together form an LSTM network. Now, there are 8 unknown weight matrices compared to 6 in the full GRU, 4 in the simplified GRU, and 2 in the standard RNN. Backpropagation here works just like what we've seen, but now for 8 matrices. Compared to a GRU, an LSTM can handle even more layers reliably, so for really deep networks LSTMs would be the method of choice for sequential data.

LSTMs are more difficult to train than the other models we've seen, and it takes more time to achieve convergence. They are also less efficient to run because more matrices have to be applied altogether. In practice, one would start to tackle a problem with RNNs and shallow architecture. If that doesn't work, one would move to GRUs and a lot more layers. If that still isn't sufficient, LSTMs might help with even more layers.

```
from IPython.display import Image
Image("img/lstm_simple.png", width=700)
```



Although LSTMs are more difficult/time-consuming in training, they are the de facto standard today for approximating sequential data.

How do GRUs and LSTMs actually help with the gradient problems? We saw that in a simplified GRU, the problem was roughly the repeated application of the weight matrices

$$\mathbf{h}_t = \mathbf{f} \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}) \odot \mathbf{g} \quad (141)$$

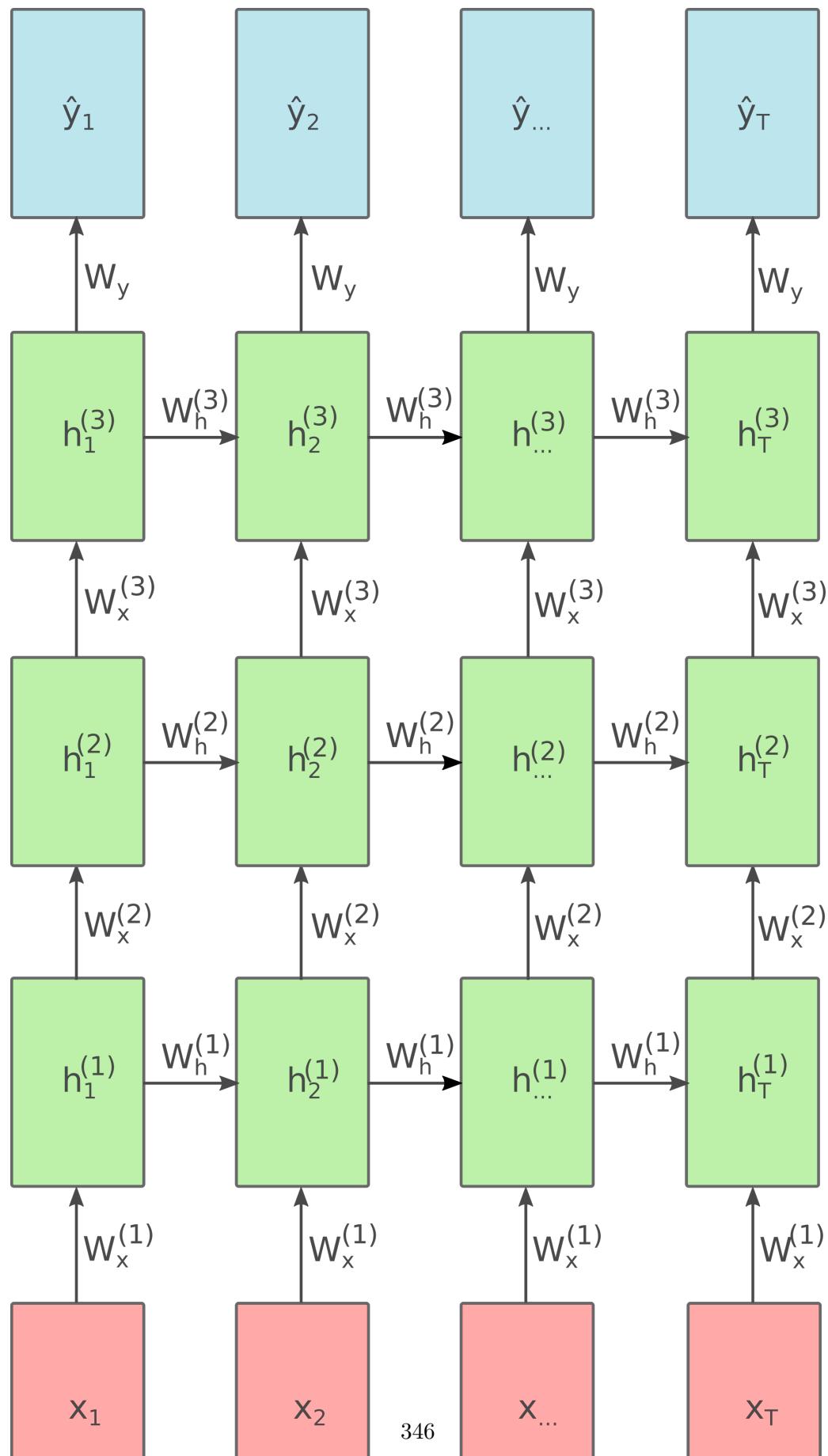
where \mathbf{f} goes with W_f and $\mathbf{W}^n \approx \lambda^n$. So if W becomes very small, the first factor will vanish and the second will become close to one. So in a certain sense, these terms balance each other

out. What really helps here is the “+” operation. Now, there’s an “alternative path” for the gradient to go through during backpropagation, where it can either go through \mathbf{h}_{t-1} or through g . The same thing happens in LSTMs. This idea of providing the gradient alternative pathways to backpropagate through is generally a good way to deal with vanishing or exploding gradient problems in algorithms. This is similar to what the skip connections in CNNs did in the last lecture. The gradients backpropagate for longer before they start vanishing.

8.6 Deep RNNs and Bidirectional RNNs

So far, we called the unrolled time steps “layers”, but actually these are not neural network layers in the usual sense. It is possible to stack multiple recurrent (or GRU or LSTM) units on top of each other, where each unit would constitute a layer, like in the following image of an unrolled deep RNN:

```
from IPython.display import Image  
Image("img/rnn_deep.png", width=400)
```



Here, every parameter in a certain layer gets a layer identifier (l) again, like before. You can think of this like a usual deep ANN (with loops around the hidden layers), but unrolled. Sideways, the structure is exactly replicated, the weights are again exactly the same again, but there are different weight matrices for each layer now:

$$\mathbf{h}_t^{(l)} = \tanh(\mathbf{W}_h^{(l)} \mathbf{h}_{t-1}^{(l)} + \mathbf{W}_x^{(l)} \mathbf{h}_t^{(l-1)}) \quad (142)$$

These deep RNNs can deal with much more complex tasks. Backpropagation is obviously much more difficult here, because there are several pathways to follow, but again, programs like TensorFlow solve this issue by expressing the connections as graphs and traversing that graph to redistribute the error correctly.

8.7 Bidirectional RNNs

The architectures we got to know need a modification when we're dealing with problems where sequences can go forwards and backwards, in other words, where predictions for past values depend on future values. A simple example could be a difficult to read handwritten word, where later letters (and a derived meaning of the word) could help in identifying ambiguous letters. Translation is another example, where you usually need to process the full sentence to correctly translate it meaningfully to a different language.

For this to work, the gradients need to be *bidirectional* and also “back”propagated in both directions. An example from engineering (or rather signal processing) is a varying signal, where parts of it are supposed to be classified. Say for example you’re looking at the signal output of a 3D accelerometer in a car and you want to classify different parts of the signal as indicating the car standing still, moving normally, accelerating, decelerating, or perhaps having a crash. Then because the baseline at first is unknown, you cannot really tell at the beginning of the graph what that portion belongs to. But later, when the development of the signal is known, inferences can be made about what the first part of the signal might mean.

```

import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0, 5, 20)
x2 = np.linspace(5, 10, 20)
x3 = np.linspace(10, 15, 20)

r1 = np.random.random(20)
r2 = np.random.random(20)+4
r3 = 4*np.random.random(20)+2

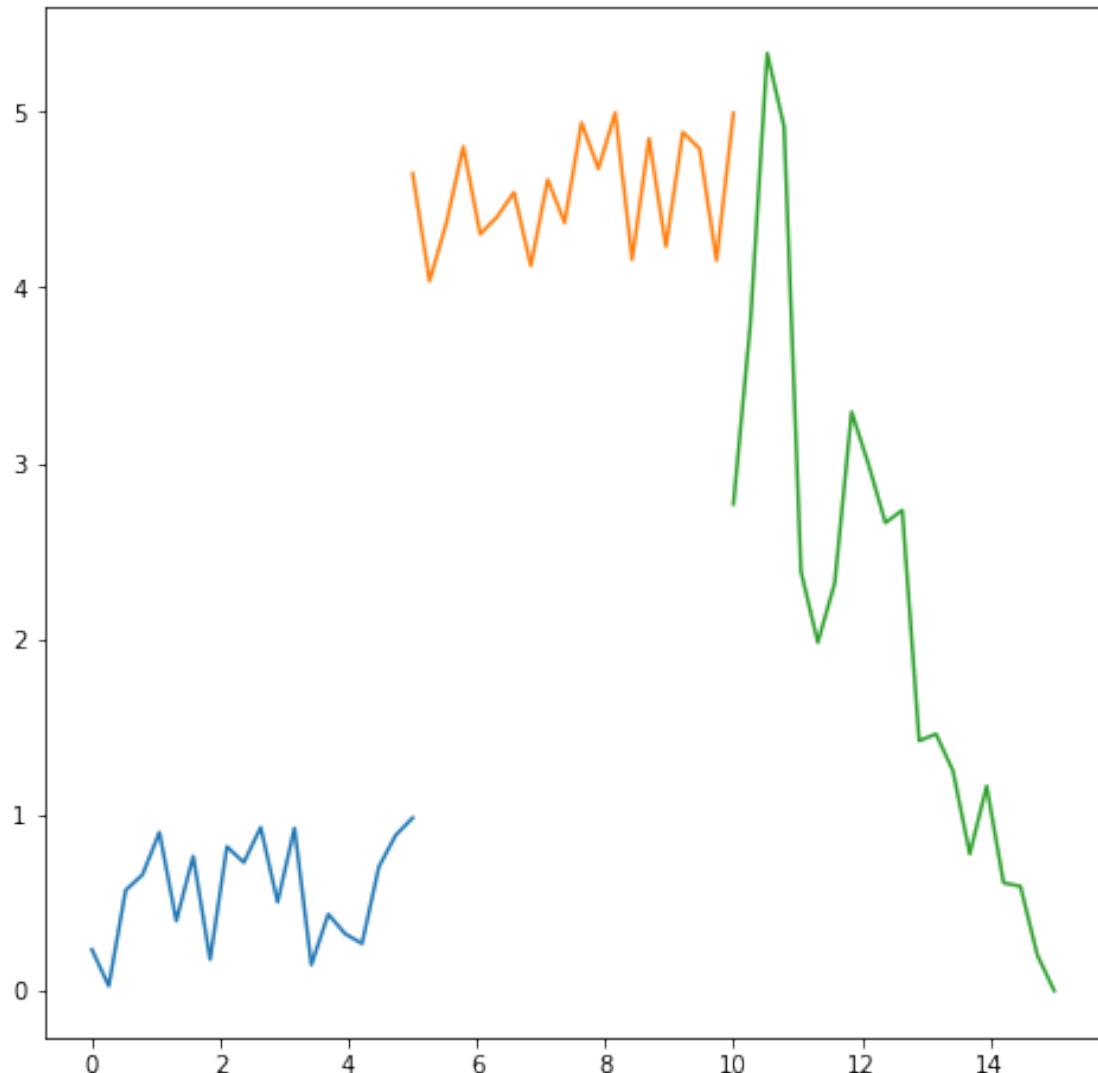
fig = plt.figure(figsize=(8,8))

plt.plot(x1, r1)

```

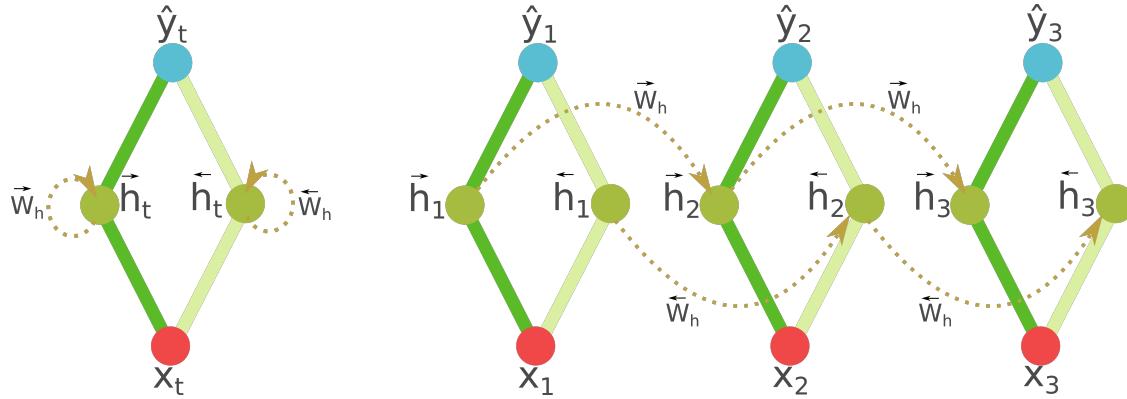
```
plt.plot(x2, r2)
plt.plot(x3, (5-x1)/5*r3)
```

```
[<matplotlib.lines.Line2D at 0x7f6978b8dc10>]
```



To create a bidirectional RNN, only a small modification is needed:

```
Image("img/rnn_bidirectional.png", width=1000)
```



At each time step, an additional “backwards” vector $\overset{\leftarrow}{h}_t$ is added, such that

$$\vec{h}_t = \tanh(\vec{W}_h \vec{h}_{t-1} + \vec{W}_x \mathbf{x}_t + \vec{b}) \quad (143)$$

$$\overset{\leftarrow}{h}_t = \tanh(\overset{\leftarrow}{W}_h \overset{\leftarrow}{h}_{t+1} + \overset{\leftarrow}{W}_x \mathbf{x}_t + \overset{\leftarrow}{b}) \quad (144)$$

$$\hat{y}_t = a(\vec{W}_y \vec{h}_t + \overset{\leftarrow}{W}_y \overset{\leftarrow}{h}_t + \mathbf{c}) \quad (145)$$

In total, 7 weight matrices are needed now. Usually, BiLSTMs are used for these kinds of problems which consist of even more weight matrices.

9 Probabilistic and Statistical Methods

9.1 Bernoulli Distribution

In random experiments with random variables that permit only two outcomes (binary variables), the *probability mass function* is the **Bernoulli distribution**. An example for such an experiment could be a coin (possibly *biased*, such that the probability for it to land on heads isn't the same as the probability for it to land on tails). The state space $X = 0, 1$ consists of two states 0, meaning heads, and 1, meaning tails here. The probability to get tails is $p(x = 1|\mu) = \mu$, where μ here is the given probability to get tails when a coin toss is performed randomly. The probability mass function is

$$\mathfrak{B}(x|\mu) = \mu^x(1 - \mu)^{1-x} \quad (146)$$

where x is either 0 or 1. You can check that this gives the correct probabilities for both heads and tails. The expectation value is $E[x] = \mu$, the variance $V[x] = \mu(1 - \mu)$.

It is possible to estimate the parameter μ for a given observed outcome of an experiment. Given a result $R = \{x_1, x_2, \dots, x_N\}$, where tails was observed m times and heads was observed $N - m$ times, we can calculate the probability to get such a result as

$$p(R|\mu) = \prod_{n=1}^N p(x_n|\mu) = \prod_{n=1}^N \mu^{x_n}(1 - \mu)^{1-x_n} \quad (147)$$

The total probability can be written as this product, since each coin toss is assumed to be independent from any other toss. Large products can cause some numerical errors, so usually the logarithm of this probability is taken (*does this look familiar?*)

$$\log p(R|\mu) = \sum_{n=1}^N \log p(x_n|\mu) = \sum_{n=1}^N [x_n \log(\mu) + (1 - x_n) \log(1 - \mu)] \quad (148)$$

which decomposes the product into a sum of logarithms. To estimate the μ for a given set of results, the derivative of the above formula can be taken and set to zero. The result is

$$\mu_{\text{MaxL}} = \frac{1}{N} \sum_{n=1}^N x_n = \frac{m}{N} \quad (149)$$

This is like optimizing a loss function, where we want to maximize the probability of observing a given result by optimizing μ to μ_{MaxL} . The index here means **maximum likelihood**, the logarithmic probability is the *log-likelihood* and p itself is the *likelihood* we talked about in lecture 01. We will need these notions again later today.

The figure below shows the probability for the maximum likelihood μ corresponding to the observed fraction of tails $\frac{m}{N}$ on the x -axis.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
N = 10
m = np.linspace(0, N, N+1)

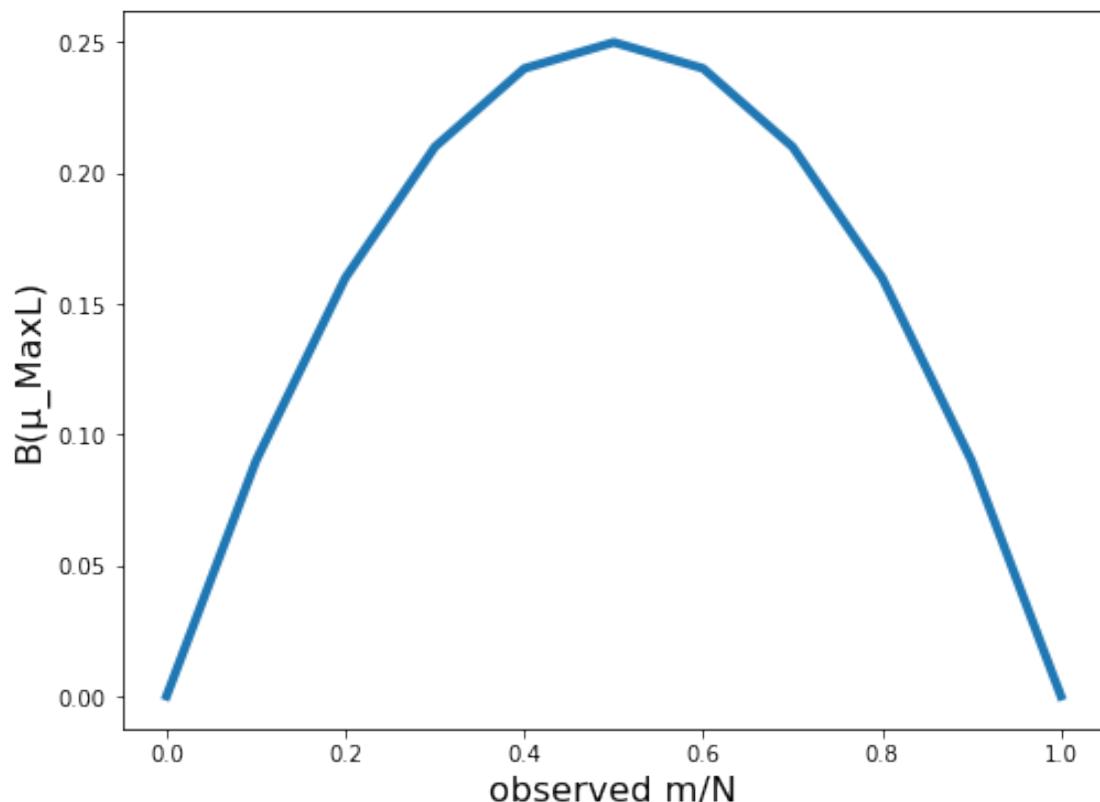
def B():
    return *(1-)

plt.figure(figsize=(8,6))

plt.plot(m/N, B(m/N), lw=4)

plt.xlabel("observed m/N", fontsize=16)
plt.ylabel("B(mu_MaxL)", fontsize=16)

plt.show()
```



9.2 Gaussian/Normal Distribution

The **Gaussian/Normal distribution** for a 1D variable is

$$\mathfrak{N}(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad (150)$$

where μ is the *mean* of the distribution and σ^2 is the *variance* of the distribution. This distribution is ubiquitous, from ground states of quantum harmonic oscillators over IQ test results, parameter distributions in soils and population properties up to the distribution of stars and dust in spiral galaxies. There is a reason for this which we already discussed a bit in assignment 01 and which we'll talk about again at the end of this lesson. It's important to keep in mind that *not everything is Gauss-distributed*.

A way to visualize a distribution is a *histogram*, where *bins* are defined, which are number intervals that are usually the same size and arranged along the x -axis of a histogram. Then, when numbers are drawn from a random distribution, the bar height above such a bin is raised by 1 if the result is inside that interval. For a Gaussian distribution it might look something like this:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact

plt.figure(figsize=(12,7))

def gauss(x, , _square):
    return 1/(2*np.pi*_square)**0.5 * np.exp(-(x- )**2/(2*_square))

def plot_hist( , _square, samples, bins, density):
    plt.cla()
    results = np.random.normal(loc= , scale=np.sqrt(_square), size=samples)
    x = np.linspace( - 4*_square, + 4*_square, 41)

    plt.hist(results, bins=bins, density=density)
    plt.plot(x, gauss(x, , _square))

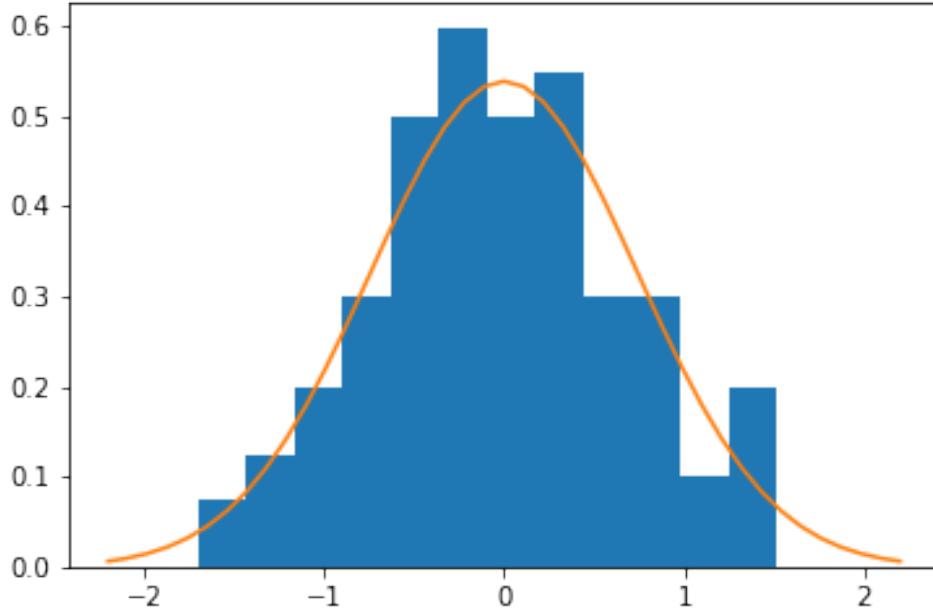
    plt.savefig("img/plot_gauss.png")

interact(plot_hist, =(-5, 5), _square=(0.1, 1.0), samples=(1, 300), bins=(4, 20), density=True)
```

<Figure size 864x504 with 0 Axes>

```
interactive(children=(IntSlider(value=0, description=' ', max=5, min=-5), FloatSlider(value=0.5,
```

```
<function __main__.plot_hist( , _square, samples, bins, density)>
```



The `density` switch creates a normalized histogram divided by the number of total samples, which is what the Gaussian distribution will model. Such a histogram is a good sanity check for many experiments.

For multivariate \mathbf{x} the distribution is

$$\mathfrak{N}(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{\sqrt{2\pi^D \det(\boldsymbol{\Sigma})}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \quad (151)$$

where x and $\boldsymbol{\mu}$ are now vectors and $\boldsymbol{\Sigma}$ is the *covariance matrix*. In multivariate situations, the Gaussian distribution looks like this:

```
%matplotlib notebook
from mpl_toolkits.mplot3d import Axes3D

grid_points = 40

= np.array([0.0, 1.0])
Σ = np.array([[1.0, -0.5], \
              [-0.5, 1.5]])

X = np.linspace(-3, 3, grid_points)
Y = np.linspace(-3, 4, grid_points)
X, Y = np.meshgrid(X, Y)
```

```

def multivar_gauss(vals, , Σ):
    dim = .shape[0]

    det_Σ = np.linalg.det(Σ)
    inv_Σ = np.linalg.inv(Σ)

    norm = np.sqrt((2*np.pi)**dim * det_Σ)
    expo = np.einsum('...k,kl,...l->...', vals-, inv_Σ, vals- )
    #expo = (vals - ) * inv_Σ * (vals - )

    return np.exp(- expo/2)/norm

vals = np.array([X,Y]).transpose(1, 2, 0)
Z = multivar_gauss(vals, , Σ)

fig = plt.figure(figsize=(8,8))
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, rstride=3, cstride=3, linewidth=1, cmap='viridis')

cset = ax.contourf(X, Y, Z, zdir='x', offset=-3, cmap='viridis')
cset = ax.contourf(X, Y, Z, zdir='y', offset=-3, cmap='viridis')
cset = ax.contourf(X, Y, Z, zdir='z', offset=-0.2, cmap='viridis')

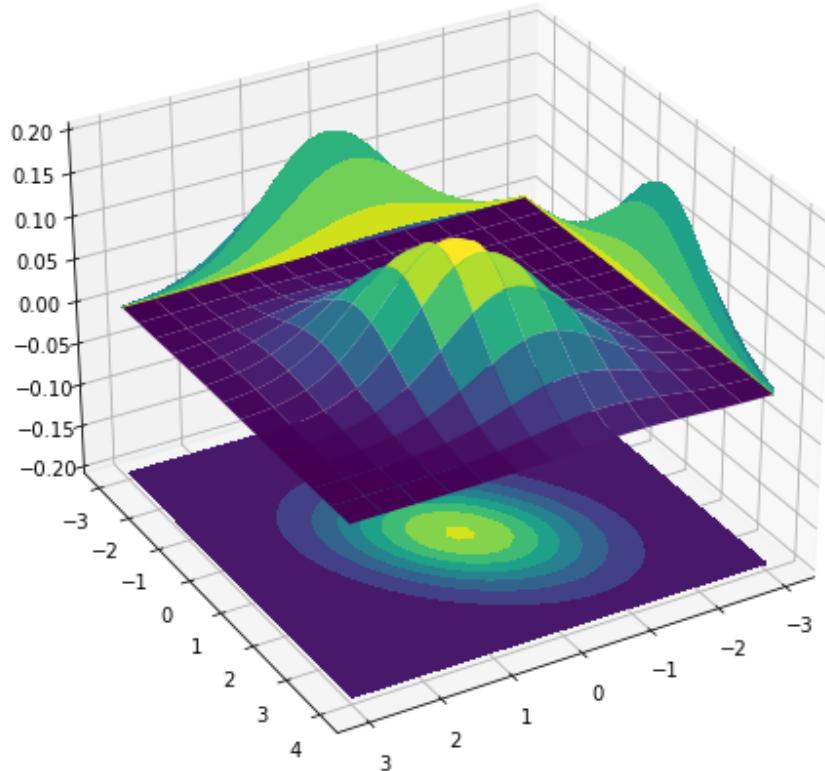
ax.set_zlim(-0.2,0.2)
ax.view_init(30, 60)

plt.savefig("img/plot_multi_gauss.png")

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



The contour plot on the bottom depicts the curves of constant probability (or rather probability density) for two random variables. These random variables could be features from a dataset. In this example, x_1 and x_2 are correlated, which can be seen as a linear trend in the contour plot or in the off-diagonal elements of the covariance matrix (which is usually unknown at first for experimental data). Doing a plot like this can hint at correlations in datasets.

The covariance matrix can generally take on three forms:

```
%matplotlib inline

Σ_general = np.array([[ 1.0, -0.5], \
                      [-0.5,  2.2]])
Σ_diagonal = np.array([[1.0,  0.0], \
                       [0.0,  1.0]])
```

```

        [0.0, 2.2]])
Σ_boring = np.array([[1.0, 0.0], \
                     [0.0, 1.0]])

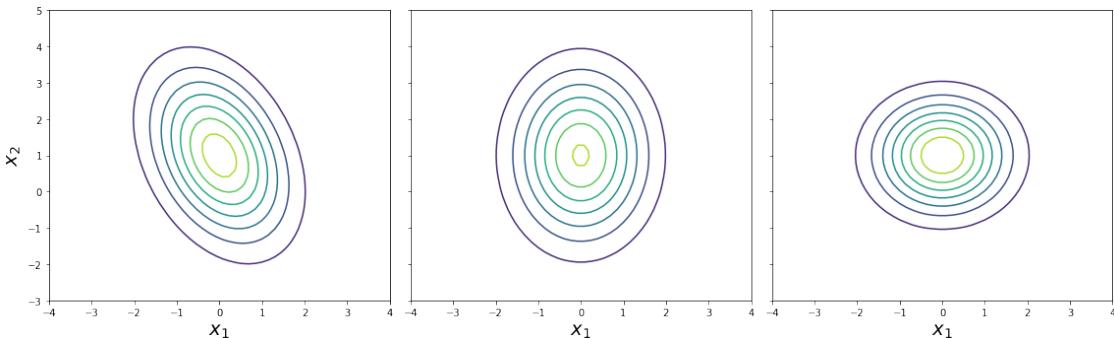
fig, axs = plt.subplots(1, 3, figsize=(16,5), sharey=True)

axs[0].contour(X, Y, multivar_gauss(vals, , Σ_general))
axs[1].contour(X, Y, multivar_gauss(vals, , Σ_diagonal))
axs[2].contour(X, Y, multivar_gauss(vals, , Σ_boring))

for ax in axs:
    ax.set_xlim([-4,4])
    ax.set_ylim([-3,5])
    ax.set_xlabel(r" $x_1$ ", fontsize=20)
    axs[0].set_ylabel(r" $x_2$ ", fontsize=20)

plt.tight_layout()
plt.show()

```



The contours on the left depict the curves of constant probability for a Gaussian distribution with a general covariance matrix with different diagonal entries and off-diagonal entries. Here, *anti-correlation* can be seen. When feature x_1 increases, feature x_2 decreases. The contour plot in the middle depicts the contour lines of a Gaussian distribution for a covariance matrix with different diagonal entries, but no off-diagonal elements. On the right is the same depiction for a covariance matrix with identical diagonal elements.

9.2.1 Gaussian Parameter Estimation

Let's say we want to estimate μ and σ^2 now for a continuous random variable, based on some observation we made. Data from observations is usually discrete (measurements taken at fixed points in time). Effectively, what we want is to maximize the *likelihood* of observing the dataset we're given. Similar to before, the total probability of observing a dataset is

$$p(x|\mu, \sigma^2) = \prod_{n=1}^N \mathfrak{N}(x_n|\mu, \sigma^2) \quad (152)$$

Just like before, we can take the logarithm to get a sum instead

$$\log p(x|\mu, \sigma^2) = \frac{1}{2\sigma^2} \sum_{n=1}^N (x_n - \mu)^2 - \frac{N}{2} \log(\sigma^2) - \frac{N}{2} \log(2\pi) \quad (153)$$

Again, to get the optimal values for μ and σ^2 we need to take the derivative (gradient now actually: $(\partial/\partial\mu, \partial/\partial\sigma^2)$), set it to zero and end up with

$$\mu_{\text{MaxL}} = \frac{1}{N} \sum_{n=1}^N x_n \quad (154)$$

$$\sigma_{\text{MaxL}}^2 = \frac{1}{N} \sum_{n=1}^N (x_n - \mu_{\text{MaxL}})^2 \quad (155)$$

This is **maximum likelihood estimation**.

9.2.2 Multivariate Maximum Likelihood Estimation

We can use the *maximum likelihood technique* to estimate optimal μ and σ^2 for some given data, which makes sense if you can sensibly assume that the data was drawn from a Gaussian distribution.

So given *independent and identically distributed (iid)* data $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$, the log-likelihood is

$$\log p(\mathbf{X}|\boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{1}{2} \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}_n - \boldsymbol{\mu}) - \frac{N}{2} \log \det \boldsymbol{\Sigma} - \frac{ND}{2} \log(2\pi) \quad (156)$$

Again, we can take the gradient with respect to μ and σ^2 and set it to zero to get

$$\boldsymbol{\mu}_{\text{MaxL}} = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \quad (157)$$

$$\boldsymbol{\Sigma}_{\text{MaxL}} = \frac{1}{N} \sum_{n=1}^N (\mathbf{x}_n - \boldsymbol{\mu}_{\text{MaxL}})^T (\mathbf{x}_n - \boldsymbol{\mu}_{\text{MaxL}}) \quad (158)$$

In a certain sense, this is a way to summarize a whole dataset by two terms with $D + \frac{D(D-1)}{2}$ parameters in D feature dimensions, which is still a huge number for, say, 100 features.

Since this would be very difficult to estimate, one usually at least assumes $\boldsymbol{\Sigma}$ to be diagonal, reducing the number of parameters to $D + D$. Sometimes it is even assumed that all diagonal elements of $\boldsymbol{\Sigma}$ are the same, such that $\boldsymbol{\Sigma} = \sigma^2 \mathbf{I}$. It's much easier to get the maximum likelihood estimates of $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$ in these cases.

9.2.3 Central Limit Theorem

We briefly discussed this important theorem in assignment 01. It roughly states that *the distribution of the sum or difference of N independent and identically distributed random variables approaches a Gaussian distribution as N increases*. We have seen this behavior in random die throws by plotting a *histogram* of the throw results which approached a Gaussian curve with increasing number of dice in a single throw.

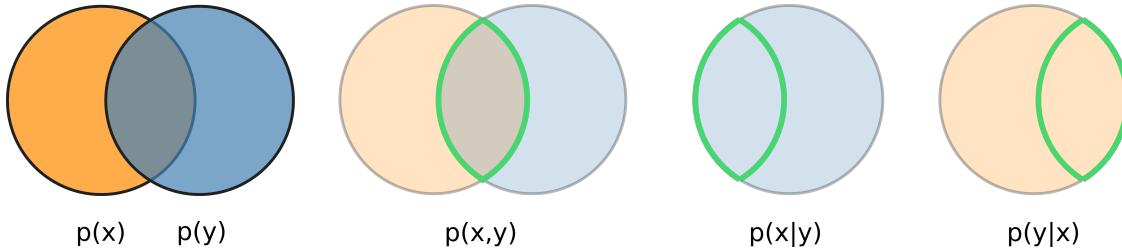
This is a justification for why we usually assume that data points can be modeled as coming from a Gaussian distribution. Each can be considered as a sum of number which have been drawn from a random uniform distribution, such that the result in the follows the normal distribution. A practical application of this fact is a CCD-chip for example, which is the photon detector in most cameras. It can't measure each photon distribution individually, but for a short time, it "counts" the number of photons landing on a specific pixel on the chip. So instead of measuring the individual photon distribution (*side note: this would be a Poisson distribution, which is also the distribution the nuclear decay from assignment 04 follows*), it accumulates the impacts and assumes a Gaussian distribution for calculating the correct color intensity.

(*Side note: the normal distribution maximizes entropy for a given mean and variance.*)

9.3 Naive Bayes Algorithms

First, let's recall a few definitions regarding probability. The following *Venn diagrams* depict different kinds of probabilities we already got to know in lecture 01:

```
from IPython.display import Image
Image("img/venn_probability.png", width=1000)
```



The orange circle by itself is the probability $p(x)$, the blue circle is the probability $p(y)$. The second image is the *joint probability* that both outcomes coincide $p(x,y)$. The third image depicts *conditional probability* $p(x|y)$, which turns out to be the same as $p(y|x)$ in the fourth image. These definitions and the relation $p(x,y) = p(y|x) \cdot p(x) = p(x|y) \cdot p(y)$ lead to *Bayes' theorem*:

$$p(y|x) = \frac{p(x|y) \cdot p(y)}{p(x)} \quad (159)$$

$$\text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}} \quad (160)$$

where the denominator $p(x) = \sum_Z p(x, z) = \sum_Z p(x|z) \cdot p(z)$ is calculated by marginalization (“summing columns or rows”) over all possible outcomes z . The *evidence* is difficult to calculate, since it involves marginalization over information we might not even have access to in a random dataset. Calculating it can be avoided (we’ll see this in a moment), at the cost that the posterior then isn’t a probability anymore, but “only” a **score**. It can be normalized later on though. The *likelihood* here is what we calculated for a dataset in the first lessons today.

Naive Bayes algorithms are a class of *probabilistic classification algorithms*, meaning that it assigns a class to a set of observed features. There are several variants, we will only look at a simple example here.

9.3.1 Training Naive Bayes

For each class c_i calculating the probability $p(c_i|\mathbf{x}) = p(\mathbf{x}|c_i) \cdot p(c_i)$. $p(c|\mathbf{x})$ is basically already the classification, since it depends on the given input \mathbf{x} . For example for three classes c_1, c_2, c_3 we’d produce three expressions

$$p(c_1|\mathbf{x}) = \frac{p(\mathbf{x}|c_1) \cdot p(c_1)}{p(\mathbf{x})} \quad (161)$$

$$p(c_2|\mathbf{x}) = \frac{p(\mathbf{x}|c_2) \cdot p(c_2)}{p(\mathbf{x})} \quad (162)$$

$$p(c_3|\mathbf{x}) = \frac{p(\mathbf{x}|c_3) \cdot p(c_3)}{p(\mathbf{x})} \quad (163)$$

In all these expressions the denominator is the same. So when comparing these probabilities to the expression above without the *evidence*, up to a multiplicative constant they are the same, hence, we can dismiss it.

So after computing the above expressions, we can assign the class c_i that gives the highest *score* to the data point we observed. Calculating *joint probabilities* is difficult, since we have to expand the expressions for each feature in the data set using the *chain rule of probabilities* (where joint probabilities are written in terms of conditional probabilities):

$$p(c_i|\mathbf{x}) = p(x_1, x_2, \dots, x_N|c_i) \cdot p(c_i) \quad (164)$$

$$= p(x_1|x_2, \dots, x_N, c_i) \cdot p(x_2, \dots, x_n, c_i) \cdot p(c_i) \quad (165)$$

$$= \dots \quad (166)$$

The *naive assumption* to solve this problem is to assume that all features are independent of each other (not correlated), which of course in real situations is not at all sensible (we’ll see and abuse this fact in model order reduction techniques like *PCA* for example). Ignoring all possible correlations, the formula from above reduces to

$$p(c_i|\mathbf{x}) = p(x_1|c_i) \cdot p(x_2|c_i) \cdot \dots \cdot p(x_N|c_i) \quad (167)$$

$$= p(c_i) \prod_{n=1}^N p(x_n|c_i) \quad (168)$$

(Side note: in quantum computing, correlations cause entanglement, which can be abused for speeding up some algorithms. Quantum states that are not entangled can be written as product states like above and represent classically interpretable states. The only difference is that quantum states are formulated in complex Hilbert spaces, which entails complex numbers.)

Assigning the class giving the largest score to a set of observed features is called the **MAP (maximum a posteriori) rule**.

Since huge products are computationally unstable (here, the probabilities are smaller than 1, so large products can easily cause underflows), we can use the log-trick from before again to reduce this computation to a sum:

$$\log p(c_i|\mathbf{x}) = \log p(c_i) + \sum_{n=1}^N \log p(x_n|c_i) \quad (169)$$

For an example, let's use Andersen's iris dataset again but convert it into a binary classification problem by saying again that we'd like to pick an *Iris versicolor* and no other species:

```
import pandas as pd
import numpy as np
from sklearn import datasets

iris = datasets.load_iris(as_frame=True)

indices = [ 96, 143, 64, 99, 25, 117, 23, 134, 58, 137, 42, 63]

iris["frame"]["target"].loc[iris["frame"]["target"]==2]=0
iris["frame"].iloc[indices]
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/indexing.py:670:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
```

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
`iloc._setitem_with_indexer(indexer, value)`

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	\
96	5.7	2.9	4.2	1.3	
143	6.8	3.2	5.9	2.3	
64	5.6	2.9	3.6	1.3	
99	5.7	2.8	4.1	1.3	
25	5.0	3.0	1.6	0.2	
117	7.7	3.8	6.7	2.2	
23	5.1	3.3	1.7	0.5	
134	6.1	2.6	5.6	1.4	
58	6.6	2.9	4.6	1.3	
137	6.4	3.1	5.5	1.8	

42	4.4	3.2	1.3	0.2
63	6.1	2.9	4.7	1.4

	target
96	1
143	0
64	1
99	1
25	0
117	0
23	0
134	0
58	1
137	0
42	0
63	1

Here, 1 means “pick” and 0 means “don’t pick”. Out of these 12 samples, 5 are the ones we would pick, so

$$p(c = \text{"pick"}) = \frac{5}{12} \quad (170)$$

$$p(c = \text{"don't pick"}) = \frac{7}{12} \quad (171)$$

Looking at a certain feature, we can infer some knowledge and present it as *probability tables*, as we’ve seen in lecture 01:

```
from IPython.display import display

table1 = pd.DataFrame([["< 3.0", "0/5", "7/7"],
                      [">> 3.0", "5/5", "0/7"]],
                      columns=["sepal width [cm]", "pick", "don't pick"])

table2 = pd.DataFrame([["< 4.3", "3/5", "3/7"],
                      [">> 4.3", "2/5", "4/7"]],
                      columns=["petal length [cm]", "pick", "don't pick"])

display(table1)
display(table2)
```

	sepal width [cm]	pick	don't pick
0	< 3.0	0/5	7/7
1	> 3.0	5/5	0/7

	petal length [cm]	pick	don't pick
0	< 4.3	3/5	3/7

So these tables describe how many times out of the classes we deem worthy of “pick”ing (5 out of the whole dataset), we also observed a sepal width ≤ 3.0 cm. Or how many times out of those classes we deem as belonging to “pick” we observed a petal length < 4.3 cm and so on. These are the probabilities $p(x_2|c_i)$ and $p(x_3|c_i)$. This can be done for every feature constituting the dataset. Then, when a new datapoint $\mathbf{x}^{\text{new}} = (5.8, 3.2, 4.1, 1.3)$ comes in (we found a plant in the woods and want to know whether we should pick it or not), we can calculate

$$p(\text{"pick"}|\mathbf{x}^{\text{new}}) = p(x_1 = 5.8|\text{"pick"}) \cdot p(x_2 = 3.2|\text{"pick"}) \cdot p(x_3 = 4.1|\text{"pick"}) \cdot p(x_4 = 1.3|\text{"pick"}) \quad (172)$$

$$p(\text{"pick"}|\mathbf{x}^{\text{new}}) = p(x_1 > 5.0|\text{"pick"}) \cdot p(x_2 > 3.0|\text{"pick"}) \cdot p(x_3 < 4.3|\text{"pick"}) \cdot p(x_4 < 1.4|\text{"pick"}) \quad (173)$$

$$(174)$$

and

$$p(\text{"don't pick"}|\mathbf{x}^{\text{new}}) = p(x_1 = 5.8|\text{"don't pick"}) \cdot p(x_2 = 3.2|\text{"don't pick"}) \cdot p(x_3 = 4.1|\text{"don't pick"}) \cdot p(x_4 = 1.3|\text{"don't pick"}) \quad (175)$$

$$p(\text{"don't pick"}|\mathbf{x}^{\text{new}}) = p(x_1 > 5.0|\text{"don't pick"}) \cdot p(x_2 > 3.0|\text{"don't pick"}) \cdot p(x_3 < 4.3|\text{"don't pick"}) \cdot p(x_4 < 1.4|\text{"don't pick"}) \quad (176)$$

$$(177)$$

where the product expansion could be done because of the *naive assumption* that all features were independent. Using the probabilities from the lookup tables, we can get the conditional probabilities and multiply them to get a posterior score. The larger score then decides which class the observed data belongs to. This is how we can build a naive Bayes classifier. But there is a problem.

There are zeros in some tables, for example the first one above. Zero entries usually appear when there are no entries for a category, or in other words, when we don’t know the likelihood of a certain category because no such entry existed in our training dataset. Why is this a problem? Since we’re multiplying the conditional probabilities, a single value of zero will make the prediction become zero as well. One way to handle this is to ignore that feature, but for bigger datasets we can’t be sure how important that feature is for a final decision.

A better approach is **Laplace/additive smoothing**. Given an observation $\mathbf{x} = \{x_1, \dots, x_N\}$:

$$\tilde{p}_\alpha(x_2 < 3.0|\text{"pick"}) = \frac{x_2 + \alpha}{M + \alpha N} \quad (178)$$

or in general

$$\tilde{p}_\alpha(x_n|c_i) = \frac{x_n + \alpha}{M + \alpha N} \quad (179)$$

where M is the number of samples in the training set and α is a smoothing parameter. $\alpha = 0$ corresponds to no smoothing at all, where $\tilde{p}(x_n|c_i) = \frac{x_n}{M}$ is simply the empirical probability. The result of smoothing will be between this empirical probability and the uniform probability $\frac{1}{N}$ for large α . In most cases $\alpha = 1$ is chosen and yields good results.

9.4 Maximum Likelihood Estimation in Regression

In regression problems, we were given a training set $\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}$ and formulated a model $h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T \mathbf{x}$ (including a bias weight and $x_0 = 1$), then tried to fit this model via adjusting the weights \mathbf{w} in such a way that a least squares loss function $L = \frac{1}{M} \sum_{m=1}^M (y_i - \mathbf{w}^T \mathbf{x}_i)^2$ was minimized. There is a justification for this procedure coming from the probabilistic approach.

The error in our model is

$$\varepsilon_i^2 = (y_i - \mathbf{w}^T \mathbf{x}_i)^2 \quad (180)$$

This error could be caused for example by noise in experimental measurements or caused by missing data/missing features, in \mathbf{x} as well as y . Assuming the errors are Gauss-distributed, we can model it as

$$\mathfrak{N}(\varepsilon_i^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\varepsilon_i^2}{2\sigma^2}\right) \quad (181)$$

$$p(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - \mathbf{w}^T \mathbf{x}_i)^2}{2\sigma^2}\right) \quad (182)$$

which has zero mean and variance σ^2 . A different interpretation of this is now that we want to maximize this *likelihood* $p(y_i|\mathbf{x}_i, \mathbf{w})$ with respect to \mathbf{x}_i and \mathbf{w} . Again, we can use the log-likelihood to make things easier and get

$$-\log p(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{2\sigma^2} (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + \log(\sqrt{2\pi\sigma^2}) \quad (183)$$

for a single data point. For M data points in the dataset for which we assume they are independent and identically distributed, the exponent will become a product of exponents for each data point and in the end yield the log-likelihood

$$-\log p(y_i|\mathbf{x}_i, \mathbf{w}) = \frac{1}{2\sigma^2} \sum_{i=1}^M (y_i - \mathbf{w}^T \mathbf{x}_i)^2 + M \log(\sqrt{2\pi\sigma^2}) \quad (184)$$

So another way of looking at linear regression is that assuming the error is Gauss-distributed, we want to model the mean and variance of a distribution $p(y|\mathbf{x}, \mathbf{w}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(\mathbf{x} - \mathbf{w})^2}{2\sigma^2}\right)$ to accurately reflect the observed data. This is *maximum likelihood estimation*, like before. Effectively, when we're optimizing a least squares loss function we're assuming a Gaussian distribution of errors, estimating the parameters of the distribution for increasing the likelihood of observing the data we have, given the features we are supplied with.

Doing the same procedure with a *Bernoulli distribution* for a binary classification problem, we would end up with *binary cross-entropy*.

$p(y_i|\mathbf{x}, \mathbf{w})$ is the likelihood of the given data. If a prior is incorporated as an additional factor, we get the *posterior*.

9.5 Maximum a Posteriori Estimation and Regression

The polynomial regression model was given by $y = \mathbf{w}^T \phi(x)$, the training data was supplied in pairs $\mathfrak{D} = \{x^{(i)}, y^{(i)}\}$ with $i = 1, \dots, M$. $\phi(x) = [1, x, x^2, \dots, x^N]$ is a function that creates polynomial features from a scalar variable. The errors are modeled as a Gaussian distribution with zero mean, leading to the likelihood

$$p(y|x, \mathbf{w}) = \prod_{i=1}^M p(y_i|x_i, \mathbf{w}) \quad (185)$$

We can take the negative logarithm again to get the least squares loss

$$-\log p(y|x, \mathbf{w}) = \sum_{i=1}^M (y_i - \mathbf{w}^T \phi(x_i))^2 \quad (186)$$

Using the Bayes rule $p(\mathbf{w}|x, y) = \frac{p(y|x, \mathbf{w}) \cdot p(\mathbf{w})}{p(y|x)}$ we get the *posterior*

$$p(\mathbf{w}|x, y) = \frac{\prod_{i=1}^M p(y_i|x_i, \mathbf{w}) \cdot p(\mathbf{w})}{\sum_{i=1}^M p(y_i|x_i)} \quad (187)$$

where the denominator $p(y_i|x_i)$ is the probability to observe the data we are given and it is constant. It does depend on the given dataset but not on the weights.

Thinking back about polynomial regression, a high polynomial degree lead to *overfitting*. It is desirable that the weights are low in value (although also not too low), such that no feature is weighted too strongly (which would result in *underfitting*). This can be ensured by sampling the weights from a Gaussian distribution with zero mean and some preset variance, or in the multivariate case with a covariance matrix that has constant diagonal values $\Sigma = \alpha^2 \mathbf{I}$. Assuming the weights are drawn from such a distribution, the PDF would look like

$$p(\mathbf{w}) = \mathbf{n} \exp \left(-\frac{\|\mathbf{w}\|^2}{2\alpha^2 \mathbf{I}} \right) \quad (188)$$

where \mathbf{n} is some normalization constant and α^2 some other preset variance.

This will take on the form of a standard Gauss curve, such that the probability to draw extremely large or extremely small values of \mathbf{w} drops to zero exponentially. By adjusting the variance α^2 , we can also control the sharpness of this curve to enforce the weights to be in an even smaller range. Hence, by imposing this kind of prior, we can ensure no under-/overfitting happens. This gives a huge advantage over just assuming a Gaussian distribution for the likelihood.

In the end, we want to model the posterior distribution, estimating \mathbf{w} given x and y . We could express this as the product of likelihood and prior by using the Bayes rule. We can again take the log-likelihood

$$-\log p(\mathbf{w}|x, y) = -\sum_{i=1}^M \log p(y_i|x_i, \mathbf{w}) - \log p(\mathbf{w}) + \text{constants} \quad (189)$$

$$= \sum_{i=1}^M \frac{(y_i - \mathbf{w}^T x_i)^2}{2\sigma^2} + \frac{\|\mathbf{w}\|^2}{2\alpha^2} + \text{constants} \quad (190)$$

This should look familiar. It's the least squares loss function with *L2-regularization*, which we used to regularize our model to have less extreme weights in lecture 04. So this is the same as evaluating the posterior score, or **maximum a posteriori estimation**. The problem of finding the parameters that maximize the posterior can be solved by using gradient descent for example. The value of \mathbf{w} that will be found is the **MAP estimate**. The Gaussian distribution can also be rewritten as having $(\mathbf{w} - \tilde{\mathbf{w}})^T \Sigma^{-1} (\mathbf{w} - \tilde{\mathbf{w}})$ as the exponent. Following the procedure from the second lesson today, we could directly estimate the covariance and the weight adjustments as

$$\Sigma^{-1} = \frac{\phi^T \phi}{\sigma^2} + \frac{1}{\alpha^2} \quad (191)$$

$$\tilde{\mathbf{w}} = \frac{\Sigma \phi^T y}{\sigma^2} \quad (192)$$

where Σ depends on the data and the constraint α^2 from the prior, the weight adjustments $\tilde{\mathbf{w}}$ depend on Σ . $\tilde{\mathbf{w}}$ is the same as the *MAP estimate*, so we get a value for it and use it to calculate the output estimate $\hat{y} = h$ for every new test data point.

9.6 Bayesian Regression

A different approach to *MAP estimation* is performing a iterative form of it. We'll predict the distribution of \hat{y} with an error estimate on it. The images below describe what is going on for a linear model $h(x; \mathbf{w}) = w_0 + w_1 x$.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.covariance import EmpiricalCovariance
from sklearn.preprocessing import PolynomialFeatures
from scipy.stats import multivariate_normal

# polynomial degree for everything
poly_deg = 3

# Training dataset sizes for which to calculate the posteriors
# The last entry is the number of training samples
sample_evals = [0, 1, 3, 20]
```

```

# Variances for errors and weights
_square = 0.04
_square = 0.5

# Training data restricted to range -1, 1
X_train = 2*np.random.rand(sample_evals[-1], 1) - 1
# Test data in the same range
test_size = 40
X_test = np.linspace(-1, 1, test_size).reshape(-1, 1)

# Ground truths for training data
w_0 = -0.25
w_1 = 0.6
# Sampled from a linear function with added noise, like before
# "scale" expects the standard deviation
y_train = w_0 + w_1 * X_train**poly_deg + np.random.normal(scale=np.
    ↪sqrt(_square), size=X_train.shape)
# Function values without noise
y_true = w_0 + w_1*X_test**poly_deg

# Feature vector for the test data, see lecture 04
_xtest = np.c_[np.ones(test_size), X_test]**poly_deg

# Number of pairs (w0, w1) to be drawn from posterior
weight_size = 5

# Create appropriate plotting landscape
fig, axs = plt.subplots(len(sample_evals), 3, figsize=(15, 15))

# Plot setup, these are the grid points to be plotted on
grid_points = 40
X_g = np.linspace(-1, 1, grid_points)
Y_g = np.linspace(-1, 1, grid_points)
#X_g, Y_g = np.meshgrid(X_g, Y_g)
flat_g = np.dstack(np.meshgrid(X_g, Y_g)).reshape(-1, 2)

# Loop over all sample sizes in "sample_evals"
# for each sample size, the posterior is calculated,
# five sample weight pairs chosen and plotted
# and the resulting prediction as well as standard
# deviation is plotted
for i, samples in enumerate(sample_evals):
    # Take the desired number of samples as training data
    X_samples = X_train[:samples]
    y_samples = y_train[:samples]

```

```

# Feature vector for these samples
_x_samples = np.c_[np.ones(samples), X_samples]**poly_deg

# Covariance matrix and mean of the posterior, as in last lesson (MAP rule)
Σ_samples = np.linalg.inv(1/_square * np.eye(_samples.shape[1]) + 1/
↪_square * _samples.T @ _samples)
↪_samples = 1/_square * Σ_samples @ _samples.T @ y_samples

# Mean of posterior to predict y_hat = y_mean
# and an "error bar" in form of its variance (see last lesson)
y_mean = _test @ _samples
y_var = _square + np.sum(_test @ Σ_samples * _test, axis=1)

# Choose 5 samples from the posterior
w_sampled = np.random.multivariate_normal(_samples.ravel(), Σ_samples, 5).T
# Calculate corresponding y values
y_sampled = _test @ w_sampled

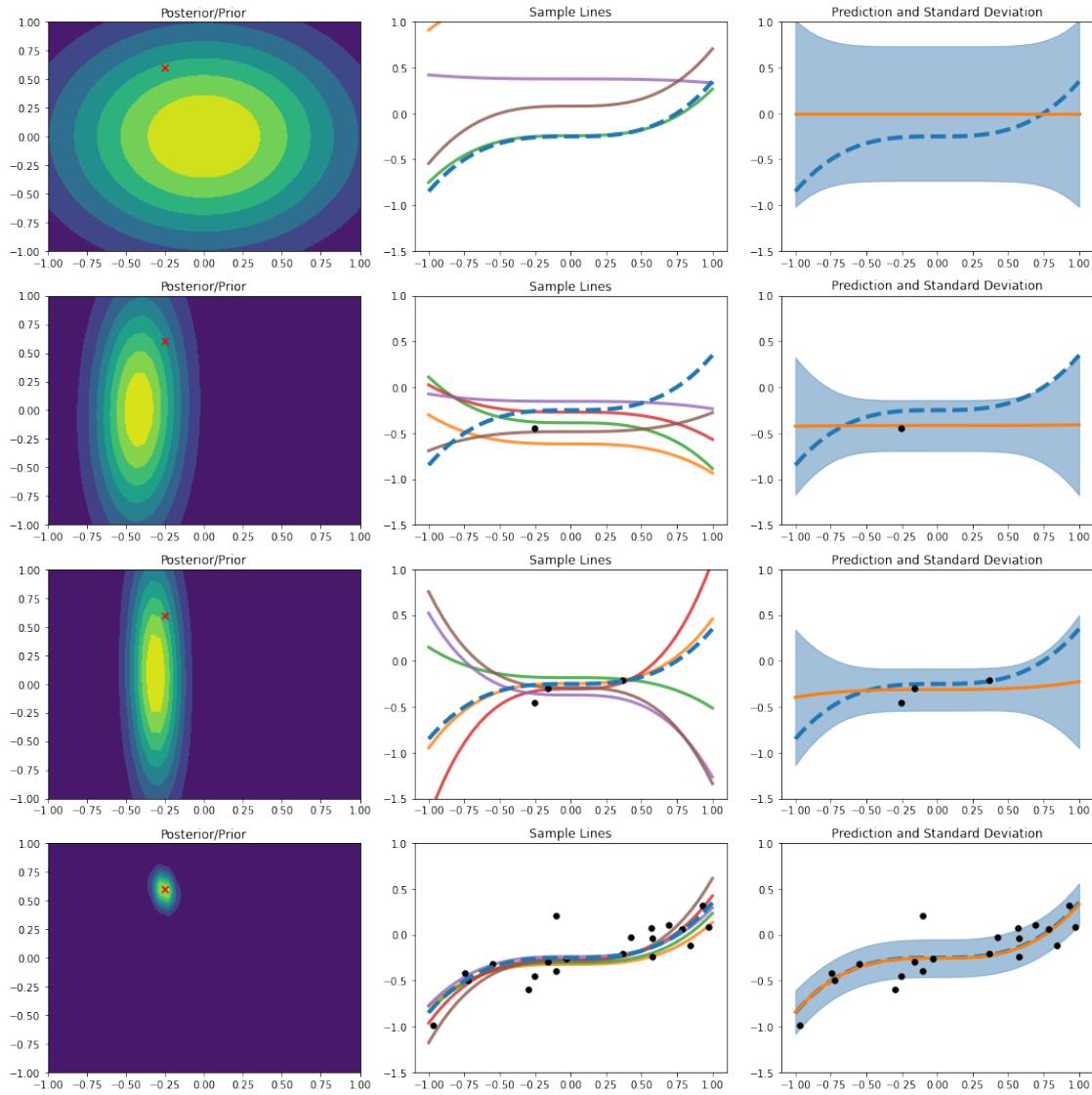
# Plot posterior
axs[i, 0].contourf(X_g, Y_g, multivariate_normal.pdf(flat_g, mean=_samples.
↪ravel(), cov=Σ_samples).reshape(grid_points, grid_points))
axs[i, 0].scatter(w_0, w_1, marker='x', c="red", s=40)
axs[i, 0].set_title("Posterior/Prior")
#axs[i, 0].legend()

# Plot the 5 chosen sample lines
axs[i, 1].scatter(X_samples, y_samples, marker='o', c="black", s=30)
axs[i, 1].plot(X_test, y_true, linestyle='--', lw=4)
for k in range(weight_size):
    axs[i, 1].plot(X_test, y_sampled[:, k], lw=3, alpha=0.8, zorder=-20)
axs[i, 1].set_ylim([-1.5, 1.0])
axs[i, 1].set_title("Sample Lines")

# Plot prediction y_mean and the standard deviation
# sqrt(y_var) as "continuous" error bars
axs[i, 2].scatter(X_samples, y_samples, marker='o', c="black", s=30, ↪
zorder=20)
axs[i, 2].plot(X_test, y_true, linestyle='--', lw=4)
axs[i, 2].plot(X_test, y_mean, lw=3)
std = np.sqrt(y_var)
axs[i, 2].fill_between(X_test.ravel(), y_mean.ravel() + std.ravel(), y_mean.
↪ravel() - std.ravel(), color='steelblue', alpha=0.5)
axs[i, 2].set_ylim([-1.5, 1.0])
axs[i, 2].set_title("Prediction and Standard Deviation")

plt.tight_layout()

```



In the first step we assume a standard Gaussian distribution of weights from which here we draw 5 samples of weights w_0 and w_1 . Using the linear hypothesis/model above, we can draw 5 lines, representing this choices of weights. These are the *predictions*. The distributions on the left are the *postriors*, which will be the *priors* for consecutive estimations. So when we have drawn the 5 prediction lines, we get a corresponding \hat{y}_i for each x_i in our test set. This is our *data space*. Given x and y for various weights we can calculate the *likelihood* $p(y|\mathbf{w}, x) = \exp\left(-\frac{(y-\mathbf{w}^T x)^2}{2\sigma^2}\right)$. The next step is to multiply the likelihood with the prior (the posterior from last step) to produce the new posterior. Again, we draw 5 samples (which are now much more restricted) and draw the corresponding lines, depicting the new data space. Now we observe more datapoints and do the same procedure again and again. Eventually, the posterior distribution will become very sharp and lead to lines in the data space that are very close to each other, such that observing a new x_i will have no impact on the error made. This is the point where \mathbf{w} is completely determined by the available training data. In the end, the arg max determines the w_0 and w_1 which produce the MAP

estimate.

