

# Numerical Methods for Algorithmic Systems and Neural Networks (NumASNN)

Sebastian Kinnewig, Leon Kolditz, Julian Roth, Thomas Wick

<https://www.ifam.uni-hannover.de/en/wick/>

<https://www.ifam.uni-hannover.de/en/>

Winter Semester 2021/2022

Last update: Mar 21, 2022

DOI: <http://dx.doi.org/10.15488/11897>



Institut für  
Angewandte Mathematik

# Table of Contents

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Why this class?

- Several classes of related nature here in Hannover; L3S<sup>1</sup> Faculty of Electrical Engineering and Computer Science<sup>2</sup>, ...
- **Why this class in maths?**
- **Basic ingredients** of AS (algorithmic systems), ML (machine learning), NN (neural networks) are **based on mathematics**: Numerik 1, Optimization, Statistics
- Rigorous **formalizations** and analysis of algorithms in terms of accuracy, efficiency, robustness are **key items in numerics**
- **One goal** of this class: Using neural networks with or for differential equations.

---

<sup>1</sup><https://www.l3s.de/en>

<sup>2</sup><https://www.et-inf.uni-hannover.de/>

# Structure of this class

- 2+1 lecture in the German system:
    - 90 minutes lecture per week
    - 90 minutes exercise every 2nd week (three classical exercises of all students in the computer pool; then individual discussions within group projects)
  - Goal/exam:
    - Project work in groups of 2 to 4 students
    - Mixture of **advanced projects** including differential equations and model order reduction as well as projects to substantiate **basic understanding** of this class
- choices depend on motivation and personal background
- Final presentation of project work including questions and discussion (approx. 30 – 45 minutes) in Feb/Mar 2022

# How to study in this class?

- The audience is **heterogeneous**, but natural for such a class
- L1-L4 are basics in machine learning, which participants specifically from Computer Science may have already had in other lectures
- Same for L5-L10 in artificial neural networks and their modern extensions
- On the other hand, students from maths may have had L11-L14 (differential equations)
- Content allows flexible handling of project work (starting earlier or later according to personal wishes)

# Prerequisites and preliminary steps

- Introduction to numerical analysis ([Numerik 1](#) at LUH), [Analysis I+II](#), [Lineare Algebra I+II](#)
- Not mandatory, but useful: [functional analysis \(FA\)](#)
- python<sup>3</sup>
- In Exercise 0, brief installation instructions to [python](#) will be given; as well as a brief introduction to [git](#)
- In Exercise 1, there will be an alternative exercise sheet with first steps into python for students that are unfamiliar with this programming language

---

<sup>3</sup><https://www.python.org/>

## Software (selection)

- 1 TensorFlow <https://www.tensorflow.org/>
- 2 PyTorch <https://pytorch.org/>
- 3 Keras <https://keras.io/>
- 4 scikit learn <https://scikit-learn.org/stable/>

Background in python:

- 1 Python: <https://www.python.org/>
- 2 Tutorial: <https://docs.python.org/3/tutorial/index.html>

## 1. Traditional AI

1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation

1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics

1.3 Lecture 3: Fundamental Algorithms

1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

2.1 Lecture 5: Artificial Neural Networks (ANN)

2.2 Lecture 6: Universal Approximation Theorem

2.3 Lecture 7: Convolutional Neural Networks (CNN)

2.4 Lecture 8: Recurrent Neural Networks (RNN)

2.5 Lecture 9: Transformer

2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

3.1 Lecture 11: Introduction to ML for Scientific Computing

3.2 Lecture 12: Neural ODE

3.3 Lecture 13: PINNs: Physics-Informed Neural Networks

3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects



# Outline lecture 1

- Definition of algorithmic systems and applications
- Numerical concepts and examples
- Three basic paradigms in learning

## Algorithmische Systeme (Definition) - in german

- DEK: Datenethikkommission
- Gutachten (report), October 2019
- [Full report \(pdf\)](#)<sup>4</sup>
- Please see in particular pages 14-15 for general, ethical and juridical remarks
- Information on **Algorithmische Systeme** on page 24ff
- 'Control of algorithms'
- General requirements for algorithmic systems and interaction between human beings and machines (algorithms):
  - **Algorithm-based** decisions (human decision based on input from algorithm)
  - **Algorithm-driven** decisions (algorithm proposes information on which human as little means for an independent decision)
  - **Algorithm-determined** decisions (algorithm proposes decision without human interactions)

---

<sup>4</sup>Brown-colored text will open a link.

# Abbreviations

- AS = algorithmic systems
- AI = artificial intelligence
- ML = machine learning
- ANN = artificial neural networks

## Further applications in other disciplines

- As previously said, we are interested in mathematical-numerical aspects in this class and applications to differential equations
- For a motivation in other fields, please see some of the following talks:  
[https://www.dhvseminare.de/symposium\\_2019](https://www.dhvseminare.de/symposium_2019)
- Therein, various views from different fields (neurosciences, computer science, ethics, economics, law) were presented.

# Algorithms (I)

- Design and analysis of algorithms is a **key branch** of **numerical mathematics** and **scientific computing**
- **Scientific computing:**
  - Mathematical modeling of practical problems (physics, engineering, finance, ...)
  - Design and analysis of algorithms
  - Research software development
- Feedback-loop of all three items
- Mathematics become experimental
- Design and analysis of algorithms can be broadly achieved with the help of **seven concepts**

# Algorithms (II)

## Definition (Algorithm)

An algorithm is an instruction for a schematic solution of a mathematical problem statement. The main purpose of an algorithm is to formulate a scheme that can be implemented into a computer to carry out so-called numerical simulations. Direct schemes solve the given problem up to round-off errors (for instance Gaussian elimination). Iterative schemes approximate the solution up to a certain accuracy (for instance Richardson iteration for solving linear equation systems, fixed-point iterations, gradient descent, Newton's method, ...). Algorithms differ in terms of **accuracy**, **robustness**, and **efficiency**.

## Seven concepts of numerical mathematics <sup>5</sup>

- 1 **Approximation:** since analytical solutions are often not possible to achieve as we just learned in the previous section, solutions are obtained by **numerical approximations**.
- 2 **Convergence:** is a qualitative expression that tells us when members  $a_n$  of a sequence  $(a_n)_{n \in \mathbb{N}}$  are sufficiently close to a limit  $a$ . In numerical mathematics this limit is often the solution that we are looking for.
- 3 **Order of convergence:** While in analysis, we are often interested in the convergence itself, in numerical mathematics we must pay attention how long it takes until a numerical solution has sufficient accuracy. The longer a simulation takes, the more time and more energy (electricity to run the computer, air conditioning of servers, etc.) are consumed. In order to judge whether a algorithm is fast or not we have to determine the order of convergence.

---

<sup>5</sup>T. Richter, T. Wick; Einführung in die numerische Mathematik - Begriffe, Konzepte und zahlreiche Anwendungsbeispiele, Springer, 2017

# Numerical concepts

- 4 **Errors:** Numerical mathematics can be considered as the branch 'mathematics of errors'. What does this mean? Numerical modeling is not wrong, inexact or non-precise! Since we cut sequences after a final number of steps or accept sufficiently accurate solutions obtained from our software, we need to say *how well the (unknown) exact solution by this numerical solution is approximated*. In other words, we need to determine the error, which can arise in various forms as we discussed in the previous section.

**Exercise:** Make a list of errors that may arise in numerical mathematics

- 5 **Error estimation:** This is one of the biggest branches in numerical mathematics. We need to derive error formulae to judge the outcome of our numerical simulations and to measure the difference of the numerical solution and the (unknown) exact solution in a certain norm.



# Numerical concepts

- 6 Efficiency:** In general we can say, the higher the convergence order of an algorithm is, the more efficient the algorithm is. Therefore, we obtain faster the numerical solution to a given problem. But numerical efficiency is not automatically related to resource-effective computing. For instance, developing a parallel code using MPI (message passing interface), hardware-optimization (CPU,GPU), software optimizations (ordering in some optimal way for-loops, arithmetic evaluations, etc.) can further reduce computational costs.
- 7 Stability:** Lastly, the robustness of algorithms and implementations with respect to parameter (model, material, numerical) variations, boundary conditions, initial conditions, uncertainties must be studied. Stability relates in the broadest sense to the third condition of Hadamard.

## Examples of previous concepts

- For the following examples, we have the **clothesline problem**<sup>6</sup> in mind
- This is a **differential equation**
- Problem statement:

$$-u''(x) = f \quad \text{in } (a, b) \quad (\text{Differential equation}) \quad (1)$$

$$u(a) = u_l \quad (\text{Left boundary condition, Dirichlet}) \quad (2)$$

$$u(b) = u_r \quad (\text{Right boundary condition, Dirichlet}) \quad (3)$$

where  $f, u_l, u_r \in \mathbb{R}$ .

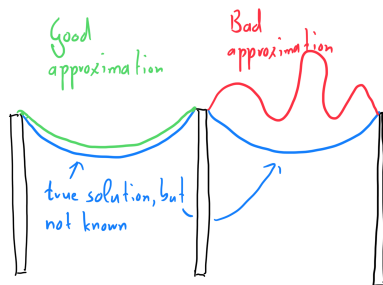
---

<sup>6</sup>T. Wick. *Numerical methods for partial differential equations*. Hannover : Institutionelles Repositorium der Leibniz Universität Hannover, DOI: <https://doi.org/10.15488/11709>. Jan. 2022. DOI: <https://doi.org/10.15488/11709>.

## Example: Approximation

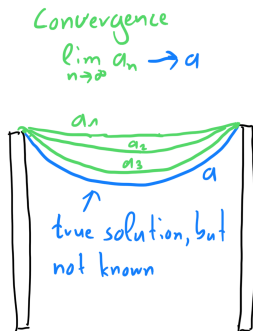
**Approximation:** Two approximations of the clothesline problem:

2022



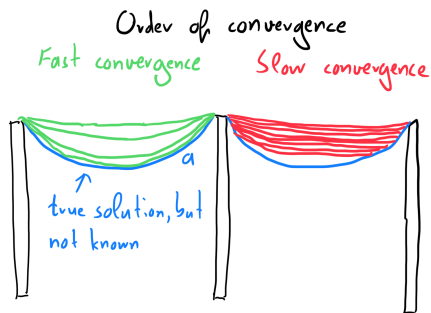
# Example: Convergence

**Convergence:** Converging approximations of the clothesline problem:



## Example: Convergence order

**Order of convergence:** Two different speeds of convergence:



## Example: Errors

**Errors:** Not all errors are equally important and sometimes, one might try to 'optimize' an error, which has no significant influence. Let's see this in more detail. Let the errors  $e_{Model}$ ,  $e_{Numerics}$ ,  $e_{Software}$  enter. The total error is defined as

$$e_{Total} = e_{Model} + e_{Numerics} + e_{Software} \quad (4)$$

Let us assume that we have the numbers

$e_{Model} = 1000$ ,  $e_{Numerics} = 0.001$ ,  $e_{Software} = 4$ , the total error is then given by

$$e_{Total} = 1000 + 0.001 + 4 = 1004.001. \quad (5)$$

Which error dominates? It is clearly  $e_{Model} = 1000$ . The relative influence is  $e_{Model}/e_{Total} = 0.996$ . So, the other two error sources are negligible and would not need further attention in this specific example.

## Example: Error estimation

**Error estimation:** Error estimation is the process to obtain the concrete numbers 1000, 0.001, 4 in the previous example. Error estimates can be classified into two categories:

- **a priori estimates** include the (unknown) exact solution  $u$ , such that  $\eta := \eta(u)$ , and yield qualitative convergence rates for asymptotic limits. They can be derived before (thus a priori) the approximation is known.
- **a posteriori error estimates** are of the form  $\eta := \eta(\tilde{u})$  explicitly employ the approximation  $\tilde{u}$  and therefore yield quantitative information with computable majorants (i.e., bounds) and can be further utilized to design **adaptive schemes**.

## Example: Efficiency

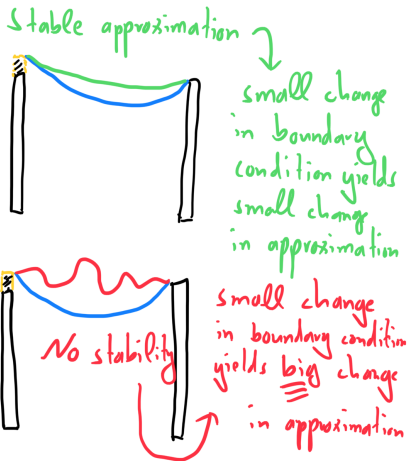
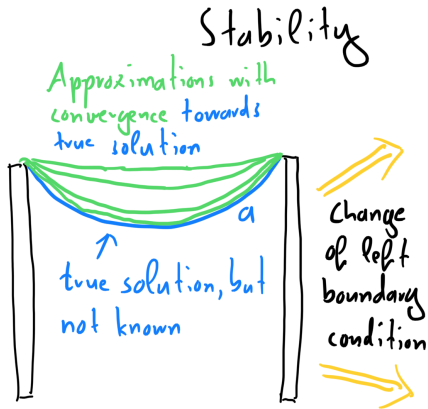
**Efficiency:** is more or less self-explaining.

- A first answer is to look at CPU or wall time: how many seconds, minutes, weeks, months does a program need to terminate and yield a result?
- A second answer is to study 'iteration numbers' or arithmetic operations.
- The latter are often given in terms of the big  $O$  notation.
- Having a linear equation system  $Ax = b$  with  $A \in \mathbb{R}^{n \times n}$  and  $O(n^3)$  complexity means that we need  $n^3$  (cubic in unknowns  $n$ ) arithmetic operations to calculate the result.
- For instance, for  $n = 100$ , we need around 1 000 000 operations.
- Having another algorithm (yielding the same result of  $Ax = b$ ) with only  $O(n)$  operations, means that we only need around 100 operations, which is a great difference.
- The development of efficient solvers for large linear equations systems is consequently a big branch in numerics and scientific computing.



## Example: Stability

**Stability:** We finally come back to the clothesline problem and change a bit the left boundary condition:



# Types of learning: three basic paradigms

- Supervised learning
- Unsupervised learning
- Reinforcement learning (later in L10)

# Supervised learning

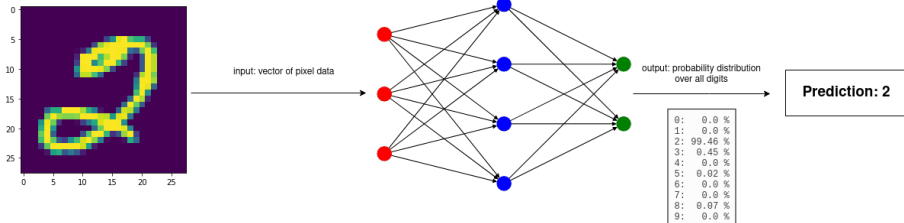
- **Labeled** examples  $\{(x_i, y_i)\}_{i=1}^N$ ,  $N \in \mathbb{N}$
- **Feature vector (components in numerics)**  $x_i$  (e.g., a person)
- **Feature (component)**  $x_i^{(j)}$  (value  $j$ : height, weight, ...)
- **Label**  $y_i$
- Label can belong to a finite set of classes  $\{1, 2, \dots, N_C\}$ , a real number, or a more complex structure (e.g., vector, matrix, ...)
- **Goal** of supervised learning: use the dataset  $\{(x_i, y_i)\}_{i=1}^N$  and produce a **model**
- $M(x) = y$ , where  $M$  is the model that takes  $x$  as input and gives some  $y$  as label
- **First simple example**: spam detection in emails, two classes  $\{spam, notspam\}$

## Example for building a model $M$

- Measurement data  $\{(x_i, y_i)\}_{i=1}^N$
- Find linear relation
- **Linear regression** (later more)
- **Linear model  $M$  with  $M(x) = b + mx$**  in which intercept  $b$  and slope  $m$  must be determined
- Thus: **'producing' or 'creating' a model** means in this case to determine  $b$  and  $m$
- Least-squares problem:

$$\min S = \min \frac{1}{N} \sum_{i=1}^N (y_i - (b + mx_i))^2 \quad (6)$$

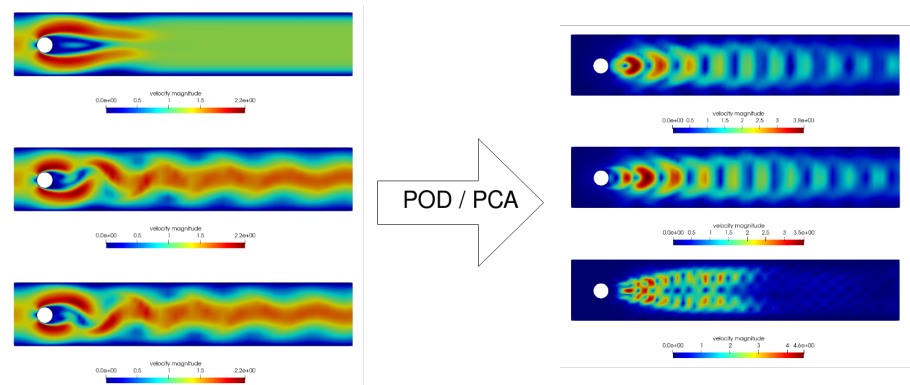
# Example for supervised learning



# Unsupervised learning

- Dataset is a collection of **unlabeled** examples  $\{x_i\}_{i=1}^N$
- Goal is finding patterns in data without supervision
- **Examples:**
  - clustering
  - dimensionality reduction
  - outlier detection

# Example for unsupervised learning



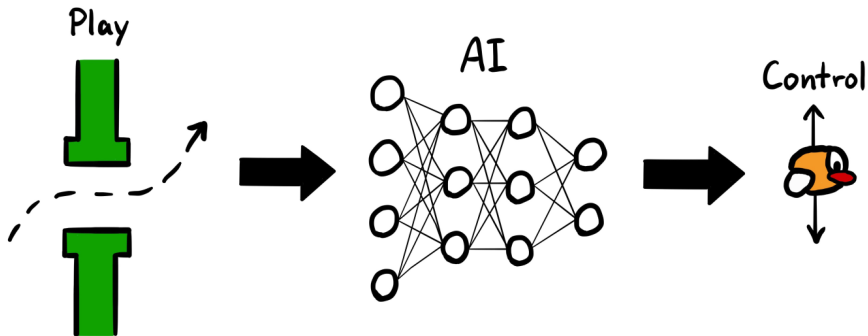
- POD = proper orthogonal decomposition (based on SVD singular value decomposition known from linear algebra - later more in Lecture 4)
- PCA = principal component analysis

# Reinforcement learning

- More details later in L10
- Machine lives in **environment**
- **State** of environment as **feature vector**
- Machine can execute actions in every state
- Goal is to learn a **policy** that maximizes **reward** function
- $f(x)$  = optimal action in a certain state
- Process is a feedback-loop
- **Examples**: game playing, AlphaGo, robotics, logistics



# Example for reinforcement learning



End of Lecture 1

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline lecture 2

- Basic ingredients from analysis and linear algebra
- Basics from probability, random processes, and statistics:
  - Sample points, sample spaces, probability spaces
  - Random variable
  - Expected value, standard deviation, variance
  - Covariance, conditional probability, Bayes' theorem
  - Parameter estimation, maximum-likelihood
- Numerics: hyperparameters versus parameters
- Basic notions in machine learning:
  - Classification, regression
  - Model-based and instance-based learning
  - Shallow and deep learning

# Ingredients from analysis and linear algebra

- Please recall yourself (self-studies) basic notions for vectors, matrices, functions, derivatives, chain rule, necessary/sufficient conditions for minimum and maximum
- Summaries typically be found in numerics books, e.g., again Richter/Wick<sup>7</sup>
- Extensive explanations in typical textbooks on Analysis<sup>8</sup> and Lineare Algebra<sup>9</sup>

---

<sup>7</sup>T. Richter and T. Wick. *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*. Dec. 2017. ISBN: 978-3-662-54177-7. DOI: 10.1007/978-3-662-54178-4.

<sup>8</sup>H. Amann and J. Escher. *Analysis I*. Birkhäuser, 2006. URL: <https://link.springer.com/book/10.1007/978-3-7643-7756-4>; H. Amann and J. Escher. *Analysis II*. Birkhäuser, 2006. URL: <https://link.springer.com/book/10.1007/3-7643-7402-0>.

<sup>9</sup>G. Fischer. *Lineare Algebra*. Springer, 2014. URL: <https://link.springer.com/book/10.1007/978-3-658-03945-5>.

# Probability, random processes, and statistics

- Main literature, where the following contents are taken from:
  - Ralph C. Smith; *Uncertainty Quantification*, SIAM, 2014 (Chapter 4)<sup>10</sup>
  - Meyer Dwass; *Probability: Theory and applications*, W.A. Benjamin, Inc., New York, 1970<sup>11</sup>
  - Hans-Otto Georgii; *Stochastik*, 2009<sup>12</sup>
  - Andriy Burkov; *The Hundred-Page Machine Learning Book*, 2019 (Chapter 2)<sup>13</sup>
  - Christopher M. Bishop; *Pattern Recognition and Machine Learning*; 2006<sup>14</sup>

---

<sup>10</sup>Ralph C. Smith. *Uncertainty Quantification*. SIAM, 2014.

<sup>11</sup>Meyer Dwass. *Probability: Theory and applications*. W.A. Benjamin, Inc., New York, 1970.

<sup>12</sup>H.-O. Georgii. *Stochastik*. de Gruyter, 2009.

<sup>13</sup>Andriy Burkov. *The hundred-page machine learning book*. Andriy Burkov, 2019.

<sup>14</sup>C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

# Sample point and sample space

- **Sample point**: formal name for a possible outcome
- **Sample space** (often  $\Omega$ ): collection of all sample points
- **Discrete sample space**: Countable number of sample points
- **Example 1**: Coin is thrown once: Sample points  $w_1 = H$  and  $w_2 = T$  for head  $H$  and tail  $T$ . Dimension of sample space is 2.
- **Example 2**: Coin is thrown twice: Sample points  $w_1 = (H, H)$ ,  $w_2 = (H, T)$ ,  $w_3 = (T, H)$ ,  $w_4 = (T, T)$
- **Event**: A set of sample points
- **Example 3**:  $E_1 = \{w_1, w_3\}$ ,  $E_2 = \{w_1, w_2, w_4\}, \dots$

# Probability measure

- Given event  $A$
- Probability of  $A$ : number  $P(A)$
- 'P' stands for Probability
- Property of  $P$ :  $0 \leq P(A) \leq 1$
- $P(Y) = 1$ , where  $Y$  is the certain event
- $P(\emptyset) = 0$  for the impossible event



# Probability space

## Definition (Probability space)

A probability space  $(\Omega, F, P)$  is defined of three components:

- $\Omega$ : sample space is the set of all possible outcomes from an experiment
- $F$ :  $\sigma$ -field of subsets of  $\Omega$  that contains all events of interest; the  $\sigma$ -field is also known as  $\sigma$  algebra
- $P : F \rightarrow [0, 1]$ : probability or measure that satisfies the postulates  $P(\emptyset) = 0$ ,  $P(\Omega) = 1$ , and thirdly, if  $A_i \in F$  and  $A_i \cap A_j = \emptyset$ , then 
$$P\left(\bigcup_{i=1}^{\infty} A_i\right) = \sum_{i=1}^{\infty} P(A_i)$$

# Random variables

- **Random variable**  $X$ : variables with output due to random phenomena
- Let the previous definition of the probability space be given. Then

## Definition (Random variable)

A random variable is a measurable function  $X : \Omega \rightarrow \mathbb{E}$ , where  $\mathbb{E}$  is a measurable space. Moreover, for  $S \subset \mathbb{E}$ , we have

$$P(X \in S) = P(\{\omega \in \Omega | X(\omega) \in S\})$$

- Discrete (see previous slides) and continuous cases are possible
- Probability distribution listed as **probability mass function** (discrete) and **probability density function** (continuous case)
- pdf = probability density function
- Often pdf not known, but some values of  $X$ : such values are called **examples**.
- Collection of examples is known as **sample** or **dataset**

## Example: six-faced die

- Numbers 1, 2, 3, 4, 5, 6 are usually assigned to its faces
- Denote outcome of sample points:  $w_1, w_2, w_3, w_4, w_5, w_6$
- Random variable  $X$  with definition

$$X(w_1) = 1, \dots, X(w_6) = 6 \quad (7)$$

## Definition of expected value (discrete situation)

- Let us assume  $X = (w_1, w_2, \dots, w_n)$  is a discrete sample space with the probability measure  $P$
- Assign  $p_i$  to  $w_i$  for  $i = 1, 2, \dots$
- If  $X$  is random variable, **expected value** (also known as mean, average) of  $X$  is denoted by  $E(X)$  and defined by

$$E(X) = \sum_{i=1}^n X(w_i) p_i \quad (8)$$

- Assumption

$$\sum_{i=1}^n |X(w_i)| p_i < \infty \quad (9)$$

(absolute convergence)

## Example (cont'd): six-faced die

- **Example:** die:

$$E(X) = \frac{1}{6} + \dots + \frac{6}{6} = \frac{21}{6} = 3.5 \quad (10)$$

- Probability of each event:  $p_i = \frac{1}{6}$
- $X(w_i) = 1, \dots, 6$  for  $i = 1, \dots, 6$

# Standard deviation and variance

- Description how a distribution is concentrated about the center of gravity (expected value)
- **Standard deviation:** Suppose  $X$  is a random variable with  $E(X^2) < \infty$ . Then, the standard deviation is defined by

$$\sigma := \sqrt{E[(X - \mu)^2]}, \quad (11)$$

where  $\mu = E(X)$

- **Variance**

$$\sigma^2 := \text{Var}(X) := E[(X - \mu)^2]. \quad (12)$$

# Expectation and variance of continuous random variable $X$

- Extension from discrete cases (taking sum) to continuous cases (integral)
- **Expectation:**

$$E(X) := \int_{\mathbb{R}} x f_X(x) dx \quad (13)$$

where  $f_X$  is the pdf of the variable  $X$

- **Variance:**

$$\text{Var}(X) = \sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 f_X(x) dx \quad (14)$$

with  $\mu := E(X)$

## Covariance

- Suppose  $X$  and  $Y$  are independent random variables and their variances  $Var(X)$  and  $Var(Y)$  exist. Then the sum variance exists and it holds

$$Var(X + Y) = Var(X) + Var(Y). \quad (15)$$

Standard proof in stochastic lecture (see e.g., Dwass, Chapter 10)

- If  $X$  and  $Y$  are not independent, there is a relation, which is known as **covariance**:

$$Var(X + Y) = Var(X) + Var(Y) + 2E[(X - E(X))(Y - E(Y))] \quad (16)$$

where the covariance is defined as

$$Cov(X, Y) = E[(X - E(X))(Y - E(Y))]. \quad (17)$$

- Be careful:  $X$  and  $Y$  independent implies  $Cov(X, Y) = 0$ . But  $Cov(X, Y) = 0$  does not imply that  $X$  and  $Y$  independent



## Conditional probability and Bayes' theorem

- **Conditional probability:**  $P(A|B)$  is the probability of  $A$  given  $B$ . It holds

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \quad (18)$$

- Remark:  $P(A \cap B) = P(A|B)P(B)$
- Since  $(A \cap B) \subset B$  and  $P(A \cap B) \leq P(B)$ , it follows

$$0 \leq P(A|B) \leq 1 \quad (19)$$

- For random variables  $X$  and  $Y$  with possible values  $x$  and  $y$ , it holds **Bayes' theorem**

$$P(X = x|Y = y) = \frac{P(Y = y|X = x)P(X = x)}{P(Y = y)} \quad (20)$$

## Example

- Coin is thrown three times
- We have obtained a total of two heads
- **Question:** What is the conditional probability of a head on the first trial?
- **Formalization:**  $A$  = head on first trial,  $B$  = two heads in three trials
- First observation: in total, we have eight sample points:  
 $(H, H, H), (H, H, T), \dots, (T, T, T)$
- Thus:  $A \cap B$  consists of the sample points  $(H, H, T)$  and  $(H, T, H)$
- And:  $B$  consists of the sample points  $(T, H, H), (H, T, H), (H, H, T)$
- It follows:

$$P(A \cap B) = \frac{2}{8}, \quad P(B) = \frac{3}{8} \quad (21)$$

- With **conditional probability:**

$$P(A|B) = \frac{2/8}{3/8} = \frac{2}{3} \quad (22)$$

- Interpretation: with the prior information that two heads have been obtained, we have  $P(A|B) = 2/3$ . Compare this result two the original probability of a head in the first trial, which is only  $P(A) = \frac{1}{2}$

# Probability mass function

- **Definition:** A continuous random variable  $X$  can be expressed by

$$F_X(x) = \int_{-\infty}^x f_X(s) ds, \quad x \in \mathbb{R} \quad (23)$$

- The derivative  $f_X = \frac{dF_X}{dx}$  is called the **probability density function (pdf)** of  $X$
- pdf properties are:

$$f_X(x) \geq 0 \quad (24)$$

$$\int_{\mathbb{R}} f_X(x) dx = 1 \quad (25)$$

$$P(x_1 \leq X \leq x_2) = F_X(x_2) - F_X(x_1) = \int_{x_1}^{x_2} f_X(x) dx \quad (26)$$

## Example: normal distribution

### Definition

A univariate density is the normal density, which is defined as

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}, \quad -\infty < x < \infty \quad (27)$$

- Notation:  $X \sim N(\mu, \sigma^2)$  means that  $X$  is normally distributed with mean  $\mu$  and variance  $\sigma^2$
- 68.25% of area within  $1\sigma$
- 95.45% of area within  $2\sigma$
- 99.73% of area within  $3\sigma$

# Parameter estimation

- Given a model  $f_\theta$  with some parameters in the form of  $\theta$
- **Example:**

$$f_\theta = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad (28)$$

with  $\theta := [\mu, \sigma]$

- Use this as a **model** of an unknown distribution of  $X$
- **Update values of parameters** inside  $\theta$  using Bayes' theorem:

$$P(\theta = \hat{\theta} | X = x) = \frac{P(X = x | \theta = \hat{\theta})P(\theta = \hat{\theta})}{P(X = x)} \quad (29)$$

- Here

$$P(X = x) = \sum_{\hat{\theta}} P(X = x | \theta = \hat{\theta}) \quad (30)$$

and  $f_{\hat{\theta}} := P(X = x | \theta = \hat{\theta})$

# Parameter estimation

- Given a sample  $S$  of  $X$
- Estimate  $P(\theta = \hat{\theta})$  by applying Bayes' rule for  $x \in S$  subsequently
- Initial value  $P(\theta = \hat{\theta})$  can be guessed such that  $\sum_{\hat{\theta}} P(\theta = \hat{\theta}) = 1$
- This guess is called **prior**
- **Iterative procedure** of updating new parameters  $\hat{\theta}$
- Best value of parameters  $\theta^*$  by **maximum-likelihood**:

$$\theta^* = \arg \max_{\hat{\theta}} \prod_{i=1}^N P(\theta = \hat{\theta} | X = x_i) \quad (31)$$

# Parameters vs. hyperparameters

- **Hyperparameters** are parameters that control the algorithm used for the machine learning process
- **Example**: learning rate (relaxation parameter)  $\omega_k$  in gradient descent, line search in Newton, preconditioners in iterative solution of  $Ax = b$
- Iteration:  $x_{k+1} = x_k + \omega_k d_k$
- Stop iteration when  $\|x_{k+1} - x_k\| < TOL$
- **Parameters** are linked to the model that shall be learned. To find optimal values is the main goal of the learning process
- **Example**: intercept  $b$  and slope  $m$  in linear regression  $y(x) = b + mx$

# On machine epsilon and tolerances of algorithms

- We define a tolerance TOL such that

$$\|x_{k+1} - x_k\| < TOL. \quad (32)$$

- In order to avoid difficulties due to machine precision we must choose  $TOL \gg \epsilon$ . For instance:
  - i) we have for machine epsilon  $\epsilon \approx 10^{-16}$  in double precision
  - ii) **Reasonable tolerances** are in the range of  $TOL = 10^{-8}, \dots, 10^{-12}$ .

- **Example:** Take Newton's method for solving  $f(x) = x^2 - 2 = 0$

- For a classical choice  $TOL = 1e - 12$  we obtain

Iter	x	f(x)
5	1.414214e+00	4.751755e-14

This means that we need 5 iterations to converge to a root value  $|f(x)| = 4.75e - 14$ .

- For  $TOL = 1e - 6$

Iter	x	f(x)
4	1.414214e+00	6.156754e-07

This means that we need 4 iterations (the scheme is more efficient!) to converge to a root value  $|f(x)| = 1.53e - 07$  (the final root is less accurate!).

- Here, we already see the **trade-off between efficiency and accuracy** (recall numerical concepts from lecture 1)



# On machine epsilon and tolerances of algorithms

- Now let us choose  $TOL = 1e - 16$ , a value of the order of the machine precision. Then:

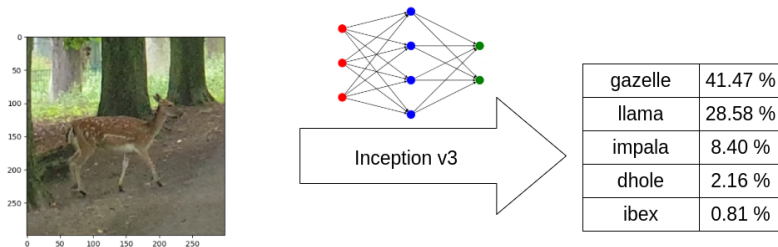
Iter	x	f(x)
...		
6502	1.414214e+00	-4.440892e-16
6503	1.414214e+00	4.440892e-16
6504	1.414214e+00	-4.440892e-16
6505	1.414214e+00	4.440892e-16
...		
16368	1.414214e+00	-4.440892e-16
16369	1.414214e+00	4.440892e-16
16370	1.414214e+00	-4.440892e-16
16371	1.414214e+00	4.440892e-16
...		

- The computation did not stop (endless loop) because of round-off errors due to machine precision such that the tolerance cannot be met.
- This shows indeed that we must stay away with  $TOL$  from  $\epsilon$  and should choose  $TOL > \epsilon$ .

# Classification vs. regression

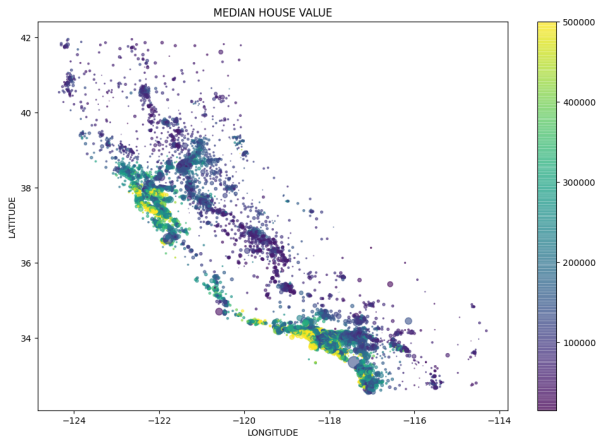
- **Classification**: automatically assigning a label to an unlabeled example
- Classification learning solution: take labeled examples and create a model
- Take now the model with unlabeled input and get a labeled output
- **Regression**: predicting a real-valued label given an unlabeled example
- Regression learning algorithm: as before: create model with labeled examples and then use that model with unlabeled examples with real-valued output

# Example for classification



- Live demo: <https://teachablemachine.withgoogle.com/>

# Example for regression

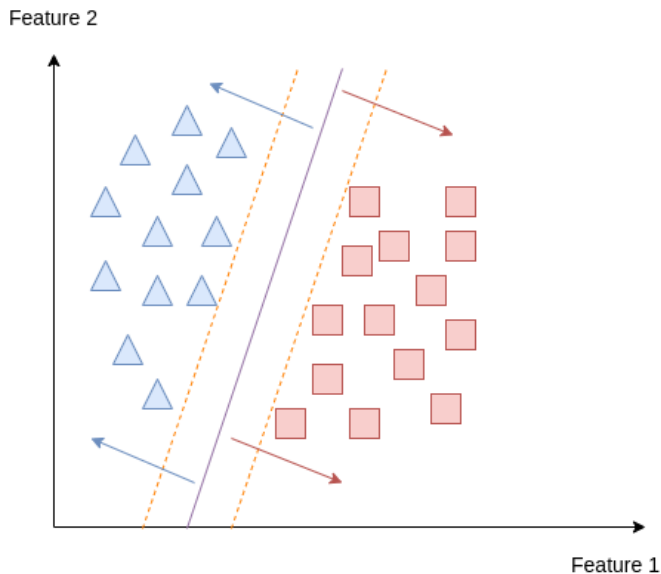


- Input: longitude, latitude, total rooms, ... , ocean proximity
- Output: house price in \$

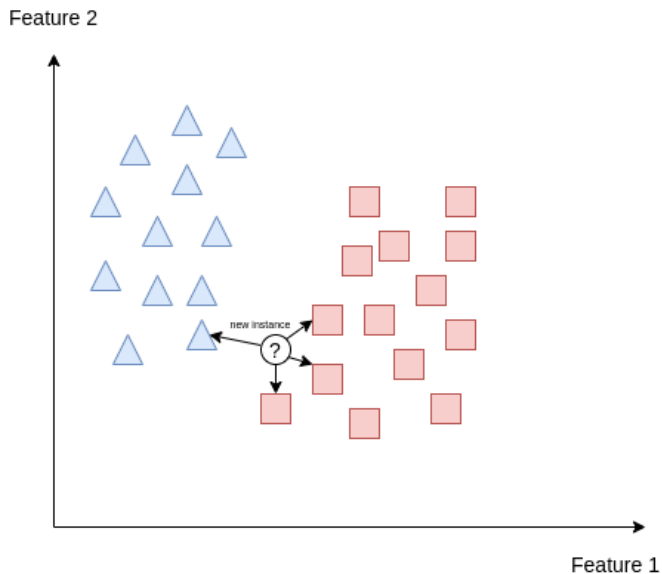
# Model-based vs. instance-based learning

- **Model-based** learning algorithms use training data for creating a model with parameters learned from the data (e.g., SVM for learning the slope  $w$  and x-axis cut  $b$ )
- **Instance-based** learning: use whole dataset as a model, e.g., k-Nearest Neighbors (kNN)

# Example for model-based learning



# Example for instance-based learning

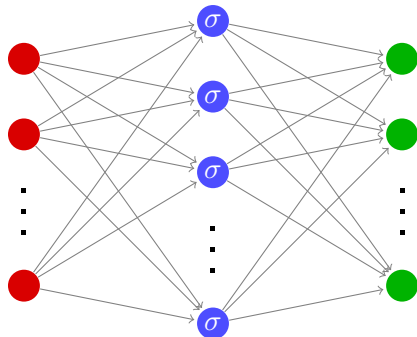


# Shallow vs. Deep learning

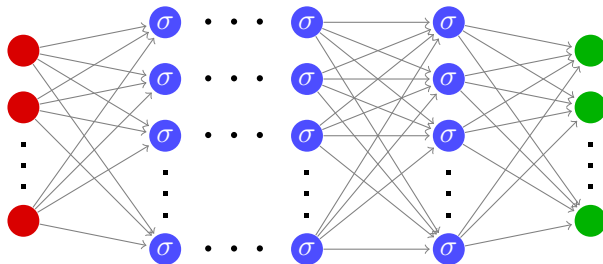
- **Shallow** learning: learning immediately from the features, e.g., decision trees, SVM, shallow neural networks (one hidden layer)
- **Deep** learning: several stages in the learning process, e.g., deep neural networks (several hidden layers)



# Example for shallow neural network



# Example for deep neural network



End of Lecture 2

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms**
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

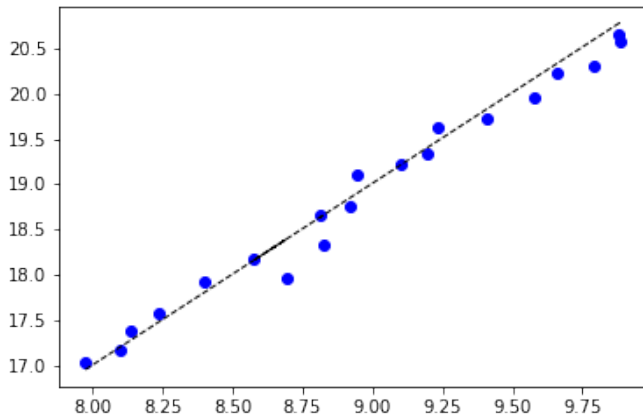
- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

## Outline lecture 3

- Linear Regression,
- Logistic Regression,
- Decision Tree Learning,
- Random Forest,
- Support Vector Machines,
- k-Nearest Neighbors,
- k-Means Clustering.

# Linear regression



# Problem statement

- 1 Collection of labeled examples  $\{(x_i, y_i)\}_{i=1}^N$ , where  $N$  is the size of the collection
- 2  $x \in \mathbb{R}^D$
- 3 **Goal 1:** Build model

$$f_{w,b}(x) = wx + b \quad (33)$$

where  $w \in \mathbb{R}^D$

- 4 **Goal 2:** Determine  $w$  and  $b$
- 5 **Solution:**

$$\min_{w,b} \frac{1}{N} \sum_{i=1}^N (f_{w,b}(x_i) - y_i)^2 \quad (34)$$

- 6 (34) is called **objective function**
- 7 The summand in (34) is called **loss function**

# Questions and problems

- ① What is a good numerical algorithm?
- ② Is there some theoretical justification?
- ③ **Overfitting**: model predicts very well labels of the given examples used for training, but does not well predict new labeled examples



## Example: We sell honey! <sup>15</sup>

- ① Situation: we want to start selling honey and ask the question about the optimal price
- ② Approach: do some test sells with different prices and see how many glasses can be sold
- ③ Measurement data: price  $p(x)$  and number of honey glasses  $x$

No $x$ of sold honey glasses	50	30	21	22	27	30	26	32	28	26	21	16	8	4
Price $p(x)$ per glass	3.5	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5	8.0	8.5	9.0	9.5	10.0

**Table:** Data of 14 measurements of different prices and the corresponding number of sold glasses.

- ④ Goal: Construct  $p(x) = mx + b$ , where  $p(x)$  denotes the price per glass

---

<sup>15</sup>Richter, Wick, Springer, 2017, p. 308ff

## Example: We sell honey!

- 1 Least-squares problem statement

$$\min_{m,b} S := S(m, b) = \frac{1}{N} \sum_{k=1}^N (p_k - (b + mx_k))^2, \quad (35)$$

- 2 Numerical solution via gradient descent

- 3 Initial guesses:  $m_0$  and  $p_0$

- 4 Step length  $\rho$  (relaxation parameter, learning rate)

- 5 Gradient of the function  $S(m, b)$ :

$$\frac{\partial S}{\partial m} = \frac{2}{N} \sum_{k=1}^N -x_k (p_k - (mx_k + b)) \quad (36)$$

$$\frac{\partial S}{\partial b} = \frac{2}{N} \sum_{k=1}^N -(p_k - (mx_k + b)) \quad (37)$$

## Example: We sell honey!

- 1 Final algorithm for gradient descent:

$$\begin{pmatrix} m_{l+1} \\ b_{l+1} \end{pmatrix} = \begin{pmatrix} m_l \\ b_l \end{pmatrix} - \rho \begin{pmatrix} \frac{\partial S(m_l)}{\partial m} \\ \frac{\partial S(b_l)}{\partial b} \end{pmatrix}. \quad (38)$$

- 2 Initial guesses  $b_0 = 10.5$  and  $m_0 = -2$
- 3 Step length  $\rho = 10^{-4}$
- 4 **Optimal values/solution** after  $N_{max} = 200$  iterations:

$$b = 10.5613025417, \quad m = -0.154072634203. \quad (39)$$

- 5 **Final linear regression function:**

$$p(x) = \underbrace{10.561}_{=b} - \underbrace{0.1541}_{=m} x. \quad (40)$$

## Example: We sell honey!

- 1 **Interpretation:** What can we do with (40)?
- 2 **Sales volume function** (Umsatzfunktion):

$$U := U(x) = p(x) \cdot x = 10.561x - 0.1541x^2 \quad (41)$$

- 3 The maximum will denote the number of glasses  $x$  at which we make the largest sale:

$$U'(x) = 0 \quad \Leftrightarrow \quad 10.5613 - 0.30815x = 0 \quad \Rightarrow \quad x = 34.273 \quad (42)$$

- 4 34 sold glasses, each for 5.28 EUR

## Example: We sell honey!

- 1 **Profit function:**  $G(x) = U(x) - K(x)$
- 2 Assume fixed costs of 2.59 EUR per glass
- 3 Then:

$$G(x) = U(x) - K(x) = p(x) \times x - K(x) \quad (43)$$

$$= 10.561x - 0.1541x^2 - 2.59x \quad (44)$$

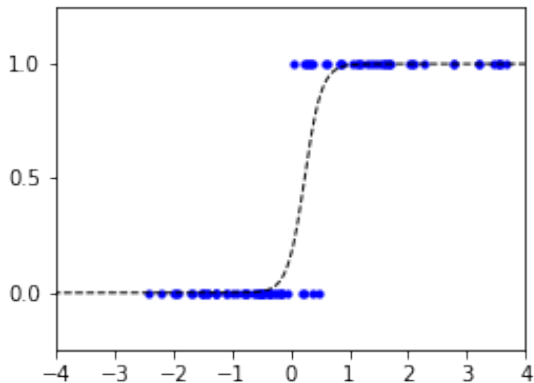
$$= 7.971x - 0.1541x^2 \quad (45)$$

- 4 Limit profit:  $G'(x)$ :

$$G'(x) = 7.971 - 0.30815x \quad (46)$$

- 5 Using  $G'(x)$  yields maximum of  $G(x)$
- 6 Here:  $25.87 \approx 26$  glasses, each for 5.28 EUR

# Logistic regression



# Logistic regression

- 1 **Logistic regression** is **not regression**, but a **classification learning** algorithm
- 2 Standard logistic function (**sigmoid function**):

$$f(x) = \frac{1}{1 + e^{-x}} \quad (47)$$

- 3 Logistic regression model:

$$f_{w,b} := \frac{1}{1 + e^{-wx+b}} \quad (48)$$

- 4 If  $w$  and  $b$  can be optimized,  $f(x)$  denotes a probability of  $y_i$  being positive (or negative)

# Logistic regression

- 1 **Goal:** maximize likelihood of our training set according to the model
- 2 **Optimization criterion:** maximum likelihood
- 3 Recall: linear regression: minimize average squared loss; mean squared error (MSE)
- 4 **Logistic regression** maximizes likelihood of the training data according to the model:

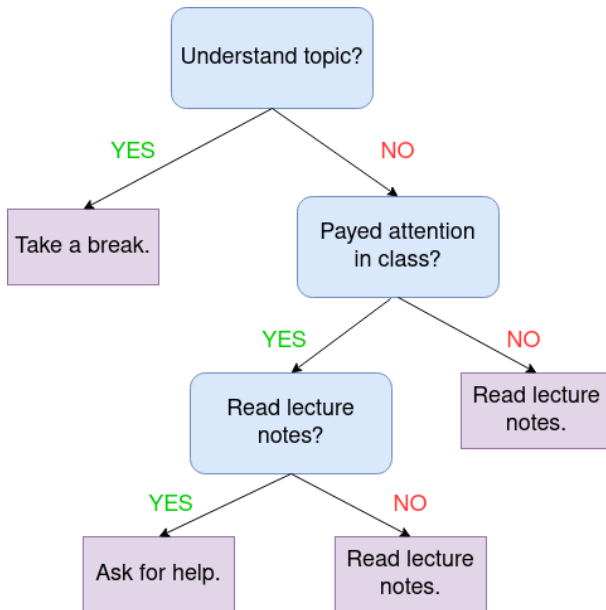
$$L_{w,b} := \prod_{i=1}^N f_{w,b}(x_i)^{y_i} (1 - f_{w,b}(x_i))^{(1-y_i)} \quad (49)$$

(see also back in Lecture 2 for the maximum likelihood function in parameter estimation)

- 5 In practice, **log-likelihood** often used in order to **avoid numerical overflow**



# Decision Tree



# Decision Tree

- 1 Acyclic graph, which can be used to make decisions
- 2 Learning again by data
- 3 Given: labeled examples, labels belong to  $\{0, 1\}$
- 4 ID3 (Iterative Dichotomiser 3)<sup>16</sup>
- 5 Optimization criterion: average log-likelihood:

$$\frac{1}{N} \sum_{i=1}^N [y_i \ln f_{ID3}(x_i) + (1 - y_i) \ln(1 - f_{ID3}(x_i))], \quad (50)$$

where  $f_{ID3}$  is a decision tree.

---

<sup>16</sup>J. R. Quinlan. "Induction of decision trees". In: *Machine learning 1* (1986), pp. 81–106. DOI: <https://doi.org/10.1007/BF00116251>.

# Decision Tree

- 1 Difference logistic regression and decision tree
- 2 Logistic regression: parametric model  $f_{w,b}$  with some optimal  $w$  and  $b$
- 3 ID3 algorithm: nonparametric model  $f_{ID3}(x) := P(y = 1|x)$

# Decision Tree: Algorithm

- 1 Let  $S$  be a set of labeled examples
- 2 At begin only one node with all examples:  $S := \{(x_i, y_i)\}_{i=1}^N$
- 3 Constant model:

$$f_{ID3}^S := \frac{1}{|S|} \sum_{(x,y) \in S} y \quad (51)$$

- 4 Go through all features  $j = 1, \dots, D$  and all thresholds  $t$
- 5 Split set  $S$  into two subsets

$$S_- := \{(x, y) | (x, y) \in S, x^j < t\} \quad (52)$$

$$S_+ := \{(x, y) | (x, y) \in S, x^j \geq t\} \quad (53)$$

- 6 With two new subsets, go to new leaf nodes
- 7 Evaluate for  $(j, t)$  how good the split is
- 8 Pick best values  $(j, t)$  and split again  $S$  into  $S_+$  and  $S_-$

# Decision Tree: Entropy

- 1 Evaluate how good split is: take entropy
- 2 **Entropy** measures uncertainty about a random variable
- 3 Maximal entropy: all values of the random variable are equiprobable
- 4 Minimal entropy: random variable can have only one value
- 5 **Definition** of a set of examples  $S$ :

$$H(S) := -f_{ID3}^S \ln f_{ID3}^S - (1 - f_{ID3}^S) \ln(1 - f_{ID3}^S) \quad (54)$$

## Decision Tree: Entropy

- 1 Split a set of examples by a certain feature  $j$  and threshold  $t$
- 2 **Entropy of a split** is denoted by  $H(S_-, S_+)$  and defined as weighted sum of two entropies:

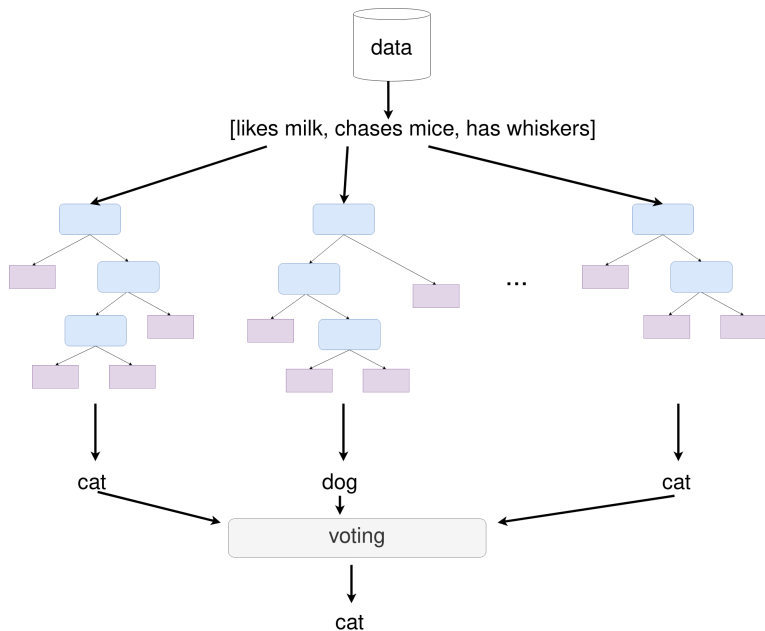
$$H(S_-, S_+) := \frac{|S_-|}{|S|} H(S_-) + \frac{|S_+|}{|S|} H(S_+) \quad (55)$$

- 3 In the ID3 algorithm at each leaf node, we find a split that minimizes the entropy using the previous weighted sum or we stop at this leaf node
- 4 Algorithm does not guarantee an optimal solution
- 5 Improvement via backtracking
- 6 Extension of ID3: C4.5 algorithm (again Ross Quinlan)<sup>17</sup>

---

<sup>17</sup>J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

# Random Forest

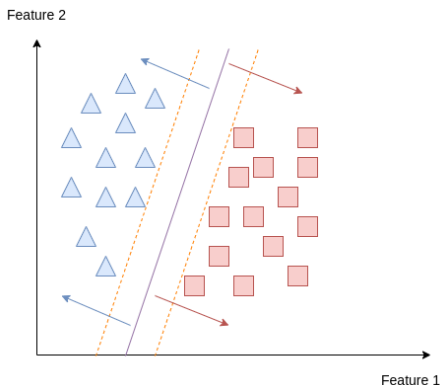


# Random Forest

- Train  $N$  decision trees on random samples of the data
- Regression: take average of the predictions of the  $N$  decision trees
- Classification: take majority vote of the predictions of the  $N$  decision trees
- Multiple samples of dataset  $\Rightarrow$  reduce variance  $\Rightarrow$  less overfitting
- Ensemble learning: combine many low-accuracy models into a high-accuracy meta-model



# Support Vector Machines



Literature: Chapter 12 of Mathematics for Machine Learning<sup>18</sup>

---

<sup>18</sup>Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. ISBN: 9781108470049. URL: <https://books.google.de/books?id=pFjPDwAAQBAJ>.

# Support Vector Machines (SVM)

- 1 SVM belongs to supervised learning
- 2 Positive label has value of  $+1$
- 3 Negative label has value of  $-1$
- 4 SVM takes every feature vector as a point in a high-dimensional space  $\mathbb{R}^D$ , where  $D \gg 1$  is the dimension
- 5 **Goal:** construct  $\mathbb{R}^{D-1}$  line, i.e., hyperplane to separate positive and negative labels
- 6 This line is the **decision boundary**
- 7 SVM is close to linear regression: again two parameters  $w$  and  $b$  must be determined

# Support Vector Machines (SVM)

- ① SVM equation:

$$wx - b = 0 \quad (56)$$

where (just to be clear)

$$wx = \sum_{i=1}^D w^{(i)} x^{(i)} \quad (57)$$

- ② Predicted label for some input vector  $x$  is

$$y = \text{sign}(wx - b) \quad (58)$$

- ③ **Goal:** Determine optimal values for  $w^*$  and  $b^*$

- ④ **Model:**

$$f(x) = \text{sign}(w^* x - b^*) \quad (59)$$

# Support Vector Machines (SVM)

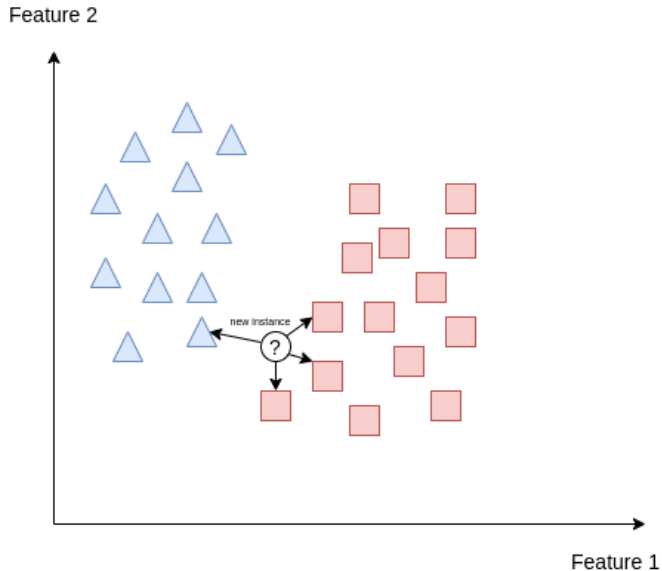
- 1 Within the above model, the separation of positive and negative labels should be as clear as possible: large **margin**
- 2 Margin is the distance between closest examples
- 3 Minimize Euclidian norm  $\|w\|$
- 4 Optimization problem:

$$\min \|w\| \quad \text{s.t.} \quad y_i(wx_i - b) \geq 1 \quad (60)$$

for  $i = 1, \dots, N$

- 5 Solution  $w^*$  and  $b^*$  is called the **statistical model**
- 6 Recall: process to build model is called **training**
- 7 SVM can have problems with noisy data (no clear separation possible) or when no hyperplane can be constructed (but possibly a higher-order polynomial)

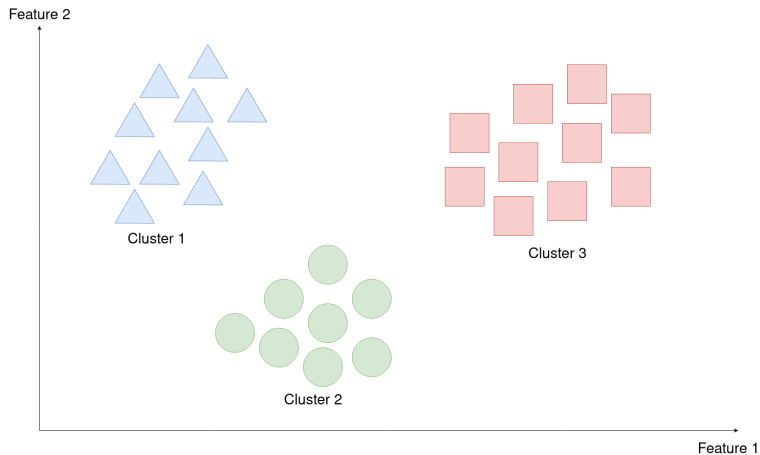
# k-Nearest Neighbors



# k-Nearest Neighbors: kNN

- 1 Nonparametric learning algorithm
- 2 kNN keeps training examples during entire usage
- 3 Recall: other training algorithms (e.g., linear regression) use training data to build model, i.e.,  $w$  and  $b$ , but then does 'forget' the training data
- 4 kNN: algorithm finds  $k$  training examples closest to a new point  $x$
- 5 Returns either majority label in classification or average label in regression
- 6 Procedure of kNN is simple: determine distance between two examples, e.g., Euclidian distance
- 7 Further popular choices: cosine similarity, Chebychev distance, Hamming distance, ...
- 8 Choice of distance is done a priori, and therefore a hyperparameter choice

# k-Means Clustering



# k-Means Clustering

- 1 Choose  $k$ , the number of clusters.
- 2 Put  $k$  random feature vectors - centroids - to the feature space
- 3 Assign the closest centroid to each example
- 4 Update the location of the centroids as the average of its surrounding feature vectors
- 5 Go to step 2 until convergence



End of Lecture 3

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

## Outline lecture 4

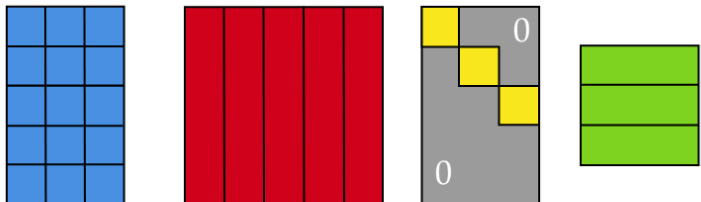
- SVD: Singular value decomposition<sup>19</sup>
- PCA: Principal component analysis<sup>20</sup>

---

<sup>19</sup>Richter and Wick, *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*.

<sup>20</sup>I.T. Jolliffe. *Principal Component Analysis*. Springer, 2002. DOI: <https://doi.org/10.1007/b98835>.

# Singular Value Decomposition



The diagram illustrates the Singular Value Decomposition (SVD) of a matrix  $A$ . It shows four matrices:  $A$  (a 5x3 grid of blue squares),  $U$  (a 5x5 grid of red vertical bars),  $\Sigma$  (a 5x3 matrix with a gray background, yellow squares on the diagonal, and '0' in the top-right and bottom-left corners), and  $V^T$  (a 3x3 grid of green horizontal bars). Below each matrix is its mathematical representation:  $A \in \mathbb{R}^{n \times m}$ ,  $U \in \mathbb{R}^{n \times n}$ ,  $\Sigma \in \mathbb{R}^{n \times m}$ , and  $V^T \in \mathbb{R}^{m \times m}$ .

$$A = U \Sigma V^T$$

$A \in \mathbb{R}^{n \times m}$        $U \in \mathbb{R}^{n \times n}$        $\Sigma \in \mathbb{R}^{n \times m}$        $V^T \in \mathbb{R}^{m \times m}$

# Singular Value Decomposition

- 1 Let  $A \in \mathbb{R}^{n \times m}$
- 2 Let  $U \in \mathbb{R}^{n \times n}$  and  $V^T \in \mathbb{R}^{m \times m}$  be orthogonal matrices
- 3 The singular value decomposition of  $A$  is defined as

$$U^T A V = \Sigma, \quad (61)$$

with  $\Sigma \in \mathbb{R}^{n \times m}$

- 4 We have

$$\Sigma = \left( \begin{array}{ccc|c} \sigma_1 & & 0 & 0 \\ & \ddots & & \\ 0 & & \sigma_n & \\ \hline 0 & \dots & 0 & \end{array} \right), \text{ if } n > m, \quad \Sigma = \left( \begin{array}{ccc|c} \sigma_1 & & 0 & 0 \\ & \ddots & & \vdots \\ 0 & & \sigma_n & 0 \end{array} \right), \text{ if } n < m. \quad (62)$$

- 5 The values  $\sigma_1 \geq \sigma_2 \cdots \sigma_p \geq 0$  with  $p = \min\{n, m\}$  are called singular values of the matrix  $A$ .

# Existence of singular values

## Theorem

*Let us suppose there exists an SVD of the matrix  $A \in \mathbb{R}^{n \times m}$  into orthogonal matrices  $U \in \mathbb{R}^{n \times n}$ ,  $V \in \mathbb{R}^{m \times m}$  and an extended diagonal matrix  $\Sigma \in \mathbb{R}^{n \times m}$ . Then, the singular values are obtained as roots of the eigenvalues of the matrix  $A^T A$  and they are uniquely determined up to permutation.*

# Proof (I)

## Proof.

- 1 We suppose there exists an SVD
- 2 Then, it holds with orthogonal (regular) matrices  $U, V$

$$AV = U\Sigma, \quad A^T U = V\Sigma. \quad (63)$$

- 3 Let  $V = [v_1, \dots, v_m]$  and  $U = [u_1, \dots, u_n]$  be column vectors

- 4 Then

$$Av_i = \sigma_i u_i, \quad i = 1, \dots, n, \quad A^T u_i = \sigma_i v_i, \quad i = 1, \dots, m. \quad (64)$$

- 5 It follows for  $i = 1, \dots, p = \min\{n, m\}$  that

$$A^T Av_i = \sigma_i A^T u_i = \sigma_i^2 v_i, \quad i = 1, \dots, p. \quad (65)$$

- 6 The  $\sigma_i^2$  are the eigenvalues of  $A^T A$ .
- 7 Since  $A^T A$  can be not regular, eigenvalues can be zero



# Proof (II)

## Proof.

- 1 Uniqueness follows from uniqueness of the eigenvalues of  $A^T A$ .
- 2 The matrices  $U, V$  are in general not uniquely determined, since

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_r > \sigma_{r+1} = \dots = \sigma_p = 0 \quad (66)$$

the last  $p - r$  eigenvalues are zero and the matrices  $U$  and  $V$  can be arbitrarily permuted.





# Existence of an SVD

## Theorem

Let  $A \in \mathbb{R}^{n \times m}$ . Then, there exist two orthogonal matrices  $U \in \mathbb{R}^{n \times n}$  and  $V \in \mathbb{R}^{m \times m}$  with

$$U^T A V = \Sigma = \text{diag}(\sigma_1, \dots, \sigma_p) \in \mathbb{R}^{n \times m}, \quad p = \min\{n, m\}, \quad (67)$$

and with

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_p. \quad (68)$$

# Proof (I)

## Proof.

① Let

$$\sigma_1 = \|A\|_2 = \max_{x \in \mathbb{R}^n, x \neq 0} \frac{\|Ax\|_2}{\|x\|_2} > 0. \quad (69)$$

② Then, there exists a  $v \in \mathbb{R}^n$  with  $\|v\|_2 = 1$  in which the maximum is taken, i.e.,

$$\|Av\|_2 = \sigma_1, \quad (70)$$

i.e.,  $Av = \sigma_1 u \in \mathbb{R}^m$ , we have as well

$$\|u\|_2 = 1. \quad (71)$$

③ We extend  $v$  and  $u$  respectively to an orthogonal basis of  $\mathbb{R}^n$  and  $\mathbb{R}^m$ , i.e.,

$$\text{span}\{v, v_2, \dots, v_n\} = \mathbb{R}^n, \quad \text{span}\{u, u_2, \dots, u_m\} = \mathbb{R}^m. \quad (72)$$



# Proof (II)

## Proof.

- ① With the  $U_1 = [u, u_2, \dots, u_m]$  and  $V_1 = [v, v_2, \dots, v_n]$  it holds due to orthogonality

$$A_1 := U_1^T A V_1 = \begin{pmatrix} \sigma_1 & w^T \\ 0 & \tilde{A}_1 \end{pmatrix}, \quad (73)$$

with a vector  $w \in \mathbb{R}^{m-1}$  and a matrix  $\tilde{A}_1 \in \mathbb{R}^{n-1 \times m-1}$ .



# Proof (III)

## Proof.

- 1 We will establish that  $w^T = 0$  holds true.
- 2 It follows from orthogonality (see linear algebra lectures) that

$$\|A_1\|_2 = \|A\|_2 = \sigma_1. \quad (74)$$

- 3 For the vector  $x = (\sigma_1, w)^T \in \mathbb{R}^m$  it holds

$$A_1 \begin{pmatrix} \sigma_1 \\ w \end{pmatrix} = \begin{pmatrix} \sigma_1^2 + \|w\|^2 \\ \tilde{A}_1 w \end{pmatrix}, \quad (75)$$

i.e.,

$$\left\| A_1 \begin{pmatrix} \sigma_1 \\ w \end{pmatrix} \right\|_2^2 = (\sigma_1^2 + \|w\|^2)^2 + \|\tilde{A}_1 w\|_2^2 \geq (\sigma_1^2 + \|w\|^2)^2. \quad (76)$$

- 4 From  $\|A_1\|_2 = \sigma_1$  it follows  $w = 0$
- 5 Therefore,

$$A_1 = U_1^T A V_1 = \begin{pmatrix} \sigma_1 & 0 \\ 0 & \tilde{A}_1 \end{pmatrix}, \quad (77)$$

- 6 We then proceed by induction for  $\tilde{A}_1$

# Proof (IV)

## Proof.

- 1 If in step  $i$  it holds

$$\sigma_i = \|\tilde{A}_{i-1}\| = 0, \quad (78)$$

then the process stops.

- 2 The matrix  $\tilde{A}_{i-1}$  is then a rectangular zero matrix and the singular value decomposition is given.



## Example (I)

- 1 We determine the SVD of the singular matrix  $A = \begin{pmatrix} 1 & 1 \\ 7 & 7 \end{pmatrix}$ .
- 2 Rank of  $A$  is one since both columns are linearly dependent
- 3 Determine first  $A^T A$  as

$$A^T A = \begin{pmatrix} 50 & 50 \\ 50 & 50 \end{pmatrix}. \quad (79)$$

## Example (II)

- 1 Eigenvalues  $\lambda_1 = 100$  and  $\lambda_2 = 0$ .
- 2 The eigenvectors are

$$v_1 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad v_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}. \quad (80)$$

- 3 Determine matrix  $V$ :

$$V = [v_1, v_2] = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \quad (81)$$

- 4 The first singular value is  $\sigma_1 = \lambda(A^T A) = \sqrt{100} = 10$ .

## Example (III)

- 1 Determine now  $U$ :

$$u_1 = \frac{Av_1}{10} = \frac{(1, 7)}{\sqrt{50}}, \quad (82)$$

$$u_2 = \frac{Av_2}{10} = \frac{(-7, 1)}{\sqrt{50}}. \quad (83)$$

It holds

$$U = [u_1, u_2] = \frac{1}{\sqrt{50}} \begin{pmatrix} 1 & -7 \\ 7 & 1 \end{pmatrix}. \quad (84)$$

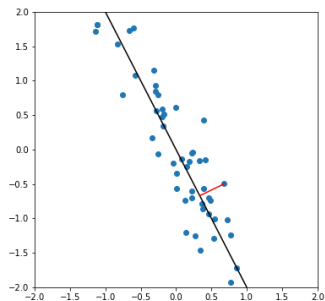
- 2 Clearly, it holds  $U^T U = I$ .  
3 The final decomposition is given by

$$A = U\Sigma V^T = \frac{1}{\sqrt{50}} \begin{pmatrix} 1 & -7 \\ 7 & 1 \end{pmatrix} \begin{pmatrix} 10 & 0 \\ 0 & 0 \end{pmatrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix}. \quad (85)$$

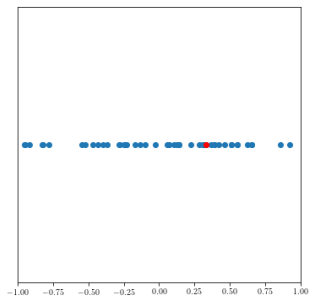


# Principal Component Analysis

$\mathbb{R}^2$



$\mathbb{R}$



# Principal Component Analysis

- 1 Nonparametric method (any data set can be treated, and no parameters to be tuned)
- 2 Reducing complex data set to a **lower dimension** by projection
- 3 PCA computes principal components
- 4 Change of basis on the data
- 5 Sometimes not all principal components needed, but only the first ones
- 6 PCA often computed using SVD of the data matrix

# Principal Component Analysis: Change of basis

- 1 Goal: identify new basis to re-express a data set
- 2 Hope: "see" a structure in the data
- 3 Given sample  $X \in \mathbb{R}^{m \times n}$
- 4 Vector space of dimension  $m$
- 5 Construct orthonormal basis

# Principal Component Analysis: Change of basis

- 1 Construct  $m \times m$  identity matrix:

$$B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix} = I \quad (86)$$

- 2 Each row is an orthonormal basis vector  $b_i$  with  $m$  components
- 3 Key tool: linearity, because vectors of the data set shall be presented with the help of linear combination of the basis vectors

# Principal Component Analysis: Change of basis

- 1 Let  $X \in \mathbb{R}^{m \times n}$  be original data set
- 2 Each column is a single sample
- 3 Let  $Y \in \mathbb{R}^{m \times n}$  related to a linear transformation  $P$
- 4  $Y$  shall become new representation
- 5 We have the basis change:

$$PX = Y \quad (87)$$

- 6  $P$  transforms  $X$  into  $Y$  and is a rotation
- 7 Rows of  $P$  are new basis vectors
- 8 Row vectors will become principal components
- 9 Question: what is a good choice of  $P$ ? How can we express the data best?