The MAP estimate is only a point estimate, producing a posterior distribution for the error in the estimation of \mathbf{w} . The prediction is made for new test data $x_{\text{test}}, y_{\text{test}}$. To determine the predictive distribution fully, one can use *Bayesian regression*.

In the end, what we would like to find is the probability distribution for the error in estimating y_{test} given the training data x, y and the corresponding x_{test} :

$$p(y_{\text{test}}|x_{\text{test}}, x, y) = \int p(y_{\text{test}}, \mathbf{w}, x_{\text{test}}, x, y) d\mathbf{w} \quad (193)$$

$$= \int p(y_{\text{test}}|x_{\text{test}}, x, y, \mathbf{w}) p(\mathbf{w}|x, y) d\mathbf{w} \quad (194)$$

$$= \int p(y_{\text{test}}|x_{\text{test}}, \mathbf{w}) p(\mathbf{w}|x, y) d\mathbf{w} \quad (195)$$

where the second line was inferred using the *sum and product rules of probability* (recall $p(x) = \sum_y p(x, y)$ and $p(x, y) = p(y|x)p(x)$) and by introducing \mathbf{w} as a random variable. For continuous weight distributions, the sums become integrals (in the Riemann sense) like above. In this reformulation, we can get rid of the training data x, y in the first term, since the information about it is contained wholly in the weights \mathbf{w} , which is why we could leave out the dependency on x, y in the last line. The rightmost term is a *posterior* for the weights, which we can model as a Gaussian like before. The middle term is assumed to be Gaussian. The product or *convolution* of Gaussians is again a Gaussian (we'll omit the proof here), which means that the term on the left will be a Gaussian distribution.

Now, if we want to estimate the mean $\bar{f}(x) = \int f(x)p(x)dx$ of some function $y = f(x)$, we can estimate the corresponding probability density function $p(x)$ by looking at training data. This $p(x)$ here is a posterior, that makes it possible to calculate the mean over all possible realizations of weights \mathbf{w} , given some training data x, y . Then, an arg max finds the value of y_{test} for which this probability distribution becomes maximal. The same approach works for the variance $\text{var}[f(x)] = \int (f(x) - \bar{f}(x))^2 p(x) dx$.

Using this approach, we can calculate the mean and variance for every new test point provided to us, under the assumption that both are Gauss-distributed. The integrals in the equations above are usually difficult to examine, which is why that assumption was made here. If some other distributions are assumed, the integrals are usually solved with Monte-Carlo methods.

Below, you can find an output of an example Gaussian regression, (the code was taken from the [scikit-learn documentation](#) and adapted):

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

from sklearn.linear_model import BayesianRidge

# This fixes the random number generator state
# Uncomment to get randomness
```

```

np.random.seed(1)

# The function we're approximating
def f(x, noise_amount):
    y = np.sqrt(x) * np.sin(x)
    noise = np.random.normal(0, 1, len(x))
    return y + noise_amount * noise

# Maximal polynomial degree used in regression
degree = 10

# Create dataset and ground truth
X = np.linspace(0, 10, 30)
y = f(X, noise_amount=0.2)

# Setup the Bayesian polynomial regression
clf_poly = BayesianRidge()

# Fit the model, vander() creates polynomial features (Vandermonde matrix)
clf_poly.fit(np.vander(X, degree), y)

# Create data to plot
X_plot = np.linspace(0, 11, 25)
y_plot = f(X_plot, noise_amount=0)

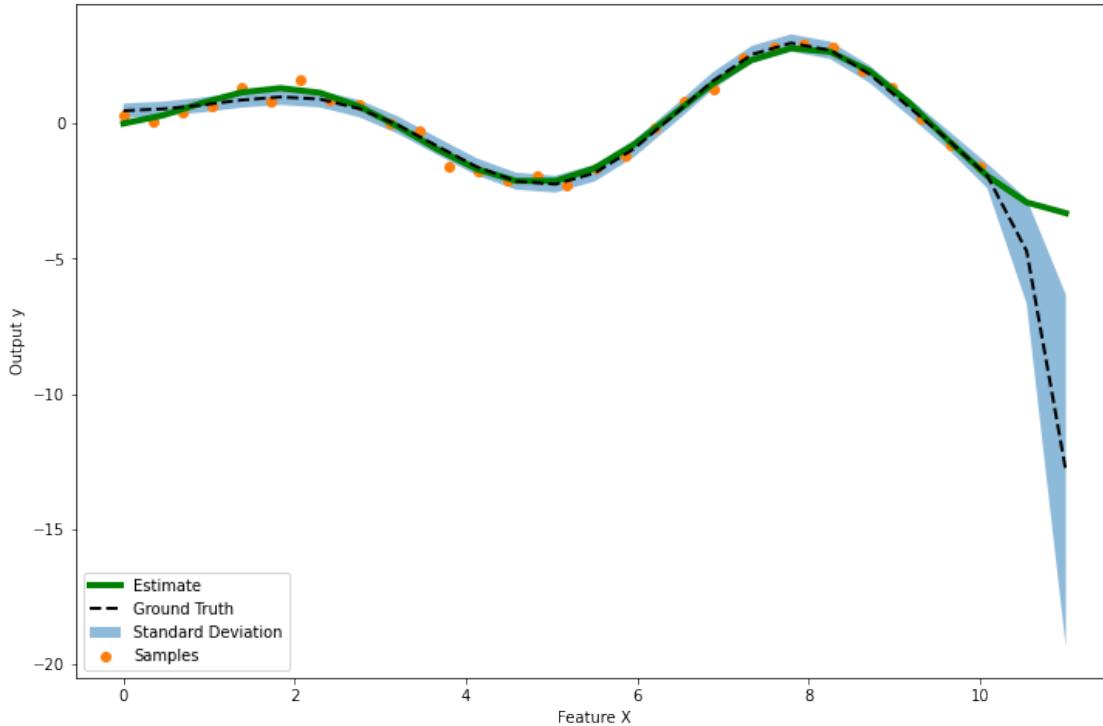
# Get predictions and standard deviations for errors
y_mean, y_std = clf_poly.predict(np.vander(X_plot, degree), return_std=True)

# Plotting setup
plt.figure(figsize=(12, 8))

# Plot ground truth, prediction, "continuous" error bars
plt.plot(X_plot, f(X_plot, 0), color='green', lw=4, label="Estimate")
plt.plot(X_plot, y_mean, color='black', linewidth=2,
         linestyle='--', label="Ground Truth")
plt.fill_between(X_plot, y_mean+y_std, y_mean-y_std, alpha=0.5, label="Standard_U_Deviation")
plt.scatter(X, y, label="Samples")

plt.ylabel("Output y")
plt.xlabel("Feature X")
plt.legend(loc="lower left")
plt.show()

```



The data points were sampled from a sine curve with added noise, like we did in lecture 02. The dashed black curve is the ground truth, the green curve is the prediction from Bayesian regression. The shaded area covers one standard deviation and gives a notion of *error*.

9.7 Advanced Regularization

9.7.1 Early Stopping

The phenomenon of *overfitting* is tightly connected to the *capacity* of a model. The larger a model, the higher the probability of overfitting. The longer such a model is trained, the more likely overfitting will become eminent. Sometimes it's possible to set a threshold for either the loss function (rarely) or some other *metric* (usually), like the *accuracy* for a classification model, and then stop the training early. The way to do so is by using a *callback function*, like we used in lecture 04 for *Bayesian hyperparameter optimization* and `tensorboard`.

We could program our own callback function like we did in lecture 04, but Keras already provides a callback for early stopping. You can see how it's used below:

```
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.backend import clear_session
from tensorflow import one_hot
```

```

clear_session()

X = np.load("/data/X.npy", allow_pickle=False).astype(float)[:1000]
y = np.load("/data/y.npy", allow_pickle=True).astype(int)[:1000]
y_oh = one_hot(y, 10).numpy()

X_train, X_test, y_train, y_test = train_test_split(X, y_oh, test_size=0.2)

# this initializes the early stopping function
early_stop = EarlyStopping(monitor='val_accuracy',
                           min_delta=0.03, # difference in metric that counts
→as no improvement
                           patience=3, # number of epochs with no
→improvement that are tolerated
                           verbose=1, # print some information
                           mode='min', # whether baseline value is a
→maximum or minimum, can be 'auto'
                           baseline=0.8, # training will be stopped if this
→isn't reached
                           restore_best_weights=False) # whether to restore
→the best weights monitored

model = Sequential()

model.add(Dense(400, input_shape=(X_train.shape[1],), activation="relu"))
model.add(Dense(256, activation="relu"))
model.add(Dense(10, activation="softmax"))

model.compile(optimizer="adam", loss="categorical_crossentropy",
→metrics=["accuracy"])

```

The callback is provided for the training:

```

history = model.fit(X_train, y_train, validation_split=0.25, epochs=10,
→callbacks=early_stop)

```

```

Epoch 1/10
19/19 [=====] - 1s 29ms/step - loss: 68.2140 -
accuracy: 0.3514 - val_loss: 16.8869 - val_accuracy: 0.6600
Epoch 2/10
19/19 [=====] - 0s 6ms/step - loss: 6.2108 - accuracy:
0.8732 - val_loss: 7.2124 - val_accuracy: 0.8000
Epoch 3/10
19/19 [=====] - 0s 6ms/step - loss: 1.4449 - accuracy:
0.9462 - val_loss: 7.2189 - val_accuracy: 0.8250
Epoch 4/10

```

```
19/19 [=====] - 0s 6ms/step - loss: 0.2275 - accuracy: 0.9760 - val_loss: 5.9999 - val_accuracy: 0.8250
Epoch 00004: early stopping
```

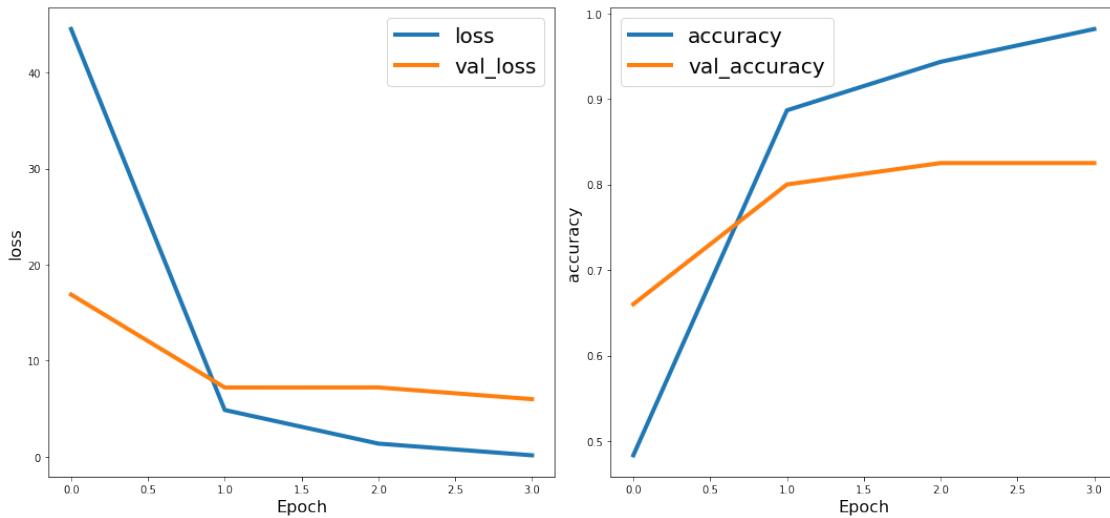
You should see that early stopping steps in at around epoch 3-4 to stop the training. We can check the observed quantities in the history plot:

```
import matplotlib.pyplot as plt

fig,axs = plt.subplots(1,2, figsize=(15,7))

for i,key in enumerate(history.history.keys()):
    axs[i%2].plot(history.history[key], lw=4, label=key)
    axs[i%2].legend(fontsize=20)
    axs[i%2].set_xlabel("Epoch", fontsize=16)
    axs[i%2].set_ylabel(key.split("_")[-1], fontsize=16)

plt.tight_layout()
```



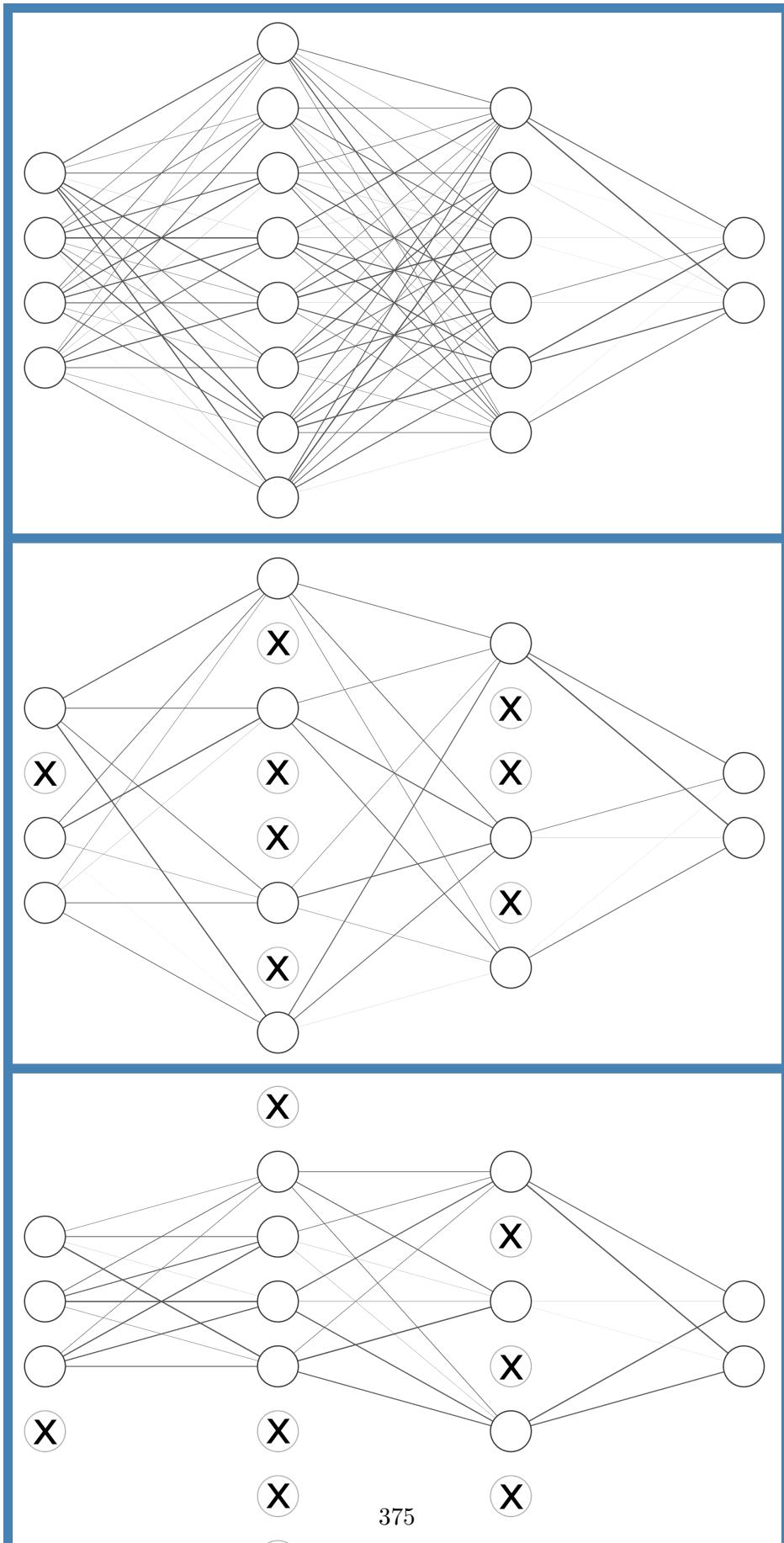
Obviously it's difficult to determine a sensible threshold, which may not even be reached in all cases, but this is still a widely used and useful method to reduce the likelihood of overfitting.

9.7.2 Dropout

A remarkably efficient and effective method to regularize an ANN is **dropout**, where the idea is to randomly deactivate a ratio p of neurons during training and backpropagation, such that all incoming and outgoing connections from that unit are removed from the architecture for an epoch. This emulates training a whole bunch of different models, since deactivating a unit produces a (slightly) different architecture. Dropout is performed layerwise, so the way to incorporate the technique would be to add a **Dropout** layer after some other layer, which will then be affected. To accommodate the missing neurons, all activations in the affected layer are scaled down by the

dropout rate p .

```
from IPython.display import Image  
Image("img/dropout_ann.png", width=400)
```



The end result of training with dropout is some sort of “average” over the different models, which produces the regularizing effect. No weight can become too large consistently, since weights are randomly excluded from weight updates. Usual values for the dropout rate are $p = 0.5$ for hidden layers and $p = 0.2$ for visible layers (input and output layers). This regularizing effect is bought with increasing convergence times (some sources estimate a doubling of the convergence time), but each epoch ends slightly faster, since a lot of units are randomly dropped.

Adding dropout layers in keras is straightforward:

```
from tensorflow.keras.layers import Dropout

clear_session()

dropout_model = Sequential()

dropout_model.add(Dense(400, input_shape=(X_train.shape[1],), activation="relu"))
dropout_model.add(Dropout(0.5))
dropout_model.add(Dense(256, activation="relu"))
dropout_model.add(Dropout(0.5))
dropout_model.add(Dense(10, activation="softmax"))

dropout_model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
```

The model is trained as usual:

```
dropout_history = dropout_model.fit(X_train, y_train, validation_split=0.25, epochs=50, callbacks=early_stop)
```

```
Epoch 1/50
19/19 [=====] - 1s 15ms/step - loss: 26.3573 - accuracy: 0.1790 - val_loss: 2.3943 - val_accuracy: 0.6250
Epoch 2/50
19/19 [=====] - 0s 7ms/step - loss: 6.3476 - accuracy: 0.3624 - val_loss: 1.1240 - val_accuracy: 0.7050
Epoch 3/50
19/19 [=====] - 0s 7ms/step - loss: 3.3786 - accuracy: 0.5076 - val_loss: 0.7654 - val_accuracy: 0.7250
Epoch 4/50
19/19 [=====] - 0s 7ms/step - loss: 2.1677 - accuracy: 0.5775 - val_loss: 0.6864 - val_accuracy: 0.7600
Epoch 0004: early stopping
```

The baseline for early stopping is set to low to see much of a difference:

```

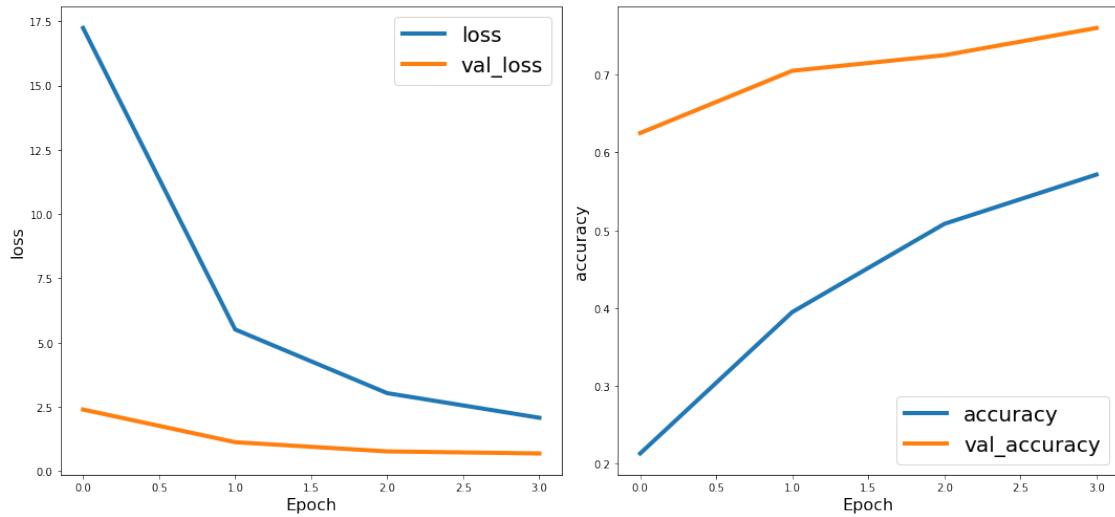
import matplotlib.pyplot as plt

fig,axs = plt.subplots(1,2, figsize=(15,7))

for i,key in enumerate(dropout_history.history.keys()):
    axs[i%2].plot(dropout_history.history[key], lw=4, label=key)
    axs[i%2].legend(fontsize=20)
    axs[i%2].set_xlabel("Epoch", fontsize=16)
    axs[i%2].set_ylabel(key.split("_")[-1], fontsize=16)

plt.tight_layout()

```



The AlphaDropout variant that is used in conjunction with *SELU* activation is the same approach, but additionally normalizes the activations in the affected layer to zero mean and unit variance.

10 Dimensionality Reduction and Clustering

10.1 The Curse of Dimensionality

We talked about how to identify and generate *features* for machine learning algorithms, which are the inputs for the methods and are an extremely important part of the complete machine learning pipeline. Apart from simply using available data x_1, x_2, \dots, x_N from, say, measurements in an experiment, we can also construct new features as combinations like $x_1 \cdot x_2$ or functions of the original features like $\sin(x_1)$. Including these in the analysis can be beneficial for reducing convergence time or even finding a solution at all. A simple example can be tested on the [TensorFlow Playground](#). When changing the activation to linear (otherwise it will almost always find a suitable mapping) for the XOR problem, only using x_1 and x_2 as the features will not converge at all. Additionally using $\sin(x_1)$ and $\sin(x_2)$ creates at least a partial classification of the whole dataset (do you know why this classification does not work?). The number of features is generally referred to as the *dimension* of a problem.

A problem that will keep coming up in machine learning settings is how many data points to generate (or measure) for training, or, making the most out of the available data points. This is related to how many features a dataset is comprised of. The more features (dimensions) are in a dataset, the “exponentially more” samples are needed to accurately probe or *sample* that space. This is an instance of **the curse of dimensionality**, where adding a linear number of dimensions creates demand for an exponential increase in some resource(s), in this case the number of samples. You can play around with the number of samples used for probing 1D, 2D and 3D space below:

```
%matplotlib notebook
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from ipywidgets import interact

samples = np.random.rand(501)

def plot_samples(data_size=2):
    fig = plt.figure(figsize=(10,4))
    ax0 = fig.add_subplot(131)
    ax1 = fig.add_subplot(132)
    ax2 = fig.add_subplot(133, projection='3d')

    ax0.cla()
    ax1.cla()
    ax2.cla()

    ax0.set_title("1D")
    ax1.set_title("2D")
    ax2.set_title("3D")

    ax0.set_xlim([0,1])
    ax0.set_ylim([0,1])
```

```

ax1.set_xlim([0,1])
ax1.set_ylim([0,1])
ax2.set_xlim([0,1])
ax2.set_ylim([0,1])
ax2.set_zlim([0,1])

ax0.hlines(0.5, 0, 1, color='gray', alpha=0.5, lw=4)
ax0.scatter(samples[:data_size], data_size*[0.5], lw=0.1)

ax1.scatter(samples[:data_size], samples[-data_size:])

ax2.scatter(samples[:data_size], samples[-data_size:], ↴
samples[-10-data_size:-10])

plt.tight_layout()

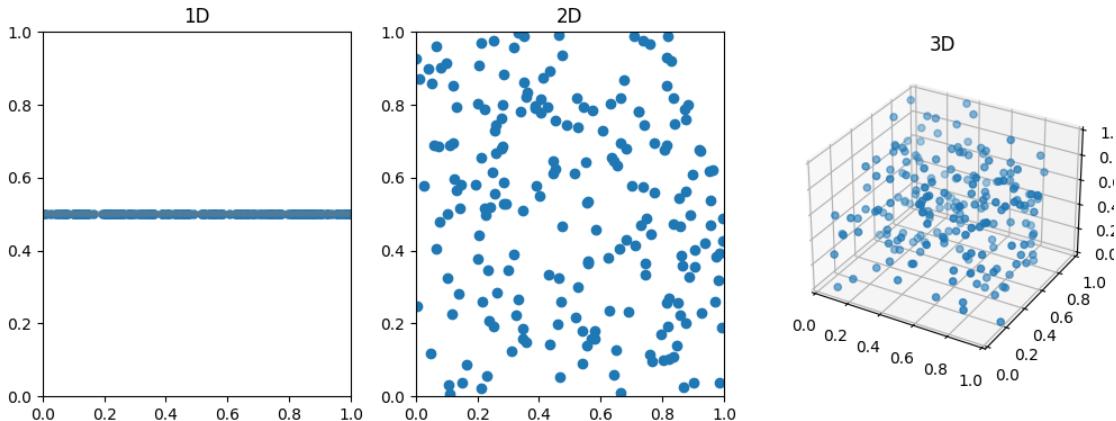
plt.savefig("img/plot_dimensions.png")

interact(plot_samples, data_size=(2, 500))

```

```
interactive(children=(IntSlider(value=2, description='data_size', max=500, min=2), Output()), _
```

```
<function __main__.plot_samples(data_size=2)>
```



Roughly speaking, when we decide that 10 samples are sufficient for describing the 1D space above, the 2D space would need around 100 samples. For the 3D space, already around 1000 datapoints are needed. This is just a coarse estimate, but here, we'd need 10^d samples for d dimensions ($= d$ features) in the dataset. Usually, we're not in the position to get arbitrary amounts of data to probe these spaces, so including more features must be carefully planned with respect to the available data. The situation in most cases is that we're given a fixed number of data points, so adding more

dimensions to the problem will make the dataset more *sparse* regarding the problem.

A solution, or at least a way to ease this problem, is **dimensionality reduction** or **model order reduction**. Theoretically, adding more features for an analysis should increase the predictive capabilities of machine learning models, but adding features also requires taking a lot more data into the training phase. The idea of dimensionality reduction is to represent the available data by fewer features than originally present.

The naive approach would be to simply discard some features and forget about them. This approach is called *feature elimination* and makes sense when it's very obvious that certain features do not play any role in the problem at hand, e.g., temperature for temperature-independent phenomena, or when it doesn't influence the dataset at all. An analytical way to do so is to perform a **principal component analysis**, which we'll come to later today.

10.2 Eigendecomposition

Before moving on to principal component analysis, we should take a quick gander at eigendecomposition and its generalization to non-square matrices, as it will make things much easier to understand from a linear algebra perspective.

All *normal* matrices \mathbf{A} , which means they satisfy the identity $\mathbf{AA}^T = \mathbf{A}^T\mathbf{A}$, can be eigendecomposed. This includes orthogonal, symmetric, and skew-symmetric matrices. The physical interpretation is that every such matrix can be thought of as a combination of a rotation and stretching, sometimes also a reflection (when the determinant is negative). See for example what the matrix

$\mathbf{A} = \begin{bmatrix} 1.5 & 0.5 \\ 0.5 & 1.0 \end{bmatrix}$ does to a square:

```

import numpy as np
import matplotlib.patches as patches
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8,8))
ax = fig.add_subplot(111, aspect='equal')
ax.set_xlim(-0.1, 2.0)
ax.set_ylim(-0.1, 2.0)

# vectors
e1 = np.array([0.0, 1.0])
e2 = np.array([1.0, 0.0])
origin = np.array([0, 0])

# matrix is defined here
A = np.array([[1.5, 0.5],
              [0.5, 1.0]])

print("Is A normal?", "Yes." if np.all(A.T @ A == A @ A.T) else "No")

# matrix applied to starting vectors
v1 = A @ e1

```

```
v2 = A @ e2

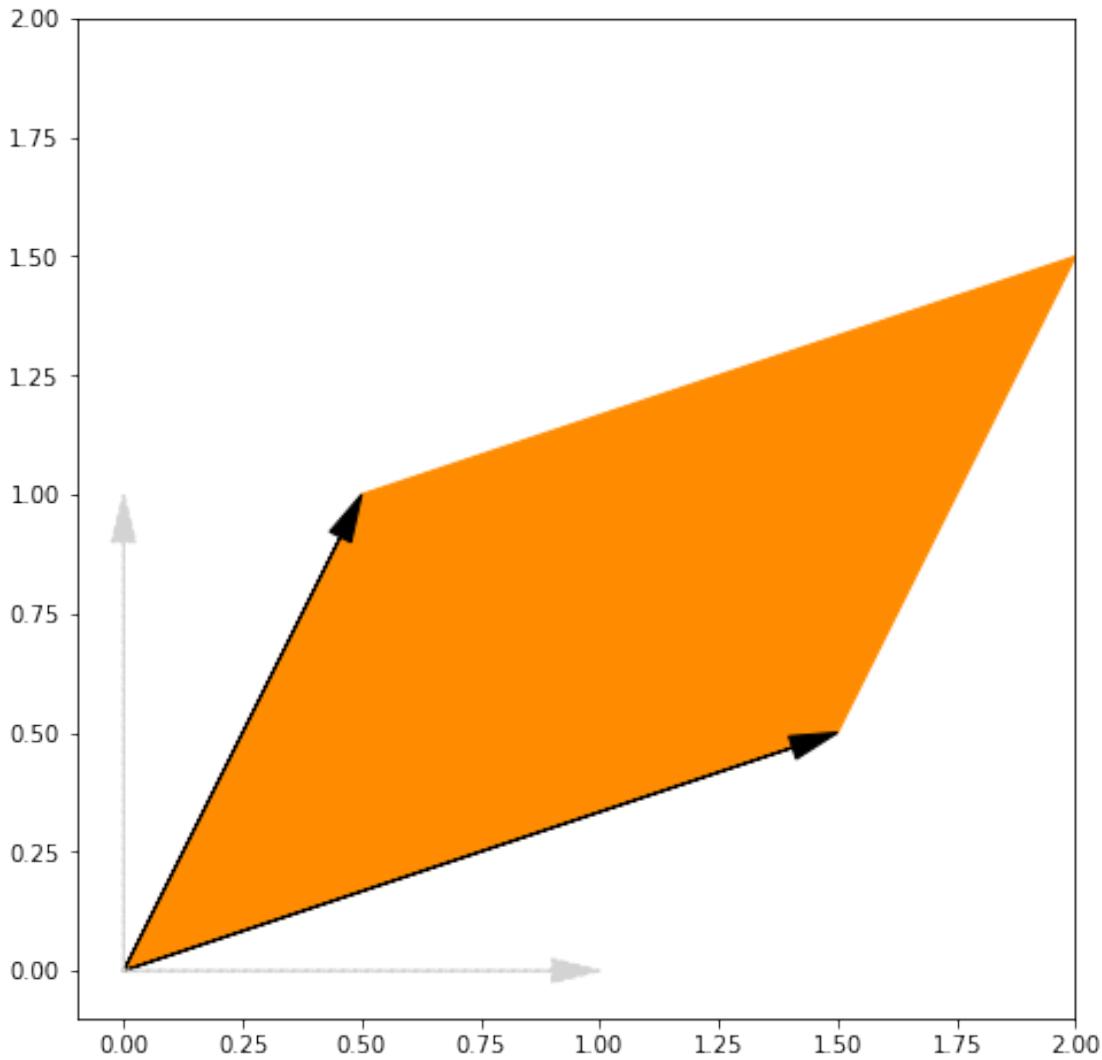
# we need the coordinates of the endpoints
ax.add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], color="darkorange"))

ax.arrow(*origin, *(v1), head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)
ax.arrow(*origin, *(v2), head_width=0.05, head_length=0.1, \
         fc='k', ec='k', length_includes_head=True)

ax.arrow(*origin, *e1, head_width=0.05, head_length=0.1, \
         fc='lightgray', ec='lightgray', ls='--', length_includes_head=True)
ax.arrow(*origin, *e2, head_width=0.05, head_length=0.1, \
         fc='lightgray', ec='lightgray', ls='--', length_includes_head=True)

plt.show()
```

Is A normal? Yes.



Feel free to try different matrices and see how changing components changes the resulting shape.

Eigenvectors are vectors that only get *stretched* under the action of a matrix. The factor λ by which that eigenvector is stretched is called its corresponding **eigenvalue**:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

So after applying the matrix, the resulting vector is *colinear* to the original vector.

The eigenvectors of a matrix span an orthogonal system, in which the matrix \mathbf{A} is *diagonal* (often denoted as \mathbf{D}). The eigenvector matrix $\mathbf{V} = [\mathbf{v}_1 \mathbf{v}_2 \dots \mathbf{v}_n]$, containing all eigenvectors \mathbf{v}_i in its columns, can be used to express this mathematically:

$$\mathbf{D} = \mathbf{V}^{-1} \mathbf{A} \mathbf{V}$$

or, equivalently:

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$$

The diagonal matrix \mathbf{D} now contains all the eigenvalues of \mathbf{A} . What happens if we apply this decomposed form sequentially to a vector? Let's try this with a more complicated matrix:

```
fig, axes = plt.subplots(2, 2, figsize=(10,10))
axes = axes.flatten()

titles = [r"$\mathbf{V}^{-1}$",
          r"$\mathbf{D} \mathbf{V}^{-1}$",
          r"$\mathbf{V} \mathbf{D} \mathbf{V}^{-1}$",
          r"$\mathbf{A}$"]

for i,ax in enumerate(axes):
    ax.set_aspect('equal')
    ax.set_xlim(-1.0, 2.5)
    ax.set_ylim(-1.0, 2.0)
    ax.set_title(label=titles[i])

# vectors
e1 = np.array([0.0, 1.0])
e2 = np.array([1.0, 0.0])
origin = np.array([0.0, 0.0])

# try different matrices to see their effects
# diagonal matrices don't rotate, they only stretch, the stretch factors are
# the eigenvalues
#A = np.array([[1.9, 0.0],
#              [0.0, 1.4]])

# rotation matrices do not stretch, they only rotate
# since the eigenvalues are imaginary here, the rotation
# happens in dimensions we do not plot here, hence we only see
# a line where perpendicular vectors should be
# = np.pi / 2.0
#A = np.array([[np.cos(), np.sin()],
#              [-np.sin(), np.cos()]])

# combining both
A = np.array([[1.5, 0.5],
              [0.5, 1.0]])

# this saves eigenvalues into D and normalized eigenvectors as ROWS into V
```

```

, V = np.linalg.eig(A)

# since we need the eigenvectors in the COLUMNS:
#V = V.T

V_inv = np.linalg.inv(V)
D = np.diag( )

print("Eigenvalues: \n", )
print("Eigenvectors: \n", V)

# plot original square
axes[0].add_patch(patches.Polygon(xy=[origin, e1, e1+e2, e2], 
    ↪color="lightgray"))

# apply first rotation
v1 = V_inv @ e1
v2 = V_inv @ e2
axes[0].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="darkorange"))

# apply stretch
axes[1].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="lightgray"))
v1 = D @ v1
v2 = D @ v2
axes[1].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="darkorange"))

# apply second rotation
axes[2].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="lightgray"))
v1 = V @ v1
v2 = V @ v2
axes[2].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="darkorange"))

# compare to action of original matrix
axes[3].add_patch(patches.Polygon(xy=[origin, e1, e1+e2, e2], 
    ↪color="lightgray"))
v1 = A @ e1
v2 = A @ e2
axes[3].add_patch(patches.Polygon(xy=[origin, v1, v1+v2, v2], 
    ↪color="darkorange"))

# plot eigenvectors everywhere

```

```

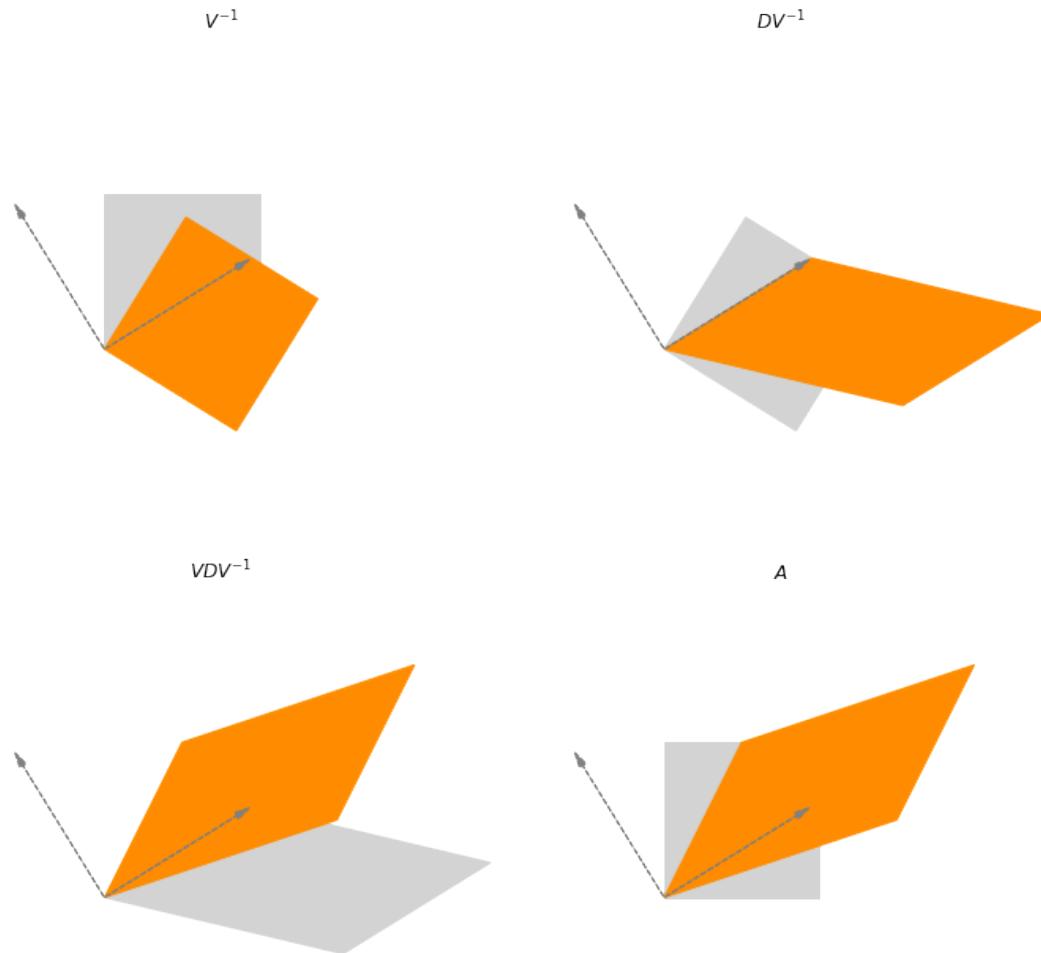
for ax in axes:
    ax.arrow(*origin, *V[:,0], head_width=0.05, head_length=0.1, fc='gray', u
    ↪ec='gray', ls='--')
    ax.arrow(*origin, *V[:,1], head_width=0.05, head_length=0.1, fc='gray', u
    ↪ec='gray', ls='--')
    ax.axis("off")

plt.tight_layout()
plt.show()

```

Eigenvalues:
 $[1.80901699 \ 0.69098301]$

Eigenvectors:
 $[[0.85065081 \ -0.52573111]$
 $[0.52573111 \ 0.85065081]]$



So indeed, the vectors are first rotated by this decomposition, then stretched along the original coordinate axes, then rotated back.

Real symmetric matrices have real eigenvalues and eigenvectors. In case \mathbf{A} is symmetric, the eigenvector matrix \mathbf{V} satisfies $\mathbf{V}^T = \mathbf{V}^{-1}$, so \mathbf{V} is orthogonal and hence, a **rotation matrix**. In this case, \mathbf{V} is sometimes denoted \mathbf{Q} or \mathbf{R} .

The eigendecomposition of a matrix might not be unique. An example of a matrix without a unique eigendecomposition is the identity matrix. The reason is that every vector is an eigenvector of the identity matrix, such that \mathbf{V} is arbitrary.

If we take the eigendecomposition and explicitly write down each term vectorwise (recall that matrix multiplication multiplies the rows of the first matrix with the columns of the second matrix) we get

$$\begin{aligned} A &= \lambda_1 v_1 v_1^T + \lambda_2 v_2 v_2^T + \dots \\ &= \lambda_1 \begin{bmatrix} | \\ v_1 \\ | \end{bmatrix} \begin{bmatrix} -- & v_1 & -- \end{bmatrix} + \lambda_2 \begin{bmatrix} | \\ v_2 \\ | \end{bmatrix} \begin{bmatrix} -- & v_2 & -- \end{bmatrix} + \dots \\ &= \sum_i \lambda_i v_i v_i^T \end{aligned}$$

which is one formulation of the **spectral theorem**, found in virtually all areas of research, especially numerical analysis.

10.3 Singular Value Decomposition

Singular value decomposition is a generalization of the eigendecomposition of normal square matrices into rotational and stretching parts to *non-square* matrices. The matrices in the decomposition now also have different dimensions.

Assuming $\mathbf{A} \in \mathbb{R}^{m \times n}$, we get

$$\mathbf{A} = \mathbf{U} \mathbf{S} \mathbf{V}^T$$

where $\mathbf{U} \in \mathbb{R}^{m \times m}$ and orthogonal, $\mathbf{S} \in \mathbb{R}^{m \times n}$ and diagonal, and $\mathbf{V} \in \mathbb{R}^{n \times n}$ and orthogonal. So \mathbf{U} and \mathbf{V} are rotations, while \mathbf{S} purely expresses stretching. All non-diagonal elements of \mathbf{S} are zero.

The columns of \mathbf{U} are eigenvectors of $\mathbf{A}\mathbf{A}^T$, called *left-singular vectors*. They are real, since \mathbf{U} is orthogonal. \mathbf{V} contains the eigenvectors of $\mathbf{A}^T\mathbf{A}$, called the *right-singular eigenvectors*. The diagonal of \mathbf{S} consists of the square roots of the eigenvalues of $\mathbf{A}^T\mathbf{A}$, called the *singular values*:

$$\text{diag}(\mathbf{S}) = \sqrt{\lambda(\mathbf{A}^T\mathbf{A})}$$

The interpretation of what is happening with this decomposition is very similar to what's happening in an eigendecomposition. \mathbf{V}^T rotates into the eigenspace of \mathbf{A} , \mathbf{S} applies stretch factors, \mathbf{U} rotates the system back. We'll visualize this behavior in the exercises today.

If we spell out the vectorwise multiplications like we did for the eigendecomposition, we get:

$$\begin{aligned}
 A &= \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots \\
 &= \sigma_1 \begin{bmatrix} | \\ u_1 \\ | \end{bmatrix} \begin{bmatrix} -- & v_1 & -- \end{bmatrix} + \sigma_2 \begin{bmatrix} | \\ u_2 \\ | \end{bmatrix} \begin{bmatrix} -- & v_2 & -- \end{bmatrix} + \dots \\
 &= \sum_i^m \sigma_i u_i v_i^T
 \end{aligned}$$

where this time the sum only goes up to the first dimension of the original matrix \mathbf{A} (can you see why?). This is especially interesting when $n \gg m$, which would be the case when samples are sparse and features are many in a data matrix.

There are several interesting applications. Very often, SVD is used for data compression, where dimensions of the system with small singular values are completely omitted, especially when the SVD of a *covariance matrix* is performed. This can reduce the order of a model significantly. In control theory, SVD is used for developing robust control. It can also be used as a tool to diagonalize matrices, or for *image/data compression*. The idea for all of these is to take the decomposition $\mathbf{U}, \mathbf{S}, \mathbf{V}$ and *truncate* their axis such that only a specific number of singular values is used. For \mathbf{U} , this would be the second axis, for \mathbf{S} both axes, and for \mathbf{V} the first axis. For the spectral decomposition above, this would be equivalent to cutting off all terms after some index.

Let's see what happens when we use SVD on an image. How many singular values do you need to be able to interpret the image? What's at the bottom right of the image?