# Principal Component Analysis: Noise

- 1 Problem: measurement noise in the data
- 2 Common measure: signal-to-noise ratio (SNR), or in terms of variances:

$$SNR = \frac{\sigma_{signal}^2}{\sigma_{noise}^2} \quad (88)$$

- 3  $SNR \gg 1$  : high precision measurement
- 4  $SNR \approx 1$  : noisy data

# Principal Component Analysis: Covariance matrix

- 1 Let two sets of measurements be given:

$$A = \{a_1, a_2, \dots, a_n\}, \quad B = \{b_1, b_2, \dots, b_n\} \quad (89)$$

- 2 Variances:

$$\sigma_A^2 = \frac{1}{n} \sum_i a_i^2, \quad \sigma_B^2 = \frac{1}{n} \sum_i b_i^2 \quad (90)$$

- 3 **Covariance** between  $A$  and  $B$ :

$$\sigma_{AB}^2 = \frac{1}{n} \sum_i a_i b_i \quad (91)$$

- 4 Measures degree of linear relationship between two variables
- 5 Large values indicate correlated data
- 6  $\sigma_{AB} = 0$  if and only if  $A$  and  $B$  are uncorrelated

# Principal Component Analysis: Covariance

- 1 Before we had

$$\sigma_{ab}^2 = \frac{1}{n} ab^T \quad (92)$$

- 2 Generalization to a matrix  $X \in \mathbb{R}^{m \times n}$

- 3 Then:

$$C_X = \frac{1}{n} XX^T \quad (93)$$

- 4 Clearly,  $C_X \in \mathbb{R}^{m \times m}$

- 5 Variance of  $C_X$  on the diagonal, large values correspond to some 'interesting' structure

- 6 Covariance on off-diagonal, high magnitude correspond to high redundancy



# Principal Component Analysis: Covariance

- 1 How does optimized covariance matrix  $C_Y$  look like?
- 2 Optimal would be:  $C_Y$  is a diagonal matrix (decorrelation)
- 3 PCA assumes that all basis vectors  $\{p_1, \dots, p_m\}$  are orthonormal

# Principal Component Analysis: Simple algorithm

- 1 Select a normalized direction in the  $m$  dimensional space and where the variance in  $X$  is maximized

→ vector  $p_1$

- 2 Find next direction being orthogonal to  $p_1$  and where variance is maximized

→ vector  $p_2$

- 3 Repeat until  $m$

- 4 Resulting ordered set of  $p_i, i = 1, \dots, m$  are the principal components

# Principal Component Analysis: Solution via eigenvalue decomposition

- 1 Goal: find some orthonormal matrix  $P$  for

$$Y = PX \quad (94)$$

and where

$$C_Y = \frac{1}{n} YY^T \quad (95)$$

is a diagonal matrix

- 2 The rows of  $P$  are the principal components of  $X$
- 3 It holds

$$C_Y = \frac{1}{n} YY^T = PC_X P^T \quad (96)$$

# Principal Component Analysis: Solution via eigenvalue decomposition

- 1 Recall from linear algebra: Given a symmetric matrix  $A$ , it holds

$$A = ZDZ^T \quad (97)$$

where  $D$  is a diagonal matrix and  $Z$  contains the eigenvectors of  $A$

- 2 For PCA, the following choice is made: construct  $P$  such that each row  $p_i$  is an eigenvector of  $\frac{1}{n}XX^T$

- 3 Then:

$$C_Y = PC_XP^T = \underbrace{(PP^T)}_{=I} D \underbrace{(PP^T)}_{=I} = D \quad (98)$$

- 4 Finally, we have that the principal components of  $X$  are the eigenvectors of  $C_X = \frac{1}{n}XX^T$
- 5 And that the  $i$ .th diagonal value of  $C_Y$  is the variance of  $X$  along the vector  $p_i$

## Principal Component Analysis: Relation to SVD

- 1 Briefly recall  $X = U_X \Sigma V_X^T$ , where  $X \in \mathbb{R}^{m \times n}$
- 2 Given  $X$  as a data matrix, construct new matrix  $Y \in \mathbb{R}^{n \times m}$ :

$$Y = \frac{1}{\sqrt{n}} X^T \quad (99)$$

- 3 Assume: Each column of  $Y$  has zero mean
- 4 Indeed

$$Y^T Y = \frac{1}{n} X X^T = C_X \quad (100)$$

where again  $C_X$  is the covariance matrix of  $X$

- 5 From before we know: principal components of  $X$  are the eigenvectors of  $C_X$
- 6 SVD of  $Y$ : columns of matrix  $V_Y$  contain the eigenvectors of  $Y^T Y = C_X$
- 7 Thus: columns of  $V_Y = U_X$  are principal components of  $X$

End of Lecture 4

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 **Lecture 5: Artificial Neural Networks (ANN)**
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects



# Outline

- Artificial Neural Networks
- Weights, biases
- Cost function
- Training
- Hidden layers
- Stochastic gradient descent
- Back propagation (chain rule)
- Main literature of this lecture Higham/Higham<sup>21</sup>
- See also Bishop<sup>22</sup> and Nielsen<sup>23</sup>

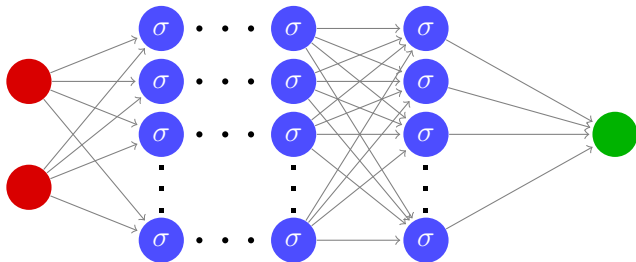
---

<sup>21</sup>C. F. Higham and D. J. Higham. “Deep Learning: An introduction for Applied Mathematicians”. In: *SIAM review* 61.4 (2019), pp. 860–891.

<sup>22</sup>Bishop, *Pattern recognition and machine learning*.

<sup>23</sup>Michael A. Nielsen. *Neural Networks and Deep Learning*. 2018. URL: <http://neuralnetworksanddeeplearning.com/>.

# Neural networks



Literature: Deepmind lecture,<sup>24</sup> [neuralnetworksanddeeplearning.com](https://neuralnetworksanddeeplearning.com)<sup>25</sup>

---

<sup>24</sup>Wojciech Czarnecki. *Lecture 2: Neural Networks Foundations*. University Lecture. 2020. URL: [https://storage.googleapis.com/deepmind-media/UCLxDeepMind\\_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf](https://storage.googleapis.com/deepmind-media/UCLxDeepMind_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf).

<sup>25</sup>Nielsen, *Neural Networks and Deep Learning*.

## Basic ingredients (clear from previous lectures)

- ① Given some labeled data  $\{x, y\}_{i=1}^N$
- ② **Activation function**: a nonlinear function, e.g. sigmoid <sup>26</sup> (Lecture 3):

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (101)$$

- ③ Scaling and shifting: weighting and biasing the input:
  - Scaling = **weighting**  $W$  = steepness of the transition zone
  - Shifting = **biasing**  $b$  = location of the transition zone
  - Example  $\sigma(3(x - 5))$ : scaling by factor 3 and shift by  $-5$
- ④ Vector-valued situation:  $z \in \mathbb{R}^m$  and  $\sigma : \mathbb{R}^m \rightarrow \mathbb{R}^m$  is treated in a component-wise fashion:

$$(\sigma(z))_i = \sigma(z_i) \quad (102)$$

---

<sup>26</sup>There is a plethora of other activation functions: hyperbolic tangent, rectified linear unit, ...

## Basic notation neural networks

- 1 (Artificial) **neuron**: outputs a single number, input from previous neurons
- 2 Current neuron receives data, multiplies by weight, adds bias, applies activation function
- 3 Collect **layers of neurons** and collect output in vector  $a \in \mathbb{R}^{s_2}$
- 4 Output from the next layer:

$$\sigma(Wa + b) \quad \text{where } W \in \mathbb{R}^{s_1 \times s_2}, b \in \mathbb{R}^{s_1} \quad (103)$$

- 5 Weights are in  $W$
- 6 Columns of  $W$ : number of neurons that yields vector  $a$  at previous layer
- 7 Rows of  $W$ : number of neurons at current layer
- 8 Biases are in  $b$  (components corresponds to current number of neurons)
- 9 Consider  $i$ th neuron, then the  $i$ th component is

$$\sigma \left( \sum_j w_{ij} a_j + b_i \right) \quad (104)$$

## Example (I)

- 1 Neural network with four layers
- 2 Input data has the values: two neurons at input layer  $l = 1$
- 3 Layer  $l = 2$  has also two neurons
- 4 Layer  $l = 3$  has three neurons
- 5 Layer  $l = 4$  has two neurons (output layer)
- 6 What are the weights and biases?
- 7  $W^{[2]} \in \mathbb{R}^{2 \times 2}$ , where [2] indicates layer  $l = 2$
- 8  $b^{[2]} \in \mathbb{R}^2$
- 9 Output from layer  $l = 2$ :

$$\sigma(W^{[2]}x + b^{[2]}) \in \mathbb{R}^2 \quad (105)$$

## Example (II)

- 1 Layer  $l = 3$  has three neurons with input from  $\mathbb{R}^2$
- 2  $W^{[3]} \in \mathbb{R}^{3 \times 2}$ ; again the three rows stand for the three current neurons and two columns are the input from two previous neurons
- 3  $b^{[3]} \in \mathbb{R}^3$
- 4 Output from layer  $l = 3$ :

$$\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) \in \mathbb{R}^3 \quad (106)$$

- 5 **Task:** quickly double-check yourself that  $W^{[3]}\sigma(W^{[2]}x + b^{[2]})$  is well-defined as matrix-vector multiplication

## Example (III)

- 1 Layer  $l = 4$  has two neurons with input from  $\mathbb{R}^3$
- 2  $W^{[4]} \in \mathbb{R}^{2 \times 3}$ ; again the two rows stand for the two current neurons and three columns are the input from three previous neurons
- 3  $b^{[4]} \in \mathbb{R}^2$
- 4 Output from layer  $l = 3$ :

$$\sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}) \in \mathbb{R}^2 \quad (107)$$

- 5 Define nonlinear function  $F : \mathbb{R}^2 \rightarrow \mathbb{R}^2$  with

$$F(x) := \sigma(W^{[4]}\sigma(W^{[3]}\sigma(W^{[2]}x + b^{[2]}) + b^{[3]}) + b^{[4]}) \quad (108)$$

- 6 Total of 23 parameters: weights  $4 + 6 + 6$ ; biases  $2 + 3 + 2$

## Example (IV)

- 1 What can we do with the previous developments?
- 2 Data points in two categories  $A$  and  $B$
- 3 Data points  $\{x^i\}_{i=1}^{10}$
- 4 Target output  $y(x^i)$  with

$$y(x^i) = \begin{cases} [1, 0]^T & \text{if } x^i \text{ is in category } A \\ [0, 1]^T & \text{if } x^i \text{ is in category } B. \end{cases} \quad (109)$$

- 5 **Cost function**, e.g. mean squared error <sup>27</sup>:

$$S := S(W^{[2]}, W^{[3]}, W^{[4]}, b^{[2]}, b^{[3]}, b^{[4]}) = \frac{1}{10} \sum_{i=1}^{10} \frac{1}{2} \|y(x^i) - F(x^i)\|_2^2. \quad (110)$$

- 6 **Training of the neural network**: determining the weights and biases
- 7 Recall: nonlinear, nonconvex, optimization problem with 23 variables

---

<sup>27</sup>For classification problems (categorical) crossentropy is also frequently used as an activation function.



# General notation of neural networks

- 1 Input layer  $l = 1$
- 2 Hidden layers  $l = 2, \dots, L - 1$
- 3 Output layer  $l = L$
- 4 **Deep learning** : many hidden layers
- 5 Per layer  $n_l$  neurons
- 6 Input dimension:  $n_1$
- 7 Output dimensions:  $n_L$
- 8 Nonlinear function  $F : \mathbb{R}^{n_1} \rightarrow \mathbb{R}^{n_L}$
- 9 Weight matrices:  $W^{[l]} \in \mathbb{R}^{n_l \times n_{l-1}}$
- 10 Element  $w_{jk}^{[l]}$  is the weight of neuron  $j$  at layer  $l$  getting output from neuron  $k$  at layer  $l - 1$

# General notation of neural networks

- 1 Given input  $x \in \mathbb{R}^{n_1}$
- 2 Output (activation)  $a_j^{[l]}$  from neuron  $j$  at layer  $l$
- 3 Then:

$$a^{[1]} = x \in \mathbb{R}^{n_1} \quad (111)$$

$$a^{[l]} = \sigma(W^{[l]} a^{[l-1]} + b^{[l]}) \in \mathbb{R}^{n_l} \quad (112)$$

for  $l = 2, 3, \dots, L$ .

- 4 Feedforward algorithm for getting final output

$$a^{[L]} \in \mathbb{R}^{n_L}. \quad (113)$$

# General notation of neural networks

- 1 Labeled data:  $\{x^i, y(x^i)\}_{i=1}^N$
- 2 Each  $x^i \in \mathbb{R}^{n_1}$  and  $y(x^i) \in \mathbb{R}^{n_L}$
- 3 Quadratic cost function:

$$S = \frac{1}{N} \sum_{i=1}^N \frac{1}{2} \|y(x^i) - a^{[L]}(x^i)\|_2^2 \quad (114)$$

- 4 This minimization problem can be solved as for instance in Lecture 3 with gradient descent; see also lectures to introduction to numerics<sup>28</sup> and of course, lectures/textbooks on numerical optimization<sup>29</sup>

---

<sup>28</sup>Richter and Wick, *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*.

<sup>29</sup>Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.

# Gradient descent

- 1 Recall that solving  $\min(S)$  means to determine all entries of  $W$  and  $b$   
→ vector  $p \in \mathbb{R}^s$  with  $s \gg 1$  (huge !)
- 2 Thus:  $S : \mathbb{R}^s \rightarrow \mathbb{R}$
- 3 For the previous example we had  $s = 23$
- 4 Gradient descent:

$$p_{k+1} = p_k - \eta \nabla S(p_k) \quad (115)$$

with the **learning rate**  $\eta > 0$

- 5 For large numbers of training points  $N$  and large number of parameters  $s$  this procedure becomes expensive

# Stochastic gradient descent

① Idea: do not go over all training points  $N$ , but choose one point randomly

② Loss function:

$$C_{x^i} = \frac{1}{2} \|y(x^i) - a^{[L]}(x^i)\|_2^2 \quad (116)$$

③ **Algorithm:** Choose  $i \in \mathbb{N}$  from  $\{1, 2, 3, \dots, N\}$ . Solve

$$p_{k+1} = p_k - \eta \nabla C_{x^i}(p_k) \quad (117)$$

④ Clearly for  $k \rightarrow \infty$  more and different training points  $i$  are chosen

⑤ Two basic methods:

- with replacement: certain  $i$  can be chosen multiple times
- without replacement: do not choose again  $i$  in order to enforce to go through (all) different training data

⑥  $N$  steps is called an **epoch**

# Stochastic gradient descent

- 1 Algorithm: Shuffle  $\{1, \dots, N\}$  into new order  $\{k_1, \dots, k_N\}$ , perform gradient descent
- 2 Variant: Choose  $\{k_1, \dots, k_m\}$  randomly from  $\{1, \dots, N\}$ , perform gradient descent
- 3 In the latter  $\{x^{k_i}\}_{i=1}^m$  is called **minibatch**
- 4 Other iterative methods such as quasi-Newton, Adam,<sup>30</sup> Newton are possible, but with the usual difficulties in approximating or computing the Hessian matrix, but with the advantage of faster convergence (superlinear, quadratic)

---

<sup>30</sup>Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

# Back propagation

- 1 It is obvious that in (stochastic) gradient descent, partial derivatives of  $S$  are required
- 2 Back propagation computes partial derivatives w.r.t. weights and biases
- 3 Let us fix the training point  $x^i$
- 4 Loss function  $C_{x^i}$  is a function of weights and biases
- 5 Short hand notation:

$$C := C_{x^i} = \frac{1}{2} \|y - a^{[L]}\|_2^2 \quad (118)$$

- 6 Output of neural network:  $a^{[L]}$
- 7 Clearly:  $C$ , i.e., weights and biases, are due to  $a^{[L]}$
- 8 Need to **propagate information** from  $a^{[L]}$  **backwards** (see also Tutorial 2 on reverse mode automatic differentiation)

# Back propagation

- 1 Introduce weighted input

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} \in \mathbb{R}^{n_l} \quad (119)$$

for  $l = 2, 3, \dots, L$

- 2 Weighted input for neuron  $j$  at layer  $l$  denoted by  $z_j^{[l]}$
- 3 With this from before (example at the beginning):

$$a^{[l]} = \sigma(z^{[l]}) \quad (120)$$

for  $l = 2, 3, \dots, L$

- 4 Partial derivative  $\delta^{[l]} \in \mathbb{R}^{n_l}$  with

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} \quad (121)$$

for  $j = 1, \dots, n_l$  and  $l = 2, \dots, L$

- 5  $\delta_j^{[l]}$  measures **sensitivity** of  $C$  w.r.t.  $z_j^{[l]}$



## Back propagation: some analysis

- 1 Define Hadamard product:  $x, y \in \mathbb{R}^n$ , then  $x \odot y \in \mathbb{R}^n$  with component-wise multiplication, i.e.,  $(x \odot y)_i = x_i y_i$
- 2 Then:

### Lemma

*It holds*

$$\delta^{[L]} = \sigma'(z^{[L]}) \odot (a^{[L]} - y) \quad (122)$$

$$\delta^{[l]} = \sigma'(z^{[l]}) \odot (W^{[l+1]})^T \delta^{[l+1]}, \quad l = 2, \dots, L-1 \quad (123)$$

$$\delta_j^{[l]} = \frac{\partial C}{\partial b_j^{[l]}}, \quad l = 2, \dots, L \quad (124)$$

$$\delta_j^{[l]} a_k^{[l-1]} = \frac{\partial C}{\partial w_{jk}^{[l]}}, \quad l = 2, \dots, L \quad (125)$$

## Back propagation: proofs

- 1 We show (122):
- 2 By  $a^{[L]} = \sigma(z^{[L]})$ , we have

$$\frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = \sigma'(z_j^{[L]}) \quad (126)$$

- 3 Also from  $C = \frac{1}{2} \|y - a^{[L]}\|_2^2$ , we obtain

$$\frac{\partial C}{\partial a_j^{[L]}} = -(y_j - a_j^{[L]}) \quad (127)$$

(chain rule and component-wise differentiation!)

- 4 Again chain rule

$$\delta_j^{[L]} = \frac{\partial C}{\partial z_j^{[L]}} = \frac{\partial C}{\partial a_j^{[L]}} \frac{\partial a_j^{[L]}}{\partial z_j^{[L]}} = (a_j^{[L]} - y_j) \sigma'(z_j^{[L]}). \quad (128)$$

## Back propagation: proofs

- 1 We show (123):
- 2 Again chain rule to compute values from  $z_j^{[l]}$  to  $\{z_k^{[l+1]}\}_{k=1}^{n_{l+1}}$
- 3 Then with previous definitions:

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \frac{\partial C}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} \quad (129)$$

- 4 From (119) we can link  $z_k^{[l+1]}$  and  $x_j^{[l]}$ :

$$z_k^{[l+1]} = \sum_{s=1}^{n_l} w_{ks}^{[l+1]} \sigma(z_s^{[l]}) + b_k^{[l+1]} \quad (130)$$

# Back propagation: proofs

- 1 Differentiation yields

$$\frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = w_{kj}^{[l+1]} \sigma'(z_j^{[l]}) \quad (131)$$

- 2 Inserting into (129) gives us

$$\delta_j^{[l]} = \frac{\partial C}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} \frac{\partial z_k^{[l+1]}}{\partial z_j^{[l]}} = \sum_{k=1}^{n_{l+1}} \delta_k^{[l+1]} w_{kj}^{[l+1]} \sigma'(z_j^{[l]}) \quad (132)$$

- 3 In compact form this means:

$$\delta_j^{[l]} = \sigma'(z_j^{[l]}) \left( (W^{[l+1]})^T \delta^{[l+1]} \right)_j \quad (133)$$

## Back propagation: proofs

- 1 We show (124):
- 2 We start from (119) and (120):

$$z_j^{[l]} = (W^{[l]} \sigma(z^{[l-1]}))_j + b_j^{[l]}. \quad (134)$$

- 3 We see from the construction that  $z^{[l-1]}$  does not depend on  $b_j^{[l]}$
- 4 Then differentiation simply yields (no chain rule this time here!)

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1 \quad (135)$$

- 5 In this next step we again use the chain rule

$$\frac{\partial C}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} = \delta_j^{[l]} \quad (136)$$

## Back propagation: proofs

- 1 We show (125):
- 2 We start again from (119):

$$z_j^{[l]} = \sum_{k=1}^{n_{l-1}} w_{jk}^{[l]} a_k^{[l-1]} + b_j^{[l]} \quad (137)$$

- 3 Since  $z_j^{[l]}$  does not depend on  $b_j^{[l]}$  we obtain by differentiation:

$$\frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = a_k^{[l-1]} \quad (138)$$

- 4 Clearly from (137) we also see

$$\frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = 0 \quad (139)$$

for  $s \neq j$

# Back propagation: proofs

- 1 Then, again chain rule:

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = \sum_{s=1}^{n_l} \frac{\partial C}{\partial z_s^{[l]}} \frac{\partial z_s^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} \frac{\partial z_j^{[l]}}{\partial w_{jk}^{[l]}} = \frac{\partial C}{\partial z_j^{[l]}} a_k^{[l-1]} = \delta_j^{[l]} a_k^{[l-1]} \quad (140)$$

- 2 Interpretation of (137) :  $j$ th neuron at layer  $l$  uses weights from  $j$ th row of  $W^{[l]}$  in a linear fashion

## Back propagation: interpretations

- 1 **Forward pass** to evaluate  $a^{[L]}$  from previous  $a^{[l]}$
- 2 This yields from (122) then  $\delta^{[L]}$
- 3 Then, using (123) we can compute  $\delta^{[l]}$  in a **backward pass**
- 4 With this, we have all ingredients to compute the partial derivatives of the cost function  $C$  w.r.t. the parameters of the neural network with the help of the relations (124) and (125)
- 5 The entire procedure is called **back propagation**



# Algorithm for neural network training

- 1  $N_{Iter}$  : maximal number of (stochastic) gradient descent iterations
- 2 For  $s = 1, \dots, N_{Iter}$
- 3   Choose  $k$  uniformly randomly from  $\{1, \dots, N\}$
- 4    $x^k$  is current training point
- 5   Set  $a^{[1]} = x^k$
- 6   For  $l = 2, \dots, L$
- 7      $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$
- 8      $a^{[l]} = \sigma(z^{[l]})$
- 9      $D^{[l]} = \text{diag}(\sigma'(z^{[l]}))$  Hadamard product
- 10   end for
- 11    $\delta^{[L]} = D^{[L]}(a^{[L]} - y(x^k))$
- 12   For  $l = L - 1, \dots, 2$
- 13      $\delta^{[l]} = D^{[l]}(W^{[l+1]})^T \delta^{[l+1]}$
- 14   end for
- 15   For  $l = L, \dots, 2$
- 16      $W_{s+1}^{[l]} := W_s^{[l]} - \eta \delta^{[l]}(a^{[l-1]})^T$  Update gradient descent
- 17      $b_{s+1}^{[l]} := W_s^{[l]} - \eta \delta^{[l]}$  Update gradient descent
- 18   end for
- 19 end for

## Algorithm: typical numerical difficulties

- 1 Choosing step length (learning rate)  $\eta$
- 2 Initializing initial guesses for weights  $W$  and  $b$
- 3 Choosing the optimal size of the neural network (bigger networks generally more accurate, but more expensive to train)
- 4 Robustness and efficiency of overall stochastic gradient algorithm
- 5 Very flat or steep gradients of  $S$
- 6 Choosing the size of the minibatches
- 7 Stopping criterion  $N_{Iter}$  (train sufficiently; but avoid overfitting)

End of Lecture 5

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem**
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Borel measures
- Some theorems from analysis and functional analysis
- Universal approximation: general version with proof
- Key assumptions: density result and discriminatory of activation function
- Proofs that sigmoid and ReLU satisfy universal approximation property

# Main literature

- Guilhoto<sup>31</sup>
- Hornik et al.<sup>32</sup>
- Werner<sup>33</sup> (basics in FA and function spaces; german)
- Brezis<sup>34</sup> (also FA; in english)

---

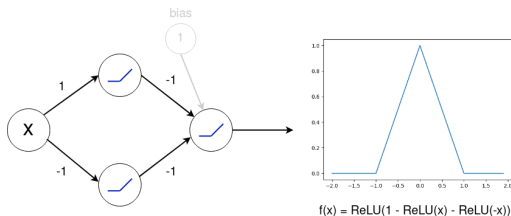
<sup>31</sup>Leonardo Ferreira Guilhoto. *An Overview Of Artificial Neural Networks for Mathematicians*. 2018. URL: <https://math.uchicago.edu/~may/REU2018/REUPapers/Guilhoto.pdf>.

<sup>32</sup>Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks 2.5* (1989), pp. 359–366. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8). URL: <https://www.sciencedirect.com/science/article/pii/0893608089900208>.

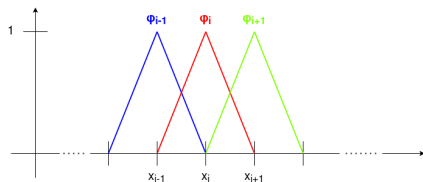
<sup>33</sup>Dirk Werner. *Funktionalanalysis*. Springer, 2018.

<sup>34</sup>H. Brezis. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Springer, 2011. DOI: <https://doi.org/10.1007/978-0-387-70914-7>.

# Universal Approximation Theorem



Creating a hat function



Hat functions form basis of piecewise linear functions

[see also (linear) Finite Elements]

## Definition

Let  $T$  be a metric (or topological) space and  $\Sigma$  the  $\sigma$  algebra of Borel sets (i.e., the induced  $\sigma$  algebra obtained from the open sets). The  $\sigma$  algebra on  $T$  is a collection of subsets of  $T$  satisfying: it includes  $T$ , the complement is closed, it is closed under countable unions and countable intersections. A measure on  $\mu$  on  $\Sigma$  is called **Borel measure**. The space  $(T, \Sigma)$  is denoted as measurable space or Borel space. A positive Borel measure  $\mu$  is regular if

- $\mu(C) < \infty$  for all compact  $C$
- For all  $A \in \Sigma$  it holds

$$\mu(A) = \sup\{\mu(C) : C \subset A, C \text{ compact}\} \quad (141)$$

---

<sup>35</sup>Dirk Werner, Funktionalanalysis, Springer 2018



# Lebesgue Dominated Convergence Theorem<sup>36</sup>

## Theorem

Let  $X$  be a measure space and  $\mu$  a Borel measure on  $X$ . Let  $g : X \rightarrow \mathbb{R}$  be an  $L^1$  regular function and  $\{f_n\}$  be a sequence of measurable functions from  $X \rightarrow \mathbb{R}$  such that

$$|f_n(x)| \leq g(x) \quad (142)$$

for all  $x \in X$  and  $\{f_n\}$  converges pointwise to a function  $f$ . Then  $f$  is integrable and

$$\lim_{n \rightarrow \infty} \int f_n(x) d\mu(x) = \int f(x) d\mu(x). \quad (143)$$

---

<sup>36</sup>Haim Brezis; Functional Analysis, Sobolev Spaces, and Partial Differential Equations, Springer, 2010.

# Hahn-Banach Theorem<sup>37 38</sup>

Theorem (Geometric form; separation of convex sets through linear continuous functionals)

*Let  $V$  be a normed space and  $A, B \subset V$  be two non-empty, closed, disjoint and convex subsets and one being compact. Then, there exists a continuous linear functional  $f \neq 0$  and some  $\alpha \in \mathbb{R}$  and  $\epsilon > 0$  such that  $f(x) \leq \alpha - \epsilon$  for any  $x \in A$  and  $f(y) \geq \alpha + \epsilon$  for any  $y \in B$ .*

## Corollary

*Let  $V$  be a normed vector space over  $\mathbb{R}$  and  $U \subset V$  be a linear subspace such that  $\bar{U} \neq V$ . Then, there exists a continuous linear mapping  $f : V \rightarrow \mathbb{R}$  with  $f(x) = 0$  for any  $x \in U$  and  $f \not\equiv 0$ .*

---

<sup>37</sup>Haim Brezis; Functional Analysis, Sobolev Spaces, and Partial Differential Equations, Springer, 2010.

<sup>38</sup>Dirk Werner; Funktionalanalysis, Springer 2018

# Riesz Representation Theorem<sup>39 40</sup>

## Definition (Space of continuous functions)

Let  $\Omega$  be a metric (or even only topological) space. Think of subsets of  $\mathbb{R}$  as simplest example. Then, we define

$$C(\Omega) := \{f : \Omega \rightarrow \mathbb{R} \mid f \text{ is continuous} \}.$$

## Theorem (Riesz representation theorem)

Let  $\Omega$  be a subset of  $\mathbb{R}^n$  and  $F : C(\Omega) \rightarrow \mathbb{R}$  be a linear functional. Then, there exists a signed Borel measure  $\mu$  on  $\Omega$  such that for any  $f \in C(\Omega)$ , we have

$$F(f) = \int_{\Omega} f(x) d\mu(x). \quad (144)$$

Often the duality product notation  $F(f) = \langle F, f \rangle$  is used.

---

<sup>39</sup>Haim Brezis; Functional Analysis, Sobolev Spaces, and Partial Differential Equations, Springer, 2010.

<sup>40</sup>Dirk Werner; Funktionalanalysis, Springer 2018

# Set of neural network functions

- We concentrate on **one single hidden layer!**
- In total: three layers
- In the following we denote the sigmoid function  $\sigma$  (Lecture 5) by simply  $f$  (reason: otherwise confusing with  $\sigma$  algebra)

## Definition

For  $f : \mathbb{R} \rightarrow \mathbb{R}$  an activation function, we define

$$\Sigma_n(f) = \text{span}\{f(y \cdot x + \theta) \mid y \in \mathbb{R}^n, \theta \in \mathbb{R}\}. \quad (145)$$

For the motivation of  $f(y \cdot x + \theta)$ , we refer the reader to Lecture 5. The  $\Sigma_n(f)$  contains all functions which can be calculated by a neural network with one single hidden layer.

# Universal approximator

## Definition

Let  $\Omega$  be a topological space and  $f : \mathbb{R} \rightarrow \mathbb{R}$ . We call a neural network with activation function  $f$  a **universal approximator** on  $\Omega$  if  $\Sigma_n(f)$  is dense in  $C(\Omega)$ .

## Definition (Density)

A metric space  $U$  is dense in  $V$  when  $\bar{U} = V$ , where  $\bar{U}$  denotes the closure of  $U$ .

- Example 1: The set  $\mathbb{Q}$  of rational numbers is dense in  $\mathbb{R}$ .
- Example 2: The space of polynomial functions  $P_k$  of degree  $k$  is dense in  $C[a, b]$  (Weierstrass approximation theorem).

# Discriminatory functions

## Definition

Let  $n \in \mathbb{N}$ . We call an activation function  $f : \mathbb{R} \rightarrow \mathbb{R}$   $n$ -discriminatory if the only signed Borel measure  $\mu$  with

$$\int f(y \cdot x + \theta) d\mu(x) = 0 \quad \forall y \in \mathbb{R}^n, \quad \theta \in \mathbb{R} \quad (146)$$

is the zero set measure.

## Definition

We say an activation function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is discriminatory if it is  $n$ -discriminatory for any  $n$ .

# Sigmoid and ReLU

## Definition

Recall from Lecture 5 and 3 the sigmoid function. A function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is called sigmoid if it satisfies the following properties:

$$\lim_{x \rightarrow \infty} f(x) = 1 \quad (147)$$

$$\lim_{x \rightarrow -\infty} f(x) = 0. \quad (148)$$

## Definition

The Rectified Linear Unit (ReLU) is  $ReLU : \mathbb{R} \rightarrow \mathbb{R}$  and is defined by

$$ReLU(x) := \max(0, x). \quad (149)$$

# Main theorem: universal approximator

## Theorem

*Let  $f$  be a continuous discriminatory function. Then, a neural network with  $f$  as activation function is a universal approximator.*

In words:

- Any activation function will lead to a network with universal approximation property if and only if this function is not a polynomial almost everywhere
- Almost everywhere is meant in the sense of Lebesgue measure theory



## Proof (I).

- 1 We prove this by contradiction
- 2 Assume that  $\Sigma_n(f)$  is not dense in  $C(I_n)$ , where  $I_n := [0, 1]^n$
- 3 According to the definition of density (see before), we have  $\overline{\Sigma_n(f)} \neq C(I_n)$ .
- 4 This fulfills the assumption of the Hahn-Banach theorem, namely for  $\bar{U} \neq V$ , there exists some continuous linear functional  $F : C(I_n) \rightarrow \mathbb{R}$  such that  $F \neq 0$ , but  $F(g) = 0$  for any  $g \in \overline{\Sigma_n(f)}$
- 5 The Riesz representation theorem yields (why? Because we have a linear continuous functional  $F$ ) that there exists some Borel measure  $\mu$  such that

$$F(g) = \int_{I_n} g(x) d\mu(x) \quad \text{for all } g \in C(I_n). \quad (150)$$

## Proof (II).

- 1 According to our definition, for  $y$  and  $\theta$  the function  $f(y \cdot x + \theta)$  is in  $\Sigma_n(f)$
- 2 Moreover, this holds for any  $y$  and any  $\theta$  and therefore it holds even that  $f(y \cdot x + \theta)$  is in  $\overline{\Sigma_n(f)}$
- 3 Since by assumption  $f$  is discriminatory, this yields that for all  $y \in \mathbb{R}^n$  and  $\theta \in \mathbb{R}$ , we have

$$\int f(y \cdot x + \theta) d\mu(x) = 0 \quad (151)$$

from which follows  $\mu = 0$ .

- 4 Thus:  $F(g) = 0$  for any  $g \in C(I_n)$
- 5 Thus  $F \equiv 0$
- 6 But this contradicts the Hahn-Banach theorem and the assertion is shown. □

# Sigmoid is discriminatory

- With the previous theorem, we now need to check the assumptions, namely that a specific activation function (e.g., sigmoid, ReLU, ...) are continuous and discriminatory.

## Lemma

*All bounded Borel measurable sigmoid functions are discriminatory.*

## Proof (I).

- 1 Clearly, sigmoid is continuous.
- 2 We show now that sigmoid is discriminatory.
- 3 Let  $f$  be a bounded, Borel measurable sigmoid function
- 4 Assume that for a given  $\mu$ , we have

$$\int_{I_n} f(y \cdot x + \theta) d\mu(x) = 0 \quad \text{for all } y \in \mathbb{R}^n, \quad \theta \in \mathbb{R} \quad (152)$$

- 5 **Goal** is to show that  $\mu = 0$

## Proof (II).

- 1 Define a function  $\gamma$  and use  $\lambda, \phi \in \mathbb{R}$  such that

$$\gamma(x) = \lim_{\lambda \rightarrow \infty} f(\lambda(y \cdot x + \theta) + \phi) = \begin{cases} 1 & y \cdot x + \theta > 0 \\ f(\phi) & y \cdot x + \theta = 0 \\ 0 & y \cdot x + \theta < 0 \end{cases} \quad (153)$$

- 2 The limits are clear and due to the definition of the sigmoid function (see this lecture before and also Lecture 3 and 5)
- 3 Employ the Lebesgue Dominated Convergence Theorem and (152):

$$\int_{I_n} \gamma(x) d\mu(x) = \lim_{\lambda \rightarrow \infty} \int_{I_n} f(\lambda(y \cdot x + \theta) + \phi) d\mu(x) = 0 \quad (154)$$

## Proof (III).

- 1 To proceed we use the construction and limits of  $\gamma(x)$
- 2 Define the three sets:

$$H^+ := \{x \in I_n \mid y \cdot x + \theta > 0\} \quad (155)$$

$$H := \{x \in I_n \mid y \cdot x + \theta = 0\} \quad (156)$$

$$H^- := \{x \in I_n \mid y \cdot x + \theta < 0\} \quad (157)$$

$$(158)$$

- 3 Then from (154), we obtain

$$\int_{I_n} \gamma(x) d\mu(x) = \int_{H^+} 1 d\mu(x) + \int_H f(\phi) d\mu(x) + \int_{H^-} 0 d\mu(x) \quad (159)$$

$$= 1 \mu(H^+) + f(\phi) \mu(H) \quad (160)$$

$$= 0 \quad (\text{zero due to the assumption}) \quad (161)$$

for any  $y, \theta$

## Proof (IV).

- ① Moreover, the previous is true for any  $\phi$ . Due to the definition of  $f$ , we have

$$f(\phi) \rightarrow 1 \quad \text{for } \phi \rightarrow \infty \quad (162)$$

- ② Then:

$$1 \mu(H^+) + f(\phi)\mu(H) \xrightarrow{\phi \rightarrow \infty} \mu(H^+) + \mu(H) = 0 \quad (163)$$

- ③ Also:

$$f(\phi) \rightarrow 0 \quad \text{for } \phi \rightarrow -\infty \quad (164)$$

- ④ Then:

$$1 \mu(H^+) + f(\phi)\mu(H) \xrightarrow{\phi \rightarrow -\infty} \mu(H^+) = 0 \quad (165)$$

## Proof (V).

- ① We now consider a bounded Lebesgue integrable functional, namely  $F : L^\infty \rightarrow \mathbb{R}$ :

$$F(h) := \int_{I_n} h(y \cdot x) d\mu(x) \quad (166)$$

- ② Recall from Analysis and/or functional analysis:
- The space  $L^\infty(E)$ , with  $E \subset \mathbb{R}^n$  of a non-empty, bounded, Lebesgue measurable set, contains the essentially bounded measurable functions up to a Lebesgue measure zero.
  - Thus: for  $f : E \rightarrow \mathbb{R}$ , we have the norm

$$\|f\|_{L^\infty(E)} = \operatorname{ess\,sup}_{x \in E} |f(x)| \quad (167)$$

- ③ For this reason, the above integral is well-defined per definition



## Proof (VI).

- 1 Take now the indicator function  $1_{[\theta, \infty)}$  on the half-open interval  $[\theta, \infty)$ , we obtain

$$F(1_{[\theta, \infty)}) = \int_{I_n} 1_{[\theta, \infty)}(y \cdot x) d\mu(x) = \mu(H^+) + \mu(H) = 0 \quad (168)$$

- 2 Same for the open interval  $(\theta, \infty)$ :

$$F(1_{(\theta, \infty)}) = \int_{I_n} 1_{(\theta, \infty)}(y \cdot x) d\mu(x) = \mu(H^+) = 0 \quad (169)$$

where  $\mu(H) = 0$ .

## Proof (VII).

- ① Due to linearity of  $F(h)$ , we clearly obtain for any indicator function  $h$ :

$$F(h) = 0. \quad (170)$$

- ② Thus, for any simple function (indicator functions are simple!<sup>41</sup>)  
 $F(h) = 0$

- ③ Since simple functions are dense in  $L^\infty(\mathbb{R})$ , it follows that

$$F \equiv 0 \quad (171)$$

- ④ We recall our goal is to show that  $\mu = 0$

---

<sup>41</sup>See Analysis textbooks, e.g., (H. Amann and J. Escher. *Analysis III*. Birkhäuser, 2008. URL: <https://link.springer.com/book/10.1007/978-3-7643-8884-3>)

## Proof (VIII).

- 1 We just have established that for simple functions  $F \equiv 0$
- 2 Take now specific realizations of simple functions, namely sine and cosine, which are clearly in  $L^\infty$
- 3 Then:

$$F(\cos) + iF(\sin) = \int_{I_n} (\cos(y \cdot x) + i \sin(y \cdot x)) d\mu(x) \quad (172)$$

$$= \int_{I_n} e^{iy \cdot x} d\mu(x) \quad (173)$$

$$= 0 \quad (\text{since } F \equiv 0) \quad (174)$$

- 4 This is true for all  $y \in \mathbb{R}^n$
- 5 We recognize that the above is the Fourier transform of  $\mu$  is zero
- 6 Due to the definition of  $e^{iy \cdot x}$  (first part of integrand strictly positive) this is only possible for  $\mu = 0$
- 7 **This finishes the proof.** □

# *ReLU* is discriminatory

## Lemma

*The ReLU function is 1-discriminatory.*

## Proof (I).

- 1 Clearly, ReLU is continuous.
- 2 We show now that ReLU is discriminatory.
- 3 Let  $\mu$  be a signed Borel measure
- 4 Assume for all  $y \in \mathbb{R}$  and  $\theta \in \mathbb{R}$ , it holds

$$\int \text{ReLU}(yx + \theta) d\mu(x) = 0 \quad (175)$$

- 5 **Goal:** we show that  $\mu = 0$

## Proof (II).

- 1 Procedure: construct sigmoid bounded, continuous function (i.e., Borel measurable)
- 2 Idea: subtract two *ReLU* functions (with different parameters) to construct such a sigmoid function  $f$
- 3 Afterward apply just previous lemma for  $f$
- 4 To this end, we use

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \in [0, 1] \\ 1 & x > 1 \end{cases} \quad (176)$$

- 5 With this, any function of the form  $g(x) = f(y \cdot x + \theta)$  with  $y \neq 0$  can be constructed via

$$g(x) = \text{ReLU}(y \cdot x + \theta_1) - \text{ReLU}(y \cdot x + \theta_2) \quad (177)$$

with  $\theta_1 = -\theta/y$  and  $\theta_2 = (1 - \theta)/y$

## Proof (III).

- ① For  $y = 0$ , we set

$$g(x) = f(\theta) = \begin{cases} \operatorname{ReLU}(f(\theta)) & f(\theta) \geq 0 \\ -\operatorname{ReLU}(-f(\theta)) & f(\theta) \leq 0 \end{cases} \quad (178)$$

- ② Then, for any  $y \in \mathbb{R}$  and  $\theta \in \mathbb{R}$ , we obtain

$$\int f(y \cdot x + \theta) d\mu(x) = \int (\operatorname{ReLU}(y \cdot x + \theta_1) - \operatorname{ReLU}(y \cdot x + \theta_2)) d\mu(x) \quad (179)$$

$$= \int \operatorname{ReLU}(y \cdot x + \theta_1) d\mu(x) - \int \operatorname{ReLU}(y \cdot x + \theta_2) d\mu(x) \quad (180)$$

$$= 0 - 0 = 0 \quad (181)$$

- ③ Since  $f$  is discriminatory by the previous lemma, we obtain the result  $\mu = 0$ , and finish the proof.  $\square$

# Density result

## Lemma

*If  $\Sigma_1(f)$  is dense in  $C([0, 1])$ , then the extension  $\Sigma_n(f)$  is dense in  $C([0, 1]^n)$ .*



## Proof (I).

- ① We first notice that

$$\overline{\text{span}\{g(a \cdot x) \mid a \in \mathbb{R}^n, g \in C([0, 1])\}} = C([0, 1]^n) \quad (182)$$

- ② This means that for any  $h \in C([0, 1]^n)$  and  $\epsilon > 0$  and  $\{g_k\}_k \subset C([0, 1])$  such that for some  $N \in \mathbb{N}$

$$\left| h(x) - \sum_{k=1}^N g_k(a_k \cdot x) \right| < \frac{\epsilon}{2} \quad (183)$$

- ③ Use assumption  $\Sigma_1(f)$  is dense in  $C([0, 1])$  and consider each  $g_k(a_k \cdot x)$ , we obtain

$$\left| g_k(a_k \cdot x) - \sum_{i=1}^{N_k} f(y_{k,i} \cdot x + \theta_{k,i}) \right| < \frac{\epsilon}{2N} \quad (184)$$

for some sequences  $\{y_{k,i}\}_{i \in \mathbb{N}}$  and  $\{\theta_{k,i}\}_{i \in \mathbb{N}}$ .

## Proof (II).

- 1 We proceed with typical arguments as classical proofs in Analysis, which means that we now apply the triangle inequality and obtain  $\epsilon/2 + \epsilon/2 = \epsilon$
- 2 In detail:

$$\left| h(x) - \sum_{k=1}^N \sum_{i=1}^{N_k} f(y_{k,i} \cdot x + \theta_{k,i}) \right| < \left| h(x) - \sum_{k=1}^N g_k(a \cdot x) \right| + N \frac{\epsilon}{2N} \quad (185)$$

$$< \frac{\epsilon}{2} + \frac{\epsilon}{2} \quad (186)$$

$$= \epsilon. \quad (187)$$

- 3 This finishes the proof. □

# Main theorem

We have thus established with the previous results:

## Theorem

*Neural networks with sigmoid or ReLU as activation functions are universal approximators.*

- As further reading, we mention that in,<sup>42</sup> these results are further extended to more hidden layers with  $l \geq 4$  (input, output and  $l - 2$  hidden layers).

---

<sup>42</sup>Guilhoto, *An Overview Of Artificial Neural Networks for Mathematicians*.

End of Lecture 6

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)**
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

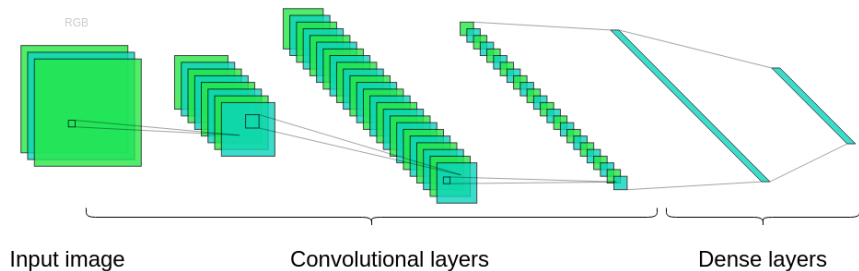
- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Convolution
- Example of  $2D$  images
- Convolutional neural networks (CNN)
- Conv layer, pooling layer, fully connected layer
- Kernel and filters
- Example of a ConvNet architecture

# Convolutional Neural Network



# Convolution of functions (I)

## Definition

If  $f, g : \mathbb{R}^n \rightarrow \mathbb{K}$  with  $\mathbb{K} \in \{\mathbb{R}, \mathbb{C}\}$  are two measurable functions, we define the **convolution** of  $f$  and  $g$  as

$$(f * g)(x) := \int_{\mathbb{R}^n} f(y)g(x - y) dy \quad (188)$$

for almost all  $x \in \mathbb{R}^n$ .

Note that the convolution is commutative, i.e. we have

$$(f * g)(x) = \int_{\mathbb{R}^n} f(y)g(x - y) dy \quad (189)$$

$$\stackrel{z:=x-y}{=} \int_{\mathbb{R}^n} f(x - z)g(z) dz \quad (190)$$

$$= (g * f)(x) \quad (191)$$



## Convolution of functions (II)

- 1 The convolution is moreover **measurable** and **well-defined in  $L^1$**
- 2 Let  $\mathbb{T} := \{z \in \mathbb{C} \mid |z| = 1\} = \{e^{it} \mid 0 \leq t \leq 2\pi\}$
- 3 Consider the normalized Lebesgue measure  $dt/2\pi$
- 4 Let  $f, g \in L^1(\mathbb{T})$  be complex-valued
- 5 Set

$$(f * g)(e^{is}) = \int_0^{2\pi} f(e^{it})g(e^{i(s-t)}) \frac{dt}{2\pi} \quad (192)$$

- 6 Then

$$\int_0^{2\pi} |(f * g)(e^{is})| \frac{ds}{2\pi} \leq \int_0^{2\pi} \int_0^{2\pi} |f(e^{it})| |g(e^{i(s-t)})| \frac{dt}{2\pi} \frac{ds}{2\pi} \quad (193)$$

$$= \int_0^{2\pi} |f(e^{it})| \frac{dt}{2\pi} \int_0^{2\pi} |g(e^{i(s-t)})| \frac{ds}{2\pi} \quad (\text{Fubini}) \quad (194)$$

$$\leq \int_0^{2\pi} |f(e^{it})| \frac{dt}{2\pi} \|g\|_{L^1} = \|f\|_{L^1} \|g\|_{L^1} \quad (195)$$

which shows  $f * g \in L^1(\mathbb{T})$ .

# Convolution of matrices in terms of an example

- 1 Given a matrix  $F \in \mathbb{R}^{p \times p}$  (called **filter** for our purposes) where  $p \in \mathbb{N}$  is the size of a patch
- 2 Some input image is black/white, with 1 for black and 0 for white pixels
- 3 Let for instance  $p = 3$
- 4 An example for a patch matrix (the image!) is

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad (196)$$

## Convolution of matrices in terms of an example

- 1 We now calculate the convolution of  $P$  and  $F$
- 2 Value will be higher, the more similar is  $F$  to  $P$
- 3 The **filter matrix** contains the effect you want to achieve
- 4 As an example let the filter matrix be given by

$$F = \begin{pmatrix} 0 & 2 & 3 \\ 2 & 4 & 1 \\ 0 & 3 & 0 \end{pmatrix} \quad (197)$$

- 5 The **convolution** or **kernel** is defined similar to the **Frobenius scalar product**:

$$P * F = \begin{pmatrix} 0 \cdot 0 & 1 \cdot 2 & 0 \cdot 3 \\ 1 \cdot 2 & 1 \cdot 4 & 1 \cdot 1 \\ 0 \cdot 0 & 1 \cdot 3 & 0 \cdot 0 \end{pmatrix} \rightarrow \sum_{ij=1}^p P_{ij} F_{ij} = 12 \quad (198)$$

- 6 Remark: We continue this example in various pieces throughout this lecture

## Motivating example<sup>43</sup>

- Tracking location of a spaceship with a laser sensor
- $x(t) \in \mathbb{R}$  is a laser measurement of the spaceship's position at time  $t$
  
- Problem: laser sensor measurements are noisy
- Goal: more accurate spaceship's position by averaging measurements

---

<sup>43</sup>Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*.  
<http://www.deeplearningbook.org>. MIT Press, 2016.

## Motivating example: continuous convolution

- Weighted average should give more weight to newer measurements
- Use a weighting function  $w : \mathbb{R} \rightarrow \mathbb{R}$
- Apply weighted average of measurements at every time instance
- Smoothed measurement of spaceship's position is given by the continuous convolution of  $x$  and  $w$ , i.e.

$$s(t) := (x * w)(t) = \int_{\mathbb{R}} x(\tau)w(t - \tau) d\tau \quad (199)$$

## Motivating example: discrete convolution

- Laser measurements are usually not continuous, but discrete, e.g. one measurement per second

⇒ Now only  $x, w : \mathbb{Z} \rightarrow \mathbb{R}$

- Smoothed measurement of spaceship's position is given by the discrete convolution of  $x$  and  $w$ , i.e.

$$s(t) := (x * w)(t) = \sum_{k=-\infty}^{\infty} x(k)w(t - k) \quad (200)$$

## Motivating example: computations

- Get 200 noisy measurements of spaceship's position
- Average over 5 neighboring measurements by using filter

$$w = \left[ \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5}, \frac{1}{5} \right] \quad (201)$$

- Smoothed measurement then given by

$$s(0) = \frac{1}{5}x(0) \quad (202)$$

$$s(1) = \frac{1}{5}x(0) + \frac{1}{5}x(1) \quad (203)$$

$$s(2) = \frac{1}{5}x(0) + \frac{1}{5}x(1) + \frac{1}{5}x(2) \quad (204)$$

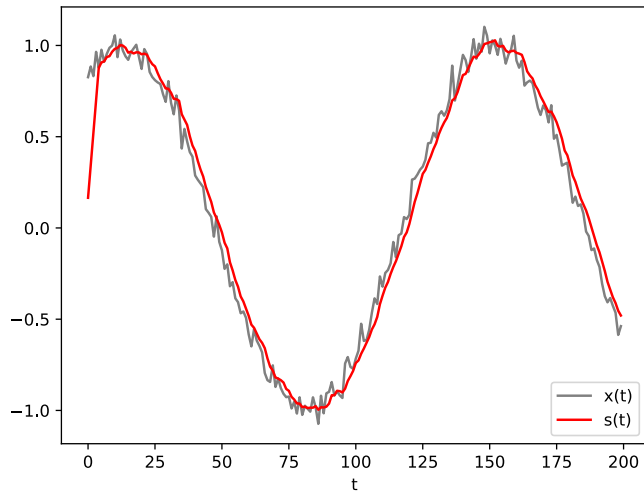
$$s(3) = \frac{1}{5}x(0) + \frac{1}{5}x(1) + \frac{1}{5}x(2) + \frac{1}{5}x(3) \quad (205)$$

$$s(4) = \frac{1}{5}x(0) + \frac{1}{5}x(1) + \frac{1}{5}x(2) + \frac{1}{5}x(3) + \frac{1}{5}x(4) \quad (206)$$

$$s(5) = \frac{1}{5}x(1) + \frac{1}{5}x(2) + \frac{1}{5}x(3) + \frac{1}{5}x(4) + \frac{1}{5}x(5) \quad (207)$$

$$\vdots \quad (208)$$

# Motivating example: result of computations





# Convolution on 2D images

- Input is a two dimensional image  $I$
- Two dimensional kernel  $K$
- Convolution of  $I$  with  $K$  is defined as

$$S(i, j) := (I * K)(i, j) = \sum_m \sum_n I(m, n)K(i - m, j - n) \quad (209)$$

- Convolution is commutative and hence

$$S(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \quad (210)$$

## Cross-correlation on 2D images

- Commutativity of convolution is useful for mathematical proofs, but not necessary for neural networks
- Hence many neural network libraries use the cross-correlation defined as

$$S(i, j) := (K * I)(i, j) = \sum_m \sum_n I(i + m, j + n) K(m, n) \quad (211)$$

- Many machine learning libraries use the cross-correlation but call it convolution and we will also use this naming convention

# Applications of convolution in image processing

- noise reduction
- blurring
- sharpness
- edge detection



Figure: Original image

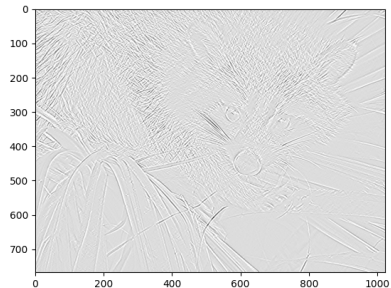
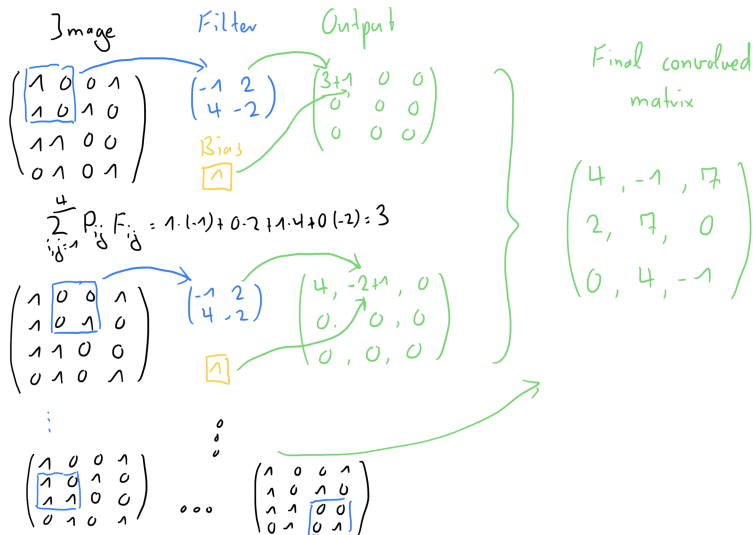


Figure: Edges detected with Sobel filter

# Convolution: Example of a filter convolving across an image

- 1 We continue the example with the patch  $P$  and the filter  $F$
- 2 The filter matrix (one for each filter in each layer) and bias values can be trained (as parameters) being solved with gradient descent and backpropagation
- 3 Typically *ReLU* is used in the hidden layers
- 4 Activation function for output layer depends on the specific problem statement's goal
- 5 Let  $s_l$  filters be given in each layer  $l$
- 6 Output of convolutional layer  $l$  will consist of  $s_l$  matrices, one for each filter

# Convolution: Example of a filter convolving across an image



# Convolutional Neural Networks (CNN)

- Before: pre-defined kernels extract a given feature from the image
  - Now: neural network learns the kernels to extract meaningful features from the image
  - Convolutional networks = neural networks that use convolution instead of normal matrix multiplication in at least one of their layers
  - Three main types of layers to build CNN: convolutional layer, pooling layer, fully-connected layer
  - Advantage of CNN: forward function becomes more efficient to implement
- Reducing amount of parameters in the NN

## Example architecture

- 1 Assume input image of size  $32 \times 32 \times 3$  (width, height, depth)
- 2 **Input:** of  $32 \times 32 \times 3$  holds the raw pixel values of the image of 32 width, 32 height and 3 color channels  $R, G, B$
- 3 **Conv layer:** computes the output of neurons that are connected to small regions in the input layer (instead to all neurons of the previous layer - see Lecture 5 for a classical ANN). This results in a volume of  $[32 \times 32 \times 12]$ , where the 12 stands for **learnable filters**
- 4 ReLU (Lectures 3 and 6) applies an element-wise activation function
- 5 **Pool** performs a downsampling operation in the spatial dimensions (width, height) resulting for instance in  $[16 \times 16 \times 12]$
- 6 **FC (fully connected):** computes class scores, resulting in  $[1 \times 1 \times 10]$  where each of the 10 numbers correspond to a class score (final output layer)

# Convolutional Layer

- 1 Conv layer parameters: set of **learnable filters**
- 2 Every filter is spatially small (going over width and height)
- 3 However extends to full depth of input volume
- 4 **Example:** typical filter on first layer may have size  $5 \times 5 \times 3$  (5 pixels width and height each, and 3 color channels)
- 5 **Forward pass:** **slide (i.e., convolve)** each filter across width and height of input volume
- 6 Compute dot products between entries of filter and input
- 7 Yields a two-dimensional activation map that gives the response of that filter at each spatial position
- 8 "Network will learn filters"



## Convolutional Layer: local connectivity

- 1 For high-dimensional input (many neurons) such as images, the computational cost becomes high
- 2 **Idea:** Connect each neuron only to a local input volume
- 3 **Spatial extent** is a hyperparameter called **receptive field**
- 4 The previous procedure is also known as **filter size**
- 5 Extent of connectivity along depth axis is equal to the depth of input volume
- 6 Spatial asymmetry : width and height are treated equally, but not the depth dimension
- 7 Connections are local in  $2D$  space, but always full along the entire depth of input volume

## Convolutional Layer: local connectivity, example

- 1 Suppose input volume has size  $[32 \times 32 \times 3]$
- 2 Size of filter (receptive field) for instance  $5 \times 5$
- 3 Each neuron on the Conv layer has weights to a  $[5 \times 5 \times 3]$  region in the input volume
- 4 Total of  $5 \cdot 5 \cdot 3 = 75$  weights plus 1 bias parameter
- 5 We notice that extent along the depth axis must be 3 because it is the depth of the input volume

# Convolutional Layer: spatial arrangement

- 1 So far only discussion how each neuron is connected to input volume
- 2 Now, we explain how many output neurons are dealt with
- 3 Three hyperparameters: **depth, stride, zero-padding**
- 4 Depth of the output: number of filters
- 5 Stride: sliding the filter. When stride is 1 then we move the filters one pixel at a time. For 2 the filters jump 2 pixels. Will result in smaller output volumes spatially
- 6 Size of zero-padding: pad the input volume with zeros around the border. Always to control spatial size of the output volumes

## Convolutional Layer: parameter sharing

- ① Controls number of parameters
- ② **Example:** Given for instance  $55 \cdot 55 \cdot 96 = 290\,400$  neurons in the first Conv layer
- ③ Connecting to a local field of size  $[11 \times 11 \times 3]$ , each has  $11 \cdot 11 \cdot 3 = 363$  weights and 1 bias
- ④ Total:  $290\,400 \cdot 364 = 105\,705\,600$  parameters on the first Conv layer  
→ very high number

# Convolutional Layer: reasoning for its name

- 1 Idea: having  $(x, y)$  use at different position  $(x_2, y_2)$  the same parameters
  - 2 For instance take 2-dimensional depth slice
  - 3 Recall:  $[55 \times 55 \times 96]$  has 96 depth slices each of size  $[55 \times 55]$
  - 4 Constrain now neurons in each depth slice to the same weights and bias
- **Parameter sharing**
- 5 If now all neurons in a single depth slice use the same weight vector, then the forward pass of the Conv layer can be computed as a **convolution**
- **Convolutional layer**
- 6 Sets of weights are called **filter** or a **kernel**, which is convolved with the input

## Convolutional Layer: summary

- 1 Given a volume  $W_1 \times H_1 \times D_1$ , where  $W_1$  (width),  $H_1$  (height),  $D_1$  (depth)
- 2 Four hyperparameters (numerical parameters): number of filters  $K$ , spatial extend  $F$ , stride  $S$  and amount of zero-padding  $P$
- 3 Results into a volume of size  $W_2 \times H_2 \times D_2$ , where

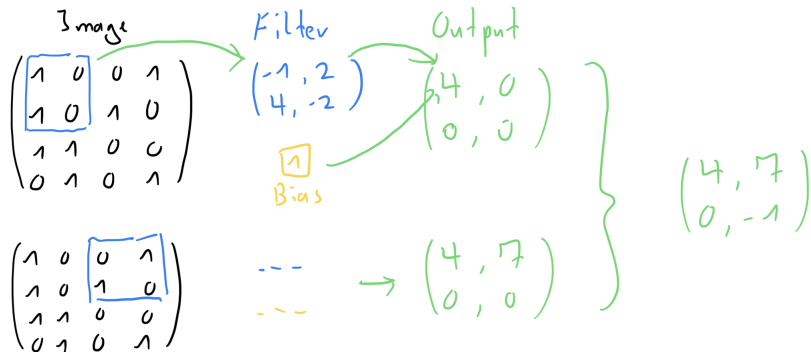
$$W_2 = \frac{(W_1 - F + 2P)}{S + 1} \quad (212)$$

$$H_2 = \frac{(H_1 - F + 2P)}{S + 1} \quad (213)$$

$$D_2 = K \quad (214)$$

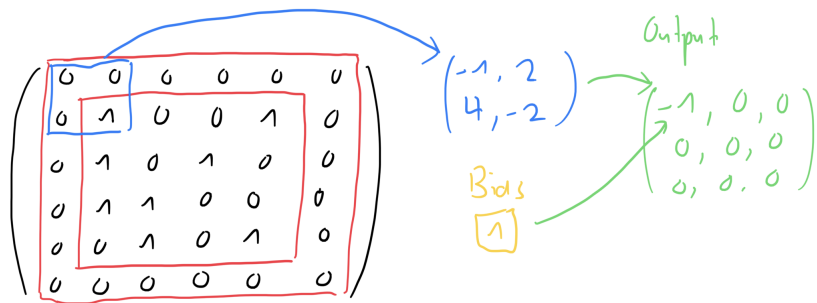
- 4 Via parameter sharing:  $F \cdot F \cdot D_1$  weights per filter; with a total of  $F \cdot F \cdot D_1 \cdot K$  weights and  $K$  biases
- 5 Output volume:  $d$ -th depth slice (size  $W_2 \times H_2$ ) is constructed via  $d$ -th filter over the input volume with a stride  $S$  and offset by  $d$ -th bias

## Example (cont'd) of a filter convolving across an image with stride 2



- Stride: the shift of the blue square
- Previously (and default is stride 1): here it is 2

## Example (cont'd) of a filter convolving across an image with stride 2 and padding layer of size 1



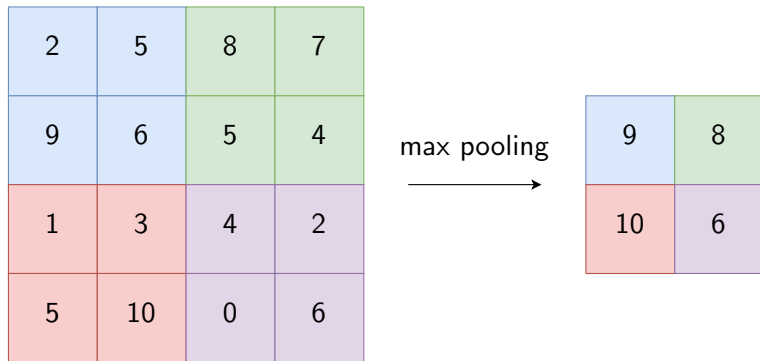
- Padding: the red-colored pad around the original image
- Previously pad was 0; here it is now 1.
- Larger pads  $\geq 2$  also possible



# Pooling Layer

- 1 From time to time insert a pooling layer between successive Conv layers
- 2 Goal: reduce progressively spatial size of the representation for reducing amount of parameters
- 3 Control overfitting
- 4 Pooling layer operates independently on every depth slice of the input
- 5 Resizes spatially using MAX operation ( $L^2$ -norm pooling possible as well)
- 6 Attempts in the literature to get rid of pooling by using a larger stride in the CONV layer from time to time
- 7 Discarding pooling also in variational autoencoders (VAEs) or generative adversarial networks (GANs)

## Pooling Layer: Example



## Pooling Layer: Algorithm

- 1 Given a volume of size  $W_1 \times H_1 \times D_1$
- 2 Two hyperparameters (numerical parameters): spatial extend  $F$  and stride  $S$
- 3 Results in a volume of size  $W_2 \times H_2 \times D_2$  where

$$W_2 = \frac{(W_1 - F)}{S + 1} \quad (215)$$

$$H_2 = \frac{(H_1 - F)}{S + 1} \quad (216)$$

$$D_2 = D_1 \quad (217)$$

- 4 Introducing zero parameters since fixed function of input is computed
- 5 No zero-padding

## Fully-connected (FC) Layer

- 1 Neurons in FC have full connections to all activations from previous layer (see also Lecture 5, classical ANN)
- 2 Activations are computed as shown in Lecture 5, namely matrix multiplication with a bias offset
- 3 Difference to CONV layer: in CONV, the neurons are connected only to a local region from the previous input and many neurons share parameters
- 4 Technically, still the usual computations (as in Lecture 5 ANN) must be computed though

# ConvNet architectures

- 1 Summarizing from before: CNN commonly consist of only three layers: CONV, POOL, FC
- 2 Often: stack CONV-ReLU layers, followed by POOL
- 3 Repeat this pattern until image has been merged spatially to a small size
- 4 At some point transition to FC
- 5 Last FC yields the output such as class scores
- 6 **Algorithm:**

INPUT  $\rightarrow$  [[CONV  $\rightarrow$  ReLU]\* $N$   $\rightarrow$  POOL?]\* $M$   $\rightarrow$  [FC  $\rightarrow$  ReLU]\* $K$   $\rightarrow$  FC (218)

where \* indicates repetition, ? being an optional feature,  
 $N, M, K \geq 0$ , where usually  $N \leq 3$  and  $K \leq 2$

End of Lecture 7

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)**
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline lecture 8

- Motivation for recurrent neural networks (RNN)
- How RNNs work
- Difficulties: vanishing gradient, exploding gradient, long-term dependencies
- Gated RNNs: long-short-term memory, gated recurrent unit



# Main literature lecture 8

- Ava Soleimany<sup>44</sup>
- Richard Socher<sup>45</sup>
- Pascanu, Mikolov, Bengio<sup>46</sup>
- Andriy Burkov,<sup>47</sup> Chapter 6, 2019

---

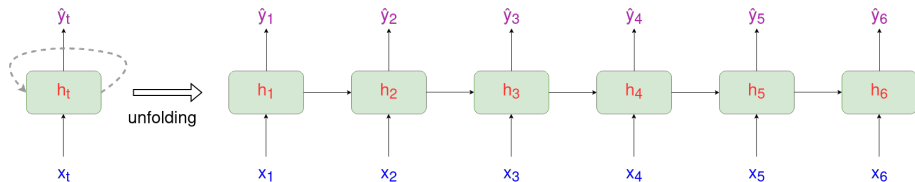
<sup>44</sup>Ava Soleimany. *MIT 6.S191 Introduction to Deep Learning: Deep Sequence Modeling*. MITDeepLearning. 2021.

<sup>45</sup>Richard Socher. *Lecture 8: Recurrent Neural Networks*. CS224d, Deep NLP. 2016.

<sup>46</sup>Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].

<sup>47</sup>Burkov, *The hundred-page machine learning book*.

# Recurrent Neural Networks: Motivating slide



# Recurrent Neural Networks: introduction

- 1 **Recurrent neural networks: abbrev. RNN**
- 2 Employed to **label, classify, or generate sequences**
- 3 Application: **Sequences of words, language models, text processing, speech processing**
- 4 **Example:** predict the next word:

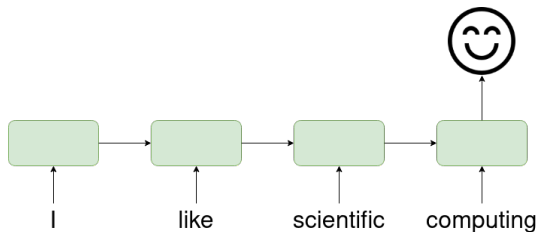
‘This morning I took my dog for a ...’ (... = walk)

→ Classical neural networks require too much memory!