```

import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
from skimage import data, img_as_float
from skimage.color import rgb2gray

grayscale = True

# taking an astronaut image from scikit-image as an example
image = img_as_float(data.astronaut())
if grayscale:
    image = rgb2gray(image)

# save this for later reshaping back
shape_original = image.shape

# get a 2d matrix of the image, so we can do SVD on it
# this is necessary because it contains 3 channels for all colors
if grayscale:
    image_2d = image

```

```

else:
    image_2d = image.reshape(shape_original[0], 3*shape_original[1])

# the SVD
U, S, V = np.linalg.svd(image_2d, full_matrices=False)

def compress_rgb_image(U,S,V, shape_original, cutoff=10):
    compressed_image = U[:, :cutoff] @ np.diag(S[:cutoff]) @ V[:cutoff, :]
    print(U[:, :cutoff].shape, np.diag(S[:cutoff]).shape, V[:cutoff, :].shape)

# get back original shape
image = compressed_image.reshape(shape_original)

fig, axes = plt.subplots(1, 2, figsize=(12,7))

axes[0].plot(S[:cutoff], lw=4)
axes[0].set_title(label="all singular values used")
axes[1].imshow((image * 255).astype(np.uint8), cmap='gray')

plt.tight_layout()

plt.savefig("img/plot_svd.png")

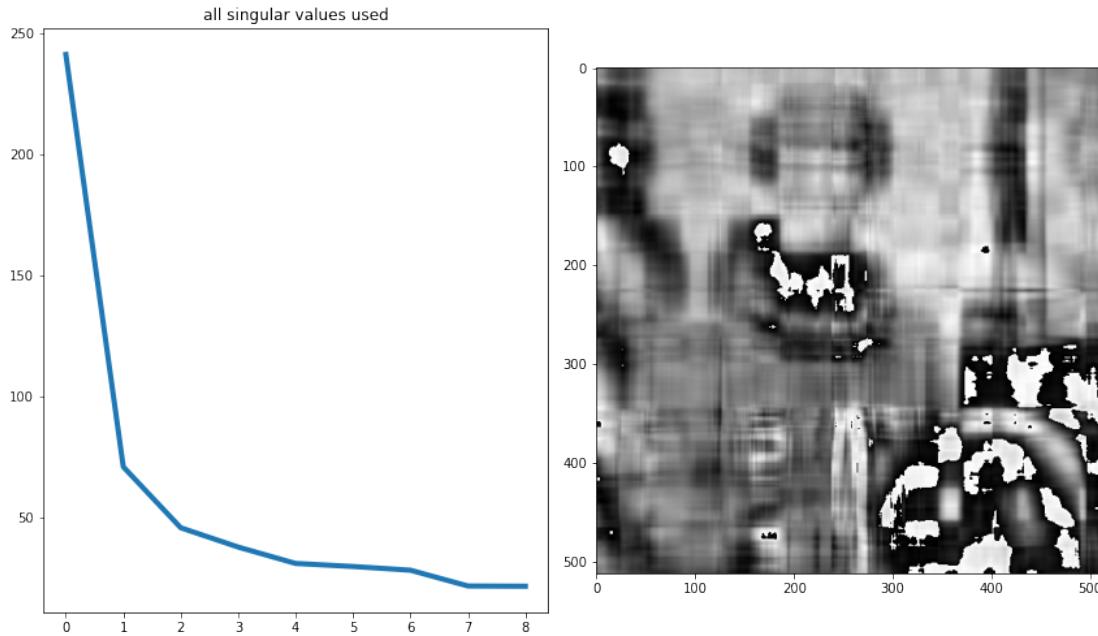
interact(compress_rgb_image,
         U=fixed(U),
         S=fixed(S),
         V=fixed(V),
         shape_original=fixed(shape_original),
         cutoff=widgets.IntSlider(min=0, max=150, value=2, continuous_update=False));

```

```

interactive(children=(IntSlider(value=2, continuous_update=False, description='cutoff', max=150, min=0),)

```



We see that not all singular values are necessary to convey meaning in the image. Depending on the level of detail desired, even a few suffice to recognize the astronaut, the US flag and the space ship. These kinds of data compression can be useful to reduce the order of a model in general, not only regarding ML techniques.

One area where this can be a great way to compress information is computational fluid dynamics. Consider a 2D flow field, the dynamics of which are calculated over some time range $[t_0, \dots, t_m]$, discretized into m time slices, reminiscent of frames from a video. Each 2D slice can be unpacked into a vector, where each entry would be the value of some field quantity at a node (or a pixel for image data) in the system. These slices are the m samples in the data matrix, and the nodal values are the n features. An SVD performed on this data matrix will find m “eigenmodes” of the flow field and express the original flow fields as linear combinations of these eigenmodes. The eigenmodes are the row vectors of \mathbf{U} , the coefficients are the corresponding singular values and entries from \mathbf{V} . Truncating this description would yield flow fields with lower resolution as approximations to the real flow field.

10.4 Principal Component Analysis

We briefly mentioned a few naive approaches to manipulating the number of features used for describing a problem. Another approach would be to combine features by linear (or later even nonlinear transformations) in a more methodical way, that respects the *correlations* in a dataset. The idea is visualized in the graph below, where two features x_1 and x_2 are correlated:

```
import numpy as np
import matplotlib.pyplot as plt
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

```

x = np.random.rand(40)
y = -0.3 + 0.5*x
y_test = -0.3 + 0.5*x + 0.2*np.random.rand(40)-0.1
p1 = np.array([0.0, -0.3])
p2 = np.array([1.0, 0.2])

points = np.c_[x,y_test]

d=np.cross(p2-p1,points-p1)/np.linalg.norm(p2-p1)

dires = np.array([-d[i]*np.array([1/np.sqrt(2), -1/np.sqrt(2)]) for i in
                  range(len(d))])
pps = points-dires

def plot_corr(line, dists):
    fig = plt.figure(figsize=(12,12))
    ax = fig.add_subplot(111)

    #ax.set_xlim([])
    ax.set_ylim([-0.4, 0.4])

    ax.set_xlabel(r"$x_1$", fontsize=20)
    ax.set_ylabel(r"$x_2$", fontsize=20)

    ax.scatter(x, y_test)
    if line:
        ax.plot(x, y, lw=4, zorder=-20)
    if dists:
        for i in range(d.shape[0]):
            ax.plot([points[i,0], pps[i,0]], [points[i,1], pps[i,1]], \
                    color='darkorange', lw=3, alpha=0.7, zorder=-10)
        ax.scatter(pps[:,0], pps[:,1], marker='x')
    #plt.axes().set_aspect("auto")#1./plt.axes().get_data_ratio()
    ax.set_aspect("auto")

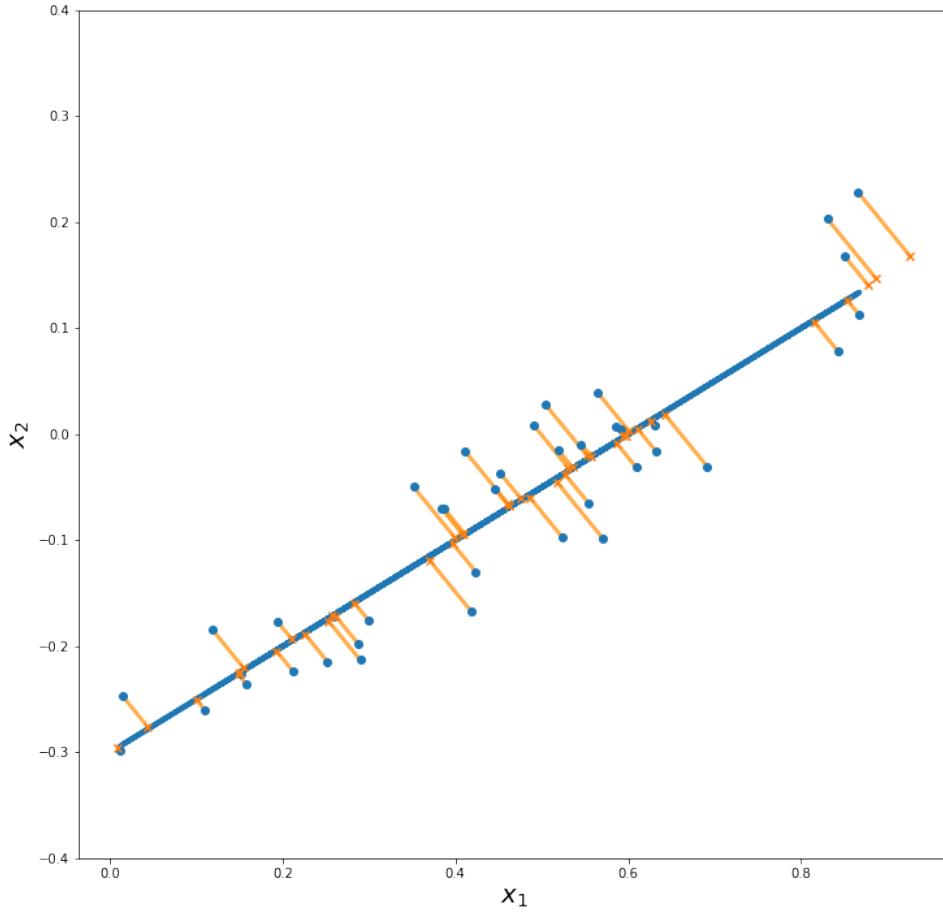
    plt.savefig("img/plot_correlation_lines.png")

interact(plot_corr, line=False, dists=False)

interactive(children=(Checkbox(value=False, description='line'), Checkbox(value=False, description='dists')))

<function __main__.plot_corr(line, dists)>

```



It is clear that there is a linear trend in the data (the features are correlated) and that we can find a new set of axes (activate line), where the distances of all the points to the new x' -axis are minimized, as in the image above when you activate dists.

These kinds of images are correlation plots. Sometimes manually deciding which features to eliminate or combine is possible by examining these correlation plots, but for hundreds of features this isn't really feasible. A technique that performs this combination of features in a way that reduces the overall number of features is principal component analysis (PCA). It tells us which features are the most important and provides the new axes like in the image above. PCA strives to maximize variance, while minimizing covariance of a dataset, so it finds rotated axes that best represent the dataset.

In the example above, the distances of the points to the new x' -axis are very small, so to reduce the dimension of the problem, instead of providing the coordinates of each point for both axes, we

could provide the projection of the points onto it and ignore the distance in y' -direction, since it is very small.

Mathematically, PCA solves the following problem: For a given dataset X with N samples x_i comprised of F features ($X \in \mathbb{R}^{N \times F}$), find an optimal $K \times F$ -matrix \mathbf{R} such that

$$\mathbf{x}'_i = \mathbf{R}^T \mathbf{x}_i$$

where $\mathbf{x}'_i \in \mathbb{R}^K$ and $K \leq N$. Let's look at another example:

```

x = np.random.rand(100)
y = -0.3 + 0.5*x
y_test = -0.3 + 0.5*x + 0.5*np.random.rand(100)-0.25
p1 = np.array([0.0, -0.3])
p2 = np.array([1.0, 0.2])

points = np.c_[x,y_test]

d=np.cross(p2-p1,points-p1)/np.linalg.norm(p2-p1)

dires = np.array([-d[i]*np.array([0.701, -0.701]) for i in range(len(d))])
pps = points-dires

def plot_corr(line, dists, new_axes):
    fig = plt.figure(figsize=(8,8))
    ax = fig.add_subplot(111)
    #plt.clf()

    ax.set_xlabel(r"$x_1$", fontsize=16)
    ax.set_ylabel(r"$x_2$", fontsize=16)

    ax.set_xlim([-0.3, 1.3])
    ax.set_ylim([-0.5, 0.6])

    ax.scatter(x, y_test)

    if line:
        ax.plot(x, y, lw=4, zorder=-20)
    if dists:
        for i in range(d.shape[0]):
            ax.plot([points[i,0], pps[i,0]], [points[i,1], pps[i,1]], \
                    color='darkorange', lw=3, alpha=0.7, zorder=-10)
            ax.scatter(pps[:,0], pps[:,1], marker='x')
    if new_axes:
        ax.arrow(0, -0.3, 1, 0.5, head_width=0.05, head_length=0.1, \
                 fc='red', ec='red', length_includes_head=True, zorder=1000)

```

```

    ax.arrow(0, -0.3, -0.701*0.1, 0.701*0.1, head_width=0.02, head_length=0.
→05, \
    fc='red', ec='red', length_includes_head=True, zorder=1000)
    #plt.quiver(0, -0.3, [1, -0.701], [0.5, 0.701], color = 'red', lw = 3, \
→zorder=1000)

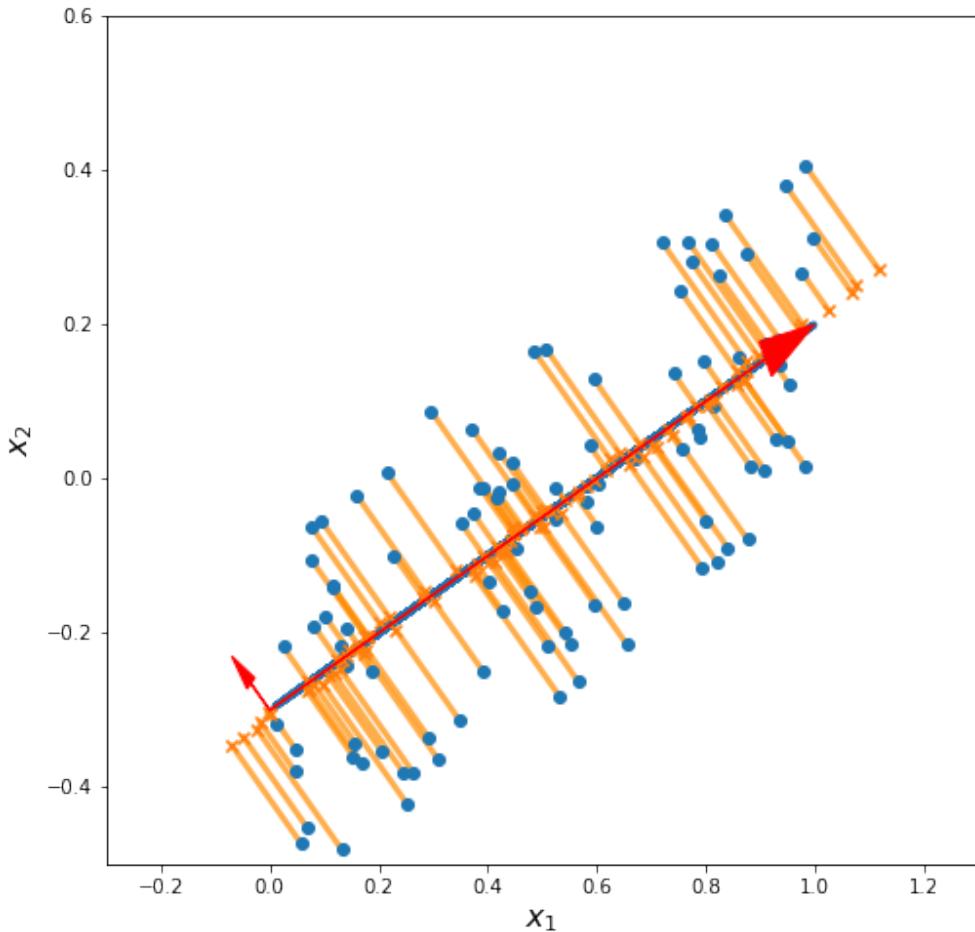
plt.savefig("img/plot_correlation_directions.png")

interact(plot_corr, line=False, dists=False, new_axes=False)

interactive(children=(Checkbox(value=False, description='line'), Checkbox(value=False, description='dists'), Checkbox(value=False, description='new axes')))

<function __main__.plot_corr(line, dists, new_axes)>

```



A lot of the information in the dataset is captured along the rotated x' -axis, which is the axis along which the variance is maximal. The y' -axis is orthogonal to this axis, maximizing the variance “perpendicular” to the first axis. If we project the points onto the x' -axis, we can retain the most information about the original dataset. To accurately do all of this, we need the directions of the axes x' and y' , as well as their “lengths” λ_1 and λ_2 . The length of an axis is the variance of the data along this direction. In this sense, PCA fits a hyperellipsoid to the data.

10.4.1 The Algorithm

To perform a PCA, the data has to be zero-centered first, which means that the mean of each feature over the whole dataset is calculated and subtracted from each respective column of every sample. The mean of feature f is calculated as

$$\mu_f = \frac{1}{N} \sum_i^N X_{if}$$

μ is a vector containing the feature means as columns. The zero-centered sample matrix is then

$$\bar{X} = X - U^T$$

where $U = \begin{bmatrix} | & | & \cdots & | \\ \mu & \mu & \cdots & \mu \\ | & | & \cdots & | \end{bmatrix}$. The next step is to calculate the covariance matrix C of the data:

$$C = \frac{1}{N} \bar{X}^T \bar{X}$$

Recall that the $F \times F$ covariance matrix contains the variance of a variable on the diagonal, and the covariances on the off-diagonal elements. When we’re trying to maximize variance (or, equivalently, minimize covariance), we’re looking for a diagonal matrix. This is the last step; diagonalize the covariance matrix (which is possible, since C is symmetric). This is done by finding the eigenvectors composing the matrix V (more specifically, the *right* eigenvectors), such that

$$C = V D V^T$$

where D is a diagonal matrix. The eigenvectors v_i are the **principal directions**. In this eigensystem, the entries λ_i of D are the **variance** of each feature in that system. In general, the eigenvalues and eigenvectors aren’t sorted. After sorting them in decreasing order of magnitude of the eigenvalues, the reduction process is performed by truncating the last $F - K$ rows of V and D , such that only the entries with the lowest variances are discarded:

$$R = V[0 : K]$$

This is the reduction matrix R from the problem description. Multiplying the full dataset by this reduction matrix gives the new dataset

$$X' = \bar{X}R \in \mathbb{R}^{N \times K}$$

with a reduced number K of features. The first row of this dataset contains the projection of the first sample in the original dataset onto the first principal component, and so on. To get the embedding of the original data into this lower-dimensional subspace, the means \bar{X} have to be added again to complete the transformation.

10.4.2 PCA by SVD

In practice, calculating and diagonalizing C is expensive and a much easier method is to use *singular value decomposition* on the data matrix itself:

$$X = USV^T$$

such that

$$C = \frac{1}{N}X^TX = \frac{1}{N}V S^T U^T U S V^T = \frac{1}{N}V S^2 V^T$$

Comparing this to the result from above, we see that the right-singular eigenvectors in V are the principal directions, and the variances are $\lambda_i = \frac{s_i^2}{N}$. You can see this by multiplying the equation above with V from the right. The columns of US are the principal components. Truncating all matrices like we saw in the last lesson to the first K entries to compute

$$X' = U[0 : K]S[0 : K]V[0 : K]^T$$

gives the *reduced* representation of the data, which now only consists of the K most important feature combinations regarding the variance they explain for the original dataset.

10.4.3 Summary

PCA is extremely common in dimensionality reduction. It can be used as a way to visualize a very high-dimensional dataset in 2D or 3D. See for example the 1D, 2D, and 3D PCA representation of *Andersen's Iris dataset* we've briefly seen in lecture 01, but is also used as a preprocessing step for machine learning algorithms. As we've seen in the last lesson about SVD, it can be applied to images to retain only the most important information about an image to reduce the data size and to be able to use much smaller deep learning models. It improves the performance of many machine learning (or classical) algorithms, since it minimizes correlations between the new axes and, if desired, removes unwanted features which are most likely noise in the dataset. The newly found axes are still orthogonal. In image-processing, this transformation is sometimes also called the *Karhunen-Loëve transformation*.

PCA can be generalized as principal curve analysis or principal surface analysis, and very commonly *kernel-PCA* which is a nonlinear version of PCA using the kernel-trick.

10.5 Generative PCA

Most machine learning algorithms can be viewed from (at least) two perspectives: linear algebra, and stochastics, although both are closely related. The implications are quite different though. In the former lessons, we introduced PCA as a projective technique that finds principal directions and projects the original samples onto a lower-dimensional subspace in a way that maximizes the variance captured by the projected samples. These projections form the **latent space** of the dataset, which is a usually lower-dimensional representation of the data, and from which the original data can be reconstructed approximately. Latent spaces are encountered in many other machine learning algorithms, such as autoencoders or generative adversarial networks. The idea is always the same:

The latent space of a problem encodes usually not interpretable, but meaningful information about a higher-dimensional problem.

This property has powerful implications. It enables a probabilistic perspective of the algorithms which turn from projection and transformation algorithms into modelling the probability distribution underlying the actual dataset, from which many things can be learned about the process behind the data. For example, the latent space vectors might be Gauss-distributed, such that outliers in the original dataset produce latent vectors that are far outside of the Gaussian-distributed latent space, enabling *anomaly detection*.

We could also perform Bayesian model comparison, by making use of the marginal likelihood (see lectures 02 and 04), but we'll omit this here. The (probably) more interesting part is that probability distributions can be sampled and hence, be used to *generate* new data points that are, in a certain sense, *similar* to the original samples. Let's take a step back first. We saw that we can construct the latent space representation of a sample x_i with the reduction matrix R like so:

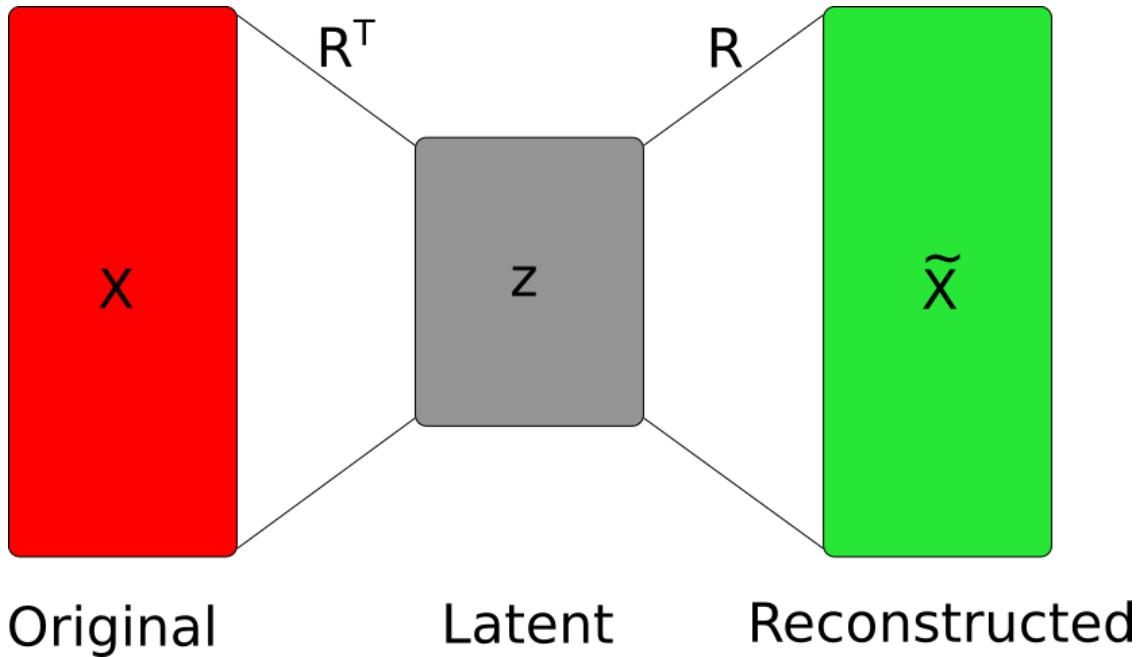
$$x'_i = z_i = R^T x_i$$

Usually, latent space vectors are denoted as z . We can also reconstruct the original sample by applying R :

$$\tilde{x}_i = RR^T x_i$$

where \tilde{x}_i is the approximate reconstruction of x_i , since by projecting onto the principal directions, we lost the information about the directions with lower variances.

```
from IPython.display import Image
Image("img/gPCA.png", width="500")
```



PCA can alternatively be expressed as an optimization problem by demanding that the reconstruction \tilde{X} of the data matrix should be as close as possible to X :

$$\text{PCA}(X) = \arg \min_{\tilde{X}} \|\tilde{X} - X\|_F$$

under the constraint $\text{rank}(\tilde{X}) = K$, where K is the number of singular values included in the reduction (which is equivalent to $\dim(Z)$). F denotes the Frobenius norm. The Eckard-Young theorem proves that the truncated SVD is the optimal solution to this problem.

What we can do now is to take a latent representation z of some sample and change its components slightly. This will yield new data points that are in a meaningful way similar to the original samples, but not contained in the original dataset.

The probabilistic way to see this is that PCA models the probability to observe the sample x_i , given its latent representation z , the reduction matrix R , mean vector μ and covariance matrix Σ :

$$p(x|z, R, \mu, \Sigma) = \mathcal{N}(x|Rz + \mu, \lambda \mathbf{1})$$

where \mathcal{N} is the normal distribution and $\lambda \mathbf{1}$ the covariance matrix in the latent space, which is diagonal and consists of the eigenvalues of the data covariance matrix that survived the truncation.

We will explore how to use this generative method in the exercises today for creating and manipulating airfoil designs that will look like this:

```
import pickle
import matplotlib.pyplot as plt
```

```

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

with open('af_pca.pkl', 'rb') as f:
    af_pca = pickle.load(f)

with open('components.pkl', 'rb') as f:
    components = pickle.load(f)

latent_dim = 10

def plot_generated(**kwargs):
    fig = plt.figure(figsize=(7,7))

    input_vector = [weight for weight in kwargs.values()]

    generated = af_pca.inverse_transform(input_vector)

    plt.xlim([-0.1, 1.1])
    plt.ylim([-0.15, 0.2])

    fig.patch.set_visible(False)
    plt.axis('off')

    plt.plot(generated.reshape(2,101)[0], generated.reshape(2,101)[1], lw=6, color="black")

    plt.savefig("img/plot_airfoil_pca.png")

base_sample = 8
component_sliders = [widgets.FloatSlider(
    value=components[base_sample][i],
    min=min(components[:,i]),
    max=max(components[:,i]),
    step=(max(components[:,i] - min(components[:,i]))/10),
) for i in range(latent_dim)]

kwargs = {'c' + str(i) : slider for i,slider in enumerate(component_sliders)}

interact(plot_generated, **kwargs)

```

```

interactive(children=(FloatSlider(value=-0.03723766920148769, description='c0', max=0.19187495

```

```

<function __main__.plot_generated(**kwargs)>

```



10.6 *t*-distributed Stochastic Neighborhood Embedding

Linear and kernel-techniques are limited in what information they can convey about highly convoluted high-dimensional data by the underlying idea that they preserve *euclidean distance* in the high-dimensional space. In convoluted cases, this can be quite misleading. See for example the swiss roll dataset:

```
%matplotlib notebook
import matplotlib.pyplot as plt
from sklearn.datasets import make_swiss_roll, make_blobs
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

```
X, y = make_swiss_roll(n_samples=400, random_state=0)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:,0], X[:,1], X[:,2], c=y)

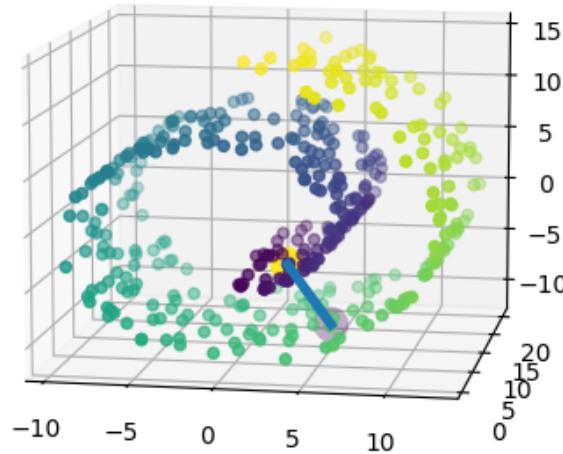
p1, p2 = 93, 332
ax.scatter(X[[p1,p2],0], X[[p1,p2],1], X[[p1,p2],2], lw=10, c=[0,1], zorder=500)
ax.plot(X[[p1,p2],0], X[[p1,p2],1], X[[p1,p2],2], lw=4, zorder=500)

ax.view_init(10, -80)

plt.savefig("img/plot_swiss_roll.png")
```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



The marked points have a small euclidean distance in the 3-dimensional space above (indicated by

the blue line), but only respecting the swiss roll itself, they are very far apart. The swiss roll is an example of a **manifold** (recall lecture 01), which has various competing, but equivalent definitions. For us it should suffice to say that at each point's immediate neighborhood, it looks like an \mathcal{R}^n . In this case, when you zoom in close enough, the data will look like an \mathcal{R}^2 . This is a 2-dimensional manifold *embedded* into a 3-dimensional space.

Euclidean distance is not a faithful measure of distance on such a manifold and hence, all methods based on this measure are bound to fail, or at least yield suboptimal results. Enter **manifold learning** techniques. There are various competing algorithms, and depending on the circumstances, some work better or faster than others. Here, we'll concentrate solely on **t-distributed Stochastic Neighborhood Embedding**, which is an immensely popular technique for nonlinear dimensionality reduction, but not straight-forward to interpret in the end. The underlying idea of all manifold learning techniques is that the data dimensions are excessive, and the true distribution is low-dimensional, but "wrapped" in high-dimensional space.

The bird's eye perspective on t-SNE is that it calculates a "similarity" measure in the high-dimensional space, another one in a low-dimensional space, and optimizing an objective function which tries to maximize the similarity. In detail, the following things happen:

- The algorithm overlays a *Gaussian distribution* over each point x_i , then calculates the probabilities for some points x_j in the vicinity of x_i with $p(x_j|x_i) = \frac{\exp(-||x_i-x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} \exp(-||x_i-x_k||^2/2\sigma_i^2)}$
- Compute the similarity $p_{ij} = \frac{p(x_j|x_i)+p(x_i|x_j)}{2N}$ with $p_{ii} = 0$, $p_{ij} = p_{ji}$, and $\sum_j p_{ij} = 1$. The interpretation of this similarity is the probability that a point x_i would accept x_j as being its neighbor under the assumption that this probability is proportional to a Gaussian distribution centered at x_i .
- Very roughly speaking, the standard deviation σ , which gives an estimate of the radius in which points are sufficiently similar to the center point, is related to **perplexity**. This is an important hyperparameter of t-SNE
- The similarity of the lower-dimensional points y_i is modeled by a *Student t-distribution* (sometimes called Cauchy distribution) $q_{ij} = \frac{(1+||y_i-y_j||^2)^{-1}}{\sum_k \sum_{l \neq k} (1+||y_k-y_l||^2)^{-1}}$
- Minimize the **Kullback-Leibler divergence** $KL(P||Q) = \sum_{i \neq j} p_{ij} \log \frac{p_{ij}}{q_{ij}}$, which measures the difference between two probability distributions.
- Minimizing the Kullback-Leibler divergence (also called relative entropy) is an optimization task, usually performed via gradient descent, and yields the low-dimensional representations y_i .

The Kullback-Leibler divergence is the *objective* or *loss function* to t-sne. *Perplexity* is a hyperparameter that balances local and global aspects of the data, meaning it provides an upper bound for how far points are allowed to deviate from the assumed underlying manifold to still count as belonging to the local neighborhood of a point.

t-SNE plots aren't straightforward to interpret due to their probabilistic nature, so let's look at a few examples:

```
%matplotlib inline
from sklearn.manifold import TSNE

X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1)
```

```

params = [[2, 1000],
          [5, 1000],
          [30, 1000],
          [50, 1000],
          [100, 1000]]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

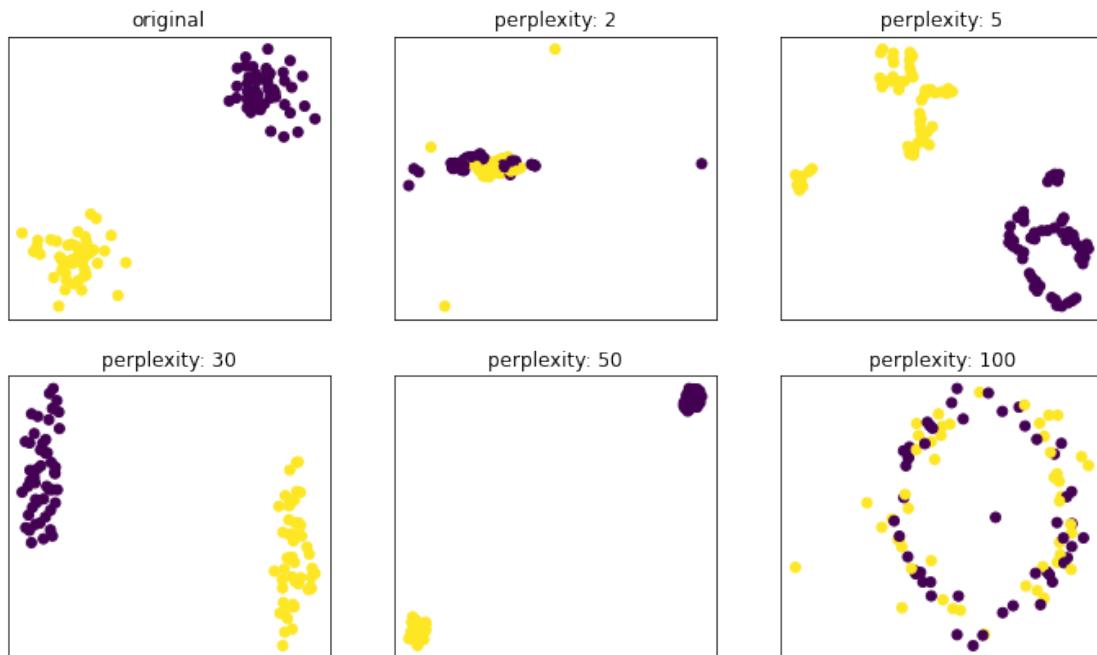
ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p[0], n_iter=p[1])
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)
    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p[0]))
    ax.set_xticks([])
    ax.set_yticks([])

```



The flattened cluster you're seeing in the fourth image often mean that the algorithm was stopped too early, so increasing the number of iterations makes sense. We see that perplexity heavily controls the results. When it's too low, the neighborhood of points is too small and many clusters appear in the result. When it's too high, too many points belong to clusters, such that no good separation is found. In the original paper, the authors suggest using values between 5 and 50 (smaller than the number of points, obviously) and testing multiple times which values make most sense.

A perplexity of 50 seems to work well. Unfortunately, the sciki-learn implementation does not allow iteration counts below 250, so we cannot visualize what happens before that.

In the plot below, the same reductions are performed for two clusters that differ in size (standard deviation):

```
X, y = make_blobs(n_samples=100, centers=2, n_features=2, random_state=1, u
→cluster_std=[1, 0.2])

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

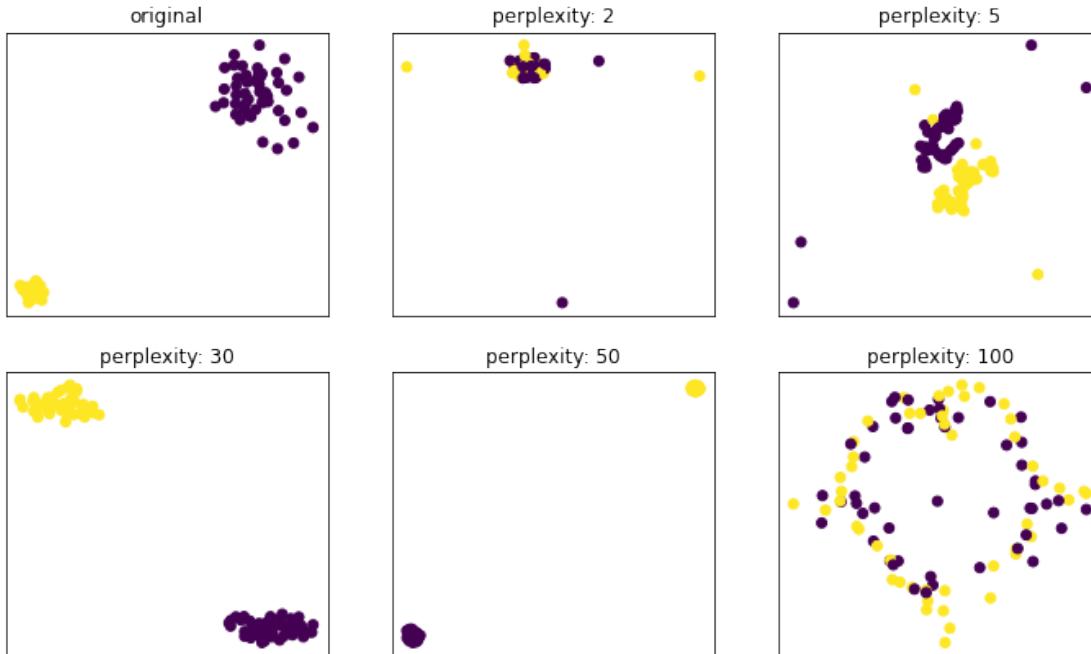
ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



The cluster size in the resulting t-SNE plot does not reflect the original cluster sizes. They look the same in the plots that converged. This is because the notion of similarity in t-SNE is a local one. As such, it expands dense clusters and contracts wide ones.

Let's see what happens to the distance of clusters themselves:

```
X, y = make_blobs(n_samples=100, centers=3, n_features=2, random_state=1) #, ↴
cluster_std=[1, 0.2])

params = [2, 5, 30, 50, 100]

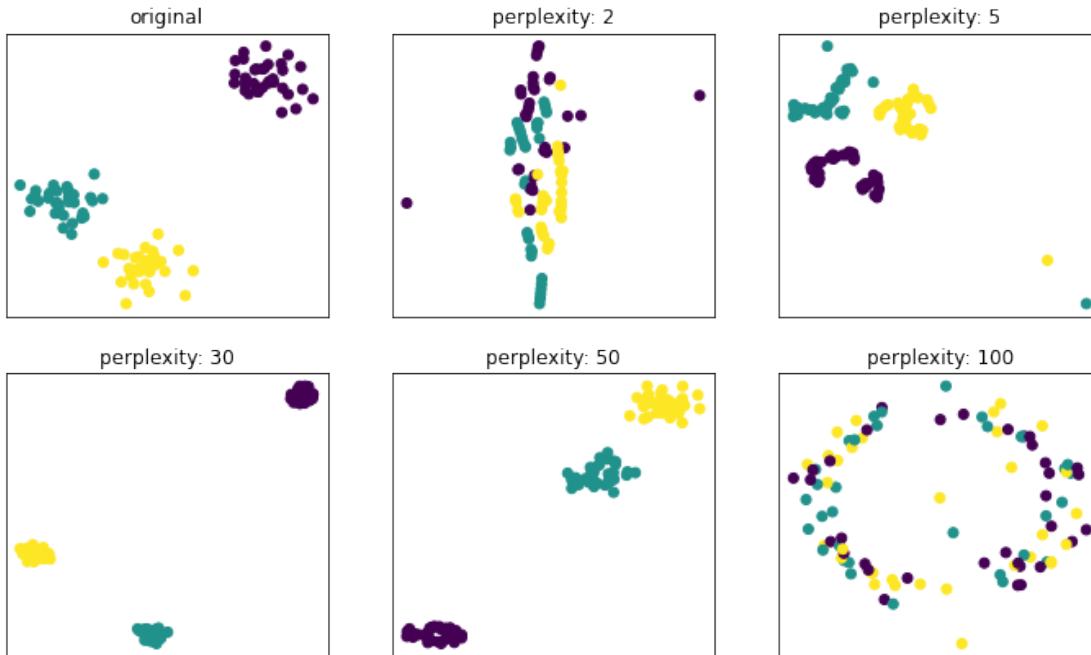
fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)
    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)
```

```
ax.set_title("perplexity: " + str(p))
ax.set_xticks([])
ax.set_yticks([])
```



In some of the plots, the global relative position of the clusters is reflected, while in others, it is not and instead they are more or less equidistant. t-SNE can, in principle, depict the relative position of clusters, but this necessitates fine-tuning the perplexity (the correct value depends on the number of points in each cluster). In general, the relative position of clusters in the reduced plots are not reliable predictors of the relative positions of the original clusters.

One reason for why it's necessary for t-SNE plots to do them multiple times with different hyperparameters is that they sometimes portray random data as having clusters:

```
from sklearn.datasets import make_gaussian_quantiles

X, Y = make_gaussian_quantiles(n_features=2)

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1])

ax.set_title("random")
ax.set_xticks([])
```

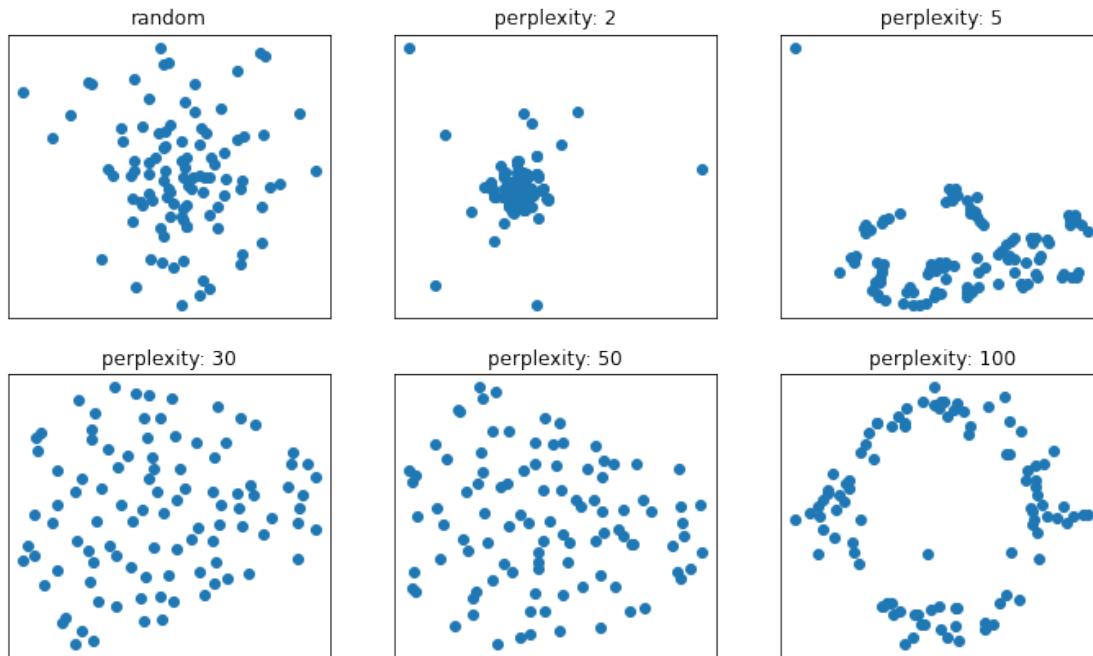
```

ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)
    ax.scatter(tsne_results[:,0], tsne_results[:,1])
    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])

```



The very low and very high perplexity plots show some structure, where clearly none can be. This can be deceiving when dismissing the step to perform multiple plots.

Let's see what happens to parallel lines:

```

import numpy as np

n_points = 100
x = np.linspace(0, 10, n_points)

y1 = 0.3*x + 0.3*np.random.random(n_points)
y2 = 0.3*x + 2 + 0.3*np.random.random(n_points)

```

```
X = np.zeros([2*n_points, 2])
X[:n_points,0] = x
X[n_points:,0] = x
X[:n_points,1] = y1
X[n_points:,1] = y2

y = np.zeros(2*n_points)
y[n_points:] = 1

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

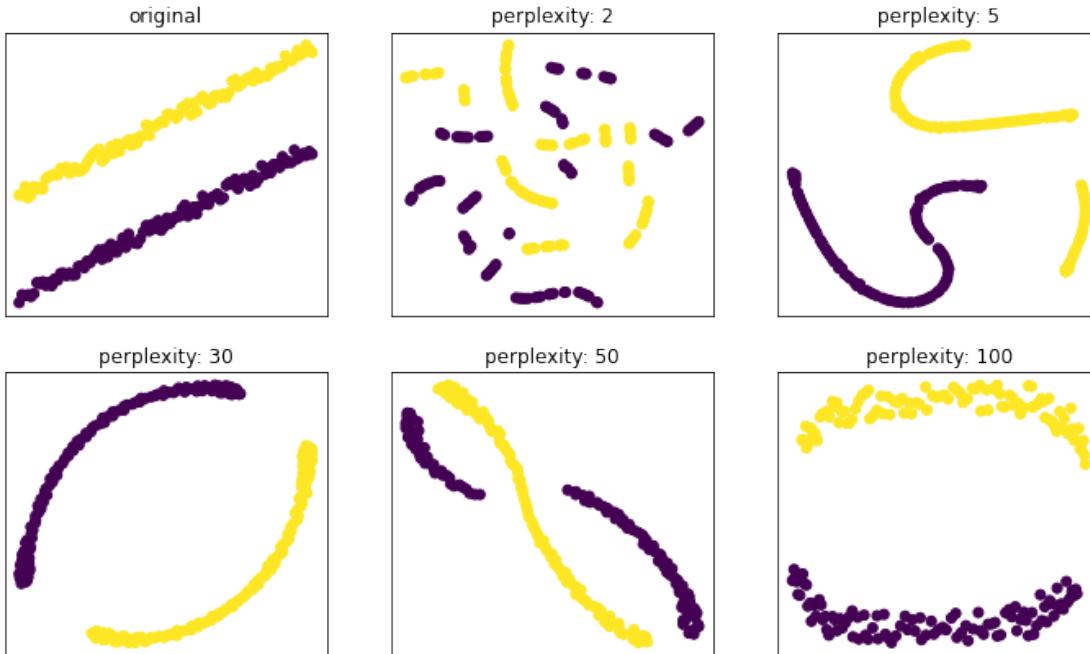
ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=500)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)

    ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])
```



Here, we needed to increase the number of iterations to get good results. The curves are depicted almost faithfully in most cases, but slightly curved outwards. The reason is that the regions in the middle of the lines are more densely populated with samples than the ends. As explained above, t-SNE is sensitive to dense and wide distributions of points. This is reflected in the examples above.

The last important property is topology preservation. See for example the enclosed rings from before:

```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples=100, factor=.1, noise=.05)

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231)
ax.scatter(X[:,0], X[:,1], c=y)

ax.set_title("original")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=500)
    tsne_results = tsne.fit_transform(X)
```

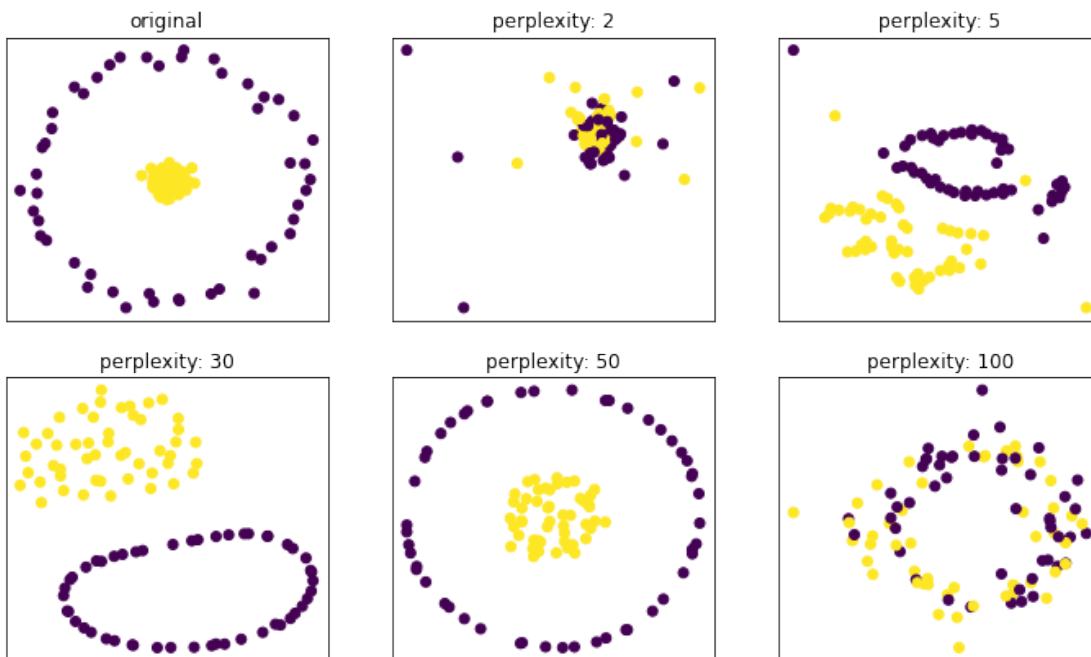
```

ax = fig.add_subplot(230+i+2)

ax.scatter(tsne_results[:,0], tsne_results[:,1], c=y)

ax.set_title("perplexity: " + str(p))
ax.set_xticks([])
ax.set_yticks([])

```



Again, preserving the topology is highly dependent on fine-tuning the perplexity. This is also apparent when reducing the trefoil knot:

```

%matplotlib notebook
from sklearn.datasets import make_circles

= np.linspace(0, 2*np.pi, 200)
x = np.sin() + 2*np.sin(2*)
y = np.cos() - 2*np.cos(2*)
z = -np.sin(3*)

X = np.c_[x,y,z]

params = [2, 5, 30, 50, 100]

fig = plt.figure(figsize=(12,7))
ax = fig.add_subplot(231, projection='3d')

```

```

ax.scatter(X[:,0], X[:,1], X[:,2])

ax.set_title("trefoil knot")
ax.set_xticks([])
ax.set_yticks([])

for i,p in enumerate(params):
    tsne = TSNE(n_components=2, perplexity=p, n_iter=1000)
    tsne_results = tsne.fit_transform(X)

    ax = fig.add_subplot(230+i+2)
    ax.scatter(tsne_results[:,0], tsne_results[:,1])

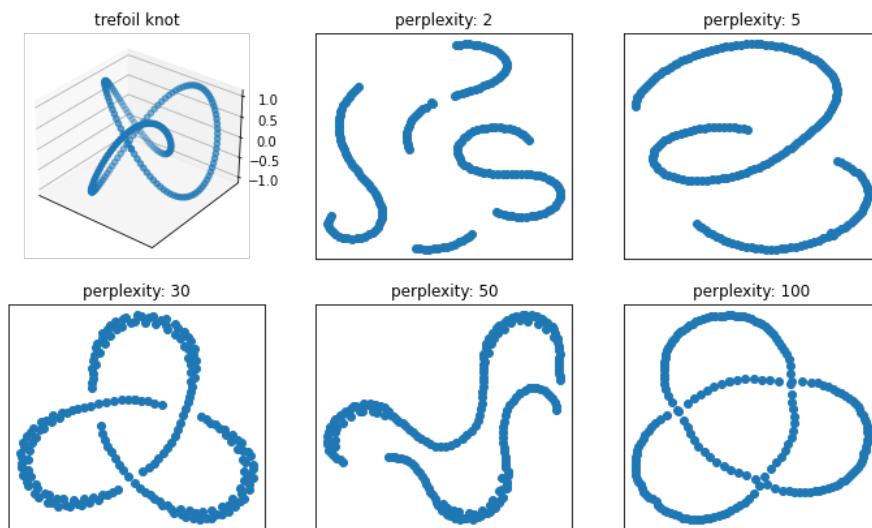
    ax.set_title("perplexity: " + str(p))
    ax.set_xticks([])
    ax.set_yticks([])

plt.savefig("img/plot_trefoil_tsne.png")

```

<IPython.core.display.Javascript object>

<IPython.core.display.HTML object>



With lower perplexities, t-SNE cuts the continuous line at some places to produce severed strings.

The actual topology is preserved with higher perplexities and appropriate numbers of iterations.

What's left is to perform t-SNE on the swiss roll and compare the result to PCA and kernel-PCA:

```
%matplotlib inline
from sklearn.decomposition import PCA, KernelPCA

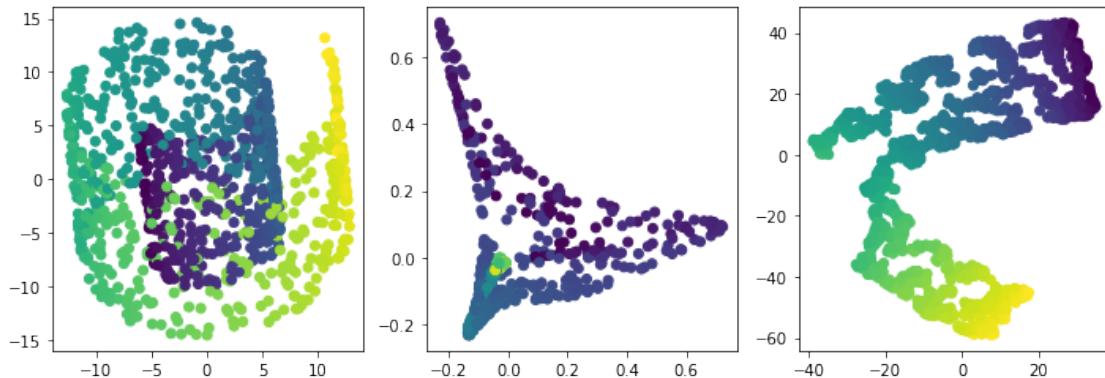
X_swiss, y_swiss = make_swiss_roll(n_samples=1000, random_state=0)

techniques = [PCA(n_components=2),
              KernelPCA(n_components=2, kernel='rbf', gamma=0.1),
              TSNE(n_components=2, perplexity=30, n_iter=5000, init='pca')]

results = [red.fit_transform(X_swiss) for red in techniques]

fig, axes = plt.subplots(1,len(results), figsize=(12,4))

for i,ax in enumerate(axes):
    ax.scatter(results[i][:,0], results[i][:,1], c=y_swiss)
```



The configuration here doesn't quite manage to completely unfold the swiss roll, but it comes close and detects the connections of the data points quite well. There are other, more advanced techniques that easily unroll the dataset, which all make use of the manifold property of the dataset. Linear PCA only projects the swiss roll onto a 2D space, kernel-PCA finds an embedding that slightly respects the position of points, but only t-SNE find an embedding that almost faithfully represents the manifold.

10.7 K-Nearest Neighbors

K-Nearest Neighbors is a simple classification algorithm. Let's take an example dataset to see what it does:

```
%matplotlib inline
import numpy as np
```

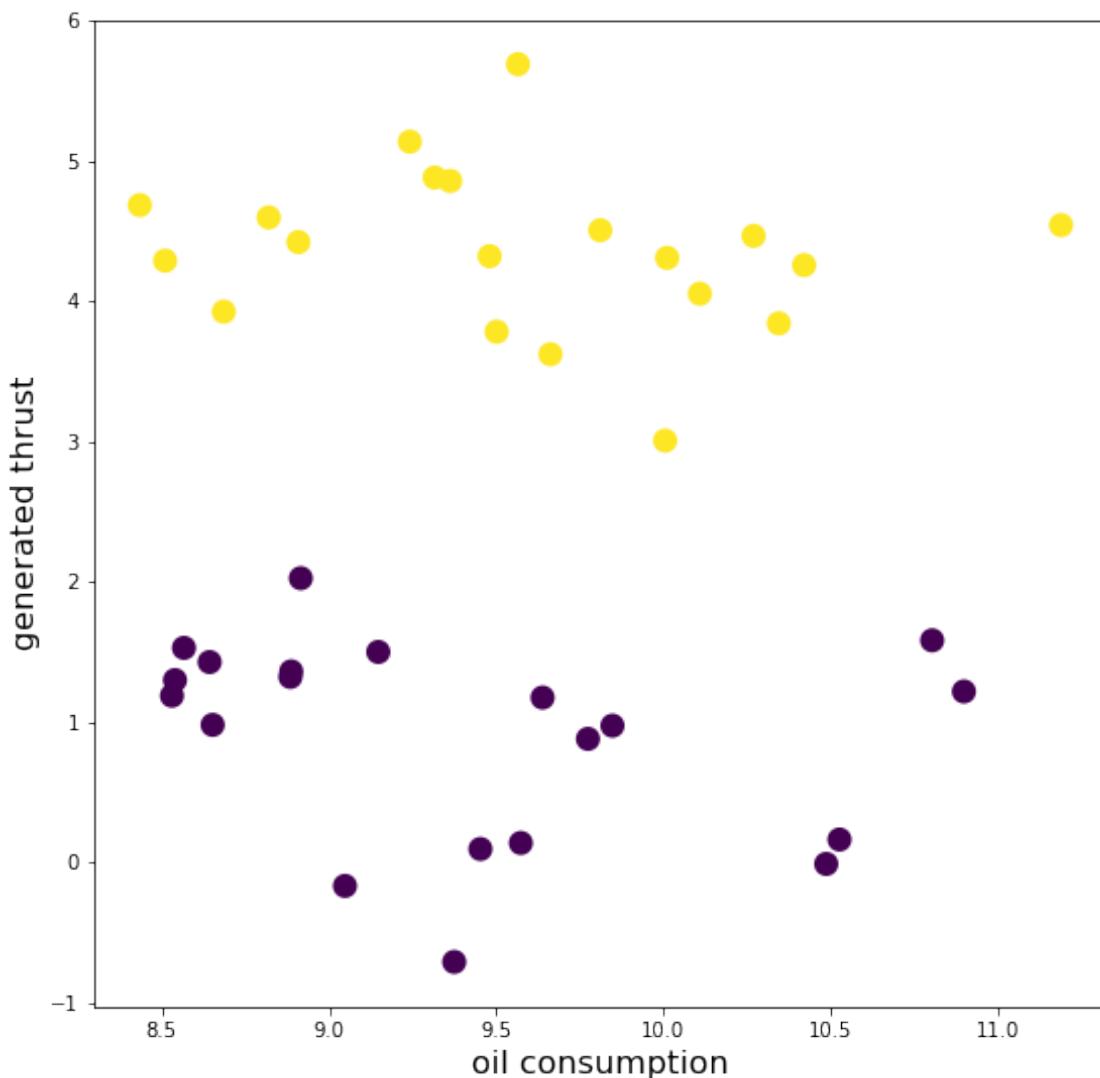
```
from matplotlib import pyplot as plt
from sklearn.datasets import make_blobs
from sklearn.cluster import KMeans

X, y = make_blobs(n_samples=40, centers=2, cluster_std=0.70, random_state=4)

fig = plt.figure(figsize=(9,9))

plt.scatter(X[:,0], X[:,1], c=y, lw=6)
plt.xlabel("oil consumption", fontsize=16)
plt.ylabel("generated thrust", fontsize=16)
#plt.xticks([])
#plt.yticks([])

plt.show()
```



In the above graph, say we have a dataset of oil consumption and thrust generated of an aircraft engine that was tested for reliability. The red dots mean the engine is going to fail soon, blue dots mean it's good to go for a sufficiently long time and can be used on a plane. So again we have training data and labels, although for K -NN, no training happens.