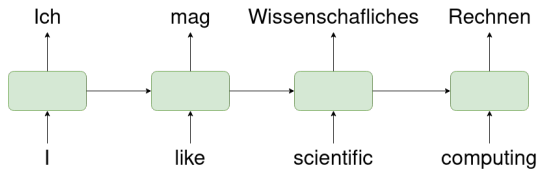
- 5 Sequence is a matrix: each row is a feature vector
- 6 **Labeling:** predict a class for each feature vector
- 7 **Classification:** determine a class to a specific sequence
- 8 **Generating:** output another (related to the input) sequence

# Recurrent Neural Networks: applications

Sentiment analysis:



Translation:



# Sequence modeling: design criteria

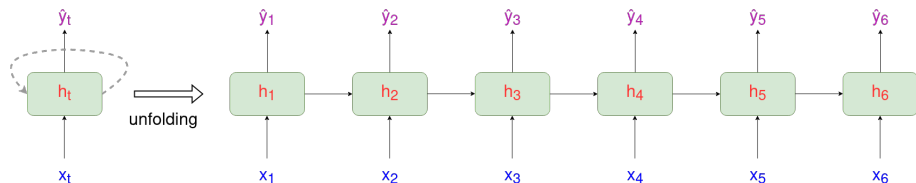
- 1 The key purpose of RNNs is to deal with **sequences**
- 2 To model sequences, design criteria are:
  - Handling variable length sequences
  - Track long-term dependencies (Example: long sentences remembering the first words)
  - Maintain information about the order (Example: order of words in a sentence)
  - Share parameters across the sequence
- 3 Recall example: predicting words in a (possibly long) sentence:

'I like scientific ...'

# Recurrent Neural Networks: specific features

- ① RNNs are **not feed-forward**
- ② RNNs contain **loops** (related to '**time steps**')
- ③ For instance language models:
  - RNNs condition NN on all previous words
  - RAM (of the computer) only scales with number of words

# Recurrent Neural Networks: diagram



Principle idea:

$$\hat{y}_t = f(x_t, h_{t-1}) \quad (219)$$

where

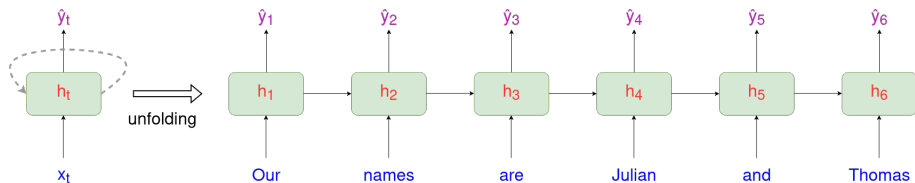
- $\hat{y}_t$ : output at time step  $t$
- $x_t$ : input at time step  $t$
- $h_{t-1}$ : memory from time steps  $1, 2, \dots, t - 1$

# Input sequence

- 1 First (left) layer receives feature vector as input
- 2 Second layer receives the output from first layer as input
- 3 Training example via matrix in which each row is a feature vector
- 4 Matrix represented as a sequence of vectors  $X \in \mathbb{R}^M$ , where  $M$  is the length of the input sequence
- 5 **Example for a word sentence as input sequence:** 'Our names are Julian and Thomas': here  $M = 6$
- 6  $X = [x_1, \dots, x_t, \dots, x_M]$
- 7 Therein  $x_t$  represents the feature vector at **position  $t$**
- 8 **Example (cont'd):**  $x_1 = \text{Our}, \dots, x_M = \text{Thomas}$



# RNN with input sequence



# Update procedure

- 1 Neural network reads input feature vectors sequentially in order of time steps  $t$
- 2 Index  $t$  denotes position, or better **time step**
- 3 **Update procedure:**

$$h_t = \sigma(W^{hh}h_{t-1} + W^{hx}x_t) \quad (220)$$

$$\hat{y}_t = \text{softmax}(W^S h_t) \quad (221)$$

- 4 Note that the sigmoid function  $\sigma$  can also be exchanged by  $\tanh$
- 5 Interpretation of  $\hat{y}_t$ :

$$\hat{y}_{t,j} = P(x_{t+1} = v_j | x_1, \dots, x_t) \quad (222)$$

$\hat{y}_{t,j}$  represents the probability that the next word in the sequence is the  $j$ .th word in the dictionary, given that we know all words until time step  $t$ .

# Softmax function

- 1 For classification, typically the output of the neural network is being plugged into the **softmax** function:

$$\text{softmax}(z) := [s^1, \dots, s^D] \quad (223)$$

where

$$s^j := \frac{\exp(z^j)}{\sum_{k=1}^D \exp(z^k)} \quad (224)$$

- 2 Softmax function is generalization of the sigmoid function (lecture 3)
- 3 Property of softmax:

$$\sum_{j=1}^D s^j = 1 \quad (225)$$

$$s^j > 0 \quad \text{for all } j \quad (226)$$

## Basic idea for two recurrent layers (cont'd)

- 1 What needs to be computed as parameters?
- 2 Values of  $W^{hh}$ ,  $W^{hx}$ ,  $W^S$  are determined with gradient descent and back propagation (lecture 5)
- 3 Difficulty: since RNN contain time steps (loops), we need **backpropagation through time**
- 4 Principal difficulties in training RNNs:
  - sigmoid, tanh and softmax suffer from **vanishing gradient**. Why? Due to sequential nature of input, backpropagation has to **unfold** the network over time (see also previous illustrations). Longer input sequences (sentences with many words), yield deeper unfolded networks.
  - Handling **long-term dependencies**: feature vectors from the beginning sequence can be 'forgotten'. State  $h_t$  (recall the 'memory') is influenced by more recent input. Problem in long sentences (sequences)

# Exploding gradient problem

- 1 Gradients very high (large values)
- 2 Solutions to cope with exploding gradients:
  - Gradient clipping (fix the gradient to a maximum value)
  - Short pseudo-code (algorithm):

$$\hat{g} = \frac{\partial E}{\partial W} \quad (227)$$

$$\text{if } \|\hat{g}\| \geq c \quad \text{where } c = \text{treshhold, then} \quad (228)$$

$$\hat{g} := \frac{c}{\|\hat{g}\|} \hat{g} \quad (229)$$

$$\text{end if} \quad (230)$$

- $L_1$  or  $L_2$  regularization

# Vanishing gradient problem (general problem)

- 1 Principal problem: during backpropagation, the gradients become too small (possible underflow, machine precision problem - Lecture 1)
- 2 If values numerically zero, then the affected parameters are actually not updated any more
- 3 No change in those weights or those biases
- 4 Due to (negative) accumulation<sup>48</sup>, gradient decreases exponentially
- 5 Earlier layers are updated only slowly or not at all
- 6 Difficultly becomes more severe in RNN due to time step loops

---

<sup>48</sup>see also Numerik 1 (Richter and Wick, *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*) in which order numerical algorithms should be carried out

## Vanishing gradient problem: RNN

- 1 Consider the following (simplified) RNN formulation:

$$h_t = Wf(h_{t-1}) + W^{hx}x_t \quad (231)$$

$$\hat{y}_t = W^S f(h_t) \quad (232)$$

where  $W$ ,  $W^{hx}$  and  $W^S$  are matrices,  $h_t$  the state at time step  $t$  and  $x_t$  the feature vector, and  $f(\cdot)$  functions carrying out the previous linear combinations within the RNN

- 2 Total error:

$$E = \sum_{t=1}^M E_t \quad (233)$$

- 3 Sensitivity of error w.r.t. recurrent weight matrix  $W$  (compare also the lecture 5, proofs of backpropagation):

$$\frac{\partial E}{\partial W} = \sum_{t=1}^M \frac{\partial E_t}{\partial W} \quad (234)$$

# Vanishing gradient problem: RNN

- 1 Chain rule (again similar to lecture 5, proofs backpropagation):

$$\frac{\partial E}{\partial W} = \sum_{t=1}^M \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \quad (235)$$

- 2 Investigate the partial derivative (sensitivity)

$$\frac{\partial h_t}{\partial h_k} \quad (236)$$

- 3 It holds

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \quad (237)$$

- 4 Component-wise notation of the previous expression is a matrix



## Vanishing gradient problem: RNN

- 1 Recall:  $h_t = Wf(h_{t-1}) + W^{hx}x_t$
- 2 Compute Jacobian matrix, i.e. each element  $\frac{\partial h_{j,m}}{\partial h_{j-1,n}}$ :

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^t W^T \text{diag}(f'(h_{j-1})) \quad (238)$$

where  $W^T$  is the transpose matrix

- 3 Let  $f'(h_{j-1})$  be bounded, i.e.,

$$\|f'(h_{j-1})\| \leq \beta_h \quad (239)$$

- 4 Furthermore, let  $\lambda_1$  the largest eigenvalue of the recurrent weight matrix  $W$
- 5 A sufficient condition for the following (vanishing gradient) is to assume that

$$\beta_W := \lambda_1 < \frac{1}{\beta_h} \quad (240)$$

## Vanishing gradient problem: RNN

- 1 As usual for such an analysis, compute norm (some compatible matrix norm!) for each factor

$$\left\| \frac{\partial h_j}{\partial h_{j-1}} \right\| = \|W^T\| \|\text{diag}(f'(h_{j-1}))\| \leq \beta_W \beta_h \quad (241)$$

where  $\beta_W$  and  $\beta_h$  are the upper bounds for  $\|W^T\|$  and  $\|\text{diag}(f'(h_{j-1}))\|$ , respectively.

- 2 Apply the previous result to  $\frac{\partial h_t}{\partial h_k}$ . Then:

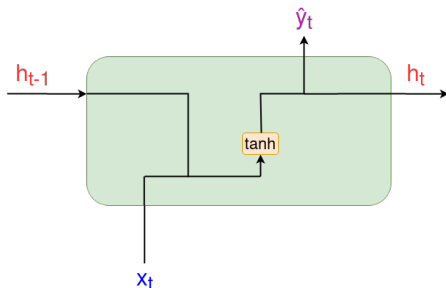
$$\left\| \frac{\partial h_t}{\partial h_k} \right\| = \left\| \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}} \right\| \leq (\beta_W \beta_h)^{t-k} \quad (242)$$

where the last inequality follows from (241)

- 3 Clearly, for the above assumption, namely  $\beta_W \beta_h < 1$ , then for large  $t - k$ , the norm goes exponentially fast to zero
- 4 For  $\beta_W \beta_h > 1$ , exploding gradients arise

# Long short-term memory (LSTM) networks

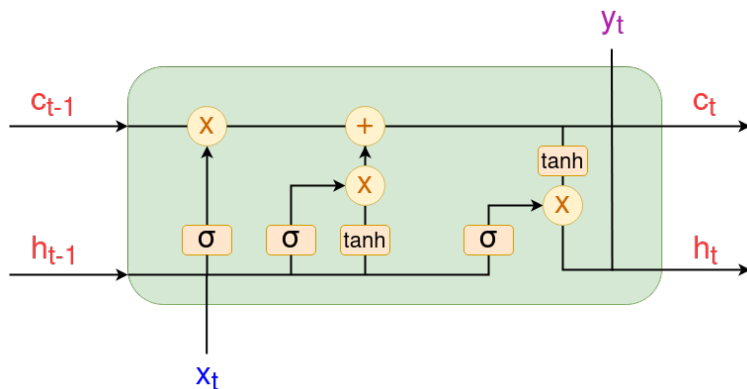
- 1 Why? Coping with the previously mentioned problems of long-term dependencies ('forgetting' information in long sequences)
  - 2 In the previous standard RNN, repeating modules contain a **simple computation node**
- Given  $h_{t-1}$  and  $x_t$ , use activation function  $\tanh$  and compute  $h_t$  and output  $y_t$  at time step  $t$ ; Then proceed to  $t \rightarrow t + 1$



# Long short-term memory (LSTM) networks

- 1 In LSTMs, **computational blocks** are introduced in order to **control flow of information**
- 2 Information can be added or removed using so-called **gates**
- 3 Gates work with the help of sigmoid neural net layers and pointwise multiplications
- 4 LSTM work according to the principle:  
**forget, store, update, output**
- 5 To this end, a control function  $c_t$  is introduced at time step  $t$
- 6 Procedure: Given  $h_{t-1}$ ,  $x_t$ , and  $c_{t-1}$ , compute state  $h_t$ , output  $y_t$ , and control  $c_t$  at time step  $t$
- 7 Then, proceed to  $t \rightarrow t + 1$

# Long short-term memory (LSTM) networks



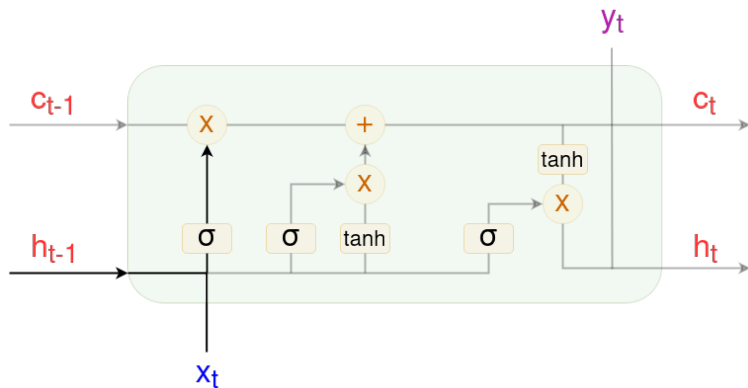
- $c_t$ : cell state, long-term memory
- $h_t$ : hidden state, short-term memory

# Long short-term memory (LSTM) networks

- 1 **Forget** irrelevant information from the previous state  $h_{t-1}$
- 2 **Store** relevant new information to current state
- 3 **Update** cell state values  $c_t$
- 4 **Output gate** controls what information is given to next time step
- 5 Gradient flow (backpropagation) is not interrupted through gates!

# Long short-term memory (LSTM) networks

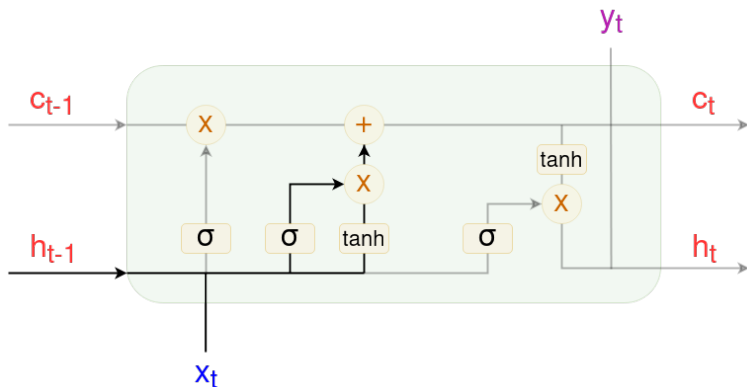
## 1. Forget



Forget irrelevant aspects of old state

# Long short-term memory (LSTM) networks

## 2. Store

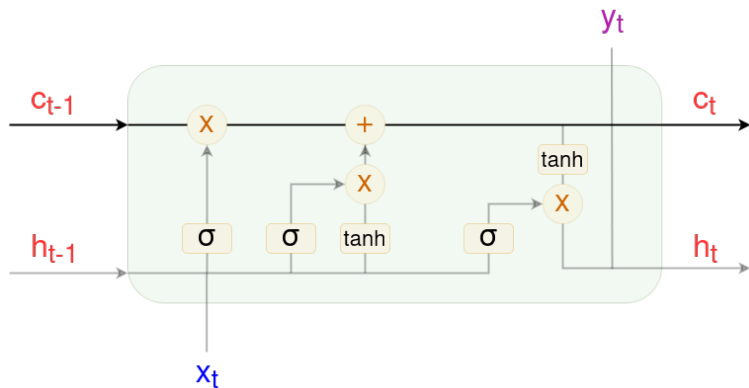


Store important new information in cell state



# Long short-term memory (LSTM) networks

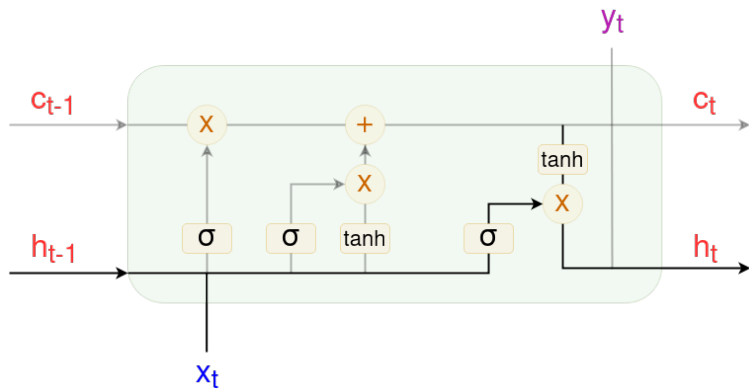
## 3. Update



Selectively update cell state values

# Long short-term memory (LSTM) networks

## 4. Output gate



Control information for next time step

# Gated recurrent unit (GRU)

① Explain here the concept of a **minimal gated unit**

→ Memory cell and forget gate

② GRU = LSTM with a forget gate

③ Fewer parameters than LSTM

# Gated recurrent unit (GRU)

Algorithm:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \quad (243)$$

$$\hat{h}_t = \tanh(W_h x_t + U_h (f_t \odot h_{t-1}) + b_h) \quad (244)$$

$$h_t = (1 - f_t) \odot h_{t-1} + f_t \odot \hat{h}_t \quad (245)$$

where  $\odot$  is still the Hadamard product, and where

- $x_t$ : input vector
- $h_t$ : output vector
- $\hat{h}_t$ : candidate activation vector
- $f_t$ : forget vector
- $W_f, U_f, W_h, U_h$ : weight matrices
- $b_f, b_h$ : bias vectors

End of Lecture 8

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer**
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Sequence to Sequence Learning (Seq2Seq)
- Attention
- Transformer: Illustrative description
- Transformer: Mathematical description
- Applications

# Main literature

- 1 Initial paper introducing Transformers: Vaswani et al.<sup>49</sup>
- 2 YouTube video on Transformer paper:  
<https://www.youtube.com/watch?v=iDulhoQ2pro>
- 3 Five pages of "precise mathematical definition of the transformer model": <https://homes.cs.washington.edu/~thickstn/docs/transformers.pdf>
- 4 Stanford lecture on Transformers:  
<http://web.stanford.edu/class/cs224n/slides/cs224n-2021-lecture09-transformers.pdf>
- 5 "Math-guided tour of Transformer":  
<http://www.columbia.edu/~jsl2239/transformers.html>
- 6 Illustrative introduction:  
<https://jalammr.github.io/illustrated-transformer/>

---

<sup>49</sup>Ashish Vaswani et al. "Attention is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.



# Motivation

① Particular NN for natural language processing tasks

→ Transformer

② **Sequence to Sequence: Seq2Seq**

③ NN that **transforms** a **given sequence** of elements (e.g., words, Lecture 8) **into another sequence**

# Seq2Seq

- 1 Seq2Seq good in translation
- 2 **Example:** Transform sequence of words in one language (e.g., German) into another language (e.g., Spanish)
- 3 Usual choice: Long-Short-Term-Memory (LSTM) models (Lecture 8)
- 4 LSTM: remembering or forgetting information (both on purpose)
- 5 Order of words is important - achieved by LSTM

# Seq2Seq

- 1 Seq2Seq: **Encoder** and **Decoder** structure
- 2 **Encoder**: takes input sequence  $x \in \mathbb{R}^n$
- 3 Mapping into higher dimensional space with fixed-length internal representation, i.e., vector  $v \in \mathbb{R}^n$
- 4 Now  $v$  given to decoder
- 5 **Decoder** generates output sequence  $y \in \mathbb{R}^m$

## Example

- 1 **Encoder:** languages German and French  
**'Wir heißen Thomas und Julian'**
- 2 **Decoder:** languages English and French  
**'Our names are Thomas and Julian'**
- 3 **Goal:** translate a sentence (sequence) from German to English
- 4 Encoder translates German to French  
**'Nous appelons Thomas et Julian'**
- 5 Decoder takes French and transforms it to English
- 6 Both the Encoder and Decoder could be trained to become fluent in French such that translation improves  
**'Wir heißen, Nous appelons, We are'**
- 7 In terms of NN, the encoder and decoder both use LSTMs, respectively
- 8 Here is not yet any Transformer ... !

# Attention

- ① The idea in transformers needs another ingredient: **Attention**
- ② Attention: looks at input sequence and decides at each step which **other parts** of this sequence are **also important**

## Example (cont'd)

- ① We continue the language example (German, French, English)
- ② Encoder does not only perform a pure translation of given sequence
- ③ Also: writing down **keywords** that are important for the semantics
- ④ **Both information** (translation plus keywords) are given to decoder
- ⑤ Decoder now knows which parts of the translation are **specifically important in terms of the context**
- ⑥ In terms of NN: we give **weights** to inputs

# Transformer

- 1 Transformer: **novel architecture** (2017)<sup>50</sup>
- 2 Follows encoder-decoder structure
- 3 Transformer: two parts Encoder and Decoder, but does **not** use RNN, i.e., LSTM

---

<sup>50</sup>Vaswani et al., “Attention is All You Need”.

# Transformer: illustrative explanation

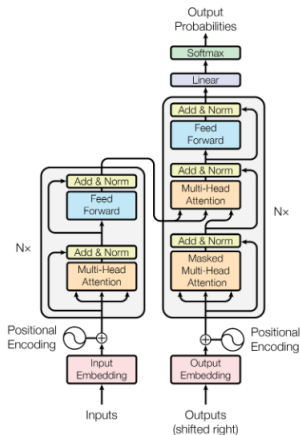


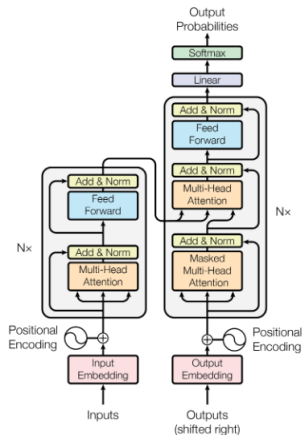
Image source: Figure 1 from original Transformer paper<sup>51</sup>

<sup>51</sup>Vaswani et al., "Attention is All You Need".



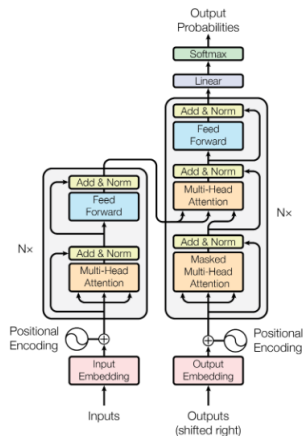
# Transformer: Encoder stack

- 1 Encoder (left): composed of modules;  $N$  identical layers
- 2 Modules consist mainly of **two sublayers**:
  - **Multi-Head-Attention**
  - Feedforward layers (fully connected; Lecture 5)
- 3 Inputs and outputs are written into  $n$ -dimensional spaces
- 4 **Positional encoding** (no RNN !), but need to remember 'certain aspects' (later more)



# Transformer: Decoder stack

- 1 Decoder (right): also composed of modules;  $N$  identical layers
- 2 Modules consist mainly of **three sublayers**:
  - **Masked Multi-Head-Attention**
  - Multi-Head-Attention
  - Feedforward layers (fully connected; Lecture 5)
- 3 Inputs and outputs are written into  $n$ -dimensional spaces
- 4 **Positional encoding** (no RNN !), but need to remember 'certain aspects' (later more)



# Multi-Head attention

- 1 **Attention:** mapping a query  $Q$  and a set of key-value pairs  $K$  and  $V$  to an output

- 2 Formula:

$$A(Q, K, V) = s \left( \frac{QK^T}{\sqrt{d_k}} \right) V \quad (246)$$

where  $A$  denotes 'Attention' and  $s$  is the softmax activation function

- 3 Most commonly used: additive attention and dot-product (multiplication)
- 4  $Q$  is a matrix that contains **query**, i.e., vector representation of one word in the sentence (sequence)
- 5  $K$  are all **keys**, i.e., vector representation of all words in the sequence (sentence)
- 6  $V$  are the **values**; also vector representations of all the words in the sequence
- 7  $d_k$  is the dimension of keys and queries;  $d_v$  is the dimension of values

# Multi-Head attention

- 1 Multi-head attention: self-attention layer
- 2 Interpretation as a look-up table
- 3 Given from before  $Q, K, V$
- 4 Let  $x_i$  be an element from the given sequence
- 5 Take  $Q(x_i)$  and test compatibility with the key  $K(x_j)$  for each  $x_j, j = 1, \dots, n$
- 6 Use inner (dot) product:  $(Q(x_i), K(x_j))_2$
- 7 If  $(Q(x_i), K(x_j))_2 \gg 1$ , then good match
- 8 If  $(Q(x_i), K(x_j))_2 \approx 0$ , not a good match
- 9 Do this for all  $x_i, i = 1, \dots, n$  and build afterward sum (later more)

# Fully connected feedforward layer per object

- 1 So far multi-attention heads
- 2 Each of the layers in encoder and decoder contains pointwise feedforward layer

→ Small NN

- 3 Identical parameters for each position
- 4 ReLU activation layer in between
- 5 Formula:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (247)$$

# Positional encoding

- ① **Problem:** basically, a transformer operates on a collection of  $n$  unordered,  $d$  dimensional features (no order!)
  - ② Transformer model does not contain recurrence or convolution to determine the position of tokens (elements)
  - ③ Order of sequence requires information of relative or absolute position of the tokens in the sequence
- **Positional encodings** to input embeddings at the bottoms of the encoder and decoder stacks

# Positional encoding: realization

- 1 Realization with sine and cosine functions:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}}) \quad (248)$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}}) \quad (249)$$

where  $pos$  is the position and  $i$  the dimension

- 2 Further methods (e.g., learning positional encoding) and discussions in<sup>52</sup>

---

<sup>52</sup>John Thickstun. *The Transformer Model in Equations*. Preprint. 2019.

# Transformer: mathematical description

- 1 **Transformer block** is building module in Transformer
- 2 Let  $\theta$  be as usual a parameter related to the entries of the weight matrices  $W$  (of the NN) and some further parameters to be specified later
- 3 Parametrized function class: transforms collection of  $n$  objects in  $\mathbb{R}^d$  to another collection of  $n$  objects in  $\mathbb{R}^d$ :

$$f_{\theta} : \mathbb{R}^{n \times d} \rightarrow \mathbb{R}^{n \times d} \quad (250)$$

- 4 Input  $x \in \mathbb{R}^{n \times d}$ : collection of  $n$  objects, each with  $d$  features
- 5 Example: sequence of length  $n$  of  $d$  vectors
- 6 Output  $z \in \mathbb{R}^{n \times d}$ : (of course) same structure (dimension) as input (clear because we want to transform input to output; think of language translation)
- 7 Then:

$$f_{\theta}(x) = z \quad (251)$$



# Transformer: mathematical description

- ① **Transformer:** composition of  $L$  transformer blocks:

$$f_{\theta_L} \circ \dots \circ f_{\theta_1}(x) \in \mathbb{R}^{n \times d} \quad (252)$$

- ② Hyperparameters (as usual of the algorithms):

$$d, k, m, H, L \quad (253)$$

- ③ Common settings:

$$d = 512, \quad k = 64, \quad m = 2048, \quad H = 8, \quad L = 6 \quad (254)$$

- ④ Attention weights:

$$\theta := \theta(W, \gamma, \beta) \quad (255)$$

## Multi-headed self-attention (1st sublayer)

- 1 For each  $h$  define **attention head** ( $H$  sets of equations)
- 2 Vectors: **queries, keys, values:**

$$Q^h(x_i) = W_{h,q}^T x_i, \quad K^h(x_i) = W_{h,k}^T x_i, \quad V^h(x_i) = W_{h,v}^T x_i \quad (256)$$

with the matrices:  $W_{h,q}, W_{h,k}, W_{h,v} \in \mathbb{R}^{d \times k}$

- 3 **Attention weights:**

$$\alpha_{i,j}^h = s_j \left( \frac{(Q^h(x_i), K^h(x_j))_2}{\sqrt{k}} \right) \quad (257)$$

where  $(\cdot, \cdot)_2$  is the Euclidian scalar product and  $s(\cdot)$  as usual the softmax function

- 4 With the matrix  $W_{c,h} \in \mathbb{R}^{k \times d}$ , we have the weighted sum:

$$u'_i = \sum_{h=1}^H W_{c,h}^T \sum_{j=1}^n \alpha_{i,j}^h V^h(x_j) \quad (258)$$

# Training and 2nd sublayer

- 1 Apply next the layer norm:

$$u_i = \text{LayerNorm}(x_i + u'_i, \gamma_1, \beta_1) \quad (259)$$

with  $\gamma_1, \beta_1 \in \mathbb{R}^d$

- 2 **Second sublayer** with feedforward NN:

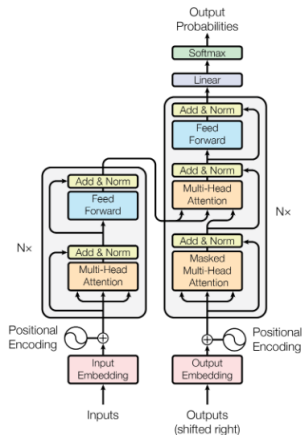
$$z'_i = W_2^T \text{ReLU}(W_1^T u_i) \quad (260)$$

with  $W_1 \in \mathbb{R}^{d \times m}$  and  $W_2 \in \mathmathbb{R}^{m \times d}$

- 3 Finally

$$z_i = \text{LayerNorm}(u_i + z'_i, \gamma_2, \beta_2) \quad (261)$$

with  $\gamma_2, \beta_2 \in \mathbb{R}^d$



# Layer norm

- 1 Define **layer norm**:

$$\text{LayerNorm}(z, \gamma, \beta) := \gamma \frac{z - \mu_z}{\sigma_z} + \beta \quad (262)$$

for  $z \in \mathbb{R}^k$  and  $\gamma, \beta \in \mathbb{R}^k$

- 2 Within, we have the mean

$$\mu_z = \frac{1}{k} \sum_{i=1}^k z_i \quad (263)$$

- 3 And the standard deviation

$$\sigma_z = \sqrt{\frac{1}{k} \sum_{i=1}^k (z_i - \mu_z)^2} \quad (264)$$

# Relation of Transformers to RNNs or CNNs

- 1 For large values of  $x$ , the function class  $f_\theta$  trained with a classical NN would be subject to **overfitting**
- 2 Transformers, as RNN or CNN, try to overcome overfitting by controlling parts of  $f_\theta$
- 3 Number of parameters in fully connected NN scales linearly with the size of  $x$  (see previous lectures)
- 4 Number of parameters in a Transformer is independent of the number  $n$  of the inputs (holds also for RNN, but they differ in their concepts how to achieve this)

## Transformer: generating text with GPT-2

- GPT: generative pre-trained transformer

The following sample has been generated by GPT-2:

**Human prompt:** In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

**AI completion:** The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. ...

# Transformer: working with code with OpenAI Codex

---

## Algorithm 1 Turn comments into code

---

```
1: """
2: Ask the user for their name and say "Hello"
3: """
4:
5: name = input("What is your name? ")
6: print("Hello " + name)
```

---

More demos can be found on [OpenAI's website](#) **Remark:** Note that language models with applications in the real world, like GPT or Codex, have a few billion parameters and are trained on datasets of a few gigabytes. It has been estimated that the cloud computing cost alone for these models can range from a few thousand dollars up to a few million dollars for the larger models.<sup>53</sup>

---

<sup>53</sup>Or Sharir, Barak Peleg, and Yoav Shoham. "The Cost of Training NLP Models: A Concise Overview". In: *CoRR* abs/2004.08900 (2020). arXiv: 2004.08900. URL: <https://arxiv.org/abs/2004.08900>.

End of Lecture 9



## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Markov chains and Markov decision processes (MDP)
- Reward function
- Set of states and set of actions
- Optimal value function and optimal policy
- Convergence
- Deep reinforcement learning

# Main literature of this class

- 1 A mathematical introduction to RL:

<https://cims.nyu.edu/~donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf>

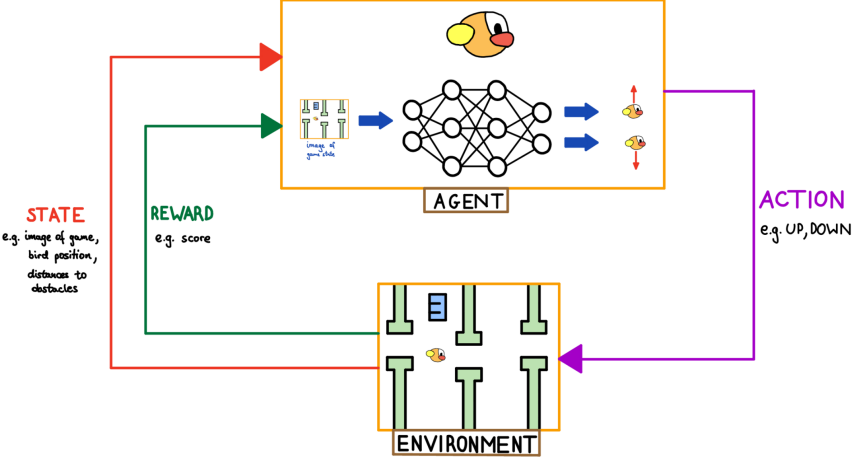
- 2 Reinforcement learning: a survey

<https://arxiv.org/pdf/cs/9605103.pdf>

- 3 Mathematical Blogpost for DEEP RL [https:](https://deeplearningmath.org/deep-reinforcement-learning.html)

[//deeplearningmath.org/deep-reinforcement-learning.html](https://deeplearningmath.org/deep-reinforcement-learning.html)

# Motivational example



# Motivation

- 1 Recall supervised learning and unsupervised learning from Lecture 1
- 2 **Reinforcement learning** differs in the following way:
  - No presentation of input/output pairs (like in supervised learning)
  - Agent is connected to its environment via perception and action
  - After choosing an action, the agent gets the immediate **reward** and the subsequent state
  - Agent however cannot judge, **which action** would have been best in terms of long-term behavior
  - Agent must learn about the system, states, actions, transitions and rewards
  - On-line performance: evaluation of the system simultaneously with learning

# Basic functioning of RL

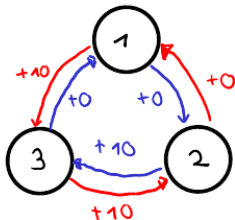
- 1 At each step:
  - Agent receives some input  $i$
  - Some indication of current state  $s$  of the environment
  - Agent chooses some action  $a$
- 2 Action changes the state of the environment
- 3 Value of the **state transition** through a scalar **reinforcement signal**  $r$
- 4 **Goal:** agent should choose actions that tend to increase long-run sum of values
- 5 **Task:** find a policy, mapping states to actions, that maximizes long-run measure of reinforcement
- 6 Often, non-deterministic environments: same action in the same state at different 'times' yield different results

## Example

- ① RL tries to mimic how humans learn new things not only from a teacher, but also in their interaction with the environment
- ② Baby learns waving hands, crying and laughing with interaction of parents (feedback process), **reward** (laughing is positive reward; crying negative reward)
- ③ **RL: machines learn to achieve goals from their interaction with the environment**
- ④ Huge applications of RL in many disciplines (see yourself in the provided literature and also searching yourself online)

# RL: Simple Game 1

- 1 This first game is very simple, we always want to go the neighbor with the highest number.