K -NN has a single hyperparameter K , which is the **neighborhood size**. The idea is to look at a *neighborhood* of a data point that is introduced into the graph above.

```

from ipywidgets import interact
from collections import Counter

max_neighbors = 6

def get_k_neighbors(dataset, new_point, neighbors):
    distances = np.array([np.linalg.norm(p-new_point) for p in dataset])
    min_idxs = np.argsort(distances)[:neighbors]
    min_dists = distances[min_idxs]
    min_lidxs = np.argsort(min_dists)

    return min_idxs[min_lidxs], min_dists[min_lidxs]

new_point = np.array([9, 2.9])
n_idx, dists = get_k_neighbors(X, new_point, max_neighbors)

fig = plt.figure(figsize=(9,9))

def plot_neighbors(k=0):
    plt.cla()

    plt.scatter(X[:,0], X[:,1], c=y)
    plt.xlabel("oil consumption", fontsize=16)
    plt.ylabel("generated thrust", fontsize=16)
    plt.xlim([7,13])

    plt.scatter(*new_point, marker='x', s=[100])

    plt.scatter(X[n_idx[:k],0], X[n_idx[:k],1])

    plt.savefig("img/plot_knn.png")
    plt.show()

    print("Class: ", Counter(y[n_idx[:k]]).most_common(1))

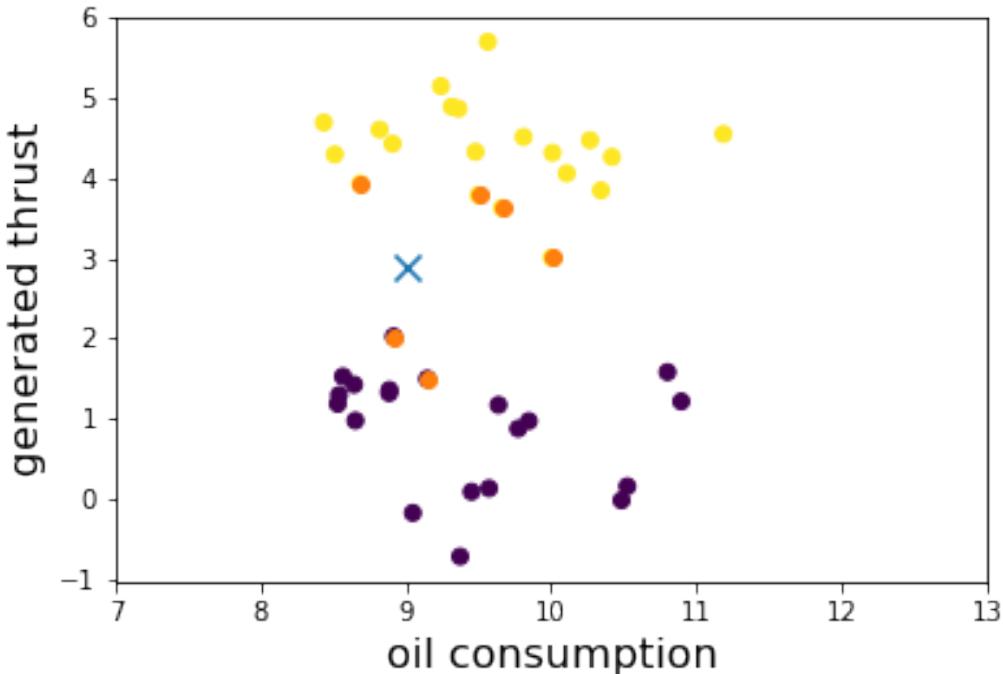
interact(plot_neighbors, k=(0, max_neighbors))

```

<Figure size 648x648 with 0 Axes>

```
interactive(children=(IntSlider(value=0, description='k', max=6), Output()), _dom_classes='wide')
```

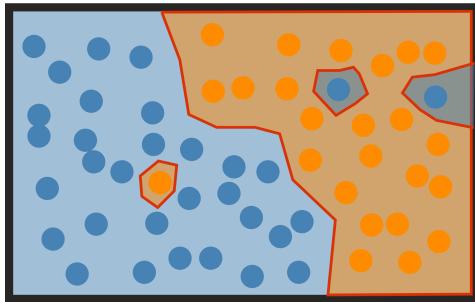
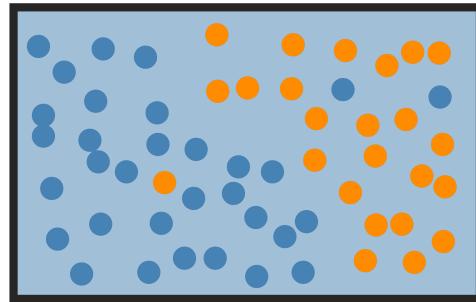
```
<function __main__.plot_neighbors(k=0)>
```



In the example above, the closest neighbor of the newly introduced point is yellow, which would result in this new point to be classified as “close to failure”. For $K = 2$, there is a yellow and a purple dot, resulting in an undecidable situation. For $K = 3$, there are two purple dots and one yellow dot, resulting in a “good to go”. These results were produced by using **majority vote**, where simply the largest number of labels in the neighborhood determines the prediction made. Alternatively, one could calculate the ratio of yellow points among the neighborhood and predict “failure happening with 33 %”.

The number K of neighboring points can be increased to see how the classification develops in order to find a sensible K . The points in the neighborhood should all come from the training set. The two extremes are $K = 1$, where only the nearest neighbor is considered, and $K = N$, where all training points are considered and every new point simply gets assigned the majority label. These situations are shown below:

```
from IPython.display import Image
Image("img/knn_k_1_vs_all.png", width="1000")
```

$K = 1$  $K = N$ 

The red line is the **decision boundary**, new points are classified as “good to go” on its left side and “close to failure” on its right side, unless they are close to the outliers.

To choose a sensible K , the available data is split into training/validation/test data again. The best K is then chosen by evaluating the performance of the algorithm on the validation set. An alternative is using *K-Fold cross validation*, where different randomly chosen train/validation splits are used and for each split K is varied to see which one gives the lowest variance and the best accuracy. Overfitting happens easily in K -NN.

To determine *closeness*, usually the euclidean distance is used. It makes sense to normalize the data before starting the algorithm, since euclidean distance would otherwise just be the absolute of the dominant feature. Alternatively, one could use L1 distance.

The generalization of K -NN to multiple classes is straightforward:

```
X, y = make_blobs(n_samples=50, centers=4, cluster_std=0.70, random_state=4)

fig = plt.figure(figsize=(6,6))

max_neighbors = 6

def get_k_neighbors(dataset, new_point, neighbors):
    distances = np.array([np.linalg.norm(p-new_point) for p in dataset])
    min_idxs = np.argsort(distances)[:neighbors]
    min_dists = distances[min_idxs]
    min_lidxs = np.argsort(min_dists)

    return min_idxs[min_lidxs], min_dists[min_lidxs]

new_point = np.array([9.5, 2.5])
n_idx, dists = get_k_neighbors(X, new_point, max_neighbors)

def plot_neighbors(k=0):
    plt.cla()
```

```
plt.scatter(X[:,0], X[:,1], c=y)
plt.xlabel("oil consumption", fontsize=16)
plt.ylabel("generated thrust", fontsize=16)
plt.xlim([7,13])

plt.scatter(*new_point, marker='x', s=[100])

plt.scatter(X[n_idx[:k],0], X[n_idx[:k],1])

plt.savefig("img/plot_multiclass_knn.png")
plt.show()

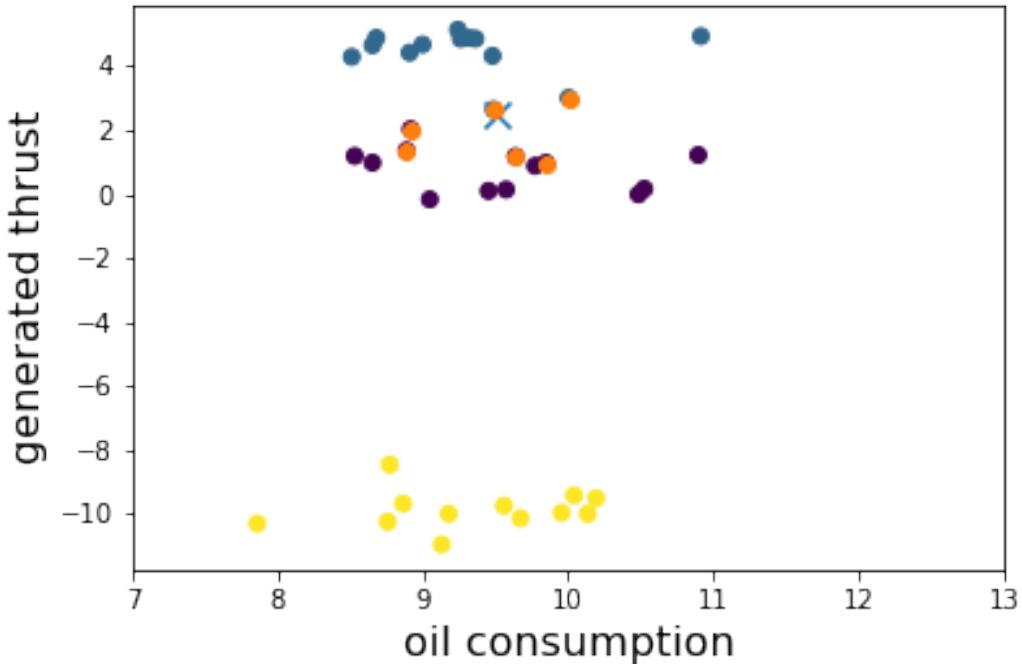
print("Class: ", Counter(y[n_idx[:k]]).most_common(1))

interact(plot_neighbors, k=(0, max_neighbors))
```

<Figure size 432x432 with 0 Axes>

```
interactive(children=(IntSlider(value=0, description='k', max=6), Output()), _dom_classes='wide')
```

```
<function __main__.plot_neighbors(k=0)>
```



10.8 K-Means

So far, we only looked at *supervised learning*, where along the data itself we had *labels/outputs* available, such that using training we could fit a model to approximate the relation between the data X and the labels y . In **unsupervised learning** there are no labels. In real-world scenarios, it's generally difficult to come by labeled data, so it makes sense to look a bit into what we can do without having access to a ground truth. A very common task is **clustering**, which trains a model on unlabeled data and outputs classes for data points, which sort them according to some metric of *closeness* and this way give a hint about which data points are related in some way.

Such a dataset where points show some relation could look like this:

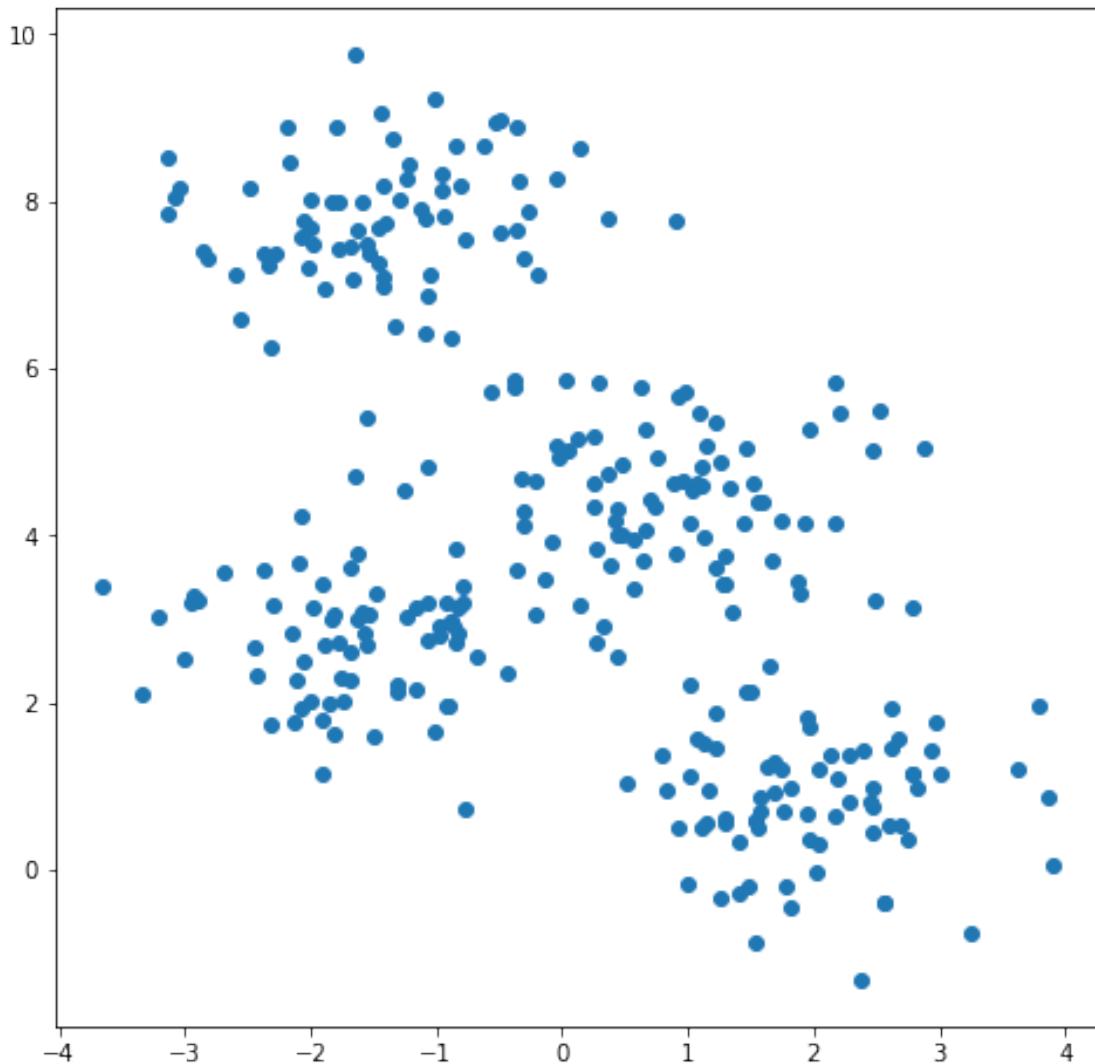
```
%matplotlib inline
import numpy as np
from matplotlib import pyplot as plt
from sklearn.datasets import make_blobs

X, y = make_blobs(n_samples=300, centers=4, cluster_std=0.80, random_state=0)

plt.figure(figsize=(8,8))

plt.scatter(X[:,0], X[:,1])
```

<matplotlib.collections.PathCollection at 0x7fa0d4ded710>



Again we're looking at a dataset with two features for ease of visualization. In earlier times, the **K-Means** algorithm was used to analyze newspaper articles (and blog posts), where similar articles built clusters. The result was then used to suggest a reader similar articles. So the above dataset could represent the frequency of certain words coming up in an article. A research group at MIT created a system roughly based on this idea (although more sophisticated) to scrape scientific papers automatically for material data. They built a database where they can quickly look up any material constant they might need.

In contrast to the example above, it is usually unknown how many clusters a dataset is comprised of. K is again a hyperparameter, that must be determined in some way. We will see how this can be done in a moment.

The algorithm itself works like this:

- * Initialize K cluster centers
- * Calculate the euclidean distance (or some other metric) of *every* point to *every* cluster center
- * Assign datapoints to the cluster that is the closest to them (this creates K clusters)
- * Calculate the mean of each cluster
- * Move each

center to its respective cluster mean * Repeat until convergence is reached

K-Means is very sensitive to initialization, which we'll see in a moment. Here, we assume the initialization was done randomly, but there are better methods to guess initial positions based on available data. A great initializer is *kmeans++* (*side note: this initialization helps avoiding worst-case scenarios for K-Means, which might become catastrophic since K-Means clustering is an NP-hard problem*). Different initializations lead to different cluster centers sometimes. There is no straightforward way to automatically determine which clustering is the correct one.

More formally, *K*-Means poses a combinatorial optimization problem with the loss function

$$L = \min_{C_{k=1\dots K}} \sum_{k=1}^K W(C_k) \quad (196)$$

where $W(C_k)$ is the *intra-cluster variance*

$$W(C_k) = \frac{1}{|C_k|} \sum_{i,i' \in C_k} \sum_{j=1}^p (x_{ij} - x_{i'j})^2 \quad (197)$$

which can be rewritten in terms of the cluster mean \bar{x}_{kj} :

$$W(C_k) = \sum_{i \in C_k} \sum_{j=1}^p (x_{ij} - \bar{x}_{kj})^2 \quad (198)$$

x_{ij} is the j th feature of the i th element in cluster C_k . So *K*-Means tries to minimize the cluster center distance to each cluster mean. These combinatorial optimization problems are very difficult to solve, so again the greedy approach in the algorithm above makes sense. The resulting clusters never intersect each other, each point gets a unique cluster label, and the union of all clusters will be the entire dataset. *K*-Means isn't the only way to solve this optimization problem, but it's a very common one.

A way to argue for a well-chosen K is to plot the intra-cluster variance as a function of K :

```
from sklearn.cluster import KMeans

plt.figure(figsize=(8,8))

# this will later contain the inter-cluster variance
# for all variations of K-Means for different K
intra_cluster_var = []

# prepare a range of K to test
K = [*range(1, 11)]

# perform kmeans for different numbers of cluster centers
for k in K:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    # calculate intra-cluster variance for current K
    intra_cluster_var.append(kmeans.inertia_)

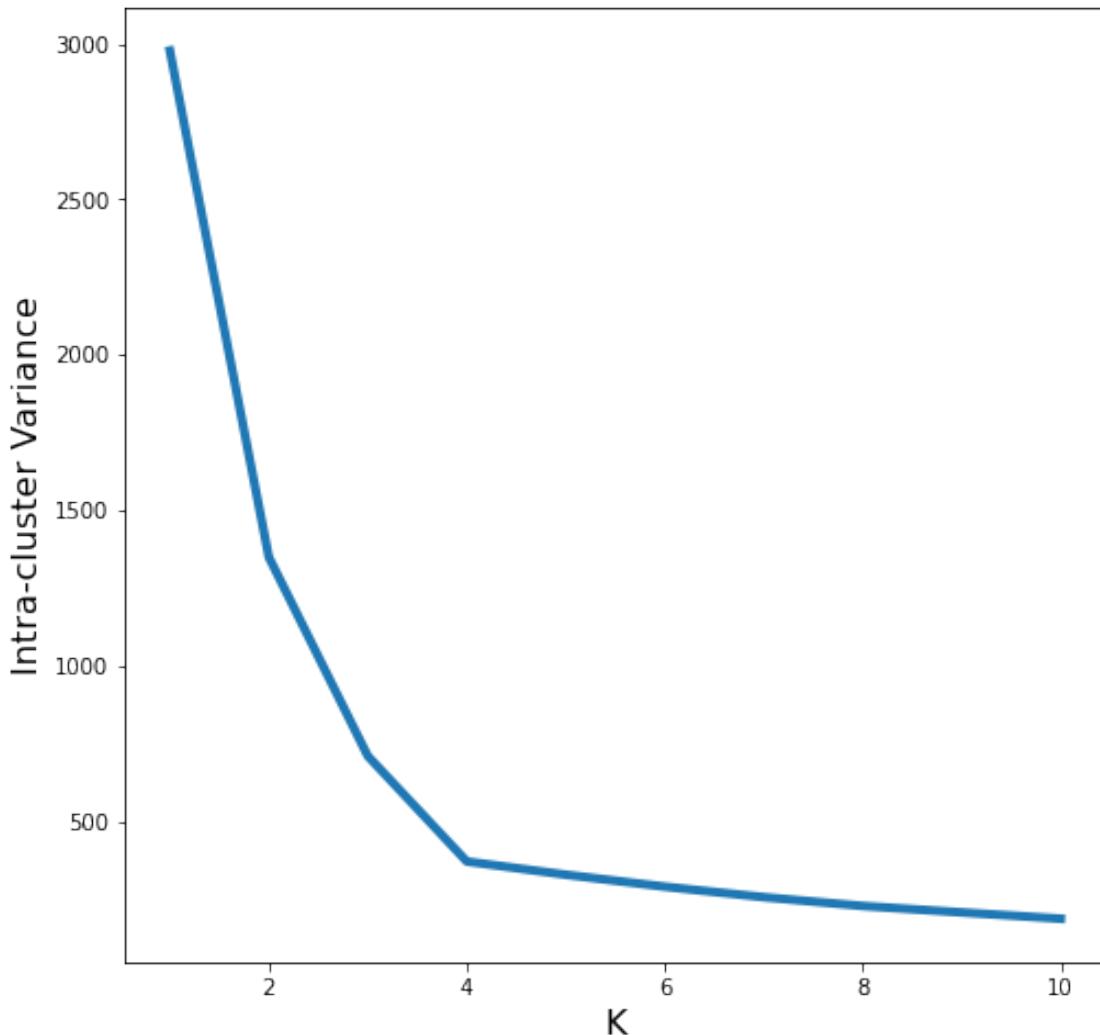
plt.plot(K, intra_cluster_var)
plt.xlabel('Number of clusters (K)')
plt.ylabel('Intra-cluster variance')
plt.title('Intra-cluster variance vs Number of clusters (K)')
```

```
kmeans.fit(X)
intra_cluster_var.append(kmeans.inertia_)

plt.plot(K, intra_cluster_var, lw=4)

plt.xlabel('K', fontsize=16)
plt.ylabel('Intra-cluster Variance', fontsize=16)

plt.show()
```



This is sometimes called the *elbow method*. The graph shows a strong kink or inflection point in most situations, which is the position of the optimal number of clusters K to assume for a given dataset. Since the result depends strongly on the initialization, the K -Means algorithm should be performed many times. The clustering with the lowest intra-cluster variance is the optimal one.

```

from ipywidgets import interact

max_iter = 10
centers = []

# note: for production runs, set init to "k-means++", which is a clever way to
# initialize the cluster centers
# random_state fixes the state of the random number generator to get
# reproducible
# results, you can delete it to get random cluster initializations
# a random_state of 4 will show a constellation that doesn't produce a good
# clustering
kmeans = KMeans(n_clusters=4, init="random", max_iter=1, n_init=1,
                 random_state=6)

# calculate cluster center coordinates for each step
for i in range(max_iter):
    kmeans.fit_predict(X)
    centers.append(kmeans.cluster_centers_)
    kmeans = KMeans(n_clusters=4, init=centers[-1], max_iter=1, n_init=1,
                    random_state=6)

centers = np.array(centers)

plt.figure(figsize=(8,8))

def plot_kmeans(step=1):
    plt.cla()

    plt.scatter(X[:,0], X[:,1])
    plt.scatter(centers[step,:,0], centers[step,:,1], s=300, c='red')

    plt.savefig("img/plot_k-means_centroids.png")
    plt.show()

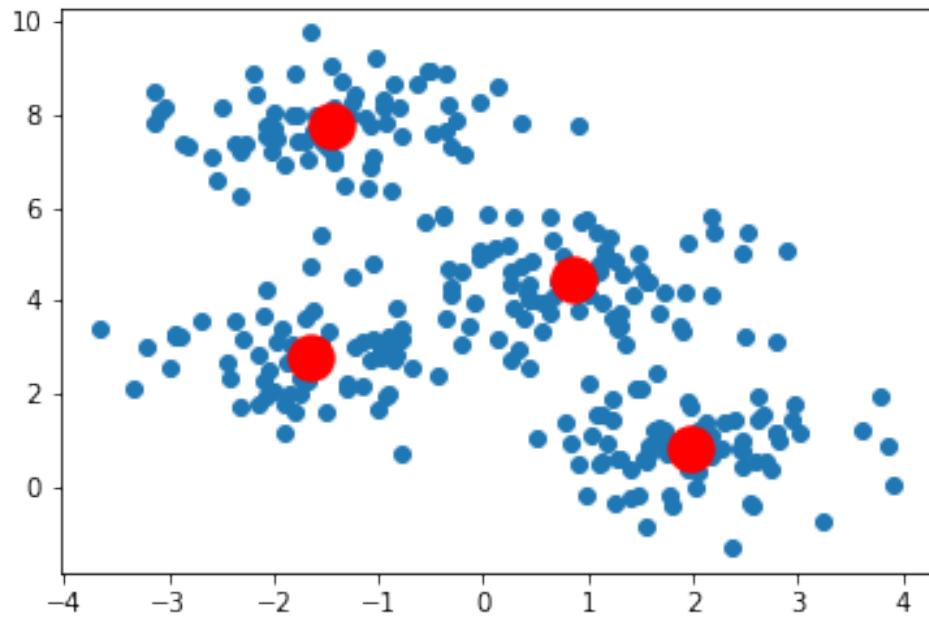
interact(plot_kmeans, step=(1, max_iter-1))

```

<Figure size 576x576 with 0 Axes>

```
interactive(children=(IntSlider(value=1, description='step', max=9, min=1), Output()), _dom_clo
```

```
<function __main__.plot_kmeans(step=1)>
```



11 Checkpoint I Repetition

11.1 Linear Regression

Define each feature, even polynomial ones, as variables, hypothesis is linear combination of weights and features, including bias $x_0 = 1$ and corresponding weight w_0 . Regression through pseudoinverse or least squares loss and optimization with gradient descent. Idea of gradient descent was to sample a point in the high-dimensional loss landscape, then calculate the gradient at that point and go into the other direction. This is a first order method, that does not use any information about curvature, i.e. no Hessian information. Other methods like L-BFGS do and may find better optima, but they don't scale well to high dimensions.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

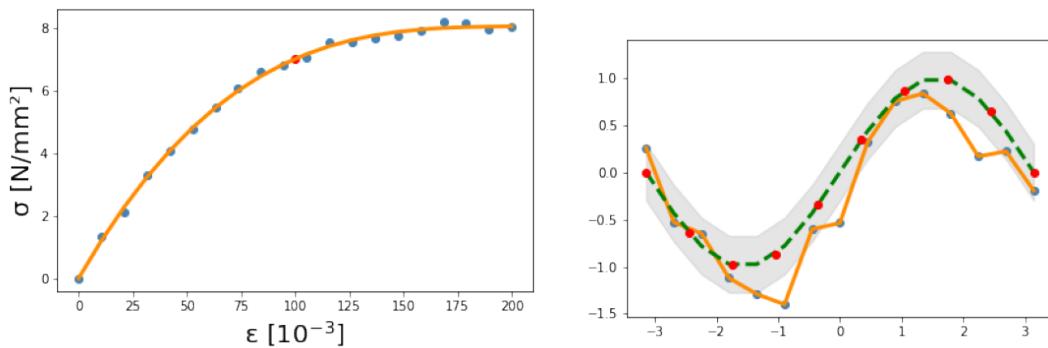
reg = mpimg.imread("img/plot_reg.png")
reg_deg = mpimg.imread("img/plot_reg_deg.png")

fig,axs = plt.subplots(1,2, figsize=(16,7))

axs[0].imshow(reg)
axs[1].imshow(reg_deg)

for ax in axs:
    ax.axis("off")

plt.tight_layout()
```

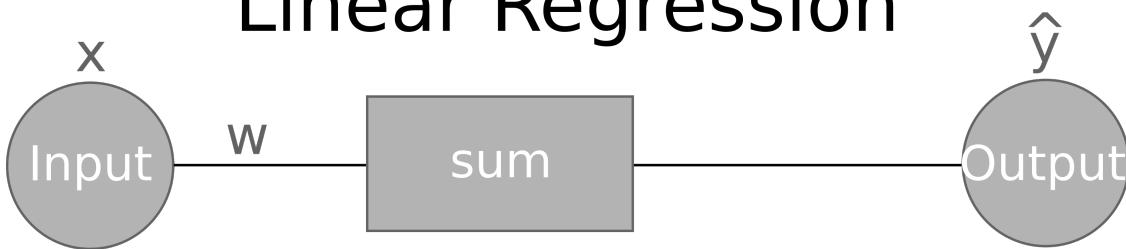


11.2 Classification

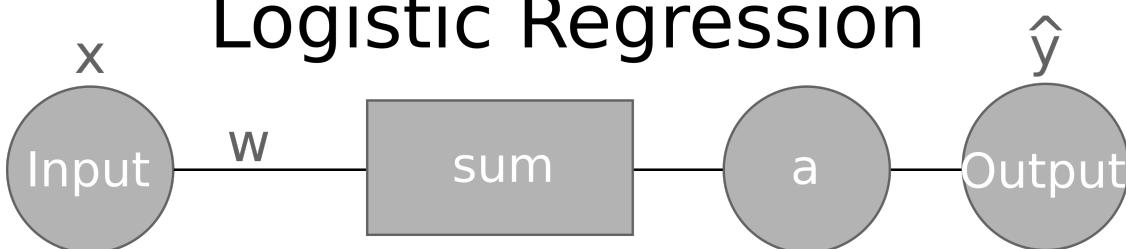
Exactly the same idea as linear regression, but with an extra nonlinearity, namely sigmoid. Hyperplane where hypothesis is zero is decision boundary, where sigmoid is 0.5. On one side, all values of sigmoid are lower, on the other, they're all higher. This is the classification of data points into classes 0 and 1.

```
from IPython.display import Image
Image("img/lin-log-reg.png", width=700)
```

Linear Regression



Logistic Regression

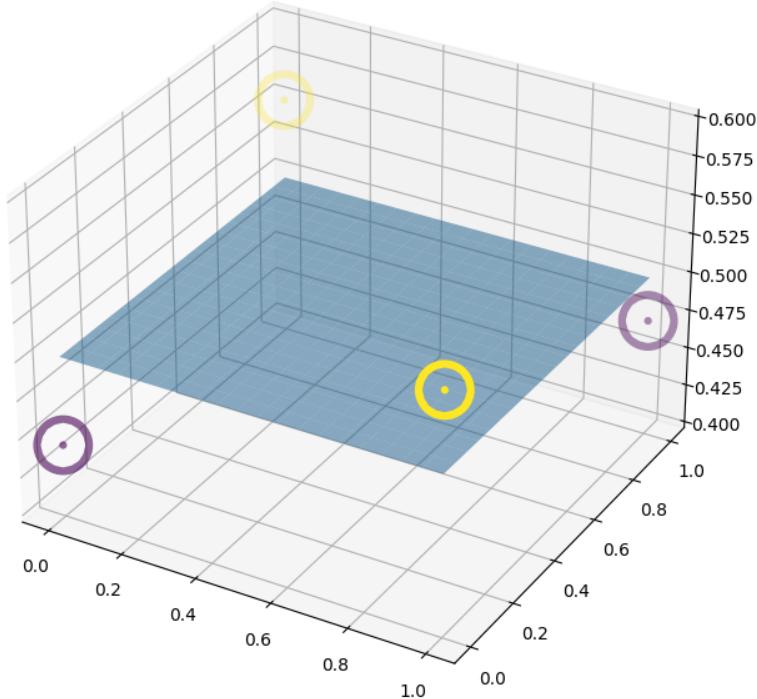


11.3 Artificial Neural Networks

Can be used for both regression and classification. For regression, something like the mean square error is used while for classification, crossentropy is used. From a probabilistic perspective, these make the ANN a maximum likelihood estimator. If additionally some presumption about the weights is made, i.e., a regularization is used such that the weights are drawn from a standard distribution, it becomes a maximum a posteriori estimator. The presumed weight distribution acts as a prior, such that the likelihood becomes a posterior.

The other way to look at ANNs is from the linear algebra perspective. Each layer finds a nonlinear transformation of the data into a higher- (in the first layers) or lower-dimensional (in the last layers) space. The effect of transforming a dataset into a higher-dimensional space is, that information about it becomes more obvious and the data can be separated more easily, as we've seen for the XOR-gate.

```
Image("img/plot_XOR_3d.png")
```



This way, the network can easily handle nonlinear datasets and find suitable transformations for any given data. (For those from AML, this is pretty much the kernel trick, but the ANN learns the kernels through training instead of us providing it.)

It's always good and sometimes necessary to normalize the data before feeding it into the network. One reason is that the optimization will work much better

```
import numpy as np

def multivar_gauss(vals, , Σ):
    dim = .shape[0]

    det_Σ = np.linalg.det(Σ)
    inv_Σ = np.linalg.inv(Σ)

    norm = np.sqrt((2*np.pi)**dim * det_Σ)
    expo = np.einsum('...k,kl,...l->...', vals-, inv_Σ, vals-)
```

```
#expo = (vals - ) * inv_Σ * (vals - )

return np.exp(- expo/2)/norm

grid_points = 40

X = np.linspace(-3, 3, grid_points)
Y = np.linspace(-3, 4, grid_points)
X, Y = np.meshgrid(X, Y)

= np.array([0.0, 1.0])
Σ_general = np.array([[ 1.0, -0.99], \
                      [-0.99,  2.2]])
Σ_boring = np.array([[1.0, 0.0], \
                      [0.0, 1.0]])

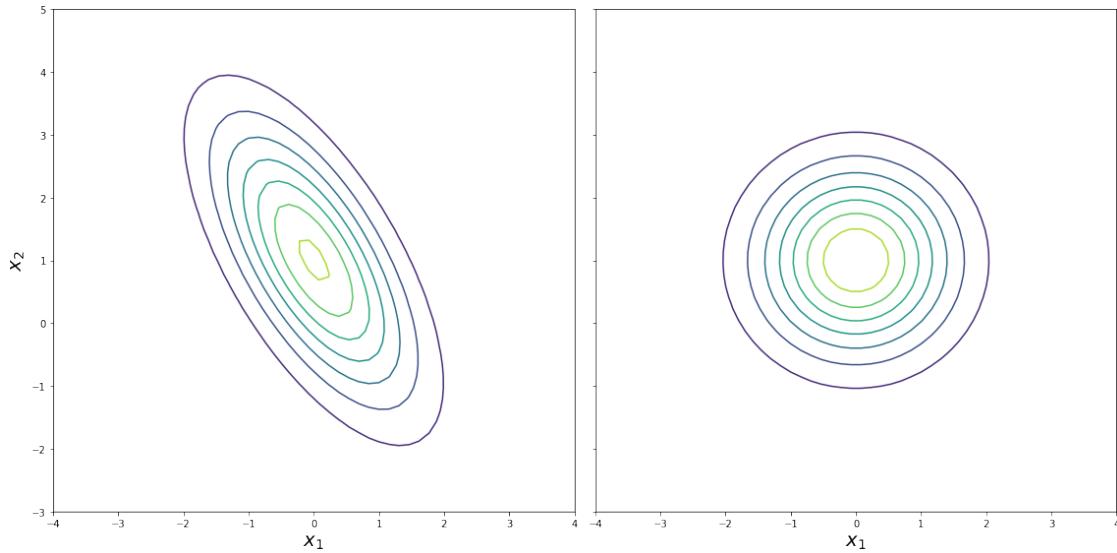
vals = np.array([X,Y]).transpose(1, 2, 0)

fig, axs = plt.subplots(1,2, figsize=(16,8), sharey=True)

axs[0].contour(X, Y, multivar_gauss(vals, , Σ_general))
axs[1].contour(X, Y, multivar_gauss(vals, , Σ_boring))

for ax in axs:
    ax.set_xlim([-4,4])
    ax.set_ylim([-3,5])
    ax.set_xlabel(r"$x_1$", fontsize=20)
    axs[0].set_ylabel(r"$x_2$", fontsize=20)

plt.tight_layout()
plt.show()
```



the other is that the terms in the hypothesis do not deviate too much from each other. Think of the condition number for system matrices. A huge condition number hints at one dimension of the problem being much larger than others. Or distorted finite elements, which yield bad results if, e.g., a rectangle is too wide compared to its width, since the Jacobian for the isoparametric concept isn't well-invertible.

The Universal Approximation Theorem says that an ANN with either a single hidden layer, sufficiently many neurons and some nonlinearity, or a single nonlinear neuron and arbitrarily many hidden layers, can approximate any given function arbitrarily well, given that it is piecewise invariant under composition ($f \odot g$ must be of the same type as f and g). Hence, we can use ANNs anywhere there's a function that we'd like to approximate. We'll go into more detail for how to identify possible applications in lecture 10. Nonlinear activations are usually only used inside of the network for regression tasks, with the last layer using linear activations, such that the full range of possible outputs can be produced. For normalized data, other activations may be used and sometimes yield much better results, but this is problem-specific. For classification, *softmax* can be used to provide a pseudo-probability for each component in the output. Usually, the data is *one-hot-encoded*, where each component indicates the probability of a sample belonging to the respective class.

$$0 \rightarrow \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

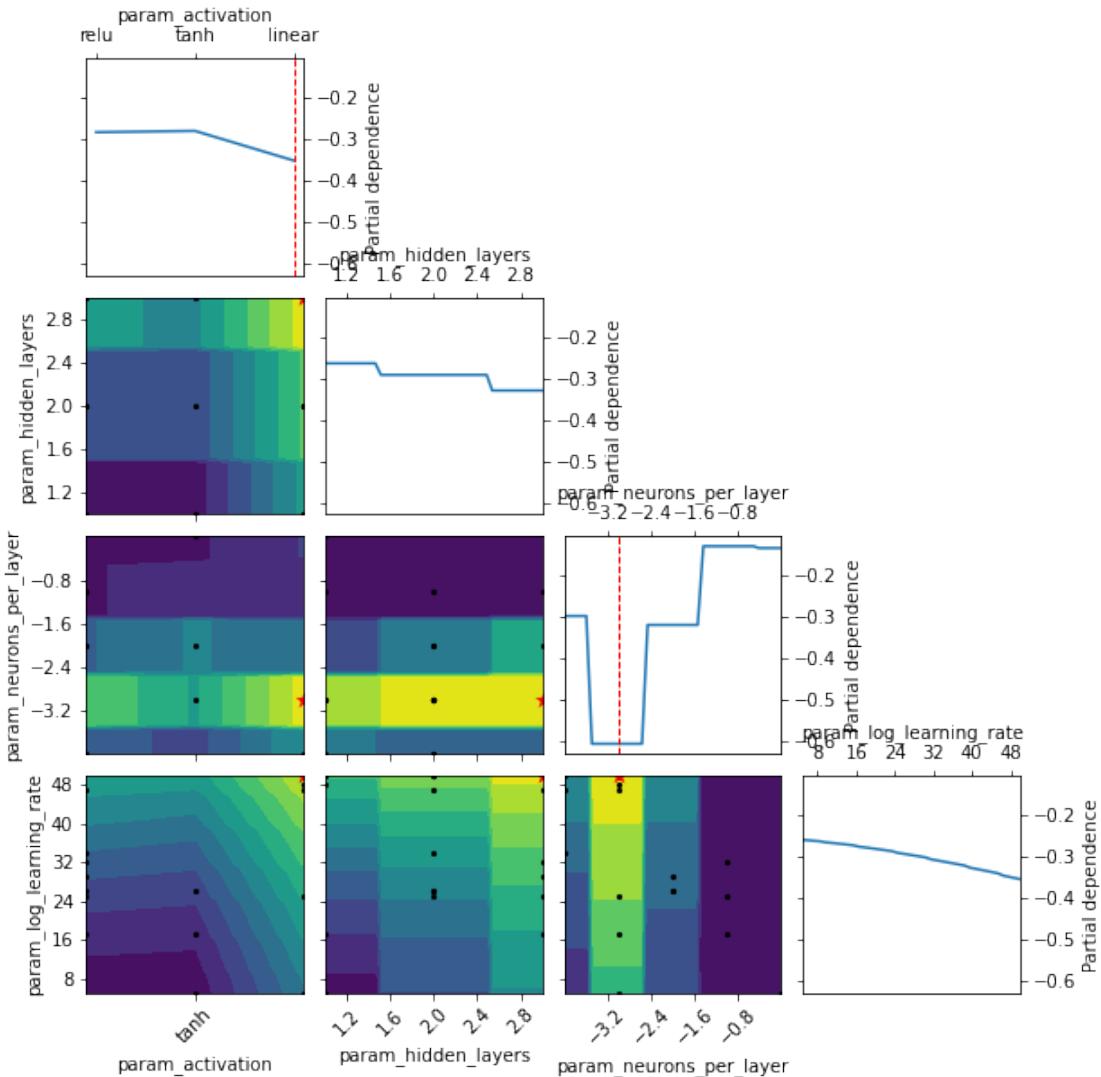
$$1 \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Training an ANN is a stochastic process and needs careful evaluation. The metric used to evaluate should be different from the optimization goal (see *Goodart's law*) and evaluated on a test set that was never used in any way during training. This is only the first step in evaluating a model. Since the training itself is stochastic, it should be performed several times (rule of thumb: at least ten

times) and the metric evaluated on the randomly chosen test set each time, such that a statistical analysis of the results can be performed. Usually the mean and variance already give good insight into how robust the model itself is.

Parameters that aren't trained but fixed before the training process are called *hyperparameters*. These need to be tuned for each specific task. This can be automated with *hyperparameter optimization*, of which we discussed three variants. Usually, Bayesian optimization is a good bet for finding suitable model parameters. For large projects, where large compute resources are available, an evolutional hyperparameter optimization usually yields better results.

```
Image("img/bayesian_hyperparameter_opti.png", width=1000)
```



11.4 Convolutional Neural Networks

Using a standard ANN for matrix-like data, e.g., images, works by flattening the input to vectors. Thereby pretty much all spatial information is at least obfuscated in the correlations between different features. CNNs retain spatial information by respecting local structures, and they are even a bit invariant and equivariant under sufficiently small translations. This happens via weight sharing in form of convolutional filters, which are small matrices, with which parts of the input matrix are convolved, i.e., multiplied and accumulated with. Not always, but usually, convolutional layers are followed by pooling layers, which heavily downsample the information contained in feature maps. The last layers can either be convolutional, if the outputs are supposed to be matrices again, or fully-connected, for regression or classification. The same loss functions are used as in the standard ANN case.

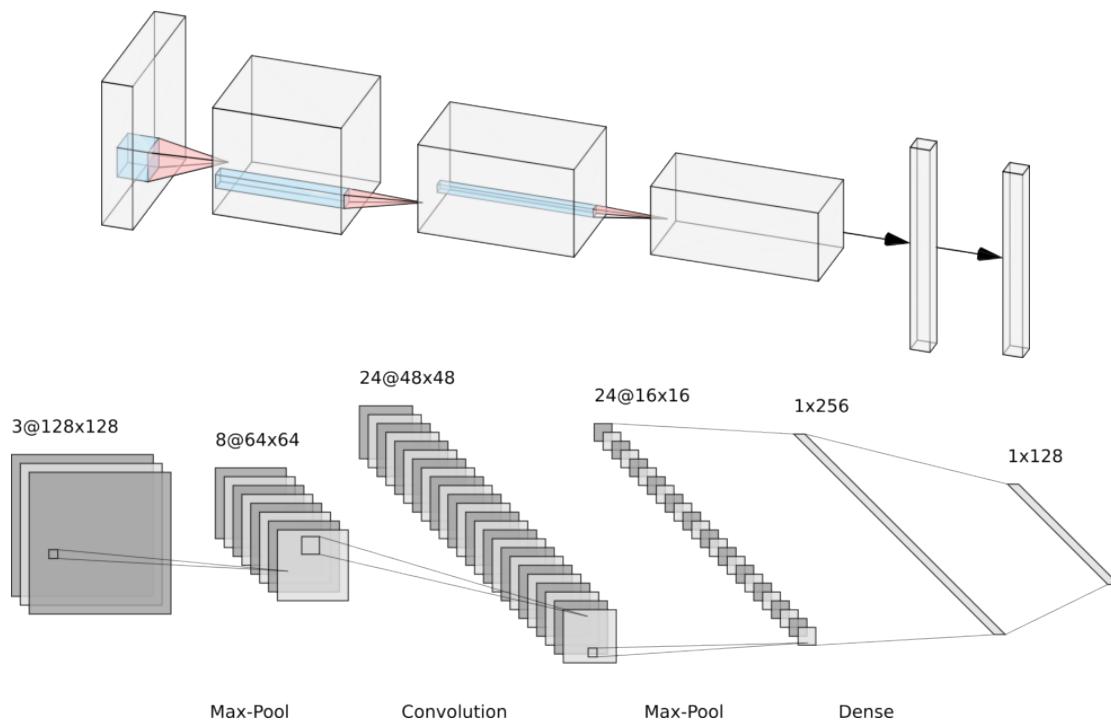
```
alex = mpimg.imread("img/cnn_example_alexnetstyle.png")
le = mpimg.imread("img/cnn_example_lenetstyle.png")

fig,axs = plt.subplots(2,1, figsize=(16,9))

axs[0].imshow(alex)
axs[1].imshow(le)

for ax in axs:
    ax.axis("off")

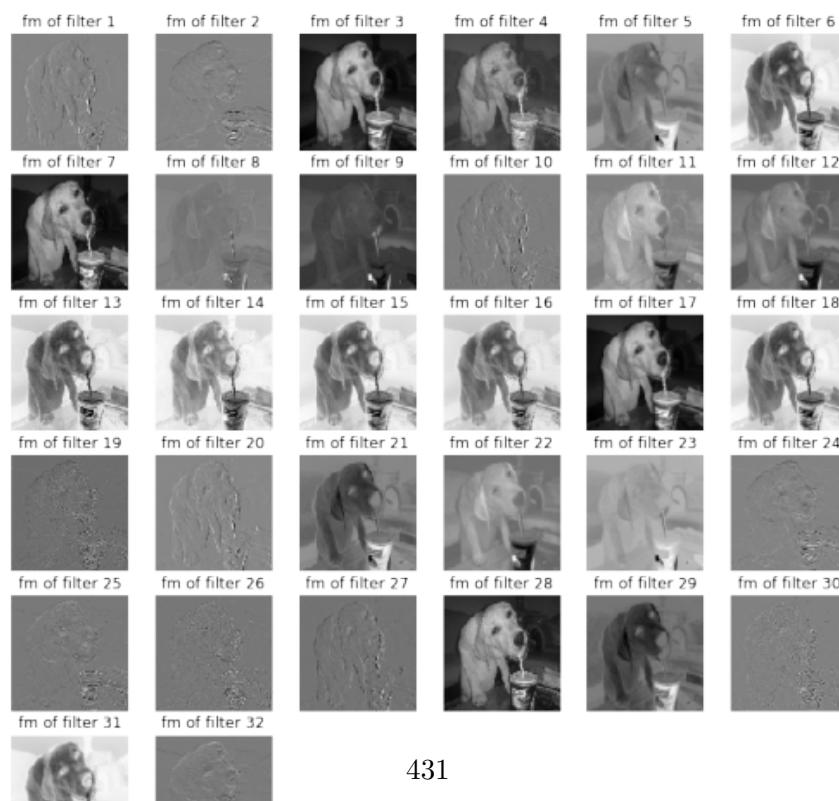
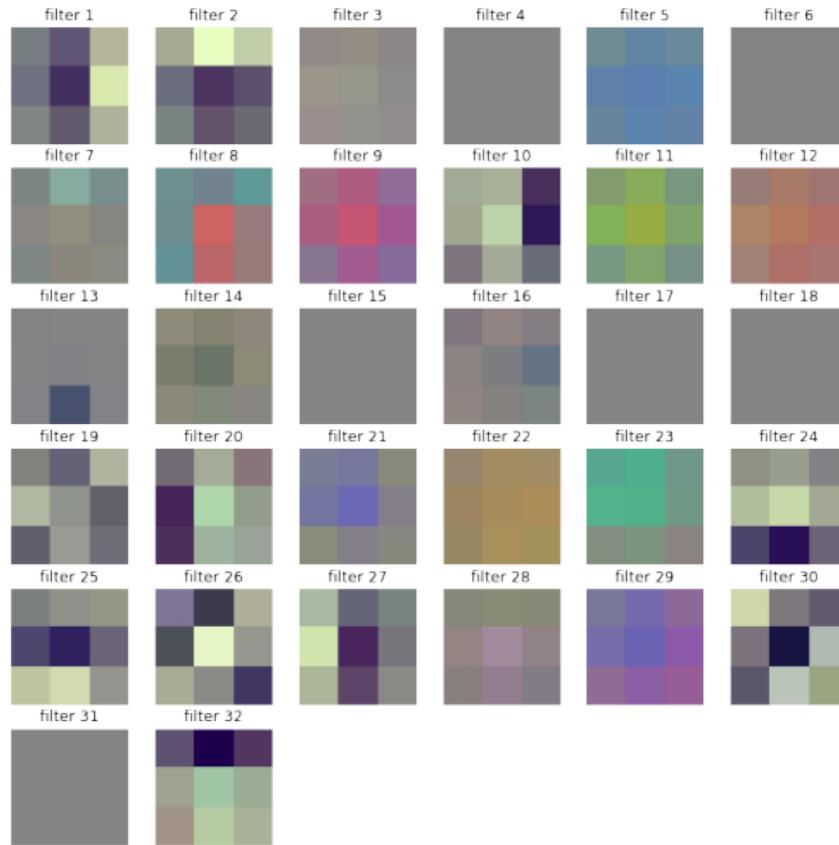
plt.tight_layout()
```



Finding a good starting point for optimization is a difficult but crucial task. In case of image-based tasks, *Transfer Learning* can be used as a good initial point in the loss landscape. The idea is to use a pretrained CNN, usually trained on imagenet data, and attach custom layers. Depending on whether data for the task at hand is scarce or abundant, the full network can be trained or just the custom layers.

Early stage filters in the network learn abstract, but simple geometric features, like edges and simple patterns, or perform Gaussian filtering or similar tasks. In later stages, filter combine the low-level features to higher-level abstractions, like combining vertical and horizontal edges to rectangles, or all kinds of edges to circles, which are later turned into, e.g., eye or mouth detectors. At even later stages, these are combined to form face or species detectors.

```
Image("img/filters_and_effects.png", width=900)
```

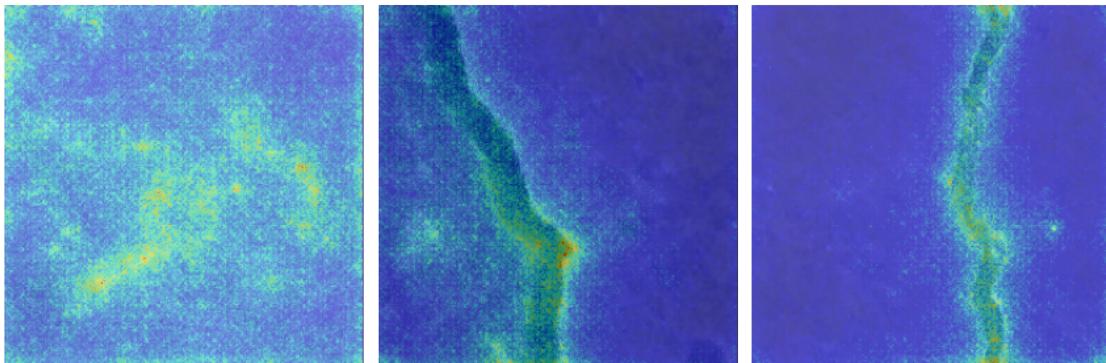


11.5 Semantic Segmentation

Semantic segmentation refers to a pixel-wise classification, where the class labels are output as colors in an image that has the same dimension as the input image. The colors represent the certainty with which the network classified the pixel. The training data consists of the images to be classified and bitmasks, which provide the color labels for each pixel and are the ground truth of the task. The loss function depends on the problem itself. If there is imbalance between classes, i.e., some classes are overrepresented or usually much larger than others, some form of *Intersection over Unity*-loss makes sense to use. There are more sophisticated losses though, which we did not cover in the course.

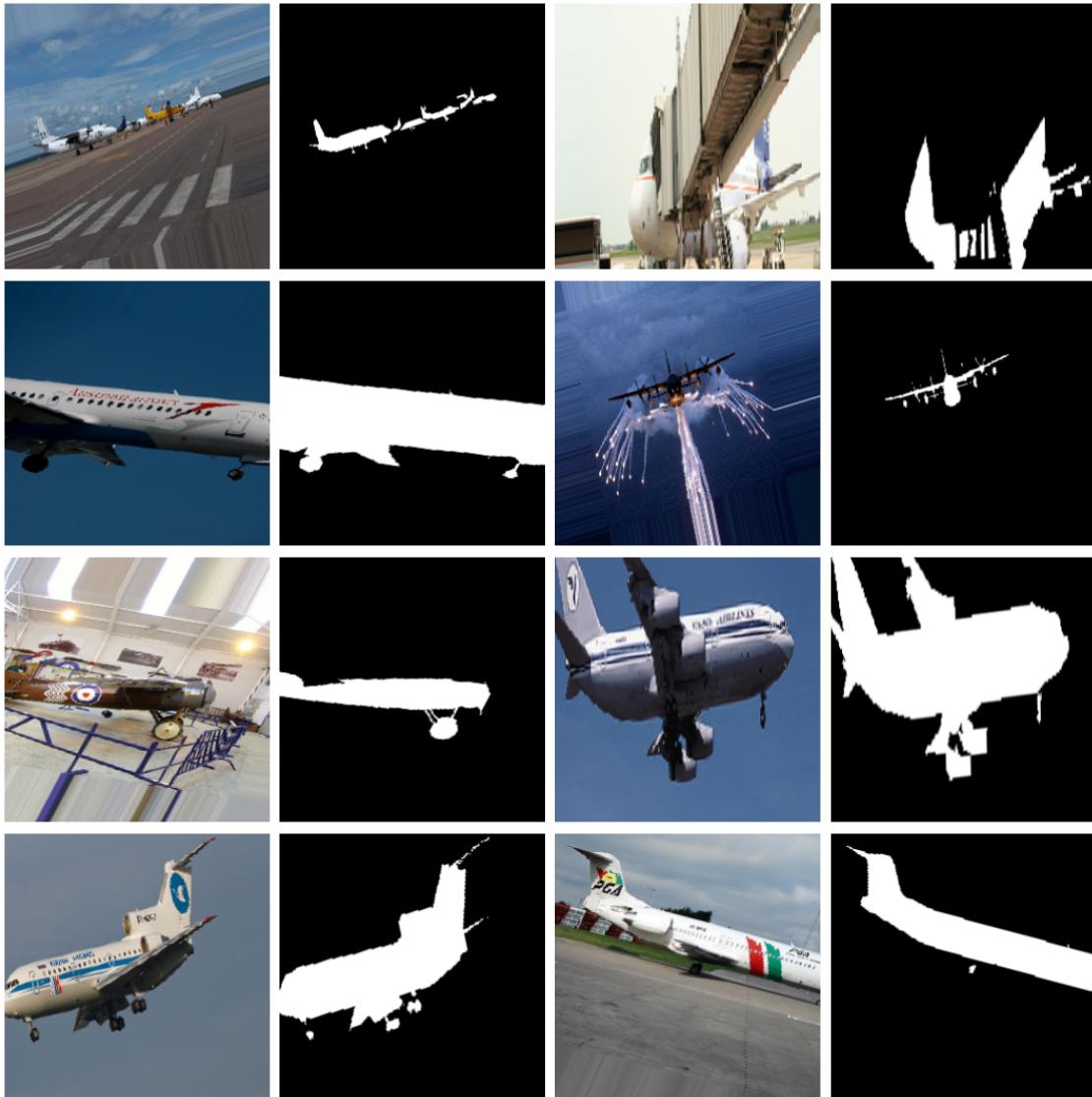
Saliency maps and Class Activation Maps can be used to visualize which parts of the image a networks is concentrating on. Saliency maps can even be used as an unguided segmentation tool, as was visible in the example with the crack detection in concrete.