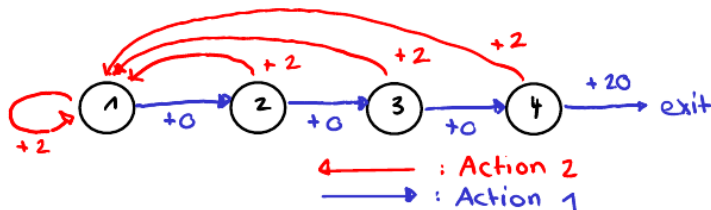
	Action 1	Action 2
State 1	0	10
State 2	10	0
State 3	0	10

- 2 If the agent chooses the action, from which it learned that yields the highest reward (effectively it learned the table from above), it would have learned how to play the game successfully.



## RL: Simple Game 2

- 1 The goal of this game is to reach the exit.



- 2 For most games the agent needs to learn to take actions, that do not lead immediately to a reward, but may result in a larger reward further down the track.

# The Q learning rule<sup>54</sup>

- 1 The agent needs to take the current state  $s$  as a variable and return a Q-value for each possible action  $a$ , i.e. it needs to return  $Q(s, a)$  for all  $s$  and  $a$ .
- 2 The Q-value is updated in training via the following iteration step:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [ r + \lambda \cdot \max_{a'} Q(s', a') - Q(s, a) ]$$

We are interested in this new value

Current approximation (can be computed via feedforward)

Learning rate  $\alpha > 0$

Reward

Discount  $\lambda > 0$

Approximation of the Q value of future actions  
! can become hard to compute

where:

- $r$  is the reward that is received, when taking action  $a$  in state  $s$ .
- $\lambda$  the discount is the weight for delayed rewards, where  $\lambda \in [0, 1]$ .
- $\alpha$  is the learning rate.
- $\max_{a'} Q(s', a') - Q(s, a)$  is the maximal reward obtained when choosing action  $a$ . Remark:  $Q(s', a')$  also contains  $Q(s'', a'')$  and so on...

<sup>54</sup>Q learning; see e.g., <https://en.wikipedia.org/wiki/Q-learning>

## Definition

Sequence of states is **Markov**<sup>55</sup> if and only if the probability of moving from  $S_t$  to the next state  $S_{t+1}$  depends only on  $S_t$ , but not on other previous states  $S_{t-1}, S_{t-2}, \dots$ . Thus for all  $t = 1, 2, 3, \dots$ , we have

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, S_2, \dots, S_t] \quad (265)$$

where  $P[\cdot]$  denotes the probability. Recall that the notation  $P[S_{t+1}|S_t]$  means that probability of  $S_{t+1}$  under the condition of  $S_t$ .

- Time-homogeneous Markov chain in RL:

$$P[S_{t+1} = s' | S_t = s] = P[S_t = s' | S_{t-1} = s] \quad (266)$$

---

<sup>55</sup>See also literature given in Lecture 2.

# Markov processes

## Definition (Markov process)

A Markov process (or Markov chain) is a tuple  $(S, P)$ , where

- $S$  is a finite set of states
- $P$  is a state transition probability matrix:

$$P_{ss'} = P[S_{t+1} = s' | S_t = s] \quad (267)$$

### A first scheme:

- Start in state  $s_0$
- Move to  $s_1$  by using  $P_{s_0s_1}$
- Move to  $s_2$  by using  $P_{s_1s_2}$
- and so forth ...

# Markov decision process (MDP)

- 1 Introduce in addition to the previous slides now **reward, action, discount**

## Definition (Markov decision process)

A Markov decision process is a tuple  $(S, A, P, \gamma, R)$ , with the following definitions:

- 1  $S$  is a finite set of states
- 2  $A$  is a finite set of **actions**
- 3  $P$  is the state transition probability matrix

$$P_{ss'}^a = P[S_{t+1} = s' | S_t = s, A_t = a] \quad (268)$$

- 4  $\gamma \in [0, 1]$  is a **discount** factor
- 5  $R : S \times A \rightarrow \mathbb{R}$  is a **reward** function

# Markov decision process (MDP): in words and a 2nd scheme

- 1 MDP: model environment in RL
- 2 Transition to next state  $S_{t+1}$  depends on current state  $S_t$  and action  $A_t$  that is made on current state
- 3 Each state-action pair is complemented with a reward function

## A second scheme:

- Start in state  $s_0$ , choose some action  $a_0 \in A$
- Move to  $s_1$  by using  $P_{s_0s_1}^{a_0}$ , choose some new action  $a_1 \in A$
- Move to  $s_2$  by using  $P_{s_1s_2}^{a_1}$ , choose some new action  $a_2 \in A$
- and so forth ...

# Return, policy and value function

- 1 **Goal** in RL: maximize expected value of the return, i.e., optimal **policy**
- 2 Return  $G_t$  : total discounted reward  $R$  at time step  $t$

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \quad (269)$$

- 3 On the discount factor:  $\gamma \approx 0$  short-sighted evaluation and  $\gamma \approx 1$  long-sighted evaluation
- 4 Why the discount factor?
  - Avoids infinite returns in cyclic Markov processes
  - Uncertainties in future rewards
- 5 **Examples:** Finance : immediate rewards may earn more interests, animals/human behavior (psychology) like better immediate rewards

# Return, policy and value function

- 1 **Policy**  $\pi$ : distribution over actions given states:

$$\pi(a|s) = P[A_t = a | S_t = s] \quad (270)$$

- 2 It is time independent
- 3 Policy guides the choice of an action at a given state



# Return, policy and value function

- 1 **State-value function:**  $v_\pi$  of an MDP is the expected return from state  $s$  and then following the policy  $\pi$ :

$$v_\pi(s) = E_\pi[G_t | S_t = s] \quad (271)$$

- 2 Gives the long-term value of state  $s$  when following the policy  $\pi$
- 3 It holds by using the previous definitions and decompositions:

$$v_\pi(s) = E_\pi[G_t | S_t = s] \quad (272)$$

$$= \dots \quad (273)$$

$$= E_\pi[R_{t+1} | S_t = s] + E_\pi[\gamma v_\pi(S_{t+1}) | S_t = s] \quad (274)$$

# Return, policy and value function

- ① **Action-value function:**  $q_\pi$  of an MDP is the expected return from state  $s$ , plus action  $a$ , and then following the policy  $\pi$ :

$$q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a] \quad (275)$$

- ② It holds by using the previous definitions and decompositions:

$$q_\pi(s) = E_\pi[G_t | S_t = s, A_t = a] \quad (276)$$

$$= \dots \quad (277)$$

$$= E_\pi[R_{t+1} | S_t = s, A_t = a] + E_\pi[\gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \quad (278)$$

- ③ Define for the sake of notation:

$$R_s^a := E_\pi[R_{t+1} | S_t = s, A_t = a] \quad (279)$$

# Relation between state-value and action-value functions

- ① We have:

$$v_{\pi} = \sum_{a \in A} \pi(a|s) q_{\pi}(s, a) \quad (280)$$

$$q_{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{\pi}(s') \quad (281)$$

- ② **Bellman equation** for  $v_{\pi}$ :

$$v_{\pi}(s) = \sum_{a \in A} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{\pi}(s') \right) \quad (282)$$

- ③ Bellman equation relates state-value function  $v_{\pi}$  of one state with other states
- ④ In time-continuous MDP, when the state space  $S$  and the action space  $A$  are continuous, the resulting equation is the **Hamilton-Jacobi-Bellman (HJB)**, which is an important PDE (partial differential equation) (Numerik 3 - NumPDE!)

# Relation between state-value and action-value functions

- ① **Bellman equation** for  $q_\pi$ :

$$q_\pi(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \sum_{a' \in A} \pi(a'|s') q_\pi(s', a') \quad (283)$$

- ② Computes value function  $q_\pi$  for a given policy  $\pi$
- ③ Combination for all  $n$  states, yields  $n$  linear equations for  $n$  unknown value functions

→ Linear equation system

- ④ Naive numerical approach: cost complexity  $O(n^3)$  (Numerik 1!!!!)

# Optimal value function and optimal policy

- 1 Principal interest is optimality!
- 2 **Optimal** state-value function  $v_*(s)$ :

$$v_*(s) = \max_{\pi} v_{\pi}(s) \quad (284)$$

- 3 Specifies best possible performance of the MDP
- 4 MDP 'solved' when optimal value function found
- 5 **Optimal** action-value function  $q_*(s, a)$ :

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a) \quad (285)$$

# Optimal value function and optimal policy

- ① For defining optimal policy, introduce partial ordering:

$$\pi \geq \pi' \quad \text{if } v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S} \quad (286)$$

## Theorem

For any MDP, we have:

- *There exists an optimal policy  $\pi_*$*
- *All optimal policies achieve the optimal value function  $v_{\pi_*}(s) = v_*(s)$*
- *All optimal policies achieve the optimal action-value function  $q_{\pi_*}(s, a) = q_*(s, a)$*
  
- For a proof, we refer to Richard Bellman<sup>56</sup>

---

<sup>56</sup>Richard Bellman. "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6.4 (1957), pp. 679–684.

## Finding the optimal policy

- The previous theorem yields an optimality criterion how to obtain the optimal policy:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg \max_{a \in A} q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \quad (287)$$

## Finding the optimal value function

- 1 Introduce Bellman optimality equations for  $v_*$  and  $q_*$
- 2 We use

$$v_*(s) = \max_a q_*(s, a) \quad (288)$$

$$q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \quad (289)$$

- 3 Then:

$$v_*(s) = \max_a \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \right) \quad (290)$$

- 4 And

$$q_*(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a') \quad (291)$$

- 5 Again: the Bellman equations are clearly nonlinear and difficult to solve numerically



# Numerical solution and convergence of MDP

- 1 Nonlinear equations to be solved!
  - 2 Various possibilities in numerics
  - 3 One idea is dynamic programming: recursive subproblems that are simpler to solve
- Iteration procedure (like fixed-point iteration):

$$x_{t+1} = f(x_t) \quad (292)$$

- 4 Convergence proof then with well-known contraction theorem (Numerik 1), Banach's fixed-point theorem:

$$\|f(x) - f(y)\| \leq \rho \|x - y\| \quad (293)$$

with  $\rho < 1$ .

# Policy evaluation

- 1 We realize the previous ideas now
- 2 Initial guess  $v_1$
- 3 Construct sequence for  $k = 1, 2, \dots$

$$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi \quad (294)$$

- 4 Iteration function:

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_k(s') \right) \quad (295)$$

- 5 Stopping criterion:

$$\|v_{k+1} - v_k\| < TOL \quad (296)$$

# Policy iteration

- 1 Initialize  $\pi$  randomly
- 2 Repeat until previous policy and current policy match
  - 1 Evaluation  $v_\pi$  by policy evaluation (previous evaluation)
  - 2 For each state  $s$ , set

$$\pi(s) := \arg \max_{a \in A} q(s, a) = \arg \max_{a \in A} \left( R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s') \right) \quad (297)$$

## Convergence proof (I)

- 1 We establish that at each policy iteration, the algorithm will improve the policy
- 2 Suppose a deterministic policy  $\pi$ , after we obtain  $\pi'$
- 3 It holds

$$q_{\pi}(s, \pi'(s)) = \max_{a \in A} q_{\pi}(s, a) \geq q_{\pi}(s, \pi(s)) = v_{\pi}(s) \quad (298)$$

- 4 We then obtain

$$v_{\pi}(s) \leq q_{\pi}(s, \pi'(s)) = E_{\pi'}[R_{t+1} + \gamma v_{\pi}(S_{t+1}) | S_t = s] \quad (299)$$

$$\leq E_{\pi'}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \quad (300)$$

$$\leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_{\pi}(S_{t+2}, \pi'(S_{t+2})) | S_t = s] \quad (301)$$

$$\leq E_{\pi'}[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s] \quad (302)$$

$$= v_{\pi'}(s) \quad (303)$$

- 5 Thus, we have shown

$$v_{\pi} \leq v_{\pi'}(s) \quad (304)$$

## Convergence proof (II)

- 1 When improvements stop, we have

$$q_{\pi}(s, \pi'(s)) = \max_{a \in A} q_{\pi}(s, a) = q_{\pi}(s, \pi(s)) = v_{\pi}(s) \quad (305)$$

- 2 Specifically, the Bellman optimality condition is satisfied:

$$v_{\pi}(s) = \max_{a \in A} q_{\pi}(s, a) \quad (306)$$

- 3 Thus

$$v_{\pi}(s) = v_{*}(s) \quad \forall s \in S \quad (307)$$

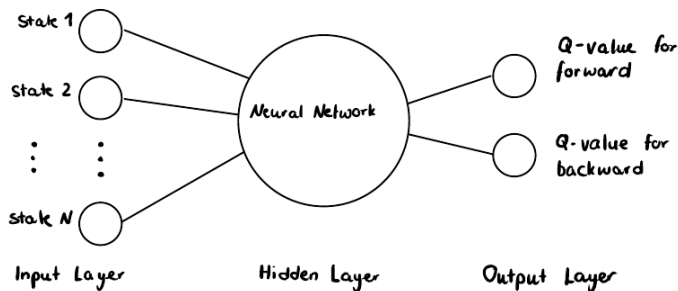
- 4 In words: Therefore,  $\pi$  is an optimal policy and the evaluation yields the optimal value function
- 5 An alternative is **value iteration**; see <sup>57</sup>

---

<sup>57</sup>Xintian Han <https://cims.nyu.edu/~donev/Teaching/WrittenOral/Projects/XintianHan-WrittenAndOral.pdf>

# Deep Reinforcement Learning

- 1 Problem:** To compute the optimal value function, we need to store the Q-list with all possible states. But the problem size can become too large to be stored.
- 2 Example:** In chess, there are 69 352 859 712 417 possible states, just after 10 moves. To save the Q-value to every possible state as a C++ integer (4 bytes) would already need about 277 terrabytes!
- 3 Solution:** Replace the Q-table with a Neural Network!



# How to perform Deep Reinforcement Learning

- 1 Let the network "play" the game and save each state and action until the game is finished.
- 2 Compute to each step the corresponding Q-value via the formula

$$Q(s,a) \leftarrow Q(s,a) + \alpha [r + \lambda \cdot \max_{a'} Q(s',a') - Q(s,a)]$$

We are interested in this new value

Current approximation (can be computed via feedforward)

Learning rate  $\alpha > 0$

Reward

Discount  $\lambda > 0$

Approximation of the Q value of future actions  
! can become hard to compute

How to compute  $\max_{a'} Q(s', a')$

- In a single player game  $\max_{a'} Q(s', a')$  can be directly computed since  $s'$  is the state after taking action  $a$ .
  - In a multiplayer game we need also to take the actions of the other players into account, for example get  $s'$  via the training game, or compute the best moves for the other players.
  - Use as training data:  $(s, \max_{a'} Q(s', a'))$
- 3 Use Backpropagation to train the network with the training data from step 2

End of Lecture 10



## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# General comments where ML may meet differential equations

- 1 Solving differential equations with ML (e.g., NN) methods
- 2 Approximation of high-dimensional problems
- 3 Improvement of numerical components with ML, e.g., linear, nonlinear solvers, and adaptive schemes
- 4 Data-driven simulations
- 5 Model-order reduction by using high-fidelity computations for obtaining characteristic modes and then computing (cheaper?!) ML approximations (useful where similar PDEs must be solved numerous times such as parameter estimation, optimal control, ...)
- 6 On the other hand (see below in L12), techniques from differential equations may further improve ML procedures themselves

# Review papers for self-studies over the next three weeks

In the three weeks of the vacation period, please have a look into these papers:

- Deep learning in fluid mechanics<sup>58</sup>
- Machine Learning for Fluid Mechanics<sup>59</sup>

---

<sup>58</sup>J. Nathan Kutz. “Deep learning in fluid dynamics”. In: *Journal of Fluid Mechanics* 814 (2017), pp. 1–4. DOI: 10.1017/jfm.2016.803.

<sup>59</sup>Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. “Machine Learning for Fluid Mechanics”. In: *Annual Review of Fluid Mechanics* 52.1 (2020), pp. 477–508. DOI: 10.1146/annurev-fluid-010719-060214. eprint: <https://doi.org/10.1146/annurev-fluid-010719-060214>. URL: <https://doi.org/10.1146/annurev-fluid-010719-060214>.

## Short questions for self control and takeaway (I)

- 1 What are possible applications of neural networks in scientific computing?
- 2 Up to how many layers are NN trained for fluid mechanics problems?
- 3 What do the abbreviations POD and DMD stand for?
- 4 What is the main idea behind POD-based modeling?
- 5 What is the mathematical basis for reduced order modeling?
- 6 What are the gains by using dimensionality reduction techniques?
- 7 What are the two main failings of POD/DMD? How can DNNs help?
- 8 What is the difference between laminar and turbulent flow?
- 9 What is the tensor basis neural network?

## Short questions for self control and takeaway (II)

- 1 What are some limitations of machine learning based scientific computing?
- 2 List some aspects of the 'future' and challenges of DNN's for fluid modeling
- 3 How are the previous challenges related to this class?
- 4 What does Nathan Kutz in his article formulate as a general rule for using DNNs?
- 5 What are challenges of ML for dynamical systems?
- 6 For how long does the history of ML and fluid dynamics date back?
- 7 What are typical goals in flow optimization using ML?

End of Lecture 11

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Parametrization the derivative of hidden state using a neural network
- Avoids using specific discrete sequence of hidden layers
- Output of NN computed via black-box differential solver
- Constant memory cost
- Allows for end-to-end training of ODEs within larger models



# Neural Ordinary Differential Equations

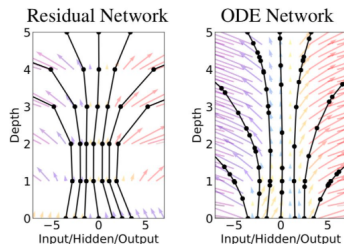


Image source: Figure 1 from NeuralODE paper<sup>60</sup>

- Left: discrete sequence of finite transformations in a residual network
- Right: ODE network defines a (continuous) vector field
- Literature: Neural ODE paper,<sup>61</sup> Yannic Kilcher video<sup>62</sup>

<sup>60</sup>Ricky T. Q. Chen et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

<sup>61</sup>Ibid.

<sup>62</sup>Yannic Kilcher. *Neural Ordinary Differential Equations*. Youtube. 2019. URL: <https://www.youtube.com/watch?v=jltgNGt8Lpg>.

## Basics: relation to forward Euler

- 1 Recall: computing the hidden state:

$$h_{t+1} = h_t + f(h_t, \theta_t) \quad (308)$$

with  $t \in \{0, \dots, T\}$  and  $h_t \in \mathbb{R}^D$

- 2 This scheme should be well recognized from Numerik 2 (Numerik of GDGL (ODEs) and Eigenwerte)<sup>63</sup>

→ Forward Euler discretization

---

<sup>63</sup>L. Samolik and T. Wick. *Numerische Mathematik II (Numerik GDGL, Eigenwerte)*. Institute for Applied Mathematics, Leibniz Universität Hannover, Germany. 2019.

## Basics: relation to forward Euler

- ① Just to recall: Solve

$$y'(t) = f(t, y) \quad (309)$$

with forward Euler

$$\frac{y_{n+1} - y_n}{\Delta t} = f(t, y_n), \quad y(0) = y_0 \quad (310)$$

for  $n = 0, \dots, N$ , and  $\Delta t = t_{n+1} - t_n$

- ② And with a parameter  $a \in \mathbb{R}$ , we have

$$f(t, y_n) := f(t, y_n, a) \quad (311)$$

e.g., ODE model problem (see any Numerik 2 lecture notes; for instance<sup>64</sup>)

$$f(t, y_n, a) = ay_n \quad (312)$$

---

<sup>64</sup>Samolik and Wick, *Numerische Mathematik II (Numerik GDGL, Eigenwerte)*.

## Basics: relation to forward Euler

- 1 Adding more layers and smaller steps yields the continuous approximation (i.e., the limit):

$$\frac{dh(t)}{dt} = f(h(t), t, \theta) \quad (313)$$

- 2 Input layer denoted by  $h(0)$
- 3 Output layer  $h(T)$
- 4 ODE problem for which some (black-box) ODE solver can be employed
- 5 Due to the construction the intermediate layers  $h(t)$  for  $0 < t < T$  are evaluated with this procedure
- 6 Accuracy depends clearly on the ODE solvers and can be adjusted by typical means known from ODE numerics

# Benefits of ODE components in NN

- ① Memory efficiency: not storage of intermediate quantities for the forward pass; no backpropagation through the operations of the solver
- ② Adaptivity (well-known expertise in my group at IfAM) for instance adaptive-time-step Euler, Runge-Kutta, ...
- ③ Continuous time-series models (compare to RNN, Lecture 8)
- ④ Utilizing well-known mathematical structures from ODE theory and numerics

# Reverse-mode automatic differentiation of ODE solutions

- 1 Main technical **challenge**: compute backpropagation (reverse-mode differentiation) with the help of the ODE solver
- 2 Idea: compute gradients with the help of an additional **adjoint** problem
- 3 Adjoint problems are very well known in numerical optimization and can also be used for a posteriori error estimation
- 4 The adjoint variable enters as Lagrange multiplier within a constraint optimization problem, e.g., Nocedal/Wright<sup>65</sup>
- 5 In time-dependent problems, the adjoint is running backward in time, but always linear, independently whether the original (primal) problem is linear or nonlinear

---

<sup>65</sup>Nocedal and Wright, *Numerical Optimization*.

## Reverse-mode automatic differentiation of ODE solutions

1 Recall:  $\frac{dz}{dt} = f(z, t, \theta)$  with the independent variable  $t$ , the unknown (dependent) variable  $z$  and a parameter  $\theta$

2 Consider first

$$z(t_1) = z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta). \quad (314)$$

3 Then within some optimization formulation:

$$L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta) dt\right) \quad (315)$$

4 Objective: optimize scalar-valued loss function  $L(\cdot)$

5 In an abstract fashion, we have

$$L(\text{ODESolve}(z(t_0), f, t_0, t_1, \theta)) \quad (316)$$

6 Then: for optimizing  $L(\cdot)$ , we need to compute gradients w.r.t.  $\theta$

→ chain rule, but expensive; for this reason adjoint ODE

## Reverse-mode automatic differentiation of ODE solutions

- 1 Adjoint: sensitivity of  $L(\cdot)$  due to variations in the primal variable
- 2 Adjoint: Gradient of loss depends on the hidden state  $z(t)$  at each time  $t$
- 3 In mathematical notation:

$$a(t) = \frac{\partial L}{\partial z(t)} \quad (317)$$

which is determined by solving another ODE

- 4 Running backward in time with the initial value  $\partial L / \partial z(t_1)$
- 5 Difficulty: since running backward in time, the primal solution of the original problem must be known (in the nonlinear case)
- 6 Relation to the chain rule:

$$\frac{da(t)}{dt} = -a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z} \quad (318)$$



# Reverse-mode automatic differentiation of ODE solutions<sup>67</sup>

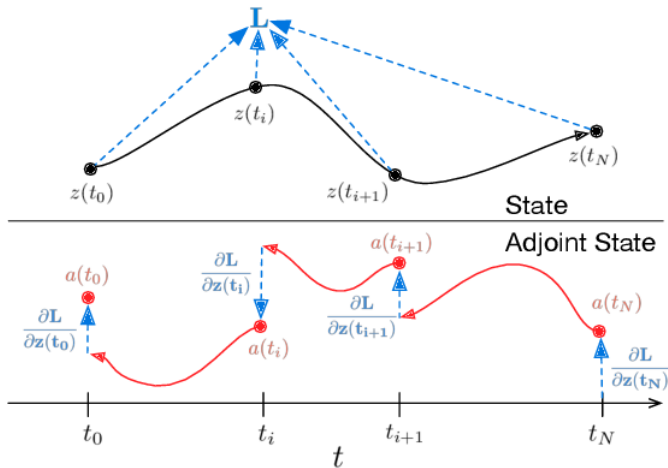


Image source: Figure 2 from NeuralODE paper<sup>66</sup>

<sup>66</sup>Chen et al., *Neural Ordinary Differential Equations*.

<sup>67</sup>Ibid.

# Reverse-mode automatic differentiation of ODE solutions

- 1 It remains to compute the variation of  $L(\cdot)$  with respect to the parameter  $\theta$
- 2 Depends on  $z(t)$  and  $a(t)$
- 3 Third ODE problem

4 Then:

$$\frac{dL}{d\theta} = - \int_{t_1}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \quad (319)$$

- 5 As before: classical ODE problem statement, which can be solved with (black-box) ODE methods

## Algorithm for reverse-mode derivative

- 1 Idea: define a triple that computes all three ODEs
- 2 Input: parameters  $\theta$ , start time  $t_0$ , end time  $t_1$ , final state  $z(t_1)$ , loss gradient  $\partial L / \partial z(t_1)$
- 3 Define initial state of triple:

$$s_0 = \left[ z(t_1), \quad \frac{\partial L}{\partial z(t_1)}, \quad 0 \in \mathbb{R}^{\dim(\theta)} \right] \quad (320)$$

- 4 Define an augmented state (solution in time t):

$$\text{aug\_dynamics}([z(t), a(t), \cdot], t, \theta) \quad (321)$$

in which we compute (and return):

$$\left[ f((z(t), t, \theta), \quad -a(t)^T \frac{\partial f}{\partial z}, \quad -a(t)^T \frac{\partial f}{\partial \theta} \right] \quad (322)$$

- 5 Solve reverse-time ODE

$$\left[ z(t_0), \quad \frac{\partial L}{\partial z(t_0)}, \quad \frac{\partial L}{\partial \theta} \right] = \text{ODESolve}(s_0, \text{aug\_dynamics}, t_1, t_0, \theta) \quad (323)$$

# Continuous backpropagation

- 1 Given the differential equation:

$$\frac{dz(t)}{dt} = f(z(t), t, \theta) \quad (324)$$

- 2 Formally the adjoint state is given by

$$a(t) = \frac{dL}{dz(t)} \quad (325)$$

## Lemma

*With the previous problem statement and definitions, it holds*

$$\frac{da(t)}{dt} = -a(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} \quad (326)$$

## Preparations for the proof

- 1 Recall from standard neural networks with the help of the chain rule:

$$\frac{dL}{dh_t} = \frac{dL}{dh_{t+1}} \frac{dh_{t+1}}{dh_t} \quad (327)$$

- 2 With our previous definitions this discrete notation becomes for a continuous-in-time approximation to

$$z(t + \epsilon) = \int_t^{t+\epsilon} f(z(\tau), \tau, \theta) d\tau + z(t) = T_\epsilon(z(t), t) \quad (328)$$

- 3 Chain rule

$$\frac{dL}{dz(t)} = \frac{dL}{dz(t + \epsilon)} \frac{dz(t + \epsilon)}{dz(t)} \quad (329)$$

or in other words

$$a(t) = a(t + \epsilon) \frac{\partial T_\epsilon(z(t), t)}{\partial z(t)} \quad (330)$$

## Proof of Lemma 31

- ① We work with the definition of the derivative:

$$\frac{da(t)}{dt} = \lim_{\epsilon \rightarrow 0} \frac{a(t + \epsilon) - a(t)}{\epsilon} \quad (331)$$

Then, we obtain

$$\frac{da(t)}{dt} = \lim_{\epsilon \rightarrow 0} \frac{a(t + \epsilon) - a(t)}{\epsilon} \quad (332)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{a(t + \epsilon) - a(t + \epsilon) \frac{\partial}{\partial z(t)} T_\epsilon(z(t))}{\epsilon} \quad (333)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{a(t + \epsilon) - a(t + \epsilon) \frac{\partial}{\partial z(t)} (z(t) + \epsilon f(z(t), t, \theta) + O(\epsilon^2))}{\epsilon} \quad (334)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{a(t + \epsilon) - a(t + \epsilon) \left( I + \epsilon \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + O(\epsilon^2) \right)}{\epsilon} \quad (335)$$

$$= \lim_{\epsilon \rightarrow 0} \frac{-\epsilon a(t + \epsilon) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + O(\epsilon^2)}{\epsilon} \quad (336)$$

$$= \lim_{\epsilon \rightarrow 0} -a(t + \epsilon) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} + O(\epsilon) \quad (337)$$

$$= -a(t) \frac{\partial f(z(t), t, \theta)}{\partial z(t)} \quad (338)$$

## Gradients with respect to $\theta$ and $t$

① We use again Lemma 31

② Then:

$$\frac{\partial \theta(t)}{\partial t} = 0, \quad \frac{dt(t)}{dt} = 1 \quad (339)$$

③ Generalize to a triple (augmented state):

$$\frac{d}{dt} \begin{bmatrix} z \\ \theta \\ t \end{bmatrix} (t) = f_{aug}([z, \theta, t]) := \begin{bmatrix} f([z, \theta, t]) \\ 0 \\ 1 \end{bmatrix} \quad (340)$$

with

$$a_{aug} := \begin{bmatrix} a \\ a_{\theta} \\ a_t \end{bmatrix}, \quad a_{\theta}(t) := \frac{dL}{d\theta(t)}, \quad a_t(t) := \frac{dL}{dt(t)} \quad (341)$$

# Jacobian

- 1 The Jacobian reads:

$$\frac{\partial f_{aug}}{\partial [z, \theta, t]} = \begin{pmatrix} \frac{\partial f}{\partial z} & \frac{\partial f}{\partial \theta} & \frac{\partial f}{\partial t} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} (t) \quad (342)$$

- 2 Then, with Lemma 31

$$\frac{da_{aug}(t)}{dt} = -[a(t), a_{\theta}(t), a_t(t)] \frac{\partial f_{aug}}{\partial [z, \theta, t]}(t) = - \left[ a \frac{\partial f}{\partial z}, a \frac{\partial f}{\partial \theta}, a \frac{\partial f}{\partial t} \right] (t) \quad (343)$$

- 3 The last element is the differential equation itself
- 4 The first element is the adjoint
- 5 The middle element is the variation w.r.t. to the parameters  $\theta$
- 6 This yields the total derivative of the functional  $L(\cdot)$



# Examples of learning an ODE with NeuralODEs

- 1 Learning the Lotka-Volterra problem (around 1920' predator-prey; sharks/fish)
- 2 Nice introductions for the governing mathematical model in<sup>68</sup> and<sup>69</sup>
- 3 Problem statement:

$$\frac{dx}{dt} = \frac{3}{2}x - xy \quad (344)$$

$$\frac{dy}{dt} = -3y + xy \quad (345)$$

with initial conditions

$$x(0) = 1, \quad y(0) = 1. \quad (346)$$

---

<sup>68</sup>M. Braun. *Differential equations and their applications*. Springer, 1993.

<sup>69</sup>A. Quarteroni and P. Gervasio. *A Primer on Mathematical Modelling*. Springer, 2020.

# Examples of learning an ODE with NeuralODEs

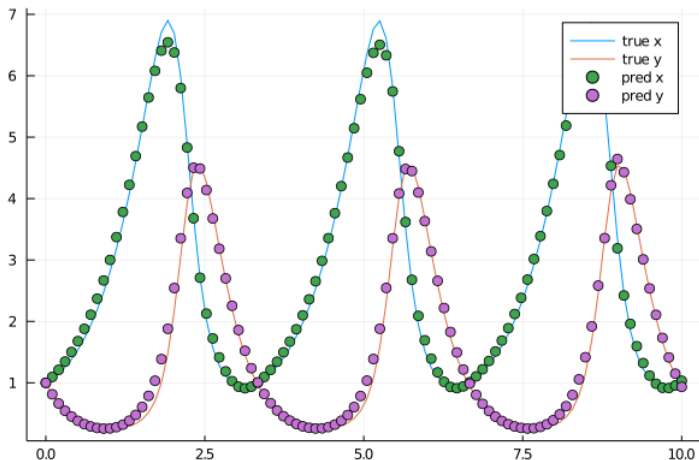


Image source: picture from Christopher Rackauckas' GitHub Gist on solving the Lotka-Volterra problem with a NeuralODE

Link: <https://gist.github.com/ChrisRackauckas/a531030dc7ea5c96179c0f5f25de9979>

End of Lecture 12

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Outline

- Neural network constructions for solving differential equations
- Basic ideas
- Program snippets with illustrative numerical examples
- Current extensions (very active research field!)

# Main literature for the first 45 minutes

- DeepXDE<sup>70</sup>  
<https://epubs.siam.org/doi/pdf/10.1137/19M1274067>
- PINNs<sup>71</sup> <https://www.sciencedirect.com/science/article/pii/S0021999118307125>

---

<sup>70</sup>Lu Lu et al. “DeepXDE: A Deep Learning Library for Solving Differential Equations”. In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: [10.1137/19M1274067](https://doi.org/10.1137/19M1274067).

<sup>71</sup>M. Raissi, P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2018.10.045>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999118307125>.

## Recall from Lecture 5

- 1  $N^L(x) : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$ :  $L$  layer neural network
- 2  $N_l$  neurons in the  $l$ th layer
- 3  $N_0 = d_{in}$  and  $N_L = d_{out}$
- 4 Weight matrix:  $W^l \in \mathbb{R}^{N_l \times N_{l-1}}$
- 5 Bias vector:  $b^l \in \mathbb{R}^{N_l}$
- 6 Activation function  $\sigma$
- 7 Feedforward neural network (FNN):

$$N^0(x) = x \in \mathbb{R}^{d_{in}} \quad (347)$$

$$N^l(x) = \sigma(W^l N^{l-1}(x) + b^l) \in \mathbb{R}^{N_l}, \quad 1 \leq l \leq L-1 \quad (348)$$

$$N^L(x) = W^L N^{L-1}(x) + b^L \in \mathbb{R}^{d_{out}} \quad (349)$$

# IBVP - initial boundary value problem

- 1 Space-time cylinder  $\Omega \subset \mathbb{R}^d \times \mathbb{R}$
- 2 Parameter  $\lambda \in \mathbb{R}$
- 3 Independent variable  $x = (x_1, \dots, x_d, t)$
- 4 Unknown variable  $u(x) : \Omega \rightarrow \mathbb{R}$
- 5 PDE (partial differential equation)<sup>72 73</sup>

$$F(D^2u, Du, u, x, \lambda) = 0, \quad x \in \Omega \quad (350)$$

- 6 Boundary and initial conditions:

$$B(u, x) = 0 \quad \text{on } \partial\Omega \quad (351)$$

---

<sup>72</sup>Notation from L.C. Evans, *Partial differential equations*, AMS, 2010

<sup>73</sup>See also (Wick, *Numerical methods for partial differential equations*)[Chapter 4].