```
Image("img/saliency_crack.png")
```



Creating the bitmasks for training is very time-consuming and often difficult. Hence, in almost all cases some kind of *Data Augmentation* ist used. Several sensible transformations are applied to the input images and the masks, which let the network think these are new samples. E.g., a mirrored image is different from the original to the network. This is a way to teach an ANN *invariance* under some transformation. Common transformations include translations, rotations, reflections, shearing, zooming, cropping, color, brightness, hue and contrast abberations, perspective changes and more. For every problem instance these should be carefully considered and checked whether they produce sensible variations of the inputs.

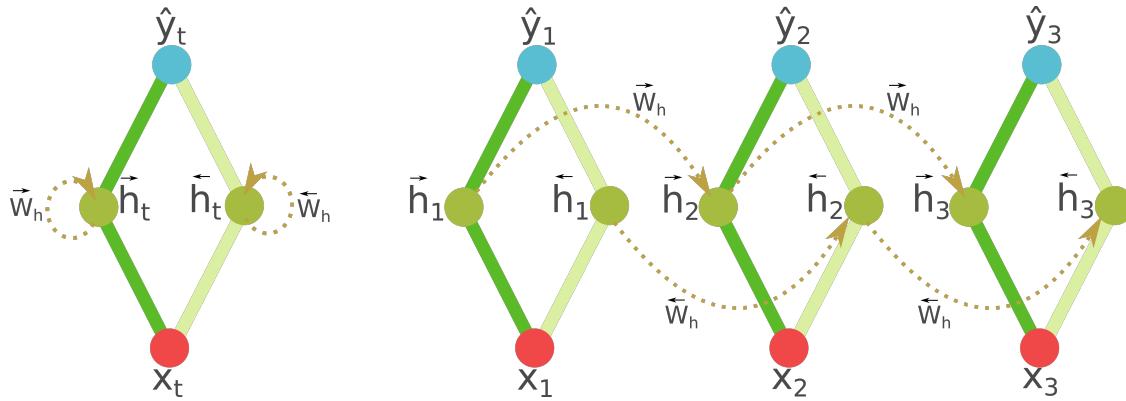
```
Image("img/data_augmentation_aeroplanes.png")
```



11.6 Recurrent Neural Networks

It's possible to predict time series with standard ANNs (or CNNs for matrices), but the models become quite large quickly. The network architecture could be that the input layer takes in n samples to provide m results in the output layer. A more sophisticated way to create such a model is to use *recurrent neural networks*. If the change of a system from one time step to the next is approximately constant over the sequence range, the weight matrices from one time step to the next don't need to change and can be used recurrently. The recurrent layers in such a network contain a *hidden state*, that carries over information from former time steps to the future (or also back to the past in case of bidirectional RNNs).

```
Image("img/rnn_bidirectional.png")
```



RNNs have problems with exploding and vanishing gradients, to their recurrent nature. Some manual ways to reduce the problem exist in the form of truncated gradients or truncated backpropagation through time, but usually different architectures are used. We discussed GRUs and LSTMs, which both provided competing pathways for the gradient to flow through while backpropagating. For exploding gradients, the past hidden state is used more, while for vanishing gradients, the future hidden state is used predominantly for determining the next hidden state.

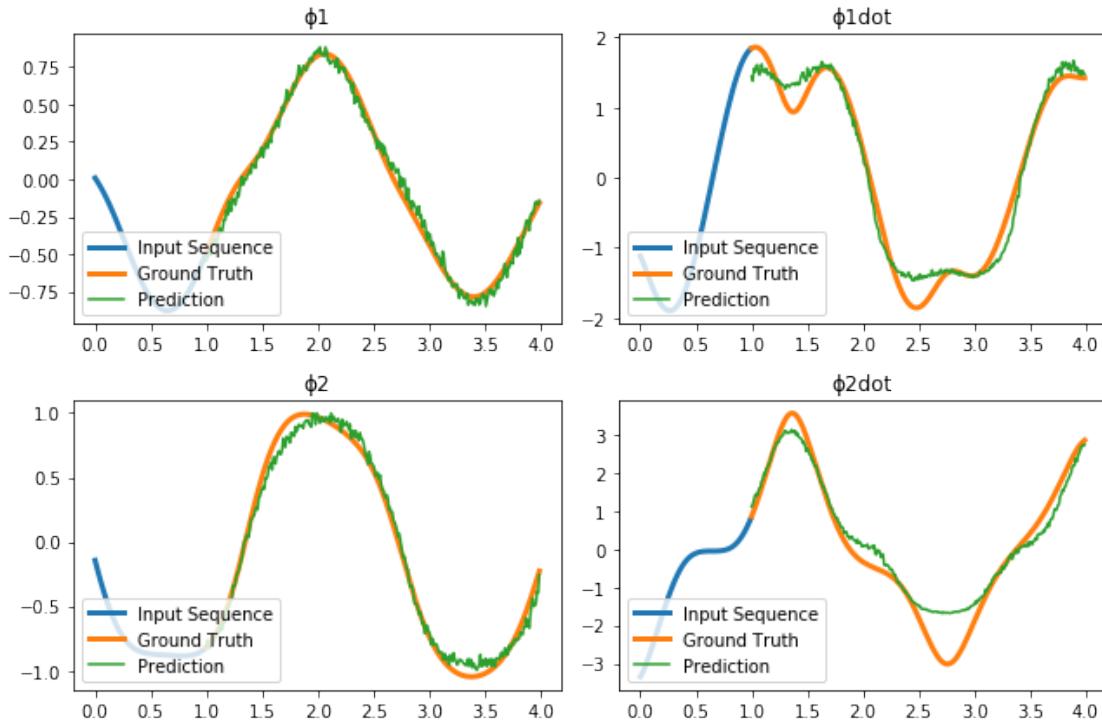
$$\mathbf{h}_t = \mathbf{f} \odot \mathbf{h}_{t-1} + (1 - \mathbf{f}) \odot \mathbf{g} \quad (199)$$

At the same time, they retain information from past time steps for longer than standard RNNs. LSTMs were famously used in natural language processing, especially as translator models, since the meaning of a sentence does not only depend on the words used in a sentence, but also on the order in which they appear.

Recurrent layers can be combined with CNN layers, e.g., as ConvLSTMs.

The training data for RNNs consists of a series as input, and a later series for output, such that the network learns to take a series of inputs and produce a future series of outputs.

```
Image("img/lstm_double_pendulum.png", width=1000)
```

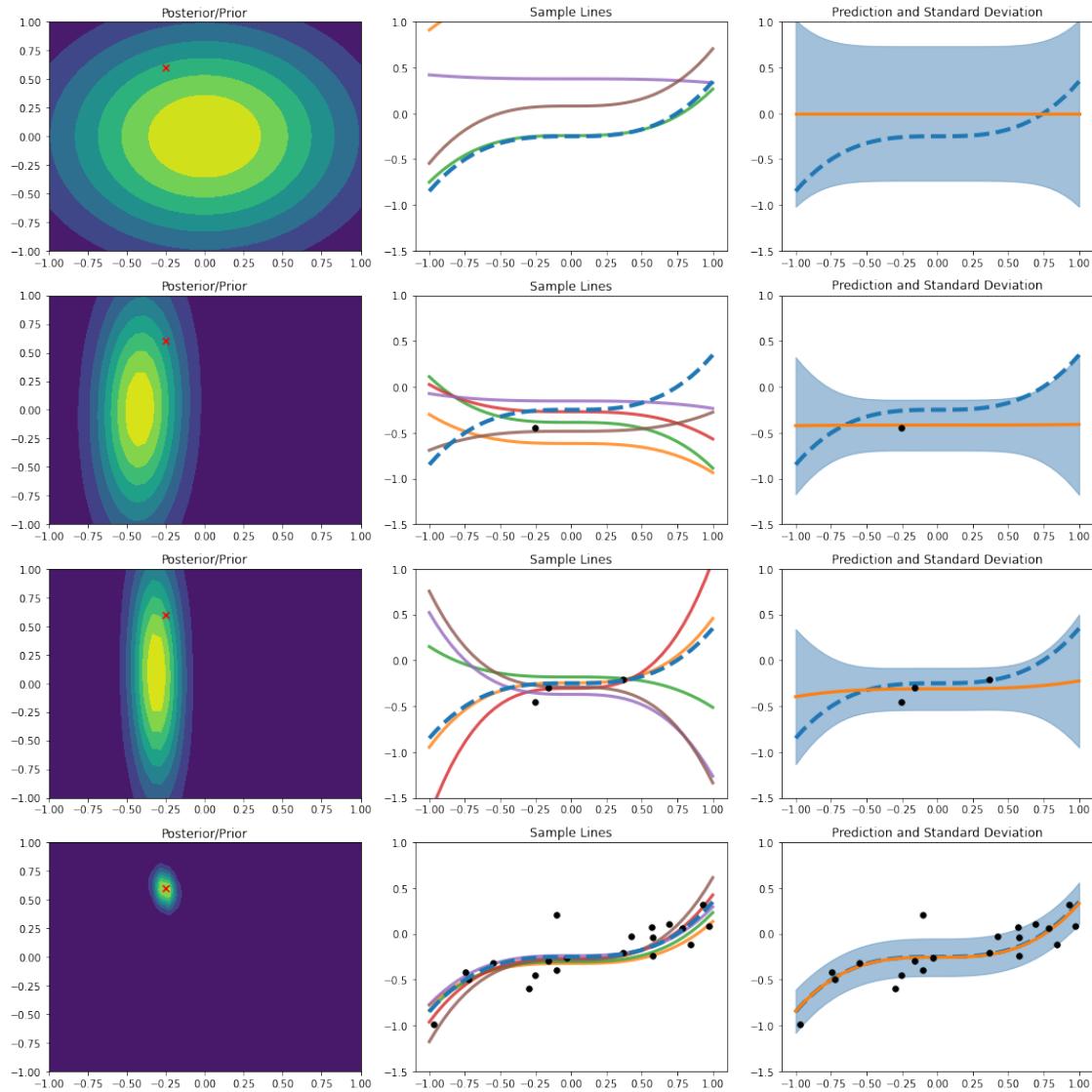


11.7 Probabilistic and Statistical Methods

We saw how ANNs can be thought of as maximum likelihood estimators or maximum a posteriori estimators respectively, if constraints are placed on the weights. Early stopping and dropout are more advanced regularization techniques.

We also introduced Bayesian regression, which iteratively samples from the posterior of the previous iteration (the current iteration's prior) to produce a sample population, from which a hypothesis can be formed and updated with more samples.

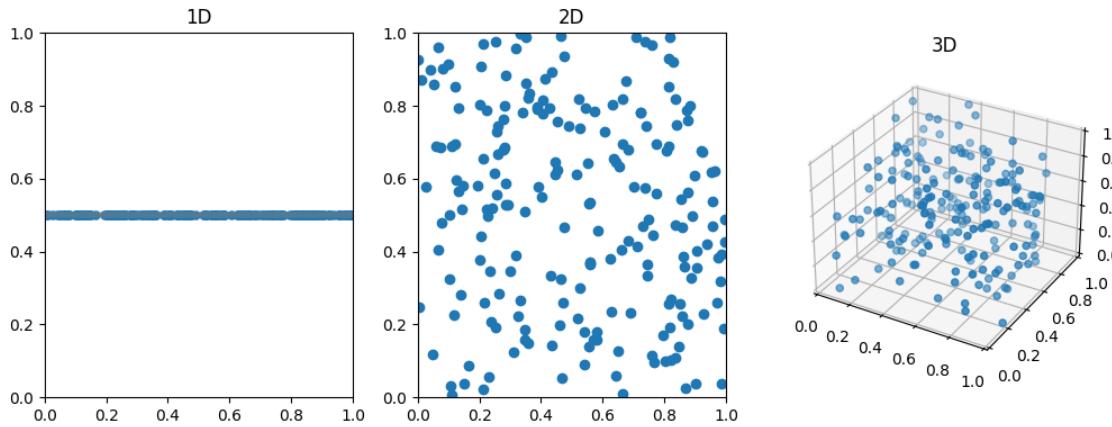
```
Image("img/plot_bayes_reg.png")
```



11.8 Dimensionality Reduction and Clustering

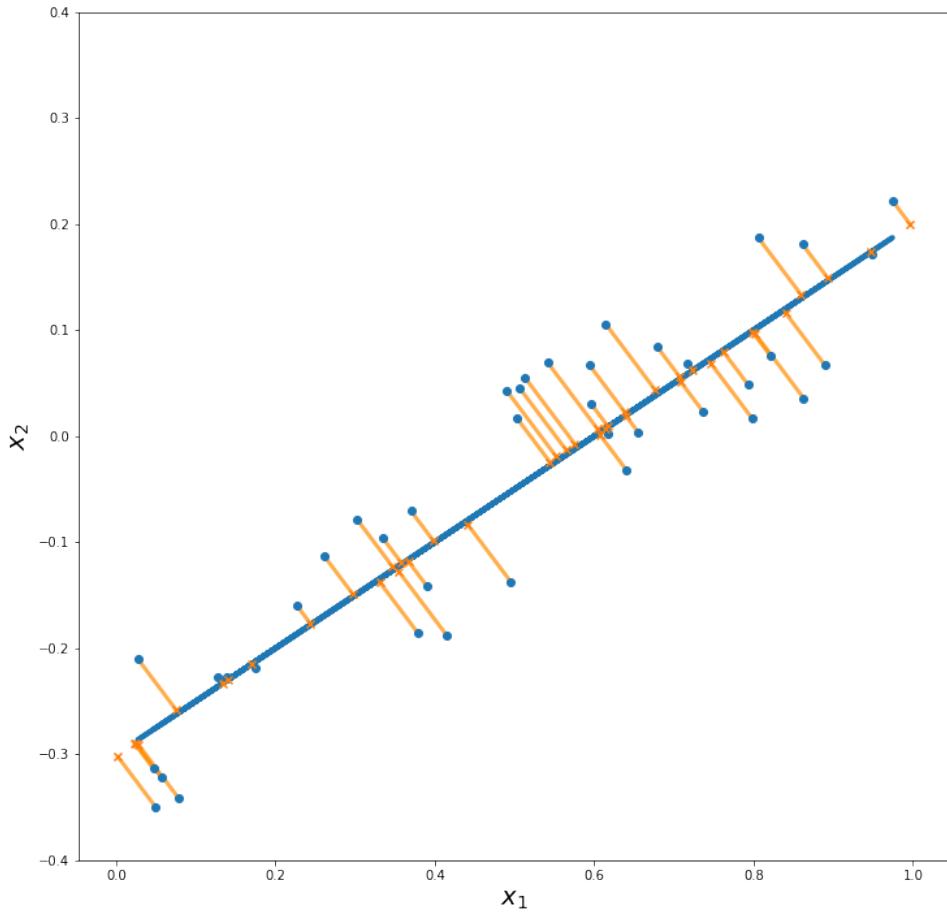
The *curse of dimensionality* makes pretty much everything in high dimensions difficult or intractable. The dimension of a problem in case of ML is the number of features in the dataset. The higher the dimension of a problem, the more samples are necessary to accurately probe it for information, for example for optimizing a high-dimensional loss landscape.

```
Image("img/plot_dimensions.png")
```



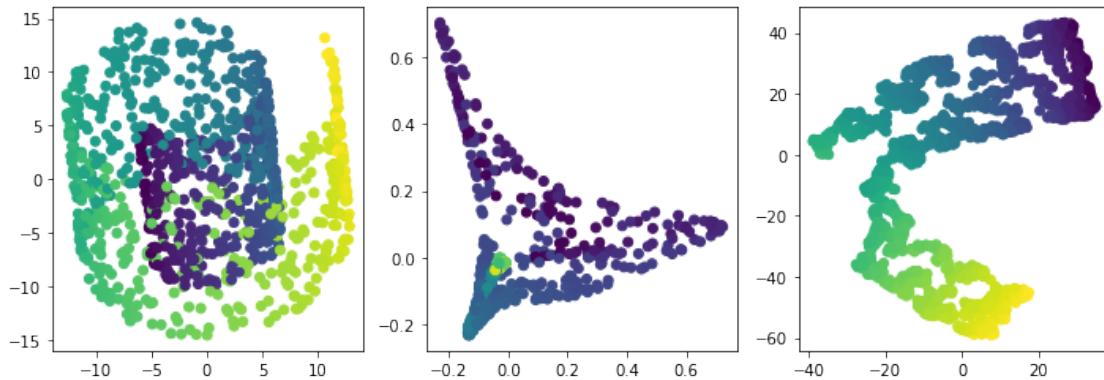
PCA can be used to reduce the dimension of a problem and getting rid of high-frequency noise in the data. This is sometimes called *PCA-whitening*. PCA finds the directions in which the data shows the highest variance, hence containing the most amount of information. Geometrically, it fits a hyperellipsoid to the data, algebraically, it is the SVD of the data matrix (or eigendecomposition of the covariance matrix). (side note: *kernel-PCA* is a simple generalization for nonlinear datasets)

```
Image("img/plot_correlation_lines.png")
```



We used t-SNE as a nonlinear dimensionality reduction technique. The plots don't convey much meaning, but show clusters in the high-dimensional datasets and often reveal hidden structure that other techniques cannot find. It's usually used in conjunction with other dimensionality reduction techniques, mostly PCA, where the other technique is used as an initialization for t-SNE. The most important hyperparameter to tune is the *perplexity*, which basically states how many points are considered neighbors for computing the similarity measures.

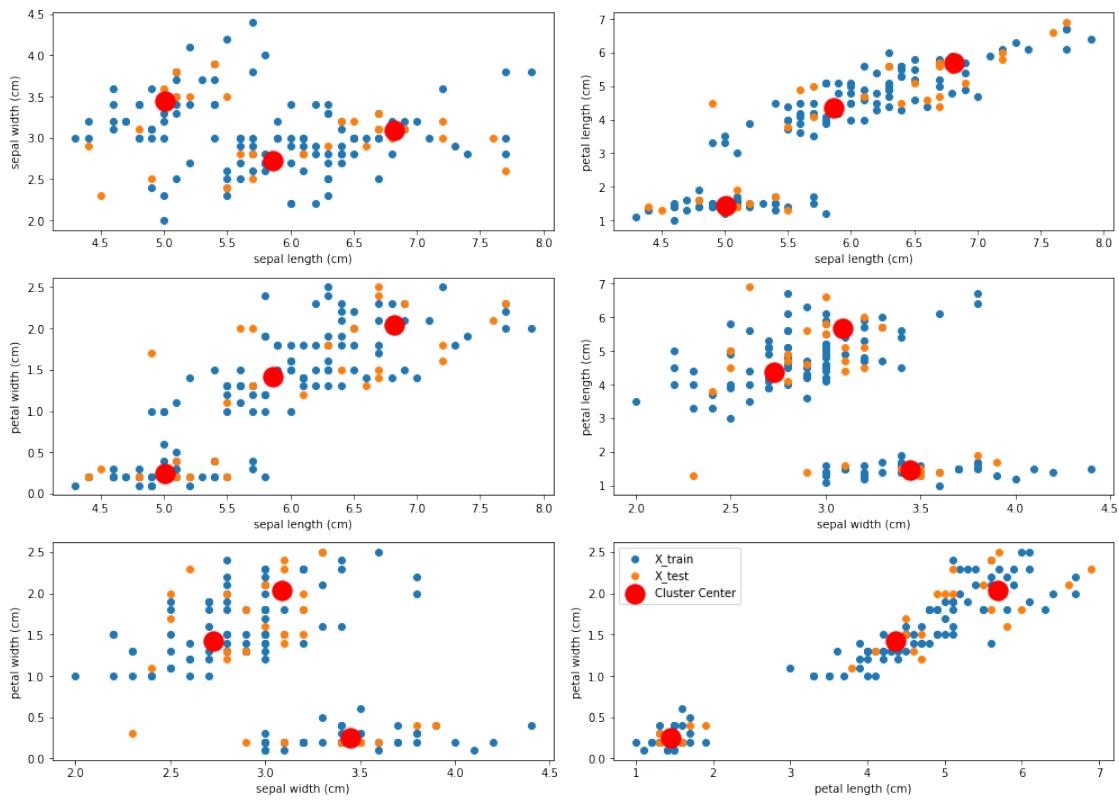
```
Image("img/t-sne_unfold_swiss.png")
```



PCA can also be used generatively, since the reduced representation of the data models a *latent space*, which we'll implore a bit more thoroughly in the next lecture.

Lastly, we got to know K -means as a simple clustering algorithm, that finds cluster centroids by minimizing the intra-cluster variance.

`Image("img/k-means_iris.png")`



K -nearest neighbors can be used as a simple classifier for datasets with obvious clusters, so it may

be a feasible post-processing step after t-sne or K -means.

12 Generative Modeling

12.1 Generative Models

So far in the course, we mostly used supervised learning techniques where pairs of data $(x^{(i)}, y^{(i)})$ were available and we mostly tried finding a mapping between the two in some way for a regression or classification model. We also talked about a few unsupervised learning techniques where only the $x^{(i)}$ were available to us, like K -Means for clustering and PCA for dimensionality reduction. The goal there was to identify a hidden representation of the data, or some underlying structure.

The image below compares a few of the techniques for classification we discussed on three datasets, where each has its own unique twist. The code was slightly modified and taken from the [scikit-learn user guide](#):

```
# Code source: Gaël Varoquaux
#              Andreas Müller
# Modified for documentation by Jaques Grobler
# License: BSD 3 clause

import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

h = .02 # step size in the mesh

names = ["Nearest Neighbors", "Linear SVM", "RBF SVM",
         "Decision Tree", "Random Forest", "Neural Network", "AdaBoost",
         "Naive Bayes"]

classifiers = [
    KNeighborsClassifier(3),
    SVC(kernel="linear", C=0.025),
    SVC(gamma=2, C=1),
    DecisionTreeClassifier(max_depth=5),
    RandomForestClassifier(max_depth=5, n_estimators=10, max_features=1),
    MLPClassifier(alpha=1, max_iter=1000),
```

```

AdaBoostClassifier(),
GaussianNB()]

X, y = make_classification(n_features=2, n_redundant=0, n_informative=2,
                           random_state=1, n_clusters_per_class=1)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [make_moons(noise=0.3, random_state=0),
            make_circles(noise=0.2, factor=0.5, random_state=1),
            linearly_separable
            ]

figure = plt.figure(figsize=(27, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=.4, random_state=42)

    x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                          np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(['#FF0000', '#0000FF'])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data", fontsize=20)
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.6,
               edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

```

```

# iterate over classifiers
for name, clf in zip(names, classifiers):
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    if hasattr(clf, "decision_function"):
        Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
    else:
        Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

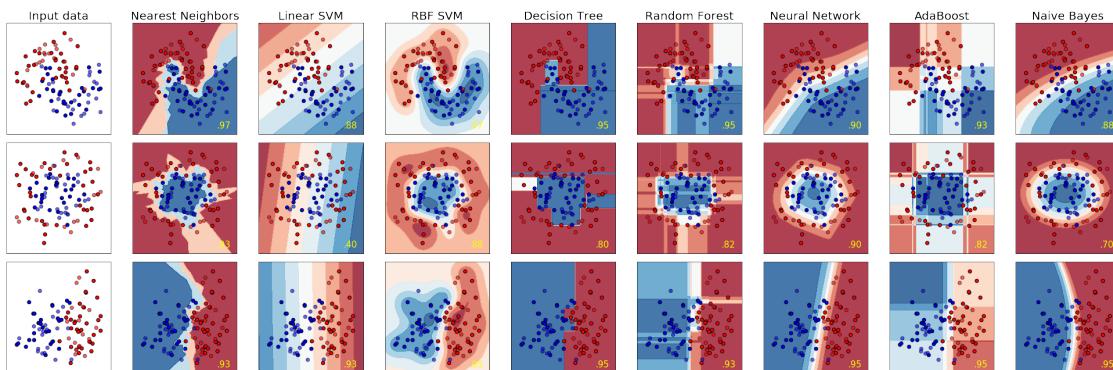
    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    ax.contourf(xx, yy, Z, cmap=cm, alpha=.8)

    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright,
               edgecolors='k')
    # Plot the testing points
    ax.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright,
               edgecolors='k', alpha=0.6)

    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    if ds_cnt == 0:
        ax.set_title(name, fontsize=20)
    ax.text(xx.max() - .3, yy.min() + .3, ('%.2f' % score).lstrip('0'),
            size=15, horizontalalignment='right', color='yellow')
    i += 1

plt.tight_layout()
plt.show()

```



For playing around a bit with parameters of artificial neural networks for classification as well as regression, the [TensorFlow Playground](#) is a nice way of understanding different configurations.

In contrast to these types of model, **generative modeling** tries to perform a *density estimation* of the available data. This is an important problem in unsupervised learning. Density estimation roughly means estimating the underlying probability distribution in such a way, that new samples from it can be drawn that are similar to the original training data. The idea is to assume that some true data PDF $p_{\text{data}}(x)$ exists, and that it's possible to take samples $X = \{x_1, x_2, \dots, x_M\}$ from it (the training data), and using these samples to estimate the original PDF by a *model PDF*

$$p_{\text{model}} \approx p_{\text{data}}$$

Sampling from this approximated model PDF would then produce samples that are similar to the training data. So a generative model learns a *joint probability*

$$p(x, y) = p(y|x)p(x)$$

while a *distributive model* learns a *conditional probability/score*

$$p(y|x)$$

i.e., a classifier. These are related by Bayes' theorem. If we have a good model for a prior $p(x)$, we can use it to calculate the joint probability from a classifier, or vice versa, but this is rarely the case. Here, we would like to model the actual underlying data distribution $p(x)$.

There are various applications for generative models. The most prominent examples come from image analysis, where the models are given labels or some other form of output specifier, then the models produce an example of the specified label. A slightly terrifying example is [thispersondoesnotexist](#), which is a *generative adversarial network* trained by researchers at NVIDIA to produce realistic-looking human faces from arbitrary sets of labels (the sources are linked at the bottom of the web page). At first glance or when reducing the image size it is extremely difficult (for humans!) to decide whether such a generated face is actually real or fake. For larger image sizes it is often possible to find minuscule details, especially around the teeth or eyes that make some of the pictures "feel unnatural", although that's obviously not a proof. A similar approach is taken by [thiscatdoesnotexist](#), although it was trained on a much smaller dataset, the model is much much smaller and the output image resolution is also much lower. Here, you can often find artifacts that will disappear with longer training and larger datasets.

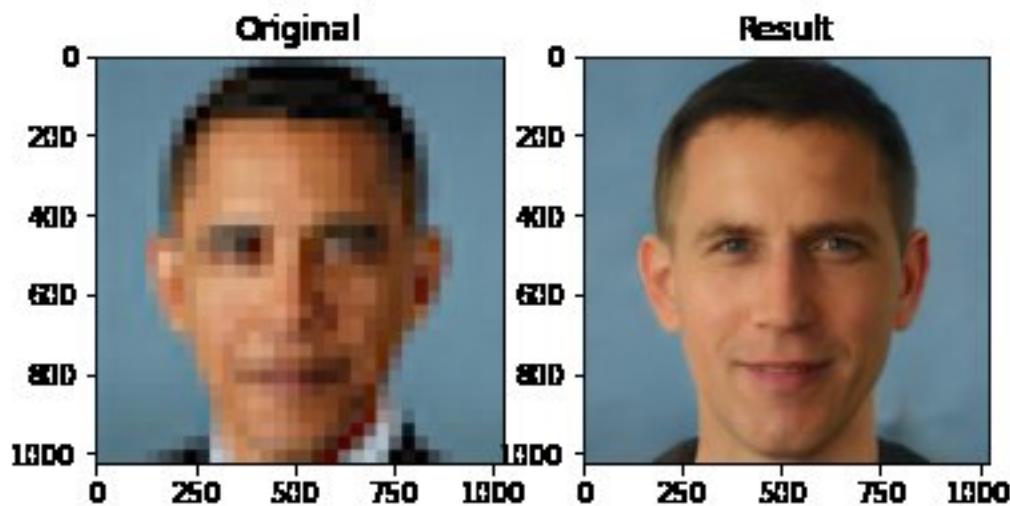
We've also reached a point in time where it's almost easily possible to manipulate video and speech data to make arbitrary people realistically say what we want them to say. The technique is called *Deep Fake* and already in use in various ways. Researchers at MIT produced a fake clip of President Richard Nixon [performing the speech](#) that was written for the case that the moon-landing would go wrong. You can find some information about how it was developed [here](#). This is a very simple example of what can be done and you'll easily find various higher-resolution, more modern examples.

Moving away from the doomsday applications, generative models can also be extremely useful for upscaling noisy images (try for yourself for example [here](#)), or repairing missing parts of an image.

```
from IPython.display import Image
Image("img/pancakefishing_upscaled_2x.png", width="1000")
```



```
Image("img/notobama.jpg", width="800")
```



While this works quite well in general, there are some caveats and you need to know your model extremely well, as you can see above (taken from [this tweet](#)), or otherwise retrieve results that may look plausible, but actually do not represent the desired result. This is an example where *sample*

bias in your training data can make your ML model produce nonsensical solutions with utmost confidence.

Then there's also generative modeling for design suggestions, which we'll briefly implore in the next lecture. It's possible to train generative models to generate e.g. stable structures under some constraint, like using few amounts of material (see for example [this marketing page](#) to see what General Motors did or [this one](#) for Airbus)), or in the context of multiscale problems generating microstructures that create a desired macroscopic effect, where for example researchers at MIT produced a microstructured material that is extremely resistant to impact while at the same time being as lightweight as possible. This doesn't mean that actual product designers will become obsolete, as most designs that are generated do not exactly look pleasant to the eye or work in all contexts. It does mean that generative design is already employed in a wide variety of product landscapes and assists product designers in exploring completely new design worlds.

Yet another application is image-to-image translation, where in the image below, aerial images of a city were transformed into a semantic map. Also recently, [AI Dungeon II](#) was announced, which uses a generative text model called GPT-3, developed by openAI (of DOTA 2 AI fame, but their blog is absolutely worth checking out for up-to-date information about developments in Machine Learning). This is a Zorg-like text-based RPG that generates narratives based on user input.

A much more tame application could be in medicine, where (especially labeled) patient-specific data is extremely scarce. Generative models could generate data very similar to actual patient data that can be used to test models on, or to develop classifiers from small sets of ground truth.

There are different types of generative models:

- Autoregressive Models (pixelRNN, pixelCNN, ...), which explicitly determine the probability density of the training data. They're called *autoregressive* because pixel values are generated based on pixel values in the vicinity of the area to be generated, e.g., columns of pixels are generated based on previous columns of pixels.
- Latent Variable Models (AutoEncoder, Variational Autoencoder, ...), which explicitly model the density of the training data.
- Generative Adversarial Networks, which do not model the underlying probability distribution explicitly (at least not in an understandable way), but sample from it directly.

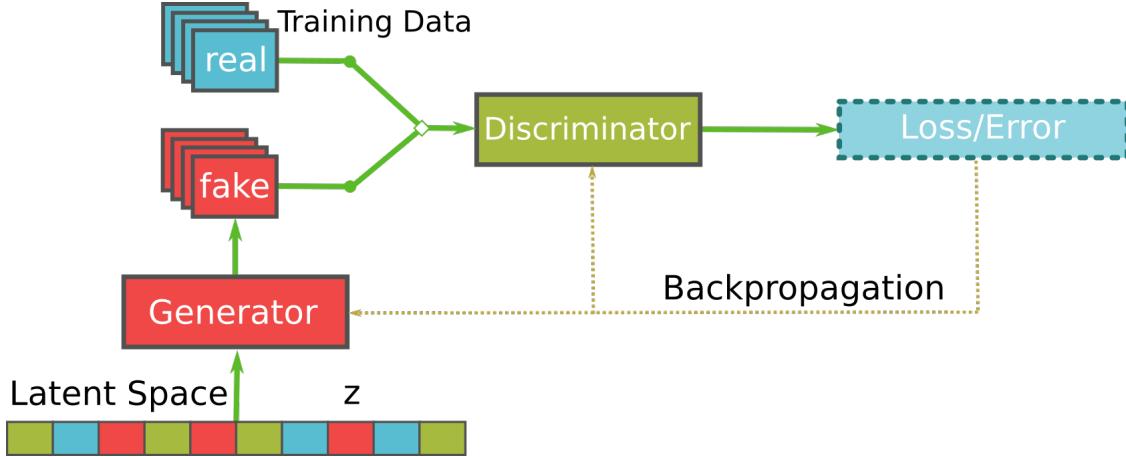
In general, the basic idea common to all of these models is to use Deep Neural Networks to model the prior distribution $p(x)$.

12.2 Generative Adversarial Networks

Generative Adversarial Networks (GANs) do not explicitly model the probability distribution of the training data set but instead directly produce samples from it. The input to a GAN is a random noise vector. The GAN transforms this random noise into a sample from the underlying data distribution.

A GAN consists of two deep neural networks; A **generator** (G) and a **discriminator** (D). The generator takes random noise and tries to transform it into a sample from the data distribution, the discriminator tries to distinguish between the output of the generator and real samples from the training data, like a classifier. The whole model is called *adversarial* because both networks try to outsmart each other, making both of them perform better and better in the training process where each relies on the performance increase of the other.

```
from IPython.display import Image
Image("img/gan_sketch.png", width="1000")
```



The input to the Generator G is $z \in \mathbb{V} \subseteq \mathbb{R}^N$, a random noise vector coming from a **latent space** \mathbb{V} , then outputs a fake sample \tilde{x} , labeled as 0. The discriminator D takes this produced fake sample \tilde{x} as input, along with a real data sample x labeled as 1 (this is a simplification for now, actually they get labeled 0.9, because otherwise the discriminator would work too well right from the start). It looks at batches of real and fake data and outputs a prediction that is used as input for a loss function that gives the probability for the data being fake or real. This loss function provides the error that is used to train both the weights of G and D at the same time.

12.2.1 Zero Sum Game

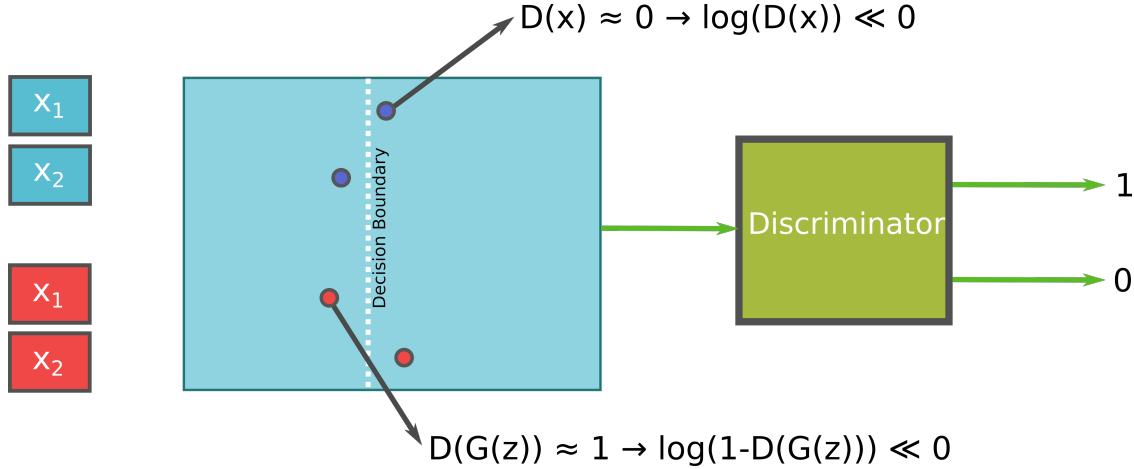
GANs try to reach a *Nash equilibrium* which essentially means they play a *zero sum game*, sometimes also called the *minimax game*:

$$\min_{w(G)} \max_{w(D)} L(x, z; w^{(G)}, w^{(D)}) = \min_{w(G)} \max_{w(D)} [\mathbb{E}_{x \sim p_{\text{data}}} [\log D(x)] + \mathbb{E}_{z \sim p_{\text{model}}} [\log(1 - D(G(z)))]] \quad (200)$$

There is a lot to digest here. This total loss function depends on two sets of parameters here, the parameters of the generator $w^{(G)}$ and the parameters of the discriminator $w^{(D)}$. These are optimized alternatively. First, the loss is maximized with respect to the parameters of D , then in the new state it is minimized with respect to the parameters of G . The left term in this loss, the expectation value of $\log D(x)$, is calculated for the real training data. The right term, the expectation value of $\log(1 - D(G(z)))$, is calculated over the fake, generated data. Assuming that we have an equal number of real and fake samples, this is very similar to binary cross entropy. Both $D(x)$ and $D(G(z))$ provide outputs in the range $[0, 1]$. Ideally, when a data sample is real, $D(\cdot)$ should output 1 and when the input comes from G , it should output 0.

Initially, D is untrained and its weights chosen randomly. A typical starting situation looks like this:

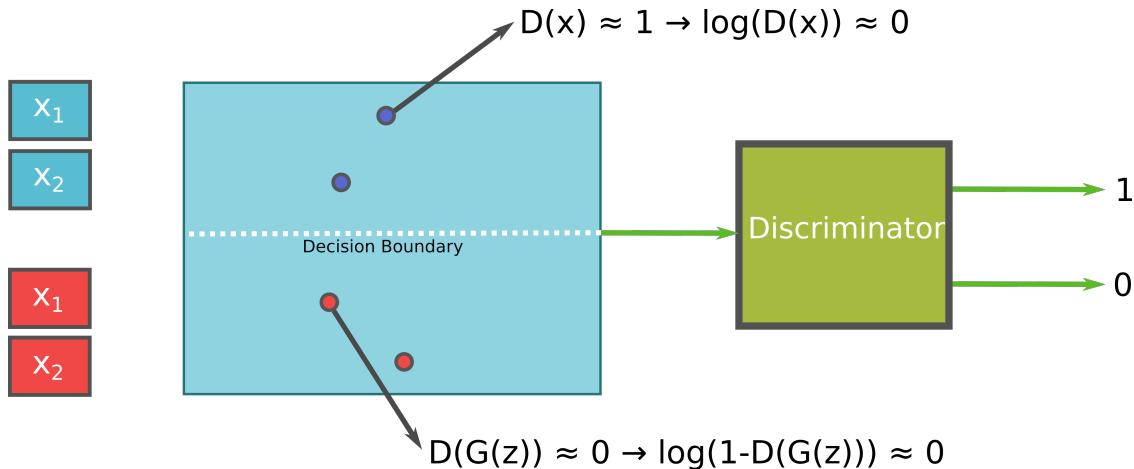
```
Image("img/gan_untrained.png", width="1000")
```



The dashed white line is the initial decision boundary. One of the real data points is misclassified, and one of the fake data points is misclassified. For the misclassified data point x_1 , $D(x_1)$ is close to 0 and hence, $\log D(x_1)$ becomes a large negative number. The same thing happens for the misclassified fake data sample x_4 below, where $D(x_4)$ becomes close to 1, such that $\log(1 - D(x_4))$ also becomes a very large negative number.

As soon as the discriminator is trained better, the situation looks more like this:

```
Image("img/gan_trained.png", width="1000")
```



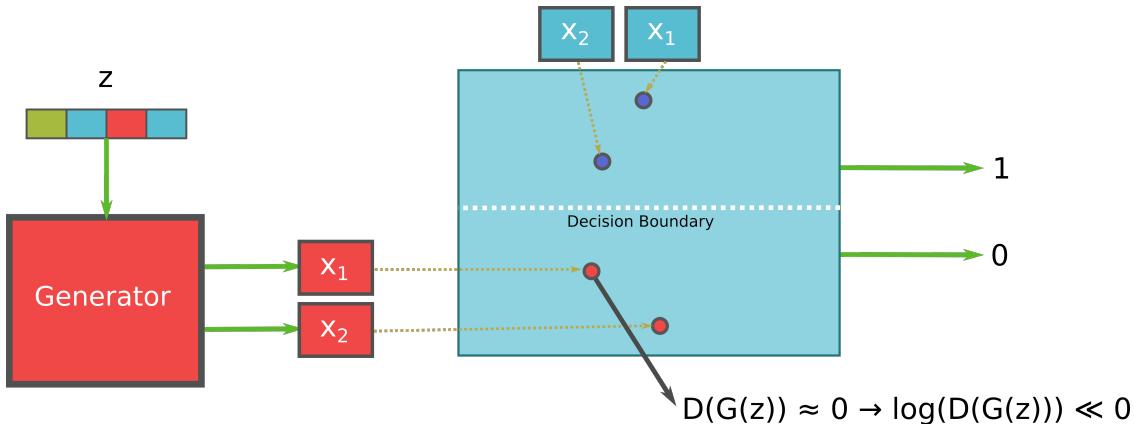
Here, both $D(\cdot) \rightarrow 1$ such that both $\log(\cdot) \rightarrow 0$, so the maximum possible loss for a perfect classification is $L_{\max} = 0$. The range of the loss is $L \in (-\infty, 0]$, so it makes sense to maximize it for the discriminator D .

It also makes sense to minimize this loss for training G . The left loss term does not depend on any

parameters from G , so here it suffices to minimize the right term, which is essentially the likelihood for D classifying the fake data samples as fake, so this will minimize the cost of correctly classifying fake data samples $G(z)$. The problem here is that the loss very quickly saturates. In the beginning, when G 's suggestions are very poor, the discriminator won't have much of a problem to discern real and fake samples, labeling the generated data as 0, saturating the loss, causing the gradient of the loss function to also become close to zero. So no substance to backpropagate is left. The trick here then is to instead optimize

$$\max_{w(G)} L_G = \max_{w(G)} \mathbb{E} [\log(D(G(z)))] \quad (201)$$

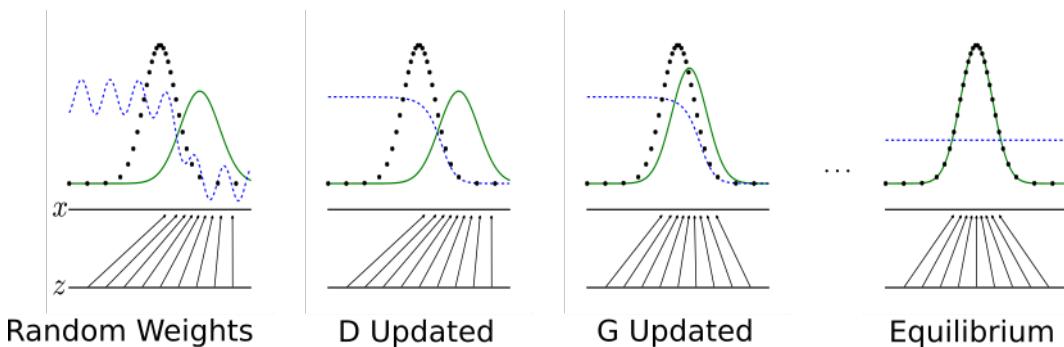
`Image("img/gen_loss.png", width="1000")`



This maximizes the error of D , such that it incorrectly classifies fake samples as 1. This is a heuristic approach to tackle the saturation problem mentioned above such that training G becomes easier. Now, as soon as G creates realistic samples $G(z)$, this loss will be close to $\log(D(G(z))) \rightarrow 0$.

The image below is taken from [Goodfellow et al., 2014](#), depicting the training process from a probabilistic perspective.

`Image("img/gan_states_from_paper.png", width="1000")`



Here, the black curve is the original data distribution, green is the model distribution represented by G , the blue curve is the output of D , the decision boundary. In the beginning on the left, D classifies points on top as real and those at the bottom as fake. Then, after updating the weights of D , the decision boundary becomes better, classifying the samples on the right correctly. Like before, z is a random noise vector and x is the real data, G maps z to $p(x)$, which is depicted by the green line. The y -axis portrays probabilities. So after updating G 's weights, the green line comes closer and closer to the dotted curve, which is the true distribution of the real data. The rightmost image depicts the situation after training. D is now unable to correctly classify generated data points.

The training algorithm (also from the original paper) goes like this:

```
Image("img/gan_algorithm_from_paper.png", width="800")
```

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k , is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.
- Sample minibatch of m examples $\{x^{(1)}, \dots, x^{(m)}\}$ from data generating distribution $p_{\text{data}}(x)$.
- Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^m \left[\log D(x^{(i)}) + \log (1 - D(G(z^{(i)}))) \right].$$

end for

- Sample minibatch of m noise samples $\{z^{(1)}, \dots, z^{(m)}\}$ from noise prior $p_g(z)$.

- Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^m \log (1 - D(G(z^{(i)}))).$$

end for

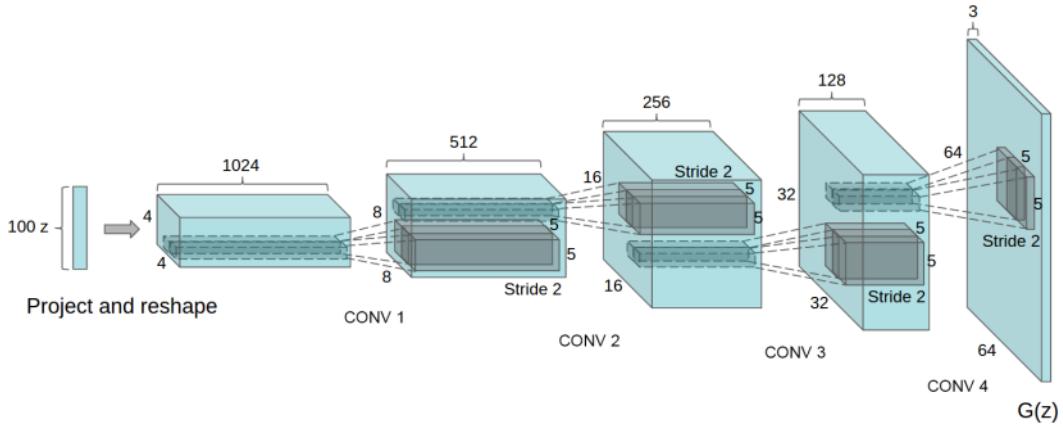
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

So in short, first D is updated, then G . Any gradient descent method can be used for performing the updates, but for D , the weights need to *ascend* instead (change the minus sign to a plus).

12.2.2 DCGANs

A popular implementation of GANs is a *deep convolutional GAN*, which uses a CNN architecture as a foundation to train on. The image below describes the generator architecture, the discriminator is pretty much just a mirror image (taken from the original paper [Radford et al.](#)):

```
Image("img/dcgan_gen_from_paper.png", width="1000")
```



The generator makes use of transpose convolutions (recall that they “undo” normal convolutions) to generate images (which are matrices) from random noise vectors z . z is sampled from random noise and has 100 components, which are projected to 1024 feature maps of size 4×4 . Then, strided convolutions or transpose convolutions are used to increase the feature map size to 8×8 and reducing the number of feature maps to 512. This process is repeated until at long last, 3 feature maps of size 64×64 finally depict an RGB image. The discriminator is a mirror image of this architecture undoing all these operations, until the output is the probability of the image input to be real.

Something extremely interesting is mentioned in the paper, that makes GANs (and all generative models) such interesting and useful concepts. z is sampled from a random distribution and creates a vector of 100 random values, which G then takes and transforms into some sensible output. When now one component of this random z is changed slightly, G produces continuous transformations of the original output $G(z)$. This is remarkable! G seems to be able to meaningfully interpolate between adjacent z . The paper itself has some fantastic examples. We will explore this a little bit in the assignment as well (and once more next week). Many big companies, who have the resources and employee power to develop, train and test gigantic generative models use this feature to create new designs for various parts of cars, motors, planes, you name it. But training such a large GAN is really difficult and demands potent hardware, especially when high resolution output is desired (cf. *this person does not exist* hardware demand). These large models are sometimes called *Big GANs*.

12.2.3 Conditional GANs

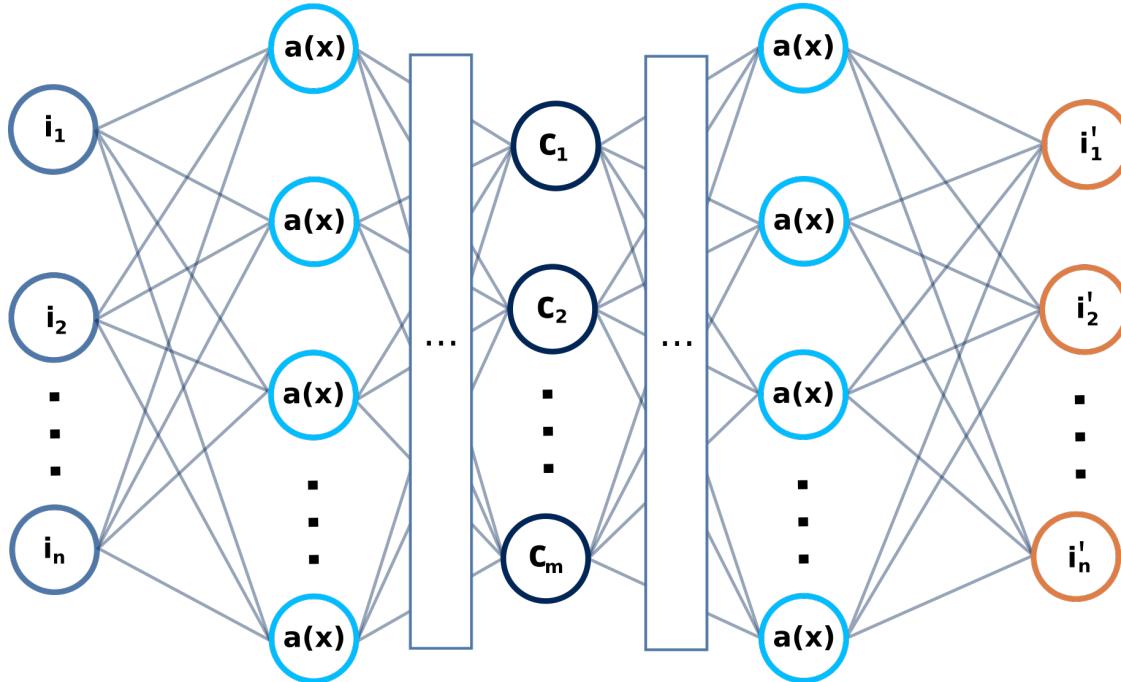
A slight modification to the GAN concept is to provide an additional input for both G and D , such that the generated outputs are *conditional* on this “excess” input parameter. We will see how this works for autoencoders in the assignment for the next lecture.

12.2.4 Data Augmentation

Since GANs produce samples that are similar to the training data, it can be used to create similar data even in cases where training data is scarce. There’s no theory behind how much data is needed to meaningfully create similar data samples with a GAN, so the generated samples need to be evaluated critically. This kind of *data augmentation* especially helps with creating classifiers.

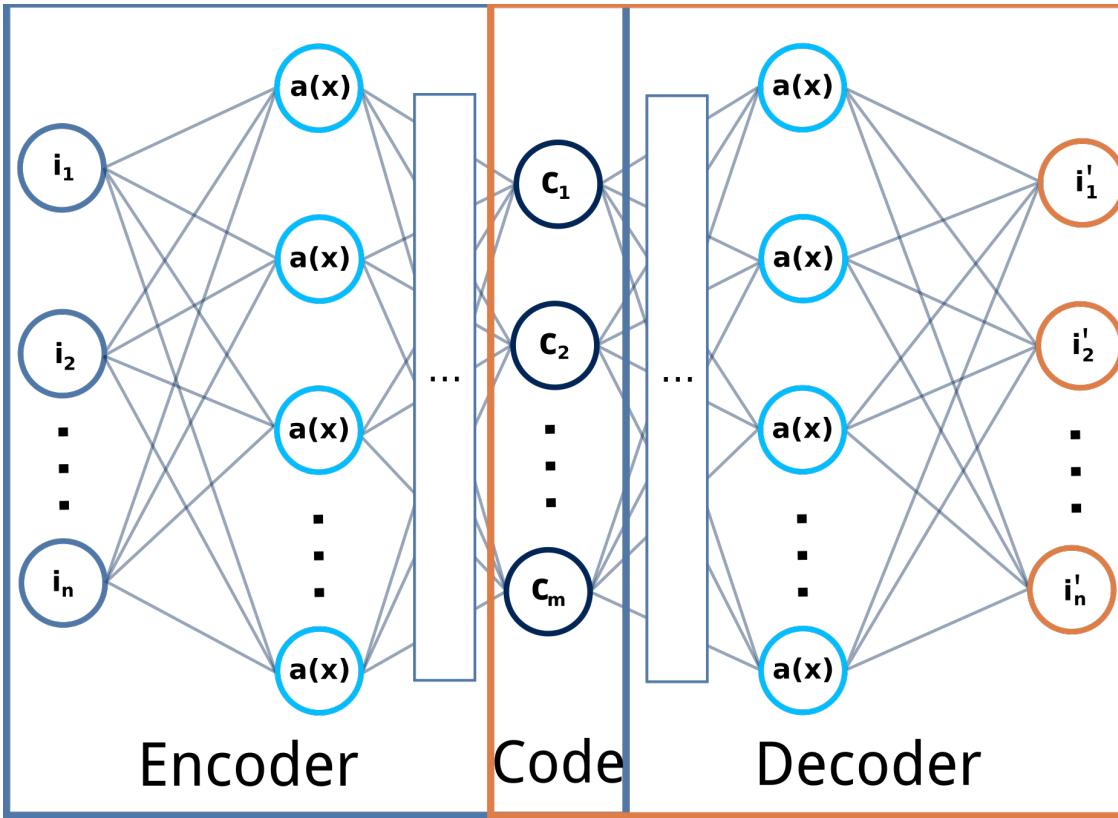
12.3 Autoencoders

```
from IPython.display import Image
Image("img/ae.png", width="800")
```



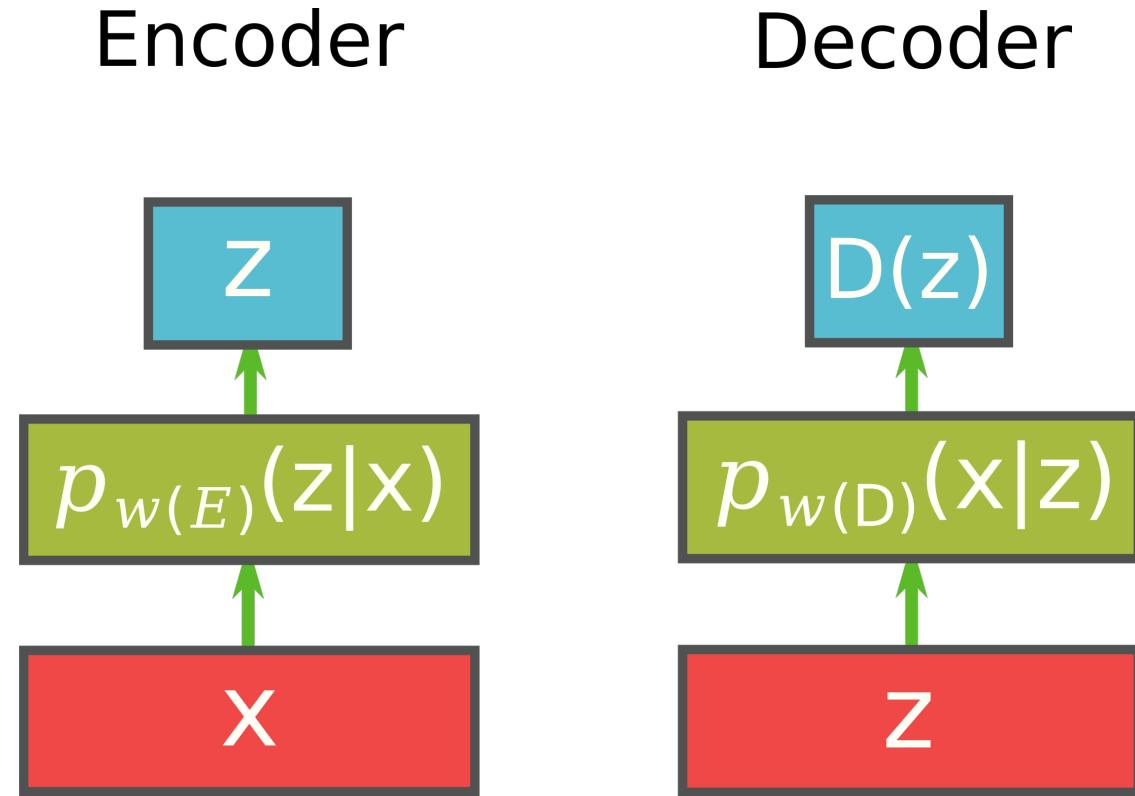
Autoencoders directly model a distribution and provide a direct way to sample from it. The idea is to make the Autoencoder reproduce the output as perfectly as possible. The ANN above is just a normal ANN as we got to know in lecture 05, with the difference that there is a *bottleneck* layer in the middle, that forces the network to learn a transformation that summarizes the dataset in a lower number of neurons than there are features in the dataset. This bottleneck layer is also called the *code*. This compressed representation of the features capture variations in the training data that is not obvious from the features themselves (does this sound familiar?). The part of the autoencoder that *encodes* the input is called the **encoder**, the mirror image network that *decodes* it again is called the **decoder**.

```
Image("img/ae_code.png", width="800")
```



The *code* contains the compressed representation, or the *latent space* z , from which the decoder reconstructs e.g. an image. This procedure can *denoise* datasets, especially images. The latent space itself is often difficult to interpret directly. Making the the latent space a representation of a probability distribution makes it much more meaningful (later).

```
Image("img/ae_enc_dec.png", width="800")
```



Autoencoders effectively learn a mapping of data to a *latent space*, which is then sampled to create outputs again. In contrast to GANs, adjacent samples from the latent space do not produce meaningfully similar outputs.

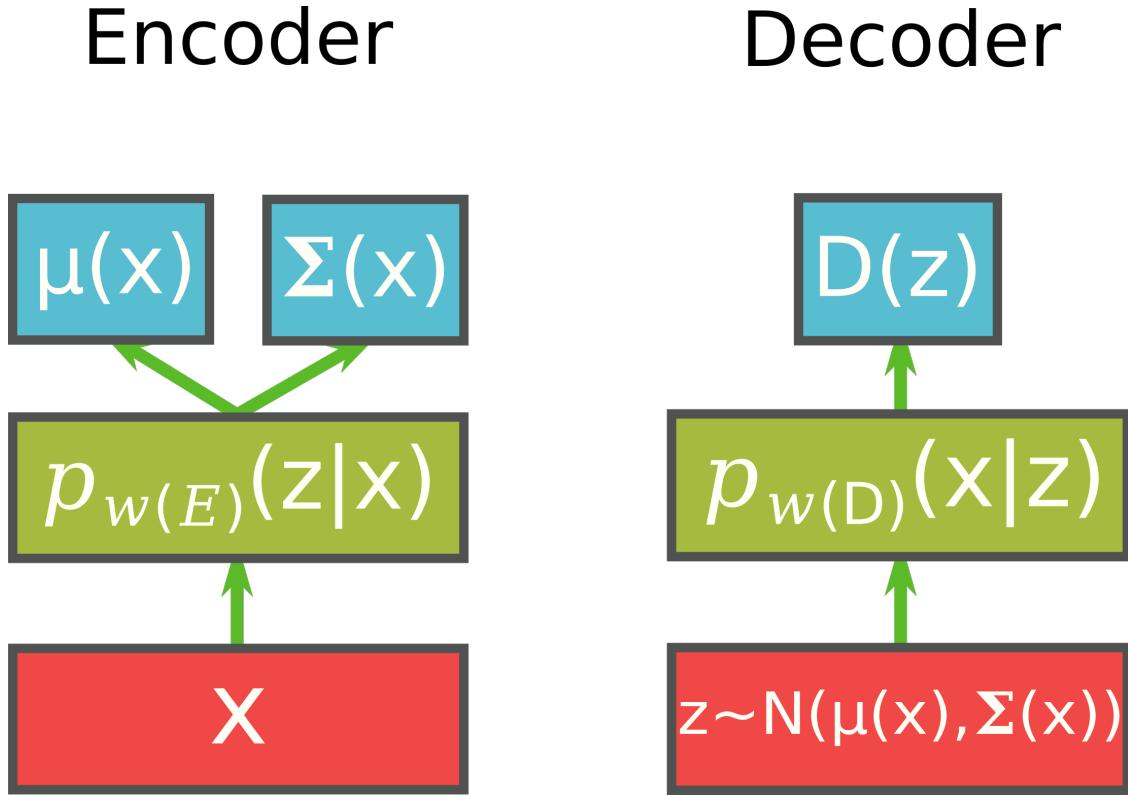
12.4 Variational Autoencoders

From a probabilistic view, z can be sampled from a probability distribution. For this, a prior distribution for z itself is needed to sample it. More interesting is a posterior distribution, which models the most likely z values given the training data. Once these are known, z can be sampled and mapped to outputs that are similar to the training data. The idea was to map training data to a latent space \mathbb{V} , which is the basically the posterior $p(z|x)$, and to have some assumption about the prior $p(z)$. Usually, the prior is modeled as a Gaussian.

Now, the output of the encoder part of the network isn't directly a sample z from $p(z|x)$ anymore, but instead two parameters, the *mean* and the *variance* of a Gaussian distribution. Since we assumed $p(z)$ to be Gaussian, given the mean and variance we can now directly sample z from the latent space. The decoder does this and maps the sample to a sensible output.

Variational Autoencoders (VAEs) still consist of the same parts as standard autoencoders, encoder, code, and decoder, plus a regularized loss function. The encoder takes training data as input and provides the parameters of the latent space probability distribution.

```
from IPython.display import Image
Image("img/vae_enc_dec.png", width="800")
```



The network could also be realized as a CNN for example, or other kinds of networks. z is then sampled from the distribution described by the encoder and given as input to the decoder, which then outputs a reconstruction of the input x .

12.4.1 VAE Loss

The loss function used for variational autoencoders is

$$L(x; w^{(E)}, w^{(D)}) = -D_{\text{KL}}(p_{w(E)}(z|x) || p_{w(D)}(z)) + \mathbb{E}_{z \sim p_{w(E)}(z|x)} [\log p_{w(D)}(x|z)] \quad (202)$$

where

$$D_{\text{KL}}(p_{w(E)}(z|x) || p_{w(D)}(z)) = \mathbb{E}_{z \sim p_{w(E)}(z|x)} [\log(p_{w(E)}(z|x)) - \log(p_{w(D)}(z))] \quad (203)$$

is the **Kullback–Leibler divergence (relative entropy)**, which measures how much one probability distribution is different from a reference distribution and acts as a *regularizer* here.

The difference here is measured between the output distribution $p_{w(E)}(z|x)$ of the encoder and the prior distribution $p_{w(D)}(z)$ of the z , both of which are usually modeled as a Gaussian. The second term is the *reconstruction loss*.

The mean $\mu(x)$ and covariance matrix $\Sigma(x)$ (which is constrained to be diagonal, how many values are regressed by the encoder for a data set with M features?) from the image further

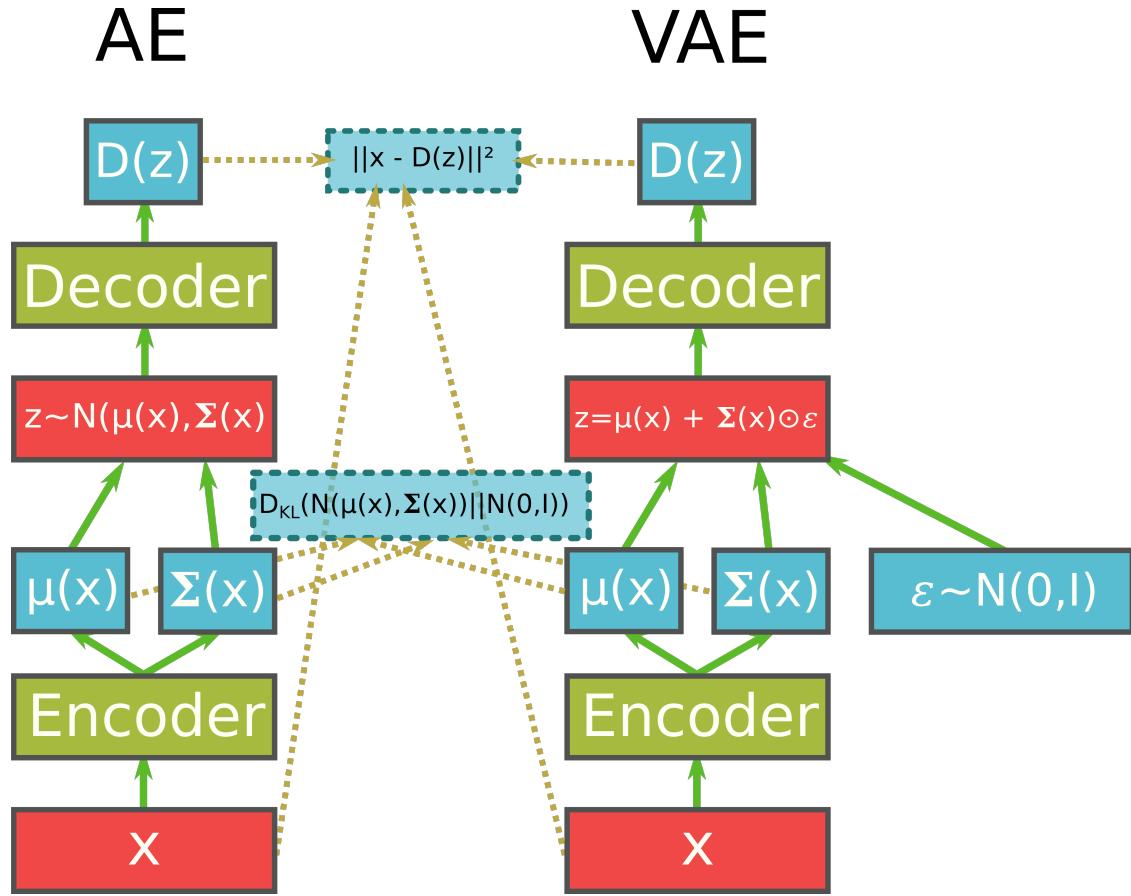
above parameterize the Gaussian $p_{w^{(E)}}(z|x)$. The prior $p_{w^{(D)}}(z)$ as mentioned is also modeled as a Gaussian, but with zero mean and unit variance. The decoder output $p_{w^{(D)}}(x'|z)$ also follows a Gaussian distribution. Taking the log of it essentially makes the reconstruction loss a mean square error loss. In cases where the data is categorical, say, in a black-and-white image, one can also use binary crossentropy. The decoder output would lie in a range $[0, 1]$ and denote the probability of a pixel to have a value of 1, with a Gaussian probability distribution.

Like GANs, variational autoencoders produce meaningfully similar outputs for adjacent z values. An autoencoder trained on the MNIST dataset for example with a latent space of size 2 would produce fake handwritten numbers that look like the real dataset, where one components of the latent space determines the number to produce and the other the “look” of the number, the stroke, rotation, or something similar.

12.5 Reparametrization

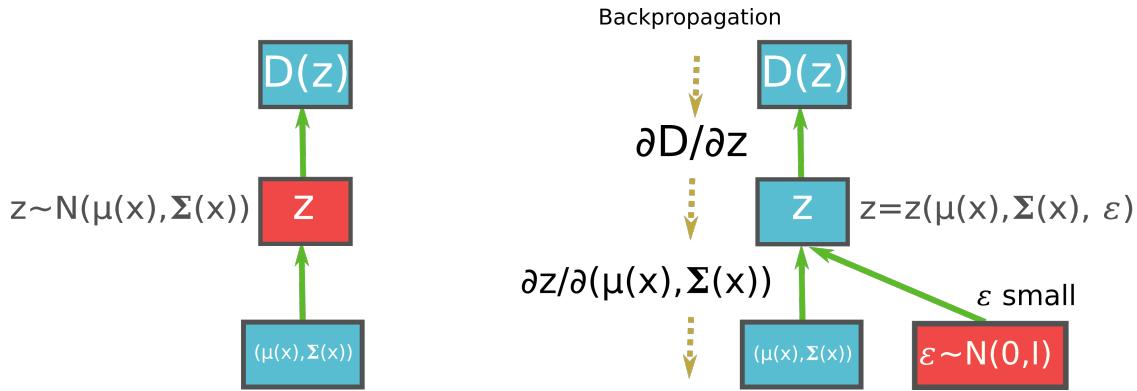
The VAE loss needs to be optimized on a sample-by-sample basis. The process could be the following: the encoder provides $\mu(x)$ and $\Sigma(x)$ for an input x , then lots of z values can be sampled using these parameters in a Gaussian distribution. These values are passed through the decoder to perform a reconstruction, and the expectation in the loss function will be performed over all z values corresponding to input sampled x . This is difficult, since not all the sampled z correspond to the input x . To make this easier, the encoder is used for a mini-batch of (or the full set of) the training data, so that the expectation value can be taken over that (mini-batch of the) *training data* (this is the reason for why an encoder is needed at all in this setup, as some of you might have wondered why we need it if we only sample from Gaussians anyway). So one z is obtained for each input sample x by using the encoder, such that the z corresponds directly to the x , then it's passed through the Decoder to get the reconstruction loss. Then backpropagation is used to redistribute the error back to the weights and adjust them. There's one problem with this approach. Sampling from a distribution is not a differentiable process, so we cannot get a gradient here. The workaround to this problem is the **reparametrization trick** (original paper mentioning this is [here](#)).

```
from IPython.display import Image
Image("img/vae_sketch.png", width="800")
```



The image on the left describes what we discussed in the last lesson, the image on the right describes the reparametrization trick. If a sample is drawn from a normal distribution with zero mean and unit variance it can easily be converted into a sample from a Gaussian distribution with different mean and variance (do you remember where we already did that?). Everything else stays the same. The encoder produces a mean and a variance, but instead of providing this sample directly, a sample ε from a standard normal distribution is drawn, then multiplied by Σ , and added to μ , such that $z = \mu + \Sigma \odot \varepsilon$. This is then given as the input to the decoder. Now backpropagation can be performed and the error backpropagated through the whole network to update all the weights, since the process of sampling from the normal distribution does not depend on either the encoder nor the decoder weights. The following schematic portraits that process in a more compact way:

```
Image("img/reparametrization.png", width="800")
```



Constraining the autoencoder to producing stochastic output makes the latent space modeling a lot smoother than was the case for standard autoencoders. Now, when input samples close to each other are drawn, the output will also vary meaningfully and smoothly. So the *structure* of the training data (say, images) is captured in the *distribution* of z . This is in contrast to standard autoencoders, where the latent space consisted of an arbitrary hidden space representation, where adjacent vectors did not provide smooth transitions between outputs. Here, the latent space produces a very structured z , all thanks to treating the output of the encoder as stochastic rather than deterministic, as was the case for standard autoencoders.

13 Machine Learning Hard- and Software

13.1 Machine Learning Hardware

Current machine learning models for huge tasks easily have some billions of weights. E.g., openAI's [GPT-3](#) has 175 billion parameters and is the largest language model ever trained. Training such a model on a cheap GPU would take around 355 years and \$4.6 million. Machine learning advances so fast and hardware supporting efficient model training grows so quickly that Google engineers managed to rediscover the infamous Higgs boson from the original LHC data in a mere [3 minutes](#)(!) live at a conference with the Google Cloud. The dataset had a size of 400 Petabytes.

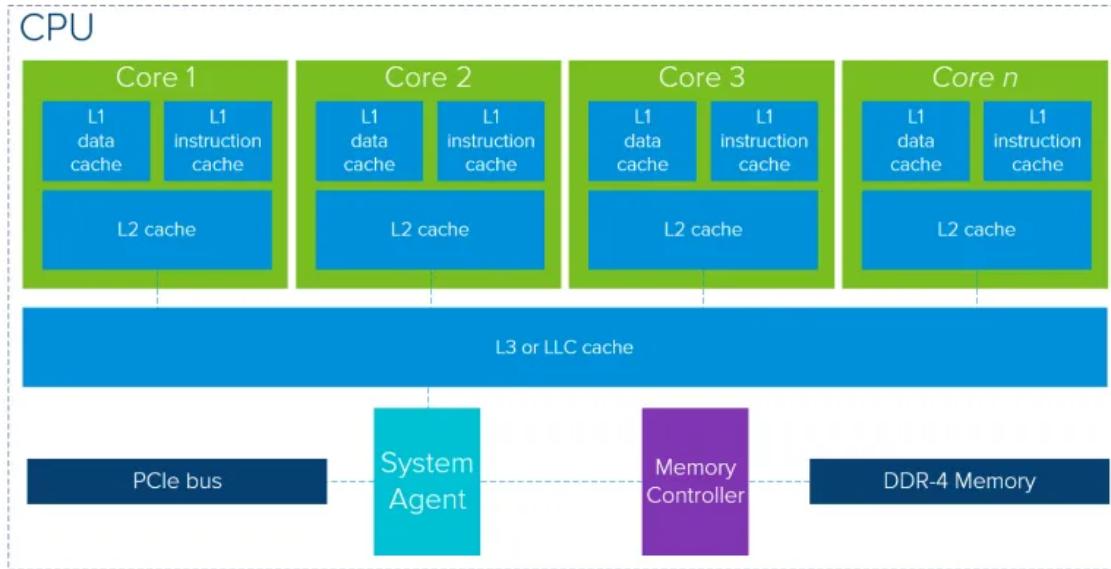
It makes sense to take a look at what happens computationally during machine learning and optimize the bottlenecks. The first step that helps is to *parallelize* the workload. Parallelization is a topic in and of itself, so we'll skip it here for the most part. There are various HPC systems available and all frameworks natively support them. The following points are the key bottlenecks for large-scale machine learning:

- Input/output (data volumes grow quickly)
- Data movement for both accelerators and parallelization
- Communication between compute nodes

These are largely HPC problems and out of scope for us. We'll concentrate on accelerators here.

CPUs are optimized for low latency, meaning quick access to memory (the CPU and GPU architecture sketches are taken from [this blog](#)):

```
from IPython.display import Image
Image("img/cpu.png")
```

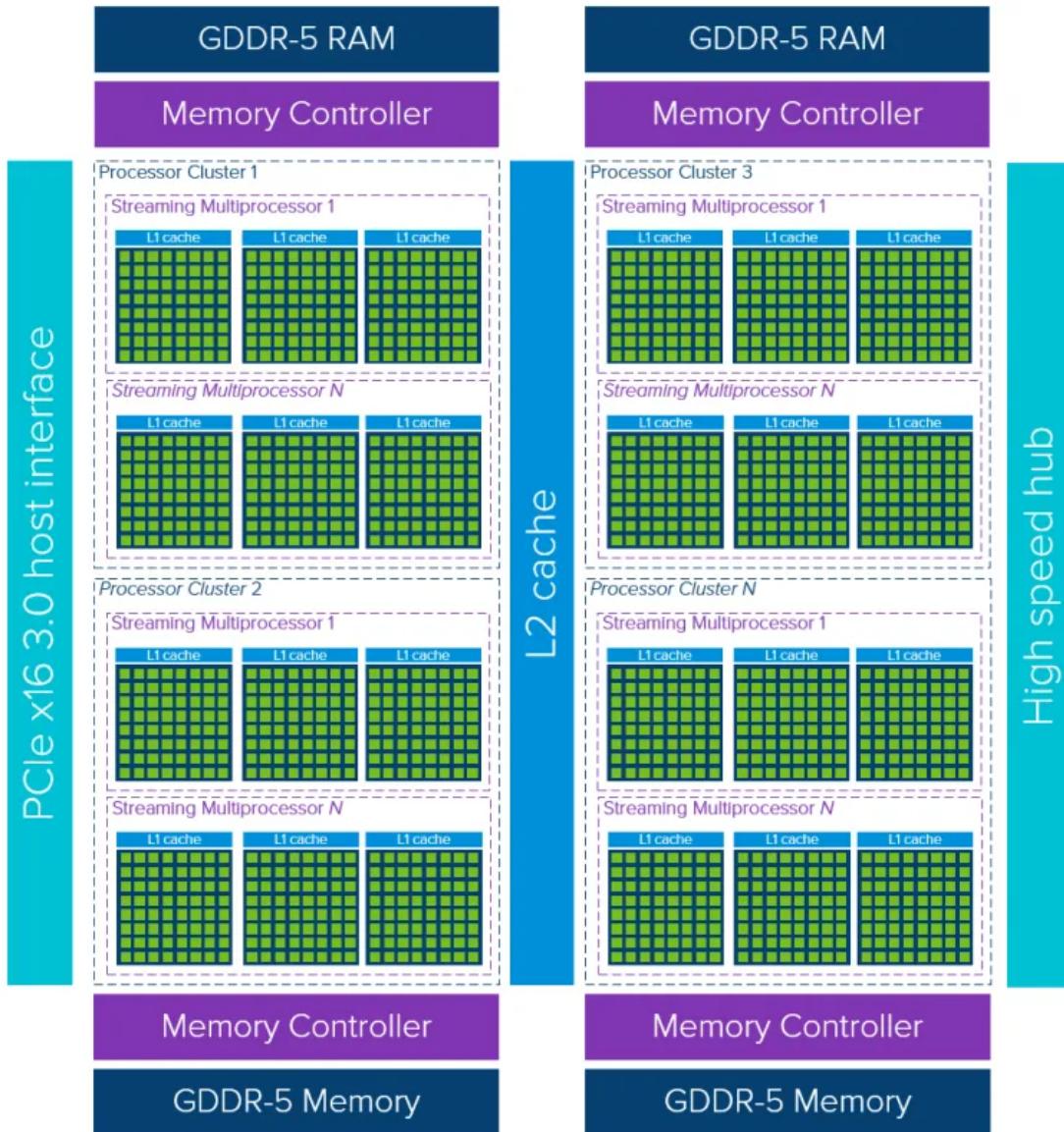


As we've seen especially in the introductory chapters, machine learning uses lots of matrix opera-

tions, especially matrix multiplication (*MMU*) and multiply and accumulate operations (*MACs*). E.g. a weight vector times an input vector, then sum everything up. Or in a CNN multiply filter and image region, then add the results. Before elementwise operations and statistical normalization, this is the most costly operation in both training and inference and several hardware architectures can be exploited to speed this step up.

13.1.1 GPUs

`Image("img/gpu.png")`



Developed for fast rendering mainly for games, GPUs offer stream processors that perform floating point logic operations highly parallel. The speed of a device is usually measured in *FLOPS - floating*

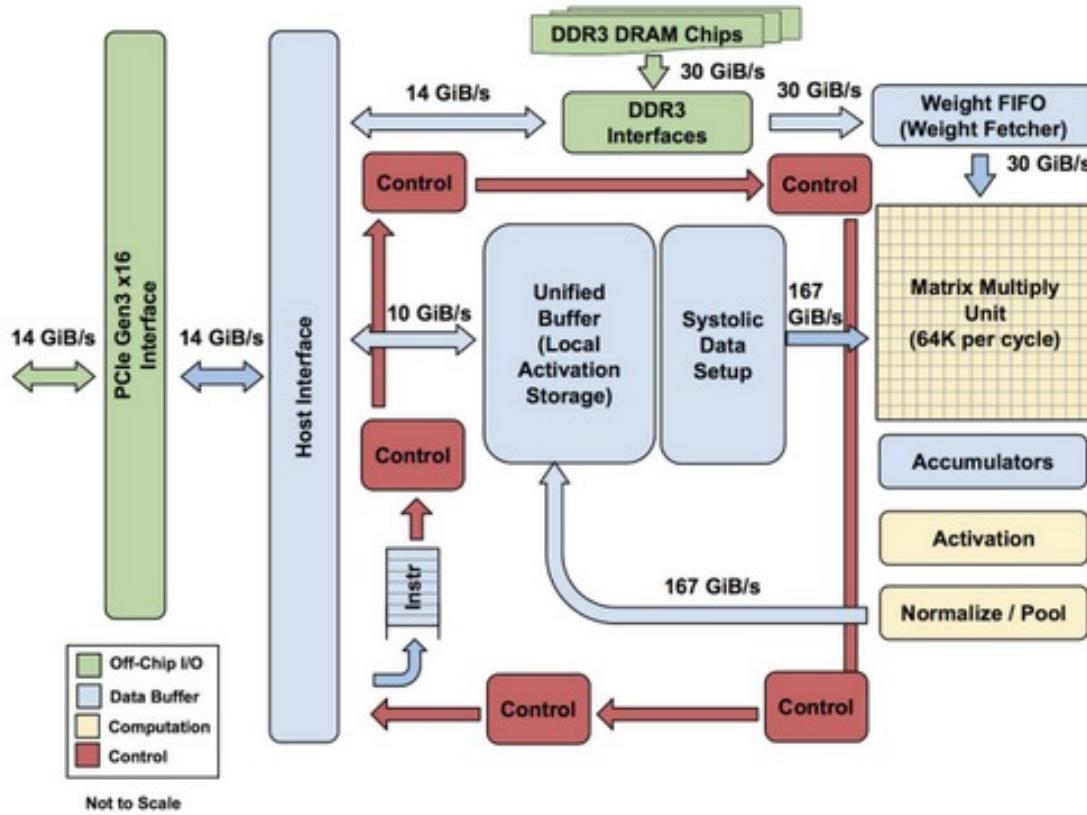
point operations. GPUs excel at this, doing lots of floating point operations in parallel. While the CPU has a cache of a few Megabytes, Modern GPUs come with several Gigabytes of dedicated RAM. Some people compare CPUs and GPUs like cars to busses. The latter is slower, but fits in more passengers than the car. In a bit more professional terms, CPUs are optimized for low latency, while GPUs are optimized for high bandwidth. So it's not necessarily the high number of stream processors on a GPU that helps training large models faster, but actually the large memory that can be accessed “broadly”. Hence, the more data you have and the larger it is, the higher is the advantage of using a GPU. TensorFlow has a [guide](#) on how to optimize your model’s performance on a GPU, and so does [NVIDIA](#). Especially the “Getting Started” section offers some quick advice on how to construct optimizable models.

13.1.2 ASICS

Most modern GPUs additionally offer **tensor cores**, which perform matrix multiply and accumulate operations efficiently. Normally, multiplying two 4×4 matrices necessitates 64 multiplications and 48 additions. Depending on the implementation of a tensor core, it can do this 8 to 32 times faster, depending on the data type (integer, float16, float32, float64, ...) and the tensor core version.

This is an example of an *Application-Specific Integrated Circuit*, or *ASIC* in short. ASICS are devices that are constructed such that they only perform a single type of computation, but do this extraordinarily well. There’s a tradeoff regarding the general applicability of a processor and its performance. The more general types of computations a processor can perform, the worse its speed. ASICS are on the other end of the extremes.

```
Image("img/tpu.jpg")
```



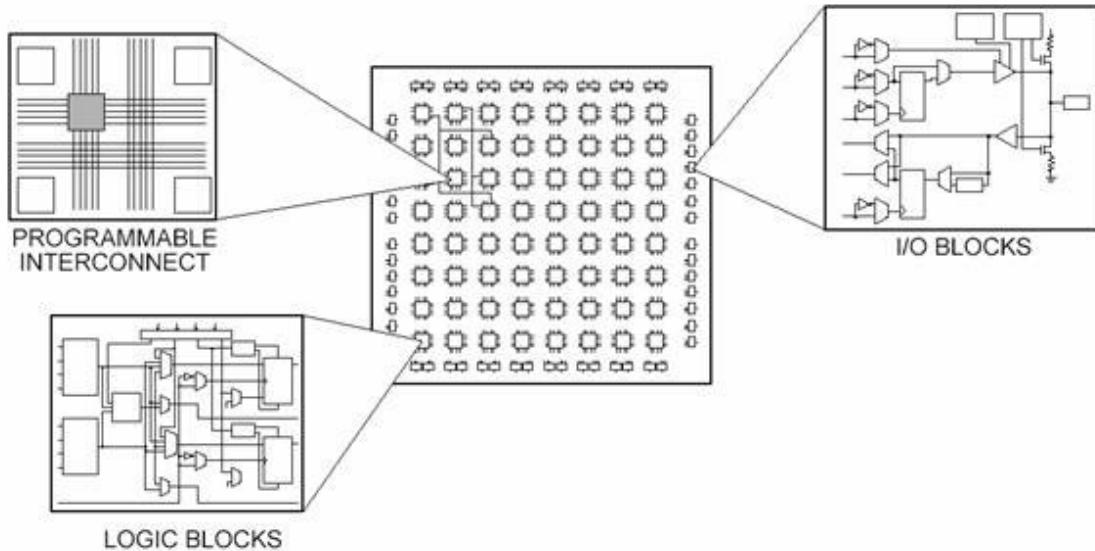
Several companies have offered ASICs for deep learning. Probably the best-known example are Google's **Tensor Processing Units (TPUs)** (image above taken from [TheNextPlatform](#)), which do the same thing as the tensor cores mentioned above plus in silico activation functions. Google also offers *Google Coral*, which is a USB stick containing a small TPU mainly for inference on “edge devices”. You could use this with a Raspberry Pi for example. Google offers some material to learn about TPUs [here](#).

NVIDIA offers a similar platform with **Jetson**. Intel offers the **Neural Compute Stick** and there's also a dev board from BeagleBone called **AI board**.

Now we talked about both extremes. Is there something in the middle?

13.1.3 FPGAs

Image("img/fpga.jpg")



Field-programmable Gate Arrays (FPGAs) (image above taken from [National Instruments](#)) offer a tradeoff between general purpose computing and ASICs. There are various implementations, but the idea is always the same. The chip itself doesn't do anything, until using some hardware description language a configuration file is created that tells the chip how to connect the different physical units. They are usually reprogrammable, so while they do not offer the full performance ASICs can, they're more general purpose since the circuitry can be redefined as desired. In fact, they're often used for prototyping ASICs before they go into production. Microsoft's Azure cloud offers access to FPGAs, but it's not that straightforward to program them efficiently.

An emerging and interesting field is **neuromorphic computing**, where the idea is to emulate processes from the human brain on a chip, such as neurons firing independently. A lot of work needs to be done before

13.1.4 Summary

Architecture	TFLOPS	Max Power Consumption [W]
CPU (AMD Ryzen 9 5950X)	0.967	142
GPU (NVIDIA Titan RTX)	32 (FP16), 16 (FP32), 0.5 (FP64)	280
TPUv3	420	450
Coral (USB)	1	2
Jetson (Xavier NX)	21	15
Compute Stick 2	1	?
FPGA	$\mathcal{O}(1 - 10)$	varies

13.2 How to Identify ML Opportunities

With all the examples we discussed you may have wondered if there is a structured approach to identifying situations in which machine learning can help. While there is no general unequivocal

answer, there are some guidelines on where and how to look.

In most settings, machine learning excels where humans have a hard time and vice versa fails where humans perform well. Think for example about image recognition. While for us it's easy to identify cats, an ANN needs quite some time of training to work approximately correctly. For detection of medical landmarks, say, fat blobs in organ sections of patients, the situation is not so clear. A layperson would likely fail at correctly differentiating fat from, say, veins or other matter. Human experts are much better, but still are far from perfect and results differ a lot from expert to expert (there are some studies on this, which are best kept obscure). Here, an unbiased approach via machine learning can help, especially in conjunction with experts that check the results.

In general, ANNs are *function approximators*. If there is a function in a simulation or other calculation that is difficult to evaluate, or difficult to even formulate (think constitutive laws for example), this is a good candidate for modelling with ANNs. Let's take a look at material modelling for example:

$$\Psi(\mathbf{C}) = \Psi(I_C, II_C, III_C) \quad (204)$$

$$I_C = \text{tr}(\mathbf{C}), II_C = \frac{1}{2}(\text{tr}(\mathbf{C})^2 - \text{tr}(\mathbf{C}^2)), III_C = \det(\mathbf{C}) \quad (205)$$

$$\sigma_{ij} = \frac{\partial \Psi}{\partial \varepsilon_{ij}} \quad (206)$$

$$C_{ijkl} = \frac{\partial \Psi}{\partial \varepsilon_{ij} \partial \varepsilon_{kl}}, \sigma_{ij} = C_{ijkl} \varepsilon_{kl} \quad (207)$$

$$\mathbf{Ku} = \mathbf{f} \quad (208)$$

The first equation is the **Helmholtz free energy**, which isn't always given and sometimes not even possible to formulate. If it exists, it offers a systematic way to derive the constitutive equations of the system (which describe the material's response to loads and deformations). Hence, it's a candidate to be modelled by an ANN, e.g. by feeding experimental data for the training such that the ANN learns an approximation of the “real”, underlying Helmholtz free energy of a material.

The Helmholtz free energy is usually given in terms of the **tensor invariants** of \mathbf{C} , given in the second equation, because they do not change under a covariant transformation of the coordinate system. They are the coefficients of the characteristic polynomial, which describes the eigenvalues of a matrix. They are invariant under certain transformations and hence, the invariants too. Sometimes it may make sense to learn these tensor invariants from data.

The third equation show how to derive the **stresses** in a material when given a Helmholtz free energy and strains (or vice versa). Instead of modeling Ψ , one can directly model this relation.

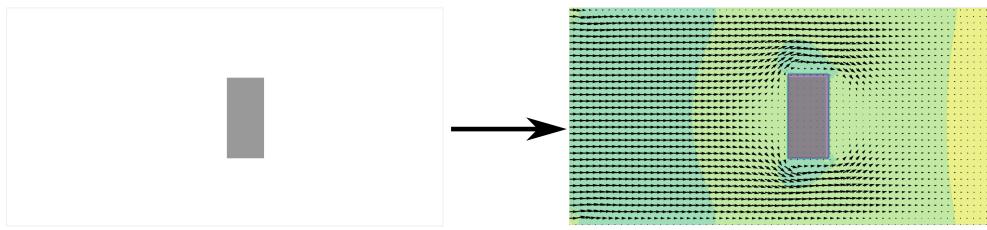
The fourth equation shows the **material stiffness tensor**, also known under different names. This quantity is highly symmetric and instead of approximating all the tensor components, only a few are important depending on the symmetry of the material. For highly complex materials, it may make sense to model this with an ANN, because symmetries may only be approximate.

The last equation is a simple linear equation system used in **finite element analysis**, where given the stiffness matrix of a system and a load, the resulting displacements are computed. Here, one could approximate the elemental stiffness matrices and elemental loads from experimental data instead of computing them individually. Both to save time and to model the system response more accurately than from purely theoretical approaches. In an extreme case, one could also model the displacements directly from the loads by creating a *surrogate model*, which we'll do in the assignment today for a simple thermomechanical system.

There are many more examples. In plasticity, you could model the return mapping algorithm instead of relying on theoretical considerations, or sometimes you're only interested in macroscopic behavior of a system. You could predict the stress field around a crack instead of calculating it, or determine how it propagates without a simulation.

Sometimes it's possible to predict the behavior of complex systems from knowledge about simple subsystems.

```
from IPython.display import Image
Image("img/cfd_cnn.png")
```



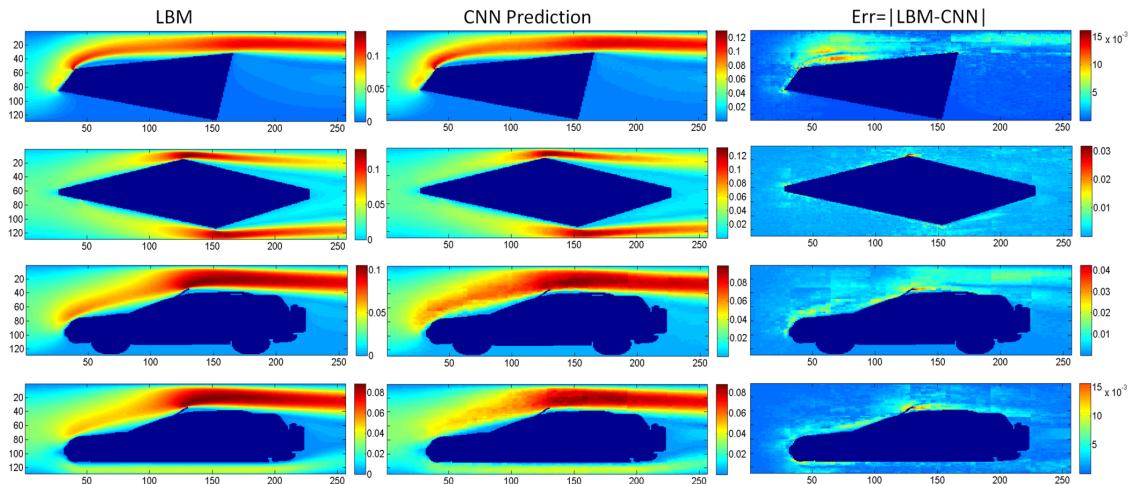
The CNN would then learn how the flow field around various shapes should look like and hopefully generalize to more complex shapes. There is just a slight modification that is needed here; If the image as depicted on the left side above is input into the network, that is, as 1s where the object is located and as 0s where no object is placed, the network does not get any information about the difference in importance of the pixels far away from the object and closely around it, since both would be represented by a pixel value of 0. It does matter a lot how far away from the object you actually are. The solution is to use a *signed distance function*, that assigns the region inside the shape a negative value, and a value to pixels outside of the shape that denotes the *closest distance* of that pixel to the shape. The idea was used for example in [this conference paper](#) and has been widely adopted since (for example also for [this hybrid approach](#) to predict air flow around airfoil shapes or in [this current paper](#)). The loss function also has to be modified slightly, since no flow should be present inside of the body. Mean square loss can be used, but multiplied with basically a Heaviside function that determines if a pixel from the prediction is compared to a pixel from the ground truth lies inside of the shape, or outside of it. If it lies inside, the loss function is multiplied by zero, otherwise by one, since we only want to compare the flow field outside of the shape.

As an architecture for the CNN, something with an encoder-decoder-like structure would be suitable to scale down the information contained in an image, where the decoder would scale it up again

and add the physics information. A slight modification makes sense; since the flow field in two dimensions has two components, the velocity in x -direction u_x and the velocity in y -direction u_y , and since both are based on the same geometry, we could use a single encoder to compress the essential information about the geometry, and two decoders which each decompress that information and add their knowledge about the flow fields in x - and y -direction, separately (to be exact, since backpropagation carries information back through the whole network, including the encoder, some information about the physics leaks to it as well).

The image below was taken from the conference paper mentioned above, where the flow fields for various simple shapes at various angles were calculated using the Boltzmann lattice method, the modified U-Net architecture trained on these examples, and predictions made for various more complex shapes:

Image("img/cfd_cnn_from_paper.png")



The flow at some points, especially close to sharp edges, is still erroneous. The overall quality of the results and the speedup (they mention a speedup of ~ 200 in the paper) is still tremendous and is very likely to be improved in the near future. There are other ideas for how to incorporate machine learning into CFD with various degrees of success. A recent development and hopeful candidate is to use *physics-informed neural networks (PINNs)*, which we'll discuss in the last lecture of this course.

13.3 Machine Learning Platforms

Various services are available for (mostly) quick and easy access to machine learning hardware and software. A popular and free service is [Google Colab](#). It offers Jupyter notebooks just like you're used to by now, with the added bonus of GPU and even TPU access for free. This is perfect for small and fast models. For larger models, it won't work for several reasons. The runtimes are restricted in memory and CPU cycles, but the worst detriment is that this is running on the paid Google Cloud hardware and as soon as a paying customer needs the resources, your server will be culled. It's still great for quick prototyping and perhaps even your final project, unless training takes more than a day. You have to be patient if it's a long computation. Google Cloud is a paid

service with a free trial, where you can run your own server instances and do pretty much whatever with them.

A service that does also exactly that but for free if you're at a university in Baden-Württemberg (that includes students!), you may like to use the [bwCluster](#). The setup process is a bit tedious and you need some HPC knowledge, but you are barely restricted in what you can do, if you have the patience to wait in sometimes particularly large queues. On HPC systems, queuing systems are used that make sure everyone gets a fair amount of computation time on the expensive hardware. On these HPC systems, you're first connecting to a gateway server where you can setup everything you need, then send your job to the queue and wait for it to execute on the actual cluster. An alternative may be [bwCloud](#) where you can setup your own instances, but are restricted in the hardware they offer. This lecture server resides on a bwCloud instance.

Similar services to that of Google are offered by [Microsoft Azure](#) and [Amazon AWS](#). While both lack TPUs, each has their individual advantage in other things. Azure offers FPGAs and will soon (only one more year guys..) incorporate their quantum capabilities into their cloud offerings. Amazon already offers quantum simulators and access to real hardware with Amazon Ket, so if you're interested in experimenting with either quantum computing or quantum machine learning, this might be the best platform for you. We'll briefly discuss quantum computing and quantum machine learning in the last lesson today.

If you need example data for testing machine learning algorithms, or for inspiration for the final projects or just for learning more, there are quite a few sources with datasets out there like [openML](#), [kaggle](#), and medicine. If you prefer to create your own data from simulations, [NanoHub](#) is a service that hosts and executes simulations for inumerous situations that could provide you with data.

The landscape of platforms for machine learning is growing quickly and a search engine will likely give you more results each year.

13.4 Quantum Machine Learning

to the hype machine learning has caused is *quantum computing*. Deriving its name from the fearsome quantum mechanics that gave birth to the ideas, it's actually not that wild as long as the physics is left out of it. Of course this short lesson won't be able to give a comprehensive overview of all things quantum, but the basic concept is actually not that difficult. Ideas from quantum computing can be used to speed up parts of machine learning tasks, or the latter can be formulated in a completely new way that leverages the advantages quantum computers may at some point provide. First tests on machines available through cloud interfaces yield promising results, but the way to lo large, scalable, and error-corrected devices still seems challenging and may turn out to be insurmountable.

13.4.1 Quantum Computing

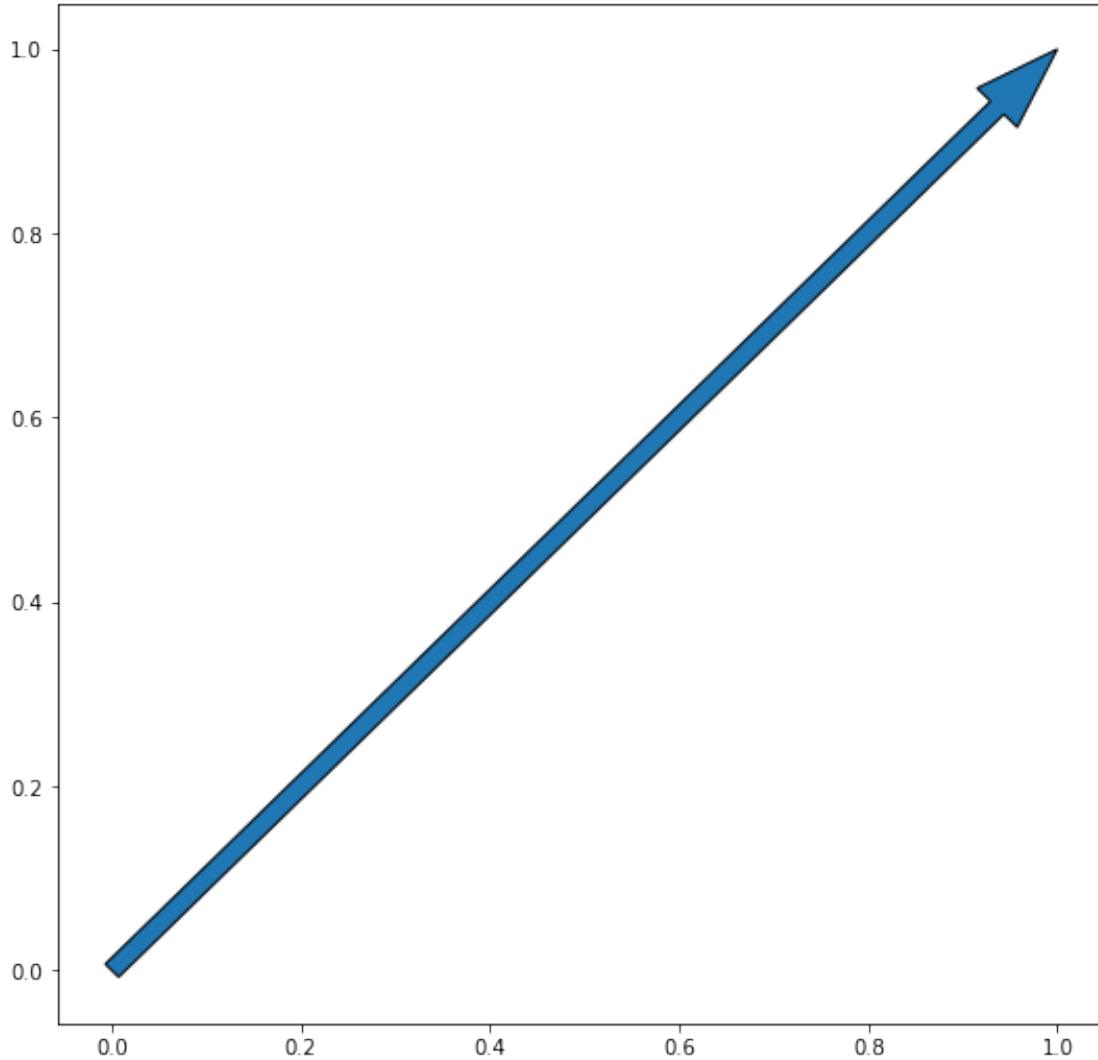
In classical computing, the basic unit of computation is the *bit* b , which can display either a 0 or a 1. Many introductory texts or presentations to quantum computing introduce the idea that *qubits*, the fundamental units of computation for quantum computers, can somehow be in a state of "both 0 and 1 at the same time". While not completely wrong (in the humble opinion of the author, it is), it fails to capture a very simple fact about quantum states and obscures the actual mechanisms.

See for example the following graph of a vector in 2D:

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(figsize=(9,9))
plt.arrow(0, 0, 1, 1, width=0.02, length_includes_head=True)
```

<matplotlib.patches.FancyArrow at 0x7f1e73e24c50>



While this vector v certainly is neither the basis vector in x -direction \hat{e}_x , nor in y -direction \hat{e}_y , it has both a certain “ x -ness” and “ y -ness”, reflected in its coordinates

$$v = \alpha \hat{e}_x + \beta \hat{e}_y, \quad \alpha, \beta \in \mathbb{R}$$

It's a *superposition* of both basis vectors, but it is not both at the same time.

In the same way, a *quantum state* is a superposition of the basis vectors describing the computational space and looks like this:

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C}$$

where $|0\rangle$ and $|1\rangle$ are the basis vectors of a complex 2D *Hilbert space*, which is a fancy term for a vector space over the complex numbers. It shouldn't worry us too much here.

So the qubit is neither 0 nor 1, but a *superposition* of the basis vectors representing 0 and 1 respectively. One peculiar thing about quantum computing is that when you add several qubits, instead of linearly the computation space grows *exponentially*, because the combined basis vectors are calculated using the tensor product instead of the cartesian product with classical systems. This exponentially growing computation space can be exploited cleverly for certain tasks.

Computations are rotations and reflections in this complex Hilbert space. It could look something like this:

```
import numpy as np
from qutip import Bloch

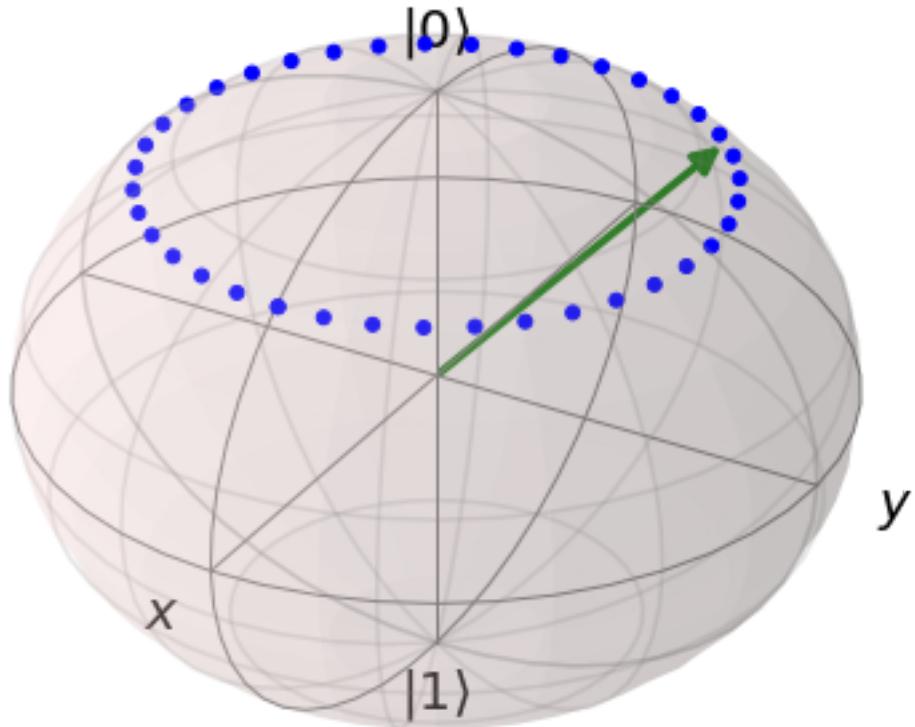
b = Bloch()

= 3.0*np.pi/4
state = 0.7*np.array([np.cos(), np.sin(), 1])

p = 40
xp = 0.7*np.array([np.cos(th) for th in np.linspace(0, 2*np.pi, p)])
yp = 0.7*np.array([np.sin(th) for th in np.linspace(0, 2*np.pi, p)])
zp = 0.7*np.ones(p)

b.add_points([xp, yp, zp])
b.add_vectors(state)

b.show()
```



The sphere is called the *Bloch sphere* and visualizes all the valid states a qubit can be in. The state as given above is actually described by four degrees of freedom, but certain constraints leave only two, such that all the states lie on this sphere.

Quantum Computers come in mainly two varieties, namely *gate-based quantum computers* and *adiabatic quantum computers*. Here, we'll concentrate on the former. Since computations are rotations and reflections, you can think of quantum computers as “linear algebra machines” that perform certain operations efficiently. These operations are the *gates*, and they operate on a number of qubits in a *quantum register*. This is usually depicted as a *quantum circuit*:

```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit

# separately create quantum and classical gates
# for clearer visualization
qreg = QuantumRegister(3)
creg0 = ClassicalRegister(1)
creg1 = ClassicalRegister(1)
```

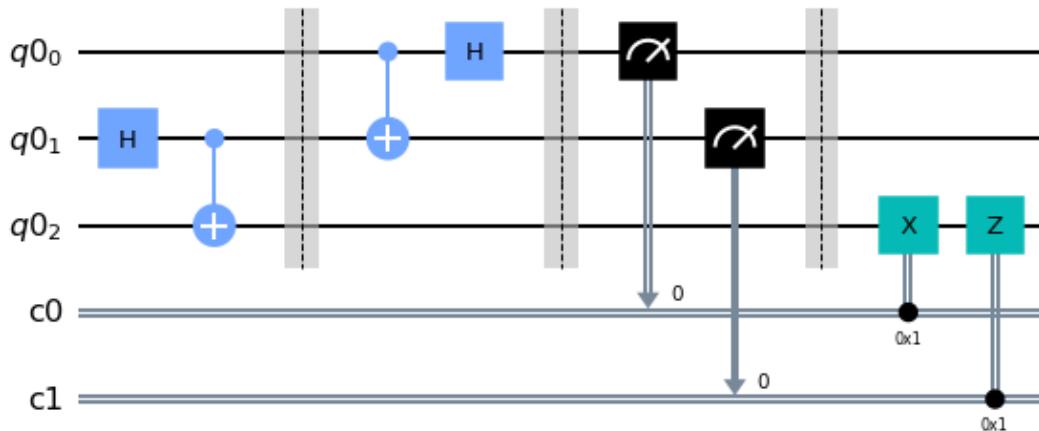
```
# put them together
circuit = QuantumCircuit(qreg, creg0, creg1)

# entangle |A> and |B>
circuit.h(1)
circuit.cnot(1, 2)
circuit.barrier()

# Bell measurement
circuit.cnot(0, 1)
circuit.h(0)
circuit.barrier()
circuit.measure(0, 0)
circuit.measure(1, 1)
circuit.barrier()

# Bob applies appropriate gates, depending on classical bits
circuit.x(2).c_if(creg0, 1)
circuit.z(2).c_if(creg1, 1)

circuit.draw('mpl')
```



It kind of reads like a musical score. The details aren't too important right now. What such a circuit does effectively is to prepare a *probability distribution*, and measuring the register is equivalent to *sampling* from this distribution.