## Example: Heat equation

- 1 Find  $u : \Omega \rightarrow \mathbb{R}$  such that

$$\partial_t u - \nabla \cdot (\lambda \nabla u) = 0 \quad \text{in } \Omega \quad (352)$$

$$u = g_D \quad \text{on } \Gamma_D \quad (353)$$

$$\partial_n u = g_N \quad \text{on } \Gamma_N \quad (354)$$

- 2 The initial condition is on the space-time cylinder boundary and therefore included in  $g_D$

# PINN (I)

- 1 First construct neural network  $\hat{u}(x, \theta)$
- 2 Serves as surrogate for the sought solution  $u(x)$
- 3 Moreover,  $\theta = \{W^l, b^l\}_{1 \leq l \leq L}$  is as usual (Lecture 5) the set of the weights and biases in the neural network  $\hat{u}$
- 4 Goal:  $\hat{u}$  should represent the physics from the IVBP:  $F = 0$  and  $B = 0$  (like a root-finding problem; and the same approach as we adopt also often for FEM)
- 5 Restrict  $\hat{u}$  to scattered points (randomly distributed or clustered (in FEM we would design a mesh))
- 6 Those scattered points (i.e., residual points) are the training data:  
 $\mathcal{T} = \{x_1, x_2, \dots, x_M\}$
- 7 We assume that  $\mathcal{T} = \mathcal{T}_f \cup \mathcal{T}_b$  with  $\mathcal{T}_f \subset \Omega$  and  $\mathcal{T}_b \subset \partial\Omega$

## PINN (II)

- 1 Known from Lecture 5: we need to measure discrepancy (regression) between network  $\hat{u}$  and the constraints ( $F = 0$  and  $B = 0$ )
- 2 Loss function: here weighted sum:

$$L(\theta, \mathcal{T}) = w_f L_f(\theta, \mathcal{T}_f) + w_b L_b(\theta, \mathcal{T}_b) \quad (355)$$

where  $w_f, w_b \in \mathbb{R}$  are weights

- 3 The single terms are given by

$$L_f(\theta, \mathcal{T}_f) = \frac{1}{M_f} \sum_{x \in \mathcal{T}_f} |F(D^2 \hat{u}, D \hat{u}, \hat{u}, x, \lambda)|^2 \xrightarrow{M_f \rightarrow \infty} \frac{1}{|\Omega|} \int_{\Omega} |F(D^2 \hat{u}, D \hat{u}, \hat{u}, x, \lambda)|^2 dx = \frac{1}{|\Omega|} \|F\|_{L^2(\Omega)}^2 \quad (356)$$

$$L_b(\theta, \mathcal{T}_b) = \frac{1}{M_b} \sum_{x \in \mathcal{T}_b} |B(\hat{u}, x)|^2 \xrightarrow{M_b \rightarrow \infty} \frac{1}{|\partial\Omega|} \int_{\partial\Omega} |B(\hat{u}, x)|^2 dx = \frac{1}{|\partial\Omega|} \|B\|_{L^2(\partial\Omega)}^2 \quad (357)$$

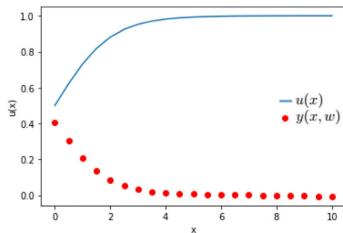
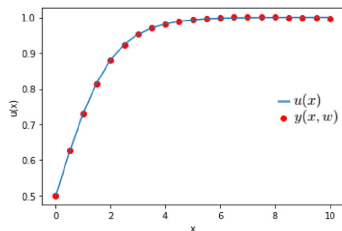
## PINN (III)

- 1 Training:

$$\min_{\theta} L(\theta, \mathcal{T}) \quad (358)$$

- 2 Lecture 5: highly nonlinear, nonconvex optimization problem
- 3 Numerics: gradient descent or second-order methods
- 4 Possibly bad local minima
- 5 Since several local minima, no unique solution (in contrast to classical FEM if well-posed PDEs are considered)
- 6 Some work to determine all hyperparameters
- 7 On the other hand, no mesh dependencies (meshless method) and flexibility in choosing the residual points

## Example of good and bad local minima<sup>74</sup>



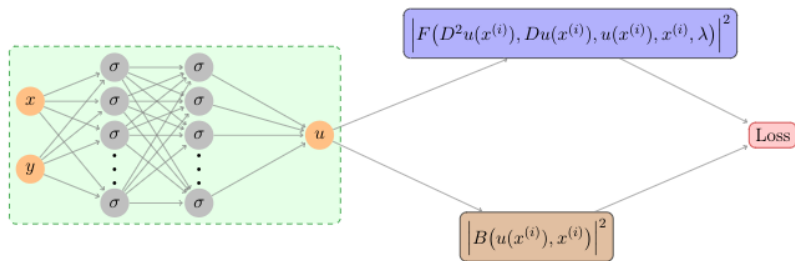
- Approximate  $u(x) = \frac{1}{1+e^{-x}}$  with the help of a neural network  $y(x, w)$ , where  $w$  are the weights
- Reasons for bad local minima: randomly generated initial weights and also some influence of the learning rate (step size of Adam's optimizer; stochastic gradient descent)

<sup>74</sup>Tobias Knoke and Thomas Wick. "Solving differential equations via artificial neural networks: Findings and failures in a model problem". In: *Examples and Counterexamples 1* (2021), p. 100035. DOI: <https://doi.org/10.1016/j.exco.2021.100035>.

# PINN algorithm

- Given PDE and boundary/initial conditions in strong form, namely  $F = 0$  and  $B = 0$ 
  - 1 Construct neural network  $\hat{u}(x, \theta)$
  - 2 Specify training sets  $\mathcal{T}_f$  and  $\mathcal{T}_b$
  - 3 Design a loss function  $L(\theta, \mathcal{T})$
  - 4 Train network to determine  $\theta^*$  (best parameters) by minimizing  $L(\theta, \mathcal{T})$

# Illustrative summary: Physics Informed Neural Networks



# Programming PINNs

- 1 Create a neural network
- 2 Define the PDE residual
- 3 Training loop
- 4 Visualize results



# Programming PINNs: Neural Network

```
import torch

# define the neural network architecture
class PINN(torch.nn.Module):
    def __init__(self):
        super(PINN, self).__init__()
        self.layer1 = torch.nn.Linear(1, 50) # [ 1 -> 50]
        self.layer2 = torch.nn.Linear(50, 50) # [50 -> 50]
        self.layer3 = torch.nn.Linear(50, 1) # [50 -> 1]

    def forward(self, x):
        tmp = torch.tanh(self.layer1(x)) #  $a = \sigma(W_1x + b_1)$ 
        tmp = torch.tanh(self.layer2(tmp)) #  $a = \sigma(W_2a + b_2)$ 
        y = self.layer3(tmp) #  $y = W_3a + b_3$ 
        return torch.reshape(y, (-1,)) # flatten neural network output

pinn = PINN()
```

# Programming PINNs: PDE residual

```
# compute order.th derivative of neural net u in direction of x
def derivative(u, x, order=1):
    ones = torch.ones_like(u)
    deriv = torch.autograd.grad(u, x, create_graph=True, grad_outputs=ones)[0]
    for i in range(1, order):
        ones = torch.ones_like(deriv)
        deriv = torch.autograd.grad(
            deriv,
            x,
            create_graph=True,
            grad_outputs=ones
        )[0]
    return deriv

# residual of PDE: -u'' = -1
def pde_residual(x):
    return derivative(pinn(x), x, order=2) - torch.ones_like(x)
```

# Programming PINNs: Training loop

```
# sample points  $x \in \Omega = (0, 1)$ 
x = torch.autograd.Variable(torch.rand(16, 1), requires_grad=True)
# left boundary:  $x = 0$ 
x_0 = torch.autograd.Variable(torch.Tensor([[0.]]), requires_grad=True)
# right boundary:  $x = 1$ 
x_1 = torch.autograd.Variable(torch.Tensor([[1.]]), requires_grad=True)

# train network that approximates the PDE
optimizer = torch.optim.Adam(pinn.parameters(), lr=1e-3)
# using mean squared error as a loss function
mse = torch.nn.MSELoss()
```

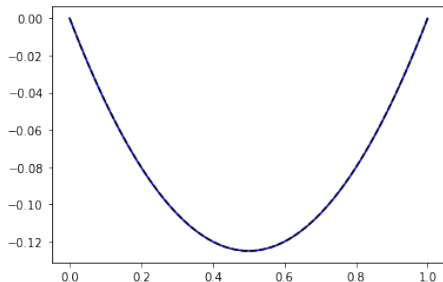
# Programming PINNs: Training loop

```
# start training
for epoch in range(1000):
    loss = mse(
        pde_residual(x),
        torch.autograd.Variable(torch.zeros(16))
    ) # learn: -u'' = -1
    loss += mse(
        torch.cat((pinn(x_0), pinn(x_1))),
        torch.autograd.Variable(torch.zeros(2))
    ) # learn: u(0) = 0 and u(1) = 0

    optimizer.zero_grad() # clear gradients for next epoch
    loss.backward() # backpropagation: compute gradients
    optimizer.step() # apply gradients
```

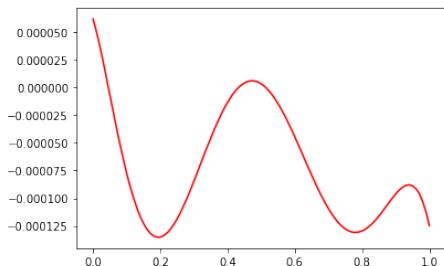
# Programming PINNs: Results

```
import matplotlib.pyplot as plt
x = torch.linspace(0, 1, 100).reshape(100, 1)
u = pinn(x) # neural network solution
analytic_solution = (-0.5*x*(1-x)).reshape(-1,) #  $u(x) = -\frac{1}{2}x(1-x)$ 
plt.plot(x.numpy(), u.detach().numpy(), color="blue")
plt.plot(x.numpy(), analytic_solution.numpy(), "--", color="black")
plt.show()
```



# Programming PINNs: Results

```
import matplotlib.pyplot as plt
x = torch.linspace(0, 1, 100).reshape(100, 1)
y = pinn(x) # neural network solution
analytic_solution = (-0.5*x*(1-x)).reshape(-1,) #  $u(x) = -\frac{1}{2}x(1-x)$ 
# plotting the error between neural network and analytical solution
plt.plot(x.numpy(), u.detach().numpy() - analytic_solution.numpy(), color="red")
plt.show()
```



# Abstract error analysis

- 1 See Lecture 1, numerical concepts, errors
- 2 In PINNs, we have three major error sources:
  - 1 Optimization error  $e_{opt}$
  - 2 Generalization error  $e_{gen}$
  - 3 Approximation error  $e_{app}$

- 3 Total error:

$$e := e_{opt} + e_{gen} + e_{app} \quad (359)$$

- 4 In more detail (typical argument via triangle inequality)

$$e := \|\tilde{u}_{\mathcal{T}} - u\| \leq \|\tilde{u}_{\mathcal{T}} - u_{\mathcal{T}}\| + \|u_{\mathcal{T}} - u_{\mathcal{F}}\| + \|u_{\mathcal{F}} - u\| \quad (360)$$

where  $\mathcal{F}$  is the family of all possible functions that can be presented by the neural network architecture

## Further methods to solve differential equations: second 45 minutes

- DGM<sup>75</sup>
- Deep Ritz<sup>76</sup>
- vPINN<sup>77</sup>

---

<sup>75</sup>Justin Sirignano and Konstantinos Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2018.08.029>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999118305527>.

<sup>76</sup>Weinan E and Bing Yu. “The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems”. In: *Communications in Mathematics and Statistics* 6.1 (2018), pp. 1–12. ISSN: 2194-671X. DOI: 10.1007/s40304-018-0127-z. URL: <https://doi.org/10.1007/s40304-018-0127-z>.

<sup>77</sup>E. Kharazmi, Z. Zhang, and G. E. Karniadakis. *Variational Physics-Informed Neural Networks For Solving Partial Differential Equations*. 2019. arXiv: 1912.00873 [cs.NE].



- 1 Very similar to PINNs
- 2 Difference in some details of the design:
  - DGM: Monte-Carlo approximation for fast computation of (second) derivatives; focus on high-dimensional PDEs up to 200 dimensions
  - PINN: automatic differentiation to calculate derivatives; physics-based data-driven modeling
- 3 Formulate PDE in strong form, boundary and initial conditions within a functional and minimize:  $\min J(u)$  with

$$J(u) = \|\partial_t u + L(u)\|_2^2 + \|u - g\|_2^2 + \|u(0) - u_0\|_2^2 \quad (361)$$

- 4 As before  $\theta \in \mathbb{R}^K$  are the neural network parameters (weights and biases)

# DGM algorithm

- 1 Generate random points (scattered; residual)  $s_n$  on  $\Omega \times [0, T]$  and  $\partial\Omega \times [0, T]$
- 2 Calculate squared error:

$$G(\theta_n, s_n) = (\partial_t u(t_n, x_n, \theta_n) + Lu(t_n, x_n, \theta_n))^2 \quad (362)$$

$$+ (u(\tau_n, z_n, \theta_n) - g(\tau_n, z_n))^2 \quad (363)$$

$$+ (u(0, w_n, \theta_n) - u_0(w_n))^2 \quad (364)$$

- 3 Descent step at the random point  $s_n$ :

$$\theta_{n+1} = \theta_n - \alpha_n \nabla_{\theta} G(\theta_n, s_n) \quad (365)$$

- 4 Repeat until stopping criterion is fulfilled, e.g.,  $\|\nabla_{\theta} G\| < TOL$  or  $\|\theta_{n+1} - \theta_n\| < TOL$  or their relative stopping criteria versions.

# Deep Ritz

- 1 Variational approximation in the energy form rather than in the strong form (as before)
- 2 Energy forms do however only exist for symmetric problems
- 3 Example: Classical elasticity (solid mechanics)
- 4 Counter example: Navier-Stokes equations (fluid mechanics)

- ① Specific example:

$$\min_{u \in H} I(u) \quad (366)$$

with

$$I(u) = \int_{\Omega} \left( \frac{1}{2} |\nabla u(x)|^2 - f(x)u(x) \right) dx \quad (367)$$

where  $H$  is the set of admissible functions (see PDE numerics classes)

# Deep Ritz: algorithm

- 1 Deep neural network-based approximation of the trial function  $u(x)$
- 2 A numerical quadrature rule for  $I(u)$ , namely the arising integral therein (somewhat similar as in FEM when evaluating the local integrals)<sup>78</sup>
- 3 Algorithm for solving the arising optimization problem (determining the optimal parameters; training, same procedure as for PINN or DGM)

---

<sup>78</sup>Wick, *Numerical methods for partial differential equations*.

- 1 Variational approximation in a weak form (conceptionally similar to FEM)
- 2 Petrov-Galerkin: trial and test functions differ
- 3 Trial space: approximation via neural network
- 4 Test space: Legendre polynomials (hopefully known from Numerik 1)
- 5 Allows to approximate larger classes of problems than Deep Ritz (non-symmetric PDEs know as well)
- 6 Closer to classical PDE numerics since weak forms (rather than strong forms) are much more often available/considered in modern numerics
- 7 Less regularity in function spaces needed than for strong forms

- 1 Nonlinear neural network approximation only in trial function
- 2 Test space due to Legendre polynomials linear (as in FEM)
- 3 Challenge: approximation of integrals arising in the weak formulation  
→ analytical expressions for integrals in general not available
- 4 Trick: the authors of VPINN consider a shallow network with only one hidden layer (not deep!) and indeed compute explicitly the integrals and derivatives

# Summary of methods in this class

- PINN<sup>79</sup>
- DGM<sup>80</sup>
- Deep Ritz<sup>81</sup>
- vPINN<sup>82</sup>

---

<sup>79</sup>Raissi, Perdikaris, and Karniadakis, “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”.

<sup>80</sup>Sirignano and Spiliopoulos, “DGM: A deep learning algorithm for solving partial differential equations”.

<sup>81</sup>E and Yu, “The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems”.

<sup>82</sup>Kharazmi, Zhang, and Karniadakis, *Variational Physics-Informed Neural Networks For Solving Partial Differential Equations*.



## Review papers; further reading

- <https://arxiv.org/pdf/2105.09506.pdf>
- <https://www.nature.com/articles/s42254-021-00314-5.pdf>
- <https://arxiv.org/pdf/2201.05624.pdf>

# Extensions of PINNs

- Self-adaptive PINN <https://arxiv.org/pdf/2009.04544.pdf>
- Gradient-enhanced PINNs  
<https://arxiv.org/pdf/2111.02801.pdf>
- Hidden Fluid Mechanics <https://arxiv.org/pdf/1808.04327.pdf>
- Physics-Informed Neural Networks with Hard Constraints for Inverse Design <https://epubs.siam.org/doi/pdf/10.1137/21M1397908>

End of Lecture 13

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Neural Operators

- ① DeepONet
- ② Fourier Neural Operators (FNO)
- ③ Further topics and current developments

# DeepONet: main literature

- DeepONet paper<sup>83</sup>
- Extensions of DeepONet<sup>84</sup>
- YouTube video by Karniadakis:  
<https://www.youtube.com/watch?v=1bS0q0RkoH0>

---

<sup>83</sup>Lu Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (2021), pp. 218–229. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00302-5. URL: <https://doi.org/10.1038/s42256-021-00302-5>.

<sup>84</sup>Lu Lu et al. *A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data*. 2021. arXiv: 2111.05512 [physics.comp-ph].

- 1 Lecture 6: UAT: NN are universal approximators of continuous functions
  - 2 Now: NN with a single hidden layer can approximate any nonlinear continuous operator
- UAT of operators
- 3 **DeepONet: Deep Operator Network**
  - 4 1st DNN for encoding the discrete input function space: **branch net**
  - 5 2nd DNN for encoding the domain of the output function: **trunk net**
  - 6 DeepONet can learn operators such as integrals and fractional Laplacians, deterministic and stochastic differential equations

# DeepONet

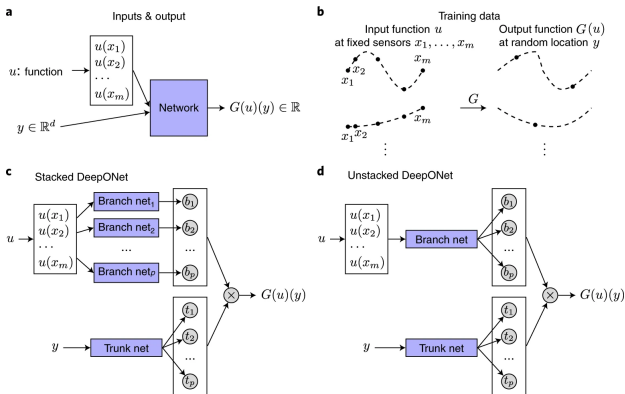


Image source: Figure 1 from DeepONet paper<sup>85</sup>

<sup>85</sup>Lu et al., “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”.



## DeepONet: Theorem 1: assumptions

- 1 Let  $\sigma$  be a continuous non-polynomial function
- 2  $X$  is a Banach space
- 3  $K_1 \subset X$  compact set
- 4  $K_2 \subset \mathbb{R}^d$  compact set
- 5  $V \subset C(K_1)$  compact set
- 6 Usual norm  $\|u\|_{C(K_1)} = \max_{x \in K_1} |u(x)|$
- 7 Nonlinear continuous operator:

$$G : V \rightarrow C(K_2) \tag{368}$$

# DeepONet: Theorem 1: statement

## Theorem

Let the previous assumptions hold true. Then, for any  $\epsilon > 0$ , there are positive  $n, p, m \in \mathbb{N}$  and constants  $c_i^k, \xi_{ij}^k, \theta_i^k, \zeta_k \in \mathbb{R}$ , and  $w_k \in \mathbb{R}^d, x_j \in K_1$  for  $i = 1, \dots, n$  and  $k = 1, \dots, p$ , and  $j = 1, \dots, m$  such that

$$\left| G(u)(y) - \sum_{k=1}^p \sum_{i=1}^n c_i^k \sigma \left( \sum_{j=1}^m \xi_{ij}^k u(x_j) + \theta_i^k \right) \sigma(w_k \cdot y + \zeta_k) \right| < \epsilon \quad (369)$$

holds for all  $u \in V$  and  $y \in K_2$

## Proof.

We refer to Chen, Chen; Universal approximation to nonlinear operators by neural networks with arbitrary activation functions and its application to dynamical systems, IEEE Trans. Neural Networks, Vol. 6, pp. 911-917, 1995 □

# DeepONet: branch and trunk

① Branch:

$$\sum_{i=1}^n c_i^k \sigma \left( \sum_{j=1}^m \xi_{ij}^k u(x_j) + \theta_i^k \right) \quad (370)$$

② Trunk:

$$\sigma(w_k \cdot y + \zeta_k) \quad (371)$$

# DeepONet: architecture

- 1 Sensors  $\{x_1, x_2, \dots, x_m\}$
- 2 1st network input  $[u(x_1), \dots, u(x_m)]^T$
- 3 2nd network input  $y \in \mathbb{R}^d$
- 4 Since dimensions may be different, namely  $m$  and  $d$ , two networks are required in general
- 5 Previous theorem guides possible network structure

# DeepONet: trunk and branch

- 1 Takes  $y$  as input
- 2 Output  $[t_1, t_2, \dots, t_p]^T \in \mathbb{R}^p$
- 3  $p$  branch networks
- 4 Each branch takes  $[u(x_1), u(x_2), \dots, u(x_m)]^T$  as input
- 5 Branch output  $b_k \in \mathbb{R}$  for  $k = 1, \dots, p$
- 6 Merging:

$$G(u)(y) \approx \sum_{k=1}^p b_k(u(x_1), u(x_2), \dots, u(x_m)) t_k(y) \quad (372)$$

- 7 In practice  $p \geq 10$  and using only one single branch network with output  $[b_1, \dots, b_p]^T \in \mathbb{R}^p$

# DeepONet: data generation

- 1 One data point is a triplet of the form

$$(u, y, G(u)(y)) \quad (373)$$

- 2 Example: dataset of size 10 000 may be generated for 100  $u$  and each is evaluated via  $G(u)(y)$  at 100 different  $y$  locations
  - 3 Task in DeepONet: Use input  $[u(x_1), \dots, u(x_m)]$  for representing  $u(x)$
- Estimate how many sensors  $m$  are required to achieve accuracy of  $\epsilon$  (Theorem from before)

# DeepONet: data generation

- 1 Example via ODE system
- 2 Consider IVP

$$s'(x) = g(s, u, x) \quad (374)$$

$$s(a) = s_0 \quad (375)$$

- 3 Input  $u \in V \subset C[a, b]$
- 4 Output  $s : [a, b] \rightarrow \mathbb{R}^K$
- 5 Operator  $G$  defined by

$$G(u)(x) = s_0 + \int_a^x g(G(u)(t), u(t), t) dt \quad (376)$$

# DeepONet: approximation theorem

- 1 Nodal points:  $x_j = a + j(b - a)/m$  for  $j = 0, \dots, m$
- 2 Define function  $u_m(x)$  via

$$u_m(x) = u(x_j) + \frac{u(x_{j+1}) - u(x_j)}{x_{j+1} - x_j}(x - x_j), \quad (377)$$

for  $x_j \leq x \leq x_{j+1}$  and  $j = 0, \dots, m - 1$

- 3 Operator mapping  $L_m : u \mapsto u_m$
- 4  $U_m = \{L_m(u) | u \in V\} \subset C[a, b]$  (compact since  $V$  is compact and  $L_m$  continuous)
- 5 For  $u \in V$  and  $u_m \in U_m$  there exists a constant  $\kappa(m, V)$  such that

$$\max_{x \in [a, b]} |u(x) - u_m(x)| \leq \kappa(m, V), \quad \kappa(m, V) \rightarrow 0 \quad \text{as } m \rightarrow \infty \quad (378)$$



# DeepONet: approximation theorem

It holds

## Theorem

Let  $m \in \mathbb{N}$  such that  $c(b-a)\kappa(m, V)e^{c(b-a)} < \epsilon$ . Then, for any  $d \in [a, b]$ , there exist matrices and biases  $W_1 \in \mathbb{R}^{n \times (m+1)}$ ,  $b_1 \in \mathbb{R}^{m+1}$ ,  $W_2 \in \mathbb{R}^{K \times n}$ ,  $b_2 \in \mathbb{R}^K$  such that

$$\left\| G(u)(d) - \left( W_2 \cdot \sigma(W_1 \cdot [u(x_0) \cdots u(x_m)]^T + b_1) + b_2 \right) \right\| < \epsilon. \quad (379)$$

## Proof.

We refer to Lu Lu et al. 2021. □

# Fourier Neural Operators: main literature

- FNO paper<sup>86</sup>
- FNO are universal approximators:  
<https://arxiv.org/pdf/2107.07562.pdf> (main theorem also mentioned in<sup>87</sup>)
- Extensions of FNO<sup>88</sup>

---

<sup>86</sup>Zongyi Li et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2021. arXiv: 2010.08895 [cs.LG].

<sup>87</sup>Lu et al., *A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data*.

<sup>88</sup>Ibid.

# Fourier Neural Operators

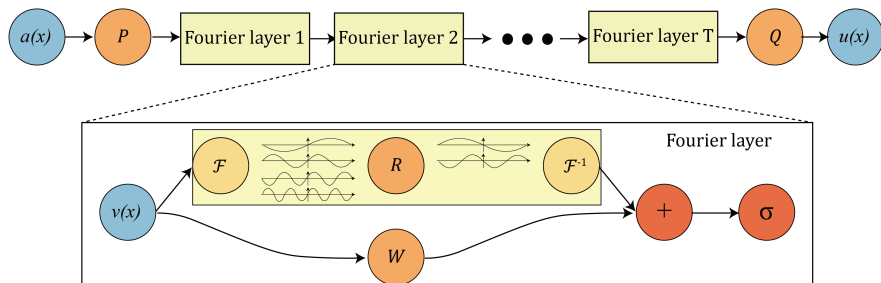


Image source: Figure 2 from Fourier Neural Operators paper<sup>89</sup>

<sup>89</sup>Li et al., *Fourier Neural Operator for Parametric Partial Differential Equations*.

# Fourier Neural Operators: motivation

- 1 **Problem:** Learning and using neural operators (e.g., DeepONet) are not yet numerically efficient
- 2 **One idea:** Use fast Fourier transform (FFT), e.g.,<sup>90</sup> [Section 8.9]
- 3 FFT not completely new in neural networks, e.g., used in the theory of UAT<sup>91</sup>
- 4 FFT used to speed-up CNN<sup>92</sup>
- 5 In the following: **Fourier neural operator**

---

<sup>90</sup>Richter and Wick, *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*.

<sup>91</sup>Hornik, Stinchcombe, and White, "Multilayer feedforward networks are universal approximators".

<sup>92</sup>M. Mathieu, M. Henaff, and Y. LeCun. *Fast training of convolutional networks through FFTs*. Preprint. 2013.

# Fourier Neural Operators: basics

- 1 Procedure: learning mappings between two infinite dimensional spaces (like in DeepONet)
- 2 Let  $\mathcal{A}$  and  $\mathcal{U}$  be separable Banach spaces <sup>93</sup>
- 3 Let  $G^+ : \mathcal{A} \rightarrow \mathcal{U}$  be a nonlinear mapping
- 4 We have maps in mind for parametric PDEs (PPDE):

$$G : \mathcal{A} \times \Theta \rightarrow \mathcal{U} \quad (380)$$

with a finite-dimensional parameter space  $\Theta$

- 5 Cost functional:  $C : \mathcal{U} \times \mathcal{U} \rightarrow \mathbb{R}$
- 6 Minimization problem:

$$\min_{\theta \in \Theta} E_{a \sim \mu} [C(G(a, \theta), G^+(a))] \quad (381)$$

---

<sup>93</sup>See Dirk Werner; FA, 2018

# Fourier Neural Operators: Learning operators

- 1 Clearly, approximating  $G^+$  much more difficult than simply computing a solution  $u \in \mathcal{U}$  of a PDE for a given parameter  $\theta \in \Theta$
- 2 Classical discretizations, such as FD (finite differences), FEM (finite element method), FV (finite volumes) or even PINNs (Lecture 12) 'only' approximate one solution of a PDE
- 3 Learning the operator itself, may significantly reduce the computational cost

# Neural operator

- 1 Iterative architecture:

$$v_0 \mapsto v_1 \mapsto \dots \mapsto v_T \quad (382)$$

with a sequence of functions  $v_j \in \mathbb{R}^{d_v}$

- 2 Input  $a \in \mathcal{A}$  lifted to higher dimensional representation  
 $v_0(x) = P(a(x))$
- 3 Local transformation  $P$  parametrized by a shallow fully-connected NN
- 4 Apply several iterations of updates  $v_t \mapsto v_{t+1}$ , which are compositions of non-local integral operator  $\mathcal{K}$  and a (usual) local, nonlinear activation function  $\sigma$
- 5 Output:  $u(x) = Q(v_T(x))$  where  $Q : \mathbb{R}^{d_v} \rightarrow \mathbb{R}^{d_u}$

# Neural operator

## Definition (Iterative updates)

The updates  $v_t \mapsto v_{t+1}$  are defined by

$$v_{t+1}(x) := \sigma(Wv_t(x) + (\mathcal{K}(a, \phi)v_t)(x)) \quad (383)$$

for all  $x \in D$ . Therein,

$$\mathcal{K} : \mathcal{A} \times \Theta_{\mathcal{K}} \rightarrow \mathcal{L}(\mathcal{U}, \mathcal{U}) \quad (384)$$

is a kernel integral transformation. In words,  $\mathcal{K}$  maps to bounded linear operators on  $\mathcal{U}$ . Moreover,  $W : \mathbb{R}^{d_v} \rightarrow \mathbb{R}^{d_v}$  is a linear transformation and as usual  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ .



# Neural operator

## Definition (Kernel integral operator)

The kernel integral operator is defined by

$$(\mathcal{K}(a, \phi)v_t)(x) := \int_D \kappa(x, y, a(x), a(y), \phi)v_t(y) dy \quad (385)$$

for all  $x \in D$  and where

$$\kappa_\phi : \mathbb{R}^{2(d+d_a)} \rightarrow \mathbb{R}^{d_v \times d_v} \quad (386)$$

is a neural network parametrized by  $\phi \in \Theta_{\mathcal{K}}$ , specifically  $\kappa_\phi$  is learned from data.

# Fourier neural operator

- 1 In case  $a \in \mathcal{A}$  is removed, such that  $\kappa(x, y, \phi) = \kappa_\phi(x - y)$ , we obtain a classical **convolution** operator<sup>94</sup>
- 2 Idea:<sup>95</sup> Parametrize  $\kappa_\phi$  directly in the Fourier space and use FFT for computing (385).

---

<sup>94</sup>See e.g., D. Werner; Funktionalanalysis, 2018

<sup>95</sup>Li et al., *Fourier Neural Operator for Parametric Partial Differential Equations*.

## Fourier neural operator

- 1 Let  $\mathcal{F}$  denote the Fourier transform of a function  $f : D \rightarrow \mathbb{R}^{d_v}$ :

$$(\mathcal{F}f)_j(k) = \int_D f_j(x) e^{-2i\pi \langle x, k \rangle} dx \quad (387)$$

- 2 Inverse Fourier transform:

$$(\mathcal{F}^{-1}f)_j(x) = \int_D f_j(k) e^{2i\pi \langle x, k \rangle} dk \quad (388)$$

for  $j = 1, \dots, d_v$  and  $i$  (is NOT an index), but the imaginary unit  
 $i^2 = -1$

- 3 Use

$$\kappa(x, y, a(x), a(y), \phi) = \kappa_\phi(x - y) \quad (389)$$

- 4 Convolution theorem:

$$(\mathcal{K}(a, \phi)v_t)(x) = \mathcal{F}^{-1}(\mathcal{F}(\kappa_\phi) \cdot \mathcal{F}(v_t))(x) \quad (390)$$

for all  $x \in D$

# Fourier neural operator

## Definition (Fourier integral operator)

The Fourier integral operator is defined by

$$(\mathcal{K}(\phi)v_t)(x) = \mathcal{F}^{-1}(R_\phi \cdot (\mathcal{F}v_t))(x) \quad \forall x \in D \quad (391)$$

where  $R_\phi$  is the Fourier transform of a periodic function  $\kappa : \bar{D} \rightarrow \mathbb{R}^{d_v \times d_v}$  parametrized by  $\phi \in \Theta_{\mathcal{K}}$

# Fourier neural operator

- 1 Since  $\kappa$  is periodic, we have a Fourier series expansion  
→ Discrete modes  $k \in \mathbb{Z}^d$
- 2 Truncating Fourier series at a maximal number of modes  $k_{max}$
- 3 Parametrize  $R := R_\phi$  as a complex-valued tensor as a collection of truncated Fourier modes

## Discrete case and FFT

- 1 Let  $D$  be discretized with  $n \in \mathbb{N}$  points
- 2 We have  $v_t \in \mathbb{R}^{n \times d_v}$
- 3  $\mathcal{F}(v_t) \in \mathbb{C}^{n \times d_v}$
- 4 Truncation:  $\mathcal{F}(v_t) \in \mathbb{C}^{k_{\max} \times d_v}$
- 5 Weight tensor  $R \in \mathbb{C}^{k_{\max} \times d_v \times d_v}$
- 6 Then:

$$(R \cdot (\mathcal{F}v_t))_{k,l} = \sum_{j=1}^{d_v} R_{k,l,j} (\mathcal{F}v_t)_{k,j}, \quad (392)$$

for  $k = 1, \dots, k_{\max}$  and  $j = 1, \dots, d_v$

# Discrete case and FFT

- 1 Uniform discretization with resolution  $s_1 \times \dots \times s_d = n$
- 2  $x = (s_1, \dots, s_d) \in D$
- 3 Definition of FFT:

$$(\hat{\mathcal{F}}f)_l(k) = \sum_{x_1=0}^{s_1-1} \dots \sum_{x_d=0}^{s_d-1} f_l(x_1, \dots, x_d) e^{-2i\pi \sum_{j=1}^d \frac{x_j k_j}{s_j}} \quad (393)$$

- 4 Definition of inverse FFT:

$$(\hat{\mathcal{F}}^{-1}f)_l(k) = \sum_{k_1=0}^{s_1-1} \dots \sum_{k_d=0}^{s_d-1} f_l(k_1, \dots, k_d) e^{2i\pi \sum_{j=1}^d \frac{x_j k_j}{s_j}} \quad (394)$$

for  $l = 1, \dots, d_v$

## Short summary of this semester

- 1 In particular, the last lecture L14 shows that **functional analysis (FA)** and **analysis** classes are fundamental in order to **design accurate, efficient, and robust numerical algorithms**
- 2 On the first glance this is somewhat surprising within machine learning, artificial intelligence, and neural networks since often we rather hear or speak about 'algorithms', 'data', and 'powerful computing machines'
- 3 However when taking a closer look (and going a bit deeper), this is not any surprise at all and well-known for decades that the ground of **good numerics** is **rigorous mathematical theory**



# Outlook (I)

- 1 Mathematics of deep learning<sup>96</sup>
- 2 Solving inverse problems using data-driven models<sup>97</sup>
- 3 Combining different areas of numerics, e.g. POD and DeepONet<sup>98</sup>

---

<sup>96</sup>Julius Berner et al. *The Modern Mathematics of Deep Learning*. 2021. arXiv: 2105.04026 [cs.LG].