There is a lecture called *Quantum Computing for Engineers* in winter semesters if the concept interests you. The course style is similar to this one.

13.4.2 Quantum-enhanced Machine Learning

Certain tasks run extremely fast on quantum computers. Depending on the exploitation trick, the speedup is often either quadratic or exponential. Two examples for such algorithms that can be used in machine learning are *Grover search* and the *HHL algorithm*. The former searches an element in an unstructured database with $\mathcal{O}(\sqrt{N})$ and is proven to be optimal, compared to the classical optimum with $\mathcal{O}(N)$. This can be used, e.g., for the k -nearest neighbors algorithm, where, well, the nearest neighbors of a data point have to be found.

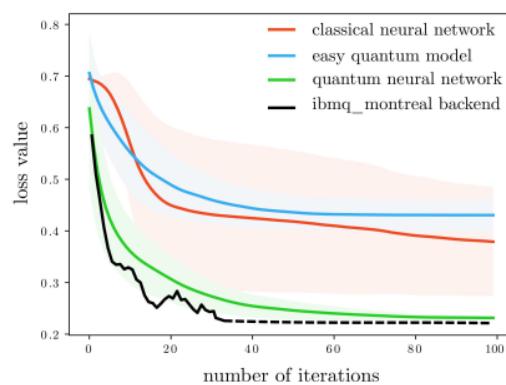
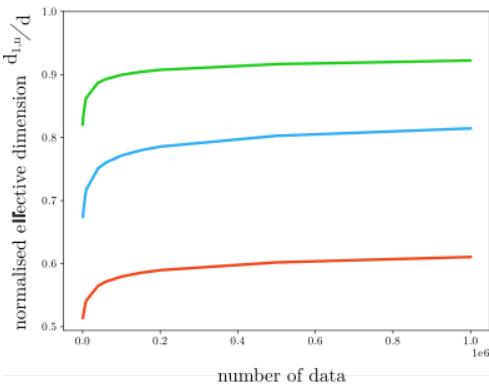
The HHL algorithm performs matrix inversion and, by extension, solves linear equation systems exponentially fast. Solving linear equation systems is pretty much ubiquitous in science and engineering, so the applications are obvious. The same is true for matrix inversion, which can be used in linear regression for example. There is a big caveat. The exponential speedup only holds as long as the solution state isn't read out from the quantum register! You can efficiently ask summary questions, like "what is the largest component?", or "what is the magnitude of the solution?", but reconstructing the whole state requires lots of readouts, such that "only" polynomial speedup is left. This sounds bad but a polynomial speedup is still a great achievement.

Furthermore, adiabatic quantum computing can make use of the *quantum tunneling effect* and thereby avoid getting stuck in local minima when solving optimization problems. This is great for training neural networks or other types of non-convex optimization problems.

13.4.3 Quantum Neural Networks

This is a highly controversial topic, and there are various architectures of which people say are quantum neural networks. There is no clear definition for what it actually means, and things will likely change drastically in the future. An idea could be to use qubits as neurons, or use a superposition of different ANNs for the training. One interesting example was found by [researchers from South Africa](#), who could also prove something about why quantum neural networks likely generalize better than their classical counterparts:

```
from IPython.display import Image
Image("img/qnn_ed.png")
```



QNNs learn much faster and achieve lower losses consistently.

As said, there is some critique about what a QNN actually is. The first part of the circuit looks like this:

```
S = 6

circuit = QuantumCircuit(S, S)

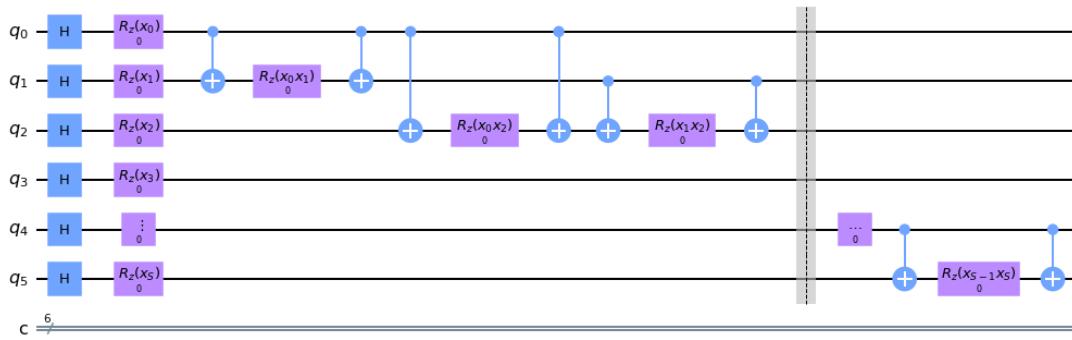
circuit.h(range(S))
for i in range(S-2):
    circuit.ry(0, i, label=r"\$R_z(x_{\text{\tiny"+str(i)+"}})\$")
circuit.ry(0, S-2, label=r"\vdots")
circuit.ry(0, S-1, label=r"\$R_z(x_S)\$")

for i in range(2):
    for j in range(i+1, 3):
        circuit.cnot(i, j)
        circuit.ry(0, j, label=r"\$R_z(x_{\text{\tiny"+str(i)+"}}x_{\text{\tiny"+str(j)+"}})\$")
        circuit.cnot(i, j)

if S > 4:
    circuit.barrier()
circuit.ry(0, S-2, label=r"\dots")

circuit.cnot(S-2, S-1)
circuit.ry(0, S-1, label=r"\$R_z(x_{\{S-1\}}x_S)\$")
circuit.cnot(S-2, S-1)

circuit.draw('mpl')
```



It prepares the “feature map” of the input for feeding through the QNN. The actual network layers look like this:

```

circuit = QuantumCircuit(S, S)

for i in range(S-2):
    circuit.ry(0, i, label=r"$R_y(\theta_{\text{"+str(i)+"}})$")
circuit.ry(0, S-2, label=r"$\vdots$")
circuit.ry(0, S-1, label=r"$R_y(\theta_S)$")

for i in range(2):
    for j in range(i+1, 3):
        circuit.cnot(i, j)
circuit.ry(0, 2, label=r"$\dots$")
if S > 4:
    circuit.barrier()
circuit.cnot(S-2, S-1)

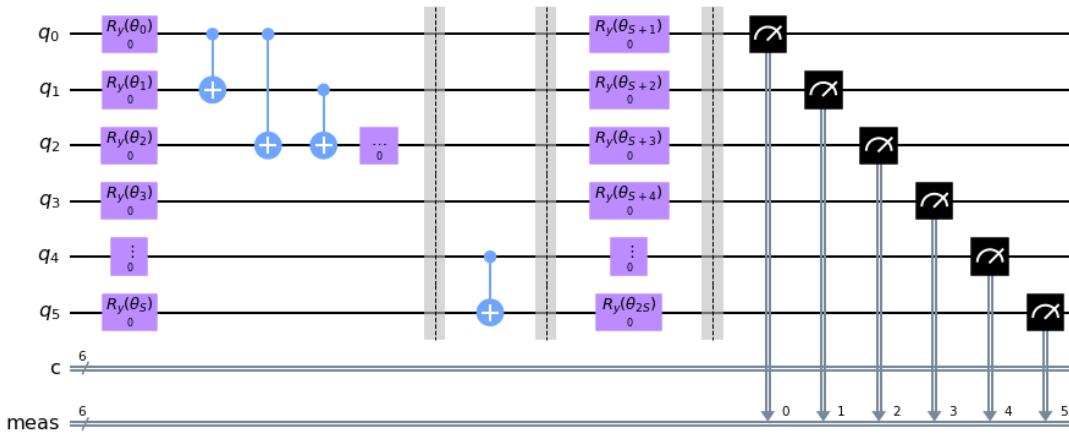
circuit.barrier()

for i in range(S-2):
    circuit.ry(0, i, label=r"$R_y(\theta_{\{S\}+\text{str(i+1)+"}})$")
circuit.ry(0, S-2, label=r"$\vdots$")
circuit.ry(0, S-1, label=r"$R_y(\theta_{\{2S\}})$")

circuit.measure_all()

circuit.draw('mpl')

```



The image above displays a single layer of such a QNN variational circuit, with depth 1 and $2S$ tunable parameters, where S is the size of the QNN, as in the number of qubits. For more layers, the above circuit would be repeated before the measurement. As you see, one could also describe this simply as a variational quantum circuit, where the parameters θ are optimized for some goal.

Sometimes thinking in quantum terms facilitates innovation in classical algorithms. There's an example of a recommender system, meaning that given some incomplete knowledge about customers it recommends them other products. A quantum algorithm was published that outperformed all known classical ones, but instead of giving up, researcher were inspired and created a quantum-inspired classical recommender system that is just as efficient as the quantum algorithm. NOTE: ADD LINKS

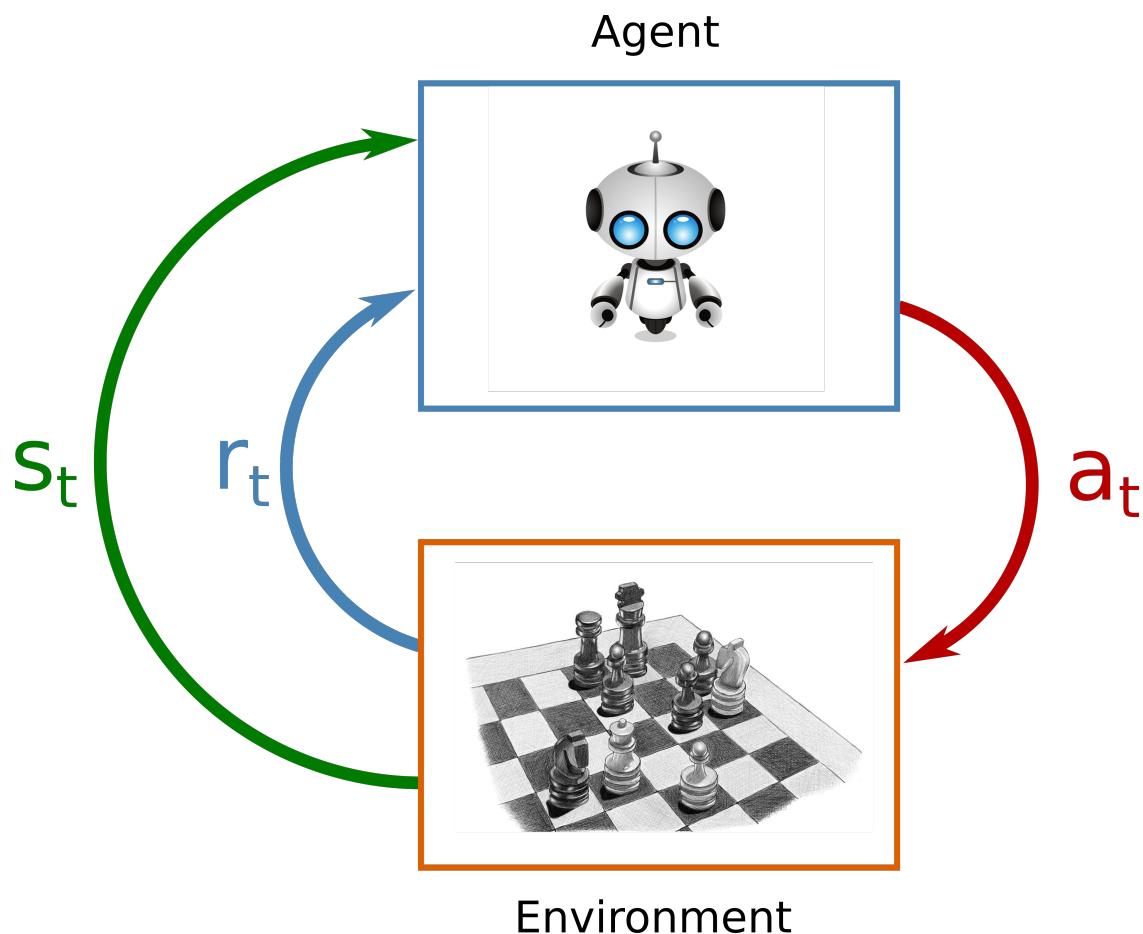
14 Reinforcement Learning

14.1 Reinforcement Learning

So far we discussed supervised techniques, e.g., regression, classification (logistic regression), object detection, semantic segmentation, and others. In these problems, a dataset consisting of *data* and corresponding *labels* or *targets* could be used to compare the output of our hypotheses, which processes the data, to the *ground truth*, consisting of the labels. We also discussed unsupervised learning, where only *data* was available without any labels, e.g., clustering, dimensionality reduction, and others. There are also semi-supervised techniques, where either incomplete data is available, or only some labels are used to guide the training process.

Reinforcement Learning is different from both approaches:

```
from IPython.display import Image
Image("img/r1.png")
```

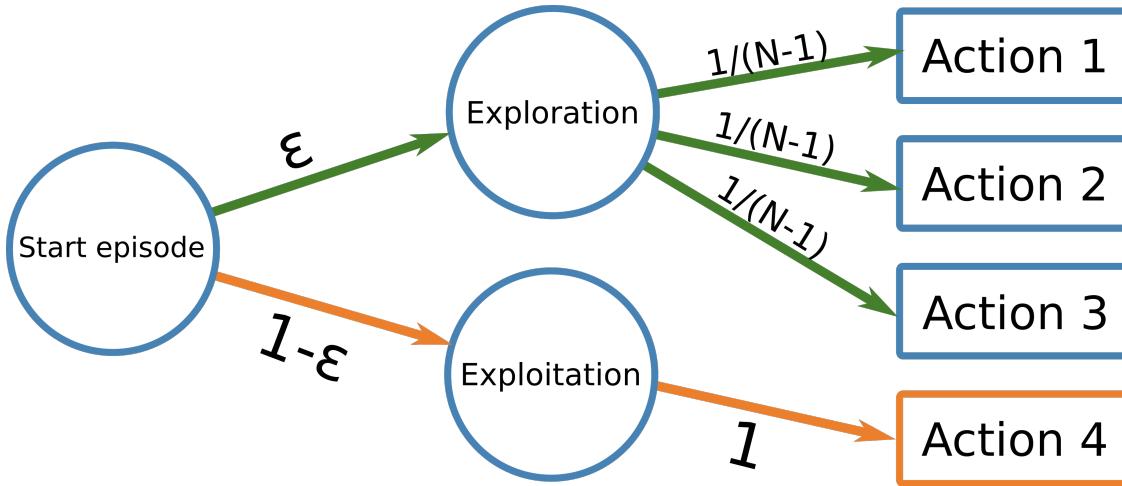


The idea here is that a problem is modelled as an **environment** in a state s_t , with which an **agent** can interact with an action a_t to receive a **reward** r_t that determines, in some sense, the quality

of its actions.

Many early examples exist for this kind of approach for teaching an agent how to play some kind of game, where the ultimate goal is to maximize some score or survive indefinitely. Reinforcement learning is predominantly used in control theory problems, but is applicable to any problem that can be *gamified*.

Image("img/exploration_vs_exploitation.png")



The idea for deep reinforcement learning is to use *artificial neural networks* for various tasks. A simple approach would be to model the agent as an ANN that performs some actions and learns from the rewards it gets from the environment. In the beginning, the ANN is initialized randomly, while in the end, it will always choose the best answers. This is highly detrimental, especially in high-dimensional problem setups. The problem is called the **exploration-vs-exploitation tradeoff**. A perfectly trained ANN will exploit the information from the environment to choose the best option as soon as it found something that rewards the chosen actions. This isn't always the globally best option. Imagine the ANN playing a Super Mario game, where it simply always goes to the right. The actions necessary for that are relatively simple and lead to the goal. A better solution would be for the ANN to also collect coins and other goodies, take out enemies and explore pipes for hidden treasures. Without exploration, this would be difficult for the agent to find. It's the equivalent of other methods becoming stuck in local minima.

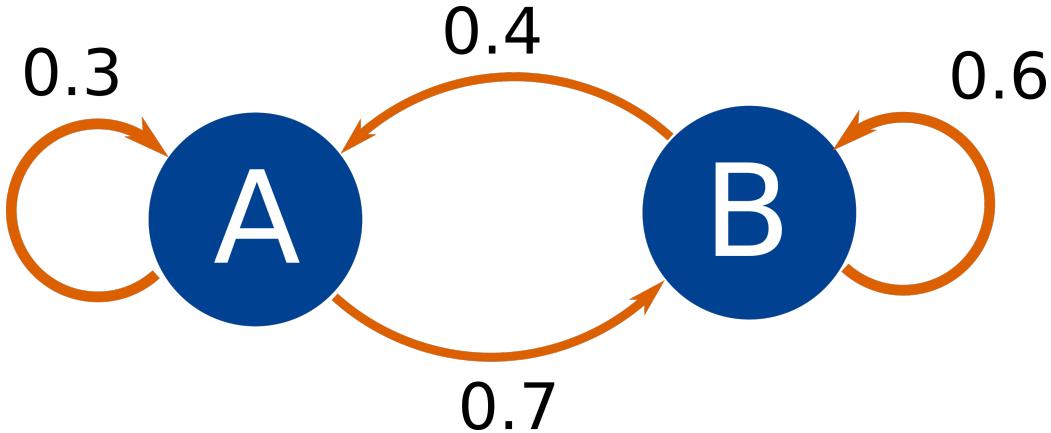
Hence, reinforcement learning needs mechanisms to enforce **exploration**, meaning that with a certain probability random actions are chosen to make the agent *explore* the environment outside the current best approach it learned. The image above shows the **ε -greedy algorithm**.

14.2 Markov Decision Processes

Mathematically, reinforcement learning problems are usually formulated in terms of **Markov Decision Processes**. They are a generalization of Markov chains, which stochastically describe sequences of events in which the probability of an event happened depends *only* on the state of the directly *previous* state of the system. This is called the *Markov property*.

An example for a super simple Markov chain is the following system:

```
from IPython.display import Image
Image("img/Markov_chain.png")
```



There are many examples of systems which are modeled as Markov chains. The example above could be interpreted as a super simple simple weather model, where the next prediction only depends on the last prediction, not older data.

Systems like these are often expressed by *transition matrices*:

$$\begin{pmatrix} 0.3 & 0.4 \\ 0.7 & 0.6 \end{pmatrix} \text{ Transition matrix}$$

$$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ Initial state}$$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} 0.3 & 0.4 \\ 0.7 & 0.6 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ Next state}$$

It obviously does not accurately reflect reality. Other examples are Brownian motion and random walks in general.

Markov decision processes add possible *actions* and *rewards* to this framework. Markov chains are the special case in which the only possible action is to *wait* or do nothing, and the reward is constant.

A Markov decision process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbf{P}, \gamma)$ with

- \mathcal{S} : the set of all attainable states, i.e., the *state space*
- \mathcal{A} : the set of all possible actions, called the *action space*
- \mathcal{R} : the distribution of rewards depending on a state-action pair (s_t, a_t)
- \mathbf{P} : the transition probability, giving the probability that in state s_t , the action a_t will lead to the state s_{t+1} : $p(s_{t+1}|s_t, a_t)$

- γ : the *discount* factor $0 \leq \gamma \leq 1$, sometimes as $\gamma = \frac{1}{1+r}$ with the *discount rate* r , determines the delay of rewards for actions

With these definitions, the Markov decision process works like this:

- $t = 0$: environment samples initial state $s_0 \sim p(s_0)$
- loop until done:
 - agent selects an action a_t
 - environment samples reward $r_t \sim \mathcal{R}(s_t, a_t)$
 - environment samples next state $s_{t+1} \sim \mathcal{S}(s_t, a_t)$
 - agent receives both reward r_t for its chosen action and next state s_{t+1}

The agent has to learn a **policy** $a_t = \pi(s_t)$ that tells it which action to choose for a specific state of the environment.

The goal is to maximize the *expected cumulative reward*

$$\bar{R}_c = E \left[\sum_{t \geq 0}^T \gamma^t r_t \right]$$

The policy that maximizes this expected cumulative reward is called the **optimal policy** π^*

$$\pi^* = \arg \max_{\pi} E \left[\sum_{t \geq 0}^T \gamma^t r_t | \pi \right], \quad s_0 \sim p(s_0), a_t \sim \pi(s_t), s_{t+1} \sim p(s_t, a_t)$$

Examples for Markov decision processes are games of all sorts. Below is a sketch for a simplified game of *Snake*, an old mobile phone game commonly found on Nokia devices, where the goal is to maneuver a snake to, let's say apples, that make the snake grow on a grid world. For ease of display and understanding, let's assume the snake won't grow and is actually just the size of one grid spot. Let's also assume the game is finished as soon as the snake reaches the apple. The snake is hurt and needs to wear a boot, so instead of crawling it has to jump discretely from one grid space to an adjacent grid space. In general, Markov decision processes do not have to be discrete.

`Image("img/bootstrap_snake_mockup.png")`



The set of possible actions is $\mathcal{A} = \{\text{left, right, up, down}\}$.

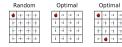
The set of possible states are all the possible positions of the snake and the apple on the grid above, where the apples never appear on the grid space where the snake resides. This is a huge state space already!

The “done” condition is reached when the snake reaches the apple.

A constant negative reward is given for each movement. Reaching the apple gives a sufficiently high positive reward. This minimizes the number of steps the agent takes.

You could also include traps that result in a high negative reward so the agent has to avoid those, or put up several apples to complicate things. The images below show the random policy and the optimal policy for the case above.

```
Image("img/bootsnake_policies.png")
```



Taking actions according to a policy and following the states and rewards associated with them gives a **trajectory** in the sense that points s_t, a_t, r_t are sampled that create a *path* $\{(s_0, a_0, r_0), (s_1, a_1, r_1), \dots\}$ in the problem space. A triple of points s_t, a_t, r_t is called a sample. A trajectory that reaches the “done” state is called an **episode**.

For a given π , the **Value function** gives the expected cumulative reward for a starting state s , so it measures the *quality* of a state:

$$V^\pi(s) = \mathbb{E} \left[\sum_{t \geq 0}^T \gamma^t r_t | s_0 = s, \pi \right]$$

For given π , starting action a and starting state s the **Q -value function** measures how good a state-action pair is by computing the expected cumulative reward from taking action a in state s :

$$Q^\pi(s, a) = \mathbb{E} \left[\sum_{t \geq 0}^T \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

The key distinction here is that the Q -value function is calculated with a given action, while the value function starts with an action suggested by the policy. The optimal Q -value function Q^* is the maximum of the Q -value function over all policies π .

Typical solution variants for this kind of problem are *dynamic programming* algorithms, which try to subdivide the problem into smaller, manageable chunks and recursively solve the subproblems to find the optimal solution. This is done a lot in, e.g., *control theory*. The relation of values for larger problems and the subproblems is captured by the

Bellman equation

$$Q^*(s, a) = \mathbb{E} \left[r + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) | s, a \right]$$

where s_{t+1} is the state we end up with after taking the action a and r is the reward for action a in state s . So the best action to take is that a , that maximizes the sum of the current reward and the optimal Q -value function at the next time step, which has to be known. The optimal policy π^* would then mean to take the action that maximizes Q^* .

The Bellmann equation can be used to formulate an optimization algorithm:

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a_{t+1}} Q_i(s_{t+1}, a_{t+1}) | s, a \right]$$

$$\lim_{i \rightarrow \infty} Q_i = Q^*$$

Obviously, this is completely unfeasible since the Q -values for each state-action pair has to be calculated and the space grows extremely quickly.

There are various ways to tackle this problem. Here, we'll concentrate on machine learning.

14.3 Q-Learning

Calculating the Q -value function for each state-action pair is unfeasible, and no problems of interest could be solved this way. Keeping in mind the last lecture and how to identify opportunities for machine learning, we can approximate this function with an artificial neural network. This is called

Q-learning

$$Q^* \approx \hat{Q}(s, a; \theta)$$

where θ are the *weights* in the ANN. If a deep ANN is used (i.e., at least one hidden layer), this is often called *deep Q-learning*. The loss for Q -learning is

$$L(\theta_i) = \mathbb{E} \left[(\mathbb{E}[y_i | s, a] - \hat{Q}(s, a; \theta_i))^2 \right]$$

$$y_i = r + \gamma \max_{a_{t+1}} \hat{Q}(s_{t+1}, a_{t+1}; \theta_{i-1})$$

$$\nabla_{\theta} L(\theta_i) = \mathbb{E} \left[(y_i - \hat{Q}(s, a; \theta_i)) \nabla_{\theta} \hat{Q}(s, a; \theta_i) \right]$$

So the idea is to minimize the difference between the current hypothesis and the sum of the current reward r for (s, a) and the prediction for the resulting Q -value from the last iteration of the training, but for the states that the chosen action a would result in. This loss will (try to) enforce the Bellmann equation for the ANN predictions.

Let's look at an example:

```
import matplotlib.pyplot as plt
import gym
from ipywidgets import interact
import skvideo.io
from pyvirtualdisplay import Display
display = Display(visible=0, size=(400, 300))
display.start()

env = gym.make("Breakout-v0")
env.seed(1)
```

```
n_actions = env.action_space.n

env.reset()

def move_board(action):
    plt.cla()

    if "start" in action:
        obs, reward, done, info = env.step(1)
    elif "left" in action:
        obs, reward, done, info = env.step(2)
    elif "right" in action:
        obs, reward, done, info = env.step(3)
    else:
        pass
    frame = env.render('rgb_array')

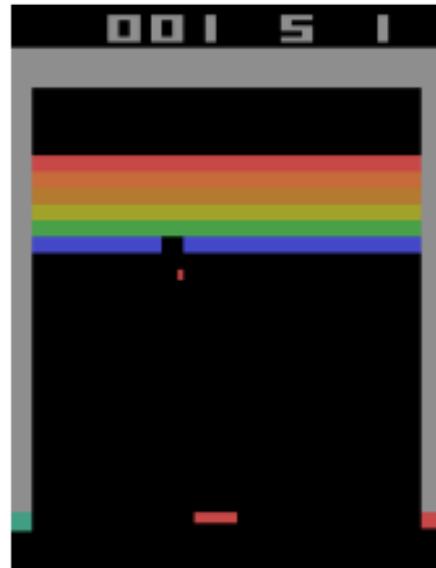
    plt.imshow(frame)

    if done:
        print("Game over (or you won.)")

    plt.axis("off")
    plt.savefig("img/plot_breakout.png")

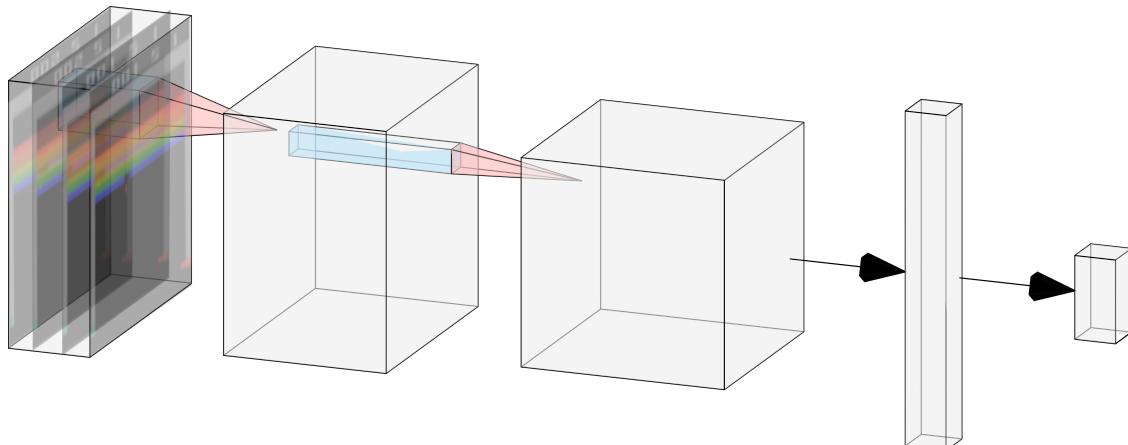
interact(move_board, action=["start", "left1", "right1", "left2", "right2"])

interactive(children=(Dropdown(description='action', options=('start', 'left1', 'right1', 'left2', 'right2')),
```



This is the classic *Breakout* Atari game. There are various ways to implement a reinforcement learning algorithm here, so let's look at what Mnih et al. did. As input, the four last consecutive images from the game screen were preprocessed and squished to 84×84 pixels and used as input for a convolutional layer, another convolutional layer, a fully connected layer of size 256 and finally a fully connected layer of size 4.

```
from IPython.display import Image
Image("img/breakout_network_ims.png")
```



The reason for the output layer containing 4 neurons is that the action space of the breakout game is of size 4:

```
env.action_space
```

```
Discrete(4)
```

Hence, the ANN can output the Q -value for each action in the action space *at once*. This way, we can directly choose the option that maximizes the Q -value function.

14.3.1 Experience Replay

Using episodes for the training that were gathered consecutively from, say, playing the game is problematic in the same way consecutive sampling is problematic for supervised learning. The sample episodes will be correlated and make training inefficient. Additionally, the next action is determined by the current training state of the ANN and the next sample episodes will be generated mostly according to this specific action. Hence, the next episodes will have a *bias* towards the current best action, which introduces bias into the network predictions and may even lead to feedback loops. For example, if one such action is “go left”, the whole network may attain a tendency to predict “go left” actions, even if other options would be better.

Experience Replay addresses these issues by introducing a **replay memory** D that saves transitions (s_t, a_t, r_t, s_{t+1}) , so that random minibatches can be sampled from this memory for training the ANN. This also increases the data efficiency, since sampling minibatches means that examples can be chosen more than once.

The full algorithm now looks like this (also taken from Mnih et al.):

```
Image("img/RL_with_ER.png")
```

```

For episode = 1, M do
    Initialize sequence  $s_1 = \{x_1\}$  and preprocessed sequence  $\phi_1 = \phi(s_1)$ 
    For  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $D$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $D$ 
        Set  $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{otherwise} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  with respect to the
        network parameters  $\theta$ 
        Every C steps reset  $Q = Q$ 
    End For
End For

```

An episode terminates when either the goal is reached or the environment failed in some way. For the game example, finishing the level or losing the ball would end an episode. ϵ determines the exploration probability and should be small. It addresses the exploration-exploitation tradeoff mentioned before, by taking a random action with a small probability instead of the best action. The replay memory is reset before each episode.

Reinforcement learning applied to games led to some pretty interesting insights about the games themselves. You can see a video from [two minute papers](#) about the breakout game from above to see that after some training, the ANN adopts expert strategies. More often than not, ANNs will find cheats, shortcuts and glitches it can exploit to reach its goal quickly and effortlessly. See for example this game of [hide and seek](#), or this agent that learned to play [Qbert](#) that found a way to get infinite points by deliberately killing the character (this was due to a bug in the emulator). Such agents are currently also evaluated as methods for bugtesting and finding glitches in software.

14.4 Policy Gradients

Q -value function become very complicated quickly. Imagine a robot gripper that is supposed to learn how to pick up and move objects (check out [this robot](#) outsmarting its creators at 0:58 onwards). It's difficult to determine the values of each exact state-action pair for every situation, since every time the object could be placed differently, or the force applied by the gripper could be different and so on. A simple policy could be to simply approach the object and close the grippers until a certain force is reached.

The idea of **Policy Gradients** is to find the best possible policy among a collection of policies without having to estimate any values.

The idea is to define a parameterized family of policies

$$\Pi = \{\pi_\theta, \theta \in \mathbb{R}^N\}$$

and the reinforcement learning goal

$$L(\theta) = \mathbb{E}[r(\tau)|\pi_\theta]$$

with the total reward $r(\tau) = \sum_{t \geq 0} \gamma^t r_t$ for a given trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, \dots)$. The optimal policy is determined by $\theta^* = \arg \max_{\theta} L(\theta)$.

An obvious choice for solving this problem is *gradient ascent*, which is simply the gradient descent rule we already know but with adding the loss gradient instead of subtracting it. It's not straightforward to apply, since τ is sampled from various probability distributions and $L(\theta)$ is an expectation value, so we'll have to use a trick to reformulate the gradient.

Assuming continuous τ , the objective function can be reformulated as

$$L(\theta) = \int_{\tau} r(\tau)p(\tau; \theta)d\tau$$

so that

$$\nabla_{\theta} L(\theta) = \int_{\tau} r(\tau)\nabla_{\theta} p(\tau; \theta)d\tau$$

We're running into the same problem that we had with the variational autoencoders. Taking the gradient of a probability distribution is intractable. If $r(\tau)$ depended on θ , this wouldn't pose a

problem but here, the probability distribution itself is parameterized. The trick is now to multiply with a “smart one”:

$$\nabla_{\theta} p(\tau; \theta) = p(\tau; \theta) \frac{\nabla_{\theta} p(\tau; \theta)}{p(\tau; \theta)} = p(\tau; \theta) \nabla_{\theta} \log p(\tau; \theta) \implies$$

$$\begin{aligned} \nabla_{\theta} L(\theta) &= \int_{\tau} (r(\tau) \nabla_{\theta} \log p(\tau; \theta)) p(\tau; \theta) d\tau \\ &= E[r(\tau) \nabla_{\theta} \log p(\tau; \theta)] \end{aligned}$$

The trick transformed the gradient of an expectation value to the expectation value of a gradient! This can now be sampled e.g. with Monte Carlo methods, i.e., sampling trajectories randomly to evaluate the integral.

The probability of a trajectory is the product of all transition probabilities and action probabilities. Just like the probability of two dice showing a 3 is the product of each individual probability:

$$p(\tau; \theta) = \prod_{t \geq 0} p(s_{t+1} | s_t, a_t) \pi_{\theta}(a_t | s_t)$$

Using the log-likelihood trick from before makes this

$$\log p(\tau; \theta) = \sum_{t \geq 0} \log p(s_{t+1} | s_t, a_t) + \log \pi_{\theta}(a_t | s_t)$$

so the gradient does *not depend on the transition probabilities anymore!*

$$\nabla_{\theta} \log p(\tau; \theta) = \sum_{t \geq 0} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

With a sampled trajectory τ , $\nabla_{\theta} L(\theta)$ can now be estimated:

$$\nabla_{\theta} L(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

This is tractable, since we parameterized the policies in the beginning and hence can take the gradient without any problem. The intuition behind this is that when $r(\tau)$ is high, the probabilities of the actions taken for that trajectory τ are increased, while the opposite happens if $r(\tau)$ is small.

It seems a bit too loose of an assumption to say that whenever a trajectory leads to a good reward, all its actions were useful and that's true, it's rarely the case. The thing here is that many such trajectories are *averaged over* and thus, bad actions are “averaged out” while good actions sort of “interfere constructively”. This approach needs a huge number of samples, because the variance in the actions will be very high.

14.4.1 Variance Reduction

To reduce the number of samples necessary and help the gradient estimator, different methods can be employed. Most of them rely on using a modified reward instead of the total reward $r(\tau)$ for the whole trajectory. One idea is to only boost the probability of actions taken by the cumulative *future* reward after the action was taken on a trajectory:

$$\nabla_{\theta} L(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

A modification of this is to use a *discount factor* γ after an action a_t for damping the rewards coming from later actions:

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

This approach “localizes” the reward to the neighborhood of an action on the trajectory. There’s still a huge problem: all rewards are positive, meaning that all probabilities of all actions will always be increased. Like with loss functions, the numerical value of the reward is pretty much useless. What counts is how much above or below a certain expectation we have it is. This expectation of which reward would make sense is called a **baseline** $b(s_t)$, with respect to which the reward can be measured instead:

$$\nabla_{\theta} J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

There are various ways to determine a baseline. A simple approach is to use the moving average with some window size of rewards from former trajectories.

A moving average operation uses a *window* of a certain size and calculates the mean of the values inside the window. It will take, e.g., 5 values of a curve and compute their mean, then display the resulting value as the 6th point.

You can try an example with different window sizes below:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from ipywidgets import interact

plt.figure(figsize=(12,9))

t = range(100)
x = t*(np.random.random(100)+1)

df = pd.DataFrame(x)
```

```
def moving_average(window_size, average=False):
    if average:
        plt.plot(df.rolling(window=window_size).sum(), lw=3, label=f"window_{window_size} {window_size}")
    else:
        plt.plot(df, lw=3, label="raw")

    plt.legend(fontsize=16)

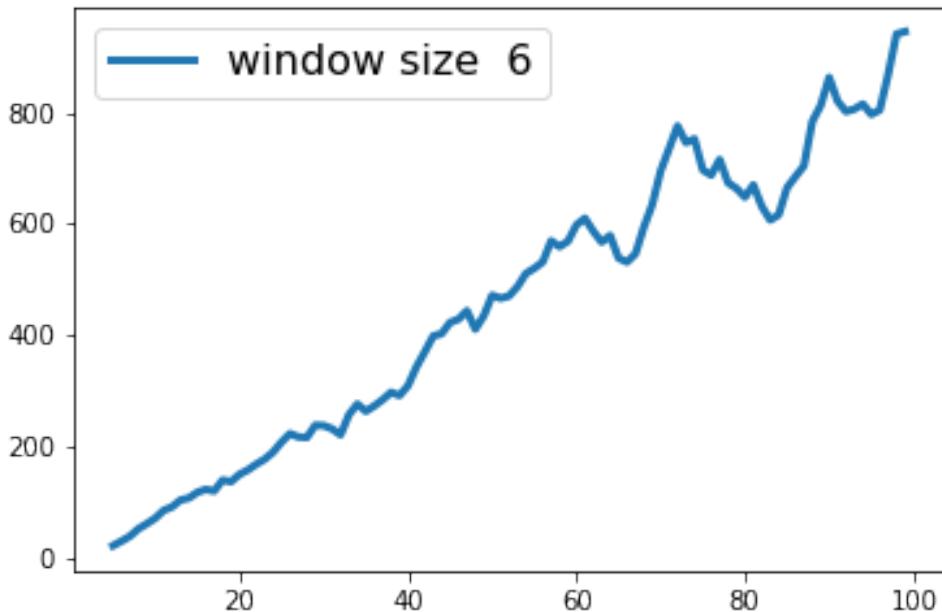
    plt.savefig("img/plot_moving_average.png")

interact(moving_average, window_size=(2,10), average=False)
```

<Figure size 864x648 with 0 Axes>

```
interactive(children=(IntSlider(value=6, description='window_size', max=10, min=2), Checkbox(va
```

```
<function __main__.moving_average(window_size, average=False)>
```



14.5 The Actor-Critic Algorithm

In the last lesson, we introduced a baseline to reduce the variance in the policy gradient approach. A better idea is to define the baseline as the expected value for a given state, i.e., using the *Q-value and value functions*. In a state s_t an action a_t is good iff the difference $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is large, while when it's small the action was bad. Hence, this difference can be used as the estimated reward and baseline:

$$\nabla_\theta L(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

Q unknown \rightarrow use Q -learning! Policy gradients trains the **actor** (the policy), and Q -learning trains the **critic** (the Q -value function).

The difference is usually called the **advantage function** $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$. This idea is similar to generative adversarial models, where generator and discriminator trained each other. Here, the actor suggests an action to take and the critic provides feedback for the quality of the move and how to adjust the probability of taking that action. Now, the critic only has to evaluate those state-action pairs that the actor generated instead of pretty much all possible combinations, so this helps as well.

The algorithm is as follows (taken and adapted from [here](#)):

- Initialize policy θ and critic weights ϕ
- **for** episode $i = 1, 2, \dots$ **do**
 - Sample N trajectories from current policy π_{θ_i}
 - $\Delta\theta = 0$
 - **for** trajectory $\tau = 1, 2, \dots$ **do**
 - * **for** time step $t = 1, 2, \dots$ **do**
 - $A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^{\tau} - V_\phi(s_t^{\tau})$
 - $\Delta\theta += A_t \nabla_\theta \log(a_t^{\tau} | s_t^{\tau})$
 - * **end for**
 - **end for**
 - $\Delta\phi = \sum_{\tau} \sum_t \nabla_\phi ||A_t^{\tau}||^2$
 - $\theta += \alpha \Delta\theta$
 - $\phi += \beta \Delta\phi$
- **end for**

14.6 Summary and Examples

14.6.1 Summary

Q -learning tries to approximate the Q -value function with an ANN. It does not explicitly need an environment model (and is often called model-free). It is difficult to train, but is very efficient in making use of the available samples if it works.

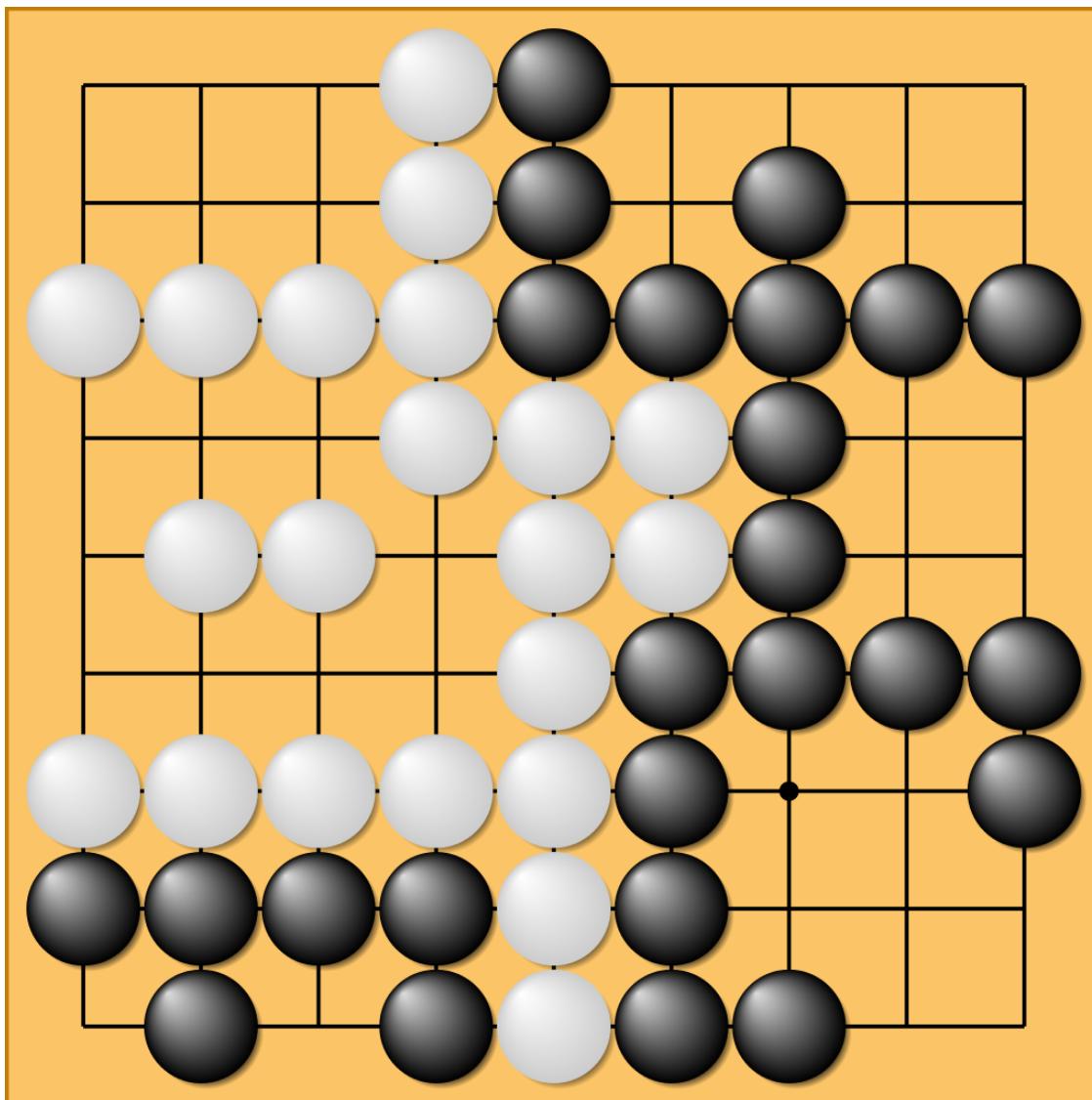
Policy gradients are a very general technique, but the high variance it suffers from makes it necessary to generate a lot of samples. It learns a stochastic policy, which naturally introduces exploration.

Actor-critic models try to combine both approaches and find a baseline value by taking the difference of the Q -value function and the value function.

14.6.2 Examples

The ancient game of *Go* was long thought to be completely intractable for bots to win. The rules are simple, but the gameplay is highly complex. Hence, the announcement of *AlphaGo* caused quite a stir in the community, especially after consistently beating world champions. It has since been updated to even better and much smaller models, but here we'll concentrate on the initial contender.

```
from IPython.display import Image  
  
Image("img/go_board_small.png")
```



- Create features describing the board (positions, colors, legal moves, ...)
- Policy network is initialized with professional games (supervised learning)

- Policy is further trained by playing against itself with a +1 reward for winning, -1 for losing
- Train the value network (the critic)
- Use Monte Carlo tree search for lookahead search to select the best actions from the combined actor-critic networks

This idea of combining reinforcement learning with Monte Carlo tree search is not restricted to games. [Sawada et al.](#) applied this combination to multicomponent material design optimization and showed that it scales at least as good as Bayesian optimization and that it is much less prone to falling for local minima. The idea is to have a structure with a number of positions, and each position has to be assigned an atom from a predefined set of viable atoms. The assignment is evaluated by a simulation or an experiment to give the agent feedback. They provide their code [MDTS on github](#).

The same idea can be applied to topology optimization, which [Cui et al.](#) have shown for designing a cargo ship, or [Kang et al.](#) for general topology optimization in a generative manner.

[Steve Brunton](#) has a YouTube channel where he and colleagues explain various machine learning concepts for computational fluid dynamics. The video shows some examples for how reinforcement learning can be applied in CFD.

In the end, only your creativity is the limit for identifying viable options.

15 Physics-informed Machine Learning

We already discussed the problem that ANNs need to learn *symmetries* of problems from data in lecture 04. That is, if we do not enforce our knowledge of a problem into the ANN model itself. There are several ways to approach this shortcoming. One way was to simply *augment* the training data, such that it reflects the symmetry we'd like the ANN to learn about the problem. This is only possible if we already know or at least assume the symmetries of a problem. In all other cases, we're lost.

The other way is to incorporate symmetries directly into the model itself. We saw how CNNs are translationally *equivariant* (not invariant), meaning that they do not have to learn this property from data. RNNs have built into them the assumption that how a transition happens from one timestep to the next is constant for the whole series.

The expectations for incorporating physics knowledge into machine learning are

- Data efficiency
- Faster convergence
- Better generalizability
- Increased trustworthiness
- Better predictions

This lecture will look at a few techniques of incorporating physical knowledge about a problem into ANN models.

15.1 Solving ODEs with Parameterized Functions

An *Ordinary Differential Equation* of order n is an equation $F(x, u'(x), u''(x), \dots, u^{(n)}(x)) = 0$, that relates different orders of derivatives of a function u with respect to the coordinate x . Any explicit ODE can be rewritten as a first-order vector ODE with new functions $u_i = u^{(i)}$ in the following way:

$$F(x, u'(x), u''(x), \dots, u^{(n-1)}(x)) = u^{(n)}(x)$$

$$\begin{aligned} u'_1 &= u_2 \\ u'_2 &= u_3 \\ &\vdots \\ u'_{n-1} &= u_n \\ u'_n &= F(x, u_1(x), u_2(x), \dots, u_2(x)) \end{aligned}$$

Or as a vector:

$$\mathbf{u}' = \mathbf{F}(x, \mathbf{u})$$

Usually, problems involving ODEs are given as *initial value problems*, such that an initial condition is given:

$$\mathbf{u}(0) = \mathbf{u}_0$$

With a little bit of clever rearrangement, the problem can be reformulated as an optimization problem for a *parameterized* solution approximator $u(x; \theta)$ with parameters θ . These considerations have been discussed in [Lagaris et al 1997](#) almost 25 years ago, and went largely unnoticed until very recently.

This reformulation automatically satisfies the initial value:

$$\hat{\mathbf{u}}(x; \theta) = \mathbf{u}_0 + x\mathbf{N}(x; \theta)$$

Objective function is the *integrated squared residual*:

$$L(\theta) = \int_0^1 \left[\frac{\partial \hat{\mathbf{u}}(x; \theta)}{\partial x} - \mathbf{F}(x, \hat{\mathbf{u}}(x; \theta)) \right]^2 dx$$

The term in the square brackets is simply the first-order vector ODE with all the terms shifted to the left. Obviously we can use stochastic gradient descent again to optimize:

$$\theta_{i+1} = \theta_i - \frac{\alpha_t}{n} \sum_{j=1}^n \nabla_\theta \left[\frac{\partial \hat{\mathbf{u}}(x_j; \theta)}{\partial x} - \mathbf{F}(x_j, \hat{\mathbf{u}}(x_j; \theta)) \right]^2$$

The x_j are sampled randomly from the interval $x_j \in [0, 1]$ in each iteration.

The form of parametrization is still open, and generally any parametrization can be used to try and solve the problem as posed above. Of course, we'd like to use ANNs as the function approximators here. This will be explained in the next lesson.

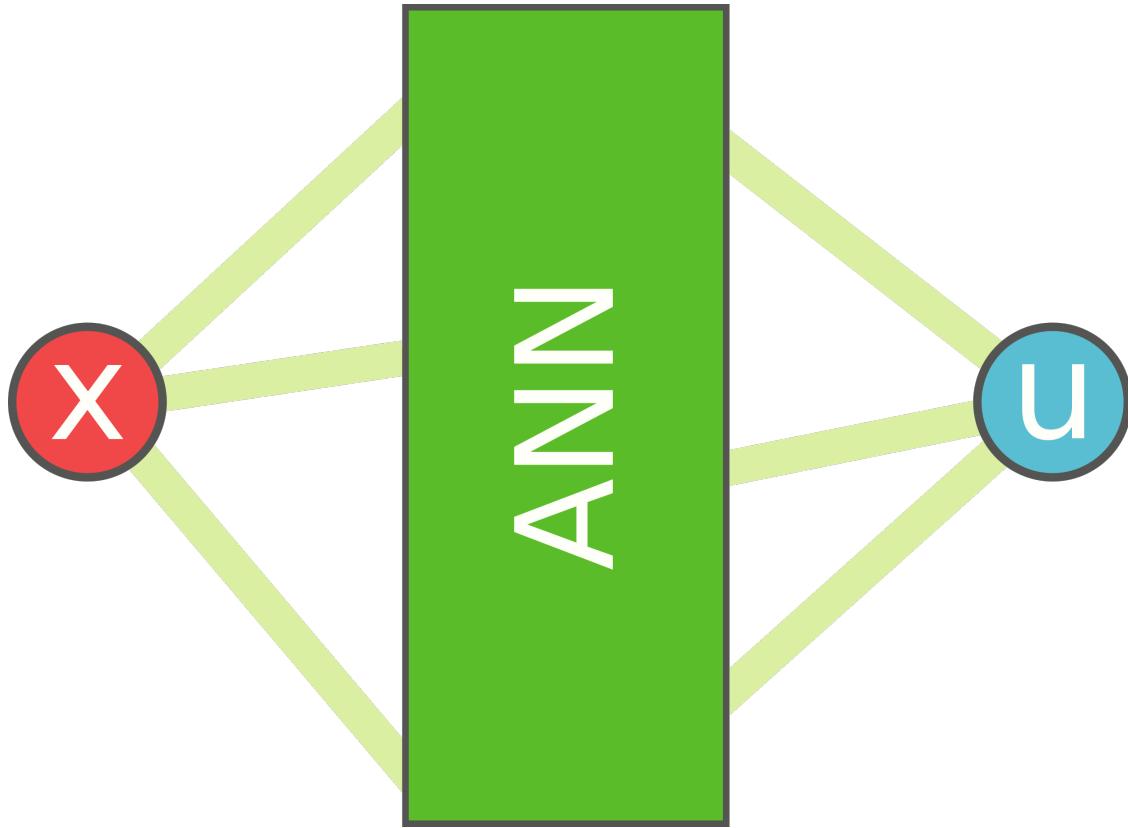
15.2 Physics-informed Neural Networks

The main task in science and to a great extent also in engineering is to develop differential equations that describe the development of certain systems, approximating natural phenomena. There is a way to use ANNs to directly “solve” differential equations. This approach is inspired by collocation methods for solving differential equations. The idea goes back to [Lagaris et al 1997](#), mentioned in the last lesson, and has since been refined several times [in this paper series](#). Consider for example the following ODE and boundary conditions:

$$\frac{\partial^2 u}{\partial x^2} + a \frac{\partial u}{\partial x} - b = 0, u(0) = u_0, u(1) = u_1 \quad (209)$$

Now a neural network that models this ODE could look like this:

```
from IPython.display import Image
Image("img/pinn.png", width="500")
```



with unknown weights. From the *universal approximation theorem* we know, that ANNs can model arbitrary nonlinear functions arbitrarily well, regardless of the underlying differential equation. Since by choosing a number of layers and a number of neurons per layer we also fixed the functional dependency of all of these variables. So now we're able to give a closed expression of the derivative of the output layer u with respect to the inputs x . This derivative can be expressed solely by parameters of the neural network. In fact, it's quite easy to find this derivative, since we also needed this exact same dependency for backpropagation, where it was found by using *automatic differentiation (autograd)*. Most machine learning frameworks have inbuilt functions for this. In TensorFlow, it looks like this (from the Burger's equation example in the paper mentioned above):

```
def u(x, t):
    return neural_net(tf.concat([x, t], 1), weights, biases)

def f(x, t):
    u = u(x,t)
    u_t = tf.gradients(u, t)[0]
    u_x = tf.gradients(u, x)[0]
    u_xx = tf.gradients(u_x, x)[0]

    return u_t + u*u_x - (0.01/tf.pi)*u_xx
```

This looks elegant and easy, but distracts a bit from the requirements that lead to such an elegant expression. You can see what work goes into this in the [github repository](#) from [Maziar Raissi](#), one

of the authors of the abovementioned paper, where he provides the full code that was used to create the results shown in the paper (be aware that this is in TensorFlow 1.x. In the assignment today, we'll see how this works with current libraries).

So now that $\hat{u}(x) = \text{ANN}(x)$ we can find expressions for $\hat{u}'(x)$, $\hat{u}''(x)$ and so on to reconstruct the original differential equation. The idea is now to take the difference of the original ODE and this ANN ODE $\widehat{\text{ODE}}$ and square it:

$$(\text{ODE} - \widehat{\text{ODE}})^2 = 0 \quad (210)$$

This effectively transforms solving (or approximating) the original ODE into an optimization problem. Now, since the original ODE is exact $\text{ODE} = 0$, the only term left here is

$$(\widehat{\text{ODE}})^2 = 0 \quad (211)$$

which solely consists of parameters of the ANN. This is the cost function to be minimized, and we can use all the techniques we got to know to do so:

- randomly initialize weights
- feed-forward some randomly chosen x
- get loss gradient
- backpropagate error and update weights
- repeat until convergence/criterion is met

This naive approach will only satisfy the ODE, but not the given boundary conditions, but it's simple to add them as an additional constraint to the loss function:

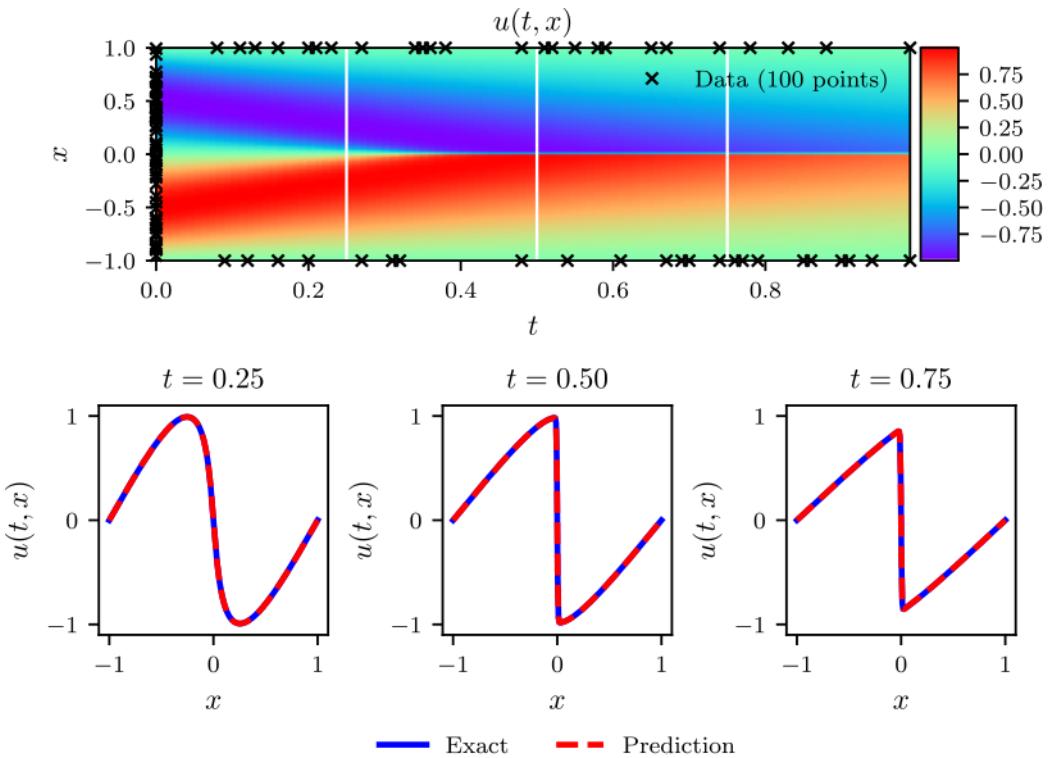
$$(\widehat{\text{ODE}})^2 + (\hat{u}(0) - u_0)^2 + (\hat{u}(1) - u_1)^2 = 0 \quad (212)$$

In this form, the ODE including the boundary conditions is completely converted to an optimization problem. This approach is called **physics-informed neural network**, because unlike in previous approaches we've seen, the loss function incorporates knowledge about the underlying physics of the problem.

The points x that are chosen in between the boundaries of where the solution is desired to be found are called *collocation points*. In the paper mentioned above, different mean versions of the loss were used for collocation points on the boundary and inside the solution domain, since only a small part of the loss comes from points on the boundary and most of it comes from the inside. Taking separate means here balances out the loss factors a bit.

The results gathered from such an approach are extremely promising, as this example from the paper mentioned on top shows for the Burger's equation:

```
Image("img/burgers_pinn_from_paper.png", width="800")
```



So far, only ODEs were mentioned but it's easy to generalize to PDEs by simply allowing more input features. Vector fields are modeled by an output layer using more than a single neuron. The rest stays exactly the same as described.

15.3 Common Problems with PINNs

15.3.1 Spectral Bias

Like all things in life, the simple approaches are rarely the holy grail of solving every last problem. The same is true for PINNs. Take a look at the following two images of a PINN trying to solve the heat diffusion problem (you'll see a video of the training process in the assignment today):

```

import matplotlib.pyplot as plt
import matplotlib.image as mpimg

heat_pinn_300 = mpimg.imread("img/heat_pinn_300.png")
heat_pinn_1000 = mpimg.imread("img/heat_pinn_1000.png")
heat_pinn_5000 = mpimg.imread("img/heat_pinn_5000.png")

fig,axs = plt.subplots(3,1, figsize=(12,21))

axs[0].imshow(heat_pinn_300)
axs[0].set_title("300 Epochs", fontsize=20)
axs[1].imshow(heat_pinn_1000)

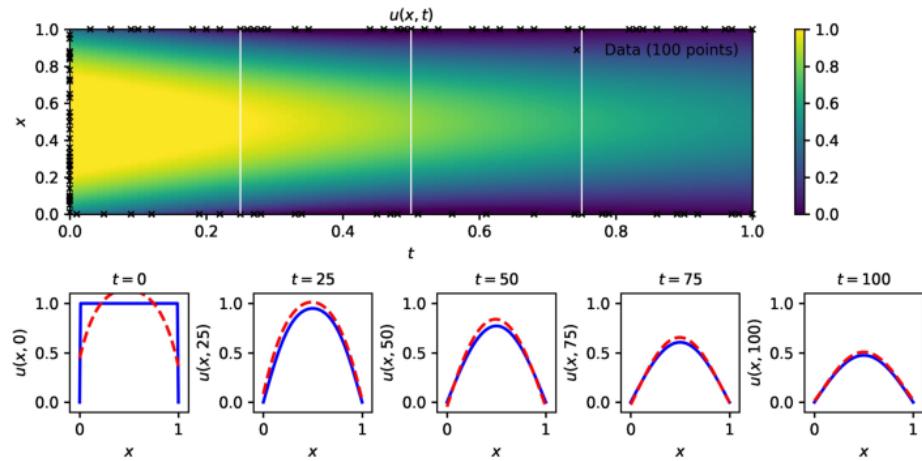
```

```
axs[1].set_title("1000 Epochs", fontsize=20)
axs[2].imshow(heat_pinn_5000)
axs[2].set_title("5000 Epochs", fontsize=20)

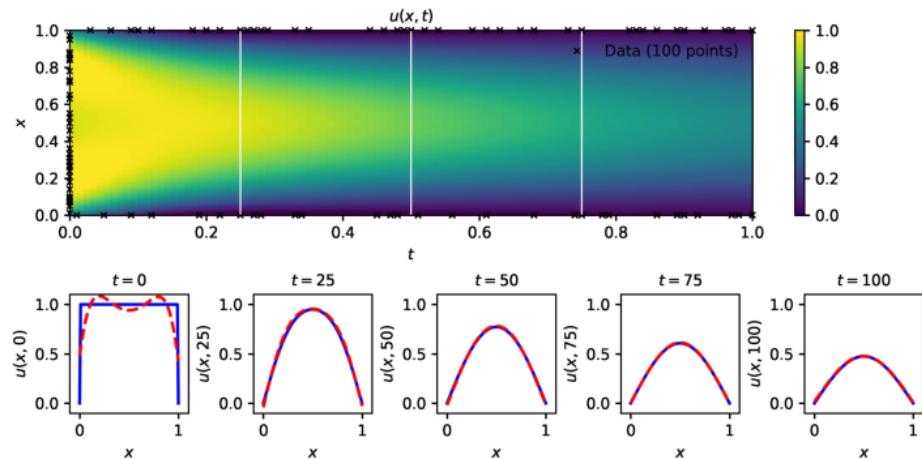
for ax in axs:
    ax.axis("off")

plt.tight_layout()
```

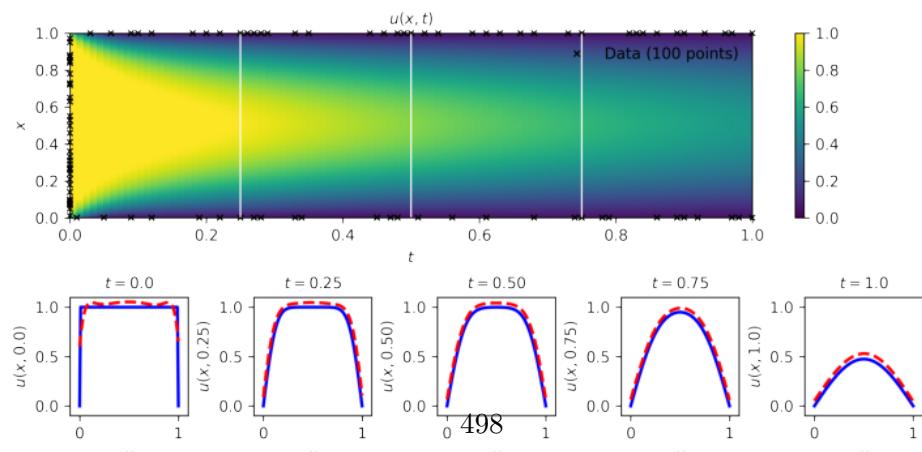
300 Epochs



1000 Epochs



5000 Epochs



The images in the bottom row depict time slices, marked as white lines in the field plot. In the first row, which is the result after around 300 epochs of training, you see that the network quickly learned to approximate a single cosine going through the beam. This does not approximate the initial condition on the left well at all. The lower image shows the training after 1000 epochs. Here, the network learned another mode with a *higher frequency*, like two cosines added up in a way that doesn't disrupt the rest of the field. It did take a very long time compared to learning the first mode with a *lower frequency*. This problem is called **spectral bias**.

In principle, given infinite time, a PINN will learn all frequencies in a dataset, given that its capacity is sufficiently high. But we don't have infinite time available. One way to help is to use **Fourier features** which the dataset is transformed to. The way the PINN tries to solve the PDE is similar to what a *Fourier transform* does, if trigonometric activation functions are used.

```

import numpy as np
import mpmath
from ipywidgets import interact

@np.vectorize
def f(t):
    if abs(t) < 0.5:
        return 1
    else:
        return 0

t0 = -1
P = 2

tt = np.linspace(t0, t0+P, 200)
t2d = tt[np.newaxis]

def plot_box_fourier(N=1, plot_cosines=False, cut_off=False):
    cs = mpmath.fourier(f, [t0, t0+P], N)

    cos_coeffs = cs[0]

    an = np.array(cos_coeffs)[:, np.newaxis]
    n = np.arange(N+1)[:, np.newaxis]

    coses = an*np.cos(2*np.pi*n*t2d/P)

    plt.figure(figsize=(12,7))

    plt.plot(tt, f(tt), lw=6, label="Ground truth")

    #plt.axhline(constant, color='blue', alpha=0.7)

```

```

plt.plot(tt, sum(coses), lw=4, label=f"Fourier approximation {N}")

if plot_cosines:
    #plt.plot(tt, coses.T, color='blue', alpha=0.5)
    plt.plot(tt, coses[0].T, alpha=0.4, label=r"0 mode, $\nu = 0$")
    for i in range(1,N+1,2):
        plt.plot(tt, coses[i].T, alpha=0.4, label=fr"${i}$ mode, $\nu = ${n[i//2+1]*t2d[:,i//2+1]/P}")

if cut_off:
    plt.gca().axis("square")
    plt.xlim([-0.51, 0.51])
    plt.ylim([-0.01, 1.25])

plt.legend(fontsize=12, framealpha=0.6)

plt.savefig("img/plot_box_fourier.png")

interact(plot_box_fourier, N=(0,7), plot_cosines=False, cut_off=False)

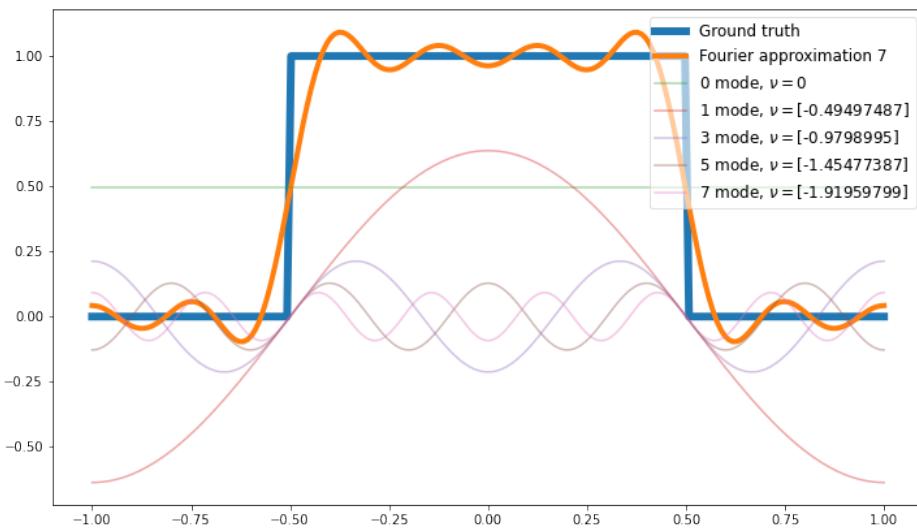
```

interactive(children=(IntSlider(value=1, description='N', max=7), Checkbox(value=False, description='plot_cosines'), Checkbox(value=False, description='cut_off')))

```

<function __main__.plot_box_fourier(N=1, plot_cosines=False, cut_off=False)>

```



The more terms are used, the better the approximation, and each term has a higher frequency than the first.

TensorFlow does not directly offer well-controllable Fourier layers, but there is an experimental `RandomFourierFeatures` layer you can introduce into your PINN like this:

```
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers.experimental import RandomFourierFeatures

model = Sequential()

model.add(Dense(4))
# laplacian instead of gaussian is also possible for appropriate PDEs
model.add(RandomFourierFeatures(output_dim=1024, kernel_initializer="gaussian"))
model.add(Dense(16, activation="tanh"))
model.add(Dense(1, activation=None))
```

This will generate random Fourier features from the input. You can also add the layer after hidden layers to emulate hidden further Fourier transforms. The layer also has a scale parameter, which you can either set yourself or set to trainable (see adaptive activations further below).

The problem with this approach to PINNs is that the data must be as noise-free as possible, since noise is also present as high-frequency information in the data. Recall the images from lecture 01 about overfitting, where the high order polynomials tended towards fitting the noise instead of the actual ground truth. The same would happen here.

15.3.2 Vanishing Gradients

Especially in deep networks, *vanishing gradients* can cause the first layers in a network to not learn anything. We saw how this happened in recurrent neural networks in lecture 05, but it happens in all deep ANNs. For RNNs, we used different architectures to provide the gradient a second pathway to propagate back through. Another way to make sure the gradients are propagated back more robustly is to use **skip/residual connections**, like in the *U-Net* from lecture 04.

The easiest ways to implement skip connections is to use keras' functional API:

```
from tensorflow.keras import Input, Model
from tensorflow.keras.layers import Dense, add
from tensorflow.keras.utils import plot_model

inputs = Input(shape=(None,3))

x = Dense(8, activation="relu")(inputs)
block_1_output = Dense(16, activation="relu")(x)

x = Dense(32, activation="relu")(block_1_output)
x = Dense(16, activation="relu")(x)
```

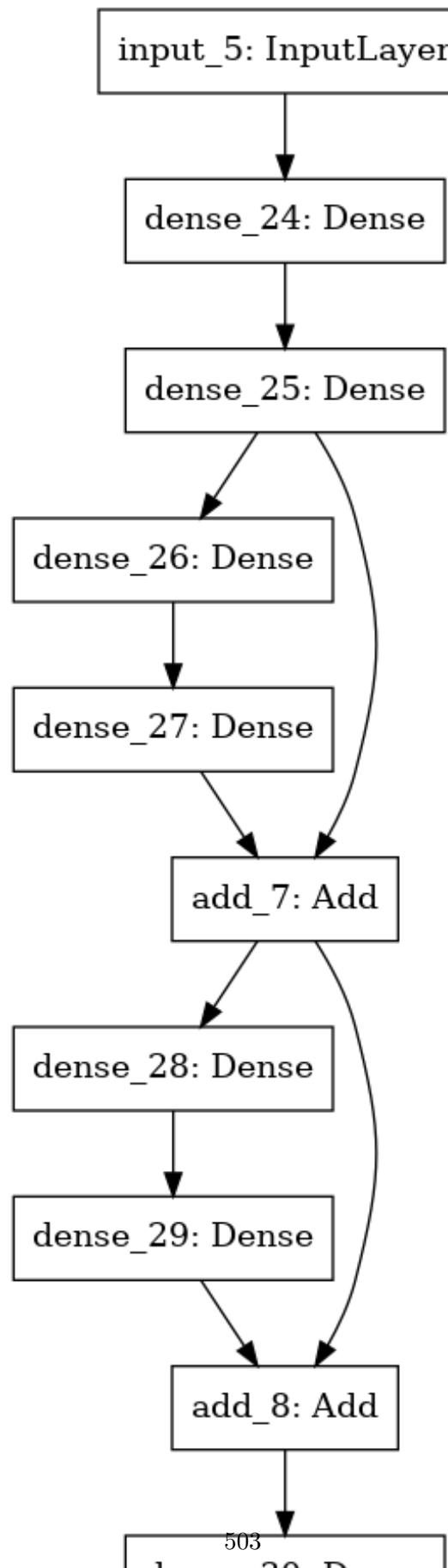
```
block_2_output = add([x, block_1_output])

x = Dense(32, activation="relu")(block_2_output)
x = Dense(16, activation="relu")(x)
block_3_output = add([x, block_2_output])

outputs = Dense(10)(block_3_output)

model = Model(inputs, outputs, name="toy_resnet")
```

```
plot_model(model)
```



These skip connections *skip* layers during backpropagation, so that the gradient is still large enough to cause noticeable change for the weights.

Optimization Optimizing PINNs is notoriously difficult, since its mostly driven by the regularization terms. So far, we mostly used the first order `adam` optimizer, which scales well, but suffers from the usual problem of stochastic gradient descent variants - it ignores information about the local curvature of the loss landscape.

Second order optimizers scale much less well to larger problems, but they converge *much* faster and are much less prone to getting stuck in local minima. Quasi-Newton methods belong to this class, of which **L-BFGS** is a commonly used and well-studied example. The **L** implies “limited memory” requirements, since it doesn’t compute the whole Hessian matrix, which is used to gather information about local curvature by computing its eigenvalues. Instead, it *approximates* it.

Unfortunately, it’s not implemented in TensorFlow natively (yet), so it’s not straightforward to use for optimizing ANNs. There are currently two options; using the `scikit-learn` implementation as described, e.g., [here](#), or using the implementation from `tensorflow_probability`, as explained [here](#).

Adaptive Activations We did discuss parameters in activation function a bit in lecture 05. E.g., the sigmoid activation function has a β parameter that controls the “stretch”, leaky ReLU has a parameter that controls the slope of the negative part, ELU has two parameters controlling the exponential decay towards negative infinity and so on. In principle, these parameters can be added to the list of weights and learned by backpropagation.

The effects are discussed in [Jagtap et al.](#), where the following image comes from:

```
from IPython.display import Image
Image("img/adaptive_activations.png")
```

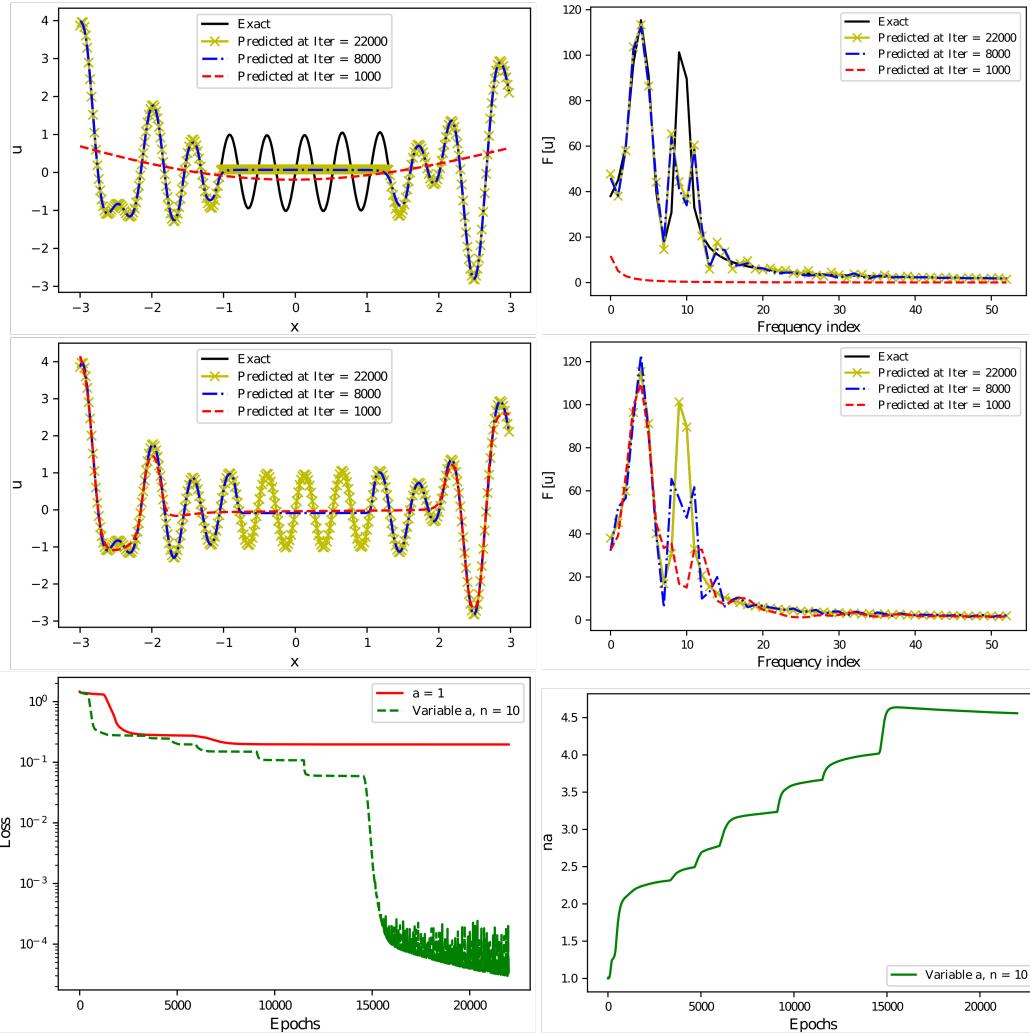


Figure 3: Smooth function: Neural network solution of $f(x)$ (top row) and adaptive activation (middle row) functions. First column (top and middle row) shows the solution, which is also plotted in frequency domain (zoomed-view) as shown by the corresponding second column. Bottom row shows loss function comparison for $f(x)$ and adaptive activation (left) and variation in a with $n = 10$ (right).

The left column shows an example field that was learned by a PINN. The first row with standard activations, the second row with adaptive activation functions, where the function parameters are learned through backpropagation. The right column shows the corresponding frequency spectrum. Adaptive activations seem to have an edge especially in high-frequency situations. This comes at the cost of slightly longer training times, so as a compromise, adaptive activations are usually learned *layer-wise*, meaning that the neurons in a single layer all use the same parameter(s).

Further Work Of course this is not the end all solution to all problems, and PINNs are under active research since they were only published 1-2 years ago (the preprint was uploaded 2017, but it was immensely difficult for the researchers to get their paper accepted. Kardianakis mentioned on a conference that it was difficult for the reviewers to believe that the approach was that simple yet so powerful). Right now, there is a whole “alphabet” of PINNs for different purposes, e.g. UQPINNs for uncertainty quantification, or stochastic sPINNs, Bayesian BPINNs, conservative cPINNs, and

so on.

15.4 Noether's Theorem

There are four main techniques for incorporating knowledge about a problem into ML models:

- Soft constraints: *Physics-informed Neural Networks* with PDE-regularization
 - Rather easy to incorporate
 - Generalizable (but not generalizing)
 - Difficult to optimize
 - No guarantees!
- Hard constraints: develop custom neurons/layers/units
 - “Easier” optimization
 - Guaranteed physics-obedient
 - Needs manual and tedious work
- Physics framework enhancement
 - Learn empirical or difficult equations
 - Physics conserved by the framework, difficult stuff handled by ML
- Incorporate physical symmetries directly into models
 - Symmetry - conserved quantity relationship by *Noether's Theorem*

Here, we'll concentrate on the last approach.

There's a mathematical formalism for incorporating symmetry into networks. **Noether's Theorem** relates *conserved quantities* to *symmetries* of a problem. For example, the symmetry of our spacetime under translation (at least in Newtonian physics) is the cause of momentum conservation, and the symmetry under translation in time causes energy to be conserved, isotropy causes angular momentum conservation. You can think of the consequences like this: no matter where you perform an experiment, it will yield the same result under the same conditions. No matter when you perform it, it will yield the same result. No matter where you point it to, the result stays the same.

The theory behind Noether's theorem is quite longish, so we won't go into too much detail and assume some prior knowledge here. The main ingredients are Lagrangian mechanics and variational calculus. The underlying principle is the *principle of least action*:

$$S[x] = \int_{t_1}^{t_2} \mathcal{L}(t, x(t), \dot{x}(t)) dt$$

Physical trajectory $x(t)$ minimizes action \rightarrow Euler-Lagrange equations

$$\frac{d}{dt} \frac{\partial \mathcal{L}}{\partial \dot{x}} = \frac{\partial \mathcal{L}}{\partial x}$$

Example: particle in gravity

$$\mathcal{L} = \frac{1}{2} m \dot{x}^2 - V(x)$$

yields

$$m\ddot{x} = -\nabla V(x)$$

Some more setup:

- *Group*: Set G with $\circ : G \times G$, associative and neutral element $1 \in G$
- *Group action*: $\cdot : G \times S \rightarrow S$, compatible with \circ $x \in S$ are *group representations*
- *Invariance*: $f(g \cdot x) = f(x) \quad \forall x \in X, g \in G$
- *Equivariance*: $f(g \cdot x) = g \cdot f(x) \quad \forall x \in X, g \in G$

So this establishes invariance and equivariance. Finding solutions to PDEs is made much simpler by identifying *symmetry groups* of a PDE. Take for example the wave equation:

$$\frac{\partial u}{\partial t^2} = c^2 \frac{\partial u}{\partial x^2}$$

$$\begin{aligned} f(x, t) &= \cos(kx + \omega t) \\ g(x, t) &= \sin(kx + \omega t) \\ \text{plus trivial solutions} \end{aligned}$$

Full solution space:

$$u(x, t) = Af(x, t) + Bg(x, t) + \text{trivial solutions}$$

Trivial solutions in this case are second or lower order polynomials and constants. PINNs can also suffer from *mode collapse*, such that they learn a trivial solution to a PDE without being able to recover from it. This often leads to PINNs predicting a constant value for the whole domain.

By knowing the symmetries of the PDE, we could easily construct the whole solution space of the wave equation.

- Induced action of g on solution space: $g \cdot u = g \cdot u(g^{-1} \cdot x, g^{-1} \cdot t)$
- G is a symmetry group of a differential operator D if G preserves $\text{sol}(D)$, i.e. $g \cdot \varphi(x) \in \text{sol}(D)$ if $\varphi(x) \in \text{sol}(D)$
- Evolution forecasting: $f(w_{t-k}, \dots, w_{t-1}) = w_t$ If G is a symmetry group of the associated D :

$$f(g \cdot w_{t-k}, \dots, g \cdot w_{t-1}) = g \cdot w_t$$

and f is G -equivariant.

Finally, we can come to Noether's theorem:

- Some one-parameter transformation T_s maps solutions $x(t)$ of a PDE to other solutions $x(s, t, T)$
- Infinitesimal transformation: $\delta x(t, x, \dot{x}) = \frac{\partial}{\partial s} x(s, t, T)|_{s=0}$, e.g. $\delta x = \dot{x}$ and $\delta x = d$ for temporal and spatial translations
- T_s is a symmetry of the action iff $\frac{\partial}{\partial s} \mathcal{L}|_{s=0} = \frac{d}{dt} K(t, x)$, since then the action only differs by boundary terms
- $\frac{\partial}{\partial s} \mathcal{L}|_{s=0} = \delta x \frac{\partial}{\partial x} \mathcal{L} + \frac{d \delta x}{dt} \frac{\partial}{\partial \dot{x}} \mathcal{L}$

- $\delta x \left(\frac{\partial}{\partial x} \mathcal{L} - \frac{d}{dt} \frac{\partial}{\partial \dot{x}} \mathcal{L} \right) + \frac{d}{dt} \left(\delta x \frac{\partial}{\partial \dot{x}} \mathcal{L} - K \right) = 0$ after some rearrangement
- Multiples of the E-L equation vanish, so: $Q = \delta x \frac{\partial}{\partial \dot{x}} \mathcal{L} - K$ is a conserved quantity, the *Noether charge* corresponding to T_s

Gravity example:

$$\dot{x} \frac{\partial}{\partial \dot{x}} \left(\frac{1}{2} m \dot{x}^2 - V(x) - K \right) = m \dot{x}^2 - \left(\frac{1}{2} m \dot{x}^2 - V(x) \right) = \frac{1}{2} m \dot{x}^2 + V(x) \rightarrow \text{energy}$$

An example for such a one-parametric transformation are simple translations, where the parameter describes the distance (and direction) of the translation, or rotations with an angle. The example for infinitesimal transformations can be understood as Taylor expansions. The first term will be a 1, i.e., nothing happening. The second term is the infinitesimal transformation, just like the second term in the Taylor expansion describes the linear neighborhood around a point on a curve.

There are many more examples where Noether's theorem generates conserved quantities that can be used to derive equations of motion, or simplify PDEs or learn something about the physics of a problem. In the next lesson, we'll look at an example application in the form of *equivariant neural networks*.

15.5 Equivariant Neural Networks

As an example for an application of Noether's theorem from the last lesson, we'll take a brief look at what [Kashinath et al.](#) did for weather modeling. The underlying theory is described by the *Navier-Stokes equations* with the following symmetries:

$$\frac{\partial w}{\partial t} = -(w \nabla) w - \frac{1}{\rho_0} \nabla p + \nu \Delta w + f; \quad \nabla w = 0; \quad \frac{\partial H}{\partial t} = \kappa \Delta H - (w \nabla) H$$

- Spatial translation: $T_d^{\text{sp}} = w(x - d, t)$
- Temporal translation: $T_{\tau}^{\text{temp}} = w(x, t - \tau)$
- Uniform motion: $T_d^{\text{um}} = w(x, t) + d$
- Rotation/Reflection: $T_R^{\text{rot}} = R w(R^{-1}x, t), R \in O(2)$
- Scaling: $T_{\lambda}^{\text{scale}} = \lambda w(\lambda x, \lambda^2 t), \lambda \in \mathbb{R}^+$

For the modeling, they used a CNN with residual connections. The key idea here is that a *composition* of equivariant functions is again equivariant.

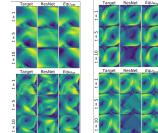
A CNN is G -equivariant iff

$$K(gv) = \rho_{\text{out}}^{-1}(g) K(v) \rho_{\text{in}} \quad \forall g \in G$$

- This implies both linear transformations and activations must be equivariant
- Vector-valued representations of G instead of scalar weights
- *Spatial and temporal equivariance*: given in CNNs
- *Rotational equivariance*: expand kernel as Fourier series and apply constraint above, then projecting onto certain harmonics \rightarrow most coefficients vanish
- *Uniform motion equivariance*: “faked” by mean shifts in each block in each layer
- *Scaling equivariance*: MinMaxScaler for each block in each layer (resolution-independent), or scale x, t, μ in each convolution operation (resolution-dependent)

ρ_{in} and ρ_{out} are matrices that map the group elements to the appropriate positions for the kernels K in each layer. Let's take a look at a few example simulations:

```
from IPython.display import Image
Image("img/equ_flows.png")
```



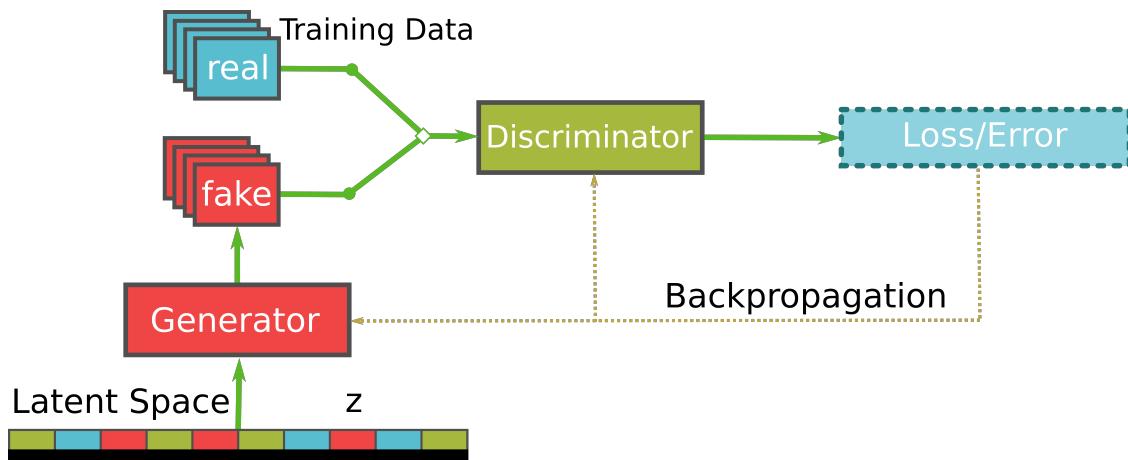
In each image, only a single equivariance technique was employed, but the results show that prediction quality improves a lot with each extra effort taken to include physical knowledge.

16 Checkpoint II

16.1 Generative Modeling

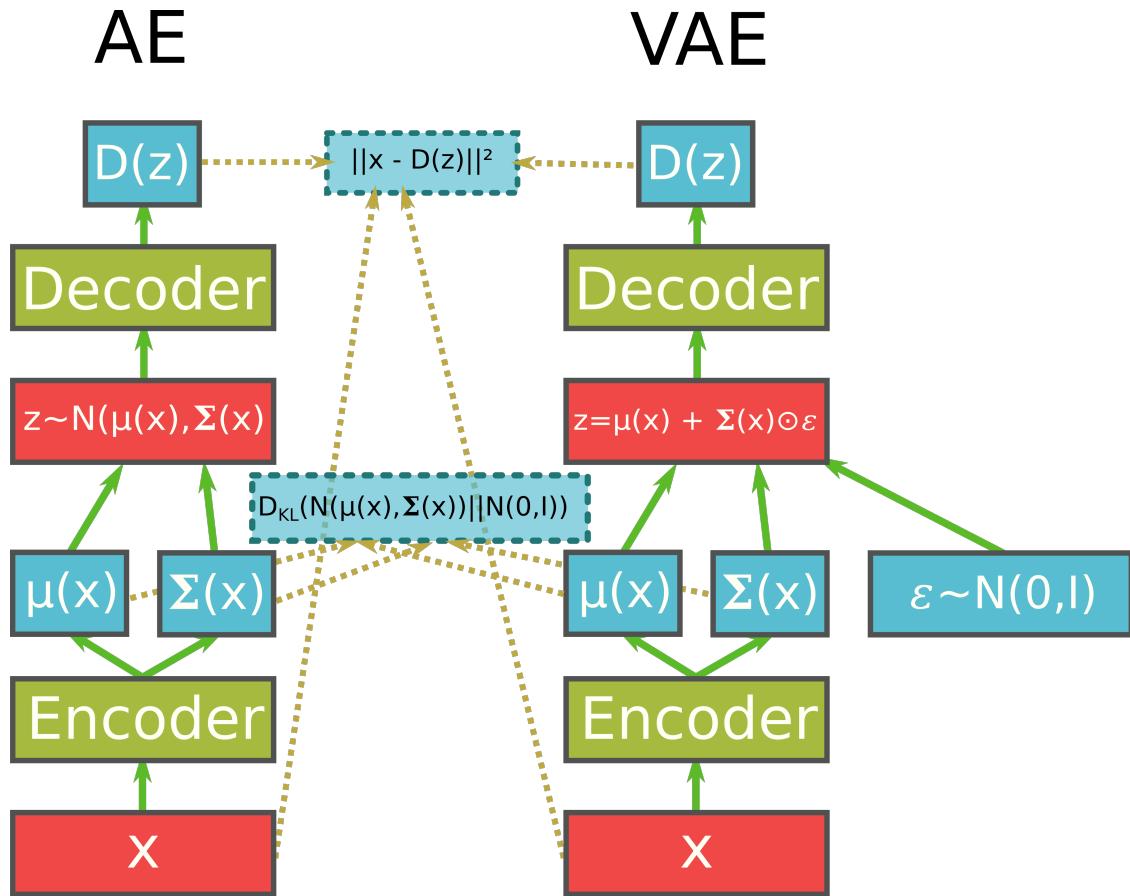
We saw how machine learning models could also be used to *generate* new data, designs, ideas by either explicitly (autoencoders) or implicitly (generative adversarial networks) modeling probability distributions. Autoencoders reduce the dimensionality of information into a *latent space*, from which they then sample. GANs directly sample from a random latent space. In the latter case, adjacent latent space vectors produce meaningfully similar outputs.

```
from IPython.display import Image
Image("img/gan_sketch.png", width=700)
```



A slight modification for autoencoders makes them *variational*, such that the latent space produces parameters for a probability distribution from which the decoder part then samples:

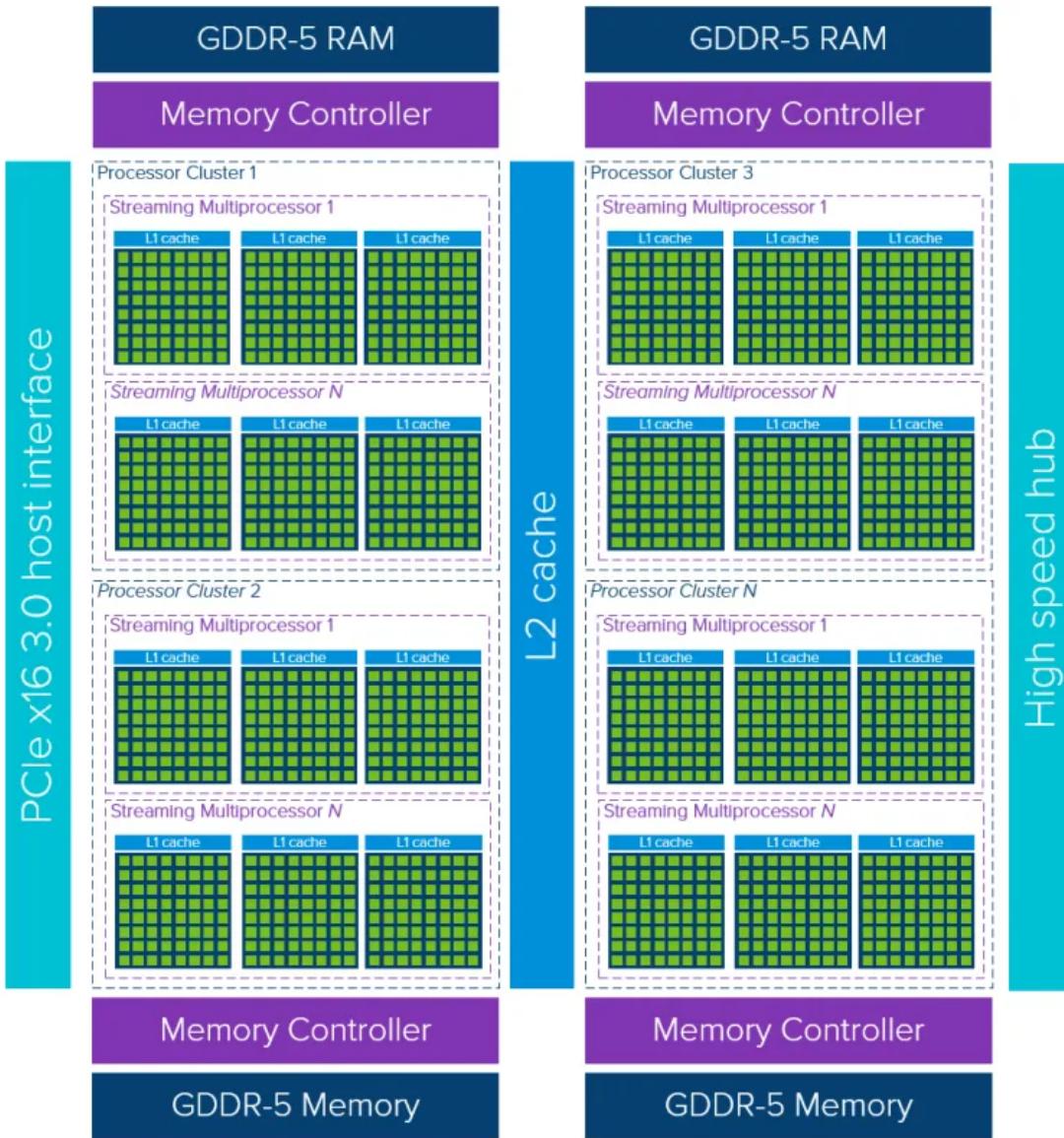
Image("img/vae_sketch.png", width=700)



16.2 ML Hard- and Software

Machine learning reached a size especially in science and engineering where specialized hardware is becoming almost necessary for industrial problems. HPC centers offer at least parallelized CPU solutions, but often additionally offer *accelerators* like GPUs or special ML accelerators. GPU usage is straightforward with most ML frameworks, as they abstract all bureaucracy away from the end user. GPUs offer stream multiprocessors, which are still “general purpose”, so not specialized to ML tasks:

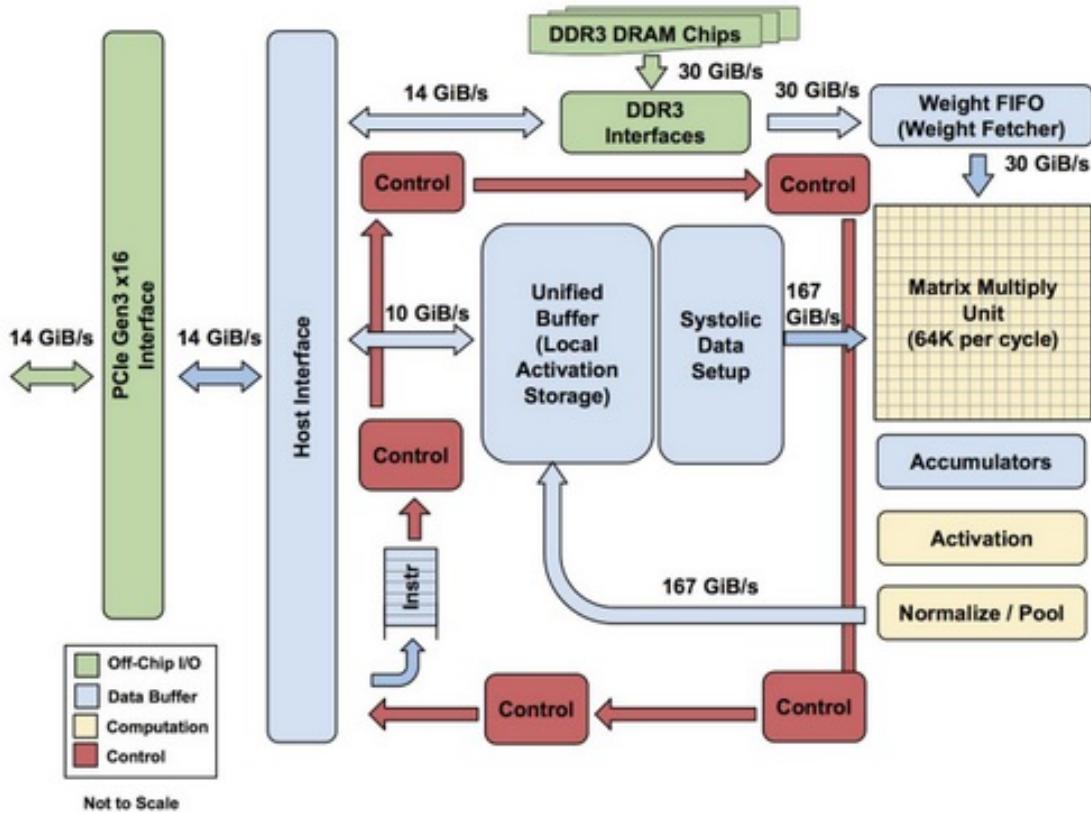
Image("img/gpu.png")



Better performance per price and energy tag can be achieved with FPGAs, which are specially programmed for certain tasks, or with ASICs, which only perform a single task but do so exceptionally well. A famous example are Google's TPUs. Pretty much all ML accelerators concentrate on multiply-accumulate operations (MACs) and bandwidth for getting data onto the device.

There are more things coming up like neuromorphic computing chips.

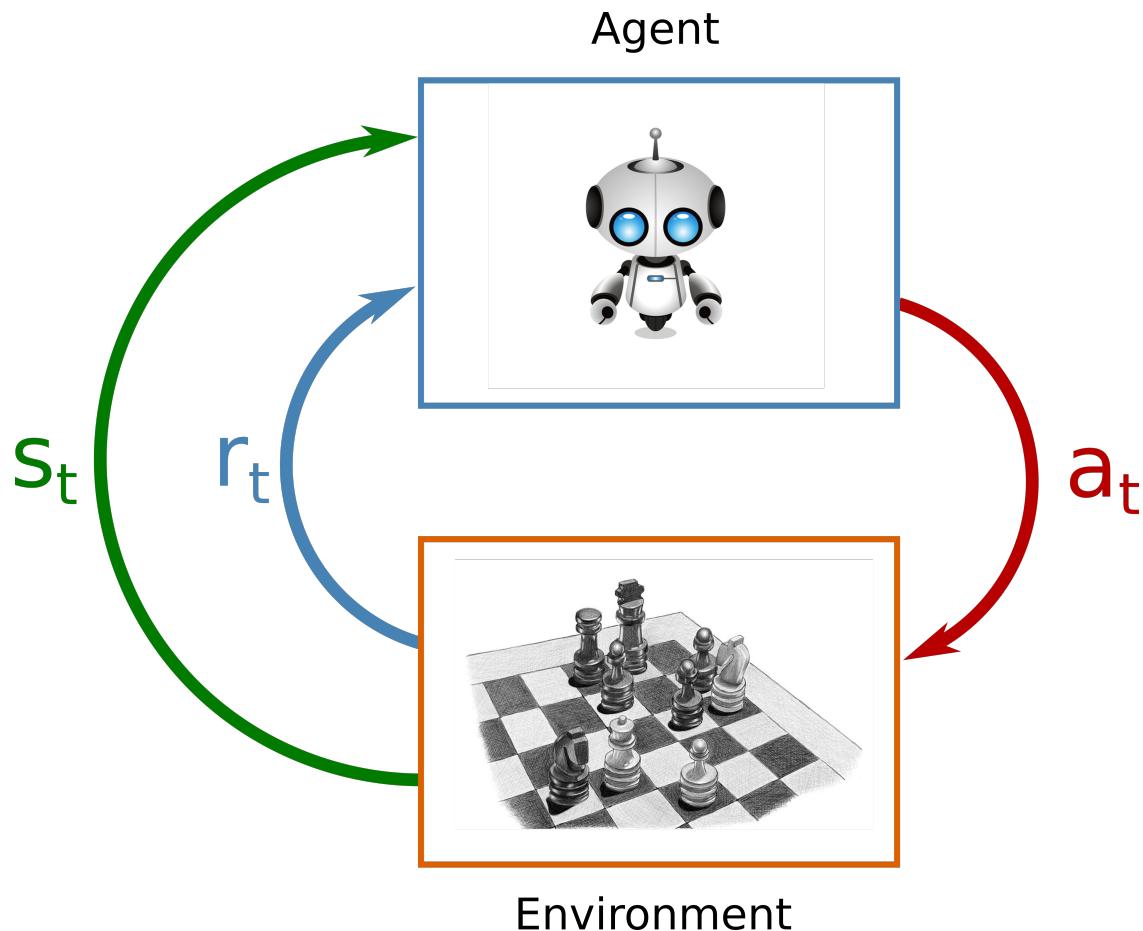
Image("img/tpu.jpg")



16.3 Reinforcement Learning

The idea of reinforcement learning is quite different from supervised and unsupervised learning, which were the main part of this course. In reinforcement learning an *agent* interacts with an *environment*, where the agent is informed about the current environment state, performs some action using this information and receives a reward for that action from the environment. Parts of this model that is mathematically captured by Markov decision processes can be approximated with ANNs.

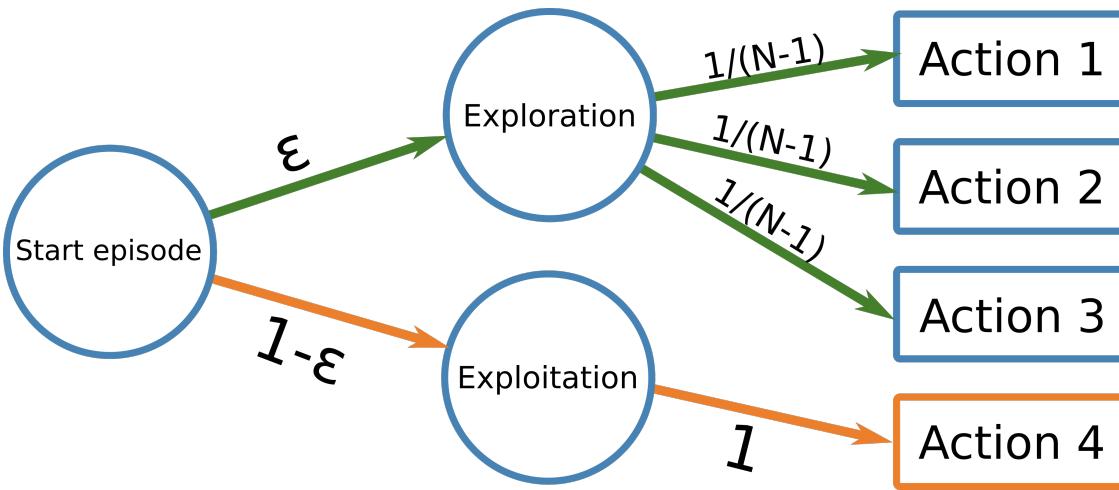
```
Image("img/r1.png", width=700)
```



There are various approaches for performing reinforcement learning with ANNs. We talked about Q -learning, policy gradients, and the actor-critic model, all with distinct advantages and disadvantages. An important point was the *exploration* capability of an agent. If it always chooses the current best action, it will get stuck in local minima and never find the optimal policy for the given environment.

One way to enforce exploration is to use an ε -greedy approach:

```
Image("img/exploration_vs_exploitation.png", width=700)
```



17 Where to Go from Here

So, we've come to the end of the course. Time for a few final remarks.

17.1 Papers

The idea of this course was twofold: firstly to give students some hands-on experience with machine learning techniques, and secondly to enable students to read ML papers, especially on current topics. This would be a good step forward. Search for ML techniques in a field that interests you and simply start reading papers. This will give you an idea of current developments in the field, but also an idea of what's important and how to communicate ideas.

17.2 Online Info

You probably already noticed this, but there are countless blogs and blog entries on all topics we've covered in the course and then some. Although a lot of them aren't rigorous and some are outright wrong, they can help in understanding the gist of ideas without having to read 10 papers on the topic.

Another important point is **inspiration**, i.e., making you think abstractly about machine learning and problems and develop the creativity to find and implement novel approaches. Apart from blog posts, YouTube videos are a great source for inspiration. Some commonly mentioned channels are Steve Brunton mainly for CFD, Two Minute Papers mainly for graphics, but also general developments, Yannik Kilcher for pretty much everything ML, but there are many more. These days more often than not conference presentation are recorded and put online, sometimes even along with resources to try on and learn from.

17.3 Further Techniques

Most methods we discussed aren't very new. In fact, most of them have been developed up to decades ago. They just experienced a second spring thanks to extreme improvements in computing power per price. The field itself develops so quickly that it's extremely difficult, if not impossible, to

keep up completely. Some further or cutting edge techniques worth checking out are *Graph Neural Networks*, especially for deformation invariance, *Transformers* and generally *attention mechanisms*, mostly used for language processing but otherwise better forms of RNNs, *TabNets* for tabular data, various improvements to *reinforcement learning* and more. *Physics-informed modeling* is actively discussed and developed right now, with varying degree of success. The next trend is *quantum computing*, where synergies with ML are already explored in *quantum machine learning*.

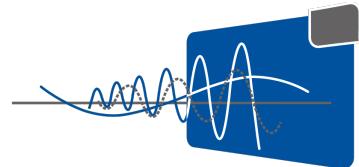
PINNs can be run “in reverse” so that they learn PDEs from data. The form of the PDE has to be known, so only parameters are learned. The form can be made arbitrarily complex though. A promising technique for learning PDEs from data is the *AI Feynman*, where the idea is to express formulae in polish notation (like “ $*AB$ ” would mean $A*B$) and impose a “linguistic regularization” that penalizes long expressions. No new physics has been found this way yet, but the concept is constantly improving and the examples quite impressive.

17.4 Current Developments

It also helps to follow along current developments outside the science and engineering world. Check out for example *GPT-3* and what it can do, or Microsoft’s prototype of the *github copilot* based on it. How is machine learning used in the financial industry? How is it used to determine whether someone is credit-worthy or not? How does spam detection work, and what about facial (or today rather individual, since not only images are used but also behavioral analysis) recognition? How is it used in robotics and which advancements were made thanks to ML? What about medicine? It’s generally good to be aware of these things, since ML has quite a lot of disadvantages and not every product advertised as “empowered by AI” is actually well-designed or sensible.

17.5 Create Your Own Solutions

Finally, build your own ML applications! These days, employers are less interested in numbers on a piece of paper and more in what you have done. This includes the final project for this lecture, but you’re highly encouraged to try your own things, especially outside of science and engineering. A nice way to present such work is using *github*, where you can also host a smallish website as a blog, where you can talk about your code and explain your ideas. These days it’s an essential part of a good portfolio. Of course it depends on the job or project you’re applying for, and how “old” the company or team is, but it can never hurt to have the experience of creating, developing, and presenting your own ideas.



FIN