<sup>97</sup>Simon Arridge et al. "Solving inverse problems using data-driven models". In: *Acta Numerica* 28 (2019), pp. 1–174. DOI: 10.1017/S0962492919000059.

<sup>98</sup>Lu et al., *A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data*.

## Outlook (II)

- 1 Transfer learning for phase-field fracture:<sup>99</sup>
- 2 DeepONet for phase-field fracture<sup>100</sup>
- 3 Current developments are also related to some of the project topics of this class

---

<sup>99</sup>Somdatta Goswami et al. "Transfer learning enhanced physics informed neural network for phase-field modeling of fracture". In: *Theoretical and Applied Fracture Mechanics* 106 (2020), p. 102447. ISSN: 0167-8442. DOI: <https://doi.org/10.1016/j.tafmec.2019.102447>. URL: <https://www.sciencedirect.com/science/article/pii/S016784421930357X>.

<sup>100</sup>Somdatta Goswami et al. "A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials". In: *Computer Methods in Applied Mechanics and Engineering* 391 (2022), p. 114587. ISSN: 0045-7825. DOI: <https://doi.org/10.1016/j.cma.2022.114587>. URL: <https://www.sciencedirect.com/science/article/pii/S004578252200010X>.

## Outlook (III), some own ideas and developments

- 1 Learning the adjoint equation in goal-oriented error estimation (strong form of the equations as in Lecture 12)<sup>101</sup>
  - Possible extension: work in weak form (VPINNs ?!) or Deep Ritz Method
- 2 Possible idea: data-driven methods for phase-field fracture or fluid-structure interaction within optimal control
  - Why? Several forward runs necessary. Approximation via NN may significantly reduce numerical efforts
- 3 Recall: for repeated runs of forward models, NN and MOR (model order reduction) may be cost-efficient ways

---

<sup>101</sup> Julian Roth, Max Schröder, and Thomas Wick. “Neural network guided adjoint computations in dual weighted residual error estimation”. In: *SN Applied Sciences* 4 (2022). DOI: <https://doi.org/10.1007/s42452-022-04938-9>.

# Refresh finally the topics we have had in this class

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

End of Lecture 14

**End of this course in this semester.**

## 1. Traditional AI

- 1.1 Lecture 1: Algorithmic Systems, Numerical Concepts, Notation
- 1.2 Lecture 2: Introduction to Probability, Random Processes, Statistics
- 1.3 Lecture 3: Fundamental Algorithms
- 1.4 Lecture 4: Dimensionality Reduction

## 2. Deep Learning in Neural Networks

- 2.1 Lecture 5: Artificial Neural Networks (ANN)
- 2.2 Lecture 6: Universal Approximation Theorem
- 2.3 Lecture 7: Convolutional Neural Networks (CNN)
- 2.4 Lecture 8: Recurrent Neural Networks (RNN)
- 2.5 Lecture 9: Transformer
- 2.6 Lecture 10: Reinforcement Learning (RL)

## 3. Applications to (and with) Differential Equations

- 3.1 Lecture 11: Introduction to ML for Scientific Computing
- 3.2 Lecture 12: Neural ODE
- 3.3 Lecture 13: PINNs: Physics-Informed Neural Networks
- 3.4 Lecture 14: Neural Operators and Outlook

## 4. Projects

# Example Projects

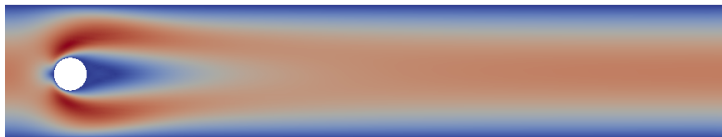
- ① PINNs for Navier-Stokes Equations
- ② Model Order Reduction for parametric PDEs
- ③ Goal-Oriented Self-Adaptive PINNs

# Project 1: PINNs for Navier-Stokes Equations

Solving the stationary Navier-Stokes 2D-1 benchmark problem with Physics Informed Neural Networks.

$$-\mu\Delta v + \rho(v \cdot \nabla)v + \nabla p = 0 \quad (395)$$

$$\nabla \cdot v = 0 \quad (396)$$





## Project 2: Model Order Reduction for parametric PDEs

- Model order reduction of parametric heat equation
- Generate low-dimensional approximation space by Proper Orthogonal Decomposition (POD)
  - ⇒ Provide rapidly computable approximations

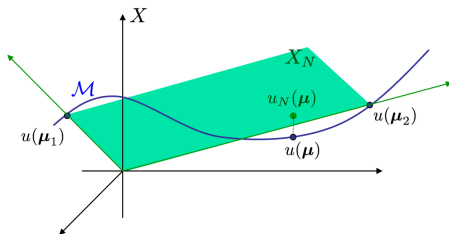


Image source: Figure 1 from Reduced Basis tutorial<sup>102</sup>

<sup>102</sup>Bernard Haasdonk. “Reduced basis methods for parametrized PDEs—a tutorial introduction for stationary and instationary problems”. In: *Model reduction and approximation: theory and algorithms* 15 (2017), p. 65.

# Project 3: Goal-Oriented Self-Adaptive PINNs (I): Problem statement

- 1 We want to solve a general PDE of the form:

$$\begin{aligned}\mathcal{N}_{x,t}[u(x,t)] &= 0, & x \in \Omega, t \in (0, T), \\ u(x,t) &= g(x,t), & x \in \partial\Omega, t \in (0, T), \\ u(x,0) &= h(x), & x \in \Omega,\end{aligned}$$

- 2 where  $\Omega \times [0, T] \subset \mathbb{R}^{d+1}$  is the space-time cylinder with spatial dimension  $d$  and  $u : \Omega \times [0, T] \rightarrow \mathbb{R}$ .

## Project 3: Goal-Oriented Self-Adaptive PINNs (II)

- 1 From this, following the PINN approach, we obtain a loss-function of the form

$$\mathcal{L}(\theta, \mathcal{T}) = w_1 \mathcal{L}_f(\theta, \mathcal{T}_f) + w_2 \mathcal{L}_0(\theta, \mathcal{T}_0) + w_3 \mathcal{L}_b(\theta, \mathcal{T}_b),$$

with  $w_1, w_2, w_3 > 0$  and  $\mathcal{L}_f, \mathcal{L}_0$ , and  $\mathcal{L}_b$  are similar to those defined on the next slide.

- 2 **Q:** How to choose the weights  $w_1, w_2, w_3$ ?
- 3 **A:** Self-Adaptive PINNs.<sup>103</sup>

---

<sup>103</sup>Levi D. McClenny and Ulisses M. Braga-Neto. “Self-Adaptive Physics-Informed Neural Networks using a Soft Attention Mechanism”. In: *CoRR* abs/2009.04544 (2020). arXiv: 2009.04544. URL: <https://arxiv.org/abs/2009.04544>.

## Project 3: Goal-Oriented Self-Adaptive PINNs (III)

- 1 The idea is, to employ weights to each summand of each term of the loss function to obtain:

$$\mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}) = \mathcal{L}_f(\theta, \lambda_f, \mathcal{T}_f) + \mathcal{L}_0(\theta, \lambda_0, \mathcal{T}_0) + \mathcal{L}_b(\theta, \lambda_b, \mathcal{T}_b),$$

- 2 where  $\lambda_f = (\lambda_f^1, \dots, \lambda_f^{N_f})$ ,  $\lambda_0 = (\lambda_0^1, \dots, \lambda_0^{N_0})$ ,  $\lambda_b = (\lambda_b^1, \dots, \lambda_b^{N_b})$  and

$$\mathcal{L}_f(\theta, \lambda_f, \mathcal{T}_f) = \frac{1}{N_f} \sum_{i=1}^{N_f} |\lambda_f^i \mathcal{N}_{x,t}[\hat{u}(x_f^i, t_f^i, \theta)]|^2$$

$$\mathcal{L}_0(\theta, \lambda_0, \mathcal{T}_0) = \frac{1}{N_0} \sum_{i=1}^{N_0} |\lambda_0^i (\hat{u}_\theta(x_0^i, 0, \theta) - h(x_0^i))|^2$$

$$\mathcal{L}_b(\theta, \lambda_b, \mathcal{T}_b) = \frac{1}{N_b} \sum_{i=1}^{N_b} |\lambda_b^i (\hat{u}_\theta(x_b^i, t_b^i, \theta) - g(x_b^i))|^2.$$

## Project 3: Goal-Oriented Self-Adaptive PINNs (IV)

- 1 Involving the adaptation of the additional Self-Adaptive weights to the usual training process leads to a min max problem:

$$\min_{\theta} \max_{\lambda_f, \lambda_0, \lambda_b} \mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}).$$

- 2 In practice this is done with a gradient descent/ascent procedure (again see also L2, L3 and L5)

$$\begin{aligned}\theta^{k+1} &= \theta^k - \eta_k \nabla_{\theta} \mathcal{L}(\theta^k, \lambda_f^k, \lambda_0^k, \lambda_b^k, \mathcal{T}), \\ \lambda_f^{k+1} &= \lambda_f^k + \eta_k \nabla_{\lambda_f} \mathcal{L}(\theta^k, \lambda_f^k, \lambda_0^k, \lambda_b^k, \mathcal{T}), \\ \lambda_0^{k+1} &= \lambda_0^k + \eta_k \nabla_{\lambda_0} \mathcal{L}(\theta^k, \lambda_f^k, \lambda_0^k, \lambda_b^k, \mathcal{T}), \\ \lambda_b^{k+1} &= \lambda_b^k + \eta_k \nabla_{\lambda_b} \mathcal{L}(\theta^k, \lambda_f^k, \lambda_0^k, \lambda_b^k, \mathcal{T}),\end{aligned}$$

where  $\eta_k$  is the learning rate at step  $k$ .

## Project 3: Goal-Oriented Self-Adaptive PINNs (V)

① We find

$$\nabla_{\lambda_f} \mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}) \geq 0,$$

$$\nabla_{\lambda_0} \mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}) \geq 0,$$

$$\nabla_{\lambda_b} \mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}) \geq 0,$$

② such that

$$\{\lambda_f^k, k = 1, 2, \dots\},$$

$$\{\lambda_0^k, k = 1, 2, \dots\},$$

$$\{\lambda_b^k, k = 1, 2, \dots\},$$

are monotonically nondecreasing under the assumption that the initial weights are nonnegative.

## Project 3: Goal-Oriented Self-Adaptive PINNs (VI)

- 1 Now, we want to apply the SAPINN approach on goal functionals
- 2 Motivation: we want to solve the PDE while controlling/minimizing the error in a specific goal functional
- 3 **What are goal functionals?**
  - Without any further postprocessing, we obtain global numerical solutions and error analyses
- 4 From a technical (engineering) point of view, however, very often not the entire solution is of interest, but only **parts** of it

## Project 3: Goal-Oriented Self-Adaptive PINNs (VII)

- 1 Such 'parts' of a solution can be: parts of the domain, such as subdomains, lines, point evaluations
- 2 Such 'parts' can also be: only parts of solution components, e.g., if vectors in  $u \in \mathbb{R}^n$  then, e.g., goal functional only controls  $\tilde{u} \in \mathbb{R}^m$ , e.g., first  $m$  components, with  $m < n$
- 3 Usually this **greater flexibility** in obtaining information with the help of goal functionals, results in a **higher computational cost** ('no free lunch')

→ GOSAPINN: goal-oriented self-adaptive PINNs



## Project 3: Goal-Oriented Self-Adaptive PINNs (VIII)

- 1 Challenge: application of self-adaptive weights to the goal-functional due to integrals
- 2 Considering a goal functional  $J(\cdot) \in \mathbb{R}$ , we want to minimize the error

$$|J(\hat{u}(x, t, \theta)) - J(u(x, t))|,$$

in addition to satisfying the PDE as closely as possible.

- 3 This leads to an additional loss function

$$\mathcal{L}_J(\theta, \mathcal{T}_J) = |\tilde{J}(\hat{u}(x, t, \theta)) - \tilde{J}(u(x, t))|^2,$$

where  $\tilde{J}$  is an approximation with a suitable quadrature formula.

## Project 3: Goal-Oriented Self-Adaptive PINNs (IX)

We follow three approaches:

- 1 No Self-Adaptive weights:

$$\begin{aligned}\mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \mathcal{T}) &= \mathcal{L}_f(\theta, \lambda_f, \mathcal{T}_f) + \mathcal{L}_0(\theta, \lambda_0, \mathcal{T}_0) \\ &\quad + \mathcal{L}_b(\theta, \lambda_b, \mathcal{T}_b) + \mathcal{L}_J(\theta, \mathcal{T}_J),\end{aligned}$$

- 2 One Self-Adaptive weight:

$$\begin{aligned}\mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \lambda_J, \mathcal{T}) &= \mathcal{L}_f(\theta, \lambda_f, \mathcal{T}_f) + \mathcal{L}_0(\theta, \lambda_0, \mathcal{T}_0) \\ &\quad + \mathcal{L}_b(\theta, \lambda_b, \mathcal{T}_b) + \mathcal{L}_J(\theta, \lambda_J, \mathcal{T}_J),\end{aligned}$$

with

$$\mathcal{L}_J(\theta, \lambda_J, \mathcal{T}_J) = |\lambda_J(\tilde{J}(\hat{u}(x, t, \theta)) - \tilde{J}(u(x, t)))|^2,$$

- 3 Multiple weights applied (one on each quadrature point):

$$\mathcal{L}_J(\theta, \lambda_J, \mathcal{T}_J) = |\tilde{J}(\hat{u}(x, t, \theta), \lambda_J) - \tilde{J}(u(x, t), \lambda_J)|^2.$$

## Project 3: Goal-Oriented Self-Adaptive PINNs (X)

- 1 Then, we train the Neural network, as before, via

$$\min_{\theta} \max_{\lambda_f, \lambda_0, \lambda_b, \lambda_J} \mathcal{L}(\theta, \lambda_f, \lambda_0, \lambda_b, \lambda_J, \mathcal{T}).$$

## Project 3: Goal-Oriented Self-Adaptive PINNs (XI)

- 1 We perform some experimental tests of the previously introduced idea on the 1+1D heat equation:
- 2 Find  $u : [0, 1]^2 \rightarrow \mathbb{R}$  such that

$$\begin{aligned}\partial_t u - \Delta u &= 0, & (x, t) &\in (0, 1) \times (0, 1), \\ u(0, t) = u(1, t) &= 0, & t &\in (0, 1), \\ u(x, 0) &= \sin(\pi x), & x &\in (0, 1).\end{aligned}$$

- 3 An analytical solution of this IBVP (initial-boundary value problem) is given by

$$u(x, t) = \sin(\pi x) \cdot \exp(-\pi^2 t).$$

## Project 3: Goal-Oriented Self-Adaptive PINNs (XII)

- ① We consider three different goal functionals:

$$J_1(\phi) = \int_{0.4}^{0.6} \int_{0.4}^{0.6} \phi(u - \phi) \, dxdt,$$

$$J_2(\phi) = \int_{0.4}^{0.6} \int_{0.4}^{0.6} \phi \, dxdt,$$

$$J_3(\phi) = \int_{0.4}^{0.6} \int_{0.4}^{0.6} \phi^2 \, dxdt,$$

- ② where  $u$  is the exact solution defined on the previous slide.

## Project 3: Goal-Oriented Self-Adaptive PINNs (XIII)

- 1 We train the neural network by minimizing the goal functional error given by

$$|J_1(\hat{u}(\theta, x, t)) - J_1(u(x, t))| \approx \left| \frac{0.04}{N_J^x \cdot N_J^t} \sum_{i=1}^{N_J^x} \sum_{j=1}^{N_J^t} (\hat{u}(\theta, x^i, t^j) - u(x^i, t^j))^2 \right|,$$
$$|J_2(\hat{u}(\theta, x, t)) - J_2(u(x, t))| \approx \left| \frac{0.04}{N_J^x \cdot N_J^t} \sum_{i=1}^{N_J^x} \sum_{j=1}^{N_J^t} \hat{u}(\theta, x^i, t^j) - u(x^i, t^j) \right|,$$
$$|J_3(\hat{u}(\theta, x, t)) - J_3(u(x, t))| \approx \left| \frac{0.04}{N_J^x \cdot N_J^t} \sum_{i=1}^{N_J^x} \sum_{j=1}^{N_J^t} \hat{u}(\theta, x^i, t^j)^2 - u(x^i, t^j)^2 \right|,$$

- 2 These are approximated via a Monte-Carlo quadrature<sup>104</sup> given by:

$$\int_{\Omega} f(x) dx \approx \frac{|\Omega|}{N} \sum_{i=1}^N f(x_i).$$

---

<sup>104</sup>Stefan Weinzierl. *Introduction to Monte Carlo methods*. 2000. arXiv: hep-ph/0006269 [hep-ph].

## Project 3: Goal-Oriented Self-Adaptive PINNs (XIV)

In our numerical experiments, we compare four different loss functions for the goal functional error:

- 1 goal functional error not considered:

$$\mathcal{L}_J = 0,$$

- 2 no Self-Adaptive weights:

$$\mathcal{L}_J(\theta, \mathcal{T}_J) = \left| \frac{0.04}{N_j^x \cdot N_j^t} \sum_{i=1}^{N_j^x} \sum_{j=1}^{N_j^t} (\hat{u}(\theta, x^i, t^j) - u(x^i, t^j))^2 \right|,$$

- 3 One Self-Adaptive weight:

$$\mathcal{L}_J(\theta, \lambda_J, \mathcal{T}_J) = \left| \lambda_J \frac{0.04}{N_j^x \cdot N_j^t} \sum_{i=1}^{N_j^x} \sum_{j=1}^{N_j^t} (\hat{u}(\theta, x^i, t^j) - u(x^i, t^j))^2 \right|,$$

- 4 One Self-Adaptive weight for each quadrature point:

$$\mathcal{L}_J(\theta, \lambda_J, \mathcal{T}_J) = \left| \frac{0.04}{N_j^x \cdot N_j^t} \sum_{i=1}^{N_j^x} \sum_{j=1}^{N_j^t} \lambda_j^{i,j} (\hat{u}(\theta, x^i, t^j) - u(x^i, t^j))^2 \right|.$$

## Project 3: Goal-Oriented Self-Adaptive PINNs (XV)

- Observed quantities: The goal functional error of the four approaches (again approximated via Monte-Carlo)
- Input layer of size 2
- 3 hidden layers of 20 neurons each
- Output layer of size 1
- Optimization Algorithm: Adam optimization algorithm<sup>105</sup> and L-BFGS algorithm<sup>106</sup> to solve the min max problem
- Used software library: TensorFlow
- Number of points selected for training/quadrature:
  - $N_f = 10000$
  - $N_0 = 100$
  - $N_b = 200$
  - $N_J = N_J^x \cdot N_J^t = 51 \cdot 21 = 1071$

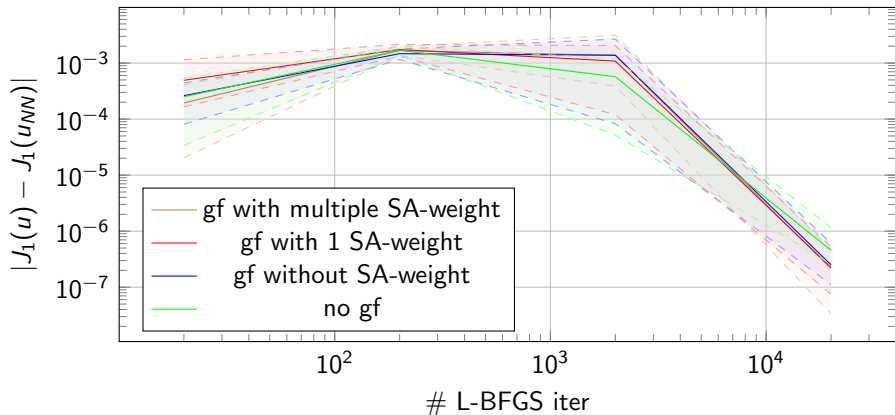
---

<sup>105</sup>Kingma and Ba, *Adam: A Method for Stochastic Optimization*.

<sup>106</sup>Dong C. Liu and Jorge Nocedal. "On the Limited Memory BFGS Method for Large Scale Optimization". In: *Math. Program.* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 0025-5610.

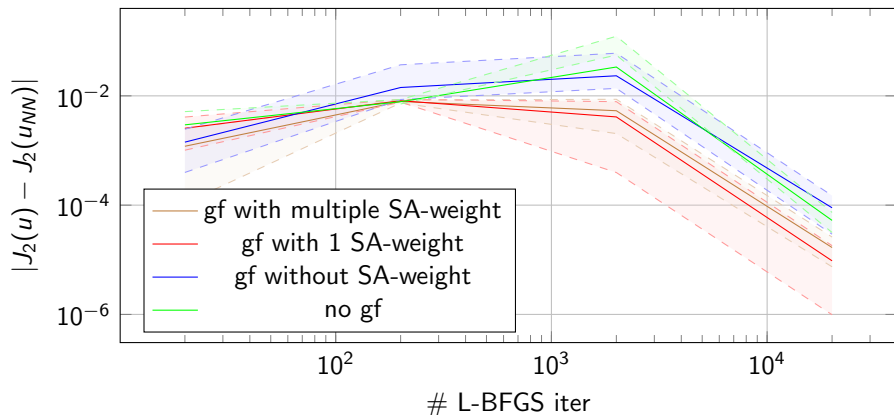


## Project 3: Goal-Oriented Self-Adaptive PINNs (XVI)



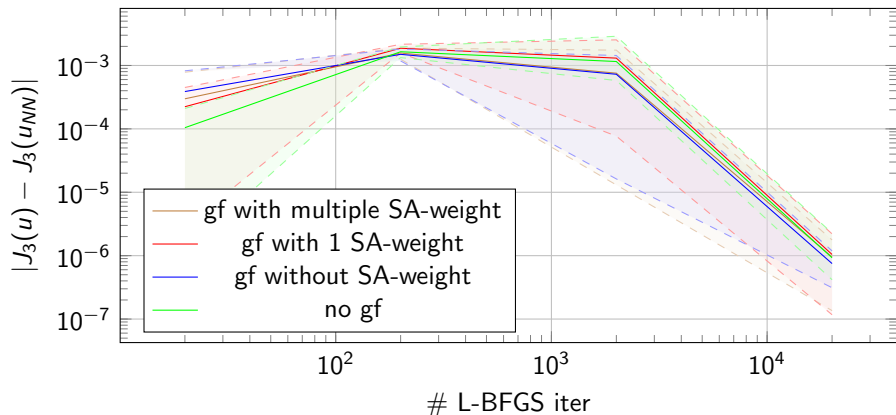
**Figure:** Plot of the goal functional error of the functional  $J_1$  with 4 different approaches. The x-axis describes the number of L-BFGS iterations and the y-axis describes the error measured in the goal functional.

## Project 3: Goal-Oriented Self-Adaptive PINNs (XVII)



**Figure:** Plot of the goal functional error of the functional  $J_2$  with 4 different approaches. The x-axis describes the number of L-BFGS iterations and the y-axis describes the error measured in the goal functional.

## Project 3: Goal-Oriented Self-Adaptive PINNs (XVIII)



**Figure:** Plot of the goal functional error of the functional  $J_3$  with 4 different approaches. The x-axis describes the number of L-BFGS iterations and the y-axis describes the error measured in the goal functional.

# Project 3: Goal-Oriented Self-Adaptive PINNs (XIX): Conclusions

## ① Advantages of this approach:

- The code for SAPINNs is open source and the extension the GOSAPINNs is simple
- Satisfying results (at least in case 2 and case 1)
- Adaptable to other PDEs and other goal functionals.
- As indicated above: greater flexibility in obtaining information on 'parts' of the solution
- Advantage of SAPINNs: improved training process; large errors are increased artificially to force the neural network to approximate the exact solution more precisely

# Project 3: Goal-Oriented Self-Adaptive PINNs (XX): Conclusions

## ③ Drawbacks:

- Higher cost (training needs more time): Around 15% more time when comparing case 1 and case 4. As mentioned before, it is a trade-off ('no free lunch')
- Convergence problems in the L-BFGS solver

## ④ Still to do:

- Implement a better approximation of the integrals (e.g. trapezoidal rule)
- Test on other, more complex PDEs and more complex goal functionals
- Until now, we require an analytical solution to handle the goal functional error. In practice, we do not have an analytical solution (in most cases), thus we have to find a way to approximate it

# Bibliography I

- Amann, H. and J. Escher. *Analysis I*. Birkhäuser, 2006. URL:  
<https://link.springer.com/book/10.1007/978-3-7643-7756-4>.
- *Analysis II*. Birkhäuser, 2006. URL:  
<https://link.springer.com/book/10.1007/3-7643-7402-0>.
- *Analysis III*. Birkhäuser, 2008. URL:  
<https://link.springer.com/book/10.1007/978-3-7643-8884-3>.
- Arridge, Simon et al. “Solving inverse problems using data-driven models”.  
In: *Acta Numerica* 28 (2019), pp. 1–174. DOI:  
[10.1017/S0962492919000059](https://doi.org/10.1017/S0962492919000059).
- Bellman, Richard. “A Markovian Decision Process”. In: *Indiana Univ. Math. J.* 6.4 (1957), pp. 679–684.
- Berner, Julius et al. *The Modern Mathematics of Deep Learning*. 2021.  
arXiv: 2105.04026 [cs.LG].
- Bishop, C. M. *Pattern recognition and machine learning*. Springer, 2006.
- Braun, M. *Differential equations and their applications*. Springer, 1993.

## Bibliography II

- Brezis, H. *Functional Analysis, Sobolev Spaces and Partial Differential Equations*. Springer, 2011. DOI:  
<https://doi.org/10.1007/978-0-387-70914-7>.
- Brunton, Steven L., Bernd R. Noack, and Petros Koumoutsakos. “Machine Learning for Fluid Mechanics”. In: *Annual Review of Fluid Mechanics* 52.1 (2020), pp. 477–508. DOI:  
10.1146/annurev-fluid-010719-060214. eprint:  
<https://doi.org/10.1146/annurev-fluid-010719-060214>. URL:  
<https://doi.org/10.1146/annurev-fluid-010719-060214>.
- Burkov, Andriy. *The hundred-page machine learning book*. Andriy Burkov, 2019.
- Chen, Ricky T. Q. et al. *Neural Ordinary Differential Equations*. 2019. arXiv: 1806.07366 [cs.LG].

## Bibliography III

- Czarnecki, Wojciech. *Lecture 2: Neural Networks Foundations*. University Lecture. 2020. URL: [https://storage.googleapis.com/deepmind-media/UCLxDeepMind\\_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf](https://storage.googleapis.com/deepmind-media/UCLxDeepMind_2020/L2%20-%20UCLxDeepMind%20DL2020.pdf).
- Deisenroth, Marc Peter, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020. ISBN: 9781108470049. URL: <https://books.google.de/books?id=pFjPDwAAQBAJ>.
- Dwass, Meyer. *Probability: Theory and applications*. W.A. Benjamin, Inc., New York, 1970.
- E, Weinan and Bing Yu. “The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems”. In: *Communications in Mathematics and Statistics* 6.1 (2018), pp. 1–12. ISSN: 2194-671X. DOI: 10.1007/s40304-018-0127-z. URL: <https://doi.org/10.1007/s40304-018-0127-z>.



## Bibliography IV

Fischer, G. *Lineare Algebra*. Springer, 2014. URL:

<https://link.springer.com/book/10.1007/978-3-658-03945-5>.

Georgii, H.-O. *Stochastik*. de Gruyter, 2009.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. *Deep Learning*.

<http://www.deeplearningbook.org>. MIT Press, 2016.

Goswami, Somdatta et al. "A physics-informed variational DeepONet for predicting crack path in quasi-brittle materials". In: *Computer Methods in Applied Mechanics and Engineering* 391 (2022), p. 114587. ISSN: 0045-7825. DOI: <https://doi.org/10.1016/j.cma.2022.114587>. URL: <https://www.sciencedirect.com/science/article/pii/S004578252200010X>.

## Bibliography V

- Goswami, Somdatta et al. “Transfer learning enhanced physics informed neural network for phase-field modeling of fracture”. In: *Theoretical and Applied Fracture Mechanics* 106 (2020), p. 102447. ISSN: 0167-8442. DOI: <https://doi.org/10.1016/j.tafmec.2019.102447>. URL: <https://www.sciencedirect.com/science/article/pii/S016784421930357X>.
- Guilhoto, Leonardo Ferreira. *An Overview Of Artificial Neural Networks for Mathematicians*. 2018. URL: <https://math.uchicago.edu/~may/REU2018/REUPapers/Guilhoto.pdf>.
- Haasdonk, Bernard. “Reduced basis methods for parametrized PDEs—a tutorial introduction for stationary and instationary problems”. In: *Model reduction and approximation: theory and algorithms* 15 (2017), p. 65.
- Higham, C. F. and D. J. Higham. “Deep Learning: An introduction for Applied Mathematicians”. In: *SIAM review* 61.4 (2019), pp. 860–891.

## Bibliography VI

- Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: [https://doi.org/10.1016/0893-6080\(89\)90020-8](https://doi.org/10.1016/0893-6080(89)90020-8). URL: <https://www.sciencedirect.com/science/article/pii/0893608089900208>.
- Jolliffe, I.T. *Principal Component Analysis*. Springer, 2002. DOI: <https://doi.org/10.1007/b98835>.
- Kharazmi, E., Z. Zhang, and G. E. Karniadakis. *Variational Physics-Informed Neural Networks For Solving Partial Differential Equations*. 2019. arXiv: 1912.00873 [cs.NE].
- Kilcher, Yannic. *Neural Ordinary Differential Equations*. Youtube. 2019. URL: <https://www.youtube.com/watch?v=jltgNGt8Lpg>.
- Kingma, Diederik P. and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

## Bibliography VII

- Knoke, Tobias and Thomas Wick. “Solving differential equations via artificial neural networks: Findings and failures in a model problem”. In: *Examples and Counterexamples 1* (2021), p. 100035. DOI: <https://doi.org/10.1016/j.exco.2021.100035>.
- Kutz, J. Nathan. “Deep learning in fluid dynamics”. In: *Journal of Fluid Mechanics* 814 (2017), pp. 1–4. DOI: [10.1017/jfm.2016.803](https://doi.org/10.1017/jfm.2016.803).
- Li, Zongyi et al. *Fourier Neural Operator for Parametric Partial Differential Equations*. 2021. arXiv: [2010.08895](https://arxiv.org/abs/2010.08895) [cs.LG].
- Liu, Dong C. and Jorge Nocedal. “On the Limited Memory BFGS Method for Large Scale Optimization”. In: *Math. Program.* 45.1–3 (Aug. 1989), pp. 503–528. ISSN: 0025-5610.
- Lu, Lu et al. *A comprehensive and fair comparison of two neural operators (with practical extensions) based on FAIR data*. 2021. arXiv: [2111.05512](https://arxiv.org/abs/2111.05512) [physics.comp-ph].

## Bibliography VIII

- Lu, Lu et al. “DeepXDE: A Deep Learning Library for Solving Differential Equations”. In: *SIAM Review* 63.1 (2021), pp. 208–228. DOI: [10.1137/19M1274067](https://doi.org/10.1137/19M1274067).
- Lu, Lu et al. “Learning nonlinear operators via DeepONet based on the universal approximation theorem of operators”. In: *Nature Machine Intelligence* 3.3 (2021), pp. 218–229. ISSN: 2522-5839. DOI: [10.1038/s42256-021-00302-5](https://doi.org/10.1038/s42256-021-00302-5). URL: <https://doi.org/10.1038/s42256-021-00302-5>.
- Mathieu, M., M. Henaff, and Y. LeCun. *Fast training of convolutional networks through FFTs*. Preprint. 2013.
- McClenny, Levi D. and Ulisses M. Braga-Neto. “Self-Adaptive Physics-Informed Neural Networks using a Soft Attention Mechanism”. In: *CoRR* abs/2009.04544 (2020). arXiv: 2009.04544. URL: <https://arxiv.org/abs/2009.04544>.
- Nielsen, Michael A. *Neural Networks and Deep Learning*. 2018. URL: <http://neuralnetworksanddeeplearning.com/>.

## Bibliography IX

- Nocedal, Jorge and Stephen J. Wright. *Numerical Optimization*. second. New York, NY, USA: Springer, 2006.
- Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].
- Quarteroni, A. and P. Gervasio. *A Primer on Mathematical Modelling*. Springer, 2020.
- Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.
- .“Induction of decision trees”. In: *Machine learning* 1 (1986), pp. 81–106. DOI: <https://doi.org/10.1007/BF00116251>.

# Bibliography X

- Raissi, M., P. Perdikaris, and G.E. Karniadakis. “Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations”. In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2018.10.045>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999118307125>.
- Richter, T. and T. Wick. *Einführung in die Numerische Mathematik: Begriffe, Konzepte und zahlreiche Anwendungsbeispiele*. Dec. 2017. ISBN: 978-3-662-54177-7. DOI: 10.1007/978-3-662-54178-4.
- Roth, Julian, Max Schröder, and Thomas Wick. “Neural network guided adjoint computations in dual weighted residual error estimation”. In: *SN Applied Sciences* 4 (2022). DOI: <https://doi.org/10.1007/s42452-022-04938-9>.

## Bibliography XI

- Samolik, L. and T. Wick. *Numerische Mathematik II (Numerik GDGL, Eigenwerte)*. Institute for Applied Mathematics, Leibniz Universität Hannover, Germany. 2019.
- Sharir, Or, Barak Peleg, and Yoav Shoham. “The Cost of Training NLP Models: A Concise Overview”. In: *CoRR* abs/2004.08900 (2020). arXiv: 2004.08900. URL: <https://arxiv.org/abs/2004.08900>.
- Sirignano, Justin and Konstantinos Spiliopoulos. “DGM: A deep learning algorithm for solving partial differential equations”. In: *Journal of Computational Physics* 375 (2018), pp. 1339–1364. ISSN: 0021-9991. DOI: <https://doi.org/10.1016/j.jcp.2018.08.029>. URL: <https://www.sciencedirect.com/science/article/pii/S0021999118305527>.
- Smith, Ralph C. *Uncertainty Quantification*. SIAM, 2014.
- Socher, Richard. *Lecture 8: Recurrent Neural Networks*. CS224d, Deep NLP. 2016.



## Bibliography XII

- Soleimany, Ava. *MIT 6.S191 Introduction to Deep Learning: Deep Sequence Modeling*. MITDeepLearning. 2021.
- Thickstun, John. *The Transformer Model in Equations*. Preprint. 2019.
- Vaswani, Ashish et al. "Attention is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.
- Weinzierl, Stefan. *Introduction to Monte Carlo methods*. 2000. arXiv: hep-ph/0006269 [hep-ph].
- Werner, Dirk. *Funktionalanalysis*. Springer, 2018.
- Wick, T. *Numerical methods for partial differential equations*. Hannover : Institutionelles Repositorium der Leibniz Universität Hannover, DOI: <https://doi.org/10.15488/11709>. Jan. 2022. DOI: <https://doi.org/10.15488/11709>.